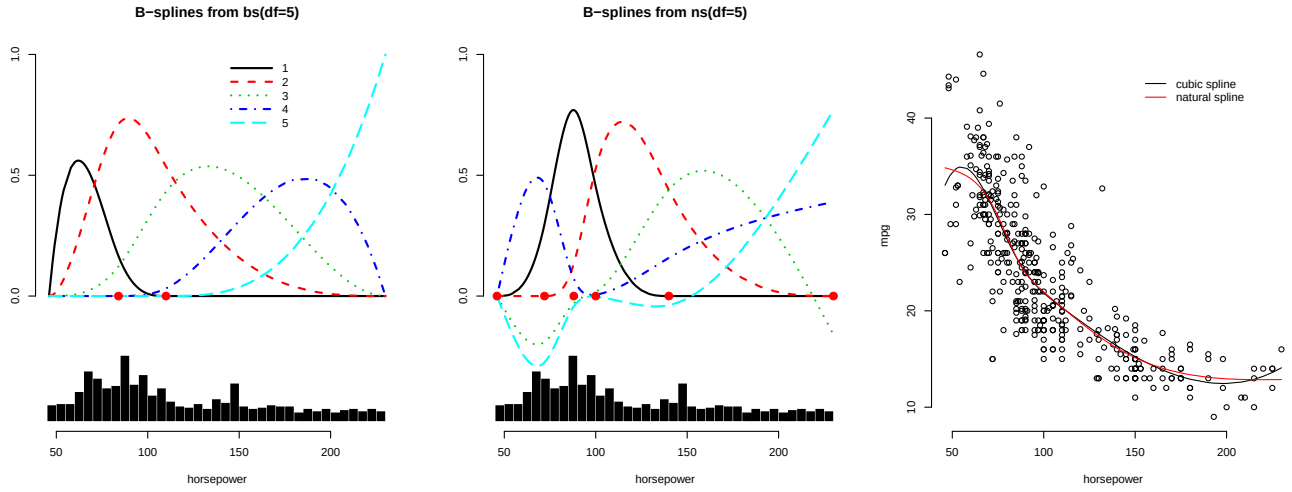
7 PQHS 471 Notes Week 7

7.1 Week 7 Day 1

Basis splines are functions of `x`. To visualize the B-splines from `bs()` and `ns()`:

```
attach(ISLR::Auto)

library(splines)
idx = order(horsepower)
xhist = hist(horsepower, breaks=50, plot=FALSE)

bsmatrix = bs(horsepower, df=5)
matplot(horsepower[idx], bsmatrix[idx,], type='l', lwd=2, bty='n', ylim=c(-0.5,1), yaxt='n',
        xlab='horsepower', ylab='', main='B-splines from bs(df=5)')
axis(2, seq(0,1,.5))
points(quantile(horsepower, probs=1:2/3), c(0,0), col=2, pch=19, cex=1.5)
legend(140, 1, col=1:5, lty=1:5, lwd=2, bty='n', seg.len=4, legend=1:5)
with(xhist, segments(mids, -0.5, mids, counts/200-0.5, lwd=8, lend=2))

nsmatrix = ns(horsepower, df=5)
matplot(horsepower[idx], nsmatrix[idx,], type='l', lwd=2, bty='n', ylim=c(-0.5,1), yaxt='n',
        xlab='horsepower', ylab='', main='B-splines from ns(df=5)')
axis(2, seq(0,1,.5))
points(c(range(horsepower), quantile(horsepower, probs=1:4/5)), rep(0,6),
       col=2, pch=19, cex=1.5)
with(xhist, segments(mids, -0.5, mids, counts/200-0.5, lwd=8, lend=2))

bs.mod1 = lm(mpg ~ bs(horsepower, df=5))
ns.mod1 = lm(mpg ~ ns(horsepower, df=5))
plot(horsepower, mpg, bty='n')
points(horsepower[idx], fitted(bs.mod1)[idx], type='l', col=1)
points(horsepower[idx], fitted(ns.mod1)[idx], type='l', col=2)
legend(150, 45, col=1:2, lty=1, bty='n',
       legend=c("cubic spline", "natural spline"))

detach()
```



7.1.1 ISLR 7.5

Smoothing splines start with a very different motivation. Consider all smooth functions $g(x)$ such that $g''(x)$ exists. We optimize the following

$$\text{minimize}_g \sum_i (y_i - g(x_i))^2 + \lambda \int g''(t)^2 dt. \tag{7.11}$$

