UWEO StatR201 Lecture 11

Clustering; Parallel and Efficient Computing

Assaf Oron, March 2013

Tonight's Menu

- Tips for final project

"Unsupervised Learning"

- Conceptual Overview
- Hierarchical clustering
- k-Means clustering
- "Model-Based" clustering

Then: **Computing Performance Enhancement**

- Parallel processing and manipulation using `foreach` and `%dopar%`
- Fast data manipulation with `data.table`

Then: **(temporary?) Parting Words:** summary of class and a look towards Spring, with some last-minute R tricks mixed in.

But first, before we get tired, one long-overdue R trick/annoyance.

Date and Time in R: `as.Date` and `as.POSIXlt`

This might have been shown before, but worth (re?)visiting before you graduate 2/3 of the program. *What to do about dates?*

```
trux = read.csv("../Datasets/summer_trucks.csv", as.is = TRUE)
head(trux, 3)
```

```
##        id      date stagnant Caline1.5 caline3
## 1 1313326 2012-8-18       52    0.4124  0.4736
## 2 1313326  2012-9-1       42    0.5132  0.5700
## 3 1313326 2012-9-15      113    0.5640  0.6386
```

```
summary(trux$date)
```

```
##    Length     Class      Mode
##       513 character character
```

`trux$date` is a string vector right now (if not for `as.is=TRUE` it would have been a `factor`). Can R speak date? Naturally.

```
trux$date = as.Date(trux$date)
summary(trux$date)
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu.
## "2012-08-18" "2012-09-15" "2012-10-13" "2012-10-13" "2012-11-10"
##         Max.
## "2012-12-08"
```

Note the capital D in `as.Date`. This class allows addition/substraction (with the unit being days) and other operations. For some functions (e.g., `smooth.spline`) you will need to convert `Date` objects to integers via `as.numeric`.

The date format in `trux` is the R default. Hence, we can actually `read.csv` the data directly as `Date`:

```
trux = read.csv("../Datasets/summer_trucks.csv", colClasses = c("character",
    "Date", rep("numeric", 3)))
summary(trux$date)
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu.
## "2012-08-18" "2012-09-15" "2012-10-13" "2012-10-13" "2012-11-10"
##         Max.
## "2012-12-08"
```

Converting other Date Strings

The format used by R is the ISO 8601 international date standard; its `origin` (Day 1) is `1970-01-01`. Note (as in the example) that R does **not** auto-interpret any string that might look like a `Date` as such; rather, it needs to be specifically told so via `as.Date()`.

Now… what about importing dates from US-style or other formats?

```
morenox = read.csv("../Datasets/LRnox.csv", as.is = TRUE)
head(morenox, 3)
```

```
##   event   on_date air_nox siteid
## 1    92  7/1/2009    13.6  LR001
## 2    91 6/17/2009    13.6  LR001
## 3    90  6/3/2009    13.6  LR001
```

```
morenox$on_date = as.Date(morenox$on_date, "%m/%d/%Y")
head(morenox, 3)
```

```
##   event    on_date air_nox siteid
## 1    92 2009-07-01    13.6  LR001
## 2    91 2009-06-17    13.6  LR001
## 3    90 2009-06-03    13.6  LR001
```

```
summary(morenox$on_date)
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu.
## "2005-11-23" "2006-09-27" "2007-09-05" "2007-09-11" "2008-08-16"
##         Max.
## "2009-07-01"
```

A more detailed standard time class is `as.POSIXlt`, which includes time as well as several useful utilities (e.g., `wday`). `Date` objects can be converted to and from that class.

Final Project Tips, Q&A

Here are my nanny-style tips:

- **The basics still apply,** regardless of problem type, method and level of sophistication.
- Use plenty of descriptives, even automated ones (with large p). You don't have to present all of them - but for yourself.
- Regarding outliers/improbables: **if using a repository dataset, do NOT exclude observations without consulting me.** If using your own dataset, still exercise caution and good judgment.
- Do not forget transformations/alignment/scaling
- **Filtering (for information content, correlation, etc.) pays back in multiple ways:** improving performance, improving clarity, saving CPU time
- Account for the dataset's "natural" grouping, if there is one.

When choosing/tuning a method, think about:

- Would weighting the features matter for some of your methods? If so, what weighting would be sensible?
- Do you have a basic grasp of what makes the method tick? If not sure, test it out by some deliberately wild tuning.
- Do you need to pre-treat categorical covariates to make them work?
- Do some back-and-forth on the training data from tuning back to descriptives, diagnosing misclassified/outlying observations and seeing whether you have missed something
- **Don't be afraid to ask questions.** When writing up, remember it is 20 pages max.

“Supervised” vs. “Unsupervised” “Learning”

Personally, I think the importance of these terms is exaggerated, but it is very standard. Here it is in a nutshell.

Classification and regression are both called Supervised Learning. Symbolically, continuous-y regression can be written as

$$E[y] = f(\mathbf{X}),$$

whereas classification (or, equivalently, categorical-outcome regression) is

$$\Pr(y = c) = f_c(\mathbf{X}), \;\; c = 1, \ldots, C.$$

On the other hand,

With **“Unsupervised Learning”,** we have this:

$$\mathbf{X}$$

What Can One Do?

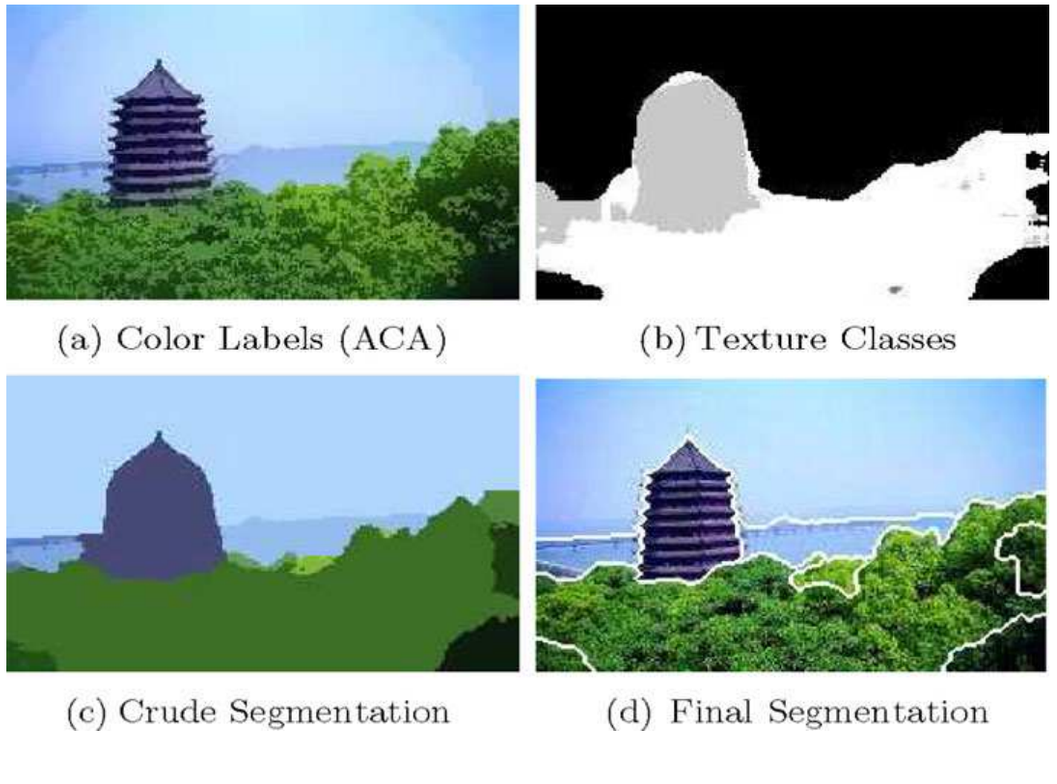
With

$$\mathbf{X}?$$

Why, of course:

- Descriptives
- Nicer Descriptives
- Even Nicer Descriptives
- **Something Really Grand (let's call it SRG).**
- One SRG we've already learned (sort-of), is **SVD/principal-components.**
- "invent" a y, usually a categorical one, and assign it to the observations. This is known as **Clustering.**
- *Another famous SRG is "picking winners" in a high-dimensional dataset: in other words, inventing a continuous y and finding its optimum. That's what Google's PageRank algorithm does (Section 14.10), but it involves material beyond our current reach.*

