5. Stop the backward deletion by optimizing a GCV score.

For example, the trace of solutions could look as follows:

$$\begin{aligned}
h_0(x) &= 1, \quad \mathcal{M} = \{1\}, \\
h_1(x) &= (x_2 - x_{72})_+, \quad h_2(x) = (x_{72} - x_2)_+, \quad \mathcal{M} = \{1, (x_2 - x_{72})_+, (x_{72} - x_2)_+\}, \\
h_3(x) &= (x_{72} - x_2)_+ \cdot (x_1 - x_{51})_+, \quad h_4(x) = (x_{72} - x_2)_+ \cdot (x_{51} - x_1)_+, \\
& \mathcal{M} = \{1, (x_2 - x_{72})_+, (x_{72} - x_2)_+, \ (x_{72} - x_2)_+ \cdot (x_1 - x_{51})_+, \ h_4(x)\ \} \\
\ldots
\end{aligned}$$

7.3.1 Hierarchical interactions and constraints

It becomes clear from the definition of MARS, that the algorithm proceeds hierarchically in the sense that a d-order interaction can only enter the model when an interaction of degree $d-1$ involving one predictor less is already in the model. For example, an interaction of degree 4 involving x_2, x_4, x_7, x_9 enters the model because an interaction of degree 3 involving the predictors x_2, x_4, x_7 is already in the model.

Quite often it is useful to restrict interactions to degree 2 or 3. Sometimes, we may even restrict the interaction degree to 1: MARS then yields an additive model where the predictor variables and the piecewise linear spline basis functions are included in a forward-backward adaptive way.

7.3.2 MARS in R

MARS is implemented in R in the function `mars` from the package `mda`. The syntax is as follows:

```
fit <- mars(x, y, degree=2)
#x is the n*p design matrix; y is the response vector

predict(fit,x=xnew)
#xnew is a new (test) set of predictors of dimension ntest*p
```

7.4 Neural Networks

Neural networks have been very popular in the 90's in the machine learning and artificial intelligence communities. From a statistical perspective, they can be viewed as high-dimensional nonlinear regression models.

We will focus here on feedforward neural nets with one hidden layer. The model is

$$g_k(x) = f_0\left(\alpha_k + \sum_{h=1}^{q} w_{hk}\phi\big(\tilde{\alpha}_h + \sum_{j=1}^{p} w_{jh}x_j\big)\right). \tag{7.2}$$

which is predicting multivariate $(g_k)_k$ where for regression only g_0 is used and for classification, one typically uses $g_0, \ldots, g_{J-1}$ and $\hat{g}(x) := \arg\max_j \hat{g}_j(x)$ which is called “softmax” in the NN literature. The function $\phi(\cdot)$ is usually the sigmoid function

