Ridge Regression and the Bias-variance Trade-off

Ruoqing Zhu

Last Updated: October 06, 2024

Contents

Ridge Regression

Ridge regression was proposed by Hoerl and Kennard (1970), but is also a special case of the Tikhonov regularization. The essential idea is very simple: Knowing that the ordinary least squares (OLS) solution is not unique in an ill-posed problem, i.e., $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is not invertible, a ridge regression adds a ridge (diagonal matrix) on $\mathbf{X}^{\mathrm{T}}\mathbf{X}$:

$$\widehat{\boldsymbol{\beta}}^{\text{ridge}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X} + n\lambda\mathbf{I})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y},$$

It provides a solution of linear regression when multicollinearity happens, especially when the number of variables is larger than the sample size. Alternatively, this is also the solution of a regularized least square estimator. We add an ℓ_2 penalty to the residual sum of squares, i.e.,

$$\widehat{\boldsymbol{\beta}}^{\text{ridge}} = \arg\min_{\boldsymbol{\beta}} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 + n\lambda\|\boldsymbol{\beta}\|_2^2 \tag{1}$$

$$= \arg\min_{\boldsymbol{\beta}} \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^{\mathrm{T}}\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}\beta_j^2, \tag{2}$$

for some penalty $\lambda > 0$. Another approach that leads to the ridge regression is a constraint on the ℓ_2 norm of the parameters, which will be introduced in the next week. Ridge regression is used extensively in genetic analyses to address "small-n-large-p" problems. We will start with a motivation example and then discuss the crucial topic this week: the bias-variance trade-off.

Motivation: Correlated Variables, Convexity and Variance

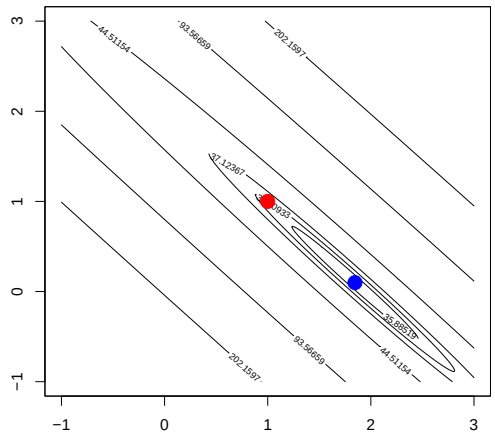
Ridge regression has many advantages. Most notably, it can address highly correlated variables. From an optimization point of view, having highly correlated variables means that the objective function (ℓ_2 loss) becomes "flat" along certain directions in the parameter domain. This can be seen from the following example, where the true parameters are both 1 while the estimated parameters concludes almost all effects to the first variable. You can change different seed to observe the variability of these parameter estimates and notice that they are quite large. Instead, if we fit a ridge regression, the parameter estimates are relatively stable.

```
  library(MASS)
  set.seed(2)
  n = 30

  # create highly correlated variables and a linear model
  X = mvrnorm(n, c(0, 0), matrix(c(1,0.99, 0.99, 1), 2,2))
  y = rnorm(n, mean = X[,1] + X[,2])

  # compare parameter estimates
  summary(lm(y~X-1))$coef
##     Estimate Std. Error    t value  Pr(>|t|)
## X1 1.8461255   1.294541 1.42608527 0.1648987
## X2 0.0990278   1.321283 0.07494822 0.9407888

  # note that the true parameters are all 1's
  # Be careful that the `lambda` parameter in lm.ridge is our (n*lambda)
  lm.ridge(y~X-1, lambda=5)
##        X1        X2
## 0.9413221 0.8693253
```

The variance of both β_1 and β_2 are quite large. This is expected because we know from linear regression that the variance of $\widehat{\boldsymbol{\beta}}$ is $\sigma^2(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$. However, since the columns of $\mathbf{X}$ are highly correlated, the smallest eigenvalue of $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is close to 0, making the largest eigenvalue of $(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$ very large. This can also be interpreted through an optimization point of view. The objective function for an OLS estimator is demonstrated in the following.

```
  beta1 <- seq(-1, 3, 0.005)
  beta2 <- seq(-1, 3, 0.005)
  allbeta <- data.matrix(expand.grid(beta1, beta2))
  rss <- matrix(apply(allbeta, 1, function(b, X, y) sum((y - X %*% b)^2), X, y),
                length(beta1), length(beta2))

  # quantile levels for drawing contour
  quanlvl = c(0.01, 0.025, 0.05, 0.2, 0.5, 0.75)

  # plot the contour
  contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
  box()

  # the truth
  points(1, 1, pch = 19, col = "red", cex = 2)

  # the data
```

