Wintersemester 2016/2017, AST

Cluster Analysis Übung

In dieser Übung werden wir

- die Stastik Software "R" kennenlernen,
- zwei unterschiedliche Clustering-Methoden anwenden..
- .. und vergleichen.

1 R

"R is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files." — R project page, `http://www.r-project.org/`

Hier gibt es zunächst eine Einführung in R. Probier alles aus und guck ob du alles verstehst.

1.1 R starten

In dieser Übung werden Befehle entweder in der Linux-Kommandozeile oder in der R-Umgebung eingegeben. Linux-Befehle sind mit "`$`" gekenzeichnet, R-Befehle mit "`>`".

Öffne ein Linux-Terminal und erstelle ein Verzeichnis:

```
$ mkdir ast-cluster
$ cd ast-cluster
```

Starte R:

```
$ R
```

Verlassen kann man R mit `q()`. Beim Verlassen kann man die aktuelle Session speichern. Startet man R im gleichen Verzeichnis nochmal, kann man da weiter machen, wo man aufgehört hat. Wie in der Linux-Kommandozeile, kann man mit den Pfeiltasten und der Tab-Taste Zeit sparen.

1.2 Variablen, Vektoren, Matrizen

Eine Variable mit einem Wert zu belegen ist einfach:

```
> antwort = 42
```

Alternativ funktioniert auch `<-` statt `=`. Den Wert einer Variable erhält man, indem man einfach ihren Namen eingibt:

```
> antwort
[1] 42
```

Der erste Teil der Ausgabe ("`[1]`") bezeichnet die Zeile. Das ist natürlich überflüssig bei einfachen Zahlen oder Vektoren, aber nützlich bei Matrizen. Mit `ls()` erhält man eine Liste aller Variablen. Lässt man die Klammern weg und tippt nur `ls` sieht man die Funktionsdefinition. Das geht allen Funktionen. Keine Sorge, falls du den angezeigten Code nicht verstehst!

Vektoren kann man mit der Funktion `c` (eine Abürzung für "combine") erstellen:

```
> v = c(1,2,3)
> v
[1] 1 2 3
```

Auf den Inhalt eines Vektors kann man mit einem Index zugreifen. ACHTUNG: Im Gegensatz zu vielen anderen Programmiersprachen hat das erste Element den Index `1` und nicht `0`! Index `0` einer beliebigen Variablen gibt den Typen der Variable zurück.

```
> v[0]
numeric(0)
> v[1]
[1] 1
```

Will man einen Vektor um eine Variable erweitern, kann man die Funktion `append` benutzen:

```
> append(v,10)
[1]  1  2  3 10
```

Man kann auch einen ganzen zweiten Vektor hinzufügen:

```
> w = (10,11,12)
> append(v,w)
[1]  1  2  3 10 11 12
```

Matrizen erstellt man auf ähnliche Weise. Die Funktion `matrix` akzeptiert als Argumente einen Vektor und zusätzliche Angaben, wie z.B. die Anzahl der Zeilen (`nrow`) oder Spalten (`ncol`). Die Matrix wird Spaltenweise mit den Werten aus dem Vektor gefüllt, solange nicht der Parameter `byrow` auf `TRUE` gesetzt wird.

```
> foo = matrix(c(1,2,3,4,5,6), nrow=3)
> foo

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

bzw.

```
> bar = matrix(c(1,2,3,4,5,6), nrow=3, byrow=TRUE)
> bar
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Der Zugriff auf die Einträge der Matrix erfolgt wie bei den Vektoren über die eckigen Klammern. Die Indizes für Zeile und Spalte sind durch ein Komma getrennt.

```
> foo[3,2]
[1] 6
```

Ein negativer Index dagegen gibt alle Elemente ausgenommen des angegebenen aus:

```
> foo[-3,2]
[1] 4 5
```

Gibt man einen Index nicht an, so wird die gesamte Zeile oder Spalte ausgewählt:

```
> foo[,2]
[1] 4 5 6
```

Matrizen lassen sich mit `rbind` und `cbind` Zeilen- und Spaltenweise erweitern.

```
> rbind(foo, c(3.5,6.5))

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
[4,]  3.5  6.5
```

```
> cbind(bar, c(7,8,9))

      [,1]  [,2]  [,3]
[1,]    1     4     7
[2,]    2     5     8
[3,]    3     6     9
```

