STAT 8020 R Session 6: Nonparametric Regression and Shrinkage Methods

Whitney Huang, Clemson University

Contents

Non-parametric Regression: Motorcycle Accident Simulation Data 1
Load and Plot the Data 1
Linear and Polynomial Regression 2
Kernel Regression 4
Local Polynomial Regression Fitting (*loess*) 5
Regression Splines 6
Generalized Additive Models 8
Smoothing Splines 10
Comparing Kernel Estimator/Regression Spline/Smoothing Spline Fits 13
Generalized Additive Models for Multiple Predictors 14
Shrinkage Methods 16
Ridge Regression 16
The Lasso 23

Non-parametric Regression: Motorcycle Accident Simulation Data

A data frame containing a series of measurements of head acceleration from a simulated motorcycle accident, commonly used for testing crash helmets.

- `times`: time in milliseconds after impact
- `accel`: head acceleration in g

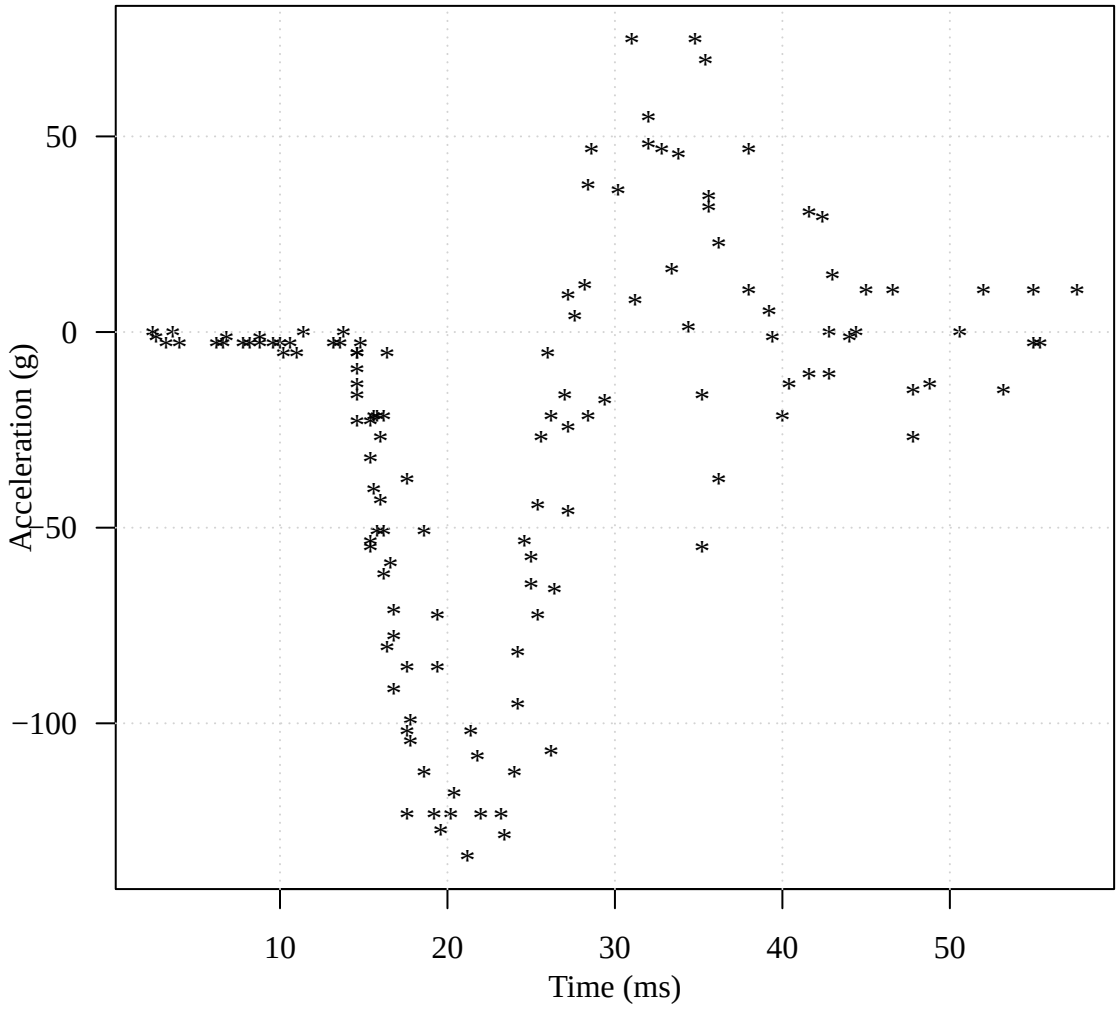
Data Source: Silverman, B. W. (1985) Some aspects of the spline smoothing approach to non-parametric curve fitting. *Journal of the Royal Statistical Society series B* 47, 1–52.

Load and Plot the Data

The `mcycle` dataset from the `R MASS` package package contains measurements of head acceleration recorded during a simulated motorcycle crash experiment.

In this example, we first load the dataset and create a scatterplot to examine the relationship between time after impact and head acceleration. Visualizing the data helps us assess potential nonlinear patterns and motivates the use of nonparametric regression methods.

```r
library(MASS)
data(mcycle)
# Attach variables for easier access
attach(mcycle)
par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*", cex = 1,
     xlab = "Time (ms)", ylab = "Acceleration (g)")
grid()
```



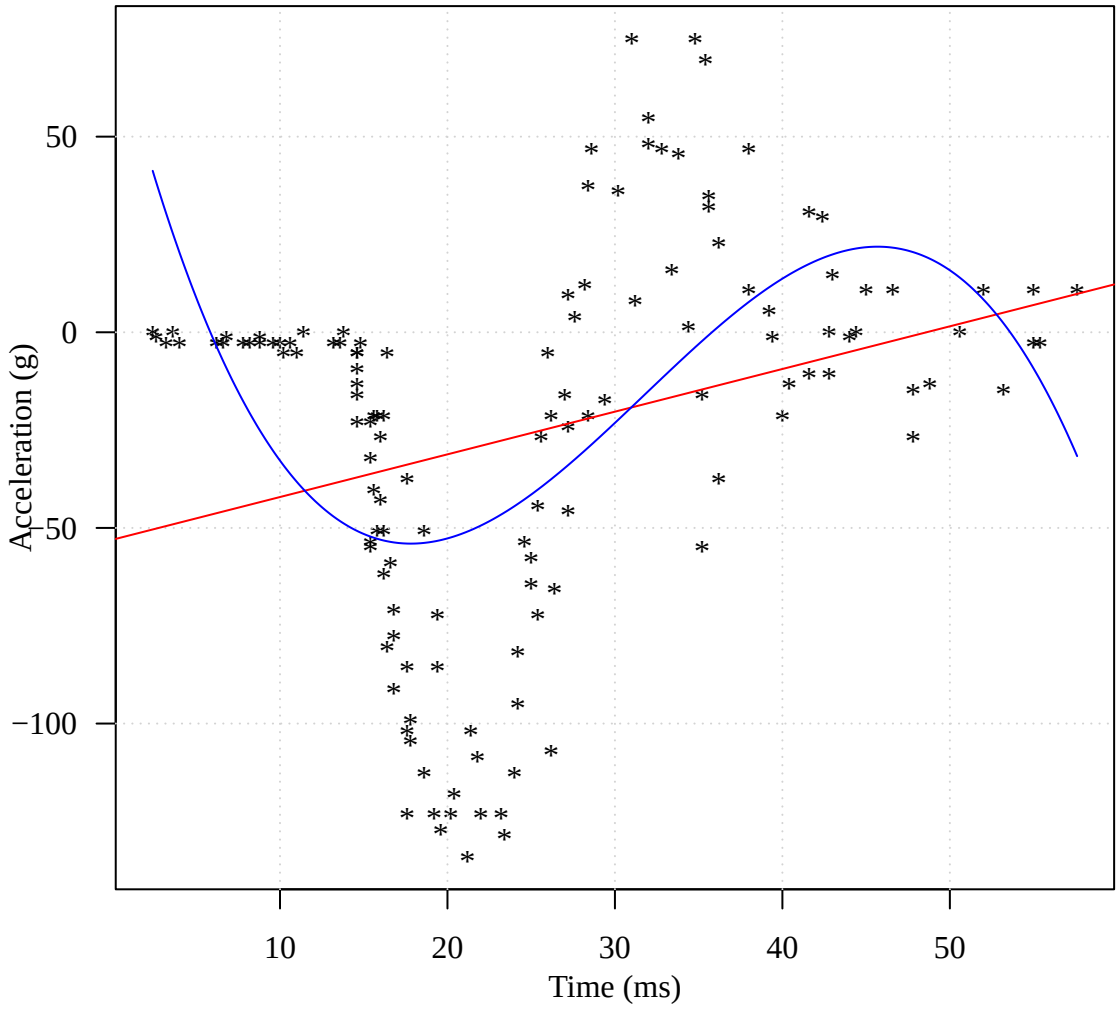
Linear and Polynomial Regression

We begin by fitting a simple linear regression model to describe the relationship between time after impact and head acceleration. Because the scatterplot suggests substantial curvature, we also fit a cubic polynomial regression model to allow for a more flexible nonlinear trend.