$$\phi(x) = \frac{\exp(x)}{1 + \exp(x)},$$

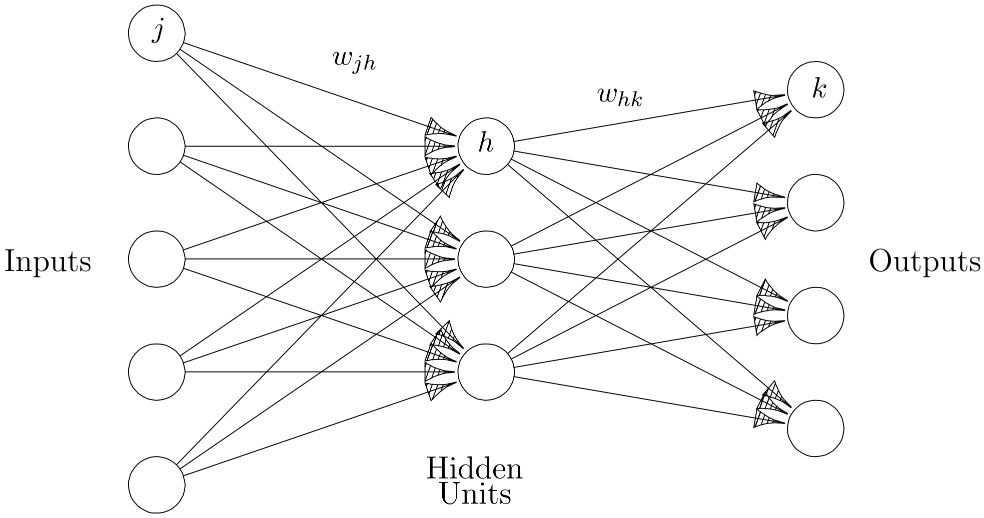

Figure 7.5: *Simple "feed-forward" neural net with one hidden layer.*

wheras $f_0(\cdot)$ is typically chosen as the identity for regression and as the sigmoid $\phi(x)$ for softmax classification. The $w_{hk}, w_{jk}, \tilde{\alpha}_h, \alpha_k$ all are unknown parameters. The so-called input layer consists of the p predictor variables; the values $w_{hk}\phi\left(\alpha_h + \sum_{j=1}^{p} w_{jh}x_j\right)$ $(h = 1, \ldots, q)$ make the so-called hidden layer with q units. And the so-called output layer is just a single unit for univariate regression.

A useful variant of (7.2) is a model which includes a linear regression component:

$$g(x) = f_0\left(\alpha + \sum_{j=1}^{p} w_{j,lin}x_j + \sum_{k=1}^{q} w_k\phi(\alpha_k + \sum_{j=1}^{p} w_{jk}x_j)\right). \tag{7.3}$$

7.4.1 Fitting neural networks in R

Feedforward neural nets can be fitted in R with the function `nnet` from the package `nnet`.

In practice, it can be very important to center and scale all the predictor variables so that they are approximately on the same scale (about "1"). This avoids that gradient methods for optimizing the likelihood get stuck in the "flat regions" of the sigmoid functions. Further note that one can regularize the problem using so called *weight decay* which stabilizes the algorithm (less dependence on random starting values $w_{*,*}$) and diminishes the importance of choosing the number of hidden units q.

```
library(nnet)
## Using 'd.ozone' data from above, using  log(O3)
## and  scaling the x-variables (to mean = 0, sd = 1):
sc.ozone <- data.frame(scale(d.ozone)[, -1],
                       log.O3 = log(d.ozone[,"ozone"]))

> set.seed(22)  ## (also try others; nnet() uses random starting values!)
> fit <- nnet(log.O3 ~ . , data= sc.ozone, size = 3, skip = TRUE,
+             decay = 4e-4, linout = TRUE, maxit = 500)
 # weights:  43
initial  value 2943.804833
iter  10 value 568.319707
```

```
......
......
final  value 34.631788
converged
> sum(residuals(fit)^2) # -> 34.211
> summary(fit)
 a 9-3-1 network with 43 weights
a 9-3-1 network with 43 weights
options were - skip-layer connections  linear output units  decay=4e-04
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1 i7->h1 i8->h1 i9->h1
 -2.28   1.27  -0.34  -2.57   1.46   0.03   0.10  -1.02  -0.39  -0.33
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2 i7->h2 i8->h2 i9->h2
-12.43   5.08   2.04   8.19  -7.66  -7.01   2.40  -0.31   3.58  -1.19
 b->h3 i1->h3 i2->h3 i3->h3 i4->h3 i5->h3 i6->h3 i7->h3 i8->h3 i9->h3
-19.79  -6.65   1.49  -4.53  -3.95   2.28   6.06   5.19  10.06  -0.20
  b->o  h1->o  h2->o  h3->o  i1->o  i2->o  i3->o  i4->o  i5->o  i6->o
  2.50  -1.81   0.68   0.71   0.11  -0.09  -0.49   0.72   0.01  -0.03
 i7->o  i8->o  i9->o
  0.03  -0.29  -0.15


## without linear model component: skip=FALSE
set.seed(22)
> fit1 <- nnet(log.O3 ~ . , data= sc.ozone, size = 3, skip=FALSE,
+               decay = 4e-4, linout = TRUE, maxit = 500)
 ..........
final  value 42.909
converged
> sum(residuals(fit1)^2) # 41.865
> summary(fit1)
 a 9-3-1 network with 34 weights
options were - linear output units  decay=4e-04
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1 i7->h1 i8->h1 i9->h1
-27.22   8.91   0.51   8.78  -8.26  10.55  -8.46  11.89 -11.51   8.51
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2 i7->h2 i8->h2 i9->h2
  0.09  -0.15  -0.07   0.07   1.49  -0.43   0.05  -0.41  -0.01  -0.24
 b->h3 i1->h3 i2->h3 i3->h3 i4->h3 i5->h3 i6->h3 i7->h3 i8->h3 i9->h3
-16.97   6.53   0.56   2.30 -20.19   0.64   6.34  17.51  -1.67  -3.21
  b->o  h1->o  h2->o  h3->o
  0.86   0.46   2.42   0.86
```