1.3 Hilfe

Hilfe zu jeder Funktion bekommt man mit dem Befehl `help`:

```
> help(plot)
```

Abkürzen lässt sich das mit

```
> ?plot
```

In der Dokumentation kann man außerdem mit dem `??` suchen:

```
> ??plot
```

1.4 Skripte

Oft ist es sinnvoller die Befehle in einem Skript zu speichern und dieses dann auszuführen. Erstelle einfach ein Skript mit einem Texteditor (z.B. `kate`). Schreibe etwas R-Code hinein - z.B. `print('bonjour le monde')` - und speichere das script (z.B. als `bonjour.r`). In der Linux-Kommandozeile kann man es nun folgendermaßen ausführen:

```
$ Rscript bonjour.r
```

Mit `source` kann man auch im interaktiven Modus in R den Inhalt aus einer Datei laden. `source('bonjour.r')` wird die Datei `bonjour.r` laden und alle Kommandos ausführen, als hätte man sie direkt eingegeben.

Beachte: Um Variablen in einem Skript auszugeben, muss man `print` benutzen.

Um ein Skript auch später noch zu verstehen, ist es hilfreich den Code mit Kommentaren zu ergänzen. Kommentare beginnen mit `#` und werden vom Interpreter ignoriert.

```
# diese Zeile wird ignoriert
print('das hier ist die Ausgabe')
# noch mehr Kommentar,
# welcher von R ignoriert wird
```

1.5 Schleifen

`for` Schleifen in `R` haben die Form `for(Name in Vektor)` und gehen durch einen Vektor und führen den Code in den geschweiften Klammern für jedes Element aus. Zum Beispiel

```
for(i in 1:6)
{
  print(i)
}
```

gibt die Zahlen von 1 bis 6 aus. Der Ausdruck `1:6` erzeugt einen Vektor mit ganzen Zahlen von 1 bis 6.

Noch ein Beispiel:

```
stuff = c('shoes', 'moose', 'goose')
for(i in stuff)
{
  print(i)
}
```

Hier wird ein Vektor mit Zeichenketten erzeugt. Anschließend werden alle Einträge ausgegeben.

1.6 Eigene Funktionen schreiben

Natürlich kann man auch eigene Funktionen in R schreiben. Die sehen dann so aus:

```
add = function(a,b)
{
  s = a + b
  s
}
```

Die beschriebene Funktion heißt `add`. Sie bekommt zwei Parameter und addiert diese. Der letzte angegebene Wert ist der Rückgabewert der Funktion. Aufrufen würde man die Funktion dann folgendermaßen:

```
> add(2,3)
[1] 5
```

1.7 Tipp

Benutze R interaktiv um Funktionen auszuprobieren und Datenstrukturen anzusehen. Auf diese Weise und mit der Hilfe-Funktion kann man meist schnell erkunden, was bestimmte Ausdrücke tun und schnell dazulernen. Schreibe längere Funktionen außerdem immer in

Skript-Dateien. So lassen sie sich einfach modifizieren, wiederverwenden und gegebenenfalls korrigieren.

2 Cluster-Analyse

Im Folgenden werden wir zunächst die Daten einlesen und in einer Grafik abbilden. Anschließend werden wir die Methoden `kmeans` und `hclust` für Clusteranalysen dieser Daten benutzen.

Hinweise:

- Im Gegensatz zu anderen Programmiersprachen ist `.` kein Operator in R. Wir verwenden ihn bei der Bezeichnung von Variablen wie einen Buchstaben.
- Nutze die Hilfe-Funktion, wenn du wissen willst, was die einzelnen Funktionen tun.

2.1 Daten einlesen und visualisieren

Öffne die Datei mit den Beispieldaten und speichere sie in Matrixform in einer Variablen:

```
> cluster.datapath = file.path('/home/petersen/ast/sampledata')
> cluster.data = as.matrix(read.table(cluster.datapath, sep=','))
```

Sie dir die Form der Matrix an:

```
> dim(cluster.data)
[1] 100 2
```

Die Matrix hat also 100 Zeilen und 2 Spalten.

Nun wollen wir den Inhalt der Datei in Form einer Grafik betrachten. Dafür gib folgendes ein:

```
> plot(cluster.data)
```

Abbildung 1 zeigt wie das Ergebnis aussehen sollte. Um das Ergebnis in einer Datei (hier mit dem Namen `myplot.png`) zu speichern gib folgendes ein:

```
> png('myplot.png')
> plot(cluster.data)
> dev.off()
```

Der letzte Befehl `dev.off()` schließt die Grafik und speichert sie in der Datei. Die nächste Grafik wird wieder auf dem Bildschirm gezeigt.