Examples for Clustering Applications



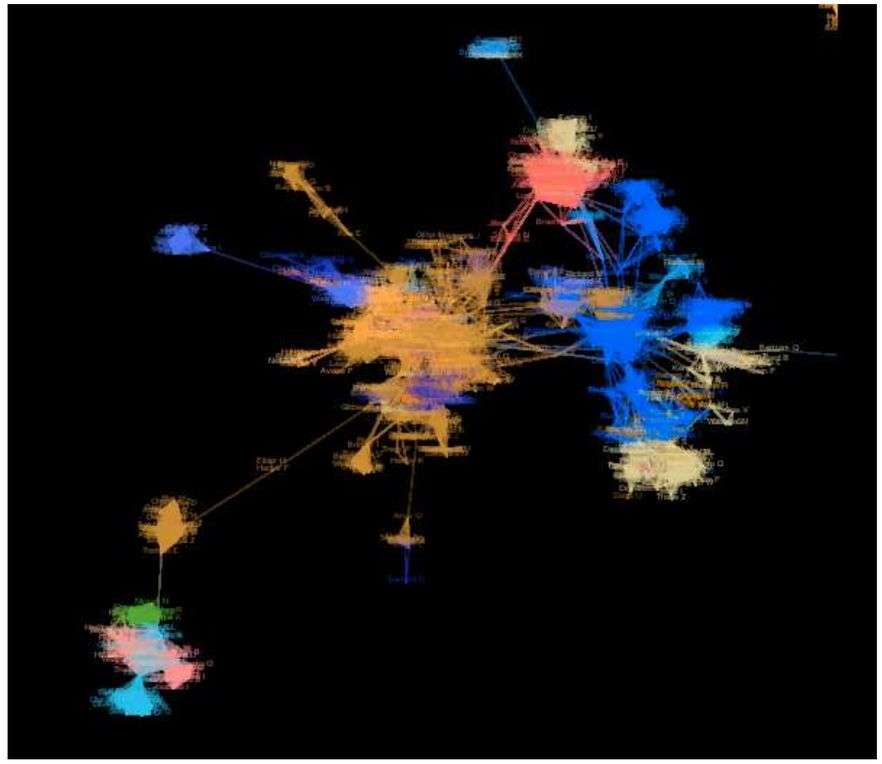
(a) Color Labels (ACA)

(b) Texture Classes

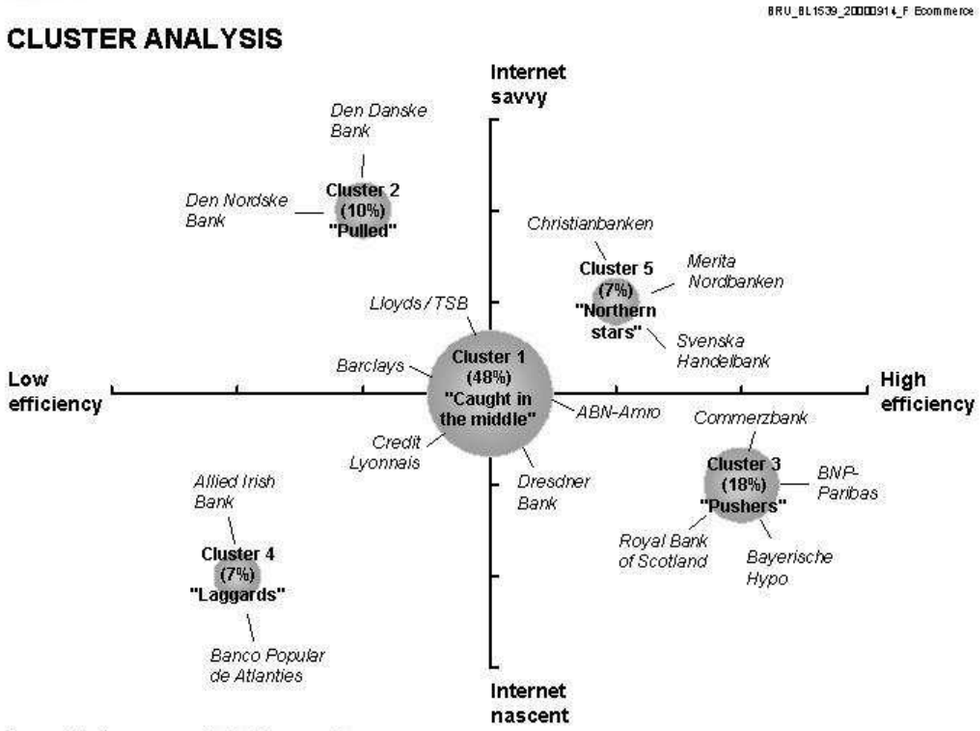
(c) Crude Segmentation

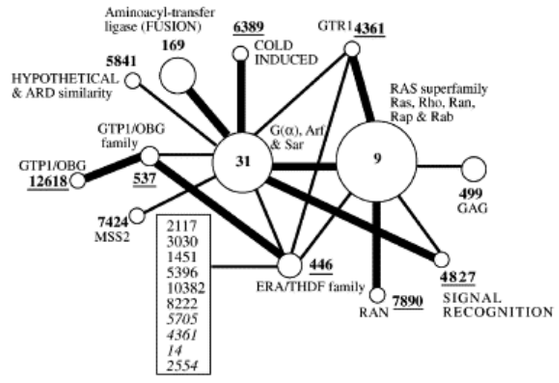
(d) Final Segmentation

Image segmentation



Social networks





Molecular biology *("-Omics")*

BS Economics

Clustering Basics

Most clustering algorithms are based on D, a pairwise **dissimilarity** or *"distance"* matrix between the observations, calculated from their features.

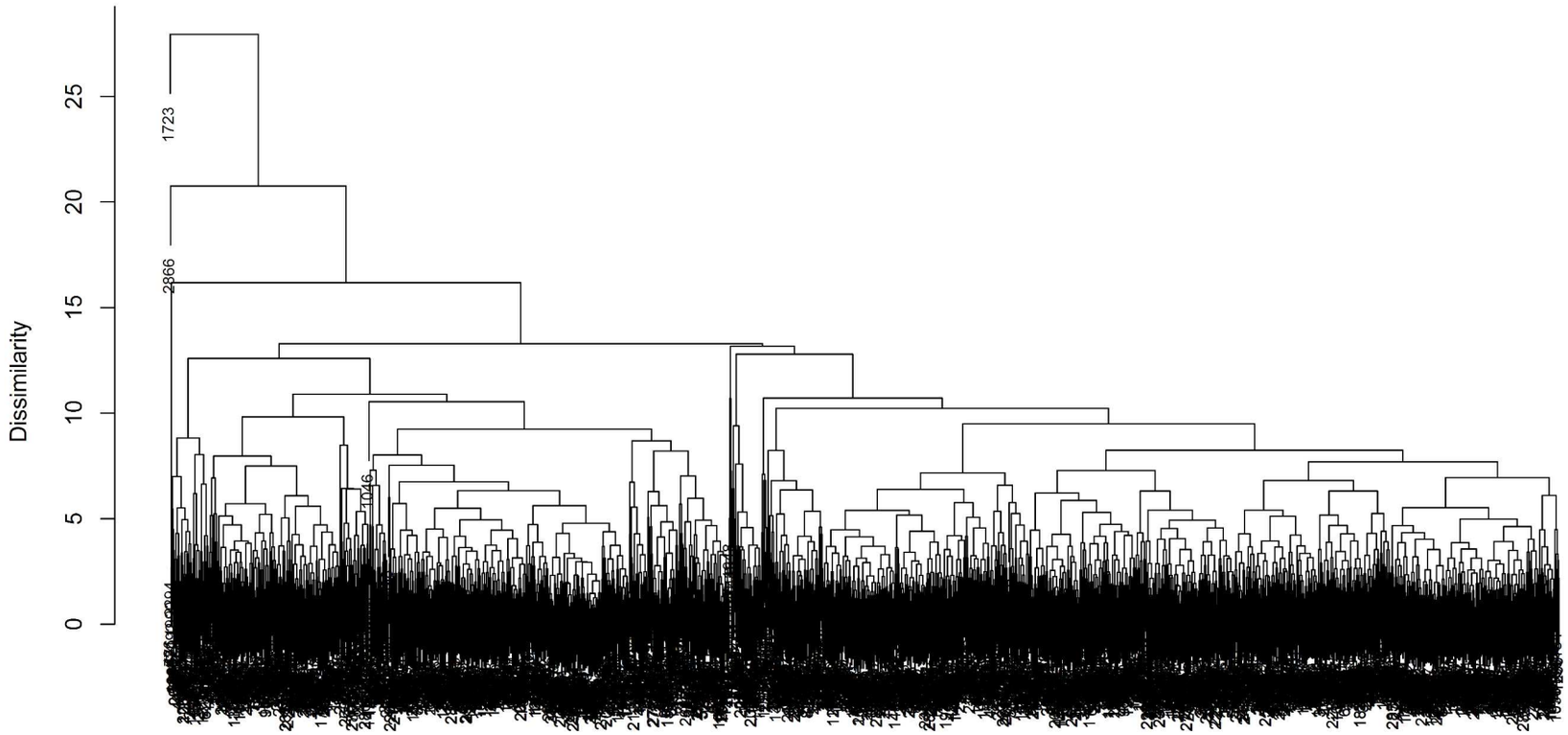
D can be a genuine distance matrix, or any measure deemed appropriate for the context (as usual, categorical variables make this harder).

