Lab 5: Clustering Methods

PSTAT 131/231, Spring 2018

Learning Objectives

- k-means clustering
  - `kmeans()` function
  - Different linkages (complete, single, average)
  - Pros and cons
- k-medoids clustering
  - `pam()` function
- Hierachical clustering
  - `hclust()` function
  - Make a dendrogram
- Cluster similarity comparison
  - By a simple table
  - `levelplot()` function

---

1. Pre-processing of data

We will look at cluster analysis performed on a set of cars from 1978-1979. The dataset includes gas mileage (coded as mpg), engine horsepower (horsepower), and other information for 38 vehicles. It is similar to the Auto dataset from ISLR, but has fewer observations and features.

In R, a number of cluster analysis algorithms are available through the library `cluster`, providing us with a large selection of methods to perform cluster analysis, and the possibility of comparing the old methods with the new to see if they really provide an advantage.

Load the package `cluster` and look at the first three observations:

```
# install.packages("cluster")
library(cluster)

# Read tab-delimited file by read.delim
car = read.delim("https://www.stat.berkeley.edu/~s133/data/cars.tab",
                 sep = "\t",  # tab-delimited file
                 header = T)

# First 3 observations
head(car, 3)
```

```
##   Country                      Car  MPG Weight Drive_Ratio Horsepower
## 1    U.S.        Buick Estate Wagon 16.9  4.360        2.73        155
## 2    U.S. Ford Country Squire Wagon 15.5  4.054        2.26        142
## 3    U.S.        Chevy Malibu Wagon 19.2  3.605        2.56        125
##   Displacement Cylinders
## 1          350         8
## 2          351         8
## 3          267         8
```

The numerical variables (`MPG`, `Weight`, `Drive_Ratio`, `Horsepower`, `Displacement` and `Cylinders`) are measured on different scales, therefore we want to standardize the data before proceeding. There are multiple

methods to re-scale the variables, we do it by subtracting the mean and dividing the standard deviation from each feature.

`scale()` function helps with standardization. It has two more arguments other than passing in the data matrix. By `center=TRUE`, centering is done by subtracting the column means (omitting NAs); by `scale=TRUE`, scaling is done by dividing the (centered) columns by their standard deviations. If `center=FALSE` or `scale=FALSE`, no centering or scaling is done.

```
# Standardize the variables by subtracting mean and divided by standard deviation
scar = scale(car[,-c(1,2)], center=TRUE, scale=TRUE)
```

2. Clustering functions

(a) k-means clustering

k-means clustering is available in R through the `kmeans()` function. The first step (and certainly not a trivial one) when using k-means cluster analysis is to specify the number of clusters (k) that will be formed in the final solution. The process begins by randomly choosing k observations to serve as centers for the clusters. Then, the **squared euclidean distance** from each of the other observations is calculated for each of the k clusters, and observations are put in the cluster to which they are the closest. After each observation has been put in a cluster, the center of the clusters is recalculated, and every observation is checked to see if it might be closer to a different cluster, now that the centers have been recalculated. The process continues until no observations switch clusters.

