

Maschinelle Mustererkennung Teil 1

Ein einfaches Python-Programm

Von Gilbert Brands

Einführung. Mustererkennung ist die Domäne der KI. Wobei die wenigsten mit dem Begriff KI etwas anfangen können, was zu mancherlei Irritationen führt. Ich möchte mit dieser Reihe versuchen, es interessierten Leuten zu ermöglichen, selbst in der Praxis etwas zu erfahren. In Teil 1 stelle ich ein kleines Programm in der Programmiersprache Python vor, dass ohne Anlernen versucht, in einer Menge von Punkten Muster zu erkennen und die Punkte zu gruppieren.

Python

Python ist eine relativ einfache Programmiersprache auf Interpreterebene, d.h. man schreibt eine Programmzeile und kann sie direkt ausführen (oder nicht, wenn man sich vertan hat). Später wird natürlich auch Python beliebig kompliziert, wenn es ans Eingemachte geht. Python ist zwar nicht meine erste Wahl (das wäre eher C++), aber gerade weil es so einfach ist, eignet es sich für den Einstieg.

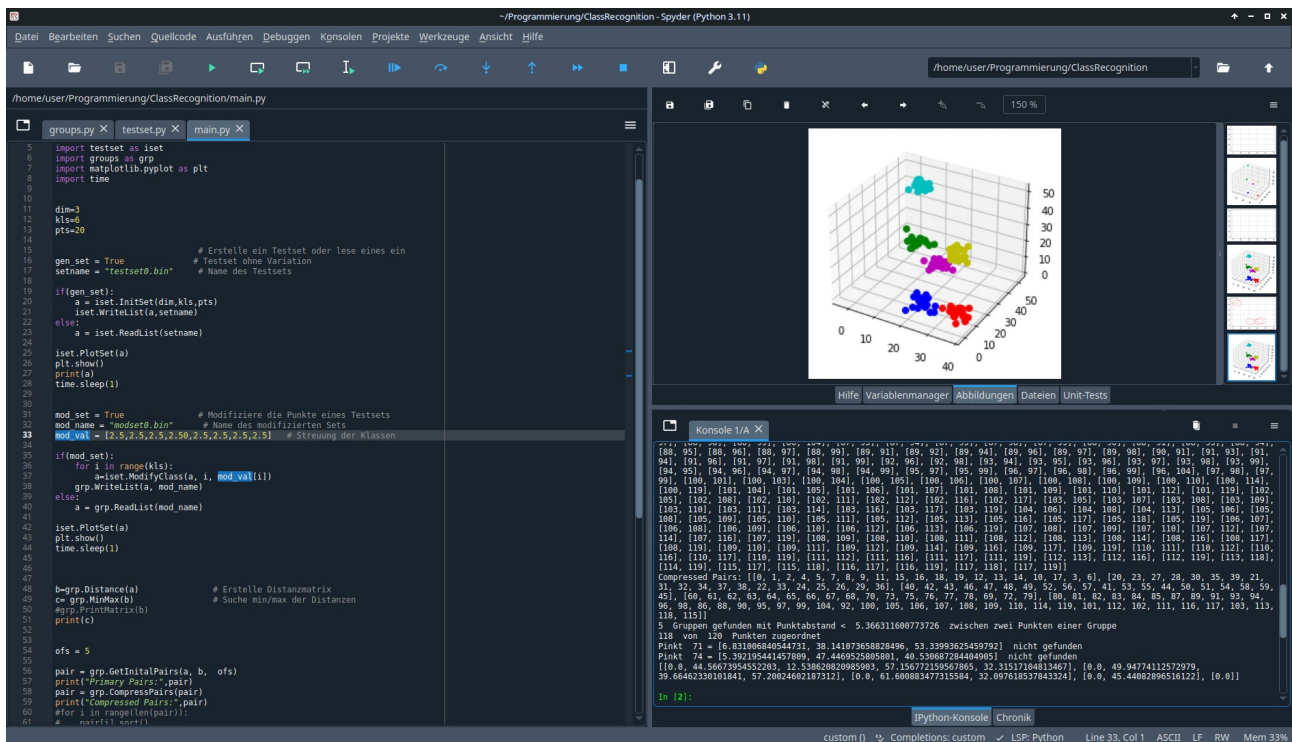
Auf vielen Rechnern ist Python ohnehin installiert, da die eine oder andere Anwendung in Python geschrieben ist. Ansonsten lädt man sich das Python3-Paket herunter und installiert es. Das funktioniert mit allen Betriebssystemen und Anleitungen sind im Internet leicht zu finden. Dazu installiert man sich einen Editor für die Programme, beispielsweise „Spider“ oder die etwas einfachere Version „Geany“. Dann kann es schon losgehen. Kleine Einführungen, wie man das erste Programm hinkommt, gibt es wie Sand am Meer, so dass ich mich nicht weiter darüber auslassen.

