

Aus den Werten im Zeitfenster sollte ein Wert vorhergesagt werden, wobei wir den Wert im nächsten Index mit  $y_{t+1}$  eingezeichnet haben, dieser jedoch auch weiter in der Zukunft liegen kann. Festzuhalten ist, dass zwischen dem ermittelten Wert und den Werten in der Vergangenheit eine funktionale Beziehung besteht.

Die *kausale Prognose* verwendet nicht die Zeitreihe der  $y_i$ , sondern berücksichtigt andere unabhängige Einflüsse, wie zum Beispiel den Wochentag, das Wetter oder die Fußball-WM. Die Einflüsse werden auch als *unabhängige/Einfluss-* bzw. *Erklärungsvariablen* bezeichnet. Das heißt, dass der zu ermittelnde Wert von Einflussvariablen funktional abhängig ist.

Das *kombinierte Prognosemodell*, auch als *dynamische Regression* bezeichnet, nutzt, wie der Name bereits andeutet, beide vorherigen Verfahren für die Prognose.

### Lernalgorithmen

Bekannte Algorithmen für das Supervised Learning sind neben den neuronalen Netzen, die wir auch in die Liste eingefügt haben:

Algorithmus	Einsatzgebiet
Classification Rule Learners	Klassifikation
Decision Trees	Klassifikation
Naive Bayes	Klassifikation
Nearest Neighbor	Klassifikation
Linear Regression	Regression
Model Trees	Regression
Regression Trees	Regression
<i>Neural Networks</i>	<i>Klassifikation/Regression</i>
Support Vector Machines	Klassifikation/Regression

**Tabelle 12.1** Lernalgorithmen überwachtes Lernen

#### 12.1.2 Unüberwachtes Lernen (Unsupervised Learning)

Die Aufgabe beim *unüberwachten Lernen* lautet folgendermaßen: »Entdecke eine interne Repräsentation der Daten nur anhand des Inputs.«

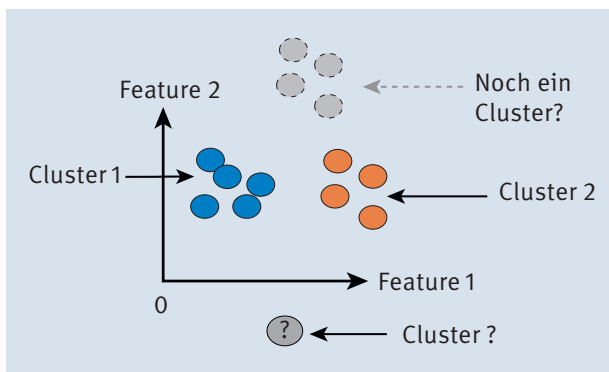
Das heißt, falls keine Output-Kategorien oder -Labels existieren, anhand derer der Algorithmus eine Beziehung herstellen kann, müssen *inhärente*, möglicherweise noch unbekannte Eigenschaften für die Ermittlung von Ähnlichkeiten entdeckt werden.

Für die Daten wird ein (Cluster-)Modell erstellt, ohne tatsächlich zu wissen, wie die einzelnen Cluster zu nennen sind. Trotzdem existieren diese Gruppierungen. Fast so, als ob Sie Tausende Bilder von Stühlen gesehen haben, aber den Begriff für Stuhl noch nicht kennen. Irgendwann werden Sie aus Bequemlichkeit den Begriff »Stuhl« erfinden, um einfacher und effizienter kommunizieren zu können. Sie verwenden dann den Begriff »Stuhl« stellvertretend für ein Element aus der Gruppe der Stuhlobjekte. Also erzeugt diese Lernstrategie eine interne Repräsentation der Daten und kann mit neuen, noch nicht trainierten Daten umgehen. Die Haupttechniken, die beim unüberwachten Lernen eingesetzt werden, sind *Clustering* und *Assoziation*.

## Clustering

Aufgabe der Clustering-Algorithmen ist es, Inputs in Gruppen/Kategorien/Klassen aufzuteilen, nur anhand der Information, die in den Inputs liegt, ohne weitere zusätzliche Information (Abbildung 12.6).

Im Gegensatz zur Klassifikation, bei der die Output-Werte vorgegeben sind, stehen diese Informationen beim Clustering nicht zur Verfügung.



**Abbildung 12.6** Cluster

Clustering wird u. a. für die *Segmentation-Analyse* verwendet, die Gruppen von Individuen mit ähnlichem Verhalten oder demografischen Informationen identifiziert, um Werbekampagnen zielgerichteter abstimmen zu können. Wichtig ist an dieser Stelle festzuhalten, dass die Algorithmen zwar das Clustering durchführen können, aber für die Bewertung und Interpretation dieser Cluster immer noch Menschen herangezogen werden.