Generally speaking, clusters are groups of **relatively** less dissimilar points. However, the completely open challenge of *"tell me how many clusters there are, and which points go into which cluster"* is usually too ill-defined to answer without imposing further constraints. Ways out of this morass include:

- Specify how many clusters there are, then let the algorithm define them, as in **k-Means Clustering;**
- Specify a nested hierarchy of probability (density) models for the general cluster structure, then compare the various maximum-likelihood solutions via nested hypotheses or AIC/BIC, as in **Model-Based Clustering;**
- Describe the entire cascade of grouping (ungrouping) into (out of) clusters, from a single point to all n (or vice versa) - and let the user decide where to make the cut, as in **Hierarchical Clustering,** which is where we start.

Hierarchical Clustering in R

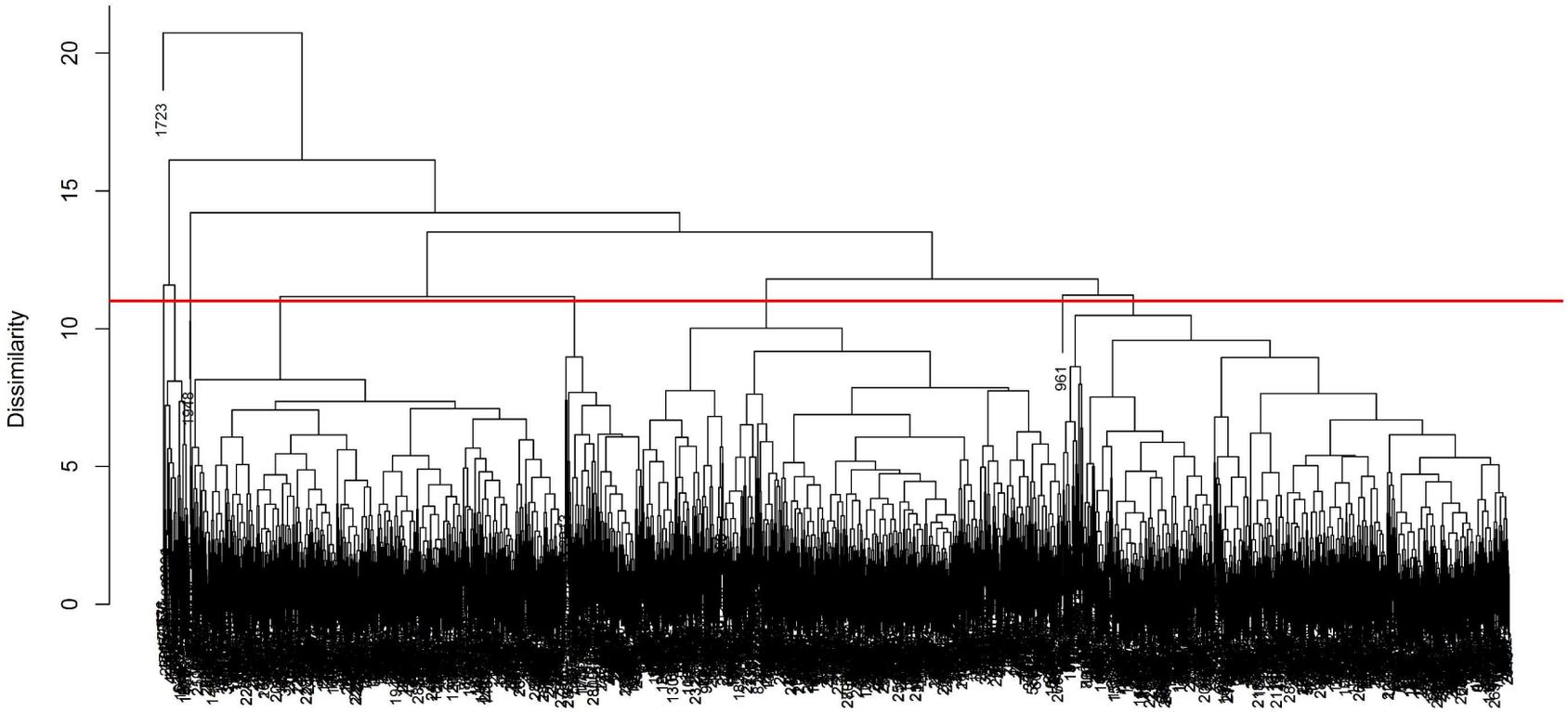
```
par(mar = c(2, 5, 1, 1))
plot(hclust(dist(scale(wine[, -12]))), cex = 0.7, xlab = "", ylab = "Dissimilarity",
    main = "")
```



Hierarchical clustering modestly aims to serve mostly as a powerful descriptive. Here, for example, the asymmetry and gross outliers remind us that perhaps `residual.sugar` and `free.sulfur.dioxide` should be log-transformed (the two singletons correspond to the highest values in each feature).

Hierarchical Clustering in R

```
par(mar = c(2, 5, 1, 1))
wine$logsugar = log10(wine$residual.sugar)
wine$logfree = log10(wine$free.sulfur.dioxide)
winedist = dist(scale(wine[, -c(4, 6, 12)]))
plot(hclust(winedist), cex = 0.7, xlab = "", ylab = "Dissimilarity", main = "")
abline(h = 11, lwd = 2, col = 2)
```



The sugar high is still lonely, but there is more structure visible overall, and the tree is less skewed. I am **Not** suggesting to force the tree into symmetric form no matter what, but with Euclidean distance (the default of `dist`) the skewness of a couple of features might reflect itself in the tree. **So how many clusters?** depends on where we draw the line… at a distance of 11 (=1 SD per feature, marked by red line) there are about 10 clusters, including at least a couple of singetons.