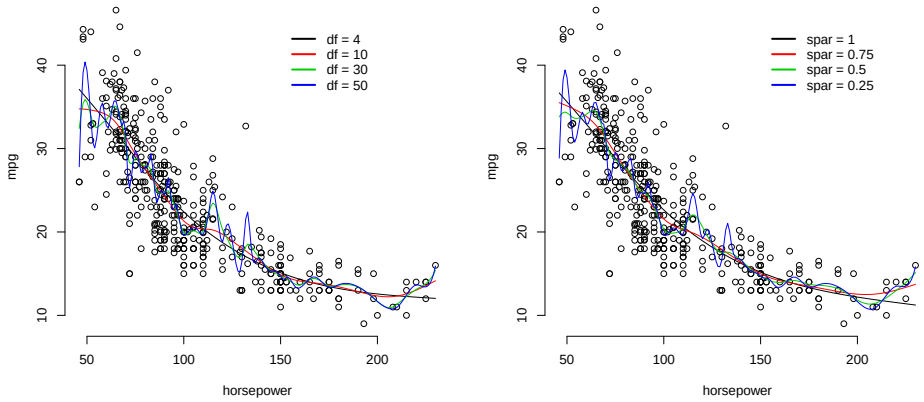
- A high $|g''(t)|$ reflects a quick change of $g'(t)$ at t, which happens when $g(t)$ is bumpy at t. So $\int g''(t)^2 dt$ is a way to measure the overall level of bumpiness/roughness of $g(t)$.
  - When $g(t) = \beta_0 + \beta_1 t$, $g''(t) = 0$ and $\int g''(t)^2 dt = 0$.
  - When $g(t) = \beta_0 + \beta_1 t + \beta_2 t^2$, $g''(t) = 2\beta_2$ and $\int_{x_{(1)}}^{x_{(n)}} g''(t)^2 dt = 4\beta_2^2 (x_{(n)} - x_{(1)})$. When $|\beta_2| = \frac{1}{\sqrt{8}} = 0.35$, the overall "roughness" of a quadratic function is similar to that of $\sin(t)$.
  - When $g(t) = \sin(t)$, $g''(t) = -\sin(t)$ and $\int_{x_{(1)}}^{x_{(n)}} g''(t)^2 dt = \int_{x_{(1)}}^{x_{(n)}} \sin^2(t) dt \approx \frac{1}{2}(x_{(n)} - x_{(1)})$. (The antiderivative of $\sin^2(t)$ is $\frac{1}{2}[t - \frac{1}{2}\sin(2t)]$.)
- It can be shown that the solution $g(x)$ to (7.11) must be a natural spline with all x_i being knots. So, a smoothing spline is a regularized natural spline. It is generalized ridge regression with a closed-form solution.

Effective degrees of freedom: The closed-form solution leads to a formula for fitted values: $\hat{y}_\lambda = S_\lambda y$, where S_λ is an $n \times n$ matrix. The effective DF is $\text{trace}(S_\lambda) = \sum_{i=1}^n \{S_\lambda\}_{ii}$. (Rationale: In linear regression, let X be the $n \times p$ design matrix. Then $\hat{y} = Hy$, where $H = X(X'X)^{-1}X'$. The trace of H is p, the degrees of freedom.)

- The effective degrees of freedom may not be integers. The value increases gradually as the penalty λ decreases.
- Instead of specifying λ, one can specify a desired DF or desired smoothness using a smoothing parameter.

R has a function `smooth.spline(x, y)`. Note the order of `x` and `y`. Use `df=` to specify the desired effective DF, or `spar=` to specify the desired smoothness (smoothing parameter in $(0, 1]$). The higher `df` the rougher, and the lower `spar` the rougher.

```
require(ISLR)
sms = with(Auto, smooth.spline(horsepower, mpg, df=4))
names(sms)

par(mfrow=c(1,2), cex=.7)
xgrid = seq(46,230)  ## for drawing fitted curves
colseq = 1:4

with(Auto, plot(horsepower, mpg, bty='n'))
dfseq = c(4, 10, 30, 50)
for(i in 1:4) {
  tmp = with(Auto, smooth.spline(horsepower, mpg, df=dfseq[i]))
  lines(predict(tmp, xgrid), col=colseq[i])
}
legend(150, 45, col=colseq, lty=1, lwd=2, bty='n', legend = paste("df =", dfseq))

with(Auto, plot(horsepower, mpg, bty='n'))
sparseq = (4:1)/4
for(i in 1:4) {
  tmp = with(Auto, smooth.spline(horsepower, mpg, spar=sparseq[i]))
  lines(predict(tmp, xgrid), col=colseq[i])
}
legend(150, 45, col=colseq, lty=1, lwd=2, bty='n', legend = paste("spar =", sparseq))
```



