The Mysteries of Mixtures of Regressions

Rolf Turner

38 Cunnegonda, 2084

1 Introduction

Back in the late 1990s I developed a package `mixreg` to accompany a paper that I had written (Turner 2000). This package wasn't actually called a "package" as such, since at that time I was working with Splus which used different terminology. In 2004 I converted the code to R code and created a genuine R package. Until a short time ago the Splus origins of the package were revealed by comment lines at the end of the help files:

```
% Converted by Sd2Rd version 1.21.
```

Recently I have undertaken a substantial revision of the `mixreg` package, cleaning up the code and adding various enhancements. The old code was very kludgy in many respects; my R-coding skills have improved substantially since the inception of the package. It might be said that my R-coding skills are still nothing to write home about. However they are a lot better than they were. The package has also been improved by the addition of some new and interesting data sets (the `kiln` data) which were kindly made available to me by Petr Pikal (Prerov, Czech Republic).

In this vignette I discuss the changes that I have made and provide some guidance in coping with these changes. I also discuss some problems that I encountered in fitting models to the new data sets and suggest an approach to solving such problems.

2 Some of the changes in the package

The most important change is that the "old" `mixreg()` function had a syntax `mixreg(x,y,...)` that involved specifying the variable names `x` and `y` (the predictor and the response, the former being allowed to be a matrix thus allowing for multiple predictors.) I have designed the new `mixreg()` function so that the old syntax still works ("backward compatibility") but the preferred syntax is now formula based: `mixreg(y ~ x, ...)`

The function determines which syntax is to be used on the basis of the presence of the `y` argument of the function. If `y` is missing, then `x` must be a formula; if isn't then an error is thrown. Likewise an error will be thrown if you supply a formula as the value of `x`, and also perversely supply a value of the `y` argument.

Another important change is the addition of a function `visualFit()` which permits the user to obtain starting values for the EM algorithm in an interactive manner. The function plots the data and then invites the user to "click" on pairs of points judged to lie on lines corresponding to the various components. Good starting values are very important and `visualFit()` provides a convenient means of obtaining pretty good ones. This function only works with models that involve a single predictor. The `mixreg()` package allows for an arbitrary number of predictors, and there exist examples in which multiple predictors occur (see the references in Turner 2000). However determining good starting values mixtures of regressions with multiple predictors remains a very challenging problem. Designing a visually based method that would be applicable in the multiple predictor context does not appear to be feasible.

The package now handles models in which there are *no* predictors, so it is capable of fitting scalar mixtures (Gaussian only) in a rudimentary manner. Users would however be better advised to avail themselves of, for example, the `mixdist` package, Macdonald (2018).

A plot method for plotting the results of a fit has been added. The existing functions for plotting confidence and prediction bands (`cband()` , `plot.cband()`) have been tidied up and slightly enhanced. The procedure for handling residuals has been tidied up a bit. A generic function `rmixreg` with a `"default"` method and a `"mixreg"` method, for simulating data from mixtures of regressions models has been added. A Monte Carlo procedure for estimating the covariance matrix of the parameter estimates has been added. A function `stepPlot()` has been added. This function fits a mixtures of regressions model, one EM step at a time, plotting the result after each step. It *might* be useful for giving insight into the reasons for convergence problems or for unexpected characteristics of the fits.

The names of some function arguments have changed and the names of the variables in the `aphids` data set have been changed from "`n.aphids`" (the number of aphids released) and "`n.inf`" (the number of plants, out of a possible 69, that were infected) to "`aphRel`" and "`plntsInf`" respectively. When the unexpected occurs, read the help!!!

3 The new data sets

The data in the `kiln` data sets are very clearly "arranged" in distinct lines (three in the case of kiln A, two in the case of kiln B), whence one might expect it to be very easy to fit mixtures of regressions models to these data. Easy or not, starting values are required, and a good way to produce these starting values is by using the function `visualFit()` mentioned in Section 2. However this function requires "interactive input" from the user so it cannot be used in the `knitr` context with which this vignette is being produced. Consequently I deployed `visualFit()` outside of the `knitr` context and saved the result. The code used looked like this:

```r
# Note that eval=FALSE was set in this code chunk.
vfit <- visualFit(y ~ x,data=kilnAoneOut,ncomp=3)
saveRDS(vfit,file="vfit.rds")
```

In the code used to create Figure 1, the previously created value of `vfit` is read in.

```r
vfit <- readRDS("vfit.rds")
fit  <- mixreg(y~x,data=kilnAoneOut,thetaStart=vfit$theta,
               verb=FALSE)
plot(fit)
```

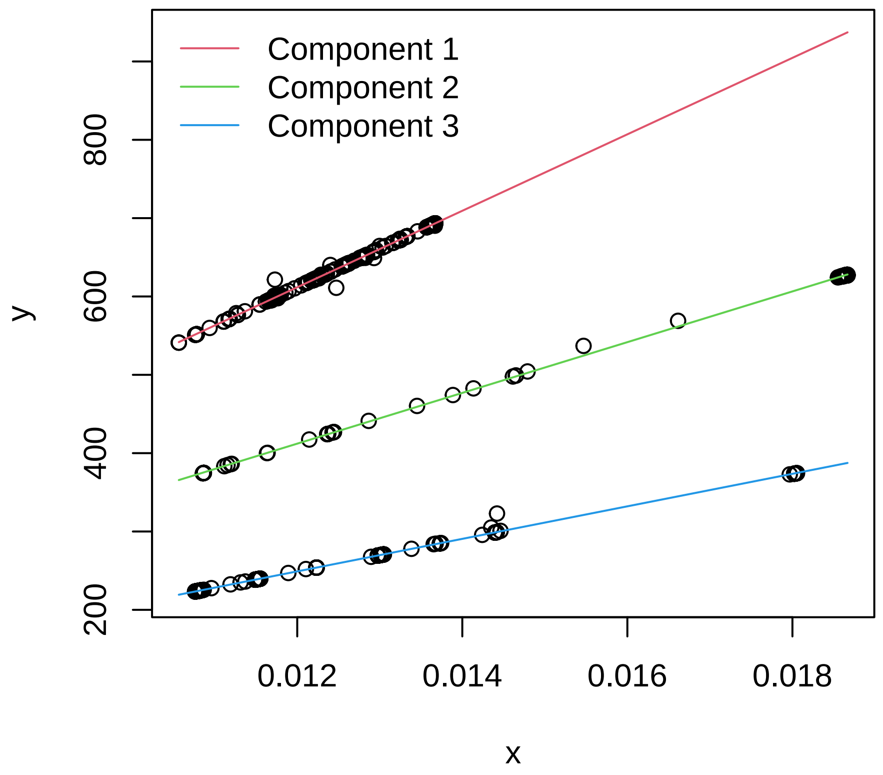

Figure 1: Fit of the `kilnAoneOut` data

A similar procedure (reading in previously saved fits that were created by `visualFit()`) will be used throughout the remainder of this vignette, without further comment, and the "reading in" step will not be shown.

