HDS 6. Penalized regression

Matti Pirinen, University of Helsinki

9.1.2024

In HDS5 notes, we considered multiple regression models with different numbers of predictors p. We had two information criteria, AIC and BIC, that find a balance between the model fit to the training data (denoted here simply by $\boldsymbol{y}$), as measured by the likelihood function $L(\boldsymbol{\beta} \,|\, \boldsymbol{y})$, and the flexibility of the model, as measured by a penalty term for the parameters $\boldsymbol{\beta}$. Both criteria are instances of a minimization problem of an **objective function** of the form

$$-2\log(L(\boldsymbol{\beta} \,|\, \boldsymbol{y})) + \lambda \cdot \text{penalty}(\boldsymbol{\beta}),$$

where the penalty function is only a function of the model parameters and $\lambda \geq 0$ is a constant. For example, we get AIC, by setting $\lambda = 2$ and $\text{penalty}(\boldsymbol{\beta}) = \sum_{j=1}^{p} \text{I}(\beta_j)$, where I is an indicator function taking value 1 if $\beta_j \neq 0$ and 0 otherwise.

This week, we will consider a more general family of non-negative penalties of the form

$$\text{penalty}_q(\boldsymbol{\beta}) = \sum_{j=1}^{p} |\beta_j|^q, \; q > 0.$$

We write the penalized objective function as

$$O_{q,\lambda}(\boldsymbol{\beta} \,|\, \boldsymbol{y}) = -2\log(L(\boldsymbol{\beta} \,|\, \boldsymbol{y})) + \lambda \cdot \text{penalty}_q(\boldsymbol{\beta}).$$

The goal is to find the minimum value for this objective function with respect to $\boldsymbol{\beta}$ as a candidate solution that appropriately balances bias and variance of the model.

Note that by agreeing that $0^0 = 0$, we could extend the above framework to include both AIC ($q = 0, \lambda = 2$) and BIC ($q = 0, \lambda = \log(n)$).

Example 6.1. For standard linear regression model the objective function is (up to an additive constant)

$$O_{q,\lambda}(\boldsymbol{\beta} \,|\, \boldsymbol{y}) = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda \cdot \text{penalty}_q(\boldsymbol{\beta}) = \text{RSS} + \lambda \cdot \text{penalty}_q(\boldsymbol{\beta}),$$

and by minimizing this function we will get a compromise between a small residual sum of squares (RSS) and a small penalty term. In other words, we will get a compromise between the ordinary least squares (OLS) estimates and the requirement that the cumulative magnitude of the coefficients is small. The parameter λ determines the importance of each component: When $\lambda = 0$, we repeat the OLS and when $\lambda \to \infty$, we shrink all coefficients to 0. For intermediate values of λ, the solution lies somewhere between OLS and 0.

Because minimizers of $O_{q,\lambda}(\boldsymbol{\beta} \,|\, \boldsymbol{y})$ depend on the scaling of the predictors, as a **pre-processing step**, we standardize the predictors before optimizing for $\boldsymbol{\beta}$. This way the magnitude of the penalty attached to each β-coefficient is equal with respect to the variance explained in the outcome variable.

Typically, the intercept term is not penalized. For linear model, we can leave the intercept β_0 out from the model after the predictors and outcome have been mean-centered. For logistic or Poisson regression models, we do not include the intercept among the coefficients that form the penalty term even though we always have the intercept in the model.

How should we determine the values of q and λ? Let's next consider what kinds of models we will get with commonly used values of $q = 2$ and $q = 1$.

Ridge regression ($q = 2$)

When $q = 2$, the penalty term is the sum of squares and the resulting approach is called **ridge regression**. For this case, we can derive the solution analytically. Let's first expand the objective function:

$$(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}) + \lambda\,\boldsymbol{\beta}^T\boldsymbol{\beta} = \boldsymbol{y}^T\boldsymbol{y} - 2\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\beta}^T\left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\right)\boldsymbol{\beta}.$$

By solving for $\widehat{\boldsymbol{\beta}}_{\text{ridge}}$, that makes the derivative with respect to $\boldsymbol{\beta}$ vanish, we have

$$0 - 2\boldsymbol{y}^T\boldsymbol{X} + 2\widehat{\boldsymbol{\beta}}_{\text{ridge}}^T\left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\right) = 0 \iff \widehat{\boldsymbol{\beta}}_{\text{ridge}} = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}\right)^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

We see that, compared to the least square solution where $\lambda = 0$, the ridge solution adds a positive diagonal term to $(\boldsymbol{X}^T\boldsymbol{X})$ before the matrix inversion. Numerically, this adds stability to the solution and makes computation of the solution possible even when the original matrix $\boldsymbol{X}$ was not of a full rank and the exact inverse $\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}$ did not exist.

Example 6.2. Suppose that the predictors in columns of $\boldsymbol{X}$ are **uncorrelated** and standardized and thus $(\boldsymbol{X}^T\boldsymbol{X}) = n\boldsymbol{I}$. (When the sample variance is used for standardization, the scaling factor would be $n - 1$ rather than n but, for simplicity, we scale by n here.) Then

$$\widehat{\boldsymbol{\beta}}_{\text{ridge}} = (n + \lambda)^{-1}\boldsymbol{X}^T\boldsymbol{y} = \frac{n}{n + \lambda}\widehat{\boldsymbol{\beta}}_{OLS},$$

where $\widehat{\boldsymbol{\beta}}_{OLS} = \boldsymbol{X}^T\boldsymbol{y}/n$ is the least squares solution of $\boldsymbol{y} = \boldsymbol{X}\beta + \boldsymbol{\varepsilon}$. We see that the ridge regression simply shrinks the least squares solution towards 0 by a multiplier that decreases as the ratio λ/n increases.

As we saw in HDS5, an important aspect of the multiple regression model is that it can tease apart the effects of correlated predictors. However, when there are many highly correlated variables in a linear regression model, their coefficients become poorly determined and exhibit high variance, which is reflected in large standard errors of each coefficient. This is because a large positive coefficient on one variable can be canceled by a similarly large negative coefficient on its correlated counterpart. By imposing a penalty term for the joint magnitude of the coefficients, this problem is alleviated. This was a main motivation to consider ridge regression in the first place (Hoerl and Kennard, Technometrics 1970).

Example 6.3. Consider the case of two predictors X_1 and X_2 and outcome variable Y. All three are standardized to have mean 0 and variance 1. Correlation between X_1 and X_2 is r. We want to understand the behavior of the least squares estimator $\widehat{\boldsymbol{\beta}} = (\widehat{\beta}_1, \widehat{\beta}_2)$ of the model $Y = X_1\beta_1 + X_2\beta_2 + \varepsilon$ as a function of r. We know from a course in linear models that the sampling distribution of the effect estimator is

$$\widehat{\boldsymbol{\beta}} \sim \mathcal{N}\left(\boldsymbol{\beta}, \sigma^2\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\right),$$

where $\sigma^2 = \text{Var}(\varepsilon)$ is the error variance. Note that since the variables are standardized, in the observed data, $\boldsymbol{x}_j^T\boldsymbol{x}_j \approx n\,\text{Var}(X_j) = n$ for $j = 1, 2$ and $\boldsymbol{x}_1^T\boldsymbol{x}_2 \approx n\,\text{Cov}(X_1, X_2) = n\,\text{Cor}(X_1, X_2) = n \cdot r$. By using the formula for the inversion of a 2-by-2 matrix, we have that

$$\sigma^2\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1} = \frac{\sigma^2}{n}\begin{pmatrix} 1 & r \\ r & 1 \end{pmatrix}^{-1} = \frac{\sigma^2}{n(1 - r^2)}\begin{pmatrix} 1 & -r \\ -r & 1 \end{pmatrix}.$$

We see that the sampling variance related to each coefficient (either β_1 or β_2) alone is $\propto \frac{1}{n(1-r^2)}$, so will go down as the sample size n grows, but will go up when $|r|$ gets closer to 1. This means that the information to learn the true value of parameter β_1 will grow as n grows but will decrease as $|r|$ grow, and the total effect of these two properties on the amount of information is determined by the product $n(1 - r^2)$. This explains why highly correlated variables in linear model cause unstability to the estimates through large standard errors. However, there is no fundamental reason why linear model could not tell the effects of correlated variables apart from each other, but we simply would need a larger sample size to get the same accuracy (as measured by SEs) for correlated variables than we have for uncorrelated ones already with a smaller

sample size. (To be precise, with correlation r, we will need $1/(1-r^2)$ times the sample size compared to the uncorrelated case to get the same precision for the estimates.)

We also see that the shape of the sampling distribution of $\widehat{\boldsymbol{\beta}}$ has correlation of $-r$, so has the opposite sign compared to the correlation of the original variables X_1 and X_2. This is reflecting that if X_1 and X_2 are highly positively correlated (say $r > 0.99$), then for the model fit, it is almost insignificant how we divide the true total value $B = \beta_1 + \beta_2$ between $\widehat{\beta}_1$ and $\widehat{\beta}_2$. By increasing the first coefficient and simultaneously decreasing the second coefficient by almost the same amount, the model fit stays almost the same. For example, we can get almost the same fitted values by a choice of $\widehat{\beta}_1 = B/2 + 100, \widehat{\beta}_2 = B/2 - 100$ as we get by a choice of $\widehat{\beta}_1 = B/2, \widehat{\beta}_2 = B/2$. Ridge regression stabilizes the estimates of highly correlated variables by favoring the choice of the coefficients that jointly have the shortest (Euclidean) distance from 0.

Questions. How does the 2-dimensional likelihood function of two highly correlated predictors look like? Where does the unstability of parameter estimates manifest in the likelihood function? How does an increasing sample size affect the shape of the likelihood surfaces of constant value and help in bringing more stability to estimates?

