KAPITEL 3

Datenimport und -vorbereitung

Daten stehen in den seltensten Fällen in genau dem richtigen Format und in perfekter Qualität zur Verfügung. Oft müssen sie sogar erst beschafft werden. In jedem Fall müssen sie aber in die von uns hier eingeführten Systeme importiert, d.h. eingelesen werden.

Diese Vorbereitung kann einen Großteil deiner Arbeit darstellen und ist selten der spannendste, bildet aber die Grundlage für die in den folgenden Kapiteln beschriebene weitere Arbeit. Ohne bereinigte Daten gibt es keine Analyse. Je mehr Arbeit du in die Vorbereitung steckst, desto besser werden auch die Ergebnisse.

In diesem Kapitel führen wir dich neben der Datenvorverarbeitung auch in die Grundlagen der Programmiersprache Python und ihrer Bibliotheken ein, die wir in den nachfolgenden Kapiteln für alle Beispiele nutzen werden. Dazu gibt es Kästen, die du überspringen kannst, wenn du dich gar nicht für die Programmierung interessierst oder die jeweiligen Programmierthemen bereits beherrschst.

Der komplette und ausführbare Code für dieses Kapitel ist wieder unter der URL *https://github.com/DJCordhose/buch-machine-learning-notebooks* als Notebook *kap3.ipynb* (*https://bit.ly/ml-kug*) erreichbar.

Datenimport

Für dieses Kapitel haben wir einen Beispieldatensatz ausgesucht, der relativ schlecht ist. Das ist zweckmäßig, da es hier ja um die Vorverarbeitung schlechter Datensätze geht. Wir verraten dir ein Geheimnis: Wir haben den Datensatz sogar mit Absicht noch schlechter

gemacht, als er ohnehin schon war. Das ist sinnvoll, da du ja lernen sollst, wie man mit schlechten Daten umgehen muss.

Am Anfang steht immer der Import der Daten in ein System, das überhaupt erst eine Vorverarbeitung erlaubt. Wir nutzen die Programmiersprache Python mit den dafür passenden Bibliotheken. Für diesen speziellen Fall des Imports und der Vorverarbeitung setzen wir die Standardbibliotheken Pandas und NumPy ein.

Python

Python ist eine Programmiersprache, die an keinen besonderen Zweck gebunden ist. Sie ist jedoch die Programmiersprache schlechthin für wissenschaftliche Datenverarbeitung und für Aufgaben im Bereich Data Science und Machine Learning. Wir werden Python in der Version 3 für alle Programmierbeispiele in diesem Buch nutzen.

Python-Bibliotheken

Neben der eigentlichen Sprache Python braucht man eine ganze Reihe von bereits von anderen Menschen geschriebenen Bibliotheken. Diese Bibliotheken bieten Funktionalitäten, die wir nur sehr schwer oder mit hohem Aufwand selbst programmieren könnten. Konkret werden wir die Bibliotheken Pandas, NumPy, matplotlib, Seaborn und Scikit-Learn nutzen. Diese wirst du nach und nach kennenlernen.

Unsere Datensätze liegen als Textdatei vor, in der in jeder Zeile ein Datensatz steht. Die einzelnen Felder der Datensätze sind durch Kommata getrennt. Das könnte für die ersten Zeilen so aussehen:

```
5.1,3.5,1.4,2 mm,Iris-setosa
4.9,3.0,1.4,2 mm,Iris-setosa
4.7,3.2,1.3,2 mm,Iris-setosa
```

Und so sieht unsere erste Zeile Python-Code aus, die eine solche Datei einliest:

```
import pandas as pd
df = pd.read_csv('iris_dirty.csv')
```

In der ersten Zeile importierst du die Pandas-Bibliothek, danach kannst du sie unter dem Namen `pd` nutzen. Das tust du in der zweiten Zeile mit `pd.read_csv`. Diese Methode braucht einen Dateipfad und geht davon aus, dass sich dahinter eine Datei im CSV-Format befindet. CSV steht für *Comma-Separated Values*, also wie oben beschrieben für Werte, die durch Kommata getrennt sind.

Wir sind aber noch nicht ganz fertig, weil der Import in der ersten Zeile der Datei die Namen der Felder als Beschreibung erwartet. Das ist in unserer Datei nicht so, daher geben wir die Felder explizit beim Import an.

```
import pandas as pd

df = pd.read_csv('iris_dirty.csv',
                 header=None, # A
                 names=['sepal length', 'sepal width', # B
                        'petal length', 'petal width', 'class']
)
```

In Zeile A geben wir an, dass die Datei keinen Header hat, in dem die Namen der Felder stehen würden. Daher geben wir diese in Zeile B ersatzweise als Liste direkt an.

