Nonparametric/Blackbox Regression and Classification Continued

See Syllabus for reference reading

K-Nearest Neighbors

- A generic nonlinear modeling tool that is extremely flexible
- Perhaps the simplest modeling idea of all
- Can be effective for data sets with large n, small k (or large k but with predictors on a low-dimensional manifold), high SNR, and no categorical predictors with > 2 categories
- Based on the name, can you guess how K-nearest neighbors works?

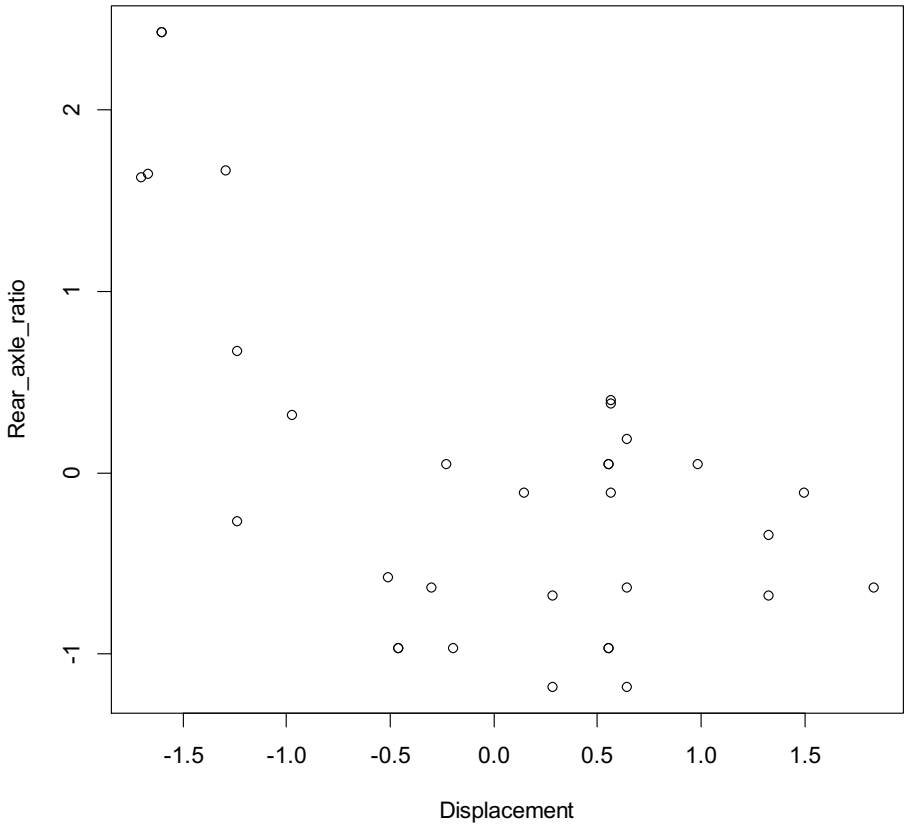
Structure of 1-Nearest Neighbors (for regression)

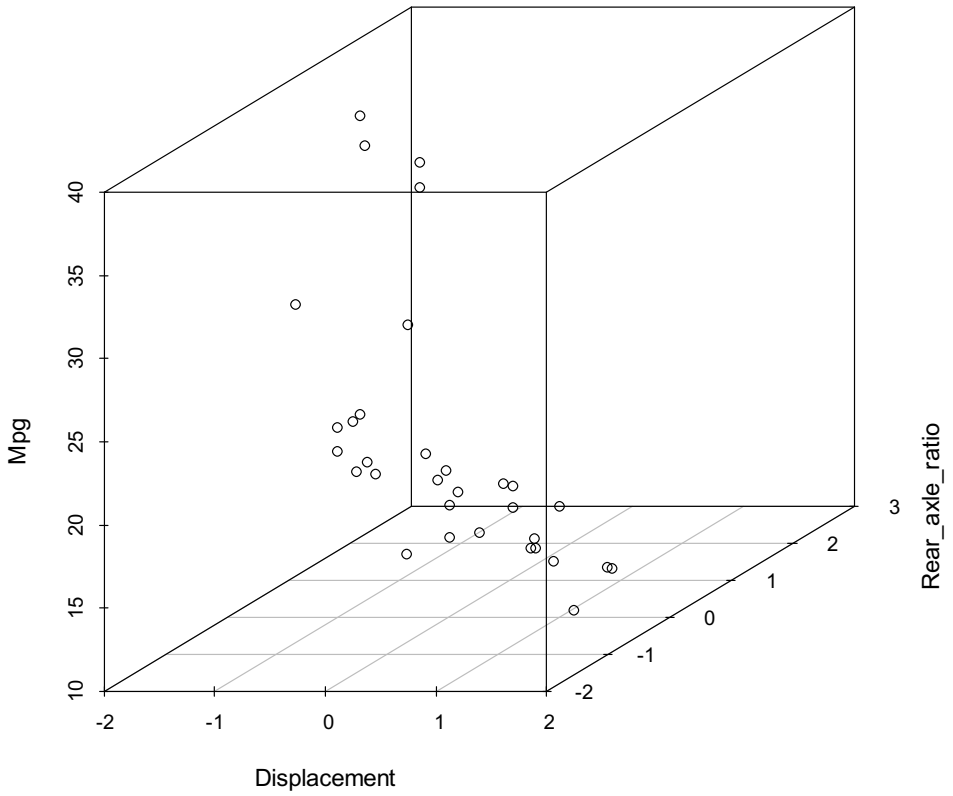
- You need a set of training data $\{y_i, \mathbf{x}_i: \ i = 1, 2, \ldots, n\}$, but you do not fit a model.
- For 1-Nearest Neighbors, to predict Y for a new case with predictors $\mathbf{x}$:
  - find the $\mathbf{x}_i$ in your training set that is the closest neighbor to $\mathbf{x}$
  - then take the predicted Y to be the response value for that training observation

Illustration of K-NN for Gas Mileage data

```
library(scatterplot3d)
library(rgl)
GAS<-read.csv("Gas_Mileage.csv",header=TRUE)
GAS1<-GAS
GAS1[,2:12]<-sapply(GAS1[,2:12], function(x) (x-mean(x[!is.na(x)]))/sd(x[!is.na(x)]))
GAS[1:10,]
attach(GAS1)
GAS1[c(1,2,6)]
plot3d(Displacement,Rear_axle_ratio,Mpg)
###
plot(Displacement,Rear_axle_ratio,type="p")
###
identify(Displacement,Rear_axle_ratio)
```

Nearest Neighbors Example: Gas_Mileage.csv





Calculating Distances to find Nearest Neighbors

If we want to predict $Y(\mathbf{x})$ for a new case with predictors $\mathbf{x}$, the distance between $\mathbf{x}$ and the predictors $\mathbf{x}_i$ for the ith training case ($i = 1, 2, \ldots, n$) is measured via

$$d(\mathbf{x}, \mathbf{x}_i) = \|\mathbf{x} - \mathbf{x}_i\| = \sqrt{(\mathbf{x} - \mathbf{x}_i)^T (\mathbf{x} - \mathbf{x}_i)} = \sqrt{\sum_{j=1}^{k} (x_j - x_{ij})^2}$$

For 1-nearest neighbor, the prediction of $Y(\mathbf{x})$ is $\hat{y}(\mathbf{x}) = y_{i_1(\mathbf{x})}$

where $i_1(\mathbf{x})$ = row index of closest neighbor of $\mathbf{x}$

```
> GAS1[c(1,2,6)]
    Mpg Displacement Rear_axle_ratio
1  18.90    0.5540441    -0.96676902
2  17.00    0.5540441    -0.96676902
3  20.00   -0.2989055    -0.63495681
4  18.25    0.5625736    -0.10796096
5  20.07   -0.5121430    -0.57640172
6  11.20    1.3216988    -0.34218134
7  22.12   -0.4609660    -0.96676902
8  21.47   -0.1965516    -0.96676902
9  34.70   -1.6661838     1.64869189
10 30.40   -1.6047715     2.42942649
11 16.50    0.5540441     0.04818596
12 36.50   -1.7037136     1.62917353
13 21.50   -0.9727358     0.32144307
14 19.70   -0.2306696     0.04818596
15 20.30   -1.2371502     0.67277364
16 17.80    0.1446283    -0.10796096
17 14.39    1.8334686    -0.63495681
18 14.89    1.3216988    -0.67399354
19 17.80    0.5540441     0.04818596
20 16.41    0.2811002    -1.18147103
21 23.54   -0.4609660    -0.96676902
22 21.47    0.6393391    -1.18147103
23 16.59    0.9805189     0.04818596
24 31.90   -1.6047715     2.42942649
25 29.40   -1.2371502    -0.26410788
26 13.27    1.4922887    -0.10796096
27 23.90   -1.2917389     1.66821026
28 19.73    0.2811002    -0.67399354
29 13.90    0.5625736     0.37999817
30 13.27    0.5625736     0.39951653
31 13.77    0.6393391     0.18481452
32 16.50    0.6393391    -0.63495681
```

What is the predicted mpg for a car with Displacement = -1.0 and Rear_axle_ratio = 0.0 (in standardized units) using 1-nearest neighbor?

i.e., what is $\hat{y}(\mathbf{x})$ for $\mathbf{x} = [-1.0, 0]^T$?

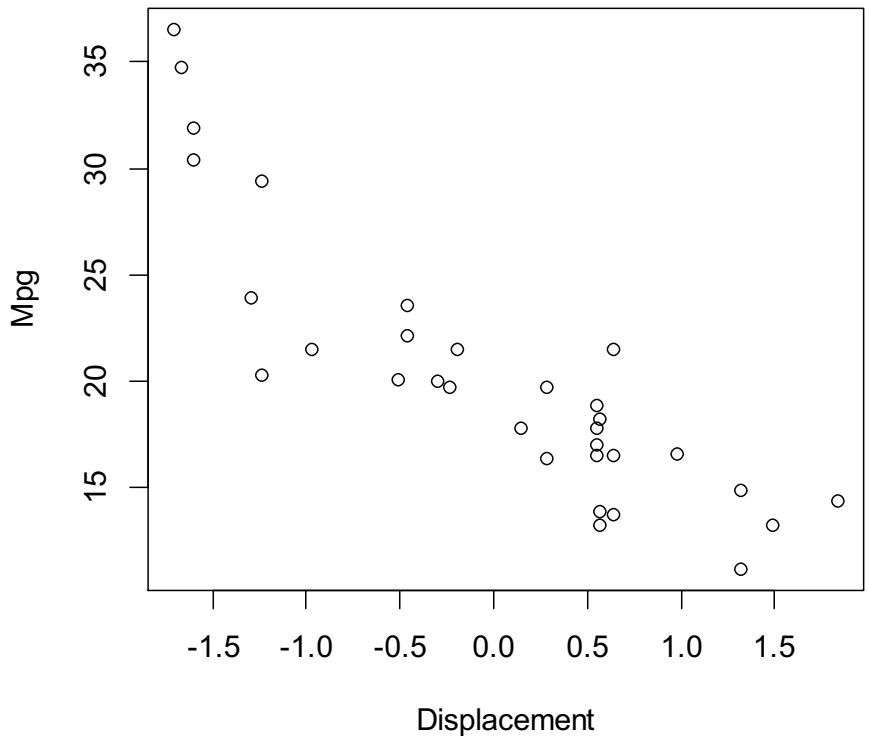
Row 13 is the nearest neighbor with $\mathbf{x}_{13} = [-0.97, 0.32]^T$ and distance

$$d(\mathbf{x}, \mathbf{x}_{13}) = \sqrt{(-1.00+0.97)^2 + (0.00-0.32)^2} = 0.32$$

So $i_1(\mathbf{x}) = 13$, and $\hat{y}(\mathbf{x}) = y_{13} = 21.5$

Discussion Points and Questions

- You should always standardize your predictors as a first step – why?
- How do you handle categorical predictors?
- For 1-nearest neighbor, what would a plot of $\hat{y}(\mathbf{x})$ versus $\mathbf{x}$ look for the gas mileage example with only Displacement as a predictor?



Structure of K-Nearest Neighbors (for regression)

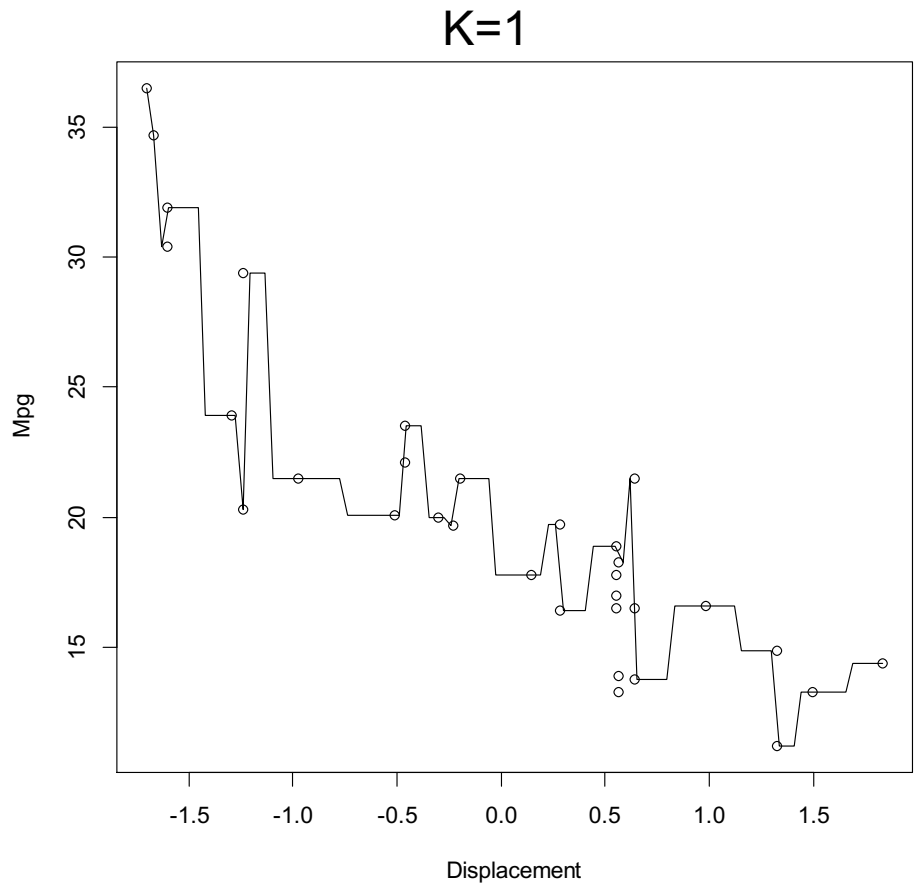
- More generally, for K-nearest neighbors, you use exactly the same procedure, except you:
  - find the K closest training $\mathbf{x}_i$'s to $\mathbf{x}$, and
  - then take the predicted Y to be the average response value for these K training observations:

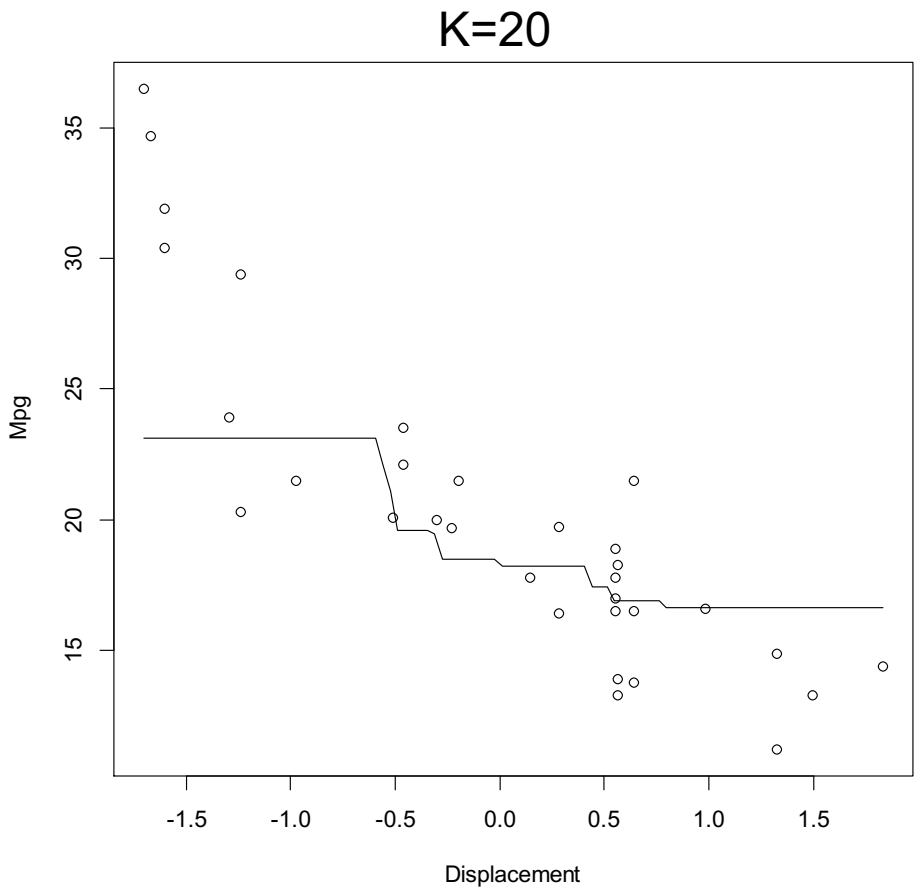
$$\hat{y}(\mathbf{x}) = \frac{1}{K} \sum_{l=1}^{K} y_{i_l(\mathbf{x})}$$

where $\{i_1(\mathbf{x}), i_2(\mathbf{x}), \ldots, i_K(\mathbf{x})\}$ = indices of K closest neighbors of $\mathbf{x}$

- The tradeoff of using large vs. small K is exactly the classic bias/variance tradeoff

Large Versus Small K (single predictor example)





- Why is the predictor in the left plot high variance and low bias?
- Why is the predictor in the right plot low variance and high bias?

Bias and Variance in Nearest Neighbors

- For K nearest neighbors (k-NN) regression with an assumed relationship $Y = g(\boldsymbol{X}) + \varepsilon$, the nonparametric predictor is of the form $\hat{g}(\boldsymbol{x}_0) = \frac{1}{K}\sum_{k=1}^{K} Y_{i_k(\boldsymbol{x}_0)}$, where $i_k(\boldsymbol{x}_0)$ is the index of the kth nearest neighbor to $\boldsymbol{x}_0$ from among the n training observations $\{\boldsymbol{x}_i : i = 1,2,\dots,n\}$ of $\boldsymbol{X}$.
- Treating the predictors $\boldsymbol{x}_i$ as fixed (for simplicity), the bias, variance, and MSE are

$$Bias[\hat{g}(\boldsymbol{x}_0)] = E[\hat{g}(\boldsymbol{x}_0)] - g(\boldsymbol{x}_0) = \frac{1}{K}\sum_{k=1}^{K} g\left(\boldsymbol{x}_{i_k(\boldsymbol{x}_0)}\right) - g(\boldsymbol{x}_0)$$

$$Var[\hat{g}(\boldsymbol{x}_0)] = \frac{\sigma_\varepsilon^2}{K}$$

$$MSE[\hat{g}(\boldsymbol{x}_0)] = \sigma_\varepsilon^2 + \left(\frac{1}{K}\sum_{k=1}^{K} g\left(\boldsymbol{x}_{i_k(\boldsymbol{x}_0)}\right) - g(\boldsymbol{x}_0)\right)^2 + \frac{\sigma_\varepsilon^2}{K}$$

- What is the effect of the "complexity parameter" K on each?

Another Example of Large Vs. Small K

This is a classification example from HTF with two response categories (blue or orange in the figures) and two predictors. The following scatterplots are x_1 vs. x_2 also showing the decision boundaries for the K-nearest neighbors classifiers with K = 15 and K = 1

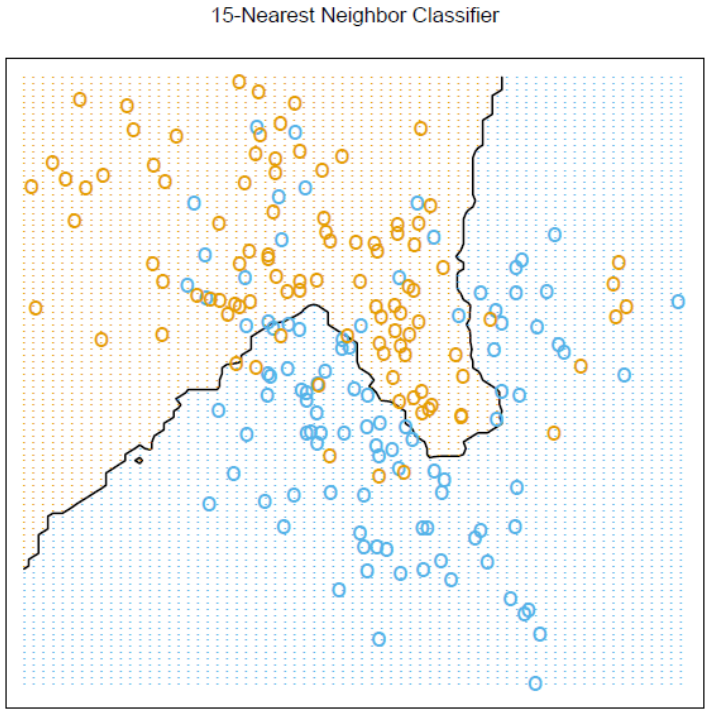


FIGURE 2.2. *The same classification example in two dimensions as in F ure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$) *then fit by* 15*-nearest-neighbor averaging as in (2.8). The predicted class is he chosen by majority vote amongst the* 15*-nearest neighbors.*

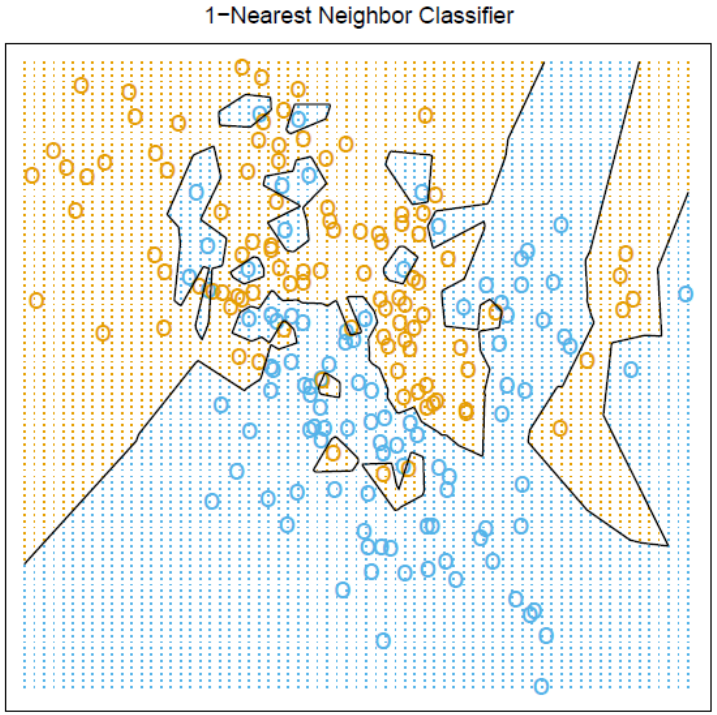


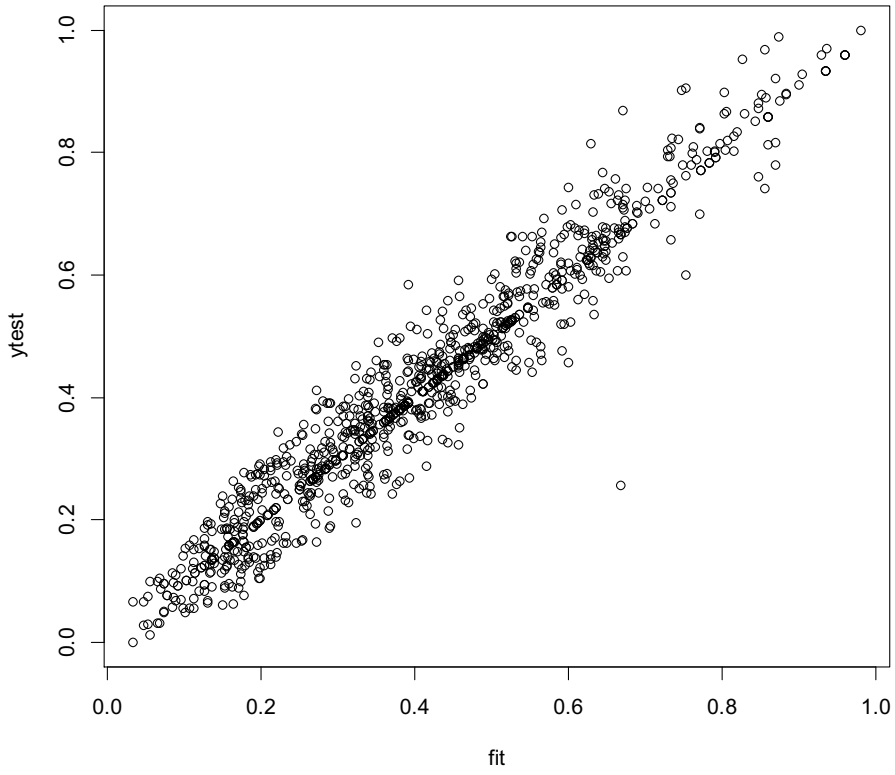
FIGURE 2.3. *The same classification example in two dimensions as in Figure 2.1. The classes are coded as a binary variable* (BLUE $= 0$, ORANGE $= 1$), *and then predicted by* 1*-nearest-neighbor classification.*

K-NN for Concrete data

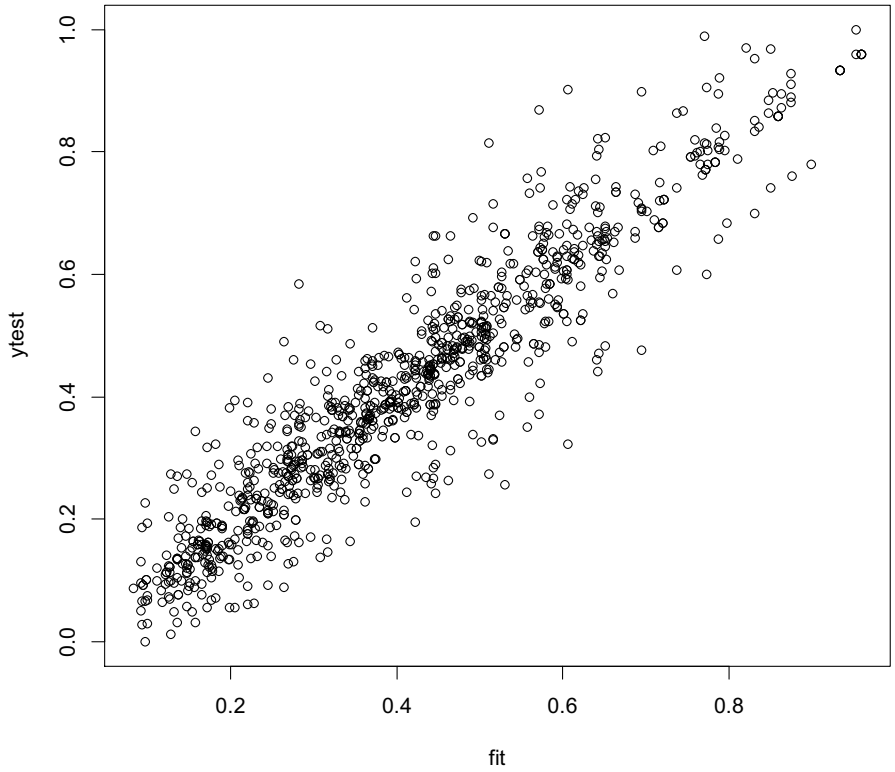
```
library(yaImpute)
CRT<-read.csv("concrete.csv", header=TRUE)
CRT1<-CRT
CRT1[1:8]<-sapply(CRT1[1:8], function(x) (x-mean(x))/sd(x)) #standardize predictors
CRT1[9]<-(CRT1[9]-min(CRT1[9]))/(max(CRT1[9])-min(CRT1[9]))
train<-as.matrix(CRT1[,1:8])
test<-as.matrix(CRT1[,1:8])
ytrain<-CRT1[,9]
ytest<-CRT1[,9]
K=3
out<-ann(train,test,K)
ind<-as.matrix(out$knnIndexDist[,1:K])
D<-as.matrix(out$knnIndexDist[,(1+K):(2*K)])
fit<-apply(ind,1,function(x) mean(ytrain[x]))
plot(fit,ytest)
1-var(ytest-fit)/var(ytest)
```

y versus yhat for training data with different K

K=2, training fit



K=3, training fit

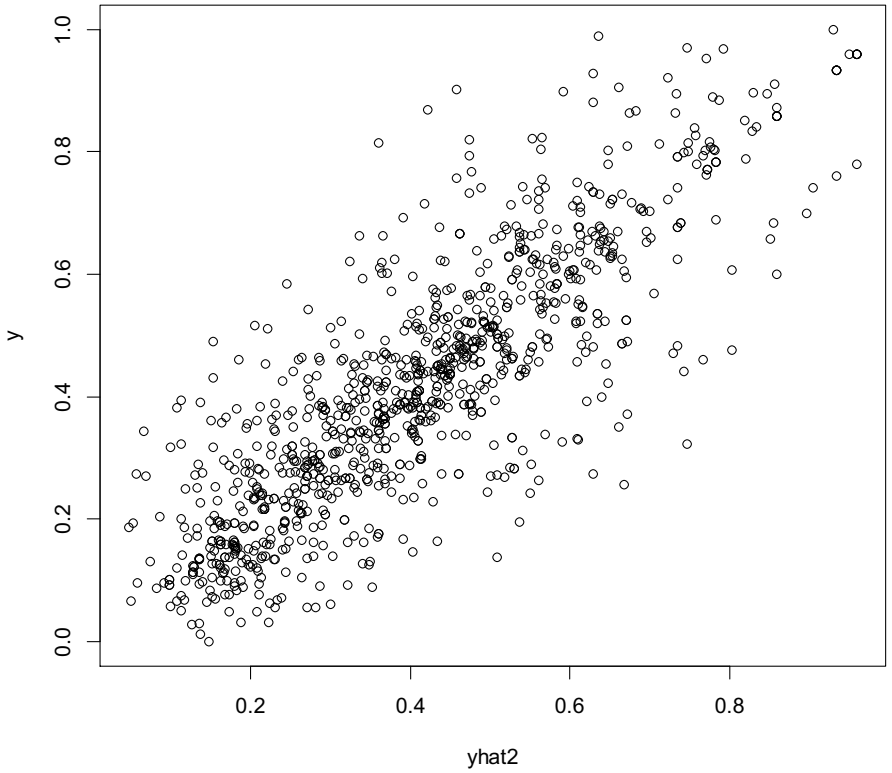


CV to Choose the Best K

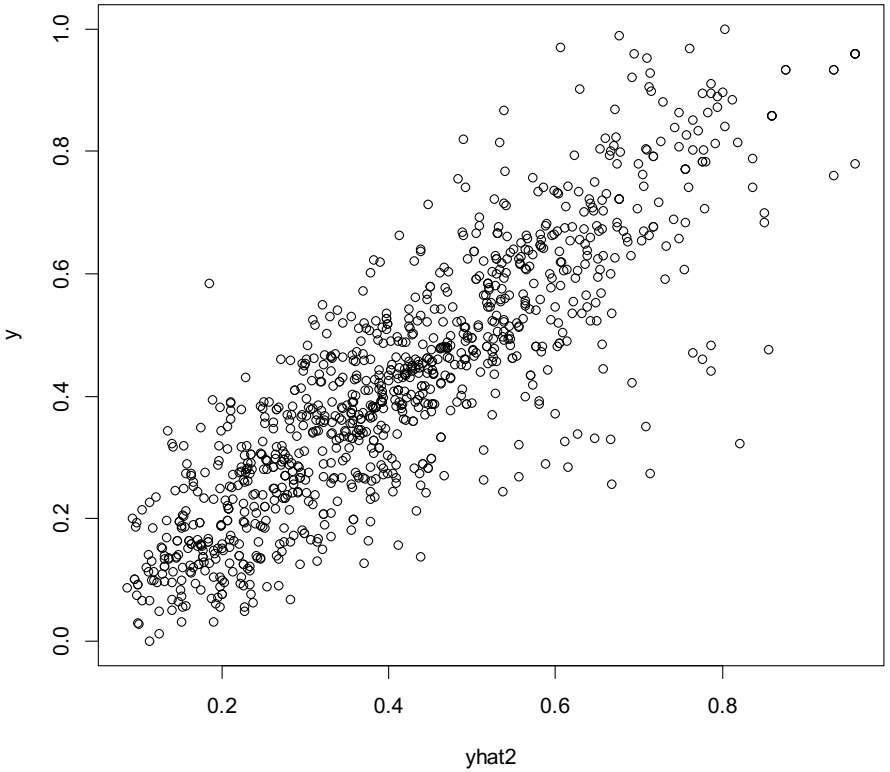
```
Nrep<-50 #number of replicates of CV
K<-10  #K-fold CV on each replicate
n.models = 2 #number of different models to fit
n=nrow(CRT1)
y<-CRT1$Strength
yhat=matrix(0,n,n.models)
MSE<-matrix(0,Nrep,n.models)
for (j in 1:Nrep) {
  Ind<-CVInd(n,K)
  for (k in 1:K) {
     train<-as.matrix(CRT1[-Ind[[k]],1:8])
     test<-as.matrix(CRT1[Ind[[k]],1:8])
     ytrain<-CRT1[-Ind[[k]],9]
     K1=3;K2=2
     out<-ann(train,test,K1,verbose=F)
     ind<-as.matrix(out$knnIndexDist[,1:K1])
     yhat[Ind[[k]],1]<-apply(ind,1,function(x) mean(ytrain[x]))
     out<-ann(train,test,K2,verbose=F)
     ind<-as.matrix(out$knnIndexDist[,1:K2])
     yhat[Ind[[k]],2]<-apply(ind,1,function(x) mean(ytrain[x]))
  } #end of k loop
  MSE[j,]=apply(yhat,2,function(x) sum((y-x)^2))/n
} #end of j loop
MSEAve<- apply(MSE,2,mean); MSEAve #averaged mean square CV error
MSEsd <- apply(MSE,2,sd); MSEsd   #SD of mean square CV error
r2<-1-MSEAve/var(y); r2  #CV r^2
plot(yhat[,2],y)
```