The fit to the `kilnAoneOut` data, shown in Figure 1, looks excellent as seems to be often the case when the starting values are obtained from `visualFit()`. However if one uses random starting values, as could conceivably be necessary in some circumstances, fits can sometimes look just awful:

```
fitBad  <- mixreg(y~x,data=kilnAoneOut,verb=FALSE,ncomp=3,
                  seed=42)
plot(fitBad)  # Looks atrocious!
```

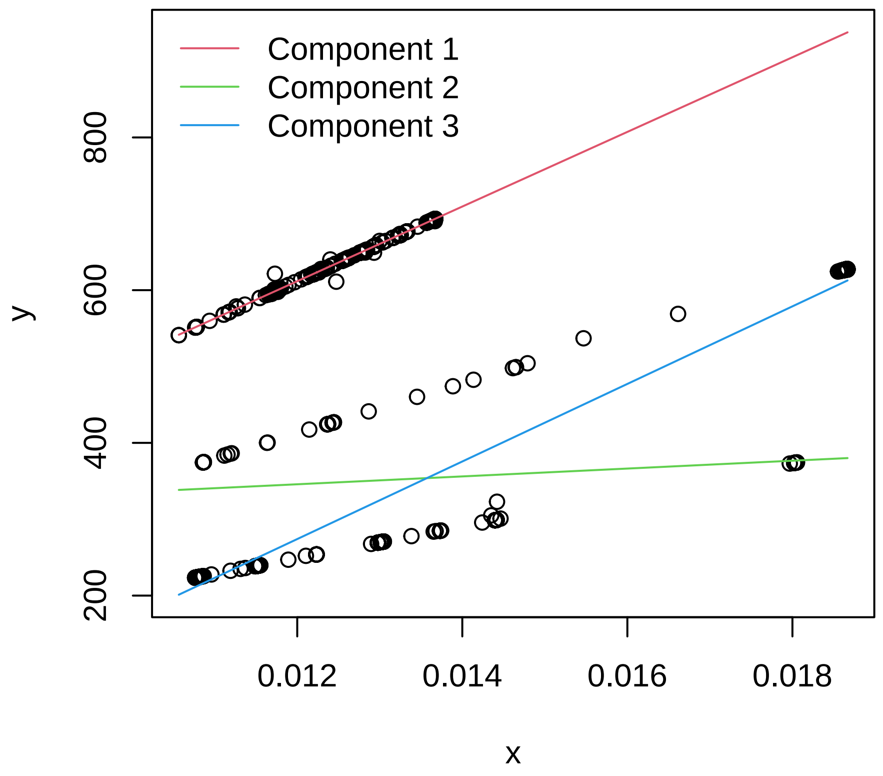

Figure 2: A bad fit to the `kilnAoneOut` data

With a different seed we get a much better looking fit,

```
fitNotSoBad  <- mixreg(y~x,data=kilnAoneOut,verb=FALSE,ncomp=3,
                       seed=43)
plot(fitNotSoBad) # Looks better.
```

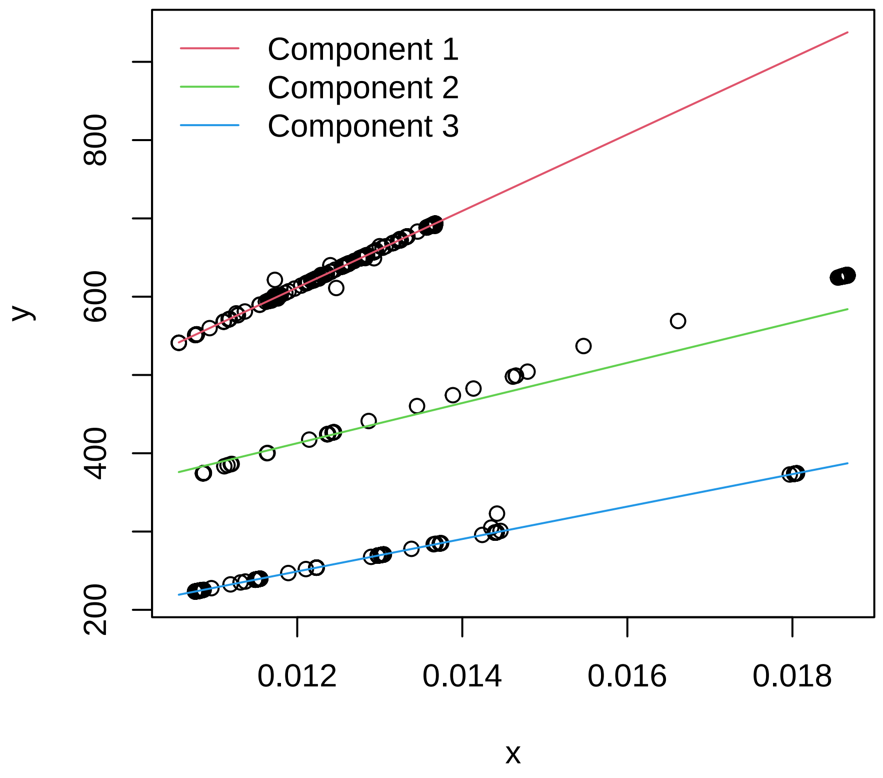

Figure 3: A better (random-start) fit to the `kilnAoneOut` data

The result is far from perfect however; the slope of the second component is too small. One can get a substantial improvement by starting off with a model in which the error variances are taken to be equal:

```
fit1  <- mixreg(y~x,data=kilnAoneOut,verb=FALSE,ncomp=3,
                seed=42,eqVar=TRUE)
fit2  <- mixreg(y~x,data=kilnAoneOut,verb=FALSE,
                thetaStart=fit1$theta)
plot(fit2) # Looks good.
```

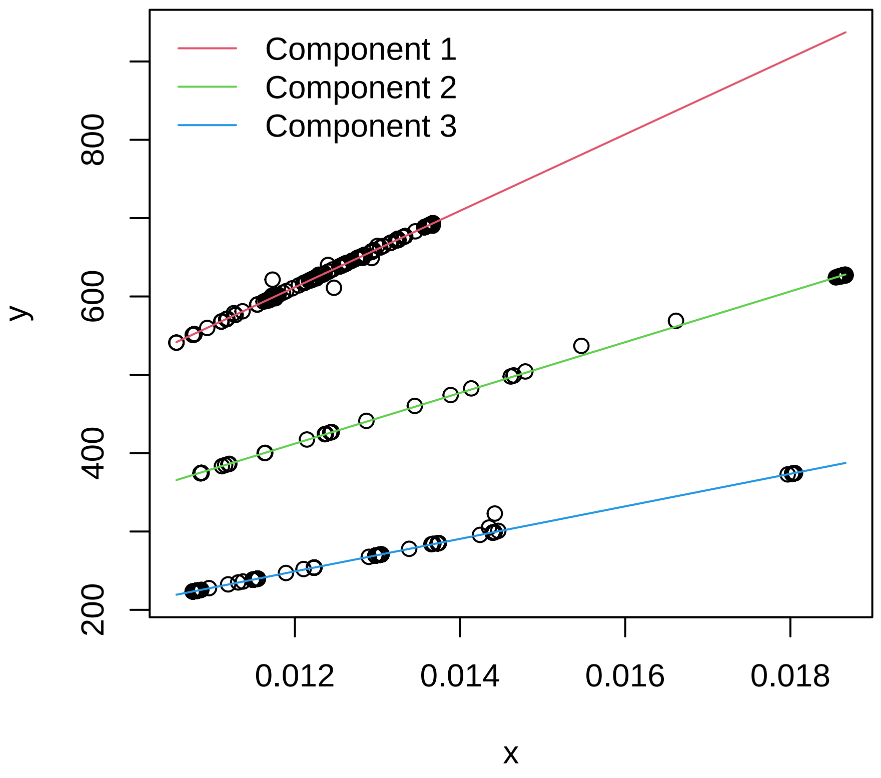

Figure 4: A refit to the `kilnAoneOut` data, using "equal variance" starting values (these having in turn been formed using random starting values).

3.1 A little exploration of the bad fits

When one looks at the plot shown in Figure 3, it seems strange that the second component is skewed off of the line of points that one's eyes tell one that it *should* be tracking. What is going on? The `plot.mixreg()` function has a capability that provides a bit of insight. This plot method can classify the points according to the component with which they are assessed to be associated. The assessment can be done on the basis of either of two criteria: probability or Euclidean distance; which criterion is used is determined by the argument `cMeth` ("classification method"). If this argument is left at its default value `"none"` then no classification is done.