`linout=TRUE` indicates that the function is fitted on the linear regression scale; for classification, we use the default `linout=FALSE`. `size = 3` chooses 3 hidden units, regularized by weight `decay = 4e-4`: these are tuning parameters, and `skip=TRUE` enforces a neural net with a linear model component. The random seed is stored because, by default, `nnet()` uses random starting values for the high-dimensional optimization.

7.5 Projection pursuit regression

Projection pursuit regression (for regression problems) bears some similarities to feed-forward neural networks. Instead of taking a linear combination of q different sigmoid function outputs (from the q hidden units in the hidden layer) in (7.2), we use the following model:

$$g_{\mathrm{PPR}}(x) = \mu + \sum_{k=1}^{q} f_k(\sum_{j=1}^{p} \alpha_{jk} x_j), \quad \text{where}$$
$$\textstyle\sum_{j=1}^{p} \alpha_{jk}^2 = 1, \quad \mathbb{E}[f_k(\sum_{j=1}^{p} \alpha_{jk} X_j)] = 0, \quad \text{for all } k.$$

The functions $f_k(\cdot) : \mathbb{R} \to \mathbb{R}$ are nonparametric (i.e. "arbitrary" smooth); the linear combinations $\sum_{j=1}^{p} \alpha_{jk} x_j$ are linear projections: For the unit vector $\boldsymbol{\alpha}_k = (\alpha_{1k}, \ldots, \alpha_{pk})^\intercal$, $\boldsymbol{\alpha}_k{}^\intercal x$ is the projection of x onto (a ray through $\vec{0}$ in direction) $\boldsymbol{\alpha}_k$. Note that the function $x \mapsto f_k(\sum_{j=1}^{p} \alpha_{jk} x_j)$ only varies along the direction $\boldsymbol{\alpha}_k$, hence the f_k's are called *ridge functions* and the model "*projection pursuit*". This model typically requires much smaller q than the hidden units in a neural network, at the expense of estimating ridge functions rather than using fixed sigmoid functions in neural nets.

Estimation of projection pursuit can be done using a backfitting algorithm. Quite often, projection pursuit yields very competitive prediction performance when choosing a reasonable number of ridge functions (e.g. by optimizing a CV score).

7.5.1 Projection pursuit regression in R

The function `ppr` in R can be used for fitting projection pursuit regression. The function in `ppr` re-scales the projection pursuit model to

$$g_{\mathrm{PPR}}(x) = \mu + \sum_{k=1}^{q} \beta_k f_k(\sum_{j=1}^{p} \alpha_{jk} x_j),$$
$$\sum_{j=1}^{p} \alpha_{jk}^2 = 1, \quad \mathbb{E}[f_k(\sum_{j=1}^{p} \alpha_{jk} X_j)] = 0, \quad \mathrm{Var}(f_k(\sum_{j=1}^{p} \alpha_{jk} X_j)) = 1, \quad \text{for all } k.$$

Consider the ozone data set from above.

```
fit <- ppr(log(O3) ~ . , data = d.ozone, nterms=4)
                              # nterms specifies the number of ridge functions
sfsmisc::mult.fig(4) # or just  par(mfrow=c(2,2))
plot(fit) ## 4 terms -> 4 plots

predict(fit,xnew) # where 'xnew' is a new set of predictor variables
```

The estimated ridge functions are shown in Figure 7.6.

