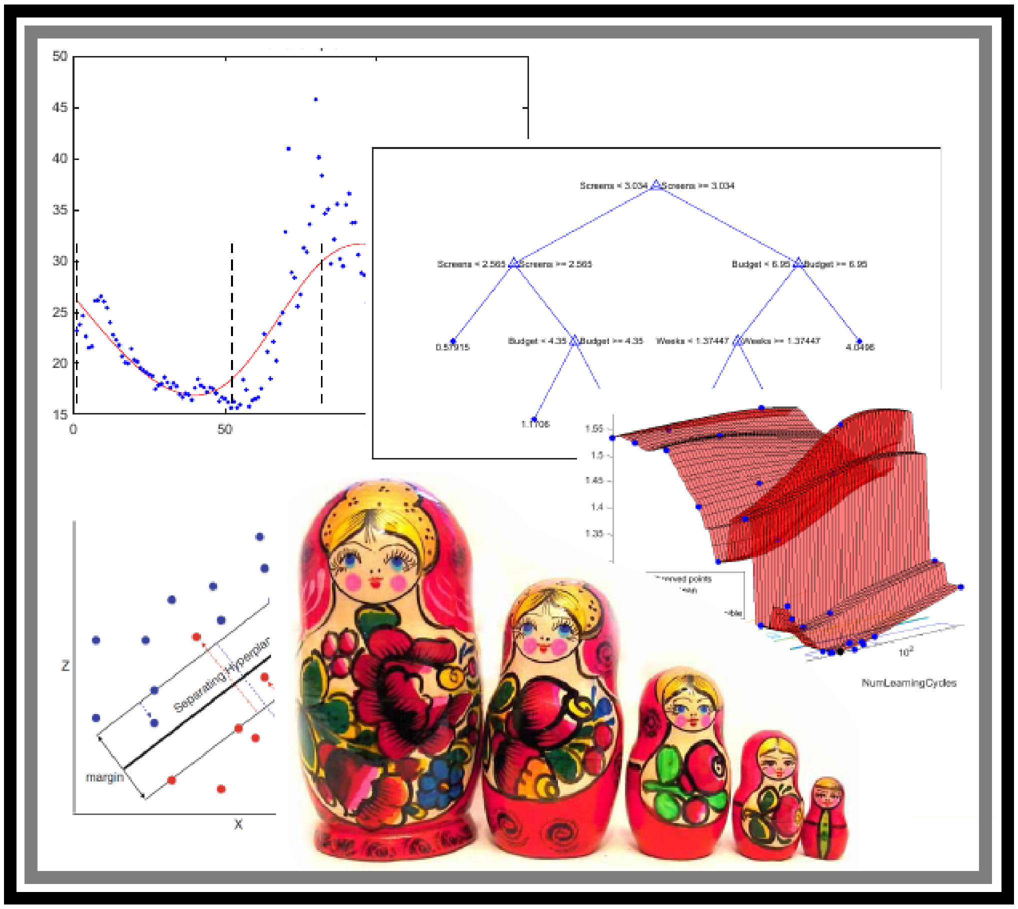
Machine Learning Lecture Notes

Tian Xie



Contents

1 Splines and Smoothing

1.1 Regression Splines

Given the following regression model:

$$\boldsymbol{y} = f(\boldsymbol{X}) + \boldsymbol{\epsilon},$$

where $\boldsymbol{y}$ is a quantitative response variable, $\boldsymbol{X}$ is a set of predictors, $\boldsymbol{\epsilon}$ is the error term, and $f(\cdot)$ is not yet specified. **Regression splines** are an algorithmic way to empirically arrive at a $f(\boldsymbol{X})$. There are various **basis functions**[1] that one can use.

1.1.1 Piecewise Linear Basis

For a **piecewise linear** basis,[2] the goal is to fit the data with a broken line (or hyperplane) such that at each break point the left-hand edge meets the right-hand edge.

Consider the following CBOE **VIX** data[3] from 2010-01-27 to 2010-08-17 demonstrated in Figure 1.1. The data is somewhat **cyclical** and a conventional linear regression will not fit the data well.

However, if we break the data into three parts following its pattern, we can fit each component quite well with its own linear regression. A key step is to decide where the break points on x will be. Such break points are often called **knots**.[4] To formalize the mathematics, we define two **indicators** that represent the break points:

$$\mathbb{I}_a = \begin{cases} 1 & \text{if } x > a \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \mathbb{I}_b = \begin{cases} 1 & \text{if } x > b \\ 0 & \text{otherwise} \end{cases}$$

[1]In mathematics, a **basis function** is an element of a particular basis for a function space. Every continuous function in the function space can be expressed as a linear combination of basis functions, just as every vector in a vector space can be represented as a linear combination of basis vectors. Consider a $n \times 2$ matrix $\boldsymbol{X}$, in which the first column contains 1s and the second column is the sole predictor for $\boldsymbol{y}$. Matrix $\boldsymbol{X}$ is called the **basis** of a bivariate regression model, which can be expanded in a linear form:

$$f(\boldsymbol{X}) = \sum_{m=1}^{M} \beta_m h_m(\boldsymbol{X}), \tag{1.1}$$

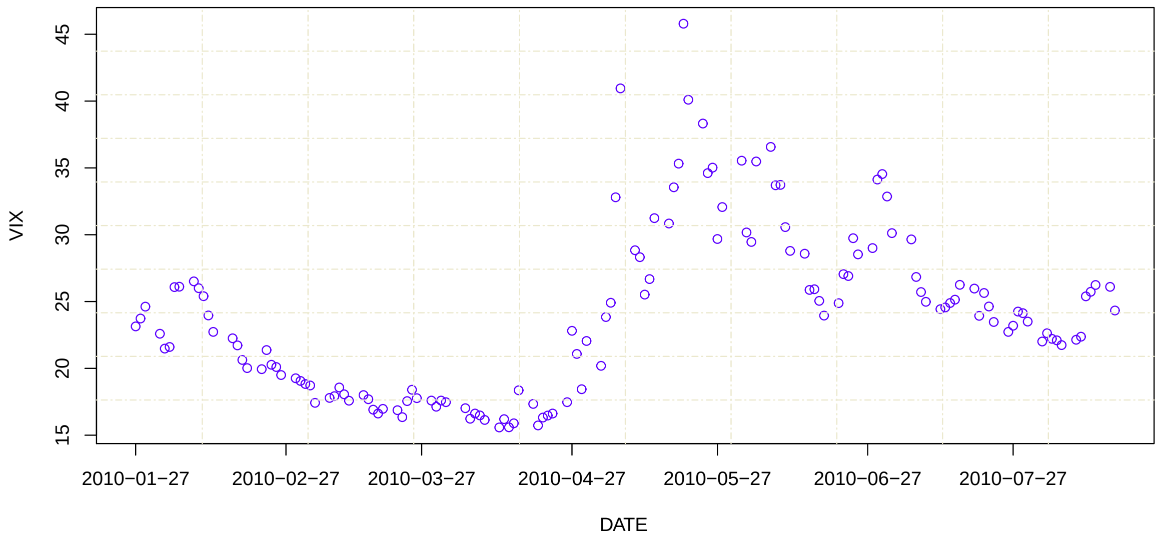
where $h_m(\cdot)$ is the m^{th} transformation of $\boldsymbol{X}$ and β_m is the weight assigned to the m^{th} transformation. Consequently, $f(\boldsymbol{X})$ is a linear combination of transformed values of $\boldsymbol{X}$. One common transformation employs **polynomial terms** such as $1, x, x^2, x^3$.

[2]When there is a single predictor, the fit is a set of straight line segments, connected end to end, henceforth, "piecewise linear".

[3]The CBOE Volatility Index, known by its ticker symbol VIX, is a popular measure of the stock market's expected volatility implied by the S&P 500 index options, calculated and published by the Chicago Board Options Exchange (CBOE). It is colloquially referred to as the fear index or the fear gauge.

[4]The first and the last observations are usually counted as boundary knots.

Figure 1.1: CBOE Volatility Index



Then, the mean function that allows for changes in the slope and the intercept is

$$f(x_i) = \beta_0 + \beta_1 x_i + \beta_2 (x_i - x_a)\mathbb{I}_a + \beta_3 (x_i - x_b)\mathbb{I}_b. \tag{1.2}$$

Equation (1.2) can be decomposed into three constituent line segments.

$$f(x_i) = \begin{cases} \beta_0 + \beta_1 x_i & \text{for } x < a \\ (\beta_0 - \beta_2 x_a) + (\beta_1 + \beta_2) x_i & \text{for } x \in [a, b] \\ (\beta_0 - \beta_2 x_a - \beta_3 x_b) + (\beta_1 + \beta_2 + \beta_3) x_i & \text{for } x > b \end{cases} \tag{1.3}$$

where each segment is a different linear regression model.[5] For a piecewise linear basis, one can simply compute mean functions such as in Equation (1.2) with OLS. With the estimated regression coefficients in hand, fitted values are easily constructed.

In the VIX exercise, we let the dates correspond to the numbers from 1 to 141. We set $a = 52$ and $b = 82$, which relate to 2010-04-12 and 2010-05-24. The associated R codes are presented below.

[5]Note that Equations (1.2) and (1.3) work when there are two knots. A more general form is needed if there are multiple knots. By increasing the number of knots, more complicated relationships can be approximated.

```
# 1. Linear Piecewise Fit for VIX2010
# 1.1 ready parameters
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
               header = TRUE, fileEncoding="UTF-8-BOM")
y = dat$VIX
x = 1:length(y)
xa <- ifelse(x>52,1,0)
xb <- ifelse(x>82,1,0)
x1 <- (x-52)*xa
x2 <- (x-82)*xb
# 1.2 estimation
out1 <- lm(y~x+x1+x2)
# 1.3 plot results
plot(x, y, col = "blue")
lines(x,out1$fitted.values, lty="dashed",col="red",lwd=3)
grid(10,10, lty = 6, col = "cornsilk2")
```

Results are plotted in Figure 1.2. The dots, the solid lines, and the dashed lines correspond to the VIX data, the linear regression fit for all segments, and the knots (including the first and the last observations). It is obvious that the end-to-end connections between line segments work well with processes that unfold overtime. In this example, a generated vector that represents **Date** is the sole predictor. Of course, there is nothing about linear regression splines requiring that date or time be a predictor. One can use other predictors, for example, stock market sentiment, oil price, or exchange rates.

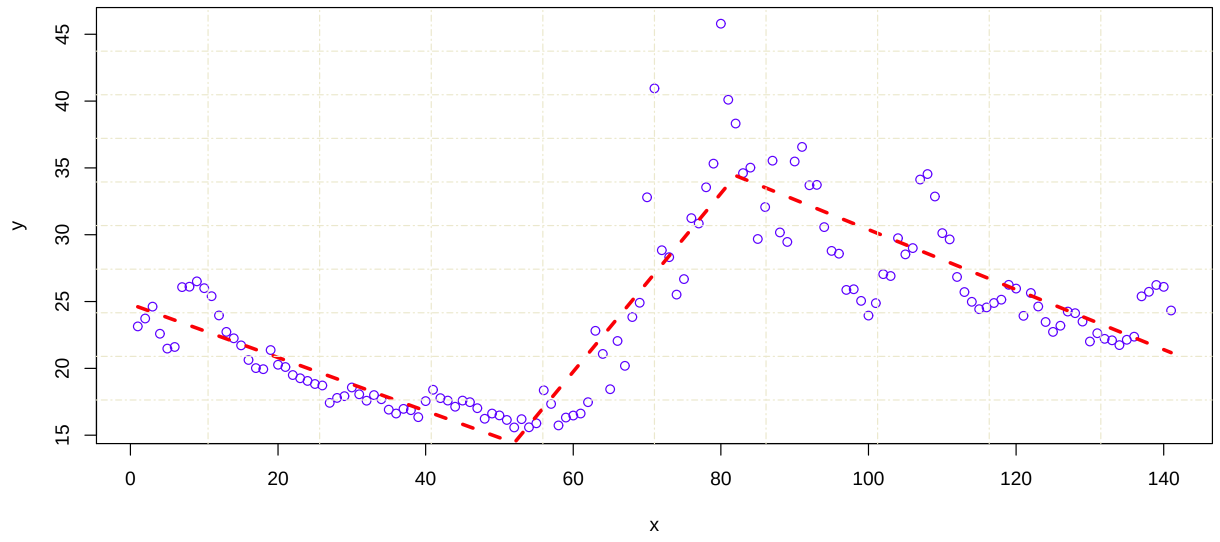
1.1.2 Piecewise Cubic Basis

Fitting line segments to data provides an example of **smoothing** a scatterplot. Using a piecewise linear basis has the great advantage of simplicity in concept and execution. However, there are also good reasons to believe that the underlying relationship is not well represented with a a set of straight line segments.

Greater continuity between line segments can be achieved by using polynomials in x for each segment. **Cubic functions** of x are a popular choice because they strike a nice balance between flexibility and complexity. The fit is called **piecewise cubic**. However, simply joining polynomial segments end to end is unlikely to result in a visually appealing fit where the polynomial segments meet. Far better visual continuity usually can be achieved by constraining the **first** and **second derivatives** on either side of each break point to be the **same**. This is usually called the **continuity constraints**.

We generalize the piecewise linear approach and impose those continuity requirements. Suppose there are K interior break points, usually called **interior knots**. These are located at $\xi_1 < ... < \xi_K$ with two boundary knots added at ξ_0 and ξ_{K+1}. We can use piecewise cubic polynomials in the following mean function exploiting linear basis

Figure 1.2: Piecewise Linear Estimation



expansions of $\boldsymbol{X}$:

$$f(x_i) = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \sum_{j=1}^{K} \theta_j (x_i - x_j)_+^3, \tag{1.4}$$

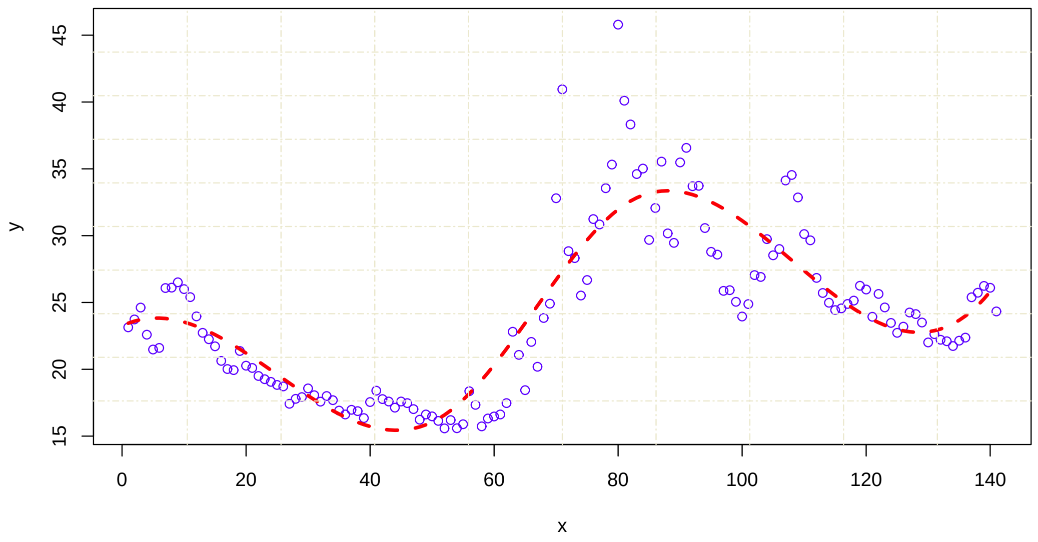
where $(\cdot)_+$ indicates the positive values from the expression inside the parentheses, and there are $K + 4$ parameters need to be computed. This leads to a conventional regression formulation with a matrix of predictor terms having $K + 4$ columns and N rows. Note that there is still only **a single** predictor, but now there are $K + 4$ **basis functions**.

We revisit the VIX exercise, only this time, we apply the piecewise cubic basis to fit the data. The associated R codes are presented below.

```
# 2. Cubic Piecewise Fit for VIX2010
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
               header = TRUE, fileEncoding="UTF-8-BOM")
library(splines)
y = dat$VIX
x = 1:length(y)
cubic <- bs(x,knots=c(52,82))
out2 <- lm(y~cubic)
plot(x, y, col = "blue")
lines(x,out2$fitted.values,lty="dashed",col="red",lwd=3)
grid(10,10, lty = 6, col = "cornsilk2")
# 2.1 compare R-square
summary(out1)$r.squared
summary(out2)$r.squared
```

Results are plotted in Figure 1.3. The dots, the solid line, and the dashed lines correspond to the VIX data, the piecewise cubic fit, and the interior knots (including the boundary knots). Figure 1.3 reveals a good eyeball fit. Comparing to Figure 1.2, we notice that the piecewise cubic basis works particularly well on the third segment, as it captures the uptrend of VIX at the end.

Figure 1.3: Piecewise Cubic Estimation



However, due to the two **continuity constraints**, for the first two segments, there is no obvious advantage of using the cubic basis rather than the simple linear basis. To have a general evaluation of the overall performance of the two methods, we can compare their estimated R^2. The results are 0.7815 and 0.7641, respectively. The piecewise linear basis has a better overall fit.

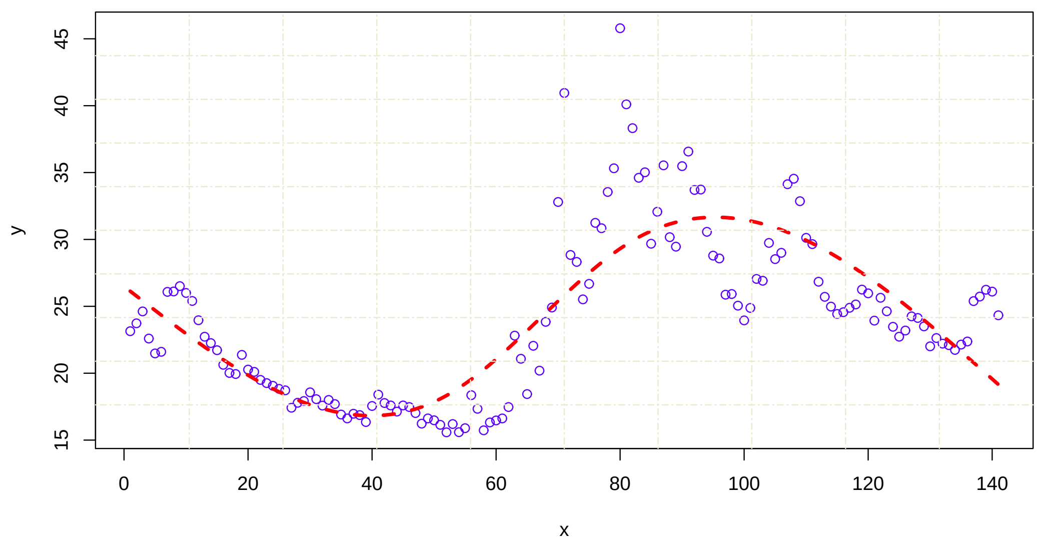
A key implication is that it is very **difficult** to arrive at a model that is demonstrably the best. Moreover, it can be **risky** to rely on statistical summaries to chose the most instructive response surface approximation. **Subject-matter knowledge**, **potential applications** and **good judgments** need to play an **important** role.

1.1.3 Natural Cubic Splines

Fitted values values for piecewise cubic polynomials near the boundaries of x can be **unstable** as there are no continuity constraints and where the data can be sparse. One common solution is to impose **linearity** on the fitted values beyond the boundaries of x as if the data beyond the boundaries were available. The result is a **natural** cubic spline under this constraint. R codes and estimation results are presented below.

```
# 3. Natural Cubic Piecewise Fit for VIX2010
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
               header = TRUE, fileEncoding="UTF-8-BOM")
library(splines)
y = dat$VIX
x = 1:length(y)
Ncubic <- ns(x,knots=c(52,82))
out3 <- lm(y~Ncubic)
plot(x, y, col = "blue")
lines(x,out3$fitted.values,lty="dashed",col="red",lwd=3)
grid(10,10, lty = 6, col = "cornsilk2")
# 3.1 compare R-square
summary(out1)$r.squared
summary(out2)$r.squared
summary(out3)$r.squared
```

Figure 1.4: Natural Cubic Estimation



1.2 Penalized Smoothing

The placement of knots, the number of knots, and the degree of the polynomial can be seen as **tuning parameters**, which are subject to **manipulation** by a data analyst. The tuning process can be very complicated, since there are at least three of them that must be tuned simultaneously. Moreover, there is little or **no formal theory** to justify the tuning.

A useful alternative is to alter the fitting process itself so that the tuning is accomplished automatically, guided by clear statistical reasoning. One popular approach is to combine a **mathematical penalty**[6] with the loss function to be optimized. This leads to a very popular approach called **penalized regression**.

Consider a conventional regression analysis with an indicator variable as the sole regressor. As the regression coefficient increases in absolute value, the resulting step function will have a step of increasing size. The difference between the conditional mean of Y when the indicator is 0 compared to the conditional means of Y when the indicator is 1 is larger. The **larger** the regression coefficient the **rougher** the fitted values.

Strategies that are designed to control the magnitude of the coefficients are called **shrinkage** or **regularization**. Two popular proposals have been offered for how to control the complexity of the fitted values:

(i) constrain the sum of the absolute values of the regression coefficients to be less than some constant C (sometimes called an L_1**-penalty**); and

[6]The penalty imposes greater losses as a mean function becomes more complicated. For greater complexity to be accepted, the fit must be improved by an amount that is larger than the penalty. The greater complexity has to be worth it.

(ii) constrain the sum of the squared regression coefficients to be less than some constant C (sometimes called an L_2-**penalty**).

1.2.1 Ridge Regression

Suppose that for a conventional fixed $\boldsymbol{X}$ regression, one adopts the constraint that the sum of the p squared regression coefficients is less than C. This constraint leads directly to **ridge regression**.[7] The task is to obtain values for the regression coefficients so that

$$\hat{\boldsymbol{\beta}} = \min_{\boldsymbol{\beta}} \left[\sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right], \tag{1.5}$$

where $\boldsymbol{\beta} = [\beta_1, ..., \beta_p]^\top$ does not include β_0. In Equation (1.5), the usual expression for SSR has a new component - the sum of the squared regression coefficients multiplied by a constant λ. This is a L_2 penalty. Note that λ is a tuning parameter that determines how much weight is given to the penalty.

It follows that the ridge regression estimator is

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda \boldsymbol{I})^{-1} \boldsymbol{X}^\top \boldsymbol{y}, \tag{1.6}$$

where $\boldsymbol{I}$ is a $p \times p$ identity matrix. Note that the column of 1s for the **intercept is dropped** from $\boldsymbol{X}$ and β_0 is estimated separately.[8] In Equation (1.6), the value of λ is added to the main diagonal of the cross-product matrix $\boldsymbol{X}^\top \boldsymbol{X}$, which determines how much the estimated regression coefficients are **shrunk toward zero**. With non-zero λ, the shrinkage becomes a new source of **bias**. However, while biased, the reduced variance of ridge estimates often result in a **smaller** mean square error when compared to least-squares estimates.

We revisit the VIX exercise. We use the one-period **lag of VIX** as the sole predictor. We consider a sequence of $\lambda = e^L$ with $L = 0.1, 0.2, ..., 10$. Using the `glmnet` function,[9] we estimate the coefficient associated with lag of VIX. Note that the `glmnet` function is capable of working with multiple values of λ. We plot the path of coefficients by log of λ. The R codes are presented below.

[7] In machine learning literature, it is sometimes called **weight decay**.