How `hclust` Works

`hclust` is an **Agglomerating Clustering Algorithm:**

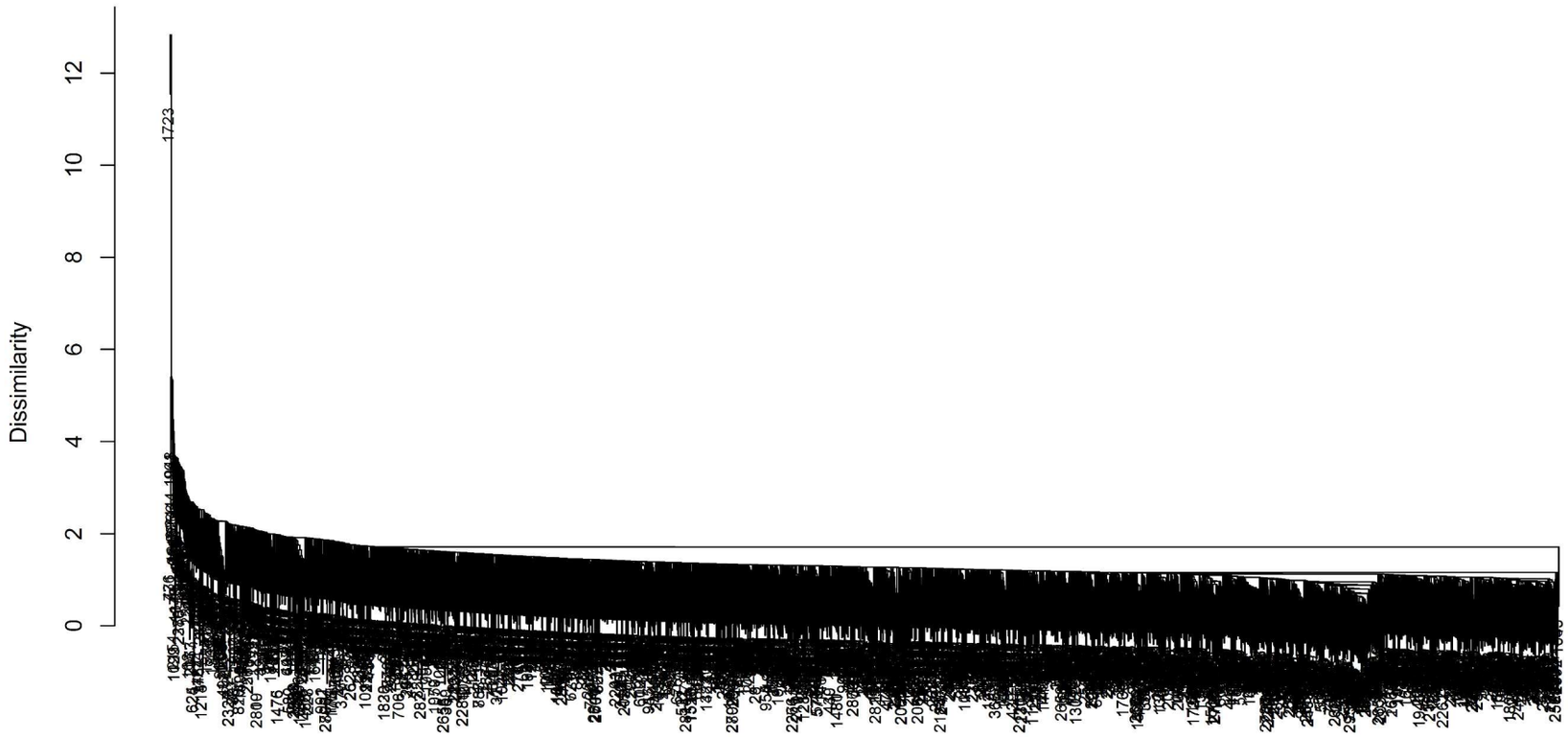
0. Calculate (or receive as input) all pairwise D values, and define a **cluster-dissimilarity metric** $\tilde{d}$ that is a function of D.
1. Start with each observation in its own cluster.
2. At each step, **join together the two least-dissimilar clusters,** according to the between-cluster metric defined in Step 0.
3. Record the recently-joined pair's $\tilde{d}$ value.
4. Continue until all observations have been joined to a single cluster (this will take exactly $n - 1$ steps).
5. Draw the tree according to the join hierarchy, with heights corresponding to each joining's $\tilde{d}$ value.

The definition of $\tilde{d}$ has a great impact upon the resulting tree. Common options:

- **Complete Linkage** (the `hclust` default): the maximum D between the two clusters' individual points;
- **Single Linkage**: the minimum D between the two clusters' individual points; -**Average Linkage**: as its name indicates, the average D between the two clusters' individual points;
- `hclust` allows for 4 more options (see its help page).

`hclust` with Single Linkage

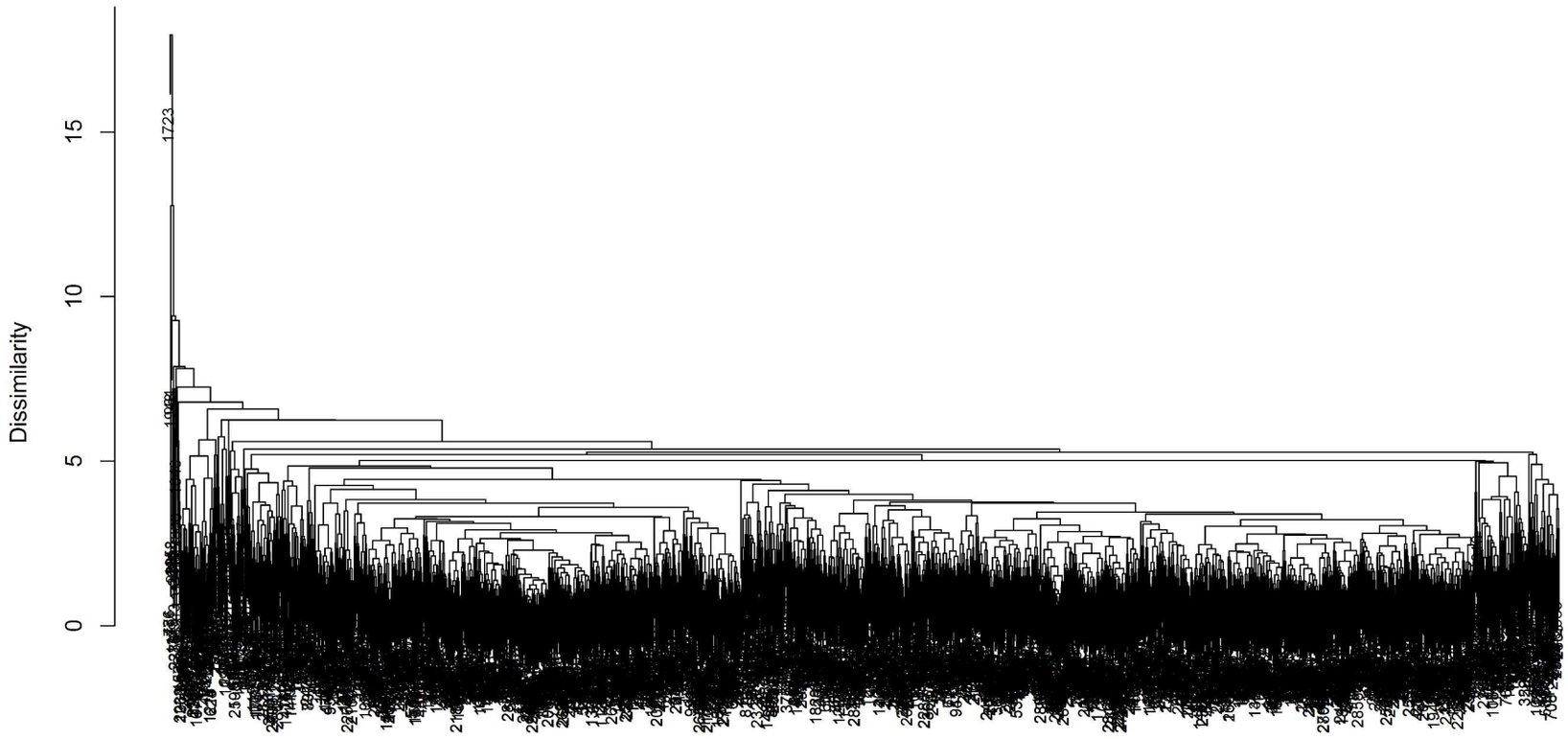
```
par(mar = c(2, 5, 1, 1))
wine$logsugar = log10(wine$residual.sugar)
wine$logfree = log10(wine$free.sulfur.dioxide)
plot(hclust(winedist, method = "single"), cex = 0.7, xlab = "", ylab = "Dissimilarity",
    main = "")
```



Single-linkage downplays dissimilarities; generally not recommended unless you have good content reason.