y versus yhat for CV with different K

K=2, 10-fold CV



K=3, 10-fold CV



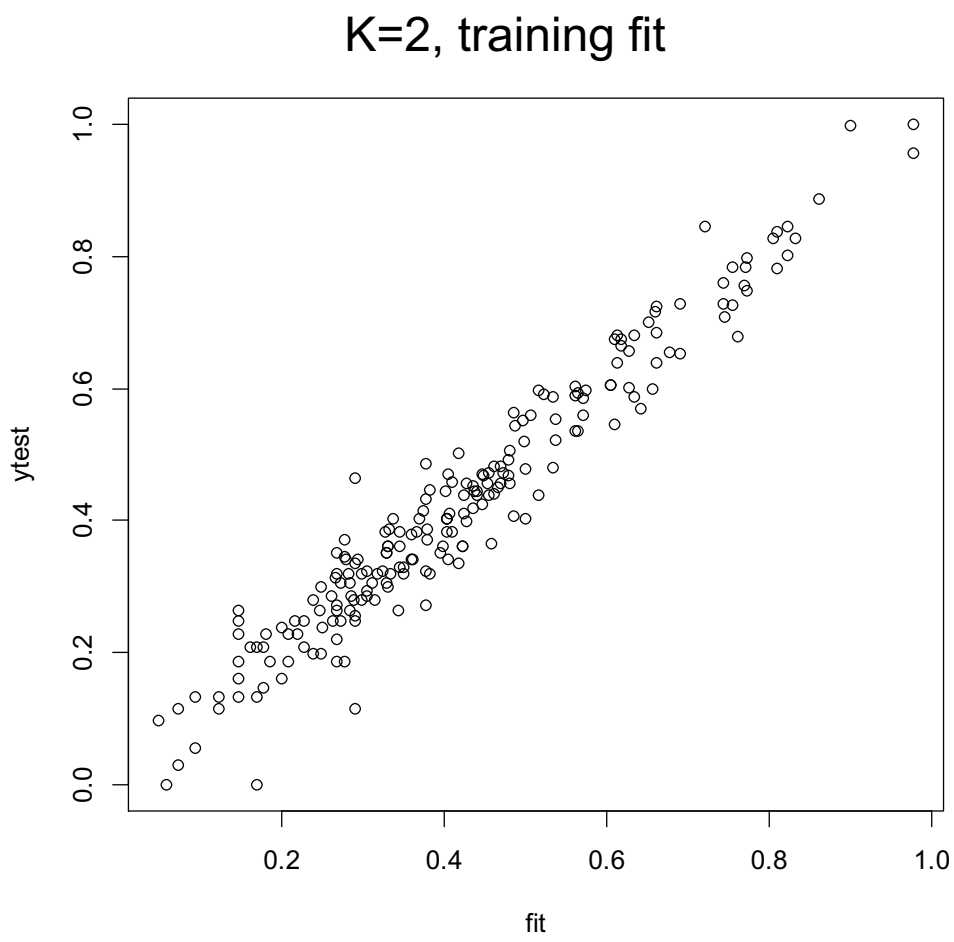
Discussion Points and Questions

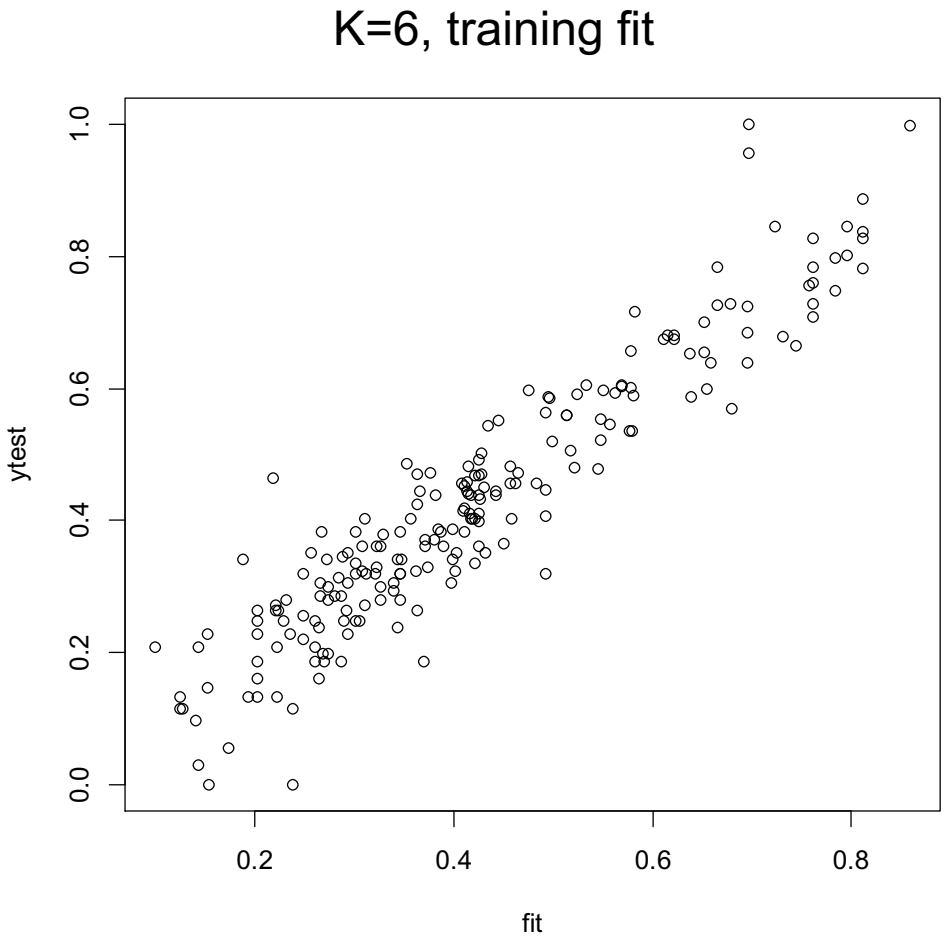
- Did the 3-nearest neighbors method do better than the best neural network model or the linear regression model?
- How can you tell which predictors have the largest effect on the response in the K-nearest neighbors model?
- What are the parameters of the fitted model?

K-NN for CPUS data

```
library(yaImpute)
CPUS<-read.table("cpus.txt",sep="\t")
CPUS1<-CPUS[2:8]
CPUS1[c(1:3,7)]<-sapply(CPUS1[c(1:3,7)], log10)  #take log of first three predictors and
    response
CPUS1[1:6]<-sapply(CPUS1[1:6], function(x) (x-mean(x))/sd(x)) #standardize predictors
CPUS1[7]<-(CPUS1[7]-min(CPUS1[7]))/(max(CPUS1[7])-min(CPUS1[7]))
train<-as.matrix(CPUS1[,1:6])
test<-as.matrix(CPUS1[,1:6])
ytrain<-CPUS1[,7]
ytest<-CPUS1[,7]
K=6
out<-ann(train,test,K)
ind<-as.matrix(out$knnIndexDist[,1:K])
D<-as.matrix(out$knnIndexDist[,(1+K):(2*K)])
fit<-apply(ind,1,function(x) mean(ytrain[x]))
plot(fit,ytest)
1-var(ytest-fit)/var(ytest)
```

y versus yhat for training data with different K



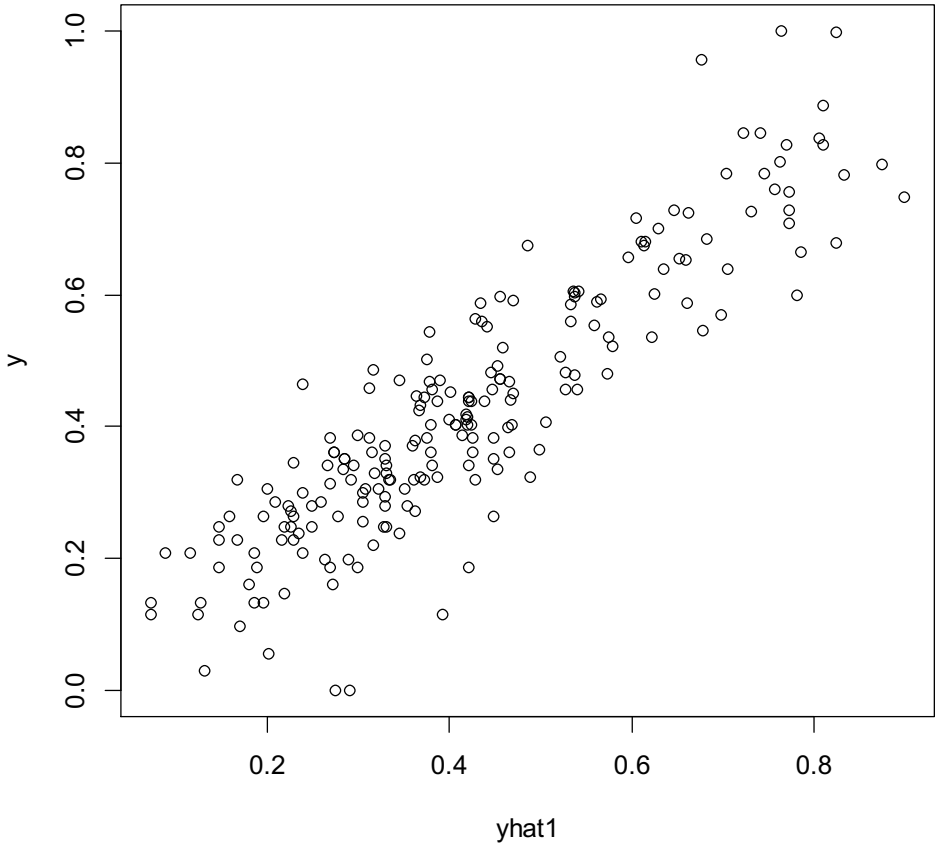


CV to Choose the Best K

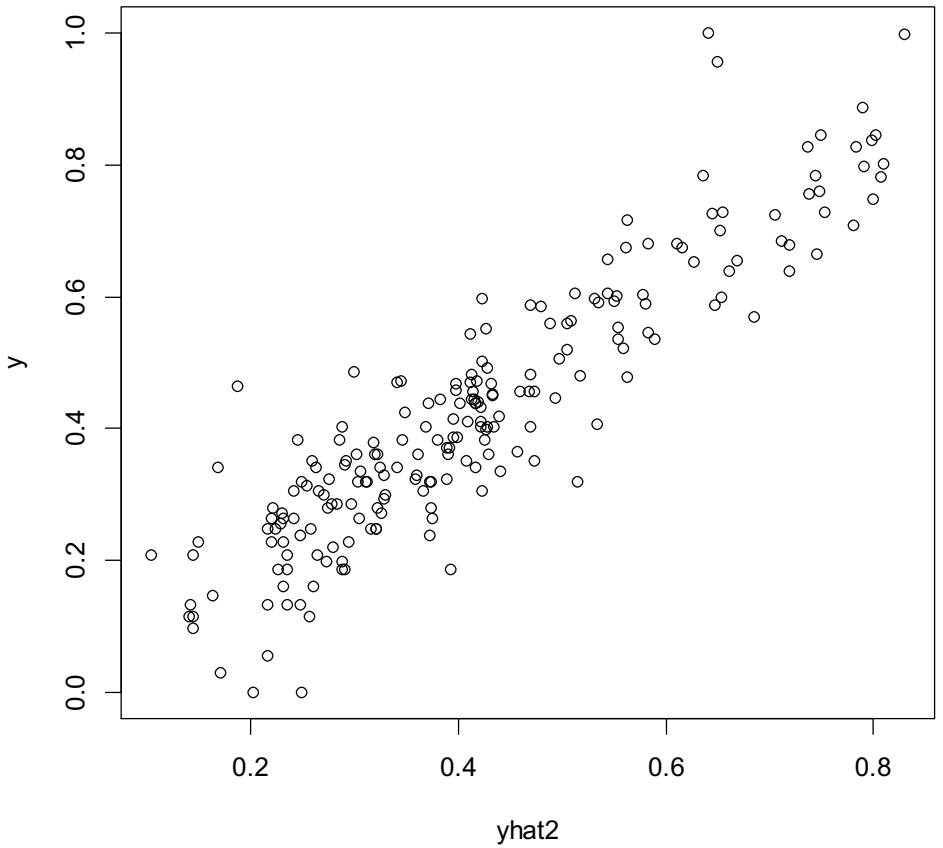
```
Nrep<-10 #number of replicates of CV
K<-10  #K-fold CV on each replicate
n=nrow(CPUS1)
y<-CPUS1$perf
SSE<-matrix(0,Nrep,2)
for (j in 1:Nrep) {
  Ind<-CVInd(n,K)
  yhat1<-y;
  yhat2<-y;
  for (k in 1:K) {
     train<-as.matrix(CPUS1[-Ind[[k]],1:6])
     test<-as.matrix(CPUS1[Ind[[k]],1:6])
     ytrain<-CPUS1[-Ind[[k]],7]
     K1=2;K2=6
     out<-ann(train,test,K1,verbose=F)
     ind<-as.matrix(out$knnIndexDist[,1:K1])
     yhat1[Ind[[k]]]<-apply(ind,1,function(x) mean(ytrain[x]))
     out<-ann(train,test,K2,verbose=F)
     ind<-as.matrix(out$knnIndexDist[,1:K2])
     yhat2[Ind[[k]]]<-apply(ind,1,function(x) mean(ytrain[x]))
  } #end of k loop
  SSE[j,]=c(sum((y-yhat1)^2),sum((y-yhat2)^2))
} #end of j loop
SSE
SSEAve<-apply(SSE,2,mean);SSEAve
plot(yhat2,y)
1-SSEAve/n/var(y)
```

y versus yhat for CV with different K

K=2, n-fold CV



K=6, n-fold CV



Discussion Points and Questions

- Did the 6-nearest neighbors method do better than the best neural network model or the linear regression model?
- How can you tell which predictors have the largest effect on the response in the K-nearest neighbors model?
- What are the parameters of the fitted model?

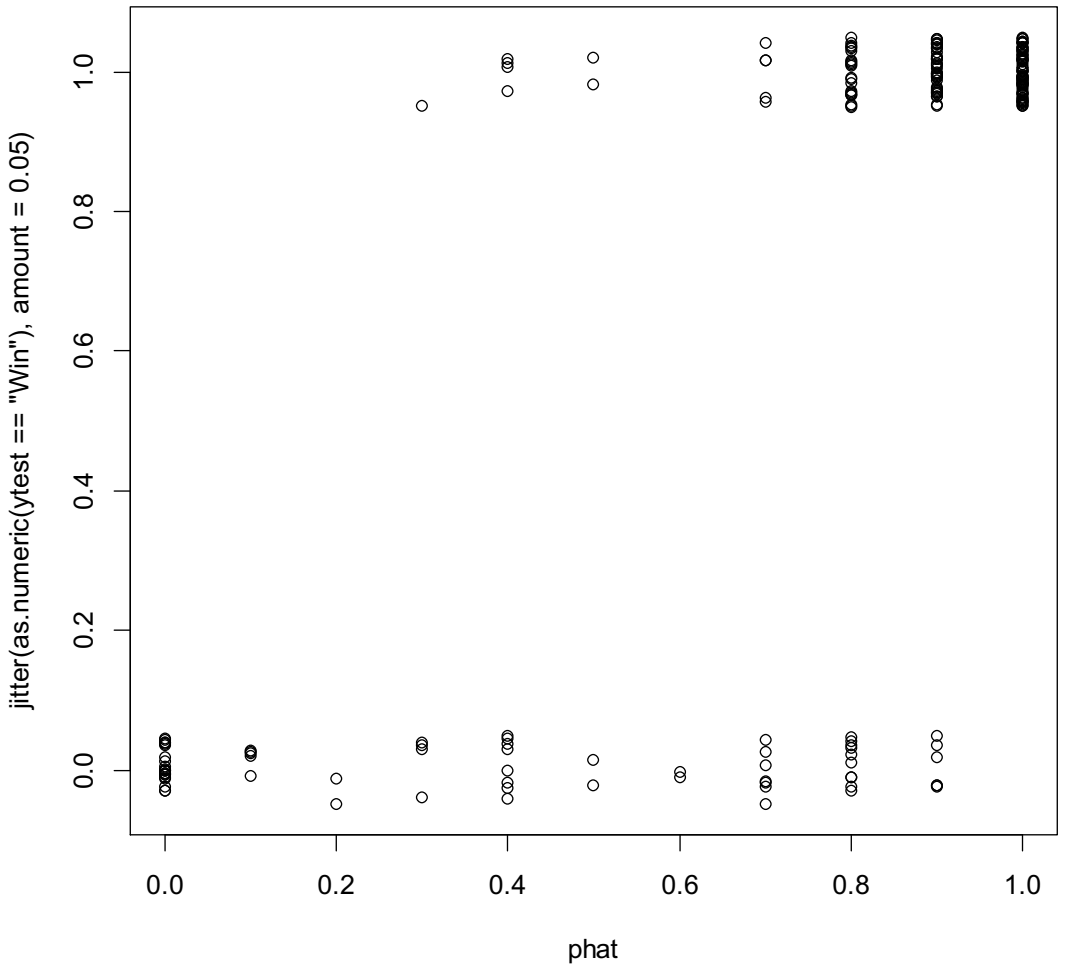
K-Nearest Neighbors for Classification

- Like CART models, it is straightforward to use nearest neighbors for classification.
- For binary classification, to predict $Pr\{Y=1 \mid \mathbf{x}\}$ for a new case, find the K nearest neighbors as before, and take the predicted $Pr\{Y=1 \mid \mathbf{x}\}$ to be the fraction of the K nearest neighbors having $y = 1$ responses
- If we have more than two response categories, we take the predicted probability for each category to be the fraction of nearest neighbors with response values belonging to that category.

K-NN for FGL data

```
FGL<-read.table("fgl.txt",sep="\t")
z<-(FGL$type == "WinF") | (FGL$type == "WinNF")
y<-as.character(FGL$type)
y[z]<-"Win"; y[!z]<-"Other"
FGL<-data.frame(FGL,"type_bin"=as.factor(y))  #add a binary factor response column
y[y == "Win"]<-1;y[y == "Other"]<-0;
FGL<-data.frame(FGL,"type01"=as.numeric(y))  #also add a binary numeric response column
FGL1<-FGL
FGL1[1:9]<-sapply(FGL1[1:9], function(x) (x-mean(x))/sd(x)) #standardize predictors
train<-as.matrix(FGL1[,1:9]); test<-as.matrix(FGL1[,1:9])
ytrain<-FGL1[,11]; ytest<-FGL1[,11]
K=5
out<-ann(train,test,K)
ind<-as.matrix(out$knnIndexDist[,1:K])
phat<-apply(ind,1,function(x) sum(ytrain[x]=="Win")/length(ytrain[x]))
plot(phat,jitter(as.numeric(ytest=="Win"),amount=.05))
####can alternatively use the following
library(class)
out<-knn(train, test, ytrain, k = 10, prob = T)
```

y (with jitter) versus phat for FGL training data



Discussion Points and Questions

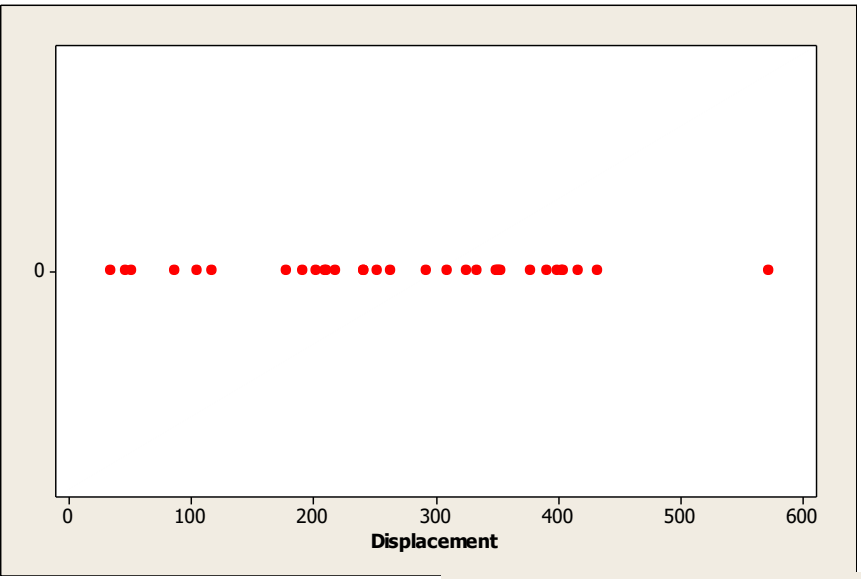
- In the preceding, what was the training misclassification rate? What would happen to the training misclassification rate if we decreased K? What would it be for K = 1?
- To find the best K for K-nearest neighbors for classification problems, you must use CV, just like for any other method.
- What CV measure would you use to find the best K for the FGL data?
- Is finding the CV errors for K-nearest neighbors substantially more computationally expensive than finding the training errors, like it is for all of the other methods we have covered?

Pros and Cons of Nearest Neighbors

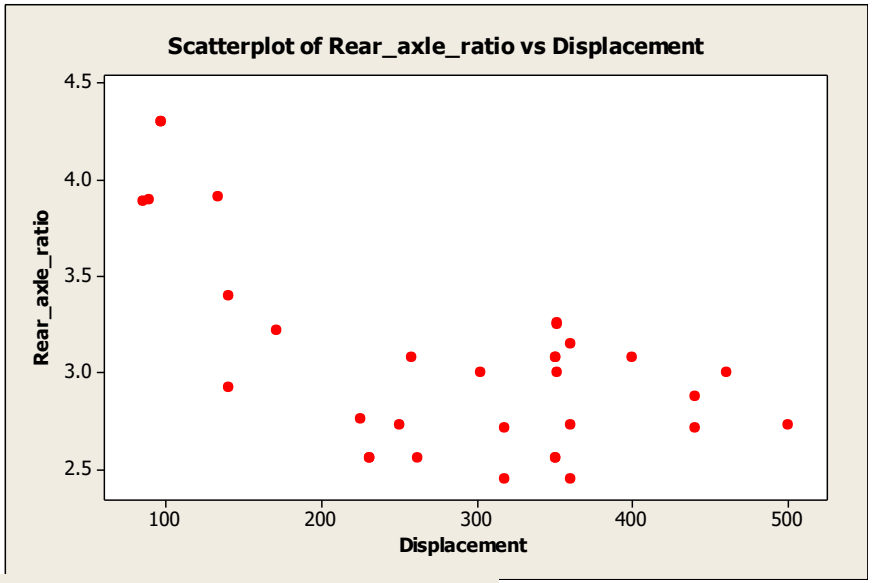
- Pros:
  - The most flexible of all – can represent any nonlinear relationship (as long as you have sufficiently large n).
  - Easy to use. No model fitting required
- Cons:
  - There is no real fitted model (nor even any indication of which predictors are most important), so not suitable for interpretation or explanatory purposes.
  - Because there is no fitted model, you need to retain all the training data to predict.
  - With large k (the number of predictors), you need very large n, because neighbors get further away in higher dimensions.
  - For most supervised learning methods, large n increases the computational expense for training, but not for new case prediction. Large n is more problematic for nearest neighbors, because the "training" occurs for every new case prediction
  - With very large n, we need computational tricks (e.g. tree-based methods) to efficiently search for nearest neighbors.
  - Not well suited for categorical predictors

Effect of dimension (k) on distance between neighbors

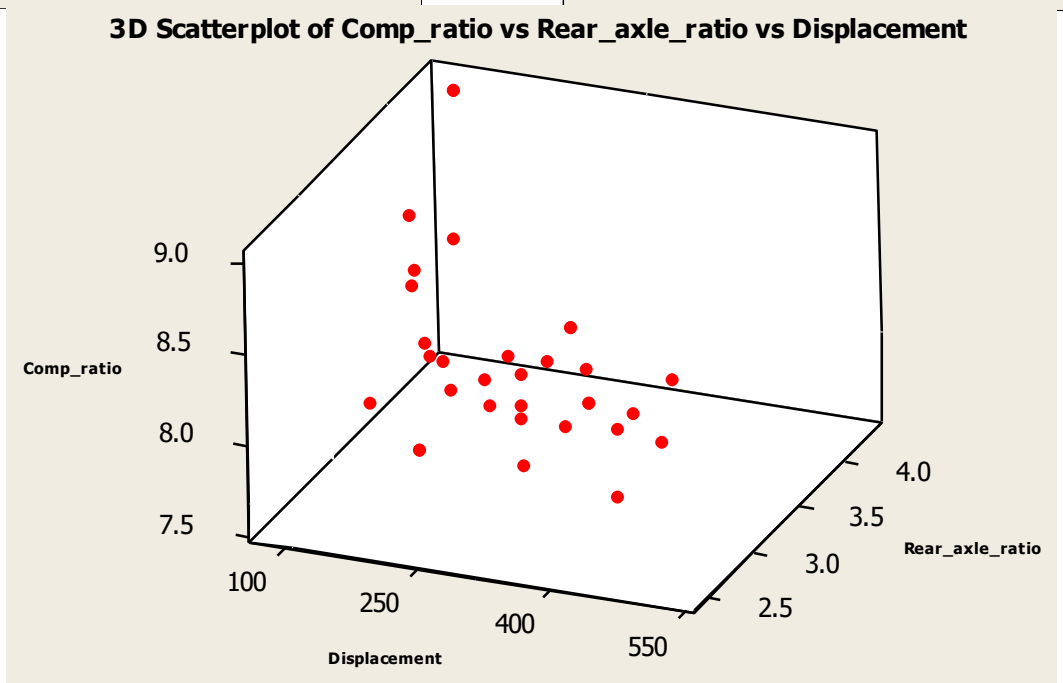
$k = 1$



$k = 2$



$k = 3$



K-D Tree Algorithm

- Given a new case $\boldsymbol{x}_0$ and n training observation in k-dimensional space, finding the nearest neighbor $\boldsymbol{x}_{i_1(\boldsymbol{x}_0)}$ is $O(kn)$ if you use a brute-force search. Likewise, finding the nearest neighbor for N new cases is $O(knN)$
- The k-d tree algorithm is a clever strategy that creates a tree-like partition of the training observation space in $O(kn\log(n))$, and then uses this to find the nearest neighbor of each new $\boldsymbol{x}_0$ in $O(\log(n))$.
- Thus, finding the nearest neighbor for N new cases is $O(kn\log(n)) + O(N\log(n))$
- Google “K-D tree for nearest neighbors” for more info. Finding nearest neighbors in a large data set has many applications outside of supervised learning

Local Weighting and Kernel Smoothing

- Can view the K-NN predictor of $Y(\boldsymbol{x})$ as a weighted average of "nearby" observations with an abruptly decaying weighting function (1 if $\boldsymbol{x}_i \in N_K(\boldsymbol{x})$; 0 otherwise):

$$\hat{y}(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{x}_i \in N_K(\boldsymbol{x})} y_i}{K}$$

where $N_K(\boldsymbol{x})$ = neighborhood defined by K closest points to $\boldsymbol{x}$

- We might expect a better predictor if we use a weighted average of nearby observations with smoothly decaying weighting function:

$$\hat{y}(\boldsymbol{x}) = \frac{\sum_{i=1}^{n} K_\lambda(\boldsymbol{x},\boldsymbol{x}_i) y_i}{\sum_{i=1}^{n} K_\lambda(\boldsymbol{x},\boldsymbol{x}_i)}$$

where λ is a parameter chosen to control neighborhood size, and $K_\lambda(\boldsymbol{x}, \boldsymbol{x}_i)$ is a kernel weighting function

Abrupt (K-NN) Vs. Smooth Weighting

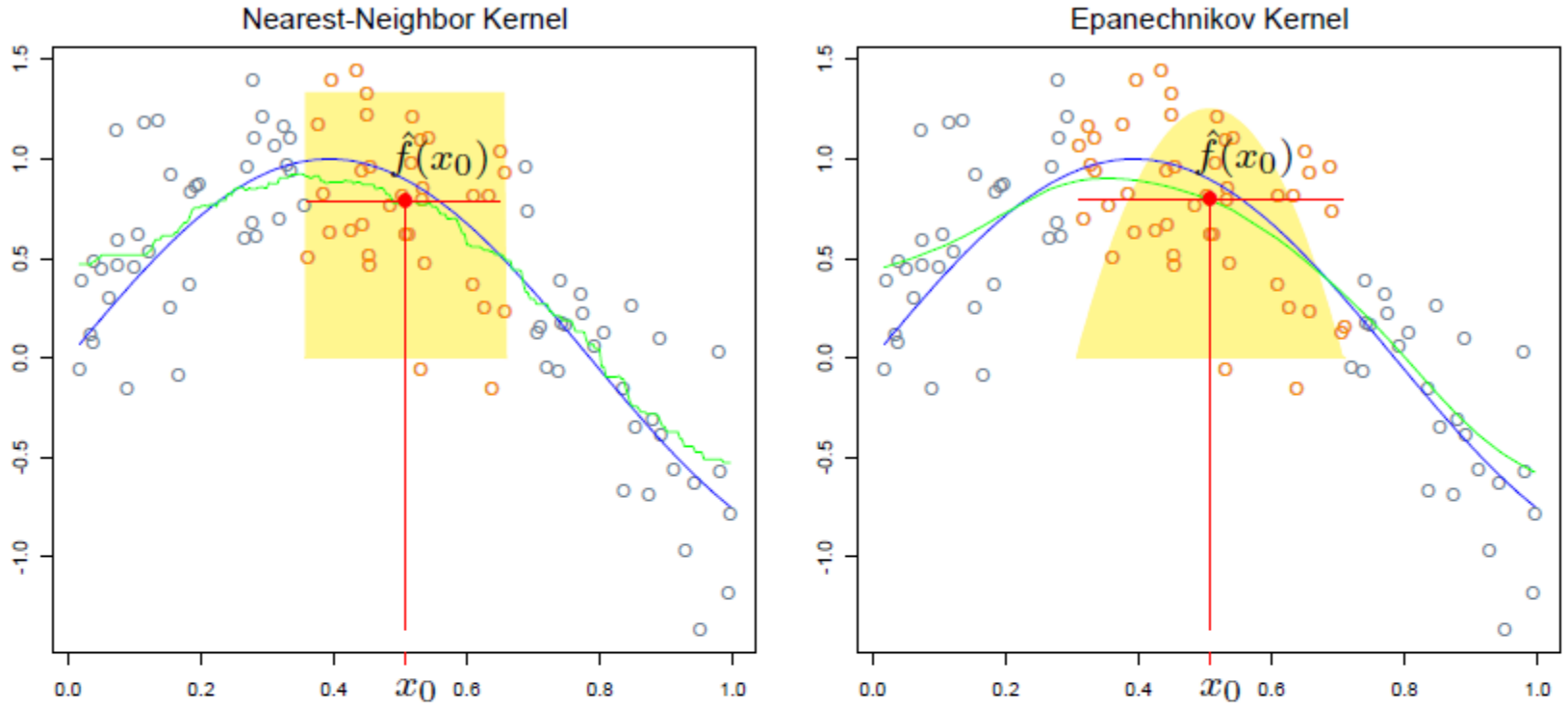


FIGURE 6.1. *In each panel* 100 *pairs* x_i, y_i *are generated at random from the blue curve with Gaussian errors:* $Y = \sin(4X) + \varepsilon$, $X \sim U[0,1]$, $\varepsilon \sim N(0, 1/3)$. *In the left panel the green curve is the result of a* 30*-nearest-neighbor running-mean smoother. The red point is the fitted constant* $\hat{f}(x_0)$, *and the red circles indicate those observations contributing to the fit at* x_0. *The solid yellow region indicates the weights assigned to observations. In the right panel, the green curve is the kernel-weighted average, using an Epanechnikov kernel with (half) window width* $\lambda = 0.2$.

