Code

Linear Model Selection

Ruoqing Zhu

Last Updated: August 20, 2024

Model Selection Criteria

We will use the `diabetes` dataset from the `lars` package as a demonstration of model selection. Ten baseline variables include age, sex, body mass index, average blood pressure, and six blood serum measurements. These measurements were obtained for each of $n = 442$ diabetes patients, as well as the outcome of interest, a quantitative measure of disease progression one year after baseline. More details are available in the `lars` package documentation (https://cran.r-project.org/web/packages/lars/lars.pdf). Our goal is to select a linear model, preferably with a small number of variables, that can predict the outcome. To select the best model, commonly used strategies include Marrow’s C_p, AIC (Akaike information criterion) and BIC (Bayesian information criterion).

Hide

```
    # load the diabetes data
    library(lars)
## Loaded lars 1.3
    data(diabetes)
    diab = data.frame(cbind(diabetes$x, "Y" = diabetes$y))

    # fit linear regression with all covariates
    lm.fit = lm(Y~., data=diab)
```

The idea of model selection is to apply some penalty on the number of parameters used in the model. On one hand, we want to model fitting to be as good as possible, i.e., the mean squared error is small. On one hand, keeping adding covariates would always (theoretically) improve the model fitting (training error) and eventually give us a perfect fit. However, is this necessarily a good thing? We will end up with too many parameters, which is hard to interpret. But also, what about the prediction error? Are they necessarily better than a simple model? Let’s investigate this with a simulation study.

A Simulation Study

The idea of simulation is to assess the performance of a model by repeatedly generating data from a known underlying model. Usually, this requires a large number of repeats so that a pattern can be revealed. In the following example, we will see how the increasing number of covariates could affect the model performance.

In **each simulation**, we will generate data and fit a linear regression. You should notice that only the first variable is useful. However, in practice, we may not know which one is truly useful. Hence, all variables will be used.

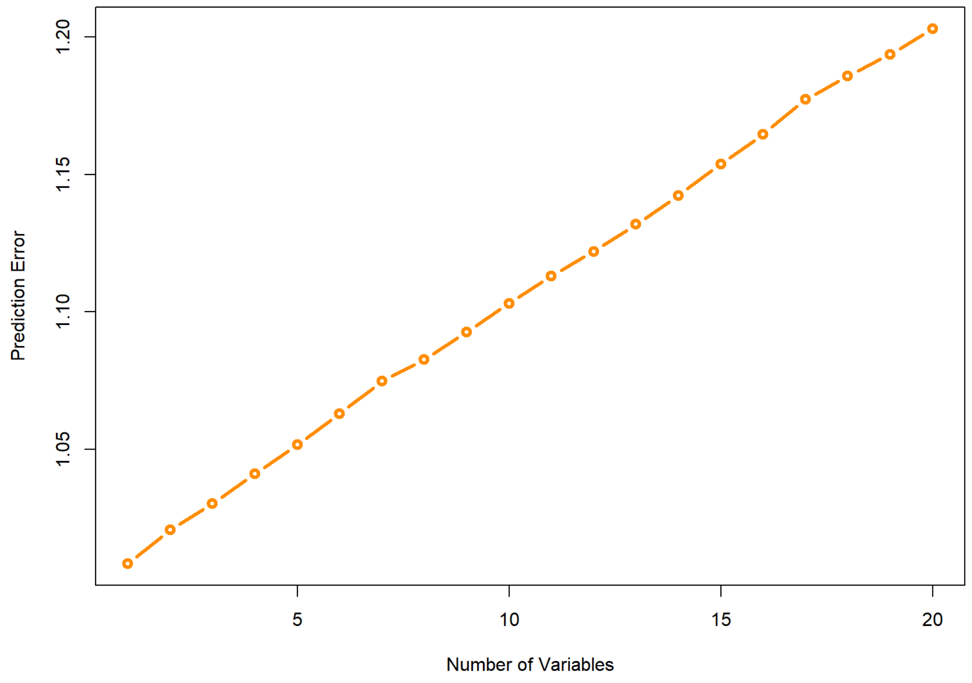
Hide

```
    set.seed(1)
    n = 100
    p = 20

    # the design matrix
    x = matrix(rnorm(n*p), n, p)

    # training data outcome
    ytrain = 0.3*x[, 1] + rnorm(n)

    # testing data outcome, using the same covariates
    ytest = 0.3*x[, 1] + rnorm(n)

    # construct the training and testing data
    traindata = data.frame("x" = x, "y" = ytrain)
    testdata = data.frame("x" = x, "y" = ytest)

    # fit model and calculate testing error
    onefit = lm(y~., data = traindata)
    ypred = predict(onefit, testdata)
    error = mean( (ypred - testdata$y)^2 )
    error
## [1] 1.273226
```

The question now is, if we use a smaller set of covariates, would the performance be better? We are going to do two things. First, we will try different model size, from 1 to 20. Second, we will repeat this process 100 times for each model size and take the averaged result. This can reasonably demonstrate the effect of using covariates.

Hide

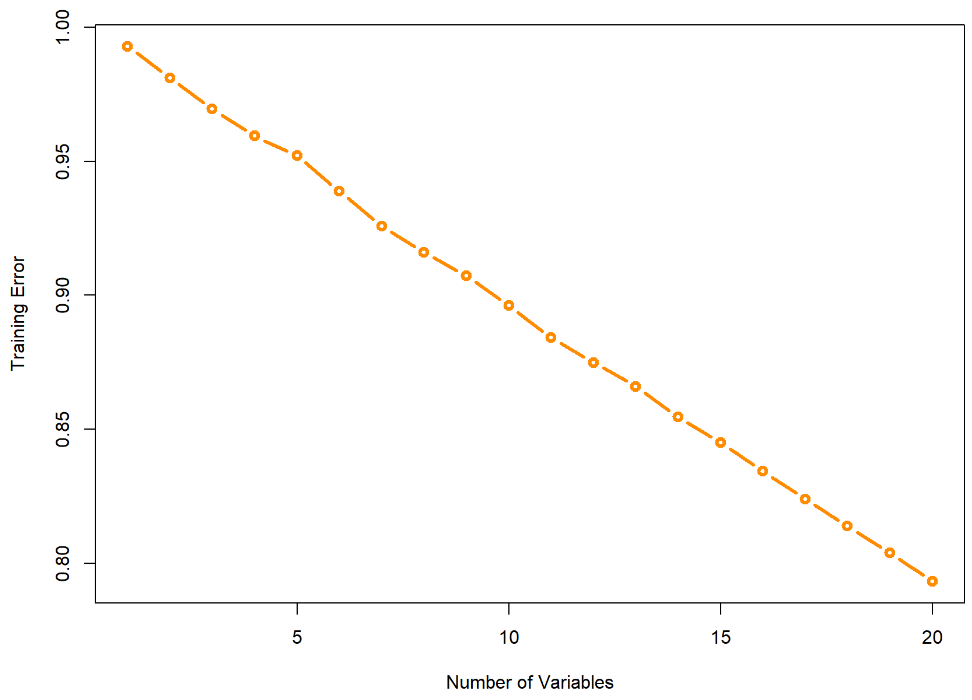
```
nsim = 100
testerrors = matrix(NA, nsim, p)
trainerrors = matrix(NA, nsim, p)

for (i in 1:nsim)
{
    # the design matrix
    x = matrix(rnorm(n*p), n, p)
    ytrain = 0.3*x[, 1] + rnorm(n)
    ytest = 0.3*x[, 1] + rnorm(n)

    # try all different dimensions
    for (j in 1:p)
    {
        # construct the data
        traindata = data.frame("x" = x[, 1:j, drop = FALSE], "y" = ytrain)
        testdata = data.frame("x" = x[, 1:j, drop = FALSE], "y" = ytest)
        # fit model and calculate testing error
        onefit = lm(y~., data = traindata)
        ypred = predict(onefit, testdata)
        error = mean( (ypred - testdata$y)^2 )

        testerrors[i, j] = error
        trainerrors[i, j] = mean( (onefit$fitted - traindata$y)^2 )
    }
}

plot(colMeans(testerrors), type = "b",
     col = "darkorange", lwd = 3,
     xlab = "Number of Variables",
     ylab = "Prediction Error")
```



Hence, it seems that when only one variable is useful, adding any other variables would only increase the prediction error. The more irrelevant variables we add, the worse the result becomes. This is over-fitting. On the other hand, the training error is still decreasing.

Hide

```
    plot(colMeans(trainerrors), type = "b",
         col = "darkorange", lwd = 3,
         xlab = "Number of Variables",
         ylab = "Training Error")
```

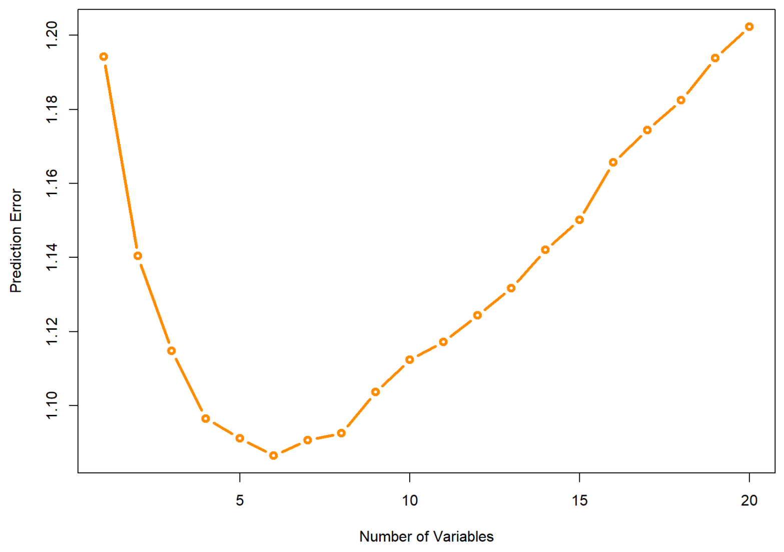


In many situations, variables are not completely useless. For example, in the following model, the effect of each variable decreases gradually:

$$\begin{aligned} Y &= \mu + \epsilon \\ &= X^{\mathrm{T}}\boldsymbol{\beta} + \epsilon \\ &= \sum_{j=1}^{p} X_j 0.4^{\sqrt{j}} + \epsilon \end{aligned}$$

Hide

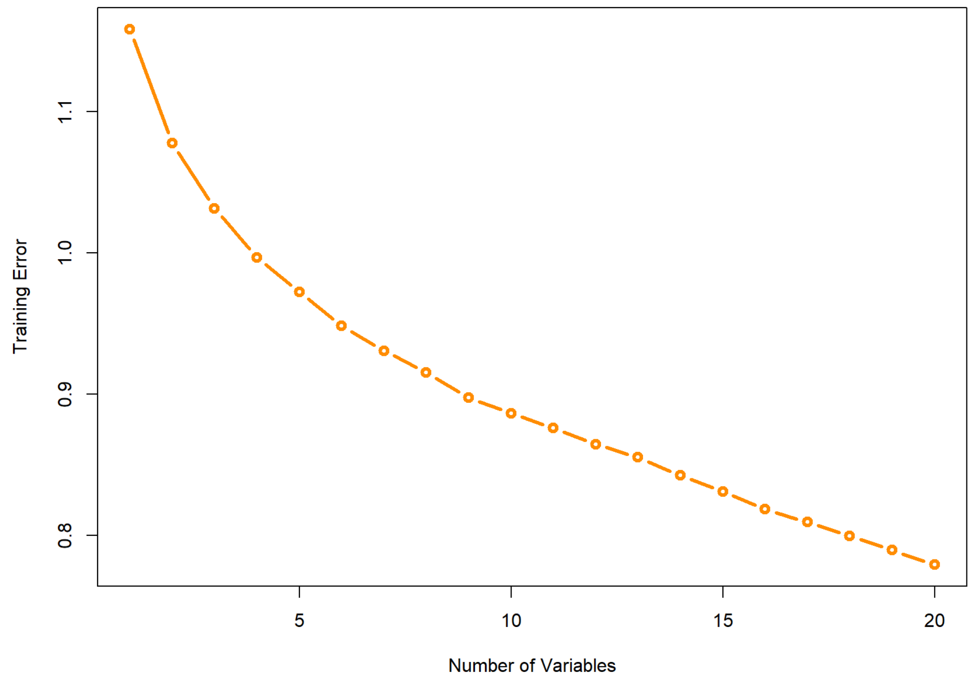
```r
testerrors = matrix(NA, nsim, p)
trainerrors = matrix(NA, nsim, p)

for (i in 1:nsim)
{
    # the design matrix
    x = matrix(rnorm(n*p), n, p)
    ytrain = x %*% 0.4^sqrt(c(1:p)) + rnorm(n)
    ytest = x %*% 0.4^sqrt(c(1:p)) + rnorm(n)

    # try all different dimensions
    for (j in 1:p)
    {
        # construct the data
        traindata = data.frame("x" = x[, 1:j, drop = FALSE], "y" = ytrain)
        testdata = data.frame("x" = x[, 1:j, drop = FALSE], "y" = ytest)

        # fit model and calculate testing error
        onefit = lm(y~., data = traindata)
        ypred = predict(onefit, testdata)
        error = mean( (ypred - testdata$y)^2 )

        testerrors[i, j] = error
        trainerrors[i, j] = mean( (onefit$fitted - traindata$y)^2 )
    }
}

plot(colMeans(testerrors), type = "b",
     col = "darkorange", lwd = 3,
     xlab = "Number of Variables",
     ylab = "Prediction Error")
```



The training error will still decrease as expected. But we can see that the decreasing trend is not linear, meaning that for the first several variables, they have a stronger effect in explaining the outcome, and later variables are close to useless.

Hide

```
    plot(colMeans(trainerrors), type = "b",
         col = "darkorange", lwd = 3,
         xlab = "Number of Variables",
         ylab = "Training Error")
```



Combining these understandings, the best model seems to be using six variables, from the testing errors. The downside here is that we are not able to explain the signals of the remaining variables. However, the upside is that (from our first example), by using less number of variables (regardless of usefulness), we can improve the prediction error. This is essentially a **bias-variance trade-off**. We will introduce several methods that select variables and obtain a better model for prediction. Their essential ideas are similar, to introduce some penalty on the number of variables.

$$\text{Goodness-of-Fit} + \text{Complexity Penality}$$

Using Marrows' C_p

For example, the Marrows' C_p criterion minimize the following quantity:

$$\text{RSS} + 2p\widehat{\sigma}^2_{\text{full}}$$

Note that the σ^2_{full} refers to the residual variance estimation based on the full model, i.e., will all variables. Hence, this formula cannot be used when $p > n$ because you would not be able to obtain a valid estimation of σ^2_{full}. Nonetheless, we can calculate this quantity with the diabetes dataset

Hide

```
    # number of variables (including intercept)
    p = 11
    n = nrow(diab)

    # obtain residual sum of squares
    RSS = sum(residuals(lm.fit)^2)

    # use the formula directly to calculate the Cp criterion
    Cp = RSS + 2*p*summary(lm.fit)$sigma^2
    Cp
## [1] 1328502
```

We can compare this with another sub-model, say, with just `age` and `glu`:

Hide

```
    lm.fit_sub = lm(Y~ age + glu, data=diab)

    # obtain residual sum of squares
    RSS_sub = sum(residuals(lm.fit_sub)^2)

    # use the formula directly to calculate the Cp criterion
    Cp_sub = RSS_sub + 2*3*summary(lm.fit)$sigma^2
    Cp_sub
## [1] 2240019
```

Comparing this with the previous one, the full model is better.

Using AIC and BIC

Calculating the AIC and BIC criteria in `R` is a lot simpler, with the existing functions. The AIC score is given by

$$-2\text{Log-likelihood} + 2p,$$

while the BIC score is given by

$$-2\text{Log-likelihood} + \log(n)p,$$

Interestingly, when assuming that the error distribution is Gaussian, the log-likelihood part is just a function of the RSS. In general, AIC performs similarly to C_p, while BIC tend to select a much smaller set due to the larger penalty. Theoretically, both AIC and C_p are interested in the prediction error, regardless of whether the model is specified correctly, while BIC is interested in selecting the true set of variables, while assuming that the true model is being considered.

The AIC score can be done using the `AIC()` function. We can match this result by writing out the normal density function and plug in the estimated parameters. Note that this requires one additional parameter, which is the variance. Hence the total number of parameters is 12. We can calculate this with our own code:

Hide

```
    # ?AIC
    # a build-in function for calculating AIC using -2log likelihood
    AIC(lm.fit)
## [1] 4795.985

    # Match the result
    n*log(RSS/n) + n + n*log(2*pi) + 2 + 2*p
## [1] 4795.985
```

Alternatively, the `extractAIC()` function can calculate both AIC and BIC. However, note that the `n + n*log(2*pi) + 2` part in the above code does not change regardless of how many parameters we use. Hence, this quantify does not affect the comparison between different models. Then we can safely remove this part and only focus on the essential ones.

Hide

```
    # ?extractAIC
    # AIC for the full model
    extractAIC(lm.fit)
## [1]   11.000 3539.643
    n*log(RSS/n) + 2*p
## [1] 3539.643

    # BIC for the full model
    extractAIC(lm.fit, k = log(n))
## [1]   11.000 3584.648
    n*log(RSS/n) + log(n)*p
## [1] 3584.648
```

Now, we can compare AIC or BIC using of two different models and select whichever one that gives a smaller value. For example the AIC of the previous sub-model is

Hide

```
    # AIC for the sub-model
    extractAIC(lm.fit_sub)
## [1]    3.000 3773.077
```

Model Selection Algorithms

In previous examples, we have to manually fit two models and calculate their respective selection criteria and compare them. This is a rather tedious process if we have many variables and a huge number of combinations to consider. To automatically compare different models and select the best one, there are two common computational approaches: best subset regression and step-wise regression. As their name suggest, the best subset selection will exhaust all possible combination of variables, while the step-wise regression would adjust the model by adding or subtracting one variable at a time to reach the best model.

Best Subset Selection with `leaps`

Since the penalty is only affected by the number of variables, we may first choose the best model with the smallest RSS for each model size, and then compare across these models by attaching the penalty terms of their corresponding sizes. The `leaps` package can be used to calculate the best model of each model size. It essentially performs an exhaustive search, however, still utilizing some tricks to skip some really bad models. Note that the `leaps` package uses the data matrix directly, instead of specifying a formula.

Hide

```
    library(leaps)

    # The package specifies the X matrix and outcome y vector
    RSSleaps = regsubsets(x = as.matrix(diab[, -11]), y = diab[, 11])
    summary(RSSleaps, matrix=T)
## Subset selection object
## 10 Variables  (and intercept)
##     Forced in Forced out
## age     FALSE      FALSE
## sex     FALSE      FALSE
## bmi     FALSE      FALSE
## map     FALSE      FALSE
## tc      FALSE      FALSE
## ldl     FALSE      FALSE
## hdl     FALSE      FALSE
## tch     FALSE      FALSE
## ltg     FALSE      FALSE
## glu     FALSE      FALSE
## 1 subsets of each size up to 8
## Selection Algorithm: exhaustive
##          age sex bmi map tc  ldl hdl tch ltg glu
## 1  ( 1 ) " " " " "*" " " " " " " " " " " " " " "
## 2  ( 1 ) " " " " "*" " " " " " " " " " " "*" " "
## 3  ( 1 ) " " " " "*" "*" " " " " " " " " "*" " "
## 4  ( 1 ) " " " " "*" "*" "*" " " " " " " "*" " "
## 5  ( 1 ) " " "*" "*" "*" " " " " "*" " " "*" " "
## 6  ( 1 ) " " "*" "*" "*" "*" "*" " " " " "*" " "
## 7  ( 1 ) " " "*" "*" "*" "*" "*" " " "*" "*" " "
## 8  ( 1 ) " " "*" "*" "*" "*" "*" " " "*" "*" "*"
```

The results is summarized in a matrix, with each row representing a model size. The `"*"` sign indicates that the variable is include in the model for the corresponding size. Hence, there should be only one of such in the first row, two in the second row, etc.

By default, the algorithm would only consider models up to size 8. This is controlled by the argument `nvmax`. If we want to consider larger model sizes, then set this to a larger number. However, be careful that this many drastically increase the computational cost.

Hide

```
    # Consider maximum of 10 variables
    RSSleaps = regsubsets(x = as.matrix(diab[, -11]), y = diab[, 11], nvmax = 10)
    summary(RSSleaps,matrix=T)
## Subset selection object
## 10 Variables  (and intercept)
##     Forced in Forced out
## age     FALSE      FALSE
## sex     FALSE      FALSE
## bmi     FALSE      FALSE
## map     FALSE      FALSE
## tc      FALSE      FALSE
## ldl     FALSE      FALSE
## hdl     FALSE      FALSE
## tch     FALSE      FALSE
## ltg     FALSE      FALSE
## glu     FALSE      FALSE
## 1 subsets of each size up to 10
## Selection Algorithm: exhaustive
##           age sex bmi map tc  ldl hdl tch ltg glu
## 1  ( 1 )  " " " " "*" " " " " " " " " " " " " " "
## 2  ( 1 )  " " " " "*" " " " " " " " " " " "*" " "
## 3  ( 1 )  " " " " "*" "*" " " " " " " " " "*" " "
## 4  ( 1 )  " " " " "*" "*" "*" " " " " " " "*" " "
## 5  ( 1 )  " " "*" "*" "*" " " " " "*" " " "*" " "
## 6  ( 1 )  " " "*" "*" "*" "*" "*" " " " " "*" " "
## 7  ( 1 )  " " "*" "*" "*" "*" "*" " " "*" "*" " "
## 8  ( 1 )  " " "*" "*" "*" "*" "*" " " "*" "*" "*"
## 9  ( 1 )  " " "*" "*" "*" "*" "*" "*" "*" "*" "*"
## 10  ( 1 ) "*" "*" "*" "*" "*" "*" "*" "*" "*" "*"

    # Obtain the matrix that indicates the variables
    sumleaps = summary(RSSleaps, matrix = T)

    # This object includes the RSS results, which is needed to calculate the scores
    sumleaps$rss
##  [1] 1719582 1416694 1362708 1331430 1287879 1271491 1267805 1264712 1264066 126
3983

    # This matrix indicates whether a variable is in the best model(s)
    sumleaps$which
##    (Intercept)   age   sex  bmi   map    tc   ldl   hdl   tch   ltg   glu
## 1         TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## 2         TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## 3         TRUE FALSE FALSE TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## 4         TRUE FALSE FALSE TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE
## 5         TRUE FALSE  TRUE TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE
## 6         TRUE FALSE  TRUE TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
## 7         TRUE FALSE  TRUE TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
## 8         TRUE FALSE  TRUE TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
## 9         TRUE FALSE  TRUE TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## 10        TRUE  TRUE  TRUE TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
    # The package automatically produces the Cp statistic
    sumleaps$cp
##  [1] 148.352561  47.072229  30.663634  21.998461   9.148045   5.560162   6.30322
1   7.248522   9.028080  11.000000
```

We can calculate different model selection criteria with the best models of each size. The model fitting result already produces the C_p and BIC results. However, please note that both quantities are modified slightly. For the C_p statistics, the quantity is divided by the estimated error variance, and also adjust for the sample size. For the BIC, the difference is a constant regardless of the model size. Hence these difference do will not affect the model selection result because the modification is the same regardless of the number of variables.

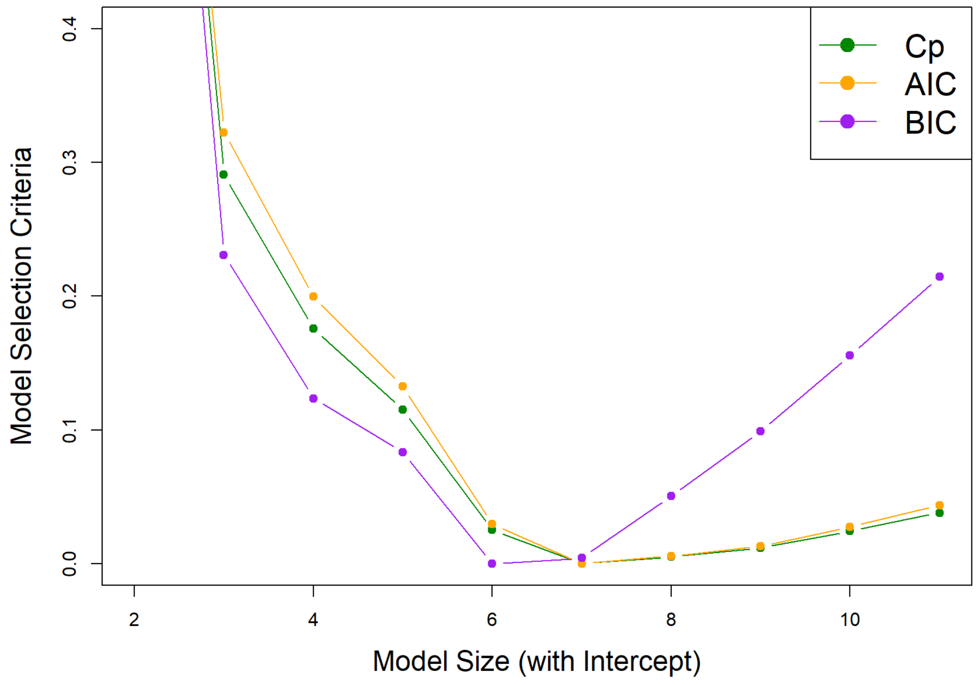
Hide

```
    modelsize=apply(sumleaps$which,1,sum)

    Cp = sumleaps$rss/(summary(lm.fit)$sigma^2) + 2*modelsize - n;
    AIC = n*log(sumleaps$rss/n) + 2*modelsize;
    BIC = n*log(sumleaps$rss/n) + modelsize*log(n);

    # Comparing the Cp scores
    cbind("Our Cp" = Cp, "leaps Cp" = sumleaps$cp)
##        Our Cp   leaps Cp
## 1  148.352561 148.352561
## 2   47.072229  47.072229
## 3   30.663634  30.663634
## 4   21.998461  21.998461
## 5    9.148045   9.148045
## 6    5.560162   5.560162
## 7    6.303221   6.303221
## 8    7.248522   7.248522
## 9    9.028080   9.028080
## 10  11.000000  11.000000

    # Comparing the BIC results. The difference is a constant,
    # which is the score of an intercept model
    cbind("Our BIC" = BIC, "leaps BIC" = sumleaps$bic,
          "Difference" = BIC-sumleaps$bic,
          "Intercept Score" = n*log(sum((diab[,11] - mean(diab[,11]))^2/n)))
##     Our BIC leaps BIC Difference Intercept Score
## 1  3665.879 -174.1108    3839.99         3839.99
## 2  3586.331 -253.6592    3839.99         3839.99
## 3  3575.249 -264.7407    3839.99         3839.99
## 4  3571.077 -268.9126    3839.99         3839.99
## 5  3562.469 -277.5210    3839.99         3839.99
## 6  3562.900 -277.0899    3839.99         3839.99
## 7  3567.708 -272.2819    3839.99         3839.99
## 8  3572.720 -267.2702    3839.99         3839.99
## 9  3578.585 -261.4049    3839.99         3839.99
## 10 3584.648 -255.3424    3839.99         3839.99
```

Finally, we may select the best model, using any of the criteria. The following code would produced a plot to visualize it. We can see that BIC selects 6 variables, while both AIC and C_p selects 7.

Hide

```
    # Rescale Cp, AIC and BIC to (0,1).
    inrange <- function(x) { (x - min(x)) / (max(x) - min(x)) }

    Cp = inrange(Cp)
    BIC = inrange(BIC)
    AIC = inrange(AIC)

    plot(range(modelsize), c(0, 0.4), type="n",
         xlab="Model Size (with Intercept)",
         ylab="Model Selection Criteria", cex.lab = 1.5)

    points(modelsize, Cp, col = "green4", type = "b", pch = 19)
    points(modelsize, AIC, col = "orange", type = "b", pch = 19)
    points(modelsize, BIC, col = "purple", type = "b", pch = 19)
    legend("topright", legend=c("Cp", "AIC", "BIC"),
           col=c("green4", "orange", "purple"),
           lty = rep(1, 3), pch = 19, cex = 1.7)
```



Step-wise regression using `step()`

The idea of step-wise regression is very simple: we start with a certain model (e.g. the intercept or the full mode), and add or subtract one variable at a time by making the best decision to improve the model selection score. The `step()` function implements this procedure. The following example starts with the full model and uses AIC as the selection criteria (default of the function). After removing several variables, the model ends up with six predictors.

Hide

```
    # k = 2 (AIC) is default;
    step(lm.fit, direction="both", k = 2)
## Start:  AIC=3539.64
## Y ~ age + sex + bmi + map + tc + ldl + hdl + tch + ltg + glu
##
##          Df Sum of Sq     RSS    AIC
## - age     1        82 1264066 3537.7
## - hdl     1       663 1264646 3537.9
## - glu     1      3080 1267064 3538.7
## - tch     1      3526 1267509 3538.9
## <none>                1263983 3539.6
## - ldl     1      5799 1269782 3539.7
## - tc      1     10600 1274583 3541.3
## - sex     1     45000 1308983 3553.1
## - ltg     1     56015 1319998 3556.8
## - map     1     72103 1336086 3562.2
## - bmi     1    179028 1443011 3596.2
##
## Step:  AIC=3537.67
## Y ~ sex + bmi + map + tc + ldl + hdl + tch + ltg + glu
##
##          Df Sum of Sq     RSS    AIC
## - hdl     1       646 1264712 3535.9
## - glu     1      3001 1267067 3536.7
## - tch     1      3543 1267608 3536.9
## <none>                1264066 3537.7
## - ldl     1      5751 1269817 3537.7
## - tc      1     10569 1274635 3539.4
## + age     1        82 1263983 3539.6
## - sex     1     45831 1309896 3551.4
## - ltg     1     55963 1320029 3554.8
## - map     1     73850 1337915 3560.8
## - bmi     1    179079 1443144 3594.2
##
## Step:  AIC=3535.9
## Y ~ sex + bmi + map + tc + ldl + tch + ltg + glu
##
##          Df Sum of Sq     RSS    AIC
## - glu     1      3093 1267805 3535.0
## - tch     1      3247 1267959 3535.0
## <none>                1264712 3535.9
## - ldl     1      7505 1272217 3536.5
## + hdl     1       646 1264066 3537.7
## + age     1        66 1264646 3537.9
## - tc      1     26840 1291552 3543.2
## - sex     1     46382 1311094 3549.8
## - map     1     73536 1338248 3558.9
## - ltg     1     97509 1362221 3566.7
## - bmi     1    178537 1443249 3592.3
##
## Step:  AIC=3534.98
## Y ~ sex + bmi + map + tc + ldl + tch + ltg
```

```
##
##        Df Sum of Sq     RSS    AIC
## - tch   1      3686 1271491 3534.3
## <none>              1267805 3535.0
## - ldl   1      7472 1275277 3535.6
## + glu   1      3093 1264712 3535.9
## + hdl   1       738 1267067 3536.7
## + age   1         0 1267805 3537.0
## - tc    1     26378 1294183 3542.1
## - sex   1     44686 1312491 3548.3
## - map   1     82154 1349959 3560.7
## - ltg   1    102520 1370325 3567.3
## - bmi   1    189970 1457775 3594.7
##
## Step:  AIC=3534.26
## Y ~ sex + bmi + map + tc + ldl + ltg
##
##        Df Sum of Sq     RSS    AIC
## <none>              1271491 3534.3
## + tch   1      3686 1267805 3535.0
## + glu   1      3532 1267959 3535.0
## + hdl   1       395 1271097 3536.1
## + age   1        11 1271480 3536.3
## - ldl   1     39378 1310869 3545.7
## - sex   1     41858 1313349 3546.6
## - tc    1     65237 1336728 3554.4
## - map   1     79627 1351119 3559.1
## - bmi   1    190586 1462077 3594.0
## - ltg   1    294094 1565585 3624.2
##
## Call:
## lm(formula = Y ~ sex + bmi + map + tc + ldl + ltg, data = diab)
##
## Coefficients:
## (Intercept)          sex          bmi          map           tc          ldl
ltg
##       152.1       -226.5        529.9        327.2       -757.9        538.6
804.2
```

We can also use different settings, such as which model to start with, which is the minimum/maximum model, and do we allow to adding/subtracting.

Hide

```
    # use BIC (k = log(n))instead of AIC
    # trace = 0 will suppress the output of intermediate steps
    step(lm.fit, direction="both", k = log(n), trace=0)
##
## Call:
## lm(formula = Y ~ sex + bmi + map + tc + ldl + ltg, data = diab)
##
## Coefficients:
## (Intercept)          sex          bmi          map           tc          ldl
ltg
##       152.1       -226.5        529.9        327.2       -757.9        538.6
804.2

    # Start with an intercept model, and use forward selection (adding only)
    step(lm(Y~1, data=diab), scope=list(upper=lm.fit, lower=~1),
         direction="forward", trace=0)
##
## Call:
## lm(formula = Y ~ bmi + ltg + map + tc + sex + ldl, data = diab)
##
## Coefficients:
## (Intercept)          bmi          ltg          map           tc          sex
ldl
##       152.1        529.9        804.2        327.2       -757.9       -226.5
538.6
```

We can see that these results are slightly different from the best subset selection. So which is better? Of course the best subset selection is better because it considers all possible candidates, which step-wise regression may stuck at a sub-optimal model, while adding and subtracting any variable do not benefit further. Hence, the results of step-wise regression may be unstable. On the other hand, best subset selection not really feasible for high-dimensional problems because of the computational cost.