Handout 3

Stefan Zohren

15 February 2016

Model selection

Let us recall the multivariate linear regression model with p predictors $\mathbf{x}_1, ..., \mathbf{x}_p$.

$$y_i = f(x_i) + \epsilon_i = a_0 + a_1 x_{i1} + ... + a_p x_{ip} + \epsilon_i$$

We are focusing our discussion here on linear regression (lecture 1), but it can be straightforwardly applied to classification using logistic regression as well (lecture 2).

One problem we are facing when analysing such a model is which predictors to include and which to possibly leave out. There might be some unrelated predictors in the data set which will not be informative for predicting the response variable and which can potentially hurt us in increasing the chance of overfitting.

In the first session we looked at a concrete example of the California Housing data. Let us use the same data set here. We first import it:

```
HousingData <- read.csv("California-Housing.txt")
```

and then split it into a training and test set:

```
set.seed(10)
n <- nrow(HousingData)
trainIndices <- sample(1:n, n*2/3)
HousingData.train <- HousingData[trainIndices,]
HousingData.test <- HousingData[-trainIndices,]
```

Best subset selection

In the first handout you were asked in one exercise to try out different combinations of predictors and for each of them evaluate the test error and thus see which variables you should include in your model. Model selection using **best subset selection** as discussed in this first part of the handout is really nothing else, but only doing it in an automated manner. In particular the function `regsubsets` of the package `leaps` allows us to call one regression function with a given maximum number of predictors `nvmax` (by default it chooses `nvmax=8`) and have the function try out all possible combinations of predictors and fit the model for each of them. More precisely, the function uses a certain quality measure (like adjusted R^2 and others) to find the best model with a given number of predictors. Let us try this on the Housing data set:

```
library(leaps)
fit.subset <- regsubsets(MedianHouseValue ~ . ,
                         data = HousingData.train)
```

We can use `summary(fit.subset)` as usual to obtain more detailed information on the fit. In particular, `summary(fit.subset)$rsq` gives us the R^2 and `summary(fit.subset)$adjr2` the adjusted R^2, which inform us about the quality of the fit. However, as we know, as long as we evaluate the error on the training set is will always decrease with increases flexibility and the R^2 conversely will increase with increasing flexibility.

The adjusted R^2 tries to compensate for this, but we are better off, as always, evaluating the test error on a separate test data set. A problem that we are facing is that there is no `predict` function implemented for objects of type `regsubsets`. We can work around this by using an explicit implementation of the model matrix for the test data set. Recall from the mathematical notes in handout 1 that

$$X = (\mathbf{1}, \mathbf{x}_1, ..., \mathbf{x}_p)$$

is called the model matrix (it is a $(n \times (p+1))$-dimensional matrix). It can be created in `R` using the function `model.matrix`

```
mmat <- model.matrix(MedianHouseValue ~ . ,
                         data = HousingData.test)
```

```
head(mmat)
```

```
##    (Intercept) MedianIncome HousingMedianAge AverageNoRooms
## 5            1       3.8462               52       6.281853
## 9            1       2.0804               42       4.294118
## 18           1       2.1202               52       4.052805
## 20           1       2.6033               52       5.465455
## 23           1       1.7250               52       5.096234
## 30           1       1.6875               52       4.703226
##    AverageNoBedrooms Population AverageOccupancy Latitude Longitude
## 5          1.0810811        565         2.181467    37.85   -122.25
## 9          1.1176471       1206         2.026891    37.84   -122.26
## 18         0.9669967        648         2.138614    37.85   -122.27
## 20         1.0836364        690         2.509091    37.84   -122.27
## 23         1.1317992       1015         2.123431    37.84   -122.27
## 30         1.0322581        395         2.548387    37.84   -122.28
```

The idea is that we can extract the vector of estimated coefficients of the model $\hat{\mathbf{a}}$ and then simply calculate the matrix vector product

