Einführung in die mathematische Datenanalyse

Jan Heiland

FAU Erlangen-Nürnberg – Sommersemester 2022

Contents

Vorwort

Das ist ein Aufschrieb der parallel zur Vorlesung erweitert wird.

Korrekturen und Wünsche immer gerne als *issues* oder *pull requests* ans github-repo.

Chapter 1

Was ist Data Science?

Data Science umfasst unter anderem folgende Aufgaben:

1. Strukturieren/Aufbereiten (Umgehen mit falschen, korrumpierten, fehlenden, unformatierten Daten)
2. Data Exploration (Daten "verstehen")
3. Data Analysis (quantitative Analysen, Hypothesen aufstellen)
4. Data Visualization (Hypothesen graphisch kommunizieren)
5. Modelle erzeugen/validieren (Regeln/Muster erkennen, Vorhersagen treffen) – *das* ist Machine Learning aber es gibt auch viele andere Ansätze.
6. Daten Reduktion

1.1 Wie passiert die Datenanalyse?

Mit mathematischen Methoden aus den Bereichen der

- linearen Algebra (z.B. Matrizen, Basen, lineare Gleichungssysteme)
- Statistik (z.B. Mittelwerte, Korrelationen, Verteilungen)
- Analysis (Grenzwerte, Abschätzungen)
- …

Dabei hilft Software, z.B.,

- Excel
- **Python**
- Matlab
- R

bei der Berechnung, Automatisierung, Visualisierung.

Python ist ein de-facto Standard in Data Science und Machine Learning.

1.2 Was sind Daten?

Wie sehen Daten aus?

- Numerisch reell, z.B. Temperatur
- Numerisch diskret, z.B. Anzahl
- Ordinal: Element einer festen Menge mit expliziter Ordnung, z.B. {neuwertig, mit Gebrauchsspuren, defekt}
- Binär: Eine von zwei Möglichkeiten, z.B. Wahr/Falsch oder aktiv/inaktiv
- Kategoriell: Element einer festen Menge ohne klare Ordnung, z.B. {Säugetier, Vogel, Fisch}
- sonstige strukturierte Daten, z.B. geographische Daten, Graphen
- reiner Text, z.B. Freitext in Restaurantbewertung

Außerdem können wir noch allgemeine Eigenschaften (Qualitätsmerkmale) von Daten unterscheiden

- strukturiert
- lückenhaft
- fehlerbehaftet (*verrauscht*)
- interpretierbar
- geordnet (oder nicht zu ordnen)

1.3 Beispiele

1.3.1 Tabellendaten – Mietpreise

Hier wären die Aufgaben von Data Science:

- Daten "verstehen", Zusammenhänge zwischen Variablen aufdecken,
- visualisieren.
- Gegebenenfalls fehlende Einträge bei (z.B.) `kaltmiete` vorhersagen

Datenexploration und -analyse für einzelne Variablen 1/3 Wir betrachten eine *numerische* Variable in einem rechteckigen Datensatz, also eine *Spalte* (z.B. `kaltmiete`). Wir bezeichnen den i-ten Eintrag in dieser Spalte mit x_i, wobei $i = 1, \ldots, N$ (N Anzahl der Zeilen).

Folgende *Schätzer/Metriken* können dabei helfen, diese Spalte besser zu verstehen:

- Mittelwert $\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$

	bundesland	stadt	baujahr	etage	hat_kueche	kaltmiete	wohnflaeche	zimmer
0	Nordrhein_Westfalen	Duisburg	1973	1	0	640	105	3
1	Baden_Württemberg	Ulm	1936	1	0	302	43	2
2	Nordrhein_Westfalen	Wuppertal	1986	3	0	150	67	2
3	Rheinland_Pfalz	Rhein_Lahn_Kreis	1970	3	0	315	47	3
4	Sachsen	Chemnitz	1900	2	0	315	63	2
5	Rheinland_Pfalz	Mainz	1989	1	0	1200	96	4
6	Bayern	Aschaffenburg_Kreis	1965	2	0	722	85	3
7	Berlin	Berlin	1952	0	1	706	63	2
8	Sachsen	Mittelsachsen_Kreis	1996	3	1	220	29	1
9	Brandenburg	Potsdam	1985	2	1	400	34	1

Figure 1.1: Abbildung: Tabelle von Wohnungsangeboten

	bundesland	stadt	baujahr	etage	hat_kueche	kaltmiete	wohnflaeche	zimmer
0	Nordrhein_Westfalen	Duisburg	1973	1	0	640	105	3
1	Baden_Württemberg	Ulm	1936	1	0	302	43	2
2	Nordrhein_Westfalen	Wuppertal	1986	3	0	150	67	2
3	Rheinland_Pfalz	Rhein_Lahn_Kreis	1970	3	0	315	47	3
4	Sachsen	Chemnitz	1900	2	0	315	63	2
5	Rheinland_Pfalz	Mainz	1989	1	0	1200	96	4
6	Bayern	Aschaffenburg_Kreis	1965	2	0	722	85	3
7	Berlin	Berlin	1952	0	1	706	63	2
8	Sachsen	Mittelsachsen_Kreis	1996	3	1	220	29	1
9	Brandenburg	Potsdam	1985	2	1	400	34	1

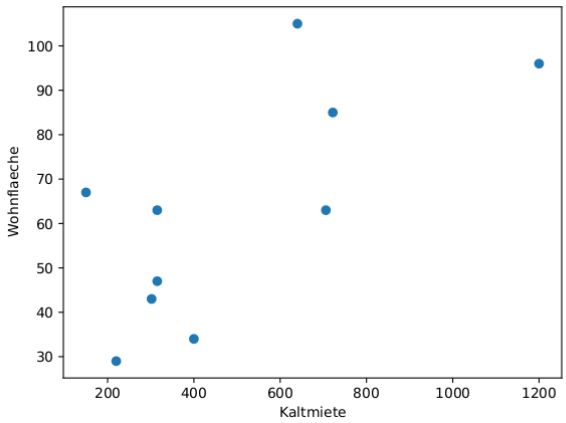
Figure 1.2: Abbildung: Eine Spalte der Tabelle

- gewichteter Mittelwert $\overline{x}_w = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{j=1}^{N} w_j}$, wobei w_i das Gewicht des i-ten Eintrages ist (z.B. eine andere Variable).
- Varianz: $s_x^2 = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})^2$
- Standardabweichung $s = \sqrt{s_x^2}$.
- Median $= \frac{315+400}{2} = 357.5$.

Datenexploration und -analyse für mehrere Variablen Wir betrachten zwei Spalten $x = (x_1, \dots, x_N)$ und $y = (y_1, \dots, y_N)$.

Das Verteilung von zwei Variablen läßt sich im sogenannte **Scatter Plot** visualisieren.

che	kaltmiete	wohnflaeche	zin
0	640	105	
0	302	43	
0	150	67	
0	315	47	
0	315	63	
0	1200	96	
0	722	85	
1	706	63	
1	220	29	
1	400	34	



Datenexploration und -analyse für mehrere Variablen Wir betrachten zwei Spalten $x = (x_1, \dots, x_N)$ und $y = (y_1, \dots, y_N)$.

- Kovarianz $s_{xy} = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \overline{x})(y_i - \overline{y})$
- Korrelation $\rho_{xy} = \frac{s_{xy}}{s_x \cdot s_y} \in [-1, 1]$.
- $\rho \approx 1$: Starke positive Korrelation, wenn x groß ist, ist y auch groß.
- $\rho \approx -1$: Starke negative Korrelation, wenn x groß ist, ist y klein
- $\rho \approx 0$: Wenig/keine Korrelation.

1.3.2 COVID-19 Daten

Vergleiche die Einführung in *Mathematik für Data Science 1* vom letzten Semester.

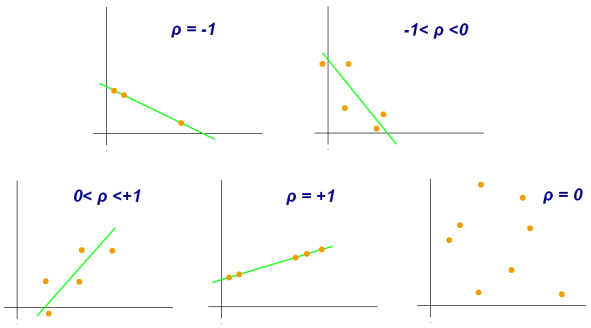


Figure 1.3: Von Kiatdd - Eigenes Werk, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=37108966

1.3.3 Netflix Prize

Hierbei geht es darum, ob aus bekannten Bewertungen von vielen verschiedenen Benutzern für viele verschiedene Filme abgeleitet werden kann, ob ein bestimmter Nutzer einen bestimmten Film mag (also positiv bewerten würde).

Vergleiche auch Wikipedia:Netflix_Prize

Das (Trainings-)Daten bestehen über `480189` Benutzer, die für `17770` Filme insgesamt `100480507` Bewertungen als ganze Zahlen zwischen `1` und `5` verteilten.

Ziel der Datenanalyse war es, für `2817131` "Paare" von Benutzern und Filmen, die Bewertung vorauszusagen. Neben der schieren Masse an Daten kamen noch Einschränkungen hinzu, die ein Mindestmaß an Qualität der Vorhersage sicherstellen sollten.

Das Problem ließe sich wie folgt darstellen.

Benutzer \ Film	`F1`	`F2`	...	`Fn`	...
`B1`	–	3	...	5	...
`B2`	3	4	...	2	...
`B3`	1	2	...	?	...
...	3	4	...	–	...

Gegeben viele (aber bei weitem nicht alle) Einträge in einer riesigen Tabelle. Können wir aus den Zusammenhängen bestimmte fehlende Einträge (z.B. wie findet Nutzer `B3` den Film `Fn`) herleiten?

Die besten Lösungen für dieses Problem basieren durchweg auf *Machine Learning* Ansätzen.

1.4 Python

Die Programmiersprache `python` wird uns durchs Semester begleiten. Einfach weil sie so wichtig ist für *Data Science* aber auch weil sie (meiner Meinung nach) einfach zu erlernen und zu benutzen ist.

1.5 Aufgaben

1.5.1 Python

Bringen sie ihr `python` zum Laufen, installieren sie `numpy`, `scipy` und

```
N = 20
xmax = 2
xmin = 0

xdata = np.linspace(xmin, xmax, N)
ydata = np.exp(xdata)

plt.figure(1)
plt.plot(xdata, ydata, '.')

plt.figure(2)
plt.semilogy(xdata, ydata, '.')
plt.show()
```

1.5.2 Einheitsmatrix

Schreiben sie ein script, dass die `5x5` Einheitsmatrix auf 3 verschiedene Arten erzeugt. (Eine Art könnte die eingebaute `numpy` Funktion `eye` sein).

```
import numpy as np

idfive = np.eye(5)
print(idfive)
```

Hinweis: schauen sie sich mal an wie `numpy`'s `arrays` funktionieren.

1.5.3 Matrizen Multiplikation und Potenz

Schreiben sie ein script, das die Übungsaufgabe aus der Vorlesung (potenzieren der Matrizen M_i, $i = 1, 2, 3, 4$) löst. Zum Beispiel mit

```
import numpy as np
mone = np.array([[0.9, 0.9], [0.9, 0.9]])

mone_ptwo = mone @ mone
print(mone_ptwo)

mone_pfour = mone_ptwo @ mone_ptwo
print(mone_pfour)
```

Oder so:

```
import numpy as np
mone = np.array([[0.9, 0.9], [0.9, 0.9]])
mone_p = np.eye(2)

for k in range(16):
```

```
    mone_p = mone_p @ mone
    if k == 1 or k == 3 or k == 15:
        print('k=', k+1)
        print(mone_p)
```

Achtung:

- bei Matrizen kann auch `*` benutzt werden – das ist aber nicht die richtige Matrizenmultiplikation (sondern die Multiplikation eintragsweise)
- Moegliche Realisierung der Matrizenmultiplikation
  - `np.dot(A, B)` – die klassische Methode
  - `A.dot(B)` – das selbe (manchmal besser, wenn `A` etwas allgemeiner ist (zum Beispiel eine `scipy.sparse` matrix)
  - `A @ B` – convenience Notation

Chapter 2

Lineare Regression

Auch bekannt als

- *lineare Ausgleichsrechnung* oder
- *Methode der kleinsten Quadrate*.

Ein wesentlicher Aspekt von *Data Science* ist die Analyse oder das Verstehen von Daten. Allgemein gesagt, es wird versucht, aus den Daten heraus Aussagen über Trends oder Eigenschaften des Phänomens zu treffen, mit welchem die Daten im Zusammenhang stehen.

Wir kommen nochmal auf das Beispiel aus der Einführungswoche zurück, werfen eine bereits geschärften Blick darauf und gehen das mit verbesserten mathematischen Methoden an.

Gegeben seien die Fallzahlen aus der CoVID Pandemie 2020 für Bayern für den Oktober 2020.

Table 2.1: Anzahl der SARS-CoV-2 Neuinfektionen in Bayern im Oktober 2020.

Tag	1	2	3	4	5	6	7	8	9	10	11
Fälle	352	347	308	151	360	498	664	686	740	418	320

Tag	12	13	14	15	16	17	18	19	20	21
Fälle	681	691	1154	1284	127	984	573	1078	1462	2239

Tag	22	23	24	25	26	27	28	29	30	31
Fälle	2236	2119	1663	1413	2283	2717	3113	2972	3136	2615

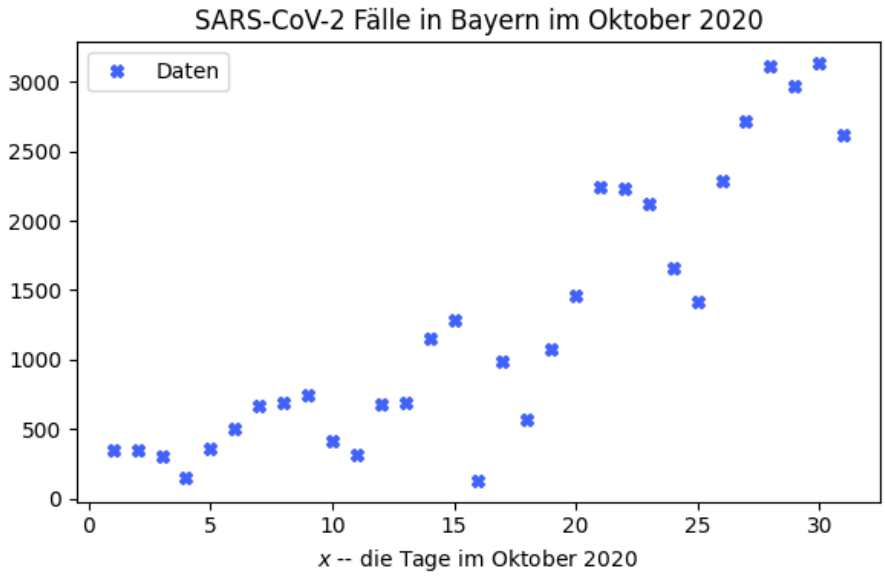


Figure 2.1: Fallzahlen von Sars-CoV-2 in Bayern im Oktober 2020

Wieder stellen wir uns die Frage ob wir **in den Daten einen funktionalen Zusammenhang** feststellen können. Also ob wir die Datenpaare

(Tag x, Infektionen am Tag x)

die wir als

(x_i, y_i)

über eine Funktion f und die Paare

$(x, f(x))$

beschreiben (im Sinne von gut darstellen oder approximieren) können.

2.1 Rauschen und Fitting

Beim obigen Beispiel (und ganz generell bei Daten) ist davon auszugehen, dass die Daten **verrauscht** sind, also einem Trend folgen oder in einem funktionalen Zusammenhang stehen aber zufällige Abweichungen oder Fehler enthalten.

Unter diesem Gesichtspunkt ist eine Funktion, die

$$f(x_i) = y_i$$

erzwingt nicht zielführend. (Wir wollen Trends und größere Zusammenhänge erkennen und nicht kleine Fehler nachzeichnen.) Das zu strenge Anpassen an möglicherweise verrauschte Daten wird **overfitting** genannt.

Vielmehr werden wir nach einer Funktion f suchen, die die Daten näherungsweise nachstellt:

$$f(x_i) \approx y_i$$

Hierbei passen jetzt allerdings auch Funktionen, die vielleicht einfach zu handhaben sind aber die Daten kaum noch repräsentieren. Jan spricht von **underfitting**.

Eine gute Approximation besteht im Kompromiss von *nah an den Daten* aber mit wenig *overfitting*.

2.2 Ansätze für lineare Regression

Um eine solche Funktion f zu finden, trifft Jan als erstes ein paar Modellannahmen. Modellannahmen legen fest, wie das f im Allgemeinen aussehen soll und versuchen dabei

1. die Bestimmung von f zu ermöglichen
2. zu garantieren, dass f auch die gewollten Aussagen liefert
3. und sicherzustellen, dass f zum Problem passt.

Jan bemerke, dass die ersten beiden Annahmen im Spannungsverhältnis zur dritten stehen.

Lineare Regression besteht darin, dass die Funktion als Linearkombination

$$f_w(x) = \sum_{j=1}^{n} w_j b_j(x)$$

von Basisfunktionen geschrieben wird und dann die *Koeffizienten* w_i so bestimmt werden, dass f die Daten bestmöglich annähert.

> Jan bemerke, dass *bestmöglich* wieder *overfitting* bedeuten kann aber auch, bei schlechter Wahl der Basis, wenig aussagekräftig sein kann. Der gute Kompromiss liegt also jetzt in der Wahl der passenden Basisfunktionen und deren Anzahl. (Mehr Basisfunktionen bedeutet möglicherweise bessere Approximation aber auch die Gefahr von *overfitting*.)

Typische Wahlen für die Basis $\{b_1, b_2, \dots, b_n\}$ sind

- Polynome: $\{1, x, x^2, \dots, x^{n-1}\}$ – für $n = 2$ ist der Ansatz *eine Gerade*

- Trigonometrische Funktionen: $\{1, \cos(x), \sin(x), \cos(2x), \sin(2x), \dots\}$
- Splines – Polynome, die abschnittsweise definiert werden
- Wavelets – Verallgemeinerungen von trigonometrischen Funktionen

2.3 Fehlerfunktional und Minimierung

Wir setzen nun also an

$$f_w(x) = \sum_{j=1}^{n} w_j b_j(x)$$

und wollen damit $y_i \approx f_w(x_i)$ *bestmöglich* erreichen (indem wir die Koeffizienten $(w_1, \dots, w_n)$ *optimal* wählen. Bestmöglich und optimal spezifizieren wir über den Mittelwert der quadratischen Abweichungen in der Approximation über alle Datenpunkte

$$\frac{1}{N} \sum_{i=1}^{N} (y_i - f_w(x_i))^2$$

Ein paar Bemerkungen

- jetzt müssen wir die w_i's bestimmen so dass dieser Fehlerterm minimal wird
- das optimale w ist unabhängig von einer Skalierung des Fehlerterms
- des wegen schreiben wir gerne einfach $\frac{1}{2} \sum_{i=1}^{N} (y_i - f_w(x_i))^2$ als das Zielfunktional, das es zu minimieren gilt.