Python-Grundlagen

Bibliotheken liefern uns nützliche Funktionen, die wir über diese aufrufen können. In unserem Fall haben wir `read_csv` aus der Pandas-Bibliothek benutzt und den Aufruf über `pd` vorgenommen, da wir Pandas unter dem Namen `pd` importiert haben:

```
import pandas as pd
df = pd.read_csv('iris_dirty.csv')
```

In Python können Werte in Variablen abgelegt werden. Dazu schreibt man einfach den frei wählbaren Namen einer Variablen hin und weist ihm mit einem Gleichheitszeichen (=) einen Wert zu. Dieser Wert kann auch das Ergebnis eines Funktionsaufrufs sein. In unserem Beispiel legen wir die Variable `df` an, die die importierten Datensätze enthält:

→

```
df = pd.read_csv('iris_dirty.csv')
print(df)
```

Auf solche Variablen kann über ihren Namen wieder zugegriffen und der Inhalt kann herausgeholt werden. Oben haben wir das getan, um den Inhalt mit der `print`-Funktion auszugeben.

Funktionen (z.B. aus Bibliotheken) brauchen häufig Parameter als zusätzliche Angaben. Diese werden zum Teil über ihre Position angegeben, zum Teil aber auch über ihren Namen. Unser Aufruf der Importfunktion enthält beide Varianten:

```
df = pd.read_csv('iris_dirty.csv', header=None)
```

Der Parameter `header` wird über seinen Namen angegeben, während der Dateiname an erster Stelle steht und dadurch seine Bedeutung bekommt. Parameter werden in Klammern eingeschlossen und mit Kommata getrennt.

Strings (Zeichenketten) werden in Python entweder von einfachen (') oder von doppelten (") Anführungszeichen umschlossen. Das ist notwendig, damit Python weiß, wo die Zeichenkette anfängt und wo sie aufhört. Dies nutzen wir in unserem Dateinamen und in dem Namen für die Felder:

```
'iris_dirty.csv'
```

Kommentare werden mit einer Raute (#) eingeleitet. Alles was in einer Zeile hinter der Raute steht, wird von Python ignoriert und dient nur deiner Information. Hier ist C nur ein Kommentar und wird von Python überlesen:

```
print(df) # C
```

Statt eines einzelnen Werts für eine Variable oder einen Parameter kannst du auch eine Liste von Werten angeben. Dazu umschließt du die Werte mit eckigen Klammern ([und]) und trennst sie wieder, wie bei Parametern, mithilfe von Kommata.

```
df = pd.read_csv('iris_dirty.csv',
                 names=['sepal length', 'sepal width',
                        'petal length', 'petal width', 'class'])
```

Eine umfassendere, aber immer noch kurze Einführung, die auch NumPy-Grundlagen abdeckt, findest du unter *http://cs231n.github.io/python-numpy-tutorial/*.

Das vorbereitete Projekt

In unserem Projekt nutzen wir wie in Kapitel 2, *Quick-Start*, den im Bereich Machine Learning sehr bekannten und weitverbreiteten Irisdatensatz (*http://archive.ics.uci.edu/ml/datasets/Iris*). Hier geht es um die Klassifizierung von Schwertlilien (Iris). Dazu liegen uns bestimmte Längen und Breiten von untersuchten Exemplaren samt einer Klassifizierung vor. Alle weiteren Eigenschaften des Datensatzes werden wir uns nach und nach erarbeiten. Dies entspricht häufig dem Vorgehen in der Praxis, da du oft nur wenig über die Datensätze weißt und dich ihnen durch den Einsatz von Werkzeugen nähern musst.