If the points are classified then they are plotted in colours determined by the component to which they are assigned. If we apply the Euclidean distance criterion to the `fitNotSoBad` object, the points are all classified to the components that our eyes tell us are the correct ones.

```r
plot(fitNotSoBad,cMeth="dist",col=c("red","green","blue"))
```

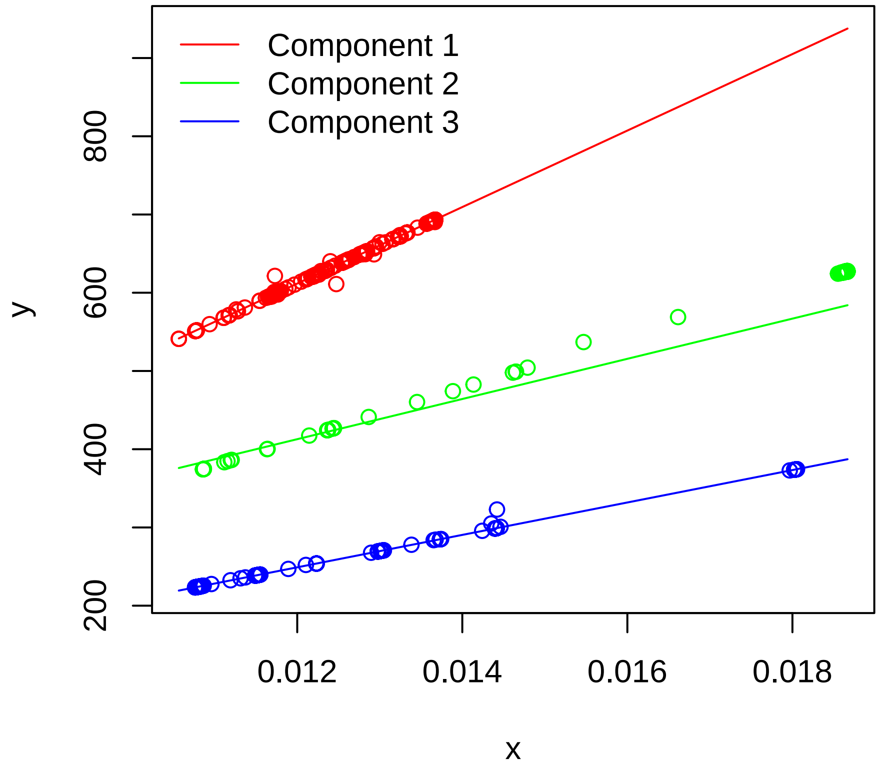

Figure 5: Points classified according to Euclidean distance

However if the probability criterion (which is in effect what the fitting algorithm uses) is applied, the result is very different.

```r
plot(fitNotSoBad,cMeth="prob",col=c("red","green","blue"))
```

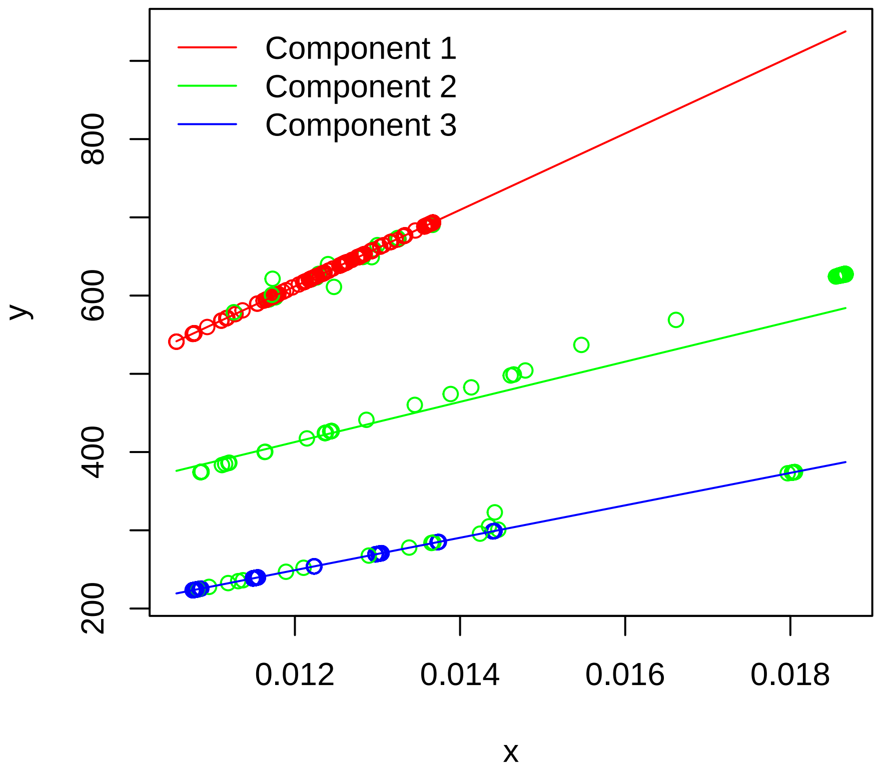

Figure 6: Points classified according to probability

In Figure 6 we see that many of the points that look like they should be associated with components 1 or 3, are coloured green, the colour for component 2. Why is this happening? The explanation is found in the error variances of the components:

```
         sigsq
1 2.535050e-02
2 1.648205e+04
3 1.136377e-05
```

The error variance for component 2 is six orders of magnitude larger than that for component 1 and nine orders of magnitude larger than that for component 3. The corresponding standard deviations are roughly 0.160, 128, and 0.003. The algorithm has found nearly perfect fits to subsets of the points that our eyes tell us are associated with components 1 and 3. It then assumes that points which are not almost exactly on those nearly perfectly fitting lines must belong to the messy component 2. Once points start getting "misclassified" by the fitting algorithm, the "obviously bad" fit becomes stable, and the algorithm gets stuck with that solution. Setting `eqVar=TRUE` prevents component 2 from becoming "messy".

Similar phenomena occur, so I understand, in the context of scalar mixtures. There too, constraining the variances of the components to be equal can often prevent spurious fits from being produced.

3.2 The full `kilnA` data set

The data set `kilnAoneOut` differs from the `kilnAfull` data set only in respect of a single point which was removed from the latter to create the former. That one point, which is highlighted in red in the left hand panel of Figure 7, and is clearly apparent anyway, constitutes in some sense an outlier. The impact of this one outlier is to induce a distortion in the fit of the model.

```
par(mfrow=c(1,2))
plot(kilnAfull,main="Full data set",cex.main=0.9)
with(kilnAfull,points(x[1171],y[1171],pch=20,col="red"))
plot(kilnAoneOut,main="One point omitted",cex.main=0.9)
```

That fit, shown in Figure 8, looks, somewhat mysteriously, like the fit to the `kilnAoneOut` data, obtained with random starting values with `seed=43`. (See Figure 4). We might try the strategy of fitting with `eqVar=TRUE` and then re-fitting with `eqVar=FALSE` and starting values taken from the previous fit.

However in this instance there is no improvement; the plot of the "new" `fitFull` looks exactly the same as the fit of the "old" one. As Old Lodge Skins said in *Little Big Man* "Sometimes the magic works, and sometimes it doesn't." A plot of the "equal variance fit" looks just fine. (As, it must be said, does a plot of the fit produced by `visualFit()`. Perhaps we should just stick with the interactive visual fitting approach and dispense with sophisticated computer-intensive analytic methods?)

The bottom line is that the outlier in `kilnAfull` seems to induce genuine problems and that it is better to use `kilnAoneOut`.

