

# Midterm Exam 2: Mystery Multivariate Data

36-402, Advanced Data Analysis, Spring 2011

## SOLUTIONS

GENERAL NOTE: The data came from a ten-dimensional Gaussian. Each variable had an expected value of 100 and a standard deviation of 15. The correlation matrix was random, different for each data set, and did *not* fit a factor model.

Begin by loading the data. Note that each line begins with a row-number; setting `row.names = 1` prevents this from becoming a variable column.

```
x = read.csv("data.csv", row.names=1)
```

(Alternately, load without `row.names` and delete the first column.) Check that it's got the right properties:

```
> dim(x)
[1] 1000  10
> colnames(x)
[1] "X.1" "X.2" "X.3" "X.4" "X.5" "X.6" "X.7" "X.8" "X.9"
[10] "X.10"
```

1. (a) ANSWER: Make a Q-Q plot:

```
par(mfrow = c(4,3), mar = c(2,2,2,2))
for(k in 1:ncol(x)) {
  qqnorm(x[,k], xlab = "", ylab = "", main = colnames(x)[k])
  qqline(x[,k], col = 2)
}
```

See Figure 1. None of the Q-Q plots seem especially incompatible with normality.

- (b) ANSWER: The test in part (a) plots the empirical quantiles of the observed data against the quantiles of a standard normal distribution. If the data in fact came from a normal distribution, the points of the Q-Q plot should converge towards the Q-Q line as the sample size grows. Slight deviations from the Q-Q line are to be expected for small sample sizes, but if the deviation is too large, this suggests that the data does not come from normal distribution. "Too large" is not defined here, but experienced statisticians can learn to know what is reasonable. The quantitative test described next does not require as much user judgement.

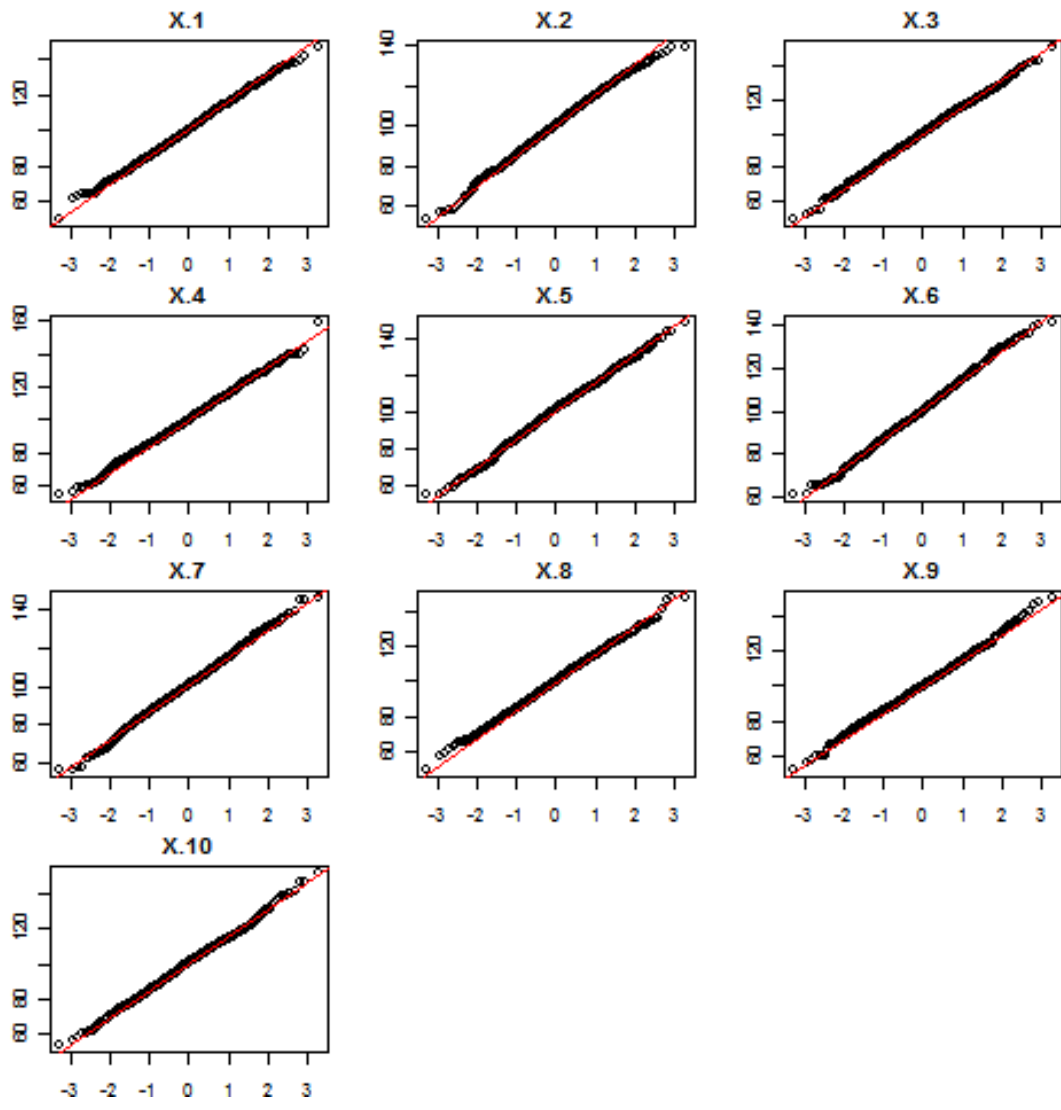


Figure 1:

- (c) ANSWER: The Shapiro-Wilk test (`shapiro.test` in R) tests the null hypothesis that the data come from a normal distribution. The test produces a test statistic  $W$ ; if  $W$  is too small (as judged by the associated  $p$ -value) we reject the null hypothesis. The statistic  $W$  itself looks at the spacing of the “order statistics” of the sample (i.e., the sample values sorted into increasing order).
- (d) ANSWER: Making a scatter-plot of the variable values can show autocorrelation in the data, which would imply they are not independent of each other, but does not directly say anything about whether the data are Gaussian. One could visually attempt to “sum up” the scatter plot from left-to-right (assuming row-number is on the x-axis) and envision a density estimate on the y-axis and think about whether this estimate looks approximately normal, but this would be an exceedingly non-human-friendly, and unreliable, way to get at the question.
2. (a) ANSWER: There are several ways to do this computationally. Here is one, based on code used in Lecture 6.