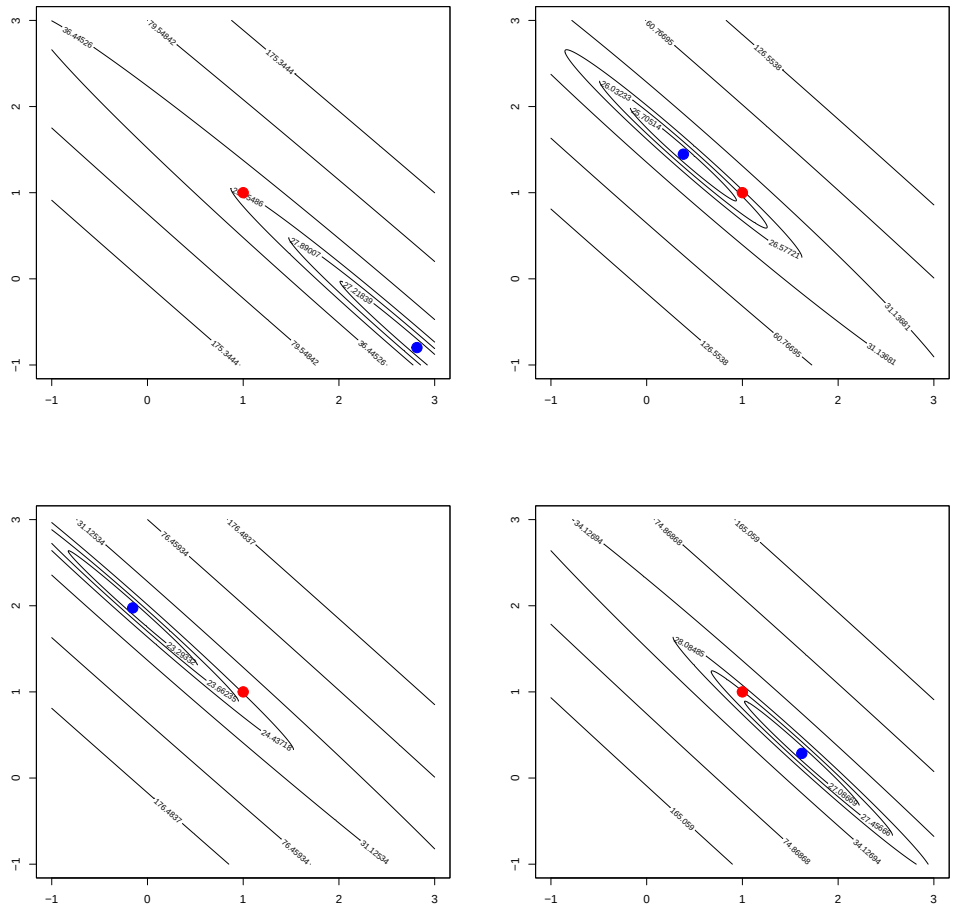
```r
betahat <- coef(lm(y~X-1))
points(betahat[1], betahat[2], pch = 19, col = "blue", cex = 2)
```



We can see a relatively "flat" valley in this objective function. This is because the (true) covariance matrix of X has a very small eigen-value. Since the variance of $\boldsymbol{\beta}$ is $\sigma^2(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}$, a small eigen-value in $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ makes the corresponding eigen-value large in the inverse.

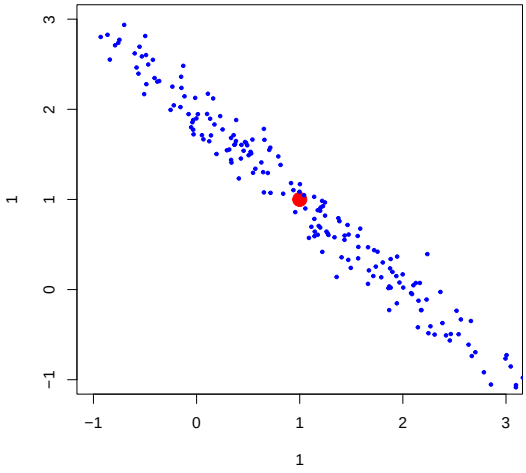
```r
    # eigen structure of X^T X
    eigen(matrix(c(1,0.99, 0.99, 1), 2,2))
## eigen() decomposition
## $values
## [1] 1.99 0.01
##
## $vectors
##           [,1]       [,2]
## [1,] 0.7071068 -0.7071068
## [2,] 0.7071068  0.7071068

    # eigen structure of (X^T X)^{-1}
    eigen(solve(matrix(c(1,0.99, 0.99, 1), 2,2)))
## eigen() decomposition
## $values
## [1] 100.0000000   0.5025126
##
## $vectors
##            [,1]       [,2]
## [1,] -0.7071068 -0.7071068
## [2,]  0.7071068 -0.7071068
```

Each time we observe a set of data, they represent some perturbed version of such an objective function. They lie on a valley centered around the truth.



Overall, this makes the solution unstable since the optimizer could land anywhere if we observe a different set of data. This can be viewed through a simulation study. This property is interpreted as the **variance of an estimator**.