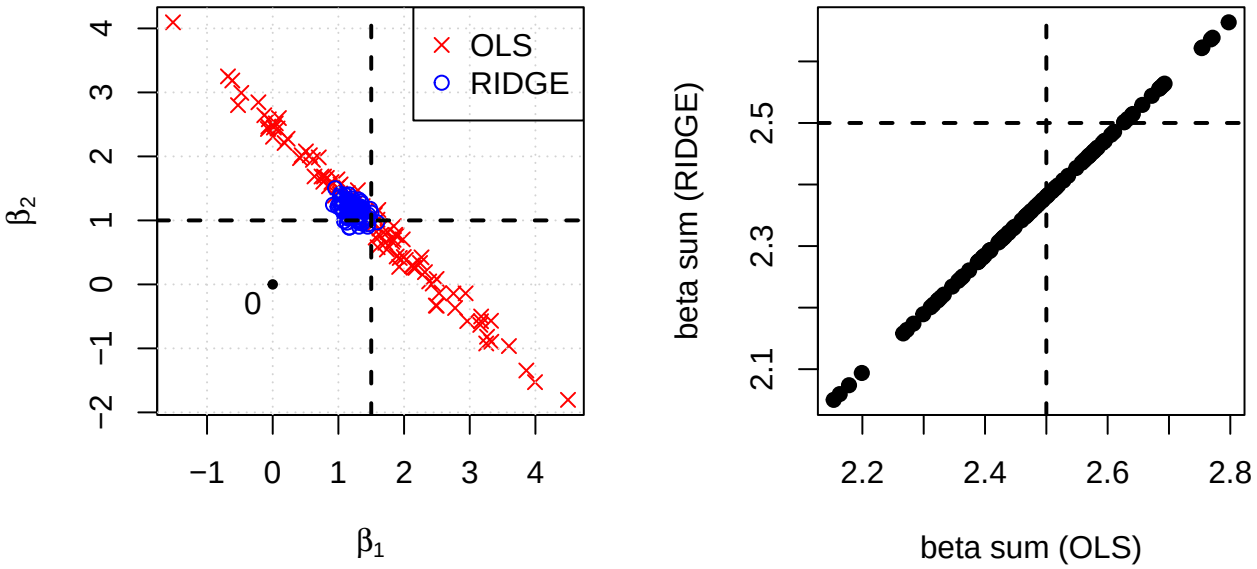
Example 6.4. Let's reuse the generation of a pair of correlated variables from the beginning of HDS5 notes. We generate two highly correlated (0.99) predictors x_1 and x_2 and then generate the outcome $y = x_1 \cdot 1.5 + x_2 \cdot 1.0 + \varepsilon$. The sample size is small ($n = 50$) and so we do not expect to have information to separate well which one of the two variables is having which effect size. Let's see how OLS (corresponding to $\lambda = 0$) and ridge regression (here we set $\lambda = 5$) behave when it comes to estimating the coefficients. We use `lm.ridge()` from `MASS` to fit the ridge regression.

```
library(MASS)
set.seed(17102017)

# function to generate a pair of correlated variables
gen.cor.pair <- function(r, n){ # r is target correlation, n is sample size
 u = rnorm(n, 0, sqrt(abs(r))) # temporary variable that is used for generating x1 and x2
 x1 = scale(u + rnorm(n, 0, sqrt(1 - abs(r))))
 x2 = scale(sign(r)*u + rnorm(n, 0, sqrt(1 - abs(r))))
 cbind(x1, x2) # returns matrix
}
n = 50
X = gen.cor.pair(r = 0.99, n = n) # generate one pair of predictors
cor(X) # check that now cor(x,z) ~ r.
```

```
##           [,1]      [,2]
## [1,] 1.0000000 0.9890795
## [2,] 0.9890795 1.0000000
```

```
true.b = c(1.5, 1.0) # true values of coeffs for cols of X

R = 100 # replications data generation. Predictors are fixed but outcome sampled for each replication
OLS.res = matrix(NA, ncol = 2, nrow = R) # to store least squares estimates across data sets
ridge.res = matrix(NA, ncol = 2, nrow = R) # to store ridge estimates across data sets

for(ii in 1:R){
  y = as.numeric( X %*% true.b + rnorm(n,0,1) )
  y = y - mean(y) # mean-center so that can ignore the intercept
  ols.fit = lm(y ~ 0 + X)
  OLS.res[ii,] = as.numeric(ols.fit$coeff) # coefficient from ordinary least squares fit
```

```
  ridge.fit = lm.ridge( y ~ 0 + X, lambda = 5 ) # ridge regression with lambda = 5 using lm.ridge()
  ridge.res[ii,] = as.numeric(coefficients(ridge.fit))
}

# plot estimates across replications and mark the true values by dotted lines
par(mfrow = c(1, 2))
plot(OLS.res, col = "red", pch = 4, xlab = expression(beta[1]), ylab = expression(beta[2]))
points(ridge.res, col = "blue", pch = 1)
grid()
abline(h = true.b[2], lty = 2, lwd = 2)
abline(v = true.b[1], lty = 2, lwd = 2)
points(0, 0, pch = 19, cex = 0.6)
text(-0.3, -0.3, "0") # mark the origin
legend("topright", leg = c("OLS", "RIDGE"), col = c("red","blue"), pch = c(4,1))

# plot the sum of two coefficients across the methods and mark the true values by dotted line
plot(rowSums(OLS.res), rowSums(ridge.res), pch = 19, xlab = "beta sum (OLS)",
     ylab = "beta sum (RIDGE)")
abline(h = sum(true.b), lty = 2, lwd = 2)
abline(v = sum(true.b), lty = 2, lwd = 2)
```

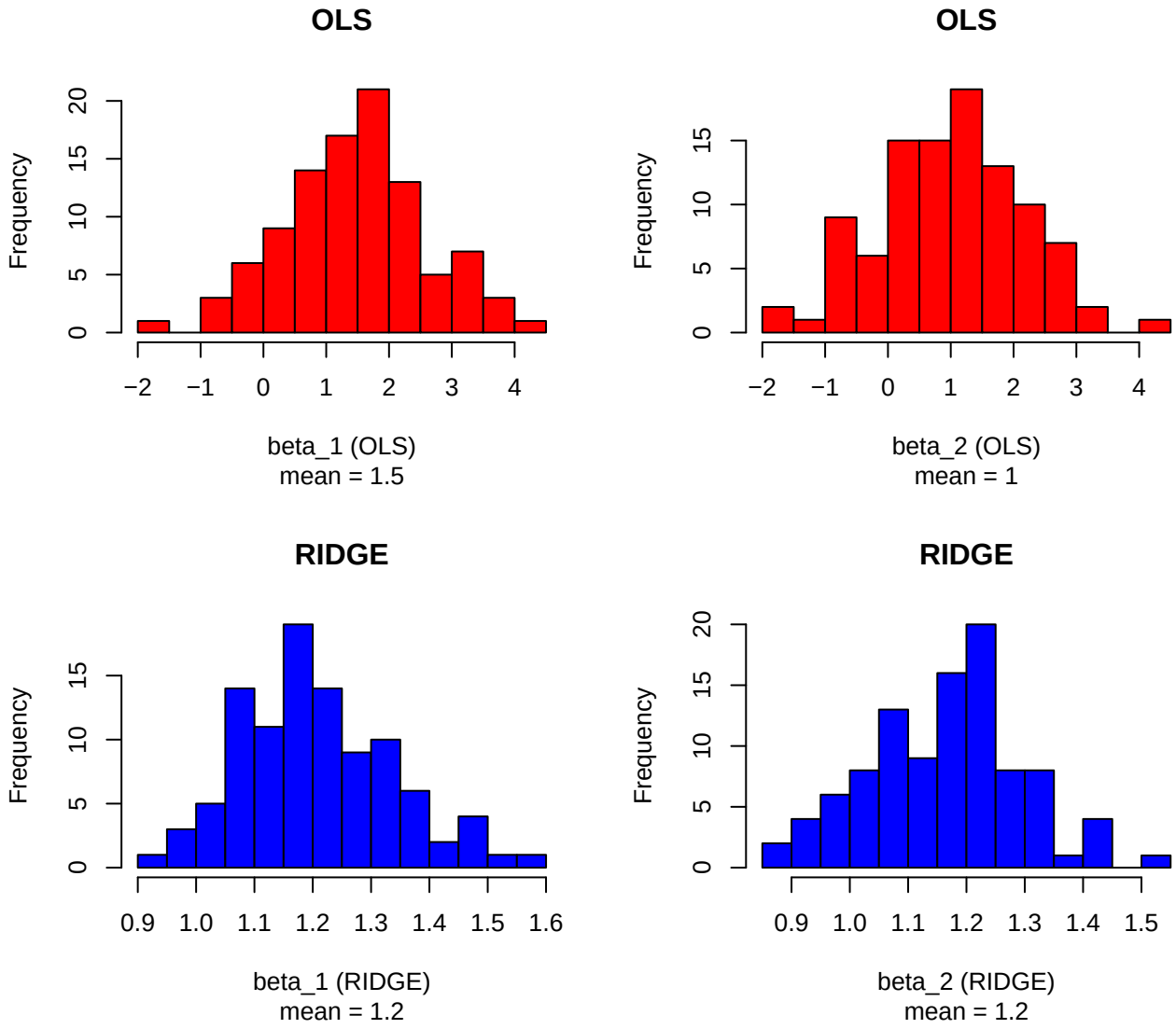


We see that the OLS estimates are unstable and vary widely across the replications along the line $\beta_1+\beta_2 \approx 2.5$ whereas the ridge regression estimates are much more stable and concentrate near to the closest point between the above mentioned line and the origin, because that point minimizes the ridge objective function among all points with the same distance from the origin (panel top left). This behavior is a clear difference between OLS and ridge regression.

With a small amount of data and highly correlated predictors, we do not have much information to split accurately the effect between the two variables. Let's instead look how we estimate the sum of the effects of the two variables (panel top right). We see that the OLS solution has, on average, the correct sum of the coefficients (2.5 is in the middle of estimates) whereas ridge regression is underestimating the sum of βs (the average is below 2.5). Let's have a closer look at the estimates of the individual coefficients.

```
par(mfrow = c(2, 2))
for(ii in 1:2){
```

```
  res =  OLS.res[,ii]
  hist(res, breaks = 15, col = "red",
     xlab = paste0("beta_",ii," (OLS)"), main = "OLS",
     sub = paste("mean =",signif(mean(res), 2)))}

for(ii in 1:2){
  res =  ridge.res[,ii]
  hist(res, breaks = 15, col = "blue",
     xlab = paste0("beta_",ii," (RIDGE)"), main = "RIDGE",
     sub = paste("mean =",signif(mean(res), 2)))}
```



We know that OLS is unbiased, that is, its mean converges to the correct value across simulations (red histograms). Ridge regression, instead, is biased (blue histograms). By penalizing likelihood we cause shrinkage to the coefficients towards each other and towards 0. This bias is the price to pay for a reduced variance compared to OLS. So, while the estimator of coefficients based on the ridge regression has a larger bias than OLS, it also has a much lower variance than OLS.