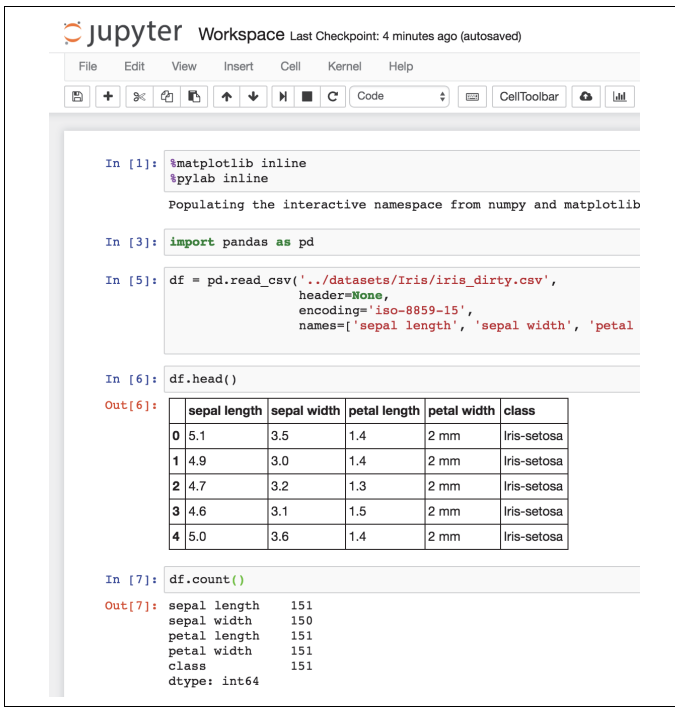

Abbildung 3-1: Dein Notebook im Browser

Wir empfehlen dir, alle Beispiele in den bereits in Kapitel 2, *Quick-Start*, eingeführten Jupyter-Notebooks nachzuvollziehen. Der Link ist wieder *https://github.com/DJCordhose/buch-machine-learning-notebooks* (*https://bit.ly/ml-kug*). Dieses Kapitel ist dort unter dem Namen *kap3.ipynb* erreichbar. Wenn du ein eigenes Notebook gestartet hast, solltest du in deinem Browser etwas Ähnliches wie das in Abbildung 3-1 sehen.

Preprocessing

Als Erstes solltest du dir bei einem Datensatz einen Überblick über seine Qualität verschaffen. Sobald du die Probleme des Datensatzes kennst, hast du schon die halbe Arbeit geleistet. Die andere Hälfte besteht darin, diese Probleme zu lösen und danach bereinigte Daten zur Verfügung zu haben.

Der folgenden Aufruf auf deinem Dataframe `df` gibt dir einen Überblick über die ersten Datensätze:

```
df.head()
```

Schließt du diese Zeile mit *Umschalt+Enter* ab, solltest du eine Tabelle wie die in dem Screenshot oben in deinem Browser sehen. Als Nächstes schauen wir, wie viele Werte es pro Spalte gibt:

```
df.count()
> sepal length    151
> sepal width     150
> petal length    151
> petal width     151
> class           151
```

Die Ausgabe zeigen wir wieder hinter dem > an. Hier sind es gleich mehrere Zeilen, und jede Zeile gibt die Anzahl der Werte für ein bestimmtes Feld an. Man hätte erwartet, dass es bei 151 Datensätzen auch 151 Werte pro Feld/Spalte gibt. Im Bereich `sepal width` sind es aber nur 150 Datensätze, es fehlt also ein Wert.

Fehlende Werte

Fehlende Werte kommen sehr häufig vor. Oft liegen Daten einfach nicht vor, manchmal ist es nicht einmal sinnvoll, an einer bestimmten Stelle überhaupt Daten zu haben.

Wenn ein einzelner Wert fehlt, kannst du mehrere Dinge tun:

1. den kompletten Datensatz löschen (wenn ein sehr wichtiger Wert fehlt, wie z. B. die Klassifizierung),
2. einen Mittelwert oder Ähnliches aus allen anderen Daten oder aus allen anderen Daten mit derselben Klassifizierung bilden (gut, wenn man wie wir nicht viele Datensätze hat),
3. den Wert selbst durch ein Machine-Learning-Verfahren bestimmen,
4. den fehlenden Wert mit einem festen anderen Wert, z. B. 0 oder –1, ersetzen.

Option 1 ist die einfachste Lösung, birgt aber große Risiken. Das Fehlen eines Werts könnte systematisch sein, das Löschen kompletter Zeilen verfälscht dann die Daten.

Zur Illustration ein Beispiel: Das Fehlen einer Festnetznummer bei einem Datensatz könnte systematisch sein, weil viele junge Menschen keinen Festnetzanschluss mehr besitzen. Werden alle diese Zeilen gelöscht, enthalten die Daten deutlich weniger junge Leute als vorher.

Wir entscheiden uns dafür, den fehlenden Wert mit einem Mittelwert aus allen Daten derselbe Klasse zu erzeugen. Dazu müssen wir erst einmal den unvollständigen Datensatz bestimmen, denn bisher wissen wir nur, dass ein Wert fehlt, aber nicht, wo.

Wir wissen, dass der fehlende Wert in der Spalte `'sepal width'` liegt, und diese Spalte kannst du in Pandas ganz einfach auslesen:

```
df['sepal width']
```

Leider haben wir dann gleich 151 Werte. Das macht es schwer, den fehlenden herauszusuchen (insbesondere weil du oft weit mehr als 150 Werte haben wirst). Zuerst fragen wir daher die Datensätze danach, ob die Spalte fehlt:

```
df['sepal width'].isnull()
```

Pandas: Dataframes und Series

Dataframe
: Der Dataframe ist die zentrale Datenstruktur von Pandas. Du kannst sie dir vorstellen wie eine Tabelle in Excel. Technisch besteht ein Dataframe aus einer Reihe von Series. Jede Series beschreibt dabei nicht eine Zeile der Tabelle, sondern eine Spalte. Über den [ ]-Operator kommst du an die einzelnen Series heran.

