KAPITEL 4

Numerische Daten verarbeiten

4.0 Einführung

Quantitative Daten gehen auf Messungen zurück – ob es sich um Klassengrößen, monatliche Umsätze oder Schulnoten handelt. Naturgemäß drückt man diese Quantitäten numerisch aus (z.B. 29 Schüler, 529.392 Euro Umsatz usw.). In diesem Kapitel behandeln wir zahlreiche Strategien, um rohe numerische Daten in Merkmale zu überführen, die auf Algorithmen für maschinelles Lernen zugeschnitten sind.

4.1 Ein Merkmal neu skalieren

Problem

Die Werte eines numerischen Merkmals sollen zwischen zwei Werten skaliert werden.

Lösung

Mit der `MinMaxScaler`-Implementierung der Bibliothek `scikit-learn` lässt sich ein Merkmalsarray skalieren:

```
# Bibliotheken laden
import numpy as np
from sklearn import preprocessing

# Merkmal erzeugen
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])
# Skalierer erzeugen
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Merkmal skalieren
scaled_feature = minmax_scale.fit_transform(feature)
```

```
# Merkmal anzeigen
scaled_feature

array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Diskussion

Skalieren ist eine häufige Aufgabe in der Vorverarbeitung beim maschinellen Lernen. Viele der Algorithmen, die das Buch später noch beschreibt, gehen davon aus, dass sich alle Merkmale auf der gleichen Skala befinden, in der Regel zwischen 0 und 1 oder zwischen –1 und 1. Von den zahlreichen Neuskalierungstechniken ist die sogenannte *Min-Max-Skalierung* eines der einfachsten Verfahren. Die Min-Max-Skalierung verwendet die Kleinst- und Größtwerte eines Merkmals, um Werte in einen bestimmten Bereich zu skalieren. Insbesondere führt diese Skalierung die folgende Berechnung aus:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Hierin ist x der Merkmalsvektor, x'_i ein individuelles Element des Merkmals x und x'_i das normalisierte Element. Im Beispiel zeigt das ausgegebene Array, dass das Merkmal erfolgreich zwischen 0 und 1 skaliert wurde:

```
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Die `MinMaxScaler`-Implementierung der Bibliothek `scikit-learn` bietet zwei Optionen, um ein Merkmal zu skalieren. Bei der ersten berechnet man die Kleinst- und Größtwerte des Merkmals mit `fit` und skaliert dann das Merkmal mit der Methode `transform`. Die zweite Option besteht darin, beide Operationen auf einmal mit `fit_transform` abzuwickeln. Mathematisch unterscheiden sich beide Optionen nicht, doch manchmal gibt es einen praktischen Nutzen, die Operationen separat auszuführen, weil sich damit die gleiche Transformation auf verschiedene Mengen von Daten anwenden lässt.

Siehe auch

- Feature scaling, Wikipedia (*http://bit.ly/2Fuug2Z*)
- About Feature Scaling and Normalization, Sebastian Raschka (*http://bit.ly/2FwwRcM*)

4.2 Ein Merkmal standardisieren

Problem

Sie möchten ein Merkmal transformieren, sodass es einen Mittelwert von 0 und eine Standardabweichung von 1 hat.

Lösung

Beide Transformationen lassen sich mit dem `StandardScaler` der Bibliothek `scikit-learn` ausführen:

```
# Bibliotheken laden
import numpy as np
from sklearn import preprocessing

# Merkmal erzeugen
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Skalierer erzeugen
scaler = preprocessing.StandardScaler()

# Das Merkmal transformieren
standardized = scaler.fit_transform(x)

# Merkmal anzeigen
standardized

array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

Diskussion

Eine häufige Alternative zur Min-Max-Skalierung, die in Rezept 4.1 diskutiert wurde, ist das Skalieren von Merkmalen, sodass sie ungefähr einer standardmäßigen Normalverteilung entsprechen. Die Standardisierung transformiert die Daten so, dass sie einen Mittelwert $\bar{x}$ von 0 und eine Standardabweisung σ von 1 aufweisen. Insbesondere wird jedes Element im Merkmal mit

$$x'_i = \frac{x_i - \bar{x}}{\sigma}$$

transformiert, wobei x'_i unsere standardisierte Form von x_i ist. Das transformierte Merkmal stellt die Anzahl der Standardabweichungen dar, die der ursprüngliche Wert vom Mittelwert entfernt ist (in der Statistik auch als *z-standardisierter Wert* oder *z-Score* bekannt).

Standardisierung ist eine gängige Skalierungsmethode für die Vorverarbeitung beim maschinellen Lernen und wird meiner Erfahrung nach öfter als die Min-Max-Skalierung verwendet. Allerdings hängt das auch vom Lernalgorithmus ab. So funktioniert die Hauptkomponentenanalyse oftmals besser mithilfe der Standardisierung, während Min-Max-Skalierung häufig für neuronale Netze empfohlen wird. (Beide Algorithmen kommen später im Buch noch zur Sprache.) Als Faustregel empfehle ich, normalerweise mit Standardisierung zu arbeiten, sofern es keinen besonderen Grund gibt, auf eine Alternative auszuweichen.

Von der Wirkung der Standardisierung können Sie sich ein Bild machen, wenn Sie sich den Mittelwert und die Standardabweichung in der Ausgabe unserer Lösung betrachten:

