Help File

Laura Julià Melis

1/12/2020

Contents

COMPUTER LABS

BLOC 1

Topic 1.

Assignment 1. Spam classification with nearest neighbors.

The data file spambase.xlsx contains information about the frequency of various words, characters etc for a total of 2740 e-mails. Furthermore, these e-mails have been manually classified as spams (spam = 1) or regular e-mails (spam = 0)

1. Import the data into R and divide it into training and test sets (50%/50%) by using the following code:

To read xlsx data, we call read_xlsx function from readxl package. Then we divide the data into training and testing data.

```r
# Importing the data:
library(readxl)
data <- read_excel("spambase.xlsx")

# Dividing it into training and test sets:
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]
test=data[-id,]
```

2. Use logistic regression (functions glm(), predict()) to classify the classification principle

$$\hat{Y} = 1 \text{ if } p(Y = 1|X) > 0.5, \text{ otherwise } \hat{Y} = 0$$

and report the confusion matrices (use table()) and the misclassification rates for training and test data. Analyse the obtained results.

```r
# Logistic regression model:
fit1 <- glm(Spam~., data= train, family = binomial)

# Predictions:
fitted_prob_test <- predict(fit1, newdata=test,  type ="response") # type response to get probabilities
fitted_prob_train<- predict(fit1, newdata=train,  type ="response")

fitted_response_train <- ifelse(fitted_prob_train > 0.5, 1, 0)
fitted_response_test <- ifelse(fitted_prob_test > 0.5, 1, 0)
```

- Confusion matrices.

```r
(confusion_matrix_test <- table(test$Spam, fitted_response_test))
```

```
##    fitted_response_test
##       0   1
##   0 808 143
##   1  92 327
```

```r
(confusion_matrix_train <- table(train$Spam, fitted_response_train))
```

```
##    fitted_response_train
##       0   1
##   0 804 127
##   1  93 346
```

(In the table() function, the first argument is for the rows and the second argument is for the columns. So, in order to use the normal convention for confusion matrices, all the confusion matrices will be created with true labels on the rows and predicted labels on the columns.)

It can be observed that the number of observations classified correctly (number of cases in the diagonal of the matrix) is similar in both confussion matrices but we can notice a little difference in favour of the train confussion matrix: the number of non-spam mails (0) classified as spam (1) is a bit greater in the test matrix.

- Misclassification rates.

For the test data:

```r
mean(fitted_response_test != test$Spam)
```

```
## [1] 0.1715328
```

For the training data:

```
mean(fitted_response_train != train$Spam)
```

```
## [1] 0.1605839
```

The misclassification rate for the test set is a bit greater than the one for the training set.

3. Use logistic regression to classify the test data by the classification principle

$$\hat{Y} = 1 \text{ if } p(Y = 1|X) > 0.8, \text{ otherwise } \hat{Y} = 0$$

and report the confusion matrices (use table()) and the misclassification rates for training and test data. Compare the results. What effect did the new rule have?

- Confusion matrices.

```
fitted_response_train <- ifelse(fitted_prob_train > 0.8, 1, 0)
fitted_response_test <- ifelse(fitted_prob_test > 0.8, 1, 0)

(confusion_matrix_test <- table(test$Spam, fitted_response_test))
```

```
##    fitted_response_test
##       0   1
##   0 931  20
##   1 314 105
```

```
(confusion_matrix_train <- table(train$Spam, fitted_response_train))
```

```
##    fitted_response_train
##       0   1
##   0 921  10
##   1 333 106
```

In this case, where a probability of 0.8 has been used as the classification principle, the number of observations in the diagonal (well classified) is greater than before for the both data sets.

- Misclassification rates.

For the test data:

```
mean(fitted_response_test != test$Spam)
```

```
## [1] 0.2437956
```

For the training data:

```
mean(fitted_response_train != train$Spam)
```

```
## [1] 0.250365
```

In this occasion, both missclassification rates are greater than before but it's worth mentioning that the number of cases being non-spam that would be classified as spam has decreased a lot so this principle is better than the one in the second question.

4. Use standard classifier kknn() with K=30 from package kknn, report the the misclassification rates for the training and test data and compare the results with step 2.

```
library(kknn)
knn_test_30 <- kknn(Spam~., train, test, k=30)
knn_train_30 <- kknn(Spam~., train, train, k=30)
```

```
knn_response_train_30 <- ifelse(knn_train_30$fitted.values > 0.5, 1, 0)
knn_response_test_30 <- ifelse(knn_test_30$fitted.values > 0.5, 1, 0)
```

- Misclassification rate for the test data:

```
mean(knn_response_test_30 != test$Spam) # for the test data
```

```
## [1] 0.3131387
```

- Misclassification rate for the training data:

```
mean(knn_response_train_30 != train$Spam) # for the training data
```

```
## [1] 0.1671533
```

If we compare these misclassification rates with the ones obtained in question 2 we can observe that the rate values for the training set are almost the same (0.161 and 0.167) while the rate for the test set is around 0.14 points bigger when using 30-nn model than the linear regression model. So, the results obtained with this new model are worse.

5. Repeat step 4 for K=1 and compare the results with step 4. What effect does the decrease of K lead to and why?

```
library(kknn)
knn_test_1 <- kknn(Spam~., train, test, k=1)
knn_train_1 <- kknn(Spam~., train, train, k=1)

knn_response_train_1 <- ifelse(knn_train_1$fitted.values > 0.5, 1, 0)
knn_response_test_1 <- ifelse(knn_test_1$fitted.values > 0.5, 1, 0)
```

- Misclassification rate for the test data:

```
mean(knn_response_test_1 != test$Spam) # for the test data
```

```
## [1] 0.3591241
```

- Misclassification rate for the training data:

```
mean(knn_response_train_1 != train$Spam) # for the training data
```

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

When choosing K=1, what we are doing is choosing only one point (the closest one), so when making predictions the model will always classify the observation to that point (because it will have probability 1). For this reason, the missclassification rate for the training set is 0 but the rate for the test set is greater than the one using K=4. Basically, the model is over-fitting very much.

Assignment 2. Inference about lifetime of machines.

The data file machines.xlsx contains information about the lifetime of certain machines, and the company is interested to know more about the underlying process in order to determine the warranty time. The variable is following:Length: shows lifetime of a machine

1. Import the data to R.

```
library("readxl")
data = unlist(read_excel("machines.xlsx"))
set.seed (12345)
#test data = data[-id ,]
```

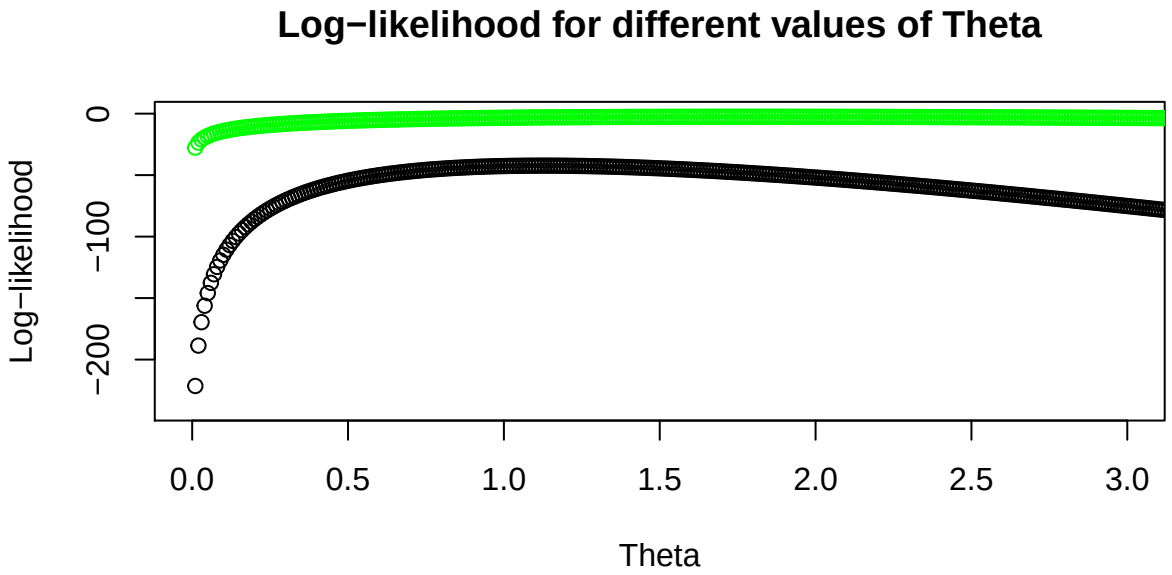
2. Assume the probability model $p(x|\theta) = \theta e^{-\theta x}$ for x=Length in which observations are independent and identically distributed. What is the distribution type of x? Write a function that computes the log-likelihood log $p(x|\theta)$ for a given θ and a given data vector x. Plot the curve showing the dependence of log-likelihood on θ where the entire data is used for fitting. What is the maximum likelihood value of θ according to the plot?

3. Repeat step 2 but use only 6 first observations from the data, and put the two log-likelihood curves (from step 2 and 3) in the same plot. What can you say about reliability of the maximum likelihood solution in each case?

The probability model is given by $p(x|\theta) = \theta e^{-\theta x}$, in which observations are independent and identically distributed. This gives that the distribution type of x is exponential. The log-likelihood was then computed for different values of θ (see appendix for code). The graph in figure 1 shows the dependence of log-likelihood on θ, both for the entire data and for only the first 6 observations.

```
# Function that computes the log-likelihood
loglike = function(theta , x) {
  lglike = 0
  for(val in x) {
    lglike = lglike + log(theta*exp(-theta*val))
  }
  return(lglike)
}

bayes = function(theta , x) {
  gamma = 10
  prior = log(gamma*exp(-gamma*theta))
  result = prior

  for (val in x) {
    result = result + log(theta*exp(-theta*val))
  }
  return(result)
}

theta_vals = seq(0, 4, 0.01)
vec = vector(length = length(theta_vals))
index = 1
for(theta in theta_vals) {
  res = loglike(theta, data)
  vec[index] = res
  index = index + 1
}

# Question 3
new_data = data[c(1:6)] #new_data = head(data)
vec_2 = vector(length = length(theta_vals))
index = 1
for(theta in theta_vals){
  res = loglike(theta, new_data)
  vec_2[index] = res
  index = index + 1
}

# Plot in order to find maximum likelihoodFor (steps 2 and 3)
```

```
{plot(theta_vals , vec, xlim = c(0, 3), ylim = c(-240, 0), main="Log-likelihood for different values of
points(theta_vals, vec_2, col="green")} # plot(theta_vals , vec_2)
```




```
max_likelihood_val = theta_vals[which.max(vec)]

max_likelihood_val_2 = theta_vals[which.max(vec_2)]
```

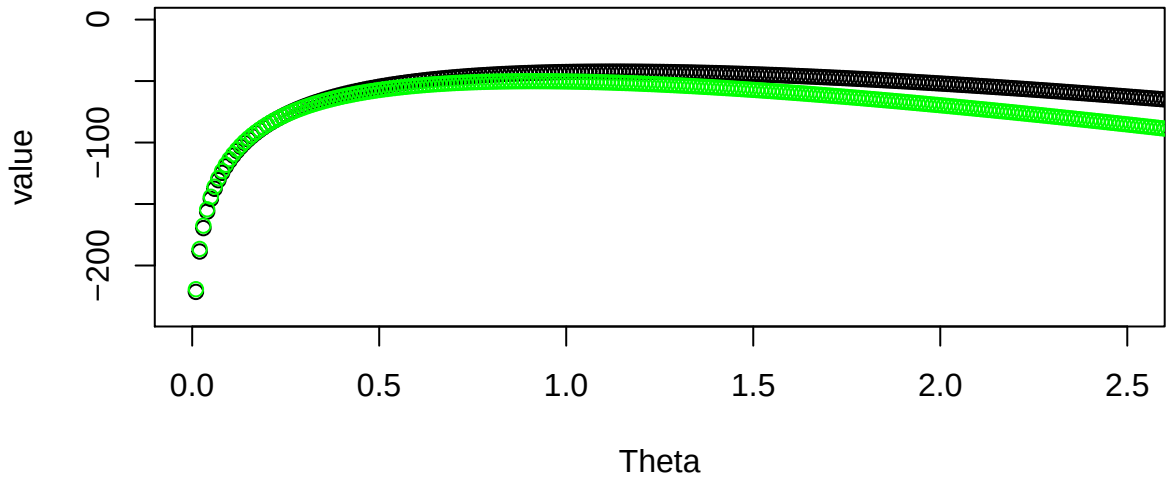
Figure 1: Dependence of log-likelihood on θ. The black points were computed from the entire data set, while the green points were computed from only the first 6 observations.

According to figure 1, the maximum likelihood value of θ (using the entire data set) is 1.13. The plot also shows that when only using the first 6 observa- tions, the maximum value is is 1.79. We can however see from the plot that the maximum value given by using the smaller data set (green curve) is much more unreliable since the curve never really starts to clearly decrease as the black curve does. Using a smaller subset would obviously yield a more unreliable result since the other values in the data set are ignored. The reliability of the value obtained for θ is therefore much bigger when the entire data set was used.

4. Assume now a Bayesian model with $p(x|\theta) = \theta e^{-\theta x}$ and a prior $p(\theta) = \lambda e^{-\lambda\theta}, \lambda = 10$. Write a function computing $l(\theta) = log(p(x|\theta)p(\theta))$. What kind of measure is actually computed by this function? Plot the curve showing the dependence of $l(\theta)$ on θ computed using the entire data and overlay it with a plot from step 2. Find an optimal θ and compare your result with the previous findings.

In this task we multiply the likelihood with the prior probability, which results in the posterior probability. In figure 2 this is plotted together with the curve from Task 2.

```
vec_3 = vector(length = length(theta_vals))
index = 1
for(theta in theta_vals){
  res = bayes(theta , data)
  vec_3[index] = res
  index = index + 1
}

{plot(theta_vals, vec, xlim = c(0, 2.5), ylim = c(-240, 0) , xlab="Theta" , ylab="value")
points(theta_vals , vec_3, col="green")}
```




```
max_theta = theta_vals[which.max(vec_3)]
```

Figure 2: The black curve represents the log-likelihood dependent on θ, while the green curve represents the posterior probability dependent on θ.

We can see that the curve for the posterior probability starts out similar to the curve for log-likelihood but then descends more quickly. The optimal θ for the posterior probability is given by 0.91.

5. Use θ value found in step 2 and generate 50 new observations from $p(x|\theta) = \theta e^{-\theta x}$ (use standard random number generators). Create the histograms of the original and the new data and make conclusions.

The function *rexp* was used to generate a random exponential distribution of 50 new observations. Figure 3 shows the histograms of both the original data and the newly generated data. We can see that the two histograms are pretty similar and that they both follow an exponential distribution.

```
new_data = rexp(50, max_likelihood_val)
par(mfrow=c(1, 2))
hist(data, main="Original data")
hist(new_data, main="New data")
```

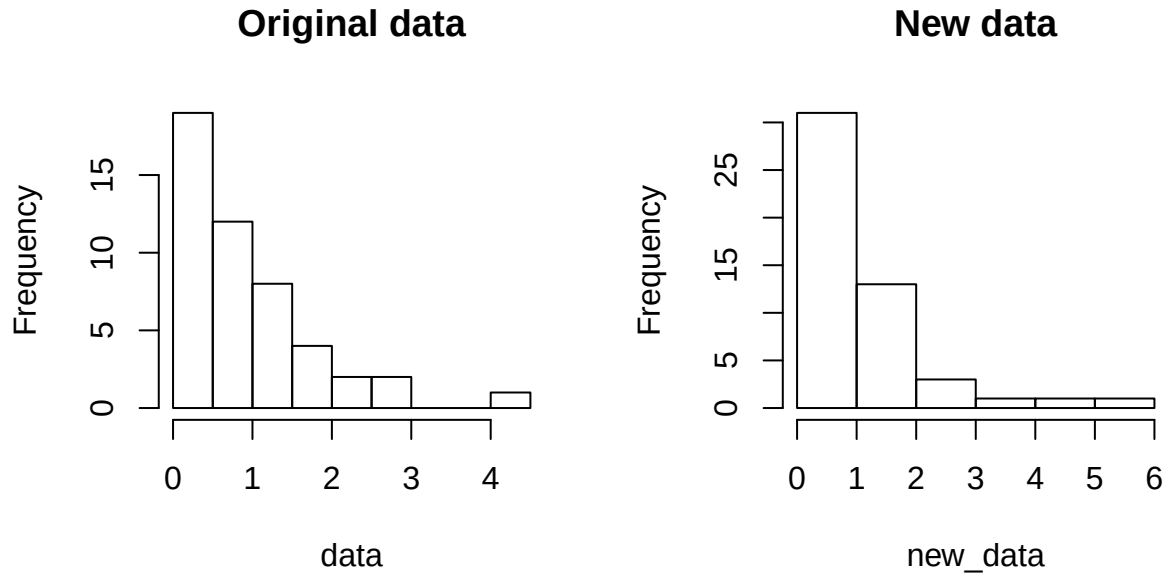


Figure 3: Histograms of the original data and the newly generated data.

Assignment 3. Feature selection by cross-validation in a linear model.

1. Implement an R function that performs feature selection (best subset selection) in linear regression by using k-fold cross-validation without using any specialized function like lm() (use only basic R functions). Your function should depend on:

```
X: matrix containing X measurements.
Y: vector containing Y measurements
Nfolds: number of folds in the cross-validation.
```

You may assume in your code that matrix X has 5 columns. The function should plot the CV scores computed for various feature subsets against the number of features, and it should also return the optimal subset of features and the corresponding cross-validation (CV) score. Before splitting into folds, the data should be permuted, and the seed 12345 should be used for that purpose.

```
mylin=function(X,Y, Xpred){
  Xpred1 = cbind(1,Xpred)
  X1 = cbind(1, X)
  beta = solve( t(X1) %*% X1) %*% t(X1) %*% Y # obtained using the "training" data matrix X
  Res = Xpred1%*%beta # y_hat for the "test" data
  return(Res)
}

myCV=function(X,Y,Nfolds){
  n=length(Y) # number of observations (rows)
  p=ncol(X) # number of covariates (variables or columns)
  set.seed(12345)
  ind=sample(n,n) # indexes are randomized
  X1=X[ind,] # randomize the order of the observations
  Y1=Y[ind]
  sF=floor(n/Nfolds) # number of observations inside each fold
  MSE=numeric(2^p-1) # vector of the length of 2^p-1 combinations
  Nfeat=numeric(2^p-1)
  Features=list() # features that will be selected
  curr=0 # current
  folds_obs <- cut(1:n,breaks=Nfolds,labels=FALSE)

  #we assume 5 features.
  for (f1 in 0:1)
    for (f2 in 0:1)
      for(f3 in 0:1)
        for(f4 in 0:1)
          for(f5 in 0:1){
            model= c(f1,f2,f3,f4,f5)
            if (sum(model)==0) next()
            SSE=0

            for (k in 1:Nfolds){
#MISSING:compute which indices should belong to current fold
            indices <- ind[which(folds_obs==k)] #indeces of the observations in fold k

#MISSING:implement cross-validation for model with features in "model" and iteration i.
            X_mylin <- X1[-indices,which(model==1)]
            XPred_mylin <- X1[indices,which(model==1)]
            Y_mylin <- Y1[-indices]
```

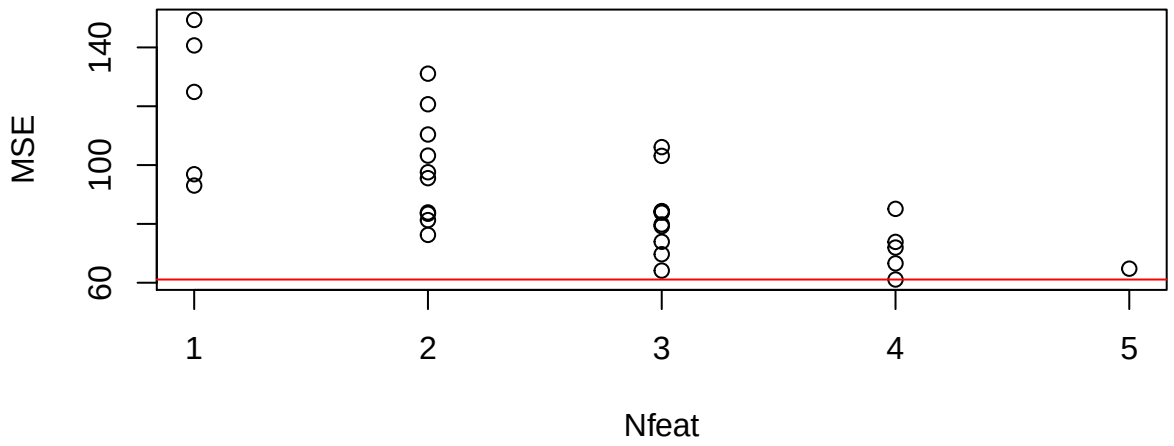
```
#MISSING:Get the predicted values for fold k, Ypred, and the original values for fold 'k', Yp.
              Ypred <- mylin(X_mylin, Y_mylin, XPred_mylin)
              Yp <- Y1[indices]

              SSE=SSE+sum((Ypred-Yp)^2)
            }
            curr=curr+1
            MSE[curr]=SSE/n
            Nfeat[curr]=sum(model)
            Features[[curr]]=model
          }
  plot(Nfeat, MSE) #MISSING: plot MSE against number of features
  abline(h=MSE[which.min(MSE)], col="red")

  i=which.min(MSE)
  return(list(CV=MSE[i], Features=Features[[i]]))
}
```

2. Test your function on data set swiss available in the standard R repository:

```
Fertility should be Y
All other variables should be X
Nfolds should be 5
```

Report the resulting plot and interpret it. Report the optimal subset of features and comment whether it is reasonable that these specific features have largest impact on the target.

```
data("swiss")
myCV(as.matrix(swiss[,2:6]), swiss[[1]], 5)
```




```
## $CV
## [1] 61.08948
##
## $Features
## [1] 1 0 1 1 1
```

In the plot we can observe the Mean Squared Errors associated to all the different possibilities of fitting a model. The red line indicates the minimum MSE so we can see that the number of features included in the fitted model with the lowest MSE is 4.

Also, from the list output we can observe that the optimal subset of features is (1 0 1 1 1) so we can write the optimal model as:

$$\text{Fertility} = \beta_0 + \beta_1 \cdot \text{Agriculture} + \beta_2 \cdot \text{Education} + \beta_3 \cdot \text{Catholic} + \beta_4 \cdot \text{Infant.Mortality}$$

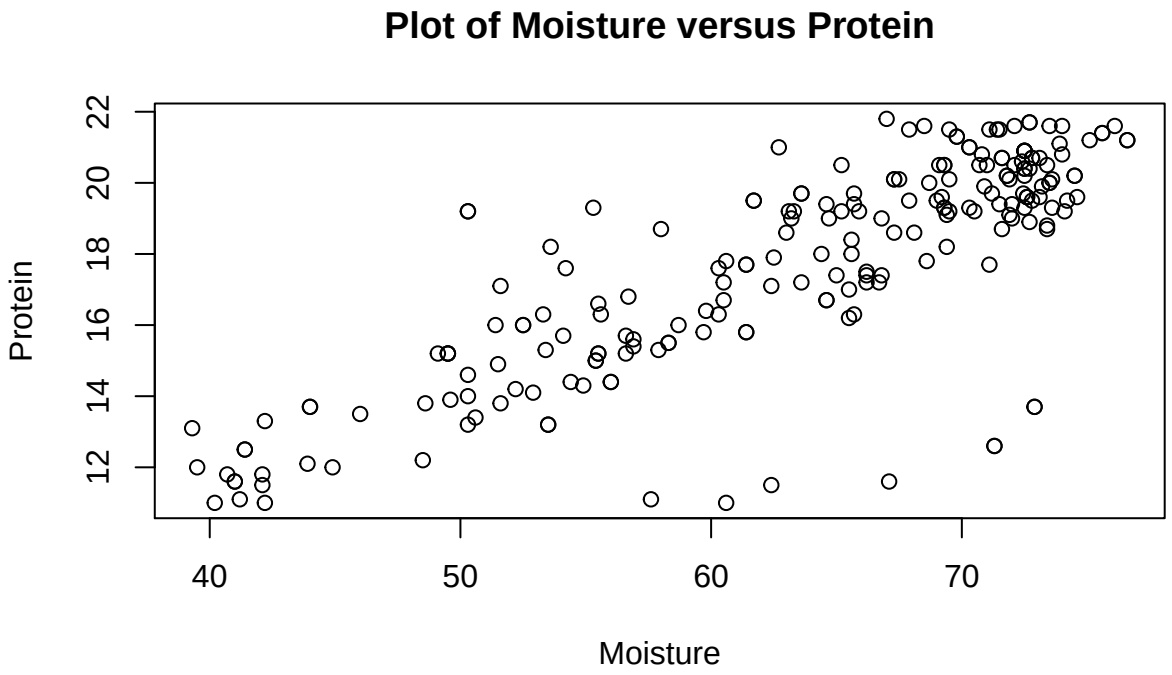
It is reasonable that Agriculture, Education, Religion (being catholic) or Infant mortality are variables that have a largest impact on fertility than having a good grade in the army exam.

Assignment 4. Linear regression and regularization.

The Excel file tecator.xlsx contains the results of study aimed to investigate whether a near infrared absorbance spectrum can be used to predict the fat content of samples of meat. For each meat sample the data consists of a 100 channel spectrum of absorbance records and the levels of moisture (water), fat and protein. The absorbance is -log10 of the transmittance measured by the spectrometer. The moisture, fat and protein are determined by analytic chemistry.

1. Import data to R and create a plot of Moisture versus Protein. Do you think that these data are described well by a linear model?

```
# Importing the data:
data <- read_excel("tecator.xlsx")
plot(data$Moisture, data$Protein, main="Plot of Moisture versus Protein",
     xlab="Moisture", ylab="Protein")
```



As can be seen from the plot, for most of the points, as Protein values increase the Moisture values do too. And this increasing relationship is linear, with some outliers though in the down right side, so a linear model whould probably fit the data but not so well.

2. Consider model M_i in which Moisture is normally distributed, and the expected Moisture is a polynomial function of Protein including the polynomial terms up to power i (i.e M_1 is a linear model, M_2 is a quadratic model and so on). Report a probabilistic model that describes M_i. Why is it appropriate to use MSE criterion when fitting this model to a training data?

A probabilistic model that describes M_i would the following:

$$\text{Moisture}_i = \beta_0 + \beta_1 \text{Protein}^1 + \cdots + \beta_i \text{Protein}^i + \epsilon_i$$

with $\epsilon_i \sim N(\mu, \sigma^2)$ (We didn't mention in the group nor in my individual group the distibution and its parameters).

Also, the probabilistic model can be written as:

$$\text{Moisture} \sim N(\beta_0 + \beta_1 \text{Protein}^1 + \cdots + \beta_i \text{Protein}^i, \sigma^2)$$

The mean squared error indicates how close a fitted regression model is to the data by taking the distances from the points (Y_i) to the fitted points ($\hat{Y}_i$) and squareing them:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

But with a polynomial function of the independent variable, the model can get very complex and overfit (the model would be too close to the training points because it would also capture the noise) meaning that the distance between the test data points and the predicted points would increase (and that would be bad, we don't want that).

Also, for a normally distributed model minimizing MSE results in the Maximum Likelihood Estimation, and this is the reason for why the MSE is appropiate in this case.

3. Divide the data into training and validation sets (50%/50%) and fit models $M_i = 1...6$. For each model, record the training and the validation MSE and present a plot showing how training and validation MSE depend on i (write some R code to make this plot). Which model is best according to the plot? How do the MSE values change and why? Interpret this picture in terms of bias-variance tradeoff.

```
# Creating new dataset with the new independent variables:
data2 <- data.frame( Moisture = data$Moisture, P1 = data$Protein,
                     P2 = data$Protein^2, P3 = data$Protein^3,
                     P4 = data$Protein^4, P5 = data$Protein^5,
                     P6 = data$Protein^6)
# Dividing the data into training and test sets:
n=dim(data2)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data2[id,]
test=data2[-id,]

# Fitting the models and calculating MSE's:
mse_test <- vector()
mse_train <- vector()
for(i in 2:7){
  fit <- lm(train$Moisture ~ ., data = train[,1:i])
  mse_test[i-1] <- mean((predict(fit, test)- test$Moisture)^2)
  mse_train[i-1] <- mean((predict(fit, train)- train$Moisture)^2)
}
```

- MSE table for the training and the test datasets.

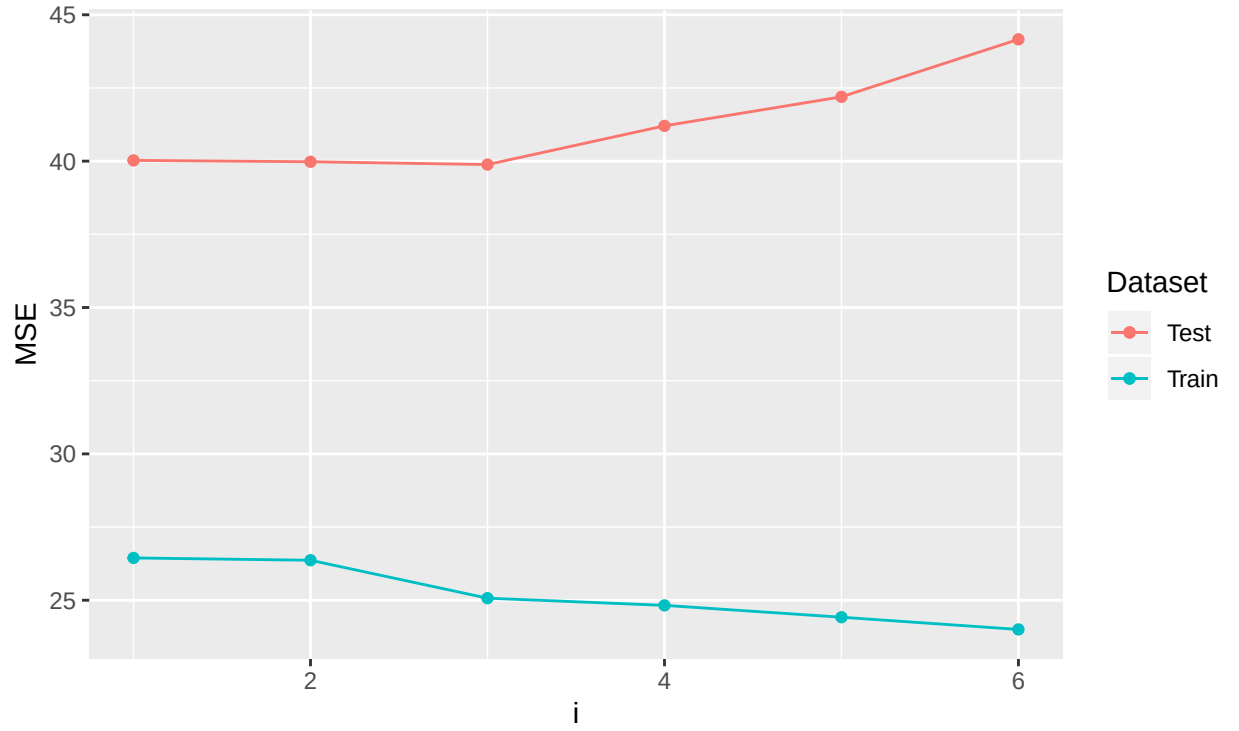
```
(df <- as.data.frame(cbind(i=1:6,mse_test, mse_train)))
```

```
##   i mse_test mse_train
## 1 1 40.02562  26.44494
## 2 2 39.97895  26.36590
## 3 3 39.88347  25.07017
## 4 4 41.20548  24.82475
```

```
## 5 5 42.19681  24.41987
## 6 6 44.16041  24.00218
```

- Plot: For a better comparison, the MSE's are graphed in the same plot:

```
library(ggplot2)

ggplot(data = df, aes( x = i)) + geom_point(data = df,
aes( x = i, y = mse_train, color="Train")) + geom_line(data = df,
aes( x = i, y = mse_train, color="Train")) + geom_point(data = df,
aes( x = i, y = mse_test, color="Test")) + geom_line(data = df, aes( x = i,
y = mse_test, color="Test")) + labs(x="i", y="MSE", colour="Dataset")
```



According to the plots, the best model is the one with $i = 3$ (third degree polynomial):

$$\text{Moisture} = \beta_0 + \beta_1 Protein + \beta_2 Protein^2 + \beta_3 Protein^3$$

We see this from the *Test dataset* plot: the lowest MSE (39.88347) is the one we get when $i = 3$.

In order to interpret this plot, first we will look at the Training dataset line, where it can be seen that as the complexity increases the MSE decreases. In other words, the larger i, the closer to the training data points the model fits and the smaller the mean squared error. But if we look at the Test dataset line, we can observe that from i = 1 to i = 3 the MSE decreases a little bit (the increase in the complexity of the model is good because the predictions obtain are better every time) but for values of i higher than 3, the MSE increases meaning that the model might be too complex and that it overfits.

In terms of bias-variance tradeoff, we can interpret the combination of both plots by observing that when the model is less complex (when i is small), the bias (expected value of the difference between the fitted and the real values) is high and the variance is "low" while when i is greater, the variance is higher and the bias lower. So the optimal model complexity is the one obtained by reducing variance at the cost of introducing some bias, and this happens when i = 3.

4. Perform variable selection of a linear model in which Fat is response and Channel1-Channel100 are predictors by using stepAIC. Comment on how many variables were selected.

```
library(MASS)
df = data[-c(1, 103, 104)]
model<- lm(Fat ~ . , data = df)
fit <- stepAIC(model, direction = "both", trace = F)
```

We used the *direction* argument to indicate that the we want the stepwise method to combine backward and forward selection.

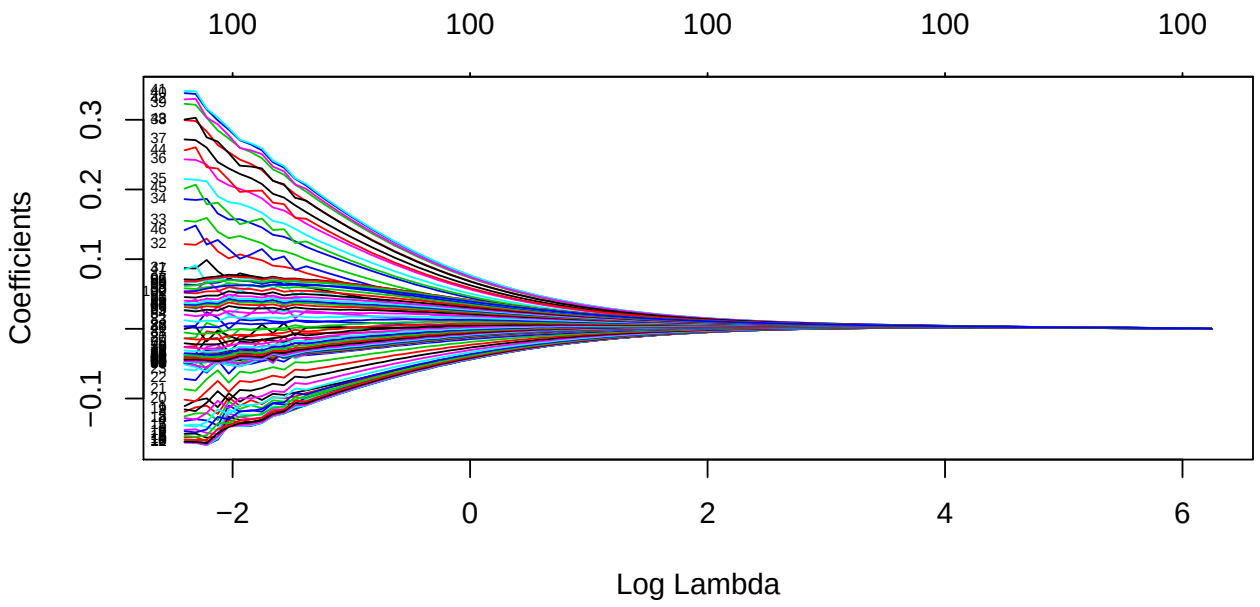
```
length(fit$coefficients)
```

```
## [1] 64
```

From `length(fit$coefficients)` we can observe that $64 - 1 = 63$ variables have been selected and the model.