Series
: Eine Liste von Daten mit einem Index. Der Index kann angegeben werden, wird aber automatisch als fortlaufende Nummerierung gesetzt, wenn er fehlt. Über den [ ]-Operator kommst du an die einzelnen Daten heran.

  Im Notebook zu diesem Kapitel findest du einige veranschaulichende Beispiele zu *Series* und *Dataframes*.

Das gibt wieder 151 Ergebnisse, dieses Mal aber immerhin `True` und `False`. So etwas können wir nun als komplexere Bedingung in den eckigen Klammern angeben und den einzigen Datensatz herausfiltern, für den `True` herauskommt:

```
df[df['sepal width'].isnull()]
```

Unser fehlender Wert ist mit `NaN` (*Not a Number* = keine Zahl) gekennzeichnet:

```
index 'sepal length' 'sepal width' 'petal length' 'petal width'       class
82          5.8         NaN         3.9           12 mm   Iris-versicolor
```

Wir sehen auch, dass es der Wert `82` ist und die Iris zur Gattung `Iris-versicolor` gehört. Unser Plan war ja, den Mittelwert der `'sepal width'` aller Irisblüten aus derselben Gattung zu bestimmen. Erst einmal basteln wir uns wieder eine passende Bedingung, um alle Datensätze für diese Irisart zu bekommen:

```
iris_versicolor = df[df['class'] == 'Iris-versicolor']
```

Daraus extrahieren wir anschließend nur die `'sepal width'` und nutzen diese für den Mittelwert, den wir in der Variablen `mean SepalWidth` speichern:

```
meanSepalWidth = pd.Series.mean(iris_versicolor['sepal width'])
> 2.7800000000000007
```

Dazu setzen wir die Funktion `mean` ein, die wir über `pd.Series` erreichen. Der Mittelwert stellt sich dann ungefähr als 2,78 dar. Diesen müssen wir nun nur noch an die leere Stelle speichern und sind fertig mit dieser Korrektur:

```
df.loc[82,'sepal width'] = meanSepalWidth
```

Wir nutzen die Funktion `loc`, mit der wir ein bestimmtes Feld an einem bestimmten Index heraussuchen können. Der abschließende Test ergibt nun ein stimmiges Bild und 151 Werte für jedes der Felder:

```
df.count()
> sepal length 151
> sepal width 151
> petal length 151
> petal width 151
> class 151
```

Dubletten

Manches Mal hat man nicht zu wenige Daten, sondern zu viele. Oft schleichen sich durch falsche Dateneingaben oder Fehler in der Zwischenverarbeitung Dubletten ein, also doppelt vorhandene Datensätze. Pandas liefert eine sehr schöne Funktion, mit der du solche Datensätze findest ...

```
df[df.duplicated(keep=False)]
```

... die wir in einer Spielart nutzen, die uns alle doppelten Datensätze ausspuckt:

```
index sepal length  sepal width petal length petal width  class
9       4.9           3.1       1.5      1 mm       Iris-setosa
34      4.9           3.1       1.5      1 mm       Iris-setosa
37      4.9           3.1       1.5      1 mm       Iris-setosa
50      7.0           3.2       4.7     14 mm   Iris-versicolor
100     7.0           3.2       4.7     14 mm   Iris-versicolor
```

Gerade bei so ähnlichen Werten ist es Interpretationssache, ob ein Datensatz für ein und dieselbe Blume tatsächlich doppelt vorkommt oder ob es sich um eine echte Dublette handelt, die entfernt

werden sollte. Wir können uns das Wissen zunutze machen, dass für jede Klasse genau 50 Datensätze existieren sollen. Wir haben 151, also nur einen zu viel. Aber von welcher Klasse?