```
# Mittelwert und Standardabweichung ausgeben
print("Mittelwert:", round(standardized.mean()))
print("Standardabweichung:", standardized.std())

Mittelwert: 0.0
Standardabweichung: 1.0
```

Signifikante Ausreißer in den Daten können sich negativ auf die Standardisierung auswirken, weil sie sich in Mittelwert und Varianz des Merkmals niederschlagen. In diesem Szenario ist es oftmals hilfreich, das Merkmal mit Median und Quartilen erneut zu skalieren. In der Bibliothek `scikit-learn` lässt sich das mit der Methode `RobustScaler` bewerkstelligen:

```
# Skalierer erstellen
robust_scaler = preprocessing.RobustScaler()

# Merkmal transformieren
robust_scaler.fit_transform(x)

array([[ -1.87387612],
       [ -0.875     ],
       [  0.        ],
       [  0.125     ],
       [ 10.61488511]])
```

4.3 Beobachtungen normalisieren

Problem

Sie möchten die Merkmalswerte von Beobachtungen zur *1-Norm* skalieren (sodass sie eine Gesamtlänge von 1 haben).

Lösung

Verwenden Sie die Methode `Normalizer` mit einem `norm`-Argument:

```
# Bibliotheken laden
import numpy as np
from sklearn.preprocessing import Normalizer

# Merkmalsmatrix erzeugen
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# Normalisierer erzeugen
normalizer = Normalizer(norm="l2")

# Merkmalsmatrix transformieren
normalizer.transform(features)

array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Diskussion

Viele Skalierungsmethoden (z. B. Min-Max-Skalierung und Standardisierung) arbeiten auf Merkmalen. Es ist aber auch möglich, über einzelnen Beobachtungen zu skalieren. Die Methode `Normalizer` skaliert die Werte für einzelne Beobachtungen entsprechend der 1-Norm (die Summe ihrer Längen ist 1). Eine derartige Skalierung wird oft angewendet, wenn mehrere gleichwertige Merkmale existieren (z. B. bei der Textklassifizierung, wenn jedes Wort oder jede Gruppe aus n Wörtern ein Merkmal ist).

Die Methode `Normalizer` bietet drei `norm`-Optionen, wobei die *euklidische Norm* (oft *L2* genannt) das Standardargument ist:

$$\| x \|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

Hierin ist x eine einzelne Beobachtung, und x_n ist der Wert dieser Beobachtung für das n-te Merkmal.

```
# Merkmalsmatrix transformieren
features_l2_norm = Normalizer(norm="l2").transform(features)

# Merkmalsmatrix anzeigen
features_l2_norm

array([[ 0.70710678,  0.70710678],
```

```
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Alternativ können Sie die *Manhattan-Norm* (*L1*) spezifizieren:

$$\| x \|_1 = \sum_{i=1}^{n} |x_i|.$$

```
# Merkmalsmatrix transformieren
features_l1_norm = Normalizer(norm="l1").transform(features)

# Merkmalsmatrix anzeigen
features_l1_norm

array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])
```

Intuitiv kann man sich die L2-Norm als Wegstrecke für einen Vogel (d. h. als gerade Linie) zwischen zwei Punkten in New York vorstellen, während L1 die Wegstrecke ist, die ein Fußgänger auf der Straße zurücklegt (einen Block nach Norden gehen, einen Block nach Osten, einen Block nach Norden, einen Block nach Osten usw.), weshalb man auch von *Manhattan-Abstand* oder *Taxi-Metrik* spricht.

Praktisch skaliert `norm='l1'` die Werte einer Beobachtung so, dass ihre Summe gleich 1 ist, was manchmal eine wünschenswerte Qualität ist:

```
# Summe ausgeben
print("Summe der Werte der ersten Beobachtung:",
   features_l1_norm[0, 0] + features_l1_norm[0, 1])

Summe der Werte der ersten Beobachtung: 1.0
```

4.4 Polynom- und Interaktionsmerkmale erzeugen

Problem

Sie möchten Polynom- und Interaktionsmerkmale erzeugen.

Lösung

Auch wenn sich mancher dafür entscheiden mag, Polynom- und Interaktionsmerkmale manuell zu erzeugen, lässt sich das mit einer integrierten Methode von `scikit-learn` einfacher bewerkstelligen:

```
# Bibliotheken laden
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Merkmalsmatrix erzeugen
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# PolynomialFeatures-Objekt erzeugen
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# Polynomiale Merkmale erzeugen
polynomial_interaction.fit_transform(features)

array([[ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.]])
```

Der Parameter `degree` bestimmt den maximalen Grad des Polynoms. Zum Beispiel erzeugt `degree=2` neue Merkmale, die zur zweiten Potenz erhoben (quadriert) werden:

$$x_1,\ x_2,\ x_1^2,\ x_2^2$$

Dementsprechend erzeugt `degree=3` neue Merkmale, die zur dritten Potenz erhoben werden:

$$x_1,\ x_2,\ x_1^2,\ x_2^2,\ x_1^3,\ x_2^3$$

Darüber hinaus schließt `PolynomialFeatures` standardmäßig Interaktionsmerkmale ein:

$$x_1 x_2$$

Wenn man `interaction_only` auf `True` setzt, lassen sich die erzeugten Merkmale auf Interaktionsmerkmale beschränken:

```
interaction = PolynomialFeatures(degree=2,
              interaction_only=True, include_bias=False)

interaction.fit_transform(features)

array([[ 2.,  3.,  6.],
       [ 2.,  3.,  6.],
       [ 2.,  3.,  6.]])
```

Diskussion

Polynomiale Merkmale werden oftmals erzeugt, wenn man von einer nichtlinearen Beziehung zwischen den Merkmalen und dem Ziel ausgehen kann. So könnte man vermuten, dass die Wahrscheinlichkeit für eine schwere Krankheit zeitlich nicht

konstant ist, sondern mit zunehmendem Alter stärker ansteigt. Diesen nicht konstanten Effekt können wir in einem Merkmal x codieren, indem wir die Formen höherer Ordnung (x^2, x^3 usw.) generieren.

Darüber hinaus haben wir es oftmals mit Situationen zu tun, in denen die Wirkung eines Merkmals von einem anderen Merkmal abhängig ist. Angenommen, wir wollten vorhersagen, ob unser Kaffee süß ist oder nicht. Dabei stützen wir uns auf zwei Merkmale: zum einen, ob der Kaffee gerührt wurde, und zum anderen, ob wir Zucker zugegeben haben. Für sich allein genommen kann kein Merkmal die Süße des Kaffees vorhersagen, doch die Kombination ihrer Wirkungen erlaubt es. Das heißt, ein Kaffee ist nur dann süß, wenn der Kaffee Zucker enthält und umgerührt wurde. Die Wirkungen jedes Merkmals auf das Ziel (die Süße) sind unabhängig voneinander. Diese Beziehung können wir codieren, indem wir ein Interaktionsmerkmal einbeziehen, das das Produkt der einzelnen Merkmale ist.

4.5 Merkmale transformieren

Problem

Sie möchten eine benutzerdefinierte Transformation in ein oder mehrere Merkmale vornehmen.

Lösung

Aus der Bibliothek `scikit-learn` wenden Sie mit der Methode `FunctionTransformer` eine Funktion auf einen Satz von Merkmalen an:

```
# Bibliothek laden
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Merkmalsmatrix erzeugen
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Eine einfache Funktion definieren
def add_ten(x):
    return x + 10