WICHTIG! Ihr könnt bei den einfachen Versuchen absolut nichts kaputt machen. Also keine Sorge, dass der Rechner hinterher die Ohren hängen lässt. Wenn er es doch tut, liegt das an etwas anderem.

Zuletzt erstellt ihr einen Ordner, in den ihr die drei kleinen Programme dieses Beispiels kopieren könnt. Öffnen im Editor und schon kann es los gehen. Die Programmdateien heißen

```
main.py  
testset.py  
groups.py
```

Öffnet alle drei, aktiviert das Fenster für „main.py“ und drückt die Taste F5 (im Spider-Editor) und schon sollte es bunt werden, etwa so:



Keine Bange, das sieht komplizierter aus, als es wirklich ist.

Erzeugen eines Testsatzes

In den Köpfen der Programmdateien findet ihr Einträge wie

```
import testset as iset
import groups as grp
import matplotlib.pyplot as plt
import time
```

Damit werden fertige Code-Bibliotheken geladen, die die Hauptarbeit übernehmen. Auch die beiden Programmdateien „testset.py“ und „groups.py“ werden hier angegeben, allerdings ohne „.py“, und gleich darauf mit anderen Bezeichnungen versehen, um die Arbeit leichter zu machen. Wie die Dateien verwendet werden, ist im Internet beschrieben und nach einigen Versuchen weiß man meist, wie es geht, vorausgesetzt, man hat eine Vorstellung davon, was man eigentlich will. Weitere Bibliotheken findet man, wenn Google „wie mache ich ...?“ fragt.

Mit den Zeilen

```
dim=2
kls=4
pts=30
```

konfiguriert ihr die Daten des Testsatzes:

- „dim“ gibt die Dimension an, also 1 für einen Zahlenstrahl, 2 für Punkte in einer Ebene, 3 für Punkte im Raum usw. Ihr könnt auch höhere Dimensionen verwenden, allerdings lassen die sich dann nicht mehr grafisch darstellen. Das geht nur bei 1 ... 3
- „kls“ gibt die Anzahl der Klassen an, die das Programm möglichst hinterher auch wiederfinden soll.
- „pts“ gibt die Anzahl der Punkte pro Klasse an, die möglichst auch wieder alle richtig zugeordnet werden sollen.

Die Programmsequenz

```

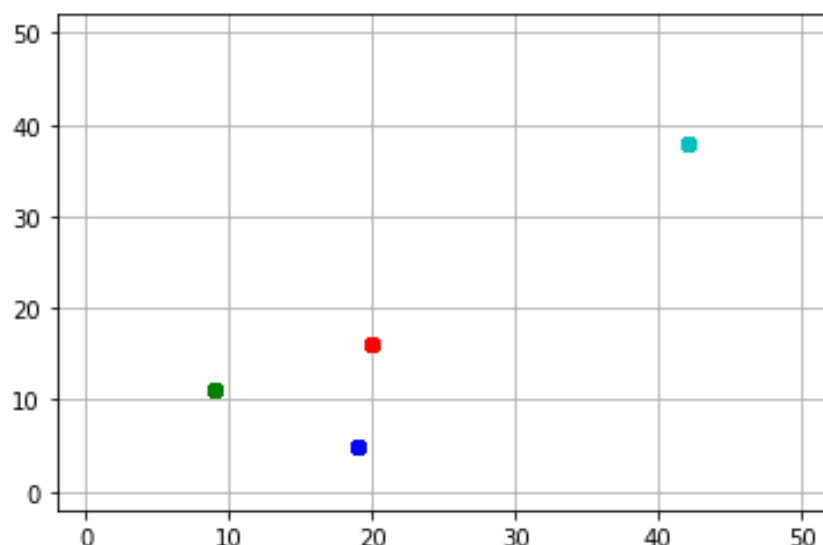
                                # Erstelle ein Testset oder lese eines ein
gen_set = True                 # Testset ohne Variation
setname = "testset0.bin"      # Name des Testsets

if(gen_set):
    a = iset.InitSet(dim,kls,pts)
    iset.WriteList(a,setname)
else:
    a = iset.ReadList(setname)

iset.PlotSet(a)
plt.show()
time.sleep(1)

