
Machine Learning

Release 1.0

Jens Pönisch

10.06.2022

1	Literatur und Online-Quellen	1
2	Organisatorisches	3
3	Einführung und Installation der Werkzeuge	5
4	Kurzeinführung in Python	11
5	Pandas	23
6	NumPy	45
7	Supervised Learning	59
8	Beispieldatensätze	69
9	Vorverarbeitung der Daten	79
10	Matplotlib	89
11	Klassifikationsverfahren	105
12	Regression	125
13	Entscheidungsbäume	143
14	Ensemble-Methoden, Random Forests	155
15	Support Vector Machines	163
16	Suche optimaler Hyperparameter	179
17	Künstliche Neuronale Netze	193
18	Neuronale Netze mit Tensorflow	199
19	Regularisierung von Neuronalen Netzen	217
20	Convolutional Neuronal Networks	235
21	Recurrent Neural Networks	249
22	Abkürzungen	263
	Literaturverzeichnis	265

Literatur und Online-Quellen

1.1 Monographien

Eine allgemeine Einführung in die Vielfalt der Verfahren auf mittlerem mathematischen Niveau bietet [bis06], die Theorie des *Deep Learning* wird in [agg18] und [goo16] auf hohem Niveau dargestellt.

Ein gut geeigneter Praxiseinstieg in die klassischen Lernverfahren mit Python sowie *ANN* mit *Tensorflow* ist [ger19], an dem sich der Kurs orientiert. Dieses Buch ist als E-Book in der Bibliothek der TU-Chemnitz verfügbar. Die deutsche Übersetzung heißt *Praxiseinstieg mit Scikit-Learn und TensorFlow*.

[muel17] beschränkt sich auf die klassischen Machine-Learning-Verfahren, stellt sie aber ausführlicher dar. Die Nutzung der verwendeten Bibliotheken *NumPy*, *Pandas* und *Matplotlib* wird in [van18] ausführlich erläutert, [der20] beschreibt dagegen die Verwendung von *TensorFlow* in Python.

1.2 Online-Ressourcen

Zum Verständnis von *ANN* und speziell für die Bildklassifikation (*CNN*) ist das E-Book [nie15] empfehlenswert.

Die *Anaconda*-Installation kann von [anaconda] heruntergeladen werden. Die Auszeichnungssprache *Markdown* für die Textzellen ist in [markdown] beschrieben. Mit [colab] stellt *Google* eine freie Machine-Learning-Plattform für *Jupyter Notebooks* zur Verfügung.

Eine hilfreiche Online-Übungssammlung zur Hilfbibliothek bietet [py101].

Die Originaldokumentation [tfhome] von *Tensorflow* beschreibt nicht nur die Schnittstellen, sondern erläutert auch ausführlich, wie *ANN*s erstellt werden können.

1.3 Datensammlungen

- *Voraussetzungen*
- *Abblauf*
- *Leistungsnachweis*
- *Hinweise*

2.1 Voraussetzungen

Für die Teilnahme am Kurs sollte folgende Kenntnisse vorhanden sein:

- Grundlagen der Programmierung (z. B. aus WI-Praktikum 1).
- Grundkenntnisse der linearen Algebra (Vektoren, Matrizen) und der Statistik.

2.2 Ablauf

- Praktikum einmal wöchentlich.
- Selbständiges Arbeiten mit Script und angegebener Literatur.
- Unterstützung durch Screencasts.
- Etwa 10 Termine.
- ~~Jeweils etwa 45 Minuten Einführung in das Thema und 45 Minuten Übung.~~ → Konsultation im Webroom.
- Arbeit mit Jupyter Notebook auf Pool-PC oder eigenem Notebook.

2.3 Leistungsnachweis

- 5 Pflichtaufgaben zur selbständigen Bearbeitung.
- Korrektur mit aktueller *Anaconda-3*-Installation unter Linux.
- Bewertung jeder Aufgabe mit Punkten.
- Zulassung zum Abschlusstest bei
 - 75% der maximal erreichbaren Punktzahl (Master und Bachelor),
 - abgeschlossene Praktika *WI-Praktikum 1* und *WI-Praktikum 2* (nur für Bachelor Wirtschaftsinformatik).
- Abschlusstest: 60 Minuten am Computer ~~im AZW~~.
- Bewertung durch Zensur.

2.4 Hinweise

- Fragen zum Praktikum und zur Bewertung werden *ausschließlich* im *OPAL*-Forum und während des Praktikums beantwortet. Mails zum Praktikum bleiben unbearbeitet.
- Abgabe der Hausaufgaben erfolgt ausschließlich über *OPAL* als *ZIP*-Archiv:
 - verwendete Datendateien (jedesmal mit abgeben),
 - *Jupyter*-Notebooks.
- Verspätete Abgaben werden nicht akzeptiert.
- Gemeinschaftsarbeiten und Plagiate gelten als Betrugsversuch und werden mit *ungenügend* bewertet.

Einführung und Installation der Werkzeuge

- *Defintion und Aufgaben des Machine Learning*
- *Werkzeuge*
- *Anaconda und Miniconda*
- *Virtuelle Umgebung*
- *Entwicklungsumgebungen*
- *Visual Studio Code*
- *Google Colaboratory*
- *Erste Schritte mit Jupyter Notebook*
- *Aufgaben*

3.1 Defintion und Aufgaben des Machine Learning

ML ist ein Teilgebiet der *Künstlichen Intelligenz* und des *Data Science*. Darunter versteht man die automatische Generierung von *verallgemeinernden* Modellen aus verfügbaren Trainingsdaten.

Definition Machine Learning

A computer program is said to learn from experience *E* with respect to some task *T* and some performance measure *P*, if its performance on *T*, as measured by *P*, improves with experience *E*.—*Tom Mitchell, 1997.*

ML ist eng verwandt mit *Knowledge Discovery in Databases* und *Data Mining*, wobei letztere mehr das Erkennen von Mustern betonen, während ersteres die Modellbildung betont. Die verwendeten Algorithmen sind häufig identisch.

Derzeit werden mit *ML* folgende drei Teilbereiche erfolgreich bearbeitet:

Supervised Learning: Für die Trainingsdaten liegen «Lösungen» vor, anhand derer die Modellqualität bewertet werden kann: Klassen (Katze–Hund–Affe, Spam/Ham) oder kontinuierliche Größen (Wert eines Einkaufs abhängig von Kundendaten, Wert einer Immobilie in Abhängigkeit der Gebäudegröße, Wohngegend, ...).

Unsupervised Learning: Für die Trainingsdaten liegt keine Bewertung vor, das Modell soll automatisch Strukturen in den Daten erkennen: Clusterung (automatische Klasseneinteilung, Erkennen von Ähnlichkeiten), Dimensionsreduktion (Aussondern unwichtiger Merkmale, Darstellung der Daten in der Ebene, Erzeugen von *Embeddings*).

Reinforcement Learning: Ein *Agent* erkundet seine Umwelt und führt dazu wahlweise verschiedene *Aktionen* aus, für die er früher oder später *belohnt* oder *beträcht* wird. Anhand dieser Bewertungen wird versucht, eine optimale Strategie zu finden (automatische Geländeerkundungen, Entwickeln von Spielstrategien).

Zwischen diesen Bereichen existieren Mischformen, ebenso können in Zukunft neue Bereiche erschlossen werden.

Eine spezielle Bedeutung hat der Begriff *Deep Learning*. Er bezieht sich nicht auf eine bestimmte Klasse von Lernproblemen, sondern auf die Technologie der Umsetzung: Hier werden *DNN* verwendet, das sind neuronale Netze mit einer großen Zahl verborgener Schichten (*hidden layer*). Aufgrund der enormen Rechenleistung moderner Graphikkarten (die speziell für solche Zwecke weiterentwickelt wurden) ist diese Technik in den letzten Jahren sehr erfolgreich geworden.

3.2 Werkzeuge

Um Algorithmen des *Data Science* zu formulieren, kommen im wesentlichen folgende Programmiersprachen und Softwarepakete zum Einsatz:

R Der «Klassiker» für statistische Untersuchungen mit vielen Erweiterungspaketen für die Datenanalyse.

Julia Höhere Programmiersprache für wissenschaftliches Rechnen, an *MATLAB* und *R* orientiert, aber mit deutlich höherer Performance.

Python Verschiedene in C/C++ implementierten Bibliotheken erlauben ein recht komfortables Bearbeiten von vielfältigen Aufgaben des *>Data Science*. Derzeit die bevorzugte und am besten dokumentierte Methode.

C++ etc. Während die vorher genannten Sprachen für die interaktive Datenanalyse und das Prototyping gut geeignet sind, werden *ML*-Modelle für den Produktiveinsatz häufig in einer maschinennäheren Sprache umgesetzt.

Wir verwenden hier als Werkzeug *Python* (Version 3.x) mit den Bibliotheken:

NumPy: Bereitstellung von Datentypen für große Vektoren, Matrizen und mehrdimensionale Felder (*Tensoren*) und dazugehörige Operationen.

Pandas: Bereitstellung von Datentypen für Zeitreihen und Tabellen (*Data Frame*). Im Gegensatz zu *NumPy* kann *Pandas* mit Datenspalten verschiedenen Typs gut umgehen und stellt zahlreiche Transformations- und Auswertungsfunktionen zur Verfügung.

Matplotlib: Graphikbibliothek zur komfortablen Darstellung von *NumPy*- und *Pandas*-Datenstrukturen.

SciPy: Bibliothek zum wissenschaftlichen Rechnen (lineare Algebra, Optimierung), wird von *scikit-learn* als Grundlage benutzt.

scikit-learn: Bibliothek für *ML*. Implementiert praktisch alle klassischen Verfahren des *ML* und *Data Mining*.

TensorFlow: Abstraktionsbibliothek zum Rechnen mit Vektoren, Matrizen, Vektoren. Erlaubt die Verteilung auf Multiprozessorsysteme, Graphikkarten, Cluster. Basisbibliothek für *ANN*.

Keras: Abstraktionsbibliothek für *ANN* auf der Basis von *TensorFlow*, *PyTorch* und *Theano*. Eine speziell angepasste Version ist in *TensorFlow* eingebaut und seit der Version *TensorFlow 2.0* die empfohlene Anwenderschnittstelle.

In Standard-Python-Installationen sind diese Pakete zunächst nicht vorhanden und müssen nachinstalliert werden. Da es teilweise recht komplexe Abhängigkeiten gibt, bietet sich die Installation einer Python-Umgebung speziell für diese Zwecke an. Wir verwenden die freie Umgebung *Anaconda 3* [anaconda].

3.3 Anaconda und Miniconda

Während *Anaconda 3* [anaconda] automatisch einen großen Paketumfang installiert, erhält man mit *Miniconda* [miniconda] nur eine Minimalinstallation, in der die notwendigen Pakete gezielt nachinstalliert werden können. Beide Umgebungen verwenden die gleiche Basisinstallation und die gleichen Verwaltungswerkzeuge.

Während die volle Anaconda-Umgebung praktisch alle notwendigen Pakete für das *Machine Learning* und viele weitere Anwendungen bereits enthält, müssen diese in *Miniconda* nachinstalliert werden. Das klingt zunächst wie ein Nachteil, ist aber an vielen Stellen wieder vorteilhaft, da die Auswahl bestimmter Paketversionen deutlich einfacher wird, da hier viel weniger Abhängigkeiten zu anderen Paketen, die vermutlich überhaupt nicht benötigt werden, nicht berücksichtigt werden müssen. Im konkreten Fall war ein notwendiges Upgrade der Python-Version bei in der *Miniconda*-Umgebung problemlos möglich, während sie in *Anaconda* scheiterte.

Die Installation erfolgt unter Linux durch Ausführung des Installationsscripts:

```
sh Anaconda3-###.sh
```

bzw.

```
sh Miniconda3-###.sh
```

Nach Akzeptieren der Lizenzvereinbarung und Auswahl des Installationsverzeichnisses (Schreibrechte müssen vorhanden sein) packt sich das Archiv an der angegebenen Stelle aus. Wir können anschließend festlegen, ob die *Anaconda*-Umgebung automatisch aktiviert werden soll (Empfehlung: nein).

Wir gehen im weiteren davon aus, dass die Installation im Home-Verzeichnis des Nutzers im Ordner `anaconda3` erfolgt ist (Vorgabe, bei *Miniconda* durch `miniconda3` ersetzen).

3.4 Virtuelle Umgebung

Um Konflikte mit anderen Python-Paketen zu vermeiden, erstellt *Anaconda* eine eigene virtuelle Umgebung, die zur Verwendung aktiviert werden muss (beachten Sie den Punkt vor dem Aufruf):

```
. ~/anaconda3/bin/activate
# oder
source ~/anaconda3/bin/activate
```

Wir sehen die Aktivierung am veränderten Promptzeichen. Das Deaktivieren dieser geänderten Python-Umgebung geschieht mit

```
conda deactivate
```

Tipp: Im *AZW* ist *Anaconda 3* unter dem Installationspfad `/opt/wiwi/anaconda3` vorhanden. Es ist keine eigene Installation erforderlich. Die Aktivierung dieser Umgebung erfolgt mit dem Aufruf:

```
. /opt/wiwi/anaconda3/bin/activate
```

Direkt nach der Installation und der Aktivierung der *Anaconda*-Umgebung bietet es sich an, ein Update auf den aktuellen Stand durchzuführen:

```
conda update conda --all
conda update anaconda
```

Die Steuerung der virtuellen Umgebung erfolgt mit dem Kommando **conda**. Wichtige Befehle sind:

conda list Anzeige aller installierten Pakete

conda list muster Anzeige aller installierten Pakete, die dieses Muster enthalten.

conda search muster Suche nach installierbaren Paketen, die dieses Muster enthalten.

conda install paketname Nachinstallation des gewünschten Pakets.

conda update paketname Update des gewünschten Pakets auf die aktuelle Version.

Hinweis: Wir sollten an dieser Stelle überprüfen, ob alle Pakete für den ersten Teil des Praktikums vorhanden sind:

```
conda list numpy
conda list pandas
conda list matplotlib
conda list scikit-learn
conda list jupyter
conda list nbconvert
conda list pillow
```

Das Paket *TensorFlow* benötigen wir erst im zweiten Teil.

Sollten wir ein Python-Paket benötigen, das nicht über die Anaconda-Repositories angeboten wird, können wir es mit dem Standard-Installationswerkzeug **pip** in der virtuellen Umgebung installieren:

pip install paket

Um eine Liste von Paketen zu installieren, können diese zeilenweise in eine Datei geschrieben und mit einem Aufruf gemeinsam installiert werden.

```
conda install --file paketliste
```

Beispiel: Installation der für das Praktikum benötigten Pakete in *Miniconda* mit der Paketliste `ml-packages.txt`. Die Python-Version wird auf 3.9 aktualisiert.

Hinweis: Es bietet sich an, das Paket `import_ipynb` zu installieren, das uns die Modularisierung von Notebooks erlaubt. Dieses befindet sich nicht in den Anaconda-Repositories und muss manuell mit **pip** nachinstalliert werden.

```
pip install import_ipynb
```

3.5 Entwicklungsumgebungen

Das *Anaconda*-Paket enthält u. a. die interaktive Stand-alone-Entwicklungsumgebung *spyder* und mit *Jupyter Notebook* eine browserbasierte Entwicklungsumgebung.

Spyder kann nach dem Aktivieren der virtuellen *Anaconda*-Umgebung ohne Eingabe des Pfadnamens gestartet werden:

```
spyder &
```

Zum interaktiven Testen und dessen Dokumentation wird gern die browsergestützte Umgebung *Jupyter Notebook* (*Jupyter* = *Julia* + *Python* + *R*) verwendet. Diese wird wieder durch vollständige Pfadangabe direkt oder nach Wechsel in die virtuelle Umgebung gestartet:

```
cd arbeitsverzeichnis
# Virtuelle Umgebung aktivieren, falls noch nicht geschehen
. ~anaconda3/bin/activate
jupyter notebook
```

Im Konsolenfenster startet der Server, es sollte sich ein neues Browserfenster öffnen, das entweder das ausgewählte Notebook oder das Arbeitsverzeichnis anzeigt, aus dem ein Notebook geöffnet bzw. ein neues Notebook erzeugt werden kann.

Um gezielt mit einem bestimmten Notebook zu arbeiten, kann dieses auch beim Aufruf spezifiziert werden:

```
jupyter notebook notebookDatei.ipynb
```

Tipp: Der Start des Jupyter-Notebook-Servers sollte *nicht* im Hintergrund (durch das &-Zeichen) erfolgen, da dann das Beenden mit `StrgC` nicht mehr funktioniert.

Im Praktikum werden wir vorrangig mit *Jupyter Notebook* arbeiten, das sich gut zum Experimentieren eignet und eine gute Kommentierung erlaubt. Für den Produktiveinsatz können die Notebooks problemlos in normale *Python*-Scripts umgewandelt werden.

Warnung: Als Lösungen der Hausaufgaben werden nur Jupyter-Notebooks, keine einfachen Python-Scripts akzeptiert.

3.6 Visual Studio Code

Alternativ zum Web-Browser ist die Arbeit mit *Jupyter Notebooks* auch im Editor *Visual Studio Code* möglich. Wir starten dazu diesen Editor (Wechsel in die *Conda*-Umgebung ist nicht erforderlich) und öffnen das gewünschte Verzeichnis mit den Notebooks. Beim ersten Aufruf wird ein *Workspace* erstellt, wir müssen bestätigen, dass wir den Autoren (also uns selbst) vertrauen. Falls noch nicht geschehen, installieren wir die Erweiterungen *Python* und *Jupyter*. Anschließend können wir ein neues Notebook erstellen (Menü: *File*→*New File ...*) und die gewünschte Sprache *Jupyter Notebook* wählen.

Die Bedienung ist vergleichbar mit dem Web-Frontend, allerdings funktioniert derzeit die kontextsensitive Hilfe noch nicht.

3.7 Google Colaboratory

Alternativ zur eigenen Installation kann die Entwicklung der Notebooks auch im *Google Colaboratory* ([[colab](#)]) erfolgen. Zur Nutzung ist ein Google-Account erforderlich.

Eigene Notebooks können hochgeladen, in *Colab* erstellte Notebooks heruntergeladen und lokal weiterbearbeitet werden.

Der Einsatz ist insbesondere bei Modellen für *ANN* sinnvoll, da *Colab* für das Training Hardwarebeschleuniger zur Verfügung stellt.

Warnung: Die Bedienung von *Colab* weicht von der hier beschriebenen lokalen *Jupyter-Notebook*-Installation etwas ab. Die Administration des Python-Kernels erfolgt anders, die Menü sind anders strukturiert, es werden teilweise andere Tastenkürzel benutzt.

Die freie *Colab*-Version darf nur für Ausbildungszwecke benutzt werden. Die erstellten Daten liegen auf Google-Servern und stehen damit nicht unter dem Schutz der *DSGVO* der EU.

3.8 Erste Schritte mit Jupyter Notebook

Um die Arbeitsweise mit *Jupyter Notebook* kennenzulernen, wollen wir ein erstes *Hello-World-Notebook* erzeugen. Wir aktivieren die virtuelle Umgebung, erstellen ein Arbeitsverzeichnis, wechseln in dieses und starten den Notebook-Server.

```
cd
. anaconda3/bin/activate
mkdir wip3
cd wip3
jupyter notebook
```

Es sollte sich nun ein Browserfenster oder -tabular mit einer Dateiliste des Arbeitsverzeichnisses öffnen. Wir wählen *New ▶ Python 3* aus und befinden uns in der passenden Entwicklungsumgebung.

Mit `Enter` oder Mausklick gehen wir in den Bearbeitungsmodus der ausgewählten Zelle und können nun Python-Befehle eingeben.

```
42 + 23
```

Mittels `StrgEnter` wird der Editiermodus beendet und die Zelle ausgeführt. Wir sehen darunter sofort das Ergebnis.

Zum Beschreiben unserer Aktion fügen wir mit `A` bzw. dem Menüpunkt *Insert ▶ Insert Cell above** eine neue Zelle vor der aktiven Zelle ein. Diese soll Dokumentation enthalten, wir wechseln deshalb mit `M` bzw. dem Typeauswahlfeld in den *Markdown*-Modus, ehe wir diese Zelle bearbeiten. Wir geben eine Überschrift und einen kurzen Absatz ein und beenden das Editieren wieder mit `StrgEnter`.

```
# Mein erstes Notebook

Das ist eine einfache Rechnung
```

Um unsere Notebooks leicht wiederzufinden, sollten wir den Name über das *File*-Menü ändern. Mit `StrgS` wird der aktuelle Stand gesichert (erfolgt auch alle paar Sekunden automatisch).

Ist unsere Arbeit beendet, sollten wir das Notebook über das *File* mittels *Close and Halt* schließen. Anschließend können wir im Konsolenfenster den Notebook-Server mit `StrgC` beenden.

3.9 Aufgaben

Aufgabe 3.1 Machen Sie sich mit den Befehlen der Jupyter-Notebook-*IDE* vertraut.

Aufgabe 3.2 Machen Sie sich mit den grundlegenden Möglichkeiten der *Markdown*-Auszeichnungssprache vertraut. Erstellen Sie

- Überschriften verschiedener Größen,
- Absätze,
- Aufzählungen mit und ohne Numerierung,
- Texthervorhebungen (fett, kursiv, Schreibschrift),
- einfache Formeln im Fließtext oder als eigenen Block.

Kurzeinführung in Python

- *Überblick*
- *Grundlegender Aufbau*
- *Verzweigungen*
- *Schleifen*
- *Listen und Dictionaries*
- *Module*
- *Eigene Funktionen und Klassen*
- *Aufgaben*

4.1 Überblick

Da die meisten *ML*-Bibliotheken eine Schnittstelle zu *Python* bereitstellen, bietet es sich an, diese Sprache als *Kleber* zu verwenden, um die einzelnen Komponenten zu verbinden.

Wir werden im Praktikum Python in der Version 3 mit der webbasierten *IDE Jupyter Notebook* verwenden. Diese bietet sich auch an, um die Sprache zu üben und verstehen zu lernen.

Hinweis: Der Name *Python* leitet sich nicht von der Schlange, sondern der britischen Komikertruppe *Monty Python* ab, die der Erfinder der Sprache, *Guido van Rossum* sehr verehrte.

Hinweis: Eine ausführlichere Einführung in Python bietet das Script des WI-Praktikum 1 (ab Wintersemester 2020/21).

4.2 Grundlegender Aufbau

Obwohl sich die Syntax von *Python* stark an *C* anlehnt, gibt es doch einige deutliche Unterschiede. Der vielleicht auffälligste ist die Erzwingung korrekter Einrückungen für *Blöcke*. Alle Anweisungen eines Blocks haben die gleiche Einrücktiefe, die auch für alle Blöcke gleicher Tiefe einheitlich sein soll (2, 4 oder 8 Leerzeichen).

```
blocksteuerung :
    blockanweisung1
    blockanweisung2
    blocksteuerung :
        anweisungInnererblock1
        anweisungInnererblock2
    blockanweisung3
anweisungNachBlock
```

Hinweis: Wenn Sie Python-Programme mit einer anderen *IDE* als *Jupyter Notebook* erstellen, schalten Sie die Einrückung mit Tabulatoren ab und verwenden dafür nur Leerzeichen.

In Python ist jedes Datum ein anonymes Objekt, Variablen sind stets Referenzen auf diese Objekte. Aus diesem Grunde haben Variablen selbst keinen Typ, sondern der Typ ist mit dem referenzierten Objekt verbunden. So können Variablen zu verschiedenen Zeiten auf Objekte verschiedenen Typs verweisen.

```
a = 3.14
type(a)
# → <type 'float'>
a = 42
type(a)
# → <type 'int'>
a = 'Ätsch!'
type(a)
# → <type 'str'>
```

Python kennt u. a. folgende eingebauten Datentypen:

int Ganze Zahlen mit beliebiger Genauigkeit (soweit der Speicher reicht).

float Gleitkommazahlen mit doppelter Genauigkeit (implementationsabhängig).

bool Wahrheitswerte `True` und `False`.

str Unicode-Strings. Bei Ein- und Ausgabe müssen evt. Konvertierungen in die systemspezifische Codierung (z. B. *UTF-8*) erfolgen.

Die Bibliothek *NumPy* stellt uns weitere numerische Datentypen zur Verfügung, die maschinennäher sind und damit schneller verarbeitet werden können.

Um Strings in Zahlen zu konvertieren, verwenden wir die Funktionen `int()` und `float()`.

```
intObjekt = int(stringwert[, basis])
floatObjekt = float(stringwert)
```

Umgekehrt kann ein numerisches Objekt einfach in einen String umgewandelt werden:

```
stringObjekt = str(numerischerAusdruck)
```

Sollen bestimmte Formatierungen vorgenommen werden, bietet sich die Verwendung von Musterstrings zusammen mit der `format()`-Methode an:

```
stringObjekt = musterstring.format(parameterliste)
```

Für jeden Wert der Parameterliste enthält der Musterstring einen Ausdruck der Form

`{[formatspezifikation]}`

Der restliche Inhalt des Musters wird einfach in das Ergebnis übernommen. Die Formatspezifikation erlaubt die Angabe von Ausgabelänge, Zahl der Nachkommastellen, Ausrichtung, Benennung der Parameter. Wir benötigen hier nur die eigentliche Formatierung, die hinter einem Doppelpunkt angegeben wird und in etwa den Mustern von `printf()` in C entspricht.

```
PI = 3.14159265358979323846264
s = "{} {:.2f} {:.10.5f}".format(PI, PI, PI)

ANSW = 42
s = "{} {: -10d} {:10d} {:010d}".format(ANSW, ANSW, ANSW, ANSW)
```

Kommentare werden durch eine Raute `#` eingeleitet und wirken bis zum Zeilenende. Wird ein *Jupyter Notebook* als Python-Code exportiert, werden alle Textblöcke in Kommentare umgewandelt.

Die Ausgabe auf die Konsole erfolgt mit der `print()`-Funktion, die eine beliebige Anzahl von Parametern akzeptiert und nach Ausgabe aller Werte einen Zeilenumbruch ausgibt. Zum Schreiben in Datenströme existieren eine Vielzahl weiterer Funktionen und Methoden.

4.3 Verzweigungen

Verzweigungen werden wie in C mit der `if`-Anweisung erzeugt:

```
if bedingung:
    DannBlock
[
    elif bedingung2:
        Dann2Block
    ...
]
[
    else:
        SonstBlock
]
```

Die `elif`-Zweige (else-if) ermöglichen verschachtelte `if`-Anweisungen: Sie werden nur ausgeführt, wenn die vorigen Bedingungen nicht erfüllt wurden. Damit können Mehrfachverzweigungen realisiert werden. Eine `switch`-Anweisung kennt *Python* nicht.

Ein bedingter Ausdruck (wie mit `?:` in C) ist mit folgendem Konstrukt möglich:

```
variable = dannWert if bedingung else sonstWert
```

4.4 Schleifen

Die `while`-Schleife verhält sich in Python wie in C:

```
while bedingung:
    Block
[
else:
    FinalBlock
]
```

Der `FinalBlock` wird abgearbeitet, wenn die Schleife komplett durchlaufen ist. Wird sie mit `break` oder `return` (in Funktionen/Methoden) verlassen, wird er übersprungen.

Dagegen ist die `for`-Schleife ein Iterator, der alle Elemente einer Menge durchläuft.

```
for variable in collection:
    Block
[
else:
    FinalBlock
]
```

Um eine Zählschleife zu realisieren, können wir mit der Funktion `range()` eine Zahlenfolge erzeugen:

```
folge = range([startwert, ]endwert[, schrittweite])
```

Der Endwert wird nicht in die Folge aufgenommen.

Beispiel: Summe der Zahlen von 11 bis 20:

```
accu = 0
for i in range(11, 21): # Obere Grenze wird nicht erreicht
    accu += i
print(accu)
```

4.5 Listen und Dictionaries

Statt Feldern wie in C stellt Python ein Listenobjekt zur Verfügung. Es lässt sich ebenso wie ein Feld indizieren, kann aber dynamisch vergrößert und verkleinert werden. Der Nachteil und die höhere Laufzeit bei vielen Operationen.

Ein leeres Feld wird wie folgt erzeugt:

```
liste1 = []
# oder
liste2 = list()
```

Das Anhängen von Elementen geschieht mit der Methode `append()`.

```
liste.append(wert)
```

Der Indexzugriff zum Lesen bzw. Verändern von Werten erfolgt wie in C.

```
liste[index] = wert
eintrag = liste[index]
```

Zum Erzeugen einer initialisierten Liste kann im einfachsten Fall der Wiederholungsoperator `*` verwendet werden:

```
liste = [initialwert] * elementzahl
```

Ist der Initialwert kein einfacher, unveränderlicher Typ (wie Zahlen oder Strings), sondern wiederum eine Liste, führt obige Konstruktion zu unerwartetem Verhalten, da unsere Liste nur Referenzen auf immer das gleiche Objekt enthält und eine Änderung eines einzelnen Elementes alle Listenelemente ändert. Wir müssen dann die Liste wie folgt anlegen:

```
liste = [ausdruck for _ in range(elementzahl)]
```

Dieses Konstrukt kann auch für die Initialisierung der Liste mit unterschiedlichen Werten genutzt werden, indem die Zählvariable einen Namen erhält und im Initialisierungsausdruck benutzt wird.

Beispiele:

```
nullen = [0]*10
geradeZahlen = [x for x in range(20) if x % 2 == 0]
```

Die Elementauswahl erfolgt über die Angabe des Listenindex:

```
element = liste[index]
```

Die Zählung beginnt mit 0. Es sind auch negative Indices möglich, die die Elemente vom Listenende her zählen. -1 bezieht sich auf das letzte, -2 auf das vorletzte Listenelement.

Neben Einzelwerten können aus Listen auch *Slices* (Abschnitte) ausgewählt werden:

```
slice = liste[index0:index1]
```

Es entsteht eine Teilliste, die die Elemente ab dem `index0` bis zum letzten Element vor `index1` enthält. Wird `index0` weggelassen, bedeutet das den Index 0, wird `index1` weggelassen, dann wird bis zum letzten Element ausgewählt. Beispiele:

```
x = ['a', 'b', 'c', 'd', 'e', 'f']
x[3]      # → 'd'
x[1:4]    # → ['b', 'c', 'd']
x[-3:-1]  # → ['d', 'e']
x[-2:]    # → ['e', 'f']
x[:2]     # → ['a', 'b']
x[:]      # → ['a', 'b', 'c', 'd', 'e', 'f'] (Kopie)
```

Hinweis: Dieser Mechanismus funktioniert auf gleiche bzw. ähnliche Art und Weise auch für *NumPy*-Array und *Pandas* Data Frames.

Häufig möchten wir Daten aber nicht nur über einen Index, sondern über einen eindeutigen Namen ansprechen. Für diese Aufgabe stellt Python den Datentyp `dict` zur Verfügung. Die Ablage erfolgt in Form von Schlüssel-Wert-Paaren, wobei der Schlüssel ein eindeutiges und unveränderliches Objekt sein muss, z. B. ein String, ein Datum oder auch eine Zahl.

```
d = dict()
# oder
d = {key1 = value1, ...}
# Wert hinzufügen
d[key2] = value2
# Wert abfragen
value = d[key]
```

Ein komplettes Auslesen eines Dictionaries ist ebenfalls mit einer Schleife möglich:

```
for key in d:
    # Verarbeite key, d[key]
    ...
```

Beispiel: Zuordnung von Zahlen zu Farbnamen, Auflisten aller Zuordnungen.

```
colors = {
    'red': 0,
    'green': 1,
    'blue': 2,
}
print('blue hat den Code {}'.format(colors['blue']))
print('Alle Zuordnungen:')
for k in colors:
    print('{}: {}'.format(k, colors[k]))
```

4.6 Module

Python-Funktionen und Klassen sind in *Modulen* hierarchisch organisiert. Für die Verwendung aller nicht automatisch geladenen Objekte muss der Modul bzw. die gewünschte Funktion bzw. allgemeiner das Objekt importiert werden.

import modul [as alias]

Bei Untermodulen werden die Pfadbestandteile durch Punkte verkettet. Beispiel:

```
import matplotlib.pyplot
```

Wollen wir statt normaler Python-Scripts ein Notebook importieren, müssen wir vorher das Modul `import_ipynb` importieren (dieses Modul muss manuell nachinstalliert werden).

```
import import_ipynb
import myNotebook
```

Der Notebook-Name muss so gewählt werden, dass er einen gültigen Bezeichner darstellt.

Ein Objekt aus einem Modul wird durch Voranstellen des Modulnamens angesprochen:

```
import math
x = math.sin(math.PI)
```

Für den Import von Untermodulen oder einzelnen Objekten ist ein direkter Import möglich, womit bei der Nutzung der Modulprefix weggelassen bzw. verkürzt werden kann:

```
from modul import submodule
# oder
from modul import objekt
```

Übliche Importe sind für uns (in den folgenden Kapiteln):

```
import numpy                # Matrixoperationen
import pandas as pd         # Pandas DataFrame
from matplotlib import pyplot as plt
    # Untermodul mit Alias, Verwendung z. B. plt.show()
from sklearn.model_selection import train_test_split
    # Einzelfunktion wird importiert, Nutzung ohne Prefix
```

4.7 Eigene Funktionen und Klassen

Eigene Funktionen werden mit dem Schlüsselwort `def` definiert:

```
def funktion (parameterliste) :
[
    """
    Dokumentation
    """

    ...

[
    return [ausdruck]
]
```

Jeder Parameter wird im Funktionskopf als symbolischer Name aufgeführt, unter dem er innerhalb der Funktion ansprechbar ist. Von rechts nach links können den Parametern Vorgabewerte zugeordnet werden, dann können diese beim Funktionsaufruf weggelassen werden.

Funktionen können mehr als einen Wert zurückgeben. Beispiel:

```
import math

def kreisdaten(r=1.0):
    """
    Berechnet Umfang und Fläche gleichzeitig
    """
    u = 2 * r * math.PI
    a = r * r * math.PI
    return u, a

# Verwendung:
u1, a1 = kreisdaten()
u2, a2 = kreisdaten(2.0)
```

Klassen dienen zum Kapseln zusammengehöriger Daten. Sie werden wie folgt deklariert:

```
class Klasse:
    def __init__(self, komponente1 [, komponente2 ...]):
        self.__komponente1 = komponente1
        ...

    def getKomponente1(self):
        return self.__komponente1

    ...

    def __repr__(self):
        'Rückgabe des Objekts als String'
        return stringAusdruck
```

Das Voranstellen von zwei Unterstrichen macht die Komponente privat, sie kann nicht mehr direkt von außerhalb des Objekts angesprochen werden – statt dessen sind *Getter*-Methoden zu verwenden.

Neben Komponenten und Gettern kann die Klasse weitere Methoden definieren, um Eigenschaften und Verhalten der Klasse zu beschreiben.

Ein Objekt bzw. eine Instanz dieser Klasse wird erzeugt mit

```
objekt = Klasse(komponente1 ...)
```

Auf die Komponenten greifen wir mittels

```
objekt.getKomponente1()
```

zu.

Beispiel: Bruchklasse. Ein angelegter Bruch kann nicht mehr verändert werden, Rechenoperationen (hier nur Addition) erzeugen neue Objekte.

Quellcode 4.1: Fraction.py: Klasse Bruch

```
#!/usr/bin/env python
# coding: utf-8

# # Bruchklasse

# Import der Funktion gcd() aus dem Modul math (zum Kürzen)

from math import gcd

# Klassendefinition (kann nicht auf mehrere Zellen aufgeteilt werden)

class Fraction:
    """
    Klasse für Bruch mit Addition.
    Dummy-Definition, um Typ bekannt zu machen
    """
    def __init__(self, nom: int, denom=1):
        """
        Konstruktor.
        Aufruf mit 2 Parametern. Fehlt der 2. wird er auf 1 gesetzt
        """
        if denom == 0:
            nom = None
            denom = None
        self.__nom = nom
        self.__denom = denom
        # normalisieren und kürzen
        if self.isValid():
            self.normalize()
            self.reduce()

    def isValid(self) -> bool:
        """
        prüfen, ob Bruch zulässig ist.
        """
        return self.__nom is not None and self.__denom is not None and self.__denom != 0

    def normalize(self):
        """
        Sichern, dass Nenner > 0
        """
        if self.__denom < 0:
            self.__nom *= -1
            self.__denom *= -1

    def reduce(self):
        """
        Kürzen
        """
        if not self.isValid():
            return
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    d = gcd(self.__nom, self.__denom)
    self.__nom //= d
    self.__denom //= d

    # Getter
    def nom(self) -> int:
        return self.__nom

    def denom(self) -> int:
        return self.__denom

    def __repr__(self):
        """
        Ausgabe in print()
        """
        if self.isValid():
            return 'Fraction({}, {})'.format(self.__nom, self.__denom)
        else:
            return 'None'

    def add(self, other):
        """
        Addition von Brüchen
        :return: neues Fraction-Objekt
        """
        if self.isValid() and other.isValid():
            commonDenom = self.__denom * other.denom()
            nom = self.__nom * other.denom() + other.nom() * self.__denom
            return Fraction(nom, commonDenom)
        else:
            return Fraction(None, None)

# Test, nur aufrufen, wenn Notebook direkt und nicht als Modul gestartet wird
if __name__ == '__main__':
    f1 = Fraction(3)
    f2 = Fraction(4, -6)
    f3 = Fraction(5, 0)
    print(f1, f2, f3)
    # Addition
    print("{} + {} = {}".format(f1, f2, f1.add(f2)))

```

Notebook: Fraction.ipynb.

4.8 Aufgaben

Aufgabe 4.1 Schreiben Sie ein Notebook, das in einem Block folgende Aufgabe löst: Gegeben ist die Folge der Zahlen von 1 bis 10. Wir wollen daraus zwei Zahlen so entnehmen, dass Ihr Produkt gleich der Summe der verbliebenen Zahlen ist.

Ersetzen Sie nun 10 durch N und formulieren Sie das Teilproblem als Funktion. Schreiben Sie nun einen Python-Block, der für alle Zahlen N von 2 bis 100 untersucht, ob zwei solche Zahlen der Folge von 1 bis N entnommen werden kann, und geben Sie im Erfolgsfall diese aus.

Aufgabe 4.2 Erweitern Sie die Klasse `Fraction` um weitere Operationen. Schreiben Sie eine Methode `__str__()`, die nur den Parameter `self` übernimmt und einen String der Form $2/3=0.6666$ zurückgibt. Was

passiert, wenn Sie nun mit `print()`

- einen einzelnen Buch
- eine Python-Liste von Brüchen

ausgeben?

Aufgabe 4.3 Grundlagen Python, Statistik Jupyter Notebook, Hausaufgabe (6 Punkte)

Wählen Sie aus einer der Machine-Learning-Datensammlungen [[uci](#)], [[kaggle](#)], [[openml](#)] einen Datensatz für die Klassifikation mit folgenden Eigenschaften aus:

- mindestens 500 Datenpunkte,
- mindestens 5 Komponenten,
- sowohl numerische als auch nominale/kategoriale Komponenten.

Sofern dieser noch nicht im CSV-Format vorliegt, wandeln Sie in mit externen Tools (Tabellenkalkulation etc.) in eine CSV-Datei mit einem Tabellenkopf um. Spaltentrenner sollte ein Komma, Dezimaltrenner ein Punkt sein. Speichern Sie diese Datei lokal ab, da Sie diese für die weiteren Hausaufgaben benötigen, und kopieren Sie in Ihr Jupyter-Notebook-Verzeichnis.

Erstellen Sie nun ein Jupyter Notebook, das folgende Aufgaben erledigt:

- Dokumentieren Sie den verwendeten Datensatz ([URL](#) und kurze Erläuterung der Eigenschaften) als Markdown-Zelle am Anfang des Notebooks
- Erstellen Sie eine Klasse, die einen Datensatz (Zeile) Ihrer CSV-Datei beschreibt. Die Spaltenwerte (Features) sollen als private Komponenten abgebildet werden, für die Getter benötigt werden.
- Lesen Sie Ihre Datendatei mit Hilfe des Moduls `csv` aus diesem Verzeichnis ein und erzeugen Sie eine Liste von Objekten, die diese Datensätze beschreiben.
- Schreiben Sie eine Funktion, mit der Sie eine bestimmte Komponente (Spalte) aus Ihrer Objektliste ermitteln und geben Sie diese als eine Python-Liste zurück. Der Komponentename soll als Stringparameter übergeben werden. Beispiel für den Aufruf (für eine Spalte *Size*):

```
sizes = getFeature(objectList, 'Size'):
sizes # → [3, 17, 5, ..., 8]
```

- Schreiben Sie folgende Funktionen, die aus der vorher erzeugten Spalte statistische Kenngrößen ermitteln und diese zurückgeben:
 - Minimum
 - Maximum
 - Mittelwert
 - Median
- Schreiben Sie eine Funktion, der Sie eine numerische Spalte als Liste übergeben, die ein Histogramm als ASCII-Graphik erzeugt. Teilen Sie dazu das Intervall zwischen kleinstem und größtem Wert in 8 gleiche Intervalle und zählen Sie die Werte, die in jedes Intervall fallen. Diese Zahl soll durch eine Reihe von Rauten repräsentiert werden, wobei die Gesamtzahl von Rauten 200 sein soll.

Für das Gesamtgewicht von Abalone würde die Ausgabe etwa so aussehen:

```
Total weight
0.00-- 0.35: #####
0.35-- 0.71: #####
0.71-- 1.06: #####
1.06-- 1.41: #####
1.41-- 1.77: #####
1.77-- 2.12: #####
2.12-- 2.47: #
2.47-- 2.83:
```

- Erstellen Sie eine oder mehrere Zellen, in denen Sie Ihre Funktionen für eine oder mehrere numerische Features (Spalten) testen.
- Fügen Sie vor den Funktionsdefinitionen und den Tests beschreibende Textzellen in Ihr Notebook ein.
- Schreiben Sie jeweils eine kurze Auswertung Ihrer berechneten Ergebnisse als Textzelle.

Hinweise

Nutzen Sie die Python-Dokumentation. Es ist nur die Verwendung von Python3-Standard-Modulen zulässig (also weder Pandas noch SciPy oder ähnliche Bibliotheken).

Die Datensätze *Abalone*, *Diamonds* (bzw. *Brilliant Diamonds*) und des Data Mining Cups 2019 werden im Script bzw. den Screencasts verwendet und dürfen deshalb nicht für die Hausaufgaben verwendet werden.

Vermeiden Sie die Verwendung der gleichen Datendatei wie Ihre Kommilitonen.

Geben Sie sowohl Datendatei als auch Ihr Jupyter Notebook gepackt als ZIP-Datei über OPAL ab.

- *Einführung*
- *Dateiarbeit*
- *Zugriff, Datenbereinigung*
- *Datenanalyse*
- *Kategoriale Daten*
- *Kategorisieren*
- *Gruppieren von Daten*
- *Graphische Auswertungen*
- *Auswahl einer Stichprobe*
- *Aufgaben*

5.1 Einführung

Pandas [[pandas](#)] ist eine Bibliothek für die Verarbeitung und Analyse statistischer Daten. Sie baut auf [NumPy](#) auf, soll jedoch zuerst diskutiert werden, da wir sie für die Aufbereitung unserer Daten zuerst benötigen.

Pandas stellt Klassen für die Bearbeitung von *Data Frames* und *Zeitreihen* (Time Series) zur Verfügung. Die Funktionalität ist der von *R* nachgeahmt. Das Paket wird üblicherweise unter dem Alias `pd` importiert:

```
import pandas as pd
```

Das für uns wichtigste Objekt ist der *Data Frame*. Er stellt eine Tabelle mit Zeilen- und Spaltennamen zur Verfügung. Für unsere weiteren Untersuchungen stellen die Zeilen einzelne Beobachtungen bzw. Datenpunkte und die Spalten die Komponenten (*features*) dar. Eine einzelne Spalte ist dabei vom Typ `Series`. Sie verhält sich weitestgehend wie eine einfache Python-Liste, stellen aber zusätzlich einen Index mit frei wählbaren Werten für jedes Element zur Verfügung. Im Gegensatz zu *NumPy*-Arrays können die Spalten eines *Data Frames* unterschiedliche Typen haben. Alle Werte einer Spalte sollten dagegen für unsere Zwecke vom gleichen Typ sein.

Ein Data Frame kann einfach aus einem Dictionary von Listen (bzw. Serien) oder aus einem *NumPy*-Array erzeugt werden:

```
# Erzeugen aus Listen und Spaltennamen
colList = {'key1': liste1, ...}
df1 = pd.DataFrame(liste1, ...)
```

Beispiel: Geschlecht, Alter und Größe einer Personengruppe.

```
sex = ['m', 'f', 'f', None]
age = [24, 31, 52, 14]
size = [185, 171, 155, 163]
persons = pd.DataFrame({'sex': sex, 'age': age, 'size': size})
```

5.2 Dateiarbeit

Eine wichtige Funktionalität ist das Einlesen von *CSV*-Dateien in Data Frames sowie das Speichern von Data Frames in Dateien.

```
df = pd.read_csv('dateiname', header=wert, names=spaltennamenListe,
                 sep='spaltentrenner', decimal='dezimaltrenner',
                 quotechar='stringbegrenzung',
                 na_values='fehlwert' ...)
```

Die meisten Parameter sind optional und nur erforderlich, wenn das Datenformat von den Vorgabewerten abweicht: Spaltentrenner ist das Komma, Dezimaltrenner ist der Punkt. Wichtig ist die Angabe von `header` und `names`. Enthält die erste Zeile der *CSV*-Datei die Spaltennamen, können diese beiden Angaben entfallen. Andernfalls ist für `header` der Wert `None` anzugeben, in `names` können die Spaltennamen individuell gesetzt werden.

Bemerkung: Die Funktion `read_csv()` kann auch mit *ZIP* komprimierte Dateien direkt lesen, ein vorheriges Entpacken ist nicht erforderlich.

Nach dem Einlesen sollten mit der Methode `info()` die Spaltennamen, der korrekte Spaltentyp und die Zahl der Werte kontrolliert werden. Die Methode `head()` liefert die ersten Zeilen des Data Frames, Ein optionaler Parameter spezifiziert deren Zahl.

```
print(df.info())
print(df.head(/zeilenzahl/))
```

Beispiel: Einlesen von `diamonds.csv` (Header vorhanden) und `abalone.data` (*UCI*-Datei ([uci]), keine Header).

```
diamonds = pd.read_csv("diamonds.csv")

# Einlesen von CSV-Daten ohne Spaltenkopf
colnames = ['sex', 'length', 'diameter', 'height', 'whole', 'shucked',
            'viscera', 'shell', 'rings',]
abalone = pd.read_csv("abalone.data", header=None, names=colnames)

# Struktur und erste Zeilen anzeigen
print(abalone.info())
print(abalone.head(10))
```

Hinweis: Statt aus einer Datei kann auch direkt von einer *URL* gelesen werden, z. B. ein *UCI*-Datensatz:

```
abalone = pd.read_csv(
    'http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data',
    header=None,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
names=colnames
)
```

Für das Einlesen anderer Datenformate stehen die Funktionen `read_json()`, `read_excel()`, `read_html()` und `read_sql()` zur Verfügung (s. Dokumentation).

Die Spalten können auch nachträglich benannt bzw. umbenannt werden. Sollen alle Spalten einen Namen bekommen, ordnet man der Eigenschaft `columns` eine Liste von Spaltennamen zu. Zum Umbenennen einzelner Spalten verwendet man die Methode `rename()`, der als Parameter ein Dictionary mit alten und neuen Namen der betreffenden Spalten mitgegeben wird.

```
# Alle Spaltennamen setzen
dataFrame.columns = stringliste

# Einzelne Spaltennamen ersetzen
dataFrame.rename(columns={'alterName': 'neuerName', ...}, inplace=True)
```

Das Schreiben eines Data Frame geschieht mit der Methode `to_csv()`:

```
dataFrame.to_csv('fileName' [, index=False...])
```

Der Parameter `index=False` verhindert das Schreiben der Zeilennummern als eigene Datenspalte und sollte deshalb normalerweise gesetzt sein.

Bemerkung: Die Methode `to_csv()` kann auch direkt [ZIP](#)-komprimierte Dateien schreiben, wenn der Dateiname auf `.zip` endet.

Notebook

Einlesen der [UCI](#)-Datei `abalone.dat`, Kontrolle der Spaltentypen und Speichern als [CSV](#)-Datei: `abalone_io.ipynb`.

5.3 Zugriff, Datenbereinigung

Einfacher ist die Spaltenauswahl über den Namen, der einfach als Index benutzt werden kann. Listen sind ebenfalls möglich.

```
spalte = df['spaltenname']
# oder:
spalte = df.spaltenName
teilFrame = df['spalte1', 'spalte2', ...]
```

Die zweite Form mit dem durch einen Punkt angehängten Spaltennamen ist immer dann möglich, wenn der Spaltenname ein gültiger Python-Bezeichner ist. Die Spalte selbst hat den Typ `Series`, der einen erweiterten Listentyp darstellt: Alle Listenobjekte haben den gleichen Datentyp, neben dem normalen numerischen Index können weitere, beschreibende Indices zugeordnet werden.

Es ist ebenfalls ein Zugriff auf Zeilen und Spalten über den numerischen Index ähnlich wie bei *Python*-Listen und Feldern möglich, wir müssen dazu allerdings die Methode `iloc()` verwenden:

```
auswahl = dataFrame.iloc[zeilenindex, spaltenindex]
```

Im Gegensatz zu normalen Feldern wird nur ein Paar eckiger Klammern benutzt, die Dimensionen (*Achsen*) sind durch Komma getrennt. Eine komplette Zeile bzw. Spalte wird durch die Angabe `:` für die entsprechende Achse erzielt.

Um mit Indices zu arbeiten, benötigen wir häufig die Zahl der Zeilen und Spalten des Dataframe, die uns die Eigenschaft `shape` in Form eines Tupels liefert:

```
df.shape
```

Die erste Wert (Achse 0) liefert die Zeilen- der zweite (Achse 1) die Spaltenzahl des Data Frames.

Beispiel: Auswahl der letzten Spalte von *Abalone* über den numerischen Index:

```
abalone.iloc[:, -1]
# oder mittels shape
abalone.iloc[:, abalone.shape[1]-1]
```

Beispiel: Auswahl der letzten 20 Zeilen und aller Spalten außer der ersten und letzten Spalte:

```
abalone.iloc[-20:, 1:-1]
# oder mittels shape
abalone.iloc[abalone.shape[0]-20:, 1:abalone.shape[1]-1]
```

Benötigen wir nur Werte aus einer Spalte, ist der Zugriff über den Spaltennamen und dann die Zeilenauswahl einfacher. Beispiel: Anzahl der Ringe für die 11. bis 20. Zeile:

```
abalone.rings[11:21]
```

Eine neue Spalte kann einfach durch Zuweisen einer Liste bzw. Serie an einen neuen Spaltennamen hinzugefügt werden:

```
df[neuerName] = listeOderSerie
```

Die Punktschreibweise ist hier nicht möglich.

Beispiel: Hinzufügen einer Spalte, die das Verhältnis aus Höhe und Durchmesser enthält, und einer Prüfspalte, die wir zunächst mit Nullen füllen. Wir können die üblichen Binäroperationen auf ganze Spalten anwenden, sie werden dann elementweise durchgeführt – das ist deutlich effizienter als die elementweise Ausführung in einer Python-Schleife. Ebenso können wir statt einer Liste einen Einzelwert verwenden, der automatisch auf die gesamte Spalte dupliziert wird.

```
abalone['relSize'] = abalone.height/abalone.diameter
abalone['check'] = 0
```

Wollen wir umgedreht Spalten ausschließen, verwenden wir die Methode `drop()`. Diese benötigt eine Spalte oder Spaltenliste und einen zweiten Parameter `axis` mit dem Wert 1 (= die zweite Dimension des Data Frames).

```
df = df.drop('spalte', axis=1)
# mehrere Spalten gleichzeitig:
df = df.drop(['spalte1', 'spalte2', ...], axis=1)
```

Beispiel: Beseitigen der beiden hinzugefügten Spalten.

```
abalone = abalone.drop(['relSize', 'check'], axis=1)
```

In der Praxis haben wir häufig mit unvollständigen Daten zu tun, z. B. wurden in Fragebögen nicht alle Felder ausgefüllt oder bei Messungen sind einzelne Sensoren ausgefallen. Für die Modellerstellung benötigen wir jedoch vollständige Daten. Es ist also eine Bereinigung erforderlich.

Wir haben folgende Möglichkeiten, mit fehlenden Daten umzugehen:

Löschen des Datensatzes Sind nur wenige Datensätze betroffen, ist das die einfachste Möglichkeit. Allerdings kann unser Modell dann auch keine Vorhersagen für unvollständige Daten treffen.

Löschen der betroffenen Komponente (Feature) Fehlen Werte nur in einiger oder in wenigen Komponenten, kann man auch diese aus der Modellbildung ausschließen. Trägt diese Komponente jedoch wesentlich zum Modell bei, wird durch das Entfernen unser Modell schwächer.

Ersetzen durch einen Defaultwert Bei numerischen Komponenten könnte man fehlende Werte durch das Mittel aller vorhandenen Werte oder auch durch einen minimalen Wert ersetzen. Bei kategorialen Komponenten könnte man entweder die häufigste Kategorie oder eine neue Kategorie unbekannt verwenden.

Je nach Problem muss entschieden werden, welche Variante am besten geeignet ist.

Wir betrachten als Beispiel zur Datenbereinigung den *UCI*-Datensatz *Pittsburgh Bridges*: <https://archive.ics.uci.edu/ml/datasets/Pittsburgh+Bridges> und verwenden Variante 2. Fehlende Werte sind durch ? dargestellt.

```
cols = ['id', 'river', 'loc', 'erected', 'purpose', 'len', 'lanes',
        'clear', 't_d', 'mat', 'span', 'rel', 'type']
bridges = pd.read_csv(
    'https://archive.ics.uci.edu/ml/machine-learning-databases/bridges/bridges.data.
    ↪version2',
    header=None,
    names=cols,
    na_values='?',
)
```

Wir prüfen mit

```
bridges.info()
```

die korrekten Datentypen der Spalten und zeigen uns mit

```
bridges.head()
```

den Tabellenkopf an. Hier sehen wir in den Spalten `len` und `span` die ersten fehlenden Werte.

Die Methode `isna()` erzeugt einen Dataframe mit logischen Werten, die an jeder Position mit einem Fehlwert den Wert `True` besitzt. Um die komplette Zeile auszuwählen, wenden wir darauf die Methode `any()` an, die alle Werte einer Zeile auf `True` setzt, wenn ein Wert entlang dieser Achse `True` ist (also in der Ursprungsmatrix einen Fehlwert enthielt). Wir müssen uns dabei innerhalb jeder Zeile über die Spalten bewegen, die *Achse* hat also den Wert 1. Diese logische Matrix können wir als Index (Maske) eines Dataframes verwenden, es werden alle Werte (also hier komplette Zeilen) übernommen, für die die Maske `True` ist:

```
naLines = bridges[bridges.isna().any(axis=1)]
```

Wir erhalten 38 Zeilen und sehen Fehlwerte sowohl in numerischen als auch kategorialen Komponenten. Die numerischen Komponenten ersetzen wir mit `interpolate()` durch den Mittelwert. Das können wir entweder einzeln pro Spalte oder für den gesamten Dataframe gemeinsam machen. Der Mittelwert soll jeweils innerhalb einer Spalte über alle Zeilen gebildet werden, wir müssen uns also entlang der Achse 0 bewegen.

```
# Einzelspalte (nur eine Spalte, keine Achsenangabe erforderlich)
bridges['lanes'] = bridges['lanes'].interpolate()
# oder gesamter Data Frame über alle numerischen Spalten
# erzeugt neuen Data Frame
bInt = bridges.interpolate(axis=0) # axis=0 ist Vorgabe und kann entfallen
```

Für die kategorialen Spalten kann die Ersetzung mit der Methode `fillna()` erfolgen. Hier ist es meist sinnvoll, die Spalten getrennt zu behandeln, da der Füllwert sich oft unterscheiden wird. Beispiel für `len`, wir ergänzen eine neue Kategorie:

```
bInt['len'] = bInt['len'].fillna('UNKNOWN')
```

Das Löschen von Zeilen bzw. Spalten mit Fehlwerten erledigt die Methode `dropna()`. Diese erwartet einen Parameter `axis`, der festlegt, ob Zeilen (`axis=0`, Vorgabe) oder Spalten (`axis=1`) mit Fehlwerten komplett gelöscht werden sollen. Beispiel: Löschen der Spalte `span` mittels der oben besprochenen Methode `dropna()` und anschließend aller Zeilen, die nach dem Auffüllen immer noch Fehlwerte enthalten.

```
bNoSpan = bInt.drop('span', axis=1)
bNoMissing = bNoSpan.dropna(axis=0) # axis=0 kann entfallen
```

Notebook zum Behandeln der Fehlwerte

Download: `bridges-handle-na.ipynb`.

5.4 Datenanalyse

Nachdem wir mit `info()` und `py:func:head` die Plausibilität der Daten sowie die Korrektheit der Datentypen geprüft haben, wollen wir nun deren statistische Eigenschaften genauer untersuchen.

Die Methode `describe()` liefert uns die statistischen Grunddaten für alle numerischen Komponenten. Wir betrachten wieder unser Beispiel *Abalone*, das wir unter dem Namen `abalone` als Data Frame geladen haben.

```
abalone.describe()
```

Uns werden für jede numerische Komponente die Anzahl der Werte (ohne Fehlwerte), Mittelwert (*mean*), Standardabweichung (*std*) und die Quartile inklusive Minimum und Maximum angezeigt.

Benötigen wir für eine bestimmte Spalte davon nur einzelne Größen, werden uns passende Methoden bereitgestellt:

`spalte.min()`, `spalte.max()` Minimum bzw. Maximum der Spalte

`spalte.mean()`, `spalte.median()` Mittelwert bzw. Median der Spalte

`spalte.std()`, `spalte.var()` Standardabweichung und Varianz (Quadrat der Standardabweichung) der Spalte

`spalte.quantile(wahrscheinlichkeiten)` Quantile der Liste der angegebenen Wahrscheinlichkeiten (Einzelwert oder Liste). Mit der Liste `[0, 0.25, 0.5, 0.75, 1]` erhält man die Quartile.

`spalte.argmin()`, `spalte.argmax()` Index des kleinsten bzw. größten Wertes.

`spalte.sum()` Summe aller Werte.

Beispiel: 10%-Quantile für den Durchmesser von *Abalone*.

```
# Liste [0.0, 0.1, ..., 0.9, 1.0] erzeugen
# besser np.arange() oder np.linspace() verwenden
quantiles = [0.1*x for x in range(11)]
print(abalone.diameter.quantile(quantiles))
```

Beim Arbeiten insbesondere mit Zeitverläufen können noch folgende Methoden nützlich sein:

`spalte.sum()` Kumulierte Summe. Es entsteht eine neue Serie, die die Summe der Werte vom ersten bis zum aktuellen Wert enthält.

`spalte.diff()` Differenz benachbarter Werte. Der erste Eintrag der neuen Serie kann nicht berechnet werden und erhält den Wert 0.

Beispiel: Differenzen der benachbarten Quadratzahlen, deren kumulierte Summe ergibt wieder die Quadratzahlen selbst.

```
x = pd.Series([x*x for x in range(11)])
dx = x.diff() # -> [NaN, 1, 3, 5, ..., 19]
# Ersten Wert ersetzen
dx[0] = x[0]
cumX = dx.cumsum() # -> [0, 1, 4, ..., 100]
```

Um einzuschätzen, ob unsere Zielgröße überhaupt von den Komponenten (Features) unserer Daten beeinflusst wird, berechnen wir die Korrelationsmatrix:

```
coMatrix = dataframe.corr()
```

Aus dieser wählen wir die Spalte unserer Zielgröße aus und sehen nun die Korrelationskoeffizienten jedes Features zu diesem Wert.

```
coMatrix['zielgröße']
```

Diese Spalte kann noch mit `sort_values()` auf- oder absteigend geordnet werden.

Beispiel: Korrelation der Ringzahl von Abalone mit den numerischen Features. Das Geschlecht ist eine kategoriale Größe und fehlt deshalb.

```
coAbalone = abalone.corr()
print(coAbalone['rings'].sort_values(ascending=False))
```

Notebook

Download: `abalone-statistics.ipynb`.

5.5 Kategoriale Daten

Um mit nichtnumerischen (kategorialen) Daten arbeiten zu können, ist eine geeignete Umwandlung in numerische Daten erforderlich. Wir unterscheiden hier zwei Typen:

Ordinaldaten: Die Daten besitzen eine natürliche Ordnung (schlecht/durchschnittlich/gut, 1 bis 5 Sterne).

Nominaldaten (nicht geordnet) Die Daten besitzen keine Ordnung (Geschlecht, Farbe)

Im ersten Fall kann man häufig die Daten einfach durch eine Zahlenfolge ersetzen. Macht man dies im zweiten Fall, fügt man dem Modell künstlich eine Ordnung zu, die in der Wirklichkeit nicht vorhanden ist, und verfälscht damit das Modell.

Wir wollen zunächst die Eigenschaften der kategorialen Daten untersuchen. Geben wir der Methode `describe()` den Parameter `include='object'` mit, erhalten wir die statistischen Grunddaten der kategorialen Komponenten:

count Anzahl der Werte ohne Fehlwerte

unique Anzahl der verschiedenen Kategorien

top Die am häufigsten auftretende Kategorie

freq Die Anzahl der Werte der häufigsten Kategorie

Beispiel: *Abalone*. Wir haben nur eine kategoriale Komponente *sex*.

```
abalone.describe(include='object')
```

Die Liste aller Kategorien erhalten wir mit der Methode

```
spalte.unique()
```

Benötigen wir die Anzahl der Werte jeder und nicht nur der häufigsten Kategorie, verwenden wir die Methode

```
spalte.value_counts()
```

Beispiel: Kategorien und Anzahl der Werte jeder Kategorie für *Abalone*.

```
print('Kategorien:', abalone.sex.unique())
print('Anzahl pro Kategorie: ')
print(abalone.sex.value_counts())
```

Wollen wir den kategorialen Daten numerische Werte zuordnen, haben wir folgende zwei Möglichkeiten:

manuelle Zuordnung der Codes sinnvoll für ordinale Daten, bei denen die Reihenfolge erhalten bleiben muss oder eine feste Zuordnung bestimmter Codes zur Kategorie gewünscht wird. Die vorhandenen Kategorien müssen bekannt sein.

automatische Zuordnung (Faktorisieren) Die Zuordnung erfolgt automatisch. Fehlwerten kann ebenfalls ein bestimmter Wert zugewiesen werden.

Für die erste Variante erstellen wir ein `Dictionary` der Form

```
mapping = {
    'schluessel1': wert1,
    # ...
}
```

und wenden dieses auf die Spalte an:

```
df['codeSpalte'] = spalte.map(mapping)
```

Für die zweite Variante wenden wir die Methode `factorize()` eine Kategoriespalte an:

```
werte, mapping = spalte.factorize()
```

Die Werte können wir einer neuen Spalte des Data Frame zuordnen, `mapping` enthält eine geordnete Liste, die es ermöglicht, den codierten Werten wieder den ursprünglichen Wert zuzuordnen.

Beispiel: Manuelles und automatisches Codieren der Spalte `sex` von *Abalone*.

```
# manuell
sexCodes = {'I': 0, 'F': 1, 'M': 2}
abalone['isex'] = abalone.sex.map(sexCodes)
# automatisch
fvals, fcodes = abalone.sex.factorize()
abalone['fsex'] = fvals
# Anzeige beider Varianten, Auswahl nur der Kategoriespalte
abalone.loc[:, ['sex', 'isex', 'fsex']].head(10)
```

Notebook

Analyse der kategorialen Daten von *Abalone*, manuelle und automatische Umwandlung. Download: `abalone-categories.ipynb`.

5.6 Kategorisieren

Umgedreht kann es auch erforderlich sein, eine stetige Größe in eine Anzahl von Kategorien umzuwandeln. Für unseren Beispieldatensatz wollen wir häufig nicht die exakte Anzahl der Ringe vorhersagen, sondern nur, ob es sehr wenig (Tier zu jung), eine mittlere Anzahl (zum Verzehr geeignet) oder eine große Anzahl Ringe (Tier zu alt) hat.

Pandas stellt uns zwei Methoden zur Verfügung. Die Methode `qcut()` (Quantile cut) erwartet nur eine Anzahl der gewünschten Klassen und erledigt die Zerlegung automatisch so, dass in jede Klasse etwa gleichviele Objekte fallen.

```
klassen = pd.qcut(datenSpalte, q=klassenZahl[labelListe])
```

Beispiel: Zerlegen der Ringzahl in 3 Klassen.

```
qClasses = pd.qcut(abalone.rings, q=3)
# Anzeige der Aufteilung und der ersten Werte
print(qClasses.value_count())
qClasses.head()
```

Ergebnis:

```
(8.0, 11.0]      1810
(0.999, 8.0]     1407
(11.0, 29.0]      960
Name: rings, dtype: int64

0      (11.0, 29.0]
1      (0.999, 8.0]
2      (8.0, 11.0]
3      (8.0, 11.0]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
4      (0.999, 8.0]
Name: rings, dtype: category
Categories (3, interval[float64]): [(0.999, 8.0] < (8.0, 11.0] < (11.0, 29.0]]
```

Vergabe eigener Labels:

```
qClasses = pd.qcut(abalone.rings, q=3, labels=[0, 1, 2])
```

Die eigenen Labels verdecken nun allerdings die Intervallgrenzen.

Mit `qcut()` ist es nicht möglich, neu anfallende Daten in der gleichen Weise in Intervalle einzuteilen, da die Intervallgrenzen abhängig vom Datensatz jedesmal neu festgelegt werden. Deshalb ist es für solche Anwendungen sinnvoller, die Intervallgrenzen selbst festzulegen. Wir benötigen eine Intervallgrenze zusätzlich zur gewünschten Klassenzahl. Die erste Intervallgrenze muss kleiner als der kleinste, die letzte größer als der größte Wert unserer Daten sein (wir können $-\infty$ und $+\infty$ verwenden).

grenzen = sortierteWertListe

klassen = `pd.cut(datenSpalte, bins=grenzen[, labels=klassenLabelListe];kw:)`

Beispiel: Zerlegen der Ringzahl in 3 Klassen. Wir können als untere bzw. obere Grenze 0 bzw. 30 verwenden, nutzen hier aber $-\infty$ und $+\infty$.

```
import math
boundaries = [-math.inf, 8, 17, math.inf]
classes = pd.cut(abalone.rings, bins=boundaries)
print(cats.value_counts(sort=False))
```

Ergebnis:

```
(-inf, 8.0]      1407
(8.0, 17.0]      2634
(17.0, inf]      136
Name: rings, dtype: int64
```

Mit eigenen Klassenbezeichnungen (laufende Nummer):

```
# boundaries wie oben
labels = range(len(boundaries)-1)
classes = pd.cut(abalone.rings, bins=boundaries, labels=labels)
print(cats.value_counts(sort=False))
```

Ergebnis:

```
0      1407
1      2634
2       136
Name: rings, dtype: int64
```

Notebook

Automatische und manuelle Zerlegung der Ringzahl in Klassen. Download: `abalone-binning.ipynb`.

5.7 Gruppieren von Daten

Für die Analyse der Daten kann es hilfreich sein, diese nach bestimmten (kategorialen) Komponenten zu gruppieren und innerhalb der Gruppe auszuwerten. Das Gruppieren geschieht mit der Methode `groupby()`:

```
dataFrame.groupby(spaltenListe[, as_index=False])
```

Das Ergebnis ist ein Gruppierungsobjekt, auf das wir die gewünschte Funktion anwenden.

Die Größe der Gruppen zeigt uns die Methode `size()`:

```
gruppierung.size()
```

Wollen wir nur eine Funktion anwenden, können wir `apply()` verwenden:

```
gruppierung.apply(funktion)
```

Die Funktion selbst ist entweder ein Funktionsname oder ein `lambda`-Ausdruck mit der Signatur

```
function(dataFrame): return Wert
```

Um z. B. alle Werte einer Spalte zu summieren, schreiben wir eine Funktion

```
df.groupby('gruppe').apply(lambda df: df['spalte'].sum())
```

oder

```
def myColSum(df):  
    return df['spalte'].sum()  
  
df.groupby('gruppe').apply(myColSum)
```

Wollen wir auf verschiedene Spalten verschiedene Gruppierungen anwenden, verwenden wir die Gruppierungsmethode `aggregate()`:

```
gruppierung.aggregate({  
    'spalte': funktionsListe,  
    ...  
})
```

Wir übergeben also ein Dictionary. Die Schlüssel sind die gewünschten Spalten, die Werte sind entweder eine Funktion oder eine Liste von Funktionen, die auf diese Spalte angewendet werden sollen.

Wollen wir z. B. für unsere Brücken je nach Verwendung die minimale, mittlere und maximale Spurzahl wissen und zusätzlich die Gruppengröße sowie die Liste aller IDs dieser Gruppe ermitteln, verwenden wir:

```
bridges.groupby('purpose').aggregate({  
    'lanes': [pd.Series.min, pd.Series.mean, pd.Series.max, pd.Series.count],  
    'id': [pd.Series.count, list],  
})
```

Bemerkung: Die Anzahl der Werte für die Spalte `id` und `lanes` unterscheiden sich, da NA-Werte nicht mitgezählt werden.

Notebook

Download: `bridges-groupby.ipynb`.

5.8 Graphische Auswertungen

Neben der Ermittlung statistischer Grundgrößen bietet *Pandas* zahlreiche Möglichkeiten der graphischen Datenanalyse. Diese bauen auf die Bibliothek *Matplotlib* auf, die wir später im Detail betrachten werden (*Matplotlib*).

Wir betrachten wieder unseren Datensatz *Abalone*, in dem wir die Komponente *sex* durch numerische Werte ersetzen und die Spalte *rings* abspalten und in 3 Klassen umwandeln.

```
boundaries = [0, 7, 15, 30]
labels = range(len(boundaries)-1)
classes = pd.cut(abalone.rings, boundaries, labels=labels)
data = abalone.drop('rings', axis=1)
sexMap = {'I':0, 'M': 1, 'F': 2}
data['sex'] = data.sex.map(sexMap)
```

Die Wertverteilung von numerischen Spalten kann mittels Boxplot dargestellt werden.

`dataFrame.boxplot (/spaltenListe/)`

Beispiel: Alle Spalten und Auswahl *diameter* und *shell*.

```
abalone.boxplot()
abalone.boxplot(['diameter', 'shell'])
```

Ergebnis:

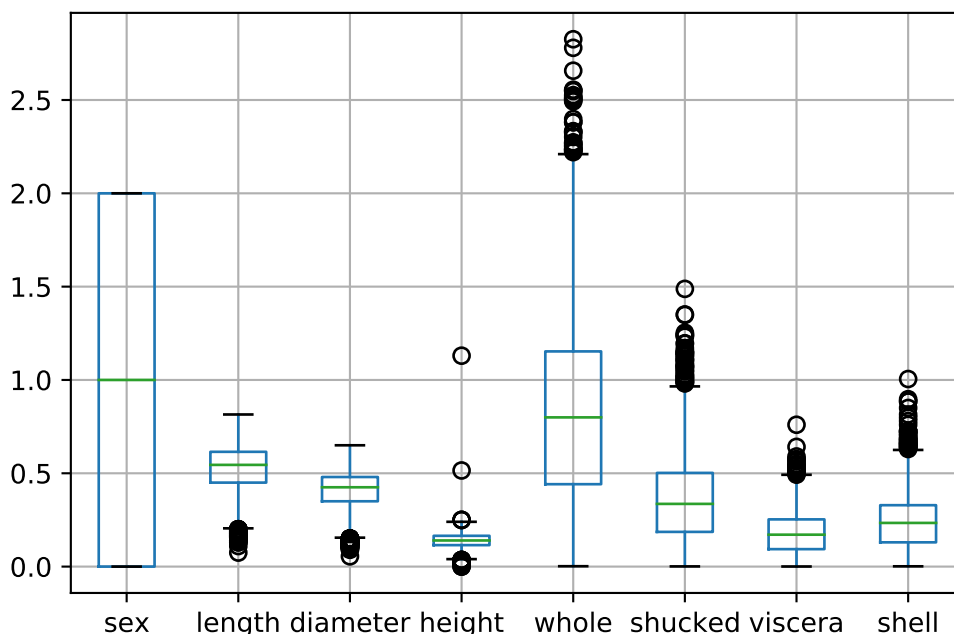


Abb. 5.1: Boxplot aller Features

Genauere Analysen der Datenverteilung liefern uns ein Histogramm

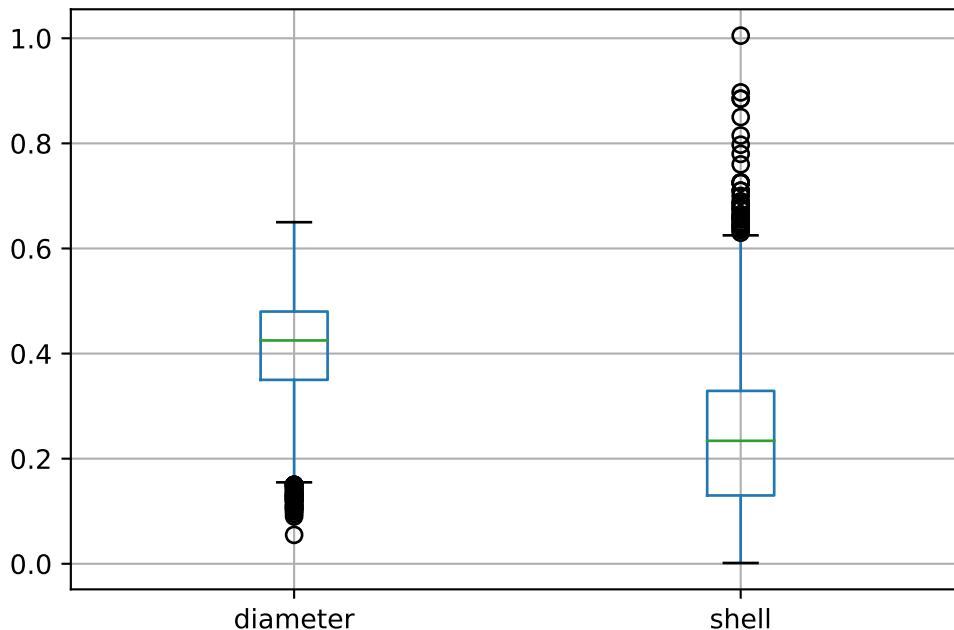


Abb. 5.2: Boxplot zweier Features

```
dataFrame.hist (/spaltenListe/)
```

oder ein Dichtediagramm (Achtung: abweichende Syntax!):

```
dataFrame.plot(kind='density',[, subplots=True])
```

Benötigen wir dies nur für eine Spalte, können wir die Methode auch mit dieser aufrufen.

Beispiele:

```
# Histogramm
abalone.hist()
# Dichte aller Features in einem Diagramm
abalone.plot(kind='density')
# Dichte aller Features in getrennten Diagrammen
abalone.plot(kind='density', subplots=True)
# Dichte des Durchmessers
abalone.diameter.plot(kind='density')
```

Haben wir kategoriale Komponenten, bieten sich Barplots oder Kuchendiagramme für die Häufigkeit der Daten in jeder Kategorie an. Wir zählen dazu zunächst die Werte mittels `value_counts()`:

```
haeufigkeiten = datenSpalte.value_counts (/sort=False/)
```

Der Parameter `sort=False` verhindert eine Sortierung nach Häufigkeit. Diese Häufigkeiten können wir nun als Barplot darstellen:

```
haeufigkeiten.plot(kind='bar')
```

Alternativ kann auch ein Tortendiagramm geeignet sein:

```
haeufigkeiten.plot(kind='pie')
```

Beispiel: Häufigkeitsverteilung der diskretisierten Zielgröße (4 Klassen).

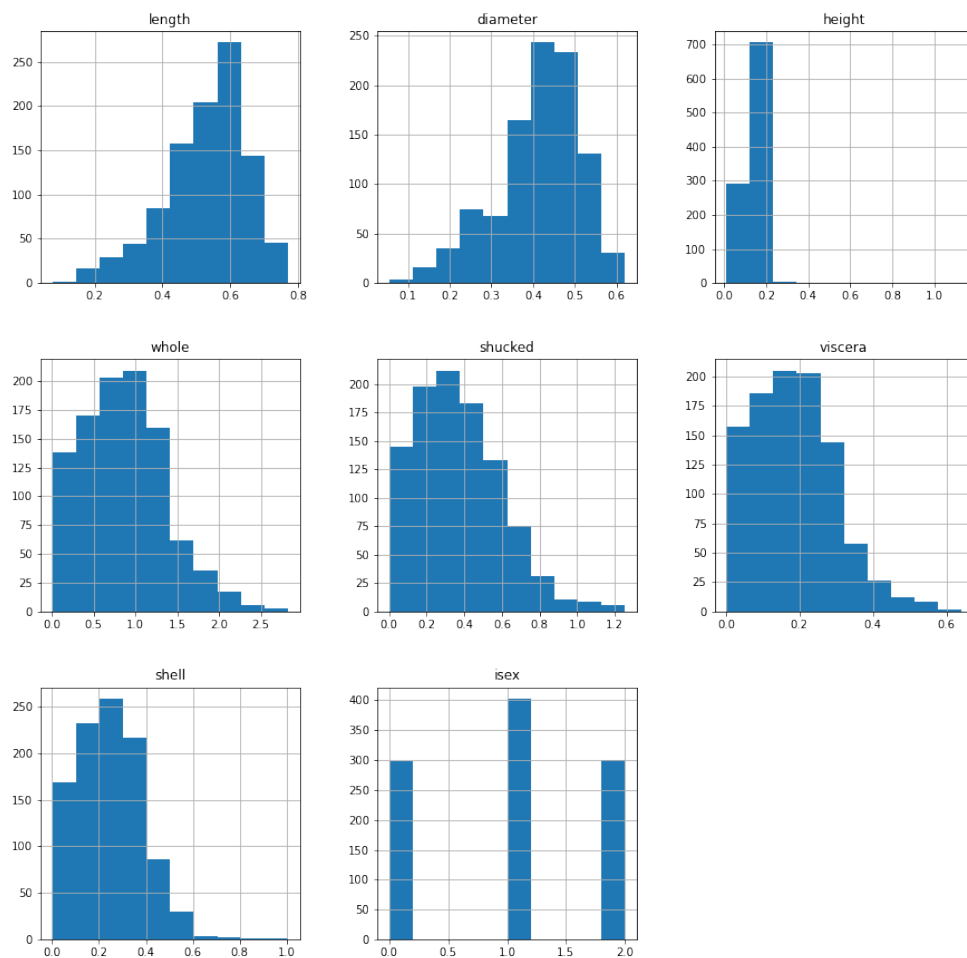


Abb. 5.3: Histogramm aller Features

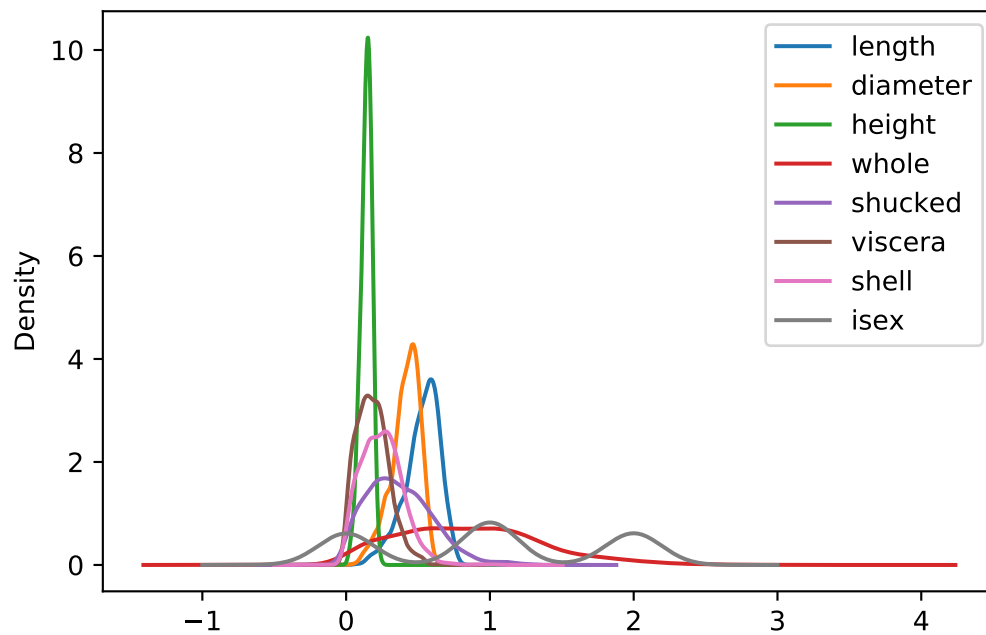


Abb. 5.4: Dichteverteilung aller Features in einem Diagramm

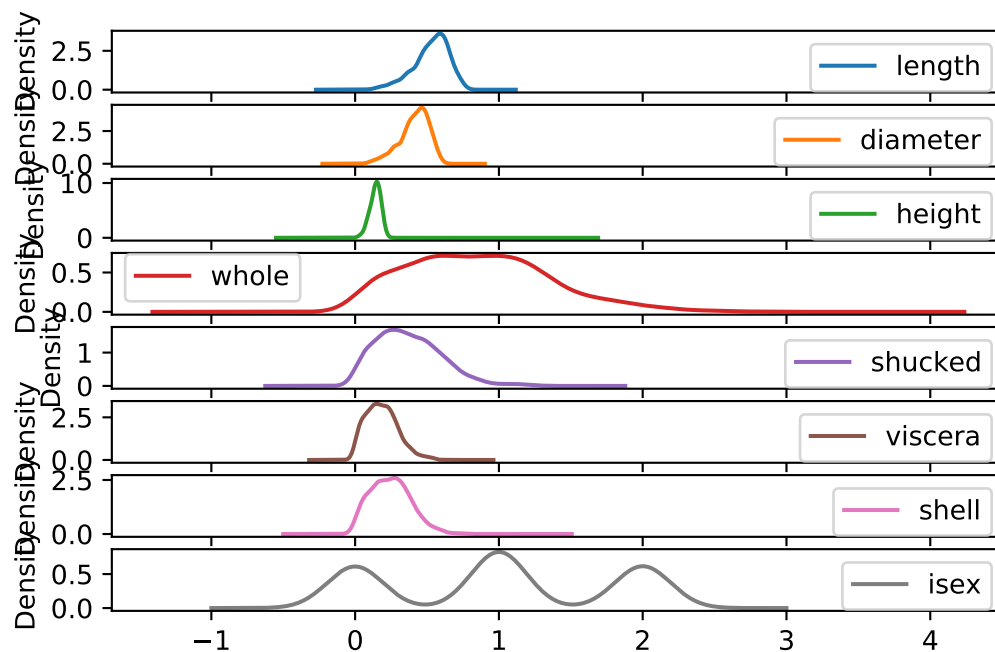


Abb. 5.5: Dichteverteilung aller Features in getrennten Diagrammen

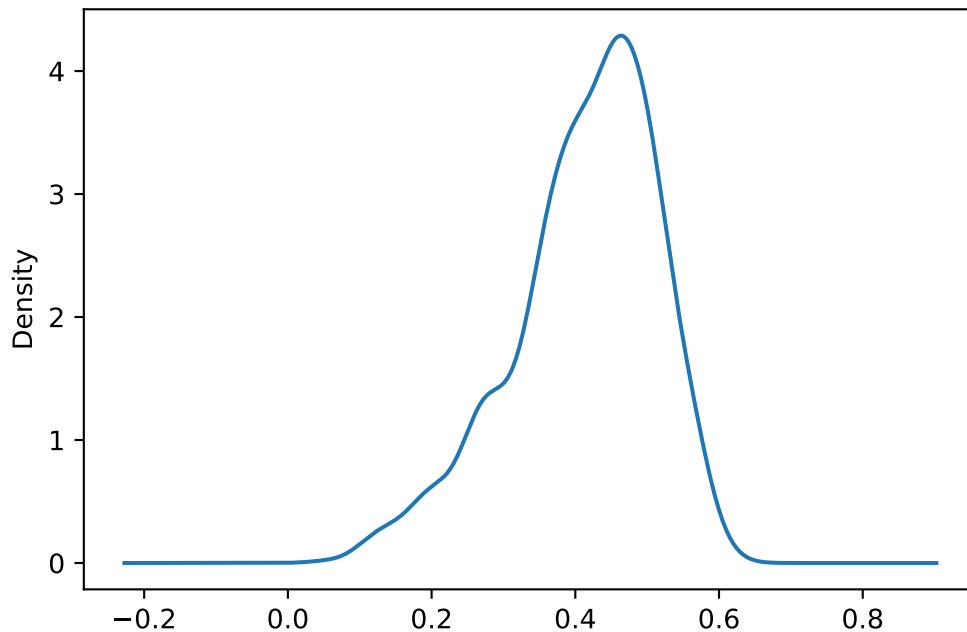


Abb. 5.6: Dichteverteilung des Durchmessers

```

boundaries = [0, 7, 12, 17, 30]
labels = ['zu jung', 'jung', 'mittel', 'zu alt']
classes = pd.cut(abalone.rings, boundaries, labels=labels)
classCounts = classes.value_counts(sort=False)
# Barplot
classCounts.plot(kind='bar')
# Tortendiagramm
classCounts.plot(kind='pie')

```

Neben der Analyse der einzelnen Features bzw. der Zielgröße ist oft auch eine Visualisierung des Zusammenhangs zwischen Features und Zielgröße erforderlich. Wir können in einem *Scatterplot* jeweils zwei Komponenten in ihrer gegenseitigen Abhängigkeit darstellen.

```
dataFrame.plot(xSpalte, ySpalte, kind='scatter')
```

Beide Spalten müssen dazu numerisch sein, wir müssen mittels `map()` oder `factorize()` Kategoriespalten vorher umwandeln.

Sinnvoll ist es häufig, im Scatterplot zwei Komponenten in ihrer Abhängigkeit darzustellen und die Zielgröße als Farbe des Datenpunktes zu codieren.

```
dataFrame.plot(xSpalte, ySpalte, kind='scatter' c='metrischeSpalte[,
cmap='farbschema'])
```

Beispiel. Scatterplot Durchmesser und Zielklasse (bei `cut()` wurden als Label die Zahlen 0 bis 3 benutzt, die Spalte muss dem Data Frame hinzugefügt werden).

```

numLabels = range(len(labels))
abalone['numClasses'] = pd.cut(abalone.rings, boundaries, numLabels)
abalone.plot('diameter', 'numClasses', kind='scatter')

```

Verhältnis Durchmesser-Gesamtgewicht. Abhängigkeit zur Zielgröße als Farbe. Die Vorgabefarben sind grau, deshalb wird eine andere Farbzuoordnung gewählt. Die Farbspalte muss hier nicht in den Data Frame eingefügt werden.