Note: The idea behind k-means clustering is that a good clustering is one for which the within-cluster variation is as small as possible. The most common choice for within-cluster variation measure is the squared euclidean distance.

```
set.seed(1)
# 3-means clustering
km = kmeans(scar, centers=3)
km
```

```
## K-means clustering with 3 clusters of sizes 13, 8, 17
##
## Cluster means:
##          MPG     Weight Drive_Ratio Horsepower Displacement  Cylinders
## 1 -0.5602631  0.2238870   0.2578965  0.3473723  -0.02489459  0.1376443
## 2 -1.1203872  1.4864539  -1.2960719  1.3712707   1.66759621  1.6252130
## 3  0.9556775 -0.8707154   0.4127012 -0.9109415  -0.76571412 -0.8700635
##
## Clustering vector:
##  [1] 2 2 2 2 3 3 3 3 1 1 1 1 1 1 1 1 2 2 2 2 3 1 3 3 3 3 3 3 1 1 3 3 3 3 3
## [36] 1 1 3
##
## Within cluster sum of squares by cluster:
## [1] 24.08477  3.74590 18.91804
##  (between_SS / total_SS =  78.9 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
## [5] "tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

Looking on the good side, the k-means technique is fast, and doesn't require calculating all of the distances between each observation and every other observation. It can be written to efficiently deal with very large data sets, so it may be useful in cases where other methods fail. On the down side, if we rearrange our data, it's very possible that we'll get a different solution every time we change the ordering of the data. Another criticism of this technique is that we may try, for example, a 3 cluster solution that seems to work pretty well, but when we look for the 4 cluster solution, all of the structure that the 3 cluster solution revealed is gone. This makes the procedure somewhat unattractive if we don't know exactly how many clusters we should have in the first place.

(b) k-medoids clustering

k-medoids is based on centroids (or medoids) calculating by minimizing the **absolute distance** between the points and the selected centroid, rather than minimizing the squared euclidean distance. As a result, it's more robust to noise and outliers than k-means. The most common realization of k-medoid clustering is the Partitioning Around Medoids (PAM) algorithm.

The R library `cluster` provides PAM algorithm in the function `pam()`, which requires that we know the number of clusters that we wish to form (like k-means clustering). What's more, PAM does more computation than k-means in order to insure that the medoids it finds are truly representative of the observations within a given cluster. `pam()` also offers some additional diagnostic information about a clustering solution and provides a nice example of an alternative technique to hierarchical clustering.

Recall that in the k-means method, the centers of the clusters (which might or might not actually correspond to a particular observation) are only recalculated after all of the observations have had a chance to move from one cluster to another. With PAM, the sums of the distances between objects within a cluster are constantly recalculated as observations move around, which will hopefully provide a more reliable solution.

`pam()`: We can pass a data frame, or a distance matrix[1] into the first argument `x`. The second argument `k` is the number of clusters that we wish to form. By `keep.diss=TRUE`, R is instructed to keep the dissimilarities. By `keep.data=TRUE`, the input data should be kept in the result.

To compute a distance matrix, two functions can be used: `dist()` (in the default packages) and `daisy` (in `cluster`).

`dist()` computes the distance matrix by using the specified distance measure. Commonly used distance measures are Euclidean distance (by specifying argument `method="euclidean"`), Manhattan distance (`method="manhattan"`) and Maximum distance (`method="maximum"`)

Let's perform a 3-medoids clustering by PAM:

```
# First, we use daisy to calculate the distance matrix
car.dist = daisy(scar)
# can also do: car.dist = dist(scar)

# 3-medoids clustering
scar.pam = pam(scar, k=3, keep.diss=TRUE)
scar.pam
```

```
## Medoids:
##      ID        MPG      Weight Drive_Ratio Horsepower Displacement
## [1,] 20 -1.0020180  1.36815073 -1.24294794  1.2578275   1.58320969
## [2,]  8  0.9377088 -0.89534762  0.53428969 -1.0110385  -0.81336768
## [3,] 22 -0.4369007  0.06663918 -0.02592652  0.2746522  -0.07076625
```

[1] Recall several dissimilarity measures: *Euclidean distance (L_2 norm)*: the usual distance between the two vectors, $(\sum_{k=1}^{n}(x_k - y_k)^2)^{1/2}$; *Manhattan distance ($L_1$ norm)*: the absolute distance between the two vectors, $\sum_{k=1}^{n}|x_k - y_k|$; *Maximum distance ($L_\infty$ norm)*: the maximum distance between two components of x and y, $\max_{1\le k\le n}|x_k - y_k|$

```
##       Cylinders
## [1,]  1.6252130
## [2,] -0.8700635
## [3,]  0.3775747
## Clustering vector:
##  [1] 1 1 1 1 2 2 2 2 3 3 3 3 3 3 3 3 1 1 1 1 2 3 2 2 2 2 2 2 3 3 2 2 2 2 2 2
## [36] 3 3 2
## Objective function:
##    build     swap
## 1.137333 1.031719
##
## Available components:
##  [1] "medoids"    "id.med"     "clustering" "objective"  "isolation"
##  [6] "clusinfo"   "silinfo"    "diss"       "call"       "data"
```

Like most R objects, we can use the `names()` to see what else is available. Further information can be found in the help page for `pam.object`.

```
names(scar.pam)
```

```
##  [1] "medoids"    "id.med"     "clustering" "objective"  "isolation"
##  [6] "clusinfo"   "silinfo"    "diss"       "call"       "data"
```