First, we load the library and do the fitting:

```
require(np)
np.dens = npudens(~X.1+X.10, data = x, tol=0.05,ftol=0.05)
```

This takes 7 seconds on my laptop. It takes much longer without the `tol` and `ftol` arguments, which make `npudens` much less aggressive about optimizing the bandwidth, without however changing the density estimate very much at all.<sup>1</sup>

For plotting purposes, `npudens` automatically creates a  $50 \times 50$  grid, adapted to the range of the data, and evaluated the density on the points of this grid. We extract the fitted values, and then the grid itself (a  $2500 \times 2$  matrix of the coordinates).

```
fhat = plot(np.dens, plot.behavior="data")
np.plot = fhat$d1
np.grid = cbind(np.plot$eval$Var1, np.plot$eval$Var2)
```

Now we make a pretty contour plot of the non-parametric density estimate:

```
require(lattice)
contourplot(np.plot$dens~np.grid[,1]*np.grid[,2],cuts=20,
  xlab="X.1", ylab="X.10", labels=list(cex=0.5),
  main = "Non parametric joint density estimate")
```

See Figure 2.

---

<sup>1</sup>See examples in lectures 2-6 of doing this to speed up non-parametric regression and density estimates.

**Non parametric joint density estimate**

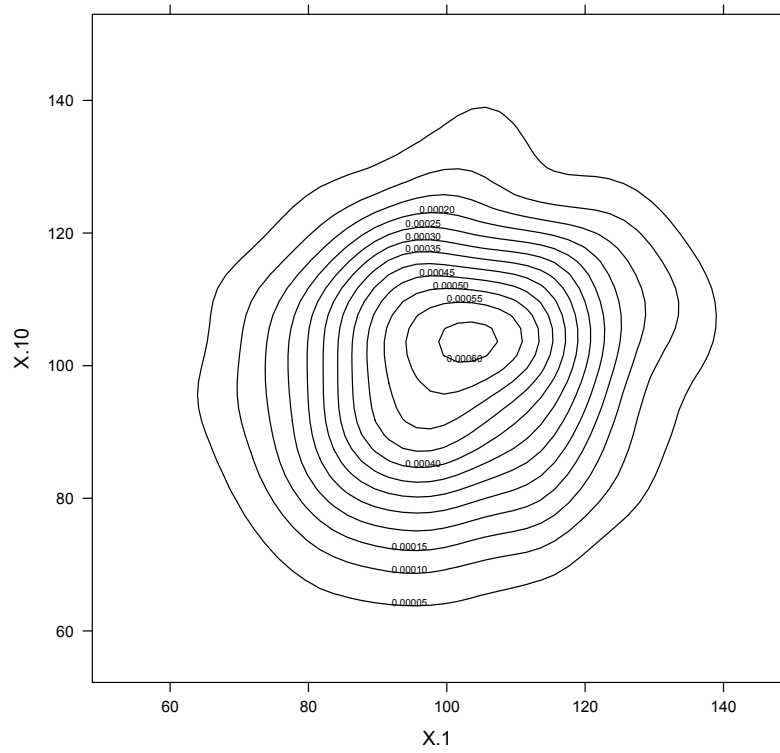


Figure 2:

- (b) ANSWER: As mentioned in class, and in the notes for Lecture 20, the maximum likelihood estimate for a multivariate Gaussian has the sample mean for its estimated mean vector, and the sample covariances for its estimated covariance matrix. (This is also easily checked by writing out the log-likelihood and maximizing.) One could find these means and covariances by hand, but there are built-in functions for the purposes:

```
means = colMeans(x[,c(1,10)])
cov.mat = cov(x[,c(1,10)])
```

Notice that we only want the first and tenth columns of the original data matrix.

Now we evaluate the multivariate Gaussian density (using the `mvtnorm` package), on the same grid as before, and plot it:

```
require(mvtnorm)
fitted.gauss = dmvnorm(np.grid,means,cov.mat)
contourplot(fitted.gauss~np.grid[,1]*np.grid[,2],cuts=20,
  xlab="X.1", ylab="X.10", labels=list(cex=0.6),
  main = "Fitted Gaussian joint density estimate")
```

See Figure 3.

- (c) ANSWER: Since we have evaluated the two density estimates on the same grid of points, we can just subtract one from the other, and plot:

```
diff.np.gauss = fitted.gauss - np.plot$dens
contourplot(diff.np.gauss~np.grid[,1]*np.grid[,2],cuts=20,
  xlab="X.1", ylab="X.10", labels=list(cex=0.6),
  main = "Gaussian fit minus non parametric fit")
```

See Figure 4. Even if the Gaussian models is right, there would be some difference between it and a non-parametric estimate. Ideally, however, this should be small, and the difference between the two fits should itself follow the pattern predicted by the parametric model. (Think about what we would get if we divided the data in two, fitted a separate Gaussian for each, and plotted the difference.)

- (d) ANSWER: The procedures described in problem 1 work for 1-dimensional data, and it is not clear how to adjust these procedures to apply for a two-dimensional distribution with correlation between the variables. Q-Q plots need quantiles, and quantiles need us to be able to put the data in order, which is hard with two variables. The Shapiro-Wilks test also relies on order statistics.

Here is an alternative test that could work: look at the difference in log likelihood between parameteric and non-parametric estimate, then get the distribution of this under the Gaussian by simulation (a la comparing regression models). Simulation would require running

Fitted Gaussian joint density estimate

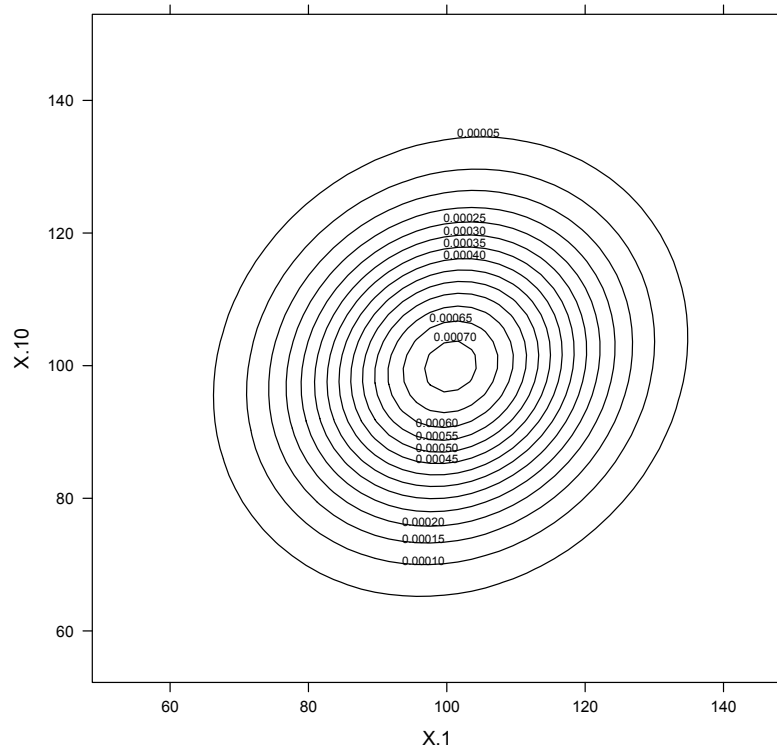


Figure 3:



the non-parametric estimate many times, which would take a substantial amount of time. This is why actually implementing the test is considered extra credit.