Choosing hyperparameter lambda:

- Cross-validation always works. This is a computational approach.
- Traditional formula-based CV approaches: Let S_λ be the matrix for λ such that $\hat{y}_\lambda = S_\lambda y$.
  - Leave-one-out CV (LOOCV): Select λ that minimizes $\text{err}_\lambda = \sum_i (y_i - \hat{y}_{i|-i})^2 = \sum_i \left[\frac{y_i - \hat{y}_{\lambda,i}}{1-\{S_\lambda\}_{ii}}\right]^2$.
  - Generalized CV (GCV): Select λ that minimizes $V(\lambda) = \frac{\frac{1}{n}\sum (y_i - \hat{y}_{\lambda,i})^2}{[\frac{1}{n} tr(I-S_\lambda)]^2}$. (Motivation: $V(\lambda) = \frac{1}{n}\sum_i (y_i - \hat{y}_{i|-i})^2 w_i^2(\lambda)$, where $w_i(\lambda) = \frac{1-\{S_\lambda\}_{ii}}{\frac{1}{n} tr(I-S_\lambda)}$. Note that $\frac{1}{n}\sum_i w_i(\lambda) = 1$.)

In `smooth.spline()`, when both `df` and `spar` are not specified, a formula-based CV will be performed. The default (`cv=F`) is the GCV. When `cv=T`, the LOOCV is performed. The document of `smooth.spline()` suggests to use GCV when there are duplicated points in `x`, which is the case in the example below.

For regression analysis of `mpg` on `horsepower`, these two versions of CV give quite different results. The GCV result seems not desirable.

```
library(ISLR)
sms.cvT = with(Auto, smooth.spline(horsepower, mpg, cv=T))  ## LOOCV
sms.cvF = with(Auto, smooth.spline(horsepower, mpg, cv=F))  ## GCV
sms.cvT$df; sms.cvF$df  ## so different
sms.cvT$lambda; sms.cvF$lambda

xgrid = seq(46,230)  ## for drawing fitted curves
with(Auto, plot(horsepower, mpg, bty='n'))
lines(predict(sms.cvT, xgrid), col=1)
lines(predict(sms.cvF, xgrid), col=2)
```

7.1.2 ISLR 7.6

Local regression: Moving-window weighted regression.

- Similar to kNN. Both are *memory-based* (in contrast to *formula-based*), because the training data are needed when computing a prediction. (Or, a fine grid of results need to be stored.)
- Factors to specify:
  - Span: Fraction of data used for every point x_0. It is a hyperparameter that could be selected with CV.
  - Kernel (weight function): How data are weighted (as a function of relative distance between x and x_0).
  - Regression model (and fitting criterion)
- Technically, all these factors are hyperparameters, although in practice they are more often specified by users than selected using CV.

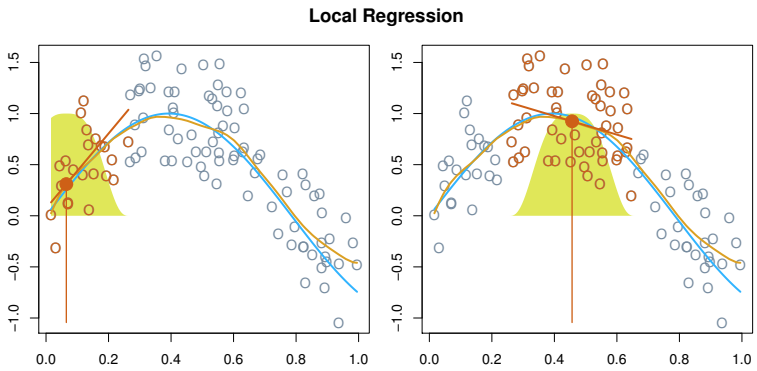


Figure 7.9 and Algorithm 7.1 in ISLR describe local *linear* regression. In the R function `loess()`, the default is `degree = 2` (local *quadratic* model fitted with least squares).

