Homework Assignment 5

36-350, Data Mining

Solutions

1. (a) *Show that*

$$f_k(x_k) = \mathbf{E}\left[Y - \alpha - \sum_{j \neq k} f_j(X_j) \middle| X_k = x_k \right]$$

i.e., the function f_k is the conditional expectation of the partial residuals. Hint: *Use smoothing, a.k.a. the law of total expectation.*

Answer: The law of total expectation says that, for any three random variables X, Y and Z, $\mathbf{E}[Y|X=x] = \mathbf{E}[\mathbf{E}[Y|Z, X=x]|X=x]$. Applied here,

$$\mathbf{E}[Y|X_k = x_k] = \mathbf{E}[\mathbf{E}[Y|X_1 = x_1, X_2 = x_2, \ldots X_p = x_p]|X_k = x_k] \tag{1}$$

Under the additive model, we know what the inner conditional expectation is,

$$\mathbf{E}[Y|X_1 = x_1, X_2 = x_2, \ldots X_p = x_p] = \alpha + \sum_{j=1}^{p} f_j(x_j) \tag{2}$$

So

$$\mathbf{E}[Y|X_k = x_k] = \mathbf{E}\left[\alpha + \sum_{j=1}^{p} f_j(x_j) \middle| X_k = x_k \right] \tag{3}$$

Therefore

$$\begin{aligned}
&\mathbf{E}\left[Y - \alpha - \sum_{j \neq k} f_j(X_j) \middle| X_k = x_k \right] \\
&= \mathbf{E}\left[\alpha + \sum_{j=1}^{p} f_j(x_j) - \alpha - \sum_{j \neq k} f_j(X_j) \middle| X_k = x_k \right] && (4)\\
&= \mathbf{E}[f_k(X_k)|X_k = x_k] && (5)\\
&= f(x_k) && (6)
\end{aligned}$$

QED.

(b) *Write a function to implement the following* **back-fitting** *procedure for estimating additive models.*

ANSWER: As mentioned in class, one can do this either in a simple way, estimating only the values of the functions at the sample points, or in a more sophisticated way estimating the whole functions.

Here is the simple way. It encapsulates the actual estimation in another function, `centered.smoothing`, which we'll write later. (For fully-commented code, see `solutions-05.R` online.)

```
addm.points = function(y,x,tolerance=sd(y)/sqrt(length(y)),max.passes=10) {
  alpha = mean(y)
  p = ncol(x)
  n = nrow(x)
  f = matrix(0,nrow=n,ncol=p)
  y.pred = rep(alpha,n) # Initially, we always predict the mean

  max.iterations = max.passes*p
  made.changes = TRUE
  iterations = 0
  j = 0
  while(made.changes & (iterations < max.iterations)) {
    iterations = iterations + 1
    j = ifelse(j == p,1,j+1)
    partial.residuals = y - (y.pred - f[,j])
    new.fj = centered.smoothing(x[,j],partial.residuals)
    new.y.pred = y.pred + (new.fj - f[,j])
    f[,j] = new.fj
    number.changes = sum(abs(new.y.pred - y.pred) > tolerance)
    made.changes = (number.changes > 0)
    y.pred = new.y.pred
  }
  if (made.changes) {
    warning("Exited addm.points before backfitting converged")
  }
  return(list(mean.response=alpha,partial.responses=f,
         fitted=alpha+apply(f,1,sum),x=x,y=y,
         iterations=iterations,converged=!made.changes,tolerance=tolerance,
         max.iterations=max.iterations))
}
```

We still need to do the actual smoothing! We'll use `ksmooth`. The one trick is that it returns its predictions in order of increasing x, for plotting, so we need to re-order them. This is a job for yet another function.

```
centered.smoothing = function(x,y) {
  g = ksmooth(x,y,kernel="normal",n.points=length(x),x.points=x)$y
  unsort.x = inverse.permutation(order(x))
  g = g[unsort.x] # puts g back into data order
  return(g - mean(g))
}

inverse.permutation = function(forward.permutation) {
  # match() returns the indices of the first occurrences of its first argument
  # its second argument.  See solutions-05.R for a (slower) approach which
  # doesn't use this.
  inv.perm = match(1:length(forward.permutation),forward.permutation)
  return(inv.perm)
}
```

Let's check that this works:

```
> z = round(rnorm(10),2)
> z
 [1]  0.37  0.14  0.81 -0.52  0.28  0.48  0.03 -1.91  0.73  0.17
> z[order(z)]
 [1] -1.91 -0.52  0.03  0.14  0.17  0.28  0.37  0.48  0.73  0.81
> inverse.permutation(order(z))
 [1]  7  4 10  2  6  8  3  1  9  5
> sort(z)[inverse.permutation(order(z))]
 [1]  0.37  0.14  0.81 -0.52  0.28  0.48  0.03 -1.91  0.73  0.17
```

(I stick in the initial `round()` for brevity.) So, `inverse.permutation` seems to work. We can also check whether `centered.smoothing` is behaving sensibly:

```
> z2 = z^2 - mean(z^2)
> round(z2,2)
 [1] -0.43 -0.54  0.09 -0.29 -0.48 -0.33 -0.56  3.09 -0.02 -0.53
> z2.ksmooth = ksmooth(z,z2,kernel="normal",n.points=length(z),x.points=z)$y
> round(z2.ksmooth-mean(z2.ksmooth),2)
 [1]  3.10 -0.28 -0.52 -0.50 -0.50 -0.46 -0.43 -0.35 -0.06 -0.01
> round(centered.smoothing(z,z2),2)
 [1] -0.43 -0.50 -0.01 -0.28 -0.46 -0.35 -0.52  3.10 -0.06 -0.50
```

Here `z2` is a centered version of z^2, and we see that just using `ksmooth` messes up the order, while `centered.smoothing` does not.

Alternately, we could just use a smoothing function with a prediction method:

```
centered.smoothing = function(x,y) {
  g = predict(loess(y ~ x))
  return(g - mean(g))
}
```

and then we wouldn't have to worry about the order.

We still should check that the over-all function works. To make things reasonably challenging, but still plottable, let $Y|X_1 = x_1, X_2 = x_2$ will be Gaussian, with a mean of $x_1^2 + 3\tanh x_2$ and a standard deviation of 0.1. The inputs X_1 and X_2 themselves will be uncorrelated standard Gaussians.

```
x = matrix(rnorm(100),ncol=2)
y = x[,1]^2 + 3*tanh(x[,2]) + rnorm(50,0,0.1)
fit = addm.points(y,x)
par(mfrow= c(2,1))
plot(x[,1],fit$partial.response[,1],
     xlab=expression(x[1]),ylab=expression(hat(f)[1]))
curve(x^2-1,add=TRUE)
plot(x[,2],fit$partial.response[,2],
     xlab=expression(x[2]),ylab=expression(hat(f)[2]))
curve(3*tanh(x) - mean(3*tanh(rnorm(1e4))), add=TRUE)
```

The last commands plot the estimated partial response functions against the true partial response functions. (Remember our convention that each of these has mean zero. The mean of x^2 when $x \sim \mathcal{N}(0,1)$ is easy to calculate, and while the mean of $3\tanh x$ is not so easy to calculate, we can get it by simulating.) Figure 1 shows the results.