7.6 Classification and Regression Trees (CART)

Quite different from the previous methods are the so-called tree models. CART is the most well known tree model or algorithm in the statistics community.

The underlying model function for CART is

$$g_{tree}(x) = \sum_{r=1}^{R} \beta_r \mathbf{1}_{[x \in \mathcal{R}_r]},$$

where $\mathcal{P} = \{\mathcal{R}_1, \ldots, \mathcal{R}_R\}$ is a partition of $\mathbb{R}^p$. Thus, the function $g(\cdot)$ is modelled as *piecewise constant.*

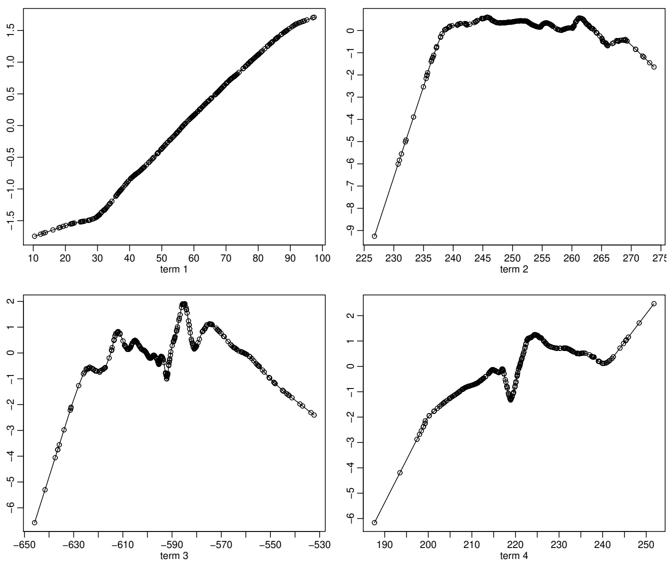

Figure 7.6: Four estimated ridge functions $f_k(\cdot)$ from a projection pursuit fit to the ozone data (`log(O3) ~ .`).

7.6.1 Tree structured estimation and tree representation

Parameter estimation $\hat{\beta}_1, \ldots, \hat{\beta}_R$ is easy when the partition $\mathcal{P} = \{\mathcal{R}_1, \ldots, \mathcal{R}_R\}$ would be given. We use

$$\hat{\beta}_r = \sum_{i=1}^{n} Y_i \mathbf{1}_{[x_i \in \mathcal{R}_r]} / \sum_{i=1}^{n} \mathbf{1}_{[x_i \in \mathcal{R}_r]}$$

for regression and binary classification (we do not model $g(\cdot)$ on the logistic scale). (For J class problems with $J > 2$, we can also use the tree model directly and do not need to go with a one against the rest approach, see section 6.4.2).

The tricky issue is to get a data-driven estimate for the partition $\mathcal{P}$. Some restrictions will be made to obtain a computationally feasible algorithm. This will be discussed next.

7.6.2 Tree-structured search algorithm and tree interpretation

We restrict the search for a good partition to partition cells $\mathcal{R}$ which are **axes parallel rectangles**. Moreover, we proceed in a **greedy** way since the space of partitions consisting of axes parallel rectangles is still huge.

A tree-structured greedy algorithm then proceeds as follows.