### Assoziation

Lernalgorithmen aus dieser Kategorie extrahieren Regeln und Muster aus den Datensätzen, erklären damit die Zusammenhänge zwischen den Variablen und zeigen Frequenzen (Wiederholungen) und Muster. Diese Regeln wiederum ermöglichen Organisationen tiefere Einblicke in ihre Datenstände. Ein einfaches Beispiel wäre die Analyse von Warenkörben und Ableitungen aus den Zusammensetzungen, wie zum Beispiel:

$\{\text{Käse}, \text{Wurst}\} \Rightarrow \{\text{Brot}\}$

Dies kann gelesen werden als: Wenn Kunden Käse und Wurst kaufen, dann kaufen sie auch Brot.

### Lernalgorithmen

Bekannte Algorithmen für Unsupervised Learning, neben den neuronalen Netzen, die wir auch in [Tabelle 12.2](#) eingefügt haben, sind:

Algorithmus	Einsatzgebiet
k-means	Clustering
Gaussian Mixture	Clustering
Hierarchical Clustering	Clustering
PCA-Hauptkomponentenanalyse (Dimensionsreduzierung)	Clustering
Hidden Markov Model	Clustering
<i>Neural Networks</i>	<i>Clustering</i>
<i>Hopfield Networks</i>	<i>Assoziation</i>
<i>Self-Organizing Map (SOM)</i>	<i>Clustering</i>
Association Rules	Pattern Discovery

**Tabelle 12.2** Lernalgorithmen für unüberwachtes Lernen

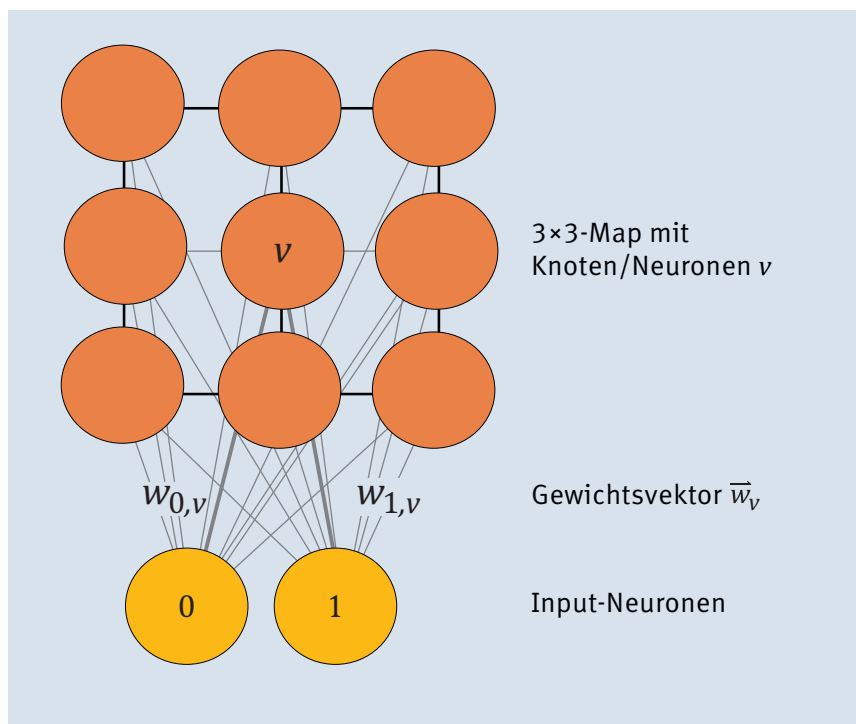
### Anwendungsgebiet

Ein Anwendungsfall für den beschriebenen (deskriptiven) Ansatz von Daten ist die *Pattern Discovery*, die Assoziationen in den Daten entdeckt. Sie wird eingesetzt, um Market-Basket-Analysen durchzuführen, die als Ziel haben festzustellen, welche Produkte gemeinsam gekauft und somit dem Kunden beim Onlinekauf vorgeschlagen oder im Laden anders platziert werden können.

### Beispiel: SOM

Mit Kohonens *Self-Organizing Maps* (SOM) und *Adaptive Resonance Theory* (ART) stehen zwei Netzarten für Unsupervised Learning zur Verfügung. Wir möchten Ihnen nun gerne die SOM näher bringen, um ein wenig praktisch zu werden und diesen vielfältig einsetzbaren Ansatz zu durchleuchten.

Eine SOM hat zwei zentrale Bauteile, den *Input Layer*, so wie wir ihn von den anderen Netzarten kennen, und die *Map*. In dieser Map sind Knoten/Neuronen so wie bei einem Fischernetz miteinander verbunden. Jedes Input-Neuron ist mit jedem Knoten  $v$  in der Map verbunden. Jeder Knoten  $v$  in der Map bekommt von allen Input-Neuronen den Input (Abbildung 12.7).



**Abbildung 12.7** SOM mit zwei Input-Neuronen und einer 3x3-Map

Der Clou ist, dass Inputs, die räumlich in der Nähe sind, auch in der Map auf benachbarte Knoten abgebildet werden. Darum könnte man auch sagen, dass die SOMs eine *topografische Organisation* darstellen, in der nahe Orte in der Map Inputs mit ähnlichen Eigenschaften repräsentieren.

Zum Lernen wird nicht die Fehlerkorrektur, so wie im Back-Propagation, eingesetzt, sondern das sogenannte *Competitive Learning*, das eine Form von Unsupervised Learning darstellt. In dieser Lernstrategie »kämpft« ein Knoten dafür, auf einen Input antworten zu dürfen.