Wie finden wir jetzt die w_i's? Zunächst gilt, dass

$$f_w(x_i) = \sum_{i=j}^{n} w_j b_j(x_i) = \begin{bmatrix} b_1(x_i) & b_2(x_i) & \dots & b_n(x_i) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

und wenn wir alle $f_w(x_i)$, $i = 1, \dots, N$ übereinander in einen Vektor schreiben, dass

$$f_w(\mathbf{x}) := \begin{bmatrix} f_w(x_1) \\ \vdots \\ f_w(x_N) \end{bmatrix} = \begin{bmatrix} b_1(x_1) & \dots & b_n(x_1) \\ \vdots & \ddots & \vdots \\ b_1(x_N) & \dots & b_n(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} =: \Phi(\mathbf{x}) w$$

Damit, mit $\mathbf{y}$ als den Vektor aller y_i's, und mit der Definition der Vektornorm, können wir unser Minimierungsproblem schreiben als

$$\frac{1}{2} \sum_{i=1}^{N} (y_i - f_w(x_i))^2 = \frac{1}{2} \|\mathbf{y} - \Phi(\mathbf{x}) w\|^2 \to \min .$$

Wir bemerken, dass

- das Fehlerfunktional immer größer und bestenfalls gleich 0 ist
- falls das lineare Gleichungssystem $\Phi(\mathbf{x})w = \mathbf{y}$ eine Lösung w hat, ist das auch eine Lösung unserer Minimierung
- im typischen Falle aber ist allerdings $N \gg n$ und das System überbestimmt ($n = N$ würde ein overfitting bedeuten…) sodass wir keine Lösung des linearen Gleichungssystems erwarten können.
- Das Minimierungsproblems selbst hat allerdings immer eine Lösung.

2.4 Berechnung der Bestlösung

Wir suchen also ein Minimum der Funktion (mit Φ, $\mathbf{x}$, $\mathbf{y}$ gegeben)

$$\begin{aligned} w \mapsto \frac{1}{2}\|\mathbf{y} - \Phi(\mathbf{x})w\|^2 &= \frac{1}{2}(\mathbf{y} - \Phi(\mathbf{x})w)^T(\mathbf{y} - \Phi(\mathbf{x})w) \\ &= \frac{1}{2}[\mathbf{y}^T\mathbf{y} - \mathbf{y}^T\Phi(\mathbf{x})w - w^T\Phi(\mathbf{x})^T\mathbf{y} + w^T\Phi(\mathbf{x})^T\Phi(\mathbf{x})w] \\ &= \frac{1}{2}[\mathbf{y}^T\mathbf{y} - 2w^T\Phi(\mathbf{x})^T\mathbf{y} + w^T\Phi(\mathbf{x})^T\Phi(\mathbf{x})w] \end{aligned}$$

wobei wir die Definition der Norm $\|v\|^2 = v^Tv$ und die Eigenschaft, dass für die skalare Größe $w^T\Phi(\mathbf{x})^T\mathbf{y} = [w^T\Phi(\mathbf{x})^T\mathbf{y}]^T = \mathbf{y}^T\Phi(\mathbf{x})w$ gilt, ausgenutzt haben.

Wären w und $\mathbf{y}$ keine Vektoren sondern einfach reelle Zahlen, wäre das hier eine Parabelgleichung $aw^2 + bw + c$ mit $a > 0$, die immer eine Minimalstelle hat.

Tatsächlich gilt hier alles ganz analog. Insbesondere ist $\Phi(\mathbf{x})^T\Phi(\mathbf{x})$ in der Regel "größer 0" (was heißt das wohl bei quadratischen Matrizen?). Und mittels "Nullsetzen" der ersten Ableitung können wir das Minimum bestimmen. In diesem Fall ist die erste Ableitung (nach w)

$$\nabla_w(\frac{1}{2}\|\mathbf{y} - \Phi(\mathbf{x})\|^2) = \Phi(\mathbf{x})^T\Phi(\mathbf{x})w - \Phi(\mathbf{x})^T\mathbf{y},$$

(den *Gradienten* ∇_w als Ableitung von Funktionen mit mehreren Veränderlichen werden wir noch genauer behandeln) was uns als Lösung, die Lösung des linearen Gleichungssystems

$$\Phi(\mathbf{x})^T\Phi(\mathbf{x})w = \Phi(\mathbf{x})^T\mathbf{y}$$

definiert.

Letzte Frage: Wann hat dieses Gleichungssystems eine eindeutige Lösung? Mit $N > n$ (also $\Phi(\mathbf{x})$ hat mehr Zeilen als Spalten) gelten die Äquivalenzen:

- $\Phi(\mathbf{x})^T\Phi(\mathbf{x})w = \Phi(\mathbf{x})^T\mathbf{y}$ hat eine eindeutige Lösung
- die Matrix $\Phi(\mathbf{x})^T\Phi(\mathbf{x})$ ist regulär
- die Spalten von $\Phi(\mathbf{x})$ sind linear unabhängig
- die Vektoren $b_i(\mathbf{x})$ sind linear unabhängig.

Praktischerweise tritt genau diese Situation im Allgemeinen ein.

- $N > n$ (mehr Datenpunkte als Parameter)
- b_i's werden als *linear unabhängig* (im Sinne ihres Funktionenraums) gewählt, was die lineare unabhängigket der $b_i(\mathbf{x})$ impliziert.

2.5 Beispiel

Unsere Covid-Zahlen "mit einer Geraden angenähert":

- $f_w(x) = w_1 + w_2 x$ – das heißt $n = 2$ und Basisfunktionen $b_1(x) \equiv 1$ und $b_2(x) = x$
- $\mathbf{x} = (1, 2, 3, \dots, 31)$ – die Tage im Februar, das heißt $N = 31$
- $\mathbf{y} = (352, 347, \dots, 2615)$ – die Fallzahlen

Wir bekommen

$$\Phi(\mathbf{x}) = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ \vdots & \vdots \\ 1 & 31 \end{bmatrix}$$

(die Spalten sind linear unabhängig) und müssen "nur" das 2x2 System

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & 31 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ \vdots & \\ 1 & 31 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & 31 \end{bmatrix} \begin{bmatrix} 352 \\ 347 \\ 308 \\ \vdots \\ 2615 \end{bmatrix}$$

lösen um die Approximation f_w zu bestimmen.

Und noch als letzte Bemerkung. Egal wie die Basisfunktionen b_i gewählt werden, die Parameterabhängigkeit von w ist immer linear. Deswegen der Name **lineare Ausgleichsrechnung**.

Chapter 3

Matrix-Zerlegungen

Die Lösung w des Problems der *linearen Ausgleichsrechnung* war entweder als Lösung eines Optimierungsproblems

$$\min_w \|Aw - y\|^2$$

oder als Lösung des linearen Gleichungssystems

$$A^T Aw = y$$

gegeben. Hierbei steht nun $A \in \mathbb{R}^{N \times n}$ für die Matrix $\Phi(\mathbf{x})$ der Daten und Basisfunktionen. Wir hatten uns überlegt, dass in den meisten Fällen

- die Matrix mehr Zeilen als Spalten hat ($N > n$) und
- die Spalten linear unabhängig sind.

3.1 QR Zerlegung

Wir betrachten nochmal das Optimierungsproblem $\min_w \|Aw - y\|^2$. Gäbe es eine Lösung des Systems $Aw = y$, wäre das sofort eine Lösung des Optimierungsproblems. Da A aber mehr Zeilen als Spalten hat, ist das System $Aw = y$ überbestimmt und eine Lösung in der Regel nicht gegeben.

Die Überlegung ist nun, die Gleichung $Aw = y$ *so gut wie möglich* zu erfüllen, indem wir die relevanten Gleichungen indentifizieren und wenigstens diese lösen. Ein systematischer (und wie wir später sehen werden auch zum Optimierungsproblem passender) Zugang bietet die QR Zerlegung.

Theorem 3.1 (QR Zerlegung)**.** *Sei $A \in \mathbb{R}^{m \times n}$, $m > n$. Dann existiert eine orthonormale Matrix $Q \in \mathbb{R}^{m \times m}$ und eine obere Dreiecksmatrix $\hat{R} \in \mathbb{R}^{n \times n}$ derart dass*

$$A = QR =: Q \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix}.$$

Hat A vollen (Spalten)Rang, dann ist $\hat{R}$ invertierbar.

Hier heißt *orthonormale Matrix* Q, dass die Spalten von Q paarweise orthogonal sind. Insbesondere gilt

$$Q^T Q = I.$$

Für unser zu lösendes Problem ergibt sich dadurch die Umformung

$$Aw = y \quad \Leftrightarrow \quad QRw = y \quad \Leftrightarrow \quad Q^T QRw = Q^T y \quad \Leftrightarrow \quad Rw = Q^T y$$

oder auch

$$\begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} w = Q^T y = \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} y$$

(wobei wir die $Q_1 \in \mathbb{R}^{m \times n}$ die Matrix der ersten n Spalten von Q ist) und als *Kompromiss* der Vorschlag, das Teilsystem

$$\hat{R}w = Q_1^T y$$

nach w zu lösen und in Kauf zu nehmen, dass der Rest, nämlich das $Q_2^T y$, nicht notwendigerweise gleich null ist.

Wir halten zunächst mal fest, dass

- Obwohl Q eine reguläre Matrix ist, bedarf der Übergang von $Aw = y$ zu $Q^T Aw = Q^T y$ einer genaueren Analyse.
- Wir bemerken, dass für eine hypothetische komplette Lösung $Aw - y$, diese Transformation keine Rolle spielt.
- Für die Kompromisslösung jedoch schon, weil beispielsweise verschiedene Konstruktionen eines invertierbaren Teils, verschiedene Residuen bedeuten und somit Optimalität im Sinne von $\min_w \|Aw - y\|^2$ nicht garantiert ist.

Allerdings, wie Sie als Übungsaufgabe nachweisen werden, löst dieser Ansatz tatsächlich das Optimierungsproblem.

3.2 Singulärwertzerlegung

Eine weitere Matrix Zerlegung, die eng mit der Lösung von Optimierungsproblemen oder überbestimmten Gleichungssystemen zusammenhängt ist die *Singulärwertzerlegung* (SVD – von english: *Singular Value Decomposition*).

Theorem 3.2 (Singulärwertzerlegung)**.** *Sei $A \in \mathbb{C}^{m \times n}$, $m \geq n$. Dann existieren orthogonale Matrizen $U \in \mathbb{C}^{m \times m}$ und $V \in \mathbb{C}^{n \times n}$ und eine Matrix*

$\Sigma \in \mathbb{R}^{m\times n}$ *der Form*

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \sigma_n \\ 0 & 0 & \dots & 0 \\ \vdots & \ddots & & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}$$

mit reellen sogenannten Singulärwerten

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

sodass gilt

$$A = U\Sigma V^*$$

wobei gilt $V^* = \overline{V^T}$ *(transponiert und komplex konjugiert).*

Ein paar Bemerkungen.

- Ist A reell, können auch U und V reell gewählt werden.
- Die Annahme $m \geq n$ war nur nötig um für die Matrix Σ keine Fallunterscheidung zu machen. (Für $m \leq n$ "steht der Nullblock rechts von den Singulärwerten"). Insbesondere gilt $A^* = V\Sigma U^*$ ist eine SVD von A^*.
- Eine Illustration der Zerlegung ist hier zu sehen.

Wir machen einige Überlegungen im Hinblick auf große Matrizen. Sei dazu $m > n$, $A \in \mathbb{C}^{m\times n}$ und $A = U\Sigma V^*$ eine SVD wie in Theorem 3.2. Sei nun

$$U = \begin{bmatrix} U_1 & U_2 \end{bmatrix}$$

partitioniert sodass U_1 die ersten n Spalten von U enthält.

Dann gilt (nach der Matrix-Multiplikations Regel *Zeile mal Spalte* die Teile U_2 und V_2 immer mit dem Nullblock in Σ multipliziert werden) dass

$$A = U\Sigma V = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \hat{\Sigma} \\ 0 \end{bmatrix} V^* = U_1 \hat{\Sigma} V^*$$

Es genügt also nur die ersten m Spalten von U zu berechnen. Das ist die sogenannte **slim SVD**.

Hat, darüberhinaus, die Matrix A keinen vollen Rang, also $\mathrm{Rg}(A) = r < n$, dann:

- ist $\sigma_i = 0$, für alle $i = r+1, \dots, n$, (wir erinnern uns, dass die Singulärwerte nach Größe sortiert sind)
- die Matrix $\hat{\Sigma}$ hat $n - r$ Nullzeilen
- für die Zerlegung sind nur die ersten r Spalten von U und V relevant – die sogenannte **Kompakte SVD**.

In der Datenapproximation ist außerdem die **truncated SVD** von Interesse. Dazu sei $\hat{r} < r$ ein beliebig gewählter Index. Dann werden alle Singulärwerte, $\sigma_i = 0$, für alle $i = \hat{r} + 1, \ldots, n$, abgeschnitten – das heißt null gesetzt und die entsprechende *kompakte SVD* berechnet.

Hier gilt nun nicht mehr die Gleichheit in der Zerlegung. Vielmehr gilt

$$A \approx A_{\hat{r}}$$

wobei $A_{\hat{r}}$ aus der *truncated SVD* von A erzeugt wurde. Allerdings ist diese Approximation von A durch optimal in dem Sinne, dass es keine Matrix vom Rang $\hat{r} \geq r = \mathrm{Rg}(A)$ gibt, die A (in der *induzierten* euklidischen Norm[1]) besser approximiert. Es gilt

$$\min_{B \in \mathbb{C}^{m \times n}, \mathrm{Rg}(B) = \hat{r}} \|A - B\|_2 = \|A - A_{\hat{r}}\|_2 = \sigma_{\hat{r}+1};$$

(vgl. Satz 14.15, Bollhöfer and Mehrmann 2004).

Zum Abschluss noch der Zusammenhang zum Optimierungsproblem. Ist $A = U\Sigma V^*$ "SV-zerlegt", dann gilt

$$A^*Aw = V\Sigma^*U^*U\Sigma V^*w = V\hat{\Sigma}^2V^*$$

und damit

$$A^*Aw = A^*y \quad \Leftrightarrow \quad V\hat{\Sigma}^2V^* = V\Sigma^*U^*y \quad \Leftrightarrow \quad w = V(\Sigma^+)^*U^*y$$

wobei

$$\Sigma^+ = \begin{bmatrix} \hat{\Sigma}^{-1} \\ 0_{m-n \times n} \end{bmatrix}$$

aus $\Sigma\hat{\Sigma}^{-1}\hat{\Sigma}^{-1}$ herrührt.

Bemerkung: Σ^+ kann auch definiert werden, wenn $\hat{\Sigma}$ nicht invertierbar ist (weil manche Diagonaleinträge null sind). Dann wird $\hat{\Sigma}^+$ betrachtet, bei welcher nur die $\sigma_i > 0$ invertiert werden und die anderen $\sigma_i = 0$ belassen werden. Das definiert eine sogenannte *verallgemeinerte Inverse* und löst auch das Optimierungsproblem falls A keinen vollen Rang hat.

3.3 Aufgaben

Erklärung: (T) heißt theoretische Aufgabe, (P) heißt programmieren.

3.3.1 Norm und Orthogonale Transformation (T)

Sei $Q \in \mathbb{R}^{n \times n}$ eine orthogonale Matrix und sei $y \in \mathbb{R}^n$. Zeigen Sie, dass

$$\|y\|^2 = \|Qy\|^2$$

gilt. (2 Punkte)

[1] Auf Matrixnormen kommen wir noch in der Vorlesung zu sprechen.

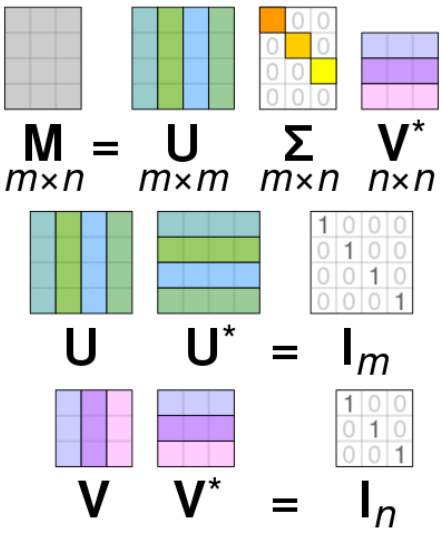


Figure 3.1: Illustration der SVD. Bitte beachten der $*$ bedeutet hier transponiert und komplex konjugiert. By Cmglee - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=67853297

3.3.2 Kleinste Quadrate und Mittelwert

Zeigen sie, dass der *kleinste Quadrate* Ansatz zur Approximation einer Datenwolke

$$(x_i, y_i), \quad i = 1, 2, \dots, N,$$

mittels einer konstanten Funktion $f(x) = w_1$ auf w_1 auf den Mittelwert der y_i führt. (6 Punkte)

3.3.3 QR Zerlegung und Kleinstes Quadrate Problem (T)

Sei $A \in \mathbb{R}^{m,n}$, $m > n$, A hat vollen Rank und sei

$$[Q_1 \quad Q_2] \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} = A$$

eine QR-Zerlegung von A. Zeigen sie, dass die Lösung von

$$\hat{R}w = Q_1^T y$$

ein kritischer Punkt (d.h. der Gradient ∇_w verschwindet) von

$$w \mapsto \frac{1}{2}\|Aw - y\|^2$$

ist. **Hinweis**: Die Formel für den Gradienten wurde in der Vorlesung 02 hergeleitet. (6 Punkte)

3.3.4 Eigenwerte Symmetrischer Matrizen (T)

Zeigen Sie, dass Eigenwerte symmetrischer reeller Matrizen $A \in \mathbb{R}^{n \times n}$ immer reell sind. (3 Punkte)

3.3.5 Singulärwertzerlegung und Eigenwerte (T)

Zeigen Sie, dass die quadrierten Singulärwerte einer Matrix $A \in \mathbb{R}^{m \times n}$, $m > n$, genau die Eigenwerte der Matrix $A^T A$ sind und in welcher Beziehung sie mit den Eigenwerten von AA^T stehen. **Hinweis**: hier ist "$m > n$" wichtig. (6 Punkte)

3.3.6 Python – Laden und Speichern von Arrays

Speichern sie die Covid-Daten aus obiger Tabelle zur späteren Verwendung als ein `numpy.array`. Beispielsweise so:

```
import numpy as np
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
        12, 13, 14, 15,  16, 17, 18, 19,  20,   21,
        22,  23,  24,   25,    26, 27,  28,   29,    30,     31]
case = [352, 347, 308, 151, 360, 498, 664, 686, 740, 418, 320,
        681, 691, 1154, 1284,  127, 984, 573, 1078,  1462,   2239,
        2236,  2119,  1663,   1413,    2283,
        2717,  3113,   2972,    3136,    2615]

data = np.vstack([days, case])
print('Shape of data: ', data.shape)

datafilestr = 'coviddata.npy'
np.save(datafilestr, data)

lddata = np.load(datafilestr)

plt.figure(1)
plt.plot(lddata[0, :], lddata[1, :], 's', label='Cases/Day')
plt.title('Covid Faelle in Bayern im Oktober 2020')
plt.legend()
plt.show()
```

3.3.7 Lineare Regression für Covid Daten (P)

Führen sie auf den Covid-daten eine lineare Regression zum Fitten

- einer konstanten Funktion $f(x) = c$

- einer linearen Funktion $f(x) = ax + b$
- einer quadratischen Funktion $f(x) = w_1 + w_2 x + w_3 x^2$

durch. Berechnen Sie die mittlere quadratische Abweichung $\frac{1}{N}\sum_{i=1}^{N} \|y_i - f(x_i)\|^2$ für alle drei Approximationen und plotten Sie die Covid-Daten zusammen mit der aus der Regression erhaltenen Funktion. Beispielsweise so:

```
import numpy as np
import matplotlib.pyplot as plt

datafilestr = 'coviddata.npy'
cvdata = np.load(datafilestr)

# ## Definieren der Basis Funktion(en)


def b_zero(x):
    ''' eine konstante Funktion '''
    return 1.


ybf = cvdata[1, :]  # die y-werte
xbf = cvdata[0, :]  # die x-werte

N = xbf.size   # Anzahl Datenpunkte
ybf = ybf.reshape((N, 1))  # reshape als Spaltenvektor

bzx = [b_zero(x) for x in xbf]  # eine Liste mit Funktionswerten
bzx = np.array(bzx).reshape((N, 1))  # ein Spalten Vektor

Phix = bzx  # hier nur eine Spalte

# Das LGS AtA w = At y
rhs = Phix.T @ ybf
AtA = Phix.T @ Phix

# ACHTUNG: das hier geht nur weil AtA keine Matrix ist in diesem Fall
w = 1./AtA * rhs
# ACHTUNG: das hier ging nur weil AtA keine Matrix ist in diesem Fall


def get_regfunc(weights, basfunlist=[]):
    ''' Eine Funktion, die eine Funktion erzeugt

    Eingang: die Gewichte, eine Liste von Basisfunktionen
    Ausgang: die entsprechende Funktion `f = w_1b_1 + ... `
```

```
    '''
    def regfunc(x):
        fval = 0
        for kkk, basfun in enumerate(basfunlist):
            fval = fval + weights[kkk]*basfun(x)
        return fval

    return regfunc


const_regfunc = get_regfunc([w], [b_zero])

const_approx = [const_regfunc(x) for x in xbf]
const_approx = np.array(const_approx).reshape((N, 1))

plt.figure(1)
plt.plot(cvdata[0, :], cvdata[1, :], 's', label='Cases/Day')
plt.plot(cvdata[0, :], const_approx, '-', label='constant fit')
plt.legend()
plt.show()
```

Hinweise:

- `liste = [f(x) for x in iterable]` ist sehr bequem um Vektoren von Funktionswerten zu erzeugen aber nicht sehr *pythonesque* (und auch im Zweifel nicht effizient). Besser ist es Funktionen zu schreiben, die *vektorisiert* sind. Zum Beispiel können die meisten *built-in* Funktionen wie `np.exp` ein `array` als Eingang direkt in ein `array` der Funktionswerte umsetzen.
- *Eine Funktion, die eine Funktion erzeugt* finde ich sehr hilfreich für viele Anwendungen (ist aber manchmal nicht so gut nachvollziehbar).