The fitted curves are then overlaid on the scatterplot for comparison. This visualization helps illustrate the limitations of a simple linear model and the improved flexibility provided by polynomial regression.

```r
# Create a sequence of time values for smooth prediction curves
rg <- range(times)
xg = seq(rg[1], rg[2], 0.1) # prediction grids
```

```r
par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*", cex = 1,
     xlab = "Time (ms)", ylab = "Acceleration (g)")
grid()
# ------------------------------------------------------------
# Fit a simple linear regression model
# ------------------------------------------------------------
lmFit <- lm(accel ~ times, data = mcycle)
# Add the fitted linear regression line to the plot
abline(lmFit, col = "red")
# ------------------------------------------------------------
# Fit a cubic polynomial regression model
# poly(times, 3) generates a degree-3 polynomial basis
# ------------------------------------------------------------
Cub.polyFit <- lm(accel ~ poly(times, 3), data = mcycle)
# Predict acceleration values over the grid of time points
Cub.polyPred <- predict(Cub.polyFit, data.frame(times = xg))
# Add the cubic polynomial regression curve
lines(xg, Cub.polyPred, col = "blue")
```



Kernel Regression

Kernel regression estimates the conditional mean function by taking a weighted average of nearby observations. Observations closer to the target point receive larger weights, while distant observations receive smaller weights.

The smoothness of the fitted curve is controlled by the bandwidth parameter h:

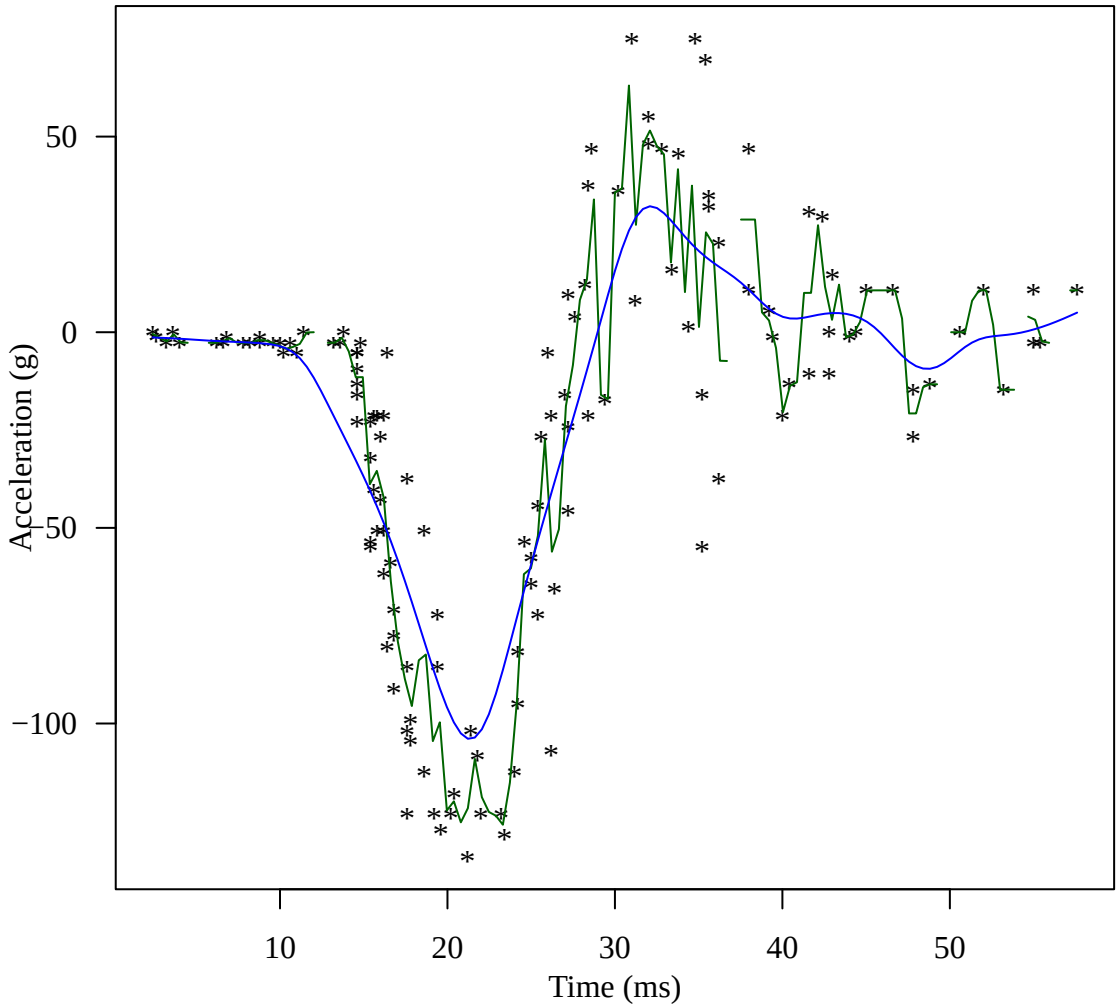
- small $h \Rightarrow$ very flexible and wiggly fit
- large $h \Rightarrow$ smoother fit with less local variation

The estimator is given by

$\hat{f}(x) = \hat{\mathbb{E}}(y|x) = \frac{\sum_{i=1}^{n} K_h((x-x_i)/h) y_i}{\sum_{i=1}^{n} K_h((x-x_i)/h)}$, where K_h is a kernel function with a bandwidth h.

In the following example, we compare two kernel smoothers with different bandwidth values to illustrate the effect of smoothing.

```
# ------------------------------------------------------------
# Kernel regression using Gaussian kernels
# ------------------------------------------------------------

# Smaller bandwidth:
# captures more local detail but may appear wiggly

KernFit <- with(mcycle, ksmooth(times, accel, kernel = "normal", bandwidth = 0.5))

# Larger bandwidth:
# produces a smoother curve with less local variation

KernFit2 <- with(mcycle, ksmooth(times, accel, kernel = "normal", bandwidth = 5))

par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*", cex = 1,
     xlab = "Time (ms)", ylab = "Acceleration (g)")
lines(KernFit$x, KernFit$y, col = "darkgreen")
lines(KernFit2$x, KernFit2$y, col = "blue")
```



Local Polynomial Regression Fitting (*loess*)

Local polynomial regression, implemented through `loess`, fits low-degree polynomial models within local neighborhoods of the predictor space. Unlike global polynomial regression, `LOESS` adapts locally and can better capture complex nonlinear patterns.

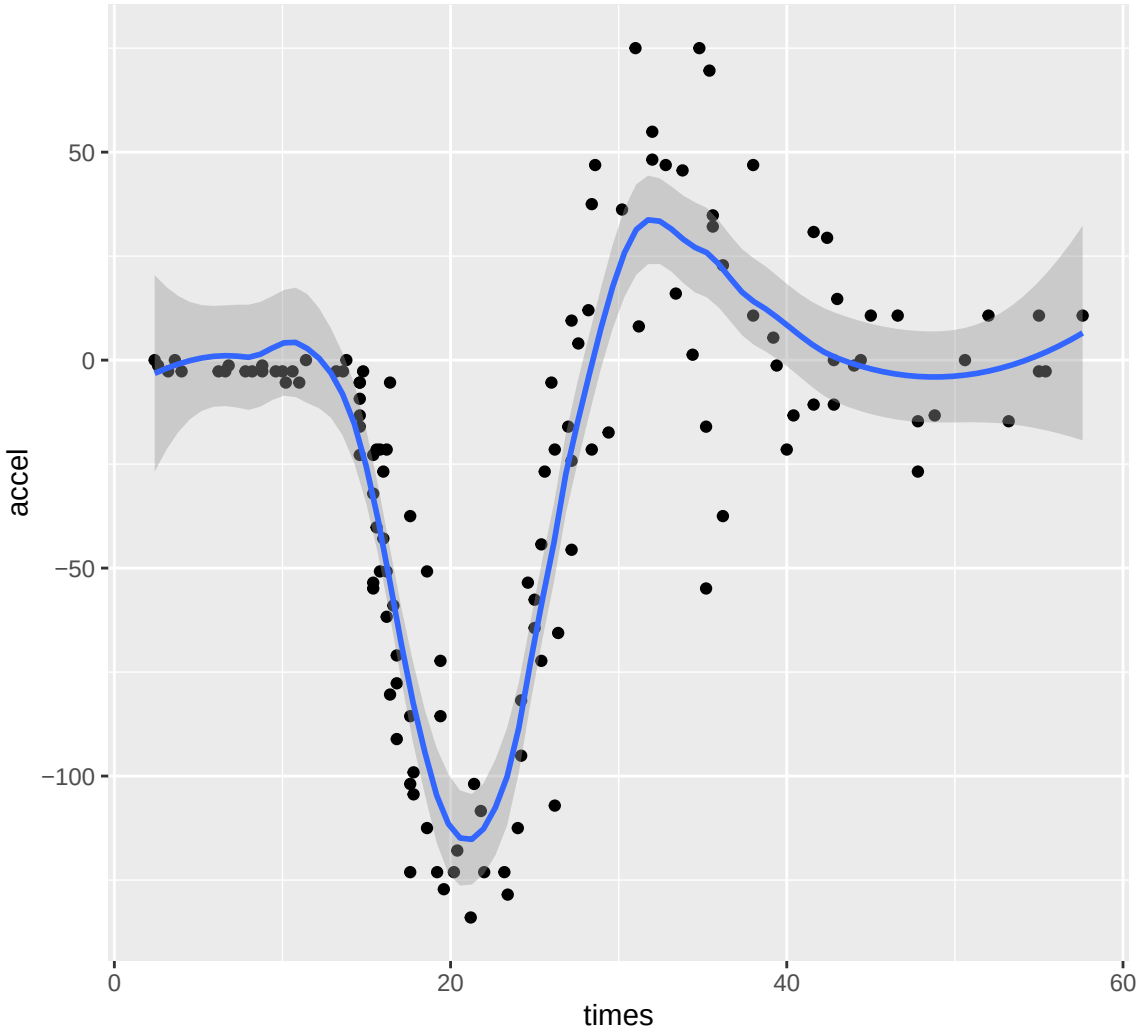
The span parameter controls the amount of smoothing:

- smaller span ⇒ more flexible local fitting
- larger span ⇒ smoother overall trend

In this example, we fit a quadratic local polynomial smoother and display the fitted curve together with a confidence band.

```
# Load ggplot2 for visualization
library(ggplot2)

# Initialize the scatterplot
plot <- ggplot(aes(x = times, y = accel), data = mcycle)
# Add observed data points
plot <- plot + geom_point()
```

```r
# ------------------------------------------------------------
# Fit a LOESS smoother
#
# degree = 2  -> local quadratic regression
# span = 0.4  -> controls smoothing level
# se = TRUE   -> display confidence band
# ------------------------------------------------------------
(plot <- plot + geom_smooth(method = "loess", degree = 2, span = 0.4, se = TRUE))
```



Regression Splines

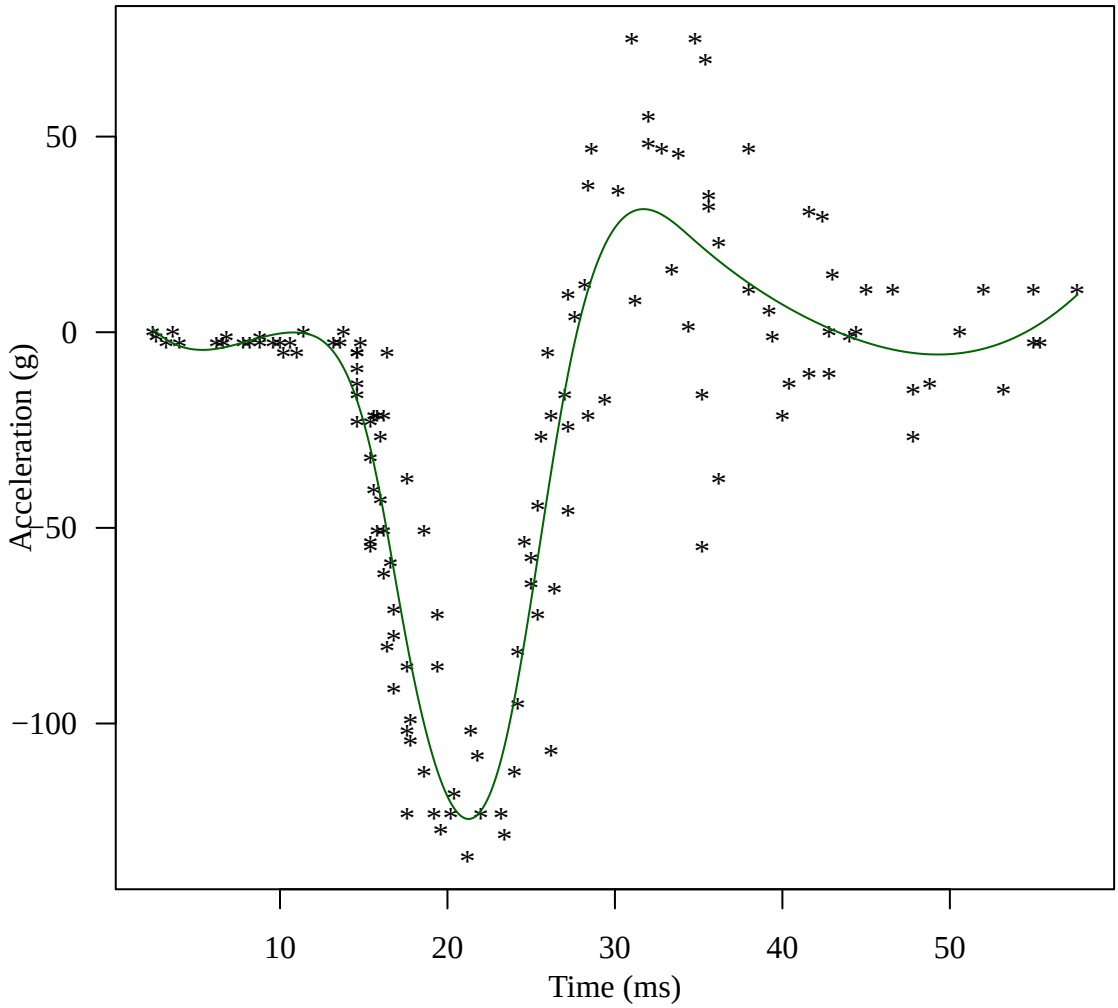
Regression splines model nonlinear relationships by dividing the predictor space into smaller regions and fitting piecewise polynomial functions. The pieces are joined smoothly at selected points called knots.

In this example, we use B-splines through the `bs()` function. The argument `df = 10` controls the flexibility of the spline fit. A larger degree of freedom allows a more flexible curve, while a smaller value produces a smoother fit.

```r
# Load the splines package for B-spline basis functions
library(splines)
# Fit a regression spline model using B-spline basis functions
# df = 10 controls the flexibility of the fitted spline curve
RegSplineFit <- lm(accel ~ bs(times, df = 10), data = mcycle)
summary(RegSplineFit)
```

```
##
## Call:
## lm(formula = accel ~ bs(times, df = 10), data = mcycle)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -76.673 -12.362  -0.557  13.139  51.740
##
## Coefficients:
##                        Estimate Std. Error t value Pr(>|t|)
## (Intercept)              0.9312    14.4492   0.064  0.94872
## bs(times, df = 10)1    -12.2008    37.5144  -0.325  0.74556
## bs(times, df = 10)2      6.2223    23.6415   0.263  0.79284
## bs(times, df = 10)3     -7.3726    18.2652  -0.404  0.68718
## bs(times, df = 10)4   -118.7497    17.9975  -6.598 1.13e-09 ***
## bs(times, df = 10)5   -152.4486    20.0955  -7.586 7.25e-12 ***
## bs(times, df = 10)6     50.0827    18.7966   2.664  0.00875 **
## bs(times, df = 10)7     19.4271    19.3827   1.002  0.31819
## bs(times, df = 10)8     -8.1814    23.9354  -0.342  0.73308
## bs(times, df = 10)9    -11.1443    29.2202  -0.381  0.70358
## bs(times, df = 10)10     8.6378    23.6119   0.366  0.71513
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 22.68 on 122 degrees of freedom
## Multiple R-squared:  0.7964, Adjusted R-squared:  0.7797
## F-statistic: 47.72 on 10 and 122 DF,  p-value: < 2.2e-16
```

```r
# Predict fitted values over a fine grid of time points
RegSplinePred <- predict(RegSplineFit, data.frame(times = xg))

par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*",
     xlab = "Time (ms)", ylab = "Acceleration (g)")
lines(xg, RegSplinePred, col = "darkgreen")
```



Generalized Additive Models

Generalized additive models, or GAMs, extend linear models by allowing nonlinear smooth functions of predictors. Instead of assuming a straight-line relationship, a GAM estimates a smooth function `s(x)` from the data.

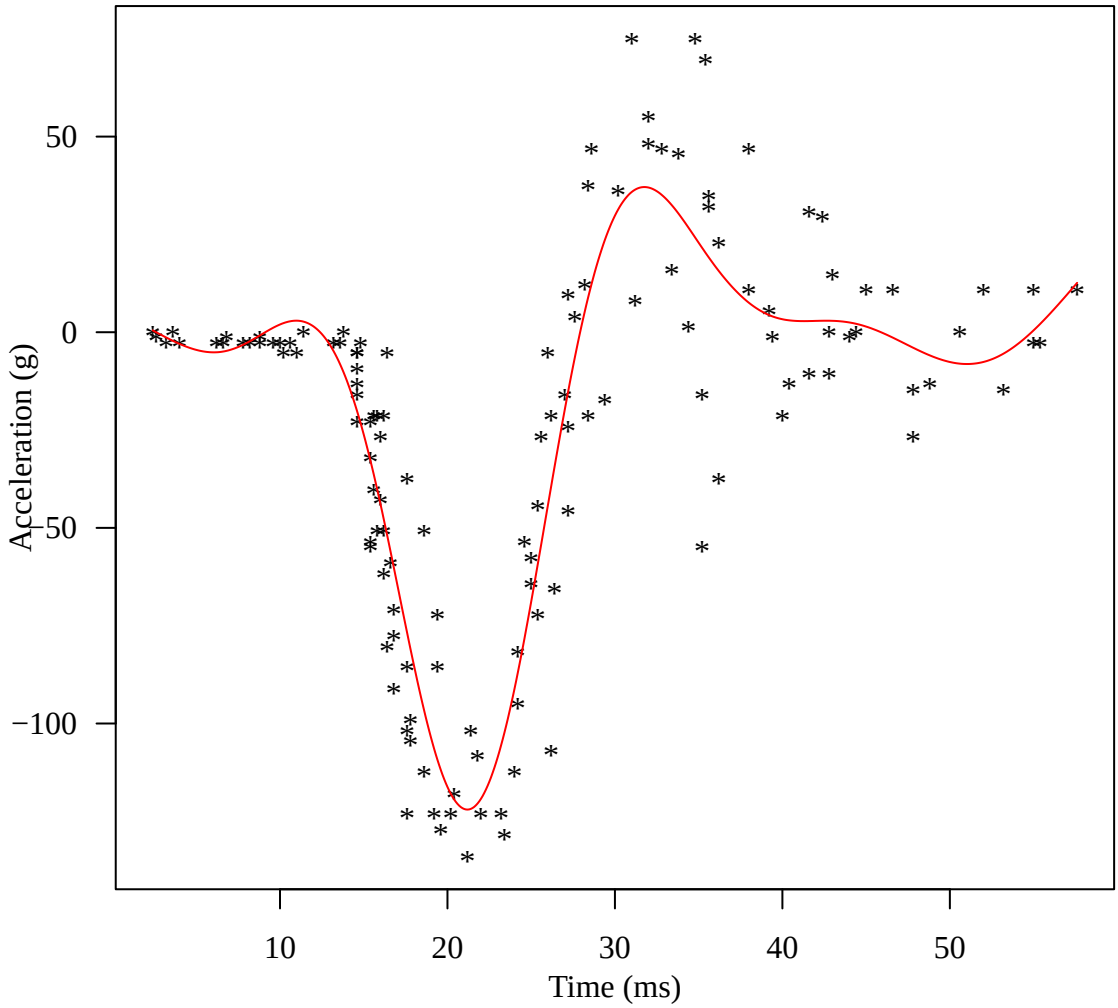
For the motorcycle accident data, we model head acceleration as a smooth function of time after impact.

```
# Load the mgcv package for fitting generalized additive models
library(mgcv)
# Fit a GAM with a smooth function of time
# s(times) allows the relationship between time and acceleration to be nonlinear
GAMFit <- gam(accel ~ s(times), data = mcycle)
summary(GAMFit)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## accel ~ s(times)
```

```
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -25.546      1.951   -13.1   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##            edf Ref.df     F p-value
## s(times) 8.693  8.972 53.52  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.783   Deviance explained = 79.8%
## GCV = 545.78  Scale est. = 506       n = 133
```

```r
# Predict fitted values over the grid of time points
GAMpred <- predict(GAMFit, data.frame(times = xg))

par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*",
     xlab = "Time (ms)", ylab = "Acceleration (g)")
lines(xg, GAMpred, col = "red")
```



Smoothing Splines

Smoothing splines provide another flexible approach for estimating a smooth nonlinear relationship. Unlike regression splines, which require choosing a fixed number of basis functions or knots, smoothing splines estimate a smooth curve by balancing goodness of fit with smoothness.

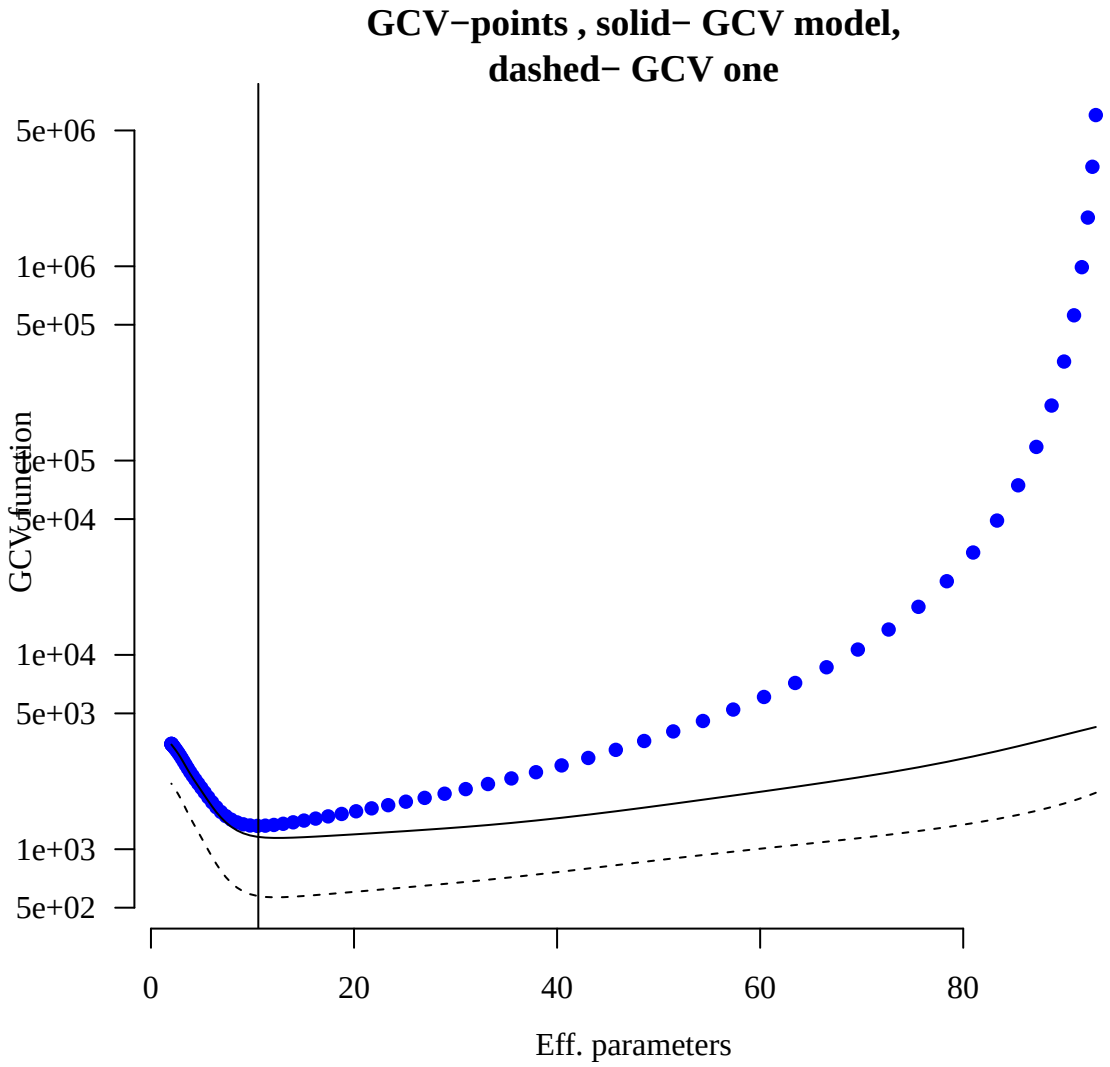
The `sreg()` function from the `fields` package fits a smoothing spline and automatically selects a smoothing level.

```
# Load the fields package for smoothing spline regression
library(fields)
# Fit a smoothing spline model
# sreg() estimates a smooth curve while controlling roughness
SpFit <- sreg(times, accel)
summary(SpFit)
```

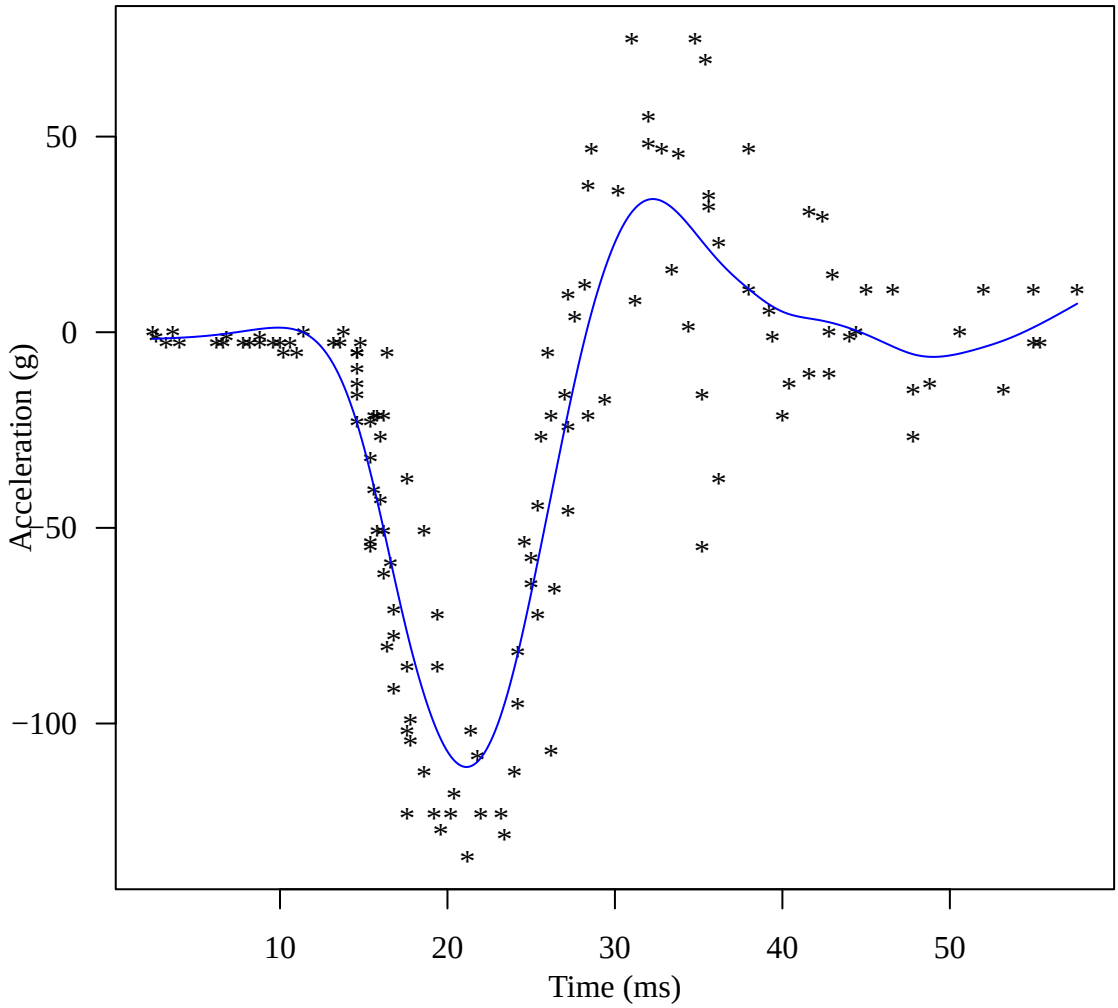
```
## CALL:
## sreg(x = times, y = accel)
##
##  Number of Observations:             133
##  Number of unique points:            133
```

```
##  Eff. degrees of freedom for spline: 10.6
##  Residual degrees of freedom:        122.4
##  GCV est. tau                        22.97
##  Pure error tau                      24.49
##  lambda                              0.3826
##
## RESIDUAL SUMMARY:
##      min    1st Q   median    3rd Q      max
## -78.1500 -13.8800  -0.7238  13.6300  49.6300
##
## DETAILS ON SMOOTHING PARAMETER:
##  Method used:      Cost:
##    lambda       trA       GCV   GCV.one GCV.model    tauHat
##    0.3826   10.5726 1318.0646  573.4152 1156.4850   22.9746
##
##  Summary of estimates for lambda
##             lambda    trA    GCV tauHat converge
## GCV         0.3826 10.573 1318.1  22.97       13
## GCV.model   0.1835 12.467 1142.5  22.64       12
## GCV.one     0.1981 12.253  565.5  22.66       12
## pure error 1.1041  8.375 1380.7  24.49       NA
```

```r
# Diagnostic plot for the smoothing spline fit
par(las = 1, mar = c(3.5, 3.5, 2, 0.5), mgp = c(2.5, 1, 0), family = "serif")
plot(SpFit, which = 3, col = "blue", pch = 16)
```




```
SpPred <- predict(SpFit, xg)

par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*",
     xlab = "Time (ms)", ylab = "Acceleration (g)")
lines(xg, SpPred , col = "blue")
```