- (e) ANSWER: EXTRA CREDIT As usual, we separate calculating the test statistic from doing the simulation. Here we find the difference in log-likelihoods:

```
diff.log.like <- function(data) {
  my.np <- npudens(~X.1+X.10,data=data,tol=0.1,ftol=0.1)
  my.means <- colMeans(data)
  my.cov <- cov(data)
  np.ll <- sum(log(predict(my.np)))
  gauss.ll <- sum(log(dmvnorm(data,my.means,my.cov)))
  return(np.ll - gauss.ll)
}
```

Applied to the data,

```
> diff.log.like(x[,c(1,10)])
[1] 15.28891
```

The simulation is very simple:

```
p2sim <- function() {
  sim <- rmvnorm(nrow(x),means,cov.mat)
  sim <- data.frame(X.1=sim[,1],X.10=sim[,2])
  return(sim)
}
```

produces the right number of random Gaussian vectors (in this case, 1000), with the right means and covariances, and the right column names (for use in `diff.log.like`.

```
test.stats <- replicate(100,diff.log.like(p2sim()))
```

repeats the simulation 100 times and stores the tests. The test statistic is right in the middle of this sampling distribution:

```
> summary(test.stats)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 10.11  14.52   16.37   15.61  17.93   19.21
```

giving no evidence whatsoever against the Gaussian distribution.

### 3. We do the PCA:

```
x.pca = prcomp(x)
```

- (a) ANSWER: This is very much the same as the problems in homework 9 asking for plots of the components. Remember that `prcomp` stores the principal component vectors in an array called `rotation`, with each column a different principal component.

```

plot(x.pca$rotation[,1],type="l",col=1, ylim = c(-1, 1), lwd = 2,
     xlab = "Column index", ylab = "Principal component projection",
     main = "Projection of components onto the 10 variables")
lines(x.pca$rotation[,2],type="l",col=2, lwd = 2)
lines(x.pca$rotation[,3],type="l",col=3, lwd = 2, lty = 2)
lines(x.pca$rotation[,4],type="l",col=4, lwd = 2, lty = 3)
lines(x.pca$rotation[,5],type="l",col=6, lwd = 2, lty = 4)
legend(1, 1.1, c("Component 1", "Component 2", "Component 3",
                 "Component 4", "Component 5"), col = c(1,2,3,4,6),
       text.col = c(1,2,3,4,6), lty = c(1,1,2,3,4), bty = "n")

```

See Figure 5. Note that the first component is all positive, indicating that all correlations are positive; the other components have alternating signs. The exact pattern you see will depend on your exact data set.

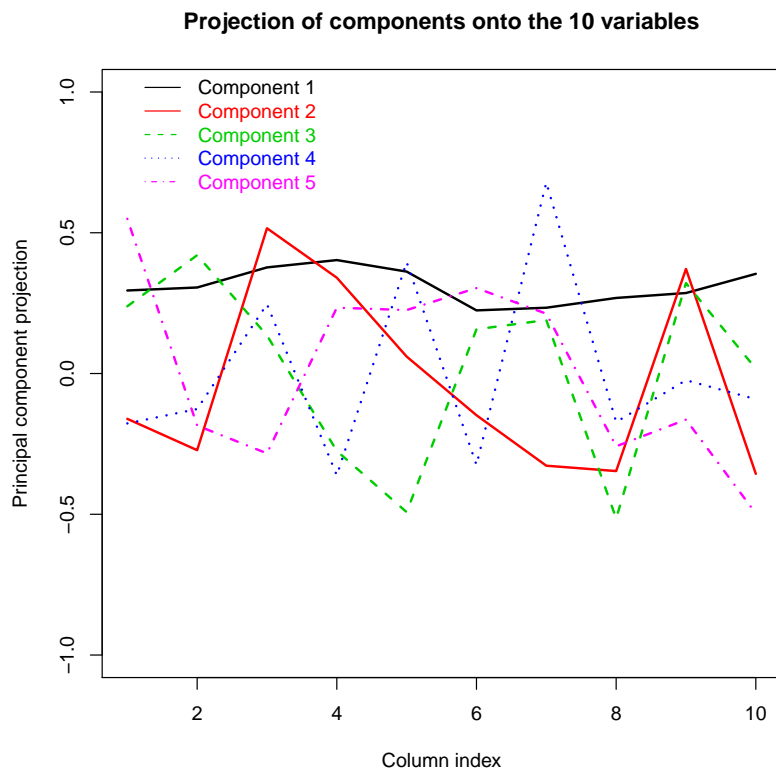


Figure 5:

(b) ANSWER: Using the same code as in Homework 9, problem 4b. See Figure 6 for the result.

```

cumulative.pct = cumsum(x.pca$sdev^2)/sum(x.pca$sdev^2)
plot(1:10, cumulative.pct, type = "b", xlab = "Number of components",
     ylab = "Fraction of variance retained by principal components")

```

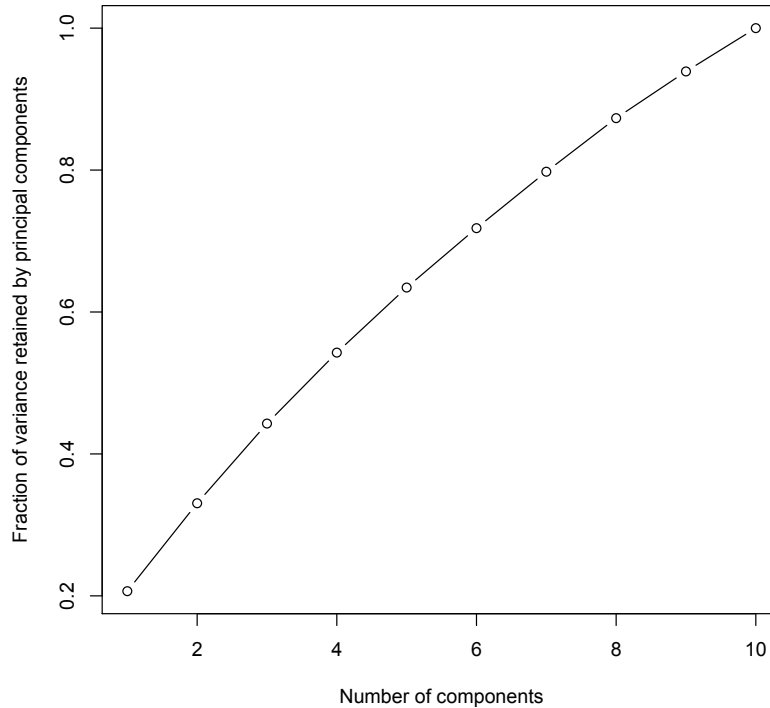


Figure 6:

4. (a) ANSWER: We do factor analysis with one factor:

```
x.fa = factanal(x,factors=1)
```

The factor loadings are stored in an array called `loadings`. We plot it along with the first principal component, to facilitate the comparison.

```

plot(x.pca$rotation[,1],type="l",col=2, ylim = c(0, 0.8), lwd = 2,
     xlab = "Column index", ylab = "Projection")
lines(x.fa$loadings,type="l",lwd = 2)
legend(1, 0.7, c("FA first factor", "PCA first component"),
     col = c(1,2), text.col = c(1,2), lty = c(1,1), bty = "n")

```

See Figure 7. The factor seems quite similar to the first principal component, as we would hope. The goal of PCA is to find the line

(or plane, hyper-plane...) which comes closest, on average, to the data. The goal of FA is to find a line (plane, etc.) which the data has a Gaussian distribution around. If we are limited to just one factor or component, both methods should at least get something similar.

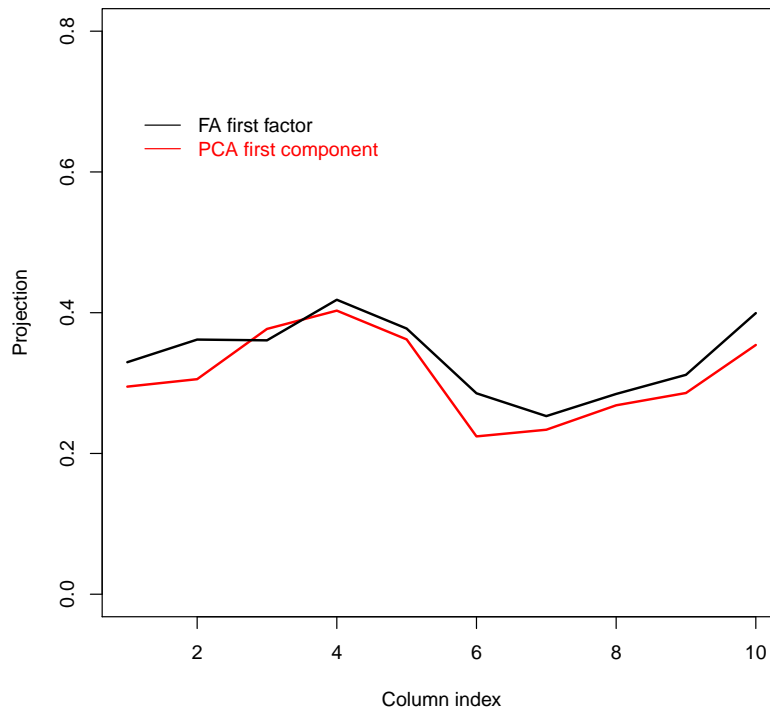


Figure 7:

(b) ANSWER: Fit a two-factor model:

```
x.fa2 = factanal(x,factors=2)
```

Plot the two factors along with the factor from part (4a) for comparison:

```
plot(x.fa2$loadings[,1],type="l", ylim = c(0, 0.8), lwd = 2,
     xlab = "Column index", ylab = "Projection")
lines(x.fa2$loadings[,2],type="l",lwd = 2, col=2)
lines(x.fa$loadings,type="l",lwd = 1, col=3, lty = 2)
legend(1, 0.7, c("First factor", "Second factor", "Former first factor"),
      col = c(1,2,3), text.col = c(1,2,3), lty = c(1,1,2), bty = "n")
```

See Figure 8, for the factor loadings of the variables for the first and second factors in black and red, respectively. The green dashed line is the factor from part (a), and we see that it is much different from the new first factor. This is typical. Remember the rotation problem: we can apply *any* orthogonal matrix to the factor loadings we like, without *any* effect on the distribution of the data. There are only two orthogonal “matrices” for one-dimensional factors (the numbers +1 and -1), but there are infinitely many in two or more dimensions, and which rotation depends heavily on the algorithmic details of `factanal`, and even on which machine this is run on.

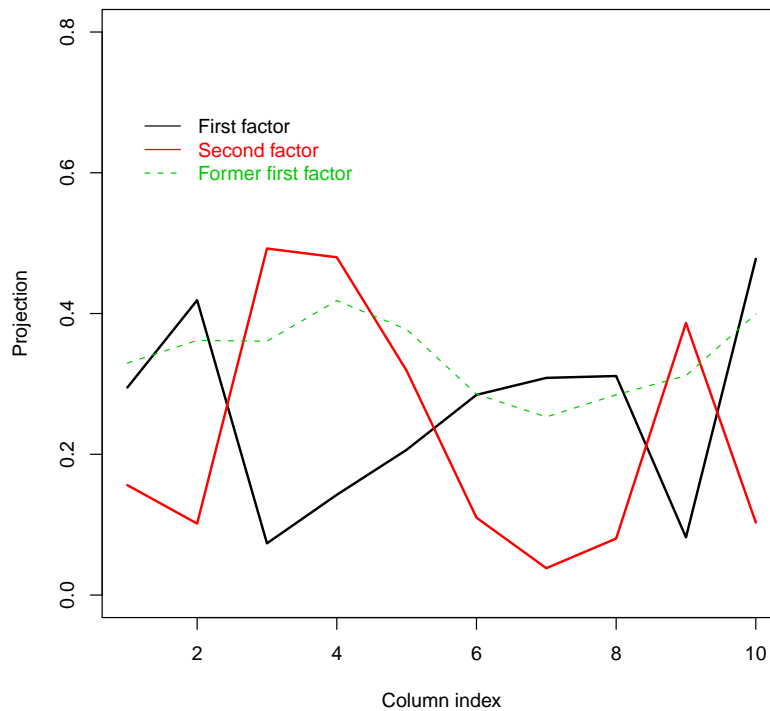


Figure 8:

- (c) ANSWER: The output for `factanal` includes the cumulative fraction of variance retained by the first  $k$  factors. But because we cannot say that (for example) the first factor in a two-factor model is the same as the first factor in a one- or three- factor model, it does not make much sense to compare this across different models. (Applying a rotation changes nothing physical but will change these fractions.)

Instead, fit models with different numbers of factors, and look at the total amount of variance retained by each one.

