Cluster Analysis through Combinatorial Optimization

Contents

List of Tables

List of Figures

Chapter 1

Preface

1.1 The Methods

A broad definition of clustering can be given as the search for homogeneous groupings of objects based on some type of available data. There are two common such tasks now discussed in (almost) all multivariate analysis texts and implemented in the commercially available behavioral and social science statistical software suites: hierarchical clustering and the K-means partitioning of some set of objects. We begin with brief reviews of these topics using several illustrative data sets that are carried along throughout the monograph for numerical illustration. Later chapters will develop hierarchical clustering through least-squares and the characterizing notion of an ultrametric; K-means partitioning is generalized by rephrasing as an optimization problem of subdividing a given proximity matrix.

The MATLAB computational environment is relied on to effect our analyses, using the Statistical Toolbox, for example, to carry out the common hierarchical clustering and K-means methods, and our own open-source MATLAB M-files when the extensions go beyond what is currently available commercially (the latter are freely available as a MATLAB Toolbox from

`http://cda.psych.uiuc.edu/clusteranalysis_mfiles_revised`

1.2 The Data

1.2.1 A Proximity Matrix for Illustrating Hierarchical Clustering: Agreement Among Supreme Court Justices in the Rehnquist Court

On Saturday, July 2, 2005, the lead headline in *The New York Times* read as follows: "O'Connor to Retire, Touching Off Battle Over Court." Opening the story attached to the headline, Richard W. Stevenson wrote, "Justice Sandra Day O'Connor, the first woman to serve on the United States Supreme Court and a critical swing vote on abortion and a host of other divisive social issues, announced Friday that she is retiring, setting up a tumultuous fight over her successor." Our interests are in the data set also provided by the *Times* that day, quantifying the (dis)agreement among the Supreme Court justices during the decade they had been together. We give this in Table 1.1 in the form of the percentage of non-unanimous cases in which the justices *dis*agree, from the 1994/95 term through 2003/04 (known as the Rehnquist Court).[1]

The dissimilarity matrix (in which larger entries reflect less similar justices) is listed in the same row and column order as the *Times* data set, with the justices obviously ordered from "liberal" to "conservative":

1: John Paul Stevens (St)

[1]For example, the value of .85 between Stevens (St) and Thomas (Th) means that these two justices disagreed on 85% of the non-unanimous cases heard by the Rehnquist Court.

	St	Br	Gi	So	Oc	Ke	Re	Sc	Th
1 St	.00	.38	.34	.37	.67	.64	.75	.86	.85
2 Br	.38	.00	.28	.29	.45	.53	.57	.75	.76
3 Gi	.34	.28	.00	.22	.53	.51	.57	.72	.74
4 So	.37	.29	.22	.00	.45	.50	.56	.69	.71
5 Oc	.67	.45	.53	.45	.00	.33	.29	.46	.46
6 Ke	.64	.53	.51	.50	.33	.00	.23	.42	.41
7 Re	.75	.57	.57	.56	.29	.23	.00	.34	.32
8 Sc	.86	.75	.72	.69	.46	.42	.34	.00	.21
9 Th	.85	.76	.74	.71	.46	.41	.32	.21	.00

Table 1.1: Dissimilarities Among Nine Supreme Court Justices.

2: Stephen G. Breyer (Br)
3: Ruth Bader Ginsberg (Gi)
4: David Souter (So)
5: Sandra Day O'Connor (Oc)
6: Anthony M. Kennedy (Ke)
7: William H. Rehnquist (Re)
8: Antonin Scalia (Sc)
9: Clarence Thomas (Th)

We use the Supreme Court data matrix of Table 1.1 for the various illustrations of hierarchical clustering that follow. It will be loaded into a MATLAB environment with the command `load supreme_agree.dat` The `supreme_agree.dat` file is in simple `ascii` form with verbatim contents as follows:

```
 .00  .38  .34  .37  .67  .64  .75  .86  .85
 .38  .00  .28  .29  .45  .53  .57  .75  .76
 .34  .28  .00  .22  .53  .51  .57  .72  .74
 .37  .29  .22  .00  .45  .50  .56  .69  .71
 .67  .45  .53  .45  .00  .33  .29  .46  .46
 .64  .53  .51  .50  .33  .00  .23  .42  .41
 .75  .57  .57  .56  .29  .23  .00  .34  .32
 .86  .75  .72  .69  .46  .42  .34  .00  .21
 .85  .76  .74  .71  .46  .41  .32  .21  .00
```

1.2.2 A Data Set for Illustrating K-Means Partitioning: The Famous 1976 Blind Tasting of French and California Wines

In the Bicentennial year for the United States of 1976, an Englishman, Steven Spurrier, and his American partner, Patricia Gallagher, hosted a blind wine tasting in Paris that compared California cabernet from Napa Valley and French cabernet from Bordeaux. Besides Spurrier and Gallagher, nine other judges were notable French wine connoisseurs (the raters are listed below). The six California and four French wines are also identified below with the ratings given in Table 1.2 (from 0 to 20 with higher scores being "better"). The overall conclusion is that Stag's Leap, a U.S. offering, is the winner. Our concern later will be in clustering the wines through the K-means procedure.[2]

Tasters:

1: Pierre Brejoux, Institute of Appellations of Origin
2: Aubert de Villaine, Manager, Domaine de la Romanée-Conti
3: Michel Dovaz, Wine Institute of France
4: Patricia Gallagher, L'Académie du Vin
5: Odette Kahn, Director, *Review of French Wines*
6: Christian Millau, *Le Nouveau Guide* (restaurant guide)
7: Raymond Oliver, Owner, Le Grand Vefour
8: Steven Spurrier, L'Académie du Vin
9: Pierre Tart, Owner, Chateau Giscours
10: Christian Vanneque, Sommelier, La Tour D'Argent
11: Jean-Claude Vrinat, Taillevent

Cabernet Sauvignons:

A: Stag's Leap 1973 (US)
B: Château Mouton Rothschild 1970 (F)
C: Château Montrose 1970 (F)
D: Château Haut Brion 1970 (F)
E: Ridge Monte Bello 1971 (US)
F: Château Léoville-Las-Cases 1971 (F)

[2]For those familiar with late 1950's TV, one can hear Sergeant Preston of the Yukon exclaiming "sacré bleu", and wrapping up with, "Well King, this case is closed".

	Taster										
Wine	1	2	3	4	5	6	7	8	9	10	11
A (US)	14	15	10	14	15	16	14	14	13	16.5	14
B (F)	16	14	15	15	12	16	12	14	11	16	14
C (F)	12	16	11	14	12	17	14	14	14	11	15
D (F)	17	15	12	12	12	13.5	10	8	14	17	15
E (US)	13	9	12	16	7	7	12	14	17	15.5	11
F (F)	10	10	10	14	12	11	12	12	12	8	12
G (US)	12	7	11.5	17	2	8	10	13	15	10	9
H (US)	14	5	11	13	2	9	10	11	13	16.5	7
I (US)	5	12	8	9	13	9.5	14	9	12	3	13
J (US)	7	7	15	15	5	9	8	13	14	6	7

Table 1.2: Taster Ratings Among Ten Cabernets.

G: Heitz “Martha’s Vineyard” 1970 (US)
H: Clos du Val 1972 (US)
I: Mayacamas 1971 (US)
J: Freemark Abbey 1969 (US)

Chapter 2

Hierarchical Clustering (Old School)

To characterize the basic problem posed by hierarchical clustering somewhat more formally, suppose S is a set of n objects, $\{O_1, \ldots, O_n\}$ (for example, in line with the two data sets just given, the objects could be supreme court justices, wines, or tasters (e.g., raters or judges)). Between each pair of objects, O_i and O_j, a symmetric proximity measure, p_{ij}, is given or possibly constructed that we assume (from now on) has a dissimilarity interpretation; these values are collected into an $n \times n$ proximity matrix $\mathbf{P} = \{p_{ij}\}_{n \times n}$, such as the 9×9 example given in Table 1.1 among the supreme court justices. Any hierarchical clustering strategy produces a sequence or hierarchy of partitions of S, denoted $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$, from the information present in $\mathbf{P}$. In particular, the (disjoint) partition $\mathcal{P}_0$ contains all objects in separate classes, $\mathcal{P}_{n-1}$ (the conjoint partition) consists of one all-inclusive object class, and $\mathcal{P}_{k+1}$ is defined from $\mathcal{P}_k$ by uniting a single pair of subsets in $\mathcal{P}_k$.

Generally, the two subsets chosen to unite in defining $\mathcal{P}_{k+1}$ from $\mathcal{P}_k$ are those that are "closest", with the characterization of this latter term specifying the particular hierarchical clustering method used. We mention three of the most common options for this notion of closeness:

(a) complete-link: the maximum proximity value attained for pairs of objects within the union of two sets (thus, we minimize the maximum link [or the subset "diameter"]);

(b) single-link: the minimum proximity value attained for pairs of objects, where the two objects from the pair belong to the separate classes (thus, we minimize the minimum link);

(c) average-link: the average proximity over pairs of objects defined across the separate classes (thus, we minimize the average link).

We generally suggest that the complete-link criterion be the default selection for the task of hierarchical clustering when done in the traditional agglomerative way that starts from $\mathcal{P}_0$ and proceeds step-by-step to $\mathcal{P}_{n-1}$. A reliance on single-link tends to produce "straggly" clusters that are not very internally homogeneous nor substantively interpretable; the average-link choice seems to produce results that are the same as or very similar to the complete-link criterion but relies on more information from the given proximities; complete-link depends only on the rank-order of the proximities.[1]

A complete-link clustering of the `supreme_agree.dat` data set is given by the MATLAB recording below along with the displayed dendrogram in Figure 2.1. (The later dendrogram is drawn directly from the MATLAB Statistical Toolbox routines except for our added two-letter labels for the justices [referred to as "terminal" nodes in the dendrogram], and the numbering of the "internal" nodes from 10 to 17 that represent the new subsets formed in the hierarchy.) The `squareform.m` M-function from the Statistics Toolbox changes a square proximity matrix with zeros along the main diagonal to one in vector form that can be used in the main clustering routine, `linkage.m`. The results of the complete-link clustering are given by the 8×3 matrix (`supreme_agree_clustering`), indicating how the objects (labeled from 1 to 9) and clusters (labeled 10 through 17) are formed and at what level. Here, the levels are the maximum proximities (or diameters) for the newly constructed subsets as the hierarchy is generated. These newly formed clusters (generally, $n-1$ in number) are labeled in Figure 2.1 along with the calibration on the vertical axis as to when they are formed.

The results could also be given as a sequence of partitions:

[1]As we anticipate from later discussion, the average-link criterion has some connections with rephrasing hierarchical clustering as a least-squares optimization task in which an ultrametric (to be defined) is fit to the given proximity matrix. The average proximities between subsets characterize the fitted values.

Partition	Level Formed
{{Sc,Th,Oc,Ke,Re,St,Br,Gi,So}}	.86
{{Sc,Th,Oc,Ke,Re},{St,Br,Gi,So}}	.46
{{Sc,Th},{Oc,Ke,Re},{St,Br,Gi,So}}	.38
{{Sc,Th},{Oc,Ke,Re},{St},{Br,Gi,So}}	.33
{{Sc,Th},{Oc},{Ke,Re},{St},{Br,Gi,So}}	.29
{{Sc,Th},{Oc},{Ke,Re},{St},{Br},{Gi,So}}	.23
{{Sc,Th},{Oc},{Ke},{Re},{St},{Be},{Gi,So}}	.22
{{Sc,Th},{Oc},{Ke},{Re},{St},{Br},{Gi},{So}}	.21
{{Sc},{Th},{Oc},{Ke},{Re},{St},{Br},{Gi},{So}}	—

```
>> load supreme_agree.dat
>> supreme_agree

supreme_agree =

         0    0.3800    0.3400    0.3700    0.6700    0.6400    0.7500    0.8600    0.8500
    0.3800         0    0.2800    0.2900    0.4500    0.5300    0.5700    0.7500    0.7600
    0.3400    0.2800         0    0.2200    0.5300    0.5100    0.5700    0.7200    0.7400
    0.3700    0.2900    0.2200         0    0.4500    0.5000    0.5600    0.6900    0.7100
    0.6700    0.4500    0.5300    0.4500         0    0.3300    0.2900    0.4600    0.4600
    0.6400    0.5300    0.5100    0.5000    0.3300         0    0.2300    0.4200    0.4100
    0.7500    0.5700    0.5700    0.5600    0.2900    0.2300         0    0.3400    0.3200
    0.8600    0.7500    0.7200    0.6900    0.4600    0.4200    0.3400         0    0.2100
    0.8500    0.7600    0.7400    0.7100    0.4600    0.4100    0.3200    0.2100         0

>> supreme_agree_vector = squareform(supreme_agree)

supreme_agree_vector =

  Columns 1 through 9

    0.3800    0.3400    0.3700    0.6700    0.6400    0.7500    0.8600    0.8500    0.2800

  Columns 10 through 18

    0.2900    0.4500    0.5300    0.5700    0.7500    0.7600    0.2200    0.5300    0.5100

  Columns 19 through 27

    0.5700    0.7200    0.7400    0.4500    0.5000    0.5600    0.6900    0.7100    0.3300

  Columns 28 through 36

    0.2900    0.4600    0.4600    0.2300    0.4200    0.4100    0.3400    0.3200    0.2100

>> supreme_agree_clustering = linkage(supreme_agree_vector,'complete')

supreme_agree_clustering =

    8.0000    9.0000    0.2100
    3.0000    4.0000    0.2200
    6.0000    7.0000    0.2300
    2.0000   11.0000    0.2900
    5.0000   12.0000    0.3300
    1.0000   13.0000    0.3800
   10.0000   14.0000    0.4600
   15.0000   16.0000    0.8600

>> dendrogram(supreme_agree_clustering)
```

Substantively, the interpretation of the complete-link hierarchical clustering result is very clear. There are three "tight" dyads in {Sc,Th}, {Gi,So}, and {Ke,Re}; {Oc} joins with {Ke,Re}, and {Br} with {Gi,So} to form, respectively, the "moderate" conservative and liberal clusters. {St} then joins with {Br,Gi,So} to form the liberal-left four-object cluster; {Oc,Ke,Re} unites with the dyad of {Sc,Th} to form the five-object conservative-right. All of this is not very surprising given the enormous literature on the Rehnquist Court. What is satisfying from a data analyst's perspective is how very clear the interpretation is, based on the dendrogram of Figure 2.1 constructed empirically from the data of Table 1.1.[2]

2.1 Ultrametrics

Given the partition hierarchies from any of the three criteria mentioned (complete-, single-, or average-link), suppose we place the values for when the new subsets were formed (i.e., the maximum, minimum, or average proximity between the united subsets) into an $n \times n$ matrix $\mathbf{U}$ with rows and columns relabeled to conform with the order of display for the terminal nodes in the dendrogram. For example, Table 2.1 provides the complete-link results for $\mathbf{U}$ with an overlay partitioning of the matrix to indicate the hierarchical clustering. In general, there are $n-1$ distinct nonzero values that define the levels at which the $n-1$ new subsets are formed in the hierarchy; thus, there are typically $n-1$ distinct nonzero values present in a matrix $\mathbf{U}$ characterizing the identical blocks of matrix entries between subsets united in forming the hierarchy.

[2]We note that the terminal node order in Figure 2.1 does not conform to the Justice ordering of Table 1.1. Until recently, there was no option to impose such an order on the dendrogram function in MATLAB. Now, however, there is an option to the dendrogram function that could impose such an order. For the dendrogram of Figure 2.1, use
dendrogram(supreme_agree_clustering,'reorder',[1 2 3 4 5 6 7 8 9])

	Sc	Th	Oc	Ke	Re	St	Br	Gi	So
8 Sc	.00	.21	.46	.46	.46	.86	.86	.86	.86
9 Th	.21	.00	.46	.46	.46	.86	.86	.86	.86
5 Oc	.46	.46	.00	.33	.33	.86	.86	.86	.86
6 Ke	.46	.46	.33	.00	.23	.86	.86	.86	.86
7 Re	.46	.46	.33	.23	.00	.86	.86	.86	.86
1 St	.86	.86	.86	.86	.86	.00	.38	.38	.38
2 Br	.86	.86	.86	.86	.86	.38	.00	.29	.29
3 Gi	.86	.86	.86	.86	.86	.38	.29	.00	.22
4 So	.86	.86	.86	.86	.86	.38	.29	.22	.00

Table 2.1: Ultrametric Values (Based on Subset Diameters) Characterizing the Complete-Link Hierarchical Clustering of Table 1.1

Given a matrix such as **U**, the partition hierarchy can be retrieved immediately along with the levels at which the new subsets were formed. For example, Table 2.1, incorporating subset diameters (i.e., the maximum proximity within a subset) to characterize when formation takes place, can be used to obtain the dendrogram and the explicit listing of the partitions in the hierarchy. In fact, any (strictly) monotone (i.e., order preserving) transformation of the $n-1$ distinct values in such a matrix **U** would serve the same retrieval purposes. Thus, as an example, we could replace the eight distinct values in Table 2.1, (.21, .22, .23, .29, .33, .38, .46, .86), by the simple integers, (1, 2, 3, 4, 5, 6, 7, 8), and the topology (i.e., the branching pattern) of the dendrogram and the partitions of the hierarchy could be reconstructed. Generally, we characterize a matrix **U** that can be used to retrieve a partition hierarchy in this way as an *ultrametric*:

A matrix $\mathbf{U} = \{u_{ij}\}_{n \times n}$ is ultrametric if for every triple of subscripts, i, j, and k, $u_{ij} \leq \max(u_{ik}, u_{kj})$; or equivalently (and much more understandably), among the three terms, u_{ij}, u_{ik}, and u_{kj}, the largest two values are equal.

As can be verified, Table 2.1 (or any strictly monotone transformation of its entries) is ultrametric; it can be used to retrieve a partition hierarchy, and the ($n-1$ distinct nonzero) values in **U** define the levels at which the $n-1$ new subsets are formed. The hierarchical clustering task will be characterized in a later chapter as an optimization problem in which we seek to identify a best-fitting ultrametric matrix, say $\mathbf{U}^*$, for a given proximity matrix $\mathbf{P}$.

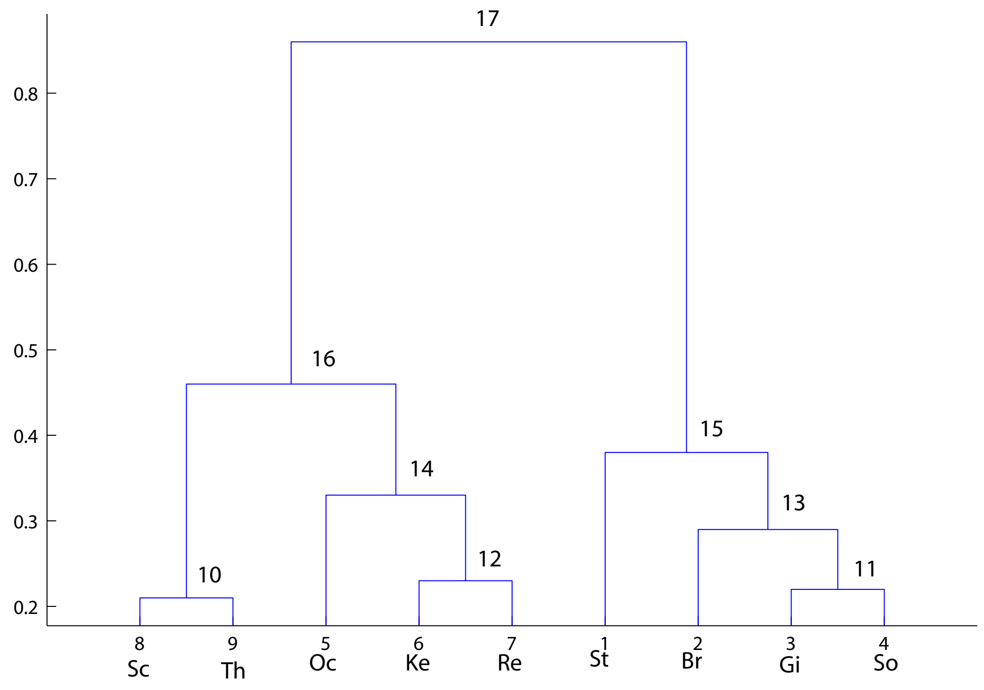


Figure 2.1: Dendrogram Representation for the Complete-link Hierarchical Clustering of the Supreme Court Proximity Matrix

Chapter 3

K-Means Clustering (or Partitioning) (Old School)

The data on which a K-means clustering is defined will be assumed in the form of a usual $n \times p$ data matrix, $\mathbf{X} = \{x_{ij}\}$, for n subjects over p variables. (We will use the example of Table 1.2, where there are $n = 10$ wines (subjects) and $p = 11$ tasters (variables). Although we will not pursue the notion here, there is typically a duality present in all such data matrices, and attention could be refocused on grouping tasters based on the wines now reconsidered to be the "variables".) If the set $S = \{O_1, \ldots, O_n\}$ defines the n objects to be clustered, we seek a collection of K mutually-exclusive and exhaustive subsets of S, say, $C_1, \ldots, C_K$, that minimizes the sum-of-squared-error (SSE):

$$SSE = \sum_{k=1}^{K} \sum_{O_i \in C_k} \sum_{j=1}^{p} (x_{ij} - m_{kj})^2, \tag{3.1}$$

for $m_{kj} = \frac{1}{n_k} \sum_{O_i \in C_k} x_{ij}$ (the mean in group C_k on variable j), and n_k, the number of objects in C_k. What this represents in the context of the usual univariate analysis-of-variance is a minimization of the within-group sum-of-squares aggregated over the p variables in the data matrix $\mathbf{X}$. We also note that for the most inward expression in (3.1), the term $\sum_{j=1}^{p}(x_{ij} - m_{kj})^2$ represents the squared Euclidean distance between the profile values over the p variables present for object O_i and the variable means (or centroid) within the cluster C_k containing O_i (it is these latter K centroids or mean vectors that lend the common name of K-means).