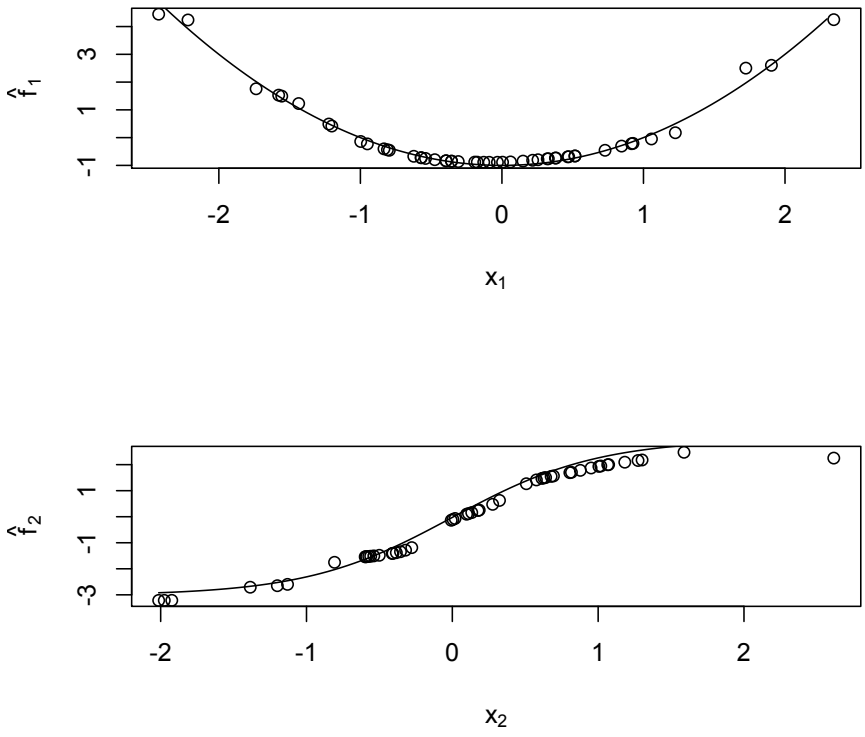


Figure 1: True partial response functions (curves) and estimated partial response functions (dots) from the model $Y = X_1^2 + 3 \tanh X_2 + \epsilon$, $\epsilon \sim \mathcal{N}(0, 0.01)$. Estimation was done with the `addm.simple` function with default settings. In both cases the true partial-response curves have been centered to have mean zero.

Now for the subtler solution, which, rather than using just estimating values at particular points, estimates whole functions. We can't do this so easily with `ksmooth`, because it doesn't return the right kind of object, but we can do it with `loess`. (There are several packages, e.g., `lokern` and `np`, which do kernel smoothing *and* return function objects, and in Problem 2b we'll see how to fake it even with ksmooth.) This involves more advanced R programming. In particular we'll want our function to return a new *type* of object, which I'll call `addm`. It will contain a list of the partial response functions, a mean response, and a matrix of input values. We may also want to add other elements to the list.

First, define a function which constructs an object of this type.

```
make.addm = function(partial.responses,mean.response,x,...) {
  my.addm = list(partial.responses=partial.responses,
                 mean.response=mean.response,x=x,...)
  class(my.addm) = "addm"
  return(my.addm)
}
```

Now we'll define a predictor function, which is what we ultimately want.

```
predict.addm = function(am,newdata=am$x) {
  p = ncol(newdata)
  n = nrow(newdata)
  alpha = am$mean.response
  f = am$partial.responses
  f.matrix = matrix(0,nrow=n,ncol=p)
  for (j in (1:p)) {
    f.matrix[,j] = f[[j]](newdata[,j])
  }
  y.pred = alpha + apply(f.matrix,1,sum)
  return(y.pred)
}
```

Now if we use `make.addm` to create an additive model object, say `am`, if we call `predict(am,newdata=x)`, R will know to look for a function named `predict.addm`. Since it finds what we just wrote, it will use it to calculate predictions on `x`. This is just like making predictions with a linear model.

Now for the actual estimator.[1]

[1] Because the two estimator functions are so very similar, it would be better coding practice to "re-factor" the programs, by extracting the common parts, and putting in a switch which controlled whether to use a list of functions or a matrix of function values. This sort of good coding saves work in the long run, but requires more initial set-up, so I'm not going to do it here.