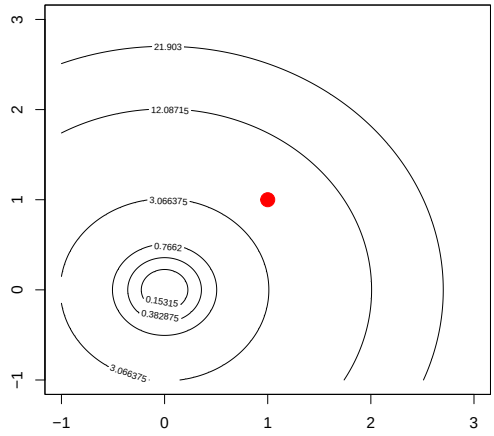
```
# the truth
plot(1, 1, xlim = c(-1, 3), ylim = c(-1, 3),
     pch = 19, col = "red", cex = 2)

# generate many datasets in a simulation
for (i in 1:200)
{
  X = mvrnorm(n, c(0, 0), matrix(c(1,0.99, 0.99, 1), 2,2))
  y = rnorm(n, mean = X[,1] + X[,2])

  betahat <- coef(lm(y~X-1))
  points(betahat[1], betahat[2], pch = 19, col = "blue", cex = 0.5)
}
```



Ridge Penalty and the Reduced Variation

As an alternative, if we add a ridge regression penalty, the contour is forced to be more convex due to the added eigenvalues. Here is a plot of the Ridge ℓ_2 penalty

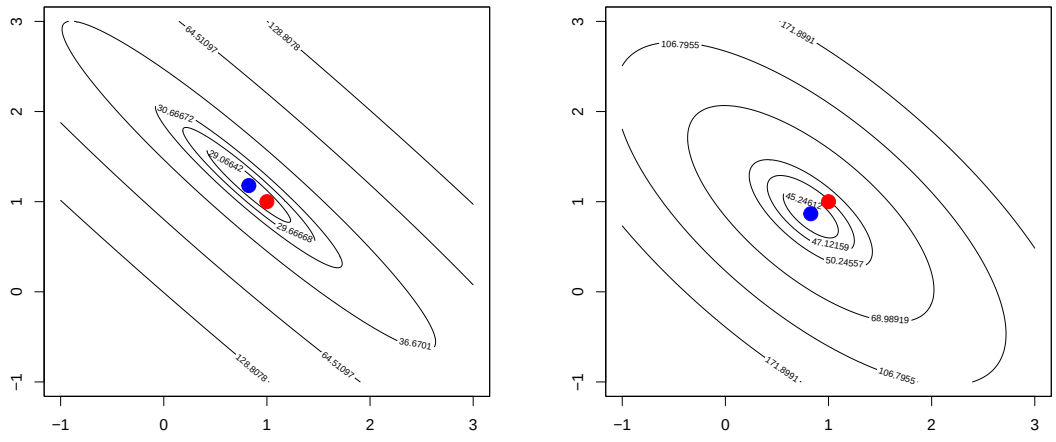
$$\lambda\|\boldsymbol{\beta}\|^2 = \lambda\boldsymbol{\beta}^{\mathrm{T}}\mathbf{I}\boldsymbol{\beta}$$



Hence, by adding this to the OLS objective function, the solution is more stable, in the sense that each time we observe a new set of data, this contour looks pretty much the same. This may be interpreted in several different ways such as: 1) the objective function is more convex and less affected by the random samples; 2) the variance of the estimator is smaller because the eigenvalues of $\mathbf{X}^{\mathrm{T}}\mathbf{X} + n\lambda\mathbf{I}$ are large.

```
par(mfrow=c(1, 2))
```

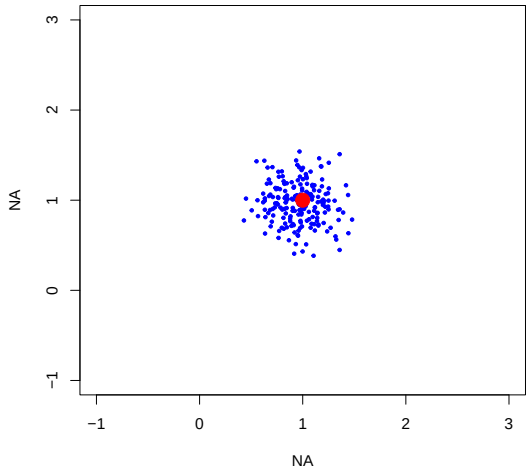
```r
# adding a L2 penalty to the objective function
rss <- matrix(apply(allbeta, 1, function(b, X, y) sum((y - X %*% b)^2) + b %*% b, X, y),
              length(beta1), length(beta2))

# the ridge solution
bh = solve(t(X) %*% X + diag(2)) %*% t(X) %*% y

contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
points(1, 1, pch = 19, col = "red", cex = 2)
points(bh[1], bh[2], pch = 19, col = "blue", cex = 2)
box()

# adding a larger penalty
rss <- matrix(apply(allbeta, 1, function(b, X, y) sum((y - X %*% b)^2) + 10*b %*% b, X, y),
              length(beta1), length(beta2))

bh = solve(t(X) %*% X + 10*diag(2)) %*% t(X) %*% y

# the ridge solution
contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
points(1, 1, pch = 19, col = "red", cex = 2)
points(bh[1], bh[2], pch = 19, col = "blue", cex = 2)
box()
```