5. Fit a Ridge regression model with the same predictor and response variables. Present a plot showing how model coefficients depend on the log of the penalty factor λ and report how he coefficients change with λ.

```
# First we have to scale de variables.
response <- scale(df[,101])
covariates <- scale(df[,1:100])

# Now we can fit the ridge regression model.
library(glmnet)
ridge <- glmnet(as.matrix(covariates), response, alpha=0, family="gaussian")
plot(ridge, xvar="lambda", label=TRUE)
```



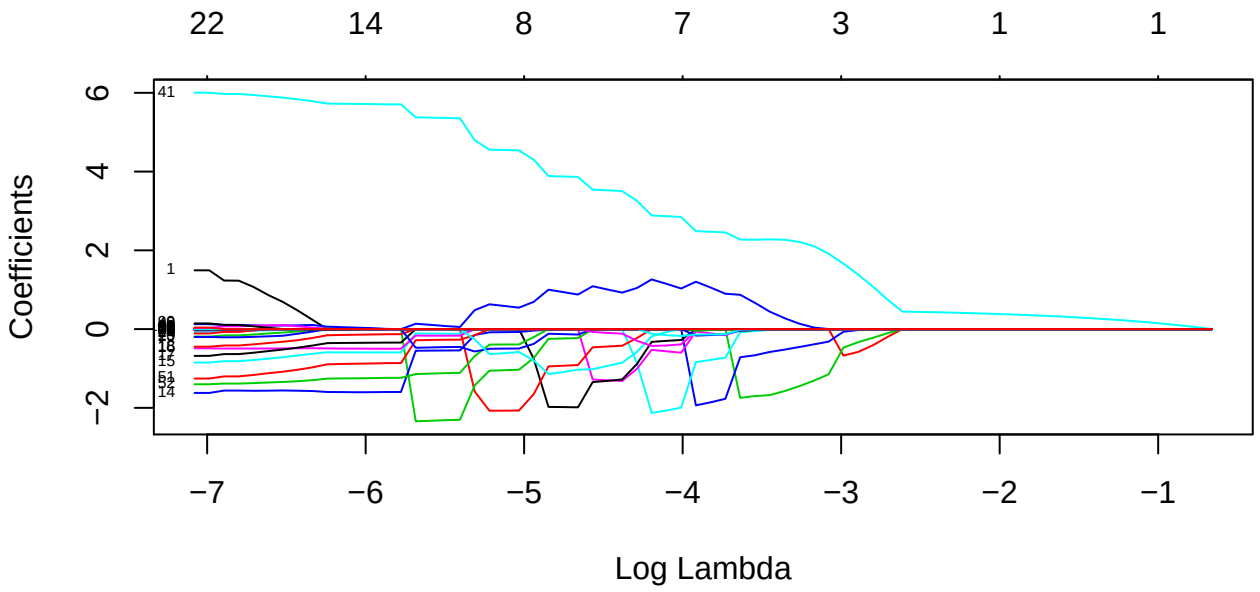
Lambda is a tuning parameter that tries to penalize the coefficients in order to get only the important variables and not overfit. Thus, it shrinks coefficients towards zero for non-important variables.

In the plot we see different combinations of lambdas on 100 variables. Ridge regression does not remove any variables, it just shrinks their coefficients. We see that as the lambda increases, all the coefficients converge to zero and that's why we don't choose very big lambdas because then we will have underfit and the coefficients will be smaller than they ought to be to get the best predictive accuracy.

6. Repeat step 5 but fit LASSO instead of the Ridge regression and compare the plots from

steps 5 and 6. Conclusions?

```
lasso <- glmnet(covariates, response, alpha=1, family="gaussian")
plot(lasso, xvar="lambda", label=TRUE)
```

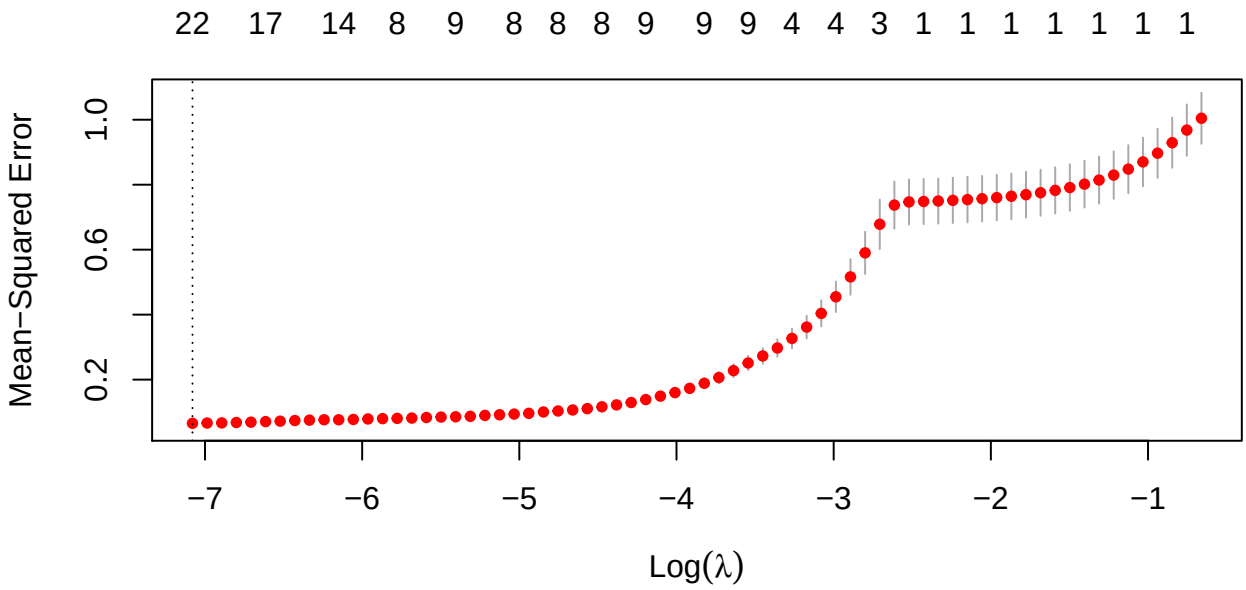


The main difference between LASSO and ridge regression is that in LASSO some coefficients can become zero and eliminated from the model, whereas in ridge regression there is no elimination of coefficients, just shrinkage. Larger penalties result in coefficient values closer to zero, which is the ideal for producing simpler models. LASSO produces sparse solutions as can be seen from the plot

If we choose $\log \lambda = 0$, for example, almost all the coefficients in the Ridge Regression plot are non-zero while all the coefficients in the LASSO plot are 0.

7. Use cross-validation to find the optimal LASSO model (make sure that case $\lambda = 0$ is also considered by the procedure).

```
lambdas_to_try<- append(lasso$lambda,0)

lasso_cv <- cv.glmnet(as.matrix(covariates),response,alpha = 1,
                      lambda =lambdas_to_try, family="gaussian")
plot(lasso_cv)
```



In the plot, the numbers in the top are the number of nonzero coefficient estimates. The lowest point in the curve indicates the optimal lambda and this exact value is:

```
lasso_cv$lambda.min #  value of lambda that gives minimum mean cross-validated error.
```

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

Also, we can consider `lambda.1se` which is the "largest value of lambda such that error is within 1 standard error of the minimum" so, it eliminates some variables while still achieving comparable MSE. (We didn't consider this value in either the group report nor my report) This value is:

```
lasso_cv$lambda.1se
```

```
## [1] 0.0008421867
```

We can compare both values and its results in the following table (from the cv.glmnet() function):

```
lasso_cv
```

```
##
## Call:  cv.glmnet(x = as.matrix(covariates), y = response, lambda = lambdas_to_try,      alpha = 1, fa
##
## Measure: Mean-Squared Error
##
##        Lambda Measure       SE Nonzero
## min 0.0000000 0.05932 0.006345     100
## 1se 0.0008422 0.06538 0.006221      22
```

So in the model using the optimal lambda ($\lambda = 0$) we will have all the variables.

8. Compare the results from steps 4 and 7.

In question 4 we used `stepAIC()` to perform the stepwise method with both backward and forward types of selection variables and as result we obtained a model with 63 variables while in question 7 the `cv.glmnet()` function has been used to perform cross-validation and find the optimal LASSO model, and as a result a model with $\lambda = 0$ and all 100 variables has been obtained.

To compare these models, we will calculate the AIC values for each one.

```
# AIC value for the stepAIC() optimal model obtained:
AIC_step <- AIC(fit)
```

```r
# LASSO model fitted with optimal lambda value
fit_lasso_cv <- glmnet(as.matrix(covariates), response, lambda = lasso_cv$lambda.min)

# AIC for the CV lasso model:
tLL <- fit_lasso_cv$nulldev - deviance(fit_lasso_cv)
k <- fit_lasso_cv$df
n <- fit_lasso_cv$nobs
AIC_lasso <- -tLL+2*k+2*k*(k+1)/(n-k-1)

# COMPARISON TABLE:
aics <- cbind("AIC" = c(AIC_step, AIC_lasso))
rownames(aics) <- c("step AIC", "lasso cross_validated")
print(aics)
```

```
##                              AIC
## step AIC                707.6913
## lasso cross_validated 173.1461
```

We can observe that the AIC associated to the stepAIC model is much greater, even though that model is quite more simple than the cross-validated lasso because in lasso no parameters have been dropped out.

Topic 2.

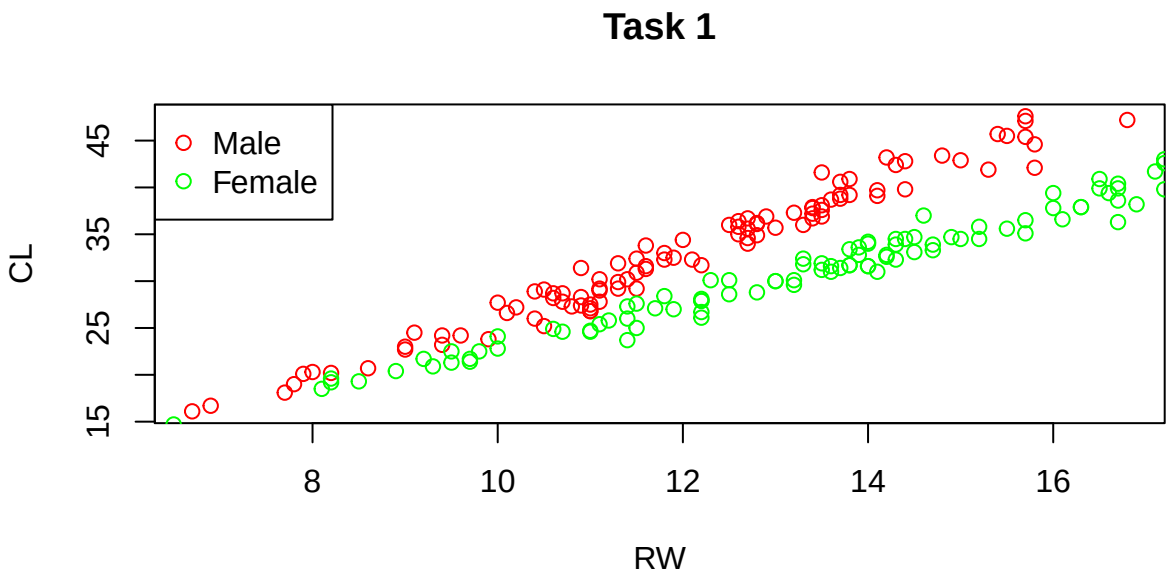
Assignment 1. LDA and logistic regression.

The data file australian-crabs.csv contains measurements of various crabs, such as Frontal lobe, Rear width and others.

1. Use australian-crabs.csv and make a scatterplot of carapace length (CL) versus rear width (RW) where observations are colored by Sex. Do you think that this data is easy to classify by linear discriminant analysis? Motivate your answer.

Yes, the data for Male and Female are nicely split and the LDA boundary can be placed between them. The distributions have similiar shapes but with different angles.

```r
data = read.csv("australian-crabs.csv", sep=";")
CL = data$CL
RW= data$RW
sex = data$sex
sexMale = which(sex %in% "Male")
sexFemale = which(sex %in% "Female")

{plot(RW[sexMale], CL[sexMale], col = "red", xlab = "RW", ylab = "CL", main = "Task 1")
points (RW[sexFemale], CL[sexFemale] , col = "green")
legend("topleft", pch = 1, col = c("red", "green"), legend = c("Male", "Female"))}
```




```
# Yes, data groups are nicely split and the LDA boundary can be placed nicely.
```

2. Make LDA analysis with target Sex and features CL and RW and proportional prior by using lda() function in package MASS. Make a scatter plot of CL versus RW colored by the predicted Sex and compare it with the plot in step 1. Compute the misclassification error and comment on the quality of fit.

With LDA we see that a few of the data points shift sex. The points that shifted are located where the groups are connected. The fit is good. The misclassification error rate is 0.035. Confusion matrix:

```
library(MASS)
model = lda(formula = sex~CL + RW, data = data)
pred_model = predict(object = model, newdata = data, type = "response")

pred_male = pred_model$class == "Male"
pred_female = pred_model$class == "Female"

# Misclassification
misclass=function (X,X1){
  n=length (X)
  return(1-sum( diag (table (X,X1)))/n)
}

# Confusion matrix and misclass . error rate = 0.035
compare = 1:length(sex)
compare[pred_male ] = "Male"
compare[pred_female] = "Female"
misclass(compare, sex)
```

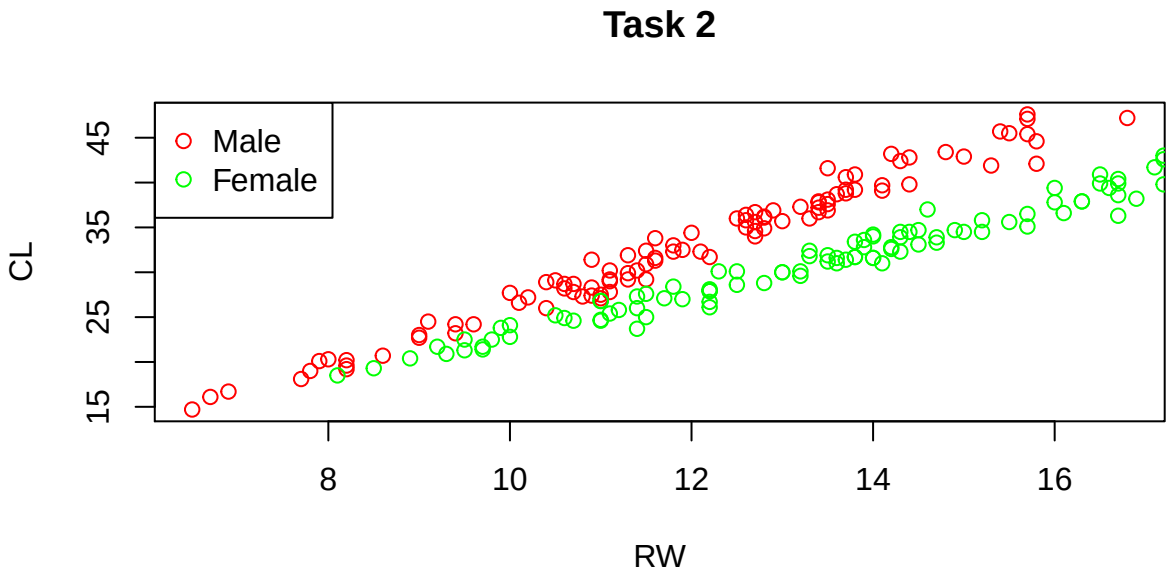
```
## [1] 0.035
```

```
table(compare, sex)
```

```
##         sex
## compare  Female Male
##   Female     97    4
##   Male        3   96
```

Figure 2: LDA implemented, a few points changed sex for task 2.

```
{plot(RW[pred_male], CL[pred_male], col = "red", xlab = "RW", ylab = "CL", main = "Task 2")
points (RW[pred_female] , CL[pred_female] , col = "green")
legend("topleft", pch = 1, col = c("red", "green"), legend = c("Male", "Female"))}
```



3. Repeat step 2 but use priors p(male)=0.8 and p(Female)=0.1 instead. How did the classification result change and why?

With the priors p(Male) = 0.9,p(Female) = 0.1 we instead have a lot more points that shift sex from Female to Male with the misclassification error rate of 0.08. The prior probabilities are the probabilities of an observation coming from a particular group. Because the prior probability for Male is higher than the prior probability for Female, the function will make it more likely that an observation will be classified as a Male. The confusion matrix is:

```
library(MASS)
model=lda(formula=sex~CL+RW,data=data,prior=c(0.1,0.9))

pred_model = predict (object = model , newdata = data , type = " response")
pred_male = pred_model$class == "Male"
pred_female = pred_model$class == "Female"

# Confusion matrix and misclass . error rate = 0.08
compare = 1:length(sex)
compare[pred_male] = "Male"
compare[pred_female] = "Female"
misclass(compare, sex)
```

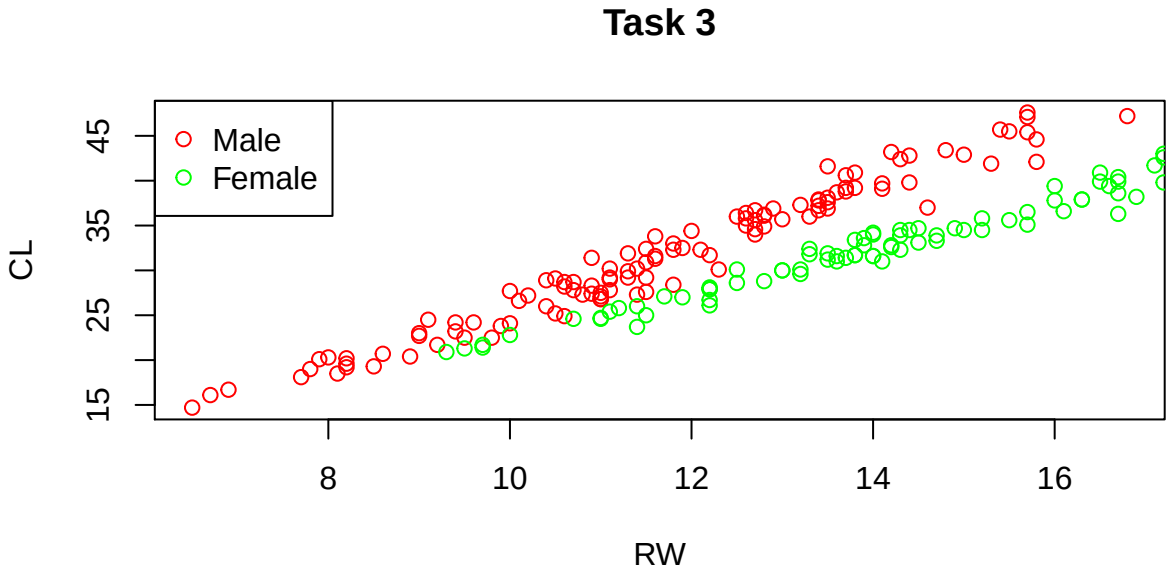
```
## [1] 0.08
```

```
table(compare, sex)
```

```
##         sex
## compare  Female Male
##   Female     84    0
##   Male       16  100
```

Figure 3: LDA with added priors, many points shift sex for task 3.

```
{plot(RW[pred_male], CL[pred_male], col = "red", xlab = "RW", ylab = "CL", main = "Task 3")
points (RW[ pred_female ] , CL[ pred_female ] , col = "green")
legend("topleft", pch = 1, col = c("red", "green"), legend = c("Male", "Female"))}
```



4. Make a similar kind of classification by logistic regression (use function glm()), plot the classified data and compute the misclassification error. Compare these results with the LDA results. Finally, report the equation of the decision boundary and draw the decision boundary in the plot of the classified data.

With the glm() function we get a misclassification error rate of 0.035 which is the same as for task 2. Certain points have, compared to task 3 with LDA, shifted from Male to Female. The actual points that are Male and Female are not the same as in task 2, even though the misclassification error rate is the same. The equation used for the decision boundary is

$$boundary = RW \cdot x + CL \cdot y + intercept$$

where y in R is

$$y = (boundary - mcoef[3] \cdot x - mcoef[1])/mcoef[2]$$

where the elements in m coef are the coefficients of the glm() model. The confusion matrix is:

```
# Task 4
model=glm(formula=sex~CL+RW,family="binomial",data= data )
pred_model = predict(object = model , newdata = data , type = "response")
pred_male = pred_model > 0.5
pred_female = pred_model <= 0.5

boundary = 0.5 #boundary = RW*x + CL*y + intercept
m_coef = model$coefficients
x = c(6, 18)
y = (boundary - m_coef[3] * x - m_coef[1]) / m_coef[2]

# Confusion matrix and misclass . error rate = 0.035
compare = 1:length(sex)
compare[pred_male] = "Male"
```

```
compare[pred_female] = "Female"
misclass(compare, sex)
```

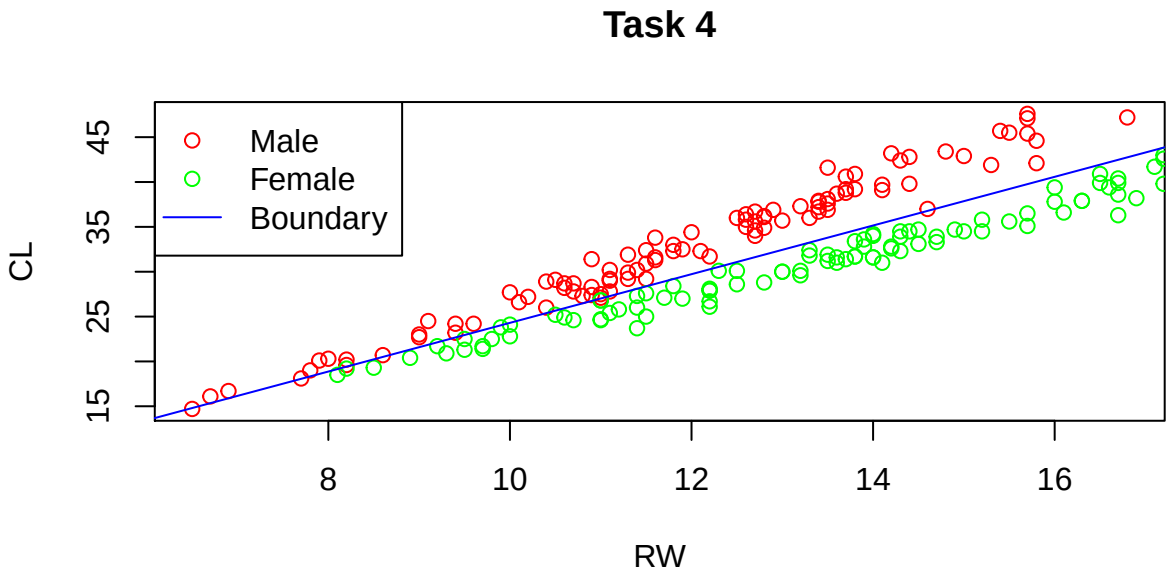
```
## [1] 0.035
```

```
table(compare, sex)
```

```
##         sex
## compare  Female Male
##   Female     97    4
##   Male        3   96
```

Figure 4: Using the glm() function instead and also adding the decision boundary for task 4.

```
{plot(RW[pred_male], CL[pred_male], col = "red", xlab = "RW", ylab = "CL", main = "Task 4")
points (RW[ pred_female ] , CL[ pred_female ] , col = "green")
legend("topleft", pch = c(1, 1, NA), lty = c(NA, NA, 1), col = c("red" , "green" , "blue") , legend = c
lines(x, y, col = "blue")}
```



Assignment 2. Analysis of credit scoring.

The data file creditscoring.xls contains data retrieved from a database in a private enterprise. Each row contains information about one customer. The variable good/bad indicates how the customers have managed their loans. The other features are potential predictors. Your task is to derive a prediction model that can be used to predict whether or not a new customer is likely to pay back the loan.

1. Import the data to R and divide into training/validation/test as 50/25/25: use data partitioning code specified in Lecture 1e.

```
# Changing the version:
RNGversion('3.5.1')

# Importing the data:
library(readxl)
data <- read_excel("creditscoring.xls")
data$good_bad <- as.factor(data$good_bad)
```

```
# Dividing it into training, validation and test sets:
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]

id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.25))
valid=data[id2,]

id3=setdiff(id1,id2)
test=data[id3,]
```

2. Fit a decision tree to the training data by using the following measures of impurity (a. Deviance, b. Gini index) and report the misclassification rates for the training and test data. Choose the measure providing the better results for the following steps.

We have fitted two decision trees using the function `tree()` from the package **tree**, choosing `split = "deviance"` and `split = "gini"` in each one.

```
library(tree)

# 2.1. Fitting the models.
fit_dev = tree(good_bad~., data=train, split = "deviance") # Impurity measure: Deviance
fit_gini = tree(good_bad~., data=train, split = "gini") # Impurity measure: Gini index

# 2.2. Predictions.
## For the test data:
fitted_dev_test <- predict(fit_dev, newdata = test, type = "class")
fitted_gini_test <- predict(fit_gini, newdata = test, type = "class")

## For the training data:
fitted_dev_train <- predict(fit_dev, newdata = train, type = "class")
fitted_gini_train <- predict(fit_gini, newdata = train, type = "class")

# 2.3. Misclassification rates.
## For the test data:
mis_rate_dev_test <- mean(fitted_dev_test != test$good_bad)
mis_rate_gini_test <- mean(fitted_gini_test != test$good_bad)

## For the training data:
mis_rate_dev_train <- mean(fitted_dev_train != train$good_bad)
mis_rate_gini_train <- mean(fitted_gini_train != train$good_bad)
```

- Missclassification rates for the fitted tree using the deviance:

```
data.frame(cbind("Test data"=mis_rate_dev_test,"Training data"=mis_rate_dev_train))
```

```
##   Test.data Training.data
## 1     0.268         0.212
```

- Missclassification rates for the fitted tree using the Gini index:

```
data.frame(cbind("Test data"=mis_rate_gini_test,"Training data"=mis_rate_gini_train))
```

```
##   Test.data Training.data
```

```
## 1     0.368          0.24
```

Given the missclassification rates, the best performance is the one obtained by the tree fitted with the deviance as measure of impurity. For both trees the rate for the training data is quite similar but the error rate for the test data is smaller in the deviance case.

For this reason, from now on we will use the tree with `deviance`.

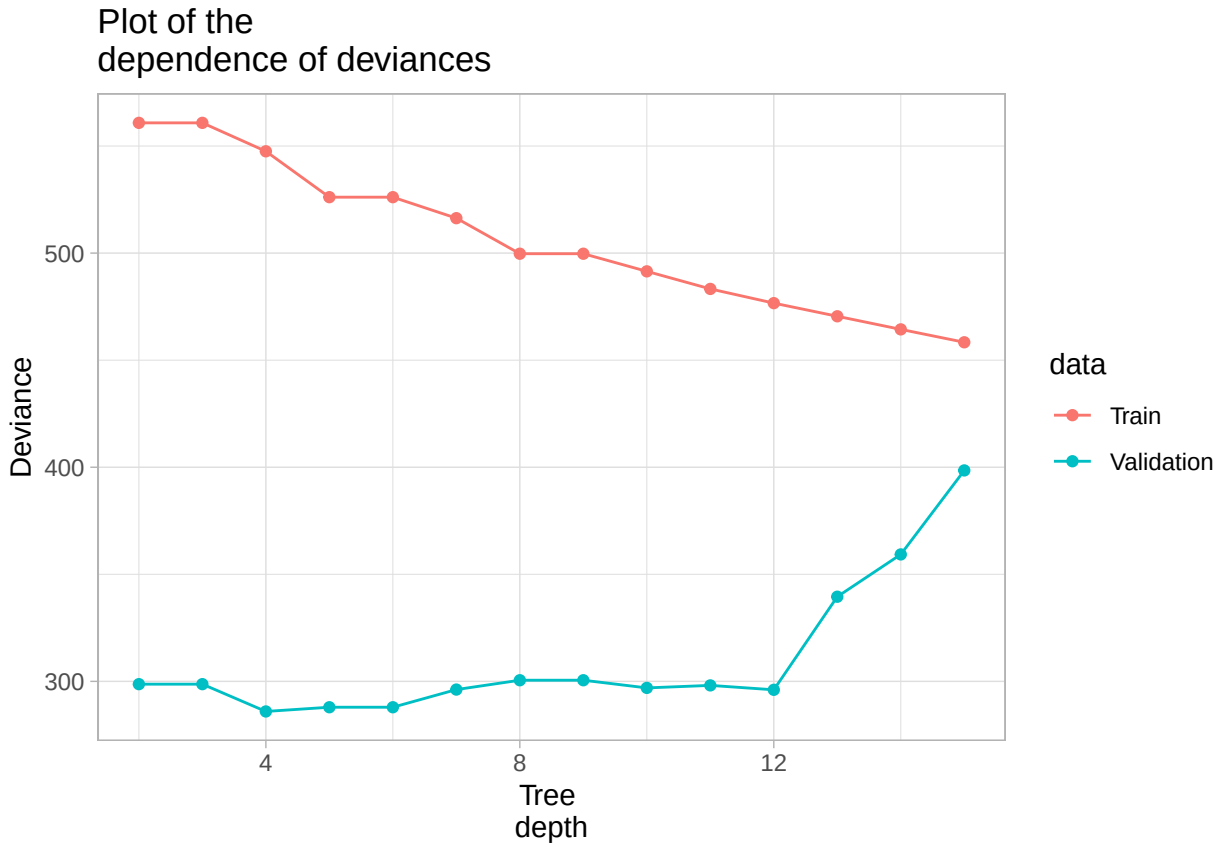
3. Use training and validation sets to choose the optimal tree depth. Present the graphs of the dependence of deviances for the training and the validation data on the number of leaves. Report the optimal tree, report it's depth and the variables used by the tree. Interpret the information provided by the tree structure. Estimate the misclassification rate for the test data.

In `summary(fit)` we have obtained that the number of terminal nodes is 15, so we have used the `prune.tree()` function with the fitted model (using the deviance) in the **tree** argument and values $2, 3, \ldots, 15$ in the **best** argument. So, 14 different trees with $2, 3, \ldots, 15$ terminal nodes in each one has been considered and the deviance of each model has been calculated with the `deviance()` function.

```
library(ggplot2)

# 3.1. Fitting the model
fit = tree(good_bad~., data=train, split = "deviance")
# summary(fit) -> Number of terminal nodes is 12

# 3.2. Pruning trees with difference depths
trainScore=rep(0,15)
testScore=rep(0,15)

for(i in 2:15) {
  prunedTree=prune.tree(fit,best=i)
  pred=predict(prunedTree, newdata=valid, type="tree")

  trainScore[i]=deviance(prunedTree)
  testScore[i]=deviance(pred)
}
```

Then, the results obtained are shown in the plot below:

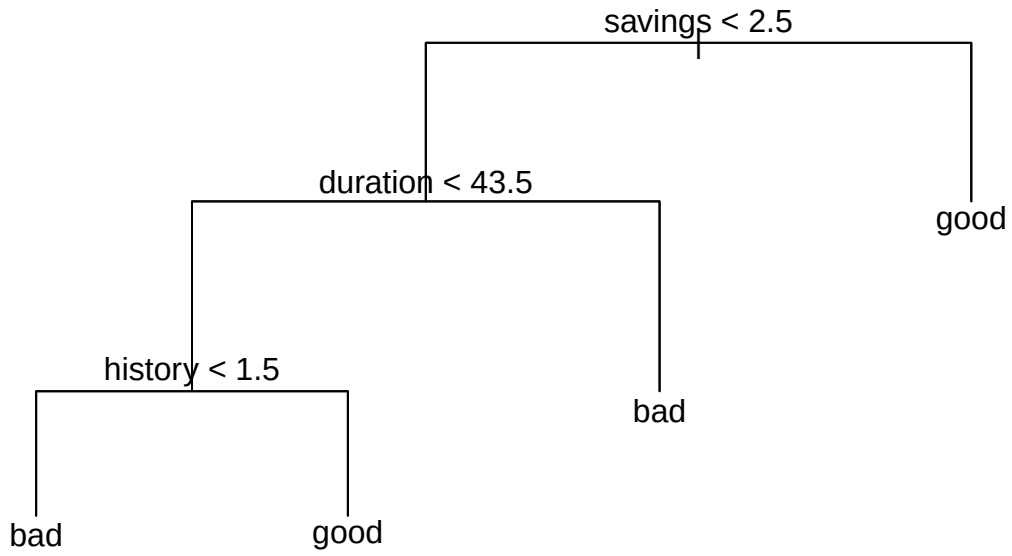
```
# 3.3. Plot of deviance vs. depth
df <- data.frame(depth=2:15, score= c(trainScore[2:15],testScore[2:15]),
                 data= rep(c("Train", "Validation"), each=14))

ggplot(df, aes(x = depth, y = score, group = data,
color = data)) + geom_point() + geom_line() + ggtitle("Plot of the
dependence of deviances") + xlab("Tree
depth") + ylab("Deviance") + theme_light()
```



Validation data graph: It appears that a tree with 4 leaves has the lowest deviance of the considered trees. More precisely the deviance when we have a tree of size 4 is 285.9425 and it's the smallest.

We can draw the optimal tree with the `plot()` function

```
# 3.4. Optimal tree.
finalTree=prune.tree(fit, best=4)
{plot(finalTree)
text(finalTree, pretty=0)}
```



This three has depth 3 and 4 terminal nodes and the variables used are: duration, history and savings.

Finally, we will estimate the misclassification rate for the test data:

```
# 3.5. Misclassification rate for the test data.
fitted_best_test=predict(finalTree, newdata=test, type="class") # predictions
# table(fitted_best_test, test$good_bad) confussion matrix
mean(fitted_best_test != test$good_bad) # error rate
```

```
## [1] 0.256
```

We have reduced the misclassification rate (compared to the tree chosen in question 2). Now, only 25.6% of the observations are classificated wrongly.

4. Use training data to perform classification using Naïve Bayes and report the confusion matrices and misclassification rates for the training and for the test data. Compare the results with those from step 3.