Why, technically, does ridge regression produce stable estimates? The likelihood part of the objective function (which is RSS) is exactly the same as for the OLS and would therefore be minimized by the OLS solution.

However, when there is little information to decide between a wide variety of combinations of $\left(\widehat{\beta}_1, \widehat{\beta}_2\right)$ that all produce almost the same likelihood value as long as their sum remains constant, then the OLS is unstable because even small changes in data can make the minimum of the RSS change widely across the large region of nearly constant likelihood value. Ridge regression instead prefers those points in the parameter space, where, in addition to low RSS, the magnitudes of all the parameters are fairly small. Thus, from among the points in the large region of almost constant likelihood value, ridge regression will choose one that is fairly close to the origin (0,0) to keep the ridge penalty low.

Why might ridge regression be useful in practice? Ridge regression gives us a family of regression models, parameterized by λ which determines the amount of shrinkage of the model coefficients. A possibility of using larger values of λ are beneficial especially in cases where p is large compared to n. There, unpenalized regression methods tend to overfit to the training data and by penalizing the model complexity we can reduce overfitting. A typical approach is to tune the value of λ to particular setting using cross-validation. Thus, in cases, where less shrinkage is enough to get good predictive accuracy, then λ can be kept smaller and ridge regression is close to the OLS. Thus, by using ridge regression, we do not necessarily shrink our coefficients compared to the OLS solution, if the cross-validation shows that λ values close to 0 do give the best performance.

Next we consider another version of penalized regression called LASSO that is particularly suitable for variable selection problems.

LASSO ($q = 1$)

The Least Absolute Shrinkage and Selection Operator (LASSO), introduced by Tibshirani 1996, is achieved by minimizing the objective function

$$O_{1,\lambda}(\boldsymbol{\beta} \,|\, \boldsymbol{y}) = -2\log(L(\boldsymbol{\beta} \,|\, \boldsymbol{y})) + \lambda \cdot \text{penalty}_1(\boldsymbol{\beta}) = -2\log(L(\boldsymbol{\beta} \,|\, \boldsymbol{y})) + \lambda \cdot \sum_{j=1}^{p} |\beta_j|.$$

Thus, the difference from ridge regression is the change from the ℓ^2 penalty β_j^2 to the ℓ^1 penalty $|\beta_j|$ per each coefficient.

Because the LASSO penalty includes the absolute value operator, the objective function is not differentiable and, as a result, lacks a closed form solution in general. However, in the special case of uncorrelated predictors ($\boldsymbol{X}^T\boldsymbol{X} = n\boldsymbol{I}$) it is possible to obtain the closed form solution for LASSO using the soft-thresholding operator $(t)_+$, which equals t, when $t > 0$, and 0 otherwise.

Suppose that the predictors in columns of $\boldsymbol{X}$ are uncorrelated. Then the LASSO solution is

$$\widehat{\beta}_j^{(\text{LASSO})} = \text{sign}\left(\widehat{\beta}_j^{(\text{OLS})}\right)\left(\left|\widehat{\beta}_j^{(\text{OLS})}\right| - \lambda\right)_+ = \begin{cases} \left(\widehat{\beta}_j^{(\text{OLS})} - \lambda\right), & \text{if } \widehat{\beta}_j^{(\text{OLS})} > \lambda \\ 0, & \text{if } \left|\widehat{\beta}_j^{(\text{OLS})}\right| \leq \lambda \\ \left(\widehat{\beta}_j^{(\text{OLS})} + \lambda\right), & \text{if } \widehat{\beta}_j^{(\text{OLS})} < -\lambda \end{cases},$$

where OLS refers to the ordinary least squares estimate from the linear model. The derivation of this result is available here. A general derivation of an efficient coordinate descent algorithm to optimize the LASSO objective function is here.

Let's check this using the `glmnet` package. You can install it from CRAN and detailed instructions of installation and other aspects are at https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html.

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-8
```

```
n = 100
X = as.matrix(scale(cbind(rnorm(n), rnorm(n)))) # two uncorrelated columns
y = as.numeric(scale(X %*% c(0.5, -0.2) + rnorm(n)))
OLS.coef = coefficients(lm(y ~ X))
OLS.coef # Ordinary least squares estimate
```

```
##   (Intercept)            X1            X2
## -2.437083e-17  5.195129e-01 -1.500554e-01
```

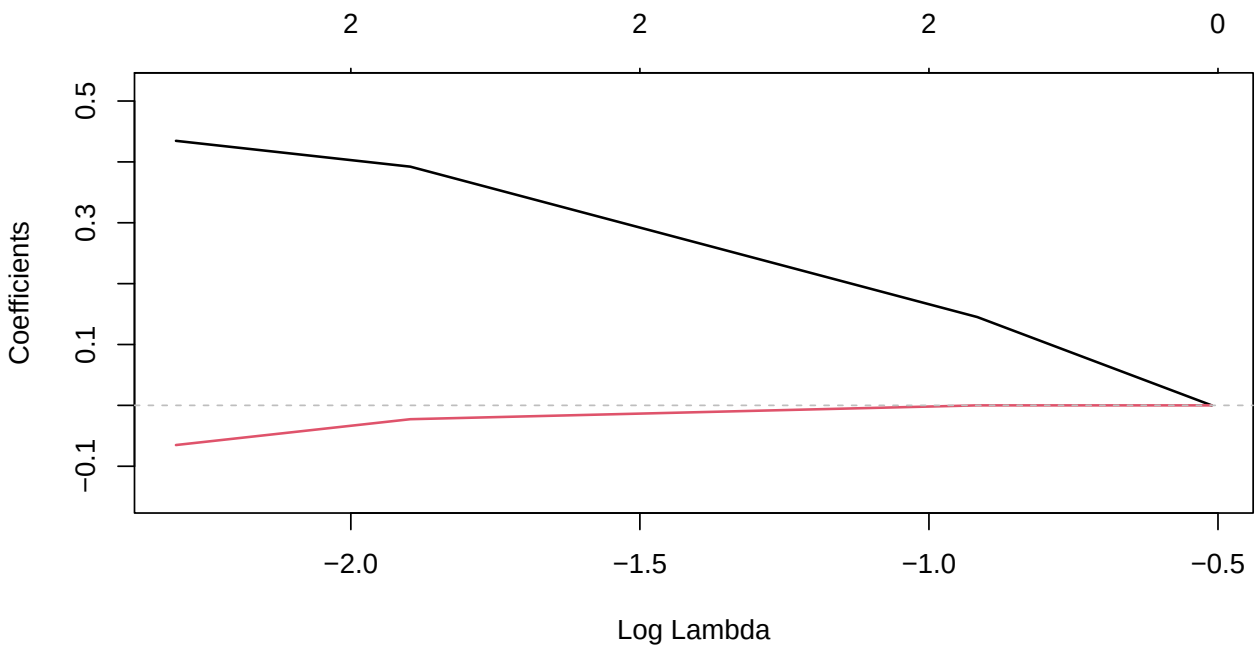
```
lambda = c(0.6, 0.4, 0.15, 0.1, 0.0) # Let's choose these to cover the range of the OLS values.

g.fit = glmnet(X, y, lambda = lambda, alpha = 1) # alpha = 1 means LASSO, we come back to this later.
data.frame(lambda = g.fit$lambda, t(as.matrix(coef(g.fit))))
```

```
##    lambda X.Intercept.        V1          V2
## s0   0.60 6.938894e-18 0.0000000  0.00000000
## s1   0.40 6.938894e-18 0.1450647  0.00000000
## s2   0.15 6.781402e-18 0.3921546 -0.02269691
## s3   0.10 6.486827e-18 0.4346076 -0.06514971
## s4   0.00 5.897675e-18 0.5195130 -0.15005541
```

`glmnet` uses symbol `s` for λ, so e.g. `s0` refers to the largest λ value of 0.6 and `s4` to the smallest value of 0.0. They are always ordered in descending order no matter how they were ordered in the original lambda vector. We see that the number of non-zero coefficients increases from 0 ($\lambda = 0.6$) to 2 (starting from $\lambda = 0.15$). We also see that whenever there is a non-zero coefficient, its distance to the OLS of the same parameter is approximately λ. For example, the OLS estimate for β_2 is -0.15 and therefore it does not survive penalization in models with $\lambda = 0.6$ or 0.4 as these are clearly > 0.15, but it starts to appear in the model when λ drops to around 0.15. Just like with ridge regression, we get back to the OLS when $\lambda = 0$.

We can also plot `glmnet` results.

```
plot(g.fit, xvar = "lambda", lwd = 1.5)
abline(h = 0, col = "gray", lty = 2)
```



Each curve corresponds to a predictor. It shows the path of its coefficient (y-axis) against log(λ) at the grid points used in `glmnet` above (from log(0.1) ~ -2.3 to log(0.6) ~ -0.51). The axis above the plot indicates the number of nonzero coefficients at each λ, which is the effective degrees of freedom (df) for LASSO. There are also other options for the x-axis (see `?plot.glmnet`).