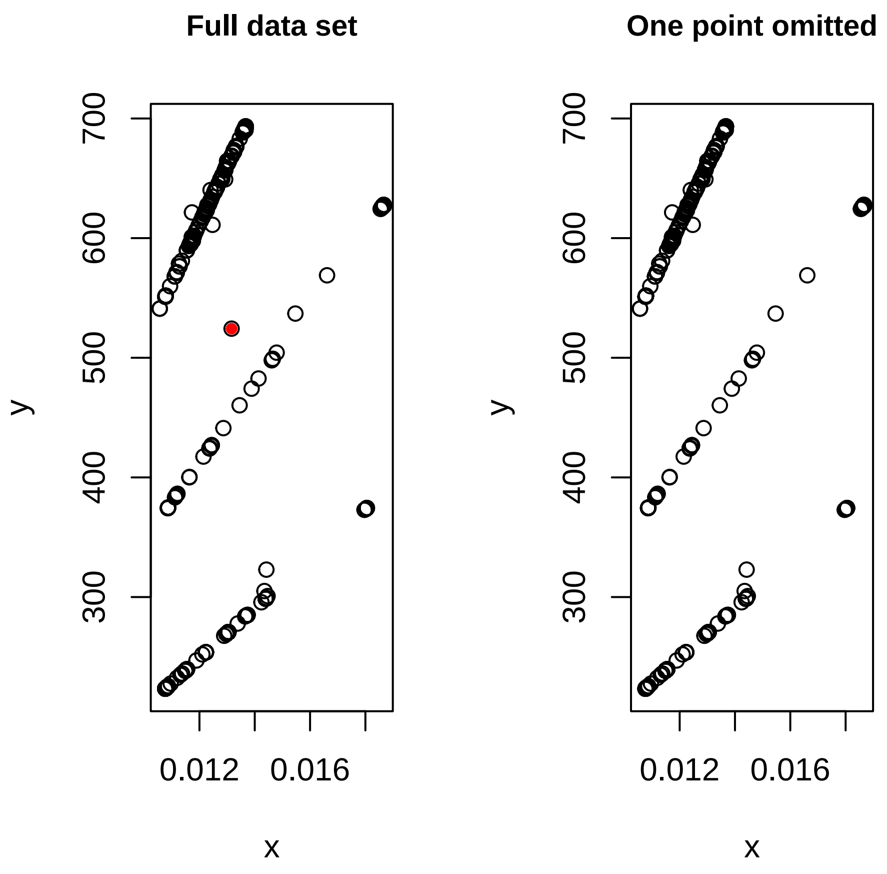

Figure 7: The full `kilnA` data set together with this data set with the outlier omitted.

```
vfitFull <- visualFit(y ~ x, ncomp=3, data=kilnAfull)
fitFull  <- mixreg(y ~ x, data=kilnAfull,
                   thetaStart=vfitFull$theta,
                   verb=FALSE)
plot(fitFull)
```

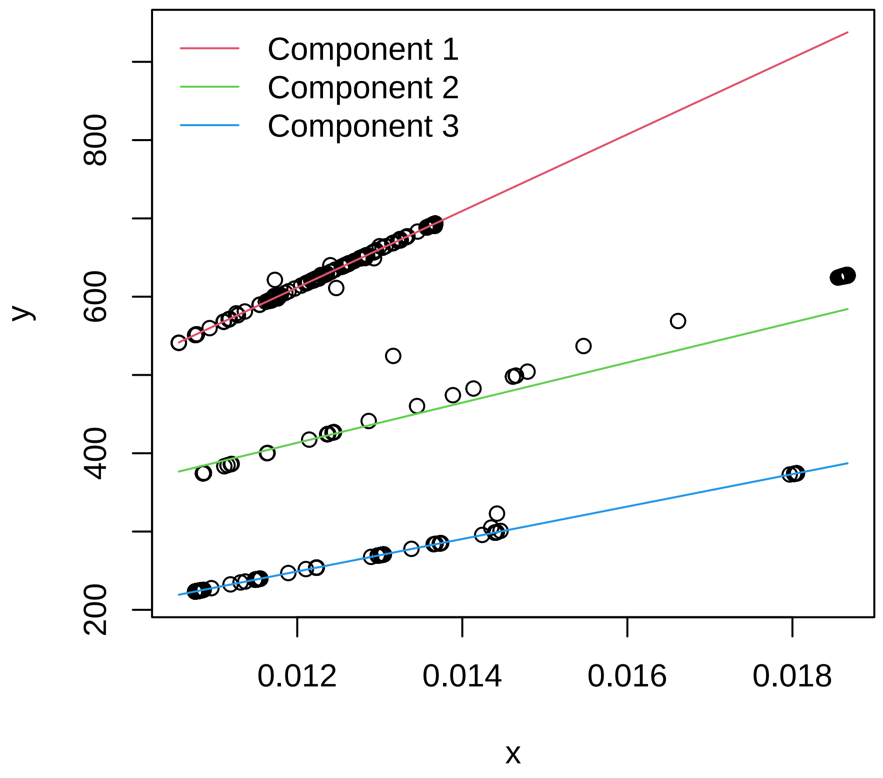

Figure 8: Fit of the full `kilnA` data set.

3.3 Residual plots

It is always a good idea to plot, in some manner, the residuals from the fit of a model. It is not completely clear what such a plot should consist of, in the mixtures of regressions setting, since each data point has a number of residuals associated with it (one for each component). The `mixreg` package solves this dilemma, using an idea due to Adrian Baddeley (*pers. comm.*) according to which each residual is plotted as a point the size of which increases with the probability that the point in question is associated with the component from which that residual is formed. (See Turner 2000 and the package help for the `plot.mixresid()` function for more detail.) This strategy has the effect of diminishing the visual impact of residuals of lesser importance.

Under the procedure described above, the points are plotted as polygons (whose shape is determined by the `shape` argument of the plotting function which is used.) Since there are a fairly large number of points in the `kilnAoneOut` data, the plotting process takes rather a long while. It can be speeded up by setting `shape="none"` which causes the points to be plotted in the usual manner. However the points are then all plotted with symbols of the same size, which in general makes the residual plot hard to interpret.

It turns out that the residual plots from the fit to the `kilnAoneOut` data are rather bizarre looking irrespective of how they are plotted. The fact that the components are so clearly defined and that the error variance is relatively small, causes a plot of the unweighted residuals against the predictor x to take the form of a number of lines that diverge from each other, from left to right. A plot of the unweighted residuals against the fitted values (Figure 9) is even more strange; it appears to roughly take the form of a number of line segments which lie at strange angles to each other.

It is difficult to interpret Figure 9. However the results from the down-weighting strategy are also rather unsatisfactory here. The down-weighted points are very hard to see in a pdf file. (They are more easily discernible on-screen, on my laptop.) The result is that the weighted residual plot (shown in Figure 18, in Appendix I) simply looks, upon cursory inspection, like a row of more-or-less horizontal dots at the zero level, with a large amount of white space above and below (where the invisible or nearly invisible down-weighted points are plotted).

If we trim away the white space, by restricting the y-axis limits, we get a plot (Figure 10) that is a little more perspicuous, with a few apparent outliers being revealed.

```r
rkA <- residuals(fit,std=TRUE)
plot(rkA,vsFit=TRUE,shape="none")
```

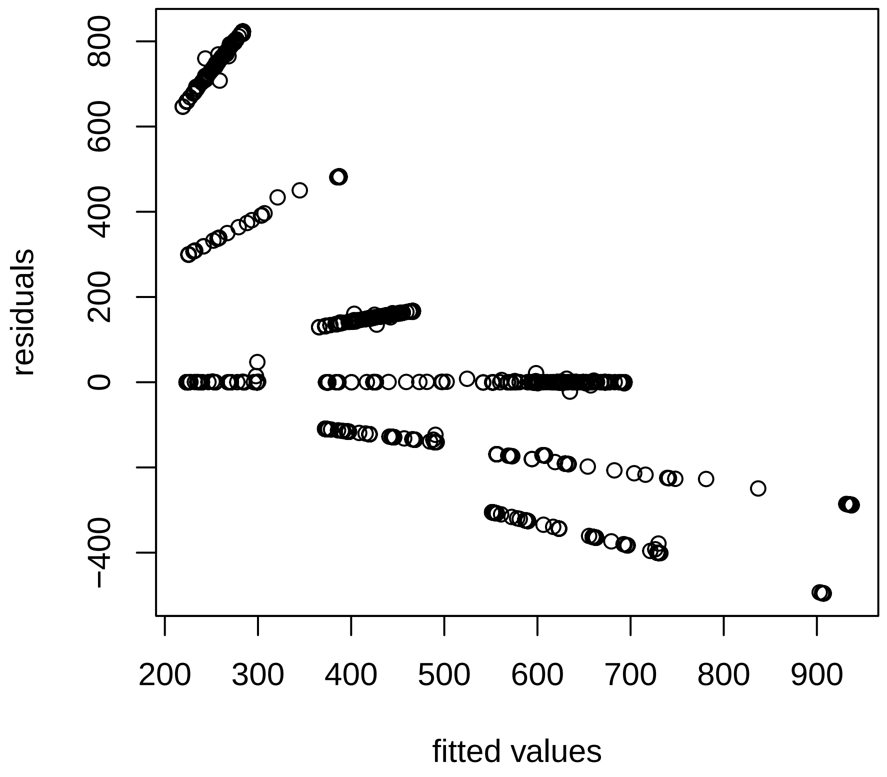

Figure 9: Unweighted residual plot for the `kilnAoneOut` data

```
plot(rkA,vsFit=TRUE,size=0.8,ylim=c(-50,50))
```

