HANSER


# Leseprobe

zum

# „Maschinellen Lernen“

von Jörg Frochte

ISBN (Buch): 978-3-446-45291-6
ISBN (E-Book): 978-3-446-45705-8

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-45291-6
sowie im Buchhandel

# Inhalt

# 11 Clustering-Verfahren

Clustering-Verfahren sind Verfahren für unüberwachtes Lernen. Das einzige andere Verfahren, das in diese Kategorie der unüberwachten Verfahren fällt, das wir besprochen haben, ist die PCA in Abschnitt 9.3, welche zur Merkmalsreduktion eingesetzt wird. Dass Clustering-Verfahren unüberwacht sind bedeutet, dass im Unterschied zur Klassifikation keine Zielwerte vorliegen. Während wir z. B. bei den Schwertlilien-Daten für alle 150 Datenbankeinträge wissen, um welche Art von Schwertlilien es sich handelt, haben wir dieses Wissen bei einem unüberwachten Verfahren nicht. Diese Spalte mit Daten würde hier fehlen und die Aufgabe unseres Algorithmus ist es dann, die Pflanzen danach zu gruppieren, welche Einträge sich am ähnlichsten sind.

Clusteranalyse ist somit die Einteilung einer Menge von Objekten in Gruppen, wobei wir folgende Ziele verfolgen:

- Wir wollen die Ähnlichkeit innerhalb der Gruppen maximieren.
- Wir wollen die Unterschiede zwischen den Gruppen maximieren, bzw. ihre Ähnlichkeit minimieren.

Weit präsenter als die Gruppierung von Pflanzen oder Tieren an Hand von Merkmalen ist den meisten Lesern sicherlich das Erlebnis im Bereich des Online-Handels. Wer ist nicht schon mal mit der Meldung konfrontiert worden: „Kunden, die diesen Artikel gekauft haben, kauften auch …“. Es geht also oft um die Identifikation gleichartiger Käufergruppen- und Interessen. Vergleichbare Anwendungen gibt es auch in anderen Beziehungen wie Business-to-Business. Darüber hinaus haben die Verfahren ein Einsatzgebiet im Bereich des Information-Retrieval.

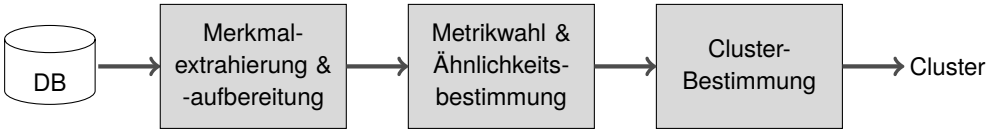


**Abbildung 11.1** Grundablauf einer Clusteranalyse

Der grundsätzliche Ablauf einer Clusteranalyse besteht aus drei Schritten, die in der Abbildung 11.1 skizziert sind. Das erste ist wie immer die Merkmalextrahierung und -aufbereitung. Viele Dinge sind natürlich ähnlich wie schon in Kapitel 9 besprochen. Hierbei gibt es einen Unterschied zu den meisten überwachten Lernverfahren, die wir bis jetzt diskutiert haben. Bei der Analyse von Ähnlichkeiten ist es wie bei bereits besprochenen überwachten Verfahren wichtig, dass die verwendeten Einheiten in den Merkmalen das Verfahren nicht negativ beeinflussen. Nehmen wir einmal an, wir hätten bei dem bekannten Beispiel mit den Schwertlilien die Kelchblattlänge wie bisher in Zentimetern angegeben, die Kronblattlänge jedoch in Millimetern. Die meisten Algorithmen würden dann die Unterschiede bzgl. der Kronblätter völlig überbewerten und den Kelchblättern zu wenig Aufmerksamkeit schenken. Das ist im Wesentlichen das Problem, das wir bisher durch Skalierung oder Standardisierung gelöst haben und

auch in diesen Fall angehen müssten. Wichtiger für das Gelingen einer Clusteranalyse ist jedoch oft eher der Abstand der Elemente in einem Cluster zueinander. Nehmen wir an, es geht um die Flügelspannweite von Vögeln. Bei Wanderalbatrossen sind wir hier im Bereich von 350 Zentimetern und bei Tannenmeisen von eher 20 Zentimetern. Dazwischen liegt eine Zehnerpotenz, jedoch ist es wichtiger wie die Abstände innerhalb der Tannenmeisen aussehen, und dort wird oft ein Bereich von 17 bis 21 cm genannt. Der Spatz hingegen hat in der Regel eine etwas größere Spannweite um 23 cm. Wenn wir nun die Flügelspannweite auf das Intervall $[0,1]$ bringen würden ... was würde das für die Unterscheidbarkeit zwischen den kleinen Vogelarten bedeuten? Pauschal ist das schwer zu sagen und daher ist Domainwissen hier oft hilfreich, da man nur mit einer Vorerwartung sagen kann, ob man durch Standardisierung, Normierung etc. vielleicht in den Daten eher Unterschiede verwischt. In der Praxis liegt man allerdings mit einer Standardisierung der Daten oft richtig, jedoch sollte man die Möglichkeit hier Nachteile zu schaffen kritisch im Hinterkopf haben.

Es gibt natürlich sehr viele Clusterverfahren, die sich in unterschiedliche Einheiten gruppieren lassen. Die Abbildung 11.2 zeigt eine von Benno Stein vorgenommene Einteilungen von

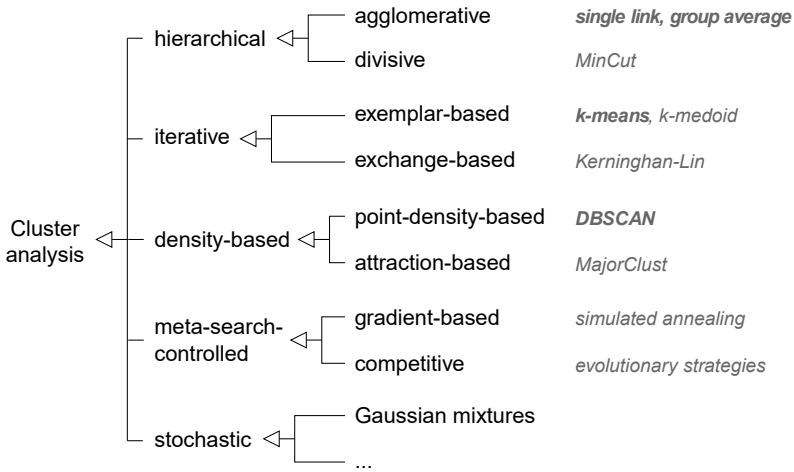


**Abbildung 11.2** Taxonomie von Clusteralgorithmen

Cluster-Verfahren aus seinen Veranstaltungen [Ste18]. Wir werden aus den drei populärsten Gruppen, den hierarchischen, den iterativen und den dichte-basierten Verfahren, jeweils einen Clusteralgorithmus vorstellen. Dabei konzentrieren wir uns wieder jeweils auf recht populäre Verfahren wie den DBSCAN und k-Means.