```
coef(g.fit, s = 0.1) # get coefficients at s = 0.1.
```

```
## 3 x 1 sparse Matrix of class "dgCMatrix"
##                        s1
## (Intercept)  6.486827e-18
## V1           4.346076e-01
## V2          -6.514971e-02
```

```
coef(g.fit, s = 0.2) # interpolation at s = 0.2 (since 0.2 was not computed above)
```

```
## 3 x 1 sparse Matrix of class "dgCMatrix"
##                        s1
## (Intercept)  6.812901e-18
## V1           3.427367e-01
## V2          -1.815753e-02
```

```
# exact coeffs at s = 0.2 from original data. Dot means 0 in sparse matrix format.
coef(g.fit, s = 0.2, exact = T, x = X, y = y)
```

```
## 3 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept) 6.938894e-18
## V1          3.460723e-01
## V2          .
```

Let's see how LASSO performs on our earlier data set from HDS5 where we had $p = 70$ predictors of which only the first four had a non-zero effect (and together explained about 20% of variance) but since

the training data sample size $n = 150$ was small, pure OLS led to bad overfitting where the full model explained 50% of variance in the training data but essentially 0% in the test data. Since these predictors are generated independently, we expect that LASSO corresponds to soft-thresholding in these data. The key question is what should λ be? To determine this, we can use `cv.glmnet()` that does an automatic (10-fold) cross-validation to find the optimal λ.

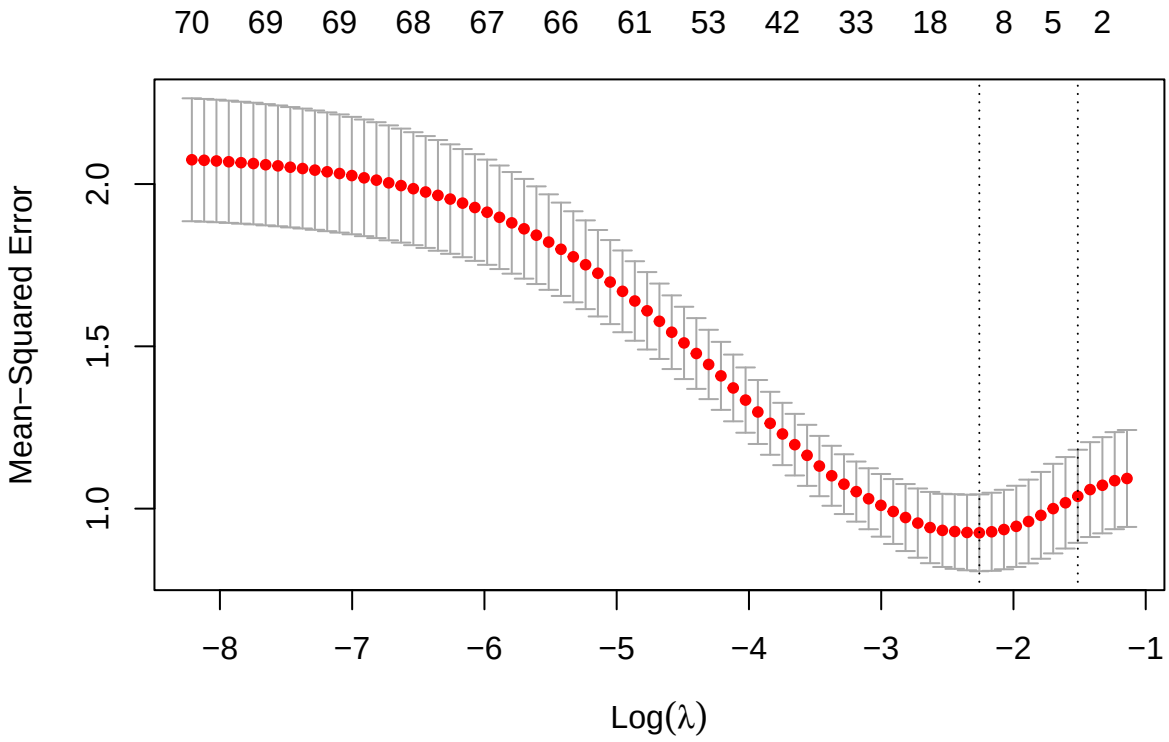
```
set.seed(20102017) # We repeat exactly the dataset from HDS5
p = 70
n.tr = 150 # training set
n.te = 150 # test set
n = n.tr + n.te
i.tr = 1:n.tr; i.te = (n.tr + 1):n # indexes for training and testing
phi = 0.05 # variance explained by x_1, should be 0 < phi < 1.
b = rep(c(sqrt(phi / (1-phi)), 0), c(4, p-4)) # effects 1,2,3,4 are non-zero, see Lect. 0.1 for "phi"
X = matrix(rnorm(n*p), nrow = n) # columns 1,...,4 of X have effects, other 66 cols are noise
eps = scale(rnorm(n, 0, 1)) # epsilon, error term
y = scale(X%*%b + eps) # makes y have mean = 1, var = 1
y.tr = y[i.tr]; y.te = y[i.te]
X.tr = data.frame(scale(X[i.tr,])); X.te = data.frame(scale(X[i.te,]))
lm.fit =  lm(y.tr ~ 0 + . , data = X.tr) # fit the model
lasso.cv = cv.glmnet(x = as.matrix(X.tr), y = y.tr, alpha = 1, type.measure = "mse")
lasso.cv$lambda.min # this is lambda corresponding to minimum MSE seen during CV
```

```
## [1] 0.1045647
```

```
lasso.cv$lambda.1se # this is the largest lambda with CV'd MSE < 1sd of min CV'd MSE
```

```
## [1] 0.2200986
```

```
plot(lasso.cv) # shows CV'd MSEs across lambdas, lines for lambda.min and lambda.1se
```



CV plot shows how CV'd MSE is about 2.0 for $\lambda \approx 0$ and drops around 0.9 at the minimum. This agrees with the results in HDS5 where we saw that CV with `cv.glm()` gave an MSE estimate of about 2.0 and this was considerably larger than the variance within the training data (1.08).

A reason one might prefer `lambda.1se` over `lambda.min` is that the former produces more sparsity (larger penalty) while still being acceptably close to the minimum observed MSE. `predict.cv.glmnet()` uses `lambda.1se` by default.

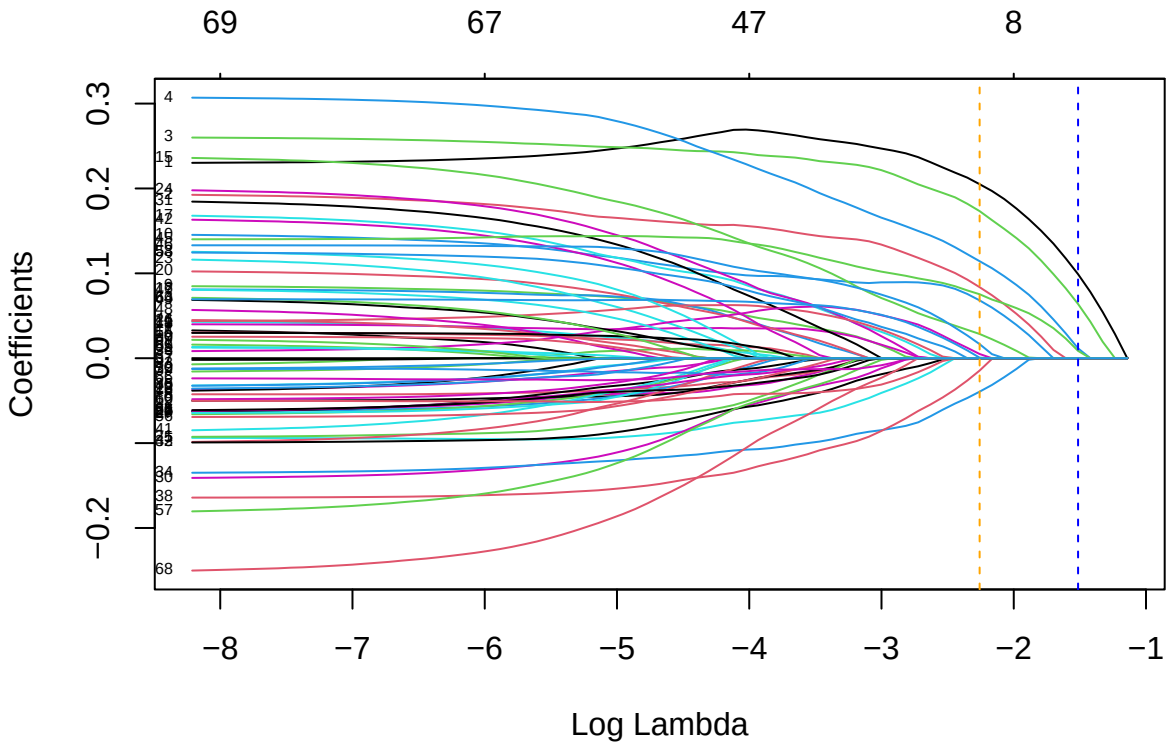
Cross-validation for linear model can be done using an error measure specified by `type.measure` parameter. The default value `mse` applies the mean-square error and the other option is `mae` (mean absolute error).

Questions.

1. Where in the CV MSE plot the method has high variance and where high bias?
2. What would it mean if your CV MSE plot were monotonic (either increasing or decreasing) from left to right and the reported minimum were at the end of the plot?

Let's plot the LASSO solution paths for each coefficient and let's plot the model for `lambda.1se`.

```
plot(lasso.cv$glmnet.fit, label = T, xvar = "lambda") #plot LASSO sequence
abline(v = log(lasso.cv$lambda.min), lty = 2, col ="orange")
abline(v = log(lasso.cv$lambda.1se), lty = 2, col = "blue")
```



The `glmnet` plot drawn from the `glmnet.fit` component of `cv.glmnet()` object shows how the 70 predictors evolve from the unpenalized OLS (left side) to the LASSO solution (dotted lines on right side). We see that only 4 or 12 variables are non-zero in LASSO solution depending whether we look at the `lambda.1se` (blue) or `lambda.min`(orange) model.

```r
#how many non-zero coefficients in each model
data.frame(model = c("min","1se"),
           lambda = c(lasso.cv$lambda.min,lasso.cv$lambda.1se),
           nzero =  c(lasso.cv$nzero[c(which(lasso.cv$lambda == lasso.cv$lambda.min),
                                       which(lasso.cv$lambda == lasso.cv$lambda.1se))]))
```

```
##     model    lambda nzero
## s12   min 0.1045647    12
## s4    1se 0.2200986     4
```

To see the coefficients, we can call `coef(glmnet.fit, s = lambda)`. Let's do that for the model `lambda.1se`.