We have fitted the required model using the training data and the function `naiveBayes()` from the package **e1071**.

```
library(e1071)

# 4.1. Fitting the model.
fit_naive=naiveBayes(good_bad~., data=train)

# 4.2. Predictions.
fitted_test <- predict(fit_naive, newdata = test, type = "class") # For the test data
fitted_train <- predict(fit_naive, newdata = train, type = "class") # For the training data
```

- Confusion matrices.

```
table(test$good_bad, fitted_test)
```

```
##       fitted_test
##        bad good
##   bad   46   30
##   good  49  125
```

```
table(train$good_bad, fitted_train)
```

```
##       fitted_train
##        bad good
##   bad   95   52
##   good  98  255
```

- Misclassification rates.

```
cat("For the test data:", mean(fitted_test != test$good_bad),"\n")
```

```
## For the test data: 0.316
```

```
cat("For the training data:", mean(fitted_train != train$good_bad),"\n")
```

```
## For the training data: 0.3
```

The misclassification rate for the test data using Naive Bayes is bigger than the one obtained by the optimal tree computed in question three. More precisely, the misclassification rate of Naive Bayes for the test data is 0.316 whereas for the optimal tree is 0.256.

5. Use the optimal tree and the Naïve Bayes model to classify the test data by using the following principle:

$$\hat{Y} = 1 \text{ if } p(Y = \text{'good'}|X) > \pi, \text{ otherwise } \hat{Y} = 0$$

where $\pi = 0.05, 0.1, 0.15, ...0.9, 0.95$. Compute the TPR and FPR values for the two models and plot the corresponding ROC curves. Conclusion?

```
# 5.1. Classification models:
## Naive Bayes model
fit_naive=naiveBayes(good_bad~., data=train)

## Optimal tree (4 terminal nodes)
fit = tree(good_bad~., data=train, split = "deviance")
fit_optimal=prune.tree(fit, best=4)

# 5.2. Predictions:
## Naive Bayes
fitted_naive <- predict(fit_naive, newdata = test, type="raw") # type raw to get probabilities

## Optimal tree
fitted_tree <- predict(fit_optimal, newdata = test)

# 5.3. Classification using the principle.
pi <- seq(0.05, 0.95, by=0.05)

fitted_naive_class <- data.frame(matrix(ncol = length(pi), nrow = nrow(test)))
fitted_tree_class <- data.frame(matrix(ncol = length(pi), nrow = nrow(test)))
colnames(fitted_naive_class) <- pi
colnames(fitted_tree_class) <- pi

for(i in 1:length(pi)){
  fitted_naive_class[,i] <- ifelse(fitted_naive[,2] > pi[i], 1, 0)
  fitted_tree_class[,i] <- ifelse(fitted_tree[,2] > pi[i], 1, 0)
}

# 5.4. Computation of TPR and FPR: TPR=TP/(TP+FN) and FPR=FP/(FP+TN)
library(EvaluationMeasures)
real_values <- ifelse( test$good_bad == "good", 1, 0)

result <- data.frame(matrix(ncol = 4, nrow = length(pi)*2))
colnames(result) <- c("pi", "model", "TPR", "FPR")
result$pi <- rep(pi,2)
result$model <- rep(c("Naive", "Tree"),each=length(pi))

for(i in 1:ncol(fitted_naive_class)){
  result[i,3] <- EvaluationMeasures.TPR(Real= real_values,
                                        Predicted = fitted_naive_class[,i],
                                        Positive = 1)
  result[i+19,3] <- EvaluationMeasures.TPR(Real= real_values,
                                           Predicted = fitted_tree_class[,i],
                                           Positive = 1)

  result[i,4] <- EvaluationMeasures.FPR(Real= real_values,
                                        Predicted = fitted_naive_class[,i],
                                        Positive = 1)
  result[i+19,4] <- EvaluationMeasures.FPR(Real= real_values,
                                           Predicted = fitted_tree_class[,i],
                                           Positive = 1)
```

```
}
```

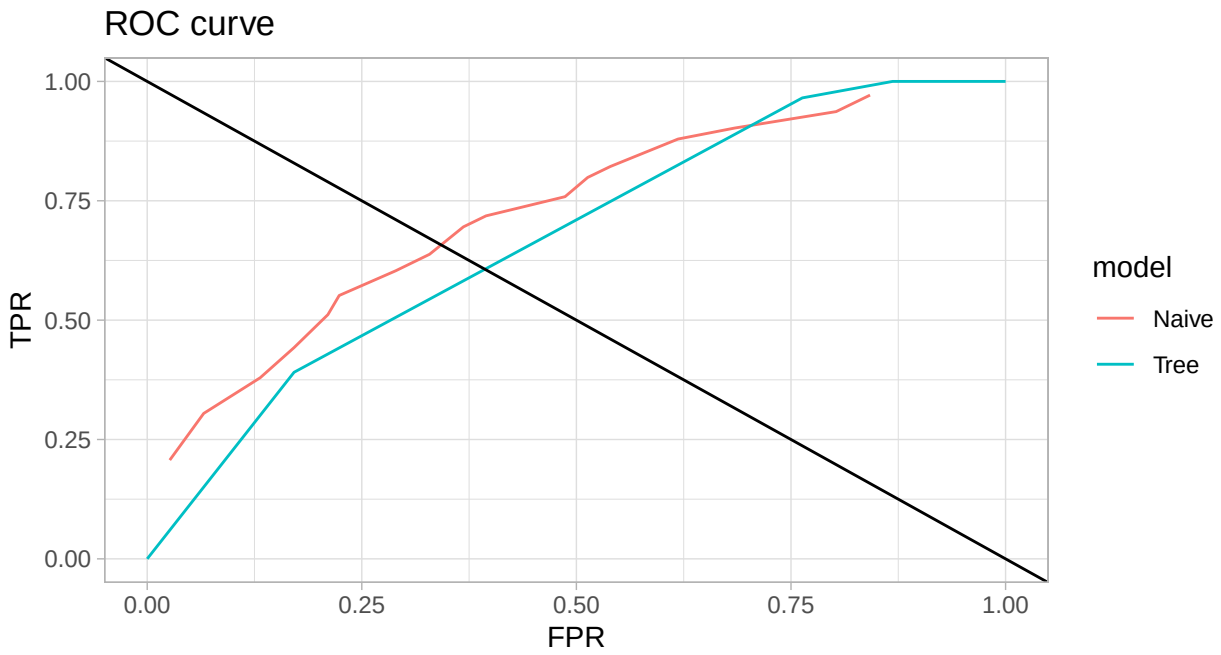
- Table TPR and FPR values:

```
cbind(result[1:19,], result[20:38,])
```

```
##       pi model    TPR    FPR   pi model    TPR    FPR
## 1   0.05 Naive 0.9713 0.8421 0.05  Tree 1.0000 1.0000
## 2   0.10 Naive 0.9368 0.8026 0.10  Tree 1.0000 1.0000
## 3   0.15 Naive 0.9253 0.7632 0.15  Tree 1.0000 1.0000
## 4   0.20 Naive 0.9023 0.6842 0.20  Tree 1.0000 0.8684
## 5   0.25 Naive 0.8793 0.6184 0.25  Tree 1.0000 0.8684
## 6   0.30 Naive 0.8218 0.5395 0.30  Tree 1.0000 0.8684
## 7   0.35 Naive 0.7989 0.5132 0.35  Tree 0.9655 0.7632
## 8   0.40 Naive 0.7586 0.4868 0.40  Tree 0.9655 0.7632
## 9   0.45 Naive 0.7356 0.4342 0.45  Tree 0.9655 0.7632
## 10 0.50 Naive 0.7184 0.3947 0.50  Tree 0.9655 0.7632
## 11 0.55 Naive 0.6954 0.3684 0.55  Tree 0.9655 0.7632
## 12 0.60 Naive 0.6379 0.3289 0.60  Tree 0.9655 0.7632
## 13 0.65 Naive 0.6034 0.2895 0.65  Tree 0.9655 0.7632
## 14 0.70 Naive 0.5517 0.2237 0.70  Tree 0.9655 0.7632
## 15 0.75 Naive 0.5115 0.2105 0.75  Tree 0.3908 0.1711
## 16 0.80 Naive 0.4425 0.1711 0.80  Tree 0.3908 0.1711
## 17 0.85 Naive 0.3793 0.1316 0.85  Tree 0.0000 0.0000
## 18 0.90 Naive 0.3046 0.0658 0.90  Tree 0.0000 0.0000
## 19 0.95 Naive 0.2069 0.0263 0.95  Tree 0.0000 0.0000
```

- Plot of FPR vs TPR (ROC curve).

```
ggplot(result, aes(x = FPR, y=TPR)) + geom_line(aes(colour = model)) + geom_abline(intercept=1,
slope =-1) + xlim(0,1)+ ggtitle("ROC curve") + theme_light()
```



We know that the best classifier is the one that has the biggest area under the ROC curve. From the plot we can see that the greatest ares is under the Naive Bayes curve, therefore we can conclude that compared to the decision tree, Naive Bayes classifier is a better model. For the same False Positive rate, in most cases

Bayes model has a higher True Positive Rate.

6. Repeat Naïve Bayes classification as it was in step 4 but use the following loss matrix:

$$L = \begin{pmatrix} 0 & 1 \\ 10 & 0 \end{pmatrix}$$

and report the confusion matrix for the training and test data. Compare the results with the results from step 4 and discuss how the rates has changed and why.

Rule to classify an observation:

$$\frac{P(C_1|x)}{P(C_2|x)} > \frac{L_{21}}{L_{12}} \rightarrow \text{predict } y \text{ as } C_1$$

In our case:

$$\frac{P(\text{“good"}|x)}{P(\text{“bad"}|x)} > 10 \rightarrow \text{predict } y \text{ as “good"}$$

```
# 6.1. Naive Bayes model.
fit =naiveBayes(good_bad~., data=train)

# 6.2. Predictions
# Probabilities (type= "raw")
fitted_prob_test <- predict(fit_naive, newdata = test, type="raw") # test data
fitted_prob_train <- predict(fit_naive, newdata = train, type="raw") # training data

# 6.3. Classification using the loss matrix.
fitted_class_test <- ifelse(fitted_prob_test[,2]/fitted_prob_test[,1] > 10, "good", "bad")
fitted_class_train <- ifelse(fitted_prob_train[,2]/fitted_prob_train[,1] > 10, "good", "bad")
```

- Confusion matrices.

```
table(test$good_bad, fitted_class_test)
```

```
##       fitted_class_test
##        bad good
##   bad   71    5
##   good 122   52
```

```
table(train$good_bad, fitted_class_train)
```

```
##       fitted_class_train
##        bad good
##   bad  137   10
##   good 263   90
```

In this case, the number of good customers predicted as good has been reduced in comparison to the confusion matrices in question (4) which happened because when using the loss matrix, our rule to classify a customer as “good” is “the propability to classify a customer as good has to be 10 times higher than the probability of being bad”. So we are being more strict.

- Misclassification rates.

```
cat("For the test data:", mean(fitted_class_test != test$good_bad),"\n")
```

```
## For the test data: 0.508
```

```
cat("For the training data:", mean(fitted_class_train != train$good_bad),"\n")
```

```
## For the training data: 0.546
```

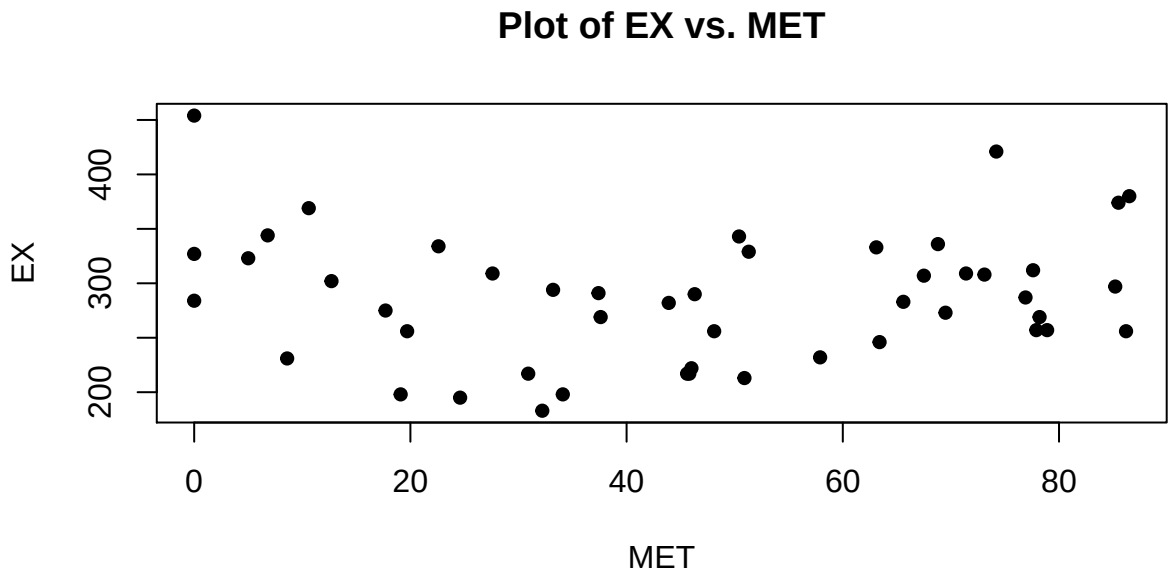
The misclassification rate for the train data is bigger when following the loss matrix than it was on question 4. Now the misclassification rate for the train data is 0.546 whereas previously it was 0.3.

The misclassification rate for the test data is also bigger when following the loss matrix than it was on question 4. Now the misclassification rate for the train data is 0.508 whereas previously it was 0.316.

Assignment 3. Uncertainty estimation.

1. Reorder your data with respect to the increase of MET and plot EX versus MET. Discuss what kind of model can be appropriate here. Use the reordered data in steps 2-5.

```
# Importing the data:
data <- read.table("State.csv", header=TRUE, sep=";", dec=",")

# Data reordered by MET
data <- data[order(data$MET),]

# Plot
plot(data$MET, data$EX, xlab="MET", ylab="EX", main="Plot of EX vs. MET", pch=19, cex=0.85)
```



The observations (black dots) are quite sparsed in the plot, so a linear regression model wouldn't be appropiate. But if we want to perform some kind of regression, quadratic model would be a viable option.

However, the best option seems to be fitting a regression tree, because the target variable is numeric.

2. Use package tree and fit a regression tree model with target EX and feature MET in which the number of the leaves is selected by cross-validation, use the entire data set and set minimum number of observations in a leaf equal to 8 (setting minsize in tree.control). Report the selected tree. Plot the original and the fitted data and histogram of residuals. Comment on the distribution of the residuals and the quality of the fit.

In order to fit the regression tree with at least 8 observations in each leaf, it has been used the option `minsize` in the control argument. Then, the `cv.tree` function has been used to selet the optimal number of leaves by cross-validation. This value can be seen executing the following code: `cv.tree$size[which(cv.tree$dev==min(cv.tree$dev))]`.

So the optimal number of leaves is 3 (the size of the tree) and we can report this tree:

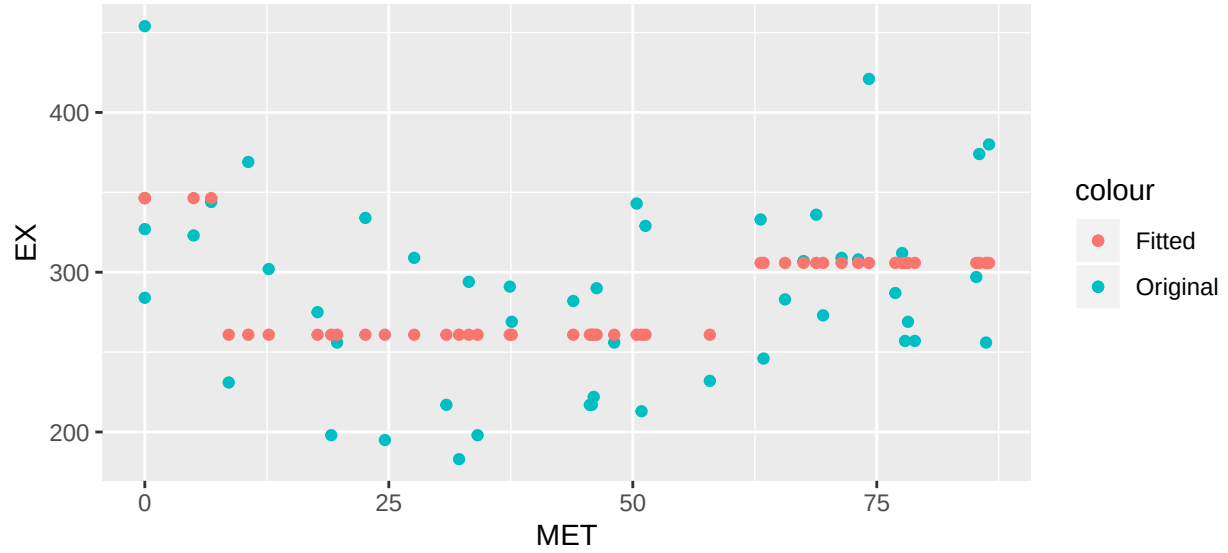
```
# 2.1. Fitting a regression tree model.
fit <- tree(EX~ MET , data=data, control=tree.control(nobs= nrow(data), minsize = 8))
```

```r
# 2.2. Selecting the optimal number of leaves by cross-validation.
set.seed(12345)
cv.tree=cv.tree(fit)

# 2.3. Fitting the optimal regression tree.
best.size <- cv.tree$size[which(cv.tree$dev==min(cv.tree$dev))]
fit_optimal <- prune.tree(fit, best=best.size)
summary(fit_optimal)
```

```
##
## Regression tree:
## snip.tree(tree = fit, nodes = 7:6)
## Number of terminal nodes:  3
## Residual mean deviance:  2698 = 121400 / 45
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -77.88  -43.88   -4.88    0.00   30.13  115.20
```

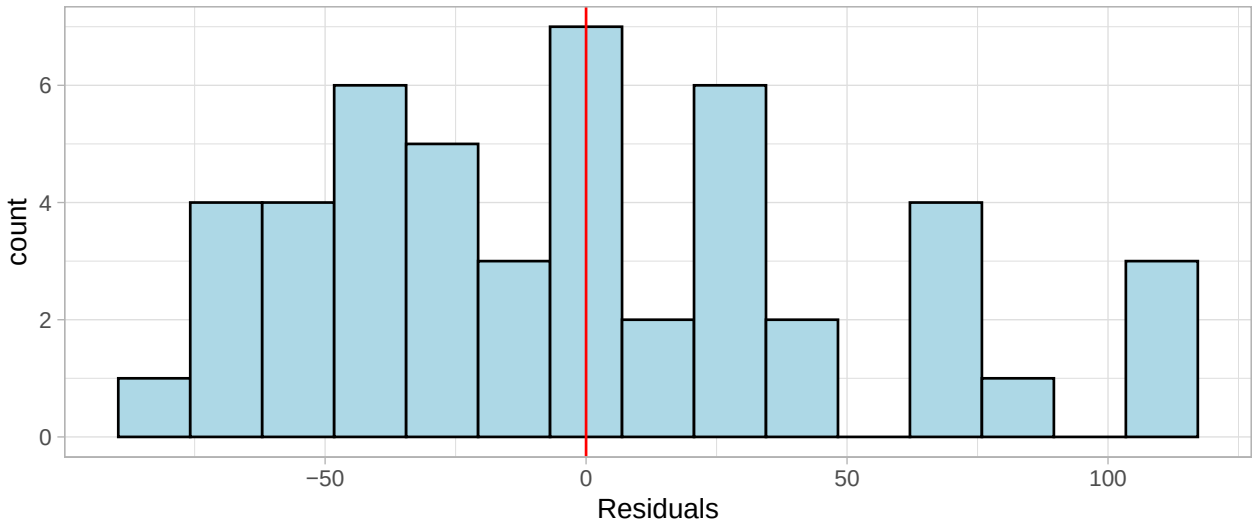
We will plot the original and the fitted data:

```r
# 2.4. Plot of the original and the fitted data.
ggplot()+ geom_point(mapping = aes(x=data$MET,y=data$EX,color='Original'))+
  geom_point(mapping = aes(x=data$MET,y=predict(fit_optimal,newdata = data),color='Fitted'))+
  labs(y="EX",x='MET')
```



The fit of the tree model doesn't look so good. The histogram of the residuals is the following:

```r
# 2.5 Histogram of residuals.
pred <- predict(fit_optimal, newdata = data)
resid <- data$EX-pred
df <- as.data.frame(resid)

ggplot(df, aes(x=resid)) + geom_histogram(color="black", fill="lightblue", bins=15)+
  theme_light() + xlab("Residuals")+ geom_vline(xintercept = 0, col="red")
```



Looking at the histogram of residuals we can conclude that they don't look normal at all.

3. Compute and plot the 95% confidence bands for the regression tree model from step 2 (fit a regression tree with the same settings and the same number of leaves as in step 2 to the resampled data) by using a non-parametric bootstrap. Comment whether the band is smooth or bumpy and try to explain why. Consider the width of the confidence band and comment whether results of the regression model in step 2 seem to be reliable.

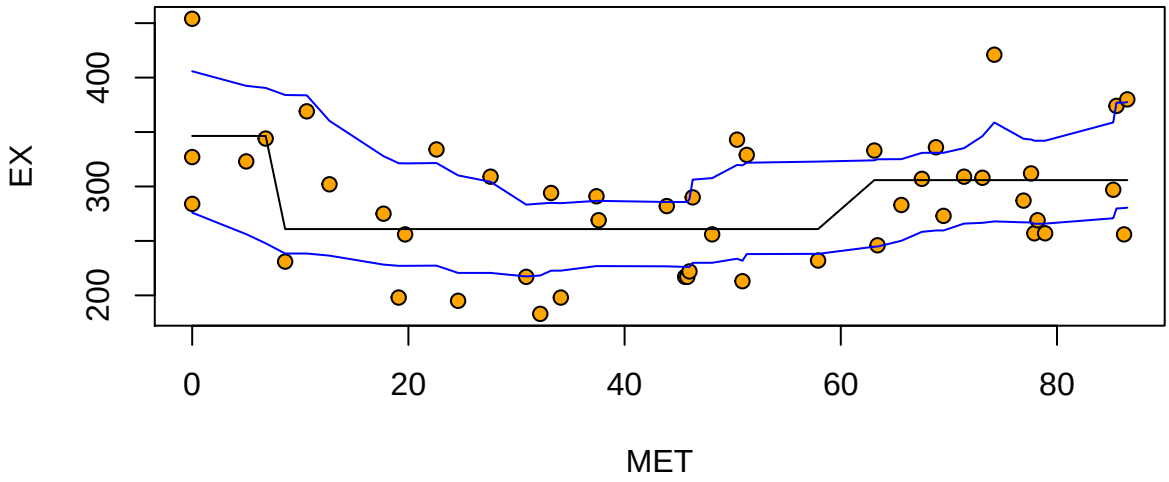
We have used the `boot()` function in order to generate 1000 bootstrap replicates. The statistic of interest in our case is the vector of predictions obtained in each regression tree (fitted with all the bootstrap sample).

Once we have had the 1000 vectors of predictions the `envelope()` function has been used to compute the confidence bands.

Then we can plot the observations with the estimate line (in black) and the confidence intervals (blue lines):

```
# 3. Non-parametric bootstrap CI
library(boot)

# 3.1. First we need to compute bootstrap samples
f=function(data, ind){
  # extract bootstrap sample
  data1 = data[ind,]

  # fit regression tree model
  fit = tree(EX~ MET , data=data1, control=tree.control(nobs= nrow(data1), minsize = 8))
  fit_optimal <- prune.tree(fit, best=3)

  # predict values from the original data
  predictions = predict(fit_optimal, newdata=data)

  return(predictions)
}

# 3.2. And now we can make bootstrap:
res=boot(data, f, R=1000)

# 3.3. Bootstrap cofidence bands
e=envelope(res) #compute confidence bands
```

```r
# 3.4. Plotting the predicted line and 95% confidence bands
fit <- tree(EX~ MET , data=data, control=tree.control(nobs= nrow(data), minsize = 8))
cv.tree=cv.tree(fit)
fit_optimal <- prune.tree(fit, best=3)

predictions =predict(fit_optimal)

{plot(data$MET, data$EX, xlab="MET", ylab="EX", pch=21, bg="orange")
points(data$MET,predictions,type="l") #plot fitted line
#plot cofidence bands
points(data$MET,e$point[2,], type="l", col="blue")
points(data$MET,e$point[1,], type="l", col="blue")}
```



We observe that the bands' shapes are bumpy: the lines are rising and falling irregularly.

Also, the width of the confidence band is quite wide, but it does not include most of the points. We only have 48 observations and many of them are outside the 95% confidence band which is an indication that our model is not good, as we saw in step (2). If the model would fit well the data, then 95% of the observations would be inside of the confidence bands.

The width of the confidence band is wide because we don't have many observations. If we had considered more data points, the confidence bands would be near the curve (fitted black line). Generally, larger sample sizes tend to produce narrower confidence intervals.

4. Compute and plot the 95% confidence and prediction bands the regression tree model from step 2 by using a parametric bootstrap, assume $Y \sim N(\mu_i, \sigma^2)$ where μ_i are labels in the tree leaves and σ^2 is the residual variance. Consider the width of the confidence band and comment whether results of the regression model in step 2 seem to be reliable. Does it look like only 5% of data are outside the prediction band? Should it be?

Steps followed to compute parametric bootstrap confidence and prediction bands:

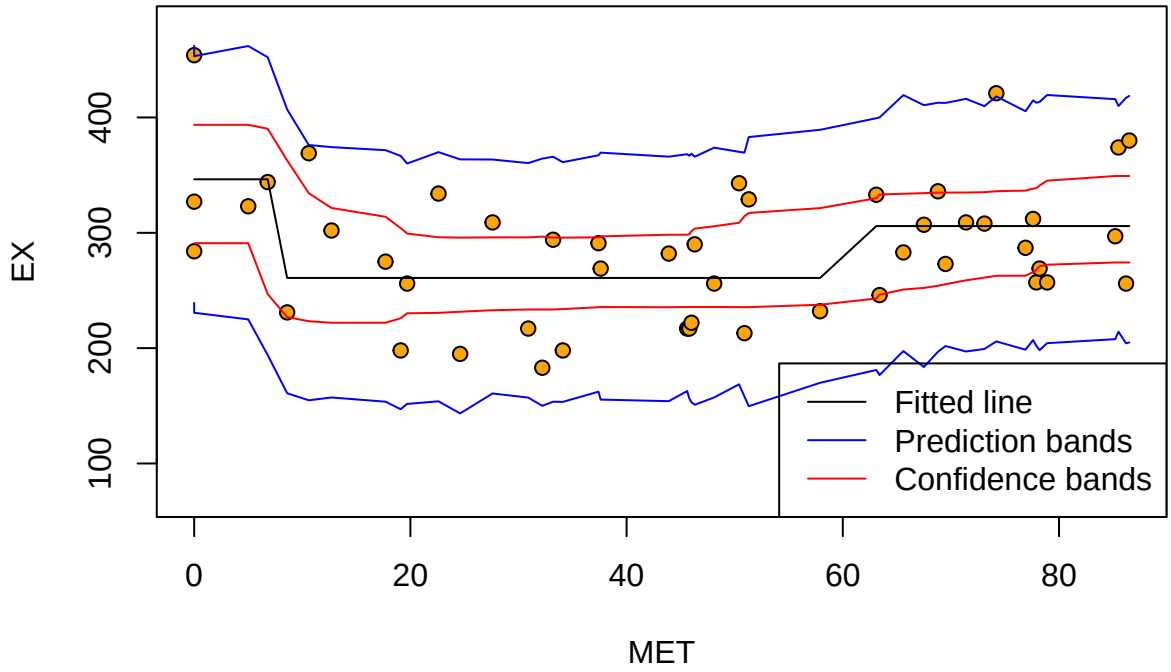
1. Compute value MLE that estimates model parameters from the data: the maximum likelihoood estimates of the parameters in our case are computed with the function `prune.tree()` because our tree regression model is pruned with only 3 leaves.
2. Write function "ran.gen" that depends on data and MLE and which generates new data. This function returns new values for the target variable "EX", generated from a normal distribution of μ = predictions of MLE and σ = residuals of MLE.
3. Write function "statistic" that depend on data which will be generated by ran.gen and should return the estimator. In this case two different statistics have been computed: prediction and confidence.

4. Make bootstrap for each statistic. We have used the `boot()`function for this in order to generate 1000 parametric bootstrap replicates of each statistic.
5. Compute the confidence and prediction bands with `envelope()`.

So, the plot of the 95% confidence and prediction bands:

```
# 4. Parametric bootstrap confidence and prediction bands
# 4.1. Compute value mle that estimates model parameters from the data:
fit <- tree(EX~ MET , data=data, control=tree.control(nobs= nrow(data), minsize = 8))
cv.tree=cv.tree(fit)
mle <- prune.tree(fit, best=3)

# 4.2. Write function ran.gen that depends on data and mle and which generates new data.
rng=function(data, mle) {
  data1=data.frame(Ex=data$EX, MET=data$MET)
  n=length(data$EX)

  #generate new EX (mle$residuals doesn't exist for prune.tree)
  data1$EX=rnorm(n,predict(mle, newdata=data1),sd(residuals(mle)))
  return(data1)
}

# 4.3. Write function "statistic" that depend on data which will be generated by ran.gen and should ret
# 4.3.1. Bootstrap samples for prediction bands
f_pred=function(data1){
  #fit linear model
  fit = tree(EX~ MET , data=data1, control=tree.control(nobs= nrow(data1), minsize = 8))
  fit_optimal <- prune.tree(fit, best=3)

  #predict values for all MET values from the original data
  n=length(data$EX)
  predictions_pred = rnorm(n,predict(fit_optimal, newdata=data), sd(residuals(mle)))
  return(predictions_pred)
}

# 4.3.2. Bootstrap samples for condicence bands
f_conf=function(data1){
  # fit regression tree model
  fit = tree(EX~ MET , data=data1, control=tree.control(nobs= nrow(data1), minsize = 8))
  fit_optimal <- prune.tree(fit, best=3)

  # predict values from the original data
  predictions_conf = predict(fit_optimal, newdata=data)

  return(predictions_conf)
}

# 4.4 Now we can make bootstrap for each statistic:
res_pred=boot(data, statistic=f_pred, R=1000, mle=mle, ran.gen=rng, sim="parametric")
res_conf=boot(data, statistic=f_conf, R=1000, mle=mle, ran.gen=rng, sim="parametric")

# 4.5. Parametric bootstrap bands
e_pred=envelope(res_pred) # compute prediction bands
e_conf=envelope(res_conf) # compute confidence bands
```

```r
# 4.5. Plotting the predicted line and 95% confidence and prediction bands
estimated_line =predict(fit_optimal)

{plot(data$MET, data$EX, xlab="MET", ylab="EX", pch=21, bg="orange", ylim=c(70,480))
points(data$MET,estimated_line,type="l") #plot fitted line
#plot prediction bands
points(data$MET,e_pred$point[2,], type="l", col="blue")
points(data$MET,e_pred$point[1,], type="l", col="blue")
#plot confidence bands
points(data$MET,e_conf$point[2,], type="l", col="red")
points(data$MET,e_conf$point[1,], type="l", col="red")
legend("bottomright",legend=c("Fitted line","Prediction bands", "Confidence bands"),
       lty = c(1,1,1), col=c("black","blue", "red"))}
```



From the plot we observe that the prediction bands are quite wider than the confidence bands and that almost all the points are inside these bands. This is totally normal because the 95% confidence bands enclose the area that contains the true curve (at 95% confidence) while the 95% prediction bands takes in the area that one can **expect** to enclose 95% of future data points.

Also, we can see that there are only 1 point is outside the prediction bands (given that we have 48 points, that is the 1.2%), this is less than 5% and it is expected to be like this because only a maximum of 5% of new points can be outside given that the prediction bands are at 95% level.

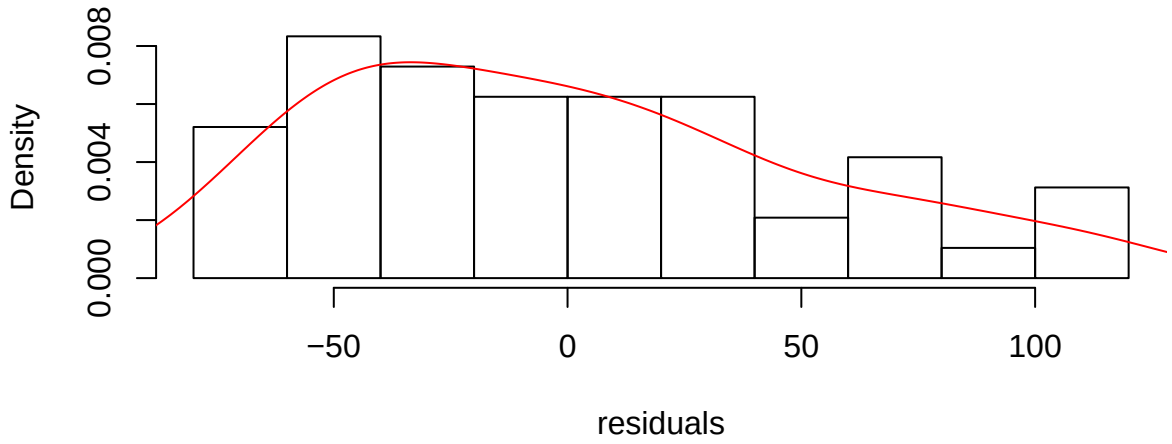
Regarding the confidence bands, we see that they are a bit smoother than the ones using non-parametric bootstap. Still, many points (more than 5%) are outside the confidence bands so, again, the results of the regression model in step (2) don't seem to be reliable.

5. Consider the histogram of residuals from step 2 and suggest what kind of bootstrap is actually more appropriate here.

If we draw the density curve in the histogram of the residuals, we obtain the following plot:

```r
dens = density(resid)
hist(resid, freq=F, xlab="residuals", main=NULL) #probability densities
```

```
lines(dens,col="red")
```



Which makes us think that a Gamma distribution could be appropiate in this case. Then, the paramteric-bootstrap assuming that the target variable (EX) has a Gamma distribution would be a better option.

Assignment 4. Principal components.

1. Conduct a standard PCA by using the feature space and provide a plot explaining how much variation is explained by each feature.

The Principal Component Analysis for all the different near-infrared spectra levels (variables 750 to 1000 from the original data frame) has been performed with the `prcomp()` function. In the following table it is possible to see how much variation is explained by each of the first five principal components as well as the cumulative proportion of variance.