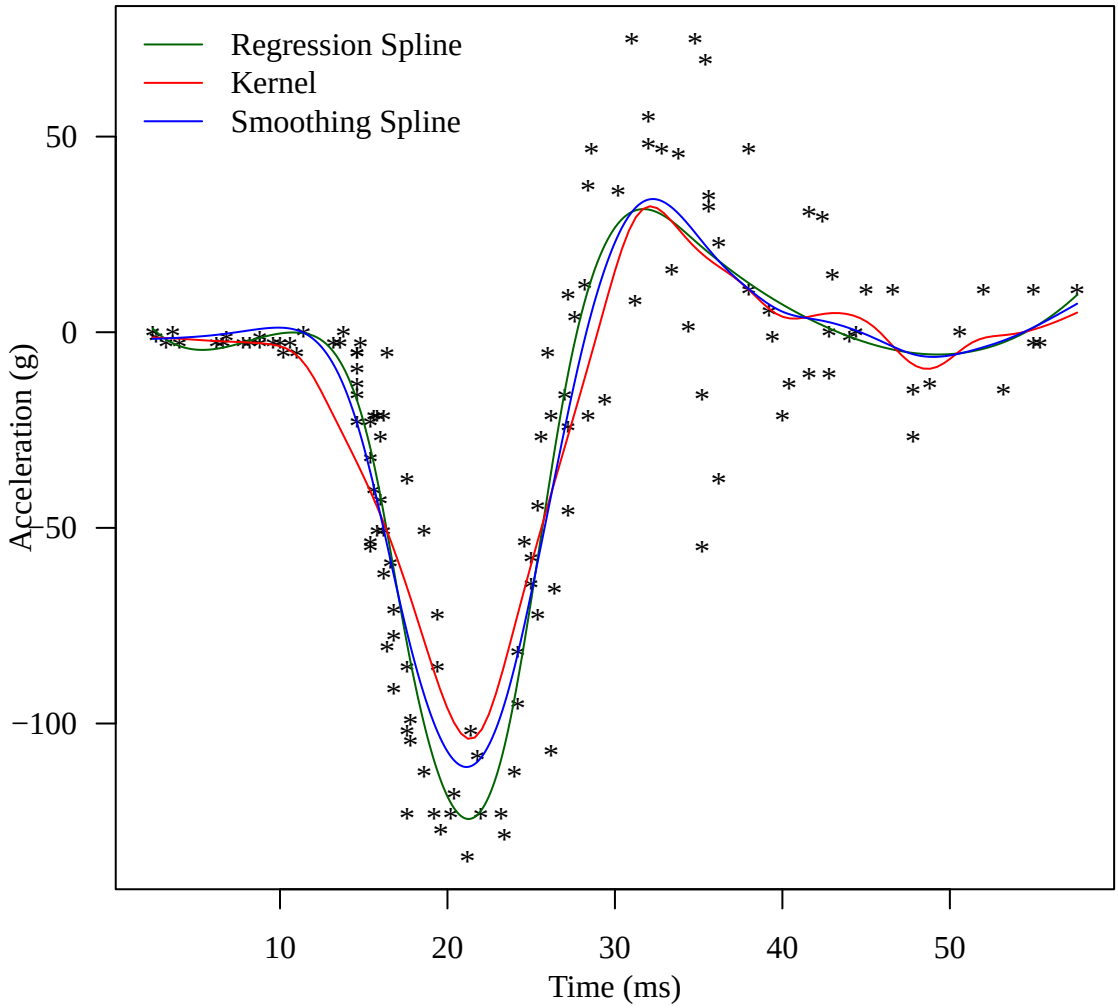
Comparing Kernel Estimator/Regression Spline/Smoothing Spline Fits

After fitting several nonparametric regression methods, we can overlay their fitted curves on the same scatterplot. This comparison helps illustrate how different smoothing approaches capture the nonlinear pattern in the data.

Although all three methods are flexible, they control smoothness in different ways:

- kernel regression uses a bandwidth;
- regression splines use basis functions and degrees of freedom;
- smoothing splines balance fit and smoothness through a smoothing penalty.

```
par(las = 1, mar = c(3.5, 3.5, 1, 0.5), mgp = c(2, 1, 0), family = "serif")
plot(times, accel, pch = "*",
     xlab = "Time (ms)", ylab = "Acceleration (g)")
lines(xg, RegSplinePred, col = "darkgreen")
lines(KernFit2$x, KernFit2$y, col = "red")
lines(xg, SpPred, col = "blue")
# Add a legend to identify the fitted curves
legend("topleft", legend = c("Regression Spline", "Kernel", "Smoothing Spline"),
       col = c("darkgreen", "red", "blue"), lty = 1, bty = "n")
```



Generalized Additive Models for Multiple Predictors

GAMs can also be used when there are multiple predictors. In this example, we use the `savings` dataset from the `faraway` package and model the savings rate as an additive combination of smooth functions of several predictors.

The model has the form

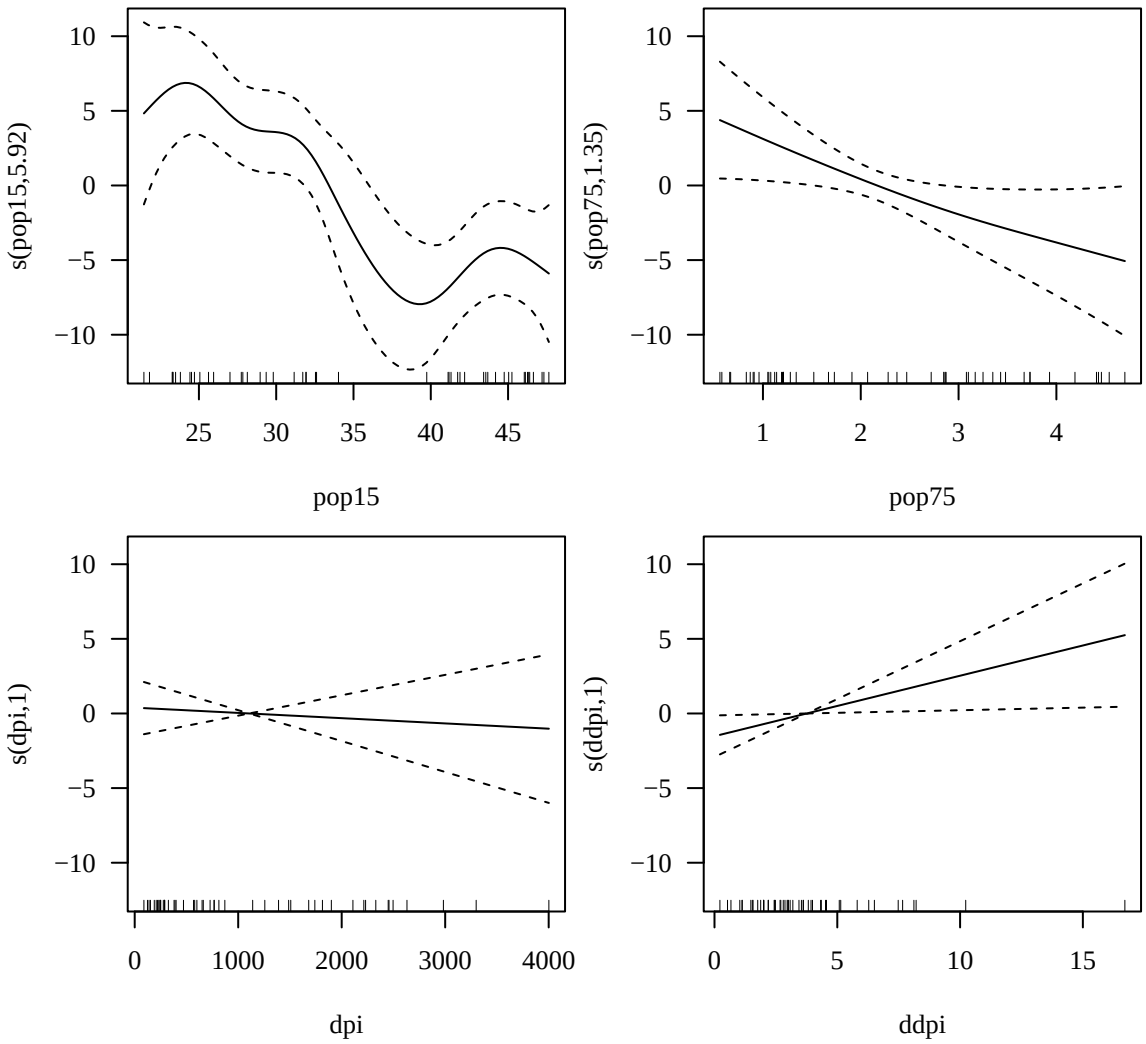
$sr = s(pop15) + s(pop75) + s(dpi) + s(ddpi) + \epsilon.$