3.3.8 Truncated SVD (P+T)

1. **(P)** Berechnen und plotten sie die Singulärwerte einer 4000×1000 Matrix mit zufälligen Einträgen und die einer Matrix mit "echten" Daten (hier Simulationsdaten einer Stroemungssimulation)[2]. Berechnen sie den Fehler der *truncated SVD* $\|A - A_{\hat{r}}\|$ für $\hat{r} = 10, 20, 40$ für beide Matrizen.
2. **(T)** Was lässt sich bezüglich einer Kompression der Daten mittels SVD für die beiden Matrizen sagen. (Vergleichen sie die plots der Singulärwerte und beziehen sie sich auf die gegebene Formel für die Differenz).
3. **(P+T)** Für die "echten" Daten: Speichern sie die Faktoren der bei $\hat{r} = 40$ abgeschnittenen SVD und vergleichen Sie den Speicherbedarf der Faktoren und der eigentlichen Matrix.

[2] Download bitte hier – Achtung das sind 370MB

Beispielcode:

```
import numpy as np
import scipy.linalg as spla
import matplotlib.pyplot as plt

randmat = np.random.randn(4000, 1000)

rndU, rndS, rndV = spla.svd(randmat)

print('U-dims: ', rndU.shape)
print('V-dims: ', rndV.shape)
print('S-dims: ', rndS.shape)

plt.figure(1)
plt.semilogy(rndS, '.', label='Singulaerwerte (random Matrix)')

realdatamat = np.load('velfielddata.npy')

# # Das hier ist eine aufwaendige Operation
rlU, rlS, rlV = spla.svd(realdatamat, full_matrices=False)
# # auf keinen Fall `full_matrices=False` vergessen

print('U-dims: ', rlU.shape)
print('V-dims: ', rlV.shape)
print('S-dims: ', rlS.shape)

plt.figure(1)
plt.semilogy(rlS, '.', label='Singulaerwerte (Daten Matrix)')

plt.legend()
plt.show()
```

Hinweise:

- Es gibt viele verschiedene Normen für Vektoren und Matrizen. Sie dürfen einfach mit `np.linalg.norm` arbeiten. Gerne aber mal in die Dokumentation schauen *welche* Norm berechnet wird.
- Die (T) Abschnitte hier bitte mit den anderen (T) Aufgaben oder als Bildschirmausgabe im Programm.

Chapter 4

Hauptkomponentenanalyse

Während in den vorherigen Kapiteln der Versuch war, einen funktionalen Zusammenhang in Daten zu bekommen, geht es jetzt darum, statistische Eigenschaften aus Daten zu extrahieren. Wir werden sehen, dass das auch nur ein anderer Blickwinkel auf das gleiche Problem *Wie können wir die Daten verstehen?* ist und auch die SVD wieder treffen.

Weil ich den Ansatz gerne *ad hoc* also am Problem entlang motivieren und einführen will, vorweg schon mal die bevorstehenden Schritte

1. Zentrierung/Skalierung der Daten
2. Berechnung der Varianzen im Standard Koordinatensystem
3. Überlegung, dass Daten in einem anderen Koordinatensystem eventuell besser dargestellt werden im Sinne
4. Berechnung eines optimalen Koordinatenvektors mittels SVD.

Wir nehmen noch einmal die Covid-Daten her, vergessen kurz, dass es sich um eine Zeitreihe handelt und betrachten sie als Datenpunkte (x_i, y_i), $i = 1, \dots, N$, im zweidimensionalen Raum mit Koordinaten x und y.

Als erstes werden die Daten **zentriert** indem in jeder Komponente der Mittelwert

$$x_c = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad y_c = \frac{1}{N}\sum_{i=1}^{N} y_i.$$

abgezogen wird und dann noch mit dem inversen des Mittelwerts skaliert.

Also, die Daten werden durch $(\frac{x_i-\bar{x}}{\bar{x}}, \frac{y_i-\bar{y}}{\bar{y}})$ ersetzt.

4.1 Variationskoeffizienten

Als nächstes kann Jan sich fragen, wie gut die Daten durch ihren Mittelwert beschrieben werden und die Varianzen berechnen, die für zentrierte Daten so

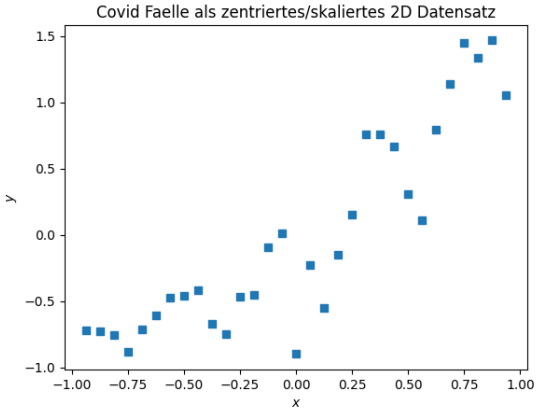


Figure 4.1: Fallzahlen von Sars-CoV-2 in Bayern im Oktober 2020 – zentriert

aussehen

$$s_x^2 = \frac{1}{N-1}\sum_{i=1}^{N} x_i^2, \quad s_y^2 = \frac{1}{N-1}\sum_{i=1}^{N} y_i^2.$$

Im gegebenen Fall bekommen wir

$$s_x^2 \approx 0.32 \quad s_y^2 \approx 0.57$$

und schließen daraus, dass in y Richtung *viel passiert* und in x Richtung *nicht ganz so viel*. Das ist jeder Hinsicht nicht befriedigend, wir können weder

- Redundanzen ausmachen (eine Dimension der Daten vielleicht weniger wichtig?) noch
- dominierende Richtungen feststellen (obwohl dem Bild nach so eine offenbar existiert)

und müssen konstatieren, dass die Repräsentation der Daten im (x, y) Koordinatensystem nicht optimal ist.

Die Frage ist also, ob es ein Koordinatensystem gibt, dass die Daten besser darstellt.

> Ein Koordinatensystem ist nichts anderes als eine Basis. Und die Koordinaten eines Datenpunktes sind die Komponenten des entsprechenden Vektors in dieser Basis. Typischerweise sind Koordinatensysteme orthogonal (das heißt eine Orthogonalbasis) und häufig noch orientiert (die Basisvektoren haben eine bestimmte Reihenfolge und eine bestimmte Richtung).

4.2 Koordinatenwechsel

Sei nun also $\{b_1, b_2\} \subset \mathbb{R}^2$ eine orthogonale Basis.

> Wie allgemein gebräuchlich, sagen wir *orthogonal*, meinen aber *orthonormal*. In jedem Falle soll gelten
>
> $$b_1^T b_1 = 1, \quad b_2^T b_2 = 1, \quad b_1^T b_2 = b_2^T b_1 = 0.$$

Wir können also alle Datenpunkte $\mathbf{x}_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$ in der neuen Basis darstellen mit eindeutig bestimmten Koeffizienten α_{i1} und α_{i2} mittels

$$\mathbf{x}_i = \alpha_{i1} b_1 + \alpha_{i2} b_2.$$

Für orthogonale Basen sind die Koeffizienten durch *testen* mit dem Basisvektor einfach zu berechnen:

$$\begin{aligned} b_1^T \mathbf{x}_i &= b_1^T(\alpha_{i1} b_1 + \alpha_{i2} b_2) = \alpha_{i1} b_1^T b_1 + \alpha_{i2} b_1^T b_2 = \alpha_{i1} \cdot 1 + \alpha_{i2} \cdot 0 = \alpha_{i1}, \\ b_2^T \mathbf{x}_i &= b_2^T(\alpha_{i1} b_1 + \alpha_{i2} b_2) = \alpha_{i1} b_1^T b_2 + \alpha_{i2} b_2^T b_2 = \alpha_{i1} \cdot 0 + \alpha_{i2} \cdot 1 = \alpha_{i2}. \end{aligned}$$

Es gilt also

$$\alpha_{i1} = b_1^T \mathbf{x} = b_1^T \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \quad \alpha_{i2} = b_2^T \mathbf{x} = b_2^T \begin{bmatrix} x_i \\ y_i \end{bmatrix}.$$

Damit, können wir jeden Datenpunkt $\mathbf{x}_i = (x_i, y_i)$ in den neuen Koordinaten $(\alpha_{i1}, \alpha_{i2})$ ausdrücken.

Zunächst halten wir fest, dass auch in den neuen Koordinaten die Daten zentriert sind. Es gilt nämlich, dass

$$\begin{aligned} \frac{1}{N}\sum_{i=1}^N \alpha_{ji} = \frac{1}{N}\sum_{i=1}^N b_j^T \mathbf{x}_i = \frac{1}{N} b_j^T \sum_{i=1}^N \begin{bmatrix} x_i \\ y_i \end{bmatrix} &= \frac{1}{N} b_j^T \begin{bmatrix} \sum_{i=1}^N x_i \\ \sum_{i=1}^N y_i \end{bmatrix} \\ &= b_j^T \begin{bmatrix} \frac{1}{N}\sum_{i=1}^N x_i \\ \frac{1}{N}\sum_{i=1}^N y_i \end{bmatrix} = b_j^T \begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0, \end{aligned}$$

für $j = 1, 2$.

Desweiteren gilt wegen der Orthogonalität von $B = [b_1\ b_2] \in \mathbb{R}^{2\times 2}$, dass

$$x_i^2 + y_i^2 = \|\mathbf{x}_i\|^2 = \|B^T \mathbf{x}_i\|^2 = \| \begin{bmatrix} b_1^T \\ b_2^T \end{bmatrix} \mathbf{x}_i \|^2 = \| \begin{bmatrix} b_1^T \mathbf{x} \\ b_2^T \mathbf{x} \end{bmatrix} \|^2 = \| \begin{bmatrix} \alpha_{i1} \\ \alpha_{i2} \end{bmatrix} \|^2 = \alpha_{i1}^2 + \alpha_{i2}^2$$

woraus wir folgern, dass in jedem orthogonalen Koordinatensystem, die Summe der beiden Varianzen die gleiche ist:

$$s_x^2 + s_y^2 = \frac{1}{N-1}\sum_{i=1}^N (x_i^2 + y_i^2) = \frac{1}{N-1}\sum_{i=1}^N (\alpha_{i1}^2 + \alpha_{i2}^2) =: s_1^2 + s_2^2.$$

Das bedeutet, dass durch die Wahl des Koordinatensystems die Varianz als Summe nicht verändert wird. Allerdings können wir das System so wählen, dass eine der Varianzen in Achsenrichtung maximal wird (und die übrige(n) entsprechend klein).

Analog gilt für den eigentlichen Mittelwert der (nichtzentrierten) Daten, dass die Norm gleich bleibt. In der Tat, für die *neuen* Koordinaten des Mittelwerts gilt in der Norm

$$\| \begin{bmatrix} \alpha_{c1} \\ \alpha_{c2} \end{bmatrix} \| = \| B^T \begin{bmatrix} x_c \\ y_c \end{bmatrix} \| = \| \begin{bmatrix} x_c \\ y_c \end{bmatrix} \|.$$

4.3 Maximierung der Varianz in (Haupt)-Achsenrichtung

Wir wollen nun also $b_1 \in \mathbb{R}^2$, mit $\|b_1\| = 1$ so wählen, dass

$$s_1^2 = \frac{1}{N-1} \sum_{i=1}^{n} \alpha_{i1}^2$$

maximal wird. Mit der Matrix $\mathbf{X}$ aller Daten

$$\mathbf{X} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_N & y_N \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \in \mathbb{R}^{N \times 2}$$

können wir die Varianz in b_1-Richtung kompakt schreiben als

$$s_1^2 = \frac{1}{N-1} \sum_{i=1}^{n} \alpha_{i1}^2 = \frac{1}{N-1} \sum_{i=1}^{n} (b_1^T \mathbf{x}_i)^2 = \frac{1}{N-1} \sum_{i=1}^{n} (\mathbf{x}_i^T b_1)^2 = \frac{1}{N-1} \|\mathbf{X} b_1\|^2$$

Wir sind also ein weiteres mal bei einem Optimierungsproblem (diesmal mit Nebenbedingung) angelangt:

$$\max_{b \in \mathbb{R}^2, \|b\|=1} \|\mathbf{X} b\|^2$$

Die Lösung dieses Problems ist mit $b = v_1$ gegeben, wobei v_1 der erste (rechte) Singulärvektor von $\mathbf{X}$ ist:

$$\mathbf{X} = U \Sigma V^T = U \Sigma \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix}.$$

Und damit rechnen wir auch direkt nach, dass im neuen Koordinatensystem $\{b_1, b_2\} = \{v_1, v_2\}$ die Varianzen s_1^2 und s_2^2 (bis auf einen Faktor von $\frac{1}{N-1}$) genau die quadrierten Singulärwerte von $\mathbf{X}$ sind:

$$(N-1)s_1^2 = \|\mathbf{X} v_1\|^2 = \|U \Sigma \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} v_1\|^2 = \|\Sigma \begin{bmatrix} v_1^T v_1 \\ v_2^T v_1 \end{bmatrix}\|^2 = \|\Sigma \begin{bmatrix} 1 \\ 0 \end{bmatrix}\|^2 = \sigma_1^2,$$

$$(N-1)s_2^2 = \|\mathbf{X} v_2\|^2 = \|U \Sigma \begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} v_2\|^2 = \|\Sigma \begin{bmatrix} v_1^T v_2 \\ v_2^T v_2 \end{bmatrix}\|^2 = \|\Sigma \begin{bmatrix} 0 \\ 1 \end{bmatrix}\|^2 = \sigma_2^2$$

Für unser Covid Beispiel ergibt sich

$$V^T \approx \begin{bmatrix} 0.5848 & 0.8111 \\ 0.8111 & -0.5848 \end{bmatrix}$$

also

$$b_1 = v_1 = \begin{bmatrix} 0.5848 \\ 0.8111 \end{bmatrix} \quad b_2 = v_2 = \begin{bmatrix} 0.8111 \\ -0.5848 \end{bmatrix}$$

als neue Koordinatenrichtungen mit

$$s_1^2 \approx 0.85, \quad s_2^2 \approx 0.04,$$

was bereits eine deutliche Dominanz der v_1-Richtung – genannt *Hauptachse* – zeigt.

Im Hinblick auf die nächste Vorlesung, in der wir Anwendungen und Eigenschaften der PCA untersuchen werden, noch ein Plot der Daten mit der v_1-Richtung als Linie eingezeichnet.

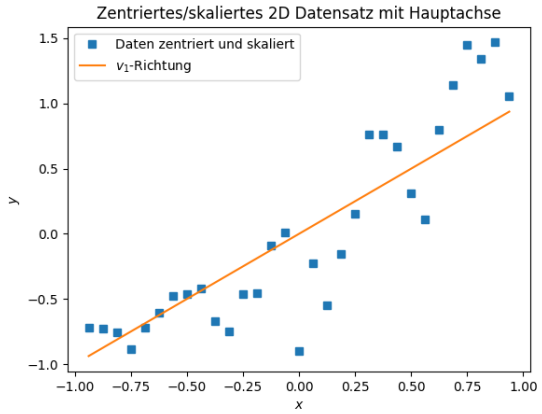


Figure 4.2: Fallzahlen von Sars-CoV-2 in Bayern im Oktober 2020 – zentriert/skaliert/Hauptachse

Chapter 5

Hauptkomponentenanalyse Ctd.

Im vorherigen Kapitel hatten wir einen zweidimensionalen Datensatz betrachtet und dafür die Richtung bestimmt, in der die Varianz maximal wird. Da wir außerdem ermittelt hatten, dass die Summe der Varianz unabhüngig vom Koordinatensystem ist (es muss nur ein orthogonales sein) bedeutete das gleichzeitig, dass die verbleibende Richtung die Richtung der minimalen Varianz war.

Jetzt wollen wir einen Datensatz mit mehr Merkmalen betrachten und sehen, wie die algorithmische Herangehensweise zum Verständnis beiträgt.

5.1 Der PENGUINS Datensatz

Die Grundlage ist der *Pinguin Datensatz*, der eine gern genommene Grundlage für die Illustration in der Datenanalyse ist. Die Daten wurden von Kristen Gorman erhoben und beinhalten 4 verschiedene Merkmale (engl. *features*) von einer Stichprobe von insgesamt 344[1]Pinguinen die 3 verschiedenen Spezies zugeordnet werden können oder sollen (Fachbegriff hier: *targets*). Im Beispiel werden die Klassen mit `0, 1, 2` codiert und beschreiben die Unterarten *Adele*, *Gentoo* und *Chinstrap* der Schwertlilien. Die Merkmale sind gemessene Länge und Höhe des Schnabels (hier *bill*), die Länge der Flosse (*flipper*) sowie das Köpergewicht[2] (*body mass*).

Wir stellen uns 2-3 Fragen:

1. Würden eventuell 3 (oder sogar nur 2) Dimensionen reichen um den Datensatz zu beschreiben?

[1]allerdings mit 2 unvollständigen Datenpunkten, die ich entfernt habe für unseere Beispiele
[2]Im Originaldatensatz ist das Gewicht in Gramm angegeben, um die Daten innerhalb einer 10er Skala zu haben, habe ich das Gewicht auf in kg umgerechnet

2. Können wir aus den Merkmalen (*features*) die Klasse (*target*) erkennen und wie machen wir gegebenenfalls die Zuordnung?