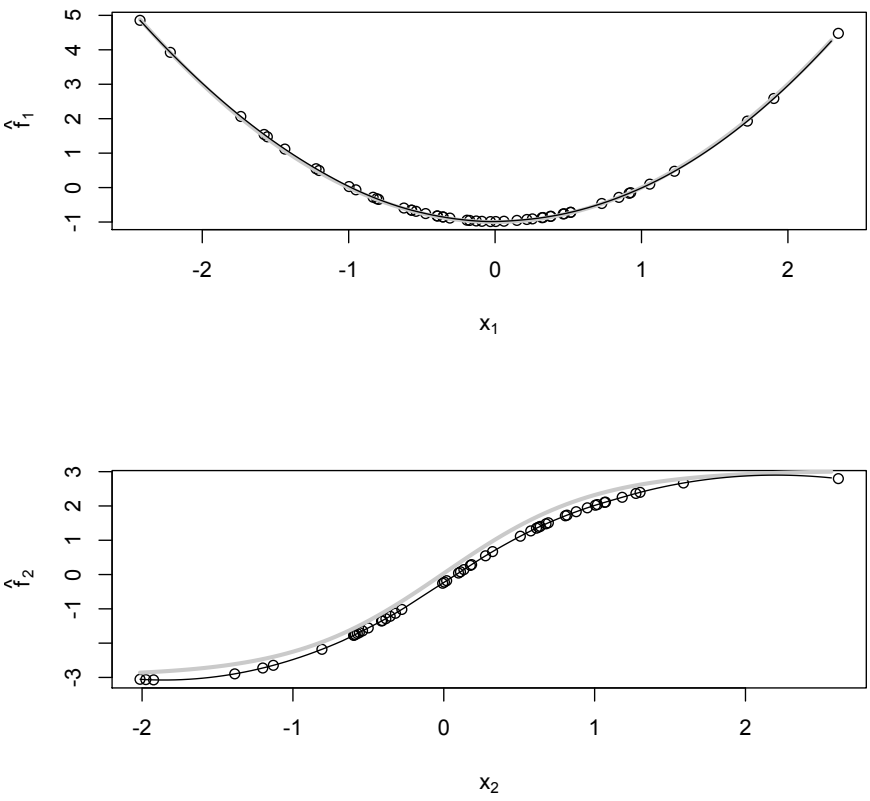
```
addm.functions = function(y,x,tolerance=sd(y)/sqrt(length(y)),max.passes=10) {
  alpha = mean(y)
  p = ncol(x)
  n = nrow(x)

  constant.0.fn = function(x) {return(rep(0,length(x)))}
  f = list(constant.0.fn)
  for (j in (2:p)) { f[[j]] = constant.0.fn }
  y.pred = rep(alpha,n)

  max.iterations = max.passes*p
  made.changes = TRUE
  iterations = 0
  j = 0
  while(made.changes & (iterations < max.iterations)) {
    iterations = iterations + 1
    j = ifelse(j==p,1,j+1)
    input.var = x[,j]
    partial.residuals = y - (y.pred - f[[j]](input.var))
    new.fj = centered.smoothing.function(input.var,partial.residuals)
    new.y.pred = y.pred + (new.fj(input.var) - f[[j]](input.var))
    number.changes = sum(abs(new.y.pred - y.pred) > tolerance)
    made.changes = (number.changes > 0)
    f[[j]] = new.fj
    y.pred = new.y.pred
  }
  if (made.changes) {
    warning("Exited addm.functions before backfitting converged")
  }
  return(make.addm(partial.responses=f,mean.response=alpha,x=x,
                   fitted=y.pred,y=y,
                   iterations=iterations,tolerance=tolerance,
                   max.iterations=max.iterations, converged=!made.changes))
}

centered.smoothing.function = function(x,y) {
  g = loess(y ~ x, surface="direct",degree=1)
  centering.constant = mean(predict(g))
  return(function(u) { predict(g,newdata=u) - centering.constant } )
}
```

Applied to the same data, we get the results in Figure 2.




```
par(mfrow=c(2,1))
fit.addm = addm.functions(y,x,max.iterations=5)
plot(x[,1],fit.addm$partial.responses[[1]](x[,1]),
     xlab=expression(x[1]),ylab=expression(hat(f)[1]))
curve(x^2-1,col="grey",add=TRUE,lwd=3)
curve(fit.addm$partial.responses[[1]](x),add=TRUE)
plot(x[,2],fit.addm$partial.responses[[2]](x[,2]),
     xlab=expression(x[2]),ylab=expression(hat(f)[2]))
curve(3*tanh(x)-mean(3*tanh(rnorm(1e4))),col="grey",add=TRUE,lwd=3)
curve(fit.addm$partial.responses[[2]](x),add=TRUE)
```

Figure 2: Estimated (black) and true (grey) partial response functions. Notice that the black curves are the actual estimated functions.

(c) *Explain how you would modify your code to choose the degree of smoothing by cross-validation. Remember that each function f_j might be more or less smooth than the others, so it needs its own bandwidth.*

ANSWER: We need to define a grid of possible bandwidths for each function. A reasonable choice would be to take say ((1:10)/10) times the range, or inter-quartile range, of each input variable. One could then evaluate the performance of the different bandwidths either inside or outside the back-fitting loop. (Either approach was acceptable for an answer.)

If done inside the back-fitting loop, one would first calculate the partial residuals for each j, and then do cross-validation among the partial residuals to select a bandwidth for estimating f_j. A cruder but faster version would select the bandwidth for each function once, in the first back-fitting iteration, and then hold it fixed. A more refined but slower version would re-do the CV in each pass. Either way, we get a bandwidth for each $\widehat{f_j}$, but it's the best bandwidth for that function *given* our current estimates of the other functions.

Done outside the backfitting loop, we'd need divide the data into training and testing sets, pick a bandwidth for each function, go through back-fitting with those bandwidths, and then evaluate predictions on the testing set. Repeated over multiple combinations of bandwidths and multiple divisions into training and testing sets, we'd see which *collection* of bandwidths worked best. This would generally be more accurate than picking the bandwidth inside the backfitting loop, but also much slower, since the number of combinations of bandwidths is very large!

Note that you did not have to write any code for this part.

(d) *Why is it helpful to set* $\alpha = \mathbf{E}[Y]$ *and require that* $\mathbf{E}[f(X_j)] = 0$*?* Hint: *What happens if you add 19740228 to* f_1 *and subtract it from* f_2*?*

ANSWER: For all x_1 and x_2, $f(x_1) + 19740228 + f(x_2) - 197402028 = f(x_1) + f(x_2)$. Thus a model in which the partial response functions are f_1 and f_2 makes the same predictions as one in which the partial response functions are $g_1(x_1) = f_1(x_1) + 19740228$ and $g_2(x_2) = f_2(x_2) - 197402828$. In fact we could add and subtract any set of constants to the functions, and so long as their sum was zero it would not alter the predictions. It would, however, prevent the estimated partial response functions from converging. Requiring each function to have expectation 0 keeps this from happening.

2. Download the California housing data set used on the exam.

   (a) *Linearly regress the* log *of the median house price on all the other variables. Report your regression coefficients, your mean squared error, and the distribution of residuals. Is the latter Gaussian? Do scatter-plots of residuals against predictors show any trends?*

   Answer: First, load the data into R:

```
> CAHousing = read.table("cadata.dat",header=TRUE)
> dim(CAHousing)
[1] 20640     9
> colnames(CAHousing)
[1] "MedianHouseValue" "MedianIncome"     "MedianHouseAge"   "TotalRooms"
[5] "TotalBedrooms"    "Population"       "Households"       "Latitude"
[9] "Longitude"
```

   Running `dim()` and `colnames()` isn't necessary, but checks that things worked OK.

```
fit1 = lm(log(MedianHouseValue) ~ MedianIncome + MedianHouseAge
            + TotalRooms + TotalBedrooms + Population + Households
            + Latitude + Longitude, data= CAHousing)
```

   Now `summary(fit1)` or `fit1$coefficients` will give the coefficients. They are:

feature	coefficient
(Intercept)	-11.8
MedianIncome	0.178
MedianHouseAge	0.00326
TotalRooms	-0.0000319
TotalBedrooms	0.000480
Population	-0.000173
Households	0.000249
Latitude	-0.280
Longitude	-0.276

