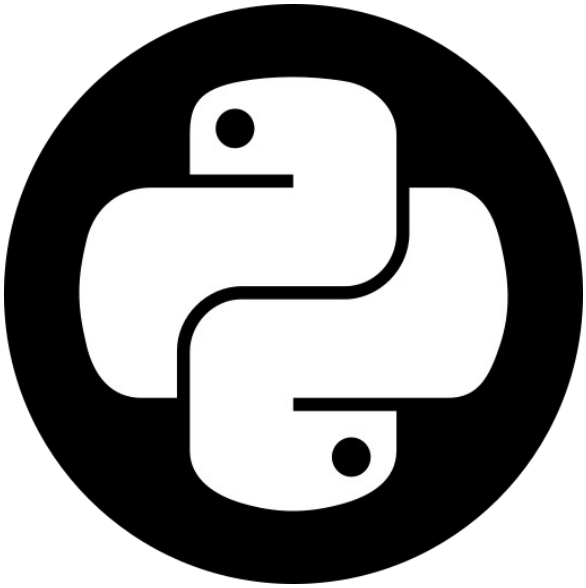
DATA SCIENCE MIT PYTHON

TUTORIALS



www.**STATISTICAL-THINKING**.de

1. TUTORIAL:

PAKETE UND DATENSTRUKTUREN

Dieses konstituierende Tutorial ermöglicht den **Einstieg** in Daten und Datenstrukturen in Python. Hier stehen die Bearbeitung von einzelnen **Fällen** (bspw. Personen in einem Datensatz), **Variablen** (bspw. Eigenschaften von Personen) und **Merkmalsausprägungen** (bspw. die konkrete Eigenschaft einer bestimmten Person) im Fokus. Dabei wird veranschaulicht, in welcher Form Daten in Python angelegt werden können, wobei die Unterschiede zwischen **Listen**, **Tuplen**, **Stacks** und **Arrays** verdeutlicht werden und wie man diese in Python bearbeiten kann. Für einen intuitiven Einstieg wird auf die **Passagierliste der R.M.S. Titanic** zurückgegriffen, deren Tragödie allen Teilnehmerinnen und Teilnehmern nicht zuletzt seit der Verfilmung im Jahr 1997 vertraut sein sollte.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial ist zunächst das Paket **pandas** über den Befehl **import** zu aktivieren. **Pakete** beinhalten mehrere **Dateien** und **Module** für Python, die so hinterlegt sind, dass während des Programmierens einfach auf diese zurückgegriffen werden kann. In den Modulen befinden sich in der Regel **vorgefertigte Codes**, so dass der Code nicht immer neu programmiert werden muss, sondern auf diese Weise abgekürzt werden kann:

```
# Vorbereitungen
import pandas as pd
```

Daraufhin kann über das Paket **pandas** mit dem Befehl **read_csv()** der Datensatz für dieses Tutorial eingelesen werden. **CSV** steht dabei für **Comma Separated Values** und bezeichnet das Dateiformat des entsprechenden Datensatzes. Alternative Dateiformate wie bspw. XLS für einen Microsoft Excel Datensatz können bspw. über den Befehl read_excel eingelesen werden. Hier wird der Datensatz als neues Objekt mit der Bezeichnung **df** für **Data Frame** eingelesen und für die weitere Verwendung in Python angelegt:

```
# Datensatz einlesen
df = pd.read_csv('datasets/data_one.csv')
```

DATENSATZ EINSEHEN

Zunächst sollen die Bezeichnungen für die einzelnen Spalten im Datensatz (hier: Variablenbezeichnungen) ausgegeben werden. Dies ist über den Befehl print() möglich, wenn die Spalten als **columns** und deren Bezeichnung als **values** spezifiziert werden:

```
# Datensatz einsehen (I)
print(df.columns.values)
```

Um zusätzlich die zu den Variablen die im Datensatz hinterlegten **Fälle** und die dazugehörigen **Merkmalsausprägungen** ausgeben zu lassen, steht der Befehl **head()** zur Verfügung. Dieser gibt nur die ersten fünf Fälle des Datensatzes aus, wobei der erste Fall - und somit die erste Zeile - in Python als 0 ausgegeben wird:

```
# Datensatz einsehen (II)
df.head()
```

LISTEN UND DIE AUSWAHL VON ELEMENTEN

Einzelne **Fälle** und ihre dazugehörigen **Merkmalsausprägungen** werden über sogenannte **Listen** und den Befel **list()** ausgegeben. Hierzu sind Angaben über die **Startzeile**, welche als Element in der Liste enthalten sein soll, und die **Endzeile**, welche nicht mehr als Element in der Liste enthalten sein soll, erforderlich. Im nachfolgenden Beispiel wird eine Liste für die ersten fünf Passagierinnen und Passagiere der R.M.S. Titanic in Bezug auf deren Geschlecht aufgerufen, Die Fokussierung auf ausgewählte Fälle nennt sich **Slicing**, oder auf Deutsch: Das Herausschneiden von Fällen:

```
# Listen und Elemente (I)
list(df.sex[0:5])
```

Legt man die **Elemente** in der Listen in Python als **neues Objekt** an, können die darin enthaltenen Merkmalsausprägungen auch verändert werden, bspw. um diese zu korrigieren. Sollte man nämlich feststellen, dass Miss. Elisabeth Walton Allen eigentlich ein Mann war, der sich nur als Frau ausgegeben hat, um auf ein Rettungsboot gelassen zu werden, so lässt sich die entsprechende Merkmalsausprägung in der Variable **sex** von **female** auf **male** ändern. Sowohl das Anlegen neuer Objekte als auch die Veränderung der darin enthaltenen Merkmalsausprägungen kann über den Rückgriff auf ein einfaches **Gleichheitszeichen** durchgeführt werden:

```
# Listen und Elemente (II)
neues_objekt = df.sex[0:5]
neues_objekt[0] = "male"
neues_objekt
```

Ebenfalls sind **einfache Rechenoperationen** mit den Elementen einer Liste möglich. Wenn die ersten fünf Fälle von der 1. Klasse in die 2. Klasse umgebucht werden sollen, könnte man entweder wie zuvor die entsprechende Merkmalsausprägung über ein Gleichheits-zeichen zuweisen, oder aber die ursprüngliche Merkmalsausprägung mit dem Wert 2 **multiplizieren**:

```
# Listen und Elemente (III)
ein_neues_objekt = df.pclass[0:5]
noch_ein_neues_objekt = ein_neues_objekt * 2
noch_ein_neues_objekt
```

TUPLES ALS SAMMLUNG VON ELEMENTEN AUS STACKS

Ein Tuple ist eine **geordnete, sequenzielle Sammlung von Elementen**. Es kann Daten aller Typen aufnehmen, die wie zuvor individuell adressiert werden können. In einem Tupel könnte bspw. hinterlegt werden, dass es **unterschiedliche Klassen** auf der R.M.S. Titanic gegeben hat und dass die Passagierinnen und Passagiere **unterschiedliche Geschlechter** hatten. Demnach würde der Tupel über die zahlenbasierten Merkmalsausprägungen **0**, **1**, **2** sowie die wortbasierten Merkmalsausprägungen **female**, **male** informieren.

```
# Tuples (I)
neuer_tuple = (1, 2, 3, "female", "male")
neuer_tuple
```

Einfache **Rechenoperationen** beziehen sich aufgrund der unterschiedlichen Datentypen allerdings **nicht auf die einzelnen Elemente** des Tuples, sondern auf den **gesamten Tuple**. Eine Multiplikation mit dem Wert 2 verdoppelt demnach einfach nur die geordnete, sequenzielle Sammlung von Elementen:

```
# Tuples (II)
neuer_tuple * 2
```

STACKS UND DAS ZIEHEN VON ELEMENTEN AUS STACKS

Möchte man keine reine Liste (oder einen Tupel) mit einer fixen Anzahl von Elementen haben, sondern wie bei einem Kartenspiel **einzelne Elemente** aus dem Stapel an Karten (Englisch: Stack) **ziehen**, so hilft ein Rückgriff auf Stacks. Diese beinhalten nicht nur Elemente, sondern Elemente können über die Befehle **append()** und **pop()** hinzugefügt und auch wieder entfernt werden, woraufhin sich die **Anzahl an Elementen in dem Stack verändern**. Hier wird zunächst eine neue 4. Klasse an Bord der R.M.S. Titanic hinzugefügt und oben (auf den Stapel) hinzugefügt:

```
# Stacks (I)
neues_stack = [1, 2, 3]
neues_stack.append(4)
neues_stack
```