1. Start with $R = 1$ subset, $\mathcal{P} = \{\mathcal{R}\} = \mathbb{R}^p$.

2. Refine $\mathcal{R}$ into $\mathcal{R}_{left} \cup \mathcal{R}_{right}$ where:

$$\begin{aligned} \mathcal{R}_{left} &= \mathbb{R} \times \mathbb{R} \times \ldots \times (-\infty, d] \times \mathbb{R} \ldots \times \mathbb{R}, \\ \mathcal{R}_{right} &= \mathbb{R} \times \mathbb{R} \times \ldots \times (d, \infty) \times \mathbb{R} \ldots \times \mathbb{R}, \end{aligned}$$

   where one of the axes is split at the split point d, where d is from the finite set of mid-points between observed values. The search for the axes to split and the split point d are determined such that the negative log-likelihood is maximally reduced with the refinement (search over $j \in \{1, \ldots, p\}$ and $d \in$ {mid-points of observed values}). Build the new partition $\mathcal{P} = \{\mathcal{R}_1, \mathcal{R}_2\}$ with $\mathcal{R}_1 = \mathcal{R}_{left}, \ \mathcal{R}_2 = \mathcal{R}_{right}$.

3. Refine the current partition $\mathcal{P}$ as in step 2 by refining **one** of the partition cells from the current partition $\mathcal{P}$. That is, we search for the best partition cell to refine which includes a search as in step 2 for the best axes to split and the best split point. Then, we up-date the partition:

   $\mathcal{P} = \mathcal{P}_{old} \setminus$ partition cell selected to be refined $\cup$ {refinement cells $\mathcal{R}_{left}, \mathcal{R}_{right}$}.

4. Iterate step 3 for a large number of partition cells.

5. Backward deletion: prune the tree (see below) until a reasonable model size, typically determined via cross-validation, is achieved.

Tree representation

The search algorithm above has a useful tree representation. Figures 7.7 and 7.8 show for part of the kyphosis dataset (2 class problem but now only with two predictor variables instead of 3) how $\mathbb{R}^2$ is recursively partitioned in a tree-structured way. As a side result, we obtain a very useful interpretation of the model in terms of a tree! Such a decision tree

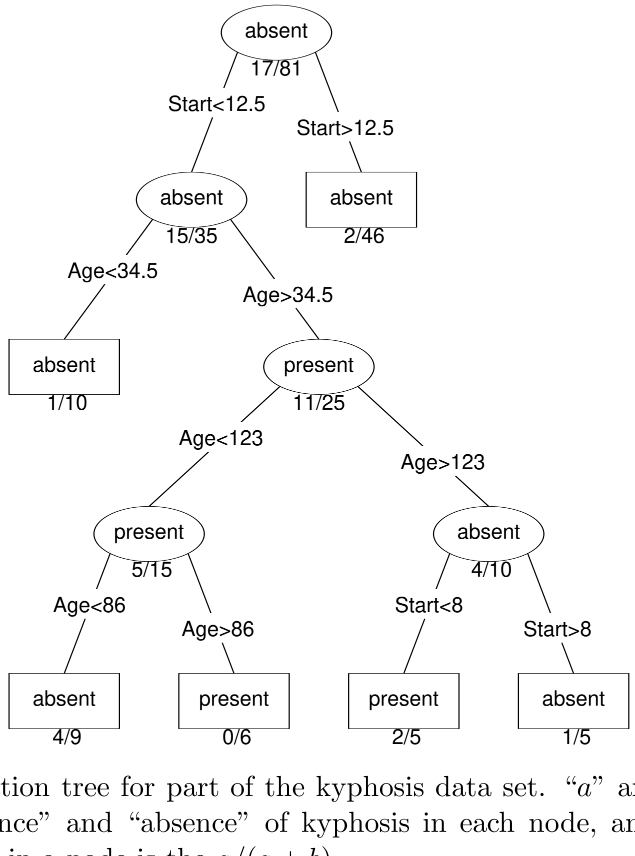

Figure 7.7: Classification tree for part of the kyphosis data set. "a" and "b" denote the frequencies of "presence" and "absence" of kyphosis in each node, and the probability estimate for kyphosis in a node is the $a/(a + b)$.

corresponds to a **recursive** partitioning scheme as shown in Figure 7.8.

The process of backward deletion in step 5 can now be more easily understood. Steps 1–4 result in a large tree τ_M ($M = R - 1$). Backward elimination, or tree pruning, then