5.2 Darstellung

In höheren Dimensionen ist schon die graphische Darstellung der Daten ein Problem. Wir können aber alle möglichen 2er Kombinationen der Daten in 2D plots visualisieren.

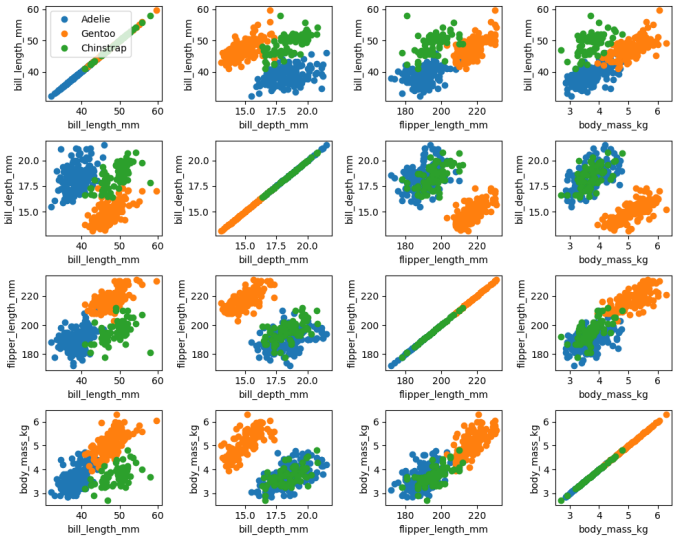


Figure 5.1: Pinguin Datenset 2D plots

Ein Blick auf die Diagonale zeigt schon, dass manche Merkmale besser geeignet als andere sind, um die Spezies zu unterscheiden, allerdings keine der Zweierkombinationen eine Eindeutige Diskriminierung erlaubt.

5.3 Korrelationen und die Kovarianzmatrix

Als nächstes suchen wir nach Korrelationen in den Daten in dem wir für alle Merkmalpaare die Korrelationen ausrechnen. Dafür berechnen wir die sogenannte *Kovarianzmatrix*

$$\mathrm{Cov}(\mathbf{X}) = \left[\rho_{\mathbf{x}_i\mathbf{x}_j}\right]_{i,j=1,\dots n} \in \mathbb{R}^{n\times n}$$

wobei n die Dimension der Daten ist und wobei

$$s_{\mathbf{x}_i\mathbf{x}_j} = \frac{1}{N-1}\sum_{k=1}^{N}(x_{ik}-\overline{\mathbf{x}_i})(x_{jk}-\overline{\mathbf{x}_j}).$$

die Kovarianzen sind, die wir auch schon in der ersten Vorlesung kennengelernt haben.

Ist $\mathbf{X} \in \mathbb{R}^{N\times n}$ die Matrix mit den Datenvektoren $\mathbf{x}_i \in \mathbb{R}^n$ als Spalten **und ist der Datensatz zentriert** so erhalten wir die Kovarianzmatrix als

$$\mathrm{Cov}(\mathbf{X}) = \frac{1}{N-1}\mathbf{X}^T\mathbf{X}$$

Wir bemerken, dass auf der Hauptdiagonalen die Varianzen in Koordinatenrichtung stehen und in den Zeilen oder Spalten ein Mass dafür, wie bspw. $\mathbf{x}_i$ und $\mathbf{x}_j$ korreliert sind. Große Zahlen bedeuten eine große Varianz oder eine starke Korrelation und umgekehrt. Für die Datenanalyse können wir $\mathrm{Cov}(\mathbf{X})$ wie folgt heranziehen:

5.4 Hauptachsentransformation

Wie im vorherigen Kapitel hergeleitet, bedeutet ein Koordinatenwechsel die Multiplikation der Datenmatrix $\mathbf{X} \in \mathbb{R}^{N\times n}$ mit einer orthogonalen Matrix $V \in \mathbb{R}^{n\times n}$:

$$\tilde{\mathbf{X}} = \mathbf{X}V,$$

wobei $\tilde{\mathbf{X}}$ die Daten in den neuen Koordinaten sind (die Basisvektoren sind dann die Zeilenvektoren von V). Wir wollen nun eine Basis finden, in der

- $\mathrm{Cov}(\mathbf{X})$ eine Diagonalmatrix ist — damit wären alle Richtungen in den Daten *unkorreliert* und könnten *unabhängig* voneinander betrachtet werden[3] –
- und in der die neuen Varianzen nach Größe geordnet sind gleichzeitig die verfügbare Varianz maximal in wenigen Richtungen konzentrieren.

Nachden den Überlegungen im vorherigen Kapitel ist die erste Hauptrichtung durch den ersten rechten Singulärvektor $v_1 \in \mathbb{R}^n$ der (*ökonomischen*) Singulärwertzerlegung

$$\mathbf{X} = U\Sigma V^* = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix} \begin{bmatrix} v_1^* \\ v_2^* \\ \vdots \\ v_n^* \end{bmatrix}$$

von $\mathbf{X} \in \mathbb{R}^{N\times n}$ gegeben. Die zugehörigen Koeffizenten berechnen wir mittels

$$\tilde{\mathbf{X}} = \mathbf{X}v_1.$$

[3] wir dürfen aber nicht vergessen, dass Daten typischerweise nur eine Stichprobe von Beobachtungen eines Phänomens sind. Die Unabhängigkeit in den *features* gilt also nur für die gesammelten Daten aber in der Regel nicht für das Phänomen. Für normalverteilte Prozesse liefern die daten-basiert ermittelten Hauptrichtungen jedoch auch die Hauptrichtungen des zugrundeliegenden Phänomens

5.5 Rekonstruktion

Um zu plausibilisieren, dass die weiteren Hauptachsen durch die weiteren (rechten) Singulärvektoren gegeben sind, betrachten wir erst die *Rekonstruktion* also die Darstellung im Ausgangskoordinatensystem (mit den messbaren oder interpretierbaren Features) die durch

$$\tilde{\tilde{\mathbf{X}}} = \mathbf{X} v_1 v_1^T$$

gegeben ist. Die letzte Formel wird vielleicht klarer, wenn Jan sich überlegt, dass für einen Datenpunkt $\mathbf{X}_j = \begin{bmatrix} x_{1j} & x_{2j} & ... & x_{nj} \end{bmatrix}$ (also die j-te Zeile von $\mathbf{X}$), der Koeffizient für v_1 gegeben ist durch $\alpha_j = v_1^T \mathbf{X}_j^T$ und die Darstellung im Vektorraum durch

$$\tilde{\tilde{\mathbf{X}}}_j = (\alpha_j v_1)^T = \alpha_j v_1^T = (v_1^T \mathbf{X}_j^T) v_1^T = \mathbf{X}_j v_1 v_1^T$$

gegeben ist.

Damit können wir mit

$$\mathbf{X} - \tilde{\tilde{\mathbf{X}}} = \mathbf{X}(I - v_1 v_1^T)$$

den Teil der Daten betrachten, der durch die Richtung v_1 nicht abgebildet wird (sozusagen den noch verbleibenden Teil). Jan kann nachrechnen, dass wiederum U und V die Singulärvektoren von $\mathbf{X} - \tilde{\tilde{\mathbf{X}}}$ bilden, mit jetzt v_2 als Richtung mit dem größten (verbleibenden) Singulärwert.

Das heißt, dass der k-te Singulärvektor die k-te Hauptrichtung bildet.

5.6 Reduktion der Daten

Ist die Datenmatrix mit linear abhängigen Spalten besetzt, äußert sich das in $\sigma_k = 0$ für ein $k < n$ (und alle noch folgenden Singulärwerte). Jan rechnet nach, dass dann

$$\mathbf{X} = \mathbf{X} V_{k-1} V_{k-1}^T$$

wobei V_{k-1} die Matrix der $k-1$ führenden Singulärvektoren ist und das entsprechende

$$\tilde{\mathbf{X}} = \mathbf{X} V_{k-1}$$

alle Informationen des Datensatzes in kleinerer Dimension parametrisiert.

In der Praxis sind schon allein durch Messfehler exakte lineare Abhängigkeiten sowie durch die näherungsweise Bestimmung auf dem Computer das Auftreten von $\sigma_k = 0$ quasi ausgeschlossen. Stattdessen werden Schwellwerte definiert, unterhalb derer die SVD abgeschnitten wird.

Wie oben, werden dann die Daten $\tilde{\mathbf{X}} = \mathbf{X} V_{\hat{k}}$ in den (reduzierten) Koordinaten der Hauptachsen betrachtet sowie die Rekonstruktion $\hat{\mathbf{X}} = \mathbf{X} V_{\hat{k}} V_{\hat{k}}^T$, wobei der $V_{\hat{k}}$ die Matrix der $\hat{k}$ führenden Singulärvektoren sind (mit Singulärwerten überhalb des Schwellwertes).

5.7 Am Beispiel der Pinguin Daten

Wir stellen die Daten noch einmal zentriert dar:

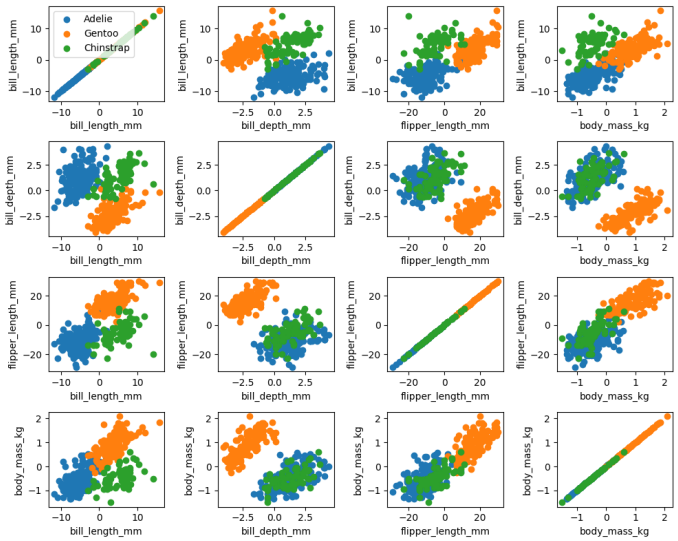


Figure 5.2: Pinguin Daten Zentriert

Für die Kovarianzmatrix der Pinguin Daten erhalten wir:

```
Covariance matrix:
[[ 29.8071  -2.5342  50.3758   2.6056]
 [ -2.5342   3.8998 -16.213   -0.7474]
 [ 50.3758 -16.213  197.7318   9.8244]
 [  2.6056  -0.7474   9.8244   0.6431]]
```

Für die Singulärwerte:

```
In: U, S, Vh = np.linalg.svd(data, full_matrices=False)

In: print(S)
Out: [269.7858  74.1334  28.4183   7.2219]
```

Zwar ist hier kein Singulärwert nah an der Null, allerdings beträgt der Unterschied zwischen dem größten und dem kleinsten schon eine gute 10er Potenz, was auf eine starke Dominanz der ersten Hauptachsen hinweist.

In der Tat, plotten wir die Daten in den Koordinaten der Hauptachsen (also $\tilde{\mathbf{X}}$), zeigen Bilder der v_1 oder v_2-Koordinaten klare Tendenzen in den Daten, während die Plots der übrigen Richtungen wie eine zufällige Punktwolke aussieht, vgl. die Abbildung hier.

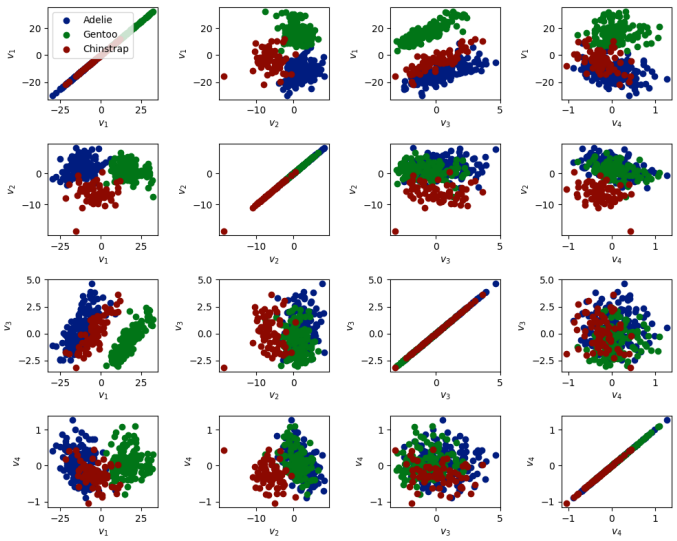


Figure 5.3: Pinguin Datenset 2D plots in Hauptachsenkoordinaten

Als letztes plotten wir noch $\hat{\mathbf{X}}$ für $\hat{k} = 2$. Das heißt wir reduzieren die Daten auf die v_1 und v_2-Richtungen und betrachten die Rekonstruktion. Im Plot sehen wir, dass in gewissen Teilen die Daten gut rekonstruiert werden. Allerdings, hat $\hat{\mathbf{X}}$ nur Rang $\operatorname{Rk} \hat{\mathbf{X}} = 2$ (warum?), sodass in den Plots notwendigerweise direkte lineare Abhängigkeiten offenbar werden.

5.8 Aufgaben

5.8.1 Kovarianzen (P+T)

Erzeugen Sie für `N` aus `Nl = [10, 1000, 100000]` zufällige Vektoren `x` und `y` der Länge `N` und berechnen Sie für den Datensatz `X = [x, x, x*y, y]` jeweils die Matrix der Korrelationskoeffizienten

$$\rho_{ij} = \frac{s_{ij}}{s_i \cdot s_j}$$

wobei s_{ij} die Kovarianz der Daten $\mathbf{x}_i$ und $\mathbf{x}_j$ ist und s_i die Varianz von $\mathbf{x}_i$. Interpretieren sie die Ergebnisse.

Hinweis: Weil Zufälligkeit involviert ist, ist die Interpretation manchmal schwierig. Lassen sie das Programm öfter laufen und beobachten sie verschiedene Realisierungen der Stichproben.

```
import numpy as np
```

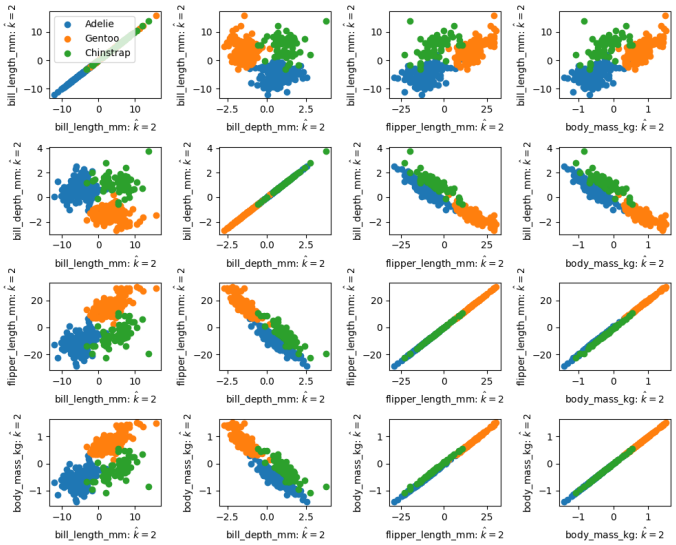


Figure 5.4: Pinguin Datenset – rekonstruiert von $\hat{k} = 2$ Hauptachsenkoordinaten

```
# N = 10

x = np.random.randn(10, 1)
y = np.random.randn(10, 1)

X = np.hstack([x, x, x*y, y])

Xcntrd = X - X.mean(axis=0)
# zentrieren der Daten

# Kovarianz Matrix ausrechnen
covX = 1/(10-1)*Xcntrd.T @ Xcntrd

# Varianzen -- stehen auf der Diagonalen im Quadrat
varvec = np.sqrt(np.diagonal(covX)).reshape(4, 1)
# Matrix mit den Kombinationen aller Varianzen
varmat = varvec @ varvec.T

matcc = covX * 1/varmat

print(f'N={10} : Matrix of Correlation Coefficients=')
print(matcc)
```

5.8.2 Pinguin Datensatz – Targets Plotten (P)

Laden Sie die Pinguin Datensatz (hier als `json` file bereitgestellt) und plotten sie `bill_length` versus `flipper_length` für die drei Spezies `Adelie`, `Gentoo`, `Chinstrap` in drei separaten Grafiken.

```
import json

import numpy as np
import matplotlib.pyplot as plt

with open('penguin-data.json', 'r') as f:
    datadict = json.load(f)

print(datadict.keys())

data = np.array(datadict['data'])
target = np.array(datadict['target'])
feature_names = datadict['feature_names']
target_names = datadict['target_names']

print('target names: ', target_names)
print('feature names: ', feature_names)

fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(10, 3))

# # Ein Vektor der aus dem Target Vektor das target 0 raussucht
trgtidx_z = (target == 0)
# # Wird gleich benutzt um die Daten nach diesem target zu filtern

target_z_daten = data[trgtidx_z, :]

axs[0].plot(target_z_daten[:, 2], target_z_daten[:, 0],
            'o', label=target_names[0])

axs[0].legend()
axs[0].set_xlabel(feature_names[3])
axs[0].set_ylabel(feature_names[0])
axs[0].set_title(target_names[0])

plt.tight_layout()
plt.show()
```

Hier wurden die `python` Datentypen

- *list* – z.B. `datenpunkte = [1, 2, 3]`
- *array* – z.B. `datenmatrix = np.array(datenpunkte)`

- *dictionary* – z.B. `datadict = {'data': datenpunkte}`

verwendet, die alle für verschiedene Zwecke gerne benutzt werden. Z.B.

- Liste – als eine Sammlung von (möglicherweise total unterschiedlichen) Objekten, über die iteriert werden kann und die einfach zu erweitern ist
- `arrays` – Matrix/Vektor von Daten eines Typs, mit denen *gerechnet* werden kann
- `dictionaries` – ein *Lookup table*. Objekte können über einen Namen addressiert werden. Ich nehme sie gerne um Daten mit ihrem Namen zum Beispiel als `json` file zu speichern.

5.8.3 Pinguin Datensatz – 2D plots (P)

Erzeugen sie die Abbildung aller Merkmalpaare zum Beispiel über ein `4x4` Feld von subplots

```
fig, axs = plt.subplots(nrows=4, ncols=4, figsize=(10, 8))
```

erzeugen, in das sie mittels

```
axs[zeile, spalte].plot(xdaten, ydaten, 'o')
```

die plots fuer die einzelnen targets "eintragen" koennen. Bitte die Achsen beschriften (die `legend` ist nicht unbedingt notwendig).

5.8.4 Kovarianz (P)

Zentrieren Sie den Datensatz (zum Beispiel unter Verwendung der `numpy.mean` Funktion) und berechnen Sie die Kovarianzmatrix.

5.8.5 Hauptachsentransformation (P)

Berechnen sie Hauptachsen und stellen Sie die Daten in den Hauptachsenkoordinaten dar (wie in dieser Abbildung).