It is worth thinking through how R calculates this; it also lets us automate it. As a preliminary step in factor analysis (done inside `factanal`, each of the  $p$  observable variables (= columns of the data frame) is centered to have mean zero and scaled to have variance 1. The total amount of variance is thus  $p$ , which here is 10. After estimation, we have a “uniqueness” parameter  $\psi_i$  for each observable  $i$ , which is the amount of variance in that variable which is *not* due to the factors, i.e., is unique to that variable. So the amount of variance which *is* due to the factors is  $1-\psi_i$ . Summing up over observables, the variance retained by the factors is  $\sum_{i=1}^p 1-\psi_i = p - \sum_{i=1}^p \psi_i$ . The fraction of variance retained is then  $\frac{p - \sum_{i=1}^p \psi_i}{p} = 1 - p^{-1} \sum_{i=1}^p \psi_i$ . Let’s encapsulate this in a function:

```
var.fraction <- function(fa) {
  psi <- fa$uniquenesses
  p <- length(psi)
  return(1-sum(psi)/p)
}
```

Check that this isn’t just crazy talk:

```
> x.fa2
# lots of stuff snipped
          Factor1 Factor2
SS loadings    0.839  0.789
Proportion Var 0.084  0.079
Cumulative Var 0.084  0.163
# more stuff snipped
> var.fraction(x.fa2)
[1] 0.1627786
```

so this agrees with R’s internal calculations. Now do this for 1–5 factors:

```
var.fraction.with.q.factors <- function(q,data=x) {
  return(var.fraction(factanal(data,factors=q)))
}
```

```
cumulative.pct.factors <- sapply(1:5,var.fraction.with.q.factors)
```

Finally, plot:

```
plot(1:5, cumulative.pct.factors, type = "l", ylim=c(0,1),
     xlab = "Number of factors or components",
     ylab = "Fraction of variance retained")
lines(1:5, cumulative.pct[1:5], lty = 2)
legend(1,0.8,legend=c("Factors","Principal Components"),lty=c(1,2),bty="n")
```

See Figure 9. The solid line shows the fraction of variance retained by the first  $q$  factors. For comparison the principal component shares are included on the plot as the dashed line. The factors account for less of the variance than the principal components. This is not surprising, because principal components are *designed* to maximize the amount of variance retained, while factors are supposed to retain the pattern of correlations.

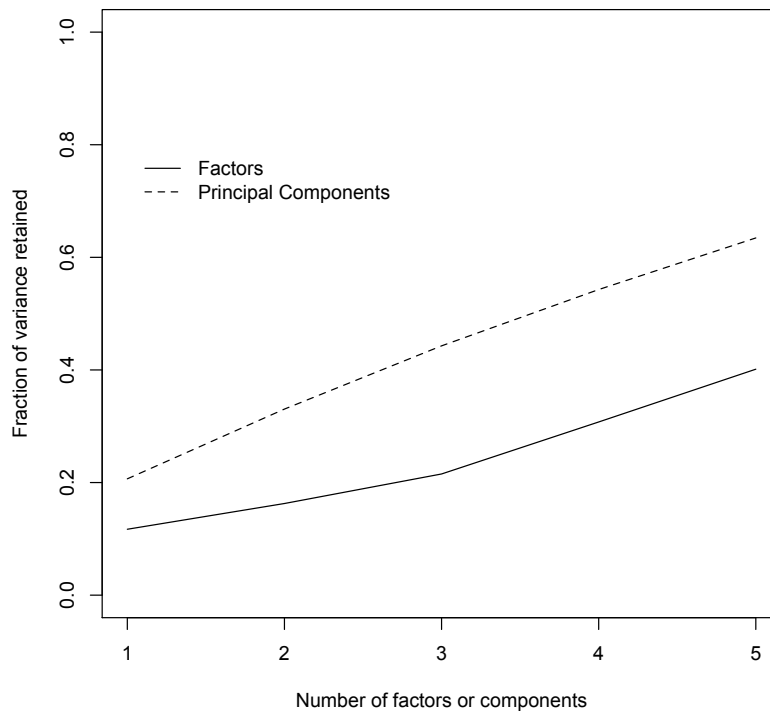


Figure 9:

5. (a) ANSWER: Lecture 18 gives the log-likelihood in equation (34). Section 7.1 in lecture 18 explains how to use the log-likelihood ratio test to choose  $q$ . Suppose you've fit a  $q$ -factor model (say, starting with  $q = 1$ ). Now you want to know if including another factor will improve the model. Fit the  $q = 2$  model, which is a generalization of the  $q = 1$  model, and compute the difference in log-likelihoods. If the one-factor model is right, *two times* the difference in log-likelihoods has a  $\chi^2$  distribution. The number of degrees of freedom for the  $\chi^2$  will be the difference in the number of parameters for the larger

model and that for the smaller model — see pp. 9–10 of Lecture 18 for the derivation of the fact that the number of free parameters in a  $q$ -factor model for  $p$  observable variables is<sup>2</sup>

$$pq - q(q - 1)/2$$

If the test statistic — the difference in log-likelihoods — is improbably large under this  $\chi^2$  distribution, then you get a small  $p$ -value, and can reject the null hypothesis that the small model is just as good. You would then conclude that it is better to use two factors rather than one, and would test whether 3 factors might not be better still, and so on until the  $p$ -values are no longer small. In short, this is just like the example of selecting the size of a mixture model in lecture 20, only we can use the  $\chi^2$  distribution rather than bootstrapping.

- (b) ANSWER: It is a slight annoyance that R doesn't actually record the log-likelihood of a factor model. We can recover it from the estimated parameters very simply, however.<sup>3</sup>

Remember (see Lecture 18) that in a factor model, (i) the observables are centered and scaled, and (ii) the scaled observations have a multivariate Gaussian distribution, with mean 0 and covariance matrix  $\mathbf{w}^T \mathbf{w} + \Psi$ , where  $\mathbf{w}$  is the  $q \times p$  loading matrix, and  $\Psi$  is the  $p \times p$  diagonal matrix of uniquenesses. For what I'm sure seemed like good reasons long ago, the `loadings` attribute of a `factanal` object is a  $p \times q$  matrix, the transpose of what we want.

```
factor.cov <- function(fa) {
  w <- t(fa$loadings)
  Psi <- diag(fa$uniquenesses)
  return( t(w) %*% w + Psi)
}
```

`t()` transposes a matrix, `%*` does matrix multiplication, and `diag()` creates a diagonal matrix.

```
dfactanal <- function(x,fa) {
  require(mvtnorm)
  p <- ncol(x)
  return(dmvnorm(scale(x),mean=rep(0,p),sigma=factor.cov(fa)))
}
```

This function assumes that `x` is an array where each row represents a vector, and calculate the probability density of a factor model at each

---

<sup>2</sup>The  $pq$  term comes from the size of the factor loading matrix; the  $q(q - 1)/2$  comes from the constraint that each factor's loading vector should be orthogonal to all the others. Ignoring it results in a somewhat more conservative test.