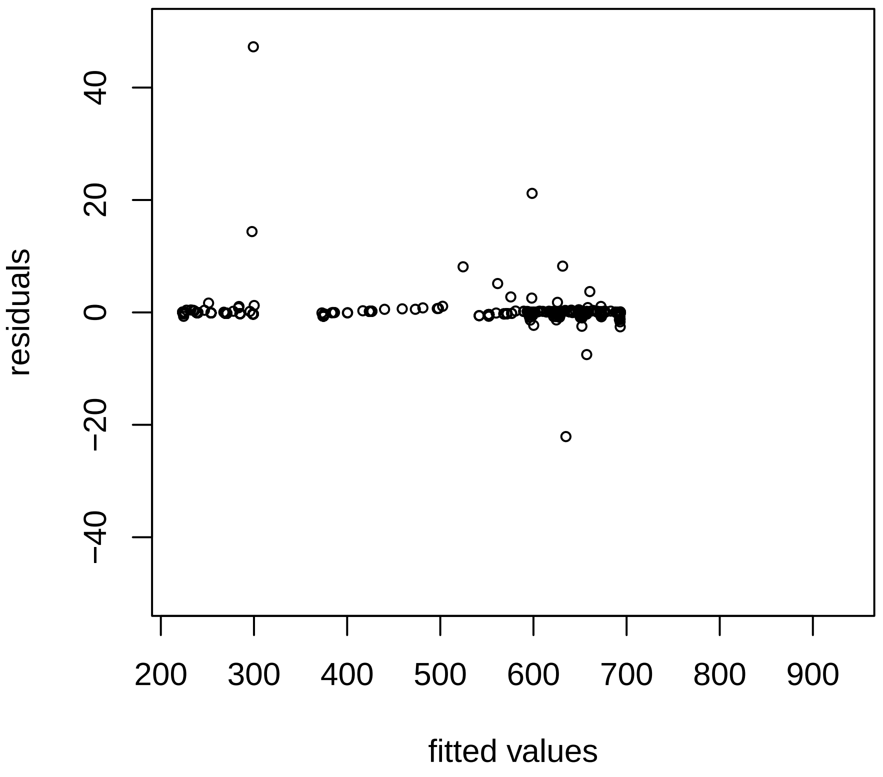

Figure 10: Weighted residual plot for the `kilnAoneOut` data, y limits restricted.

3.4 Zero intercept models

If one plots the fit of the `kilnAoneOut`, with the axes extended to include the origin, one sees that the three regression line all "almost" pass through the origin.

```
plot(fit,xlim=c(0,0.02),ylim=c(0,900))
```

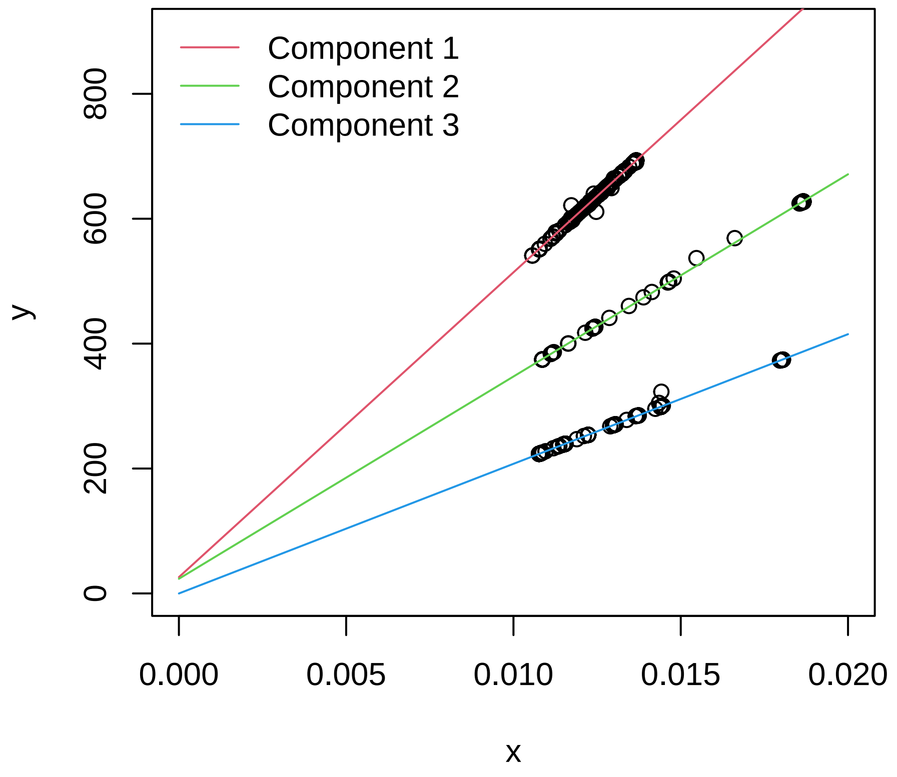

Figure 11: Fit of the `kilnAoneOut` data, showing the origin.

This suggests that a zero intercept model might be appropriate.

```
vfitzi <- visualFit(y ~ x - 1, ncomp=3, data=kilnAoneOut)
fitzi  <- mixreg(y~x-1,data=kilnAoneOut,
                 thetaStart=vfitzi$theta,
                 verb=FALSE,eqVar=TRUE)
plot(fitzi,xlim=c(0,0.02),ylim=c(0,900))
```

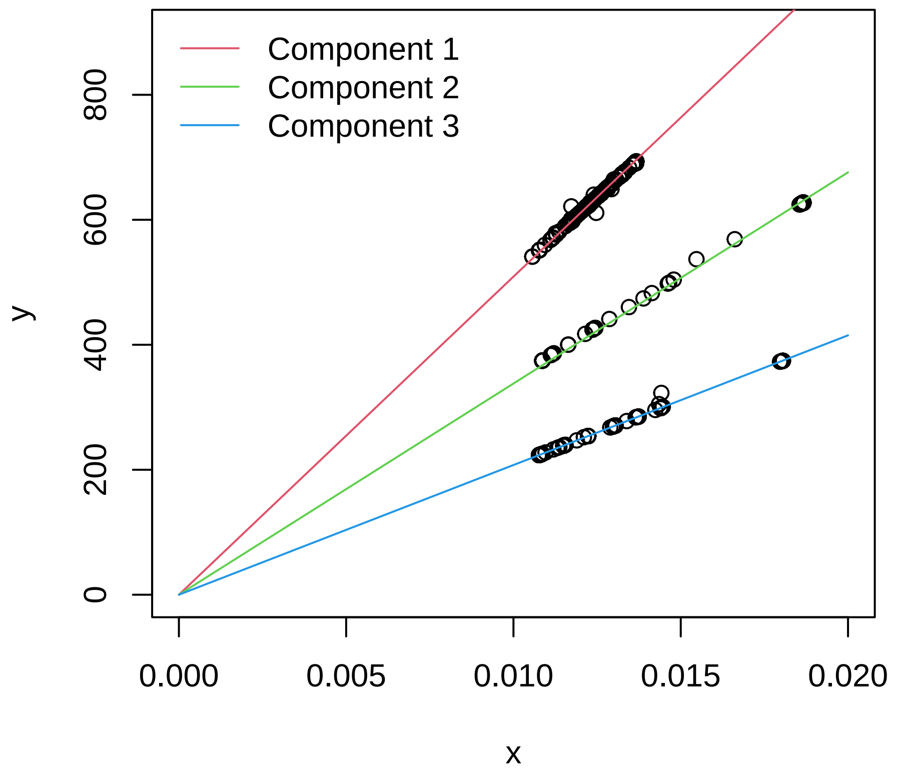

Figure 12: Zero-origin fit to the `kilnAoneOut` data set.