```r
coef(lasso.cv$glmnet.fit, s = lasso.cv$lambda.1se)
```

```
## 71 x 1 sparse Matrix of class "dgCMatrix"
##                        s1
## (Intercept) -0.048789265
## X1           0.099011807
## X2           .
## X3           0.063869199
## X4           0.011605324
## X5           .
## X6           .
## X7           .
## X8           .
## X9           .
## X10          .
## X11          .
## X12          .
## X13          .
## X14          .
## X15          .
## X16          .
## X17          .
## X18          .
## X19          .
## X20          .
## X21          .
## X22          .
## X23          .
## X24          .
## X25          .
## X26          .
## X27          .
## X28          .
## X29          .
## X30          .
## X31          .
## X32          .
## X33          .
## X34          .
```

```
## X35          .
## X36          .
## X37          .
## X38          .
## X39          .
## X40          .
## X41          .
## X42          .
## X43          .
## X44          .
## X45          0.008645144
## X46          .
## X47          .
## X48          .
## X49          .
## X50          .
## X51          .
## X52          .
## X53          .
## X54          .
## X55          .
## X56          .
## X57          .
## X58          .
## X59          .
## X60          .
## X61          .
## X62          .
## X63          .
## X64          .
## X65          .
## X66          .
## X67          .
## X68          .
## X69          .
## X70          .
```

Since the first coefficient corresponds to the intercept, we will remove the first coefficient and then our indexes correspond to the predictors 1...70.

```
b.1se = as.numeric(coef(lasso.cv$glmnet.fit, s = lasso.cv$lambda.1se))[-1]
data.frame(predictor = which(b.1se != 0.0), coeff = b.1se[which(b.1se != 0.0)])
```

```
##   predictor       coeff
## 1         1 0.099011807
## 2         3 0.063869199
## 3         4 0.011605324
## 4        45 0.008645144
```

Thus `lambda.1se` kept 3 out of the 4 correct variables (1,2,3,4) and additionally included variable 45 with a small coefficient.

Let's see how well LASSO models trained on the training data work on the test data. Let's compute MSE in the test data and compare it to the values from HDS5.

```
pred.lasso.1se = predict(lasso.cv, newx = as.matrix(X.te)) #default is at lambda.1se
pred.lasso.min = predict(lasso.cv, newx = as.matrix(X.te), s = "lambda.min")
mse <- function(x,y){mean((x-y)^2)}
data.frame(MSE = c(mse(y.te, pred.lasso.1se),
                   mse(y.te, pred.lasso.min)),
           row.names = c("1se","min"))
```

```
##           MSE
## 1se 0.8885584
## min 0.8621403
```

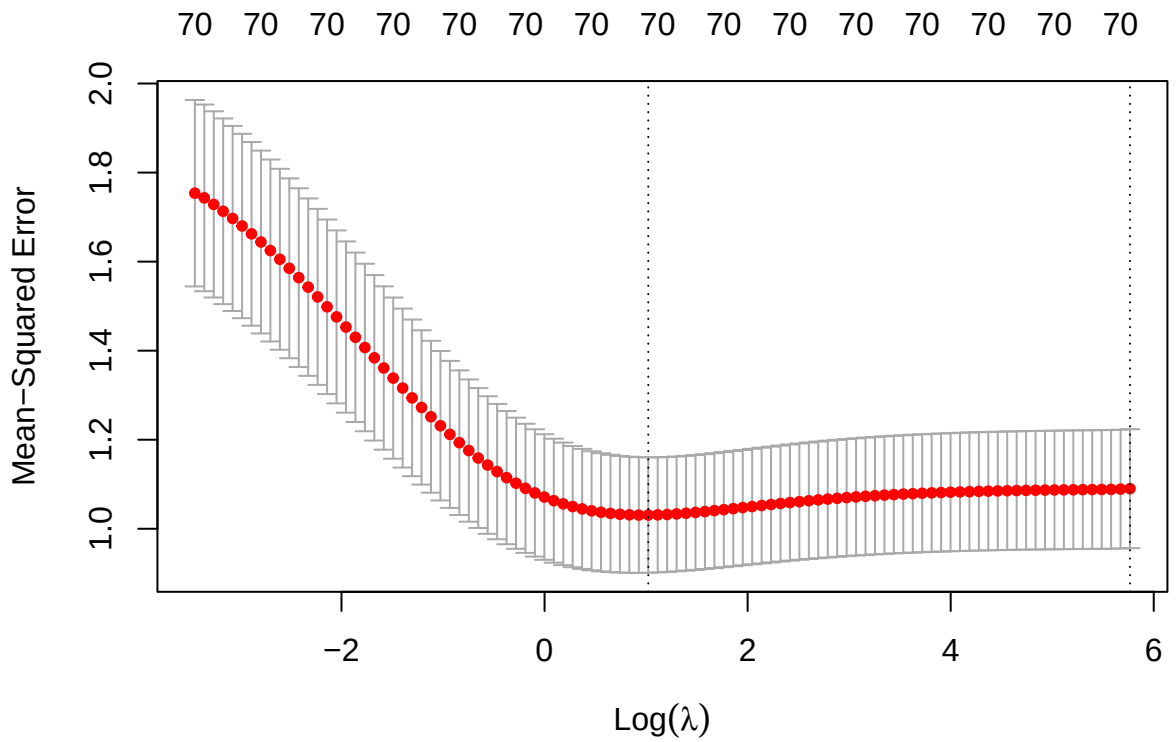
Remember that in HDS5 the variance in the test data was 0.92 and MSEs in the test data were

OLS	AIC back	AIC fwd	BIC back	BIC fwd
1.60	1.32	1.00	0.82	0.86

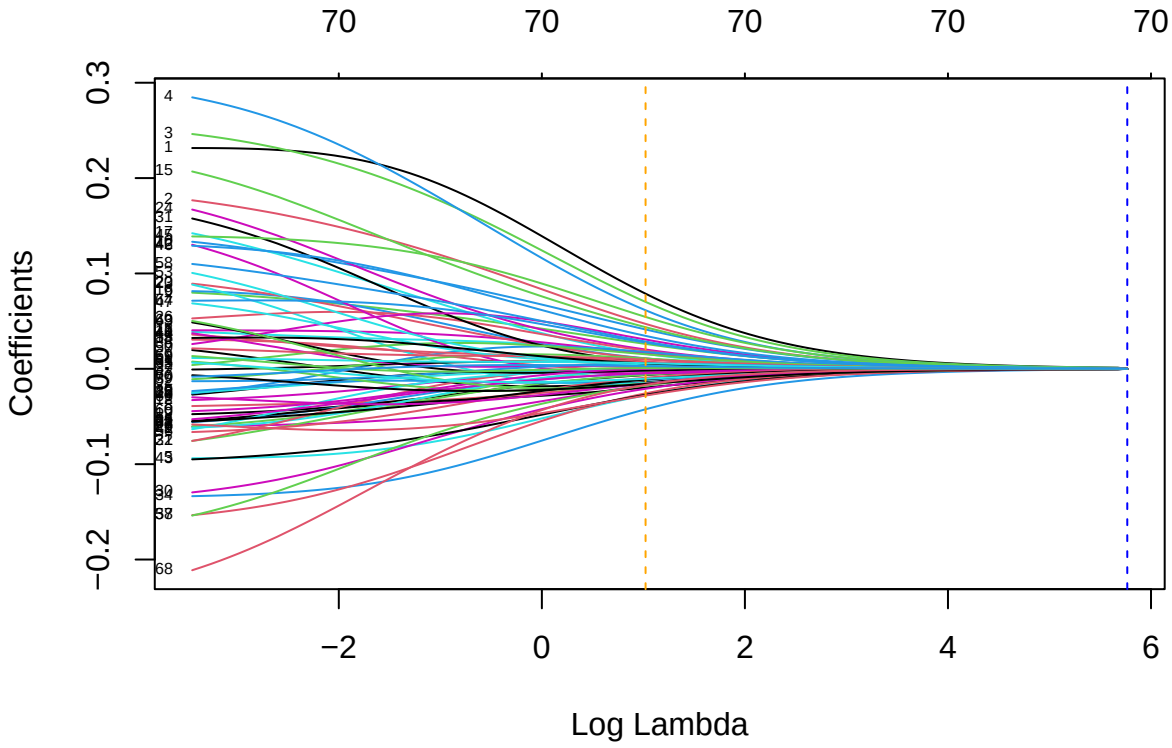
It seems that with LASSO we do quite similarly to BIC models.

What about ridge regression on the same problem? It can also be done with `glmnet` by setting `alpha=0`.

```
ridge.cv = cv.glmnet(x = as.matrix(X.tr), y = y.tr, alpha = 0)
plot(ridge.cv) #shows CV'd MSEs across lambdas, lines for lambda.min and lambda.1se
```




```
plot(ridge.cv$glmnet.fit, label = T, xvar = "lambda")
abline(v = log(ridge.cv$lambda.min), lty = 2, col ="orange")
abline(v = log(ridge.cv$lambda.1se), lty = 2, col ="blue")
```




```
pred.ridge.1se = predict(ridge.cv, newx = as.matrix(X.te)) #default is at lambda.1se
pred.ridge.min = predict(ridge.cv, newx = as.matrix(X.te), s = "lambda.min")
data.frame (MSE = c( mse(y.te, pred.ridge.1se),
                     mse(y.te, pred.ridge.min)),
            row.names = c("1se","min"))
```

```
##           MSE
## 1se 0.9222097
## min 0.9187955
```

We see that ridge regression model does not work on these data well as its prediction accuracy measured by MSE is no smaller than the variance of test data, 0.92. We also see from the coefficient plot that the `lambda.1se` model essentially shrinks all coefficients to very small values so would correspond to a pure intercept model that predicts everything by the mean of the training data which explains the numerical similarity of MSE to the variance of the outcome variable. Remember that predicted values from pure OLS had much higher MSE than the variance of the test data, whereas ridge regression coefficients at `lambda.1se` are shrunk in such a way that no such overfitting happens.

This particular example is a case where sparse models, such as LASSO, are preferred over ridge regression. Typically, however, when there are many non-zero coefficients, then ridge regression often provides a better prediction accuracy than LASSO.

Ridge regression vs. LASSO

Constrained optimization interpretation It can be shown by using a constrained optimization technique called Lagrange multipliers, that ridge regression and LASSO can be formulated as optimization problems within a certain region around the origin $\mathbf{0}$, whose size is controlled by a parameter t that is a function of parameter λ and the likelihood function.

- Ridge regression (with $t = t(\lambda, \boldsymbol{y})$):

$$\text{Find minimum of } -\log(L(\boldsymbol{\beta}|\boldsymbol{y})) \text{ when } \sum_{j=1}^{p} \beta_j^2 \leq t.$$

- LASSO (with $t = t(\lambda, \boldsymbol{y})$):

$$\text{Find minimum of } -\log(L(\boldsymbol{\beta}|\boldsymbol{y})) \text{ when } \sum_{j=1}^{p} |\beta_j| \leq t.$$

In both cases, for large enough $t \geq t_0$, the maximum likelihood estimate (MLE) will belong to the search region and therefore the MLE will be the solution of the problem. This threshold value t_0 corresponds to $\lambda = 0$ and values $t < t_0$ are in one-to-one correspondence with values $\lambda > 0$. Obviously, for ridge regression $t_0 = \sum_{j=1}^{p} \widehat{\beta}_j^2$ and for LASSO $t_0 = \sum_{j=1}^{p} \left|\widehat{\beta}_j\right|$, where $\widehat{\beta}_j$ is the MLE of β_j.

Bayesian interpretation Let's consider the linear model with Gaussian errors with known variance σ^2.

$$y_i \sim \mathcal{N}(\beta_0 + \boldsymbol{x}_i^T \boldsymbol{\beta}, \sigma^2), \text{ for } i = 1, \ldots, n.$$

Let's assume that the parameters β_j have a Gaussian prior with known variance τ^2,

$$\beta_j \sim \mathcal{N}(0, \tau^2), \text{ for } j = 1, \ldots, p.$$

By Bayes rule, the posterior distribution will be proportional to the product of prior and likelihood. Consequently, the log-posterior is, up to an additive constant, the sum of logarithms of prior and likelihood. Thus, up to a multiplicative constant C that does not depend on $\boldsymbol{\beta}$,

$$\log(\Pr(\boldsymbol{\beta}|\boldsymbol{y})) = C - \frac{1}{2\tau^2}\boldsymbol{\beta}^T\boldsymbol{\beta} - \frac{1}{2\sigma^2}\left(\boldsymbol{y} - \beta_0 - \boldsymbol{X}\boldsymbol{\beta}\right)^T\left(\boldsymbol{y} - \beta_0 - \boldsymbol{X}\boldsymbol{\beta}\right)$$

The maximum value of the posterior distribution (maximum a posteriori estimate, MAP estimate) is the same value as the solution to the ridge regression problem with $\lambda = \sigma^2/\tau^2$. Thus, ridge regression is equivalent to the MAP estimate from a Bayesian linear model with a zero-centered Gaussian prior on the parameters.

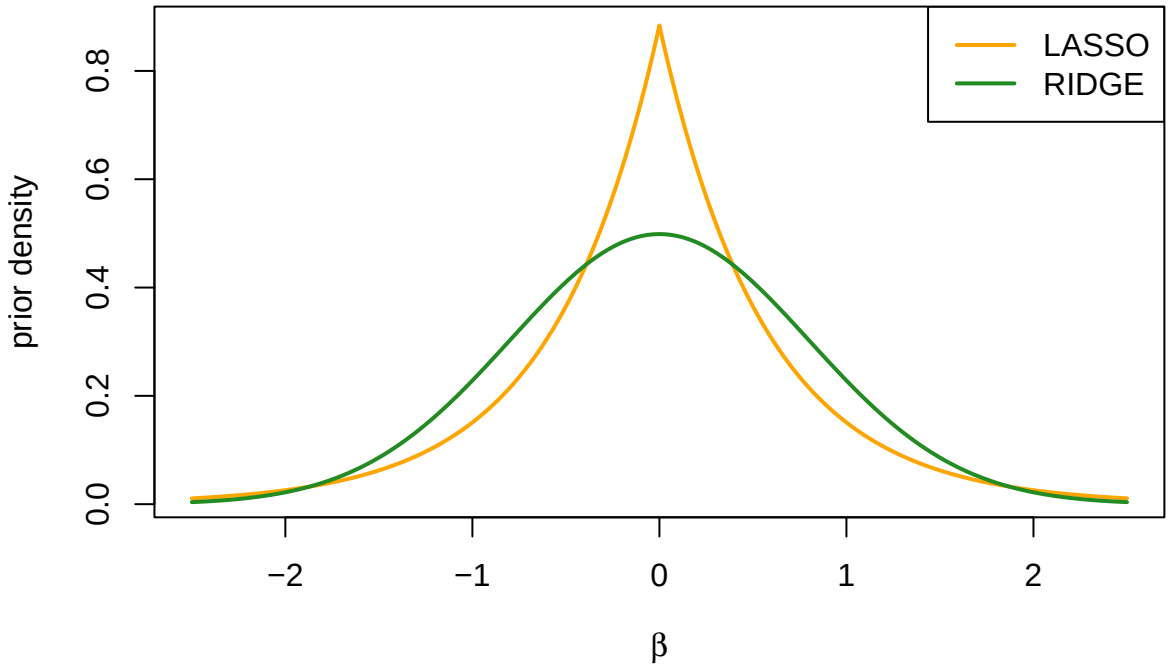
This gives a new interpretation for the λ values. If we assume a priori that the linear model coefficients $\boldsymbol{\beta}$ are quite close to zero, then we make τ^2 small and consequently $\lambda \propto \tau^{-2}$ large. If instead we assume a flat prior ($\tau^2 \to \infty$) then $\lambda \to 0$ and we are considering the pure maximum likelihood estimate, which coincides with the OLS solution for this linear model. This relationship shows how a prior assumption about the parameters of the statistical model leads to ridge regression, whereas our original definition of ridge regression through the use of a "penalized likelihood" was an optimization formulation of an ad-hoc objective function. In Bayesian statistics, the penalized likelihood is simply the posterior distribution of the parameters corresponding to a particular prior distribution.

We can do a similar interpretation for the LASSO model. There the prior of each coefficient is the Laplace distribution (a.k.a double exponential distribution) with scale $\tau > 0$. So $\beta_j \sim \text{Laplace}(0, \tau)$ which means that the prior density is

$$\Pr(\beta_j) = \frac{1}{2\tau}\exp(-|\beta_j|/\tau).$$

By combining this with the likelihood from the linear model with Gaussian errors, we see that the MAP estimate is the LASSO estimate with $\lambda = 2\sigma^2/\tau$. Let's plot the priors corresponding to ridge regression and LASSO regression so that they have the same variance.

