10 Tree Based Clustering Methods

10.1 Dissimilarity measures

10.1.1 Dissimilarity measures in R

R includes a function, `dist()`, for calculating some of the most basic dissimilarity measures including Euclidean, Minkowski, and Manhattan metrics among others. The typical input to `dist()` is a data frame or matrix and a `method` argument specifying the type of distance measure to use. The `upper` argument specifies whether the upper diagonal of the calculated distance matrix should be printed (by default only the lower diagonal is printed).

To start with let's create a small 4×3 matrix where we can easily calculate the distances between the 4 points by pencil and paper.

```
> # create a 4 x 3 matrix
> z <- matrix(c(0,0,0,
                1,0,0,
                0,1,0,
                0,0,1), 4, 3, byrow=T)
> dist(z)

         1        2        3
2 1.000000
3 1.000000 1.414214
4 1.000000 1.414214 1.414214
```

The default distance measure is Euclidean distance. Let's apply Manhattan distance to the same matrix.

```
> dist(z, method='manhattan')
  1 2 3
2 1
3 1 2
4 1 2 2
```

10.1.2 Dissimilarity Measures in Python

The SciPy module `scipy.spatial.distance` (see the SciPy manual) supports computation of a large number of dissimilarity measures. The primary function is `pdist()` which computes pairwise distances between observations (rows) of an array.

```
>>> import numpy as np
>>> from scipy.spatial import distance
>>> z = np.array ([[0,0,0],
                  [1,0,0],
                  [0,1,0],
                  [0,0,1]], dtype=np.float)
>>> distance.pdist(z)

array([ 1.        ,  1.        ,  1.        ,  1.41421356,  1.41421356,
        1.41421356])
```

SciPy's `pdist()` function returns a condensed representation of the corresponding distance matrix, corresponding to the elements of the lower triangle of the distance matrix. To convert this condensed form to a more standard square matrix we can use the `squareform()` function defined in the same module.

```
>>> from scipy.spatial.distance import pdist, squareform
>>> squareform(pdist(z))

array([[ 0.        ,  1.        ,  1.        ,  1.        ],
       [ 1.        ,  0.        ,  1.41421356,  1.41421356],
       [ 1.        ,  1.41421356,  0.        ,  1.41421356],
       [ 1.        ,  1.41421356,  1.41421356,  0.        ]])
```

As with the R function `dist`, the default distance measure is Euclidean distance. `pdist` can use a wide range of other distance measures as illustrated here:

```
 >>> if true:
        print "Cityblock (Manhattan) distance: "
        print squareform(pdist(z, "cityblock"))
        print
        print "Hamming distance: "
        print squareform(pdist(z, "hamming"))
        print
        print "Chebyshev distance: "
        print squareform(pdist(z, "chebyshev"))

Cityblock (Manhattan) distance:
[[ 0.  1.  1.  1.]
 [ 1.  0.  2.  2.]
 [ 1.  2.  0.  2.]
 [ 1.  2.  2.  0.]]

Hamming distance:
[[ 0.          0.33333333  0.33333333  0.33333333]
 [ 0.33333333  0.          0.66666667  0.66666667]
 [ 0.33333333  0.66666667  0.          0.66666667]
 [ 0.33333333  0.66666667  0.66666667  0.        ]]

Chebyshev distance:
[[ 0.  1.  1.  1.]
 [ 1.  0.  1.  1.]
 [ 1.  1.  0.  1.]
 [ 1.  1.  1.  0.]]
```

10.2 Hierarchical Clustering

10.2.1 Hierarchical Clustering in R

The function `hclust()` provides a simple mechanism for carrying out standard hierarchical clustering in R. The `method` argument determines the group distance function used (single linkage, complete linkage, average, etc.).