Entsprechend lässt sich das **oberste Element** des Stacks **ziehen**:

```
# Stacks (II)
neues_stack.pop()
neues_stack
```

FOKUS AUF RELEVANTE VARIABLEN ALS FEATURES

Für die Analyse relevante Variablen werden auch Features genannt. Um insbesondere bei größeren Datensätzen einen **besseren Überblick** behalten zu können, aber auch um **ressourcenschonend** zu analysieren, kann ein Fokus auf eine Auswahl an Variablen gelegt werden. Die verbleibenden Variablen werden dabei sprichwörtlich über den **drop()** Befehl "fallengelassen". Der **shape()** Befehl informiert dabei anschaulich über die Anzahl an **Fällen** und **Variablen** im Datensatz:

```
# Variablenauswahl
print("Alle Variablen:", df.shape)
smaller_df = df.drop(['ticket', 'cabin'], axis='columns')
print("Variablenauswahl:", smaller_df.shape)
```

FEHLWERTE

Viele Datensätze beinhalten **fehlende**, **nicht definierte** oder **nicht darstellbare** Werte: Diese werden in Python als **Not a Number (NaN)** ausgegeben. Mathematische und statistische Operationen erfordern bei einer tabellarischen Datenstruktur allerdings vollständig vorliegende, definierte und darstellbare Werte. Im Datensatz zur R.M.S. Titanic liegen NaN-Werte bspw. innerhalb der Variable **boat** für diejenigen Fälle vor, die es leider nicht an Bord eines Rettungsbootes mit einer vorliegenden, definierten und darstellbaren Nummer geschafft haben. Mittels **pandas** lässt sich deren Anzahl über den Befehl **isna()** aufrufen:

```
# Fehlwerte
df['boat'].isna().sum()
```

FEATUREÜBERGREIFENDE ZUSAMMENHÄNGE

Um Zusammenhänge analysieren zu können, ist ein **variablenübergreifender Fokus** erforderlich. Dazu werden in Python einfach die **relevanten Variablen** (bzw. Spalten) ausgewählt und mit statistischen Formeln wie bspw. dem **mean()** Befehl in Verbindung gesetzt. Über zusätzliche Vorgaben kann hierbei spezifiziert werden, dass der Output g**ruppiert nach der Schiffsklasse** und dabei **sortiert in aufsteigender Reihenfolge** ausgegeben werden sollen. Dies wird über die Befehle **groupby()** und **sort_values()** spezifiziert:

```
# Featureübergreifende Zusammenhänge
df[['pclass', 'survived']].groupby(['pclass'], as_index=False).mean().sort_values(by='survived',
ascending=False)
```

Die **absteigende Reihenfolge der Überlebenden bei gleichzeitigem Anstieg der Schiffsklasse** deutet einen entsprechenden Zusammenhang zwischen diesen beiden Variablen an. Wie derartige Zusammenhänge analysiert werden können, ist Bestandteil der weiteren Tutorials.

2. TUTORIAL:

REGRESSIONSANALYSE

Das zweite Tutorial führt durch die **uni-**, **bi- und multivariate Statistik** zum beschreiben, erklären und vorhersagen von Variablen in Datensätzen. Dies ermöglicht nicht nur einen Überblick über die **Häufigkeiten der Merkmalsausprägungen** in den einzelnen Variablen, sondern auch bezüglich deren **Zusammenhänge** im Falle von jeweils zwei Variablen und deren **Interaktionen** bei mehr als zwei Variablen. Die zusammenfassenden Befunde werden über ein lineares Regressionsmodell ausgewiesen.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **pandas**, **matplotlib.pylot** sowie **sklearn.linear_model** über den Befehl **import** zu aktivieren:

```
# Vorbereitungen
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

Daraufhin kann über das Paket **pandas** wieder mit dem Befehl **read_csv()** der Datensatz für dieses Tutorial eingelesen werden. Hier wird der **Datensatz als neues Objekt** mit der Bezeichnung **df** für Data Frame eingelesen und für die weitere Verwendung in Python angelegt:

```
# Datensatz einlesen
df = pd.read_csv('datasets/data_two.csv')
```

UNIVARIATE STATISTIK

In der univariaten Statistik wird der **Fokus auf die einzelnen Variablen** und deren Merkmalsausprägungen gelegt. Der Befehl **describe()** soll sich dabei auf den Datensatz **df** beziehen, weshalb diese über einen Punkt verbunden in Python adressiert werden. Das Ergebnis ist eine Darstellung der **Minimum- und Maximumwerte** der Variablen, in Verbindung mit ausgewählten **Lagemaßen**. Diese beschreiben den Schwerpunkt (auch: das Zentrum) in der Verteilung von Merkmalsausprägungen innerhalb einer Variable. Hier werden **mean** als **arithmetisches Mittel** und der Median als ausgewiesen. Das arithmetische Mittel ist der errechenbare Durchschnittswert und der **Median** ist der **Wert in der Mitte**, wenn man die Merkmalsausprägungen in aufsteigender Reihenfolge sortiert. Darüber hinaus wird die Standardabweichung **std** ausgewiesen, welche ein Maß für die **Streuung der Merkmalsausprägungen um das arithmetische Mittel** darstellt:

```
# Univariate Statistik (I)
df.describe()
```

Die Kennwerte der univariaten Statistik lassen sich über den Befehl **boxplot()** in sogenannte Boxplots überführen. Dabei markieren die "Antennen" die Minimum- und

Maximumwerte und die untere bzw. obere Grenze der **"Box"** die Merkmalsausprägungen die für bis zu **25% der Fälle** gelten, bzw. für bis zu **75% der Fälle**. Entsprechend liegt der **Median mit 50%** innerhalb der "Box". **Statistische Ausreißer** sind dabei als einzelne Punkte abgebildet:

```
# Univariate Statistik (II)
df.boxplot()
title_boxplot = 'Descriptive Statistics'
plt.title(title_boxplot)
```

BIVARIATE STATISTIK

In der bivariaten Statistik geht es um die **Zusammenhänge zwischen jeweils zwei Variablen**. Von besonderem Interesse sind dabei zunächst die Zusammenhänge zwischen den **unabhängigen Variablen (X)** mit der **abhängigen Variable (Y)**. Bei intervall- oder verhältnisskalierten Variablen kann der Zusammenhang über den Befehl **corr()** mittels Korrelation **auf Linearität überprüft** werden. Liegen alle X-Y-Datenpunkte auf einer Geraden, so handelt es sich in Abhängigkeit der Steigung um einen Zusammenhang von r=1.000 bzw. r=-1.000. Liegen die X-Y-Datenpunkte weiter vom möglichen Verlauf einer Geraden weg, so kann der **Korrelationskoeffizient** bis auf den Wert r=0.000 sinken.

```
# Bivariate Statistik (I)
df.corr()
```

Zunächst wird der Zusammenhang zwischen den Variablen X1 und X2 mit der Y-Variable mit dem Befehl **scatter()** dargestellt. Darüber hinaus können Sie vergleichen, **wie linear die X-Y-Datenpunkte verlaufen**. Vergleichen Sie die Linearität mit den vorherigen Korrelationskoeffizienten:

```
# Bivariate Statistik (II)
plt.scatter(df.Y, df.X1)
plt.scatter(df.Y, df.X2)
plt.show()
```

MULTIVARIATE STATISTIK

Für die multivariate Statistik muss der Datensatz **df** in ein für die lineare Regression **interpretierbares Format** überführt werden. Dies bedeutet, dass die **Y-Variable** über den Befehl **array()** aus dem Paket **np** eindeutig **als abhängige Variable in Python hinterlegt** wird. Der **iloc[]** Befehl wählt darüber hinaus die **X-Variablen** in den Spalten 0 bis 1 aus und hinterlegt diese entsprechend. Merke: In Python beginnt die Spaltenzählung bei "0":

```
# Multivariate Statistik (I)
y = np.array(df.Y)
x = df.iloc[:,[0,1]]
```

Die lineare Regression wird schließlich über den Befehl **LinearRegression()** aufgerufen und als neues Objekt **model** hinterlegt:

```
# Multivariate Statistik (II)
model = LinearRegression()
model.fit(x, y)
```

Dem Objekt model lassen sich daraufhin relevante Kennwerte entnehmen. Relevante Kennwerte sind u.a. der **Determinationskoeffizient** als Bestimmtheitsmaß, welcher in Python über den Befehl **score()** als **r_sq** angelegt werden kann. Der Determinationskoeffizient gibt Auskunft darüber, wie viel **Prozent der Varianz der abhängigen Variable durch die unabhängigen Variablen erklärt** werden kann. Darüber hinaus werden mit **intercept** der Intercept-Wert, also der **Schnittpukt mit der Y-Achse** und mit **coef**_ die Koeffizienten als **Regressionsgewichte der unabhängigen Variablen** aufgerufen. Vergleichen Sie Stärke und Richtung der Regressionsgewichte mit den Korrelationskoeffizienten. Welche Unterschiede ergeben sich dabei?

