eNote 9

Discriminant Analysis: LDA, QDA, k-NN, Bayes, PLSDA, cart, Random forests

Indhold

9.1 Reading material

9.1.1 LDA, QDA, k-NN, Bayes

Read in the Varmuza book: (not covering CARTS and random forests)

- Section 5.1, Intro, 2.5 pages
- Section 5.2, Linear Methods, 12 pages
- Section 5.3.3, Nearest Neighbourg (k-NN), 3 pages
- Section 5.7, Evaluation of classification, 3 pages

Alternatively read in the Wehrens book: (not covering CARTS and random forests)

- 7.1 Discriminant Analysis 104
  - 7.1.1 Linear Discriminant Analysis 105
  - 7.1.2 Crossvalidation 109
  - 7.1.3 Fisher LDA 111
  - 7.1.4 Quadratic Discriminant Analysis 114
  - 7.1.5 Model-Based Discriminant Analysis 116
  - 7.1.6 Regularized Forms of Discriminant Analysis 118
- 7.2 Nearest-Neighbour Approaches 122

- 11.3 Discrimination with Fat Data Matrices 243
  - 11.3.1 PCDA 244
  - 11.3.2 PLSDA 248

9.1.2 Classification trees (CART) and random forests

Read in the Varmuza book about classification (and regression) trees:

- Section 5.4 Classification Trees
- Section 5.8.1.5 Classification Trees
- (Section 4.8.3.3 Regression Trees)

Read in the Wehrens book:

- 7.3 Tree-Based Approaches 126-135
- 9.7 Integrated Modelling and Validation 195
  - (9.7.1 Bagging 196)
  - 9.7.2 Random Forests 197
  - (9.7.3 Boosting 202)

9.2 Example: Iris data

```
# we use the iris data:
data(iris3)
#library(MASS)

Iris <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
                   Sp = rep(c("s","c","v"), rep(50,3)))

# We make a test and training data:
set.seed(4897)
train <- sample(1:150, 75)
test <- (1:150)[-train]
Iris_train <- Iris[train,]
Iris_test <- Iris[test,]

# Distribution in three classes in training data:
table(Iris_train$Sp)
```

```
 c  s  v
23 24 28
```

9.2.1 Linear Discriminant Analysis, LDA

We use the `lda` function from the MASS package:

```
# PART 1: LDA
library(MASS)
lda_train_LOO <- lda(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
                     Iris_train, prior = c(1, 1, 1)/3, CV = TRUE)
```

The Species factor variable is expressed as the response in a usual model expression with the four measurement variables as the x's. The `CV=TRUE` option choice performs full LOO cross validation. The `prior` option works as:

the prior probabilities of class membership. If unspecified, the class proportions for the training set are used. If present, the probabilities should be specified in the order of the factor levels.

First we must assess the accuracy of the prediction based on the cross validation error, which is quantified simply as relative frequencies of errornous class predictions, either in total or detailed on the classes:

```
# Assess the accuracy of the prediction
# percent correct for each category:
ct <- table(Iris_train$Sp, lda_train_LOO$class)
diag(prop.table(ct, 1))
```

```
        c         s         v
0.9130435 1.0000000 1.0000000
```

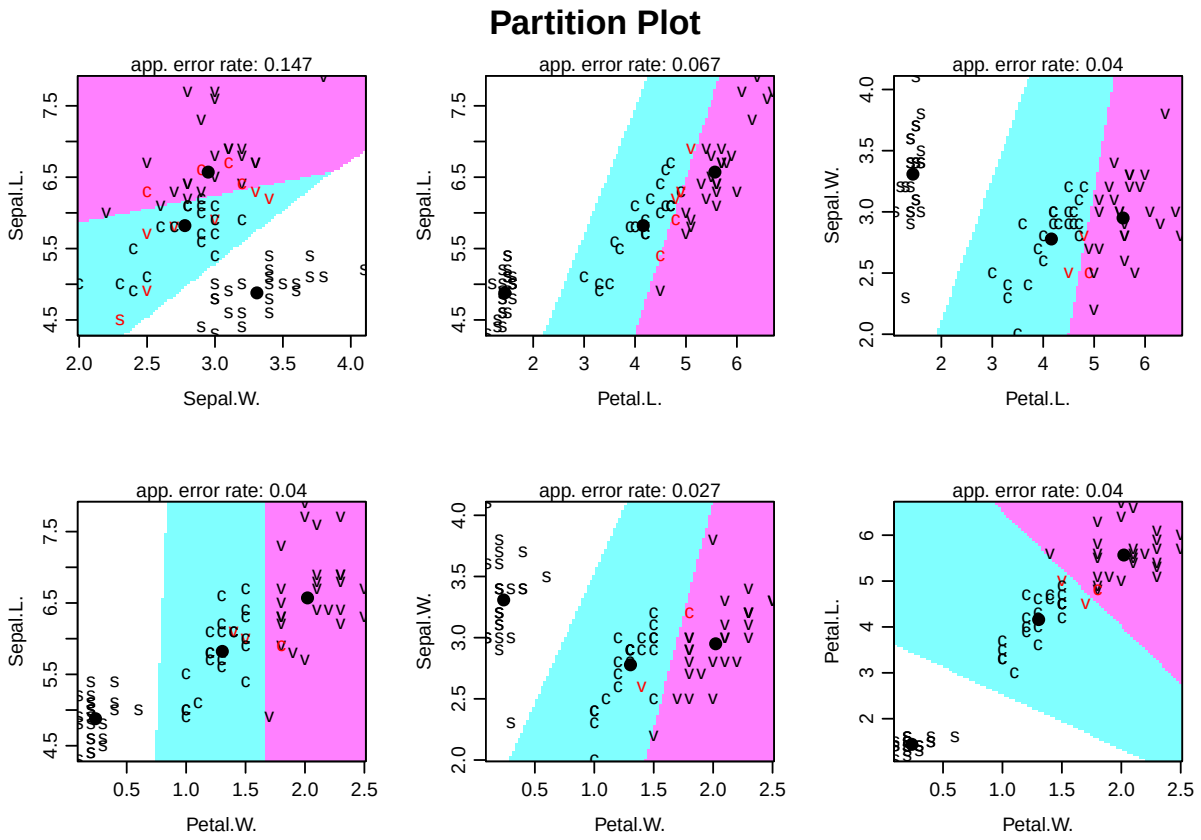
```
# total percent correct
sum(diag(prop.table(ct)))
```

```
[1] 0.9733333
```

So the overal CV based error rate is $0.0267 = 2.7\%$.

Som nice plotting only works without the CV-stuff using the klaR-package:

```
library(klaR)
partimat(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
         data = Iris_train, method = "lda", prior=c(1, 1, 1)/3)
```



9.2.2 Quadratic Discriminant Analysis, QDA

It goes very much like above:

```
# PART 2: QDA
# Most stuff from LDA can be reused:

qda_train_LOO <- qda(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
                     Iris_train, prior = c(1, 1, 1)/3, CV = TRUE)

# Assess the accuracy of the prediction
# percent correct for each category:
ct <- table(Iris_train$Sp, qda_train_LOO$class)
ct
```

```
    c  s  v
  c 21  0  2
  s  0 24  0
  v  1  0 27
```

```
diag(prop.table(ct, 1))
```

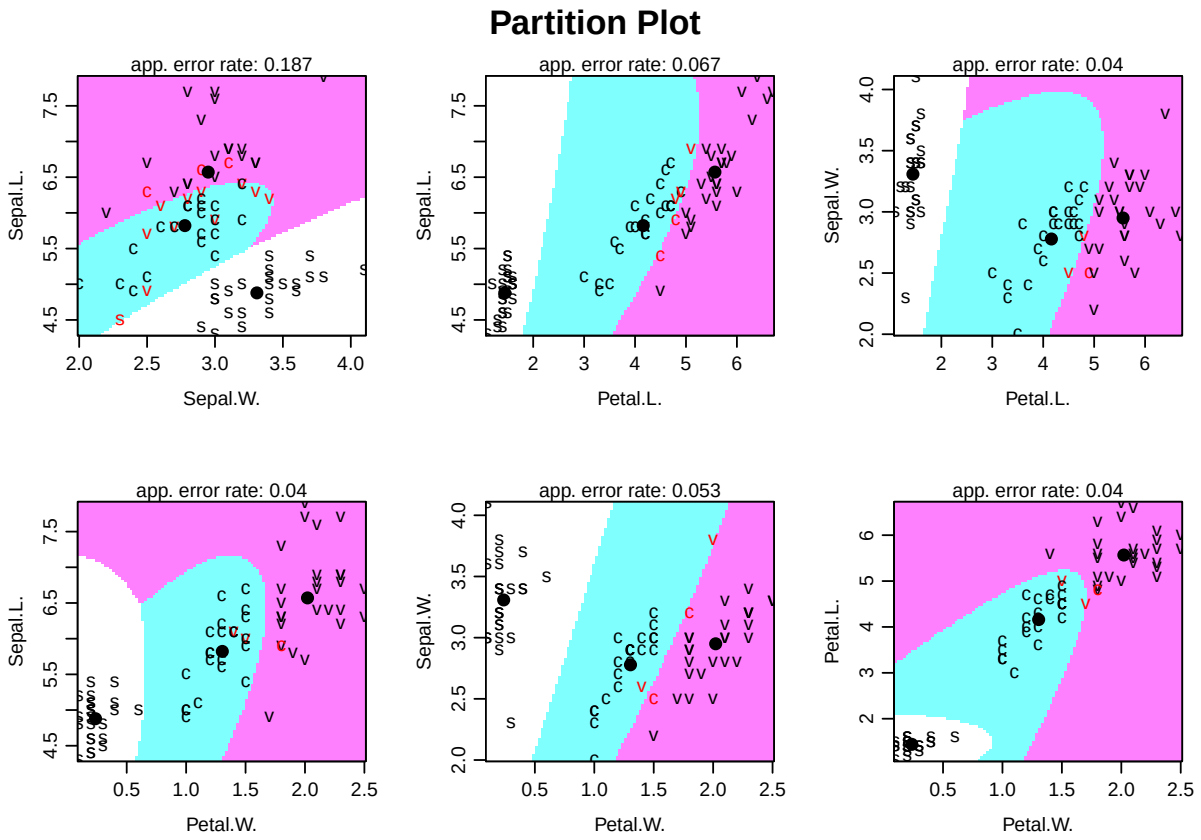
```
        c         s         v
0.9130435 1.0000000 0.9642857
```

```
# total percent correct
sum(diag(prop.table(ct)))
```

```
[1] 0.96
```

For this example the QDA performs slightly worse than the LDA.

```
partimat(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
         data = Iris_train, method = "qda", prior = c(1, 1, 1)/3)
```



9.2.3 Predicting NEW data

```
# And now predicting NEW data:
predict(qda_train, Iris_test)$class
```

```
Error in predict(qda_train, Iris_test): object 'qda_train' not found
```

```
# Find confusion table:
ct <- table(Iris_test$Sp, predict(qda_train, Iris_test)$class)
```

```
Error in predict(qda_train, Iris_test): object 'qda_train' not found
```

```
ct
```

```
    c  s  v
  c 21  0  2
  s  0 24  0
  v  1  0 27
```

```
diag(prop.table(ct, 1))
```

```
        c         s         v
0.9130435 1.0000000 0.9642857
```

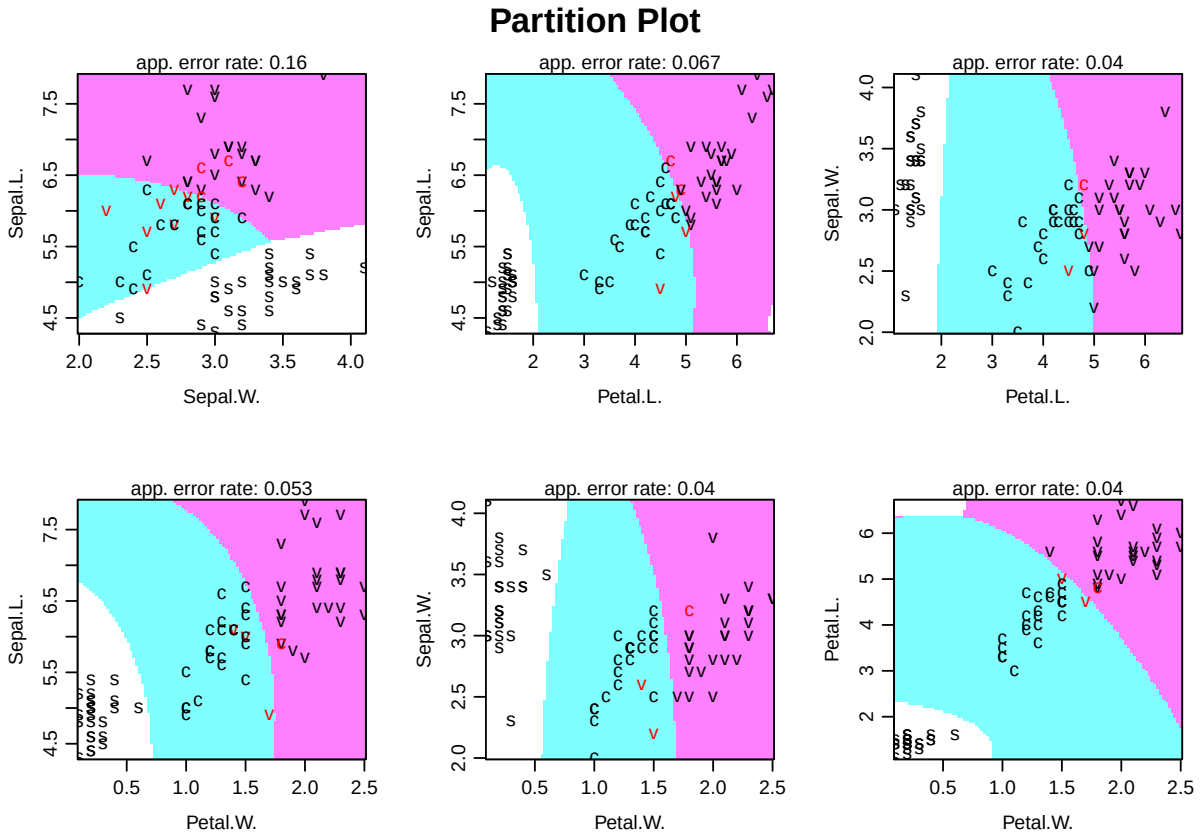
```
# total percent correct
sum(diag(prop.table(ct)))
```

```
[1] 0.96
```

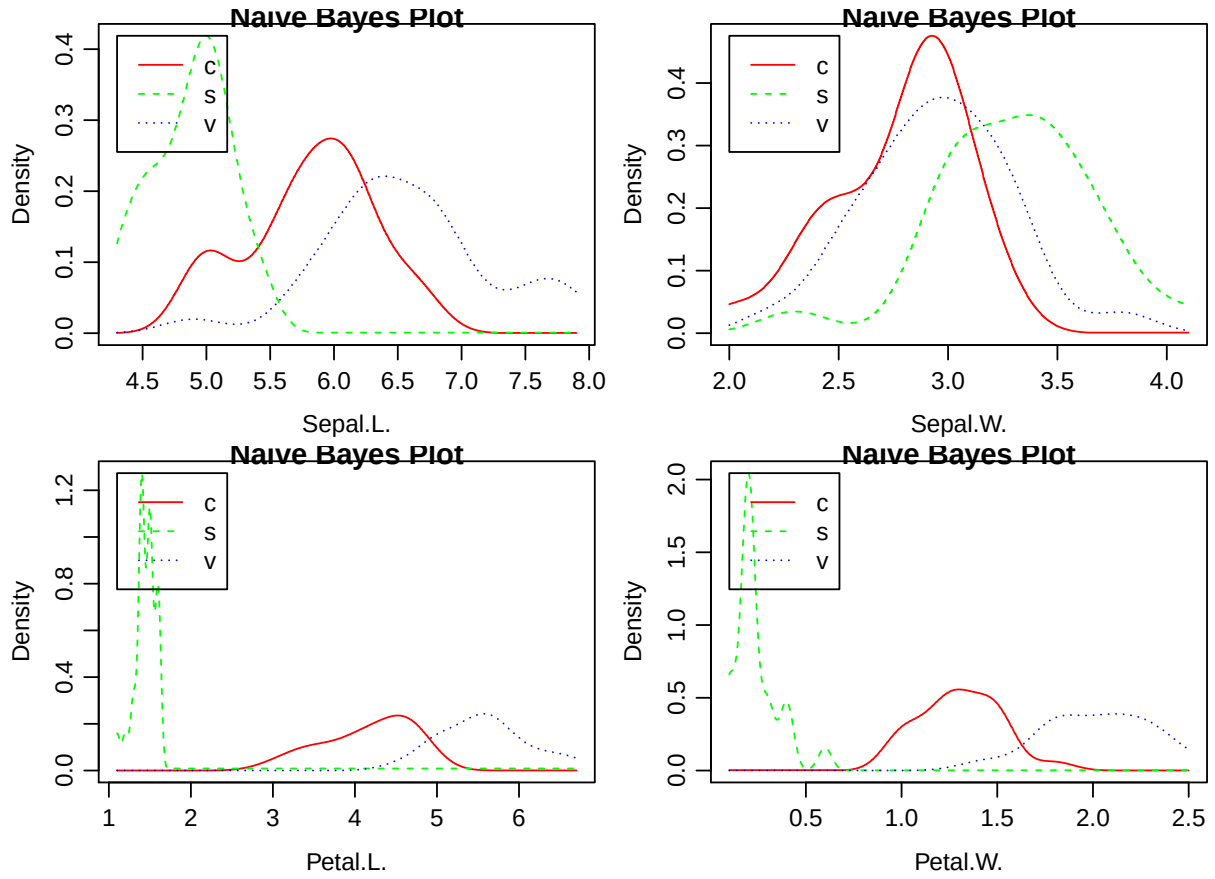
9.2.4 Bayes method

We can fit a density to the observed data and use that instead of the normal distributions implicitly used in LDA:

```
# Part 3: (Naive) Bayes method
# With "usekernel=TRUE" it fits a density to the observed data and uses that
# instead of the normal distribution

# Let's explore the results first:
# (Can take a while dur to the fitting of densities)
partimat(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
         data = Iris_train, method = "naiveBayes",
         prior=c(1, 1, 1)/3, usekernel = TRUE)
```




```
bayes_fit <- suppressWarnings(NaiveBayes(Sp ~ Sepal.L. + Sepal.W. +
                                           Petal.L. + Petal.W.,
                        data = Iris_train, method = "naiveBayes",
                        prior = c(1, 1, 1)/3, usekernel = TRUE))
par(mfrow = c(2, 2))
plot(bayes_fit)
```




```
# And now predicting NEW data:
bayespred <- predict(bayes_fit, Iris_test)$class

# Find confusion table:
ct <- table(Iris_test$Sp, bayespred)
ct
```

```
   bayespred
     c  s  v
  c 25  0  2
  s  0 26  0
  v  1  0 21
```

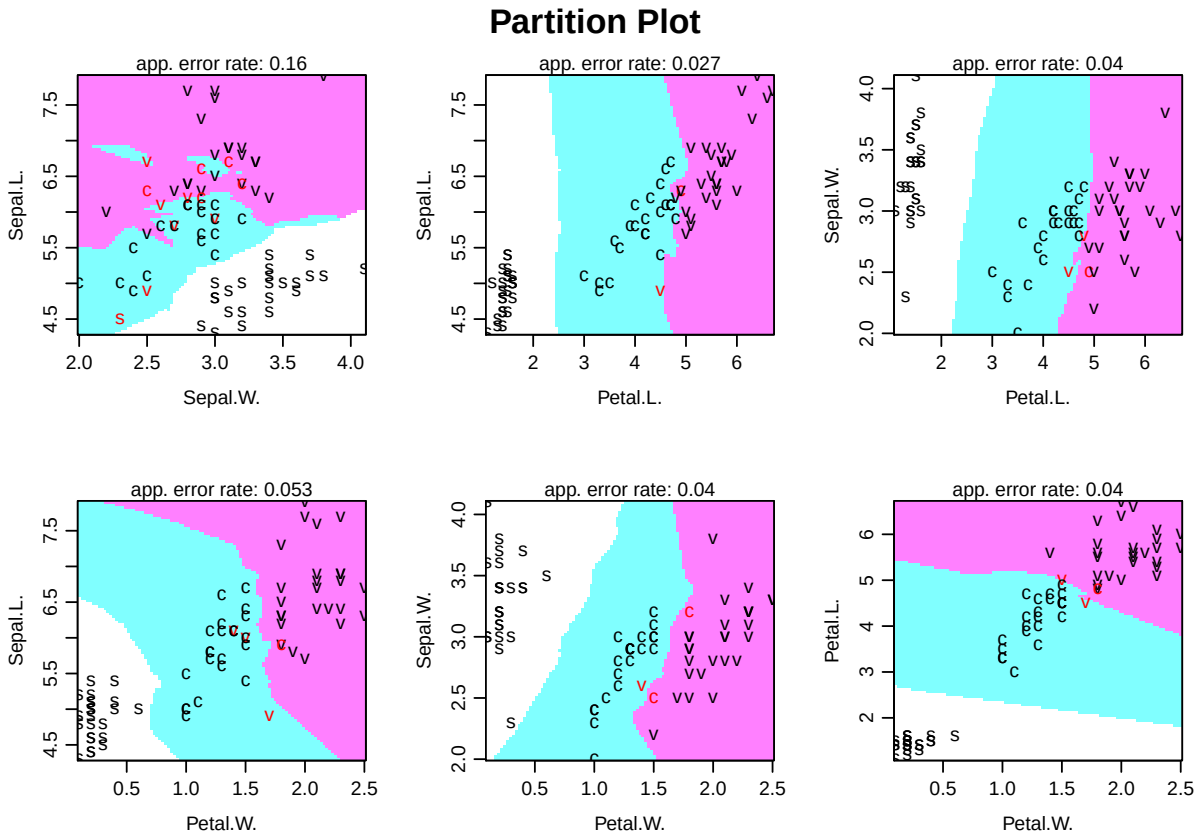
```
diag(prop.table(ct, 1))
```

```
        c         s         v
0.9259259 1.0000000 0.9545455
```

```
# total percent correct
sum(diag(prop.table(ct)))


[1] 0.96
```

9.2.5 k-nearest neighbourgh

```
# PART 4: k-nearest neighbourgh:

# Explorative plot WARNING: THIS may take some time to produce!!
partimat(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
         data = Iris[train,], method = "sknn", kn = 3)
```



```
knn_fit_5 <- sknn(Sp ~ Sepal.L. + Sepal.W. + Petal.L. + Petal.W.,
                  data = Iris_train, method = "sknn", kn = 5)

# And now predicting NEW data:
knn_5_preds <- predict(knn_fit_5, Iris_test)$class

# Find confusion table:
ct <- table(Iris_test$Sp, knn_5_preds)
ct
```

```
   knn_5_preds
     c  s  v
  c 25  0  2
  s  0 26  0
  v  0  0 22
```

```
diag(prop.table(ct, 1))
```

```
        c         s         v
0.9259259 1.0000000 1.0000000
```

```
# total percent correct
sum(diag(prop.table(ct)))
```

```
[1] 0.9733333
```

9.2.6 PLS-DA

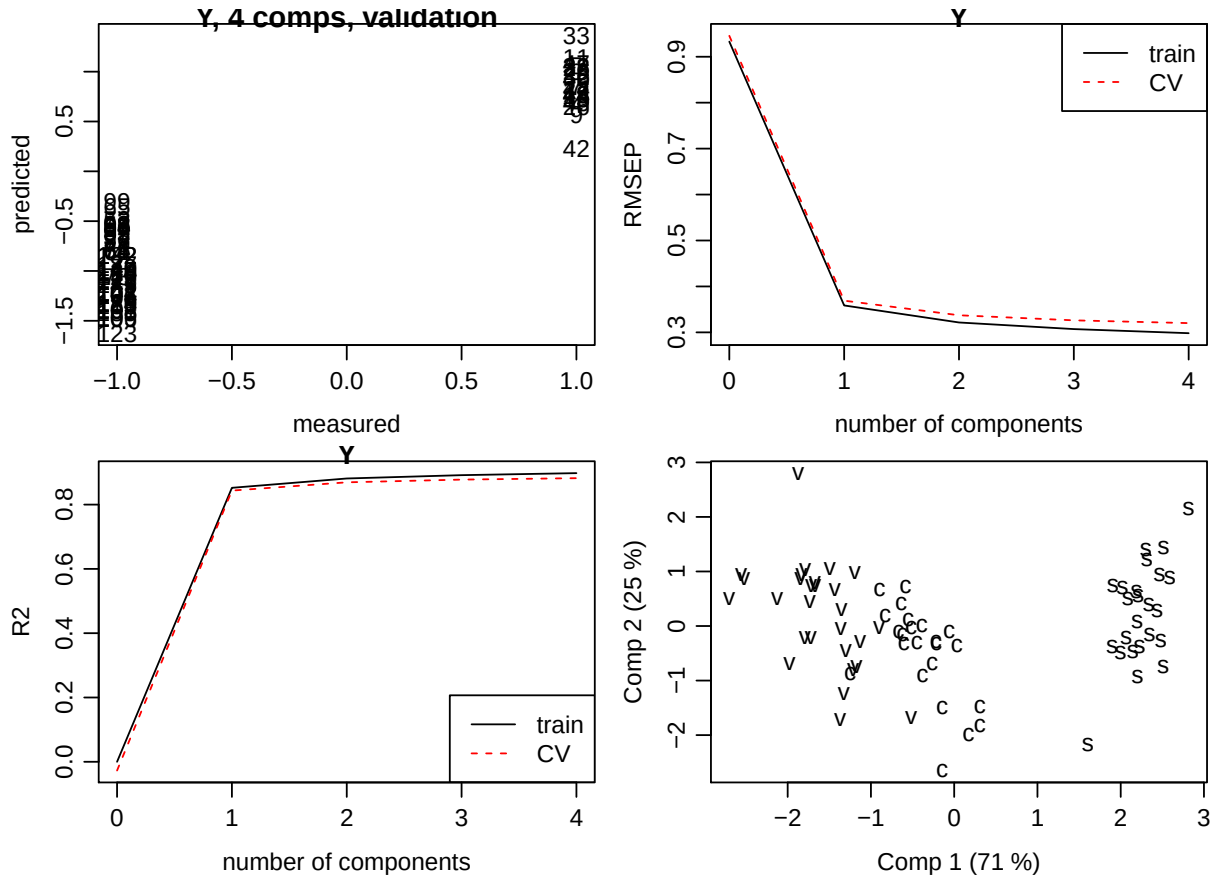
```
# PART 5: PLS-DA
# We have to use the "usual" PLS-functions

# Define the response vector (2 classes) OR matrix (>2) classes:
```

```r
# Let's try with K=2: Group 1: s, Group -1: c and v
Iris_train$Y <- -1
Iris_train$Y[Iris_train$Sp == "s"] <- 1
table(Iris_train$Y)
```

```
-1  1
51 24
```

```r
Iris_train$X <- as.matrix(Iris_train[, 1:4])
```

```r
# From here use standard PLS1 predictions, e.g.:
library(pls)
# Pls:

mod_pls <- plsr(Y ~X , ncomp = 4, data =Iris_train,
                validation = "LOO", scale = TRUE, jackknife = TRUE)

# Initial set of plots:
par(mfrow = c(2, 2))
plot(mod_pls, labels = rownames(Iris_train), which = "validation")
plot(mod_pls, "validation", estimate = c("train", "CV"),
     legendpos = "topright")
plot(mod_pls, "validation", estimate = c("train", "CV"),
     val.type = "R2", legendpos = "bottomright")
pls::scoreplot(mod_pls, labels = Iris_train$Sp)
```



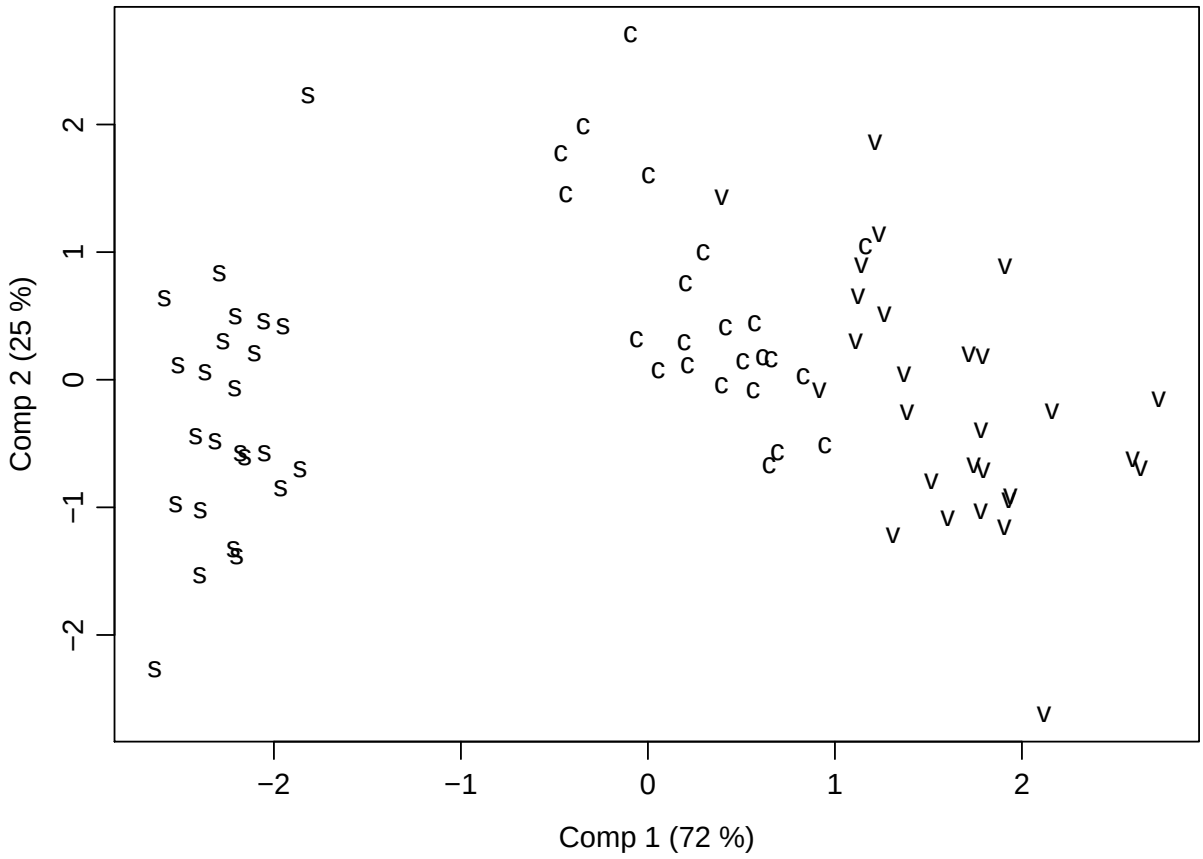
Be carefull about interpretations due to the binary setting You should do a CV based confusion table for each component, really:

```
preds <- array(dim = c(length(Iris_train[, 1]), 4))

for (i in 1:4) preds[, i] <- predict(mod_pls, ncomp = i)
preds[preds<0] <- -1
preds[preds>0] <- 1

# Look at the results from each of the components:

for (i in 1:4) {
ct <- table(Iris_train$Y, preds[,1])
CV_error <- 1-sum(diag(prop.table(ct)))
print(CV_error)
}


[1] 0
```

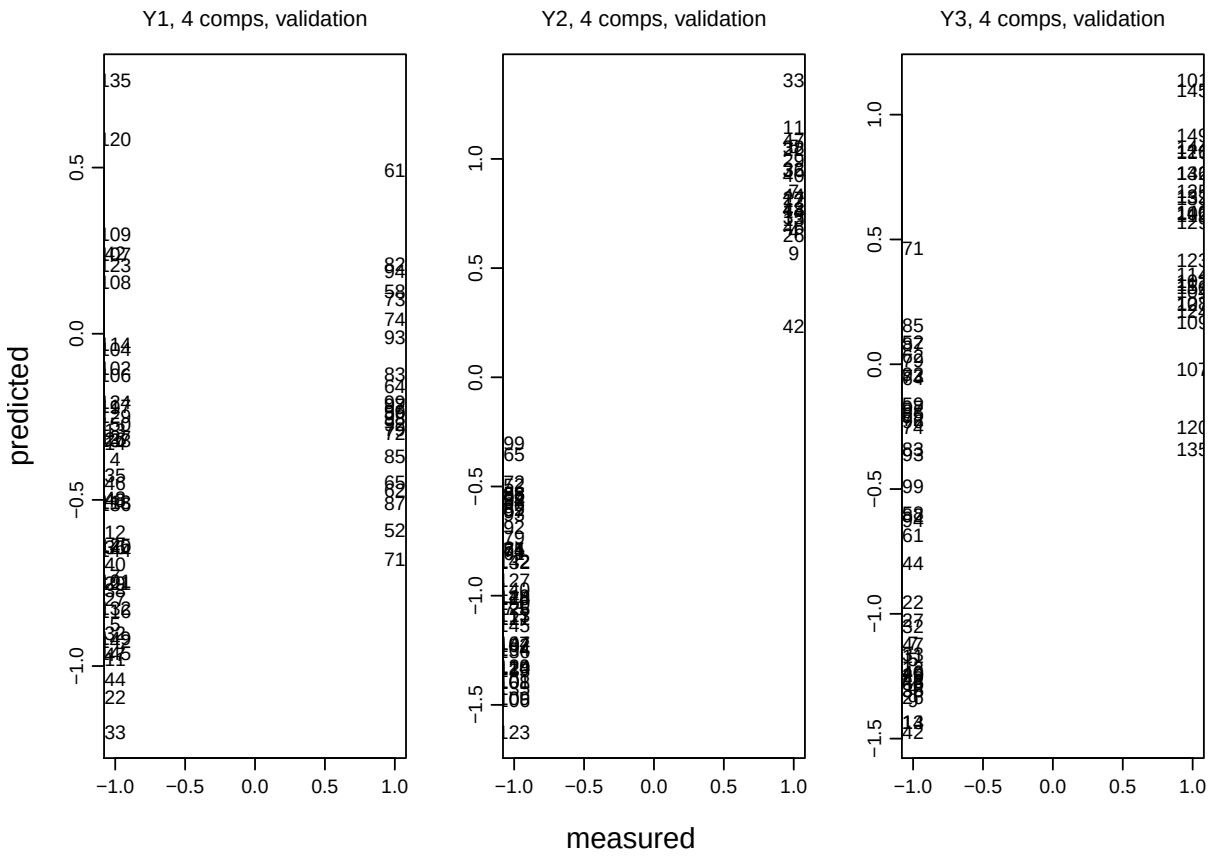
```
[1] 0
[1] 0
[1] 0
```

The prediction of new data would be handled similarly (not shown).

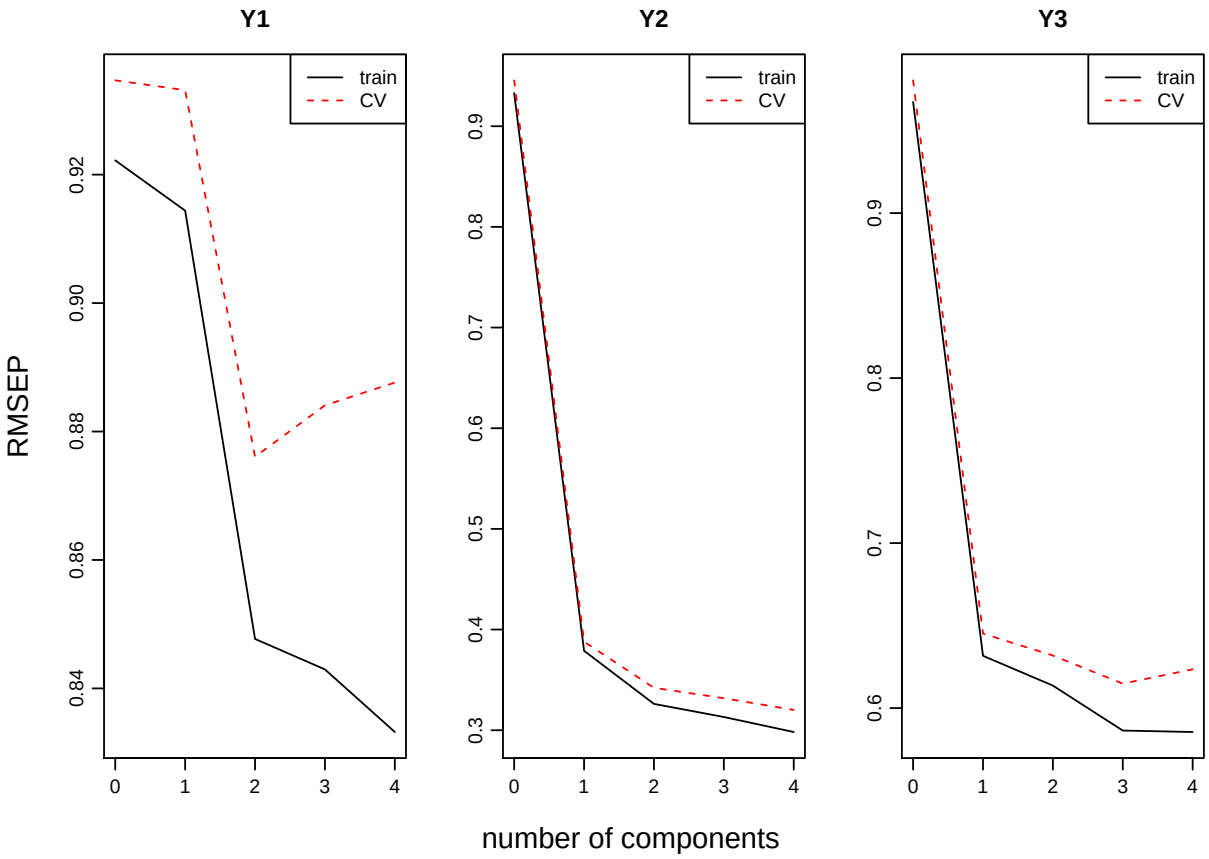
```
# WHAT if in this case K=3 classes:
# Look at Varmuza, Sec. 5.2.2.3
K=3
Iris_train$Y=matrix(rep(1,length(Iris_train[,1])*K),ncol=K)

Iris_train$Y[Iris_train$Sp!="c",1]=-1
Iris_train$Y[Iris_train$Sp!="s",2]=-1
Iris_train$Y[Iris_train$Sp!="v",3]=-1

mod_pls <- plsr(Y ~ X, ncomp = 4, data = Iris_train,
                validation="LOO", scale = TRUE, jackknife = TRUE)

# Initial set of plots:
par(mfrow=c(1, 1))
pls::scoreplot(mod_pls, labels = Iris_train$Sp)
```



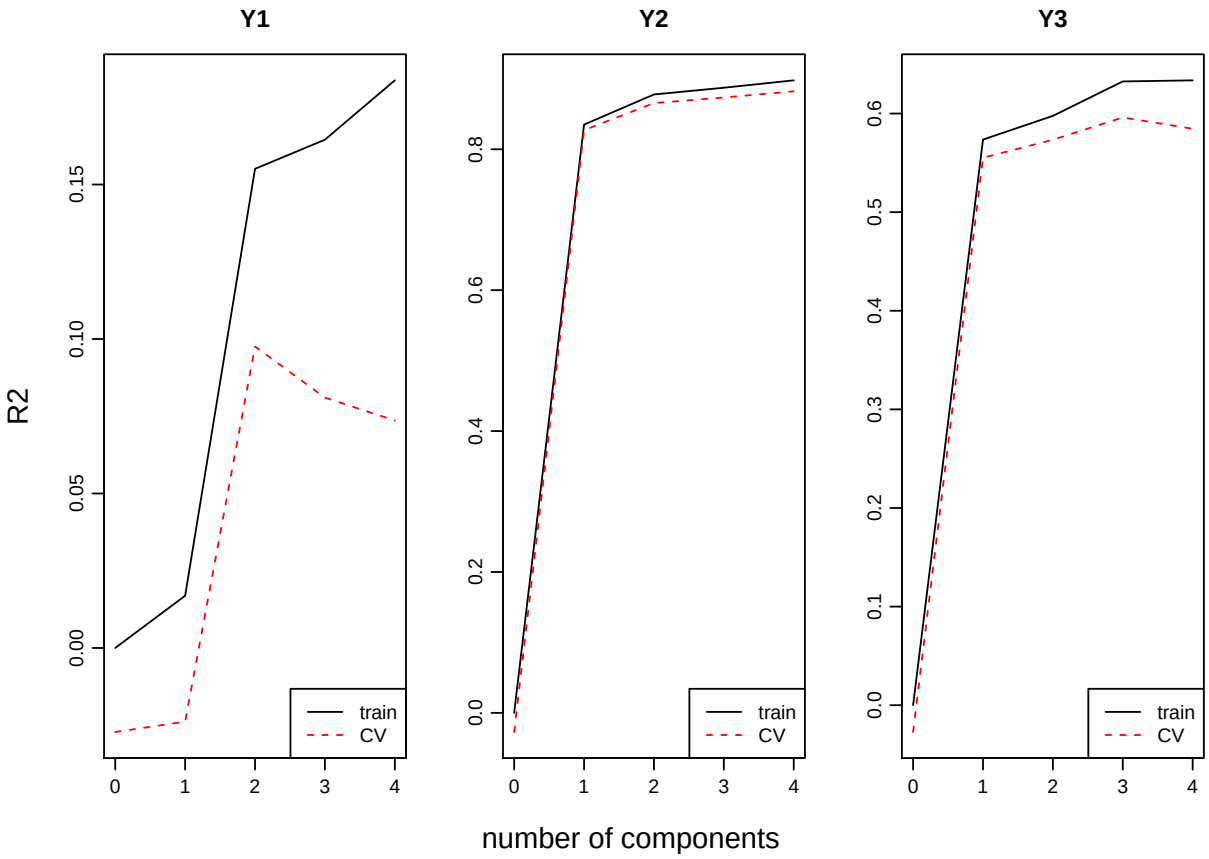

```
plot(mod_pls, labels = rownames(Iris_train), which = "validation")
```




```
plot(mod_pls, "validation", estimate = c("train", "CV"),
     legendpos = "topright")
```




```
plot(mod_pls, "validation", estimate = c("train", "CV"),
     val.type = "R2", legendpos = "bottomright")
```




```r
# Predictions from PLS need to be transformed to actual classifications:
# Select the largest one:
preds <- array(dim = c(length(Iris_train[, 1]), 4))
for (i in 1:4) preds[,i]<- apply(predict(mod_pls, ncomp = i),
                                 1, which.max)

preds2 <- array(dim = c(length(Iris_train[, 1]), 4))
preds2[preds==1] <- "c"
preds2[preds==2] <- "s"
preds2[preds==3] <- "v"

# Look at the results from each of the components:
for (i in 1:4) {
  ct <- table(Iris_train$Sp, preds2[, i])
  CV_error <- 1 - sum(diag(prop.table(ct)))
  print(CV_error)
}
```

```
[1] 0.96
[1] 0.24
[1] 0.2
[1] 0.1733333
```

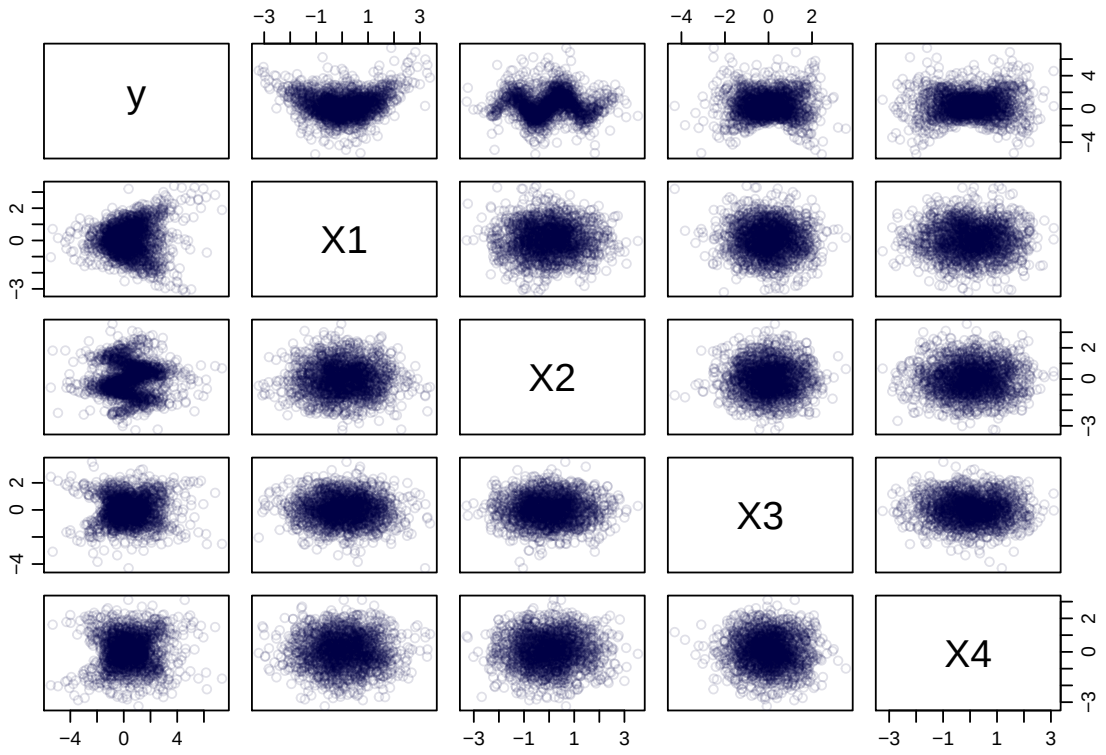
9.2.7 Random forests

Wellcome to examples of randomForest and forestFloor. Random forests is one of many decision tree ensemble methods, which can be used for non-linear supervised learning. In R, the standard cran package is randomForest and the main function is also called randomForest.

Randomforest can be seen as a black-box algorithm which output a model-fit(the forest) when inputted a training data set and a set of paremters. The forest, can take an input of features and output predictions. Each discision tree in the forest will make a prediction, and the votes by each tree will be combined in an ensemble. The predicted response is either a number/scalar (regression model, mean of each tree) or one label out of K classes (classification, majority vote) or a vector of K length with pseudo probability of any K class membership(classification, pluralistic vote).

```
library(randomForest)


Warning: pakke 'randomForest' blev bygget under R version 3.1.3

randomForest 4.6-10
Type rfNews() to see new features/changes/bug fixes.


#a simple dataset of normal distributed features
obs = 2000
vars = 6
X = data.frame(replicate(vars,rnorm(obs,mean=0,sd=1)))
#the target/response/y depends of X by some 'unknown hidden function'
hidden.function = function(x,SNR=4) {
  ysignal <<- with(x, .5 * X1^2 + sin(X2*pi) + X3 * X4) #y = x1^2 + sin(x2) + x3 * x
  ynoise = rnorm(dim(x)[1],mean=0,sd=sd(ysignal)/SNR)
  cat("actucal signal to noise ratio, SNR: \n", round(sd(ysignal)/sd(ynoise),2))
  y = ysignal + ynoise
```

```
}
y = hidden.function(X)


actucal signal to noise ratio, SNR:
 4.06


#scatter plot of X y relations
plot(data.frame(y,X[,1:4]),col="#00004520")
```




```
#no noteworthy linear relationship
print(cor(X,y))


           [,1]
X1 0.072139080
X2 0.016109836
```

```
X3 0.017312007
X4 0.040905414
X5 0.008382099
X6 0.001777402
```

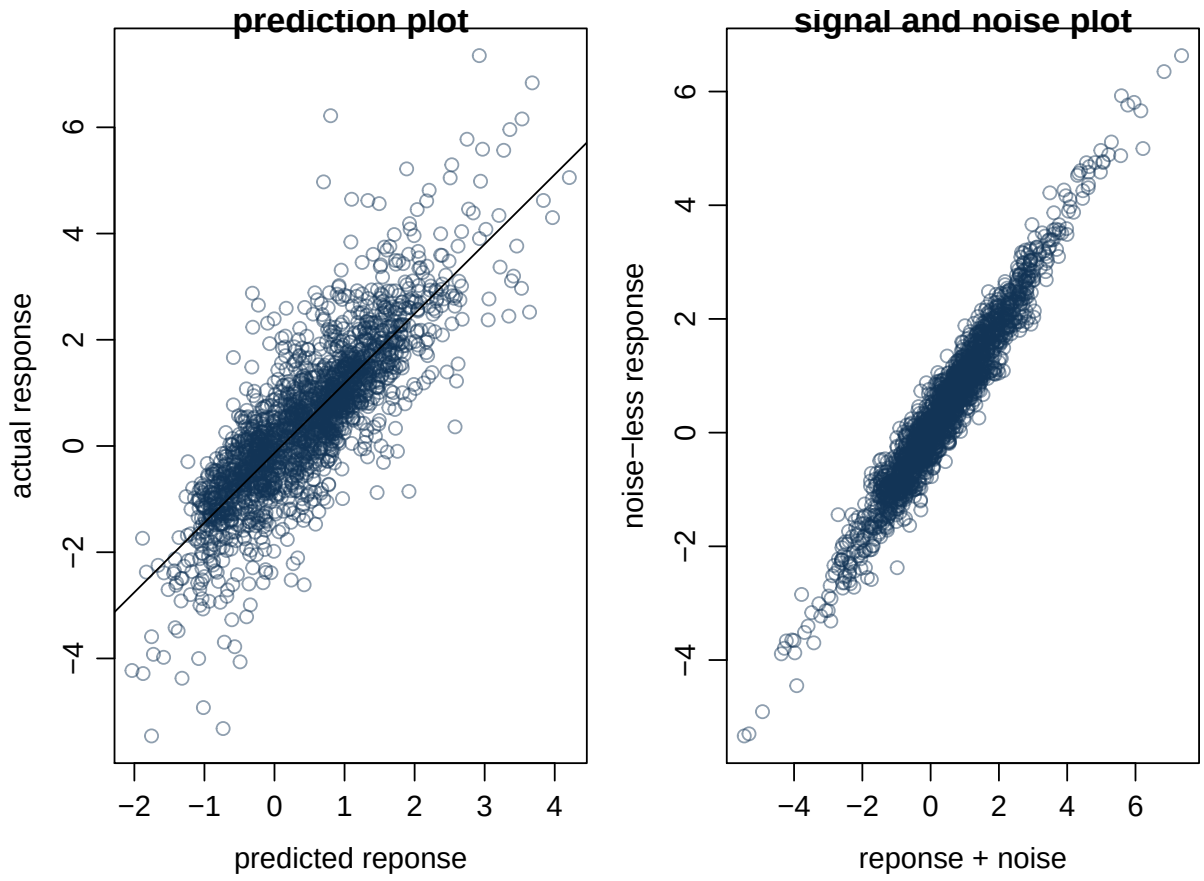
```
#RF is the forest_object, randomForest is the algorithm, X & y the training data
RF = randomForest(x=X,y=y,
                  importance=TRUE, #extra diagnostics
                  keep.inbag=TRUE, #track of bootstrap process
                  ntree=500,        #how many trees
                  replace=FALSE)   #bootstrap with replacement?
print(RF)
```

```
Call:
 randomForest(x = X, y = y, ntree = 500, replace = FALSE, importance = TRUE,      kee
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 2

          Mean of squared residuals: 0.7332501
                    % Var explained: 65.28
```

The modelfit prints some information. This is regression random forest, 500 trees were grown and in each split 2 variables we tried (mtry=vars/3). Next the model automatically does out-of-bag cross-validation (OOB-CV). The results obtained from OOB-CV is similar to 5-fold cross validation. But to perform an actual 5-fold-CV, 5 random forests must be trained. That would compute 5 times slower. The RF object is a list of class randomForest, which contains the ”forest”and alot of pre-made diagnostics. Let's try understand the most dentral statistics.

```
#RF£predicted is the OOB-CV predictions
par(mfrow=c(1,2))
plot(RF$predicted,y,
     col="#12345678",
     main="prediction plot",
     xlab="predicted reponse",
     ylab="actual response")
abline(lm(y~yhat,data.frame(y=RF$y,yhat=RF$predicted)))
```

```
#this performance must be seen in the light of the signal-to-noise-ratio, SNR
plot(y,ysignal,
     col="#12345678",
     main="signal and noise plot",
     xlab="reponse + noise",
     ylab="noise-less response")
```




```
cat("explained variance, (pseudo R^2) = 1- SSmodel/SStotal = \n",
    round(1-mean((RF$pred-y)^2)/var(y),3))
```

```
explained variance, (pseudo R^2) = 1- SSmodel/SStotal =
 0.653
```

```
cat("model correlation, (Pearson R^2) = ",round(cor(RF$pred,y)^2,2))
```

```
model correlation, (Pearson R^2) =  0.69
```

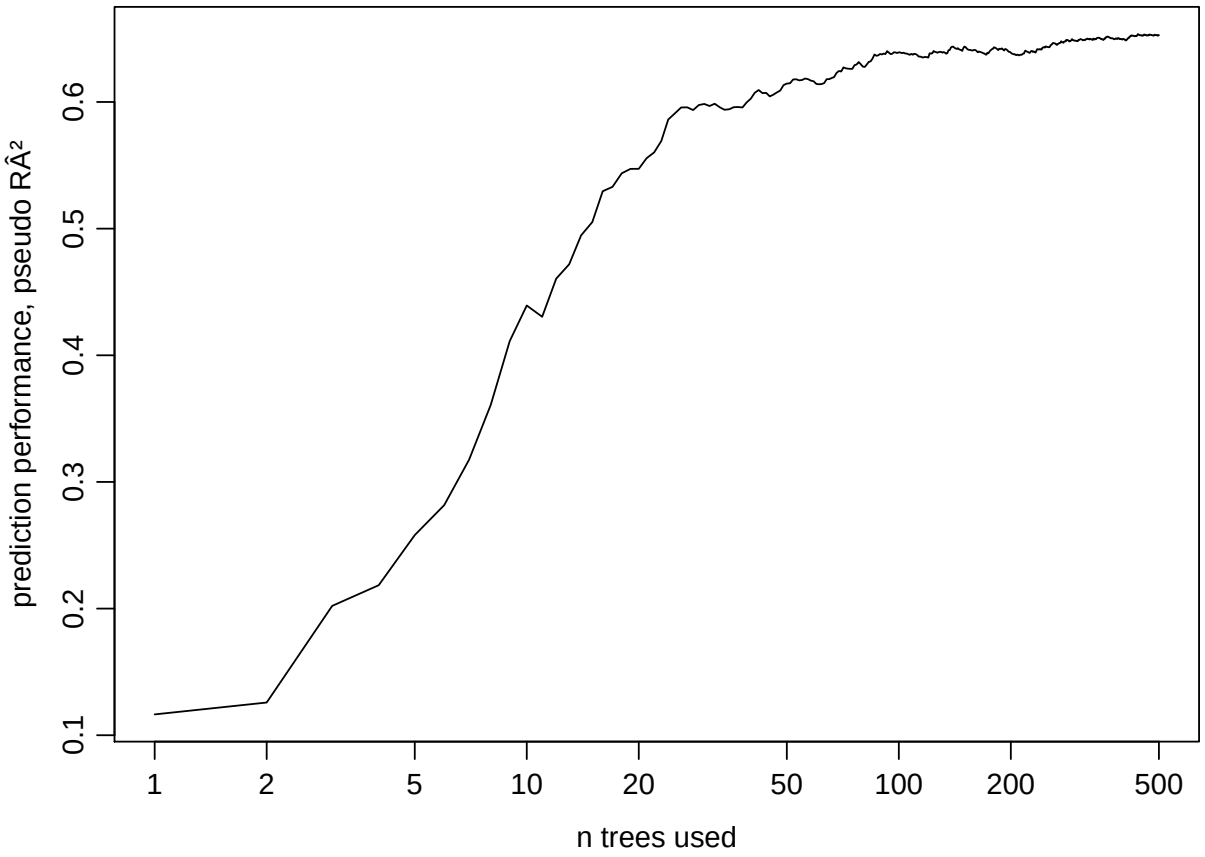
```
cat("signal correlation, (Pearson R^2) = ",round(cor(y,ysignal)^2,2))
```

```
signal correlation, (Pearson R^2) =  0.94
```

The random forest will never make a perfect fit. But if the number samples and trees goes toward a sufficient large number, the model fit will in practice be good enough. If the true model function has a complex curvature, signal-to-noise level is low and relatively few samples is used for training (e.g. 100), the model fit will be highly biased (complex curvature is inhibited) in order to retain some stability/robustness.

RF$rsq contains a vector of ntree elements. Each element is the OOB-CV (pseudo R^2) if only this many trees had been grown. Often it is enough to grow 50 trees. 10 Trees is rarely good enough. In theory prediction performance will increase assymtotic to a maximum level. In practices prediction performance can decrease, but such incidents are small, random and cannot be reproduced. More trees is never worse, only waste of time.

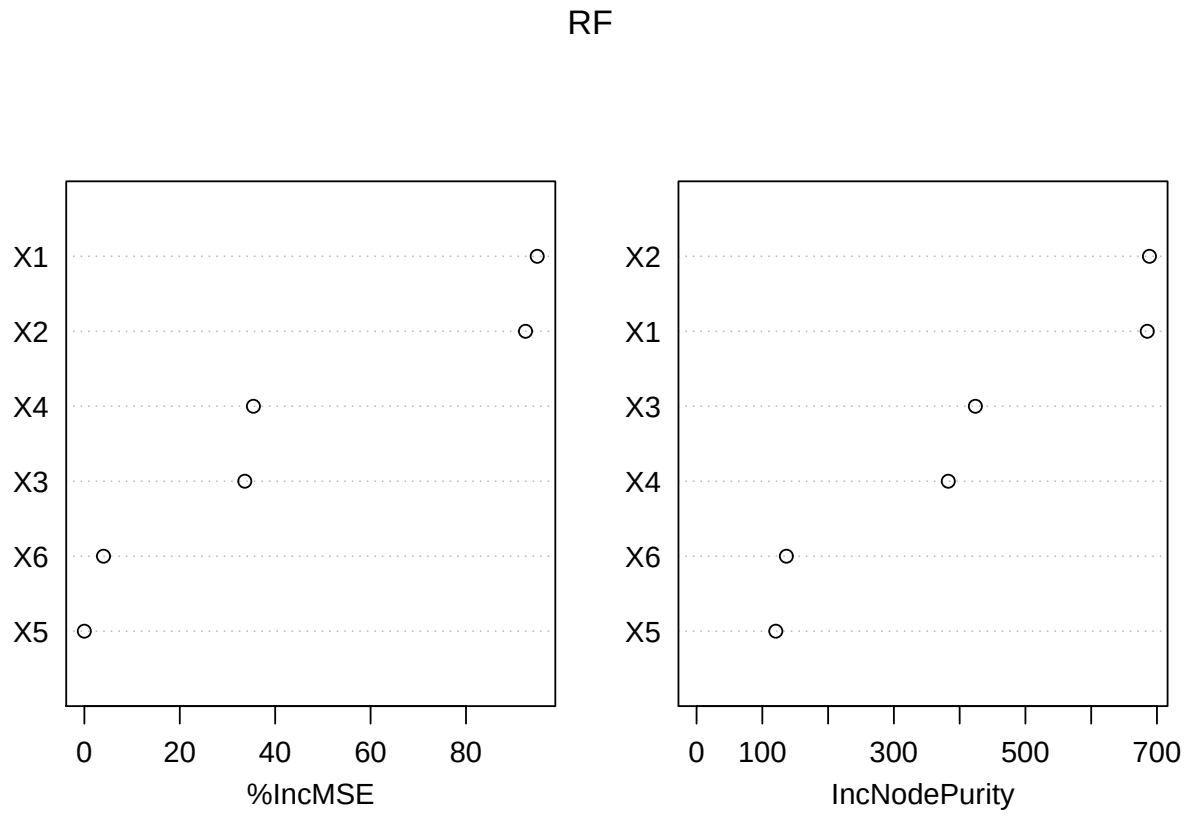
```
par(mfrow=c(1,1))
plot(RF$rsq,type="l",log="x",
     ylab="prediction performance, pseudo R^2",
     xlab="n trees used")
```



The RF object contains a matrix called importance(RF$importance). First column (when importance=TRUE) is variable permutation importance (VI). VI is a value for each variable. It is the change of OOB-CV mean square error% if this variable was permuted(shuffled) after the model was trained. Permuting a variable makes it random and with no useful information. If the model deteriorated the most by permuting a given variable, this variable is the most 'important variable'. Thus without trying to understand the model itself, it is possible to ruin the variable and measure how much CV performance deteriorates. Sometimes is the purpose of random forest modeling not the prediction itself, but to obtain the variable importance. E.g. for selecting genes associated with cancer, one can use variable importance to choose 'important' genes for further examiniation.

```
##variable importance
RF$importance


         %IncMSE IncNodePurity
X1  8.413839e-01      685.2656
X2  8.610915e-01      688.5898
```

```
X3  5.293861e-01       423.9268
X4  5.433030e-01       382.7031
X5 -4.741791e-05       120.4465
X6  1.434149e-02       136.5640


#short cut to plot variable importance
varImpPlot(RF)
```

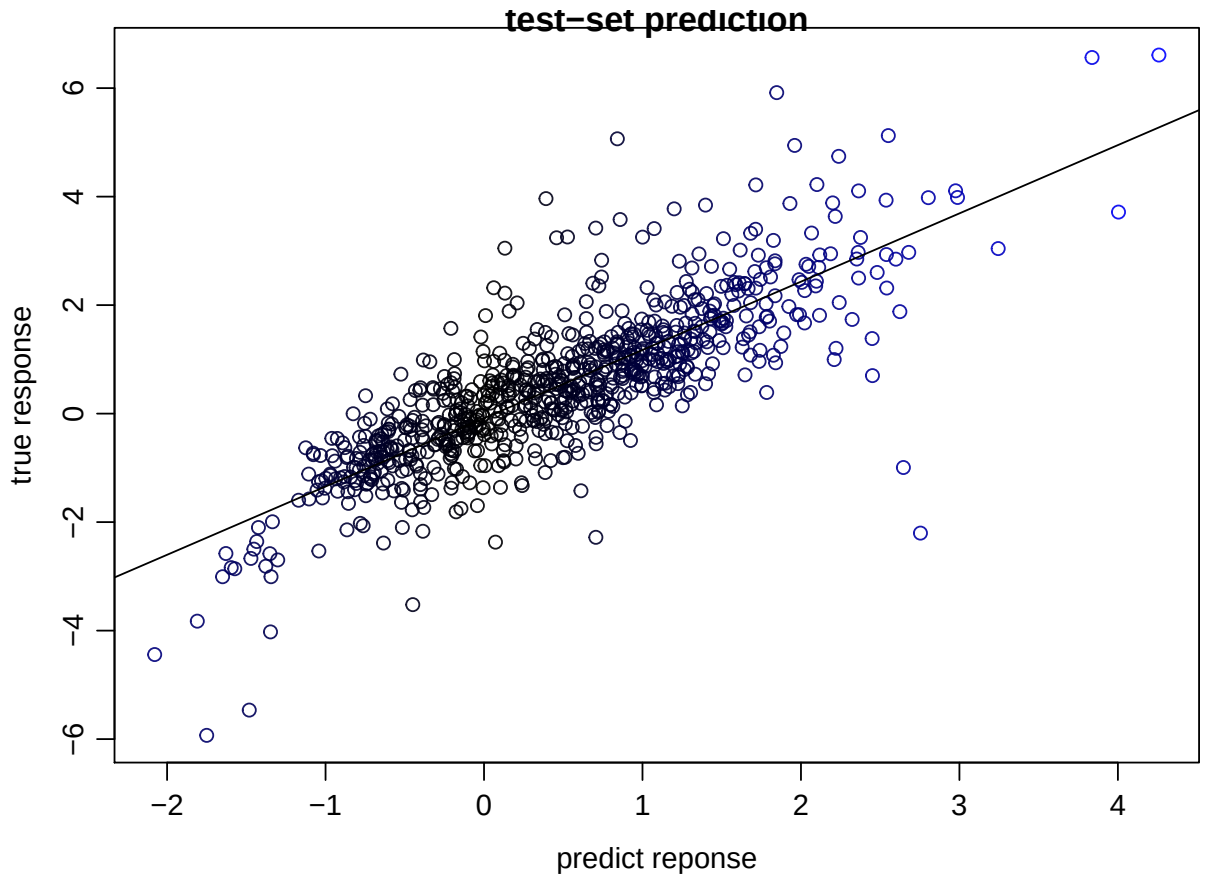


Notice variable 5 and 6 are rather unimportant. Gini-importance, the second importance term, is completely inferrior, and should not be used.

The RF object can be used to make new predictions.

```
#lets try to predict a test set of 800 samples
X2 = data.frame(replicate(vars,rnorm(800)))
true.y = hidden.function(X2)
```

```
actucal signal to noise ratio, SNR:
 4.12


pred.y = predict(RF,X2)
par(mfrow=c(1,1))
plot(pred.y,true.y,
     main= "test-set prediction",
     xlab= "predict reponse",
     ylab= "true response",
     col=rgb(0,0,abs(pred.y/max(pred.y)),alpha=.9))
abline(lm(true.y~pred.y,data.frame(true.y,pred.y)))
```

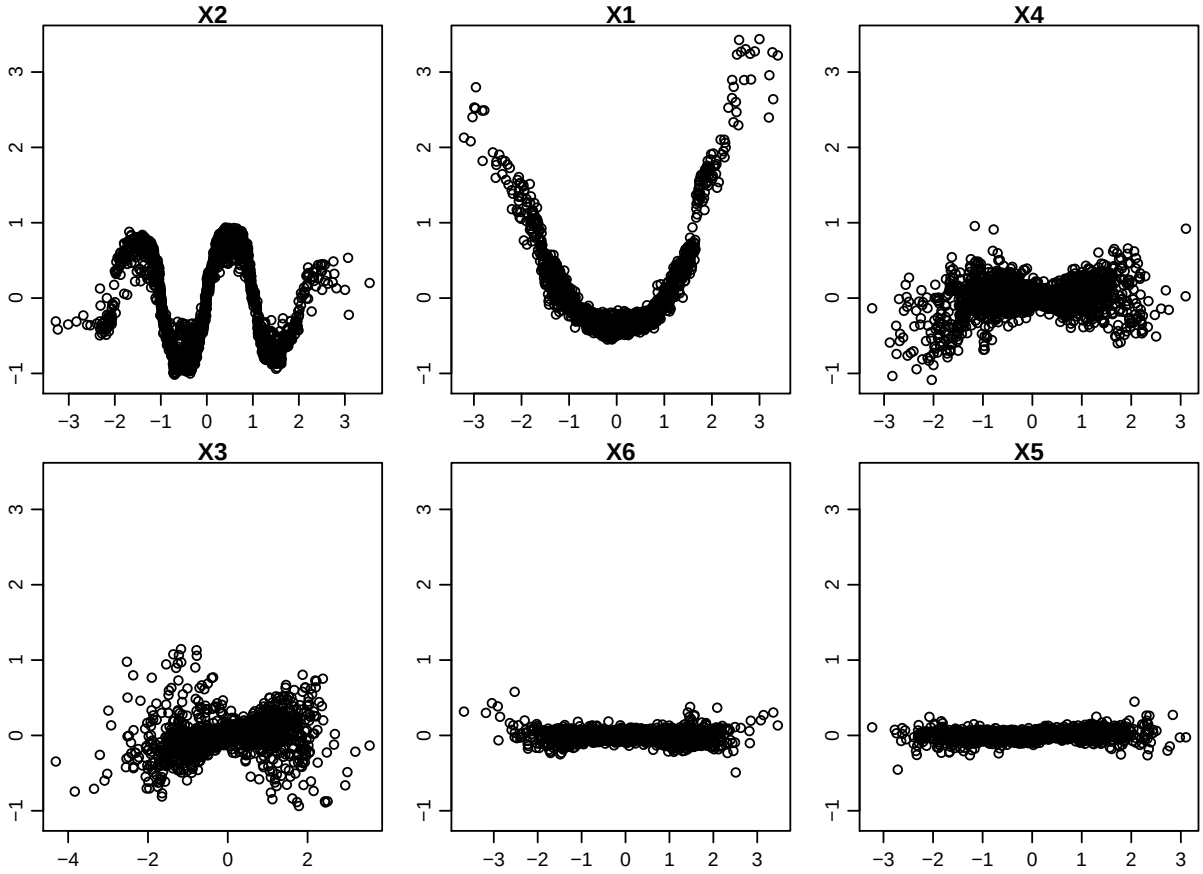


9.2.8 forestFloor

Random forest is regularly only used to either variable selection with 'variable importance' or to build a black-box forecast machine. The package forestFloor can be used to explore and visulize the curvature of a RF model-fit. The curvature of multivariate models are high dimensional and cannot be plotted properly. Multi linear regression is fairly easy as it only comprise som coefficients describing a hyper-plane. For non-linear models, the user often have not the slightest idea of the actual curvature of the model.

-If a non-linear model predicts a given problem well, its curvature might inspire and elaborate the scientists understanding of the problem.

```
#get packge from rForge. Maybe additional packages such trimTrees, mlbench, rgl and
#must be installed firts.
#install.packages("forestFloor", repos="http://R-Forge.R-project.org")
library(forestFloor, warn.conflicts=FALSE, quietly = TRUE)

#the function forestFloor outputs a forestFloor-object, here called ff
ff = forestFloor(RF,X)
print(ff)


this is a forestFloor('x') object

this object can be plotted in 2D with plot(x), see help(plot.forestFloor)

this object can be plotted in 3D with show3d(x), see help(show3d)



x contains following internal elements:
  FCmatrix imp_ind importance X Y


#ff can be plotted in different ways. As our data have six dimensions we cannot
#plot everything at once. Lets start with one-way forestFloor plots
plot(ff)
```



We can see that the contribution from variable X1 and X2 to the reponse looks a lot like X^2 and sin(X). The y-axis is called feature contributions(F). Each sample has one feature contribution for each variable, thus each sample is somewhere in each figure. The x-axis is the value of any sample by a given variable/feature. If sample i, at one time was placed in a node with response mean of 1.0 and was split by variable j, and sent to a new node of response mean -4.2, then sample i experienced a local increment of -5.2 by variable j. A feature contribution is the sum of steps (local increments) one sample took by a given variable through the forest divided by number of trees. Feature contributions are stored in matrix, called F or FCmatrix, of same size as X. Thus one feature contribution for each combination of variables and samples. $\tilde{F}$ denotes a OOB-CV version feature contributions only summed over those trees where the i^{th} sample is OOB diveded with how may times i^{th} sample was OOB. $\tilde{F}$ is more robust than F.

The following equation applies,
$\tilde{y}_i = \sum_{j=i}^{j} \tilde{F}_{ij} + \overline{y}_i$

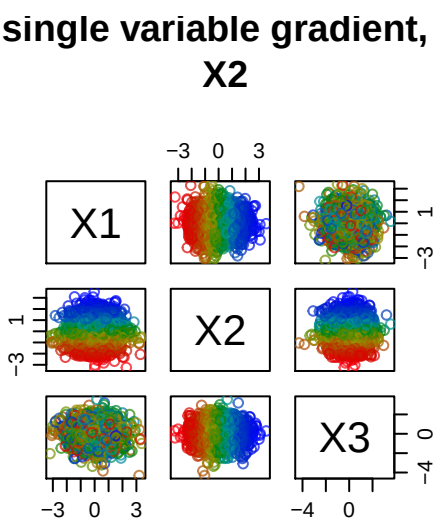
This means that any given OOB-CV prediction $\tilde{y}_i$ is equal to the sum of its feature con-

tributions (one for each variable in the model) plus the i^{it} bootstrap mean. The bootstrap mean is for any practical concern very very close to grand mean, mean(y). It deviates slightly from grand mean as each tree starts out with slightly different bootstrap mean and that each sample is OOB in different trees. The equation above states in practice, that any prediction is equal to the sum of the steps taking in the process plus the starting point, the grand mean.

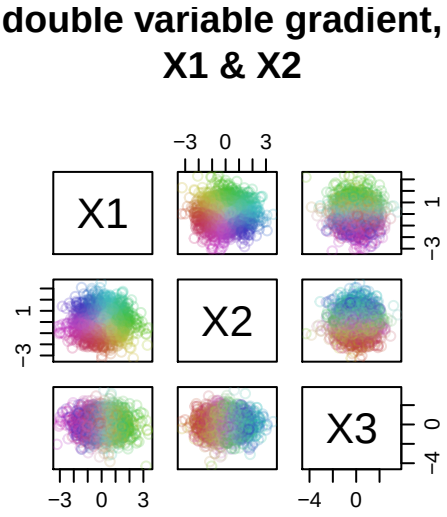
```
#following lines demonstrate the above equation
#this part can be skipped...

#this is the starting prediction of each tree
tree.bootstrap.mean = apply(RF$inbag,2,function(x) sum(x*y/sum(x)))

#this is the individual OOB bootstrap.mean for any observation
y.bootstrap.mean = apply(RF$inbag,1,function(x)  {
  sum((x==0) * tree.bootstrap.mean)/sum(x==0)
})

#OOB prediction = FC.rowsum + bootstrap.mean
yhat = apply(ff$FCmatrix,1,sum) + y.bootstrap.mean

#yhat and RF£predicted are almost the same, only limited by double precision.
##plot(yhat-RF£predicted) #try plot this
```

Next variable 3 and 4 do not look well explained in the one-way forestFloor plot. As $x3$ and $x4$ interacts, neither variable alone can explain its influence on the predicted response. We can use a colour-gradient to add another dimension to the plot. First we need to learn the function fcol().

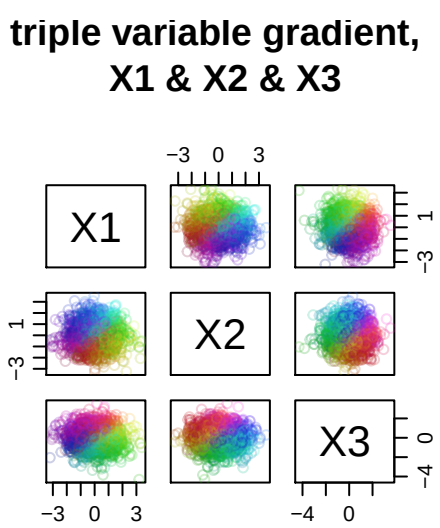
```
#fcol outputs a colourvector as function of a matrix or forestFloor object
#first input, what to colour by, 2nd input, what columns to use
par(mfrow=c(2,2))
pairs(X[,1:3],col=fcol(X,cols=2),main="single variable gradient, \n X2")
```



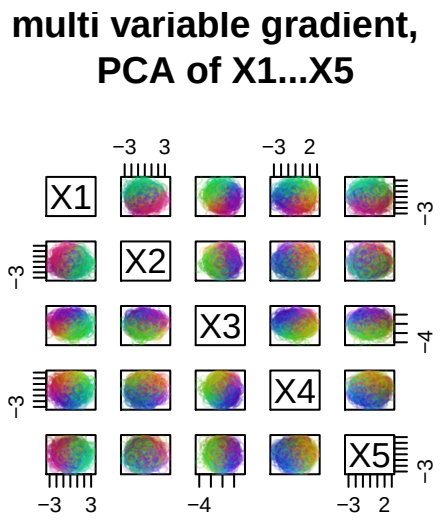

```
pairs(X[,1:3],col=fcol(X,cols=c(1:2)),main="double variable gradient, \n X1 & X2")
```




```
pairs(X[,1:3],col=fcol(X,cols=c(1:3)),main="triple variable gradient, \n X1 & X2 & X3
```

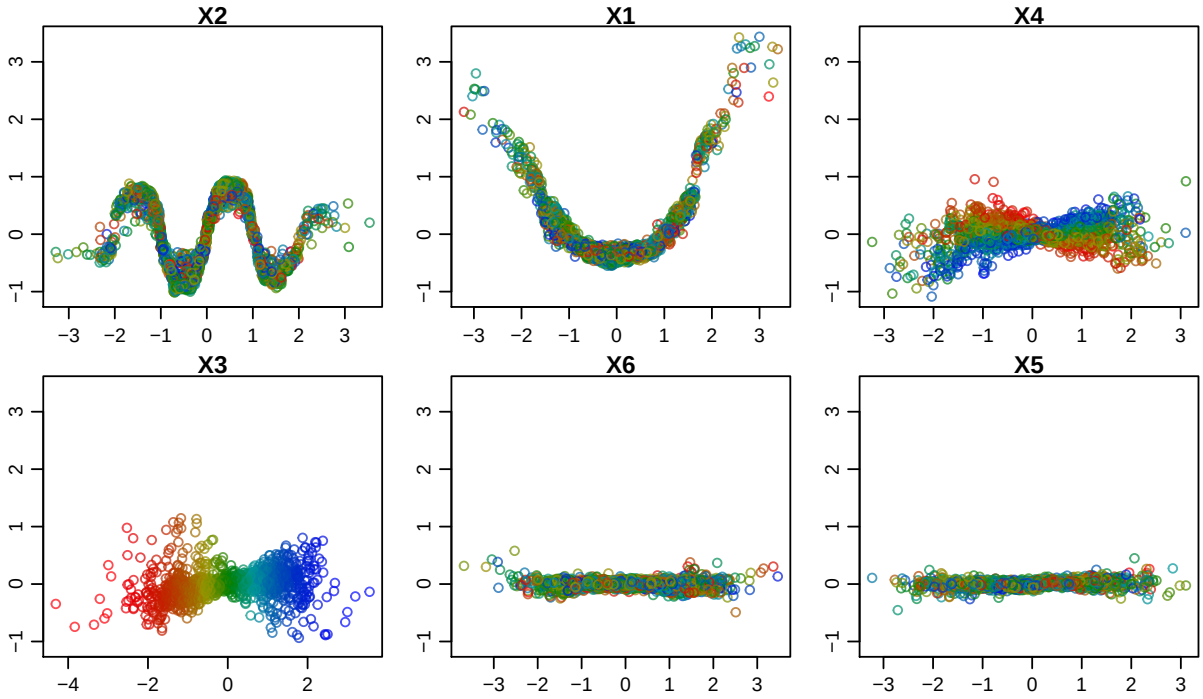



```
pairs(X[,1:5],col=fcol(X,cols=c(1:5)),main="multi variable gradient, \n PCA of X1...X
```




```
#the latter gradient uses PCA to chose three linear orthogonal gradients,
# which best can summarize the high dimensional variance of X.
#As variables of X is uncorrelated, this is not easy. For collinear data,
# multivariable gradients work great.
```

```
#print fcol outputs colours defined as RGB+alpha(transparency)
Col = fcol(X,cols=c(1:3))
print(Col[1:5])


[1] "#B0297833" "#0683AA33" "#1FBB5B33" "#B62D1933" "#C4FD3233"


#first two hexidemimals describe red, then green, then blue, then alpha.
# two hexidecimals is equal to a 10base number from 1 to 256


#now we apply a color gradient along variable x3
Col = fcol(ff,3,order=F)
plot(ff,col=Col)
```




```
#one sample will have same colour in every plot
#samples with similar colors have similar x3-values
```

We can see that a horizontal colour gradient emerges in plot X3, as we chose to color by this axis. Other plots seems randomly colored becaused the varibles are uncorrela-

ted and do not interact within the model. But for variable X4, the same colur gradient emerges horisontal. Two understand the colour pattern, it is easier to use a 3D-plot

```
show3d_new(ff,3:4,col=Col)
#one awesome 3D plot in a external window!!
```

Thus now it is visulized that one-way forestFloor plots X3 and X4 simply are 2D projections of af 3D saddle point curvature

9.3 Exercises

|||| Exercise 1 Exercise: Wine data

Use the wine data previously used.

a) Predict the wine-class using the different methods suggested.

b) Try also a PCA based version of some of the methods.

c) What are the results? Make your own test- and training data. Apply CV on the training data and predict the test data.

d) By looking at various plots (in addition to prediction ability) consider which models seem to be mostly appropriate for the data.

|||| Exercise 2 Exercise: random forest prostate

train a model to predict prostate cancer use variable importance to identify 'genes' related

a) How well can prostate cancer be explained from gene expression levels?

```
No. of variables tried at each split: 77
OOB estimate of  error rate: 2.94%
```

b) What genes code for prostate cancer?

```
RF$importance
varImpPlot(RF)
```

c) This model has 100 samples and 6000 variables. What does the curvature of a very sparse random forest model look like?

|||| Exercise 3 Exercise: Random forest exercise 2 - abelone

a) How well can a random forest model predict the age of abelones

OOB estimate of error rate: 44.75%

Confusion matrix:

	F	I	M	class.error
F	529	155	623	0.5952563
I	120	1027	195	0.2347243
M	525	251	752	0.5078534

b) What are the main effects (main trends) of the model? (use one-way forestFloor)

c) Are the any large interactions besides main-effect and what variables are implicated? (use one way forestFloor + fcol)

d) Describe the two-way forestFloor plot between shell.weight and shrucked.weight: Why would the model use such a curvature? (use show3d_new and fcol)

	F	I	M	MeanDecreaseAccuracy	MeanDecreaseGini
length	0.008414153	0.04534589	0.0168381171	0.02335849	170.1257
diameter	0.021652500	0.06688541	0.0119871710	0.03260325	177.9561
height	-0.003357832	0.08153409	0.0007117459	0.02539909	167.1482
whole_weight	0.029615672	0.17420313	0.0230613926	0.07359024	287.7510
shucked_weight	-0.027972688	0.01460232	0.0503146712	0.01433858	233.4261
viscera_weight	-0.008542939	0.16119629	0.0144930743	0.05438009	294.9048
shell_weight	0.003847196	0.07758786	0.0036528176	0.02748029	251.1108
rings	-0.007140879	0.16212373	0.0001918313	0.04993962	169.9785