$$\hat{\mathbf{y}} = X\hat{\mathbf{a}}$$

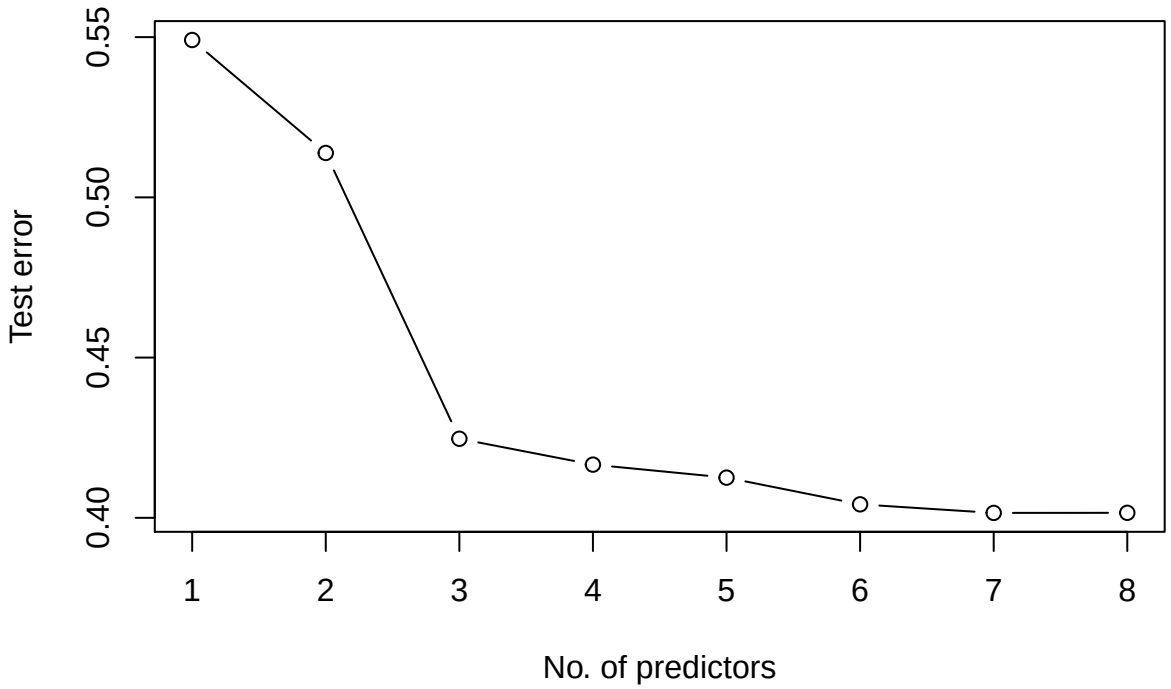
We can extract $\hat{\mathbf{a}}$ using `coef` e.g.

```
coef(fit.subset,id=2)
```

```
##      (Intercept)     MedianIncome HousingMedianAge
##      -0.07329656       0.42289600       0.01555377
```

We can now evaluate the test error for the model with 1 predictor, 2 predictors etc. This is done using the above ingredients inside of a for-loop:

```
testErrors <- rep(NA,8)
for(i in 1:8){
  ahat<- coef(fit.subset,id=i)
  pred <- mmat[,names(ahat)] %*% ahat
  testErrors[i] <- mean((HousingData.test$MedianHouseValue-pred)^2)
}
plot(testErrors, ylab="Test error", xlab = "No. of predictors", type="b")
```



Here `names(ahat)` selects the names of the predictors in `ahat`. As we learned in the first handout, we can index matrices by names, which is what we are doing here. Then `%*%` invokes the standard matrix vector product.

Exercise: Write your own `predict` function for objects of type `regsubsets` which should be `predict.regsubset <- function(object,newdata,id)`. This exercise might be challenging for you. If you get stuck, consult Sec. 6.5.3 of the text book where this and the above are worked out in detail.

Note that if we would have to choose a model for prediction, at first sight, we might consider choosing the model with six predictors. In this case the relevant predictors are:

```
names(coef(fit.subset,id=6))[2:7]
```

```
## [1] "MedianIncome"      "HousingMedianAge"  "AverageNoRooms"
## [4] "AverageNoBedrooms" "Latitude"          "Longitude"
```