Note that it was necessary to set `eqVar=TRUE` in order to get a decent fit. As in previous examples, unless the error variances are forced to be equal, the fitting process gets the second ("middle") component wrong. Here if we try to refit *without* the `eqVar=TRUE` restriction but using `fitzi$theta` as starting values, the result still gets the second component wrong. Again we find that "Sometimes the magic works, and sometimes it doesn't."

The plot of the zero-origin fit (Figure 12) is visually very similar to the plot of the full fit (Figure 11). However the linear parameter estimates are vastly different, as are the log likelihoods: -9635.981 and -6641.876 respectively. A likelihood ratio test, of the full model against the zero-origin model, of course yields a p-value of 0. A residual plot for the zero-origin fit (Figure 13) does not look materially different from the corresponding plot for the full fit (Figure 10).

```r
rkAzi<-residuals(fitzi,std=TRUE)
plot(rkAzi,vsFit=TRUE,size=0.8,ylim=c(-50,50))
```

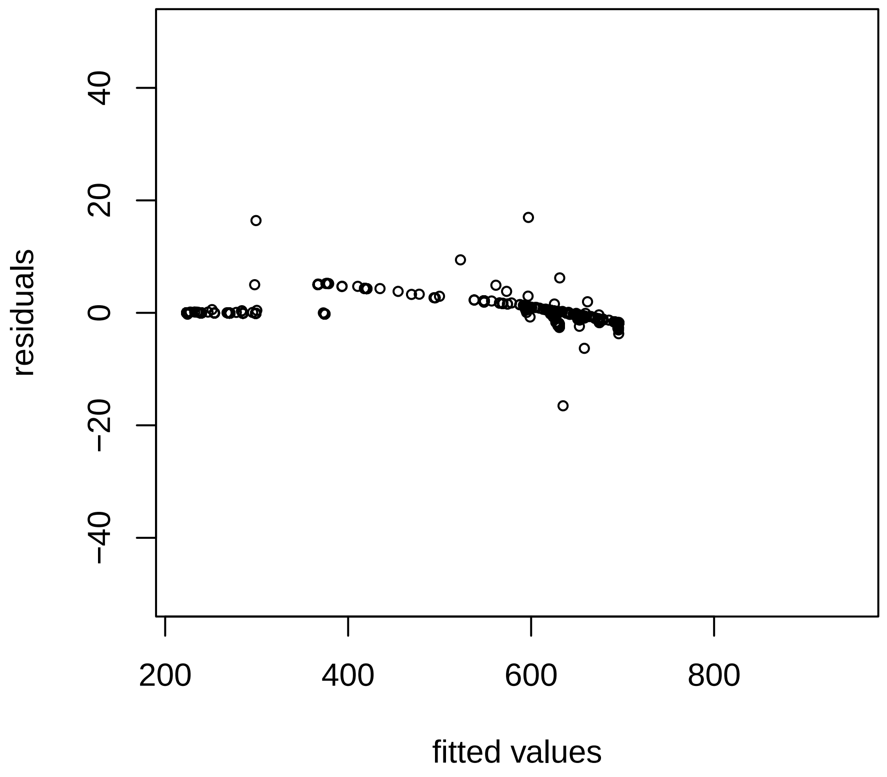

Figure 13: Weighted residual plot for the `kilnAoneOut` data, zero-origin model, y limits restricted.

4 Revisiting the aphids data

These data were thoroughly discussed in Turner (2000) so I shall not say very much here. These data look a lot more “messy” than the kiln data; the components are not nearly as clearly defined. Nevertheless they appear to be much more tractable in respect of the fitting process and random starting values (almost?) always seem to work without difficulty. (The fact that the error variances are relatively large might actually be an advantage; see the discussion in Section 5.) Four different fits (using four different seeds for the random number generator) yielded results in which the parameter estimates differed from each other by a maximum of 2.56×10^{-7}.

Since `visualFit()` was not available when Turner (2000) was written, it is of some interest to look at the application of this function to the aphids data. Figure 14 shows a plot of the visually obtained fit which may be compared with the analytic fit, shown in Figure 15. The latter was obtained using the visually obtained fit to provide starting values. The plots of the visual and analytic fits are effectively indistinguishable. However it should be noted that the AIC from the analytic fit is 270.1302 which is appreciably smaller than that from the visual fit, 279.1757.

4.1 Residual plots for the aphids data

It is interesting to compare the unweighted and weighted residual plots for the aphids data. We see (Figure 16) that the plot of the unweighted residuals gives an impression that the error variance increases as `aphRel` increases, whereas the plot of the weighted residuals (Figure 17) gives no such impression.

```r
vfitAph <- visualFit(plntsInf ~ aphRel,ncomp=2,data=aphids)
afitAph <- mixreg(plntsInf ~ aphRel,ncomp=2,data=aphids,
                  verb=FALSE,thetaStart=vfitAph$theta)
plot(vfitAph,main="Visual fit")
plot(afitAph,main="Analytic fit")
rAph <- residuals(afitAph,std=TRUE)
plot(rAph,shape="n")
plot(rAph)
```

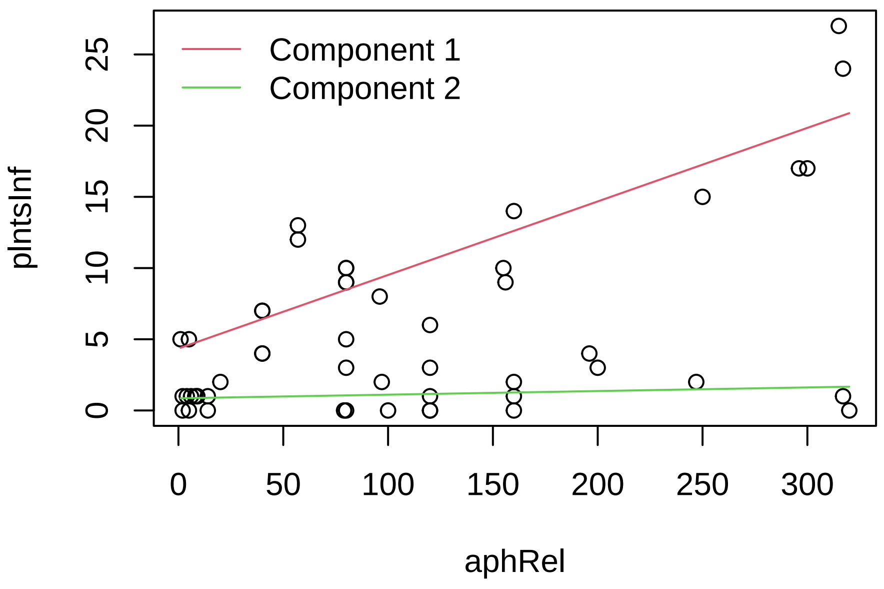

Figure 14: Visual fit for the aphids data.