Three Common Kernel Weighting Functions

Gaussian kernel: $$K_\lambda(\mathbf{x},\mathbf{x}_i) = exp\left(-\left[\frac{\|\mathbf{x}-\mathbf{x}_i\|}{\lambda}\right]^2\right)$$

Epanechnikov quadratic kernel:

$$K_\lambda(\mathbf{x},\mathbf{x}_i) = \begin{cases} \frac{3}{4}\left(1-\left[\frac{\|\mathbf{x}-\mathbf{x}_i\|}{\lambda}\right]^2\right): & \|\mathbf{x}-\mathbf{x}_i\| \le \lambda \\ 0: & \text{otherwise} \end{cases}$$

Tri-cube kernel: $$K_\lambda(\mathbf{x},\mathbf{x}_i) = \begin{cases} \left(1-\left[\frac{\|\mathbf{x}-\mathbf{x}_i\|}{\lambda}\right]^3\right)^3 : & \|\mathbf{x}-\mathbf{x}_i\| \le \lambda \\ 0: & \text{otherwise} \end{cases}$$

Plot of Three Common Kernel Functions

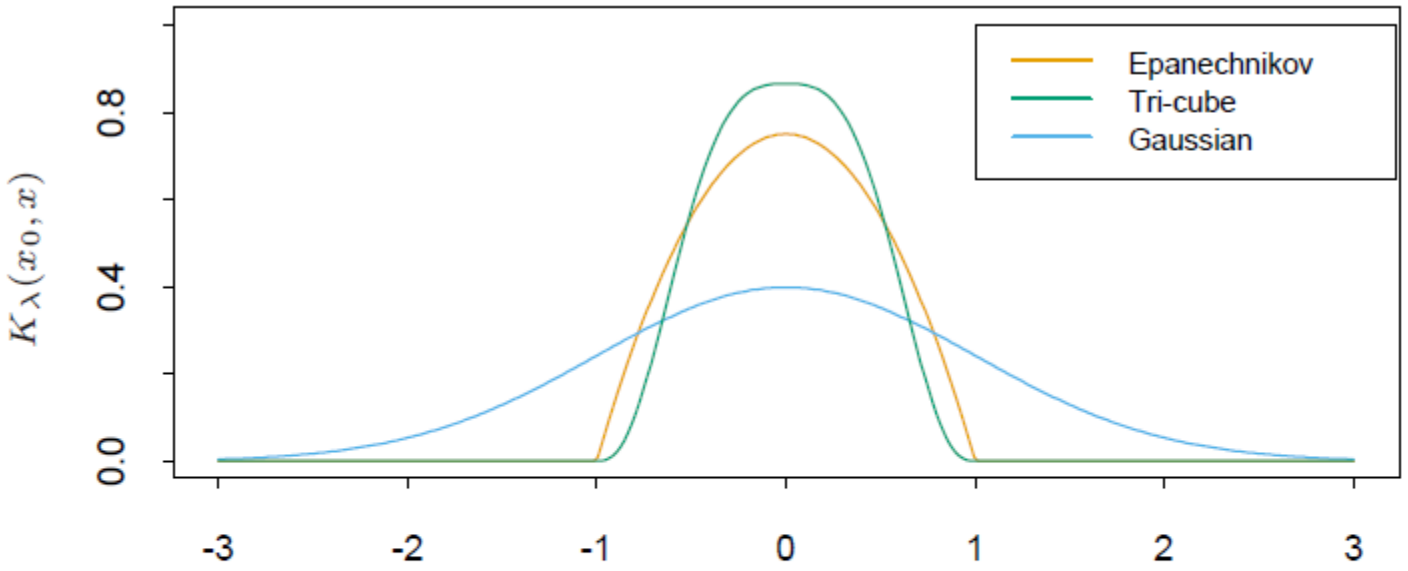


FIGURE 6.2. *A comparison of three popular kernels for local smoothing. Each has been calibrated to integrate to* 1. *The tri-cube kernel is compact and has two continuous derivatives at the boundary of its support, while the Epanechnikov kernel has none. The Gaussian kernel is continuously differentiable, but has infinite support.*

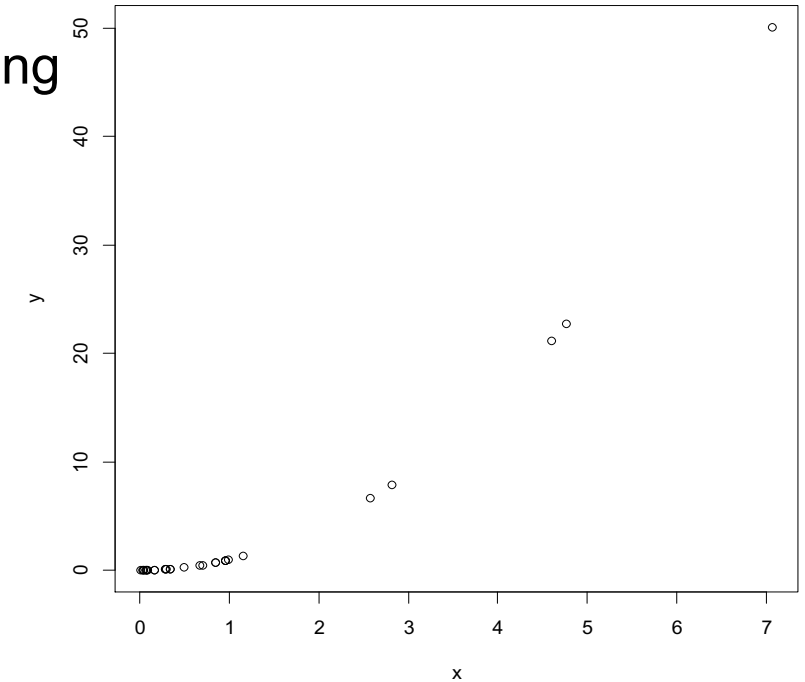
- In the above plot, the vertical axis is $K_\lambda(\mathbf{x},\mathbf{x}_i)$, and the horizontal axis is $\pm\|\mathbf{x}-\mathbf{x}_i\|/\lambda$ as $\mathbf{x}_i$ deviates from $\mathbf{x}$

Discussion Points and Questions

- The Gaussian kernel weights decay more smoothly, but the other two “compact support” kernels have computational advantages. Why?
- How would increasing λ change the predictor plotted three slides earlier?
- Regarding the omnipresent bias/variance tradeoff, would increasing λ
  - increase or decrease the variance of the predictor?
  - increase or decrease the bias of the predictor?
- λ is a tuning parameter that you choose. How would you choose it?

Comparison of Adaptive Neighborhood Sizes

- In addition to having a more smoothly decaying weighting function relative to K-NN, another difference with kernel weighting is that in regions of sparser data:
  - the K-NN neighborhood size grows larger (giving larger potential bias but roughly constant variance)
  - the kernel neighborhood size remains the same (giving roughly constant bias but larger variance)
  - for the data set shown in the following scatterplot of y vs x, which seems more appropriate?



- This boils down to the familiar bias-variance tradeoff?

A Problem with N-W Kernel-Weighted Average (and also with Nearest Neighbors)

- The kernel-weighted average can have severe bias at the boundaries of the $\mathbf{x}$-domain
- In high dimensional space (large k), points are inherently closer to the boundaries.

In one dimension, 2 out of 9 points are on the boundary

But in two dimensions, 8 out of 9 are on the boundary

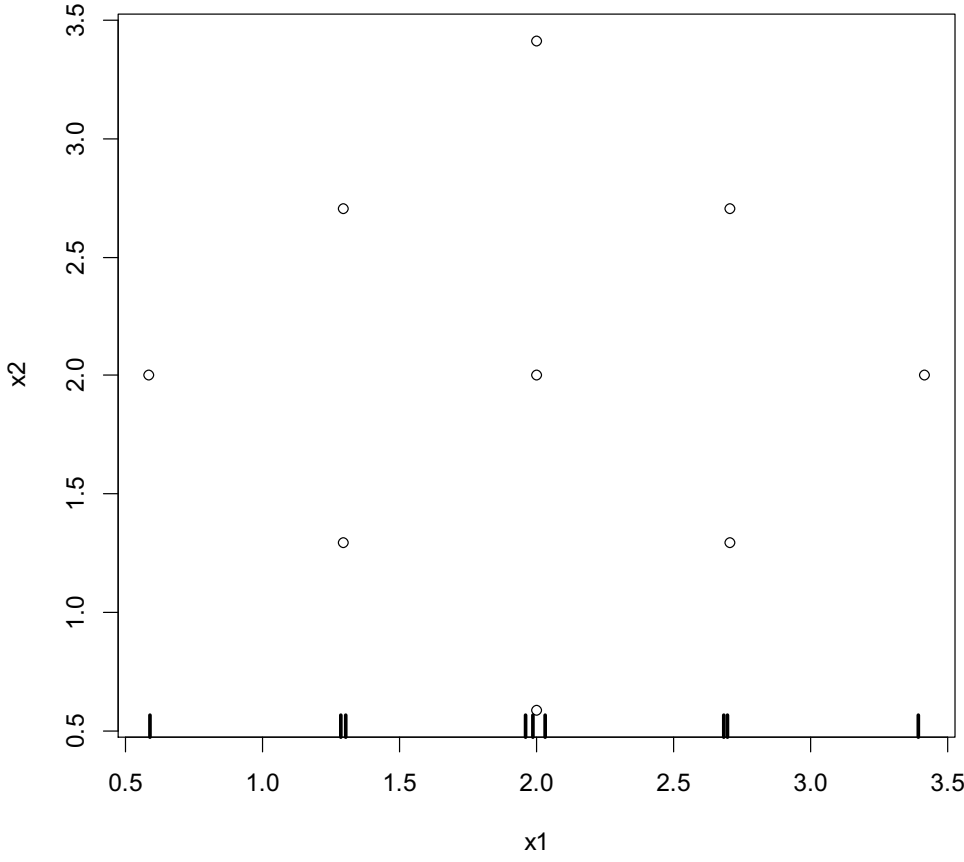


Illustration of Boundary Bias for Kernel Average

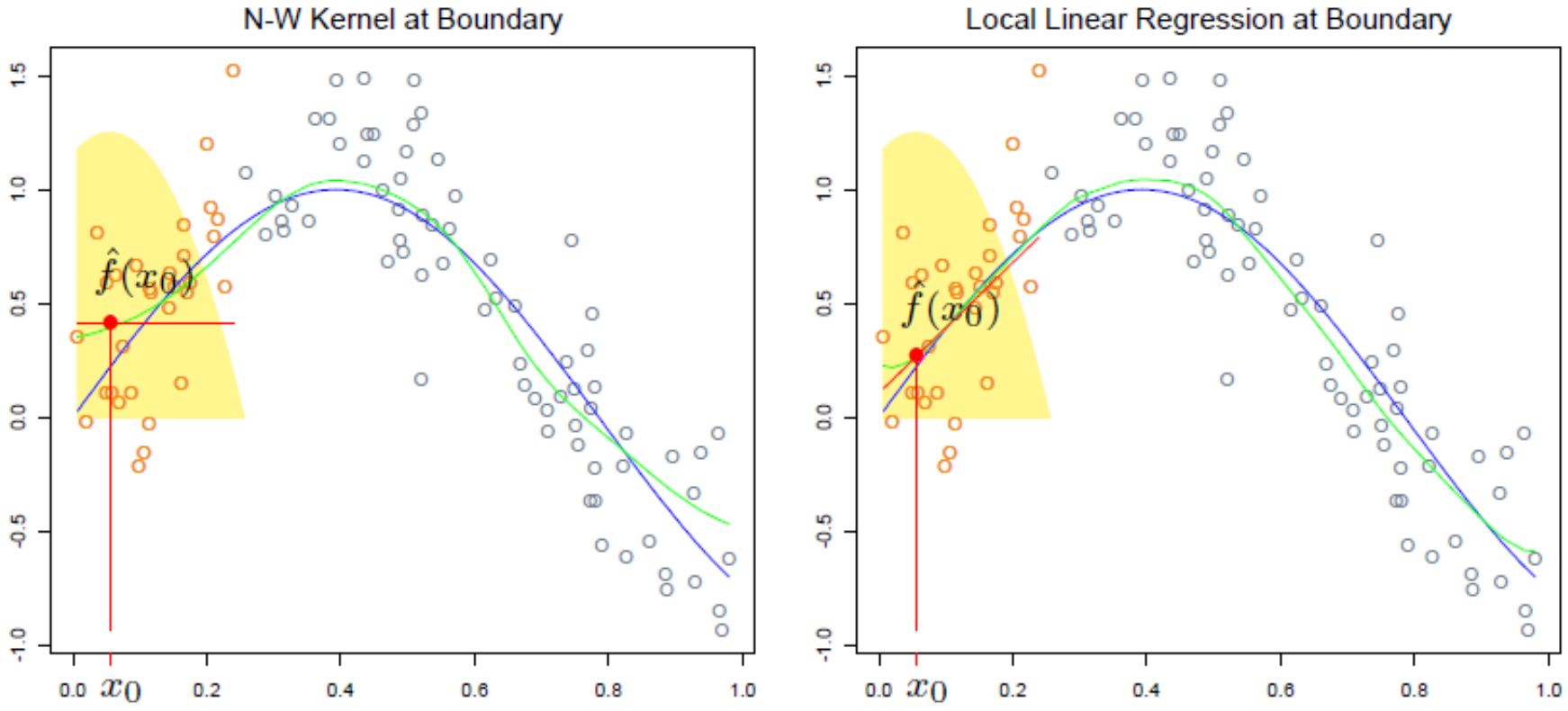


FIGURE 6.3. *The locally weighted average has bias problems at or near the boundaries of the domain. The true function is approximately linear here, but most of the observations in the neighborhood have a higher mean than the target point, so despite weighting, their mean will be biased upwards. By fitting a locally weighted linear regression (right panel), this bias is removed to first order*

Solution: Local Linear Regression

- Instead of using a local average for the predictor, fit a local linear regression model:

To predict Y at specific $\boldsymbol{x}$, find

$$\widehat{\boldsymbol{\beta}}(\mathbf{x}) = argmin_{\boldsymbol{\beta}} \sum_{i=1}^{n} K_{\lambda}(\boldsymbol{x}, \boldsymbol{x}_i)[y_i - \boldsymbol{\beta}^T \boldsymbol{x}_i]^2$$
$$= argmin_{\boldsymbol{\beta}} \sum_{i=1}^{n} K_{\lambda}(\boldsymbol{x}, \boldsymbol{x}_i)[y_i - (\beta_0 + \beta_1 x_{i1} + \cdots + \beta_k x_{ik})]^2$$

and use the predictor $\hat{y}(\boldsymbol{x}) = \left(\widehat{\boldsymbol{\beta}}(\mathbf{x})\right)^T \boldsymbol{x}$

- For each $\boldsymbol{x}$ at which you want to predict Y, you must refit the model and minimize the above weighted SSE, which will give a different set of coefficients $\widehat{\boldsymbol{\beta}}(\mathbf{x})$ for **every** $\boldsymbol{x}$

Discussion Points and Questions

- A local linear regression model is nonparametric, just like kernel-weighted average and nearest neighbors. There is no single fitted model – the coefficients $\hat{\boldsymbol{\beta}}(\mathbf{x})$ are different for each $\mathbf{x}$ at which you want to predict Y
- What is the relationship between a local linear regression model and a linear regression model? Is a local linear regression model "almost" linear?
- Where is a local linear regression predictor most likely to be biased?

Local Linear Vs. Local Quadratic Regression

- Local linear regression effectively removes linear boundary bias, but is still subject to bias in regions of high curvature
  - Local quadratic regression will remove much of the bias in regions of high curvature, but also increases the variance of the predictor
  - Local polynomials of higher order than 2 will remove even more of the bias, but are almost never used

Local Linear Vs. Local Quadratic Regression

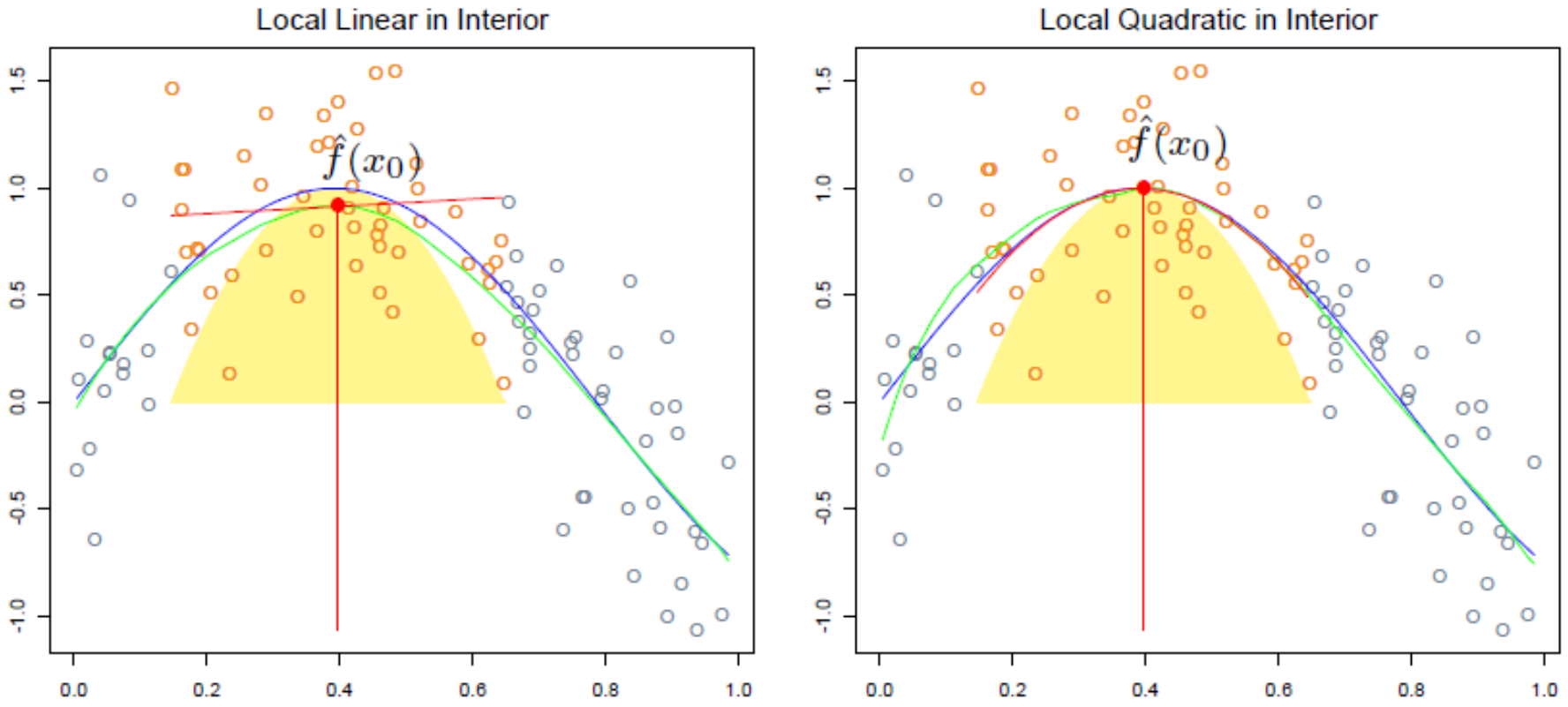


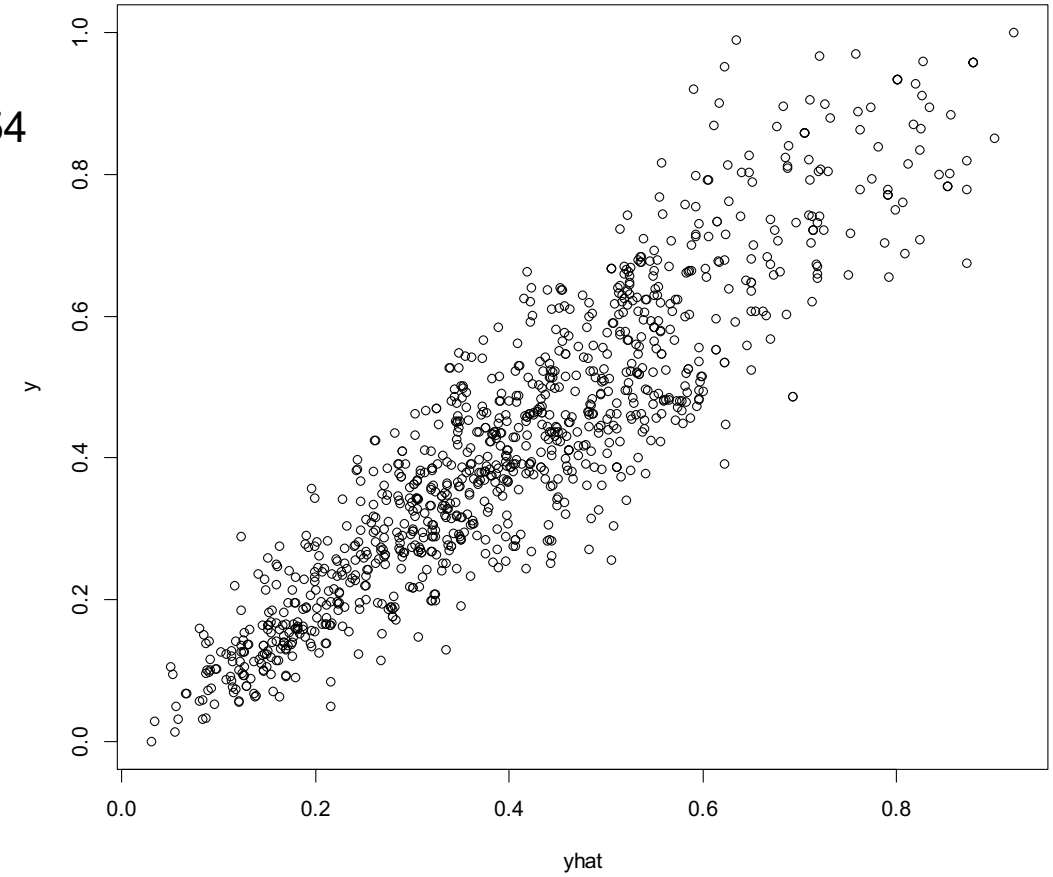
FIGURE 6.5. *Local linear fits exhibit bias in regions of curvature of the true function. Local quadratic fits tend to eliminate this bias.*

- What are the pros and cons of using a local linear, versus local quadratic, model for the above data?

Local Regression for Concrete data

```
CRT<-read.csv("concrete.csv", header=TRUE)
CRT1<-CRT
CRT1[1:8]<-sapply(CRT1[1:8], function(x) (x-mean(x))/sd(x)) #standardize predictors
CRT1[9]<-(CRT1[9]-min(CRT1[9]))/(max(CRT1[9])-min(CRT1[9]))
out<-loess(Strength~., CRT1[,c(1,2,4,8,9)],degree=1,span=.3)
summary(out)
names(out)
y<-CRT1[[9]]
yhat<-predict(out)
SSE<-sum((y-yhat)^2);SSE
1-var(y-yhat)/var(y)
plot(yhat,y)
```

```
> summary(out)
Call:
loess(formula = Strength ~ ., data = CRT1[, c(1, 2, 4, 8, 9)],
    span = 0.3, degree = 1)

Number of Observations: 1030
Equivalent Number of Parameters: 21.54
Residual Standard Error: 0.089
Trace of smoother matrix: 32.38
Control settings:
  normalize:  TRUE
  span      :  0.3
  degree    :  1
  family    :  gaussian
  surface   :  interpolate         cell = 0.2

> 1-var(y-yhat)/var(y)
[1] 0.8285685
```



Choosing the best λ

- Can use cross-validation (left as an exercise), or
- (arguably the best method) Can choose the λ that minimizes a version of Mallow's C_p adapted to the case of kernel regression:

$$C_p = \frac{SSE}{n} + \frac{2trace(\mathbf{S})\hat{\sigma}^2}{n}$$

where the smoother matrix $\mathbf{S}$ is such that $\hat{\mathbf{Y}} = \mathbf{SY}$, and the low-bias error variance estimate is $\hat{\sigma}^2 = \frac{SSE}{\text{trace}\{(\mathbf{I}-\mathbf{S})^T(\mathbf{I}-\mathbf{S})\}}$ for some relatively small λ

- This version of C_p applies for any predictor that has the form $\hat{\mathbf{y}} = \mathbf{Sy}$, e.g., ridge regression (for which $\mathbf{S} = \mathbf{X}[\mathbf{X}^T\mathbf{X}+\lambda\mathbf{I}]^{-1}\mathbf{X}^T$), kernel regression, etc.

Using C_p to choose λ for Concrete example

```
##first find sigma_hat for a low-bias model###
for (lambda in seq(.02,.2,.02)) {out<-loess(Strength ~., CRT1[, c(1,2,4,8,9)],degree=1,
    span=lambda); print(c(lambda,out$s))}
[1] 0.0200000 0.5845109
[1] 0.0400000 0.1251313
[1] 0.06000000 0.09536982    Choose about this as the low-bias estimate of sigma
[1] 0.0800000 0.1023739
[1] 0.10000000 0.08853284
[1] 0.12000000 0.08879249
[1] 0.1400000 0.1034087
[1] 0.16000000 0.09696377
[1] 0.18000000 0.09403151
[1] 0.20000000 0.09007899

sig_hat<-0.1
```

Using C_p to choose λ for Concrete example

```
##now find Cp for various lambda###
for (lambda in c(seq(.01,.05,.01), seq(.1,1,.2))) {out<-loess(Strength ~., CRT1[,
    c(1,2,4,8,9)],degree=1, span=lambda); SSE<-sum((CRT1[,9]-out$fitted)^2); Cp <-
    (SSE+2*out$trace.hat*sig_hat^2)/nrow(CRT1); print(c(lambda,Cp))}
[1]  0.01000 71.52311
[1] 0.0200000 0.2694297
[1] 0.0300000 0.0978872
[1] 0.04000000 0.01558056
[1] 0.05000000 0.01136318
[1] 0.100000000 0.008515458
[1] 0.300000000 0.008216944
[1] 0.500000000 0.009435424
[1] 0.70000000 0.01025778
[1] 0.90000000 0.01112647
```

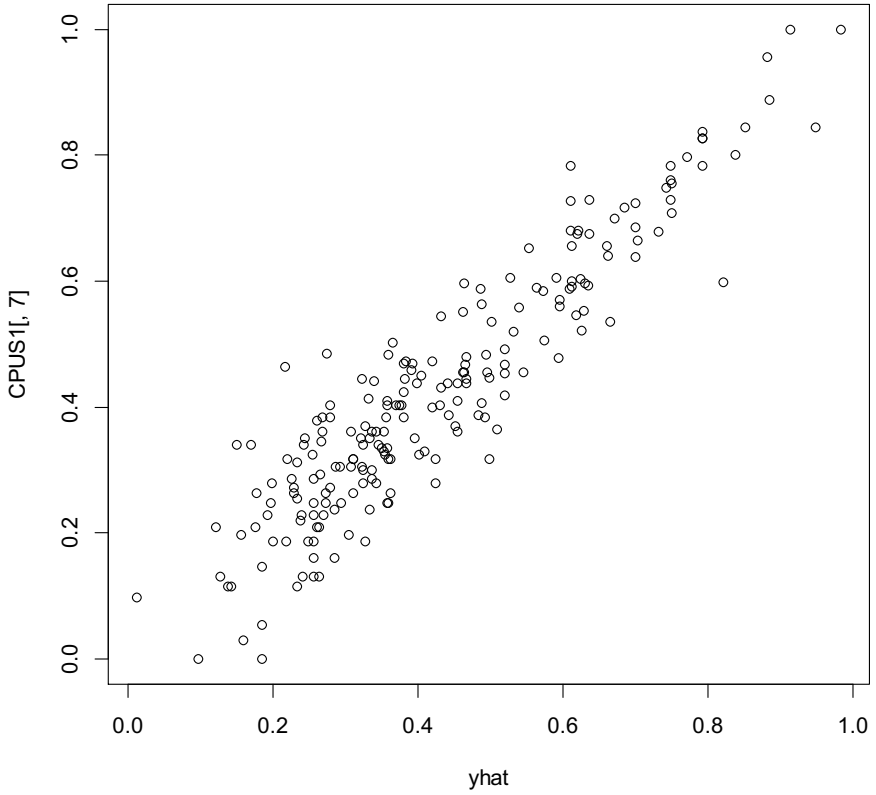
lambda = 0.3 is about the best

- An estimate of the corresponding test error SD is sqrt(Cp) = sqrt(0.00822) = 0.0906
- Exercise: Compare this λ and error SD with those from CV

Local Regression for CPUS data

```
CPUS<-read.table("cpus.txt",sep="\t")
CPUS1<-CPUS[2:8]
CPUS1[c(1:3,7)]<-sapply(CPUS1[c(1:3,7)], log10)  #take log of first three predictors and response
CPUS1[1:6]<-sapply(CPUS1[1:6], function(x) (x-mean(x))/sd(x)) #standardize predictors
CPUS1[7]<-(CPUS1[7]-min(CPUS1[7]))/(max(CPUS1[7])-min(CPUS1[7]))
out<-loess(perf~.,CPUS1[,c(1:4,7)],degree=1,span=.6)
summary(out)
names(out)
y<-CPUS1[[7]]
yhat<-predict(out)
SSE<-sum((y-yhat)^2);SSE
1-var(y-yhat)/var(y)
plot(yhat,y)
```

```
> summary(out)
Call:   loess(formula = perf ~ ., data = CPUS1[, c(1:4, 7)], span = 0.6,
    degree = 1)

Number of Observations: 209
Equivalent Number of Parameters: 9.89
Residual Standard Error: 0.07711
Trace of smoother matrix: 14.31

Control settings:
  normalize:  TRUE
  span      :  0.6
  degree    :  1
  family    :  gaussian
  surface   :  interpolate         cell = 0.2

> 1-var(y-yhat)/var(y)
[1] 0.8634346
```



Using C_p to choose λ for CPUS example

```
##first find sigma_hat for a low-bias model###
for (lambda in seq(.1,1,.1)) {out<-loess(perf~.,CPUS1[,c(1:4,7)],degree=1, span=lambda);
    print(c(lambda,out$s))}
[1] 0.1000000 0.1281004
[1] 0.20000000 0.08491299
[1] 0.30000000 0.07729764
[1] 0.40000000 0.07617725
[1] 0.50000000 0.07632685
[1] 0.60000000 0.07711136
[1] 0.70000000 0.07889357
[1] 0.80000000 0.07855612
[1] 0.90000000 0.07884745
[1] 1.00000000 0.08208199

sig_hat<-0.0773
```

chose this as the estimate of sigma for a low-bias model

Using C_p to choose λ for CPUS example

```
##now find Cp for various lambda###
for (lambda in seq(.1,1,.1)) {out<-loess(perf~.,CPUS1[,c(1:4,7)],degree=1, span=lambda);
    SSE<-sum((CPUS1[,7]-out$fitted)^2); Cp <-
    (SSE+2*out$trace.hat*sig_hat^2)/nrow(CPUS1); print(c(lambda,Cp))}
[1] 0.10000000 0.01467463
[1] 0.200000000 0.007535495
[1] 0.300000000 0.006428755
[1] 0.400000000 0.006216846
[1] 0.500000000 0.006165853
[1] 0.60000000 0.00623147
[1] 0.700000000 0.006453386
[1] 0.800000000 0.006383156
[1] 0.900000000 0.006405543
[1] 1.000000000 0.006864742
```

the best lambda is 0.5

- An estimate of the corresponding test error SD is sqrt(Cp) = sqrt(.00617) = 0.0785
- Exercise: Compare this λ and error SD with those from CV

Local Kernel Weighting For Classification

- Very similar to local regression, except that instead of fitting a local model to minimize the locally weighted SSE, maximize the locally weighted log-likelihood for some local probabilistic class model with parameters $\boldsymbol{\beta}(\mathbf{x})$:

$$\hat{\boldsymbol{\beta}}(\mathbf{x}) = arg\,max_{\boldsymbol{\beta}} \sum_{i=1}^{n} K_{\lambda}(\mathbf{x}, \mathbf{x}_i) log\, f(y_i; \boldsymbol{\beta})$$

- To classify Y at a specific $\mathbf{x}$, estimate the class probabilities based on the assumed local model structure with parameters $\hat{\boldsymbol{\beta}}(\mathbf{x})$