Wie in [Abbildung 12.7](#) angedeutet, besitzt jeder Knoten  $v$  in der Map einen Gewichtsvektor  $\vec{w}_v$ , wobei jeder Knoten mit jedem Input-Neuron verbunden ist. Mit diesen Hintergrundinformationen können wir uns den Lernalgorithmus genauer ansehen.

### Algorithmus

1. Initialisiere die Gewichte der Gewichtsvektoren  $\vec{w}_v$ , zu jedem Knoten  $v$  mit kleinen zufälligen Werten aus dem Intervall  $[0,1]$ .
2. Wiederhole für eine bestimmte Anzahl an Iterationen  $I$ :
  - Wähle einen Input-Vektor  $\vec{x}(s)$  aus der Menge der Trainingsbeispiele  $X$  zufällig aus, wobei  $s$  der Iterationsschritt ist.
  - Berechne die Distanz  $d$  vom Input-Vektor zu den Gewichtsvektoren aller Knoten mit  $d(\vec{x}(s), \vec{w}(s))$ .
  - Bestimme den Knoten  $u$  mit der kleinsten Distanz; dieser wird auch als *Best Matching Unit (BMU)* bezeichnet.
  - Passe den BMU-Gewichtsvektor  $\vec{w}_u(s)$  und den der Knotennachbarn in Richtung des Input-Vektors an. Die Anpassung der Gewichte erfolgt anhand folgender Formel:

$$\vec{w}_v(s+1) = \vec{w}_v(s) + \theta(u, v, s) \cdot \alpha(s) \cdot (\vec{x}(s) - \vec{w}_v(s))$$

Dabei haben die unterschiedlichen Elemente der Formel folgende Bedeutung:

- ▶  $\vec{w}_v(s)$  – der Gewichtsvektor für Knoten mit Index  $v$  zum Iterationsschritt  $s$
- ▶  $s$  – der Iterationsschritt
- ▶  $u$  – der Index für die BMU bezüglich des Input-Vektors  $\vec{x}(s)$
- ▶  $\vec{x}(s)$  – der Input-Vektor
- ▶  $\alpha(s)$  – eine Lernrate, die im Laufe der Zeit abnimmt
- ▶  $\theta(u, v, s)$  – die Nachbarschaftsfunktion zur Ermittlung der Nachbarn mit Index  $v$  für BMU mit Index  $u$  zum Iterationsschritt  $s$  ([Abbildung 12.10](#)). Mit zunehmendem Iterationsschritt  $s$  schrumpft die Nachbarschaft und bekommt Werte im Intervall  $[0,1]$ .

Einige Aspekte müssen wir uns noch genauer ansehen, wie zum Beispiel die Berechnung der *Distanz* zwischen zwei Vektoren, beispielsweise dem Input-Vektor  $\vec{x}(s)$  und einem Gewichtsvektor  $\vec{w}_v(s)$ . Diese ist festgelegt als sogenannter *euklidischer Abstand* mit

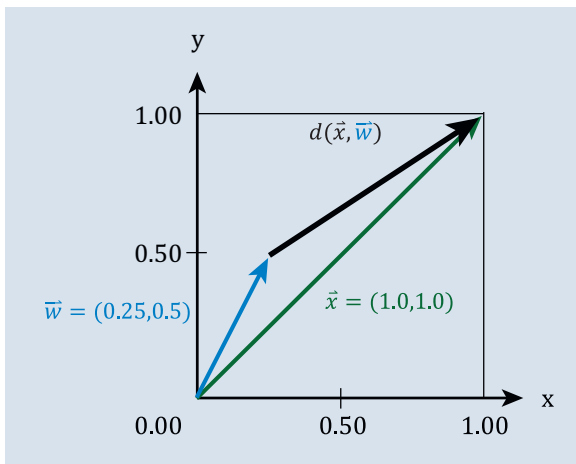
$$\begin{aligned} d(\vec{x}, \vec{w}) &= \|\vec{x} - \vec{w}\| \\ &= \sqrt{(x_1 - w_1)^2 + (x_2 - w_2)^2 + \dots} = \sqrt{\sum_i (x_i - w_i)^2} \end{aligned}$$

So formulieren es die Mathematiker. Importieren Sie das Modul `math`, um die Sache in gut lesbaren Python-Code zu übersetzen:

```
def distance(self, vec1, vec2):
    return math.sqrt((vec1[0]-vec2[0])**2+(vec1[1]-vec2[1])**2)
```

**Listing 12.1** Euklidischer Abstand der zwei zweidimensionalen Vektoren »vec1« und »vec2«

Zeichnerisch können Sie sich den Abstand auch gut vorstellen, indem Sie sich in [Abbildung 12.8](#) die geometrische Darstellung genauer ansehen. Wir haben Beispielwerte für den Abstand eingefügt, um die Berechnung zu üben.