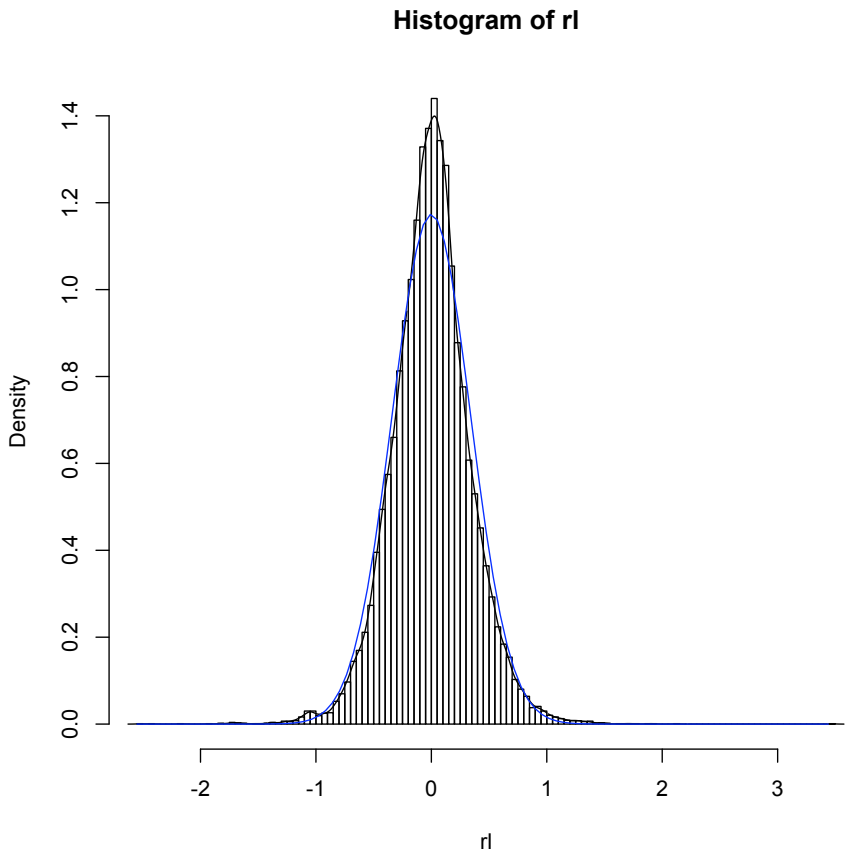
   To get the mean squared error, we square the residuals and average them:

```
> mean(fit1$residuals^2)
[1] 0.1155802
```

   We can look at the distribution of the residuals visually using commands like `hist` and `density` — the latter does a kernel *density* estimate, which is related to, but different than, the kernel *regression* estimates we've seen[2] Look at Figure 3.

   The figure suggests that the distribution isn't quite Gaussian. Another way to check this is to use a Q-Q plot against a Gaussian. This

[2] We're getting there.




```
rl = fit1$residuals
hist(rl,n=101,probability=TRUE)
lines(density(rl))
curve(dnorm(x,mean=mean(rl),sd=sd(rl)),col="blue",add=TRUE)
```

Figure 3: Histogram (bars) and smoothed density estimate (black curve) for the linear-model residuals. The blue curve is a Gaussian distribution with the same mean and standard deviation as the residuals. Notice that it has a lower peak, but broader tails, than the empirical distribution.

should be approximately straight if the residuals are Gaussian; instead it's curved at both ends, a lot. The residuals are not Gaussian.

(Of course there are more formal tests of normality, which you can use. However, R's implementation of the common Shapiro-Wilks test for normality, `shapiro.test`, doesn't allow sample sizes bigger than 5000. A somewhat dubious dodge is to use `shapiro.test(sample(rl,5000))` — that is, randomly pick 5000 of the residual values. Repeated 100 times,

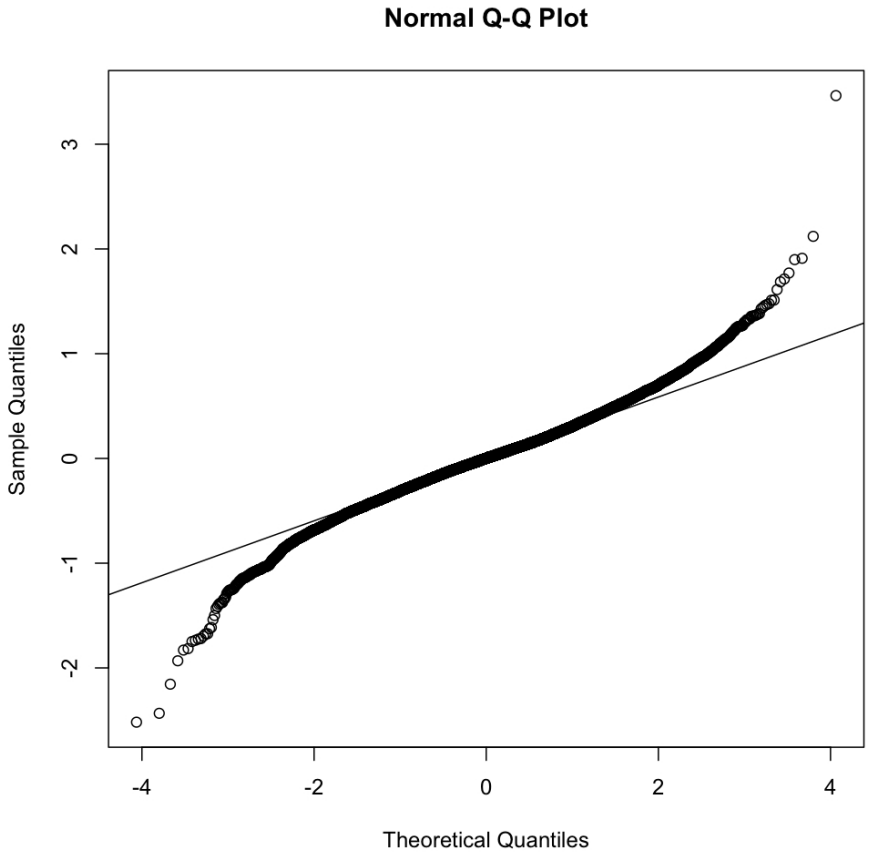
```
max(replicate(100,shapiro.test(sample(rl,5000))$p.value))
```

the maximum p-value I get is 3.9×10^{-15}, but, as I said, this is dubious.)

To plot the residuals against the median income, use

```
plot(CAHousing$MedianIncome,rl,ylab="residuals",xlab="Median Income")
```

and so on for the other predictor variables. The results are in Figure 5 and 6. Looking them over, the only one which is really bad is the plot of the residuals against median income, where there is a clear downward trend as income grows, and the variance clearly narrows. The variance of the residuals may seem to narrow as we go to high values of say population, but there just aren't many census units with large populations.




```
qqnorm(rl)
qqline(rl)
```

Figure 4: Q-Q plot for the linear model residuals against a Gaussian distribution. Notice the substantial curvature at both high and low quantiles.