Für unsere Cluster-Verfahren werden wir 4 Testfälle im Zweidimensionalen betrachten, da sie sich gut visualisieren lassen.

Die ersten beiden Fälle bestehen aus vier Bällen, die unterschiedlich viele Elemente beinhalten können. Das nächste ist eine Datenmenge, die aus drei Bällen besteht, zwischen denen es einen dünner besetzten Übergangsbereich gibt. Solche Datensätze werden oft als **Mouse-Dataset** bezeichnet. Nun folgen die schon aus Abschnitt 9.4 bekannten zwei Halbmonde und als letzte Datenmenge mit Struktur kommen zwei ineinanderliegende Kreise. Die sechste Menge von Daten hat gar keine Struktur und soll zeigen, was passiert, wenn die Verfahren auf Rauschen angewendet werden. Wichtig ist, sich später vor Augen zu halten, dass die Art, wie

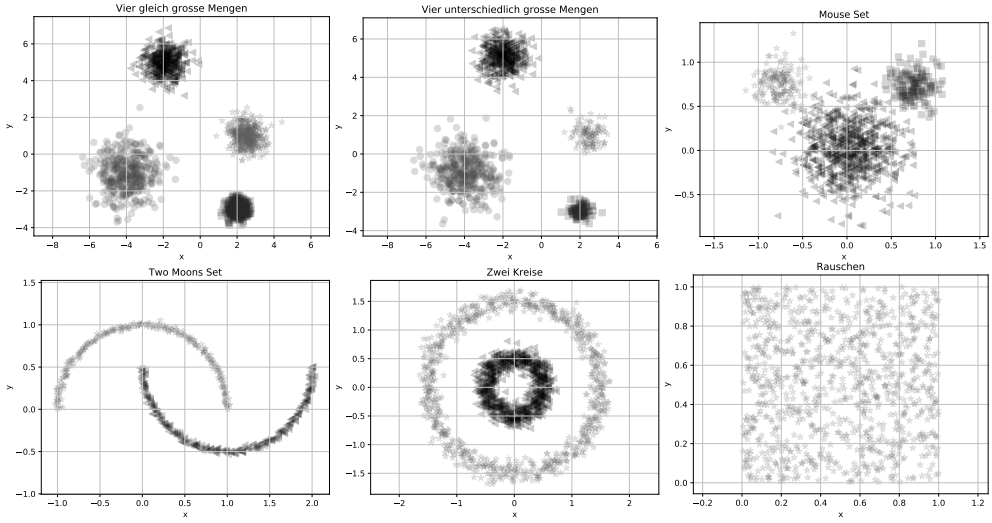


**Abbildung 11.3** Testbeispiele für die Clusteralgorithmen

die Daten erzeugt wurden, nicht automatisch die einzige mögliche Art ist, diese als Cluster zu gruppieren. Als Menschen sehen wir Dinge anders als sie u. U. sind. Nehmen wir einmal Max Wertheimer, einen der Begründer der Gestaltpsychologie, und seine 1923 formulierten sechs wesentlichen Faktoren für Zusammenhänge in der Wahrnehmung als Grundlage. Menschen bevorzugen bei dem Beispiel mit den zwei Monden eben diese Zuordnung in zwei Halbkreise, da diese u. a. dem Gesetz der Nähe und der guten Gestalt entsprechen. Das ist jedoch nicht die einzige Gruppenbildung, die denkbar ist. Immerhin haben die beiden Enden der Bögen kaum etwas gemeinsam und liegen deutlich näher an Elementen des jeweils anderen Bogens. Ähnliches gilt für die beiden Kreise. Durch Farbe und Struktur käme kaum jemand auf die Idee, andere Gruppen als zwei Kreise zu bilden, aber vielleicht könnte man ja auch an der y-Achse die Mengen teilen und so zwei Gruppen erzeugen? Wir werden also gleich kritisch hinterfragen, warum uns ein Algorithmus welches Ergebnis zurückliefert. Manchmal ist es eine Schwäche der Methode, wenn das Ergebnis nicht unseren Erwartungen entspricht, jedoch ist unsere Erwartung auch manchmal zu stark von unseren Anlagen geprägt und rein formal nicht die einzige sinnvolle Lösung.

Die Quelltexte, um diese Beispiele zu erhalten, sind unten angeben. Die Codezeilen zum Plotten der Daten sind nur für den ersten Fall angegeben, da diese sich in leichten Variationen immer wiederholen:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def bubbleSetNormal(mx,my,number,s):
5      x = np.random.normal(0, s, number) + mx
6      y = np.random.normal(0, s, number) + my
7      return(x,y)
8
9  def fourBalls(n1,n2,n3,n4):
10     np.random.seed(42)
11     dataset = np.zeros( (n1+n2+n3+n4,2) )
```

```
    (dataset[0:n1,0],dataset[0:n1,1])         = bubbleSetNormal( 2.5, 1.0,n1,0.5)
    (dataset[n1:n1+n2,0],dataset[n1:n1+n2,1]) = bubbleSetNormal( 2.0,-3.0,n2,0.3)
    (dataset[n1+n2:n1+n2+n3,0],dataset[n1+n2:n1+n2+n3,1]) = bubbleSetNormal(-2.0, 5.0,n3,0.6)
    (dataset[n1+n2+n3:n1+n2+n3+n4,0],dataset[n1+n2+n3:n1+n2+n3+n4,1]) = bubbleSetNormal
     (-4.0,-1.0,n4,0.9)
    return (dataset)

plt.close('all')
n1=n2=n3=n4=400
# Testbeispiel 1
XBalls = fourBalls(n1,n2,n3,n4)