We can check the ridge solution over many simulation runs

```r
# the truth
plot(NA, NA, xlim = c(-1, 3), ylim = c(-1, 3))

# generate many datasets in a simulation
for (i in 1:200)
{
  X = mvrnorm(n, c(0, 0), matrix(c(1,0.99, 0.99, 1), 2,2))
  y = rnorm(n, mean = X[,1] + X[,2])
```

```r
  # betahat <- solve(t(X) %*% X + 2*diag(2)) %*% t(X) %*% y
  betahat <- lm.ridge(y ~ X - 1, lambda = 2)$coef
  points(betahat[1], betahat[2], pch = 19, col = "blue", cex = 0.5)
}

points(1, 1, pch = 19, col = "red", cex = 2)
```



A Biased Estimator

However, this causes some **bias** too. Since we know the OLS estimator is unbiased, adding a ridge penalty would change the expected value. We have not formally discussed the bias, but one of the quantity to best illustrate this is the variance estimation (biased vs. unbiased in HW1). We know that an unbiased estimation of σ^2 is $\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2$, while an MLE estimator $\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2$ is biased. There is of course derivation of this commonly known fact, but let's introduce the concept of a simulation study, which we can use in the future. Consider many researchers and each of them samples $n = 3$ observations to estimate σ^2, and they are all using the biased formula. On average, what would happen? Consider the following code. I am writing it in the most naive way without considering any computational efficiency.

```r
set.seed(1)
# number of researchers
nsim = 1000
# number of observations
n = 3

# define a function to calculate the biased variance estimation
biasedsigma2 <- function(x) sum((x - mean(x))^2)/length(x)

# define a function to calculate the unbiased variance estimation
unbiasedsigma2 <- function(x) sum((x - mean(x))^2)/(length(x) - 1)

# save all estimated variance in a vector
allbiased = rep(NA, nsim)
allunbiased = rep(NA, nsim)
```

```r
  # generate 3 observations for each researcher and
  # record their biased estimation
  for (i in 1:nsim)
  {
    datai = rnorm(n)
    allbiased[i] = biasedsigma2(datai)
    allunbiased[i] = unbiasedsigma2(datai)
  }

  # the averaged of all of them
  mean(allbiased)
## [1] 0.7113691
  mean(allunbiased)
## [1] 1.067054
```

On average, the researchers using the biased estimation estimate the σ^2 to be 0.7114. Since the true variance is 1, this is obviously biased. With the same data, the unbiased estimation is 1.0670. We do need to consider that the variation across different researchers is quite large, however, we know from the theory that our conclusion should be correct.

Bias Caused by the Ridge Penalty

Now, let's apply the same analysis on the ridge regression estimator. For the theoretical justification of this analysis, please read the SMLR textbook. We will set up a simulation study with the following steps, with both $\widehat{\boldsymbol{\beta}}^{\text{ridge}}$ and $\widehat{\boldsymbol{\beta}}^{\text{ols}}$:

1) Generate a set of $n = 100$ observations
2) Estimate the ridge estimator $\widehat{\boldsymbol{\beta}}^{\text{ridge}}$ with $\lambda = 0.3$. Hence, $n\lambda = 30$.
3) Repeat steps 1) and 2) nsim $= 200$ runs
4) Average all estimations and compare that with the truth $\boldsymbol{\beta}$
5) Compute the variation of these estimates across all runs

```r
  set.seed(1)
  # number of researchers
  nsim = 200
  # number of observations
  n = 100

  # lambda
  lambda = 0.3

  # save all estimated variance in a vector
  allridgebeta = matrix(NA, nsim, 2)
  allolsbeta = matrix(NA, nsim, 2)

  for (i in 1:nsim)
  {
    # create highly correlated variables and a linear model
    X = mvrnorm(n, c(0, 0), matrix(c(1,0.99, 0.99, 1), 2,2))
    y = rnorm(n, mean = X[,1] + X[,2])
```

```
    allridgebeta[i, ] = solve(t(X) %*% X + lambda*n*diag(2)) %*% t(X) %*% y
    allolsbeta[i, ] = solve(t(X) %*% X) %*% t(X) %*% y
  }

  # compare the mean of these estimates
  colMeans(allridgebeta)
## [1] 0.8675522 0.8677563
  colMeans(allolsbeta)
## [1] 1.0081553 0.9895799

  # compare the var of these estimates
  apply(allridgebeta, 2, var)
## [1] 0.002577951 0.002542013
  apply(allolsbeta, 2, var)
## [1] 0.4845030 0.4785982
```