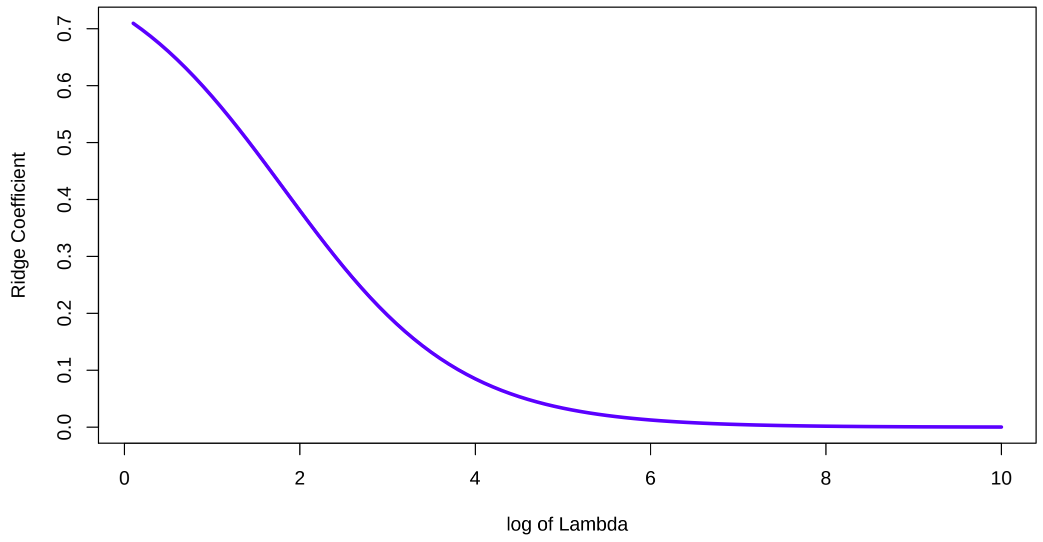
[8] By default, $\boldsymbol{\beta}$ is computed after **centering and scaling** the predictors to have mean 0 and standard deviation 1. The model does not include a constant term, and $\boldsymbol{X}$ should not contain a column of 1s.

[9] Note that you may need to install the `glmnet` package first.

```
# 4. ridge regression
# 4.1 ready parameters
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
y = dat$VIX
int = rep(1,length(y))
ly = c(0,y[1:length(y)-1]) # lag of VIX
x = matrix(c(int,ly), nrow=length(y))
L = seq(0.1, 10, by=0.1)
# 4.2 estimation (may need to install glmnet package)
# install.packages("glmnet")
library(glmnet)
outr <- glmnet(x, y, alpha = 0, lambda = exp(L))
# 4.3 plot results
plot(log(outr$lambda),outr$beta[2,],col="blue",type="l",lwd=3,
     xlab="log of Lambda", ylab="Ridge Coefficient")
```

Results are presented in Figure 1.5. The coefficient decreases as we increase the value of λ. This makes sense since the more weight we put on the penalty term, the "flatter" the ridge coefficient becomes.

Figure 1.5: Path of Coefficients by Log of λ



1.2.2 The Least Absolute Shrinkage and Selection Operator (LASSO)

Suppose that one proceeds as in ridge regression but now adopts the constraint that the sum of the **absolute values** of the regression coefficients (L_1-penalty) is less than some constant. This leads to a regression procedure known as the LASSO (Tibshirani, 1996) whose estimated regression coefficients are defined by

$$\hat{\beta} = \min_{\beta} \left[\sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right]. \tag{1.7}$$

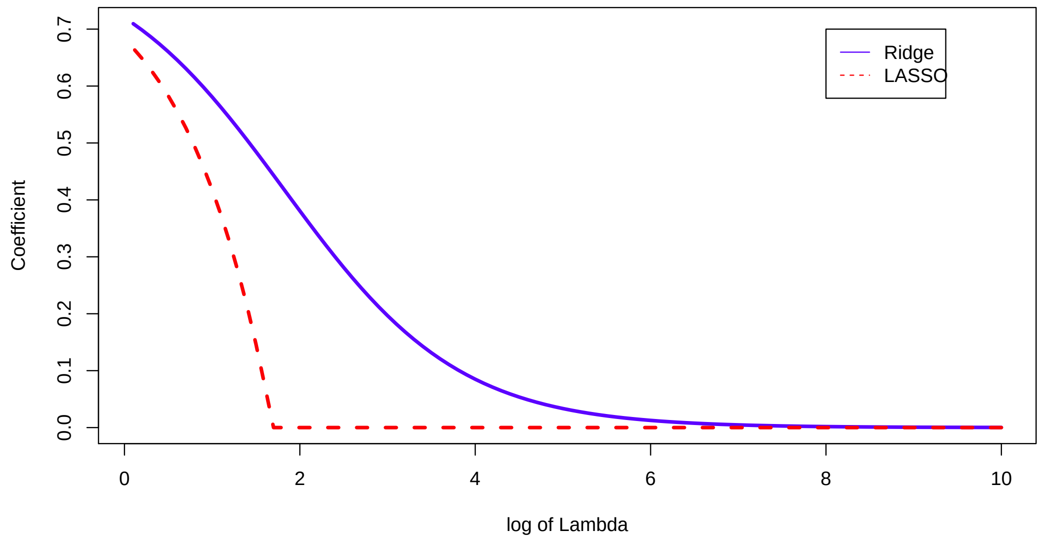
Unlike the ridge penalty, the LASSO penalty leads to a **nonlinear estimator**, and a **quadratic programming** solution is needed. As before, the value of λ is a tuning parameter, a λ of zero yields the usual least squares results, and as the value of λ increases, the regression coefficients are shrunk toward zero.

Again, we revisit the VIX exercise using the one-period lag of VIX as the sole predictor. We consider a sequence of $\lambda = e^L$ with $L = 0.1, 0.2, ..., 10$. Using the built-in command `lasso`, we estimate the coefficient associated with lag of VIX. To illustrate the differences between Ridge and LASSO estimators, we replicate the exercise in Figure 1.5 and compare the estimated coefficients by Ridge with LASSO.The R codes for the VIX exercise using LASSO are presented below.

```
# 5. lasso regression
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
y = dat$VIX
int = rep(1,length(y))
ly = c(0,y[1:length(y)-1]) # lag of VIX
x = matrix(c(int,ly), nrow=length(y))
L = seq(0.1, 10, by=0.1)
library(glmnet)
outl <- glmnet(x, y, alpha = 1, lambda = exp(L))
# 4.3 plot results
plot(log(outr$lambda),outr$beta[2,],col="blue",type="l",lwd=3,
     xlab="log of Lambda", ylab="Coefficient")
lines(log(outl$lambda),outl$beta[2,],lty="dashed",col="red",lwd=3)
legend(8, 0.7, legend=c("Ridge", "LASSO"),
       col=c("blue", "red"), lty=1:2)
```

Estimated results are plotted in Figure 1.6, in which the solid line and the dashed line correspond to the path of coefficients for Ridge and LASSO estimator, respectively. Under the same penalty weight (λ), LASSO shrinks the coefficient towards 0 at a faster rate than Ridge.

Figure 1.6: Path of Coefficients for LASSO and Ridge



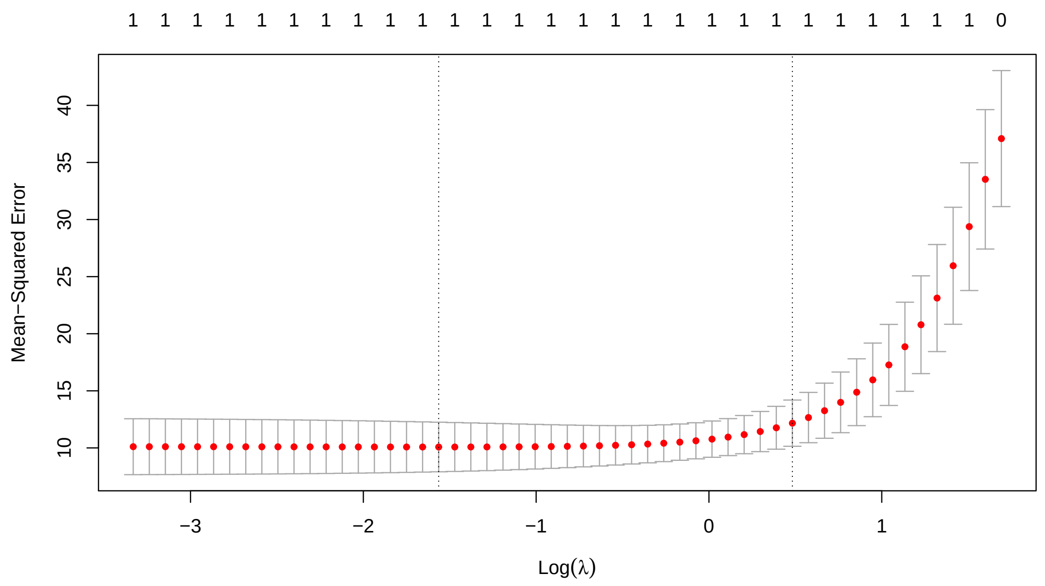
1.2.3 Tuning λ by Cross-validation

There are significant complication if the value of λ is determined through data snooping. Ideally, if there are training data and evaluation data, one solution is trial and error. In ***k*-fold cross-validation**, the original sample is randomly partitioned into k equal sized subsamples. Different values of λ are tried with one subset (training data) until there is a satisfactory fit in the other subset (evaluation data) by some measure such as **mean squared error**.

For the VIX exercise, we apply 5-fold cross-validation on the LASSO λ using the `cv.glmnet`. R codes are presented below. The cross-validated MSEs of LASSO fit are plotted against values of λ in Figure 1.7. The optimal λ that minimizes MSE is reported by `$lambda.min`. Try the codes a few times, are the optimal λs the same and why?

```
# 6. lasso regression - Tuning lambda
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
y = dat$VIX
int = rep(1,length(y))
ly = c(0,y[1:length(y)-1]) # lag of VIX
x = matrix(c(int,ly), nrow=length(y))
L = seq(0.1, 10, by=0.1)
# 6.1 lasso by tuning
library(glmnet)
outlcv <- cv.glmnet(x, y, alpha = 1, nfolds = 5)
# 6.2 plot results
plot(outlcv)
outlcv$lambda.min
```

Figure 1.7: LASSO with 5-fold Cross-validation



1.3 Smoothing Splines

For the spline-based procedures considered earlier, the number and location of knots had to be determined a **priori** or by some measure of fit. We are now proceeding in the same spirit as Ridge regression and the LASSO, but we are allowing for nonlinear associations between $\boldsymbol{X}$ and $\boldsymbol{y}$ and introducing a different kind penalty function.

For a single predictor and a quantitative response variable, there is a function $f(\boldsymbol{X})$ with two derivatives over its entire surface. The goal is to minimize a penalized error sum of squares of the form[10]

$$C(f, \lambda) = (1 - \lambda) \sum_{i=1}^{n} [y_i - f(\boldsymbol{x}_i)]^2 + \lambda \int [f''(t)]^2 \, dt, \tag{1.8}$$

where $\lambda \in [0, 1]$ is, as before, a tuning parameter and the integral quantifies the roughness penalty. As λ decreases toward 1, the fitted values becomes flatter. At the other extreme, as λ increases to 0, the fitted values approach an interpolation of the values of the response variable. Equation (1.8) can be minimized with respect to the $f(\boldsymbol{X})$, given a value for λ.

1.3.1 Cubic Smoothing Spline

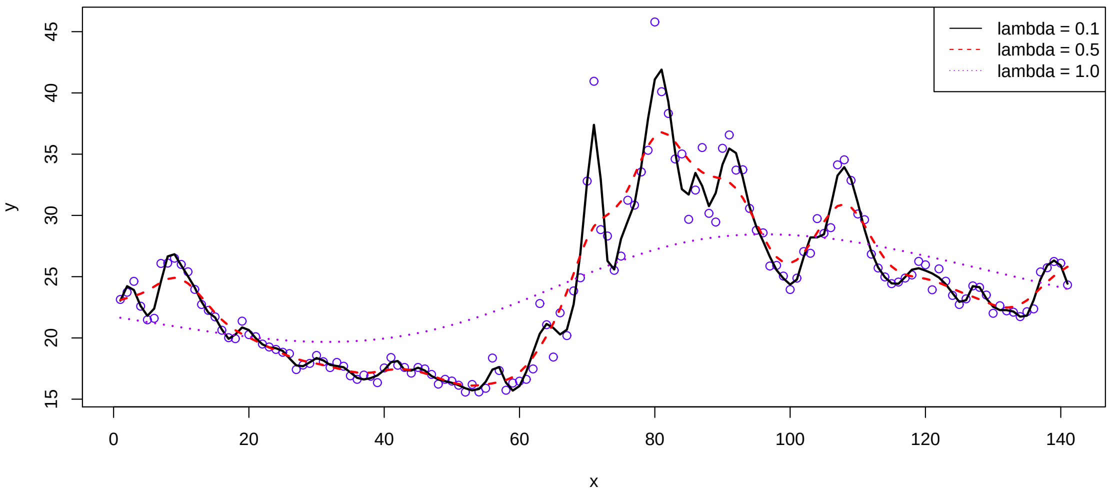
We start with simple **linear basis** function. For instance, based on **cubic splines**, we can smooth the VIX time series using the gam command. Again, you may need to install the gam package first. We try three values of $\lambda = 0.1, 0.5$, and 1. Estimated smoothing results are plotted in Figure 1.8, in which the dots, solid line, dashed line, and dotted line correspond to the VIX data, $\lambda = 0.1$, 0.5, and 1, respectively.

R codes are presented below.

```
# 7. cubic smoothing splines
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
               header = TRUE, fileEncoding="UTF-8-BOM")
# install.packages("gam")
library(gam)
y = dat$VIX
x = 1:length(y)
f = data.frame(y,x);
# 7.1 estimation
out1 <- gam(y~s(x, spar=0.1),data=f)
out2 <- gam(y~s(x, spar=0.5),data=f)
out3 <- gam(y~s(x, spar=1),data=f)
plot(x, y, col = "blue")
lines(x,out1$fitted.values,lty="dashed",col="black",lwd=2)
lines(x,out2$fitted.values,lty="solid",col="red",lwd=2)
lines(x,out3$fitted.values,lty="dotted",col="purple",lwd=2)
legend('topright', legend=c("lambda = 0.1","lambda = 0.5",
         "lambda = 1.0"),col=c("black", "red","purple"), lty=1:3)
```

[10]Note that our criterion is slightly different from the textbook.

Figure 1.8: Smoothing Results with Various Penalty Weights



1.3.2 Local Regression Smoothing

Local regression is a generalization of moving average and polynomial regression. Its most common methods initially developed for scatterplot smoothing, is **LOWESS** (locally weighted scatterplot smoothing). It fits simple models to localized subsets of the data to build up a function that describes the deterministic part of the variation in the data, point by point.

We consider the simple case of one predictor x. For any given value of x_0, a polynomial regression is constructed only from observations with x-values that are **nearest neighbors** of x_0. Among these, observations with x-values **closer** to x_0 are **weighted more heavily** for LOWESS. Then, $\hat{y}_0$ is computed from the fitted regression and used as the smoothed value of the response y at x_0. The precise weight given to each observation depends on the **weighting function** employed.[11] The process is repeated for all other values of x.

Originally, the authors of the above procedure called the method LOESS since the original method does not assign different weights to different observations. This becomes particularly confusing in R coding, since both functions (`lowess` and `loess`) exist and the latter is a newer, more generalized version of LOWESS. Specifically, both functions use the tricube weighting functions (yes, `loess` also weights the observations from near to far) and `loess` allows you to incorporate more fitting functions.

One important tuning parameter for the local regression smoothing is the **span param-**

[11]The **normal distribution** is one option. The **tricube** is another option. Differences between x_0 and each value of x in the window are divided by the length of the window along x. This standardizes the differences. Then the differences are transformed as $(1 - |z|^3)^3$, where z is the standardized difference. Values of x outside the window are given weights of 0. There seems to be no formal justification for any particular weighting function.

eter, f, a number between 0 and 1 that decides the proportion of observations included for each x_0. The larger the proportion of observations included, the smoother are the fitted values. The span plays the same role as the number of knots in regression splines or λ in smoothing splines.

Again, we revisit the VIX exercise and smooth VIX using local regression. We use the simpler function `lowess` and try various span parameters $f = 0.1, 0.5$, and 1. R codes are presented below.

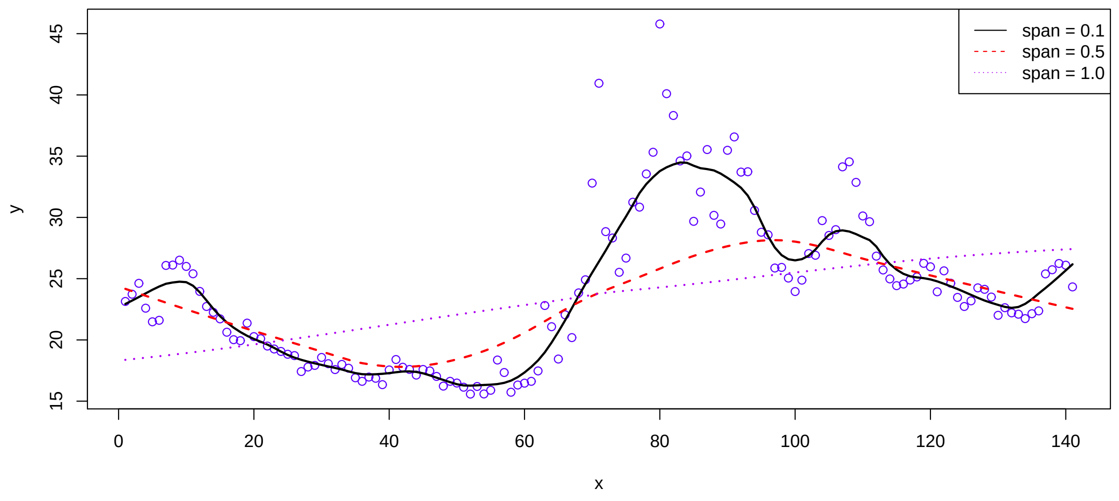
```
# 8. local regression smoothing - LOWESS
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# install.packages("gam")
library(gam)
y = dat$VIX
x = 1:length(y)
# 8.1 estimation
out1 <- lowess(x,y, f = 0.1)
out2 <- lowess(x,y, f = 0.5)
out3 <- lowess(x,y, f = 1)
plot(x, y, col = "blue")
lines(out1$x,out1$y,lty="solid",col="black",lwd=2)
lines(out2$x,out2$y,lty="dashed",col="red",lwd=2)
lines(out3$x,out3$y,lty="dotted",col="purple",lwd=2)
legend('topright', legend=c("span = 0.1","span = 0.5",
          "span = 1.0"),col=c("black", "red","purple"), lty=1:3)
```

Estimated smoothing results are plotted in Figure 1.9, in which the dots, dashed line, solid line, and dotted line correspond to the VIX data, $f = 0.1$, 0.5, and 1, respectively. The smoothed value becomes flatter when f gets larger.

Function `lowess` can also iterate the results to provide more robust estimation.[12] We repeat the exercise in Figure 1.9 and focus on $f = 0.1$. We compare LOWESS with no iteration with LOWESS with 1000 iterations. R codes are presented below and results are plotted in Figure 1.10, in which solid line and dashed line represent no iteration and 1000 iteration results respectively. It is obvious that the curve is smoother after 1000 iteration.

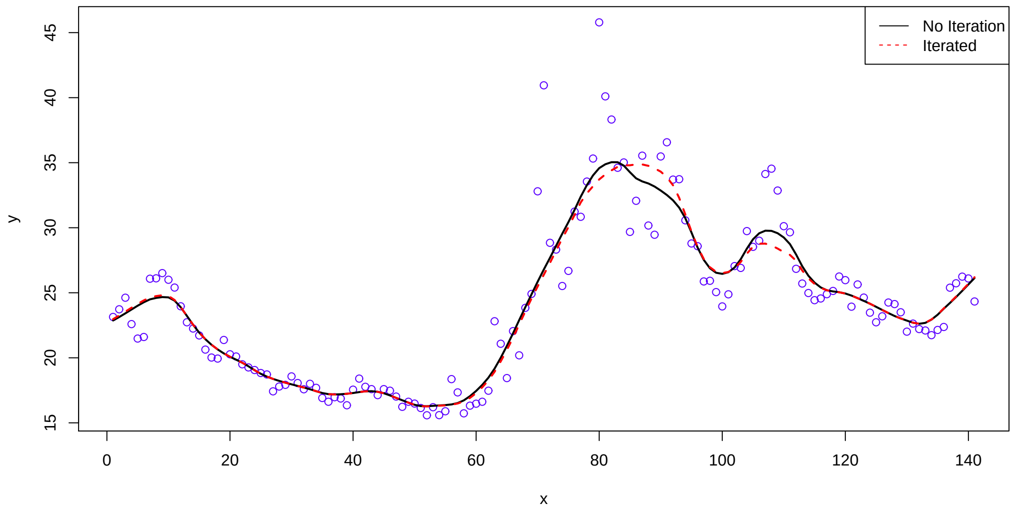
[12]In fact, the lowess function iterates 3 times by default. See `?lowess` for details.

Figure 1.9: LOWESS Smoothing Results with Different Span Parameters



```
# 9. local regression smoothing - iterated LOWESS
dat=read.csv("D:/Dropbox/R/teaching/machine learning/vix2010.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# install.packages("gam")
library(gam)
y = dat$VIX
x = 1:length(y)
# 9.1 estimation
out1 <- lowess(x,y, f = 0.1, iter = 1)
out2 <- lowess(x,y, f = 0.1, iter = 1000)
plot(x, y, col = "blue")
lines(out1$x,out1$y,lty="solid",col="black",lwd=2)
lines(out2$x,out2$y,lty="dashed",col="red",lwd=2)
legend('topright', legend=c("No Iteration","Iterated"),
       col=c("black", "red"), lty=1:2)
```

Figure 1.10: LOWESS Smoothing with and without Iteration



2 Classification and Regression Trees

Conventional classification and regression trees (**CART**),[13] introduced by Breiman, Friedman, and Stone (1984), is a form of stagewise regression with predictors that are indicator variables. CART output is typically displayed in a **tree-like** structure, which accounts for how the technique is named.

CART uses a fast divide and conquer **greedy algorithm**[14] that recursively partitions the data into smaller subsets. As the name indicates, CART incorporates two (separate) strategy: **classification tree** and **regression tree (RT)**. Classification trees give responses that are **nominal**, such as `true` or `false`. Regression trees give **numeric** responses.

Procedures that assign observations to classes are sometimes called **classifiers**. When CART is used with categorical response variables, it is an example of a classifier. One grows a classification tree. When applied to real number predicted outcomes, CART differs sharply from econometric strategies by **not linearizing** the relationship in the DGP. CART aims to estimate $y = f(x)$ while trying to avoid overfitting. Further, smoothness conditions are **not** required in contrast to many nonparametric approaches in econometrics. Due to the numeric natural of economic data, **we pay more attention to the RT strategy**.

Work on tree-based regression models traces back to Morgan and Sonquist (1963) and within machine learning, most research efforts concentrate on classification (or decision) trees (Hunt, Martin, and Stone, 1966, Quinlan, 1986) and work on regression trees started

[13] In computer science literature, CART is also called the **decision tree**.

[14] CART's algorithm is called "greedy" because it searches for the best outcome without looking back to past splits or forward to future splits. The algorithm lives only in the present.

with RETIS (Karalic and Cestnik, 1991) and M5 (Quinlan, 1992).

2.1 The Basic Ideas

Suppose one has a single quantitative response variable and several predictors. There is interest in $\hat{Y}|X$. The immediate task is to find the single best binary predictor from among a set of predictors, all of which may be numerical. To do this, two kinds of searches are undertaken.

(i) **First**, for each predictor, all possible binary splits of the predictor values are considered and the **best split** is determined. For quantitative response variables, the baseline is the response variable **sum of squares**.[15]

(ii) With the best split of each predictor determined, the best split overall is determined as the **second** step. That is, the best split for each predictor is compared by the reduction in the sum of squares. The predictor with the largest reduction wins the competition. It is the predictor that when split, leads to the greatest reduction in the sum of squares.

With the two-step search completed, the winning split is used to subset the data. There are now two partitions of the original data, defined by best split within and between the predictors. Next, the same two-step procedure is applied to each partition separately. The **recursive partition** process can continue until there is no meaningful reduction in the sum of squares of the response variable. Then, the results are conventionally displayed as an **inverted tree**: roots at the top and canopy at the bottom.

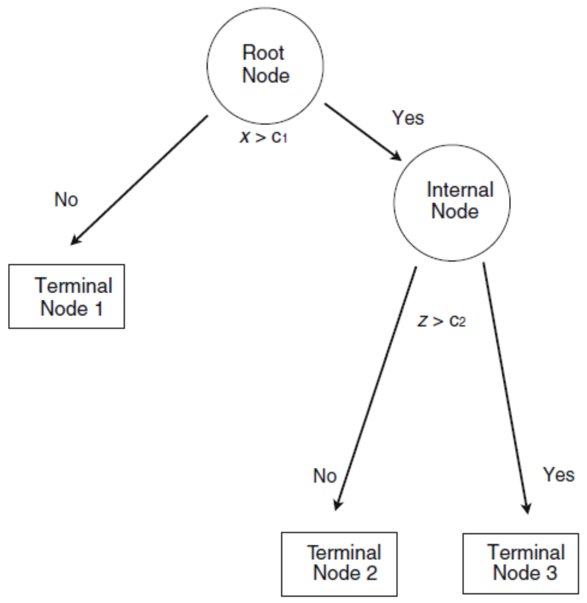
Figure 2.1 is a simple, schematic illustration of an inverted tree. The full dataset is contained in the **root node**. The data are then broken into two mutually exclusive pieces. Cases with $X > C1$ go to the right, and cases with $X \leq C1$ go to the left. The latter are then in **terminal node** 1, which is not subject to any more partitioning. The former are in an **internal node** that can be usefully subdivided further. Observations with $Z > C2$ go to the right and into **terminal node** 3. Observations with $Z \leq C2$ go to the left and into terminal node 2. Further subsetting is not productive.