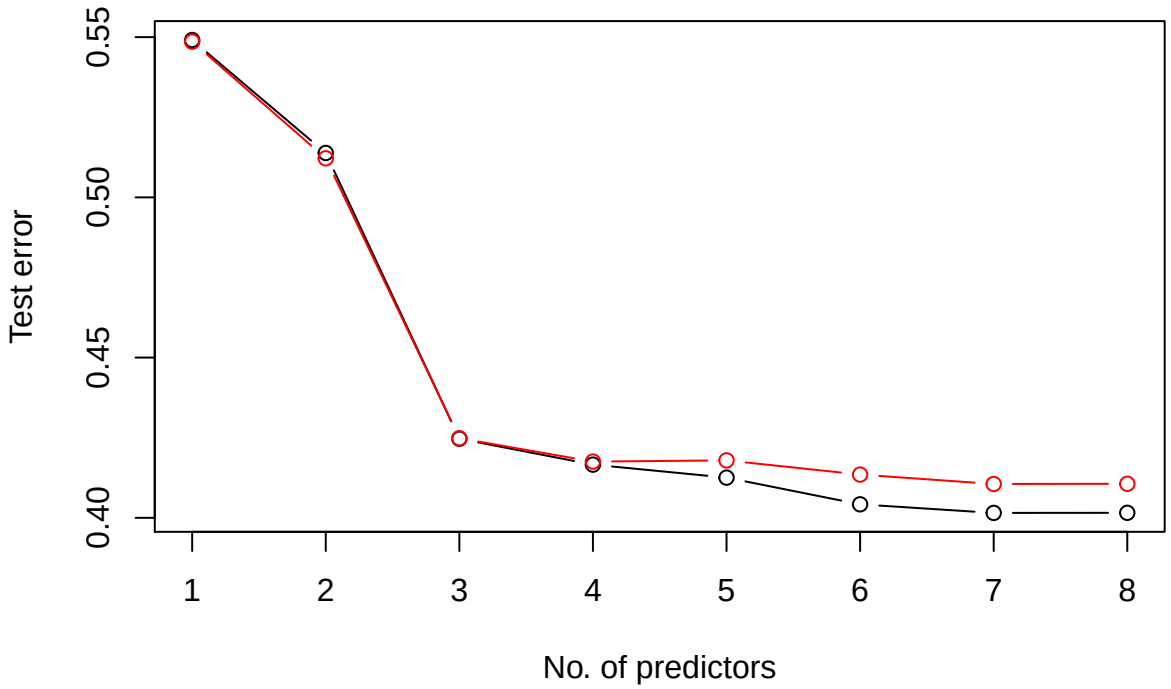
However, the rule is to always choose the least flexible model, if the error changes only very little. Considering this, we could also argue that after having introduced three or four predictors we are only gaining very little when introducing new predictors and we might for robustness and interpretability stick with the model with only three predictors. In this case the relevant predictors are

```
names(coef(fit.subset,id=3))[2:4]
```

```
## [1] "MedianIncome" "Latitude"     "Longitude"
```

i.e. median income and the house location are most informative for inferring its value. You might find the choice a bit arbitrary, but there is a general rule of thumb: What we can do is to check the variance of those points by repeating the above procedure with different random splittings of the data. Then we chose the model which is the least flexible model within one standard deviation from the model with minimal test error, the so-called **one standard deviation rule**.

As an example we redo the analysis with a different random seed and add the new points to the figure (in red).



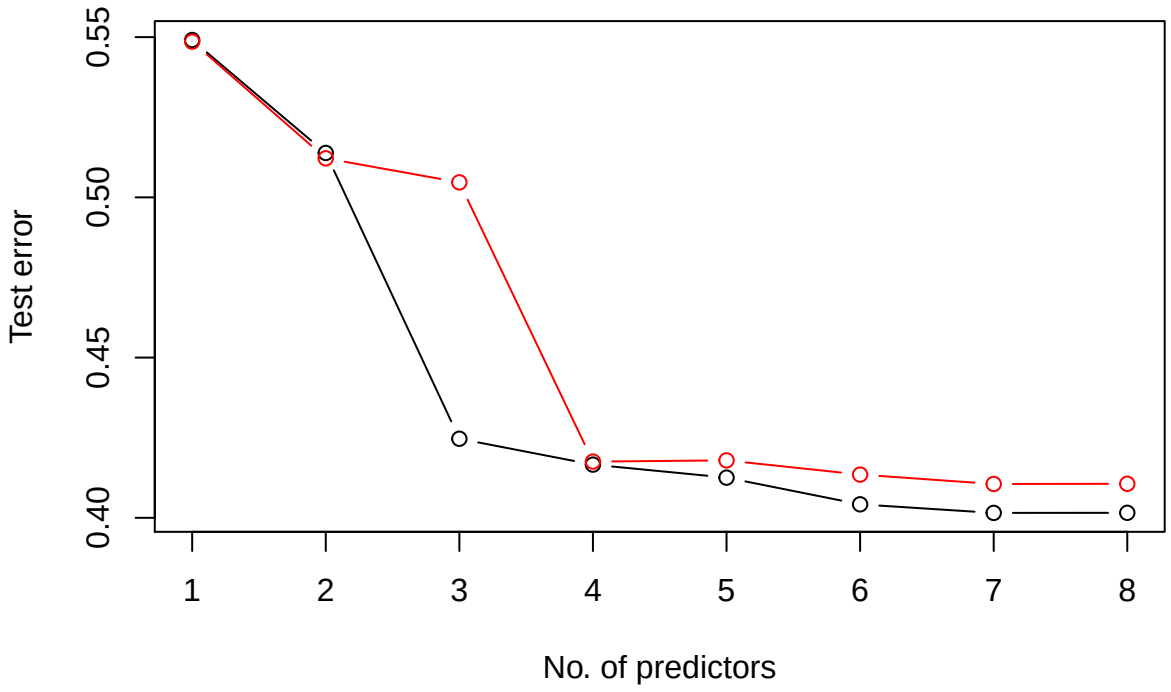
Now it is clearer that the final model should really be the one with 3 or 4 predictors (the latter also includes `HousingMedianAge` in addition to the other predictors).