In `loess()`, the default neighborhood is `span = 0.75` (75% data are used for every point). One can also specify the desired DF (approximate "equivalent number of parameters") with the `enp.target=` option. When doing moving average (`degree=0`), span should be much smaller than the default of 0.75.

```
attach(ISLR::Auto)
aa = loess(mpg ~ horsepower)
aa  ## Note the ENP is provided
summary(aa)

idx = order(horsepower)
plot(horsepower, mpg)
lines(horsepower[idx], predict(loess(mpg ~ horsepower))[idx], col=1)   ## quadratic
lines(horsepower[idx], predict(loess(mpg ~ horsepower, degree=1))[idx], col=2)  ## linear
lines(horsepower[idx], predict(loess(mpg ~ horsepower, degree=0))[idx], col=3)  ## span=.75 too large
lines(horsepower[idx], predict(loess(mpg ~ horsepower, degree=0, span=.2))[idx], col=4)
lines(horsepower[idx], predict(loess(mpg ~ horsepower, enp.target=10))[idx], col=1, lty=2)
detach()
```

Note that `loess()` is prefered to the older version `lowess()`. These functions have different defaults.

High-dimensional loess: The default `span=0.75` is often too high. For example, suppose x_1 and x_2 are two predictors with values uniformly distributed in $[0,1]$. A span of 75% neighboring points in the space of $[0,1] \times [0,1]$ is effectively a span of 86.6% on x_1 and x_2 (because $0.866^2 = 0.75$). For 3-dimensional predictors, a span of 75% in 3D is effectively a span of 90.9% on each predictor (because $0.909^3 = 0.75$).

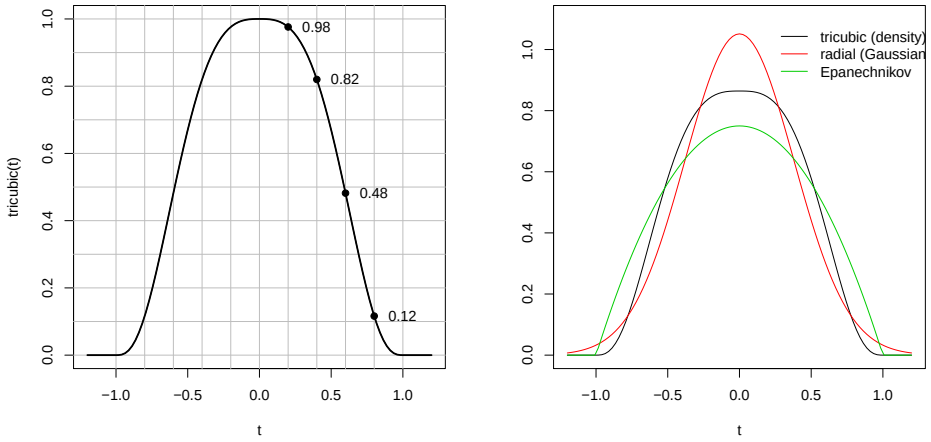
Tricubic function: The weight function used in `loess()` is tricubic: $f(t) = (1 - |t|^3)^3 I(|t| \leq 1)$, where $t = dist(x, x_0)/\text{maxd}$ is the relative distance of x from x_0, with maxd being the maximum distance from x_0 in the neighborhood of x_0.

```
tricubic = function(x) (1 - abs(x)^3)^3 *(abs(x)<=1)
curve(tricubic(x), xlim=c(-1.2,1.2), xlab='t', ylab='tricubic(t)')
abline(h=seq(0,1,0.1), v=seq(-1,1,0.2), col='grey')
curve(tricubic(x), lwd=2, add=T)
xgrid = seq(.2, .8, .2)
points(xgrid, tricubic(xgrid), pch=19)
text(xgrid, tricubic(xgrid), round(tricubic(xgrid), 2), adj=-.5)
```

Compare tricubic with other kernel functions. The tricubic function has AUC 1.157. Thus the corresponding density function is $f(t)/1.157$, which has mean 0 and variance 0.144. For comparison, the Epanechnikov kernel is $f(t) = .75(1 - t^2)I(|t| \leq 1)$, which has AUC 1 (i.e., it is a density function), mean 0, and variance 0.2. The Epanechnikov kernel is not smooth at -1 and 1.

```
integrate(tricubic, -1, 1)  ## AUC is 1.157143
integrate(function(x) x^2 * tricubic(x)/1.157143, -1, 1)  ## VAR is 0.1440329
```

```
## tricubic density function
curve(tricubic(x)/1.157143, xlim=c(-1.2, 1.2), ylim=c(0,1.1), xlab='t', ylab='')
curve(dnorm(x, 0, sqrt(0.1440329)), add=T, col=2)  ## Gaussian kernel with same variance
curve(.75*(1-x^2) *(abs(x)<=1), add=T, col=3)  ## Epanechnikov kernel
legend(1,1, col=1:3, lty=1, bty='n',
       legend=c("tricubic", "radial (Gaussian)", "Epanechnikov"))
```