2.2 Splitting A Node

The first problem that the CART algorithm needs to solve is how to split each node using information contained in the set of predictors. For a quantitative predictor with m distinct values, there are $m-1$ splits that **maintain the existing ordering of values**. So, $m-1$ splits on that variable need to be evaluated. Moreover, there are often algorithmic shortcuts that improve computational efficiency. If order is **not** maintained, which is

[15]The response variable sum of squares is computed separately within each partition and added together. That sum will be equal to or less than the sum of squares for the response variable before the partitioning. The **best split** for each predictor is defined as the split that reduces the sum of squares the most.

Figure 2.1: A Simple CART Tree Structure



true for **categorical** predictors, one needs to search through $2^{k-1} - 1$ possible splits for k categories. The computational burdens can be quite heavy, even with some shortcuts.

A node τ containing n_τ observations with **mean outcome** $\bar{y}(\tau)$ can only by split by one selected variable into two leaves, denoted as τ_L and τ_R. The split is made at the variable where

$$\Delta = \text{SSR}(\tau) - \text{SSR}(\tau_L) - \text{SSR}(\tau_R),$$

reaches its global maximum;[16] where the within-node sum of squares is

$$\text{SSR}(\tau) = \sum_{i}^{n_\tau}(y_i - \bar{y}_\tau)^2.$$

This splitting process continues at each new node until the $\bar{y}(\tau)$ at nodes can no longer be split since it will not add any additional value to the prediction.

To better illustrate how the nodes are split, we consider the following **movie forecasting** exercise. This is a subset of the data originally used in the movie forecasting exercises in Lehrer and Xie (2017). The main response is actual **opening weekend box office** for 94 movies. The **6 non-constant predictors** include: **3 genre** dummy variables stand for Animation, Crime, Romance, and **3 core parameters** that include the movie budget excluding advertising and both the pre-determined number of weeks and screens the movie studio forecasted that the specific movie will be in theatres measured approximately six weeks

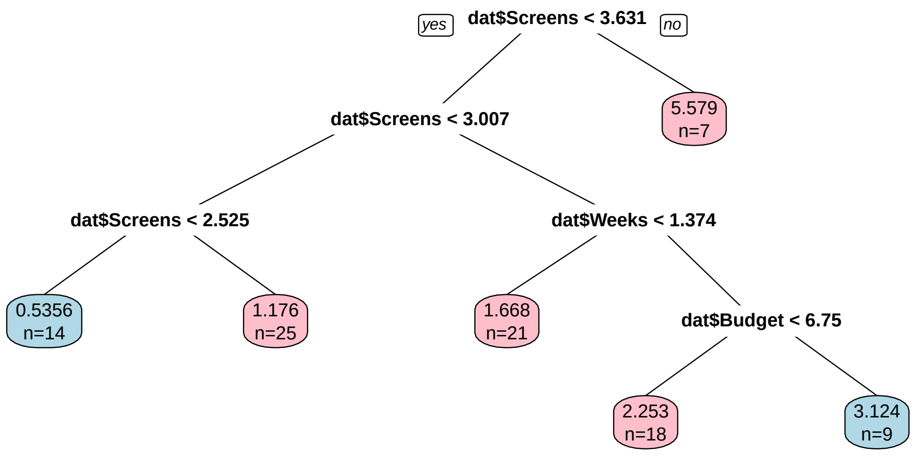
[16]Implicitly it is assumed that there are no unobservables relevant to the estimation. That said, the standard methodology to induce regression trees is based on the minimization of the squared error.

prior to opening.

We load the `moviedata.csv` and generate its summary of statistics. We use the `rpart` package to construct the tree and plot the results using the `rpart.plot` command.[17] R codes are presented below. The constructed tree is shown in Figure 2.2.

```
# 1. simple regression tree
# 1.1 load data and describe summary
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
               header = TRUE, fileEncoding="UTF-8-BOM")
summary(dat)
# 1.2 RT estimation
# install.packages("rpart")
# install.packages("rpart.plot")
library(rpart)
library(rpart.plot)
out<-rpart(OpenBox~Constant+Animation+Crime+Romance+Budget
           +Weeks+Screens,data=dat, method="anova")
# 1.3 plot tree
prp(out,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
```

Figure 2.2: A Simple Regression Tree Structure



Note that this tree is small as it is created using **default setting**. The node splits rely heavily on the three continuous variables: budget, weeks, and screens. This is not a

[17]You may need to install these packages first, depending on your R version.

surprise. If we run an OLS estimation, we will find out that the three continuous variables are important predictors that determine the open box office.

```
# 2. ols estimation
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
out2 <- lm(OpenBox~Constant+Animation+Crime+Romance+Budget
           +Weeks+Screens,data=dat)
summary(out2)
```

Take a note that the estimated centered R^2 by OLS is **0.4135**.

2.3 Predicting Using Trees

You may notice that in Figure 2.2, there is **one specific number** under each terminal node, or more vividly, **tree leaf**. These numbers are the the simple average of the response within each leaf l: $\bar{y}_{i \in l}$. There are two major reasons for such execution. First, for the sake of **computational efficiency**, and second, the belief that no **within group heterogeneity**.

CART is essentially a classification device. Ideally, after partitioning the dataset into numerous final leaf nodes, there should be no within group heterogeneity and leave only the **cross-group heterogeneity**, identified by the leaves. In this case, responses within each leaf should be no different from one another and they can be represented perfectly by their simple average.

Once the tree is constructed with its tree leaves estimated, it is ready for predicting exercises. Assume that we have an Romance movie with values on budget, weeks, screens equal to 4, 1, 3, respectively. We construct the inputs as $\boldsymbol{X}_p = [1, 0, 0, 1, 5, 4, 3.5]$. To make prediction, we start from the root node and apply the input $\boldsymbol{X}_p$ through the tree structure following the splitting rule displayed in Figure 2.2 for each node:

Step 1. The split rule in the root node is: go left if Screen < 3.631, go right otherwise. Given Screen = 3.5 in $\boldsymbol{X}_p$. **We go left**.

Step 2. Similarly, **we go right** in the next internal node.

Step 3. The next split rule in the root node is: left if Weeks < 1.374, right otherwise. Given Weeks = 4 in $\boldsymbol{X}_p$. **We go right**.

Step 4. Now, we are one step away from the internal node. The final split rule is: left if Budget < 6.75, right otherwise. Given Budget = 5 in $\boldsymbol{X}_p$. **We go left**.

The above procedure leads to the **terminal node: 2.253**. Although it sounds a bit tedious, in practice, these prediction exercises are carried out in a very efficient fashion. In R, we can use the `predict` command to perform prediction. All we need is a **constructed tree** and proper **input data**:

```
# 3. predict with RT
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
out<-rpart(OpenBox~Constant+Animation+Crime+Romance+Budget
           +Weeks+Screens,data=dat, method="anova")
# 3.1 define new INPUT
INPUT <- data.frame(Constant=1,Animation=0,Crime=0,Romance=1,
                   Budget=5,Weeks=4,Screens=3.5)
# 3.2 make prediction
predict(out,INPUT,type="vector")
```

2.3.1 The R-square of Tree Estimation

The prediction technique can be applied to out-of-sample as well as in-sample. If we replace the prediction input $\boldsymbol{X}_p$ with the original input $\boldsymbol{X}$, we are predicting the results of $\boldsymbol{y}$ conditional on $\boldsymbol{X}$ in the sense of in-sample estimation. The prediction $\hat{\boldsymbol{y}}$ can be interpreted as the in-sample fit using the tree algorithm. Then, it is straightforward to compute the residual and estimate the centered R^2.

$$\begin{aligned} \boldsymbol{e} &= \boldsymbol{y} - \hat{\boldsymbol{y}}, \\ R_c^2 &= 1 - \frac{\boldsymbol{e}^\top \boldsymbol{e}}{(\boldsymbol{y} - \bar{\boldsymbol{y}})^\top (\boldsymbol{y} - \bar{\boldsymbol{y}})}, \end{aligned}$$

Note that all these estimates are based on highly nonlinear assumption. We revisit the movie forecasting exercise and compute the R^2. R codes are presented below.

```
# 4. estimate R-square
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
y <- dat$OpenBox
out<-rpart(y~Constant+Animation+Crime+Romance+Budget
           +Weeks+Screens,data=dat, method="anova")
INPUT <- dat[,2:ncol(dat)]
yf <- predict(out,INPUT,type="vector")
e = y-yf
# 4.1 compute R-square
1-sum(e^2)/sum((y-mean(y))^2)
```

Note that the estimated tree R^2 is **0.4720**, which is higher than the R^2 by OLS.

The R^2 we introduced in this section is the overall R^2. In R, there is function `rsq.rpart` that computes and plots approximate R-squared and relative error for different splits (2 plots). You can try that and compare the differences.

2.4 Pruning

An overly complex tree can be computationally expensive to construct and it for sure will cause all the issues that overfitting does. There are various ways that we can do to contain the level of complexity for a tree, or **pruning** a tree. Here we introduce several popular methods of pruning.

In R, the pruning (tuning) process is implemented through the `rpart.control` command. Note that you must properly define its feature before incorporating it into the `rpart` command. Its features include:

- `minsplit`: the minimum number of observations that must exist in a node in order for a split to be attempted.
- `minbucket`: the minimum number of observations in any terminal "leaf" node. If only one of minbucket or minsplit is specified, the code either sets minsplit to minbucket*3 or minbucket to minsplit/3, as appropriate.
- `cp`: complexity parameter. Any split that does not decrease the overall lack of fit by a factor of cp is not attempted. For instance, with anova splitting, this means that the overall R-squared must increase by cp at each step. The main role of this parameter is to save computing time by pruning off splits that are obviously not worthwhile. Essentially, the user informs the program that any split which does not improve the fit by cp will likely be pruned off by cross-validation, and that hence the program need not pursue it.
- `maxcompete`: the number of competitor splits retained in the output. It is useful to know not just which split was chosen, but which variable came in second, third, etc.
- `maxsurrogate`: the number of surrogate splits retained in the output. If this is set to zero the compute time will be reduced, since approximately half of the computational time (other than setup) is used in the search for surrogate splits.
- `usesurrogate`: how to use surrogates in the splitting process. 0 means display only; an observation with a missing value for the primary split rule is not sent further down the tree. 1 means use surrogates, in order, to split subjects missing the primary variable; if all surrogates are missing the observation is not split. For value 2 ,if all surrogates are missing, then send the observation in the majority direction. A value of 0 corresponds to the action of tree, and 2 to the recommendations of Breiman et.al (1984).
- `xval`: number of cross-validations.

- `surrogatestyle`: controls the selection of a best surrogate. If set to 0 (default) the program uses the total number of correct classification for a potential surrogate variable, if set to 1 it uses the percent correct, calculated over the non-missing values of the surrogate. The first option more severely penalizes covariates with a large number of missing values.
- `maxdepth`: Set the maximum depth of any node of the final tree, with the root node counted as depth 0. Values greater than 30 rpart will give nonsense results on 32-bit machines.

The **default values** for each feature is presented below:

```
rpart.control(minsplit = 20, minbucket = round(minsplit/3),
    cp = 0.01, maxcompete = 4, maxsurrogate = 5,
    usesurrogate = 2, xval = 10, surrogatestyle = 0,
    maxdepth = 30, ...)
```

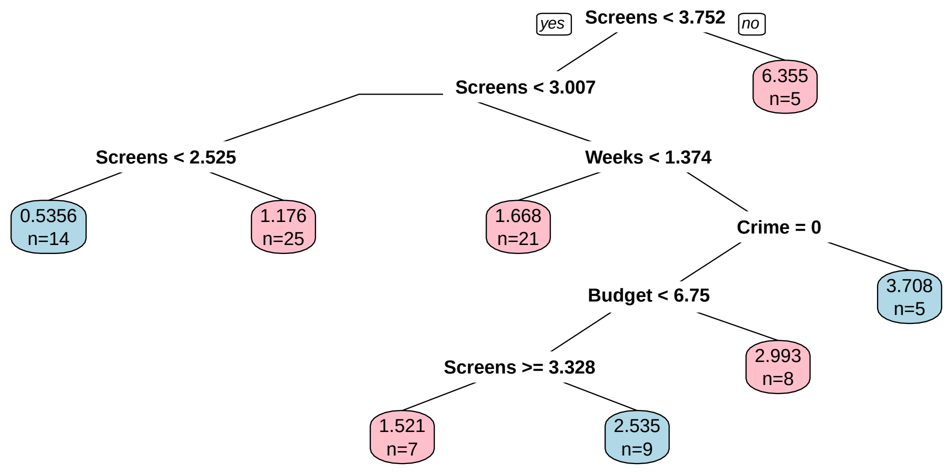
Note that this is exactly the pruning values we used for **previous** regression tree estimation.

2.4.1 Setting the Minimum Number on Each Node Split

A straightforward way is to change the minimum allowed number of observations for each node split. For example, instead of the default value at 20, let us set the value of `minsplit` to 10 using the `rpart.control` command. R codes are presented below. The pruned tree is shown in Figure 2.3.

```
# 5. pruning: set min on splits
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
# 5.1 prune the tree
y <- dat$OpenBox
CON = rpart.control(minsplit = 10)
out<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
           +Screens,data=dat,method="anova",control=CON)
prp(out,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
# 5.2 compute R-square
INPUT <- dat[,2:ncol(dat)]
yf <- predict(out,INPUT,type="vector")
e = y-yf
1-sum(e^2)/sum((y-mean(y))^2)
```

Figure 2.3: A Pruned Tree by Tuning `minsplit`



As expected, the pruned tree is larger (deeper) than the one in Figure 2.2. In fact, the numbers in a node can be smaller than 20. Since the tree is larger and more complicated, it is not a surprise to see that the estimated centered $R^2 = 0.5324$. But be cautious here, a good in-sample fit does not guarantee a sound out-of-sample performance.

2.4.2 Setting the Minimum Number of Observations for Terminal Nodes

A less direct but also efficient way is to **set a minimum sample size for each terminal node** using the `minbucket` feature. As this number increases, less terminal nodes are needed, which certainly leads to a simplified tree structure. If this number decreases, a deeper tree will be grown. Note that `minbucket` usually pairs with `minsplit`. Here, let us try the following various exercises.

(a) minsplit = 10, minbucket = 1;

(b) minsplit = 50, minbucket = 20;

(c) minsplit = 2, minbucket = 1.

R codes are presented below. The pruned trees are shown in Figure 2.4.

```
# 6. more pruning exercises
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
library(rpart.plot)
y <- dat$OpenBox
# 6.1 exercises
CON1 = rpart.control(minsplit = 10,minbucket = 1)
CON2 = rpart.control(minsplit = 50,minbucket = 20)
CON3 = rpart.control(minsplit = 2,minbucket = 1)
out1<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
            +Screens,data=dat,method="anova",control=CON1)
out2<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
            +Screens,data=dat,method="anova",control=CON2)
out3<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
            +Screens,data=dat,method="anova",control=CON3)
prp(out1,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
prp(out2,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
prp(out3,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
# 6.2 compute R-square
INPUT <- dat[,2:ncol(dat)]
yf1 <- predict(out1,INPUT,type="vector")
yf2 <- predict(out2,INPUT,type="vector")
yf3 <- predict(out3,INPUT,type="vector")
1-sum((y-yf1)^2)/sum((y-mean(y))^2)
1-sum((y-yf2)^2)/sum((y-mean(y))^2)
1-sum((y-yf3)^2)/sum((y-mean(y))^2)
```

The pruned trees can be more or less complicated than the default tree, depending on the tuning parameters. Case (b) is definitely an extreme case, where the split occurs just once and the number of observations in two leaves is even.

We also compute the R-squares for the above three cases, where

(a) 0.8586,

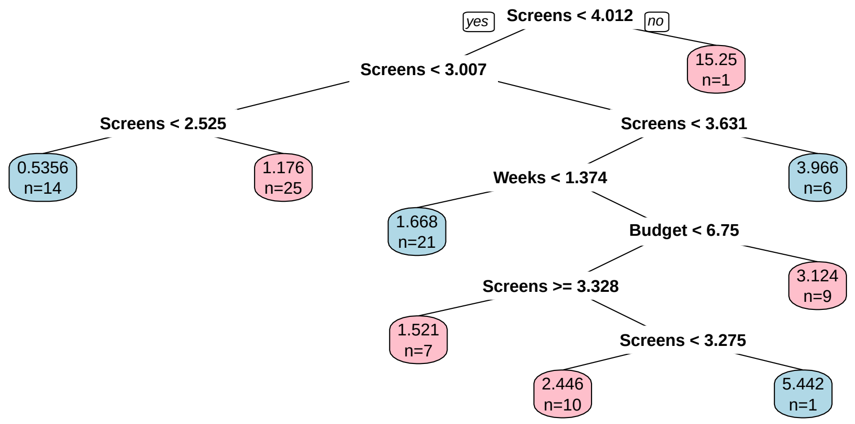
(b) 0.2000,

(c) 0.9006.

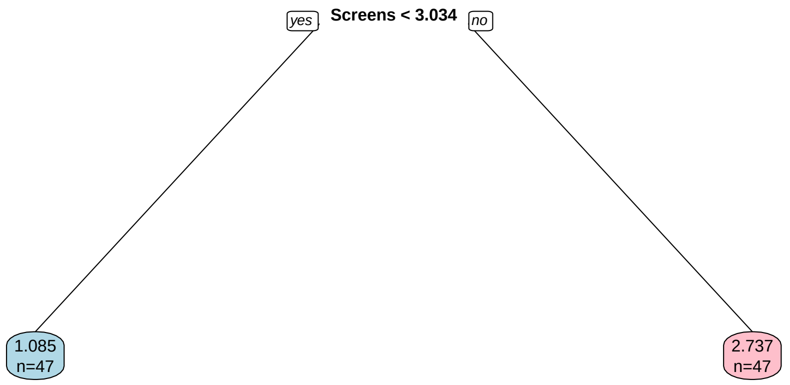
It is shown that as tree gets more complicated, the in-sample fit turns better. Why is the R^2 in case (b) at a fairly low level?

Figure 2.4: Pruned Trees by Tuning `minsplit` and `minbucket`

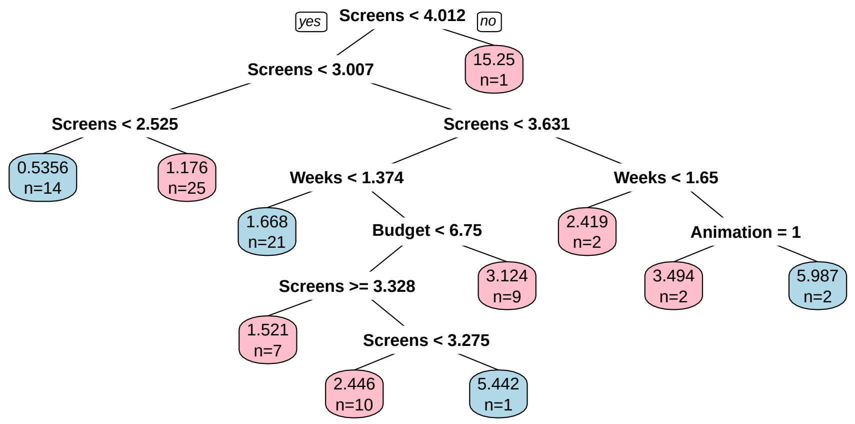
(a) `minsplit = 10, minbucket = 1`



(b) `minsplit = 50, minbucket = 20`



(c) `minsplit = 2, minbucket = 1`



2.4.3 Tuning the Complexity Parameter

The balance between fit and complexity of a tree resembles the bias–variance tradeoff we studied in Section 1. For larger trees with a given sample size, there will be fewer classification errors, implying less bias. But larger trees will have terminal nodes with less observations, which implies a degree of greater instability and variance.

We can also tune the complexity parameter cp by assigning the following two values: (a) 0.05 and (b) 0.0001. Note that the default value of cp is 0.01. R codes are presented below, and the Pruned trees are plotted in Figure 2.5.

```
# 7. pruning: complexity
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
library(rpart.plot)
y <- dat$OpenBox
# 7.1 exercises
CON1 = rpart.control(cp = 0.05)
CON2 = rpart.control(cp = 0.0001)
out1<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
            +Screens,data=dat,method="anova",control=CON1)
out2<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
            +Screens,data=dat,method="anova",control=CON2)
prp(out1,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
prp(out2,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
    box.col=c("pink","lightblue"))
```

Obviously, the tree with a harsher restriction on cp is less complicated, since the splits cannot be easily executed due to the requirement of larger R^2 improvement. If you estimate the R^2, you should find that the second tree has a better fit than the default tree and the first case.

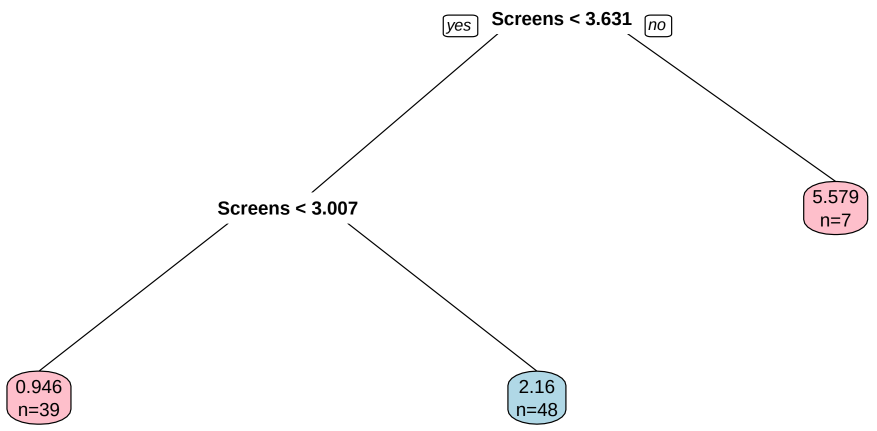
2.4.4 A Huge Tree

In this section, we want to construct a highly complicated tree, in other words, a huge tree. This means, we must tune the above three tuning parameters simultaneously to avoid **indirect** restrictions. We consider the following arrangement

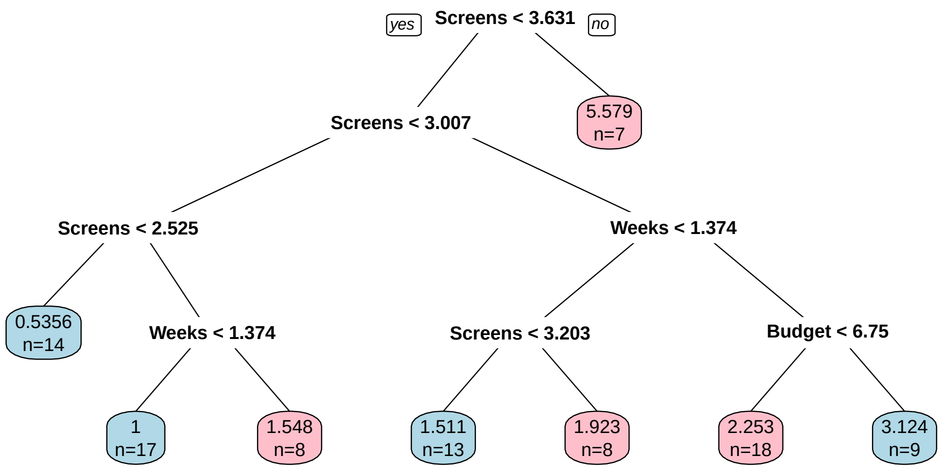
```
minsplit = 5, minbucket = 1, cp = 0.0001
```

Figure 2.5: Pruned Trees by Tuning `cp`

(a) `cp = 0.05`



(b) `cp = 0.0001`



The R codes are presented below and the huge tree is plotted in Figure 2.6. Note that we are using the default `rpart.plot` function.