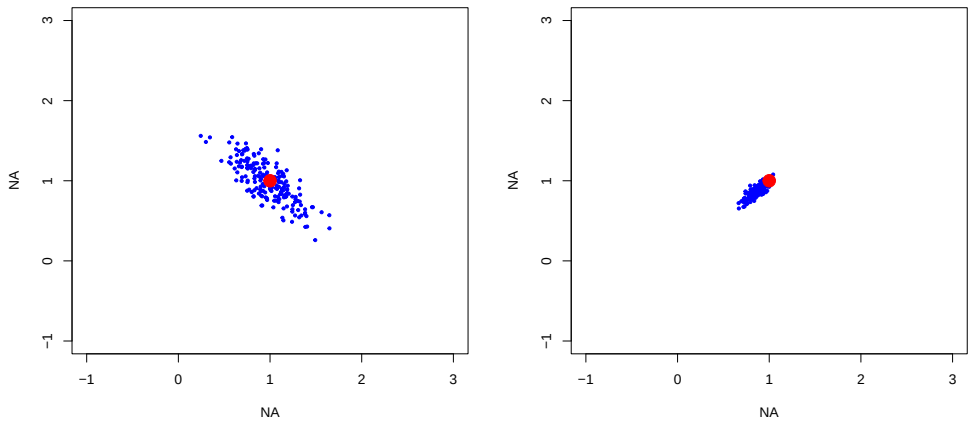
We can easily see that the ridge estimations are $(0.8646, 0.8658)$ respectively, which is biased from the truth $(1, 1)$. The OLS estimator is still unbiased, however, their variation $(0.5314, 0.5275)$ is huge compared with the ridge estimations $(0.0028, 0.0027)$.

The Bias-variance Trade-off

This effect is gradually changing as we increase the penalty level. The following simulation shows how the variation of $\boldsymbol{\beta}$ changes. We show this with two penalty values, and see how the estimated parameters are away from the truth.

```
    par(mfrow = c(1, 2))

  # the truth
  plot(NA, NA, xlim = c(-1, 3), ylim = c(-1, 3))

  # generate many datasets in a simulation
  for (i in 1:200)
  {
    X = mvrnorm(n, c(0, 0), matrix(c(1,0.99, 0.99, 1), 2,2))
    y = rnorm(n, mean = X[,1] + X[,2])

    betahat <- lm.ridge(y ~ X - 1, lambda = 2)$coef
    points(betahat[1], betahat[2], pch = 19, col = "blue", cex = 0.5)
  }

  points(1, 1, pch = 19, col = "red", cex = 2)

  # a plot
  plot(NA, NA, xlim = c(-1, 3), ylim = c(-1, 3))

  # generate many datasets in a simulation
  for (i in 1:200)
  {
    X = mvrnorm(n, c(0, 0), matrix(c(1,0.99, 0.99, 1), 2,2))
    y = rnorm(n, mean = X[,1] + X[,2])
```

```r
  # betahat <- solve(t(X) %*% X + 30*diag(2)) %*% t(X) %*% y
  betahat <- lm.ridge(y ~ X - 1, lambda = 30)$coef
  points(betahat[1], betahat[2], pch = 19, col = "blue", cex = 0.5)
}

points(1, 1, pch = 19, col = "red", cex = 2)
```



In general, the penalization leads to a biased estimator (since the OLS estimator is unbiased) if we use any nonzero λ. Choosing the tuning parameter is a balance of the bias-variance trade-off.

- As $\lambda \to 0$, the ridge solution is eventually the same as OLS
- As $\lambda \to \infty$, $\widehat{\boldsymbol{\beta}}^{\text{ridge}} \to 0$

On the other hand, the variation of $\widehat{\boldsymbol{\beta}}^{\text{ridge}}$ decreases as λ increases. Hence, there is a **bias-variance trade-off**:

- As $\lambda \downarrow$ decrease, bias $\downarrow$ decrease and variance $\uparrow$ increases
- As $\lambda \uparrow$ increases, bias $\uparrow$ increases and variance $\downarrow$ decrease

This will be an issue for most if not all machine learning models. In fact, the overall effect for estimating $\boldsymbol{\beta}$ can be explained as

$$\text{Bias}^2 + \text{Variance}$$

Now, we may ask the question: is it worth it? In fact, this bias and variance will be then carried to the predicted values $x^{\mathrm{T}}\widehat{\boldsymbol{\beta}}^{\text{ridge}}$. Hence, we can judge if this is beneficial from the prediction accuracy. And we need some procedure to do this.

Remark: The bias-variance trade-off will appear frequently in this course. And the way to evaluate the benefit of this is to see if it eventually reduces the prediction error (Bias^2 + Variance plus a term called **irreducible error**, which will be introduced in later chapter).