7.1.3 Assignment

1. Reading for next lecture: ISLR 8; HOML Chapter 6
2. ISLR Chapter 8 R Labs

7.2 Week 7 Day 2

7.2.1 ISLR 7.7 Generalized additive models (GAMs)

$$y = \beta_0 + f_1(x_1) + \cdots + f_p(x_p) + \epsilon, \tag{7.15}$$

where $f_1, \ldots, f_p$ can be different functions with different levels of smoothness.

- Easy to interpret.
- One can plan on the DFs spent on the predictors.
- Can be fit using backfitting (or LS when the functions are explicit).
- Flexible modeling of the effects of individual predictors. The functions can be global or local or piecewise or 2-dim or 3-dim.

Backfitting is an iterative algorithm for fitting additive models. For example, suppose our model is $y = \beta_0 + f_1(x_1) + f_2(x_2) + f_3(x_3) + \epsilon$. Given the current estimates $\hat{\beta}_0$, $\hat{f}_1$, and $\hat{f}_2$, we calculate partial residuals $r_i = y_i - \hat{\beta}_0 - \hat{f}_1(x_i) - \hat{f}_2(x_i)$ and then fit r_i to $f_3(x_i)$ to obtain a new estimate $\hat{f}_3$. We then repeat this process to estimate another component in the model. Repeat several cycles until convergence.

The R `gam` package has the function `gam()`. In `gam()`, a smoothing spline on a predictor `x` is specified through `s(x)`, and a loess fit is specified through `lo(x)`. Other basis generators such as `ns()`, `bs()`, and `poly()` can be used. Traditional model terms are also allowed, such as `x` (linear effect if `x` is quantitative, or categorical effect if `x` is qualitative), `I(x>10)`, and interaction term `x1*x2`, etc.

```
library(gam)
library(ISLR)

gam.m3 = gam(wage ~ s(year,4) + s(age,5) + education, data=Wage)
```

```
names(gam.m3)
summary(gam.m3)
gam.m3$coefficients
par(mfrow=c(1,3))
plot(gam.m3, se=T, col="blue", ylim=c(-30,40))  ## Figure 7.12
```

We can test for difference between nested models using `anova()`.

```
gam.m1 = gam(wage ~ s(age,5) + education, data=Wage)  ## no year
gam.m2 = gam(wage ~ year + s(age,5) + education, data=Wage)  ## linear in year
anova(gam.m1, gam.m2, gam.m3, test="F")
```

Compare `s()` with `ns()` and `lo()`

```
gam.m3b = gam(wage ~ ns(year,4) + ns(age,5) + education, data=Wage)
gam.m4 = gam(wage ~ s(year,df=4) + lo(age,span=0.7) + education, data=Wage)
par(mfrow=c(3,3))
plot(gam.m3, se=T, col="red", ylim=c(-30,40))  ## Figure 7.12
plot(gam.m3b, se=T, col="blue", ylim=c(-30,40))  ## Figure 7.11
plot(gam.m4, se=T, col="green", ylim=c(-30,40))
```

A 2-dimensional loess fit (to capture 2D interaction) can be specified:

```
gam.lo.i = gam(wage ~ lo(year, age, span=0.25) + education, data=Wage)
par(mfrow=c(1,2))
plot(gam.lo.i)  ## the plot for lo(year, age) is not informative

bb = preplot(gam.lo.i)  ## redraw it using rgl
str(bb)
library(rgl)
with(bb[[1]], plot3d(x[[1]], x[[2]], y))
```

Traditional interaction such as `x1*x2` looks okay. But interaction with splines seems wrong.

```
gam.m5 = gam(wage ~ s(age,5) + year*education, data=Wage)
anova(gam.m2, gam.m5, test="F")  ## df = 4, as expected

gam.m6 = gam(wage ~ s(age,5)*year + education, data=Wage)
anova(gam.m2, gam.m6, test="F")  ## df = 1, wrong
```

7.2.2 ISLR 8.1

A tree has a root, a few branches, terminal nodes (leaves), and internal nodes. See Figures 8.1–8.3 for illustrations.