The typical relocation algorithm would proceed as follows: an initial set of “seeds” (e.g., objects) is chosen and the sum-of-squared-error criterion is defined based on the distances to these seeds. A reallocation of objects to groups is carried out according to minimum distance, and centroids recalculated. The minimum distance allocation and recalculation of centroids is performed until no change is possible — each object is closest to the group centroid to which it is now assigned. Note that at the completion of this stage, the solution will be locally optimal with respect to each object being closest to its group centroid. A final check can be made to determine if any single-object reallocations will reduce the sum-of-squared-error any further; at the completion of this stage, the solution will be locally optimal with respect to (3.1).

We present a verbatim MATLAB session below in which we ask for two to four clusters for the wines using the `kmeans.m` routine from the Statistical Toolbox on the `cabernet_taste.dat` data matrix from Table 1.2. We choose one-hundred random starts (`‘replicates’,100`) by picking two to four objects at random to serve as the initial seeds (`‘start’,‘sample’`). Two local optima were found for the choice of two clusters, but only one for three. The control phrase (`‘maxiter’,1000`) increases the allowable number of iterations; (`‘display’,‘final’`) controls printing the end results for each of the hundred replications; most of this latter output is suppressed to save space and replaced by `...`). The results actually displayed for each number of chosen clusters are the best obtained over the hundred replications with `idx` indicating cluster membership for the n objects; `c` contains the cluster centroids; `sumd` gives the within-cluster sum of object-to-centroid distances (so when the entries are summed, the objective function in (3.1) is generated); `d` includes all the distances between each object and each centroid.

```
>> load cabernet_taste.dat

>> cabernet_taste

cabernet_taste =

   14.0000   15.0000   10.0000   14.0000   15.0000   16.0000   14.0000   14.0000   13.0000   16.5000   14.0000
   16.0000   14.0000   15.0000   15.0000   12.0000   16.0000   12.0000   14.0000   11.0000   16.0000   14.0000
   12.0000   16.0000   11.0000   14.0000   12.0000   17.0000   14.0000   14.0000   14.0000   11.0000   15.0000
   17.0000   15.0000   12.0000   12.0000   12.0000   13.5000   10.0000    8.0000   14.0000   17.0000   15.0000
   13.0000    9.0000   12.0000   16.0000    7.0000    7.0000   12.0000   14.0000   17.0000   15.5000   11.0000
   10.0000   10.0000   10.0000   14.0000   12.0000   11.0000   12.0000   12.0000   12.0000    8.0000   12.0000
   12.0000    7.0000   11.5000   17.0000    2.0000    8.0000   10.0000   13.0000   15.0000   10.0000    9.0000
   14.0000    5.0000   11.0000   13.0000    2.0000    9.0000   10.0000   11.0000   13.0000   16.5000    7.0000
```

```
    5.0000   12.0000    8.0000    9.0000   13.0000    9.5000   14.0000    9.0000   12.0000    3.0000   13.0000
    7.0000    7.0000   15.0000   15.0000    5.0000    9.0000    8.0000   13.0000   14.0000    6.0000    7.0000

>> [idx,c,sumd,d] = kmeans(cabernet_taste,2,'start','sample','replicates',100,'maxiter',1000,'display','final')

2 iterations, total sum of distances = 633.208

3 iterations, total sum of distances = 633.063 ...

 idx =

     2
     2
     2
     2
     1
     2
     1
     1
     2
     1


c =

   11.5000    7.0000   12.3750   15.2500    4.0000    8.2500   10.0000   12.7500   14.7500   12.0000    8.5000
   12.3333   13.6667   11.0000   13.0000   12.6667   13.8333   12.6667   11.8333   12.6667   11.9167   13.8333


sumd =

  181.1875
  451.8750


d =

  329.6406   44.3125
  266.1406   63.3125
  286.6406   27.4792
  286.8906   76.8958
   46.6406  155.8125
  130.3906   48.9792
   12.5156  249.5625
   50.3906  290.6458
  346.8906  190.8958
   71.6406  281.6458

-------------------------------------------------------------------------------------------------------------
>> [idx,c,sumd,d] = kmeans(cabernet_taste,3,'start','sample','replicates',100,'maxiter',1000,'display','final')

3 iterations, total sum of distances = 348.438 ...

idx =

     1
     1
     1
     1
     2
     3
     2
     2
     3
     2


c =

   14.7500   15.0000   12.0000   13.7500   12.7500   15.6250   12.5000   12.5000   13.0000   15.1250   14.5000
   11.5000    7.0000   12.3750   15.2500    4.0000    8.2500   10.0000   12.7500   14.7500   12.0000    8.5000
    7.5000   11.0000    9.0000   11.5000   12.5000   10.2500   13.0000   10.5000   12.0000    5.5000   12.5000


sumd =

  117.1250
  181.1875
   50.1250


d =

   16.4688  329.6406  242.3125
   21.3438  266.1406  289.5625
   34.8438  286.6406  155.0625
   44.4688  286.8906  284.0625
  182.4688   46.6406  244.8125
  132.0938  130.3906   25.0625
  323.0938   12.5156  244.8125
  328.2188   50.3906  357.8125
  344.9688  346.8906   25.0625
  399.5938   71.6406  188.0625

--------------------------------------------------------------------------------------------------------------

>> [idx,c,sumd,d] = kmeans(cabernet_taste,4,'start','sample','replicates',100,'maxiter',1000,'display','final')

3 iterations, total sum of distances = 252.917

3 iterations, total sum of distances = 252.917

4 iterations, total sum of distances = 252.917

3 iterations, total sum of distances = 289.146 ...

idx =

     4
     4
     4
     4
     2
     3
     2
     2
     3
     1


c =

    7.0000    7.0000   15.0000   15.0000    5.0000    9.0000    8.0000   13.0000   14.0000    6.0000    7.0000
   13.0000    7.0000   11.5000   15.3333    3.6667    8.0000   10.6667   12.6667   15.0000   14.0000    9.0000
    7.5000   11.0000    9.0000   11.5000   12.5000   10.2500   13.0000   10.5000   12.0000    5.5000   12.5000
   14.7500   15.0000   12.0000   13.7500   12.7500   15.6250   12.5000   12.5000   13.0000   15.1250   14.5000
```

```
sumd =

         0
   85.6667
   50.1250
  117.1250


d =

  485.2500  309.6111  242.3125   16.4688
  403.0000  252.3611  289.5625   21.3438
  362.0000  293.3611  155.0625   34.8438
  465.2500  259.2778  284.0625   44.4688
  190.2500   30.6111  244.8125  182.4688
  147.0000  156.6944   25.0625  132.0938
   76.2500   23.1111  244.8125  323.0938
  201.2500   31.9444  357.8125  328.2188
  279.2500  401.2778   25.0625  344.9688
         0  127.3611  188.0625  399.5938
```

The separation of the wines into three groups (having objective function value of 348.438) results in the clusters: $\{A, B, C, D\}, \{E, G, H, J\}, \{F, I\}$. Here, $\{A, B, C, D\}$ represents the four absolute "best" wines with the sole U.S. entry of Stag's Leap (A) in this mix; $\{E, G, H, J\}$ are four wines that are rated at the absolute bottom (consistently) for four of the tasters (2,5,6,11) and are *all* U.S. products; the last class, $\{F, I\}$, includes one French and one U.S. label with more variable ratings over the judges. This latter group also coalesces with the best group when only two clusters are sought. From a nonchauvinistic perspective, the presence of the single U.S. offering of Stag's Leap in the "best" group of four (within the three-class solution) does not say very strongly to us that the U.S. has somehow "won".

3.1 K-Means and Matrix Partitioning

The most inward expression in (3.1),

$$\sum_{O_i \in C_k} \sum_{j=1}^{p} (x_{ij} - m_{kj})^2, \tag{3.2}$$

can be interpreted as the sum of the squared Euclidean distances between every object in C_k and the centroid for this cluster. These sums are aggregated, in turn, over k (from 1 to K) to obtain the sum-of-squared-error criterion that we attempt to minimize in K-means clustering by the judicious choice of $C_1, \ldots, C_K$. Alternatively, the expression in (3.2) can be re-expressed as

$$\frac{1}{2n_k} \sum_{O_i, O_{i'} \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2, \tag{3.3}$$

or a quantity equal to the sum of the squared Euclidean distances between all object pairs in C_k divided by twice the number of objects, n_k, in C_k. If we define the proximity, $p_{ii'}$, between any two objects, O_i and $O_{i'}$, over the p variables as the squared Euclidean distance, then (3.3) could be rewritten as

$$\frac{1}{2n_k} \sum_{O_i, O_{i'} \in C_k} p_{ii'}.$$

Or, consider the proximity matrix $\mathbf{P} = \{p_{ii'}\}$ and for any clustering, $C_1, \ldots, C_K$, the proximity matrix can be schematically represented as

$$\begin{array}{c|ccccc|}
 & C_1 & \cdots & C_k & \cdots & C_K \\
\hline
C_1 & \mathbf{P}_{11} & \cdots & \mathbf{P}_{1k} & \cdots & \mathbf{P}_{1K} \\
\vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\
C_k & \mathbf{P}_{k1} & \cdots & \mathbf{P}_{kk} & \cdots & \mathbf{P}_{kK} \\
\vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\
C_K & \mathbf{P}_{K1} & \cdots & \mathbf{P}_{Kk} & \cdots & \mathbf{P}_{KK}
\end{array}$$

where the objects in S have been reordered so each cluster C_k represents a contiguous segment of (ordered objects) and $\mathbf{P}_{kk'}$ is the $n_k \times n_{k'}$ collection of proximities between the objects in C_k and $C_{k'}$. In short, the sum-of-squared-error criterion is merely the sum of proximities in $\mathbf{P}_{kk}$ weighted by $\frac{1}{2n_k}$ and aggregated over k from 1 to K (i.e., the sum of the main diagonal blocks of $\mathbf{P}$). In fact, any clustering evaluated with the sum-of-squared-error criterion could be represented by such a structure defined with a reordered proximity matrix having its rows and columns grouped to contain the contiguous objects in $C_1, \ldots, C_K$.

To give an example of this kind of proximity matrix for our cabernet example, the squared Euclidean distance matrix among the wines is given

Class / Wine	C_1/A	C_1/B	C_1/C	C_1/D	C_2/E	C_2/G	C_2/H	C_2/J	C_3/F	C_3/I
C_1/A	.00	48.25	48.25	86.50	220.00	400.50	394.00	485.25	160.25	374.50
C_1/B	48.25	.00	77.00	77.25	195.25	327.25	320.25	403.00	176.00	453.25
C_1/C	48.25	77.00	.00	131.25	229.25	326.25	410.25	362.00	107.00	253.25
C_1/D	86.50	77.25	131.25	.00	202.50	355.50	305.50	465.25	202.25	416.00
C_2/E	220.00	195.25	229.25	202.50	.00	75.50	102.00	190.25	145.25	394.50
C_2/G	400.50	327.25	326.25	355.50	75.50	.00	79.50	76.25	160.25	379.50
C_2/H	394.00	320.25	410.25	305.50	102.00	79.50	.00	201.25	250.25	515.50
C_2/J	485.25	403.00	362.00	465.25	190.25	76.25	201.25	.00	147.00	279.25
C_3/F	160.25	176.00	107.00	202.25	145.25	160.25	250.25	147.00	.00	100.25
C_3/I	374.50	453.25	253.25	416.00	394.50	379.50	515.50	279.25	100.25	.00

Table 3.1: Squared Euclidean Distances Among Ten Cabernets.

in Table 3.1 with the row and column objects reordered to conform to the three-group K-means clustering. The expression in (3.3) in relation to Table 3.1 would be given as

$$\frac{1}{2n_1}\sum_{O_i,O_{i'}\in C_1} p_{ii'} + \frac{1}{2n_2}\sum_{O_i,O_{i'}\in C_2} p_{ii'} + \frac{1}{2n_3}\sum_{O_i,O_{i'}\in C_3} p_{ii'} =$$

$$\frac{1}{2(4)}(937.00) + \frac{1}{2(4)}(1449.50) + \frac{1}{2(2)}(200.50) =$$

$$117.1250 + 181.1875 + 50.1250 = 348.438$$

This is the same objective function value from (3.1) reported in the verbatim MATLAB output.

Chapter 4

Hierarchical Clustering (New School)

A brief introduction to the two dominant tasks of hierarchical clustering and K-means partitioning have been provided in the previous two chapters. Here, we begin to make several extensions of these ideas to make the analysis techniques generally more useful to the user. In contrast to earlier sections where the cited MATLAB routines were already part of the Statistics Toolbox, the M-files from this chapter on are available (open-source) from the web site:

```
http://cda.psych.uiuc.edu/clusteranalysis_mfiles_revised
```

We provide the help "header" files for all of these M-files in an Appendix to this monograph; these should be generally helpful in explaining both syntax and usage.

4.1 The Least-Squares Finding and Fitting of Ultrametrics

The hierarchical clustering task can be reformulated as locating a best-fitting ultrametric, say $\mathbf{U}^* = \{u^*_{ij}\}$, to the given proximity matrix, $\mathbf{P}$, such that the least-squares criterion

$$\sum_{i<j}(p_{ij} - u^*_{ij})^2 \ ,$$

is minimized. The approach can either be confirmatory (in which we look for the best-fitting ultrametric defined by some monotone transformation of the $n-1$ values making up a fixed ultrametric), or exploratory (where we merely look for the best-fitting ultrametric without any prior constraint as to its form). In both cases, a convenient normalized loss measure is given by the variance-accounted-for (VAF):

$$\mathrm{VAF} = 1 - \frac{\sum_{i<j}(p_{ij} - u^*_{ij})^2}{\sum_{i<j}(p_{ij} - \bar{p})^2} \ ,$$

where $\bar{p}$ is the average off-diagonal proximity value in $\mathbf{P}$. This is directly comparable to the usual VAF measure familiar from multiple regression.

A least-squares approach to identifying good ultrametrics is governed by two M-files, `ultrafit.m` (for confirmatory ultrametric fitting) and `ultrafnd.m` (for exploratory ultrametric finding). The syntaxes for both are as follows:

```
[fit,vaf] = ultrafit(prox,targ)

[find,vaf] = ultrafnd(prox,inperm)
```

Here, `prox` refers to the input proximity matrix; `targ` is of the same size as `prox`, with the same row and column order, and contains values conforming to an ultrametric (e.g., the complete-link ultrametric values of Table 2.1); `inperm` is an input permutation of the n objects that controls the heuristic search process for identifying the ultrametric constraints to impose (this is usually given by the built-in random permutation `randperm(n)`, where `n` is replaced by the actual number of objects; different random starts can be tried in the heuristic search to investigate the distribution of possible local optima); `fit` and `find` refer to the confirmatory or exploratory identified ultrametric matrices, respectively, with the common meaning of variance-accounted-for given to `vaf`.

A MATLAB session using these two functions is reproduced below. The complete-link target ultrametric matrix, `sc_completelink_target.dat`, with the same row and column ordering as `supreme_agree.dat` induces a least-squares confirmatory fitted matrix having VAF of 73.69%. The monotonic function, say $f(\cdot)$, between the values of the fitted and input target matrices can be given as follows: $f(.21) = .21$; $f(.22) = .22$; $f(.23) = .23$; $f(.29) = .2850$; $f(.33) = .31$; $f(.38) = .3633$; $f(.46) = .4017$; $f(.86) = .6405$. Interestingly, an exploratory use of `ultrafnd.m` produces exactly this same result; also, there appears to be only this one local optimum identifiable over many random starts (these results are not explicitly reported here but can be replicated easily by the reader. Thus, at least for this particular data set, the complete-link method produces the optimal [least-squares] branching structure as verified over repeated random initializations for `ultrafnd.m`).

```
>> load sc_completelink_target.dat

>> sc_completelink_target

sc_completelink_target =

         0    0.3800    0.3800    0.3800    0.8600    0.8600    0.8600    0.8600    0.8600
    0.3800         0    0.2900    0.2900    0.8600    0.8600    0.8600    0.8600    0.8600
    0.3800    0.2900         0    0.2200    0.8600    0.8600    0.8600    0.8600    0.8600
    0.3800    0.2900    0.2200         0    0.8600    0.8600    0.8600    0.8600    0.8600
    0.8600    0.8600    0.8600    0.8600         0    0.3300    0.3300    0.4600    0.4600
    0.8600    0.8600    0.8600    0.8600    0.3300         0    0.2300    0.4600    0.4600
    0.8600    0.8600    0.8600    0.8600    0.3300    0.2300         0    0.4600    0.4600
    0.8600    0.8600    0.8600    0.8600    0.4600    0.4600    0.4600         0    0.2100
    0.8600    0.8600    0.8600    0.8600    0.4600    0.4600    0.4600    0.2100         0

>> load supreme_agree.dat;

>> [fit,vaf] = ultrafit(supreme_agree,sc_completelink_target)

fit =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369

>> [find,vaf] = ultrafnd(supreme_agree,randperm(9))

find =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369
```

As noted in an earlier footnote, the ultrametric fitted values obtained through least-squares are actually average proximities of a similar type used in average-link hierarchical clustering. This should not be surprising given that any sum-of-squared deviations of a set of observations from a common value is minimized when that common value is the arithmetic mean. For the monotonic function reported above, the various values are the average proximities between the subsets united in forming the partition hierarchy:

$$.21 = .21;\ .22 = .22;\ .23 = .23;\ .2850 = (.28 + .29)/2;$$

$$.31 = (.33 + .29)/2;\ .3633 = (.38 + .34 + .37)/3;$$

$$.4017 = (.46 + .42 + .34 + .46 + .41 + .32)/6;$$

$$.6405 = (.67 + .64 + .75 + .86 + .85 + .45 + .53 + .57 + .75 + .76 +$$

$$.53 + .51 + .57 + .72 + .74 + .45 + .50 + .56 + .69 + .71)/20.$$

4.2 Order-Constrained Ultrametrics

In identifying a best-fitting ultrametric and displaying it subsequently through a dendrogram, there is a degree of arbitrariness in how the terminal nodes are ordered. If we treat the dendrogram as a "mobile" and allow the internal nodes to act as universal joints with freedom of 360° degree rotation, there are 2^{n-1} equivalent orderings of the terminal nodes (in our example, 2^8 is 256), and none is preferred a priori. To impose some meaning on the terminal node ordering, we provide two routines that either impose a given ordering or look for a "best" one that could be used for display in the exploratory identification of a best-fitting ultrametric. These routines rely on a preliminary identification of a least-squares best-fitting anti-Robinson matrix (an anti-Robinson (AR) matrix is one in which the entries never decrease when moving within the rows or columns away from the main diagonal entries). Treating the fitted AR matrix as the collection of "proximities" in their own right, the process of finding a best-fitting ultrametric is then carried out, producing a dendrogram that is consistently displayable with respect to the constraining order. In effect, we are combining the two (somewhat) different tasks of hierarchical clustering and the seriation of an object set by reordering the rows and columns of **P** to display as closely as possible, a particularly appealing AR gradient in its entries.

To identify a good-fitting (in a least-squares sense) ultrametric that could be displayed consistently with respect to a given fixed order, we provide the M-file, `ultrafnd_confit.m`, and give an application below to the `supreme_agree.dat` data. The input proximity matrix (`prox`) is `supreme_agree.dat`; the permutation that determines the order in which the heuristic optimization strategy seeks the inequality constraints to define the obtained ultrametric is chosen at random (`randperm(9)`); thus, the routine could be rerun to see whether local optima are obtained in identifying the ultrametric (but still constrained by exactly the same object order (`conperm`), given here as the identity (the colon notation, 1:9, can be used generally in MATLAB to produce the sequence, 1 2 3 4 5 6 7 8 9). For output, we provide the ultrametric identified in `find` with VAF of 73.69%. For completeness, the best AR matrix (least-squares) to the input proximity matrix using the same constraining order (`conperm`) is given by `arobprox` with a VAF of 99.55%.