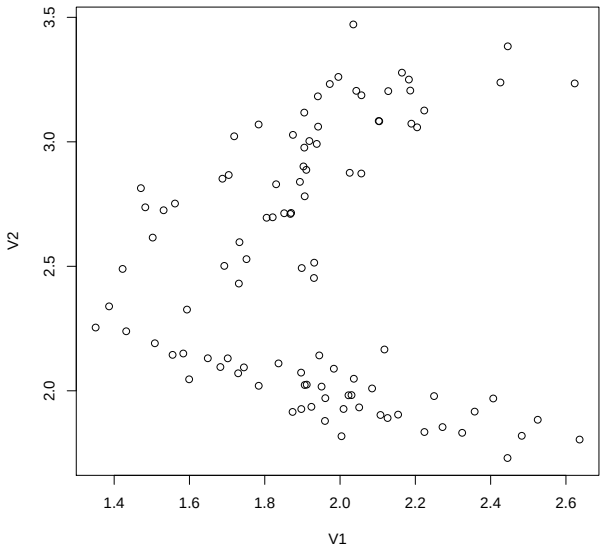

Abbildung 1: Die Testdaten

2.2 K-means

Hat man die Daten wie eben beschrieben eingelesen, kann man k-means mit einem Befehl ausführen (ersetze `<Anzahl Cluster>` durch eine Zahl):

```
> cluster.kmeans = kmeans(cluster.data, <Anzahl Cluster>)
```

Das Ergebnis ist eine umfangreiche Datenstruktur. Gibt man `cluster.kmeans` ein, kann man diese betrachten. Mit dem `$`-Zeichen kann man auf die verschiedenen Elemente zugreifen.

Übung 2.1
Lade die Daten und führe kmeans wie beschrieben aus. Betrachte die Struktur in `cluster.kmeans`. Was befindet sich in `cluster.kmeans$cluster` und was in `cluster.kmeans$centers`?

Nun wollen wir das Ergebnis visualisieren (das Fenster nicht schließen vor der Eingabe des zweiten Befehls!):

```
> plot(cluster.data, col = cluster.kmeans$cluster)
> points(cluster.kmeans$centers, col = 1:6, pch=7, lwd=3)
```

Die Funktion `points` fügt dem Plot weitere Punkte hinzu. Die Parameter `col`, `pch` und `lwd` bestimmen Farbe, Form und Größe der Punkte. Für genauere information nutze `?plot` und `?points`.

Übung 2.2
Führe `kmeans` mit unterschiedlich vielen Clustern durch. Visualisiere die Ergebnisse.

Nun wollen wir jetzt die Ergebnisse vergleichen, wenn wir zwei mal mit der gleichen Clusterzahl clustern. Dafür führen wir `kmeans` zwei mal aus und speichern die Ergebnisse in unterschiedlichen Variablen, z.B. `cluster.kmeans.1` und `cluster.kmeans.2`. Nun nutzen wir die Funktion `table` um die Ergebnisse zu vergleichen. Die Eingabe sieht folgendermaßen aus:

```
> tab = table(cluster.kmeans.1$cluster, cluster.kmeans.2$cluster)
```

Betrachte nun das Ergebnis mit

```
> tab
```

Dies ist eine Konfusionsmatrix. An jeder Position steht wie viele Punkte jeweils in den entsprechenden Clustern gelandet sind bei den beiden Analysen. Ist in Zeile 1 in Spalte 3 eine 7, bedeutet das, dass 7 Punkte, die in einem Lauf in Cluster 1 gelandet sind beim anderen mal in Cluster 3 landeten.

Unglücklicherweise ist die Numerierung in den verschiedenen Läufen nicht gleich, da die initialisierung der Clusterzentren zu Beginn zufällig ist. Um die Ungleichheit der Ergebnisse zu bestimmen, müssen wir also erst die Cluster-Nummern einander zuordnen. Mit der folgenden Funktion ordnen wir die Cluster-Nummern einander zu und berechnen die Rate der ungleich zugeordneten Datenpunkte:

```
cluster.mismatchrate = function(x)
{
    miss = 0              # initialisierung der 'Mismatch'-Rate
    for(i in 1:nrow(x))   # betrachte alle Zeilen
    {
        thisrow = as.vector(x[i,])      # die aktuelle Zeile
        maxindex = which.max(thisrow)   # finde die zusammen gehörenden Cluster
        miss = miss+sum(x[i,-maxindex]) # Addiere alle Werte aus
                                        # unterschiedlichen Clustern
    }
    miss = miss/sum(x)   # normalisieren
    miss                 # Rückgabe
}
```

Für den von uns verwendete Datansatz reicht diese Funktion aus.