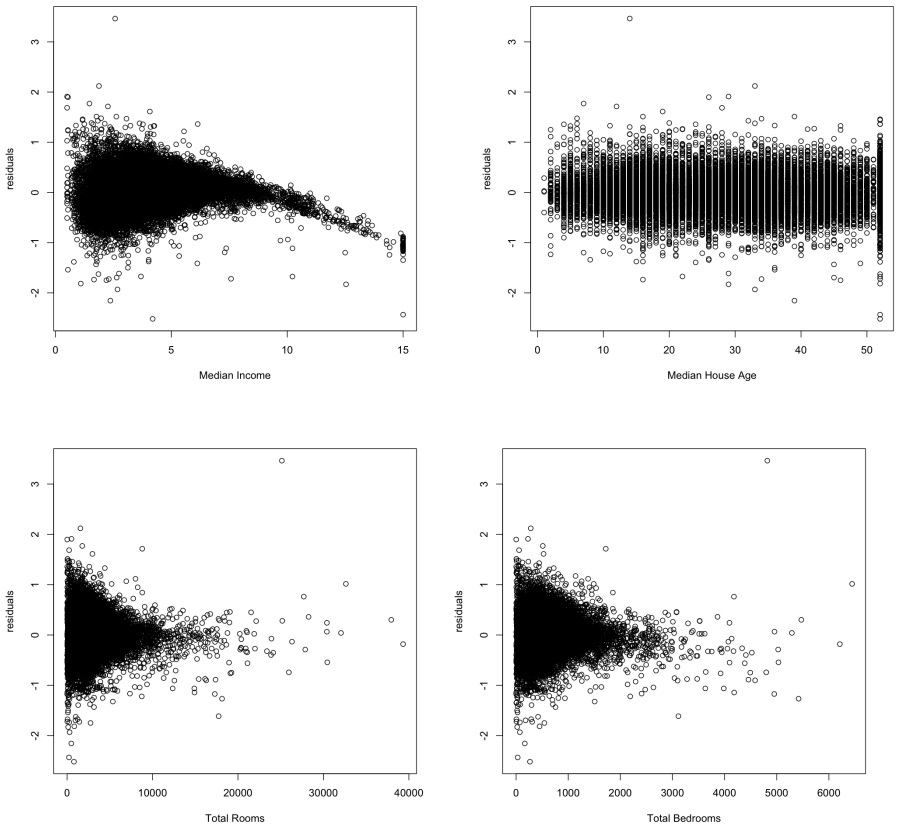


Figure 5: Residuals of the linear model plotted against its predictor variables: in reading order, median income, median house age, total rooms and total bedrooms.

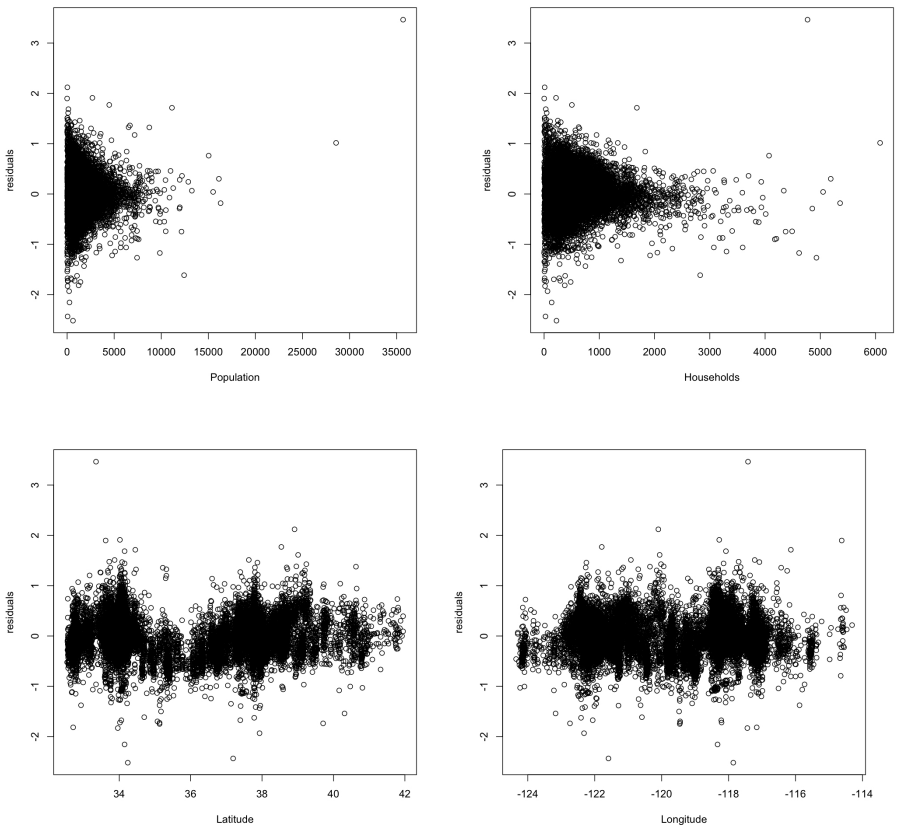


Figure 6: Residuals of the linear model plotted against its predictor variables: in reading order, population, number of households, latitude and longitude.

(b) *Using a kernel smoother, regress the log of the median house price on the median income. Use cross-validation to pick the bandwidth. What is the mean squared error? Plot the estimated regression function; is it linear? Plot the distribution of residuals; are they Gaussian? Plot the residuals versus the median income; do you see any trends?*

ANSWER: There are three approaches to doing this problem. One is to go and find a kernel regression function. The CRAN package `np` includes this, along with other tools for nonparametric regression. Another is make the built-in function `ksmooth` do the work. The third, of course, is to write our own kernel regression function.

Let's use `ksmooth`. Looking at `help(ksmooth)`, we see that we can control the number and location of points at which the fit is evaluated by using the options `n.points` and `x.points`. We can also control the range over which it is prepared to estimate values using the `range.x` option.

```
ksmooth(train.x,train.y,
        kernel="normal",
        range.x=range(c(train.x,test.x)),
        n.points=length(test.x),
        x.points = test.x)
```

will use the `train.x`,`train.y` pairs to make predictions at each of the `test.x` values. So here's how to do one CV run.

```
one.cv.run.ksmooth = function(x,y,p,h.grid) {
  n = length(x)
  n.sample = round(n*p)
  train.rows = sample(1:n,n.sample) # Pick training data
  train.x = x[train.rows]
  train.y = y[train.rows]
  test.x = x[-train.rows] # The rest is testing data
  test.y = y[-train.rows]
  # ksmooth orders its output by increasing x --- to avoid confusion let's
  # take care of that ourselves
  test.y = test.y[order(test.x)] # Sorts y values in order of increasing x
  test.x = sort(test.x) # Puts x itself in order
  mse = vector(length=length(h.grid))
  for (i in 1:length(h.grid)) {
    y.pred = ksmooth(train.x,train.y,kernel="normal",bandwidth=h.grid[i],
              range.x=range(x),n.points=length(test.x),x.points=test.x)$y
    mse[i] = mean((test.y - y.pred)^2)
  }
  return(mse)
}
```