5.8.6 Kovarianzmatrix (T)

Zeigen sie, dass für zentrierte Datensätze $\mathbf{X} \in \mathbb{R}^{N \times n}$ gilt, dass

$$\mathrm{Cov}(X) = \frac{1}{N-1}\mathbf{X}^T\mathbf{X}.$$

5.8.7 Gesamtvarianz (T)

Zeigen sie, dass auch für $n > 2$ die Summe der Varianzen in orthogonalen Achsenrichtungen unabhängig von der Wahl des Koordinatensystems sind. (Vergleiche Kapitel 4.2 Koordinatenwechsel)

Chapter 6

Clustering und Hauptkomponentenanalyse

Mit der Hauptkomponentenanalyse haben wir eine Methode kennengelernt, Daten gemäß in Koordinaten darzustellen, die nach inhaltsschwere[1] absteigend sortiert sind. Die Konzentrierung der Varianz in den Hauptachsenrichtungen ermöglicht uns

1. die Hauptkomponenten zu ermitteln, die den Datensatz optimal in niedrigerer Dimension darstellen (wobei optimal hier bedeutet dass die Varianz in der Differenz $\mathbf{X} - \hat{\mathbf{X}}$ minimal ist)
2. und in diesem Sinne die Daten optimal auf niedrigere Dimensionen zu reduzieren.

> Die noch offene Frage war, ob wir mit universalen Methoden aus den Merkmalen (oder deren Kombination) auf die Spezies schliessen können.

Ein einfacher Blick auf die Plots der Pinguin Merkmale (Abbildung 5.2) läßt uns schließen, dass

- ein Merkmal (z.B. `bill_depth`) auf jeden Fall nicht ausreichend ist für eine Unterscheidung) aber
- zwei Merkmale (z.B. `bill_length` vs. `flipper_length`) die farbigen Punkte (also die Spezies) schon etwas im Raum separieren, während
- mehrere Merkmale die Datenwolken im höher-dimensionalen Raum eventuell noch besser separieren, dass aber schwerlich nachvollziehbar ist.

[1] wenn wir Inhalt mit Varianz gleich setzen

Im 2D Fall allerdings, würde Jan einen zusätzlich gefundenen Pinguin vermessen, den neuen Datenpunkt im Diagramm eintragen und dann schauen, in welchem Bereich er landet um daraus die Spezies abzuleiten. Alles mit reichlich Vorwissen (z.B. dass es 3 Spezies gibt und welche Merkmale die Unterscheidung am besten erlauben).

Für allgemeine Fälle wird die Identifikation der Bereiche (manchmal ist es sogar gar nicht klar wieviele Bereiche nötig oder zielführend sind) und die Zuweisung der Datenpunkte von sogenannten **clustering** Algorithmen übernommen.

6.1 Clustering im Allgemeinen

Das Ziel ist es in einer Datenwolke Bereiche (**cluster**) zu identifizieren, sodass die vorhandenen (oder auch neu hinzukommende) Datenpunkte anhand ihrer Merkmale den Bereichen zugeordnet (*klassifiziert*) werden können.

Je nach Anwendungsfall und Vorwissen kann diese Aufgabe verschieden definiert werden

- Ist die Anzahl der cluster bekannt (wie bei unseren Pinguinen wo es einfach 3 Arten zu unterscheiden gibt), soll der Datenraum optimal in entsprechend viele nicht überlappende Bereiche geteilt.
- Anderenfalls sollen die Anzahl der Cluster und die zugehörigen Bereiche simultan optimal bestimmt werden.

Typischerweise werden die cluster durch ihre Mittelpunkte (**centroids**) $c_j \in \mathbb{R}^n$ bestimmt, $j = 1, \dots, K$, wobei K die Anzahl der cluster ist sowie eine Zuweisungsregel $k\colon \mathbf{x}_i \mapsto \{1, \dots, K\}$ die zu einem Datenpunkt das entsprechende cluster aussucht. Die Zuweisung passiert generell so, dass einem Datenpunkt $\mathbf{x}_i$ der naheste Zentroid zugewiesen wird, also $k(\mathbf{x}_i)$ ist der Index j^*, sodass

$$\|\mathbf{x}_i - c_{j^*}\| = \min_{j=1,\dots K} \|\mathbf{x}_i - c_{j^*}\|$$

Die Qualität des clusterings wird über die summierte Differenz

$$e = \sum_{i=1}^{N} \|\mathbf{x}_i - c_{k(\mathbf{x}_i)}\|$$

bewertet.

> Wir sehen, dass zum Bewerten (und damit Definieren) der Cluster eine Norm (oder auch einfach eine Metrik) benutzt und machen uns klar, dass das Ergebnis des clusterings stark von der Wahl der Norm abhängen kann.

Ist die Anzahl der Cluster nicht vorgegeben, ist wiederum ein guter Kompromiss zwischen

- *overfitting* – je höher die Anzahl der cluster ist, desto "näher" können die Daten an den centroids liegen, allerdings wird die Klassifizierung oder Kategorisierung weniger aussagekräftig wenn quasi jeder Datenpunkt sein eigenes cluster bildet.
- *underfitting* – wenige Cluster erlauben zwar sehr konkrete Gruppierung von Daten, allerdings mit dem möglichen Nachteil, dass wesentliche Merkmale nicht berücksichtigt oder falsch klassifiziert werden

In der Praxis behilft Jan sich gerne mit einer *L-Kurve* (oder auch *ellbow plot*), die den Klassifikationsfehler gegenüber der Anzahl der zugelassenen cluster aufträgt: Diese Kurve $e(K)$ ist – in der Theorie – positiv, maximal für $K = 1$ und fällt monoton mit der Anzahl K der cluster. Zeigt diese Kurve einen merklichen Knick bei K^* wird dieser Wert gerne als passender Kompromiss genommen.

6.2 K-means Clustering

Aus der Definition und der Bewertung der Cluster ergibt sich ein einfacher aber sehr oft und erfolgreich genutzter Algorithmus zum clustering, der *k-means* Algorithmus. Er funktioniert wie folgt

```
# Initialisierung: K zufaellige centroids c_j
#
# Wiederholung bis zur Stagnation
#   Berechnung der Zuordnung: k: x_i -> c_j
#   Update der centroids: c_j <- mean{ x_i | x_i aus cluster c_j }
```

Ausgehend von zufällig gewählten centroids, wird die Zuordnung berechnent und dann der Mittelwert aus jedem Cluster als neuer centroid definiert. Dieses Verfahren wird solange wiederholt bis eine maximale Iterationszahl erreicht ist oder die centroids sich nur noch wenig oder gar nicht mehr verändern. In Abbildung 6.1 ist eine beispielhafte Ausführung des Algorithmus auf Beispieldaten dargestellt.

6.3 Clustering und Hauptkomponentenanalyse

Der k-means Algorithmus basiert allein auf der Bestimmung von Abständen und Mittelwerten – beides kann in beliebigen Dimensionen n der Daten realisiert werden. Allerdings ist die Klassifizierung in höheren Datendimensionen ungleich schwieriger da

- für eine vergleichsweise ähnlich große Einzugsbereiche, müssen ungleich mehr clusters verwendet werden. Beispielsweise lößt sich ein Quadrat der Seitenlänge 2 in $4 = 2^2$ Quadrate der Größe 1 aufteilen. Für einen Würfel (eine Dimension mehr), braucht Jan entsprechend $8 = 2^3$ (das heisst doppelt so viele) Unterbereiche. In vier Dimensionen wären es bereits 2^4. Und so weiter.

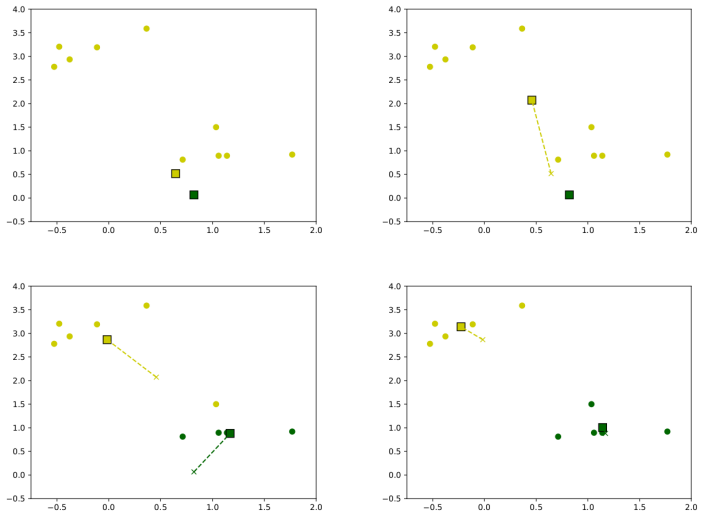


Figure 6.1: Entwicklung der Centroids im Verlauf des k-means clustering Algorithmus auf Beispieldaten

- die Berechnung der Abstände und Zuordnung zu den clusters wird aufwändiger
- es gibt ungleich mehr Konfigurationen für die centroids. Es braucht unter Umständen mehr Iterationen um einen stabilen Zustand zu erreichen.

Ein einfacher Zugang wäre es, eine reduzierte Anzahl der Merkmale zur Klassifizierung einzusetzen. Allerdings wird die Qualität der Klassifizierung davon abhöngen, wie geeignet die ausgesuchten Merkmale dafür sind. Ein Blick auf die verschiedenen Plots in Abbildung 5.1 kann helfen zu verstehen, dass manche Paare von Koordinaten besser als andere geeignet sind um die 3 Pinguinarten zu unterscheiden.

Folgen wir hingegen der Annahme, dass eine hohe Varianz einen hohen Informationsgehalt bedeutet, können wir die Hauptkomponentenanalyse einsetzen um Richtungen von höchster Varianz zu identifizieren und auch Richtungen die überhaupt keine Korrelationen aufzeigen auszuschließen. In der Tat suggerieren die Plots der Pinguindaten in den Hauptkoordinaten (Abbildung 5.3), dass die ersten Hauptrichtungen die Spezies gut unterscheiden, während die letzten Richtungen ein scheinbar zufälliges Rauschen zeigen.

Auch wenn die Hauptrichtungen die Daten nicht unbedingt besser separieren, ermöglichen sie doch das ausschließen der (transformierten) Merkmale ohne Informationsgehalt und damit

einen strukturierten und allgemeinen Ansatz zur Klassifizierung hochdimensionaler Daten mit reduzierten Merkmalen.

6.4 Training und Testing

Das Klassifizieren ist ein erstes konkretes Beispiel dafür, wie aus Daten ein Modell abgeleitet wird. Für die gemessenen Pinguine kennen wir die Spezies, das Ziel ist, ein Modell zu haben mit dem wir zukünftig angetroffene Pinguine anhand ihrer gemessenen Merkmale klassifizieren können.

Allgemein wird für einen solchen neuen Datenpunkt wie folgt vorgegangen

1. (Gegebenenfalls) Normalisierung des Datenpunktes durch Skalierung und Abziehen des (zuvor ermittelten) Mittelwertes.
2. (Gegebenenfalls) Transformation in (reduzierte) Hauptkoordinaten.
3. Test zu welchen centroid c_j, $j = 1, \dots, K$ der (transformierte) Datenpunkt am nächsten liegt.

Der in (3.) ermittelte Index j^* ist dann das Cluster, dem der Datenpunkt zugeordnet wird.

Der Sprachgebrauch hier ist, dass das Modell mit bekannten Daten *trainiert* wird und dann für neue Daten *angewendet* wird[2]. Um zu evaluieren, ob das Modell "funktioniert" ist das folgende Vorgehen ebenso naheliegend wie standardmäßig angewandt.

Die vorhandenen Daten werden aufgeteilt in **Trainingsdaten**, mit denen das Modell trainiert werden, und **Testdaten**, mit denen die Vorhersagekraft des Modells evaluiert wird. Wie die Aufteilung passiert, wird anhand von Erfahrungswerten und im Hinblick auf die vorliegende Aufgabe entschieden.

6.5 Am Beispiel der Pinguine

6.6 Aufgaben

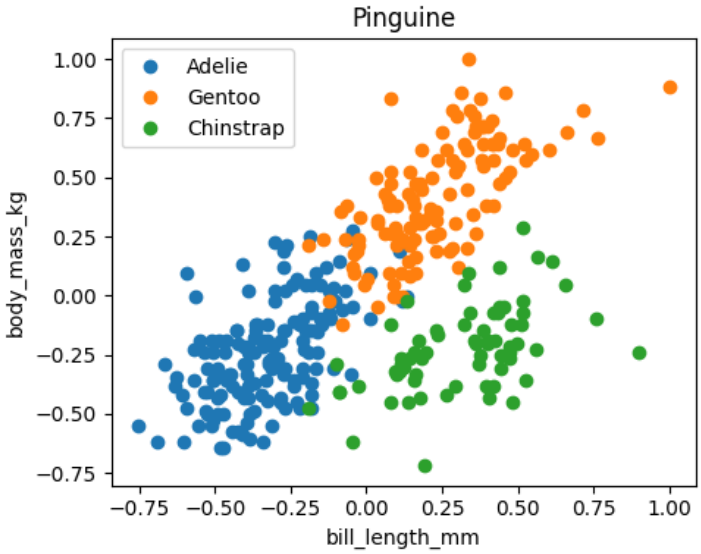
6.6.1 K-means von Hand (T)

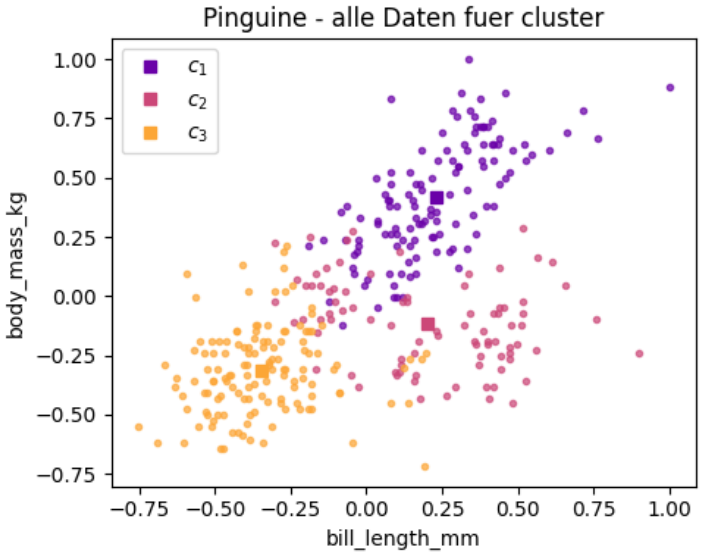
Führen sie händisch und nach Augenschein zwei Iterationen des K-means Algorithmus auf den vom folgenden Skript erzeugten `(x,z)` Daten durch. Dazu bitte, Beispielsweise in Abbildung 6.4, in jedem Schritt die Clusterzugehörigkeiten markieren und die neuen (geschätzten) Mittelpunkte c_1 und c_2 eintragen.

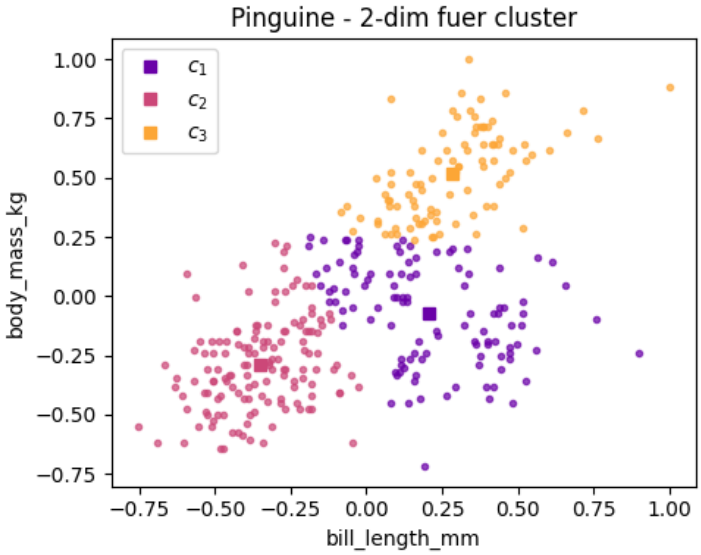
```
from numpy.random import default_rng
import matplotlib.pyplot as plt

# Zufallszahlengenerator mit "seed=1"
```

[2] Im *machine learning* wird gerne von *generalization* gesprochen


Pinguine
Adelie
Gentoo
Chinstrap
body_mass_kg
bill_length_mm
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00


Pinguine - alle Daten fuer cluster
c_1
c_2
c_3
body_mass_kg
bill_length_mm
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00


Pinguine - 2-dim fuer cluster
c_1
c_2
c_3
body_mass_kg
bill_length_mm
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00

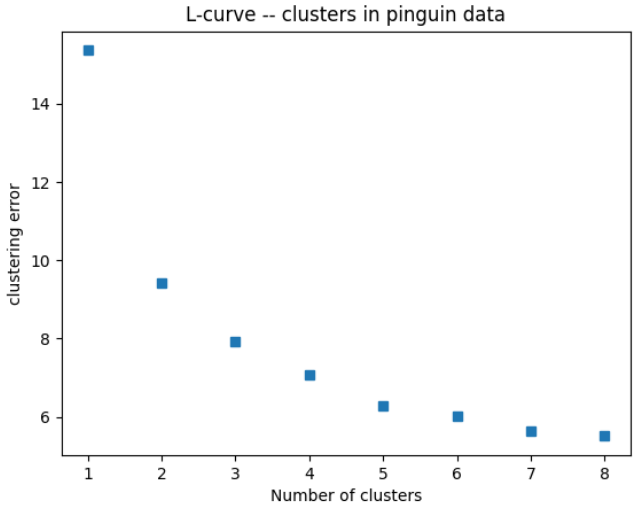


Figure 6.3: Die L-Kurve fuer die Pinguin Daten

```
rng = default_rng(1)

# Zufallsdaten mit etwas Korrelation fuer z
x = rng.beta(1, 1, 9)
y = rng.beta(1, 1, 9)
z = (x*x*y)**(1/3)

# Zufallswahl der initialen centroids
cntro = rng.beta(1, 1, 2)
cntrt = rng.beta(1, 1, 2)

plt.figure(1, figsize=(5, 3))
plt.plot(x, z, '.', label='data')
plt.plot(cntro[0], cntro[1], 's', label='$c_1$')
plt.plot(cntrt[0], cntrt[1], 's', label='$c_2$')
plt.legend(loc='upper left')
plt.title('K-means: initialer Zustand')
plt.show()
```

6.6.2 K-means als Algorithmus (P)

Schreiben sie eine Funktion, die mittels des k-means Algorithmus einen Datensatz in beliebiger Dimension in `K` vorgegebene cluster gruppiert und die labels der Daten sowie die Koordinaten der centroids zurueckgeben.