<sup>3</sup>The `criteria` attribute of a `factanal` object contains something which is actually monotonically related to the log-likelihood, but by the time we turn it back into a log-likelihood we've done so much work we're better off starting from scratch.

of those vectors. (Cautious programming would check that `x` has the right number of columns to work with the factor-model object `fa`.) The `scale()` function centers and scales each column separately.) It returns as many numbers as there are rows of `x`.

```
loglik.factanal <- function(x,fa) {
  return(sum(log(dfactanal(x,fa))))
}
```

Here at last is the log-likelihood function.

```
logliks <- vector(length=5)
num.params <- vector(length=5)
p=10
for (q in 1:5) {
  logliks[q] <- loglik.factanal(x,factanal(x,factors=q))
  num.params[q] <- p*q-q*(q-1)/2
}
test.stats <- 2*diff(logliks)
dfs <- diff(num.params)
```

Now that we have the test statistics and the number of degrees of freedom, we can calculate  $p$ -values:

```
> signif(pchisq(test.stats,df=dfs,lower.tail=FALSE),3)
[1] 1.42e-12 2.73e-11 2.24e-06 3.93e-03
```

Since even the largest of these, the  $p$ -value for comparing 4 factors to 5, is less than  $4 \times 10^{-3}$ , we can be quite confident in preferring a five-factor model to any smaller factor model.

- (c) ANSWER: As explained in Lecture 18, the usual goodness-of-fit test for factor models uses the observation that every factor model is a multivariate Gaussian distribution, but with a special, constrained covariance matrix. We can thus use the log-likelihood ratio test to test the null hypothesis that the factor model is right against the alternative that an unrestricted Gaussian is genuinely better. This is, in effect, testing whether the covariance matrix obeys the restrictions of the factor model to within sampling error. We could do this by hand, just as above (the number of parameters for the unrestricted Gaussian is  $p(p-1)/2$  — why?), but `factanal` actually does this for us. The  $p$ -value for this  $\chi^2$  test is given in the `PVAL` element:

```
> x.fa5 <- factanal(x,5)
> x.fa5$PVAL
  objective
0.02098555
```

- (d) ANSWER: The very small  $p$ -value shows that the model does not match the data. This might not be a practically significant issue. We

have only shown, with high statistical significance, that the model is at least a little off. In fact, as mentioned at the beginning of these notes, the factor model is here completely wrong (but nonetheless “explains” 40% of the variance).

6. (a) ANSWER: This fits a two-component multivariate Gaussian mixture model by EM — the same sort of thing `mvnormalmixEM` does (only that allows for more components). It is *exactly* what was described in Lecture 19. The inputs are the data array, `x`; the number of E-M step pairs, `t`; and the amount by which the posterior probability of each point is allowed to change before the iteration halts, `eps`. The output is a list with the means of the two components, the covariance matrices, the two component weights, the set of posterior probabilities for each point, the number of iterations performed, and the amount by which posterior probabilities changed at the last step.

- (b) ANSWER: See the commented code:

```
# Fit two-component multivariate Gaussian mixture by EM
# Inputs: data array (x)
# maximum number of EM steps tried (t)
# tolerance level: iteration stops when difference in posterior probability is
# below this; adjusted to number of data points (eps)
# Presumes: mixtools library is available for dmnorm()
# Output: See above
myfunction <- function(x, t=100, eps=1e-2/sqrt(nrow(x))) {
  # Make surer mixtools loaded and x is an array; if either condition
  # is not met, quit and explain what's wrong:
  stopifnot(require(mixtools), is.array(x))
  # Record the number of observations:
  n <- nrow(x)
  # Make w a vector of alternating 0's and 1's, one per observation
  w <- rep(c(0,1), length.out=n)
  # Re-arrange w randomly:
  w <- sample(w)
  # Set del so it is bigger than eps for any finite eps:
  del <- Inf
  # Initialize a counter:
  i <- 0
  # Do EM as long as it has not converged to within eps, for up to t iterations
  while ((del > eps) && (i < t)) {
    # Keep track of how many iterations you've done:
    i <- i+1
    # Get the mean of absolute values in w, the total weight of component 1
    l1 <- mean(abs(w))
    # The mean will always be in [0,1]; compute its complement:
    l2 <- 1 - l1
```

```

# Find the weighted mean of each column of x with weights w:
m1 <- apply(x,2,weighted.mean,w=w)
# Find the weighted mean of each column of x with complementary
# weights 1-w:
m2 <- apply(x,2,weighted.mean,w=(1-w))
# Compute the covariance matrix for x with rows weighted by w
s1 <- cov.wt(x,wt=w)$cov
# Compute the covariance matrix for x with row weights 1-w
s2 <- cov.wt(x,wt=(1-w))$cov
# For each row of x find likelihood of generating that row from
# a Gaussian distribution with means m1 and covariances s1
p1 <- dmvnorm(x,m1,s1)
# Compute the same likelihoods with m2, s2
p2 <- dmvnorm(x,m2,s2)
# Make new weights. For each observation, the new weight is the likelihood
# of the data coming from the first Gaussian over the total likelihood of
# coming from either of the two proposed Gaussians, with likelihoods
# weighted by the mixing weights of the components --- Bayes's rule
wn <- l1*p1/(l1*p1+l2*p2)
# Find the maximum difference between old and new weights; if this is
# smaller than eps, we're done
del <- max(abs(wn-w))
# Replace the old weights with the new weights
w <- wn
}
# Return everything in a list
return(list(m1=m1,m2=m2,s1=s1,s2=s2,l1=l1,l2=l2,w=w,t=i,del=del))
}

```

- (c) ANSWER: The first `abs` commands is not necessary; tracing through the function, we see that a `w` value can never become negative. The second `abs` is necessary; the algorithm should not stop if all weights *decrease* by more than `eps`.

- (d) ANSWER: To run the function, put the data in array form first:

```
out = myfunction(as.matrix(x))
```

Evidently the algorithm didn't converge in 100 iterations:

```
> out$t
[1] 100
```

The last deviation was outside of the tolerance substantially:

```
> 1e-2/sqrt(nrow(x))
[1] 0.0003162278
> out$del
[1] 0.0104
```

We can re-run with more iterations until it does converge; 400 steps almost always works here. (Doing this is not required for the problem.)

We get two ten-dimensional Gaussians, arbitrarily numbered “1” and “2”, with respective mixing weights of 0.453 and 0.547 (`out$11` and `out$12`). Figure 10 shows the mean vectors for the two Gaussians, in the same style as the earlier plots of principal components and factor loadings. Figures 11–12 shows heat-maps of their covariance matrices.