# Transformer erzeugen
ten_transformer = FunctionTransformer(add_ten)

# Merkmalsmatrix transformieren
ten_transformer.transform(features)

array([[12, 13],
       [12, 13],
       [12, 13]])
```

Die gleiche Transformation lässt sich in pandas mithilfe von apply erzeugen:

```
# Bibliothek laden
import pandas as pd

# DataFrame erzeugen
df = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Funktion anwenden
df.apply(add_ten)
```

	feature_1	feature_2
0	12	13
1	12	13
2	12	13

Diskussion

Es ist üblich, dass man einige benutzerdefinierte Transformationen für ein oder mehrere Merkmale vornehmen möchte. Zum Beispiel könnten wir ein Merkmal erzeugen wollen, das der natürliche Logarithmus der Werte des anderen Merkmals ist. Hierzu können Sie eine Funktion erstellen und sie dann entweder mit der Methode FunctionTransformer der Bibliothek scikit-learn oder mit der Methode apply der Bibliothek pandas auf Merkmale abbilden. In der angegebenen Lösung haben wir mit add_ten eine sehr einfache Funktion erzeugt, die 10 auf jede Eingabe addiert. Es gibt aber keinen Grund, warum wir nicht eine weitaus komplexere Funktion definieren könnten.

4.6 Ausreißer erkennen

Problem

Es sollen extreme Beobachtungen identifiziert werden.

Lösung

Ausreißer zu erkennen, ist leider mehr eine Kunst als eine Wissenschaft. Eine übliche Methode ist es aber, von normalverteilten Daten auszugehen und auf dieser Annahme aufbauend eine Ellipse um die Daten zu »zeichnen«, was jede Beobachtung innerhalb der Ellipse als innen liegend klassifiziert (mit 1 beschriftet) und jede Beobachtung außerhalb der Ellipse als Ausreißer (mit -1 beschriftet):

```
# Bibliotheken laden
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs
```

```
# Simulierte Daten erzeugen
features, _ = make_blobs(n_samples = 10,
                         n_features = 2,
                         centers = 1,
                         random_state = 1)

# Die Werte der ersten Beobachtung durch extreme Werte ersetzen
features[0,0] = 10000
features[0,1] = 10000

# Detektor erzeugen
outlier_detector = EllipticEnvelope(contamination=.1)

# Detektor anpassen
outlier_detector.fit(features)

# Ausreißer vorhersagen
outlier_detector.predict(features)

array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

Eine wesentliche Einschränkung dieses Verfahrens ist die Notwendigkeit, mit dem Parameter `contamination` den Anteil der Ausreißer an den Beobachtungen anzugeben – einen Wert, den wir nicht kennen. Stellen Sie sich `contamination` als unsere Schätzung für die Reinheit unserer Daten vor. Wenn wir nur wenige Ausreißer in unseren Daten erwarten, können wir `contamination` auf einen kleinen Wert setzen. Ist es dagegen sehr wahrscheinlich, dass die Daten Ausreißer enthalten, setzen wir den Parameter auf einen höheren Wert.

Anstatt Beobachtungen als Ganzes zu betrachten, können wir uns einzelne Merkmale ansehen und extreme Werte in diesen Merkmalen mithilfe des *Interquartilsabstands* (*Interquartile Range*, IQR) identifizieren:

```
# Ein Merkmal erzeugen
feature = features[:,0]

# Eine Funktion erstellen, um den Index von Ausreißern zurückzugeben
def indicies_of_outliers(x):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# Funktion ausführen
indicies_of_outliers(feature)

(array([0]),)
```

Der Interquartilsabstand (IQR) ist die Differenz zwischen dem ersten und dritten Quartil einer Stichprobe. Den IQR können Sie sich als Verteilung der Datenmasse vorstellen, wobei die Ausreißer die Beobachtungen sind, die weit entfernt von der Hauptkonzentration der Daten liegen. Als Ausreißer definiert man üblicherweise

jeden Wert, der 1,5 IQRs kleiner als das erste Quartil oder 1,5 IQRs größer als das dritte Quartil ist.

Diskussion

Es gibt keine beste Technik für das Erkennen von Ausreißern. Stattdessen arbeiten wir mit einer ganzen Kollektion von Techniken, die alle ihre eigenen Vor- und Nachteile haben. Als beste Strategie erweist es sich oftmals, mehrere Techniken auszuprobieren (z.B. sowohl `EllipticEnvelope` als auch IQR-basierte Erkennung) und die Ergebnisse als Ganzes zu bewerten.