Now use `replicate` to repeat this several times.

```
k.fold.cv.ksmooth = function(x,y,p,h.grid,k) {
  k.fold.results = replicate(k,one.cv.run.ksmooth(x,y,p,h.grid))
  mean.over.k.fold = apply(k.fold.results,1,mean)
  return(list(best.h = h.grid[which.min(mean.over.k.fold)],h.grid=h.grid,
              mean.mse.over.k.fold = mean.over.k.fold))
}
```

Finally, run it:

```
CAHousing.CV = k.fold.cv.ksmooth(CAHousing$MedianIncome,
                                 log(CAHousing$MedianHouseValue),
                                 0.9,c(0.1,1,10),10)
```

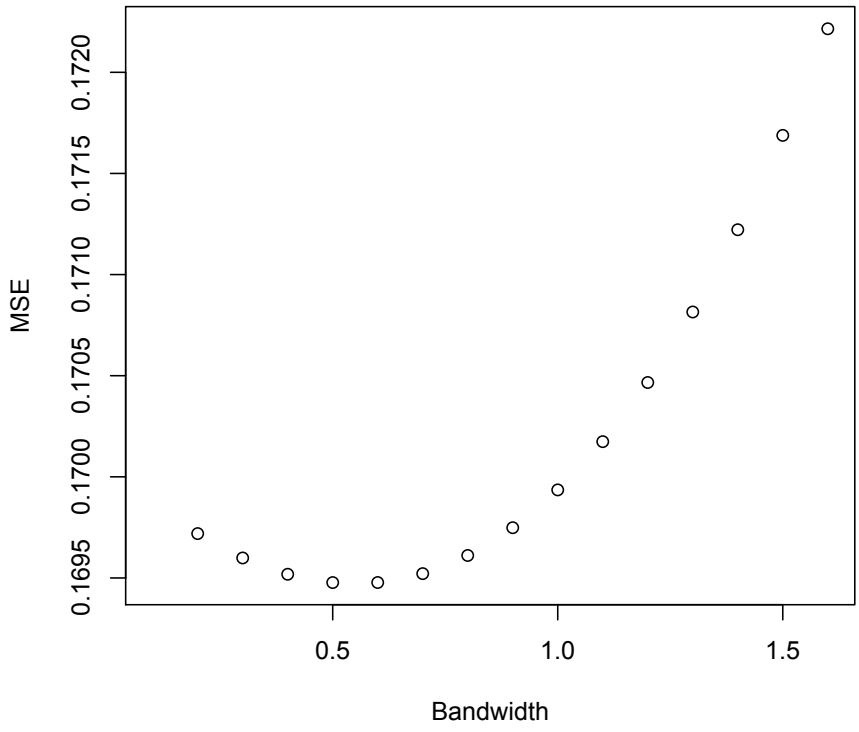
This uses a grid of only 3 bandwidths, 0.1, 1.0 and 10, but does 10-fold cross-validation, with a 90/10 train/test split in each fold. It doesn't take very long, but the result shows that 1 is a much better bandwidth than either the larger or smaller values. Now I expand the grid around 1:

```
CAHousing.CV = k.fold.cv.ksmooth(CAHousing$MedianIncome,
                                 log(CAHousing$MedianHouseValue),
                                 0.9,(1:16)/10,10)
```

This checks bandwidths from 0.1 to 1.6, in even steps of 0.1. Having this many bandwidths takes rather longer to check, several minutes on my laptop. I save the results in `CAHousing.CV` partly because they take so long to produce, but mostly so that I can make a plot of the out-of-sample error versus bandwidth (Figure 7). The best bandwidth is $h = 0.5$, but it turns out that it really makes very, very little difference what bandwidth we choose from this grid. Thus it's not worth refining it even more.

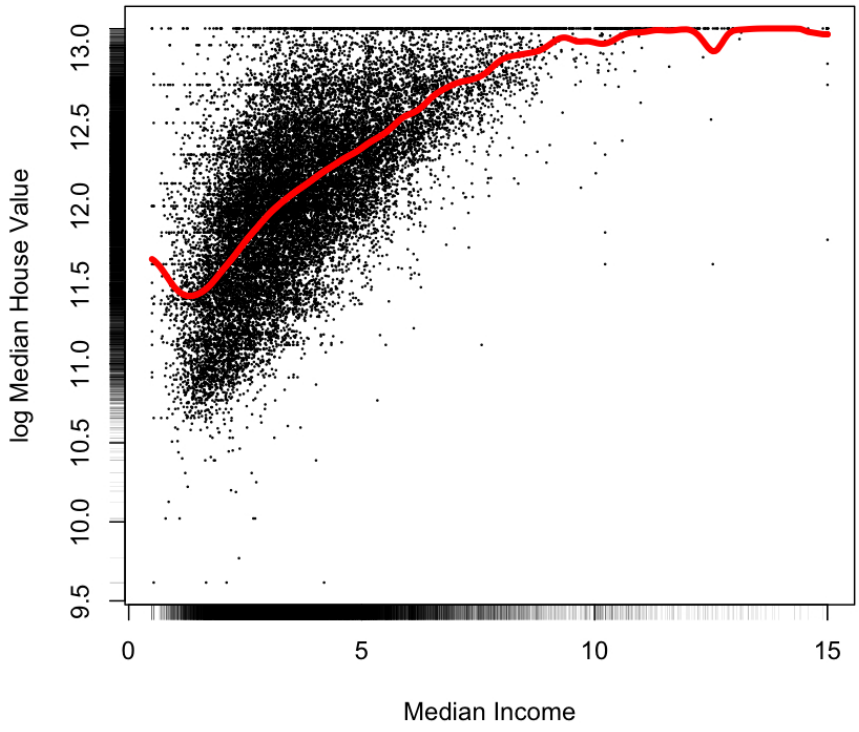
```
CAHousing.ks = ksmooth(CAHousing$MedianIncome,
                       log(CAHousing$MedianHouseValue),
                       kernel="normal",
                       bandwidth=CAHousing.CV$best.h,
                       x.points=CAHousing$MedianIncome)
```

`CAHousing.ks` now contains as its `$x` values the median income numbers, and its `$y` values the fitted log median prices. Figure 8 shows the fit to the data, and Figure 9 the distribution of residuals.