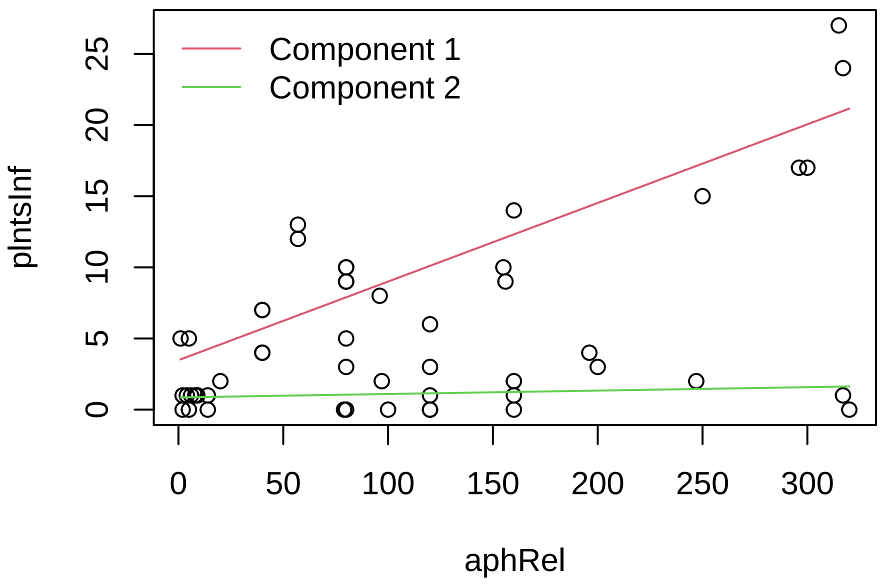

Figure 15: Analytic fit for the aphids data.

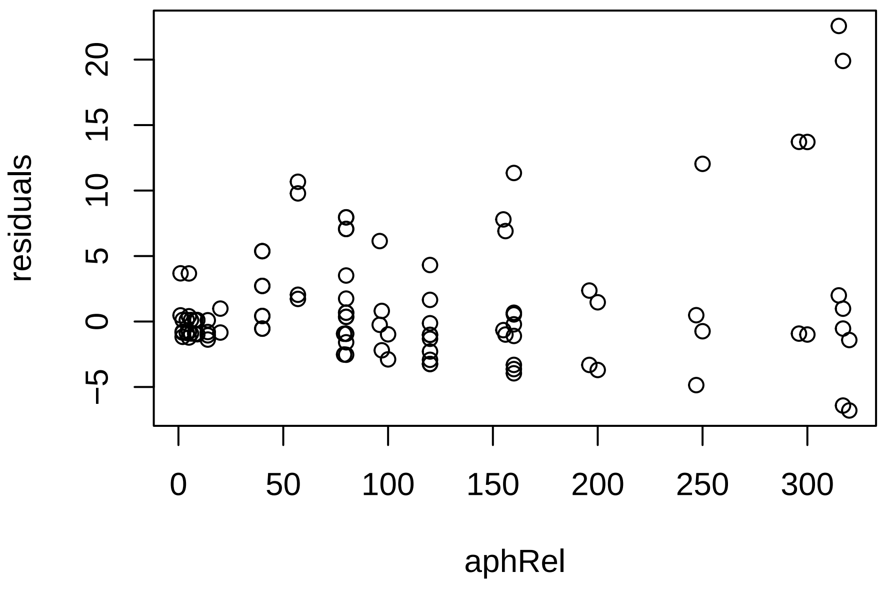

Figure 16: Unweighted residual plots for the aphids data.

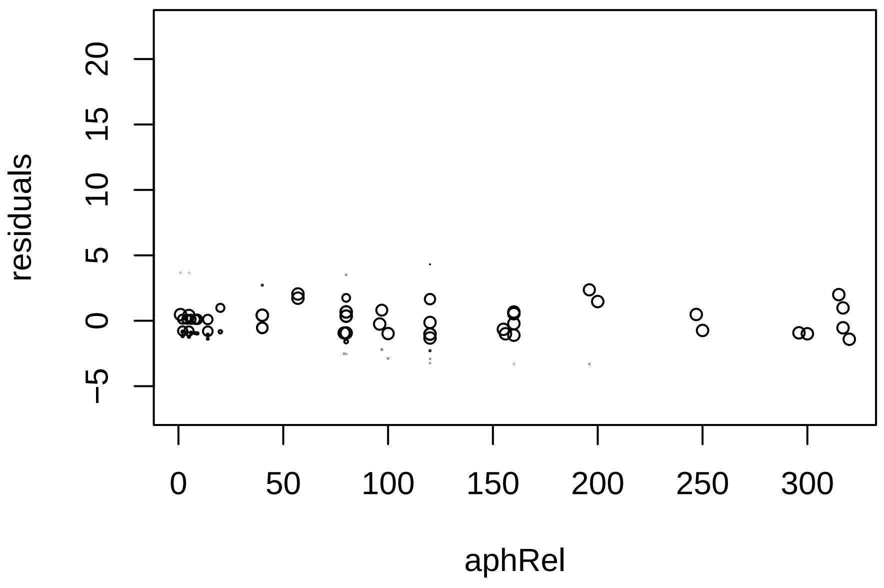

Figure 17: Weighted residual plots for the aphids data.

5 Concluding remarks

I am confident that the `mixreg` package provides a satisfactory collection of tools for analysing mixtures of regressions. However it must be emphasised that it is a collection of *tools* and that it is up to the user to apply the tools correctly. The functions in the `mixreg` package should not be used blindly nor should all results be taken at face value. Mixtures are tricky concepts, even in the scalar context, and are probably even trickier in the regression context. It is important that you think about what you are doing and at the very least examine the results visually (using the `plot.mixreg()`, `plot.cband()` and `plot.mixresid()` plot methods, and possibly `qqMix()`).

At this point I have a confession to make: I am *not* an expert on mixtures of regressions. I have successfully fitted a mixtures of regressions model in the context of an interesting application (Turner 2000). I have written what is, in my humble opinion, a damned good package for analysing mixtures of regressions. However I am not qualified to give more than the most basic advice on what to watch out for when fitting mixtures of regressions, how to detect lurking problems or how such problems might be revealed by graphical means. All I can really say is "Be on your guard" (or as was said in *Carmen* and as I might have said when I lived in Canada,"Prends gard à toi.") The details of how to maintain effective vigilance are beyond me.

As I said above, mixtures can be tricky even in the scalar context. Judicious reading of the literature on mixture models might provide some insight into what to look out for. The classic work by McLachlan and Peel (2000) is the obvious place to start. It may also be worth the user's while to look into the handbook Mengersen et al. (2011).

One point worthy of note is that variance considerations can be subtle and are often worthy of attention. Petr Pikal, who provided the kiln data sets, has observed (*pers. comm.*) that in respect of the kiln data, the fact that the error variance is relatively *small* appears to be problematic. He reports getting more reliable results by jittering the data so as to increase the error variance.

It should also be borne in mind that forcing the components all to have the same error variance can have a stabilising effect and improve the fit, at least in some settings. This was seen in the examples, involving the kiln data, given above. If the user does not want to fit a model in which the error variances are equal, a possible approach is to fit a model with equal variances and then re-fit, without the equal variance constraint, using the parameter estimates from the previous fit as starting values. As we have seen above this approach sometimes works but does not always give satisfactory results.

In closing I would like to draw the reader's attention to the paper, Tsai (2019), in which it is shown that for small samples (including samples of the size of the `aphids` data, i.e. 51) the covariance matrices for the parameter estimates, as produced by the methods used in the `mixreg` package, underestimate the true variability of the estimates. The consequence is that confidence intervals for the parameters are too narrow whence the empirical coverage rate is lower than the specified theoretical coverage rate. The `covMat()` function in the `mixreg` package calculates the covariance matrix as the inverse of the observed Fisher information matrix. This calculation is not straightforward inasmuch as parameter estimation is effected by means of the EM algorithm. The `covMat()` function employs the technique from Louis (1982) to carry out the required calculation.