fig = plt.figure(1)
ax = fig.add_subplot(1,1,1)
ax.scatter(XBalls[0:n1,0]  ,XBalls[0:n1,1],c='red',s=60,alpha=0.2,marker='*')
ax.scatter(XBalls[n1:n1+n2,0],XBalls[n1:n1+n2,1],c='blue',s=60,alpha=0.2,marker='s')
ax.scatter(XBalls[n1+n2:n1+n2+n3,0],XBalls[n1+n2:n1+n2+n3,1],c='black',s=60,alpha=0.2,marker='
    <')
ax.scatter(XBalls[n1+n2+n3:n1+n2+n3+n4,0],XBalls[n1+n2+n3:n1+n2+n3+n4,1],c='green',s=60,alpha
    =0.2,marker='o')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.grid(True,linestyle='-',color='0.75')
ax.set_aspect('equal', 'datalim')
ax.set_title("Vier gleich grosse Mengen")

n1=n2=100
n3=n4=400
# Testbeispiel 2
XBalls2 = fourBalls(n1,n2,n3,n4)
```

Nun folgt der das *Mouse Dataset*. Es gibt da im Netz und in der Literatur unterschiedliche Varianten von. Teilweise gehen die Ohren noch stärker in den Kopf über. Wenn sie an dem Fall interessiert sind, können Sie das durch die Parameter in *bubbleSetNormal* leicht erreichen.

```
def mouseShape():
    np.random.seed(42)
    dataset = np.zeros( (1000,2) )
    (dataset[0:150,0],dataset[0:150,1])       = bubbleSetNormal(-0.75, 0.75,150,0.15)
    (dataset[150:300,0],dataset[150:300,1])   = bubbleSetNormal( 0.75, 0.75,150,0.15)
    (dataset[300:1000,0],dataset[300:1000,1]) = bubbleSetNormal( 0, 0,700,0.29)
    return (dataset)
# Testbeispiel 3
XBMouse = mouseShape()
```

Noch einmal der bekannte Fall mit den beiden Halbmoden, hier jedoch für ein Aufgabe beim unüberwachten Lernen.

```
def twoMoonsProblem( SamplesPerMoon=240, pNoise=2):
    np.random.seed(42)
    tMoon0 = np.linspace(0, np.pi, SamplesPerMoon)
    tMoon1 = np.linspace(0, np.pi, SamplesPerMoon)
    Moon0x = np.cos(tMoon0)
    Moon0y = np.sin(tMoon0)
    Moon1x = 1 - np.cos(tMoon1)
    Moon1y = 0.5 - np.sin(tMoon1)
    X = np.vstack((np.append(Moon0x, Moon1x), np.append(Moon0y, Moon1y))).T
    X = X + pNoise/100*np.random.normal(size=X.shape)
```

```
    Y = np.hstack([np.zeros(SamplesPerMoon), np.ones(SamplesPerMoon)])
    return X, Y
# Testbeispiel 4
(XMoons,_) = twoMoonsProblem()
```

Nun zwei die zwei ineinander liegenden Kreise.

```
def circels():
    np.random.seed(42)
    phi = np.linspace(0,2*np.pi, 800)
    x1 = 1.5*np.cos(phi)
    y1 = 1.5*np.sin(phi)
    x2 = 0.5*np.cos(phi)
    y2 = 0.5*np.sin(phi)
    X = np.vstack((np.append(x1,x2), np.append(y1,y2))).T
    X = X + 0.1*np.random.normal(size=X.shape)
    return(X)
# Testbeispiel 5
Xcircels = circels()
```

Abschließend noch das Rauschen im Einheitsquadrat.

```
np.random.seed(42)
# Testbeispiel 6
XRauschen = np.random.random( (1000,2) )
```

Beachten Sie, dass oben zwei unterschiedliche Arten zum Erzeugen von Zufallszahlen verwendet wurden. `np.random.random` erzeugt gleichmäßige Zufallszahlen zwischen 0 und 1, während `np.random.normal` entsprechend einer Normalverteilung diese erzeugt. Große Zufallszahlen sind also unwahrscheinlicher als keine, jedoch nicht ausgeschlossen. Daher enthalten die Mengen auch Einträge, die der eine oder andere Algorithmus als Ausreißer interpretieren wird.

## 11.1 k-Means und k-Means++

Ziel von k-Means ist es, die Elemente $x_j$ des Datenbestandes in $k$ Cluster aufzuteilen. Das Besondere bei dieser Clustertechnik ist, dass es einen Repräsentanten $\mu_i$ pro Cluster $C_i$ gibt. Er ist quasi der Prototypvertreter des Clusters, jedoch im Regelfall kein Datenbankeintrag, sondern ein Mittelwert der Daten des Clusters. Die Frage ist nun, nach welchen Gesichtspunkten die Aufteilung erfolgt. Das Kriterium für die Qualität dieser Aufteilung ist, dass die Summe der Abweichungen von den Cluster-Repräsentanten in der gewählten Metrik minimal ist. Mathematisch entspricht dies der Optimierung der Funktion, hier bzgl. eines Minimums,

$$J = \sum_{i=1}^{k} \sum_{x_j \in S_i} d(x_j, \mu_i)$$

wobei $d(x_j, \mu_i)$ die Distanz in der entsprechenden Metrik ist. Die häufigste Interpretation erfolgt jedoch über den **Lloyd-Algorithmus**, wenn wir statt einer beliebigen Metrik das Quadrat der Euklid-Norm als Grundlage für den Abstand spezialisieren, dann erhalten wir mit

$$d(x_j, \mu_i) = \|x_j - \mu_i\|^2$$

$$J = \sum_{i=1}^{k} \underbrace{\sum_{x_j \in C_i} \|x_j - \mu_i\|^2}_{(C)} . \tag{11.1}$$

Die äußere Summe durchläuft alle Cluster. Die Summe $(C)$ besteht dabei aus dem Abstand aller Elemente des Clusters $C_i$ vom Repräsentanten. $J$ hängt hier nun mit der Methode der kleinsten Fehlerquadrate zusammen und man kann das Ziel im Sinne einer Varianzminimierung interpretieren. Unabhängig davon, welche Metrik man einsetzt, ist der Algorithmus in drei Phasen aufgeteilt.

1. Initialisiere $k$ Repräsentanten $\mu_i$ für die Cluster
2. Ordne jedes Element dem Cluster zu, bei welchem die Distanz zum Repräsentanten des Clusters am kleinsten ist
3. Berechne durch Mittelwertbildung die neuen Repräsentanten $\mu_i$ der Cluster

Wenn der Schritt Nummer 3 abgeschlossen ist, wird wieder zu Nummer 2 gesprungen, bis die Lage der Repräsentanten stabil ist.

Der folgende Pseudocode des Algorithmus lässt sich leicht nach Python aber auch MATLAB oder R überführen. $D$ ist dabei der gesamte Datenbestand, der vorliegt, und $v$ ein Element daraus.