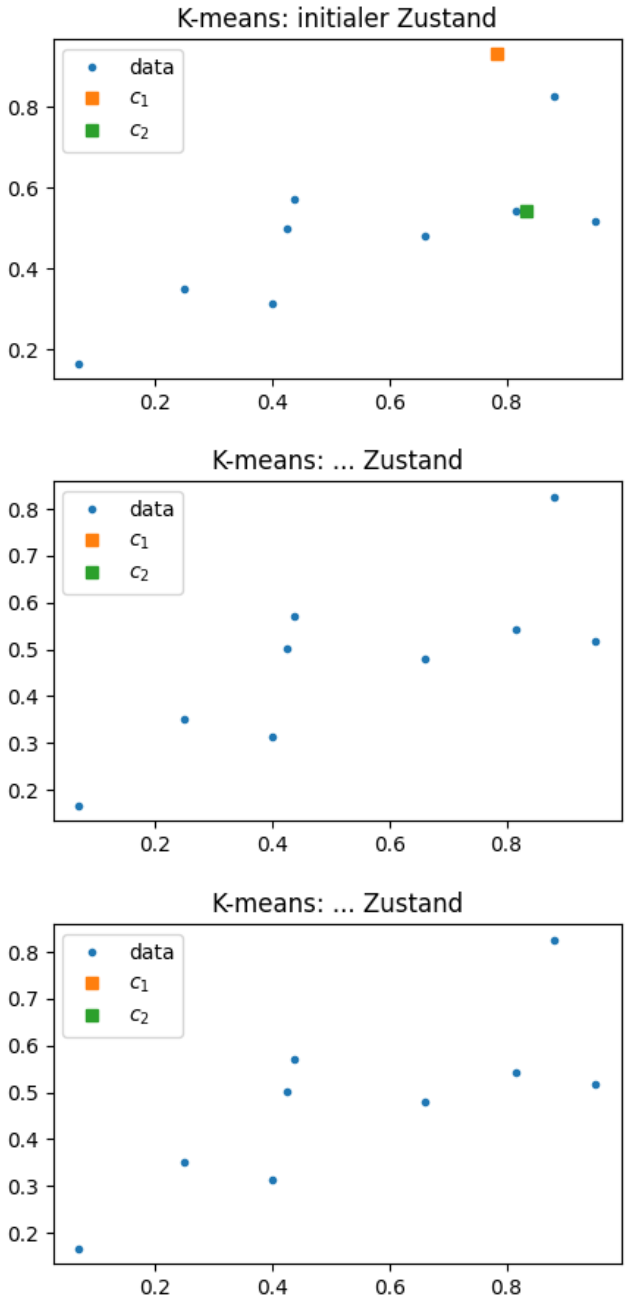


Figure 6.4: Bitte gerne hier die Evolution einzeichnen

Testen sie die Funktion am obigen theoretischen Beispiel.

Hier zum Download angehängt eine Datei mit ein paar hilfreichen Routinen und schon einer Vorgabe für die "Hauptroutine". **Bitte machen Sie sich damit vertraut, wie in Python Routinen aus einem *module* (d.h. einer anderen Datei mit Python Funktionen) importiert werden kann.**

```
from kmeans_utils import kmeans, plot_cluster_data
# import routines from a module

# construct data as above
N = 9
K = 2

x = rng.beta(1, 1, N)
y = rng.beta(1, 1, N)
z = (x*x*y)**(1/3)

cntro = rng.beta(1, 1, 2)
cntrt = rng.beta(1, 1, 2)

cntrdlist = [cntro, cntrt]
datamatrix = np.hstack([x.reshape((x.size, 1)), z.reshape((z.size, 1))])

# run the kmeans algorithm
cntrdlist, labels, cerror = kmeans(datamatrix, K=2, cntrdlist=cntrdlist)

# plot the result
plot_cluster_data(datamatrix, labels, cntrdlist,
                  xlabel='feature', ylabel='other feature',
                  fignum=10101, figsize=(5, 4),
                  titlestr='K-means: finaler Zustand')
```

6.6.3 K-means fuer die Pinguine

Wenden sie den Algorithmus auf die (zentrierten und auf in jeder Richtung auf $[-1, 1]$ skalierten) Pinguindaten an. Und vergleichen sie mit den Bildern oben.

1. In der gesamten Datendimension
2. Für zwei beliebig gewählte Merkmale
3. Für die ersten drei Hauptkomponenten

```
import json

import numpy as np
import matplotlib.pyplot as plt

from kmeans_utils import kmeans
```

```
with open('penguin-data.json', 'r') as f:
    datadict = json.load(f)

data = np.array(datadict['data'])

# center and scale
data = data - data.mean(axis=0)
maxvals = np.max(np.abs(data), axis=0)
data = data * 1/maxvals
# now the data has zero mean and is scaled to fit into [-1, 1]
```

> Achtung – auch hier ist wieder Zufall involviert. Wenn das Ergebnis seltsam aussieht, gegebenfalls das Experiment mehrfach wiederholen um auszuschließen, dass es sich nur um einen statistischen Ausreißer handelt.

6.6.4 L-curve

Bestimmen sie die L-Kurve fuer die Pinguindaten fuer `K=1,2,3,4,...,8` – fuer das clustering auf dem gesamten Datensatz.

Chapter 7

Optimierung

Einige Optimierungsprobleme haben wir schon kennengelernt und zwar

- bei der linearen Regression (vgl. besonders Kapitel 2.3) bei der die Parameter der Ansatzfunktion **bestmöglich** an die Daten gefittet wurden
- bei der Hauptkomponentenanalyse (vgl. besonders Kapitel 4.3) als wir die Hauptrichtungen so ermittelt, dass in ihrer Richtung die Varianz **maximal** wurde.

Hier lag in beiden Fällen ein sogenanntes linear-quadratisches Optimierungsproblem vor, für welche die Existenz der Lösungen bewiesen werden kann und die mit Methoden aus der linearen Algebra direkt gelöst werden können.

Sogenannte *nichtlineare Optimierungsprobleme* liegen vor, wenn die zu erfüllenden Ziele wirklich nichtlineare Gleichungen enthalten. Zum Bespiel sind gängige Elemente von *Neuronalen Netzen* als

$$f(x) = \tanh(a\,x + b)$$

gegeben, mit Parametern a, b die aus der Minimierung eines *mean square error* Terms der Form

$$\frac{1}{N}\sum_{k=1}^{N} |y_k - \tanh(a\,x_k + b)|^2$$

für gegebene Trainingsdaten (x_k, y_k) bestimmt werden.

Wir sehen, dass die Optimierung ein ganz wesentlicher Teil der *Data Science* ist (und ganz unabhängig davon ein eigenes Forschungsgebiet ist und quasi überall in der Praxis mittel– oder unmittelbar benutzt wird).

Nachfolgend ein paar Begriffe aus der Optimierung

- *glatte Optimierung* – das Modell beinhaltet ausreichend stetige (bzw. differenzierbare) Funktionen. Ein Optimum könnte durch Bestimmung von Ableitungen berechnet werden.
- *nichtglatte Optimierung* – das Modell beinhaltet Teile, die nicht differenzierbar sind. Das kann am Problem selbst liegen oder, was bei *machine learning* im Speziellen und *Data Science* an sich eine Rolle spielt, dass die Daten verrauscht sind. Jetzt ist eine (klassische) Ableitung nicht mehr möglich aber es existieren diverse Ansätze zur Abhilfe.
- *diskrete Optimierung* – in obigen Fällen sind wir implizit davon ausgegangen, dass die Lösung und die Gleichungen auf den reellen oder komplexen Zahlen leben. Denkbar ist aber auch, dass diskrete Lösungen (z.B. Anzahlen oder binäre Zustände) gesucht werden.
- *kombinatorische Optimierung* – hier ergeben sich die möglichen Lösungen als Kombination von Möglichkeiten. Oft ist die Anzahl möglicher Lösungen endlich aber sehr groß.
- *restringierte Optimierung* – zusätzlich zu einem Optimalitätskriterium sind noch Nebenbedingungen zu erfüllen.

Hier werden wir uns erstmal mit glatter Optimierung ohne Nebenbedingungen beschäftigen.

7.1 Multivariable Funktionen

Um unsere Optimierungsprobleme in allgemeiner Form zu formulieren und auch um allgemeine Lösungsansätze herzuleiten, brauchen wir erstmal ein paar Begriffe aus der Analysis von multivariablen Funktionen.

Es sei also eine Funktion

$$f\colon \mathbb{R}^n \to \mathbb{R}, \quad f\colon (x_1, x_2, \dots, x_n) \mapsto f(x_1, x_2, \dots, x_n)$$

gegeben, die ein n-tupel von Variablen auf einen reellen Wert abbildet. Ein Beispiel wäre

$$g(x_1, x_2, x_3) = \sin(x_1) + x_3 \cos(x_2) + x_1^2 x_2^2 x_3^2 - 2x_2.$$

Hier verwenden wir jetzt x_i für die i-te Variable der multivariablen Funktion. Bitte nicht verwechseln mit der Bezeichnung für den i-ten Datenpunkt zum Beispiel in Kapitel 2.

Außerdem wäre es besser wir würden die Funktion mit einem Definitionsbereich $D(f)$ angeben, also schreiben

$$f\colon D(f) \subset \mathbb{R}^n \to \mathbb{R}$$

da eine Funktion eventuell nicht überall definiert ist (bzw. definiert sein muss). Im Sinne der besseren Lesbarkeit, ersparen wir uns dieses Detail.

7.2 Partielle Ableitungen und der Gradient

Wenn wir alle Variablen bis auf die i-te für einen Augenblick einfrieren (also auf einen konstanten Wert $\bar{x}_j$ festlegen und nicht verändern), können wir die Teilfunktion

$$\bar{f}_{(i)} \colon \mathbb{R} \to \mathbb{R}, \quad \bar{f}_{(i)} \colon x_i \to f(\bar{x}_1, \dots, \bar{x}_{i-1}, x_i, \bar{x}_{i+1}, \dots, \bar{x}_n)$$

als Funktion mit einer Veränderlichen (und $n-1$ Parametern) betrachten und wie gehabt ableiten. Die erhaltene, sogenannte *partielle Ableitung* hängt von den Parametern $\bar{x}_j$ ab und wird mit

$$\frac{\partial f}{\partial x_i}$$

bezeichnet. Alle n möglichen partiellen Ableitungen in einen Vektor geschrieben ergeben den sogenannten *Gradienten*

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \colon \mathbb{R}^n \to \mathbb{R}^n,$$

der wiederum eine (vektorwertige) Funktion von n Variablen ist. Für unser obiges Beispiel bekommen wir

$$\nabla g(x_1, x_2, x_3) = \begin{bmatrix} \cos(x_1) + 2x_1 x_2^2 x_3^2 \\ -x_3 \sin(x_2) + x_1^2 x_2 x_3^2 - 2 \\ \cos(x_2) + 2x_1^2 x_2^2 x_3 \end{bmatrix}$$

7.3 Richtungs-Ableitung

Die Betrachtung der Funktionen $\bar{f}_{(i)}$ ermöglicht die Analysis des Verhaltens der Funktion entlang der i-ten Koordinaten Richtung. Um die Funktion in einer beliebigen Richtung um einen festen Punkt

$$\bar{x} = (\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n)$$

zu untersuchen, kann die Funktion entlang einer Richtung $v \in \mathbb{R}^n$ parametrisiert werden:

$$\bar{f}_{(v)} \colon \mathbb{R} \to \mathbb{R}, \quad t \mapsto \bar{f}_{(v)}(t) = f(\bar{x} + tv)$$

Für $v = e_i$ bekommen wir direkt, dass $\bar{f}_{(v)} = \bar{f}_{(i)}$.

Wiederum handelt es sich um eine reelle Funktion in einer Variablen (hier der Parameter t), die wir schulbuchmäßig ableiten könnten. Für diese sogenannte *Richtungsableitung* gilt der folgende Zusammenhang[1]

$$\frac{\mathrm{d}\bar{f}_{(v)}}{\mathrm{d}t}(0) = \sum_{i}^{n} \frac{\partial f}{\partial x_i}(\bar{x}_i)v_i = \langle \nabla f(\bar{x}), v \rangle$$

> Die Veränderungsrate der Funktion f im Punkt $\bar{x}$ in der Richtung von v ist gleich dem Skalarprodukt aus Gradient im Punkt $\bar{x}$ und der gewählten Richtung.

Sei nun v normiert, also $\|v\| = 1$, dann gilt nach der Winkelformel, dass

$$\langle \nabla f(\bar{x}), v \rangle = \|\nabla f(\bar{x})\| \cos(\alpha)$$

wobei α der von $\nabla f(\bar{x})$ und v eingeschlossene Winkel ist. Wir bemerken, dass die Veränderung maximal wird, wenn $\alpha = 0$ oder

$$v = \frac{1}{\|\nabla f(\bar{x})\|} \nabla f(\bar{x})$$

als ein Vielfaches des Gradientenvektors gewählt wird. Andersherum können wir einige ganz wesentliche Aspekte folgern:

- der Gradient selbst in die Richtung des stärksten Anstiegs
- der negative Gradient $-\nabla f(\bar{x})$ zeigt in die Richtung des stärksten Abfalls
- ist der Gradient der Nullvektor, dann findet keine Veränderung mehr statt, d.h. dass aus $\nabla f(\bar{x}) = 0$ folgt, dass $\bar{x}$ ein kritischer Punkt von f ist
- ist der Gradient nicht der Nullvektor, kann $\bar{x}$ kein Minimum und kein Maximum sein (weil die Funktion in $-\nabla f(\bar{x})$ Richtung abfällt und in $\nabla f(\bar{x})$ Richtung ansteigt)

7.4 Optimierung

Ganz allgemein können wir unsere bisherigen Optimierungsprobleme schreiben als

$$f(x) \to \min_{x \in \mathbb{R}^n},$$

das heißt für eine Funktion $f \colon \mathbb{R}^n \to \mathbb{R}$ suchen wir das $x \in D(f) \subset \mathbb{R}^n$, für das $f(x)$ minimal wird.

Definition 7.1. Gegeben sei $f \colon \mathbb{R}^n \to \mathbb{R}$. Wir nennen $x^* \in D(f) \subset \mathbb{R}^n$ die Stelle

[1] Das folgt aus der Kettenregel für multivariable Funktionen, die wir in der Vorlesung *Mathematik für Data Science 2* noch beweisen werden

- eines *lokales Minimum* (von f) falls es ein $\epsilon > 0$ gibt, sodass

$$f(x^*) \leq f(x)$$

für alle $x \in D(f) \cap B_\epsilon(x^*)$.

- eines globales Minimum, falls $f(x^*) \leq f(x)$ für alle $x \in D(f)$ ist.

In dieser Definition benutzen wir "$\leq$", da das im Hinblick auf Minimierung (der Wert von f ist wichtig, die Stelle, an der das Minimum angenommen wird, ist weniger entscheidend) praktikabel ist. Wir könnten aber auch von strikten (oder "echten") Minima sprechen und "$<$" verlangen.

Der Schnitt der Umgebung

$$B_\epsilon(x^*) := \{x \in \mathbb{R}^n | \|x - x^*\| < \epsilon\}$$

mit dem Definitionsbereich $D(f)$ ist relevant, wenn x^* genau auf dem Rand des Definitionsbereichs liegt.

Nun erstmal ein hinreichendes Kriterium für ein Extremum (vgl. Thm. 2.2, Nocedal and Wright 2006):

Theorem 7.1. *Sei $D(f) \subseteq \mathbb{R}^n$ offen, und sei $f\colon D(f) \to \mathbb{R}$ stetig partiell differenzierbar. Besitzt f in $\mathbf{x}^* \in D(f)$ ein lokales Extremum[2], dann ist $\nabla f(\mathbf{x}^*) = 0$.*

> Das können wir wie in der einfachen Kurvendiskussion verstehen: *Liegt in x^* ein Extremum vor und x^* liegt nicht auf dem Rand, so verschwindet die erste Ableitung.* Um festzustellen, ob ein Minimum, Maximum und vor allem nicht einfach ein Sattelpunkt vorliegt, wurde geschaut ob die zweite Ableitung positiv, negativ oder gleich 0 ist. Bei multivariablen Funktionen wird die zweite Ableitung durch eine Matrix repräsentiert. Und positiv, negativ oder indefinit wird wird über die Eigenwerte definiert.

Definition 7.2 (Hesse-Matrix)**.** Sei $D(f) \subseteq \mathbb{R}^n$ offen, und sei $f\colon D(f) \to \mathbb{R}$ eine 2-mal stetig partiell differenzierbare Funktion. Für $\mathbf{x}^\star \in D$ heißt die symmetrische $(n \times n)$-Matrix

$$H_f(\mathbf{x}^\star) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(\mathbf{x}^\star) & \frac{\partial^2 f}{\partial x_2 \partial x_1}(\mathbf{x}^\star) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_1}(\mathbf{x}^\star) \\ \frac{\partial^2 f}{\partial x_1 \partial x_2}(\mathbf{x}^\star) & \frac{\partial^2 f}{\partial x_2 \partial x_2}(\mathbf{x}^\star) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_2}(\mathbf{x}^\star) \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n}(\mathbf{x}^\star) & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\mathbf{x}^\star) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}(\mathbf{x}^\star) \end{pmatrix}$$

Hesse-Matrix von f in $\mathbf{x}^\star$.

[2] könnte auch ein Maximum sein, was wir aber einfach als ein Minimum von $-f$ einordnen

Hierbei ist $\frac{\partial^2 f}{\partial x_i \partial x_j} := \frac{\partial}{\partial x_i}(\frac{\partial f}{\partial x_j})$ die wiederholte partielle Differentiation in möglicherweise verschiedene Koordinatenrichtungen. Die postulierte Symmetrie der Hesse-Matrix folgt aus der Gleichheit

$$\frac{\partial}{\partial x_i}(\frac{\partial f}{\partial x_j}) = \frac{\partial}{\partial x_j}(\frac{\partial f}{\partial x_i})$$

deren Gültigkeit wir noch in der Mathematik Vorlesung beweisen werden. Als nächstes ein Hinreichendes Kriterium für Minimalität (vgl. Theorem 2.4, Nocedal and Wright 2006):

Theorem 7.2 (Hinreichende Optimalitätsbedingung zweiter Ordnung)**.** *Sei $D(f) \subset \mathbb{R}^n$ offen, und sei $f\colon D(f) \to \mathbb{R}$ eine 2-mal stetig partiell differenzierbare Funktion. Sei $x^\star \in D(f)$ mit $\nabla f(x^\star) = 0$ Ist $H_f(x^\star)$ positiv definit, dann ist $\mathbf{x}^\star$ ein lokales Minimum.*

Ein paar Bemerkungen dazu

- Eine symmetrische Matrix M heißt *positiv definit*, wenn $x^T M x > 0$ ist für alle $x \in \mathbb{R}^n \quad \{0\}$. Wir schreiben $M \prec 0$.
- Hinreichende Kriterien dafür sind
  - alle Eigenwerte sind positiv oder
  - alle Hauptminoren sind positiv (das sind die Determinante der Matrizen, die aus M durch Streichung der letzten r Zeilen und Spalten entstehen $r = 0, 1, \dots, n-1$.)
- Ist $-M \prec 0$, dann liegt ein lokales Minimum von $-f$ vor (oder eben ein Maximum von f)
- Gilt $M \preccurlyeq 0$ (also M ist positiv semidefinit, das heißt die Eigenwerte sind nur "≥ 0" bzw. es gilt nur $x^T M x \geq 0$), dann ist **keine** Aussage möglich.
- Ist die Matrix bestimmt indefinit, also alle Eigenwerte "$\neq 0$" aber manche positiv und manche negativ, dann liegt ein Sattelpunkt vor (und sicher kein Extremum).

7.5 Gradientenabstiegsverfahren

Aus dem Wissen, dass in einem Minimum x^*, der Gradient $\nabla f(x^*)$ verschwindet und dass in einem beliebigen Punkt x^0 (der kein Extremum ist), der negative Gradient $-\nabla f(x^0)$ in die Richtung des stärksten Abstiegs zeigt, ergibt sich der folgende Zusammenhang:

> Für einen beliebigen Startpunkt x^0 gibt es eine Schrittweite α^0, so dass
>
> $$f(x^1) := f(x^0 - \alpha^0 \nabla f(x^0)) < f(x_0)$$

Das heißt in einem Startpunkt können wir einen kleinen Schritt in die Richtung des stärksten Abstiegs gehen und so die Funktion etwas minimieren. Das wiederholte Anwenden dieses Prinzips ergibt das Gradientenabstiegsverfahren,

$$f(x^{n+1}) := f(x^n - \alpha^n \nabla f(x^n)) < f(x_n)$$

, das die Funktion immer weiter minimiert bis entweder der Definitionsbereich verlassen ist oder ein kritischer Punkt mit $\nabla f(x^k) = 0$ erreicht wurde. An diesem könnte nun die Hessematrix ausgewertet werden um festzustellen ob tatsächlich ein Minimum vorliegt.