As our computational mechanism for imposing the given ordering on the obtained ultrametric, the best-fitting AR matrix is used as a point of departure. Also, the ultrametric fitted values considered as averages from the original proximity matrix could just as well be calculated directly as averages from the best-fitting AR matrix. The results must be the same due to the best-fitting AR matrix itself being least-squares and therefore constructed using averages from the original proximity matrix.

```
>> load supreme_agree.dat

>> [find,vaf,vafarob,arobprox,vafultra] = ultrafnd_confit(supreme_agree,randperm(9),1:9)

find =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =
    0.7369


vafarob =

    0.9955


arobprox =

         0    0.3600    0.3600    0.3700    0.6550    0.6550    0.7500    0.8550    0.8550
    0.3600         0    0.2800    0.2900    0.4900    0.5300    0.5700    0.7500    0.7600
    0.3600    0.2800         0    0.2200    0.4900    0.5100    0.5700    0.7200    0.7400
    0.3700    0.2900    0.2200         0    0.4500    0.5000    0.5600    0.6900    0.7100
    0.6550    0.4900    0.4900    0.4500         0    0.3100    0.3100    0.4600    0.4600
    0.6550    0.5300    0.5100    0.5000    0.3100         0    0.2300    0.4150    0.4150
    0.7500    0.5700    0.5700    0.5600    0.3100    0.2300         0    0.3300    0.3300
    0.8550    0.7500    0.7200    0.6900    0.4600    0.4150    0.3300         0    0.2100
    0.8550    0.7600    0.7400    0.7100    0.4600    0.4150    0.3300    0.2100         0


vafultra =

    0.7402
```

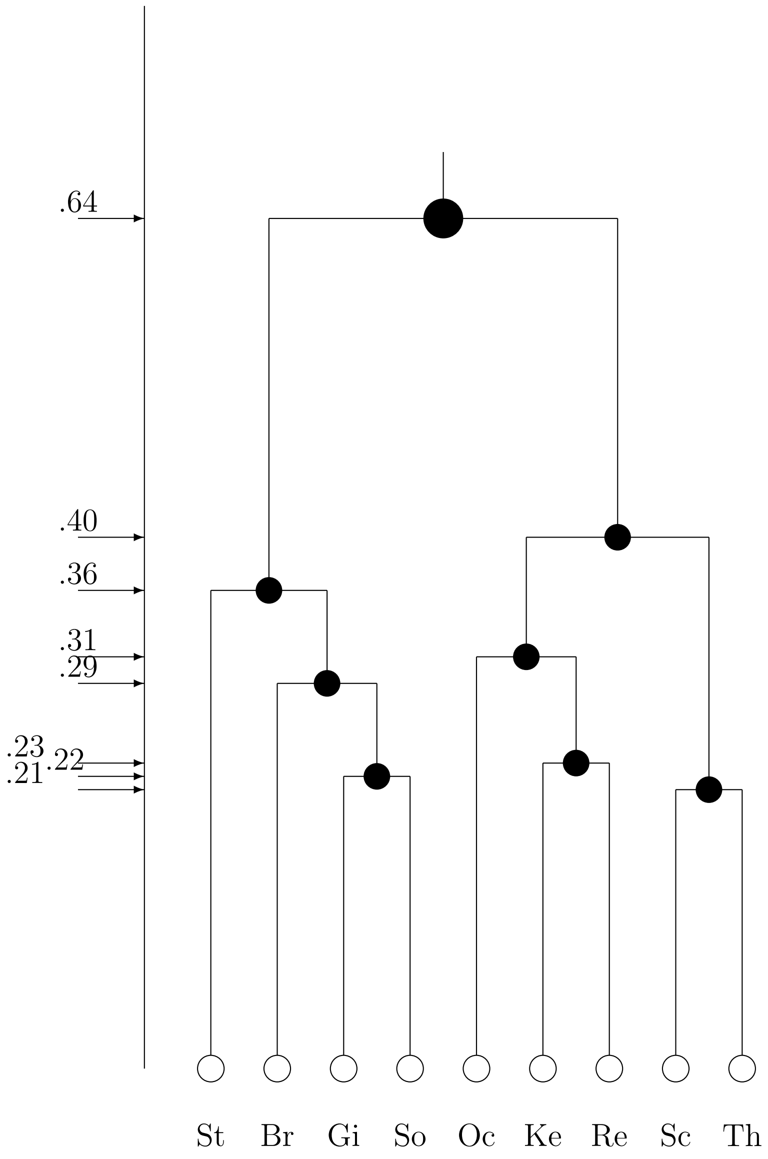
The M-file, `ultrafnd_confnd.m`, carries out the identification of a good initial constraining order, and does not require one to be given a priori. As the syntax below shows (with the three dots indicating the MATLAB continuation command when the line is too long), the constraining order (`conperm`) is provided as an output vector, and constructed by finding a best AR fit to the original proximity input matrix. We note here that the identity permutation would again be retrieved, not surprisingly, as the "best" constraining order.

```
[find,vaf,conperm,vafarob,arobprox,vafultra] = ...
ultrafnd_confnd(prox,inperm)
```

Figure 4.1 illustrates the ultrametric structure graphically as a dendrogram where the terminal nodes now conform explicitly to the "left-to-right" gradient identified using `ultrafnd_confnd.m`, with its inherent meaning over and above the structure implicit in the imposed ultrametric. It represents the object order for the best AR matrix fit to the original proximities, and is identified before we further impose an ultrametric. Generally, the object order chosen for the dendrogram should place similar objects (according to the original proximities) as close as possible. This is very apparent here where the particular (identity) constraining order imposed has an obvious meaning.

We note that Figure 4.1 is not drawn using the `dendrogram.m` routine from MATLAB but was done "by hand" in the LaTeX `picture` environment with an explicit left-to-right order imposed among the justices.

Figure 4.1: A Dendrogram (Tree) Representation for the Ordered-Constrained Ultrametric Described in the Text (Having VAF of 73.69%)



Chapter 5

K-Means Clustering (or Partitioning) (New School)

In observing that the K-means criterion could be reinterpreted through a proximity matrix defined by squared Euclidean distances, it was also noted that the clusters could be represented as contiguous segments of ordered objects in a reordered proximity matrix. We exploit this connection by rephrasing the search for the better (in the sense of hopefully being more substantively interpretable) partitions by imposing a preliminary order on the squared Euclidean proximity matrix; then, for a given number of clusters, a (globally) optimal subdivision is found based on the K-means criterion (the M-file that carries this out is an implementation of an order-constrained dynamic programming (DP) routine that can handle a very large number of objects with guaranteed (order-constrained) optimality for the traditional K-means criterion). It appears that this tandem strategy of finding an order first and then carrying out a K-means subdivision, does well in its generation of substantively interpretable partitions. It's as if we are simultaneously optimizing two objective functions — one that provides a typically good approximate AR ordering for the squared Euclidean distances (an AR ordering that, in fact, might be interpretable more-or-less "as is"), and a second that is not prone to the local optimum problem plaguing all K-means iterative methods because it is based on a DP strategy guaranteeing global optimality (albeit within an order-constrained context).

To illustrate how order-constrained K-means clustering might be implemented, we go back to the wine tasting data and adopt as a constraining order the permutation [9 10 7 8 5 6 3 2 1 4] identified through `order.m` discussed in Appendix D. In the verbatim analysis below, it should be noted that an input matrix of

`wineprox = sqeuclid([9 10 7 8 5 6 3 2 1 4],[9 10 7 8 5 6 3 2 1 4])`

is used in `partitionfnd_kmeans.m`, which then induces the mandatory constraining identity permutation for the input matrix. This also implies a labeling of the columns of the `membership` matrix of [9 10 7 8 5 6 3 2 1 4] or [I J G H E F C B A D]. Considering alternatives to the earlier K-means analysis, it is interesting (and possibly substantively more meaningful) to note that the two-group solution puts the best wines ({A,B,C,D}) versus the rest ({E,F,G,H,I,J}) (and is actually the second local optima identified for $K = 2$ with an objective function loss of 633.208). The four-group solution is very interpretable and defined by the best ({A,B,C,D}), the worst ({E,G,H,J}), and the two "odd-balls" in separate classes ({F} and {I}). Its loss value of 298.300 is somewhat more than the least attainable of 252.917 (found for the less-than-pleasing subdivision, ({A,B,C,D},{E,G,H},{F,I},{J}).

```
>> load cabernet_taste.dat

>> [sqeuclid] = sqeuclidean(cabernet_taste)

sqeuclid =

         0   48.2500   48.2500   86.5000  220.0000  160.2500  400.5000  394.0000  374.5000  485.2500
   48.2500         0   77.0000   77.2500  195.2500  176.0000  327.2500  320.2500  453.2500  403.0000
   48.2500   77.0000         0  131.2500  229.2500  107.0000  326.2500  410.2500  253.2500  362.0000
   86.5000   77.2500  131.2500         0  202.5000  202.2500  355.5000  305.5000  416.0000  465.2500
  220.0000  195.2500  229.2500  202.5000         0  145.2500   75.5000  102.0000  394.5000  190.2500
  160.2500  176.0000  107.0000  202.2500  145.2500         0  160.2500  250.2500  100.2500  147.0000
  400.5000  327.2500  326.2500  355.5000   75.5000  160.2500         0   79.5000  379.5000   76.2500
  394.0000  320.2500  410.2500  305.5000  102.0000  250.2500   79.5000         0  515.5000  201.2500
  374.5000  453.2500  253.2500  416.0000  394.5000  100.2500  379.5000  515.5000         0  279.2500
  485.2500  403.0000  362.0000  465.2500  190.2500  147.0000   76.2500  201.2500  279.2500         0

>> wineprox = sqeuclid([9 10 7 8 5 6 3 2 1 4],[9 10 7 8 5 6 3 2 1 4])

wineprox =

         0  279.2500  379.5000  515.5000  394.5000  100.2500  253.2500  453.2500  374.5000  416.0000
  279.2500         0   76.2500  201.2500  190.2500  147.0000  362.0000  403.0000  485.2500  465.2500
  379.5000   76.2500         0   79.5000   75.5000  160.2500  326.2500  327.2500  400.5000  355.5000
  515.5000  201.2500   79.5000         0  102.0000  250.2500  410.2500  320.2500  394.0000  305.5000
  394.5000  190.2500   75.5000  102.0000         0  145.2500  229.2500  195.2500  220.0000  202.5000
  100.2500  147.0000  160.2500  250.2500  145.2500         0  107.0000  176.0000  160.2500  202.2500
  253.2500  362.0000  326.2500  410.2500  229.2500  107.0000         0   77.0000   48.2500  131.2500
  453.2500  403.0000  327.2500  320.2500  195.2500  176.0000   77.0000         0   48.2500   77.2500
  374.5000  485.2500  400.5000  394.0000  220.0000  160.2500   48.2500   48.2500         0   86.5000
  416.0000  465.2500  355.5000  305.5000  202.5000  202.2500  131.2500   77.2500   86.5000         0
```

```
>> [membership,objectives] = partitionfnd_kmeans(wineprox)

membership =

     1     1     1     1     1     1     1     1     1     1
     2     2     2     2     2     2     1     1     1     1
     3     2     2     2     2     2     1     1     1     1
     4     3     3     3     3     2     1     1     1     1
     5     4     3     3     3     2     1     1     1     1
     6     5     4     4     4     3     2     2     2     1
     7     6     6     5     4     3     2     2     2     1
     8     7     6     5     4     3     2     2     2     1
     9     8     7     6     5     4     3     2     2     1
    10     9     8     7     6     5     4     3     2     1


objectives =

  1.0e+003 *

    1.1110
    0.6332
    0.4026
    0.2983
    0.2028
    0.1435
    0.0960
    0.0578
    0.0241
         0
```

Chapter 6

Generalizing Ultrametrics Based on Ordered Partitions

6.1 An Alternative and Generalizable View of Ultrametric Matrix Decomposition

A general mechanism exists for decomposing any ultrametric matrix $\mathbf{U}$ into a (nonnegatively) weighted sum of dichotomous (0/1) matrices, each representing one of the partitions of the hierarchy, $\mathcal{P}_0, \ldots, \mathcal{P}_{n-2}$, induced by $\mathbf{U}$ (note that the conjoint partition is explicitly excluded in these expressions for the numerical reason we allude to below). Specifically, if $\mathbf{P}_t = \{p_{ij}^{(t)}\}$, for $0 \le t \le n-2$, is an $n \times n$ symmetric (0/1) dissimilarity matrix corresponding to $\mathcal{P}_t$ in which an entry $p_{ij}^{(t)}$ is 0 if O_i and O_j belong to the same class in $\mathcal{P}_t$ and otherwise equal to 1, then for some collection of suitably chosen nonnegative weights, $\alpha_0, \alpha_1, \ldots, \alpha_{n-2}$,

$$\mathbf{U} = \sum_{t=0}^{n-2} \alpha_t \mathbf{P}_t \ .$$

Generally, the nonnegative weights, $\alpha_0, \alpha_1, \ldots, \alpha_{n-2}$, are given by the (differences in) partition increments that calibrate the vertical axis of the dendrogram. Moreover, because the ultrametric represented by Figure 4.1 was generated by optimizing a least-squares loss function in relation to a given proximity matrix $\mathbf{P}$, an alternative interpretation for the obtained weights is that they solve the nonnegative least-squares task of

$$\min_{\{\alpha_t \geq 0,\ 0 \leq t \leq n-2\}} \sum_{i<j} (p_{ij} - \sum_{t=0}^{n-2} \alpha_t p_{ij}^{(t)})^2, \tag{6.1}$$

for the fixed collection of dichotomous matrices $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_{n-2}$. Although the solution to (6.1) is generated indirectly in this case from the least-squares optimal ultrametric directly fitted to $\mathbf{P}$, in general, for any fixed proximity matrix $\mathbf{P}$ and collection of dichotomous matrices, $\mathbf{P}_0, \ldots, \mathbf{P}_{n-2}$, however obtained, the nonnegative weights α_t, $0 \leq t \leq n-2$, solving (6.1) can be obtained with any nonnegative least-squares optimization method. We will routinely use in particular (and without further comment) the code rewritten in MATLAB for a subroutine originally provided by Wollan and Dykstra (1987) based on a strategy for solving linear inequality constrained least-squares tasks called iterative projection.

In the verbatim script below, the M-file, `partitionfit.m`, is used to reconstruct the order-constrained ultrametric for the `supreme_agree.dat` data set. The crucial component is in constructing the $m \times n$ matrix (`member`) that defines class membership for the $m = 8$ nontrivial partitions generating the ultrametric. Note in particular that the unnecessary conjoint partition involving a single class is not included (in fact, its inclusion would produce a numerical error in the least-squares subcode integral to `partitionfit.m`; thus, there would be a nonzero value for `end_condition`). The M-file `partitionfit.m` will be relied upon again when we further generalize the type of structural representations possible for a proximity matrix in the next section.

```
>> member = [1 1 1 1 2 2 2 2 2;1 1 1 1 2 2 2 3 3;1 2 2 2 3 3 3 4 4;1 2 2 2 3 4 4 5 5;
1 2 3 3 4 5 5 6 6;1 2 3 3 4 5 6 7 7;1 2 3 4 5 6 7 8 8;1 2 3 4 5 6 7
8 9]

member =

     1     1     1     1     2     2     2     2     2
     1     1     1     1     2     2     2     3     3
     1     2     2     2     3     3     3     4     4
     1     2     2     2     3     4     4     5     5
     1     2     3     3     4     5     5     6     6
     1     2     3     3     4     5     6     7     7
     1     2     3     4     5     6     7     8     8
     1     2     3     4     5     6     7     8     9

>> [fitted,vaf,weights,end_condition] = partitionfit(supreme_agree,member)

fitted =
         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369


weights =

    0.2388
    0.0383
    0.0533
    0.0250
    0.0550
    0.0100
    0.0100
    0.2100


end_condition =

     0
```

6.2 Order-Constrained Partitioning and Ultrametric Generalizations

The idea of providing an optimal mechanism for subdividing an order-constrained proximity matrix (and not one just based on squared Euclidean distances), gives a natural means for generalizing the usual (agglomerative) hierarchical clustering methods, such as complete- or average-link. Defining a good preliminary constraining order for the proximity matrix, an optimization routine (based on dynamic programming) is implemented that will give optimal partitions into 2 to $n-1$ classes respecting the preliminary order (having classes containing objects contiguous with respect to it), and minimizing the maximum such measure obtained over the classes making up the partitions (the maximum proximity [or diameter] within a class for the complete-link criterion; the average of the proximities within a class for the average-link criterion). The "minimum of the maximum" is used because otherwise a tendency will exist to produce just one large class for each optimal partition; also, this seems a closer analogue to agglomerative hierarchical clustering when we try to minimize a maximum as each partition is constructed from the proceeding one. Stated alternatively, the best single partition optimization analogue to hierarchical clustering, with the latter's myopic process and greedy "best it can do" at each next level, would be the optimization goal of minimizing the maximum subset measure over the classes of a partition.[1]

Given the broad characterization of the properties of an ultrametric described earlier, the generalization to be mentioned within this subsection rests on merely altering the type of partition allowed in the sequence, $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{n-1}$. Specifically, we will use an object order assumed without loss of generality to be the identity permutation, $O_1 \prec \cdots \prec O_n$, and a collection of partitions with fewer and fewer classes consistent with this order by requiring the classes within each partition to contain contiguous objects. When necessary, and if an input constraining order is given by, say, `inperm`, that is not the identity, we merely use the input matrix, `prox_input` = `prox(inperm,inperm)`; the identity permutation then constrains the analysis automatically, although in effect the constraint is given by `inperm` which also labels the columns of `membership`.

The M-files introduced below remove the requirement that the new classes in $\mathcal{P}_t$ are formed by uniting only existing classes in $\mathcal{P}_{t-1}$. Although class contiguity is maintained with respect to the same object order in the partitions identified, the requirement that the classes be nested is relaxed so that if a class is present in $\mathcal{P}_{t-1}$, it will no longer need to appear either as a class by itself or be properly contained within some class in $\mathcal{P}_t$. The M-files for constructing the collection of partitions respecting the given object order are called `partitionfnd_averages.m` and `partitionfnd_diameters.m`, and use dynamic programming to construct a set of partitions with from 1 to n ordered classes. The criteria minimized is the maximum over clusters of the average or of the maximum proximity within subsets, respectively. In the verbatim listing below, we note that the collection of partitions con-

[1]In the case of our K-means interpretation, a simple sum over the classes can be optimized that does not generally lead to the "one big class" triviality, apparently because of the divisions by twice the number of objects within each class in the specific loss function used in the optimization.

structed using `partitionfnd_averages.m` is actually hierarchical and produces the same order-constrained classification we have been working with all along; `partitionfnd_diameters.m` produces a "slightly non-hierarchical" set of partitions, and gives a larger vaf value of .7425 (compared to .7369 for the exact ultrametric).

```
>> load supreme_agree.dat
>> [membership,objectives] = partitionfnd_averages(supreme_agree)

membership =

     1     1     1     1     1     1     1     1     1
     2     2     2     2     1     1     1     1     1
     3     3     3     3     2     2     2     1     1
     4     3     3     3     2     2     2     1     1
     5     4     4     4     3     2     2     1     1
     6     5     4     4     3     2     2     1     1
     7     6     5     5     4     3     2     1     1
     8     7     6     5     4     3     2     1     1
     9     8     7     6     5     4     3     2     1


objectives =

    0.5044
    0.3470
    0.3133
    0.2833
    0.2633
    0.2300
    0.2200
    0.2100
         0

>> [fitted,vaf,weights,end_condition] = partitionfit(supreme_agree,membership(2:end,:))

fitted =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369


weights =

    0.2388
    0.0383
    0.0533
    0.0250
    0.0550
    0.0100
    0.0100
    0.2100


end_condition =

     0

>> [membership,objectives] = partitionfnd_diameters(supreme_agree)

membership =

     1     1     1     1     1     1     1     1     1
     2     2     2     2     1     1     1     1     1
     3     3     3     3     2     2     1     1     1
     4     3     3     3     2     2     2     1     1
     5     4     4     4     3     2     2     1     1
     6     5     4     4     3     2     2     1     1
     7     6     5     5     4     3     2     1     1
     8     7     6     5     4     3     2     1     1
     9     8     7     6     5     4     3     2     1


objectives =

    0.8600
    0.4600
    0.3800
    0.3300
    0.2900
    0.2300
    0.2200
    0.2100
         0

>> [fitted,vaf,weights,end_condition] = partitionfit(supreme_agree,membership(2:end,:))

fitted =

         0    0.3648    0.3648    0.3648    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3648         0    0.2840    0.2840    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3648    0.2840         0    0.2020    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3648    0.2840    0.2020         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.2840    0.3381    0.4190    0.4190
    0.6405    0.6405    0.6405    0.6405    0.2840         0    0.2561    0.4190    0.4190
    0.6405    0.6405    0.6405    0.6405    0.3381    0.2561         0    0.3648    0.3648
    0.6405    0.6405    0.6405    0.6405    0.4190    0.4190    0.3648         0    0.2020
    0.6405    0.6405    0.6405    0.6405    0.4190    0.4190    0.3648    0.2020         0


vaf =

    0.7425


weights =

    0.2215
    0.0541
    0.0809
         0
    0.0820
```

```
         0
         0
    0.2020


end_condition =

     0
```

Chapter 7

Extending Ultrametrics to Additive Trees

A currently popular alternative to the use of a simple ultrametric in classification, and what might be considered an extension, is that of an additive tree. Generalizing the earlier characterization of an ultrametric, an $n \times n$ matrix, $\mathbf{D} = \{d_{ij}\}$, can be called an additive tree metric (matrix) if the ultrametric inequality condition is replaced by

$d_{ij} + d_{kl} \leq \max\{d_{ik} + d_{jl}, d_{il} + d_{jk}\}$ for $1 \leq i, j, k, l \leq n$ (the additive tree metric inequality). Or equivalently (and again, much more understandable), for any object quadruple O_i, O_j, O_k, and O_l, the largest two values among the sums $d_{ij} + d_{kl}$, $d_{ik} + d_{jl}$, and $d_{il} + d_{jk}$ are equal.

Any additive tree metric matrix $\mathbf{D}$ can be represented (in many ways) as a sum of two matrices, say $\mathbf{U} = \{u_{ij}\}$ and $\mathbf{C} = \{c_{ij}\}$, where $\mathbf{U}$ is an ultrametric matrix, and $c_{ij} = g_i + g_j$ for $1 \leq i \neq j \leq n$ and $c_{ii} = 0$ for $1 \leq i \leq n$, based on some set of values $g_1, \ldots, g_n$. The multiplicity of such possible decompositions results from the choice of where to place the root in the type of graphical representation we give in Figure 7.2.

To eventually construct the type of graphical additive tree representation of Figure 7.2, the process followed is to first graph the dendrogram induced by $\mathbf{U}$, where (as for any ultrametric) the chosen root is equidistant from all terminal nodes. The branches connecting the terminal nodes are then lengthened or shortened depending on the signs and absolute magnitudes of $g_1, \ldots, g_n$. If one were willing to consider the (arbitrary) inclusion of a sufficiently large additive constant to the entries in $\mathbf{D}$, the values of $g_1, \ldots, g_n$ could be assumed nonnegative. In this case, the matrix $\mathbf{C}$ would represent what is called a centroid metric, and although a nicety, such a restriction is not absolutely necessary for the extensions we pursue.

The number of "weights" an additive tree metric requires could be equated to the maximum number of "branch lengths" that a representation such as Figure 7.2 might necessitate, i.e., n branches attached to the terminal nodes, and $n-3$ to the internal nodes only, for a total of $2n-3$. For an ultrametric, the number of such "weights" could be identified with the $n-1$ levels at which the new subsets get formed in the partition hierarchy, and would represent about half of that necessary for an additive tree. What this implies is that the VAF measures obtained for ultrametrics and additive trees are not directly comparable because a very differing number of "free weights" must be specified for each. We are reluctant to use the word "parameter" due to the absence of any explicit statistical model and because the topology (e.g., the branching pattern) of the structures that ultimately get reified by imposing numerical values for the "weights", must first be identified by some type of combinatorial optimization search process. In short, there doesn't seem to be an unambiguous way to specify, for example, the number of estimated "parameters", the number of "degrees-of-freedom" left over, or how to "adjust" the VAF value as we can do in multiple regression so it has an expected value of zero when there is "nothing going on".