Wir nutzen die Funktion mit der zuvor berechneten Konfusionsmatrix folgendermaßen:

```
> cluster.mismatchrate(tab)
```

Übung 2.3
Implementiere und teste die Funktion `cluster.mismatchrate`. Um einzelne Funktionen zu verstehen nutze `?` und/oder probiere die Funktion in der interaktiven Umgebung aus.

2.3 Hierarchische Clusteranalyse

Im Gegensatz zu k-means, welches auf Koordinaten rechnet, arbeiten die hierarchischen Clustermethoden in R mit einer Distanzmatrix. Glücklicherweise lässt sich diese in R leicht berechnen.

```
> cluster.dist = dist(cluster.data, method='euclidean')
```

Das Ergebnis ist eine Dreiecksmatrix, welche an jedem Eintrag (i,j) die Distanz zwischen den Punkten i und j enthält. Jetzt können wir clustern:

```
> cluster.hclust = hclust(cluster.dist)
```

Das Ergebnis ist die Reihenfolge, in welcher Cluster zu einem Baum zusammengeführt wurden, wobei zu Beginn jeder Datenpunkt ein eigenes Cluster darstellt. Dieses Ergebnis kann mit dem Befehl

```
> plot(cluster.hclust)
```

als *Dendrogramm* visualisiert werden. Um dies mit der Ausgabe von k-means zu vergleichen, wird der Baum mit dem Befehl

```
> cluster.hcut = cutree(cluster.hclust,<number of clusters>)
```

geschnitten. Der Baum wird einfach in die `k` Cluster zerteilt, welche als letztes zusammengeführt wurden. Der Vektor `cluster.hcut` enthählt nun die gleiche Art Information wie zuvor `cluster.kmeans$cluster` und kann dementsprechend behandelt werden.

3 Aufgaben

1. Schreibe eine Funktion, die 500 Paare k-means mit der gleichen Anzahl Cluster durchführt und die durchnittliche "mismatch"-Rate berechnet. Vergleiche das Ergebnis für unterschiedliche Clusterzahlen. Wann und warum unterscheiden sich die Ergebnisse? Entspricht dies deiner Erwartung?

2. Vergleiche 500 Paare k-means und hierarchische Cluster. Gibt es einen systematischen Unterschied zwischen den Ergebnissen der beiden Methoden? Erkläre deine Beobachtungen.

3. Was ist die "linkage method" bei der hierarchischen Clusterarnalyse und wie funktioniert sie? Ändert sich das Ergebnis wenn man die Methode ändert? Beginne damit die Dokumentation in R help für hclust zu lesen.

4. Füge dem Datensatz bis zu 5 Punkte hinzu. Benutze keine überdimensional großen oder kleinen Werte, bleib im Rahmen der gegebenen Werte. Kannst du Werte so hinzufügen, die das Ergebnis von `hclust` stark verändern? Wo hast du sie hinzugefügt und warum? Ändern sie das Ergebnis von k-means?

4 Bonus Aufgabe

Man kann K-means als eine Methode betrachten, welche folgenden Term - im folgenden Quadratsumme genannt - minimiert:

$$\sum_{i=1}^{k} \sum_{x \in C_i} ||x - \mu_i||^2$$

k ist die Anzahl der Cluster, μ_i das Zentrum des Clusters i, und $||x - \mu_i||^2$ der quadratische Abstand zwischen Punkt x und Zentrum μ_i. Beachte: Die Funktion $||x||$ ist die Länge des Vektors x (auch Norm genannt).

Dieser Wert wird von der Funktion `kmeans` bereits berechnet und befindet sich in dem Array `kmeans(data, centers=k)$withinss`, welcher den durchschnittlichen Quadratischen Abstand aller Punkte vom Zentrum für jedes Cluster enthält. Die gesamte Summe bekommt man also mit `sum(kmeans(data, centers=k)$withinss)`.

Erstelle Plots für Quadratsummen von Clustern mit Größen 2 bis 15. Die Quadratsumme wird kleiner wenn die Anzahl der Cluster steigt, aber wir hätten gerne nur so wenige Cluster wie nötig. Fällt die Quadratsumme bei einer bestimmten Anzahl von Clustern besonders stark? Im besten Fall fällt der Wert an einer Stelle besonders schnell und wir sehen eine Ellenbogenförmige Kurve. Die Stelle mit einem besonders stark fallenden Wert betrachten wir als eine gute Clusterzahl.

Hier ist ein Beispiel von einem anderen Datensatz: