Assignments: Inference for High Dimensional Data

Joydeep Chowdhury

Contents

Introduction

This document contains the assignment problems and the materials needed to solve them for the course *Inference for High Dimensional Data* in M.Stat 2nd year, 2019–2020, at Indian Statistical Institute, Kolkata.

Chapter 1

Regularized regression

Consider a linear regression problem, where Y is the $n \times 1$ vector of real valued responses, X is an $n \times p$ matrix of corresponding covariate values, β is the $p \times 1$ vector of coefficients, and we model $E[Y] = X\beta$. When $p > n$, or, in general when $X'X$ is singular or nearly singular (e.g., in the presence of multicollinearity), the usual least squares method of estimating β fails to work satisfactorily. Further, for large p, i.e., in the presence of a large number of covariates in the model, one may be interested in finding whether only a few covariates disproportionately affect the response. One may be also interested in imposing a degree of smoothness among the consecutive elements in the coefficient vector β. In such situations, we apply regularization in the form of a penalty term based on β.

Among many different regularization methods, we shall describe the following and demonstrate their computations in simulated data:

- Lasso
- Ridge regression
- Elastic net
- Adaptive lasso
- Group lasso
- Fused lasso

1.1 Lasso and other regularizations

In this section, different types of regularizations and their utilities are described.

1.1.1 Lasso

In the lasso setup, we minimize the loss function

$$L_{lasso}(\beta) = \frac{1}{2n}\|Y - X\beta\|_2^2 + \lambda\|\beta\|_1$$

with respect to β. Here, $\lambda \geq 0$ is a tuning parameter. For $\lambda = 0$, the lasso formulation reduces to the usual least squares setup. For large enough λ, the minimizer β is 0.

The utility of the lasso formulation in variable selection stems from the fact that $q = 1$ is the smallest value for which the l_q norm is convex. The variable selection property of l_q penalized methods improves with decreasing q. On the other hand, it is relatively far easier computationally to solve a convex optimization problem than a non-convex one. For example, the l_0 penalization is the purest form of variable selection method. However, the minimization of a l_0 penalized criterion is computationally very challenging.

The lasso (**l**east **a**bsolute **s**hrinkage and **s**election **o**perator) method was proposed by Tibshirani (1996).

During numerical computations, the tuning parameter λ is selected using cross validation.

Usually, before computing the lasso estimate, the response as well as the covariates are centered so that average of Y and the averages of the columns of X are all equal to 0. This justifies the omission of the intercept term in the linear regression model (i.e., omission of β_0 in the model $Y = \beta_0 1 + X\beta$). It is easy to recover the coefficeints for the uncentered case from those for the centerd case. In case the covariates are recorded in different units, additionally we may also standardize them so that $X'_{\cdot i}X_{\cdot i} = 1$ for all j, where $X_{\cdot i}$ is the ith column of X. This makes the model free of the effect of the units of the covariates.

1.1.2 Ridge regression

In the ridge regression setup, the loss function is

$$L_{ridge}(\beta) = \frac{1}{2n}\|Y - X\beta\|_2^2 + \frac{\lambda}{2}\|\beta\|_2^2.$$

Here again, $\lambda \geq 0$ is a tuning parameter.

Unlike the lasso, the ridge regression exhibits no variable selection property. It shrinks all the coefficients together rather than reducing some of them to 0. The ridge regression method is also known as the Tikhonov regularization, named after Andrey Tikhonov (Tikhonov, 1943). The ridge regression method works well in the presence of multicollinearity: when some covariate variables are highly correlated.

The tuning parameter λ is selected via cross validation while computing the estimate of β.

1.1.3 Elastic net

it is observed that the lasso method does not work well when some covariate variables are highly correlated. However, in this particular area, ridge regression excels. Based on this observation, the elastic net method is developed combining the penalization terms of the lasso and the ridge regression, and the loss function here is

$$L_{elnet}(\beta) = \frac{1}{2n}\|Y - X\beta\|_2^2 + \lambda\left[((1-\alpha)/2)\|\beta\|_2^2 + \alpha\|\beta\|_1\right].$$

Here, $0 \leq \alpha \leq 1$ balances the charactersitics of the lasso and the ridge regression in the elastic net formulation, and $\lambda \geq 0$ is a tuning parameter. For $\alpha = 0$, the ealstic net formulation coincides with the ridge regression method, and $\alpha = 1$ makes the eastic net method identical to the lasso.

The elastic net method was first proposed by Zou and Hastie (2005). What is behind the curious name *elastic net*? In the words of its authors,

> Similar to the lasso, the elastic net simultaneously does automatic variable selection and continuous shrinkage, and it can select groups of correlated variables. It is like a stretchable fishing net that retains 'all the big fish'. — Zou and Hastie (2005).[1]

For moderate values of α, say, $\alpha = 0.5$, the elastic net method tends to either select groups of highly correlated covariates together or discards them together.

A problem that the lasso method suffers is that when $p > n$, it can select at most n non-zero coefficients. This limitation may be undesirable in some high dimensional setup. The elastic net for $\alpha < 1$ does not suffer from this issue.

While computing the estimate of β from the sample using the elastic net method, one may choose both the parameters α and λ through cross validation. Or, one may fix the elastic net parameter α at some appropriate value, and choose λ through cross validation.

1.1.4 Adaptive lasso

Sometimes, the lasso method is not found to be efficient enough for variable selection, and it ends up selecting too many variables. The adaptive lasso method, proposed by Zou (2006), is useful in such situations. The adaptive lasso is a two-step method, which is started with an initial estimate $\tilde{\beta}$. Then, the loss function becomes

$$L_{adaptive}(\beta) = \frac{1}{2n}\|Y - X\beta\|_2^2 + \lambda\sum_{i=1}^{p} w_i|\beta_i|,$$

[1] There is also a story behind the name *lasso*, involving an execution device and a gentle Canadian. Look up Tibshirani (2011) if you are interested in the history of the lasso!

where $w_i = \left|\tilde{\beta}_i\right|^{-\gamma}$ for some $\gamma > 0$, $i = 1, \ldots, p$, and $\lambda \geq 0$ is a tuning parameter. Note that given $\tilde{\beta}$, $L_{adaptive}(\beta)$ is convex in β. In fact, using a scale transformation of β and the covariates, $L_{adaptive}(\beta)$ can be expressed in the form of the lasso loss function $L_{lasso}(\beta)$, and hence can be solved using the algorithms developed for the lasso method. The parameter γ is often taken to be 1 for the sake of simplicity (Bühlmann and Van De Geer, 2011).

Taking $r/0 = \infty$ for $r > 0$, it follows that whenever $\lambda > 0$, for any $\tilde{\beta}_i = 0$, we must have $\hat{\beta}_i = 0$, where $\hat{\beta}$ is the adaptive lasso estimate. Therefore, the solution of the adaptive lasso problem is sparser than the initial estimate. Further, if for any particular i, $\tilde{\beta}_i$ is large, then the adaptive lasso estimate for β_i experiences less shrinkage due to a low value of the weight w_i and corresponding low penalty. This implies low bias for the estimate of that particular β_i.

There are many ways to choose the initial estimate $\tilde{\beta}$. When $n > p$, one can take the ordinary least squares estimate as $\tilde{\beta}$ (Zou, 2006). When $p \geq n$, for $i = 1, \ldots, p$, one can take $\tilde{\beta}_i$ to be the ordinary least squares estimate for the univariate regression problem with Y as the response and $X_{\cdot i}$, the ith column of X, as the covariate and no intercept term. Particularly for high dimensional models or in case of variable selection problems, it is beneficial to take the lasso estimate with a cross-validated tuning parameter as $\tilde{\beta}$, because the two-step setup of the adaptive lasso formulation then yields a sparser solution than the usual lasso estimate (Bühlmann and Van De Geer, 2011).

The sparsness inducing property of the adaptive lasso can be amplified by extending it to a multi-step method instead of a two-step method. Here, at each step, the weights are computed based on the estimate obtained at the last step (Bühlmann and Meier, 2008).

Here again, the tuning parameter λ of the adaptive lasso can be selected via cross validation while computing the estimate.

The basic idea of the adaptive lasso is of introducing weights to individual coefficients, based on which we can control the level of shrinking of those coefficients in our adaptive lasso estimate. Different customizations of the usual lasso or elastic net formulations are possible based on this concept of weight introduction.

1.1.5 Group lasso

In some situations, it is required that we either select or drop a group of covariate variables together in our model. For example, if a covariate variable is categorical in nature, its levels are represented using dummy variables. Consider the following problem for illustration. Suppose we have one categorical covariate, Z, with K levels $z_1, \ldots, z_K$, and p numeric covariates, which we represent using the covariate matrix X. Then, our regression model is $Y = \sum_{i=1}^{K} \alpha_i U_i + X\beta$, where $U_i = 1$ if $Z = z_i$ and 0 otherwise. Naturally, we would like to either include all

of the variables $U_1, \ldots, U_K$ together in the model or drop them together. But the usual lasso has no such provision.

This problem is addressed by the group lasso method proposed by Yuan and Lin (2006). Suppose the covariate variables are partitioned in G groups $X_1, \ldots, X_G$. Then, the group lasso loss function is

$$L_{group}(\beta_1, \ldots, \beta_G) = \frac{1}{2}\left\| Y - \sum_{g=1}^{G} X_g \beta_g \right\|_2^2 + \lambda \sum_{g=1}^{G} w_g \|\beta_g\|_2.$$

Here, w_g's are weights assigned to the groups. One may take $w_g = 1$ for all g, which assigns equal weights to all the groups. However, one may also want to make larger groups less likely to be selected than a smaller group. The way to do that would be to pick a weighing scheme which assigns higher weights to larger groups. In particular, the authors Yuan and Lin (2006) recommended to take $w_g = s_g$, where s_g is the size of the gth group, $g = 1, \ldots, G$, to address this problem. We shall take $w_g = 1$ for all g for the sake of simplicity. As before, $\lambda \geq 0$ is a tuning parameter.

We minimize $L_{group}(\beta_1, \ldots, \beta_G)$ with respect to $\beta_1, \ldots, \beta_G$ to get the group lasso estimate. When all the group sizes are 1, the group lasso method coincides with the usual lasso method.

One drawback of the ususal group lasso is that it does not induce sparsity within the groups, i.e., if a group is selected in the model, all the coefficients within the group are usually non-zero. The casue of this is the l_2 norm within the groups, similar to the ridge regression. This problem can be mitigated using the underlying idea of the elastic net, and modifying the group lasso loss function to

$$L_{modgroup}(\beta_1, \ldots, \beta_G) = \frac{1}{2}\left\| Y - \sum_{g=1}^{G} X_g \beta_g \right\|_2^2 + \lambda \sum_{g=1}^{G} [(1-\alpha)\|\beta_g\|_2 + \alpha \|\beta_g\|_1],$$

where $0 \leq \alpha \leq 1$. When $\alpha = 1$, the modified loss function $L_{modgroup}(\beta_1, \ldots, \beta_G)$ reduces to the usual lasso loss function, yielding the usual lasso estimate.

The tuning parameter λ is selected based on cross validation. One may choose α either through cross validation, or may fix it beforehand.

1.1.6 Fused lasso

Sometimes, there is a spatial or temporal structure behind the data, and we want a degree of smoothness among the adjacent coefficents. Suppose the coefficients β_i correspond to some quantity recorded on a one dimensional grid, and we do not want β_{i+1} and β_{i-1} to be much different from β_i. To achieve this goal, we

may take the loss function as

$$L_{fused}(\beta) = \frac{1}{2}\|Y - X\beta\|_2^2 + \lambda_1\|\beta\|_1 + \lambda_2 \sum_{i=2}^{p} |\beta_i - \beta_{i-1}|.$$

Here, $\lambda_1 \geq 0, \lambda_2 \geq 0$ are tuning parameters. For positive λ_2, higher differences among consecutive coefficients are penalized higher. For spatial models, the coefficients may correspond to entities on a two or higher dimensional grid, and the concept of *adjacent* coefficients is meaningful. To impose a degree of smoothness among the adjacent coefficients, we may use the loss function

$$L_{fusedgrid}(\beta) = \frac{1}{2}\|Y - X\beta\|_2^2 + \lambda_1\|\beta\|_1 + \lambda_2 \sum_{i,j \in A} |\beta_i - \beta_j|,$$

where A is the set of all pairs of adjacent indices. This way of penalization imposes a degree of *fusion* among the adjacent coefficients (and hence the name fused lasso).

In some cases, we may not even have a regression problem, but the idea behind the fused lasso can still be applied. For example, suppose the response values are recorded on a one dimensional spatial or temporal grid, and we want to model the mean of the response variable. The mean is here a function over the grid, and we do not want much variation over adjacent grid points. To achieve that, we may take the loss function as

$$L_{simplefused}(\beta) = \frac{1}{2}\|Y - \beta\|_2^2 + \lambda_1\|\beta\|_1 + \lambda_2 \sum_{i=2}^{p} |\beta_i - \beta_{i-1}|.$$

The methodology of fused lasso was proposed by Tibshirani et al. (2005).

We have two tuning parameters here. Using careful arguments, some procedures for computing the fused lasso estimate reduce the problem of selecting two tuning parameters to the problem of selecting just one. Then, that one tuning parameter may be selected via cross validation. Other procedures also exist.

1.1.7 Other regularizations

Above, we have described several types of regularizations based on variations of the penalty term in the loss function. Through combinations of the underlying ideas behind them, many other forms of regularizations are possible. However, we have also change the first term of the loss function, which is fixed to be a squared error loss in all the formulations above. Observe that the squared error loss can be interpreted as the negative of the log likelihood of a Gaussian distribution (the associated constants would not matter in the minimization problem). So, based on this idea, we can readily construct other loss functions based on the negatives of the log likelihoods of other distributions.

We can also formulate such regularizations for generalized linear models, where, rather than modeling $E[Y]$ as a linear function of the covariate X, we model $g(E[Y]) = X\beta$.

In any such formulations, as long as the loss function is convex in β, solving the the minimization problem is not hard. However, some non-convex formulations also exist, but for general non-convex problems, getting the solution is computationally hard, and actually reaching the global minimum may not be guaranteed.

1.2 Numerical demonstration

We shall use the `glmnet` package in R for the computations related to lasso, ridge regression, elastic net and adaptive lasso. The `glmnet` package is developed by Friedman et al. (2021).

Let Σ be a 100×100 covariance matrix defined by $\sigma_{ij} = 0.5 + 0.5\mathbb{I}(i = j)$. Let X be a 100 dimensional random vector with $X \sim N_{100}(0, \Sigma)$. Denote the ith coordinate of X as X_i, $i = 1, \ldots, 100$. We consider the following regression problem: $Y = (1 - X_1 + 2X_2 - 3X_3 + 4X_4 - 5X_5 + 6X_6) + e$, where e is an error variable independent of X. We have a sample of size 30 on Y and X, based on which we shall estimate the coefficient vector using different types of regularizations.

```
# Data generation; setting the seed for reproducible outcome
set.seed(1234)
Mu = rep(0, 100)
SIGMA = matrix(0.5, nrow = 100, ncol = 100) +
  diag(0.5, nrow = 100, ncol = 100)
X = MASS::mvrnorm(n = 30, mu = Mu, Sigma = SIGMA)
X_with_intercept = cbind(1, X)
Beta = c(c(1, -1, 2, -3, 4, -5, 6), rep(0, (100 - 6)))
Y = X_with_intercept %*% Beta + rnorm(30)
```

First we need to install the package `glmnet`.

```
install.packages('glmnet')
```

1.2.1 Lasso computation

We now demonstrate the lasso method on this model.

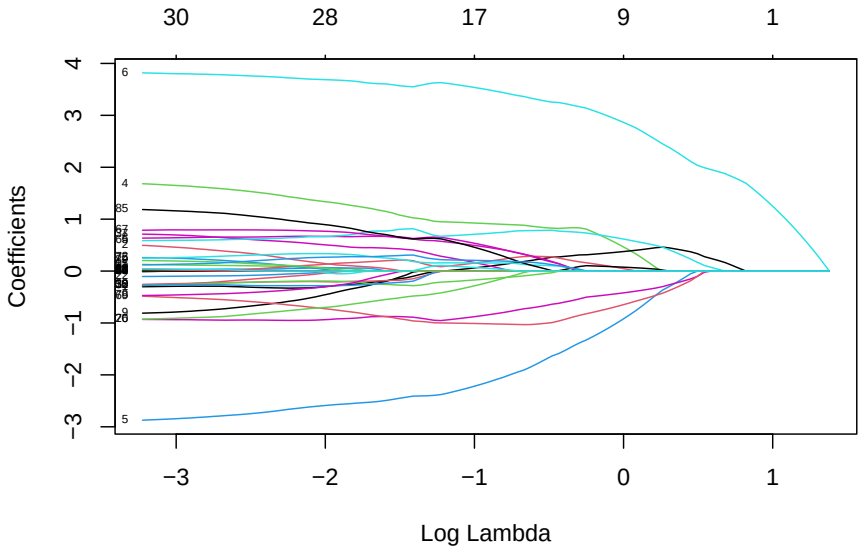
```
# Loading the 'glmnet' package
require(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.1.2
```

```
# Constructing the lasso solution paths
fit_lasso = glmnet(X, Y)
```

We can plot the lasso solution paths using the `plot` function. The indices of the covariate variables are written on the left side. Notice that the coefficient paths for X_6 and X_5 are the most prominent, and then comes X_4. Do you understand the underlying cause?

```
plot(fit_lasso, xvar = 'lambda', label = TRUE)
```

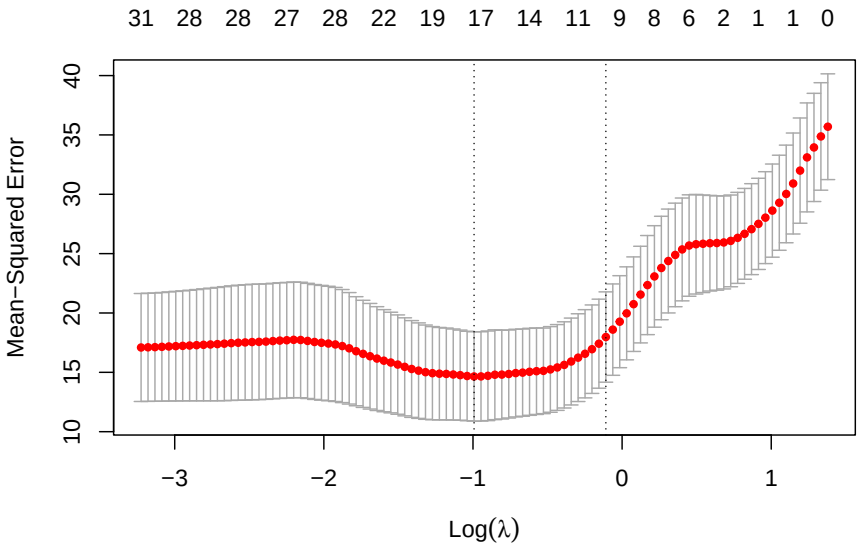


Using the function `cv.glmnet`, we can find the value of the cross-validated tuning parameter λ.

```
cv_fit_lasso = cv.glmnet(X, Y)
```

We can visually inspect the mean-squared error for different values of lambda; the two bars along the mean-squared error curve are the upper and the lower standard deviation curves.

```
plot(cv_fit_lasso)
```



In the plot above, we note that two specific λ values are marked by vertical dotted lines. One of them, denoted by `lambda.min`, is the value of λ which corresponds to the minimum cross-validated error. The other one, denoted by `lambda.1se`, is the largest value of λ such that the corresponding cross-validated error is within one standard error of the minimum. This value of λ corresponds to the *most regularized model* with the cross-validated error being within one standard error of the minimum cross-validated error. In case of the lasso, since with increasing λ the number of non-zero coefficients decreases, the number of non-zero coefficients corresponding to `lambda.1se` is typically lower than that corresponding to `lambda.min`.

We can access the lambda value corresponding to the minimum cross-validated error in the following way.

```
lambda_min_lasso = cv_fit_lasso$lambda.min
writeLines(paste('Cross-validated lambda for lasso:',
                 lambda_min_lasso))
```

```
## Cross-validated lambda for lasso: 0.370549314393551
```

We can get the coefficients of the fitted model at the cross-validated lambda in the following way.

```
coefficients_lasso = coef(cv_fit_lasso, s = 'lambda.min')
print(coefficients_lasso)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
```

```
##                       s1
## (Intercept)  1.80973757
## V1           .
## V2           .
## V3           .
## V4           0.92449282
## V5          -2.21218789
## V6           3.53342728
## V7           .
## V8           0.53117420
## V9           .
## V10          .
## V11          .
## V12          .
## V13          .
## V14          .
## V15          .
## V16          .
## V17          .
## V18          .
## V19          .
## V20          .
## V21          .
## V22          .
## V23          .
## V24          .
## V25          .
## V26         -0.87597982
## V27          0.06242635
## V28          .
## V29          .
## V30          .
## V31          .
## V32         -0.17019138
## V33          .
## V34          .
## V35          .
## V36          .
## V37          0.16030531
## V38          .
## V39          .
## V40          .
## V41          .
## V42          .
## V43          .
## V44          .
```

```
## V45          .
## V46          .
## V47          0.15672602
## V48          .
## V49          .
## V50          .
## V51          .
## V52          .
## V53          .
## V54          .
## V55          .
## V56          .
## V57          .
## V58          .
## V59          .
## V60          .
## V61          .
## V62          .
## V63          .
## V64          .
## V65          0.70974838
## V66          .
## V67          0.49131462
## V68          .
## V69         -1.01075761
## V70         -0.27985153
## V71          .
## V72          .
## V73          .
## V74          .
## V75          0.15711661
## V76          .
## V77          .
## V78          .
## V79          .
## V80          .
## V81          .
## V82          .
## V83          .
## V84          .
## V85          0.45198207
## V86          .
## V87          .
## V88          .
## V89          .
## V90          .
```

```
## V91          0.20589889
## V92          .
## V93          .
## V94          .
## V95          .
## V96          .
## V97          .
## V98          0.06768265
## V99          .
## V100         .
```

As can be seen above, the coefficients are returned in a sparse matrix format. We can convert that to the usual numeric format in the following way. Remember that the intercept term will occupy the first position in the resulting vector.

```
coefficients_lasso = as.numeric(coefficients_lasso)

# Finding the number of zero coefficients
number_zero_coefficients_lasso = sum(coefficients_lasso == 0)
writeLines(paste('Number of zero coefficients in lasso for',
                 'cross-validated lambda:\n',
                 number_zero_coefficients_lasso))
```

```
## Number of zero coefficients in lasso for cross-validated lambda:
##  83
```

```
# Finding which covariates have a non-zero coefficient in the
# fitted model; the '-1' accounts for the intercept term
indices_nonzero_coefficients_lasso =
  which(coefficients_lasso != 0) - 1
```

To get the coefficients at the value of λ which gives the most regularized model such that the cross-validated error is within one standard error of the minimum, we use the following command.

```
coefficients_lasso_most_regularized =
  coef(cv_fit_lasso, s = 'lambda.1se')
coefficients_lasso_most_regularized =
  as.numeric(coefficients_lasso_most_regularized)

# Finding the number of zero coefficients
number_zero_coefficients_lasso_most_regularized =
  sum(coefficients_lasso_most_regularized == 0)
writeLines(paste('Number of zero coefficients in lasso for \n',
                 'the most regularized model:',
                 number_zero_coefficients_lasso_most_regularized))
```

```
## Number of zero coefficients in lasso for
```

```
##  the most regularized model: 91
```

Note that the number of zero coefficients corresponding to `lambda.1se` is higher than that corresponding to `lambda.min`. We can find the coefficients at some other value of lambda also, in the same way.

```
coefficients_lasso_s = coef(cv_fit_lasso, s = 0.5)
print(coefficients_lasso_s)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept)  1.767178719
## V1           .
## V2           .
## V3           .
## V4           0.886211639
## V5          -1.930076221
## V6           3.373783767
## V7           .
## V8           0.337440156
## V9           .
## V10          .
## V11          .
## V12          .
## V13          .
## V14          .
## V15          .
## V16          .
## V17          .
## V18          .
## V19          .
## V20          .
## V21          .
## V22          .
## V23          .
## V24          .
## V25          .
## V26         -0.739860240
## V27          0.173458826
## V28          .
## V29          .
## V30          .
## V31          .
## V32         -0.099445511
## V33          .
## V34          .
## V35          .
```

```
## V36          .
## V37          .
## V38          .
## V39          .
## V40          .
## V41          .
## V42          .
## V43          .
## V44          .
## V45          .
## V46          .
## V47          0.272255598
## V48          .
## V49          .
## V50          .
## V51          .
## V52          .
## V53          .
## V54          .
## V55          .
## V56          .
## V57          .
## V58          .
## V59          .
## V60          .
## V61          .
## V62          .
## V63          .
## V64          .
## V65          0.775881117
## V66          .
## V67          0.342180469
## V68          .
## V69         -1.030152116
## V70         -0.049587703
## V71          .
## V72          .
## V73          .
## V74          .
## V75          0.136389337
## V76          .
## V77          .
## V78          .
## V79          .
## V80          .
## V81          .
```

```
## V82          .
## V83          .
## V84          .
## V85          0.173826167
## V86          .
## V87          .
## V88          .
## V89          .
## V90          .
## V91          0.178492348
## V92          .
## V93          .
## V94          .
## V95          .
## V96          .
## V97          .
## V98          0.002456585
## V99          .
## V100         .
```

To predict the response at some value of the covariates, we use the following command. The argument `newx` must be supplied as a matrix and not a vector.

```
x_0 = matrix(rnorm(100), nrow = 1, ncol = 100)
predicted_Y = predict(cv_fit_lasso, newx = x_0, s = 'lambda.min')
writeLines(paste('Predicted response:', predicted_Y))
```

```
## Predicted response: 7.6561153394393
```

We can predict at several sets of values of the covariates using the same command.

```
x_0_values = matrix(rnorm(3*100), nrow = 3, ncol = 100)
predict(cv_fit_lasso, newx = x_0_values, s = 'lambda.min')
```

```
##      lambda.min
## [1,]   4.901129
## [2,]  -6.576428
## [3,]  10.584152
```

We can also predict at different values of lambda.

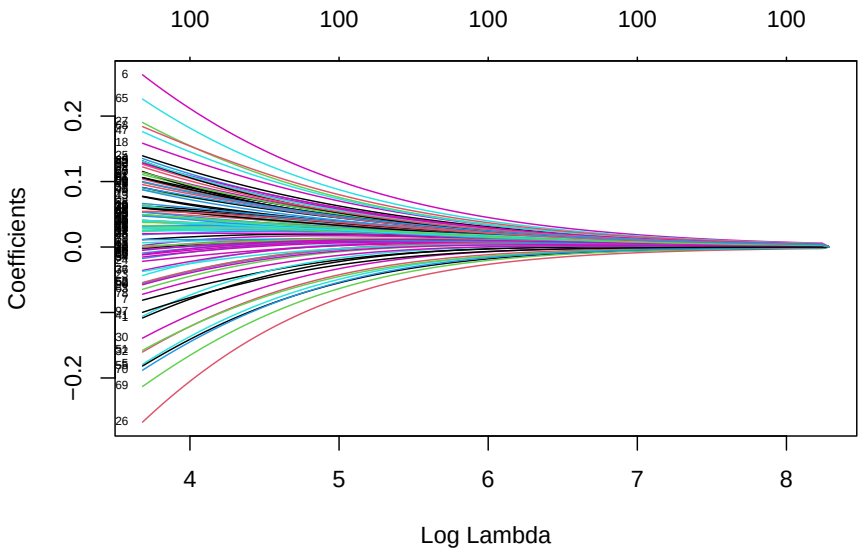
```
x_0_values = matrix(rnorm(3*100), nrow = 3, ncol = 100)
predict(cv_fit_lasso, newx = x_0_values, s = c(0.5, 0.8))
```

```
##              s1         s2
## [1,] -1.3774519 -0.5203943
## [2,]  2.0691667  1.9047106
## [3,] -0.5446435 -0.9182558
```

1.2.2 Ridge regression computation

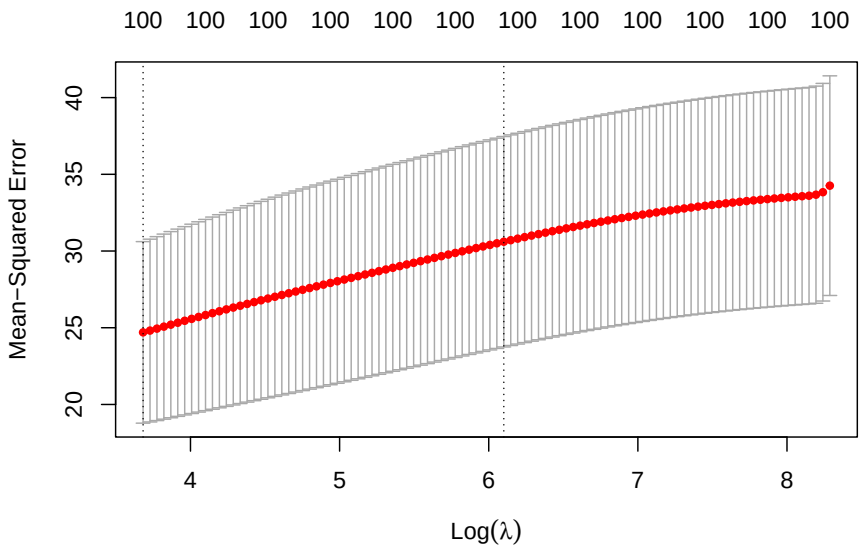
We next demonstrate the ridge regression method. The commands for the ridge regression method are identical with the lasso, except passing the value of another parameter `alpha = 0` to the `glmnet` function, which overwrites the default value of `alpha = 1`. The default value `alpha = 1` corresponds to the lasso method, and the value `alpha = 0` corresponds to ridge regression method.

```
# Constructing the ridge regression solution paths; 'alpha = 0'
# corresponds to the ridge regression method
fit_ridge = glmnet(X, Y, alpha = 0)

# Plotting the ridge regression solution paths
plot(fit_ridge, xvar = 'lambda', label = TRUE)
```



Note that the coefficients do not become zero with inreasing λ. The function 'cv.glmnet' with `alpha = 0` yields the value of the cross-validated λ for the ridge regression.

```
# 'alpha = 0' corresponds to the ridge regression
cv_fit_ridge = cv.glmnet(X, Y, alpha = 0)

# Plotting the mean-squared error for different values of lambda
plot(cv_fit_ridge)
```




```
# Accessing the lambda value corresponding to the minimum
# cross-validated error
lambda_min_ridge = cv_fit_ridge$lambda.min
writeLines(paste('Cross-validated lambda for ridge:',
                 lambda_min_ridge))
```

```
## Cross-validated lambda for ridge: 39.7327883962604
```

Below we obtain the coefficients of the fitted ridge regression model at the cross-validated lambda.

```
coefficients_ridge = coef(cv_fit_ridge, s = 'lambda.min')
print(coefficients_ridge)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                          s1
## (Intercept)  1.7109491793
## V1          -0.1083085252
## V2           0.0289382847
## V3           0.0874521464
## V4           0.1351638781
## V5          -0.1791263703
## V6           0.2630018842
## V7          -0.0811946856
## V8           0.0953859520
## V9          -0.0567241489
```

```
## V10         -0.0092322109
## V11          0.0237973432
## V12          0.0023652436
## V13          0.0627102051
## V14          0.0407541258
## V15          0.0643875971
## V16         -0.0029789548
## V17          0.0239322364
## V18          0.1584882590
## V19          0.0775910315
## V20          0.0384100813
## V21          0.0988288390
## V22         -0.0006264409
## V23         -0.0435152903
## V24         -0.0218175823
## V25          0.1393288854
## V26         -0.2674413054
## V27          0.1901822555
## V28          0.0189121325
## V29          0.0622712798
## V30         -0.1391918711
## V31         -0.0019822391
## V32         -0.1603607172
## V33          0.0386014016
## V34         -0.0175624312
## V35          0.0396806825
## V36         -0.0358246174
## V37          0.0910033961
## V38          0.0600965338
## V39          0.0374900113
## V40          0.0471962575
## V41         -0.1057321981
## V42          0.1279890565
## V43          0.0769178361
## V44          0.0532354798
## V45          0.0255123294
## V46          0.0874539653
## V47          0.1761787143
## V48         -0.0050762766
## V49          0.0596021880
## V50          0.1263894979
## V51         -0.1577883648
## V52         -0.0553016596
## V53          0.0495713022
## V54         -0.0152558777
## V55         -0.1817212669
```

```
## V56          0.1037481509
## V57          0.1107477311
## V58          0.0905209792
## V59          0.0279373188
## V60          0.1140120010
## V61          0.1060607720
## V62         -0.0543002733
## V63         -0.0646601364
## V64          0.0535504253
## V65          0.2259343785
## V66         -0.0119965275
## V67          0.1153978113
## V68          0.1839346145
## V69         -0.2130429880
## V70         -0.1881321515
## V71          0.0064749093
## V72          0.1286295454
## V73          0.0595155004
## V74          0.0307595428
## V75          0.1139984622
## V76          0.0322028103
## V77         -0.0374223743
## V78         -0.0721740256
## V79          0.0112816650
## V80         -0.0009252485
## V81          0.0326936742
## V82          0.0665158297
## V83          0.0296031295
## V84          0.0204508045
## V85          0.1315162061
## V86          0.0545900253
## V87          0.0408206633
## V88          0.0112714252
## V89          0.1312721221
## V90         -0.0575812931
## V91          0.1059293356
## V92          0.1223877048
## V93          0.0013630864
## V94          0.0490835636
## V95          0.0408392873
## V96         -0.0113738133
## V97         -0.0997274582
## V98          0.0990380910
## V99          0.0497533757
## V100         0.1000894670
```

Note that the coefficients are nonzero unlike the lasso. In case of the ridge regression, the value `lambda.1se` is not so interesting as in case of the lasso, as here there is no question of reducing any coefficient to zero.

```
coefficients_ridge = as.numeric(coefficients_ridge)

# Finding the number of nonzero coefficients; the '-1' accounts
# for the intercept term
number_nonzero_coefficients_ridge =
  sum(coefficients_ridge != 0) - 1
writeLines(paste('Number of nonzero coefficients for',
                 'the ridge regression:\n',
                 number_nonzero_coefficients_ridge))
```

```
## Number of nonzero coefficients for the ridge regression:
##  100
```

Finding the coefficients at some other value of lambda is the same as in the case of the lasso.

```
coefficients_ridge_s = coef(cv_fit_ridge, s = 0.5)
print(coefficients_ridge_s)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                          s1
## (Intercept)  1.7109491793
## V1          -0.1083085252
## V2           0.0289382847
## V3           0.0874521464
## V4           0.1351638781
## V5          -0.1791263703
## V6           0.2630018842
## V7          -0.0811946856
## V8           0.0953859520
## V9          -0.0567241489
## V10         -0.0092322109
## V11          0.0237973432
## V12          0.0023652436
## V13          0.0627102051
## V14          0.0407541258
## V15          0.0643875971
## V16         -0.0029789548
## V17          0.0239322364
## V18          0.1584882590
## V19          0.0775910315
## V20          0.0384100813
## V21          0.0988288390
```

```
## V22         -0.0006264409
## V23         -0.0435152903
## V24         -0.0218175823
## V25          0.1393288854
## V26         -0.2674413054
## V27          0.1901822555
## V28          0.0189121325
## V29          0.0622712798
## V30         -0.1391918711
## V31         -0.0019822391
## V32         -0.1603607172
## V33          0.0386014016
## V34         -0.0175624312
## V35          0.0396806825
## V36         -0.0358246174
## V37          0.0910033961
## V38          0.0600965338
## V39          0.0374900113
## V40          0.0471962575
## V41         -0.1057321981
## V42          0.1279890565
## V43          0.0769178361
## V44          0.0532354798
## V45          0.0255123294
## V46          0.0874539653
## V47          0.1761787143
## V48         -0.0050762766
## V49          0.0596021880
## V50          0.1263894979
## V51         -0.1577883648
## V52         -0.0553016596
## V53          0.0495713022
## V54         -0.0152558777
## V55         -0.1817212669
## V56          0.1037481509
## V57          0.1107477311
## V58          0.0905209792
## V59          0.0279373188
## V60          0.1140120010
## V61          0.1060607720
## V62         -0.0543002733
## V63         -0.0646601364
## V64          0.0535504253
## V65          0.2259343785
## V66         -0.0119965275
## V67          0.1153978113
```

```
## V68          0.1839346145
## V69         -0.2130429880
## V70         -0.1881321515
## V71          0.0064749093
## V72          0.1286295454
## V73          0.0595155004
## V74          0.0307595428
## V75          0.1139984622
## V76          0.0322028103
## V77         -0.0374223743
## V78         -0.0721740256
## V79          0.0112816650
## V80         -0.0009252485
## V81          0.0326936742
## V82          0.0665158297
## V83          0.0296031295
## V84          0.0204508045
## V85          0.1315162061
## V86          0.0545900253
## V87          0.0408206633
## V88          0.0112714252
## V89          0.1312721221
## V90         -0.0575812931
## V91          0.1059293356
## V92          0.1223877048
## V93          0.0013630864
## V94          0.0490835636
## V95          0.0408392873
## V96         -0.0113738133
## V97         -0.0997274582
## V98          0.0990380910
## V99          0.0497533757
## V100         0.1000894670
```

We can also predict at several sets of values of the covariates and at different values of lambda for the fitted ridge regression model. The argument **newx** always must be a matrix, even if we are predicting at only one set of values of the covariates.

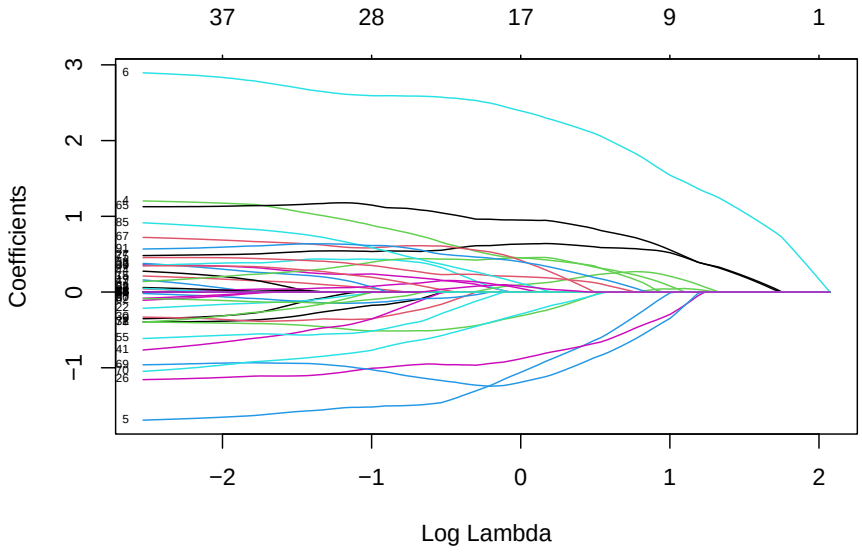
```
x_0_values = matrix(rnorm(3*100), nrow = 3, ncol = 100)
predict(cv_fit_ridge, newx = x_0_values, s = c(0.5, 0.8))
```

```
##             s1        s2
## [1,] 2.2248989 2.2248989
## [2,] 2.5238977 2.5238977
## [3,] 0.3758271 0.3758271
```

1.2.3 Elastic net computation

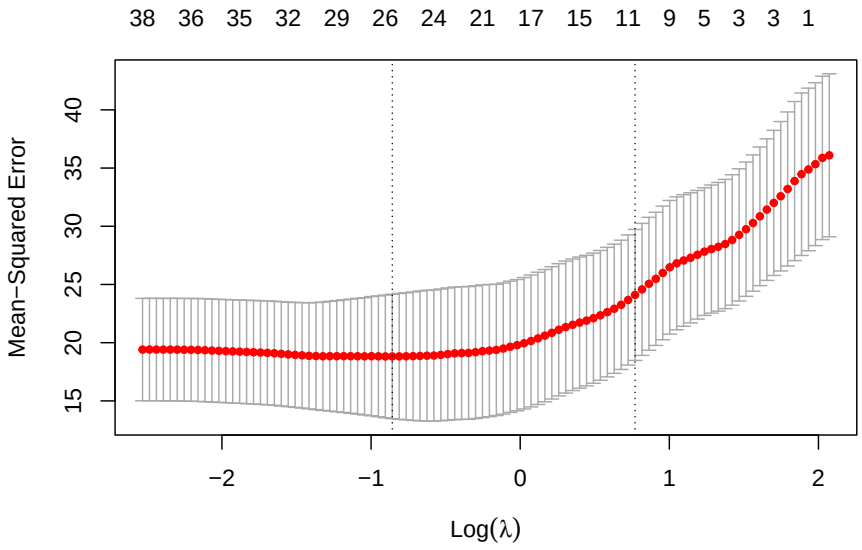
From the description of the elastic net in subsection 1.1.3, and the codes for computing the lasso and the ridge regression solutions in subsection 1.2.1 and subsection 1.2.2 respectively, you might have guessed that the function `glmnet` actually computed the elastic net solutions for different values of the parameter α: $\alpha = 1$ corresponds to the lasso and $\alpha = 0$ corresponds to the ridge regression method. So, the same code with a value of the argument `alpha` in $(0, 1)$ will return the elastic net solution for that value of α. Below, we demonstrate the computation for $\alpha = 0.5$.

```
# Constructing the elastic net solution paths for 'alpha = 0.5'
fit_elnet = glmnet(X, Y, alpha = 0.5)

# Plotting the elastic net solution paths
plot(fit_elnet, xvar = 'lambda', label = TRUE)
```



Note that the coefficients become zero with inreasing λ like in the case of the lasso.

```
# Cross-validation for elastic net with 'alpha = 0.5'
cv_fit_elnet = cv.glmnet(X, Y, alpha = 0.5)

# Plotting the mean-squared error for different values of lambda
plot(cv_fit_elnet)
```




```
# Accessing the lambda value corresponding to the minimum
# cross-validated error
lambda_min_elnet = cv_fit_elnet$lambda.min
writeLines(paste('Cross-validated lambda for elastic net:',
                 lambda_min_elnet))
```

```
## Cross-validated lambda for elastic net: 0.424083882576011
```

```
# Coefficients of the fitted elastic net at the cross-validated
# lambda
coefficients_elnet = coef(cv_fit_elnet, s = 'lambda.min')
print(coefficients_elnet)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept)  2.11279990
## V1          -0.15092287
## V2           .
## V3           .
## V4           0.82409055
## V5          -1.50317945
## V6           2.59241473
## V7           .
## V8           0.21545204
## V9           .
## V10          .
```

```
## V11          .
## V12          .
## V13          .
## V14          .
## V15          0.02007609
## V16          .
## V17          .
## V18          .
## V19          .
## V20          .
## V21          .
## V22          .
## V23          .
## V24          .
## V25          .
## V26         -0.98502236
## V27          0.54189086
## V28          .
## V29          .
## V30         -0.30190044
## V31          .
## V32         -0.51413064
## V33          .
## V34          .
## V35          .
## V36          .
## V37          0.42556645
## V38          .
## V39          .
## V40          .
## V41         -0.25103695
## V42          .
## V43          .
## V44          .
## V45          .
## V46          .
## V47          0.42890488
## V48          .
## V49          .
## V50          .
## V51         -0.11850576
## V52          .
## V53          .
## V54          .
## V55         -0.46259584
## V56          .
```

```
## V57          .
## V58          .
## V59          .
## V60          .
## V61          .
## V62          .
## V63          .
## V64          .
## V65          1.11118958
## V66          .
## V67          0.60206411
## V68          .
## V69         -1.07119524
## V70         -0.70044279
## V71          .
## V72          0.09785610
## V73          .
## V74          .
## V75          0.31266992
## V76          .
## V77          .
## V78          .
## V79          .
## V80          .
## V81          .
## V82          .
## V83          .
## V84          .
## V85          0.51997636
## V86          .
## V87          .
## V88          .
## V89          0.15174385
## V90         -0.07005343
## V91          0.60397991
## V92          .
## V93          .
## V94          .
## V95          .
## V96          .
## V97          .
## V98          .
## V99          .
## V100         .
```

Note that many of the coefficients are zero like the lasso. However, the number

of zero coefficients is lower than the lasso.

```
coefficients_elnet = as.numeric(coefficients_elnet)

# Finding the number of nonzero coefficients; the '-1' accounts
# for the intercept term
number_nonzero_coefficients_elnet =
  sum(coefficients_elnet != 0) - 1
writeLines(paste('Number of nonzero coefficients for',
                 'the elastic net:',
                 number_nonzero_coefficients_elnet))
```

```
## Number of nonzero coefficients for the elastic net: 25
```

```
# Computing the coefficients at some other value of lambda
coefficients_elnet_s = coef(cv_fit_elnet, s = 0.5)
print(coefficients_elnet_s)
```

```
## 101 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept)  2.04051028
## V1          -0.09756072
## V2           .
## V3           .
## V4           0.75068788
## V5          -1.49223014
## V6           2.58752206
## V7           .
## V8           0.18902712
## V9           .
## V10          .
## V11          .
## V12          .
## V13          .
## V14          .
## V15          .
## V16          .
## V17          .
## V18          .
## V19          .
## V20          .
## V21          .
## V22          .
## V23          .
## V24          .
## V25          .
## V26         -0.95807495
```

```
## V27          0.53735119
## V28          .
## V29          .
## V30         -0.23907252
## V31          .
## V32         -0.51110747
## V33          .
## V34          .
## V35          .
## V36          .
## V37          0.39406628
## V38          .
## V39          .
## V40          .
## V41         -0.12997961
## V42          .
## V43          .
## V44          .
## V45          .
## V46          .
## V47          0.44055872
## V48          .
## V49          .
## V50          .
## V51         -0.09072844
## V52          .
## V53          .
## V54          .
## V55         -0.39664432
## V56          .
## V57          .
## V58          .
## V59          .
## V60          .
## V61          .
## V62          .
## V63          .
## V64          .
## V65          1.08905420
## V66          .
## V67          0.61168096
## V68          .
## V69         -1.12571292
## V70         -0.63958967
## V71          .
## V72          0.11705065
```

```
## V73          .
## V74          .
## V75          0.27052397
## V76          .
## V77          .
## V78          .
## V79          .
## V80          .
## V81          .
## V82          .
## V83          .
## V84          .
## V85          0.43965564
## V86          .
## V87          .
## V88          .
## V89          0.09232706
## V90         -0.01848038
## V91          0.56203662
## V92          .
## V93          .
## V94          .
## V95          .
## V96          .
## V97          .
## V98          .
## V99          .
## V100         .
```

```
# Predicting at several sets of values of the covariates and at
# different values of lambda for the fitted elastic net
x_0_values = matrix(rnorm(3*100), nrow = 3, ncol = 100)
predict(cv_fit_elnet, newx = x_0_values, s = c(0.5, 0.8))
```

```
##             s1        s2
## [1,] -6.386536 -4.828346
## [2,]  3.403320  2.583454
## [3,] -3.042642 -3.058112
```

1.2.4 Adaptive lasso computation

Recall the description of the adaptive lasso method in subsection 1.1.4. We shall demonstrate the adaptive lasso with $\gamma = 1$ taking the usual cross-validated lasso solution as the initial estimate.

First we compute the cross-validated lasso solution.

```
# Computing the coefficient for the cross-validated lasso estimate
cv_fit_lasso = cv.glmnet(X, Y)
coefficients_lasso = coef(cv_fit_lasso, s = 'lambda.min')

# Dropping the intercept term in the cross-validated lasso
# estimate
coefficients_lasso = as.numeric(coefficients_lasso)[-1]

# Computing the adaptive lasso estimate. The argument
# 'penalty.factor' assigns weights to the coefficients, with the
# convention of r / 0 = Infinity for r > 0.
cv_fit_adaptive =
  cv.glmnet(X, Y, penalty.factor = 1 / abs(coefficients_lasso))
coefficients_adaptive = coef(cv_fit_adaptive, s = 'lambda.min')

# Dropping the intercept term for the adaptive lasso estimate
coefficients_adaptive = as.numeric(coefficients_adaptive)[-1]
```

Recall from subsection 1.1.4 that the adaptive lasso estimate is sparser than the lasso estimate. From the construction of our simulation setup, we know that only the coefficients of X_1, X_2, X_3, X_4, X_5 and X_6 are nonzero in the underlying model. Let us see the indices of the covariates with nonzero coefficients in the lasso estimate and the adaptive lasso estimate.

```
writeLines(paste('Indices of the covariates with nonzero',
                 'coefficients \n for the lasso estimate:\n',
                 paste(which(coefficients_lasso != 0),
                       collapse = ' ')))
```

```
## Indices of the covariates with nonzero coefficients
##  for the lasso estimate:
##  4 5 6 8 26 27 32 37 47 65 67 69 70 75 85 91 98
```

```
writeLines(paste('Indices of the covariates with nonzero',
                 'coefficients \n for the adaptive lasso',
                 'estimate:\n',
                 paste(which(coefficients_adaptive != 0),
                       collapse = ' ')))
```

```
## Indices of the covariates with nonzero coefficients
##  for the adaptive lasso estimate:
##  4 5 6 8 26 65 67 69 85
```

We note that the lasso estimate has 17 nonzero coordinates, while the adaptive lasso has 9 nonzero coordinates. So, the adaptive lasso estimate is indeed sparser than the lasso estimate. However, both of them erroneously estimate several zero coefficients as nonzero, and fail to capture several nonzero coefficients.

1.2.5 Group lasso computation

For the group lasso computation, we shall use the `gglasso` package by Yang et al. (2020). We shall use the same simulation setup, but the covariates will be divided in groups in the following way: $X_1, \ldots, X_5$ in the first group, $X_6, \ldots, X_10$ in the second group, and so on. We form the group indices below, which will be required.

```
group_indices = c()
for (i in 1:20)
  group_indices = c(group_indices, rep(i, 5))
writeLines('First 20 group indices are:')
```

```
## First 20 group indices are:
```

```
print(group_indices[1:20])
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4
```

The following command installs the package `gglasso`.

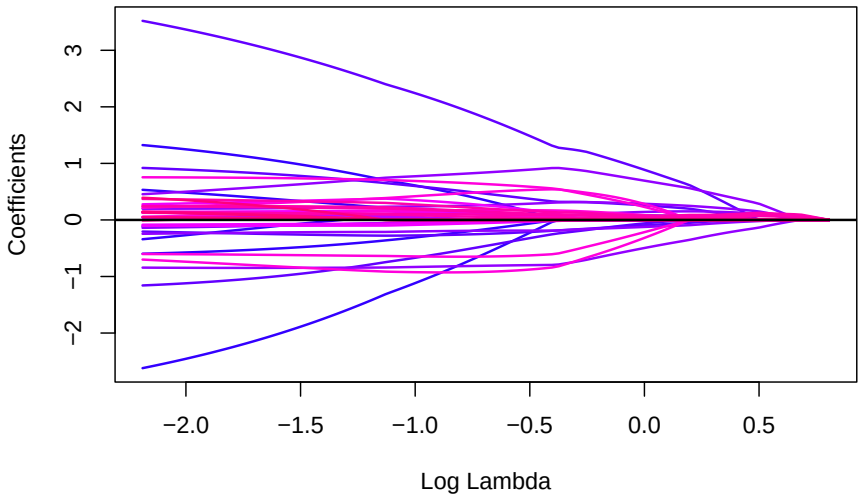
```
install.packages('gglasso')
```

We now demonstrate the computation for the group lasso method. The commands are very similar to those in `glmnet`.

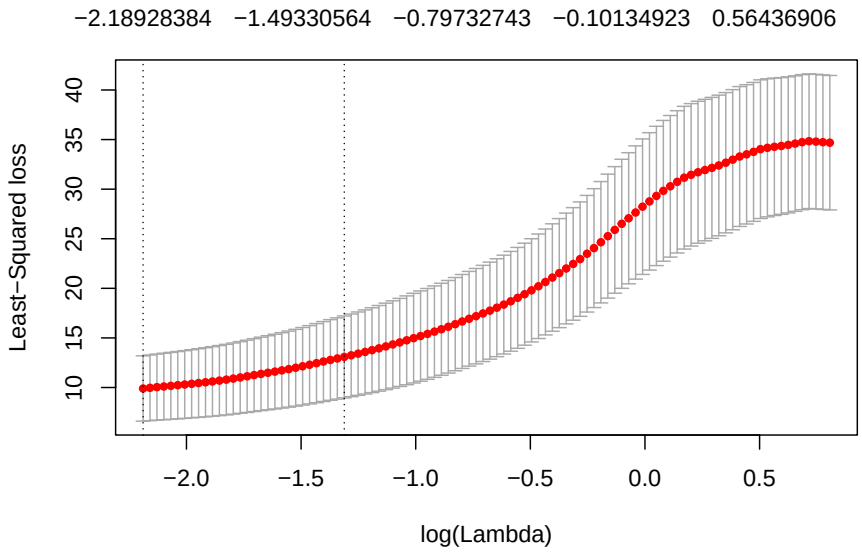
```
# Loading the 'gglasso' package
require(gglasso)
```

```
## Warning: package 'gglasso' was built under R version 4.1.2
```

```
# Constructing the group lasso solution paths
fit_group = gglasso(X, Y, group = group_indices)

# Plotting the ridge regression solution paths; the
# 'label' argument does not work here, unfortunately.
plot(fit_group, xvar = 'lambda')
```




```
# Cross-validation for group lasso; 'nfolds' is a
# cross-validation parameter whose default value
# was 10 in our earlier calculations using the 'glmnet'
# package. Since its default parameter for the 'gglasso'
# package is different, we set that explicitly to 10
# here to maintain uniformity.
cv_fit_group = cv.gglasso(X, Y, group = group_indices,
                          nfolds = 10)

# Plotting the mean-squared error for different values of
# lambda in the group lasso cross-validation
plot(cv_fit_group)
```




```
# Getting the lambda value corresponding to the minimum
# cross-validated error in the group lasso
lambda_min_group = cv_fit_group$lambda.min
writeLines(paste('Cross-validated lambda for group lasso:',
                 lambda_min_group))
```

```
## Cross-validated lambda for group lasso: 0.111996927450172
```

```
# Coefficients of the fitted group lasso at the
# cross-validated lambda
coefficients_group = coef(cv_fit_group, s = 'lambda.min')
print(coefficients_group)
```

```
##                        1
## (Intercept)  2.299964945
## V1          -0.597276460
## V2           0.532650084
## V3          -0.341296086
## V4           1.324008186
## V5          -2.620777857
## V6           3.520635169
## V7          -0.205739578
## V8           0.919717629
## V9          -1.158546608
## V10         -0.115243566
## V11          0.000000000
```

```
## V12          0.000000000
## V13          0.000000000
## V14          0.000000000
## V15          0.000000000
## V16          0.000000000
## V17          0.000000000
## V18          0.000000000
## V19          0.000000000
## V20          0.000000000
## V21          0.000000000
## V22          0.000000000
## V23          0.000000000
## V24          0.000000000
## V25          0.000000000
## V26         -0.843269703
## V27          0.454718935
## V28         -0.139688699
## V29          0.198558378
## V30         -0.245208698
## V31          0.000000000
## V32          0.000000000
## V33          0.000000000
## V34          0.000000000
## V35          0.000000000
## V36          0.000000000
## V37          0.000000000
## V38          0.000000000
## V39          0.000000000
## V40          0.000000000
## V41          0.000000000
## V42          0.000000000
## V43          0.000000000
## V44          0.000000000
## V45          0.000000000
## V46          0.049197887
## V47          0.133784587
## V48          0.006353655
## V49          0.000623473
## V50          0.051280996
## V51          0.000000000
## V52          0.000000000
## V53          0.000000000
## V54          0.000000000
## V55          0.000000000
## V56          0.000000000
## V57          0.000000000
```

```
## V58          0.000000000
## V59          0.000000000
## V60          0.000000000
## V61          0.127846059
## V62         -0.102298726
## V63         -0.084174083
## V64          0.140865202
## V65          0.276307972
## V66          0.055334287
## V67          0.754238086
## V68          0.215257426
## V69         -0.700822526
## V70         -0.602696078
## V71          0.147769008
## V72          0.245420916
## V73          0.242587489
## V74          0.126254804
## V75          0.371727647
## V76          0.000000000
## V77          0.000000000
## V78          0.000000000
## V79          0.000000000
## V80          0.000000000
## V81          0.048401665
## V82         -0.008353210
## V83          0.140023712
## V84          0.139982310
## V85          0.392935904
## V86          0.000000000
## V87          0.000000000
## V88          0.000000000
## V89          0.000000000
## V90          0.000000000
## V91          0.000000000
## V92          0.000000000
## V93          0.000000000
## V94          0.000000000
## V95          0.000000000
## V96          0.000000000
## V97          0.000000000
## V98          0.000000000
## V99          0.000000000
## V100         0.000000000
```

```
# Computing the coefficients at some other value of lambda
# for the group lasso estimate
```

```
coefficients_group_s = coef(cv_fit_group, s = 0.5)
print(coefficients_group_s)
```

```
##                        1
## (Intercept)  2.446545801
## V1          -0.164606301
## V2           0.094789057
## V3           0.120037112
## V4           0.347808507
## V5          -0.593803459
## V6           1.804429746
## V7          -0.246614547
## V8           0.464199928
## V9          -0.460855429
## V10          0.019794050
## V11          0.000000000
## V12          0.000000000
## V13          0.000000000
## V14          0.000000000
## V15          0.000000000
## V16          0.000000000
## V17          0.000000000
## V18          0.000000000
## V19          0.000000000
## V20          0.000000000
## V21          0.000000000
## V22          0.000000000
## V23          0.000000000
## V24          0.000000000
## V25          0.000000000
## V26         -0.810113871
## V27          0.839345222
## V28          0.008647552
## V29          0.261184248
## V30         -0.190066878
## V31          0.000000000
## V32          0.000000000
## V33          0.000000000
## V34          0.000000000
## V35          0.000000000
## V36          0.000000000
## V37          0.000000000
## V38          0.000000000
## V39          0.000000000
## V40          0.000000000
```

```
## V41          0.000000000
## V42          0.000000000
## V43          0.000000000
## V44          0.000000000
## V45          0.000000000
## V46          0.007239078
## V47          0.015033387
## V48          0.003418754
## V49          0.004555346
## V50          0.010936767
## V51          0.000000000
## V52          0.000000000
## V53          0.000000000
## V54          0.000000000
## V55          0.000000000
## V56          0.000000000
## V57          0.000000000
## V58          0.000000000
## V59          0.000000000
## V60          0.000000000
## V61          0.131674540
## V62         -0.054091098
## V63         -0.033632054
## V64          0.092040935
## V65          0.258461382
## V66          0.172478602
## V67          0.634772294
## V68          0.494381080
## V69         -0.917556135
## V70         -0.647221136
## V71          0.053041142
## V72          0.137248090
## V73          0.147020100
## V74          0.094450596
## V75          0.202158872
## V76          0.000000000
## V77          0.000000000
## V78          0.000000000
## V79          0.000000000
## V80          0.000000000
## V81          0.000000000
## V82          0.000000000
## V83          0.000000000
## V84          0.000000000
## V85          0.000000000
## V86          0.000000000
```

```
## V87          0.000000000
## V88          0.000000000
## V89          0.000000000
## V90          0.000000000
## V91          0.000000000
## V92          0.000000000
## V93          0.000000000
## V94          0.000000000
## V95          0.000000000
## V96          0.000000000
## V97          0.000000000
## V98          0.000000000
## V99          0.000000000
## V100         0.000000000
```