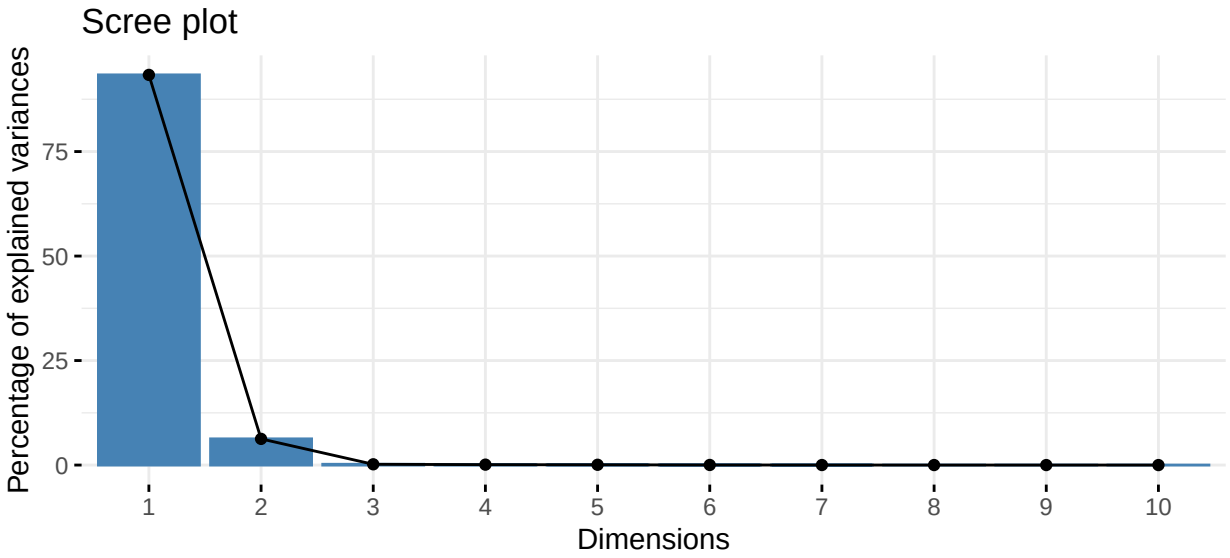
```
# Importing the data:
data4 <- read.table("NIRspectra.csv", header=TRUE, sep=";", dec=",")

# Standard PCA
data_pca <- data4[,-127] #  the feature space
pca <- prcomp(data_pca)
round(summary(pca)$importance[,1:5],5)
```

```
##                             PC1     PC2     PC3     PC4     PC5
## Standard deviation      0.12206 0.03162 0.00544 0.00401 0.00330
## Proportion of Variance 0.93332 0.06263 0.00185 0.00101 0.00068
## Cumulative Proportion  0.93332 0.99596 0.99781 0.99882 0.99950
```

From the table we can observe that the first two principal components explain 99.6% of the total variance.

Also, we will plot the percentage of explained variation for each dimension (PC1 to PC10) with the function`fviz_eig()` from the **factoextra** package:

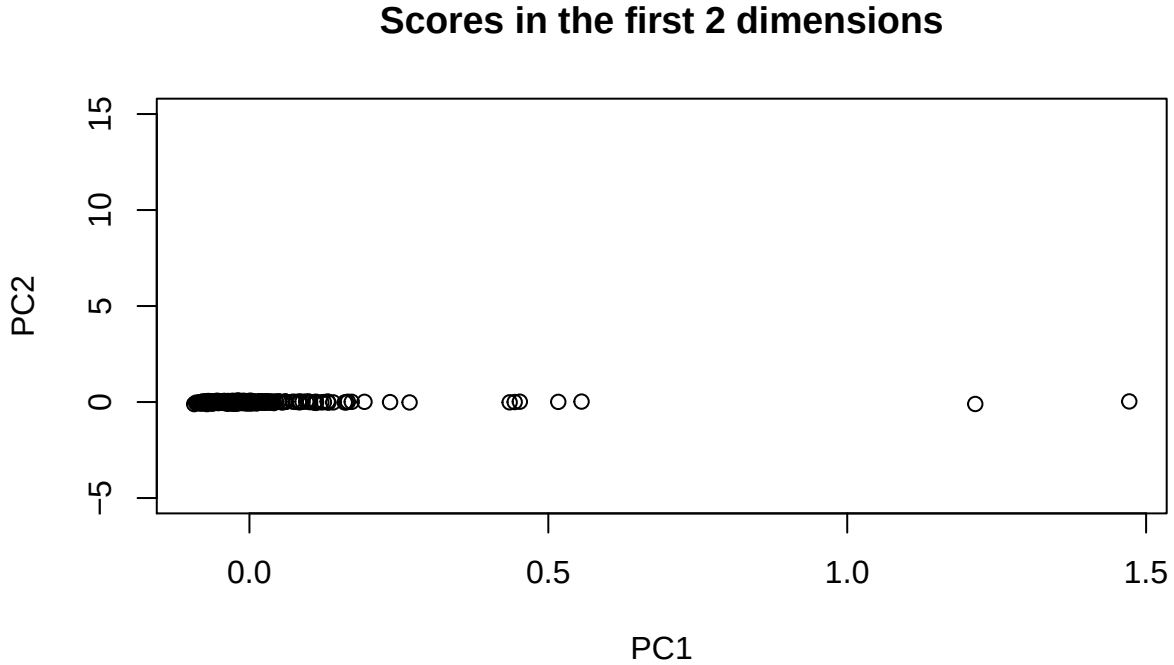
```
# Plot of the variation explained by each PC.
library(factoextra)
fviz_eig(pca)
```



Again, we see that the first principal component explains more than 90% of the variance. Also, it is made evident that we should only extract 2 PC's in order to explain almost all the variation in our data (nearly without losing information).

Finally, we will plot the scores in in the coordinates (PC1, PC2).

```
plot(pca$x[,1], pca$x[,2], ylim=c(-5,15), xlab="PC1", ylab="PC2", main="Scores in the first 2 dimensions
```



According to this plot there are unusual diesel fuels (outliers), those with high values in the x axis (PC1).There are two diesel fuels with very big scores compared to the others (1.4718634 & 1.2141754) which we can see on the far right part of the scores plot. Those are the more significant, but we can observe around 4 more fuels having quite big scores (0.5555528 0.5166276 0.4523158 0.4437367), those who are around in the middle of the plot.

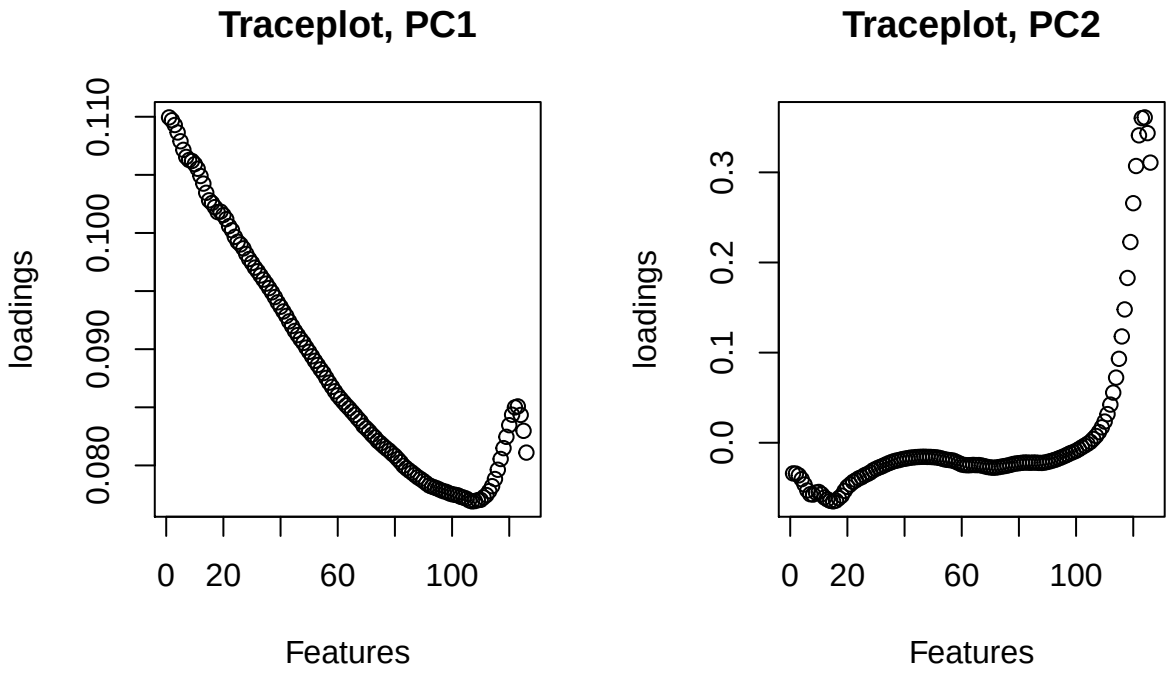
More or less we can say that the unusual diesel fuels have a PC1 score greater than 0.4.

2. Make trace plots of the loadings of the components selected in step 1. Is there any principle

component that is explained by mainly a few original features?

Since in question (1) we selected PC1 and PC2, now we will make a trace plot for each one:

```
U= pca$rotation
par(mfrow=c(1,2))
plot(U[,1], main="Traceplot, PC1", xlab="Features",ylab="loadings")
plot(U[,2],main="Traceplot, PC2", xlab="Features", ylab="loadings")
```




```
cat("Variables with higest loadings in PC1:\n")
```

```
## Variables with higest loadings in PC1:
```

```
head(sort(U[,1], decreasing = TRUE))
```

```
##      X750      X752      X754      X756      X758      X760
## 0.1099439 0.1096993 0.1092886 0.1086458 0.1078912 0.1071514
```

```
cat("Variables with higest loadings in PC2:\n")
```

```
## Variables with higest loadings in PC2:
```

```
head(sort(U[,2], decreasing = TRUE))
```

```
##      X996      X994      X998      X992     X1000      X990
## 0.3609749 0.3601228 0.3435218 0.3410492 0.3108059 0.3070516
```

However, the variables with highest loadings don't differ too much from the rest (as observed in the PC1 plot), so we can't choose only a few original features that explain PC1.

Regarding the traceplot for PC2, we can observe that PC2 is basically explained by the 124th (X996) and 123th (X994) variables which have the biggest loadings (0.3609748774 and 0.3601227691, respectively)

3. Perform Independent Component Analysis with the number of components selected in step 1 (set seed 12345). Check the documentation for the fastICA method in R and do the following:

a. Compute $W' = K \cdot W$ and present the columns of W' in form of the trace plots. Compare

with the trace plots in step 2 and make conclusions. What kind of measure is represented by the matrix W'?

Matrix definitions, obtained from `fastICA()` function:

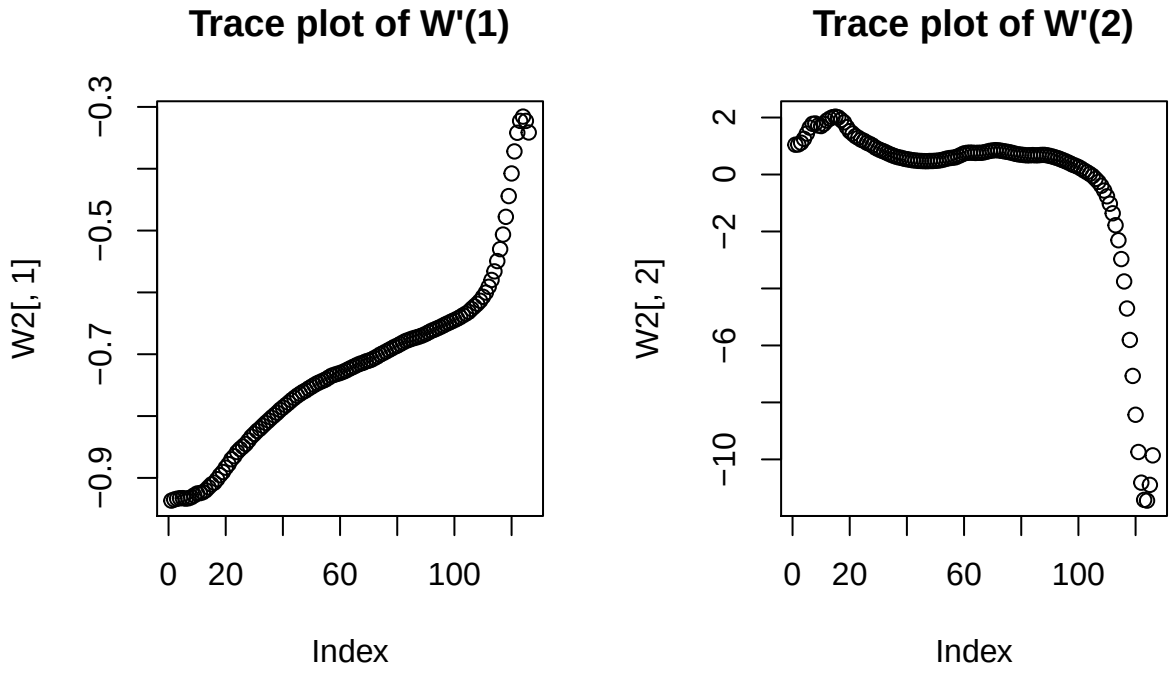
```
K: pre-whitening matrix that projects data onto the first n.comp principal components.
W: estimated un-mixing matrix that maximizes the non-gaussianity of the sources.
```

We use `fastica()` with the data frame containung the variables 750 to 1000 from the original data set (*NIRspectra*) and indicating in the **n.comp** argument that the number of components to be extracted is 2.

So, the output W matrix is a 2x2 matrix and K has 126 rows and 2 columns. Then, when computing $W' = K \cdot W$ we only need to use the command "%*%" and the obtained matrix (W') will have 126x2 dimensions.

We will plot the W' values in each principal component, which represent the loadings of the variables in the Independent Component Analysis:

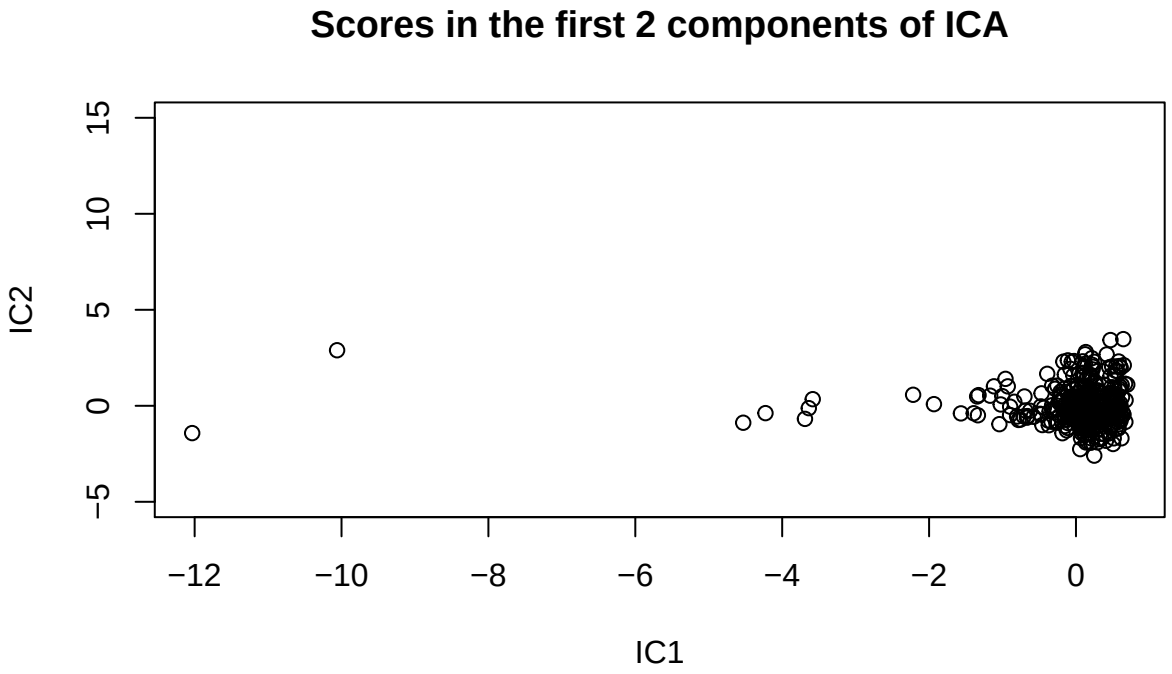
```
library(fastICA)
set.seed(12345)

ICA <- fastICA(data_pca,2)
W2 <- ICA$K %*% ICA$W

par(mfrow=c(1,2))
plot(W2[,1], main = "Trace plot of W'(1)")
plot(W2[,2], main = "Trace plot of W'(2)")
```



If we compare these two trace plots with the ones obtained in step (2) we observe that the loadings in ICA are the reversed PCA loadings.

b. Make a plot of the scores of the first two latent features and compare it with the score plot from step 1.

```
plot(ICA$S[,1], ICA$S[,2],ylim=c(-5,15), xlab="IC1", ylab="IC2", main="Scores in the first 2 components
```



Again, we can observe that the plot in ICA is the reversed image of the PCA's scores plot. So, we still observe some unusual diesel fuels values.

Topic 3.

Assignment 1. Kernel Methods.

We have defined that the latitude and longitud of the point of interest is $(55.90000, 12.71660)$ and the date of interest is "2004-05-28". As said in the exercise, our times of interest, for which we will make predictions, are: "04:00:00", "06:00:00", ... , "22:00:00", "24:00:00".

Now, the first step is to compute the differences from the data points, dates and times with the point, date and times of interest (*See code in the Appendix session to see how these are calculated*).

```
# 1.1. Importing the data:
RNGversion('3.5.1')
set.seed(1234567890)
library(geosphere)

stations <- read.csv("stations.csv", sep=",", header=TRUE, stringsAsFactors=FALSE,
                     fileEncoding="latin1")
temps <- read.csv("temps50k.csv")
st <- merge(stations,temps,by="station_number")

# 1.2. Computing Gaussian kernels:
a <- 55.90000 # latitude of interest
b <- 12.71660 # longitude of interest
date <- "2004-05-28" # The date to predict
times <- as.data.frame(as.factor(c("04:00:00", "06:00:00", "08:00:00", "10:00:00","12:00:00",
           "14:00:00", "16:00:00", "18:00:00", "20:00:00","22:00:00", "24:00:00")))
colnames(times) <- "time_of_interest"
times$time_ind <- row.names(times)
st <- merge.data.frame(st,times,all=TRUE)
```

```r
# Filtering our date (removind observations posteriors to our date)
filter_date <- ifelse(as.character(st$date) < date, TRUE, FALSE)
st <- st[which(filter_date==TRUE),]

### A. Distance from a station to the point of interest.
point_interest <- c(a,b)
st$dif_distance <- distHaversine(p1=point_interest, p2=st[,c(4,5)])

### B. Distance between the day a temp measurement was made and the day of interest.
st$dif_date <- abs(as.integer(difftime(date, st$date, units = "days")))
st$dif_date <- st$dif_date %% 365

### C. Distance between the hour of the day a temp measurement was made and the hour of interest.
st$dif_time <- abs(as.numeric(difftime(strptime(st$time_of_interest,"%H:%M:%S"),
                                       strptime(st$time,"%H:%M:%S") ,units = "hours")))
st$dif_time <-ifelse(st$dif_time <= 12 , st$dif_time , (24-st$dif_time))
```

Choose an appropriate smoothing coefficient or width for each of the three kernels above. Answer to the following questions:

a) Show that your choice for the kernels' width is sensible, i.e. that it gives more weight to closer points. Discuss why your of definition of closeness is reasonable.

The parameter h is called smoothing factor or width. There is a natural bias–variance tradeoff as we change the width of the kernel. When we use a small width we consider few points, whereas when we use a large width we consider many points. To show that the kernels' width is sensible, we will choose small and large widths for each of the kernels. The chosen small widths are: $h_1 = 30000$, $h_2 = 10$,$h_3 = 0.5$ and the large widths:$h_1 = 1000000$, $h_2 = 300$,$h_3 = 11$.

To apply kernel method, we first define the Gaussian kernel. Gaussian kernel can be written as:

$$k_i(x, x_n) = exp\left(- \left(\frac{||x - x_n||}{h_i} \right)^2 \right)$$

where x are our points of interest, x_n are the points from the dataset and $|| \cdot ||$ is Euclidean norm.

Then we combine the kernels by summing them.

$$K(combined) = K(distance) + K(date) + K(hour)$$

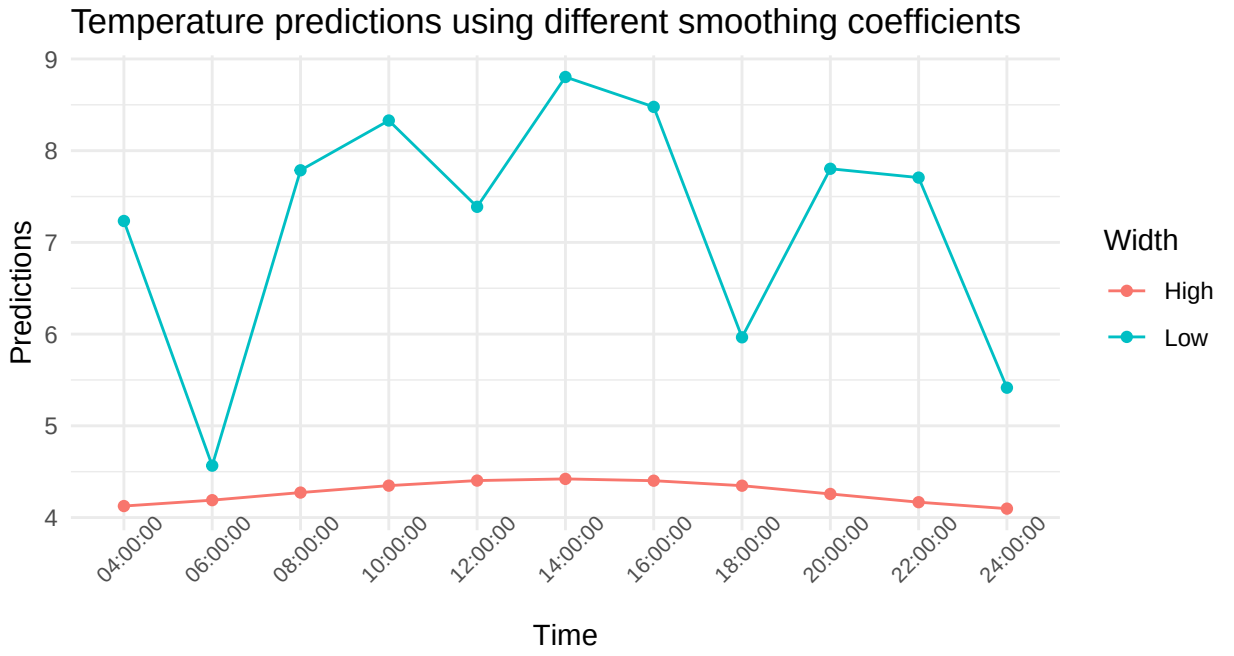
and finally we calculate the result using following formula:

$$y_k(X) = \frac{\sum_n K(\frac{x - x_n}{h}) t_n}{\sum_n K(\frac{x - x_n}{h})}$$

We will plot now the predicted temperatures for the same point date and times obtained with each kernel:

```r
# New data frame for this question:
st_1 <- st

# Setting low and high smoothing factors:
low_h1 <- 30000
low_h2 <- 10
low_h3 <- 0.5
```

```r
high_h1 <- 1000000
high_h2 <- 300
high_h3 <- 11

# Calculating guassian kernels
st_1$h_distance_low <- exp(-(st_1$dif_distance/low_h1)^2)
st_1$h_date_low <- exp(-(st_1$dif_date/low_h2)^2)
st_1$h_time_low <- exp(-(st_1$dif_time/low_h3)^2)

st_1$h_distance_high <- exp(-(st_1$dif_distance/high_h1)^2)
st_1$h_date_high <- exp(-(st_1$dif_date/high_h2)^2)
st_1$h_time_high <- exp(-(st_1$dif_time/high_h3)^2)

# Kernels combinations: summing and multiplying them up.
st_1$kernel_sum_low <- (st_1$h_distance_low + st_1$h_date_low + st_1$h_time_low)
st_1$kernel_sum_high <- (st_1$h_distance_high + st_1$h_date_high + st_1$h_time_high)

# Predicted temperatures:
result<- NULL
for(i in 1:11){
  temp <- st_1[st_1$time_ind==i,]
  pred_sum_low <- sum(temp$kernel_sum_low*temp$air_temperature)/sum(temp$kernel_sum_low)
  pred_sum_high <- sum(temp$kernel_sum_high*temp$air_temperature)/sum(temp$kernel_sum_high)

  temp <- cbind(i, pred_sum_low, pred_sum_high)
  result <- rbind(result,temp)
}
result <- as.data.frame(result)
result <- merge(x =result, y = times, by.x = "i", by.y = "time_ind", all.x = TRUE)
result$pred_sum <- as.numeric(as.character(result$pred_sum_low))
result$pred_prod <- as.numeric(as.character(result$pred_sum_high))

# Plot of predicted temperatures for each kernel:
library(ggplot2)

df <- data.frame("Time"= result$time_of_interest ,
                 "Predictions"= c(result$pred_sum_low, result$pred_sum_high),
                 "Width"= rep(c("Low", "High"), each=11))

ggplot(data=df, aes(x=Time, y=Predictions, group=Width)) + geom_line(aes(color=Width)) +
  geom_point(aes(color=Width)) + ggtitle("Temperature predictions using different smoothing coefficients
  theme_minimal() + theme(axis.text.x =element_text(size = rel(0.9), angle = 45))
```



From the plot above we can observe that when the kernel widths are low (30km, 10 days and half an hour, for the distance kernel, the date kernel and the time kernel, respectively) there are a lot of variations in the values (the result is very noisy).

On the other hand, when the widths are high (1000km, 300 days and 11 hours) the line is really smooth and the values are smaller.

From the theory, "the best density model is obtained for some intermediate value of h".

b) Instead of combining the three kernels into one by summing them up, multiply them. Compare the results obtained in both cases and elaborate on why they may differ.

In this quetion we want to compare models where the combination of the kernels is done by summing or multiplying them. So, for both cases we will choose widths that we think that are reasonable for the nature of the variables and our problem:

For the kernel of the distance, we will set a width of 500km because we think it is a distance from which the temperature can be different. For the kernel of the date, we will choose a width of 100 days, and for the kernel of the time, we will set a smoothing parameter of 5 hours. Temperature could change in a period of 5 hours (day and night, for example) and in a period of 100 days (in approximately three months the season might change and thus, also the temperatures).

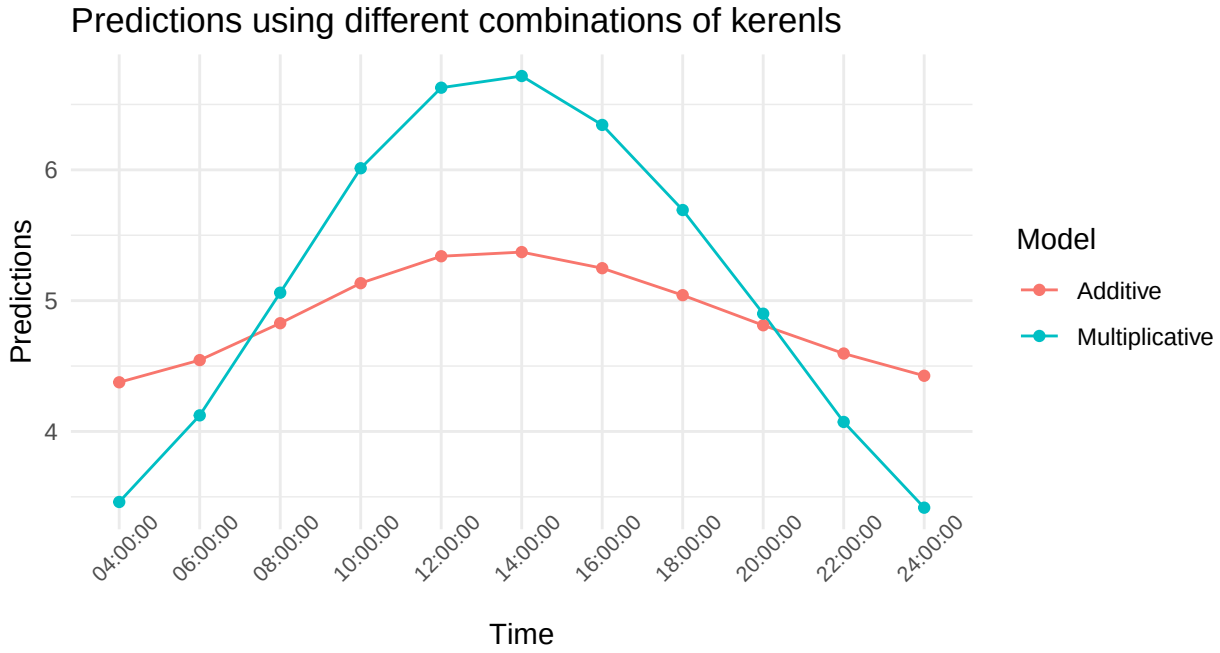
To combine the kernels by mutyplying them we will compute:

$$K(combined) = K(distance) \times K(date) \times K(hour)$$

To compare the results for the additive and the multiplicative kernel combinations, we will plot the predictions obtained in each case:

```
# The parameter h is called smoothing factor or width.
h1 <- 500000 # h1: width for kernel 1
h2 <- 100 # h2: width for kernel 2
h3 <- 5 # h3: width for kernel 3

# Calculating guassian kernels
st$h_distance <- exp(-(st$dif_distance/h1)^2)
st$h_date <- exp(-(st$dif_date/h2)^2)
```

```r
st$h_time <- exp(-(st$dif_time/h3)^2)

# Kernels combinations: summing and multiplying them up.
st$kernel_sum <- (st$h_distance + st$h_date + st$h_time)
st$kernel_prod <- (st$h_distance * st$h_date *st$h_time)

# Predicted temperatures:
result<- NULL
for(i in 1:11){
  temp <- st[st$time_ind==i,]
  pred_sum <- sum(temp$kernel_sum*temp$air_temperature)/sum(temp$kernel_sum)
  pred_prod <- sum(temp$kernel_prod*temp$air_temperature)/sum(temp$kernel_prod)

  temp <- cbind(i, pred_sum, pred_prod)
  result <- rbind(result,temp)
}
result <- as.data.frame(result)
result <- merge(x =result, y = times, by.x = "i", by.y = "time_ind", all.x = TRUE)
result$pred_sum <- as.numeric(as.character(result$pred_sum))
result$pred_prod <- as.numeric(as.character(result$pred_prod))

# Plot of predicted temperatures for each kernel:
library(ggplot2)

df <- data.frame("Time"= result$time_of_interest ,
                 "Predictions"= c(result$pred_sum, result$pred_prod),
                 "Model"= rep(c("Additive", "Multiplicative"), each=11))

ggplot(data=df, aes(x=Time, y=Predictions, group=Model)) + geom_line(aes(color=Model)) + geom_point(aes
  theme_minimal() + theme(axis.text.x =element_text(size = rel(0.9), angle = 45))
```



Now that we are using more intermediate values for the widths, in both models we can clearly observe the gaussian bell. However, in the additive kernel the variance of the model is higher as the curve is more flat

and has longer tails.

Then, with the multiplicative model we are obtaining more diverse predictions (higher and lower than those obtained with the additive model).

Assignment 2. Support Vector Machines.

Use the function ksvm from the R package kernlab to learn a SVM for classifying the spam dataset that is included with the package. Consider the radial basis function kernel (also known as Gaussian) with a width of 0.05. For the C parameter, consider values 0.5, 1 and 5. This implies that you have to consider three models.

a) Perform model selection, i.e. select the most promising of the three models (use any method of your choice except cross-validation or nested cross-validation).

The SVM model with "rbfdot" (Radial Basis kernel "Gaussian") has been performed with the function `ksvm()` from the package **kernlab**. We have set in the *kpar* argument that the width for the Radial Basis kernel that we want to use is 0.05: `kpar=list(sigma=0.05)`. We will fit three models with this characteristics, each one with a different cost of constraints violation $C = 0.5, 1, 5$.

To method used for the model selection will be the holdout method so the spam dataset will be separated into three sets: training validation and test, with 0.5, 0.25 and 0.25 of the original observations (chosen randomly), respectively. Then, the models will be fitted using the training data and the predictions will be performed for the validation dataset.

The best model will be chosen based on the validation set by calculating the misclassification errors.

```
# 2.0. Dataset:
library(kernlab)
data(spam) # spam[,58] or spam$type is the target variable

# Divide data into training, validation and test sets:
n=dim(spam)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=spam[id,]

id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.25))
valid=spam[id2,]

id3=setdiff(id1,id2)
test=spam[id3,]


x_train <- as.matrix(train[,-58])
y_train <- train[,58]

x_valid <- as.matrix(valid[,-58])
y_valid <- valid[,58]

svm_fit1 <- ksvm(x_train, y_train, scaled=FALSE, kernel="rbfdot", kpar=list(sigma=0.05), C=0.5)
svm_fit2 <- ksvm(x_train, y_train, scaled=FALSE, kernel="rbfdot", kpar=list(sigma=0.05), C=1)
svm_fit3 <- ksvm(x_train, y_train, scaled=FALSE, kernel="rbfdot", kpar=list(sigma=0.05), C=5)

pred_svm1 <- predict(svm_fit1, x_valid)
```

```
pred_svm2 <- predict(svm_fit2, x_valid)
pred_svm3 <- predict(svm_fit3, x_valid)
```

The confusion matrices for the test data for every model are:

```
cat("Model with C=0.5\n")
```

```
## Model with C=0.5
```

```
table(y_valid, pred_svm1)
```

```
##          pred_svm1
## y_valid   nonspam spam
##   nonspam     690   13
##   spam        276  171
```

```
cat("Model with C=1\n")
```

```
## Model with C=1
```

```
table(y_valid, pred_svm2)
```

```
##          pred_svm2
## y_valid   nonspam spam
##   nonspam     634   69
##   spam        189  258
```

```
cat("Model with C=5\n")
```

```
## Model with C=5
```

```
table(y_valid, pred_svm3)
```

```
##          pred_svm3
## y_valid   nonspam spam
##   nonspam     593  110
##   spam         84  363
```

Finally, we will show a table with the test misclassification errors obtained in each model:

```
data.frame("Model"=c("SVM with C=0.5","SVM with C=1","SVM with C=5"),
           "Misclassification errors"=c(mean(pred_svm1 != y_valid),
                                        mean(pred_svm2 != y_valid),mean(pred_svm3 != y_valid)))
```

```
##            Model Misclassification.errors
## 1 SVM with C=0.5                0.2513043
## 2   SVM with C=1                0.2243478
## 3   SVM with C=5                0.1686957
```

The model using C = 5 is the one that offers the lowest error. This means that this model predicts better than the other two. For this reason, this is the model that will be selected as the optimal.

However, its worth mentioning that, although only 16.87% of the observations are classified wrong when using the third model, the number of nonspam mails classified as spam in this model is greater than in the others.

b) Estimate the generalization error of the SVM selected above (use any method of your choice except cross-validation or nested cross-validation).

We have computed the predictions and misclassification errors on the test dataset, which is unseen data for the model. So now we can use "train" and "valid" datasets to train the model.

```
new_train <- rbind(train, valid)
svm_fit <- ksvm(as.matrix(new_train[,-58]), new_train[,58], scale=FALSE,
                kernel="rbfdot", kpar=list(sigma=0.05), C=5)
pred_svm <- predict(svm_fit,  test[,-58])

cat("The confussion matrix is:\n")
```

```
## The confussion matrix is:
```

```
table(test[,58], pred_svm)
```

```
##          pred_svm
##           nonspam spam
##   nonspam     614   60
##   spam        158  319
```

```
cat("The  generalization error is:\n")
```

```
## The  generalization error is:
```

```
mean(pred_svm != test[,58])
```

```
## [1] 0.1894005
```

So, the final model predicts wrongly around 19% of the observations. Also, from the confussion matrix we can observe that the number of nonspam mails classified as spam is considerably smaller compared to the number of spam mails classified as nonspam.

c) Produce the SVM that will be returned to the user, i.e. show the code.