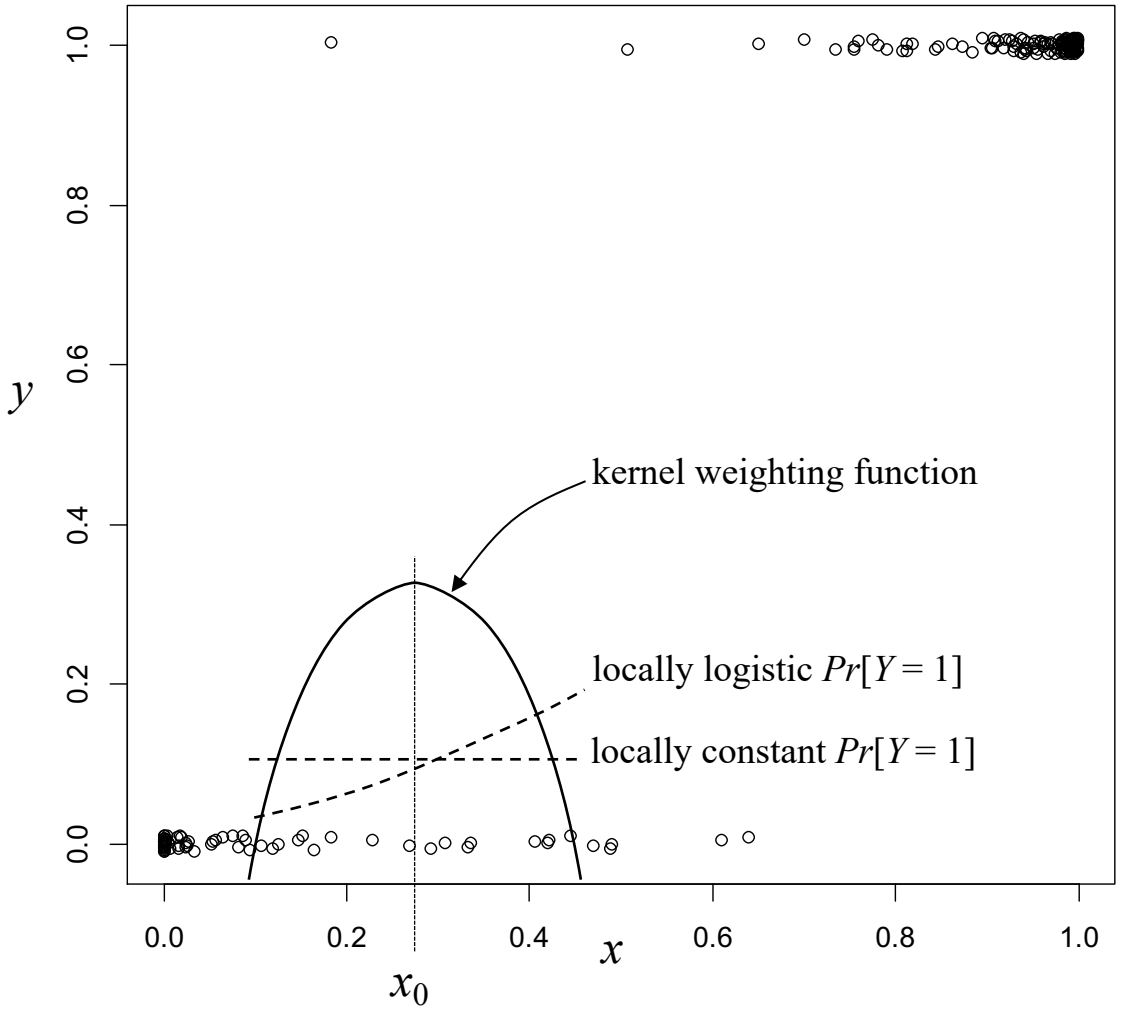
What to Use for Locally Constant Class Probabilities

- For locally constant class probabilities, the local likelihood to use in the previous slide is obtained from the simple model $Pr[Y_i \in \text{Class } j] = p_j$ (constant for $\mathbf{x}_i$ in neighborhood of $\mathbf{x}$), i.e., a local intercept-only logistic regression model
- Plugging this local likelihood into the equation on the previous slide and minimizing to estimate $\beta(\mathbf{x})$ is equivalent to taking the predicted class probabilities to be the locally weighted fraction of observations in each class:

$$\hat{p}_j(\mathbf{x}) = \frac{\sum_{i=1}^{n} K_\lambda(\mathbf{x}, \mathbf{x}_i) I[y_i \in Class\ j]}{\sum_{i=1}^{n} K_\lambda(\mathbf{x}, \mathbf{x}_i)}$$

Locally Constant Vs. Locally Logistic $Pr[Y \in \text{Class } j]$

scatterplot of binary y vs a single predictor x



What to Use for Local Binary Logistic Regression Class Probabilities

- With assumed local binary logistic regression class probabilities:

$$Pr[Y_i = 1] = \frac{exp\{\boldsymbol{\beta}^T \mathbf{x}_i\}}{1 + exp\{\boldsymbol{\beta}^T \mathbf{x}_i\}}$$

($\boldsymbol{\beta}$ constant for $\mathbf{x}_i$ in neighborhood of $\mathbf{x}$)

The estimated class probabilities are obtained by plugging $\mathbf{x}$ into a logistic regression model with coefficients $\hat{\boldsymbol{\beta}}(\mathbf{x})$ from a locally weighted logistic regression fit:

$$\hat{p}(\mathbf{x}) = \frac{exp\{\hat{\boldsymbol{\beta}}^T(\mathbf{x})\mathbf{x}\}}{1 + exp\{\hat{\boldsymbol{\beta}}^T(\mathbf{x})\mathbf{x}\}}$$

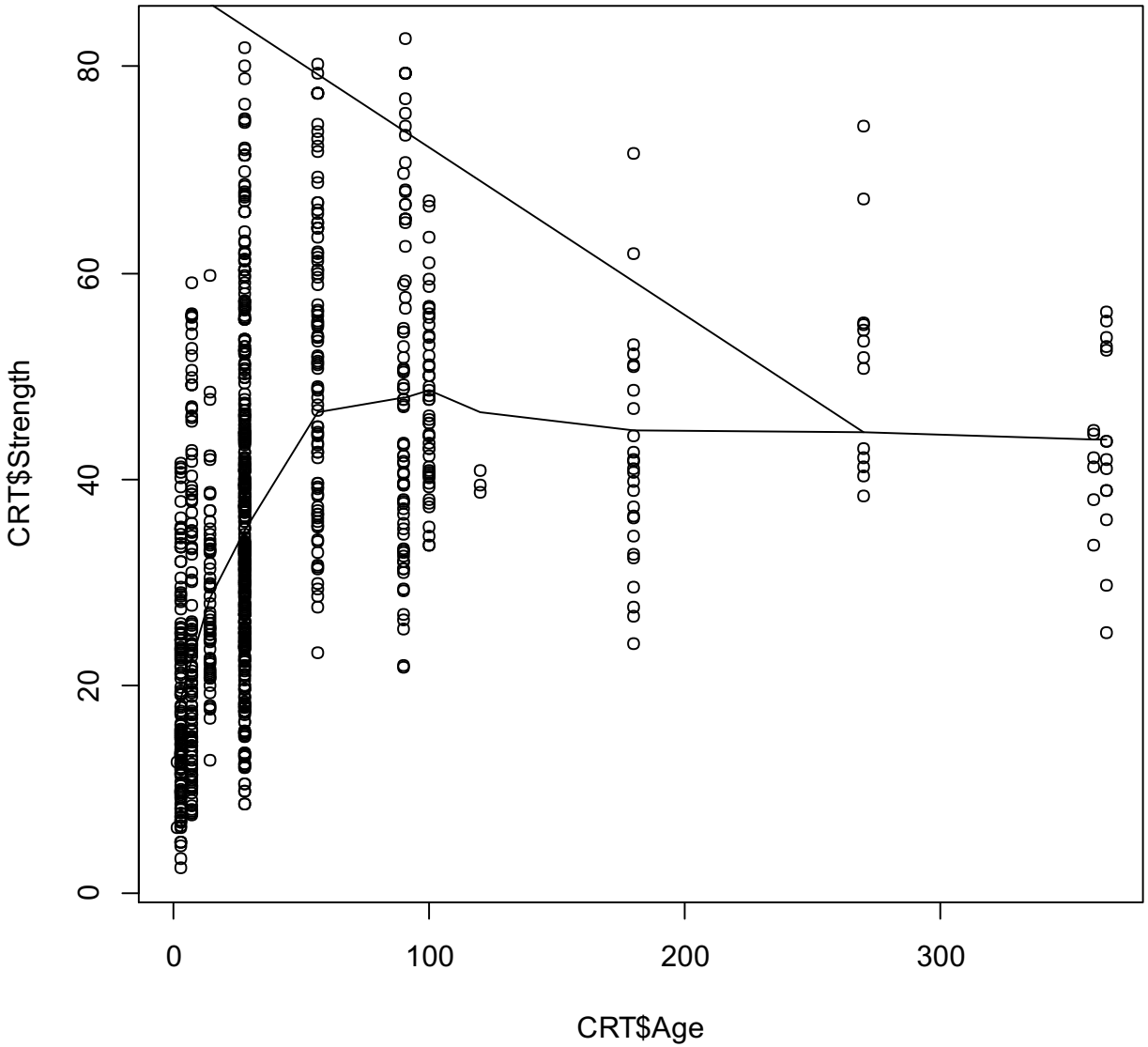
Summary of Local Methods (K-NN or Kernel)

- No assumed structure and completely nonparametric, so very flexible and avoids specifying the "wrong" model structure (pro), but poorer bias/variance tradeoff (con)
- Computational/Memory: No training step (pro), but must fit a new local model for each case to predict (con) and retain all of the original "training" data in memory (con)
- Comp. expense of K-fold CV independent of K (pro) and can be fast using the same K-NN computational tricks
- No good way to handle categorical predictors, especially for kernel methods
- Not well suited for high dimensions
  - Points are inherently sparser in high dimensions
  - Originally intended as a scatterplot smoother for visualization [which accounts for the ss in loess(), which allows at most 4 predictors]
  - Additive nonparametric models [ppr() and gam() in mgcv package in R] borrow ideas from local kernel regression, but are better suited for high dimensions

lowess (similar to loess) as a Scatterplot Smoother

```
plot(CRT$Age, CRT$Strength)
lines(lowess(CRT$Age,
      CRT$Strength))
```

It is easier to visualize a trend in a scatter plot if a loess curve is superimposed



Generalized Additive Models (GAMs)

- Local methods work well in relatively low dimensions, but break down in higher dimension (sparser neighbors)
- For higher dimensional data, generalized additive models model the response as the sum of functions of one or two predictors at a time, e.g. (for k predictors):

$$\text{GAM: } Y(\mathbf{x}) = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_k(x_k) + \varepsilon$$

$$\text{or sometimes } + \sum_{j=1}^{k} \sum_{l=j+1}^{k} f_{j,l}(x_j, x_l)$$

- Each $f_j(\bullet)$ or $f_{j,l}(\bullet)$ function is estimated nonparametrically, typically using an iterative backfitting algorithm, until convergence

Backfitting Algorithm (without interactions $f_{j,l}(\bullet)$)

0) Initialize $\hat{\alpha} = \frac{1}{n}\sum_{i=1}^{n} y_i$ and $\hat{f}_j(x_j) = 0$ for all j

1) For $j = 1, 2, \ldots, k$,

   i) Calculate errors $e_i = y_i - \hat{\alpha} - \sum_{l \neq j} \hat{f}_l(x_{il})$ $(i = 1, 2, \ldots, n)$

   ii) Fit nonparametric $\hat{f}_j(x_j)$ to $\{e_i \; vs. \; x_{ij}: \; i = 1, 2, \ldots, n\}$

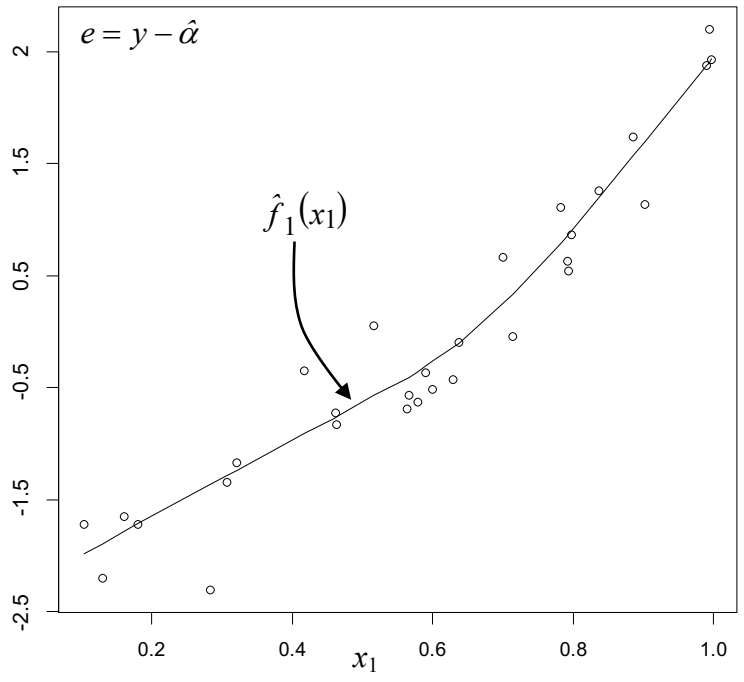
   iii) Center $\hat{f}_j(x_j) \leftarrow \hat{f}_j(x_j) - \frac{1}{n}\sum_{i=1}^{n} \hat{f}_j(x_{ij})$

2) Repeat Step 1 until convergence, i.e., no further changes in the $\hat{f}_j(x_j)$'s

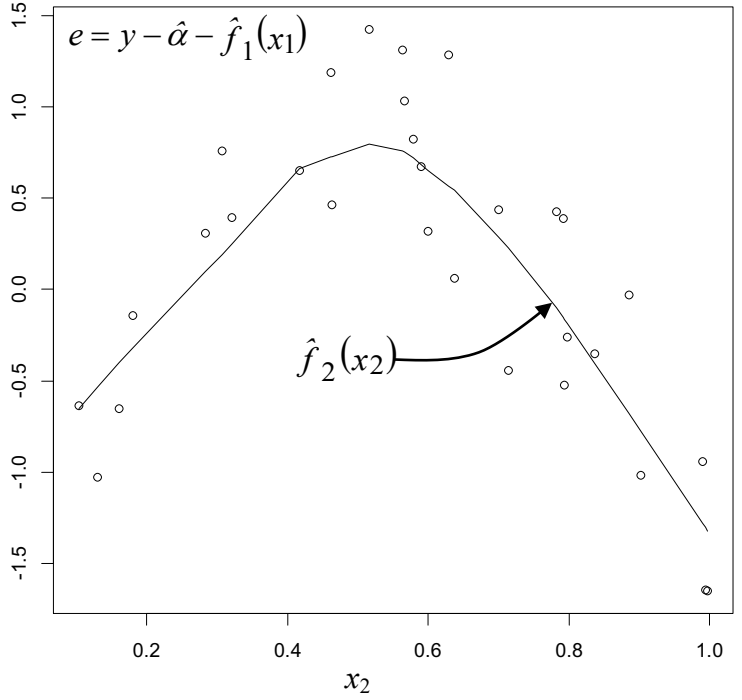
The function in Step 1-ii can be any nonparametric smoother (e.g., local regression, smoothing spline, etc)

Illustration of Backfitting Step 1 on the First Iteration

Step 1 for j = 1



Step 1 for j =2



Discussion Points and Questions

- With no interactions, what are the implications of the GAM model structure $Y(\mathbf{x}) = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_k(x_k) + \varepsilon$, in terms of the type of predictive relationships it can capture?
- What types of relationships is it incapable of capturing?
- How would you handle categorical predictors in a GAM model?

Projection Pursuit Regression (PPR)

- If the relationship is not truly additive in the predictors, including interaction terms in a GAM can capture this, but this is inefficient for large k (~ $k^2/2$ interaction terms)
- PPR is similar to GAMs, but instead of each function being a 1-D function of one predictor, allow it to be a 1-D function of some linear combination of all predictors:

$$\text{PPR model: } Y(\mathbf{x}) = \alpha + f_1(\boldsymbol{\beta}_1{}^T\mathbf{x}) + f_2(\boldsymbol{\beta}_2{}^T\mathbf{x}) + \ldots + f_M(\boldsymbol{\beta}_M{}^T\mathbf{x}) + \varepsilon$$

$$= \alpha + f_1(v_1) + f_2(v_2) + \ldots + f_M(v_M) + \varepsilon$$

$$\text{where } v_j = \boldsymbol{\beta}_j{}^T\mathbf{x} = \beta_{j1}x_1 + \beta_{j2}x_2 + \ldots + \beta_{jk}x_k \quad \text{(linear combo of } x\text{'s)}$$

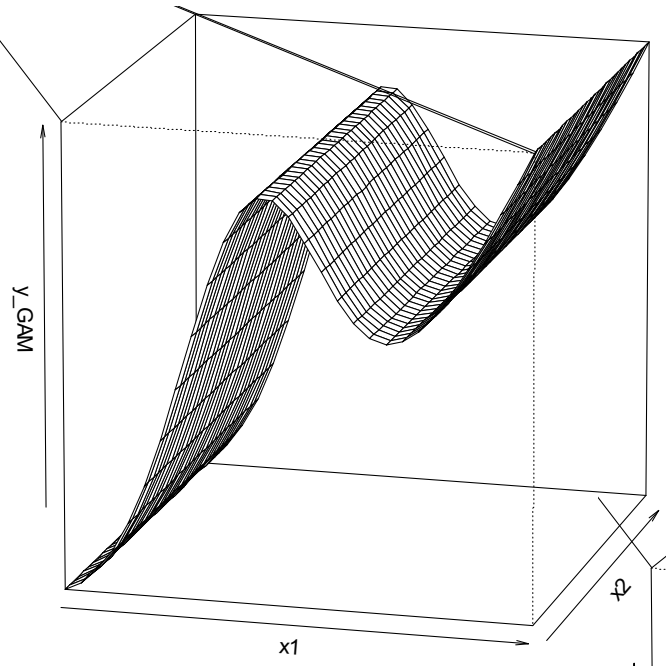
- You specify M (or determine it via CV), and the fitting algorithm estimates the functions $f_j(\bullet)$ nonparametrically and also estimates the direction vectors $\boldsymbol{\beta}_j$

Discussion Points and Questions

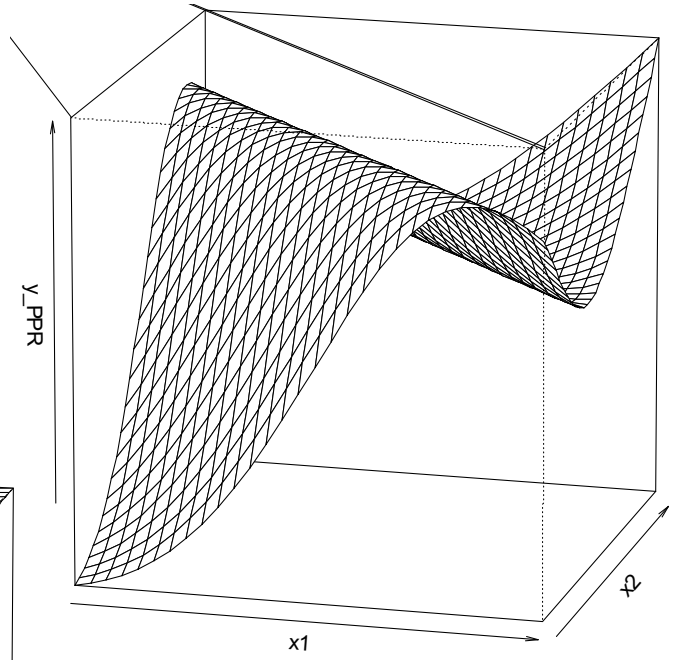
- Neural networks with a linear output activation function and one hidden layer are very similar to PPR, but with the neural network logistic $H_j(\bullet)$ replaced by the completely nonparametric PPR $f_j(\bullet)$
- More generally, what is the relationship between GAMs, PPR, and neural networks?
- Which model (GAM, PPR, or neural net) is easiest to interpret and to visualize, in terms of the nature of the predictive response surface and the effects of the predictors?

Individual Constituent Functions for GAM vs. PPR vs. Neural Network

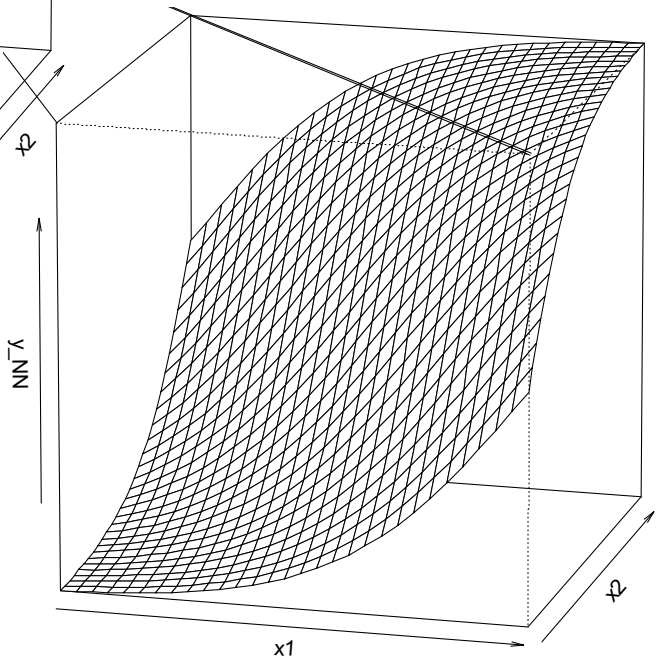
GAM $f_1(\mathbf{x}) = f_1(x_1)$



PPR $f_1(\boldsymbol{\beta}_1{}^T\mathbf{x})$



Neural Net $H_1(\mathbf{x})$



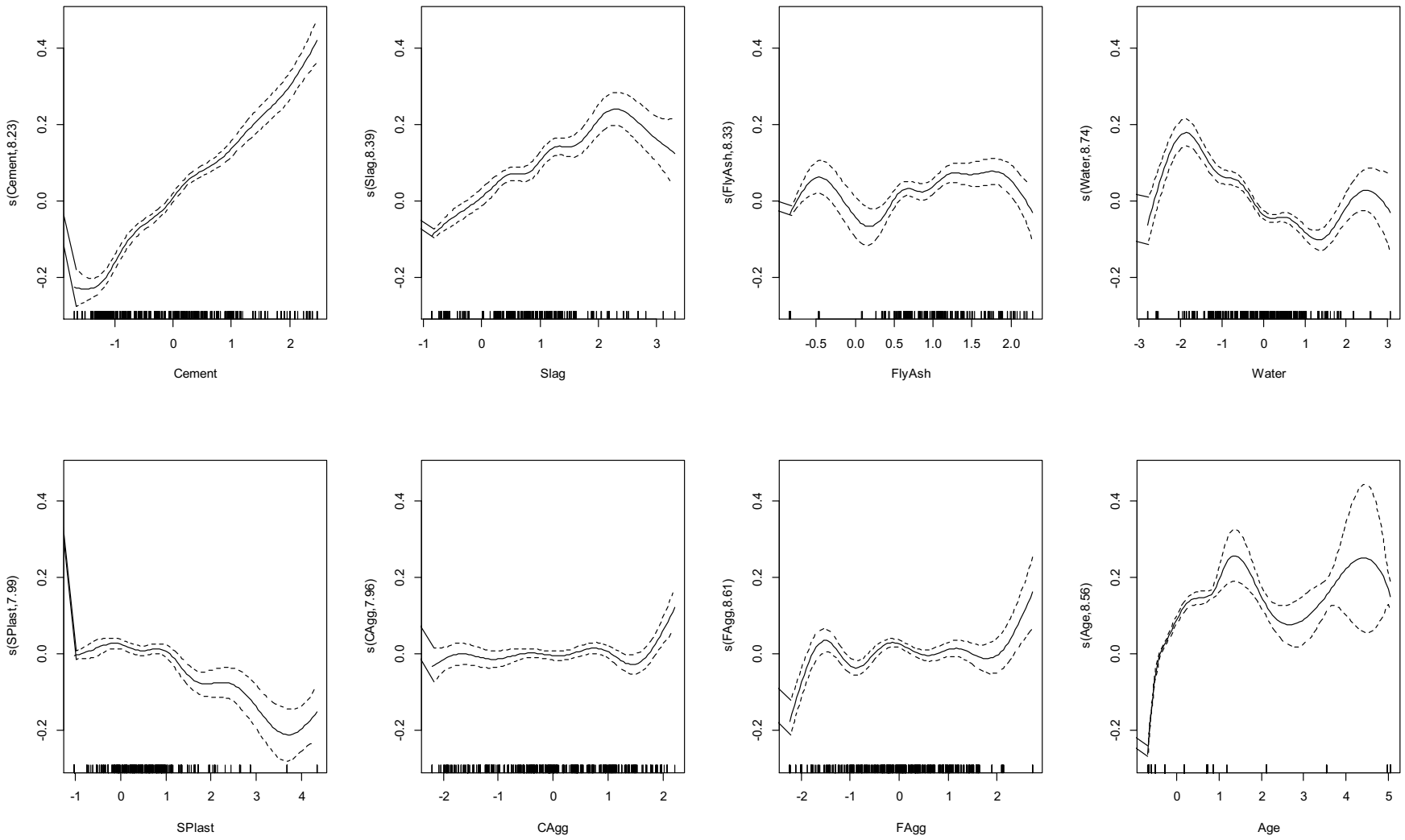
GAM Fit for Concrete data

```
CRT<-read.csv("concrete.csv", header=TRUE); CRT1<-CRT
CRT1[1:8]<-sapply(CRT1[1:8], function(x) (x-mean(x))/sd(x)) #standardize predictors
CRT1[9]<-(CRT1[9]-min(CRT1[9]))/(max(CRT1[9])-min(CRT1[9]))
library(mgcv)  #stands for “Mixed GAM Computation Vehicle”
out<-gam(Strength~s(Cement)+s(Slag)+s(FlyAsh)+s(Water)+ s(SPlast) + s(CAgg) +
      s(FAgg) + s(Age), data=CRT1, family=gaussian(), sp=c(-1,-1,-1,-1,-1,-1,-1,-1))
summary(out)
out$sp  ##estimated smoothing parameters for each constituent function
yhat<-predict(out)
plot(yhat,CRT1$Strength)  #probably quite a bit of overitting
##
par(mfrow=c(2,4))
plot(out)  #plot component functions
```

- If you want to control the degree of smoothing in each constituent function, you can use the k parameter within the s() argument, where k is the d.f. of the smoother. E.g., replacing s(Age) by s(Age, k = 2) in the above would force a smoother $f_4(x_4)$, and using s(Age, k = 20) would be rougher
- You can also control the degree of smoothing use the sp parameter in gam(). sp=c(-1,-1,-1,-1,-1,-1,-1,-1) (or omitting sp, I think) will result in all smoothing parameters being estimated using GCV or something similar. Using sp=c(-1,-1,-1,.1,-1,-1,-1,-1) gives a smoother $f_4(x_4)$, and using Using sp=c(-1,-1,-1,0.000001,-1,-1,-1,-1) is rougher.
- For binary response, use family = “binomial”; for multi-category response, use family = "poisson"

```
> summary(out)
Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.417191   0.002082   200.4   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
             edf Ref.df       F  p-value
s(Cement) 8.228  8.833  48.281  < 2e-16 ***
s(Slag)   8.387  8.874  24.605  < 2e-16 ***
s(FlyAsh) 8.331  8.851   9.714 4.67e-14 ***
s(Water)  8.742  8.974  25.925  < 2e-16 ***
s(SPlast) 7.986  8.713  11.127 4.41e-16 ***
s(CAgg)   7.956  8.702   3.592 0.000266 ***
s(FAgg)   8.614  8.950  18.297  < 2e-16 ***
s(Age)    8.561  8.901 367.214  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =  0.897   Deviance explained = 90.4%
GCV score = 0.004778  Scale est. = 0.0044634  n = 1030
```

Visualizing a GAM by Plotting Component f()'s



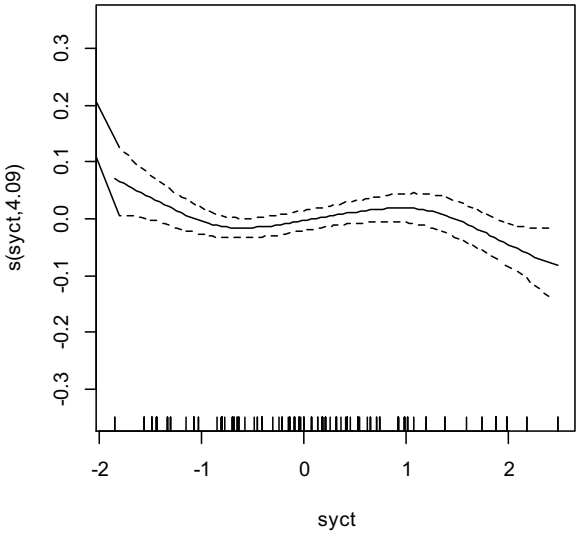
GAM Fit for CPUS data

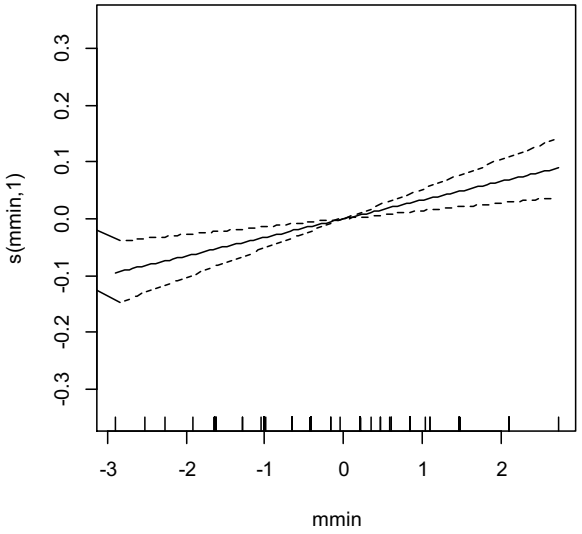
```
library(mgcv)  #stands for “Mixed GAM Computation Vehicle”
out<-gam(perf~s(syct)+s(mmin)+s(mmax)+s(cach)+s(chmin)+s(chmax), data=CPUS1,
      family=gaussian(), sp=c(-1,-1,-1,-1,-1,-1))
summary(out)
out$sp  ##estimated smoothing parameters for each constituent function
yhat<-predict(out)
plot(yhat,CPUS1$perf)  #probably quite a bit of overitting
##
par(mfrow=c(2,3))
plot(out)  #plot component functions
```

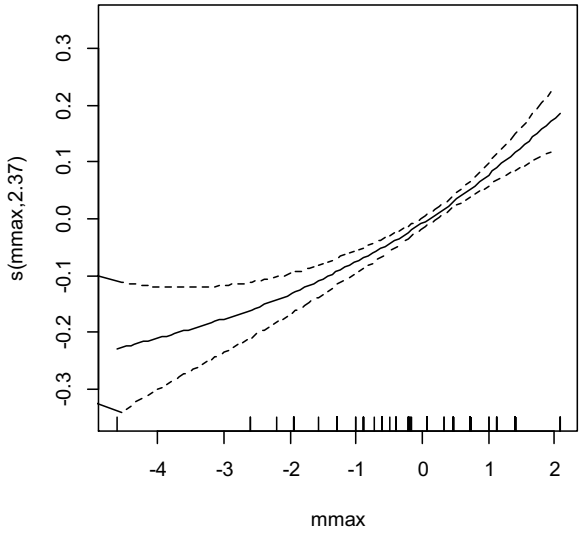
- If you want to control the degree of smoothing in each constituent function, you can use the k parameter within the s() argument, where k is the d.f. of the smoother. E.g., replacing s(cach) by s(cach, k = 2) in the above would force a smoother $f_4(x_4)$, and using s(cach, k = 20) would be rougher
- You can also control the degree of smoothing use the sp parameter in gam(). sp=c(-1,-1,-1,-1,-1,-1) (or omitting sp, I think) will result in all smoothing parameters being estimated using GCV or something similar. Using sp=c(-1,-1,-1,.1,-1,-1) gives a smoother $f_4(x_4)$, and using Using sp=c(-1,-1,-1,.000001,-1,-1) is rougher.
- For binary response, use family = “binomial”; for multi-category response, use family = "poisson"

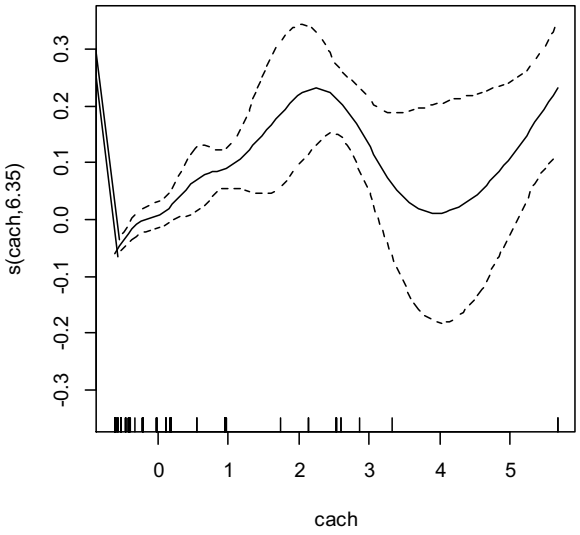
```
> summary(out)

Parametric coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 0.427234   0.004858   87.95   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Approximate significance of smooth terms:
           edf Ref.df      F  p-value
s(syct)  4.089  5.055  3.856 0.002298 **
s(mmin)  1.000  1.000 12.060 0.000637 ***
s(mmax)  2.368  3.032 23.123 3.66e-13 ***
s(cach)  6.351  7.163 11.762 5.68e-13 ***
s(chmin) 5.399  6.355  1.547 0.160295
s(chmax) 2.605  3.173  1.804 0.143871
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

R-sq.(adj) =  0.876   Deviance explained = 88.9%
GCV score = 0.0055362  Scale est. = 0.0049319  n = 209
```

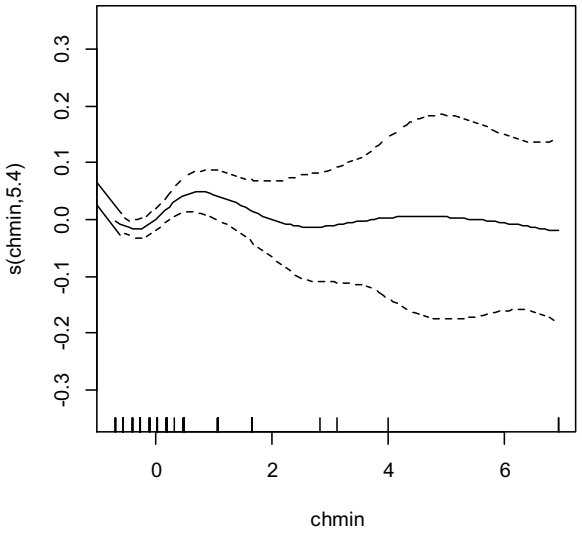
Visualizing a GAM by Plotting Component f()'s

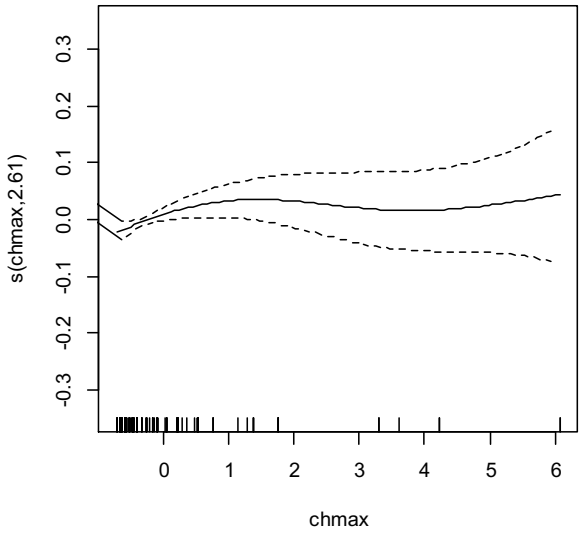












Visualizing Additive Models in General

- By definition, a model is additive if there are no interactions, in which case you can always write it as

$$Y(\mathbf{x}) = \alpha + f_1(x_1) + f_2(x_2) + \ldots + f_k(x_k) + \varepsilon$$

- For visualizing the effect of x_j on Y, you can simply plot $\hat{f}_j(x_j)$ vs. x_j, or equivalently $\hat{y}(\mathbf{x})$ vs. x_j using any set of fixed values for the other predictors

- The effect of x_j on Y is the same regardless of the levels of the other predictors because of the model additivity:

$$\hat{y}(\mathbf{x}) = \left[\hat{\alpha} + \sum_{l \neq j} \hat{f}_l(x_l)\right] + \hat{f}_j(x_j)$$

$$= \left[\begin{array}{c}\text{constant offset} \\ \text{that depends on} \\ \text{the other predictors}\end{array}\right] + \hat{f}_j(x_j)$$

Another Convenient Use of GAMs

- Plotting the component $f_j()$'s can be useful for getting a rough idea of the effects of individual predictors and selecting a subset of relevant predictors with which to fit some other nonlinear model that performs poorly in high dimensions (e.g., nearest neighbors, local linear regression, etc) – choose the predictors with the largest $f_j()$'s
- Trees are also good for this purpose, since they automatically omit irrelevant predictors – you can overgrow and choose the predictors to include based on order of importance
- What are the relative advantages and disadvantages of GAMs versus trees for selecting a subset of important predictors?