```
# Multivariate Statistik (III)
r_sq = model.score(x, y)
print('coefficient of determination:', r_sq)
print('intercept:', model.intercept_)
print('coefficients:', model.coef_)
```

Zur **Demonstration der Vorhersage** wird über den **iloc[]** Befehl exemplarisch **ein Fall mit den dazugehörigen Merkmalsausprägungen** in den jeweiligen Variablen hervorgehoben. Dessen Merkmalsausprägungen werden in Verbindung mit den Befunden des linearen Regressionsmodells für eine lineare Gleichung zur **Vorhersage der dazugehörigen Y-Variable** herangezogen. Wie präzise ist die Vorhersage?

```
# Probe
df.iloc[12,]
# Y1 = -57.99 + 4.71 * 11.4 + 0.34 * 76
```

Über den Befehl **predict(x)** werden die Merkmalsausprägungen der unabhängigen Variablen in das lineare Regressinsmodell überführt, so dass **neue Merkmals-ausprägungen für die Y-Variable errechnet** werden können. Die neue Y-Variable ermöglicht daraufhin den **Abgleich mit den tatsächlichen Merkmalsausprägungen** der abhängigen Variable, um die Präzision des linearen Regressionsmodells einschätzen zu können:

```
# Multivariate Statistik (IV)
model.predict(x)
```

Zur Einschätzung der **Präzision des linearen Regressionsmodells** lässt sich die Differenz zwischen den Modellvorhersagen, hier **predictions**, mit den tatsächlichen Merkmalsausprägungen der abhängigen Variable, hier **y**, errechnen. Diese Differenz wird als **residuals** in Python, auch **Residuum** genannt, angelegt:

```
# Multivariate Statistik (V)
predictions = model.predict(x)
residuals = y - predictions
print('residuals:', residuals)
```

Schließlich lässt sich das **Residuum grafisch darstellen**. Hierbei sollten **keine Muster oder Strukturen erkennbar** sein, da die Muster und Strukturen des Datensatzes mit dem linearen Modell bereits abgedeckt sein sollten. Ist über den Befehl **scatter()** im Plot des

Residuums ein Muster oder eine Struktur erkennbar, könnte dies ein Hinweis sein, dass ein lineares Regressionsmodell nicht das Mittel der Wahl gewesen ist:

```
# Multivariate Statistik (VI)
plt.scatter(predictions, residuals)
plt.hlines(0, 5, 70, color="red", linestyle="dotted")
plt.show()
```

Das hier vorliegende Residuum verhält sich **aufgrund der Linearität** des zugrundeliegenden Zusammenhangs **erwartungsgemäß**.

3. TUTORIAL:

INTERAKTIONSEFFEKTE

In diesem Tutorial geht es um **Interaktionseffekte in multivariaten Datensätzen**. Ein Interaktionseffekt bezeichnet in statistischen Verfahren **nicht-additive Effekte zweier oder mehrerer unabhängiger Variablen** in einem Wahrscheinlichkeitsmodell. Das bedeutet, dass für die durch den Beobachtungsraum repräsentierte Ereignismenge angenommen wird, dass die **Wirkung der Ausprägungen einer dieser Variablen von den Ausprägungen der jeweils andere(n) Variable(n) abhängt**.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **pandas**, **matplotlib.pylot** sowie **sklearn.linear_model** über den Befehl **import** zu aktivieren:

```
# Vorbereitungen
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

Daraufhin kann über das Paket **pandas** wieder mit dem Befehl **read_csv()** der Datensatz für dieses Tutorial eingelesen werden. Hier wird der **Datensatz als neues Objekt** mit der Bezeichnung **df** für Data Frame eingelesen und für die weitere Verwendung in Python angelegt:

```
# Datensatz einlesen
df = pd.read_csv('datasets/data_three.csv')
```

UNIVARIATE STATISTIK

Erneut wird in der univariaten Statistik der **Fokus auf die einzelnen Variablen** und deren Merkmalsausprägungen gelegt. Der Befehl **describe()** soll sich dabei auf den Datensatz **df** beziehen, weshalb diese über einen Punkt verbunden in Python adressiert werden. Das Ergebnis ist eine Darstellung der **Minimum- und Maximumwerte** der Variablen, in Verbindung mit ausgewählten **Lagemaßen**. Diese beschreiben den Schwerpunkt (auch: das Zentrum) in der Verteilung von Merkmalsausprägungen innerhalb einer Variable. Hier werden **mean** als **arithmetisches Mittel** und der Median als ausgewiesen. Das arithmetische Mittel ist der errechenbare Durchschnittswert und der **Median** ist der **Wert in der Mitte**, wenn man die Merkmalsausprägungen in aufsteigender Reihenfolge sortiert. Darüber hinaus wird die Standardabweichung **std** ausgewiesen, welche ein Maß für die **Streuung der Merkmalsausprägungen um das arithmetische Mittel** darstellt:

```
# Univariate Statistik (I)
df.describe()
```

Die Kennwerte der univariaten Statistik lassen sich über den Befehl **boxplot()** in sogenannte Boxplots überführen. Dabei markieren die "Antennen" die Minimum- und

Maximumwerte und die untere bzw. obere Grenze der **"Box"** die Merkmalsausprägungen die für bis zu **25% der Fälle** gelten, bzw. für bis zu **75% der Fälle**. Entsprechend liegt der **Median mit 50%** innerhalb der "Box". **Statistische Ausreißer** sind dabei als einzelne Punkte abgebildet:

```
# Univariate Statistik (II)
df.boxplot()
title_boxplot = 'Descriptive Statistics'
plt.title(title_boxplot)
```

BIVARIATE STATISTIK

Auch in der bivariaten Statistik geht es wieder um die **Zusammenhänge zwischen jeweils zwei Variablen**. Von besonderem Interesse sind dabei zunächst die Zusammenhänge zwischen den **unabhängigen Variablen (X)** mit der **abhängigen Variable (Y)**. Bei intervall- oder verhältnisskalierten Variablen kann der Zusammenhang über den Befehl **corr()** mittels Korrelation **auf Linearität überprüft** werden. Liegen alle X-Y-Datenpunkte auf einer Geraden, so handelt es sich in Abhängigkeit der Steigung um einen Zusammenhang von r=1.000 bzw. r=-1.000. Liegen die X-Y-Datenpunkte weiter vom möglichen Verlauf einer Geraden weg, so kann der **Korrelationskoeffizient** bis auf den Wert r=0.000 sinken.

```
# Bivariate Statistik (I)
df.corr()
```

Zunächst wird der Zusammenhang zwischen den Variablen X1 und X2 mit der Y-Variable mit dem Befehl **scatter()** dargestellt. Darüber hinaus können Sie vergleichen, **wie linear die X-Y-Datenpunkte verlaufen**. Vergleichen Sie die Linearität mit den vorherigen Korrelationskoeffizienten und erweitern Sie den Plot eigenständig um eine **Darstellung der Variablen X3 bis X5**:

```
# Bivariate Statistik (II)
plt.scatter(df.Y, df.X1)
plt.scatter(df.Y, df.X2)
plt.show()

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

MULTIVARIATE STATISTIK

Für die multivariate Statistik wird der Datensatz **df** wieder in ein für die lineare Regression **interpretierbares Format** überführt. Dies bedeutet, dass die **Y-Variable** über den Befehl **array()** aus dem Paket **np** eindeutig **als abhängige Variable in Python hinterlegt** wird. Der **iloc[]** Befehl wählt darüber hinaus die **X-Variablen** in den Spalten 1 bis 5 aus und hinterlegt diese entsprechend:

```
# Multivariate Statistik (I)
y = np.array(df.Y)
x = df.iloc[:,[1,2,3,4,5]]
```

Die lineare Regression wird schließlich wieder über den Befehl **LinearRegression()** aufgerufen und als neues Objekt **model** hinterlegt:

```
# Multivariate Statistik (II)
model = LinearRegression()
model.fit(x, y)
```

Über den Befehl **predict(x)** werden die Merkmalsausprägungen der unabhängigen Variablen schließlich wieder in das lineare Regressinsmodell überführt, so dass **neue Merkmalsausprägungen für die Y-Variable errechnet** werden können. Die neue Y-Variable ermöglicht daraufhin den **Abgleich mit den tatsächlichen Merkmalsausprägungen** der abhängigen Variable, um die Präzision des linearen Regressionsmodells einschätzen zu können:

```
# Multivariate Statistik (III)
model.predict(x)
```

Insbesondere mit Blick auf mögliche Interaktionseffekte lassen sich dem Objekt **model** relevante Kennwerte entnehmen. Relevante Kennwerte sind u.a. der **Determinationskoeffizient** als Bestimmtheitsmaß, welcher in Python über den Befehl **score()** als **r_sq** angelegt werden kann. Der Determinationskoeffizient gibt Auskunft darüber, wie viel **Prozent der Varianz der abhängigen Variable durch die unabhängigen Variablen erklärt** werden kann. Darüber hinaus werden mit **intercept** der Intercept-Wert, also der **Schnittpukt mit der Y-Achse** und mit **coef**_ die Koeffizienten als **Regressionsgewichte der unabhängigen Variablen** aufgerufen. Vergleichen Sie hier insbesondere Stärke und Richtung der Regressionsgewichte mit den Korrelationskoeffizienten. Welche unabhängige Variable hat einen **Vorzeichenwechsel** erfahren und welche **unabhängigen Variablen haben sich in ihrer Effektstärke angeglichen**?