```

erzeugt einen Testsatz und speichert ihn auf der Platte ab oder liest einen Testsatz von der Platte ein, falls da schon einer ist. Ihr könnt also einen Testsatz beliebig oft verwenden, um heraus zu bekommen, wie weit ihr gehen könnt. Die Syntax könnt ihr in einführenden Programmieranleitungen nachlesen. Das ist einfacher, als wenn ich hier den Text durcheinander bringe. Das Ergebnis des ersten Versuches sieht beispielsweise so aus:



Ihr habt 4 Klassen, also Punkte mit 4 verschiedenen Koordinaten erzeugt, wobei unter jedem der Punkte mehrere Punkte mit den gleichen Koordinaten liegen.

Das Programm dazu hat die Form

```
#
# Initiiert ein Feld mit dim Dimensionen, kls Klassen und jeweils pts
# gleichen Werten
# mit Hilfe von Zufallzahlen im Bereich 0 - 50
#
def InitSet(dim,kls,pts):

    pc = []
    global ddim
    ddim = dim
    global dkls
    dkls = kls
    global dpts
    dpts = pts
    for j in range(kls):
        pd = []
        for i in range(dim):
            pd.append(rd.randrange(50))
        for i in range(pts):
            pc.append(pd)

    return pc
```

und erzeugt eine Liste mit dem Inhalt (ich habe nur 2 Klassen mit 3 Punkten erzeugen lassen)

```
----- Matrix begin -----
6.00, 18.00
6.00, 18.00
6.00, 18.00
39.00, 17.00
39.00, 17.00
39.00, 17.00
----- Matrix end -----

# print(pc)

[[49, 36], [49, 36], [49, 36], [6, 27], [6, 27], [6, 27]]
```

So eine Liste von Punkten (pc) besteht aus kleineren Listen (pd), da jeder Punkt ja eine List von 3 Koordinaten ist. Weshalb dort zu Beginn des Programms Zeilen mit dem Begriff „global“ auftreten, hat damit zu tun, dass die Daten innerhalb des Unterprogramm gespeichert werden sollen. Eine Suche mit „Python global“ bringt schnell Klärung.

Schaut euch den Rest an und probiert ruhig ein wenig herum. Kaputt machen kann man nichts. Wenn eine Zeile nicht ausgeführt werden soll, kommt einfach „#“ an den Zeilenbeginn. Auf diese Art kann man auch feststellen, welche Aufgabe eine bestimmte Zeile hat.