PPR Fit for Concrete data

```
out<-ppr(Strength~., data=CRT1, nterms=6)
summary(out)
yhat<-predict(out)
plot(yhat,CRT1$Strength)
##
par(mfrow=c(2,3)); plot(out)  #plot component functions
```

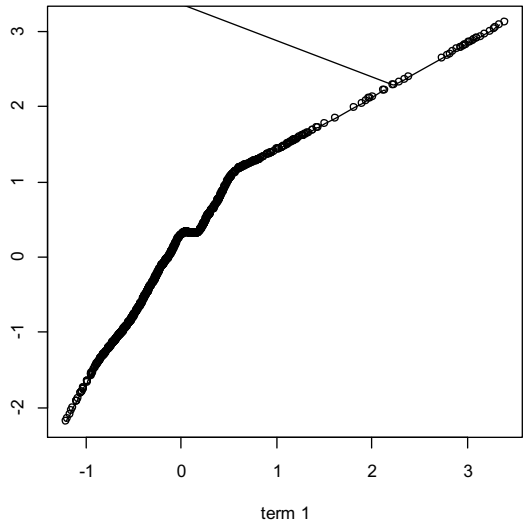
Projection direction vectors:

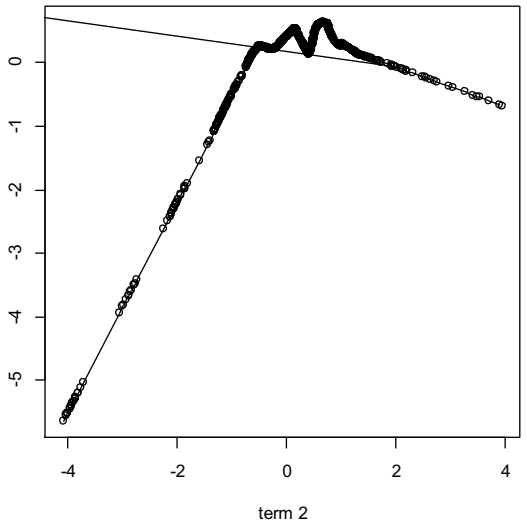
	term 1	term 2	term 3	term 4	term 5	term 6
Cement	0.56038180	0.14875663	-0.51181859	0.19253651	0.32375148	-0.18080845
Slag	0.36971039	0.02906004	-0.42377081	0.30425184	0.60764538	-0.35929921
FlyAsh	0.24818052	-0.05312206	-0.40260714	-0.61037330	0.54251333	-0.12058484
Water	-0.02835874	0.04016617	-0.03668892	0.25889810	0.19846385	-0.68332281
SPlast	0.04247494	0.74361516	0.36594101	0.10989266	-0.05797348	-0.33548973
CAgg	0.13441209	-0.01507782	-0.29928555	0.24351108	0.25379468	-0.33372838
FAgg	0.14182243	0.02512404	-0.30127236	0.24231468	0.30552913	-0.36370375
Age	0.66850905	-0.64712413	0.28420371	-0.54834867	0.17641641	0.02309131

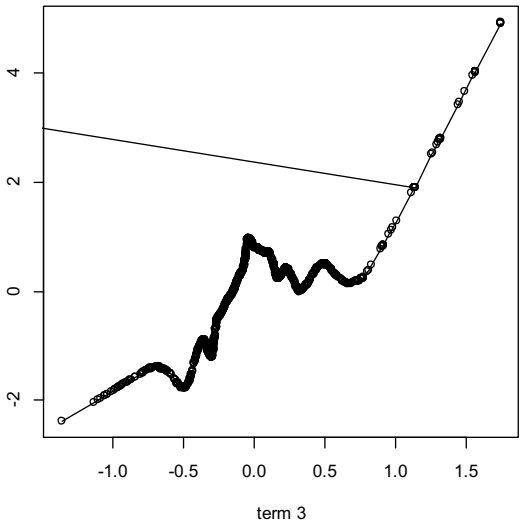
Coefficients of ridge terms:

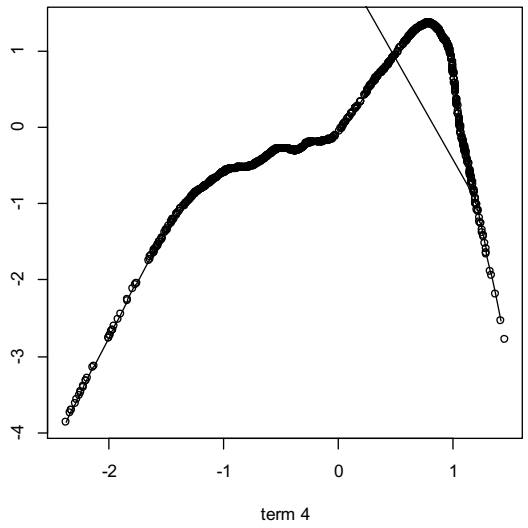
term 1	term 2	term 3	term 4	term 5	term 6
0.20686296	0.10395489	0.05538284	0.06171454	0.03707326	0.03622624

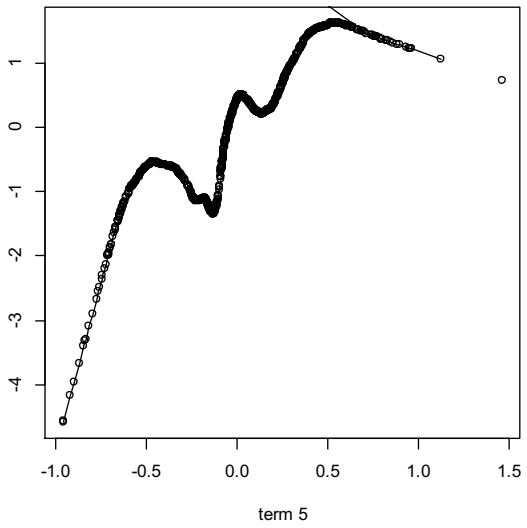
Plots of the PPR Component Functions

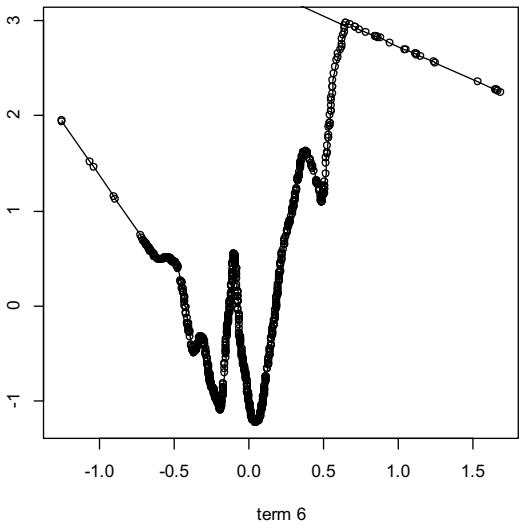












Discussion Points and Questions

- How can you interpret the PPR response surface and component functions?
- Which of the six component PPR functions is the most important?
- Which predictors appear the most important, and what are their effects?
- Does nterms = 6 seem like a reasonable choice?

CV to Find Best nterms in PPR on Concrete Data

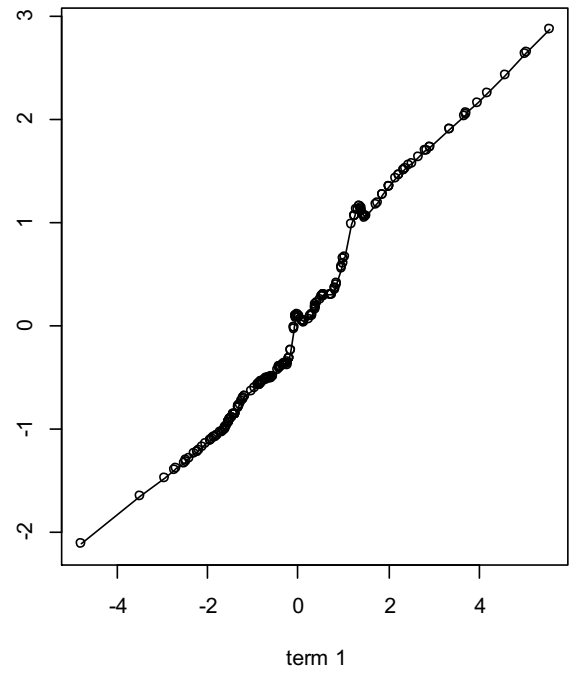
```
Nrep<-5 #number of replicates of CV
K<-10  #K-fold CV on each replicate
n.models = 2 #number of different models to fit
n=nrow(CRT1)
y<-CRT1$Strength
yhat=matrix(0,n,n.models)
MSE<-matrix(0,Nrep,n.models)
for (j in 1:Nrep) {
  Ind<-CVInd(n,K)
  for (k in 1:K) {
     out<- ppr(Strength~., data=CRT1[-Ind[[k]],], nterms=10)
     yhat[Ind[[k]],1]<-as.numeric(predict(out,CRT1[Ind[[k]],]))
     out<- ppr(Strength~., data=CRT1[-Ind[[k]],], nterms=20)
     yhat[Ind[[k]],2]<-as.numeric(predict(out,CRT1[Ind[[k]],]))
  } #end of k loop
  MSE[j,]=apply(yhat,2,function(x) sum((y-x)^2))/n
} #end of j loop
MSEAve<- apply(MSE,2,mean); MSEAve #averaged mean square CV error
MSEsd <- apply(MSE,2,sd); MSEsd   #SD of mean square CV error
r2<-1-MSEAve/var(y); r2  #CV r^2
MSE
```

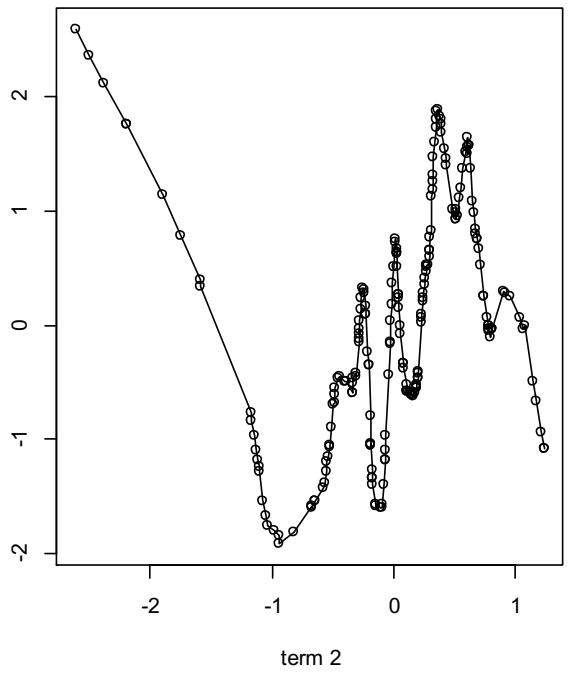
PPR Fit for CPUS data

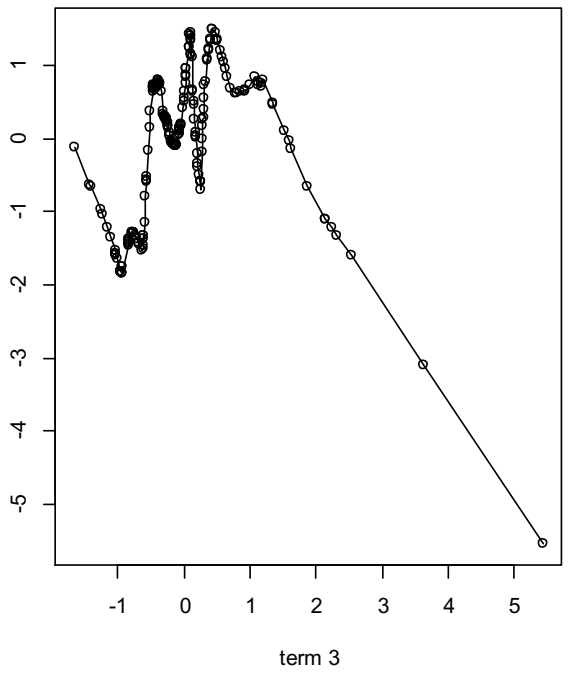
```
out<-ppr(perf~., data=CPUS1, nterms=3)
summary(out)
yhat<-predict(out)
plot(yhat,CPUS1$perf)
##
par(mfrow=c(1,3))
plot(out)  #plot component functions
```

```
Projection direction vectors:
       term 1      term 2      term 3
syct  -0.25632765 -0.02650517  0.38934668
mmin   0.34270188 -0.25311005 -0.11243881
mmax   0.64436620  0.80500176 -0.38123957
cach   0.58713347 -0.45158231  0.80839663
chmin  0.12233875 -0.28025561  0.18320722
chmax  0.20482010  0.06881243  0.05790573

Coefficients of ridge terms:
    term 1     term 2     term 3
0.19074005 0.05432089 0.05744858
```

Plots of the PPR Component Functions







Discussion Points and Questions

- How can you interpret the PPR response surface and component functions?
- Which of the three component PPR functions is the most important?
- Which predictors appear the most important, and what are their effects?
- Does nterms = 3 seem like a reasonable choice?

CV to Find Best nterms in PPR on CPUS Data

```
Nrep<-10 #number of replicates of CV
K<-10  #K-fold CV on each replicate
n=nrow(CPUS1)
y<-CPUS1$perf
SSE<-matrix(0,Nrep,2)
for (j in 1:Nrep) {
  Ind<-CVInd(n,K)
  yhat1<-y;
  yhat2<-y;
  for (k in 1:K) {
     out<- ppr(perf~., data=CPUS1[-Ind[[k]],], nterms=1)
     yhat1[Ind[[k]]]<-as.numeric(predict(out,CPUS1[Ind[[k]],]))
     out<- ppr(perf~., data=CPUS1[-Ind[[k]],], nterms=2)
     yhat2[Ind[[k]]]<-as.numeric(predict(out,CPUS1[Ind[[k]],]))
  } #end of k loop
  SSE[j,]=c(sum((y-yhat1)^2),sum((y-yhat2)^2))
} #end of j loop
SSE
apply(SSE,2,mean)
```

Basis Functions

- Many supervised learning methods can be viewed as modeling the response as a linear combination of a set of nonlinear functions of $\mathbf{x}$, each of specified form

$$Y = \beta_1 f_1(\mathbf{x}) + \beta_2 f_2(\mathbf{x}) + \ldots + \beta_M f_M(\mathbf{x}) + \varepsilon$$

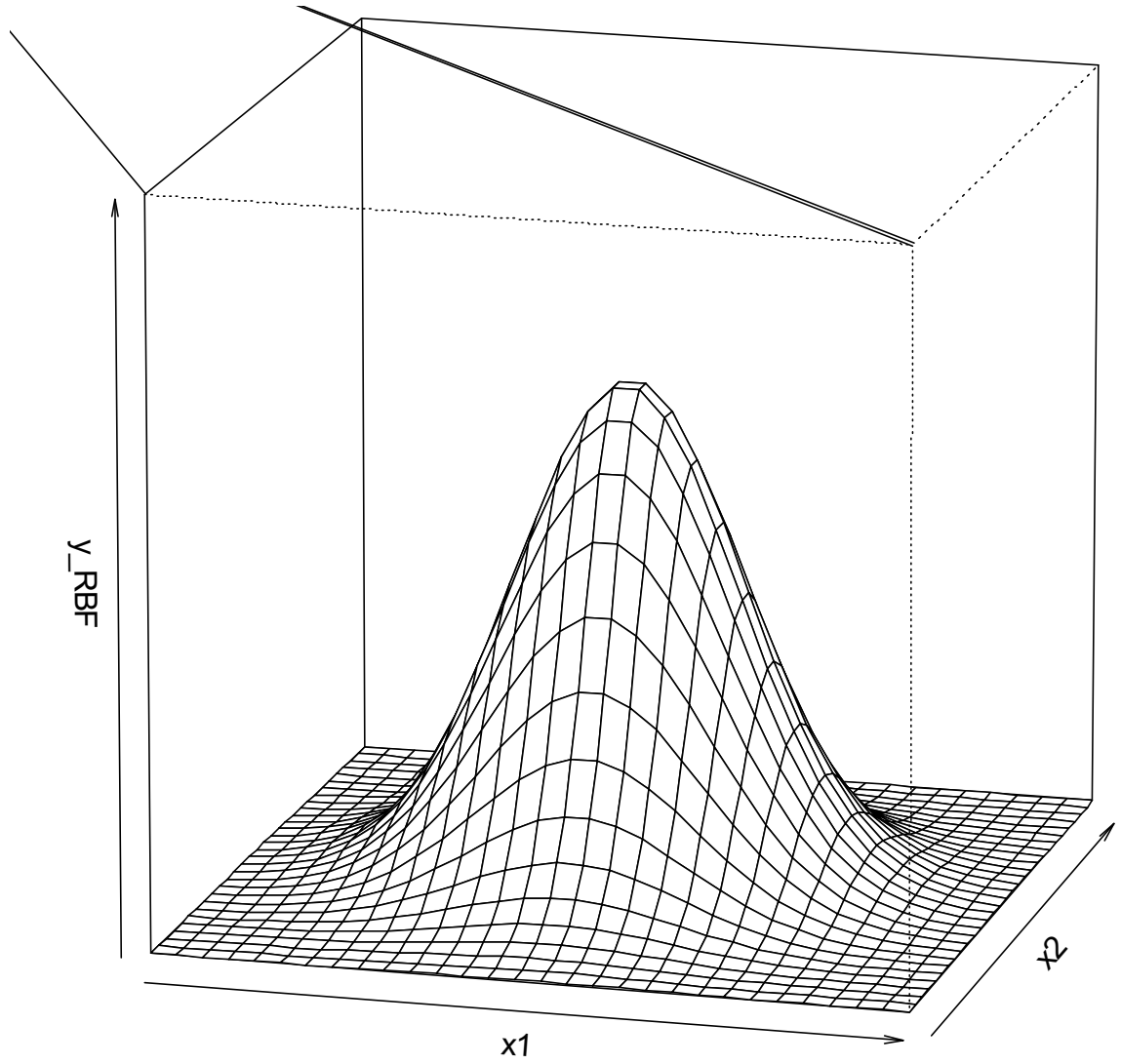
where $f_j(\bullet)$ are specified functions of $\mathbf{x}$, e.g.:

polynomial: $f_1(\mathbf{x}) = x_1, f_2(\mathbf{x}) = x_1^2, f_3(\mathbf{x}) = x_1 x_2$, etc

Gaussian RBF: $f_j(\mathbf{x}) = exp\left\{ - \frac{\left\| \mathbf{x} - \mathbf{x}_j \right\|^2}{2\alpha^2} \right\}$

- Fitting such a model involves estimating:
  - the coefficients (β_j's) in the linear combination, usually under some regularization/shrinkage penalty
  - some complexity or other parameters on which the $f_j(\bullet)$'s depend

Example of Gaussian RBF Basis Function



Basis Functions continued

- Almost all methods can be cast as fitting basis functions if the $f_j(\bullet)$'s are allowed to depend on the training $\mathbf{x}$'s (and sometimes the training y's): polynomial regression, Gaussian RBFs, neural networks, trees, smoothing splines, local regression, wavelets, etc.
- To understand any method, it is helpful to think about what the basis functions associated with it look like
- Basis functions are really more of a way to interpret supervised learning methods, rather than a distinct method of their own

Ensemble/Committee Methods

- Ensemble methods combine a number of (what are usually) “weak learners” to form a single more powerful learner
- Summary of most common ensemble methods:
  - Bootstrap aggregation (bagging), for averaging a similar set of learners, not necessarily weak, each learner fit to a different bootstrapped training sample
  - Stacking, for linearly combining any set of learners, not necessarily weak, with coefficients determined by regression
  - Boosting, for a weighted average of weak learners, usually simple trees, each fitted sequentially to the errors from the previous fit, with weights based on the errors
  - Random forests, for averaging trees, each fit to a different bootstrapped training sample, with some tricks for giving less correlated learners

Bagging

- Straightforward idea: Generate many different bootstrap samples, fit a model to each, and take the predicted response (predicted probabilities if classification) at any $\mathbf{x}$ to be the average of the predicted responses at $\mathbf{x}$ for the individual models, i.e.:

  for $b = 1, 2, \ldots, B$:

  - generate bootstrap sample of size n
  - fit model $g\left(\mathbf{x}; \hat{\boldsymbol{\theta}}^b\right)$ (any type of model you like)

  Take $\hat{y}(\mathbf{x}) = \frac{1}{B} \sum_{b=1}^{B} g\left(\mathbf{x}; \hat{\boldsymbol{\theta}}^b\right)$

When Bagging Might Be Useful

- For predictors that are linear ($\hat{\mathbf{y}}$ a linear function of training $\mathbf{y}$), bagging has no effect on the predictor
- The types of predictors that have potential to be most improved by bagging are ones for which:
  - fitting is unstable (i.e., a small change in the data gives a very different fitted model) and/or
  - the structure is such that the sum of multiple predictors no longer has the same structure
- Trees are the most important type of predictor that can be improved by bagging
  - Random forests are bagging with trees, but with a few whistles and bells added (boosted trees are related to random forests, but not quite bagging)
  - Bagging is not commonly used with predictors other than trees, so we will not cover bagging in its generality here

Illustration of How Bagging/Boosting Trees Changes Their Structure (HTF, Fig 8.12)

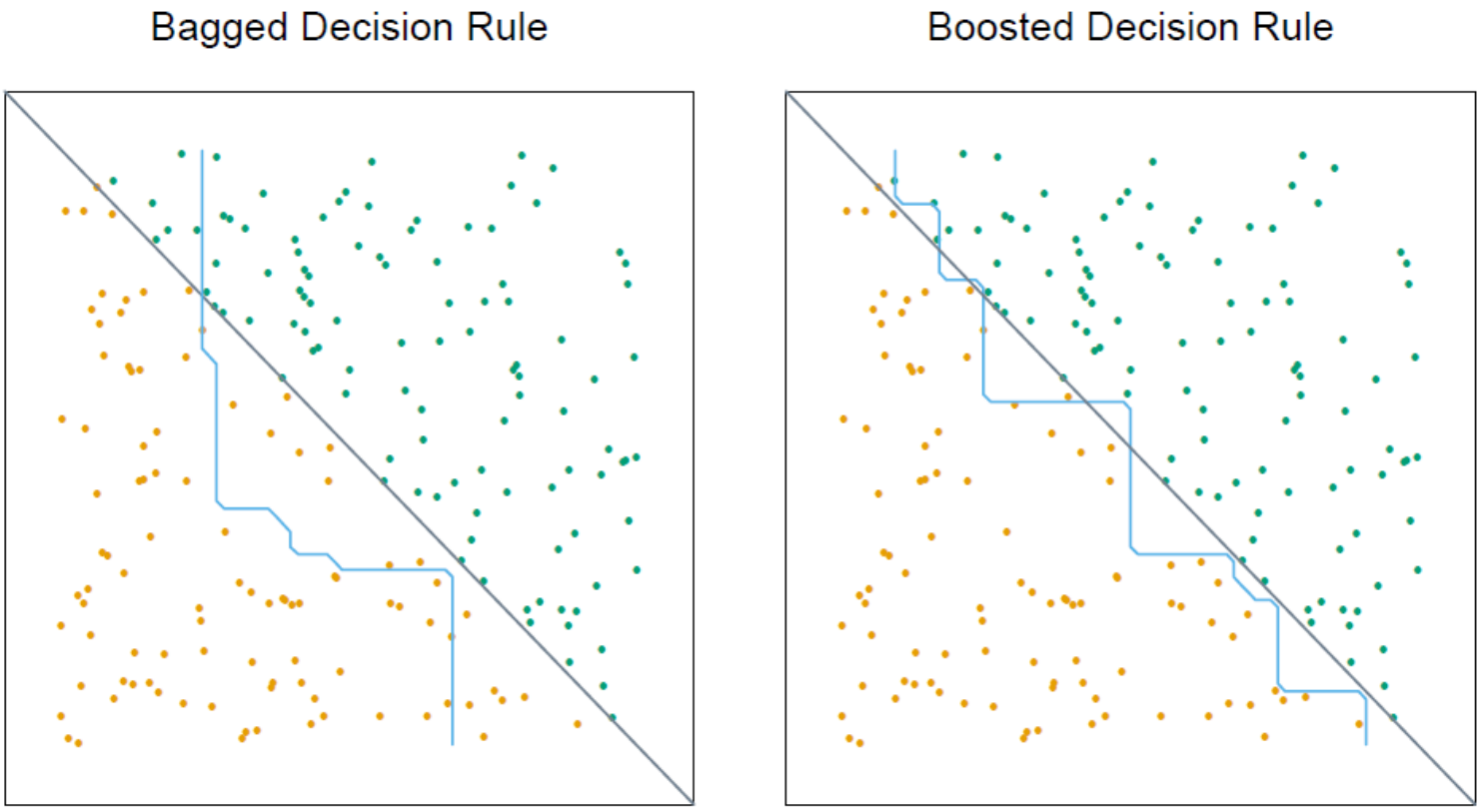


FIGURE 8.12. *Data with two features and two classes, separated by a linear boundary. (Left panel:) Decision boundary estimated from bagging the decision rule from a single split, axis-oriented classifier. (Right panel:) Decision boundary from boosting the decision rule of the same classifier. The test error rates are* 0.166, *and* 0.065, *respectively. Boosting is described in Chapter 10.*

Stacking

- Whereas in bagging you usually average an ensemble of similarly structured models, each fit to a different bootstrapped sample, in stacking you usually average differently structured models, each fit to the entire training data
- Straightforward idea: Fit a number of different models, each possibly with entirely different structure. Then take a linear combination (i.e., a weighted average) of the constituent models as the predictor, using linear regression to determine the coefficients (i.e., the weights), with the constituent models as basis functions
- The only subtlety is that when conducting regression to find the β's, a leave-one-out CV SSE must be used

Stacking Algorithm (almost but not quite)

For each $m = 1, 2, \ldots, M$:

fit model $g_m\left(\mathbf{x}; \hat{\boldsymbol{\theta}}^m\right)$ (any set of M models that you like)

Then take $\hat{y}(\mathbf{x}) = \sum_{m=1}^{M} \hat{\beta}_m g_m\left(\mathbf{x}; \hat{\boldsymbol{\theta}}^m\right)$

with $\hat{\beta}_m$ found by minimizing $\sum_{i=1}^{n}\left[y_i - \left\{ \sum_{m=1}^{M} \beta_m g_m\left(\mathbf{x}_i; \hat{\boldsymbol{\theta}}^m\right)\right\}\right]^2$

- Question: What is the problem with using this approach exactly?