```
# Multivariate Statistik (IV)
r_sq = model.score(x, y)
print('coefficient of determination:', r_sq)
print('intercept:', model.intercept_)
print('coefficients:', model.coef_)
```

Die Diagnostik des Regressionsmodels erfolgt ebenfalls analog dem vorherigen Tutorial. Zur Einschätzung der **Präzision des linearen Regressionsmodells** lässt sich wieder die Differenz zwischen den Modellvorhersagen, hier **predictions**, mit den tatsächlichen Merkmalsausprägungen der abhängigen Variable, hier **y**, errechnen. Diese Differenz wird als **residuals** in Python, auch **Residuum** genannt, angelegt:

```
# Multivariate Statistik (V)
predictions = model.predict(x)
residuals = y - predictions
print('residuals:', residuals)
```

Schließlich lässt sich das **Residuum grafisch darstellen**. Hierbei sollten **keine Muster oder Strukturen erkennbar** sein, da die Muster und Strukturen des Datensatzes mit dem linearen Modell bereits abgedeckt sein sollten. Ist über den Befehl **scatter()** im Plot des

Residuums ein Muster oder eine Struktur erkennbar, könnte dies ein Hinweis sein, dass ein lineares Regressionsmodell nicht das Mittel der Wahl gewesen ist:

```
# Multivariate Statistik (VI)
plt.scatter(predictions, residuals)
plt.hlines(0, 35, 90, color="red", linestyle="dotted")
plt.show()
```

Auch das hier vorliegende Residuum verhält sich **aufgrund der Linearität** des zugrundeliegenden Zusammenhangs **erwartungsgemäß**.

4. TUTORIAL:

MUSTERERKENNUNG

Das vierte Tutorial dient einer Einführung in die **Mustererkennung** auf Basis der **Merkmalsausprägungen** in den einzelnen Variablen, den sogenannten **Features**. Diese können bspw. herangezogen werden, um ähnliche Fälle im Datensatz zu **identifizieren** und entsprechend zu **klassifizieren**.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **pandas**, sowie **matplotlib.pylot** über den Befehl **import** zu aktivieren:

```
# Vorbereitungen
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Daraufhin kann über das Paket **pandas** wieder mit dem Befehl **read_csv()** der Datensatz für dieses Tutorial eingelesen werden. Hier wird der **Datensatz als neues Objekt** mit der Bezeichnung **df** für Data Frame eingelesen und für die weitere Verwendung in Python angelegt:

```
# Datensatz einlesen
df = pd.read_csv('datasets/data_four.csv')
```

UNIVARIATE STATISTIK

Diesmal ist die univariate Statistik der Schlüssel zur eigentlichen Mustererkennung. Der Befehl **describe()** soll sich dabei auf den Datensatz **df** beziehen, weshalb diese über einen Punkt verbunden in Python adressiert werden. Das Ergebnis ist die bereits bekannte Darstellung der **Minimum- und Maximumwerte** der Variablen, in Verbindung mit ausgewählten **Lagemaßen**. Diese beschreiben den Schwerpunkt (auch: das Zentrum) in der Verteilung von Merkmalsausprägungen innerhalb einer Variable. Hier werden **mean** als **arithmetisches Mittel** und der Median als ausgewiesen. Das arithmetische Mittel ist der errechenbare Durchschnittswert und der **Median** ist der **Wert in der Mitte**, wenn man die Merkmalsausprägungen in aufsteigender Reihenfolge sortiert. Darüber hinaus wird die Standardabweichung **std** ausgewiesen, welche ein Maß für die **Streuung der Merkmalsausprägungen um das arithmetische Mittel** darstellt:

```
# Univariate Statistik (I)
df.describe()
```

Die Kennwerte der univariaten Statistik lassen sich über den Befehl **boxplot()** in sogenannte Boxplots überführen. Dabei markieren die "Antennen" die Minimum- und Maximumwerte und die untere bzw. obere Grenze der **"Box"** die Merkmalsausprägungen die für bis zu **25% der Fälle** gelten, bzw. für bis zu **75% der Fälle**. Entsprechend liegt der **Median mit 50%** innerhalb der "Box". **Statistische Ausreißer** sind dabei wieder als

einzelne Punkte abgebildet. Allerdings ermöglicht diese Darstellung **keinen detaillierten Einblick** in die unterschiedlichen **Muster der Gruppen A**, **B und C**:

```
# Univariate Statistik (II)
df.boxplot()
title_boxplot = 'Descriptive Statistics'
plt.title(title_boxplot)
```

DATENAUFBEREITUNG

Die Y-Variable ermöglicht mit ihren **Merkmalsausprägungen** die **Unterscheidung zwischen den drei Gruppen** A, B und C. Über die entsprechenden Merkmalsausprägungen lassen sich in Verbindung mit dem Befehl **loc[]** jeweils **Teildatensätze** anlegen, die **ausschließlich Fälle einer Gruppe** beinhalten. Diese Teildatensätze ermöglichen im nächsten Schritt eine **Identifikation der Muster** und somit der Unterschiede zwischen den drei Gruppen:

```
# Datenaufbereitung (I)
group_a_df = df.loc[(df.Y == "A")]
group_b_df = df.loc[(df.Y == "B")]
group_c_df = df.loc[(df.Y == "C")]
```

Die Identifikation der Muster und Unterschiede kann über den bereits bekannten Befehl **boxplot()** erfolgen. Um den Vergleich zwischen den **Gruppen A**, **B und C** zu vereinfachen, werden die **Achsenabschnitte** in den jeweiligen Plots über den Befehl **ylim()** einheitlich vorgegeben. Das Ergebnis sind unterschiedliche Minimumwerte und Maximumwerte bei den X-Variablen, welche über den Befehl **subplot** anschaulich nebeneinander abgebildet werden können:

```
# Datenaufbereitung (II)
plt.subplot(1, 3, 1)
group_a_df.boxplot()
plt.ylim(0, 8)
title_boxplot = "Gruppe A"
plt.title(title_boxplot)
plt.subplot(1, 3, 2)
group_b_df.boxplot()
plt.ylim(0, 8)
title_boxplot = "Gruppe B"
plt.title(title_boxplot)
plt.subplot(1, 3, 3)
group_c_df.boxplot()
plt.ylim(0, 8)
title_boxplot = "Gruppe C"
plt.title(title_boxplot)
plt.show()
```

Um die **exakten Minimumwerte und Maximumwerte** in den jeweiligen Gruppen aufzurufen, wird erneut auf den Befehl **describe()** zurückgegriffen. Für Gruppe A ist der Output exemplarisch dargestellt. **Wiederholen Sie diesen Schritt für Gruppe B und C**:

```
# Datenaufbereitung (III)
group_a_df.describe()

##################################
### SIEHE AUFGABENSTELLUNG ###
##################################
```

IDENTIFIKATION ÜBER ARGUMENTE

Anschließend können über die exakten Minimumwerte und Maximumwerte die **Argumente definiert** werden, die in den einzelnen X-Variablen gegeben sein müssen, um eine Gruppe, hier exemplarisch für Gruppe A dargestellt, **korrekt identifizieren** zu können. Ausschlaggebend hierfür sind die **unterschiedlichen Merkmalsausprägungen** in den jeweiligen Gruppen. **Wiederholen Sie auch diesen Schritt für die Gruppen B und C**:

```
# Manuelle Identifikation
identification_group_a_df = df.loc[(df.X3 <= 1.9) & (df.X4 <= 0.6)]
print('correct:', identification_group_a_df.Y.count(), '/ 50')

##################################
### SIEHE AUFGABENSTELLUNG ###
##################################
```

In Gruppe A können **50 von 50 Fällen korrekt identifiziert** werden. Wie viele Fälle konnten Sie in den Gruppen B und C korrekt identifizieren?