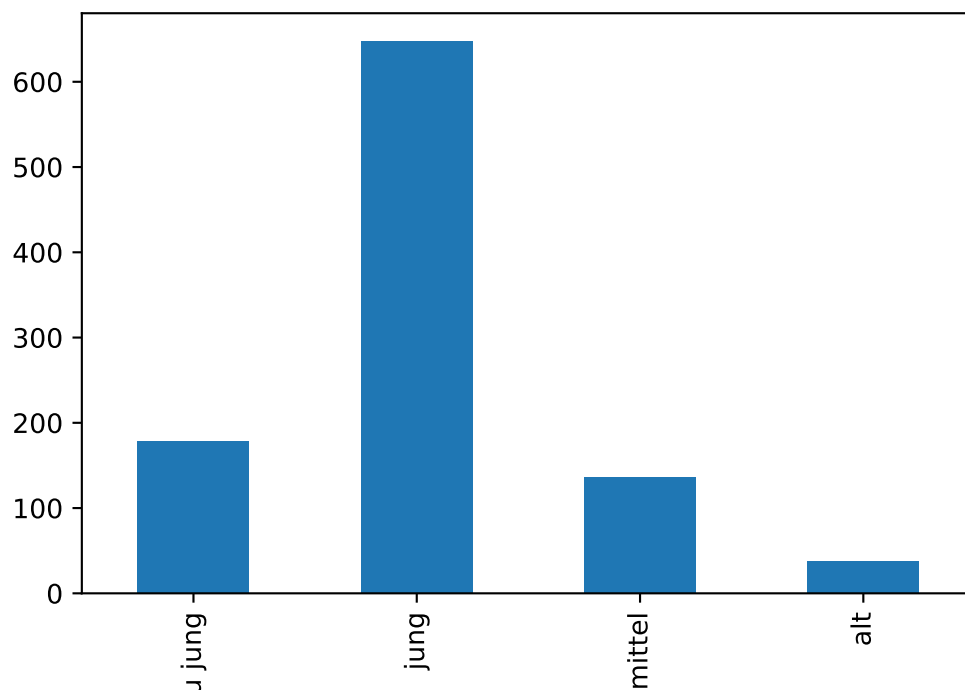


Abb. 5.7: Barplot diskretisierte Zielgröße

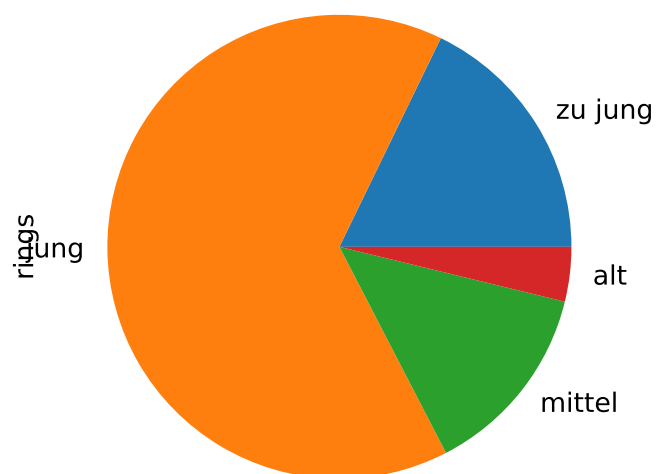


Abb. 5.8: Tortendiagramm diskretisierte Zielgröße

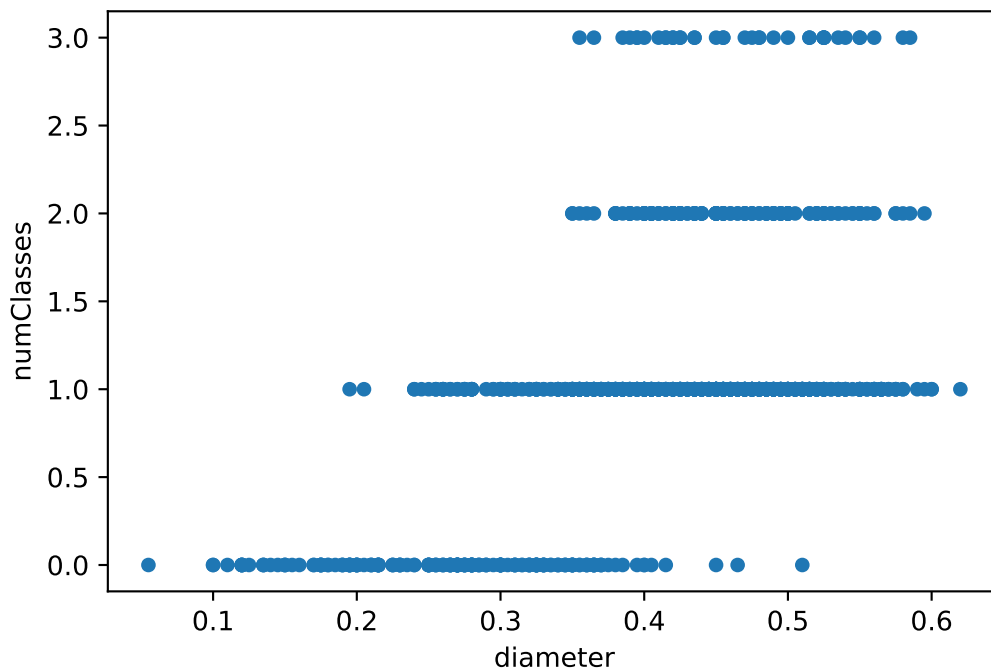


Abb. 5.9: Verhältnis Durchmesser-Ringzahl

```
abalone.plot('diameter', 'whole', kind='scatter', c=abalone.numClasses,
             cmap='hot')
```

Um diese Abhängigkeiten automatisch für alle numerischen Komponenten darzustellen, bietet sich die *Scatter Matrix* an.

```
pd.plotting.scatter_matrix(dataFrame ...)
```

Mit dem Parameter `c` kann wiederum der Farbvektor der Zielgröße, mit `cmap` das Farbschema gesetzt werden. Da die Scattermatrix recht groß wird, bietet es sich an, die Vorgabegröße des Diagramms mit dem Parameter

```
figsize=(breite, höhe)
```

zu überschreiben (z. B. (15, 15)).

Hinweis: Da die Berechnung der Scatter Matrix bei vielen Daten sehr lange dauert, empfiehlt es sich, hierfür nur eine zufällig ausgewählte Teilmenge zu verwenden (*Auswahl einer Stichprobe*).

Beispiel: Scattermatrix von Abalone. Die Klassen wurden wieder als Zahlen (0 bis 3) codiert, aber nicht dem Data Frame hinzugefügt.

```
pd.plotting.scatter_matrix(abalone, c=numClasses, cmap='hot',
                           figsize=(15,15))
```

Wir erkennen, dass viele Punkte der gleichen Farbe jeweils nahe beieinander liegen, unsere Zielgröße ist also von den Komponenten abhängig. Damit sollte die Erstellung eines Vorhersagemodells möglich sein.

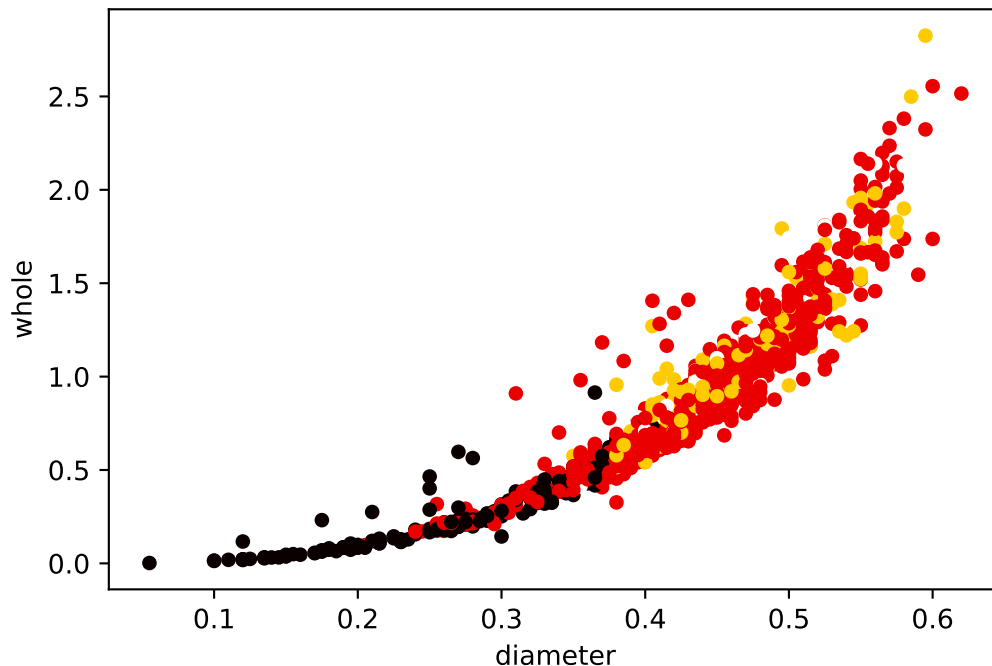


Abb. 5.10: Verhältnis Durchmesser-Gesamtgewicht, Ringzahl als Farbe

Notebook graphische Auswertungen

Download: `abalone-plots.ipynb`

5.9 Auswahl einer Stichprobe

Um die Güte unseres Modells prüfen zu können, dürfen wir nicht alle Daten zum Training verwenden, sondern wir müssen einen bestimmten Anteil für den Test zurückhalten. Sofern unser Datensatz noch nicht entsprechend aufgeteilt ist, müssen wir das selbst erledigen. Wenn auch *Scikit-Learn* dafür bequeme Methoden für die dynamische Zerlegung zur Verfügung stellt, kann es häufig sinnvoll sein, den Datensatz einmalig aufzuteilen und die Trainings- und Testdaten getrennt zu speichern.

Diese Aufteilung sollte zufällig erfolgen, ebenso sollten die Daten vorher gemischt werden, um systematische Differenzen zwischen Trainings- und Testdaten, die durch eine eventuelle Sortierung entstehen können, zu vermeiden.

Wir verwenden hier für die Zufallsfunktionen das Standardmodule `random`.

```
import random
```

Statt der Daten selbst wählen wir nur die gewünschten Indices aus und selektieren mit diesen die benötigten Daten. Dazu erzeugen wir uns zunächst eine Liste aller Indices:

```
indices = [i for i in range(dataFrame.shape[0])]
```

Unsere ausgewählten Data Frame erhalten wir dann mit der Teilliste der Indices:

```
teilFrame = dataFrame.iloc[teilIndices, :]
```

Wollen wir nur eine Stichprobe aus unseren Gesamtdaten ziehen (weil z. B. der Datensatz zu groß ist), können wir einfach die gewünschte Anzahl von Werten ziehen.

```
teilIndices = random.sample(indices, anzahl)
```

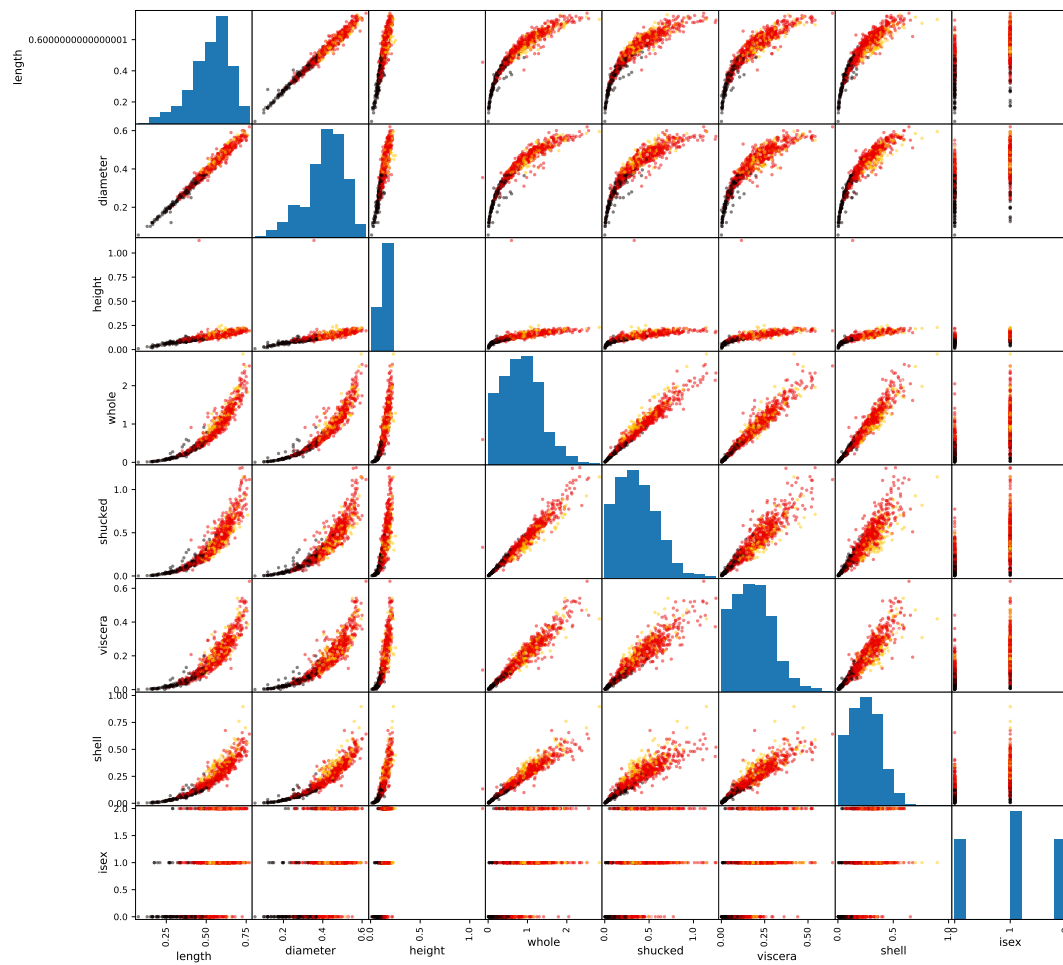


Abb. 5.11: Scattermatrix Abalone, Zielgröße als Farbe

Wir erhalten eine gemischte Liste ohne Doppelungen. Auf diese Weise lässt sich allerdings der Data Frame mit den nicht gewählten Indices nur schwer erzeugen. Deshalb können wir statt der Auswahl der Indices diese auch einfach mischen:

```
random.shuffle(indices)
```

Nun können wir diese durchmischten Werte einfach in zwei disjunkte Teillisten zerlegen:

```
trainingIndices = indices[:anzahl]
testIndices = indices[anzahl:]
```

Notebooks

Einlesen von *Abalone*, Codieren der Kategorien, Erzeugen von Kategorien aus Ringzahl, Zerlegen in Trainings- und Testmenge und Speichern beider Mengen.

Download: `abalone-train-test-split.ipynb`.

Um diese Daten einfach einzulesen, verwenden dazu ein Hilfs-Notebook `abalone_helper.ipynb`.

Quellcode 5.1: `abalone_helper.py`

```
#!/usr/bin/env python
# coding: utf-8

# # Funktionen zum Abalone-Import

import pandas as pd

def readFile(fileName):
    data = pd.read_csv(fileName)
    return data

def toLabels(data, isBinary):
    if isBinary:
        boundaries = [0, 15, 30]
    else:
        boundaries = [0, 7, 17, 30]
    labels = range(len(boundaries)-1)
    data['labels'] = pd.cut(data.rings, boundaries, labels=labels)
    # Umwandeln von Kategorie in Zahl
    data['labels'] = data['labels'].astype('int')
    return data.drop('rings', axis=1)

if __name__ == '__main__':
    data = readFile('abalone-test.csv')
    print(data.shape)
    print(data.head(10))

if __name__ == '__main__':
    dataLab = toLabels(data, True)
    print(dataLab.head(10))

if __name__ == '__main__':
    dataLab = toLabels(data, False)
    print(dataLab.head(10))
```

Beispiel: Einlesen der Testdaten.

```
import pandas as pd
import import_ipynb
from abalone_helper import readFile, toLabels

testData = readFile('abalone-test.csv')
abalone2 = toLabels(testData, False) # 2 Klassen
```

5.10 Aufgaben

Aufgabe 5.1 Analyse des ausgewählten Datensatzes

Lesen Sie den von Ihnen ausgewählten Datensatz mittels *Pandas* ein und analysieren Sie diesen. Behandeln Sie eventuell fehlende Werte, ersetzen Sie ordinale Daten durch (geordnete) Zahlen.

Untersuchen Sie numerisch und graphisch, wie stark die Zielgröße (Klasse) von den Features der Daten abhängt. Falls Sie feststellen, dass Ihr Datensatz für die weiteren Analysen ungeeignet erscheint, können Sie einen neuen Datensatz wählen. Die Zielgröße kann auch ein stetiger Wert (Gleitpunktzahl) sein, zerlegen Sie diese dann geeignet in 2 bis 4 Klassen.

Teilen Sie den Datensatz in Trainings- und Testdaten auf (ca. 80%, aber höchstens 5000 Daten für das Training, der Rest für den Test) und speichern Sie beide Datensätze als eigene Dateien.

Diese Vorbereitungen benötigen Sie für die nächsten Hausaufgaben.

- *Einführung*
- *NumPy-Array und Pandas Data Frame*
- *Indexzugriff*
- *Änderung der Form*
- *Spezielle Felder und Vektoren*
- *Funktionen*
- *Kategoriale Daten*
- *Lineare Algebra*
- *Meshgrid*
- *Aufgaben*

6.1 Einführung

Während *Pandas* Tabellen mit Datenspalten (*Features*) von unterschiedlichen Datentypen verwaltet, benötigen die meisten Lernverfahren die Daten in Form von zwei- oder mehrdimensionalen Feldern, in denen jedes Feldelement den gleichen Datentyp (meist eine Gleitpunktzahl) hat.

In Python können zweidimensionale Felder mit Hilfe von verschachtelten Listen erzeugt werden.

```
# z und s werden nur als Platzhalter benötigt  
matrix = [[startwert for s in range(spaltenZahl)] for z in range(zeilenzahl)]
```

Da die Arbeit mit derartigen Feldern nicht sonderlich effizient ist, verwenden Lernalgorithmen wie auch mathematische Verfahren spezielle Felder und Vektoren aus dem *NumPy*-Paket [[numpy](#)].

Hinweis: Dieses Kapitel soll lediglich einen kleinen Überblick über die Möglichkeiten zu *NumPy* geben. Wir werden diese Funktionen nicht unabhängig, sondern im Zusammenhang mit den weiter vorgestellten Bibliotheken benötigen.

Die hier vorgestellten Beispiele beziehen sich deshalb auch nicht direkt auf unsere Daten des Machine Learning.

Wir importieren dazu diese Paket und stellen es unter dem Alias `np` bereit:

```
import numpy as np
```

Wir können eine Liste von Listen einfach in ein solches Feld umwandeln:

```
array = np.array([zeilenliste], dtype='datentyp')
```

Die Zeilenliste darf wiederum eine Liste von Werten oder von Listen sein, um mehrdimensionale Felder (*Tensoren*) zu erzeugen. Alle Werte müssen den gleichen Datentyp haben. Wichtige Typen für uns sind: `int8`, `intc` (`int` in `C`), `bool`, `float32`, `float64`. Für die Features der Datenpunkte verwenden wir meist einen Fließkommadaten. Häufig genügt hier einfache Genauigkeit (`float32`), was bei bestimmten Lernverfahren (insbesondere mittels *ANN*) die Rechnungen beschleunigen kann.

Beispiele:

```
vector = np.array([1, 2, 3], dtype='float32') # eindimensional
matrix = np.array([[1, 2, 3], [4, 5, 6]], dtype='float32') # 2x3-Matrix
```

6.2 NumPy-Array und Pandas Data Frame

In den meisten Fällen wollen wir das *NumPy*-Array jedoch nicht manuell erzeugen, sondern die Daten aus dem *Pandas* Data Frame übernehmen, den wir aus einer Datei eingelesen und eventuell schon vorverarbeitet haben.

Unser Data Frame sollte vor der Umwandlung in ein *NumPy*-Feld nur noch numerische Werte enthalten.

Hinweis: Das später für das Machine Learning verwendete Framework *scikit-learn* stellt uns für die Datenkonvertierungen eine Vielzahl von Funktionen zur Verfügung.

Die Umwandlung geschieht durch den Aufruf der Methode

```
dataFrame.to_numpy(dtype='typeName')
```

oder kurz (ohne Möglichkeit der Typangabe):

```
dataFrame.values
```

Umgedreht kann ein *NumPy*-Array in einen Data Frame konvertiert werden:

```
df = pd.DataFrame(numpyArray[, columns=spaltenNamen])
```

Beispiel: Umwandlung der *Abalone*-Trainingsdaten in ein *NumPy*-Array.

```
import numpy as np
import pandas as pd

abalone = pd.read_csv('abalone-test.csv')
# Prüfen, dass alle Spalten numerisch sind
print(abalone.info())
X = abalone.to_numpy(dtype='float32')
# Erste Zeilen anzeigen, head() geht nicht!
print(X[:5,])
```

Beispiel: Erzeugen eines Data Frames aus einer *NumPy*-Matrix.

```
multtab = [[x*y for y in range(5)] for x in range(5)]
X = np.array(multtab)
columns = ['col {}'.format(x) for x in range(5)]
df = pd.DataFrame(X, columns=columns)
```

Notebook

Download: `numpy-pandas.ipynb`

6.3 Indexzugriff

Die Eigenschaften eines *NumPy*-Feldes (Datentyp `ndarray`) können wie folgt abgefragt werden:

`array.ndim` Zahl der Achsen (Dimensionen) des Feldes (in *TensorFlow* als *Rang* (*rank*) bezeichnet).

`array.shape` Dimension (Anzahl Komponenten) je Achse. Rückgabe als Tupel. Bei zweidimensionalen Feldern ist der erste Wert die Zeilen- der zweite die Spaltenzahl des Feldes.

Im Gegensatz zu *C*-Feldern und verschachtelten *Python*-Listen erfolgt der Zugriff auf *NumPy*-Array-Elemente durch ein Paar getrennte Klammern, die Indices der Dimensionen sind dabei durch Kommata getrennt:

```
array[index0, index1 ...]
```

Beispiel: Summe aller Werte eines zweidimensionalen Feldes `X` (z. B. erzeugt aus dem Data Frame *Abalone*).

```
rows, cols = X.shape
accu = 0.0
for r in range(rows):
    for c in range(cols):
        accu += X[r,c]
print(accu)
```

Hinweis: Für diese Operation stellt *NumPy* die Funktion `sum()` bereit, die wesentlich effizienter arbeitet.

Beispiel: Skalieren aller Werte bezüglich dieser Summe.

```
for r in range(rows):
    for c in range(cols):
        X[r,c] /= accu
```

Neben Einzelementen können wir auch ganze Zeilen, Spalten oder Bereiche des Feldes auswählen. Es funktionieren die gleichen Schreibweisen für Bereiche wie bei *Python*-Listen.

`feld[zeile, :]` Auswahl einer Zeile (liefert einen Vektor)

`feld[:, spalte]` Auswahl einer Spalte (liefert einen Vektor)

`feld[zeilenIntervall, spaltenIntervall]` Teilmatrix des angegebenen Bereiches

Es ist sogar möglich, für einen der Indices eine *Python*-Liste anzugeben, der zweite Index muss dann allerdings ein Einzelwert oder ein zusammenhängender Bereich sein.

Beispiele.

```
# 5x6-Matrix
M = np.array([
    range(6),
    range(6, 12),
    range(12, 18),
    range(18, 24),
    range(24, 30)
])
# Erzeugte Matrix:
# 0  1  2  3  4  5
# 6  7  8  9 10 11
# 12 13 14 15 16 17
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
# 18 19 20 21 22 23
# 24 25 26 27 28 29

# Zeile auswählen
M[3, :] # → [18, 19, 20, 21, 22, 23, 24]

# Spalte auswählen
M[:, 3] # → [3, 9, 15, 21, 27]

# Zusammenhängende Teilmatrix auswählen (Zeile 0, 1, Spalten 2 bis 4)
M[:2, 2:5] # → [ [2, 3, 4], [8, 9, 10] ]

# Auswahl mit Liste
M[[1, 3], 1:4] # → [ [7, 8, 9] [19, 20, 21]

# Auswahl mit zwei Listen (funktioniert (so) nicht)
M[[1, 3], [0, 2, 4]] # → Fehlermeldung

# Lösung über Hilfsfeld
mm = M[[1, 3], :]
mm[:, [0, 2, 4]] # → [ [6, 8, 10], [18, 20, 22] ]
```

Interessant ist oft auch eine Auswahl mit einer Maske. Wir übergeben als Index ein Boolean-Feld mit gleicher Größe und Form wie die Matrix. Das Ergebnis ist eine Liste aller Werte des Feldes, an deren Position die Maske den Wert True hatte.

Beispiel: Ersetze alle geraden Zahlen durch `np.nan` (*not a number*) und anschließend diese durch 0.

```
# 10×10-Feld, reshape() wird unten beschrieben
m = nd.array(range(100), dtype='float32').reshape(10, 10)
# Anzeige der geraden Zahlen
print(m[m % 2 == 0])
# Ersetzen durch NaN
m[m % 2 == 0] = np.nan
# Ersetzen der Fehlwerte durch 0 (direkter Vergleich mit np.nan ist nicht möglich)
m[np.isnan(m)] = 0
```

Wir können Operationen nicht nur auf einzelne Feldelemente, sondern auch auf ganze Zeilen, Spalten oder das ganze Feld anwenden. Die einfachen Rechenoperationen wie `+` `-` `*` `/` `%` `==` `<` funktionieren dabei elementweise.

Verknüpfen wir dabei eine Matrix mit einem Skalar (also einer einfachen Zahl), wird die Zahl automatisch zu einer Matrix oder einem Vektor gleicher Größe expandiert. Das gleiche gilt auch für Matrix und Vektor: hier wird der Vektor als Zeilenvektor betrachtet und solange wiederholt, bis die Größe der Matrix erreicht ist.

Beispiele.

```
# Matrix mit doppelten Werten
MM = M * 2
# Erhöhe erste Zeile um 1
MM[0, :] = MM[0, :] + 1
# Verringere zweite Spalte um 1
MM[:, 1] = MM[:, 1] - 1
# Setze letzte Spalte auf 0
MM[:, -1] = 0
```

Beispiel: Skalieren aller Werte auf Summe 1, Prüfen der Summe der skalierten Werte (nur zur Demonstration, die Summe kann direkt mittels `np.sum()` berechnet werden).

```
accu = ... # wie oben
normX = X/accu # auch X /= accu möglich
# Summe prüfen
rows, cols = X.shape
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

accuVector = normX[0, :] # erste Zeile
for r in range(1, rows):
    accuVector += normX[r, :] # Zeile addieren
# Werte des accuVector addieren
accu = 0.0
for c in range(cols):
    accu += accuVector[c]
print(accu)

```

Notebook

Download: `simple-ops.ipynb`.

6.4 Änderung der Form

Wollen wir aus einem Vektor eine Matrix oder umgekehrt eine Matrix in einen Vektor überführen, können wir die Methode `reshape()` verwenden.

`matrixOderMatrix = vektorOderMatrix.reshape(dim1[, dim2 ...])`

Das Produkt der ursprünglichen Dimensionen muss dabei mit dem Produkt der neuen Dimensionen übereinstimmen.

Beispiel. Erzeugen einer 10×10-Matrix aus einer Liste der Zahlen von 0 bis 99, Umwandlung in eine Matrix mit 25 Zeilen und 4 Spalten und Rückverwandlung in einen Vektor.

```

m = np.array(range(100)).reshape(10, 10)
mm = m.reshape(25, 4)
v = m.reshape(100)

```

Eine der Dimensionen kann mit `-1` angegeben werden, sie wird dann automatisch berechnet. Wir können also auch schreiben:

```

m = np.array(range(100)).reshape(-1, 10)
mm = m.reshape(25, -1) # oder m.reshape(-1, 4)
v = m.reshape(-1)

```

Wollen wir Bilder in klassischen Lernverfahren bearbeiten, die üblicherweise nur mit Datenvektoren arbeiten, können wir die Bildpunktmatrix mit `reshape()` zu einem langen Vektor machen.

Wir müssen häufig Matrizen zusammensetzen, um z. B. neue Komponenten hinzuzufügen oder neue Datensätze zu erfassen. Dafür gibt es viele Möglichkeiten, wir betrachten `np.hstack()` für das horizontale und `np.vstack()` für das vertikale Zusammenfügen.

```

result = np.hstack(matrixliste)
# bzw.
result = np.vstack(matrixliste)

```

Für das horizontale Verbinden muss die Zeilenzahl, für das vertikale Verbinden die Spaltenzahl übereinstimmen. Wollen wir einen Vektor zu einer Matrix hinzufügen, wird dieser als Zeile betrachtet, `np.vstack()` kann direkt benutzt werden. Wollen wir ihn dagegen als Spalte hinzufügen, so müssen wir ihn in eine Matrix mit n Zeilen und einer Spalte konvertieren:

```

# matrix_n_m ist n×m-Matrix, vektor hat Dimension n
matrix_n_1 = vektor.reshape(-1, 1)
result = np.vstack((matrix_n_m, matrix_n_1))

```

Beispiel: Horizontales bzw. vertikales Verdoppeln einer Matrix m , Hinzufügen eines Vektors v als Zeile bzw. Spalte.

```
# 10x10-Matrix mit Werten 0...99
m = nd.array(range(100), dtype='float32').reshape(10, 10)
# Vektor mit Werten 0...9
v = nd.array(range(10), dtype='float32')
# Horizontal aneinanderfügen → 10x20-Matrix
mh = np.hstack((m, m)) # Matrizen müssen zur Liste geklammert werden
# Vertikal aneinanderfügen → 20x10-Matrix
mv = np.vstack((m, m)) # Matrizen müssen zur Liste geklammert werden
# Vektor als Zeile anhängen → 11x10-Matrix
m1 = np.vstack((m, v))
# Vektor als Zeile anhängen → 10x11-Matrix
m2 = np.hstack((m, v.reshape(-1,1)))
```

6.5 Spezielle Felder und Vektoren

Für spezielle Matrizen stehen zum einfachen Erzeugen spezielle Funktionen zur Verfügung. Alle Funktionen kennen zusätzlich den Parameter `dtype`.

np.eye(n) Einheitsmatrix der Größe $n \times n$ erzeugen.

np.zeros(dimensionstupel) Nullfeld mit gegebenen Dimensionen erzeugen, z. B. `np.zeros((4, 3))`.
Beim Erzeugen von Vektoren kann die Dimension als Einzelwert angegeben werden: `np.zeros(5)`

np.ones(dimensionstupel) Einsfeld mit gegebenen Dimensionen erzeugen, z. B. `np.ones((4,))`.

np.full(dimensionstupel, wert) Feld mit gegebenen Dimensionen und festem Wert erzeugen, z. B. `np.full((3, 3, 3), 42)`.

np.random.rand(dimensionsliste) Feld mit den angegebenen Dimensionen mit gleichverteilten Zufallszahlen aus dem Intervall $[0, 1)$ füllen. z. B. `np.random.rand(3, 3)`.

np.random.randint(minimum, maximum, dimensionsTupel) Feld der angegebenen Dimension mit ganzzahligen Zufallszahlen füllen, die zwischen Minimum (incl.) und Maximum (excl.) liegen. Die Dimensionen müssen hier als Tupel angegeben werden. Beispiel Würfelserie mit 5 Würfeln und 3 Würfeln: `np.random.randint(1, 7, (5, 3))`.

Wollen wir ein neues Feld gemäß einer Bedingung erzeugen, verwenden wir die Funktion `where()`. Sie erwartet ein Feld mit logischen Werten, einen Wert, wenn die Bedingung erfüllt, und einen Wert, wenn sie nicht erfüllt ist.

`neuesArray = np.where(boolArray, trueWert, falseWert)`

Beispiel: Wir wollen in einer Serie von Würfeln mit einem Würfel nur zwischen 6 und weniger als 6 unterscheiden und die Werte 1 bzw. 0 zuordnen.

```
dice = np.random.randint(1, 7, 25)
sixes = np.where(dice == 6, 1, 0)
```

Häufig müssen wir unsere Daten zufällig durchmischen oder eine zufällige Auswahl aus ihnen treffen, damit unsere Lernverfahren korrekt funktionieren. Für eine Matrix steht dafür die Funktion `random.shuffle()` zur Verfügung, die die Zeilen durchmischt:

`gemischt = np.random.shuffle(vektorOderMatrix)`

Bei Matrizen werden dabei die Zeilen (Achse 0), aber nicht die Spalten gemischt.

Oft ist es jedoch sinnvoller, nur den Index zu durchmischen und die Matrixzeilen mit diesem gemischten Index auszuwählen, da wir die gleiche Auswahl aus verschiedenen Matrizen bzw. Vektoren benötigen (beim Lernen benötigen wir zufällig ausgewählte Datenpunkte aus einer Matrix und deren Zielgrößen aus einem Vektor, die natürlich identisch ausgewählt werden müssen).

Dazu erzeugen wir einen durchmisches Indexfeld:

`idx = np.random.permutation(anzahl)`

Die (nach Zeilen) durchmischte Matrix bzw. Vektoren erhalten wir mit

```
matrix[idx, :]
vektor[idx]
```

Beispiel: Aufteilung einer *NumPy*-Datenmatrix *X* und des zugehörigen Zielvektors *y* (Dimension = Zeilenzahl von *X*) in Trainings- und Testdaten.

```
rowCount = X.shape[0]
idxSel = np.random.permutation(rowCount)
# 30% Trainingsdaten
splitSize = int(0.3 * rowCount)
idxTest = idxSel[:splitSize]
idxTrain = idxSel[splitSize:]
Xtest, Xtrain = X[idxTest, :], X[idxTrain, :]
ytest, ytrain = y[idxTest], y[idxTrain]
```

Wollen wir sicherstellen, dass wir immer die gleiche Auswahl erhalten (was für unsere Lernverfahren notwendig ist, wie wir später sehen werden), initialisieren wir unseren Zufallsgenerator mit einer festen Zahl:

```
np.random.seed(festerWert)
```

In vielen Tutorials und Büchern ist dieser Wert 42.

Wollen wir Diagramme darstellen oder Parameterbereiche nach optimalen Werten durchmustern, benötigen wir einheitlich unterteilte Intervalle. Dafür können wir entweder die Schrittweite oder die Zahl der Unterteilungen angeben.

```
folge1 = np.arange(start, endwert, schrittweite[, dtype='typ'])
folge2 = np.linspace(start, endwert[, anzahl], dtype='typ')
```

Bei der Funktion `arange()` ist der Endwert nicht Teil der Folge!

Für eine effektive Suche nach Hyperparametern müssen wir große Wertebereiche effizient durchsuchen. Es bietet sich hier häufig an, von einem Startwert ausgehend jedesmal den Wert zu verdoppeln oder zu verzehnfachen. Solche Folgen können einfach mit der Funktion `logspace()` erzeugt werden.

```
geomFolge = np.logspace(startExponent, endExponent[, anzahl], base=basis[, dtype='typ'])
```

Beispiel. Zweierpotenzen von 1 bis 2:sup:10 als ganze Zahlen.

```
np.logspace(0, 10, 11, base=2, dtype='int')
```

Notebook spezielle Felder und Vektoren

Download: `special-array.ipynb`.

6.6 Funktionen

Das *NumPy*-Paket stellt auch die mathematischen Funktionen so zur Verfügung, dass sie elementweise auf ganze Vektoren und Matrizen angewandt werden können. Einige wichtige Funktionen sind:

np.sqrt() Wurzelfunktion

np.exp() Exponentialfunktion bzgl. der Basis *e*.

np.log() Logarithmus bzgl. der Basis *e*.

np.abs() Absolutbetrag der Werte

np.around() Runden der Werte. Der zweite Parameter gibt die Anzahl der gewünschten Nachkommastellen an.
Beispiel: `np.around(vektor, 2)`.

np.sin(), np.cos(), ... Winkelfunktionen, Argument im Bogenmaß ($180^\circ = \pi$).

np.radians(), **np.degrees()** Umrechnung von Grad nach Radiant und von Radiant nach Grad

Beispiel (ohne konkrete Anwendung). Berechnung des Sinus für jedes Matrixelement und Ersetzen der 6. Spalte durch ihren Logarithmus.

Quellcode 6.1: numpy-ops.py: Einfache NumPy-Funktionen

```
#!/usr/bin/env python
# coding: utf-8

import numpy as np

# Erzeugen einer 5x5-Matrix mit Zufallszahlen zwischen 1 und 2
m = np.random.rand(5, 5)+1
m

# Erzeugen einer neuen Matrix mit dem Sinus aller Werte.
sinM = np.sin(m)
sinM

# Ersetzen der mittleren Spalte durch den Logarithmus und anschließend wieder
↳ durch die Exponentialfunktion.

print('Orig:\n', m)
m[:, 2] = np.log(m[:,2])
print('Log:\n', m)
m[:, 2] = np.exp(m[:,2])
print('Exp:\n', m)

# Erzeugen eines neuen Vektor mit der Wurzel der letzten Spalte.
roots = np.sqrt(m[:,-1])
roots

# Anhängen des Vektors als neue Zeile an die Matrix.

vm = np.vstack((m, roots))
print('Shape:', vm.shape)
print(vm)

# Anhängen einer neuen Spalte. Der Vektor muss in eine 5x1-Matrix umgewandelt
↳ werden.

roots5_1 = roots.reshape(-1, 1)
print('Shape der Spalte:', roots5_1.shape)
vh = np.hstack((m, roots5_1))
print('Shape:', vh.shape)
print(vh)
```

Download Notebook: `numpy-ops.ipynb`.

Hin und wieder benötigen wir auch eine Typkonvertierung, z. B. von `int` nach `float32` oder vom Kategorietyt (der von *Pandas* durch `pd.cut()` beim Export entsteht) nach `int`. Dies geschieht mit der Methode `astype()`:

```
neuesObjekt = feldOderMatrixastype('neuerTyp')
```

Der Typ wird wie schon beim Anlegen von Feldern und Vektoren als String angegeben.

Neben den elementweisen Funktionen finden wir auch Aggregatfunktionen, die die Werte einer Zeile oder Spalte oder auch der ganzen Matrix zusammenfassen.

np.min(), **np.max()** Minimum bzw. Maximum.

np.sum() Summe aller Werte.

np.median(), **np.mean()**, **np.var()**, **np.std()** Median, Mittelwert, Varianz und Standardabweichung.

np.cumsum(), **np.diff()** Kumulierte Summe, paarweise Differenz zwischen Nachbarn. Nur für Vektoren bzw. Matrixzeilen oder -spalten sinnvoll.

Diese Funktionen werden wie folgt genutzt:

```
np.aggreatFunction(matrix[, axis=achse][, keepdims=True])
```

Ohne Angaben wird das Aggregat auf die gesamte Matrix (bzw. den Vektor oder Tensor) berechnet. Mit der Achsenangabe 0 werden die Aggregate spaltenweise (also Zusammenfassung entlang der Zeilen), mit der Achse 1 zeileweise berechnet. Mit dem Parameter `keepdims=True` erhalten wir das Ergebnis als zweidimensionales Feld statt als Vektor und können es so mit dem Originalfeld verknüpfen.

Beispiel: Skalieren aller Spalten einer Matrix `m` auf das Intervall `[0 1]`.

```
colMin = np.min(m, axis=0, keepdims=1)
colMax = np.max(m, axis=0, keepdims=1)
normM = (m - colMin) / (colMax - colMin)
```

Hinweis: Zum Skalieren unserer Daten für das Machine Learning stellt uns das Paket *Scikit-learn* geeignete Funktionen zur Verfügung, die auf diesem Vorgehen aufbauen.

Statt des Minimums oder Maximums benötigen wir häufig den Index der Matrix oder des Vektors, an dem dieser Wert steht. Dazu stehen uns die Funktionen `np.argmin()` und `np.argmax()` zur Verfügung.

```
index = np.argmax(matrix[, axis=achse])
```

Berechnen wir das Minimum oder Maximum über die gesamte Matrix, erhalten wir nur einen einzelnen Wert, der dem Index der in einen Vektor umgewandelten Matrix entspricht. Um die Matrixposition zu erhalten, verwenden wir die Funktion `np.unravel_index()`:

```
matrixIndex = np.unravel_index(index, matrix.shape)
```

Zur Auswahl aller Zeilen oder Spalten, die eine bestimmte Bedingung erfüllen, verwenden wir `np.where()`, allerdings nur mit einem Parameter:

```
index, _ = np.where(bedingungsFeld)
```

Wir erhalten eine Liste der jeweiligen Indexlisten, mit denen wir auf die Feldelemente zugreifen können.

Beispiele:

```
# Alle Zeilenindices, für die die letzte Spalte > 1 ist
idx = np.where(m[:, -1] > 1)
# Auswahl dieser Teilmatrix
m[idx, :]
# Alle Matrix-Indices, bei denen der Wert kleiner 0 ist
idx = np.where(m < 0)
# Auswahl der Teilmatrix
m[idx]
```

Interessieren wir uns für alle Zeilen der Matrix, in denen der Wert in einer oder in allen Spalten die Bedingung erfüllt, wenden wir die Methoden `any()` oder `all()` auf unsere Bedingungsmatrix an:

```
bedingungsmatrix.any(axis=achse)
bedingungsmatrix.all(axis=achse)
```

Mit der Achse 0 werden die Spalten, mit 1 die Zeilen ausgewählt. Die Rückgabe ist eine Liste mit einem Element. Für die Auswahl aus der Matrix müssen wir vorher das erste Element auswählen.

Beispiele:

```
# Zeilen mit mindestens einem Wert > 1
idx = np.where((m > 1).any(axis=1))[0]
# Zeilenauswahl
m[idx, :]
# Spalten mit allen Werten < 0
idx = np.where((m < 0).all(axis=0))[0]
# Spaltenauswahl
m[:, idx]
```

Warnung: Die Methoden `any()` und `all()` kennen zwar einen Parameter `keepdims=True`, die Expansion des Ergebnisses erfolgt aber nicht vollständig, so dass der entstandene Index nicht zur Matrixauswahl verwendet werden kann.

Ebenso können wir mit `np.argsort` einen Index (einer Zeile oder Spalte) sortieren, was uns ermöglicht, eine Matrix nach einer bestimmten Zeile oder Spalte zu sortieren.

```
# Indices für Sortierung der Spalte berechnen
sortIndex = np.argsort(m[:, spaltenIndex])
# Sortierte Werte der Spalte
m[sortIndex, spaltenIndex]
# Sortierte Matrix nach dieser Spalte
sortM = m[sortIndex, :]
```

Notebook

Beispiele für weitere Aggregatfunktionen. Download: `numpy-aggregate.ipynb`.

6.7 Kategoriale Daten

Da auch in *NumPy*-Feldern kategoriale Daten verwaltet werden müssen, die nun allerdings als Zahlenwerte codiert sind, stehen uns auch hier entsprechende Funktionen zur Auswertung und Verarbeitung zur Verfügung.

Das Ermitteln der vorhandenen Kategorien und der Anzahl der Werte je Kategorie geschieht mit der Funktion `np.unique()`

```
np.unique(matrix [return_counts=True])
```

Ohne weiteren Parameter erhalten wir einen sortierten Vektor mit den Kategorien. Wird der Parameter `return_counts=True` angegeben, wird zusätzlich ein Feld in gleicher Ordnung zurückgegeben, das die Anzahl der Werte jeder Kategorie enthält. Weitere mögliche Parameter wollen wir hier nicht betrachten.

Wollen wir über beide Felder gleichzeitig z. B. in einer Schleife iterieren, benötigen wir die Funktion `np.nditer()` (`zip()` funktioniert hier nicht):

```
for val1, val2 [, val3 ...] in np.nditer((vektor1, vektor2 [, vektor3 ...])) :
    # Verarbeite val1, val2, ...
    ...
```

Beispiel: Auswertung eines Würfels mit 1000 Würfeln.

```
dices = np.random.randint(1, 7, 1000)
# Augenzahlen
print('Augenzahlen: ', np.unique(dices))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
# Augenzahlen und Würfe für jede Augenzahl
cat, catCnt = np.unique
# Ausgabe von Augenzahl und Anzahl der Würfe mit dieser Zahl
for val, cnt = np.nditer((cat, catCnt)):
    print('Augenzahl: {} Anzahl: {}'.format(val, cnt))
```

Umgekehrt können wir mit der Funktion `np.digitize()` auch stetige Werte in Kategorien einteilen.

```
labels = np.digitize(vektor, rechteGrenzen)
```

Es entsteht ein Vektor der gleichen Länge. Jedem Wert wird dabei der Index seiner rechten Intervallgrenze zugeordnet. Werte, die größer als die letzte Grenze sind, werden dem letzten Intervall zugeordnet, es entstehen also keine NA-Werte.

Beispiel: Einteilen eines Vektors mit Werten zwischen 0 und 100 in 5 Klassen.

```
boundaries = np.arange(20, 120, 20) # oder np.linspace(20, 100, 5)
labels = np.digitize(values, boundaries)
# Originalwerte, die zwischen 20 und 40 liegen = Label 1
val0 = values[labels==1]
```

Notebooks

Erzeugen eines Vektors von Zufallszahlen, Zerlegen in Kategorien, Auswertung: `random-digitize.ipynb`.

Datensatz *Abalone*: Skalieren der Werte, Kategorisieren der Zielgröße. `abalone-scale-binning.ipynb`.

6.8 Lineare Algebra

Versuchen wir, eine Matrix mit einem Vektor zu multiplizieren, erhalten wir merkwürdige Ergebnisse.

```
X = np.array(range(25)).reshape(5, 5)
y = np.array([1, 2, 3, 4, 5])
print(X * y)
```

Ergebnis:

```
[ [ 0  2  6 12 20]
  [ 5 12 21 32 45]
  [10 22 36 52 70]
  [15 32 51 72 95]
  [20 42 66 92 120]]
```

Der Grund liegt in der elementweisen Ausführung der Operationen. Stoßen dabei Matrizen und Vektoren aneinander, der Vektor zunächst zu einer $1 \times m$ -Matrix umgewandelt (also eine Matrix mit einer Zeile). Anschließend wird die Zeile so oft wiederholt, bis die Zeilenzahl der Matrix erreicht ist, ehe die angegebene Operation elementweise ausgeführt wird. Dieses Verfahren wird *Broadcasting* genannt.

Wir wollen hier jedoch das sogenannte *innere Produkt* berechnen. Dazu benutzen wir die Funktion `dot()` bzw. in neueren *Python*-Versionen den Operator `@`.

```
spaltenVektor = matrix @ spaltenVektor
zeilenVektor = zeilenVektor @ matrix
skalarProdukt = vektor @ vektor
```

Zwischen Zeilen- und Spaltenvektor wird dabei nur über die Position des Operanden im Produkt entschieden. Steht der Vektor vor der Matrix, werden die Einzelprodukte und die Summe über den ersten Index, also die Zeilen, steht der Vektor nach der Matrix, werden sie über den letzten Index, also die Spalten, gebildet. Stoßen also in einem inneren

Produkt zwei Matrizen aneinander, werden der letzte Index (Spalten) der ersten Matrix mit dem ersten Index der zweiten Matrix (Zeilen) verknüpft – wie wir das bei einer Matrixmultiplikation erwarten.

Mit Hilfe dieser und weiterer Matrixoperationen können wir in *NumPy* auch *lineare Gleichungssysteme* lösen. Wir formulieren diese in Matrixschreibweise $A \cdot x = b$. Die Lösung erhalten wir mit Hilfe der inversen Matrix $x = A^{-1} \cdot b$.

```
Ainv = np.linalg.inv(matrix)
x = Ainv @ rechteSeite
```

Effizienter ist jedoch die Anwendung eines Lösungsverfahrens, was die aufwendige Berechnung der inversen Matrix vermeidet.

```
x = np.linalg.solve(matrix, rechteSeite)
```

Beispiel: Bei einer Nerd-Party trinkt jeder Mathematikstudent 3 Flaschen Bier und 2 Glas Wein und jeder Wirtschaftsinformatiker 2 Flaschen Bier und 4 Glas Wein. Insgesamt werden 31 Flaschen Bier und 34 Glas Wein ver-
nichtet. Wieviele Mathematiker und wieviele Wirtschaftsinformatiker waren anwesend?

$$\begin{aligned} 3m + 2w &= 31 & (\text{Bier}) \\ 2m + 4w &= 34 & (\text{Wein}) \end{aligned}$$

Python-Code:

```
A = np.array([[3, 2],
              [2, 4]])
b = np.array([31, 34])
# Lösung über Inverse
Ainv = np.linalg.inv(A)
x = Ainv @ b
#Probe
bProbe = A @ x

# Lösung über Solver
x = np.linalg.solve(A, b)
```

Weitere Funktionen wie die Berechnung von Determinanten, Eigenwerten und Eigenvektoren werden von *NumPy* ebenfalls bereitgestellt. Komplexere Algorithmen stellt die auf *NumPy* aufsetzende Bibliothek *SciPy* zur Verfügung, die wir in unseren Lernverfahren im Hintergrund mit benutzen werden, ohne sie explizit anzusprechen.

Notebook

Lösen eines Gleichungssystems, Bestimmen der Eigenwerte der Koeffizientenmatrix. Download: `linalg1.ipynb`.

6.9 Meshgrid

Um Funktionen zweier Veränderlicher darzustellen, müssen wir die Funktionswerte an jedem Punkt eines Rasters erstellen. Ein solches Raster kann leicht mit der Funktion `meshgrid()` erzeugt werden.

```
xx, yy = np.meshgrid(xVektor, yVektor)
```

Es entstehen zwei Matrizen, wobei die erste die x-Werte und die zweite die y-Werte an der jeweiligen Stelle des Gitters enthält. Wir können nun den Funktionswert an jeder Stelle berechnen und darstellen.

Beispiel: Berechnen der Funktionswerte $x^2 + y^2$ auf einem Gebiet $[-5, 5] \times [-5, 5]$.

```
# Rasterteilungen für x und y (je 6 Werte)
x = np.linspace(-5, 5, 6)
y = np.linspace(-5, 5, 6)
# Gitter
xx, yy = np.meshgrid(x, y)
print("x-Gitter")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
print(xx)
print("y-Gitter")
print(yy)
# Berechnen der Funktionswerte
zz = xx*xx + yy*yy
print("Funktionswerte")
print(zz)
```

Zur graphischen Veranschaulichung verwenden wir die (im folgenden besprochene) Bibliothek *Matplotlib*.

```
import matplotlib.pyplot as plt

plt.matshow(zz)
plt.colorbar()
plt.show()
```

Notebook

Demonstration der Nutzung eines Meshgrids. Download: `meshgrid.ipynb`.

6.10 Aufgaben

Aufgabe 6.1 Umwandlung Data Frame

Erzeugen Sie aus dem Data Frame Ihres gewählten Beispiels ein *NumPy*-Array mit allen Komponenten sowie getrennt einen *NumPy*-Vektor für die Zielgröße. Überprüfen Sie Dimensionen, Datentyp und Inhalt der Daten.

Supervised Learning

- Modelle
- Verfahrensauswahl
- Trainings- und Testdaten
- Einführendes Beispiel
- Ablauf
- Under- und Overfitting
- Aufgaben

7.1 Modelle

Das Ziel von *Supervised Learning*-Algorithmen ist das Treffen von Vorhersagen anhand vorhandener Daten der Vergangenheit. Für diese Daten ist das Ergebnis (Zielgröße) bekannt:

- Kategorie (SPAM oder HAM, kauft oder kauft nicht, Schulart nach der Grundschule, ...)
- Stetige Größe (Wert eines Grundstücks, Quartalsverkaufszahl)

Ist die Zielgröße eine Kategorie (endliche Anzahl von Möglichkeiten), spricht man von *Klassifikation*, ist sie eine stetige Größe, handelt es sich um ein *Regressionsproblem*. Ein häufiger Spezialfall ist die *binäre Klassifikation*, bei der die Zielgröße genau zwei Werte haben kann – eine bestimmte Eigenschaft ist erfüllt oder nicht.

Zum Ableiten der Vorhersage werden für jedes Datum verschiedene Eigenschaften ausgewertet, von denen wir vermuten, dass sie Einfluss auf die Zielgröße haben (Alter, Geschlecht, Beruf, Einkommen einer Person; Maße verschiedener Pflanzenteile zur Bestimmung der Pflanzenart; Grundstücksgröße, Zimmerzahl, Abstand zum Meer zur Bestimmung eines Grundstückspreises; ...). Diese Werte müssen meist in Zahlen codiert und für jeden Datenpunkt zu einem Vektor zusammengefasst werden. Unsere gesamten Trainingsdaten stellen wir als Matrix dar, wobei jede Zeile einen Datenpunkt und jede Spalte eine Komponente (Feature) repräsentiert. Die Zielgröße wird als eigener Vektor dargestellt, die Komponenten sind die jeweiligen Zielgrößen der Daten.

Das gesuchte Modell ist eine Berechnungsvorschrift oder Abbildung, die aus den Komponenten der Daten die Zielgröße möglichst korrekt vorhersagt. Unser Ziel ist es, dieses Modell aus den bekannten Trainingsdaten automatisch

zu lernen. Das Modell soll dabei möglichst gut *generalisieren*, d. h. die inneren Zusammenhänge abbilden und damit gute Vorhersagen treffen, und nicht einfach die Daten auswendig lernen.

Wichtig für das Erstellen eines guten Modells ist eine ausreichende Zahl von Trainingsdaten, in denen möglichst viele Merkmalsausprägungen vorhanden sind. Wie groß diese Zahl sein muss, hängt stark vom konkreten Problem und der Güte der Daten ab. Allgemein kann man sagen, dass mit größerer Merkmalszahl auch mehr Daten erforderlich sind. So genügen beim *UCI-Iris-Datensatz* mit 4 Merkmalen und 3 Klassen bereits 100 Datensätze, während eine moderne Bilderkennung mit einigen 100000 Bildpunkten (Merkmalen) und 100 Klassen einige Millionen Trainingsdaten benötigt.

7.2 Verfahrensauswahl

Wir werden für die Untersuchung der klassischen Machine-Learning-Verfahren die Python-Bibliothek *scikit-learn* [[sklearn](#)] verwenden, die als Modul unter dem Namen `sklearn` zur Verfügung steht.

Je nach Aufgabe, vorhandener Datenmenge und Art der Zielgröße eignen sich verschiedene Verfahren am besten. Eine erste Empfehlung gibt die Dokumentation zu *scikit-learn*: [Choosing the right estimator](#). Einige Verfahren und ihre Güte werden wir im Kurs genauer betrachten.

Diese Übersicht keine *ANN*, sondern nur klassische Lernverfahren, die für viele Probleme jedoch völlig ausreichend sind.

Hinweis: Das *No-Free-Lunch-Theorem* besagt, dass es kein allgemein bestes Klassifikationsverfahren gibt, sondern je nach Art der Daten ein geeignetes Verfahren gewählt werden muss.

Umgekehrt wurde mehrfach empirisch belegt, dass mit wachsender Zahl der Trainingsdaten die Güte aller *ML*-Verfahren zunimmt.

7.3 Trainings- und Testdaten

Wenn wir alle unsere Daten, für die das Ergebnis bereits bekannt ist, dafür verwenden, um das Modell zu erstellen, kann es passieren, dass das Modell einfach diese Daten «auswendig lernt» und für neue Daten, die davon abweichen, keinerlei sinnvolle Vorhersage treffen kann. Das Modell soll statt dessen die inneren Zusammenhänge zwischen Daten und Zielgröße lernen, um diese auf neue Daten anwenden zu können.

Um sicherzustellen, dass eine solche Generalisierung tatsächlich stattfindet, teilen wir unsere vorhandenen Daten in eine Trainings- und eine Testmenge auf. Unser Modell wird allein anhand der Trainingsmenge erzeugt. Anschließend prüfen wir die Modellqualität anhand der zurückgehaltenen Testdaten, indem wir die Modellvorhersage mit dem tatsächlichen Ergebnis vergleichen.

Warnung: Die Testdaten müssen einmalig ausgewählt werden und dürfen niemals in die Modellerstellung einfließen, um dieses «Auswendiglernen» zu vermeiden.

Hinweis: Für die Testmenge verwendet man üblicherweise 20 bis 30% der vorhandenen Daten. Wichtig ist dabei, dass die Daten in Trainings- und Testmenge gut durchmischt sind, um möglichst alle Merkmale und Zielgrößen ins Modell und dessen Bewertung einzubeziehen.

Viele Modelle werden durch *Hyperparameter* gesteuert, die nicht direkt berechnet werden können, aber Einfluss auf die Modellqualität haben und geeignet manuell bestimmt werden müssen. Um deren Eignung zu prüfen, spaltet man häufig von der Trainingsmenge eine *Validierungsmenge* ab. Für jeden Wert des Hyperparameters wird das Modell mit dieser kleineren Trainingsmenge erstellt und die Güte mit der Validierungsmenge geprüft. Ist man mit der Wahl dieser Parameter zufrieden, wird nun das Modell nochmals mit der gesamten Trainingsmenge berechnet.

7.4 Einführendes Beispiel

Wir wollen weiterhin unseren Datensatz *Abalone* aus dem *UCI Machine Learning Repository* [uci] verwenden und Modelle erstellen, die die Zahl der Ringe aus den sonstigen Eigenschaften der Muscheln abzuleiten versuchen.

Wir benötigen hier folgende Module:

numpy (importiert mit Alias **np**) Matrixdarstellung der Daten.

pandas (importiert mit Alias **pd**) Einlesen der *CSV*-formatierten Datendatei, erste Analysen.

sklearn.tree Entscheidungsbaum-Klasse für die Modellbildung.

sklearn.metrics Werkzeuge zur Modellbewertung.

Um unser Notebook mit den Hilfsfunktionen zum Einlesen und zur Klasseneinteilung direkt importieren zu können, benötigen wir noch das Modul `import_ipynb`. Alternativ können wir dieses auch in ein einfaches Python-Script exportieren und dann direkt einbinden:

```
# Ins Notebook-Verzeichnis wechseln und Anaconda vorher aktivieren
jupyter nbconvert abalone_helper.ipynb --to python
```

Es entsteht Python-Script `abalone_helper.py`, das nun direkt importiert werden kann:

```
import abalone_helper
```

7.5 Ablauf

Wir gehen schrittweise wie folgt vor.

7.5.1 Bereitstellen und Einlesen der Daten

Wir legen die beiden Dateien `abalone-train.csv` und `abalone-test.csv` (s. *Auswahl einer Stichprobe*) sowie unser Script mit den Hilfsfunktionen `abalone_helper.ipynb` in unserem Arbeitsverzeichnis ab, in dem wir anschließend unser *Jupyter* Notebook anlegen.

Zunächst implementieren wir alle notwendigen Importe:

```
import pandas as pd
import numpy as np
from sklearn import tree, metrics
# Nur für direkten Notebook-Import erforderlich
import import_ipynb
# Hilfsmodul als Notebook oder Python-Script
import abalone_helper
```

Sowohl Trainings- als auch Testdaten werden nun eingelesen.

```
# alternativ ... = pd.read_csv(...)
abaloneTrain = abalone_helper.readFile('abalone-train.csv')
abaloneTest = abalone_helper.readFile('abalone-test.csv')
# kurze Datenkontrolle
abaloneTrain.info()
```

Hier sind bereits die kategorialen Daten als Zahlen codiert. Da wir uns zunächst mit einem Klassifikationsproblem beschäftigen wollen, wandeln wir die Zielgröße, die Zahl der Ringe, in drei Klassen (0 = jung, 1 = mittel, 2 = alt) um. Hier verwenden wir unsere Hilfsfunktion `toLabels()`, um sicherzustellen, dass wir immer die gleichen Grenzen verwenden, was uns den Vergleich zwischen verschiedenen Modellen erleichtert.

```
labeledTrain = abalone_helper.toLabels(abaloneTrain, False)
labeledTest = abalone_helper.toLabels(abaloneTest, False)
```

Wir können die korrekte Umwandlung mit der Methode `value_counts()` leicht prüfen:

```
labeledTrain.labels.value_counts(sort=False)
```

7.5.2 Umwandeln in NumPy-Feld

Unsere Lernverfahren benötigen die Eigenschaften und Labels unserer zu klassifizierenden Daten als *NumPy*-Datenstrukturen. Jedes Objekt wird durch eine Zeile einer Matrix repräsentiert, die Spalten beschreiben die einzelnen Eigenschaften (Features). Die Zielgröße wird durch einen einzelnen Vektor dargestellt, dessen Einträge die bekannte Klasse bzw. den Zielwert des jeweiligen Objekts enthalten.

Wir trennen also vom Data Frame die Zielgröße ab und erzeugen mit der Eigenschaft `values` die *NumPy*-Matrix bzw. den -Zielvektor.

```
Xtrain = labeledTrain.drop('labels', axis=1).values
Xtest = labeledTest.drop('labels', axis=1).values
yTrain = labeledTrain.labels.values
yTest = labeledTrain.labels.values
```

Auch diese Daten sollten wir durch Anzeige der Dimension und einiger Datensätze vor der weiteren Verarbeitung kontrollieren.

7.5.3 Klassifikationsobjekt erzeugen und trainieren

Aus der Lehrveranstaltung *Data Mining* ist einigen das Verfahren des *Entscheidungsbaumes* bekannt. Hier wird in jedem Schritt aus den Daten die Komponente ausgewählt, nach der sich die Daten am »besten« in die gewünschten Klassen aufteilen lassen und diese Bedingung als Verzweigungsknoten in einem (Binär-)Baum abgelegt. Für jede der beiden Teilmengen der Trainingsmengen wird dieser Vorgang wiederholt, bis die Menge so weit in Teilmengen zerlegt ist, dass jede Menge nur noch Objekte der gleichen Klasse enthält.

Dieses Verfahren (mit verschiedenen, konfigurierbaren Modifikationen) ist in *scikit-learn* implementiert, wir wollen es auf unsere Daten anwenden.

Das *scikit-learn* bietet für alle derartige Klassifikationen eine einheitliche Schnittstelle:

1. Erzeugen eines Klassifikationsobjektes
2. Training des Objektes mit der Methode `fit()`.
3. Bewerten der Modellqualität mit der Methode `score()`.
4. Vorhersagen des Labels oder Zielwertes für eine Objektmenge mit der Methode `predict()`.

Für einen Entscheidungsbaum erzeugen wir ein Objekt der Klasse `tree.DecisionTreeClassifier`.

```
classifier = tree.DecisionTreeClassifier()
```

Diesen trainieren wir nun mit der Methode `fit()` und unseren Trainingsdaten: der Feature-Matrix und den Labels.

```
classifier.fit(Xtrain, yTrain)
```

Dieser Methodenruf gibt das Klassifikationsobjekt selbst zurück, was eine einfache Verkettung von weiteren Methoden erlaubt.

7.5.4 Bewerten des Modells

Die Methode `score()` erlaubt eine einfache Bewertung unserer Modellqualität. Sie benötigt eine Feature-Matrix und einen Vektor der korrekten Zielgrößen der Datensätze als Parameter und liefert uns den Anteil der korrekt vorhergesagten Daten an der Gesamtanzahl. Wir sollten diese Methode sowohl auf unsere Trainings- als auch auf unsere Testdaten anwenden.

```
scoreTrain = classifier.score(Xtrain, yTrain)
scoreTest  = classifier.score(Xtest, ytest)
```

Im konkreten Beispiel erhalten wir auf den Trainingsdaten eine Genauigkeit von 1.0, auf den Testdaten jedoch nur von 0.78. Dies ist ein deutliches Zeichen von *Overfitting*: unser Modell hat die Trainingsdaten »auswendig« gelernt, statt aus ihnen die tatsächlichen Abhängigkeiten zur Zielgröße zu ermitteln.

7.5.5 Vorhersagen treffen

Unser Ziel ist es ja in der Praxis, mit unserem Modell tatsächlich Vorhersagen zu treffen. Wir ordnen dazu die zu untersuchenden Daten in einer Matrix mit gleichem Aufbau wie die Trainingsdaten an und nutzen die Methode `predict()` unseres Modellobjektes:

```
yPredict = classifier.predict(Xvorhersage)
```

Wir erhalten einen Vektor, der für jeden übergebenen Datensatz das vorhergesagte Label enthält. Hier können wir natürlich auch unsere Trainings- oder Testdaten übergeben und den vorhergesagten Vektor mit den tatsächlichen Daten vergleichen. Dies kann z. B. durch Berechnung des Differenzvektors und den tatsächlichen Werten geschehen, der bei allen Übereinstimmungen eine Null und sonst einen davon abweichenden Wert enthält. Wir zählen nun die Werte ungleich Null mit der Funktion `np.count_nonzero()` und vergleichen mit der Zahl der Daten:

```
# Testlabel vorhersagen
predTest = classifier.predict(Xtest)
diffTest = predTest - yTest
# Anzahl fehlerhafte Vorhersagen
countErrTest = np.count_nonzero(diffTest)
# Anzahl korrekte Vorhersagen
countOkTest = yTest.shape[0] - countErrTest
# relativer Anteil korrekte Vorhersagen
score = countOkTest/yTest.shape[0]
```

7.5.6 Confusion Matrix

Liegen uns die Vektoren für die tatsächlichen Werte und die Vorhersage vor, können wir sehr einfach eine genauere Analyse mit Hilfe der *Confusion Matrix* vornehmen:

```
confTrain = metrics.confusion_matrix(yTrain, predTrain)
confTest  = metrics.confusion_matrix(yTest, predTest)
```

Wir übergeben als ersten Parameter die tatsächlichen, als zweiten Parameter die Daten der Modellvorhersage und erhalten eine Matrix, deren Zeilen die Daten mit dem korrekten Label repräsentieren. In den jeweiligen Spalten stehen die Zahlen, wie oft welches Label vorhergesagt wurde.

Für unseren *Abalone*-Datensatz erhalten wir z. B. für die Trainingsdaten:

```
[[ 536    0    0]
 [   0 2325    0]
 [   0    0   63]]
```

und für die Testdaten:

```
[[204  99   0]
 [ 83 758  36]
 [  1  62 10]]
```

Im Idealfall stehen in der Matrix außer auf der Diagonale nur Nullen, dann wurden alle Daten korrekt vorhergesagt, wie das in unserem Beispiel für die Trainingsdaten der Fall ist. Bei den Testdaten, mit denen das Modell nicht trainiert wird, wird z. B. das Label 0 (erste Zeile) in 204 Fällen (1. Spalte) korrekt, in 99 Fällen wird das Label 1 und in keinem Fall das Label 2 vorhergesagt.

Wir erkennen im Beispiel recht gut, dass die Vorhersagen für die Klassen 0 und 1 recht gut, für die Klasse 2 jedoch zum großen Teil falsch sind.

Unser Ziel wird nun sein, Modelle zu finden, die aus den Trainingsdaten besser generalisierende Merkmale extrahieren, die uns auch bessere Vorhersagen erlauben. Folgende Vorgehensweisen sind hier denkbar:

- Beschaffung von weiteren Trainingsdaten, falls möglich. Allgemein kann man sagen, dass sich unabhängig vom konkreten Verfahren die Modellqualität mit der Zahl der Trainingsdaten verbessert.
- Modifikation der *Hyperparameter*. Viele Modelle (wie auch unser verwendeter Entscheidungsbaum) können über weitere Parameter angepasst werden.
- Verwendung eines anderen Klassifikationsverfahrens. Je nach konkretem Datensatz (Umfang der Daten, Zahl und Typ der Features, statistische Eigenschaften der Daten) können sich unterschiedliche Verfahren des Machine Learning sinnvoll sein.

7.5.7 Notebook

Eine Zusammenfassung aller obigen Schritte findet sich im Notebook `abalone-simple-classification.ipynb`.

7.6 Under- und Overfitting

Unser Ziel ist es, ein möglichst gut generalisierendes Modell zu finden, das die Gemeinsamkeiten der Komponentenwerte für die Auswahl einer Klasse erkennt.

Da das Modell anhand der Trainingsdaten erstellt wird, ist das einfachste Vorgehen, einfach die Trainingsdaten «auswendig» zu lernen. Versuchen wir nun allerdings, neue Daten zu klassifizieren, werden wir sehr schlechte Ergebnisse erhalten, da diese nur mehr oder weniger zufällig eingeordnet werden. Dieser Effekt kann bei den meisten Algorithmen auftreten und wird als *Overfitting* bezeichnet.

Wir erkennen ein Overfitting meist daran, dass es nahezu perfekte Vorhersagen für die Trainingsdaten, aber sehr schlechte für die nicht im Lernprozess benutzten Testdaten liefert.

Umgekehrt kann unser Modell auch zu «einfach» sein und zu wenig Informationen aus den Trainingsdaten verarbeiten, so dass unser Modell generell schlechte Vorhersagen trifft. Wir sprechen dann von *Underfitting*.

Wir erkennen ein Underfitting daran, dass die Vorhersagegenauigkeit sowohl auf den Trainings- als auch auf den Testdaten schlecht ist.

Um ein gut angepasstes Modell zu erhalten, müssen wir neben der Auswahl eines geeigneten Verfahrens meist auch die *Hyperparameter* des Verfahrens geeignet wählen. Im obigen Beispiel haben wir keinerlei Parameter für den Entscheidungsbaum angegeben, ein Blick in die Dokumentation von *scikit-learn* verrät uns jedoch, dass wir beim Erzeugen eines Entscheidungsbaum-Objekts hier eine Vielzahl von Parametern zur Feinkonfiguration wählen können. So kann z. B. die Baumtiefe oder die minimale Größe der Blattknoten beschränkt werden, um eine bessere Generalisierung zu erhalten. Darauf kommen wir später zurück.

Wie man geeignete Hyperparameter findet, wollen wir später betrachten. Zunächst werden wir bei der Untersuchung der Verfahren mit den von *scikit-learn* vordefinierten Standardwerten oder andere uns plausible Werte verwenden.

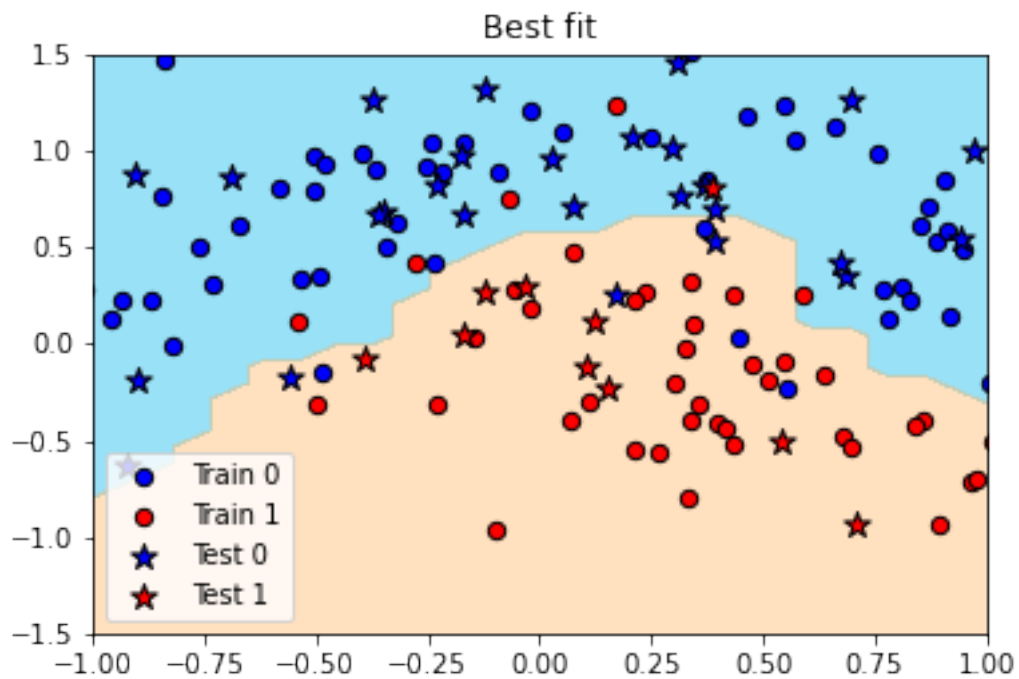


Abb. 7.1: Gute Generalisierung

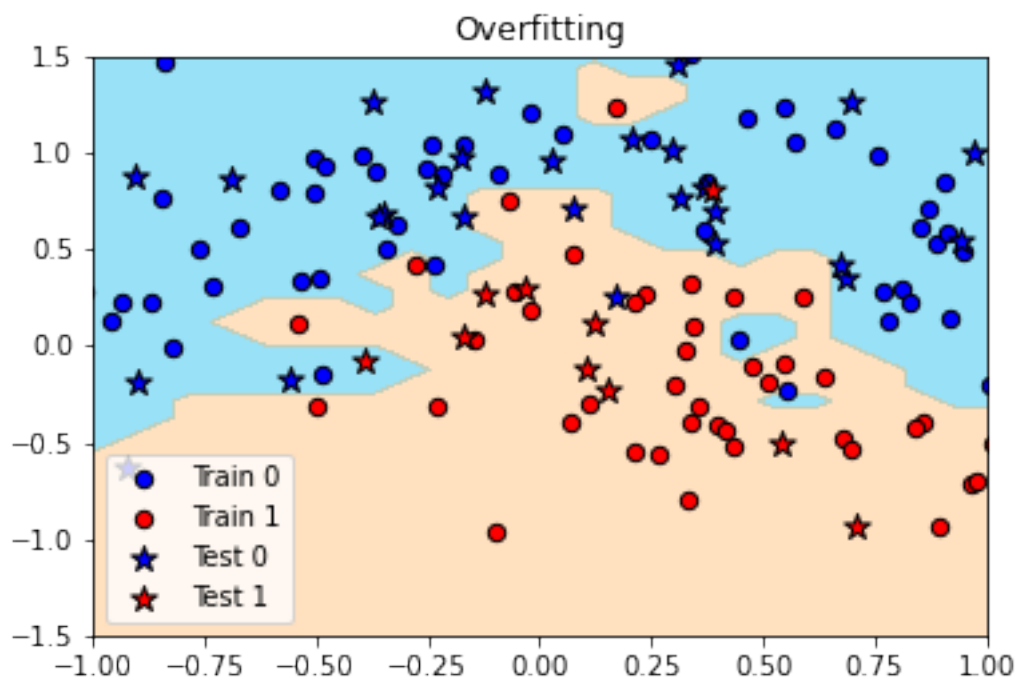


Abb. 7.2: Overfitting, die Entscheidungsgrenze ist stark zerklüftet.

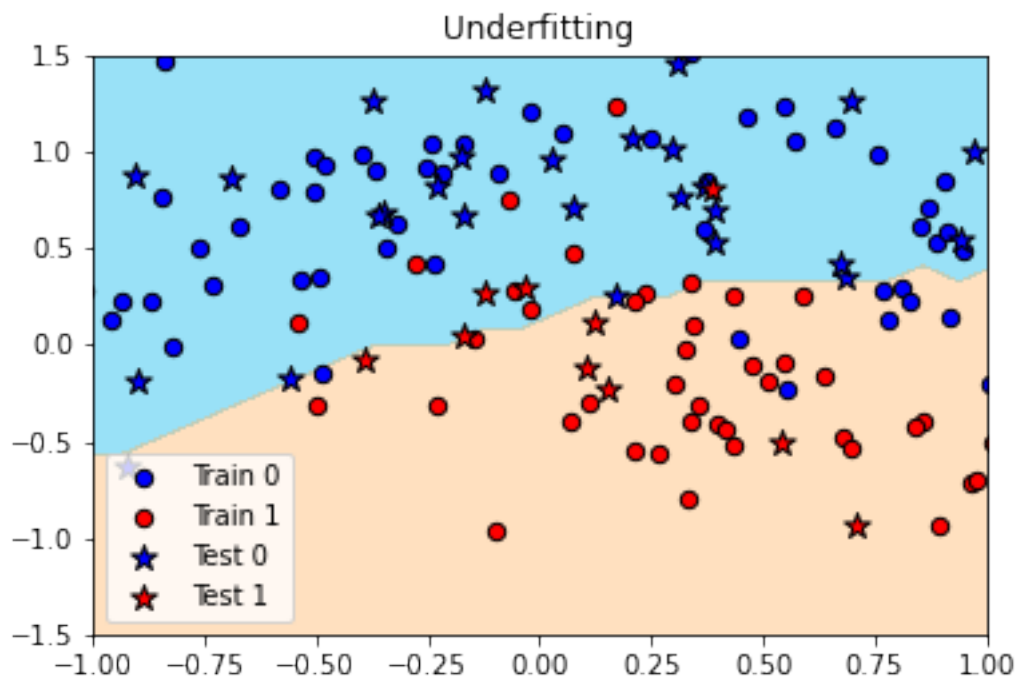


Abb. 7.3: Underfitting, die Entscheidungsgrenze ist fast eine Gerade.

Notebook

Notebook zum Erzeugen obiger Graphiken (zur Information für Interessierte). Es wurden ein künstlicher Datensatz *moons* aus dem *scikit-learn*-Paket und ein *KNN*-Modell mit verschiedenen Hyperparametern gewählt): `over_underfitting.ipynb`.

7.7 Aufgaben

Aufgabe 7.1 Auswahl und Analyse eines Trainingsdatensatzes, erste Klassifikation, Hausaufgabe (8 Punkte)

Analysieren Sie Ihren ausgewählten Datensatz mit den bisher kennengelernten Werkzeugen von Pandas bzw. NumPy sowohl numerisch als auch graphisch.

Bearbeiten Sie folgende Teilaufgaben:

- Einlesen des Datensatzes mit der *scikit-learn*-Methode bzw. *Pandas*.
- Umwandlung der Features in einen Pandas Data Frame mit Beschriftung der Komponentenspalten.
- Umwandlung der Zielgröße (Klassen- bzw. Labelvektor) in einen ganzzahligen *NumPy*-Vektor.
- Ausgabe der Instanzen- und Featurezahl sowie der Datentypen.
- Umwandlung der kategorialen Features in entsprechende Zahlenwerte mit fester Zuordnung zwischen Kategorie und verwendetem Zahlencode.
- Anzeige der ersten 13 Datenpunkte und der zugehörigen Label.
- Durchmischen des Datensatzes, Auftrennen in Trainings- und Testdaten und getrenntes Speichern in *CSV*-Dateien für die weitere Nutzung.

Die Trainingsmenge sollte dabei etwa 80% der Daten, jedoch maximal 5000 Datensätze umfassen.

- Statistische Informationen über alle numerischen und alle kategorialen Spalten (getrennte Ausgabe). Es sollen dabei die Objektzahlen für alle Kategorien jeder Nominalspalte angezeigt werden (nicht nur für die häufigste Kategorie).
- Paarweise Korrelation zwischen allen numerischen Spalten. Zwischen welchen Spalten besteht eine starke stochastische Abhängigkeit?
- Graphische Analyse der Verteilungen aller numerischen Komponenten.
- Auswahl eines Teildatensatzes mit 10%, aber maximal 500 Datensätzen, aller Daten.
- Scatter Matrix des Teildatensatzes mit Einfärbung der Datenpunkte entsprechend der Zielklasse.
- Scatter Plot der beiden Features mit der betragsmäßig höchsten Korrelation zur Zielgröße und Einfärben der Datenpunkte entsprechend der Zielklasse. Die höchste Korrelation ist mit dem Gesamtdatensatz zu ermitteln, der Plot soll wiederum für den Teildatensatz erstellt werden.
- Getrenntes Einlesen der Trainings- und Testdatei und Umwandlung in NumPy-Datenstrukturen.
- Training eines Decision Tree Classifiers (ohne Parameter) und Bestimmen der Güte (Score) auf den Trainings- und Testdaten.

Sämtliche Ergebnisse jeweils unterhalb der Auswertung innerhalb von *Markdown*-Zellen zu interpretieren.

Hinweis: Die Abgabe der Lösung umfasst das Jupyter Notebook und die benutzte Datendatei. Die Quelle der Daten ist im Notebook zu dokumentieren.

Beispieldatensätze

- *Mitgelieferte Daten*
- *Synthetische Datensätze*
- *Datenquellen*
- *openml.org*
- *Kaggle, UCI etc.*
- *Aufgaben*

Zum Erlernen und Bewerten der Verfahren des *Machine Learning* benötigen wir passende Trainingsdatensätze. Beliebte und gut dokumentierte Datensätze sind häufig schon gut aufbereitet in die Bibliotheken eingebaut. Viele weitere Datensätze finden sich im Internet.

8.1 Mitgelieferte Daten

Die Bibliothek *scikit-learn* enthält zum Erlernen der Verfahren sogenannte *Toy Datasets*, aufbereitete kleinere Sammlungen von Daten, die einfach verwendet werden können. Zur Nutzung importieren wir das Modul `datasets`.

```
from sklearn import datasets
dataset = datasets.load_name()
```

Wir erhalten ein Dictionary, das üblicherweise Einträge mit folgenden Schlüsseln enthält:

data Die Matrix der Komponenten (Features). Die Datensätze sind zeilenweise organisiert.

target Der Vektor der Zielgröße (Klassennummer oder stetige Größe).

DESCR Beschreibung des Datensatzes.

feature_names Die Namen der Komponenten bzw. Spalten der Datenmatrix.

target_names Die Namen der Klassen bei Klassifizierungsproblemen, die in den Daten als ganze Zahlen codiert sind.

Wollen wir lediglich das Verhalten der Machine-Learning-Modelle untersuchen und keine unbekannten Daten klassifizieren, benötigen wir normalerweise weder die tatsächliche Bedeutung der Komponenten noch die Namen der

zu bestimmenden Klassen. Wir können uns bei einer Klassifikation damit auf die beiden Datenmengen `data` und `target` beschränken.

```
X = dataset.data
y = dataset.target
```

Die Beschreibung benötigen wir nur für die Entscheidung, ob eine bestimmte Komponente (Feature) umskaliert oder umcodiert werden muss.

Die mitgelieferten Datensätze umfassen derzeit:

load_boston() (Regression) *Boston House-prices*. Modell für Immobilienpreise anhand von Größe, sozialem Umfeld etc.

load_iris() (Klassifikation) *Iris*. Klassifikation verschiedener Iris-Spezies anhand von verschiedenen Abmessungen der Blüte.

load_diabetes() (Regression) *Diabetes*. Modell für den Diabetes-Verlauf (als Zahl ausgedrückt) anhand von Alter, Gewicht, Geschlecht und Blutwerten.

load_digits() (Klassifikation) *Digits*. Erkennen handgeschriebener Ziffern aus 8×8-Bitmaps. Vereinfachter Datensatz der *MNIST Handwritten Digits*.

load_wine() (Klassifikation) *Wine recognition*. Einteilung von Wein in 3 Klassen anhand seiner chemischen Bestandteile.

load_breast_cancer() (Klassifikation) *Breast Cancer Wisconsin*. Entscheidung, ob ein Tumor gut- oder bösartig ist anhand numerischer Messwerte.

Weitere Bibliotheken wie *TensorFlow* (Bibliothek für [ANN](#)) oder *Seaborn* (Ergänzungen zu *matplotlib*) bringen weitere Datensammlungen mit, die einfach genutzt werden können.

Beispiel. Laden des *Iris*-Datensatzes. Dieser Datensatz mit 4 metrischen Komponenten und 3 Zielklassen ist aufgrund seiner Kleinheit und guten Klassifizierbarkeit gut geeignet, um sich insbesondere mit den Grundlagen und verschiedenen technischen Aspekten des *Machine Learning* vertraut zu machen.

Quellcode 8.1: load_iris.py: Laden und graphische Analyse des Iris-Datensatzes

```
#!/usr/bin/env python
# coding: utf-8

# # Iris-Beispieldatensatz

# Import der scikit-learn-Datensätze.

import pandas as pd
from sklearn import datasets

# Laden des Datensatzes und Anzeigen der Beschreibung.

data = datasets.load_iris()
print(data['DESCR'])

# Auslesen der notwendigen Daten sowie der Feature-Namen für einen Pandas-Data-
↳Frame.

X = data['data']
y = data['target']
colNames = data['feature_names']

# Anzeigen der Dimensionen sowie der ersten Datensätze.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

print('Data shape', X.shape, 'Anzahl Label', y.shape)
print('X:')
print(X[:5, :])
print('y:')
print(y[:5])

# Erzeugen eines Data Frame für schnelle statistische Analysen.

dfIris = pd.DataFrame(X, columns=colNames)
dfIris.info()

# Statistische Grundgrößen.

dfIris.describe()

# Dichteplot der Features.

ax = dfIris.plot.density(subplots=True)

# Scatter Matrix, Farbwahl über den Zielvektor y.

ax = pd.plotting.scatter_matrix(dfIris, c=y, cmap='Set1',
                                figsize=(10, 10))

```

Notebook: load_iris.ipynb.

8.2 Synthetische Datensätze

Zur Veranschaulichung der Arbeitsweise von Klassifikationsverfahren bietet *scikit-learn* die Möglichkeit, synthetische Datensätze zu generieren. Die dazu erforderlichen Methoden finden sich ebenfalls im Modul `datasets`.

```
from sklearn import datasets
```

Die Funktion `make_blobs()` erzeugt ohne weitere Parameter 100 zweidimensionale Datentypenpunkte, die sich in 3 Cluster (Klassen) verteilen.

```
X, y = datasets.make_blobs([centers=klassenzahl, cluster_std=streuung,
                             random_state=seed])
```

Mit dem Parameter `center` kann die Zahl der gewünschten Klassen angepasst werden. Die Streuung `cluster_std` gibt an, wie weit die Datenpunkte vom Klassenzentrum entfernt liegen dürfen. Je größer dieser Wert, desto weiter sind die Werte gestreut. Mit dem Parameter `random_state` setzen wir den Zufallszahlengenerator auf einen definierten Startwert, um stets die gleiche Punktmenge und Klasseneinteilung zu erzeugen.

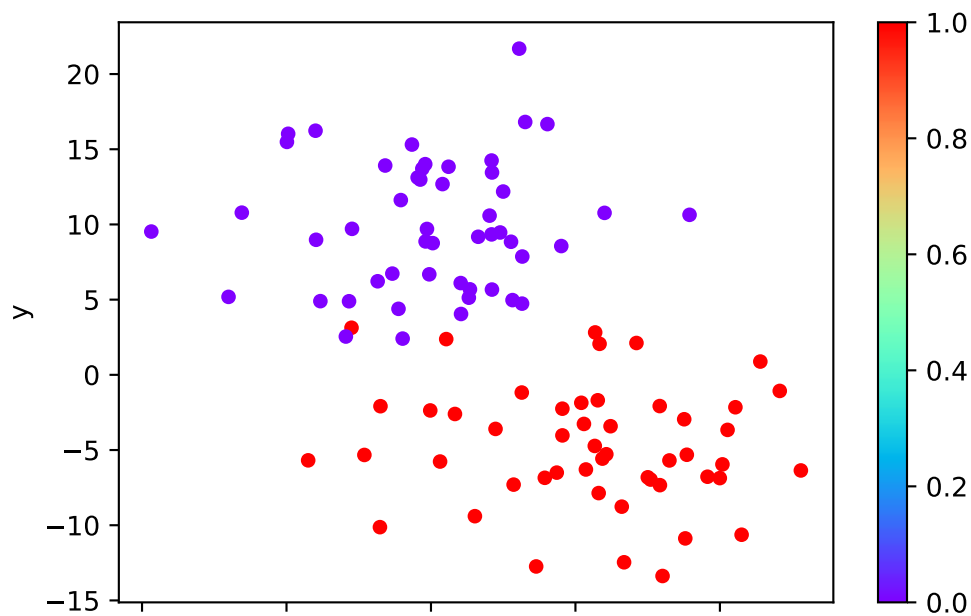
Beispiel:

```
X, y = dataset.make_blobs(centers=2, cluster_std=4, random_state=23)
```

Ergebnis:

Flexibler ist die Funktion `make_classification()`, die das Einfügen irrelevanter bzw. korrelierter Komponenten und das Erzeugen mehrerer Cluster pro Klasse erlaubt (s. Dokumentation).