One of the difficulties in working with additive trees and displaying them graphically is to find some sensible spot to site a root for the tree. Depending on where the root is placed, a differing decomposition of $\mathbf{D}$ into an ultrametric and a centroid metric is implied. The ultrametric components induced by the choice of root can differ widely with major substantive differences in the branching patterns of the hierarchical clustering. The two M-files discussed below, `cent_ultrafnd_confit.m` and `cent_ultrafnd_confnd.m`, both identify best-fitting additive trees to a given proximity matrix but where the terminal nodes of (an) ultrametric portion of the fitted matrix are then ordered according to a constraining order (`conperm`) that is either input (in `cent_ultrafnd_confit.m`), or is identified as a good one to use (in `cent_ultrafnd_confnd.m`) and then given as an output vector. In both cases, a centroid metric is first fit to the input proximity matrix; the residual matrix is carried over to the order-constrained ultrametric constructions routines (`ultrafnd_confit.m` or `ultrafnd_confnd.m`), and thus, the root is chosen naturally for the ultrametric component. The whole process then iterates with a new centroid metric estimation, an order-constrained ultrametric re-estimation, and so on until convergence is achieved for the VAF values.

We illustrate below what occurs for our `supreme_agree.dat` data and the imposition of the identity permutation (1:9) for the terminal nodes of the ultrametric. The relevant outputs are the ultrametric component in `targtwo` and the lengths for the centroid metric in `lengthsone`. To graph the additive tree, we first add .60 to the entries in `targtwo` to make them all positive and graph this ultrametric as in Figure 7.1. Then, (1/2)(.60) = .30 is subtracted from each term in `lengthsone`; the branches attached to the terminal nodes of the ultrametric are then stretched or shrunk accordingly to produce Figure 7.2. (These stretching/shrinking factors are as follows: St: (.07); Br: (−.05); Gi: (−.06); So: (−.09); Oc: (−.18); Ke: (−.14); Re: (−.10); Sc: (.06); Th: (.06).) We note that if `cent_ultrafnd_confnd.m` were invoked to find a good constraining order for the ultrametric component, the VAF could be increased slightly (to 98.56% from 98.41% for Figure 7.2) using the `conperm` of [3 1 4 2 5 6 7 9 8]. No real substantive interpretative difference, however, is apparent from the structure given for a constraining identity permutation.

```
>> [find,vaf,outperm,targone,targtwo,lengthsone] = cent_ultrafnd_confit(supreme_agree,randperm(9),1:9)

find =

         0    0.3800    0.3707    0.3793    0.6307    0.6643    0.7067    0.8634    0.8649
    0.3800         0    0.2493    0.2579    0.5093    0.5429    0.5852    0.7420    0.7434
    0.3707    0.2493         0    0.2428    0.4941    0.5278    0.5701    0.7269    0.7283
    0.3793    0.2579    0.2428         0    0.4667    0.5003    0.5427    0.6994    0.7009
    0.6307    0.5093    0.4941    0.4667         0    0.2745    0.3168    0.4736    0.4750
    0.6643    0.5429    0.5278    0.5003    0.2745         0    0.2483    0.4051    0.4065
    0.7067    0.5852    0.5701    0.5427    0.3168    0.2483         0    0.3293    0.3307
    0.8634    0.7420    0.7269    0.6994    0.4736    0.4051    0.3293         0    0.2100
    0.8649    0.7434    0.7283    0.7009    0.4750    0.4065    0.3307    0.2100         0


vaf =

    0.9841


outperm =

     1     2     3     4     5     6     7     8     9
```

```
targone =

         0    0.6246    0.6094    0.5820    0.4977    0.5313    0.5737    0.7304    0.7319
    0.6246         0    0.4880    0.4606    0.3763    0.4099    0.4522    0.6090    0.6104
    0.6094    0.4880         0    0.4454    0.3611    0.3948    0.4371    0.5939    0.5953
    0.5820    0.4606    0.4454         0    0.3337    0.3673    0.4097    0.5664    0.5679
    0.4977    0.3763    0.3611    0.3337         0    0.2830    0.3253    0.4821    0.4836
    0.5313    0.4099    0.3948    0.3673    0.2830         0    0.3590    0.5158    0.5172
    0.5737    0.4522    0.4371    0.4097    0.3253    0.3590         0    0.5581    0.5595
    0.7304    0.6090    0.5939    0.5664    0.4821    0.5158    0.5581         0    0.7163
    0.7319    0.6104    0.5953    0.5679    0.4836    0.5172    0.5595    0.7163         0


targtwo =

         0   -0.2446   -0.2387   -0.2027    0.1330    0.1330    0.1330    0.1330    0.1330
   -0.2446         0   -0.2387   -0.2027    0.1330    0.1330    0.1330    0.1330    0.1330
   -0.2387   -0.2387         0   -0.2027    0.1330    0.1330    0.1330    0.1330    0.1330
   -0.2027   -0.2027   -0.2027         0    0.1330    0.1330    0.1330    0.1330    0.1330
    0.1330    0.1330    0.1330    0.1330         0   -0.0085   -0.0085   -0.0085   -0.0085
    0.1330    0.1330    0.1330    0.1330   -0.0085         0   -0.1107   -0.1107   -0.1107
    0.1330    0.1330    0.1330    0.1330   -0.0085   -0.1107         0   -0.2288   -0.2288
    0.1330    0.1330    0.1330    0.1330   -0.0085   -0.1107   -0.2288         0   -0.5063
    0.1330    0.1330    0.1330    0.1330   -0.0085   -0.1107   -0.2288   -0.5063         0


lengthsone =

    0.3730    0.2516    0.2364    0.2090    0.1247    0.1583    0.2007    0.3574    0.3589


>> [find,vaf,outperm,targone,targtwo,lengthsone] = cent_ultrafnd_confnd(supreme_agree,randperm(9))

find =

         0    0.3400    0.2271    0.2794    0.4974    0.5310    0.5734    0.7316    0.7301
    0.3400         0    0.3629    0.4151    0.6331    0.6667    0.7091    0.8673    0.8659
    0.2271    0.3629         0    0.2556    0.4736    0.5072    0.5495    0.7078    0.7063
    0.2794    0.4151    0.2556         0    0.4967    0.5303    0.5727    0.7309    0.7294
    0.4974    0.6331    0.4736    0.4967         0    0.2745    0.3168    0.4750    0.4736
    0.5310    0.6667    0.5072    0.5303    0.2745         0    0.2483    0.4065    0.4051
    0.5734    0.7091    0.5495    0.5727    0.3168    0.2483         0    0.3307    0.3293
    0.7316    0.8673    0.7078    0.7309    0.4750    0.4065    0.3307         0    0.2100
    0.7301    0.8659    0.7063    0.7294    0.4736    0.4051    0.3293    0.2100         0


vaf =

    0.9856


outperm =

     3     1     4     2     5     6     7     9     8


targone =

         0    0.6151    0.4556    0.4787    0.3644    0.3980    0.4404    0.5986    0.5971
    0.6151         0    0.5913    0.6144    0.5001    0.5337    0.5761    0.7343    0.7329
    0.4556    0.5913         0    0.4549    0.3406    0.3742    0.4165    0.5748    0.5733
    0.4787    0.6144    0.4549         0    0.3637    0.3973    0.4397    0.5979    0.5964
    0.3644    0.5001    0.3406    0.3637         0    0.2830    0.3253    0.4836    0.4821
    0.3980    0.5337    0.3742    0.3973    0.2830         0    0.3590    0.5172    0.5158
    0.4404    0.5761    0.4165    0.4397    0.3253    0.3590         0    0.5595    0.5581
    0.5986    0.7343    0.5748    0.5979    0.4836    0.5172    0.5595         0    0.7163
    0.5971    0.7329    0.5733    0.5964    0.4821    0.5158    0.5581    0.7163         0


targtwo =

         0   -0.2751   -0.2284   -0.1993    0.1330    0.1330    0.1330    0.1330    0.1330
   -0.2751         0   -0.2284   -0.1993    0.1330    0.1330    0.1330    0.1330    0.1330
   -0.2284   -0.2284         0   -0.1993    0.1330    0.1330    0.1330    0.1330    0.1330
   -0.1993   -0.1993   -0.1993         0    0.1330    0.1330    0.1330    0.1330    0.1330
    0.1330    0.1330    0.1330    0.1330         0   -0.0085   -0.0085   -0.0085   -0.0085
    0.1330    0.1330    0.1330    0.1330   -0.0085         0   -0.1107   -0.1107   -0.1107
    0.1330    0.1330    0.1330    0.1330   -0.0085   -0.1107         0   -0.2288   -0.2288
    0.1330    0.1330    0.1330    0.1330   -0.0085   -0.1107   -0.2288         0   -0.5063
    0.1330    0.1330    0.1330    0.1330   -0.0085   -0.1107   -0.2288   -0.5063         0


lengthsone =

    0.2397    0.3754    0.2159    0.2390    0.1247    0.1583    0.2007    0.3589    0.3574
```

In addition to the additive tree identification routines just described, there are two more illustrated below, called `atreefit.m` and `atreefnd.m`; these two M-files are direct analogues of `ultrafit.m` and `ultrafnd.m` introduced in an earlier chapter. Again, the complete-link target, `sc_completelink_target.dat`, can be used in `atreefit.m` (producing a structure with VAF of 94.89%); `atreefnd.m` generates the same additive tree as `cent_ultrafnd_confnd.m` with a VAF of 98.56%. The M-file, `atreedec.m`, provides a mechanism for decomposing any given additive tree matrix into an ultrametric and a centroid metric matrix (where the root is situated halfway along the longest path). The form of the usage is

```
[ulmetric,ctmetric] = atreedec(prox,constant)
```

where `prox` is the input (additive tree) proximity matrix (still with a zero main diagonal and a dissimilarity interpretation); `constant` is a nonnegative number (less than or equal to the maximum proximity value) that controls the positivity of the constructed ultrametric values; `ulmetric` is the ultrametric component of the decomposition; `ctmetric` is the centroid metric component (given by values, $g_1, \ldots, g_n$, assigned to each of the objects, some of which may be negative depending on the input proximity matrix and constant used).

There are two additional utility files, `ultraorder.m` and `ultraplot.m`,

Figure 7.1: A Dendrogram (Tree) Representation for the Ordered-Constrained Ultrametric Component of the Additive Tree Represented in Figure 7.2

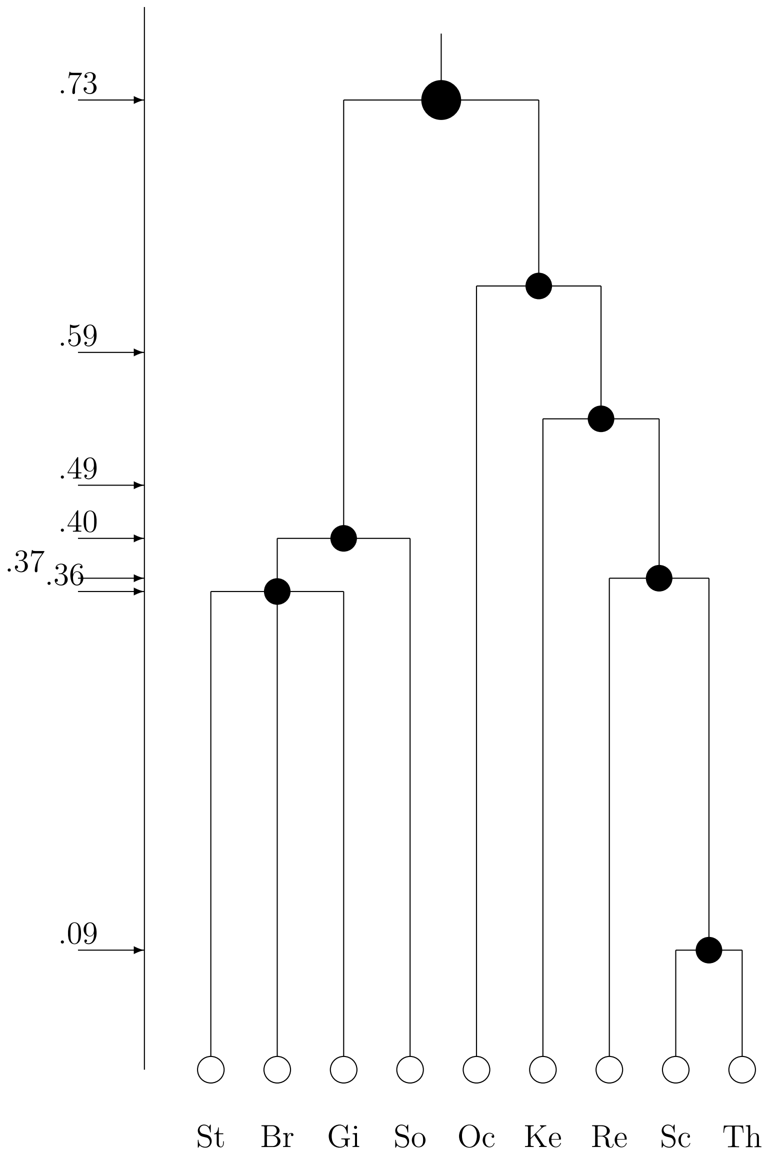
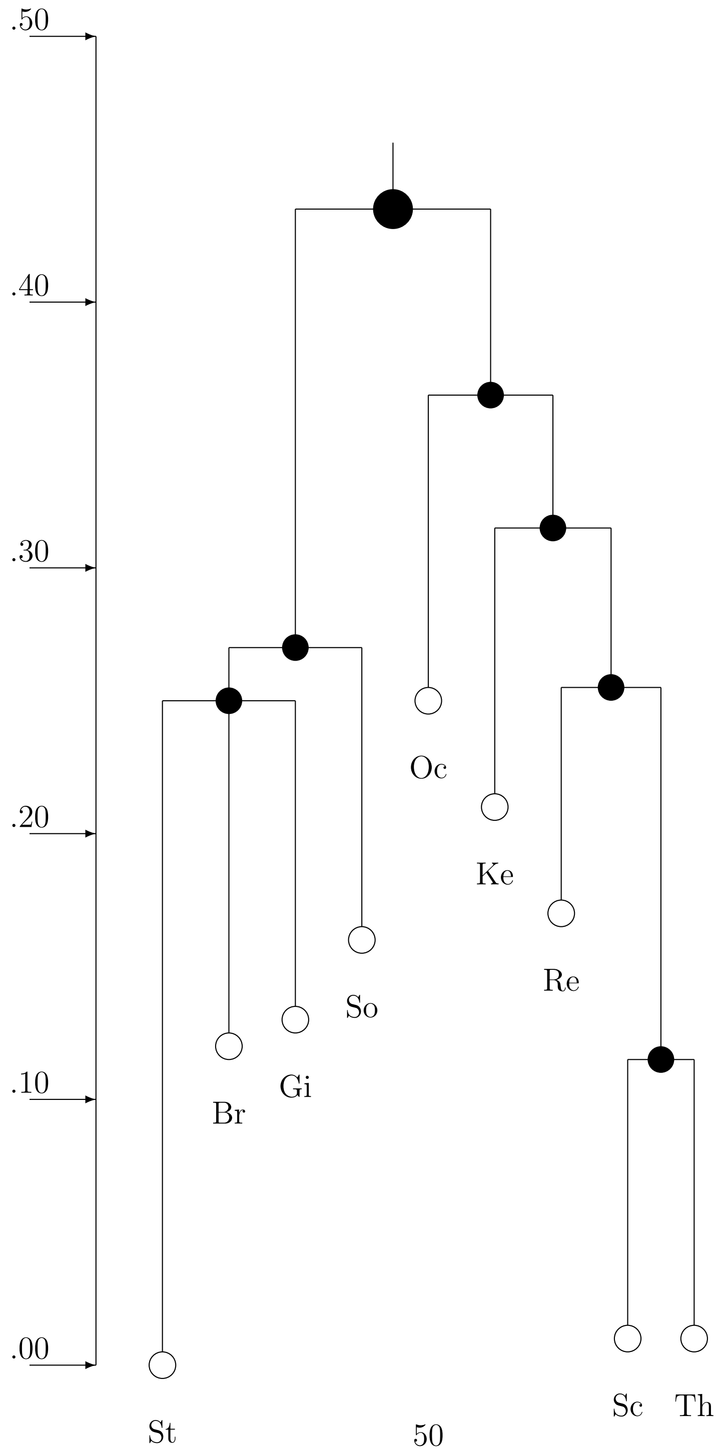


Figure 7.2: An Graph-Theoretic Representation for the Ordered-Constrained Additive Tree Described in the Text (Having VAF of 98.41%)



that may prove useful in explaining the ultrametric components identified from an application of `atreedec.m`. In the explicit usage

```
[orderprox,orderperm] = ultraorder(prox)
```

the matrix `prox` is assumed to be ultrametric; `orderperm` is a permutation used to display an AR form in `orderprox`, where

```
orderprox = prox(orderperm,orderperm)
```

The second utility's usage of `ultraplot(ultra)`, where `ultra` is a matrix presumed to satisfy the ultrametric inequality, first adds a constant to all values in `ultra` to make the entries positive; the MATLAB Statistics Toolbox routine, `dendrogram.m`, is then invoked to plot the resulting ultrametric matrix.

```
>> load supreme_agree.dat

>> load sc_completelink_target.dat

>> [fit,vaf] = atreefit(supreme_agree,sc_completelink_target)

fit =

         0    0.3972    0.3850    0.3678    0.6748    0.6670    0.6756    0.8456    0.8470
    0.3972         0    0.2635    0.2464    0.5533    0.5456    0.5542    0.7242    0.7256
    0.3850    0.2635         0    0.2200    0.5411    0.5333    0.5419    0.7119    0.7133
    0.3678    0.2464    0.2200         0    0.5239    0.5162    0.5247    0.6947    0.6962
    0.6748    0.5533    0.5411    0.5239         0    0.2449    0.2535    0.4235    0.4249
    0.6670    0.5456    0.5333    0.5162    0.2449         0    0.2300    0.4158    0.4172
    0.6756    0.5542    0.5419    0.5247    0.2535    0.2300         0    0.4243    0.4258
    0.8456    0.7242    0.7119    0.6947    0.4235    0.4158    0.4243         0    0.2100
    0.8470    0.7256    0.7133    0.6962    0.4249    0.4172    0.4258    0.2100         0


vaf =

    0.9489

>> [find,vaf] = atreefnd(supreme_agree,randperm(9))

find =

         0    0.4151    0.3400    0.3629    0.6329    0.6668    0.7091    0.8659    0.8673
    0.4151         0    0.2794    0.2556    0.4965    0.5304    0.5727    0.7295    0.7309
    0.3400    0.2794         0    0.2271    0.4972    0.5311    0.5734    0.7302    0.7316
    0.3629    0.2556    0.2271         0    0.4734    0.5073    0.5496    0.7064    0.7078
    0.6329    0.4965    0.4972    0.4734         0    0.2745    0.3168    0.4736    0.4750
    0.6668    0.5304    0.5311    0.5073    0.2745         0    0.2483    0.4051    0.4065
    0.7091    0.5727    0.5734    0.5496    0.3168    0.2483         0    0.3293    0.3307
    0.8659    0.7295    0.7302    0.7064    0.4736    0.4051    0.3293         0    0.2100
    0.8673    0.7309    0.7316    0.7078    0.4750    0.4065    0.3307    0.2100         0


vaf =
    0.9856

>> [ulmetric,ctmetric] = atreedec(find,1.0)

ulmetric =

         0    0.8168    0.7410    0.7877    1.1327    1.1327    1.1327    1.1327    1.1327
    0.8168         0    0.8168    0.8168    1.1327    1.1327    1.1327    1.1327    1.1327
    0.7410    0.8168         0    0.7877    1.1327    1.1327    1.1327    1.1327    1.1327
    0.7877    0.8168    0.7877         0    1.1327    1.1327    1.1327    1.1327    1.1327
    1.1327    1.1327    1.1327    1.1327         0    0.9748    0.9748    0.9748    0.9748
    1.1327    1.1327    1.1327    1.1327    0.9748         0    0.8724    0.8724    0.8724
    1.1327    1.1327    1.1327    1.1327    0.9748    0.8724         0    0.7543    0.7543
    1.1327    1.1327    1.1327    1.1327    0.9748    0.8724    0.7543         0    0.4768
    1.1327    1.1327    1.1327    1.1327    0.9748    0.8724    0.7543    0.4768         0


ctmetric =

   -0.1327
   -0.2691
   -0.2684
   -0.2922
   -0.3671
   -0.3332
   -0.2909
   -0.1341
   -0.1327

>> [orderprox,orderperm] = ultraorder(ulmetric)

orderprox =

         0    0.7877    0.7877    0.8168    1.1327    1.1327    1.1327    1.1327    1.1327
    0.7877         0    0.7410    0.8168    1.1327    1.1327    1.1327    1.1327    1.1327
    0.7877    0.7410         0    0.8168    1.1327    1.1327    1.1327    1.1327    1.1327
    0.8168    0.8168    0.8168         0    1.1327    1.1327    1.1327    1.1327    1.1327
    1.1327    1.1327    1.1327    1.1327         0    0.7543    0.7543    0.8724    0.9748
    1.1327    1.1327    1.1327    1.1327    0.7543         0    0.4768    0.8724    0.9748
    1.1327    1.1327    1.1327    1.1327    0.7543    0.4768         0    0.8724    0.9748
    1.1327    1.1327    1.1327    1.1327    0.8724    0.8724    0.8724         0    0.9748
    1.1327    1.1327    1.1327    1.1327    0.9748    0.9748    0.9748    0.9748         0


orderperm =

     4     3     1     2     7     8     9     6     5
```