`hclust` with Average Linkage

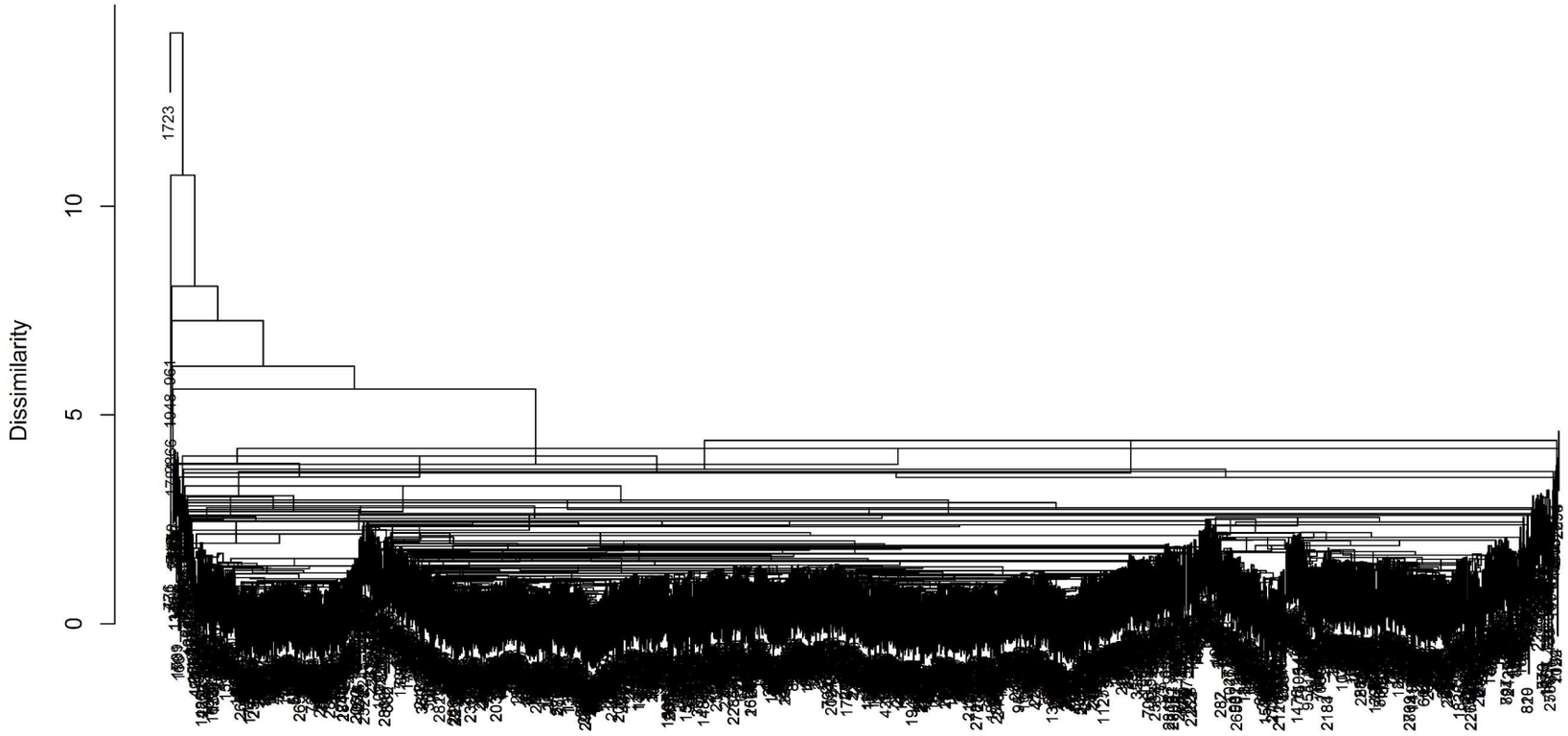
```
par(mar = c(2, 5, 1, 1))
wine$logsugar = log10(wine$residual.sugar)
wine$logfree = log10(wine$free.sulfur.dioxide)
plot(hclust(winedist, method = "average"), cex = 0.7, xlab = "", ylab = "Dissimilarity",
    main = "")
```



If we’re going for average, why not the median? It will be more robust to outliers…

`hclust` with Median Linkage

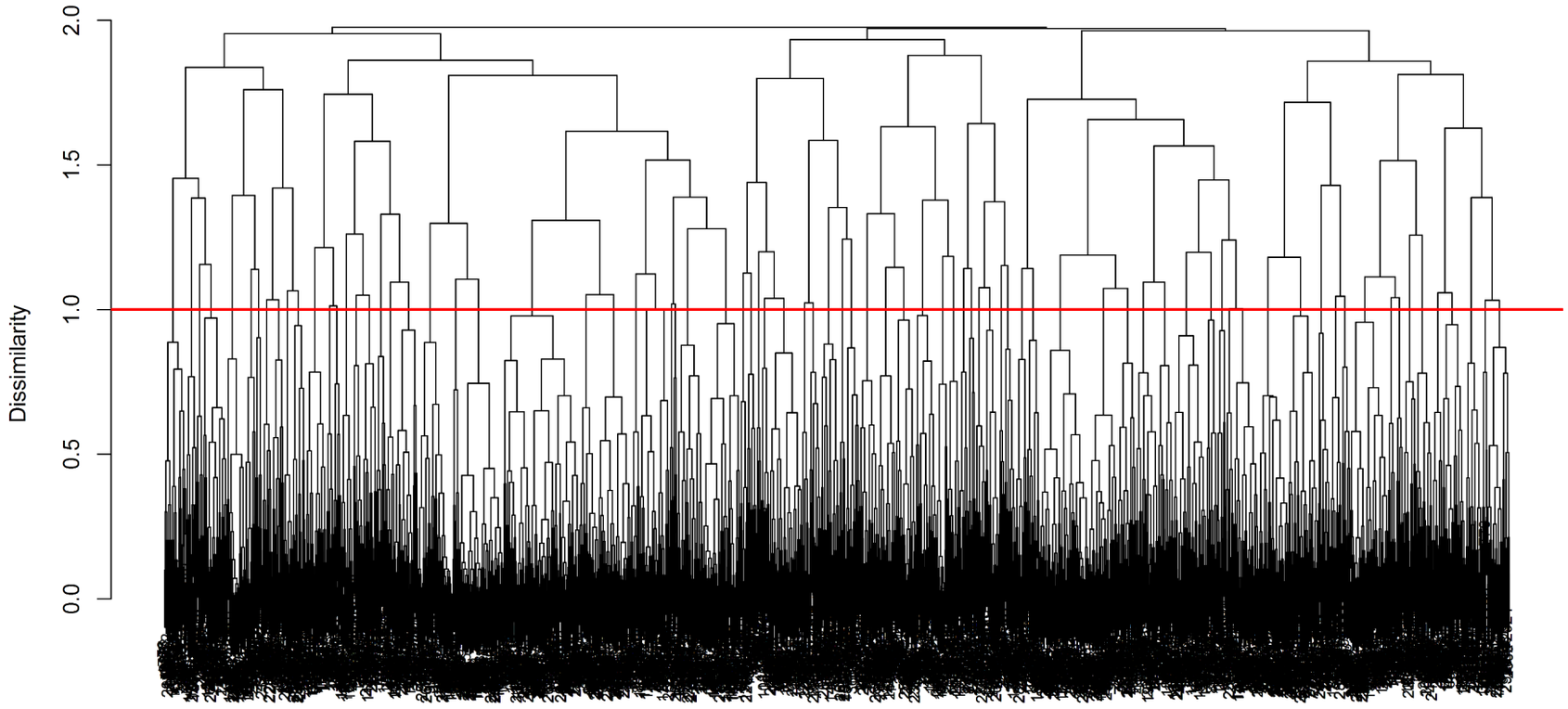
```
par(mar = c(2, 5, 1, 1))
wine$logsugar = log10(wine$residual.sugar)
wine$logfree = log10(wine$free.sulfur.dioxide)
plot(hclust(winedist, method = "median"), cex = 0.7, xlab = "", ylab = "Dissimilarity",
    main = "")
```



Oopsie!… What's going on? Well, some methods (single, complete, average) guarantee **monotonicity** - i.e., that higher joinings always take place at a larger value of $\tilde{d}$. The median method apparently does **not.**