Nach Möglichkeit sollten wir uns die als Ausreißer erkannten Beobachtungen genauer ansehen und versuchen, sie zu verstehen. Wenn beispielsweise in einer Stichprobe von Häusern eines der Merkmale die Anzahl der Räume beschreibt, sollten Sie sich fragen, ob es sich bei einem Ausreißer mit 100 Räumen wirklich um ein Haus handelt oder tatsächlich um ein Hotel, das falsch klassifiziert ist.

Siehe auch

- Drei Verfahren, Ausreißer zu erkennen (und die Quelle der IQR-Funktion, die in diesem Rezept verwendet wurde) (*http://bit.ly/2FzMC2k*)

4.7 Mit Ausreißern umgehen

Problem

Die Daten enthalten Ausreißer.

Lösung

In der Regel stehen uns drei Strategien zur Verfügung, um mit Ausreißern umgehen zu können. Erstens können wir sie löschen:

```
# Bibliothek laden
import pandas as pd

# DataFrame erzeugen
houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# Beobachtungen filtern
houses[houses['Bathrooms'] < 20]
```

	Price	Bathrooms	Square_Feet
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

Zweitens können wir sie als Ausreißer markieren und als Merkmal einbinden:

```
# Bibliothek laden
import numpy as np

# Merkmal basierend auf boolescher Bedingung erzeugen
houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)

# Daten anzeigen
houses
```

	Price	Bathrooms	Square_Feet	Outlier
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

Schließlich können wir das Merkmal transformieren, um den Ausreißereffekt zu dämpfen:

```
# Merkmal logarithmieren
houses["Log_Of_Square_Feet"] = [np.log(x) for x in houses["Square_Feet"]]

# Daten anzeigen
houses
```

	Price	Bathrooms	Square_Feet	Outlier	Log_Of_Square_Feet
0	534433	2.0	1500	0	7.313220
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.313220
3	4322032	116.0	48000	1	10.778956

Diskussion

Ähnlich wie beim Erkennen von Ausreißern gibt es keine verbindliche Regel, wie mit Ausreißern umzugehen ist. Wie man sie behandelt, sollte sich nach zwei Aspekten richten. Zum einen sollte man überlegen, was sie zu Ausreißern macht. Ist davon auszugehen, dass es sich um Fehler in den Daten handelt, etwa durch einen defekten Sensor oder einen falsch codierten Wert, kann man die Beobachtung fallen lassen oder Ausreißerwerte durch `NaN` ersetzen, da man diesen Werten nicht trauen kann. Wenn man jedoch annimmt, dass die Ausreißer echte Extremwerte

sind (z. B. ein Haus (eine Villa) mit 200 Badezimmern), ist es besser, sie als Ausreißer zu markieren oder ihre Werte zu transformieren.

Zum anderen sollte der Umgang mit Ausreißern auf unserem Ziel für maschinelles Lernen basieren. Wenn wir zum Beispiel Hauspreise anhand der Merkmale des Hauses vorhersagen wollen, könnten wir vernünftigerweise davon ausgehen, dass der Preis für Villen mit über 100 Badezimmern von einer anderen Dynamik getrieben wird als für normale Einfamilienhäuser. Wenn wir darüber hinaus ein Modell trainieren, um es als Teil einer Webanwendung für Online-Immobilienfinanzierung einzusetzen, können wir annehmen, dass unsere potenziellen Benutzer keine Milliardäre sind, die eine Villa kaufen wollen.

Was also sollen wir tun, wenn Ausreißer vorhanden sind? Überlegen, warum sie Ausreißer sind, ein Endziel für die Daten im Auge haben und vor allem daran denken, dass die Entscheidung, Ausreißer nicht zu beachten, wiederum eine Entscheidung mit Konsequenzen ist.

Beachten Sie auch: Wenn es Ausreißer gibt, ist eine Standardisierung möglicherweise nicht sinnvoll, weil Mittelwert und Varianz erheblich von den Ausreißern beeinflusst werden. Verwenden Sie in diesem Fall eine Skalierungsmethode wie `RobustScaler`, die robuster gegenüber Ausreißern ist.

Siehe auch

- RobustScaler-Dokumentation (*http://bit.ly/2DcgyNT*)

4.8 Merkmale diskretisieren

Problem

Sie möchten ein numerisches Merkmal in getrennte Intervalle aufteilen.