7.1 Visualization of Ultrametrics and Additive Trees Using the Bioinformatics Toolbox

An alternative mechanism for visualizing ultrametrics and additive trees is through the plot function of the Bioinformatics Toolbox used on phylogenetic classes constructed with `seqlinkage.m` (on our fitted ultrametric matrices) or with `seqneighjoin.m` (on our fitted additive tree matrices). The script below produces the representations of Figures 7.3 and 7.4 for our best ultrametric and best additive tree found for the `supreme_agree.dat`.

```
 load supreme_agree.dat
>> [find,vaf] = ultrafnd(supreme_agree,randperm(9))

find =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369

>> find_vector = squareform(find)

find_vector =

  Columns 1 through 12

    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405    0.2850    0.2850    0.6405    0.6405

  Columns 13 through 24

    0.6405    0.6405    0.6405    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405    0.6405    0.6405    0.6405

  Columns 25 through 36

    0.6405    0.6405    0.3100    0.3100    0.4017    0.4017    0.2300    0.4017    0.4017    0.4017    0.4017    0.2100

>> names{1} = 'St';
>> names{2} = 'Br';
>> names{3} = 'Gi';
>> names{4} = 'So';
>> names{5} = 'Oc';
>> names{6} = 'Ke';
>> names{7} = 'Re';
>> names{8} = 'Sc';
>> names{9} = 'Th';
>> names

names =

    'St'    'Br'    'Gi'    'So'    'Oc'    'Ke'    'Re'    'Sc'    'Th'

>> phylotree_ultrametric = seqlinkage(find_vector,'complete',names)
    Phylogenetic tree object with 9 leaves (8 branches)
>> plot(phylotree_ultrametric,'Type','angular','Orientation','top')
>> [find,vaf] = atreefnd(supreme_agree,randperm(9))

find =

         0    0.4151    0.3400    0.3629    0.6329    0.6668    0.7091    0.8659    0.8673
    0.4151         0    0.2794    0.2556    0.4965    0.5304    0.5727    0.7295    0.7309
    0.3400    0.2794         0    0.2271    0.4972    0.5311    0.5734    0.7302    0.7316
    0.3629    0.2556    0.2271         0    0.4734    0.5073    0.5496    0.7064    0.7078
    0.6329    0.4965    0.4972    0.4734         0    0.2745    0.3168    0.4736    0.4750
    0.6668    0.5304    0.5311    0.5073    0.2745         0    0.2483    0.4051    0.4065
    0.7091    0.5727    0.5734    0.5496    0.3168    0.2483         0    0.3293    0.3307
    0.8659    0.7295    0.7302    0.7064    0.4736    0.4051    0.3293         0    0.2100
    0.8673    0.7309    0.7316    0.7078    0.4750    0.4065    0.3307    0.2100         0


vaf =

    0.9856

>> find_vector = squareform(find)

find_vector =

  Columns 1 through 12

    0.4151    0.3400    0.3629    0.6329    0.6668    0.7091    0.8659    0.8673    0.2794    0.2556    0.4965    0.5304

  Columns 13 through 24

    0.5727    0.7295    0.7309    0.2271    0.4972    0.5311    0.5734    0.7302    0.7316    0.4734    0.5073    0.5496

  Columns 25 through 36

    0.7064    0.7078    0.2745    0.3168    0.4736    0.4750    0.2483    0.4051    0.4065    0.3293    0.3307    0.2100


>> phylotree_additivetree = seqneighjoin(find_vector,'equivar',names)
    Phylogenetic tree object with 9 leaves (8 branches)
>> plot(phylotree_additivetree,'Type','angular','Orientation','top')
```

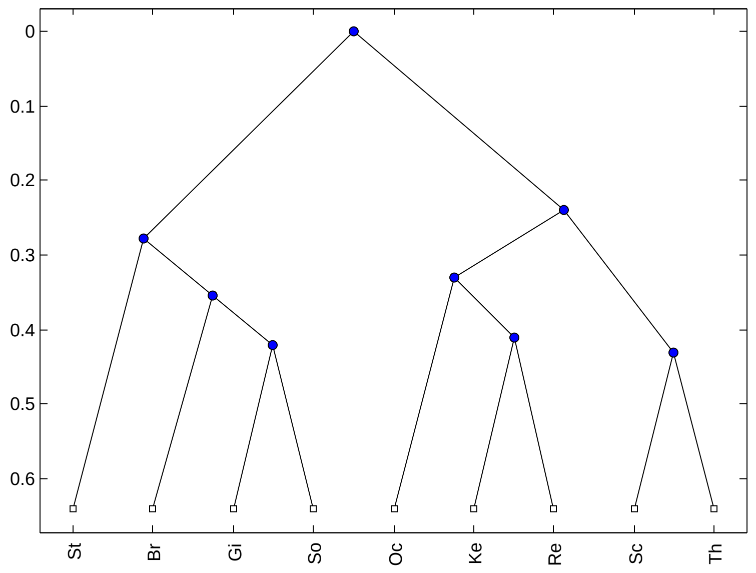


Figure 7.3: Tree Representation Using the Bioinformatics Toolbox for the Best-fitting Ultrametric for the Supreme Court Proximity Data (VAF of 73.69%)

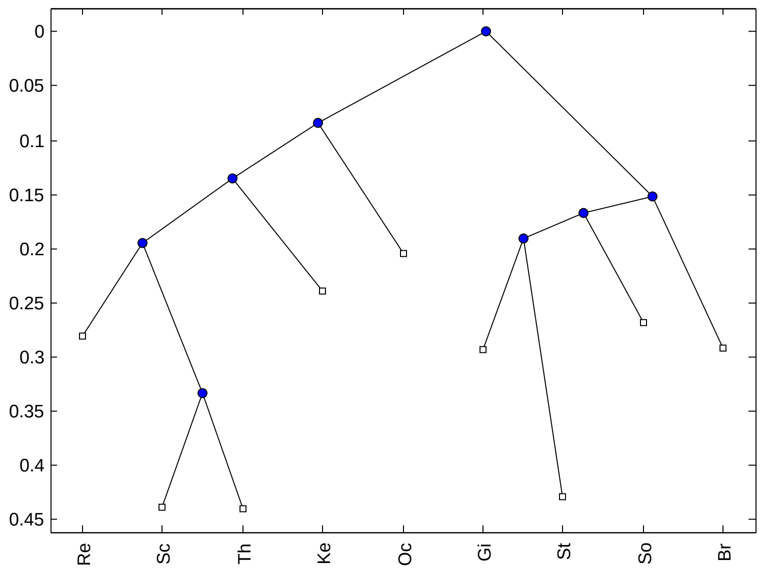


Figure 7.4: Tree Representation Using the Bioinformatics Toolbox for the Best-fitting Additive Tree for the Supreme Court Proximity Data (VAF of 98.56%)

Appendix A

Ultrametrics and Additive Trees for Two-Mode (Rectangular) Proximity Data

The proximity data considered thus far for obtaining some type of structure, such as an ultrametric or an additive tree, have been assumed to be on one intact set of objects, $S = \{O_1, \ldots, O_n\}$, and complete in the sense that proximity values are present between all object pairs. Suppose now that the available proximity data are two-mode, and between two distinct object sets, $S_A = \{O_{1A}, \ldots, O_{n_aA}\}$ and $S_B = \{O_{1B}, \ldots, O_{n_bB}\}$, containing n_a and n_b objects, respectively, given by an $n_a \times n_b$ proximity matrix $\mathbf{Q} = \{q_{rs}\}$. Again, we assume that the entries in $\mathbf{Q}$ are keyed as dissimilarities, and a joint structural representation is desired for the combined set $S_A \cup S_B$. We might caution at the outset of the need to have legitimate proximities to make the analyses to follow very worthwhile or interpretable. There are many numerical elicitation schemes where subjects (e.g., raters) are asked to respond to some set of objects (e.g., items). If the elicitation is for, say, preference, then proximity may be a good interpretation for the numerical values. If, on the other hand, the numerical value is merely a rating given on some more-or-less objective criterion where only errors of observation induce the variability from rater to rater, then probably not.

Conditions have been proposed in the literature for when the entries in a matrix fitted to $\mathbf{Q}$ characterize an ultrametric or an additive tree representation. In particular, suppose an $n_a \times n_b$ matrix, $\mathbf{F} = \{f_{rs}\}$, is fitted to $\mathbf{Q}$

through least squares subject to the constraints that follow:

Ultrametric (Furnas, 1980):

for all distinct object quadruples, O_{rA}, O_{sA}, O_{rB}, O_{sB}, where O_{rA}, $O_{sA} \in S_A$ and O_{rB}, O_{sB}, $\in S_B$, and considering the entries in $\mathbf{F}$ corresponding to the pairs, (O_{rA}, O_{rB}), (O_{rA}, O_{sB}), (O_{sA} O_{rB}), and (O_{sA}, O_{sB}), say $f_{r_A r_B}$, $f_{r_A s_B}$, $f_{s_A r_B}$, $f_{s_A s_B}$, respectively, the largest two must be equal.

Additive trees (Brossier, 1987):

for all distinct object sextuples, O_{rA}, O_{sA}, O_{tA}, O_{rB}, O_{sB}, O_{tB}, where O_{rA}, O_{sA}, $O_{tA} \in S_A$ and O_{rB}, O_{sB}, O_{tB}, $\in S_B$, and considering the entries in $\mathbf{F}$ corresponding to the pairs (O_{rA}, O_{rB}), (O_{rA}, O_{sB}), (O_{rA}, O_{tB}), (O_{sA}, O_{rB}), (O_{sA}, O_{sB}), (O_{sA}, O_{tB}), (O_{tA}, O_{rB}), (O_{tA}, O_{sB}), and (O_{tA}, O_{tB}), say $f_{r_A r_B}$, $f_{r_A s_B}$, $f_{r_A t_B}$, $f_{s_A r_B}$, $f_{s_A s_B}$, $f_{s_A t_B}$, $f_{t_A r_B}$, $f_{t_A s_B}$, $f_{t_A t_B}$, respectively, the largest two of the following sums must be equal:

$f_{r_A r_B} + f_{s_A s_B} + f_{t_A t_B}$;
$f_{r_A r_B} + f_{s_A t_B} + f_{t_A s_B}$;
$f_{r_A s_B} + f_{s_A r_B} + f_{t_A t_B}$;
$f_{r_A s_B} + f_{s_A t_B} + f_{t_A r_B}$;
$f_{r_A t_B} + f_{s_A r_B} + f_{t_A s_B}$;
$f_{r_A t_B} + f_{s_A s_B} + f_{t_A r_B}$.

A.1 Two-Mode Ultrametrics

To illustrate the fitting of a given two-mode ultrametric, a two-mode target is generated by extracting a 5×4 portion from the 9×9 ultrametric target matrix, `sc_completelink_target.dat`, used earlier. This file has contents as follows (`sc_completelink_target5x4.dat`):

```
0.3800     0.3800     0.8600     0.8600
0.2900     0.2200     0.8600     0.8600
0.8600     0.8600     0.3300     0.4600
0.8600     0.8600     0.2300     0.4600
0.8600     0.8600     0.4600     0.2100
```

The five rows correspond to the judges, St, Gi, Oc, Re, Th; the four columns to Br, So, Ke, Sc. As the two-mode 5×4 proximity matrix, the appropriate portion of the `supreme_agree.dat` proximity matrix will be used in the fitting process; the corresponding file is called `supreme_agree5x4.dat`, with contents:

```
    0.3000    0.3700    0.6400    0.8600
    0.2800    0.2200    0.5100    0.7200
    0.4500    0.4500    0.3300    0.4600
    0.5700    0.5600    0.2300    0.3400
    0.7600    0.7100    0.4100    0.2100
```

Because of the way the joint set of row and columns objects is numbered, the five rows are labeled from 1 to 5 and the four columns from 6 to 9. Thus, the correspondence between the justices and the numbers obviously differs from earlier applications:

1:St; 2:Gi; 3:Oc; 4:Re; 5:Th; 6:Br; 7:So; 8:Ke; 9:Sc

The M-file, `ultrafittm.m`, fits a given ultrametric to a two-mode proximity matrix, and has usage

```
[fit,vaf] = ultrafittm(proxtm,targ)
```

where `proxtm` is the two-mode (rectangular) input proximity matrix (with a dissimilarity interpretation); `targ` is an ultrametric matrix of the same size as `proxtm`; `fit` is the least-squares optimal matrix (with variance-accounted-for of `vaf`) to `proxtm` satisfying the two-mode ultrametric constraints implicit in `targ`. An example follows using `sc_completelink_target5x4.dat` for `targ` and `supreme_agree5x4.dat` as `proxtm`:

```
>> load supreme_agree5x4.dat

>> load sc_completelink_target5x4.dat

>> supreme_agree5x4

supreme_agree5x4 =

    0.3000    0.3700    0.6400    0.8600
    0.2800    0.2200    0.5100    0.7200
    0.4500    0.4500    0.3300    0.4600
    0.5700    0.5600    0.2300    0.3400
    0.7600    0.7100    0.4100    0.2100

>> sc_completelink_target5x4

sc_completelink_target5x4 =

    0.3800    0.3800    0.8600    0.8600
    0.2900    0.2200    0.8600    0.8600
    0.8600    0.8600    0.3300    0.4600
    0.8600    0.8600    0.2300    0.4600
    0.8600    0.8600    0.4600    0.2100
```

```
>> [fit,vaf] = ultrafittm(supreme_agree5x4,sc_completelink_target5x4)

fit =

    0.3350    0.3350    0.6230    0.6230
    0.2800    0.2200    0.6230    0.6230
    0.6230    0.6230    0.3300    0.4033
    0.6230    0.6230    0.2300    0.4033
    0.6230    0.6230    0.4033    0.2100


vaf =

    0.7441
```

A VAF of 74.41% was obtained for the fitted ultrametric; the easily interpretable hierarchy is given below with indications of when the partitions were formed using both the number and letter schemes to label the justices:

Partition	Level
{{4:Re,8:Ke,3:Oc,5:Th,9:Sc,1:St,2:Gi,7:So,6:Br}	.6230
{{4:Re,8:Ke,3:Oc,5:Th,9:Sc},{1:St,2:Gi,7:So,6:Br}}	.4033
{{4:Re,8:Ke,3:Oc},{5:Th,9:Sc},{1:St,2:Gi,7:So,6:Br}	.3350
{{4:Re,8:Ke,3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So,6:Br}}	.3300
{{4:Re,8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So,6:Br}}	.2800
{{4:Re,8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So},{6:Br}}	.2300
{{4:Re},{8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So},{6:Br}}	.2200
{{4:Re},{8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi},{7:So},{6:Br}}	.2100
{{4:Re},{8:Ke},{3:Oc},{5:Th},{9:Sc},{1:St},{2:Gi},{7:So},{6:Br}}	—

The M-file, `ultrafndtm.m` locates a best-fitting two-mode ultrametric with usage

```
[find,vaf] = ultrafndtm(proxtm,inpermrow,inpermcol)
```

where `proxtm` is the two-mode input proximity matrix (with a dissimilarity interpretation); `inpermrow` and `inpermcol` are permutations for the row and column objects that determine the order in which the inequality constraints are considered; `find` is the found least-squares matrix (with variance-accounted-for of `vaf`) to `proxtm` satisfying the ultrametric constraints. The example below for `supreme_agree5x4.dat` (using random permutations for both `inpermrow` and `inpermcol`), finds exactly the same ultrametric as above with `vaf` of .7441.

```
>> [find,vaf] = ultrafndtm(supreme_agree5x4,randperm(5),randperm(4))
```

```
find =

    0.3350    0.3350    0.6230    0.6230
    0.2800    0.2200    0.6230    0.6230
    0.6230    0.6230    0.3300    0.4033
    0.6230    0.6230    0.2300    0.4033
    0.6230    0.6230    0.4033    0.2100


vaf =

    0.7441
```

A.2 Two-Mode Additive Trees

The identification of a best-fitting two-mode additive tree will be done somewhat differently than for a two-mode ultrametric representation, largely because of future storage considerations when huge matrices might be considered. Specifically, a (two-mode) centroid metric and a (two-mode) ultrametric matrix will be identified so that their sum is a good-fitting two-mode additive tree. Because a centroid metric can be obtained in closed-form, we first illustrate the fitting of just a centroid metric to a two-mode proximity matrix with the M-file, `centfittm.m`. Its usage is of the form

```
[fit,vaf,lengths] = centfittm(proxtm)
```

giving the least-squares fitted two-mode centroid metric (`fit`) to `proxtm`, the two-mode rectangular input proximity matrix (with a dissimilarity interpretation). The n values (where $n =$ number of rows(n_a) + number of columns(n_b)) serve to define the approximating sums, $u_r + v_s$, where the u_r are for the n_a rows and the v_s for the n_b columns; these u_r and v_s values are given in the vector `lengths` of size $n \times 1$, with row values first followed by the column values. The closed-form formula used for u_r (or v_s) can be given simply as the r^{th} row (or s^{th} column) mean of `proxtm` minus one-half the grand mean. In the example below using the two-mode matrix, `supreme_agree5x4.dat`, a two-mode centroid metric by itself has a (paltry) `vaf` of .1090.

```
>> [fit,vaf,lengths] = centfittm(supreme_agree5x4)

fit =

    0.5455    0.5355    0.4975    0.5915
    0.4355    0.4255    0.3875    0.4815
    0.4255    0.4155    0.3775    0.4715
    0.4280    0.4180    0.3800    0.4740
    0.5255    0.5155    0.4775    0.5715


vaf =

    0.1090


lengths =

    0.3080
    0.1980
    0.1880
    0.1905
    0.2880
    0.2375
    0.2275
    0.1895
    0.2835
```

The identification of a two-mode additive tree with the M-file, `atreefndtm.m`, proceeds iteratively. A two-mode centroid metric is first found and the original two-mode proximity matrix residualized; a two-mode ultrametric is then identified for the residual matrix. The process repeats with the centroid and ultrametric components alternatingly being refit until a small change in the overall VAF occurs. The M-file has the explicit usage

```
[find,vaf,ultra,lengths] = atreefndtm(proxtm,inpermrow,inpermcol)
```

Here, `proxtm` is the rectangular input proximity matrix (with a dissimilarity interpretation); `inpermrow` and `inpermcol` are permutations for the row and column objects that determine the order in which the inequality constraints are considered; `find` is the found least-squares matrix (with variance-accounted-for of `vaf`) to `proxtm` satisfying the two-mode additive tree constraints. The vector `lengths` contains the row followed by column values for the two-mode centroid metric component; `ultra` is the ultrametric component. In the example given below, the identified two-mode additive-tree for `supreme_agree5x4.dat` has `vaf` of .9953. The actual partition hierarchy is given in the next section along with an indication of when the various partitions are formed using a utility M-file that completes a two-mode ultrametric so it is defined over the entire joint set.

```
>> [find,vaf,ultra,lengths] = atreefndtm(supreme_agree5x4,randperm(5),randperm(4))

find =
    0.3000    0.3768    0.6533    0.8399
    0.2732    0.2200    0.5251    0.7117
    0.4625    0.4379    0.3017    0.4883
    0.5755    0.5510    0.2333    0.3400
    0.7488    0.7243    0.4067    0.2100


vaf =

    0.9953


ultra =

   -0.2658   -0.1644    0.1576    0.1576
   -0.1644   -0.1931    0.1576    0.1576
    0.1576    0.1576    0.0668    0.0668
    0.1576    0.1576   -0.1145   -0.1945
    0.1576    0.1576   -0.1145   -0.4978


lengths =

    0.3368
    0.2086
    0.0759
    0.1889
    0.3623
    0.2290
    0.2045
    0.1589
    0.3456
```

A.3 Completing a Two-Mode Ultrametric to One Defined on the Combined Object Set

Instead of relying only on our general intuition (and problem-solving skills) to transform a fitted two-mode ultrametric to one we could interpret directly as a sequence of partitions for the joint set $S_A \cup S_B$, the M-file, `ultracomptm.m`, provides the explicit completion of a given two-mode ultrametric matrix to a symmetric proximity matrix (defined on $S_A \cup S_B$ and satisfying the usual ultrametric constraints). Thus, this completion, in effect, estimates the (missing) ultrametric values that must be present between objects from the same cluster and from the same mode. The general syntax has the form

```
[ultracomp] = ultracomptm(ultraproxtm)
```

where `ultraproxtm` is the $n_a \times n_b$ fitted two-mode ultrametric matrix; `ultracomp` is the completed $n \times n$ proximity matrix having the usual ultrametric pattern for the complete object set of size $n = n_a + n_b$. As seen in the example below, the use of `ultrafndtm.m` on `supreme_agree5x4.dat`, and the subsequent application of `ultracomptm.m` (plus `ultraorder.m` on `ultracomp`), leads directly to the partition hierarchy given following the verbatim output.

```
>> [ultracomp] = ultracomptm(ultra)

ultracomp =

        0   -0.1644    0.1576    0.1576    0.1576   -0.2658   -0.1644    0.1576    0.1576
  -0.1644         0    0.1576    0.1576    0.1576   -0.1644   -0.1931    0.1576    0.1576
   0.1576    0.1576         0    0.0668    0.0668    0.1576    0.1576    0.0668    0.0668
   0.1576    0.1576    0.0668         0   -0.1945    0.1576    0.1576   -0.1145   -0.1945
   0.1576    0.1576    0.0668   -0.1945         0    0.1576    0.1576   -0.1145   -0.4978
  -0.2658   -0.1644    0.1576    0.1576    0.1576         0   -0.1644    0.1576    0.1576
  -0.1644   -0.1931    0.1576    0.1576    0.1576   -0.1644         0    0.1576    0.1576
   0.1576    0.1576    0.0668   -0.1145   -0.1145    0.1576    0.1576         0   -0.1145
   0.1576    0.1576    0.0668   -0.1945   -0.4978    0.1576    0.1576   -0.1145         0

>> [orderprox,orderperm] = ultraorder(ultracomp)

orderprox =

        0   -0.2658   -0.1644   -0.1644    0.1576    0.1576    0.1576    0.1576    0.1576
  -0.2658         0   -0.1644   -0.1644    0.1576    0.1576    0.1576    0.1576    0.1576
  -0.1644   -0.1644         0   -0.1931    0.1576    0.1576    0.1576    0.1576    0.1576
  -0.1644   -0.1644   -0.1931         0    0.1576    0.1576    0.1576    0.1576    0.1576
   0.1576    0.1576    0.1576    0.1576         0   -0.4978   -0.1945   -0.1145    0.0668
   0.1576    0.1576    0.1576    0.1576   -0.4978         0   -0.1945   -0.1145    0.0668
   0.1576    0.1576    0.1576    0.1576   -0.1945   -0.1945         0   -0.1145    0.0668
   0.1576    0.1576    0.1576    0.1576   -0.1145   -0.1145   -0.1145         0    0.0668
   0.1576    0.1576    0.1576    0.1576    0.0668    0.0668    0.0668    0.0668         0


orderperm =

     6     1     7     2     9     5     4     8     3
```

Partition	Level
{{6:Br,1:St,7:So,2:Gi,9:Sc,5:Th,4:Re,8:Ke,3:Oc}}	.1576
{{6:Br,1:St,7:So,2:Gi},{9:Sc,5:Th,4:Re,8:Ke,3:Oc}}	.0668
{{6:Br,1:St,7:So,2:Gi},{9:Sc,5:Th,4:Re,8:Ke},{3:Oc}}	−.1145
{{6:Br,1:St,7:So,2:Gi},{9:Sc,5:Th,4:Re},{8:Ke},{3:Oc}}	−.1644
{{6:Br,1:St},{7:So,2:Gi},{9:Sc,5:Th,4:Re},{8:Ke},{3:Oc}}	−.1931
{{6:Br,1:St},{7:So},{2:Gi},{9:Sc,5:Th,4:Re},{8:Ke},{3:Oc}}	−.1945
{{6:Br,1:St},{7:So},{2:Gi},{9:Sc,5:Th},{4:Re},{8:Ke},{3:Oc}}	−.2658
{{6:Br},{1:St},{7:So},{2:Gi},{9:Sc,5:Th},{4:Re},{8:Ke},{3:Oc}}	−.4978
{{6:Br},{1:St},{7:So},{2:Gi},{9:Sc},{5:Th},{4:Re},{8:Ke},{3:Oc}}	—

Appendix B

Generalizations of Ultrametrics and Additive Trees

B.1 Multiple Tree Structures

The use of multiple structures to represent additively a given proximity matrix, whether they be ultrametrics or additive trees, proceeds directly through successive residualization and iteration. We restrict ourselves to the fitting of two such structures but the same process would apply for any such number. Initially, a first matrix is fitted to a given proximity matrix and a first residual matrix obtained; a second structure is then fitted to these first residuals, producing a second residual matrix. Iterating, the second fitted matrix is now subtracted from the original proximity matrix and a first (re)fitted matrix obtained; this first (re)fitted matrix in turn is subtracted from the original proximity matrix and a new second matrix (re)fitted. This process continues until the `vaf` for the sum of both fitted matrices no longer changes substantially.