Forward and backward selection

In best subset selection, as described in the previous section, the algorithm looks first at models with one predictor, then models with two predictors etc. until reaching the maximum number of predictors specified. For each of those cases the algorithm searches for the best possible subset of predictors, i.e. for one predictor this would be the best one, for two predictors this would be any subset of size two, not necessarily including the best predictor obtained in the model with one predictor. This exhaustive search over all possible subsets can easily get very computational intensive if the total number of predictors and the number of predictors in the subset become large. If we have p predictors in total and are searching of subsets of size q, then this number will be 'p choose q'. It is thus often better to use a more 'greedy' approach. For example we could look for the best predictor in the case q=1 and then for q=2 look which would be the best predictor to add to this one and so on. This is called **forward selection**. Alternatively, we could also start off with all predictors and then remove them one by one, always checking which is the best one to remove. This in turn is called **backward selection**. Both of these methods are much less computationally expensive than the full subset selection using exhaustive search. In `R` all those methods can be implemented by choosing different attributes for `method` in the `regsubsets` function. In particular, `method = "exhaustive"` is the standard option and refers to full subset selection using exhaustive search. `method = "forward"` refers to forward selection while `method = "backward"` refers to backward selection.

Exercise: Repeat the analysis from the previous section using both forward and backward selection.

You should see something like the following figure for forward selection, where the black line is our previous result using exhaustive search while the red line shows the corresponding result for forward selection:



Exercise: Exercise can you explain why we see the difference. (Hint: Think of the informative power when just adding one of longitude and latitude without the other.)

Regularisation

So far we have only tuned the flexibility of our models by including or removing certain predictors. In tutorial 1 we did this by hand, which could be done by simply trying out various combinations or using some kind of domain knowledge which supports the choice of certain predictors. In this tutorial we automated this process by using subset selection, using either an exhaustive search or forward and backward selection. In this section we aim to further automate this by introducing a continuous parameter which tunes the flexibility of the model. This is called **regularisation**. We explain the basic principles of regularisation using rigid regression which is one form of its implementation. We then also discuss the lasso, which is another very popular method of regularisation, particularly suited for very high-dimensional problems.

Rigid regression

The basic idea behind rigid regression and regularisation in general is very simple. When we considered subset selection in the previous section, all we did was basically solving the usual least-squares optimisation problem:

$$\min_{\mathbf{a}} \sum_{i=1}^{n} (y_i - f(x_i))^2.$$

subject to the constraint that at a given number of the a_i is set to zero, i.e. all but q entries of $\mathbf{a}$ are fixed to be zero. **Rigid regression** is simply another way to put a constraint on the coefficients a_i to force them to be closer to zero. In particular the optimisation problem of rigid regression can be written as

$$\min_{\mathbf{a}} \sum_{i=1}^{n} \left[y_i - (a_0 + a_1 x_{i1} + ... + a_p x_{ip}) \right]^2 , \qquad \text{subject to} \quad \sum_{i=1}^{p} a_i^2 \leq C$$

In other words, instead of choosing a certain number of coefficients a_i to be zero, we simply limit the value of the sum of their squared values. Here C is called the cost. The smaller the value of C the less flexible the model is, e.g. with very small C only very few coefficients can have a value significantly different from zero. With C becoming larger the model becomes more flexible and for $C \to \infty$ we recover standard linear regression, i.e. an infinite cost is the same as having no constraint.