The final model is performed by using the following code:

```
svm_fit <- ksvm(as.matrix(new_train[,-58]), new_train[,58], scale=FALSE,
                kernel="rbfdot", kpar=list(sigma=0.05), C=5)
```

d) What is the purpose of the parameter C ?

Our goal is to maximize the margin while softly penalizing points that lie on the wrong side of the margin boundary. For this reason we minimize

$$C\sum_{n=1}^{N}\xi_n + \frac{1}{2}\|\mathbf{w}\|^2$$

where the parameter C $>$ 0 controls the trade-off between the slack variable penalty and the margin.

Assignment 3. Neural Networks.

Train a neural network to learn the trigonometric sine function. To do so, sample 50 points uniformly at random in the interval [0, 10]. Apply the sine function to each point. The resulting pairs are the data available to you. Use 25 of the 50 points for training and the rest for vali- dation. The validation set is used for early stop of the gradient descent. That is, you should use the validation set to detect when to stop the gradient descent and so avoid overfitting. Stop the gradient descent when the partial derivatives of the error function are below a given threshold value. Check the argument threshold in the documentation. Consider threshold values i/1000 with i = 1,. . . , 10. Initialize the weights of the neural network to random values in the interval [-1, 1]. Use a neural network with a single hidden layer of 10 units. Use the de- fault values for the arguments not mentioned here. Choose the most

appropriate value for the threshold. Motivate your choice. Provide the final neural network learned with the chosen threshold. Feel free to use the following template.

By following the instructions from assignment 3, a neural network could be calculated as described. By creating ten different neural networks with ten different values for the threshold, from 0.001 to 0.01, the minimum square error could be calculated for each of them and compared with. The results can be seen in figure 3. As can be observed, the best threshold which gives the least amount of MSE is with index 4, which gives the threshold $4/1000 = 0.004$. Because it gives the least amount of MSE, it is therefore the most appropriate neural network of the ten. Figure 5 shows the final neural network and figure 4 shows the difference between the actual data with red points together with the predicted data with black points. By observing figure 4, one can observe that the predicted data is very similar to the real data and therefore the neural network was good.

```r
RNGversion('3.5.1')
library(neuralnet)
set.seed(1234567890)

Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin=sin(Var))
tr <- trva[1:25,] # Training
va <- trva[26:50,] # Validation

winit <- runif(31, -1, 1)
calculate_MSE <- function(obs, pred) {
  mean((obs - pred)^2)
}

all_mse <- c()
min_mse <- 10000000
best_threshold <- 0

for(i in 1:10) {
  # Your code here
  nn <- neuralnet(Sin ~ Var, tr, threshold = i/1000, startweights=winit, hidden = 10)
  prediction <- predict(nn, va)
  temp_mse <- calculate_MSE(va$Sin, prediction)
  all_mse[i] <- temp_mse
  if (temp_mse < min_mse){
    min_mse <- temp_mse
    best_threshold <- i/1000
  }
}

plot(all_mse, type="o") # Your code here
```

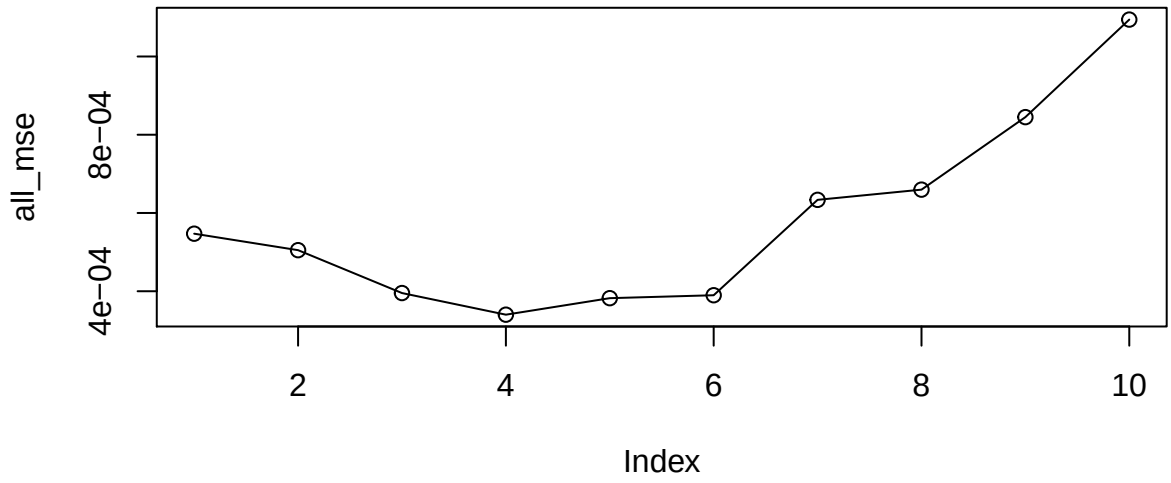


Figure 1. MSE-values for ten different thresholds.

```
best_nn <- neuralnet(Sin~Var, trva, threshold = best_threshold,
                     startweights=winit, hidden = 10)

# Plot of the predictions (black dots) and the data (red dots)
{plot(prediction(best_nn)$rep1)
points(trva, col = "red")}
```

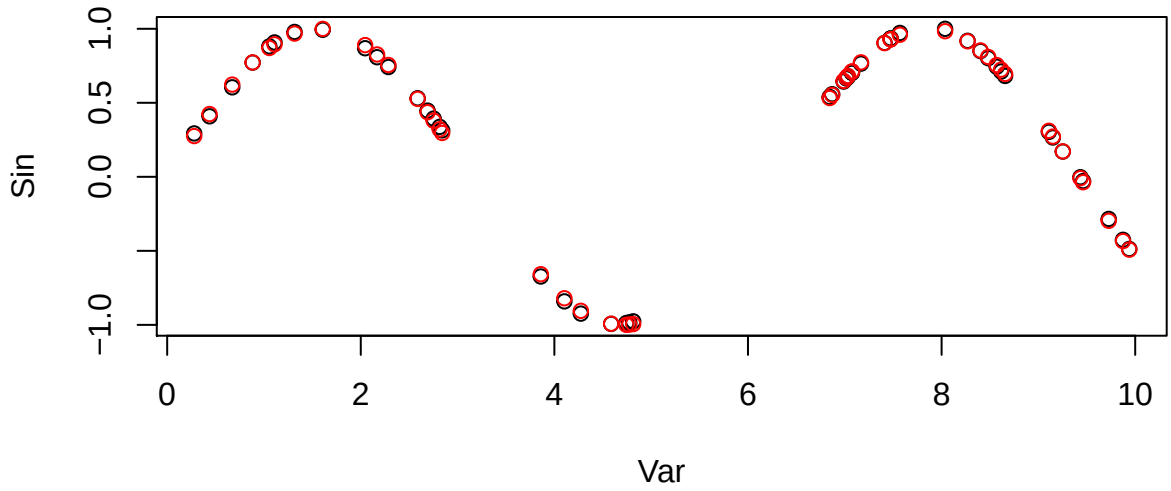


Figure 2. Comparing the data (red) with the prediction (black).

```
plot(best_nn)
```

Figure 3. The optimal neural network.

BLOC 2

Topic 1.

Assignment 1. Ensemble Methods.

Before starting, it is necessary to load the `mboost` and `randomForest` packages, import the file to R and then split the dataset into training and hold-out sets. See *Appendix* to see the code used.

```
# Changing the version:
RNGversion('3.5.1')
```

```r
# Loading the packages
library(mboost)
library(randomForest)
library(ggplot2)

# Importing the data:
sp <- read.csv2("spambase.csv")
sp$Spam <- as.factor(sp$Spam)

# Spliting the data in training and hold-out datasets:
n=dim(sp)[1]
set.seed(12345)
id=sample(1:n, floor(n*(2/3)))
train=sp[id,]
holdout=sp[-id,]
```

1.1. Adaboost classification tree.

To fit an Adaboost classification tree the function `blackboost()` from the **mboost** package has ben used. Also, the `family` argument has been used to specify the desired loss function which, in our case, has been the `AdaExp()` because we want the function to perform the AdaBoost algorithm and for that, the exponential loss function is needed. Finally, with the `control` argument, it has been indicated the number of trees to be considered $(10, 20, ..., 100)$.

Once the model has been fitted, predictions and error rates for the hold-out and the training data sets have been calculated.

```r
# Number of trees to consider.
ntrees <- seq(10, 100, by = 10)

rate_ada_holdout <- vector()
rate_ada_train <- vector()
for(i in ntrees){
  # Fitting the model with the training dataset
  fit_ada <- blackboost(Spam~., data = train, family = AdaExp(), control=boost_control(mstop = i))
          # "AdaExp() uses the exponential loss, which essentially leads to the AdaBoost algorithm"
          # "mstop = an integer giving the number of initial boosting iterations."

  # Predictions
  fitted_ada_holdout <- predict(fit_ada, newdata = holdout, type = "class")
  fitted_ada_train <- predict(fit_ada, newdata = train, type = "class")

  # Error rates
  rate_ada_holdout[(i/10)] <- mean(fitted_ada_holdout != holdout$Spam)
  rate_ada_train[(i/10)] <- mean(fitted_ada_train != train$Spam)
}
```

We use the Adaboost algorithm to classify our data. This algorithm combines weak classifiers to produce a more accurate one.The weak learners are almost always stumps(a tree with one node and two leaves). It starts by giving the same weight to all the samples and later updates every time the weights by increasing the weight of the misclassified and decreasing the weight of the correctly classified ones. The final classifier is the weighted average of the classifiers obtained.

1.2. Random forest model.

In this section, the `randomForest()` from the **randomForest** package has been used to fit the Random forest models, specifying with the `ntrees` argument that the number of trees to grow in each case are $10, 20, ..., 100$.

Then, for each fitted model, predictions and misclassification rates have been calculated.

```
rate_randomforest_holdout <- vector()
rate_randomforest_train <- vector()
for(i in ntrees){
  # Fitting the model with the training dataset
  fit_randomforest <- randomForest(Spam~., data = train, ntree=i) #ntree="Number of trees to grow."

  # Predictions
  fitted_randomforest_holdout <- predict(fit_randomforest, newdata = holdout, type = "class")
  fitted_randomforest_train <- predict(fit_randomforest, newdata = train, type = "class")

  # Error rates
  rate_randomforest_holdout[(i/10)] <- mean(fitted_randomforest_holdout != holdout$Spam)
  rate_randomforest_train[(i/10)] <- mean(fitted_randomforest_train != train$Spam)
}
```

Random forest consists of a large number of decision trees. Each tree gives a class prediction and the class with the most votes becomes our model's prediction. It is important that the trees are relatively uncorrelated and thus they don't get affected by the errors of the other trees and that's what makes random forest a powerful classification method. The basic idea behind this is that while some trees will make wrong predictions, many others will give the correct ones, so the trees as a group will move in the correct direction in the end.

1.3. Performance evaluation.

- Error rates for the Adaboost model:

```
as.data.frame(cbind(ntrees, rate_ada_holdout, rate_ada_train))
```

```
##    ntrees rate_ada_holdout rate_ada_train
## 1      10       0.13298566     0.11998696
## 2      20       0.10625815     0.10205412
## 3      30       0.10169492     0.09455494
## 4      40       0.08344198     0.08346919
## 5      50       0.08083442     0.07890447
## 6      60       0.08018253     0.07727421
## 7      70       0.07822686     0.07433975
## 8      80       0.07692308     0.07368764
## 9      90       0.07366362     0.07303554
## 10    100       0.07366362     0.07173133
```

- Error rates for the Random Forest model:

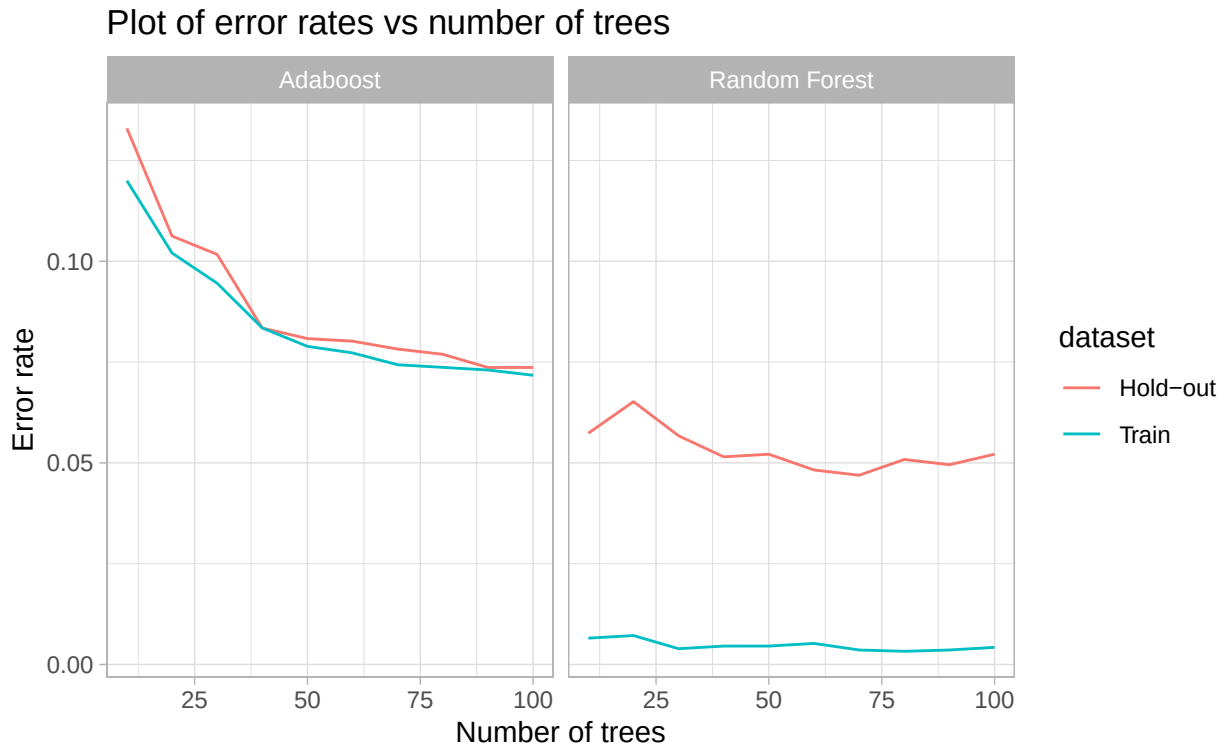
```
as.data.frame(cbind(ntrees, rate_randomforest_holdout, rate_randomforest_train))
```

```
##    ntrees rate_randomforest_holdout rate_randomforest_train
## 1      10                0.05736636             0.006521030
## 2      20                0.06518905             0.007173133
## 3      30                0.05671447             0.003912618
## 4      40                0.05149935             0.004564721
## 5      50                0.05215124             0.004564721
## 6      60                0.04823990             0.005216824
## 7      70                0.04693611             0.003586567
## 8      80                0.05084746             0.003260515
## 9      90                0.04954368             0.003586567
```

```
## 10     100                 0.05215124              0.004238670
```

- Plot showing the error rates by model.

```
# Dataframe to use ggplot()
df <- data.frame(ntrees,  error_rates= c(rate_ada_holdout, rate_ada_train,
                                         rate_randomforest_holdout, rate_randomforest_train),
                 dataset  = rep(c("Hold-out", "Train", "Hold-out", "Train"), each=10),
                 model = rep(c("Adaboost", "Adaboost",  "Random Forest", "Random Forest"), each=10))

# Plot of error_rates vs ntrees:
ggplot(df, aes(x = ntrees, y = error_rates, group = dataset, color = dataset)) +
  geom_line() + ggtitle("Plot of error rates vs number of trees") +
  xlab("Number of trees") + ylab("Error rate") + theme_light() +
  facet_grid(cols = vars(model))
```



From the plots above we can observe that the Adaboost model has higher error rates than the Random forest model for all the tree numbers considered. Also, the erro rates for the hold-out and the training data sets are more similar in the Adaboost model while in Random Forest the error rate for the training data set is really small compared to the rate for the hold-out data. Finally, we can also comment that the error rate in Adaboost decreases steadily as the number of trees to consider increases and the errors for the Random Forest model remain more or less the same (around 0.05 for the hold-out data), especially at from 40 number of trees.

Comparing the training errors for both methods we see that Random forest gives a much smaller error for the training data (0.003260515) and this is achieved with 80 trees, whereas the smallest training error with Adaboost is 0.07173133 and is achived with 100 trees. This means that Random forest gave better class predictions.

Regarding the test data, Random forest again gave a smaller error (0.0482399), using 20 trees, whereas Adaboost achieved the lowest error (0.07366362) for 100 trees. In this case also, Random forest had a better performance. Those results lead to the conclusion that Random Forest is a better classification method for

our data set as it gave us better results.

Finally, we can also comment that the error rate in Adaboost decreases steadily as the number of trees to consider increases and the errors for the Random Forest model remain more or less the same (around 0.05 for the hold-out data), especially after around 20 to 30 trees.

Assignment 2. Mixture models.

2.1. EM ALGORITHM EXPLANATION.[1] [2]:Source: Chapter 9 "Mixture models and EM" from the book "Pattern Recognition and Machine Learning" of Christopher M. Bishop.

Let $\boldsymbol{z}$ be a latent variable that denotes from which distribution the sample $\boldsymbol{x} = (x_1, x_2, \cdots, x_D)^T$ is coming from and the probability of observing x given μ for a multivariate Bernoulli distribution:

$$p(\boldsymbol{x}|\boldsymbol{\mu}) = \prod_{d=1}^{D} \mu_{kd}^{x_d}(1-\mu_{kd})^{(1-x_d)}$$

Then, the mixture model for a multivariate Bernoulli distribution is

$$p(\boldsymbol{x}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_{k=1}^{K} \pi_k \cdot (\boldsymbol{x}|\boldsymbol{\mu}_k)$$

where:

- π are the mixing coefficients ($0 \leq \pi_k \leq 1$).
- μ are the Bernoulli parameters indicating probability of success ($0 \leq \mu_k \leq 1$).

And the log-likelihood function for a sample of size N is:

$$\ln p(\boldsymbol{x}|\mu, \pi) = \sum_{n=1}^{N} \ln \left[\sum_{k=1}^{K} \pi_k \cdot p(\boldsymbol{x}_n|\boldsymbol{\mu}_k) \right]$$

Our goal is to find maximum likelihood estimates for the parameters in the mixture model above, and in order to achieve that, the expectation–maximization (EM) algorithm will be implemented.

The EM iteration alternates between performing an expectation (E) step, which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and a maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the E step. These parameter-estimates are then used to determine the distribution of the latent variables in the next E step.[3]

The EM Algorithm has different steps:

1. Set $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ to some initial values.
2. *E step*: Compute the posterior distribution $p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})$ for each point.
   2.1. This is calculated using the Bayes's rule: $p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}) = \frac{\pi_k \cdot p(\boldsymbol{x}_n|\boldsymbol{\mu}_k)}{\sum_{k=1}^{K} \pi_k \cdot p(\boldsymbol{x}_n|\boldsymbol{\mu}_k)}$\$.
   2.2. Compute the ML estimation (log-likelihood function shown above).
3. *M step*: Adjust π and μ to fit points assigned to them:
   3.1. Set π_k to $\pi_k^{ML} = \frac{\sum_{n=1}^{N} p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{N}$.
   3.2. Set μ_{ki} to $\mu_k^{ML} = \frac{\sum_{n=1}^{N} x_n \cdot p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_{n=1}^{N} p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$.

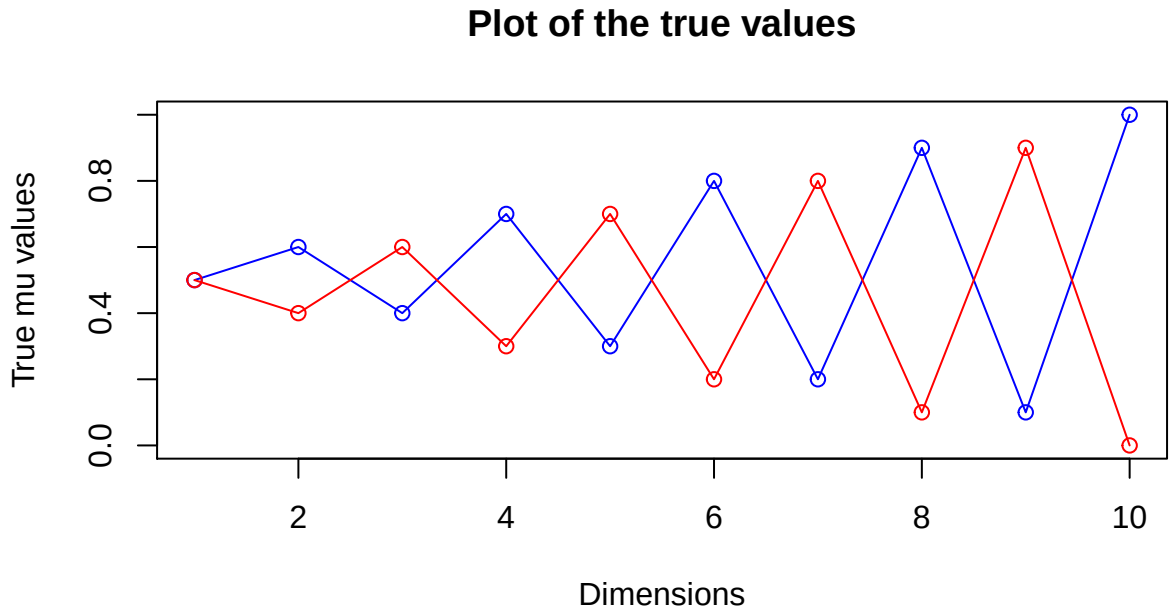
[1] (https://rpubs.com/JanpuHou/298239)
[2] (https://rpubs.com/JanpuHou/298239)
[3] (https://rpubs.com/JanpuHou/298239)

4. Iterate until it converges (repeat until $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ don't change).

2.2. RESULTS FOR K=2 COMPONENTS

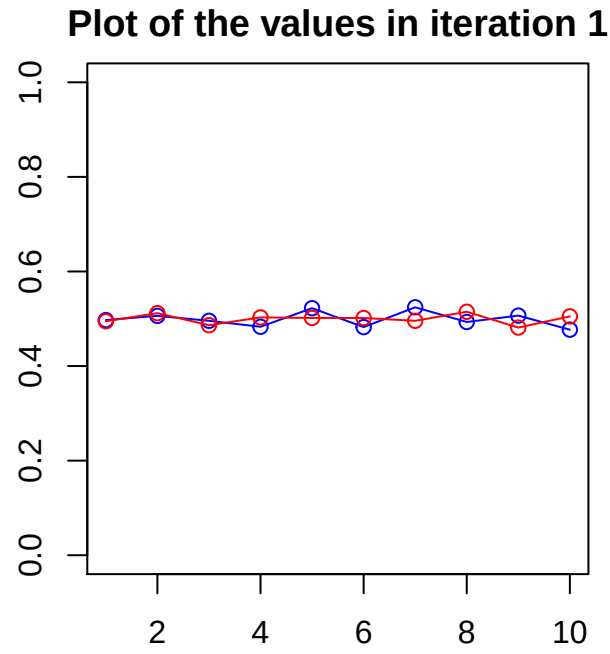
```
### STEP 0. INITIALIZATING THE DATA
set.seed(1234567890)
max_it <- 100 # max number of EM iterations
min_change <- 0.1 # min change in log likelihood between two consecutive EM iterations
N=1000 # number of training points
D=10 # number of dimensions
x <- matrix(nrow=N, ncol=D) # training data

true_pi <- vector(length = 2) # true mixing coefficients
true_mu <- matrix(nrow=2, ncol=D) # true conditional distributions
true_pi=c(1/3, 1/3)
true_mu[1,]=c(0.5,0.6,0.4,0.7,0.3,0.8,0.2,0.9,0.1,1)
true_mu[2,]=c(0.5,0.4,0.6,0.3,0.7,0.2,0.8,0.1,0.9,0)

# 0.1. Producing the training data
for(n in 1:N) {
  k <- sample(1:2,1,prob=true_pi)
  for(d in 1:D) {
    x[n,d] <- rbinom(1,1,true_mu[k,d])
  }
}

K=2 # number of guessed components
z <- matrix(nrow=N, ncol=K) # fractional component assignments
pi <- vector(length = K) # mixing coefficients
mu <- matrix(nrow=K, ncol=D) # conditional distributions
llik <- vector(length = max_it) # log likelihood of the EM iterations

### STEP 1. Random initialization of the paramters
pi <- runif(K,0.49,0.51)
pi <- pi / sum(pi)
for(k in 1:K) {
  mu[k,] <- runif(D,0.49,0.51)
}

for(it in 1:max_it){
  Sys.sleep(0.5)

  ###  STEP 2. E-step: Computation of the fractional component assignments

   # 2.1. We have to compute bayes rule: p(z,x|mu,pi)/ sum(p(z,x|mu,pi))

          prob_x <- exp(x%*%log(t(mu))+(1-x)%*%log(t(1-mu)))
          pi_prob_x <- prob_x * matrix(rep(pi, N), nrow=N, byrow =T)
          sum_pi_prob_x <- rowSums(pi_prob_x)


          z <- pi_prob_x/sum_pi_prob_x


   # 2.2. Log likelihood computation. sum_N ln(sum_K pi * bernoulli)
```

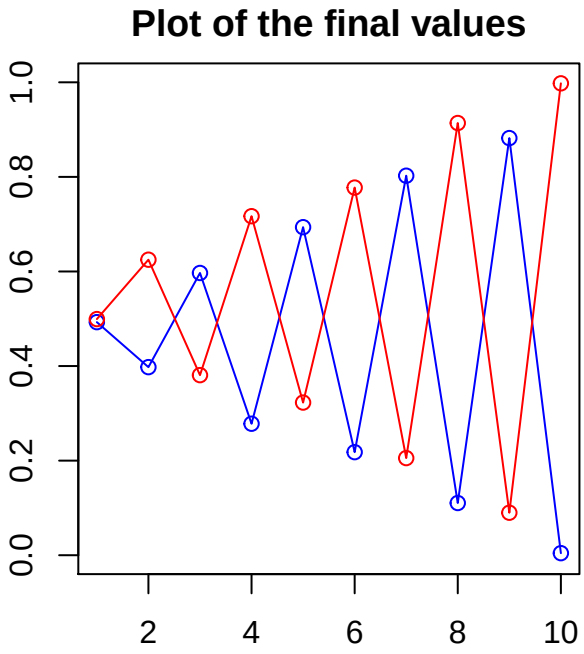
```
        llik[it] <- sum(log(sum_pi_prob_x))
        llik

  #cat("iteration: ", it, "log likelihood: ", llik[it], "\n")
  flush.console()

    # 2.2.1. Stop if the lok likelihood has not changed significantly
      if(it > 1){
        change <- abs(llik[it]-llik[it-1])
        if(change < min_change){
          break
        }
      }

  ###  STEP 3. M-step: ML parameter estimation from the data and fractional component assignments
     # 3.1 Setting new mu. mu_ML= sum_kn(x*z)/sum_k(z)
          mu <- (t(z)%*%x)/colSums(z)

     # 3.2. Setting new pi: pi_ML= sum_k(z)/N (of all n)
          pi <- colSums(z)/N

  ### Plot of the initial values:
   if(it==1){
    plot_mu <- mu
    plot_pi <- pi
  }
}

{plot(true_mu[1,], type="o", col="blue", ylim=c(0,1), main="Plot of the true values", xlab="Dimensions"
points(true_mu[2,], type="o", col="red")}
```




```
par(mfrow=c(1,2), mar=c(3,2,2,1)+0.1)
{plot(plot_mu[1,], type="o", col="blue", ylim=c(0,1), main = "Plot of the values in iteration 1", xlab=
points(plot_mu[2,], type="o", col="red")}
```

```
{plot(mu[1,], type="o", col="blue", ylim=c(0,1), main = "Plot of the final values", xlab="Dimensions", y
points(mu[2,], type="o", col="red")}
```






```
cat("Pi values for K=2: ", "\n")
```

```
## Pi values for K=2:
```

```
pi
```

```
## [1] 0.5101402 0.4898598
```

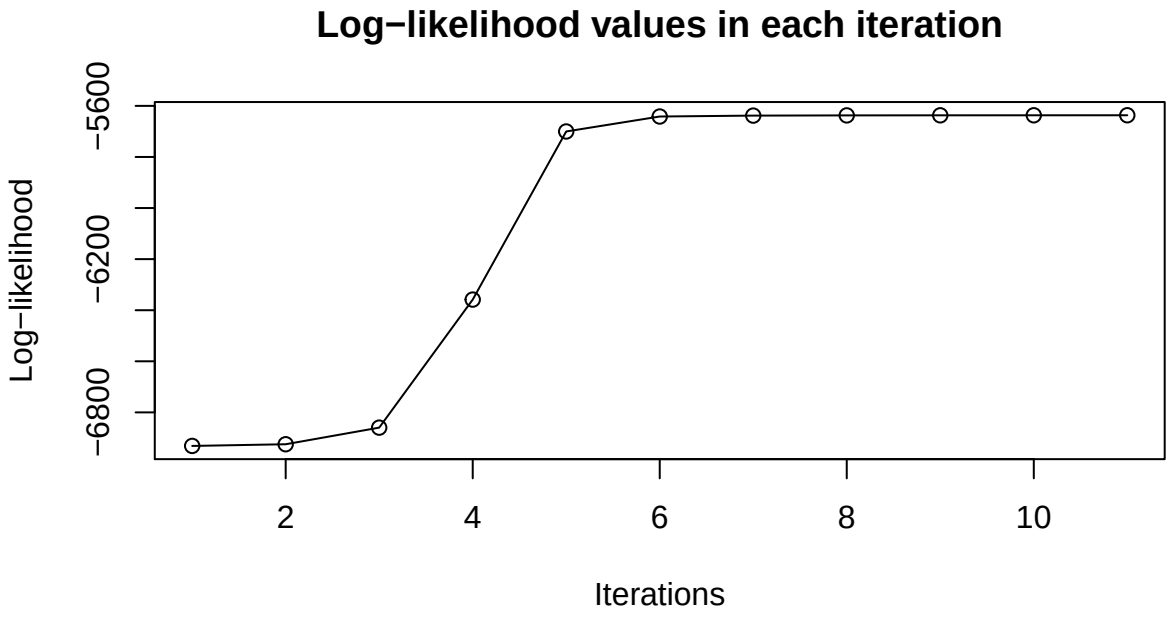
```
cat("Mu values for K=2: ", "\n")
```

```
## Mu values for K=2:
```

```
mu
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4928148 0.3976832 0.5968198 0.2780429 0.6936406 0.2179508 0.8024894
## [2,] 0.4993170 0.6249254 0.3807992 0.7168564 0.3228394 0.7773950 0.2054015
##           [,8]       [,9]       [,10]
## [1,] 0.1103907 0.88203975 0.004305102
## [2,] 0.9139049 0.08989523 0.997844285
```

```
par(mfrow=c(1,1), mar=c(5, 4, 4, 2) + 0.1)
plot(llik[1:it], type="o", main="Log-likelihood values in each iteration", ylab="Log-likelihood", xlab=
```



2.3. RESULTS FOR K=3 COMPONENTS

```
### STEP 0. INITIALIZATING THE DATA
set.seed(1234567890)
max_it <- 100 # max number of EM iterations
min_change <- 0.1 # min change in log likelihood between two consecutive EM iterations
N=1000 # number of training points
D=10 # number of dimensions
x <- matrix(nrow=N, ncol=D) # training data

true_pi <- vector(length = 3) # true mixing coefficients
true_mu <- matrix(nrow=3, ncol=D) # true conditional distributions
true_pi=c(1/3, 1/3, 1/3)
true_mu[1,]=c(0.5,0.6,0.4,0.7,0.3,0.8,0.2,0.9,0.1,1)
true_mu[2,]=c(0.5,0.4,0.6,0.3,0.7,0.2,0.8,0.1,0.9,0)
true_mu[3,]=c(0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5)

# 0.1. Producing the training data
for(n in 1:N) {
  k <- sample(1:3,1,prob=true_pi)
  for(d in 1:D) {
    x[n,d] <- rbinom(1,1,true_mu[k,d])
  }
}

K=3 # number of guessed components
z <- matrix(nrow=N, ncol=K) # fractional component assignments
pi <- vector(length = K) # mixing coefficients
mu <- matrix(nrow=K, ncol=D) # conditional distributions
llik <- vector(length = max_it) # log likelihood of the EM iterations

### STEP 1. Random initialization of the paramters
pi <- runif(K,0.49,0.51)
pi <- pi / sum(pi)
for(k in 1:K) {
```

```
  mu[k,] <- runif(D,0.49,0.51)
}

for(it in 1:max_it){
  Sys.sleep(0.5)

  ###  STEP 2. E-step: Computation of the fractional component assignments

   # 2.1. We have to compute bayes rule: p(z,x|mu,pi)/ sum(p(z,x|mu,pi))

          prob_x <- exp(x%*%log(t(mu))+(1-x)%*%log(t(1-mu)))
          pi_prob_x <- prob_x * matrix(rep(pi, N), nrow=N, byrow =T)
          sum_pi_prob_x <- rowSums(pi_prob_x)


          z <- pi_prob_x/sum_pi_prob_x


   # 2.2. Log likelihood computation. sum_N ln(sum_K pi * bernoulli)

          llik[it] <- sum(log(sum_pi_prob_x))
          llik

  #cat("iteration: ", it, "log likelihood: ", llik[it], "\n")
  flush.console()

    # 2.2.1. Stop if the lok likelihood has not changed significantly
        if(it > 1){
          change <- abs(llik[it]-llik[it-1])
          if(change < min_change){
            break
          }
        }

  ###  STEP 3. M-step: ML parameter estimation from the data and fractional component assignments
     # 3.1 Setting new mu. mu_ML= sum_kn(x*z)/sum_k(z)
            mu <- (t(z)%*%x)/colSums(z)

     # 3.2. Setting new pi: pi_ML= sum_k(z)/N (of all n)
             pi <- colSums(z)/N

  ### Plot of the initial values:
   if(it==1){
    plot_mu <- mu
    plot_pi <- pi
  }
}

{plot(true_mu[1,], type="o", col="blue", ylim=c(0,1), main="Plot of the true values", xlab="Dimensions"
points(true_mu[2,], type="o", col="red")
points(true_mu[3,], type="o", col="green")}

par(mfrow=c(1,2), mar=c(3,2,2,1)+0.1)
```

```
{plot(plot_mu[1,], type="o", col="blue", ylim=c(0,1), main = "Plot of the values in iteration 1", xlab=
points(plot_mu[2,], type="o", col="red")
points(plot_mu[3,], type="o", col="green")}

{plot(mu[1,], type="o", col="blue", ylim=c(0,1), main = "Plot of the final values", xlab="Dimensions", y
points(mu[2,], type="o", col="red")
points(mu[3,], type="o", col="green")}

cat("Pi values for K=3: ", "\n")
pi
cat("Mu values for K=3: ", "\n")
mu

par(mfrow=c(1,1), mar=c(5, 4, 4, 2) + 0.1)
plot(llik[1:it], type="o", main="Log-likelihood values in each iteration", ylab="Log-likelihood", xlab=
```

2.4. ANALYSIS OF RESULTS.

The number of iterations for K= 2,3,4 has been 11, 26 and 54, respectively. Also, comparing the plots of the true and the final μ values in each case, we can observe that with $K = 2$ both plots are quite similar (almost the same) while when increasing K's, the final results are every time more different than the true values. So we see that when $K = 4$, the μ estimations obtained with the EM algorithm are not as good as when K is smaller.

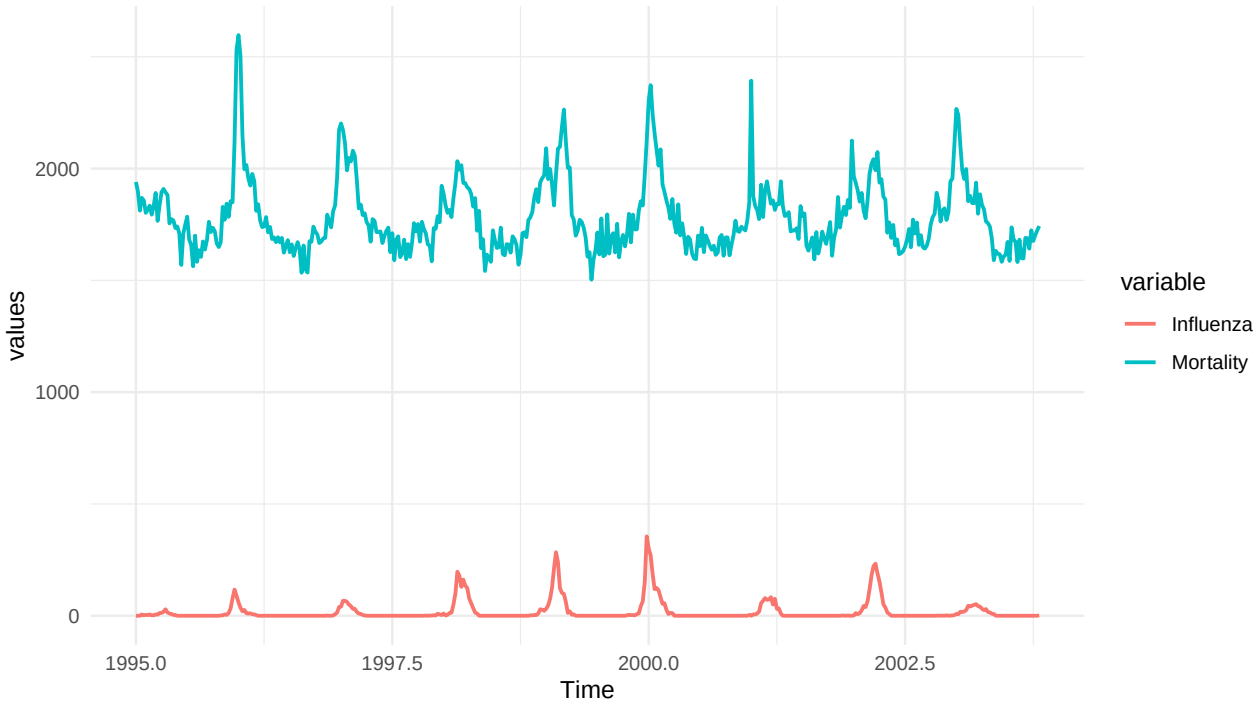
Topic 2.

Assignment 1. Using GAM and GLM to examine the mortality rates.

```
# Changing the version:
RNGversion('3.5.1')

# 0. Importing the data:
library(readxl)
data <- read_excel("influenza.xlsx")
```

1. Use time series plots to visually inspect how the mortality and influenza number vary with time (use Time as X axis). By using this plot, comment how the amounts of influenza cases are related to mortality rates.

```
# 1. Time series plot.
library(ggplot2)

# 1.1  Data frame to use in ggplot:
df <- data.frame(Time= data$Time, values= c(data$Mortality, data$Influenza),
                 variable  = rep(c("Mortality", "Influenza"), each=459))

# 1.2. Multiple line plot:
ggplot(df, aes(x = Time, y = values)) +
  geom_line(aes(color = variable), size = 0.8) +
  theme_minimal()
```



Looking at the time series of Mortality against time we can observe peaks at the first weeks of every year, which means that the highest number of mortalities are during the first weeks of each year. This is an indication that there is anual seasonality in our data.

Regarding the Influenza time series we see that there is an increase in influenza cases from 1995 to 2000 with the peak being at 2000 and then there is a relative decrease again. The peaks here also appear during the first weeks of every year.

It's obvious that the mortality rate is highly connected to influenza cases, as we can observe that during the periods of each year that the influenza cases are on their peak, the highest number of death cases are recorded at that exact time too.

2. Use gam() function from mgcv package to fit a GAM model in which Mortality is normally distributed and modelled as a linear function of Year and spline function of Week, and make sure that the model parameters are selected by the generalized cross-validation. Report the underlying probabilistic model.

In order to fit a GAM model we have used the `gam()` function indicating that the dependent variable (Mortality) is normally distributed by using the argument *family = gaussian()*. Also, we have set that the smoothing parameter estimation method should be generalized cross-validation with the argument *method = "GCV.Cp"*. Finally, in the formula argument, we have written the Week variable as *s(data$Week, k)*, where k is the amount of unique values of the Week variable, because the `s()` function helps to set up a model using spline-based smooths.

```
# 2. Fitting a GAM model.
library(mgcv)
gam_fit <- gam(data$Mortality ~ data$Year + s(data$Week, k=length(unique(data$Week))),
               family = gaussian(), data=data, method = "GCV.Cp" )
# s() sets up a model using spline-based smooths
# "GCV.Cp" to use generalized cross-validation for unknown scale parameter
```

Given that a generalized additive model is

$$EY = \alpha + s_1(X_1) + \cdots + s_p(X_p) + \sum_{j=1}^{q} \beta_j X_{p+j}$$

The probabilistic model in our case can be written as:

$$\text{Mortality} = \alpha + s(\text{Week}) + \beta_1 \cdot \text{Year} + \epsilon$$

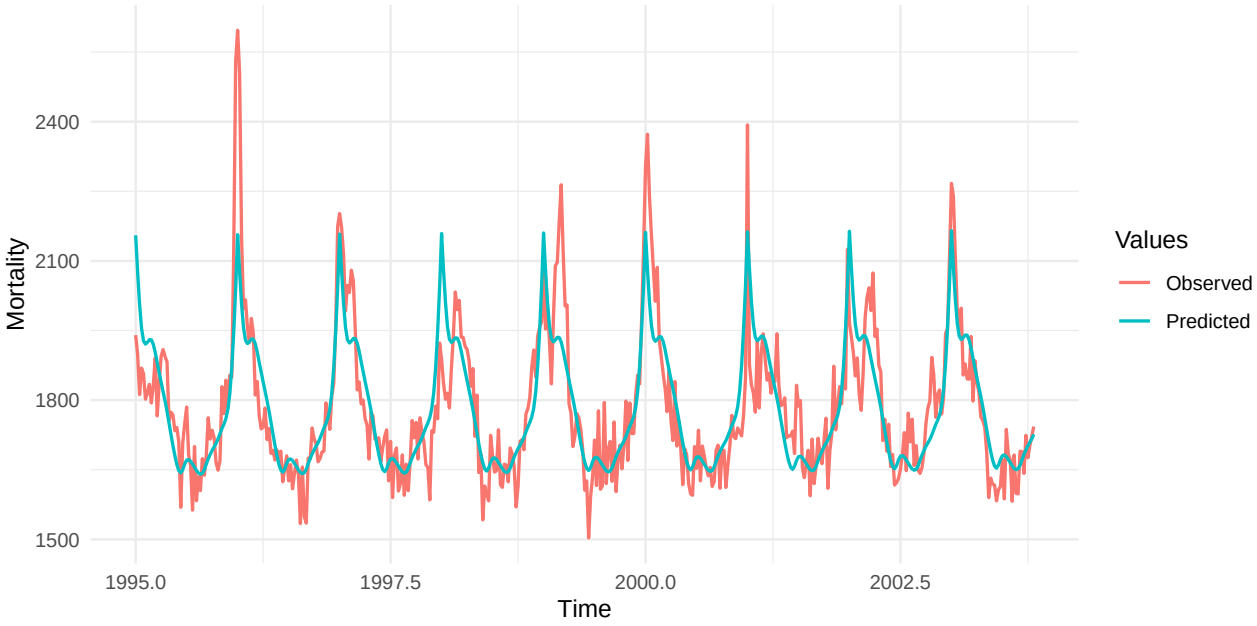
where Mortality $\sim$ Normal(μ, σ^2) and $\epsilon \sim$ Normal$(0, \sigma^2)$.

3. Plot predicted and observed mortality against time for the fitted model and comment on the quality of the fit. Investigate the output of the GAM model and report which terms appear to be significant in the model. Is there a trend in mortality change from one year to another? Plot the spline component and interpret the plot.

We already have the fitted model predictions of expected value for each observation in our gam object, in the model fitted in step (2): `gam_fit$fitted.values`.

We can create a series plot of the predicted and the observed values of mortality:

```
# 3.1. Predicted mortality.
pred <- gam_fit$fitted.values

# 3.2  Data frame to use in ggplot:
df <- data.frame(Time= data$Time, Mortality= c(data$Mortality, pred),
                 Values  = rep(c("Observed", "Predicted"), each=459))

# 3.3. Plot of predicted and observed mortality against time:
ggplot(df, aes(x = Time, y = Mortality)) +
  geom_line(aes(color = Values), size=0.7) +
  theme_minimal()
```



The quality of the fit seems to be good because the GAM model is able to predict all the peaks at the right time although the length of the peaks is always the same so that the highest values of the peaks are not always predicted correctly.

In addition, we observe a trend in mortality: mortality values appear to be cyclic. Around each beginning or end of the year (during winter) the mortality values increase significantly compared to those of the other periods of the year.

The output of the fitted GAM model is as follows:

```
summary(gam_fit)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## data$Mortality ~ data$Year + s(data$Week, k = length(unique(data$Week)))
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -680.598   3367.760  -0.202    0.840
## data$Year      1.233      1.685   0.732    0.465
##
## Approximate significance of smooth terms:
##                edf Ref.df     F p-value
## s(data$Week) 14.32  17.87 53.86  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Rank: 52/53
## R-sq.(adj) =  0.677   Deviance explained = 68.8%
## GCV = 8708.6  Scale est. = 8398.9    n = 459
```

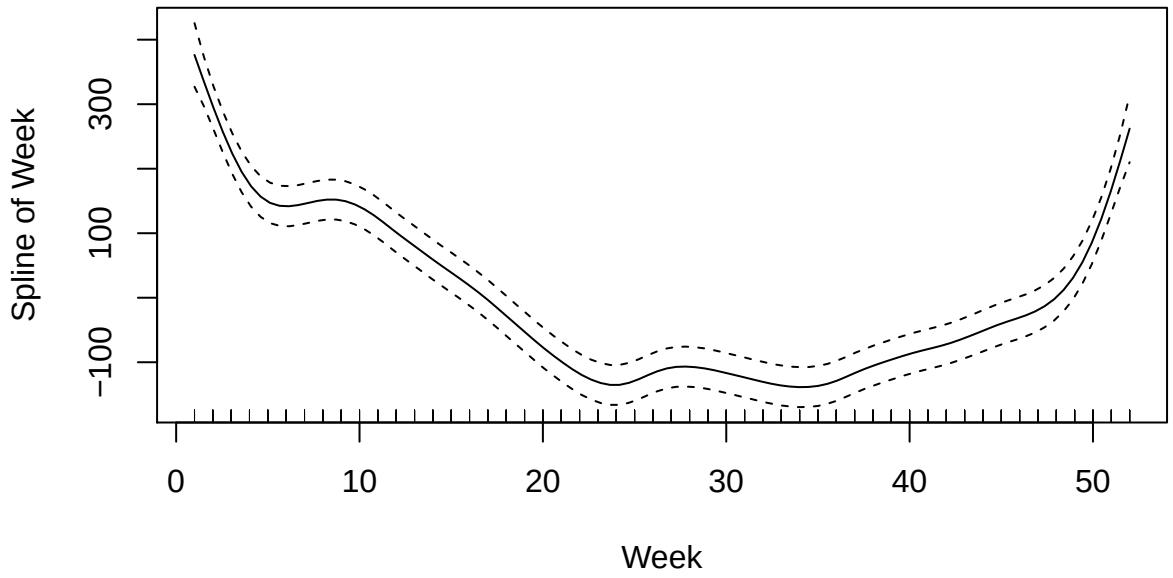
From the output values, we can see that neither the intercept (α) nor the coefficient associated with the Year are significant in the model because their p-values are much greater than 0,05 (at 95% confidence level)[4]. This means that the variable Year is not relevant to explain the Mortality values. On the other hand, the spline of the Week is smaller than 0.05 meaning that is significant.

Also, we observe that the adjusted R^2 is 0.677 which indicates that 67.7% of all the variability of the response data is explained by the model.

Finally, we will plot the spline component:

```
plot(gam_fit, xlab="Week", ylab="Spline of Week")
```

[4]When p-value < 0.05 we reject $H_0 : \beta_i = 0$ which expresses that the coefficient i is not significant (equals 0).



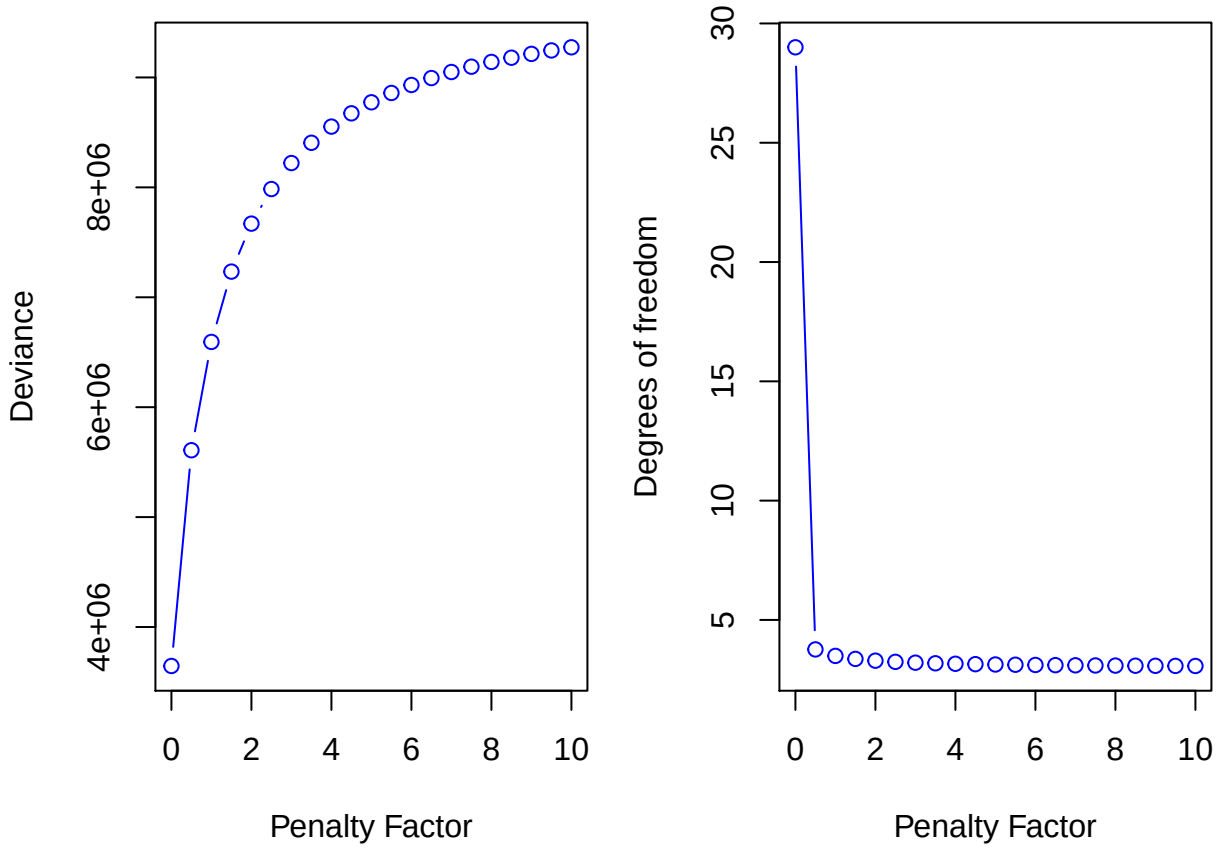
The dashed curves show pointwise 2σ limits for the fitted curve. Looking at the spline plot we can detect the seasonal pattern, where the values are very high during the first weeks, then they drop quickly and in the very ending of the year they rise again.

4. Examine how the penalty factor of the spline function in the GAM model from step 2 influences the estimated deviance of the model. Make plots of the predicted and observed mortality against time for cases of very high and very low penalty factors. What is the relation of the penalty factor to the degrees of freedom? Do your results confirm this relationship?

We can set different values for the smoothing penalty factor with the *sp* argument of the function `s()`. So, we will fit a GAM model for penalty factor values sp $= 0, 0.1, 0.2..., 4.9, 5$ and we will see how the deviance (`gam_fit$deviance`) and the degrees of freedom (difference between the number of observations included in the model and the model residual degrees of freedom) change in each case.

The plots of the different smoothing penalty factors against the deviance and the degrees of freedom of the model are:

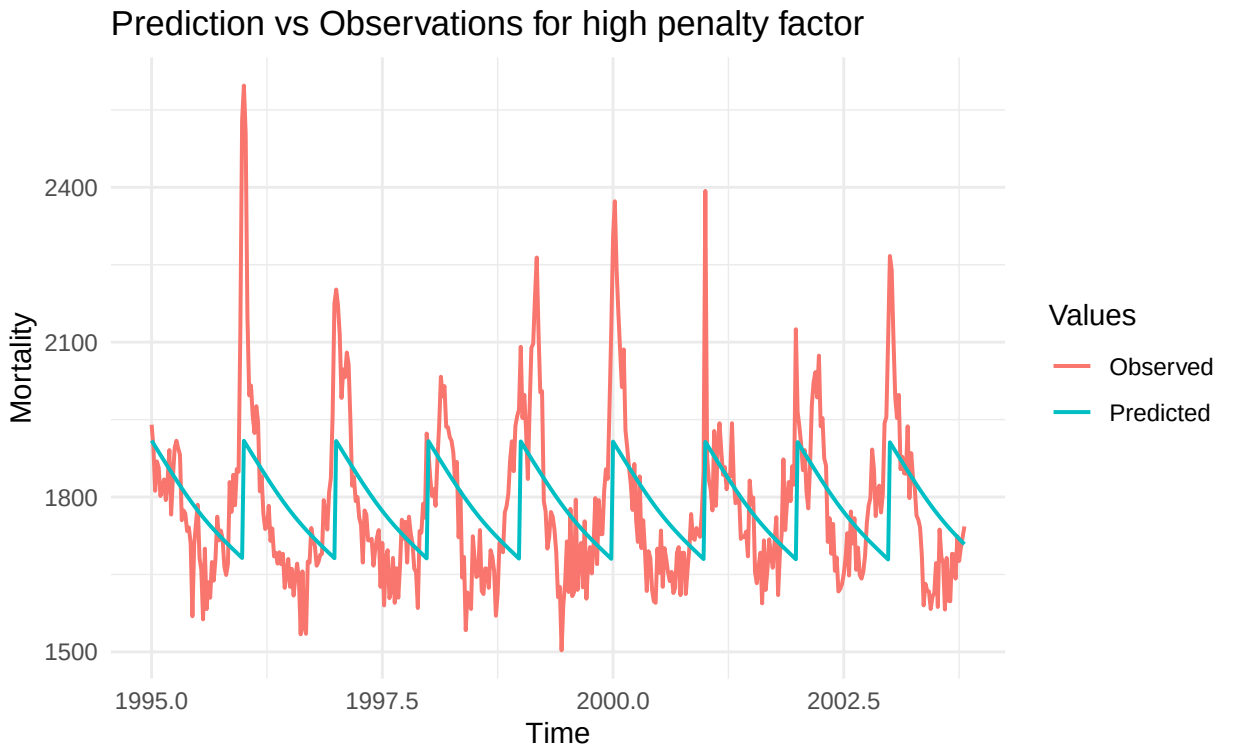
```
# 4.1. Fitting GMA models with different smoothing penalty factors.
penalty <- seq(from=0, to= 10, by=0.5)

dev_values <- vector()
d_freedom <- vector()
for(i in penalty){
  gam_fit <- gam(data$Mortality ~ data$Year + s(data$Week, k=length(unique(data$Week)), sp=i),
                 family = gaussian(), data=data, method = "GCV.Cp" )
  dev_values <- c(dev_values, gam_fit$deviance)
  d_freedom <- c(d_freedom, (nrow(data)-gam_fit$df.residual))
}

# 4.2. Plots of penalty factor vs deviance and vs degrees of freedom.
par(mfrow=c(1,2), mar=c(3.9, 3.8, 1, 1) + 0.1)
plot(penalty, dev_values, type="b", col="blue", xlab="Penalty Factor", ylab="Deviance")
plot(penalty, d_freedom, type="b", col="blue", xlab="Penalty Factor", ylab="Degrees of freedom")
```



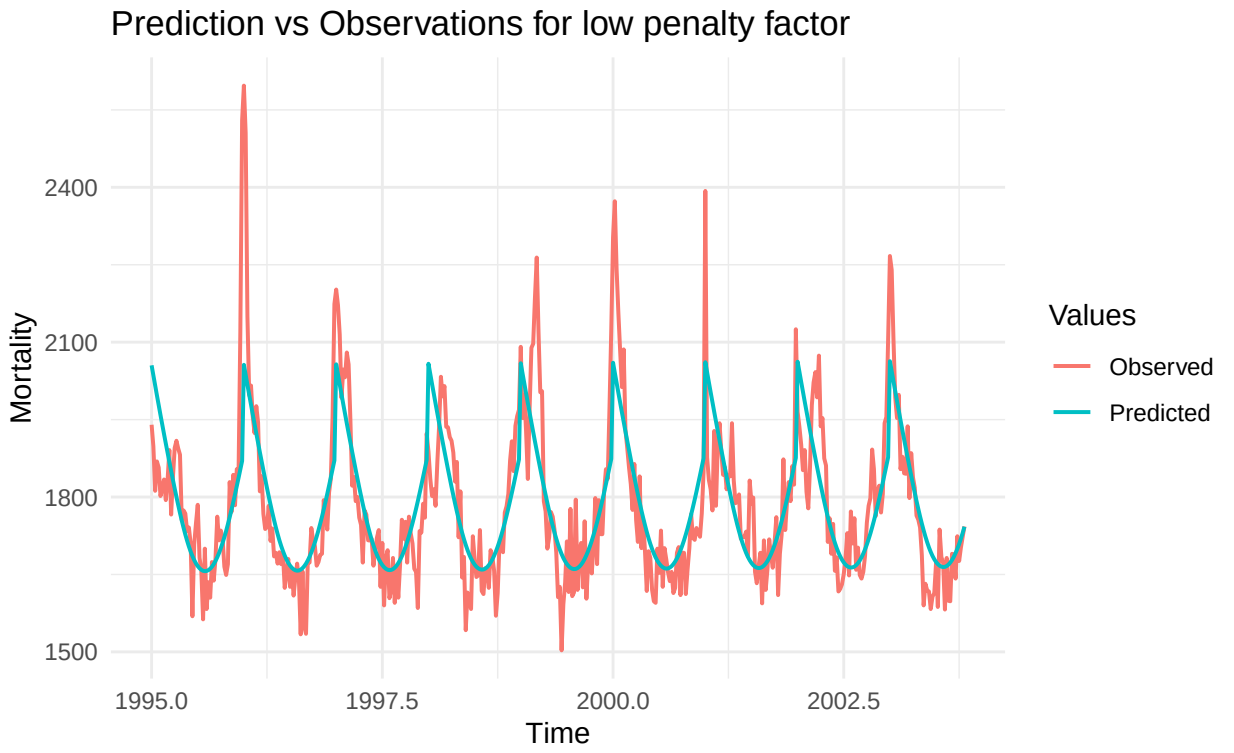
The penalty factor penalizes the coefficients in order to control the degree of smoothness. So, when a high penalty factor is used to fit the GMA model, the smoothness performed is higher and as a result, we obtain a simpler model. From the plots, we observe this because as the penalty factor increases, the degrees of freedom decrease and the deviance increase.

Now, we will fit two different models: one with a high penalty factor (sp=10) and one with a low penalty factor (sp=0.1). Then, if we plot the the predicted and observed mortality values against time we obtain the following plots:

```
# 4.3. Plot of the predicted and observed mortality against time for high penalty factor.
gam_fit_high <- gam(data$Mortality ~ data$Year + s(data$Week, k=length(unique(data$Week)), sp=10),
                    family = gaussian(), data=data, method = "GCV.Cp" )
pred_high <- gam_fit_high$fitted.values

df_high <- data.frame(Time= data$Time, Mortality= c(data$Mortality, pred_high),
                  Values  = rep(c("Observed", "Predicted"), each=459))

ggplot(df_high, aes(x = Time, y = Mortality)) +
  geom_line(aes(color = Values), size=0.7) +
  theme_minimal() + ggtitle("Prediction vs Observations for high penalty factor")
```



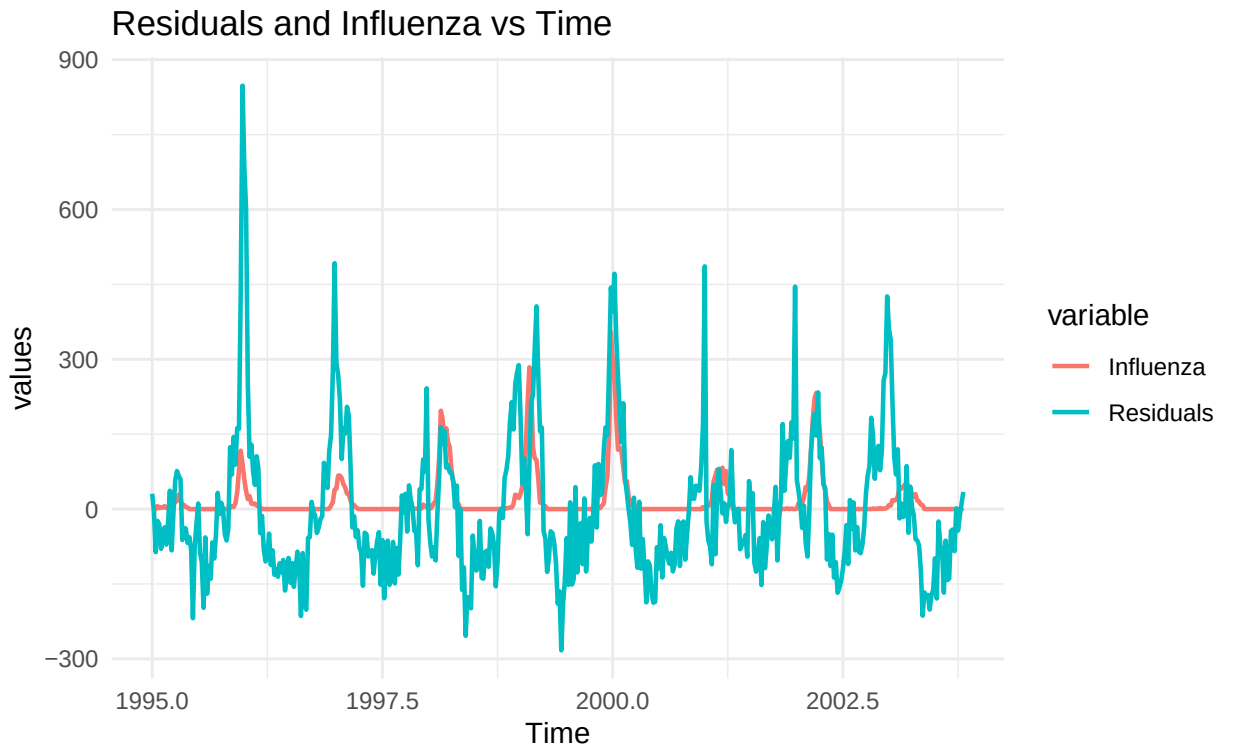

```
# 4.4. Plot of the predicted and observed mortality against time for low penalty factor.
gam_fit_low <- gam(data$Mortality ~ data$Year + s(data$Week, k=length(unique(data$Week)), sp=0.1),
                   family = gaussian(), data=data, method = "GCV.Cp" )
pred_low <- gam_fit_low$fitted.values

df_low<- data.frame(Time= data$Time, Mortality= c(data$Mortality, pred_low),
                Values  = rep(c("Observed", "Predicted"), each=459))

ggplot(df_low, aes(x = Time, y = Mortality)) +
  geom_line(aes(color = Values), size=0.7) +
  theme_minimal() + ggtitle("Prediction vs Observations for low penalty factor")
```



These plots confirm what we said at the beginning of this question: the higher the penalty factor, the simpler the model. So, the model with a low penalty factor predicts better than the model with a high penalty factor.

5. Use the model obtained in step 2 and plot the residuals and the influenza values against time (in one plot). Is the temporal pattern in the residuals correlated to the outbreaks of influenza?

```
# 5 Plot of the residuals and the influenza values against time
df <- data.frame(Time= data$Time, values= c(gam_fit$residuals, data$Influenza),
                 variable  = rep(c("Residuals", "Influenza"), each=459))

ggplot(df, aes(x = Time, y = values)) +
  geom_line(aes(color = variable), size = 0.8) +
  theme_minimal()+ ggtitle("Residuals and Influenza vs Time")
```



Looking at the plot of influenza and residuals against time we can see that there is a correlation between them. More precisely, we observe that the peaks of the residuals follow in some way the peaks of influenza.

6. Fit a GAM model in R in which mortality is be modelled as an additive function of the spline functions of year, week, and the number of confirmed cases of influenza. Use the output of this GAM function to conclude whether or not the mortality is influenced by the outbreaks of influenza. Provide the plot of the original and fitted Mortality against Time and comment whether the model seems to be better than the previous GAM models.

In this case, we want to fit the following probability model:

$$\text{Mortality} = \alpha + s(\text{Year}) + s(\text{Week}) + s(\text{Influenza}) + \epsilon$$

where Mortality $\sim$ Normal(μ, σ^2), $\epsilon \sim$ Normal$(0, \sigma^2)$ and the dimension of the basis used to represent the smooth term (argument k) is the number of unique values of each variable in each case.

The output of the fitted GAM model is:

```
# 6.1. Fitting the GAM model.
add_gam_fit <- gam(data$Mortality ~ s(data$Year, k=length(unique(data$Year))) + s(data$Week, k=length(u
                     s(data$Influenza, k=length(unique(data$Influenza))) ,
                   family = gaussian(), data=data, method = "GCV.Cp" )

# 6.2. Model output.
summary(add_gam_fit)
```

```
##
## Family: gaussian
## Link function: identity
##
## Formula:
## data$Mortality ~ s(data$Year, k = length(unique(data$Year))) +
##     s(data$Week, k = length(unique(data$Week))) + s(data$Influenza,
```

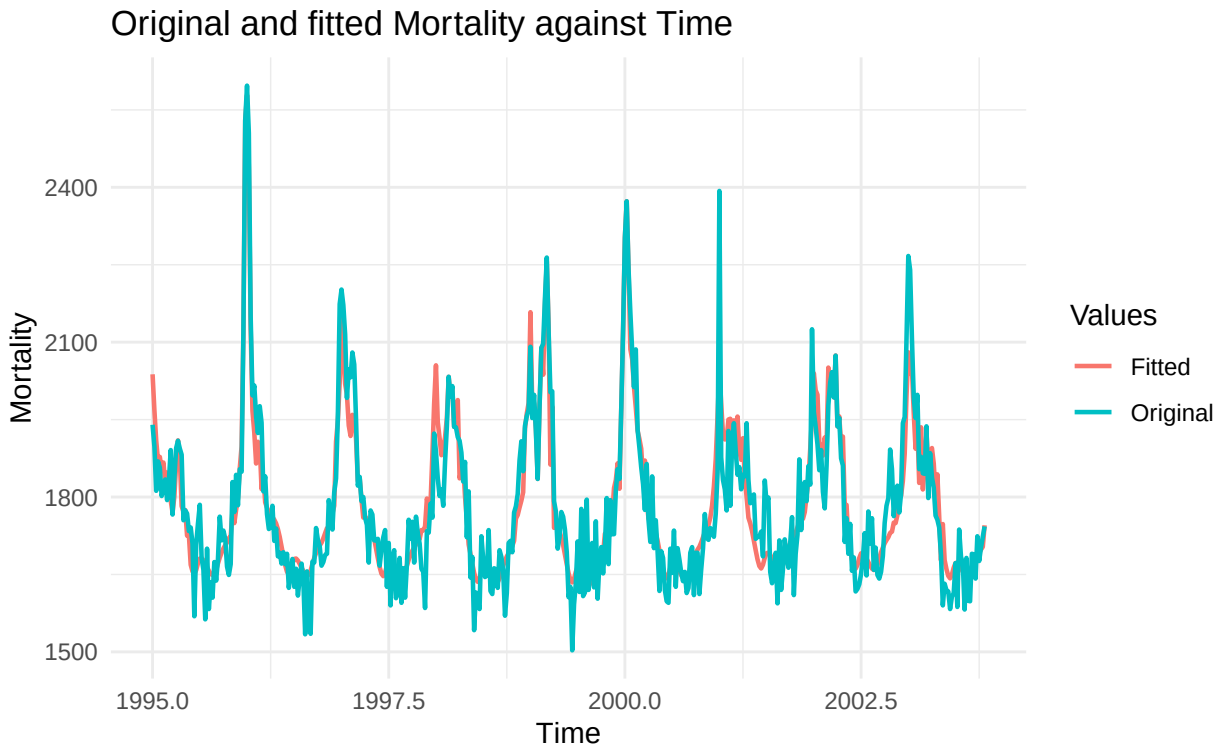
```
##     k = length(unique(data$Influenza)))
##
## Parametric coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1783.765      3.198   557.8   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Approximate significance of smooth terms:
##                       edf Ref.df      F p-value
## s(data$Year)        4.587  5.592  1.500   0.178
## s(data$Week)       14.431 17.990 18.763  <2e-16 ***
## s(data$Influenza) 70.094 72.998  5.622  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Rank: 134/144
## R-sq.(adj) =  0.819   Deviance explained = 85.4%
## GCV = 5840.5  Scale est. = 4693.7    n = 459
```

We can observe that in this new model, the only coefficient that seems to be not significant is the spline of the Year. Then, as the spline of Influenza is smaller than 0.05, we conclude that the variable is significant to explain the behaviour of the Mortality and consequently, the mortality is somehow influenced by the outbreaks of influenza.