Surprisingly (to me) the use of Monte Carlo methods (as effected, e.g., by the `covMatMC()` function in the `mixreg` package) is of no help here. Covariance matrices produced by Monte Carlo methods underestimate the variability of the parameter estimates to the same extent as do those produced by inverting the observed Fisher information. Tsai solves the problem by using *fiducial* inference based on "generalized pivotal quantities". (See Tsai 2019 for references.) The technique appears to produce satisfactory results, but the underlying theory is difficult and the implementation (which requires the use of a Markov Chain Monte Carlo procedure) is computationally expensive. It would be desirable to find a less demanding technique for calculating variance estimates that is accurate for small samples.

Acknowledgements

I would like to express my appreciation to Peter Macdonald (Emeritus Professor, McMaster University) for a useful conversation about the subtleties of dealing with mixtures.

I would also like to express my thanks to Duncan Murdoch (Emeritus Professor, Western University) for pointing out to me a problem induced by the approach that I had been using in trying to accommodate both the "old" and "new" syntax for the `mixreg()` function. Duncan suggested the approach that is now used to accommodate both syntaxes.

References

T. A. Louis. Finding the observed information matrix when using the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 44:226 – 233, 1982.

Peter Macdonald. *mixdist: Finite Mixture Distribution Models*, 2018. URL `https://CRAN.R-project.org/package=mixdist`. R package version 0.5-5; contributions from Juan Du.

Geoffrey J. McLachlan and David Peel. *Finite Mixture Models*. John Wiley & Sons, Hoboken, NJ, 2000.

Kerrie L. Mengersen, Christian P. Robert, and D. Michael Titterington, editors. *Mixtures: Estimation and Applications*. John Wiley & Sons, Chichester, 2011. Wiley Series in Probability and Statistics.

Shin-Fu Tsai. Comparing coefficients across subpopulations in Gaussian mixture regression models. *Journal of Agricultural, Biological, and Environmental Statistics*, 24(4):610 – 633, 2019.

T. Rolf Turner. Estimating the rate of spread of a viral infection of potato plants via mixtures of regressions. *Applied Statistics, Part 3*, 49:371 – 384, 2000.

Appendix I

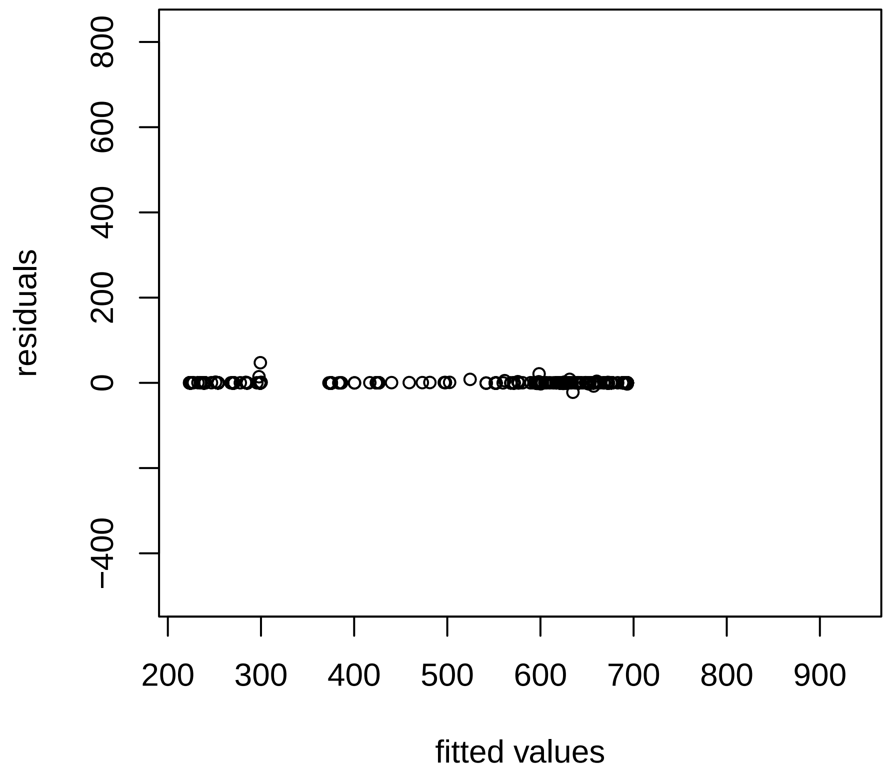

Figure 18: Weighted residual plot for the `kilnAoneOut` data

Appendix II

The estimates of the error variances produced by `mixreg()` merit some discussion. The estimates of the linear parameters for the kth component are taken directly from the output of `lm()` applied to the kth component with weights equal to $\{\gamma_{ik}\}_{i=1}^n$ where γ_{ij} is the probability, given the observations, that the ith observation was generated by component j. The estimates of the error variances are not taken from the output of `lm()` but rather are calculated as

$$\hat{\sigma}_k^2 = \frac{1}{\sum_{i=1}^n \gamma_{ik}} \sum_{i=1}^n (y_i - \boldsymbol{x}_i^\top \hat{\boldsymbol{\beta}}_k)^2 \gamma_{ik} \tag{1}$$

The model assumed by `lm()` is that the ith weight is equal to $1/\tau_i^2$ where the error variance associated with the ith observation is $\tau_i^2\tau^2$. That is, the weights are so-called "precision" weights. When `lm()` is applied to the kth component, the role of τ^2 is played by σ_k^2 and $1/\tau_i^2$ is equal to γ_{ik}.

Under the precision weights model, the estimate of τ^2 is

$$\hat{\tau}^2 = \frac{1}{n-p} \sum_{i=1}^n \frac{(y_i - \boldsymbol{x}_i^\top \hat{\boldsymbol{\beta}}_k)^2}{\tau_i^2} = \frac{1}{n-p} \sum_{i=1}^n \gamma_{ik}(y_i - \boldsymbol{x}_i^\top \hat{\boldsymbol{\beta}}_k)^2 \tag{2}$$

where p is the number of linear parameters, and this estimate is unbiased for τ^2.

However in the mixtures of regression context the precision weights model makes no sense. The weights are more of the nature of sampling weights, and the appropriate estimate of σ_k^2 is given by (1) and not by (2). Note that the estimates of the linear coefficients under the precision weights model *are* the appropriate estimates. They are exactly the same as what is obtained from the mixtures of regressions model.

The question then arises: Is the estimate of σ_k^2 given by (1) unbiased, and in any case, what is the expected value of this estimate? Answering this latter question, to quote Bill Venables (`fortunes::fortune(211)`) "looks difficult, but on closer inspection turns out to be impossible." Part of the reason for the impossibility is that the probabilities γ_{ik} that appear in (1) are *conditional* upon the observations. Calculating expected values conditionally (even in part) upon the observations, makes no sense. To obtain an expected value of $\hat{\sigma}_k^2$ one would have to somehow re-express (1) in such a way that it did not depend upon the γ_{ik} and this is not feasible.

In order to investigate a little bit further, I did a perfunctory simulation experiment:

```
library(mixreg)
```

```
Nsim <- 1000
theta <- vector("list",3)
theta[[1]] <- list(beta=c(2,3),sigsq=4,lambda=0.5)
theta[[2]] <- list(beta=c(1,2),sigsq=16/9,lambda=0.3)
theta[[3]] <- list(beta=c(0,1),sigsq=1,lambda=0.2)
x <- (1:100)/10
rslt <- matrix(nrow=Nsim,ncol=3)
set.seed(42)
for(i in 1:Nsim) {
    simdat <- rmixreg(x,theta)
    fit    <- mixreg(y~x,ncomp=3,data=simdat,
                     thetaStart=theta,verb=FALSE)
    rslt[i,] <- fit$parmat[,"sigsq"]
    cat(i,"")
    if(i%%10 == 0) cat("\n")
}
if(i%%10 != 0) cat("\n")
```

Applying Hotelling's T^2 test to `rslt`, with $\mu = (4, 16/9, 1)^\top$, produced a p-value of 0, so it can be said incontrovertibly that the estimates are indeed biased. In this instance the mean of the estimates (given by `apply(rslt,2,mean)` is `c(3.8895,1.5723,0.8855)` to four decimal places, so (in this instance) the estimates appear to be biased low.