```
1:  t = 0
2:  for i = 1 to k do μ_i = choose(D)
3:  end for
4:  repeat
5:      t = t + 1
6:      for i = 1 to k do C_i = ∅
7:      end for
8:      for all v ∈ D do
9:          i = argmin_{j∈{1..k}} d(μ_i, v)
10:         C_i = C_i ∪ {d}
11:     end for
12:     for i = 1 to k do
13:         μ_i^old = μ_i
14:         μ_i = Mittelwert(C_i)
15:     end for
16:     e = Σ_{i=1}^{k} d(μ_i^old, μ_i)
17: until e < ε OR t > tmax)
```

Die Laufzeit von k-Means ist $O(n \cdot k \cdot r \cdot i)$, wobei $n$ die Anzahl der Einträge in unserer Datenbank $D$ ist. $r$ steht für die Dimension des Vektorraums, also die Anzahl der Merkmale. $k$ ist die Zahl der Cluster, die k-Means finden soll, und $i$ die Anzahl der Iterationen, welche bis zur Konvergenz benötigt werden. Für sinnvolle Einsatzfälle ist $i$ dabei eine eher kleine Zahl im Bereich unter Hundert. Da für eine feste Fragestellung $d$ konstant ist und ebenso die Anzahl der gesuchten Cluster, wird der Algorithmus oft als Methode mit einer linearen Komplexität bezeichnet, da er abhängig von der Datenbankgröße $n$ im Wesentlichen nur linear wächst.

Im Allgemeinen arbeiten Clusteralgorithmen auf einer Gesamtmenge und teilen diese in einzelnen Cluster auf. Kommt ein neues Element in einer Datenbank hinzu, muss die Cluster-

analyse auch erneut durchgeführt werden. Bei k-Means gibt es jedoch die Repräsentanten, die man dazu nutzen kann, eine Vorhersage zu treffen, in welchem Cluster ein neues Element wohl wäre. Hierzu wird die Distanz zu jedem Repräsentanten berechnet und dann die Clusterzuordnung analog zur Zeile 9 in Pseudocode zurück geliefert. Daher können wir hier ausnahmsweise wie bei den überwachten Verfahren die Methoden `fit` und `predict` zur Verfügung stellen.

Wir beginnen mit der Methode, um ein Objekt zu instanziieren. Wir beschränken uns bei unserer Umsetzung darauf, $p$-Normen für die Distanz zu verwenden, wobei der Default die Euklid-Norm mit $p = 2$ ist. Notwendiger Parameter hingegen ist die Anzahl der gewünschten Cluster. Zusätzlich initialisieren wir noch eine Liste, um uns die Veränderung der Repräsentanten während des Algorithmus ansehen zu können.

```
1  import numpy as np
2
3  class kmeans:
4      def __init__(self, noOfClusters ,p=2):
5          self.noOfClusters = noOfClusters
6          self.p = p
7          self.fitHistory = []
```

Bei k-Means geht es darum Abstände von allen Zentren bzw. Repräsentanten der Cluster zu berechnen. Die Methode unten erlaubt dies in einer möglichst vektorisierten Form. Der erste Cluster wird dabei als Spezialfall vor der Schleife berechnet und abschließend alle weiteren Distanzen mittels `vstack` hinzugefügt.

```
8
9      def _computeDistances(self,X,centres):
10         distances = ( np.sum( np.abs(X-centres[0,:])**self.p,axis=1) )**(1/self.p)
11         for j in range(1,self.noOfClusters):
12             distancesAdd = ( np.sum( np.abs(X-centres[j,:])**self.p,axis=1) )**(1/self.p)
13             distances = np.vstack( (distances,distancesAdd) )
14         return(distances)
```

Die `fit`-Methode setzt mit dieser Distanzberechnung den oben angegebenen Pseudocode um.

```
15
16     def fit(self, X, maxIterations=42):
17         Xmin = X.min(axis=0)
18         Xmax = X.max(axis=0)
19         newcentres = np.random.rand(self.noOfClusters,X.shape[1])*(Xmax - Xmin)+Xmin
20         oldCentres = newcentres + 1
21         count = 0
22         while np.sum(np.abs(oldCentres-newcentres))!= 0 and count<maxIterations:
23             count = count + 1
24             oldCentres = np.copy(newcentres)
25             self.fitHistory.append(newcentres.copy())
26             distances =self._computeDistances(X,newcentres)
27             cluster = distances.argmin(axis=0)
28
29             for j in range(self.noOfClusters):
30                 index = np.flatnonzero(cluster == j)
31                 if index.shape[0]>0:
32                     newcentres[j,:] = np.sum(X[index,:],axis=0)/index.shape[0]
33                 else:
34                     distances = ( np.sum( (X-newcentres[j,:])**self.p,axis=1) )**(1/self.p)
35                     i = distances.argmin(axis=0)
```

```
36                      newcentres[j,:] = X[i,:]
37                      cluster[i]=j
38          self.centers = newcentres
39          return(newcentres,cluster)
```

Auffällig hierbei ist der `else`-Zweig von Zeile 34 bis 37. Hierbei geht es um einen unangenehmen Spezialfall, in welchem ein Cluster leer bleibt. Tatsächlich ist es nämlich nicht ausgeschlossen, dass im Laufe einer Iteration einem Repräsentanten alle *Mitglieder* durch andere Repräsentanten abspenstig gemacht werden. Ist dies der Fall, wird der Eintrag in der Datenbank herausgesucht, der am nächsten an diesem Zentrum von Garnichts liegt, und dieser Eintrag wird dem leeren Cluster zugewiesen. Da er nur aus diesem einen Element besteht, wird dann auch der Repräsentant gleich dem einzigen Eintrag gesetzt.

Wie schon erwähnt, können wir für k-Means eine `predict`-Methode bereitstellen:

```
40
41      def predict(self,X):
42          distances =self._computeDistances(X,self.centres)
43          cluster = distances.argmin(axis=0)
44          return cluster
```

Nun testen wir unsere Implementierung auf dem Testfall 1, bei dem wir es mit vier Gruppen ähnlicher Mächtigkeit zu tun haben. Wie man in Abbildung 11.4 sehen kann, konvergiert das

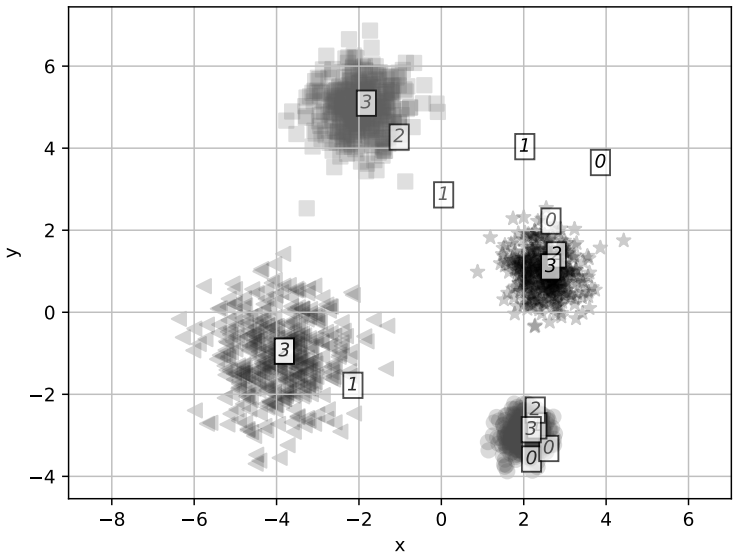


**Abbildung 11.4** Testbeispiel 1 für k-Means

Verfahren sehr schnell. Nach nur vier Iterationen stehen die Repräsentanten und damit auch die einzelnen Cluster fest. Der Algorithmus war dabei auch in der Lage, mit etwas unglücklichen Startwerten umzugehen. Wenn man einmal sucht wo die vier mit 0 angegebenen Startpunkte liegen, so sind zwei nah beieinander in der grünen Menge gestartet. Auch die beiden später zum roten (Dreieck nach unten) und schwarzen (Kreis) Cluster gehörigen Startwerte sind eher eng beieinander. Während sich die nach oben und nach links zeigenden Dreiecke als Gruppen schnell getrennt haben, wurden die Kreise und nach unten zeigenden Dreiecke

zunächst in die gleiche Richtung gezogen und haben sich dann ab 2 deutlich aufgeteilt. Die unterschiedliche Dichte der Gruppen hatte hier keinen Einfluss auf das Clustering.

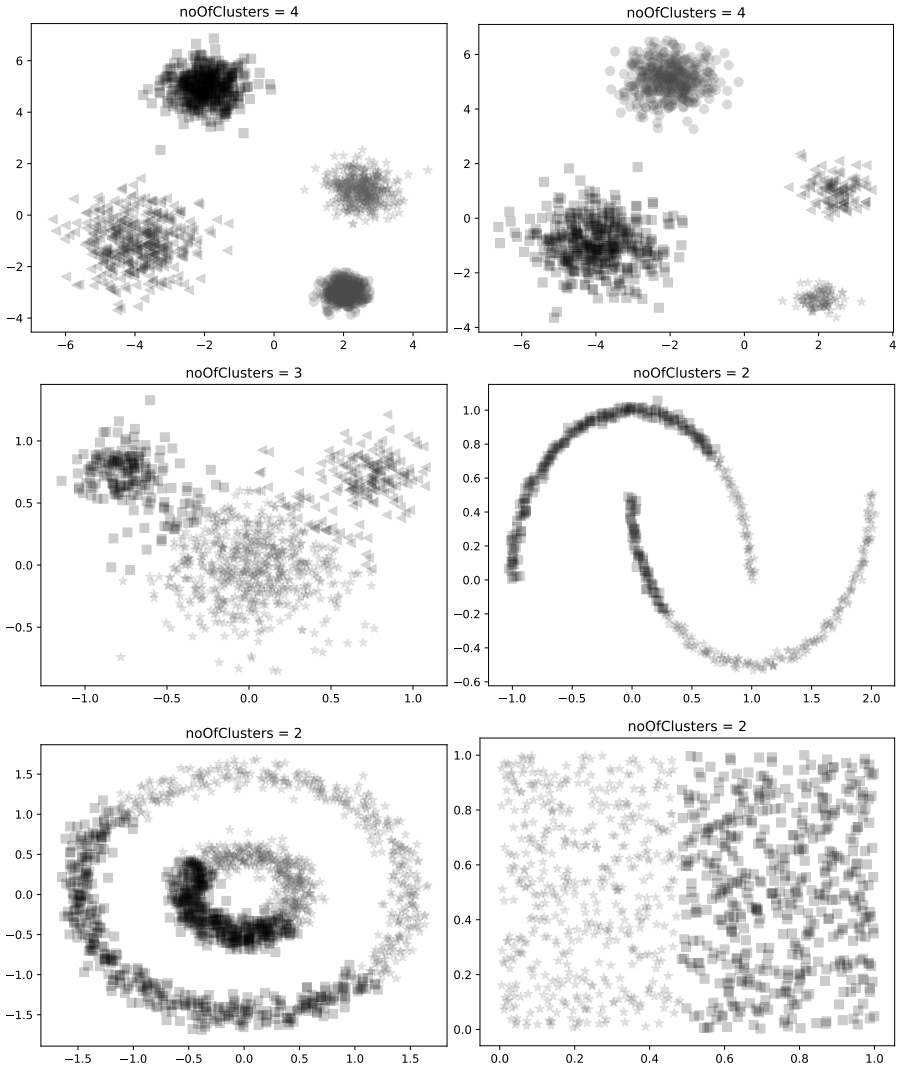


**Abbildung 11.5** Clusterbildung mit k-Means auf den Testproblemen

Die restlichen in Abbildung 11.5 dargestellten Ergebnisse auf den Testproblemen sind auch zufriedenstellend. Beim Mouse-Dataset sieht man, dass k-Means Gruppen gleicher Größe bevorzugt, weshalb die Ohren etwas auf den mittleren Bereich übergreifen und drei etwa gleich große Gruppen erzeugt werden. Die Testfälle 4 und 5 sehen für menschliche Betrachter etwas unglücklich aus, sind jedoch bzgl. der Minimierung des oben erwähnten Funktionals (11.1) stimmig. Eine für Menschen vertrautere Aufteilung erfolgt eher durch dichtebasierte Ansätze, wie wir sie in Abschnitt 11.3 besprechen werden. Wichtig ist jedoch vor allem der Testfall 2. Hier sind zwar die richtigen vier Cluster entstanden, jedoch ist dieses Ergebnis nicht stabil. Bei

unglücklichen Startwerten können auch die beiden rechten Gruppen zusammengelegt werden und eine der beiden größeren Gruppen stattdessen geteilt. Hier ist der Algorithmus also sehr abhängig von den Startwerten.

Um hier robustere Startbedingungen zu schaffen, wurde eine Variante von k-Means, nämlich **kmeans++**, entworfen. Der Unterschied zu k-Means bezieht sich dabei ausschließlich auf die Initialisierung der Repräsentanten. Mit der Menge $D$ aller Datenbankeinträge $x$ lässt sich das Vorgehen in wenigen Punkten zusammenfassen:

1. Wähle einen Repräsentanten $\mu_1 \in D$ mit einer uniformen Wahrscheinlichkeit
2. Berechne für jeden Eintrag $x$ den Abstand $D(x)$ zum nächstgelegenen bereits gewählten Repräsentanten
3. Wähle zufällig einen neuen Datenpunkt als neuen Repräsentanten. Hierbei nutzt man jedoch keine uniforme Wahrscheinlichkeit, sondern gewichtet diese proportional zu $D(x)^2$
4. Kehre zu Schritt 2 zurück, bis $k$ Repräsentanten gewählt wurden
5. Führe nun den bekannten k-Means durch.

Wie man sieht sind hier die Startwerte automatisch Elemente der Datenbank und nicht rein zufällige Werte in dem Bereich, der durch die Datenbank plausible erscheint. Durch den anschließenden Algorithmus werden jedoch wieder Mittelwerte gebildet, weshalb erneut die Repräsentanten in der Regel nicht mit Einträgen aus der Datenbank übereinstimmen werden.

Nutzen Sie k-Means, um das u. a. schon aus Abschnitt 3.5 bekannte *Iris flower data set* in Cluster einzuteilen; natürlich ohne die Zuordnung zu der jeweiligen Klasse zu verwenden. Wie Sie sehen werden, ist es hier nicht möglich, drei Cluster zu bilden, die genau die drei Gruppen von Lilien umfassen. Der Grund ist, dass diese Aufgabe als überwachtes Problem mit Zieldaten recht einfach ist, als unüberwachte Fragestellung jedoch nicht.

## 11.2 Fuzzy-C-Means

Während k-Means zeitlich im Bereich der späten fünfziger und sechziger Jahre aufkam, wurde etwa ein Jahrzehnt später eine Fuzzy-Variante in [Dun73] durch J.C. Dunn vorgestellt und anschließend weiterentwickelt. Dazu muss man sich klar machen, dass der Begriff einer Fuzzymenge erst Mitte der Sechziger durch Lotfi Zadeh [Zad65] geprägt wurde. Die Grundidee ist etwas, was sofort intuitiv einleuchtet. Über die meisten realen Dinge sind Aussagen wie *X ist Y* selten unbestreitbar wahr, sonder eher graduell richtig. Beispielsweise könnten wir nun eine Reihe von Flussnamen aufzählen mit der Aussage *X ist ein langer Fluß*. Bei dem Nil und dem Amazonas ist die Zustimmung vermutlich gesichert. In dem Vorort, in dem ich wohne, fließt ein Bach mit Namen Kerbecke. Ohne eine Suchmaschine nicht auffindbar, wenn man hier nicht wohnt und daher sicherlich bzgl. der Aussage oben zu verneinen. Was machen wir denn nun mit dem Rhein? 1232 km sind gegen die 6852 km des Nil jetzt deutlich kürzer. Komplett verneinen würde jedoch nicht nur Rheinländer verärgern, sondern vermutlich auch damit alle Flüsse in Deutschland disqualifizieren. Die Ausweg ist eben die Fuzzymenge. Während man in der klassischen Logik diese Aussage nur mit Wahr (1) und Falsch (0) beantworten

kann, können wir hier einer Aussage einen Wahrheitswert zwischen 0 und 1 zuweisen. Mit dieser Fuzzy-Aussage zum Wahrheitsgehalt geht auch die Fuzzy-Zugehörigkeit mit einer Menge einher. Der Rhein wird also vielleicht nur mit einem Wahrheitswert von z. B. 0.6 zur Menge der langen Flüsse gehören.