Zur Untersuchung von Daten, die sich nicht linear (also durch eine Gerade bzw. Hyperebene) trennen lassen, dienen die Funktionen `make_moons()` und `make_circles()`.



```
X, y = datasets.make_moons([noise=rauschen, random_state=seed/])
```

Ohne weitere Parameter werden zwei Halbkreise erzeugt, die ineinander greifen. Der Parameter `noise` sorgt für die Streuung der Punkte um den Radius, `random_state` stellt wiederum den Zufallsgenerator auf einen definierten Startwert.

Beispiel:

```
X, y = datasets.make_moons(noise=0.1, random_state=23)
```

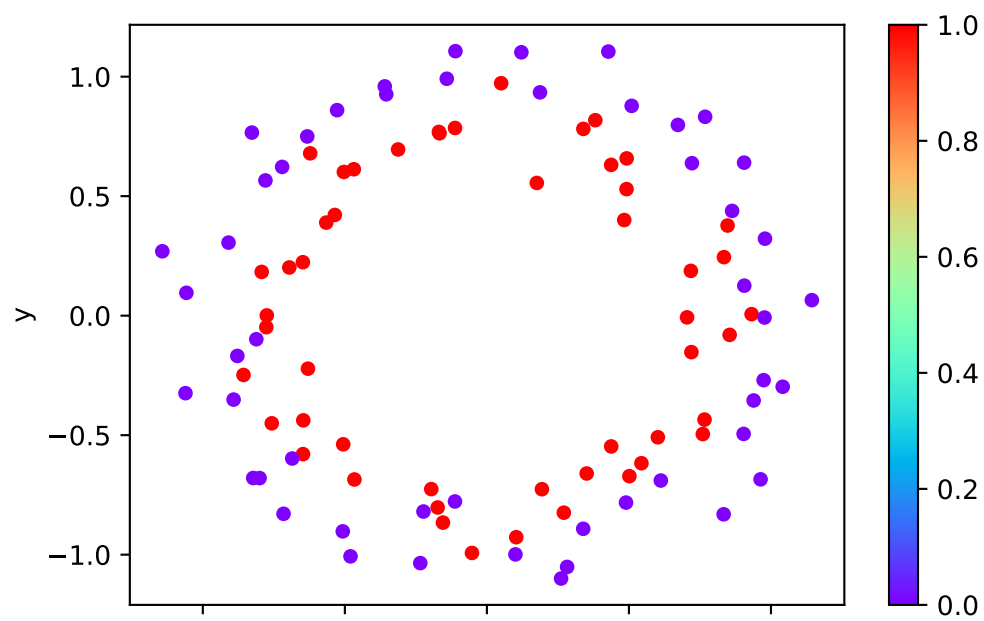
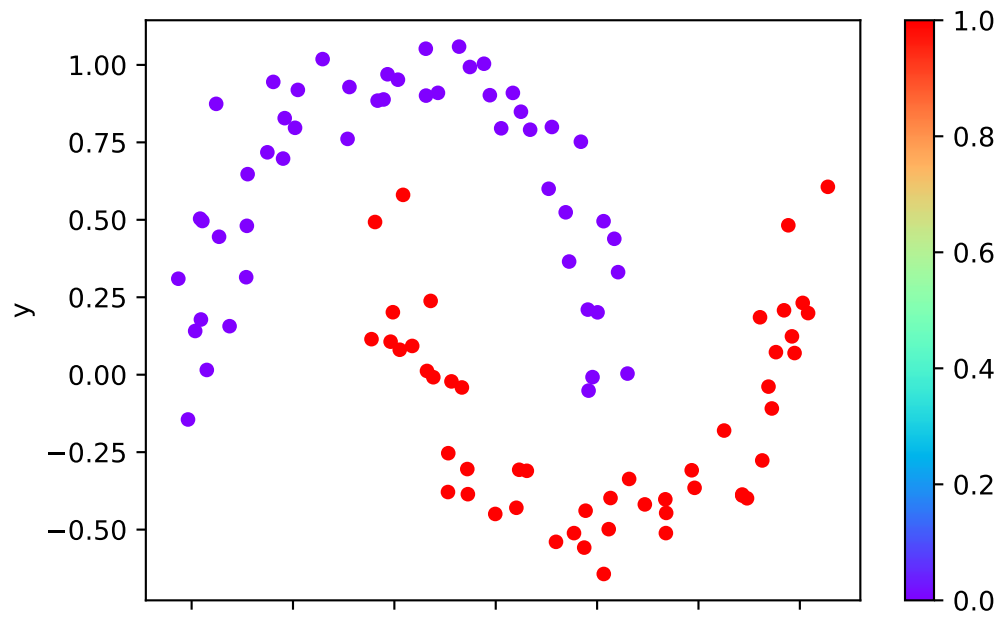
Ergebnis:

Die Funktion `make_circles` besitzt die gleichen Parameter, sie erzeugt zwei verschachtelte Kreise.

Beispiel:

```
X, y = datasets.make_circles(noise=0.1, random_state=23)
```

Ergebnis:



Notebook

Code zum Erzeugen der Datensätze und graphische Darstellung: `synthetic-datasets.ipynb`.

8.3 Datenquellen

Wollen wir unsere Verfahren an realitätsnäheren und umfangreicheren Daten untersuchen, finden wir im Internet eine Vielzahl von Repositories für Machine-Learning-Daten. Die bekanntesten sind:

[uci] UCI Machine Learning Repository. Es stehen ca. 500 beliebte Datensätze zur Verfügung, die meist im *CSV*-Format vorliegen. Bei vielen Datensätzen existieren Verweise auf weitergehende Literatur.

[kaggle] kaggle Anmeldung erforderlich. Datensätze in verschiedenen Formaten, meist *CSV* oder *JSON*. Zu vielen Datensätzen gibt es zusätzliche Informationen, Diskussionen und Beispiel-Notebooks. Auf der Seite werden regelmäßig auch Wettbewerbe durchgeführt und allgemeine Tutorials bereitgestellt.

[openml] OpenML Etwa 3000 Datensätze, freier Zugang. Eigenes Datenformat, für das *scikit-learn* eine spezielle Schnittstelle bereitstellt (s. u.).

[awsdata] Open Data in AWS Anmeldung erforderlich. Bisher keine Erfahrungen.

[googledata] Google's Dataset Search Engine Kein Repository, sondern Suchmaschine speziell Datensätze. Hilfreich, wenn der Name, aber nicht der Speicherort eines bestimmten Datensatzes bekannt ist.

Beispiel: Nutzung des *UCI*-Datensatzes *Bank Marketing*.

Wir suchen den Datensatz in der Übersicht und schauen uns die Details an. Der Datensatz enthält ca. 45000 Datenpunkte mit 17 Features. Es gibt keine fehlenden Werte. Im Datenverzeichnis liegen zwei Dateien, `bank.zip` und `bank-additional.zip` mit einer Auswahl aus den Originaldaten. Wir laden die erste der Dateien herunter und packen sie aus.

```
wget http://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank.zip
unzip bank.zip
```

Eine Inspektion mit einem Texteditor zeigt uns, dass die Daten eine Kopfzeile enthalten und die Spalten durch Semikolon getrennt sind. Eine Trennung in Trainings- und Testdaten ist bereits erfolgt.

Für erste Analysen lesen wir den Testdatensatz mit dem geringeren Datenumfang ein.

```
bank = pd.read_csv('bank.csv', sep=';')
bank.info()
```

Wir sehen, dass der Datensatz viele Nominaldaten enthält.

Wir zeigen uns die wichtigsten statischen Kenngrößen an.

```
bank.describe()
# und in einer neuen Notebook-Zelle:
bank.describe(include='object')
```

Die Wertebereiche der numerischen Komponenten sehr unterschiedlich, was für die meisten Lernverfahren recht ungünstig ist. Es hier eine Vorverarbeitung der Daten sinnvoll.

8.4 openml.org

Für die Datensätze aus *OpenML* stellt *scikit-learn* eine eigene Schnittstelle zur Verfügung.

```
from sklearn import datasets
```

Der zu ladende Datensatz wird über seinen Namen und einer optionalen Versionsnummer geladen. Wir suchen uns also zunächst mit <https://www.openml.org/search?type=data> den gewünschten Datensatz. Dieser kann nun mit der Funktion `fetch_openml()` geladen werden.

```
data = datasets.fetch_openml('name', version=nr)
```

Wir erhalten ein Dictionary mit folgenden Schlüsseln.

DESCR Beschreibung des Datensatzes als Text.

categories Ein Dictionary, das den numerisch codierten Nominaldaten die jeweiligen Kategoriebezeichnungen zuordnet. Jeder Schlüssel entspricht einer Tabellenspalte, die Werte sind die Listen der Kategoriebezeichnungen dieser Spalte.

data Matrix der Daten als *NumPy*-Array oder *Pandas* Data Frame. Die Kategoriedaten sind als Zahlen, fehlende Werte als *NaN* codiert.

details Dictionary mit Metadaten zum Datensatz.

feature_names Liste der Spaltennamen.

target Zielgröße, Liste der Funktionswerte (Label) der Daten.

Beispiel. Lesen des Datensatzes *Glass* von *openml.org*. Es gibt 2 Versionen dieses Datensatzes, wir verwenden hier die 2. Version mit nur zwei verschiedenen Klassen. Wir müssen zunächst untersuchen (oder alternativ als Parameter spezifizieren), ob wir die Daten als *NumPy*- oder *Pandas*-Datenstruktur erhalten (das ist auch von der konkreten Version der Bibliotheken abhängig).

Quellcode 8.2: `load_openml_glass.py`: Laden des Glass-Datensatzes

```
#!/usr/bin/env python
# coding: utf-8

# # Laden eines Open-ML-Datensatzes (Glass)

# Notwendige Importe: ``sklearn.datasets`` und ``pandas`` für erste Analysen.

import pandas as pd
import numpy as np
```

(Fortsetzung auf der nächsten Seite)

```
from sklearn import datasets

# Lesen des Datensatzes (Version 2 mit nur 2 Klassen).

data = datasets.fetch_openml('glass',version=2)
dataX = data['data']
dataY = data['target']
print('Typ X:', type(dataX), 'Typ y:', type(dataY))

# Beschreibung des Datensatzes anzeigen.

print(data['DESCR'])

# Wir erhalten hier *Pandas* Data Frames bzw. Series. Analyse der Spaltentypen.

print(dataX.info())
print('Verteilung Zielgröße:\n', dataY.value_counts())

# y ist als String codiert, Umwandlung in numerische Kategorie.
#
# *Achtung*: Die y-Werte haben noch nicht den Typ ``int``.

numY = dataY.map({'N': 0, 'P': 1})
numY.value_counts()

# Exportieren der Daten als *NumPy*-Datenstrukturen. ``y`` muss noch nach ``int``
↳ konvertiert werden. (Diese Umwandlung werden wir später
# mit sklearn-Werkzeugen durchführen).

X = dataX.values
y = numY.astype('int').values
# Kontrolle von Dimension und Datentyp
print('X:', X.shape, X.dtype)
print('y', y.shape, y.dtype)

# Statistische Analyse der Daten.

dataX.describe()

# Dichteplot der Features. Wir sehen unterschiedliche Wertebereiche.

ax = dfGlass.plot.density(subplots=True)

# Scatter-Matrix der Features.

ax = pd.plotting.scatter_matrix(dfGlass,c=numY,figsize=(15,15))
```

8.5 Kaggle, UCI etc.

Für weitere Repositories wie kaggle [[kaggle](#)] und *UCI* Machine Learning Repository [[uci](#)] stellt uns *scikit-learn* keine solchen bequemen Funktionen zum Laden der Daten zur Verfügung. Dies ist durch die uneinheitliche Bereitstellung der Daten bzw. eine notwendige Anmeldung für den Zugriff bedingt.

Wir laden diese Beispieldaten einfach im bereitgestellten *CSV*-Format in unser Arbeitsverzeichnis und lesen diese wie unter *Pandas* beschrieben in einen Data Frame, den wir dann weiter aufbereiten.

8.6 Aufgaben

Aufgabe 8.1 *Klassifikation Datensatz *Iris**

Laden Sie den Datensatz Iris und bestimmen Sie die beiden Komponenten mit der höchsten Korrelation zur Zielgröße (Pflanzenart). Beschränken Sie Ihre Daten auf diese beiden Komponenten und die Zielgröße und erstellen Sie ein Entscheidungsbaummodell, das Sie anschließend mit vorher zurückbehaltenen Testdaten bewerten.

Aufgabe 8.2 *Klassifikation mit synthetischen Daten*

Erzeugen Sie einen synthetischen Datensatz mit 500 Punkten, zwei Komponenten und 5 Klassen. Experimentieren Sie dabei mit Blobs, Monden und verschiedenen Werten für die Standardabweichung.

Erzeugen Sie daraus einen Pandas Data Frame und analysieren Sie numerisch und graphisch die statistischen Eigenschaften dieses Datensatzes.

Zerlegen Sie den Datensatz in Trainings- und Testdaten und klassifizieren Sie ihn mit einem Decision-Tree Classifier. Untersuchen Sie die Güte der Vorhersage für die Testdaten. Beobachten Sie ein Overfitting?

Vorverarbeitung der Daten

- *Fehlende Informationen*
- *Zerlegen in Trainings- und Testmenge*
- *Arbeitsweise von scikit-learn*
- *Zerlegen der Daten für verschiedene Transformationen*
- *Umskalieren numerischer Daten*
- *Umcodierung von Nominaldaten*
- *Auffüllen fehlender Daten*
- *Aufgaben*

Bevor wir unsere Lernverfahren auf unsere Trainingsdaten anwenden können, müssen diese meist vorverarbeitet werden. Wir müssen uns um fehlende Komponenten in den Datensätzen, nichtnumerische Komponenten und um die Skalierung der numerischen Komponenten kümmern.

9.1 Fehlende Informationen

Die meisten Lernverfahren können nicht mit fehlenden Informationen in Datensätzen umgehen, da sie die Daten als Vektoren in einem hochdimensionalen Raum betrachten. Fehlen Komponenten, kann den Daten hier keine Position zugeordnet werden.

Fehlende Komponenten sind in numerischen Daten üblicherweise als *NaN* (not a number) mit dem *NumPy*-Wert `np.nan` codiert. Bei Objektdaten wird das Spezialobjekt `None` verwendet.

Wir haben im drei verschiedene Möglichkeiten, mit fehlenden Daten umzugehen.

Löschen der Datensätze mit solchen Fehldaten. Damit können dann allerdings auch neu zu bewertende Realdaten mit fehlenden Werten nicht verarbeitet werden.

Löschen des kompletten Merkmals, in dem Daten fehlen. Ist nur eine Komponente betroffen, von der wir vermuten, dass sie nur wenig Einfluss auf das Ergebnis hat, und fehlt dieser Wert sehr häufig, kann das sinnvoll sein, andernfalls verlieren wir sehr viel Information für die Modellbildung.

Ersetzen fehlender Werte durch Defaultwert. Bei kategorialen Daten kann das entweder die am häufigsten auftretende Kategorie oder auch eine neue Kategorie *unbekannt* sein, bei numerischen Daten vielleicht ein Mittelwert oder Median.

Je nachdem, ob wir unsere Daten mittels *Pandas* oder direkt mit *NumPy* einlesen, bieten sich verschiedene Methoden zur Bereinigung an. Den ersten Fall haben wir im Abschnitt *Zugriff, Datenbereinigung* besprochen.

Mit *NumPy* ist das Vorgehen etwas aufwendiger: Wir ermitteln zunächst mittels `np.isnan()` eine Matrix logischer Werte, die allen Fehlstellen den Wert `True` enthält. Diese Matrix negieren wir mittels `np.logical_not()` (oder `maske == False`). Nun ermitteln wir daraus mit `all()` alle Zeilen (`axis=1`, über den Spaltenindex) oder Spalten (`axis=0`), die nur `True`-Werte enthalten, in denen in der Ausgangsmatrix also kein Wert fehlt. Wir erhalten einen (eindimensionalen) Vektor, den wir als Zeilen- bzw. Spaltenindex verwenden, um aus der Ursprungsmatrix die vollständigen Daten zu ermitteln.

Für die Zeilen gehen wir so vor:

```
rowMask = np.all(np.logical_not(np.isnan(matrix)), axis=1)
matrixBereinigt = matrix[rowMask, :]
# oder kurz mit Broadcasting
matrixBereinigt = matrix[mask]
# Falls die Zielgröße in einem eigenen Vektor gespeichert ist:
zielBereinigt = ziel[mask]
```

Für die Spalten funktioniert das Broadcasting nicht auf diese Weise, wir benutzen nur die erste Variante.

```
colMask = np.all(np.logical_not(np.isnan(matrix)), axis=0)
matrixBereinigt = matrix[:, rowMask]
# Vektor der Zielgröße bleibt unverändert.
```

Beispiel: Löschen der Zeilen bzw. Spalten mit Fehlwerten aus dem *OpenML*-Datensatz [\[openml\]](#) *mushrooms*.

Quellcode 9.1: `remove-missing-values.py`: Löschen von Zeilen bzw. Spalten mit Fehlwerten

```
#!/usr/bin/env python
# coding: utf-8

# # Entfernen von Zeilen bzw. Spalten mit Fehlwerten

# *scikit-learn* wird für das Lesen eines OpenML-Datensatzes benötigt.

import numpy as np
from sklearn import datasets

# Einlesen des Datensatzes als *NumPy*-Matrix (ohne ``as_frame=False`` wird er als
↳Data Frame gelesen).
#
# Anzeige der Komponenten des Datensatzes.

rawData = datasets.fetch_openml('mushroom', as_frame=False)
dir(rawData)

# Anzeige der Beschreibung (auskommentiert).

#print(rawData['DESCR'])

# Herauslösen der Feature-Matrix und der Zielgröße.

data = rawData['data']
target = rawData['target']
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

# Anzeige der Struktur von Daten und Zielgröße.

print('Data Matrix: ', data.shape, data.dtype)
print('Target:', target.shape, target.dtype)

# ## Untersuchen von Zeilen mit Fehlwerten
#
# Ermitteln der Zeilen mit Fehlwerten.

x = np.any(np.isnan(data), axis=1)
print('Mask incomplete rows:', repr(x))
print('Number of incomplete rows:', np.count_nonzero(x))

# Herauslösen aller vollständigen Datenzeilen.

nonNans = np.all(np.logical_not(np.isnan(data)), axis=1)
dataCleanedRows = data[nonNans, :]
targetCleanedRows = target[nonNans]
print(dataCleanedRows.shape, targetCleanedRows.shape)

# Kontrolle der bereinigten Daten, keine Fehlwerte mehr vorhanden.

np.count_nonzero(np.isnan(dataCleanedRows))

# ## Behandlung von Spalten mit Fehlwerten.
#
# Ermitteln der Spalten mit Fehlwerten. Anzeigen der Spaltenindices.

y = np.any(np.isnan(data), axis=0)
print(np.where(y)[0])

# Herauskopieren aller Spalten ohne Fehlwerte.
#
# Die Zeilenzahl bleibt nun gleich, der Vektor der Zielgröße muss nicht verändert_
→ werden.

nonNans = np.all(np.logical_not(np.isnan(data)), axis=0)
dataCleanedCols = data[:, nonNans]
print(dataCleanedCols.shape)

# Kontrolle, ob noch Fehlwerte vorhanden sind.

np.count_nonzero(np.isnan(dataCleanedCols))

```

Download Notebook: `remove-missing-values.ipynb`.

Wenn wir ganze Spalten löschen wollen und bereits wissen, welche das sind, geht das Löschen noch einfacher:

Pandas:

```
dfBereinigt = dataframe.drop([spaltenListe], axis=1)
```

NumPy:

```
matrixBereinigt = np.delete(matrix, indexListe, axis=1)
```

Das Auffüllen fehlender Werte mit Vorgabewerten erledigen wir entweder in Pandas mit den Methoden `interpolate()` bzw. `fillna()` oder verwenden besser die Transformationen von *scikit-learn*.

9.2 Zerlegen in Trainings- und Testmenge

Um unsere Machine-Learning-Modelle bewerten zu können, benötigen wir stets eine *Testmenge*. Das ist eine Anzahl von Datensätzen, für die die Zielgröße (Label/Klasse oder Zielwert) bekannt ist und die vor dem Erstellen des Modells zurückgelegt wird. Damit können wir bewerten, wie sich das Modell bei Anwendung auf bisher unbekannte Daten verhält.

Wir haben im Abschnitt *Auswahl einer Stichprobe* bereits gesehen, wie wir dies in *Pandas* erledigen können. Wollen wir diese Aufteilung nicht dauerhaft speichern, sondern nur temporär durchführen, können wir hier auch die Funktion `train_test_split()` aus dem Modul `model_selection` der *scikit-learn*-Bibliothek verwenden.

```
from sklearn.model_selection import train_test_split
trainDataFrame, testDataFrame, ... = train_test_split(dataframe, ... test_size=testRatio,
random_state=seed)
```

Diese Funktion kann mehrere Dataframes oder *NumPy*-Arrays mit gleicher Zeilenzahl (erster Index) gleichzeitig zerlegen. Das ist nützlich, wenn die Komponenten und die Label bzw. Zielgröße der Daten bereits in getrennten Feldern vorliegen.

Beispiel: Temporäres Aufteilen des UCI-Datensatzes *Abalone* in Trainings- und Testdaten mit `train_test_split()`. Der Datensatz hat keine Kopfzeile, diese wird ebenfalls ergänzt.

Quellcode 9.2: *abalone-split.py*: Aufteilen mit `sklearn.train_test_split()`

```
#!/usr/bin/env python
# coding: utf-8

# # Aufteilen des Abalone-Datensatzes mit scikit-learn

import pandas as pd
import numpy as np
from sklearn import model_selection

# Einlesen des Data Frame und Ergänzen der Spaltennamen, Kontrollanzeige der
↳Struktur.

colnames = ['sex', 'length', 'diameter', 'height', 'whole', 'shucked',
            'viscera', 'shell', 'rings',]

abalone = pd.read_csv(
    'http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.
↳data',
    header=None,
    names=colnames
)
print(abalone.info())

# Aufteilen des Data Frame. Es entstehen zwei Data Frames, deren Dimensionen zur
↳Kontrolle ausgegeben werden.

abaloneTrain, abaloneTest = model_selection.train_test_split(
    abalone, test_size=0.2, random_state=42)
print(abaloneTrain.shape, abaloneTest.shape)

# Kontrolle der Struktur eines Data Frames.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
abaloneTrain.info()
```

Download Notebook: `abalone-split.ipynb`.

Wir werden im weiteren jedoch unsere bereits mit *Pandas* vorbereiteten und aufgeteilten Datensätze verwenden.

Beispiel: Auftrennen des mitgelieferten Iris-Datensatzes. Hier liegt die Zielgröße (Irisart) in einem eigenen Array, Datenpunkte und Label müssen also parallel in Trainings- und Testdaten aufgeteilt werden.

Quellcode 9.3: iris-split: Aufteilung von NumPy-Feldern in Trainings- und Testmenge

```
#!/usr/bin/env python
# coding: utf-8

# # Aufteilen von NumPy-Datenstrukturen in Trainings- und Testdaten

import numpy as np
import pandas as pd
from sklearn import datasets, model_selection

# Der Iris-Datensatz liegt bereits aufgeteilt in Features und Zielgröße in Form
# von *NumPy*-Datenstrukturen vor.

iris = datasets.load_iris()
X = iris.data
y = iris.target
print('Feature-Matrix:', X.shape, X.dtype)
print('Zielvektor:', y.shape, y.dtype)

# Gleichzeitiges Aufteilen mehrerer *NumPy*-Felder. Es wird entlang der ersten
# Achse (axis=0) geteilt. Für jedes Feld, das als Parameter auftaucht, entstehen
# zwei Felder (Training und Test) im Rückgabewert.

Xtrain, Xtest, ytrain, ytest = model_selection.train_test_split(
    X, y, test_size=0.25, random_state=42
)
print('Trainingsdaten:', Xtrain.shape, ytrain.shape)
print('Testdaten:', Xtest.shape, ytest.shape)
```

Download Notebook: `iris-split.ipynb`.

Hinweis: Wenn auch zum Aufteilen der Daten in Trainings- und Testmenge eine komfortable Funktion zur Verfügung steht, werden wir an anderen Stellen trotzdem das manuelle Mischen der Daten und Aufteilen in einzelne Abschnitte benötigen. So müssen für jeden Trainingsdurchlauf eines *ANN* die Trainingsdaten neu durchmischt, in einzelne Blöcke zerlegt und diese sequentiell eingelesen werden, was über das Mischen und Aufspalten der Indexmenge effizient erfolgt.

Wollen wir lediglich das Verhalten verschiedener Machine-Learning-Modelle analysieren und nicht wirklich zukünftige Daten bewerten, können wir meist auf das dauerhafte Zerlegen der Daten in Trainings- und Testmenge und das dauerhafte speichern verzichten und statt dessen die Zerlegung bei Bedarf jedesmal neu vorzunehmen. Um dabei aber sicherzustellen, dass wir beim Erstellen des Modells niemals auch die Testdaten verwenden, muss die Zerlegung immer gleich passieren. Wir erreichen das durch die Initialisierung des Zufallsgenerators vor dem Mischen mit immer dem gleichen Wert.

9.3 Arbeitsweise von scikit-learn

Die Objekte zur Datenverarbeitung in *scikit-learn* werden einheitlich als *Estimatoren* bezeichnet. Wir unterscheiden nochmals *Filter* für die Datentransformation und eigentliche Vorhersagemodelle, die aus den Daten eine Zielgröße schätzen (die eigentlichen *Estimatoren*). Alle diese Objekte haben eine einheitliche *API*, was ihre Handhabung und die Erstellung neuer Objekte sehr vereinfacht.

Ein Objekt wird zunächst mit eventuell notwendigen Parametern erzeugt.

```
objekt = SklearnObjekt(hyperParameter)
```

Nun wird das Objekt auf unsere Daten *angelern*t.

```
[objekt =] objekt.fit(daten)
```

Die `fit()`-Methode liefert immer das Objekt selbst zurück, um Operationen auf dieses Objekt verketteten zu können.

Filter (Objekte zum Umwandeln von Daten) bieten stets noch eine Methode `transform()`.

```
datenTransformiert = objekt.transform(daten)
```

Meist steht noch eine Methode `fit_transform()` zur Verfügung, die beide Schritte zusammenfasst.

```
datenTransformiert = objekt.fit_transform(daten)
# Zusammenfassung von:
datenTransformiert = objekt.fit(daten).transform(daten)
```

Häufig wollen wir jedoch die Anpassung der Transformation nur mit unseren Trainingsdaten vornehmen, die eigentliche Transformation muss aber auch auf unsere Testdaten und neue Daten vor der weiteren Verarbeitung angewendet werden. Deshalb ist in den meisten Fällen die Trennung der beiden Schritte sinnvoll.

Estimatoren für Vorhersagen, also die eigentlichen Lernverfahren, besitzen keine `transform()`-Methode, sondern statt dessen eine Methode `predict()` für die Vorhersage der Zielgröße und weitere Methoden zur Evaluierung des Modells.

```
zielGröße = objekt.predict(daten)
```

Die Zielgröße ist üblicherweise ein Vektor, der die Vorhersagen für alle Datensätze unserer Daten enthält.

9.4 Zerlegen der Daten für verschiedene Transformationen

Üblicherweise müssen unterschiedliche Merkmale unserer Daten unterschiedlich transformiert werden. Dazu zerlegen wir unsere Daten zunächst in verschiedene Spaltengruppen, auf die wir die gleiche Transformation ansetzen wollen. Jeder dieser Gruppen ordnen wir ein Transformationsobjekt zu.

```
from sklearn import compose

tr = compose.ColumnTransformer(
    [
        ('identifizier', Transformer, spaltenListe),
        ...
        ('unverändert', 'passthrough', spaltenListe),
    ]
)

tr.fit(testDaten)

datenBereinigt = tr.transform(daten)
```

Die Daten können entweder als *Pandas* Data Frame oder als *NumPy*-Matrix übergeben werden. Es werden nur Spalten übernommen, die explizit aufgeführt sind. Spalten, die unverändert übernommen werden sollen, müssen wir mit der identischen Transformation durchreichen, die mit dem String 'passthrough' spezifiziert wird.

Die Arbeitsweise für die Datentransformation ist stets gleich: wir erzeugen ein Filterobjekt mit den gewünschten Parametern, ermitteln anhand der *Trainingsmenge* die notwendigen Skalierungsparameter mit der Methode `fit()` und wandeln sowohl Trainings- als auch Testmenge mit der Methode `transform()` um. Das `ColumnTransformer`-Objekt sorgt dafür, dass alle unterschiedlichen Spaltentransformationen bei der Anpassung und der Datenumwandlung einbezogen werden.

Warnung: Auch für die Datentransformation ist es wichtig, dass die Anpassung nur für die Trainingsmenge vorgenommen wird, da ansonsten Informationen der Testdaten in das Modell einfließen könnten, was die Bewertung der Güte beeinflusst.

Die eigentliche Datentransformation muss dann allerdings auch für die Testmenge und für neu zu bewertende Daten erfolgen.

Beispiel: Unser Beispieldatensatz *Abalone* enthält eine kategoriale und 7 metrische Spalten und die Zielspalte. Wir benutzen einen `MinMaxScaler` für unsere metrischen und einen `OneHotEncoder` für unsere kategorialen Spalten (s. nächster Abschnitt) und lassen die Zielgröße `labels` unverändert.

```
# Importe
import numpy as np
import pandas as pd
from sklearn import compose, preprocessing
import abalone_helper

# Einlesen der Trainings- und Testdaten
dfTrain = pd.read_csv('abalone-train.csv')
dfTest = pd.read_csv('abalone-test.csv')

# Umwandeln der Zielspalte in Label
dfTrainClass = abalone_helper.toLabels(dfTrain, 4)
dfTestClass = abalone_helper.toLabels(dfTest, 4)

# Numerische und kategoriale Spalten aufteilen in Listen
numCols = ['length', 'diameter', 'height', 'whole', 'shucked',
           'viscera', 'shell']
catCols = ['isex']

# Transformationsobjekt erzeugen
transform = compose.ColumnTransformer([
    ('num', preprocessing.MinMaxScaler(), numCols),
    ('cat', preprocessing.OneHotEncoder(), catCols),
    ('pass', 'passthrough', ['labels']), # Spalte als Liste übergeben!
])

# Transformation lernen mit den Trainingsdaten
transform.fit(dfTrainClass)

# Trainings- und Testdaten transformieren
dataTrain = transform.transform(dfTrainClass)
dataTest = transform.transform(dfTestClass)

# Zerlegen in Features und Zielgröße
Xtrain = dataTrain[:, :-1]
yTrain = dataTrain[:, -1]
Xtest = dataTest[:, :-1]
yTest = dataTest[:, -1]
```

Warnung: Wenn wir mit unserem Modell später tatsächlich neue Daten bewerten wollen, bei denen die Zielgröße nicht bekannt ist, dann muss die Zielgröße vor der Transformation abgetrennt werden, die Transformation darf nur auf die Komponenten trainiert und angewendet werden, damit wir die Transformation genauso auf die zukünftigen Daten anwenden können.

Beispiel: Transformation nur der Komponenten ohne Zielgröße für *Abalone*.

```
# Labels abtrennen
dfTrainFeatures = dfTrainClass.drop('labels', axis=1)
dfTestFeatures = dfTestClass.drop('labels', axis=1)
# Transformation wie oben, aber ohne Label-Spalte
trFeatures = compose.ColumnTransformer([
    ('num', preprocessing.MinMaxScaler(), numCols),
    ('cat', preprocessing.OneHotEncoder(sparse=False), catCols),
])
# Transformation der Features
Xtrain = trFeatures.fit(dfTrainFeatures).transform(dfTrainFeatures)
Xtest = trFeatures.transform(dfTestFeatures)
# Labels müssen nur in NumPy-Vektor umgewandelt werden
yTrain = dfTrainClass['labels'].values
yTest = dfTestClass['labels'].values
```

Download Notebook: `abalone-columntransformer.ipynb`.

9.5 Umskalieren numerischer Daten

Da sehr viele Modelle des *Machine Learning* die Ähnlichkeit von Daten aufgrund ihres *euklidischen Abstandes* bewerten, ist es wichtig, dass die Wertebereiche der Komponenten die gleiche Größenordnung besitzen. Dieser Abstand berechnet sich nach dem Satz des Pythagoras, indem wir den Abstand zwischen jeder Komponente quadrieren, diese Abstände alle aufsummieren und die Wurzel aus der Summe ziehen:

$$\text{abstand}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

Für den Datensatz *Abalone* reicht z. B. die Länge von fast 0 bis zu 0.81, das Gesamtgewicht dagegen fast bis zum Wert 3. Damit wirkt sich eine Gewichtsänderung von einem Prozent beim Gewicht wesentlich stärker auf den Abstand aus als eine Längenänderung von einem Prozent.

Es ist aus diesem Grunde meist sinnvoll, alle Daten auf gleichgroße Intervalle umzurechnen, so dass relative Änderungen einer Komponente immer den gleichen Einfluss auf den Abstand zwischen den Daten haben. Dazu haben sich zwei Vorgehensweisen etabliert:

MinMaxScaler Alle Werte der Komponente werden so skaliert, dass die transformierten Werte in einem vorgegebenen Intervall liegen (z. B. $[0, 1]$ oder $[1, 1]$).

StandardScaler Die Daten werden so transformiert, dass sie einer Wahrscheinlichkeitsverteilung mit Mittelwert 0 und Varianz 1 entsprechen.

Beide Skalierungen sind lineare Transformationen, i. e. die Daten werden lediglich verschoben und mit einem Faktor gleichmäßig vergrößert und verkleinert. Welche der beiden Skalierungen besser geeignet ist, hängt vom konkreten Modell und von den zu untersuchenden Daten ab und kann meist nur experimentell ermittelt werden. Außer diesen beiden am häufigsten benutzten Transformationen stellt *scikit-learn* weitere Skalierer zur Verfügung.

Das `StandardScaler`-Objekt wird ohne Parameter initialisiert:

```
from sklearn import preprocessing

scaler = preprocessing.StandardScaler()
```

Für den `MinMaxScaler` kann das gewünschte Intervall angegeben werden, der Vorgabewert ist das Intervall $[0, 1]$.

```
from sklearn import preprocessing
```

```
scaler = preprocessing.MinMaxScaler([feature_range=(minimum, maximum)])
```

Diese Transformationen werden beide mit der Methode `fit()` mit Hilfe der Trainingsdaten eingestellt, anschließend werden sowohl Trainings- als auch Testdaten mit der Methode `transform()` umgewandelt. Werden sie in das `ColumnTransformer`-Objekt wie oben eingebaut, wird der Aufruf von diesem Objekt automatisch ausgelöst.

9.6 Umcodierung von Nominaldaten

Um mit Nominalskalen (Kategoriedaten) arbeiten zu können, müssen wir diese geeignet in numerische Werte umsetzen. Das einfachste Vorgehen ist dabei eine sogenannte *Faktorisierung*: wir ordnen einfach jeder Kategorie eine Zahl zu, die wir eventuell noch auf das Intervall $[0, 1]$ umskalieren. Dieses Vorgehen hat jedoch den Nachteil, dass wir diesen Kategorien damit automatisch eine Ordnung und geben und in ein festes Verhältnis zueinander setzen, was unsere Modelle verfälscht (Beispiel: Auswirkung der Haarfarbe auf Bereitwilligkeit zum Studium).

Aus diesem Grund ist es meist besser, mit einer *One-Hot-Codierung* zu arbeiten: Wir zerlegen die Kategorievariable mit k verschiedenen Kategorien in k Einzelkomponenten, von denen genau eine den Wert 1 hat, wenn der Datensatz den Wert hat, der dieser Komponente entspricht, während alle anderen Komponenten für diesen Datensatz 0 sind.

Beispiel: Wir haben eine kategoriale Komponenten `cat` mit den drei Werten `a`, `b` und `c`. Aus dieser erzeugen wir drei neue Komponenten `cat_a`, `cat_b` und `cat_c`. Ein Datensatz mit dem Wert `b` erzeugt in der Spalte `cat_b` den Wert 1 und in den Spalten `cat_a` und `cat_c` den Wert 0.

Diese Codierung erzeugen wir mit einem Objekt der Klasse `OneHotEncoder`.

```
MyCatTransform = preprocessing.OneHotEncoder([sparse=False])
```

Hinweis: Falls sehr viele Spalten mit sehr vielen Kategorien auftreten, wird normalerweise eine *sparse matrix* erzeugt, die uns im folgenden Schwierigkeiten machen kann. Deshalb sollten wir diese kompakte Speicherung besser abschalten.

Für die Zielgröße einer Klassifikation, die ebenfalls eine Nominalvariable ist, genügt es für die meisten klassischen Verfahren, jede Kategorie als Zahl zu codieren, da hier die zugeordnete Zahl keine Reihenfolge in das Modell interpretiert. Wir haben in unseren Beispielen diese Umcodierung bereits mit den *Pandas*-Werkzeugen vorgenommen, können dies aber auch mit einem `LabelEncoder`-Objekt während der Transformation erledigen.

```
from sklearn import preprocessing
```

```
labelEncoder = preprocessing.LabelEncoder()
```

```
labelEncoder.fit(trainingsVektor)
```

```
yTrain = labelEncoder.transform(trainingVektor)
```

```
yTest = labelEncoder.transform(testVektor)
```

Die Zuordnung zwischen Klassen und Labels, die für die Klassifikation neuer Daten erforderlich ist, kann aus der Eigenschaft `classes_` des Transformer-Objekts oder mit der inversen Transformation `inverse_transform()` ermittelt werden.

Wir werden in unseren Beispielen diese Möglichkeit nicht nutzen.

9.7 Auffüllen fehlender Daten

Hinweis: Da wir im Rahmen des Praktikums mit passend aufbereiteten Datensätzen ohne Fehlzeiten arbeiten können oder solche Datensätze einfach entfernen werden, dient dieser Abschnitt nur der Vollständigkeit.

Zum Auffüllen fehlender Daten steht uns die Klasse `SimpleImputer` zur Verfügung.

```
from sklearn.impute import SimpleImputer

auffueller = SimpleImputer(strategy='strategie', fill_value=wert)
```

Für alle Datentypen können wir die Strategien `most_frequent` und `constant` verwenden. Für die letztere benötigen wir den Parameter `fill_value`. Für numerische Daten stehen zusätzlich die Strategien `mean` und `median` zur Verfügung.

Nach dem Auffüllen ist im allgemeinen noch ein Umskalieren bzw. Umcodieren der Daten erforderlich. Wir können dazu mehrere Filter hintereinander schalten.

```
from sklearn.pipeline import Pipeline

filterFolge = Pipeline([
    filter,
    ...
])
```

Diesen neuen Filter können wir dann in unserem `ColumnTransformer`-Objekt statt einer einfachen Transformation verwenden.

9.8 Aufgaben

Aufgabe 9.1 Untersuchen Sie in Ihrem gewählten Datensatz die Wertebereiche Ihrer numerischen Komponenten. Transformieren Sie diese mit zum einen mit einem `StandardScaler`, zum anderen mit einem `MinMaxScaler` und analysieren Sie mittels Boxplot und Dichtediagramm in die Werteverteilung für beide Verfahren (wandeln Sie dazu die transformierten Daten wieder in einen Pandas Frame um).

Schreiben Sie nun eine Transformation, die auch die kategorialen Daten passend umwandelt und Ihre Zielgröße durchreicht.

Transformieren Sie Trainings- und Testdaten in `NumPy`-Felder und zerlegen Sie diese jeweils in die Feature-Matrix und den Vektor der Zielgröße.

- *Einführung*
- *Plotten einfacher Funktionen und Einstellungen*
- *Subplots und Achsenskalierung*
- *Logarithmische Skala, Darstellungsintervall*
- *Statistische Plots*
- *Plotten 3-dimensionaler Daten*
- *Aufgaben*

10.1 Einführung

Zum Visualisieren unserer Daten und der Modelle verwenden wir die Bibliothek *matplotlib*, die mit *NumPy*, *Pandas* zusammenarbeitet und bequem in *Jupyter Notebooks* eingebettet werden kann.

Erläuterungen zur Funktionsweise und Verwendung findet man z. B. auf der Homepage von [matplotlib] <https://matplotlib.org/3.1.1/tutorials/introductory/usage.html> und Beispiele unter https://matplotlib.org/3.1.1/tutorials/introductory/sample_plots.html.

Die Bibliothek wird üblicherweise unter dem Alias `plt` eingebunden.

```
import matplotlib.pyplot as plt
```

Es existieren sowohl zustands- als auch objektorientierte Schnittstellen, wir werden hier meist die erste Form verwenden.

10.2 Plotten einfacher Funktionen und Einstellungen

Zum Plotten einfacher Funktionen benötigen wir zwei Vektoren, einen für die Parameterwerte (Stützstellen) und einen zweiten für die Funktionswerte. Den ersten Vektor können wir mit den *NumPy*-Funktionen `np.linspace()` oder `np.logspace()` einfach erzeugen.

```
import numpy as np

xlin = np.linspace(start, ende, anzahl)
xlog = np.logspace(startExponent, endExponent, anzahl, basis)
```

Auf jeden Wert von x wenden wir nun unsere Funktion an und erhalten den Vektor unserer Funktionswerte. Die Werte können entweder punktweise oder mit *NumPy* automatisch für den ganzen Vektor berechnet werden.

```
yManuell = [f(p) for p in x]
yNumPy = np.f(x)
```

Beispiel: Berechnung der logistischen Funktion vektorweise mit *NumPy* und von `atan()` punktweise.

```
x = np.linspace(-10, 10, 50)
y1 = 1.0 / (1.0 + np.exp(-x))
y2 = [math.atan(v) for v in x]
```

Eine Funktion kann nun einfach wie folgt dargestellt werden:

```
plt.plot(xVektor, yVektor[, linienstil][, label='legendentext'] ...)
...
```

Oft werden die Plots im *Jupyter Notebook* recht klein dargestellt, wir können deshalb vorher die Größe anpassen.

```
plt.figure(figsize=(breite, höhe))
```

Der Linienstil kann in einer Kurzform als Kombination aus Farbe, Linienstil und Marker angegeben werden. Für die Farben stehen die Buchstaben `r` (red), `g` (green), `b` (blue), `samp:c` (cyan), `samp:m` (magenta), `samp:y` (yellow), `samp:k` (black) und `samp:w` (white) zur Verfügung. Für die wichtigsten Linienstile werden die Kürzel `-` (durchgezogene Linie), `--` (Strichlinie), `-.` (Strich-Punkt-Linie) und `:samp:`` (gepunktete Linie) verwendet. Wollen wir die Funktionswerte an den Stützstellen markieren (was insbesondere beim Darstellen eingelesener Beobachtungsdaten interessant ist), können wir u. a. die Zeichen `.` (Punkt), `,` (Pixel = kleiner Punkt), `o` (Kreis = großer Punkt), `+` (Pluszeichen) `x` (x-Zeichen), `^` (Dreieck), `v` (gedrehtes Dreieck) und `*` (Stern) verwenden. Die Kombination `k--*` erzeugt also eine schwarze, gestrichelte Linie mit Sternchen an den Stützstellen.

Wir können diese Parameter auch getrennt spezifizieren, was uns weitere Farben erlaubt.

c='farbe' Kurzbezeichnung, als *RGB*-Tupel mit Werten zwischen 0 und 1 oder als Hexadezimalstring mit vorangestellter Raute (z. B. `'#005f50'` für das TU-Grün).

ls='linienstil:kw:' Linienstil wie oben, ein leerer String zeichnet keine Linie.

lw='dicke' Linienstärke.

Wir können mehrere Kurven in ein Diagramm durch wiederholten Aufruf von `plt.plot()` zeichnen.

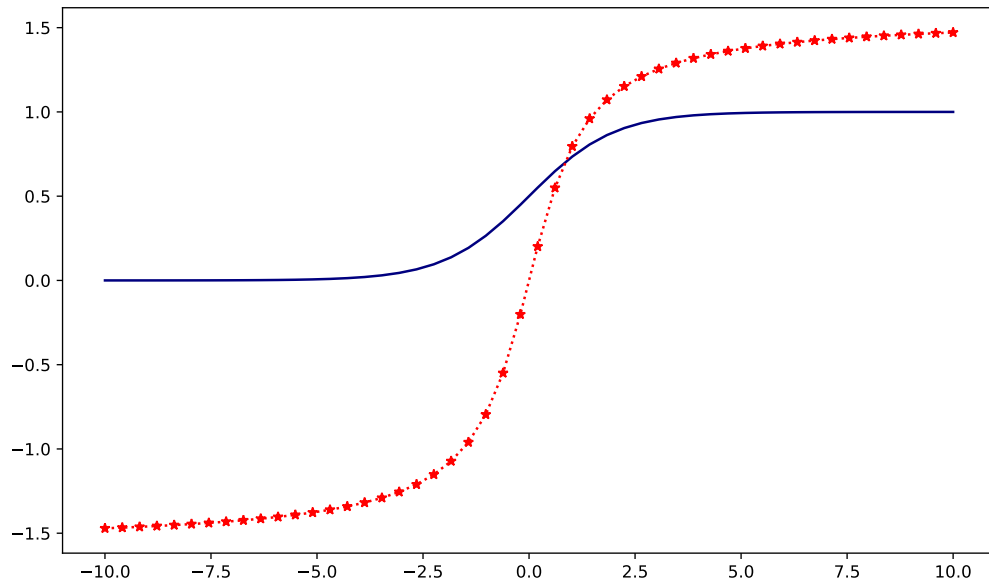
Beispiel: Darstellung unserer beiden Funktionen.

```
# dicke, dunkelblaue, durchgezogene Linie
plt.plot(x, y1, ls='-', c='#00007f', lw=3, label='logit(x)')
# rote, gepunktete Linie mit * an Stützstellen
plt.plot(x, y2, 'r:*', label='arctan(x)')
```

Ergebnis:

Der Abschluss und Ausgabe des Diagramms erfolgt mit der Funktion `plt.show()`. Dieser Aufruf erfolgt automatisch am Ende der Zelle und muss nicht explizit aufgeführt werden.

```
plt.show()
```

Abb. 10.1: Logistische und *atan()*-Funktion.

Vor der Ausgabe müssen auch alle anderen Zeichenbefehle wie Ausgabe von Achsenbeschriftungen oder Legenden ausgeführt werden.

Wir können das Diagramm einfach um waagerechte und senkrechte Linien, z. B. durch den Koordinatenursprung (0, 0), erweitern.

```
plt.axhline(yWert[, stil])
plt.axvline(xWert[, stil])
```

Die abkürzende Formatschreibweise ist hier nicht möglich, für Farbe und Linienstil müssen die beiden Parameter `c` und `ls` einzeln verwendet werden.

Beispiel: Einzeichnen der Achsen $x=0$ und $y=0$ als Strichlinien.

```
plt.axhline(0, ls='--', c='g', label='y=0')
plt.axvline(0, ls='--', c='k', label='x=0')
```

Um das Diagramm zu beschriften, verwenden wir die Funktionen `plt.title()` für den Diagrammtitel, `plt.xlabel()` und `plt.ylabel()` für die Achsenbeschriftung und `plt.legend()` für eine Legende, die die `label`-Parameter aus den Plot-Befehlen übernimmt.

Beispiel: Ergänzung von Titel, Achsenbeschriftung und Legende.

```
plt.title('Aktivierungsfunktionen')
plt.xlabel('Parameter x')
plt.ylabel('Funktionswert y')
plt.legend()
```

Um unsere Plots in anderweitigen Dokumentationen verwenden zu können, verwenden wir die Funktion `plt.savefig()`, die wir unmittelbar vor `plt.show()` aufrufen.

```
plt.savefig('dateiname'[, ...])
```

Der Dateityp wird automatisch aus dem Dateiname abgeleitet. Sinnvoll sind PNG- oder SVG-Bilder für Online-Dokumentationen und PDF-Dateien für den Ausdruck. Mit zusätzlichen Parametern können die Bildgröße und -ausschnitt bestimmt werden.

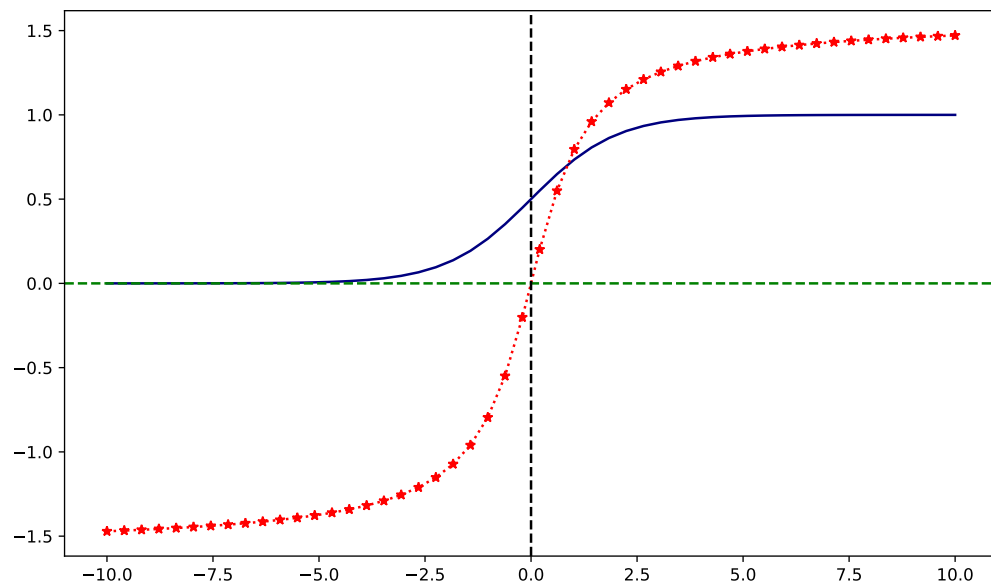
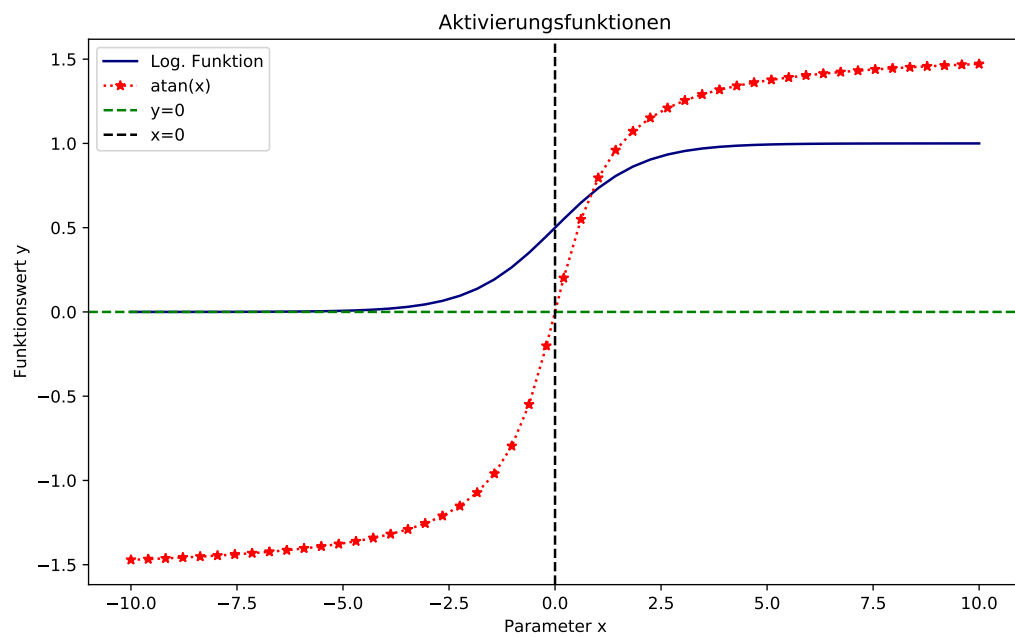
Abb. 10.2: Ergänzung der Achsen $x=0$ und $y=0$ 

Abb. 10.3: Ergänzung von Beschriftungen und Legende.

Notebook

Plotten der obigen Beispiele: `simplefunc.ipynb`.

10.3 Subplots und Achsenskalierung

Zur Anordnung mehrerer Graphiken in einem Raster verwenden wir die Funktion `plt.subplot()`.

`plt.subplot` (zeilenZahl, spaltenZahl, nr)

Die Numerierung erfolgt zeilenweise und startet mit 1. Beispiel: zwei Diagramme nebeneinander.

```
x = np.linspace(-5, 5, 50)
plt.figure(figsize=(10, 4))
# 1 Zeile, 2 Spalten, 1. Plot
plt.subplot(1, 2, 1)
plt.plot(x, np.sin(x), label='sin(x)')
plt.legend()
# 2. Plot
plt.subplot(1, 2, 2)
plt.plot(x, np.cos(x), label='cos(x)')
plt.legend()
plt.show()
```

Eine gemeinsame Beschriftung für alle Subplots kann mit `plt.suptitle()` erzeugt werden.

`plt.suptitle` ('titel')

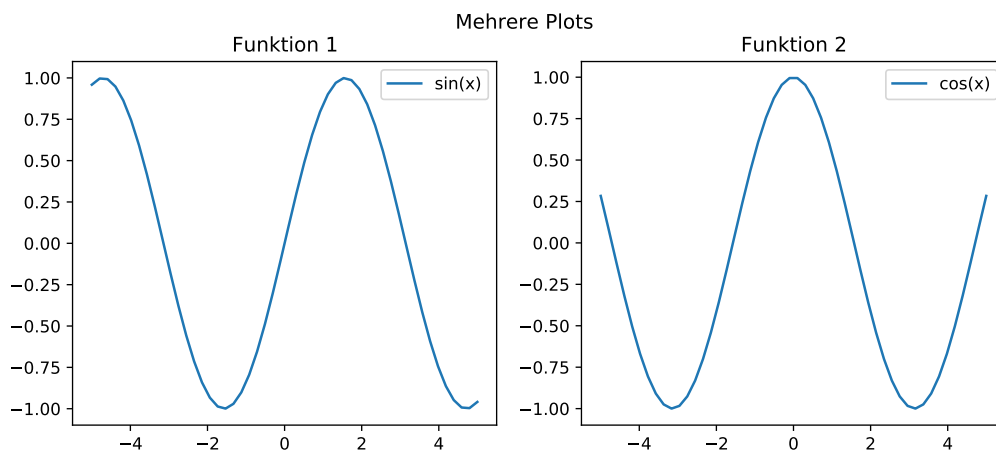


Abb. 10.4: Subplots von Funktionen mit gemeinsamem und Einzeltiteln und -legenden.

Notebook

Darstellung des obigen Subplots: `subplot1.ipynb`.

10.4 Logarithmische Skala, Darstellungsintervall

Überstreichen unsere Werte mehrere Zehnerpotenzen, ist es oft sinnvoll, die Skalen auf eine logarithmische Darstellung zu ändern.

```
plt.xscale('log')
plt.yscale('log')
```

Mit den Funktionen `plt.xlim()` und `plt.ylim()` können wir das Darstellungsintervall einschränken.

```
plt.xlim(minimum, maximum)
plt.ylim(minimum, maximum)
```

Beispiel: Laufzeiten verschiedener Sortierverfahren in Abhängigkeit der Datengröße. Datensatz `sort.dat`. Die Spalten sind durch Leerzeichen getrennt, die Kopfzeile ist auskommentiert. Bei linearer y-Skalierung wird der Wertebereich auf das Intervall $[100, 10^5]$ begrenzt.

Quellcode 10.1: `sorting_plot.py`: Lineare und logarithmische Skalen

```
#!/usr/bin/env python
# coding: utf-8

# # Plots mit logarithmischen Skalen

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Einlesen des Datensatzes. Sortierzeit in Abhängigkeit von Verfahren und
↳ Eingabegröße.

runtime = pd.read_csv('sort.dat', sep=' ', comment='#', names=['method', 'size',
↳ 'time'])
runtime.info()

# Ausgabe der Verfahren und Anzahl der Werte.

runtime['method'].value_counts(sort=False)

# Zerlegen in einzelne Data Frames für jedes Verfahren.

runb = runtime[runtime['method']=='b'].drop('method', axis=1)
runh = runtime[runtime['method']=='h'].drop('method', axis=1)
runi = runtime[runtime['method']=='i'].drop('method', axis=1)
runs = runtime[runtime['method']=='s'].drop('method', axis=1)

# Hilfsfunktion zum Plotten der Kurven für alle 4 Verfahren.

def plotLines():
    plt.plot(runb['size'], runb['time'], marker='o', label='Bubble')
    plt.plot(runh['size'], runh['time'], marker='.', label='Heap')
    plt.plot(runi['size'], runi['time'], marker='v', label='Insertion')
    plt.plot(runs['size'], runs['time'], marker='^', label='Selection')
    plt.legend()
    plt.xlabel('Array Size')
    plt.ylabel('Runtime (ms)')
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
# Plot mit linearen bzw. logarithmischen Skalen.

plt.figure(figsize=(12, 12))
plt.subplot(2, 2, 1)
plotLines()
# Begrenzung der y-Achse
plt.ylim(100, 1e5)
plt.title('Linear Scales')
plt.subplot(2, 2, 2)
plotLines()
plt.yscale('log')
plt.title('Logarithmic y-scale')
plt.subplot(2, 2, 3)
plotLines()
plt.xscale('log')
# Begrenzung der y-Achse
plt.ylim(100, 1e5)
plt.title('Logarithmic x-scale')
plt.subplot(2, 2, 4)
plotLines()
plt.xscale('log')
plt.yscale('log')
plt.title('Logarithmic x- and y-scale')
plt.suptitle('Different scales for sorting times')
plt.savefig('axscale.png')
plt.savefig('axscale.svg')
plt.show()
```

Download Notebook: `sorting_plot.ipynb`.

10.5 Statistische Plots

Vor der Erstellung eines Machine-Learning-Modells ist es meist sinnvoll, die zur Verfügung stehenden Daten statistisch zu analysieren. Für die graphische Auswertung stehen verschiedene Diagrammtypen zur Verfügung.

Einige Möglichkeiten von *Pandas*, die ebenfalls auf *Matplotlib* aufbauen, haben wir bereits kennengelernt. Wir können ähnliche Analysen aber auch direkt auf *NumPy*-Datenstrukturen ausführen, was immer dann sinnvoll ist, wenn unsere Daten bereits in dieser Form vorliegen oder aus einer Berechnung stammen.

Als Beispieldatensatz verwenden wir die Datensätze *Iris* aus dem *scikit-learn*-Paket für die Klassifikation mit rein numerischen Daten und *Abalone*, den wir von [openml] mit den *scikit-learn*-Werkzeugen als *NumPy*-Feld laden können.

```
dataIris = datasets.load_iris()
Xiris = dataIris['data']
yiris = dataIris['target']
colIris = dataIris['feature_names']
dataAbalone = datasets.fetch_openml('abalone', version='active',
    as_frame=False)
Xabalone = dataAbalone['data']
# Vektor enthält Strings, die konvertiert werden müssen.
yAbalone = dataAbalone['target'].astype('int')
colAbalone = dataAbalone['feature_names']
```

Ein *Scatter Plot* zwischen zwei Komponenten wird ähnlich wie in *Pandas* erzeugt. Wir wählen zwei Spalten für die x- und die y-Komponente und eventuell weitere Spalten für Farbe und Größe aus.

```
plt.scatter(xVektor, yVektor[, c=zVektor][, s=größenVektor])
[
    plt.colorbar()
```

Different scales for sorting times

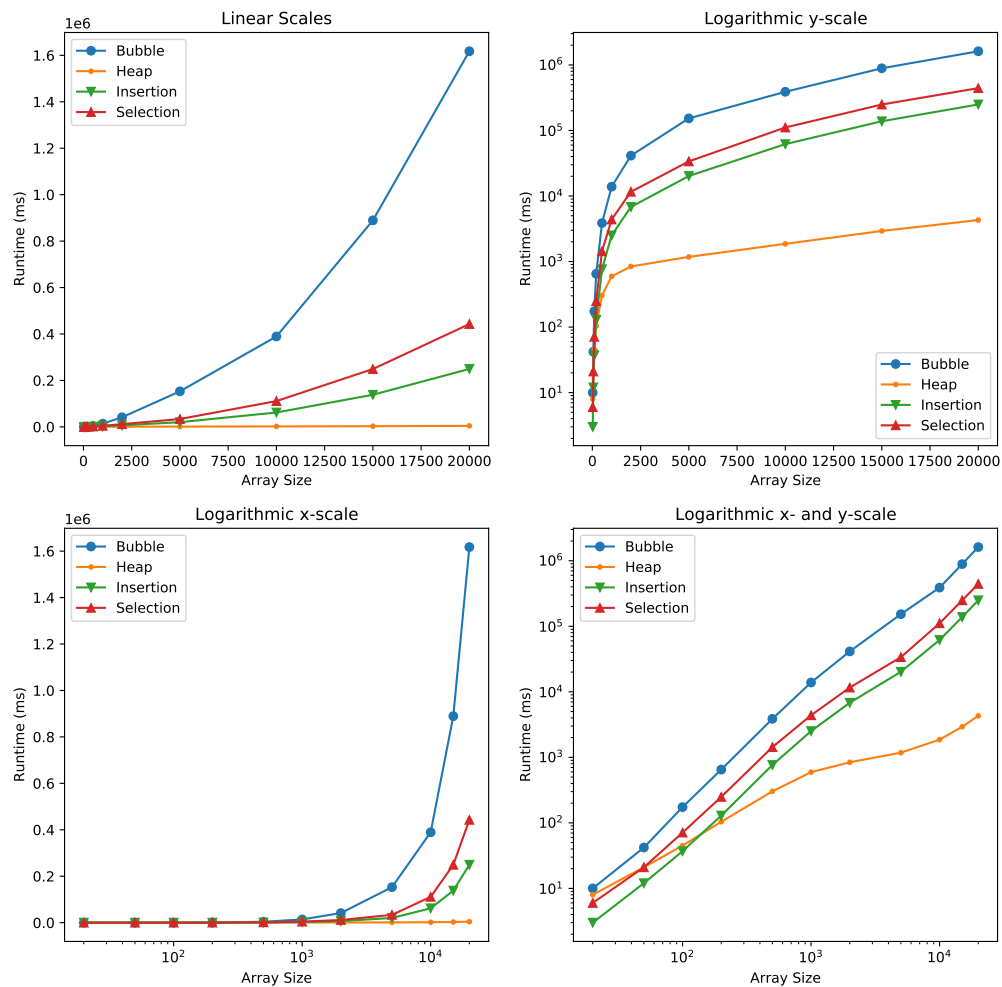


Abb. 10.5: Verschiedene Achsenskalierungen

]

Die Farben müssen einfach Zahlen sein, die Auswahl erfolgt entsprechend einer *Palette*, die mit dem Parameter `cmap()` gewählt werden kann.

Beispiel: Scatterplot der ersten beiden Spalten von *Iris*. Die Farbe stellt die Klasse bzw. den Zielwert dar.

```
plt.scatter(Xiris[:,0], Xiris[:,1], c=yiris)
plt.xlabel(colIris[0])
plt.ylabel(colIris[1])
plt.title('Iris')
```

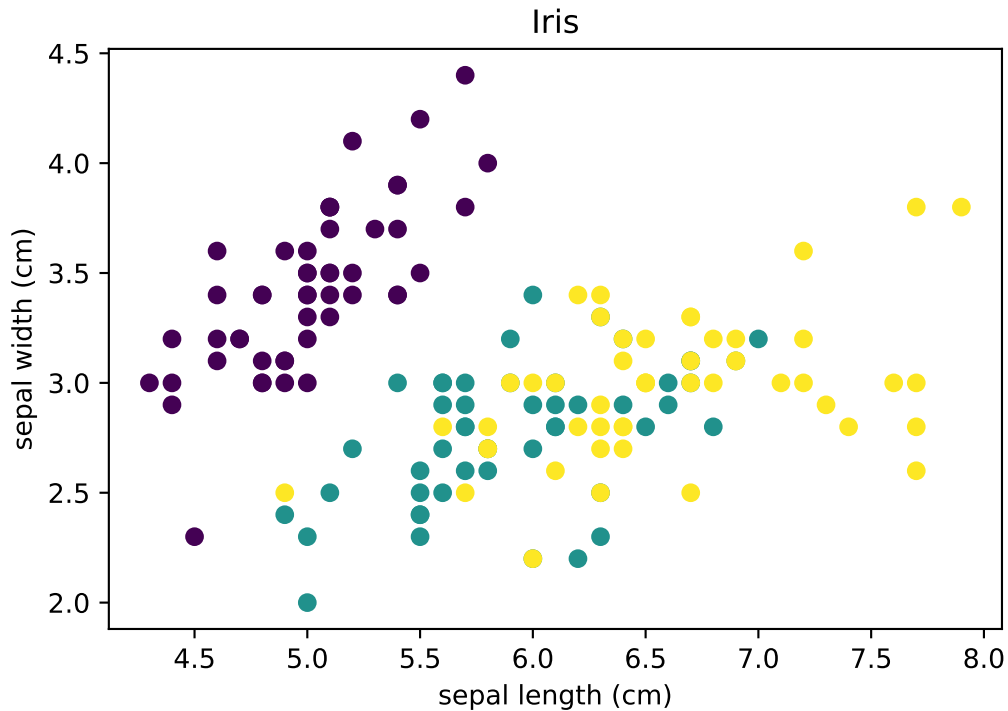


Abb. 10.6: Scattermatrix für 2 Komponenten von *Iris*. Der Farbbalken ist hier nicht sinnvoll.

Beispiel: Plot von 2 numerischen Features von *Abalone*. Die Farbe codiert die Zielgröße, ein drittes Feature ist in der Punktgröße codiert (muss umskaliert werden). Zur besseren Erkennbarkeit werden die Punkte halbtransparent dargestellt (Parameter `alpha`).

```
plt.scatter(Xabalone[:, 1], Xabalone[:, 4], c=yAbalone,
            cmap='hot', s=20*Xabalone[:, 5], alpha=0.5)
plt.xlabel(colAbalone[1])
plt.ylabel(colAbalone[4])
plt.title('Abalone')
plt.colorbar()
plt.show()
```

Wollen wir Daten verschiedener Klassen mit verschiedenen Symbolen plotten, müssen wir sie nach ihren Klassen aufspalten und getrennt zeichnen, da für das Symbol kein Vektor übergeben werden kann. Beispiel für den *Iris*-Datensatz. Da sich das Diagramm aus drei Einzelplots zusammensetzt, erhalten wir automatisch verschiedene Farben für die Marker.

```
m = ['+', 'o', '*']
for c in range(3):
    Xc = Xiris[yiris==c,:]
    plt.scatter(Xc[:,0], Xc[:,1], label='Klasse {}'.format(c), marker=m[c])
```

(Fortsetzung auf der nächsten Seite)

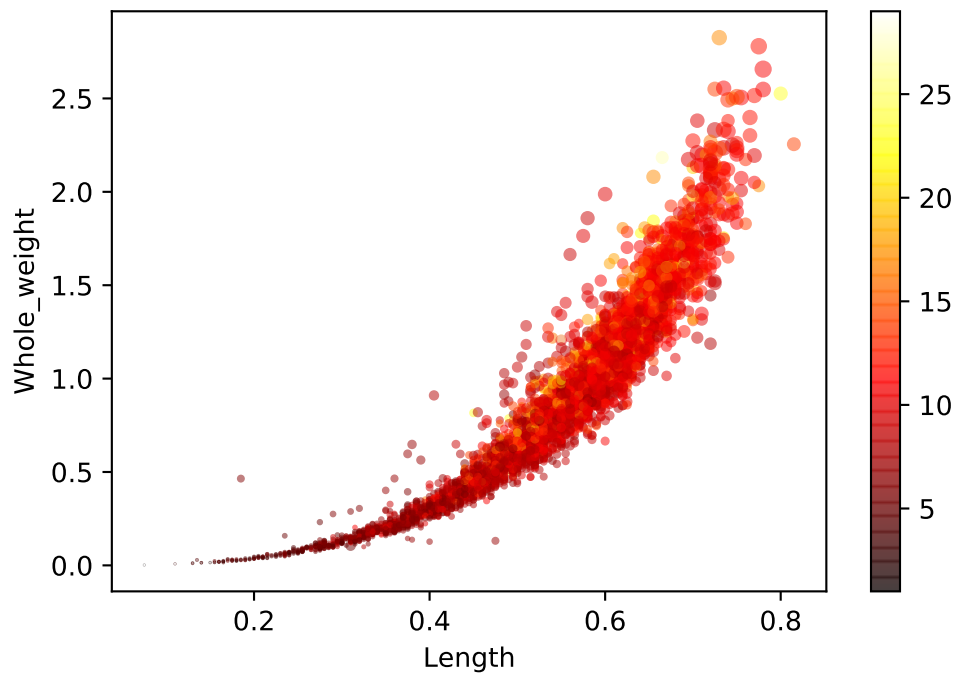


Abb. 10.7: Scattermatrix für 2 Komponenten von Abalone. Eine dritte Komponente ist als Punktgröße codiert.

(Fortsetzung der vorherigen Seite)

```
plt.legend()
plt.show()
```

Im Gegensatz zu Pandas stellt uns *Matplotlib* keine Funktion für eine Scatter Matrix zur Verfügung. Wir müssen diese selbst zusammensetzen.

Beispiel: Scatter Plot aller Komponentenpaare (*Scatter Matrix*) des Iris-Datensatzes. Wir ordnen die 4×4 Diagramme in einer Matrix an. Auf der Hauptdiagonale sehen wir die Abhängigkeit der Klasse von einer einzelnen Komponente.

```
plt.figure(figsize=(15, 15))
for i in range(4):
    for k in range(4):
        plt.subplot(4, 4, 4*k+i+1)
        if i == k:
            plt.hist(Xiris[:, i])
        else:
            plt.scatter(Xiris[:, i], Xiris[:, k], c=yIris)
plt.show()
```

Zum Analysieren einer einzelnen metrischen Komponente stehen uns Histogramm und Boxplot zur Verfügung. Das Histogramm unterteilt den Wertebereich in mehrere Klassen und zeigt an, wieviele Datenpunkte in die jeweilige Klasse fallen. Ein Boxplot kann die statistischen Daten – Quartile, Median, Minimum und Maximum – eines oder mehrerer ähnlicher Vektoren darstellen.

```
plt.hist(vektor)
plt.boxplot(matrix[, labels='spaltenNamen'])
```

Beispiel: Histogramme für den *Iris*-Datensatz.

```
plt.figure(figsize=(15, 5))
for i in range(4):
```

(Fortsetzung auf der nächsten Seite)

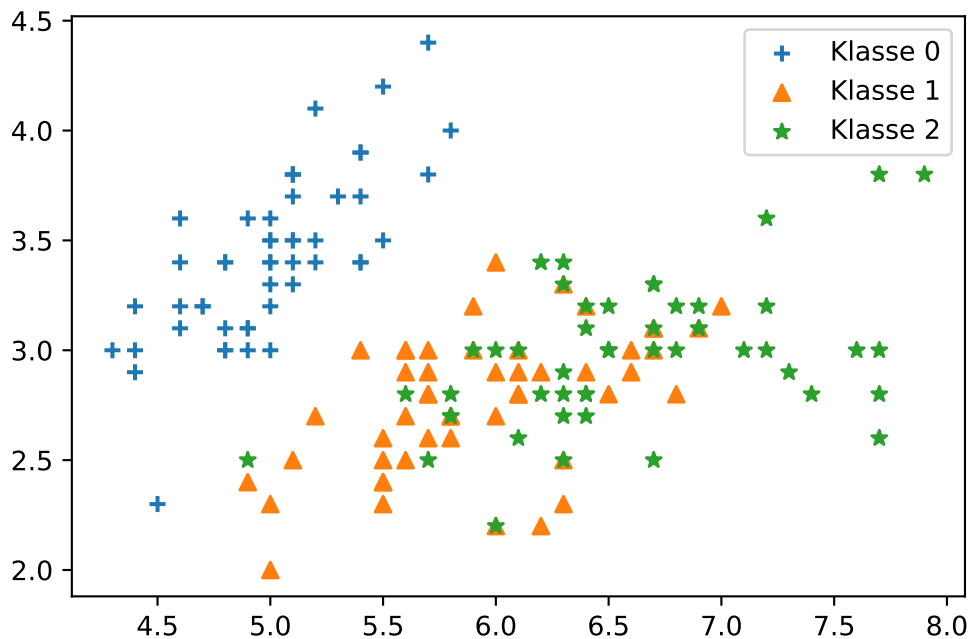
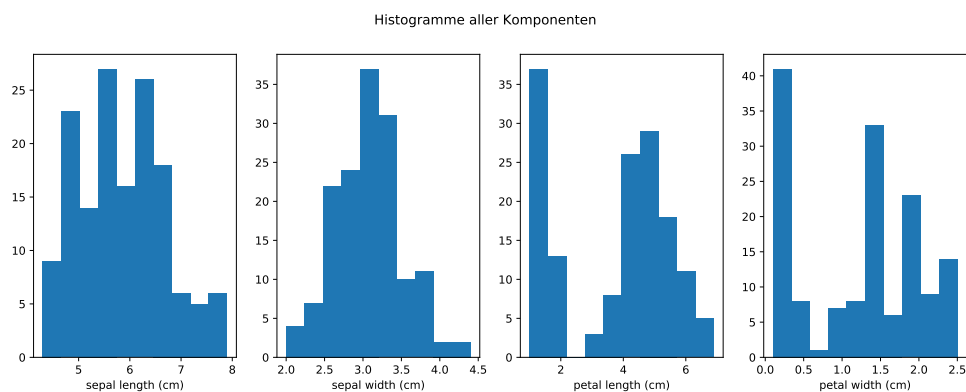


Abb. 10.8: Darstellung der Klassen durch verschiedene Symbole.

(Fortsetzung der vorherigen Seite)

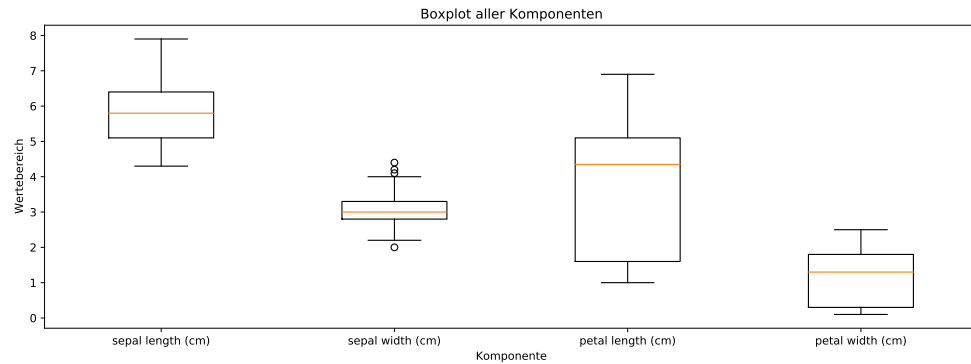
```
plt.subplot(1, 4, i+1)
plt.hist(Xiris[:, i])
plt.title(iris.feature_names[i])
plt.suptitle('Histogramme')
plt.show()
```

Beispiel: Boxplot für den *Iris*-Datensatz.

```
plt.boxplot(Xiris, labels=['sepal l', 'sepal w', 'pedal l', 'pedal w'])
plt.show()
```

Für die Darstellung nominaler Komponenten (Kategorien) ist ein Barplot besser als ein Histogramm geeignet, da dieses zum einen nur mit numerischen Klassenbezeichnern umgehen kann und zum anderen benachbarte Werte zu Klassen zusammenfasst. Wir müssen dazu die Anzahl der Kategorien und die Zahl der Werte je Kategorie bestimmen.

```
klassenListe, anzahlListe = np.unique(vektor, return_counts=True)
```



Das Plotten erfolgt nun mit der Funktion `bar()`.

`plt.bar(klassenListe, anzahlListe)`

Beispiel: Häufigkeit der Ringzahlen von *Abalone*.

```
rings, ringCounts = np.unique(yAbalone, return_counts=True)

plt.figure(figsize=(15,8))
plt.subplot(1, 2, 1)
plt.hist(yAbalone)
plt.subplot(1, 2, 2)
plt.bar(rings, ringCounts)
```

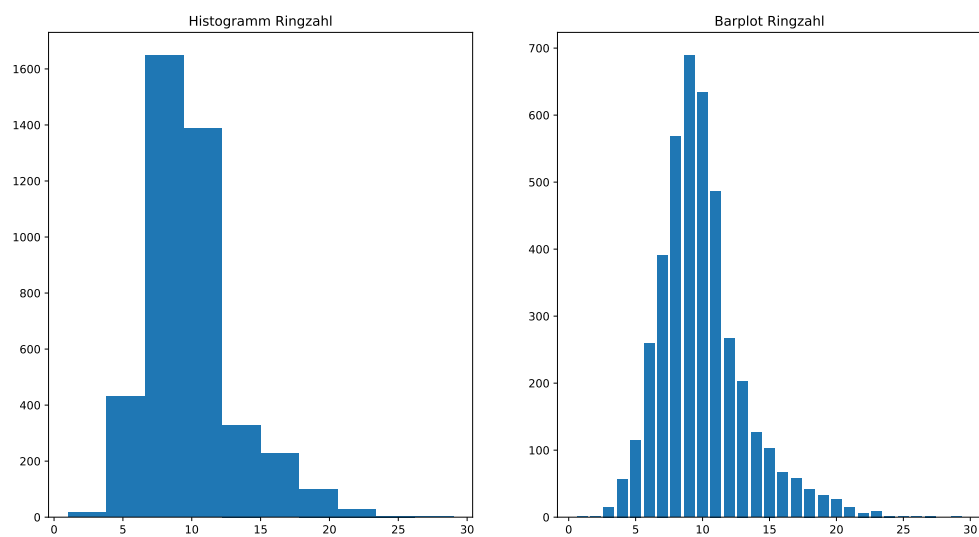


Abb. 10.9: Histogramm und Barplot der Ringhäufigkeiten.

Notebook

Notebook zum Erzeugen der obigen Diagramme: `iris-abalone-statplots.ipynb`.

10.6 Plotten 3-dimensionaler Daten

Wir haben schon beim *Scatter Plot* gesehen, wie wir eine dritte Dimension durch die Farbe darstellen können. Oft stehen wir vor der Aufgabe, einen Funktionswert an jedem Punkt eines rechteckigen Rasters darzustellen (z. B. Bilder) oder auch Funktionen zweier Veränderlicher darzustellen (z. B. Modellgüte in Abhängigkeit zweier Hyperparameter oder die Entscheidungsgrenzen zwischen den Klassen bei Daten mit 2 Komponenten).

Die Grundidee ist hier immer, die gewünschten Funktionswerte auf einem zweidimensionalen Raster zu berechnen und erzeugte Matrix geeignet darzustellen. Dies kann direkt mit der Funktion `plt.matshow()` bzw. `plt.imshow()` geschehen. Dabei ist `plt.matshow()` ein *Wrapper* für `plt.imshow()`, der einige Parameter geeignet umdefiniert.

`plt.matshow(matrix)`

Als weiteren Parameter können wir mit `cmap` eine *Color map* angeben, mit der Funktion `colorbar()` erhalten wir eine Farblegende.

Beispiel: Plotten der Korrelationskoeffizientenmatrix des *Abalone*-Datensatzes. Auf der Hauptdiagonale müssen natürlich Einsen stehen.

```
# Matrix muss transponiert werden
m = np.corrcoef(Xabalone.T)
plt.matshow(m)
# Achsen mit Merkmalsnamen beschriften
plt.xticks(range(len(colAbalone)), colAbalone, rotation='vertical')
plt.yticks(range(len(colAbalone)), colAbalone)
plt.colorbar()
```

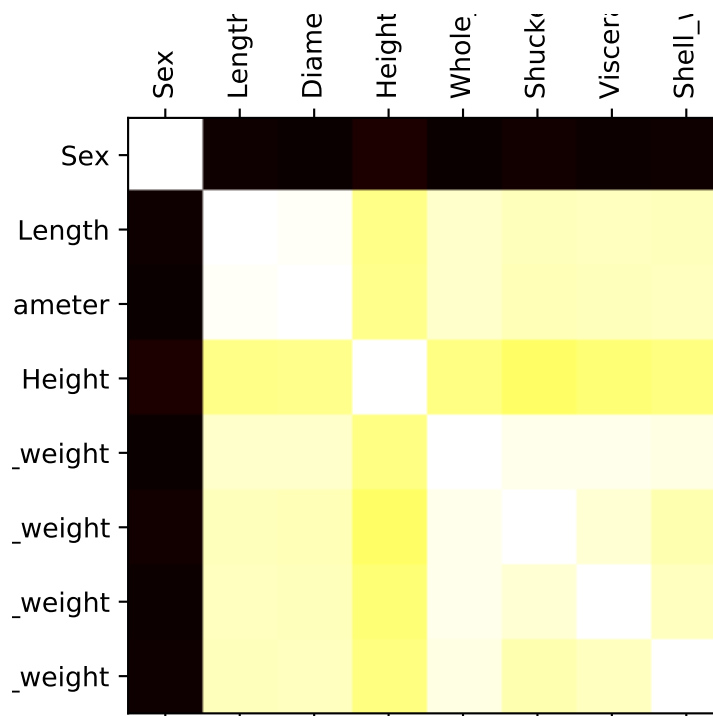


Abb. 10.10: Korrelationsmatrix Abalone

Die Funktion `plt.imshow()` kann auch sinnvoll zum Anzeigen von Bildern benutzt werden, die mit dem Paket `PIL` bzw. `pillow` eingelesen werden, da sie intern als *NumPy*-Arrays dargestellt werden. Wir können damit einfach eine Vorschau von Trainingsbildern für das Machine Learning erzeugen.

Beispiel: Ziffern aus *scikit-learn*-Datensatz *digits* anzeigen.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

digits = datasets.load_digits()
Xdigits = digits['data']
yDigits = digits['target']
shuffleIdx = np.random.permutation(Xdigits.shape[0])
Xplot = Xdigits[shuffleIdx[:25], :]
yPlot = yDigits[shuffleIdx[:25]]

plt.figure(figsize=(20,20))
for i in range(25):
    plt.subplot(5, 5, i+1)
    # Bilddaten von Vektor der Länge 64 in 8x8-Matrix umwandeln
    plt.imshow(Xplot[i].reshape(8,8), cmap='gray')
    plt.title('Ziffer: {}'.format(yPlot[i]))
plt.show()
```

Für stetige Daten bieten sich Kontourdiagramme an. Wir müssen dazu zunächst ein Messgitter (*Mesh grid*) aus unserem X- und Y-Intervall erzeugen.

```
x = np.linspace(xMin, xMax, anzahl)
y = np.linspace(yMin, yMax, anzahl)
xMesh, yMesh = np.meshgrid(x, y)
```

Nun berechnen wir die Funktionswerte in jedem Gitterpunkt.

```
zMesh = f(xMesh, yMesh)
```

Beispiel: Funktion $f(x) = x^2 + y^2 + \frac{5}{y^2+1} - y$.

```
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
xx, yy = np.meshgrid(x, y)

zz = xx*xx + 0.5*yy*yy + 5/(yy*yy+1) - yy
```

Diese Funktion kann nun mit `plt.contour()` (Niveaulinien) bzw. `plt.contourf()` (gefüllte Flächen) dargestellt werden.

```
plt.contour(xMesh, yMesh, zMesh[, levels=farbstufen])
plt.contourf(xMesh, yMesh, zMesh[, levels=farbstufen])
```

Auch hier kann mit `cmap` eine Palette angegeben und mit der Funktion `plt.colorbar()` die Farbzuoordnung angezeigt werden.

Eine echte 3-D-Darstellung ist ebenfalls möglich und soll hier kurz vorgestellt werden, obwohl wir diese nicht weiter nutzen werden. Wir müssen dazu das Modul `mplot3d` aus dem Paket `mpl_toolkits` laden.

```
from mpl_toolkits.mplot3d import Axes3D
```

Die damit bereitgestellten Funktionen benötigen das Objekt-Interface von *Matplotlib*, das Vorgehen weicht gegenüber dem bisherigen etwas ab.

Wir erzeugen eine neue Abbildung als Subplot-Objekt

```
fig = plt.figure(...)
```

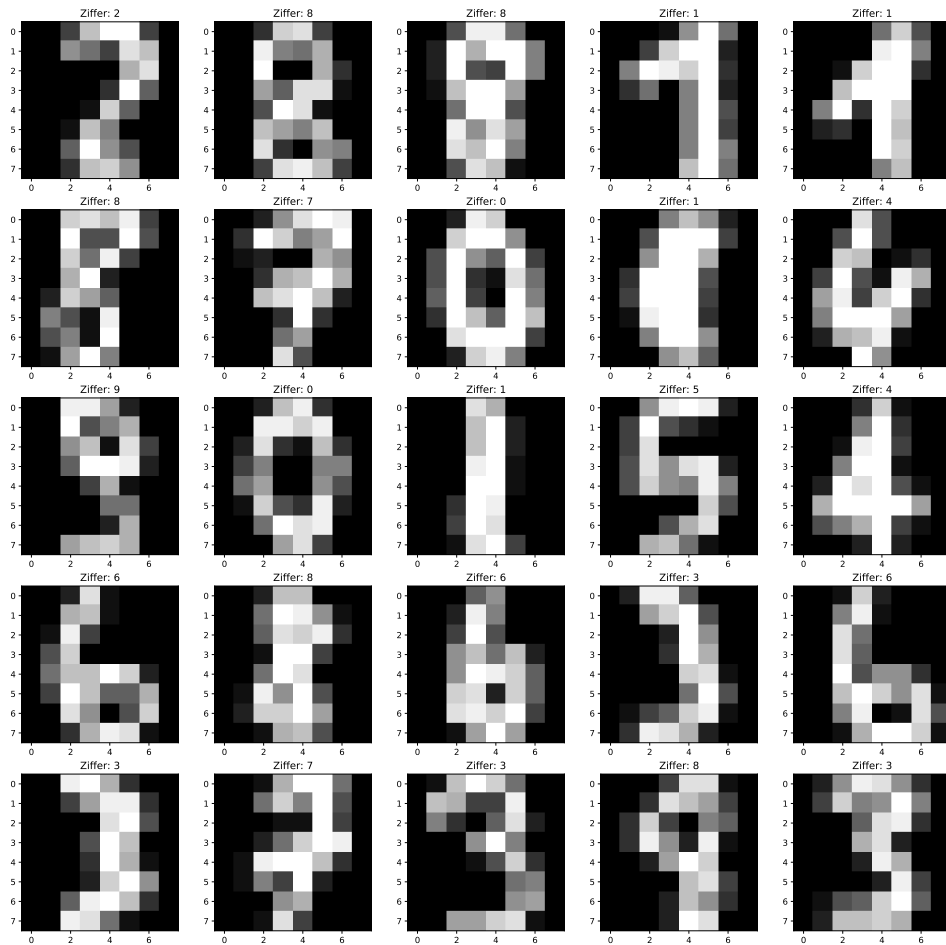
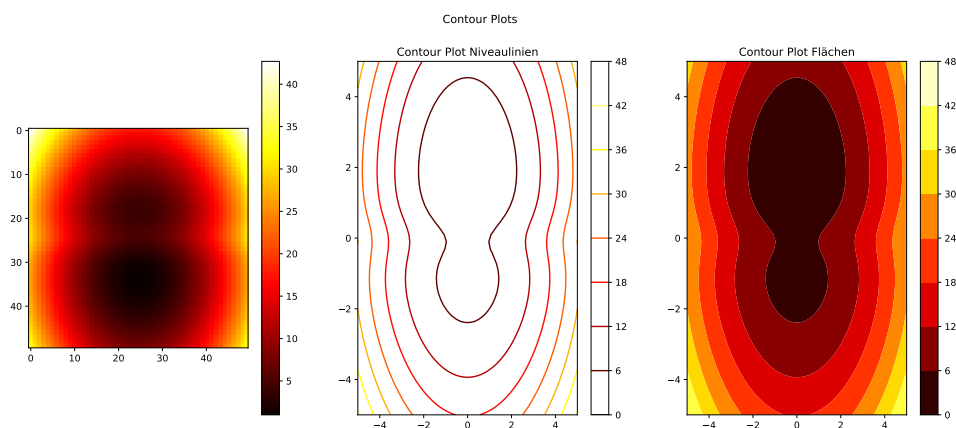


Abb. 10.11: Einige Ziffern aus dem Digits-Datensatz

Abb. 10.12: Heatmap und zwei Konturplots für die Funktion $f(x) = x^2 + y^2 + \frac{5}{y^2+1} - y$.