Each smooth term allows the corresponding predictor to have a potentially nonlinear effect on the response, while the additive structure keeps the model interpretable.

```
# Load the faraway package and the savings dataset
library(faraway)
# Fit a generalized additive model with multiple smooth predictors
# sr    = savings rate
# pop15 = percentage of population under age 15
# pop75 = percentage of population over age 75
# dpi   = real per-capita disposable income
# ddpi  = growth rate of real per-capita disposable income
gamod <- gam(sr ~ s(pop15) + s(pop75) + s(dpi) + s(ddpi), data = savings)
summary(gamod)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## sr ~ s(pop15) + s(pop75) + s(dpi) + s(ddpi)
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   9.6710     0.4816   20.08   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##            edf Ref.df     F p-value
## s(pop15) 5.924  7.064 3.608 0.00406 **
## s(pop75) 1.350  1.623 2.811 0.06586 .
## s(dpi)   1.000  1.000 0.168 0.68403
## s(ddpi)  1.000  1.000 4.789 0.03455 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## R-sq.(adj) =  0.422   Deviance explained = 53.2%
## GCV = 14.593  Scale est. = 11.595    n = 50
```

```
par(mfrow = c(2, 2), mar = c(4, 3.85, 0.8, 0.5), family = "serif")
plot(gamod, las = 1)
```



Shrinkage Methods

The remainder of this R session is largely based on the `R` lab "Ridge Regression and the Lasso" from the book *An Introduction to Statistical Learning* by *Gareth James*, *Daniela Witten*, *Trevor Hastie*, and *Robert Tibshirani*.

We will use the `glmnet` package to perform ridge regression and the lasso for predicting `Salary` using the `Hitters` dataset.

Ridge Regression

Ridge regression is a shrinkage method that modifies ordinary least squares (OLS) by adding a penalty term to the loss function. The penalty shrinks the regression coefficients toward zero, which can reduce variability and improve prediction accuracy, especially when predictors are highly correlated.

1. Data Setup

```
# Load the ISLR package and the Hitters dataset
library(ISLR)
```

```r
data(Hitters)
# Remove observations containing missing values
Hitters = na.omit(Hitters)
head(Hitters)
```

```
##                   AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun
## -Alan Ashby         315   81     7   24  38    39    14   3449   835      69
## -Alvin Davis        479  130    18   66  72    76     3   1624   457      63
## -Andre Dawson       496  141    20   65  78    37    11   5628  1575     225
## -Andres Galarraga   321   87    10   39  42    30     2    396   101      12
## -Alfredo Griffin    594  169     4   74  51    35    11   4408  1133      19
## -Al Newman          185   37     1   23   8    21     2    214    42       1
##                   CRuns CRBI CWalks League Division PutOuts Assists Errors
## -Alan Ashby         321  414    375      N        W     632      43     10
## -Alvin Davis        224  266    263      A        W     880      82     14
## -Andre Dawson       828  838    354      N        E     200      11      3
## -Andres Galarraga    48   46     33      N        E     805      40      4
## -Alfredo Griffin    501  336    194      A        W     282     421     25
## -Al Newman           30    9     24      N        E      76     127      7
##                   Salary NewLeague
## -Alan Ashby        475.0         N
## -Alvin Davis       480.0         A
## -Andre Dawson      500.0         N
## -Andres Galarraga   91.5         N
## -Alfredo Griffin   750.0         A
## -Al Newman          70.0         A
```

```r
summary(Hitters)
```

```
##      AtBat            Hits           HmRun            Runs      
##  Min.   : 19.0   Min.   :  1.0   Min.   : 0.00   Min.   :  0.00  
##  1st Qu.:282.5   1st Qu.: 71.5   1st Qu.: 5.00   1st Qu.: 33.50  
##  Median :413.0   Median :103.0   Median : 9.00   Median : 52.00  
##  Mean   :403.6   Mean   :107.8   Mean   :11.62   Mean   : 54.75  
##  3rd Qu.:526.0   3rd Qu.:141.5   3rd Qu.:18.00   3rd Qu.: 73.00  
##  Max.   :687.0   Max.   :238.0   Max.   :40.00   Max.   :130.00  
##       RBI             Walks            Years            CAtBat       
##  Min.   :  0.00   Min.   :  0.00   Min.   : 1.000   Min.   :   19.0  
##  1st Qu.: 30.00   1st Qu.: 23.00   1st Qu.: 4.000   1st Qu.:  842.5  
##  Median : 47.00   Median : 37.00   Median : 6.000   Median : 1931.0  
##  Mean   : 51.49   Mean   : 41.11   Mean   : 7.312   Mean   : 2657.5  
##  3rd Qu.: 71.00   3rd Qu.: 57.00   3rd Qu.:10.000   3rd Qu.: 3890.5  
##  Max.   :121.00   Max.   :105.00   Max.   :24.000   Max.   :14053.0  
##      CHits            CHmRun           CRuns             CRBI        
##  Min.   :   4.0   Min.   :  0.00   Min.   :   2.0   Min.   :   3.0  
##  1st Qu.: 212.0   1st Qu.: 15.00   1st Qu.: 105.5   1st Qu.:  95.0  
##  Median : 516.0   Median : 40.00   Median : 250.0   Median : 230.0  
##  Mean   : 722.2   Mean   : 69.24   Mean   : 361.2   Mean   : 330.4  
##  3rd Qu.:1054.0   3rd Qu.: 92.50   3rd Qu.: 497.5   3rd Qu.: 424.5  
##  Max.   :4256.0   Max.   :548.00   Max.   :2165.0   Max.   :1659.0  
##      CWalks       League  Division    PutOuts          Assists      
##  Min.   :   1.0   A:139   E:129    Min.   :   0.0   Min.   :  0.0  
##  1st Qu.:  71.0   N:124   W:134    1st Qu.: 113.5   1st Qu.:  8.0
```

```
##  Median : 174.0                    Median : 224.0   Median : 45.0
##  Mean   : 260.3                    Mean   : 290.7   Mean   :118.8
##  3rd Qu.: 328.5                    3rd Qu.: 322.5   3rd Qu.:192.0
##  Max.   :1566.0                    Max.   :1377.0   Max.   :492.0
##      Errors           Salary       NewLeague
##  Min.   : 0.000   Min.   :  67.5   A:141
##  1st Qu.: 3.000   1st Qu.: 190.0   N:122
##  Median : 7.000   Median : 425.0
##  Mean   : 8.593   Mean   : 535.9
##  3rd Qu.:13.000   3rd Qu.: 750.0
##  Max.   :32.000   Max.   :2460.0
```

```r
# Load the glmnet package
library(glmnet)
# Construct the predictor matrix
# model.matrix() automatically creates dummy variables
# Remove the intercept column using [, -1]
X <- model.matrix(Salary ~ ., data = Hitters)[, -1]
# Define the response variable
y <- Hitters$Salary
```

The `glmnet()` function has an `alpha` argument that determines what type of model is fit. If `alpha = 0` then a ridge regression model is fit, and if `alpha = 1` then a lasso model is fit. We first fit a ridge regression model, which minimizes

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda\sum_{j=1}^{p}\beta_j^2,$$

where $\lambda \geq 0$ is a *tuning parameter* to be determined.

2. Fit Ridge Regression over a grid of λ values

```r
# Create a grid of lambda values
# Large lambda -> strong shrinkage
# Small lambda -> weak shrinkage
grid <- 10^seq(10, -2, length = 100)

# Fit ridge regression models across the lambda grid
ridge.mod <- glmnet(X, y, alpha = 0, lambda = grid)
```

3. Ridge Regression Coefficients

```r
# Display dimensions of the coefficient matrix
# Rows    -> coefficients
# Columns -> lambda values
dim(coef(ridge.mod))
```

```
## [1]  20 100
```

As λ increases, the ridge penalty becomes stronger, causing the coefficient estimates to shrink toward zero.