Streuung von Punkten erzeugen

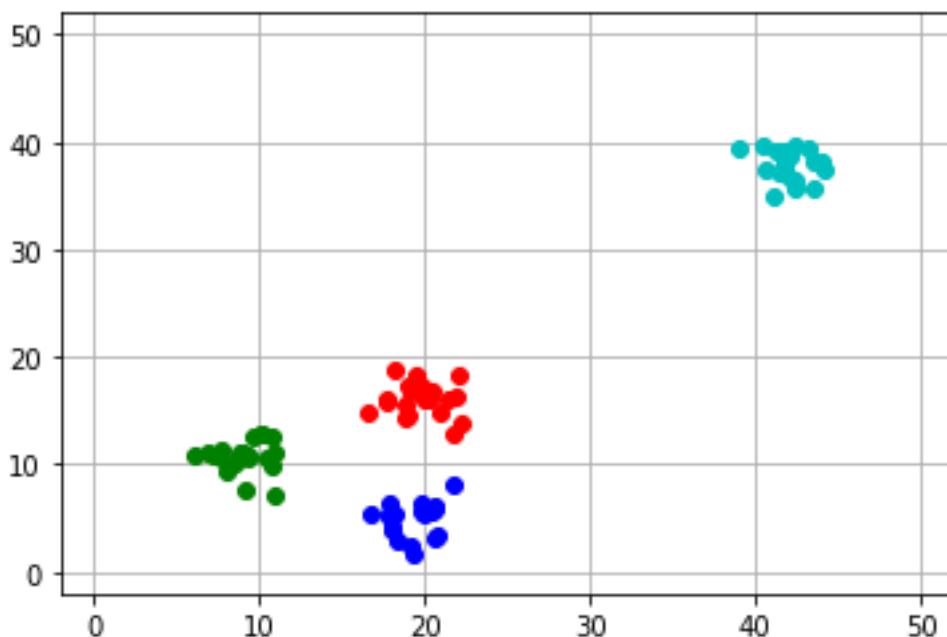
Im nächsten Programmteil werden die Punkte modifiziert, so dass nicht mehr alle die gleichen Koordinaten haben:

```
mod_set = True          # Modifiziere die Punkte eines Testsets
mod_name = "modset0.bin" # Name des modifizierten Sets
mod_val = [1.5,1.5,1.5,1.50] # Streuung der Klassen

if(mod_set):
    for i in range(kls):
        a=iset.ModifyClass(a, i, mod_val[i])
        grp.WriteList(a, mod_name)
else:
    a = grp.ReadList(mod_name)

iset.PlotSet(a)
plt.show()
time.sleep(1)
```

Im Vergleich zum ersten Teil sieht dieser Teil fast gleich aus, so dass er schnell zu verstehen ist. „mod_val“ enthält die Standardabweichung, mit der die Punkte modifiziert werden. Das Ergebnis ist:



Mit Veränderung von „mod_val“ kann man die Punkte auch stärker streuen lassen, so dass sie ineinander übergehen. Ihr könnt selbst ausprobieren, wie weit ihr gehen könnt, bevor das Programm aufgibt. Hier sind 4 deutlich getrennte Punktwolken vorhanden, die sich leicht identifizieren lassen sollten. Das dazu gehörende Programm hat die Form

```
#
# Modifiziert die Werte der Klassen kls mit normalverteilten Zufallsdaten
# der Streuung r
```

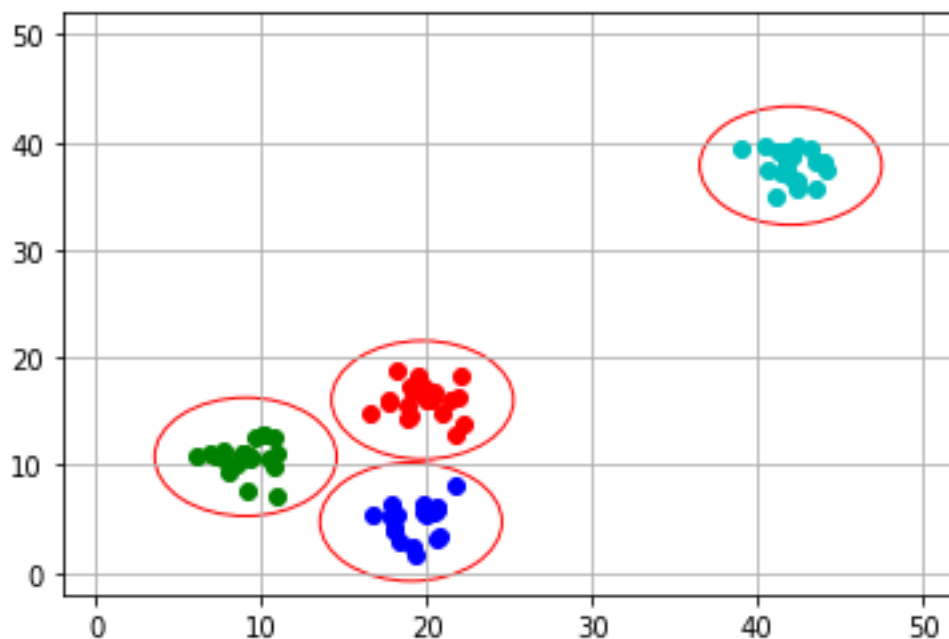
```
#
def ModifyClass(a,kls,r):
    for i in range(dpts):
        b=[]
        for j in range(ddim):
            b.append(a[i+dpts*kls][j] + rd.normalvariate(0.0, r))
        a[i+dpts*kls]=b
    return a;
```