Dazu zählen wir die Datensätze pro Klasse:

```
df.groupby('class').count()
```

Wir bekommen heraus:

- `Iris-setosa`: 49
- `Iris-setsoa`: 1
- `Iris-versicolor`: 51
- `Iris-virginica`: 50

Offensichtlich haben wir hier – so ganz nebenbei – noch ein zweites Problem entdeckt, aber erst einmal sehen wir, dass wir von `Iris-versicolor` tatsächlich ein Exemplar zu viel haben. Das muss die Dublette sein, die wir einfach löschen. Dabei spielt es keine Rolle, ob wir den ersten oder zweiten Datensatz löschen, da sie ja identisch sind. Wir erzeugen einen neuen Datenframe, in dem der entsprechende Datensatz (`100`) fehlt:

```
df = df.drop(df.index[[100]])
```

Zeichendreher/Tippfehler

Wie bereits bemerkt, gibt es eine weitere Auffälligkeit: Es gibt einen Datensatz der Klasse `Iris-setsoa`. Dabei handelt es sich offensichtlich um einen Tippfehler, denn auch dieser Datensatz sollte die Klasse `Iris-setosa` haben. Das können wir schnell reparieren. Erst einmal finden wir heraus, dass der Datensatz den Index `49` hat ...

```
df[df['class']  == 'Iris-setsoa']
> 49
```

... und dann setzen wir die Klasse neu:

```
df.loc[49,'class'] = 'Iris-setosa'
```

Nun erhalten wir gleichmäßige Daten, wenn wir die Anzahl der Datensätze pro Klasse durchzählen:

```
df.groupby('class').count()
>             sepal length sepal width petal length petal width
> class
> Iris-setosa            50          50           50          50
> Iris-versicolor        50          50           50          50
> Iris-virginica         50          50           50          50
```

Uneinheitliche Einheiten

Schauen wir uns unsere Daten noch einmal mit

```
df.head()
```

an:

```
sepal length  sepal width  petal length petal width class
5.1           3.5          1.4          2 mm        Iris-setosa
4.9           3.0          1.4          2 mm        Iris-setosa
4.7           3.2          1.3          2 mm        Iris-setosa
4.6           3.1          1.5          2 mm        Iris-setosa
5.0           3.6          1.4          2 mm        Iris-setosa
```

Alle Werte sollten in Zentimetern mit einer Nachkommastelle vorliegen. Es fällt aber auf, dass die Variable `petal width` leider in Millimetern angegeben ist und auch nicht in einem einfach zu verarbeitenden Format, da sie den Text `mm` enthält. Das wollen wir vereinheitlichen und auch dieses Feld in cm umwandeln.

Eine solche Umwandlung bekommst du dadurch hin, dass du `' mm'` löschst, die Zeichenkette in eine Zahl wandelst und durch 10 teilst. Genau das tun wir hier erst einmal beispielhaft für die Zeichenkette `'2 mm'`:

```
pd.to_numeric('2 mm'.replace(' mm', '')) / 10
> 0.20000000000000001
```

Der besseren Übersicht halber definieren wir für die Umwandlung erst einmal eine Funktion mit dem Namen `convert_from_mm` und dem Parameter `row`. Unter diesem Namen kann im Code der Funktion auf die aktuelle Zeile zugegriffen werden. Das sieht in Python so aus:

```
def convert_from_mm(row):
  return pd.to_numeric(row['petal width'].replace(' mm', '')) / 10
```

Funktionen in Python

Du definierst eine Funktion mit dem Schüsselwort `def`, dem Namen der Funktion und einer Liste von erwarteten Parametern. Das Ergebnis wird mit `return` geliefert.

Blöcke, wie der Körper einer Funktion, werden in Python einzig durch Einrückungen ausgedrückt. Das bedeutet, der Block fängt mit der Einrückung an und endet, wenn auch die Einrückung endet.

Diesen Code müssen wir nun auf jeden einzelnen Wert von `'petal width'` anwenden und nicht nur einmal auf 2 mm. Mit der `apply`-Funktion kannst du genau das tun, *nämlich* einen bestimmten Code für jede Zeile ausführen. Dazu geben wir unsere oben definierte Funktion an, die *dann* nach und nach für alle Werte ausgeführt wird.

```
df['petal width'] = df.apply(convert_from_mm, axis='columns')
```

`axis='columns'` ist etwas verwirrend. Es besagt, dass wir die Datensätze über die Spaltenachse einfüttern, was effektiv bedeutet: zeilenweise und nicht etwa spaltenweise. Dies ist meistens das, was wir wollen.