The input to `hclust()` is a dissimilarity matrix as computed by the `dist()` function discussed above. If you have a set of distance data that was produced by some other means you can convert an arbitrary square matrix to a distance object by applying the `as.dist()` function to the matrix.

```
> iris.data <- subset(iris, select=-Species)
> iris.cl <- hclust(dist(iris.data), method='single')
> plot(iris.cl) # plot a dendrogram
# let's improve the look a little bit
> plot(iris.cl, labels=iris$Species, cex=0.7)
> # use neg. values of hang to make labels on leaves line up
> plot(iris.cl, labels=iris$Species,  hang=-0.1, cex=0.7)
```

Other functions of interested related to dendrograms include `cuttree()` for cutting the tree at a specified height (or number of groups) and `identify()` for graphically highlighting a cluster of interest in a dendrogram.

```
> plot(iris.cl, labels=iris$Species, cex=0.7)

# IMPORTANT NOTE: the identify fxn doesn't work in R-Studio
> interesting.cluster <- identify(iris.cl) # use left-mouse to choose,
    right-mouse to stop choosing
> interesting.cluster
# [output ommitted]
```

Fancy formatting of dendrogram plots in R is awkward. You need to use the `plot()` function in combination with the `as.dendrogram()` function to access many options. See the help for 'dendrogram' in R for a discussion of options and type `example(dendrogram)` to see some possibilities. A few of them are illustrated here:

```
> plot(as.dendrogram(iris.cl)) # contrast this with plot(iris.cl)
> plot(as.dendrogram(iris.cl), horiz=T) # draw horizontally
> # here's one way to change the labels
> iris.cl$labels <- iris$Species
> levels(iris.cl$labels) <- factor(c("S","Ve","Vi"))
> iris.dend <- as.dendrogram(iris.cl)
> plot(iris.dend)
```

The `heatmap()` function combines a false color image of a matrix with a dendrogram. Here's we apply it to the yeast-subnetwork data set from previous weeks.

```
> yeast <- read.delim('yeast-subnetwork-clean.txt')
> ymap <- heatmap(as.matrix(yeast), labRow=NA) # suppress the numerous row
    labels
> ymap <- heatmap(as.matrix(yeast),labRow=rownames(yeast)) # w/row labels,
    kinda messy
```

The R package `cluster` provides some slightly fancier clustering routines. The basic agglomerative clustering methods in `cluster` are accessed via the function `agnes()`

Compare the results of different hierarchical clustering methods (single linkage, complete linkage, etc.) as applied to the iris data set using the hclust() or agnes() functions. For single and average linkage use both Euclidean and Manhattan distance as the dissimilarity measures.

10.2.2 Hierarchical clustering in Python

The `scipy` library provide a variety of hierarchical clustering routines for Python. These are found in the module `scipy.cluster.hierarchy`. The clustering routines take as input an array giving the pairwise distances between the objects you want to cluster, of the type returned by the `pdist()` function discussed above. In our first example we will carry out single-linkage clustering using Euclidean distance as our dissimilarity measures.

```
In [6]: import numpy as np
In [7]: iris = np.loadtxt('iris.txt',skiprows=1,usecols=range(4))
In [8]: iris.shape
Out[8]: (150, 3)

In [9]: import scipy.spatial.distance as dist
In [10]: d = dist.pdist(iris, 'euclidean')

In [11]: d.shape
Out[11]: (11175,)

In [12]: (150*149)/2  # check number of pairs of specimens
Out[12]: 11175

In [13]: import scipy.cluster.hierarchy as hier
In [14]: ilink = hier.linkage(d)
In [15]: dendro = hier.dendrogram(ilink)
In [16]: dendro = hier.dendrogram(ilink, color_threshold=0.5) # colors the
    subtrees at a different threshold

# get species names from data file
In [17]: species = np.loadtxt('iris.txt', dtype=str, skiprows=1, usecols
    =[4])

# redraw dendrogram w/species names as labels
# root of tree to the left
```

```
In [18]: dendro = hier.dendrogram(ilink, color_threshold=0.5,labels=species
    , orientation='right', leaf_font_size=10)
```

And here's the equivalent version using city block (i.e. Manhattan) distance and UPGMA.

```
In [26]: d2 = dist.pdist(iris, 'cityblock')
In [27]: iupgma = hier.average(d2)
In [30]: dendro2 = hier.dendrogram(iupgma, labels=species, orientation='
    right', leaf_font_size=10)
```

See the SciPy documentation for the full details on scipy.cluster.hierarchy.