To grow a tree, we use a *recursive binary splitting* algorithm (a greedy algorithm):

- Search all terminal nodes, and all possible dichotomizations of each node, $x < t$ and $x \geq t$, across all the predictors x.
- Identify the dichotomization that yields the most reduction in a splitting index.
  - The splitting index often is a measure of total within-subset impurity/heterogeneity with respect to y.
  - Impurity/heterogeneity can be quantified in various ways (see below).
- Every terminal node has a constant prediction value.
  - Continuous y: mean y for the node.
  - Categorical y: the majority class for the node.

Properties:

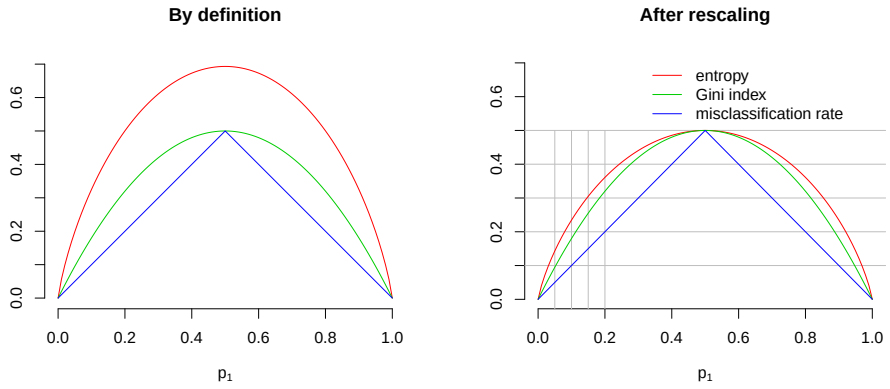
- They are "easy" to interpret.
- The terminal nodes form a partition of the whole space. Thus
  - A tree is a multi-dimensional step function. DF = #terminal nodes.

  - Trees are "local" but with "locality" determined by data (both X and y).
- Trees only rely on the order of the values of the predictors (i.e., scale-independent).
  - They are very fast to grow.
  - They can handle continuous, ordered categorical, and categorical variables well.
- Tree models are very flexible but tend to have a high variance.
- Trees are inefficient with respect to the number of parameters.
- Trees are very good base models for ensemble methods (random forests, boosted trees).
  - In random forests, individual trees are often grown to maximum possible depth.
  - In boosting, individual trees are often very shallow.

Stopping rules: to avoid unnecessary computation. `rpart.control()` and `tree.control()` show some examples.

Regression tree for continuous outcomes: RSS is often used as the measure of impurity. When splitting a node into two subsets, the total RSS for the node changes from $T = \sum_{k=1}^{2} \sum_j (y_{kj} - \bar{y})^2$ to $W = \sum_j (y_{1j} - \bar{y}_1)^2 + \sum_j (y_{2j} - \bar{y}_2)^2$. We seek to identify the split with the largest $B = T - W = [n_1(\bar{y}_1 - \bar{y})^2 + n_2(\bar{y}_2 - \bar{y})^2]$, the between-subset variance. (Notation: $\bar{y}$: average outcome for the node to be split; $\bar{y}_1$, $\bar{y}_2$: average outcomes of the two subsets; n_1, n_2: sizes of the two subsets.)

Classification tree for categorical outcomes: Measures of impurity include: (1) entropy/information, $D = -\sum_j p_j \log(p_j)$; (2) Gini index, $G = \sum_j p_j(1 - p_j) = 1 - \sum_j p_j^2$; and (3) misclassification rate, $E = 1 - \max(p_j)$. For example, when a node with size n and Gini index G is split into two subsets with sizes n_1 and n_2 and Gini indices G_1 and G_2, we evaluate nG and $n_1G_1 + n_2G_2$ to identify the split with the largest $nG - (n_1G_1 + n_2G_2)$. (1) and (2) are often used, as they are harder to push to zero than (3) relative to their maximum, as shown below for binomial distributions. The same is true for multinomial distributions. ISLR Figure 8.6 shows an example that improvement in one measure (entropy or Gini) may not guarantee improvement in another (misclassification rate).



The R `rpart` package is often used. To fit a tree, use `rpart(formula, data=, method=, control=)`.