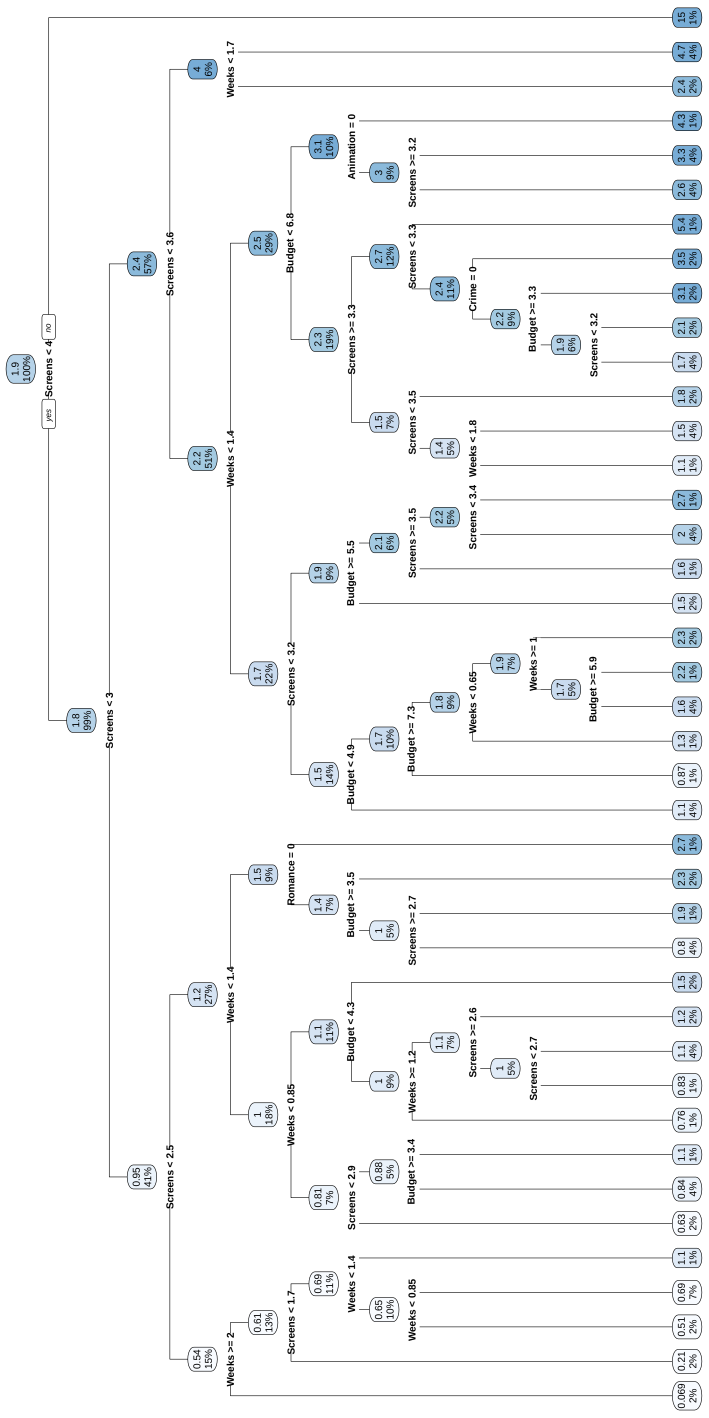
```
# 8. a huge tree
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
library(rpart.plot)
y <- dat$OpenBox
CON = rpart.control(minsplit = 5, minbucket = 1, cp = 0.0001)
out<-rpart(y~Constant+Animation+Crime+Romance+Budget+Weeks
            +Screens,data=dat,method="anova",control=CON)
rpart.plot(out)
```

2.5 Optimizing Regression Tree

In previous sections, we have introduced many tuning parameters. In the tree literature, these tuning parameters are also called **hyperparameters**. We can perform a grid search to automatically search across a range of models with various tuning parameters to identify the optimal hyerparameter setting. In the next example, we test a range of values on `minsplit` from 2-40 and vary `minbucket` from 1-10. R codes are presented below.

```
# 9. optimize hyperparameters
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
library(rpart.plot)
# 9.1 get parameter space
hyper_grid <- expand.grid(minsplit=seq(2, 40, 2),
                          minbucket=seq(1, 10, 1))
# 9.2 start loops
models = list()
for (i in 1:nrow(hyper_grid)) {
  # retrieve minsplit, minbucket values at row i
  minsplit <- hyper_grid$minsplit[i]
  minbucket <- hyper_grid$minbucket[i]
  # train a model and store in the list
  models[[i]] <- rpart(OpenBox~Constant+Animation+Crime+Romance
                  +Budget+Weeks+Screens,data=dat,method="anova",
    control = list(minsplit = minsplit, minbucket = minbucket)
  )}
```

Figure 2.6: A Huge Tree



Note that we save each model into its own list. We can then create a function to extract the minimum error associated with the optimal value of cost complexity for each model.

```
# 9.3 determine the best model (combination)
# define functions to get min error
get_min_error <- function(x) {
  min    <- which.min(x$cptable[, "xerror"])
  xerror <- x$cptable[min, "xerror"]
}
error = matrix(NA, nrow=20, ncol=10)
for (i in 1:nrow(hyper_grid)) {
  error[i] <- get_min_error(models[[i]])
}
INDEX <- which.min(error)
c(minsplit=hyper_grid$minsplit[INDEX],
  minbucket=hyper_grid$minbucket[INDEX],
  error=error[INDEX])
```

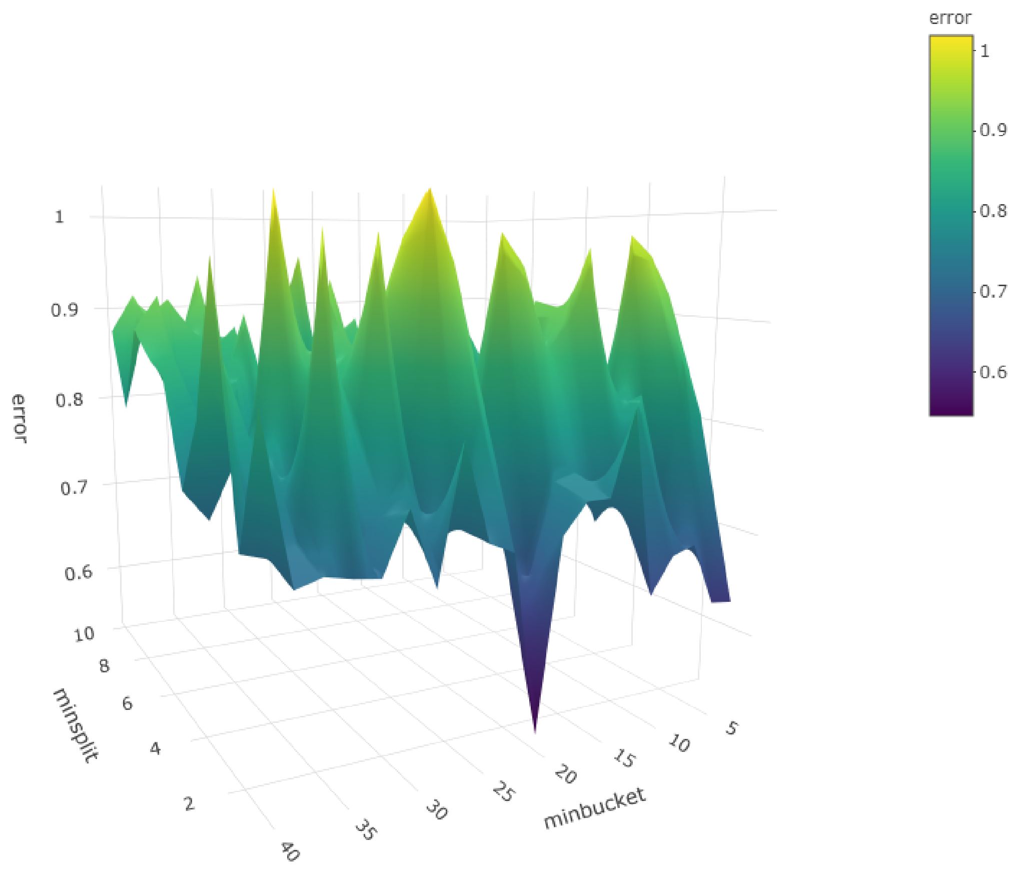
Note that since the estimation errors are computed via ten-fold validation. Therefore, the optimal value should change each time you run your codes. In our estimation, this result is

```
  minsplit  minbucket      error
20.0000000  1.0000000  0.5460681
```

To have a better visualization, we plot the error values against `minsplit` and `minbucket` in a 3D surface plot. We use the `plotly` package. R codes are presented below.

```
# 9.4 plot the surface
# install.packages("plotly") <= you may need to install
library(plotly)
plot_ly(y=~seq(2, 40, 2), x=~seq(1, 10, 1), z=~error) %>%
  add_surface() %>%
layout(scene = list(xaxis = list(title = 'minsplit'),
                    yaxis = list(title = 'minbucket'),
                    zaxis = list(title = 'error')))
```

Figure 2.7: Optimization Process



3 Bootstrap and Bagging Tree

According to Merriam-Webster, the **definition of learning** can be interpreted as **modification of a behavioral tendency by experience**. We have thus far focused on statistical procedures that produce a **single** set of results: regression coefficients, measures of fit, residuals, classifications, and others. There is but **one** regression equation, **one** set of smoothed values, or **one** classification tree. Obviously, one won't learn much from just one set of results.

In this section, we shift to statistical learning that builds on **many sets of outputs** aggregated to produce the final outcomes. Such algorithms make a number of passes over the data. On each pass, inputs are linked to outputs just as previously. But the ultimate results of interest are the **collection** of all the results from all passes over the data.

Bagging, which stands for **Bootstrap Aggregation**, is perhaps the earliest procedure to exploit sets of fitted values over random samples of the data. Bagging is invented by Breiman (1996) as an algorithm that can help improve the performance of fitted values from a given statistical procedure. Theoretically, it can work on many estimators or algorithms. In this section, we first introduce the concept of bootstrap, then we combine bootstrap with the regression tree from Section 2.

3.1 Bootstrap

The term **bootstrap**, which was introduced to statistics by Efron (1979), is taken from the phrase "to pull oneself up by one's own bootstraps". It is the practice of estimating properties of an estimator (such as its variance) by measuring those properties through sampling from an **approximating distribution**. Bootstrapping relies heavily on **random sampling with replacement**.[18] A bootstrap sample is always a subset of the original sample.

3.1.1 The Basic Concept

A Russian **matryoshka doll** is a nest of wooden figures, usually with slightly different features painted on each. Call the outer figure **doll 0**, the next figure **doll 1**, and so on. See Figure 3.1. Suppose we are not allowed to observe doll 0 – it represents the population in a sampling scheme. We wish to estimate the area n_0 of **red cheek**[19] on her face. Let n_i

[18]The principle of simple random sampling is that every object has the same probability of being chosen. In small populations and often in large ones as well, such a sampling procedure is typically done **without replacement**, i.e., one deliberately avoids choosing any member of the population more than once. Although simple random sampling can be conducted with replacement instead, this is less common and would be formally referred as simple random sampling **with replacement**, in which the random sampling exhibits independence. Note that sampling implemented without replacement is no longer independent, but still satisfies exchangeability, and hence many results still hold.

[19]This is actually an art representation of freckles.

denote the area of red cheek on the face of doll i. Since doll 1 is smaller than doll 0, n_1 is likely to be an underestimate of n_0, but it seems reasonable to suppose that the ratio of n_1 to n_2 should be close to the ratio of n_0 to n_1. That is $n_1/n_2 \approx n_0/n_1$, so that $\hat{n}_0 = n_1^2/n_2$ might be a reasonable estimate of n_0.

Figure 3.1: A Russian Matryoshka Doll

The key feature of this argument is our hypothesis that the relationship between n_2 and n_1 should closely resemble that between n_1 and the unknown n_0. Under the (fictitious) assumption that the relationships remain **identical**, we equated the two ratios and obtained our estimate of $\hat{n}_0$. Of course, we could refine the argument by delving more deeply into the nest of dolls, adding correction terms to $\hat{n}_0$ so as to take account of the relationship between doll i and doll $i + 1$ for $i \geq 1$.

The above intuition implies that the population underlying a sample data (sample $\rightarrow$ population) can be modeled by resampling the sample data, which also provides the inference about the sample data (resampled $\rightarrow$ sample). As the population is unknown, the true error in a sample statistic away from its population value is unknown. In **bootstrap samples**, the population is in fact the sample data, which is known beforehand; hence the quality of inference of the **true sample** is measurable from the **resampled data** (resampled $\rightarrow$ sample).

3.1.2 Generating Bootstrap Samples

In this section, we practice generating bootstrap samples. Since each observations in a bootstrap sample is treated as independent, they have equal probability of being chosen. We revisit the movie forecasting exercise in Section 2, and generate 1000 bootstrap samples using the built-in `sample` command.[20] Moreover, we compute the average open box office for each bootstrap sample and compare the 1000 mean values with the actual sample mean in a histogram. R codes are presented below.

```
# 1. Bootstrapping data
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
y = dat$OpenBox
# 1.1 create a bootstrap sample matrix
B = 1000
n = length(y)
bootsamples = matrix(sample(y, size = n*B, replace = TRUE), n, B)
# 1.2 compute the mean and plot the histogram
bootmean = apply(bootsamples, 2, mean)
hist(bootmean,xlab="Bootstrap Means",col="lightblue",
     breaks=15,ylim=c(0,20))
lines(c(mean(y),mean(y)),c(0,20),lwd=3,col="red")
legend("topright",legend=c("Sample Mean"),col=c("red"),lty=1)
```

The histogram that compares bootstrap sample means with the actual sample mean is contained in Figure 3.2. The bar and the solid line represent the histogram of bootstrap sample means and the actual sample mean, respectively. The 1000 bootstrap sample means assemble its own **empirical** distribution and the histogram mimics its **empirical distribution function (EDF)**. Not a surprise that the EDF centers around the actual sample mean, which equals 1.9110.

In any sample data, a specific dependent observation y_i is always tied to a vector of the independent observations $\boldsymbol{X}_i$, as if they are a **pair**. In this fashion, each bootstrap sample consists of a pair $\{\boldsymbol{y}^{(b)}, \boldsymbol{X}^{(b)}\}$ for $b = 1, ..., B$. The bootstrapping procedure described above is also called **pairs bootstrap**.

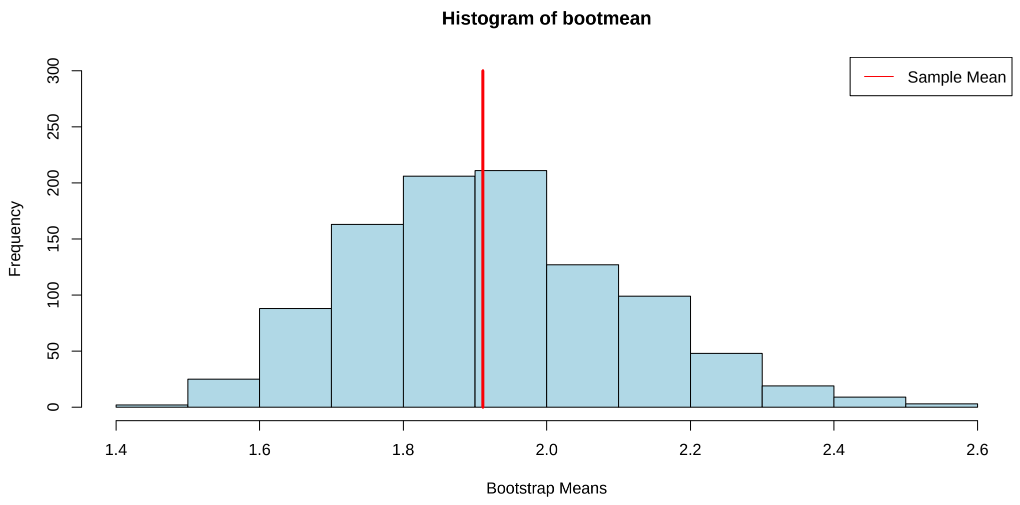
3.1.3 Applying RT to Each Bootstrap Sample

Although all the B bootstrap samples are subsets of the original data sample, each bootstrap sample is different and hence contains **heterogeneous information** that approximate the true DGP. Incorporating these information into analyses is like **learning** from past experience. Moreover, bootstrap samples are generated from the original data without using any extra data, and the number of bootstrap sample is determined by the parameter B. A larger value of B implies more information to learn from.

We follow the exercise in Section 3.1.2 and generate $B = 1000$ bootstrap samples. Note that we make use of the index parameter `IN` and apply it to both $\boldsymbol{X}$ and $\boldsymbol{y}$ to generate boot-

[20]Make sure you tell `sample` to do random sampling **with replacement**.

Figure 3.2: Histogram of Bootstrap Sample Means and Actual Sample Mean



strap samples. Then, we apply regression tree to each bootstrap sample. As an example, we pick the first 4 trees to present in Figure 3.3. The R codes are presented below.

```
# 2. bootstrap with RT
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
B = 1000
n = nrow(dat)
# 2.1 obtain bootstrap index
IN = matrix(sample(1:n, size = n*B, replace = TRUE), n, B)
# 2.2 apply RT to each bootstrap sample
library(rpart)
library(rpart.plot)
models = list()
for (i in 1:B) {
  dat2 <- dat[IN[,i],]
  models[[i]] <- rpart(OpenBox~Constant+Animation+Crime+Romance
        +Budget+Weeks+Screens,data=dat2,method="anova")
  }
# 2.3 plot first 4 results
par(mfrow=c(2,2))
for (i in 1:4) {
  TAR = models[[i]]
  prp(TAR,extra=1,faclen=10,varlen=15,cex=1.2,digits=4,
      box.col=c("pink","lightblue"),
      main=paste("(",letters[i],") ","Tree Number ",i,sep=""))
  }
```

It is not a surprise that all the 4 trees are different, since they are constructed using separate (bootstrap) samples. Although they are not identical, but they do share certain level of similarity. Moreover, we notice that the predictors used as a part of the splitting rule within each splitting node are quite similar among the trees (for instance, `screens`). It seems the **important predictors** are still important even if the bootstrap sample is used. We will further investigate this phenomenon in Section 3.4.

3.2 Algorithm of Bagging Tree

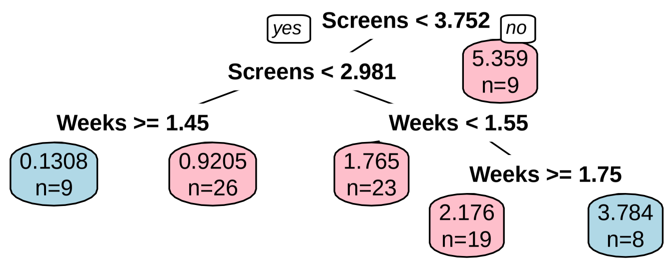
In Section 3.1.3, we apply RT to each bootstrap sample, and briefly compare the generated trees by each bootstrap sample. In this section, we aggregate the results of all the trees (in other words, a **forest**) to produce one general final output. In a prediction exercise, this involves averaging the prediction from each tree (**learning process**) to obtain the final prediction.

For the original data having n observations, bagging tree follows the below procedure:

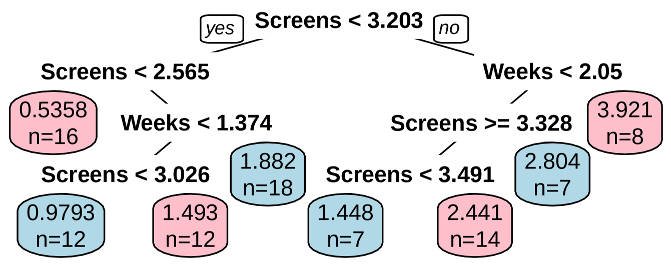
Step 1. Take a random sample of size n with replacement from the data (bootstrapping).

Figure 3.3: Plot Trees for Selected Bootstrap Samples

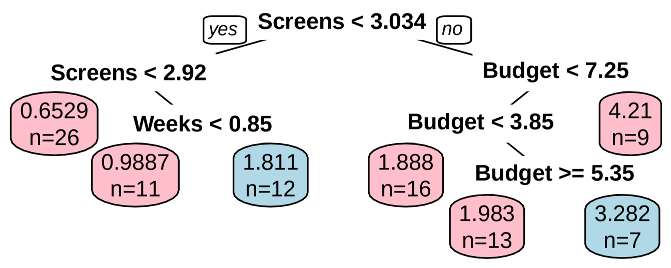
(a) Tree Number 1



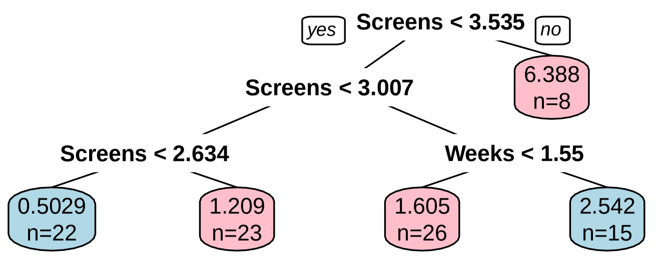
(b) Tree Number 2



(c) Tree Number 3



(d) Tree Number 4



Step 2. Construct a regress tree based on the bootstrap sample.

Step 3. Obtain the prediction using the constructed regression tree, denoted it as $\hat{y}^{(1)}$.

Step 4. Repeat Steps 1 to 3 for B times and collect $\{\hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(B)}\}$.

Step 5. Compute the final prediction as an average of B predictions

$$\hat{y}_{\text{BAG}} = \frac{1}{B}\sum_{b=1}^{B}\hat{y}^{(b)}. \tag{3.1}$$

Since each bootstrap sample is treated **independently**, intuitively, they are **equally** important to us and carry the **equal** amount of information asymptotically. Therefore, it is sensible to consider a **simple average** of all the predictions in Equation (3.1).

In R, we can simply use the improved prediction package ipred to conduct bagging tree. The function we use is bagging. Note that most of pruning-related optional settings applicable for rpart also work on bagging. See bagging for more details. To make a prediction, we simply apply the new input data to the trained results and predict with the command predict just as in rpart. For simplicity, we define inputs as $\boldsymbol{X}_p = [1,0,0,1,5,4,3.5]$.

Compare the results with rpart, what do you observe?

```
# 3. bagging tree
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# install.packages(ipred)
library(ipred)
library(rpart)
B = 100
out1 <- bagging(OpenBox~Constant+Animation+Crime+Romance+Budget
                +Weeks+Screens,data=dat,nbagg=B)
out2 <- rpart(OpenBox~Constant+Animation+Crime+Romance+Budget
           +Weeks+Screens,data=dat, method="anova")
# 3.1 predict with bagging tree and RT
INPUT <- data.frame(Constant=1,Animation=0,Crime=0,Romance=1,
                    Budget=5,Weeks=4,Screens=3.5)
BAG = predict(out1,INPUT)
RT = predict(out2,INPUT,'vector')
data.frame(BAG,RT=as.numeric(RT))
```

3.2.1 A More Complicated Comparison between RT and BAG

We now consider bagging tree more seriously in a prediction exercise. In the movie data, we have $n = 94$ observations. We set the first 80 observations as the **training set** $\{\boldsymbol{y}^{tr}, \boldsymbol{X}^{tr}\}$ and the rest 14 observations as the **evaluation set** $\{\boldsymbol{y}^{ev}, \boldsymbol{X}^{ev}\}$. We first apply the bagging tree algorithm to the training set and construct the collection of trees. Then, we predict $\boldsymbol{y}^{ev}$ using the constructed bagging trees and $\boldsymbol{X}^{ev}$ as the input variable.

```
# 4. bagging prediction
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(ipred)
library(rpart)
B = 100
# 4.1 training
data_train = dat[1:80,]
out1 <- bagging(OpenBox~Constant+Animation+Crime+Romance+Budget
                +Weeks+Screens,data=data_train,nbagg=B)
out2 <- rpart(OpenBox~Constant+Animation+Crime+Romance+Budget
              +Weeks+Screens,data=data_train, method="anova")
```

To evaluate the forecast accuracy of $\hat{\boldsymbol{y}}^{ev}$, we compute the mean squared forecast error (MSFE) as follows:

$$\text{MSFE} = \frac{1}{14}\sum_{i=1}^{14}(y_i^{ev} - \hat{y}_i^{ev})^2,$$

where y_i^{ev} is the actual observation in the evaluation set and $\hat{y}_i^{ev}$ is the related forecast. Finally we compare the MSFEs of bagging tree with those of regression tree. The R codes are presented below.

```
# 4.2 evaluate prediction
INPUT <- dat[81:nrow(dat),2:ncol(dat)]
BAG = predict(out1,INPUT)
RT = predict(out2,INPUT,'vector')
yf = data.frame(BAG,RT=as.numeric(RT))
y0 = dat[81:nrow(dat),1]
apply((yf-y0)^2,2,mean)
```

If we try the above codes multiple times, you will find out that the result of bagging tree changes each time, whereas the regression tree result remains the same. This is due to the fact the **randomness** is introduced during the bootstrapping procedure for bagging tree. Moreover, if we set B to be a very large number,[21] the result of bagging tree becomes less volatile.