12

**Abbildung 12.8** Euklidische Distanz von zwei Vektoren

Nehmen wir mal an, wir haben die zwei Vektoren  $\vec{w} = (0.25, 0.5)$  und  $\vec{x} = (1.0, 1.0)$ . Wenn wir sie in die Formel für den euklidischen Abstand einsetzen, dann bekommen wir das folgende Ergebnis:

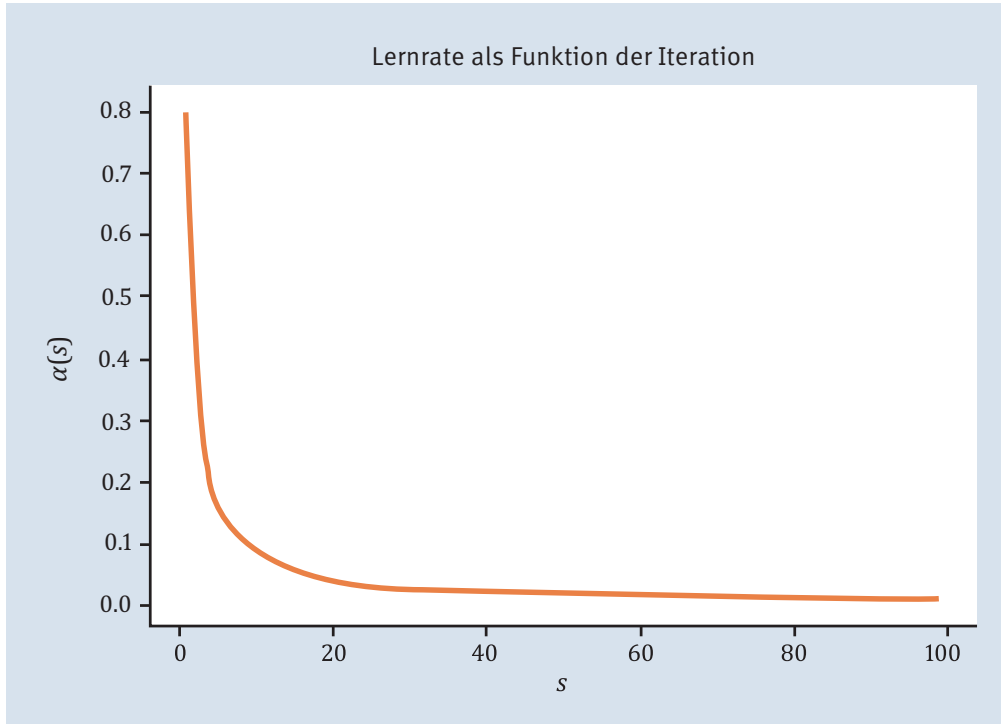
$$\begin{aligned} d(\vec{x}, \vec{w}) &= \|\vec{x} - \vec{w}\| \\ &= \sqrt{(1.0 - 0.25)^2 + (1.0 - 0.5)^2} \\ &= \sqrt{0.5625 + 0.25} \\ &\cong 0.9 \end{aligned}$$

Super, damit können Sie nun den Abstand zwischen zwei Vektoren ausrechnen. Wenn Sie so wollen, ist das die Grundrechenoperation für den SOM-Algorithmus.

Für die Definition der *Lernrate* können unterschiedliche Funktionen verwendet werden. Eine einfach zu implementierende Funktion, die gleichmäßig fällt, kann abhängig vom Iterationsschritt  $s$  und von der initialen Lernrate  $\alpha(0)$  folgendermaßen definiert werden:

$$\alpha(s) = \alpha(0) \cdot \frac{1}{s}$$

Dies entspricht der Forderung, dass am Anfang des Lernens starke Änderungen gewünscht sind und dann im Laufe der Zeit eine Stabilisierung einsetzt ([Abbildung 12.9](#)).



**Abbildung 12.9** Abfall der Lernrate im Laufe der Zunahme der Iterationen

Als Beispiel haben wir die Lernrate  $\alpha(0) = 0.80$  angesetzt und die Anzahl der Iterationen mit 100 festgelegt. Zum Iterationsschritt 10 wäre die Lernrate dann

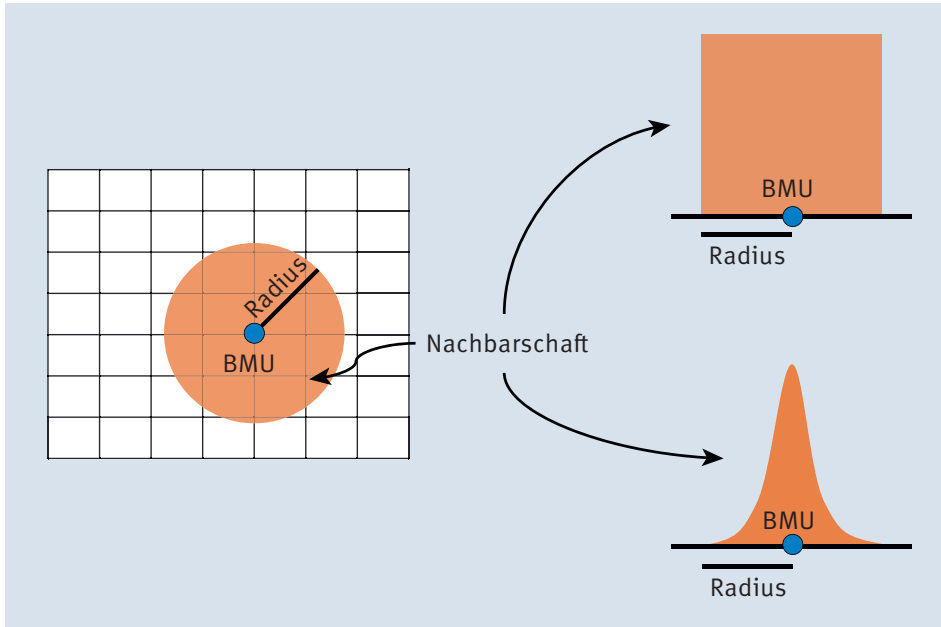
$$\alpha(10) = \frac{\alpha(0)}{10} = \frac{0.80}{10} = 0.08$$