Also, we can see that this new model manages to explain nearly 82% of all the variability of Mortality. Since the model fitted in question 2 explains 67.7%, we can conclude that this model is better.

Lastly, we will plot the original and fitted Mortality against Time:

```r
# 6.3. Plot of the original and fitted Mortality against Time
pred <- add_gam_fit$fitted.values

df <- data.frame(Time= data$Time, Mortality= c(data$Mortality, pred),
                 Values  = rep(c("Original", "Fitted"), each=459))

ggplot(df, aes(x = Time, y = Mortality)) +
  geom_line(aes(color = Values), size = 0.8) +
  theme_minimal()+ ggtitle("Original and fitted Mortality against Time")
```



Plotting the original and fitted data of Mortality we can see that the predictions are very good and catch very well the seasonal changes of the data. Comparing this plot with the previous one shown in step (2) we observe that this one has a better predictive performance.

Assignment 2. High-dimensional methods.

1.Perform nearest shrunken centroid classification of training data in which the threshold is chosen by cross-validation. Provide a centroid plot and interpret it. How many features were selected by the method? List the names of the 10 most contributing features and comment whether it is reasonable that they have strong effect on the discrimination between the conference mails and other mails? Report the test error.

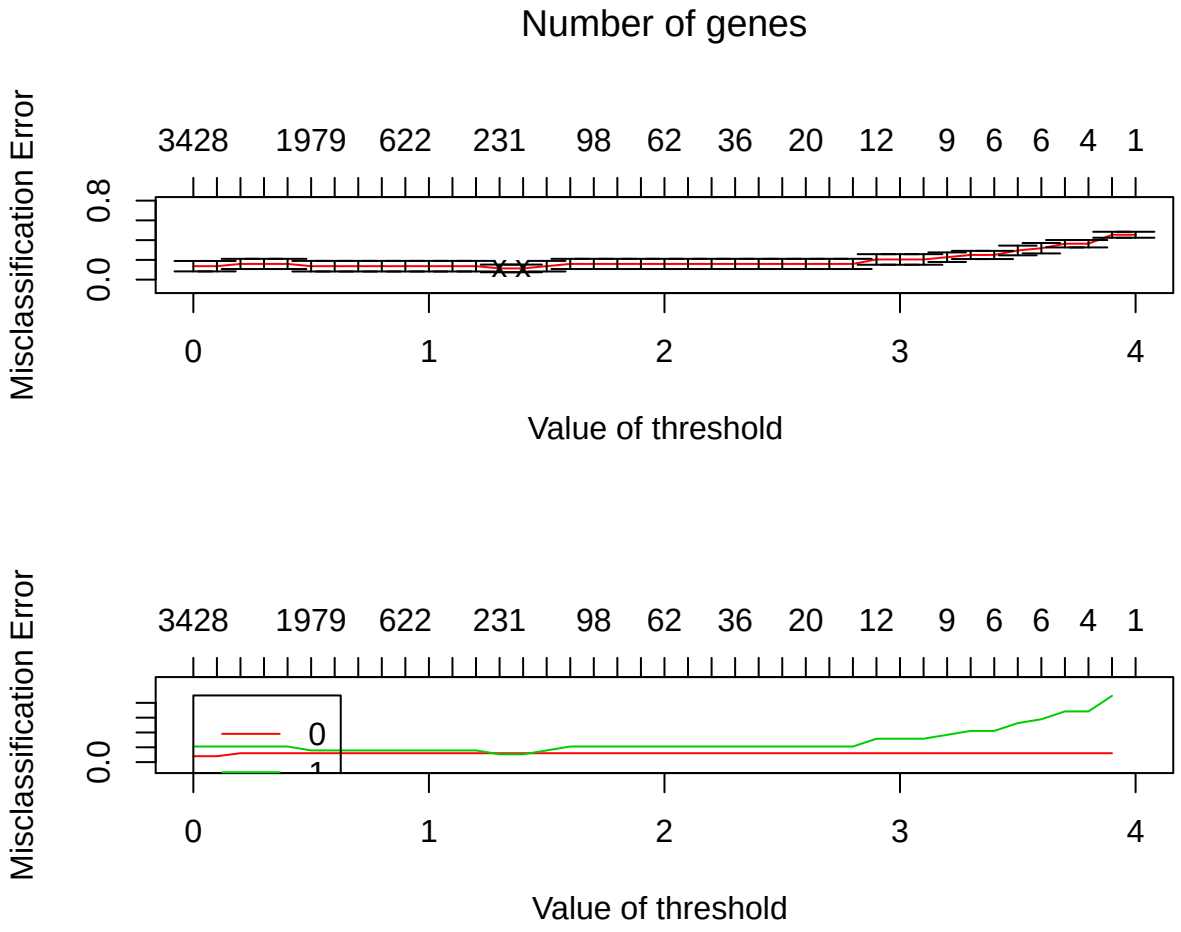
We have performed the NSC classification model using the function `pamr.train()` and then, the obtained model has been used in the `pamr.cv()` function in order to choose the optimal threshold by cross validation.

```
# 1.1. Importing the data:
data <- read.csv(file = "data.csv", sep=";", header=TRUE, stringsAsFactors=FALSE, fileEncoding="latin1")
data$Conference <- as.factor(data$Conference)

# 1.2. Divide data into training and test sets:
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.7))
train=data[id,]
test=data[-id,]

# 1.3. Nearest shrunken centroid classification with threshold chosen by cross-validation.
library(pamr)

rownames(train)=1:nrow(train)
x_train=t(train[,-4703])
y_train=train[[4703]]
```

```
mydata_train=list(x=x_train,y=as.factor(y_train),geneid=as.character(1:nrow(x_train)),
                  genenames=rownames(x_train))

## Fitting the NSC model
model = pamr.train(mydata_train, threshold=seq(0, 4, 0.1))

## Selecting the optimal threshold by cross-validation.
cvmodel = pamr.cv(model,mydata_train)
optimal <- cvmodel$threshold[which(min(cvmodel$error)==cvmodel$error)]

## Fitting the optimal NSC model
optimal_model = pamr.train(mydata_train, threshold=1.4)
```

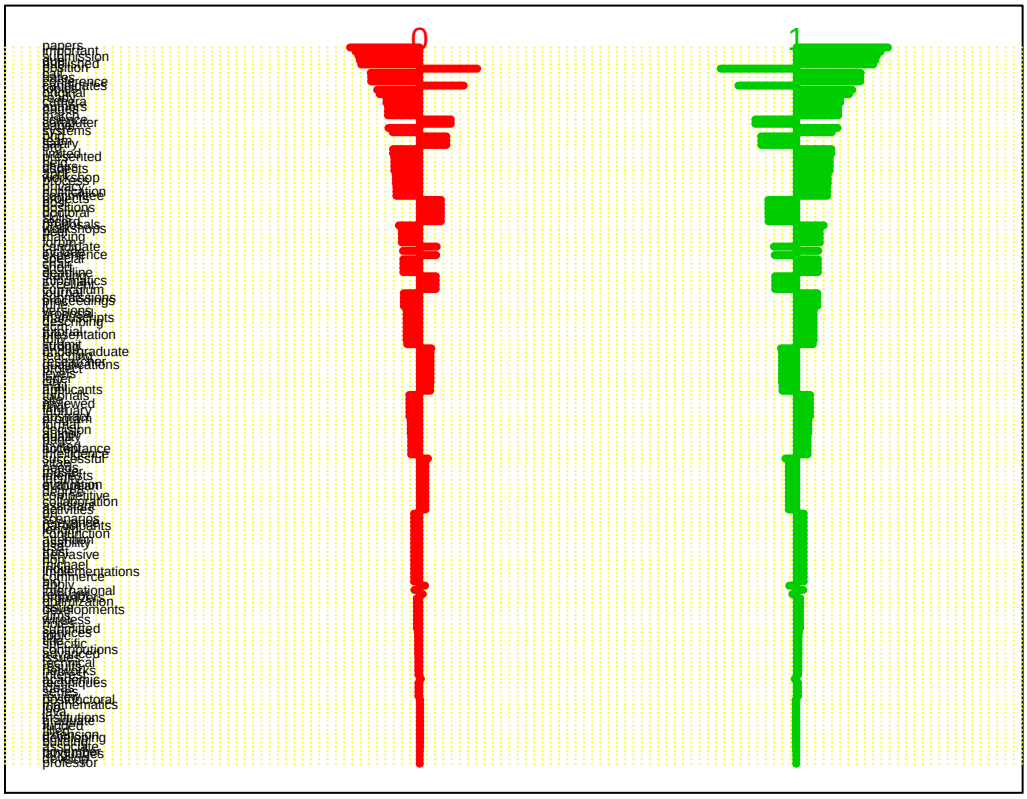
From the following plot it can be seen graphically that the lowest misclassification error is obtained when the value of the threshold is 1.3 or 1.4. Also, we can observe that the number of features selected decreases as the threshold increases, so we will chose a threshold value of 1.4 because by selecting 170 features we keep having the minimum error.

```
pamr.plotcv(cvmodel)
```



Also, we will provide a centroid plot of the NSC model fitted with the optimal threshold value:

```
pamr.plotcen(optimal_model, mydata_train, threshold=1.4)
```



The function `pamr.plotcen()` plots the shrunken class centroids for each class, in our case class 1 is "announces of conferences" and class 0 is "everything else". We can see two centroids profiles d_{kj} for the data. The red and green bars are shrunken versions d'_{kj} of these centroids, obtained by soft-thresholding using $\Delta = 1.4$.

The list of the names of the 10 most contributing features is as follows:

```
contribution <- pamr.listgenes(optimal_model, mydata_train,  threshold=1.4)
```

```
df <- as.data.frame(colnames(data)[as.numeric(contribution[,1])][1:10])
colnames(df) <- "Features"
df
```

```
##      Features
## 1      papers
## 2   important
## 3  submission
## 4         due
## 5   published
## 6    position
## 7        call
## 8  conference
## 9       dates
## 10 candidates
```

It seams reasonable that those features have strong effect on the discrimination between the conference mails and other mails as they are all features that can be related to a conference. For example, submission may has to do with the submission of projects that will be presented during the conference, published may concern published papers, dates are also and important thing about a conference and of course the candidates that

will participate there. So, it seems reasonalbe that all those features played a leading role in the classification of the mails.

Finally, the misclassification error for the test dataset is:

```
x_test=t(test[,-4703])
test_pred <- pamr.predict(fit=optimal_model, newx= x_test, threshold=1.4)
mean(test_pred != test$Conference)
```

```
## [1] 0.1
```

2. Compute the test error and the number of the contributing features for the following methods fitted to the training data:

a. Elastic net with the binomial response and alpha = 0.5 in which penalty is selected by the cross-validation
b. Support vector machine with "vanilladot" kernel.

Compare the results of these models. Which model would you prefer and why?

The elastic net model has been performed by using the function `glmnet()` and then, the obtained model has been used in the `cv.glmnet()` function in order to choose the optimal lambda (the penalty factor) by cross validation. This value can be obtained with `cv.net_fit$lambda.min` and in this case $\lambda_{min} = 0.1655087$. So, the optimal elastic net model is the one fitted with this lambda.

The SVM model with "vanilladot" kernel has been performed with the function `ksvm()` from the package **kernlab**. We have set in the *kernel* argument that the kernel function we want to use is the vanilladot (linear kernel).

```
# 2.0. Variables:
x_train <- as.matrix(train[,-4703])
y_train <- train[,4703]

x_test <- as.matrix(test[,-4703])
y_test <- test[,4703]

# 2.1. Elastic net with binomial response
library(glmnet)
net_fit <- glmnet(x_train, y_train, alpha = 0.5, family = "binomial")
cv.net_fit <- cv.glmnet(x_train, y_train, alpha = 0.5, family="binomial")
net_fit_optimal <- glmnet(x_train, y_train, alpha = 0.5, family = "binomial",
                          lambda = cv.net_fit$lambda.min)

# 2.2. SPV ith vanilladot kernel.
library(kernlab)
svm_fit <- ksvm(x_train, y_train, scaled=FALSE, kernel="vanilladot")
```

The total number of contributing features is as follows:

```
contributing_net <- colnames(data)[which(net_fit_optimal$beta !=0 )]
coeff_net <- net_fit_optimal$beta[which(net_fit_optimal$beta !=0 )]
cat("In the elastic net model is:", length(contributing_net), "\n")
```

```
## In the elastic net model is: 32
```

```
contributing_svm <- colnames(data)[SVindex(svm_fit)]
coeff_svm <- coef(svm_fit)[[1]]
cat("In the SVM model is:", length(contributing_svm), "\n")
```

```
## In the SVM model is: 43
```

```
df <- as.data.frame(cbind(contributing_net[1:10], round(coeff_net[1:10],4),
                          contributing_svm[1:10], round(coeff_svm[1:10],5)))
colnames(df) <- c("Elastic net","Coefficients", "SVM","Coefficients")
```

Finally, the misclassification errors for the test dataset is:

```
pred_net <- predict(net_fit_optimal, x_test, type = "class")
pred_svm <- predict(svm_fit, x_test, type = "response")
cat("For the elastic net model:", mean(pred_net != test$Conference), "\n")
```

```
## For the elastic net model: 0.1
```

```
cat("For the SVM model:", mean(pred_svm != test$Conference), "\n")
```

```
## For the SVM model: 0.05
```

- Comparison with results in step (1).

```
df <- as.data.frame(cbind(c("NSC", "Elastic net", "SVM"), c(170,32,43), c(0.1,0.1,0.05)))
colnames(df) <- c("Model", "Number of features", "Test error")
df
```

```
##         Model Number of features Test error
## 1         NSC                170        0.1
## 2 Elastic net                 32        0.1
## 3         SVM                 43       0.05
```

The Support vector machine with "vanilladot" kernel method is the best model: it has a few more features than the Elastic net model but the misclassification error for the test data is 0.05 units lower. The nearest shrunken centroid classification method is the worst among these three models because although it has the same test error as the elastic net model, it has model 6 time more features and thus it's a much more complex model.

3. Implement Benjamini-Hochberg method for the original data, and use t.test() for computing p-values. Which features correspond to the rejected hypotheses? Interpret the result.

The Benjamini-Hochberg method has been perform in order to asses the significance of each of the $M = 4702$ features. First, we have computed a p-value for each feature using the theoretical t-distribution probability, obtained using the function `t.test()`. Then, using this set of p-values, we have tested the followiing hypothesis:

$$H_{0j}: \text{ feature j has no "effect" on Conference}$$
$$H_{1j}: \text{ feature j has an "effect" on Conference}$$

for all j$= 1, 2, \ldots, M$.

Given that the BH threshold is the largest j for which the p-value is smaller than $\alpha\frac{j}{M}$, setting $\alpha = 0.05$, we will reject those hypothesis with a p-value smaller than the threshold and so, we will conclude that the features associated to those hypothesis are not significant to explain our target variable "Conference".

After performing the method, we observe that 39 features have no effect on Conference. These features and its p-values are the following:

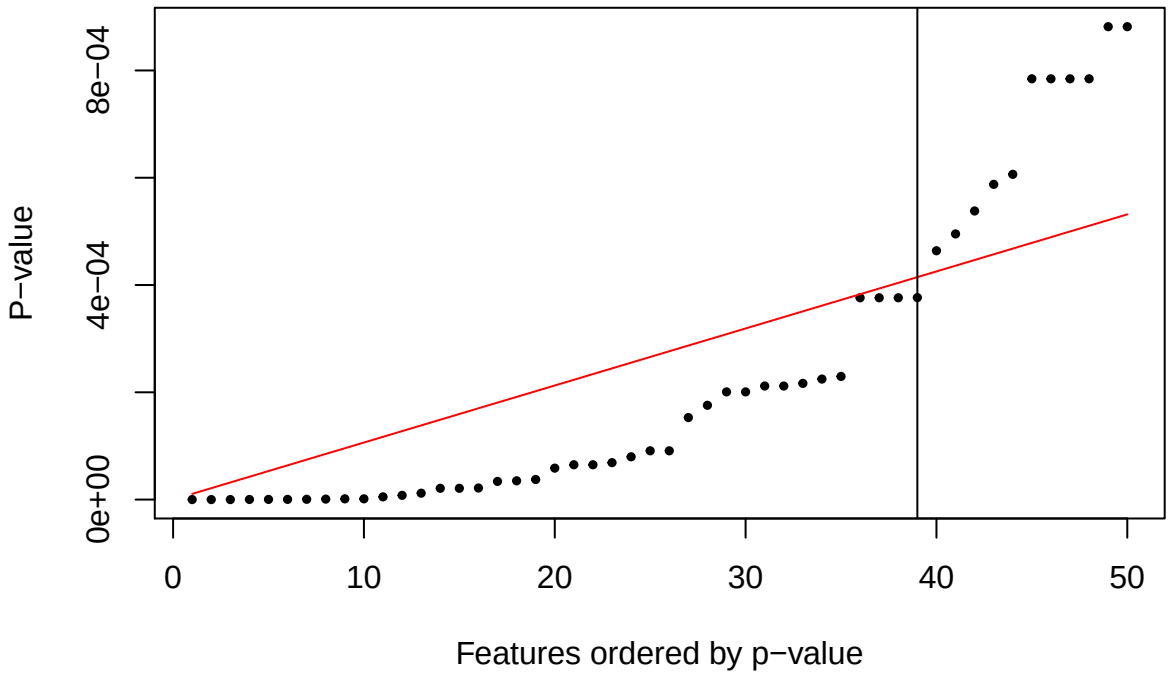
```
# Variables from the original data:
x <- as.matrix(data[,-4703])
y <- as.factor(data[,4703])

# Computing p-value for each variable (1,2,..,p)
alpha <- 0.05
```

```r
M <- ncol(x)
pval <- data.frame("Features"=colnames(x), "pvalues"=rep(0,M))

for(j in 1:M){
  ttest <- t.test(x[,j] ~ y)
  pval[j,2] <- ttest$p.value
}

# Ordering p-values
sort_pval <- pval[order(pval$pvalues, decreasing = F),]
sort_pval$Index <- 1:M
sort_pval$critical <- (sort_pval$Index *alpha)/M

# Non significant features:
# max(sort_pval$Index[which(sort_pval$pvalues < sort_pval$critica)]) =39
rejected <- sort_pval[which(sort_pval$pvalues < sort_pval$critical),1:2]
library(rowr)
cbind.fill(rejected[1:20,],rejected[21:39,], fill=NA)
```

```
##      Features      pvalues      Features      pvalues
## 1      papers 1.116910e-10           due 6.488781e-05
## 2  submission 7.949969e-10      original 6.488781e-05
## 3    position 8.219362e-09  notification 6.882210e-05
## 4   published 1.835157e-07        salary 7.971981e-05
## 5   important 3.040833e-07        record 9.090038e-05
## 6        call 3.983540e-07        skills 9.090038e-05
## 7  conference 5.091970e-07          held 1.529174e-04
## 8  candidates 8.612259e-07          team 1.757570e-04
## 9       dates 1.398619e-06         pages 2.007353e-04
## 10      paper 1.398619e-06      workshop 2.007353e-04
## 11     topics 5.068373e-06     committee 2.117020e-04
## 12    limited 7.907976e-06   proceedings 2.117020e-04
## 13  candidate 1.190607e-05         apply 2.166414e-04
## 14     camera 2.099119e-05        strong 2.246309e-04
## 15      ready 2.099119e-05 international 2.295684e-04
## 16    authors 2.154461e-05        degree 3.762328e-04
## 17        phd 3.382671e-05     excellent 3.762328e-04
## 18   projects 3.499123e-05          post 3.762328e-04
## 19        org 3.742010e-05     presented 3.765147e-04
## 20     chairs 5.860175e-05          <NA>         <NA>
```

Finally, we can plot the slope ($\frac{0.05}{4702} \cdot j$) and the ordered p-values:

```r
# Plot:
{plot(1:50, sort_pval$pvalues[1:50], pch=19, cex=0.5, ylab="P-value", xlab="Features ordered by p-value
points(sort_pval$critical[1:50], type="l", col="red")
abline(v=39)}
```



We observe that the 39th feature is the largest j for wich the p-value falls bellow the line.

OTHER EXERCICES

Assignment 1.

The data file video.csv contains characteristics of a sample of Youtube videos. Import data to R and divide it randomly (50/50) into training and test sets.

1. Perform principal component analysis using the numeric variables in the training data except of "utime" variable. Do this analysis with and without scaling of the features. How many components are necessary to explain more than 95% variation of the data in both cases? Explain why so few components are needed when scaling is not done. (2p)

```
data0=read.csv("video.csv")

data1=data0
data1$codec=c()

n=dim(data1)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data1[id,]
test=data1[-id,]
```

- Variation without scaling:

```
data11=data1
data11$utime=c()
res=prcomp(data11)
lambda=res$sdev^2
sprintf("%2.3f",cumsum(lambda)/sum(lambda)*100)
```

```
##  [1] "99.723"  "99.935"  "99.989"  "100.000" "100.000" "100.000" "100.000" "100.000" "100.000" "100.
```

- Variation with scaling:

```
res=prcomp(scale(data11))
lambda=res$sdev^2
sprintf("%2.3f",cumsum(lambda)/sum(lambda)*100)
```

```
##  [1] "35.076"  "50.934"  "63.495"  "70.112"  "76.285"  "81.969"  "87.425"  "92.320"  "95.623"  "97.5
```

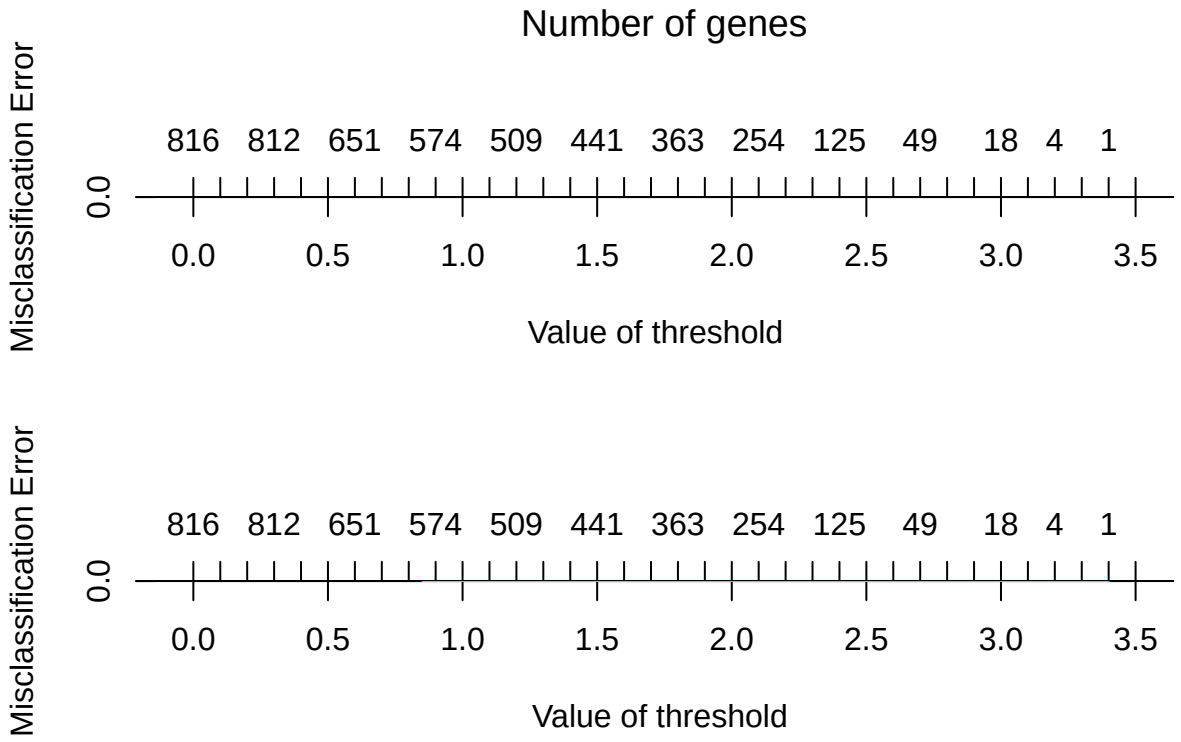
1 component in unscaled data, 9 components in the scaled data are needed to explain 95% variation. The reason in that the original data is on a very different scale variation in one feature dominates variation in the other features.

2. Use only first 100 rows from the entire data and all numeric variables and create interaction variables up to the third order with the following code: data=t(apply(as.matrix(your_data_with_100rows_a 1, combn, 3, prod))

Use the obtained matrix to perform a Nearest Shrunken Centroid (NSC) analysis in which "codec" is target and all other interaction variables are features. Obtain the cross-validation plot and interpret it in terms of bias-variance tradeoff. How does model complexity change when the threshold λ increases? (2p)

```
data11=t(apply(as.matrix(data1[1:100,1:18]), 1, combn, 3, prod))
library(pamr)
mydata=as.data.frame(scale(data11))
rownames(mydata)=1:nrow(mydata)
x=t(mydata)
y=data0$codec[1:100]
mydata1=list(x=x,y=as.factor(y),geneid=as.character(1:nrow(x)), genenames=rownames(x))
model=pamr.train(mydata1,threshold=seq(0,4, 0.1))
set.seed(12345)
cvmodel=pamr.cv(model,mydata1)
```

```
pamr.plotcv(cvmodel)
```



Higher threshold leads to less complex models, the higher the complexity the lower the bias and the higher is the variance.

3. Use the cross-validated NSC model to extract the threshold giving the optimal value of the log- likelihood (log-likelihood values are also available in the cross-validated model). Why is the multinomial log-likelihood applied for this model? (1p)

```
cvmodel$threshold[which.max(cvmodel$loglik)]
```

```
## [1] 2.5
```

Multinomial likelihood is used because this is a classification problem.

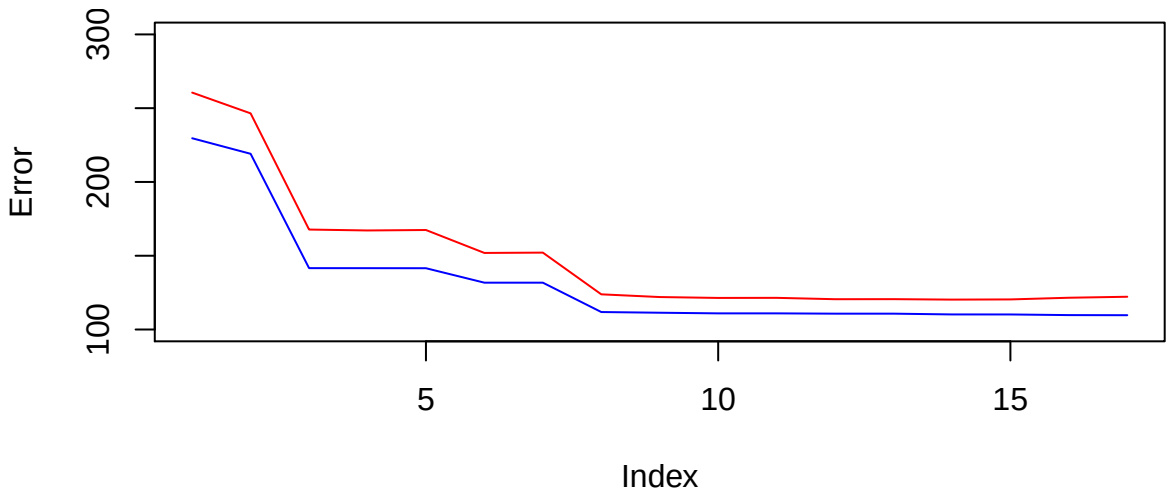
2 TDE. Write a code that fits a principal component regression("ultimate" as response and all scaled numerical variables as features) with M components to the training data and estimates the training and test errors, do this for all feasibleM values. Plot dependence of the training and test errors on M and explain this plot in terms of bias-variance tradeoff. (Hint: prediction function for principal component regression has some peculiarities, see predict.mvr) (2p)

```
library(pls)
```

```
##
## Attaching package: 'pls'
```

```
## The following object is masked from 'package:stats':
##
##     loadings
```

```
trE=numeric(17)
testE=numeric(17)

for (i in 1:17){
  pcrN=pcr(utime~., 17, data=train,  scale=T)
  Yf=predict(pcrN, ncomp=i)
  Yt=predict(pcrN, newdata=test, ncomp=i)
  trE[i]=mean((train$utime-Yf)^2)
  testE[i]=mean((test$utime-Yt)^2)
}

{plot(testE, type="l", col="red", ylim=c(100,300), ylab="Error")
points(trE, type="l", col="blue")}
```



When the number of components increases, the model becomes more complex and the bias goes down while variance goes up. The optimal model should have the lowest test error, in this case M=14. However, there are much simpler models with similar test errors, for ex. M=8.

3 TDE. Use PCR model with M = 8 and report a fitted probabilistic model that shows the connection between the target and the principal components. (1p)

```
pcrF=pcr(utime~., 8, data=train, validation="none", scale=T)
```

```
mean(residuals(pcrF)^2)
```

```
## [1] 156.092
```

```
Yloadings(pcrF)
```

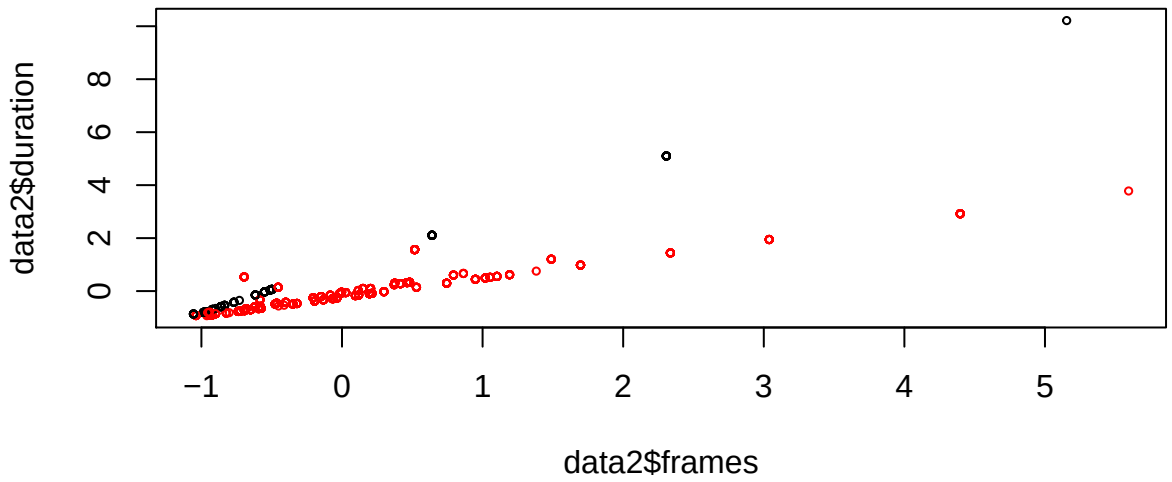
```
##
## Loadings:
##       Comp 1 Comp 2 Comp 3 Comp 4 Comp 5 Comp 6 Comp 7 Comp 8
## utime -1.196  1.966  6.048  0.156  0.195 -3.136         4.988
##
##                Comp 1 Comp 2 Comp 3 Comp 4 Comp 5 Comp 6 Comp 7 Comp 8
## SS loadings     1.429  3.865 36.576  0.024  0.038  9.838  0.008 24.878
## Proportion Var  1.429  3.865 36.576  0.024  0.038  9.838  0.008 24.878
## Cumulative Var  1.429  5.295 41.871 41.895 41.933 51.771 51.779 76.657
```

- Equation:

$$\text{Ultimate}_{scaled} \sim N(-1.196 \cdot Comp1 + 1.966 \cdot Comp2 + 6.048 \cdot Comp3 + 0.156 \cdot Comp4 + 0.195 \cdot Comp5 - 3.136 \cdot Comp6 + 4.988 \cdot Comp8$$

4. Use original data to create variable "class" that shows "mpeg" if variable "codec" is equal to "mpeg4", and "other" for all other values of "codec". Create a plot of "duration" versus "frames" where cases are colored by "class". Do you think that the classes are easily separable by a linear decision boundary? (1p)

```
data2=data0
data2$class=ifelse(data2$codec=="mpeg4", "mpeg4", "other")
data2$codec=c()
data2$frames=scale(data2$frames)
data2$duration=scale(data2$duration)

plot(data2$frames,data2$duration, col=as.factor(data2$class), cex=0.5)
```

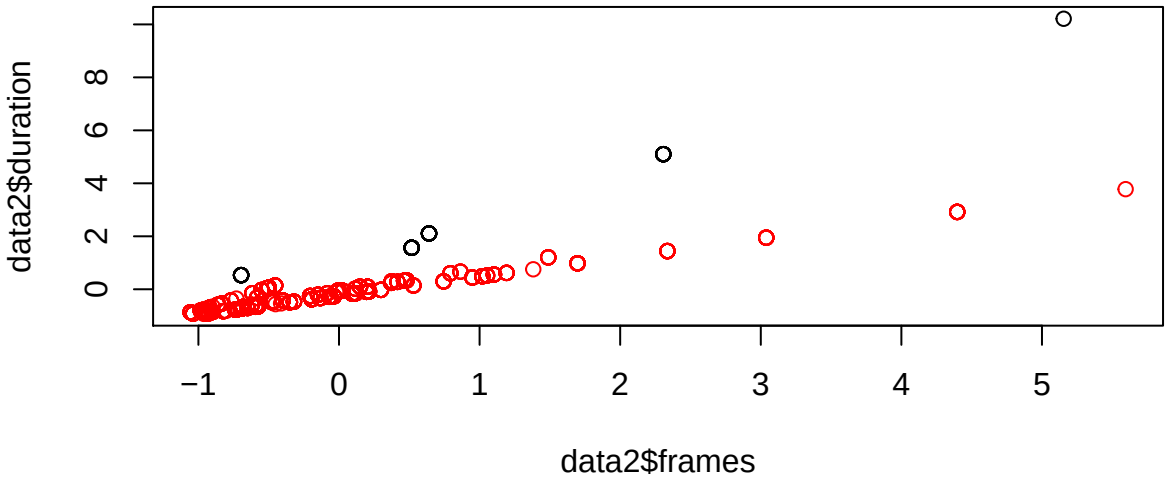


Classes seen to be rather clearly linearly separable (with exception of a few cases near to the origin).