We first examine the coefficients associated with the 50th value of λ.

```r
# Display the 50th lambda value
ridge.mod$lambda[50]
```

```
## [1] 11497.57
```

```r
# Display coefficients associated with the 50th lambda
coef(ridge.mod)[, 50]
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
## 407.356050200   0.036957182   0.138180344   0.524629976   0.230701523
##           RBI         Walks         Years        CAtBat         CHits
##   0.239841459   0.289618741   1.107702929   0.003131815   0.011653637
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##   0.087545670   0.023379882   0.024138320   0.025015421   0.085028114
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
##  -6.215440973   0.016482577   0.002612988  -0.020502690   0.301433531
```

```r
# Compute the L2 norm of the coefficient vector
# Exclude the intercept term
sqrt(sum(coef(ridge.mod)[-1, 50]^2))
```

```
## [1] 6.360612
```

Next, we examine coefficients corresponding to a smaller value of λ. Because the penalty is weaker, the coefficients tend to have a larger ℓ_2 norm.

```r
# Display the 60th lambda value
ridge.mod$lambda[60]
```

```
## [1] 705.4802
```

```r
# Display coefficients associated with the 60th lambda
coef(ridge.mod)[, 60]
```

```
##  (Intercept)        AtBat         Hits        HmRun         Runs          RBI
##  54.32519950   0.11211115   0.65622409   1.17980910   0.93769713   0.84718546
##        Walks        Years       CAtBat        CHits       CHmRun        CRuns
##   1.31987948   2.59640425   0.01083413   0.04674557   0.33777318   0.09355528
##         CRBI       CWalks      LeagueN    DivisionW      PutOuts      Assists
##   0.09780402   0.07189612  13.68370191 -54.65877750   0.11852289   0.01606037
##       Errors    NewLeagueN
##  -0.70358655   8.61181213
```

```r
# Compute the L2 norm
sqrt(sum(coef(ridge.mod)[-1, 60]^2))
```

```
## [1] 57.11001
```

We can use the `predict()` function for a number of purposes. For instance, we can obtain the ridge regression coefficients for a new value of λ, say 50:

```r
# Obtain ridge regression coefficients for lambda = 50
predict(ridge.mod, s = 50, type = "coefficients")[1:20, ]
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##  4.876610e+01 -3.580999e-01  1.969359e+00 -1.278248e+00  1.145892e+00
##           RBI         Walks         Years        CAtBat         CHits
##  8.038292e-01  2.716186e+00 -6.218319e+00  5.447837e-03  1.064895e-01
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##  6.244860e-01  2.214985e-01  2.186914e-01 -1.500245e-01  4.592589e+01
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -1.182011e+02  2.502322e-01  1.215665e-01 -3.278600e+00 -9.496680e+00
```

4. Training/Testing

We now split the samples into a training set and a test set in order to estimate the test error of ridge regression and later on the lasso.

```r
# Set seed for reproducibility
set.seed(1)

# Randomly select half of the observations for training
train <- sample(1:nrow(X), nrow(X) / 2)

# Remaining observations form the test set
test <- (-train)

# Store test response values
y.test <- y[test]

# ------------------------------------------------------------
# Fit ridge regression on the training data
# ------------------------------------------------------------
ridge.mod <- glmnet(X[train, ], y[train],
                    alpha = 0, lambda = grid, thresh = 1e-12)

# ------------------------------------------------------------
# Predict test responses using lambda = 4
# ------------------------------------------------------------
ridge.pred <- predict(ridge.mod, s = 4, newx = X[test, ])

# Compute test RMSE
sqrt(mean((ridge.pred - y.test)^2))
```

```
## [1] 377.093
```

For comparison, we compute the RMSE from an intercept-only model that predicts the training mean for all test observations.

```r
# RMSE for intercept-only model
sqrt(mean((mean(y[train]) - y.test)^2))
```

```
## [1] 473.9936
```

We now investigate the effect of extremely large and extremely small values of λ.

```
# ------------------------------------------------------------
# Very large lambda:
# coefficients shrink strongly toward zero
# ------------------------------------------------------------
ridge.pred <- predict(ridge.mod, s = 1e10, newx = X[test, ])

sqrt(mean((ridge.pred - y.test)^2))
```

```
## [1] 473.9935
```

```
# ------------------------------------------------------------
# Lambda = 0:
# ridge regression becomes ordinary least squares
# ------------------------------------------------------------
ridge.pred <- predict(ridge.mod, s = 0, newx = X[test, ])

sqrt(mean((ridge.pred - y.test)^2))
```

```
## [1] 409.6215
```

5. Cross-Validation (CV)

Rather than arbitrarily selecting a value of λ, we can use *cross-validation* to choose the tuning parameter automatically.

The function `cv.glmnet()` performs *K-fold cross-validation*. By default, it uses 10-fold CV.

```
# Set seed for reproducibility
set.seed(1)

# ------------------------------------------------------------
# Perform cross-validation for ridge regression
# ------------------------------------------------------------
cv.out <- cv.glmnet(X[train, ], y[train], alpha = 0)

# Display the lambda value that minimizes CV error
(bestLambda <- cv.out$lambda.min)
```

```
## [1] 326.0828
```

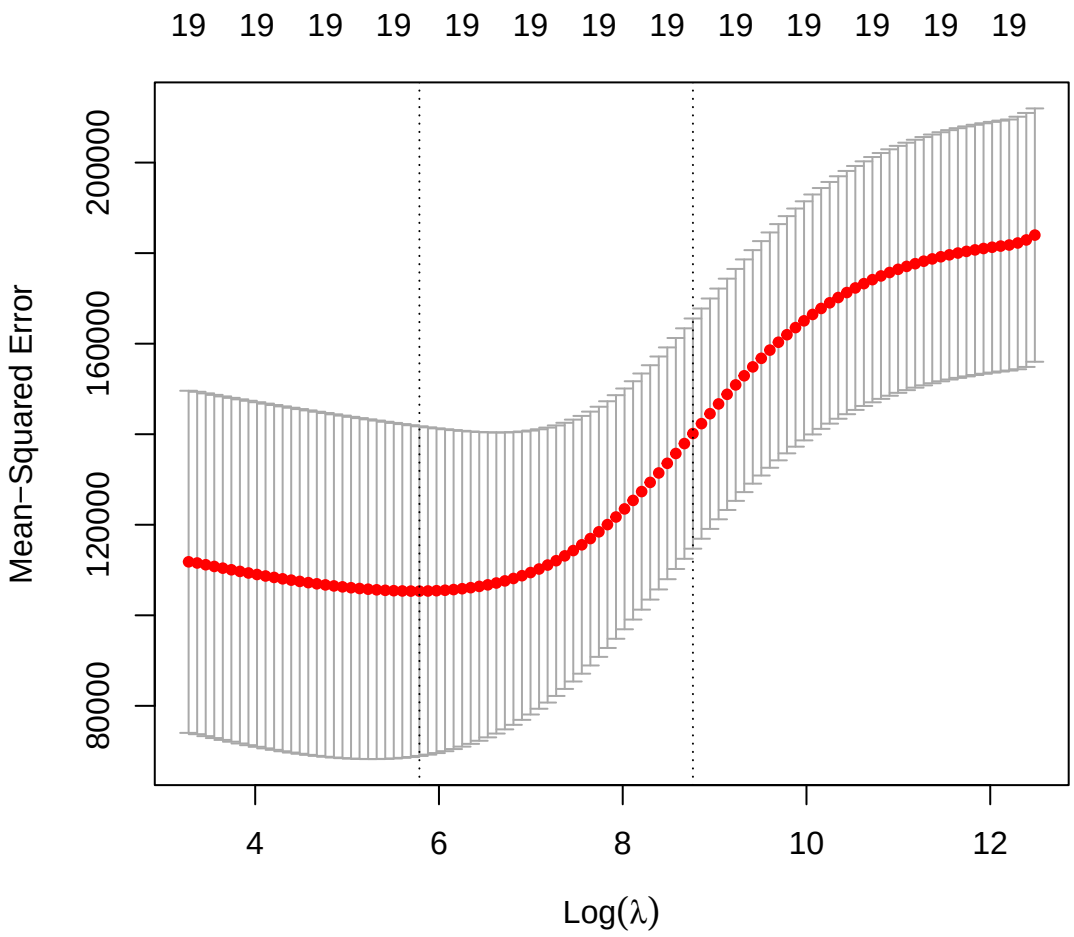
We now evaluate prediction performance using the selected value of λ.

```
# Predict on the test set using the optimal lambda
ridge.pred <- predict(ridge.mod, s = bestLambda, newx = X[test, ])

# Compute test RMSE
sqrt(mean((ridge.pred - y.test)^2))
```

```
## [1] 373.9741
```

The following plot displays the cross-validation error as a function of λ.

```
# Plot CV error curve
plot(cv.out)
```



Finally, we refit the ridge regression model using the full dataset and the optimal value of λ.

```
# Fit ridge regression on the full dataset
out <- glmnet(X, y, alpha = 0)

# Display coefficient estimates using the optimal lambda
predict(out, type = "coefficients", s = bestLambda)[1:20, ]
```

```
##  (Intercept)        AtBat         Hits        HmRun         Runs          RBI
##  15.44383120   0.07715547   0.85911582   0.60103106   1.06369007   0.87936105
##        Walks        Years       CAtBat        CHits       CHmRun        CRuns
##   1.62444617   1.35254778   0.01134999   0.05746654   0.40680157   0.11456224
##         CRBI       CWalks      LeagueN    DivisionW      PutOuts      Assists
##   0.12116504   0.05299202 22.09143197 -79.04032656   0.16619903   0.02941950
##       Errors   NewLeagueN
##  -1.36092945   9.12487765
```