Eine alternative Funktion, die wir auch in der Implementierung für die Lernrate verwenden können, sorgt für einen linearen Abbau während der  $I$  Iterationen:

$$\alpha(s, I) = \alpha(0) \cdot \left(1 - \frac{s}{I}\right)$$

Der nächste Aspekt, den wir betrachten, ist die *Nachbarschaft* eines Vektors. In [Abbildung 12.10](#) sehen Sie auf der linken Seite eine Nachbarschaft, die durch den Radius festgelegt ist. Immer wenn innerhalb des Kreises eine Kreuzung vorkommt, dann bedeutet das, dass ein Knoten aus der SOM in der Nachbarschaft liegt. *Der Radius sollte mit den Iterationen abnehmen*. Auf der rechten Seite sehen Sie zwei unterschiedliche Möglichkeiten, die Entfernung von der BMU zum Nachbarn zu berücksichtigen. Im oberen Fall ist die Bewertung immer gleich, egal wie weit weg der Nachbar und solange er noch inner-

halb des Radius ist. Im unteren Fall wird für die Bewertung eine *Glockenkurve* verwendet, die den Wert für die Entfernung zur BMU nicht immer gleich bewertet.



**Abbildung 12.10** Unterschiedliche Nachbarschaftsfunktionen

Diesen Sachverhalt gießen wir wieder in eine berechenbare Form. Für die Berechnung, die wir für die **Nachbarschaft** zwischen Knoten  $u$  und  $v$  verwenden werden, gilt die folgende hübsche Formel:

$$\theta(u, v, s) = \begin{cases} h_{u,v}(s), & \text{falls } d(u, v) \leq r(s, I) \\ 0, & \text{sonst} \end{cases}$$

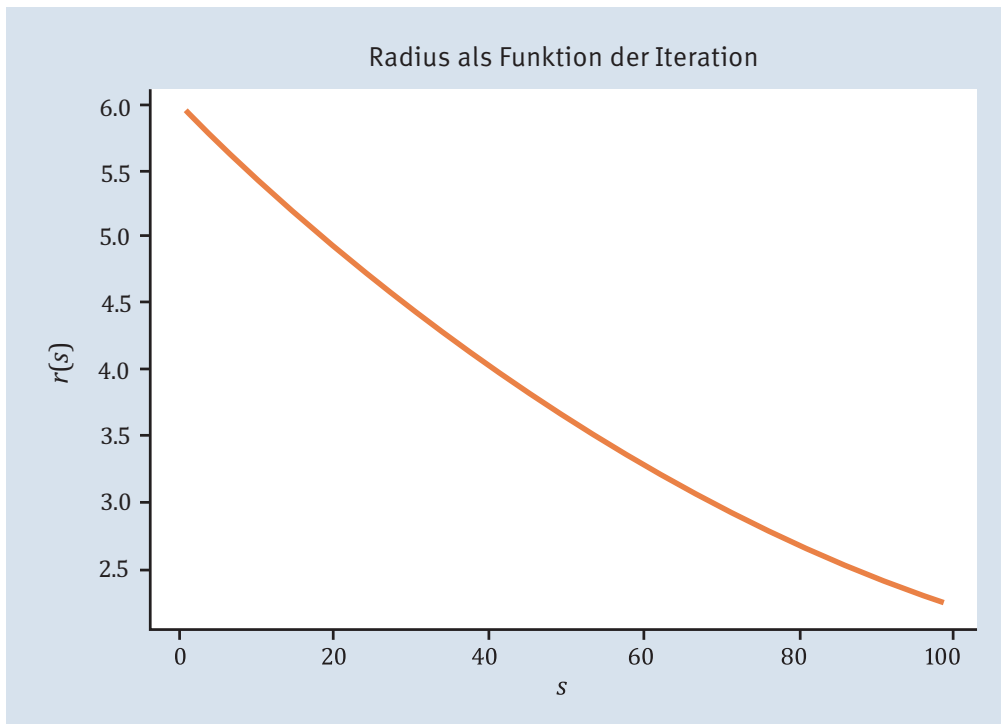
Das heißt, falls ein Knoten mit Index  $v$  innerhalb der Distanz  $r(s)$  der BMU  $u$  liegt, wird die Nachbarschaft mit der Funktion  $h_{u,v}(s)$  zum Zeitpunkt  $s$  bewertet. Für den Wert der Nachbarschaft können Sie eine beliebige Funktion verwenden, die über die Zeit an Wert verliert, damit mit der Zeit die Nachbarschaft kleiner wird, wie zum Beispiel  $\alpha(s)$ , so wie vorher definiert.