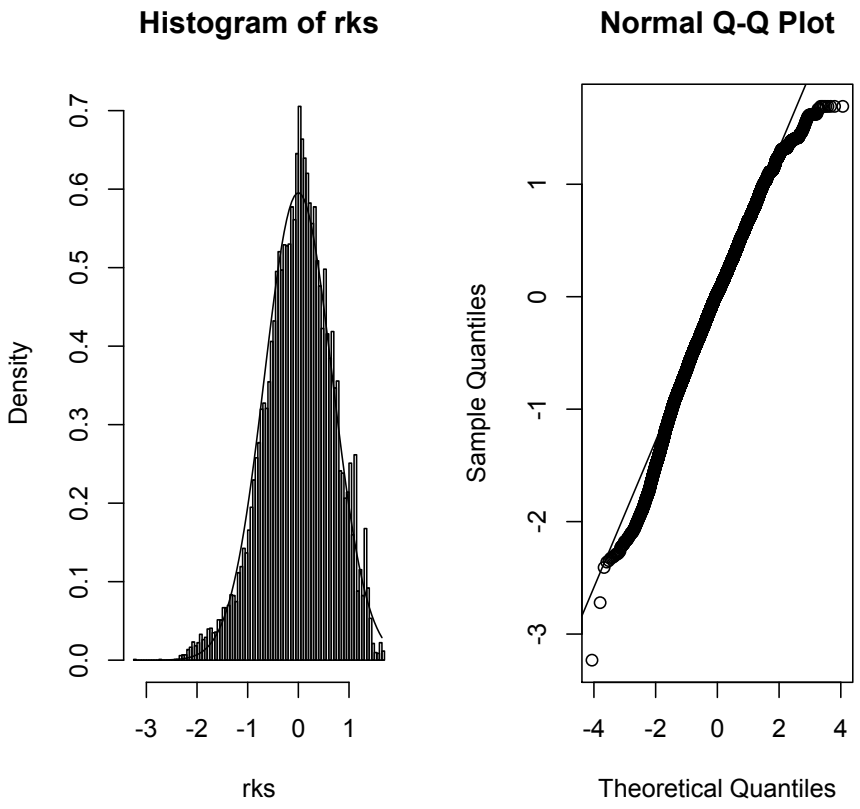

```
plot(CAHousing.CV$h.grid,CAHousing.CV$mean.mse.over.k.fold,xlab="Bandwidth",
     ylab="MSE")
```

Figure 7: Estimates of out-of-sample error for different bandwidths in `ksmooth` on the California housing data (10-fold CV with a 90/10 training/testing split). Note the vertical scale.




```
plot(CAHousing$MedianIncome,log(CAHousing$MedianHouseValue),
     xlab="Median Income",ylab="log Median House Value",cex=0.1,pch=19)
axis(1,at=CAHousing$MedianIncome,labels=FALSE,lwd=0.05)
axis(2,at=log(CAHousing$MedianHouseValue),labels=FALSE,lwd=0.05)
lines(CAHosuing.ks,lwd=4,col="red")
```

Figure 8: Kernel smoothing of log median house price on median income: data shown as small black dots, kernel regression (with CV-selected bandwidth) as thick red line. (The thickness is just for visual clarity and does not inlcude any uncertainty estimates.) Tick-marks along the axes give an indication of the marginal densities. The regression curve is not very linear over-all, but it is pretty linear in the region where we have the most data, which is before top-coding of the house prices kicks in.




```
par(mfrow=c(1,2))
hist(rks,n=101,probability=TRUE)
curve(dnorm(x,mean(rks),sd(rks)),add=TRUE)
qqnorm(rks)
qqline(rks)
```

Figure 9: Left, histogram of the residuals from the kernel smoothing, plus the density of a Gaussian with the same mean and variance. Right, Q-Q plot of the residuals. This is pretty normal.

(c) *Fit an additive model to the same data. Report the mean squared error and plots of the estimated functions. Are they close to linear? What is the distribution of total (i.e., not partial) residuals?*

ANSWER: Because of the difficulties people had with previous parts, this one has become EXTRA CREDIT.

First, using the home-made code:

```
am.fit = addm.functions(log(CAHousing[,1]),CAHousing[,2:9])
```

Here's the plot of the results (Figure 10.

```
par(mfrow=c(3,3))
for (j in 2:9) {
  curve(am.fit$partial.responses[[(j-1)]](x),
        from=min(CAHousing[,j]),to=max(CAHousing[,j]),
        xlab=colnames(CAHousing)[j],
        ylab=expression(hat(f)))
  axis(1,at=CAHousing[,j],lwd=0.1,labels=FALSE,col="grey")
}
plot(am.fit$y,am.fit$fitted,xlab="actual",ylab="predicted")
abline(0,1,lwd=4,col="blue")
```

This tells R to use a 3×3 grid for plotting, then plots each of the partial response functions (adding tick-marks for the training data), and finally plots the actual values of the response against the fitted values of the response; ideally these would cluster around the overlaid diagonal line.

Figure 11 shows the same thing, only using `addm.points` to fit the model. Notice the general similarity in shape to the partial response in Figure 10, but much greater roughness in some of them (e.g., `Households`, `TotalBedrooms`), and the larger range for $\widehat{f}$. This is much less smooth, and the fit to the data is tighter. (The MSEs are 0.0390 for `addm.points` and 0.118 for `addm.functions`.) A production version of the code should really include some way of picking the amount of smoothing (i.e., the bandwidth).

In neither case, however, are the partial response functions especially linear.

The residuals look pretty good.