The M-files, `biultrafnd.m` and `biatreefnd.m` fit (additively) two ultrametric or additive tree matrices in the least-squares sense. The explicit usages are

```
[find,vaf,targone,targtwo] = biultrafnd(prox,inperm)
```

```
[find,vaf,targone,targtwo] = biatreefnd(prox,inperm)
```

where `prox` is the given input proximity matrix (with a zero main diagonal and a dissimilarity interpretation); `inperm` is a permutation that determines the order in which the inequality constraints are considered (and thus can be made random to search for different locally optimal representations); `find` is the obtained least-squares matrix (with variance-accounted-for of `vaf`) to `prox`, and is the sum of the two ultrametric or additive tree matrices `targone` and `targtwo`.

We will not given an explicit illustration of using two fitted structures to represent a proximity matrix. The data set we have been using, `supreme_agree.dat`, is not a good example for multiple structures because only one such device is really needed to explain everything present in the data. More suitable proximity matrices would probably themselves be obtained by a mixture or aggregation of other proximity matrices, reflecting somewhat different underlying structures; hopefully, these could be "teased apart" in an analysis using multiple additive structures.

B.2 Individual Differences

One aspect of the given M-files introduced in earlier sections but not emphasized, is their possible use in the confirmatory context of fitting individual differences. Explicitly, we begin with a collection of, say, N proximity matrices, $\mathbf{P}_1, \ldots, \mathbf{P}_N$, obtained from N separate sources, and through some weighting and averaging process, construct a single aggregate proximity matrix, $\mathbf{P}_A$. On the basis of $\mathbf{P}_A$, suppose an ultrametric or additive tree is constructed; we label the latter the "common space" consistent with what is usually done in the (weighted) Euclidean model in multidimensional scaling. Each of the N proximity matrices then can be used in a confirmatory fitting of an ultrametric (with, say, `ultrafit.m`) or an additive tree (with, say, `atreefit.m`). A very general "subject/private space" is generated for each source and where the branch lengths are unique to that source, subject only to the topology (branching) constraints of the group space. In effect, we would be carrying out an individual differences analysis by using a "deviation from the mean" philosophy. A group structure is first identified in an exploratory manner from an aggregate proximity matrix; the separate matrices that went into the aggregate are then fit in a confirmatory way, one-by-one. There does not seem to be any particular a priori advantage in trying to carry out this process "all at once"; to the contrary, the simplicity of the deviation approach and its immediate generalizability to a variety of possible structural representations, holds out the hope of greater substantive interpretability.

B.3 Transformations on Proximities

In the use of either a one- or two-mode proximity matrix, the data were assumed "as is", and without any preliminary transformation. It was noted that some analyses leading to negative values might be more pleasingly interpretable if they were positive, and thus, a sufficiently large additive constant was imposed on the original proximities (without any real loss of generality). In other words, the structures fit to proximity matrices have an invariance with respect to linear transformations of the proximities. A second type of transformation is implicit in the use of additive trees where a centroid (metric), fit as part of the whole representational structure, has the effect of double-centering (i.e., for the input proximity matrix deviated from the centroid, zero sums are present within rows or columns). In effect, the analysis methods iterate between fitting an ultrametric and a centroid, attempting to squeeze out every last bit of VAF. Maybe a more direct strategy (and one that would most likely not affect the substantive interpretations materially) would be to initially double-center (either a one- or two-mode matrix), and then treat the later to the analyses we wish to carry out, without again revisiting the double-centering operation during the iterative process.

A more serious consideration of proximity transformation would involve monotonic functions of the type familiar in nonmetric multidimensional scaling. We provide two utilities, `proxmon.m` and `proxmontm.m`, that will allow the user a chance to experiment with these more general transformations for both one- and two-mode proximity matrices. The usage is similar for both M-files in providing a monotonically transformed proximity matrix that is closest in a least-squares sense to a given (usually the structurally fitted) matrix:

```
[monproxpermut,vaf,diff] = proxmon(proxpermut,fitted)
```

```
[monproxpermuttm,vaf,diff] = proxmontm(proxpermuttm,fittedtm)
```

Here, `proxpermut` (`proxpermuttm`) is the input proximity matrix (which may have been subjected to an initial row/column permutation, hence the suffix `permut`), and `fitted` (`fittedtm`) is a given target matrix (typically the representational matrix such as the identified ultrametric); the output matrix, `monproxpermut` (`monproxpermuttm`), is closest to `fitted` (`fittedtm`) in a least-squares sense and obeys the order constraints obtained from each pair of entries in (the upper-triangular portion of) `proxpermut` or `proxpermuttm`. As usual, `vaf` denotes "variance-accounted-for" but here indicates how much variance in `monproxpermut` (`monproxpermuttm`) can be accounted for by `fitted` (`fittedtm`); finally, `diff` is the value of the least-squares loss function and is one-half the squared differences between the entries in `fitted` (`fittedtm`) and `monproxpermut` (`monproxpermuttm`).

A script M-file is listed in the verbatim output below that gives an application of `proxmon.m` using the best-fitting ultrametric structure found for `supreme_agree.dat`. First, `ultrafnd.m` is invoked to obtain a fitted matrix (`fit`) with `vaf` of .7369; `proxmon.m` then generates the monotonically transformed proximity matrix (`monproxpermut`) with `vaf` of .9264 and `diff` of .0622. The strategy is repeated one-hundred times (i.e., finding a fitted matrix based on the monotonically transformed proximity matrix, finding a new monotonically transformed matrix, and so on). To avoid degeneracy (where all matrices would just converge to zeros), the sum of squares of the fitted matrix is normalized (and held constant). Here, a perfect `vaf` of 1.0 is achieved (and a `diff` of 0.0); the structure of the proximities is now pretty flattened with only four new clusters explicitly identified in the partition hierarchy (the levels at which these are formed are given next to the clusters): (Sc,Th):.2149; (Gi,So):.2251; (Ke,Re):.2353; (Br,Gi,So):.2916. Another way of stating this is to observe that the monotonically transformed proximity matrix has only five distinct values, and over 86% are at a common value of .5588, the level at which the single all-inclusive subset is formed.

```
>> type ultra_monotone_test

load supreme_agree.dat [fit,vaf] = ultrafnd(supreme_agree,randperm(9))
```

```
[monprox,vaf,diff] = proxmon(supreme_agree,fit)

sumfitsq = sum(sum(fit.^2));

for i = 1:100

    [fit,vaf] = ultrafit(monprox,fit);

    sumnewfitsq = sum(sum(fit.^2));

    fit = sqrt(sumfitsq)*(fit/sumnewfitsq);

    [monprox,vaf,diff] = proxmon(supreme_agree,fit);

end

fit

vaf

diff

monprox

supreme_agree


>> ultra_monotone_test

fit =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369


monprox =

         0    0.3761    0.3633    0.3761    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3761         0    0.2850    0.2850    0.5211    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3761    0.2850    0.2200         0    0.5211    0.6405    0.6405    0.6405    0.6405
    0.6405    0.5211    0.6405    0.5211         0    0.3558    0.3100    0.5211    0.5211
    0.6405    0.6405    0.6405    0.6405    0.3558         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.3761    0.3558
    0.6405    0.6405    0.6405    0.6405    0.5211    0.4017    0.3761         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.5211    0.4017    0.3558    0.2100         0


vaf =

    0.9264
```

```
diff =

    0.0622


fit =

         0    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588         0    0.2916    0.2916    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916         0    0.2251    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916    0.2251         0    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588         0    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2353    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.2353         0    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2149
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.2149         0


vaf =

     1


diff =

     0


monprox =

         0    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588         0    0.2916    0.2916    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916         0    0.2251    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916    0.2251         0    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588         0    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2353    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.2353         0    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2149
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.2149         0


supreme_agree =

         0    0.3800    0.3400    0.3700    0.6700    0.6400    0.7500    0.8600    0.8500
    0.3800         0    0.2800    0.2900    0.4500    0.5300    0.5700    0.7500    0.7600
    0.3400    0.2800         0    0.2200    0.5300    0.5100    0.5700    0.7200    0.7400
    0.3700    0.2900    0.2200         0    0.4500    0.5000    0.5600    0.6900    0.7100
    0.6700    0.4500    0.5300    0.4500         0    0.3300    0.2900    0.4600    0.4600
    0.6400    0.5300    0.5100    0.5000    0.3300         0    0.2300    0.4200    0.4100
    0.7500    0.5700    0.5700    0.5600    0.2900    0.2300         0    0.3400    0.3200
    0.8600    0.7500    0.7200    0.6900    0.4600    0.4200    0.3400         0    0.2100
    0.8500    0.7600    0.7400    0.7100    0.4600    0.4100    0.3200    0.2100         0
```

B.4 Finding and Fitting Best Ultrametrics in the Presence of Missing Proximities

The various M-files discussed thus far have required proximity matrices to be complete in the sense of having all entries present. This was true even for the two-mode case where the between-set proximities are assumed available although all within-set proximities were not. Three different M-files are mentioned here (analogues of `order.m` (see Appendix D), `ultrafit.m`, and `ultrafnd.m`) allowing some of the proximities in a symmetric matrix to be absent. The missing proximities are identified in an input matrix, `proxmiss`, having the same size as the input proximity matrix, `prox`, but otherwise the syntaxes are the same as earlier:

```
[outperm,rawindex,allperms,index] = ...
order_missing(prox,targ,inperm,kblock,proxmiss)

[fit,vaf] = ultrafit_missing(prox,targ,proxmiss)

[find,vaf] = ultrafnd_missing(prox,inperm,proxmiss)
```

The `proxmiss` matrix guides the search and fitting process so the missing data are ignored whenever they should be considered in some kind of comparison. Typically, there will be enough other data available that this really doesn't pose any difficulty.

As an illustration of the M-files just introduced, Table B.1 provides data on the ten supreme court justices present at some point during the 2005/6 term, and the percentage of times justices disagreed in non-unanimous decisions during the year. (These data were in the *New York Times* on July 2, 2006, as part of a "first-page, above-the-fold" article bylined by Linda Greenhouse entitled "Roberts Is at Court's Helm, But He Isn't Yet in Control".) There is a single missing value in the table between O'Connor (Oc) and Alito (Al) because they shared a common seat for the term until Alito's confirmation by Congress. Roberts (Ro) served the full year as Chief Justice so no missing data entries involve him. As can be seen in the verbatim output to follow, an empirically obtained ordering (presumably from "left" to "right") using

	St	So	Br	Gi	Oc	Ke	Ro	Sc	Al	Th
1 St	.00	.28	.32	.31	.43	.62	.74	.70	.87	.76
2 So	.28	.00	.17	.36	.14	.50	.61	.64	.64	.75
3 Br	.32	.17	.00	.36	.29	.57	.56	.59	.65	.70
4 Gi	.31	.36	.36	.00	.43	.47	.52	.61	.59	.72
5 Oc	.43	.14	.29	.43	.00	.43	.33	.29	*	.43
6 Ke	.62	.50	.57	.47	.43	.00	.29	.35	.13	.41
7 Ro	.74	.61	.56	.52	.33	.29	.00	.12	.09	.18
8 Sc	.70	.64	.59	.61	.29	.35	.12	.00	.22	.16
9 Al	.87	.64	.65	.59	*	.13	.09	.22	.00	.17
10 Th	.76	.75	.70	.72	.43	.41	.18	.16	.17	.00

Table B.1: Dissimilarities Among Ten Supreme Court Justices for the 2005/6 Term. The Missing Entry Between O'Connor and Alito is Represented With an Asterisk.

`order_missing.m` is

$$1{:}\mathrm{St} \succ 4{:}\mathrm{Gi} \succ 3{:}\mathrm{Br} \succ 2{:}\mathrm{So} \succ 5{:}\mathrm{Oc} \succ 6{:}\mathrm{Ke} \succ 7{:}\mathrm{Ro} \succ 8{:}\mathrm{Sc} \succ 9{:}\mathrm{Al} \succ 10{:}\mathrm{Th}$$

suggesting rather strongly that Kennedy will most likely now occupy the middle position (although possibly shifted somewhat to the right) once O'Connor is removed from the court's deliberations. The best-fitting ultrametric obtained with `ultrafnd_missing.m` has VAF of 72.75%, and is given below in partition hierarchy form using the justice ordering from `order_missing.m`, except for the slight interchange of Sc and Al (this allows the fitted ultrametric to display its perfect AR form, as the verbatim output shows).

```
>> load supreme_agree_2005_6.dat

>> load supreme_agree_2005_6_missing.dat

>> supreme_agree_2005_6

supreme_agree_2005_6 =

         0    0.2800    0.3200    0.3100    0.4300    0.6200    0.7400    0.7000    0.8700    0.7600
    0.2800         0    0.1700    0.3600    0.1400    0.5000    0.6100    0.6400    0.6400    0.7500
    0.3200    0.1700         0    0.3600    0.2900    0.5700    0.5600    0.5900    0.6500    0.7000
    0.3100    0.3600    0.3600         0    0.4300    0.4700    0.5200    0.6100    0.5900    0.7200
    0.4300    0.1400    0.2900    0.4300         0    0.4300    0.3300    0.2900         0    0.4300
    0.6200    0.5000    0.5700    0.4700    0.4300         0    0.2900    0.3500    0.1300    0.4100
    0.7400    0.6100    0.5600    0.5200    0.3300    0.2900         0    0.1200    0.0900    0.1800
    0.7000    0.6400    0.5900    0.6100    0.2900    0.3500    0.1200         0    0.2200    0.1600
    0.8700    0.6400    0.6500    0.5900         0    0.1300    0.0900    0.2200         0    0.1700
    0.7600    0.7500    0.7000    0.7200    0.4300    0.4100    0.1800    0.1600    0.1700         0

>> supreme_agree_2005_6_missing

supreme_agree_2005_6_missing =
     0     1     1     1     1     1     1     1     1     1
     1     0     1     1     1     1     1     1     1     1
     1     1     0     1     1     1     1     1     1     1
     1     1     1     0     1     1     1     1     1     1
     1     1     1     1     0     1     1     1     0     1
     1     1     1     1     1     0     1     1     1     1
     1     1     1     1     1     1     0     1     1     1
     1     1     1     1     1     1     1     0     1     1
     1     1     1     1     0     1     1     1     0     1
     1     1     1     1     1     1     1     1     1     0

>> [outperm,rawindex,allperms,index] = ...
order_missing(supreme_agree_2005_6,targlin(10),randperm(10),3,supreme_agree_2005_6_missing);

>> outperm

outperm =

    10     9     8     7     6     5     2     3     4     1

>> [find,vaf] = ultrafnd_missing(supreme_agree_2005_6,randperm(10),supreme_agree_2005_6_missing)

find =

         0    0.3633    0.3633    0.3100    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633         0    0.2300    0.3633    0.1400    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.2300         0    0.3633    0.2300    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3100    0.3633    0.3633         0    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.1400    0.2300    0.3633         0    0.5954    0.5954    0.5954         0    0.5954
    0.5954    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.2950    0.2950    0.2950
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950         0    0.1725    0.0900    0.1725
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725         0    0.1725    0.1600
    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.0900    0.1725         0    0.1725
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725    0.1600    0.1725         0


vaf =

    0.7275

>> find([1 4 3 2 5 6 7 9 8 10],[1 4 3 2 5 6 7 9 8 10])

ans =

         0    0.3100    0.3633    0.3633    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3100         0    0.3633    0.3633    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.3633         0    0.2300    0.2300    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.3633    0.2300         0    0.1400    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.3633    0.2300    0.1400         0    0.5954    0.5954         0    0.5954    0.5954
    0.5954    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.2950    0.2950    0.2950
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950         0    0.0900    0.1725    0.1725
    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.0900         0    0.1725    0.1725
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725    0.1725         0    0.1600
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725    0.1725    0.1600         0
```

Partition	Level
{{1:St,4:Gi,3:Br,2:So,5:Oc,6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.5954
{{1:St,4:Gi,3:Br,2:So,5:Oc},{6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.3633
{{1:St,4:Gi},{3:Br,2:So,5:Oc},{6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.3100
{{1:St},{4:Gi},{3:Br,2:So,5:Oc},{6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.2950
{{1:St},{4:Gi},{3:Br,2:So,5:Oc},{6:Ke},{7:Ro,9:Al,8:Sc,10:Th}}	.2300
{{1:St},{4:Gi},{3:Br},{2:So,5:Oc},{6:Ke},{7:Ro,9:Al,8:Sc,10:Th}}	.1725
{{1:St},{4:Gi},{3:Br},{2:So,5:Oc},{6:Ke},{7:Ro,9:Al},{8:Sc,10:Th}}	.1600
{{1:St},{4:Gi},{3:Br},{2:So,5:Oc},{6:Ke},{7:Ro,9:Al},{8:Sc},{10:Th}}	.1400
{{1:St},{4:Gi},{3:Br},{2:So},{5:Oc},{6:Ke},{7:Ro,9:Al},{8:Sc},{10:Th}}	.0900
{{1:St},{4:Gi},{3:Br},{2:So},{5:Oc},{6:Ke},{7:Ro},{9:Al},{8:Sc},{10:Th}}	—

Appendix C

Ultrametric Extensions By Fitting Partitions Containing Contiguous Subsets

The M-file, `partitionfit.m`, is a very general routine giving a least-squares approximation to a proximity matrix based on a given collection of partitions. Thus, no matter how the set of candidate partitions might be chosen, a least-squares fitted matrix to the given proximity matrix is achieved. For example, if we simply use the nested partitions constructed from an ultrametric, the ultrametric would be retrieved when the latter is used as the input proximity matrix. In this section, we show how `partitionfit.m` can also be used to select partitions from a predefined set (this selection is done by those partitions assigned strictly positive weights) that might serve to reconstruct the proximity matrix well. The M-file, `consec_subsetfit.m`, defines $(n(n-1)/2)-1$ candidate partitions each characterized by a single contiguous cluster of objects, with all objects before and after this contiguous set forming individual clusters of the partition (the minus 1 appears in the count due to the (conjoint) partition defined by a single contiguous set being excluded). The M-file, `consec_subsetfit_alter.m`, varies the specific definition of the partitions by including all objects before and all objects after the contiguous set (when nonempty) as separate individual clusters of the partitions.