Overall, despite what computer you are using and what number you assign to B (as long as it is reasonable), we find that bagging tree always yields a smaller MSFE than regression tree in almost all the cases. This implies that the bagging tree algorithm is effective in this exercise, as the **learning process is useful**.

3.3 Out-Of-Bag Error

In previous examples, a tree is grown with **training data**, and the training data used to grow the tree are **used again** to compute the terminal nodes and make predictions. The training data are dropped down the tree to determine how well the tree performs. The training data are said to be **resubstituted** when tree performance is evaluated.

Each time we draw a bootstrap sample, we use only **63%** of the observations asymptotically. The rest of the observations are called **out-of-bag (OOB)** observations and they are an ideal test set to evaluate the constructed tree.

For each input observation $\boldsymbol{X}_i$, we find the prediction $\hat{y}_i^{(b)}$ for all bootstrap samples b which do not contain $\boldsymbol{X}_i$. Theoretically, there should be around $0.37B$ of them. We average these predictions to obtain $\hat{y}_i^{oob}$. The **OOB error** is then $y_i - \hat{y}_i^{oob}$ and we compute its mean squared forecast error as usual

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i^{oob})^2$$

to evaluate all the observations. In practice, people also use the root MSE to control the magnitude of errors.

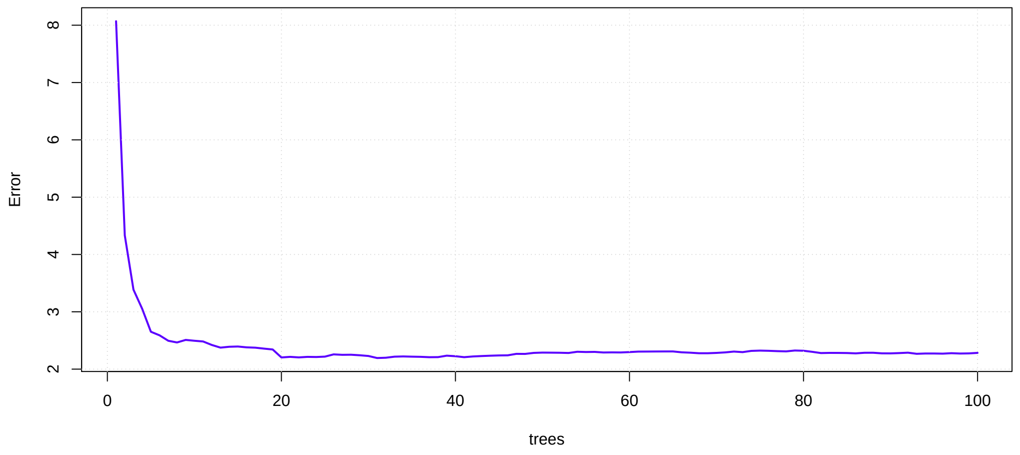
Here, we employ another package called `randomForest` to conduct such an experiment.[22] As the name indicates `randomForest` is designed for random forest estimation. In fact, the bagging tree method is a special case of random forest. The main difference is that bagging use **all the predictors** to generate each tree in the forest. In this exercises, we first conduct bagging tree estimation, then we plot the OOB MSE against the number of trees. Note that by default the OOB feature is on for `randomForest`. The R codes are presented below. Results are plotted in Figure 3.4.

[21] Although it varies from one computer to another, the computational cost generally increases for a larger B.

[22] The `ipred` package also has OOB feature, but it lacks the function that plots the OOB MSE against the number of trees.

```
# 5. OOB MSE
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# 5.1 bagging with randomForest
k = ncol(dat)-1
out <- randomForest(OpenBox~.,data=dat, mtry=k, ntree=100)
# 5.2 plot error against number of trees
plot(out,col="blue",lwd=2,main="")
grid()
```

Figure 3.4: OOB MSE on Number of Grown Trees



As we can see, the OOB MSE turns smaller as the number of trees increases. In fact, it becomes stable after the 20th tree.

3.3.1 Finding the Optimal Leaf Size

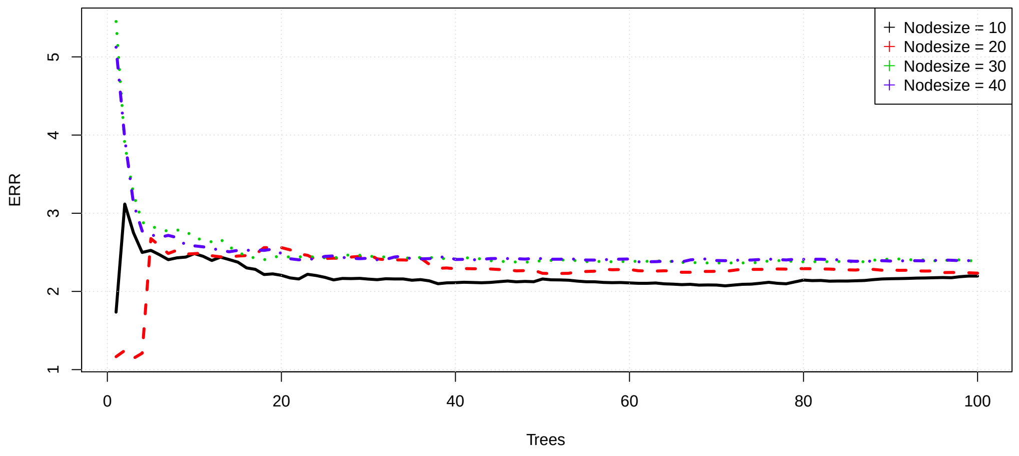
In Section 2.4.2, we prune the tree by arbitrarily setting the minimum leaf size for each terminal node. In bagging tree, we can use the OOB MSE to evaluate various numbers of minimum leaf size, and thus find the optimal leaf size.

We again consider the movie forecasting exercise, where four alternative minimum leaf sizes are examined: 10, 20, 30, and 40. We grow bagging trees by imposing one of the minimum leaf sizes **each time**, using the optional feature `nodesize` in `randomForest`. We then compute the associated OOB MSE for each selected leaf size.

The R codes are presented below. We plot the MSEs against the number of grown trees in Figure 3.5.

```
# 6. tuning by OOB MSE
library(randomForest)
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
k = ncol(dat)-1
B = 100;
# 6.1 estimation
ERR = matrix(NA,B,4);
for (i in 1:4){
  out <- randomForest(OpenBox~.,data=dat, mtry=k, ntree=B,
                      nodesize=i*10)
  ERR[,i] = out$mse;
}
# 6.2 plot results
matplot(ERR, type = "l",pch=1:4,col = 1:4,lwd=3,xlab="Trees")
legend("topright",legend=c("Nodesize = 10",
                           "Nodesize = 20",
                           "Nodesize = 30",
                           "Nodesize = 40"), col=1:4, pch=3)
grid()
```

Figure 3.5: Tuning Leaf Size by OOB MSE



Results in Figure 3.5 clearly prefer the minimum leaf size of 10. Of course, due to randomness, you may not obtain the same results each time and the results may not be optimal due to the limited number of bootstraps. Set B to 1000, and check if the pattern becomes more convergent.

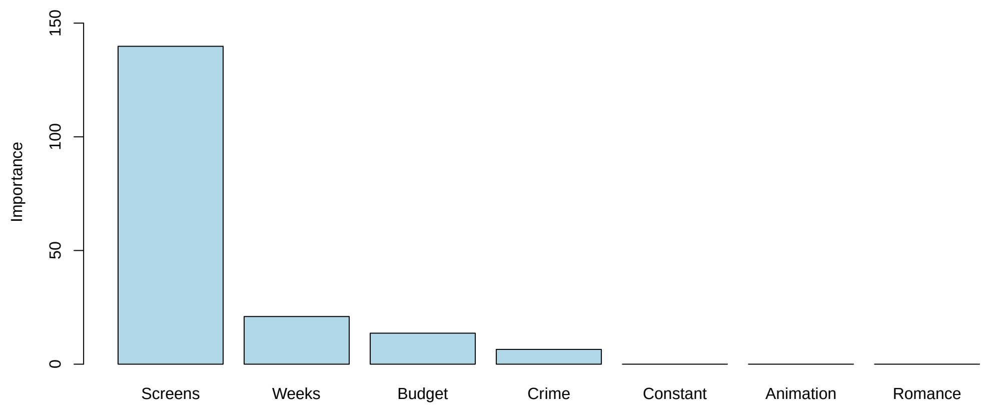
3.4 Predictor Importance

From Figure 2.2, we notice that certain variable appears **more frequently** in the splitting nodes than others. This implies that we can measure importance of each predictor by regression tree. In R, we define the feature `variable.importance` of `rpart`, part of the result for tree, by **summing changes in the SSR** attributed to splits on every predictor and dividing the sum by the number of branch nodes. This output contains k values associated with the k predictors in the exactly same order. A higher value indicates greater importance of the predictor.

The R codes are presented below. The results are demonstrated by a `bar` plot in Figure 3.6.

```
# 7. variable importance
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# 7.1 RT
library(rpart)
out1<-rpart(OpenBox~.,data=dat, method="anova")
RT=c(out1$variable.importance,Constant=0,Animation=0,Romance=0)
barplot(RT,ylab="Importance",col="lightblue",ylim=c(0,160))
```

Figure 3.6: Predictor Importance by Regression Tree



Not a surprise that the variable `Screens` is the most important predictor. On the other hand, predictors `Animation` and `Romance` have zero contribution. They are not effective at reducing the SSR.

3.4.1 Measuring Importance Using OOB Errors

We can apply the above measure to Bagging easily as well. Instead of focusing on one tree, we can record the total amount that the SSR is reduced due to splits over a given predictor, **averaging across all B trees**. A large value indicates an important predictor.

Of course, the above idea needs some standardization, since different bootstrap samples correspond to varying levels of SSR reduction. Hence, an improved measure is to compare the average "reduction ratio", which is the ratio of total SSR reduction to the centered total sum of squares.

By means of bagging, each tree is grown with its respective randomly drawn bootstrap sample and the excluded data from the OOB for that tree. The OOB sample can be used to evaluate the tree without the risk of overfitting since the observations are not involved in constructing the tree. To determine importance, a given predictor is randomly permuted in the OOB sample and the prediction error of the tree on the modified OOB sample is compared with the prediction error of the tree in the untouched OOB sample. This process is repeated for each tree and each predictor variable. The average of the gaps in prediction errors across all OOB samples provides an estimate of the overall decrease in accuracy that the permutation of removing a specific predictor induces.

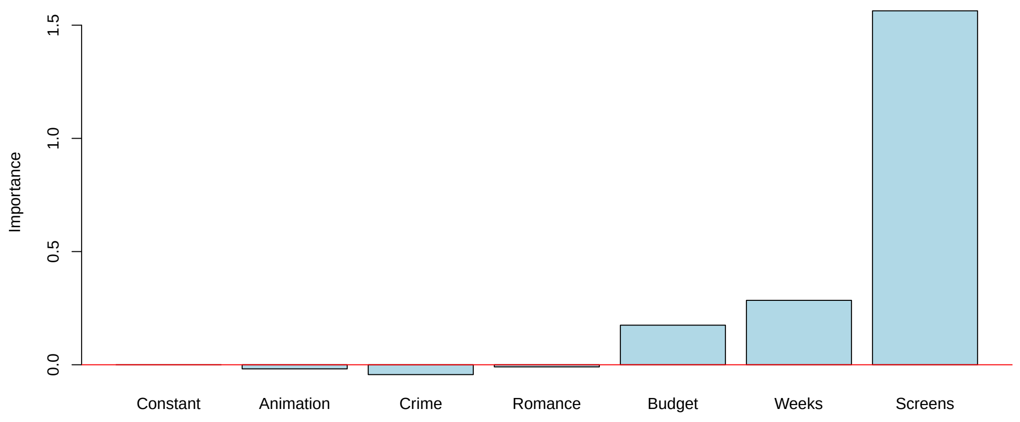
We try this on the `movie data` and the R codes are listed below. The results are illustrated in Figure 3.7.

```
# 8. variable importance - BAG
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
k = ncol(dat)-1
out <- randomForest(OpenBox~.,data=dat, mtry=k, ntree=100,
                    importance=T)
# 8.1 plot results
BAG = out$importance[,1]
barplot(BAG,ylab="Importance",col="lightblue")
lines(c(-3,10),c(0,0),col="red",lwd=1)
```

The OOB importance estimates also reveal that the predictor `Screens` is the most crucial variable. We also notice that the OOB importance estimates can sometimes be negative[23] for certain predictors. Moreover, these estimates are **not in the same magnitude** as the estimates for regression tree, since one is ratio-based and the other is an absolute quantity (the SSR reduction).

[23] **Why? Figure out some logical explanation yourself.**

Figure 3.7: Predictor Importance by Bagging Tree



4 Random Forest

Just as in bagging, imagine growing a large number regression trees with bootstrap samples from training data. But now, as each tree is grown, take a **random** sample of predictors **before** each node is split. Repeat this process for each prospective split. **Do not prune**. Thus, each tree is produced from a random sample of cases, and at each split a **random sample of predictors**. Finally, average over trees as in bagging, but only when that case is OOB. Such procedure is denoted as **random forest** in Breiman (2001).

4.1 Random Forest Algorithm

The Random Forest algorithm is very much like the bagging algorithm. In fact, bagging tree can be regarded as a **special** case of random forest. For original data having n observations, random forest takes the following form:

Step 1. Take a random sample of size n with replacement from the data (bootstrapping).

Step 2. Construct a regress tree based on the bootstrap sample.

- (i) For each potential partitioning of the data, **a random sample (without replacement)** of predictors is used.
- (ii) **Do not prune**.

Step 3. Obtain the prediction using the constructed regression tree, denoted it as $\hat{y}^{(1)}$.[24]

[24]For a given observation, the average of the tree-by-tree predicted values is computed using only the predicted values from trees in which that observation was **not** used to grow the tree.

Step 4. Repeat Steps 1 to 3 for B times and collect $\{\hat{y}^{(1)}, \hat{y}^{(2)}, ..., \hat{y}^{(B)}\}$.

Step 5. Compute the final prediction

$$\hat{y}_{\text{RF}} = \frac{1}{B}\sum_{b=1}^{B} \hat{y}^{(b)}. \tag{4.1}$$

Although we can determine the number of predictors that we actually use in the random selection procedure of Step 2(i), it is customary to use approximately **one third** of the total number of predictors, which makes random forest more computationally efficient than bagging.

We apply `randomForest` function to the movie dataset. This time, we do not assign value to `mtry` and set $B = 100$ as usual. R codes and results are presented below. Note that due to randomness, MSE and Var explained varies.

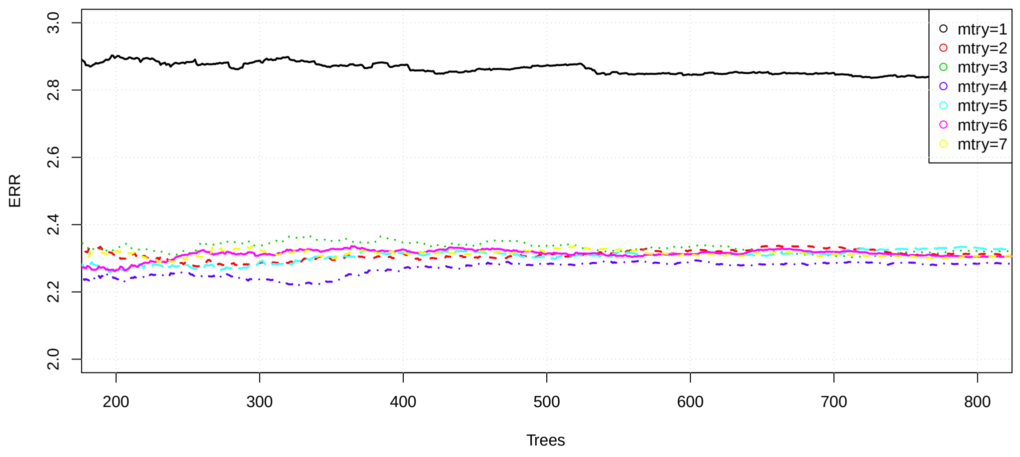
```
# 1. comparing BAG with RF
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
B = 100
out1 <- randomForest(OpenBox~.,data=dat, ntree=B)
out1

> Call:
 randomForest(formula = OpenBox ~ ., data = dat, ntree = 100)
               Type of random forest: regression
                     Number of trees: 100
No. of variables tried at each split: 2

          Mean of squared residuals: 2.436304
                    % Var explained: 28.3
```

Number of variables tried at each split is fixed at 2. As usual, we can manipulate this value through the feature `mtry`. We consider `mtry` values from 1 to 7. Note that when `mtry = 7`, the results are **identical to bagging tree**. For each value of `mtry`, we generate a random forest of 1000 trees. We plot their OOB MSEs against the number of trees and zoom in the $[200, 800]$ interval. R codes are presented below:

Figure 4.1: RF OOB MSEs for Different Values of `mtry`



```
# 2. compare RF with various mtry
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
B = 1000
k = ncol(dat)-1
ERR = matrix(NA,B,k)
for (i in 1:k){
  out <- randomForest(OpenBox˜.,data=dat, mtry=i, ntree=B)
  ERR[,i] = out$mse
}
matplot(ERR, type = "l",pch=1:k,col=1:k,lwd=2,xlab="Trees",
        xlim=c(200,800),ylim=c(2,3))
legend("topright",col=1:k,legend=c("mtry=1","mtry=2","mtry=3",
        "mtry=4","mtry=5","mtry=6","mtry=7"),pch=1)
grid()
```

Results are presented in Figure 4.1. RF with single predictor is noticeably worse than all others. Bagging tree which corresponds to `mtry = 7` does not dominate all others. Among all values of `mtry`, it seems that `mtry = 4` has the best performance.

4.2 Random Forest Prediction Exercise

We revisit the prediction exercise in Section 3.2. We use the first 80 observations as training set and compute the MSFE on the rest evaluation set. We compare the MSFEs by random forest, bagging tree, and regression tree. R codes are presented below.

```
# 3. RF prediction exercises
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(rpart)
k = ncol(dat)-1
# 3.1 training
data_train = dat[1:TAR,]
out1 <- rpart(OpenBox~.,data=data_train, method="anova")
out2 <- randomForest(OpenBox~.,data=dat, mtry=k, ntree=100)
out3 <- randomForest(OpenBox~.,data=dat, mtry=4, ntree=100)
# 3.2 evaluate prediction
INPUT <- dat[(TAR+1):nrow(dat),2:ncol(dat)]
RT = predict(out1,INPUT,'vector')
BAG = predict(out2,INPUT)
RF = predict(out3,INPUT)
# 3.3 print results
yf = data.frame(RT=as.numeric(RT),BAG,RF)
y0 = dat[(TAR+1):nrow(dat),1]
apply((yf-y0)^2,2,mean)
```

Results show that random forest in general has **worse** forecasting accuracy than the rest. Besides learning from bootstrap samples, random forest random drops variables at splits hence may cause further uncertainty.

Note that random forest works particularly well when the number of predictors is large. Comparing to bagging tree, random forest always has better computational efficiency since not all the predictors are included in the estimation process. Moreover, when the number of predictors is large, there is higher change that many of the predictors are redundant and excluding them can improve forecasting accuracy.

We now consider a simple **Monte Carlo** simulation to compare the general performance of random forest comparing with bagging and regression tree.

(i) Let the DGP be $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + u$.

(ii) $\boldsymbol{X}_1$ is a vector of ones and $\boldsymbol{X}_i \sim \mathrm{N}(0,1)$ for $i = 2, ..., k$.

(iii) The error term $\boldsymbol{u}$ also follows standard normal distribution.

(iv) We set $k = 10$ and the coefficient $\beta_i = 0.1/i$ for $i = 1, ..., k$. That is the coefficient gets smaller for larger i and the corresponding predictor becomes less significant.

(v) We set $n = 100$ and let the first 90 observations as training set and the last 10 observations as evaluation set.

(vi) We follow the exercise in Section 3.2 and compute the MSFEs by random forest, bagging tree, and regression tree.

(vii) We repeat the whole process 100 times and report the average MSFEs by the three estimators.

R codes are presented below. Because some of the predictors are really weak (especially those with larger i), random forest has better forecasting accuracy by yielding smaller MSFE in general. Moreover, both random forest and bagging tree have significantly better accuracy comparing to regression tree. This results imply that learning process is very useful and considering model (predictor) uncertainty can make the results even better.

Of course, random forest has its advantage when k is large and some predictors are highly insignificant, like the simulation design above. If the all the predictors are highly significant, then randomly discard 2/3 of the predictors can be troublesome. We modify the design step (iv) while keeping all other settings the same:

(iv) We set $k = 3$ and the coefficient $\beta_i = 10/i$ for $i = 1, ..., k$.

In this setting, all the coefficients are large which makes the predictors significant.

```
# 4. RF simulation
library(rpart)
#install.packages(tictoc)
library(tictoc)
tic()
# 4.1 nested function
MSFE <- function(yt,xt,ye,xe,B){
  dat = data.frame(yt,xt);
  out1 <- rpart(yt~.,dat,method="anova")
  out2 <- randomForest(yt~.,dat,mtry=ncol(xt),ntree=B)
  out3 <- randomForest(yt~.,dat,ntree=B)
  RT = predict(out1,xe,'vector')
  BAG = predict(out2,xe)
  RF = predict(out3,xe)
  yf = data.frame(RT=as.numeric(RT),BAG,RF)
  R = as.numeric(apply((yf-ye)^2,2,mean))}
# 4.2 parameter setup
n = 100
nt = 90
B = 100
k = 10
b = as.matrix(0.1*(1:k)^(-1))
# 4.3 simulation
R = matrix(NA,B,3)
for (i in 1:B){
  # generate data
  x = matrix(NA,n,k)
  x[,1] = rep(1,n)
  x[,2:k] = matrix(rnorm(n*(k-1)),nrow=n)
  u = as.matrix(rnorm(n))
  y = x%*%b+u
  yt = y[1:nt,]
  xt = x[1:nt,]
  ye = y[(nt+1):n,]
  xe = data.frame(x[(nt+1):n,])
  # compute MSE
  R[i,] = MSFE(yt,xt,ye,xe,100)}
# 4.4 print results
R = as.data.frame(R)
names(R) = c("RT","BAG","RF")
apply(R,2,mean)
toc()
```

We rerun the simulation and find out that not only random forest performs worse than bagging tree, it is also much worse than regression tree. Discarding 2/3 of important predictors can be problematic. The results implies that we need to be cautious while using the random forest method in practice.

4.3 Parallel Estimation

In the previous exercises, we included a library named `tictoc` to record total computational time. Note that you may need to install its package first. We considered $B = 100$ repetition and it costs us about **4 seconds** to complete. In practice, setting B to such a small number may not be robust. Imagine how much time do we need if we set $B = 1000$?

In theory, the total time should increase 10 times. In reality, this may be longer or shorter depending on the situation. Apparently, each repetition is **independent** from each other. Therefore, it would not be harmful if we run each repetition separately (by different CPU cores or even different machines).

Parallel computing is a type of computation in which many calculations or the execution of processes are carried out **simultaneously**. There are many forms of parallel computing, however, it has become the dominant paradigm in computer architecture, mainly in the form of **multi-core processors**.