5. TUTORIAL:

MACHINE LEARNING

Das vierte Tutorial knüpft an dem vorherigen Tutorial zur Einführung in die **Mustererkennung** auf Basis der **Merkmalsausprägungen** in den einzelnen Variablen an. Dabei sollen die **Ähnlichkeiten** über sogenannte Machine Learning Algorithmen automatisch identifiziert werden. In diesem Tutorial werden einzelne Schritte wie die Einteilung in einen **Trainings- und einen Testdatensatz** sowie die Festlegung der zu klassifizierenden Merkmalsausprägungen in der Zielvariabe manuell vorgenommen, weshalb es sich um das sogenannte **Supervised Machine Learning** handelt. Die unabhängigen Variablen werden in diesem Kontext **Features** genannt.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **pandas**, **matplotlib.pylot** sowie **sklearn** über den Befehl **import** zu aktivieren. Innerhalb von **sklearn** stehen verschiedene Machine Learning Algorithmen zur Verfügung, darunter **Support Vector Machine**, **K-Means**, **K-Nearest-Neighbor**, **Random Forest** und **Linear-Discriminant-Analysis**. Um die Performance der Machine Learning Algorithmen zu vergleichen, wird zusätzlich auf **accuracy_score** zurückgegriffen. Für die Datenaufbereitung stehen schließlich **LabelEncoder** und **train_test_split** zur Verfügung:

```
# Vorbereitungen
import numpy as np
import pandas as pd
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

Daraufhin kann wieder über das Paket **pandas** mit dem Befehl **read_csv()** der Datensatz für dieses Tutorial eingelesen werden. Hier wird der Datensatz wie in den vorherigen Tutorials als neues Objekt mit der Bezeichnung **df** für Data Frame eingelesen und für die weitere Verwendung in Python angelegt:

```
# Datensatz einlesen
df = pd.read_csv('datasets/data_five.csv')
```

DATENAUFBEREITUNG

Im ersten Schritt, der sogenannten Datenaufbereitung, werden über die Befehle **array()** und **iloc[]** die **Zielvariable als Output** y und die **Features als Input** x angelegt. Weil die Merkmalsausprägungen in der **Zielvariable alphanumerisch** ausgeprägt sind, werden

diese über den **LabelEncoder** und den Befehl **fit_transform()** zusätzlich vor der Anwendung der Machine Learning Algorithmen in ein **numerisches Format** überführt.

```
# Datenaufbereitung für Supervised Machine Learning
y = np.array(df.Y)
x = df.iloc[:,[0,3]]
encoder = LabelEncoder()
y = encoder.fit_transform(y)
```

TRAININGS- UND TESTDATENSATZ

Beim Supervised Machine Learning werden Datensätze in Trainings- und Testdatensätze aufgeteilt, um überprüfen zu können, wie gut die zugrundeliegenden **Algorithmen Vorhersagen für ihnen unbekannte Daten** treffen können. Im vorliegenden Tutorial beträgt der Trainingsdatensatz 80 Prozent der Größe des ursprünglichen Datensatzes. Auf dieser Grundlage wird zunächst ein Algorithmus trainiert. Dieser lernt im Training eine **Funktion f(x)**, die es ermöglichen soll, jedem **Input x** einen **Output y** zuzuordnen. Dadurch wären dem Algorithmus entsprechende **Vorhersagen** möglich, die es schließlich über den **Testdatensatz zu überprüfen** gilt. In diesem Tutorial beträgt die Größe des Testdatensatzes 20 Prozent des ursprünglichen Datensatzes. Bei komplexeren Datensätzen empfielt sich der zusätzliche Rückgriff auf einen sogenannten **Validierungsdatensatzes** zu Lasten der Größe des Trainingsdatensatzes, um die erlernte Funktion besser überprüfen zu können. Der dazu erforderliche Befehl lautet **train_test_split**:

```
# Trainingsdatensatz (80%) und Testdatensatz (20%) einteilen
res = train_test_split(x, y,
    train_size=0.8,
    test_size=0.2,
    random_state=12)
train_data, test_data, train_labels, test_labels = res
```

K–NEAREST–NEIGHBOR ALGORITHMUS

Der K-Nearest-Neighbor Algorithmus ist ein einfaches Klassifikationsverfahren, bei dem eine **Klassenzuordnung** unter Berücksichtigung seiner **k nächsten Nachbarn** vorgenommen wird. Der Teil des Lernens besteht aus simplem **Abspeichern der Trainingsbeispiele**, was auch als **lazy learning** bezeichnet wird. Eine **Daten-normalisierung** kann die Genauigkeit dieses Algorithmus erhöhen:

```
# K-Nearest-Neighbor
knn = KNeighborsClassifier()
knn.fit(train_data, train_labels)
print("Predictions from the classifier:")
knn_predicted_data = knn.predict(test_data)
print(knn_predicted_data)
print("Target values:")
print(test_labels)
```

```
# K-Nearest-Neighbor Accuracy
print(accuracy_score(knn_predicted_data, test_labels))
```

LINEAR–DISCRIMINANT–ANALYSIS ALGORITHMUS

Die Linear-Discriminant-Analysis kommt als Algorithmus in Betracht, wenn die Objekte jeweils genau einer von **mehreren gleichartigen Klassen** angehören. Dafür muss im Sinne des Supervised Machine Learnings bekannt sein, welcher Klasse jedes einzelne Objekt angehört. Dies ist entsprechend im **Trainingsdatensatz** hinterlegt. An jedem Objekt werden daraufhin die Merkmalsausprägungen beobachtet. Aus diesen Informationen sollen **lineare Grenzen zwischen den Klassen** gefunden werden, um später Objekte, deren Klassenzugehörigkeit unbekannt ist, einer der Klassen zuordnen zu können. Die lineare Diskriminanzanalyse ist demnach ein klassisches **Klassifikations-verfahren**:

```
# Linear-Discriminant-Analysis
lda = LinearDiscriminantAnalysis()
lda.fit(train_data, train_labels)
print("Predictions from the classifier:")
lda_predicted_data = lda.predict(test_data)
print(lda_predicted_data)
print("Target values:")
print(test_labels)
```

```
# Linear-Discriminant-Analysis Accuracy
print(accuracy_score(lda_predicted_data, test_labels))
```

RANDOM FOREST ALGORITHMUS

Der Random Forest Algorithmus ist ein **Klassifikations- und Regressionsverfahren**, das aus mehreren unkorrelierten Entscheidungsbäumen besteht. Alle **Entscheidungsbäume** sind unter einer bestimmten Art von **Randomisierung während des Lernprozesses** gewachsen. Für eine Klassifikation darf jeder Baum in diesem Wald eine Entscheidung treffen und die Klasse mit den meisten Stimmen entscheidet die endgültige Klassifikation. Random Forests können auch zur Regression eingesetzt werden.

```
# Random Forest
rf = RandomForestClassifier()
rf.fit(train_data, train_labels)
print("Predictions from the classifier:")
rf_predicted_data = rf.predict(test_data)
print(rf_predicted_data)
print("Target values:")
print(test_labels)
```

```
# Random Forest Accuracy
print(accuracy_score(rf_predicted_data, test_labels))
```

K–MEANS ALGORITHMUS

Beim K-Means Algorithmus wird aus einer **Menge von ähnlichen Objekten** eine **vorher bekannte Anzahl von k Gruppen** gebildet. Der Algorithmus ist eine der am häufigsten verwendeten Techniken zur Gruppierung von Objekten, da er schnell die **Zentren der Cluster** findet. Ausgehend von diesen Zentren werden die **naheliegenden Datenpunkte**

so lange einer Gruppe zugewiesen, bis der **durchschnittliche Abstand** zu große Sprünge aufweist. Daher bevorzugt der Algorithmus Gruppen mit geringer Varianz und ähnlicher Größe, verlangsamt sich aber deutlich bei zu großen Datensätzen. Eine passende Anzahl von k Gruppen kann vorab über die **Within-Cluster-Sum-of-Squares** Methode ermittelt werden:

```
# K-Means
kmeans = KMeans(3)
kmeans.fit(train_data, train_labels)
print("Predictions from the classifier:")
kmeans_predicted_data = kmeans.predict(test_data)
print(kmeans_predicted_data)
print("Target values:")
print(test_labels)
```