Jede einzelne Punktkoordinate wird einzeln modifiziert. Wenn ihr das genauer sehen wollt, könnt ihr beispielsweise in der Zeile der „for“-Schleife einen so genannten Breakpoint hinzufügen (kurz vor die Zeilennummer mit der linken Maustaste klicken [Spider]), das Programm bis dahin laufen lassen und dann auf der rechten Seite im Fenster „Variablenmanger“ jeden Schritt einzeln verfolgen.

Die Plot-Programme erkläre ich hier nicht weiter. Ihr bekommt das sicher auch selbst heraus.

Der weitere Ablauf

Besteht in der Identifizierung der einzelnen Punktgruppen. In diesem Fall kommt das hier heraus:



Das Programm konnte alle Punktgruppen identifizieren und hat sie eingekreist. Schauen wir uns die einzelnen Schritte an:

(1) Zunächst wird eine dreieckige Matrix erzeugt, in der die Abstände zwischen allen Punkte berechnet sind. Da „ $\text{abstand}(p_1, p_2) = \text{abstand}(p_2, p_1)$ “ ist, können wir Speicherplatz sparen: #

```
# Metrik: Abstand zweier Punkte
#
def Metrik(p1,p2):
```

```

d=0
for i in range(len(p1)):
    q=p1[i] - p2[i]
    d=d+q*q
return math.sqrt(d)

#
# Erstellt eine Distanzmatrix (Dreiecksmatrix)
#
def Distance(a):
    m=[]
    for i in range(len(a)):
        d=[]
        for j in range(i,len(a)):
            d.append(Metrik(a[i],a[j]))
        m.append(d)
    return m

```

(2) Dann wird eine Liste erzeugt, die alle Punkt enthält, deren Abstand unter einer bestimmten Schwelle liegt.

```

# Sucht Punktpaare, deren Abstand eine bestimmte Größe nicht überchreitet
#
def GetInitalPairs(a,b,r):
    p=[]
    rg = len(a)
    for i in range(rg-1):
        for j in range(1,rg-i):
            if(r >= b[i][j]):
                p.append([i,i+j])
    return p

```

Die sieht dann beispielsweise so aus:

```
[[0, 1], [0, 2], [0, 4], [0, 8], [0, 10], ...
```

Man sieht, dass man die Punkte [0,1,2,4,8,10] auch in eine Liste schreiben kann, da sie über eine Kette kleiner Abstände verbunden sind. Das macht:

```

# verdichtet Punktpaare aus der vorhergehenden Funktion auf zusammen
# gehörende Punktlisten
# Die Verdichtung erfolgt rekursiv, da getrennte Listen entstehen können,
# obwohl Punkte zusammen gehören
#
def CompressPairs(pts):
    cp = []
    il=len(pts)
    while len(pts) > 0:
        ps = copy.copy(pts[0])
        pts.pop(0)
        i=0
        while i < len(pts):
            for j in range(len(pts[i])):

```

```

        if(pts[i][j] in ps):
            ps = ps + pts[i]
            pts.pop(i)
            break
    else:
        i=i+1
    cp.append(list(dict.fromkeys(ps)))

if(len(cp) < il):
    return CompressPairs(cp)
return cp

```

Das sieht etwas kompliziert aus, weil die Methode sich selbst aufruft. Ich empfehle euch, den Ablauf mit wenigen Punkten im Debugger zu beobachten.

Wichtig ist die Zeile „ps = copy.copy(pts[0])“. Normalerweise erzeugen Programmiersprachen eine Kopie der Daten. Python macht das nicht, d.h. die Zeile „ps=pts[0]“ führt dazu, dass Änderungen in „ps“ auch in „pts[0]“ zu sehen sind. Deshalb müssen wir hier explizit eine Kopie anfertigen. Die Ergebnisliste hat die Form

```

[[0, 2, 5, 6, 9, 14, 15, 18, 1, 7, 8, 10, 11, 12, 16, 19, 3, 4, 13, 17],
 [20, 22, 25, ...