```
ax = fig.add_subplot(zeilenZahl, spaltenZahl, nummer, projection='3d')
```

Das Zeichnen geschieht nun mit Methoden des Achsen-Objekts `ax`. Wir können entweder geschlossene Oberflächen oder ein Drahtmodell zeichnen.

```
ax.plot_surface(xMesh, yMesh, zMesh[, cmap= 'palette'])
ax.plot_wireframe(xMesh, yMesh, zMesh)
```

Beispiel. Plotten unserer obigen Funktion.

```
fig = plt.figure(figsize=(15, 7))
ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_surface(xx, yy, zz, cmap='magma')
ax = fig.add_subplot(2, 1, 2, projection='3d')
ax.plot_wireframe(xx, yy, zz)
```

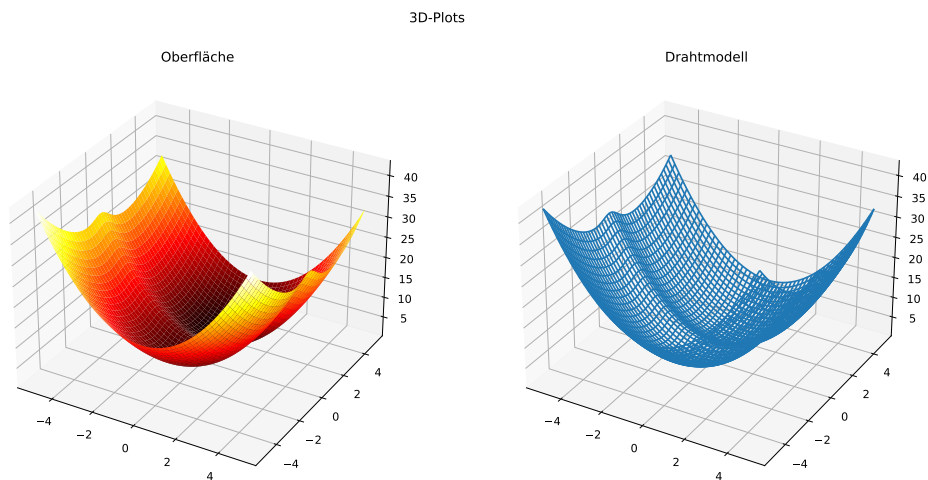


Abb. 10.13: 3D-Plots der Funktion $f(x) = x^2 + y^2 + \frac{5}{y^2+1} - y$.

Notebook

Notebook zum Erzeugen der 3D-Plots: `3d_plots.ipynb`.

10.7 Aufgaben

Aufgabe 10.1 Wählen Sie einen Datensatz aus dem [UCI Learning repository](#) mit Fließkomma- und Kategoriedaten aus. Beschränken Sie die Zahl der Datensätze auf ca. 500. Plotten Sie zwei gut korrelierte Features als Scatterplot und färben Sie die Datenpunkte gemäß der Zielgröße ein. Plotten Sie die Häufigkeit eines kategorialen Merkmals als Barplot.

Klassifikationsverfahren

- *Grundlagen*
- *Dimensionsreduktion*
- *Allgemeines Vorgehen*
- *K-Nearest-Neighbors*
- *Naive-Bayes-Classifer*
- *Logistische Regression*
- *Weitere Verfahren*
- *Mehrfachklassifikation*
- *Persistenz*
- *Aufgaben*

11.1 Grundlagen

Die *Klassifikation* gehört zu den Verfahren des *Supervised Learning*. Ziel ist es, Daten anhand von bekannten Eigenschaften (Features, Komponenten) in verschiedene Klassen einzuteilen.

Unser Klassifikationsverfahren muss also anhand der Trainingsdaten aus den Komponenten der Datenpunkte die richtige Klasse ableiten. Wir gehen davon aus, dass ähnliche Datenpunkte, i. e. deren Komponenten ähnliche Werte haben, der gleichen Klasse angehören. Diese Ähnlichkeit wird meist über den *euklidischen Abstand* definiert, weshalb alle unsere Komponenten einen numerischen Typ haben müssen und die Werte in einer vergleichbaren Größenordnung liegen.

Um die Qualität unseres Modells bewerten zu können, trennen wir von den Daten mit bekannten Klassen eine feste Testmenge ab, die niemals in den Trainingprozess einfließt. Für diese Testmenge vergleichen wir die Modellvorhersage mit den tatsächlichen Klassen und erhalten damit ein Genauigkeitsmaß.

Um einen optischen Eindruck von der Funktionsweise der verschiedenen Verfahren zu erhalten, bietet es sich an, mit einer zweidimensionalen Datenmenge zu arbeiten. Wir können dann die Klassen und Entscheidungsgrenzen durch einen Konturplot visualisieren.

Wir beschränken uns in diesem Kapitel auf einfache klassische Klassifikationsverfahren.

11.2 Dimensionsreduktion

Um unsere Daten sinnvoll darstellen zu können, beschränken wir uns auf zwei Features. Wir könnten dazu die beiden Features mit dem größten Einfluss auf die Zielgröße auswählen, die wir über die Korrelationsmatrix bestimmen können (*Datenanalyse*). Wir wollen hier jedoch eine Methode des *Unsupervised Learning* kennen lernen, die uns zwei künstliche neue Dimensionen erzeugt, die voneinander stochastisch unabhängig sind und einen größeren Einfluss auf die Zielgröße haben.

Generell wird die *Dimensionsreduktion* eingesetzt, um die Zahl der Features auf die wirklich notwendigen zu begrenzen und damit die Modelle zu verkleinern, die dann auch schneller trainiert werden können.

Das verwendete Verfahren ist die *PCA* (Hauptkomponentenanalyse, Hauptachsentransformation): Wir bestimmen die Kovarianzmatrix Σ unserer Komponenten und bestimmen ihre Eigenwerte λ_i und die Eigenvektoren $\mathbf{g}^i$, die wir zur Matrix Γ anordnen. Transformieren wir nun unsere Datenpunkte mit dieser Matrix in das Koordinatensystem der Eigenvektoren, dann ist unsere Kovarianzmatrix eine Diagonalmatrix. Wir wählen nun die beiden transformierten Komponenten mit den größten Varianzen (also den größten Einträgen der Kovarianzmatrix) aus und verwenden diese.

Vor der Berechnung der Transformation sollten wir unbedingt alle Komponenten skalieren, da unterschiedliche Wertebereiche einen starken Einfluss auf die Kovarianzen haben. Da diese Operationen immer gemeinsam notwendig sind, bietet es sich an, diese beiden Operationen zu einer *Pipeline* zusammenzufassen:

```
from sklearn import pipeline

transformation = pipeline.Pipeline([
    ('name', TransformerObjekt1),
    ('name', TransformerObjekt2),
    ...
])

transformation.fit(trainingsDaten)
trainingTransformiert = transformation.transform(trainingsDaten)
testTransformiert = transformation.transform(testDaten)
```

Wir erzeugen eine *PCA*-Transformation mit der Angabe der gewünschten Zahl der Hauptkomponenten:

```
from sklearn import decomposition

pcaTransform = decomposition.PCA(n_components=anzahl)
```

Weitere Parameter betrachten wir hier nicht. Lassen wir den Parameter `n_components` weg, erhalten wir die vollständige Transformation, also genauso viele Komponenten wie unsere Ausgangsdaten. Allerdings sind die Komponenten nun paarweise stochastisch unabhängig, was wir durch Berechnung der Kovarianzmatrix leicht prüfen können:

```
np.cov(data.T)
```

Berechnen wir statt dessen die Korrelationsmatrix, entsteht die Einheitsmatrix, da die Varianzen alle auf 1 skaliert werden.

Vergleichen wir die Wertebereiche unserer transformierten Komponenten, stellen wir ebenfalls fest, dass diese recht unterschiedlich sind. Da diese jedoch die Stärke des Einflusses der Komponente unmittelbar beschreiben, skalieren wir *nicht* nochmals.

Beispiel: Skalieren und Transformieren des *Abalone* Data Frames mit *PCA* und Berechnen der Kovarianzen.

Beispiel: PCA für Datensatz *Abalone* mit allen und reduziert auf 2 Komponenten.

Quellcode 11.1: abalone-pca.py: PCA für Abalone

```
#!/usr/bin/env python
# coding: utf-8

# # Dimensionsreduktion mit PCA

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import pipeline, preprocessing, decomposition

# Einlesen der Daten, Anzeige der Spaltennamen.

dfTrain = pd.read_csv('abalone-train.csv')
dfTest = pd.read_csv('abalone-test.csv')
print('Spalten:', dfTrain.columns)

# Auswahl der numerischen Feature-Spalten. Die Zielgröße wird nicht benötigt.

numCols = ['length', 'diameter', 'height', 'whole', 'shucked', 'viscera', 'shell']
dfTrainNum = dfTrain[numCols]
dfTestNum = dfTest[numCols]

# Anzeige von Kovarianz und Korrelation der Features der Trainingsdaten.
# Wir verwenden hier die *NumPy*-Funktionen.
#
# Alle Spalten sind stark korreliert.

print('Kovarianz')
print(np.around(np.cov(dfTrainNum.values.T), 2))
print('Korrelation')
print(np.around(np.corrcoef(dfTrainNum.values.T), 2))
plt.imshow(np.cov(dfTrainNum.values.T), cmap='hot')
plt.colorbar()
plt.show()

# Pipeline zum Skalieren und anschließenden Transformieren der Daten.
# Wir erhalten zunächst alle Features.

trAll = pipeline.Pipeline([
    ('scale', preprocessing.StandardScaler()),
    ('pca', decomposition.PCA()),
])
# Training und Transformation der Trainingsdaten
XtrainAll = trAll.fit(dfTrainNum).transform(dfTrainNum)
# Transformation der Testdaten mit trainiertem Filter.
XtestAll = trAll.transform(dfTestNum)

# Anzeige der Kovarianzen der Trainings- und Testdaten. Die letzten drei
# ↪ Komponenten haben kaum einen Einfluss auf ein zukünftiges Modell.
#
# Auch die Kovarianzmatrix der Testdaten ist auf die Diagonale konzentriert, deren
# ↪ Verteilung entspricht also der der Trainingsdaten.

print('Kovarianz Training (tr)')
print(np.around(np.cov(XtrainAll.T), 2))
```

(Fortsetzung auf der nächsten Seite)

```

print('Kovarianz Test (tr)')
print(np.around(np.cov(XtestAll.T), 2))
plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
plt.imshow(np.cov(XtrainAll.T), cmap='hot')
plt.colorbar()
plt.subplot(1, 2, 2)
plt.imshow(np.cov(XtestAll.T), cmap='hot')
plt.colorbar()
plt.show()

# Reduktion auf die beiden Features mit der größten Kovarianz (Hauptkomponenten).

tr2 = pipeline.Pipeline([
    ('scale', preprocessing.StandardScaler()),
    ('pca', decomposition.PCA(n_components=2)),
])
Xtrain = tr2.fit(dfTrainNum).transform(dfTrainNum)
Xtest = tr2.transform(dfTestNum)

# Anzeige der Kovarianzen für Trainings- und Testdaten. Wir sollten die
# gleichen Werte wie in der 2x2-Teilmatrix der Kovarianzmatrix der vollständigen
# Transformation erhalten.

print('Kovarianz Training (2 Komp.)')
print(np.around(np.cov(Xtrain.T), 2))
print('Kovarianz Test (2 Komp.)')
print(np.around(np.cov(Xtest.T), 2))

```

Download Notebook: `abalone-pca.ipynb`.

Wir übernehmen nun diese Vorverarbeitung in einen Python-Modul, um diese einfach für jedes der folgenden Verfahren einsetzen zu können.

Download: `abalone_helper2.py`.

Unsere beiden transformierten Komponenten liegen nun im wesentlichen in den Intervallen $[-6, 6]$ bzw. $[-1.5, 1.5]$.

Warnung: Die *PCA* untersucht lediglich die Abhängigkeiten zwischen den Komponenten, aber nicht den Einfluss der Komponenten auf die Zielgröße, die ja in der Berechnung nicht auftaucht. Sie ist also nur sinnvoll zur Eliminierung redundanter Komponenten, die keine weitere Information beitragen.

11.3 Allgemeines Vorgehen

Die Arbeitsweise für alle Klassifikationsverfahren ist identisch: Wir initialisieren ein Modellobjekt mit den gewünschten *Hyperparametern*, trainieren es mit den Trainingsdaten und bewerten anschließend die Modellgüte sowohl für Trainings- als auch Testdaten. Sind wir zufrieden, können wir nun die Klassen für neue Daten vorhersagen.

Initialisierung:

```
classifier = MyClassifier(hyperparameter)
```

Training. Die Datenpunkte liegen als Matrix numerischer Werte vor, jede Zeile ist ein Datenpunkt, jede Spalte eine Komponente.

```
classifier.fit(trainingsdatenMatrix, trainingLabelVektor)
```

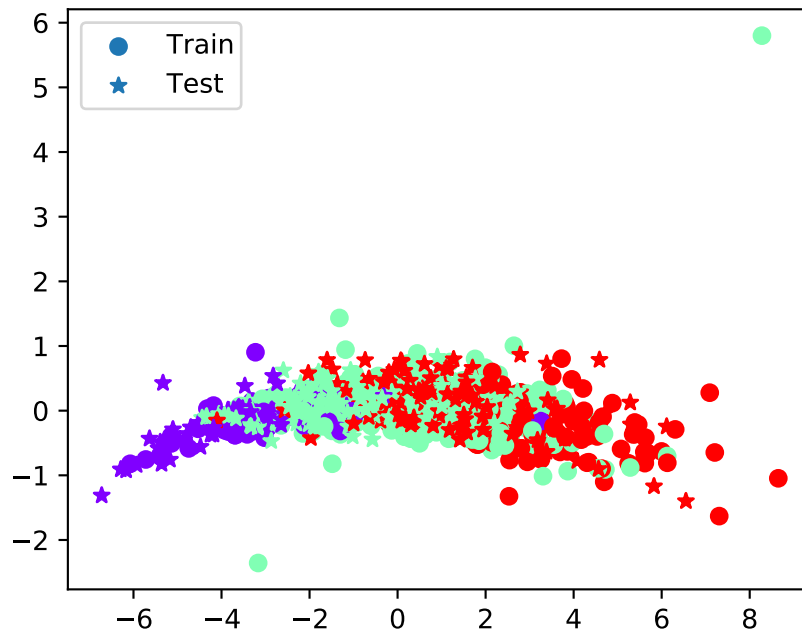


Abb. 11.1: Verteilung der transformierten Abalone-Daten.

Die erlernten Parameter des Modells werden über die Eigenschaft `parameters_` bereitgestellt und können so analysiert werden.

Bewerten des Modells:

```
scoreTraining = classifier.score(trainingsDatenMatrix, trainingLabelVektor)
scoreTest = classifier.score(testDatenMatrix, testLabelVektor)
```

Wir können diese Bewertung auch manuell vornehmen, indem wir die Vorhersagemethode `predict()` des Modells verwenden und diesen Vorhersagevektor mit den bekannten Daten vergleichen.

```
yTestPredict = classifier.predict(testDaten)
countTestCorrect = np.count_nonzero(yTestPredict == testLabelVektor)
```

Um neue Daten zu klassifizieren, bringen wir diese ebenfalls in Matrixform, führen mit ihnen die trainierte Vorverarbeitung durch und erzeugen die Klassenvorhersage mit der Methode `predict()`.

```
vorhersageVektor = classifier.predict(neueDatenMatrix)
```

Als Beispiel wählen wir wieder unseren *Abalone*-Datensatz, den wir bereits in Trainings- und Testdaten zerlegt haben. Wir untersuchen zum einen den kompletten Datensatz mit allen Komponenten, die wir entweder skaliert bzw. in One-Hot-Darstellung umcodiert haben.

Um die Entscheidungsgrenzen darstellen zu können, reduzieren wir zum anderen die Dimensionen auf 2 und reduzieren auch die Anzahl der Datenpunkte.

Für beide Varianten wird ein entsprechendes Modell erzeugt und bewertet. Für den zweidimensionalen Datensatz erzeugen wir eine Graphik, die neben den eingefärbten Datenpunkten auch die Entscheidungsgrenzen darstellt. Dazu erzeugen wir ein zweidimensionales Raster über unserem Wertebereich der Daten und erstellen für jeden Datenpunkt des Rasters eine Vorhersage.

```
xlim = np.linspace(-6, 6)
ylim = np.linspace(-2, 2)
xx, yy = np.meshgrid(xlim, ylim)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
# beide Matrizen in n×1-Matrizen umwandeln und als Spalten einer
# n×2-Matrix verwenden
xxCol = xx.reshape(-1, 1)
yyCol = yy.reshape(-1, 1)
xy = np.hstack(xxCol, yyCol)
# Vorhersage als Vektor berechnen
z = classifier.predict(xy)
# Wieder in 2dimensionale Matrix (Raster) umwandeln
zz = z.reshape(xx.shape)
# Plot
plt.countourf(xx, yy, zz)
```

In dieser Graphik ergänzen wir je einen Scatterplot für unsere zweidimensionalen Trainings- und Testdaten mit ihren tatsächlichen Labels.

Diese Funktionalität legen wir wieder zur einfachen Verwendung in einem Python-Script ab: `plot_decision.py`.

Beispiel: Demonstration dieses Vorgehens anhand eines Entscheidungsbaummodells.

Quellcode 11.2: `abalone_decision_workflow.py`: Ablauf Untersuchung Entscheidungsgrenzen

```
#!/usr/bin/env python
# coding: utf-8

# # Vorgehen Darstellung Entscheidungsgrenzen
#
# Beispiel: Entscheidungsbaum

import numpy as np
import matplotlib.pyplot as plt
from sklearn import compose, pipeline, preprocessing, metrics, decomposition
import abalone_helper2, plot_decision
from sklearn import tree, neighbors, linear_model

# Vorbereiten aller Daten: Xtrain/yTrain, Xtest/yTest
#
# Reduktion auf 2 Dimensionen und der Anzahl der Datensätze:
# Xtrain2/yTrain2, Xtest2/yTest2

Xtrain, Xtest, yTrain, yTest = abalone_helper2.getData()
Xtrain2, Xtest2, yTrain2, yTest2 = abalone_helper2.getData2(0.2)
print('Alle Daten:', Xtrain.shape, yTrain.shape, Xtest.shape, yTest.shape)
print('Reduziert:', Xtrain2.shape, yTrain2.shape, Xtest2.shape, yTest2.shape)

# Verteilung der reduzierten Daten.

plt.figure(figsize=(5, 4))
plt.scatter(Xtrain2[:, 0], Xtrain2[:, 1], c=yTrain2, label='Train', cmap='rainbow')
plt.scatter(Xtest2[:, 0], Xtest2[:, 1], marker='*', c=yTest2, label='Test', cmap=
↪ 'rainbow')
plt.legend()
plt.savefig('abalone-features2.svg')
plt.savefig('abalone-features2.pdf')

# Verteilung der Zielgröße entsprechend der Reihenfolge der Datensätze.

print('Verteilung Zielgröße Training:',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

    np.unique(yTrain2, return_counts=True)[1])
print('Verteilung Zielgröße Test:',
      np.unique(yTest2, return_counts=True)[1])
plt.subplot(1, 2, 1)
tTrain = np.arange(yTrain2.shape[0])
plt.scatter(tTrain, yTrain2)
plt.title('Training')
plt.subplot(1, 2, 2)
tTest = np.arange(yTest2.shape[0])
plt.scatter(tTest, yTest2)
plt.title('Test')
plt.suptitle('Verteilung Zielgröße')
plt.show()

# Erzeugen und Bewerten eines Modells für alle Daten/Features.

classifier = tree.DecisionTreeClassifier()
classifier.fit(Xtrain, yTrain)
print('Alle Features: ', classifier.score(Xtrain, yTrain),
      classifier.score(Xtest, yTest))

# Erzeugen und Bewerten eines äquivalenten Modells für den reduzierten
# Datensatz mit 2 Features.

classifier2 = tree.DecisionTreeClassifier()
classifier2.fit(Xtrain2, yTrain2)
print('2 Features: ', classifier2.score(Xtrain2, yTrain2), classifier2.score(Xtest2,
↪ yTest2))

# Graphische Darstellung der Entscheidungsgrenzen für den reduzierten,
# zweidimensionalen Datensatz.

plt.figure(figsize=(5,4))
plot_decision.plotDecision(classifier2, Xtrain2, yTrain2, Xtest2, yTest2,
                           'Decision Tree', 'abalone-dt')

```

Download Notebook: [abalone_decision_workflow.ipynb](#).

11.4 K-Nearest-Neighbors

Eines der einfachsten Klassifikationsverfahren ist das *KNN*-Verfahren. Wir legen eine Zahl k von Nachbarpunkten als *Hyperparameter* fest. Unser Modell sind einfach alle Trainingspunkte mit ihrer bekannten Klasse. Um einen neuen Punkt einer neuen Klasse zuzuordnen, bestimmen wir seine k nächsten Nachbarn und wählen die Klasse, die unter diesen am häufigsten auftritt.

```
from sklearn.neighbors import KNeighborsClassifier
```

```
classifier = KNeighborsClassifier(n_neighbors=nachbarZahl)
```

Ohne Angabe der Anzahl werden 5 Nachbarn verwendet.

Beispiel: *KNN* für *Abalone* mit Vorgabewert von 5 Nachbarn.

Wir erreichen folgende Genauigkeiten:

- Training (alle Features): 75%

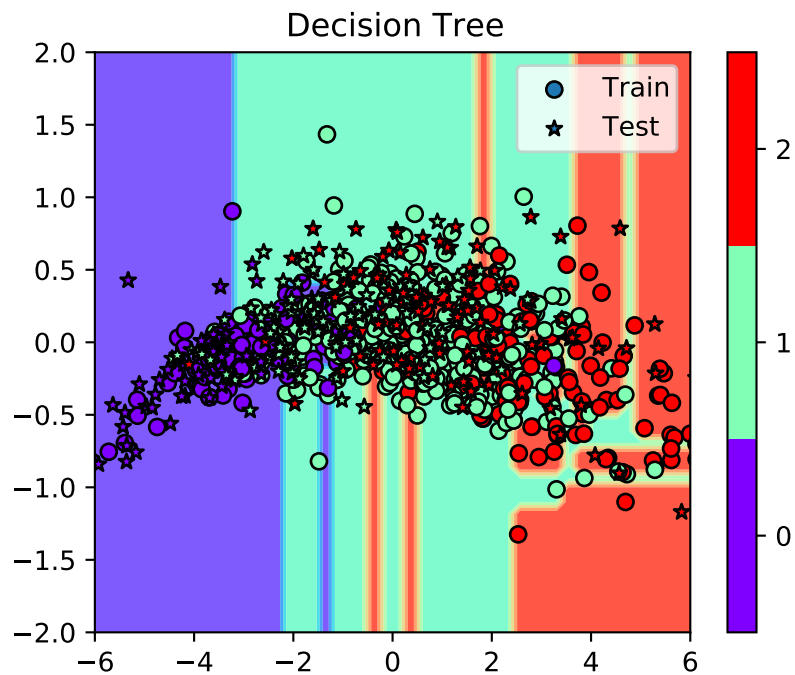
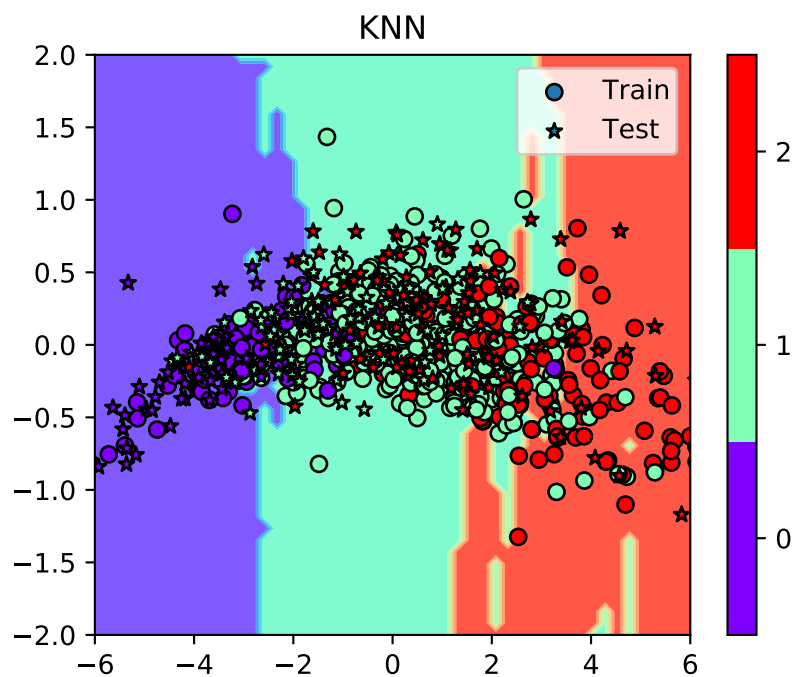


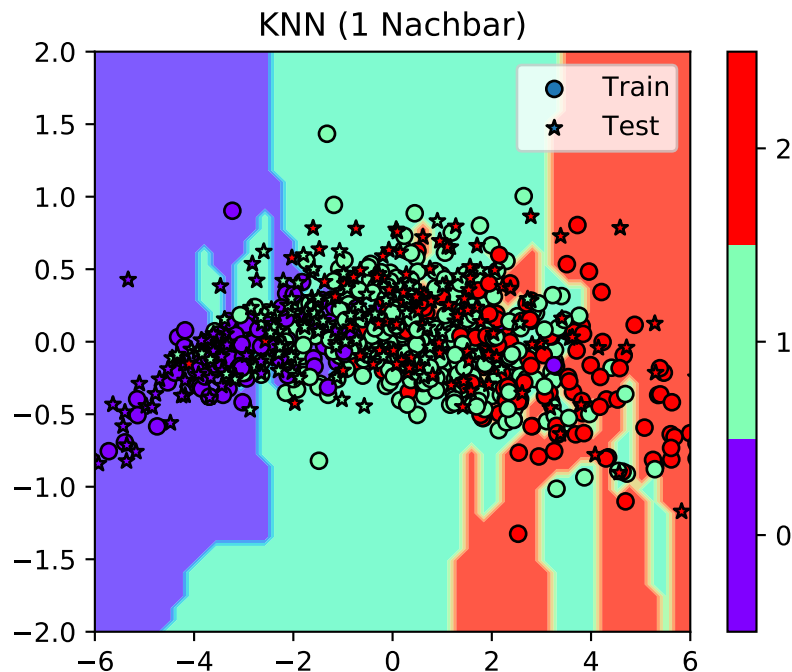
Abb. 11.2: Entscheidungsgrenzen Decision Tree (Hyperparameter sind die automatischen Vorgabewerte)



- Test (alle Features): 64%
- Training (2 Features): 78%
- Test (2 Features): 49%

Unser einziger Hyperparameter ist die Zahl der Nachbarn. Je mehr Nachbarn wir einbeziehen, desto besser generalisiert das Modell, zufällige Fehler werden ausgeglichen. Verwenden wir dagegen weniger Nachbarn, wird das Modell empfindlicher und stärker abhängig von einzelnen Ausreißern, die die Vorhersage massiv verfälschen können.

Beispiel: 1 Nachbar.



Wir erreichen folgende Genauigkeiten:

- Training (alle Features): 100%
- Test (alle Features): 58%
- Training (2 Features): 100%
- Test (2 Features): 50%

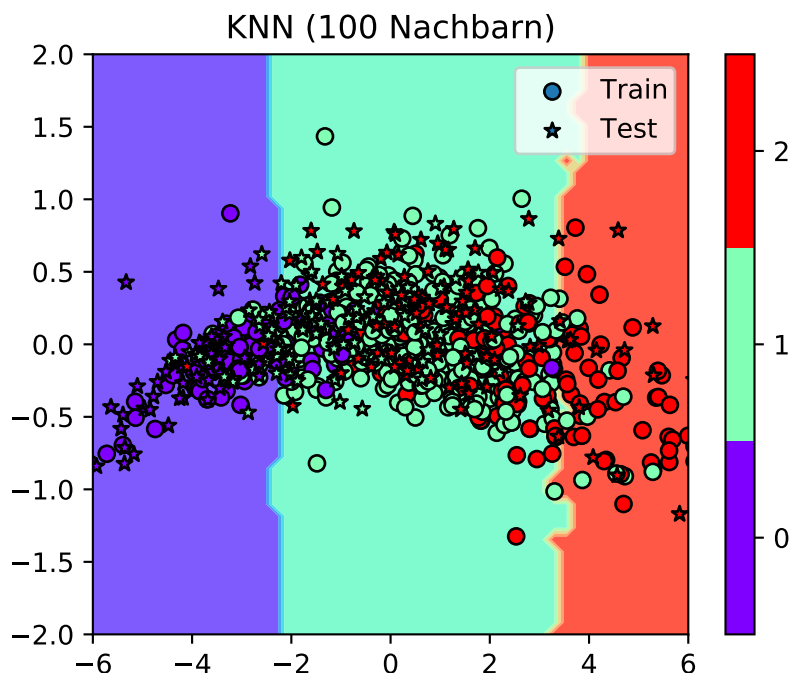
Wählen wir dagegen zuviele Nachbarn, generalisiert unser Modell zu stark. Diskriminierende Features wegen zu stark geglättet, die Klassifikation wird auf Trainings- und Modelldaten schlechter.

Beispiel: 100 Nachbarn.

Wir erreichen folgende Genauigkeiten:

- Training (alle Features): 68%
- Test (alle Features): 61%
- Training (2 Features): 75%
- Test (2 Features): 50%

Die Abbildung zeigt eine starke Glättung der Entscheidungsgrenzen, die Modellqualität für die Trainingsdaten hat sich verschlechtert, ohne die Qualität auf den Testdaten zu verbessern.



Der *KNN*-Klassifizierer beherrscht verfahrensbedingt eine Mehrfachklassifikation (wir werten ja die Klassen der benachbarten Datenpunkte direkt aus und entscheiden uns für die am häufigsten auftretende Klasse), allerdings müssen alle Komponenten einheitlich skaliert werden, um keine davon zu bevorzugen.

Die Hauptnachteile dieses Verfahrens sind das große Modell (das aus allen Trainingsdaten besteht), die Gefahr des *Overfittings* sowie die geringe Datenabstraktion. Es werden für die Vorhersage stets alle Datenpunkte benötigt.

Notebook

Download: `abalone_knn.ipynb`.

11.5 Naive-Bayes-Classifier

Die Naive-Bayes-Klassifikation wird schon seit längerer Zeit erfolgreich für hochdimensionale Daten, wie z. B. der SPAM-Erkennung bei E-Mails eingesetzt. Vereinfacht sind unsere Spalten eine riesige Liste von häufigen Wörtern. Jede E-Mail produziert einen Datensatz, der für jedes dieser Wörter die Anzahl ihres Auftretens enthält.

Die Grundidee des Verfahrens ist der *Satz von Bayes* über bedingte Wahrscheinlichkeiten:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Beispiel: Wir wollen einschätzen, ob eine Mail *Spam* (A) ist, in der das Wort *Dollar* (B) vorkommt (bedingte Wahrscheinlichkeit $P(A|B)$). Aus vergangenen Mails wissen wir, wie hoch der Anteil der Mails ist, die das Wort *Dollar* enthalten ($P(B)$) und wie hoch der Anteil der Spam-Mails an allen Mails ist ($P(A)$). Ebenso wissen wir, wie hoch der Anteil der Spam-Mails ist, die das Wort *Dollar* enthalten ($P(B|A)$). Daraus errechnen wir die Wahrscheinlichkeit, dass die neue Mail Spam ist.

Diese Idee führen wir nun nicht nur für ein, sondern für viele Wörter durch und betrachten dabei auch nicht nur, ob das Wort überhaupt auftritt, sondern auch die Häufigkeit seines Auftretens.

Je nach Datentyp unserer Komponenten verwenden wir unterschiedliche Varianten des *Naive-Bayes*-Verfahrens.

GaussianNB Die Komponenten sind metrische Größen.

MultinomialNB Die Komponenten sind Klassenhäufigkeiten (z. B. Worthäufigkeiten in Texten).

BernoulliNB Die Komponenten sind binär, also nehmen nur die Werte 0 oder 1 an.

Wir können dieses Verfahren also nur dann anwenden, wenn alle unsere Komponenten den gleichen Typ haben.

In *scikit-learn* verwenden wir die Naive-Bayes-Klassifikation wie folgt:

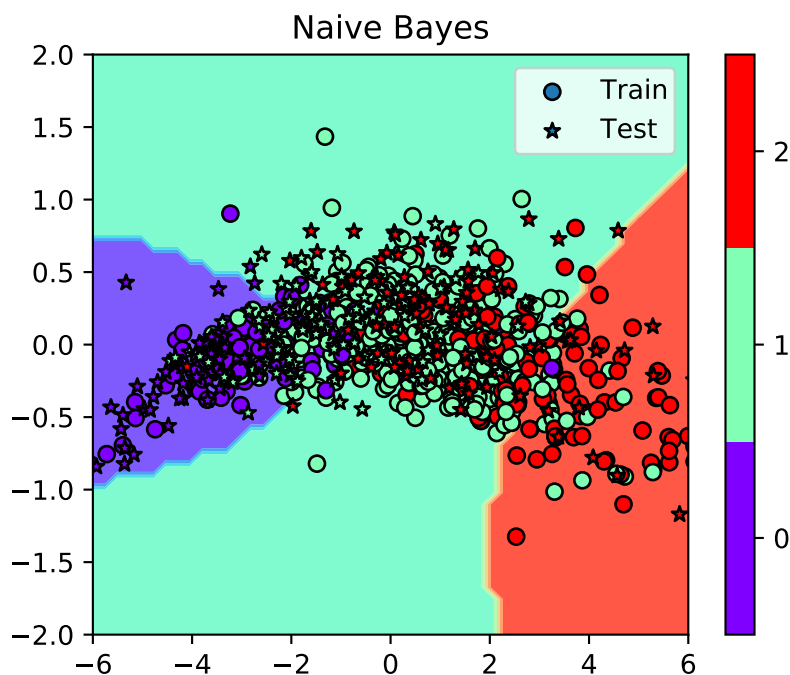
```
from sklearn import naive_bayes

metricClassifier = naive_bayes.GaussianNB()
catClassifier = naive_bayes.MultinomialNB(parameter)
binfeatureClassifier = naive_bayes.BernoulliNB(parameter)
```

Die beiden Kategoriemodelle können mit einem Parameter `alpha` gesteuert (geglättet bzw. regularisiert) werden, wobei der Wert 0 keine Glättung bewirkt.

Neben der Methode `predict()` für die Klassenvorhersage finden wir hier auch eine Methode `predict_proba()`, die uns die Wahrscheinlichkeiten für die Zugehörigkeit zu jeder Klasse liefert. Das Ergebnis ist also eine $N \times K$ -Matrix bei N Datenpunkten und K Klassen.

Beispiel: Da unser Datensatz *Abalone* neben numerischen auch eine kategoriale Spalte enthält, lassen wir diese hier bei der Klassifikation weg, um den `GaussianNB` Classifier anwenden zu können.



Wir erreichen folgende Genauigkeiten:

- Training (alle Features): 60%
- Test (alle Features): 55%
- Training (2 Features): 74%
- Test (2 Features): 48%

Im Gegensatz zum *KNN* ist das Modell relativ klein und seine nicht von der Zahl der Datensätze, sondern nur von der Anzahl der Features und Klassen abhängig. Die Gefahr eines Overfittings ist recht gering. Allerdings werden recht viele Datensätze benötigt, um ein Underfitting zu vermeiden.

Da der Naive-Bayes-Klassifikator implizit davon ausgeht, dass alle Komponenten stochastisch unabhängig sind, kann es sinnvoll sein, diese vorher per *PCA* zu transformieren. Bei einer großen Zahl von Komponenten ist das jedoch nicht immer praktikabel. Eine Skalierung der Komponenten ist jedoch nicht unbedingt erforderlich, da jede Komponente getrennt betrachtet wird.

Notebook

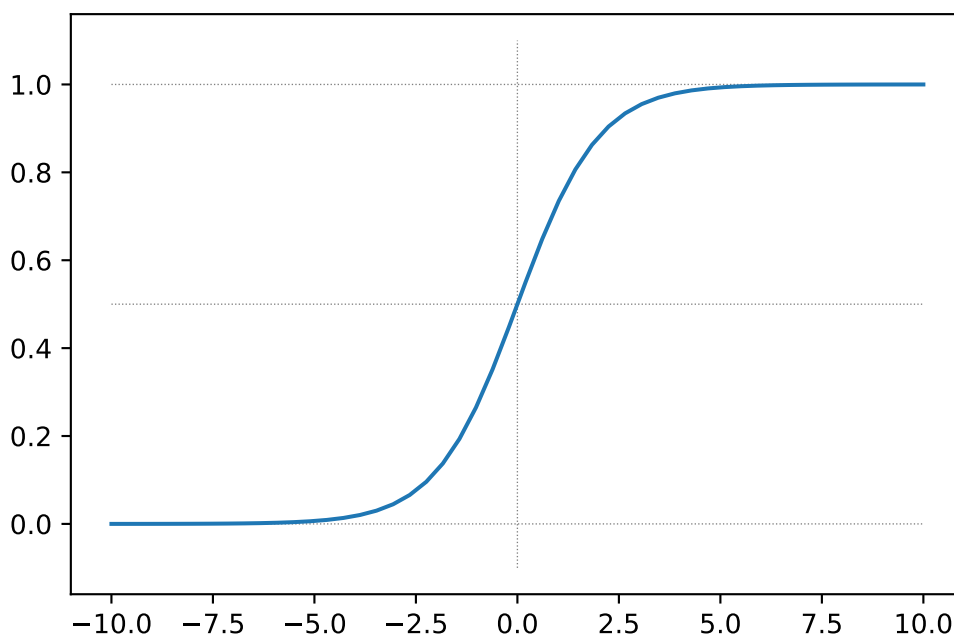
Download: `abalone_naive_bayes.ipynb`.

11.6 Logistische Regression

Die Idee der *Logistischen Regression* ist zunächst die Erstellung eines linearen Modells, indem jede Komponente eines Datenpunktes mit einem Faktor multipliziert und dazu eine Konstante addiert wird:

$$\bar{y} = \sum_{i=1}^N w_i \cdot x_i + b$$

Der Wert $\bar{y}$ ist dabei eine Schätzung für einen tatsächlichen Funktionswert $y = f(\mathbf{x})$. Wir wollen aber keine stetige Größe vorhersagen, sondern eine von (zunächst) zwei Klassen, den wir die Label 0 bzw. 1 zuordnen. Dazu wenden wir auf $\bar{y}$ die logistische Funktion $\sigma(x) = \frac{1}{1+e^{-x}}$ an:



Wir berechnen also

$$\hat{y} = \sigma(\bar{y})$$

und ordnen einen Punkt zur Klasse 1 zu, wenn der berechnete Wert größer als 0,5, und zur Klasse 0, wenn er kleiner als 0,5 ist.

In *scikit-learn* verwenden wir dafür die Klasse `LogisticRegression`:

```
from sklearn.linear_model import LogisticRegression
```

```

classifier = LogisticRegression(
    max_iter=zahlIterationen[,
    C=regularisierungsStaerke
)

```

Wenn wir eine Warnung erhalten, dass das Verfahren nicht konvergiert, können wir die Zahl der Iterationen (Vorgabe: 100) mit dem Parameter `max_iter` vergrößern. Mit weiteren Parametern können wir das Lösungs- und Regularisierungsverfahren sowie die Behandlung von Mehrklassenproblemen (s. u.) einstellen.

Der Parameter `C` dient zur *Regularisierung*. Neben der Differenz zwischen unserer Vorhersage $\hat{y}$ und der tatsächlichen Klasse wollen wir auch die Größe der einzelnen Koeffizienten w_i begrenzen, indem wir die mit C skalierte Summe von deren Quadraten ebenfalls minimieren:

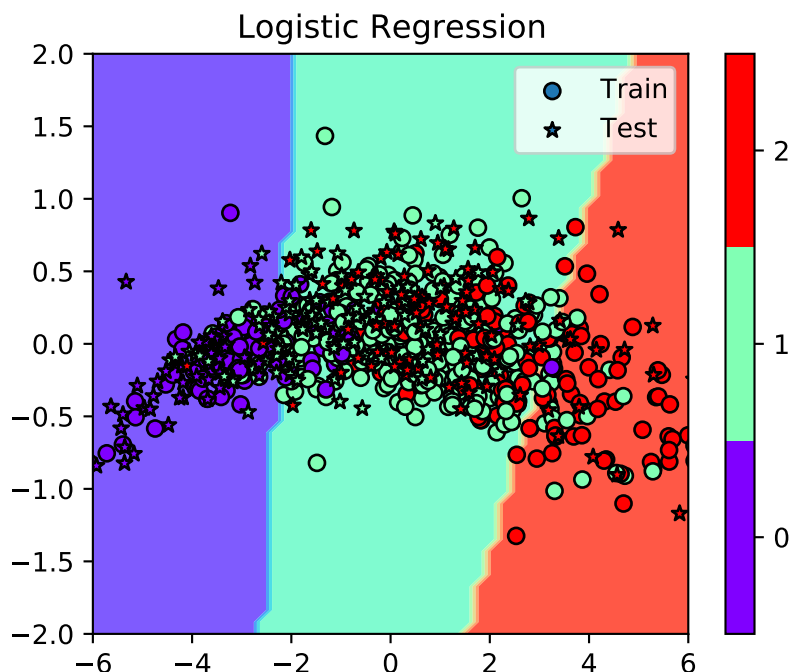
$$\text{Strafe} = \frac{1}{C} \cdot \sum_{i=1}^N (w_i)^2$$

Große Werte von C beschränken die Koeffizienten wenig (Gefahr Overfitting), kleine Werte führen zu einer starken Regularisierung (starker Beschränkung der Parameter) und können damit ein Underfitting verursachen.

Beim Einsatz anderer Lösungsverfahren, die über den Parameter `solver` ausgewählt werden können, sind auch andere Verfahren zur Begrenzung der Koeffizienten möglich (Parameter `penalty`). Hier sei auf die *scikit-learn*-Dokumentation verwiesen.

Auch dieses Klassifikationsverfahren unterstützt die Methode `predict_proba()`, mit der wir die Sicherheit einer Entscheidung für eine Klasse besser einordnen können. Diese Methode liefert wieder eine Matrix zurück, deren Zeilen die jeweilige Vorhersage und deren Spalten die Wahrscheinlichkeiten für jede Klasse enthalten.

Beispiel: Ohne weitere Parameter (außer Erhöhung der Iterationszahl).

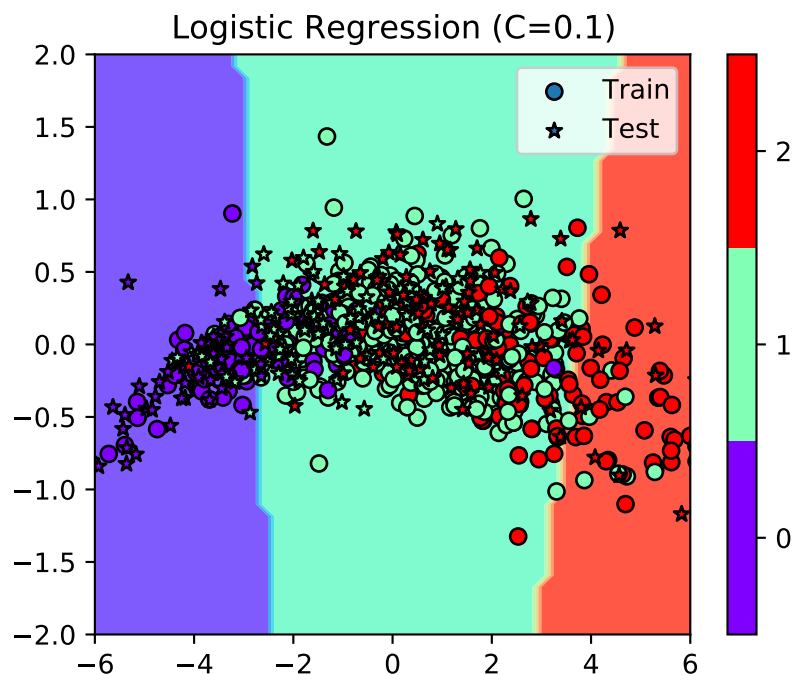


Genauigkeit:

- Training (alle Features): 68%
- Test (alle Features): 67%
- Training (2 Features): 76%

- Test (2 Features): 51%

Mit dem Regulierungsparameter $C=0.01$ (starke Regularisierung) erhalten wir:



Genauigkeit:

- Training (alle Features): 67%
- Test (alle Features): 62%
- Training (2 Features): 75%
- Test (2 Features): 53%

Wählen wir mit $C=1000$ eine starke Spezialisierung, ergibt sich folgendes Bild:

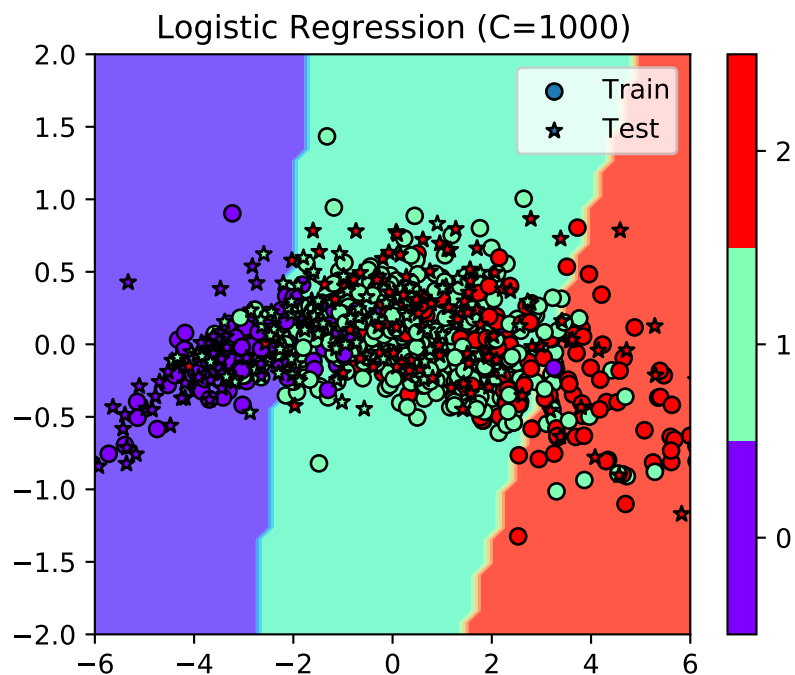
Genauigkeit:

- Training (alle Features): 68%
- Test (alle Features): 67%
- Training (2 Features): 77%
- Test (2 Features): 51%

Sind unsere Daten unbalanciert, i. e. die Klassengrößen für die Zielgröße sehr verschieden, können wir mit dem Parameter `class_weights` das Modell entsprechend beeinflussen. Wir können entweder für jede Klasse ein Gewicht angeben oder mit dem String `balanced` die Klassen reziprok zu ihrer Häufigkeit wichten. Für unseren Beispieldatensatz erhalten wir ohne Angabe von C folgende Genauigkeiten:

- Training (alle Features): 66%
- Test (alle Features): 68%
- Training (2 Features): 70%
- Test (2 Features): 45%

Wir erhalten hier keine Verbesserung. Allerdings können wir auch statt des Wertes `balanced` auch ein Dictionary übergeben, dass für jede Klasse eine Wichtung enthält.



Beispiel: Wie wir aus der Confusion Matrix sehen, erhalten wir für die Klassen 1 und 2 viele Fehler, deshalb versuchen wir, gezielt für diese beiden Klassen die Wichtung zu erhöhen.

```
class_weight={0:1, 1:2, 2:4}
```

Wir erhalten folgende Genauigkeiten:

- Training (alle Features): 66%
- Test (alle Features): 70%
- Training (2 Features): 73%
- Test (2 Features): 56%

Die logistische Regression führt zu recht kleinen Modellen, da die Zahl der Modellparameter der Zahl der Komponenten entspricht und von der Zahl der Trainingsdaten unabhängig ist. Das Verfahren funktioniert gut, wenn sich die Daten *linear* trennen lassen, i. e. wenn wir Hyperebenen finden, dass die Punkte verschiedener Klassen auf verschiedenen Seiten dieser Ebenen liegen, was bei hochdimensionalen Daten oft der Fall ist.

Notebook

Logistische Regression mit verschiedenen Regularisierungswerten und Klassengewichten: `abalone_logistic_regression.ipynb`.

11.7 Weitere Verfahren

Neben diesen einfachen Verfahren sind Entscheidungsbäume und *SVMs* sehr beliebte Klassifikationsverfahren, die wir später betrachten wollen. Ebenso werden wir die Klasse der *ANN*, die insbesondere bei einer großen verfügbaren Zahl von Trainingsdaten deutlich bessere Ergebnisse liefern können, getrennt untersuchen.

11.8 Mehrfachklassifikation

Wir haben in der Analyse der Verfahren gesehen, dass diese häufig nur zwei Klassen unterscheiden können, aber in unserem Beispiel mehr als zwei Klassen trennen.

Um die Daten in mehr als zwei Klassen aufteilen zu können, wenden wir wiederholt die Trennung in zwei Klassen an. Dafür gibt es zwei prinzipielle Vorgehensweisen.

Bei der *One-versus-One*-Klassifikation (*OvO*) wählen wir alle Paare von zwei verschiedenen Klassen und die dazu gehörigen Datenpunkte aus und erzeugen ein eigenes Modell für jedes Paar. Wir erhalten also für k Klassen $\binom{k}{2} = \frac{k(k-1)}{2}$ Modelle.

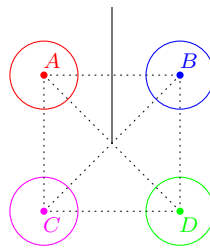


Abb. 11.3: One-versus-One

Um einen neuen Datenpunkt zu klassifizieren, bestimmen wir eine Vorhersage mit jedem unserer Modelle und ordnen ihn der Klasse zu, die am häufigsten vorhergesagt wurde.

Bei der *One-versus-Rest*-Klassifikation (*OvR*) erzeugen wir für k Klassen nur k Modelle und trainieren jedes mit allen Trainingsdaten. Jedes dieser Modelle trifft dabei die Entscheidung, ob der Datenpunkt zu dieser Klasse gehört oder nicht.

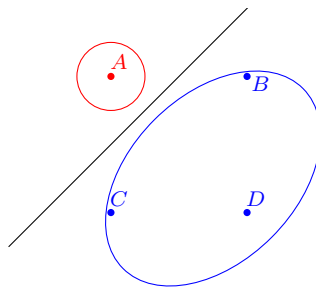


Abb. 11.4: One-versus-Rest

Um die Klasse für einen neuen Datenpunkt vorherzusagen, berechnen wir ebenfalls die Vorhersage jedes Modells und wählen nun diejenige Klasse, bei der der Abstand des Punktes von der trennenden Hyperebene am größten ist (allgemein: die Klasse mit der höchsten Wahrscheinlichkeit im Sinne des Modells).

In *scikit-learn* wird für die lineare *SVM* die One-versus-Rest-Methode und für die Kernel-*SVM* die One-versus-One-Methode benutzt. Die erste Methode verwendet für jedes Modell alle Datenpunkte und erreicht deshalb häufig eine höhere Modellqualität, bei der zweiten Methode sinkt dagegen der Trainingsaufwand je Modell, da nur die Datenpunkte der beiden beteiligten Klassen eingehen.

11.9 Persistenz

Wollen wir ein erstelltes Modell für zukünftige Vorhersagen verwenden und es nicht jedesmal neu aus den Trainingsdaten erstellen, sollten wir es dauerhaft in einer Datei aufbewahren. Im Gegensatz zu *TensorFlow* stellt *Scikit-learn* dafür keine eigenen Methoden zur Verfügung, sondern setzt auf Standard-Module. Die Standardbibliothek bietet dafür das Modul *pickle*, für komplexere Modelle mit einer großen Zahl von Parametern (wie z. B. *KNN*-Modelle mit vielen Trainingsdaten) ist das Modul *joblib* besser geeignet.

```
import joblib
joblib.dump(modell, dateiName)
```

Dieses Modell kann später wieder geladen werden:

```
modell = joblib.load(dateiName)
```

Oft müssen mehrere Objekte, z. B. zusätzlich die Datentransformation, gespeichert werden. Wir übergeben der `dump()`-Methode dann einfach ein Tupel von Objekten.

```
joblib.dump((transformation, modell, ...), dateiName)
```

Beim Lesen benutzen wir die Mehrfachzuweisung.

```
transformation, modell, ... = joblib.load(dateiName)
```

Beispiel: Trainieren einer Transformation und einer Klassifikation für den *Abalone*-Datensatz und Speichern dieses Modells. Wir trainieren die Transformation mit einem Data Frame *ohne* Zielgröße, damit unser Modell auch für bisher unbekannte Daten verwendet werden kann, bei denen wir die Zielgröße nicht kennen.

Quellcode 11.3: *abalone-savemodel.py*: Training und Speichern des Modells.

```
#!/usr/bin/env python
# coding: utf-8

# # Modell speichern

import joblib
import abalone_helper
import numpy as np
from sklearn import compose, preprocessing, linear_model

# Abalone Trainingsdaten lesen und Abtrennen der Zielgröße.

dfTrain = abalone_helper.toLabels(
    abalone_helper.readFile('abalone-train.csv'), False
)
dfTrainX = dfTrain.drop('labels', axis=1)
yTrain = dfTrain['labels'].values

# Transformation anlegen, trainieren und anwenden.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```

numCols, catCols = abalone_helper.colTypes()
tr = compose.ColumnTransformer([
    ('num', preprocessing.StandardScaler(), numCols),
    ('cat', preprocessing.OneHotEncoder(), catCols)
])
Xtrain = tr.fit(dfTrainX).transform(dfTrainX)

# Klassifikationsobjekt erzeugen und trainieren.

classifier = linear_model.LogisticRegression(max_iter=200)
classifier.fit(Xtrain, yTrain)

# Modell speichern: Transformation und Klassifikationsobjekt.

joblib.dump((tr, classifier), 'abalone-model.joblib')

```

Beispiel: Einlesen dieses Modells und Klassifizieren der Testdaten mit diesem Modell.

Quellcode 11.4: abalone-loadmodel.py: Laden des Modells und Klassifizieren der Testdaten.

```

#!/usr/bin/env python
# coding: utf-8

# # Modell laden

# Wir müssen hier die *sklearn*-Module für Preprocessing und Klassifikation nicht
↪ importieren, da sämtliche Informationen in den importierten Objekten verfügbar
↪ sind.

import joblib
import abalone_helper
from sklearn import metrics

# Testdaten lesen. Label abtrennen.

dfTest = abalone_helper.toLabels(
    abalone_helper.readFile('abalone-test.csv'), False
)
dfTestX = dfTest.drop('labels', axis=1)
yTest = dfTest['labels'].values

# Modell einlesen. Wir erhalten ein Tupel (Transformation, Klassifikation).

tr, classifier = joblib.load('abalone-model.joblib')

# Testdaten transformieren mit unserer trainierten Transformation.

Xtest = tr.transform(dfTestX)

# Testdaten vorhersagen und bewertung mit unserem trainierten
↪ Klassifikationsobjekt.

print('Score:', classifier.score(Xtest, yTest))

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
predTest = classifier.predict(Xtest)
print('Confusion:')
print(metrics.confusion_matrix(yTest, predTest))
```

Mit diesem Modell können wir auch neue Daten klassifizieren, nur die Bewertung der Modellqualität ist aufgrund der unbekannten Label nicht mehr möglich.

Notebooks

Trainieren und Speichern des Modells: `abalone-savemodel.ipynb`, Laden des Modells und Klassifikation der Testdaten: `abalone-loadmodel.ipynb`, Helper-Modul (erweitert um eine Funktion für die Spaltenlisten): `abalone_helper.py`.

11.10 Aufgaben

Aufgabe 11.1 Einfache Klassifikationsverfahren

Erzeugen Sie auf den von Ihnen ausgewählten Datensatz verschiedene Klassifikationsmodelle und experimentieren Sie mit den Hyperparametern, um eine möglichst gute Vorhersage zu erhalten.

Berechnen Sie dazu die Güte (Anteil korrekter Vorhersagen) für diese Modelle sowohl für Trainings- als auch die Testdaten.

- Grundlagen
- Beispieldatensatz
- Lineare Regression in scikit-learn
- Weitere Regressionsmodelle
- Nichtlinearer Zusammenhang
- Aufgaben

12.1 Grundlagen

Im Gegensatz zur Klassifikation wollen wir hier keine Klasse (aus einer endlichen Auswahl), sondern eine stetige Größe, z. B. einen Preis oder eine Lebensdauer vorhersagen.

Der klassische Ansatz für dieses Problem ist die *Lineare Regression*, bei der wir jede Komponente unserer Datenpunkte mit einem Faktor multiplizieren diese aufaddieren und dazu einen *Bias* hinzufügen:

$$\hat{y} = \sum_{i=1}^N w_i x_i + b$$

Die Koeffizienten werden dabei so gewählt, dass die Differenz zwischen tatsächlicher und vorhergesagter Zielgröße für unsere Trainingsdaten möglichst klein wird. Dazu werden für alle Trainingspunkte die Differenzen zwischen Vorhersage und tatsächlichem Wert quadriert und aufsummiert und dieser Gesamtfehler minimiert.

$$\min_{\mathbf{w}, b} \sum_{k=1}^M (\hat{y}_k - y_k)^2$$

Dieses einfache Vorgehen findet allerdings nur lineare Zusammenhänge zwischen den Komponenten und der Zielgröße. Es sind deshalb auch andere Vorgehensweisen denkbar.

Wie auch bei der Klassifikation benutzen einige Modelle euklidische Abstände zwischen den Datenpunkten zur Bestimmung ihrer Ähnlichkeit. Es ist also auch hier oft erforderlich, eine Vorverarbeitung der Daten wie bei der Klassifikation vorzunehmen.

12.2 Beispieldatensatz

Wir verwenden hier weiter unseren Datensatz *Abalone* und betrachten nun direkt die Ringzahl als vorherzusagende Zielgröße, die wir nun nicht mehr künstlich in Klassen einteilen.

Beim Preprocessing der Komponenten müssen wir genauso wie bei Klassifikationsproblemen vorgehen:

- Behandlung von Fehlwerten: Entfernen von Datensätzen, Komponenten oder Ersetzen durch Default-Werte.
- Skalieren der metrischen Größen mit `MinMaxScaler` oder `StandardScaler`.
- Umcodieren der kategorialen Größen mit `OneHotEncoder`.

Beispiel: Preprocessing des *Abalone*-Datensatzes (Wiederholung).

```
import pandas as pd
import numpy as np
from sklearn import compose, preprocessing
import abalone_helper

dfTrain = pd.read_csv('abalone-train.csv')
dfTest = pd.read_csv('abalone-test.csv')

# Abtrennen Zielgröße
dfTrainX = dfTrain.drop('rings', axis=1)
dfTestX = dfTest.drop('rings', axis=1)
yTrain = dfTrain.rings.values
yTest = dfTest.rings.values

# Umskalieren bzw. -codieren der Komponenten
numCols, catCols = abalone_helper.colTypes()
tr = compose.ColumnTransformer([
    ('num', preprocessing.StandardScaler(), numCols),
    ('cat', preprocessing.OneHotEncoder(), catCols),
])
```

Aufgabe 12.1 Unterschiede Preprocessing.

Ersetzen Sie den `StandardScaler` durch einen `MinMaxScaler` und vergleichen Sie die Vorhersagequalität der folgenden Modelle für beide Vorverarbeitungsmethoden.

Wir ermitteln nun die beiden Komponenten mit der größten Korrelation zur Zielgröße, um parallel eine graphische Darstellung der Vorhersagefunktion zu ermöglichen:

```
corr = dfTrain.corr()
corr.rings.sort_values(ascending=False)
```

Wir erhalten:

```
rings      1.000000
shell      0.632480
diameter   0.575516
length     0.560935
whole      0.549559
height     0.539213
viscera    0.513855
shucked    0.441658
isex       0.365583
Name: rings, dtype: float64
```

Da wir in unserer Transformation zuerst die numerischen Komponenten transformieren, erhalten wir die Spaltennummern für `shell` und `diameter` einfach aus der Funktion `colTypes()` unseres Hilfsmoduls. Wir erzeugen parallel einen Datensatz mit diesen beiden Komponenten.

```
numCols, catCols = abalone_helper.colTypes()
idxShell = numCols.index('shell')
idxDia = numCols.index('diameter')
Xtrain2 = Xtrain[:, [idxShell, idxDia]]
Xtest2 = Xtest[:, [idxShell, idxDia]]
```

Für diese beiden Komponenten erstellen wir wieder einen Estimator und wenden diesen auf jeden Rasterpunkt an, um Vorhersagefunktion darzustellen. Zusätzlich tragen wir unsere Trainings- und Testpunkte in das Diagramm ein:

```
def plotPredict(est2, Xtrain2, yTrain, Xtest2, yTest):
    x = np.linspace(-3, 5)
    y = np.linspace(-4, 3)
    xx, yy = np.meshgrid(x, y)
    xy = np.hstack((xx.reshape(-1,1), yy.reshape(-1,1)))
    zz = est2.predict(xy).reshape(xx.shape)
    plt.contourf(xx, yy, zz, cmap='hot', alpha=0.5, levels=50)
    plt.scatter(Xtrain2[:,0], Xtrain2[:,1], c=yTrain, cmap='hot',
                label='Training')
    plt.scatter(Xtest2[:,0], Xtest2[:,1], c=yTest, cmap='hot',
                marker='*', label='Test')
    plt.legend()
    plt.colorbar()
```

Um die Güte der Vorhersage zu veranschaulichen, stellen wir die Wertpaare (*tatsächlicher Wert*, *Vorhersage*) als Scatter Plot dar. Je besser unser Modell ist, desto näher müssen die Punkte an der Linie $y = x$ liegen.

```
def plotAccuracy(yTrain, predTrain, yTest, predTest):
    plt.scatter(yTrain, predTrain, marker='^', label='Train')
    plt.scatter(yTest, predTest, marker='*', label='Test')
    plt.plot([0, 30], [0,30], label='GT=Predict')
    plt.xlabel('GT')
    plt.ylabel('Predict')
    plt.legend()
```

12.3 Lineare Regression in scikit-learn

Das oben theoretisch vorgestellte Verfahren der *linearen Regression* wird im Modul `linear_model` in der Klasse `LinearRegression` bereitgestellt.

```
from sklearn import linear_model

estimator = linear_model.LinearRegression(...)

estimator.fit(Xtrain, ytrain)
```

Eine Vorhersage ist wieder mit der Methode `predict()` möglich.

```
ypred = estimator.predict(Xneu)
```

Um eine Vorstellung zu erhalten, sagen wir die Preise für unsere Testdaten vorher und vergleichen sie mit den tatsächlichen Werten.

```
predTest = estimator.predict(Xtest)
# absoluter Fehler
absErr = np.abs(predTest - yTest)
relErr = absErr/yTest
# DataFrame für bequeme Darstellung
dfErr = pd.DataFrame({'GT': ytest, 'Pred': predTest, 'absErr': absErr,
                      'relErr': relErr})
dfErr.head(10)
```

Ergebnis:

	GT	Pred	absErr	relErr
0	15	9.065880	5.934120	0.395608
1	7	7.824840	0.824840	0.117834
2	9	10.607413	1.607413	0.178601
3	10	9.397839	0.602161	0.060216
4	7	6.754806	0.245194	0.035028
5	8	7.803425	0.196575	0.024572
6	20	12.926303	7.073697	0.353685
7	16	11.052268	4.947732	0.309233
8	9	9.506747	0.506747	0.056305
9	19	12.533376	6.466624	0.340349

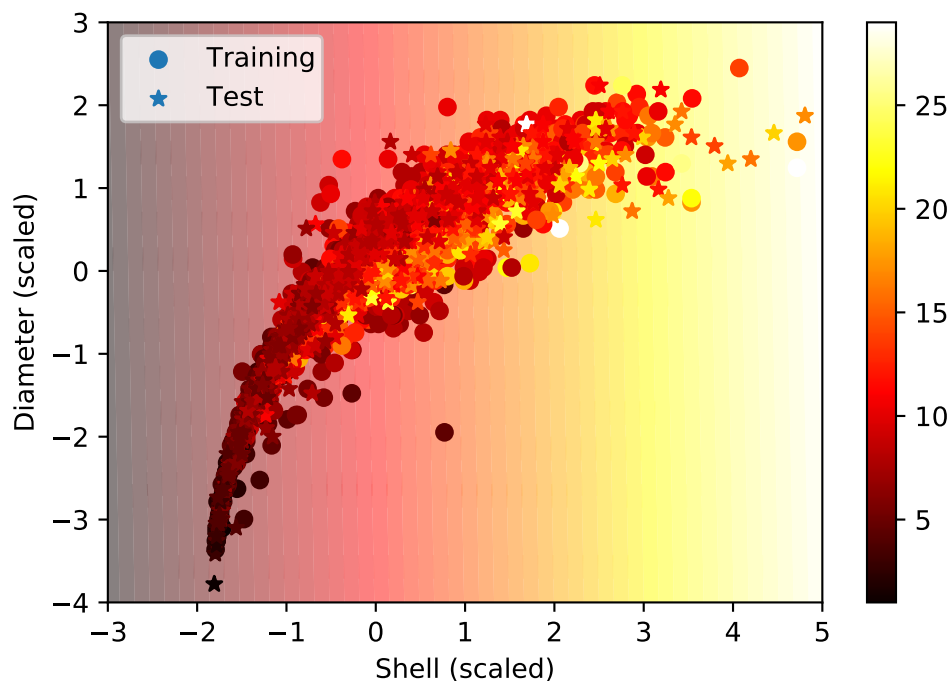


Abb. 12.1: Vorhersagefunktion für 2 Komponenten

Um auch hier eine sinnvolle Bewertung der Vorhersagequalität durchführen zu können, wird auch hier eine Methode `score()` bereitgestellt, die die quadratischen Fehler (also die quadrierte Differenz zwischen Vorhersage und wahrem Wert) aufsummiert und so normiert, dass wir für vollständig korrekte Vorhersagen den Wert 1 und sonst einen kleineren (möglicherweise auch negativen) Wert erhalten.

```
score = estimator.score(X, y)
```

Für die hier gewählten Trainings- und Testdaten erhalten wir die Werte 0.52 (Training) und 0.54 (Test).

Wir sehen, dass insbesondere bei großen Werten der Vorhersagewert zu niedrig ist, aber auch anhand der Abweichung der Trainingsdaten von der Geraden, dass Modelle dieser Art keine besseren Vorhersagen liefern können. Grund ist entweder, dass die gegebenen Komponenten zu wenig Einfluss auf die Zielgröße haben oder der Zusammenhang nicht linear ist.

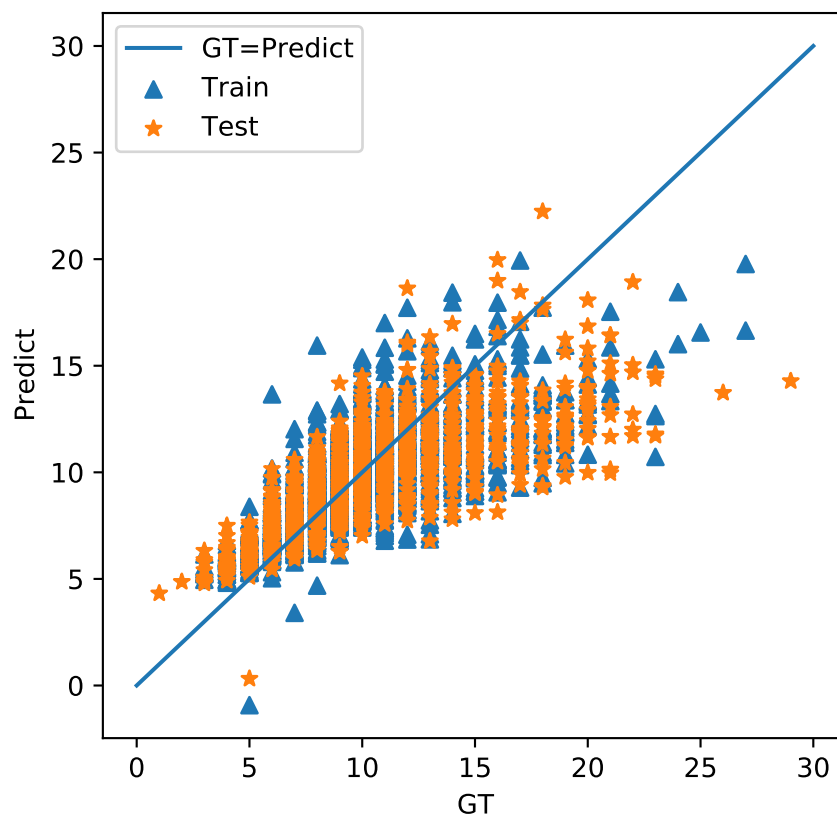


Abb. 12.2: Vergleich Tatsächliche Daten–Modell

12.4 Weitere Regressionsmodelle

Wie schon bei der logistischen Regression können auch bei der linearen Regression die Modellkoeffizienten in ihrer Größe beschränkt werden. Während wir bei der normalen linearen Regression in unserer Kostenfunktion nur den Fehler zwischen Vorhersage und tatsächlichem Wert minimieren, fügen wir nun zu dieser Funktion noch einen Strafterm hinzu, der die Gewichte begrenzt.

Üblich sind 3 Verfahren:

Ridge (l2-Regularisierung). Wir verwenden als Strafterm die Summe der Quadrate der Gewichte:

$$\alpha \sum_{i=1}^N w_i^2$$

Der Parameter α ist ein Regularisierungsparameter, der beschreibt, wie stark große Gewichte bestraft werden. Je größer α , um so stärker werden diese beschränkt.

Die Verwendung der Quadrate führt zu einer stärkeren Bestrafung großer Gewichte.

Lasso (l1-Regularisierung) Statt der Quadrate der Faktoren werden nun deren Beträge verwendet.

$$\alpha \sum_{i=1}^N |w_i|$$

Der Effekt ist hier, dass einige Faktoren zu Null werden, so dass unwichtigere Komponenten ausgeblendet werden.

Elastic-Net Dieses Verfahren verwendet eine Kombination aus Lasso- und Ridge-Regularisierung und besitzt für jeden Anteil einen Hyperparameter.

Im Gegensatz zur logistischen Regression wird die kann die Regularisierung nicht als Hyperparameter bei der Modellerstellung angegeben werden, sondern jede der Regularisierungen erfordert eine eigene Modellklasse.

```
ridgeModel = linear_model.Ridge([alpha=wert] ... )
lassoModel = linear_model.Lasso([alpha=wert] ... )
elasticModel = linear_model.ElasticNet([alpha=wert, l1_ratio=wert] ... )
```

Der Vorgabewert für `alpha` ist 1.0, der für `l1_ratio` ist 0.5.

Für unseren Datensatz erhalten wir mit der *Ridge*-Regularisierung keine Verbesserung, *Lasso* und *Elastic-Net* verschlechtern das Ergebnis.

Die Methode *KNN* ist ebenfalls für Regressionsprobleme anwendbar. Wir bestimmen dazu für den gesuchten Datenpunkt die k nächsten Nachbarn aus der Trainingsdatenmenge und errechnen aus deren Zielwerten den Mittelwert für die Vorhersage.

```
from sklearn import neighbors

estimator = neighbors.KNeighborsRegressor(...)
```

Mit dem Vorgabewert von 5 Nachbarn erhalten wir eine Modellgenauigkeit Für unseren Beispieldatensatz erhalten wir eine Modellgenauigkeit von 0.65 auf den Trainings- und 0.44 auf den Testdaten, das Modell neigt verfahrensbedingt also stärker zum Overfitting.

Wählen wir nur einen Nachbarn, sehen wir das Overfitting noch deutlicher: alle Trainingsdaten werden korrekt vorhergesagt und liegen auf der Diagonale, der Fehler auf den Testdaten wird größer, die Genauigkeit sinkt auf 0.2 (am Bild nicht so deutlich zu erkennen).

Auch die zerklüftete Vorhersagefunktion deutet auf das Overfitting hin.

Auch der Entscheidungsbaum kann für Regressionsprobleme genutzt werden. Beim Erstellen des Baums wird dabei die Zielgröße automatisch in Intervalle zerlegt, jedes Blatt des Baumes entspricht also einem Intervall. Für die Vorhersage wird das Blatt ermittelt, in das der Datenpunkt fällt, und aus der Zielgröße der Trainingsdaten in diesem Blatt ein Mittelwert für die Vorhersage gebildet. Wir erzeugen das Modellobjekt `tree.DecisionTreeRegressor`.

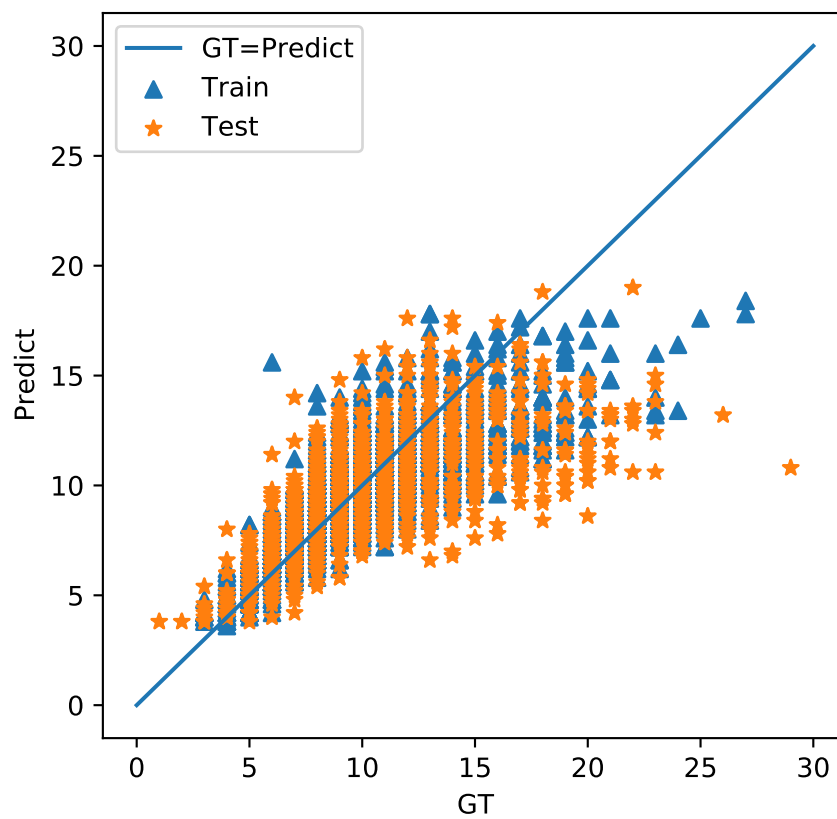


Abb. 12.3: Genauigkeit KNN mit 5 Nachbarn.

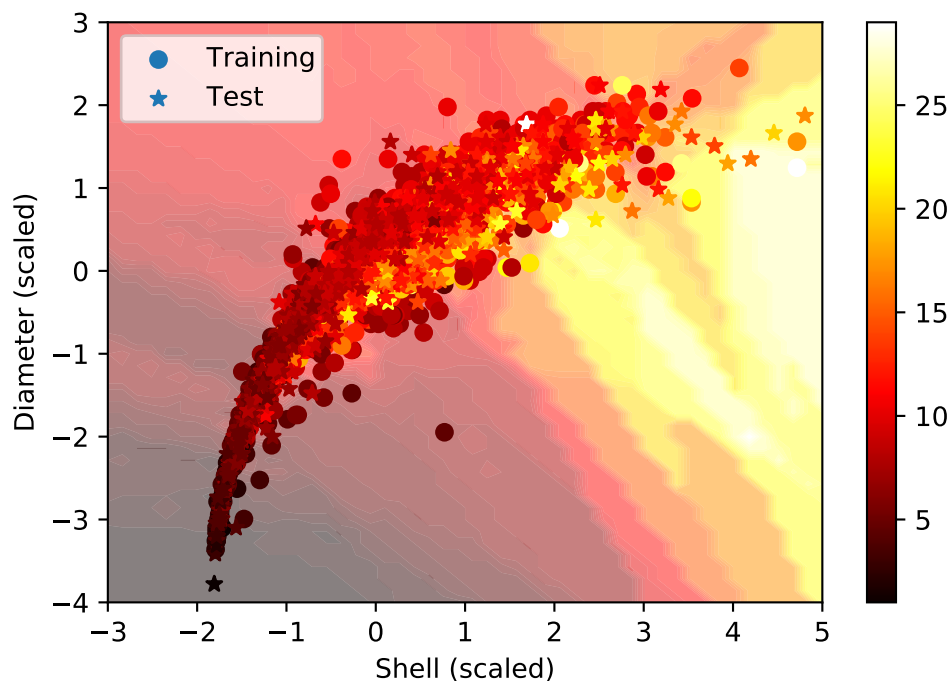


Abb. 12.4: Vorhersagefunktion KNN mit 5 Nachbarn.

```
from sklearn import tree
```

```
estimator = tree.DecisionTreeRegressor(...)
```

Der Entscheidungsbaum kennt eine Vielzahl von Parametern zur Regularisierung, die wir später besprechen. Wenden wir das Modell mit den Vorgabewerten auf unseren Datensatz an, erhalten wir eine Genauigkeit von 1.0 auf den Trainings- und 0.23 auf den Testdaten, also ein starkes Overfitting.

Die Vorhersagefunktion zeigt wieder eine starke Zerklüftung, wobei alle Grenzen waagrecht und senkrecht verlaufen (aufgrund der Funktionsweise des Verfahrens).

Um den Einfluss der Hyperparameter zu zeigen, begrenzen wir die Baumtiefe auf 4 (`max_depth=4`) und erhalten eine Genauigkeit von nur noch 0.50 auf den Trainings- aber den verbesserten Wert von 0.45 auf den Testdaten.

Weitere leistungsfähige Regressionsmodelle werden wir in den folgenden Kapiteln betrachten.

Notebook

Alle obigen Regressionsmodelle: `abalone-linregression.ipynb`.

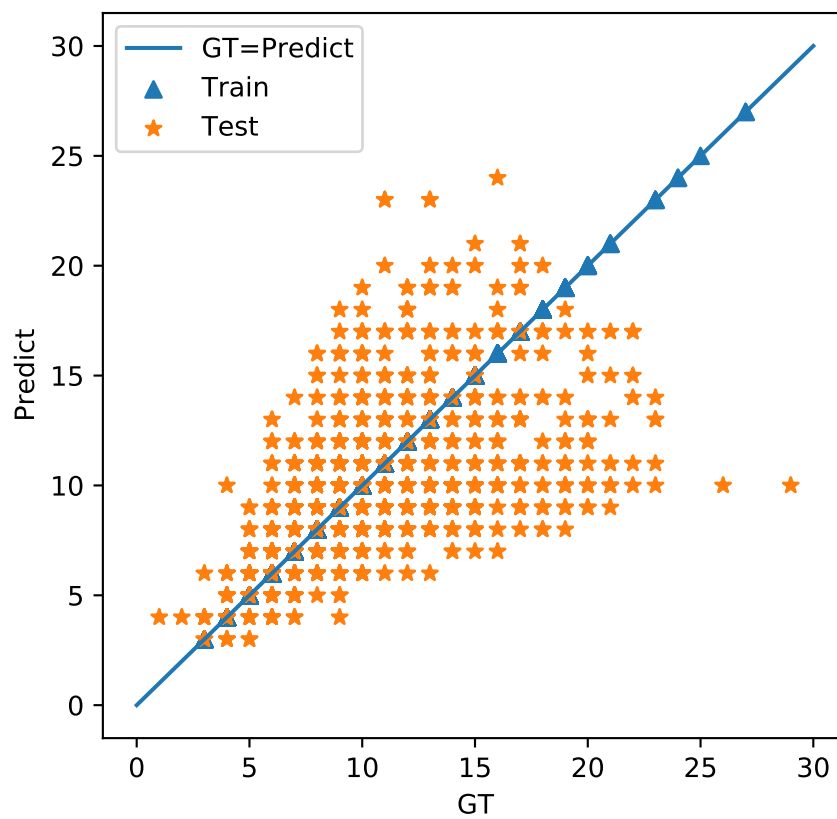


Abb. 12.5: Genauigkeit KNN mit 1 Nachbarn.

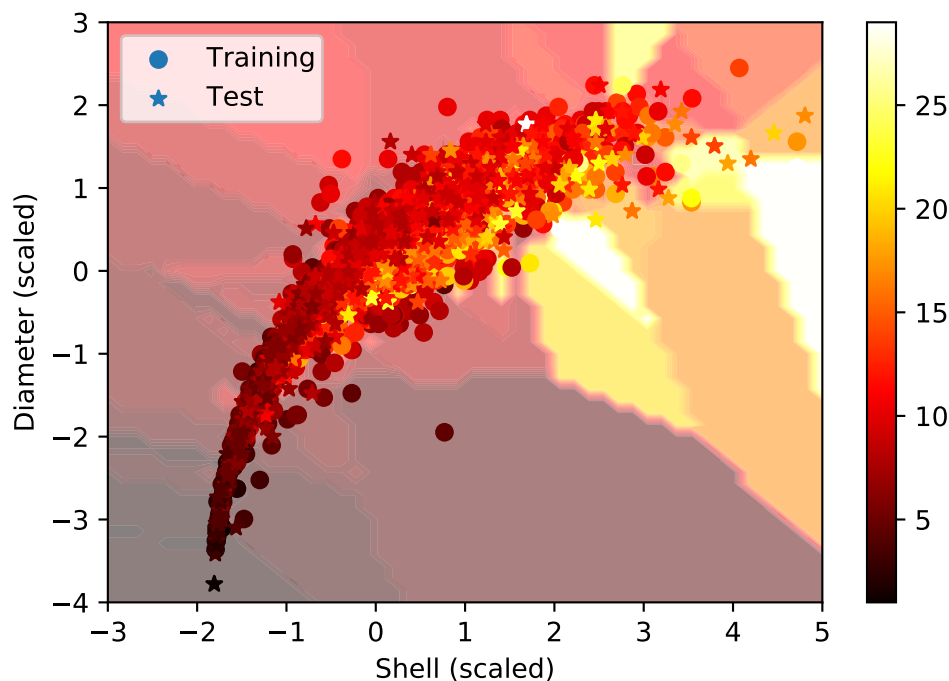


Abb. 12.6: Vorhersagefunktion KNN mit 1 Nachbarn.

12.5 Nichtlinearer Zusammenhang

Unser lineares Regressionsmodell kann nur lineare Zusammenhänge modellieren. Wie wir schon oben am Beispiel gesehen haben, können wir durch geeignete Transformation unserer Zielgröße (hier durch Logarithmieren) auch nichtlineare Zusammenhänge darstellen. Oft hängt aber auch die Zielgröße nicht direkt von einzelnen Komponenten, sondern von Kombinationen zwischen diesen ab.

Beispiel: Der Funktionswert unserer zweidimensionalen Daten ist vom Abstand der Daten vom Ursprung abhängig. Ein lineares Modell kann lediglich eine geneigte Ebene darstellen, aber niemals einen Paraboloid.

Hinweis: Die oben ebenfalls betrachteten Verfahren *KNN* und Entscheidungsbaum sind keine linearen Modelle und können deshalb mit derartigen Zusammenhängen umgehen.

Um solche Zusammenhänge in einem linearen Modell darstellen zu können, erweitern wir unsere Komponenten um alle Kombinationen von paarweisen Produkten aus ihnen (oder auch höheren Potenzen).

```
poly = preprocessing.PolynomialFeatures (degree=grad)
poly.fit (Xtrain)
X2train = poly.transform (Xtrain)
X2test = poly.transform (Xtest)
```

Sinnvoll ist meist ein Grad von 2, andernfalls wächst die Zahl der Komponenten und damit die Modellkomplexität zu stark an, ohne dass wir eine wesentliche Vorhersageverbesserung erreichen.

Da die Komponenten vorher skaliert werden sollten, ehe wir die polynomiellen Kombinationen hinzufügen, bietet es sich an, beide Transformationen in einer *Pipeline* zu verknüpfen, so dass sie immer gemeinsam als Einheit behandelt werden können.