Notice that the group lasso method reduces to zero all the coefficients together in any group. Next, we predict the group lasso estimate at some sets of values of the covariates.

```
# Predicting at several sets of values of the covariates and at
# different values of lambda for the fitted group lasso; here
# 'type' is an argument whose value 'link' corresponds to
# regression prediction.
x_0_values = matrix(rnorm(3*100), nrow = 3, ncol = 100)
predict(cv_fit_group, newx = x_0_values, s = c(0.5, 0.8),
        type = 'link')
```

```
##                1         2
## [1,]  5.10055137  4.8965983
## [2,]  0.41015246 -0.2829483
## [3,] -0.02520761  0.8646178
```

1.2.6 Fused lasso computation

We shall demonstrate the idea of the fused lasso in a simpler model unlike the simulated regression setup used in the previous cases. Assume the response values are Y_i, where $i = 1, \ldots, n$, and $Y_i \sim \theta_i + e_i$, where e_i are independent and identically distributed $N(0, 0.25)$ random variables. Suppose the sequence $\{\theta_i\}$ is piecewise constant, such that $\theta_1 = \ldots = \theta_{20} = 2$, $\theta_{21} = \ldots = \theta_{30} = 3$, $\theta_{31} = \ldots = \theta_{40} = 0$, $\theta_{41} = \ldots = \theta_{60} = 5$, $\theta_{61} = \ldots = \theta_{85} = 1$ and $\theta_{85} = \ldots = \theta_{100} = 0$. We want to find

$$\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^n} \left[\sum_{i=1}^{n} (Y_i - \theta_i)^2 + \lambda \sum_{i=1}^{n-1} |\theta_i - \theta_{i+1}| \right].$$

Below, we generate a sample from this model with $n = 100$.

```
# Data generation for fused lasso with fixed seed
set.seed(1234)
Theta = c(rep(2, 20), rep(3, 10), rep(0, 10), rep(5, 20),
          rep(1, 25), rep(0, 15))
Y = Theta + rnorm(100, mean = 0, sd = sqrt(0.25))
```

We need the `genlasso` package (Arnold and Tibshirani, 2020) for the fused lasso computation. The following command would install this package.

```
install.packages('genlasso')
```

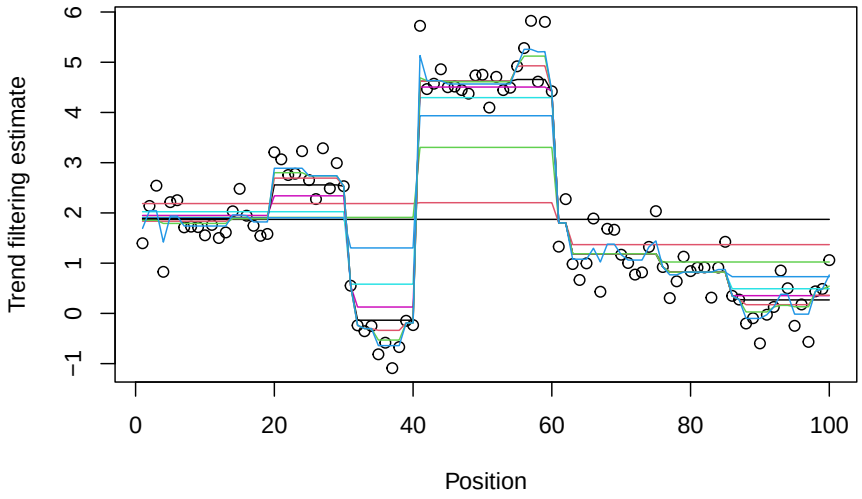
The fused lasso computation is demonstrated below.

```
# Loading the 'genlasso' package
require(genlasso)
```

```
## Warning: package 'genlasso' was built under R version 4.1.2
```

```
## Warning: package 'igraph' was built under R version 4.1.2
```

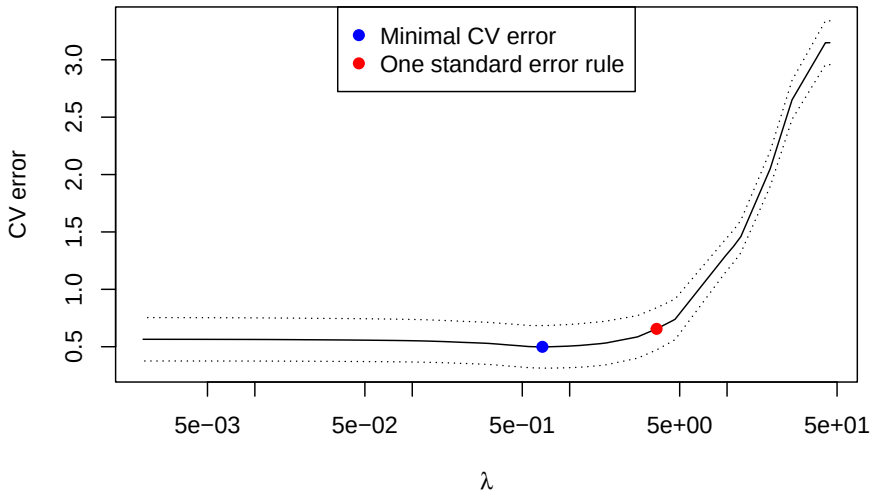
```
# Constructing the fused lasso solution paths; since
# the coefficients here are recorded on a one dimensional
# grid, we use the function 'fusedlasso1d'.
fit_fused = fusedlasso1d(Y)

# Plotting the fused lasso solution paths along with the
# actual observations; the points are the actual observations.
plot(fit_fused)
```



We note that the plot here is very different from the earlier examples, due to the difference in the nature of the problems (and also the packages used).

```
# Cross-validation for fused lasso; 'k' is  the
# cross-validation parameter corresponding to 'nfolds' in
# earlier cases. Cross-validation is done using the
# 'cv.trendfilter' function.
cv_fit_fused = cv.trendfilter(fit_fused, k = 10)
```

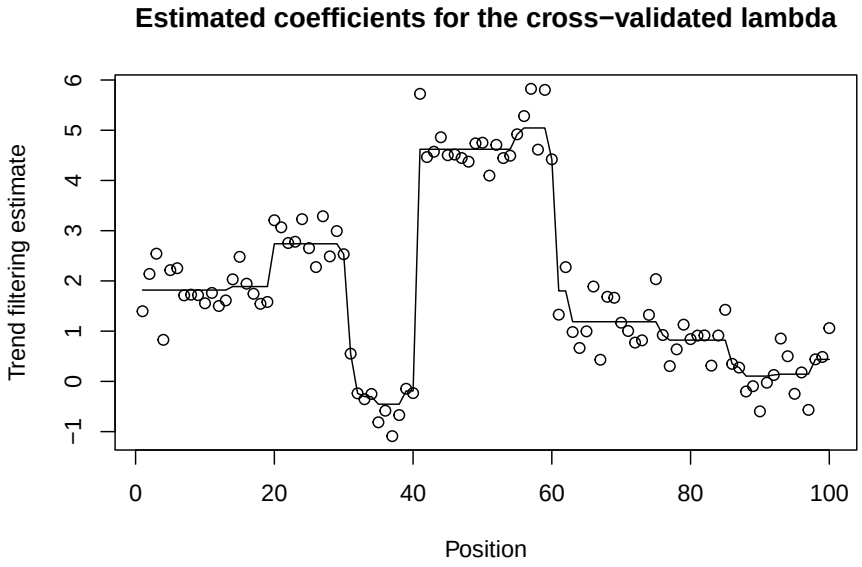
```
# Plotting the mean-squared error for different values of
# lambda in the fused lasso cross-validation
plot(cv_fit_fused)
```




```
# Getting the lambda value corresponding to the minimum
# cross-validated error in the fused lasso
lambda_min_fused = cv_fit_fused$lambda.min
writeLines(paste('Cross-validated lambda for fused lasso:',
                 lambda_min_fused))
```

```
## Cross-validated lambda for fused lasso: 0.67106063258858
```

```
# Plotting the fused lasso estimate for the
# cross-validated lambda along with the actual observations
plot(fit_fused, lambda = lambda_min_fused,
     main = paste('Estimated coefficients for the',
                  'cross-validated lambda'))
```


Estimated coefficients for the cross−validated lambda
Trend filtering estimate
−1
0
1
2
3
4
5
6
0
20
40
60
80
100
Position

Chapter 2

Multiple testing

In many situations, we face with the problem of conducting a collection of statistical tests simultaneously based on the same experiment. Consider the following problems:

1. A researcher is investigating which genes make one individual susceptible to a particular disease, say, a cancer. To investigate that, data on gene expression levels are collected over 10000 genes from 100 normal individuals and 100 patients. The culprit genes making one susceptible to the particular disease can be discovered by finding the genes whose expression level differ significantly between a normal individual and a patient. Consequently, the researcher faces a problem of conducting 10000 statistical tests simultaneously based on a sample of 200. Such problems are common in genetic association studies.

2. A pharmaceutical company considers a new drug for the treatment of a particular disease, and proceeds to investigate its efficacy. Tearment of a disease often consists of alleviating of its symptoms, which can be many. So, the company is faced with conducting a randomized controlled trial on patients, administering on group placebo and the other the experimental drug, and then conducting statistical tests to identify significant differences between the experimental group undergoing the treatment and the control group for each of the plethora of symptoms.