Mathematically speaking, we can rewrite the constraint optimisation problem above in its conjugate form:

$$\min_{\mathbf{a}} \left(\sum_{i=1}^{n} \left[y_i - (a_0 + a_1 x_{i1} + ... + a_p x_{ip}) \right]^2 \quad + \quad \lambda \sum_{i=1}^{p} a_i^2 \right)$$

Here λ is conjugate to C, i.e. small λ corresponds to large C and vice versa.

Exercise: If λ increases does flexibility go up and down? What about C? Argue using the above optimisation problems.

The first term in the minimisation above is our standard least-squared **cost function**, while the second term in the minimisation is the **penalty term**. This structure of cost function and penalty is very general and appears in many machine learning models of different kinds. The difficulty with more advanced models, like neural networks, is that the cost function becomes non-convex which makes the optimisation problem very difficult, especially if the dimensionality of the parameter space (i.e. the number of coefficients) becomes very large. Nevertheless, the basic principle of cost function and penalty term remains the same.

In `R` rigid regression is implemented in the package `glmnet`.

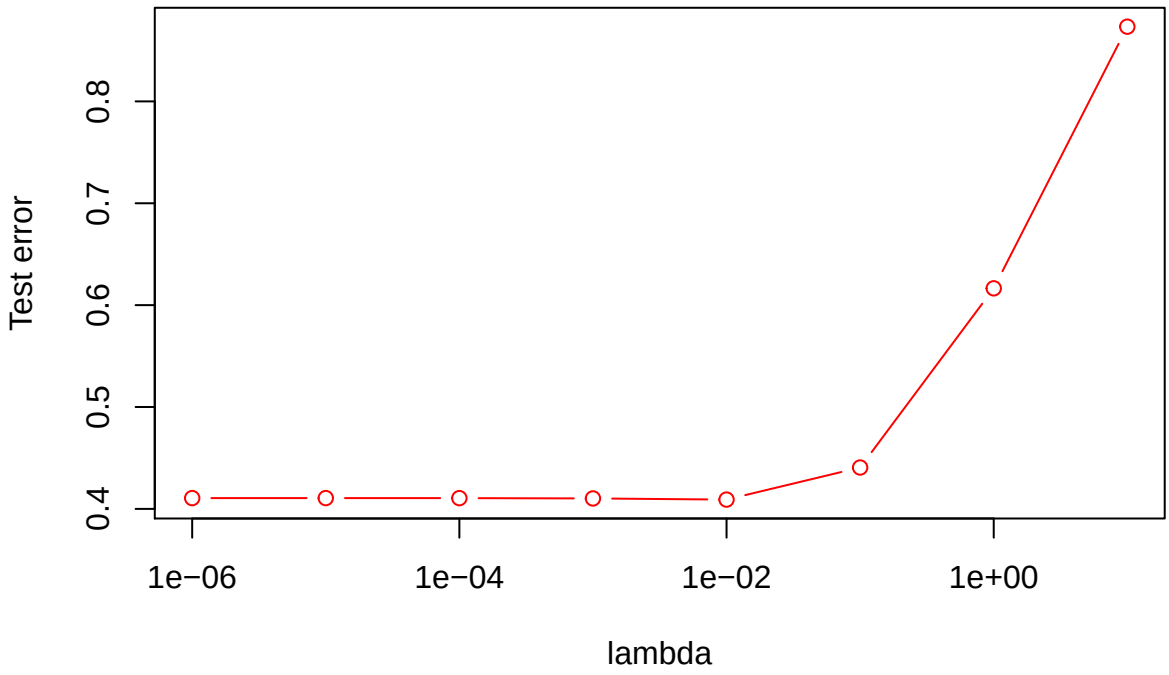
```
library(glmnet)
```

The corresponding function is also called `glmnet`. It can be used the first various regularised models; the parameter `alpha=0` indicates that we want to perform rigid regression. The particular form of the function requires us to split the data table in a y vector and a data matrix containing all other columns, see `?glmnet` for details. Recall that our response variable `MedianHouseValue` is in the first column of the data table, i.e. `HousingData.train[,1]`. We can fit the model using

```
X.train <- as.matrix(HousingData.train[,-1])
y.train <- as.matrix(HousingData.train[,1])
fit.rigid <- glmnet(X.train,y.train,alpha=0,lambda=0.1)
```

Luckily for objects of type `glmnet` the function `predict` is implemented.

Exercise: Perform rigid regression for a range of values for the tuning/shrinkage parameter λ, for example for all elements of the vector `lam.list <- c(1e-6,1e-5,1e-4,1e-3,1e-2,1e-1,1,10)`. For each of those rigid regression models evaluate the test error on the test data set and save it in a vector of the same length as `lam.list`. Produce a plot like the one shown below. How does this relate to the results of the previous section?



Lasso

In the previous section we learned about rigid regression. The **lasso** is nearly the same algorithm with the only difference that in the penalty term we consider $|a_i|$ as opposed to a_i^2. The optimisation problem for the lasso is thus given by

$$\min_{\mathbf{a}} \left(\sum_{i=1}^{n} [y_i - (a_0 + a_1 x_{i1} + ... + a_p x_{ip})]^2 \quad + \quad \lambda \sum_{i=1}^{p} |a_i| \right)$$

You might ask: Does this slight change really make a difference? Indeed is does and we will explore this in the following.

Exercise: Repeat the previous analysis using the lasso. (Simply change `alpha=0` to `alpha=1` in your code.)

The plot for the test error produced above should look roughly the same as for rigid regression. To understand the difference, let us look in more detailed at the estimated coefficients (which are accessed in `glmnet` using `beta`).

```
fit.rigid <- glmnet(X.train,y.train,alpha=0,lambda=0.05)
fit.lasso <- glmnet(X.train,y.train,alpha=1,lambda=0.05)
```

Here are the estimated coefficients for rigid regression:

```
fit.rigid$beta
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                              s0
## MedianIncome       4.094442e-01
## HousingMedianAge   1.046672e-02
```

```
## AverageNoRooms    -7.407463e-02
## AverageNoBedrooms  4.199576e-01
## Population         1.587798e-05
## AverageOccupancy  -3.612073e-03
## Latitude          -2.307629e-01
## Longitude         -2.321618e-01
```

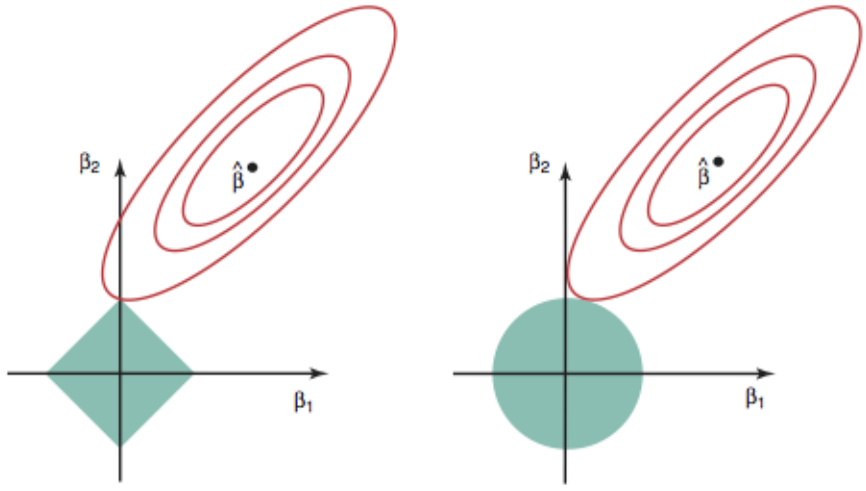
and here are the estimated coefficients for the lasso:

```
fit.lasso$beta
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                              s0
## MedianIncome       0.369265264
## HousingMedianAge   0.008869284
## AverageNoRooms     .
## AverageNoBedrooms  .
## Population         .
## AverageOccupancy   .
## Latitude          -0.098901196
## Longitude         -0.085580703
```

We know that in rigid regression the penalty term forces the coefficients to be smaller, but looking at the output, we cannot really tell which ones are more and which less important. This is mainly because we do not know the scale. One coefficient is of order 10^{-5}, but it could be the coefficient of a predictor which is stated in units of millions. Hence, it might be an important coefficient even though it is apparently small. The lasso has the big advantage that it automatically chooses a number of (less relevant) predictors exactly equal to zero. This is particular important if we, for example, want to interpret the model, e.g. in genetics we might want to know which gene markers were relevant in predicting a certain type of cancer. This feature made the lasso very popular!