Using the `lm.ridge()` function

We have seen how the `lm.ridge()` can be used to fit a Ridge regression. However, keep in mind that the `lambda` parameter used in the function actually specifies the $n\lambda$ entirely we used in our notation. Regardless, our goal is mainly to tune this parameter to achieve a good balance of bias-variance trade off and good perdition error. The difficulty here is to evaluate the performance without knowing the truth. Let's first use a simulated example, in which we do know the truth and then introduce the cross-validation approach for real data where we do not know the truth.

We use the Motor Trend Car Road Tests (`mtcars`) dataset. The goal of this dataset is to predict the mpg using various features of a car.

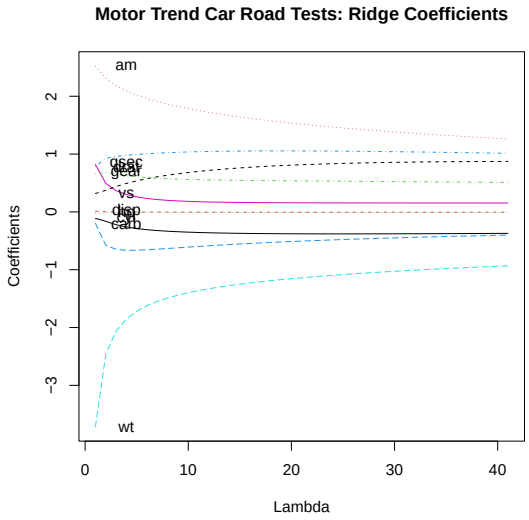
```
  data(mtcars)
  head(mtcars)
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

  # lm.ridge function from the MASS package
  library(MASS)
  lm.ridge(mpg ~., data = mtcars, lambda = 1)
##                      cyl         disp           hp         drat           wt         qsec
## 16.537660830 -0.162402755  0.002333078 -0.014934856  0.924631319 -2.461146015  0.492587517  0.374651
```

We can also specify multiple λ values:

```
  library(MASS)
  fit1 = lm.ridge(mpg ~., data = mtcars, lambda = seq(0, 40, by=1))
```

You must use the `coef()` function to obtain the fitted coefficients. However, be careful that this is different than using `$coef` from the fitted object. For details of this issue,please see the SMLR text book.

```
    matplot(coef(fit1)[, -1], type = "l", xlab = "Lambda", ylab = "Coefficients")
    text(rep(4, 10), coef(fit1)[1,-1], colnames(mtcars)[2:11])
    title("Motor Trend Car Road Tests: Ridge Coefficients")
```



To select the best λ value, there can be several different methods. But the key issue is that we need some testing data which are independent from the training data. For this idea, we will discuss k-fold cross-validation. In the linear model setting, there is also another approach called the generalized cross-validation (GCV).

Practice Question

Use the first 24 observations from the `mtcars` data as the training set, and use the others as the testing set. Fit a ridge regression using the `lm.ridge()` function with $\lambda = 3$. Based on the definition $x^{\mathrm{T}}\widehat{\boldsymbol{\beta}}$, calculate the predicted values of the testing data and report the prediction error.

```
traindata = mtcars[1:24, ]
testdata = mtcars[-(1:24), ]

fit = lm.ridge(mpg ~., data = traindata, lambda = 3)
pred = data.matrix(cbind(1, testdata[, -1])) %*% coef(fit)
mean((pred - testdata[, 1])^2)
```

k-fold cross-validation

Cross-validation (CV) is a technique to evaluate the performance of a model on an independent set of data. The essential idea is to separate out a subset of the data and do not use that part during the training, while using it for testing. We can then rotate to or sample a different subset as the testing data. Different cross-validation methods differs on the mechanisms of generating such testing data. **K-fold cross-validation** is probably the the most popular among them. The method works in the following steps:

1. Randomly split the data into K equal portions
2. For each k in $1, \ldots, K$: use the kth portion as the testing data and the rest as training data, obtain the testing error
3. Average all K testing errors

Here is a graphical demonstration of a 10-fold CV:

add image here

There are also many other CV procedures, for example, the **Monte Carlo cross-validation** randomly splits the data into training and testing (instead of fix K portions) each time and repeat the process as many times as we like. The benefit of such procedure is that if this is repeated enough times, the estimated testing error becomes fairly stable, and not affected much by the random mechanism. On the other hand, we can also repeat the entire K-fold CV process many times, then average the errors. This is also trying to reduced the influence of randomness.