```

(3) Darstellung des Erreichten:

4 Gruppen gefunden mit Punktabstand < 2.1230672483875694 zwischen zwei Punkten einer Gruppe

77 von 80 Punkten zugeordnet, nicht zugeordnet

Punkt 33 = [5.213484418152401, 12.524957719961511] nicht gefunden

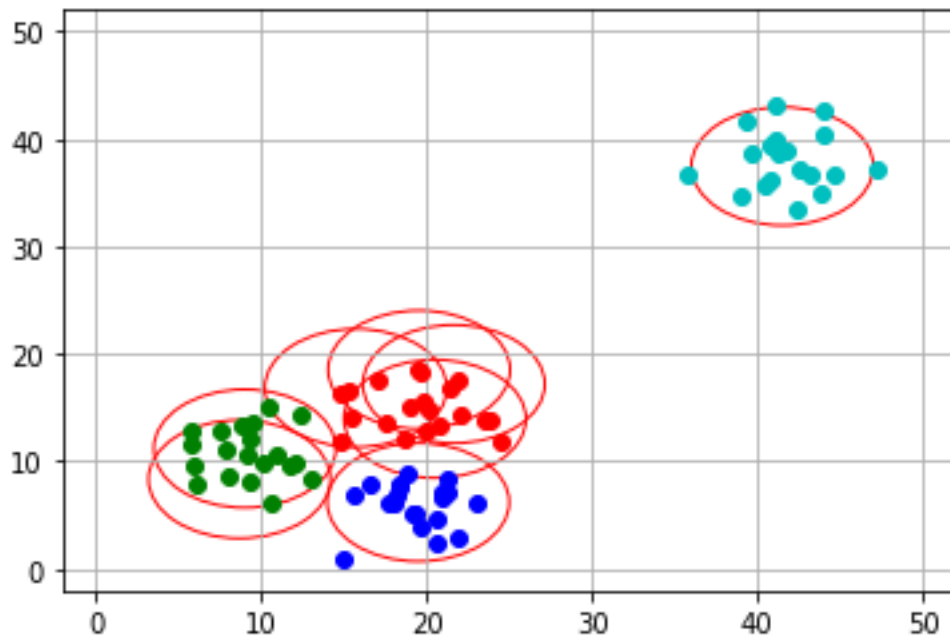
Punkt 36 = [8.19280153028141, 15.511342474336491] nicht gefunden

Punkt 78 = [41.94839775143278, 41.60322433707399] nicht gefunden

Zum Abschluss wird der Schwerpunkt der Punktgruppen berechnet und die Zeichnung mit den Kreisen erzeugt. Damit sind wir erst einmal fertig.

Diskussion

Wie am Beispiel schon zu sehen war, sind 3 Punkte nicht korrekt eingeordnet worden. Die Sache kann aber auch deutlich schlimmer ausgehen, beispielsweise mit einer Standardabweichung von 2,5 bei der Variation der Punkte:



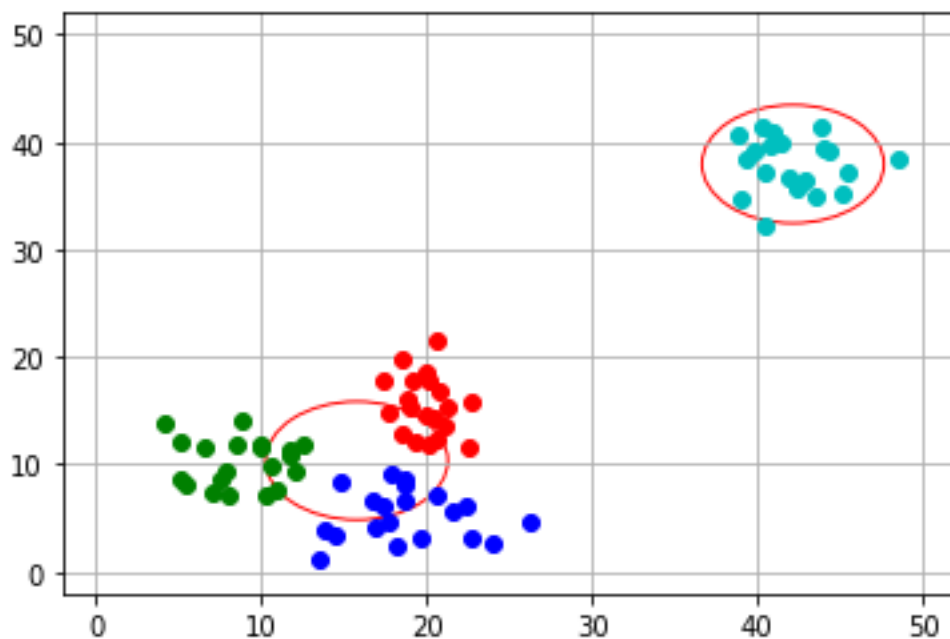
8 Gruppen gefunden mit Punktabstand < 2.172119381617514 zwischen zwei Punkten einer Gruppe

67 von 80 Punkten zugeordnet

Punkt 11 = [14.917697371871535, 1.045138590942651] nicht gefunden

Punkt 21 = [12.380908723087446, 14.25924584466496] nicht gefunden

Würde mit einem Punktabstand von 5,5 gearbeitet, findet man:

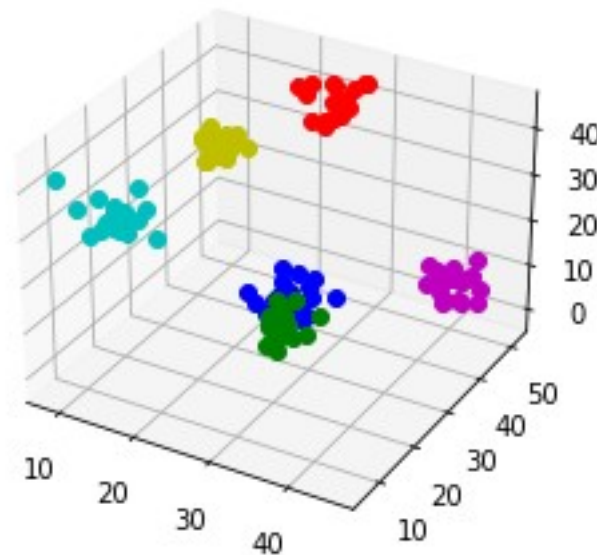


2 Gruppen gefunden mit Punktabstand < 5.34499437878396 zwischen zwei Punkten einer Gruppe

alle 80 Punkte gruppiert

Auch da sieht man, dass das nicht stimmt, denn die ersten drei Gruppen werden als eine Gruppe erkannt, aber nicht alle Punkte liegen im Erkennungskreis.

Die Werte für die Variationen usw. sind in diesen Beispielen willkürlich gesetzt worden. Alles funktioniert auch mit höheren Dimensionen, hier im 3D



6 Gruppen gefunden mit Punktabstand < 5.525555839212305 zwischen zwei Punkten einer Gruppe
119 von 120 Punkten zugeordnet

Hier könnt ihr euch nun austoben und eine Reihe von Fragestellungen bearbeiten, z.B.

- Wie müssen verschiedene Parameter gewählt werden, um alle Gruppen wieder zu finden? Kann man zur Lösung des Problems aus den Punktwolken zusätzliche Parameter berechnen, die hilfreich sind.
- Wie stark dürfen Gruppen überlagern, um zumindest einen Teil der Punkte korrekt zuzuordnen (auch hier kann man ev. wieder etwas berechnen)?
- Wie kann man die Kreise so berechnen, dass auch Ausreißer noch in der richtigen Gruppe landen?
- Kann man die Abweichungen der Punkte für alle Koordinaten verschieden gestalten und trotzdem noch sinnvolle Klassifizierungen bekommen?
- ...

Zum Schluss solltet ihr ein Programm haben, das mit zufälligen Werten so gut ist, wie ihr selbst, wenn ihr die Farbe aus den Grafiken herausnimmt und eben nicht mehr einfach erkennen könnt, was wo hin gehört.