Lösung

Abhängig davon, wie Sie die Daten aufteilen möchten, kommen vor allem zwei Techniken infrage. Erstens können Sie das Merkmal in Bezug auf einen bestimmten Schwellenwert binarisieren:

```
# Bibliotheken laden
import numpy as np
from sklearn.preprocessing import Binarizer

# Merkmal erzeugen
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])
```

```
# Binarisierer erzeugen
binarizer = Binarizer(18)

# Merkmal transformieren
binarizer.fit_transform(age)

array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Zweitens lassen sich numerische Merkmale in Bezug auf mehrere Schwellenwerte aufteilen:

```
# Merkmal in Intervalle aufteilen
np.digitize(age, bins=[20,30,64])

array([[0],
       [0],
       [1],
       [2],
       [3]])
```

Beachten Sie, dass die Argumente für den Parameter `bins` den linken Rand jedes Intervalls kennzeichnen. So schließt zum Beispiel das Argument `20` nicht das Element mit dem Wert 20 ein, sondern nur die beiden Werte kleiner als 20. Dieses Verhalten lässt sich ändern, indem man den Parameter `right` auf `True` setzt:

```
# Merkmal auf Intervalle aufteilen
np.digitize(age, bins=[20,30,64], right=True)

array([[0],
       [0],
       [0],
       [2],
       [3]])
```

Diskussion

Diskretisierung kann eine erfolgreiche Strategie sein, wenn es Grund zur Annahme gibt, dass sich ein numerisches Merkmal eher wie ein kategorisches Merkmal verhalten sollte. Zum Beispiel könnten wir glauben, dass sich die Konsumgewohnheiten 19- und 20-Jähriger kaum unterscheiden, es aber einen signifikanten Unterschied gibt zwischen den 20- und 21-Jährigen (dem Alter in den USA, ab dem Jugendliche Alkohol konsumieren dürfen). In diesem Beispiel könnte es hilfreich sein, die Daten der Einzelpersonen aufzuteilen in zwei Gruppen – diejenigen, die Alkohol trinken dürfen, und diejenigen, die es nicht dürfen. In ähnlicher Weise könnte es in anderen Fällen zweckmäßig sein, die Daten in drei oder mehr Intervalle zu diskretisieren.

Die Lösung hat zwei Methoden der Diskretisierung gezeigt – die Methode `Binarizer` der Bibliothek `scikit-learn` für zwei Bereiche und die Methode `digitize` der Bibliothek `NumPy` für drei oder mehr Bereiche. Allerdings kommt auch `digitize` infrage, um genau wie `Binarizer` Merkmale zu binarisieren, indem man lediglich einen einzigen Schwellenwert angibt:

```
# Merkmal binarisieren
np.digitize(age, bins=[18])

array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Siehe auch

- Dokumentation zu `digitize` (*http://bit.ly/2HSciFP*)

4.9 Beobachtungen durch Clustern gruppieren

Problem

Sie möchten Beobachtungen clustern, sodass ähnliche Beobachtungen zusammen gruppiert werden.

Lösung

Wenn Sie wissen, dass es *k* Gruppen gibt, können Sie mit dem *k*-Means-Algorithmus ähnliche Beobachtungen gruppieren und ein neues Merkmal ausgeben, das die Gruppenmitgliedschaft jeder Beobachtung enthält:

```
# Bibliotheken laden
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Simulierte Merkmalsmatrix erzeugen
features, _ = make_blobs(n_samples = 50,
                         n_features = 2,
                         centers = 3,
                         random_state = 1)

# DataFrame erzeugen
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# k-Means-Clusterer erzeugen
clusterer = KMeans(3, random_state=0)
```

```
# Clusterer anpassen
clusterer.fit(features)

# Werte vorhersagen
dataframe["group"] = clusterer.predict(features)

# Die ersten Beobachtungen anzeigen
dataframe.head(5)
```

	feature_1	feature_2	group
0	–9.877554	–3.336145	0
1	–7.287210	–8.353986	2
2	–6.943061	–7.023744	2
3	–7.440167	–8.791959	2
4	–6.641388	–8.075888	2

Diskussion

Die Clustering-Algorithmen streifen wir hier nur kurz. Später im Buch gehen wir ausführlicher darauf ein. Momentan soll der Hinweis genügen, dass sich Clustern als Vorverarbeitungsschritt gut eignet. Insbesondere können wir nicht überwachte Lernalgorithmen wie *k*-Means verwenden, um Beobachtungen in Gruppen zu clustern. Das Ergebnis ist ein kategorisches Merkmal, bei dem ähnliche Beobachtungen zur selben Gruppe gehören.