`hclust` with Correlation Distance

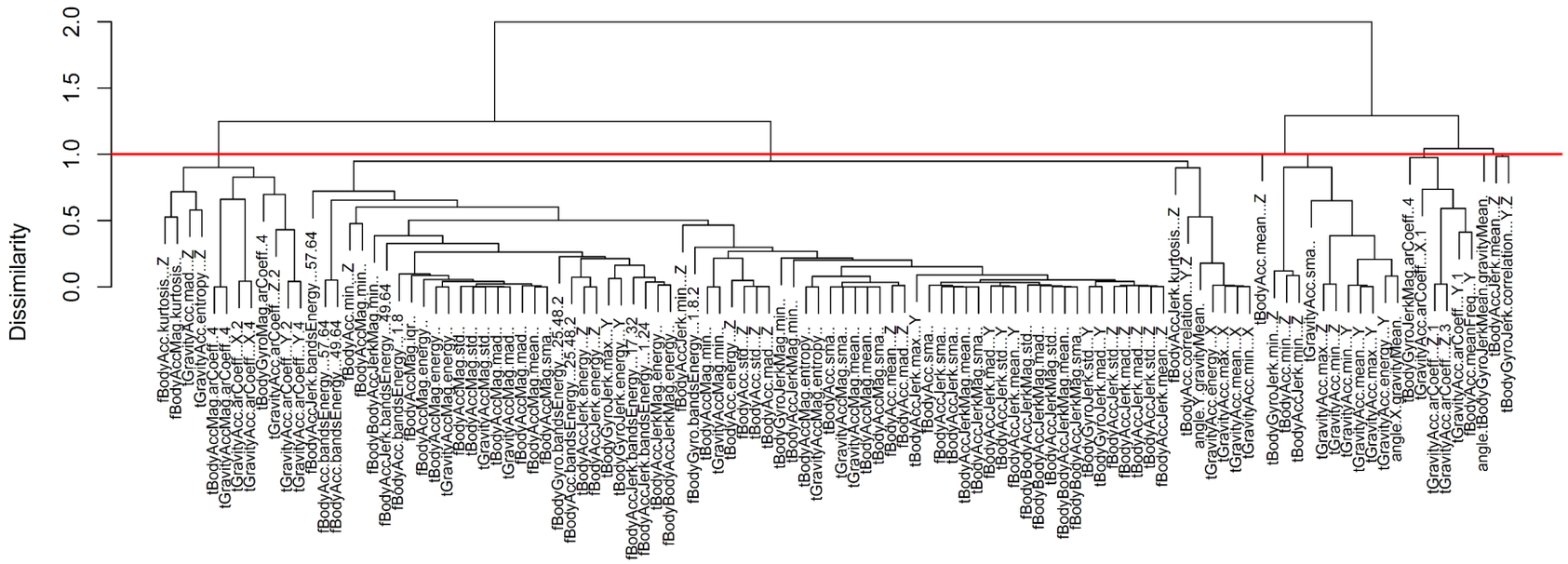
```
par(mar = c(2, 5, 1, 1))
wineRdist = as.dist(1 - cor(t(scale(wine[, -c(4, 6, 12)]))))  # note the 't'
plot(hclust(wineRdist), cex = 0.7, xlab = "", ylab = "Dissimilarity", main = "",
    sub = "")
abline(h = 1, lwd = 2, col = 2)
```



Generally, correlation-based dissimilarity is more robust than Euclidean distance. This tree suggests almost no structure: note the high join points (there are > 10 clusters), the **very** symmetric look, and the scarcity of any clusters with strongly positive pairwise correlation (the red line marks $r = 0$).

Clustering the Features

```
plot(hclust(smartRdist), cex = 0.7, xlab = "", ylab = "Dissimilarity", main = "",
    sub = "")
abline(h = 1, lwd = 2, col = 2)
```



More often than not, `hclust` is more helpful on the **features** than on the observations. Here, with the (between/within filtered) smartphone dataset, we see quite a bit of structure. Clearly, the division into the walking-related and non-walking-related features, as well as quite a few *very* highly correlated pairs (suggesting, for those working on HW6, that correlation-based filtering is probably needed here). But also: some additional structure that is possibly worth investigating. Questions? *Online* questions? Practice this a bit, using the `iris` dataset (hint: set `labels=iris$Species`)

k-Means Clustering

You would *think* that k-Means clustering might be a variant of k-Nearest-Neighbors classification. Ha. k is just the statistician's kneejerk response to the request, *"Pick a letter at random…"* (my first dissertation paper title: "The k-in-a-row Up-and-Down design, Revisited" - and that k has nothing to do with either of the two :)

Seriously: k-Means is more closely related to that classification method this class made history by skipping: LDA. Even more closely, as the book says, to **Gaussian mixture modeling** (=density estimation in p dimensions).

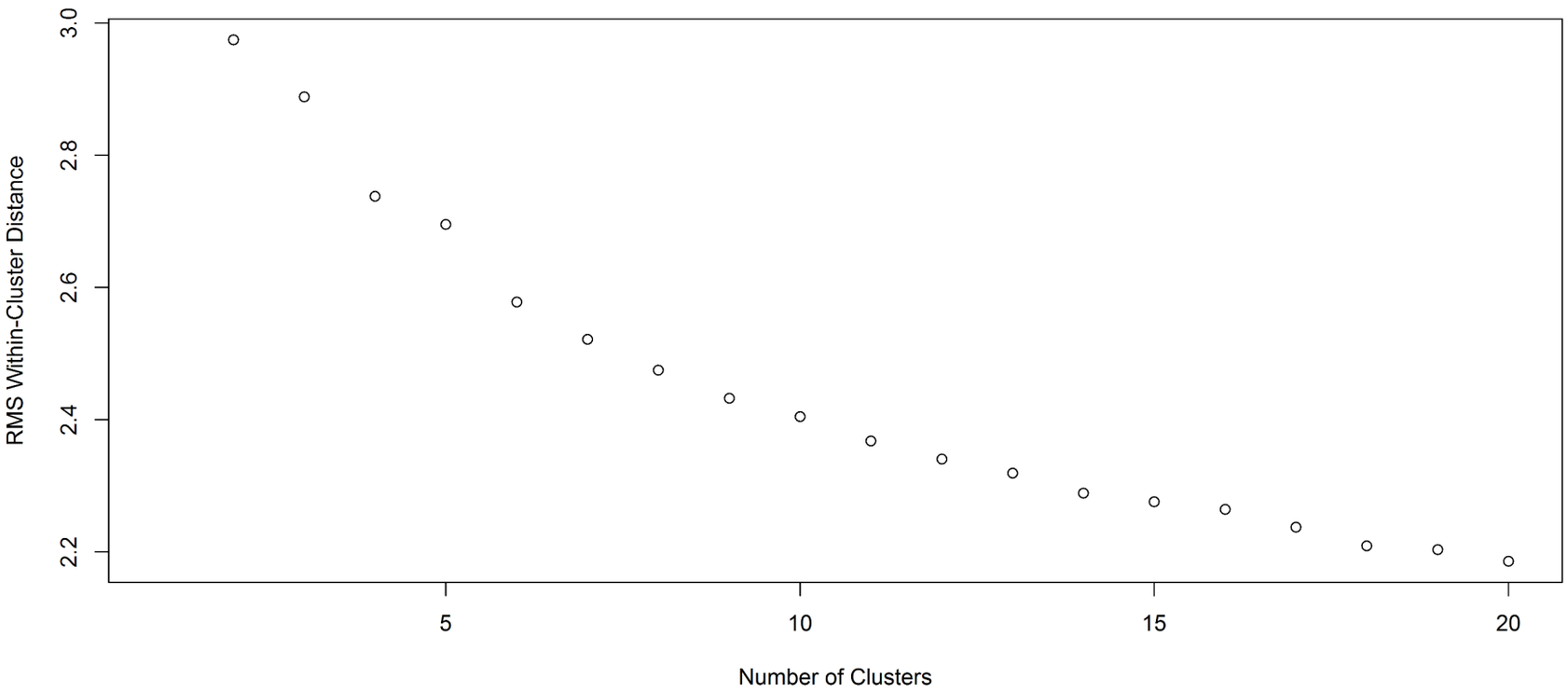
k-Means tries to define k clusters (with k given) that optimize the between/within sum-of-squares ratio. As the book explains, this deceptively simple task is actually a combinatorial explosion. So instead, the algorithm iterates between

- Finding the centers-of-mass of the k clusters (=the "Means")
- Re-assigning points to the closest "Mean"

This algorithm is actually lightning-quick.

k-Means Clustering

```
kwine = list()
for (a in 2:20) kwine[[a]] = kmeans(scale(wine[, -c(4, 6, 12)]), a, iter.max = 100)
plot(1:20, sapply(kwine, function(x) sqrt(x$tot.withinss/(length(x$cluster) -
    length(x$size)))), xlab = "Number of Clusters", ylab = "RMS Within-Cluster Distance")
```



Ok, let's give k-Means a road test. Do the same drill above, for `iris` (don't need to go all the way to $k = 20$; 6-7 should suffice). Also possibly for the **features** of `smartscale`.