Stacking Algorithm (exactly)

For each $m = 1, 2, \ldots, M$:

fit n separate models $\left\{ g_m\left(\mathbf{x}; \hat{\boldsymbol{\theta}}^m_{-i}\right) : \; i = 1,2,\cdots n \right\}$

where $\hat{\boldsymbol{\theta}}^m_{-i} =$ coefficients for mth model, fitted with ith row $\{y_i, \mathbf{x}_i\}$ left out (as in leave-one-out CV)

Then take $\hat{y}(\mathbf{x}) = \sum_{m=1}^{M} \hat{\beta}_m \, g_m\left(\mathbf{x}; \hat{\boldsymbol{\theta}}^m\right)$

with $\hat{\beta}_m$ found by minimizing $\sum_{i=1}^{n} \left[y_i - \left\{ \sum_{m=1}^{M} \beta_m g_m\left(\mathbf{x}_i; \hat{\boldsymbol{\theta}}^m_{-i}\right) \right\} \right]^2$

Discussion Points and Questions

- What is the computational expense associated with stacking? Which part of the stacking algorithm is the most computationally expensive?
- Would the computational expense be better or worse if you used K-fold, instead of n-fold CV
- How does stacking relate to model selection using CV?
- Stacking has not been extensively investigated and the jury is still out regarding whether it is worth the effort
  - very computationally expensive if the constituent models are expensive to fit
  - same issue as in bagging (not necessary for model classes that are closed under summation)
  - there are much less expensive ensemble methods like boosting

Boosting

- Bagging and stacking fit a set of models in parallel and take a weighted or unweighted average for the final model
- (Gradient) boosting sequentially fits weak models (almost always simple trees) to the residuals from the previous iteration, taking the final model to be the sum of the individual models from each iteration
- Notation:

M = # of trees/iterations to fit
$f(\mathbf{x})$ = predictor of $y(\mathbf{x})$
$L(y_i, f(\mathbf{x}_i))$ = loss function to be minimized when fitting
r_{im} = psuedo-residual of $\{y_i, \mathbf{x}_i\}$ at iteration m
J_m = # nodes for mth tree

Rough Idea Behind Boosting with Trees

Fit tree T^1 with response $\{y_i\colon\ i = 1, 2, \ldots, n\}$ and calculate (for $i = 1, 2, \ldots, n$)

Stage 1 predictions: $\hat{y}^1(\mathbf{x}_i) = T^1(\mathbf{x}_i)$

Stage 1 residuals $e_i^1 = y_i - \hat{y}^1(\mathbf{x}_i)$

Fit tree T^2 with response $\left\{e_i^1\colon\ i = 1, 2, \ldots, n\right\}$ and calculate (for $i = 1, 2, \ldots, n$)

Stage 2 predictions: $\hat{y}^2(\mathbf{x}_i) = \hat{y}^1(\mathbf{x}_i) + T^2(\mathbf{x}_i) = T^1(\mathbf{x}_i) + T^2(\mathbf{x}_i)$

Stage 2 residuals $e_i^2 = y_i - \hat{y}^2(\mathbf{x}_i)$

Fit tree T^3 with response $\left\{e_i^2\colon\ i = 1, 2, \ldots, n\right\}$ and calculate (for $i = 1, 2, \ldots, n$)

Stage 3 predictions: $\hat{y}^3(\mathbf{x}_i) = \hat{y}^2(\mathbf{x}_i) + T^3(\mathbf{x}_i) = T^1(\mathbf{x}_i) + T^2(\mathbf{x}_i) + T^3(\mathbf{x}_i)$

Stage 3 residuals $e_i^3 = y_i - \hat{y}^3(\mathbf{x}_i)$

$\vdots$

Fit tree T^M with response $\left\{e_i^{M-1}\colon\ i = 1, 2, \ldots, n\right\}$ and calculate (for $i = 1, 2, \ldots, n$)

Final Stage *M* predictions: $\hat{y}(\mathbf{x}) = \hat{y}^{M-1}(\mathbf{x}_i) + T^M(\mathbf{x}_i)$

$= T^1(\mathbf{x}) + T^2(\mathbf{x}) + \ldots + T^M(\mathbf{x})$

A Modification (Slow Learning) with Shrinkage Parameter $\lambda \in (0,1]$

Fit tree T^1 with response $\{y_i\colon\ i = 1, 2, \ldots, n\}$ and calculate (for $i = 1, 2, \ldots, n$)

Stage 1 predictions: $\hat{y}^1(\mathbf{x}_i) = \lambda \times T^1(\mathbf{x}_i)$

Stage 1 residuals $e_i^1 = y_i - \hat{y}^1(\mathbf{x}_i)$

Fit tree T^2 with response $\left\{e_i^1\colon\ i = 1, 2, \ldots, n\right\}$ and calculate (for $i = 1, 2, \ldots, n$)

Stage 2 predictions: $\hat{y}^2(\mathbf{x}_i) = \hat{y}^1(\mathbf{x}_i) + \lambda \times T^2(\mathbf{x}_i)$

Stage 2 residuals $e_i^2 = y_i - \hat{y}^2(\mathbf{x}_i)$

Fit tree T^3 with response $\left\{e_i^2\colon\ i = 1, 2, \ldots, n\right\}$ and calculate (for $i = 1, 2, \ldots, n$)

Stage 3 predictions: $\hat{y}^3(\mathbf{x}_i) = \hat{y}^2(\mathbf{x}_i) + \lambda \times T^3(\mathbf{x}_i)$

Stage 3 residuals $e_i^3 = y_i - \hat{y}^3(\mathbf{x}_i)$

$\vdots$

Fit tree T^M with response $\left\{e_i^{M-1}\colon\ i = 1, 2, \ldots, n\right\}$ and calculate (for $i = 1, 2, \ldots, n$)

Final Stage *M* predictions: $\hat{y}(\mathbf{x}) = \hat{y}^{M-1}(\mathbf{x}_i) + \lambda \times T^M(\mathbf{x}_i)$

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.

2. For $m = 1$ to M:

   (a) For $i = 1, 2, \ldots, N$ compute

   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$

   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.

   (c) For $j = 1, 2, \ldots, J_m$ compute

   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.

3. Output $\hat{f}(x) = f_M(x)$.

TABLE 10.2. *Gradients for commonly used loss functions.*

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \leq \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha\text{th-quantile}\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Tuning a Boosting Algorithm

- Using a large number M of iterations will eventually overfit, so must always terminate before "convergence"
- Main tuning parameters to choose:
  - loss function (based on whether you are doing regression vs classification and/or want robustness to outliers in y)
  - shrinkage parameter λ and number of iterations M (select based on test, CV, or OOB error)
  - size of each constituent tree ($4 \leq J \leq 8$ recommended)
- R's gbm package has some whistles and bells. Usually reasonable choices for tuning parameters are:
  - n.trees = M (use built in CV to choose)
  - interaction.depth = # internal splitting nodes = J–1
  - n.minobsinnode = min # obs in leaf nodes
  - shrinkage = 0.001

Discussion Points and Questions

- Smaller λ generally gives better predictive performance but worse computational expense, since it requires a larger M
- Regarding choice of loss function for regression:
  - Laplace loss function is $|y_i - f(\mathbf{x}_i)|$
  - Gaussian loss function is $[y_i - f(\mathbf{x}_i)]^2$

  Why are they called “Laplace” and “Gaussian”?
- What is the potential advantage of using a Laplace loss function versus a Gaussian loss function?
- Loss function is chosen via the "distribution" parameter in R's gbm():
  - What loss would you choose for binary classification?
  - What loss for classification with > 2 categories?
  - coxph and quantile distributions are also very useful

Gradient Boosting Trees for Concrete data

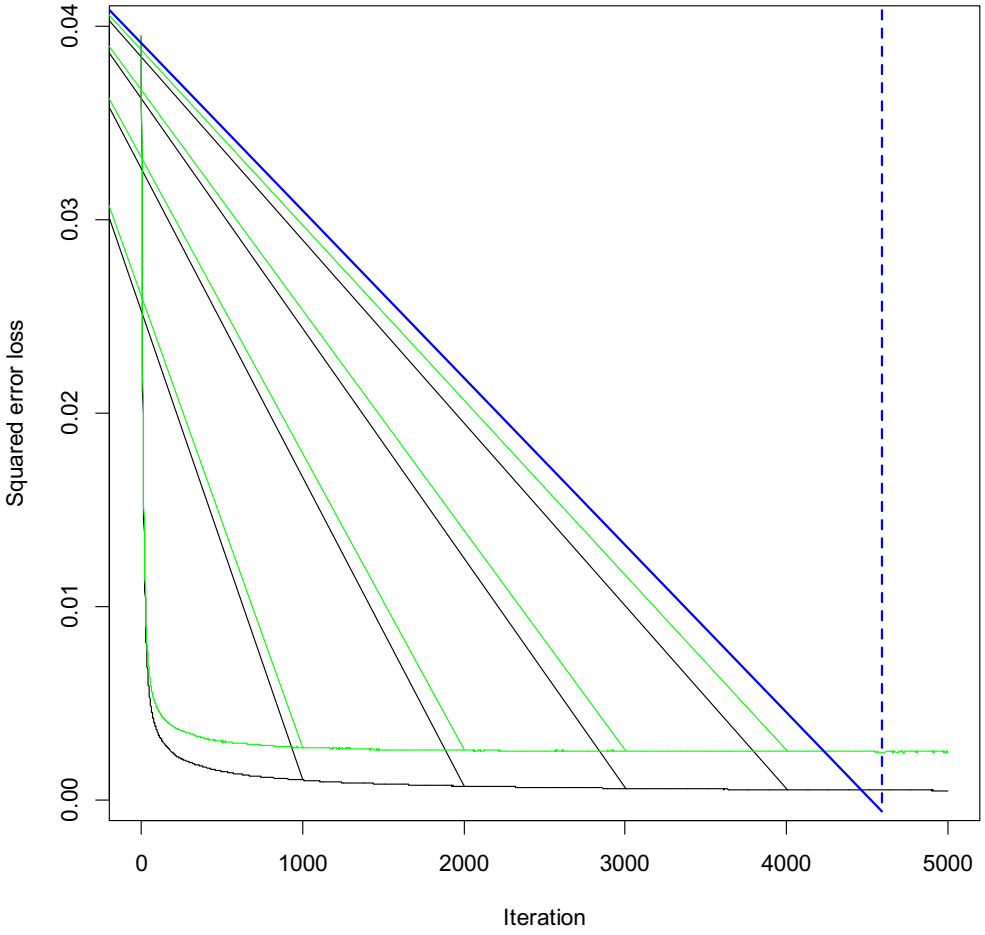
```
library(gbm)
gbm1 <- gbm(Strength~., data=CRT1, var.monotone=rep(0,8), distribution="gaussian",
    n.trees=5000, shrinkage=0.1, interaction.depth=3, bag.fraction = .5, train.fraction = 1,
    n.minobsinnode = 10, cv.folds = 10, keep.data=TRUE, verbose=FALSE)
best.iter <- gbm.perf(gbm1,method="cv");best.iter
sqrt(gbm1$cv.error[best.iter]) #CV error SD
1-gbm1$cv.error[best.iter]/var(CRT1$Strength)  #CV r^2
yhat<-predict(gbm1,CRT1, n.trees = best.iter); sd(CRT1$Strength-yhat)  #training error SD
##
summary(gbm1,n.trees=best.iter)  # based on the optimal number of trees
##
par(mfrow=c(2,4))
for (i in c(8,1,4,2,7,5,6,3)) plot(gbm1, i.var = i, n.trees = best.iter)
par(mfrow=c(1,1))
##
plot(gbm1, i.var = c(8,1), n.trees = best.iter)
##
print(pretty.gbm.tree(gbm1,1))  #show the first tree
##
print(pretty.gbm.tree(gbm1,best.iter))  #show the last tree
```

Gradient Boosting Trees for Concrete data

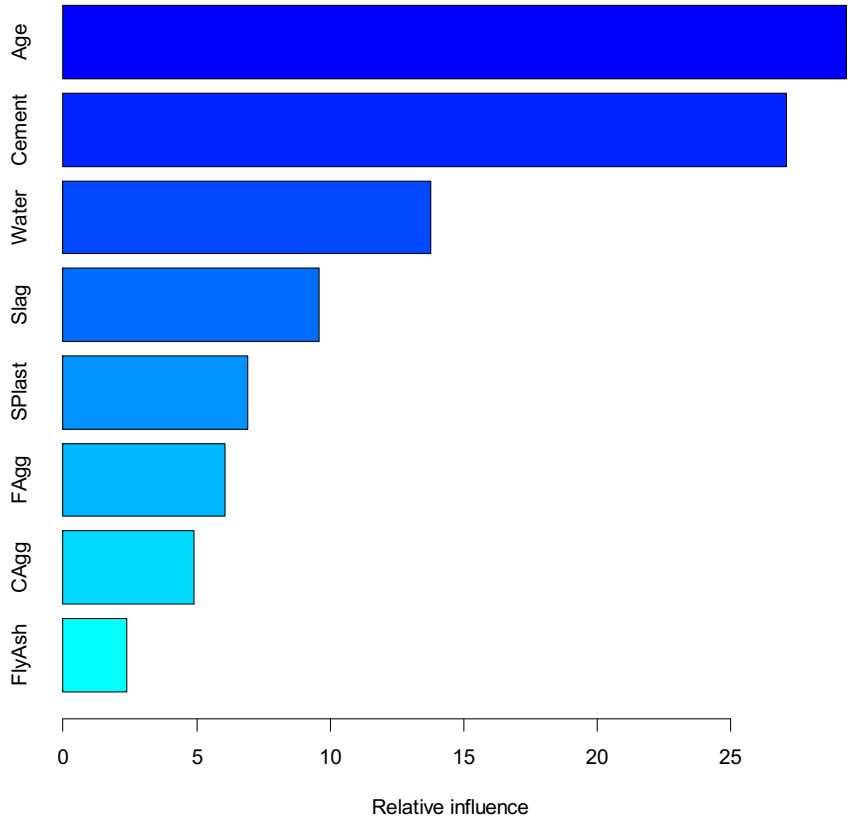
```
library(gbm)
gbm1 <- gbm(Strength~., data=CRT1, var.monotone=rep(0,8), distribution="gaussian",
    n.trees=5000, shrinkage=0.1, interaction.depth=3, bag.fraction = .5, train.fraction = 1,
    n.minobsinnode = 10, cv.folds = 10, keep.data=TRUE, verbose=FALSE)
best.iter <- gbm.perf(gbm1,method="cv");best.iter
sqrt(gbm1$cv.error[best.iter]) #CV error SD
1-gbm1$cv.error[best.iter]/var(CRT1$Strength)  #CV r^2
yhat<-predict(gbm1,CRT1, n.trees = best.iter); sd(CRT1$Strength-yhat)  #training error SD
##
summary(gbm1,n.trees=best.iter)  # based on the optimal number of trees
##Construct PD main effect plots for predictors c(8,1,4,2,7,5,6,3)
i=8; plot(gbm1, i.var = i, n.trees = best.iter)
## Construct a PD interaction plot for predictors 8 and 1
plot(gbm1, i.var = c(8,1), n.trees = best.iter)
##
print(pretty.gbm.tree(gbm1,1))  #show the first tree
##
print(pretty.gbm.tree(gbm1,best.iter))  #show the last tree
```

Boosting Results for Concrete Data

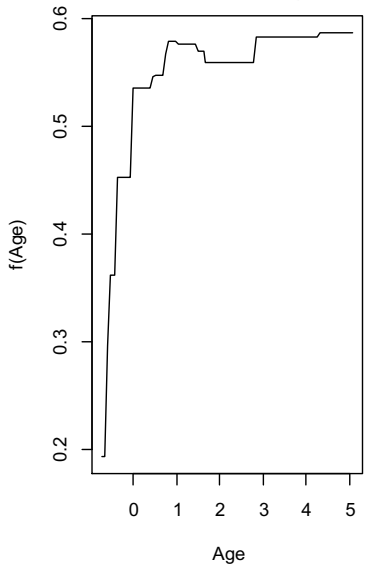
CV and Training Error vs. # Iterations

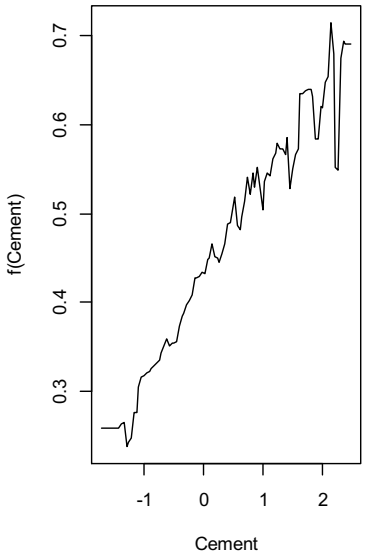


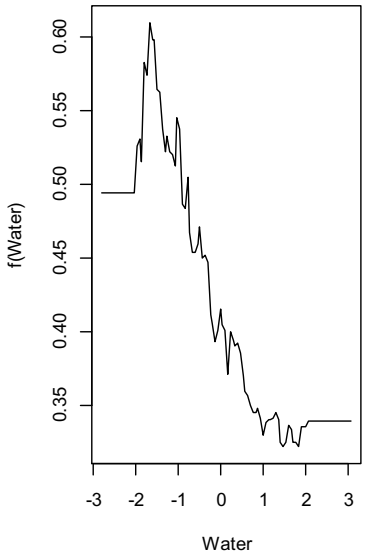
Variable Importance

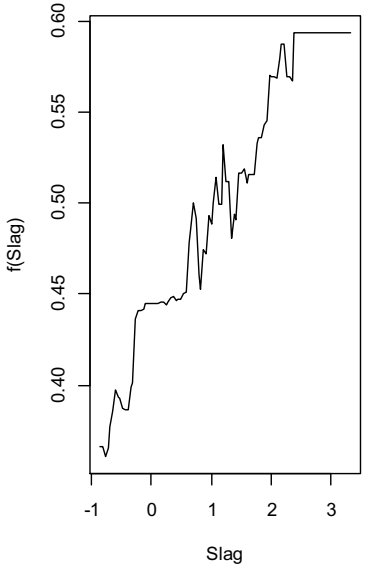


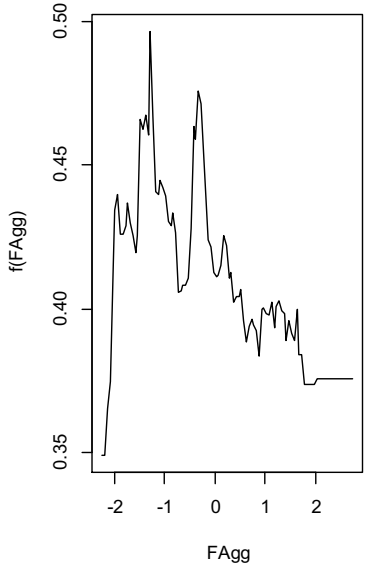
Marginal Plots for Concrete Boosted Trees

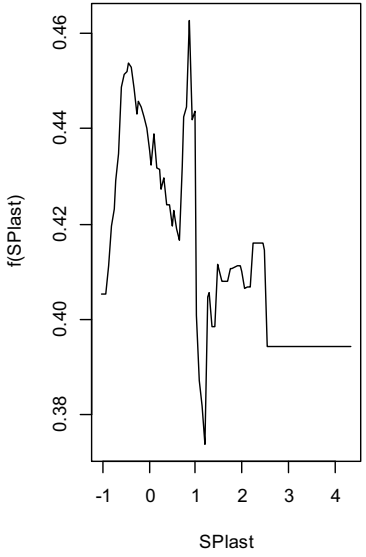


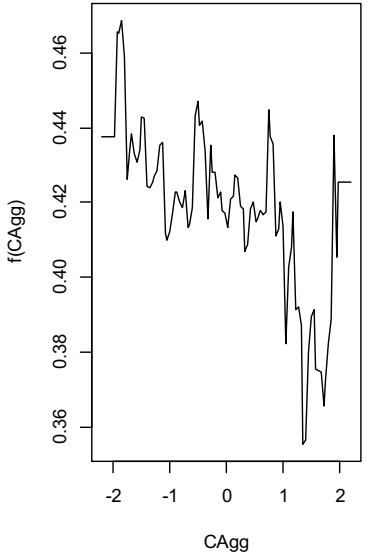


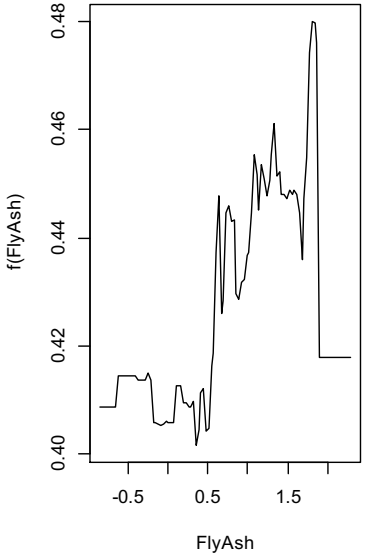




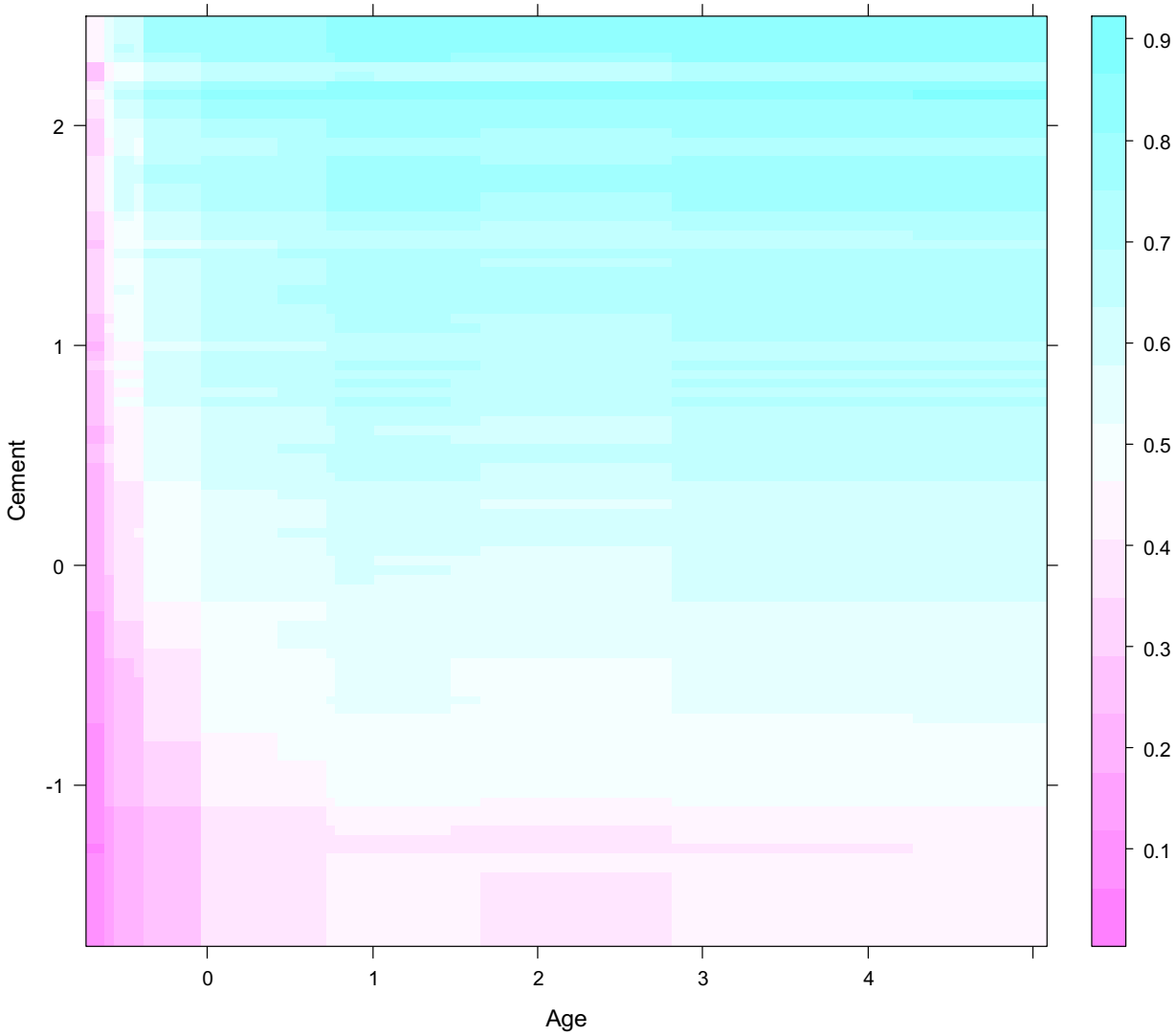








Pairwise Marginal Plot for Concrete Boosted Trees for Visualizing Interactions



Discussion Points and Questions

- The variable importance measure is simply the sum of the variable importance measures for each of the M individual trees
- For interpreting the model and the effects of the predictors
  - Individual marginal plots are very useful but cannot show any interactions (and can be misleading if interactions are strong)
  - Pairwise marginal plots show interactions between pairs of variables
  - Printing out the first few trees using print(pretty.gbm.tree(gbm1,i)) (for i = 1, 2, . . .) and attempting to interpret these may also help, but this is usually difficult
- Of all the methods we have tried for the Concrete data, which did the best?

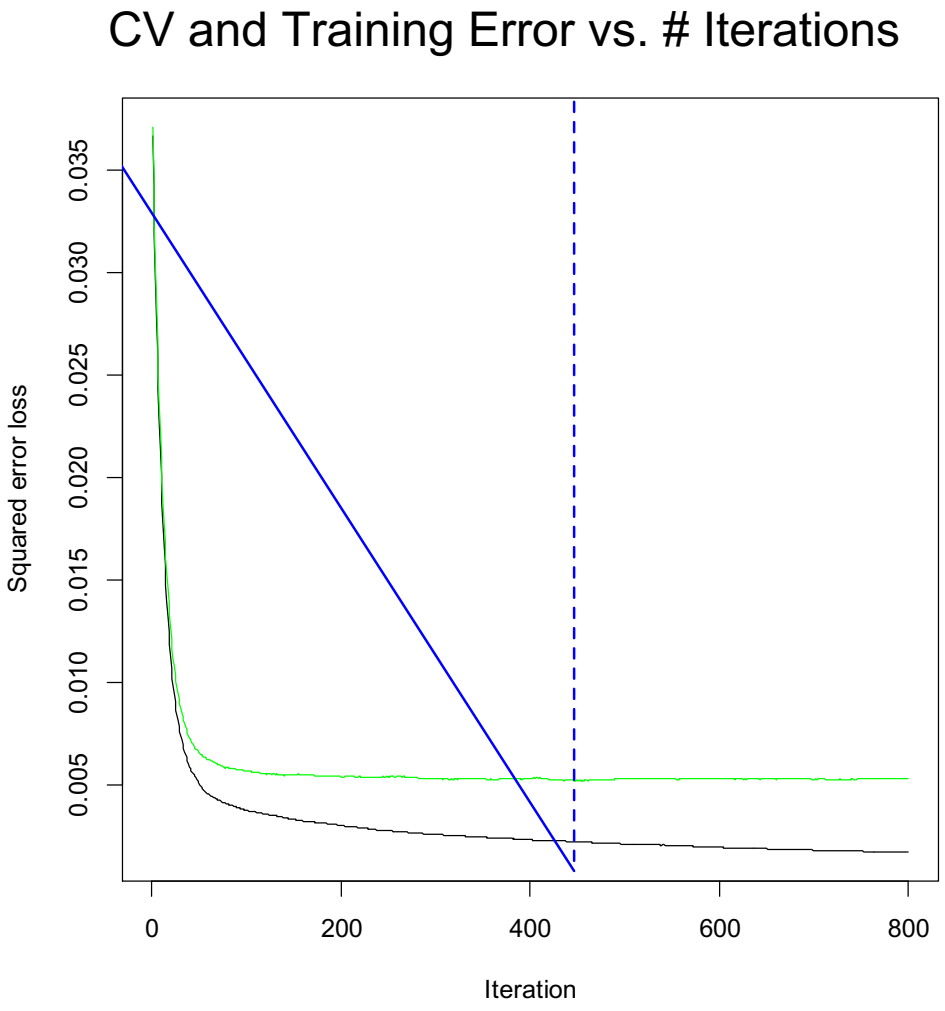
Comparison of CV r^2 for Various Methods for Concrete Data

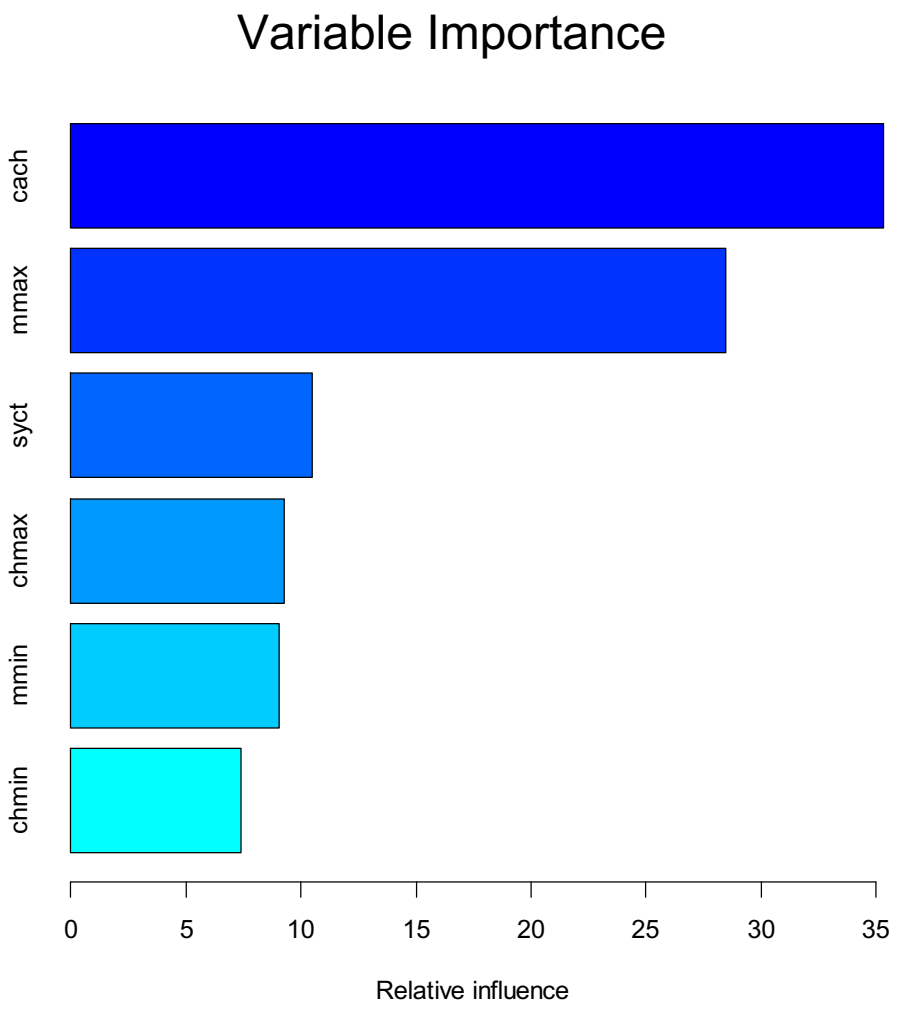
	CV error SD	CV r^2
Boosted tree	3.91	0.945
PPR	5.25	0.901
NNet	5.46	0.893
GAM	5.60	0.888
Loc. Lin.	7.27	0.811
Tree		0.793
K-NN	8.60	0.735
Lin. Reg.	10.5	0.605

Gradient Boosting Trees for CPUS data

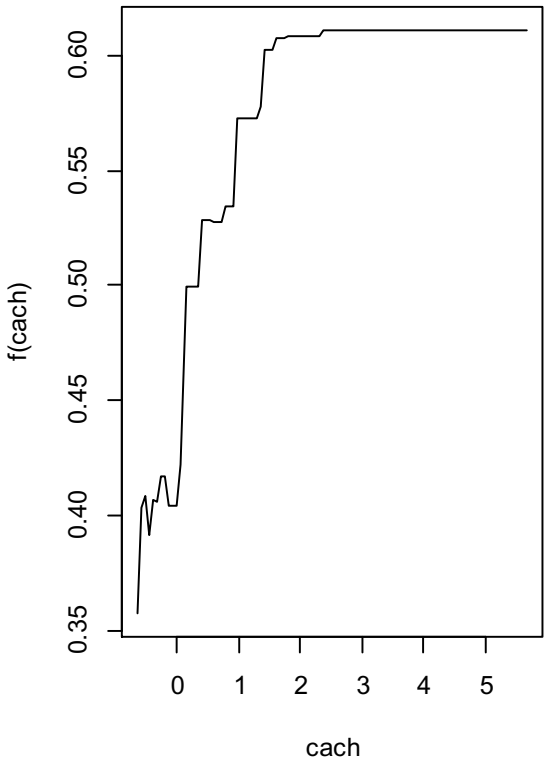
```
library(gbm)
gbm1 <- gbm(perf~., data=CPUS1, var.monotone=c(0,0,0,0,0,0), distribution="gaussian",
    n.trees=1000, shrinkage=0.05, interaction.depth=3, bag.fraction = .5, train.fraction = 1,
    n.minobsinnode = 10, cv.folds = 10, keep.data=TRUE, verbose=FALSE)
best.iter <- gbm.perf(gbm1,method="cv");best.iter
sqrt(gbm1$cv.error[best.iter])
##
summary(gbm1,n.trees=best.iter)  # based on the optimal number of trees
##
par(mfrow=c(1,3))
for (i in c(4,3,1)) plot(gbm1, i.var = i, n.trees = best.iter)
##
plot(gbm1, i.var = c(3,4), n.trees = best.iter)
##
print(pretty.gbm.tree(gbm1,1))  #show the first tree
##
print(pretty.gbm.tree(gbm1,best.iter))  #show the last tree
```