It might be surprising how the small change from a_i^2 to $|a_i|$ in the penalty function has such a drastic effect. But we can illustrate this in the following figure which is taken from An Introduction to Statistical Learning, Figure 6.7:



Here we consider the optimisation problem for the lasso and rigid regression in its constraint form

$$\min_{\mathbf{a}} \sum_{i=1}^{n} \left[y_i - (a_0 + a_1 x_{i1} + ... + a_p x_{ip})\right]^2, \qquad \text{subject to} \quad \sum_{i=1}^{p} |a_i|^\delta \leq C$$

with $\delta = 1$ for the lasso and $\delta = 2$ for rigid regression. In the above plot, the optimisation problem for the lasso is illustrated on the left-hand-side and the one for rigid regression on the right-hand-side. The red ellipses illustrate the height lines of the energy surface associated with the cost function $\sum_{i=1}^{n} \left[y_i - (a_0 + a_1 x_{i1} + ... + a_p x_{ip})\right]^2$. If the optimisation problem would be unconstraint, like in ordinary least-squares regression, we would choose the optimal parameters to be the point at the centre of the ellipses. However, due to the constraints we must have $\sum_{i=1}^{p} |a_i| \leq C$ for the lasso, which corresponds to the inside of the route in the left figure, and $\sum_{i=1}^{p} |a_i|^2 \leq C$ which corresponds to the inside of the circle in the right figure. The solution to the optimisation problem is the point which is closest to the centre of the ellipses, but still within the route or circle respectively. This is exactly where in the figure the outer ellipse touches the route or circle.

Interestingly, as is illustrated in the figure, for the case of the lasso, the ellipse is more likely to touch the route on the axis, where precisely one of the two parameters is zero. This also holds for higher-dimensional parameter spaces and is essentially the reason why in the case of the lasso, as opposed to rigid regression, we have that many coefficients are exactly chosen to be zero.

High-dimensional data

In the previous section we have analysed rigid regression and the lasso as ways of tuning the flexibility of the regression model. This is particularly helpful when we have a very large number of predictors, where subset selection becomes very inconvenient. More interestingly, it can even be employed in cases where ordinary least-squares regression fails. This is the case when we have more predictors p than observations n. We can see this mathematically as follows. Recall from the first handout that we can analytically solve the unconstraint optimisation problem of ordinary least-squares regression by differentiating with respect to the coefficients. The solution was given by

$$\hat{\mathbf{a}} = (X^T X)^{-1} X^T \mathbf{y}$$

where the term $(X^T X)^{-1}$ is also called the pseudo-inverse of X. What happens for $p > n$ is that this pseudo-inverse ceases to exist (it becomes infinite because $X^T X$ will have zero eigenvalues). However, if we redo the analysis leading to the above equation, but now starting from the optimisation problem for rigid regression, we obtain

$$\hat{\mathbf{a}} = (X^T X + \lambda I)^{-1} X^T \mathbf{y}$$

Now the smallest eigenvalue of $X^T X + \lambda I$ is not zero anymore, but rather λ. This regularises the inverses, which is the reason why we call it **regularisation**.

We thus see that rigid regression or the lasso are essential when working with data where $p > n$. A particular case are data sets where there are many more predictors than observations, $p \gg n$,in which case we speak about **high-dimensional data**. The problem with high-dimensional data is that the parameter space we optimise over is of very large dimension which makes it particularly difficult to fit the data considering that we have only very few points.

A specific field where high-dimensional data appears very often are biomedical or genetic applications. For example we could have a study on cancer patients, where we only have data from say n=100 patients, but for each we have collected around p=100000 gene markers. Apart from predicting cancer given the data of future patients, we might also want to understand which gene factors are associated with such cancer. In this case, one could use the lasso, to find a small selection of the predictors which explains the data best. In the following section we look at a case study of this type.

Case study: The Arcene data set

In this case study we look at the Arcene data set which is a nice example of high-dimensional data (see Arcene data set webpage for more information). The Arcene data set is data for a classification problem where we want to infer cancer (label `1`) or no cancer (label `-1`) from mass-spectroscopy data. The training set consists of only n=100 observations. However, the mass-spectrometric data gives rise to p=10000 predictors. Thus we have a typical example of high-dimensional data with $p \gg n$.