Dies wird nun für jede Zeile ausgeführt, und das Ergebnis überschreibt die bisherige `'petal width'`-Serie. Zur Überprüfung gucken wir uns noch einmal die Werte an und sehen, dass alles korrekt nach Zentimeter konvertiert wurde:

```
sepal length  sepal width  petal length petal width  class
5.1           3.5          1.4          0.2          Iris-setosa
4.9           3.0          1.4          0.2          Iris-setosa
4.7           3.2          1.3          0.2          Iris-setosa
4.6           3.1          1.5          0.2          Iris-setosa
5.0           3.6          1.4          0.2          Iris-setosa
```

Unplausible Daten (Ausreißer)

Das erste Problem bei unplausiblen Daten: Man muss sie erst einmal finden. Jeden einzelnen Datensatz durchzugehen, ist selbst bei einer relativ kleinen Menge von Datensätzen zumindest fehleranfäl-

lig und nervtötend, sehr bald aber auch schlicht nicht praktikabel. Wir bedienen uns daher den Mitteln der Statistik, um überhaupt erst einmal eine Idee davon zu bekommen, ob wir verdächtige Werte in unseren Daten haben. Mit dieser Information können wir dann weitere Untersuchungen vornehmen. Zuerst aber die statistische Untersuchung:

```
df.describe()
```

Diese Funktion ist beinahe Magie und liefert dir eine umfassende, aber dennoch kompakte Darstellung statistischer Maße.

Die `describe()`-Funktion arbeitet nur dann gut, wenn deine Daten bereits strukturell aufgeräumt sind, d.h., wenn sie tatsächlich durchgehend numerisch sind und keine Lücken mehr enthalten.

Sieh dir dazu bitte das Ergebnis in Abbildung 3-2 an.

	sepal length	sepal width	petal length	petal width
count	150.000000	150.000000	150.000000	150.000000
mean	6.191333	3.054533	3.758667	1.198667
std	4.338310	0.433205	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	58.000000	4.400000	6.900000	2.500000

Abbildung 3-2: Die statistische Auswertung unserer Daten

Pro Zeile findest du jeweils einen wichtigen statistischen Wert für jeden Wert in unserem Datensatz.

count, mean, std
: Die Anzahl der Werte, der Mittelwert und die Standardabweichung.

min, max
: Der jeweils kleinste und größte Wert.

25%, 50%, 75%

Die drei Standardperzentile. Sie geben Aufschluss über die Verteilung der Werte. Das *25%*-Perzentil gibt den Wert an, der größer ist als 25% aller anderen Werte. Dasselbe gilt dementsprechend für *50%* und *75%*. *50%* heißt auch *Median*.

Was fällt dir auf? Ein maximaler Wert von 58 bei *sepal length* sieht schon ganz schön verdächtig aus, meinst du nicht? Eine Blume mit einem Blütenblatt von mehr als einem halben Meter? Noch klarer wird es, wenn wir uns das grafisch aufbereitet in Histogrammen ansehen, die wir mit einem minimalen Aufruf erstellen. Ein Histogramm bildet alle Werte auf der x-Achse ab. Auf der y-Achse siehst du dann die Häufigkeit jedes vorkommenden Werts. Dabei kannst du auch angeben, ob du nun jeden einzelnen Wert mit seiner Häufigkeit sehen willst oder Werte zusammenfassen möchtest, die nebeneinanderliegen. Diese Zusammenfassungen nennt man auch *Bins*.

Pro Variable gibt es ein Histogramm, per Default werden dabei nur zehn Bins angezeigt, also nicht jeder Wert einzeln.

```
df.hist()
```

Das Ergebnis in Abbildung 3-3 ist besonders spannend in dem Histogramm unten links für *sepal length*. Ganz rechts gibt es genau einen einzigen Wert 58, alle anderen Werte sind unter 10. Das ist also unser Ausreißer.

Wir haben eigentlich jetzt genug gesehen, aber weil dies so tolle Funktionen sind: Sie arbeiten auch gruppiert für die einzelnen Klassen:

```
df.groupby('class').describe()
df.groupby('class').hist()
```

Damit bekommst du dann dieselbe Art Grafik, allerdings pro Klasse eine eigene.

Wir wollen jetzt aber unseren Ausreißer korrigieren. Wir könnten ihn wie einen fehlenden Wert betrachten und hätten dann die gleichen Optionen, die wir oben beschrieben haben.

Allerdings drängt sich hier ein anderer Verdacht auf: Nur das Komma ist verrutscht! Alle anderen Werte sind nämlich eine Größenordnung kleiner, und ein verrutschtes oder vergessenes Komma ist ein häufig auftretender Fehler. Dazu kommt etwas Domänenwissen, das wir mit dem kompletten Datensatz abgleichen. Die folgende Abfrage für den Datensatz mit der `'sepal length'` 58 ...

```
df[df['sepal length'] == 58]
```

... ergibt diese Antwort:

```
index sepal length sepal width petal length petal width   class
143       58.0          2.7         5.1         1.9   Iris-
virginica
```