```
# To check the dissimilarity matrix which was saved by keep.diss=TRUE, use the following command
# scar.pam$diss
# We do not actually run above line since the output is too long to display
```

Another feature available with `pam()` is a plot known as a silhouette plot. We will talk about it next week.

(c) Hierarchical Clustering

One potential disadvantage of k-means clustering or PAM is that it requires us to pre-specify the number of clusters k. Hierarchical clustering is an alternative approach which does not require that we commit to a particular choice of k. Hierarchical clustering has an added advantage over k-means clustering in that it results in an attractive tree-based representation of the observations, called a dendrogram.

To perform hierarchical clustering, we can use function `hclust()` by passing the distance matrix into it. By default, R uses complete linkage (defaulting `method="complete"`). We can use single linkage (by specifying `method="single"`) and average linkage (`method="average"`) too. The `cluster` library provides a similar function, called `agnes` to perform hierarchical cluster analysis. Naturally, the first step is calculating a distance matrix. Again, there are two functions that can be used to calculate distance matrices, `dist()` and `daisy()`. The Hierarchical clustering is shown below.

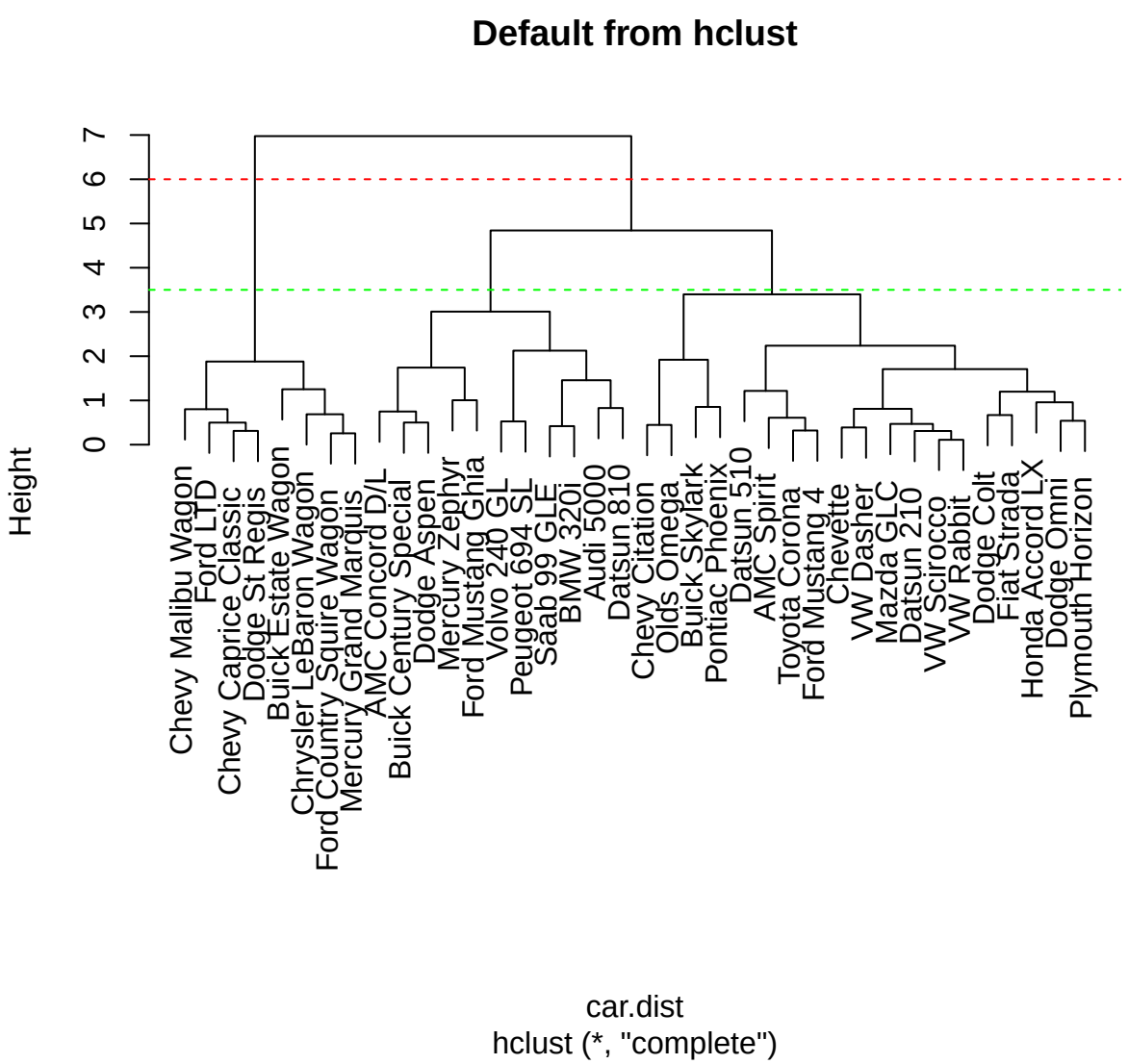
```
# We use the dist function in this example.
car.dist = dist(scar)
# Can also do: car.dist = daisy(scar)