Nehmen wir an, dass wir $n$ Datensätze haben und $C$ Cluster, die wir bilden wollen. Da wir für jeden der $n$ Datensätze somit $C$ Werte benötigen, für die Aussagen in welchem Maße der Datensatz zu einem Cluster zugehörig ist, ergibt sich eine Matrix $W \in \mathbb{R}^{c \times n}$. Diese Matrix enthält Werte von 0 bis 1. Jeder Eintrag $w_{ij}$ gibt somit den Grad an, dem sich der Datensatz $j$ dem Cluster $i$ zugehörig fühlt. Ansonsten bleiben alle Basisideen von k-Means erhalten und es geht uns auch hierbei um eine Minimierung der Funktion

$$J = \sum_{i=1}^{C} \sum_{j=1}^{n} \left(w_{ij}\right)^m d(x_j, \mu_i) \,.$$

Neben der eher kosmetischen Änderung, dass $k$ jetzt $C$ heißt, geht es natürlich um den Faktor $w_{ij}^m$ und seine Interpretation. Zum einen fließt die oben erwähnte Fuzzy-Zugehörigkeit so in das Funktional ein. Zum anderen geht es aber um die Rolle des Parameters $m \geq 1$. Dieser oft **fuzzifier** genannte Parameter verändert, wie scharf die Zugehörigkeit zu Clustern gewertet wird. Für $m = 1$ geht die Zuordnung unverändert ein, je größer $m$ wird, desto stärker werden jedoch Werte kleiner Eins reduziert. Ein großes $m$ führt also zu kleineren Werten in der Matrix $W$, was einer gleichmäßigeren Zugehörigkeit zu den Clustern entspricht und deshalb zu unschärferen Clustern führt. Wählt man $m = 1$, erhält man nachdem der Algorithmus konvergiert ist, scharfe Mitgliedschaften wie schon beim k-Means. Werte größer als 3 sind unüblich und wenig erfolgversprechend. Sollte kein spezieller Grund durch Expertenwissen vorliegen, wird daher in der Regel die Mitte, also $m = 2$, als Ansatz gewählt.

Mit einem festen $m$ ist der Algorithmus zum Auffinden des Minimums nun wieder sehr analog zu dem von k-Means. Wir erhalten nur eine weitere Nebenbedingung. Bei k-Means hatten wir nur die Bedingung: Die Cluster sind nicht-leer.

Im Fuzzy-Ansatz ist das noch leichter zu erfüllen. Es genügt zu fordern:

$$\sum_{j=1}^{n} w_{ij} > 0 \text{ für alle } i = 1 \ldots C$$

Die neue Nebenbedingung ist, dass die Summe der Zugehörigkeiten zu den Clustern für jeden Datensatz 1 ist, das bedeutet

$$\sum_{i=1}^{C} w_{ij} = 1 \text{ für alle } j = 1 \ldots n \,. \tag{11.2}$$

Man könnte sagen, die eine Bedingung bezieht sich auf die Zeilen unserer Matrix und die andere auf die Spalten.

Zur Lösung des Minimierungsproblems gehen wir nun wie folgt vor:

1. Initialisiere $k$ Repräsentanten $\mu_i$ für die Cluster
2. Berechne für jedes Element bzgl. jedes Clusters ein Maß der Zugehörigkeit mittels:

$$w_{ij} = \frac{1}{\sum_{k=1}^{C} \left( \frac{\|x_i - \mu_j\|}{\|x_i - \mu_k\|} \right)^{\frac{2}{m-1}}} = \frac{1}{\|x_i - \mu_j\|^{\frac{2}{m-1}} \cdot \sum_{k=1}^{C} \|x_i - \mu_k\|^{\frac{-2}{m-1}}} \tag{11.3}$$

3. Berechne durch gewichtete Mittelwertbildung die neuen Repräsentanten $\mu_i$ der Cluster

$$\mu_i = \sum_{j=1}^{n} \underbrace{\frac{(w_{ij})^m}{\sum_{j=1}^{n}(w_{ij})^m}}_{=\omega_j} x_j = \frac{1}{\sum_{j=1}^{n}(w_{ij})^m} \sum_{j=1}^{n} (w_{ij})^m x_j \tag{11.4}$$

Die Gewichte in der Gleichung (11.3) findet man in der Literatur üblicherweise in der linken Formulierung. Die anschließende Umformung ist weniger intuitiv, eignet sich jedoch für eine vektorisierte Umsetzung in Python besser.

Jedoch ist schon die linke Formulierung der Gewichte vielleicht nicht direkt intuitiv zugänglich und man fragt sich beim Lesen: was passiert hier eigentlich? Man nimmt den Abstand zu einem Zentrum $\mu_j$ und teilt diesen durch die Summe der Abstände zu allen Zentren, um sicherzustellen, dass die Summe aller dieser Gewichte eben entsprechend (11.2) 1 ergibt.

Schauen wir uns zum besseren Verständnis zwei Beispiele an, beide mit $m = 2$ und drei Clusterzentren. Stellen wir uns ein Gewicht vor, bei dem der Abstand zu $\mu_j$ sehr klein ist, also $\varepsilon$, und der Abstand zu den beiden anderen jeweils 1. Was erhalten wir dann als Wert?

$$\frac{1}{\left(\frac{\varepsilon}{\varepsilon}\right)^2 + \left(\frac{\varepsilon}{1}\right)^2 + \left(\frac{\varepsilon}{1}\right)^2} \approx 1$$

Das stimmt mit der Intuition überein. Wenn der Datensatz fast direkt neben dem Zentrum liegt, sollte das Gewicht auch fast 1 sein. Nun noch der Fall, in dem der Datensatz zu allen Zentren den gleichen Abstand, sagen wir 1/2 hat. Hier ergibt sich mit

$$\frac{1}{\left(\frac{1/2}{1/2}\right)^2 + \left(\frac{1/2}{1/2}\right)^2 + \left(\frac{1/2}{1/2}\right)^2} = \frac{1}{3}$$

wie gewünscht eine Gleichverteilung für alle Zentren.

Die Gleichung (11.4) folgt demselben Muster. Die Zentren ergeben sich als gewichtete Summe der Datensätze. Die Gewichte bestehen hier aus den Maßen für die Zugehörigkeit, also $w_{ij}$ und der Summe aller dieser Werte für diesen konkreten Cluster. Durch die Konstruktion entsteht auch wieder der Effekt, dass die Summe aller $\omega_j$ für einen Cluster 1 ergibt. Die rechte Formulierung von (11.4) ist wieder etwas schöner für die vektorisierte Umsetzung.

Mit dieser Umsetzung beginnen wir nun auch direkt, wobei die ersten 21 Zeilen identisch mit der k-Means Implementierung von Seite 310 sind. Lediglich der Klassenname muss durch `fuzzyCmeans` ersetzt und in Zeile 16 eine neue Option hinzugefügt werden.

```
16      def fit(self, X, maxIterations=42, vareps=10**-3):
```

Die wesentlichen Änderungen liegen in der `fit`-Methode. Hier starten wir mit Zeile 22 und ändern die Bedingung bzgl. des Abbruchs von 0 auf den neuen Parameter `vareps`. Eine entsprechende Anpassung lohnt sich auch ggf. für den k-Means, hier ist es jedoch wesentlich weil sonst zu viele Iterationen erfolgen.

```
22          while np.sum(np.abs(oldCentres-newcentres)) > vareps and count<maxIterations:
23              count = count + 1
24              oldCentres = np.copy(newcentres)
25              self.fitHistory.append(newcentres.copy())
```

Ab Zeile 26 setzen wir nun die Gleichungen (11.3) und (11.4) um. Hierzu berechnen wir wie zuvor erneut die Abstände und erlauben wieder nur p-Normen bzw. -Metriken. Darüber hinaus sind die Umsetzungen der Formeln auf den häufigsten Fall von $m = 2$ spezialisiert. Wie man im Listing unten sieht, können alle Berechnungen vektorisiert erfolgen. Wegen $m = 2$ haben wir es überall mit den Quadraten des Abstandes zu tun, weshalb wir den Output unserer Methode in Zeile 28 zunächst quadrieren. Anschließend berechnen wir in Zeile 29 die Summe aus (11.3). In Zeile 30 wird dann der Nenner von (11.3) gebildet und in Zeile 31 das Reziproke. Damit haben wir die Gewichte aus (11.3) fertig berechnet und können zu (11.4) übergehen. In Zeile bilden wir die benötigte Summe der Quadrate der Gewichte.