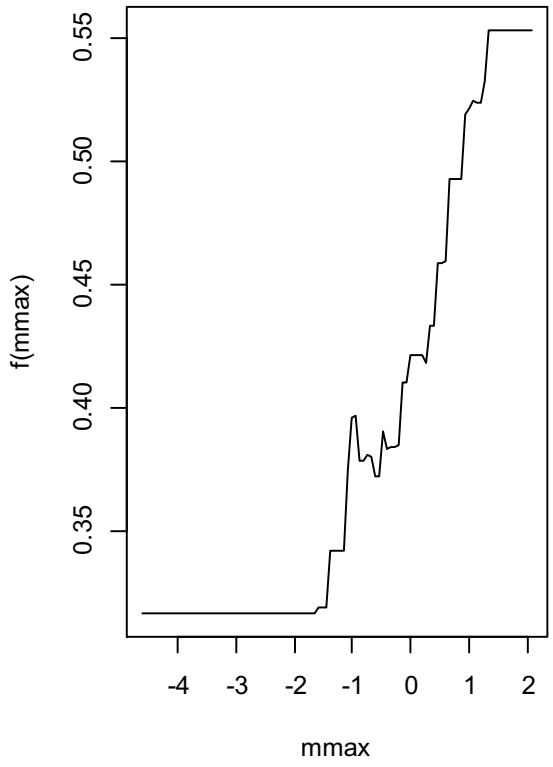
Boosting Results for CPUS Data

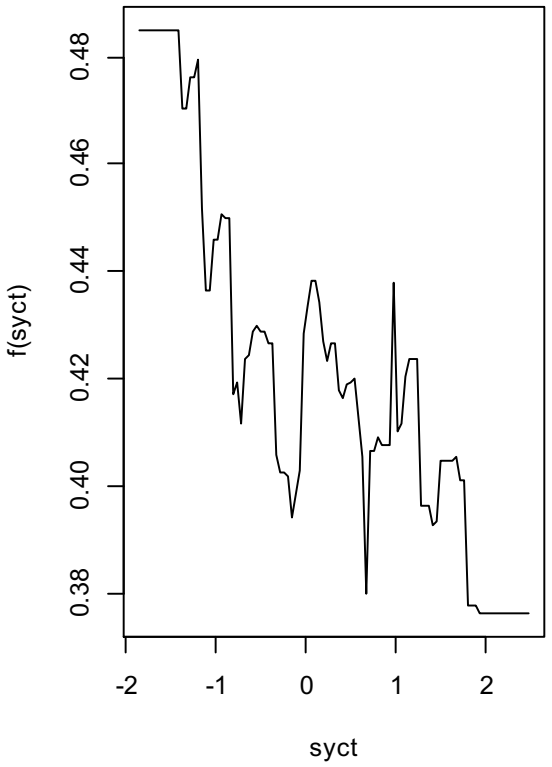




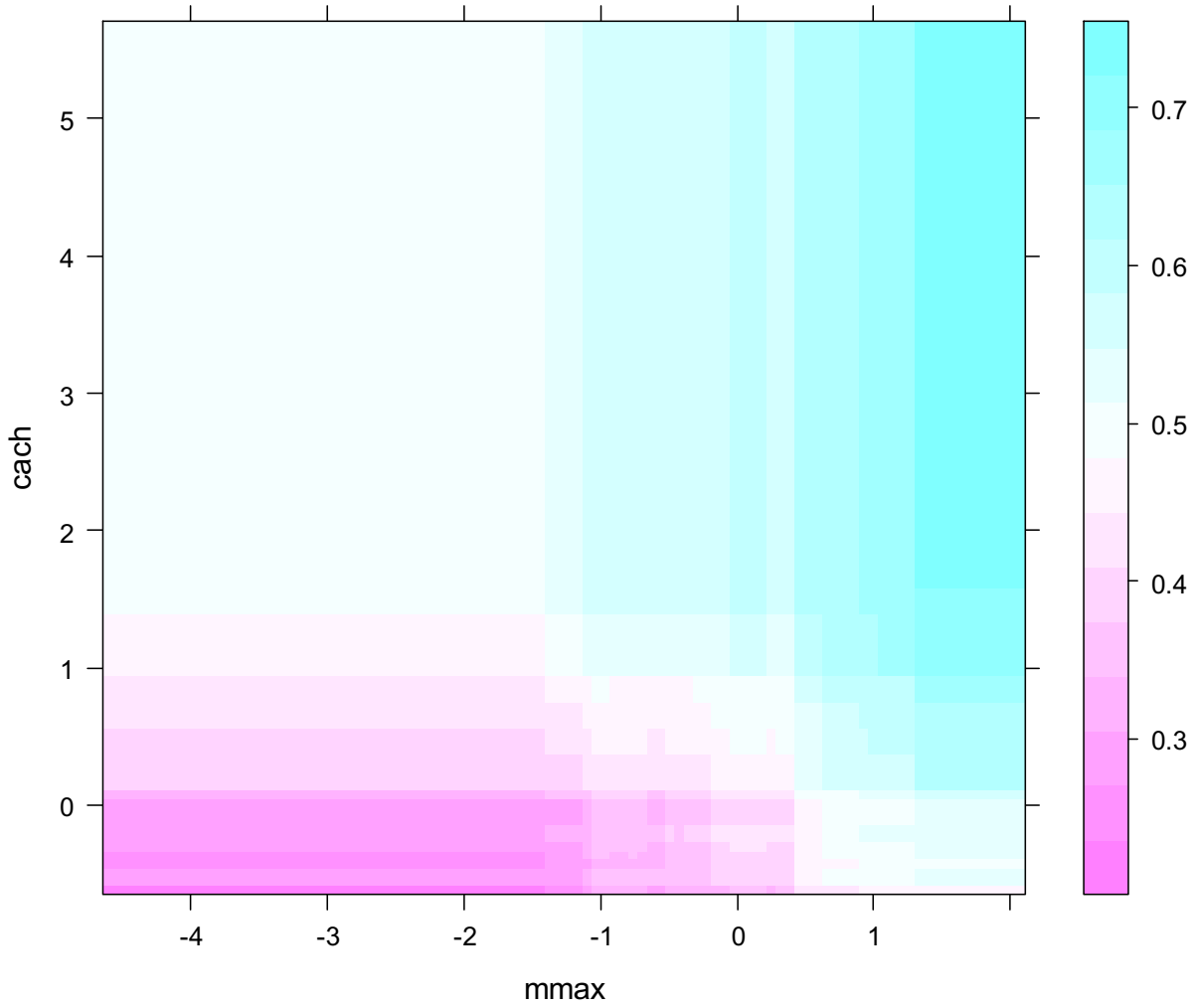
Marginal Plots for CPUS Boosted Trees







Pairwise Marginal Plot for CPUS Boosted Trees for Visualizing Interactions



Discussion Points and Questions

- The variable importance measure is simply the sum of the variable importance measures for each of the M individual trees
- For interpreting the model and the effects of the predictors
  - Individual marginal plots are very useful but cannot show any interactions (and can be misleading if interactions are strong)
  - Pairwise marginal plots show interactions between pairs of variables
  - Printing out the first few trees using print(pretty.gbm.tree(gbm1,i)) (for i = 1, 2, . . .) and attempting to interpret these may also help, but this is usually difficult
- Of all the methods we have tried for the CPUS data, which did the best?

Comparison of CV r^2 for Various Methods for CPUS Data

	CV error SD	CV r^2	
Boosted tree	0.073	86.6%	
Loc. Lin.	0.078	84.7%	
NNet	0.079	84.3%	
K-NN	0.083	82.7%	
PPR	0.085	81.8%	
Lin. Reg.	0.086	81.4%	
GAM	0.086	81.4%	
Tree	0.099	75.2%	(this was using tree, not rpart)

Advantages of Boosted Trees

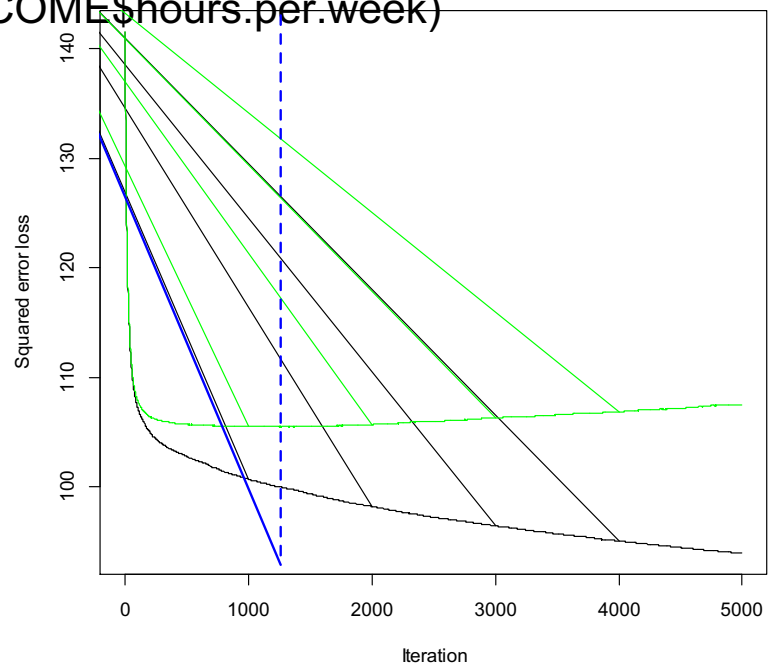
- Trees have relatively poor predictive power, but have many nice properties for data mining problems:
  - handles missing data
  - robust to outliers in predictor space
  - insensitive to monotone transformation of predictors
  - computationally scalable (to large n)
  - handles mixed and categorical predictors with many categories
  - automatically discards irrelevant predictors
  - interpretable (for moderate sized trees)
  - convenient variable importance measure
- Boosting trees can substantially improve predictive power, while retaining almost all the advantages (except interpretability, but this can be improved with individual and pairwise marginal plots)

Return to the Income Data Example

- Reconsider the data in adult_train.csv
- Previously we tried neural network and tree model for regression (predicting the number of hours per week worked) and classification (predicting the binary income categorization <= 50k vs. > 50k)
- Let's try boosted trees now

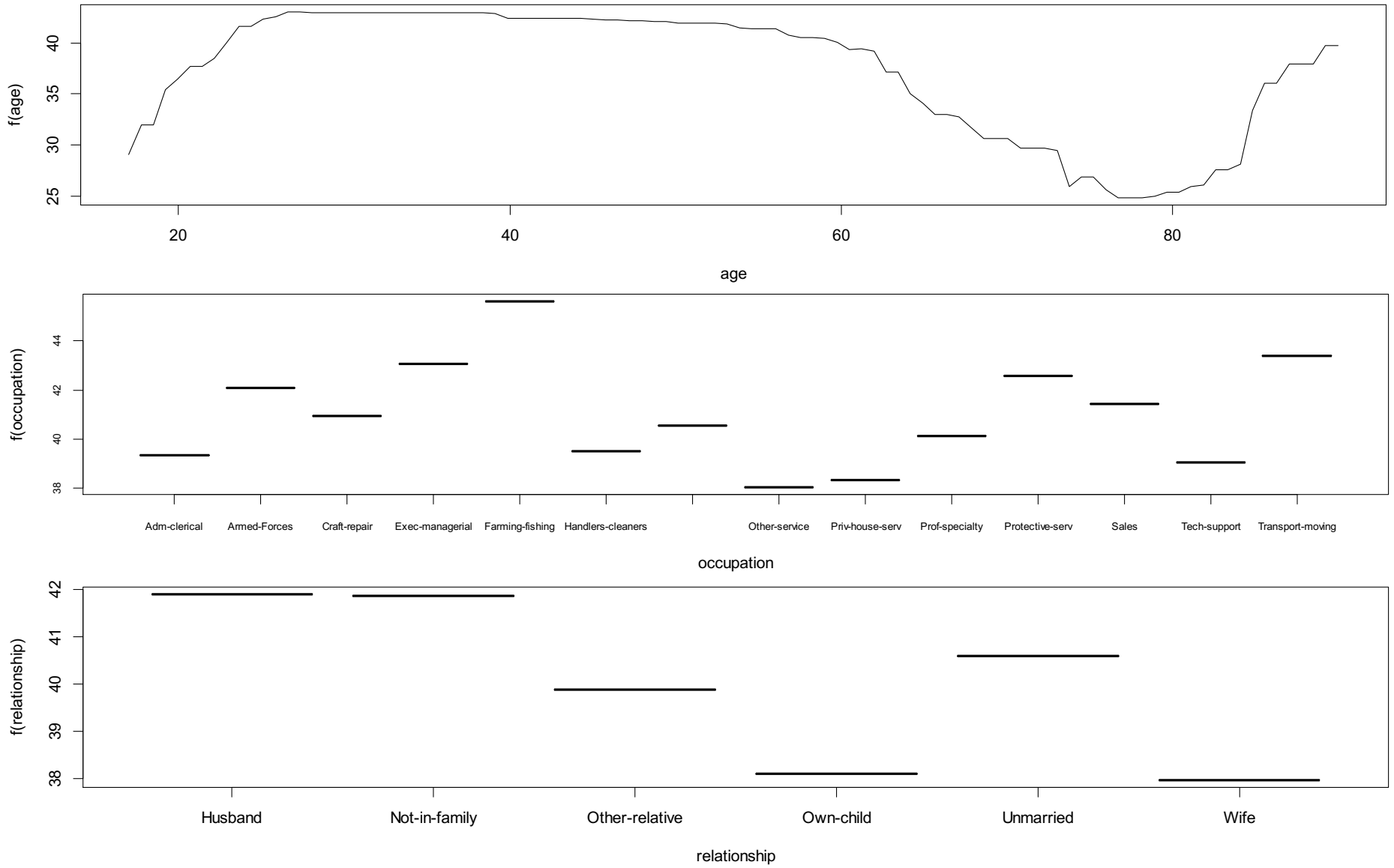
Try a Boosted Regression Tree

```
library(gbm)
XX<-read.table("adult_train.csv",sep=",",header=TRUE,strip.white=TRUE,na.strings="?")
XX<-na.omit(XX)
INCOME<-XX  #there is no need to standardize the predictors with trees (why not)
Inc.gbm1 <- gbm(hours.per.week ~ ., data=INCOME[,-c(3,4)], distribution="gaussian",
      n.trees=5000, shrinkage=0.05, interaction.depth=3, bag.fraction = .5, train.fraction =
      1, n.minobsinnode = 10, cv.folds = 10, keep.data=TRUE, verbose=FALSE)
best.iter <- gbm.perf(Inc.gbm1,method="cv");best.iter
VarCV<-Inc.gbm1$cv.error[best.iter]; 1-VarCV/var(INCOME$hours.per.week)
summary(Inc.gbm1,n.trees=best.iter)
Inc.gbm1$var.names
```

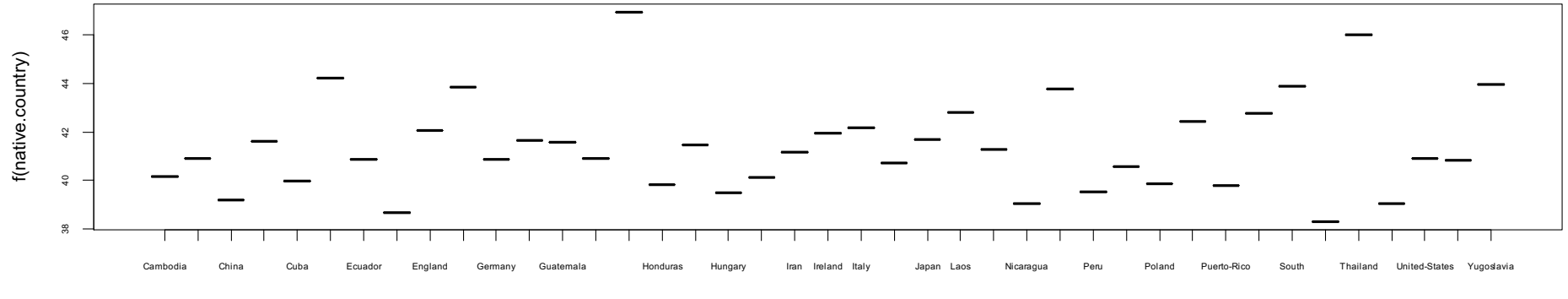


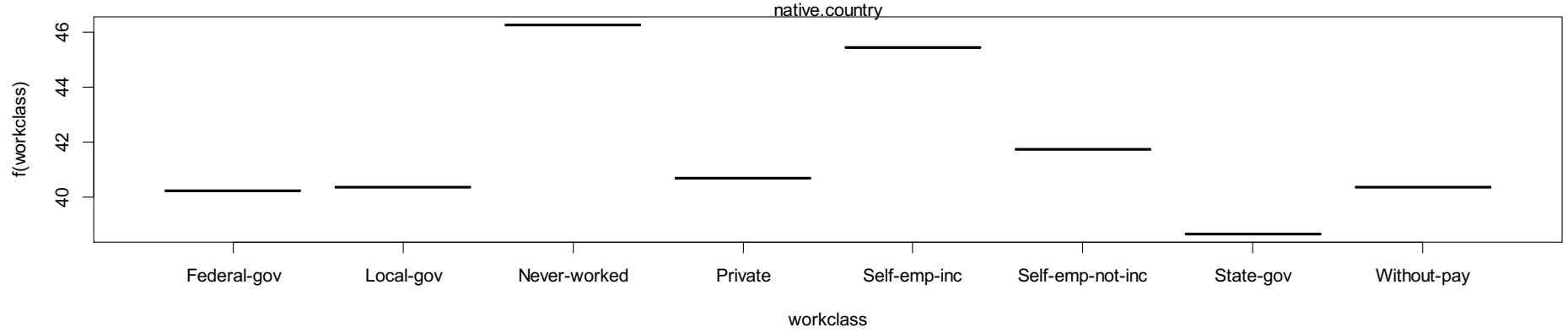
```
> best.iter
[1] 1265
> VarCV<-Inc.gbm1$cv.error[best.iter]; 1-VarCV/var(INCOME$hours.per.week)
[1] 0.2650076

> summary(Inc.gbm1,n.trees=best.iter)
                       var    rel.inf
age                    age 34.9590262
occupation          occupation 18.5648540
native.country native.country 12.6696441
relationship     relationship  9.9231513
workclass           workclass  6.4316208
sex                     sex  5.5651013
income                 income  4.0661108
education.num   education.num  3.5632509
marital.status marital.status  2.0806690
capital.loss     capital.loss  1.0607433
capital.gain     capital.gain  0.8191883
race                   race  0.2966401
```

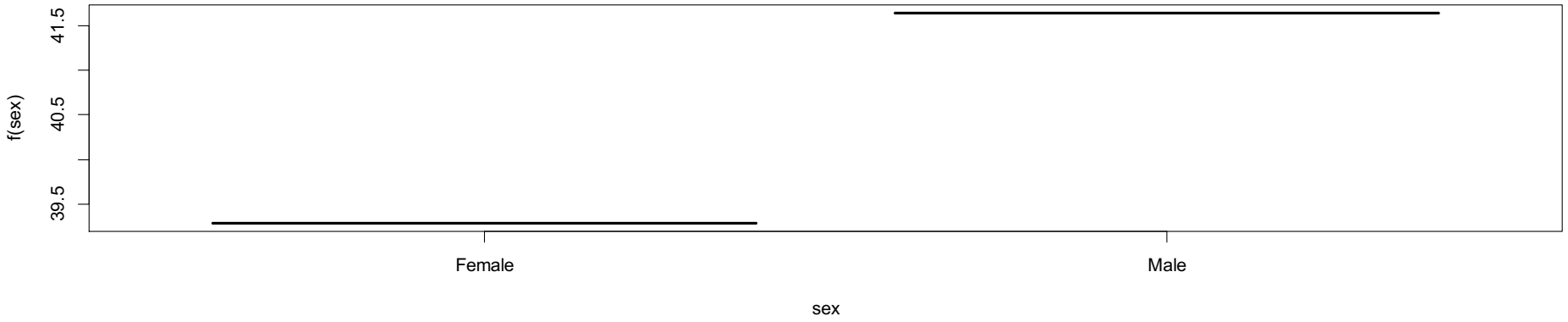
```
plot(Inc.gbm1, i.var = 1, n.trees = best.iter)
plot(Inc.gbm1, i.var = 5, n.trees = best.iter, cex.axis=.7)
plot(Inc.gbm1, i.var = 6, n.trees = best.iter, cex.axis=1)
```



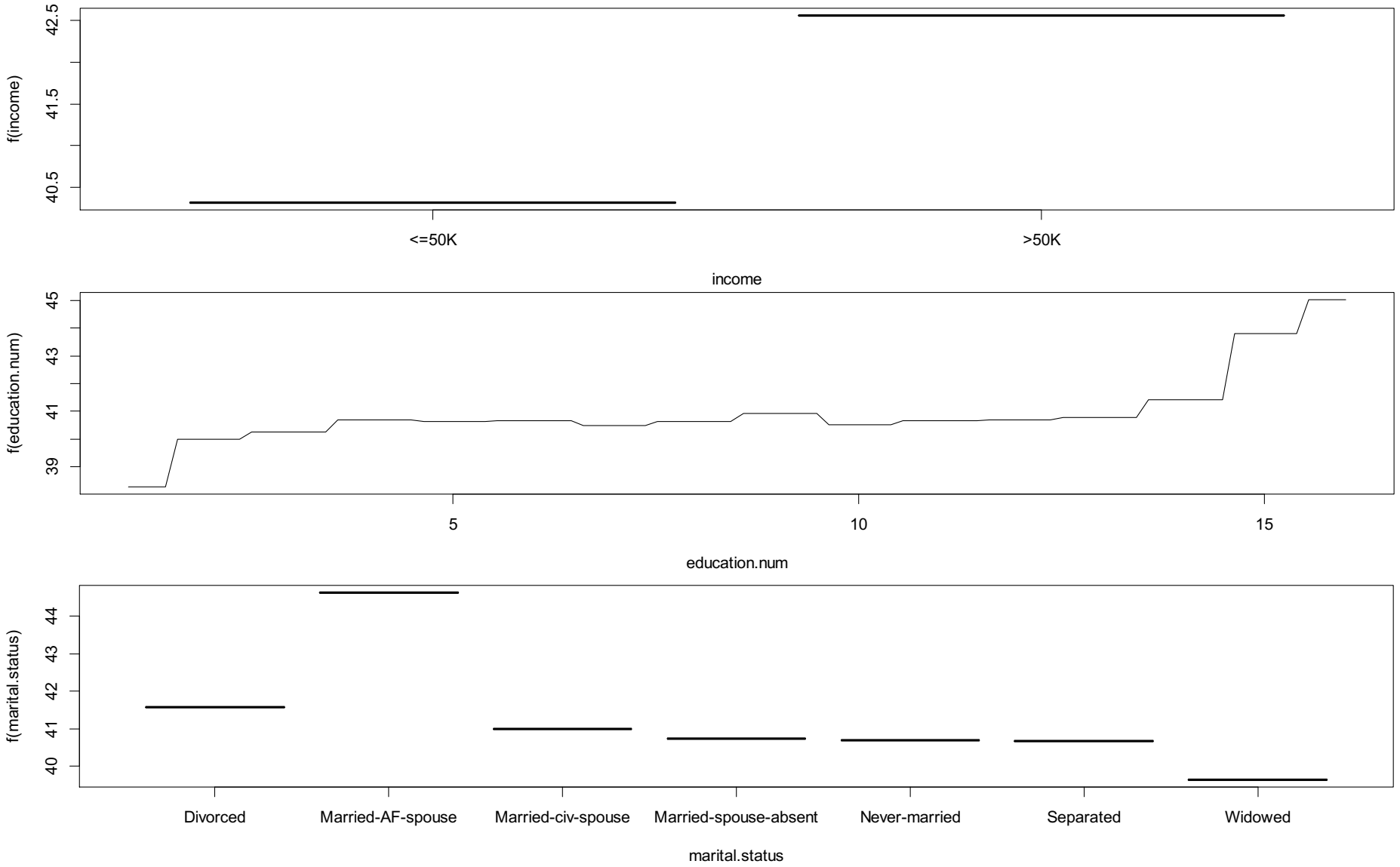
```
plot(lnc.gbm1, i.var = 11, n.trees = best.iter, cex.axis=.5)
plot(lnc.gbm1, i.var = 2, n.trees = best.iter, cex.axis=1)
plot(lnc.gbm1, i.var = 8, n.trees = best.iter, cex.axis=1)
```





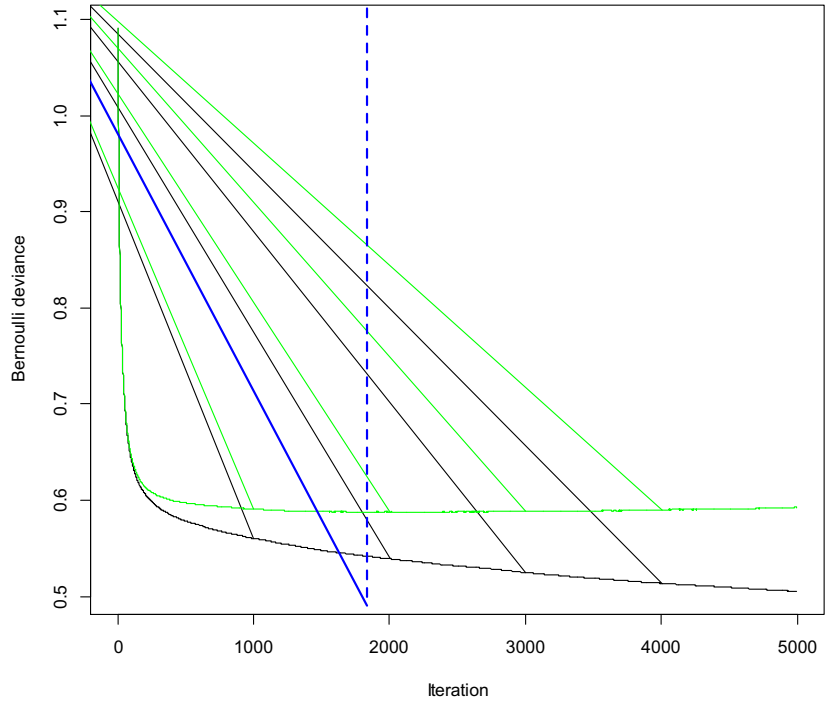


```
plot(Inc.gbm1, i.var = 12, n.trees = best.iter, cex.axis=1)
plot(Inc.gbm1, i.var = 3, n.trees = best.iter, cex.axis=1)
plot(Inc.gbm1, i.var = 4, n.trees = best.iter, cex.axis=1)
```



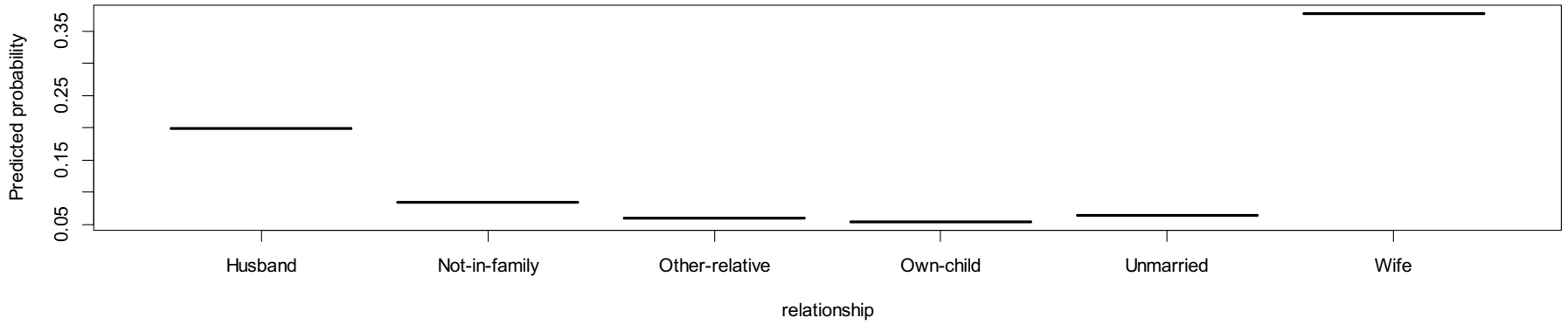
Try a Boosted Classification Tree

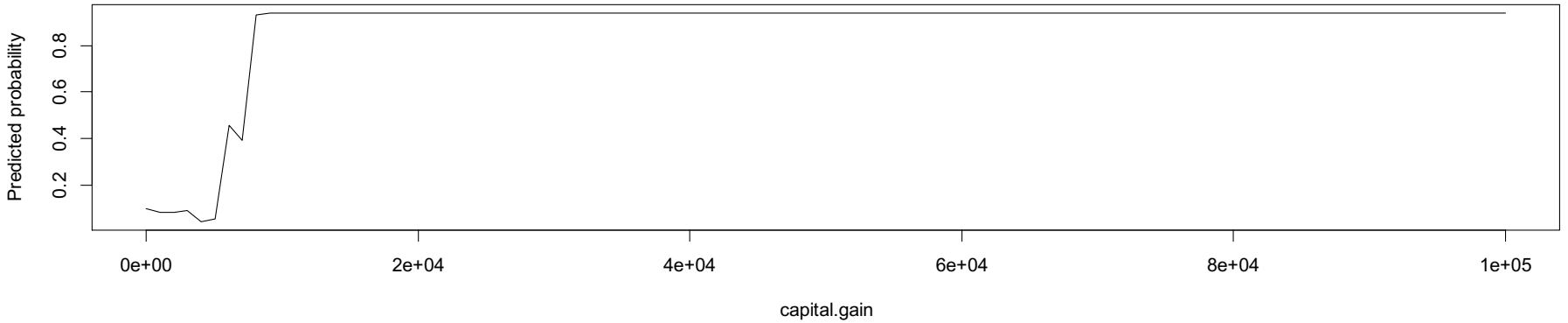
```
lnc.gbm2 <- gbm(income==">50K" ~ ., data=INCOME[,-c(3,4)], distribution="bernoulli",
      n.trees=5000, shrinkage=0.05, interaction.depth=3, bag.fraction = .5, train.fraction =
      1, n.minobsinnode = 10, cv.folds = 10, keep.data=TRUE, verbose=FALSE)
best.iter <- gbm.perf(lnc.gbm2,method="cv");best.iter
lnc.gbm2$cv.error[best.iter]
summary(lnc.gbm2,n.trees=best.iter)
lnc.gbm2$var.names
```

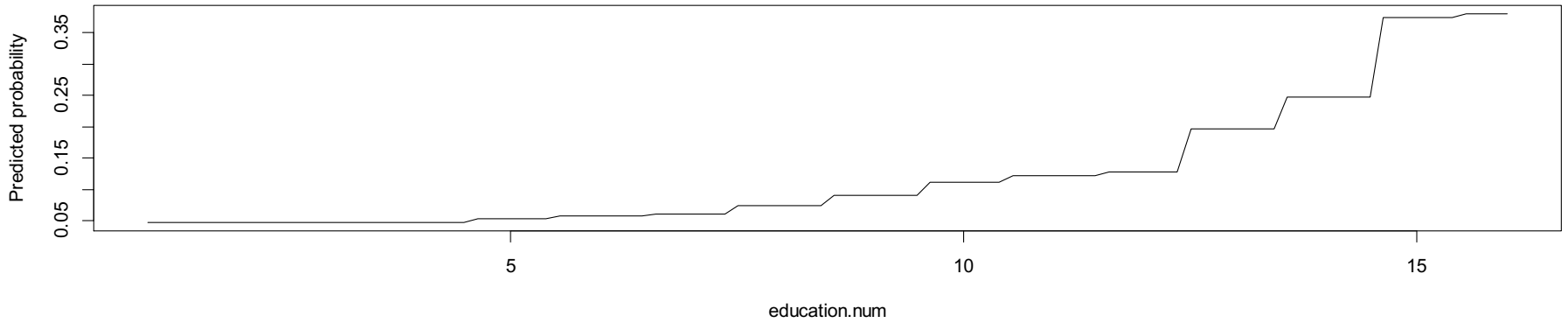


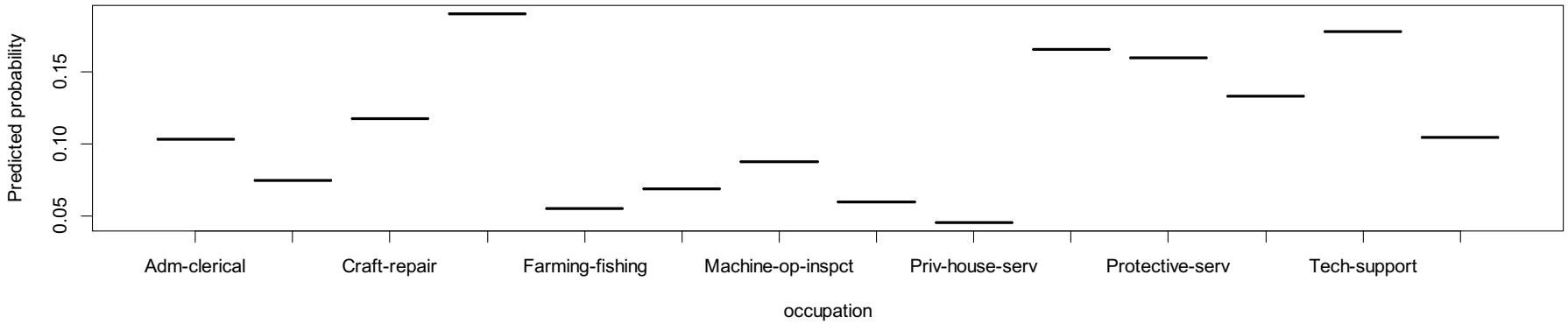
```
> best.iter <- gbm.perf(Inc.gbm2,method="cv");best.iter
[1] 1831
> Inc.gbm2$cv.error[best.iter]
[1] 0.5919548
> summary(Inc.gbm2,n.trees=best.iter)
                          var    rel.inf
relationship     relationship 27.8029637
capital.gain     capital.gain 19.7517894
education.num   education.num 12.6580751
occupation         occupation 12.0258575
age                       age  6.1660624
marital.status marital.status  5.8939557
capital.loss     capital.loss  5.7317558
native.country native.country  4.3856221
hours.per.week hours.per.week  3.5760496
workclass           workclass  1.5492630
sex                       sex  0.3552002
race                     race  0.1034056
```

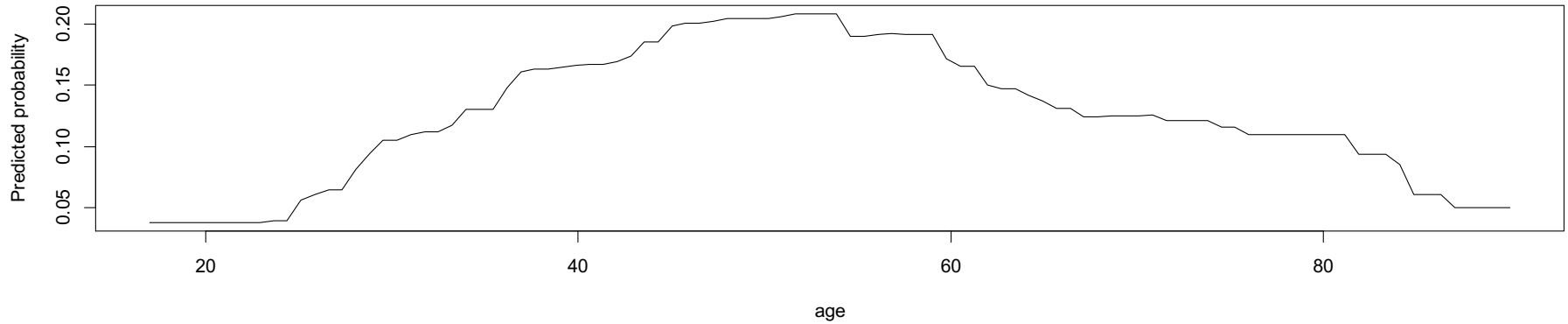
```
for (i in c(6,9,3,5,1,4)) {plot(Inc.gbm2, i.var = i, type="response",n.trees = best.iter);
      readline()}
```

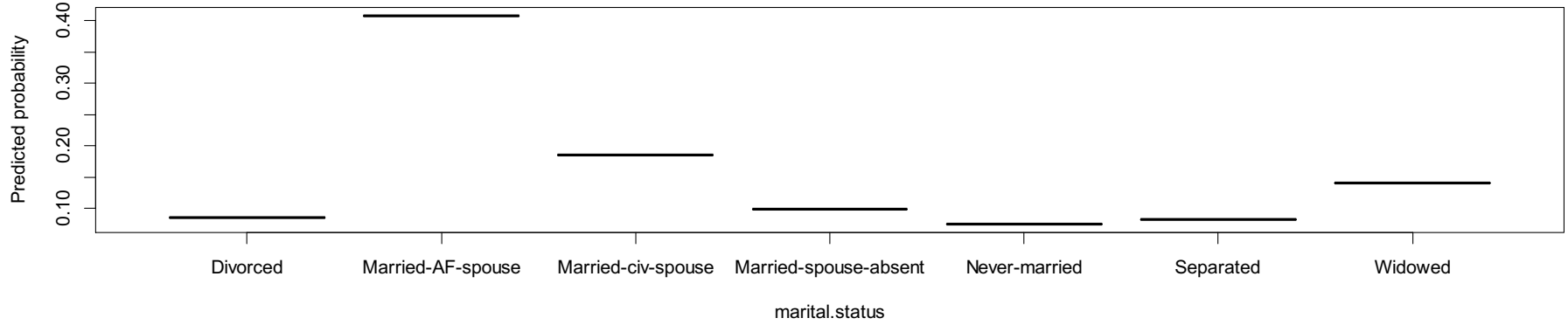







Predicted probability
0.05
0.10
0.15
Adm-clerical
Craft-repair
Farming-fishing
Machine-op-inspct
Priv-house-serv
Protective-serv
Tech-support
occupation


Predicted probability
0.05
0.10
0.15
0.20
20
40
60
80
age


Predicted probability
0.10
0.20
0.30
0.40
Divorced
Married-AF-spouse
Married-civ-spouse
Married-spouse-absent
Never-married
Separated
Widowed
marital.status

Discussion Points and Questions

- For classification, gbm() calculates the CV deviance, not the misclassification rate. How can we interpret this to assess how good the overall predictive power is?

  the best CV deviance was 0.592

  deviance $= -2\times\text{ave}\{\log\text{-}f(y_i)\}$

  $$= -2\times\text{ave}\{\log[p_i^{y_i}(1-p_i)^{(1-y_i)}]\}$$
  $$= -2\times\text{ave}\{y_i\log(p_i)+(1-y_i)\log(1-p_i)\}$$
  $$= -2\times\text{ave}\{\log(\alpha_i)\}$$

  where $\alpha_i = \begin{cases} p_i: & y_i = 1 \\ 1-p_i: & y_i = 0 \end{cases}$ = predicted probability for correct class of y_i

  thus, $\text{ave}\{\log(\alpha_i)\} = -\text{deviance}/2$, so that (very approximately)

  $$\text{ave}\{\alpha_i\} \approx \exp\{-\text{deviance}/2\} = \exp\{-0.592/2\} = 0.74$$

Random Forests

- This is an application of bagging, but with some twists that are tailored to the specific case of bagging trees
- They share all the desirable characteristics of boosted trees, and also have much better predictive performance than trees by themselves (sometimes better, sometimes worse than boosted trees)
- They may (arguably) be somewhat easier to tune than tree boosting, but with good software and CV, it is not difficult to tune (select M, J and shrinkage parameter) a tree boosting algorithm
- See randomForest package in R

Random Forest Algorithm

1) For $b = 1,2, \ldots, B$:
   a) Draw a bootstrap sample $\{\mathbf{Y}^b, \mathbf{X}^b\}$ of size n rows
   b) Grow an individual tree T^b to the bootstrapped sample by iteratively repeating the following steps for each terminal node of the tree at the current iteration, until the minimum node size (an important tuning parameter specified by the user) is reached:
      i. Select m (another tuning parameter) predictor variables randomly from among the full set of k predictors
      ii. From among these m predictors, choose the best predictor and split point the same way you do for regular trees
      iii. Spit the node into two child nodes according to ii
2) Output the ensemble of trees $\{T^b : b = 1,2, \ldots . B\}$

To predict the response Y at any new $\boldsymbol{x}$:

For regression: $\hat{Y}(\boldsymbol{x}) = \frac{1}{B}\sum_{b=1}^{B} T^b(\boldsymbol{x})$ (ave. predicted Y from all B trees)

For classification: $\hat{Y}(\boldsymbol{x})$ = majority vote among predicted class Y class from all B trees, and

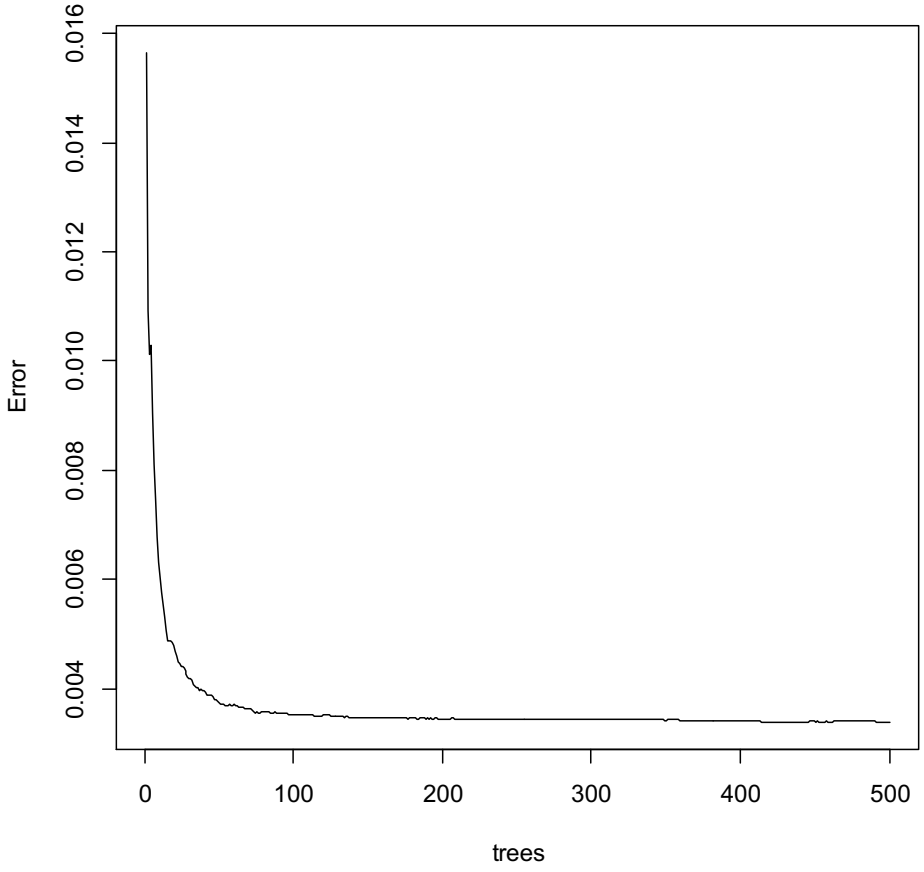
$\widehat{Pr}[Y \text{ in class } j|\boldsymbol{x}]$ = average predicted probability for class j from all B trees

Random Forest for Concrete data

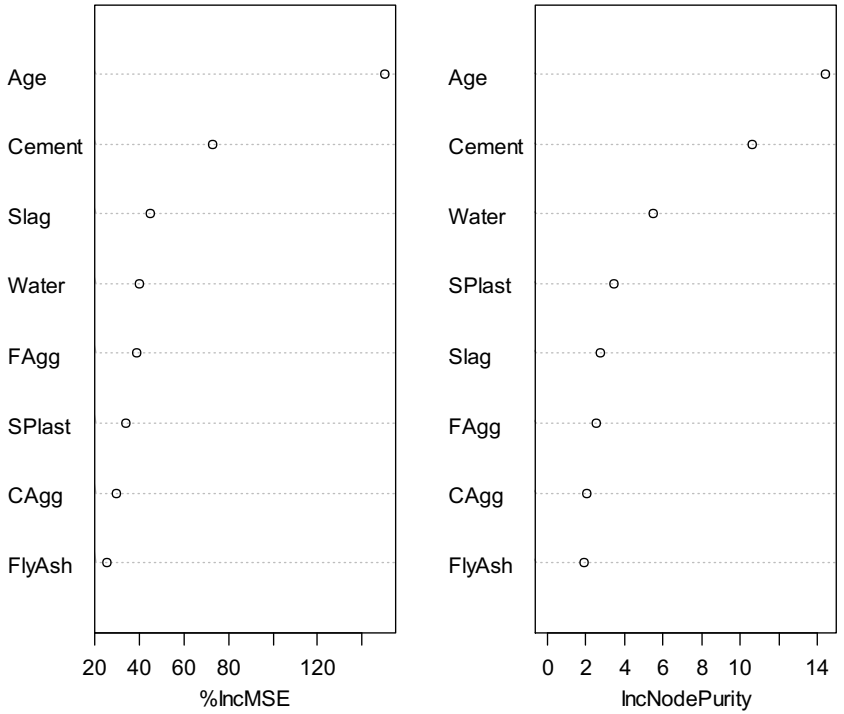
```
library(randomForest)
rForest1 <- randomForest(Strength~., data=CRT1, mtry=3, ntree = 500, nodesize = 3,
    importance = TRUE)
plot(rForest1)  #plots OOB mse vs # trees
rForest1 #check the OOB mse and r^2
importance(rForest1); varImpPlot(rForest1)
par(mfrow=c(2,4))
for (i in c(8,1,4,2,7,5,6,3)) partialPlot(rForest1, pred.data=CRT1, x.var = names(CRT1)[i],
    xlab = names(CRT1)[i], main=NULL) #creates "partial dependence" plots
par(mfrow=c(1,1))
c(rForest1$mse[rForest1$ntree], sum((rForest1$predicted -
    CRT1$Strength)^2)/nrow(CRT1)) #both give the OOB MSE
```

Random Forest Results for Concrete Data

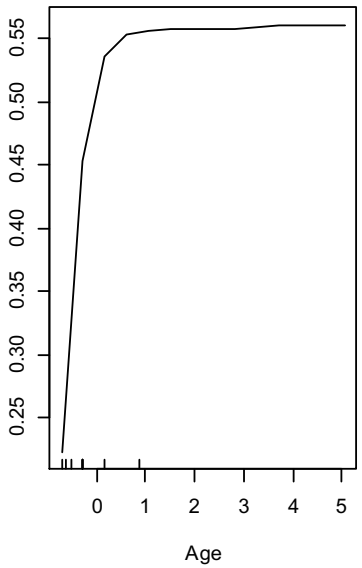
OOB MSE vs. # trees

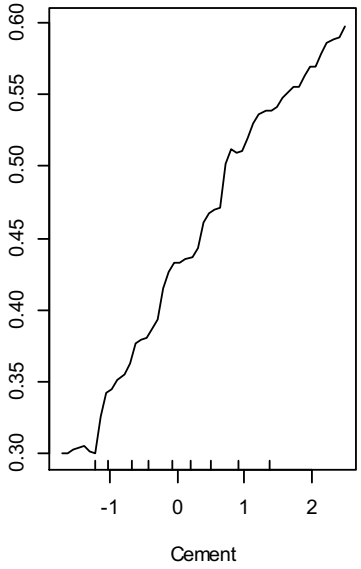


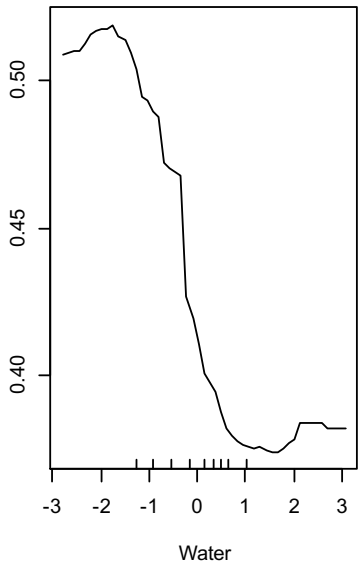
Variable Importance

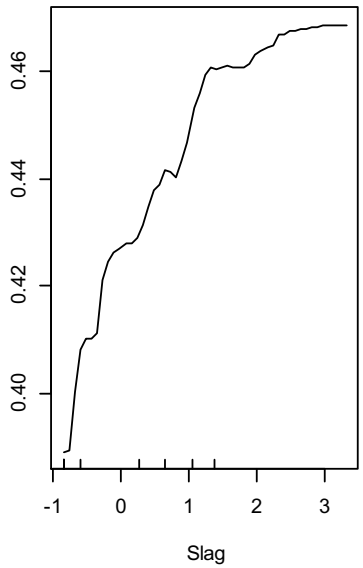


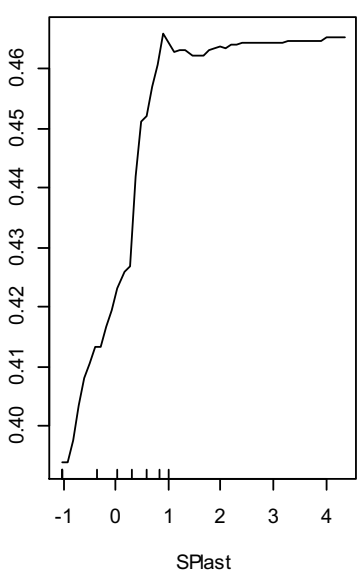
Partial Dependence (Marginal) Plots for Concrete RF

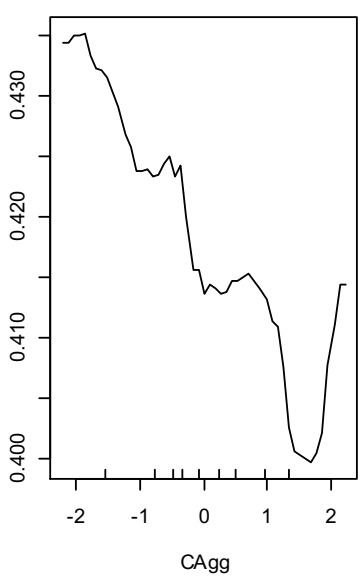


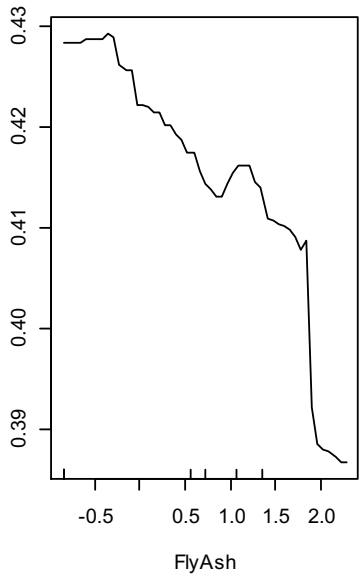












Discussion Points and Questions

- The out-of-bag (OOB) mse plotted vs. # trees replaces the CV mse for evaluating the model and is interpreted similarly.
- The individual trees in a random forest are generally much larger (controlled by `nodesize` and `maxnodes`) than the individual trees in a boosted tree. Choosing the right size for the individual trees in a random forest is important. You should try different values and use the OOB mse to select the best size.
- Random forests can overfit if the individual trees are chosen too large, but as the # trees increases they do not increasingly overfit, unlike boosted trees (why?), so you do not have to worry about choosing ntrees too large
- There are two built-in variable importance measures (permutation-based measure and the usual tree-based measure). It's good to look at both
- The partial dependence plots are the same as what is called "marginal" plots in the gbm package

A General Permutation-Based Variable Importance Measure (VIM)

- The split-by-split reduction in deviance as a VIM applies only to tree-based models. The permutation-based VIM was originally proposed in Breiman (2001, "Random Forests," Machine Learning) for random forests, but the concept is general and applies to any model.
- For the general approach, to find the VIM of each x_j:
  - Predict the response as usual in CV (or replace the CV predictions by the OOB predictions if using bagging). Call these predictions $\{\hat{y}_{i,CV}: i = 1,2,\dots,n\}$.
  - Repeat, but before plugging the training data array of predictors $\{\boldsymbol{x}_i: i = 1,2,\dots,n\}$ into the model, randomly permute the column for x_j. That is, randomly permute $\{x_{i,j}: i = 1,2,\dots,n\}$, while leaving $\{x_{i,l}: i = 1,2,\dots,n; l \neq j\}$ unchanged. Call these predictions $\{\hat{y}_{i,CV,\pi}: i = 1,2,\dots,n\}$, where π is for "permutation".
  - The VIM for x_j is the difference between the prediction accuracy for the regular predictions vs. the permuted predictions, using your favorite measure of prediction accuracy (e.g., $MSE_{CV,\pi} - MSE_{CV}$)

The Permutation-Based VIM in the randomForest Package

- As the accuracy measure, it uses MSE for regression and misclass rate for classification.
- For computational reasons:
  - It uses OOB predictions, instead of CV predictions
  - The accuracy difference is calculated for each individual tree in the random forest, and then this is averaged across all *ntree* trees
  - The average difference is then normalized by the standard deviation of the differences
- In general, when the predictor variables are highly correlated, the permutation-based VIM is subject to the same “extrapolation” problem as partial dependence plots (why?)

Comparison of CV r^2 for Various Methods for Concrete Data

	CV error SD	CV r^2
Boosted tree	3.91	0.945
Random Forest		0.921 (OOB r^2)
PPR	5.25	0.901
NNet	5.46	0.893
GAM	5.60	0.888
Loc. Lin.	7.27	0.811
Tree		0.793
K-NN	8.60	0.735
Lin. Reg.	10.5	0.605

Random Forests Vs. Boosted Trees

- Random forests have virtually all the same many advantages of boosted trees (see earlier slide on boosted tree advantages)
- They inherit almost all the advantages of trees, but, like boosted trees, usually have excellent predictive power
- Unlike trees, they lose interpretability. But this can be improved with the built-in partial dependence (aka marginal) plots and the variable importance measures
- Overall: random forests may usually give a little smoother model than boosted trees, whereas boosted trees may be a little better at capturing more complex nonlinearities

Random Forest (regression) for the Income Data

```
library(randomForest)
XX<-read.table("adult_train.csv",sep=",",header=TRUE,strip.white=TRUE,na.strings="?")
XX<-na.omit(XX)
INCOME<-XX  #there is no need to standardize the predictors with trees (why not)

rForest <- randomForest(x=INCOME[,-c(3,4,13)], y=INCOME$hours.per.week, data=XX,
    mtry=4, ntree = 100, nodesize = 50, importance = TRUE)
plot(rForest)  #plots OOB mse vs # trees
rForest #check the OOB mse and r^2
importance(rForest); varImpPlot(rForest)

i=1; partialPlot(rForest, pred.data=XX, x.var = names(XX)[i], xlab = names(XX)[i],
    main=NULL) #a partial dependence plot for age

#For the above parameters, the OOB r^2 was about 26.9%, which is about the same as
    the CV r^2 for the boosted tree
#using smaller nodesize and/or larger ntree in the above will take longer. Specifying the
    predictors "x =" and response "y = " as random.Forest arguments is a little faster than
    specifying the formula and data.frame.
```

Some Other Supervised Learning Methods

- Supervised learning models within proportional hazards models
- Quantile regression
- Support vector machines (SVM)
- Multivariate adaptive regression splines (MARS)
- Naive Bayes
- Gaussian process regression

- In light of the supervised learning methods that you have already learned, it should be relatively easy for you to learn any additional methods on your own (HTF discusses a number of such)

Proportional Hazards (PH) Survival Models

- Survival models are used to model time-to-event data in reliability (event = failure of a product), credit risk scoring (event = default on loan), healthcare (event = mortality), etc.
- The simplest case is where you just have a single variable $Y = T =$ time-to-failure, in which case you just use some appropriate failure time distribution like exponential, gamma, Weibull, log-normal.
- A "Cox proportional hazards" (PH) survival model is used when you also have a vector of predictor variables $\mathbf{X}$ and you want to model the failure-time distribution of $Y(= T)$ as a function of $\mathbf{X}$

Main Idea Behind PH Survival Models

For response $Y = T =$ time to failure and $\mathbf{X} =$ vector of predictors

pdf of T: $f(t)$

cdf of T: $F(t)$

survival function: $S(t) = 1 - F(t)$

hazard function: $h(t) = \frac{f(t)}{1-F(t)}$

Cox PH model: $h(t) = h_0(t)\exp\{g(\mathbf{X};\boldsymbol{\theta})\}$ where

$h_0(t)$ = some baseline hazard function (e.g., Weibull, gamma, etc)

Classic linear model: $g(\mathbf{X};\boldsymbol{\theta}) = \mathbf{X}^T\boldsymbol{\theta}$

Or can use any nonparametric $g(\mathbf{X};\boldsymbol{\theta})$ like a boosted tree (in gbm)

- In rpart, use "method=exp" with the response a survival object
- In gbm, use “distribution=coxph" with the response a survival object
- Censored observations are typical in survival data, so most PH survival packages will handle at least right-censored data

Quantile Regression

- Regression models predict the conditional mean $\mathrm{E}[Y|\boldsymbol{X}=\boldsymbol{x}]=g(\boldsymbol{x};\boldsymbol{\theta})$ as a function of $\boldsymbol{x}$
- Under the classic regression assumption that $Y=g(\boldsymbol{X};\boldsymbol{\theta})+\epsilon$ with iid ϵ, this essentially gives us the entire conditional distribution $f_{Y|\boldsymbol{X}}(y|\boldsymbol{x})$:
  - Estimate $f_\epsilon(\epsilon)$ as the marginal distribution of the regression errors $\{y_i-g(\boldsymbol{x}_i;\widehat{\boldsymbol{\theta}}):i=1,2,\ldots,n\}$
  - Take $f_{Y|\boldsymbol{X}}(y|\boldsymbol{x})=f_\epsilon(y-g(\boldsymbol{x};\widehat{\boldsymbol{\theta}}))$
- But what if the distribution of ϵ depends on $\boldsymbol{X}$?
  - If we are confident ϵ is zero-mean (which is always reasonable) and Gaussian with $\mathrm{Var}[Y|\boldsymbol{X}=\boldsymbol{x}]=\sigma^2(\boldsymbol{x})$ a function of $\boldsymbol{x}$, then we can try to estimate $\sigma^2(\boldsymbol{x})$ along with $\mathrm{E}[Y|\boldsymbol{X}=\boldsymbol{x}]=g(\boldsymbol{x};\boldsymbol{\theta})$ and use $f_{Y|\boldsymbol{X}}(y|\boldsymbol{x})=N(g(\boldsymbol{x};\widehat{\boldsymbol{\theta}}),\hat{\sigma}^2(\boldsymbol{x}))$
  - Quantile regression is more general

Basic Idea Behind Quantile Regression

- If we have a sample of univariate observations $\{y_i : i = 1,2, \dots, n\}$, the 0.5 sample quantile $q_{0.5}$ (i.e., the median) satisfies

  $q_{0.5} = argmin_q \sum_{i=1}^{n} |y_i - q| = argmin_q \sum_{i=1}^{n} \rho_{0.5}(y_i - q)$

  where $\rho_{0.5}(u) = 0.5|u| = u[0.5 - I(u < 0)]$ is the absolute value function (scaled by 0.5)

- More generally, for $\tau \in (0,1)$, the τ sample quantile q_τ satisfies

  $q_\tau = argmin_q \sum_{i=1}^{n} \rho_\tau(y_i - q)$

  where $\rho_\tau(u) = u[\tau - I(u < 0)] = \begin{cases} (1-\tau)|u|: & u < 0 \\ \tau|u|: & u \geq 0 \end{cases}$

  is the "tilted absolute value function"

- For a specified $\tau \in (0,1)$, quantile regression estimates the τ quantile of $f_{Y|\boldsymbol{X}}(y|\boldsymbol{x})$ (as a function of $\boldsymbol{x}$) as

  $q_\tau(\boldsymbol{x}; \widehat{\boldsymbol{\theta}}) = argmin_{\boldsymbol{\theta}} \sum_{i=1}^{n} \rho_\tau(y_i - q_\tau(\boldsymbol{x}; \boldsymbol{\theta}))$

  where $q_\tau(\boldsymbol{x}; \boldsymbol{\theta})$ is some parametric (or nonparametric) function of $\boldsymbol{x}$

Basic Support Vector Classifier Concepts

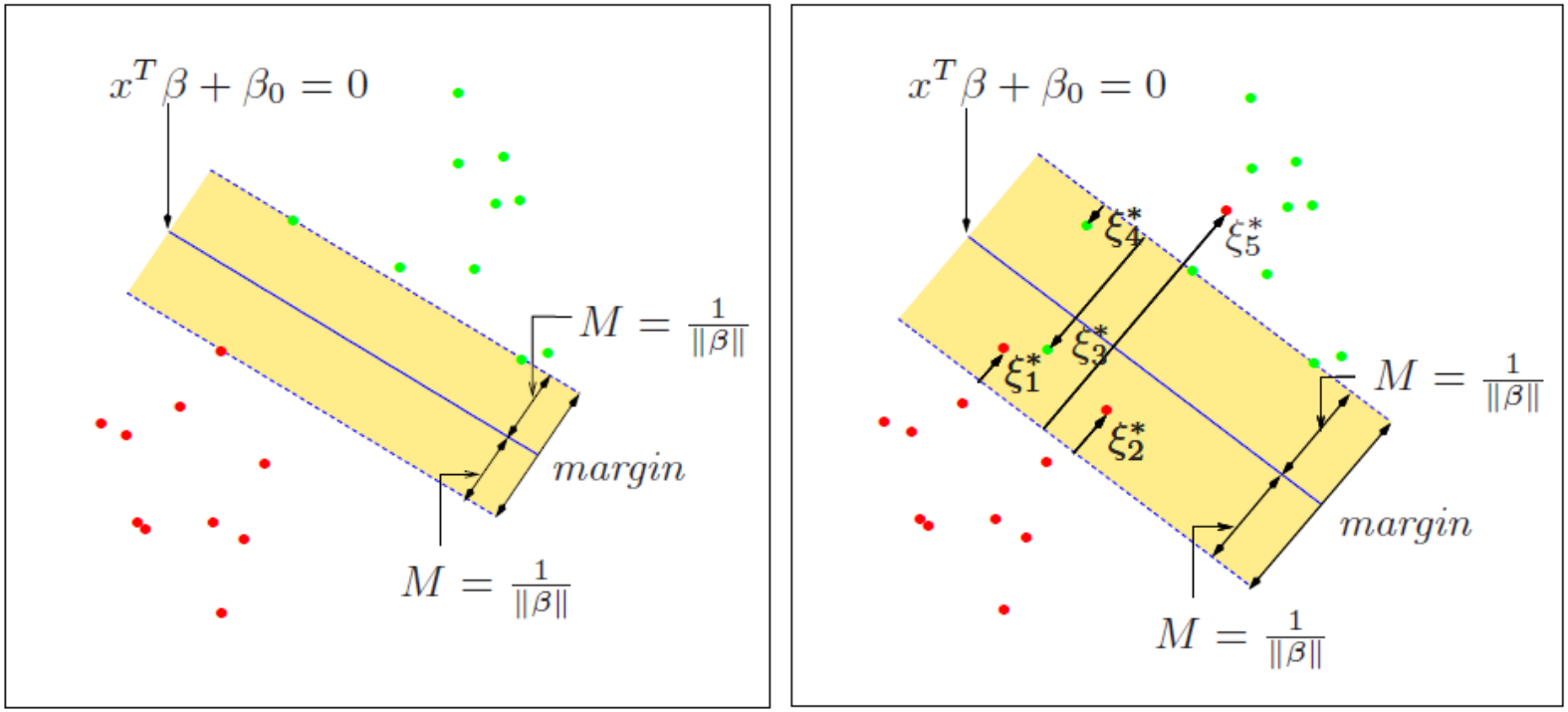


FIGURE 12.1. *Support vector classifiers. The left panel shows the separable case. The decision boundary is the solid line, while broken lines bound the shaded maximal margin of width* $2M = 2/\|\beta\|$. *The right panel shows the nonseparable (overlap) case. The points labeled* ξ_j^* *are on the wrong side of their margin by an amount* $\xi_j^* = M\xi_j$; *points on the correct side have* $\xi_j^* = 0$. *The margin is maximized subject to a total budget* $\sum \xi_i \leq$ *constant. Hence* $\sum \xi_j^*$ *is the total distance of points on the wrong side of their margin.*

Basic Naive Bayes Concepts

- Consider the ideal Bayes classifier (see last few slides of Notes 1), which is based on the posterior class probabilities

$$f_{Y|\mathbf{X}}(y|\mathbf{x}) = \frac{f_{\mathbf{X}|Y}(\mathbf{x}|y)f_Y(y)}{f_{\mathbf{X}}(\mathbf{x})} = \frac{f_{\mathbf{X}|Y}(\mathbf{x}|y)f_Y(y)}{\sum_{j=1}^{K} f_{\mathbf{X}|Y}(\mathbf{x}|j)f_Y(j)}$$

- The biggest problem with using the ideal Bayes classifier is that it is virtually impossible to reliably and nonparametrically estimate the within-class predictor distributions $\{f_{\mathbf{X}|Y}(\mathbf{x}|y): y = 1, 2, \ldots, K\}$, because of the curse of dimensionality
- Linear and quadratic discriminant analysis assume (parametric) multivariate normal distributions for $\{f_{\mathbf{X}|Y}(\mathbf{x}|y): y = 1, 2, \ldots, K\}$
- In contrast, the naive Bayes classifier assumes the predictors are conditionally independent within-class, i.e., that

$$f_{\mathbf{X}|Y}(\mathbf{x}|y) = \prod_{j=1}^{k} f_{X_j|Y}(x_j|y)$$

and uses nonparametric *univariate* density estimators for $\{f_{X_1|Y}(x_1|y), f_{X_2|Y}(x_2|y), \ldots, f_{X_k|Y}(x_k|y): y = 1, 2, \ldots, K\}$