```
# K-Means Accuracy
print(accuracy_score(kmeans_predicted_data, test_labels))
```

SUPPORT VECTOR MACHINE ALGORITHMUS

Der Support Vector Machine Algorithmus unterteilt eine Menge von Objekten so in Klassen, dass um die **Klassengrenzen** herum ein möglichst breiter Bereich frei von Objekten bleibt. Jedes Objekt wird dabei durch einen **Vektor in einem Vektorraum** repräsentiert. Aufgabe der Support Vector Machine ist es, in diesen Raum eine **Hyperebene** einzupassen, die als **Trennfläche** fungiert und die Trainingsobjekte in zwei Klassen teilt. Der **Abstand** derjenigen Vektoren, die der Hyperebene am nächsten liegen, wird dabei **maximiert**. Dieser breite, leere Rand soll später dafür sorgen, dass auch Objekte, die nicht genau den Trainingsobjekten entsprechen, möglichst zuverlässig klassifiziert werden. Zum Abschluss dieses Tutorials können Sie die **Accuracy selbst aufrufen**:

```
# Support Vector Machine
svm = SVC(kernel='linear')                    # Alternativen: 'poly', 'rbf', 'sigmoid', etc.
svm.fit(train_data, train_labels)
print("Predictions from the classifier:")
svm_predicted_data = svm.predict(test_data)
print(svm_predicted_data)
print("Target values:")
print(test_labels)
```

```
# Support Vector Machine Accuracy

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

6. TUTORIAL:

STANDARDISIERUNG

In diesem Tutorial geht es um die sogenannte **Standardisierung der Merkmals-ausprägungen** von Variablen. Ein gängiges Standardisierungsverfahren ist die z-Transformation. Die z-Transformation überführt Werte, die mit **unterschiedlichen Messinstrumenten** erhoben wurden, in eine neue **gemeinsame Einheit**: in Standard-abweichungs-Einheiten. Unabhängig von den Ursprungseinheiten können zwei (oder mehr) Werte nun unmittelbar miteinander verglichen werden.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **pandas**, sowie über den Befehl **import** zu aktivieren. Innerhalb von **sklearn** steht darüber hinaus **Standard-Scaler** für die z-Transformation zur Verfügung:

```
# Vorbereitungen
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()
```

Erneut wird über das Paket **pandas** mit dem Befehl **read_csv()** der Datensatz für dieses Tutorial eingelesen. Hier wird der Datensatz wie in den vorherigen Tutorials als neues Objekt mit der Bezeichnung **df** für Data Frame eingelesen und für die weitere Verwendung in Python angelegt:

```
# Datensatz einlesen
df = pd.read_csv('datasets/data_six.csv')
```

UNIVARIATE STATISTIK

Der Befehl **describe()** aus der univariaten Statistik stellt die unterschiedlichen **Minimum- und Maximumwerte** sowie die unterschiedlichen arithmetischen Mittel der Variablen tabellarisch dar. Darüber hinaus wird die Standardabweichung **std** ausgewiesen, welche ein Maß für die Streuung der Mekrmalsausprägungen um das arithmetische Mittel darstellt:

```
# Univariate Statistik
df.describe()
```

Z – TRANSFORMATION

Zunächst wird für die z-Transformation die zu transformierende Variable über den **iloc[]** Befehl adressiert. Über den Befehl **scale.fit_transform()** erfolgt anschließend die eigentliche z-Transformation. Das Ergebnis der z-Transformation kann über die Befehle **mean** und **std** aufgerufen werden. Die entsprechenden Werte sollten **0 für das arithmetische Mittel** und **1 für die Standardabweichung** betragen:

```
# z-Transformation der ersten unabhängigen Variable
x = df.iloc[:,[0]]
scaled_x = scale.fit_transform(x)
```

```
# Mittelwert der standardisierten Variable
mean_scaled_x = np.mean(scaled_x)
round(mean_scaled_x)
```

```
# Standardabweichung der standardisierten Variable
sd_scaled_x = np.std(scaled_x)
round(sd_scaled_x)
```

Wiederholen Sie diese Schritte für die **zweite unabhängige Variable** und die **abhängige Variable** des Datensatzes:

```
# Zweiten unabhängige Variable

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

```
# Abhängige Variable

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

Auch hier sollte im Anschluss an die z-Transformation **0 für das arithmetische Mittel** und **1 für die Standardabweichung** herauskommen.

7. TUTORIAL:

CLUSTERANALYSE

In diesem Tutorial werden verschiedene **Algorithmen der Clusteranalyse** vorgestellt, die entsprechend der Beschaffenheit des Datensatzes unterschiedlich gute Resultate erzielen. Die Algorithmen werden dabei zur Entdeckung von Ähnlichkeitsstrukturen in Datensätzen und -beständen herangezogen. Die so gefundenen Gruppen von **ähnlichen Objekten** werden als **Cluster** bezeichnet, deren Gruppenzuordnung entsprechend als Clustering.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **matplotlib** sowie **sklearn** über den Befehl **import** zu aktivieren. Innerhalb von **sklearn** stehen verschiedene Algorithmen der Clusteranalyse zur Verfügung, darunter **Birch**, **DBSCAN**, **MeanShift**, **GaussianMicture**, **SpectralClustering** und **AgglomerativeClustering**. Diesmal wird nicht auf einen externen Datensatz zurückgegriffen, sondern über den Befehl **make_blops** ein anschaulicher Übungsdatensatz simuliert:

```
# Vorbereitungen
from numpy import unique
from numpy import where
from matplotlib import pyplot
from sklearn.cluster import Birch
from sklearn.cluster import DBSCAN
from sklearn.cluster import MeanShift
from sklearn.mixture import GaussianMixture
from sklearn.cluster import SpectralClustering
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
```

DATENSATZ SIMULIEREN

Mit dem Befehl **make_blops** lässt sich ein **Übungsdatensatz** mit entsprechender Beschaffenheit für die Clusteranalyse simulieren. Der Datensatz soll insgesamt **500 Fälle** aufweisen, sich aus **vier Clustern** zusammensetzen, jeweils mit einer **Standardabweichung von 1.75**. Das Zufallsprinzip der simulierten Cluster wird dabei über **random_state** initiiert, so dass sich bei entsprechender Variation jeweils unterschiedliche Übungsdatensätze simulieren lassen:

```
# Datensatz für Plot simulieren
X, y = make_blobs(n_samples=500, centers=4, cluster_std=1.75, random_state=7)
```

Und so sieht der Übungsdatensatz mit dem Befehl **scatter()** aus:

```
# Simulierten Datensatz plotten
for class_value in range(4):
    row_ix = where(y == class_value)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

AGGLOMERATIVES CLUSTERING ALGORITHMUS

Agglomerative Verfahren der Clusteranalyse sind sogenannte **bottom-up-Verfahren**. Dabei bildet zunächst **jedes Objekt ein Cluster** und schrittweise werden die bereits gebildeten Cluster zu immer größeren Clustern **aufgrund der Nähe der Datenpunkte zusammengefasst**, bis alle nahestehenden Objekte zu einem Cluster gehören:

```
# Agglomeratives Clustering Algorithmus
model = AgglomerativeClustering(n_clusters=4)
yhat = model.fit_predict(X)
clusters = unique(yhat)
```

```
# Clusterlösung visualisieren
for cluster in clusters:
    row_ix = where(yhat == cluster)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

BIRCH ALGORITHMUS

Der **Balanced Iterative Reducing and Clustering** Algorithmus eignet sich insbesondere für **große Datenmengen** (Anmerkungen: Andere Algorithmen können den Arbeitsspeicher auslasten). Dabei betrachtet der Birch Algorithmus die **Datenpunkte zunächst lokal**, so dass nicht alle Objekte oder Cluster parallel analysiert werden. Wie die anderen Verfahren nutzt auch der Birch Algorithmus die Tatsache aus, dass die **Objekte nicht gleichmäßig verteilt** sind, bzw. dass die Objekte für das Clustering nicht alle gleich wichtig sind:

```
# Birch Algorithmus
model = Birch(threshold=0.01, n_clusters=4)
model.fit(X)
yhat = model.predict(X)
clusters = unique(yhat)
```

```
# Clusterlösung visualisieren
for cluster in clusters:
    row_ix = where(yhat == cluster)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

DBSCAN ALGORITHMUS

Unabhängig von einer Dichtefunktion versucht der **Density-Based Spatial Clustering of Applications with Noise** Algorithmus die Anzahl an Clustern und eine Zuordnung der Datenpunkte zu diesen zu identifizieren. Auch hier wird davon ausgegangen, dass eine **zunehmende Dichte der Datenpunkte** ein Cluster repräsentieren kann, wobei die **abnehmende Dichte** durch weiter entfernte Datenpunkte **nicht über die Standardabweichungen** entlang einer Normalverteilung berücksichtigt, sondern als Rauschen - hier: Noise - klassifiziert wird:

```
# DBSCAN Agorithmus
model = DBSCAN(eps=1.00, min_samples=4)
yhat = model.fit_predict(X)
clusters = unique(yhat)
```

```
# Clusterlösung visualisieren
for cluster in clusters:
    row_ix = where(yhat == cluster)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

MEAN SHIFT ALGORITHMUS

Der Mean Shift Algorithmus versucht eigenständig und somit **ohne vorherige Spezifikation** die Anzahl an Clustern über die **Konzentrationen innerhalb der Datenpunkte** zu identifizieren. Dazu wechselt der mögliche Schwerpunkt eines Clusters **iterativ zwischen verschiedenen Datenpunkten**, bis sich eine Konzentration einstellt:

```
# Mean Shift Algorithmus
model = MeanShift()
yhat = model.fit_predict(X)
clusters = unique(yhat)
```

```
# Clusterlösung visualisieren
for cluster in clusters:
    row_ix = where(yhat == cluster)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

SPECTRAL CLUSTERING ALGORITHMUS

Bei diesem Verfahren werden die zu clusternden **Objekte als Knoten eines Graphen** betrachtet. Die **Distanzen** oder Unähnlichkeiten zwischen den Objekten werden durch die gewichteten **Kanten zwischen den Knoten** des Graphen repräsentiert. Der Algorithmus versucht den Graphen zu reduzieren. Übergeordnetes Ziel des Algorithmus ist es, alle **Kanten mit zu großen Gewichten über mehrere Schritte aus aus dem Graphen zu entfernen**, bis die zuvor spezifizierte Anzahl an Cluster übrigbleibt:

```
# Spectral Clustering Algorithmus
model = SpectralClustering(n_clusters=4)
yhat = model.fit_predict(X)
clusters = unique(yhat)
```

```
# Clusterlösung visualisieren
for cluster in clusters:
    row_ix = where(yhat == cluster)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

GAUSSIAN MIXTURE ALGORITHMUS

Der Gaussian Mixture Algorithmus greift alternativ auf die **gewichtete Summe mehrerer Gauß-Funktionen** zurück. Dabei liegt die Annahme zugrunde, dass die Datenpunkte innerhalb der Cluster jeweils auf einer **Normalverteilung** basieren. Bei Normalverteilungen handelt es sich formal gesehen um Dichtefunktionen, weshalb oftmals auch von einem Dichtefunktion-basiertem Verfahren gesprochen wird. Dabei entspricht der **Schwerpunkt eines Clusters** dem jeweiligen **Erwartungswert** und ausgehend von der dazugehörigen Standardabweichung lässt sich die Grenze zu anderen Clustern approximieren:

```
# Gaussian Mixture Algorithmus
model = GaussianMixture(n_components=4)
model.fit(X)
yhat = model.predict(X)
```

```
# Clusterlösung visualisieren
for cluster in clusters:
    row_ix = where(yhat == cluster)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

Recherchieren Sie in der **sklearn** Datenbank nach **weiteren Algorithmen** der Clusteranalyse und bringen Sie diesen **nach dem oben genannten Schema zur Anwendung**:

```
# Ausgewählter Algorithmus der Clusteranalyse

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

8. TUTORIAL:

WITHIN CLUSTER SUM OF SQUARES

In Ergänzung zu dem Tutorial über die **Algorithmen der Clusteranalyse** wird in diesem Tutorial das Within Cluster Sum of Squares Verfahren vorgestellt, welches dann zur Anwendung kommt, wenn die **Anzahl an Clustern nicht bekannt** und somit zunächst bestimmt werden muss. Diesem Verfahren liegt die Annahme zugrunde, dass **innerhalb eines Clusters die Quadratsumme niedriger** ausfallen wird als unter Berücksichtigung eines weiteren Datenpunktes aus einem anderen - und weiter entfernten - Cluster.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **matplotlib** sowie **sklearn** über den Befehl **import** zu aktivieren. Innerhalb von **sklearn** wird über **make_blops** wieder ein anschaulicher Übungsdatensatz simuliert, der mittels **K-Means Algorithmus** analysiert werden soll:

```
# Vorbereitungen
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
```

DATENSATZ SIMULIEREN

Mit dem Befehl **make_blops** lässt sich ein **Übungsdatensatz** mit entsprechender Beschaffenheit für die Clusteranalyse simulieren. Der Datensatz soll insgesamt **500 Fälle** aufweisen, sich aus **vier Clustern** zusammensetzen, jeweils mit einer **Standardabweichung von 1.75**. Das Zufallsprinzip der simulierten Cluster wird dabei über **random_state** initiiert, so dass sich bei entsprechender Variation jeweils unterschiedliche Übungsdatensätze simulieren lassen:

```
# Datensatz für Plot simulieren
X, y = make_blobs(n_samples=500, centers=4, cluster_std=1.75, random_state=7)
```

Und so sieht der Übungsdatensatz mit dem Befehl **scatter()** aus:

```
# Cluster visualisieren
plt.scatter(X[:,0], X[:,1])
```

WITHIN CLUSTER SUM OF SQUARES

Über den Befehl **KMeans** lässt sich die zu bestimmende Anzahl an Clustern zunächst über **n_cluster=i** definieren. Die weiteren Parameter legen dabei fest, mit welchen Voraussetzungen der Algorithmus beginnen soll und können in der entsprechenden Dokumentation von **sklearn** nachgeschlagen werden. Wie im vorherigen Tutorial wird der Algorithmus über den Befehl **fit()** ausgeführt, so dass sich daraufhin über den Befehl **apend()** die Anzahl an Clustern adressieren lässt. Das Ergebnis wird über den Befehl plot schließlich visualisiert. Bei der sogenannten **Ellenbogengrafik** kann die Anzahl an

Clustern am **Ellenbogenknick** abgelesen werden. An dieser Stelle ist die Within Cluster Sum of Squares minimiert (Y-Achse) und die dazugehörige Anzahl an Clustern ist entsprechend auf der X-Achse abgetragen:

```
# Within Cluster Sum of Squares
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=200, n_init=10, random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```

Schließlich kann der **K-Means Algorithmus** vollständig zur Anwendung gebracht werden. Dafür ist diesmal im Befehl **KMeans** die Anzahl an Clustern über **c_cluster=4** definieren. Der Algorithmus ist in der Lage, aus einer Menge ähnlicher Objekte mit einer vorher bekannten Anzahl von Gruppen die jeweiligen **Zentren der Cluster** zu ermitteln. Da es sich um ein sehr **effizientes Verfahren** handelt, das mit vielen verschiedenen Datentypen zurecht kommt, und der Speicherbedarf gering ist, eignet sich der K-Means Algorithmus für die Datenanalyse im Big Data Kontext:

```
# K-Means Cluster visualisieren
kmeans = KMeans(n_clusters=4, init='k-means++', max_iter=200, n_init=10, random_state=5)
pred_y = kmeans.fit_predict(X)
plt.scatter(X[:,0], X[:,1])
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=500, c='yellow')
plt.show()
# plt.savefig('k_means.jpg', dpi=600)
```

Über den Befehl **plt.savefig()** ließe sich der Output schließlich speichern. Zum Abschluss dieses Tutorials gibt es nur eine kleine Aufgabe: Ändern Sie **Größe und Farbe der dargestellten Zentren der Cluster** im Plot.

```
# K-Means Cluster grafisch anpassen

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

9. TUTORIAL:

NEURONALE NETZE / DEEP LEARNING

Deep Learning unterscheidet sich vom Machine Learning insbesondere darin, wie Algorithmen lernen. Deep Learning **automatisiert einen Großteil der Merkmals-extraktion**, wodurch ein Teil der erforderlichen manuellen Eingriffe über sogenannte **neuronale Netze** entfällt und die Verwendung größerer Datensätze ermöglicht wird. Deep Learning ist somit ein **Teilgebiet von Machine Learning** und neuronale Netze sind folglich ein Teilgebiet von Deep Learning. In diesem Tutorial wird ein neuronales Netz aufgesetzt und mit moderaten Variationen zur Anwendung gebracht.