Questions? *Online* questions?

Model-Based Clustering with `mclust`

The RMS-within vs. k plot almost begs the question “where to draw the line?”. By now you might have guessed that the within-distance is pretty much guaranteed to go down as we add clusters.

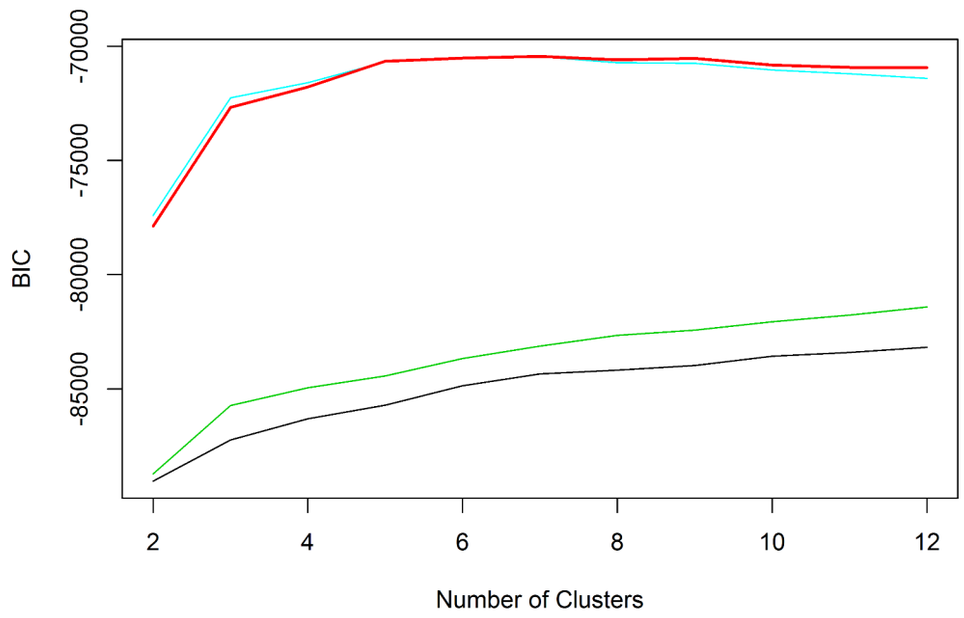
You might have also realized, that (hyper-)spherical clusters in feature space are analogous to a very specific **probability model: Normal.** In fact, independent, equal-variance Normal.

Adrian Raftery and many others right here at the UW stats department, decided to combine these two nuggets and turn the cluster-finding exercise in Euclidean space, into nested-hypothesis model selection.

```
library(mclust)
mout = Mclust(scale(wine[, -c(4, 6, 12)]), G = 2:12)
c(mout$G, mout$modelName, mout$df)
```

```
## [1] "7"   "VEV" "485"
```

```
plot(2:12,mout$BIC[,"EII"],ylim=range(mout$BIC),type='l',
     ylab="BIC",xlab="Number of Clusters")
lines(2:12,mout$BIC[,'VII'],col=3)
lines(2:12,mout$BIC[,'VVV'],col=5)
lines(2:12,mout$BIC[,'VEV'],col=2,lwd=2)
```



Model-Based Clustering, Mixture Models and LDA

Model-based clustering assumes the features follow a **Gaussian mixture distribution:**

$$\begin{aligned}\Pr(\mathbf{X}) &= \sum_k \Pr(\mathbf{X} \in \text{Cluster } k) \Pr(\mathbf{X} \mid \mathbf{X} \in k) \\ &= \sum_k \pi_k I(\mathbf{X} \in k) \operatorname{Normal}\big(\mu_k, \Sigma_k\big),\end{aligned}$$

where $I(\cdot)$ is the indicator function, and μ_k, Σ_k are the mean and covariance matrix of cluster k in p-dimensional feature space.

Linear Discriminant Analysis (LDA) and its direct extensions such as Quadratic Discriminant Analysis, make essentially the same assumption. However, with LDA/QDA, both the number of classes and their (training-set) assignments are fully known.

LDA/QDA are not good classifiers, because they constrain class boundaries to be *very* smooth. OTOH, when it comes to cluster discovery, the exact boundaries might concern us less than identifying high-density centers of mass.

Model-Based Clustering: Specification

A multivariate Gaussian can be more or less constrained, for example with respect to **each single cluster**:

- Equal variance and i.i.d. features? **‘Spherical’**
- Unequal variance, but i.i.d. features? **‘Diagonal’**
- Unequal variance and correlated features? **‘Ellipsoidal’**

And also, **between-cluster constraints:**

- Are they all the same size?
- Are they all the same shape?
- Are they all the same orientation?

Some constraints are associated with a whole bunch of degrees of freedom (e.g., going from “Diagonal” to “Ellipsoidal”, or allowing each cluster to have its own shape or orientation).

Since `Mclust` criterion is BIC, the fit has to be *much* better to justify these moves. Oftentimes, it is easier for the model to retain the constraints, and just add a couple more clusters.

If we have time, we will explore ellipsoids interactively with `rgl`.

Questions? *Online* questions?

`foreach` and %do%: Yet Another Flavor of `Xapply`?

We will now learn the basics of parallel processing, using the packages provided by Revolution Analytics. This private company attempts to establish R community "street cred" by providing performance-enhancing packages.

As infrastructure for parallel processing, Revolution developed `foreach`, with the namesake command paired with `%do`.

```
library(foreach)
library(randomForest)
system.time(rf0 <- foreach(ntree = rep(500, 4), .combine = combine) %do% randomForest(factor(quality) ~
    ., data = wine, ntree = ntree))
```

```
##    user  system elapsed
##   15.55    0.58   16.13
```

`randomForest` has a built-in `combine` functionality, that allows to break up a single "Forest" into pieces and put it together again for the final calculations.

At first glance, `foreach ... %do%` looks like just another way to do loop. However,

1. You can have multiple arguments inside `foreach`, including lists etc. The %do% will cycle over all of them element-wise, just like `mapply`.
2. Replace %do% with `%dopar%`, add some syntax… and Voila!

Processing with `foreach` and %dopar%

```
library(doSNOW); library(randomForest); options(width=80)
cl <- makeCluster(3, type = "SOCK")
registerDoSNOW(cl)
system.time(rf1 <-
    foreach(ntree=rep(667,3), .combine=combine,.packages='randomForest')
            %dopar% randomForest(factor(quality)~.,data=wine,ntree=ntree))
```

```
##    user  system elapsed
##    5.25    0.49   10.20
```

```
stopCluster(cl)
cl <- makeCluster(6, type = "SOCK")
registerDoSNOW(cl)
system.time(rf2 <-
    foreach(ntree=rep(333,6), .combine=combine,.packages='randomForest')
            %dopar% randomForest(factor(quality)~.,data=wine,ntree=ntree))
```

```
##    user  system elapsed
##    3.45    0.76    7.23
```

Notes:

- Until not long ago, users had to

```
stopCluster(cl)
```

specify every variable passed to parallel processes. `foreach ... %dopar%` make it far more convenient, “fishing out” any needed variable out of the current environment.

- If you use functions from additional packages, the package names need to be passed down via `.packages`.
- This was a toy example, also to demonstrate one of parallel processing’s **futility bounds: sending jobs that are too small, is not worth the overhead.**
- Our main use for `foreach ... %dopar%` is in tuning, where each job is a complete run.

Parallel Processing with `foreach` and `%dopar%`

```
options(width = 130)
source("../Code/foreachRF.r")
serial = rfTune0(factor(quality) ~ ., data = wine, mvals = 1:6, nodevals = 1:5)  # %do% with no parallel overhead
cl3 = rfTune(factor(quality) ~ ., data = wine, mvals = 1:6, nodevals = 1:5, nclust = 3)
cl6 = rfTune(factor(quality) ~ ., data = wine, mvals = 1:6, nodevals = 1:5, nclust = 6)
c(serial$times[3], cl3$times[3], cl6$times[3])
```

```
## elapsed elapsed elapsed
##   31.26   12.46    8.46
```

Run it, and look at the main dataset returned by `rfTune0`, `rfTune` (that would be the `performance` component), to see if they give the same tuning answer!