```
from sklearn import pipeline
```

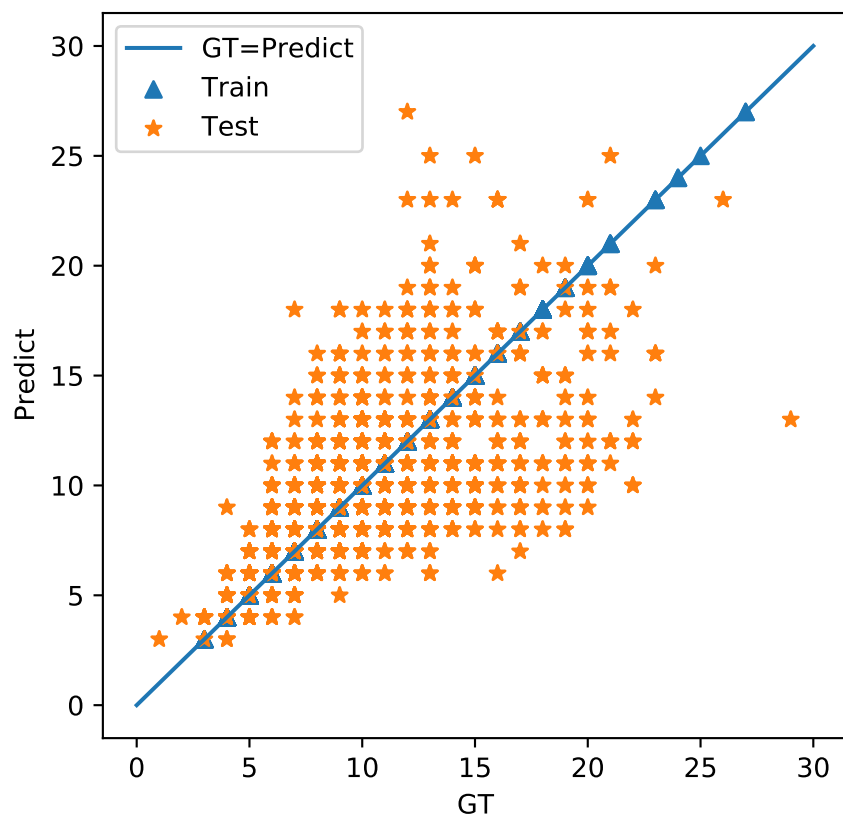


Abb. 12.7: Genauigkeit DT.

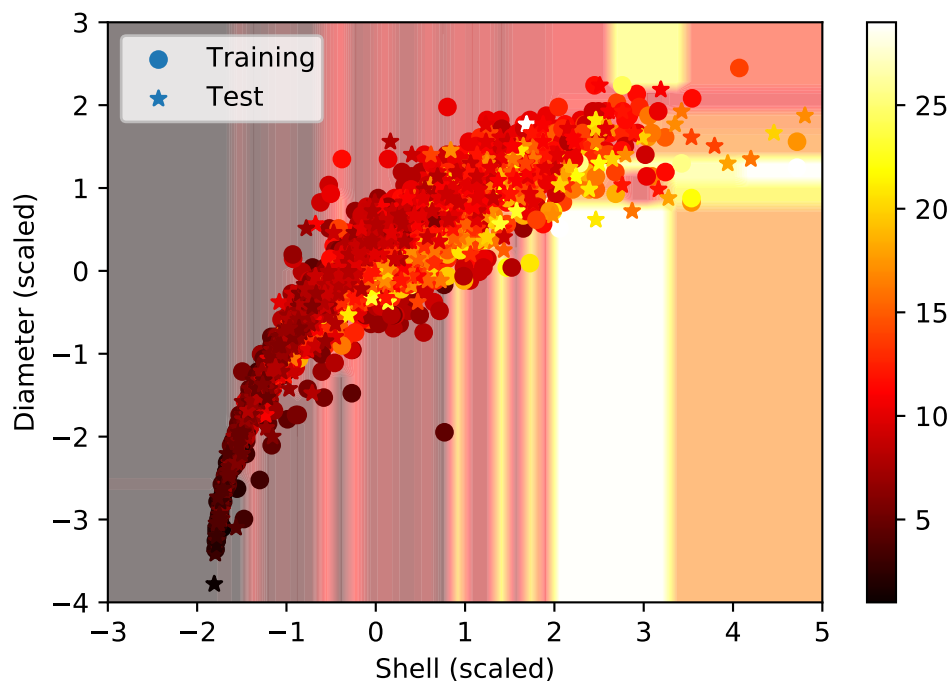


Abb. 12.8: Vorhersagefunktion DT.

```
tr = pipeline.Pipeline([
    ('name', estimatorObjekt),
    ...
])
```

Dieses Pipeline-Objekt unterstützt wiederum die Methoden `fit()` und `transform()` und kann auch wieder in einem `ColumnTransformer` für eine Spaltengruppe verwendet werden.

Beispiel: Skalierung und quadratische Features für die numerischen Komponenten des *Abalone*-Datensatzes, One-Hot-Encoding der kategorialen Spalte.

```
# dfTrainX, dfTestX: Data Frame der Features mit entfernter Zielgröße
numCols, catCols = abalone_helper.colTypes()
numFilter = pipeline.Pipeline([
    ('scale', preprocessing.StandardScaler()),
    ('poly', preprocessing.PolynomialFeatures(degree=2)),
])
tr = compose.ColumnTransformer([
    ('num', numFilter, numCols),
    ('cat', preprocessing.OneHotEncoder(), catCols),
])
Xtrain = tr.fit(dfTrainX).transform(dfTrainX)
Xtest = tr.transform(dfTestX)
dfTrainX.shape, Xtrain.shape
```

Aus unseren 8 Originalkomponenten entstehen nach der Transformation 39 Komponenten:

- 36 numerische Komponenten (7 Quadrate x_i^2 , $\binom{7}{2} = 21$ Produkte $x_i \cdot x_k$, 7 lineare Terme x_i und ein Bias-Term),
- 3 kategoriale One-Hot-Komponenten.

Erzeugen wir für diesen transformierten Datensatz ein lineares Regressionsmodell, erhalten wir die Genauigkeit von 0.58 auf den Trainings- und 0.56 auf den Testdaten.

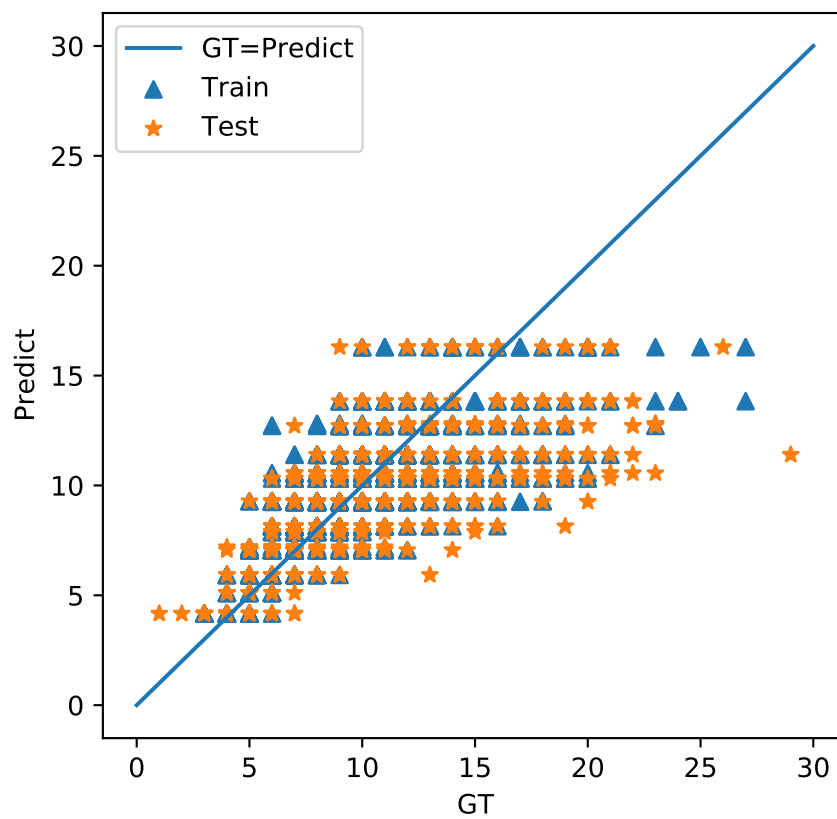


Abb. 12.9: Genauigkeit DT mit maximaler Tiefe 4.

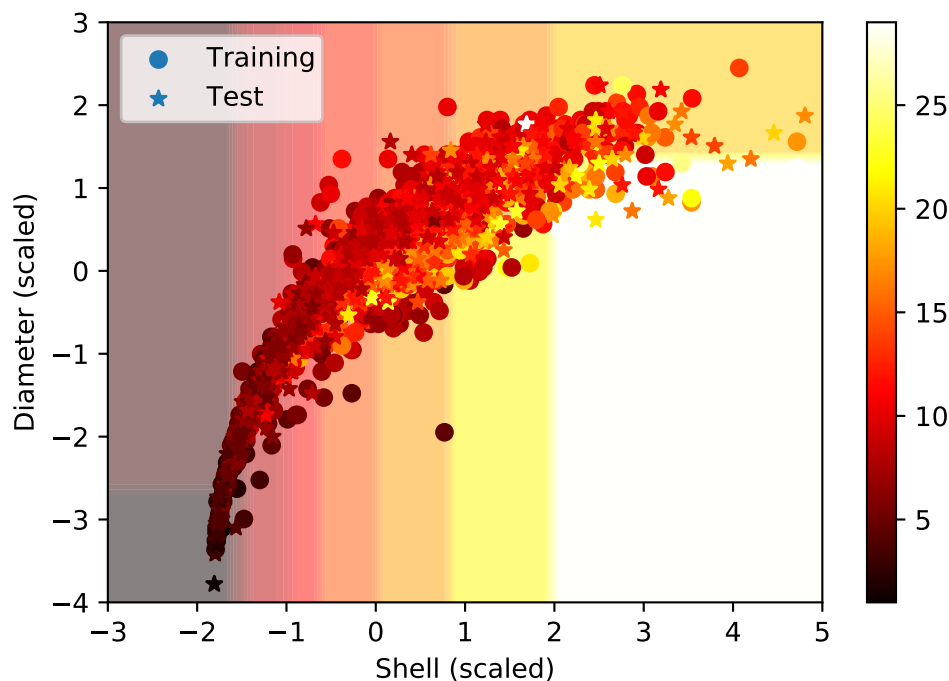


Abb. 12.10: Vorhersagefunktion DT mit maximaler Tiefe 4.

Für die ersten 10 Testdatensätze erhalten wir folgende Vorhersagen bzw. Fehler:

GT	Pred	absErr	relErr	
0	15	8.864771	6.135229	0.409015
1	7	7.646388	0.646388	0.092341
2	9	10.768910	1.768910	0.196546
3	10	9.850275	0.149725	0.014973
4	7	6.692841	0.307159	0.043880
5	8	8.078341	0.078341	0.009793
6	20	14.836897	5.163103	0.258155
7	16	11.212075	4.787925	0.299245
8	9	9.590300	0.590300	0.065589
9	19	13.489999	5.510001	0.290000

Wir sehen eine kleine, aber nicht allzu deutliche Verbesserung.

Zur Anzeige unserer Vorhersagefunktion beschränken wir uns wiederum auf die beiden Originalkomponenten `shell` und `diameter`. Hier müssen wir jedoch beachten, dass daraus durch die Ergänzung der polynomialen Features die Anzahl der Komponenten vergrößert wird. Wir müssen also im Diagramm die Achsen auf die Originaldaten beziehen und jeden Rasterpunkt ebenfalls erst transformieren, ehe wir die Klassifikation anwenden können.

Wir erkennen hier deutlich, dass der Funktionsverlauf keine Ebene im Raum mehr darstellt, die Niveaulinien sind alle gleichmäßig gekrümmt. Es entsteht eine konvexe, quadratische Funktion.

Hinweis: Bei der polynomiellen Transformation steigt die Zahl der Komponenten stark an, was bei der Modellbildung leicht zum Overfitting führen kann. Wir sollten deshalb die Größe der Faktoren durch Verwenden einer Regularisierung begrenzen.

Umgekehrt erfordert die höhere Zahl der Komponenten eine größere Zahl von Trainingsdaten, um ein Underfitting zu vermeiden (*Fluch der hohen Dimensionen*).

In unserem Beispiel erhalten wir beim Einsatz der Ridge-Regularisierung (l_2) ($\alpha=0.4$) eine Genauigkeit von

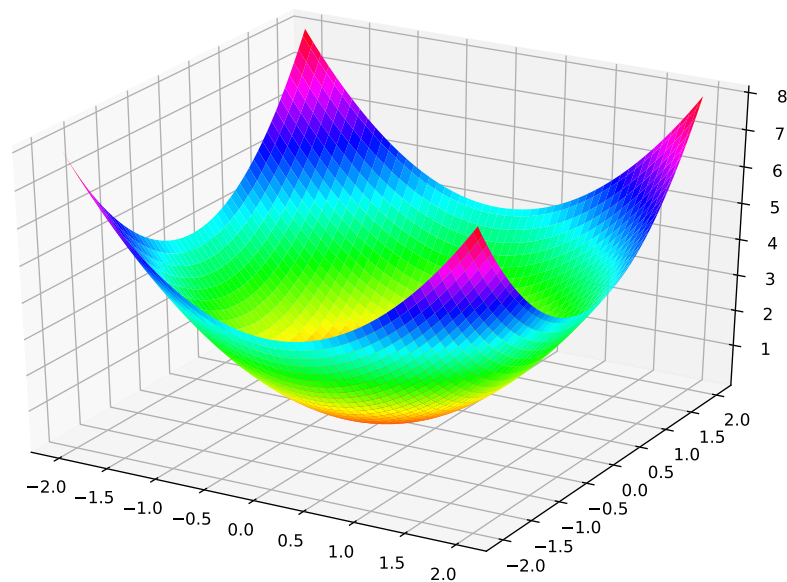
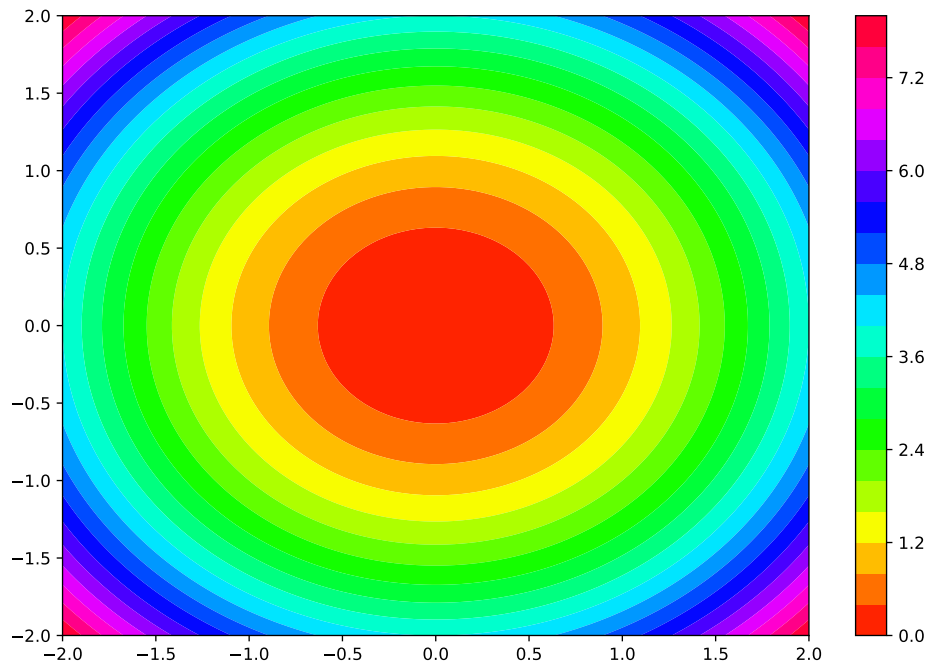


Abb. 12.11: Nichtlinearer Zusammenhang $y = x_1^2 + x_2^2$

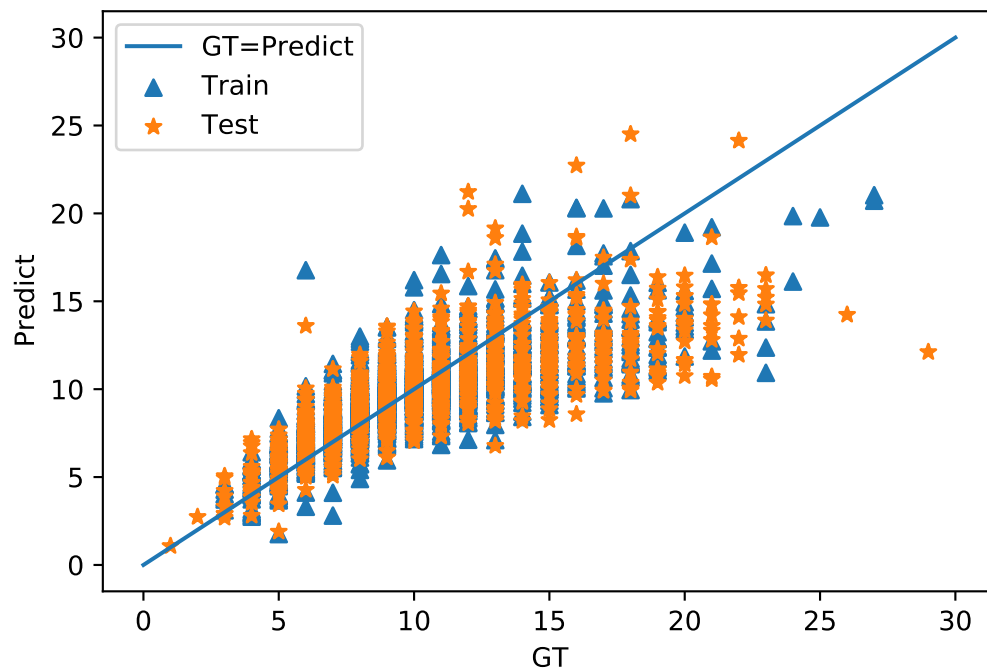


Abb. 12.12: Abweichung korrekte Daten–Modellvorhersage für Lineare Regression mit polynomiellen Features.

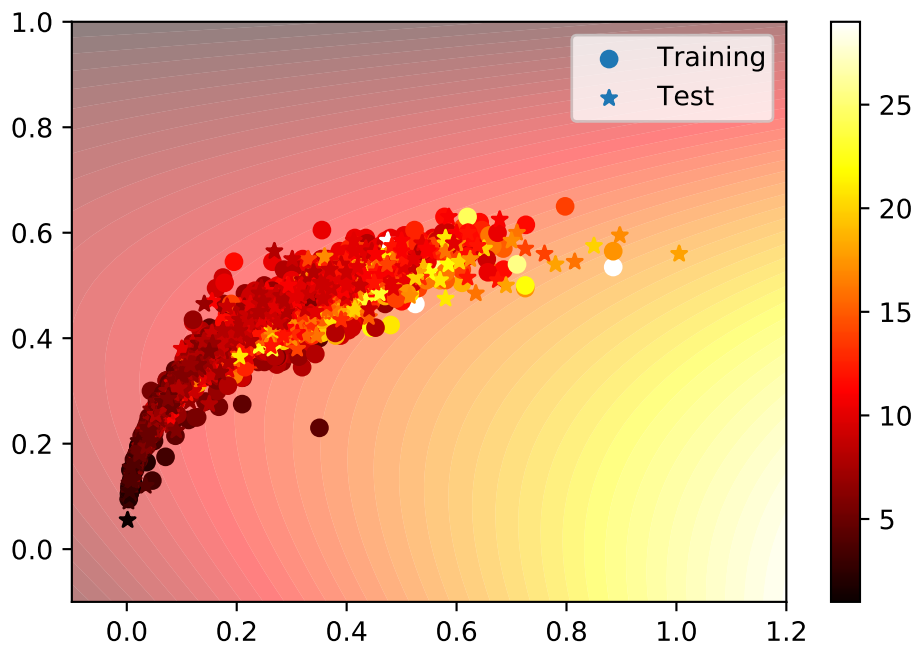


Abb. 12.13: Vorhersagefunktion lineare Regression mit polynomieller Transformation.

0.58 auf den Trainings- und 0.56 auf den Testdaten und damit fast die gleichen Werte wie ohne Regularisierung. Lasso und Elastic-Net führen zu einer deutlichen Verschlechterung und sind hier ungeeignet.

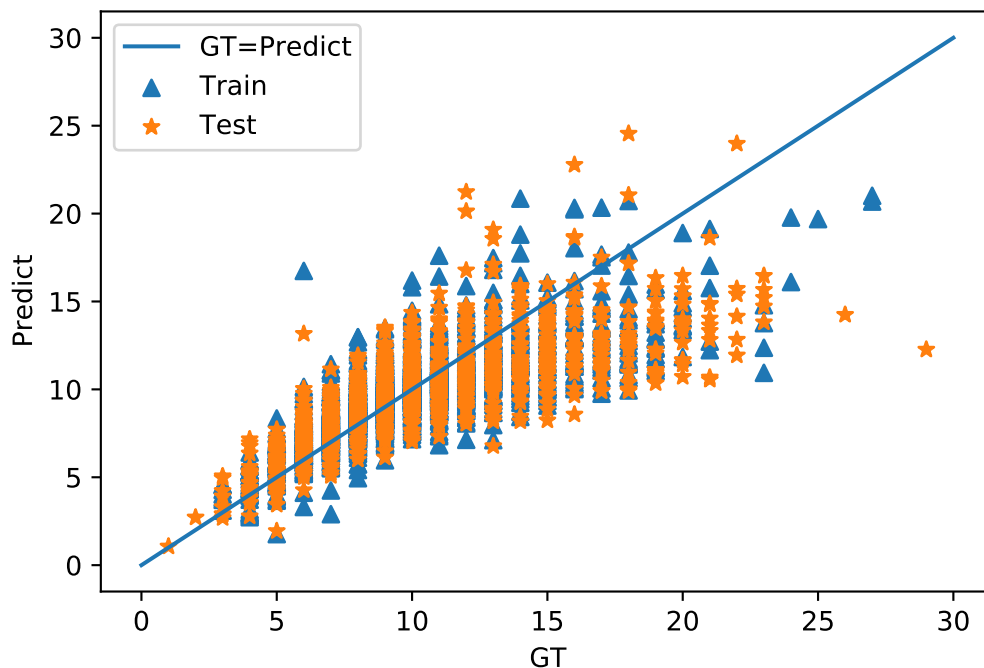


Abb. 12.14: Korrekte Werte und Verhersage bei polynomieller Transformation und Ridge-Regulasierung.

Notebook

Polynomialer Zusammenhang mit linearer Regression ohne und mit ℓ_2 -Regulierung: `abalone-polyregression.ipynb`.

12.6 Aufgaben

Aufgabe 12.2 Eigene Regression

Wählen Sie aus einem der bekannten Machine-Learning-Repositories oder einer der Probleme des [Data Mining Cup](#) der Vorjahre einen Regressions-Datensatz aus und untersuchen Sie, mit welchem Modell Sie gute Vorhersagen erhalten.

Aufgabe 12.3 Polynomielle Features

Experimentieren Sie beim Abalone-Datensatz mit höheren Polynomgraden und verschiedenen Regularisierungen.

- Wieviele Features entstehen durch die Transformation?
- Verbessert sich die Vorhersage?
- Wie verhalten sich die Regularisierungsverfahren?

Wenden Sie auch andere Regressionsverfahren auf den transformierten Datensatz an und bewerten Sie die Ergebnisse.

Entscheidungsbäume

- *Überblick*
- *Entscheidungsbäume in scikit-learn*
- *Analyse des Entscheidungsbaums*
- *Aufgaben*

13.1 Überblick

Eines der beliebtesten klassischen Klassifikationsverfahren sind *Entscheidungsbäume*. Dazu konstruieren wir einen (meist binären) Baum. In jedem Knoten definieren wir eine Entscheidung derart, dass wir eine Komponente der Daten auswählen und anhand eines Schwellwertes festlegen, welche Daten zum linken und welche zum rechten Kind wandern. Hier treffen wir erneut solche (links und rechts durchaus unterschiedliche) Entscheidungen. Der Baum wird durch Blätter abgeschlossen, denen die Klassen zugeordnet sind. Dabei gibt es meist etliche Blätter, die zur gleichen Klasse gehören.

Ein Datensatz durchläuft den Baum von der Wurzel und wird durch die jeweilige Bedingung in den Baumknoten auf einem Pfad durch den Baum transportiert und landet dann hoffentlich in einem Blatt, das zur richtigen Klasse gehört.

Um den Baum aufzubauen, betrachten wir den Trainingsdatensatz und wählen jeweils die Entscheidung, deren Aufteilung der Daten den höchsten *Informationsgewinn* liefert, im Idealfall also in der Weise, dass links nur Daten der ersten und rechts nur die der zweiten Klasse landen. Da dies normalerweise nicht möglich ist, definieren wir uns eine *Entropie* (ein Maß der Unbestimmtheit), die die *Reinheit* einer Menge S von Datenpunkten beschreibt. Wir zählen dazu die Datenpunkte jeder Klasse k (aus der Menge aller Klassen K) und bestimmen ihren relativen Anteil

$$p_k = \frac{\text{Anzahl Datenpunkte der Klasse } k \text{ in } S}{\text{Anzahl aller Datenpunkte in } S}$$

Die Entropie $H(S)$ berechnet sich nach folgender Formel (die aus der Informationstheorie stammt):

$$H(S) = - \sum_{k \in K} p_k \log_2 p_k$$

Enthält ein Knoten nur Daten einer Klasse, ist die Entropy 0, treten dagegen Knoten von zwei Klassen gleichhäufig auf ($p_1 = p_2 = \frac{1}{2}$), ist die Entropy

$$\begin{aligned} H(S) &= -\left(\frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{2} \log_2 \frac{1}{2}\right) \\ &= -(0.5 \cdot (-1) + 0.5 \cdot (-1)) \\ &= 1 \end{aligned}$$

Je kleiner die Entropy einer Datenmenge ist, desto sicherer können wir diese Menge komplett einer Klasse zuordnen, ohne dabei viele Datenpunkte falsch zu klassifizieren.

Alternativ zur Entropy verwendet man auch den sogenannten *Gini-Index*:

$$g(S) = 1 - \sum_{k \in K} p_k^2$$

Dieser kann etwas einfacher berechnet werden und hat nahezu die gleichen mathematischen Eigenschaften wie die Entropy: der Wert ist 0, wenn eine Datenmenge nur Punkte einer Klasse enthält, und umso größer, je »durchmischter« die Menge ist. Für eine Menge mit gleichhäufigen Datenpunkten aus zwei Klassen erhalten wir

$$\begin{aligned} g(S) &= 1 - \left(\frac{1}{2}^2 + \frac{1}{2}^2\right) \\ &= 1 - \left(\frac{1}{4} + \frac{1}{4}\right) \\ &= \frac{1}{2} \end{aligned}$$

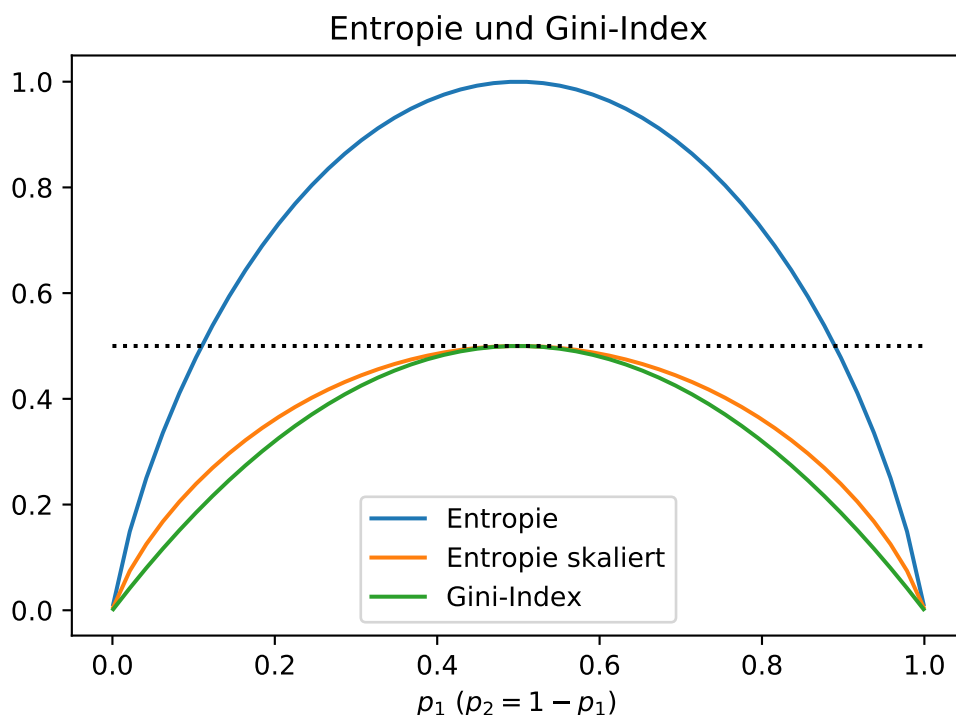


Abb. 13.1: Vergleich Entropie und Gini-Index für zwei Klassen.

Für den Entscheidungsbaum wird nun eine Komponente der Datensätze ausgewählt und die Menge der Datensätze anhand eines Schwellwertes für diese Komponente in zwei Teilmengen zerlegt. Die Auswahl der Komponente und

des Schwellwertes erfolgt dabei so, dass die relative Entropie der beiden Teilmengen dabei sinkt:

$$\frac{|S_1|}{S} \cdot H(S_1) + \frac{|S_2|}{S} \cdot H(S_2) < H(S)$$

$$S_1 \cup S_2 = S, \quad S_1 \cap S_2 = \emptyset$$

Diese Zerlegung von Knoten wird dabei solange fortgesetzt, bis die Entropie jedes Knotens 0 ist. Ein solcher Knoten enthält dann nur noch Elemente einer Klasse.

Da die vollständige Aufteilung der Daten in »reine« Knoten meist zum *Overfitting* neigt (unser Modell nur die Trainingsdaten »auswendig« lernt) und mit unbekannten Daten wenig anzufangen weiß, müssen wir die Aufteilung der Daten in die Knoten künstlich begrenzen:

- Begrenzen der Baumhöhe,
- Mindestzahl von Daten, die ein Knoten enthalten muss,
- erlaubt Komponenten, die für die Auswahl der besten Aufteilung herangezogen werden (zufällige Auswahl).

13.2 Entscheidungsbäume in scikit-learn

Die Erstellung von einfachen Entscheidungsbaum-Modellen mit *scikit-learn* haben wir bereits kennengelernt:

```
from sklearn import tree

classifier = tree.DecisionTreeClassifier(hyperparameter)
```

Wichtige Hyperparameter für den Entscheidungsbaum sind:

criterion Reinheitsmaß des Knotens. Vorgabe ist 'gini', alternativ kann 'entropy' verwendet werden.

max_depth Maximale Höhe des Entscheidungsbaums. Ohne Angabe wird die Datenmenge solange aufgespalten, bis alle Knoten nur noch »reine« Datenmengen enthalten.

min_samples_split Mindestzahl von Datenpunkten in einem Knoten, um diesen bei der Teilung zu berücksichtigen.

min_samples_leaf Mindestzahl von Datenpunkten in Blattknoten. Eine Knotenteilung wird also nur dann durchgeführt, wenn in beiden Kindknoten mindestens diese Zahl von Datenpunkten erhalten bleibt.

Wenn wir einen solchen Hyperparameter suchen, dürfen wir für die Bewertung nicht die Testmenge benutzen, da diese dann über diesen Parameter in die Modellbildung eingehen würde. Statt dessen spalten wir eine Validierungsmenge von unserer Trainingsmenge ab:

```
from sklearn import model_selection

XtrainHyp, Xvalid, yTrainHyp, yValid = model_selection.train_test_split(
    Xtrain, yTrain, test_size=0.2, random_state=23)
bestParam = None
bestScore = 0
for hyp in wertelisteHyperParameter:
    classifier = MyClassifier(hyp)
    classifier.fit(XtrainHyp, yTrainHyp)
    scoreValid = classifier.score(Xvalid, yValid)
    if scoreValid > bestScore:
        bestScore = scoreValid
        bestParam = hyp
# Modell mit bestem Hyperparameter erzeugen und mit allen Trainingsdaten
# trainieren
classifier = MyClassifier(bestParam)
classifier.fit(Xtrain, yTrain)
```

Beispiel: Wirkung der verschiedenen Hyperparameter im Datensatz *Abalone*. Zur Darstellung der Entscheidungsgrenzen beschränken wir uns wieder auf einen reduzierten Datensatz mit den beiden numerischen Komponenten, die am stärksten mit der (ursprünglichen stetigen) Zielgröße korreliert sind.

Baumtiefe

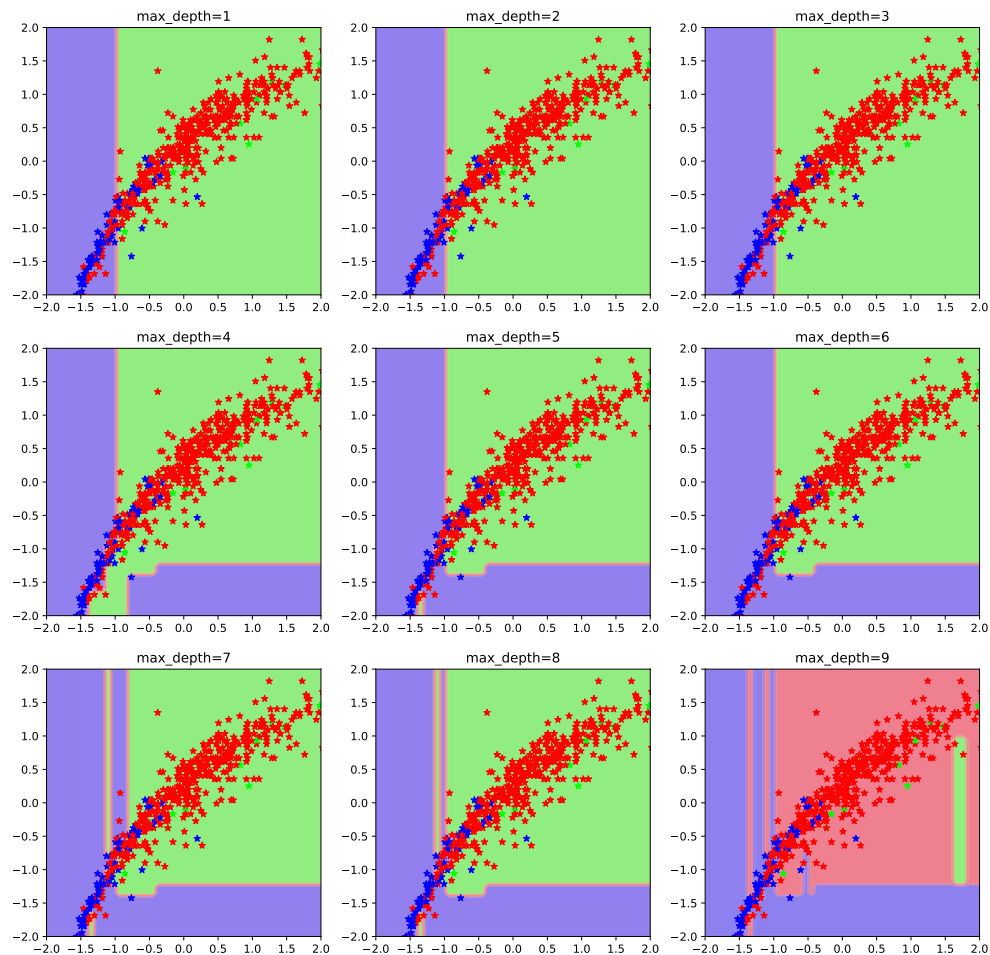


Abb. 13.2: Entscheidungsgrenzen bei unterschiedlicher Baumtiefe.

Wir erkennen bei Tiefe 2 ein Underfitting, ab Tiefe 8 sinkt die Genauigkeit auf den Testdaten stark ab (*Overfitting*).

Minimale Blattgröße

Wir erkennen hier bei zu kleiner Blattgröße ein *Overfitting* (jeder Punkt erhält praktisch seinen eigenen Blattknoten), eine Vergrößerung verbessert die Generalisierung, bis diese so stark wird, dass das Modell zu einfach wird und seine Vorhersagekraft verliert.

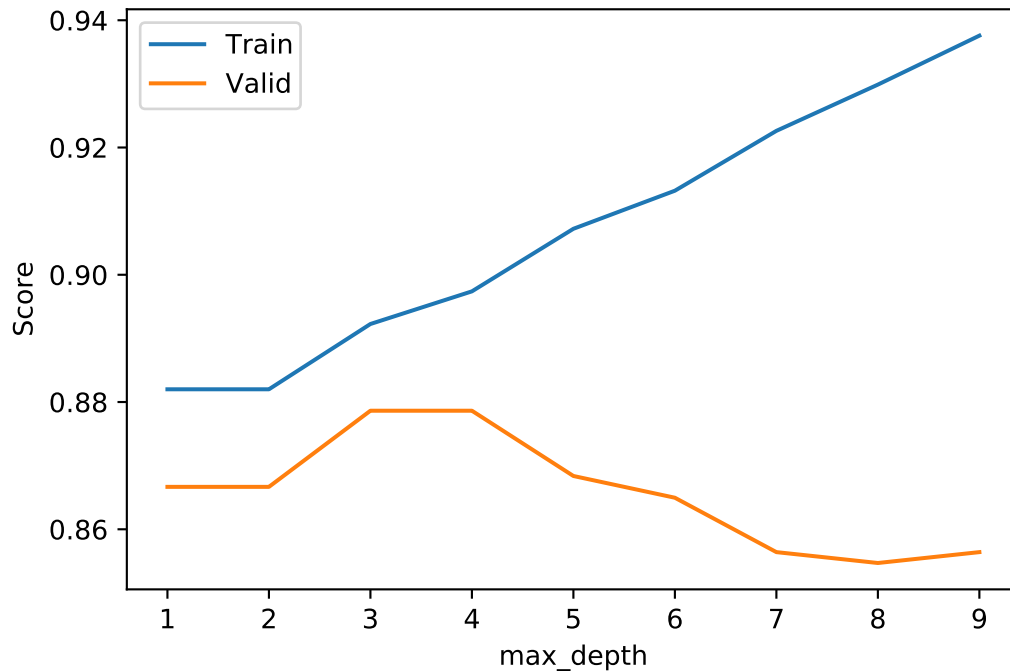


Abb. 13.3: Genauigkeit bei Trainings- und Testdaten bei unterschiedlicher Baumtiefe.

Minimale Splitgröße

Auch hier führt eine zu kleine Splitgröße zum *Overfitting*, eine größere Mindestzahl dagegen zu einer besseren Generalisierung, bis diese auch hier das Modell zu stark vereinfacht.

Notebook

Notebook zum Erzeugen der Diagramme: `abalone-tree-hyperparameter.ipynb`.

13.3 Analyse des Entscheidungsbaums

Ein Vorteil des Entscheidungsbaums gegenüber anderen Modellen ist die relativ einfache Nachvollziehbarkeit der Modellbildung, die durch entsprechende Funktionen des *scikit-learn*-Pakets unterstützt wird:

- Darstellung des Baums und der getroffenen Entscheidungen,
- Analyse der wichtigsten Features.

Eine graphische Visualisierung des Baums ist mit der Funktion `tree.plot_tree` möglich:

```
plt.figure(figsize=(breite, höhe))
tree.plot_tree(classifier[, ...])
...
plt.show()
```

Wichtige weitere Parameter für die Darstellung sind:

feature_names Liste der Komponentennamen, ohne Angabe wird nur der Komponentenindex angezeigt.

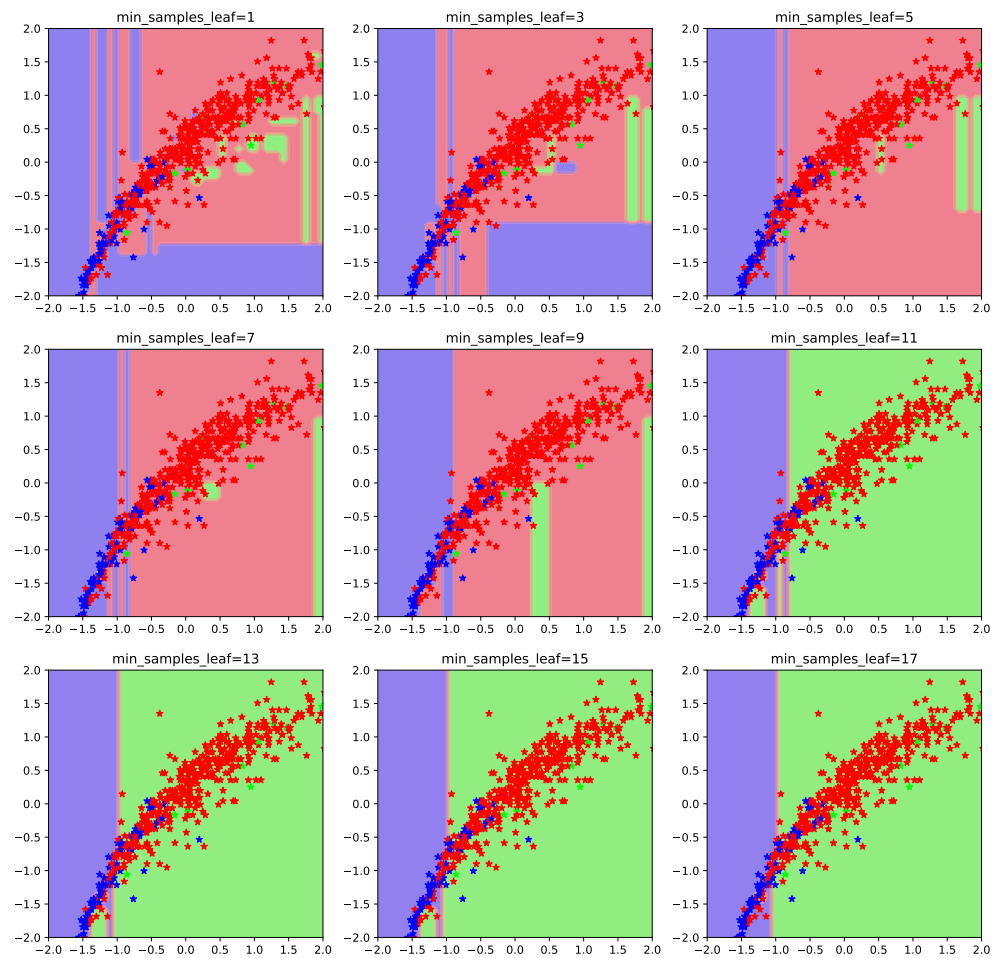


Abb. 13.4: Entscheidungsgrenzen bei verschiedener minimaler Blattgröße.

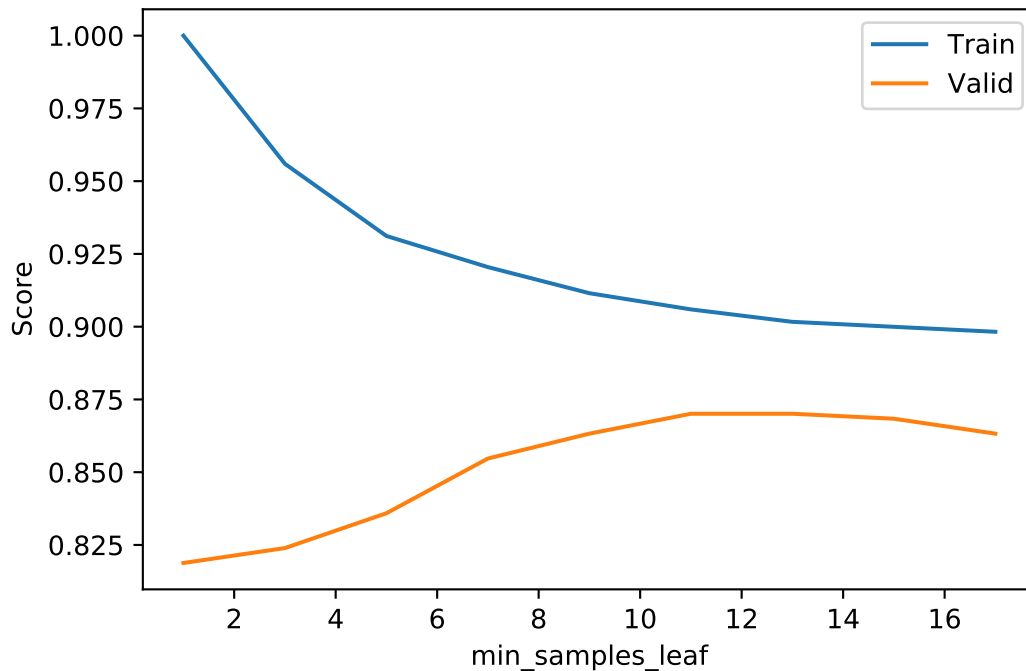


Abb. 13.5: Genauigkeit bei Trainings- und Testdaten bei unterschiedlicher minimaler Blattgröße.

max_depth Begrenzung der Darstellungstiefe, nur die wichtigsten Entscheidungsknoten (obersten Ebenen) werden angezeigt. Ohne diese Begrenzung ist meist keine lesbare Darstellung möglich. Der Wurzelknoten wird bei der Tiefenberechnung nicht gezählt, die tatsächliche Ebenenzahl ist also um eins größer.

class_names Liste der Klassennamen für die Anzeige. Ohne Angabe wird nur die Klassennummer angezeigt.

filled=True Einfärben der Knoten mit der in ihnen häufigsten Klasse.

Hinweis: Da der Baum sehr groß wird, ist eine Anpassung der Größe des Plots meist notwendig (`plt.figure(figsize=(Größe))`).

Beispiel: Entscheidungsbaum *Abalone*, Darstellung mit verschiedener Tiefe (der Baum selbst ist in der Tiefe nicht begrenzt).

Jeder Knoten zeigt an, nach welcher Komponente für welchen Wert die Daten getrennt werden. Die Entropie ist ein Maß der *Reinheit* des Knotens. `Samples` gibt die Anzahl der Trainingsdaten an, die diesen Knoten erreichen, `Value` eine Liste der Verteilung der Daten auf alle möglichen Klassen.

Schönere Darstellungen sind mit dem Paket *graphviz* möglich (siehe Dokumentation).

Unter Umständen ist eine Textrepräsentation übersichtlicher.

```
text = tree.export.export_text(classifier[, ...])
```

Hier sind u. a. folgende Parameter möglich:

feature_names Spaltennamen als Python-Stringliste.

max_depth Maximale Darstellungstiefe des Baums.

show_weights=True Anzeige der Anzahlen der Datenpunkte je Klasse in den Blattknoten.

spacing Einrücktiefe für Teilbäume (Vorgabe: 3)

Die Angabe der Klassennamen ist hier allerdings nicht möglich.

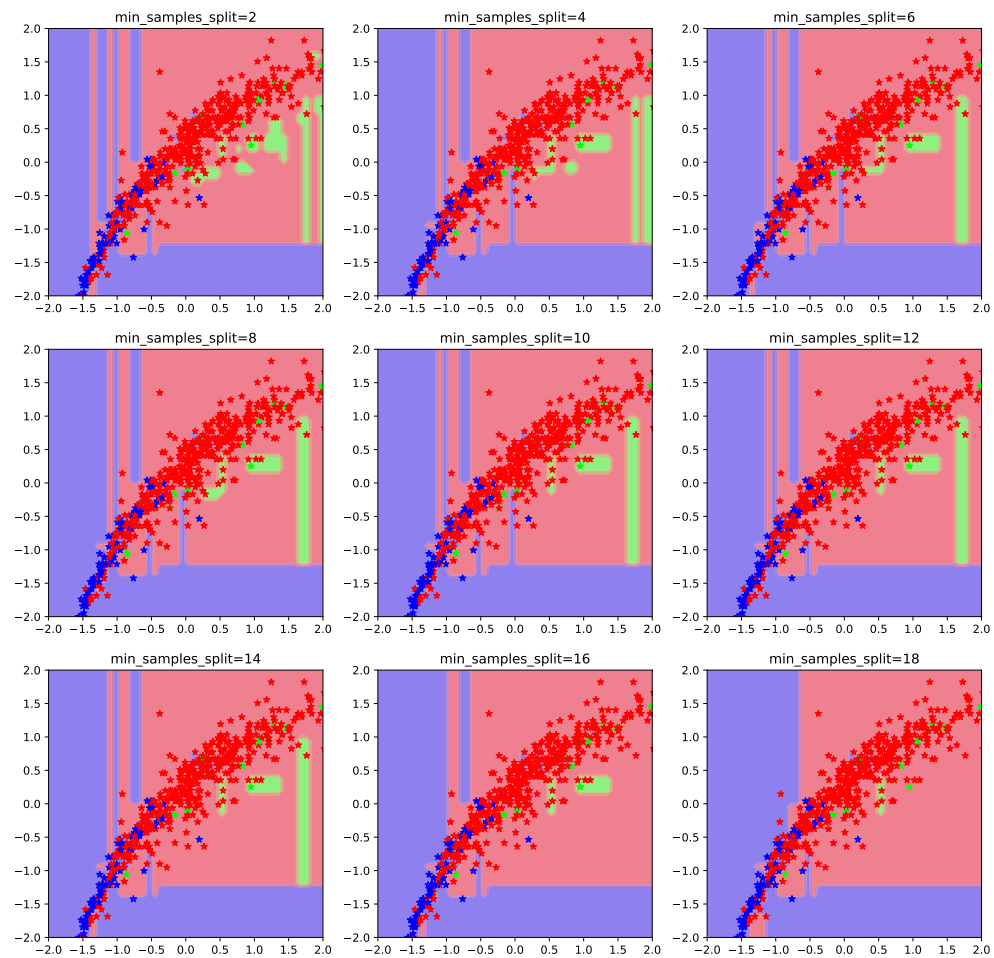


Abb. 13.6: Entscheidungsgrenzen bei verschiedener minimaler Splitgröße.

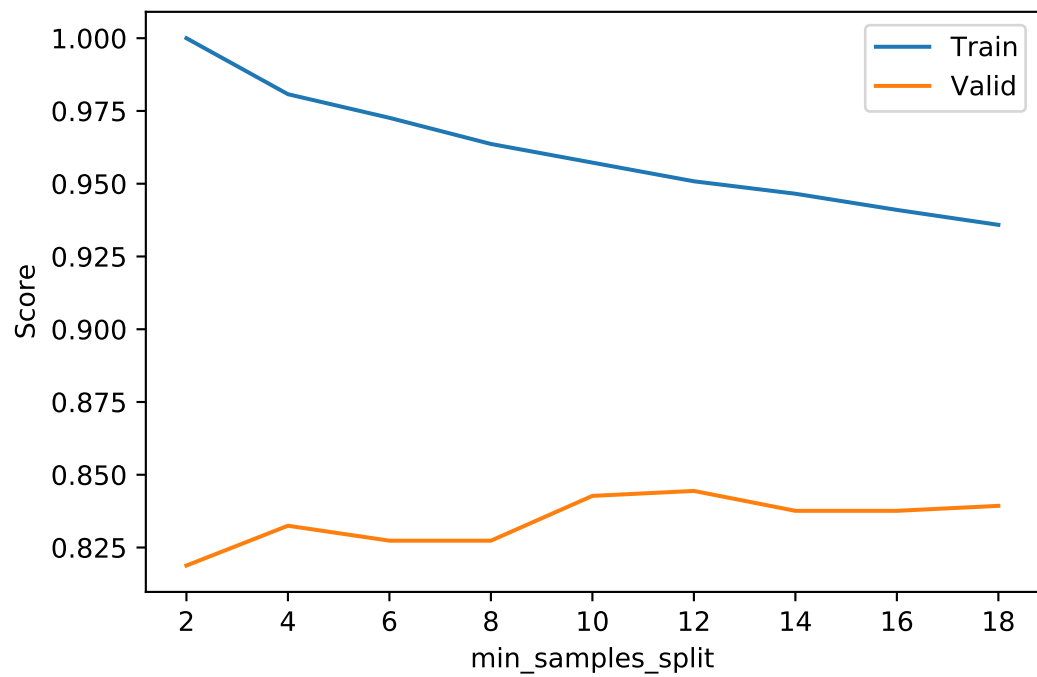


Abb. 13.7: Genauigkeit bei Trainings- und Testdaten bei unterschiedlicher minimaler Splitgröße.

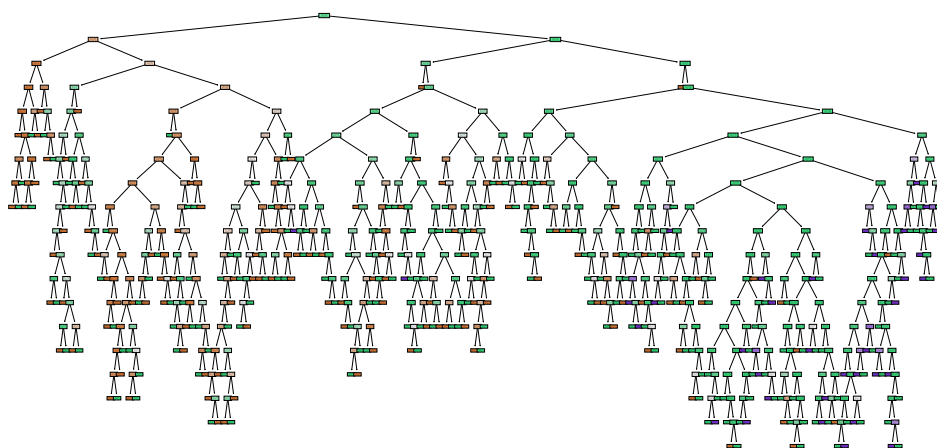


Abb. 13.8: Vollständiger Entscheidungsbaum.

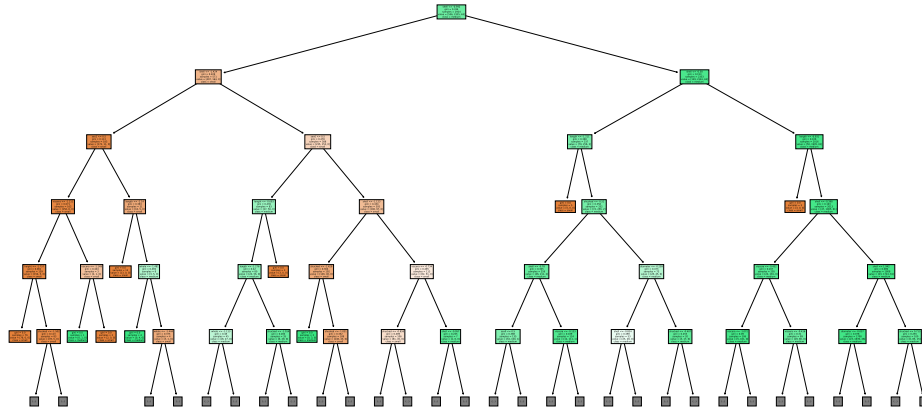


Abb. 13.9: Entscheidungsbaum, begrenzt auf 5 Ebenen.

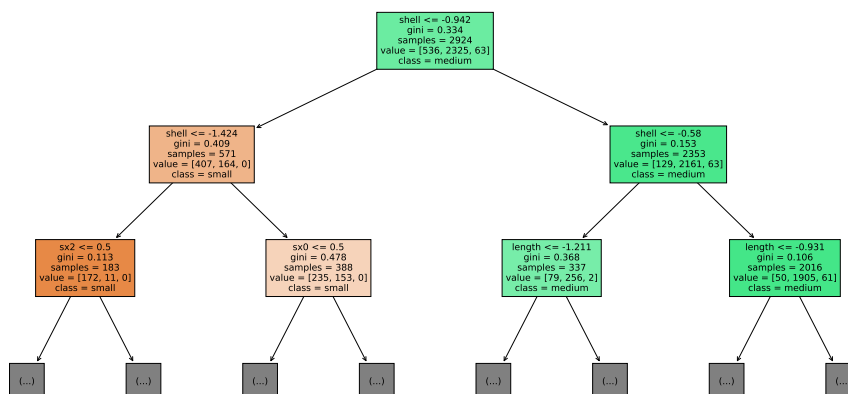


Abb. 13.10: Entscheidungsbaum, begrenzt auf 2 Ebenen.

Ergebnis für unser Modell (max_depth=2):

Quellcode 13.1: Baumstruktur, Darstellung mit tree.export_text()

```
|--- shell <= -0.94
|   |--- shell <= -1.42
|   |   |--- sex2 <= 0.50
|   |   |   |--- truncated branch of depth 6
|   |   |   |--- sex2 > 0.50
|   |   |   |--- truncated branch of depth 4
|   |   |--- shell > -1.42
|   |   |--- sex0 <= 0.50
|   |   |   |--- truncated branch of depth 12
|   |   |   |--- sex0 > 0.50
|   |   |   |--- truncated branch of depth 15
|--- shell > -0.94
|   |--- shell <= -0.58
|   |   |--- length <= -1.21
|   |   |   |--- weights: [5.00, 0.00, 0.00] class: 0
|   |   |   |--- length > -1.21
|   |   |   |--- truncated branch of depth 13
|   |   |--- shell > -0.58
|   |   |--- length <= -0.93
|   |   |   |--- weights: [3.00, 0.00, 0.00] class: 0
|   |   |   |--- length > -0.93
|   |   |   |--- truncated branch of depth 16
```

Der Entscheidungsbaum erlaubt uns, den Einfluss der Komponenten auf die Klassifikation genauer zu analysieren. Nach dem Training (`fit()`) des Modells enthält die Objekteigenschaft `feature_importances_` eine Liste mit der Wichtigkeit jeder Komponente der Datenpunkte.

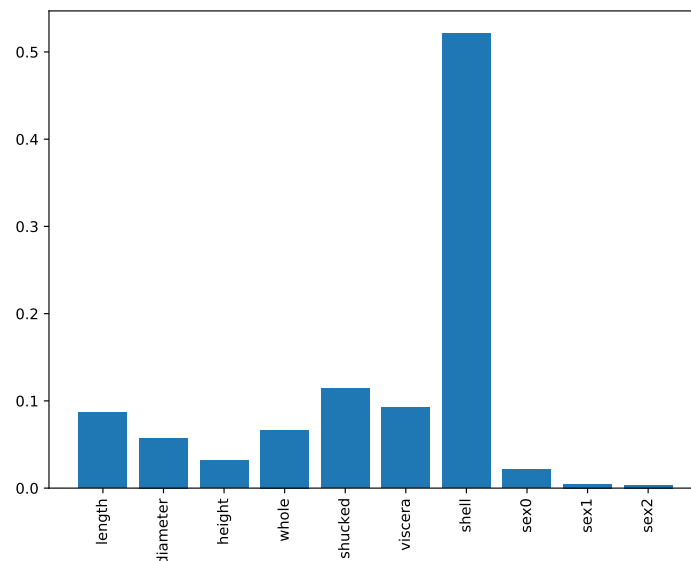
Beispiel: Wichtung der Komponenten für *Abalone*:

```
# Numerische Spalten + 3 Spalten der One-Hot-Codierung
colNames = numCols + ['sx0', 'sx1', 'sx2']
for feature, val in zip(colNames, classifier.feature_importance_):
    print(feature, val)
```

Wir erhalten:

```
length      : 0.087
diameter    : 0.057
height      : 0.032
whole       : 0.067
shucked     : 0.115
viscera     : 0.092
shell       : 0.521
sex0        : 0.021
sex1        : 0.005
sex2        : 0.004
```

Wir können wir diese auch mit einem Barplot visualisieren.



Notebook

Erzeugen der Diagramme: `abalone-plot-tree.ipynb`. *Achtung:* Die Erzeugung des Diagramms für den kompletten Entscheidungsbaum dauert relativ lange.

13.4 Aufgaben

Aufgabe 13.1 Vergleich mit Vorlesung Data Mining

Für die Teilnehmer der Vorlesung Data Mining: Wählen Sie ein Beispiel aus der Vorlesung oder Übung zum Entscheidungsbaum aus und vergleichen Sie die dort manuell berechneten Entscheidungsgrenzen und -bäume mit den von *scikit-learn* erstellten. Falls sich die Werte unterscheiden, versuchen Sie die Ursache zu ergründen.

Aufgabe 13.2 Vergleich Entscheidungsbaum mit anderen Verfahren., Hausaufgabe (10 Punkte)

Erzeugen Sie auf den von Ihnen ausgewählten Datensatz einen Entscheidungsbaum und ein weiteres Klassifikationsmodell. Versuchen Sie für den Entscheidungsbaum die optimale Tiefe zu ermitteln, für die der Fehler auf den Testdaten am geringsten ist.

Vergleichen Sie die Genauigkeit auf Trainings- und Testdaten mit mindestens zwei weiteren klassischen Verfahren (*KNN*, Logistische Regression, Naive-Bayes-Klassifikation). Nutzen Sie auch die Confusion Matrix für die Fehleranalyse.

Wählen Sie zwei Komponenten Ihrer Daten aus, die einen möglichst großen Einfluss auf die Klassifikation haben, berechnen Sie für diese ein neues Entscheidungsbaum-Modell und stellen Sie die Entscheidungsgrenzen und die Datenpunkte graphisch dar.

Ensemble-Methoden, Random Forests

14.1 Grundlagen, Verfahren

Die Genauigkeit von Vorhersagemodellen kann häufig durch Kombination verschiedener Modelle verbessert werden. Grundgedanke ist dabei, dass die einzelnen Modelle unterschiedliche Fehler machen, die als zufällig und stochastisch unabhängig betrachtet werden. Nehmen wir dazu drei verschiedene Modelle mit einer Vorhersagegenauigkeit von je $p = 0.8$ an und fragen, mit welcher Wahrscheinlichkeit mindestens 2 der Modelle die korrekte Klasse voraussagen:

$$\begin{aligned} P(\text{gesamt}) &= P(2 \text{ Modelle korrekt}) + P(3 \text{ Modelle korrekt, 1 Modell falsch}) \\ &= p^3 + 3 \cdot (p^2 \cdot (1 - p)) \\ &= 0.8^3 + 3 \cdot 0.8^2 \cdot 0.2 \\ &= 0.896 \end{aligned}$$

Das ist eine Verbesserung von mehr als 10% gegenüber des Einzelmodells.

Wir können dazu entweder

- völlig verschiedene Verfahren kombinieren und Mehrheitsentscheidung wählen (*Voting*),
- wie oben, aber Kombination der Einzelentscheidungen durch ein weiteres Lernverfahren (*Stacking*),
- mehrfach das gleiche Verfahren mit unterschiedlichen Modellparametern trainieren und daraus ein gemeinsames Modell bilden (*Bagging and Pasting*),
- Verfahren sequentiell verketteten, wobei nachfolgende Modelle gezielt versuchen, die Fehlklassifikationen der Vorgänger zu beseitigen (*Boosting*).

14.2 Voting

Für ein Voting-Modell müssen wir zunächst die einzelnen Klassifikationsobjekte erstellen, die wir dann in einem Voting-Modell zusammenfassen.

```
from sklearn import ensemble

classifier = ensemble.VotingClassifier(
    estimators=[
        ('name', klassifikator),
```

```
...
],
voting='strategie',
...
)
```

Die einzelnen Klassifikationsobjekte werden als Liste von Tupeln, bestehend aus einem String zur Bezeichnung und dem eigentlichen Objekt, übergeben. Der Parameter `voting` beschreibt die Strategie der Auswahl. Die Strategie `hard` wählt die Mehrheitsentscheidung. Mit der Strategie `soft` werden statt der harten Klassenvorhersage die Wahrscheinlichkeitsvorhersagen der einzelnen Klassifizierer verwendet und die Klasse mit der höchsten kumulierten Gesamtwahrscheinlichkeit verwendet. Diese Strategie kann nur verwendet werden, wenn die Einzelklassifizierer alle die Methode `predict_proba()` unterstützen.

Das Training dieses Klassifizierers trainiert automatisch alle hier verwendeten Klassifizierer automatisch.

Beispiel: Kombination von *KNN*, *Logistic Regression* und Entscheidungsbaum (mit optimaler Tiefe 4) zu einem Voting-Modell.

```
classifier = ensemble.VotingClassifier([
    ('tree', tree.DecisionTreeClassifier(max_depth=4)),
    ('knn', neighbors.KNeighborsClassifier()),
    ('logit', linear_model.LogisticRegression(max_iter=500)),
],
voting='hard'
)
classifier.fit(Xtrain, yTrain)
```

Wir erhalten eine Genauigkeit von

- 0.895 auf den Trainings- und 0.854 auf den Testdaten.

Die Komponente `estimators_` enthält eine Liste der trainierten Einzelmodelle, die wir zum Vergleich ebenfalls bewerten können:

```
for cl in classifier.estimators_:
    print(cl.score(Xtrain, yTrain), cl.score(Xtest, yTest))
```

Wir erhalten folgende Genauigkeiten:

- *DT*: 0.893 auf den Trainings- und 0.844 auf den Testdaten,
- *KNN*: 0.904 auf den Trainings- und 0.829 auf den Testdaten,
- Logistische Regression: 0.881 auf den Trainings- und 0.843 auf den Testdaten.

Das Voting bringt uns hier also noch 1% Genauigkeitsgewinn.

Statt eines deterministischen Votings können wir die Ausgaben der Einzelmodelle wiederum als Features eines neuen Lernmodells betrachten, das die korrekte Klassifizierung aus diesen lernen soll. Dieses Verfahren nennt man *Stacking*:

```
from sklearn import ensemble

classifier = ensemble.StackingClassifier(
    estimators=[
        ('name', klassifikator),
        ...
    ],
    [
        final_estimator='zusammenfassungsklassifikator',
    ]
    ...
)
```

Die einzelnen Verfahren werden mit dem im Parameter `final_estimator` kombiniert. Vorgabe ist hier die logistische Regression.

Wenden wir statt des Votings das Stacking auf die Ergebnisse der 3 Einzelklassifizierer an, erhalten wir eine Genauigkeit von

- 0.891 auf den Trainings- und 0.847 auf den Testdaten,

also etwas schlechtere Werte als beim Voting, aber bessere als jedes der Einzelmodelle. Wir können natürlich auch noch Hyperparameter des Voting-Klassifizierers verändern oder auch diesen komplett austauschen, um nach weiteren Modellverbesserungen zu suchen.

Hinweis: Auch für Regressionsprobleme werden entsprechende Klassen für Voting und Stacking bereitgestellt: `VotingRegressor`, `StackingRegressor`. Die einzelnen Modelle müssen nun natürlich Regressionsmodelle sein, ebenso das Modell zum Zusammenfassen der Daten.

Notebook

Voting- und Stacking- Modell für *Abalone*: `abalone-voting.ipynb`.

14.3 Random Forests (Bagging and Pasting)

Wir teilen hier unsere Trainingsmenge auf und erzeugen für jede Teilmenge einen eigenen Entscheidungsbaum. Diesen *Wald* von Bäumen fassen wir in der Form zusammen, dass jeder neue Datenpunkt mit jedem dieser Bäume klassifiziert und die Klasse per Mehrheitsentscheid festgelegt wird.

Das zufällige Erzeugen der Trainingsmengen für jedes Modell geschieht durch zufälliges Ziehen von Datenpunkten aus der Gesamtmenge mit Zurücklegen, was bedeutet, dass in jedem Modell manche Datenpunkte mehrfach vorkommen und andere fehlen. Dieses Vorgehen nennt man *Bootstrapping*.

In *scikit-learn* erzeugen wir das Modell wie folgt:

```
from sklearn import ensemble
```

```
classifier = ensemble.RandomForestClassifier(n_estimators==anzahl, ...)
```

```
classifier.fit(Xtrain, ytrain)
```

Für den *Random Forest* wählt man typischerweise 10 bis 100 Einzelbäume. Die weiteren möglichen Hyperparameter entsprechen denen des einzelnen Entscheidungsbaums:

criterion Reinheitsmaß, `gini` oder `entropy`.

max_depth Maximale Baumtiefe.

min_samples_leaf Minimale Zahl von Datenpunkten pro Blatt.

min_samples_split Minimale Zahl von Datenpunkten, für der ein Knoten nochmals geteilt werden darf.

Nach dem Training können wir auf die einzelnen Entscheidungsbäume über die Komponente `estimators_` des Forest-Objekts zugreifen und uns deren Einzelgüte anzeigen.

Beispiel: *Random-Forest*-Klassifikation für *Abalone* mit 50 Einzelbäumen.

```
# Vergleich: Einfacher Baum
dt = tree.DecisionTreeClassifier()
dt.fit(Xtrain, yTrain)
print('Score Training {}, Test {}'.format(
    dt.score(Xtrain, yTrain),
    dt.score(Xtest, yTest)
```

(Fortsetzung auf der nächsten Seite)

```

))

# Random Forest
rf = ensemble.RandomForestClassifier(
    n_estimators=50,
    random_state=23
)
rf.fit(Xtrain, yTrain)
print('Score Training {}, Test {}'.format(
    rf.score(Xtrain, yTrain),
    rf.score(Xtest, yTest)
))

```

Wir erhalten folgende Genauigkeiten:

- Einzelner Entscheidungsbaum: Training: 1.0, Test: 0.775
- Random Forest: Training: 0.999, Test: 0.840

Die Komponente `estimators_` enthält nach dem Training eine Liste der einzelnen, trainierten Entscheidungsbäume. Wir können zum Vergleich die Genauigkeiten jedes einzelnen Baums für unsere Trainingsdaten bestimmen. Wir beschränken uns hier auf die ersten 10 Bäume.

```

for i in range(10):
    cl = rf.estimators_[i]
    print('Tree {}, Score Train {:.3f} Test {:.3f}'.format(i,
        cl.score(Xtrain, yTrain),
        cl.score(Xtest, yTest)
    ))

```

Wir erhalten:

```

Tree 0, Score Train 0.939 Test 0.777
Tree 1, Score Train 0.935 Test 0.789
Tree 2, Score Train 0.936 Test 0.767
Tree 3, Score Train 0.931 Test 0.764
Tree 4, Score Train 0.938 Test 0.771
Tree 5, Score Train 0.939 Test 0.765
Tree 6, Score Train 0.931 Test 0.759
Tree 7, Score Train 0.937 Test 0.770
Tree 8, Score Train 0.939 Test 0.788
Tree 9, Score Train 0.940 Test 0.778

```

Man sieht, dass die einzelnen Bäume etwa ähnliche Genauigkeiten liefern wie unser Einzelbaum. Hier fallen die teilweise gedoppelten und die fehlenden Trainingsdaten kaum ins Gewicht (bei kleineren Trainingsdatensätzen ergibt sich teilweise eine deutliche Verschlechterung für die Einzelbäume). Die gemeinsame Entscheidung bringt dagegen eine erhebliche Steigerung in der Genauigkeit auf den Testdaten.

Beschränken wir uns auf zwei Komponenten der Trainingsdaten, können wir die Entscheidungsgrenzen der einzelnen Bäume auch wieder graphisch darstellen. Wir verwenden hier die wichtigsten Komponenten `shell` und `diameter` und beschränken zu besserer Darstellung die Baumhöhe auf 5.

Wir erhalten mit diesen Einschränkungen folgende Genauigkeiten:

- Einzelbaum: Training: 0.893, Test: 0.804
- Random Forest: Training: 0.893, Test: 0.814

Das *Bagging-and-Pasting*-Verfahren ist nicht auf Entscheidungsbäume beschränkt, sondern kann im Prinzip mit beliebigen Klassifikationsverfahren verwendet werden. Dafür wird die Klasse `BaggingClassifier` zur Verfügung gestellt:

```

from sklearn import ensemble

```

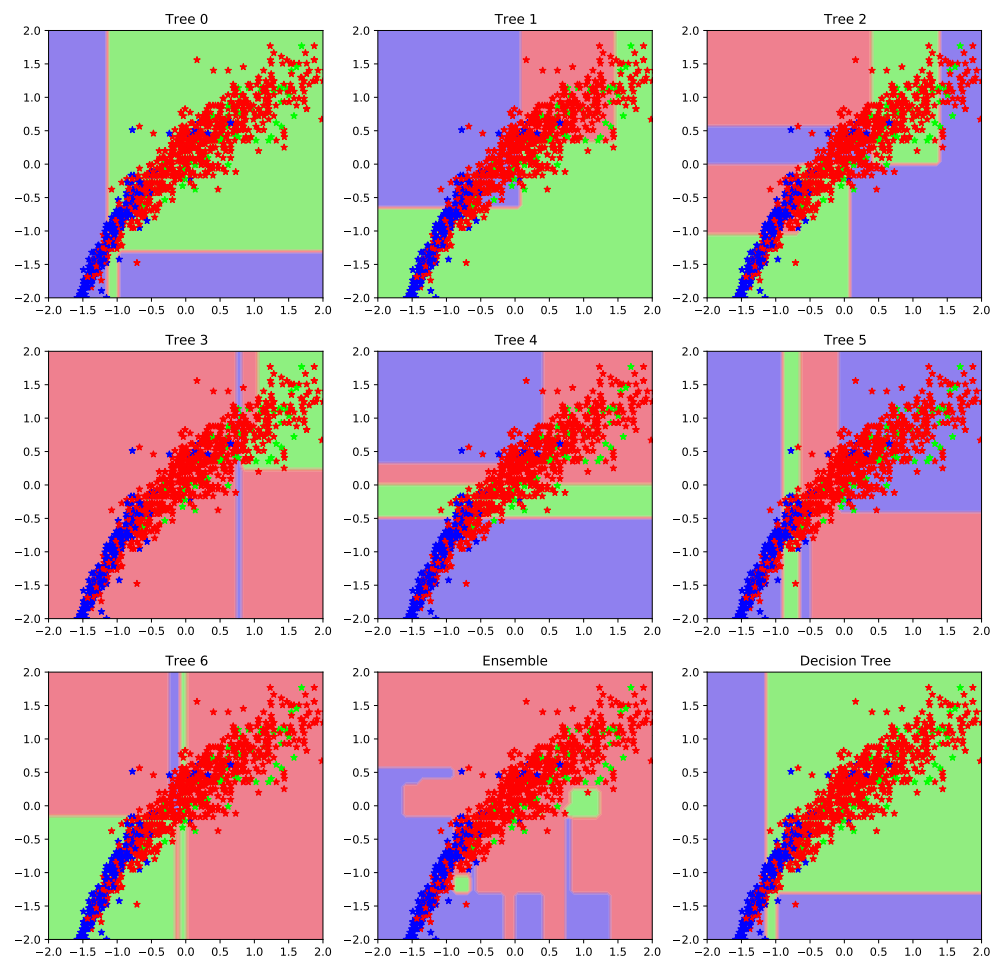


Abb. 14.1: Entscheidungsgrenzen der ersten 8 Einzelbäume, des Ensembles und eines einzelnen Entscheidungsbaums.

```
classifier = ensemble.BaggingClassifier(  
    base_estimator=Klassifikationsobjekt,  
    [  
        n_estimators=anzahlKlassifikatoren,  
    ]  
    ...  
)
```

Beispiel: Bagging mit *KNN*-Klassifikatoren. Die Verwendung ist verfahrensbedingt nur sinnvoll, wenn zur Klassifikation mehrere Nachbarn verwendet werden.

```
classifier = ensemble.BaggingClassifier(  
    base_estimator = neighbors.KNeighborsClassifier(n_neighbors=25)  
)  
classifier.fit(Xtrain, yTrain)
```

Für unseren *Abalone*-Datensatz erhalten gegenüber eines einfachen *KNN*-Modells kaum eine Verbesserung. Bei Verfahren, die noch stärker von den einzelnen Datenpunkten abstrahieren, indem sie trennende Ebenen suchen, wie die *logistische Regression* oder die später behandelten *SVM*, bringt das Bagging kaum verbesserte Modelle. Die größten Verbesserungen lassen sich beim Entscheidungsbaum erzielen, weshalb für diesen mit dem *RandomForest*-Klassifizierer ein angepasstes Modell bereitgestellt wird.

Notebooks

- Random Forest: `abalone-forest.ipynb`,
- Bagging mit *KNN*: `abalone-bagging.ipynb`,
- Erweiterte Hilfsfunktionen: `abalone_helper2.py`.

14.4 Boosting

(Vertiefung)

Im Gegensatz zum Voting- und Bagging-Ansatz werden beim *Boosting* Klassifizierer nacheinander ausgeführt in der Form, dass folgende Klassifizierer die Fehler der vorangegangenen korrigieren.

Das *Adaboost*-Verfahren erstellt dazu eine Folge von Klassifizierern. Beim ersten Klassifizierer werden für die Berechnung des Fehlers alle Datenpunkte gleich gewichtet. Anschließend werden die fehlerhaft klassifizierten Datenpunkte mit höheren Gewichten als die korrekt klassifizierten versehen und damit ein weiterer Klassifizierer wieder mit allen Daten trainiert, der aber nun durch die höhere Wichtung bemüht ist, diese Punkte richtig zu klassifizieren (wodurch vielleicht bisher richtig klassifizierte Punkte nun falsch klassifiziert werden). Auch hier betrachten wir wieder die falsch klassifizierten Daten, versehen sie mit höheren Gewichten und trainieren einen weiteren Klassifizierer. Zum Schluss treffen alle Klassifizierer gemeinsam die Entscheidung, wobei die Wertung der ersten Klassifizier stärker gewichtet wird als die späterer Klassifizierer.

```
from sklearn import ensemble  
  
classifier = ensemble.AdaBoostClassifier(  
    [base_estimator=KlassifizierungsObjekt, ]  
    [n_estimators=anzahl, ]  
    ...  
)
```

Das verwendete Klassifikationsverfahren kann frei gewählt werden, wobei dieses Verfahren insbesondere bei Entscheidungsbäumen häufig gute Resultate erzielt. Ohne Angabe wird deshalb auch dieses Verfahren benutzt. Weitere Parameter sind die Lernrate (die die Wichtung der fehlerhaften Datenpunkte und der Einzelklassikatoren bei der Zusammenfassung des Ergebnisses beeinflusst) und der verwendete Algorithmus.

Wenden wir dieses auf unseren Datensatz *Abalone* mit dem Basisklassifizierer Entscheidungsbaum an:

```
ada = ensemble.AdaBoostClassifier(
    base_estimator=tree.DecisionTreeClassifier(max_depth=13),
    n_estimators=20
)
ada.fit(Xtrain, yTrain)
```

erhalten wir eine Genauigkeit von

- 1.0 auf den Trainings- und 0.832 auf den Testdaten (schlechter als der optimale Einzelbaum, aber besser als Baum mit Tiefe 13).

Hinweis: Wird keine Baumtiefe für den Base Estimator angegeben, wird nur ein Einzelbaum und kein Ensemble berechnet. Wir sollten uns bei merkwürdigen Ergebnissen deshalb nach dem Training davon überzeugen, dass tatsächlich ein Ensemble erzeugt wurde:

```
print(len(ada.estimators_))
```

Beim *Gradient Boosting* betrachten wir nicht die Fehler der einzelnen Datenpunkte und versuchen diese stärker zu bewerten, sondern wir berechnen den Differenzvektor zwischen den wahren Werten y und unserer ersten Vorhersage $\hat{y}$

$$y_1 = y - \hat{y}$$

Nun erstellen wir ein neues Modell, dass die Werte y_1 vorhersagen soll. Dieses Verfahren setzen wir mehrfach fort. Unsere Gesamtvorhersage ergibt sich aus der Summe der Vorhersage der Einzelmodelle (Vorhersagewert der Klasse aus dem ersten Modell plus die Differenzen bei falschen Vorhersagen aus den Folgemodellen).

Das *Gradient-Boosting*-Verfahren basiert auf der Annahme einer stetigen Zielgröße, was insbesondere für die Behandlung der Differenzvektoren notwendig ist. Trotzdem kann es auch für Klassifikationsprobleme eingesetzt werden, indem man einen bestimmten Vorhersagewert als Grenze zwischen den Klassen betrachtet.

Obwohl auch dieses Verfahren prinzipiell für eine Vielzahl von Vorhersagemodellen als Verbesserung eingesetzt werden kann, ist es in *scikit-learn* lediglich für den Entscheidungsbaum implementiert.

```
from sklearn import ensemble

classifier = ensemble.GradientBoostClassifier(
    [n_estimators=anzahl, ]
    ...
)
```

Neben einigen weiteren Parametern zur Ensemblebildung können wir hier alle Hyperparameter für den Entscheidungsbaum übergeben wie maximale Baumtiefe (`max_depth`) oder minimale Blattgröße (`min_samples_leaf`).

Beispiel: Gradient Boosting für *Abalone*.