10.3 Minimum Spanning Trees

10.3.1 MST in R

A number of R packages including minimum spanning tree implementations. We'll use the implementation in a pacakge called `vegan`. Several others packages, including `ape` also have minimum spanning tree functions.

The `spantree()` function `vegan` takes a dissimilarity matrix as its input and returns a `spantree` object that includes information about the edges (links) in the MST, and the corresponding nodes labels.

We'll apply the MST to a matrix of pairwise travel distances between pairs of American cities, found in the data set `cities.txt` (see the course website).

```
> cities <- read.table('cities.txt')
> typeof(cities)
[1] "list"
> cities <- as.matrix(cities)
> cities
     BOS  CHI   DC  DEN   LA  MIA   NY  SEA   SF
BOS    0  963  429 1949 2979 1504  206 2976 3095
CHI  963    0  671  996 2054 1329  802 2013 2142
DC   429  671    0 1616 2631 1075  233 2684 2799
DEN 1949  996 1616    0 1059 2037 1771 1307 1235
LA  2979 2054 2631 1059    0 2687 2786 1131  379
MIA 1504 1329 1075 2037 2687    0 1308 3273 3053
NY   206  802  233 1771 2786 1308    0 2815 2934
SEA 2976 2013 2684 1307 1131 3273 2815    0  808
SF  3095 2142 2799 1235  379 3053 2934  808    0

> library(vegan) # install ape first if necessary
> city.mst <- spantree(as.dist(cities))
> city.mst
$kid
[1] 3 7 2 4 3 1 9 5

$dist
```

```
[1]  671  233  996 1059 1075  206  808  379

$labels
[1] "BOS" "CHI" "DC"  "DEN" "LA"  "MIA" "NY"  "SEA" "SF"

$call
spantree(d = as.dist(cities))

attr(,"class")
[1] "spantree"
```

It is straightforward to create a plot from the `spantree` object, as shown below. We'll discuss in the next lecture how the the 2D geometry of the points is chosen, and what the output such as "stress" means.

```
> plot(city.mst, type='t')
Initial stress        : 0.00632
stress after  10 iters: 0.00074, magic = 0.500
stress after  20 iters: 0.00074, magic = 0.500
```

10.3.2 MST in Python

The SciPy library `scipy.sparse.csgraph` has an MST implementation. This library has a variety of "sparse graph" routines for dealing with mathematical and statistical analysis of graph structures. We can easily adopt it for calculating our simple MST.

First, let's read in the `cities.txt` data set.

```
>>> import pandas as pd
>>> cities = pd.read_table('cities.txt', index_col=0, sep="\s*")
>>> cities
      BOS   CHI    DC   DEN    LA   MIA    NY   SEA    SF
BOS     0   963   429  1949  2979  1504   206  2976  3095
CHI   963     0   671   996  2054  1329   802  2013  2142
DC    429   671     0  1616  2631  1075   233  2684  2799
DEN  1949   996  1616     0  1059  2037  1771  1307  1235
LA   2979  2054  2631  1059     0  2687  2786  1131   379
MIA  1504  1329  1075  2037  2687     0  1308  3273  3053
NY    206   802   233  1771  2786  1308     0  2815  2934
SEA  2976  2013  2684  1307  1131  3273  2815     0   808
SF   3095  2142  2799  1235   379  3053  2934   808     0
```