Please, **try and write, RIGHT NOW, an analogous parallel-tuning function** for a classifier/regression you plan to use in your project. Questions? *Online* questions?

Cool R Trick 1: `expand.grid`

Did you note this in the `foreachRF` function?

```
mvals = 1:6
nodevals = 1:5
tgrid = expand.grid(mtry = mvals, nodesize = nodevals)
dim(tgrid)
```

```
## [1] 30  2
```

```
head(tgrid)
```

```
##   mtry nodesize
## 1    1        1
## 2    2        1
## 3    3        1
## 4    4        1
## 5    5        1
## 6    6        1
```

This wonderful little base R utility generates a data frame including all level combinations.

Data Manipulation 4.1.1: Yet Another `Xapply`?

Last week, we got a quick tasting of `plyr` and its `ddply` functions, that can carry out manipulations that under the standard `Xapply`s are either cumbersome or impossible.

Nice. But `ddply` et al. do not improve *efficiency*. For this and other purposes, a new package called `data.table` has been developed. It is a closer analogue of SQL than the others. The package defines a new class `data.table`, which inherits from `data.frame` - meaning that you can convert data frames to `data.table` with (nearly?) no adverse impace on the ability to work with them using other functions (plotting, modeling, etc.)

```
library(data.table)
smartrain = read.csv("../Datasets/smarTrain.csv", as.is = TRUE)
smartrain = data.table(smartrain)
```

Test-Running `data.table` vs. `plyr`

```
library(plyr)
system.time(smartscale1 <- ddply(smartrain, "Subject", function(x) data.frame(cbind(scale(x[, 2:562]), class = x$Class))))
```

```
##    user  system elapsed
##    1.68    0.08    1.76
```

```
smartscale2 = copy(smartrain)
scalenames = names(smartscale2)[2:562]
system.time(smartscale2[, `:=`(eval(scalenames), as.data.table(scale(.SD))), .SDcols = scalenames, by = Subject])
```

```
##    user  system elapsed
##    0.61    0.00    0.61
```

Ok… a time-savings of 3x to 5x. Not bad for a fairly short task; `data.table` authors demonstrate much larger savings on heavier tasks. It often helps to define a **key** variable via `setkey` (yes, like with SQL, upon which `data.table` draws heavily). But did we get the same results in this little exercise?

```
table(smartscale1 == data.frame(smartscale2))
```

```
##
##    TRUE
## 4139176
```

We have to “notch down” the `data.table` back to a `data.frame` for the comparison. Also, note that doing e.g., `smartscale2[,2:562]` will try to run a *function* rather than just return the specified columns. You need to do `smartscale2[,2:562,with=FALSE]` to get the latter.

Learn more about `data.table`: http://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.pdf, http://datatable.r-forge.r-project.org/datatable-faq.pdf.

R Trick 2: What Shall We Do to Fill those Empty Spaces?

Reading in data from Excel or other point-click software, we sometimes see this: *(opening spreadsheet…)*

Which, in R, turns into this:

```
missydat = read.csv("../Datasets/naLOCF.csv", as.is = TRUE)
head(missydat, 10)
```

```
##       id time1 value1 time2 value2 time3 value3
## 1   A433   500    6.0  1800      0     0      1
## 2          700    8.0  2000      8   400      2
## 3          900    0.0    NA     NA   800      3
## 4         1100    2.5    NA     NA  1200      2
## 5         1300     NA    NA     NA  1319      1
## 6           NA     NA    NA     NA  1600      6
## 7           NA     NA    NA     NA  2000      6
## 8           NA     NA    NA     NA  2359      6
## 9   A266  1300    6.0  1630      4     0      4
## 10        1500    6.0  1800      4   400      4
```

The empty `time` and `value` entries are probably ok - there's no data there. But the `id` variable needs to be filled down. Turns out, in R it is quite easy:

```
library(zoo)
missydat$id[missydat$id == ""] = NA
missydat$id = na.locf(missydat$id)
head(missydat, 10)
```

```
##       id time1 value1 time2 value2 time3 value3
## 1   A433   500    6.0  1800      0     0      1
## 2   A433   700    8.0  2000      8   400      2
## 3   A433   900    0.0    NA     NA   800      3
## 4   A433  1100    2.5    NA     NA  1200      2
## 5   A433  1300     NA    NA     NA  1319      1
## 6   A433    NA     NA    NA     NA  1600      6
## 7   A433    NA     NA    NA     NA  2000      6
## 8   A433    NA     NA    NA     NA  2359      6
## 9   A266  1300    6.0  1630      4     0      4
## 10  A266  1500    6.0  1800      4   400      4
```

`zoo` is a time-series analysis package. It contains fancier, time-series-appropriate ways of filling those empty space (Eli might teach you more tricks from there in Spring).

Last Trick… But Not Least: Error Bars, the Simple Way!

So… not for the first time, I needed to plot error bars for an article at work. Previously I made do with the somewhat kludgey solutions `plotCI` and `plotmeans` in the `gplots` package.

But now… I'm an R teacher, ain't I? So I searched far and wide. There are error bars in `xYplot` (an expansion of `lattice` functionality, part of `Hmisc`). Also in the super-hyped `ggplot2` - a package with which I still feel ill at ease (Revolution offers a *specific* `ggplot2` class!).

So I fell back to the most primitive solution, courtesy of **anthropologist James Holland Jones:**

One common frustration that I have heard expressed about R is that there is no automatic way to plot error bars (whiskers really) on bar plots. I just encountered this issue revising a paper for submission and figured I'd share my code. The following simple function will plot reasonable error bars on a bar plot.

```
options(width = 130)
error.bar <- function(x, y, upper, lower = upper, length = 0.1, ...) {
    if (length(x) != length(y) | length(y) != length(lower) | length(lower) != length(upper))
        stop("vectors must be same length")
    arrows(x, y + upper, x, y - lower, angle = 90, code = 3, length = length, ...)
}
```

Last Trick… But Not Least: Error Bars, the Simple Way!

```
options(width = 130)
dummyd = data.frame(method = rep(1:3, 3), day = rep(1:3, each = 3), means = sqrt(1:3) + rnorm(9), SEs = runif(9, 0, 0.5))
dummyd$xshift = (dummyd$method - 2)/20  # 'jigger' so that bars don't hide each other
plot(means ~ I(day + xshift), data = dummyd[dummyd$method == 1, ], type = "b", main = "Fun with Simple Error Bars!", xlim = c(0.8,
    3.2), ylim = c(min(dummyd$means - 2 * dummyd$SEs), max(dummyd$means + 2 * dummyd$SEs)), pch = 19, xlab = "Day", xaxt = "n")
lines(means ~ I(day + xshift), data = dummyd[dummyd$method == 2, ], type = "b", col = 2, pch = 19)
lines(means ~ I(day + xshift), data = dummyd[dummyd$method == 3, ], type = "b", col = 3, pch = 19)
axis(1, at = 1:3)
error.bar(dummyd$day + dummyd$xshift, dummyd$means, 2 * dummyd$SEs, col = dummyd$method, length = 0.07)
```

TTFN… and Looking Ahead to Spring

Thank you for investing your time, efforts and resources in this class. I am honored to have taught such a group of people.

I tried to deliver the class from the perspective of an applied statistician, but without losing sight of the theoretical and conceptual angles.

My apologies for the rocky start, and for *still* not coming up with some HW keys… With all that I am satisfied with the breadth and coherence of the statistical material we've covered. It's a lot to digest in a couple of months, but I hope you now have a basic toolset and general mindset to successfully approach modeling problems.

With the rush to get through major regression/classification highlights and examples, the programming angle – as well as some "side" statistical topics – often took a back seat. Items tentatively planned for Winter, now delegated to the Spring wish-list include:

- Direct treatment of publishing via `knitr`, including LaTex equations;
- More graphics, including the fearsome `ggplot2`;
- Smoothing splines and Generalized Additive Models (GAM)

More major topics for Spring include

- Simulation inference: permutation and bootstrap
- Bayesian principles and methods
- Hierarchical and mixed models
- Spatial and temporal models
- Package creation and other advanced-programming topics (S3/S4 and other infrastructure, incorporating other languages, "Big Data", etc.)

We probably won’t be able to cover everything… especially considering that Spring quarter is 1 week shorter.