7. (a) The argument `maxit` is the maximum number of E-M steps which `mvnormalmixEM` will perform; if the algorithm hasn’t converged by then, it stops anyway. (This corresponds to `t` in problem 6.) `epsilon` is the convergence criterion: if the log-likelihood changes by less than this over one iteration of EM, the algorithm is declared converged. This is conceptually similar to the `eps` of problem 6, but works with log-likelihood and not the posterior probabilities.

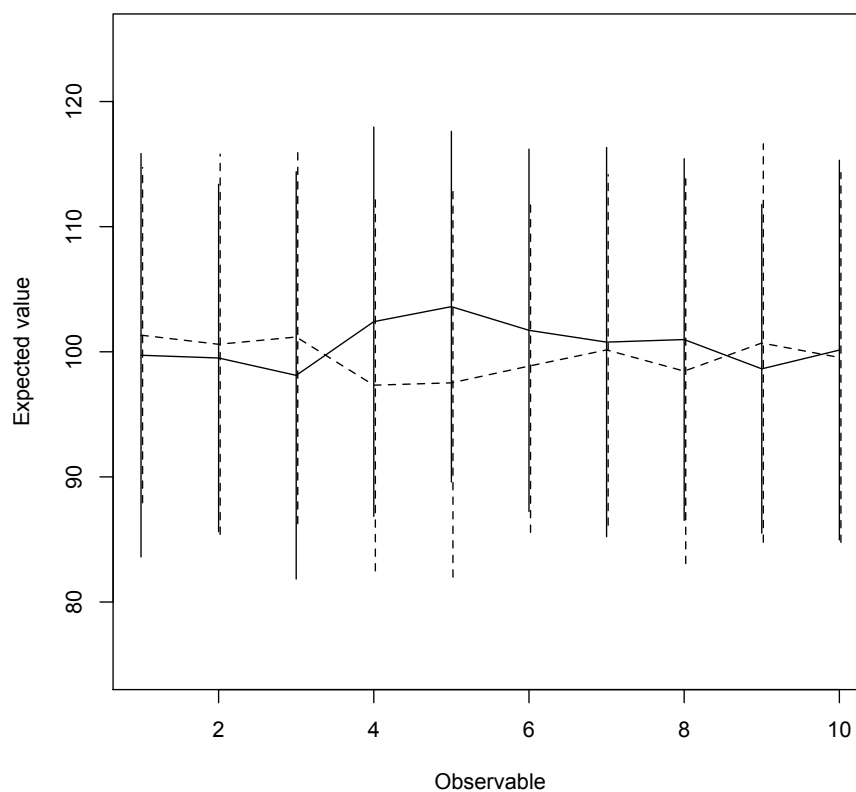
```
library(mixtools)
mix2 <- mvnormalmixEM(x,k=2,maxit=100,epsilon=1e-1)
mix3 <- mvnormalmixEM(x,k=3,maxit=100,epsilon=1e-1)
mix4 <- mvnormalmixEM(x,k=4,maxit=100,epsilon=1e-1)
mix5 <- mvnormalmixEM(x,k=5,maxit=100,epsilon=1e-1)
mix6 <- mvnormalmixEM(x,k=6,maxit=100,epsilon=1e-1)
```

*Note:* If you have previously loaded the `mvtnorm` package, you might need to unload it at this stage, since that package and `mixtools` define some functions with the same names and the same purposes, but different syntaxes, and R can get them confused.

With the models around, we can plot the log-likelihoods; I have added the log-likelihood from a plain multivariate Gaussian, which is, so to speak, a one-component mixture, as well.

```
logliks.mixes <- c(mix2$loglik,mix3$loglik,mix4$loglik,mix5$loglik,mix6$loglik)
loglik.mvn <- sum(log(dmvnorm(as.matrix(x),colMeans(x),cov(x))))
logliks.mixes <- c(loglik.mvn,logliks.mixes)
plot(logliks.mixes,type="b",xlab="Number of mixture components",
      ylab="In-sample log likelihood")
```

- (b) See the notes for lecture 20, where there is also a worked example of using `boot.comp`.
- (c) `source("http://www.stat.cmu.edu/~cshalizi/402/lectures/20-mixture-examples/bootcomp")`  
`bc <- boot.comp(x,max.comp=6,mix.type="mvnormalmix",maxit=100,epsilon=1e-1,B=50)`  
 After running overnight, this selects a single component.
- (d) Cross-validation would be a reasonable way of comparing the two models. We want to know which one fits better, i.e., which one better describes the under-lying population. Cross-validation measures this quite directly.

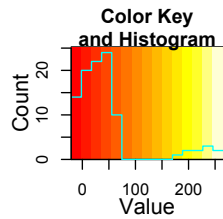


```

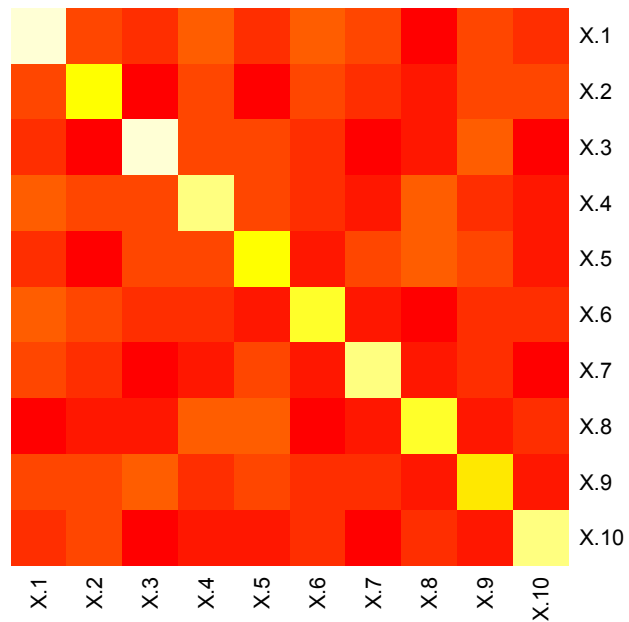
stddevs1 <- sqrt(diag(out$s1))
stddevs2 <- sqrt(diag(out$s2))
plot(1:10,out$m1,ylim=c(100-25,100+25),type="l",xlab="Observable",
     ylab="Expected value")
segments(x0=1:10,x1=1:10,y0=out$m1-stddevs1,y1=out$m1+stddevs1)
lines(1:10,out$m2,lty=2)
segments(x0=(1:10)+0.02,x1=(1:10)+0.02,y0=out$m2-stddevs2,y1=out$m2+stddevs2,lty=2)

```

Figure 10: Solid line: expected values of the observables from the first component. Dashed line: expected values from the second component. Vertical lines:  $\pm 1$  standard deviation, based on the covariance matrix of each component. (The standard deviation lines for component 2 are plotted very slightly off-set to the right for visual clarity.)