Cross-validation can be setup using the `caret` package. However, you should be careful that not all models are available in `caret`, and you should always check the documentation to see how to implement them. For example, if you use `method = "ridge"`, they do not use `lm.ridge` to fit the model, instead, they use a package called `elasticnet`, which can do the same job. However, the definition of parameters may vary. Hence, it is always good to check the main help pages for the package. We will use the caret package later for other models.

```
    library(caret)
## Loading required package: ggplot2
## Loading required package: lattice
    library(elasticnet)
## Loading required package: lars
## Loaded lars 1.3

    # set cross-validation type
    ctrl <- trainControl(method = "cv", number = 10)

    # set tuning parameter range
    lambdaGrid <- data.frame("lambda" = seq(0, 0.4, length = 20))

    # perform cross-validation
    ridge.cvfit <- train(mtcars[, -1], mtcars$mpg,
                         method = "ridge",
                         tuneGrid = lambdaGrid,
                         trControl = ctrl,
                         ## center and scale predictors
                         preProc = c("center", "scale"))
    ridge.cvfit
## Ridge Regression
##
## 32 samples
## 10 predictors
##
## Pre-processing: centered (10), scaled (10)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 30, 28, 28, 29, 29, 29, ...
## Resampling results across tuning parameters:
##
##   lambda      RMSE      Rsquared   MAE
##   0.00000000  3.351929  0.8281178  2.822493
##   0.02105263  2.912045  0.8355756  2.482691
##   0.04210526  2.781361  0.8631187  2.398992
##   0.06315789  2.721107  0.8826360  2.358213
##   0.08421053  2.689978  0.8948057  2.332843
##   0.10526316  2.674866  0.9027020  2.314222
##   0.12631579  2.670220  0.9081052  2.299992
```

```
##   0.14736842  2.673220  0.9119715  2.291663
##   0.16842105  2.682240  0.9148370  2.298623
##   0.18947368  2.696241  0.9170199  2.312089
##   0.21052632  2.714511  0.9187187  2.332895
##   0.23157895  2.736526  0.9200633  2.354645
##   0.25263158  2.761880  0.9211418  2.377123
##   0.27368421  2.790248  0.9220162  2.400170
##   0.29473684  2.821355  0.9227308  2.423661
##   0.31578947  2.854966  0.9233185  2.448980
##   0.33684211  2.890873  0.9238042  2.488918
##   0.35789474  2.928887  0.9242066  2.528845
##   0.37894737  2.968840  0.9245407  2.571743
##   0.40000000  3.010573  0.9248180  2.615566
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was lambda = 0.1263158.
```

Generalized cross-validation

The generalized cross-validation (GCV) is a modified version of the leave-one-out CV (n-fold cross-validation). The interesting fact about leave-one-out CV in the linear regression setting is that we do not need to explicitly fit all leave-one-out models. The details of those derivations is beyond the scope of this course, but can be found here. The GCV criterion is given by

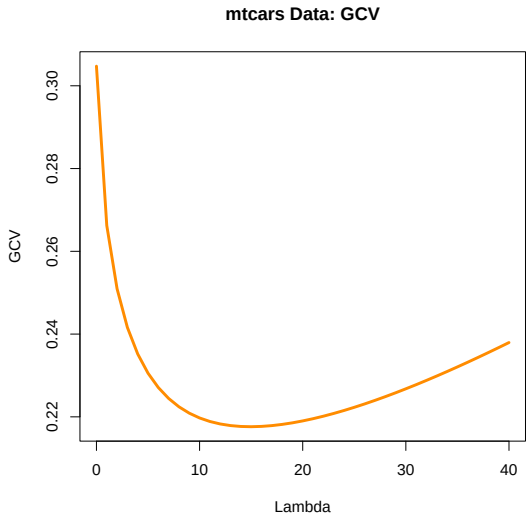
$$\text{GCV}(\lambda) = \frac{\sum_{i=1}^{n}(y_i - x_i^{\text{T}}\widehat{\boldsymbol{\beta}}_{\lambda}^{\text{ridge}})}{(n - \text{Trace}(\mathbf{S}_{\lambda}))}$$

where $\mathbf{S}_{\lambda}$ is the hat matrix corresponding to the ridge regression:

$$\mathbf{S}_{\lambda} = \mathbf{X}(\mathbf{X}^{\text{T}}\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^{\text{T}}$$

The following plot shows how GCV value changes as a function of λ.

```
    # use GCV to select the best lambda
    plot(fit1$lambda[1:100], fit1$GCV[1:100], type = "l", col = "darkorange",
         ylab = "GCV", xlab = "Lambda", lwd = 3)
    title("mtcars Data: GCV")
```



We can select the best λ that produces the smallest GCV.

```
    fit1$lambda[which.min(fit1$GCV)]
## [1] 15
    round(coef(fit1)[which.min(fit1$GCV), ], 4)
##             cyl    disp      hp    drat      wt    qsec      vs      am    gear    carb
## 21.1318 -0.3732 -0.0053 -0.0116  1.0536 -1.2285  0.1615  0.7721  1.6180  0.5441 -0.5459
```