VORBEREITUNGEN

Als Vorbereitungen für das Tutorial sind zunächst die Pakete **numpy**, **matplotlib** sowie **keras** über den Befehl **import** zu aktivieren. Das Paket **keras** erweitert Python um die grundlegenden Funktionen des Deep Learnings. Innerhalb des Pakets werden zusätzlich **Sequential** und **Dense** importiert:

```
# Vorbereitungen
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
```

DATENSATZ GENERIEREN

Zur Veranschaulichung der Funktionsweise wird ein **einfacher Datensatz** mit unabhängigen variablen als **training_data** und einer abhängigen Variable als **target_date** angelegt. Dafür steht der Befehl **array** zur Verfügung. Über den Parameter **float32** können die Zahlen im Datensatz in **32 Bit** und als **Gleitkommazahl** gespeichert werden, so dass auch gebrochene Zahlen (bspw. 3.1415 als Kreiszahl Pi) gespeichert und berechnet werden können.

```
# Datensatz generieren
training_data = np.array([[0,0,0],[0,1,1],[1,0,1],[1,1,0]], "float32")
target_data = np.array([[0],[1],[1],[0]], "float32")
```

NEURONALES NETZ SPEZIFIZIEREN

Das Paket **keras** teilt die Implementierung von neuronalen Netzen in mehrere Objekte auf. Dabei werden einzelne **Layer** als Objektinstanzen zu einem Modell hinzugefügt, wobei zwei Modelle zur Auswahl stehen: **sequentielle Modelle**, bei denen alle Layer in einer Sequenz abgebildet werden und **funktionale Modelle**, bei denen die Layer direkt miteinander verbunden werden. Hier wird über den Befehl **Sequential()** folglich ein sequentielles Modell angelegt. Über den Befehl **Dense** kann spezifiziert werden, wie viele **Neuronen** des neuronalen Netzes aufweisen soll. Als **Aktivierungsfunktion** im ersten Layer wird auf die **ReLu-Funktion** zurückgegriffen, um im neuronalen Netz andere als lineare Zusammenhänge berücksichtigen zu können. Der Befehl **compile()** ermöglicht schließlich den Rückgriff auf verschiedene **Metriken**, um die Performance des neuronalen

Netzes zu **validieren**. Schließlich wird das neuronale Netz über den Befehl **fit()** trainiert, wobei mit dem Parameter **epochs** die **Anzahl der Durchläufe** durch die Trainingsdaten festgelegt wird. Zunächst werden 15 Durchläufe festgelegt. Bei sehr großen Datensätzen kann zusätzlich über den Parameter **batch_size** festgelegt werden, **wie vielen Datensätze gleichzeitig trainiert** werden sollen. Abschließend verbindet der Befehl **evaluate** das neuronale Netz mit den zuvor angelegten Daten:

```
# Neuronales Netz spezifizieren
model = Sequential()
model.add(Dense(4, activation='relu'))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=15)              # 15 Durchläufe
scores = model.evaluate(training_data, target_data)
```

Die **Performance** des neuronalen Netzes kann anschließend **in Prozent** ausgewiesen und dieses somit validiert werden:

```
# Performance des neuronalen Netzes
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
print(model.predict(training_data).round())
```

NEURONALES NETZ MODIFIZIEREN

Als erste Modifikation wird die **Anzahl an Durchläufen** erhöht, um dem neuronalen Netz eine bessere Möglichkeit zum Lernen einzuräumen. Hierzu wird der Parameter **epochs** entsprechend modifiziert:

```
# Neuronales Netz modifizieren
model = Sequential()
model.add(Dense(4, activation='relu'))
model.add(Dense(8, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['binary_accuracy'])
model.fit(training_data, target_data, epochs=150)             # 150 Durchläufe
scores = model.evaluate(training_data, target_data)
```

Erneut kann die **Performance** des neuronalen Netzes anschließend **in Prozent** ausgewiesen und dieses somit validiert werden:

```
# Performance des neuronalen Netzes
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
print(model.predict(training_data).round())
```

VISUALISIERUNG DES LERNVERHALTENS

Das Lernverhalten des neuronalen Netzes kann über die **binary_accuracy** - hier blau dargestellt - und die sogenannte **loss-Funktion** - hier orange dargestellt - eingesehen werden. Diese bilden zum einen den Zuwachs **korrekter Klassifikationen** mit steigender Anzahl an Durchläufen ab. Zum anderen wird die Diskrepanz zwischen den beobachteten Daten und den von dem über die Durchläufe angepassten neuronalen Netz vorhergesagten Daten dargestellt. Bei einer **linearen Regression** würde sich bspw. das **Residuum** bei einer zielführenden Anpassung des Intercept-Wertes und der Regressionsgewichte entsprechend **minimieren**. Beim **Deep Learning** übernimmt die **loss-Funktion** diese Aufgabe je nach zugrundeliegendem Funktionsverlauf:

```
# Visualisierung des Lernverhaltens
plt.plot(history.history['binary_accuracy'])
plt.plot(history.history['loss'])
plt.show()
```

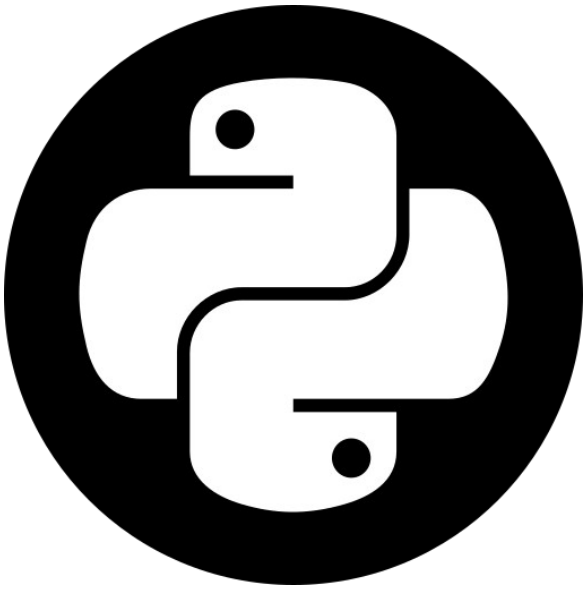
Erproben Sie abschließend **alternative Aktivierungsfunktionen** und vergleichen Sie die Performance mit dem hier vorgestellten neuronalen Netz:

```
# Neuronales Netz mit tanh-Aktivierungsfunktion

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

```
# Neuronales Netz mit sigmoid-Aktivierungsfunktion

###################################
### SIEHE AUFGABENSTELLUNG ###
###################################
```

DATA SCIENCE MIT PYTHON

B E I S P I E L E



www.**STATISTICAL-THINKING**.de

1. BEISPIEL:

HANDSCHRIFTERKENNUNG

```
from matplotlib import pyplot
import tensorflow as tf
import numpy as np

mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

for i in range(9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(x_train[i], cmap=pyplot.get_cmap('gray'))
    pyplot.show()

model = tf.keras.models.Sequential([
    tf.keras.Input(shape=(28, 28)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)

example = x_train[1701]
pyplot.imshow(example.reshape(28, 28), cmap="gray")

output_predict = model.predict(example.reshape(1, 28, 28, 1))
print("Predicted figure:")
np.argmax(output_predict)
```

2. BEISPIEL:

GESICHTSERKENNUNG

```
import cv2
import matplotlib.pyplot as plt
```

```
image_path = 'images/facial_recognition_image.jpg'
image = cv2.imread(image_path)
```

```
face_classifier = cv2.CascadeClassifier('images/haarcascade_frontalface_default.xml')
face = face_classifier.detectMultiScale(image, scaleFactor=1.1,minNeighbors=5,minSize=(40, 40))
for (x, y, w, h) in face:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 4)
```

```
img_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
plt.figure(figsize=(20,10))
plt.imshow(img_rgb)
plt.axis('off')
```

Bild: https://www.statistical-thinking.de/python_workshops/facial_recognition_image.jpg

3. BEISPIEL:

NATURAL LANGUAGE PROCESSING

```
import numpy as np
```

```
lexicon = {}

def update_lexicon(current : str, next_word : str) -> None:
    if current not in lexicon:
        lexicon.update({current: {next_word: 1} })
        return

    options = lexicon[current]
    if next_word not in options:
        options.update({next_word : 1})
    else:
        options.update({next_word : options[next_word] + 1})

    lexicon[current] = options

with open('datasets/text.txt') as dataset:
    for line in dataset:
        words = line.strip().split(' ')
        for i in range(len(words) - 1):
            update_lexicon(words[i], words[i+1])

    for word, transition in lexicon.items():
        transition = dict((key, value / sum(transition.values())) for key, value in transition.items())
        lexicon[word] = transition
```

```
line = input('> ')
word = line.strip().split(' ')[-1]
if word not in lexicon:
    print('Sorry...')
else:
    options = lexicon[word]
    predicted = np.random.choice(list(options))
    print(predicted)
```

```
list(options.keys())
```

Text: https://www.statistical-thinking.de/python_workshops/text.txt