In genetic studies as described in the first example, subseaquent investigation is carried out for each of the genes discovered to have a significant effect. Often, the actual number of genes associated with a disease does not exceed a few dozens. If that is the case here, and still one carries out usual hypothesis testing at 5% level for each of the 10000 genes, it is expected that the researcher will get nearly $(10000 \times 0.05) = 500$ *false positives*, which are genes that are incorrectly identified as having a significant association with the disease. This is far higher than the number of actual genes influencing the susceptibility to the disease,

which means significant resources lost in further investigation in all these false positive genes.

In the second example, suppose the new drug is no better than the placebo or no intervention. However, if there are 20-30 symptoms and usual statistical tests are carried out for each of these at 5% level, it is very probable that the new drug is found to have a significant effect for at least one syndrom.

The above examples demonstrate the limitation of usual methods of hypothesis testing while considering a family of hypotheses simultaneously for an experiment. We state the problem formally below.

Suppose we want to test m hypotheses $H_1, \ldots, H_m$ simultaneously for an experiment, and m_0 of these are actually true. Let R be the number of hypotheses rejected. The following table describes the different variables on which we shall concentrate further.

Table 2.1: Classification of outcomes in multiple testing

	Null hypothesis is true	Null hypothesis is false	Total
Fails to reject null hypothesis	U	W	$m-R$
Rejects null hypothesis	V	S	R
Total	m_0	$m-m_0$	m

Of the elements in the above table, R is the only observable variable. R is the total number of *discoveries*. S is the number of *true positives* (also called true discoveries), while V is the number of *false positives* (also called false discoveries). In situations like the one described in the second example, one would like to bound the probability of $V \geq 1$ (because, for example, rejection of even one hypothesis means accepting the new drug as better than placebo). The quantity $P[V \geq 1]$ is called the *familywise error rate* (**FWER**).

2.1 Controlling the familywise error rate

Since $P[V \geq 1]$ is the probability of rejecting at least one true null hypothesis, by ensuring $P[V \geq 1] \leq \alpha$, we bound the maximum familywise error rate by α. Below, several methods to achieve this are described.

2.1.1 Bonferroni correction

Suppose the level of each of the m hypotheses $H_1, \ldots, H_m$ is β. From the Boole's inequality, we have $P[V \geq 1] = P\left[\cup_{i=1}^{m}\{H_i \text{ is rejected while being true}\}\right] \leq \sum_{i=1}^{m} P[H_i \text{ is rejected while being true}] = m\beta$. Therefore, to ensure FWER $\leq \alpha$, we take the level of each of the individual hypotheses as α/m.

The Bonferroni correction is very conservative in nature (has a high type II error), as the level of each of the individual tests is very low for even moderate p. Also, when the test statistics for the hypotheses are not independent, which is often the case for high dimensional setup, the actual FWER corresponding to the Bonferroni correction is considerably lower than α. This can be understood from the fact that $P[A \cup B] = P[A] + P[B] - P[A \cap B]$, which implies that for A, B being dependent events, $P[A \cup B] < P[A] + P[B]$.

Several methods were developed subsequently to mitigate the conservative nature of the Bonferroni correction. These are based on the principle of rejecting the hypotheses with the lowest p-values only, and accepting the rest.

2.1.2 Holm's method

The Holm's method, developed by Holm (1979), first orders the p-values $P_1, \ldots, P_m$ corresponding to the m hypotheses $H_1, \ldots, H_m$ in the increasing order. Let the ordered p-values be $P_{(1)}, \ldots, P_{(m)}$, and the corresponding hypotheses be $H_{(1)}, \ldots, H_{(m)}$. For a given level α, let k be the smallest integer such that

$$P_{(k)} > \frac{\alpha}{m - k + 1}.$$

The Holm's method rejects the hypotheses $H_{(1)}, \ldots, H_{(k-1)}$, and does not reject $H_{(k)}, \ldots, H_{(m)}$. If $k = 1$, it does not reject any null hypotheses, and if no such k exists, it rejects all the null hypotheses.

It is easy to see that, whenever the Bonferroni correction rejects a particular hypothesis, the Holm's method also rejects it, but not the contrary. It can be proved that FWER $\leq \alpha$ for the Holm's method. Hence, the Holm's method is uniformly more powerful than the Bonferroni correction.

2.1.3 Hochberg's method

Hochberg's method (Hochberg, 1988) also orders the p-values in the increasing order as in 2.1.2. However, it finds the largest integer k such that

$$P_{(k)} \leq \frac{\alpha}{m - k + 1},$$

and then rejects $H_{(1)}, \ldots, H_{(k)}$ but does not reject $H_{(k+1)}, \ldots, H_{(m)}$.

At the first glance, the Holm's method and the Hochberg's method may seem identical, but in fact Hochberg's method is uniformly more powerful than the former. This is due to the fact that the Hochberg's method rejects $H_{(1)}, \ldots, H_{(k)}$ whenever the Holm's method rejects them, but not the other way. Intuitively, because the Hochberg's method considers the p-values from the highest to the lowest, it is always inclined to reject more hypotheses than the Holm's method. However, unlike the Holm's method, whose bound on FWER does not need any assumption on the underlying distribution or dependence among the p-values or test statistics, the Hochberg's method does (Sarkar and Chang (1997), Sarkar

(1998)). Its bound on FWER is valid in particular when the test statistics for the individual hupotheses are indepdendent or positively correlated.

2.1.4 Hommel's method

In the Hommel's method (Hommel, 1988), one again orders the p-values as in 2.1.2, then finds the maximum integer j in $1, \dots, n$ such that

$$P_{(n-j+k)} > \frac{k\alpha}{j}$$

for all $k = 1, \dots, j$. If such a maximum j does not exist, one rejects all the null hypotheses H_i, $i = 1, \dots, m$. Otherwise, one rejects the hypotheses whose corresponding p-values satisfy $P_i \leq \alpha/j$.

It can be shown that the Hommel's method is more powerful than the Hochberg's method. However, the validity of the bound on its FWER also requires certain assumptions.

2.2 False discovery rate and its control

In certain situations, it may not be appropriate to put a bound on the FWER. Consider the example of the gene expression experiment described in the beginning of Chapter 2. If there are, say, 30 genes with actual effect on the disease, and in the testing 2 false positives occur, then it is not a very significant problem in terms of drawing wrong conclusions and wastage of resources. Similarly, if there are 200 genes affecting the disease in reality, and we have 20 false positives, then again it is not a big issue. In such cases, controlling the FWER makes the statistical investigation too conservative for the present purpose, and a certain kind of adaptablity in the procedure is desired based on the number of true positives. In such cases, we want to control the number of false positives compared to the true positives, as opposed to the occurance of any false positives.

Consider the quantities described in Table 2.1. Recall that the variable R denotes the number of rejected null hypotheses (discoveries) and V is the number of rejected true hypotheses (false discoveries). We want to put a bound on the expected value of the ratio (V/R). The quantity (V/R) is called the *false discovery proportion*, and $E[V/R]$ is called the *false discovery rate* (**FDR**). We assume the convention of defining $0/0 = 0$. Below, two methods are described which control the FDR.

2.2.1 Benjamini-Hochberg method

The Benjamini-Hochberg method (Benjamini and Hochberg, 1995) finds the largest integer k such that

$$P_{(k)} \leq \frac{\alpha}{m}k$$

for a given level α. It then rejects all $H_{(i)}$ for $i = 1, \ldots, k$.

This method can be represented by a plot of the ordered p-values along with a line passing through the origin having slope (α/m). The hypotheses, whose corresponding p-values are below the line, are rejected.

It can be shown that this method has a FDR less than or equal to when the test statistics are independent, and under certain depdendence conditions. However, this is not valid for arbitrary dependence structures.

2.2.2 Benjamini–Yekutieli method

The Benjamini–Yekutieli method (Benjamini and Yekutieli, 2001) modifies the Benjamini-Hochberg method so that it controls the FDR under general dependene structures. This method finds the largest k such that

$$P_{(k)} \leq \frac{\alpha}{mh(m)}k,$$

where $h(m) = \sum_{i=1}^{m} \frac{1}{i}$, and then rejects all $H_{(i)}$ for $i = 1, \ldots, k$. However, if the test statistics are independent or the dependence conditions in the Benjamini-Hochberg method are satisfied, one can take $h(m) = 1$.

The FDR of the Benjamini–Yekutieli method is bounded above by α, as indicated earlier.

2.3 Adjusted p-values

In hypothesis testing, we are often more interested in the p-value of the test instead of accepting or rejecting the null hypothesis at some specified level. Reporting the p-value only makes our inference independent of the level of the test. In multiple testing, we have seen that the methods are designed to reject or not reject the individual hypotheses based on their p-values and a pre-specified level. Nevertheless, for each of the methods described above, one can modify the p-values obtained for the individual hypothesis so that one can follow the usual procedure of hypothesis testing based on those modified p-values and any level and still control the FWER or FDR as intended. These modified p-values are called *adjusted p-values*. The adjusted p-values are numbers lying in the interval [0, 1], but they should not be interpreted as some probabilities like actual p-values.

- In the Bonferroni correction (2.1.1), one rejects hypothesis H_i at level α if the corresponding p-value P_i satisfies $P_i \leq \alpha/m$. Therefore the adjusted p-value $\widehat{P}_i$ for hypothesis H_i for the Bonferroni correction is $\widehat{P}_i = \min\{mP_i, 1\}$, $i = 1, \ldots, m$. The minimum with 1 is taken to bound the p-value in the interval $[0, 1]$.

- The adjusted p-values for the Holm's method (2.1.2) are given by $\hat{P}_{(i)} = \min\left\{\max\{(m+1-j)P_{(j)} \,|\, j = 1, \ldots, i\}, 1\right\},\ i = 1, \ldots, m.$ (Multiplication by $(m+1-j)$ is obvious. Why the cumulative maximum is taken? Convince yourself that it is needed.)
- The adjusted p-values for the Hochberg's method (2.1.3) are given by $\hat{P}_{(i)} = \min\left\{\min\{(m+1-j)P_{(j)} \,|\, j = i, \ldots, m\}, 1\right\},\ i = 1, \ldots, m.$ Note that, though the difference in the descriptions of the Holm's method and the Hochberg's method may not be obvious initially, the formulae for their adjusted p-values are clearly distinct.
- Calculation of the adjusted p-values for the Hommel's method (2.1.4) is relatively complicated. These are computed algorithmically (given by Wright (1992)), unlike the straightforward formulae for the other methods.
- The adjusted p-values for the Benjamini-Hochberg method (2.2.1) are given by $\hat{P}_{(i)} = \min\left\{\min\left\{\frac{m}{j}P_{(j)} \,\middle|\, j = i, \ldots, m\right\}, 1\right\},\ i = 1, \ldots, m.$
- The adjusted p-values for the Benjamini–Yekutieli method (2.2.2) are given by $\hat{P}_{(i)} = \min\left\{\min\left\{\frac{mh(m)}{j}P_{(j)} \,\middle|\, j = i, \ldots, m\right\}, 1\right\},\ i = 1, \ldots, m,$ where $h(m) = \sum_{j=1}^{m} \frac{1}{j}$.

2.4 Summary

While testing simultaneously many hypotheses, one may be interested in controlling the probability of rejecting atleast one true null hypothesis (FWER) or the expected proportion of rejected true null hypotheses (FDR). There are several methods to achieve either one. The Bonferroni correction, the Holm's method, the Hochberg's method and the Hommel's method control the FWER. The Benjamini-Hochberg method and the Benjamini–Yekutieli method control the FDR.