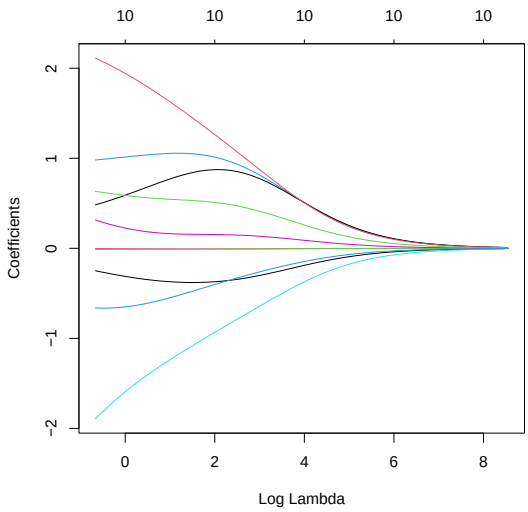
You can clearly see that the GCV decreases initially, as λ increases, this is because the reduced variance is more beneficial than the increased bias. However, as λ increases further, the bias term will eventually dominate and causing the overall prediction error to increase. The fitted MSE under this model is

```
  pred1 = data.matrix(cbind(1, mtcars[, -1])) %*% coef(fit1)[which.min(fit1$GCV), ]
  mean((pred1 - mtcars$mpg)^2)
## [1] 5.389046
```

The glmnet package

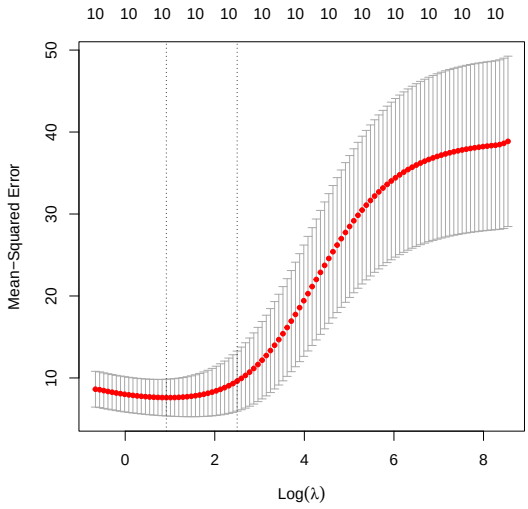
The `glmnet` package implements the k-fold cross-validation. To perform a ridge regression with cross-validation, we need to use the `cv.glmnet()` function with $\alpha = 0$. Here, the α is a parameter that controls the ℓ_2 and ℓ_1 (Lasso) penalties. In addition, the lambda values are also automatically selected, on the log-scale.

```
  library(glmnet)
## Loading required package: Matrix
## Loaded glmnet 4.1-8
  set.seed(3)
  fit2 = cv.glmnet(x = data.matrix(mtcars[, -1]), y = mtcars$mpg, nfolds = 10, alpha = 0)
  plot(fit2$glmnet.fit, "lambda")
```



It is useful to plot the cross-validation error against the λ values , then select the corresponding λ with the smallest error. The corresponding coefficient values can be obtained using the `s = "lambda.min"` option in the `coef()` function. However, this can still be subject to over-fitting, and sometimes practitioners use `s = "lambda.1se"` to select a slightly heavier penalized version based on the variations observed from different folds.

```
plot(fit2)
```




```
coef(fit2, s = "lambda.min")
## 11 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept) 21.171285892
## cyl         -0.368057153
## disp        -0.005179902
## hp          -0.011713150
```

```
## drat         1.053216800
## wt          -1.264212476
## qsec         0.164975032
## vs           0.756163432
## am           1.655635460
## gear         0.546651086
## carb        -0.559817882
  coef(fit2, s = "lambda.1se")
## 11 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept) 19.616654089
## cyl         -0.344649351
## disp        -0.004992867
## hp          -0.008953011
## drat         0.934081832
## wt          -0.787923081
## qsec         0.147331992
## vs           0.852306730
## am           1.071393941
## gear         0.470928251
## carb        -0.329659985
```

The mean prediction error (in this case, the in-sample, training prediction) can be calculated as

```
  pred2 = predict(fit2, newx = data.matrix(mtcars[, -1]), s = "lambda.min")

  # the training error
  mean((pred2 - mtcars$mpg)^2)
## [1] 5.328687
```

This is pretty much the same as the GCV criterion. However, these are both training errors. Can you calculate the testing errors if we separate out a testing data before hand? And would `lambda.1se` perform better than `lambda.min`?

Scaling Issues of Ridge Regression

Both the `lm.ridge()` and `cv.glmnet()` functions will produce a ridge regression solution different from our own naive code. You can try this yourself. This is because they will internally scale all covariates to standard deviation 1 before using the ridge regression formula. The main reason of this is that the bias term will be affected by the scale of the parameter, causing variables with larger variation to be affected less. However, this scale can be arbitrary, and such side effects should be removed. Hence both methods will perform the standardization first. Since this is the practice standard nowadays, we will just rely on the functions to perform these for us and do not worry about the internal operations.