```
26
27              distances =self._computeDistances(X,newcentres)
28              d2 = distances**2
29              d2Sum = np.sum(1/d2,axis=0)
30              W = d2*d2Sum
31              W = 1/W
32              WSum = np.sum(W**2,axis=1)
```

Nun ist es unser Ziel, die $\omega_j$, die wir für jedes $\mu_i$ berechnen, in einer einzigen $i \times j$-Matrix zu speichern, um so effektiver die Zentren zu berechnen. Dazu teilen wir die Matrix der Quadrate der Gewichte $W$ durch die zuvor berechnete Summe. Damit die Dimensionen passen, muss transponiert werden. Anschließend kann jedoch die Verknüpfung von $\omega_{ij}$ mit den Merkmalsvektoren direkt als Matrix-Vektor-Multiplikation durchgeführt werden, wodurch auch Schleifen zur Umsetzung von (11.4) vermieden werden können.

```
33              omega = ((W**2).T/WSum).T
34              newcentres = omega@X
35
36          self.fitHistory.append(newcentres.copy())
37          self.centers = newcentres
38          return(newcentres,W.T)
```

Am Schluss liefert die Methode die berechneten Zentren $\mu_i$ in der Variablen `newcenters` zurück; darüber hinaus noch die Wahrscheinlichkeit, dass ein Element zu einem dieser Cluster gehört in Form der Matrix $W$. Eine `predict`-Methode auf der Basis dieser Zentren bzw. Repräsentanten kann wieder wie im Fall von k-Means umgesetzt werden.

Nimmt man nun den Fuzzy-C-Means und wendet ihn auf die Testprobleme an, so ergibt sich ein sehr ähnliches Bild wie für den k-Means auf Seite 11.5; jedenfalls wenn man einfach einen Datensatz dem Cluster zuschlägt, für den die größte Wahrscheinlichkeit vorliegt. Mittels

```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.scatter(X[:,0]  ,X[:,1],c=W[:,0],s=60, cmap='binary')
```

können wir uns den Grad der Zugehörigkeit zu einem Cluster visualisieren.

Wie man in Abbildung 11.6 sieht, ist dies nicht einfach eine Frage des Abstandes vom Repräsentanten. Die Ränder der *Ohren* zum Beispiel im ersten Bild sind dunkler. Der Grund ist, dass diese weniger sicher zu den anderen Clustern gehören und dadurch die Möglichkeit steigt, zum zentralen Cluster zu gehören. Der einzelne Punkt unten links hat wiederum zu keinem Cluster eine sehr hohe Zugehörigkeit. Diese Information des Fuzzy-Ansatzes kann man nutzen, um zum Beispiel Randbereiche abzutrennen oder die Gewichte in der späteren Verarbeitung von Daten neu zu bewerten. Werden die Gewichte hingegen nur im Sinn einer Zuordnung

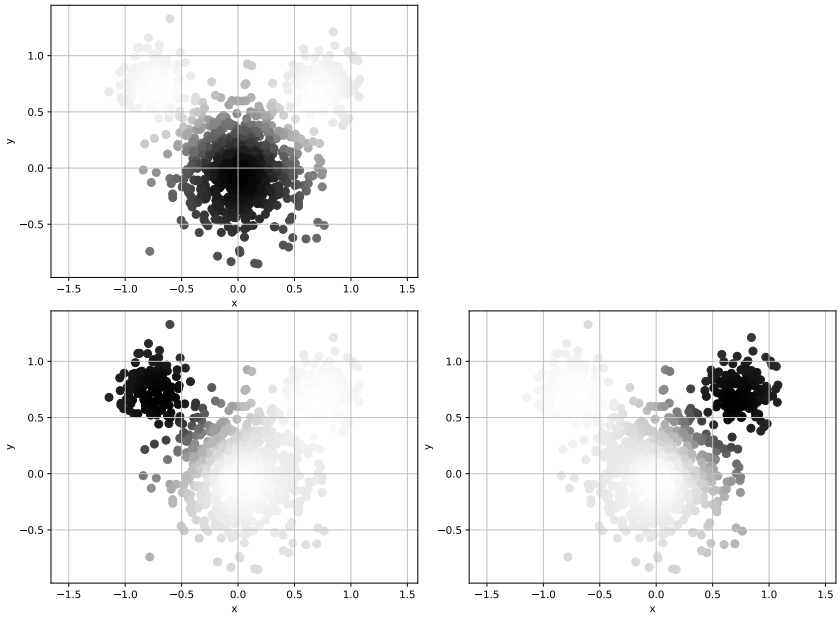


**Abbildung 11.6** Wahrscheinlichkeit, zu einem speziellen Cluster zu gehören

über das Maximum verwendet, ist es in der Regel günstiger, den normalen k-Means zu verwenden. Die Ergebnisse sind in der Regel vergleichbar.

Wir haben im letzten Abschnitt bereits erlebt, dass die Clusteranalyse für das *Iris flower data set* nicht unproblematisch ist. Wie verhält sich die Fuzzy-Variante hier? Visualisieren Sie sich die Gebiete, in denen das Verfahren besonders sichere oder unsichere Zuordnungen macht, mit Hilfe mehrerer 2D- oder 3D-Plots.

## 11.3 Dichte-basierte Cluster-Analyse mit DBSCAN

**DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) gehört zur Klasse der dichte-basierten Clusteralgorithmen. Er wurde 1996 in der Veröffentlichung [EEHJ96] vorgestellt. Später wurde die Grundidee in Formen wie **OPTICS** (Ordering Points To Identify the Clustering Structure) [ABKS99] oder **MajorClust** [SN99] abgewandelt, wobei wir hier nur die Grundform von DBSCAN besprechen. Tatsächlich ist auch diese Grundform diejenige, die man in Bibliotheken und Veröffentlichungen am meisten verwendet findet. Dieser dichte-basierte Algorithmus kann mehrere Cluster erkennen, ohne dass wie bei den k-Means-Varianten zuvor die Anzahl der Cluster bekannt sein muss. Darüber hinaus werden Rauschpunkte im Laufe der Clusteranalyse erkannt, für das Clustering ignoriert und separat zurückgeliefert. Zu den Nach-

# Index