# Agglomerative Hierarchical clutering
set.seed(1)
car.hclust = hclust(car.dist) # complete linkage
# Can also do: car.hclust = agnes(car.dist)
```

Note: We're using the default method of `hclust()`, which is to update the distance matrix using what R calls "complete" linkage. Using this method, when a cluster is formed, its distance to other objects is computed as the maximum distance between any object in the cluster and the other object. Other linkage methods will provide different solutions, and should not be ignored. For example, using `method=ward` tends

to produce clusters of fairly equal size, and can be useful when other methods find clusters that contain just a few observations.

Now that we've got a cluster solution (actually a collection of cluster solutions), the main graphical tool for looking at a hierarchical cluster solution is known as a dendrogram. This is a tree-like display that lists the objects which are clustered along the x-axis, and the distance at which the cluster was formed along the y-axis. (Distances along the x-axis are meaningless in a dendrogram; the observations are equally spaced to make the dendrogram easier to read.) To create a dendrogram from a cluster solution, simply pass it to the `plot()` function. The dendrogram can be generated as below.

```
# Plot dendogram
plot(car.hclust, labels=car$Car, main='Default from hclust')
# Add a horizontal line at a certain height
abline(h=6, col="red", lty=2)
abline(h=3.5, col="green", lty=2)
```



If we choose any height along the y-axis of the dendrogram, and move across the dendrogram counting the number of lines that we cross, each line represents a group that was identified when objects were joined together into clusters. The observations in that group are represented by the branches of the dendrogram that spread out below the line. For example, the red line gives us a 2-cluster solution, and the green line

gives a 3-cluster result.

Looking at the dendrogram for the car data, there are clearly two very distinct groups; the right hand group seems to consist of two more distinct cluster, while most of the observations in the left hand group are clustering together at about the same height. It looks like either two or three groups might be an interesting place to start investigating.

One of the first things we can look at is how many cars are in each of the groups. We'd like to do this for both the 2-cluster and 3-cluster solutions. we can create a vector showing the cluster membership of each observation by using the `cutree()` function. Since the object returned by a hierarchical cluster analysis contains information about solutions with different numbers of clusters, we pass the `cutree()` function the cluster object and the number of clusters we're interested in.

```
clus = cutree(car.hclust, 3)
```

A good step is to use the table function to see how many observations are in each cluster. We'd like a solution where there aren't too many clusters with just a few observations, because it may make it difficult to interpret our results. For the three cluster solution, the distribution among the clusters looks good:

```
table(clus)
```

```
## clus
##  1  2  3
##  8 19 11
```

3. Cluster similarity and comparison by a simple table

We can use `table` to compare the results of the `hclust()` and `pam()` solutions:

```
table(hclust=clus, pam=scar.pam$clustering)
```

```
##       pam
## hclust  1  2  3
##      1  8  0  0
##      2  0 17  2
##      3  0  0 11
```

The solutions seem to agree, except for 2 observations that hclust put in group 2 and pam put in group 3. Which observations were they?

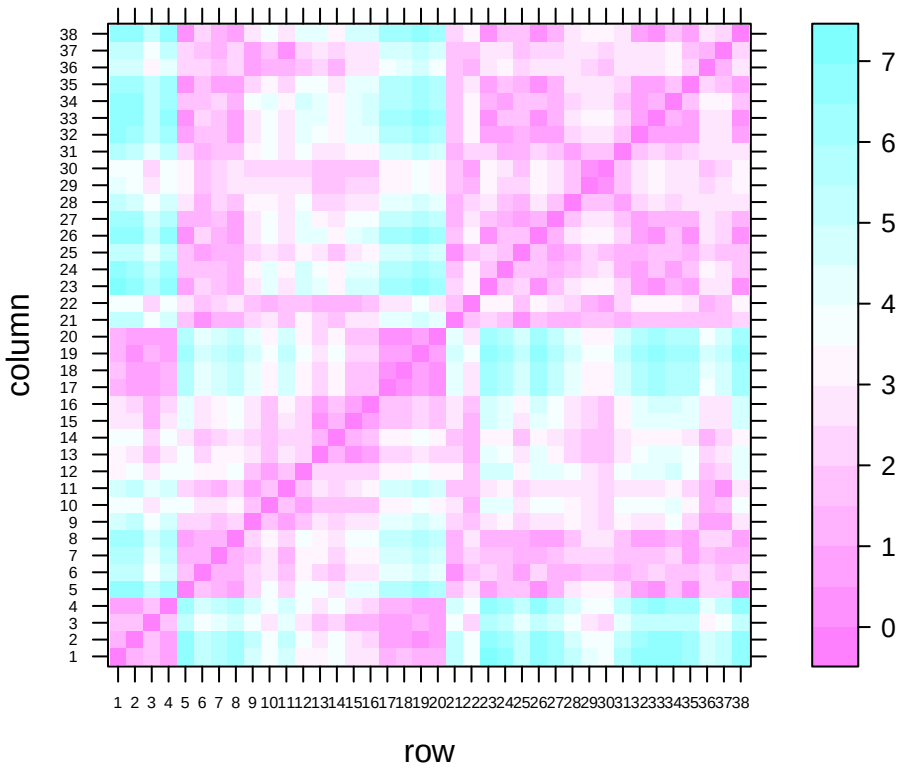
```
car$Car[clus != scar.pam$clustering]
```