Der **Radius**  $r(s, I)$  ist das letzte Element, das wir noch definieren müssen. Auch dafür gibt es natürlich eine Fülle an Möglichkeiten, er sollte aber so definiert werden, dass er mit jedem Iterationsschritt  $s$  immer kleiner wird, wie zum Beispiel:

$$r(s, I) = r(0) \cdot e^{\left(-\frac{s}{I}\right)}$$

In Abbildung 12.11 sehen Sie ein Beispiel für die Entwicklung des Radius mit den Iterationen. Er startet bei 6 und reduziert sich unter 2 im Laufe der 100 Iterationen.





**Abbildung 12.11** Radius als Funktion der Iteration

Bevor wir zur Implementierung schreiten, sehen wir uns noch einmal die Lernformel für die SOM mit dem bisher Gehörten an:

$$\vec{w}_v(s+1) = \vec{w}_v(s) + \underbrace{\theta(u, v, s)}_{\text{kleiner 1}} \cdot \underbrace{\alpha(s)}_{\text{kleiner 1}} \cdot \underbrace{(\vec{x}(s) - \vec{w}_v(s))}_{\text{Vektor zur BMU}}$$

Wir können die Formel nun folgendermaßen verbal zusammenfassen:

### SOM-Lernen

Ändere pro Iterationsschritt das Gewicht eines Map-Knotens in Richtung der BMU.

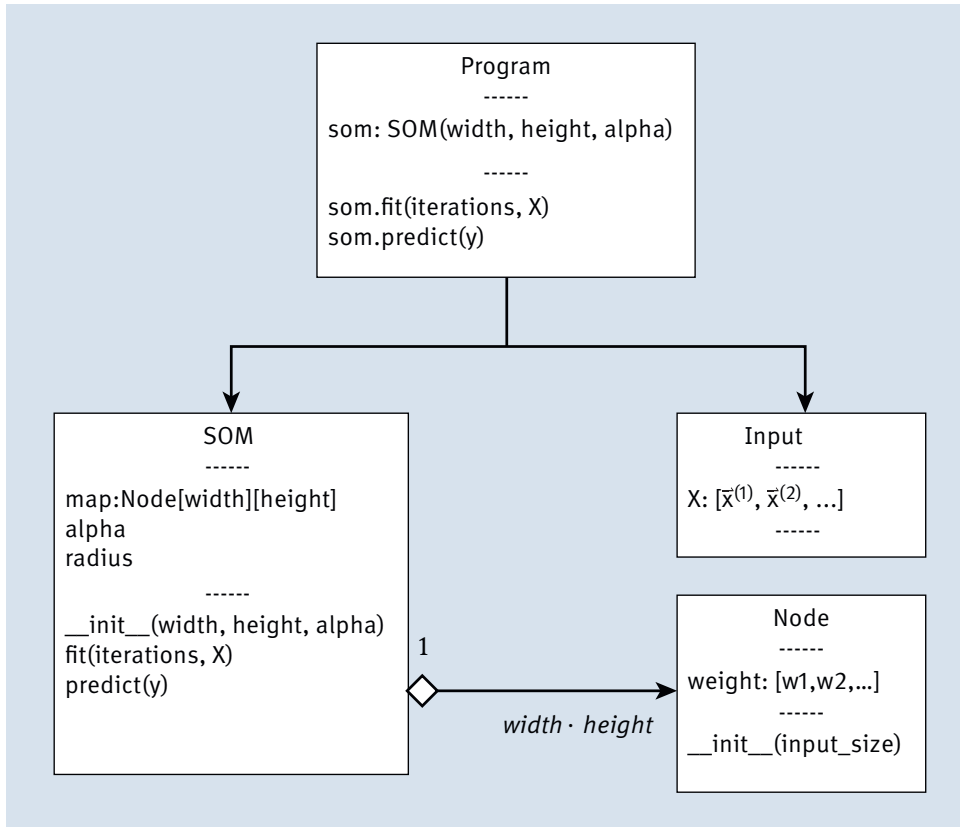
Die Richtung vom Gewichtsvektor zur BMU ergibt sich aufgrund der Differenz der Vektoren  $(\vec{x}(s) - \vec{w}_v(s))$ , die im Anhang B detailliert erläutert wird.

Noch eine Anmerkung: Wenn zum Vektor  $\vec{w}_v(s)$  der Differenzvektor dazugezählt wird, dann zeigt der neue Vektor  $\vec{w}_v(s+1)$  mehr in Richtung Input-Vektor.

### Der Code

Damit sind wir für die Implementierung gerüstet und nehmen uns den Python-Code vor. Natürlich wollen wir wieder die Funktionalitäten sauber verkapseln, legen nicht zu großen Wert auf Performance und wollen auch die Objektorientierung einsetzen. Die

Grundbausteine, die wir in der Einleitung zur SOM besprochen haben, sind: Inputs, Map mit Knoten und Gewichte (Abbildung 12.12).



12

**Abbildung 12.12** Die Bausteine zur SOM und ihre Zusammenhänge

Die Hauptaufgabe erledigt wie immer die Methode `fit()` in der Klasse `SOM`. Dort werden die Gewichtsadjustierungen durchgeführt, also das Lernen implementiert, so wie eingangs beschrieben. Wir haben den Code ausführlich beschrieben und kommentiert, daher sollte das Nachvollziehen nicht allzu schwer fallen. Als Beispiel haben wir wieder unser beliebtes XOR-Problem verwendet.

# Teuvo Kohonens SOM

```

import random
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as col
  
```

```

# Leichter Overkill, da nur Gewichte, aber ausbaufähig
class Node(object):
    def __init__(self, input_size=2):
        self.input_size = input_size
        self.weight = np.array([random.random() \
                                for e in range(self.input_size)])

# Das ist die SOM
class SOM(object):
    def __init__(self, map_width=10, map_height=10, alpha=0.005):
        """Initialisierung der SOM
        """
        self.map_width = map_width
        self.map_height = map_height
        self.radius = 0.6 # Radius
        self.alpha = alpha # Lernrate
        self.map = [[Node() for j in range(self.map_width)] \
                    for i in range(self.map_height)] # Die Map mit den Knoten

    def fit(self, iterations, X):
        """ Competitive Learning für die SOM
        """
        for s in range(1, iterations+1):
            radius_s = self.radius * math.exp(-1.0*s/iterations)
            # Alpha: V1
            # alpha_s = self.alpha / s
            # Alpha: V2
            alpha_s = self.alpha*(1.0 - s/iterations)
            # Zufälligen Input-Vektor auswählen
            x = X[random.randint(0,X.shape[0]-1)]
            distances = np.empty((self.map_width, self.map_height))
            # Alle Distanzen berechnen
            for i in range(self.map_width):
                for j in range(self.map_height):
                    distances[i][j] = self.distance(x, self.map[i][j].weight)
            # Best Matching Unit Index finden
            bmu_index = np.unravel_index(np.argmin(distances, axis=None), \
                                         distances.shape)
            for i in range(self.map_width):
                for j in range(self.map_height):
                    v=self.map[i][j].weight # Gewichtsvektor eines Knotens

```

```

        u=self.map[bmu_index[0]][bmu_index[1]].weight # BMU
        distance=self.distance(u,v) # Distanz BMU und Gewichtsvektor
        if distance <= radius_s:
            neighborhood = radius_s # Nachbarschaftswert
            self.map[i][j].weight += \
                neighborhood * alpha_s * (x - v) # Gewichts Anpassung
    # Weight-Vektoren zeichnen in 2-dim Plot
    self.plot_weights(x)

# Welcher Vektor ist am nächsten?
def predict(self, y):
    distances = np.empty((self.map_width, self.map_height))
    # Alle Distanzen berechnen
    for i in range(self.map_width):
        for j in range(self.map_height):
            distances[i][j] = self.distance(y,self.map[i][j].weight)
    # Knoten mit kleinstem Abstand zwischen Gewichtsvektor und Target y
    min_dist_index = np.unravel_index(np.argmin(distances, axis=None),\
        distances.shape)
    # Rückgabe des Gewichtsvektors
    return self.map[min_dist_index[0]][min_dist_index[1]].weight

# Distanz zwischen zwei Vektoren, Berechnung optimiert
def distance(self, u, v):
    return np.linalg.norm(u-v)

# Gewichte als Plot ausgeben
def plot_weights(self,x):
    # Für Ausgabe
    fig, ax = plt.subplots()
    weights_x = [] # x-Koordinaten
    weights_y = [] # y-Koordinaten
    # Alle Gewichte
    for i in range(self.map_width):
        for j in range(self.map_height):
            weights_x.append(self.map[i][j].weight[0])
            weights_y.append(self.map[i][j].weight[1])
    ax.scatter(weights_x, weights_y, color='b')
    plt.title('Gewichtsvektoren - Iteration ' + str(i))
    plt.xlabel('x')