As can be seen from the verbatim output provided below, the nonnegative weighted partitions from `consec_subsetfit.m`, producing a fitted matrix with VAF of 92.61%, are as follows:

Partition	Partition Increment
{{St,Br,Gi,So},{Oc},{Ke},{Re},{Sc},{Th}}	.1939
{{St,Br,Gi,So,Oc},{Ke},{Re},{Sc},{Th}}	.0300
{{St,Br,Gi,So,Oc,Ke},{Re},{Sc},{Th}}	.0389
{{St,Br,Gi,So,Oc,Ke,Re},{Sc},{Th}}	.1315
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1152
{{St},{Br},{Gi,So,Oc,Ke,Re,Sc,Th}}	.0052
{{St},{Br},{Gi},{So,Oc,Ke,Re,Sc,Th}}	.0153
{{St},{Br},{Gi},{So},{Oc,Ke,Re,Sc,Th}}	.2220
{{St},{Br},{Gi},{So},{Oc},{Ke,Re,Sc,Th}}	.0633
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re,Sc,Th}}	.0030

Similarly, we have a very high VAF of 98.12% based on the more numerous partitions generated from `consec_subsetfit_alter.m`:

Partition	Partition Increment
{{St,Br},{Gi,So,Oc,Ke,Re,Sc,Th}}	.0021
{{St,Br,Gi},{So,Oc,Ke,Re,Sc,Th}}	.0001
{{St,Br,Gi,So},{Oc,Ke,Re,Sc,Th}}	.0001
{{St,Br,Gi,So,Oc,Ke},{Re,Sc,Th}}	.0100
{{St,Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.1218
{{St},{Br,Gi},{So,Oc,Ke,Re,Sc,Th}}	.0034
{{St},{Br,Gi,So,Oc},{Ke,Re,Sc,Th}}	.0056
{{St},{Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.0113
{{St},{Br,Gi,So,Oc,Ke,Re,Sc},{Th}}	.0038
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1170
{{St,Br},{Gi,So},{Oc,Ke,Re,Sc,Th}}	.0165
{{St,Br},{Gi,So,Oc,Ke,Re,Sc,Th}}	.0095
{{St,Br,Gi},{So,Oc},{Ke,Re,Sc,Th}}	.0197
{{St,Br,Gi},{So,Oc,Ke,Re,Sc,Th}}	.0115
{{St,Br,Gi,So},{Oc,Ke,Re,Sc,Th}}	.2294
{{St,Br,Gi,So,Oc},{Ke,Re,Sc,Th}}	.0353
{{St,Br,Gi,So,Oc,Ke},{Re,Sc,Th}}	.0400
{{St,Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.0132
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re},{Sc},{Th}}	.2050

```
>> load supreme_agree.dat
>> [fitted,vaf,weights,end_condition,member] = consec_subsetfit(supreme_agree);
>> fitted

fitted =

         0    0.4239    0.4239    0.4239    0.6178    0.6478    0.6866    0.8181    0.8181
    0.4239         0    0.3087    0.3087    0.5026    0.5326    0.5715    0.7029    0.7029
    0.4239    0.3087         0    0.3035    0.4974    0.5274    0.5663    0.6977    0.6977
    0.4239    0.3087    0.3035         0    0.4821    0.5121    0.5510    0.6824    0.6824
    0.6178    0.5026    0.4974    0.4821         0    0.2901    0.3290    0.4604    0.4604
    0.6478    0.5326    0.5274    0.5121    0.2901         0    0.2657    0.3972    0.3972
    0.6866    0.5715    0.5663    0.5510    0.3290    0.2657         0    0.3942    0.3942
    0.8181    0.7029    0.6977    0.6824    0.4604    0.3972    0.3942         0    0.3942
    0.8181    0.7029    0.6977    0.6824    0.4604    0.3972    0.3942    0.3942         0

>> vaf

vaf =

    0.9261

>> weights
```

```
weights =

         0
         0
    0.1939
    0.0300
    0.0389
    0.1315
         0
         0
         0
         0
         0
         0
         0
    0.1152
         0
         0
         0
         0
         0
    0.0052
         0
         0
         0
         0
    0.0153
         0
         0
         0
    0.2220
         0
         0
    0.0633
         0
    0.0030
         0
         0

>> end_condition

end_condition =

     0

>> member

member =

     1     1     3     4     5     6     7     8     9
     1     1     1     4     5     6     7     8     9
     1     1     1     1     5     6     7     8     9
     1     1     1     1     1     6     7     8     9
     1     1     1     1     1     1     7     8     9
     1     1     1     1     1     1     1     8     9
     1     1     1     1     1     1     1     1     9
     1     2     2     4     5     6     7     8     9
     1     2     2     2     5     6     7     8     9
     1     2     2     2     2     6     7     8     9
     1     2     2     2     2     2     7     8     9
     1     2     2     2     2     2     2     8     9
     1     2     2     2     2     2     2     2     9
     1     2     2     2     2     2     2     2     2
     1     2     3     3     5     6     7     8     9
     1     2     3     3     3     6     7     8     9
     1     2     3     3     3     3     7     8     9
     1     2     3     3     3     3     3     8     9
     1     2     3     3     3     3     3     3     9
     1     2     3     3     3     3     3     3     3
     1     2     3     4     4     6     7     8     9
     1     2     3     4     4     4     7     8     9
     1     2     3     4     4     4     4     8     9
     1     2     3     4     4     4     4     4     9
     1     2     3     4     4     4     4     4     4
     1     2     3     4     5     5     7     8     9
     1     2     3     4     5     5     5     8     9
     1     2     3     4     5     5     5     5     9
     1     2     3     4     5     5     5     5     5
     1     2     3     4     5     6     6     8     9
     1     2     3     4     5     6     6     6     9
     1     2     3     4     5     6     6     6     6
     1     2     3     4     5     6     7     7     9
     1     2     3     4     5     6     7     7     7
     1     2     3     4     5     6     7     8     8
     1     2     3     4     5     6     7     8     9

>> [fitted,vaf,weights,end_condition,member] = consec_subsetfit_alter(supreme_agree)

fitted =

         0    0.3460    0.3740    0.4053    0.6347    0.6700    0.7200    0.8550    0.8550
    0.3460         0    0.2330    0.2677    0.4971    0.5380    0.5880    0.7342    0.7380
    0.3740    0.2330         0    0.2396    0.4855    0.5264    0.5764    0.7227    0.7264
    0.4053    0.2677    0.2396         0    0.4509    0.5114    0.5614    0.7076    0.7114
    0.6347    0.4971    0.4855    0.4509         0    0.2655    0.3155    0.4617    0.4655
    0.6700    0.5380    0.5264    0.5114    0.2655         0    0.2550    0.4012    0.4050
    0.7200    0.5880    0.5764    0.5614    0.3155    0.2550         0    0.3512    0.3550
    0.8550    0.7342    0.7227    0.7076    0.4617    0.4012    0.3512         0    0.2087
    0.8550    0.7380    0.7264    0.7114    0.4655    0.4050    0.3550    0.2087         0


vaf =

    0.9812


weights =

    0.0021
    0.0001
    0.0001
         0
    0.0100
    0.1218
         0
    0.0034
         0
    0.0056
         0
    0.0113
    0.0038
    0.1170
    0.0165
         0
         0
         0
         0
    0.0095
    0.0197
         0
         0
         0
    0.0115
         0
         0
         0
    0.2294
         0
         0
    0.0353
         0
    0.0400
    0.0132
    0.2050


end_condition =

     0


member =

     1     1     9     9     9     9     9     9     9
     1     1     1     9     9     9     9     9     9
     1     1     1     1     9     9     9     9     9
     1     1     1     1     1     9     9     9     9
     1     1     1     1     1     1     9     9     9
     1     1     1     1     1     1     1     9     9
     1     1     1     1     1     1     1     1     9
     1     2     2     9     9     9     9     9     9
     1     2     2     2     9     9     9     9     9
     1     2     2     2     2     9     9     9     9
     1     2     2     2     2     2     9     9     9
     1     2     2     2     2     2     2     9     9
     1     2     2     2     2     2     2     2     9
     1     2     2     2     2     2     2     2     2
     1     1     3     3     9     9     9     9     9
     1     1     3     3     3     9     9     9     9
     1     1     3     3     3     3     9     9     9
     1     1     3     3     3     3     3     9     9
     1     1     3     3     3     3     3     3     9
     1     1     3     3     3     3     3     3     3
     1     1     1     4     4     9     9     9     9
     1     1     1     4     4     4     9     9     9
     1     1     1     4     4     4     4     9     9
     1     1     1     4     4     4     4     4     9
     1     1     1     4     4     4     4     4     4
     1     1     1     1     5     5     9     9     9
     1     1     1     1     5     5     5     9     9
     1     1     1     1     5     5     5     5     9
     1     1     1     1     5     5     5     5     5
     1     1     1     1     1     6     6     9     9
     1     1     1     1     1     6     6     6     9
     1     1     1     1     1     6     6     6     6
     1     1     1     1     1     1     7     7     9
     1     1     1     1     1     1     7     7     7
     1     1     1     1     1     1     1     8     8
     1     2     3     4     5     6     7     8     9
```

To see how well one might do by choosing only eight partitions (i.e., the same number needed to define the order-constrained best-fitting ultrametric), the single (disjoint) partition defined by nine separate classes is chosen in both instances, plus the seven partitions having the highest assigned positive weights. For those picked from the partition pool identified by `consec_subsetfit.m`, the VAF drops slightly from 92.61% to 92.51% based on the partitions:

Partition	Partition Increment
{{St,Br,Gi,So},{Oc},{Ke},{Re},{Sc},{Th}}	.1923
{{St,Br,Gi,So,Oc},{Ke},{Re},{Sc},{Th}}	.0301
{{St,Br,Gi,So,Oc,Ke},{Re},{Sc},{Th}}	.0396
{{St,Br,Gi,So,Oc,Ke,Re},{Sc},{Th}}	.1316
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1224
{{St},{Br},{Gi},{So},{Oc,Ke,Re,Sc,Th}}	.2250
{{St},{Br},{Gi},{So},{Oc},{Ke,Re,Sc,Th}}	.0671
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re},{Sc},{Th}}	.0000

For those selected from the set generated by `consec_subsetfit_alter.m`, the VAF again drops slightly from 98.12% to 97.97%. But in some absolute sense given the size of the VAF, the eight partitions listed below seem to be about all that can be extracted from this particular justice data set.

Partition	Partition Increment
{{St,Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.1466
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1399
{{St,Br},{Gi,So},{Oc,Ke,Re,Sc,Th}}	.0287
{{St,Br,Gi},{So,Oc},{Ke,Re,Sc,Th}}	.0326
{{St,Br,Gi,So},{Oc,Ke,Re,Sc,Th}}	.2269
{{St,Br,Gi,So,Oc},{Ke,Re,Sc,Th}}	.0316
{{St,Br,Gi,So,Oc,Ke},{Re,Sc,Th}}	.0500
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re},{Sc},{Th}}	.2051

The three highest weighted partitions have very clear interpretations: {Sc,Th} versus the rest; {St} versus the rest; {St,Br,Gi,So} as the left versus {Oc,Ke,Re,Sc,Th} as the right. The few remaining partitions revolve around several other less salient (adjacent) object pairings that are also very interpretable in relation to the object ordering from liberal to conservative.

```
>> member = [1 1 1 1 5 6 7 8 9;1 1 1 1 1 6 7 8 9;1 1 1 1 1 1 7 8 9;1 1 1 1 1 1 1 8 9;
1 2 2 2 2 2 2 2 2;1 2 3 4 5 5 5 5 5;1 2 3 4 5 6 6 6 6;1 2 3 4 5 6 7
8 9]

member =
     1     1     1     1     5     6     7     8     9
     1     1     1     1     1     6     7     8     9
     1     1     1     1     1     1     7     8     9
     1     1     1     1     1     1     1     8     9
     1     2     2     2     2     2     2     2     2
     1     2     3     4     5     5     5     5     5
     1     2     3     4     5     6     6     6     6
     1     2     3     4     5     6     7     8     9

>> [fitted,vaf,weights,end_condition] = partitionfit(supreme_agree,member)

fitted =

         0    0.4245    0.4245    0.4245    0.6168    0.6469    0.6865    0.8181    0.8181
    0.4245         0    0.3021    0.3021    0.4944    0.5245    0.5641    0.6957    0.6957
    0.4245    0.3021         0    0.3021    0.4944    0.5245    0.5641    0.6957    0.6957
    0.4245    0.3021    0.3021         0    0.4944    0.5245    0.5641    0.6957    0.6957
    0.6168    0.4944    0.4944    0.4944         0    0.2895    0.3291    0.4607    0.4607
    0.6469    0.5245    0.5245    0.5245    0.2895         0    0.2620    0.3936    0.3936
    0.6865    0.5641    0.5641    0.5641    0.3291    0.2620         0    0.3936    0.3936
    0.8181    0.6957    0.6957    0.6957    0.4607    0.3936    0.3936         0    0.3936
    0.8181    0.6957    0.6957    0.6957    0.4607    0.3936    0.3936    0.3936         0


vaf =

    0.9251


weights =

    0.1923
    0.0301
    0.0396
    0.1316
    0.1224
    0.2350
    0.0671
         0


end_condition =

     0

>> member = [1 1 1 1 1 1 1 9 9;1 2 2 2 2 2 2 2 2;1 1 3 3 9 9 9 9 9;1 1 1 4 4 9 9 9 9;
1 1 1 1 5 5 5 5 5;1 1 1 1 1 6 6 6 6;1 1 1 1 1 1 7 7 7;1 2 3 4 5 6 7
8 9]

member =

     1     1     1     1     1     1     1     9     9
     1     2     2     2     2     2     2     2     2
     1     1     3     3     9     9     9     9     9
     1     1     1     4     4     9     9     9     9
     1     1     1     1     5     5     5     5     5
     1     1     1     1     1     6     6     6     6
     1     1     1     1     1     1     7     7     7
     1     2     3     4     5     6     7     8     9

>> [fitted,vaf,weights,end_condition] = partitionfit(supreme_agree,member)

fitted =
         0    0.3450    0.3736    0.4062    0.6331    0.6647    0.7147    0.8613    0.8613
    0.3450         0    0.2337    0.2664    0.4933    0.5248    0.5748    0.7215    0.7215
    0.3736    0.2337         0    0.2377    0.4933    0.5248    0.5748    0.7215    0.7215
    0.4062    0.2664    0.2377         0    0.4606    0.5248    0.5748    0.7215    0.7215
    0.6331    0.4933    0.4933    0.4606         0    0.2693    0.3193    0.4659    0.4659
    0.6647    0.5248    0.5248    0.5248    0.2693         0    0.2551    0.4017    0.4017
    0.7147    0.5748    0.5748    0.5748    0.3193    0.2551         0    0.3517    0.3517
    0.8613    0.7215    0.7215    0.7215    0.4659    0.4017    0.3517         0    0.2051
    0.8613    0.7215    0.7215    0.7215    0.4659    0.4017    0.3517    0.2051         0


vaf =

    0.9797


weights =

    0.1466
    0.1399
    0.0287
    0.0326
    0.2269
    0.0316
    0.0500
    0.2051


end_condition =

     0
```

Appendix D

A Useful Utility: Obtaining a Constraining Order

In implementing an order-constrained K-means clustering strategy, an appropriate initial ordering must be generated to constrain the clustering in the first place. Although many strategies might be considered, a particularly powerful one appears definable through what is called the quadratic assignment (QA) task and a collection of local-improvement optimization heuristics. As typically defined, a QA problem involves two $n \times n$ matrices, $\mathbf{A} = \{a_{ij}\}$ and $\mathbf{T} = \{t_{ij}\}$, and we seek a permutation to maximize the cross-product statistic

$$\Gamma(\rho) = \sum_{i \neq j} a_{\rho(i)\rho(j)} t_{ij} \ . \qquad \text{(D.1)}$$

The notation $\{a_{\rho(i)\rho(j)}\}$ implies a reordering (by the permutation $\rho(\cdot)$) of the rows and simultaneously the columns of $\mathbf{A}$ so that the rows (and columns) now appear in the order $\rho(1) \succ \rho(2) \succ \cdots \succ \rho(n)$. For our purposes, the first matrix $\mathbf{A}$ could be identified with the proximity matrix $\mathbf{P}$ containing squared Euclidean distances between the subject profiles over the p variables; the second matrix contains a target defined by a set of locations equally-spaced along a line, i.e., $\mathbf{T} = \{|j - i|\}$ for $1 \leq i, j \leq n$. (More generally, $\mathbf{P}$ could be any proximity matrix having a dissimilarity interpretation; use of the resulting identified permutation, for example, would be one way of implementing an order-constrained DP proximity matrix subdivision.)

In attempting to find ρ to maximize $\Gamma(\rho)$, we try to reorganize the proximity matrix as $\mathbf{P}_\rho = \{p_{\rho(i)\rho(j)}\}$, to show the same pattern, more or less, as the fixed target $\mathbf{T}$; equivalently, we maximize the usual Pearson product-moment correlation between the off-diagonal entries in $\mathbf{T}$ and $\mathbf{P}_\rho$. Another way of rephrasing this search is to say that we seek a permutation ρ that provides a structure as "close" as possible to an AR form for $\mathbf{P}_\rho$, i.e., the degree to which the entries in $\mathbf{P}_\rho$, moving away from the main diagonal in either direction never decrease (and usually increase); this is exactly the pattern exhibited by the equally-spaced target matrix $\mathbf{T}$. In our order-constrained K-means application, once the proximity matrix is so reordered by ρ, we look for a K-means clustering result that respects the order generating the "as close as we can get to an AR" patterning for the row/column permuted matrix.

The type of heuristic optimization strategy we use for the QA task implements simple object interchange/rearrangement operations. Based on given matrices $\mathbf{A}$ and $\mathbf{T}$, and beginning with some permutation (possibly chosen at random), local interchanges and rearrangements of a particular type are implemented until no improvement in the index can be made. By repeatedly initializing such a process randomly, a distribution over a set of local optima can be achieved. Three different classes of local operations are used in the M-file, `order.m`: (i) the pairwise interchanges of objects in the current permutation defining the row and column order of the data matrix $\mathbf{A}$. All possible such interchanges are generated and considered in turn, and whenever an increase in the cross-product index would result from a particular interchange, it is made immediately. The process continues until the current permutation cannot be improved upon by any such pairwise object interchange. The procedure then proceeds to (ii): the local operations considered are all reinsertions of from 1 to `kblock` (which is less than n and set by the user) consecutive objects somewhere in the permutation defining the current row and column order of the data matrix. When no further improvement can be made, we move to (iii): the local operations are now all possible rotations (or inversions) of from 2 to `kblock` consecutive objects in the current row/column order of the data matrix. (We suggest a use of `kblock` equal to 3 as a reasonable compromise between the extensiveness of local search, speed of execution, and quality of solution.) The three collections of local changes are revisited (in order) until no alteration is possible in the final permutation obtained.

The use of `order.m` is illustrated in the verbatim recording below, first on the squared Euclidean distance matrix among the ten cabernets (see Table 3.1) to produce the constraining order used in the order-constrained K-means clustering. Among the two local optima found, we will choose the one with the higher `rawindex` in (D.1) of 100458, and corresponding to the order in `outperm` of [9 10 7 8 5 6 3 2 1 4]. There are `index` permutations stored in the MATLAB cell-array `allperms`, from the first randomly generated one in `allperms{1}`, to the found local optimum in `allperms{index}`. (These have been suppressed in the output.) Notice, that retrieving entries in a cell-array requires the use of curly braces, `{,}`. The M-file, `targlin.m`, provides the equally-spaced target matrix as an input. We also show that starting with a random permutation and the `supreme_agree.dat` data matrix, the identity permutation is retrieved (in fact, it would be the sole local optimum found upon repeated starts using random permutations). It might be noted that an empirically constructed constraining order for an ultrametric (which leads in turn to a best-fitting AR matrix) is carried out with exactly this same type of QA routine (and used internally in the M-file, `ultrafnd_confnd.m`, discussed in an earlier chapter).

```
>> load cabernet_taste.dat

>> [sqeuclid] = sqeuclidean(cabernet_taste)

sqeuclid =

         0   48.2500   48.2500   86.5000  220.0000  160.2500  400.5000  394.0000  374.5000  485.2500
   48.2500         0   77.0000   77.2500  195.2500  176.0000  327.2500  320.2500  453.2500  403.0000
   48.2500   77.0000         0  131.2500  229.2500  107.0000  326.2500  410.2500  253.2500  362.0000
   86.5000   77.2500  131.2500         0  202.5000  202.2500  355.5000  305.5000  416.0000  465.2500
  220.0000  195.2500  229.2500  202.5000         0  145.2500   75.5000  102.0000  394.5000  190.2500
  160.2500  176.0000  107.0000  202.2500  145.2500         0  160.2500  250.2500  100.2500  147.0000
  400.5000  327.2500  326.2500  355.5000   75.5000  160.2500         0   79.5000  379.5000   76.2500
  394.0000  320.2500  410.2500  305.5000  102.0000  250.2500   79.5000         0  515.5000  201.2500
  374.5000  453.2500  253.2500  416.0000  394.5000  100.2500  379.5000  515.5000         0  279.2500
  485.2500  403.0000  362.0000  465.2500  190.2500  147.0000   76.2500  201.2500  279.2500         0

>> [outperm,rawindex,allperms,index] = order(sqeuclid,targlin(10),randperm(10),3)

outperm =

    10     8     7     5     6     2     4     3     1     9


rawindex =

      100333
```

```
index =

    11

>> [outperm,rawindex,allperms,index] = order(sqeuclid,targlin(10),randperm(10),3)

outperm =

     9    10     7     8     5     6     3     2     1     4


rawindex =

      100458


index =

    18

>> load supreme_agree.dat

>> [outperm,rawindex,allperms,index] = order(supreme_agree,targlin(9),randperm(9),3)

outperm =

     1     2     3     4     5     6     7     8     9


rawindex =

  145.1200


index =

    19
```

Appendix E

Some Bibliographic Comments

There are a number of book-length presentations of cluster analysis methods available (encompassing differing collections of subtopics within the field). We list several of the better ones to consult in the reference section to follow, and note these here in chronological order: Anderberg (1973); Hartigan (1975); Späth (1980); Barthélemy and Guénoche (1991); Mirkin (1996); Arabie, Hubert, and DeSoete (1996); Everitt and Rabe-Hesketh (1997); Gordon (1999). The items that would be closest to the approaches taken here with MATLAB and the emphasis on least-squares, would be the monograph by Hubert, Arabie, and Meulman (2006), and the review articles by Hubert, Köhn, and Steinley (2009, 2010); and Steinley and Hubert (2008).

Bibliography

[1] Anderberg, M. (1973). *Cluster analysis for applications.* New York: Academic Press.

[2] Arabie, P., Hubert, L., & DeSoete, G. (Eds.) (1996). *Clustering and classification.* River Edge, NJ: World Scientific.

[3] Barthélemy, J.-P., & Guénoche, A. (1991). *Trees and proximity representations.* Chichester: Wiley.

[4] Brossier, G. (1987). Étude des matrices de proximité rectangulaires en vue de la classification [A study of rectangular proximity matrices from the point of view of classification]. *Revue de Statistiques Appliquées, 35(4)*, 43–68.

[5] Everitt, B. S., & Rabe-Hesketh, S. (1997). *The analysis of proximity data.* New York: Wiley.

[6] Furnas, G. W. (1980). *Objects and their features: The metric representation of two class data.* Unpublished doctoral dissertation, Stanford University.

[7] Gordon, A. D. (1999). *Classification.* London: Chapman & Hall/CRC.

[8] Hartigan, J. (1975). *Clustering algorithms.* New York: Wiley.

[9] Hubert, L., Arabie, P., & Meulman, J. (2006). *The structural representation of proximity matrices with MATLAB.* ASA-SIAM Series on Statistics and Applied Probability. Philadelphia: SIAM.

[10] Hubert, L., Köhn, H.-F., & Steinley, D. (2009). Cluster analysis: A Toolbox for MATLAB. In R. E. Millsap & A. Maydeu-Olivares (Eds.),

The SAGE handbook of quantitative methods in psychology (pp. 444–511). Los Angeles: SAGE.