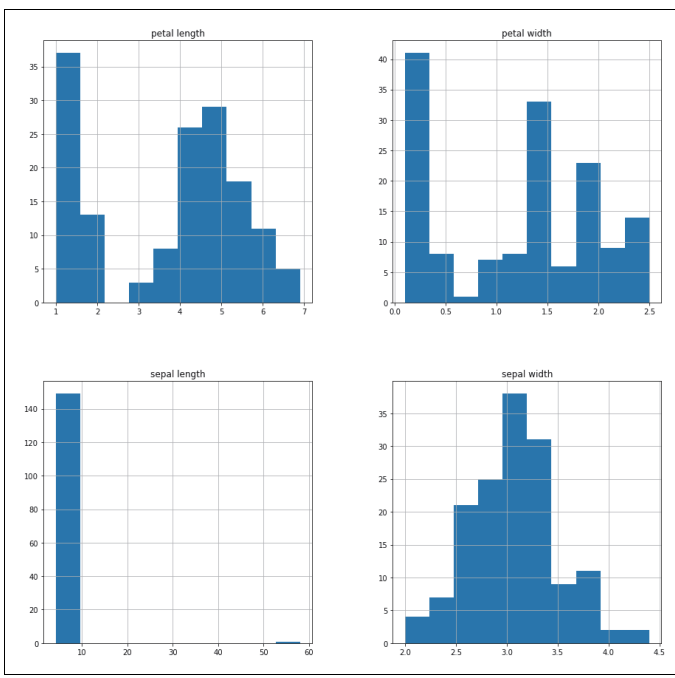

Abbildung 3-3: Histogramm: Wie häufig hat eine Variable einen gewissen Wert angenommen?

Dabei ist Sepalum (Sepal) der grüne Teil einer Blüte, der die Blüte z. B. nachts umschließt und schützt. Das Petal sind die Blütenblätter selbst. Es ist also anzunehmen, dass das Sepalum etwas größer ist als das Petal, aber nicht um mehr als den Faktor 10. Dass eine solche Korrelation zwischen den beiden Werten besteht, können wir sehr einfach mit der Bibliothek Seaborn darstellen:

```
import seaborn as sns
sns.jointplot(df['sepal length'], df['petal length'])
```

Das Ergebnis in Abbildung 3-4 zeigt sehr deutlich die annähernd lineare Korrelation mit genau einem Ausreißer. Der Pearson-Korrelationskoeffizient *r* von 0,23 gibt dabei an, wie stark die Werte tatsächlich linear korrelieren. 1 wäre perfekt, 0 gar nicht. Der relativ schwache Wert kommt durch den Ausreißer.

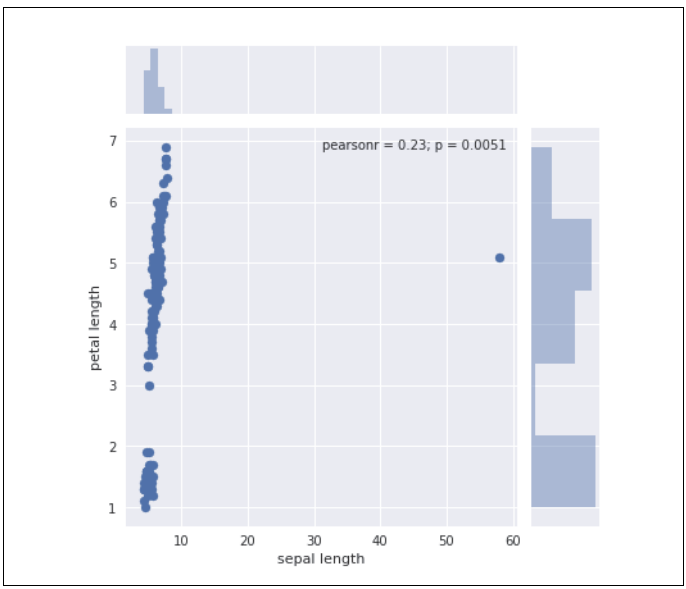

Abbildung 3-4: Annähernd linear mit einem Ausreißer: Korrelation zwischen der Länge des Kelchblatts und der des Blütenblatts

Korrigieren wir diesen Wert also erst einmal von Hand auf 5,8:

```
df.loc[143, 'sepal length'] = 5.8
```

Wenn wir die gleiche Grafik nach der Korrektur noch einmal berechnen, bekommen wir ein erstaunlich verändertes Bild, das du in Abbildung 3-5 sehen kannst. Hier ist der Pearson-Korrelationskoeffizient *r* auch erwartet nahe an 1, d.h., die beiden Werte sind stark linear korreliert.