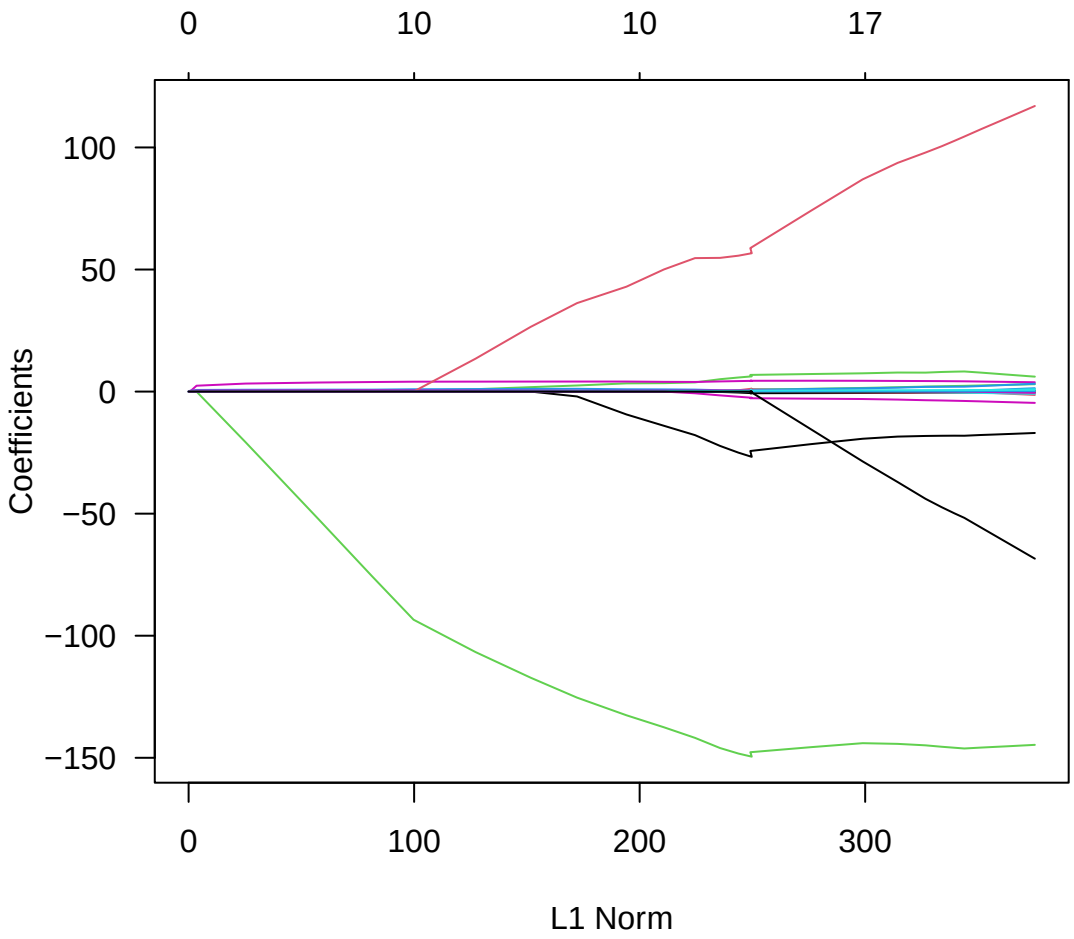
The Lasso

We saw that ridge regression with a wise choice of λ can outperform least squares as well as the null model on the Hitters data set. We now ask whether the lasso, which minimizes

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda \sum_{j=1}^{p}|\beta_j|$$

can yield either a more accurate or a more interpretable model than ridge regression. In order to fit a lasso model, we once again use the `glmnet()` function; however, this time we use the argument `alpha=1`.

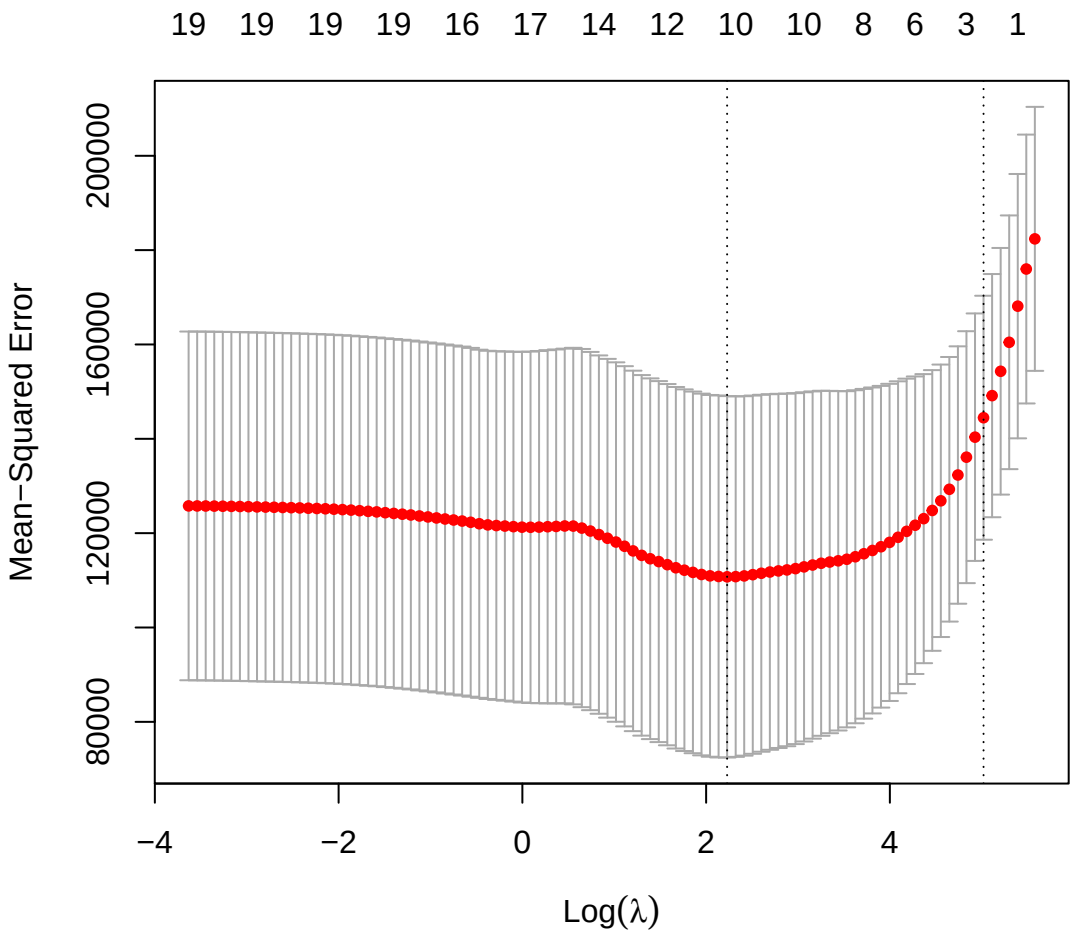
```
# ---------------------------------------------------------
# Fit lasso regression models over the lambda grid
# alpha = 1 specifies lasso regression
# ---------------------------------------------------------
lasso.mod <- glmnet(X[train, ], y[train],
                    alpha = 1, lambda = grid)

# Plot coefficient paths as lambda changes
# Each curve corresponds to one predictor coefficient
plot(lasso.mod, las = 1)
```



Notice from the coefficient path plot that, depending on the value of the tuning parameter λ, some coefficient estimates become exactly zero. This is one of the major differences between the lasso and ridge regression.

We now use cross-validation to select the optimal value of λ and evaluate prediction performance on the test set.

```
# Set seed for reproducibility
set.seed(1)

# ------------------------------------------------------------
# Perform cross-validation for lasso regression
# ------------------------------------------------------------
cv.out <- cv.glmnet(X[train, ], y[train], alpha = 1)

# Plot cross-validation error versus lambda
plot(cv.out)
```




```
# Extract the lambda value that minimizes CV error
bestLambda <- cv.out$lambda.min

# ------------------------------------------------------------
# Predict test responses using the selected lambda
# ------------------------------------------------------------
lasso.pred <- predict(lasso.mod, s = bestLambda, newx = X[test, ])

# Compute test RMSE
sqrt(mean((lasso.pred - y[test])^2))
```

```
## [1] 379.043
```

The resulting test RMSE is substantially lower than that of the null model and ordinary least squares, and is very similar to the test RMSE obtained from ridge regression with cross-validated λ.

However, the lasso has an important advantage over ridge regression: the fitted model is sparse and therefore easier to interpret. Some coefficients are shrunk exactly to zero, effectively removing unimportant predictors from the model.

```
# ---------------------------------------------------------
# Fit lasso model on the full dataset
# ---------------------------------------------------------
out <- glmnet(X, y, alpha = 1, lambda = grid)

# Display coefficient estimates using the optimal lambda
(lasso.coef <- predict(out, type = "coefficients", s = bestLambda)[1:20, ])
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##    1.27479059   -0.05497143    2.18034583    0.00000000    0.00000000
##           RBI         Walks         Years        CAtBat         CHits
##    0.00000000    2.29192406   -0.33806109    0.00000000    0.00000000
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##    0.02825013    0.21628385    0.41712537    0.00000000   20.28615023
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -116.16755870    0.23752385    0.00000000   -0.85629148    0.00000000
```

```
# Display only the nonzero coefficient estimates
lasso.coef[lasso.coef != 0]
```

```
##   (Intercept)         AtBat          Hits         Walks         Years
##    1.27479059   -0.05497143    2.18034583    2.29192406   -0.33806109
##        CHmRun         CRuns          CRBI       LeagueN     DivisionW
##    0.02825013    0.21628385    0.41712537   20.28615023 -116.16755870
##       PutOuts        Errors
##    0.23752385   -0.85629148
```

In this example, several coefficient estimates are exactly zero, illustrating the variable selection capability of the lasso.