Machen Sie sich keine Sorgen, sollten Sie noch nicht alles sofort verstanden haben: Lassen Sie erst einmal den Gedanken beiseite, dass Clustern bei der Vorverarbeitung verwendet werden kann. Und wenn Sie es wirklich nicht erwarten können, springen Sie einfach zu Kapitel 19.

4.10 Beobachtungen mit fehlenden Werten löschen

Problem

Beobachtungen, die fehlende Werte enthalten, sollen gelöscht werden.

Lösung

Beobachtungen mit fehlenden Werten lassen sich geschickt mit einer entsprechenden Zeile von `NumPy` löschen:

```
# Bibliothek laden
import numpy as np

# Merkmalsmatrix erzeugen
```

```
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# Nur Beobachtungen, die nicht als fehlend (mit ~ gekennzeichnet)
# markiert sind
features[~np.isnan(features).any(axis=1)]

array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

Alternativ können Sie unvollständige Beobachtungen mithilfe von pandas löschen:

```
# Bibliothek laden
import pandas as pd

# Daten laden
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Beobachtungen mit fehlenden Werten entfernen
dataframe.dropna()
```

	feature_1	feature_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Diskussion

Die meisten Algorithmen für maschinelles Lernen sind nicht in der Lage, fehlende Werte im Ziel und in den Merkmalsarrays zu behandeln. Deshalb dürfen wir fehlende Werte in unseren Daten nicht ignorieren und müssen uns mit diesem Problem während der Vorverarbeitung auseinandersetzen.

Die einfachste Lösung löscht jede Beobachtung, die einen oder mehrere fehlende Werte enthält. Diese Aufgabe lässt sich schnell und einfach mit NumPy oder pandas bewerkstelligen.

Dennoch sollten Sie sehr vorsichtig sein, Beobachtungen mit fehlenden Werten zu löschen. Sie zu löschen, ist die »nukleare Option«, da unser Algorithmus den Zugriff auf die Informationen verliert, die in den nicht fehlenden Werten der Beobachtung enthalten sind.

Genauso wichtig ist es, dass das Löschen von Beobachtungen je nach Ursache für die fehlenden Werte zu Verzerrungen in den Daten führen kann. Man unterscheidet drei Typen fehlender Daten:

Missing Completely At Random (MCAR)
: Die Wahrscheinlichkeit, dass ein Wert fehlt, ist von allen anderen Faktoren unabhängig. Beispielsweise könnte ein Umfrageteilnehmer seine Antwort auswürfeln: Wenn er eine Sechs würfelt, überspringt er diese Frage.

Missing At Random (MAR)
: Die Wahrscheinlichkeit, dass ein Wert fehlt, ist nicht vollkommen zufällig, sondern hängt von den Informationen ab, die in anderen Merkmalen erfasst werden. Wenn etwa eine Umfrage nach Geschlechtszugehörigkeit und Jahresgehalt fragt, werden Frauen eher die Frage überspringen; allerdings hängt ihre Nichtbeantwortung nur von den Informationen ab, die wir in unserem Merkmal zur Geschlechtszugehörigkeit erfasst haben.

Missing Not At Random (MNAR)
: Die Wahrscheinlichkeit, dass ein Wert fehlt, ist nicht zufällig und hängt von Informationen ab, die in unseren Merkmalen nicht erfasst worden sind. Zum Beispiel könnte eine Umfrage nach der Geschlechtszugehörigkeit fragen, und Frauen werden eher die Frage nach dem Gehalt übergehen, und wir haben kein Merkmal für Geschlechtsidentifikation in unseren Daten.

Manchmal ist es akzeptabel, Beobachtungen zu löschen, wenn sie vom Typ MCAR oder MAR sind. Wenn jedoch der Wert den Typ MNAR hat, ist die Tatsache, dass ein Wert fehlt, selbst eine Information. Das Löschen von MNAR-Beobachtungen kann die Verteilung der Daten verzerren, da wir Beobachtungen entfernen, die durch einen unbeobachteten systematischen Effekt hervorgerufen werden.

Siehe auch

- Die drei Typen fehlender Daten identifizieren (*http://bit.ly/2Fto4bx*)
- Imputation fehlender Daten (*http://bit.ly/2FAkKLI*)

4.11 Fehlende Werte imputieren

Problem

Es gibt fehlende Werte in den Daten, und Sie möchten diese Werte auffüllen oder vorhersagen.

Lösung

Wenn der Umfang der Daten gering ist, können Sie fehlende Werte mit dem *k*-nächste-Nachbarn-Algorithmus (KNN) vorhersagen:

```
# Bibliotheken laden
import numpy as np
from fancyimpute import KNN
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Eine simulierte Merkmalsmatrix erzeugen
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Die Merkmale standardisieren
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Den ersten Wert des ersten Merkmals durch fehlenden Wert ersetzen
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Die fehlenden Werte in der Merkmalsmatrix vorhersagen
features_knn_imputed = KNN(k=5, verbose=0).complete(standardized_features)

# Echte und imputierte Werte vergleichen
print("Echter Value:", true_value)
print("Imputierter Wert:", features_knn_imputed[0,0])

Echter Wert: 0.8730186114
Imputierter Wert: 1.09553327131
```

Alternativ können Sie das Modul `Imputer` der Bibliothek `scikit-learn` verwenden, um fehlende Werte mit dem Mittelwert, dem Median oder dem häufigsten Wert zu füllen. Die Ergebnisse sind in der Regel jedoch schlechter als mit KNN:

```
# Bibliothek laden
from sklearn.preprocessing import Imputer

# Imputer erzeugen
mean_imputer = Imputer(strategy="mean", axis=0)

# Werte imputieren
features_mean_imputed = mean_imputer.fit_transform(features)

# Echte und imputierte Werte vergleichen
print("Echter Wert:", true_value)
print("Imputierter Wert:", features_mean_imputed[0,0])

Echter Wert: 0.8730186114
Imputierter Wert: -3.05837272461
```

Diskussion

Es gibt zwei Hauptstrategien, um fehlende Daten gegen Ersatzwerte auszutauschen, jede mit Stärken und Schwächen. Erstens können wir mit maschinellem Lernen die Werte der fehlenden Daten vorhersagen. Dazu behandeln wir das Merkmal mit fehlenden Werten als Zielvektor und verwenden die verbleibende Teilmenge der Merkmale, um fehlende Werte vorherzusagen. Uns steht eine umfangreiche

Palette von Algorithmen für maschinelles Lernen zur Verfügung, um Werte zuzuschreiben, doch eine beliebte Wahl ist KNN. Kapitel 14 befasst sich ausführlicher mit KNN, hier sei nur kurz erwähnt, dass der Algorithmus anhand der *k* nächsten Beobachtungen (die nach einer Abstandsmetrik bestimmt werden) den fehlenden Wert vorhersagt. In unserer Lösung haben wir den fehlenden Wert anhand der fünf nächsten Beobachtungen bestimmt.

Allerdings hat KNN den Nachteil, dass die Abstände zwischen dem fehlenden Wert und jeder einzelnen Beobachtung berechnet werden müssen, um festzustellen, welche Beobachtungen dem fehlenden Wert am nächsten liegen. Bei kleineren Stichproben ist dies noch praktikabel, doch es wird problematisch, sobald eine Stichprobe Millionen von Beobachtungen umfasst.

Eine alternative und besser skalierbare Strategie füllt sämtliche fehlenden Werte mit einem Durchschnittswert. So haben wir in unserer Lösung `scikit-learn` verwendet, um die fehlenden Werte mit dem Mittelwert eines Merkmals zu füllen. Der zugeschriebene Wert liegt oftmals nicht so nahe am wahren Wert wie bei Verwendung von KNN, aber wir können das Auffüllen mit dem Mittelwert ohne Weiteres auf Daten mit Millionen von Beobachtungen skalieren.

Wenn Sie fehlende Werte ersetzen, sollten Sie ein binäres Merkmal einrichten und damit anzeigen, ob die Beobachtung imputierte Werte enthält oder nicht.

Siehe auch

- Eine Studie zu *k*-nächsten-Nachbarn als Imputationsmethode (*http://bit.ly/2HS9sAT*)