```r
par(mfrow=c(1,1))
x = seq(-2.5, 2.5, 0.001) #plot priors on these grid points
eff.var = 0.8^2 #both distributions will have this same variance
y.ridge = dnorm(x, 0, sqrt(eff.var)) #prior for ridge
y.lasso = 1/2/sqrt(eff.var/2) * exp(-abs(x) / sqrt(eff.var/2) )#prior for LASSO
plot(x, y.lasso, t = "l", lty = 1, lwd = 2,
     col = "orange", xlab = expression(beta), ylab = "prior density")
lines(x, y.ridge, lty = 1, lwd = 2, col = "forestgreen")
legend("topright", leg = c("LASSO","RIDGE"), col = c("orange","forestgreen"), lwd = 2)
```



For the original formulation of the penalized regression problem, λ was simply a nuisance parameter that needed to be chosen to estimate the β-coefficients that are our main interest. However, with the Bayesian interpetation we know how the value of λ is related to the parameters controlling the coefficients' prior distribution ($\beta_j \sim \mathcal{N}(0, \sigma^2/\lambda)$ for ridge and $\beta_j \sim$ Laplace$(0, 2\sigma^2/\lambda)$ for LASSO). Therefore, by learning a value of λ from the data (e.g. by cross-validation) we learn about the coefficient distribution that best describes the data. Hence λ is not only a nuisance parameter but also tells about the magnitude of the effects that we see in the data.

While ridge regression and LASSO are connected to Bayesian models via being MAP estimates, we should keep in mind that the Bayesian regression models typically aim to estimate the whole posterior distribution of the coefficients, not just a point estimate such as the MAP. This is computationally a much more challenging task, and ridge and LASSO are not necessarily the best models for such a complete Bayesian analysis, since they lack flexibility to tune the shrinkage factor of each individual coefficient and instead they only consider a single global λ parameter. You can find some more discussion about more advanced Bayesian approaches here. by M. Betancourt.

Correlated predictors Earlier we figured out how each method works in the simplest case of uncorrelated predictors:

- Ridge regression shrinks the OLS towards zero by a multiplicative factor that gets smaller (increasing shrinkage) as λ grows

- LASSO does soft-thresholding where each coefficient is pulled towards zero by a constant amount determined by λ until the coefficient drops to exactly zero

But how do they work with correlated predictors?

Ridge regression is known to shrink the coefficients of correlated predictors towards each other. In the extreme case of k identical predictors, they each get identical coeffcients with $1/k$th the size that any single one would get if fit alone. From a Bayesian point of view, the ridge penalty is ideal if there are many predictors, and all have non-zero coeffcients (drawn from a Gaussian distribution). LASSO, on the other hand, is somewhat indifferent to very correlated predictors, and will tend to pick one and ignore the rest. In the extreme case above, the LASSO problem breaks down because several combinations of parameter values give same value for the objective function. To overcome this problem, a combination of ridge and LASSO penalties have been introduced (so called *elastic net* penalty).

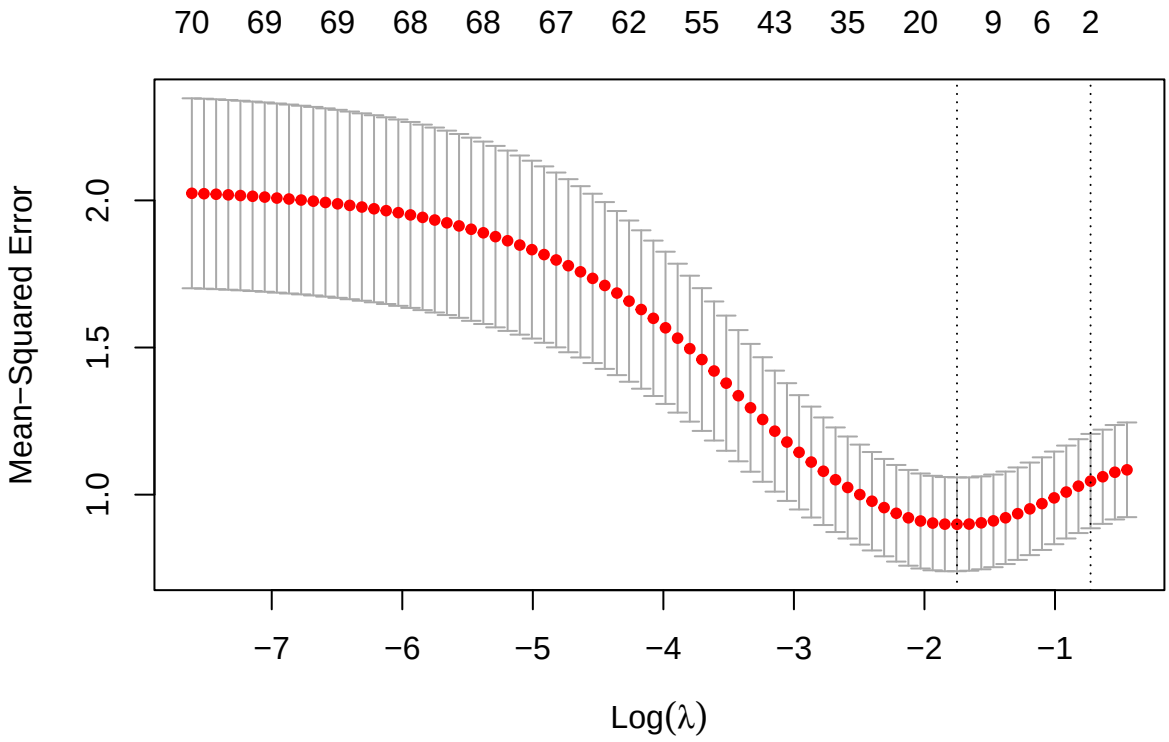
Other penalties? What about other forms of penalties than $q = 1$ (LASSO) and $q = 2$ (ridge)? Values of $q \in (1, 2)$ suggest a compromise between LASSO and ridge regression. Although this is the case, with $q > 1$, $|\beta_j|^q$ is differentiable at 0, and so does not share the ability of LASSO for setting coefficients exactly to zero. Partly for this reason, as well as for computational tractability, Zou and Hastie (2005) introduced the *elastic net* penalties as linear combinations of the LASSO and ridge regression penalties

$$\text{penalty}^{\alpha}_{\text{enet}}(\boldsymbol{\beta}) = \sum_{j=1}^{p}((1-\alpha)\beta_j^2 + \alpha|\beta_j|).$$

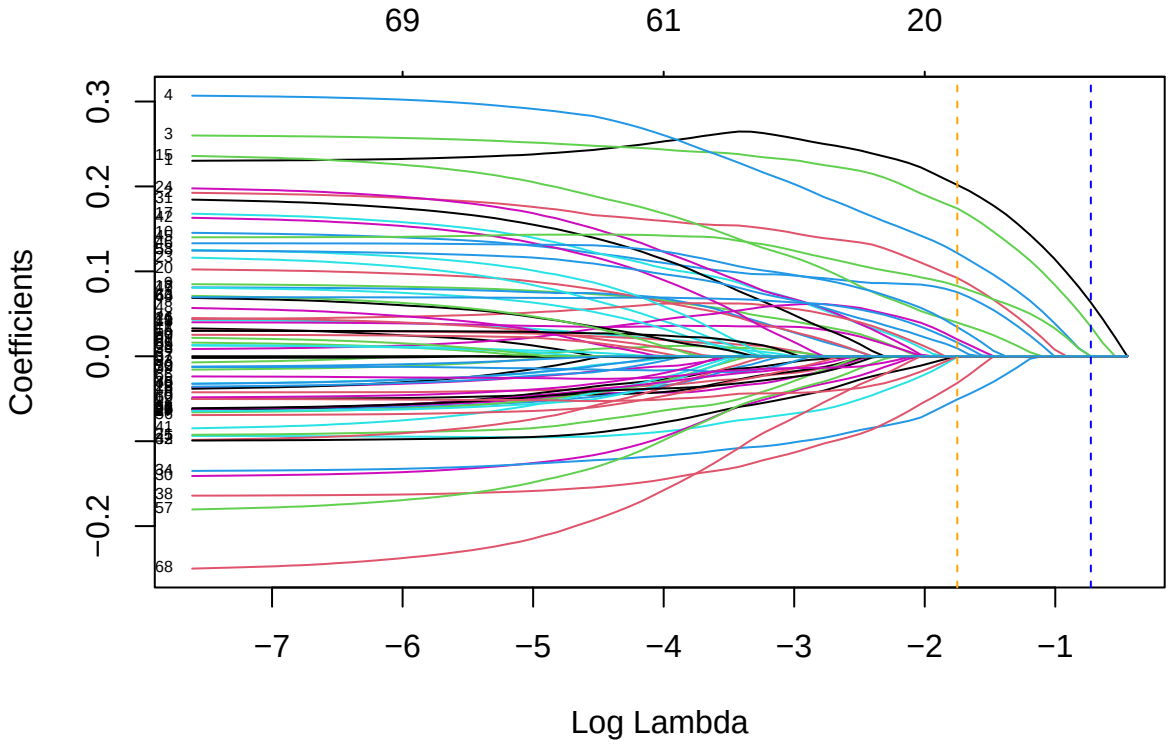
This also explains why in `glmnet()` $\alpha = 1$ corresponds to LASSO and $\alpha = 0$ to ridge regression.

Let's test elastic net with $\alpha = 0.5$ on the data we have been considering where $n = 150$ and $p = 70$ and only the first four predictors have truly an effect.

```
enet.cv = cv.glmnet(x = as.matrix(X.tr), y = y.tr, alpha = 0.5)
plot(enet.cv) # shows CV'd MSEs across lambdas, lines for lambda.min and lambda.1se
```



```
plot(enet.cv$glmnet.fit, label = T, xvar = "lambda") # plot coefficients
abline(v = log(enet.cv$lambda.min), lty = 2, col = "orange")
abline(v = log(enet.cv$lambda.1se), lty = 2, col = "blue")
```




```
pred.enet.1se = predict(enet.cv, newx = as.matrix(X.te)) # default is at lambda.1se
pred.enet.min = predict(enet.cv, newx = as.matrix(X.te), s = "lambda.min")
data.frame(MSE = c(mse(y.te, pred.enet.1se),
                   mse(y.te, pred.enet.min)),
           row.names = c("1se", "min"))
```

```
##           MSE
## 1se 0.9031464
## min 0.8724816
```

With this particular elastic net model ($\alpha = 0.5$), the results are slightly worse than with LASSO but better than with ridge regression in this data set that truly benefits from a sparse model.

It would be possible to use cross-validation also to estimate α to given an optimal elastic net model.

Logistic regression We can apply `glmnet` to fit also logistic, multinomial, Poisson or Cox regression models. The syntax is similar to the linear regression case and the distribution is specified by `family =` parameter whose options are "gaussian","binomial","poisson","multinomial","cox" or "mgaussian".

Let's study logistic regression here and leave the other models to be introduced by the glmnet manual.

In logistic regression, we model the binary outcome $y_i \in \{0, 1\}$ for $i = 1, \ldots, n$, using the p predictors in columns of matrix $\boldsymbol{X}$ whose row i is denoted as $\boldsymbol{x}_i$. The model assumes that the log-odds of the event $y_i = 1$ depends linearly on the predictors

$$\log\left(\frac{\Pr(y_i = 1)}{1 - \Pr(y_i = 1)}\right) = \beta_0 + \boldsymbol{x}_i^T \boldsymbol{\beta}$$

or equivalently that the probability of event $y_i = 1$ is

$$\Pr(y_i = 1) = \frac{\exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})}{1 + \exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})} = \frac{1}{1 + \exp(-\beta_0 - \boldsymbol{x}_i^T\boldsymbol{\beta})}.$$

The likelihood function comes from the Bernoulli distribution

$$L(\boldsymbol{\beta}\,|\,\boldsymbol{y}) = \prod_i \Pr(y_i = 1)^{y_i} \Pr(y_i = 0)^{(1-y_i)} = \prod_i \frac{\exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})^{y_i}}{1 + \exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta})}$$

and the log-likelihood is

$$\log(L(\boldsymbol{\beta}\,|\,\boldsymbol{y})) = \sum_i \left[y_i(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta}) - \log(1 + \exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta}))\right].$$

Thus, with glmnet, we are searching for $\boldsymbol{\beta}$ that minimizes the function

$$-2\sum_i \left[y_i(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta}) - \log(1 + \exp(\beta_0 + \boldsymbol{x}_i^T\boldsymbol{\beta}))\right] + \lambda \cdot \text{penalty}_{\text{enet}}^{\alpha}(\boldsymbol{\beta}).$$

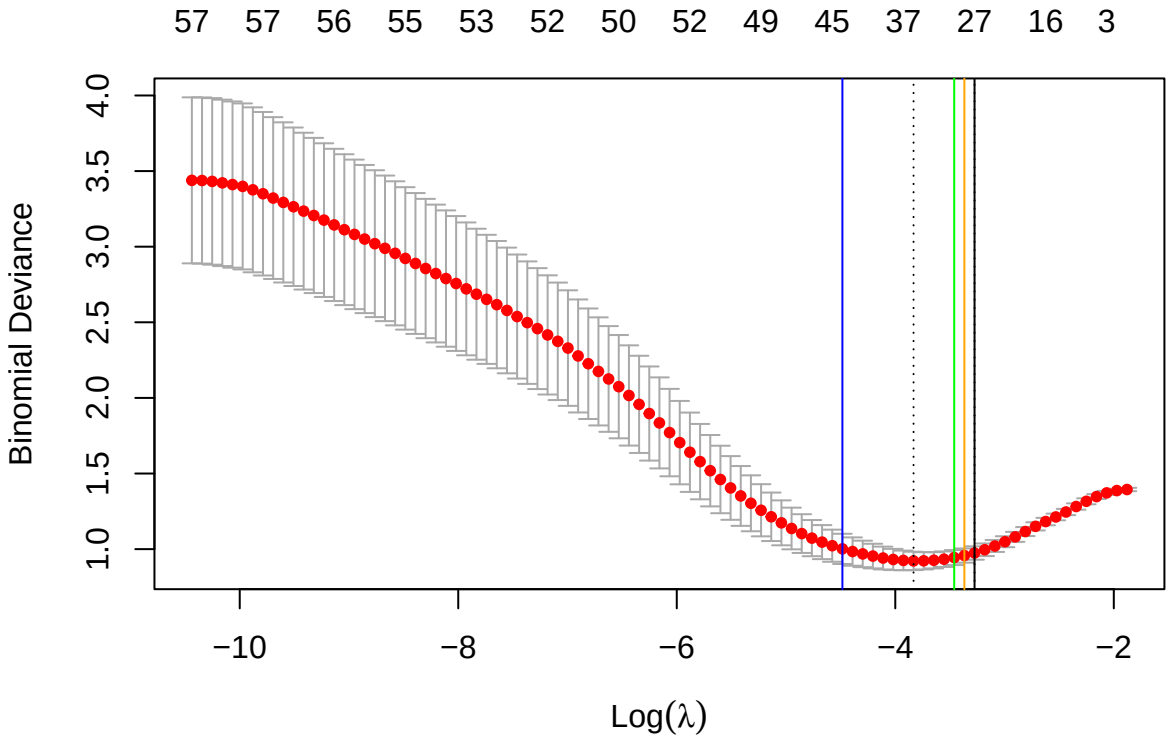
Let's use the existing data ($p = 70$, $n = 150$ training and test samples) to illustrate logistic regression.