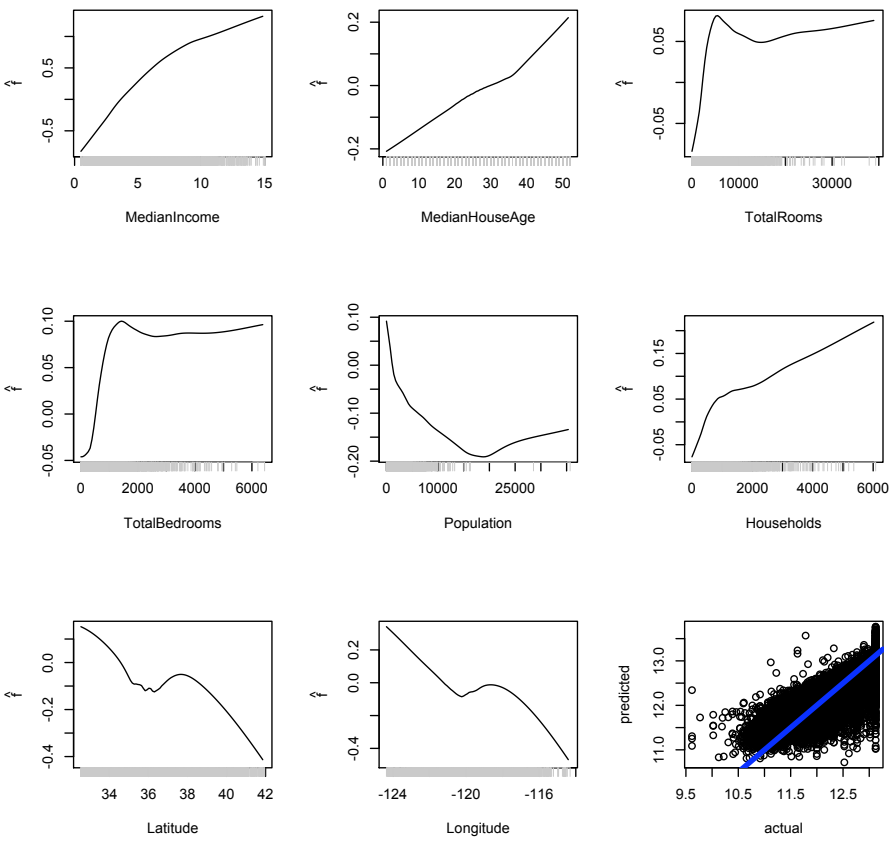


Figure 10: Estimated partial response functions, and predicted vs. actual response values, using the `addm.functions` additive-model fitter.

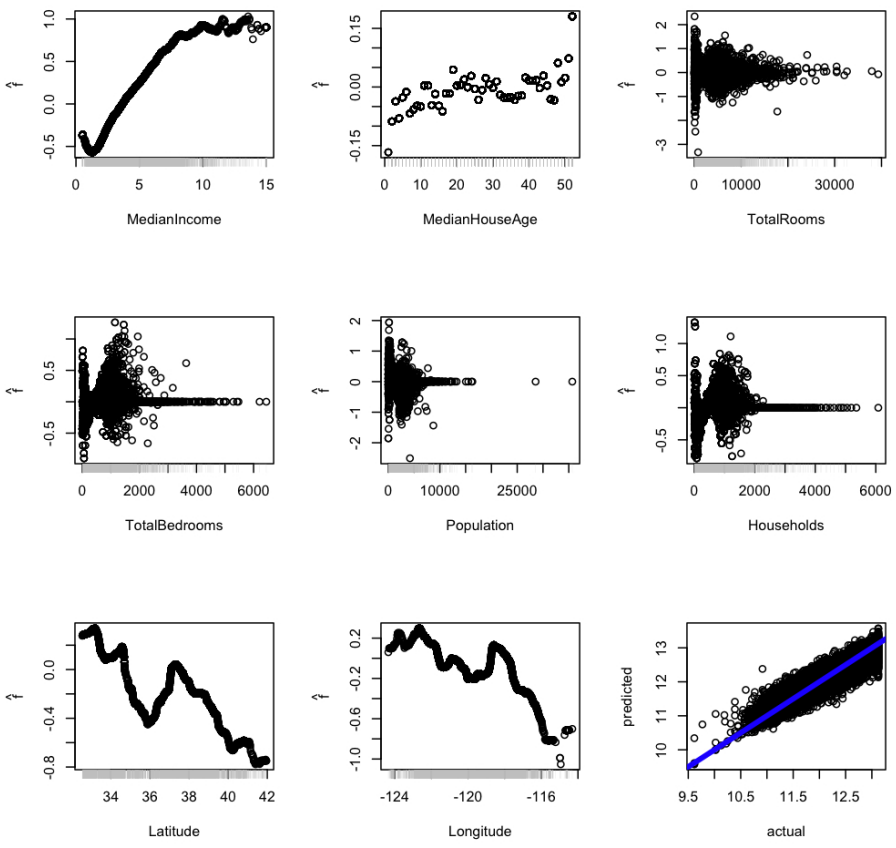
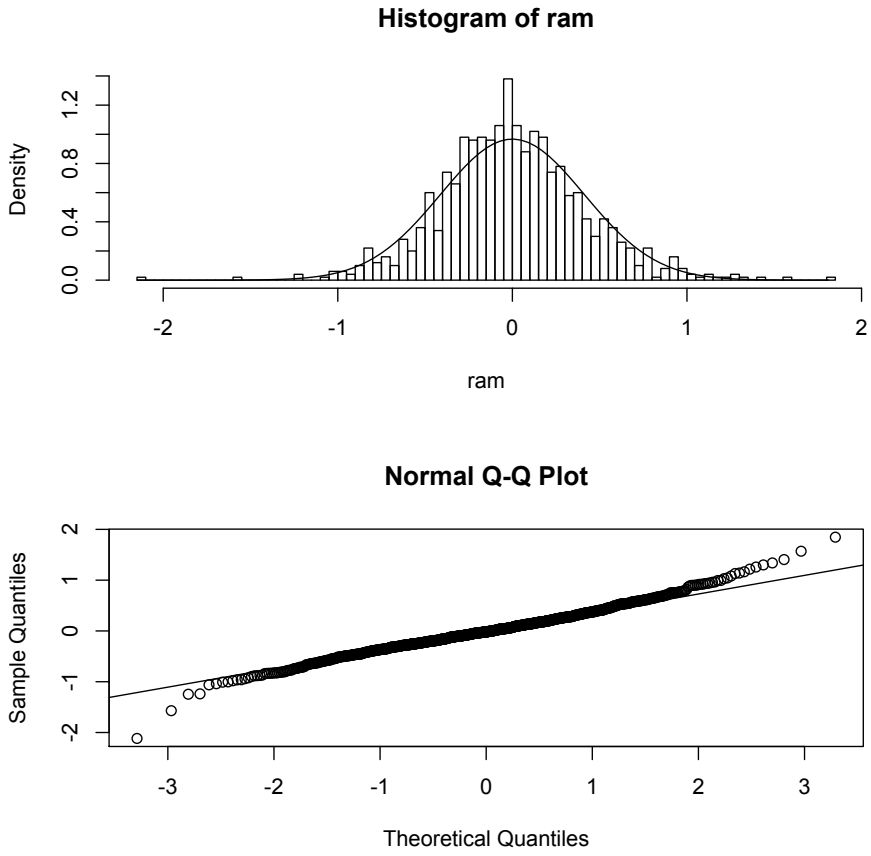


Figure 11: Estimated partial response functions, and predicted vs. actual response values, using the `addm.points` additive-model fitter.




```
par(mfrow=c(2,1))
hist(ram,n=101,probability=TRUE)
curve(dnorm(x,mean(ram),sd(ram)),add=TRUE)
qqnorm(ram)
qqline(ram)
```

Figure 12: Distribution of residuals from the additive model of Figure 10, showing the histogram, a Gaussian with the same man and standard deviation, and a Q-Q plot.