- Neben der Berechnung des Gradienten, die für große Dimensionen sehr aufwändig werden kann, ist die Bestimmung der Schrittweite α^k in jedem Schritt entscheidend.
  - Dazu kann beispielsweise in jeder Iteration die skalare Funktion $\alpha \mapsto f(x^k - \alpha \nabla f(x^k))$ minimiert werden (*exact line search*)
  - Da das aber auch aufwändig sein kann, wurden zahlreiche Approximationen an diese Zwischenoptimierung entwickelt (*inexact line search*).
  - Vgl. auch den Wikipedia Artikel zum Gradientenverfahren#Bestimmen_der_Schrittweite
- Für passend gewählte Schrittweiten und wenn f hinreichend glatt ist (der Gradient soll Lipshitz-stetig sein), lässt sich Konvergenz und sogar Konvergenzrate der Iteration zu einem stationären Punkt x^* mit $\nabla f(x^*) = 0$ bestimmen (vgl. Section 3.2 in Nocedal and Wright 2006).
- Um tatsächlich die Konvergenz zu einem Minimum zu beweisen, wird beispielsweise vorausgesetzt, dass ein striktes lokales Minimum existiert (*lokale Konvexität*) und dass die Hesse-Matrix stetig ist (also f zweimal stetig partiell differenzierbar ist) (vgl. Section 3.2 in Nocedal and Wright 2006).
- Wir können uns leicht überlegen, dass dieser Algorithmus immer zu einem lokalen Minimum konvergieren wird und das entscheidend von der Wahl des Startpunktes abhängt.

7.6 Extra: Nichtglatte Optimierung

Ist die Zielfunktion nicht differenzierbar (oder ist die Berechnung des Gradienten zu aufwändig) kann auf Optimierungsverfahren zurückgegriffen werden, die keinen Gradienten benötigen. Die bekanntesten sind

- *Nelder-Mead* oder auch (*nonlinear Simplex*) Verfahren: Berechnet für $n+1$ Stützpunkte (ein Simplex im $\mathbb{R}^n$ die Werte von f und ersetzt den *schlechtesten* nach Kriterien die eine (vergleichsweise) schnelle Konvergenz zu einem lokalen Minimum garantieren können.
- Sogenannte *generische* oder *genetische Algorithmen*: nach dem Prinzip der Variation (oder Mutation) und Selektion wird der Suchraum $D(f)$ in

zufälliger Weise aber mit einer gewissen Systematik abgesucht – diese Algorithmen konvergieren (in Wahrscheinlichkeit) zu einem globalen Minimum sind aber vergleichsweise langsam bzw. rechenaufwändig.

7.7 Extra: Automatisches Differenzieren

Wir sehen, dass in der glatten Optimierung die Berechnung des Gradienten die wichtigste und gleichzeitig aufwändigste Komponente ist. Kann der Gradient analytisch berechnet und als Funktion zur Verfügung gestellt werden, ist dieses Problem elegant umgangen. In der Praxis sind die Probleme aber oft als eine mehrschichitige Verknüpfung von Funktionen in einem Programmcode gegeben. Andererseits sind die kleinste Elemente dieser Funktion typischerweise Elementarfunktionen, die einfach und analytisch differenziert werden können.

Die Darstellung einer Funktion, zum Beispiel,

$$f(x_1, x_2) = \|(x_1, x_2)\| = \sqrt{x_1^2 + x_2^2}$$

die als Koordinatenfunktion für x_1 dargestellt werden kann als

$$\tilde{f}_1(x_1, z) = g_3(g_2(g_1(x_1); z)), \quad g_3(y) = \sqrt{y}, \quad g_2(y; z) = y + z^2, \quad g_1(y) = y^2$$

und mittels Kettenregel die partielle Ableitung als Kombination elementarer Ableitungen berechnet als

$$\frac{\partial f}{\partial x_1}(x_1, x_2) = g_3'(g_2(g_1(x_1); x_2)) \cdot g_2'(g_1(x_1); x_2) \cdot g_1'(x_1)$$

macht sich *Automatisches Differenzieren* zu Nutze und kann zu einem Programmcode

```
def ffun(x):
    # ..... alles was f machen soll
    return fx
```

einen Programmcode erzeugen, der `grad_ffun(x)` berechnet. Das ist der sogenannten Vorwärtsmodus des *Automatischen Differenzierens*.

Der sogenannte *Rückwärtsmodus* ist effizienter und kann leicht ein Programme eingebaut werden und damit zu jeder Funktionsevalution, direkt den Gradienten an dieser Stelle mitliefern.

> Die systematische Verwendung des *Automatischen Differenzierens* ist einer der wesentlichsten Garanten für den Erfolg von *machine learning*, da AD das träinieren von grossen Netzwerken überhaupt erst möglich macht.

7.8 Aufgaben

7.8.1 Numerische Berechnung des Gradienten (P+T)

Berechnen sie näherungsweise den Gradienten der Beispielfunktion

$$f(x_1, x_2, x_3) = \sin(x_1) + x_3 \cos(x_2) + -2x_2 + x_1^2 + x_2^2 + x_3^2$$

(**Achtung** dieses f ist nur fast das g wie oben) im Punkt $(x_1, x_2, x_3) = (1, 1, 1)$, indem sie die partiellen Ableitungen durch den Differenzenquotienten, z.B.,

$$\frac{\partial g}{\partial x_2}(1, 1, 1) \approx \frac{g(1, 1 + h, 1) - g(1, 1, 1)}{h}$$

für $h \in \{10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}\}$ berechnen. Berechnen sie auch die Norm der Differenz zum exakten Wert von $\nabla g(1, 1, 1)$ (s.o.) und interpretieren sie die Ergebnisse.

```
import numpy as np

def gfun(x):
    return np.sin(x[0]) + x[2]*np.cos(x[1]) \
        - 2*x[1] + x[0]**2 + x[1]**2 \
        + x[2]**2

def gradg(x):
    return np.array([np.NaN,
                     -x[2]*np.sin(x[1]) - 2 + 2*x[1],
                     np.NaN]).reshape((3, 1))


# Inkrement
h = 1e-3

# der x-wert und das h-Inkrement in der zweiten Komponente
xzero = np.ones((3, 1))
xzeroh = xzero + np.array([0, h, 0]).reshape((3, 1))

# partieller Differenzenquotient
dgdxtwo = 1/h*(gfun(xzeroh) - gfun(xzero))
# Alle partiellen Ableitungen ergeben den Gradienten
hgrad = np.array([np.NaN, dgdxtwo, np.NaN]).reshape((3, 1))

# Analytisch bestimmter Gradient
gradx = gradg(xzero)

# Die Differenz in der Norm
hdiff = np.linalg.norm((hgrad-gradx)[1])
```

```
# bitte alle Kompenenten berechnen
# und dann die Norm ueber den ganzen Vektor nehmen

print(f'h={h:.2e}: diff in gradient {hdiff.flatten()[0]:.2e}')
```

7.8.2 Optimierung ohne Gradienten (P+T)

Werten sie g an 1000 zufällig gewählten Punkten aus und lokalisieren sie das $\bar{x} \in \mathbb{R}^3$ an welchem g in diesem Sample den kleinsten Wert annimmt.

Überlegen Sie sich eine mögliche Verbesserung dieses rein auf Zufall basierenden Verfahrens. Wer Inspiration braucht, kann mal nach *Generischen Algorithmen* suchen.

7.8.3 Gradientenabstiegsverfahren (P)

Implementieren Sie ein Gradientenabstiegsverfahren mit *line search* durch Bisektion und berechnen Sie ein Minimum von g damit.

```
import numpy as np

from loesung_5_1 import gfun, gradg


def min_by_btls(func, v=None, curx=None,
                azero=10., tau=.8, c=.9, maxiter=200):
    ''' find a smaller value of f in the direction

    f(x+s*v)

    for s in [0, azero]
    by backtracking linesearch

    https://en.wikipedia.org/wiki/Backtracking_line_search
    '''

    m = -v.T.dot(v)
    cstep = azero
    t = -c*m

    cval = func(curx)

    for kkk in range(maxiter):
        newx = curx+cstep*v
        if cval - func(newx) >= cstep*t:
            return newx
```

```
        else:
            cstep = tau*cstep
    raise UserWarning('no smaller value could be found')
    return


def gradienten_abstieg_linesearch(func=None, grad=None, inix=None,
                                  maxiter=100, grad_tol=1e-6):
    '''sucht ein Minimum einer Funktion `func` durch
     * Evaluieren des Gradienten `grad`
     * und anpassen von `inix` in Richtung von `-grad`
     * mit einem `line search Verfahren`
    '''

    # initialisierung
    curx = inix
    for kkk in range(maxiter):
        # compute the current gradient
        # finde ein neues `newx` in -Gradienten Richtung
        # check ob der Gradient unter der Toleranz ist
        # anderenfalls:
        curx = newx  # naechster Iterationschritt

    print(f'max iter {maxiter} reached -- size of grad {nrmcgrd:.2e}')
    return newx

# Startwert
xzero = np.ones((3, 1))

minx = gradienten_abstieg_linesearch(func=gfun, grad=gradg, inix=xzero)

print(f'found minimum f(x) = {gfun(minx)}')
print('at x = ', minx)
```

7.8.4 Built-in Optimierung in Scipy (P)

1. Benutzen sie die `scipy.optimize.minimize` Routine um ein Minimum von g zu finden und messen Sie die Zeit die der Algorithmus braucht.

2. Geben Sie dem Algorithmus eine Funktion mit, die den Gradienten berechnet. Berechnen sie ein Minimum und vergleichen Sie die benötigte Zeit. **Hinweis**: die Routine braucht den Gradienten als *flachen* Vektor. Das kann (wie unten im Beispiel) durch eine *wrapper* Funktion erreicht werden.

```
import numpy as np
from timeit import default_timer
```

```
from scipy.optimize import minimize

from loesung_5_1 import gfun, gradg


def gradforopt(x):
    return gradg(x).flatten()


starttime = default_timer()
result = minimize(...., jac=gradforopt)
runtime = default_timer() - starttime

print(f'with gradient: found minimum f(x) = {result.fun}')
print('at x = ', result.x)
print(f'in {runtime:.4f}s time')
```

Chapter 8

Optimierung unter Nebenbedingungen

Oftmals werden an die gesuchte Lösung $x^* \in \mathbb{R}^n$ eines Optimierungsproblems a-priori bestimmte Anforderungen gestellt, wie:

- $x_1 + x_2 \geq 0$ (die Lösung soll in einem bestimmten Abschnitt liegen)
- $x_1^2 + x_2^2 - c^2 = 0$ (die ersten beiden Komponenten der Lösung sollen auf einer Kreisbahn liegen)

Allgemein lässt sich so ein Optimierungsproblem schreiben als

Definition 8.1 (Optimierungsproblem mit Nebenbedingungen)**.**

$$f(x) \to \min_{x \in \mathbb{R}^n}, \quad \text{s.t.} \quad g_i(x) \geq 0, \quad h_j(x) = 0,$$

- mit der *Zielfunktion* $f \colon \mathbb{R}^n \to \mathbb{R}$
- und zusätzlichen Nebenbedingungen (*s.t.* – *subject to*), die in
  - *Ungleichungsform* über Funktionen $g_i \colon \mathbb{R}^n \to \mathbb{R}$, mit i aus der Indexmenge $\mathcal{I} \subset \mathbb{N}$ oder
  - *Gleichungsform* über Funktionen $h_j \colon \mathbb{R}^n \to \mathbb{R}$, mit j aus der Indexmenge $\mathcal{J} \subset \mathbb{N}$ vorliegen können.

heißt *restringiertes Optimierungsproblem* oder *Optimierungsproblem mit Nebenbedingungen*

Liegen keine Gleichheits– oder keine Ungleichungsnebenbedingungen vor, setzen wir $\mathcal{I} = \emptyset$ oder $\mathcal{J} = \emptyset$.

Ein Optimierungsproblem ohne Nebenbedingungen heißt auch *unrestringiertes* Problem.

Jetzt stellt sich zur Frage ob ein x^* *optimal* ist, noch die Frage, ob es auch *zulässig* (engl.: *feasible*) ist. Dazu lässt sich formal der Zulässigkeitsbereich (engl.: *feasible set*) definieren

$$\Omega = \{x \in \mathbb{R}^n \mid g_i(x) \geq 0, \quad h_j(x) = 0, \quad i \in \mathcal{I}, j \in \mathcal{J}\}$$

und das Optimierungsproblem als

$$f(x) \to \min_{x \in \Omega}$$

schreiben.

> Die Definition 7.1 einer Minimalstelle gilt unmittelbar weiter, wenn nun auch der Zulässigkeitsbereich noch beachtet wird.

Die Unterscheidung von $D(f)$ und Ω in der Betrachtung des Optimierungsproblems und potentieller Minimalstellen hat diverse vor allem praktische Gründe:

- der Definitionsbereich von f ist eine vorgegebene Eigenschaft der Funktion während Ω vielleicht variiert werden soll
- liegen nichttriviale Gleichheitsbedingungen vor, ist beispielsweise kein Punkt von Ω mehr ein innerer Punkt (im $\mathbb{R}^n$) und der Gradient von f wäre erstmal nicht definiert
- die Modellierung ist oftmals einfacher, wenn Nebenbedingungen einfach zum Problem hinzugefügt werden können
- ebenso die Feststellung ob ein gegebener Punkt im Zulässigkeitsbereich liegt

Die Analysis restringierter Optimierungsprobleme und entsprechend die Umsetzung von Algorithmen zur Lösung wird durch die Nebenbedingungen ungleich schwieriger, da

- in der Theorie nicht mehr einfach differenziert werden kann (insbesondere bei Gleichheitsbedingungen oder wenn der kritische Punkt am Rand der Ungleichungsbedingungen liegt) und
- in der Umsetzung auf dem Computer darauf geachtet werden muss, dass der Zulässigkeitsbereich nicht verlassen wird.

8.1 Richtungen und Nebenbedingungen

Bei *restringierten Optimierungsproblemen* ist die Richtung in der analysiert wird entscheidend. Hierbei geht es darum ob in einer Richtung v

1. die Zielfunktion eventuell kleiner wird
2. der Zulässigkeitsbereich eventuell verlassen wird (bspw. charakterisiert durch $g(x) = 0$ oder $h(x) \geq 0$)

Da es stets nur um die unmittelbare Umgebung eines Punktes geht, und wir aus der Diskussion der Ableitung (in 1D) wissen, dass eine differenzierbare Funktion lokal gut durch ihre Tangente angenähert werden kann[1], können wir beide Betrachtungen in der *lineaeren Approximation* anstellen.

Wir schauen also ob für eine Richtung v gilt, ob nicht

1. $f(x^*) > f(x^* + hv) \approx f(x^*) + \nabla f(x^*)^T v$ bzw.
2. $0 \neq g(x^* + hv) \approx \nabla g(x^*)^T v$ oder $0 > h(x^* + hv) \approx \nabla h(x^*)^T v$.

8.2 Restringierte Optimierungsprobleme und der Gradient

Zur Illustration und zur Einführung der Konzepte betrachten wir Probleme, die entweder nur Gleichungsnebenbedingungen oder nur Ungleichungsnebenbedingungen haben.

Wir beginnen mit Gleichungsnebenbedingungen und das auch nur im zweidimensionalen Fall:

$$f(x, y) \to \min, \quad \text{s.t. } g(x, y) = c.$$

Aus der schematischen Darstellung in Abbildung 8.1 können wir ablesen, dass in einem potentiellen Extremum, die Gradienten von f und g in die gleiche Richtung zeigen, es also eine Skalar λ geben muss, sodass in einem kritischen Punkt (x^*, y^*) gilt, dass

$$\nabla f(x^*, y^*) + \lambda \nabla g(x^*, y^*) = 0$$

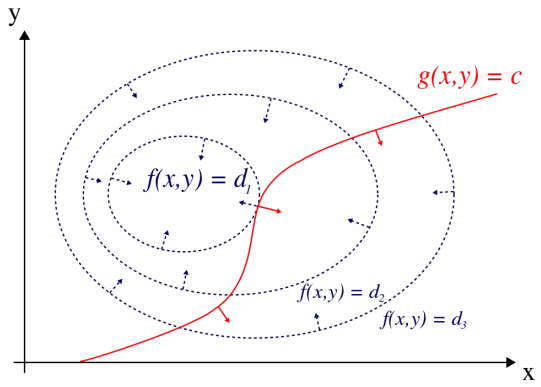


Figure 8.1: Schematische Darstellung eines Optimierungsproblems mit Gleichungsnebenbedingungen in 2D.

[1] *gut* im Sinne von der Fehler $f(x + h) - f(x) - f'(x)h$ geht schneller zur 0 als h

Wir könnten also versuchen, eine Lösung (x, y, λ) für das Gleichungssystem

$$\begin{aligned} \nabla f(x,y) + \lambda \nabla g(x,y) &= 0 \\ g(x,y) &= c \end{aligned} \tag{8.1}$$

zu finden.

> Dieses Gleichungssystem ist ein Spezialfall der *Karush-Kuhn-Tucker* Bedingungen, die ein notwendiges Kriterium für einen optimalen Punkt darstellen. Das involvierte λ wird *Lagrange Multiplikator* genannt.

Dass an einem potentiellen Minimum gelten muss, dass $\nabla f(x^*) = -\lambda \nabla g(x^*)$ haben wir anhand einer Zeichnung im 2D Fall festgestellt.

Für $x \in \mathbb{R}^n$ und einer Gleichungsnebenbedingung g, ist die Argumentation wie folgt.

Lemma 8.1. *Sei $x^* \in \mathbb{R}^n$ mit $g(x^*) = 0$ so, dass ∇f und ∇g **nicht** parallel sind. Dann gilt für die Richtung*

$$v := -\left(I - \frac{1}{\|\nabla g(x^*)\|^2} \nabla g(x^*) g(x^*)^T\right) \nabla f(x^*)$$

dass $\nabla g(x^*)^T v = 0$ und $\nabla f(x^*)^T v < 0$ ist.

Lemma 8.1 sagt, dass es im skizzierten Fall also eine Richtung gibt, in der (in erster Ableitung) die Funktion f minimiert werden kann ohne den Zulässigkeitsbereich zu verlassen.

8.3 Linear Quadratische Probleme

Ist die Zielfunktion als quadratische Funktion

$$f(x) = x^T Q x + c^T x$$

gegeben mit $Q \in \mathbb{R}^{n \times n}$ und $c \in \mathbb{R}^n$ und sind die Nebenbedingungen linear gegeben als

$$Ax \geq b$$

mit $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ und mit $Ax \geq b$ bedeuten soll, dass **alle Einträge** des Vektors $Ax - b$ kleiner oder gleich 0 sind, dann sprechen wir von einem (linearen) quadratischen Optimierungsproblem.

Hier müssen Gleichheitsbedingungen nicht explizit angeführt werden, da sie durch Ungleichungsbedingungen ausgedrückt werden können (vgl. $x = 0$ gdw. $x \geq 0$ und $-x \geq 0$).