- Use `method='class'` for classification trees and `method='anova'` for regression trees.
- `?rpart.control` shows control parameters (mostly stopping rules). By default, `cp=0.01` and `xval=10` (10-fold CV).
- For classification trees, additional controls can be set in `parms=` (defaults are: `prior`, prior probabilities are proportional to the data counts; `loss`, loss matrix for the 0–1 loss; `split`, 'gini'.) `split` is the criterion for choosing the "best" split (alternative choice is 'information'). `loss` is the criterion for evaluating the performance of a node.

```
library(rpart)
library(ISLR)
str(Wage)
table(Wage$health, useNA='always')

## Fit a tree to predict "health", a binary variable
tree1 = rpart(health ~ ., data=Wage, method='class')
tree1
```

```
plot(tree1); text(tree1)
plot(tree1, uniform=T); text(tree1, all=T, use.n=T)
post(tree1, file='')  ## post(tree1) saves to a ps file

tree1$frame  ## the result as a data frame
tree1$where  ## leaves the observations fall into
table(tree1$where)
tree1$frame$yval[tree1$where]  ## fitted values as numerical levels
Wage$health[tree1$frame$yval[tree1$where]]  ## fitted values in original values
```

Note that `tree1$where` gives the row numbers in `tree1$frame`, NOT the row names in `tree1$frame`.

The `summary()` function gives more information at every step.

```
summary(tree1)
summary(tree1, cp=.02)  ## This is to trim the summary, not to prune the tree.

myfun = function(k1, k2) {## compute gini and information for result checking
    kk = k1+k2; p1 = k1/kk; p2 = 1-p1
    list(gini = kk * (1-p1^2-p2^2), info = -kk * (p1*log(p1)+p2*log(p2)))
}

## improvement of Gini for the split of the root node
myfun(858, 2142)$gini - myfun(643, 1246)$gini - myfun(215, 896)$gini
## improvement of Gini for the split of node 2
myfun(643, 1246)$gini - myfun(215, 241)$gini - myfun(428, 1005)$gini

## Refit using entropy/information as the splitting index
tree2 = rpart(health ~ ., data=Wage, method='class', parms=list(split='information'))
tree2
summary(tree2)
myfun(858, 2142)$info - myfun(643, 1246)$info - myfun(215, 896)$info
```

Importance of a variable: Every split involves a splitting variable and an improvement in the splitting index. The importance of a variable can be defined as the total improvement attributable to the variable. In `rpart`, it is calculated with the additional contribution if the variable is a surrogate at some nodes. In the `Wage` dataset, `wage` and `logwage` are redundant for tree building because they have exactly the same order. They should have the same importance.

```
tree1$variable.importance
```

Surrogate variables: In `rpart`, surrogate variables serve two purposes: (1) They help classify observations in a node that have missing data for the splitting variable. (2) They are used in calculation of importance.

Identification of surrogates for a node: Once a splitting variable x and a split point t have been decided for the node, treat $x < t$ vs. $x \geq t$ as a binary outcome and consider all stumps (trees with only one split) using other predictors to predict this new outcome. Surrogates are the variables whose best stump has a lower misclassification rate than the "blind rule" of going with the majority without any input variable. To avoid artifacts, stumps with one of the subsets containing only one observation are ignored.

The observation that has missing value for the splitting variable is classified using the best surrogate variable. If it is missing, use the next best surrogate variable, etc. The last choice is the "blind rule".

Pruning: Cost complexity pruning: For all subtrees, consider

$$R(T) + \alpha|T|, \tag{8.4}$$

where $R(T)$ is the total impurity for subtree T, $|T|$ is the number of terminal nodes in tree T, and $\alpha > 0$ is the "complexity parameter" (the "cost" of adding a parameter to the model). The subtree that minimizes (8.4) is the "optimal" subtree for cost α. Note the similarity with the lasso.

In `rpart`, this is done with `prune()`, which takes a complexity parameter `cp` to determine where to prune a tree to. A tree created by `rpart()` has a `cptable` to help tree pruning. For example, we may prune to the `CP` value that has the smallest `xerror` in `cptable`. There are also functions `printcp()` and `plotcp()`.

```
tree1$cptable
plotcp(tree1)
prune(tree1, cp= tree1$cptable[which.min(tree1$cptable[,"xerror"]), "CP"])
```

What happens when `rpart()` is called: (a) A full tree T is grown (up to where the stopping rules allow); (b) a nested set of "optimal" subtrees, $|T_1| > |T_2| > \cdots > |T_m|$, are identified, where $T_1 = T$ and T_m is the tree with no splits; (c) the corresponding "typical" costs $0 = \beta_1 < \beta_2 < \cdots < \beta_k = \infty$ are calculated; (d) CV to identify the cost in $\{\beta_1, \ldots, \beta_k\}$ that has the best CV performance.