We mainly focus on **multi-CPU-core** computing. In R, there are many packages that you can use, `parallel`, `snow`, `foreach`, etc. The package we will be using in this section is called `doParallel` which can be seen as a parallel backend for the `foreach` package.[25]

We first active the `doParallel` library. We can obtain the total number of available CPU logical cores using `detectCores()`.[26] This gives you a sense of how much computational power that is available to you using your current station. Once you know your total available cores, you can specify a cluster using the `makeCluster` command. [27] Once you assemble your team of cores, you register it in R by using the `registerDoParallel()` command. We can get the name of our current parallel estimation module using `getDoParName()`. It should be `snow` for PCs and `multicore` for MACs. R codes are presented below.

```
library(doParallel)
detectCores()
cl <- makeCluster(10)
registerDoParallel(cl)
getDoParName()
```

[25]Note that since the infrastructure of each OS (Windows, iOS, Linux, Unix, etc.) is different from one another, the ways (commands, packages, syntax,...) of parallel computing in R under different operating system can be quite different.

[26] **Physical cores** are number of physical cores, actual hardware components. **Logical cores** are the number of physical cores times the number of **threads** that can run on each core through the use of hyperthreading. For example, my 6-core processor runs two threads per core, so I have 12 logical processors.

[27]Note that you may not want to use all your available cores in your system to prevent R from freezing.

Now the cluster is ready. We start with the following easy demonstration. We generate a series of square root for $i = 1, ..., B$ with $B = 1000$. The loop command we use is not `for` but `foreach` as in for **each** of the workers in the cluster. There are three major syntax differences besides the name of the command:

(i) Within the `foreach` loop, it is "`i=1:B`" not "`i in 1:B`" as in `for`.

(ii) We need the command `%dopar%` between ")" and "{".

(iii) To save the results from `foreach`, we need to assign a variable to `foreach` **outside** the loop. Calculation within loop are not saved.

```
# 5.1 a simple demonstration
B = 1000
r <- foreach(i=1:B) %dopar% {
  sqrt(i)
}
stopCluster(cl)
```

You may also want to close the cluster by `stopCluster()` to save resources.

The above exercise is just a demonstration. In fact, it is so simple that we don't really need parallel computing. Let revisit the random forest simulation exercise with $B = 1000$. We apply parallel computing to it and see how much we can improve the computational efficiency.

Note that we can still use the nested function `MSFE` in the parallel estimation. However, **each worker in the cluster needs to be notified about specific libraries**. Note that within `foreach` loop, we need to state `library(rpart)` and `library(randomForest)` for completeness. Also, don't forget to `stopCluster(cl)` in order to free memory.

R codes are presented below. It is clear that computational efficiency can be improved significantly using parallel estimation. The more workers (cores) you included in the cluster the faster the codes run. Try different values of workers and make comparison.

```
# 6. PARALLEL estimation: RF simulation
library(doParallel)
cl <- makeCluster(12)
registerDoParallel(cl)
library(rpart)
library(tictoc)
library(randomForest)
tic()
MSFE <- function(yt,xt,ye,xe,B){
  dat = data.frame(yt,xt);
  out1 <- rpart(yt~.,dat,method="anova")
  out2 <- randomForest(yt~.,dat,mtry=ncol(xt),ntree=B)
  out3 <- randomForest(yt~.,dat,ntree=B)
  RT = predict(out1,xe,'vector')
  BAG = predict(out2,xe)
  RF = predict(out3,xe)
  yf = data.frame(RT=as.numeric(RT),BAG,RF)
  R = as.numeric(apply((yf-ye)^2,2,mean))}
n = 100
nt = 90
B = 1000
k = 10
b = as.matrix(10*(1:k)^(-1))
R <- foreach(i=1:B) %dopar% {
  library(rpart)    # library must be noted for each worker
  library(randomForest)
  x = matrix(NA,n,k)
  x[,1] = rep(1,n)
  x[,2:k] = matrix(rnorm(n*(k-1)),nrow=n)
  u = as.matrix(rnorm(n))
  y = x%*%b+u
  yt = y[1:nt,]
  xt = x[1:nt,]
  ye = y[(nt+1):n,]
  xe = data.frame(x[(nt+1):n,])
  MSFE(yt,xt,ye,xe,100)}
R <- data.frame(matrix(unlist(R), nrow=length(R),
                       byrow=T))
names(R) = c("RT","BAG","RF")
apply(R,2,mean)
toc()
stopCluster(cl)
```

4.4 Random Forest OOB Error and R-square

Similar to bagging tree, we can use OOB error to estimate variable importance and compute R^2. We still use the command `randomForest`. Note that the bagging tree we studied in the previous section is only a special case of random forest, with number of variables at split setting at full. For random forest, we can manipulate this number by setting different values on `mtry`. If we don't set this value, `randomForest` will randomly use 1/3 of all the variables.

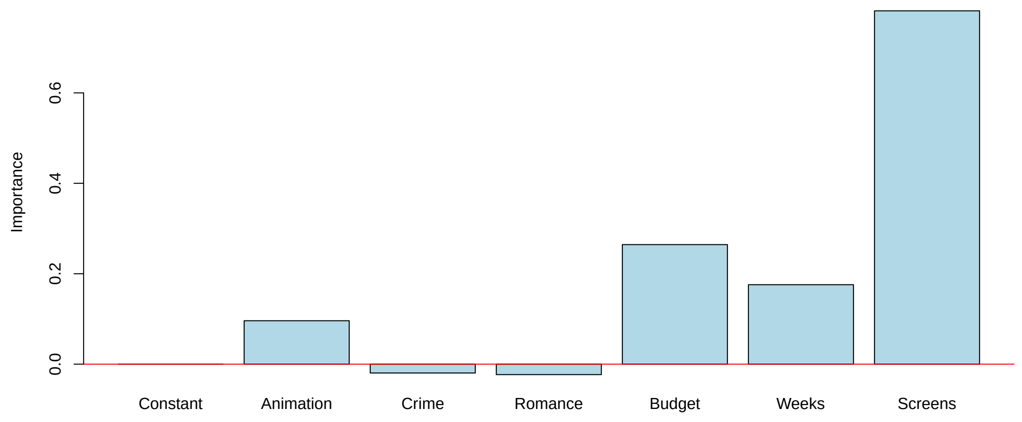
R codes are presented below and predictor importance is shown in Figure 4.2. As we can see, predictor importance by random forest is quite similar to those by bagging tree. `Budget`, `Weeks`, and `Screens` are much more important than the other predictors. Among all, predictor `Screens` is the most important. The estimated R^2 should be approximately 0.75 which is higher than regression tree.

```
# 7. OOB and variable importance
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(randomForest)
out <- randomForest(OpenBox~.,data=dat, ntree=100,
                    importance=T)
# 7.1 plot results
RF = out$importance[,1]
barplot(RF,ylab="Importance",col="lightblue")
lines(c(-3,10),c(0,0),col="red",lwd=1)
# 7.2 estimate Rsquare
y = dat$OpenBox
INPUT <- dat[,2:ncol(dat)]
yf <- predict(out,INPUT)
e = y-yf
1-sum(e^2)/sum((y-mean(y))^2)
```

We can play with `mtry`. Set it from 1 to 7. For **simplicity**, we reconstruct the estimation of R^2 into a nested function, called it `RSQUARE` Observe the changes in centered R-square. The two input variables are the data `dat` and the value of `mtry`. We set `mtry` in a loop from 1 to 7 (bagging). Then, compute the centered R^2s, which are stored in the matrix `s`. R codes are presented below. The pattern is straightforward, as the value of `mtry` increases, the estimated R^2 also increases.

Note that you may not get the same pattern, since B is set to a small number. Try larger values of B, see it the pattern becomes stable.

Figure 4.2: Predictor Importance by Random Forest



```
# 8. RF R^2, play with mtry
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(randomForest)
# 8.1 define the Rsquare function
RSQUARE <- function(dat,k){
  out <- randomForest(OpenBox~.,mtry = k, data=dat, ntree=100,
                      importance=T)
  y = dat$OpenBox
  INPUT <- dat[,2:ncol(dat)]
  yf <- predict(out,INPUT)
  e = y-yf
  R = 1-sum(e^2)/sum((y-mean(y))^2)
}
# 8.2 estimate and present results
s = matrix(NA,nrow = ncol(dat)-1,ncol = 1)
for (i in 1:(ncol(dat)-1)){
  s[i,] = RSQUARE(dat,i)
}
s
```

4.5 Quantile Prediction

All previous predictions focus on **mean prediction**. That is we estimate the condition mean of the response variable given certain values of the predictors. In practice, especially in risk management, we are more interested in predictions that happen under extreme situation (tail event) other than the conditional mean, which happens most likely.

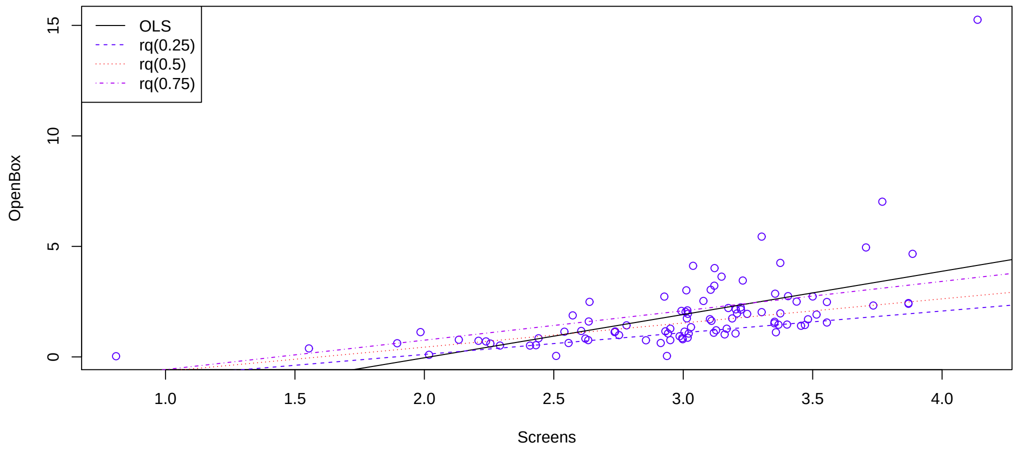
In statistics and probability **quantiles** are cut points dividing the range of a probability distribution into continuous intervals with equal probabilities, or dividing the observations in a sample in the same way. q-quantiles are values that partition a finite set of values into q subsets of (nearly) equal sizes. Quantiles can also be applied to continuous distributions, providing a way to generalize rank statistics to continuous variables. When the cumulative distribution function of a random variable is known, the q-quantiles are the application of the quantile function (the inverse function of the cumulative distribution function) to the values $\{1/q, 2/q, ..., (q-1)/q\}$.

Quantile prediction aims at estimating either the conditional **median** or other **quantiles** of the response variable. Essentially, quantile prediction is the extension of mean prediction. We start with a simple exercise with linear regression model. We use the most important predictor `Screens` as the sole predictor of open box office. We predict the mean (by OLS), 25%, 50%, and 75% quartiles. We plot all the estimated results in one figure. We need to load the `quantreg` library.[28]. R codes are presented below and the figure is plotted in Figure 4.3.

```
# 9. quantile regression - linear
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# install.packages("quantreg")
library(quantreg)
# 9.1 estimation
rq1 <- rq(OpenBox ~ Screens, data = dat, tau = 0.25)
rq2 <- rq(OpenBox ~ Screens, data = dat, tau = 0.5)
rq3 <- rq(OpenBox ~ Screens, data = dat, tau = 0.75)
lmfit <- lm(OpenBox ~ Screens, data = dat)
# 9.2 plot results
plot(OpenBox ~ Screens, data = dat, pch = 1,col="blue")
abline(lmfit,col="black")
abline(rq1,col="blue",lty=2)
abline(rq2,col="red",lty=3)
abline(rq3,col="purple",lty=4)
legend("topleft", legend = c("OLS","rq(0.25)","rq(0.5)","rq(0.75)"),
       col = c("black","blue","red","purple"), lty = c(1:4))
```

[28]You may need to install this package first

Figure 4.3: Quantile Prediction Results - Linear Model



The syntax of `quantreg` is very similar to `lm` except for the `tau` feature, which measures the quantile. By default `quantreg` focuses on the median ($\tau = 0.5$). The circles in Figure 4.3 correspond to the actual data for the `screens` and `OpenBox` pair. The black solid line represents OLS, the dashed, dotted, and dash-dotted lines correspond to linear quantile regression with $\tau = 0.25, 0.5$, and 0.75, respectively. Not a surprise that lower quantile results are associated with lower value of `OpenBox`. Moreover, medium estimation results are quite **different** from mean, indicating that the empirical distribution of the data is **asymmetric**.

4.5.1 Quantile Random Forest Prediction

Recently, statisticians brought quantile estimation to random forest (and bagging). The basic intuition is the same with one difference: random forest is nonlinear. In R, we can use the `quantregForest` package. Again, you may need to **install** this package before using it.

The training process using `quantregForest` is **not identical** to `randomForest`. The conventional regression type syntax: `(y∼x1+x2)` **does not** work under `quantregForest`. You must properly define the **input matrix**[29] and the **response variable** and put them in in sequence. See `?quantregForest` for more information.

R codes are presented below.

[29]That is right, **matrix**. A single vector does not work, as `quantregForest` does not addin intercept automatically. As in our example, we need to include the constant term manually.

```
# 9. quantile regression - RF
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# 9.1 training
#install.packages("quantregForest")
library(quantreg)
library(quantregForest)
xt = dat[,c(2,8)]
yt = dat[,1]
qrf <- quantregForest(xt,yt,ntree=100)
```

Once we have trained a proper quantile regression random forest. We can predict for different quantiles given input data. For an easy illustration, we consider ten equally-spaced `Screens` between the minimum and maximum in-sample values. Then, we predict conditional mean responses and conditional 25%, 50%, and 75% quartiles. R codes are presented below and results are plotted in Figure 4.4.

```
# 9.2 quantile prediction
x = seq(0.5,4.5,0.5)
tmp = rep(1,length(x))
INPUT = data.frame(Constant=tmp,Screens=x)
rf1 <- predict(qrf,INPUT,what=0.25)
rf2 <- predict(qrf,INPUT,what=0.5)
rf3 <- predict(qrf,INPUT,what=0.75)
# 9.3 plot results with linear qr
linear <- rq(OpenBox~Screens,data=dat)
plot(OpenBox ~ Screens, data = dat, pch = 1,col="blue")
abline(linear,col="black")
lines(x,rf1,col="blue",lty=2)
lines(x,rf2,col="red",lty=3)
lines(x,rf3,col="purple",lty=4)
legend("topleft", legend = c("Linear","qrf(0.25)","qrf(0.5)",
                             "qrf(0.75)"),
       col = c("black","blue","red","purple"), lty = c(1:4))
```

The 25% to 75% prediction intervals for different values of `Screens` vary significantly. Moreover, random forest predictions are clearly nonlinear, which contrasts to the results of linear quantile prediction. We can conduct a more robust estimation, by changing the number of trees to a 1000 and set the increment of `Screens` input value to 0.1. Results are plotted in Figure 4.5. Notice the big differences at the **upper quantile** of `Screens`. This implies the importance of robustness check.

Figure 4.4: Quantile Prediction Results - RF

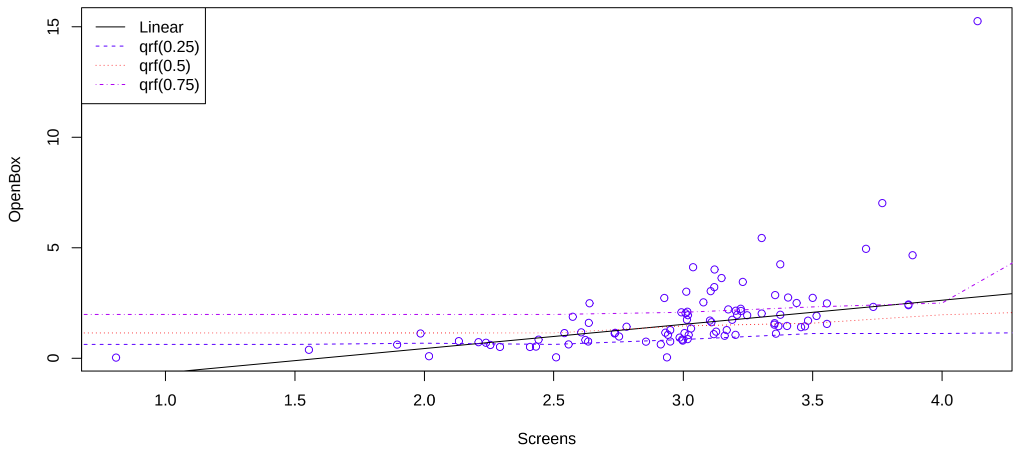
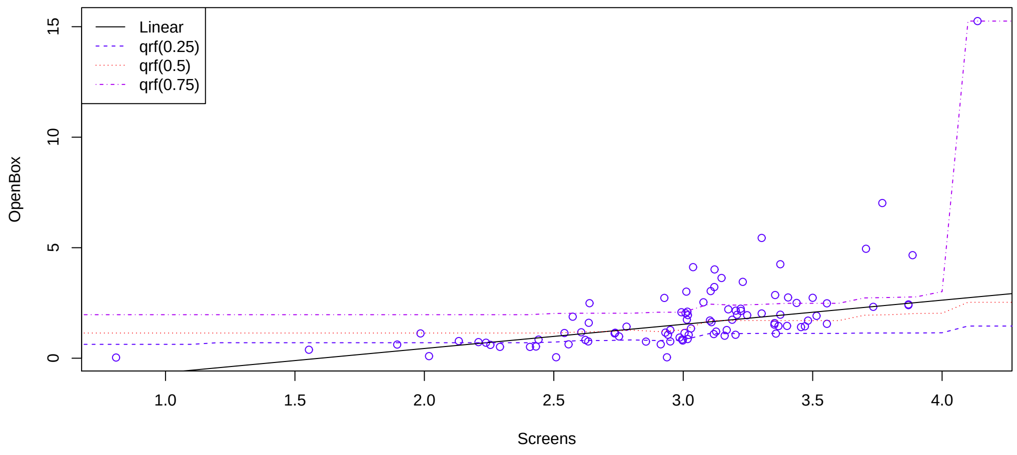


Figure 4.5: Quantile Prediction Results - RF Robust



5 Boosting Tree and Support Vector Machine

5.1 Boosting Tree

One of the reasons why random forest is so effective for a complex $f(x)$ is that it capitalizes interpolation. As a result, it can respond to highly **local features** of the data in a robust manner. Such flexibility is desirable because it can substantially reduce the bias in fitted values. But the flexibility usually comes at a price: the risk of **overfitting**. Random forests consciously addresses overfitting using OOB observations to construct the fitted values and measures of fit, and by averaging over trees. The former provides ready-made test data while the latter is a form of regularization. Experience to date suggests that this two-part strategy can be highly effective.

But the two-part strategy, broadly conceived, can be implemented in other ways. Some argue that an alternative method to accommodate highly local features of the data is to give the observations **responsible for the local variation more weight** in the fitting process. If in the binary case, for example, a fitting function misclassifies those observations, that function can be applied again, but with extra weight given to the observations misclassified. Then, after a large number of fitting attempts, each with difficult-to-classify observations given relatively more weight, overfitting can be reduced if the fitted values from the different fitting attempts are averaged in a sensible fashion.

Ideas such as these lead to very powerful statistical learning procedures that can compete with random forests. These procedures are called **boosting**. There are many boosting algorithms. In the textbook, the `AdaBoost` is discussed in length. Note that `Adaboost` is suitable for classification, not for quantitative response variable. In this section, we mainly focus on the **gradient boost algorithm**.

5.1.1 Forward Stagewise Boosting

Boosting is a way of fitting an additive expansion in a set of elementary basis functions. More generally, basis function expansions take the form

$$f(\boldsymbol{X}) = \sum_{m=1}^{M} \beta_m b(\boldsymbol{X}; \gamma_m),$$

where β_m, $m = 1, 2, ..., M$ are the expansion coefficients, and $b(\boldsymbol{X}; \gamma) \in \mathbb{R}$ are usually simple functions of the multivariate argument $\boldsymbol{X}$, characterized by a set of parameters $\boldsymbol{\gamma}$. Typically these models are fit by **minimizing a loss function** averaged over the training data, such as the squared-error,

$$\min_{\{\beta_m, \gamma_m\}_1^M} \sum_{i=1}^{n} L\left(y_i, \sum_{m=1}^{M} \beta_m b(\boldsymbol{X}_i; \gamma_m)\right). \tag{5.1}$$

Forward stagewise modeling approximates the solution to (5.1) by sequentially adding new basis functions to the expansion without adjusting the parameters and coefficients of those that have already been added. This is outlined in the following algorithm:

Algorithm 5.1: Forward Stagewise Boosting

```
1. Initialize $f_0(\boldsymbol{X}) = 0$.
2. For $m = 1$ to $M$:
    (a) Compute
        $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{n} L\big(y_i, f_{m-1}(\boldsymbol{X}_i) + \beta b(\boldsymbol{X}_i; \gamma)\big).$$
    (b) Set $f_m(\boldsymbol{X}) = f_{m-1}(\boldsymbol{X}) + \beta_m b(\boldsymbol{X}; \gamma_m)$.
```

At each iteration m, one solves for the optimal basis function $b(\boldsymbol{X}; \gamma_m)$ and corresponding coefficient β_m to add to the current expansion $f_{m-1}(\boldsymbol{X})$. This produces $f_m(\boldsymbol{X})$, and the process is repeated. Previously added terms are not modified.

For squared-error loss

$$L\big(y, f(\boldsymbol{X})\big) = \big(y - f(\boldsymbol{X})\big)^2, \tag{5.2}$$

one has

$$\begin{aligned} L\big(y_i, f_{m-1}(\boldsymbol{X}_i) + \beta b(\boldsymbol{X}_i, \gamma)\big) &= \big(y_i - f_{m-1}(\boldsymbol{X}_i) - \beta b(\boldsymbol{X}_i; \gamma)\big)^2 \\ &= \big(r_{im} - \beta b(\boldsymbol{X}_i; \gamma)\big)^2, \end{aligned}$$

where $r_{im} = y_i - f_{m-1}(\boldsymbol{X}_i)$ is simply the residual of the current model on the i^{th} observation. Thus, for squared-error loss, the term $\beta_m b(\boldsymbol{X}_i; \gamma_m)$ that best fits the current residuals is added to the expansion at each step, which has the sense of **least squares** estimation.

5.1.2 Gradient Tree Boosting

Regression trees partition the space of all joint predictor variable values into disjoint regions R_j, $j = 1, 2, ..., J$, as represented by the terminal nodes of the tree. A constant j is assigned to each such region and the predictive rule is

$$\boldsymbol{X} \in R_j \Rightarrow f(\boldsymbol{X}) = \gamma_j.$$

Thus a tree can be formally expressed as

$$T(\boldsymbol{X}, \Theta) = \sum_{j=1}^{J} \gamma_j \mathbb{I}(\boldsymbol{X} \in R_j),$$

with parameters $\Theta = \{R_j, \gamma_j\}_{j=1}^J$. The parameters are found by minimizing the risk

$$\hat{\Theta} = \arg\min_{\Theta} \sum_{j=1}^{J} \sum_{\boldsymbol{X}_i \in R_j} L(y_i, \gamma_j).$$

The **boosted tree** model is a sum of all trees,

$$f_M(\boldsymbol{X}) = \sum_{m=1}^{M} T(\boldsymbol{X}; \Theta_m)$$

induced in a forward stagewise manner (Algorithm 5.1). At each step in the forward stagewise procedure one must solve

$$\hat{\Theta}_m = \arg\min_{\Theta_m} \sum_{i=1}^{n} L\big(y_i, f_{m-1}(\boldsymbol{X}_i) + T(\boldsymbol{X}_i; \Theta_m)\big). \tag{5.3}$$

for the region set and constants $\Theta_m = \{R_{jm}, \gamma_{jm}\}_1^{J_m}$ of the next tree, given the current model $f_{m-1}(\boldsymbol{X})$.

Given the regions R_{jm}, finding the optimal constants γ_{jm} in each region is typically straightforward:

$$\hat{\gamma}_{jm} = \arg\min_{\Theta_m} \sum_{i=1}^{n} L\big(y_i, f_{m-1}(\boldsymbol{X}_i) + \gamma_{jm}\big).$$

Finding the regions is difficult, and even more difficult than for a single tree. For squared-error loss, however, the solution is quite straightforward. It is simply the regression tree that **best predicts the current residuals** $y_i - f_{m-1}(\boldsymbol{X}_i)$, and $\hat{\gamma}_{jm}$ is the mean of these residuals in each corresponding region.