Above, you'll notice the use of the `sep` argument to `read_table()`. Here we're passing a 'regular expression' indicating that columns in the cities data set are separated by one or more spaces. Once we've read in the data we can put it in the form expected by the `minimum_spanning_tree` function in `scipy.sparse.graph`.

```
>>> import numpy as np
>>> from scipy.sparse import csgraph
>>> from scipy.sparse import csr_matrix
```

```
# the csgraph matrices expect upper triangular inputs
# so use the numpy triu function
>>> np.triu(cities.values)
array([[   0,  963,  429, 1949, 2979, 1504,  206, 2976, 3095],
       [   0,    0,  671,  996, 2054, 1329,  802, 2013, 2142],
       [   0,    0,    0, 1616, 2631, 1075,  233, 2684, 2799],
       [   0,    0,    0,    0, 1059, 2037, 1771, 1307, 1235],
       [   0,    0,    0,    0,    0, 2687, 2786, 1131,  379],
       [   0,    0,    0,    0,    0,    0, 1308, 3273, 3053],
       [   0,    0,    0,    0,    0,    0,    0, 2815, 2934],
       [   0,    0,    0,    0,    0,    0,    0,    0,  808],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0]])

>>> cities_csr = csr_matrix(np.triu(cities.values))
# the compressed sparse row format is not viewer friendly
>>> cities_csr
<9x9 sparse matrix of type '<type 'numpy.int64'>'
        with 36 stored elements in Compressed Sparse Row format>

# calculate the minimum spanning tree
>>> mst_csr = csgraph.minimum_spanning_tree(cities_csr)

# turn the CSR format into something we can actually look at
>>> mst_csr.toarray().astype(int)
array([[   0,    0,    0,    0,    0,    0,  206,    0,    0],
       [   0,    0,  671,  996,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0, 1075,  233,    0,    0],
       [   0,    0,    0,    0, 1059,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,  379],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0],
       [   0,    0,    0,    0,    0,    0,    0,    0,  808],
       [   0,    0,    0,    0,    0,    0,    0,    0,    0]])

# for convenience let's turn it back into a DataFrame
>>> mstDF = pd.DataFrame(mst_csr.toarray().astype(int), index=cities.index,
     columns=cities.columns)
>>> mstDF
     BOS  CHI   DC  DEN    LA   MIA   NY  SEA   SF
BOS    0    0    0    0     0     0  206    0    0
CHI    0    0  671  996     0     0    0    0    0
DC     0    0    0    0     0  1075  233    0    0
DEN    0    0    0    0  1059     0    0    0    0
LA     0    0    0    0     0     0    0    0  379
MIA    0    0    0    0     0     0    0    0    0
NY     0    0    0    0     0     0    0    0    0
SEA    0    0    0    0     0     0    0    0  808
SF     0    0    0    0     0     0    0    0    0
```

Compare the edges (and corresponding distances) to the `$dist` list in `spantree` object we calculated in R. We'll explore appropriate approaches for drawing the MST next week after we discussion multidimensional scaling methodds.

10.4 Neighbor Joining Trees

10.4.1 Neighbor joining in R

The package ape provides an implementation of neighbor joining in R (and many other useful phylogenetic methods). Here's a couple of examples of using neighbor joining taken from the ape documentation:

```
> library(ape) # install ape if need be

### From Saitou and Nei (1987, Table 1):
> x <- c(7, 8, 11, 13, 16, 13, 17, 5, 8, 10, 13, 10, 14, 5, 7, 10, 7, 11,
    8, 11, 8, 12, 5, 6, 10, 9, 13, 8)
> M <- matrix(0, 8, 8)

# create a symmetric matrix by filling upper and lower triangles
# of the matrix M
> M[row(M) > col(M)] <- x
> M[row(M) < col(M)] <- x
> rownames(M) <- colnames(M) <- 1:8
> tree <- nj(M)
> plot(tree, "u")

### a less theoretical example
> ?woodmouse  # check out the info about the
            # woodmouse data set in the ape package
> data(woodmouse)
> dist <- dist.dna(woodmouse) # see the help on the dist.dna fxn
> tree.mouse <- nj(dist)
> plot(tree.mouse)
```

Assignment 10.1

Harding and Sokal (1998; PNAS 85:9370-9372; see course wiki) used hierarhical clustering analysis and minimum spanning trees to explore the relationship between European language families as measured by genetic distances among the people who speak those languages. The classification they derived at large reflects geographic proximity but there are some language families that have distant genetic relationships to their geographic neighbors.

Harding and Sokal provide a table of genetic distances that they used in their analyses. Use R to reconstruct the cluster analysis they report (Fig. 1) and repeat this analysis using neighbor joining. In a similar manner recreate the MST shown in their figure 2.

Submit your code as an R knitr document, and include a brief paragraph describing what differences, if any, you found in your re-analysis of Harding and Sokal data. Are these differences significant (i.e. do they change your interpretation of the data)?