Below is an example of regression tree. For ISLR Figure 8.4, the authors say they used 9 features to build a tree. (I assume they used `AtBat`, `Hits`, `HmRun`, `Runs`, `RBI`, `Walks`, `Years`, `PutOuts`, and `Assists` to predict `Salary`.)

```
library(rpart)
library(ISLR)
names(Hitters)
Hitters2 = Hitters[,c(1:7,16,17,19)]

tree3 = rpart(Salary ~ ., data=Hitters2, method='anova')
plot(tree3); text(tree3)
post(tree3, file="")
plotcp(tree3)

par(mfrow=c(1,2))
rsq.rpart(tree3)

tree3$frame  ## the result as a data frame
table(tree3$where)
tree3$frame$yval[tree3$where]  ## fitted values as numerical levels

# prune the tree
tree3$cptable
prune(tree3, cp= tree3$cptable[which.min(tree3$cptable[,"xerror"]),"CP"])
```

The R `tree` package has a function `tree()`. It sometimes gives different results than `rpart()` because of different splitting criterion and stopping rules. The `rpart` package is recommended because it has richer features and it treats ordered factors correctly. For example, for an ordered factor with levels A < B < C < D, the only splits considered by `rpart()` are A-BCD, AB-CD, ABC-D. But `tree()` also considers other dichotomizations such as B-ACD.

```
library(tree)
tree4 = tree(Salary ~ ., data=Hitters2)
plot(tree4); text(tree4)
names(tree4)  ## has frame and where, similar to trees built with rpart()

tree4b = tree(Salary ~ ., Hitters2, control=tree.control(nobs=3000, mindev=0.005))
tree4b
plot(tree4b); text(tree4b)

cv.tree(tree4b)  ## default 10-fold CV
```

Example: Below I try to repeat the analysis shown in Figure 8.6. I cannot generate the same result because I do not know the various parameters the authors used. `tree()` gives the same splits at the first two levels as in Figure 8.6, `rpart()` starts to be different after level 1.

```
library(tree)
library(rpart)

Heart = read.csv("Heart.csv", row.names=1)  ## first column is row name, not a variable
names(Heart); dim(Heart)  ## 303, 14
table(Heart$AHD, useNA="ifany")  ## outcome distribution

(tree1 = rpart(AHD ~ ., data=Heart, method='class'))
plot(tree1); text(tree1)

(tree2 = tree(AHD ~ ., data=Heart))
plot(tree2); text(tree2)
```

Ordered categorical predictors: By default, categorical variables are coded as factors in R. Factors are NOT ordered, although their levels have an order (this is needed for efficient coding of factors). Ordered factors are treated as ordered by `rpart()` but still as categorical by `tree()`.

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

## Thallium stress test.  Increasing severity: normal, reversible defect, or fixed defect
levels(Heart$Thal)
## [1] "fixed"      "normal"     "reversable"
class(Heart$Thal)  ## "factor"
Heart$Thal

## This won't make it an ordered factor.  It just changes the internal coding.
##Heart$Thal = factor(Heart$Thal, c('normal', 'reversable', 'fixed'))

## This makes it an ordered factor.
Heart$Thal = factor(Heart$Thal, c('normal', 'reversable', 'fixed'), ordered=T)
levels(Heart$Thal)
class(Heart$Thal)  ## "ordered" "factor"
Heart$Thal

## Chest pain: The alphabetical order happens to be the correct order of increasing severity.
levels(Heart$ChestPain)
## [1] "asymptomatic" "nonanginal"   "nontypical"   "typical"
Heart$ChestPain = factor(Heart$ChestPain, levels(Heart$ChestPain), ordered=T)

(tree1 = rpart(AHD ~ ., data=Heart, method='class'))
plot(tree1); text(tree1)
```

Notes on other R packages for trees:

- `rpart.plot` gives nicer plots for trees built with `rpart`.
- `caret` (document) provides a unified interface (and a wrapper) for using `rpart` and many other R packages.
- `rpartScore` is for ordered categorical outcomes.
- `party` privides functions to build "conditional inference trees" (ctree) and to fit random forest models with conditional inference trees as base models.

7.2.3 Assignment

1. **Homework**: ISLR Chapter 8 Exercises 9
2. Reading for next lecture: ISLR 8.2; HOML Chapter 7
3. ISLR Chapter 8 R Labs