In importing the data, note that the training and cross-validation sets are in different files. Furthermore, the predictors and the response variable are in different files. We first import the training data and put it in the correct format for using it with `glmnet`.

```
X.train <- read.csv("arcene_train.data.txt", sep=" ",header=FALSE)
X.train$V10001 <- NULL  # this is an artefact from the line break which we remove
X.train <- as.matrix(X.train)
y.train <- read.csv("arcene_train.labels.txt", sep=" ",header=FALSE,col.names='label')
y.train <- as.factor(y.train$label)
```

Exercise: Make sure that you understand what each of the above commands does.

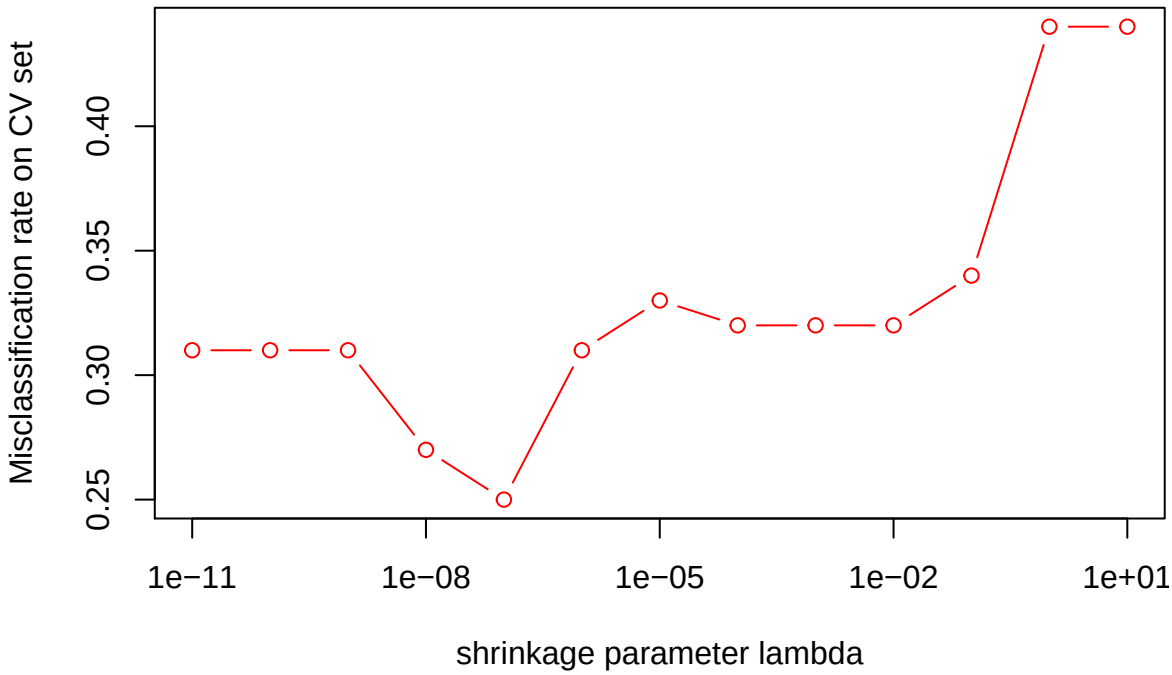
Now we import the cross-validation data in the same way:

```
X.cv <- read.csv("arcene_valid.data.txt", sep=" ",header=FALSE)
X.cv$V10001 <- NULL
X.cv <- as.matrix(X.cv)
y.cv <- read.csv("arcene_valid.labels.txt", sep=" ",header=FALSE,col.names='label')
y.cv <- as.factor(y.cv$label)
```

You can fit a logistic regression with lasso penalty using the `glmnet` function by including the attribute `family="binomial"` to indicate that we are doing logistic regression. This is the same attribute as in the `glm` function. For example:

```
fit.lasso <- glmnet(X.train,y.train,alpha=1,lambda=0.05,family="binomial")
```

Exercise: Write a loop which runs over a range of values for the shrinkage parameter λ. For each of those values, fit a logistic regression with lasso penalty on the training set and then evaluate the error on the cross-validation set. The error in this case is the misclassification rate. Plot the result. Your figure should look like the one below.



Note that the error for $\lambda = 10$ is the misclassification rate which we get when we classify every label as `-1`. Since 56% of all labels are `+1` and 44% are `-1`, we have a misclassification rate of 44% if we simply predict every label as `+1`. Obviously, we want to beat this.

Exercise: For your best model: Investigate the fitted coefficients. Which predictors are relevant for inferring the response variable?