All the methods described in this chapter make very broad assumptions about the dependence among the test statistics for the individual tests (or does not make any assumption about dependence; for example: the Bonferroni correction). As a consequence, these are relatively conservative procedures, and it is possible to develop methods which are more powerful than them by taking into account the dependence structure of the test statistics in the particular multiple testing scenario. Resampling based methods (bootstrap and permutation) are develop with this aim, and they generally are more powerful than the methods described before. Those are out of scope of the present document.

2.5 Numerical demonstration

All the methods for adjusted p-value computation described in 2.3 are implemented in the function `p.adjust` available in the R package **stats** (included

with base R distribution). Below, the use of this function is demonstrated.

Consider a simple case where we have 10 hypotheses each having a p-value 0.04. What will be the adjusted p-values for the various methods described in sections 2.1 and 2.2? Their computation is presented below.

```
# Vector of p-values with length 10 and each element being 0.04
p = rep(0.04, 10)
writeLines(paste('Actual p-values:\n', paste(p, collapse = ' ')))

# Bonferroni correction
p_bonferroni = p.adjust(p, method = 'bonferroni')
writeLines(paste('Adjusted p-values for the',
                 'Bonferroni correction:\n',
                 paste(p_bonferroni, collapse = ' ')))

# Holm's method
p_holm = p.adjust(p, method = 'holm')
writeLines(paste('Adjusted p-values for the',
                 'Holm\'s method:\n',
                 paste(p_holm, collapse = ' ')))

# Hochberg's method
p_hochberg = p.adjust(p, method = 'hochberg')
writeLines(paste('Adjusted p-values for the',
                 'Hochberg\'s method:\n',
                 paste(p_hochberg, collapse = ' ')))

# Hommel's method
p_hommel = p.adjust(p, method = 'hommel')
writeLines(paste('Adjusted p-values for the',
                 'Hommel\'s method:\n',
                 paste(p_hommel, collapse = ' ')))

# Benjamini-Hochberg method
p_BH = p.adjust(p, method = 'BH')
writeLines(paste('Adjusted p-values for the',
                 'Benjamini-Hochberg method:\n',
                 paste(p_BH, collapse = ' ')))

# Benjamini-Yekutieli method
p_BY = p.adjust(p, method = 'BY')
writeLines(paste('Adjusted p-values for the',
                 'Benjamini-Yekutieli method:\n',
                 paste(round(p_BY, 3), collapse = ' ')))
```

```
## Actual p-values:
```

```
##  0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04
## Adjusted p-values for the Bonferroni correction:
##  0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4
## Adjusted p-values for the Holm's method:
##  0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4 0.4
## Adjusted p-values for the Hochberg's method:
##  0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04
## Adjusted p-values for the Hommel's method:
##  0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04
## Adjusted p-values for the Benjamini-Hochberg method:
##  0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04 0.04
## Adjusted p-values for the Benjamini-Yekutieli method:
##  0.117 0.117 0.117 0.117 0.117 0.117 0.117 0.117 0.117 0.117
```

From the output above, the difference between the adjusted p-values of the Holm's method and the Hochberg's method is noticeable. It corroborates our earlier assertion in subsection 2.1.3 that the Hochberg's method is uniformly more powerful than the Holm's method, as every adjusted p-value for the Hochberg's method is smaller than the corresponding adjusted p-value for the Holm's method. However, though the Holm's method is uniformly more powerfull than the Bonferroni correction and the Hommel method is more powerful than the Hochberg's method, this toy example cannot confirm them, as all the corresponding adjusted p-values are identical. If we conduct simultaneous testing at level 0.05, the Bonferroni correction, the Holm's method and the Benjamini–Yekutieli method fail to reject any hothesis, whereas the Hochberg's method, the Hommel's method and the Benjamini-Hochberg method reject all the hypotheses.

Next, we consider a real dataset: the `golub` dataset from the R package **multtest** available in Bioconductor. This dataset contains the expression levels of 3051 genes based on 38 tumor mRNA samples from a leukemia microarray study presented in Golub et al. (1999). Among the 38 tumor samples, 27 are acute lymphoblastic leukemia (ALL) cases and 11 are acute myeloid leukemia (AML) cases. The `golub` datatset is a matrix with 3051 rows and 38 columns. Each row corresponds to a gene, the first 27 columns correspond to ALL cases and the last 11 columns correspond to AML cases. We are interested in investigating the presence of statistically significant differences in expression levels for the 3051 genes among the two types of cancers.

First we need to install the 'multtest' package from Bioconductor, which can be done using the function 'install' from the package 'BiocManager'. The code in the next line would do this.

```
BiocManager::install('multtest')
```

Now, the computation procedure is demonstrated below.

```
# Loading the data from the package 'multtest'
data('golub', package = 'multtest')

# Storing the data corresponding to ALL cases in the
# variable golub_ALL
golub_ALL = golub[,1:27]

# Storing the data corresponding to AML cases in the
# variable golub_AML
golub_AML = golub[,28:38]

# m is the number of tests, which is equal to the
# number of genes
m = nrow(golub)

# Computing the p-values using a two-sample t test
p = c()
for (i in 1:m)
  p[i] = t.test(golub_ALL[i,], golub_AML[i,],
                alternative = 'two.sided')$p.value

# Calculating the adjusted p-values
p_bonferroni = p.adjust(p, method = 'bonferroni')
p_holm = p.adjust(p, method = 'holm')
p_hochberg = p.adjust(p, method = 'hochberg')
p_hommel = p.adjust(p, method = 'hommel')
p_BH = p.adjust(p, method = 'BH')
p_BY = p.adjust(p, method = 'BY')

# Calculating the number of rejections based on level 0.05.
alpha = 0.05
writeLines(paste('Number of rejections based on',
                 'unadjusted p-values:', sum(p <= alpha)))
writeLines(paste('Number of rejections based on',
                 'Bonferroni correction:',
                 sum(p_bonferroni <= alpha)))
writeLines(paste('Number of rejections based on',
                 'Holm\'s method:',
                 sum(p_holm <= alpha)))
writeLines(paste('Number of rejections based on',
                 'Hochberg\'s method:',
                 sum(p_hochberg <= alpha)))
writeLines(paste('Number of rejections based on',
                 'Hommel\'s method:',
                 sum(p_hommel <= alpha)))
```

```
writeLines(paste('Number of rejections based on',
                 'Benjamini-Hochberg method:',
                 sum(p_BH <= alpha)))
writeLines(paste('Number of rejections based on',
                 'Benjamini-Yekutieli method:',
                 sum(p_BY <= alpha)))
```

```
## Number of rejections based on unadjusted p-values: 1078
## Number of rejections based on Bonferroni correction: 103
## Number of rejections based on Holm's method: 103
## Number of rejections based on Hochberg's method: 103
## Number of rejections based on Hommel's method: 108
## Number of rejections based on Benjamini-Hochberg method: 695
## Number of rejections based on Benjamini-Yekutieli method: 293
```

We note that the Bonferroni's correction is the most conservative procedure, as expected. However, here the Holm's method and the Hochberg's method are also as much conservative as the Bonferroni's correction. The FDR controlling procedures are relatively less conservative compared to the FWER controlling procedures, again as expected.

Chapter 3

Assignments

1. Fix your roll number as the seed. Let Σ be a 80×80 covariance matrix defined by $\sigma_{ij} = 0.6 + 0.6\mathbb{I}(i = j)$. Let X be a 80 dimensional random vector with $X \sim N_{80}(0, \Sigma)$. Denote the ith coordinate of X as X_i, $i = 1, \ldots, 80$. Consider the following regression problem: $Y = (1 + X_1 + 2X_2 - 3X_3 - 4X_4 + 5X_5) + e$, where e is an error variable independent of X. Generate a sample of size 30 on Y and X. Generate also a sample of size 5 from the distribution of X, and name the data matrix as X_0. Based on this sample,

   - Compute and plot the lasso solution paths.
   - Compute and visually present the outputs of the cross-validation process for the tuning parameter λ.
   - Print the value of λ giving the minimum cross-validated error.
   - Print the value of λ giving the most regularized model such that its cross-validated error is within one standard error of the minimum.
   - Print the estimated coefficients of the fitted lasso model for the value λ being the one giving the minimum cross-validated error.
   - Print the estimated coefficients of the fitted lasso model for $\lambda = 0.5$.
   - Print the number of nonzero coefficients for the value λ being the one giving the minimum cross-validated error.
   - Predict the response values at the covariate values in X_0 for λ being the minimizer of the cross-validated error.

2. Based on the same data as above, solve all the above problems for the ridge regression method.

3. Again, based on the same data as above, solve all the above problems for the elastic net method, taking $\alpha = 0.5$.

4. Generate an independent sample of size 1000 from the $Beta(1, 100)$ distribution, and assume that they are the p-values of 1000 hypotheses tested simultaneously.

    - Compute the corresponding adjusted p-values for the Bonferroni correction, the Holm's method, the Hochberg's method, the Hommel's method, the Benjamini-Hochberg method and the Benjamini–Yekutieli method.
    - Compute the number of rejections in each case for the level being 0.05.

Marks: (1): 10, (2): 3, (3): 3, (4): 4

Bibliography

Arnold, T. B. and Tibshirani, R. J. (2020). *genlasso: Path Algorithm for Generalized Lasso Problems*. R package version 1.5.

Benjamini, Y. and Hochberg, Y. (1995). Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B (Methodological)*, 57(1):289–300.

Benjamini, Y. and Yekutieli, D. (2001). The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, pages 1165–1188.

Bühlmann, P. and Meier, L. (2008). Discussion: One-step sparse estimates in nonconcave penalized likelihood models. *Annals of Statistics*, 36(4):1534–1541.

Bühlmann, P. and Van De Geer, S. (2011). *Statistics for high-dimensional data: methods, theory and applications*. Springer Science & Business Media.

Friedman, J., Hastie, T., Tibshirani, R., Narasimhan, B., Tay, K., Simon, N., and Yang, J. (2021). *glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models*. R package version 4.1-3.

Golub, T. R., Slonim, D. K., Tamayo, P., Huard, C., Gaasenbeek, M., Mesirov, J. P., Coller, H., Loh, M. L., Downing, J. R., Caligiuri, M. A., Bloomfield, C. D., and Lander, E. S. (1999). Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science*, 286(5439):531–537.

Hochberg, Y. (1988). A sharper bonferroni procedure for multiple tests of significance. *Biometrika*, 75(4):800–802.

Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70.

Hommel, G. (1988). A stagewise rejective multiple test procedure based on a modified bonferroni test. *Biometrika*, 75(2):383–386.

Sarkar, S. K. (1998). Some probability inequalities for ordered mtp2 random variables: a proof of the simes conjecture. *Annals of Statistics*, pages 494–504.

Sarkar, S. K. and Chang, C.-K. (1997). The simes method for multiple hypothesis testing with positively dependent test statistics. *Journal of the American Statistical Association*, 92(440):1601–1608.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 58(1):267–288.

Tibshirani, R. (2011). Regression shrinkage and selection via the lasso: a retrospective. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(3):273–282.

Tibshirani, R., Saunders, M., Rosset, S., Zhu, J., and Knight, K. (2005). Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(1):91–108.

Tikhonov, A. N. (1943). On the stability of inverse problems. In *Doklady Akademii Nauk SSSR*, volume 39, pages 195–198.

Wright, S. P. (1992). Adjusted p-values for simultaneous inference. *Biometrics*, pages 1005–1013.

Yang, Y., Zou, H., and Bhatnagar, S. (2020). *gglasso: Group Lasso Penalized Learning Using a Unified BMD Algorithm*. R package version 1.5.

Yuan, M. and Lin, Y. (2006). Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(1):49–67.

Zou, H. (2006). The adaptive lasso and its oracle properties. *Journal of the American Statistical Association*, 101(476):1418–1429.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.