```
## [1] Chevy Citation Olds Omega
## 38 Levels: AMC Concord D/L AMC Spirit Audi 5000 ... VW Scirocco
```

4. Finding block structures in the distance matrix

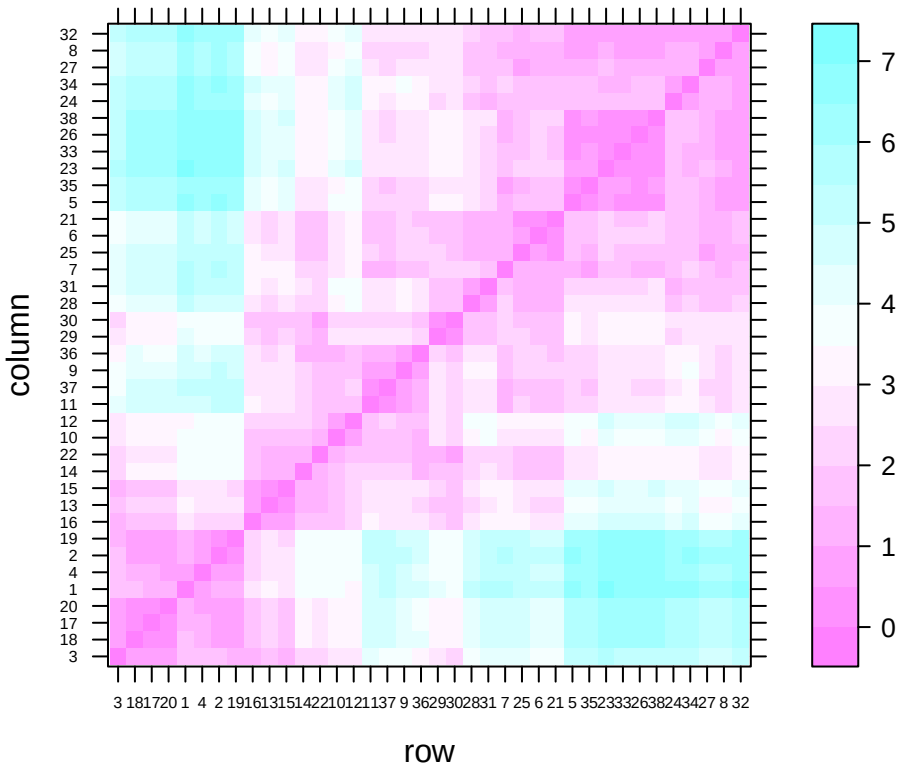
The `levelplot()` function from the `lattice` library produces a colorful visualization of a data matrix. Below you can see a visualization of the distance matrix obtained for the clustering methods.

```
library(lattice)
levelplot(as.matrix(car.dist),scales=list( y=list(cex=.5), x=list(cex=.5)))
```



From the above representation it is not clear what is the block structure in the data. To figure it out, we can use the `order` output value obtained from `hclust`. This will give us the vector with a permutation of the original observations used for plotting the dendogram obtained from the hierarchical clustering. We use it in the plot below and now we can have a better idea of the block structures.

```
levelplot(as.matrix(car.dist)[car.hclust$order, car.hclust$order],
          scales=list( y=list(cex=.5), x=list(cex=.5)))
```



The colorbar denotes distances. Purple values means distances close to zero whereas light blue values denotes high distances. The small region in purple on the lower left indicates that observations {3,18,17,20,1,4,2,19} are very close to each other, separated from the rest of the observations.

Your turn

```
# Compute the distance matrix using Manhattan distance,
# and save the resulting matrix as "dis""

# dis = ?


# Perform a hierachical clustering (based on dis) using single linkage,
# and save this hclust object as "s.hclust"

# s.hclust = ?


# Compare the results of above hierachical clustering with car.hclust by a simple table
```

Credit: Adopted from http://www.stat.berkeley.edu/classes/s133/Cluster2a.html