```
gbt = ensemble.GradientBoostingClassifier(
    max_depth=13,
    n_estimators=20
)
gbt.fit(Xtrain, yTrain)
```

Wir erhalten eine Genauigkeit von

- 1.0 auf den Trainings- und 0.805 auf den Testdaten, also ebenfalls keine Verbesserung gegenüber dem optimalen Einzelbaum oder dem Random Forest.

Notebook

AdaBoost und Gradient Boost für *Abalone*: `abalone-boosting.ipynb`.

14.5 Aufgaben

Aufgabe 14.1 *Random Forest*

Erstellen Sie für Ihren Datensatz Random-Forest-Modelle mit verschiedenen Parametern und versuchen Sie, die Vorhersage zu verbessern.

Aufgabe 14.2 *Voting*

Kombinieren Sie nun Ihr bestes Random-Forest-Modell mit zwei anderen Klassifikationsverfahren zu einem Voting- oder Stacking-Modell. Lässt sich die Genauigkeit weiter verbessern?

Support Vector Machines

- *Grundidee*
- *Duales Problem*
- *Nichtlineare (Kernel-)SVM*
- *Soft Margin*
- *SVM in scikit-learn*
- *Entscheidung zwischen Linearer oder Kernel-SVM*
- *Regression*
- *Aufgaben*

15.1 Grundidee

SVM sind eines der leistungstärksten, klassischen Klassifizierungsverfahren des *Machine Learning*. Die ersten Ideen entstanden in den 60er Jahren, das heute übliche Vorgehen wurde um 1995 entwickelt.

Die Grundidee ist die Suche nach einer optimalen Trennebene zwischen den Datenpunkten zweier Klassen.

Um ein möglichst gut generalisierendes Modell zu erhalten, wählen wir die Ebene so, dass der Abstand *aller* Datenpunkte zur Ebene möglichst groß ist.

Eine n -dimensionale Ebene lässt sich durch einen Stellungsvektor $\mathbf{w}$ und ihren Abstand b vom Ursprung beschreiben:

$$\mathbf{x} \in H \Leftrightarrow \langle \mathbf{w}, \mathbf{x} \rangle + b = 0$$

Die Klammern $\langle \cdot, \cdot \rangle$ bezeichnen das Skalarprodukt. Liegt ein Punkt nicht in dieser Ebene, dann ist sein Abstand von ihr:

$$d = \frac{\langle \mathbf{w}, \mathbf{x} \rangle + b}{\|\mathbf{w}\|}$$

Je nachdem, ob ein Punkt oberhalb (weiter vom Nullpunkt entfernt) oder unterhalb der Ebene liegt, ist dieser Wert positiv oder negativ, wodurch wir die beiden Seiten unterscheiden können. Die Länge des Vektors $\mathbf{w}$ spielt keine

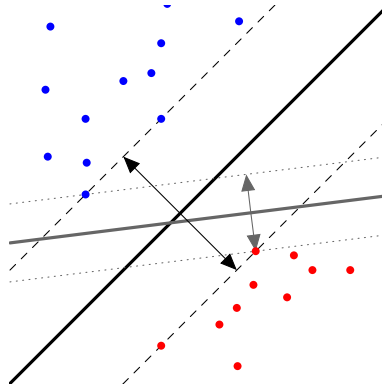


Abb. 15.1: Verschiedene Trennebenen

Rolle, da wir durch seine Länge teilen, lediglich seine Richtung bestimmt die Ebene und damit die Abstände der Datenpunkte von ihr.

Um unsere Datenpunkte $\mathbf{x}_i$, $i = 1, \dots, M$ in zwei Klassen aufzuteilen, verwenden wir nun die Klassenlabel $+1$ für Punkte oberhalb und -1 für Punkte unterhalb der Ebene und multiplizieren die Abstandsgleichung mit dem jeweiligen Klassenlabel t durch. Damit erhalten wir für jeden Punkt einen positiven Abstandswert. Dieser soll für eine gute Trennung möglichst groß sein, wir müssen die Parameter $\mathbf{w}$ und b der Ebene also so optimieren, dass der kleinste Punktabstand zur Ebene möglichst groß wird.

$$\max_{\mathbf{w}, b} \left\{ \min_i \left\{ \frac{(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)t_i}{\|\mathbf{w}\|} \right\} \right\}$$

Da die innere Optimierung nur von den Datenpunkten, aber nicht von $\mathbf{w}$ abhängt, können wir den Quotient herausziehen:

$$\max_{\mathbf{w}, b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_i \{ (\langle \mathbf{w}, \mathbf{x}_i \rangle + b)t_i \} \right\}$$

Da Doppeloptimierungsprobleme schwierig zu lösen sind, nutzen wir nun aus, dass wir die Länge von $\mathbf{w}$ frei wählen dürfen. Wir nehmen einmal an, wir haben das Optimierungsproblem schon gelöst und herausgefunden, dass der kleinste Abstand eines Punktes von der Hyperebene κ ist. Dann gilt für alle Punkte (nach Durchmultiplizieren mit $\|\mathbf{w}\|$).

$$(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)t_i \geq \kappa \|\mathbf{w}\|$$

Wählen wir nun die Länge des Vektors $\mathbf{w}$ gerade $1/\kappa$, dann wird die rechte Seite der Ungleichung gerade 1. Damit ersetzen wir die innere Optimierung durch die Nebenbedingung

$$(\langle \mathbf{w}, \mathbf{x}_i \rangle + b)t_i \geq 1 \quad \forall i$$

und erhalten das vereinfachte Optimierungsproblem für obige Nebenbedingungen.

$$\max_{\mathbf{w}, b} \left\{ \frac{1}{\|\mathbf{w}\|} \right\}$$

Dies vereinfachen wir noch ein wenig. Da der Reziprokwert eine monoton fallende Funktion ist, können wir statt dessen auch das Minimum der Norm von $\mathbf{w}$ bestimmen.

$$\min_{\mathbf{w}, b} \{ \|\mathbf{w}\| \}$$

Statt dessen können wir auch das Quadrat der Norm minimieren (womit eine Wurzelberechnung entfällt).

$$\begin{aligned} \min_{\mathbf{w}, b} \{ \|\mathbf{w}\|^2 \} &= \min_{\mathbf{w}, b} \{ \langle \mathbf{w}, \mathbf{w} \rangle \} \\ &= \min_{\mathbf{w}, b} \left\{ \sum_{i=1}^M w_i^2 \right\} \end{aligned}$$

Statt $\|\mathbf{w}\|^2$ schreibt man meist $\frac{1}{2}\|\mathbf{w}\|^2$, was rein ästhetische Gründe hat (wir müssen zum Durchführen der Optimierung die Ableitung berechnen, dann kürzt sich dieser Faktor weg). Unser Optimierungsproblem für die primale SVM sieht also wie folgt aus:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \left\{ \frac{1}{2} \|\mathbf{w}\|^2 \right\} \\ \text{s. t.} \quad & (\langle \mathbf{w}, \mathbf{x}_i \rangle + b) t_i \geq 1, \quad i = 1 \dots M \end{aligned}$$

15.2 Duales Problem

Wie zu jedem Optimierungsproblem können wir das *duale Problem* formulieren. Mit dieser Formulierung drücken wir die Klassifikationsentscheidung nicht durch die trennende Hyperebene aus, sondern durch die Datenpunkte, die sich am dichtesten an der Hyperebene befinden. Diese Punkte bzw. Vektoren «stützen» also den Rand, also das leere Band um die Trennebene, deshalb werden sie als *Support Vectors* bezeichnet, woraus sich der Name des Verfahrens ableitet.

Das *duale Problem* ist folgendes Optimierungsproblem:

$$\begin{aligned} \min_{\mathbf{a}} \quad & \left\{ \frac{1}{2} \sum_{i=1}^M \sum_{k=1}^M a_i a_k t_i t_k \langle x_i, x_k \rangle - \sum_{i=1}^M a_i \right\} \\ \text{s. t.} \quad & a_i \geq 0, \quad \sum_{i=1}^M a_i t_i = 0, \quad i = 1, \dots, M \end{aligned}$$

Wir berechnen durch die Optimierung den Faktor a_i für jeden Datenpunkt $\mathbf{x}_i$. Dieser ist nur dann von Null verschieden, wenn der Datenpunkt auf dem Rand liegt. Nur diese Datenpunkte werden weiter benötigt, wir bezeichnen sie als *Supportvektoren*. Alle anderen Datenpunkte gehen nicht mehr ins Modell ein und können «entsorgt» werden. Um einen neuen Punkt zu klassifizieren, berechnen wir

$$\begin{aligned} y(\mathbf{x}) &= \sum_{s \in S} a_s t_s \langle \mathbf{x}, \mathbf{x}_s \rangle + b \quad S \text{ Indices der Supportvektoren} \\ \text{mit } b &= \frac{1}{|S|} \sum_{s \in S} \left(t_s - \sum_{u \in S} a_u t_u \langle \mathbf{x}_s, \mathbf{x}_u \rangle \right) \end{aligned}$$

Schauen wir uns die Gleichungen an, sehen wir, dass die Datenpunkte sowohl im Optimierungsproblem als auch in der Klassifikation nur in Form von Skalarprodukten auftreten. Diese Eigenschaft werden wir im folgenden ausnutzen.

Hinweis: Sowohl das primale als auch das duale Optimierungsproblem sind sogenannte *QOP* mit Randbedingungen. Für diese gibt es effiziente iterative Lösungsverfahren. Je nach Problem ist die primale oder die duale Form effizienter lösbar.

15.3 Nichtlineare (Kernel-)SVM

Oftmals lassen sich Daten generell nicht durch eine Hyperebene trennen. Im folgenden Beispiel sind die beiden Klassen in kreisförmigen Ringen um ein Zentrum angeordnet.

Wir können diese Daten allerdings problemlos trennen, wenn wir eine zusätzliche Komponente, den Abstand vom Nullpunkt (Radius) berechnen und jedem Datenpunkt hinzufügen.

Wir haben die Daten hier *nichtlinear* durch die Komponente $x_3^2 = r^2 = x_1^2 + x_2^2$ in einen höherdimensionalen Raum transformiert, in dem es nun einfach ist, eine Trennebene zu finden.

Leider ist es nicht einfach, die richtige Transformationsfunktion zu finden, zum anderen erfordert diese unter Umständen sehr viele Parameter (z. B. die Koeffizienten für ein Polynom), die alle geeignet gewählt werden müssen.

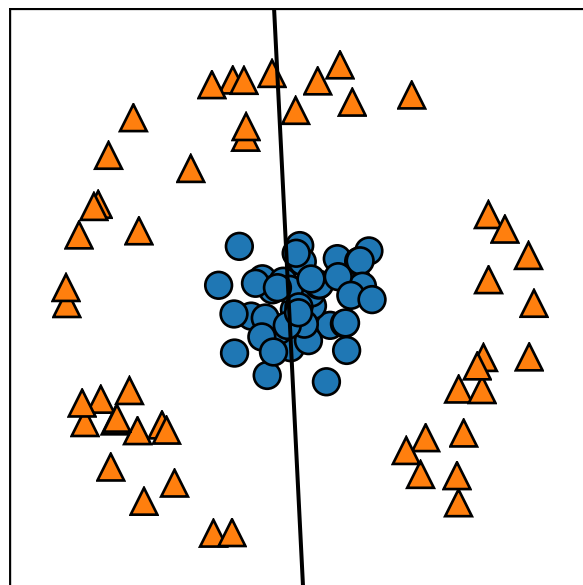


Abb. 15.2: Linear nicht trennbare Daten

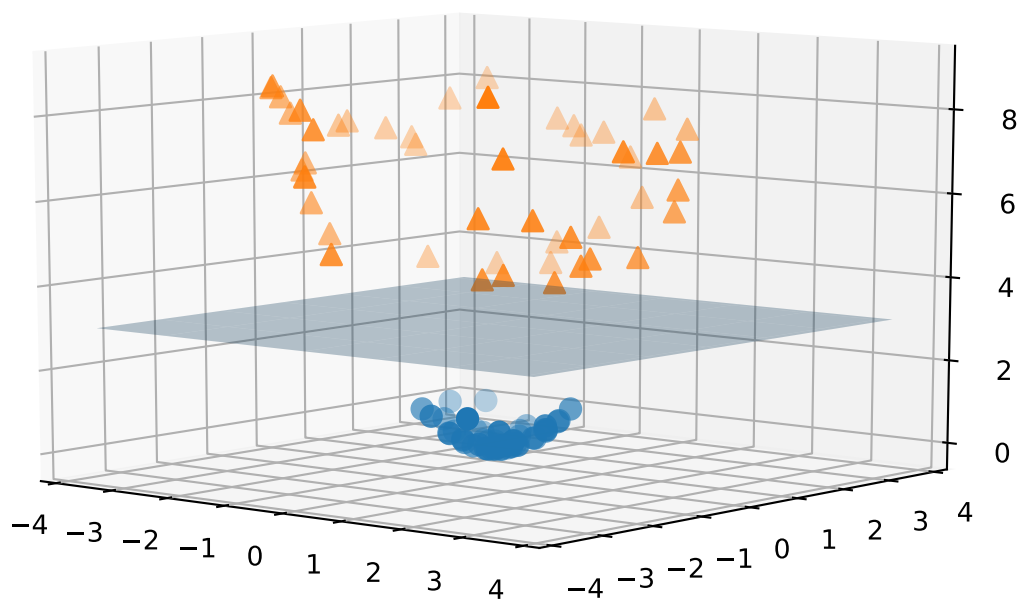


Abb. 15.3: Trennung nach nichtlinearer Transformation

Aufgabe 15.1 Versuchen Sie, eine geeignete Transformation zu finden, nach deren Anwendung der synthetische Datensatz `make_moons()` linear getrennt werden kann.

Es kann auch erforderlich sein, eine Transformation in einen unendlichdimensionalen Raum vorzunehmen, in dem wir auf einem Computer nichts berechnen können. Man geht deshalb etwas anders vor und sucht nicht nach einer Transformationsfunktion für einzelne Datenpunkte, sondern ersetzt die Skalarprodukte zwischen den Datenpunkten im *dualen Problem* durch eine sogenannte *Kernel Function*. Statt $\langle \mathbf{x}_i, \mathbf{x}_k \rangle$ berechnen wir die Funktion $K(\mathbf{x}_i, \mathbf{x}_k)$, die zwei Vektoren auf eine reelle Zahl abbildet. Eine solche *Kernel Function* oder kurz *Kern* muss sich so ähnlich wie ein Skalarprodukt verhalten: sie ist symmetrisch (wir können die Reihenfolge der Parameter vertauschen) und positiv, wenn beide Parameter gleich (und nicht der Nullvektor) sind.

Folgende Funktionen haben sich als praktisch gut geeignete Kernelfunktionen herausgestellt.

- Gauß-Kernel bzw. *Radial Basis Function*:

$$K(\mathbf{x}_i, \mathbf{x}_k) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_k\|^2}$$

γ ist ein Hyperparameter. Je größer der Wert, desto komplexer wird das Modell.

- Polynomieller Kernel:

$$K(\mathbf{x}_i, \mathbf{x}_k) = (\gamma \langle \mathbf{x}_i, \mathbf{x}_k \rangle + r)^d$$

Die Hyperparameter sind γ , d und r (Exponent).

Nach Aussage der *LIBSVM*-Dokumentation hat die konkrete Wahl des Kernels nur geringen Einfluss auf die Modellgüte. Hier wird empfohlen, vorzugsweise den *RBF-Kernel* zu verwenden, da er zum einen nur einen Hyperparameter benötigt und zum anderen günstige Berechnungseigenschaften aufweist.

15.4 Soft Margin

Oft lassen sich unsere Daten aber auch mit einer nichtlinearen Kernelfunktion nicht trennen, da sie Ausreißer oder fehlerklassifizierte Punkte enthalten. Um trotzdem erfolgreich zu klassifizieren, erlauben wir für jeden Datenpunkt $\mathbf{x}$ einen *Schlupf* ξ , der angibt, wie weit ein Punkt in Richtung Rand verschoben werden muss, damit unsere Nebenbedingungen eingehalten werden. Dabei soll die notwendige Gesamtkorrektur möglichst klein bleiben.

Wir erweitern dazu unser ursprüngliches Minimierungsproblem, indem wir die Gesamtkorrektur geeignet hinzufügen:

$$\begin{aligned} \min_{\mathbf{w}, b, \xi} \quad & \left\{ \|\mathbf{w}\|^2 + C \cdot \sum_{i=1}^N \xi_i \right\}, \quad C > 0 \\ \text{s. t.} \quad & (\langle \mathbf{w}, \mathbf{x}_i \rangle + b)t_i + \xi_i \geq 1 \quad \forall i \\ & \xi_i \geq 0 \quad \forall i \end{aligned}$$

Während wir für die Norm unserer Ebenenparameter $\mathbf{w}$ stets die *euklidische Norm* wählen, benutzt man für den Fehlerterm entweder die Summe der Schlupfvariablen (*Lasso Regularisierung*) oder ersetzt diese durch deren *euklidische Norm* (*Ridge Regularisierung*). Die erste Form führt dazu, dass nur wenige Punkte innerhalb des Randes bzw. auf der falschen Seite der Hyperebene liegen dürfen, die zweite, dass die falsch klassifizierten Punkte nicht weit entfernt von der richtigen Seite des Randes sein können.

Der Parameter C heißt *Regularisierungsparameter* und ist ein *Hyperparameter* des Modells, der geeignet gewählt werden muss. Ein kleiner Wert führt zu einer starken Generalisierung (sehr viele Punkte dürfen falsch klassifiziert werden) und zum *Underfitting*, ein hoher Wert zu einer hohen Komplexität des Modells (die meisten Punkte werden zwar richtig klassifiziert, aber dadurch der trennende Rand sehr schmal) und damit zum *Overfitting*. Geht C gegen unendlich, erlauben wir praktisch keinen Schlupf, wir haben dann also das ursprüngliche *Hard Margin* Problem vor uns.

Die Soft-Margin-Erweiterung lässt sich auch auf das *duale Problem* übertragen. Die Schlupfvariablen verschwinden

dabei, nur der Parameter C geht noch in das Optimierungsproblem ein (und zwar in die Nebenbedingungen).

$$\begin{aligned} \min_{\mathbf{a}} \quad & \left\{ \frac{1}{2} \sum_{i=1}^N \sum_{k=1}^N a_i a_k t_i t_k k(\mathbf{x}_i, \mathbf{x}_k) - \sum_{i=1}^N a_i \right\} \\ \text{s. t.} \quad & 0 \leq a_i \leq C \\ & \sum_{i=1}^N a_i t_i = 0 \end{aligned}$$

Auch hier benötigen wir im Modell nur noch die Datenpunkte, deren berechneter Faktor a_i ungleich 0 ist (*Support-Vektoren*).

15.5 SVM in scikit-learn

In *scikit-learn* werden uns sowohl lineare als auch nichtlineare Modelle für das *SVM*-Verfahren mit *Soft margin* zur Verfügung gestellt.

```
from sklearn import svm
```

```
classifier = svm.LinearSVC(C=softMargin, dual=boolWert)
classifier.fit(Xtrain, ytrain)
```

Ohne die Angabe von Parametern werden $C=1$ gewählt und das duale Problem gelöst. Da hier oft die voreingestellte Zahl der Iterationen für eine ausreichende Genauigkeit nicht ausreicht, kann man versuchen, diese Zahl zu erhöhen (Parameter `max_iter`) oder das primale Problem zu lösen.

Beispiel: Lineare *SVM* für unseren *Abalone*-Datensatz. Die metrischen Komponenten müssen für dieses Verfahren unbedingt geeignet skaliert werden, da das Verfahren die euklidischen Abstände der Datenpunkte auswertet. Es empfiehlt sich, das primale Problem zu lösen, da für das duale Problem die Lösung nur langsam konvergiert.

Die Klasseneinteilung und die Reduktion auf den zweidimensionalen Datensatz mittels *Hauptkomponentenanalyse* erfolgte wie im Kapitel *Klassifikationsverfahren*, um die Ergebnisse vergleichbar zu machen. Sinnvoller ist statt der *PCA* meist die Auswahl der Komponenten mit dem höchsten Einfluss auf die Zielgröße, wie wir dies beim Entscheidungsbaum (*Entscheidungsbäume*) und im Abschnitt *Regression* getan haben.

```
classifier = svm.LinearSVC(dual=False)
classifier.fit(Xtrain, yTrain)
```

Wir erhalten folgende Genauigkeiten:

- Training (alle Features): 68%
- Test (alle Features): 66%
- Training (zwei Features): 74%
- Test (zwei Features): 54%

Wir erhalten ähnliche Werte wie bei der logistischen Regression.

Hinweis: Aufgrund der anderen Klasseneinteilung sind diese Werte nicht direkt mit unseren Ergebnissen des Entscheidungsbaums vergleichbar.

Für den reduzierten Datensatz erhalten wir folgende Entscheidungsgrenzen:

Wir sehen wieder geradlinige Entscheidungsgrenzen wie bei der logistischen Regression, jedoch eine andere Lage.

Wir haben nur einen Hyperparameter C , für den wir nach dem besten Wert suchen können. Um ein großes Intervall zu durchmustern, bietet sich eine logarithmische Aufteilung des Intervalls an:

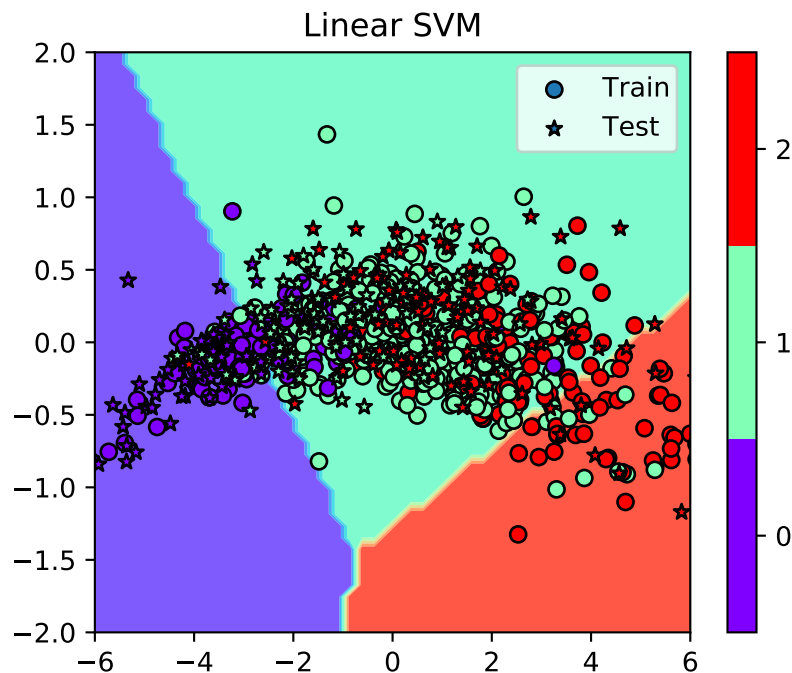


Abb. 15.4: Entscheidungsgrenzen der linearen SVM

```
XtrainHyp, Xvalid, yTrainHyp, yValid = model_selection.train_test_split(
    Xtrain, yTrain, test_size=0.2, random_state=23)
params = np.logspace(-4, 4, 19)
for param in params:
    cl = svm.LinearSVC(C=param, dual=False)
    cl.fit(XtrainHyp, yTrainHyp)
    scoreValid = cl.score(Xvalid, yValid)
    # ... (Auswahl bester Parameter)
```

Wir erhalten folgende Genauigkeiten in Abhängigkeit von C:

Für zu kleine Werte erkennen wir das Underfitting, es tritt jedoch kaum ein Overfitting für zu große Werte von C auf. Damit erhalten wir mit dem optimalen Wert für C auch kaum eine Modellverbesserung.

Eine Kernel-SVM erzeugen wir ähnlich:

```
from sklearn import svm

classifier = svm.SVC(kernel='kernName', C=softMargin, gamma=kernParameter ...)
classifier.fit(Xtrain, ytrain)
```

Hier müssen wir das duale Problem benutzen, da nur in diesem Falle der Kernel-Trick verwendet werden kann. Der voreingestellte Kernel ist `rbf`, also der *Gauß-Kernel*, der über den Parameter `gamma` gesteuert wird. Bei anderen Kernel-Funktionen tauchen abweichende Parameter auf. Wir können mit dieser Klasse auch eine lineare *SVM* erzeugen, indem wir den Kernel `dot` (Skalar- bzw. Punktprodukt) verwenden. Das ist allerdings nicht zu empfehlen, da das hier verwendete Optimierungsverfahren langsamer als das der linearen *SVM* ist.

Beispiel: Kernel-*SVM* mit Vorgabewerten für den *Abalone*-Datensatz.

```
classifier = svm.SVC()
classifier.fit(Xtrain, yTrain)
```

Wir erhalten folgende Genauigkeiten:

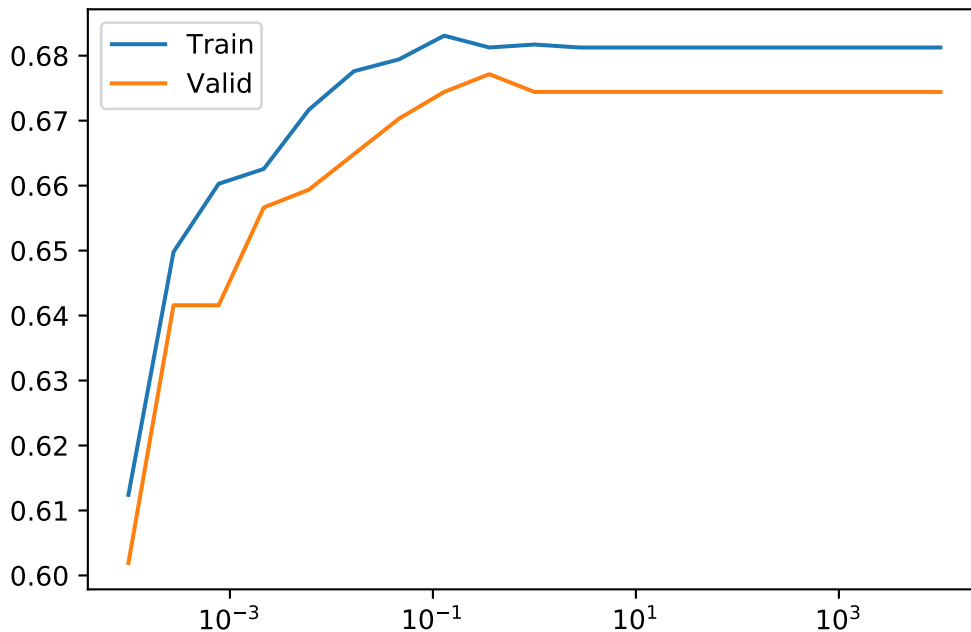


Abb. 15.5: Genauigkeit auf Trainings- und Validierungsdaten in Abhängigkeit von C.

- Training (alle Features): 71%
- Test (alle Features): 67%
- Training (zwei Fetures): 77%
- Test (zwei Features): 52%

Unsere Entscheidungsgrenzen sind nun nicht mehr geradlinig, da die Daten nichtlinear in einen hochdimensionalen Raum transformiert wurden, in dem diese eine Hyperebene bilden. Die Rücktransformation führt dann zu den gewünschten gekrümmten Grenzen im Ausgangsraum.

Wählen wir für die Hyperparameter C und gamma sehr große Werte, wird unser Modell spezifischer und neigt zum Overfitting.

Wir suchen nun für ein festes C (hier 2 gewählt) den optimalen Parameter gamma. Auch hier müssen wir wieder mehrere Zehnerpotenzen durchsuchen und wählen deshalb wieder eine logarithmische Teilung:

```
params = np.logspace(-5, 1, 10)
for param in params:
    cl = svm.SVC(gamma=param, C=2)
    cl.fit(XtrainHyp, yTrainHyp)
    scoreValid = cl.score(Xvalid, yValid)
    # ... Parameter mit optimalem Score auswählen
```

Für diesen Parameter erkennen wir links das Underfitting (schlechte Güte auf Trainings- und Validierungsdaten) und rechts das Overfitting (Genauigkeit auf den Trainingsdaten steigt, aber fällt auf der Validierungsmenge wieder ab).

Verwenden wir den optimalen Wert gamma=0.5 für das fixierte C=2, erhalten wir:

- Training (alle Features): 73%
- Test (alle Features): 68%
- Training (zwei Fetures): 77%

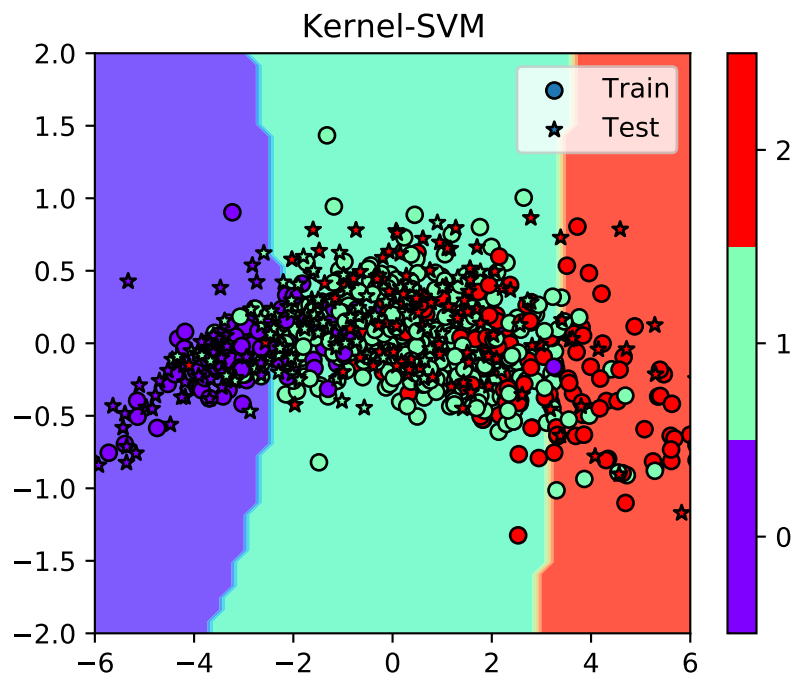
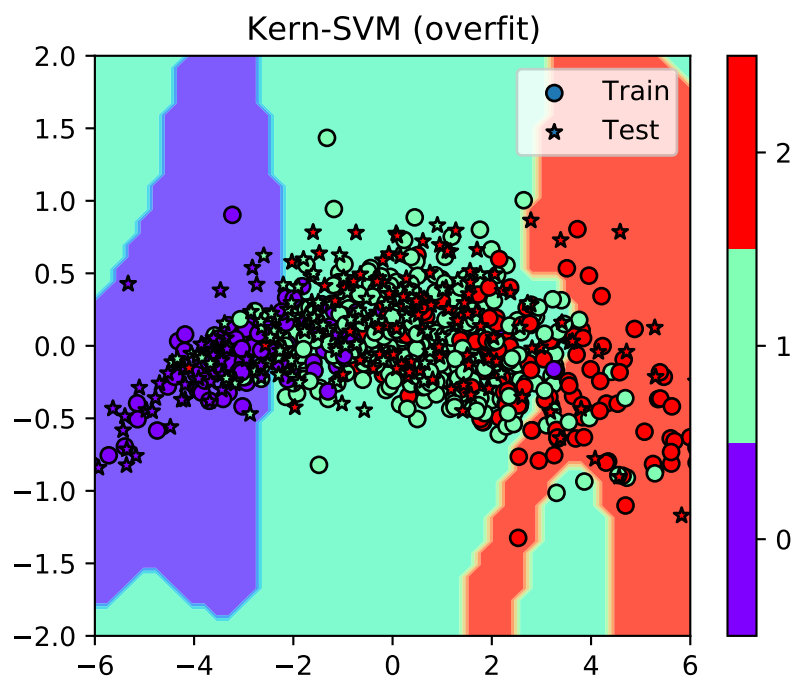


Abb. 15.6: Entscheidungsgrenzen der Kernel-SVM

Abb. 15.7: Entscheidungsgrenzen der Kernel-SVM mit $C=100$ und $\gamma=1$.

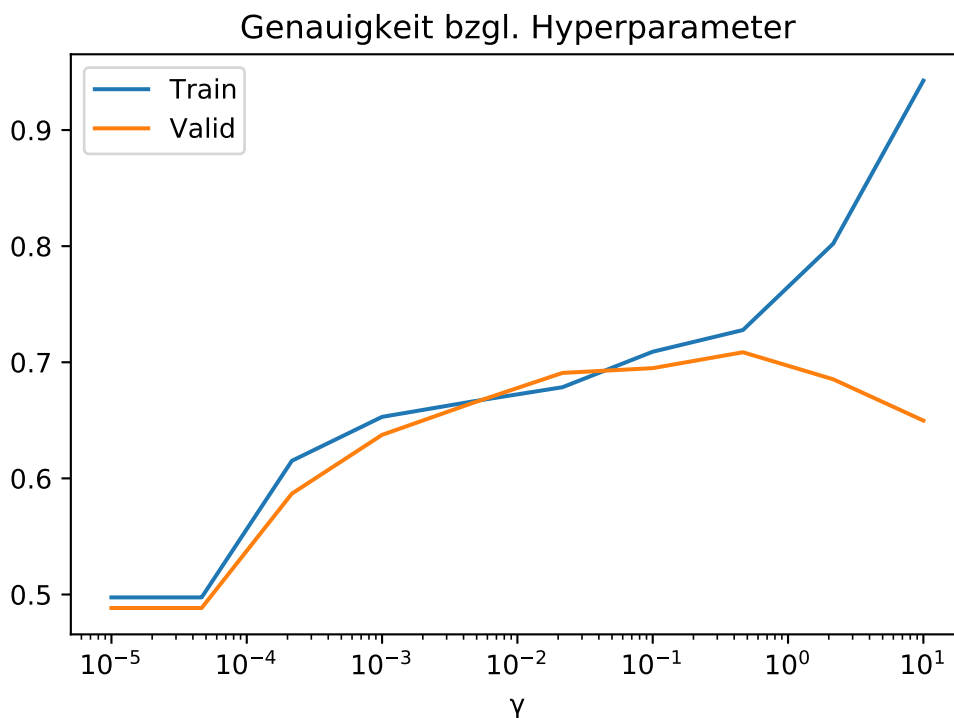


Abb. 15.8: Genauigkeit auf Trainings- und Validierungsdaten in Abhängigkeit von gamma.

- Test (zwei Features): 54%

Unsere Entscheidungsgrenze für den zweidimensionalen Datensatz sieht nun so aus (die optimalen Werte wurden allerdings für das Gesamtmodell ermittelt):

Notebook

Lineare und Kernel-*SVM* mit verschiedenen Hyperparametern: `abalone_svm.ipynb`.

15.6 Entscheidung zwischen Linearer oder Kernel-SVM

Generell ist die Verwendung eines nichtlinearen Kernels aufwendiger als ein lineares Modell. Während man bei einer Kernel-*SVM* bei einigen Zehntausend Datensätzen mit 10–20 Komponenten (Dimensionen) bei der Modellbildung an die Grenzen der Berechenbarkeit (bzgl. der Laufzeit) stößt, können lineare Modelle problemlos mit Zehntausend Dimensionen und Hunderttausenden Datensätzen berechnet werden. Solche hochdimensionalen Modelle entstehen häufig im Bereich des *NLP*. Jeder Text ist ein Datenpunkt, dessen Komponenten alle in allen Texten auftretenden Wörter sind. Der Wert der Komponente ist dann z. B. die Häufigkeit des Auftretens dieses Wortes. Hier tritt häufig der Effekt auf, dass die Zahl der Dimensionen höher als die Zahl der Datenpunkte ist. In einem solchen Falle ist eine lineare Trennung immer möglich, was aber leicht zum Overfitting führt. Man spricht hier vom *Fluch der hohen Dimensionen*.

Generell für alle Klassifikationsverfahren gilt die Faustregel, dass eine Erhöhung der Trainingsmenge einen größeren Einfluss auf die Genauigkeit hat als das konkret gewählte Verfahren.

Hinweis: Stehen zum Training nur einige tausend bis zehntausend Datenpunkte mit einer relativ geringen Anzahl von Dimensionen zur Verfügung, erhält man meist mit einer Kernel-*SVM* bessere Ergebnisse. Bei einer hohen Dimensionszahl und sehr vielen Datenpunkten bietet sich dagegen aus Effizienzgründen die Verwendung der linearen

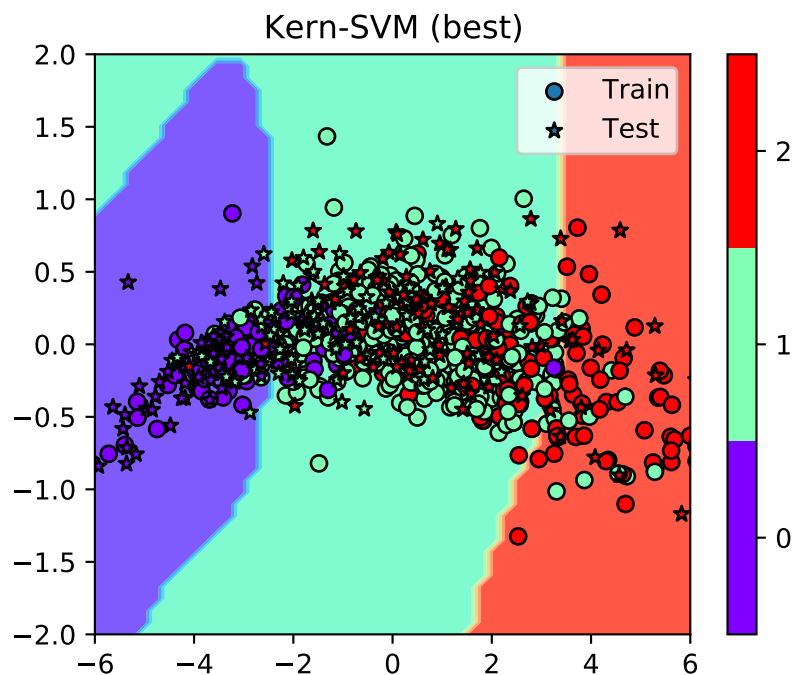


Abb. 15.9: Entscheidungsgrenzen für beste Parameter.

SVM an.

15.7 Regression

Für unsere Klassifikation haben wir die Hyperebene so gewählt, dass der Abstand der Datenpunkte von ihr möglichst groß ist. Bei der Regression legen wir diese nun so, dass die Abstände möglichst klein werden. Da die Hyperebene stets die Vorhersagewerte repräsentiert, ist unser Modellfehler um so kleiner, je näher die tatsächlichen Daten an der Ebene liegen.

```
estimator = svm.LinearSVR(...)
estimator.fit(Xtrain, ytrain)

# Vorhersage
yPredict = estimator.predict(Xpredict)
```

Die möglichen Parameter sind weitestgehend die gleichen wie bei der Klassifikation, der wichtigste ist der Regularisierungsparameter C für den *soft margin*.

Hinweis: Verwenden wir den Parameter `dual=False`, müssen wir ebenfalls den Parameter `loss='squared_epsilon_insensitive'` setzen, da der Defaultwert zu Fehlern führt.

Beispiel: Vorhersage der Ringzahl für den *Abalone*-Datensatz. Die numerischen Komponenten sind wieder mit dem `StandardScaler` umskaliert, die kategorialen Daten in eine *One-Hot*-Codierung überführt. Wir lösen aufgrund der besseren Konvergenz das *primale Problem*.

```
classifier = svm.LinearSVR(dual=False, loss='squared_epsilon_insensitive')
classifier.fit(Xtrain, yTrain)
```

Mit der Methode `score()` erhalten wir wieder ein Maß für die Genauigkeit der Vorhersage. Dieses beruht auf dem mittleren quadratischen Fehler zwischen realen Daten und der Vorhersage und wird so skaliert, dass das Maximum 1 für vollständige Übereinstimmung erreicht wird. Hier sind auch negative Score-Werte möglich.

```
scoreTrain = classifier.score(Xtrain, yTrain)
scoreTest = classifier.score(Xtest, yTest)
```

Wir erhalten für unser Modell folgende Genauigkeiten:

- Training: 0.52
- Test: 0.54

Eine optische Bewertung ist durch einen Scatterplot zwischen wahren Daten und der Vorhersage möglich. Je genauer die Vorhersage, desto dichter liegen die Punkte an der Diagonale $y = x$.

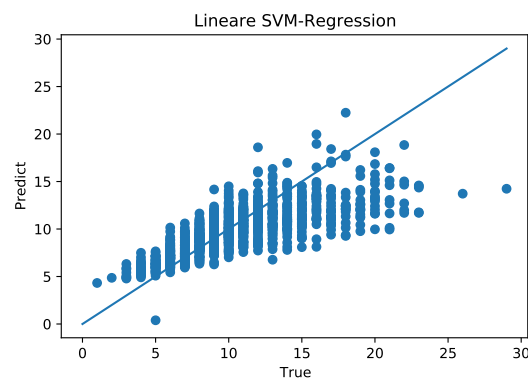


Abb. 15.10: Scatterplot lineare SVM tatsächliche Daten im Vergleich zur Vorhersage.

Suchen wir im Intervall $[10^{-4}, 10^4]$ nach einem optimalen Wert für C mit Hilfe der Methode `score()`, erhalten wir folgenden Verlauf:

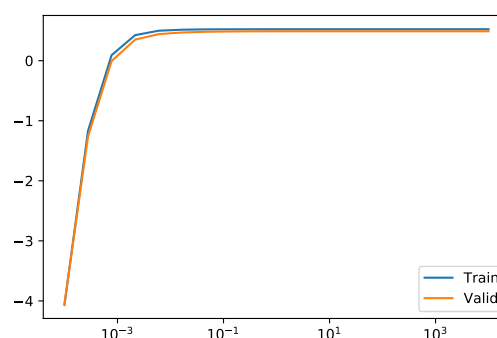


Abb. 15.11: Genauigkeit der linearen SVM (Regression) in Abhängigkeit von C .

Wir sehen für kleine Werte von C (starke Regularisierung = viele Ausreißer erlaubt) ein Underfitting, rechts dagegen kaum ein Overfitting, was sich durch die Beschränkung des Modells auf lineare Zusammenhänge automatisch ergibt.

Beschränken wir uns auf einen Datensatz mit 2 Komponenten, können wir mit einem Kontur- oder einem 3D-Plot die Vorhersagefunktion unserer *SVM* darstellen.

Auch bei Regressionsproblemen können wir wieder den *Kernel-Trick* anwenden, d. h. wir können unsere Komponenten nichtlinear in einen hochdimensionalen Raum transformieren und hier beste Hyperebene suchen.

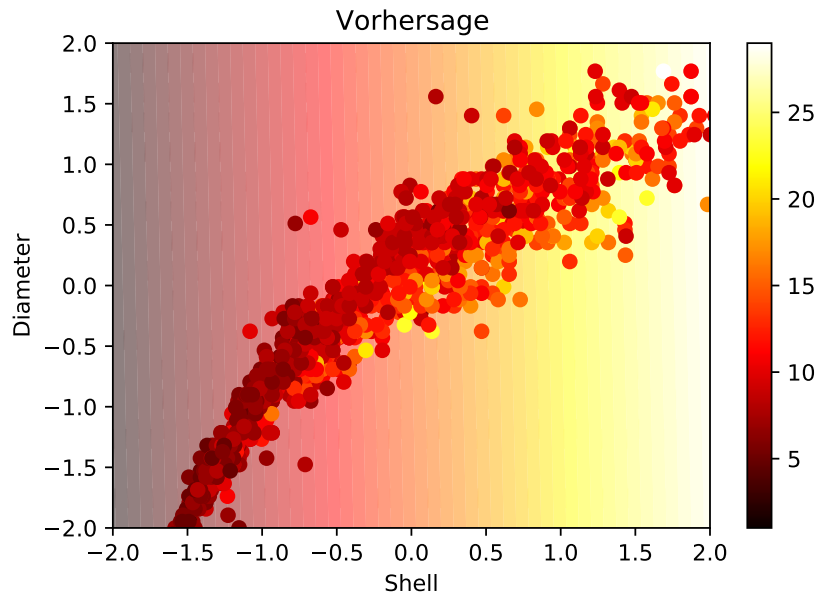


Abb. 15.12: Vorhersagefunktion der linearen SVM für 2 Komponenten sowie die Darstellung der wahren Werte für die Testdaten.

```
estimator = svm.SVR(...)
estimator.fit(Xtrain, ytrain)
# Vorhersage
yPredict = estimator.predict(Xpredict)
```

Mit dem *BBF*-Kernel und den Parametern $\gamma=0.01$ und $C=50$ erhalten wir eine Güte von

- Training: 0.56
- Test: 0.51

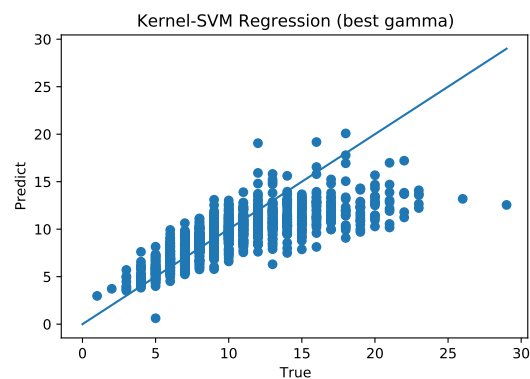


Abb. 15.13: Scatterplot für Kernel-SVM. Wir erkennen optisch kaum einen Unterschied zur linearen SVM.

Für einen Datensatz mit 2 Komponenten können wir wiederum die Vorhersage darstellen ($C=1000$, $\gamma=0.4$). Wir erkennen deutlich mehrere Gipfel und Senken.

Suchen wir hier nach optimalen Parametern (im Beispiel Suche nach γ im Intervall $[10^{-5}, 10^5]$ mit fixiertem $C=1000$), erkennen wir wieder für kleine Werte für γ ein Underfitting (zu einfaches Modell), aber rechts ein Overfitting (zu komplexes Modell).

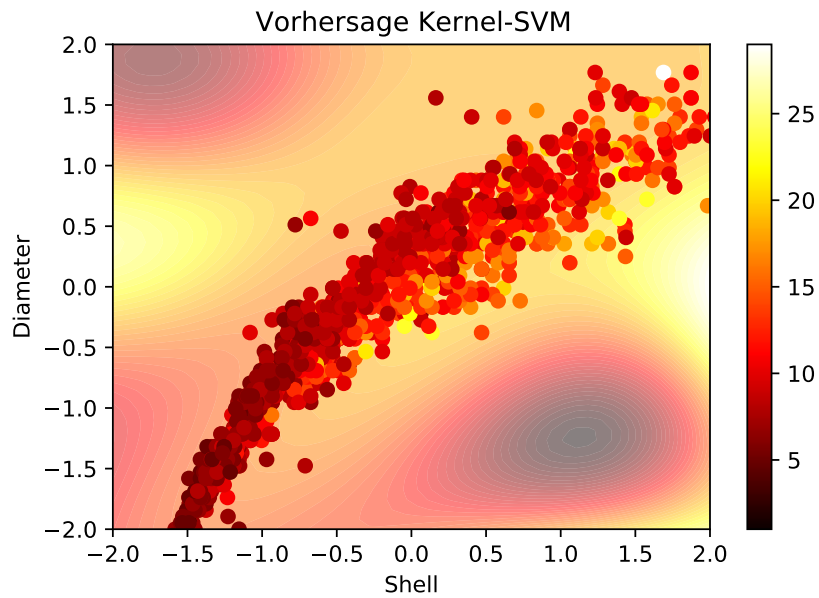


Abb. 15.14: Vorhersagefunktion der Kernel-SVM für Datensatz mit 2 Komponenten.

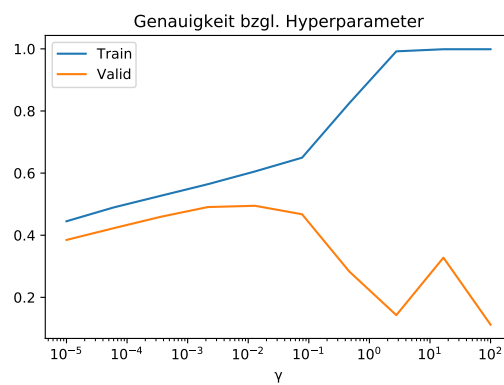


Abb. 15.15: Genauigkeit in Abhängigkeit von gamma (C fest)

Notebook

Kernel-SVM für Regression auf *Abalone*-Datensatz: `abalone_svm_regression.ipynb`.

15.8 Aufgaben

Aufgabe 15.2 Klassifizieren Sie Ihren ausgewählten Datensatz sowohl mit einer linearen als auch mit einer Kernel-*SVM*. Testen Sie verschiedene Parameter und vergleichen Sie die Qualität.

Suche optimaler Hyperparameter

- *Gittersuche*
- *Cross Validation*
- *Hyperparametersuche in scikit-learn*
- *Random Search*
- *Alternative Suchstrategien*
- *Aufgaben*

Viele unserer Machine-Learning-Modelle werden durch Hyperparameter konfiguriert. So können wir beim *KNN*-Verfahren die Anzahl der zu berücksichtigenden Nachbarn oder bei der linearen oder logistischen Regression den Dämpfungsfaktor der Koeffizienten einstellen. Beim Entscheidungsbaum lassen sich u. a. Baumtiefe, die Mindestanzahl von Daten pro Blatt oder für einen Split festlegen, bei Kernel-*SVM*-Modellen müssen wir den *Soft-Margin*- und den oder die *Kernel*-Parameter geeignet wählen. Die Wahl der Parameter kann nicht unabhängig erfolgen, da sie sich gegenseitig beeinflussen.

Wir benötigen also Verfahren, um systematisch in mehreren Dimensionen nach geeigneten Parametern zu suchen.

16.1 Gittersuche

Zum systematischen Suche nach Hyperparametern verwendet man häufig die Gittersuche. Wir betrachten dazu ein Kernel-*SVM*-Modell mit Gauß-Kernel. Hier müssen wir geeignete Werte für die Parameter C und γ suchen, die in einem großen Wertebereich liegen können, C von etwa 0.1 (viele Randverletzungen erlaubt) bis 10000 (harter Rand) und γ von 10^{-5} (Einfluss eines Supportvektors reicht sehr weit) bis 1 oder 10 (Einfluss eines einzelnen Supportvektors ist sehr klein).

Für die gezielte Suche denken wir uns ein 2-dimensionales Raster für die Wertpaare und teilen die Achsen logarithmisch ein.

Für eine Kernel-SVM für unseren *Abalone*-Datensatz könnten wir dabei wie folgt vorgehen:

- Wir trennen von unserer Trainingsmenge eine Validierungsmenge ab. Zur Beschleunigung des Trainings wählen wir diese relativ groß (und damit die verbleibende Trainingsmenge klein).

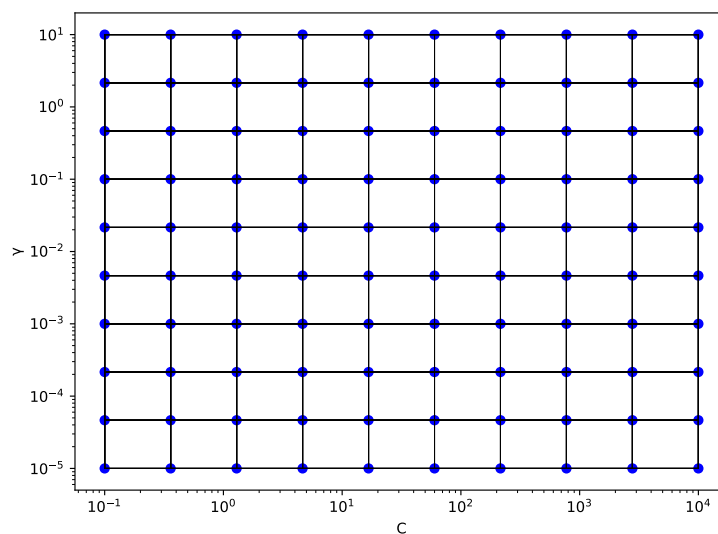


Abb. 16.1: Raster für Gittersuche

```
XtrainHyp, Xvalid, yTrainHyp, yValid = model_selection.train_test_split(
    Xtrain, yTrain, test_size=0.6, random_state=23
)
```

- Erzeugen aller zu durchsuchenden Wertpaare und Berechnung der Güte für jedes Paar auf der Validierungsmenge. Speichern aller Werte für die Auswertung, Auswahl des besten Parameterpaars.

```
paramC = np.logspace(-1, 4, 10)
paramGamma = np.logspace(-5, 1, 10)
bestScore = 0.0
bestC = None
bestGamma = None
scores = list()
for C in paramC:
    for gamma in paramGamma:
        cl = svm.SVC(C=C, gamma=gamma)
        cl.fit(XtrainHyp, yTrainHyp)
        scoreTrain = cl.score(XtrainHyp, yTrainHyp)
        scoreValid = cl.score(Xvalid, yValid)
        if scoreValid > bestScore:
            bestScore = scoreValid
            bestC = C
            bestGamma = gamma
        scores.append((C, gamma, scoreTrain, scoreValid,))
scores = np.array(scores)
```

Wir erhalten hier z. B. die Werte $C=774$ und $\gamma=0.02$ und eine Genauigkeit von 1.0.

Die einzelnen erreichten Genauigkeiten können wir einfach in einem Scatterplot darstellen:

```
plt.scatter(scores[:,0], scores[:,1], c=scores[:,3]) # Validierungsdaten
for row in scores:
    plt.annotate('{:.0f}'.format(row[3]*100), # Text
                row[0:2], # Datenposition als Paar
                xytext=(5, 5), # Textposition als Paar
                textcoords='offset points' # Positionsangabe als Offset)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
)
plt.xscale('log')
plt.yscale('log')
```

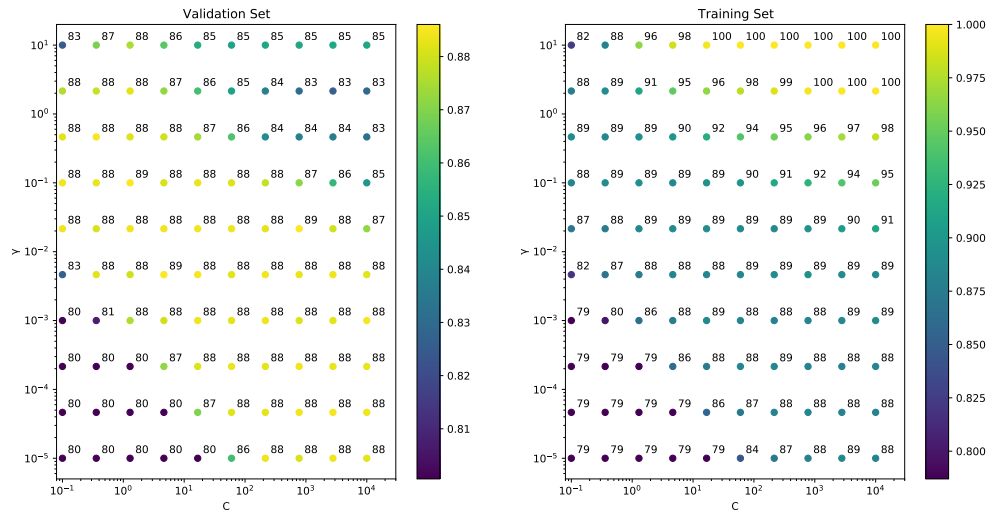


Abb. 16.2: Naive Gittersuche, Genauigkeit für Validierungs- und Trainingsdaten.

Ein vielleicht übersichtlicheres Bild bietet uns ein Konturplot. Unsere Matrix `scores` enthält der Reihe nach für jeden Wert von `C` alle Werte von `gamma`. Wir müssen dazu die Spalten für die Parameter und die Genauigkeiten jeweils in eine Matrix mit $|C|$ Zeilen und $|\gamma|$ Spalten.

```
lenC = len(paramC)
lenGamma = len(paramGamma)
matC = scores[:, 0].reshape(lenC, lenGamma)
matGamma = scores[:, 1].reshape(lenC, lenGamma)
matTrain = scores[:, 2].reshape(lenC, lenGamma)
matValid = scores[:, 3].reshape(lenC, lenGamma)
plt.contourf(matC, matGamma, matValid)
plt.xscale('log')
plt.yscale('log')
```

Wir erhalten nun folgendes Bild:

Vergleichen wir das linke und rechte Bild, sehen wir links unten (kleine Werte für `C` und `gamma` = starke Generalisierung) eine schlechte Güte sowohl für Trainings- als auch Validierungsdaten, also ein Zeichen von Underfitting. Rechts oben dagegen (große Werte für `C` und `gamma`) sehen wir eine hohe Güte auf den Trainings- aber eine schlechte Güte auf den Validierungsdaten, also ein Zeichen für Overfitting.

Halten wir nun aber den Parameter `C` fest und bestimmen die Genauigkeit sowohl auf den Trainings- wie auf den Testdaten in Abhängigkeit von `gamma`.

```
Cmean = paramC[lenC//2] # mittlerer Wert unserer Parameterliste
scoresFixed = scores[scores[:,0]==Cmean,]
plt.plot(scoresFixed[:,1], scoresFixed[:,2], label='Train')
plt.plot(scoresFixed[:,1], scoresFixed[:,3], label='Valid')
plt.xscale('log')
plt.legend()
```

Hier sehen wir mit steigendem `gamma` zunächst eine Verbesserung sowohl auf den Trainings- als auch auf den Testdaten, dann verschlechtert sich das Ergebnis auf den Testdaten allerdings wieder – wir haben hier also ein *Overfitting*.

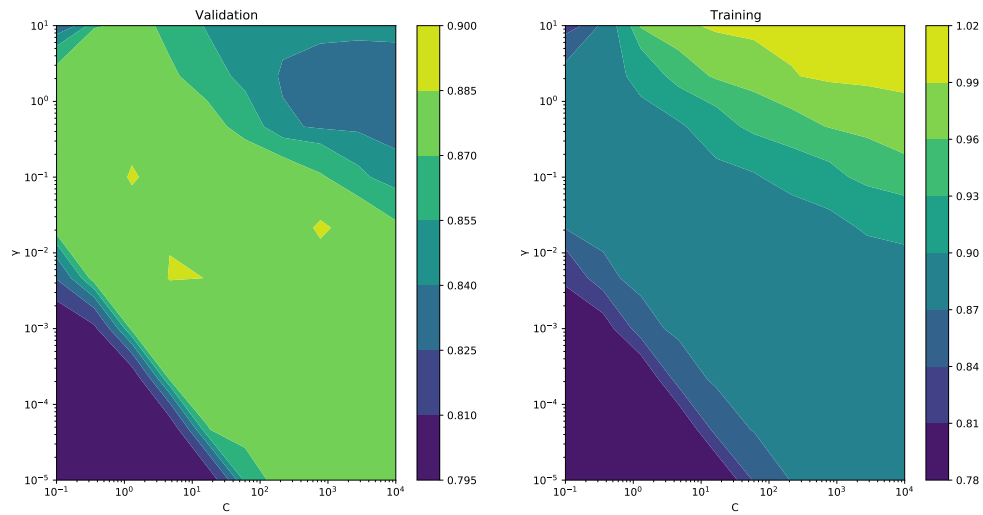
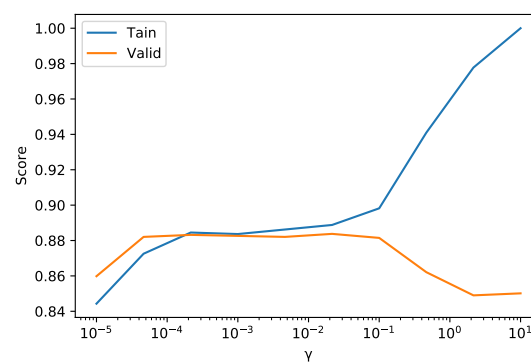


Abb. 16.3: Konturplot der Güte für verschiedene Hyperparameter.



16.2 Cross Validation

Unsere Gittersuche benutzt bisher eine feste, abgetrennte Validierungsmenge zum Bestimmen der Hyperparameter. Dies hat jedoch zum einen den Nachteil, dass sich unsere Trainingsmenge verkleinert und zum anderen unsere gefundenen Hyperparameter stark von der konkret gewählten Validierungsmenge abhängen und damit für unseren Gesamtdatensatz weniger gut angepasst sind. Um diesen Effekt zu vermeiden, verwendet man das Verfahren der *Kreuzvalidierung* (*Cross Validation*).

Die *Trainingsdaten* werden dazu in k gleichgroße Teilmengen aufgeteilt. Nun wird jeweils eine dieser Teilmengen als *Validierungsmenge* zurückgehalten und das Modell mit den $k-1$ restlichen Teilmengen trainiert.

Anschließend wird die Genauigkeit dieses Modells mit der Validierungsmenge geprüft. Für jeden Parametersatz wird dieses Training k -mal durchgeführt, wobei jede der Teilmengen einmal die Validierungsmenge ist.

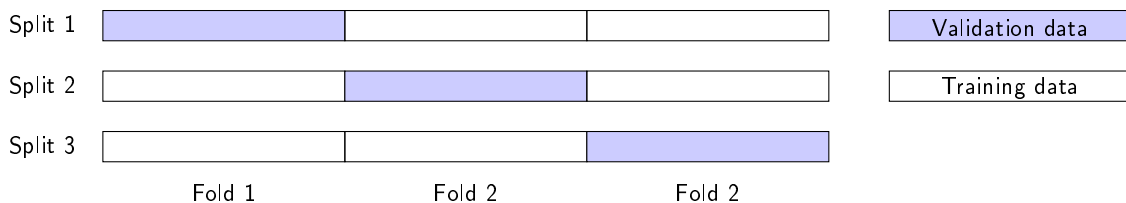


Abb. 16.4: Aufteilung der Daten für Cross Validation

Hinweis: Wenn wir also für eine Kernel-*SVM* g verschiedene Werte für γ , c verschiedene Werte für C bei k -facher Kreuzvalidierung durchmustern wollen, müssen wir insgesamt $k \cdot g \cdot c$ Modelle berechnen und evaluieren. Wir müssen für größere Datensätze hier einen beträchtlichen Zeitaufwand planen.

Für unser endgültiges Modell wählen wir dann ein Modell mit dem besten Parametersatz der Kreuzvalidierung, verwenden aber nun alle Trainingsdaten zur Modellerstellung.

Für die Zahl der Teilmengen für die Kreuzvalidierung verwendet man häufig 3 oder 5 (33% bzw. 20% der Daten als Validierungsmenge). Je größer man diese Zahl wählt, um so länger dauert das Training, da ja für jede Kombination von Hyperparametern k Modelle erstellt und bewertet werden müssen.

16.3 Hyperparametersuche in scikit-learn

Unsere Bibliothek stellt uns mit der Klasse `GridSearchCV` eine bequeme Möglichkeit zur Gittersuche mit Kreuzvalidierung zur Verfügung. Wir müssen zunächst unser abzusuchendes Parametergitter als Python-Dictionary zur Verfügung stellen. Die verwendeten Schlüssel sind die Parameternamen der jeweiligen Lernmodellklasse, die Werte sind *NumPy*-Vektoren.

```
paramGrid = {
    'parameter1': np.linspace(start, stop, anzahl),
    'parameter2': np.logspace(start, stop, anzahl, base=basis),
    'parameter3': np.arange(start, stop),
    'parameter4': [wert1, wert2, ...],
    ...
}
```

Die Varianten `parameter1` und `parameter2` sind für stetige Hyperparameter geeignet, die jeden reellen Wert annehmen können, die Variante `parameter3` für ganzzahlige Werte und `parameter4` generell für diskrete Werte (Liste von Zahlen oder anderen Objekten wie Strings oder Dictionaries).

Beispiel für Random-Forest-Modelle:

```
paramGrid = {
    'n_estimators': np.arange(10, 100, 10),
    'max_depth': np.array([10, 15, 20]),
    'min_samples_leaf': np.arange(1, 10),
}
```

Beispiel für Kernel-*SVM* mit Gauß-Kernel:

```
paramGrid = {
    'C': np.logspace(-1, 4, 10),      # 0.1 bis 10000
    'gamma': np.logspace(-5, 1, 30),  # 0.00001 bis 10
    'class_weight': [{0: 1, 1: 1},
                     {0: 2, 1: 1},
                     {0: 1, 1: 2}],   # Liste von 3 Dictionaries (2 Klassen)
}
```

Beispiel für lineare *SVM* mit unterschiedlicher Regularisierung:

```
paramGrid = {
    'C': np.logspace(-1, 4, 10),      # 0.1 bis 10000
    'penalty': ['l1', 'l2'],          # Liste
}
```

Dieses Gitter übergeben wir zusammen mit der Modellklasse (die wir ohne die zu suchenden Hyperparameter initialisieren) der Klasse `GridSearchCV`.

```
from sklearn.model_selection import GridSearchCV

classifier = MyClassifier()
gridSearch = GridSearchCV(
    [estimator=]classifier,
    param_grid=paramGrid,
    cv=folds
    ...
)
```

Die Aufteilung der Folds erfolgt einmalig für das gesamte Training in der Form, dass jede Teilmenge den gleichen Prozentsatz von Daten jeder Klasse enthält (*Stratified K-Folds*).

Dieses Gittersuchobjekt verhält sich nun wie jede andere Klassifikation, wird also mit dem Aufruf `fit()` trainiert.

```
gridSearch.fit(Xtrain, ytrain)
```

Wir können dieses Objekt direkt für Vorhersagen oder zur Bewertung der Güte auf den Trainingsdaten verwenden.

```
score = gridSearch.score(X, y) # Trainings- oder Testdaten
yPredict = gridSearch.predict(X)
```

Verwendet wird dazu das beste erlernte Modell, auf das wir über die Komponente

```
bestesModel = gridSearch.best_estimator_
```

zugreifen können. Ebenso können wir die Parameter für dieses Modell auslesen:

```
besteParameter = gridSearch.best_params_
```

Das Ergebnis ist ein Dictionary mit den Parameternamen als Schlüssel und den jeweiligen besten Parameterwerten.

Hinweis: Wir können die Gittersuche beschleunigen, indem wir zusätzlich den Parameter `n_jobs` spezifizieren. Er gibt an, wieviele parallele Prozesse zur Berechnung genutzt werden. Ein Wert von -1 nutzt die maximal verfügbare Zahl von Prozessorkernen.

Beispiel: Gittersuche für einen *Random-Forest*-Classifier für unseren *Abalone*-Datensatz. Wir suchen dabei nach optimalen Parametern für die minimale Zahl von Datenpunkten, bei der ein Knoten geteilt wird, und die Zahl der Bäume.

```
classifier = ensemble.RandomForestClassifier()
clSearch = model_selection.GridSearchCV(
    classifier, param_grid=paramGrid,
    cv=3,
    n_jobs=-1,
    verbose=1,
)
clSearch.fit(Xtrain, yTrain)
print('Best parameters:', clSearch.best_params_)
print('Best score:', clSearch.best_score_)
```

Wir erhalten im Beispiel als beste Parameter

```
{'min_samples_split': 28, 'n_estimators': 50}
```

Wollen wir die Suche genauer analysieren, können wir uns die Komponente `cv_results_` anschauen. Sie enthält ein Dictionary, welches u. a. die Liste der Parameterkombinationen und der dazugehörigen Genauigkeit enthält. Hier ist dem Parameternamen jeweils der Präfix `param_` vorangestellt. Für schnelle Analysen können wir dieses Dictionary einfach in einen *Pandas* Data Frame umwandeln:

```
dfResults = pd.DataFrame(clSearch.cv_results_)
dfResult.head()
```

Wichtige Spalten sind:

param_parameter Wert des Parameters `parameter` für dieses Modell.

mean_test_score Mittlere Genauigkeit dieses Modells für alle Folds.

std_test_score Standardabweichung der Genauigkeiten dieses Modells für alle Folds.

Einen einfachen Scatterplot erhalten wir, indem wir zwei Parameterspalten als Achsen auswählen und die Genauigkeitsspalte als Farbe (und Größe) codieren.

Beispiel für die *Random-Forest*-Gittersuche.

```
plt.scatter(
    clSearch.cv_results_['param_min_samples_split'],
    clSearch.cv_results_['param_n_estimators'],
    c=clSearch.cv_results_['mean_test_score'],
)
```

Wollen wir lieber einen Kontur-Plot erzeugen, müssen wir die Parameterwerte und die Genauigkeiten in einem zwei-dimensionalen Raster anordnen. Wir benötigen dazu die Reihenfolge der Gruppierung, die wir aus der Reihenfolge der Komponente `best_params_` ablesen können, sowie die Anzahl der Werte jedes Parameters. Für unsere Hyperparameter des *Random Forest* erhalten wir die Reihenfolge `min_samples_split` `n_estimators`. Die Anzahl der Werte lesen wir aus dem Parametergitter aus:

```
lSplit = len(paramGrid['min_samples_split'])
lTrees = len(paramGrid['n_estimators'])
```

Nun können wir alle drei Spalten im Raster anordnen und den Kontur-Plot erzeugen.

```
xx = clSearch.cv_results_['param_min_samples_split'].reshape(lSplit, lTrees)
yy = clSearch.cv_results_['param_n_estimators'].reshape(lSplit, lTrees)
zz = clSearch.cv_results_['mean_test_score'].reshape(lSplit, lTrees)
plt.contourf(xx, yy, zz)
```

Wir erhalten folgendes Bild:

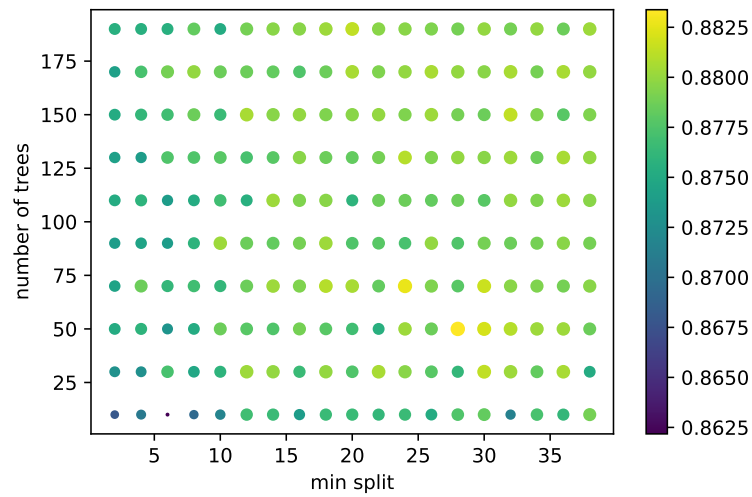


Abb. 16.5: Raster der Hyperparameter und Modellgüte.

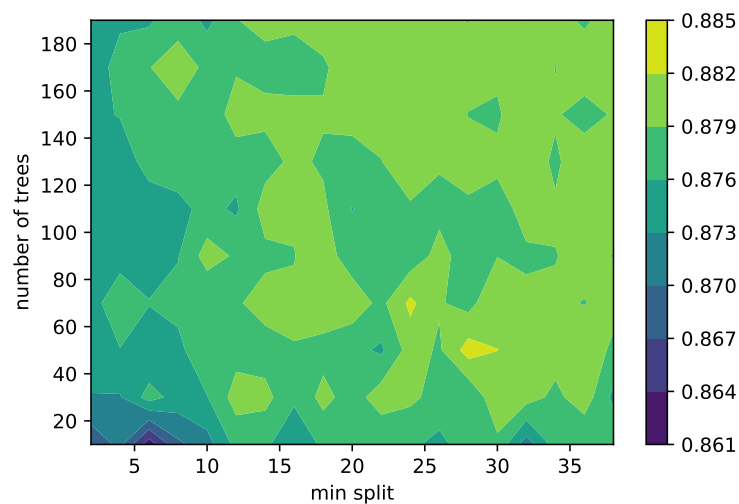


Abb. 16.6: Genauigkeit in Abhängigkeit der Hyperparameter.

Warnung: Die Reihenfolge der Dimensionen muss vor dem Reshape geprüft werden, sonst erhält man unsinnige Bilder.

Für eine Kernel-*SVM* müssen wir die Hyperparameter C und γ bestimmen. Hier müssen mehrere Zehnerpotenzen abgesucht werden (für γ etwa Werte zwischen 10^{-5} bis 1 und für C Werte zwischen 0.1 und 10^4). Um diesen großen Bereich abzudecken, verwenden wir ein gleichmäßiges Raster auf den logarithmierten Daten:

```
paramGrid = {
    'C': np.logspace(-1, 4, 10),
    'gamma': np.logspace(-4, 0, 8),
}
```

Das weitere Vorgehen ist analog:

```
clSearch = model_selection.GridSearchCV(
    svm.SVC(),
    param_grid=paramGrid,
    cv=3,
    n_jobs=-1,
    verbose=1,
)
clSearch.fit(Xtrain, yTrain)
```

Wir erhalten als beste Parameter $C=60$ und $\gamma=0.07$ und damit eine Genauigkeit von 0.91 auf den Trainings- und 0.83 auf den Testdaten.

Wollen wir die Daten wieder graphisch als Scatter- oder Kontur-Plot darstellen, müssen wir die logarithmische Einteilung des Gitters berücksichtigen:

```
plt.xscale('log')
plt.yscale('log')
```

Wir erhalten folgende Genauigkeiten auf dem Gitter:

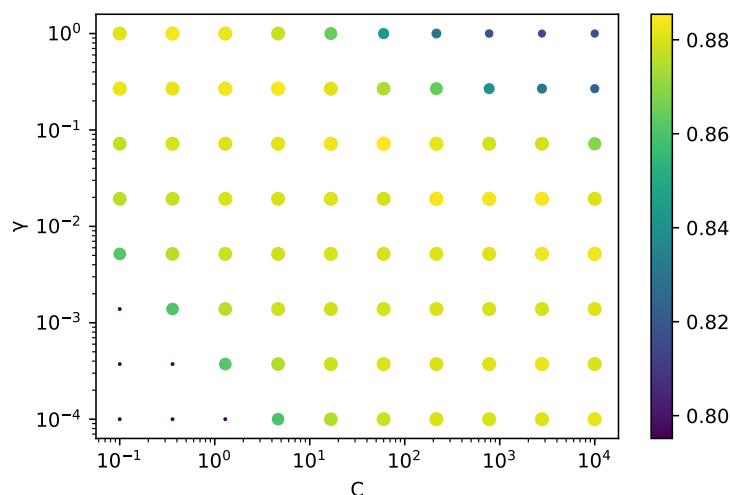


Abb. 16.7: Modelgüte auf dem Parametergitter (SVM).

Wir sehen links unten (kleine Werte für C und γ) eine zu starke Regularisierung (Underfitting) und rechts oben (große Werte für C und γ) eine zu starke Spezialisierung und damit ein Overfitting.

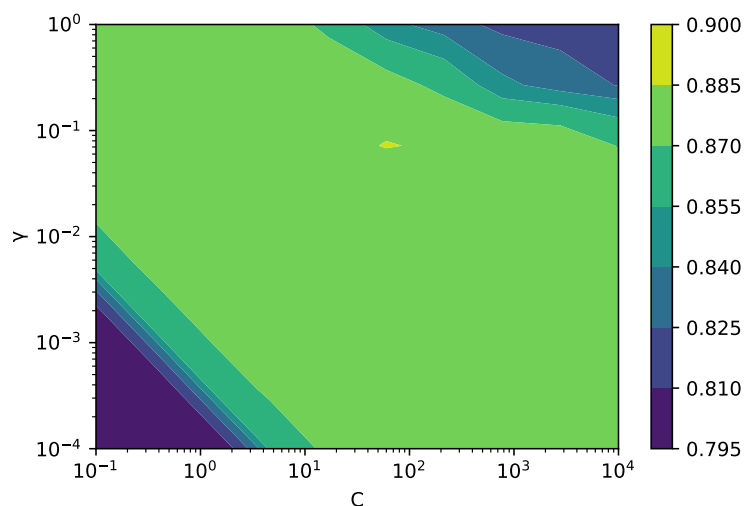


Abb. 16.8: Parametergüte als Kontur-Plot.

Notebook

Gittersuche für *Abalone* für *Random Forest* und *SVM* mit graphischer Auswertung: `abalone-hyper-gridsearch.ipynb`.

16.4 Random Search

Da je nach Modell das Durchmustern eines Gitters besonders bei vielen Modellparametern sehr lange dauern kann, können auch alternative Suchstrategien eingesetzt werden. Eine Möglichkeit ist die randomisierte Suche. Wir wählen dazu aus unserem Suchraum zufällige Kombinationen der Hyperparameter aus und trainieren und bewerten für diese unser Modell. Damit wird im allgemeinen ein größerer Bereich des Parameterraums als bei der Gittersuche in kürzerer Zeit abgesucht, da wir beim systematischen Durchmustern zuerst die Ränder betrachten, an denen selten die optimalen Parameter zu finden sind.

Voraussetzung für diese Vorgehensweise ist die Angabe einer Wahrscheinlichkeitsverteilung für jeden Parameter. Hierzu benötigen wir einen Zufallsgenerator, der die Generator `rvs()` bereitstellt. Passende Generatoren finden wir im Modul `stats` der Bibliothek *SciPy* ([SciPy.org](https://www.scipy.org)), die von *scikit-learn* ebenfalls benötigt wird und damit automatisch installiert ist.

```
import scipy
```

Wichtige Verteilungen sind:

`scipy.stats.uniform(a, b)` Gleichverteilte stetige Zufallszahlen im Intervall $[a, b]$.

`scipy.stats.loguniform(a, b)` Logarithmisch verteilte stetige Zufallszahlen im Intervall $[a, b]$. Für `scipy.stats.loguniform(1e-2, 1e2)` werden also die Werte 0.01, 0.1, 1, 10 und 100 mit gleicher Wahrscheinlichkeit ausgewählt.

`scipy.stats.randint(a, b)` Gleichverteilte ganzzahlige Zufallszahlen im Intervall $[a, b)$ (also inklusive a und exklusive b).

Die Wahrscheinlichkeitsverteilungen für jeden Parameter werden wieder als Dictionary spezifiziert:

```
paramDistribution = {
    'parameterName': verteilung,
    'parameterName': wertListe,
    ...
}
```

```
}
```

Für die Suche der von Hyperparametern für unser *Random-Forest*-Modell könnten wir folgende Verteilungen verwenden:

```
distr = {
    'n_estimators': scipy.stats.randint(2, 200),
    'min_samples_split': scipy.stats.randint(2, 50),
}
```

Für die zufällige Suche verwenden wir ein Objekt der Klasse `RandomizedSearchCV` des Moduls `model_selection`, das im wesentlichen die gleichen Parameter wie die Gittersuche erwartet, nur das Gitter wird durch die Verteilungen ersetzt. Hinzu kommt ein Parameter `n_iter`, der angibt, wieviele Parameterkombinationen untersucht werden sollen.

```
from sklearn.model_selection import RandomizedSearchCV

classifier = MyClassifier()
clSearch = RandomizedSearchCV(
    [estimator=]classifier,
    param_distribution=paramDistribution,
    cv=folds,
    n_iter=versuchsZahl,
    ...
)
clSearch.fit(Xtrain, yTrain)
```

Beispiel: Hyperparametersuche für *Random-Forest*-Modell für *Abalone*.

```
classifier = ensemble.RandomForestClassifier()
clSearch = model_selection.RandomizedSearchCV(
    classifier, param_distributions=distr,
    cv=3,
    n_iter=40,
    n_jobs=-1,
    random_state=23,
    verbose=1,
)
clSearch.fit(Xtrain, yTrain)
```

Wir erhalten zum beispieelsweise als beste Parameter `min_samples_split=43`, `n_estimators=81` und folgende Genauigkeiten:

- Training: 0.90
- Test: 0.84

Eine graphische Anzeige mittels Scatterplot ist hier ebenfalls möglich.

Einen Kontur-Plot können wir allerdings nicht erzeugen, da wir dafür die Werte auf einem regelmäßigen Gitter benötigen.

Für eine Kernel-*SVM* benötigen wir für `C` und `gamma` eine gleichmäßige Verteilung über die logarithmierten Werte:

```
distr = {
    'C': scipy.stats.loguniform(0.1, 1e4),
    'gamma': scipy.stats.loguniform(1e-5, 1),
}
```

Wir erhalten mit 40 Versuchen z. B. die Hyperparamter `C=88` und `gamma=0.07` und damit die Genauigkeiten

- Training: 0.89,
- Test: 0.84,

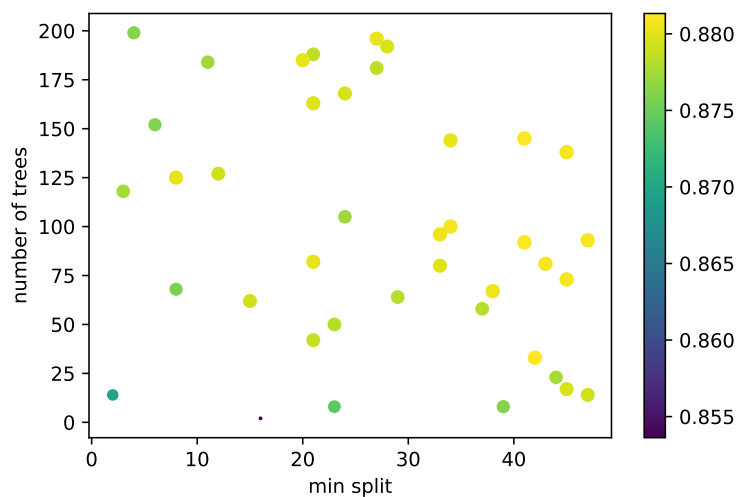


Abb. 16.9: Zufällige Parametersuche Random Forest.

was nahe an den Werten der Gittersuche liegt aber mit deutlich weniger Suchaufwand gewonnen wurde.

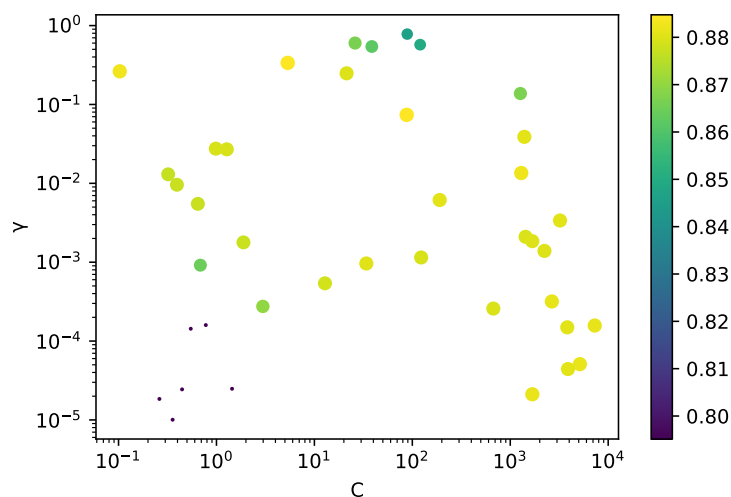


Abb. 16.10: Zufällige Parametersuche SVM.

Insbesondere bei größerer Zahl von Hyperparametern ist die Zufallssuche vorzuziehen.

Beispiel: Wir suchen zusätzlich noch den optimalen Kernel für unsere [SVM](#). Dieser Parameter wird als String angegeben, wir geben in unserem Dictionary für die Wahrscheinlichkeitsverteilungen einfach eine Liste aller möglichen Werte an:

```
distr = {
    'C': scipy.stats.loguniform(0.1, 1e4),
    'gamma': scipy.stats.loguniform(1e-5, 1),
    'kernel': ['rbf', 'poly', 'sigmoid']
}
```

Wir erhalten als optimale Hyperparameter z. B. `kernel=rbf`, `C=5.4`, `gamma=0.22` und die Genauigkeiten

- Training: 0.89

- Test: 0.84

Da wir nun 3 Hyperparameter darstellen möchten, benutzen wir verschiedene Symbole für den jeweils gewählten Kernel.

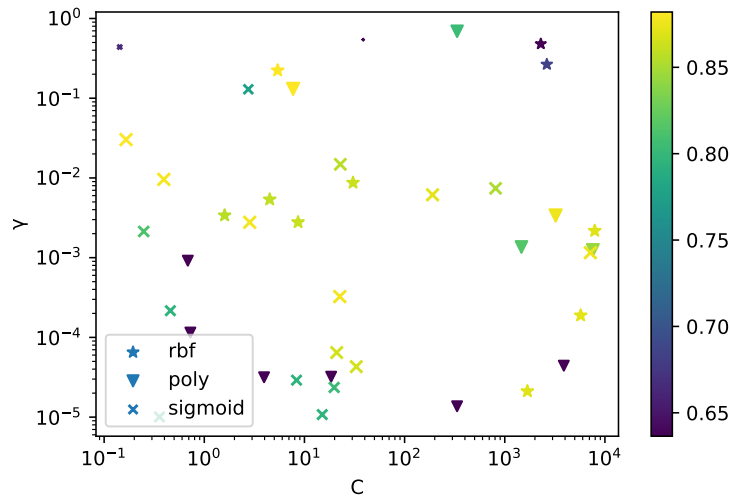


Abb. 16.11: Zufällige Parametersuche SVM mit 3 Hyperparametern.

Weitere Informationen enthält das Paper *Bergstra, J. and Bengio, Y., Random search for hyper-parameter optimization* ([Link](#)).

Notebook

Zufallsuche für *Abalone* für *Random Forest* und *SVM* mit graphischer Auswertung: `abalone-hyper-random.ipynb`.

16.5 Alternative Suchstrategien

Neben diesen beiden besprochenen existieren weitere Strategien, um die Suche nach guten Hyperparametern zu beschleunigen. Die beiden Klassen `HalvingGridSearchCV` und `HalvingRandomizedSearchCV` schränken sukzessiv das Suchintervall auf vielversprechende Parameterwerte ein. Diese sollen hier nicht weiter untersucht werden.

Da je nach Modell das Durchmustern eines Gitters besonders bei vielen Modellparametern sehr lange dauern kann, können auch alternative Suchstrategien eingesetzt werden. Eine Möglichkeit ist die randomisierte Suche. Wir wählen dazu aus unserem Suchraum zufällige Kombinationen der Hyperparameter aus und trainieren und bewerten für diese unser Modell. Damit wird im allgemeinen ein größerer Bereich des Parameterraums in kürzerer Zeit abgesucht. Voraussetzung für diese Vorgehensweise ist die Angabe einer Wahrscheinlichkeitsverteilung für jeden Parameter.

scikit-learn stellt dieses Verfahren mit der Klasse `RandomizedSearchCV` zur Verfügung (s. Dokumentation). Wir wollen hier nicht weiter darauf eingehen.

16.6 Aufgaben

Aufgabe 16.1 Hausaufgabe (10 Punkte)

Verwenden Sie zur Klassifikation Ihres Datensatzes sowohl einen *Entscheidungsbaum* als auch eine Kernel-*SVM* mit *Gauß-Kernel*. Ermitteln Sie für beide Modelle mittels *Cross-Validation* für jeweils zwei Hyperparameter die optimalen Werte und stellen sie die Parametersuche graphisch dar.

Überprüfen Sie die Güte beider Modelle anhand Ihrer Testdaten. Für welches Modell erhalten Sie bessere Ergebnisse? Erhalten Sie bessere Ergebnisse als mit den vorher betrachteten Verfahren?

Untersuchen Sie, ob Sie diese beiden optimierten Modelle und eventuell weitere Modelle zu einem Ensemble kombinieren können, um die Vorhersagekraft weiter zu verbessern.

Hinweis: Falls Sie einen sehr großen Datensatz ausgewählt haben, sollten Sie die Menge der Trainingsdaten auf maximal 10000 Datensätze begrenzen, da sonst die Berechnung des *SVM*-Modells extrem lange dauern kann.

Künstliche Neuronale Netze

- *Überblick*
- *Training des Netzwerkes*
- *Initialisierung*
- *Vorbereitung der Daten*

17.1 Überblick

Die Idee der *ANN* entstand bereits in den 1940er Jahren und orientiert sich an der (vermuteten) Funktionsweise des menschlichen Gehirns. Ein *Neuron* bzw. *Perceptron* verfügt über mehrere Eingänge, von denen es Signale erhält. Diese werden gewichtet und addiert. Wird dabei ein Schwellwert überschritten, *feuert* das Neuron, es sendet ein Ausgangssignal.

Bei einem künstlichen neuronalen Netz betrachten wir jede Komponente unserer Daten als einen Sender von Signalen (Eingabeknoten), der mit einer Anzahl von Neuronen verbunden ist. Deren Ausgänge sind wiederum mit weiteren Neuronen verbunden. Für jede Komponente definieren wir also einen eigenen Eingabeknoten, der diese als reellen Wert bereitstellt.

Der Übersichtlichkeit wegen ordnen wir die Neuronen in einzelnen Schichten (Layers) an. Bei einem *Fully Connected Feed Forward Network* werden alle Knoten einer Schicht mit allen Knoten der Folgeschicht verbunden. Eine Ausgangsschicht enthält so viele Knoten, wie wir Vorhersagewerte erwarten. Bei einem Klassifikationsproblem benötigen wir für jede Klasse einen eigenen Knoten, der den Wert 1 für die korrekte Klasse und sonst den Wert 0 haben sollte. Bei einem Regressionproblem (Vorhersagen einer stetigen Größe) benötigen wir dagegen nur einen Ausgabeknoten.

Alle Schichten außer der Ein- und der Ausgabeschicht heißen *verborgene Schichten* (*hidden layers*). Die Zahl der Schichten ohne die Eingabeschicht, aber inklusive der Ausgabeschicht wird als *Tiefe* des Netzes bezeichnet. Ein *Deep Network* ist also ein neuronales Netz mit einer großen Zahl verborgener Schichten. Die Anzahl der Neuronen in einem Layer ist die *Breite* dieses Layers.

Jedes einzelne Neuron übernimmt die Werte der verbundenen Knoten der vorigen Schicht, multipliziert jeden mit einem individuellen Gewicht und summiert diese auf. Auf diese Summe wird ein weiterer Wert, der *Bias* addiert:

$$z = \sum_{i=1}^{n^{(l-1)}} \left(w_i \cdot y_i^{(l-1)} \right) + b$$

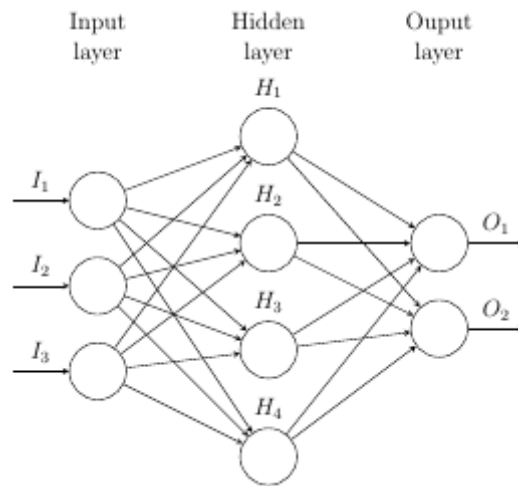
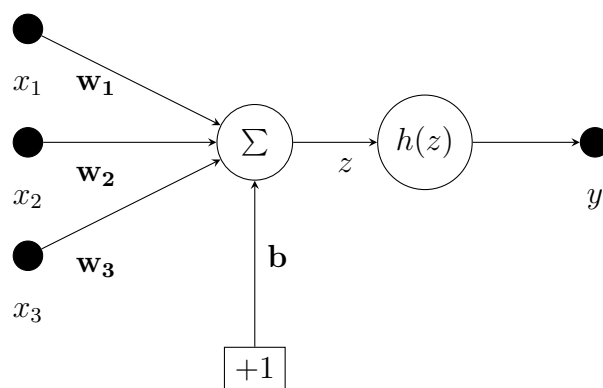
Abb. 17.1: *MLP*

Abb. 17.2: Aufbau eines einzelnen Neurons

Dabei bezeichnen $y_i^{(l-1)}$ die Ausgabewerte der Neuronen der vorigen Schicht, w die Gewichte des Neurons für jeden Vorgängerknoten und b den Bias des Neurons.

Diese Summe wird an eine Aktivierungsfunktion übergeben, die daraus den Ausgabewert produziert. Im klassischen Ansatz arbeitet sie so, dass der Ausgang solange 0 ist, wie ein bestimmter Schwellwert nicht überschritten wird. Beim Überschreiten dieses Wertes schaltet der Ausgang auf 1, das Neuron *feuert*.

$$y^{(l)} = h(z)$$

Bei Netzen für Regressionsmodelle entfällt die Aktivierungsfunktion des (einzelnen) Ausgabeknotens, da wir hier ja nicht nur die Werte 0 und 1, sondern beliebige reelle Werte berechnen wollen.

Für Vorhersagen übergeben wir also die Komponenten eines Datenpunktes an den jeweiligen Knoten der Eingangsschicht und berechnen nun schichtweise für jedes Neuron den Ausgabewert, der nun wiederum als Eingabewert für die Folgeschicht dient. An den Werten der Ausgabeschicht lesen wir das Vorhersageergebnis ab.

17.2 Training des Netzwerkes

Damit unser Netzwerk für einen Datenpunkt das korrekte Ergebnis vorhersagen kann, müssen wir es *trainieren*. Dies geschieht durch Anpassung der *Gewichte*, i. e. der Faktoren w und der Biasse b aller Knoten.

Wir berechnen dazu die Vorhersage unseres Modells für jeden Trainingspunkt und betrachten die Differenz zum erwarteten Ergebnis. Diese beschreiben wir über eine sogenannte *Loss function* (*Kostenfunktion*) $L_{w,b}(y, t)$, die umso kleiner wird, je näher die Vorhersage y mit den tatsächlichen Werten t übereinstimmt.

Diese Funktion ist von unseren Gewichten abhängig, die wir nun so anpassen wollen, um den Fehler zu minimieren. Wir bestimmen dazu partiellen Ableitungen der Funktion nach jedem Gewicht

$$\begin{aligned} dw &= \frac{\partial L}{\partial w} \\ db &= \frac{\partial L}{\partial b} \end{aligned}$$

Eine kleine Änderung der Gewichte in Richtung dw bzw. db würde zu einer Erhöhung der Kostenfunktion führen, wir gehen deshalb einen kleinen Schritt in die Gegenrichtung:

$$\begin{aligned} w_{neu} &= w - \eta dw \\ b_{neu} &= b - \eta db \end{aligned}$$

Der Faktor η heißt dabei *Lernrate*. Diese muss klein genug sein, dass sich die Kostenfunktion durch Anwenden dieser Differenzen tatsächlich verkleinert, zum anderen aber auch so groß, dass tatsächlich eine spürbare Verbesserung des Modells erzielt wird. Dieses Vorgehen nennt man *Gradient-Descent-Verfahren* (*Gradientenabstiegsverfahren*).

Damit wir dieses Verfahren überhaupt anwenden können, muss unsere Kostenfunktion F bezüglich der Gewichte differenzierbar sein. Dazu müssen sowohl L selbst als auch die Aktivierungsfunktionen der Knoten differenzierbar sein und einen Anstieg ungleich Null haben (sonst verschwindet die Ableitung). Durch ursprünglich vorgeschlagene Sprungfunktion ist also unbrauchbar. Wir verwenden statt dessen z. B. folgende Funktionen:

- $\tanh(z)$, die Tangens-Hyperbolicus-Funktion,
- $\logit(z)$, die logistische Funktion,
- $\text{ReLU}(z) = \max\{0, z\}$, *Rectified Linear Units*.

Die letzte Funktion ist sehr einfach, für negative Werte ist Null, für positive Werte der Wert selbst. Ihre Ableitung ist für negative Werte 0 (in einem solchen Neuron wird für den konkreten Datenpunkt nichts gelernt) und 1 für positive Werte. Obwohl diese Funktion nicht überall differenzierbar ist, wird sie heutzutage sehr häufig verwendet, da mit ihr meist schnellere Lernerfolge erzielt werden.

Als Fehlerfunktion verwendet man für Regressionsmodelle meist den mittleren quadratischen Fehler (*Mean Squared Error*), indem man für alle Trainingspunkte die quadrierten Differenzen zwischen Vorhersage und tatsächlichem Wert mittelt.

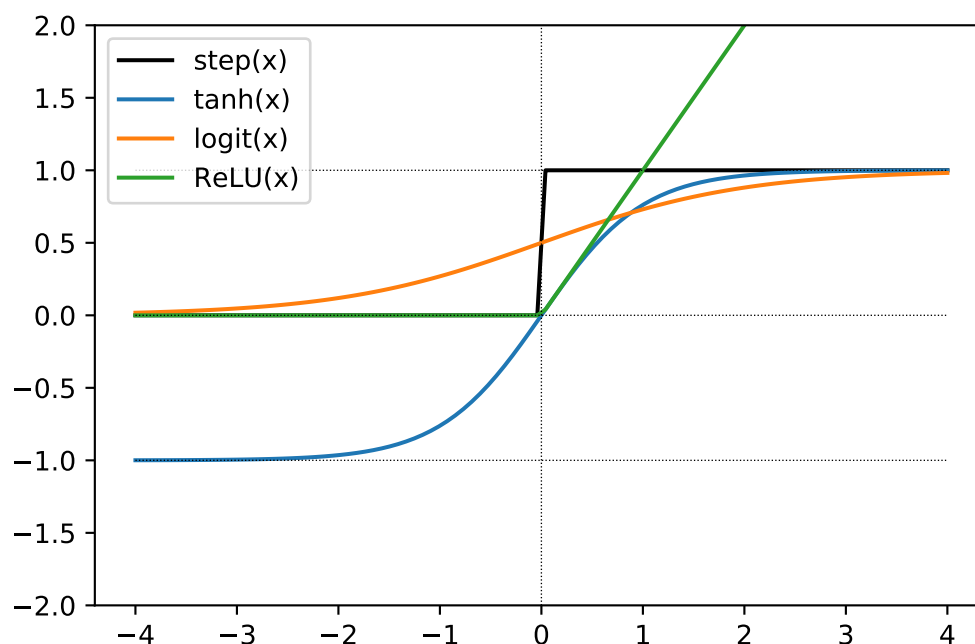


Abb. 17.3: Aktivierungsfunktionen

Für Klassifikationsprobleme ist das Vorgehen etwas komplizierter. Der Ausgangslayer hat für jede vorherzusagende Klasse genau einen Knoten. Unser Netz berechnet nun für eine bestimmte Eingabe für jeden Ausgangsknoten einen Wert. Wir wählen den Knoten k mit dem höchsten Wert für unsere Klassenvorhersage aus. Aus mathematischen Gründen verwenden wir den Wert des Knotens nicht direkt, sondern nutzen als Aktivierung die *Softmax*-Funktion, die dafür sorgt, dass zum einen die größten Werte verstärkt und die kleinsten weiter abgeschwächt werden, und zum anderen, dass die Summe aller Werte gerade 1 wird.

$$p_k = \frac{e^{z_k}}{\sum_{i=1}^K e^{z_i}}, \quad k = 1, \dots, K$$

Wir können die Ausgabe deshalb als eine Wahrscheinlichkeitsverteilung betrachten. Der Wert eines Knotens sagt aus, mit welcher Wahrscheinlichkeit der Datenpunkt zu dieser Klasse gehört. Eine passende Kostenfunktion für diese Ausgabefunktion ist die *Cross Entropy*.

$$L(y, t) = -\frac{1}{K} \sum_{i=1}^K t_i \log p_i$$

Zum Training berechnen wir nun die Vorhersagen für alle Trainingspunkte und daraus die Kostenfunktion summiert über alle Daten. Mit deren Ableitungen werden die Gewichte aller Knoten angepasst und anschließend der Prozess erneut gestartet.

Da dies sehr lange dauern kann, ist es ebenso möglich, nach der Vorhersage für jeden einzelnen Datenpunkt die Kostenfunktion nur für diesen zu berechnen und sofort die Gewichte zu korrigieren. Diese Methode nennt man *Online Learning*. Sinnvoll ist es, die Reihenfolge der Trainingspunkte zufällig zu wählen, um systematische Fehler zu vermeiden (*SGD*).

Der Nachteil ist ein starkes Schwanken der Gewichtswerte und der Modellqualität. Man wählt deshalb meist einen Kompromiss aus dem ersten und zweiten Vorgehen, das *Minibatch Learning*. Die Trainingsmenge wird für jeden Durchlauf zufällig in gleichgroße Teilmengen aufgeteilt. Nach der Berechnung aller Vorhersagewerte einer Teilmenge werden die Gewichte justiert.

Die Verarbeitungsschritte zu einem kompletten Durchlauf durch alle Trainingsdaten nennt man *Epoche*.

17.3 Initialisierung

Neben dem Training ist auch die Anfangsinitialisierung der Gewichte wichtig. Haben nämlich alle Knoten einer Schicht anfangs die gleichen Gewichte, lernt verfahrensbedingt auch jeder Knoten das gleiche, und wir können auf alle Knoten bis auf einen in jeder Schicht verzichten.

Aus diesem Grunde erfolgt die Initialisierung der Gewichte zufällig. Meist wird dazu eine Gleich- oder Normalverteilung mit dem Mittelwert 0 gewählt. Die Varianz wird in Abhängigkeit der Knotenzahl dieser Schicht passend gewählt (*Xavier*- bzw. *He*-Initialisierung).

17.4 Vorbereitung der Daten

Damit das Training eines neuronalen Netzes erfolgreich ist, müssen alle Gewichte der Neuronen einen Wert haben, der nahe bei Null liegt, andernfalls werden entweder die Gradienten für die Korrektur zu groß (*exploding gradients*) oder zu klein (*vanishing gradients*). Dies wird erleichtert, indem wir wie schon bei den klassischen *ML*-Verfahren die Daten vorher passend skalieren. Ebenso müssen wir kategoriale Größen wieder per *One-Hot-Encoding* in numerische Werte umwandeln.

Neuronale Netze mit Tensorflow

- *TensorFlow*
- *Ein einfaches ANN für Regression*
- *ANN für 2-Klassen-Modelle*
- *ANN für Mehrklassenmodelle*
- *Funktionale Modelle*
- *Klassenmodelle*
- *Speichern und Laden*
- *Ausgabeschicht, Kostenfunktion und Metriken*
- *Optimierungsverfahren*
- *Aufgaben*

18.1 TensorFlow

TensorFlow ist eine von *Google* entwickelte Bibliothek für schnelle Berechnungen auf Matrizen und Tensoren, wie sie für das Training und die Vorhersagen mit *ANN* benötigt werden. Die Berechnungen werden dazu automatisch auf Graphikkarten oder mehrere Computer verteilt. Unsere Datenbereitstellung erfolgt wie üblich mittels *NumPy*-Datenstrukturen, das Präprocessing (Aufteilen in Trainings- und Testdaten, Skalierung) kann wie bisher mit den *scikit-learn*-Werkzeugen erfolgen.

Zur Unterstützung von *ANN* implementiert *TensorFlow* die *Keras*-Bibliothek als Abstraktionsschicht.

Diese Pakete sind standardmäßig noch nicht in *Anaconda 3* installiert, wir müssen diese manuell nachinstallieren:

```
conda install tensorflow
# oder bei vorhandener Nvidia-Graphikkarte mit CUDA-Unterstützung
conda install tensorflow-gpu
```

Hinweis: Für die Nutzung der *GPU* wird das *CUDA*-Toolkit von *NVIDIA* benötigt, eventuell sind deshalb manuelle

Nachinstallationen, auch auf Betriebssystemebene, erforderlich.

Anschließend sollten wir die Funktion nach dem Start von Python prüfen:

```
import tensorflow as tf

print(tf.__version__)
```

Hinweis: Erhalten wir hier eine Fehlermeldung `Illegal instruction (core dumped)`, ist unser Prozessor für die aktuelle TensorFlow-Version zu alt. Wir können auf eine ältere Version wechseln, müssen allerdings auch Python in einer älteren Version nutzen:

```
conda uninstall tensorflow
conda install python=3.7.9
conda install tensorflow=2.1.0
```

Die Verwendung älterer Versionen als 2.0.0 ist nicht sinnvoll.

Um zu prüfen, ob tatsächlich die *GPUs* verwendet werden, benutzen wir folgende Funktion:

```
tf.config.list_physical_devices()
```

Für die Nutzung müssen wir folgende Pakete importieren:

```
# TensorFlow-Basisfunktionen
import tensorflow as tf
# Keras-Abstraktionsschicht
from tensorflow import keras
```

18.2 Ein einfaches ANN für Regression

Nachdem wir unsere Daten wie üblich vorbereitet haben:

- Einlesen, Aufteilen in Trainings- und Testmenge,
- Abtrennen der Zielfunktion,
- Umskalieren der numerischen, Umcodieren der kategorialen Komponenten,

müssen wir zunächst die Topologie des *ANN* erstellen. Dazu gibt es drei Methoden. Wir verwenden zunächst den einfachsten Ansatz mit der Klasse `keras.models.Sequential()` in der alle Schichten einfach nacheinander angeordnet und vollständig miteinander verbunden werden (*Fully Connected Network*).

```
model = keras.Sequential()
model.add(LayerKlasse(...))
...
```

oder alternativ:

```
model = keras.Sequential([
    LayerKlasse(...)
    ...
])
```

Wir benötigen zunächst nur vollständig vernetzte Schichten, die mit der Klasse `keras.layers.Dense` erzeugt werden:

```
keras.layers.Dense(neuronenZahl[, activation=aktivierungsfunktion...])
```

Für die erste Schicht müssen wir die Zahl der Komponenten angeben. Das geschieht entweder über den Parameter `input_shape` dieser Schicht oder das Voranstellen eines Pseudo-Layers `keras.layers.Input`, wie dies später beim funktionalen Modell erforderlich ist.

```
keras.layers.Input(neuronenZahl, input_shape=dimensionsListe)
```

Die erste Dimension ist dabei die Zahl der gleichzeitig zu verarbeitenden Daten und wird weggelassen. Wir müssen also nur die Zahl der Komponenten angeben, aber unbedingt in Form einer Liste.

Unsere Leistungsfähigkeit erhält unser Netz durch die nichtlineare Aktivierungsfunktion, die wir in mindestens einer verborgenen Schicht verwenden. Praktisch sinnvoll ist hier vor allem die *ReLU*-Funktion.

Die Ausgabeschicht soll jedoch beliebige reelle Werte ausgeben können, wir lassen deshalb die Aktivierung weg und berechnen unsere Vorhersage allein aus der Summe der gewichteten Eingangsdaten und dem Bias.

Unser Beispielnetz für den *Abalone*-Datensatz, der nach der One-Hot-Codierung 10 Komponenten enthält, könnte wie folgt aussehen:

```
model = keras.layers.Sequential([
    keras.layers.Input(Xtrain.shape[-1]),
    keras.layers.Dense(20, activation='relu', name='L1'),
    keras.layers.Dense(1, name='out'),
])

# alternativ:
model = keras.models.Sequential()
model.add(keras.layers.Dense(20,
    input_shape=Xtrain.shape[1:], # statt Input()-Layer, Liste mit 1 Wert
    activation='relu', name='L1'))
model.add(keras.layers.Dense(1, name='out'))
```

Wir können uns die Struktur dieses Modells anschauen:

```
model.summary()
```

Quellcode 18.1: Struktur des ANN

```
Model: "sequential"
_____
Layer (type)                Output Shape                Param #
=====
L1 (Dense)                  (None, 20)                  220
_____
out (Dense)                 (None, 1)                   21
=====
Total params: 241
Trainable params: 241
Non-trainable params: 0
_____
```

Wir sehen für jeden Layer den Name, die Ausgabedimension und die Zahl der Parameter. Für L1 ist diese 220 = 10 Komponenten × 20 Neuronen + 20 Bias-Werte.

Unter der Tabelle sehen wir nochmals die Summe aller Parameter. Für das *Transfer Learning* aus vorhandenen Modellen ist es sinnvoll, wenn bestimmte Schichten beim Training fixiert werden, dann unterscheiden sich die Zahlen für die Gesamtzahl und die der trainierbaren Parameter.

Eine graphische Anzeige ist ebenfalls möglich:

```
keras.utils.plot_model(model, show_shapes=True)
```

Damit diese Methode gerufen werden kann, muss das Paket `pydot` installiert werden (sollte dies wegen falscher Abhängigkeiten mittels **conda** fehlschlagen, verwenden wir `pip install pydot`).

Wir können mit unserem noch untrainierten Netz versuchen, Vorhersagen zu treffen:

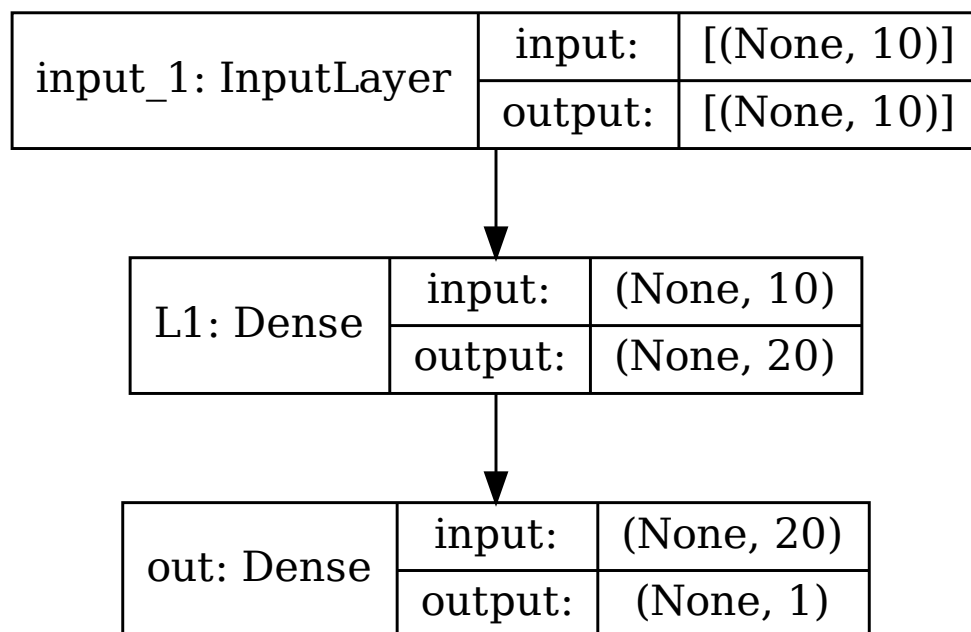


Abb. 18.1: Einfaches sequentielles Regressionsmodell.

```
predTest = model.predict(Xtest)
```

Schauen wir uns die Struktur der Ausgabe an, sehen wir, dass diese eine Matrix mit n Zeilen und einer Spalte ist. Das ist eine generelle Eigenschaft dieser Art von Netzen, dass die Zahl der Ausgabeknoten der Spaltenzahl der Vorhersage entspricht. Es bietet sich an, bei Regressionsproblemen eine Umwandlung in einen Vektor vorzunehmen:

```
predTest = model.predict(Xtest).reshape(-1)
```

Wir können mittels Scatterplot unsere wahren Funktionswerte mit der Vorhersage vergleichen:

```
plt.scatter(yTest, predTest)
```

Vor dem Training sollten alle vorhergesagten Werte in der Nähe der Null liegen.

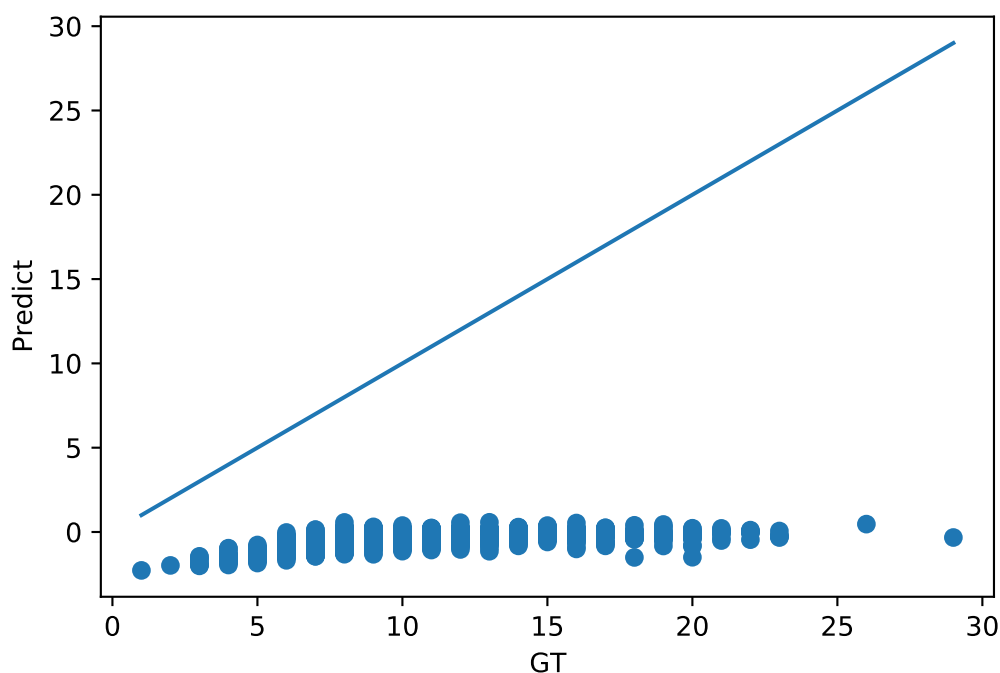


Abb. 18.2: Vorhersage des untrainierten Netzes.

Um das Netz trainieren zu können, müssen wir nun die Trainingsschleife erzeugen:

```
model.compile(
    [optimizer=optimierungsVerfahren,]
    loss=kostenFunktion,
    [metrics=[metriken],]
    ...
)
```

Zwingend ist lediglich die Angabe einer Kostenfunktion. Für Regressionsprobleme ist meist der mittlere quadratische Fehler die beste Wahl, wir können diese einfach als String 'mse' angeben. Als weitere Metriken können wir z. B. den mittleren absoluten Fehler berechnen.

```
model.compile(
    loss='mse',
    metrics='mae'
)
```

Nun können wir das Modell tatsächlich trainieren.

```
history = model.fit(X, y,  
                    [epochs=epochen, ]  
                    [batch_size=minibatchGröße, ]  
                    [verbose=detailGrad, ]  
                    ...  
                    )
```

Neben den Trainingskomponenten und der Zielgröße sollten wir die Anzahl der Epochen angeben, sonst erfolgt nur ein einzelner Durchlauf. Für erste Tests sollte ein Wert zwischen 10 und 50 gewählt werden. Wird die Minibatch-Größe nicht angegeben, wird automatisch ein sinnvoller Wert berechnet.

In unserem Beispiel verwenden wir 20 Epochen.

```
hist = model.fit(Xtrain, yTrain, epochs=20)
```

Wenn wir die Ausgabe nicht abschalten, werden uns pro Epoche der Lernfortschritt, der Wert der Kostenfunktion und die Metriken angezeigt.

```
Epoch 1/20  
92/92 [=====] - 1s 1ms/step - loss: 91.3275 - mae: 8.9952  
Epoch 2/20  
92/92 [=====] - 0s 1ms/step - loss: 56.6537 - mae: 6.7259  
...  
Epoch 20/20  
92/92 [=====] - 0s 1ms/step - loss: 4.1122 - mae: 1.4006
```

Das History-Objekt enthält diese Daten ebenfalls. Die Komponente `epoch` enthält die Liste der Epochen, die Komponente `history` ein Dictionary mit den Listen der Kostenfunktionswert und den Metriken für die Epochen. Wir können den Lernfortschritt einfach darstellen:

```
plt.plot(hist.epoch, hist.history['loss'])  
plt.plot(hist.epoch, hist.history['mae'])
```

Die »Güte« des Modells kann mit der Methode `evaluate()` bestimmt werden (eine Methode `score()` existiert hier nicht).

```
model.evaluate(Xtrain, yTrain)
```

Es werden der Wert der Kostenfunktion und die der spezifizierten Metriken berechnet. Wir erhalten z. B.

```
40/40 [=====] - 0s 1ms/step - loss: 6.8450 - mae: 1.8194  
[6.844976902008057, 1.8193706274032593]
```

Ein Scatterplot des trainierten Netzes zeigt uns optisch die Vorhersage des trainierten Modells.

Wir sehen deutlich den Lernfortschritt, wenn auch die Vorsagegenauigkeit noch nicht besonders hoch ist. Dies kann verbessert werden durch:

- geeignetere Netztopologie,
- längeres Training,
- mehr Trainingsdaten,
- ...

Mit einigen Möglichkeiten wollen wir uns später beschäftigen.

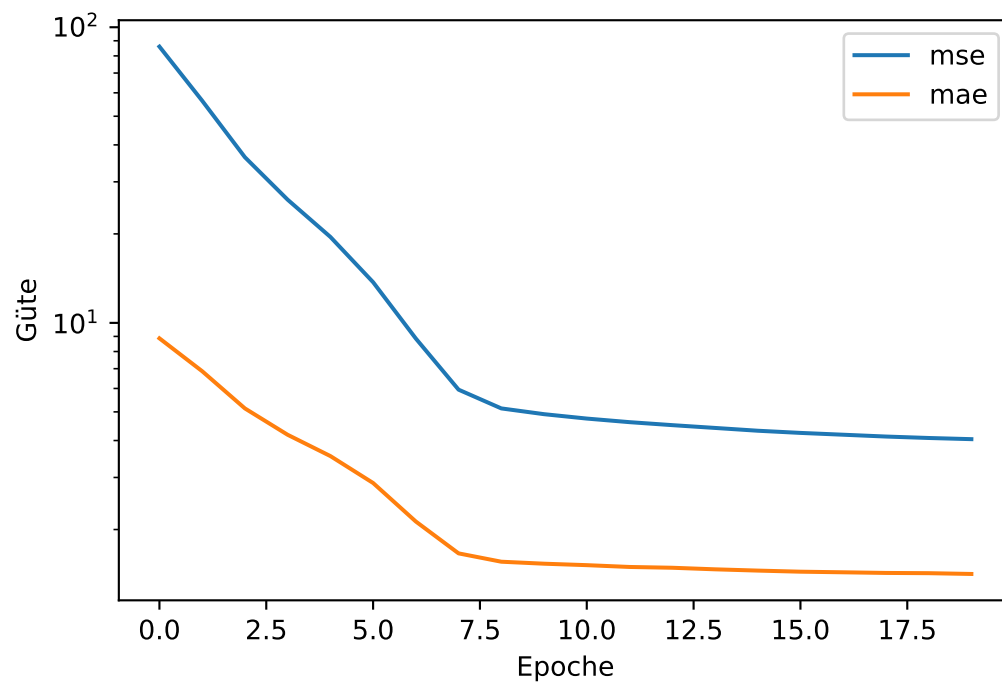


Abb. 18.3: Trainingsverlauf.

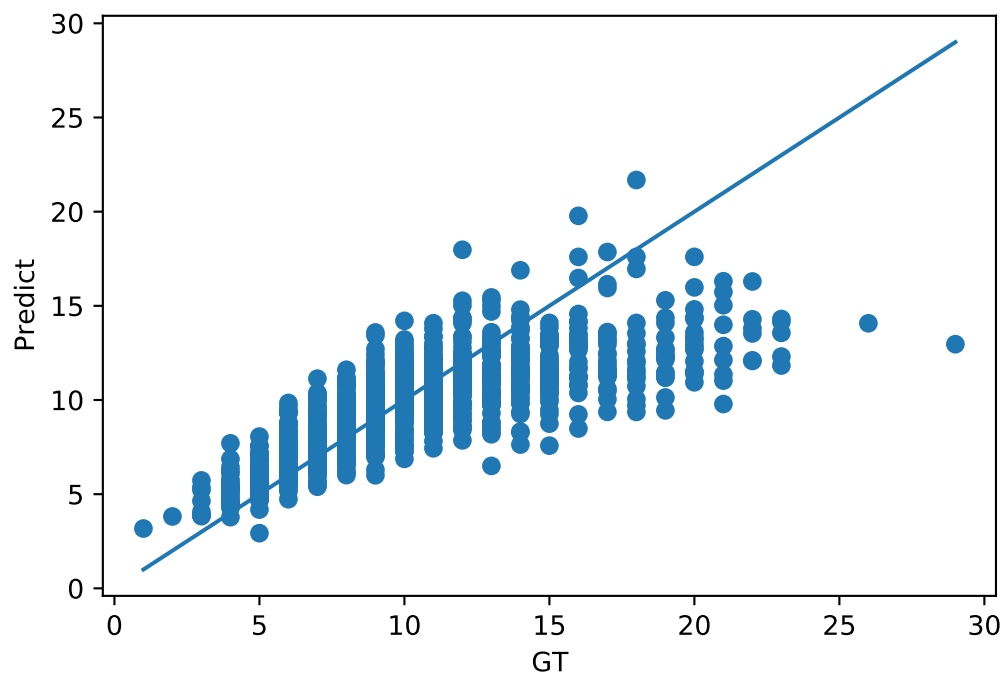


Abb. 18.4: Vorhersage des trainierten Netzes.

Notebook

Download: `abalone-tf-regr.ipynb` ([Preview](#))

18.3 ANN für 2-Klassen-Modelle

Warnung: Wenn wir mit *GPU*-Unterstützung arbeiten, darf immer nur ein Notebook aktiv sein, sonst entstehen merkwürdige Fehler. Wir müssen also die Arbeit mit dem vorigen Notebook unbedingt beenden oder dessen Kernel anhalten.

Wollen wir statt einer stetigen Größe zwei verschiedene Klassen (negativ/positiv bzw. 0/1) vorhersagen, gehen wir bei der *logistischen Regression* vor: wir definieren eine Ausgabeschicht mit einem Knoten und wenden auf dessen Ausgabe die logistische Funktion an:

$$\text{logit}(x) = \frac{1}{1 + e^{-x}}$$

Diese transformiert alle Werte in das Intervall $[0, 1]$, die wir dann als Wahrscheinlichkeiten für die Zugehörigkeit zur Klasse 1 betrachten – ist die Wahrscheinlichkeit kleiner als $\frac{1}{2}$, dann ordnen wir die Klasse 0, ist sie größer als $\frac{1}{2}$, dann ordnen wir die Klasse 1 zu. Die logistische Funktion geben wir entweder über ihren Identifikationsstring als 'sigmoid' oder als *Keras*-Objekt `keras.activations.sigmoid()` an.

```
model = keras.models.Sequential()
# ...
# Ausgabeschicht
model.add(keras.layers.Dense(1, activation='sigmoid', name='out'))
```

Unser Netz mit 2 verborgenen Schichten könnte nun z. B. so aussehen:

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(30, input_shape=Xtrain.shape[1:],
                             activation='relu', name='L1'))
model.add(keras.layers.Dense(5, activation='relu', name='L2'))
model.add(keras.layers.Dense(1, activation='sigmoid', name='out'))
```

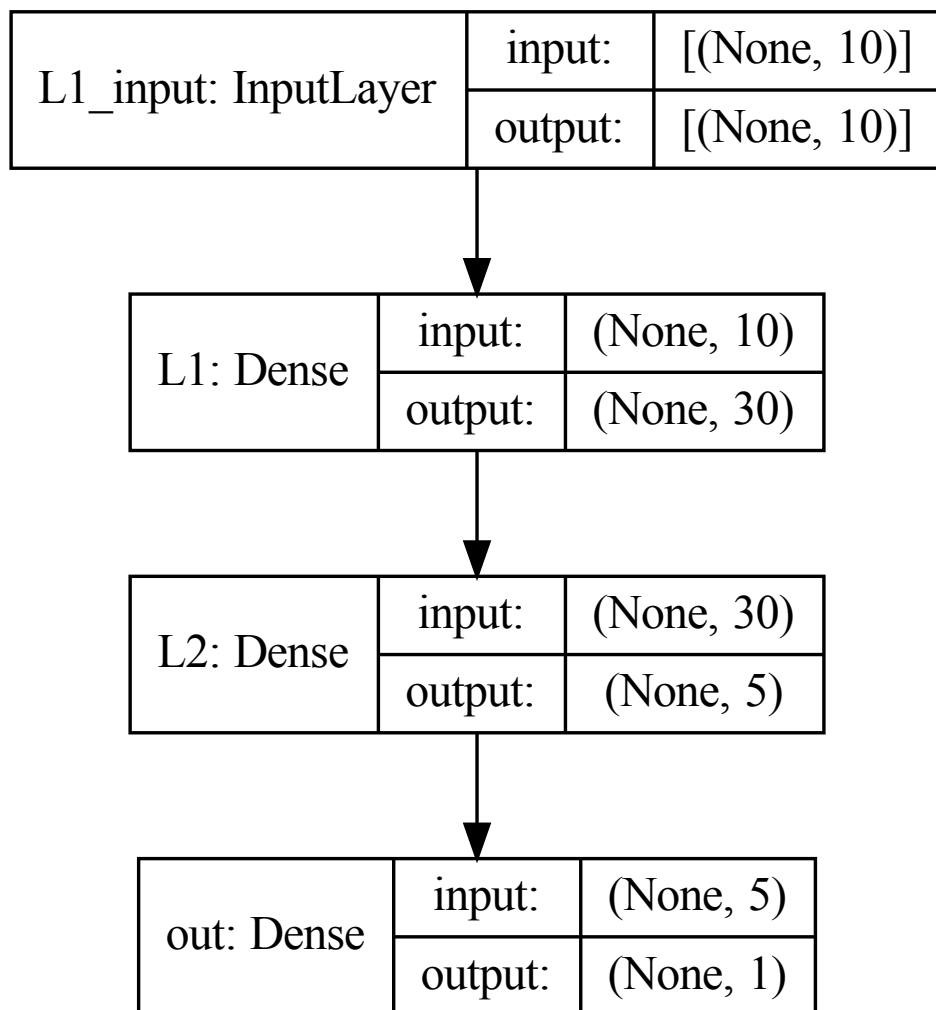
Die Methode `summary()` zeigt uns wieder die Topologie an:

```
Model: "sequential"
Layer (type)                Output Shape                Param #
=====
L1 (Dense)                   (None, 30)                  330
L2 (Dense)                   (None, 5)                   155
out (Dense)                  (None, 1)                   6
=====
Total params: 491
Trainable params: 491
Non-trainable params: 0
```

bzw. als Graphik:

Beim Erzeugen der Trainingsschleife benötigen wir eine zur logistischen Funktion passende Kostenfunktion, die *Binary Crossentropy*:

$$L = t \cdot \log p + (1 - t) \cdot \log(1 - p)$$



Hier ist t die tatsächliche Klasse (0 oder 1) und p die Vorhersage (stetiger Wert zwischen 0 und 1).

Die Kostenfunktion wird wie folgt erzeugt:

```
keras.losses.BinaryCrossentropy()
```

Es existiert hier auch ein Stringkürzel 'binary_crossentropy'.

Warnung: Die Kostenfunktion `BinaryCrossentropy()` besitzt einen Parameter `from_logits`. Dieser Parameter darf nur dann auf `True` gesetzt werden, wenn die Ausgabe unseres Netzes beliebige Werte zwischen $-\infty$ und $+\infty$ annehmen kann. Die *Sigmoid*-Aktivierung liefert dagegen *Wahrscheinlichkeiten* mit Werten zwischen 0 und 1, dann muss dieser Parameter `False` sein bzw. entfallen.

Als Bewertungsmaße können wir an erster Stelle die Genauigkeit verwenden (Schlüssel `accuracy`), die den Anteil der richtig klassifizierten Daten beschreibt. Viele andere Maße stehen als Klassen bereit, z. B. `keras.metrics.FalsePositives`.

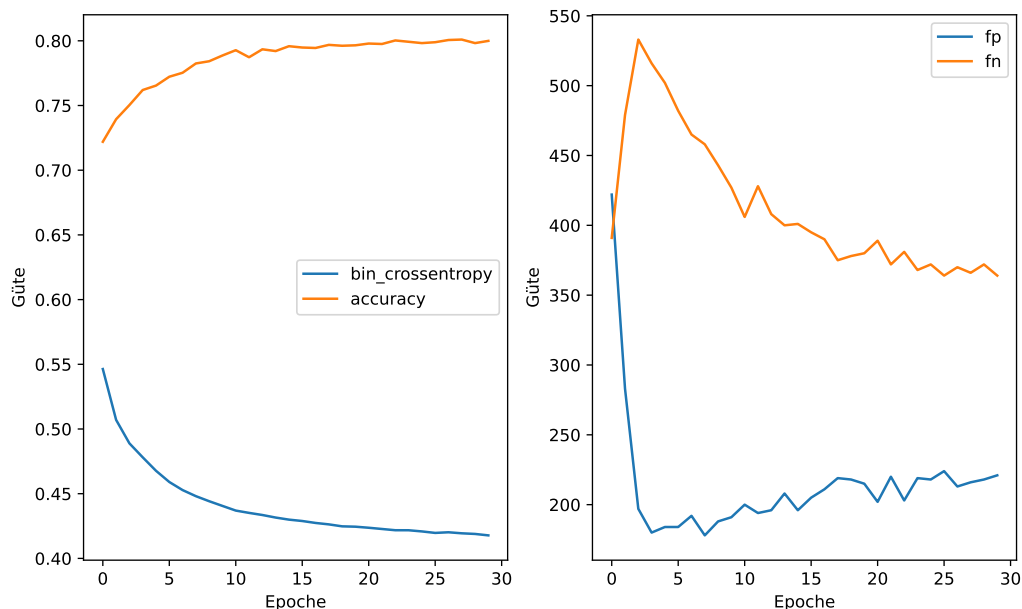
Wir definieren unsere Trainingsschleife für die binäre Klassifikation:

```
model.compile(
    loss=keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=['accuracy',
            keras.metrics.FalsePositives(name='fp'),
            keras.metrics.FalseNegatives(name='fn')]
)
```

Das Training läuft völlig identisch zum Regressionsmodell.

```
model.fit(Xtrain, yTrain, epochs=30)
```

Wir erhalten folgende Lernkurven, aufgetrennt in zwei Teildiagramme:



Notebook

Download: `abalone-tf-binary.ipynb` ([Preview](#)).

18.4 ANN für Mehrklassenmodelle

Wollen wir mehr als zwei Klassen vorhersagen, sehen wir in unserer Ausgabeschicht pro Klasse einen eigenen Knoten vor. Als Aktivierung verwenden wir die Softmax-Funktion, die uns die Ausgabewerte der Knoten in einen Vektor mit Wahrscheinlichkeiten umwandelt:

```
keras.layers.Dense(klassenZahl, activation='softmax' ...)
# oder
keras.layers.Dense(klassenZahl, activation='keras.activation.softmax(
...)
```

Um die Trainingsschleife erstellen zu können, benötigen wir eine geeignete Kostenfunktion. Im Prinzip bestimmen wir dazu für jede Klassenwahrscheinlichkeit die Differenz zu 0 (falsche Klasse) bzw. 1 (korrekte Klasse). Dazu müssen wir die tatsächlichen Klassenwerte in eine One-Hot-Codierung umwandeln, so dass wir jedem Ausgabeknoten den korrekten Wert 0 oder 1 zuordnen können. Diese Umwandlung kann entweder manuell beim Vorbereiten der Daten geschehen. Einfacher ist jedoch die automatische Umwandlung durch die passende Kostenfunktion selbst:

```
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    ...
)
```

Statt des Klassennamens können wir auch den Stringbezeichner `'sparse_categorical_crossentropy'` verwenden.

Als vordefinierte Metrik steht uns hier nur die Genauigkeit (`'accuracy'`) zur Verfügung.

Das Netz und die Trainingsschleife für den *Abalone*-Datensatz mit 3 Klassen könnte wie folgt aussehen:

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(20, input_shape=Xtrain.shape[1:],
                             activation='relu', name='L1'))
model.add(keras.layers.Dense(20, activation='relu', name='L2'))
model.add(keras.layers.Dense(3, activation='softmax', name='out'))
model.summary()

model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)
```

Wir erhalten folgende Topologie:

```
Model: "sequential"
_____
Layer (type)                Output Shape                Param #
=====
L1 (Dense)                  (None, 20)                  220
_____
L2 (Dense)                  (None, 20)                  420
_____
out (Dense)                 (None, 3)                   63
=====
Total params: 703
Trainable params: 703
Non-trainable params: 0
_____
```

Der Trainingsprozess verläuft nun völlig identisch zur Regression und binären Klassifikation, ebenso können wir mit

```
model.evaluate(Xtest, yTest)
```

unser Modell bewerten.

Erstellen wir eine Vorhersage des Modells, ist unser Ergebnis kein Vektor, sondern eine Matrix, die je Datenpunkt eine Zeile und je Klasse eine Spalte besitzt.

```
probTest = model.predict(Xtest)
probTest.shape
```

Für die Klassenvorhersage müssen wir aus unserer Vorhersagematrix jeweils die Spalte auswählen, die den größten Wahrscheinlichkeitswert hat.

```
yTest = np.argmax(probTest, axis=1)
```

Zum besseren Verständnis zeigen wir für die ersten 10 Testdaten die tatsächliche Klasse, die vorhergesagten Wahrscheinlichkeiten und die vorhergesagte Klasse an.

```
probTest = model.predict(Xtest)
predTest = np.argmax(probTest, axis=1)
print('GT  C10  C11  C12 Pred')
print('-----')
for i in range(10):
    print('{:2d} {:.2f} {:.2f} {:.2f} {:.2d}'.format(
        yTest[i], probTest[i,0], probTest[i,1], probTest[i, 2],
        predTest[i]
    ))
```

und erhalten

```
GT  C10  C11  C12 Pred
-----
1 0.21 0.79 0.00 1
0 0.42 0.58 0.00 1
1 0.03 0.96 0.01 1
1 0.11 0.89 0.00 1
0 0.86 0.14 0.00 0
1 0.53 0.47 0.00 0
2 0.01 0.89 0.10 1
1 0.03 0.96 0.01 1
1 0.08 0.91 0.00 1
2 0.01 0.90 0.09 1
```

Unsere Vorhersage können wir nun wie bei den klassischen Modellen nutzen und bewerten, z. B. wieder die Konfusionsmatrix berechnen und graphisch darstellen.

```
# absolut
conf = metrics.confusion_matrix(yTest, predTest)
# relativ je Zeile: Anteil der korrekten/falschen Vorhersagen je Klasse
relConf = conf/np.sum(conf, axis=1, keepdims=True)
# graphische Darstellung
plt.imshow(relConf, cmap='hot')
```

Wir erhalten z. B. als Konfusionsmatrix

```
[[229  74   0]
 [ 41 832   4]
 [  0  71  21]]
```

und relativ

```
[[0.76 0.24 0.  ]
 [0.05 0.95 0.  ]
 [0.   0.97 0.03]]
```

Graphische Darstellung:

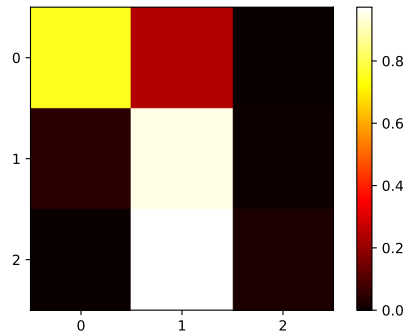


Abb. 18.5: Konfusionsmatrix Mehrfachklassifikation.

Notebook

Download: `abalone-tf-multiclass.ipynb` ([Preview](#)).

18.5 Funktionale Modelle

Flexibler als das sequentielle Modell ist die funktionale Methode. Hier definieren wir selbst an, wie die Verknüpfung zwischen den Ebenen stattfinden soll.

```
input = keras.layers.Input(datenSatzShape)
l1 = LayerKlasse(...) (vorigerLayer)
...
output = LayerKlasse(...) (vorigerLayer)

model = keras.Model(input, output)
```

Die Eingabeschicht `keras.layers.Input` ist hier zwingend erforderlich. Die Schichten werden wieder der Reihe nach definiert und anschließend wie eine Funktion aufgerufen, deren Parameter die Eingabe dieser Schicht ist, also die Ausgabe der gewünschten vorhergehenden Schicht.

Es geschehen also jeweils zwei Schritte in einer Zeile:

- Erzeugen des Objekts für eine Schicht,
- Verknüpfen dieses Objekts mit einer Eingabe und Definition der Ausgabe dieser Schicht.

Um unser Modell zu erstellen, verknüpfen wir die Eingabeschicht mit der gewünschten Ausgabeschicht. Das muss nicht unbedingt die letzte Schicht sein. Wenn wir z. B. wissen wollen, wie die Ausgabe eines Netzes nach einer bestimmten Schicht aussieht, so erstellen wir einfach ein neues Modell mit dieser Schicht als Ausgabeschicht und machen mit ihm eine Vorhersage.

Die Schichten werden wieder wie beim sequentiellen Modell verbunden. Um die neuen Möglichkeiten zu zeigen, verwenden wir noch einen neuen Schichttyp, der mehrere Eingaben verknüpfen kann und zu einer Ausgabe zusammenfasst:

```
concat = keras.layers.Concatenate() ([eingabe1, eingabe2 ...])
```

Beispiel: Funktionales Modell für *Abalone*, das zum einen die Ausgaben von Schicht 1 direkt und zum anderen nach einer Transformation durch Schicht 2 verwendet. Beide Ausgaben werden zusammengefasst und gehen dann in die Ausgabeschicht ein (*Skip Connections*: Schicht 2 wird übersprungen).

```
inp = keras.layers.Input(Xtrain.shape[1:])
l1 = keras.layers.Dense(20, activation='relu', name='L1')(inp)
l2 = keras.layers.Dense(20, activation='relu', name='L2')(l1)
skip = keras.layers.Concatenate(name='Merge')([l1, l2])
out = keras.layers.Dense(3, activation='softmax', name='out')(skip)

model = keras.models.Model(inp, out)
```

Für funktionale Modelle zeigt uns die Methode `summary()` auch die Eingangsdaten der jeweiligen Schicht an:

Quellcode 18.2: Funktionales Modell mit Skip-Connections

Model: "model"			
↪			
Layer (type)	Output Shape	Param #	Connected to
=====			
input_1 (InputLayer)	[(None, 10)]	0	
↪			
L1 (Dense)	(None, 20)	220	input_1[0][0]
↪			
L2 (Dense)	(None, 20)	420	L1[0][0]
↪			
Merge (Concatenate)	(None, 40)	0	L1[0][0] L2[0][0]
↪			
out (Dense)	(None, 3)	123	Merge[0][0]
=====			
Total params: 763			
Trainable params: 763			
Non-trainable params: 0			
↪			

In der graphischen Darstellung ist die Topologie besser zu erkennen:

Hinweis: In unserem Beispiel dient diese Netztopologie nur zur Demonstration und bringt keine Modellverbesserungen. Im allgemeinen kommt diese Topologie in Form von *Residual Networks* insbesondere bei der Bildklassifikation erfolgreich zur Anwendung.

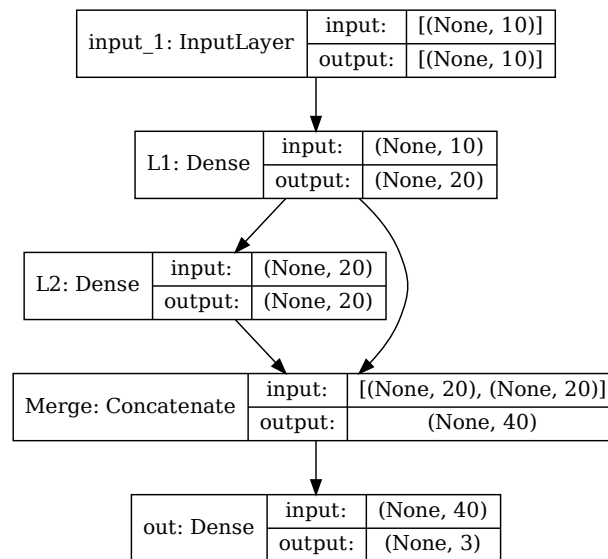


Abb. 18.6: Topologie eines Netzes mit Skip Connections.

Notebook

Beispiel funktionale Netzdefinition: `abalone-tf-functional.ipynb` ([Preview](#)).

18.6 Klassenmodelle

Wir können mit der Idee des funktionalen Modells auch ein eigenes, parametrisierbares Modell ableiten. Im Konstruktor erzeugen wir die Schichtobjekte. Eine Methode `call()` verknüpft dann diese Schichten zu einem Modell.

Wir leiten dazu von `keras.models.Model` eine eigene Klasse ab:

```

class MyModel(keras.models.Model):
    def __init__(self, ...):
        super().__init__()
        self.layer1 = keras.layers.Layer(...)
        self.layer2 = keras.layers.Layer(...)
        ...

    def call(self, inputs):
        x = self.layer1(inputs)
        x = self.layer2(x)
        ...
        return x
  
```

Die Modellinstanz wird nun einfach mit dieser Modellklasse erzeugt:

```
model = MyModel(hyperparameter)
```

Ohne weitere Spezifikation der Eingabedimension können wir die Trainingsschleife mittels `compile()` erzeugen und anschließend das Modell trainieren.

Beispiel für *Abalone* mit definierter Schichtbreite.

```

class AbaloneModel(keras.models.Model):
    def __init__(self, depth, classes):
        super().__init__()
        self.l1 = keras.layers.Dense(depth, activation='relu', name='L1')
        self.l2 = keras.layers.Dense(depth, activation='relu', name='L2')
        self.skip = keras.layers.Concatenate(name='Merge')
        self.out = keras.layers.Dense(classes, activation='softmax',
                                        name='out')
  
```

Der Vorteil ist, dass wir recht einfach verschiedene Modelle mit ähnlicher Topologie und verschiedenen Hyperparametern erzeugen und vergleichen können, der Nachteil ist die schlechtere Analysemöglichkeit. So liefert uns `summary()` eine Fehlermeldung, sofern wir nicht vorher explizit ein Modell mit vordefinierter Eingabestruktur mittels `build()` erzeugen.

Beispiel: Vergleich zweier Modelle für *Abalone* mit Schichtbreiten 10 und 5 nach 30 Epochen. Das erste Modell (mit breiteren Schichten) konvergiert schneller.

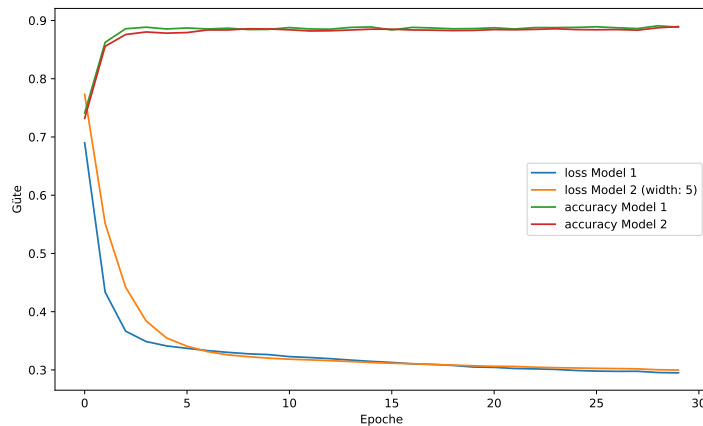


Abb. 18.7: Vergleich zweier Modelle.

Notebook

Objektorientierte Modellerstellung und Vergleich zweier Modellinstanzen: `abalone-tf-oo.ipynb` ([Preview](#)).

18.7 Speichern und Laden

Wollen wir unser Modell aufbewahren (oder auch mit den Startgewichten einen neuen Test starten), können wir unser Modell speichern.

```
model.save('dateiname')
# Nur die Gewichte
model.save_weights('dateiname')
```

Endet der Dateiname auf `h5`, dann wird das kompaktere *HDF5*-Format verwendet, andernfalls werden die Daten auf mehrere Dateien verteilt. Wir können entweder das komplette Modell speichern und laden oder aber das Modell in unserem Code neu erzeugen und nur die Gewichte (also die trainierbaren Parameter) sichern und wiederverwenden.

Das Laden eines kompletten Modells geschieht mit der Methode `load_model()`.

```
model = keras.models.load_model('dateiname')
```

Haben wir nur die Gewichte gespeichert, erzeugen wir unser Modell manuell mit der gleichen Topologie wie beim Speichern und laden anschließend die Daten.

```
model.load_weights('dateiname')
```

Wollen wir für unser Modell verschiedene Trainingsarten ausprobieren, aber die Netztopologie beibehalten, benutzt die `fit()`-Methode die von vorigen Trainingsläufen gelernten Parameter weiter, auch wenn wir `compile()` erneut aufrufen oder auch das Modell neu aus den bisherigen Layerobjekten zusammenstellen. Wollen wir also für Vergleiche immer mit unseren Initialdaten starten, speichern wir diese sofort nach der Modellerzeugung.

18.8 Ausgabeschicht, Kostenfunktion und Metriken

Die Kostenfunktion muss passend zur Aktivierungsfunktion der Ausgabeschicht gewählt werden. Ebenso sind je nach Ausgabeschicht unterschiedliche Metriken (Bewertungsmaße) erforderlich.

Die wichtigsten Kombinationen sind:

Binäre Klassifikation:

- Label: Vektor mit Werten 0 bzw. 1,
- Breite Ausgabeschicht: 1,
- Aktivierung: `sigmoid`,
- Ausgabe: Matrix mit 1 Spalte mit Wahrscheinlichkeit für Klasse 1
- **Kostenfunktion:** `binary_crossentropy` (**besser:** `keras.losses.BinaryCrossentropy(from_logits=True)`),
- Metriken:
 - `keras.metrics.Accuracy()`, 'accuracy',
 - `keras.metrics.FalsePositives()`,
 - `keras.metrics.FalseNegatives()`,
 - ...

Mehrfachklassifikation

- Label: Vektor mit Klassenlabel 0 ... k-1,
- Klassen: 0, ..., k-1,
- Breite Ausgabeschicht: k,
- Aktivierung: `softmax`,
- Ausgabe: Matrix mit k Spalten und Wahrscheinlichkeit für jeweilige Klasse,
- Kostenfunktion: `sparse_categorical_crossentropy`,
- Metriken:
 - `keras.metrics.Accuracy()`, 'accuracy',

Falls die Label als Matrix in One-Hot-Codierung vorliegen (Zeilen: Datenpunkte, Spalten: Klassen), dann ist die Kostenfunktion `categorical_crossentropy`.

Regression

- Zielgröße: Vektor mit reellen Zahlen,
- Breite Ausgabeschicht: 1,
- Aktivierung: keine,
- Ausgabe: Vektor mit stetigen Zielwerten,
- Kostenfunktion: `mean_squared_error`,
- Metriken:
 - `keras.metrics.MeanAbsoluteError()`, 'mae',

```
- keras.metrics.MeanAbsolutePercentageError(), 'mape',  
- keras.metrics.MeanRelativeError(),  
- ...
```

18.9 Optimierungsverfahren

Die vordefinierten Optimierungsverfahren beruhen alle auf dem *Gradient-Descent*-Algorithmus, verbessern ihn aber durch eine automatische Parametersteuerung. Hier ist es sinnvoll, nicht mit der Stringrepräsentation zu arbeiten, sondern tatsächlich eine Klasseninstanz zu erzeugen, da man dann die Lernrate mit angeben kann.

```
model.compile(  
    optimizer=OptimizerKlasse(lernRate:ms:`...`,  
    ...  
)
```

Obwohl auch für die Optimierungsverfahren die Kurzangabe mittels String möglich ist, sollte hier explizit eine Klasse instanziiert werden, da alle Optimierer durch *Hyperparameter* wie z. B. die Lernrate gesteuert werden.

Wichtige Optimiererklassen sind:

SGD Das klassische *Stochastic-Gradient-Descent*-Verfahren. Der erste Parameter ist die Lernrate, die meist angepasst werden muss. Dieses Verfahren ist häufig instabil und ist deshalb weniger zu empfehlen.

Adam Der *Adam*-Algorithmus stabilisiert das klassische Gradientenverfahren, indem die Gradientenrichtung beim nächsten Wert der Kostenfunktion nicht abrupt geändert, sondern die bisherige Richtung der Gewichtsänderung mit einbezogen wird. Dieser Optimierer funktioniert häufig sehr gut. Neben der Lernrate kann auch der Grad der Einbeziehung der bisherigen Richtung (*Moment*) eingestellt werden.

RMSprop Dieser Algorithmus stabilisiert die Gradientenrichtung durch Einbeziehen des Durchschnitts der vorigen Gradienten. Auch hier kann neben der Lernrate der Grad dieses Einflusses gesteuert werden. Dieses Verfahren wird benutzt, wenn wir beim Aufbau der Trainingsschleife mittels `compile()` keinen Optimierer angeben.

AdaGrad Auch hier wird die Gradientenrichtung durch der vorigen Gradienten beeinflusst.

Der erste Parameter ist bei allen drei Verfahren die Lernrate, die häufig angepasst werden muss. Bei einem zu kleinen Wert gibt es keinen Lernfortschritt, bei einem zu großen Wert wird das Ziel – das Minimum der Kostenfunktion – übersprungen und das Verfahren oszilliert ohne einen Lernfortschritt.

Hinweis: In den meisten Situationen funktionieren die Verfahren `RMSprop` und `Adam` am besten. `SGD` ist nicht zu empfehlen.

18.10 Aufgaben

Aufgabe 18.1 Erstellen Sie verschiedene Modelle für Ihren Datensatz. Untersuchen Sie, wie sich Anzahl von Schichten, Schichtbreiten und auch die Lernrate auf den Trainingsprozess auswirken.

Erstellen Sie Ihre Netze nicht nur mit dem sequentiellen Modell, sondern versuchen Sie, den funktionalen Ansatz zu verstehen.

Regularisierung von Neuronalen Netzen

- *Vorbemerkungen*
- *Verwenden einer Validierungsmenge*
- *Early Stopping*
- *L2-Regularisierung*
- *Dropout*
- *Batch-Normalisierung*
- *Anpassen der Lernrate*
- *Aufgaben*

19.1 Vorbemerkungen

Unter *Regularisierung* fasst man alle Methoden zusammen, die dafür sorgen, dass das Modell nicht zu komplex wird, also *Overfitting* vermieden wird. Da gerade *ANN* sehr viele erlernbare Parameter haben, die die Zahl der Trainingsdaten leicht übersteigt, ist diese Gefahr recht groß, insbesondere, wenn nicht ausreichend Trainingsdaten zur Verfügung stehen.

Hinweis: Bei unserem Beispieldatensatz *Abalone* sind durch die beschriebenen Verfahren nur geringe oder keine Verbesserungen erzielbar. Die Ursachen könnten sein:

- Die Daten enthalten zu wenig Informationen über die Zielgröße.
 - Der Umfang der Daten ist zu gering.
 - Die gewählte Netzwerktopologien sind nicht optimal: wir wählen hier recht einfache, kleine Netze.
-

19.2 Verwenden einer Validierungsmenge

Bevor wir uns mit Regularisierungstechniken beschäftigen, wollen wir zunächst den Lernprozess besser kontrollieren, um ein Overfitting zu vermeiden. Wir trennen dazu wie bei der Suche nach Hyperparametern von unseren Trainingsdaten eine Validierungsmenge ab. Dies kann entweder manuell mit der *Scikit-learn*-Methode `model_selection.train_test_split()` oder automatisch über einen Parameter der `fit`-Methode des Modells geschehen.

Wir trennen zunächst manuell eine Validierungsmenge ab:

```
Xtrain1, Xvalid, yTrain1, yValid = model_selection.train_test_split(
    Xtrain, yTrain, test_size=0.2, random_state=23
)
```

Beim Training stellen wir die Validierungsmenge als Paar (X, y) über den Parameter `validation_data` bereit.

```
hist = model.fit(Xtrain1, yTrain1, validation_data=(Xvalid, yValid),
    epochs=50,
    # weitere Parameter
)
```

Während des Trainings, das weiterhin nur anhand der (neuen) Trainingsdaten erfolgt, werden die Kostenfunktion und die Metriken zusätzlich für die Validierungsmenge bestimmt und in das `History`-Objekt übernommen.

Beispiel *Abalone* für Mehrfachklassifikation.

```
model.compile(
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
Xtrain1, Xvalid, yTrain1, yValid = model_selection.train_test_split(
    Xtrain, yTrain, test_size=0.2, random_state=23
)
hist = model.fit(Xtrain1, yTrain1, validation_data=(Xvalid, yValid),
    epochs=50,
    batch_size=32,
    verbose=2)
```

Das `History`-Objekt `hist` enthält in der Komponente `history` ein Dictionary mit den Schlüsseln

loss Kostenfunktion der Trainingsdaten,

val_loss Kostenfunktion der Validierungsdaten,

accuracy Genauigkeit auf den Trainingsdaten,

val_accuracy Genauigkeit auf den Validierungsdaten.

Graphische Darstellung:

Das zweite Diagramm zeigt statt der Genauigkeit den Fehler an, also den Wert *1-Genauigkeit*. Diesen können wir einfach mittels *NumPy* ermitteln, wir müssen lediglich das Listenobjekt aus dem `History`-Objekt in ein *NumPy*-Array umwandeln:

```
errorVector = 1.0 - np.array(history.history['accuracy'])
```

Wir erkennen eine Verbesserung der Kostenfunktion und auch der Genauigkeit über den gesamten Trainingszeitraum, jedoch kaum noch eine Verbesserung auf den Validierungsdaten nach der 20. Epoche. Wir stellen hier nicht die Genauigkeit selbst, sondern die Differenz zu 1, also den Fehler, dar.

Statt der manuellen Abtrennung der Trainingsdaten können wir die Aufteilung auch automatisch von der `fit()`-Methode erledigen lassen.

```
hist = model.fit(Xtrain, yTrain,
    validation_split=anteilValidierung,
    ...
```

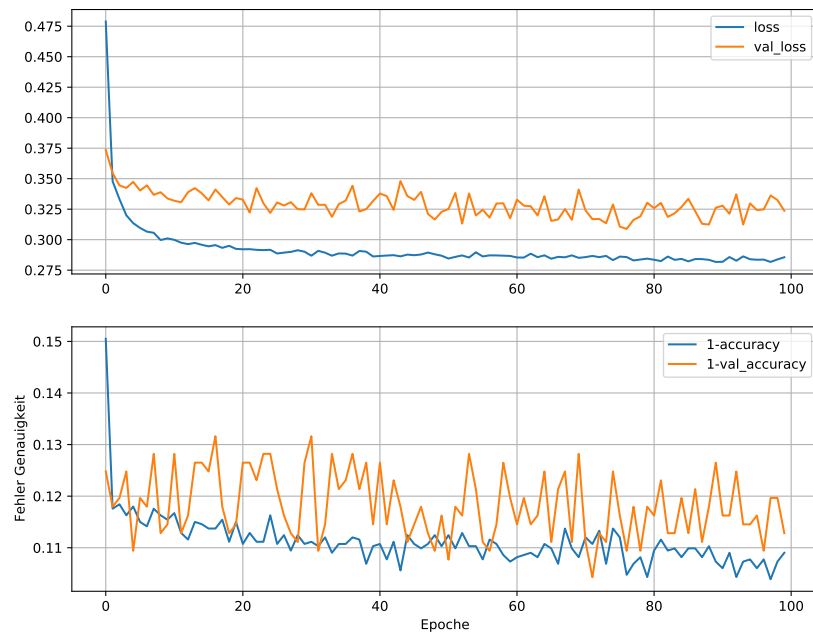


Abb. 19.1: Trainingsfortschritt.

)

Beispiel für *Abalone* mit anderem Netz und anderem Optimierungsverfahren. Wir verringern hier die Lernrate etwas.

```
model2.compile(
    optimizer=keras.optimizers.Adam(0.0005),
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)

hist2 = model2.fit(Xtrain, yTrain, validation_split=0.2,
    epochs=50,
    batch_size=8,
    verbose=2
)
```

Schauen wir uns den Trainingsfortschritt graphisch an, erkennen wir eine Verbesserung der Kostenfunktion auf den Trainingsdaten über alle Epochen, jedoch keine weitere Verbesserung auf den Validierungsdaten nach der 30. Epoche. An dieser Stelle beginnt eventuell das Overfitting.

Die Überwachung der Metriken der Validierungsmenge kann zur Regularisierung benutzt werden, indem wir automatisch das Training beenden, sobald keine Verbesserungen mehr auf der Validierungsmenge erreicht werden können (*Early Stopping*).

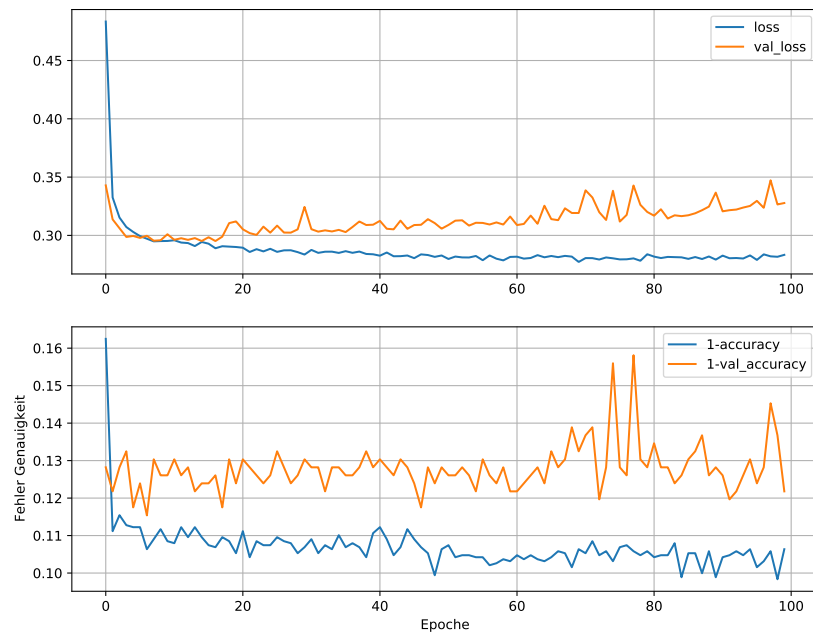


Abb. 19.2: Trainingsfortschritt bei automatischer Wahl der Validierungsmenge.

Notebook

Download: `abalone-validation-manual.ipynb`.

19.3 Early Stopping

Um den Trainingprozess zu beenden, sobald sich keine Verbesserung mehr auf den Validierungsdaten ergibt, registrieren wir in unserer `fit()`-Methode ein oder mehrere *Callback*-Objekte:

```
hist = model.fit(
    Xtrain, yTrain,
    callbacks=[callback1[, callback2 ...]],
    ...
)
```

Für das *Early Stopping* definieren wir

```
callback = keras.callbacks.EarlyStopping(
    monitor='metrik',
    mode='mode',
    patience=epochen,
    [restore_best_weights=True,]
    [verbose=1,]
    ...
)
```

Mit dem Parameter `monitor` sollte entweder die Kostenfunktion oder eine der Metriken auf der *Validierungsmenge* überwacht werden. Diese Metriken müssen in der Trainingsschleife (Methode `compile`) aufgeführt sein und einen Namen erhalten (sofern nicht vorhanden). Für die Validierungsmenge wird das Prefix `val_` davor gesetzt.

Der Parameter `mode` kann die Werte `min`, `max` oder `auto` (Vorgabe) annehmen und gibt an, ob ein kleinerer oder ein größerer Wert eine Verbesserung bedeutet. Verwenden wir Keras-Standard-Metriken, findet `auto` das korrekte Verhalten automatisch.

Der Parameter `patience` spezifiziert, wieviele Epochen wir nach einem gefundenen Minimum weitergesucht wird, ob sich eine weitere Verbesserung ergibt.

Nach jeder Epoche wird nun geprüft, ob sich Güte sich verbessert hat. Wird sie im folgenden Wartezeitraum nur noch schlechter, wird das Training abgebrochen. Mit dem Parameter `restore_best_weights` wird unser Modell auf die Parameter des besten Modells zurückgesetzt (da sich ja durch die weitere Suche das Modell wieder verschlechtert hat).

Beispiel für unseren Datensatz.

```
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=30,
    restore_best_weights=True,
    verbose=1
)

history = model.fit(
    Xtrain, yTrain,
    validation_split=0.2,
    epochs=200, batch_size=8,
    callbacks=[early_stopping],
    verbose=1
)
```

Schauen wir uns hier den Trainingsfortschritt an, sehen wir, dass unser Training abbricht, sobald sich die Genauigkeit nicht mehr verbessert, was weit vor der 200. Epoche passieren sollte.

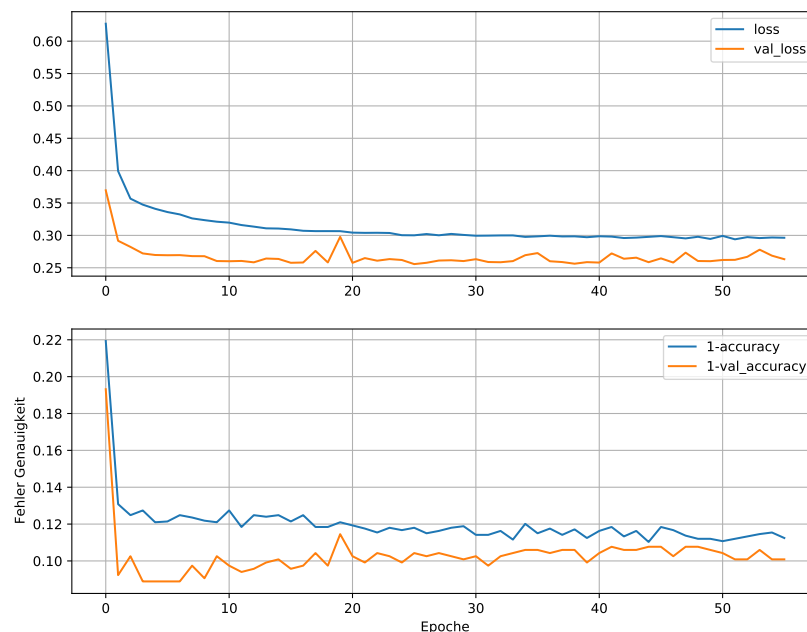


Abb. 19.3: Trainingsprozess mit Early Stopping

Warnung: Für das Early Stopping sollte unbedingt eine Metrik der Validierungsmenge verwendet werden. Verwenden wir dagegen eine der Trainingsmenge, lernen wir diese auswendig und erzeugen ein Overfitting, das wir gerade vermeiden wollten.

Notebook

Download: `abalone-early-stopping.ipynb`

19.4 L2-Regularisierung

Während wir beim *Early Stopping* lediglich den Trainingsprozess in der Dauer beschränken, können wir wir auch direkt im Modell Regularisierungen vornehmen.

Mit der *L2-Regularisierung* sorgen wir dafür, dass die Gewichte unserer Neuronen nicht zu groß werden dürfen, damit nicht einzelne Neuronen in der Berechnung einen zu großen Einfluss auf das Ergebnis bekommen. Dazu berechnen wir wieder die Summe der Quadrate aller Gewichte und nehmen diese als zusätzlichen Term in die Kostenfunktion auf, der damit ebenfalls minimiert wird.

Wir geben dazu bei unserer *Layer*-Definition einen zusätzlichen Parameter an.

```
layer = keras.layers.Dense(breite,
                             activation=Aktivierungsfunktion,
                             kernel_regularizer=keras.regularizers.l2(parameter),
                             ...
                             )
```

Je größer der Parameter, desto mehr werden die Gewichte beschränkt. Im Beispiel wählen wir 0.001.

```
def make_model(input_shape, width=16):
    inp = keras.layers.Input(input_shape, name='Inp')
    l1 = keras.layers.Dense(
        width, activation='relu',
        kernel_regularizer=keras.regularizers.l2(0.001),
        name='L1'
    )(inp)
    l2 = keras.layers.Dense(
        width, activation='relu',
        kernel_regularizer=keras.regularizers.l2(0.001),
        name='L2')(l1)
    out = keras.layers.Dense(3, activation='softmax', name='out')(l2)

    model = keras.models.Model(inp, out)
    model.compile(
        loss=keras.losses.SparseCategoricalCrossentropy(),
        metrics=['accuracy']
    )
    return model
```

Beobachten wir den Trainingsprozess über 50 Epochen, sehen wir generell einen höheren Wert der Kostenfunktion, da nun die mittlere quadratische Summe der Gewichte mit eingeht, die natürlich nicht 0 sein kann (sonst wären alle Gewichte 0 und unser Netz könnte nichts lernen).

Wir können die *L2-Regularisierung* natürlich auch mit dem *Early-Stopping*-Verfahren verknüpfen, so dass unser Training abbricht, sobald keine wesentlichen Verbesserungen mehr erreicht werden.

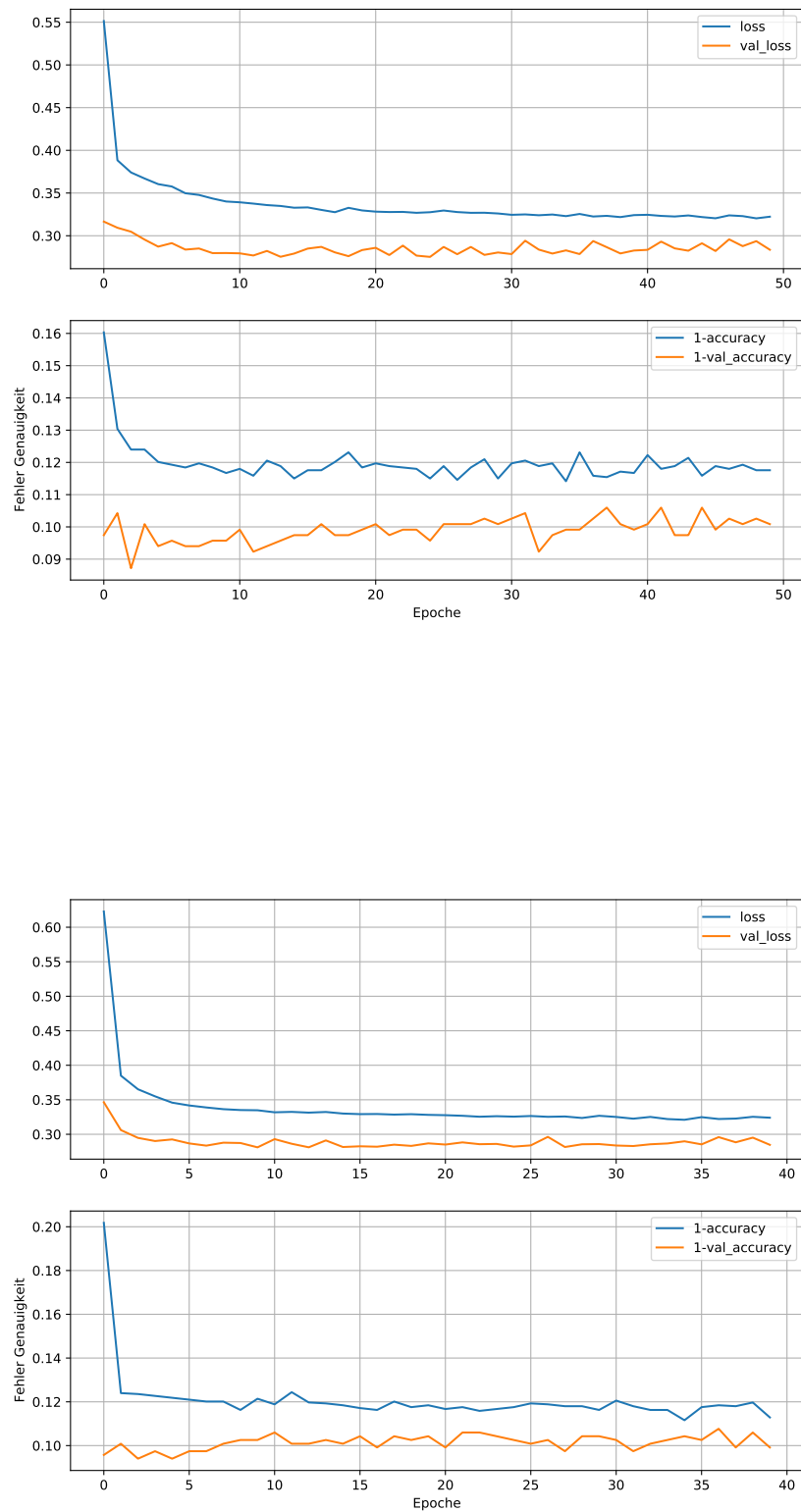


Abb. 19.4: Lernfortschritt mit L2-Regularisierung und Early Stopping.

Notebook

L2-Regularisierung ohne und mit Early Stopping: `abalone-l2reg.ipynb`.

19.5 Dropout

Eine weitere Form der Regularisierung übernimmt die Idee der Ensemble-Methoden der klassischen Lernverfahren, die wir beim *Random-Forest*-Verfahren kennengelernt haben: wir könnten einfach mehrere Netze parallel erstellen und trainieren und diese Netze gemeinsam eine Entscheidung per Voting treffen zu lassen.

Um die Erstellung das Training mehrerer Netze zu vermeiden, erstellen wir lediglich ein Netz (mit eventuell breiteren Schichten) und schalten in jedem Trainingsschritt zufällig einige Knoten in einem oder mehreren Schichten aus, so dass auf diese Weise jeweils ein (zufälliges) Teilnetz trainiert wird. Ist das Netz komplett trainiert, werden für die Vorhersage alle Knoten des Netzes verwendet, wir erhalten damit eine gewichtete gemeinsame Vorhersage aller Teilnetze.

Die Realisierung erfolgt durch Zwischenschalten von zusätzlichen Dropout-Layern nach bestimmten oder jedem normalen Layer. Dieser verhindert für zufällig ausgewählte Knoten die Weitergabe der berechneten Werte an den Folgelayer und im Optimierungsschritt die Anpassung der Gewichte dieser Knoten.

`dropout = keras.layers.dropout(rate)`

Die Dropout-Schicht kann unmittelbar nach der Eingabe (meist mit einer Rate von 0.05 bis 0.1) und hinter der Aktivierung (meist mit einer Rate von 0.3 bis 0.5) eingefügt werden.

Beispiel für den *Abalone*-Datensatz:

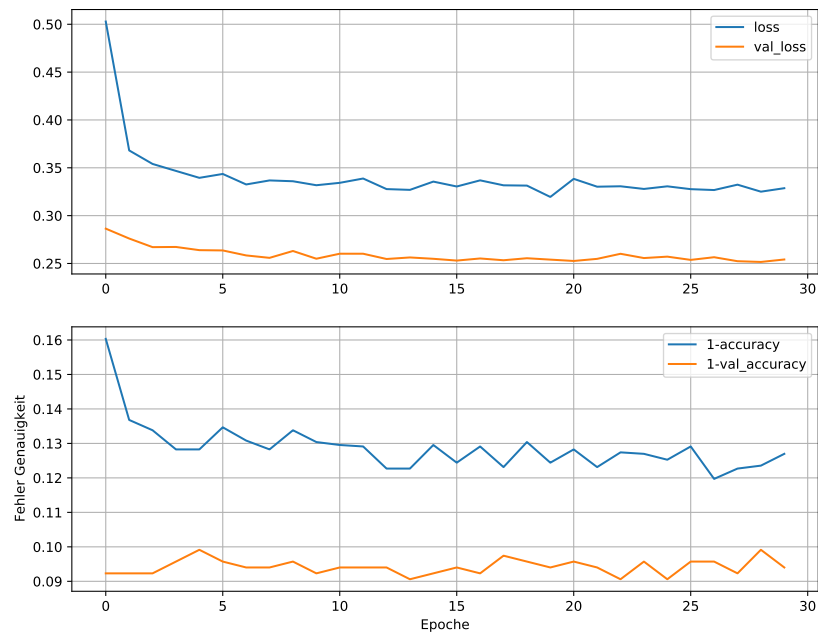
```
WIDTH = 48
LEARN_RATE = 5e-4

model = keras.models.Sequential()
model.add(keras.layers.Input(Xtrain.shape[1:], name='Inp'))
# Dropout für Eingabe
model.add(keras.layers.Dropout(0.05, name='Drop0'))
model.add(keras.layers.Dense(
    WIDTH, activation='relu', name='L1'
))
# Dropout nach 1. verborgener Schicht
model.add(keras.layers.Dropout(0.05, name='Drop1'))
model.add(keras.layers.Dense(
    WIDTH, activation='relu', name='L2'
))
# Dropout nach 2. verborgener Schicht
model.add(keras.layers.Dropout(0.05, name='Drop2'))
# Ausgabe
model.add(keras.layers.Dense(3, activation='softmax', name='out'))
# Trainingsschleife
model.compile(
    optimizer=keras.optimizers.RMSprop(LEARN_RATE),
    loss=keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)
```

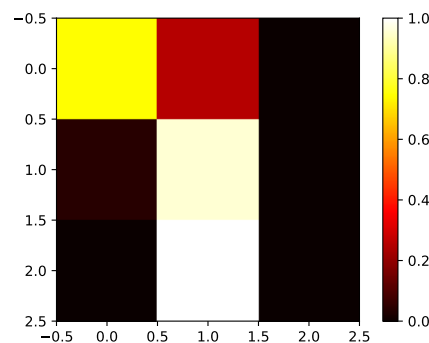
Mit einem Training über 30 Epochen und einer Minibatchgröße von 16 erhalten wir z. B. folgenden Trainingsfortschritt:

und folgendes Ergebnis:

```
[[225  78  0]
 [ 40 836  1]
 [  0  73  0]]
```



und als Graphik:



Zum Vergleich betrachten wir ein Netz mit zwei verborgenen Schichten der Breite 16.

Wir sehen hier deutlich das gleichmäßigere Abnehmen der Kostenfunktion und damit die immer bessere Anpassung an die Trainingsdaten.

Confusion Matrix:

```
[[241  62   0]
 [ 53 813  11]
 [  0  65   8]]
```

Auch beim Training dieses Netzes können wir zur weiteren Regularisierung das *Early-Stopping*-Verfahren verwenden. Ebenso bietet es sich an, die Lernrate nach den ersten Epochen zu verringern, um die starken Schwankungen der Kostenfunktion zu vermeiden (s. *Anpassen der Lernrate*).

In unserem Beispiel schließen wir nun zufällig 5% der Komponenten (also Knoten der Eingabeschicht) und 50% der verborgenen Knoten aus.

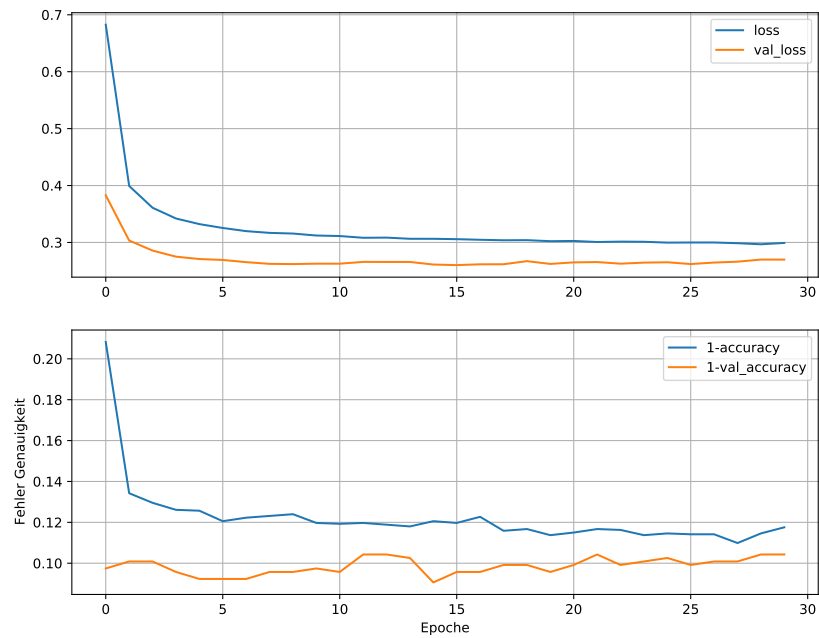
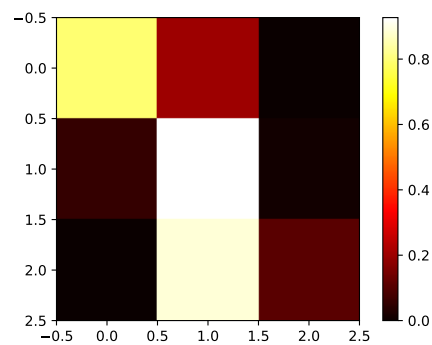


Abb. 19.5: Trainingsfortschritt des nichtregulierten Netzes.



```
inp = layers.Input(29)
d1 = layers.Dropout(0.05)(inp)
h2 = layers.Dense(100, activation='relu')(d1)
d3 = layers.Dropout(0.5)(h2)
out = layers.Dense(1, activation='sigmoid')(d3)

model4 = keras.Model(inp, out)
```

Hinweis: Die Methode des *Early Stopping* kann problemlos mit der *Dropout*- oder der *L2*-Regularisierung verknüpft werden. Von einer gleichzeitigen Nutzung von *Dropout* und *L2* wird in der Literatur dagegen abgeraten.

19.6 Batch-Normalisierung

Als Alternative der *Dropout*-Regularisierung wird gern die sogenannte *Batch-Normalisierung* verwendet. Die Aktivierungsfunktionen sind meist so konstruiert, dass sie die Daten mit einer statistischen Verteilung mit dem Erwartungswert 0 und einer Varianz von 1 voraussetzen, wie wir dies auch bei der Vorverarbeitung metrischer Daten schon genutzt haben.

Da wir die Ausgabeverteilung einer Schicht des Netzes nicht kennen, müssen wir diese Normalisierung ebenfalls während des Trainings lernen. Wir trennen dazu unsere bisherigen Schichten in zwei Teile, die gewichtete Summierung der Eingangsdaten und die Anwendung der Aktivierung, in zwei getrennte Schichten auf und schalten zwischen diese beiden die Normalisierungsschicht.

```
model = keras.models.Sequential([
    ...
    keras.layers.Dense(breite[...]),
    keras.layers.BatchNormalization(),
    .keras.layers.Activation(aktivierung),
    ...
])
```

Die *Dense*-Schicht enthält keine Aktivierungsfunktion, sondern leitet ihre Ausgabe direkt an die Normalisierungsschicht. Die Aktivierungsfunktion kann entweder als String ('relu', 'tanh', ...) oder als *Keras*-Funktion (`keras.activations.relu()`, `keras.activations.tanh()`, ...) angegeben werden.

Hinweis: Bei Verwendung der *ReLU*-Aktivierung kann die Batch-Normalisierung auch nach die Aktivierungsschicht verschoben werden, was jedoch zu einem anderen Modellverhalten führt, da die Normalisierung ja den Nullpunkt der Daten verschiebt und damit auch den Lernprozess beeinflusst (da ja die Ableitung für negative Werte Null ist und damit in diesem Fall nichts gelernt wird). Hier muss getestet werden, wann das bessere Lernverhalten erreicht wird.

Wir können auch eine Batch-Normalisierung als erste Schicht verwenden und damit die externe Skalierung unserer metrischen Komponenten weglassen. In diesem Fall müssen wir allerdings den Parameter `input_shape` in diese erste Schicht verschieben oder eine *Input*-Schicht mit definierter Breite voranstellen.

Beispiel: Netz für *Abalone* mit Batch-Normalisierung.

```
LAYER_WI = 20
model = keras.models.Sequential()
model.add(keras.layers.Input(Xtrain.shape[1:], name='Inp'))
model.add(keras.layers.Dense(LAYER_WI, name='L1'))
model.add(keras.layers.BatchNormalization(name='BN1'))
model.add(keras.layers.Activation('relu', name='ReLU1'))
model.add(keras.layers.Dense(LAYER_WI, name='L2'))
model.add(keras.layers.BatchNormalization(name='BN2'))
```

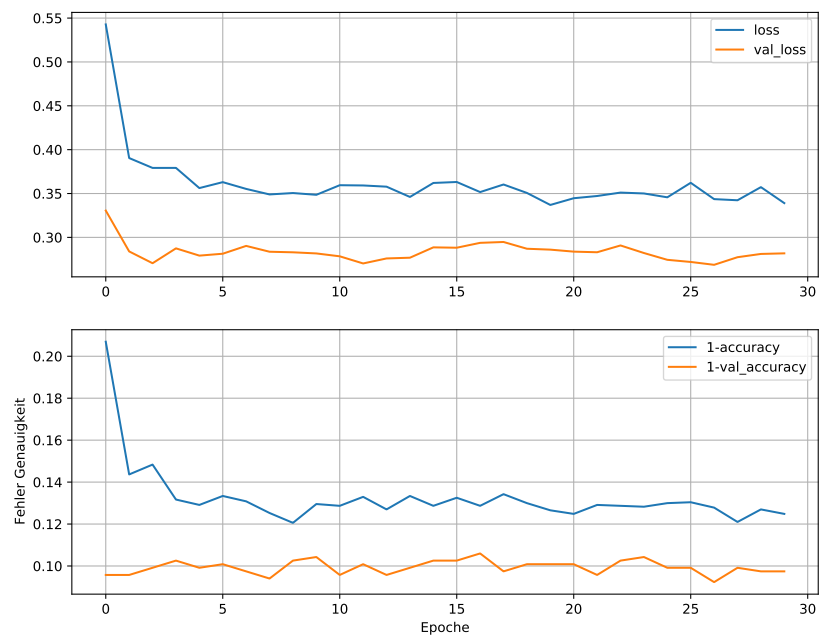
(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
model.add(keras.layers.Activation('relu', name='ReLU2'))
model.add(keras.layers.Dense(3, activation='softmax', name='out'))
```

Trainieren wir dieses Modell über 30 Epochen mit einer Minibatch-Größe von 8, erhalten wir folgendes Bild:

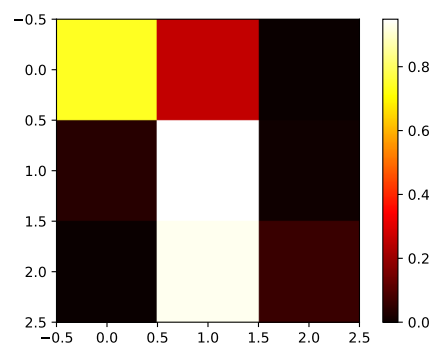
Training mit Batch Size 8



Unsere Confusion Matrix ist z. B.

```
[[224  79   0]
 [ 38 832   7]
 [  0  68   5]]
```

bzw. graphisch:

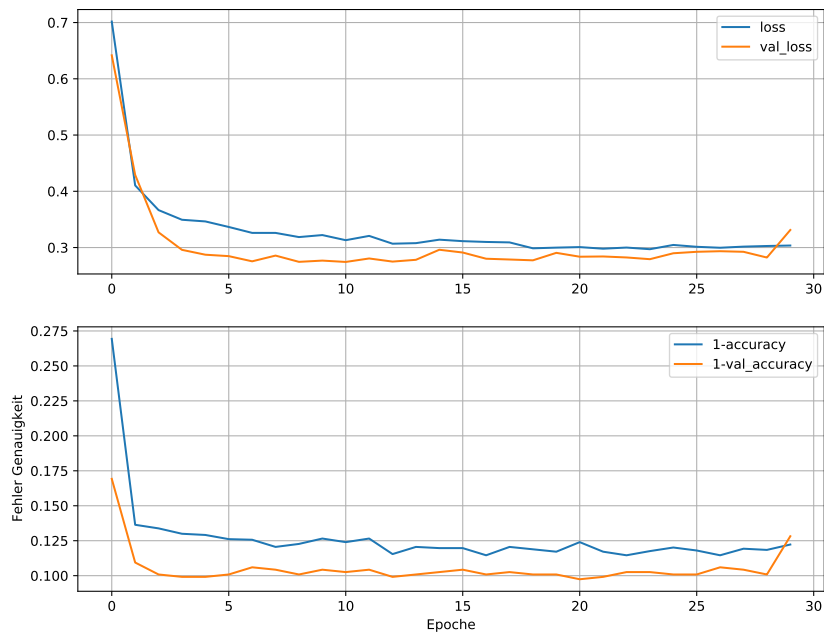


Mit einer Minibatch-Größe von 32 tritt ein stärkerer Regularisierungseffekt auf:

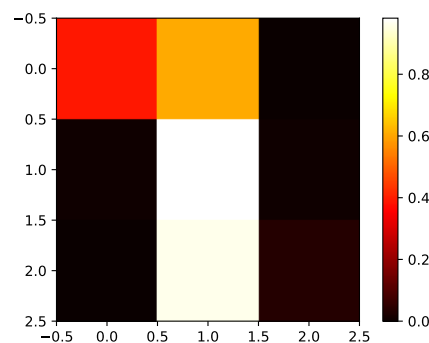
Unsere Confusion Matrix ist hier z. B.

```
[[119 184   0]
 [  8 861   8]
 [  0  70   3]]
```

Training mit Batch Size 32



bzw. graphisch:



Treten im weiteren Trainingsverlauf große Schwankungen der Kostenfunktion auf den Trainingsdaten auf, kann eventuell die Verringerung der Lernrate während des Trainings helfen. Ebenso ist eine Kombination mit *Early Stopping* möglich.

Hinweis: Es zeigt sich hier, dass die Batch-Normalisierung mit größeren Minibatches besser funktioniert.

Prinzipiell können Dropout und Batch-Normalisierung gemeinsam in einem Modell verwendet werden, die Literatur rät jedoch davon ab.

Notebook

Download: `abalone-batchnorm.ipynb`.

19.7 Anpassen der Lernrate

Wir haben mehrfach gesehen, dass während des Lernprozesses die Kostenfunktion zunächst sinkt, dann aber auf und ab springt. Dies ist häufig ein Zeichen einer zu hohen Lernrate: wenn wir ein Gewicht des Netzwerkes innerhalb einer ϵ -Umgebung entgegen seines Gradienten verringern, sollte sich die Kostenfunktion ebenfalls verringern. Wenn unsere Lernrate jedoch so groß ist, dass das Produkt mit dem Gradient diese sehr kleine Umgebung verlässt, dann gilt diese Aussage nicht mehr. Verringern wir aber die Lernrate zu stark, dann lernt unser Modell nur sehr langsam. Sinnvoller ist es deshalb oft, die Lernrate während des Trainingsprozesses anzupassen.

Wir definieren dazu eine *Scheduling*-Funktion, die als Parameter die Nummer der Epoche (gezählt ab 0) und die bisherige Lernrate erhält und daraus eine neue Lernrate bestimmt.

Beispiel: Wir wollen in den ersten 10 Epochen die Lernrate konstant lassen und anschließend in jeder Epoche um 0.5% verringern:

```
def scheduler(epoch, learn_rate):
    if epoch < 10:
        return learn_rate
    else:
        return learn_rate * 0.995
```

Diese Funktion wird mit Hilfe eines Callback-Objekts in den Lernprozess eingefügt:

```
learnrate_cb = keras.callbacks.LearningRateScheduler(
    schedule=scheduler,
    [verbose=1]
)
```

Dieses Objekt fügen wir in unsere `callbacks`-Liste des Trainings ein:

```
model.fit(
    ...
    callbacks=[callbacks],
    ...
)
```

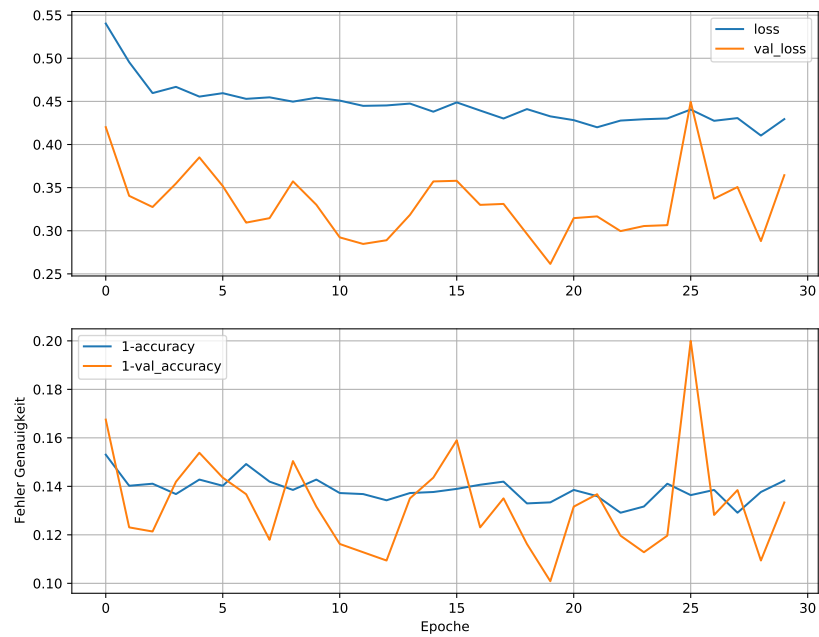
Beispiel: Wir starten wieder mit einem unreguliertem Netz mit zwei verborgenen Schichten mit der Breite von 32. Die Lernrate erhöhen wir auf $5 \cdot 10^{-3}$. Unser Training sieht wie folgt aus:

```
hist1 = model1.fit(
    Xtrain, yTrain,
    validation_split=0.2,
    epochs=30,
    batch_size=16,
    class_weight={0:1, 1:1, 2:4},
    verbose=2
)

model1.evaluate(Xtest, yTest)
```

Der Parameter `class=weight` wichtet die Datenpunkte der Klasse 2 stärker, da diese in der Trainingsmenge unterrepräsentiert sind. Wir erhalten zunächst eine gleichmäßige Verringerung der Kostenfunktion, die ab Epoche 15 jedoch zu schwingen beginnt.

Wir verringern deshalb nach 5 Epochen kontinuierlich die Lernrate:



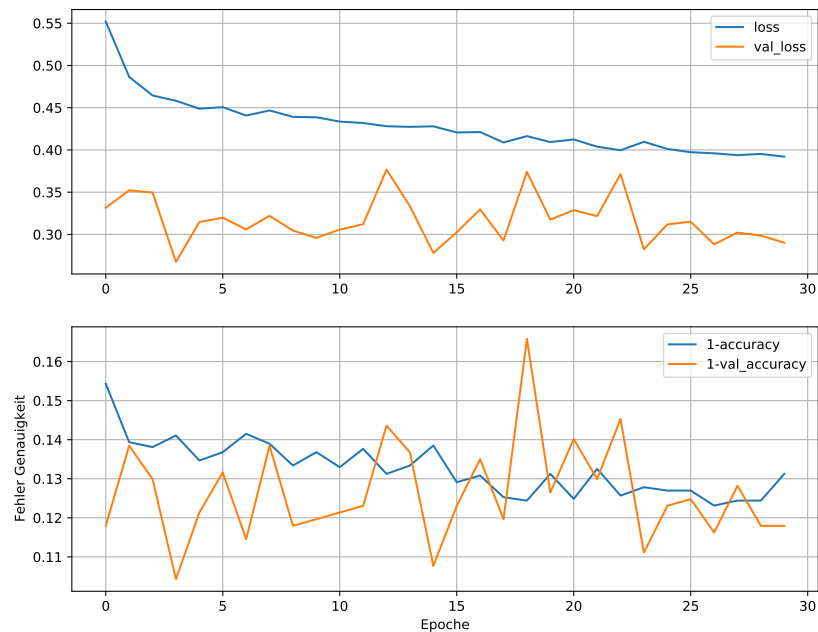
```
def lr_scheduler(epoch, lr):
    if epoch < 5:
        return lr
    else:
        return lr*0.95

lr_callback = keras.callbacks.LearningRateScheduler(
    lr_scheduler,
    verbose=1
)

hist2 = model1.fit(
    Xtrain, yTrain,
    validation_split=0.2,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=[lr_callback],
    class_weight={0:1, 1:1, 2:4},
    verbose=2
)
```

Die Kostenfunktion fällt auf den Trainingsdaten nun kontinuierlich ab, auch die Schwankungen auf den Validierungsdaten fallen nicht mehr so stark aus.

Wir erhalten nun z. B. folgende Confusion Matrix:



Quellcode 19.1: Confusion Matrix mit Anpassung Lernrate

```
[[257  46   0]
 [ 72 778  27]
 [  0  62  11]]
```

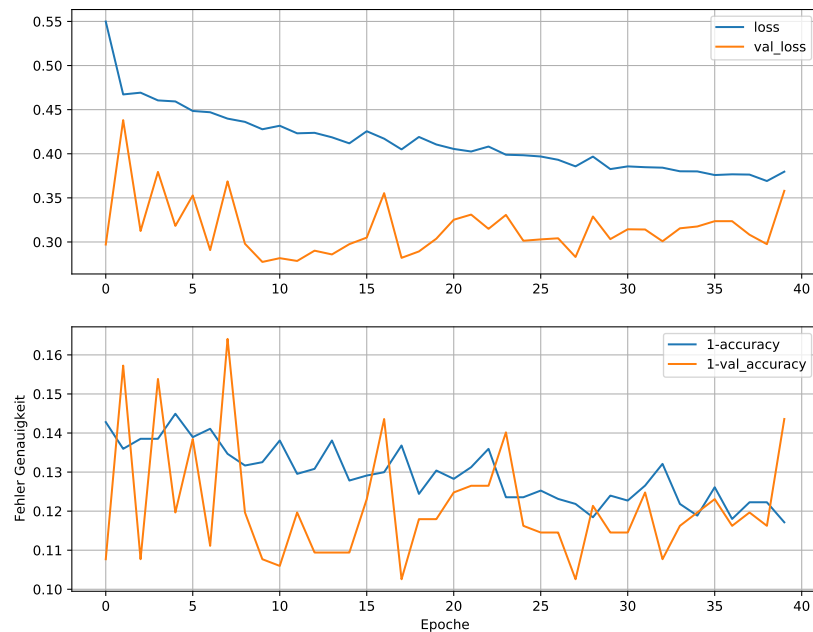
Der Trainingsverlauf deutet an, dass das Modell noch nicht austrainiert ist. Zum anderen besteht die Gefahr des Overfittings, da wir keine weitere Regularisierung vornehmen. Wir verwenden deshalb zusätzlich das *Early-Stopping*-Verfahren zum Beenden des Trainings.

```
es_callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=30,
    restore_best_weights=True,
    verbose=1
)

hist3 = model1.fit(
    Xtrain, yTrain,
    validation_split=0.2,
    epochs=200,
    batch_size=BATCH_SIZE,
    callbacks=[lr_callback, es_callback],
    class_weight={0:1, 1:1, 2:4},
    verbose=2
)
model1.evaluate(Xtest, yTest)
```

Wir sehen weiterhin einen gleichmäßigen Trainingsverlauf für die Kostenfunktion.

Eine deutliche Verbesserung der Vorhersagequalität ist jedoch nicht erkennbar.



Quellcode 19.2: Confusion Matrix mit Anpassung Lernrate und Early Stopping

```
[[228  75  0]
 [ 42 798 37]
 [  0  64  9]]
```

19.8 Aufgaben

Aufgabe 19.1 Hausaufgabe (10 Punkte)

Erzeugen Sie für Ihren Beispieldatensatz je ein *ANN* mit einer bzw. zwei verborgenen Schichten und trainieren Sie diese. Stellen Sie den Lernprozess graphisch dar, werten Sie ebenfalls die Klassifikationsgüte für die Testdaten mit einer Konfusionsmatrix aus.

Regularisieren Sie den Lernprozess durch:

- Verwendung einer Validierungsmenge,
- Hinzufügen von Dropout,
- Batch-Normalisierung (in einem zweiten Modell),
- Early Stopping.

Weitere Regularisierungen dürfen ebenfalls genutzt werden.

Erstellen Sie mindestens zwei Modelle mit unterschiedlichen Regularisierungsstrategien (und damit unterschiedlicher Topologie).

Wenn Sie für das gleiche Modell unterschiedliche Lernstrategien untersuchen, starten Sie jeweils mit den gleichen Gewichten.

Versuchen Sie, ein besseres Modell als mit Ihrem besten klassischen Lernverfahren zu erzielen (und seien Sie nicht enttäuscht, wenn Ihnen das nicht gelingt).

Convolutional Neuronal Networks

- *Motivation*
- *Convolution (Faltung)*
- *Pooling*
- *Beispieldaten*
- *Ein einfaches CNN*
- *Verbessertes Modell*
- *Nutzung für die Vorhersage*
- *Aufgaben*

20.1 Motivation

Wenn wir mit unseren bisherigen *ANN* Bilder analysieren wollten, mussten wir diese in einen eindimensionalen Vektor umwandeln, indem wir alle Bildzeilen hintereinander hängten. Handelt es sich um ein mehrfarbiges Bild, müssen wir das Bild entweder vorher in Graustufen umrechnen oder die Vektoren für alle Farbkkanäle hintereinander in den Vektor einfügen. Dabei gehen uns aber zum einen sämtliche Nachbarschaftsinformationen verloren: wird das Bild ein wenig verschoben oder gedreht, entsteht ein völlig veränderter Eingabevektor, und es wird sehr schwer, die Ähnlichkeit zum ersten Vektor zu erkennen.

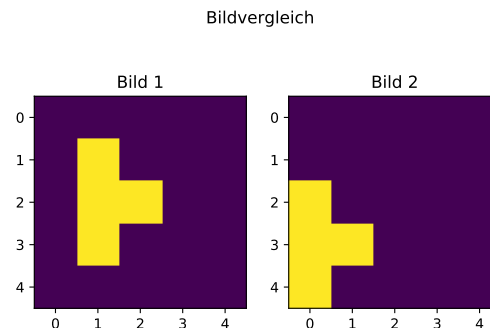
Beispiel: Erzeugen eines einfachen Bildes und einer Verschiebung.

```
Bild 1
[[0 0 0 0 0]
 [0 1 0 0 0]
 [0 1 1 0 0]
 [0 1 0 0 0]
 [0 0 0 0 0]]
Bild 2
[[0 0 0 0 0]
 [0 0 0 0 0]
 [1 0 0 0 0]]
```

(Fortsetzung auf der nächsten Seite)

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

Darstellung als Graphik:



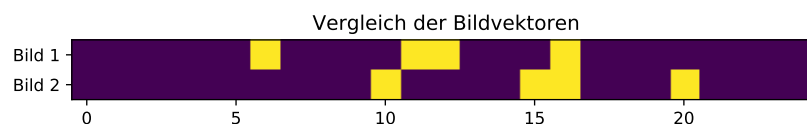
Wandeln wir beide Bilder in einen Vektor um:

```
bildVektor = bild.reshape(-1)
```

und vergleichen beide Vektoren, unterscheiden sich diese stark:

```
Vektor Bild 1: [0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0]
Vektor Bild 2: [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0]
```

Graphische Darstellung (jedes Bild ist eine Zeile).



Die Ähnlichkeit beider Bilder ist in dieser Darstellung komplett verlorengegangen, unser Modell kann diese kaum erlernen.

Zum anderen wird das Netz sehr groß. So benötigt ein Schwarzweißbild der Größe 100×100 bereits 10000 Eingangs-knoten. Mit einer einzigen verdeckten Schicht von 100 Knoten benötigen wir bereit 1 Million Gewichte.

Als Beispiel betrachten wir ein klassisches Problem der Erkennung handgeschriebener Ziffern (<http://yann.lecun.com/exdb/mnist/>). Dieser Datensatz umfasst 60.000 Trainings- und 10.000 Testdaten und ist als Beispieldatensatz in der *TensorFlow*-Bibliothek enthalten.

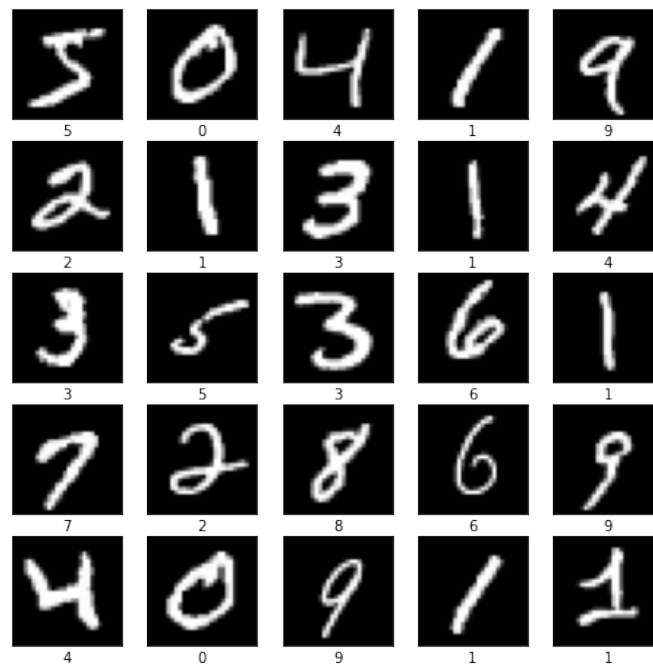
Wir können die Daten einfach laden:

```
(X2train, yTrain), (X2test, yTest) = keras.datasets.mnist.load_data()
```

Die Bilder sind jeweils 28×28 = 784 Pixel groß und enthalten in jedem Bildpunkt die Helligkeit als Wert zwischen 0 und 255. Einige Beispiele des Datensatzes sehen wir hier:

Wir skalieren die Bilder auf das Intervall [0, 1] um und stellen sie als Vektor der Länge 784 dar:

```
Xtrain = X2train.reshape(-1, 28*28)/255.0
Xtest = X2test.reshape(-1, 28*28)/255.0
```



Nun können wir ein einfaches Netz erzeugen. Wir wählen eine verdeckte Schicht mit 300 Knoten, keine Regularisierung und eine Ausgabeschicht mit 10 Knoten und `softmax`-Aktivierung.

```
model = keras.models.Sequential()
model.add(keras.layers.Input(28*28))
model.add(keras.layers.Dense(300, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))
```

Wir erhalten die Netztopologie:

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_2 (Dense)	(None, 300)	235500
dense_3 (Dense)	(None, 10)	3010
Total params: 238,510		
Trainable params: 238,510		
Non-trainable params: 0		

Ein Training über 30 Epochen mit einer Minibatch-Größe von 128 liefert auf dem Testdatensatz eine Genauigkeit von 98%. Dieses Ergebnis ist schon recht gut, wenn wir aber bedenken, dass Postleitzahlen aus fünf Ziffern bestehen, sinkt die Wahrscheinlichkeit für die korrekte Erkennung der gesamten Zahl auf

$$0.98^5 = 0.90$$

10% der Briefe würden falsch zugestellt. Trotzdem ist dieses Ergebnis deutlich besser als die auf obiger Webseite gelisteten Genauigkeiten für einfache neuronale Netze, was vermutlich durch die besser geeignete Aktivierungsfunktion und bessere Optimierer erreicht wurde, als sie zum Zeitpunkt der zitierten Tests zur Verfügung standen.

Die Idee der *Convolutional Neural Networks (CNN)* ist nun, die Nachbarschaften in Bildern auszunutzen. Aus der Bildverarbeitung ist die Technik der *Filter* bekannt: zu jedem Bildpunkt eines Bildes werden die benachbarten Pixel in alle Richtungen bis zu einer bestimmten Entfernung betrachtet, mit geeigneten Gewichten multipliziert und anschließend addiert. Aus dem Ergebnis entsteht ein neues Bild. Diese Operation nennt man *Faltung* (englisch *Convolution*).

Notebooks

- Demo Bildverschiebung: `img-shift-flatten.ipynb`,
- *MNIST* Digits: `minst-ffn.ipynb`.

20.2 Convolution (Faltung)

Für eine zweidimensionale Faltung lassen wir eine Faltungsmatrix schrittweise über eine andere (meist größere) Matrix, z. B. unser Bild, wandern. Jeder Wert des Fensters wird mit dem Wert der Faltungsmatrix multipliziert, dann wird die Summe gebildet und diese an die nächste Position der Ergebnismatrix geschrieben. Hat unsere Faltungsmatrix z. B. die Dimensionen 3×3 , dann ist unsere Ergebnismatrix je zwei Einheiten in Breite und Höhe kleiner als die Ausgangsmatrix.

Beispiel: Sobel-Filter zur Erkennung/Verstärkung senkrechter Kanten in Bildern.

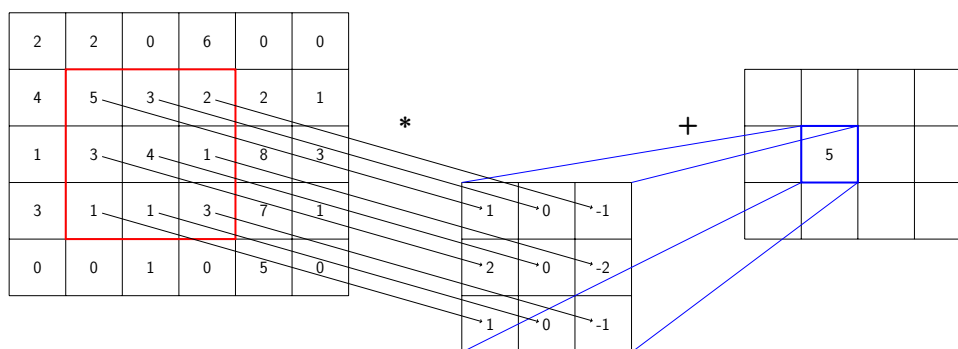


Abb. 20.1: Schematische Darstellung der Convolution: wir rechnen:

$$\begin{aligned}
 &5 \cdot 1 + 3 \cdot 0 + 2 \cdot -1 \\
 &+ 3 \cdot 2 + 4 \cdot 0 + 1 \cdot -2 \\
 &+ 1 \cdot 1 + 1 \cdot 0 + 3 \cdot -1 \\
 &= 5
 \end{aligned}$$

Komplette Convolution:

```

Filter:
[[ 1  0 -1]
 [ 2  0 -2]
 [ 1  0 -1]]
Originalmatrix:
[[2 2 0 6 0 0]
 [4 5 3 2 2 1]
 [1 3 4 1 8 3]
 [3 1 1 3 7 1]
 [0 0 1 0 5 0]]
Nach der Convolution:
[[ 1  4 -2  6]
 [-3  5 -13 -1]
 [ 0 -2 -20  2]]

```

Hinweis: Meist wird die Convolution-Matrix vor der Anwendung an der horizontalen und vertikalen Achse gespiegelt. Das hat theoretische Gründe, spielt aber für das Verständnis keine Rolle.

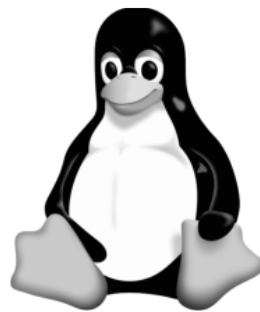
Beispiel: Wirkung des Sobel-Filters für waagerechte und senkrechte Kantenerkennung bei einem Bild.

Wir verwenden einen kleinen Tux `tux.png`. Dieses Bild können wir mit der *Matplotlib*-Bibliothek einlesen

```
img = plt.imread('tux.png')
```

Wir erhalten einen dreidimensionalen Tensor mit den Dimensionen 500×596×3. Die dritte Achse beschreibt die Farbanteile Rot, Grün und Blau. Wir wandeln das Bild zunächst in ein Graustufenbild um und erhalten eine normale Matrix:

```
bwimg = 0.2126 * img[:, :, 0] + 0.7152 * img[:, :, 1] + 0.0722 * img[:, :, 2]
```



Auf dieses Bild wenden wir nun die Sobel-Filter für senkrechte und waagerechte Kantenerkennung an:

```
import scipy.signal

sobelX = np.array(
    [1, 0, -1,
     2, 0, -2,
     1, 0, -1]
).reshape(3, 3)
sobelY = sobelX.T

trX = scipy.signal.convolve2d(bwimg, sobelX)
trY = scipy.signal.convolve2d(bwimg, sobelY)
```

Zeigen wir uns die Bilder an, sehen wir die Betonung der senkrechten bzw. waagerechten Kanten als dunkle bzw. helle Linien.

Die Grundidee der *CNN* ist nun, eine Vielzahl derartiger Filter automatisch zu erlernen, die jeder bestimmte Bildeigenschaften wie Kanten, Formen, Helligkeiten und Farben erkennen. Nach diesen Transformationen, die wir meist mehrfach hintereinander verketteten, schließen wir ein klassisches neuronales Netz an, das die Ergebnisse der Einzeltransformationen zusammenfasst und daraus die Zielgröße (Klasse des erkannten Objekts, Lage, Vergrößerung, ...) ermittelt.

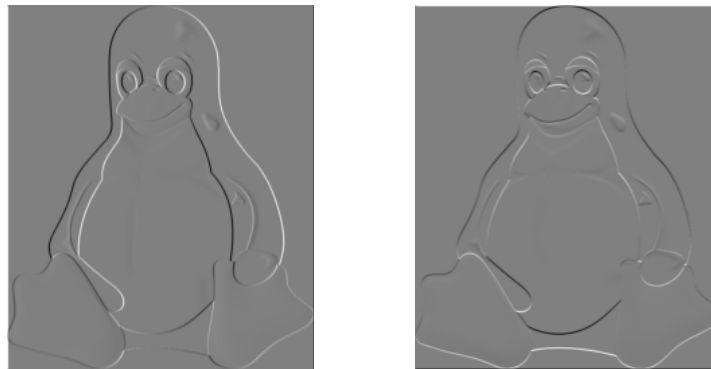


Abb. 20.2: Anwendung des Sobel-Filters für senkrechte (links) und waagerechte Kanten (rechts).

20.3 Pooling

Um die Daten zu verdichten und gegenüber kleinen Bildverschiebungen invariant zu sein, fügt man hinter den Convolution-Schichten sogenannte *MaxPool*-Schichten ein: Unser Bild wird in kleine Kacheln, z. B. mit der Größe 2×2 , zerlegt und von jeder Kachel wird nur der größte Wert übernommen. Dies verringert gleichzeitig die weiter zu verarbeitende Bildgröße (bei 2×2 -Kacheln auf $1/4$).

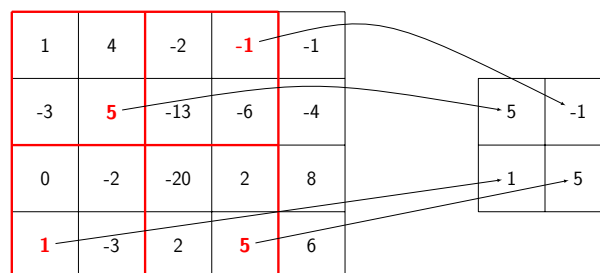


Abb. 20.3: Funktionsweise des Max-Poolings.

Statt der Wahl des Maximums kann man alternativ den Durchschnitt der Kachelwerte verwenden, was jedoch höheren Rechenaufwand bedeutet und meist schlechter funktioniert.

Wir dürfen zwischen den Kacheln auch Lücken lassen oder ein Überlappen der Kacheln erlauben. Der Wert der Verschiebung der Kachel in jedem Schritt wird als *Stride* bezeichnet. Meist wählt man diese Verschiebung jedoch genauso groß wie die Kachelgröße, wir decken das Original also komplett ohne Lücken ab. Kann der Bildrand nicht mehr vollständig durch eine Kachel überdeckt werden, wird er meist ignoriert.

20.4 Beispieldaten

Zum Erlernen der Technologie und zum Testen stellen *Tensorflow* und *Keras* einige Beispieldatensätze zur Verfügung, für die eine bequeme Schnittstelle zum Laden bereitgestellt wird.

```
(Xtrain, yTrain), (Xtest, yTest) = keras.datasets.dataSet.load_data(...)
```

Die Zahl der Dimensionen der Trainings- und Testdaten unterscheidet sich bei den Beispielen und sollte unbedingt geprüft werden. So werden in *CIFAR10* die Label als $n \times 1$ -Matrix, in *MNIST* dagegen als Vektor bereitgestellt.

Folgende Datensätze für die Bildbearbeitung finden wir in Keras:

mnist Handgeschriebene Ziffern. Bildgröße 28×28 .

fashion_mnist Kleidungsstücke in 10 Kategorien. Bildgröße 28×28. Verwendung identisch zu den *MNIST*-Ziffern.

cifar10 Farbige Bilder der Größe 32×32 in 10 Kategorien.

cifar100 Farbige Bilder der Größe 32×32 in 100 Kategorien.

Die Bilder müssen vor dem Lernen üblicherweise noch auf das Intervall $[0, 1]$ umskaliert werden. Die Label sind teilweise als Matrix und nicht als Vektor organisiert und müssen je nach geplanter Nutzung neu angeordnet werden. Beispiel:

```
(XoriTrain, yOriTrain), (Xoritest, yOriTest) = keras.datasets.mnist.load_data()
Xtrain = XoriTrain/255.0
Xtest = Xoritest/255.0
yTrain = yOriTrain.reshape(-1) # für CIFAR10
yTest = yOriTest.reshape(-1)
```

Wir benötigen für unsere Trainings- und Testdaten einen 4-dimensionalen Tensor mit den Dimensionen:

- Index des Bildes,
- Breite,
- Höhe,
- Kanal (Farbkomponente).

Liegen die Bilder nur mit zwei Dimensionen (wie beim *MNIST*-Datensatz), müssen wir die Komponente für den Kanal hinzufügen. Ebenso sollte der Datentyp von `float64` auf `float32` umkonvertiert werden, um Speicherplatz zu sparen.

```
shp = XoriTrain.shape
Xtrain = XoriTrain.astype('float32').reshape(shp[0], shp[1], shp[2], 1)/255.0
Xtest = Xoritest.astype('float32').reshape(shp[0], shp[1], shp[2], 1)/255.0
```

Neben *Keras* stellt auch *Tensorflow* eine Vielzahl von Datensätzen zur Verfügung. Diese werden in einem eigenen Paket `tensorflow.data.Datasets` bereitgestellt. Dies sind im Gegensatz zu den *Keras*-Beispielen deutlich größere und wirklichkeitsnähere Datensammlungen, die als Ausgangspunkt für produktive Anwendungen dienen können. Entsprechend komplizierter ist jedoch auch die Arbeit mit diesen Daten.

Einige Beispiele aus dem Bereich der Bilderkennung sind:

bigearthnet 500.00 Fotoausschnitte der Erdoberfläche zur Landnutzung.

caltech101 Allgemeine Fotos mit 200 bis 300 Pixeln Größe aus 101 Klassen.

caltech_birds2011 17.000 Fotos von 200 Vogelarten, zusätzlich mit Bounding Box und Segmentierungsmaske.

imagenet_v2 Auswahl aus dem *ImageNet*-Datensatz. Die Bilder sind verschieden groß.

quickdraw 50 Millionen symbolische Zeichnungen aus 345 Kategorien aus dem Spiel *Quick, Draw!*. Größe der Bilder 28×28.

20.5 Ein einfaches CNN

Normalerweise haben unsere Bilder 3 Dimensionen: Breite, Höhe und Farbkanäle (inclusive Transparenz). Unsere Eingabe ist also 4-dimensional, da wir normalerweise eine Batchverarbeitung nutzen:

- Bildnummer
- Breite
- Höhe
- Farb-/Transparenzkanal

Nachdem das Bild die erste Convolution-Schicht durchlaufen hat, stellt die dritte Bildachse (= 4. Achse der Eingabedaten) nicht mehr die Farbe, sondern den Filterwert des jeweiligen Filters dar.

Eine Convolution-Schicht für derartige Bilder wird wie folgt definiert:

```
conv = keras.layers.Conv2D (
    filterZahl,
    kernel_size=(breite, höhe) ,
    [activation=aktivierungsfunktion],
    ...
)
```

Wir müssen unbedingt die Anzahl der gewünschten Filter (= Breite der Convolution-Schicht) sowie die Größe des gewünschten Filters (`kernel_size`) angeben. Als Aktivierung bietet sich wiederum die `relu`-Aktivierung an. Folgt anschließend eine MaxPool-Schicht, kann sie meist auch entfallen, da die Folgeschicht für die Nichtlinearität sorgt.

Weitere häufig genutzte Parameter sind:

padding='same'/'valid' Verhalten am Rand. Beim Wert `valid` (Vorgabe) überdeckt der Filter stets komplett den Bildbereich, die Randpunkte, die vom Filter nicht komplett überdeckt werden können, werden nicht transformiert. Das Ergebnis ist also ein kleineres Bild. Beim Wert `valid` wird das Bild mit einem Rand von Nullen umgeben, so dass der Filter jeden Bildpunkt in das Ergebnis übernimmt. Das Ergebnis ist also genauso groß wie das Original.

strides=(dx, dy) Verschiebung des Filters nach jedem Schritt. Statt über jeden Bildpunkt zu wandern, können auch Punkte übersprungen werden, um die Bildgröße zu verringern.

kernel_regularizer=regularisierungsObjekt Die Größe der gelernten Gewichte des Filters kann zur Regularisierung begrenzt werden (L2-, L1- oder andere Regularisierung). Dazu geht die Summe aller Gewichte mit einem Straffaktor in die Kostenfunktion ein.

Eine MaxPool-Schicht benötigt mindestens 2 Parameter, die Kachelgröße (`pool_size`) und der Abstand der Kacheln (`strides`). Wir können diese unmittelbar aneinander anschließen, diese überlappen lassen oder auch Abstände zwischen den Kacheln lassen.

```
pool = keras.layers.MaxPool2D (
    pool_size=(breite, höhe) ,
    strides=(xVerschiebung, yVerschiebung) ,
    ...
)
```

Nach einer oder mehreren Convolution-Schichten benötigen wir üblicherweise ein klassisches [ANN](#), das pro Datenpunkt einen eindimensionalen Vektor als Eingabe erwartet. Wir müssen deshalb die 3-dimensionale Ausgabe entsprechend umwandeln. Dazu dient eine `keras.layers.Flatten`-Schicht.

```
flatten = keras.layers.Flatten()
```

Beispiel für ein einfaches [CNN](#) für die [MNIST](#)-Ziffern. Wir schließen an eine Convolution- und MaxPool-Schicht sofort unsere Ausgabeschicht mit 10 Knoten und Softmax-Aktivierung an.

```
model = keras.models.Sequential()
model.add(keras.layers.Input((28, 28, 1)))
model.add(keras.layers.Conv2D(25, kernel_size=(3, 3)))
model.add(keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2, 2)))
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation='softmax'))
```

Unser Modell sieht wie folgt aus:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
--------------	--------------	---------

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

conv2d (Conv2D)	(None, 26, 26, 25)	250
max_pooling2d (MaxPooling2D)	(None, 13, 13, 25)	0
flatten (Flatten)	(None, 4225)	0
dense (Dense)	(None, 10)	42260
Total params: 42,510		
Trainable params: 42,510		
Non-trainable params: 0		

Die Parameterzahl von 250 der Convolution-Schicht ergibt sich aus den 3×3 -Filterparametern und einem Bias-Wert, der am Ende aufaddiert wird. Die Ausgabeschicht benötigt für jeden der $13 \times 13 \times 25 = 4225$ Bildpunkte ein Gewicht für jeden der 10 Knoten sowie 10 Bias-Werte.

Die Parameterzahl ist also nur etwa 1/6 so groß wie die unseres vollständig verbundenen Netzes.

Trainieren wir dieses Netz 30 Epochen mit einer Minibatch-Größe von 128, erhalten wir eine Genauigkeit von ebenfalls 0.98 auf den Testdaten.

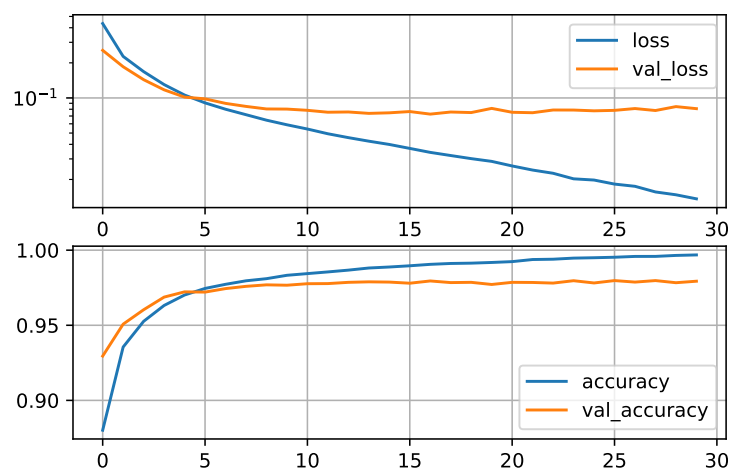


Abb. 20.4: Trainingsfortschritt des CNN.

Notebook

Download: `mnist-cnn-min.ipynb`.

20.6 Verbessertes Modell

Durch weitere Convolution- und vollständig verbundene verborgene Schichten lässt sich das Modell weiter verbessern. Wir orientieren uns an einem der ersten erfolgreichen Netze *LeNet-5* ([Wikipedia](#)) mit einigen kleinen Modifikationen:

- Convolution der Größe 5×5 und der Breite 6 (also 6 parallele Filter),
- Max-Pooling der Größe 2×2,
- Dropout,
- Convolution der Größe 3×3 und der Breite 16,
- Max-Pooling der Größe 2×2,
- Umwandlung aller 16 Bilder in einen Vektor,
- vollständig verbundene Schicht der Breite 120,
- Dropout,
- vollständig verbundene Schicht der Breite 84,
- Ausgabeschicht mit 10 Knoten (= Klassen).

Das Original enthielt keine Dropout-Schichten und weniger Verbindungen zwischen den Schichten. Für die Aktivierung wurde `tanh` verwendet, wir nutzen besser `relu`.

```
def make_model():
    model = keras.models.Sequential()
    model.add(keras.layers.Input((28, 28, 1)))
    model.add(keras.layers.Conv2D(
        6, kernel_size=(5, 5)))
    model.add(keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(keras.layers.Dropout(0.3))
    model.add(keras.layers.Conv2D(
        16, kernel_size=(3, 3)))
    model.add(keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2, 2)))
    model.add(keras.layers.Flatten())
    model.add(keras.layers.Dense(120, activation='relu')) # Ori: 120
    model.add(keras.layers.Dropout(0.3))
    model.add(keras.layers.Dense(84, activation='relu'))
    model.add(keras.layers.Dense(10, activation='softmax'))
    return model
```

Dieses Netz hat folgende Topologie:

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 6)	156
max_pooling2d (MaxPooling2D)	(None, 12, 12, 6)	0
dropout (Dropout)	(None, 12, 12, 6)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	880
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 16)	0

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 120)	48120
dropout_1 (Dropout)	(None, 120)	0
dense_1 (Dense)	(None, 84)	10164
dense_2 (Dense)	(None, 10)	850
=====		
Total params: 60,170		
Trainable params: 60,170		
Non-trainable params: 0		

Trainieren wir dieses Netz über 30 Epochen (*Adam*-Optimierer mit Vorgabewerten, Validierungsmenge: 20%), erhalten wir eine Genauigkeit von etwa 99% auf den Testdaten. Mit Anpassungen der Lernrate und einem längeren Training lässt sich das Modell weiter verbessern.

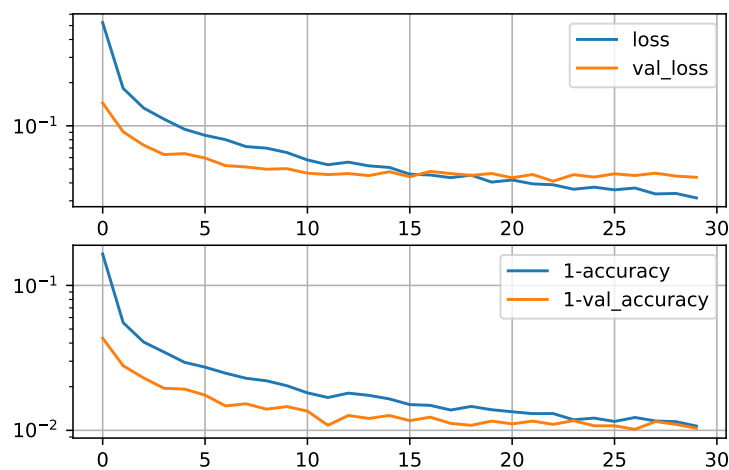


Abb. 20.5: Lernfortschritt des verbesserten Modells.

Neuere Netze enthalten wesentlich mehr Convolution-/Max-Pooling-Schichten, sind dann allerdings auch schwerer zu trainieren.

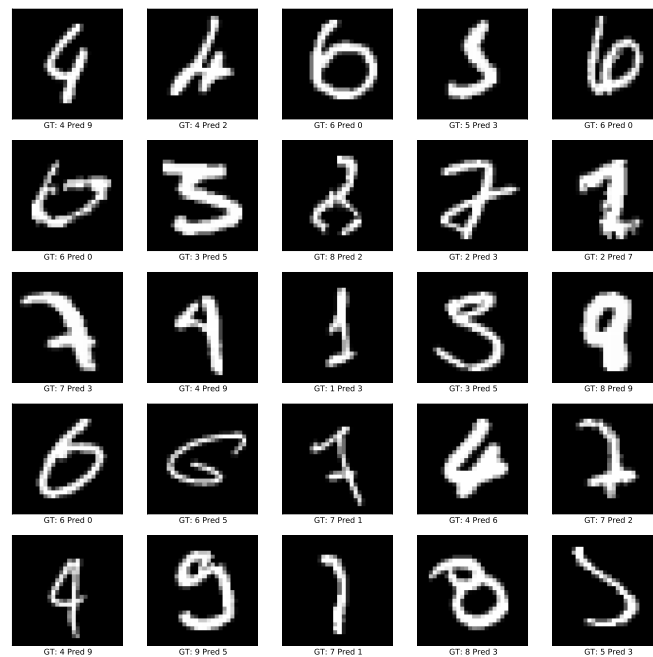


Abb. 20.6: Einige falsch vorhergesagte Ziffern.

Notebook

Download: `mnist-lenet-drop.ipynb`.

20.7 Nutzung für die Vorhersage

Sowohl *Keras* als auch *TensorFlow* stellen zahlreiche Objekte für das Einlesen und Vorverarbeiten von Daten zur Verfügung. Alternativ können wir auch entsprechende Werkzeuge aus *Matplotlib* oder *Pillow* verwenden.

Die Eingabebilder müssen auf die Größe und Farbtiefe umkonvertiert werden, die unser Modell erwartet. Mittels *Keras* können wir eine Bilddatei wie folgt lesen und skalieren:

```
img = keras.preprocessing.image.load_image(
    dateiPfad,
    color_mode='farbmode',
    target_size=(breite, höhe),
    ...
)
```

Aus diesem Bildobjekt können wir die Bilddaten als *NumPy*-Array auslesen:

```
imgData = keras.preprocessing.image.img_to_array(img)
```

Anschließend müssen eventuell noch der Datentyp angepasst und die Farbwerte auf das Intervall $[0, 1]$ skaliert werden.

Beispiel: Lesen aller Bilder eines Verzeichnisses als Graustufenbilder der Größe 28×28.

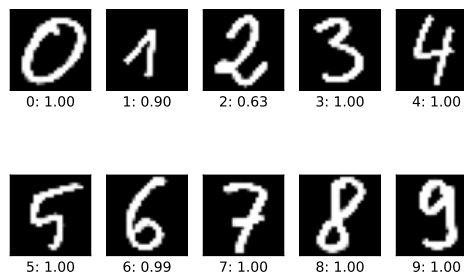


Abb. 20.7: Vorhersage unseres Netzes auf neuen Daten, jeweils Label und Wahrscheinlichkeit.

```
import os

def read_images(img_dir):
    count = len(os.listdir(img_dir)):
    # count Bilder der Größe 28x28, 1 Farbkanal
    images = np.zeros(count, 28, 28, 1)
    for i, fn in enumerate(os.listdir(img_dir)):
        path = os.path.join(img_dir, fn)
        img = keras.preprocessing.image.load_img(
            path, color_mode='grayscale', target_size=(28,28)
        )
        images[i] = img.astype('float32')/255.0
    return images
```

Wir sollten vor der Vorhersage Shape und Datentyp unserer Daten prüfen, dann erfolgt diese wie bei unseren Testdaten:

```
probabilities = model.predict(images)
classes = np.argmax(probabilities, axis=1)
```

Für einige selbstgeschriebene Ziffern erhalten wir z. B. folgende Vorhersage:

Notebook

Notebook zum Laden des trainierten LeNet-Modells und Vorhersage von selbstgeschriebenen Ziffern (im Ordner `mnist-pred`). Die Bilder wurden vorher invertiert und in Graustufen umgewandelt.

- Notebook: `mnist-lenet-predict.ipynb`,
- Daten: `mnist-predict-imgs.zip`.

20.8 Aufgaben

Aufgabe 20.1 LeNet-Regularisierung

Was passiert, wenn wir in unserem LeNet-ähnlichen Modell die Dropout-Schichten weglassen? Können wir stattdessen auch Batch-Normalisierung oder L2-Regularisierung verwenden?

Recurrent Neural Networks

- *Motivation*
- *Vorbereiten der Daten*
- *Basismodelle*
- *Recurrent Neural Networks*
- *Vorhersage mehrerer Werte*
- *Vorhersage für jeden Zeitschritt*

21.1 Motivation

Häufig wollen wir Folgen aufeinanderfolgender Daten untersuchen, wie z. B.

- Zeitreihen,
- Texte (z. B. Artikelbewertungen),

um z. B. Vorhersagen (Wetter, Aktienkurse) machen zu können, Anomalien zu erkennen, den Inhalt des Textes zu verstehen (gute oder schlechte Bewertung, Klassifikation der Aussage) oder auch verarbeiten zu können (automatische Antworten, Übersetzung in andere Sprachen).

Hier müssen wir folgende Eigenschaften von Folgen berücksichtigen:

- Ein zu untersuchendes Datum ist eine komplette Folge dieser Art.
- Die Daten einer Folge haben eine Ordnung und stehen miteinander in Zusammenhang.
- Die Reihenfolge der Daten in der Folge ist wichtig.
- Die Länge der Folgen können sich unterscheiden.

Wir wollen uns an dieser Stelle zunächst mit Zeitreihen, also Folgen von Zahlen, beschäftigen.

21.2 Vorbereiten der Daten

Wir wählen als Beispiel den Datensatz [Household Electric Power Consumption](#) von *Kaggle* (lokale Kopie: `household_power_consumption.zip`), der den Energieverbrauch zu ca. 2 Millionen Zeitpunkten enthält. Diesen teilen wir in einzelne Zeitabschnitte auf und versuchen, den Verbrauch zum nächsten Zeitpunkt vorherzusagen.

```
data = pd.read_csv('household_power_consumption.zip', sep=';',
    na_values=['', '?'])
```

Wir beschränken uns auf die Spalte `Global_active_power` und entfernen alle Fehlwerte.

```
data = data.dropna(subset=['Global_active_power'])
data0 = data[['Date', 'Time', 'Global_active_power']].copy()
```

Um mit den Zeitstempeln vernünftig arbeiten zu können, wandeln wir sie in einen Zeit-Datentyp um. Dazu verknüpfen wir die beiden Strings für Datum und Zeit und verwenden die Konvertierungsmethode `pd.to_datetime()` (*Achtung: zeitaufwendig*).

```
data0['date_time'] = pd.to_datetime(data0['Date'] + ' ' + data0['Time'])
```

Nun können wir die Daten nach dem Zeitstempel sortieren (vermutlich nicht erforderlich, da die Daten sicher bereits geordnet vorliegen). Der alte Index wird verworfen, da er keine weitere Bedeutung für uns hat.

```
data0 = data0.sort_values('date_time').reset_index(drop=True)
```

Da diese Transformation recht lange dauert, speichern wir den bereinigten Data Frame und lesen diesen in späteren Notebooks neu ein.

```
data0.to_csv('power_cleaned.zip', index=False)
```

Notebook

Download: [electric-power-rnn0.ipynb](#) ([Preview](#))

Das Einlesen geschieht dann wie üblich, nur die Datumsspalte muss spezifiziert werden:

```
data0 = pd.read_csv('power_clean.zip', parse_dates=['date_time'])
```

Da wir hier nur eine Zeitreihe haben, müssen wir diese längs der Zeitachse in Trainings-, Validierungs und Testmenge aufteilen. Es ist sicher sinnvoll, die neuesten Daten zum Test zu verwenden, da wir an diesen am ehesten abschätzen können, wie die Güte unserer Vorhersage für zukünftige Daten ist. Diese Mengen sollten sich nicht überschneiden.

Wir verwenden die Daten der letzten Woche für die Testmenge, die der beiden vorhergehenden Wochen als Validierungsmenge und den Rest für das Training.

```
split_test_date = data0.date_time.max() - pd.Timedelta(7, 'D')
split_valid_date = data0.date_time.max() - pd.Timedelta(21, 'D')

data_train = data0[data0.date_time <= split_valid_date]
data_valid = data0[(data0.date_time > split_valid_date) &
    (data0.date_time <= split_test_date)]
a_test = data0[data0.date_time > split_test_date]
print(data_train.shape, data_valid.shape, data_test.shape)
print('Train', data_train.date_time.min(), data_train.date_time.max())
print('Valid', data_valid.date_time.min(), data_valid.date_time.max())
print('Test ', data_test.date_time.min(), data_test.date_time.max())
```

Ergebnis:

```
(2025525, 4) (13676, 4) (10079, 4)
Train 2006-12-16 17:24:00 2010-11-20 23:59:00
Valid 2010-11-21 00:00:00 2010-12-04 23:59:00
Test 2010-12-05 00:00:00 2010-12-11 23:59:00
```

Aus diesen Daten generieren wir nun jeweils Zeitreihen hintereinanderliegender Daten, verwenden alle bis auf den letzten Wert als Komponente und den letzten Wert als vorherzusagende Zielgröße. Aufgrund der ausreichend großen Datenmenge verzichten wir hier auf Überlappungen.

Jeder Datenpunkt besteht also aus `n_steps` Werten und einem zugehörigen Label. Da wir häufig nicht nur einen, sondern mehrere Werte für einen Zeitpunkt vorhersagen müssen, benötigen wir das Label als Vektor mit der entsprechenden Dimension (hier 1). Ebenso stehen häufig zu jedem Zeitpunkt mehrere Komponenten zur Verfügung (hier könnten wir statt nur der Gesamtenergie auch noch die Spannung und die einzelnen Zählerwerte einbeziehen), wir müssen also auch hier Vektoren bereitstellen. Jeder Datenpunkt besteht also aus den Komponenten zu allen möglichen Zeitpunkten.

Betrachten wir alle Datenpunkte, bilden diese einen Tensor mit den Dimensionen `[#Daten, n_steps, #Features]` und unsere Label einen Tensor mit den Dimensionen `[#Daten, label_dim]`.

Da diese Zeitreihen zeitlich geordnet sind, mischen wir sie anschließend durch, um systematische Fehler durch Abhängigkeiten zwischen benachbarten Zeitreihen zu vermeiden.

```
def create_xy(data, n_steps):
    values = data.Global_active_power.values
    n_batches = values.size // (n_steps+1)
    val_reduced = values[:n_batches*(n_steps+1)]
    x_y = val_reduced.reshape(-1, (n_steps+1), 1)
    x_y = shuffle(x_y, random_state=23)
    return x_y[:, :n_steps, :], x_y[:, -1:, :]

n_steps = 50
Xtrain, yTrain = create_xy(data_train, n_steps)
Xvalid, yValid = create_xy(data_valid, n_steps)
Xtest, yTest = create_xy(data_test, n_steps)
```

21.3 Basismodelle

Um die Güte unserer Modelle zu bewerten, bietet es sich an, zunächst triviale Vorhersagen zu erstellen. Dazu können wir einfach den letzten Wert unserer Trainingsdaten als Vorhersage verwenden:

```
def naive_predict(X):
    return X[:, -1, :] # Alle Datenpunkte, letzter Wert, 1-dim. Vektor
```

Für die ersten 5 Trainingswerte ergibt sich folgendes Bild.

Wenden wir dieses Modell auf die Testdaten an, erhalten wir einen Vektor mit den gleichen Dimensionen wie unser Label-Vektor. Zur Bewertung von Regressionsmodellen verwenden wir meist den *MSE*.

```
predTest = naive_predict(Xtest)
print('MSE', np.mean(keras.losses.mean_squared_error(yTest, predTest)))
```

Wir erhalten einen Wert von 0.069, den es zu verbessern gilt. Dazu können wir im ersten Schritt eine lineare Regression verwenden, die sich einfach mit einem *ANN* mit einem Layer ohne Aktivierungsfunktion realisieren lässt. Da ein Dense-Layer einen Input-Shape von `[batch_size, #features]` erwartet, unsere Daten aber den Shape `[batch_size, n_steps, #features]` haben, müssen wir mit dem Layer `Flatten` erst den Shape umwandeln, indem die Features aller Zeitpunkte eines Datensatzes zu einem Vektor verkettet werden.

```
def build_model0(n_steps):
    inp = keras.layers.Input([n_steps, 1])
```

(Fortsetzung auf der nächsten Seite)

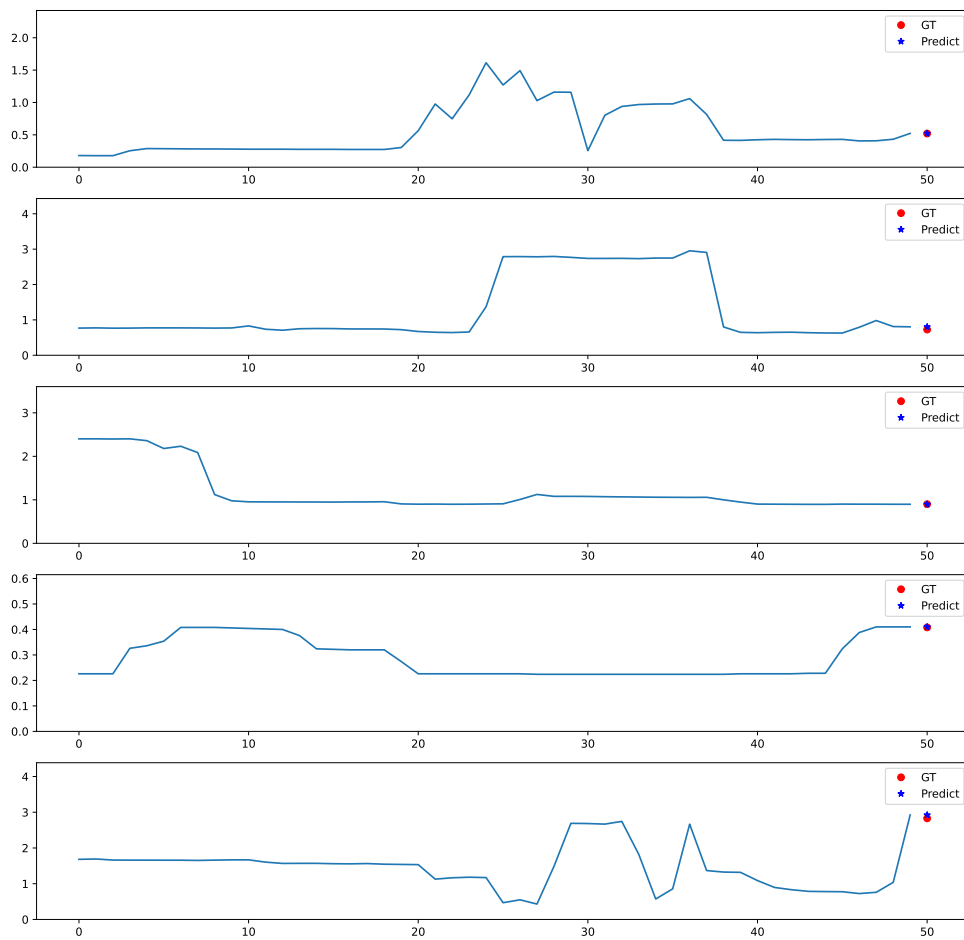


Abb. 21.1: Wahre Werte und Vorhersage des naiven Modells.

(Fortsetzung der vorherigen Seite)

```

x = keras.layers.Flatten()(inp)
out = keras.layers.Dense(1)(x)
return keras.models.Model(inp, out)

model0 = build_model0()

history = model0.fit(
    Xtrain, yTrain,
    batch_size=64,
    epochs=20,
    validation_data=(Xvalid, yValid),
)

model0.evaluate(Xtest, yTest)

```

Wir erhalten hier ebenfalls einen *MSE* von 0.069. Mit weiteren Layern mit nichtlinearer Aktivierung sind sicher Verbesserungen möglich.

21.4 Recurrent Neural Networks

Der Nachteil von von *MLP* für diese Art von Problemen ist jedoch, dass der zeitlichen Zusammenhang nicht berücksichtigt wird. Dies führte zur Einführung von *RNNs*. Wir senden nun die Datenpunkte einer Zeitreihe der Reihe nach alle durch das gleiche Netz. Zusätzlich zum Datenpunkt nehmen wir aber auch die Ausgabe des Netzes aus dem vorigen Zeitschritt in die Eingabe auf (im ersten Zeitschritt verwenden wir einen Null-Vektor).

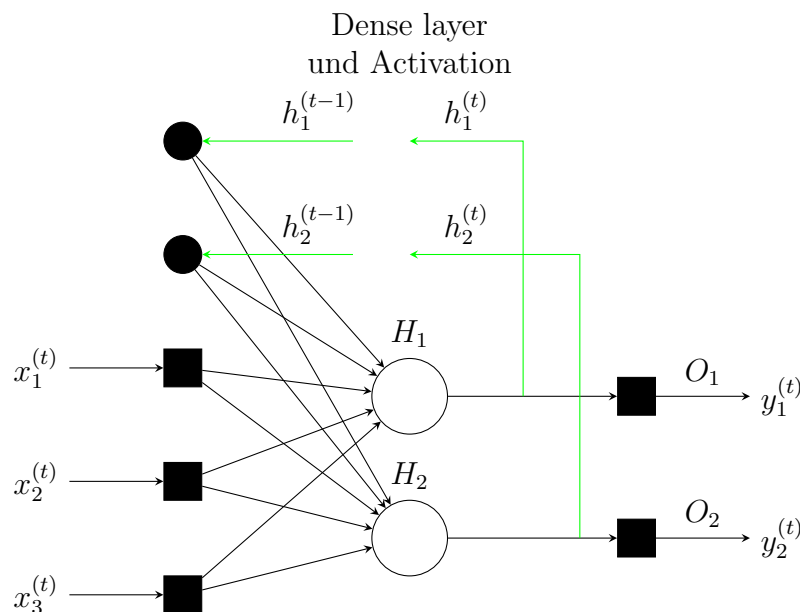


Abb. 21.2: RNN-Layer (Eingabe-Dimension 3, Ausgabe-Dimension 2)

Die Ausgabe $\mathbf{y}^{(t)}$ erhalten wir für jeden Zeitpunkt t der Eingabefolge und können diese damit als Eingabe eines weiteren *RNN*-Layers benutzen. In unserem Falle benötigen wir aber nur die Ausgabe zum letzten Zeitpunkt, die unsere Vorhersage ist.

Ursprünglich wurde für den *RNN*-Layer ein einfaches Neuron mit der Funktion $\tanh()$ als Aktivierungsfunktion verwendet. Es stellte sich jedoch heraus, dass der Zustandsvektor $\mathbf{h}^{(t)}$ nur sehr kurz in die Vergangenheit schauen kann, deshalb verwendet man heute eine kompliziertere Struktur für die *RNN*-Knoten:

- *LSTM*
- *GRU*

Die Schnittstellen aller drei Knotentype in *TensorFlow* ist identisch, sie können beliebig ausgetauscht werden. Sie erwarten als Eingabe-Shape stets einen Tensor der Form `[batch_size, folgenlänge, feature_dim]`, die Ausgabe des kompletten Layers hat die Form `[batch_size, folgenlänge, layer_breite]`. Bei der Spezifikation des Netzes entfällt die Angabe der Batch-Größe, die Folgenlänge kann mit `None` angegeben werden. Wichtig ist, dass der Wertebereich der Ausgabe nur das Intervall $[-1, 1]$ umfasst. Für Werte außerhalb dieses Intervalls benötigen wir anschließend einen normalen Dense-Layer, der auf jeden Zeitschritt angewendet wird.

```
keras.layers.GRU(breite,
                 [input_shape=dimensionen],
                 [return_sequences=True],
                 ...
                )
```

Normalerweise gibt der *RNN*-Layer nur den letzten Wert zurück. Wollen wir die Werte aller Zeitschritte haben (was zum Stapeln mehrerer *RNN*-Layer erforderlich ist), müssen wir dies mit `return_sequences=True` einschalten.

Wir können nun unser Netz konstruieren:

```
def build_model1():
    inp = keras.layers.Input([None, 1])
    x = keras.layers.GRU(20)(inp)
    out = keras.layers.Dense(1)(x)
    return keras.models.Model(inp, out)

model1 = build_model1()

history = model1.fit(
    Xtrain, yTrain,
    batch_size=64,
    epochs=20,
    validation_data=(Xvalid, yValid),
)

model1.evaluate(Xtest, yTest)
```

Unser einfaches *RNN*-Modell erreicht einen *MSE* von 0.063 und damit eine leichte Verbesserung gegenüber unseren einfachen Modellen.

Wir können nun versuchen, mehrere *RNN*-Layer zu stapeln. Wir setzen dazu den Parameter `return_sequences` außer für den letzten Layer auf `True`. Wir müssen hier die Lernrate etwas verringern, da sonst der Optimierer nicht stabil den Fehler verringert.

```
def build_model2(layer_width=20):
    inp = keras.layers.Input([None, 1])
    x = keras.layers.GRU(layer_width, return_sequences=True)(inp)
    x = keras.layers.GRU(layer_width)(x)
    out = keras.layers.Dense(1)(x)
    return keras.models.Model(inp, out)

model2 = build_model2()

model2.compile(
    optimizer=keras.optimizers.Adam(lr=2e-4),
    loss='mse',
    metrics=['mae'],
)

history = model2.fit(
    Xtrain, yTrain,
    batch_size=64,
    epochs=20,
    validation_data=(Xvalid, yValid),
)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
model2.evaluate(Xtest, yTest)
```

Wir erhalten eine weitere Verbesserung auf einen *MSE* von 0.060.

Bemerkung: Der Einfachheit halber wurden die Eingabedaten nicht umskaliert. Werden diese standardisiert, ist eine weitere Modellverbesserung zu erwarten. Allerdings ist dies mit den *Scikit-learn*-Mitteln nicht direkt möglich, da unsere Daten nun 3- und nicht mehr 2-dimensional sind.

Notebook

Download: `electric-power-rnn1.ipynb` ([Preview](#))

21.5 Vorhersage mehrerer Werte

Oft wollen wir nicht nur einen, sondern eine ganze Folge von Werten vorhersagen. Wir können dazu entweder immer den zuletzt vorhergesagten Wert zu unseren Features hinzunehmen, den bisherigen ersten Wert weglassen und mit den neuen, um 1 verschobenen Daten die nächste Vorhersage treffen.

Wir wollen beispielsweise die nächsten 10 Werte vorhersagen. Dazu modifizieren wir die Funktion `create_xy()` für unsere Testdaten. Für das Training möchten wir dagegen weiter den ersten Wert in der Zukunft benutzen:

```
# identisch zur vorigen Funktion create_xy()
def create_train(data, n_steps):
    values = data.Global_active_power.values
    n_batches = values.size // (n_steps+1)
    val_reduced = values[:n_batches*(n_steps+1)]
    x_y = val_reduced.reshape(-1, (n_steps+1), 1)
    x_y = shuffle(x_y, random_state=23)
    return x_y[:, :n_steps], x_y[:, -1, :1]

def create_test(data, n_steps, n_predict):
    values = data.Global_active_power.values
    n_batches = values.size // (n_steps+n_predict)
    val_reduced = values[:n_batches*(n_steps+n_predict)]
    x_y = val_reduced.reshape(-1, (n_steps+n_predict), 1)
    x_y = shuffle(x_y, random_state=23)
    return x_y[:, :n_steps], x_y[:, -n_predict:, :1]
```

Die Label enthalten nun für jede Folge 10 Werte. Für das Training benutzen wir nur den ersten Wert davon.

```
n_steps = 50
n_predict = 10
Xtrain, yTrain = create_train(data_train, n_steps, n_predict)
Xvalid, yValid = create_train(data_valid, n_steps, n_predict)
Xtest, yTest = create_test(data_test, n_steps, n_predict)
```

Wir erzeugen zunächst wieder unser Baseline-Modell, das den letzten Wert vorhersagt, schreiben nun aber dafür eine Klasse mit der Methode `predict()`.

```
class BaseLineModel:
    def predict(self, X):
        """
        X.shape = (n_batch, n_steps, n_features)
        y.shape = (n_batch, n_features)
```

(Fortsetzung auf der nächsten Seite)

```
"""
return X[:, -1, :]
```

Um `n_predict` Schritte vorherzusagen, hängen wir jeweils das Ergebnis der letzten Vorhersage an die Eingabefolge an, schneiden den ersten Wert ab, um die Länge zu erhalten (nicht erforderlich, da *RNNs* mit beliebig langen Folgen arbeiten können), und wenden unser Modell auf diese neue Folge an, um den nächsten Wert vorherzusagen (*Autoregressionsmodell*).

```
def predict_n_steps(model, X, n_predict):
    Xtmp = X.copy()
    res = np.zeros((X.shape[0], n_predict, X.shape[2]))
    for i in range(n_predict):
        pred = model.predict(Xtmp)
        res[:, i, :] = pred[:, :]
        Xtmp = np.concatenate([
            X[:, 1:, :],
            pred.reshape(pred.shape[0], 1, pred.shape[1])
        ], axis=1)
    return res
```

Wir schreiben nun eine Bewertungsfunktion für diese `n_predict` Vorhersagen.

```
def eval_predict(y, pred):
    return np.mean(keras.losses.mean_squared_error(y, pred))
```

Für unser Baseline-Modell erhalten wir einen Wert von 0.24, der *MSE* für die Vorhersage nur des ersten Zeitschritts ist dagegen 0.064 (die Abweichung ergibt sich aus der abweichenden Erstellung der Testdaten).

Wir verwenden eines unserer vorigen *RNN*-Modelle für das Training.

```
history = model1.fit(
    Xtrain, y1Train,
    batch_size=64,
    epochs=20,
    validation_data=(Xvalid, y1Valid),
)
```

Damit erzeugen wir nun unsere `n_predict` Vorhersagen:

```
predTest = predict_n_steps(model1, Xtest, n_predict)
eval_predict(yTest, predTest)
```

Das Ergebnis ist jedoch nicht besonders gut, wir erhalten einen *MSE* von 0.33 und damit keine Verbesserung gegenüber dem Baseline-Modell, allerdings sehen wir im Bild, dass durchaus sinnvolle Vorhersagen getroffen werden.

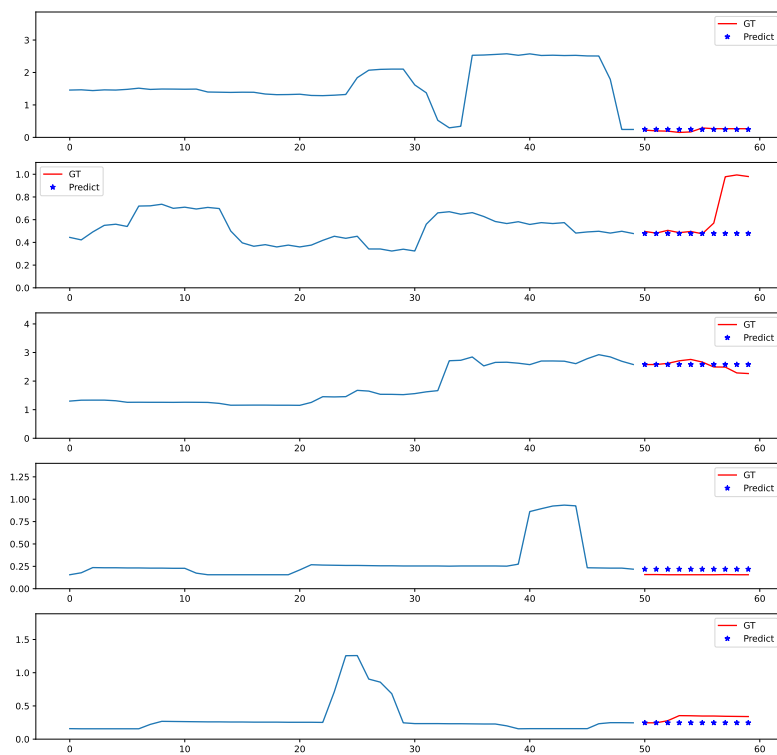


Abb. 21.3: Vorhersage des Baseline-Modells.

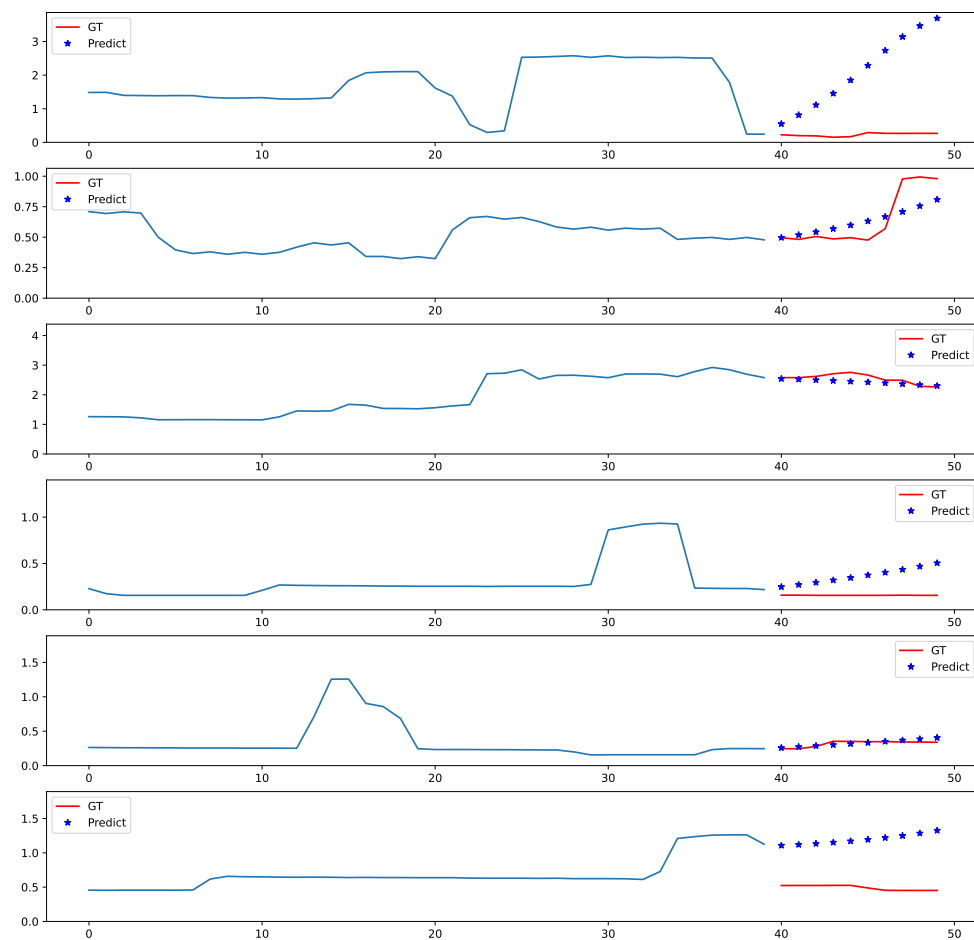


Abb. 21.4: Sequentielle Vorhersage von 10 Werten.

Notebook

Download: `electric-power-rnn2.ipynb`. ([Preview](#))

21.6 Vorhersage für jeden Zeitschritt

Wir können unser *RNN*-Modell verbessern, indem wir nicht nur eine Ausgabe nach dem letzten, sondern nach jedem Zeitschritt ermöglichen und dafür entsprechende Label erzeugen. Zum Zeitpunkt t erzeugt unser *RNN* eine Vorhersage $\hat{y}^{(t)}$, die wir mit dem tatsächlichen Funktionswert $y^{(t+1)}$ zum Zeitpunkt $t+1$ vergleichen.

Dazu müssen wir für unsere Trainingsdaten nicht nur für den letzten, sondern nun für jeden Zeitpunkt ein Label erzeugen.

```
def create_train(data, n_steps):
    values = data.Global_active_power.values
    n_batches = values.size // (n_steps+1)
    val_reduced = values[:n_batches*(n_steps+1)]
    x_y = val_reduced.reshape(-1, (n_steps+1), 1)
    x_y = shuffle(x_y, random_state=23)
    # neu: X enthält die Werte (0...n_steps-1), y die Werte (1...n_steps)
    return x_y[:, :n_steps], x_y[:, 1:n_steps+1, :1]

Xtrain, yTrain = create_train(data_train, n_steps)
Xvalid, yValid = create_train(data_valid, n_steps)
```

Wir passen zum Vergleich auch unser Baseline-Modell an, indem wir für jeden Zeitpunkt einen Wert zurückgeben (also einfach die Trainings-Features X).

```
class BaseLineModel:
    def predict(self, X):
        """
        X.shape = (n_batch, n_steps, n_features)
        y.shape = (n_batch, n_steps, n_features)
        """
        return X
```

Für Vorhersagen benötigen wir allerdings nur den letzten Zeitschritt, wir passen also unsere Funktion `predict_n_steps()` entsprechend an.

```
def predict_n_steps(model, X, n_predict):
    Xtmp = X.copy()
    res = np.zeros((X.shape[0], n_predict, X.shape[2]))
    for i in range(n_predict):
        pred = model.predict(Xtmp)[:, -1:, :] # nur letzter Zeitschritt
        res[:, i, :] = pred[:, 0, :]
        Xtmp = np.concatenate(
            [X[:, 1:, :], pred.reshape(pred.shape[0], 1, pred.shape[1])],
            axis=1
        )
    return res
```

Unser Modell müssen wir nun so anpassen, dass auch der letzte *RNN*-Layer Ausgaben erzeugt:

```
x = keras.layers.GRU(layer_width, return_sequences=True)(x)
```

Nun benötigen wir aber anschließend noch einen Dense-Layer, der für jeden Zeitschritt ausgeführt werden soll. Um diese zu realisieren, also den gleichen Layer auf alle Zeitpunkte anzuwenden, stellt uns *Keras* einen passenden Layer `TimeDistributed` zur Verfügung:

```
keras.layers.TimeDistributed(einfacherLayer)
```

Damit konstruieren wir unser neues Netz:

```
def build_model1(layer_width=20):
    inp = keras.layers.Input([None, 1])
    x = keras.layers.GRU(layer_width, return_sequences=True)(inp)
    x = keras.layers.GRU(layer_width, return_sequences=True)(x)
    out = keras.layers.TimeDistributed(keras.layers.Dense(1))(x)
    return keras.models.Model(inp, out)
```

Unser Modell hat nun folgendes Layout:

Model: "model"

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	[(None, None, 1)]	0

gru (GRU)	(None, None, 20)	1380

gru_1 (GRU)	(None, None, 20)	2520

time_distributed (TimeDistri	(None, None, 1)	21
=====		

Total params: 3,921
Trainable params: 3,921
Non-trainable params: 0

Nach dem Training treffen wir wieder unsere Vorhersage für `n_predict` Zeitschritte.

```
predTest = predict_n_steps(model2, Xtest, n_predict)
eval_predict(yTest, predTest)
```

Wir erzielen einen *MSE* von 0.24 und damit zumindest die gleiche Güte wie unser Baseline-Modell.

Notebook

Download: [electric-power-rnn3.ipynb](#). ([Preview](#))

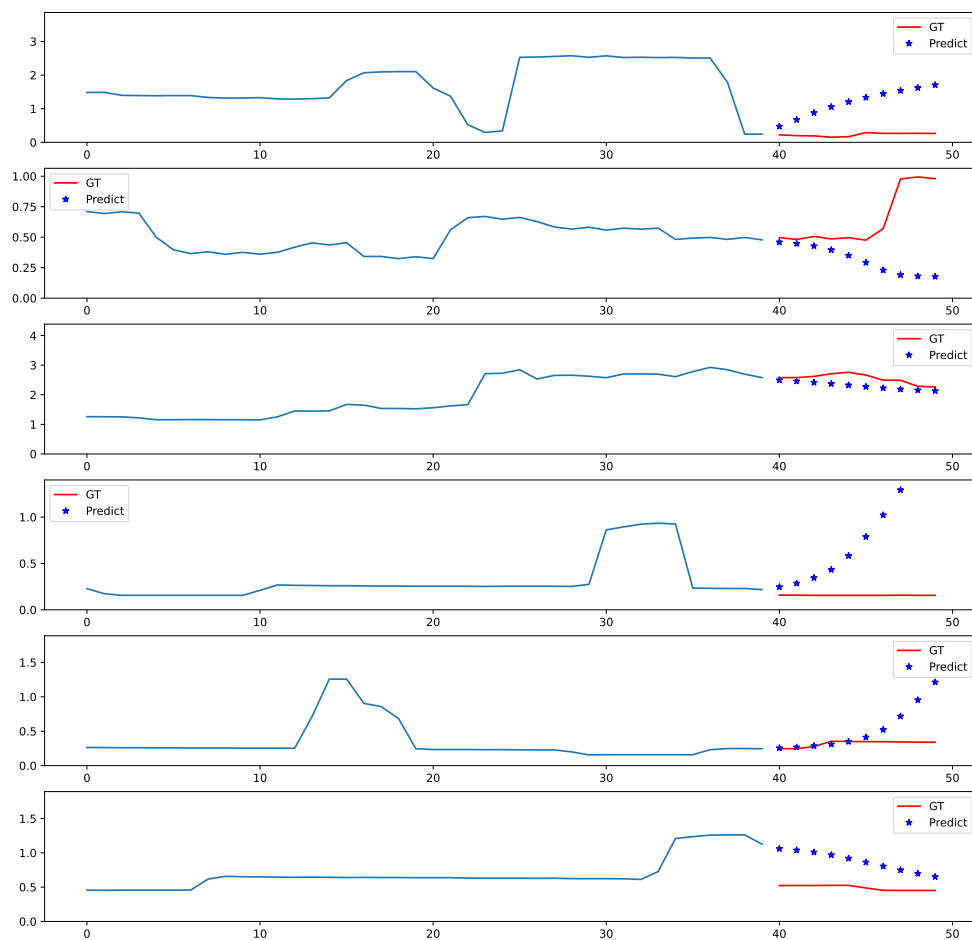


Abb. 21.5: Sequentielle Vorhersage von 10 Werten mit Vorhersage für jeden Zeitschritt.

ANN *Artificial Neural Network*

Künstliches Neuronales Netzwerk.

API *Application Programming Interface*

AWS *Amazon Web Services*

AZW *Ausbildungszentrum Wirtschaftsinformatik*

Computerpools der Wirtschaftsinformatik (C25.031, C25.148, C25.149).

CNN *Convolutional Neural Network*

Spezielles neuronales Netz hauptsächlich für die Bildverarbeitung auf der Basis von Faltungen (Convolutions)

CSV *Comma Separated Values*

Typisches textbasiertes Exportformat von Tabellenkalkulationsprogrammen, Zellwerte sind durch Komma bzw. Semikolon getrennt.

CUDA *Compute Unified Device Architecture*. Programmierschnittstelle für Graphikkarten der Firma *Nvidia*.

DNN *Deep Neural Network*

Neuronales Netz mit einer großen Zahl verborgener Schichten.

DSGVO *DatenSchutzGrundVerOrdnung*

DT *Decision Tree* (Entscheidungsbaum)

GloVE *GLobal Vectors for Word Representation*

GPU *Graphics Processing Unit*. Die Recheneinheiten einer Graphikkarte.

GRU *Gated Recurrent Unit*. Spezieller Knoten eines *RNN*

GT *Ground Truth*. Die tatsächlichen Werte der Zielgröße.

IDE *Integrated Development Environment*

JSON *Java Objects Over Network*

Beliebtes textbasiertes Format zum Speichern strukturierter Daten

KNN *k-Nearest Neighbors*

LSTM *Long-Short-Term Memory*. Spezieller Knoten eines *RNN*

ML *Machine Learning*

MLP *MultiLayer Perceptron*

Vollständig verbundenes Feedforward Netz mit mindestens einer verborgenen Schicht.

MNIST *Modified National Institute of Standards and Technology* (Database)

Datensammlung von 60.000 Trainings und 10.000 Testdaten handgeschriebener Ziffern.

MSE *Mean Squared Error*

Mittlerer quadratischer Fehler.

NLP *Natural Language Processing*

PCA *Principal Component Analysis*

Hauptkomponentenanalyse

PDF *Portable Document Format*

PNG *Portable Network Graphics*

QOP *Quadratic Optimizing Problem*

Spezielle Klasse von konvexen Optimierungsproblemen, die ein eindeutiges Minimum haben und recht effizient lösbar sind.

RBF *Radial Base Function*

Auch *Gauß-Kernel*. Hier: Transformationsverfahren für unsere Komponenten in einen hochdimensionalen Raum.

RGB *Red-Green-Blue*

Farbraumbezeichnung für Bildschirmdarstellung

RNN *Recurrent Neural Network*

SGD *Stochastic Gradient Descent*

Optimierungsverfahren

SVM *Support Vector Machine*

SVG *Scalable Vector Graphics*

SQL *Structured Query Language*

UCI *University of California, Irvine*

URL *Uniform Resource Locator*

ZIP Kompressionsformat (Name bedeutet *herumrennen*, *abzischen*)

Literaturverzeichnis

- [agg18] Aggarwal, C. C.: *Neural Networks and Deep Learning*. Springer 2018.
- [bis06] Bishop, C. M.: *Pattern Recognition and Machine Learning*. Springer 2006.
- [der20] Deru, M., Ndiaye, A: *Deep Learning mit TensorFlow, Keras und TensorFlow.js*. Rheinwerk 2020.
- [ger19] Géron, Aurélien: *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. 2nd ed. O'Reilly 2019.
- [goo16] Goodfellow, Bengio, Courville: *Deep Learning*. MIT 2016.
- [muel17] Müller, Guido: *Einführung in Machine Learning mit Python*. O'Reilly 2017.
- [van18] VanderPlas, Jake: *Data Science mit Python*. mitp 2018.
- [anaconda] <https://www.anaconda.com/> Anaconda 3 Data Science Platform
- [colab] <https://colab.research.google.com/> Google Colab
- [markdown] <https://help.github.com/en/articles/getting-started-with-writing-and-formatting-on-github> Getting started with writing and formatting on GitHub. Benötigte *Markdown*-Syntax für *Jupyter Notebook*.
- [matplotlib] <https://matplotlib.org/> Matplotlib
- [miniconda] <https://docs.conda.io/en/latest/miniconda.html> Miniconda Installer
- [numpy] numpy.org <<https://numpy.org>> NumPy-Homepage.
- [py101] <https://www.machinelearningplus.com/python/101-numpy-exercises-python/101> NumPy Exercises for Data Analysis (Python).
- [pandas] pandas.pydata.org Pandas-Homepage.
- [sklearn] scikit-learn.org scikit-learn-Homepage.
- [tfhome] <https://www.tensorflow.org/> TensorFlow-Homepage mit zahlreichen Tutorials.
- [nie15] Nielson, A., *Neural Networks and Deep Learning*. <http://neuralnetworksanddeeplearning.com/> Determination Press 2015.
- [uci] <http://archive.ics.uci.edu/ml/> UCI Machine Learning Repository.
- [kaggle] <https://www.kaggle.com/> Kaggle (Account erforderlich).
- [openml] <https://www.openml.org/> OpenML (Scikit-learn-Interface vorhanden).
- [awsdata] <https://aws.amazon.com/de/opendata/> AWS Open Data
- [googledata] <https://toolbox.google.com/datasetsearch> Google Data Search Engine

A

AdaBoost, 160
ANN, **263**
 Dropout, 224
 L2-Regularisierung, 222
API, **263**
Aufteilen in Trainings- und Test-
 daten, 40
Autoregression, 256
AWS, **263**
AZW, **263**

B

Bagging, 157
Batch-Normalisierung, 227
Beispieldaten
 scikit-learn, 69
 synthetische, 71

C

CNN, **263**
Cross Validation, 181
CSV, **263**
CUDA, **263**

D

Data Frame
 Gruppieren, 31
Datenanalyse, 28
DNN, **263**
Dropout, 224
DSGVO, **263**
DT, **263**

E

Ensemble-Verfahren
 AdaBoost, 160
 Bagging, 157
 Gradient Boosting, 160
 Random Forest, 157
 Stacking, 155

Voting, 155

G

Gittersuche, 179
GloVE, **263**
GPU, **263**
Gradient Boosting, 160
Grid Search, 179
GRU, **263**
Gruppieren, 31
GT, **263**

H

Hyperparameter
 Cross Validation, 181
 Gittersuche, 179

I

IDE, **263**

J

JSON, **263**

K

K-Nearest-Neighbors, 111
KNN, 111, **263**

L

L2-Regularisierung
 ANN, 222
Lernrate, 230
Lineare Regression, 127
Logistische Regression, 116
LSTM, **263**

M

Machine Learning
 Aufgaben, 5
 Definition, 5
Mehrfachklassifikation, 120
ML, **264**

MLP, [264](#)

MNIST, [264](#)

MSE, [264](#)

N

Naive-Bayes-Klassifikation, [114](#)

NLP, [264](#)

O

One versus One, [120](#)

One versus Rest, [120](#)

P

Pandas, [28](#)

 Dateiarbeit, [24](#)

 Datenbereinigung, [25](#)

 Elementzugriff, [25](#)

 Kategoriale Features, [29](#)

PCA, [264](#)

PDF, [264](#)

PNG, [264](#)

Q

QOP, [264](#)

R

Random Forest, [157](#)

RBF, [264](#)

Regression

 Elastic-Net, [128](#)

 Entscheidungsbaum, [128](#)

 KNN, [128](#)

 Lasso, [128](#)

 linear, [127](#)

 nichtlinear, [132](#)

 Ridge, [128](#)

 SVM, [173](#)

Regularisierung

 Lernrate, [230](#)

RGB, [264](#)

RNN, [264](#)

S

SGD, [264](#)

Spezielle Matrizen, [50](#)

Spezielle Vektoren, [50](#)

SQL, [264](#)

Stacking, [155](#)

Stichprobe, [40](#)

SVG, [264](#)

SVM, [264](#)

 Duales Problem, [165](#)

 Kernel-Trick, [165](#)

 Nichtlineare Trennung, [165](#)

 Regression, [173](#)

U

UCI, [264](#)

URL, [264](#)

V

Voting, [155](#)

Z

ZIP, [264](#)