[11] Hubert, L., Köhn, H.-F., & Steinley, D. (2010). Order-constrained proximity matrix representations: Ultrametric generalizations and constructions with MATLAB. In S. Kolenikov, D. Steinley, & L. Thombs (Eds.), *Current methodological developments of statistics in the social sciences* (pp. 81–112). New York: Wiley.

[12] Mirkin, B. (1996). *Mathematical classification and clustering.* Dordrecht: Kluwer.

[13] Späth, H. (1980). *Cluster analysis algorithms.* Chichester, U.K.: Ellis Horwood.

[14] Steinley, D., & Hubert, L. (2008). Order constrained solutions in K-means clustering: Even better than being globally optimal. *Psychometrika*, *73*, 647–664.

[15] Wollan, P. C., & Dykstra, R. L. (1987). Minimizing linear inequality constrained Mahalanobis distances. *Applied Statistics*, *36*, 234–240.

Appendix F

Header Comments for the M-files Mentioned in the Text or Used Internally by Other M-files; Given in Alphabetical Order

arobfit.m

```
function [fit, vaf] = arobfit(prox, inperm)

% AROBFIT fits an anti-Robinson matrix using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit, vaf] = arobfit(prox, inperm)
%
% PROX is the input proximity matrix ($n \times n$ with a zero main
% diagonal and a dissimilarity interpretation);
% INPERM is a given permutation of the first $n$ integers;
% FIT is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX having an anti-Robinson form for
% the row and column object ordering given by INPERM.
```

arobfnd.m

```
function [find, vaf, outperm] = arobfnd(prox, inperm, kblock)

% AROBFND finds and fits an anti-Robinson
% matrix using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm based on a
% permutation identified through the use of iterative quadratic
% assignment.
%
% syntax: [find, vaf, outperm] = arobfnd(prox, inperm, kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero main
% diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation of the first $n$ integers;
% FIND is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX having an anti-Robinson
% form for the row and column object ordering given by the ending
% permutation OUTPERM. KBLOCK defines the block size in the use of the
% iterative quadratic assignment routine.
```

atreedec.m

```
function [ulmetric,ctmetric] = atreedec(prox,constant)

% ATREEDEC decomposes a given additive tree matrix into an
% ultrametric and a centroid metric matrix (where the root is
% halfway along the longest path).
%
% syntax: [ulmetric,ctmetric] = atreedec(prox,constant)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% CONSTANT is a nonnegative number (less than or equal to the
% maximum proximity value) that controls the
% positivity of the constructed ultrametric values;
% ULMETRIC is the ultrametric component of the decomposition;
% CTMETRIC is the centroid metric component of the decomposition
% (given by values $g_{1},...,g_{n}$ for each of the objects,
% some of which may actually be negative depending on the input
% proximity matrix used).
```

atreefit.m

```
function [fit,vaf] = atreefit(prox,targ)

% ATREEFIT fits a given additive tree using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit,vaf] = atreefit(prox,targ)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% TARG is a matrix of the same size as PROX with entries
% satisfying the four-point additive tree constraints;
% FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX satisfying the
% additive tree constraints implicit in TARG.
```

atreefnd.m

```
function [find,vaf] = atreefnd(prox,inperm)

% ATREEFND finds and fits an additive tree using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf] = atreefnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints.
```

atreefndtm.m

```
function [find,vaf,ultra,lengths] = ...
    atreefndtm(proxtm,inpermrow,inpermcol)

% ATREEFNDTM finds and fits a two-mode additive tree;
% iterative projection is used
% heuristically to find a two-mode ultrametric component that
% is added to a two-mode centroid metric to
% produce the two-mode additive tree.
%
% syntax: [find,vaf,ultra,lengths] = ...
%      atreefndtm(proxtm,inpermrow,inpermcol)
%
% PROXTM is the input proximity matrix
% (with a dissimilarity interpretation);
% INPERMROW and INPERMCOL are permutations for the row and column
% objects that determine the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROXTM satisfying the additive tree constraints;
% the vector LENGTHS contains the row followed by column values for
% the two-mode centroid metric component;
% ULTRA is the ultrametric component.
```

biatreefnd.m

```
function [find,vaf,targone,targtwo] = biatreefnd(prox,inperm)

% BIATREEFND finds and fits the sum
% of two additive trees using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf,targone,targtwo] = biatreefnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX and is the sum of
% the two additive tree matrices TARGONE and TARGTWO.
```

biultrafnd.m

```
function [find,vaf,targone,targtwo] = biultrafnd(prox,inperm)

% BIULTRAFND finds and fits the sum
% of two ultrametrics using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf,targone,targtwo] = biultrafnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX and is the sum
% of the two ultrametric matrices TARGONE and TARGTWO.
```

cent_ultrafnd_confit.m

```
function [find,vaf,outperm,targone,targtwo,lengthsone] = ...
```

```
    cent_ultrafnd_confit(prox,inperm,conperm)

% CENT_ULTRAFND_CONFIT finds and fits an additive tree by first fitting
% a centroid metric and secondly an ultrametric to the residual
% matrix where the latter is constrained by a given object order.
%
% syntax: [find,vaf,outperm,targone,targtwo,lengthsone] = ...
%      cent_ultrafnd_confit(prox,inperm,conperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); CONPERM is the given
% input constraining order (permutation) which is also given
% as the output vector OUTPERM;
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered in identifying the ultrametric;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints. TARGTWO is
% the ultrametric component of the decomposition; TARGONE is the centroid
% metric component defined by the lengths in LENGTHSONE.
```

cent_ultrafnd_confnd.m

```
function [find,vaf,outperm,targone,targtwo,lengthsone] = ...
    cent_ultrafnd_confnd(prox,inperm)

% CENT_ULTRAFND_CONFND finds and fits an additive tree by first fitting
% a centroid metric and secondly an ultrametric to the residual
% matrix where the latter is displayed by a constraining object order that
% is also identified in the process.
%
% syntax: [find,vaf,outperm,targone,targtwo,lengthsone] = ...
%      cent_ultrafnd_confnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered in identifying the ultrametric;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints. TARGTWO is
% the ultrametric component of the decomposition; TARGONE is the centroid
% metric component defined by the lengths in LENGTHSONE; OUTPERM is the
% identified constraining object order used to display the ultrametric
% component.
```

centfit.m

```
function [fit,vaf,lengths] = centfit(prox)

% CENTFIT finds the least-squares fitted centroid metric (FIT) to
% PROX, the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation).
%
% syntax: [fit,vaf,lengths] = centfit(prox)
%
% The $n$ values that serve to define the approximating sums,
% $g_{i} + g_{j}$, are given in the vector LENGTHS of size $n \times 1$.
```

centfittm.m

```
function [fit,vaf,lengths] = centfittm(proxtm)

% CENTFITTM finds the least-squares fitted two-mode centroid metric
% (FIT) to PROXTM, the two-mode rectangular input proximity matrix
% (with a dissimilarity interpretation).
%
% syntax: [fit,vaf,lengths] = centfittm(proxtm)
%
% The $n$ values (where $n$ = number of rows + number of columns)
% serve to define the approximating sums,
% $u_{i} + v_{j}$, where the $u_{i}$ are for the rows and the $v_{j}$
% are for the columns; these are given in  the vector LENGTHS of size
% $n \times 1$, with row values first followed by the column values.
```

consec_subsetfit.m

```
function [fitted,vaf,weights,end_condition,member] =
consec_subsetfit(prox)

% CONSEC_SUBSETFIT defines a collection of partitions involving
% consecutive subsets for the object set and then calls partitionfit.m
% to fit a least-squares approximation to the input proximity matrix based
% on these identified partitions.
%
% syntax [fitted,vaf,weights,end_condition,member] = consec_subsetfit(prox)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n matrix
% indicating cluster membership, where each row corresponds to a specific
% partition (there are m partitions in general); the columns of MEMBER
% are in the same input order used for PROX. The partitions are defined
% by a single contiguous cluster of objects, with all objects before and
% after this contiguous set forming individual clusters of the partitions.
% The value of m is (n*(n-1)/2) - 1; the partition defined by a single
% contiguous partition is excluded.
% FITTED is an n x n matrix fitted to PROX (through least-squares)
% constructed from the nonnegative weights given in the m x 1 WEIGHTS
% vector corresponding to each of the partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

consec_subsetfit_alter.m

```
function [fitted,vaf,weights,end_condition,member] =
consec_subsetfit_alter(prox)

% CONSEC_SUBSETFIT_ALTER defines a collection of partitions involving
% consecutive subsets for the object set and then calls partitionfit.m
% to fit a least-squares approximation to the input proximity matrix based
% on these identified partitions.
%
% syntax [fitted,vaf,weights,end_condition,member] = ...
%                     consec_subsetfit_alter(prox)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n matrix
% indicating cluster membership, where each row corresponds to a specific
% partition (there are m partitions in general); the columns of MEMBER
% are in the same input order used for PROX. The partitions are defined
% by a single contiguous cluster of objects, with all objects before and
% all objects after this contiguous set (when nonempty) forming
% separate individual clusters of the partitions.
% (These possible three-class partitions when before and after subsets are
% both nonempty) distinguish consec_subsetfit_alter.m from consec_subsetfit.m).
% The value of m is (n*(n-1)/2) - 1; the partition defined by a single
% contiguous partition is excluded.
% FITTED is an n x n matrix fitted to PROX (through least-squares)
% constructed from the nonnegative weights given in the m x 1 WEIGHTS
% vector corresponding to each of the partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

dykstra.m

```
code only; no help file

function [solution, kuhn_tucker, iterations, end_condition] = ...
    dykstra(data,covariance,constraint_array,constraint_constant,equality_flag)
```

insertqa.m

```
function [outperm, rawindex, allperms, index] = ...
   insertqa(prox, targ, inperm, kblock)

% INSERTQA carries out an iterative
% Quadratic Assignment maximization task using the
% insertion of from 1 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix.
%
% syntax: [outperm, rawindex, allperms, index] = ...
%   insertqa(prox, targ, inperm, kblock)
%
% INPERM is the input beginning permutation
% (a permutation of the first $n$ integers).
% PROX is the $n \times n$ input proximity matrix.
% TARG is the $n \times n$ input target matrix.
% OUTPERM is the final permutation of PROX with the cross-product
% index RAWINDEX with respect to TARG.
% ALLPERMS is a cell array containing INDEX entries corresponding
% to all the permutations identified in the optimization from
% ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.
```

order.m

```
function [outperm,rawindex,allperms,index] =
order(prox,targ,inperm,kblock)

% ORDER carries out an iterative Quadratic Assignment maximization
% task using a given square ($n x n$) proximity matrix PROX (with
% a zero main diagonal and a dissimilarity interpretation).
%
% syntax: [outperm,rawindex,allperms,index] = ...
%   order(prox,targ,inperm,kblock)
%
```

```
% Three separate local operations are used to permute
% the rows and columns of the proximity matrix to maximize the
% cross-product index with respect to a given square target matrix
% TARG: pairwise interchanges of objects in the permutation defining
% the row and column order of the square proximity matrix;
% the insertion of from 1 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix; the rotation of from 2 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix. INPERM is the input beginning permutation (a permutation
% of the first $n$ integers).
% OUTPERM is the final permutation of PROX with the
% cross-product index RAWINDEX
% with respect to TARG. ALLPERMS is a cell array containing INDEX
% entries corresponding to all the
% permutations identified in the optimization from ALLPERMS{1} =
% INPERM to ALLPERMS{INDEX} = OUTPERM.
```

order_missing.m

```
function [outperm,rawindex,allperms,index] =  ...
    order_missing(prox,targ,inperm,kblock,proxmiss)

% ORDER_MISSING carries out an iterative Quadratic Assignment maximization
% task using a given square ($n x n$) proximity matrix PROX (with
% a zero main diagonal and a dissimilarity interpretation; missing entries
% PROX are given values of zero).
%
% syntax: [outperm,rawindex,allperms,index] = ...
%   order_missing(prox,targ,inperm,kblock,proxmiss)
%
% Three separate local operations are used to permute
% the rows and columns of the proximity matrix to maximize the
% cross-product index with respect to a given square target matrix
% TARG: pairwise interchanges of objects in the permutation defining
% the row and column order of the square proximity matrix;
% the insertion of from 1 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix; the rotation of from 2 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix. INPERM is the input beginning permutation (a permutation
% of the first $n$ integers).  PROXMISS is the same size as PROX (with
% main diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
% OUTPERM is the final permutation of PROX with the
% cross-product index RAWINDEX
% with respect to TARG. ALLPERMS is a cell array containing INDEX
% entries corresponding to all the
% permutations identified in the optimization from ALLPERMS{1} =
% INPERM to ALLPERMS{INDEX} = OUTPERM.
```

pairwiseqa.m

```
function [outperm, rawindex, allperms, index] = ...
   pairwiseqa(prox, targ, inperm)

% PAIRWISEQA carries out an iterative
% Quadratic Assignment maximization task using the
% pairwise interchanges of objects in the
% permutation defining the row and column
% order of the data matrix.
%
% syntax: [outperm, rawindex, allperms, index] = ...
%   pairwiseqa(prox, targ, inperm)
%
% INPERM is the input beginning permutation
% (a permutation of the first $n$ integers).
% PROX is the $n \times n$ input proximity matrix.
% TARG is the $n \times n$ input target matrix.
% OUTPERM is the final permutation of
% PROX with the cross-product index RAWINDEX
% with respect to TARG.
% ALLPERMS is a cell array containing INDEX entries corresponding
% to all the permutations identified in the optimization from
% ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.
```

partitionfit.m

```
function [fitted,vaf,weights,end_condition] =
partitionfit(prox,member)

% PARTITIONFIT provides a least-squares approximation to a proximity
% matrix based on a given collection of partitions.
%
```

```
% syntax: [fitted,vaf,weights,end_condition] = partitionfit(prox,member)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n matrix
% indicating cluster membership, where each row corresponds to a specific
% partition (there are m partitions in general); the columns of MEMBER
% are in the same input order used for PROX.
% FITTED is an n x n matrix fitted to PROX (through least-squares)
% constructed from the nonnegative weights given in the m x 1 WEIGHTS
% vector corresponding to each of the partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

partitionfnd_averages.m

```
function [membership,objectives] = partitionfnd_averages(prox)

% PARTITIONFND_AVERAGES uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
% syntax: [membership,objectives] = partitionfnd_averages(prox)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX.
% OBJECTIVES is the vector of merit values minimized in the
% construction of the ordered partitions, each defined by the
% maximum over clusters of the average proximities within subsets.
```

partitionfnd_diameters.m

```
function [membership,objectives] = partitionfnd_diameters(prox)

% PARTITIONFND_DIAMETERS uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
```

```
% syntax: [membership,objectives] = partitionfnd_diameters(prox)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX.
% OBJECTIVES is the vector of merit values minimized in the
% construction of the ordered partitions, each defined by the
% maximum over clusters of the maximum proximities within subsets.
```

partitionfnd_kmeans.m

```
function [membership,objectives] = partitionfnd_kmeans(prox)

% PARTITIONFND_KMEANS uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
% syntax: [membership,objectives] = partitionfnd_kmeans(prox)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX.
% OBJECTIVES is the vector of merit values minimized in the
% construction of the ordered partitions, each defined by the
% sum over clusters of the average (using a division by twice the
% number of objects in the class) of the proximities within subsets.
```

proxmon.m

```
function [monproxpermut, vaf, diff] = proxmon(proxpermut, fitted)

%  PROXMON produces a monotonically transformed proximity matrix
%  (MONPROXPERMUT) from the order constraints obtained from each
%  pair of entries in the input proximity matrix PROXPERMUT
%  (symmetric with a zero main diagonal and a dissimilarity
%  interpretation).
%
```

```
% syntax: [monproxpermut, vaf, diff] = proxmon(proxpermut, fitted)
%
% MONPROXPERMUT is close to the
% $n \times n$ matrix FITTED in the least-squares sense;
% the variance accounted for (VAF) is how
% much variance in MONPROXPERMUT can be accounted for by
% FITTED; DIFF is the value of the least-squares criterion.
```

proxmontm.m

```
function [monproxpermuttm, vaf, diff] = ...
    proxmontm(proxpermuttm, fittedtm)

% PROXMONTM produces a monotonically transformed
% two-mode proximity matrix (MONPROXPERMUTTM)
% from the order constraints obtained
% from each pair of entries in the input two-mode
% proximity matrix PROXPERMUTTM (with a dissimilarity
% interpretation).
%
% syntax: [monproxpermuttm, vaf, diff] = ...
%      proxmontm(proxpermuttm, fittedtm)
%
% MONPROXPERMUTTM is close to the $nrow \times ncol$
% matrix FITTEDTM in the least-squares sense;
% The variance accounted for (VAF) is how much variance
% in MONPROXPERMUTTM can be accounted for by FITTEDTM;
% DIFF is the value of the least-squares criterion.
```

sqeuclidean.m

```
code only; no help file

 function [sqeuclid] = sqeuclidean(data)
```

targlin.m

```
function [targlinear] = targlin(n)

% TARGLIN produces a symmetric proximity matrix of size
% $n \times n$, containing distances
% between equally and unit-spaced positions
% along a line: targlinear(i,j) = abs(i-j).
```

```
%
%  syntax: [targlinear] = targlin(n)
```

ultracomptm.m

```
function [ultracomp] = ultracomptm(ultraproxtm)

% ULTRACOMPTM provides a completion of a given two-mode ultrametric
% matrix to a symmetric proximity matrix satisfying the
% usual ultrametric constraints.
%
% syntax: [ultracomp] = ultracomptm(ultraproxtm)
%
% ULTRAPROXTM is the $nrow \times ncol$ two-mode ultrametric matrix;
% ULTRACOMP is the completed symmetric
% $n \times n$ proximity matrix having the usual
% ultrametric pattern for $n = nrow + ncol$.
```

ultrafit.m

```
function [fit,vaf] = ultrafit(prox,targ)

% ULTRAFIT fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit,vaf] = ultrafit(prox,targ)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% TARG is an ultrametric matrix of the same size as PROX;
% FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX satisfying the ultrametric
% constraints implicit in TARG.
```

ultrafit_missing.m

```
function [fit,vaf] = ultrafit_missing(prox,targ,proxmiss)

% ULTRAFIT_MISSING fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit,vaf] = ultrafit_missing(prox,targ,proxmiss)
%
```

```
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); also, missing entries in the input
% proximity matrix PROX are given values of zero.
% TARG is an ultrametric matrix of the same size as PROX;
% FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX satisfying the ultrametric
% constraints implicit in TARG. PROXMISS is the same size as PROX (with main
% diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
```

ultrafittm.m

```
function [fit,vaf] = ultrafittm(proxtm,targ)

% ULTRAFITTM fits a given (two-mode) ultrametric using iterative
% projection to a two-mode (rectangular) proximity matrix in the
% $L_{2}$-norm.
%
% syntax: [fit,vaf] = ultrafittm(proxtm,targ)
%
% PROXTM is the input proximity matrix (with a dissimilarity
% interpretation); TARG is an ultrametric matrix of the same size
% as PROXTM; FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROXTM satisfying the
% ultrametric constraints implicit in TARG.
```

ultrafnd.m

```
function [find,vaf] = ultrafnd(prox,inperm)

% ULTRAFND finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf] = ultrafnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints.
```

ultrafnd_confit.m

```
function [find,vaf,vafarob,arobprox,vafultra] = ...
    ultrafnd_confit(prox,inperm,conperm)

% ULTRAFND_CONFIT finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm,
% constrained by a given object order.
%
% syntax: [find,vaf,vafarob,arobprox,vafultra] = ...
%    ultrafnd_confit(prox,inperm,conperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered in obtaining the ultrametric;
% CONPERM is the given constraining object order;
% VAFAROB is the VAF of the anti-Robinson matrix fit, AROBPROX, to PROX;
% VAFULTRA is the VAF of the ultrametric fit to AROBPROX;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints, and given
% in CONPERM order.
```

ultrafnd_confnd.m

```
function [find,vaf,conperm,vafarob,arobprox,vafultra] = ...
    ultrafnd_confnd(prox,inperm)

% ULTRAFND_CONFND finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm, and
% also locates a initial constraining object order.
%
% syntax: [find,vaf,conperm,vafarob,arobprox,vafultra] = ...
%    ultrafnd_confnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered in obtaining the ultrametric;
% CONPERM is the identified constraining object order;
% VAFAROB is the VAF of the anti-Robinson matrix fit, AROBPROX, to PROX;
% VAFULTRA is the VAF of the ultrametric fit to AROBPROX;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints, and given
% in CONPERM order.
```

ultrafnd_missing.m

```
function [find,vaf] = ultrafnd_missing(prox,inperm,proxmiss)

% ULTRAFND_MISSING finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf] = ultrafnd_missing(prox,inperm,proxmiss)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); also, missing entries in the input
% proximity matrix PROX are given values of zero.
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints. PROXMISS is
% the same size as PROX (with main
% diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
```

ultrafndtm.m

```
function [find,vaf] = ultrafndtm(proxtm,inpermrow,inpermcol)

% ULTRAFNDTM finds and fits a two-mode ultrametric using
% iterative projection heuristically on a rectangular proximity
% matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf] = ultrafndtm(proxtm,inpermrow,inpermcol)
%
% PROXTM is the input proximity matrix (with a
% dissimilarity interpretation);
% INPERMROW and INPERMCOL are permutations for the row and column
% objects that determine the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROXTM satisfying the ultrametric constraints.
```

ultraorder.m

```
function [orderprox,orderperm] = ultraorder(prox)
```

```
% ULTRAORDER finds for the input proximity matrix PROX
% (assumed to be ultrametric with a zero main diagonal)
% a permutation ORDERPERM that displays the anti-
% Robinson form in the reordered proximity matrix
% ORDERPROX; thus, prox(orderperm,orderperm) = orderprox.
%
% syntax:  [orderprox,orderperm] = ultraorder(prox)
```

ultraplot.m

```
function [] = ultraplot(ultra)

% ULTRAPLOT gives a dendrogram plot for the input ultrametric
% dissimilarity matrix ULTRA.
%
% syntax: [] = ultraplot(ultra)
```