```

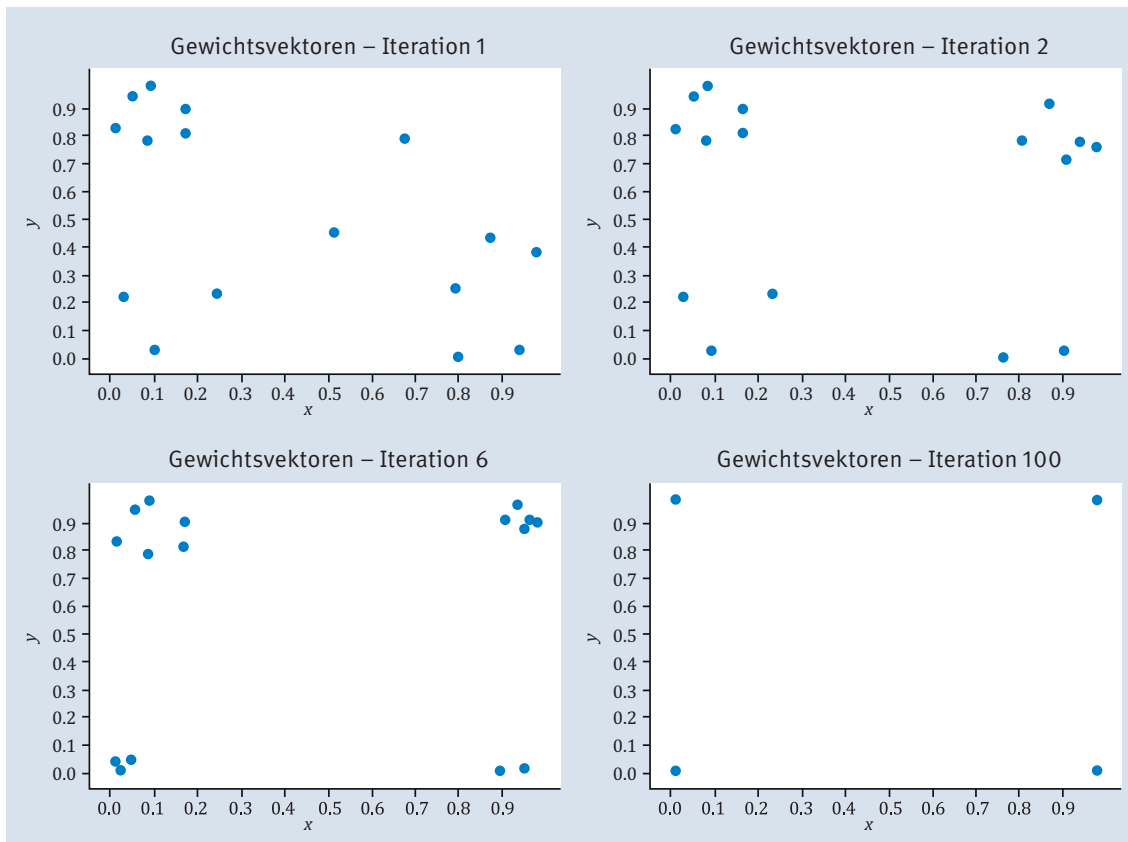
```

plt.ylabel('y')
xticks=np.arange(0,1,0.1)
yticks=np.arange(0,1,0.1)
plt.yticks(yticks)
plt.xticks(xticks)
plt.show()
##### Let the SOM begin #####
#####
# Steuerung des SOM-Trainings und der Auswertung
# Breite, Höhe der Map
map_width = 4
map_height = 4
# Lernrate
alpha = 0.8
# Iterationen
iterations = 100
# Zufall
random.seed(1)
# Die SOM instanziiieren
som=SOM(map_width,map_height,alpha)
# Training
# Initialisierung der Trainingsbeispiele
X=np.array([[0.0,0.0],[0.0,1.0],[1.0,0.0],[1.0,1.0]])
print("Trainieren die XOR-Funktion")
som.fit(iterations,X)
# Auswertung
# Drei Nachkommastellen ausgeben
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})
print("Aussagen zur XOR-Funktion")
print("Aussage 0 0, {}".format(som.predict([0,0])))
print("Aussage 1 0, {}".format(som.predict([1,0])))
print("Aussage 0 1, {}".format(som.predict([0,1])))
print("Aussage 1 1, {}".format(som.predict([1,1])))

```

### Listing 12.2 SOM-Implementierung

Wenn Sie den Code laufen lassen, dann wird in der Methode `fit()` nach jedem Iterationsschritt die Methode `plot_weights()` aufgerufen. Dies sorgt dafür, dass ein Diagramm mit den Gewichtsvektoren gezeichnet wird und wir die Wanderbewegung der Gewichte zu den BMU hin und das Formen von Clustern entdecken können.



**Abbildung 12.13** SOM und die Clusterbildung während des Lernvorgangs

In [Abbildung 12.13](#) haben wir einige der Iterationsschritte herausgenommen, um Ihnen die Entwicklung der Cluster zu zeigen, die letztlich zu einer beeindruckenden Klassifikationsgenauigkeit führt.

# Ausgabe:

```
Aussagen zur XOR-Funktion
Ausgabe 0 0, [0.001 0.000]
Ausgabe 1 0, [0.999 0.000]
Ausgabe 0 1, [0.001 0.999]
Ausgabe 1 1, [0.999 1.000]
```

**Listing 12.3** Das Ergebnis der Auswertung des XOR-Problems

Wir wünschen Ihnen noch viel Spaß beim Anpassen des Codes an Ihre Bedürfnisse und Aufgabenstellungen. Mit obiger Implementierung und der dazu besprochenen Theorie bekommen Sie ein mächtiges Instrument an die Hand, das Sie vielseitig einsetzen können. Wir müssen aber leider weiter und zum nächsten Thema wechseln, *Reinforcement Learning*.