Statistical Learning Home Assignment II
Model assessment and selection

Group 3
Eva Flonner, Anja Hahn, Theresa Traxler

April 2022

Exercise 1

Assume the following data generating process:

$$y = x + x^2 + \epsilon,$$

with $\epsilon \sim N(0, \sigma_\epsilon^2)$ and $x \sim N(0, \sigma_x^2)$ and x and ϵ independent.

- Determine analytically the test error using the squared error loss given parameter estimates $\hat{\vec{\beta}}$.

  The test error is defined as

  $$Err_\tau = E[L(Y, \hat{f}(X)) \mid \tau],$$

  where both X and Y are drawn randomly from their joint distribution and τ is fixed. The loss function $L(.,.)$ in this example is the squared error. The quadratic regression model

  $$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \epsilon_i \; (i = 1, 2, \cdots, N)$$

  can be expressed in matrix form in terms of a design matrix $\mathbf{X}$, a response vector $\vec{y}$, a parameter vector $\vec{\beta}$ and a vector $\vec{\epsilon}$ of random errors, i.e.

  $$y = \mathbf{X}\vec{\beta} + \vec{\epsilon},$$

  or written as a system of linear equations

  $$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & X_N & X_N^2 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_N \end{bmatrix}$$

  The OLS estimate vector is

  $$\hat{\vec{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\vec{y}, \; (p \leq N).$$

  Thus

  $$Err(x_0) = E[(Y - \hat{f}_p(x_0))^2 \mid X = x_0] = \underbrace{\sigma_\epsilon^2 + [E(\hat{f}_p(x_0)) - f(x_0)]^2 + E[\hat{f}_p(x_0) - E(\hat{f}_p(x_0))]^2}_{\text{Bias-Variance decomposition}}$$

  $$= \sigma_\epsilon^2 + [E(\hat{f}_p(x_0)) - f(x_0)]^2 + Var(\hat{f}_p(x_0)) = \sigma_\epsilon^2 + [E(\hat{f}_p(x_0)) - f(x_0)]^2 + Var(\underbrace{x_0^T(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T}_{\text{hat matrix}}\vec{y})$$

  $$= \sigma_\epsilon^2 + [f(x_0) - E[\hat{f}_p(x_0)]^2 + \| \mathbf{h}(x_0) \|^2 \, \sigma_\epsilon^2.$$

  Where $\mathbf{h}(x_0) = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}x_0$. Note that this variance depends on x_0. However, we can take the average and get $(p/N)\sigma_\epsilon^2$, which yields the in-sample error

  $$\frac{1}{N}\sum_{i=1}^{N} Err(x_i) = \sigma_\epsilon^2 + \frac{1}{N}\sum_{i=1}^{N}[f(x_i) - E[\hat{f}(x_i)]]^2 + \frac{p}{N}\sigma_\epsilon^2.$$

- Assume $\sigma_\epsilon^2 = \sigma_x^2 = 1$. Draw a sample of size $N = 40$ as training data and determine the test error using the squared error loss using the analytical formula as well as simulation when estimating the regression coefficients β using OLS.

```
set.seed(1)
#draw sample for epsilon
epsilon <- rnorm(40,0,1)

#draw sample for x

x <- rnorm(40,0,1)

X <- matrix(cbind(rep(1,length(x)),x,x^2))

#note that in the case of this example the
#squared bias is zero due to the form of the true data generating process
test_error <- function(x_0,var_epsilon,X){
    h_squared <- norm(X%*%solve(t(X)%*%X)*x_0,"F")^2
    return(var_epsilon+h_squared*var_epsilon)
}
test_error_vec <- numeric(nrow(X))
for(i in 1:nrow(X)){
    test_error_vec[i] <- test_error(X[i,],1,X)
}
#simulation result based on OLS
test_error_vec

##   [1] 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758
##   [9] 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758
##  [17] 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758
##  [25] 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758
##  [33] 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758 1.005758
##  [41] 1.000156 1.000370 1.002797 1.001784 1.002732 1.002882 1.000765 1.003401
##  [49] 1.000073 1.004471 1.000913 1.002157 1.000670 1.007345 1.011825 1.022585
##  [57] 1.000777 1.006278 1.001869 1.000105 1.033214 1.000009 1.002740 1.000005
##  [65] 1.003181 1.000205 1.018760 1.012368 1.000135 1.027181 1.001302 1.002902
##  [73] 1.002148 1.005024 1.009050 1.000489 1.001132 1.000000 1.000032 1.002001
##  [81] 1.000004 1.000024 1.001359 1.000553 1.001296 1.001443 1.000102 1.002009
##  [89] 1.000001 1.003471 1.000145 1.000808 1.000078 1.009368 1.024284 1.088577
##  [97] 1.000105 1.006844 1.000607 1.000002 1.191568 1.000000 1.001303 1.000000
## [105] 1.001758 1.000007 1.061119 1.026565 1.000003 1.128303 1.000294 1.001463
## [113] 1.000801 1.004384 1.014223 1.000042 1.000222 1.000000 1.000000 1.000696

mean(test_error_vec)
```

```
## [1] 1.008333

#analytic result
analytic_error <- function(var_epsilon,N,p){
    return(var_epsilon+(p/N)*var_epsilon)
}
analytic_error(1,40,2)

## [1] 1.05

var(x)

## [1] 0.8571607
```

- Assume $\sigma_\epsilon^2 = \sigma_x^2 = 1$. Estimate the expected test error for samples of size $N = 40$ used as training data for the squared error loss when estimating the regression coefficients β using OLS and test data sets of suitable size drawn from the data generating process.

  The expected test error is defined as

  $$Err = E[L(Y, \hat{f}(X))] = E[Err_\tau],$$

  averaging over the randomness in the training set.

```
#we repeat the calculation 100 times in order to use
#the average as a proxy for the expectation of the error
set.seed(1)
error_vec <- numeric(100)
var_vec <- numeric(100)
for(i in 1:100){
#draw sample for epsilon
epsilon <- rnorm(40,0,1)
#draw sample for x
x <- rnorm(40,0,1)

y <- x+x^2+epsilon
X <- cbind(rep(1,length(x)),x,x^2)
y_hat <- predict(lm(y~X))
error_vec[i] <- mean((y-y_hat)^2)
var_vec[i] <- var(x)
}
#the expected prediction (or test) error is
mean(error_vec)
```

```
## [1] 0.940969

mean(var_vec)

## [1] 1.062149
```

Interestingly, we observe that the expected test error is lower than the test error. This could mean that we were 'unlucky' with our training data set in the first example. Indeed if we look again at the variation in our x's when calculating the test error this is quite low compared to the true variance in our data generating process $\sigma_x = 1$.

Exercise 2

$$\mathrm{Err}_{in} = \frac{1}{N}\sum_{i=1}^{N} \mathrm{E}_{Y^0}\left[\left(Y_i^0 - \hat{f}(x_i)\right)^2\right]$$

$$\overline{\mathrm{err}} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{f}(x_i)\right)^2$$

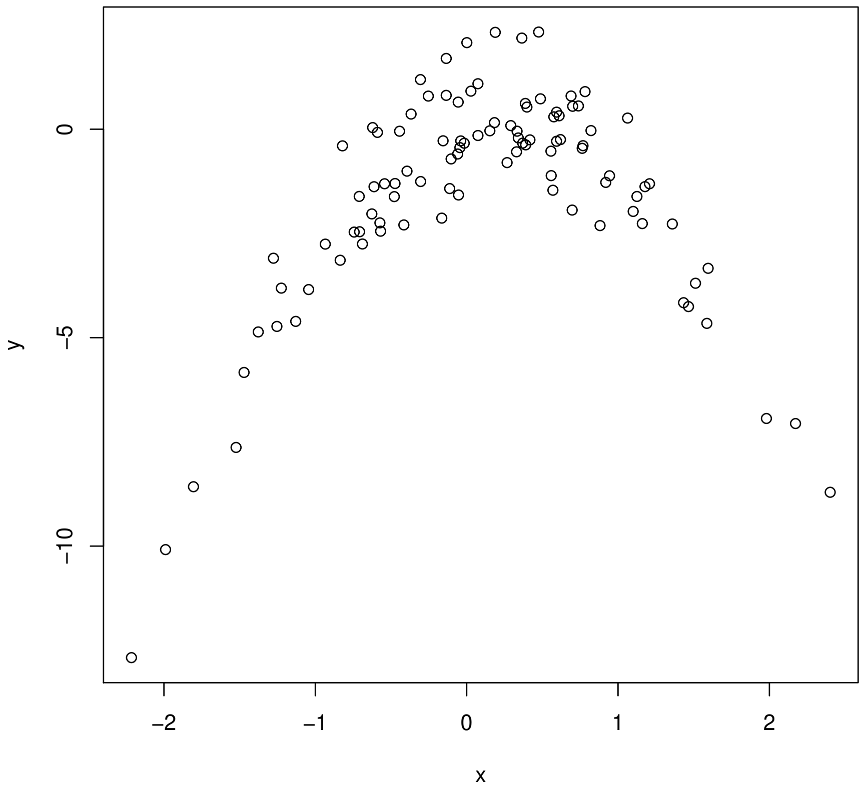
The average optimism is

$$\begin{aligned}
\omega &= \mathrm{E}_y\left(\mathrm{Err}_{in} - \overline{\mathrm{err}}\right)\\
&= \mathrm{E}_y\left\{\frac{1}{N}\sum_{i=1}^{N}\mathrm{E}_{Y^0}\left[\left(Y_i^0 - \hat{f}(x_i)\right)^2\right] - \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{f}(x_i)\right)^2\right\}\\
&= \mathrm{E}_y\left\{\frac{1}{N}\sum_{i=1}^{N}\mathrm{E}_{Y^0}\left[\left(Y_i^0 - \hat{f}(x_i)\right)^2\right] - \left(y_i - \hat{f}(x_i)\right)^2\right\}\\
&\overset{\text{linearity of expectation}}{=} \frac{1}{N}\sum_{i=1}^{N}\mathrm{E}_y\left\{\mathrm{E}_{Y^0}\left[\left(Y_i^0 - \hat{f}(x_i)\right)^2\right]\right\} - \mathrm{E}_y\left[\left(y_i - \hat{f}(x_i)\right)^2\right]\\
&= \frac{1}{N}\sum_{i=1}^{N}\mathrm{E}_y\left\{\mathrm{E}_{Y^0}\left[\left(Y_i^0\right)^2\right] - 2\underbrace{\mathrm{E}_{Y^0}\left[Y_i^0\hat{f}(x_i)\right]}_{=\mathrm{E}_{Y^0}\left[Y_i^0\right]\mathrm{E}_{Y^0}\left[\hat{f}(x_i)\right]} + \mathrm{E}_{Y^0}\left[\left(\hat{f}(x_i)\right)^2\right]\right\} - \mathrm{E}_y\left[y_i^2 - 2y_i\hat{f}(x_i) + \hat{f}(x_i)^2\right]\\
&\overset{\text{tower property}}{=} \frac{1}{N}\sum_{i=1}^{N}\mathrm{E}_y\left[(y_i)^2\right] - 2\mathrm{E}_y(y_i)\,\mathrm{E}_y(\hat{y}_i) + \mathrm{E}_y\left[(\hat{y}_i)^2\right] - \mathrm{E}_y\left(y_i^2\right) + 2\mathrm{E}_y(y_i\hat{y}_i) - \mathrm{E}_y\left(\hat{y}_i^2\right)\\
&= \frac{2}{N}\sum_{i=1}^{N}\mathrm{E}_y(y_i\hat{y}_i) - \mathrm{E}_y(y_i)\,\mathrm{E}_y(\hat{y}_i) = \frac{2}{N}\sum_{i=1}^{N}\mathrm{Cov}_y(\hat{y}_i, y_i)
\end{aligned}$$

Exercise 3

We generate a simulated data set and plot X against Y.

```
set.seed(1)
generate <- function(n){
  x <- rnorm(100)
  y <- x - 2*x^2 + rnorm(100)
  data.frame(cbind(x,y))
}
data <- generate(100)
plot(data)
```



The relationship between clearly is not linear but of higher order. We calculate the AIC values for four models using least squares:

1. $Y = \beta_0 + \beta_1 X + \epsilon$

2. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$
3. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$
4. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4 + \epsilon$

The AIC can be computed using

$$AIC = 2K - 2\ln(L),$$

where K gives the number of independent variables and L is the likelihood for the respective model. The best-fit model is the one that explains the greatest amount of variation using the fewest possible variables.

```
model1 <- glm(y ~ x, data, family = gaussian()) #lm would give the same here
model2 <- glm(y ~ x + I(x^2), data, family = gaussian())
model3 <- glm(y ~ x + I(x^2) + I(x^3), data, family = gaussian())
model4 <- glm(y ~ x + I(x^2) + I(x^3) + I(x^4), data, family = gaussian())
models <- list(model1,model2,model3,model4)

AIC(model1,model2,model3,model4)

##        df      AIC
## model1  3 478.8804
## model2  4 280.1670
## model3  5 282.0886
## model4  6 282.2963

#or by hand
AIC_function <- function(model){
  AIC <- lapply(models, function(x) -2*as.numeric(logLik(x)) + 2*(length(x$coefficients)+1)
  unlist(AIC)
}
AIC_function(models)

## [1] 478.8804 280.1670 282.0886 282.2963
```

We observe that the second model has the lowest AIC and therefore has the most parsimonious fit, as we expected it from the plot.

Finally we determine in-sample error estimates by drawing suitable test data from the data generating process using twice the negative log-likelihood as loss function.

```
test <- generate(100)
m1_test <- glm(y ~ x, test, family = gaussian()) #lm would give the same here
m2_test <- glm(y ~ x + I(x^2), test, family = gaussian())
m3_test <- glm(y ~ x + I(x^2) + I(x^3), test, family = gaussian())
m4_test <- glm(y ~ x + I(x^2) + I(x^3) + I(x^4), test, family = gaussian())
```

```
models_test <- list(m1_test,m2_test,m3_test,m4_test)

error.estimates <- lapply(models_test, function(x) -2*logLik(x))
error.estimates

## [[1]]
## 'log Lik.' 526.7693 (df=3)
##
## [[2]]
## 'log Lik.' 279.7447 (df=4)
##
## [[3]]
## 'log Lik.' 278.8 (df=5)
##
## [[4]]
## 'log Lik.' 278.7059 (df=6)
```

When comparing the values to the AIC values, we can see that again the linear model has the worse fit. However, the error estimates are now further decreasing with higher order, since the penalty for increasing number of parameters of the model which is included in the AIC is now missing.

Exercise 4

We generate a simulated data set and plot X against Y.

```
set.seed(1)
x <- rnorm(100)
y <- x - 2*x^2 + rnorm(100)
data <- data.frame(cbind(x,y))
```

We set a random seed and then compute the LOOCV, kCV and empirical bootstrap errors based on the mean squared error loss that results from fitting the following four models using least squares:

1. $Y = \beta_0 + \beta_1 X + \epsilon$
2. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$
3. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$
4. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4 + \epsilon$

```
set.seed(123)
model1 <- glm(y ~ x, data, family = gaussian()) #lm would give the same here
```

```r
model2 <- glm(y ~ x + I(x^2), data, family = gaussian())
model3 <- glm(y ~ x + I(x^2) + I(x^3), data, family = gaussian())
model4 <- glm(y ~ x + I(x^2) + I(x^3) + I(x^4), data, family = gaussian())
models <- list(model1,model2,model3,model4)

#LOOCV
library(boot)
#not specifying K in the cv.glm() function gives LOOCV
LOOCV_errors <- do.call(rbind.data.frame,lapply(models, function(x)
                      cv.glm(data,x)$delta))
colnames(LOOCV_errors) <- c("LOOCV error","adjusted LOOCV error")
LOOCV_errors

##   LOOCV error adjusted LOOCV error
## 1   7.2881616            7.2847441
## 2   0.9374236            0.9371789
## 3   0.9566218            0.9562538
## 4   0.9539049            0.9534453

#kCV
kCV_errors <- do.call(rbind.data.frame,lapply(models, function(x)
                      cv.glm(data,x,K=10)$delta))
colnames(kCV_errors) <- c("kCV error","adjusted kCV error")
kCV_errors

##   kCV error adjusted kCV error
## 1 7.4053689          7.3624788
## 2 0.9377124          0.9351325
## 3 0.9704647          0.9658854
## 4 0.9609511          0.9560252

#bootstrap
MSE <- function(formula, Data,i){
  fit <- glm(formula, Data[i,], family = gaussian())
  return(mean(fit$residuals^2))
}
boot1 <- boot(data=data,statistic = MSE,R=1000, formula=y~x)
boot2 <- boot(data=data,statistic = MSE,R=1000, formula=y~x+I(x^2))
boot3 <- boot(data=data,statistic = MSE,R=1000, formula=y~x+I(x^2)+I(x^3))
boot4 <- boot(data=data,statistic = MSE,R=1000, formula=y~x+I(x^2)+I(x^3)+I(x^4))
bootstrap_errors <- rbind(boot1[1],boot2[1],boot3[1],boot4[1])
colnames(bootstrap_errors) <- "bootstrap error"
bootstrap_errors

##      bootstrap error
## [1,] 6.625504
## [2,] 0.8902911
## [3,] 0.8895936
## [4,] 0.8737907

#summary
errors1 <- cbind(LOOCV_errors[,1], kCV_errors[,1], bootstrap_errors)
colnames(errors1) <- c("LOOCV","kCV","bootstrap")
errors1

##      LOOCV     kCV       bootstrap
## [1,] 7.288162  7.405369  6.625504
## [2,] 0.9374236 0.9377124 0.8902911
## [3,] 0.9566218 0.9704647 0.8895936
## [4,] 0.9539049 0.9609511 0.8737907
```

We repeat the computations using another random seed and report our results.

```
set.seed(345)
model1 <- glm(y ~ x, data, family = gaussian()) #lm would give the same here
model2 <- glm(y ~ x + I(x^2), data, family = gaussian())
model3 <- glm(y ~ x + I(x^2) + I(x^3), data, family = gaussian())
model4 <- glm(y ~ x + I(x^2) + I(x^3) + I(x^4), data, family = gaussian())
models <- list(model1,model2,model3,model4)

#LOOCV
LOOCV_errors <- do.call(rbind.data.frame,lapply(models, function(x)
                      cv.glm(data,x)$delta))
colnames(LOOCV_errors) <- c("LOOCV error","adjusted LOOCV error")
LOOCV_errors

##   LOOCV error adjusted LOOCV error
## 1   7.2881616            7.2847441
## 2   0.9374236            0.9371789
## 3   0.9566218            0.9562538
## 4   0.9539049            0.9534453

#kCV
kCV_errors <- do.call(rbind.data.frame,lapply(models, function(x)
                      cv.glm(data,x,K=10)$delta))
colnames(kCV_errors) <- c("kCV error","adjusted kCV error")
kCV_errors

##   kCV error adjusted kCV error
## 1 7.0479657          7.0256411
## 2 0.9456304          0.9426465
## 3 0.9447099          0.9415802
## 4 0.9580756          0.9533565

#bootstrap
MSE <- function(formula, Data,i){
  fit <- glm(formula, Data[i,], family = gaussian())
  return(mean(fit$residuals^2))
}
boot1 <- boot(data=data,statistic = MSE,R=1000, formula=y~x)
boot2 <- boot(data=data,statistic = MSE,R=1000, formula=y~x+I(x^2))
boot3 <- boot(data=data,statistic = MSE,R=1000, formula=y~x+I(x^2)+I(x^3))
boot4 <- boot(data=data,statistic = MSE,R=1000, formula=y~x+I(x^2)+I(x^3)+I(x^4))
bootstrap_errors <- rbind(boot1[1],boot2[1],boot3[1],boot4[1])
colnames(bootstrap_errors) <- "bootstrap error"
bootstrap_errors

##      bootstrap error
## [1,] 6.625504
## [2,] 0.8902911
## [3,] 0.8895936
## [4,] 0.8737907

#summary
errors2 <- cbind(LOOCV_errors[,1], kCV_errors[,1], bootstrap_errors)
colnames(errors2) <- c("LOOCV","kCV","bootstrap")
errors2

##      LOOCV     kCV       bootstrap
## [1,] 7.288162  7.047966  6.625504
## [2,] 0.9374236 0.9456304 0.8902911
## [3,] 0.9566218 0.9447099 0.8895936
## [4,] 0.9539049 0.9580756 0.8737907
```

The results are the same for both random seeds, except for kCV. This is because in general the randomness lies in the data generation, while in the case of kCV the split into k subsets is an additional source of randomness. LOOCV suggests that the second model is the best, whereas for kCV the error for the third model is even slightly lower before it goes up again. The empirical bootstrap error decreases further with model complexity, reflecting potential overfitting problems of this method. The outcome is in line with our expectation, because from the plot we saw that higher order regression equations fit much better to the data than the linear case. Finally we have a look at the statistical significance of the coefficient estimates that result from fitting each of the models using least squares.

```
set.seed(123)
model1 <- glm(y ~ x, data, family = gaussian())
model2 <- glm(y ~ x + I(x^2), data, family = gaussian())
model3 <- glm(y ~ x + I(x^2) + I(x^3), data, family = gaussian())
model4 <- glm(y ~ x + I(x^2) + I(x^3) + I(x^4), data, family = gaussian())
models <- list(model1,model2,model3,model4)
lapply(models,function(x) summary(x)$coefficients)
```

```
## [[1]]
##              Estimate Std. Error   t value     Pr(>|t|)
## (Intercept) -1.625427  0.2619366 -6.205420 1.309300e-08
## x            0.692497  0.2909418  2.380191 1.923846e-02
##
## [[2]]
##                Estimate Std. Error    t value     Pr(>|t|)
## (Intercept)  0.05671501  0.1176555   0.482043 6.308613e-01
## x            1.01716087  0.1079827   9.419666 2.403287e-15
## I(x^2)      -2.11892120  0.0847657 -24.997388 4.584330e-44
##
## [[3]]
##                Estimate Std. Error     t value     Pr(>|t|)
## (Intercept)  0.06150718 0.11950374   0.5146883 6.079538e-01
## x            0.97528027 0.18728149   5.2075636 1.089350e-06
## I(x^2)      -2.12379099 0.08700251 -24.4106856 5.873444e-43
## I(x^3)       0.01763858 0.06429037   0.2743580 7.843990e-01
##
## [[4]]
##                 Estimate Std. Error     t value     Pr(>|t|)
## (Intercept)  0.156702953 0.13946192   1.1236253 2.640034e-01
## x            1.030825643 0.19133655   5.3874999 5.174326e-07
## I(x^2)      -2.409898183 0.23485506 -10.2612148 4.575229e-17
## I(x^3)      -0.009132904 0.06722881  -0.1358481 8.922288e-01
## I(x^4)       0.069785421 0.05324006   1.3107691 1.930956e-01
```

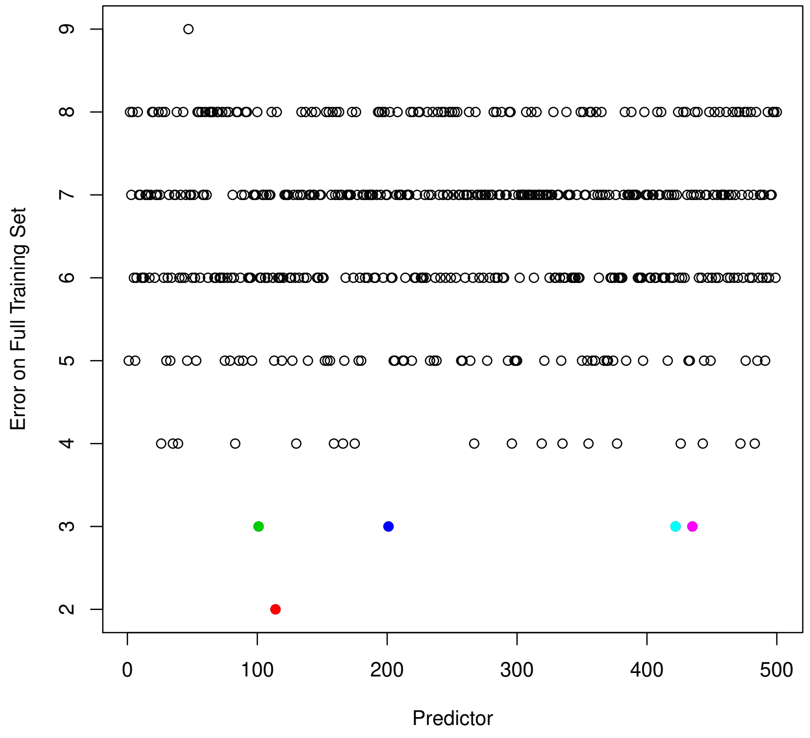
The results agree with our previous conclusions, as the quadratic term has the lowest p-value.

Exercise 6

Consider a scenario with $N = 20$ observations. Half of the observations have class label “0” and the other half class label “1” assigned. In addition for each observation $p = 500$ quantitative predictors are available which are independent from the class labels and which are drawn from a multivariate standard normal distribution. The true error rate thus is 50%. Consider a single univariate classifier: for each predictor variable a single split that minimizes the misclassification error (a “stump”). Assume that 5-fold cross-validation is performed. Compare the following visually:

- The predictor variable versus the error on the full training set for this predictor (in-sample). Mark the predictors where the error on the full training set is minimal.

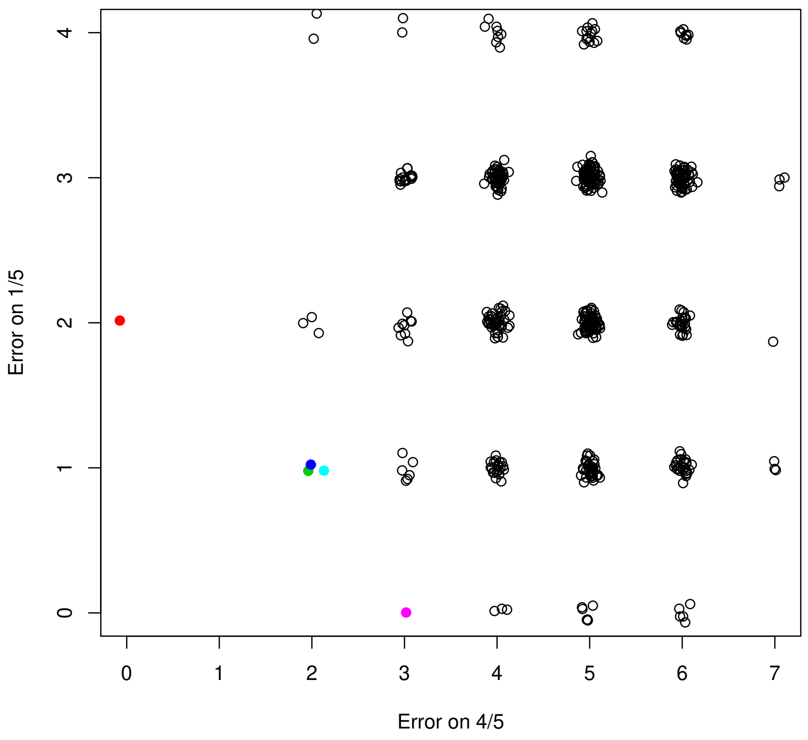
```
set.seed(12)
library(MASS)

#define stump function - taken from 'freestats'


decisionStump <- function(X,y,nstep){

    theta <- vector()
    best.cost <- vector()
    plot_x <- c()
    plot_y <- c()
    thresholds <- c()
    dir <- c()
    counter <- 1
    numSteps <- nstep
    for(d in 1:dim(X)[2]){
      Compare <- 10
      plot_y[counter] <- Compare
      rangeMin <- min(X[,d])
      rangeMax <- max(X[,d])
      stepSize <- (rangeMax-rangeMin)/numSteps
      for (n in 1:dim(X)[1]){
      #for (j in -1:(numSteps+1)){
        #threshVal = (rangeMin + j * stepSize)
        threshVal = X[n,d]
        for (inequal in c('lt', 'gt')){
          if(inequal == "gt"){
            yhat <- 2*(X[,d]>threshVal)-1
          }else{
            yhat <- 2*(X[,d]<threshVal)-1
            }
            cost.temp <- sum(y!=yhat)
            #plot_x[counter] <- d
            #plot_y[counter] <- cost.temp
            #counter <- counter + 1
            if (cost.temp<=Compare){
              plot_y[counter] <- cost.temp
              thresholds[counter] <- threshVal
              dir[counter] <- inequal
              Compare <- cost.temp

            }
        }
      }
      plot_x[counter] <- d
      #plot_y[counter] <- cost.temp
      counter <- counter + 1
    }
    res <-list(directions=dir,threshold_values=thresholds,plot_y=plot_y,plot_x=plot_x)
    class(res) <- 'ds'
    return(res)
}

N <- 20
p <- 500

class_one <- rep(-1,N/2)
class_two <- rep(1,N/2)

labels <- c(class_one,class_two)
shuffled_labels <- sample(labels,20)

predictors <- mvrnorm(20,rep(0,500),diag(500))

res <- decisionStump(X=predictors, y=labels,nstep=5)
index_best_intermediate <- which(res$plot_y %in% sort(res$plot_y)[1:4])
err_best <- res$plot_y[index_best_intermediate]
table <- matrix(c(err_best,index_best_intermediate),ncol=2,nrow=length(index_best_inter
index_best <-table[order(table[,1],decreasing=FALSE),][,2]
print(index_best)

## [1] 114 101 201 422 435

plot(res$plot_x,res$plot_y,ylab="Error on Full Training Set",
xlab="Predictor")
for(i in 1:length(index_best)){
  points(res$plot_x[index_best[i]],res$plot_y[index_best[i]],col=i+1,pch=19)

}
```



We highlight the predictors where the error is on the two lowest levels. The red dot corresponds to the minimal error.

- The error on 4/5 of the data set versus the error on 1/5 of the data set trained on 4/5 of the data set. Mark the same predictors as before.

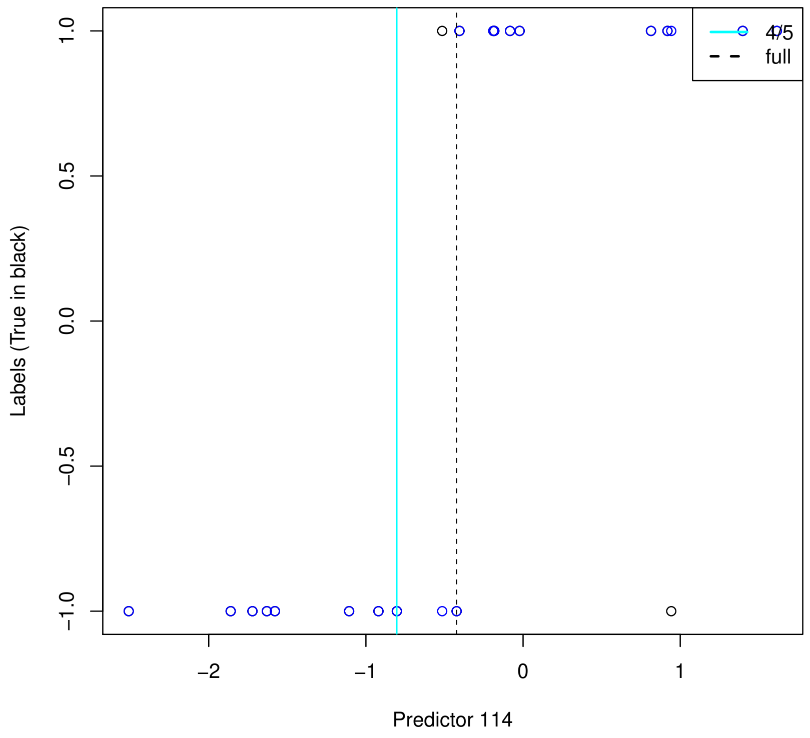
```
index <- sample(1:20,4)
test_labels <- labels[index]
test_predictors <- predictors[index,]

train_labels <- labels[-index]
train_predictors <- predictors[-index,]

res2 <- decisionStump(X=train_predictors, y=train_labels,nstep=5)
test_error <- c()
for(d in 1:dim(test_predictors)[2]){
  if(res2$directions[d]=="gt"){
    yhat_test <- 2*(test_predictors[,d]>res2$threshold_values[d])-1
  }else{
    yhat_test <- 2*(test_predictors[,d]<res2$threshold_values[d])-1
  }
  test_error[d] <- sum(test_labels!=yhat_test)
}
plot(res2$plot_y[-index_best]+rnorm(length(res2$plot_y[-index_best]),0,0.05),test_error
ylab="Error on 1/5",xlab="Error on 4/5",xlim=c(0,7),ylim=c(0,4))
for(i in 1:length(index_best)){
  points(res2$plot_y[index_best[i]]+rnorm(1,0,0.05),test_error[index_best[i]]+rnorm(1,0
}
```



- Select one of the marked predictors and compare the values of the predictor with the class labels and indicate the split points determined for 4/5 and the full data set.

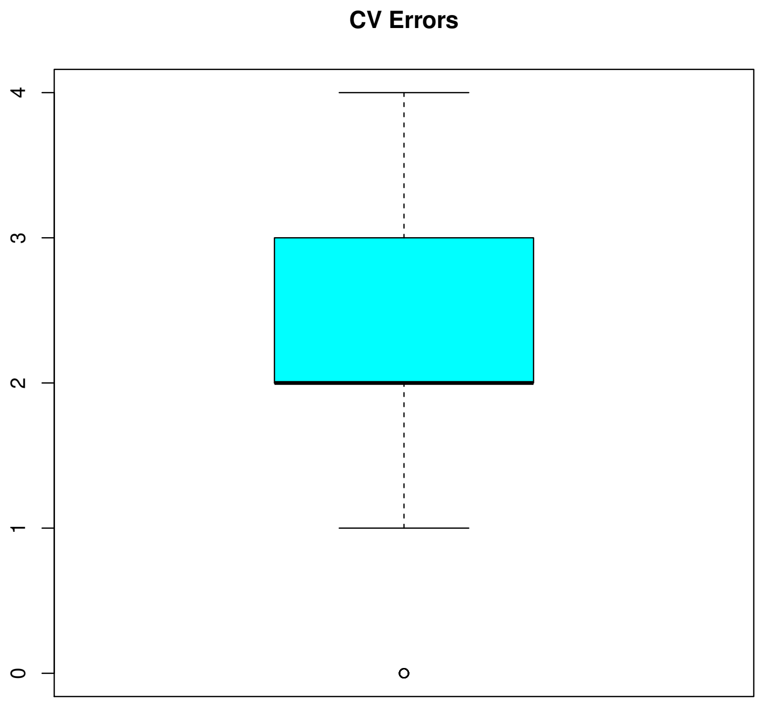
```
index <- index_best[1]
plot(predictors[,index],labels,xlab=paste("Predictor",index_best[1]),
ylab="Labels (True in black)")
```

```r
if(res$directions[index]=="gt"){
  pred_vals <- 2*(predictors[,index]>res$threshold_values[index])-1
}else{
  pred_vals <- 2*(predictors[,index]<res$threshold_values[index])-1
}
points(predictors[,index],pred_vals,col="blue")
abline(v=res2$threshold_values[index],col=5)
abline(v=res$threshold_values[index],lty=2)
legend(x = "topright",          # Position
       legend = c("4/5", "full"),  # Legend texts
       lty = c(1, 2),           # Line types
       col = c(5, 1),           # Line colors
       lwd = 2)
```



- Create a boxplot of the CV errors, i.e., the error on 1/5 of the data set trained on 4/5 of the data set.

```r
boxplot(test_error,main ="CV Errors",col="cyan")
```



- Explain why even if misleadingly good predictors might be identified based on the full training dataset, CV still works if CV is suitably applied

  We see that the stump for the red predictor (whose stump was the best for the full data set), makes two out of four test errors (50%), and is no better than random. The best predictor and corresponding split point are found from 4/5ths of the data. Since the class labels are independent of the predictors, the performance of a stump on the 4/5ths training data contains no information about its performance in the remain- ing 1/5th. Let us focus on the plot of the effect of the choice of split point. The split based on the full data makes no errors on the 1/5ths data. But cross-validation must base its split on the 4/5ths data, and this incurs two errors out of four samples.

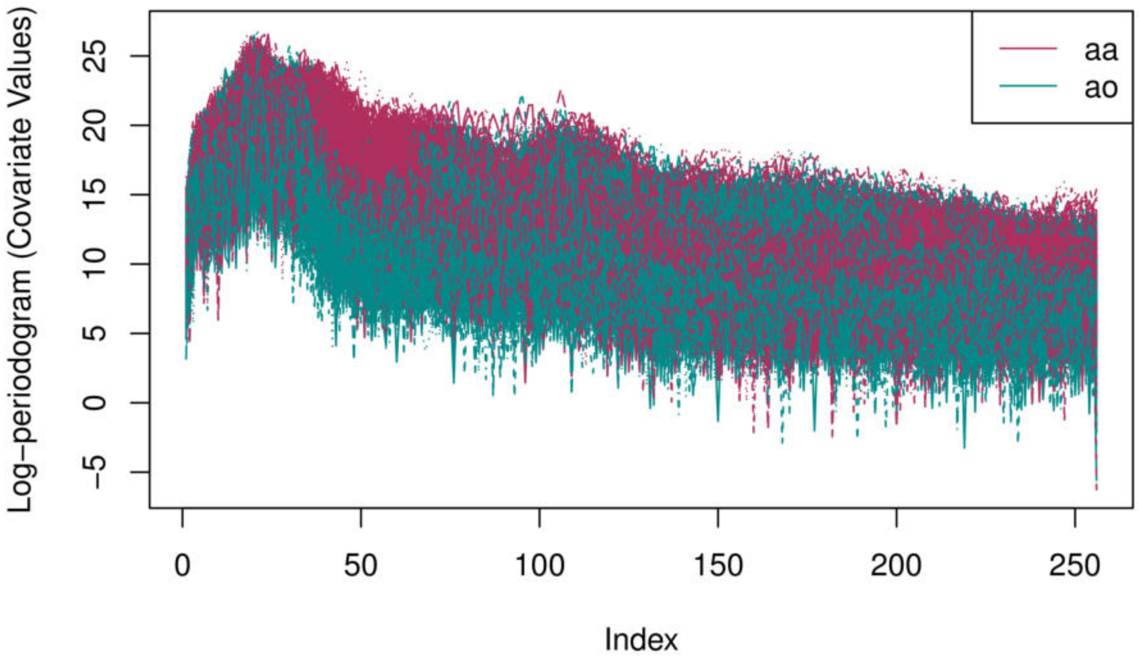
Statistical_Learning_Exercise 5

2022-04-13

Exercise 5

First, we load the data and construct the required subset. Then, we plot the resulting 1,717 datapoints – the values of the covariates are on the y-axis, the label of the respective covariate on the x-axis.

```r
phoneme <- ElemStatLearn::phoneme #load data
phoneme <- subset(phoneme, g == "aa" | g == "ao")
phoneme$y <- ifelse(phoneme$g == "aa", 0, 1)

y <- phoneme[,259]                #vector of group classification (dependent
                                  #variable), aa = 0, ao = 1
X <- as.matrix(phoneme[,1:256])   #covariate matrix
data <- as.data.frame(cbind(y,X))

colorcode <- ifelse(y == 0, "maroon", "darkcyan")
matplot(1:256, t(X[1:1717,]), type="l", col = colorcode, xlab = "Index",
        ylab = "Log-periodogram (Covariate Values)")
legend("topright", legend = c("aa", "ao"), lty = c("solid", "solid"),
       col = c("maroon", "darkcyan"))
```



```r
#matplot for plotting multiple sets of observations
```

From visual inspection, "aa" seems to have generally higher values than "ao". Next, we select a sample of 1,000 and use it as training data.

```r
train_index <- sample(1:nrow(data), 1000, replace = FALSE)
train <- data[train_index, ]
X_train <- as.matrix(train[,-1])
y_train <- train[,1]

test <- data[-train_index, ]
X_test <- as.matrix(test[,-1])
y_test <- test[,1]
```

Next, we fit a logistic regression using all the covariates.

```r
fit.train <- glm(y ~ ., data = train, family = binomial)
```

We compute the misclassification rate by comparing predicted and actual values. Therefore, we predict/calculate probabilities of being either "aa" or "ao" by using the training as well as the test data.

```r
probabilities_train <- predict(fit.train, type = "response")
head(probabilities_train)
```

```
##        3414        2688        2960        2653        1633        2393
## 0.996884722 0.999894926 0.054876194 0.002474771 0.997866876 0.997626679
```

```
probabilities_test <- predict(fit.train, newdata = test, type = "response")
head(probabilities_test)
```

```
##            5            7            9           14           19
## 0.00059742083 0.00519309908 0.54246317847 0.00008293665 0.99999999099
##           20
## 0.99999999617
```

These are the predicted probabilities of our datapoints being classified as "ao" (=1 in our classification scheme). Assuming that datapoints with probability above 0.5 belong to class "ao", we have the following predicted classes:

```
train$predicted_y <- ifelse(probabilities_train > 0.5, 1, 0)
table(train$predicted_y)
```

```
##
##   0   1
## 388 612
```

```
test$predicted_y <- ifelse(probabilities_test > 0.5, 1, 0)
table(test$predicted_y)
```

```
##
##   0   1
## 309 408
```

We now compare this to the actual class by

```
misclassification_train <- nrow(train[train$y != train$predicted_y,]) / nrow(train)
misclassification_train
```

```
## [1] 0.073
```

```
misclassification_test <- nrow(test[test$y != test$predicted_y,]) / nrow(test)
misclassification_test
```

```
## [1] 0.2161785
```

which gives us a misclassification rate of about 7% for the training and 22% for the test data. The log likelihood from the model based on the training data can be extracted by the logLik() command (for the fitted model) or manually by deriving it from the likelihood function (multivariate Bernoulli)

$$\prod_{i=1}^{N} P^{y_i}(1-P)^{1-y_i}$$

which gives us

$$\sum_{i=1}^{N} y_i \log(P) + (1 - y_i)\log(1 - P)$$

```
loglike_train <- sum(train$y * log(probabilities_train) + (1 - train$y) *
                         log(1-probabilities_train))
loglike_train
```

```
## [1] -192.006
```

```
loglike_test <- sum(test$y * log(probabilities_test) + (1 - test$y) *
                        log(1-probabilities_test))
loglike_test
```

```
## [1] -840.2134
```

```
logLik(fit.train) #(to double-check)
```

```
## 'log Lik.' -192.006 (df=257)
```

The log likelihood can be used as a goodness of fit measuremenet for deciding between models. The higher the log likelihood, the better the model fits the data.

In order to deal with the complexity of the model, we can use splines. This restricts the coefficients to vary smoothly over the given covariates, resulting in similar coefficients for close covariates.

```
H <- ns(1:256, df = 12)
Xstar_train <- X_train %*% H
train12 <- as.data.frame(cbind(y_train, Xstar_train))

Xstar_test <- X_test %*% H
test12 <- as.data.frame(cbind(y_test, Xstar_test))

fit.train12 <- glm(y_train ~ ., data = train12, family = binomial)


#get the probabilities
probabilities_train12 <- predict(fit.train12, type = "response")
head(probabilities_train12)
```

```
##        3414        2688        2960        2653        1633        2393
## 0.928068679 0.922122021 0.383420470 0.007209573 0.801963150 0.792415768
```

```
probabilities_test12 <- predict(fit.train12, newdata = test12, type = "response")
head(probabilities_test12)
```

```
##         5         7         9        14        19        20
## 0.3768331 0.5577701 0.7027988 0.1207759 0.8855641 0.9503101
```

```
#get the predicted values
train12$predicted_y <- ifelse(probabilities_train12 > 0.5, 1, 0)
table(train12$predicted_y)
```

```
##
##   0   1
## 366 634
```

```
test12$predicted_y <- ifelse(probabilities_test12 > 0.5, 1, 0)
table(test12$predicted_y)
```

```
##
##   0   1
## 284 433
```

```
#calculate the misclassification rate
misclassification_train12 <- nrow(train12[train12$y != train12$predicted_y,]) / nrow(train12)
misclassification_train12
```

```
## [1] 0.179
```

```
misclassification_test12 <- nrow(test12[test12$y != test12$predicted_y,]) / nrow(test12)
misclassification_test12
```

```
## [1] 0.167364
```

We see that the misclassification rate gets considerably higher for the train dataset and considerably smaller for the test dataset. The two rates now get very similar for the training and the test data.

```
#calculate the log likelihood

log_like_train12 <- sum(train12$y * log(probabilities_train12) + (1 - train12$y) * log(1-probabilities_
log_like_train12
```

```
## [1] -399.8681
```

```
log_like_test12  <- sum(test12$y * log(probabilities_test12) + (1 - test12$y) *
                    log(1-probabilities_test12))
log_like_test12
```

```
## [1] -278.4986
```

```
logLik(fit.train12) #to double-check
```

```
## 'log Lik.' -399.8681 (df=13)
```

The same happens to the log likelihood which gets worse for the train data but better for the test data and more similar between train and test data.

Now we vary the degrees of freedom in the spline basis expansion and therefore use values $2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8$. Again, we calculate misclassification rates and log likelihoods.

```r
#create help functions
#n as degrees of freedom
get.models <- function(n){
  H_n <- ns(1:256, df=n)
  Xstar_train_n <- X_train %*% H_n
  train_n <- as.data.frame(cbind(y_train, Xstar_train_n))
  model_n <- glm(y_train ~ ., data = train_n, family = binomial)
  return(model_n)
}

predicted_y_train <- function(n, model_n){
  H_n <- ns(1:256, df=n)
  Xstar_train_n <- X_train %*% H_n
  train_n <- as.data.frame(cbind(y_train, Xstar_train_n))
  train_n$probabilities_n <- predict(model_n, type = "response")
  predicted_y_n <- ifelse(train_n$probabilities_n > 0.5, 1, 0)
  train_n$predicted_y_n <- predicted_y_n
  names(train_n)[1] <- "y"
  return(train_n)
}

predicted_y_test <- function(n, model_n){
  H_n <- ns(1:256, df=n)
  Xstar_test_n <- X_test %*% H_n
  test_n <- as.data.frame(cbind(y_test, Xstar_test_n))
  test_n$probabilities_n <- predict(model_n, newdata = test_n, type = "response")
  predicted_y_n <- ifelse(test_n$probabilities_n > 0.5, 1, 0)
  test_n$predicted_y_n <- predicted_y_n
  names(test_n)[1] <- "y"
  return(test_n)
}
```

Now we fit the various models and create datasets with predicted probabilities and predicted values.

```r
fit.train2 <- get.models(2)
train2 <- predicted_y_train(2, fit.train2)
test2 <- predicted_y_test(2, fit.train2)

fit.train4 <- get.models(4)
train4 <- predicted_y_train(4, fit.train4)
test4 <- predicted_y_test(4, fit.train4)

fit.train9 <- get.models(9)
train9 <- predicted_y_train(9, fit.train9)
test9 <- predicted_y_test(9, fit.train9)

fit.train16 <- get.models(16)
train16 <- predicted_y_train(16, fit.train16)
test16 <- predicted_y_test(16, fit.train16)

fit.train32 <- get.models(32)
train32 <- predicted_y_train(32, fit.train32)
test32 <- predicted_y_test(32, fit.train32)
```

```
fit.train64 <- get.models(64)
train64 <- predicted_y_train(64, fit.train64)
test64 <- predicted_y_test(64, fit.train64)

fit.train128 <- get.models(128)
train128 <- predicted_y_train(128, fit.train128)
test128 <- predicted_y_test(128, fit.train128)

fit.train256 <- get.models(256)
train256 <- predicted_y_train(256, fit.train128)
test256 <- predicted_y_test(256, fit.train128)
```

Now we make a list of the fitted models as well as of train vs. test datasets which contain y, predicted_y and probabilities i.e. the values we need for further calculations.

```
fitted_models <- list(fit.train2, fit.train4, fit.train9, fit.train16, fit.train32,
                      fit.train64, fit.train128, fit.train256)
train_datasets <- list(train2, train4, train9, train16, train32, train64, train128, train256)
test_datasets <- list(test2, test4, test9, test16, test32, test64, test128, test256)

misclassification_rates_train <- lapply(train_datasets, function(data_n)
    nrow(data_n[data_n$y != data_n$predicted_y,]) / nrow(data_n))
unlist(misclassification_rates_train)
```

```
## [1] 0.326 0.254 0.197 0.175 0.165 0.160 0.131 0.131
```

```
misclassification_rates_test <- lapply(test_datasets, function(data_n)
    nrow(data_n[data_n$y != data_n$predicted_y,]) / nrow(data_n))
unlist(misclassification_rates_test)
```

```
## [1] 0.3263598 0.2370990 0.1938633 0.1785216 0.1645746 0.1757322 0.1910739
## [8] 0.5453278
```

```
log_lik_train <- lapply(train_datasets, function(data_n)
    sum(data_n$y * log(data_n$probabilities_n) + (1 - data_n$y) *
    log(1-data_n$probabilities_n)))
unlist(log_lik_train)
```

```
## [1] -609.0962 -520.4964 -442.0999 -394.8035 -372.2838 -350.3744 -299.7019
## [8] -299.7019
```

```
log_lik_test <- lapply(test_datasets, function(data_n)
    sum(data_n$y * log(data_n$probabilities_n) + (1 - data_n$y) *
    log(1-data_n$probabilities_n)))
unlist(log_lik_test)
```

```
## [1] -439.7577 -357.8914 -305.1742 -285.9385 -282.7932 -293.4403 -364.9703
## [8] -819.5840
```

```r
#AIC based on log lik
AIC_train <- lapply(train_datasets, function(data_n) -2 * sum(data_n$y *
                    log(data_n$probabilities_n) + (1 - data_n$y) *
                    log(1-data_n$probabilities_n)) + 2 * (ncol(data_n)-2))
#ncol - 2 gives us the number of k (covariates + intercept)
unlist(AIC_train)
```

```
## [1] 1224.1923 1050.9929  904.1998  823.6069  810.5676  830.7488  857.4038
## [8] 1113.4038
```

```r
AIC_test <- lapply(test_datasets, function(data_n) -2 * sum(data_n$y *
                    log(data_n$probabilities_n) + (1 - data_n$y) *
                        log(1-data_n$probabilities_n)) + 2 * (ncol(data_n)-2))
#ncol - 2 gives us the number of k (covariates + intercept)
unlist(AIC_test)
```

```
## [1]  885.5154  725.7827  630.3483  605.8770  631.5864  716.8805  987.9405
## [8] 2153.1680
```

```r
#AIC (for the models estimated with train data) can also be based on deviance and
#the estimated error sigma.hat.sq
deviance_train <- lapply(fitted_models, function (fitted_model_n)
    sum(abs((fitted_model_n$y - round(fitted_model_n$fitted.values))))/1000)#n = 1000
sigma.hat.sq <- sum((y_train - mean(y_train))^2)/(1000-c(2,4,9,16,32,64,128,256))

AIC_2 <- unlist(deviance_train) + 2 * c(2,4,9,16,32,64,128,256) * sigma.hat.sq/1000
AIC_2
```

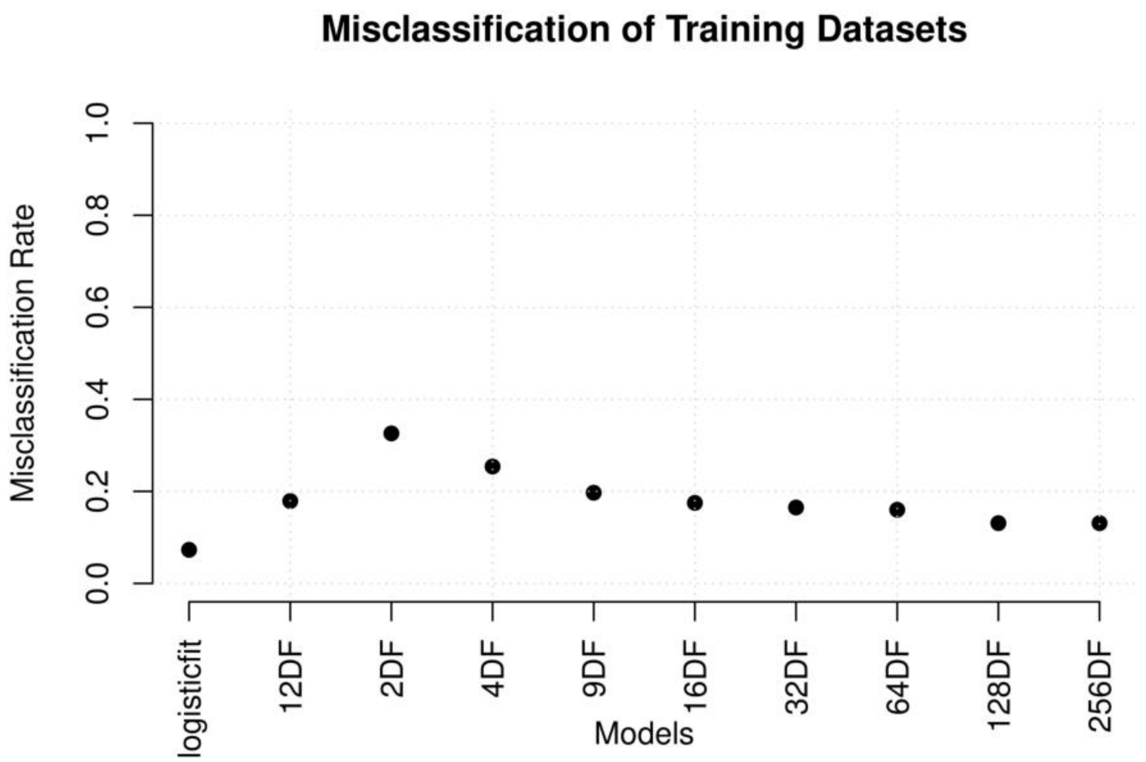
```
## [1] 0.3269561 0.2559161 0.2013329 0.1827578 0.1807720 0.1926224 0.2010333
## [8] 0.2341641
```

Now we visualise these results (misclassification rate, log likelihood and AICs) for the various models. First, we still have to calculate the AICs for the unrestricted model and the one with 12 df.

```r
AIC_logfit_train <- -2 * sum(train$y * log(probabilities_train) +
                    (1 - train$y) * log(1-probabilities_train)) + 2 * 257
AIC_df12_train <- -2 * sum(train12$y * log(probabilities_train12) +
                    (1 - train12$y) * log(1-probabilities_train12)) + 2 * 13
AIC_logfit_test <- -2 * sum(test$y * log(probabilities_test) +
                    (1 - test$y) * log(1-probabilities_test)) + 2 * 257
AIC_df12_test <- -2 * sum(test12$y * log(probabilities_test12) + (1 - test12$y)
                    * log(1-probabilities_test12)) + 2 * 13

AIC_2_logfit_train <-
    sum(abs((fit.train$y - round(fit.train$fitted.values))))/1000 + 2 * 256 *
    sum((y_train - mean(y_train))^2)/(1000-256)/1000

AIC_2_df12_train <-
    sum(abs((fit.train12$y - round(fit.train12$fitted.values))))/1000 + 2 *
    12 * sum((y_train - mean(y_train))^2)/(1000-12)/1000
```

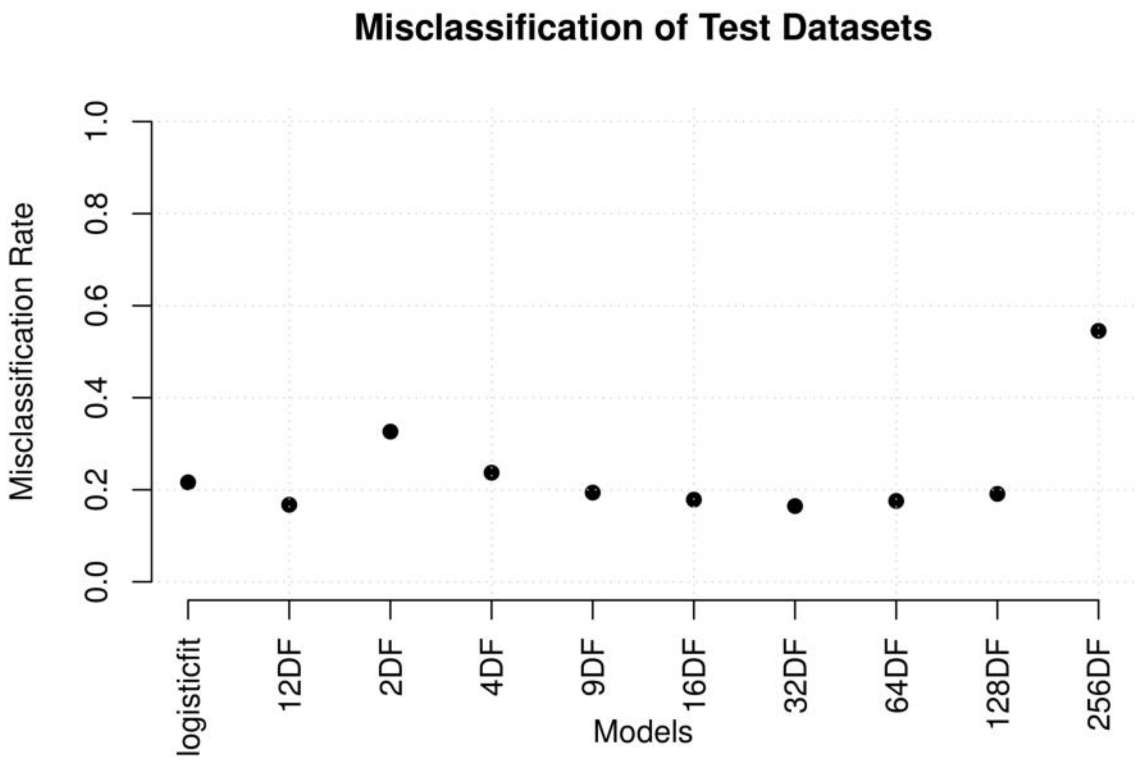
Now we build a dataframe of all the models and their respective rates.

```r
summary_train <-
    as.data.frame(cbind(c(misclassification_train, misclassification_train12,
    unlist(misclassification_rates_train)),
    c(loglike_train, log_like_train12, unlist(log_lik_train)),
    c(unlist(AIC_logfit_train), unlist(AIC_df12_train), unlist(AIC_train)),
    c(AIC_2_logfit_train, AIC_2_df12_train, AIC_2)))

rownames(summary_train) <- c("logisticfit", "12DF", "2DF", "4DF", "9DF", "16DF",
                             "32DF", "64DF", "128DF", "256DF")
colnames(summary_train) <- c("Misclassification", "Log_Likelihood", "AIC",
                             "AIC version 2")

plot(summary_train$Misclassification, main = "Misclassification of Training Datasets",
     xaxt = "n", xlab = "Models", ylab = "Misclassification Rate", ylim = c(0, 1),
     bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_train), las = 2)
```



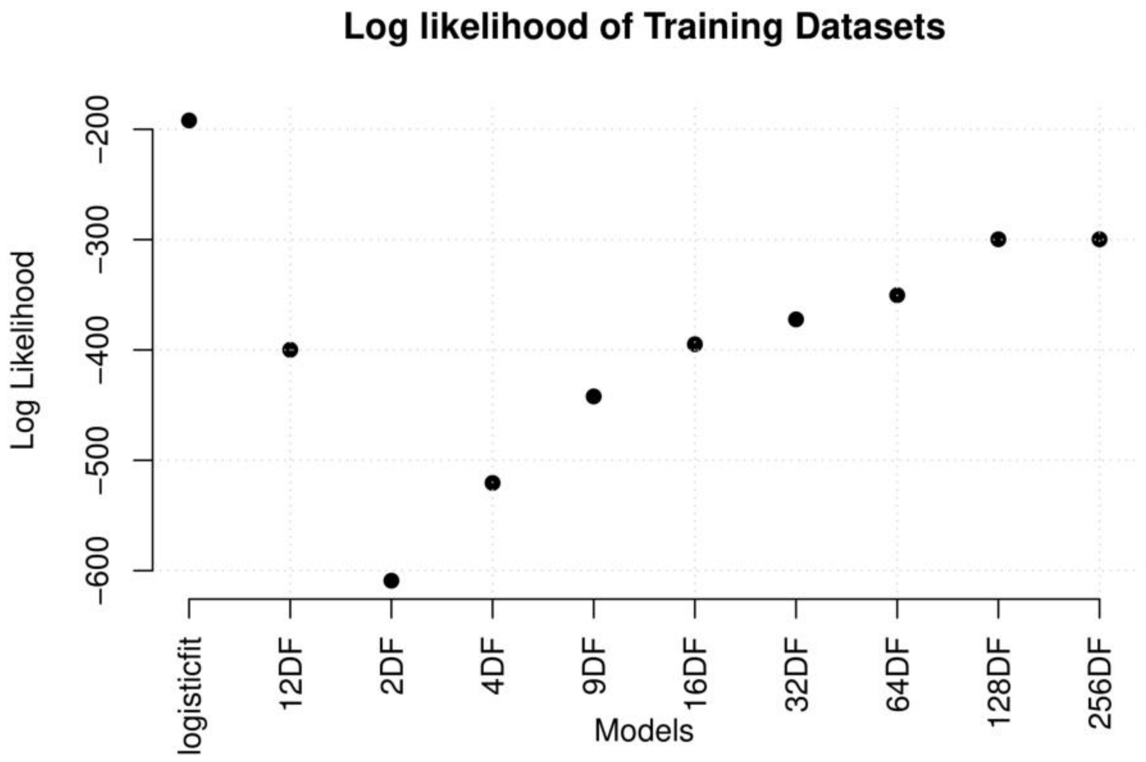
The higher the smooth of the covariate matrix, the higher the misclassification rate for the training data. Higher smooths do not account so much for the specifities of the respective training dataset and therefore might avoid overfitting. We can compare the above values to the ones of the test dataset:

```r
summary_test <-
    as.data.frame(cbind(c(misclassification_test, misclassification_test12,
    unlist(misclassification_rates_test)),
    c(loglike_test, log_like_test12, unlist(log_lik_test)),
    c(unlist(AIC_logfit_test), unlist(AIC_df12_test), unlist(AIC_test))))


rownames(summary_test) <- c("logisticfit", "12DF", "2DF", "4DF", "9DF", "16DF",
                            "32DF", "64DF", "128DF", "256DF")
colnames(summary_test) <- c("Misclassification", "Log_Likelihood", "AIC")

plot(summary_test$Misclassification, main = "Misclassification of Test Datasets",
     xaxt = "n", xlab = "Models", ylab = "Misclassification Rate", ylim = c(0, 1),
     bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_test), las = 2)
```

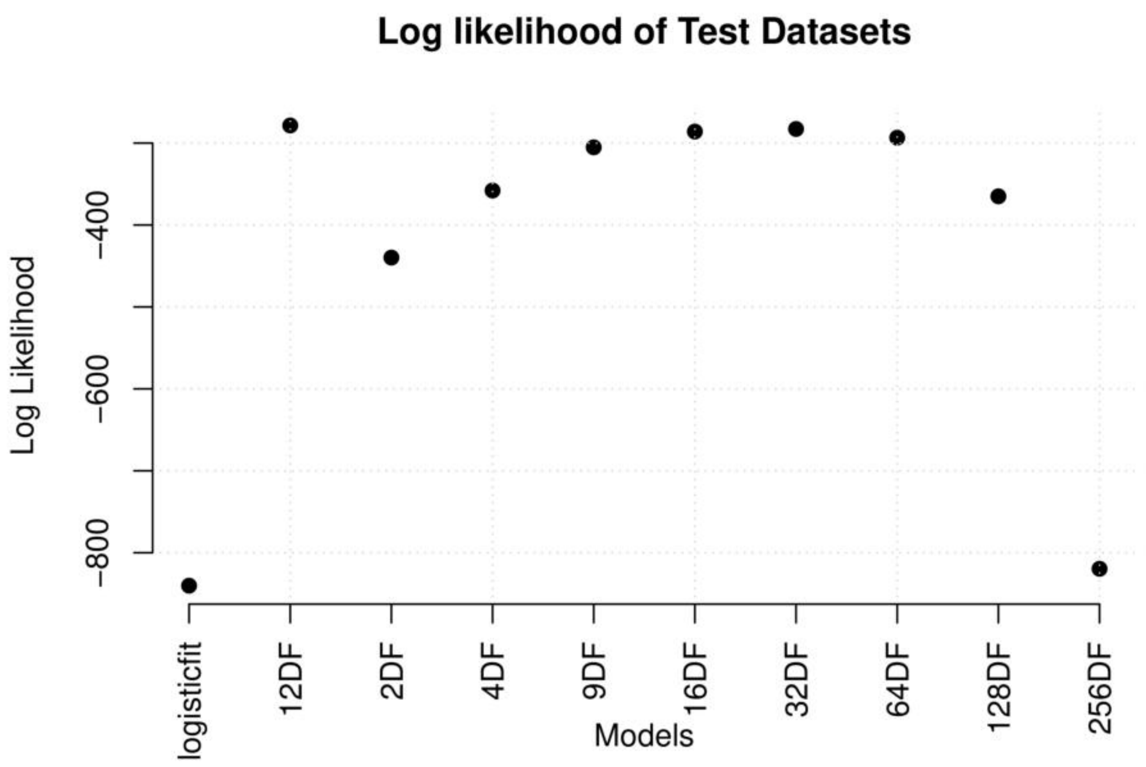


For the test dataset, we see a better fit for the smoothed covariate matrix – at least for those with more than 4 degrees of freedom. We can assume that a certain smooth avoids overfitting and therefore fits the test data better than the model without any smoothing. However, very high smooths (2 df, 4 df) apparently lose important information of the data and therefore lead to worse misclassification rates.

```
plot(summary_train$Log_Likelihood, main = "Log likelihood of Training Datasets",
     xaxt = "n", xlab = "Models", ylab = "Log Likelihood", bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_train), las = 2)
```

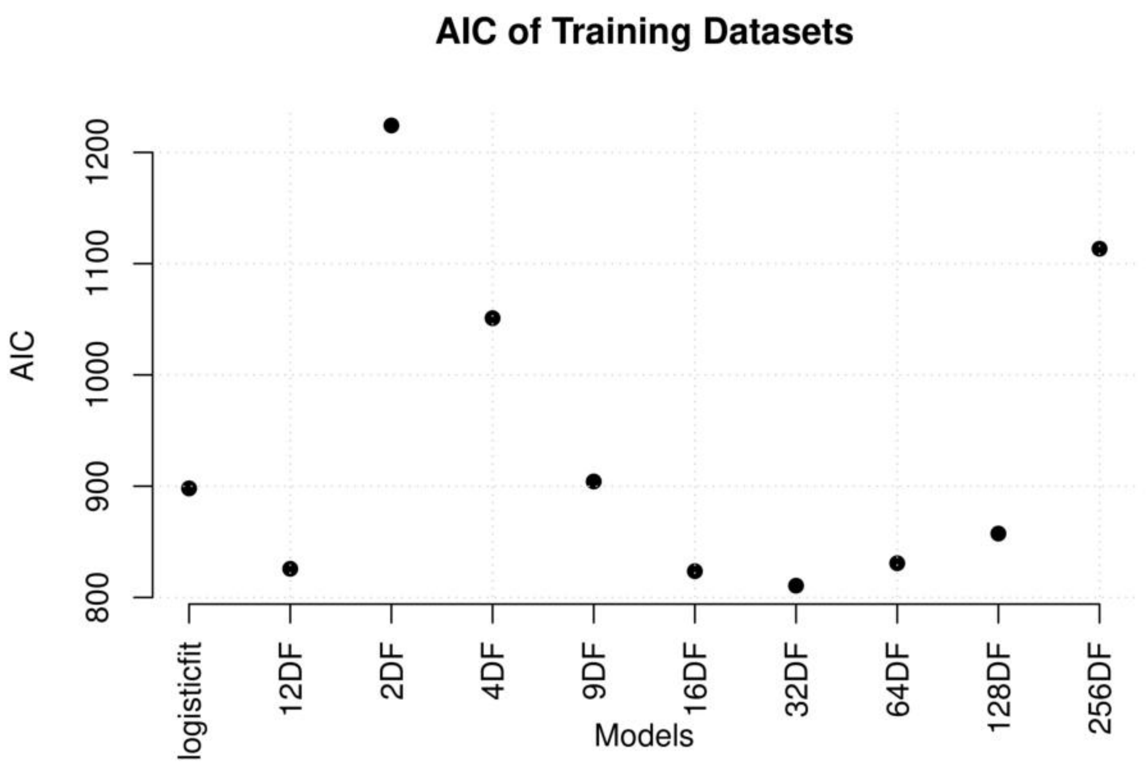


```r
plot(summary_test$Log_Likelihood, main = "Log likelihood of Test Datasets",
     xaxt = "n", xlab = "Models", ylab = "Log Likelihood", bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_test), las = 2)
```

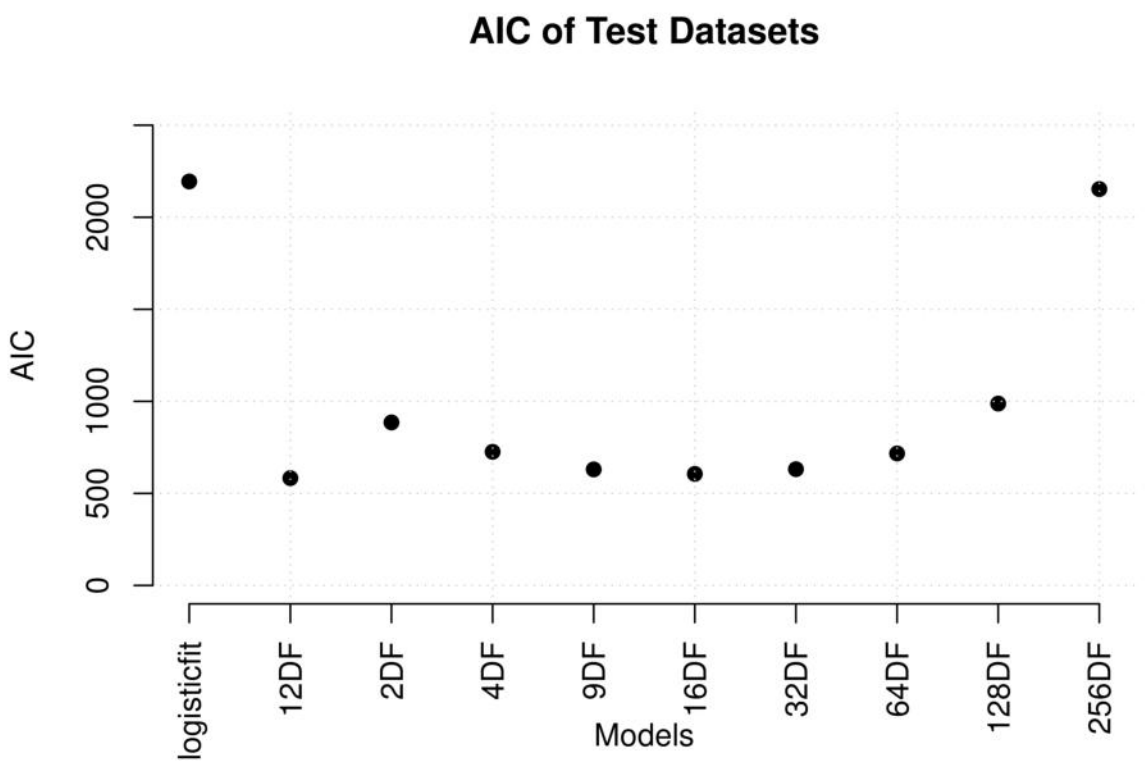


For the training dataset, the log likelihood gets worse the higher the smooth of the covariate matrix. This does not come as a surprise, since the smooth does not cover as many specifities of the respective dataset. Looking at the test dataset, we see a different picture: a certain smooth improves the log likelihood (because a possible overfitting to the training dataset was avoided). However, very high smooths (2 df, 4 df) again deteriorate the log likelihood.

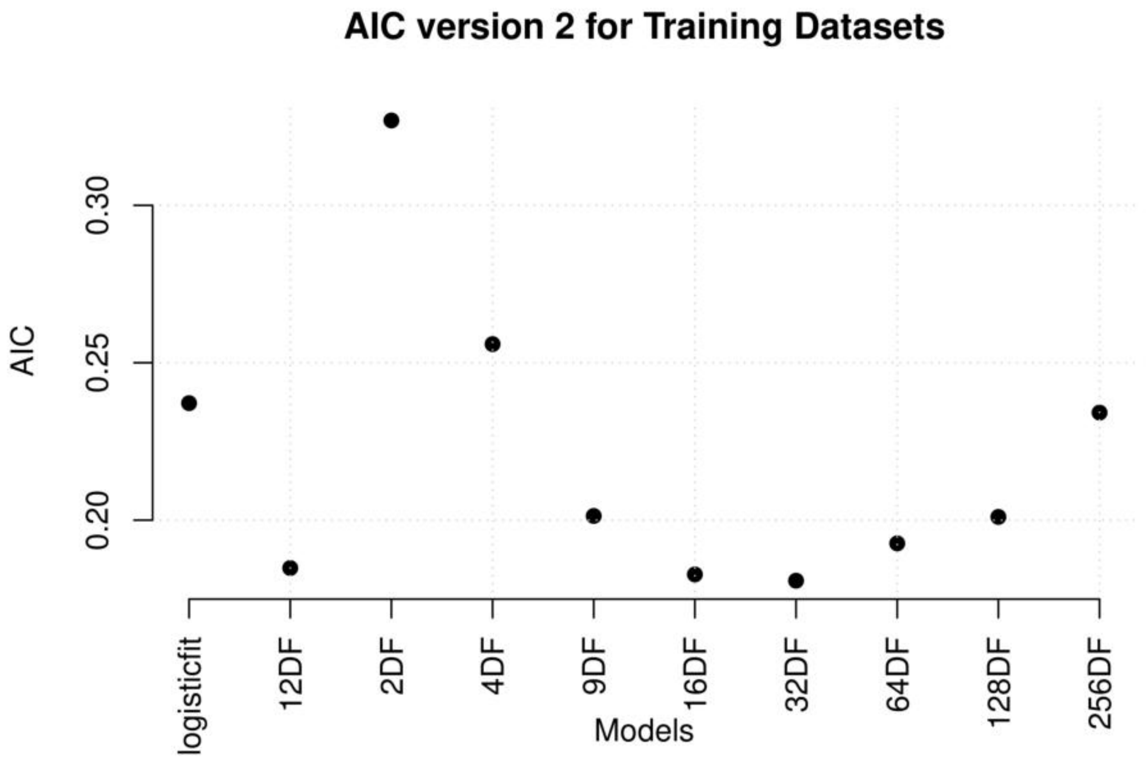
```
plot(summary_train$AIC, main = "AIC of Training Datasets", xlab = "Models",
     xaxt = "n", ylab = "AIC", bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_train), las = 2)
```



```
plot(summary_test$AIC, main = "AIC of Test Datasets", xlab = "Models",
     xaxt = "n", ylim = c(0, 2500), ylab = "AIC", bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_test), las = 2)
```



```
plot(summary_train$`AIC version 2`, main = "AIC version 2 for Training Datasets",
     xlab = "Models", xaxt = "n", ylab = "AIC", bty = "n", pch = 19)
grid()
axis(1, at = 1:10, labels=rownames(summary_train), las = 2)
```



The AIC tries to balance a possible lack of fit and high model complexity by taking into account both – the degrees of freedom used (with a penalty for highly complex models) and the log likelihood (which is usually better for complex models). For the training dataset, we see the best AIC values for the 12-64 df models. These seem to be a kind of compromise between the unrestricted, highly complex model (logisticfit) and the models with a high smooth (2 df, 4 df). For the test datasets, we also have the best AIC for the models with a certain but not too high smooth, i.e. those with 12-32 df. AIC version 2 shows a similar picture for the training data, only on another scale.