Sind allerdings alle Bedingungen letztlich Gleichheitsbedingungen, liegt also das Problem als

$$x^T Q x + c^T x \to \min_{x \in \mathbb{R}^n} \quad \text{s.t. } Ax = b$$

vor, dann sind die notwendigen Optimalitätsbedingungen durch

$$\begin{bmatrix} Q + Q^T & A^T \\ A & 0 \end{bmatrix} = \begin{bmatrix} -c \\ b \end{bmatrix}.$$

Aus der Betrachtung dieser Optimalitätsbedingungen können wir folgern, dass, wenn Q symmetrisch ist und positiv definit, sowie alle Zeilen von A linear unabhängig sind, dass dann das Optimierungsproblem eine eindeutige Lösung hat, die durch die Optimalitätsbedingungen eindeutig charakterisiert ist.

8.4 Sequential Quadratic Programming

Aus der Beobachtung, dass wir LQP Probleme direkt mit Q symmetrisch positiv definit direkt lösen können und dass wir für kleine Abweichungen ξ von einem gebenen Punkt x multivariable Funktionen über quadratische (für unser f) und lineare Approximationen (für unsere g_i) gut annähern können:

$$f(x^*) = f(x + \xi) = f(x) + \nabla f(x)^T \xi + \xi^T H_f(x) \xi + o(\|\xi\|^2)$$

und

$$g_i(x^*) = g_i(x + \xi) = g_i(\xi) + \nabla g_i(\xi)^T h + o(\|\xi\|), \quad i \in \mathcal{I},$$

ergibt sich das folgende iterative Verfahren genannt *Sequential Linear Programming* (SQP):

Beginnend von einem Näherungswert x^k

1. Berechne d^k aus der Minimierung von[2]

$$\tilde{f}(x^k + d) := \nabla f(x^k)^T d + d^T H_f(x^k) d \to \min_{d \in \mathbb{R}^n},$$

unter der Nebenbedingung

$$\tilde{g}_i(x^k + d) := g(x^k) + \nabla g_i(x^k) d = 0 \quad i \in \mathcal{I}$$

2. ein Update $x^{k+1} = x^k + d^k$.

Es ist also in jeder Iteration ein linear quadratisches Optimierungsproblem

$$x^T Q_k x + c_k^T x \to \min_{x \in \mathbb{R}^n} \quad \text{s.t. } A_k x = b_k$$

zu lösen, wobei (in der obigen Notation mit Q, c, A und b) gilt dass

$$Q_k = H_f(x^k), \quad c_k = \nabla f(x_k)$$

[2]Die korrekte quadratische Approximation von f würde noch den Term $f(x^k)$ enthalten. Dieser kann aber einfach weggelassen werden bei der Suche der Minimalstelle.

und

$$A_k = [\nabla g_i(x_k)^T]_{i\in\mathcal{J}} \in \mathbb{R}^{|\mathcal{J}|\times n}, \quad b_k = -[g_i(x_k)]_{i\in\mathcal{J}} \in \mathbb{R}^{|\mathcal{J}|}.$$

Ist die Funktion f konvex und zweimal stetig differenzierbar (in einer Umgebung von x_k), dann ist die Hesse-matrix $H_f(x_k)$ positiv definit und das LQP hat eine eindeutige Lösung x_{k+1}.

8.5 Aufgaben

8.5.1 KKT für LQP (T)

Verifizieren Sie, dass die notwendigen Optimalitätsbedingungen für das LQP Problem mit Gleichungsnebenbedingungen (mit $m = 1$) genau der Optimalitäsbedingung aus Gleichung (8.1) entspricht. **Hinweis**: Mit der Definition der *totalen Ableitung* und der Relation zum Gradienten für reellwertige multivariable Funktionen (vgl. die Vorlesung *Mathe für DS2* vom 6. Juli) ist eine Darstellung der Gradienten $\nabla f(x)$ und $\nabla g(x)$ direkt herleitbar.

8.5.2 Nicht parallele Gradienten (T)

Verifizieren sie die Aussage aus Lemma 8.1.

8.5.3 Hauptkomponente aus SQP (T+P)

Skizzieren sie das Optimierungsproblems, das die **letzte** Hauptkomponentenrichtung definiert (also die Richtung mit minimaler Varianz) eines zentrierten Datensatzes definiert (analog dazu, wie in Abschnitt 4.3 eingeführt). **Hinweis**: Alles wird vielleicht etwas einfacher, wenn sie die Nebenbedingung $\|x\| = 1$ durch $\|x\|^2 = 1$ bzw. $x^Tx = 1$ ersetzen. (T)

Skizzieren einen SQP-Schritt zur Lösung dieses Problems. (T)

Implementieren sie die SQP Iteration zur Berechnung der letzten Hauptkomponentenrichtung für die Pinguin Daten. Berechnen Sie die Abweichung zur *exakten* (mit der SVD berechneten) Lösung und geben sie ihn in jedem Schritt beispielsweise den Winkel zwischen dem Iteranten und der exakten Loesung berechnen. **Hinweis**: Nur die Norm der Differenz könnte irritieren, weil das Vorzeichen der Richtungen nicht festgelegt ist.

```
import json
import numpy as np

with open('penguin-data.json', 'r') as f:
    datadict = json.load(f)

data = np.array(datadict['data'])
```

```
# center the data
X = data - data.mean(axis=0)
XtX = X.T @ X  # for later use

U, S, Vh = np.linalg.svd(X)

pcfour = Vh[-1:, :].T  # die "letzte" Hauptrichtung
nrmpcf = np.linalg.norm(pcfour)  # brauchen wir um den Winkel zu berechnen

xk = np.zeros((4, 1))  # Startvektor
xk[0] = 1
arccosalpha = xk.T @ pcfour / (np.linalg.norm(xk)*nrmpcf)
print(f'Iteration: {0} -- `arccos(sol, xk)` {arccosalpha}')


def solve_sadpoint_sys_x(Q=None, A=None, fone=None, ftwo=None):
    ''' Loesung des Sattelpunktprobles

    | Q  A.T | . | x |    | f1 |
    | A   0  |   | l |  = | f2 |

    Notes:
    -----

    Es wird nur `x` zurueckgegeben

    '''
    m = A.shape[0]
    lineone = [Q, A.T]
    linetwo = [A, np.zeros((m, m))]
    sdptmat = np.vstack([np.hstack(lineone),
                         np.hstack(linetwo)])
    rhs = np.vstack([fone, ftwo])
    xl = np.linalg.solve(sdptmat, rhs)
    return xl[:-m]  # return only the `x-part`


for kkk in range(1, 11):
    # ...
    # die SQP Iteration
    # ...
    xk = xk + solve_sadpoint_sys_x()

    arccosalpha = xk.T @ pcfour / (np.linalg.norm(xk)*nrmpcf)
    print(f'Iteration: {kkk} -- `arccos(sol, xk)` {arccosalpha}')
```

Chapter 9

Stochastistic Gradient and Learning

Dieses Kapitel ist kopiert (und, teilweise von mir und teilweise automatisch, übersetzt) aus dem Wikipedia Artikel (Stochastistic Gradient, Wikipedia contributors 2022). Dieses Kapitel steht unter der *WP:CC BY-SA* Lizenz.

Der stochastische Gradientenabstieg (oft als SGD abgekürzt) ist ein iteratives Verfahren zur Optimierung einer Zielfunktion mit geeigneten Glattheitseigenschaften (z. B. differenzierbar oder subdifferenzierbar). Sie kann als stochastische Näherung der Gradientenabstiegsoptimierung angesehen werden, da sie die tatsächliche Steigung (berechnet aus dem gesamten Datensatz) durch eine Schätzung davon ersetzt (berechnet aus einer zufällig ausgewählten Teilmenge der Daten). Insbesondere bei hochdimensionalen Optimierungsproblemen reduziert dies den sehr hohen Rechenaufwand, wodurch schnellere Iterationen, allerdings im Ausgleich für eine niedrigere Konvergenzrate, erreicht werden.

Während die Grundidee der stochastischen Approximation auf den Robbins-Monro-Algorithmus der 1950er Jahre zurückgeht, hat sich der stochastische Gradientenabstieg zu einer wichtigen Optimierungsmethode im maschinellen Lernen entwickelt.

9.1 Hintergrund

Sowohl in der Berechnung statistischer Schätzer als auch im *Maschinellen Lernen* spielt die Minimierung von Zielfunktionalen in Summenform

$$Q(w) = \frac{1}{N}\sum_{i=1}^{N} Q_i(w)$$

eine Rolle, wobei der Parameter w der Q minimiert gefunden oder geschätzt werden soll. Jede der Summandenfunktionen Q_i ist typischerweise assoziiert mit einem i-ten Datenpunkt (einer Beobachtung) beispielsweise aus einer Menge von Trainingsdaten.

Um obige Funktion zu minimieren, würde ein sogenannter Gradientenabstiegsverfahren den folgenden Minimierungsschritt

$$w^{k+1} := w^k - \eta \nabla Q(w^k) = w^k - \eta \frac{1}{N}\sum_{i=1}^{N} \nabla Q_i(w^k),$$

iterativ anwenden, wobei η die Schrittweite ist, die besonders in der *ML* community oft auch *learning rate* genannt wird.

Die Berechnung der Abstiegsrichtung erfordert hier also in jedem Schritt die Bestimmung von N Gradienten $\nabla Q_i(w^k)$ der Summandenfunktionen. Wenn N groß ist, also beispielsweise viele Datenpunkte in einer Regression beachtet werden sollen, dann ist die Berechnung entsprechend aufwändig.

Andererseits entspricht die Abstiegsrichtung

$$\frac{1}{N}\sum_{i=1}^{N} \nabla Q_i(w^k)$$

dem Mittelwert der Gradienten aller Q_is am Punkt w_k, der durch ein kleineres Sample

$$\frac{1}{N}\sum_{i=1}^{N} \nabla Q_i(w^k) \approx \frac{1}{|\mathcal{J}|}\sum_{j\in\mathcal{J}} \nabla Q_j(w^k),$$

wobei $\mathcal{J} \subset \{1, \ldots, N\}$ eine Indexmenge ist, die den *batch* der zur Approximation gewählten Q_is beschreibt.

9.2 Iterative method

Beim stochastischen (oder "Online") Gradientenabstieg wird der wahre Gradient von $Q(w^k)$ durch einen Gradienten bei einer einzelnen Probe angenähert:

$$w^{k+1} = w^k - \eta \nabla Q_j(w^k),$$

mit $j \in \{1, \dots, N\}$ zufällig gewählt (ohne zurücklegen).

Während der Algorithmus den Trainingssatz durchläuft, führt er die obige Aktualisierung für jede Trainingsprobe durch. Es können mehrere Durchgänge über den Trainingssatz gemacht werden, bis der Algorithmus konvergiert. Wenn dies getan wird, können die Daten für jeden Durchlauf gemischt werden, um Zyklen zu vermeiden. Typische Implementierungen können eine adaptive Lernrate verwenden, damit der Algorithmus konvergiert.

Die wesentlichen Schritte als Algorithmus sehen wie folgt aus:

```
######################################################
# The basic steps of a stochastic gradient method #
######################################################

w = ...  # initialize the weight vector
eta = ... # choose the learning rate
I = [1, 2, ..., N]  # the full index set

for k in range(maxiter):
    J = shuffle(I)  # shuffle the indices
    for j in J:
        # compute the gradient of Qj at current w
        gradjk = nabla(Q(j, w))
        # update the w vector
        w = w - eta*gradjk
    if convergence_criterion:
       break

######################################################
```

Die Konvergenz des *stochastischen Gradientenabstiegsverfahren* als Kombination von *stochastischer Approximation* und *numerischer Optimierung* ist gut verstanden. Allgemein und unter bestimmten Voraussetzung lässt sich sagen, dass das stochastische Verfahren ähnlich konvergiert wie das *exakte Verfahren* mit der Einschränkung, dass die Konvergenz *fast sicher* stattfindet.

In der Praxis hat sich der Kompromiss etabliert, der anstelle des Gradienten eines einzelnen Punktes $\nabla Q_j(w_k)$, den Abstieg aus dem Mittelwert über mehrere Samples berechnet, also (wie oben beschrieben)

$$\frac{1}{N}\sum_{i=1}^{N} \nabla Q_i(w^k) \approx \frac{1}{|\mathcal{J}|}\sum_{j\in\mathcal{J}} \nabla Q_j(w^k).$$

Im Algorithmus wird dann anstelle der zufälligen Indices $j \in \{1, \dots, N\}$, über zufällig zusammengestellte Indexmengen $\mathcal{J} \subset \{1, \dots, N\}$ iteriert.

Da die einzelnen Gradienten $\nabla Q_j(w^K)$ unabhängig voneinander berechnet werden können, kann so ein *batch* Verfahren effizient auf Computern mit mehreren

Prozessoren realisiert werden. Die Konvergenztheorie ist nicht wesentlich verschieden vom eigentlichen *stochastistischen Gradientenabstiegsverfahren*, allerdings erscheint die beobachte Konvergenz weniger erratisch, da der Mittelwert statistische Ausreißer ausmitteln kann.

9.3 Aufgabe

Schreiben Sie ein Programm, dass mit Hilfe eines neuronalen Netzes (NN) mit einer *hidden layer*

$$\eta_i = NN(x_i) := \tanh(A_2 \tanh(A_1 x_i + b_1) + b_2)$$

für einen Datenpunkt $x_i \in \mathbb{R}^{n_0}$, Gewichten $A_1 \in \mathbb{R}^{n_1 \times n_0}$, $b_1 \in \mathbb{R}^{n_1}$, $A_2 \in \mathbb{R}^{1,n_1}$, $b_2 \in \mathbb{R}^1$ und dem Ergebnisvektor $\eta_i \in \mathbb{R}^1$, anhand der gemessenen Daten x_i die bekannte Pinguin Population in zwei Gruppen aufteilt, so dass in der ersten Gruppe eine Spezies enthalten ist und in der anderen die beiden anderen Spezies.

Dazu kann eine Funktion $\ell\colon X \mapsto \{-1, 1\}$ definiert werden, die die bekannten Pinguine x_i aus dem Datensatz X ihrer Gruppe zuordnet. Dann können die Koeffizienten des NN über das Optimierungsproblem

$$\frac{1}{|X|} \sum_{x_i \in X} \|\ell(x_i) - NN(x_i)\|^2 \to \min_{A_1, b_1, A_2, b_2}$$

mittels des *stochastischen (batch) Gradientenabstiegs* bestimmt werden.

Hinweis: Für eine Funktion $f\colon w \mapsto f(w) \in \mathbb{R}$, können sie den Gradienten $\nabla_w f(w^*)$ an der Stelle w^* numerisch mit der Funktion `scipy.optimize.approx_fprime` bestimmen lassen.

Führen Sie die Optimierung auf einem Teil (z.B. 90%) der Pinguin Daten durch und testen sie wie gut ihr Netz funktioniert auf den verbleibenden Datenpunkten.

Der Beginn könnte also wie folgt aussehen (wobei die Größe der *hidden layer* zu $n_1 = 2$ gesetzt ist).

```
import json
import numpy as np
from scipy.optimize import approx_fprime

with open('penguin-data.json', 'r') as f:
    datadict = json.load(f)

data = np.array(datadict['data'])
# centering the data
data = data - data.mean(axis=0)
lbls = np.array(datadict['target'])
```

```
# a dictionary that maps the labels(=targets) of the data into labels {1, -1}
# that will use for distinction of two groups
mplbldict = {0: np.array([-1]),
             1: np.array([-1]),
             2: np.array([1])}

# sizes of the layers
sxz, sxo, sxt = data.shape[1], 2, mplbldict[0].size
# define also the sizes of the weightmatrices

# parameters for the training -- these worked fine for me
batchsize = 30  # how many samples for the stochastic gradients
lr = 0.25  # learning rate
iterations = 200  # how many gradient steps

# the data
traindataratio = .9  # the ratio of training data vs. test data
ndata = data.shape[0]
trnds = np.int(ndata*traindataratio)
allidx = np.arange(ndata)
trnidx = np.random.choice(allidx, trnds, replace=False)
tstidx = np.setdiff1d(allidx, trnidx)
```

Und die Funktion, die das neuronale Netz realisiert, so:

```
def fwdnn(xzero, Aone=None, bone=None, Atwo=None, btwo=None):
    ''' a neural networks of two layers

    '''
    xone = np.tanh(Aone @ xzero + bone)
    xtwo = np.tanh(Atwo @ xone + btwo)
    return xtwo
```

Für die Umsetzung ist es hilfreich, alle Koeffizienten in einen Vektor w vorzuhalten. Damit kann dann optimiert werden und bei Bedarf die Matrizen wieder hergestellt werden:

```
def wvec_to_wmats(wvec):
    ''' helper to turn the vector of weights into the system matrices

    '''
    Aone = wvec[:sxz*sxo].reshape((sxo, sxz))
    cidx = sxz*sxo
    bone = wvec[cidx:cidx+sxo]
    cidx = cidx + sxo
    Atwo = wvec[cidx:cidx+sxo*sxt].reshape((sxt, sxo))
```

```
    cidx = cidx + sxo*sxt
    btwo = wvec[cidx:]
    if Aone.size + bone.size + Atwo.size + btwo.size == wvec.size:
        return Aone, bone, Atwo, btwo
    else:
        raise UserWarning('mismatch weightsvector/matrices')
```

Damit fehlen zum Programm insbesondere noch

- die Zielfunktion der Optimierung in einem Datenpunkt x_i
- eine Funktion, die den stochastistischen Gradienten über einem kleinen *batch* von Daten ausrechnet
- eine Iteration, die das Gradientenabstiegsverfahren zusammen mit der *learning rate* durchführt
- ein Loop, der testet, wie das optimierte NN, die verbliebenen Daten klassifiziert

Hinweis: Auch hier ist wieder sehr viel Zufall involviert. Im Zweifel lieber zweimal testen.

Hinweis: Der letzte Testloop könnte so aussehen:

```
print('***** testing the classification *****')
faillst = []  # list to collect the failures for later examination
for cti in tstidx:  # iteration over the test data points
    itrgt = data[cti, :]
    ilbl = mplbldict[lbls[cti]]
    # the prediction of the neural network -- Aonex, ... are the optimized weights
    nnlbl = fwdnn(itrgt, Aone=Aonex, bone=bonex, Atwo=Atwox, btwo=btwox)
    sccs = np.sign(ilbl) == np.sign(nnlbl)
    print(f'label: {ilbl.item()} -- nn: {nnlbl.item():.4f} -- success: {sccs}')
    if not sccs:
        faillst.append((cti, ilbl.item(), nnlbl.item(),
                        datadict['target_names'][lbls[cti]]))
    else:
        pass

print('\n***** Results *****')
print(f'{100-len(faillst)/tstidx.size*100:.0f}% was classified correctly')
print('***** Misses *****')
if len(faillst) == 0:
    print('None')
else:
    for cfl in faillst:
        cid, lbl, nnlbl, name = cfl
        print(f'ID: {cid} ({name} pinguin) was missclassified ' +
              f'with score {nnlbl:.4f} vs. {lbl}')
```

Referenzen

Bollhöfer, M., Mehrmann, V.: Numerische Mathematik. Eine projektorientierte Einführung für Ingenieure, Mathematiker und Naturwissenschaftler. Vieweg (2004)

Nocedal, J., Wright, S.J.: Numerical optimization. Springer (2006)

Wikipedia contributors: Stochastic gradient descent — Wikipedia, the free encyclopedia, https://en.wikipedia.org/w/index.php?title=Stochastic_gradient_descent&oldid=1098148439, (2022)