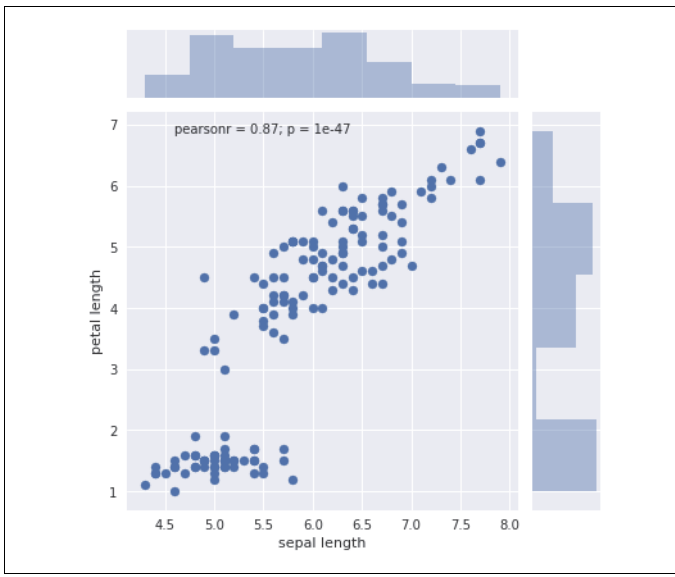

Abbildung 3-5: Feinere Auflösung: Korrelation zwischen der Länge des Kelchblatts und der des Blütenblatts

Es fallen zwei Dinge auf: Zum einen sieht die Grafik ganz anders aus und auch gar nicht mehr so schön linear. Da kommt daher, dass nun anders skaliert wird, da in der Grafik der Ausreißer jetzt nicht mehr angezeigt werden muss. Was wir vorher annähernd als Gerade gesehen haben, entpuppt sich bei genauerem Hinsehen als zweiteilige Gruppierung. Dennoch ist das Maß der linearen Korrela-

tion nun mit 0,87 deutlich höher als vorher. Das ergibt Sinn, da wir unseren Ausreißer ja los sind.

Damit haben wir nun einen komplett bereinigten Datensatz. Für einen schönen Abschluss sehen wir uns noch einmal in der Übersicht eine Korrelationsmatrix zwischen allen Werten in Abbildung 3-6 an.

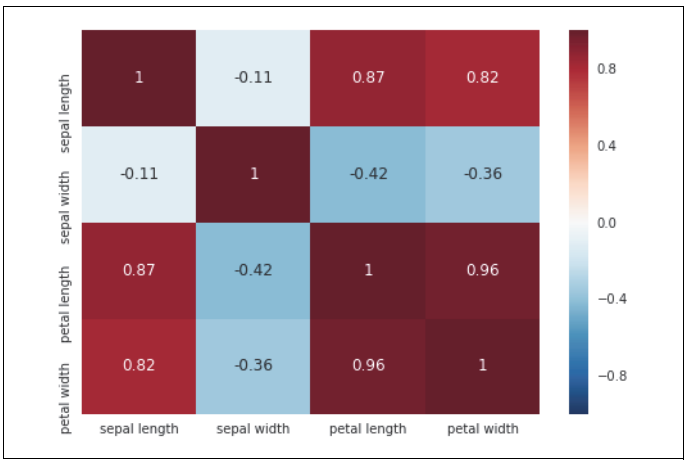

Abbildung 3-6: Überblick über alle Korrelationen

Diese Matrix in Abbildung 3-6 zeigt einen Überblick über alle Korrelationen. Sie ist spiegelsymmetrisch an der Diagonale, das heißt, du musst dir nur die untere Hälfte ansehen. Manche Korrelationen erscheinen offensichtlich, andere aber sind erstaunlich. Solch eine Darstellung ist extrem dicht und ideal für einen sehr schnellen Überblick. Du kannst sie mit nur zwei Zeilen Code mit Seaborn erzeugen:

```
corrmat = df.corr()
sns.heatmap(corrmat, annot=True)
```

Die eigentliche Matrix erzeugt Pandas, Seaborn stellt sie als Heatmap dar. Dabei wird die Farbe dunkler, je höher der absolute Wert ist.

Mit dem Wissen und den Techniken dieses Kapitels wirst du schon sehr weit kommen. Vieles in diesem Kapitel erfordert jedoch vor allem Fleißarbeit und ein wenig Intuition, wonach zu suchen ist und wie die Daten visualisiert werden sollten. In dem Notebook zu diesem Kapitel haben wir noch einige weitere Visualisierungen als Inspiration für dich zusammengestellt.

Weiterführende Links

- Pandas: *https://pandas.pydata.org/*
- NumPy: *https://numpy.org/*
- Matplotlib: *https://matplotlib.org/*
- Seaborn: *https://seaborn.pydata.org/*
- Ein gutes NumPy-Tutorial: *https://cs231n.github.io/python-numpy-tutorial/*

Inhalt