Fast approximate algorithms for solving (5.3) with any **differentiable loss criterion** can be derived by analogy to numerical optimization.

$$\hat{\boldsymbol{f}} = \arg\min_{\boldsymbol{f}} L(\boldsymbol{f}),$$

where the "parameters" $\boldsymbol{f} \in \mathbb{R}^n$ are the values of the approximating function $f(\boldsymbol{X}_i)$ at each of the n data points $\boldsymbol{X}_i$. Its solution is a sum of component vectors

$$\boldsymbol{f}_M = \sum_{m=0}^{M} \boldsymbol{h}_m, \qquad \boldsymbol{h}_m \in \mathbb{R}^n,$$

where $\boldsymbol{f}_0 = \boldsymbol{h}_0$ is the initial guess, and each successive $\boldsymbol{f}_m$ is induced based on the current parameter vector $\boldsymbol{f}_{m-1}$. Each **increment vector** $\boldsymbol{h}_m = -\rho_m \boldsymbol{r}_m$, where ρ_m is a scalar and $\boldsymbol{r}_m \in \mathbb{R}^n$ is the gradient of $L(\boldsymbol{f})$ evaluated at $\boldsymbol{f} = \boldsymbol{f}_{m-1}$. The components of the **gradient**

$\boldsymbol{r}_m$ are

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

The **step length** ρ_m is the solution to

$$\rho_m = \arg\min_{\rho} L(\boldsymbol{f}_{m-1} - \rho \boldsymbol{r}_m).$$

The current solution is then updated

$$\boldsymbol{f}_m = \boldsymbol{f}_{m-1} - \rho_m \boldsymbol{r}_m$$

and the process repeated at the next iteration.

The following Algorithm 5.2 presents the generic gradient tree-boosting algorithm for regression. Specific algorithms are obtained by inserting different loss criteria $L(\boldsymbol{y}, f(\boldsymbol{X}))$. The first line of the algorithm initializes to the optimal constant model, which is just a single terminal node tree. The components of the negative gradient computed at line 2(a) are referred to as generalized or pseudo residuals, r.

Algorithm 5.2: Gradient Tree Boosting Algorithm

```
1. Initialize $f_0(\boldsymbol{X}) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$.
2. For $m = 1$ to $M$:
    (a) For $i = 1, 2, ..., n$ compute
            $r_{im} = -\left[\frac{\partial L(y_i, f(\boldsymbol{X}_i))}{\partial f(\boldsymbol{X}_i)}\right]_{f=f_{m-1}}.$
    (b) Fit a regression tree to the targets $g_{im}$ giving terminal regions $R_{jm}$,
        $j = 1, 2, ..., J_m$.
    (c) For $j = 1, 2, ..., J_m$ compute
            $\hat{\gamma}_{jm} = \arg\min_{\gamma} \sum_{\boldsymbol{X}_i \in R_{jm}} L(y_i, f_{m-1}(\boldsymbol{X}_i) + \gamma).$
    (d) Update $f_m(\boldsymbol{X}) = f_{m-1}(\boldsymbol{X}) + \sum_{j=1}^{J_m} \hat{\gamma}_{jm} \mathbb{I}(\boldsymbol{X} \in R_{jm})$.
3. Output $\hat{f}(\boldsymbol{X}) = f_M(\boldsymbol{X})$.
```

5.1.3 Gradient Boosting in R

There are many packages in R that are capable of performing gradient boosting. The most famous (classic) one is perhaps the `bgm` package. In this section, we study another package named `xgboost` which is suppose to be more computationally efficient than all other existing gradient boosting implementations. Again, you may need to install the package before using it.

The `xgboost` can conduct linear-model based and tree based boosting. Its default booster is tree based, which is also our focus. The syntax in `xgboost` is also different from other packages (what a surprise...). We now apply `xgboost` to the movie data. The `xgboost` command takes **only matrix-class inputs**. That means, we need to convert our dataframe format to matrix first using the built-in `as.matrix` command. The R codes are presented below

```
# 1. xgboost application
# install.packages("xgboost")
library(xgboost)
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# 1.1 define data
x = as.matrix(dat[,2:ncol(dat)])
y = as.matrix(dat[,1])
```

Once the data are prepared, we can apply them to `xgboost`. The number of trees in `xgboost` is controlled by a different feature `nrounds` now. We set this number to $B = 25$ learning cycles and name the output as `xgb`.[30] By default `xgboost` will demonstrate the convergence of train-rmse (root mean squared error[31]). We plot this number against B in Figure 5.3.

The R codes are presented below:

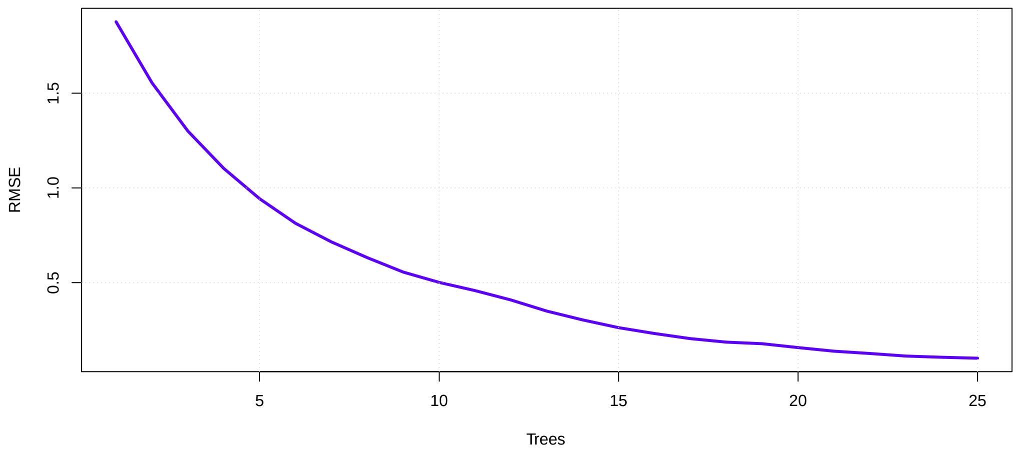
```
# 1.2 train
B = 25
BT <- xgboost(x,y,nrounds=B)
RMSE = BT$evaluation_log$train_rmse
plot(1:B,RMSE,type="l",lwd=3,col="blue",xlab="Trees")
grid()
```

Note a surprise that as the number of trees increases, RMSE decreases. That is, the model fit improves as we iterate more (include more trees). Try $B = 1000$ and plot the figure again. What do you observe? What do you conclude?

[30] Note that you don't want to set B too high to avoid overfitting.

[31] What is this value? Can you replicate the last number?

Figure 5.1: RMSE against Number of Trees



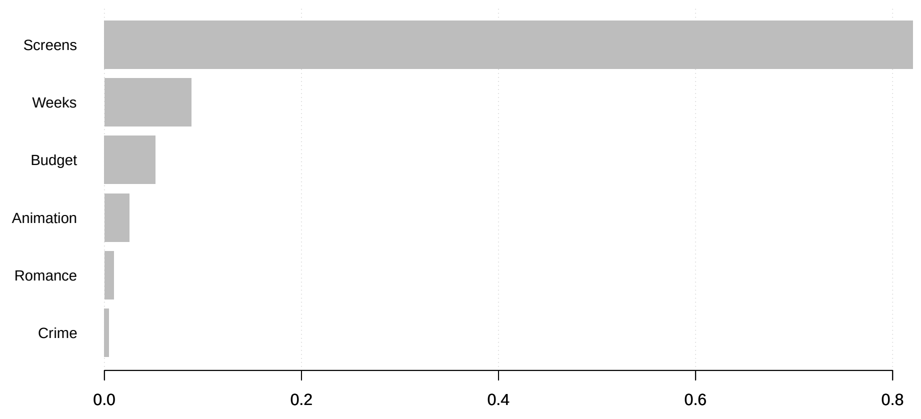
In the next exercise, we predict using boosting. We reconsider the hypothetical input data $\boldsymbol{X}_p = [1, 0, 0, 1, 5, 4, 3.5]$ and apply it to the trained boosting model BT. Note that we need to convert the input data to matrix type. We also compare the result with the prediction of regression tree. R codes are presented below:

```
# 1.3 prediction
INPUT <- data.frame(Constant=1,Animation=0,Crime=0,Romance=1,
                    Budget=5,Weeks=4,Screens=3.5)
predict(BT, as.matrix(INPUT))
library(rpart)
predict(rpart(OpenBox~.,data=dat, method="anova"),
        INPUT,type="vector")
```

Finally, we can use xgboost to calculate **variable importance**. Since there is no OOB observations in boosting, we do not rank variables using the method developed in BAG or RF. The variable ranking in boosting is rather similar to the one in regression tree. We first calculate variable importance for a single decision tree by the amount that each attribute split point improves the performance measure, weighted by the number of observations the node is responsible for. We average this variable importance across all the trees and report the fraction that each variable takes as the final output value. Therefore, the summation of variable importance in xgboost equals 1. We show variable importance using the xgb.importance command. We can also plot this value using the xgb.plot.importance command. R codes are presented below.

```
# 1.4 importance
IMP = xgb.importance(model=BT)
IMP
xgb.plot.importance(IMP)
```

Figure 5.2: RMSE against Number of Trees



Obviously, there is no negative number. `Screens` is still the most important variable. `Weeks` is more important than `Budget`. All the genres have marginal contribution.

5.2 Support Vector Machine

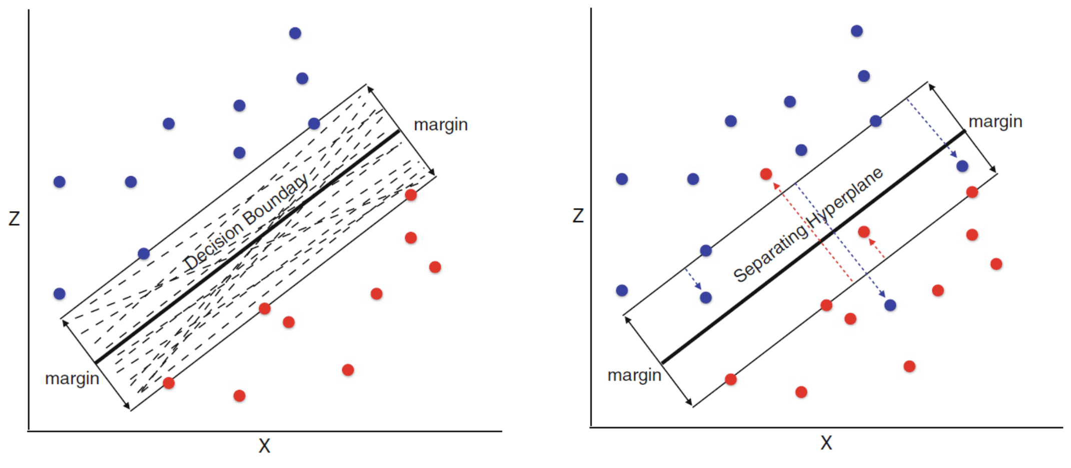
In machine learning, support vector machines (SVM) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. The theory behind SVM is due to Vapnik and is described in Vapnik (1996). A SVM constructs an optimal **hyperplane** as a **decision surface** such that the margin of separation between the two classes in the data is maximized. **Support vectors** refer to a small subset of the training observations that are used as support for the optimal location of the decision surface.

The classic SVM was designed for classification and a version of SVM for regression, later known as **support vector regression** (SVR), was proposed in by Drucker, Burges, Kaufman, Smola, and Vapnik (1996). The goal of SVR is to find a function $f(\boldsymbol{X}_t)$ that deviates from y_t by a value no greater than a predetermined ϵ for each observations $\boldsymbol{X}_t$, and at the same time is as flat as possible.

5.2.1 Basic Concept

Figure 5.3(a) shows a scatter plot. There are two predictors (X and Z) and a binary response Y, that can take on values of red or blue. In this figure, the blue circles and red circles are each located in quite different areas of the two-dimensional space defined by the predictors. In fact, a linear **separating hyperplane** could be drawn to produce separation between the two classes.

Figure 5.3: Basic Concept of SVM



(a) Separable Case (b) Nonseparable Case

Clearly, there is a limitless number of **linear decision boundaries** producing separation. These are represented by the dashed lines in Figure 5.3(a). The SVM seeks two parallel fences that maximize their perpendicular distance from the decision boundary. There can be only one straight line parallel to the fences and midway between them. That decision boundary is shown with the solid black line. Observations that fall in top of the fences are called **support vectors** because they directly determine where the fences will be located and hence, the optimal decision boundary. The distance between the decision boundary and either fence is called the **margin**.[32]

Cases that fall on one side of the decision boundary are labeled as one class, and *vice versa*. Subsequently, any new cases for which the outcome class is not known will be assigned the class determined by the side of the decision boundary on which they fall. The classification rule that follows from the decision boundary is called **hard thresholding**, and the decision boundary is often called the **separating hyperplane**. Sometimes the two fences are called the **margin boundary**.

Figure 5.3(b) shows a plot that is much like Figure 5.3(a), but the two sets of values are no longer linearly separable. One possible response is to **permit violations** of the buffer zone. One can specify some number of the observations that would be allowed to fall on

[32]Some may define the margin as the distance between the two fences (which amounts to the same thing). The wider the margin, the greater the separation between the two classes.

the wrong side of their margin boundary. These are called **slack variables**. But that is not quite enough. Some slack variables fall just across their margin boundary, and some fall far away. In response, the **distance** between the relevant fence and the location of the slack variable can be taken into account. The **sum of such distances** can be viewed as a measure of how permissive one has been when the margin is maximized. If one is more permissive by allowing for a larger sum, it may be possible to locate a separating hyperplane within a larger margin. A form of the bias-variance tradeoff reappears. It follows that the sum of the distances can be a tuning parameter when a SVM is applied to data. Fitting the SVM with slack variables is sometimes called **soft thresholding**.

5.2.2 SVR with Linear Kernel

We first consider the SVR for the linear regression model (**SVR$_L$**)

$$y_t = f(\boldsymbol{X}_t) + e_t = \boldsymbol{X}_t\boldsymbol{\beta} + e_t = \beta_0 + \tilde{\boldsymbol{X}}_t\boldsymbol{\beta}_1 + e_t,$$

where $\boldsymbol{X}_t = [1, \tilde{\boldsymbol{X}}_t]$ and $\boldsymbol{\beta} = [\beta_0, \boldsymbol{\beta}_1^\top]^\top$. We estimate $\boldsymbol{\beta}$ through the minimization of

$$H(\boldsymbol{\beta}) = \sum_{t=1}^{n} V_\epsilon\big(y_t - f(\boldsymbol{X}_t)\big) + \frac{\lambda}{2}\|\boldsymbol{\beta}_1\|^2, \tag{5.4}$$

where the loss function

$$V_\epsilon(r) = \begin{cases} 0 & \text{if } |r| < \epsilon \\ |r| - \epsilon & \text{otherwise} \end{cases}$$

is called an ϵ-**insensitive error measure** that ignores errors of size less than ϵ. As a part of the loss function V_ϵ, the parameter ϵ is usually predetermined. On the other hand, λ is a more traditional regularization parameter, that can be estimated by cross-validation.

Let $\hat{\boldsymbol{\beta}} = \left[\hat{\beta}_0, \hat{\boldsymbol{\beta}}_1^\top\right]^\top$ be the minimizers of function (5.4), the solution function can be shown to have the form

$$\begin{aligned} \hat{\boldsymbol{\beta}}_1 &= \sum_{t=1}^{n}(\hat{\alpha}_t^* - \hat{\alpha}_t)\tilde{\boldsymbol{X}}_t^\top, \\ \hat{f}(\boldsymbol{X}) &= \sum_{t=1}^{n}(\hat{\alpha}_t^* - \hat{\alpha}_t)\boldsymbol{X}\boldsymbol{X}_t^\top + \hat{\beta}_0\boldsymbol{\iota}_n, \end{aligned}$$

where $\boldsymbol{\iota}_n$ is an $n \times 1$ vector of ones and the parameters $\hat{\alpha}_t$ and $\hat{\alpha}_t^*$ are the nonnegative multiplier of the following Lagrangian equation

$$\min_{\hat{\alpha}_t, \hat{\alpha}_t^*} \epsilon\sum_{t=1}^{n}(\hat{\alpha}_t^* + \hat{\alpha}_t) - \sum_{t=1}^{n} y_i(\hat{\alpha}_t^* - \hat{\alpha}_t) + \frac{1}{2}\sum_{t=1}^{n}\sum_{t'=1}^{n}(\hat{\alpha}_t^* - \hat{\alpha}_t)(\hat{\alpha}_{t'}^* - \hat{\alpha}_{t'})\boldsymbol{X}_t\boldsymbol{X}_{t'}^\top$$

subject to the constraints $0 \leq \hat{\alpha}_t^*, \hat{\alpha}_t \leq 1/\lambda$, $\sum_{t=1}^{n}(\hat{\alpha}_t^* - \hat{\alpha}_t) = 0$, and $\hat{\alpha}_t\hat{\alpha}_t^* = 0$ for all

$t = 1, ..., n$. We usually called the non-zero values of $\hat{\alpha}_t^* - \hat{\alpha}_t$ for $t = 1, ..., n$ the support vector.

In R, we use the `kernlab` package to conduct SVM estimation. The main function we use is `ksvm`. The syntax is **NOT** conventional. Although we can bring in `dataframe` type data, `ksvm` cannot process the constant term, because this package **standardize** the data before processing it. Remember to remove the `Constant` term before applying the data to `ksvm`. There are many default values for the tuning parameters, which you need to pay attention to. Some of the most **important tuning parameters** are

(i) `kernel = rbfdot`, the default kernel is Gaussian

(ii) `epsilon = 0.1`, the default insensitive parameter is 0.1

(iii) `C = 1`, the default penalty coefficient is 1 (note that $C = 1/\lambda$ in our case)

We revisit the movie forecasting exercise. We apply the data to function `ksvm` under **linear** kernel, while keeping all other default values. R codes are presented below:

```
# 2. SVR application - Linear
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# 2.1 training
library(kernlab)
dat$Constant = NULL
SVR <- ksvm(OpenBox~.,data=dat,kernel="vanilladot")
SVR
```

Note that we need to remove the `Constant` variable and the estimated results show that the kernel we used is linear with $\epsilon = 0.1$ and $\lambda = 1/C = 1$. There is no randomness, you should obtain exactly the following results:

```
Support Vector Machine object of class "ksvm"

SV type: eps-svr  (regression)
 parameter : epsilon = 0.1  cost C = 1

Linear (vanilla) kernel function.

Number of Support Vectors : 71

Objective Function Value : -28.9148
Training error : 0.685817
```

What is that training error? RMSE? Can you replicate it?

This is not the conventional RMSE as in boosting or random forest. This is actually the RMSE of the **standardized data**. Try the following codes instead:

```
# replicate the training error
e = SVR@ymatrix-SVR@fitted
mean(e^2)
SVR@error
```

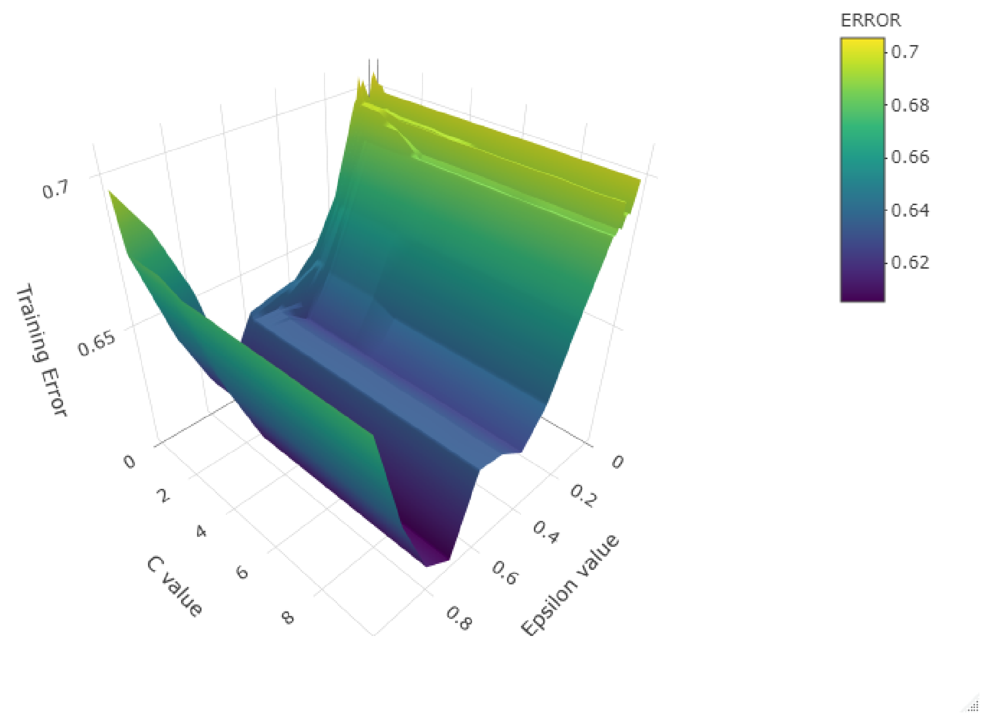
We can try various values of tuning parameters. Let consider a sequence of $C = 0.1, ..., 0.9, 1, ..., 10$ and a sequence of $\epsilon = 0.01, ..., 0.09, 0.1, 1$. We try every combination and evaluate their performance by the **model training error**. We plot the estimated errors against C and ϵ values in a surface plot. R codes are presented below. The results are plotted in Figure 5.4.

```
# 2.2 try different tuning parameters
CM=c(seq(0.1,0.9,0.1),seq(1,10,1))
EM=c(seq(0.01,0.09,0.01),seq(0.1,1,0.1))
hyper_grid <- expand.grid(CM=CM,EM=EM)
ERROR = matrix(NA,nrow=length(CM),ncol=length(EM))
for (i in 1:nrow(hyper_grid)) {
  cp <- hyper_grid$CM[i]
  eps <- hyper_grid$EM[i]
  SVRt <- ksvm(OpenBox~.,data=dat,kernel="vanilladot",
              C=cp,epsilon=eps)
  ERROR[i] = SVRt@error
}
merge(hyper_grid[which.min(ERROR),],min(ERROR))
# plot results
library(plotly)
plot_ly(y=~CM, x=~EM, z=~ERROR) %>%
  add_surface() %>%
  layout(scene = list(xaxis = list(title = 'Epsilon value'),
                      yaxis = list(title = 'C value'),
                      zaxis = list(title = 'Training Error')))
```

The current optimized tuning parameters are $C = 4$ and $\epsilon = 0.7$ with the model training error being 0.6053. Of course, more detailed grid search will result in better tuning results.

Prediction by SVR is straightforward. Once all the model is trained, we can make prediction given input data. We apply the input data $\boldsymbol{X}_p = [1, 0, 0, 1, 5, 4, 3.5]$ to the trained linear SVR model using the `predict` command. R codes are presented below.

Figure 5.4: Search Through Various Tuning Parameters



```
# 2.3 prediction
INPUT <- data.frame(Animation=0,Crime=0,Romance=1,
                    Budget=5,Weeks=4,Screens=3.5)
predict(SVR,INPUT)
```

Comparing to the results by boosting and regression tree, we notice that the predicted value by linear SVR is larger.

5.2.3 SVR with Nonlinear Kernels

We now extend the above SVR framework for linear regression to nonlinear regression (SVR$_N$). We approximate the nonlinear regression function $f(\boldsymbol{X}_t)$ in terms of a set of basis function $\{h_m(\tilde{\boldsymbol{X}}_t)\}$ for $m = 1, ..., M$:

$$y_t = f(\boldsymbol{X}_t) + e_t = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(\tilde{\boldsymbol{X}}_t) + e_t$$

and we estimate the coefficients $\boldsymbol{\beta} = \left[\beta_0, \beta_1, ..., \beta_M\right]^\top$ through the minimization of

$$H(\boldsymbol{\beta}) = \sum_{t=1}^{n} V_\epsilon\big(y_t - f(\boldsymbol{X}_t)\big) + \frac{\lambda}{2}\sum_{m=1}^{M}\beta_m^2. \tag{5.5}$$

The solution of (5.5) has the form

$$\hat{f}(\boldsymbol{X}) = \sum_{t=1}^{n}(\hat{\alpha}_t^* - \hat{\alpha}_t)K(\boldsymbol{X}, \boldsymbol{X}_t) + \hat{\beta}_0\boldsymbol{\iota}_n$$

with $\hat{\alpha}_t^*$ and $\hat{\alpha}_t$ being the nonnegative multiplier of the following Lagrangian equation

$$\min_{\hat{\alpha}_t, \hat{\alpha}_t^*} \epsilon\sum_{t=1}^{n}(\hat{\alpha}_t^* + \hat{\alpha}_t) - \sum_{t=1}^{n} y_i(\hat{\alpha}_t^* - \hat{\alpha}_t) + \frac{1}{2}\sum_{t=1}^{n}\sum_{t'=1}^{n}(\hat{\alpha}_t^* - \hat{\alpha}_t)(\hat{\alpha}_{t'}^* - \hat{\alpha}_{t'})K(\boldsymbol{X}_t, \boldsymbol{X}_{t'})$$

similar to the SVR$_L$ case. In the SVR$_N$ case, a kernel function

$$K(\boldsymbol{X}_t, \boldsymbol{X}_{t'}) = \sum_{m=1}^{M} h_m(\boldsymbol{X}_t)h_m(\boldsymbol{X}_{t'})$$

is used to replace the inner product of the predictors $\boldsymbol{X}_t\boldsymbol{X}_{t'}^\top$ as in the SVR$_L$ case.