```
logist.coeff = c(rnorm(10, 1, 0.2), rnorm(10, -1, 0.2), rep(0, p-20) ) # coeff 1,...,20 are nonzero
prob.tr = 1/(1 + exp(-as.matrix(X.tr) %*% logist.coeff)) # prob of 1 in log. regression
bin.tr = rbinom(n.tr, size = 1, prob = prob.tr) # generate a binary outcome
prob.te = 1/(1 + exp(-as.matrix(X.te) %*% logist.coeff))
bin.te = rbinom(n.te, size = 1, prob = prob.te)
```

The default CV error measure for logistic regression is "deviance", which is defined as $2(\log(L(\widehat{\boldsymbol{\beta}}_{\text{saturated}}|\boldsymbol{y})) - \log(L(\widehat{\boldsymbol{\beta}}|\boldsymbol{y})))$, where the saturated model has so many parameters that it fits the data perfectly. Thus the deviance measures distance ("deviance") from the highest possible likelihood value, and a smaller deviance means a better fit. (Note that for logistic regression, the log-likelihood value of the saturated model is 0, corresponding to the case where each probability value in the likelihood function has taken the value of 1.)

Other error measures that could be used with the logistic model include MSE (`mse`) and mean absolute error (`mae`) that both are computed from the errors on the observed scale, i.e., outcome is 0 or 1 and the prediction is a probability value between 0 and 1. Additionally, the error can be measured by the misclassification rate (`class`) where probabilities are rounded into the best guesses (0 or 1) or by the area under curve of the ROC curve (`auc`). Let's compare λs from all these possibilities on a plot that shows the CV results of LASSO as measured by deviance.

```
types = c("mae", "class", "auc", "deviance")
lambda = c()
for(ii in 1:length(types)){
  cv.bin = cv.glmnet(x = as.matrix(X.tr), y = bin.tr, family = "binomial",
                     type.measure = types[ii], alpha = 1) # LASSO
  lambda[ii] = cv.bin$lambda.1se
}
plot(cv.bin) # uses deviance as the error measure
abline(v = log(lambda), col = c("blue","green","orange","black"))
```




```
data.frame(types, lambda)
```

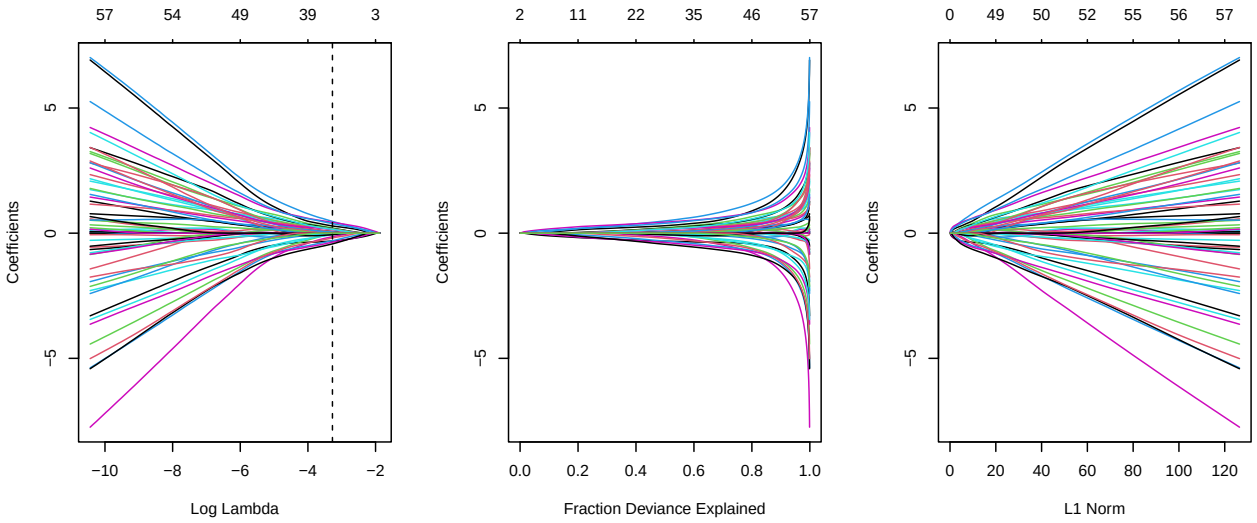
```
##      types     lambda
## 1      mae 0.01128669
## 2    class 0.03140587
## 3      auc 0.03446791
## 4 deviance 0.03782849
```

These seem close to each other and we are happy to use the deviance.

Question: Why does the deviance in the plot above not go down when $\lambda \to 0$ ($\log(\lambda) \to -\infty$), even though the model becomes less penalized and can fit the data better then?

The coefficient plots can be shown on three scales. So far, we have used option `lambda` which puts $\log(\lambda)$ on the x-axis. The other two options are the scaled deviance `dev` (1 means the fit is perfect and 0 means the intercept-only model), and `norm` showing the norm of the coefficient vector on the x-axis, where the norm is $\sum_j |\beta_j|$ for LASSO and $\sum_j \beta_j^2$ for ridge regression.

```
par(mfrow = c(1, 3))
plot(cv.bin$glmnet.fit, xvar = "lambda")
abline(v = log(cv.bin$lambda.1se), lty = 2) # put CV chosen level of penalty
plot(cv.bin$glmnet.fit, xvar = "dev")
plot(cv.bin$glmnet.fit, xvar = "norm")
```



Note how in the deviance plot (middle panel), at the right hand side end, the deviance is not anymore increasing much but the coefficients still get very quickly larger in magnitude. Such behavior suggests overfitting to the training data that happens when the penalization is becoming so weak that the model is free to find a perfect fit in the training data.

Prediction from a logistic regression model can be returned on the log-odds scale (default), on the probability scale `response`, or as the best guess outcome values `class`.

```
predict(cv.bin, type = "response", newx = as.matrix(X.te[1:2,])) # probability of being 1
```

```
##   lambda.1se
## 1  0.5666249
## 2  0.7439141
```

```
predict(cv.bin, type = "class", newx = as.matrix(X.te[1:2,])) # best guess (0 or 1)
```

```
##   lambda.1se
## 1        "1"
## 2        "1"
```

```
predict(cv.bin, newx = as.matrix(X.te[1:2,])) # log-odds, i.e., log( p/(1-p) )
```

```
##   lambda.1se
## 1   0.268094
## 2   1.066413
```

Standardization of $\boldsymbol{X}$ and $\boldsymbol{y}$ in `glmnet` In the beginning of these notes it was mentioned that the predictors in penalized regression models should be standardized. This is because, in penalized regression, the penalty terms are defined as sums over $|\beta_j|^q$, and therefore it is important that the scales of different predictors are chosen appropriately with respect to the other predictors. If we have no reason to do otherwise, then the default choice is to scale every predictor to have the same variance (=1), and then the same value for any β_j means the same proportion of variance of the outcome variable explained by the predictor. `glmnet` does the standardization of $\boldsymbol{X}$ automatically, whether or not you gave the predictors as standardized (unless you explicitly specify `standardize = FALSE`). Therefore, the default model fit from `glmnet` does not depend on whether you input $\boldsymbol{X}$ as standardized or not. However, the coefficients that `glmnet` outputs are always

on the scale on which you input the predictors. The crucial thing is that when you do predictions from a `glmnet` model, the test data predictors must be given on the same scale as the training data predictors were given in the original `glmnet` call.

Thus, in the exercises, you don't need to standardize $\boldsymbol{X}$ yourself (except if explicitly asked to), but you can just let `glmnet` do it automatically (as long as you don't change the default option: `standardize = TRUE`).

The outcome variable $\boldsymbol{y}$ does not need to be standardized. In practice, it will be mean-centered by the default intercept term in the model (unless you specify `intercept = FALSE`). If both $\boldsymbol{y}$ and all the columns of $\boldsymbol{X}$ are mean-centered prior to calling `glmnet`, then the intercept in linear model will be 0, but even in that case, it does not harm to have the intercept in the model. In non-linear models, you should always have the intercept in the model.

READ: "6.2 Shrinkage Methods" from ISL

Questions.

- How would you need to pre-process the predictors and outcome variable before applying penalized regression?
- What do ridge regression, LASSO, and subset selection do when predictors are uncorrelated?
- What are advantages of penalized regression methods over stepwise selection methods?
- If you want to choose a subset of important variables would you use ridge regression or LASSO?
- In which cases would you prefer ridge regression over LASSO?