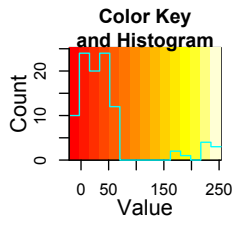


Covariance matrix for component 1



```
library(gplot)
heatmap.2(out$s1,Rowv=FALSE,Colv=FALSE,dendrogram="none",scale="none",
          trace="none",main="Covariance matrix, component 1")
heatmap.2(out$s2,Rowv=FALSE,Colv=FALSE,dendrogram="none",scale="none",
          trace="none",main="Covariance matrix, component 2")
```

Figure 11: This figure and next: heatmaps depicting the covariance matrices for the two components of the Gaussian mixture estimated by myfunction.



**Covariance matrix, component 2**

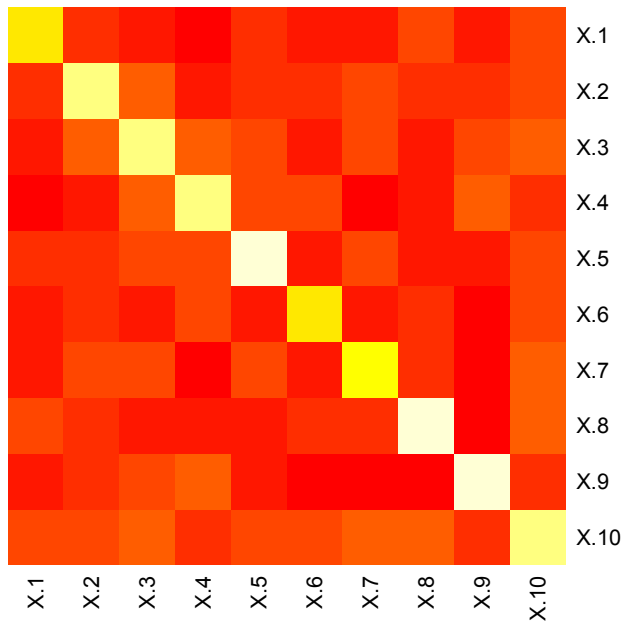
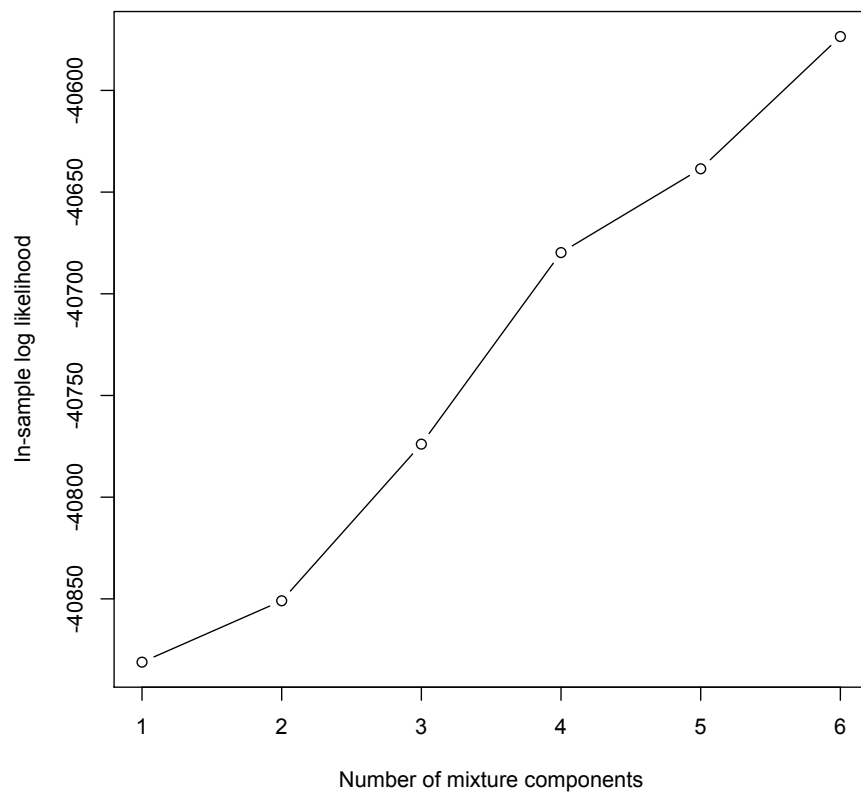


Figure 12:



- (e) Since `boot.comp` selected just a single mixture component, we are comparing a five-factor model to an unrestricted multivariate Gaussian by cross-validation.

```

cv.factors.vs.mixtures <- function(n.folds=5,q.factors,k.mixes,x,...) {
  x <- scale(x)
  n <- nrow(x)
  case.folds <- rep(1:n.folds,length.out=n)
  case.folds <- sample(case.folds)
  factor.logliks <- vector(length=n.folds)
  mixture.logliks <- vector(length=n.folds)
  for (fold in 1:n.folds) {
    test <- x[case.folds == fold,]
    train <- x[case.folds != fold,]
    fa <- factanal(train,factors=q.factors)
    factor.logliks[fold] <- loglik.factanal(test,fa)
    if (k.mixes > 1) {
      mvnmix <- mvnormalmixEM(train,k=k.mixes,...)
      mixture.logliks[fold] <- loglike.mvnormalmix(test,mvnmix)
    } else {
      mu <- colMeans(train)
      sigma <- cov(train)
      mixture.logliks[fold] <- sum(log(dmvnorm(test,mu,sigma)))
    }
  }
  cv.mixture <- mean(mixture.logliks)
  cv.factor <- mean(factor.logliks)
  choice <- ifelse(cv.mixture > cv.factor,"mixture","factor")
  return(list(choice=choice,cv.mixture=cv.mixture,cv.factor=cv.factor,
             mixture.logliks=mixture.logliks,factor.logliks=factor.logliks))
}

```

Notes: (i) The  $k = 1$  case requires special handling, because while conceptually this is just fitting a single multivariate Gaussian to the whole data, the `mvnormalmixEM` algorithm chokes when asked to do so. (ii) Since we want to use out-of-sample log likelihood, we need to write a function, `loglike.mvnormalmix`, to calculate it for mixtures. We saw how to write this for mixtures of one-dimensional Gaussians in Lecture 20; modifying that to work with multivariate Gaussians is trivial. (iii) The `...` meta-argument lets us pass control settings to `mvnormalmixEM`.

When I run this, it selects the one-component mixture.