The popular nonlinear kernels include

$$\begin{aligned} K(\boldsymbol{X}_t, \boldsymbol{X}_{t'}) &= \exp\left(-\sigma\|\boldsymbol{X}_t - \boldsymbol{X}_{t'}^\top\|^2\right), & (5.6)\\ K(\boldsymbol{X}_t, \boldsymbol{X}_{t'}) &= \left(o + s\boldsymbol{X}_t\boldsymbol{X}_{t'}^\top\right)^d \quad \text{with } d \in \{1, 2, 3, ...\}, & (5.7) \end{aligned}$$

where Equation (5.6) represent the Gaussian kernel (with σ being the hyperparameter) and Equation (5.7) stands for the polynomial kernel with hyperparameters: degree d, scale s, and offset o. The default values for these hyperparameters are

(i) `sigma` for Gaussian kernel: if no value was assigned, ksvm will calculate `sigma` using cross-validation.

(ii) for the polynomial kernel: the default values are `degree = scale = offset = 1`.

Note that if we set $K(\boldsymbol{X}_t, \boldsymbol{X}_{t'}) = \boldsymbol{X}_t\boldsymbol{X}_{t'}^\top$, the SVR$_N$ becomes identical to SVR$_L$.

We repeat the movie forecasting exercises. This time, we apply the nonlinear SVRs to the data set. We consider the following kernels: (i) Gaussian with $\sigma = 1$ and (ii) polynomial with $d = 3$ and $o = s = 1$. We set ϵ and C to their default values. R codes are presented below.

```
# 3. SVR application - nonlinear
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
# 3.1 training
library(kernlab)
dat$Constant = NULL
SVRg <- ksvm(OpenBox~.,data=dat,kernel="rbfdot",
             kpar=list(sigma=1))
SVRp <- ksvm(OpenBox~.,data=dat,kernel="polydot",
             kpar=list(degree=3))
SVRg
SVRp
```

Both models report smaller training error. It is obvious that the model with polynomial kernel has better performance. We search through various values of d, o, and s:

$$\begin{aligned} d &\in [1, 2, ..., 10] \\ o &\in [0.5, 1, ..., 5] \\ s &\in [0.2, 0.4, ..., 2] \end{aligned}$$

(Crude) R codes are presented below. Remember to use `tictoc` to record CPU time. This may take a while.

```
# 4. search through tuning parameter space
dM = seq(1,10,1)
oM = seq(0.5,5,0.5)
sM = seq(0.2,2,0.2)
hyper_grid <- expand.grid(dM=dM,oM=oM,sM=sM)
ERROR = matrix(NA,nrow=nrow(hyper_grid))
library(tictoc)
tic()
for (i in 1:nrow(hyper_grid)) {
  d <- hyper_grid$dM[i]
  o <- hyper_grid$oM[i]
  s <- hyper_grid$sM[i]
  SVRt <- ksvm(OpenBox~.,data=dat,kernel="polydot",
               kpar=list(degree=d,offset=o,scale=s))
  ERROR[i] = SVRt@error
}
toc()
merge(hyper_grid[which.min(ERROR),],min(ERROR))
```

The optimized hyperparameters are $d = 9$, $o = 0.5$, and $s = 0.8$ with training error 0.0086. The whole program takes about 200 seconds to complete. See if you can improve the computational efficiency. [**Click to find answer**]

We revisit the prediction exercise in Section 3.2. We use the first 80 observations as training set and compute the MSFE on the rest evaluation set. We compare the MSFEs by SVR under linear kernel, Gaussian kernel ($\sigma = 1$), polynomial kernel ($d = 3, o = s = 1$), and optimized polynomial kernel ($d = 9$, $o = 0.5$, and $s = 0.8$). R codes are presented below.

```
# 5. SVR prediction exercises
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
library(kernlab)
dat$Constant = NULL
# 5.1 training
data_train = dat[1:80,]
SVR1 <- ksvm(OpenBox~.,data=data_train,kernel="vanilladot")
SVR2 <- ksvm(OpenBox~.,data=data_train,kernel="rbfdot",
             kpar=list(sigma=1))
SVR3 <- ksvm(OpenBox~.,data=data_train,kernel="polydot",
             kpar=list(degree=3))
SVR4 <- ksvm(OpenBox~.,data=data_train,kernel="polydot",
             kpar=list(degree=9,offset=0.5,scale=0.8))
# 5.2 evaluate prediction
INPUT <- dat[81:nrow(dat),2:ncol(dat)]
F1 = predict(SVR1,INPUT)
F2 = predict(SVR2,INPUT)
F3 = predict(SVR3,INPUT)
F4 = predict(SVR4,INPUT)
# 3.3 print results
yf = data.frame(SVR1=F1,SVR2=F2,SVR3=F3,SVR4=F4)
y0 = dat[81:nrow(dat),1]
apply((yf-y0)^2,2,mean)
```

What is your result? Why the result behaves like this?

5.3 An Overall Comparison

In this section, we conduct an overall performance of the methods we have learnt so far via the Monte Carlo simulation in Section 4.2. For completeness, we replicate the design here.

(i) Let the DGP be $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + u$.

(ii) $\boldsymbol{X}_1$ is a vector of ones and $\boldsymbol{X}_i \sim \mathrm{N}(0, 1)$ for $i = 2, ..., k$.

(iii) The error term $\boldsymbol{u}$ also follows standard normal distribution.

(iv) We set $k = 10$ and the coefficient $\beta_i = 0.1/i$ for $i = 1, ..., k$. That is the coefficient gets smaller for larger i and the corresponding predictor becomes less significant.

(v) We set $n = 100$ and let the first 90 observations as training set and the last 10 observations as evaluation set.

(vi) We compute the MSFE following MSFE $= \frac{1}{10}\sum_{i=1}^{10}(y_i^{ev} - \hat{y}_i^{ev})^2$, where $\hat{y}_i^{ev}$ is the prediction by a specific method.

(vii) We repeat the whole process 1000 times and report the average MSFEs of all the estimators. You may want to consider **computational cost**. Code carefully.

We consider the following estimators:

(i) Regression Tree;

(ii) Bagging Tree ($B = 100$);

(iii) Random Forest ($B = 100$);

(iv) Gradient Boosting Tree ($B = 25$);

(v) SVR with linear kernel;

(vi) SVR with Gaussian kernel ($\sigma = 1$);

(vii) SVR with polynomial kernel ($d = 3, o = s = 1$);

(viii) Ridge estimation; and

(ix) LASSO estimation.

All tuning parameters are set at default unless declared specifically. Compare the performance of the above 9 estimators. Which one is the best? If we set the coefficient to $\beta_i = 10/i$ for all i. Will the results change? What can you conclude?

[**Click to find answer**]

Answer 1:

```
####### A faster way #############
dat=read.csv("D:/Dropbox/R/teaching/machine learning/movie.csv",
             header = TRUE, fileEncoding="UTF-8-BOM")
dat$Constant = NULL
# setup parameter space
dM = seq(1,10,1)
oM = seq(0.5,5,0.5)
sM = seq(0.2,2,0.2)
hyper_grid <- expand.grid(dM=dM,oM=oM,sM=sM)
# search through space by foreach
library(tictoc)
library(doParallel)
cl <- makeCluster(12)
registerDoParallel(cl)
tic()
R <- foreach(i=1:nrow(hyper_grid)) %dopar% {
  library(kernlab)
  d <- hyper_grid$dM[i]
  o <- hyper_grid$oM[i]
  s <- hyper_grid$sM[i]
  SVRt <- ksvm(OpenBox~.,data=dat,kernel="polydot",
               kpar=list(degree=d,offset=o,scale=s))
  SVRt@error
}
toc()
# check results
R <- matrix(unlist(R), nrow=length(R), byrow=T)
merge(hyper_grid[which.min(R),],min(R))
stopCluster(cl)
```

Answer 2:

```
####### An overall comparison #############
library(doParallel)
cl <- makeCluster(12)
registerDoParallel(cl)
library(tictoc)
# define MSFE function
MSFE <- function(yt,xt,ye,xe){
  dat = data.frame(yt,xt)
  k = ncol(xt)
  out1 <- rpart(yt~.,dat,method="anova")
  out2 <- randomForest(yt~.,dat,mtry=ncol(xt),ntree=100)
  out3 <- randomForest(yt~.,dat,ntree=100)
  out4 <- xgboost(xt,yt,nrounds=25)
  out5 <- ksvm(xt[,2:k],yt,kernel="vanilladot")
  out6 <- ksvm(xt[,2:k],yt,kernel="rbfdot",
               kpar=list(sigma=1))
  out7 <- ksvm(xt[,2:k],yt,kernel="polydot",
               kpar=list(degree=3))
  out8 <- glmnet(xt, yt, alpha=0, lambda=1)
  out9 <- glmnet(xt, yt, alpha=1, lambda=1)
  RT = predict(out1,xe,'vector')
  BAG = predict(out2,xe)
  RF = predict(out3,xe)
  BT = predict(out4,as.matrix(xe))
  SVR1 = predict(out5,xe[,2:k])
  SVR2 = predict(out6,xe[,2:k])
  SVR3 = predict(out7,xe[,2:k])
  RID = predict(out8,as.matrix(xe))
  colnames(RID)="RID"
  LAS = predict(out9,as.matrix(xe))
  colnames(LAS)="LAS"
  yf = data.frame(RT,BAG,RF,BT,SVR1,SVR2,SVR3,RID,LAS)
  R = as.numeric(apply((yf-ye)^2,2,mean))
}
...
...
```

```
...
...
# set up parameters
n = 100
nt = 90
B = 1000
k = 10
b = as.matrix(10*(1:k)^(-1))
# parallel estimation
tic()
R <- foreach(i=1:B) %dopar% {
  library(rpart)
  library(randomForest)
  library(kernlab)
  library(xgboost)
  library(glmnet)
  x = matrix(NA,n,k)
  x[,1] = rep(1,n)
  x[,2:k] = matrix(rnorm(n*(k-1)),nrow=n)
  u = as.matrix(rnorm(n))
  y = x%*%b+u
  yt = y[1:nt,]
  xt = x[1:nt,]
  ye = y[(nt+1):n,]
  xe = data.frame(x[(nt+1):n,])
  MSFE(yt,xt,ye,xe)
}
toc()
# print results
R <- data.frame(matrix(unlist(R), nrow=length(R),
                       byrow=T))
names(R) = c("RT","BAG","RF","BT","SVR-L","SVR-G",
             "SVR-P","RIDGE","LASSO")
SAVE = apply(R,2,mean)
as.matrix(SAVE)
stopCluster(cl)
```

References

AKAIKE, H. (1973): "Information Theory and an Extension of the Maximum Likelihood Principle," *Second International Symposium on Information Theory*, 267–281.

AMEMIYA, T. (1980): "Selection of Regressors," *International Economic Review*, 21, 331–354.

ANDERSEN, T., T. BOLLERSLEV, F. DIEBOLD, AND H. EBENS (2001a): "The distribution of realized stock return volatility," *Journal of Financial Economics*, 61, 43–76.

ANDERSEN, T. G. AND T. BOLLERSLEV (1998): "Answering the Skeptics: Yes, Standard Volatility Models Do Provide Accurate Forecasts," *International Economic Review*, 39, 885–905.

ANDERSEN, T. G., T. BOLLERSLEV, AND F. X. DIEBOLD (2007): "Roughing It Up: Including Jump Components in the Measurement, Modeling, and Forecasting of Return Volatility," *The Review of Economics and Statistics*, 89, 701–720.

ANDERSEN, T. G., T. BOLLERSLEV, F. X. DIEBOLD, AND P. LABYS (2001b): "The Distribution of Realized Exchange Rate Volatility," *Journal of the American Statistical Association*, 96, 42–55.

——— (2003): "Modeling and Forecasting Realized Volatility," *Econometrica*, 71, 579–625.

AUDRINO, F. AND S. D. KNAUS (2016): "Lassoing the HAR Model: A Model Selection Perspective on Realized Volatility Dynamics," *Econometric Reviews*, 35, 1485–1521.

BARNDORFF-NEILSEN, O. E., S. KINNEBROUK, AND N. SHEPHARD (2010): "Measuring Downside Risk: Realised Semivariance," in *Volatility and Time Series Econometrics: Essays in Honor of Robert F. Engle*, ed. by T. Bollerslev, J. Russell, and M. Watson, Oxford University Press, 117–136.

BELLONI, A. AND V. CHERNOZHUKOV (2012): "Supplement to 'Least Squares After Model Selection in High-dimensional Sparse Models'," *DOI:10.3150/11-BEJ410SUPP*.

——— (2013): "Least Squares After Model Selection in High-dimensional Sparse Models," *Bernoulli*, 19, 521–547.

BICKEL, P. J., Y. RITOV, AND A. B. TSYBAKOV (2009): "Simultaneous Analysis of Lasso and Dantzig Selector," *The Annals of Statistics*, 37, 1705–1732.

BOLLERSLEV, T., J. LITVINOVA, AND G. TAUCHEN (2006): "Leverage and Volatility Feedback Effects in High-Frequency Data," *Journal of Financial Econometrics*, 4, 353–384.

BREIMAN, L. (1996): "Bagging Predictors," *Machine Learning*, 26, 123–140.

——— (2001): "Random Forests," *Machine Learning*, 45, 5–32.

BREIMAN, L., J. FRIEDMAN, AND C. J. STONE (1984): *Classification and Regression Trees*, Chapman and Hall/CRC.

BRITTEN-JONES, M. AND A. NEUBERGER (2000): "Option Prices, Implied Price Processes, and Stochastic Volatility," *The Journal of Finance*, 55, 839–866.

CANDES, E. AND T. TAO (2007): "The Dantzig Selector: Statistical Estimation when p is Much Larger than n," *The Annals of Statistics*, 35, 2313–2351.

CHEN, Y., W. K. HÄRDLE, AND U. PIGORSCH (2010): "Localized Realized Volatility Modeling," *Journal of the American Statistical Association*, 105, 1376–1393.

CORSI, F. (2009): "A Simple Approximate Long-Memory Model of Realized Volatility," *Journal of Financial Econometrics*, 7, 174–196.

CORSI, F., F. AUDRINO, AND R. RENÒ (2012): "HAR Modeling for Realized Volatility Forecasting," in *Handbook of Volatility Models and Their Applications*, John Wiley & Sons, Inc., 363–382.

CORSI, F., D. PIRINO, AND R. RENÒ (2010): "Threshold bipower variation and the impact of jumps on volatility forecasting," *Journal of Econometrics*, 159, 276 – 288.

CRAIOVEANU, M. AND E. HILLEBRAND (2012): "Why It Is OK to Use the HAR-RV $(1,5,21)$ Model," *Technical Report*, University of Central Missouri.

DACOROGNA, M. M., U. A. MŰLLER, R. J. NAGLER, R. B. OLSEN, AND O. V. PICTET (1993): "A geographical model for the daily and weekly seasonal volatility in the foreign exchange market," *Journal of International Money and Finance*, 12, 413 – 438.

DRUCKER, H., C. J. C. BURGES, L. KAUFMAN, A. J. SMOLA, AND V. VAPNIK (1996): "Support Vector Regression Machines," in *Advances in Neural Information Processing Systems 9*, ed. by M. C. Mozer, M. I. Jordan, and T. Petsche, MIT Press, 155–161.

EFRON, B. (1979): "Bootstrap Methods: Another Look at the Jackknife," *The Annals of Statistics*, 7, 1–26.

ENGLE, R. F. AND G. M. GALLO (2006): "A Multiple Indicators Model for Volatility Using Intra-daily Data," *Journal of Econometrics*, 131, 3 – 27.

FERNANDES, M., M. C. MEDEIROS, AND M. SCHARTH (2014): "Modeling and Predicting the CBOE Market Volatility Index," *Journal of Banking & Finance*, 40, 1–10.

FREUND, Y. AND R. E. SCHAPIRE (1997): "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences*, 55, 119 – 139.

FRIEDMAN, J. H. (1991): "Multivariate Adaptive Regression Splines," *The Annals of Statistics*, 19, 1–67.

GIACOMINI, R. AND H. WHITE (2006): "Tests of Conditional Predictive Ability," *Econometrica*, 74, 1545–1578.

HANSEN, B. E. (2008): "Least-squares forecast averaging," *Journal of Econometrics*, 146, 342–350.

HANSEN, B. E. AND J. S. RACINE (2012): "Jackknife Model Averaging," *Journal of Econometrics*, 167, 38–46.

HANSEN, P. R., Z. HUANG, AND H. H. SHEK (2012): "Realized GARCH: a joint model for returns and realized measures of volatility," *Journal of Applied Econometrics*, 27, 877–906.

HASTIE, T., R. TIBSHIRANI, AND J. FRIEDMAN (2009): *The Elements of Statistical Learning:*, Springer Series in Statistics, New York, NY, USA: Springer New York Inc.

HENDRY, D. F. AND B. NIELSEN (2007): *Econometric Modeling: A Likelihood Approach,*, Princeton University Press, chap. 19, 286–301.

HIRANO, K. AND J. H. WRIGHT (2017): "Forecasting With Model Uncertainty: Representations and Risk Reduction," *Econometrica*, 85 (2), 617–643.

HUANG, X. AND G. TAUCHEN (2005): "The Relative Contribution of Jumps to Total Price Variance," *Journal of Financial Econometrics*, 3, 456–499.

HUNT, E., J. MARTIN, AND P. STONE (1966): *Experiments in Induction*, Academic Press, New York.

ING, C.-K. AND C.-Z. WEI (2003): "On same-realization prediction in an infinite-order autoregressive process," *Journal of Multivariate Analysis*, 85, 130 – 155.

JIANG, G. J. AND Y. S. TIAN (2005): "The Model-Free Implied Volatility and Its Information Content," *Review of Financial Studies*, 18, 1305–1342.

KARALIC, A. AND B. CESTNIK (1991): "The Bayesian Approach to Tree-structured Regression," *Proceedings of Information Technology Interfaces 091*.

KEMP, G. C. (1997): "Linear Combinations of Stationary Processes—Solution," *Econometric Theory*, 13, 897–898.

KREISS, J.-P. AND S. N. LAHIRI (2012): "Bootstrap Methods for Time Series," in *Time Series Analysis: Methods and Applications, Volume 30*, ed. by T. S. Rao, S. S. Rao, and C. Rao, North Holland, chap. 1, 3–26.

KUERSTEINER, G. AND R. OKUI (2010): "Constructing Optimal Instruments by First-Stage Prediction Averaging," *Econometrica*, 78, 697–718.

KULPERGER, R. J. AND B. L. S. PRAKASA RAO (1989): "Bootstrapping a Finite State Markov Chain," *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)*, 51, 178–191.

KÜNSCH, H. R. (1989): "The Jackknife and the Bootstrap for General Stationary Observations," *The Annals of Statistics*, 17, 1217–1241.

LEHRER, S. F. AND T. XIE (2017): "Box Office Buzz: Does Social Media Data Steal the Show from Model Uncertainty When Forecasting for Hollywood?" *The Review of Economics and Statistics*, 99, 749–755.

——— (2018): "The Bigger Picture: Combining Econometrics with Analytics Improve Forecasts of Movie Success," *Working Paper*.

LEHRER, S. F., T. XIE, AND X. ZHANG (2018): "Twits versus Tweets: Does Adding Social Media Wisdom Trump Admitting Ignorance when Forecasting the CBOE VIX?" *Working Paper*.

LIN, Y.-N. (2007): "Pricing VIX futures: Evidence from integrated physical and risk-neutral probability measures," *Journal of Futures Markets*, 27, 1175–1217.

LIU, Q. AND R. OKUI (2013): "Heteroskedasticity-robust C_p Model Averaging," *The Econometrics Journal*, 16, 463–472.

LIU, Y. AND T. XIE (2018): "Machine Learning Versus Econometrics: Prediction of Box Office," *Applied Economics Letter*, Forthcoming.

MCALEER, M. AND M. C. MEDEIROS (2008): "A multiple regime smooth transition Heterogeneous Autoregressive model for long memory and asymmetries," *Journal of Econometrics*, 147, 104–119.

MCCULLOCH, W. S. AND W. PITTS (1943): "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, 5, 115–133.

MORGAN, J. N. AND J. A. SONQUIST (1963): "Problems in the Analysis of Survey Data, and a Proposal," *Journal of the American Statistical Association*, 58, 415–434.

MÜLLER, U. A., M. M. DACOROGNA, R. D. DAVÉ, O. V. PICTET, R. B. OLSEN, AND J. WARD (1993): "Fractals and Intrinsic Time - a Challenge to Econometricians," Tech. rep.

NEAL, R. M. (1996): *Bayesian Learning for Neural Networks*, Secaucus, NJ, USA: Springer-Verlag New York, Inc.

PATTON, A. J. AND K. SHEPPARD (2015): "Good Volatility, Bad Volatility: Signed Jumps and The Persistence of Volatility," *The Review of Economics and Statistics*, 97, 683–697.

QUINLAN, J. R. (1986): "Induction of Decision Trees," *Machine Learning*, 1, 81–106.

——— (1992): "Learning With Continuous Classes," World Scientific, 343–348.

RUMELHART, D. E., G. E. HINTON, AND R. J. WILLIAMS (1986): "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1," Cambridge, MA, USA: MIT Press, chap. Learning Internal Representations by Error Propagation, 318–362.

SCHARTH, M. AND M. MEDEIROS (2009): "Asymmetric effects and long memory in the volatility of Dow Jones stocks," *International Journal of Forecasting*, 25, 304–327.

TIBSHIRANI, R. (1996): "Regression Shrinkage and Selection Via the Lasso," *Journal of the Royal Statistical Society, Series B*, 58, 267–288.

VAPNIK, V. N. (1996): *The Nature of Statistical Learning Theory*, New York, NY, USA: Springer-Verlag New York, Inc.

VORTELINOS, D. I. (2017): "Forecasting realized volatility: HAR against Principal Components Combining, neural networks and GARCH," *Research in International Business and Finance*, 39, 824 – 839.

WAGER, S. AND S. ATHEY (2017): "Estimation and Inference of Heterogeneous Treatment Effects using Random Forests," *Journal of the American Statistical Association*, Forthcoming.

WANG, T., Y. SHEN, Y. JIANG, AND Z. HUANG (2017): "Pricing the CBOE VIX Futures with the Heston–Nandi GARCH Model," *Journal of Futures Markets*, 37, 641–659.

WANG, Y., F. MA, Y. WEI, AND C. WU (2016): "Forecasting Realized Volatility in A Changing World: A Dynamic Model Averaging Approach," *Journal of Banking & Finance*, 64, 136–149.

WHALEY, R. E. (2000): "The Investor Fear Gauge," *The Journal of Portfolio Management*, 26, 12–17.

XIE, T. (2015): "Prediction Model Averaging Estimator," *Economics Letters*, 131, 5–8.

——— (2017): "Heteroscedasticity-robust Model Screening: A Useful Toolkit for Model Averaging in Big Data Analytics," *Economics Letters*, accepted.

ZHANG, J. E. AND Y. ZHU (2006): "VIX Futures," *Journal of Futures Markets*, 26, 521–531.

ZHANG, X., A. T. WAN, AND G. ZOU (2013): "Model Averaging by Jackknife Criterion in Models with Dependent Data," *Journal of Econometrics*, 174, 82–94.

ZHANG, X., G. ZOU, AND R. J. CARROLL (2015): "Model Averaging Based on Kullback-Leibler Distance," *Statistica Sinica*, 25, 1583–1598.

ZHU, S.-P. AND G.-H. LIAN (2012): "An Analytical Formula for VIX Futures and Its Applications," *Journal of Futures Markets*, 32, 166–190.