5. Fit a Linear Discriminant Analysis model with "class" as target and "frames" and "duration" as features to the entire dataset (scale features first). Produce the plot showing the classified data and report the training error. Explain why LDA was unable to achieve perfect (or nearly perfect) classification in this case. (2p)

- Plot:

```
library(MASS)
m3=lda(as.factor(class)~frames+duration, data=data2)

plot(data2$frames,data2$duration, col=predict(m3)$class)
```



- Missclassification error:

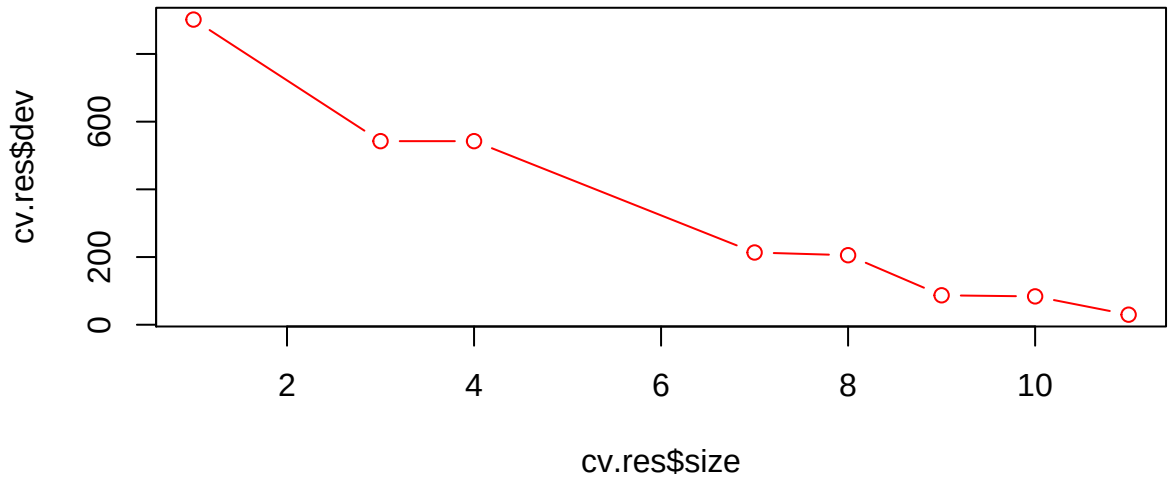
```
missclass=function(X,X1){
  n=length(X)
  return(1-sum(diag(table(X,X1)))/n)
}

missclass(data2$class, predict(m3, type="class")$class)
```

```
## [1] 0.172
```

The result of classification is rather bad. It is clear that covariance matrices per class are very different. In addition, class-conditional distributions do not look like multivariate normal.

6. Fit a decision tree model with "class" as target and "frames" and "duration" as features to the entire dataset, choose an appropriate tree size by cross-validation. Report the training error. How many leaves are there in the final tree? Explain why such a complicated tree is needed to describe such a simple decision boundary. (2p)

```
library(tree)
m4=tree(as.factor(class)~frames+duration, data=data2)
set.seed(12345)
cv.res=cv.tree(m4)
plot(cv.res$size, cv.res$dev, type="b", col="red") # print(m4) // plot(m4)
```



- Missclassification error:

```
missclass(data2$class, predict(m4, type="class"))
```

```
## [1] 0.001
```

According to the cross-validation plot, the optimal tree is the largest one among the ones that were grown with default settings. The optimal tree among these has 11 leaves.

Such a complicated tree is needed because the optimal decision boundary is linear but not parallel to any of the coordinate axes. Accordingly, decision tree would need to make many splits and produce a stair-kind of decision boundary that would approximate this linear decision boundary.

Assignment 2. SVM, Online Learning and Neural Networks.

1. Implement the budget online support vector machine (SVM). Check the course slides for the pseudo-code. Feel free to use the template below. Note that you are not using all the attributes and points in the file. Run your code on the spambase.csv file for the (M, β) values (500,0) and (500,-0.05). Plot the error rate as a function of the number of training point.(3p)

2. Analyze the results obtained. In particular, (1) explain why (500,0) gives better results than (500,-0.05), and (2) explain why the setting (50,0) is the slowest (you do not need to run your code until completion for this setting). (2p)

```
set.seed(1234567890)
spam <- read.csv2("spambase.csv")
ind <- sample(1:nrow(spam))
spam <- spam[ind,c(1:48,58)]
h <- 1
beta <- 0
M <- 50
N <- 500 # number of training points

gaussian_k <- function(x, h) { # It is fine if students use exp(-x**2)/h instead
  return (exp(-(x**2)/(2*h*h)))
}

SVM <- function(sv,i) { #SVM on point i with support vectors sv
  yi <- 0
  for(m in 1:length(sv)) {
    xixm <- rbind(spam[i,-49],spam[sv[m],-49]) # do not use the true label when computing the distance
```

```
    tm <- 2 * spam[sv[m],49] - 1 # because the true labels must be -1/+1 and spambase has 0/1
    yi <- yi + tm * gaussian_k(dist(xixm, method="euclidean"), h)
  }
  return (yi)
}

errors <- 1
errorrate <- vector(length = N)
errorrate[1] <- 1
sv <- c(1)
for(i in 2:N) {
  yi <- SVM(sv,i)
  ti <- 2 * spam[i,49] - 1

  if(ti * yi < 0) {
    errors <- errors + 1
  }
  errorrate[i] <- errors/i

   cat(".") # iteration ", i, "error rate ", errorrate[i], ti * yi, "sv ", length(sv), "\n")
   flush.console()

  if(ti * yi <= beta) {
    sv <- c(sv, i)

    if (length(sv) > M) {
      for(m in 1:length(sv)) { # remove the support vector that gets classified best without itself
        sv2 <- sv[-m]
        ym2 <- SVM(sv2,sv[m])
        tm <- 2 * spam[sv[m],49] - 1

        if(m==1) {
          max <- tm * ym2
          ind <- 1
        }
        else {
          if(tm * ym2 > max) {
            max <- tm * ym2
            ind <- m
          }
        }
      }
      sv <- sv[-ind]

      # cat("removing ", ind, max, "\n")
      # flush.console()
    }
  }
}
```

For the setting (500, 0), the final error rate is 0.212, the SMV has 106 support vectors, and the error rate plot looks like the one below.

```
plot(errorrate[seq(from=1, to=N, by=10)], ylim=c(0.2,0.4), type="o")
M
beta
length(sv)
errorrate[N]
```

For the setting (500, -0.05), the final error rate is 0.248, the SMV has 46 support vectors, and the error rate plot looks like the one below.

The reason why the first setting gives better results is that every misclassified point is added as support vector, whereas in the second setting only seriously misclassified points are added, i.e. only those that are beta units beyond the decision boundary. This can also be seen by looking at the number of support vectors at the end of the two runs. Adding a misclassified point as support vector implies correcting the SVM so as to make that point (and other similar points) less likely to be misclassified in the future.

Finally, the setting (50,0) is the slowest because it is the one that removes more vectors. Removing is an expensive operation as one has to evaluate the result of removing each vector, and there are 50 of them.

SUPPORT VECTOR MACHINES You are asked to use the function ksvm from the R package kernlab to learn a support vector machine (SVM) for classifying the spam dataset that is included with the package. Consider the radial basis function kernel (also known as Gaussian) with a width of 0.05. For the C parameter, consider values 0.5, 1 and 5. This implies that you have to consider three models.

```
library(kernlab)
set.seed(1234567890)

data(spam)
```

1 TDE. Perform model selection, i.e. select the most promising of the three models (use any method of your choice except cross-validation or nested cross-validation). (2p)

```
# Model selection
index <- sample(1:4601)
tr <- spam[index[1:2500], ]
va <- spam[index[2501:3501], ]
te <- spam[index[3502:4601], ]

filter <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(sigma=0.05),C=0.5)
mailtype <- predict(filter,va[,-58])
t <- table(mailtype,va[,58])
(t[1,2]+t[2,1])/sum(t)
```

```
## [1] 0.08791209
```

```
filter <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(sigma=0.05),C=1)
mailtype <- predict(filter,va[,-58])
t <- table(mailtype,va[,58])
(t[1,2]+t[2,1])/sum(t)
```

```
## [1] 0.08091908
```

```
filter <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(sigma=0.05),C=5)
mailtype <- predict(filter,va[,-58])
t <- table(mailtype,va[,58])
(t[1,2]+t[2,1])/sum(t)
```

```
## [1] 0.08291708
```

2 TDE. Estimate the generalization error of the SVM selected above (use any method of your choice except cross-validation or nested cross-validation).(1p)

```
# Error estimation
filter <- ksvm(type~.,data=spam[index[1:3501], ],kernel="rbfdot",kpar=list(sigma=0.05),C=1)
mailtype <- predict(filter,te[,-58])
t <- table(mailtype,te[,58])
(t[1,2]+t[2,1])/sum(t)
```

```
## [1] 0.07818182
```

3 TDE. Produce the SVM that will be returned to the user, i.e. show the code.(1p)

```
# Final model
filter <- ksvm(type~.,data=spam,kernel="rbfdot",kpar=list(sigma=0.05),C=1)
```

4 TDE. What is the purpose of the parameter C ?(1p) The data is split into three folds: Training, validation, and testing. The validation data is used to select the best model (C = 0.05 in this case), whose generalization error is estimated with the help of the test data (note that we use all the data for learning except the test data). The model returned to the user is the result of using all the data for learning and C = 0.05. The estimated error of the model returned to the user is precisely the error we computed on the test data.

NEURAL NETWORKS 3. Train a neural network (NN) to learn the trigonometric sine function. To do so, sample 50 points uniformly at random in the interval [0, 10]. Apply the sine function to each point. The resulting pairs are the data available to you. Use 25 of the 50 points for training and the rest for validation. The validation set is used for early stop of the gradient descent. Consider threshold values i/1000 with i = 1,. . . , 10. Initialize the weights of the neural network to random values in the interval [-1, 1]. Consider two NN architectures: A single hidden layer of 10 units, and two hidden layers with 3 units each. Choose the most appropriate NN architecture and threshold value. Motivate your choice. Feel free to reuse the code of the corresponding lab.(3p)

```
library(neuralnet)

# two layers
set.seed(1234567890)

Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin=sin(Var))
tr <- trva[1:25,] # Training
va <- trva[26:50,] # Validation
restr <- vector(length = 10)
resva <- vector(length = 10)
winit <- runif(22, -1, 1) # Random initializaiton of the weights in the interval [-1, 1]
for(i in 1:10) {
  nn <- neuralnet(formula = Sin ~ Var, data = tr, hidden = c(3,3), startweights = winit,
                  threshold = i/1000, lifesign = "full")

  # nn$result.matrix
  aux <- compute(nn, tr[,1])$net.result # Compute predictions for the trainig set and their squared err
  restr[i] <- sum((tr[,2] - aux)**2)/2

  aux <- compute(nn, va[,1])$net.result # The same for the validation set
  resva[i] <- sum((va[,2] - aux)**2)/2
}
plot(restr, type = "o")
```

```r
plot(resva, type = "o")
restr
resva

# one layer
set.seed(1234567890)

Var <- runif(50, 0, 10)
trva <- data.frame(Var, Sin=sin(Var))
tr <- trva[1:25,] # Training
va <- trva[26:50,] # Validation

# plot(trva)
# plot(tr)
# plot(va)

restr <- vector(length = 10)
resva <- vector(length = 10)
winit <- runif(41, -1, 1) # Random initializaiton of the weights in the interval [-1, 1]
for(i in 1:10) {
  nn <- neuralnet(formula = Sin ~ Var, data = tr, hidden = c(10), startweights = winit,
                  threshold = i/1000, lifesign = "full")

  # nn$result.matrix

  aux <- compute(nn, tr[,1])$net.result # Compute predictions for the trainig set and their squared err
  restr[i] <- sum((tr[,2] - aux)**2)/2

  aux <- compute(nn, va[,1])$net.result # The same for the validation set
  resva[i] <- sum((va[,2] - aux)**2)/2
}
plot(restr, type = "o")
plot(resva, type = "o")
restr
resva
```

4. Estimate the generalization error of the NN selected above (use any method of your choice).(1p)

```r
# estimate generalization error for the best run above (one layer with threshold 4/1000)
Var <- runif(50, 0, 10)
te <- data.frame(Var, Sin=sin(Var))

winit <- runif(31, -1, 1)
nn <- neuralnet(formula = Sin ~ Var, data = trva, hidden = 10, startweights = winit,
                    threshold = 4/1000, lifesign = "full")
sum((te[,2] - compute(nn, te[,1])$net.result)**2)/2
```

5. In the light of the results above, would you say that the more layers the better ? Motivate your answer.(1p) The best model is that with one layer of 10 neurons and threshold for early stopping equal to 4/1000. The generalization error is estimated by sampling additional test data, since we have access to the true function. Therefore, we conclude that more layers is not always better.

THEORY

THEORY INDEX

BLOC 1

1. Basic concepts in ML. Regression, regularization and model selection.

Simple linear regression:

Model:

$$y \sim N(w_0 + w_1 x, \sigma^2)$$

or

$$y = w_0 + w_1 x + \epsilon,$$
$$\epsilon \sim N(0, \sigma^2)$$

or

$$p(y|x, w) = N(w_0 + w_1 x, \sigma^2)$$

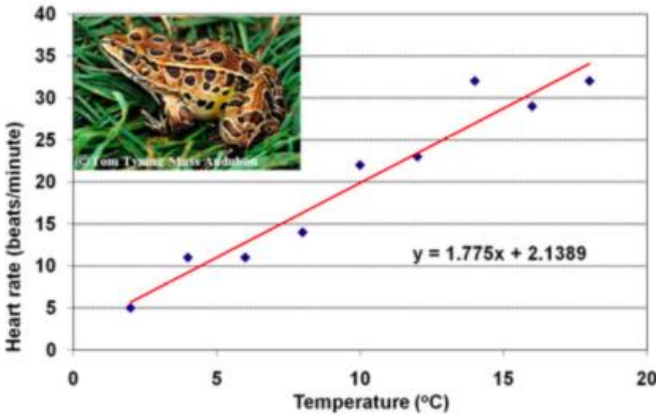
Terminology:

w_0: **intercept (or bias)**

w_1: **regression coefficient**

Response

The target responds directly and linearly to changes in the feature



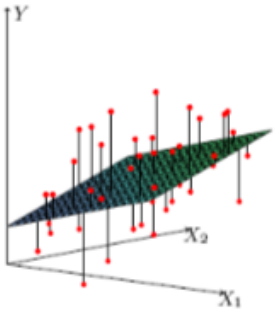
Model:

$$y \sim N(\boldsymbol{w}^T \boldsymbol{x}, \sigma^2)$$

where

$$\boldsymbol{w} = \{w_0, \ldots w_d\}$$
$$\boldsymbol{x} = \{1, x_1, \ldots x_d\}$$

Why is "1" here?



The response variable responds directly and linearly to changes in each of the inputs

- Least squares estimates of the parameters

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

- Predicted values $\hat{y} = X\hat{w} = X(X^T X)^{-1} X^T y = Py$

Hat matrix

- Linear regression belongs to the class of **linear smoothers**

Why is it called so?

Linear regression (OLS) in R:

- fit=lm(formula, data, subset, weights,…)
  - **data** is the data frame containing the predictors and response values
  - **formula** is expression for the model
  - **subset** which observations to use (training data)?
  - **weights** should weights be used?

fit is object of class **lm** containing various regression results.

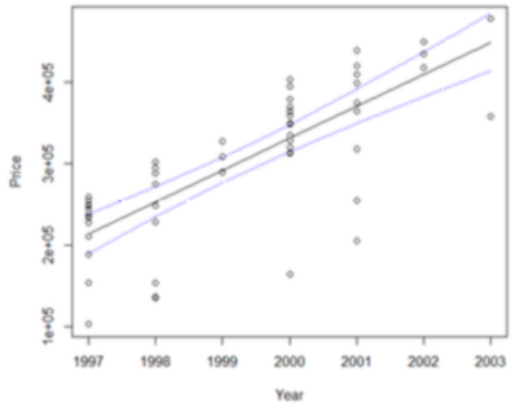
- Useful functions (many are generic, used in many other models)
  - Get details about the particular function by ".", for ex. predict.lm

```
summary(fit)
predict(fit, newdata, se.fit, interval)
coefficients(fit) # model coefficients
confint(fit, level=0.95) # CIs for model parameters
fitted(fit) # predicted values
residuals(fit) # residuals
```

```
mydata=read.csv2("Bilexempel.csv")
fit1=lm(Price~Year, data=mydata)
summary(fit1)
fit2=lm(Price~Year+Mileage+Equipment,
data=mydata)
summary(fit2)
```

- Prediction

```
fitted <- predict(fit1, interval =
"confidence")

# plot the data and the fitted line
attach(mydata)
plot(Year, Price)
lines(Year, fitted[, "fit"])

# plot the confidence bands
lines(Year, fitted[, "lwr"], lty = "dotted",
col="blue")
lines(Year, fitted[, "upr"], lty = "dotted",
col="blue")
detach(mydata)
```



Ridge Regression:

- **Idea**: Keep all predictors but shrink coefficients to make model less complex

minimize $-loglikelihood + \lambda_0 \|w\|_2^2$

→ **l_2 regularization**

  - Given that model is Gaussian, we get **Ridge regression:**

$$\hat{w}^{ridge} = \text{argmin}\left\{\sum_{i=1}^{N}(y_i - w_0 - w_1 x_{1j} - \ldots - w_p x_{pj})^2 + \lambda \sum_{j=1}^{p} w_j^2\right\}$$

- $\lambda > 0$ is **penalty factor**

$$\hat{\boldsymbol{w}}^{\boldsymbol{ridge}} = \left(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X} + \lambda \boldsymbol{I}\right)^{-1}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\hat{\boldsymbol{w}} = \boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{P}\boldsymbol{y}$$

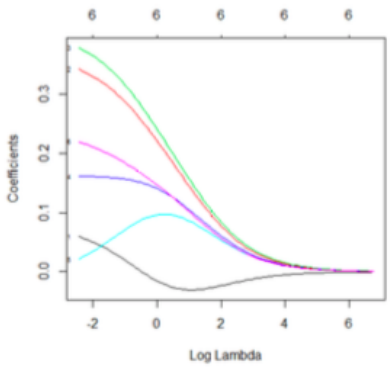
Hat matrix

How do we compute degrees of freedom here?

Ridge Regression in R:

- R code: use package **glmnet** with alpha=0 (Ridge regression)
- Seeing how Ridge converges

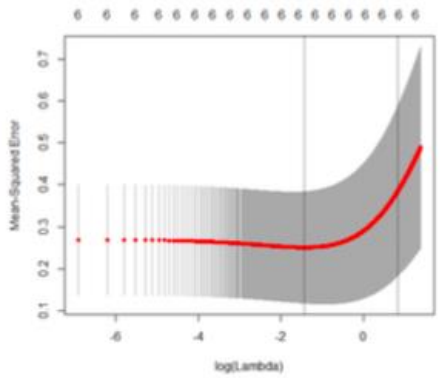
```
data=read.csv("machine.csv", header=F)
covariates=scale(data[,3:8])
response=scale(data[, 9])

model0=glmnet(as.matrix(covariates),
response, alpha=0,family="gaussian")
plot(model0, xvar="lambda", label=TRUE)
```



- Choosing the best model by cross-validation:

```
model=cv.glmnet(as.matrix(covariates),
response, alpha=0,family="gaussian")
model$lambda.min
plot(model)
coef(model, s="lambda.min")
```

```
> coef(model, s="lambda.min")
7 x 1 sparse Matrix of class "dgC
                      1
(Intercept) -4.530442e-17
V3           3.420739e-02
V4           3.085696e-01
V5           3.403839e-01
V6           1.593470e-01
V7           5.489116e-02
V8           1.970982e-01
```



> model$lambda.min

- How good is this model in prediction?

```
ind=sample(209, floor(209*0.5))
data1=scale(data[,3:9])
train=data1[ind,]
test=data1[-ind,]

covariates=train[,1:6]
response=train[, 7]
model=cv.glmnet(as.matrix(covariates), response, alpha=1,family="gaussian",
lambda=seq(0,1,0.001))
y=test[,7]
ynew=predict(model, newx=as.matrix(test[, 1:6]), type="response")

#Coefficient of determination
sum((ynew-mean(y))^2)/sum((y-mean(y))^2)

sum((ynew-y)^2)
```

Note that data are so small so numbers change much for other train/test

```
> sum((ynew-mean(y))^2)/sum((y-mean(y))^2)
[1] 0.5438148
> sum((ynew-y)^2)
[1] 18.04988
>
```

Lasso:

- **Idea**: Similar idea to Ridge
- Minimize minus loglikelihood plus **linear** penalty factor→ $\mathbf{l_1}$ **regularization**
  - Given that model is Gaussian, we get **LASSO** (least absolute shrinkage and selection operator):

$$\hat{w}^{lasso} = \text{argmin}\left\{\sum_{i=1}^{N}(y_i - w_0 - w_1 x_{1j} - \ldots - w_p x_{pj})^2 + \lambda\sum_{j=1}^{p}|w_i|\right\}$$

- $\lambda > 0$ is **penalty factor**

LASSO yields sparse solutions!

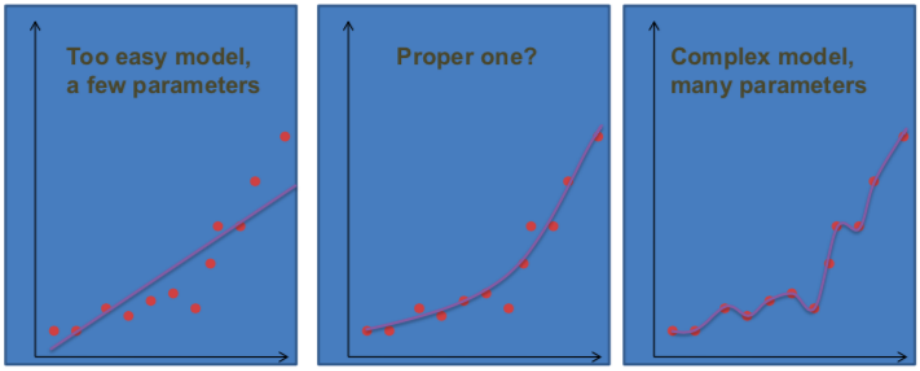
- **Lasso is widely used when $p \gg n$**
  - Linear regression breaks down when $p > n$

Lasso Regression in R: use glmnet() with alpha=1

Overfitting anf fitting a model:

- Observed: Maximum likelihood can lead to overfitting.

- Complex model can overfit your data



- Solutions
  - Selecting proper parameter values
    - Regularized risk minimization
  - Selecting proper model type, for ex. number of parameters
    - Houldout method
    - Cross-validation

Loss Functions:

- Given a model, choose the optimal parameter values
  - Decision theory
- Define loss $L(Y, \hat{y})$
  - How much we loose in guessing true Y incorrectly
- If we know the true distribution $p(y, x|w)$ then we choose $\hat{y}$

$$\min_{\hat{y}} EL(y, \hat{y}) = \min_{\hat{y}} \int L(y, \hat{y}) p(y, x|w) dx dy$$

- How to define loss function?
  - No unique choice, often defined by application
  - **Normal practice: Choose the loss related to minus loglikelihood**

Example: Predicting the amount of the product at the storage:

$$L(Y, \hat{y}) = \begin{cases} 10 - \frac{\hat{y}}{Y}, \hat{y} \leq Y \\ 1000, \hat{y} > Y \end{cases}$$

Example: Compute loss function related to

- Normal distribution

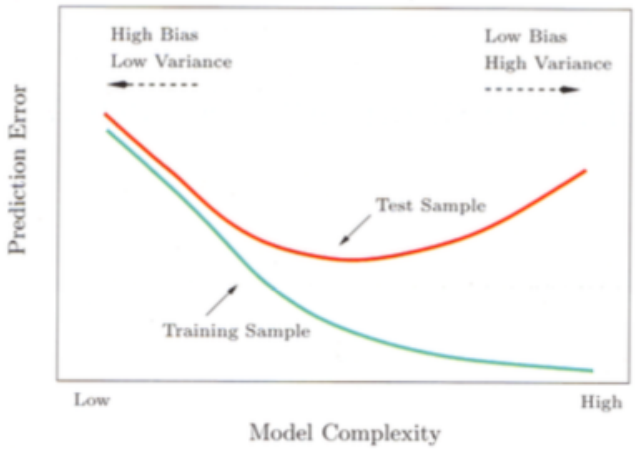
Guess why such loss function was chosen

- Classification problems
  - Common loss function $L(\mathrm{Y}, \hat{y}) = \begin{cases} 0, Y = \hat{y} \\ 1, Y \neq \hat{y} \end{cases}$
  - When minimizing the loss, equivalent to misclassification rate

Bias-variance trade-off:

- Bias of an estimator $Bias(\hat{y}\ (x_0)) = E[\hat{y}(x_0) - f(x_0)]$, $f(x_0)$ is expected response
  - If $Bias(\hat{y}(x_0)) = 0$, the estimator is **unbiased**
  - ML estimators are asymptotically unbiased if the model is enough complex
  - However, unbiasedness does not mean a good choice!
- Assume loss is $L(Y, \hat{y}) = (Y - \hat{y})^2$

$$R(Y(x_0), \hat{y}(x_0)) = \sigma^2 + Bias^2(\hat{y}(x_0)) + Var(\hat{y}(x_0))$$



When loss is not quadratic, no such nice formula exist

Cross-vaidation:

- Compared to holdout method:
  - Why do we use only some portion of data for training- can we use more (increase accuracy)?

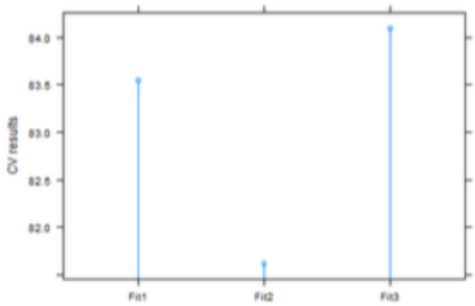
Cross-validation (Estimates Err)

K-fold cross-validation (rough scheme, show picture):

1. Permute the observations randomly
2. Divide data-set in K roughly equally-sized subsets
3. Remove subset #i and fit the model using remaining data.
4. Predict the function values for subset #i using the fitted model.
5. Repeat steps 3-4 for different i
6. CV= squared difference between observed values and predicted values (another function is possible)

- Try models with different predictor sets

```
data=read.csv("machine.csv", header=F)
library(cvTools)


fit1=lm(V9~V3+V4+V5+V6+V7+V8, data=data)
fit2=lm(V9~V3+V4+V5+V6+V7, data=data)
fit3=lm(V9~V3+V4+V5+V6, data=data)
f1=cvFit(fit1, y=data$V9, data=data,K=10,
foldType="consecutive")
f2=cvFit(fit2, y=data$V9, data=data,K=10,
foldType="consecutive")
f3=cvFit(fit3, y=data$V9, data=data,K=10,
foldType="consecutive")
res=cvSelect(f1,f2,f3)
plot(res)
```



- Holdout is easy to do (a few model fits to each data)
- Cross validation is computationally demanding (many model fits)
- Holdout is applicable for large data
  - Otherwise, model selection performs poorly
- Cross validation is more suitable for smaller data

2. Classification methods. Dimensionality reduction and uncertainty estimation.

Linear Classificaiton methods:

- **Deterministic**: decide a rule that directly maps X into $\hat{Y}$
- **Probabilistic:** define a model for $P(Y = C_i|X), i = 1 \dots K$

Disanvantages of deterministic classifiers:

- Sometimes simple mapping is not enough (risk of cancer)
- Difficult to embed loss-> rerun of optimizer is often needed
- Combining several classifiers into one is more problematic
  - Algorithm A classifies as spam, Algorithm B classifies as not spam → ???
  - P(Spam|A)=0.99, P(Spam|B)=0.45 → better decision can be made

Loss matrix:

- Costs of classifying $Y = C_k$ to C_j:
  - Rows: true, columns: predicted

$$L = \|L_{ij}\|, i = 1, \dots, n, j = 1, \dots, n$$

- Example 1: 0/1-loss

$$L = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- Example 2: Spam

$$L = \begin{pmatrix} 0 & 100 \\ 1 & 0 \end{pmatrix}$$

- How to minimize *EL with two classes?*

- Rule:
  - $L_{12}p(x, C_1) > L_{21}p(x, C_2)$ →predict y as C_1

- 0/1 Loss: classify to the class which is more probable!

$$\frac{p(C_1|x)}{p(C_2|x)} > \frac{L_{21}}{L_{12}} \rightarrow predict\ y\ as\ C_1$$

ROC curves:

- Binary classification
- The choice of the thershold $\hat{x} = \frac{L_{21}}{L_{12}}$ affects prediction→ what if we don't know the loss? Which classifier is better?
- **Confusion matrix**

		PREDICTED		
TRUE		1	0	Total
	1	TP	FN	N_+
	0	FP	TN	N_-

- True Positive Rates (TPR) = sensitivity = recall
  - Probability of detection of positives: TPR=1 positives are correctly detected

$$TPR = TP/N_+$$

- False Positive Rates (FPR)
  - Probability of false alarm: system alarms (1) when nothing happens (true=0)

$$FPR = FP/N_-$$

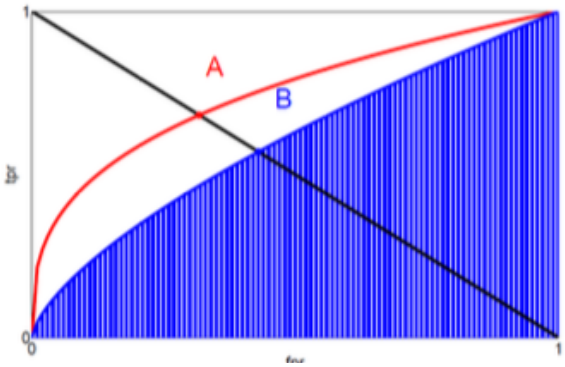
- Specificity

$$Specificity = 1 - FPR$$

- Precision

$$Precision = \frac{TP}{TP + FP}$$

- **ROC**=Receiver operating characteristics
- Use various thresholds, measure TPR and FPR
- Same FPR, higher TPR→ better classifier
- Best classifier = greatest Area Under Curve (**AUC**)



Linear discriminant analysis (LDA):

- Difference LDA vs logistic regression??
  - Coefficients will be estimated differently! (models are different)
- How to estimate coefficients
  - find MLE.

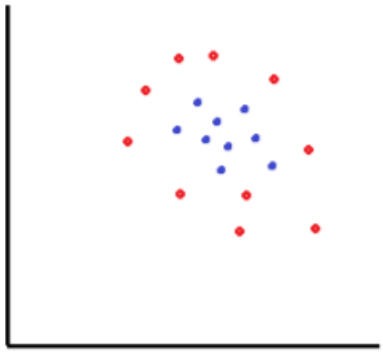
$$\hat{\boldsymbol{\mu}}_c = \frac{1}{N_c} \sum_{i:y_i=c} \mathbf{x}_i, \quad \hat{\boldsymbol{\Sigma}}_c = \frac{1}{N_c} \sum_{i:y_i=c} (\mathbf{x}_i - \hat{\boldsymbol{\mu}}_c)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_c)^T$$

$$\hat{\Sigma} = \frac{1}{N} \sum_{c=1}^{k} N_c \, \hat{\Sigma}_c$$

  - Sample mean and sample covariance are MLE!
  - If class priors are parameters (**proportional priors**),

$$\hat{\pi}_c = \frac{N_c}{N}$$

- Generative classifiers are easier to fit, discriminative involve numeric optimization
- LDA and Logistic have same model form but are fit differently
- LDA has stronger assumptions than Logistic, some other generative classifiers lead also to logistic expression
- New class in the data?
  - Logistic: fit model again
  - LDA: estimate new parameters from the new data
- Logistic and LDA: complex data fits badly unless interactions are included



- LDA (and other generative classifiers) handle missing data easier

- Standardization and generated inputs:
  - Not a problem for Logistic
  - May affect the performance of the LDA in a complex way

- Outliers affect $\Sigma \rightarrow$ LDA is not robust to gross outliers

- LDA is often a good classification method even if the assumption of normality and common covariance matrix are not satisfied.

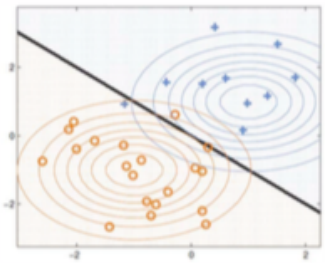
Naïve Bayes Classifiers (Discrete):

- Given $D = \{(X_{m1}, \dots X_{mp}, Y_m),\ m = 1, \dots n\}$
- Assume $X_i \in \{x_1, \dots x_J\}, i = 1, \dots p, Y \in \{y_1 \dots y_K\}$
- Denote $\theta_{ijk} = p(X_i = x_j | Y = y_k)$
  - How many parameters? $(J-1)Kp$
- Denote $\pi_k = p(Y = y_k)$

- Maximum likelihood: assume θ_{ijk} and π_k are constants
  - $\hat{\theta}_{ijk} = \frac{\#\{X_i = x_j \,\&\, Y = y_k\}}{\#\{Y = y_k\}}$
  - $\hat{\pi}_k = \frac{\#\{Y = y_k\}}{n}$
  - Classification using 0-1 loss: $\hat{Y} = \arg\max_y p(Y = y|X)$

Naïve Bayes Classifiers (Continuous):

- X_i are continuous
- Assumption A: $x_j|y = C$ are univariate Gaussian
  - $p(x_j|y = C_i, \theta) = N(x_j|\mu_{ij}, \sigma_{ij}^2)$
- Therefore $p(\boldsymbol{x}|y = C_i, \theta) = N(\boldsymbol{x}|\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i})$
  - $\boldsymbol{\Sigma_i}$=diag$(\sigma_{i1}^2, \dots, \sigma_{ip}^2)$



- Naive bayes is a special case of LDA (given A)
  - $\rightarrow$MLE are means and variances (per class)

- Assumption B: $p(x_j|y = C)$ are unknown functions of x_j that can be estimated from data
  - Nonparametric density estimation (kernel for ex.)

1. Estimate $p(X_i = x_j|Y = y_k)$ using nonparametric methods
2. Estimate $p(Y = y_k)$ as class proportions
3. Use Bayes rule and 0-1 loss to classify


```
library(MASS)
library(e1071)
n=dim(housing)[1]
ind=rep(1:n, housing[,5])
housing1=housing[ind,-5]

fit=naiveBayes(Sat~., data=housing1)
fit

Yfit=predict(fit, newdata=housing1)
table(Yfit,housing1$Sat)
```

Decision Trees (Classification tree, target is qualitative model):

- Classification probability $p_{mk} = p(Y = k|X \in R_m)$ is estimated for every class in a node
- How to estimate p_{mk} for class k and node R_m?

Class proportions

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

- For any node (leave), a label can be assigned

$$k(m) = \arg\max_k \hat{p}_{mk}$$

- Impurity measure $Q(R_m)$
  - R_m is a tree node (region)
  - Node can be split unless it is pure

Misclassification error:	$\frac{1}{N_m} \sum_{i \in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}$
Gini index:	$\sum_{k \neq k'} \hat{p}_{mk} \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$
Cross-entropy or deviance:	$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$.

- Note: In many sources, deviance is $Q(R_m)\, N(R_m)$

Example: Cross –entropy is MLE of $Y_j|T \sim Multinomial(p_{j1}, \ldots p_{jc})$

Decision Trees (Regression tree, target is contnuous model):

Step 1: Finding optimal tree: grow the tree in order to minimize global objective

1. Let C_0 be a hypercube containing all observations
2. Let queue C={C_0}
3. Pick up some C_i from C and find a variable X_j and value *s* that split C_j into two hypercubes

$$R_1(j,s) = \{X|X_j \leq s\} \text{ and } R_2(j,s) = \{X|X_j > s\}$$

and solve

$$\min_{j,s}[N_1 Q(R_1) + N_2 Q(R_2)]$$

4. Remove C_j from C and add R_1 and R_2
5. Repeat 3-4 as many times as needed (or until each cube has only 1 observation)
   - The largest tree will interpolate the data → large trees = overfitting the data
   - Too small trees=underfitting (important structure may not be captured)

Decision Trees in R:

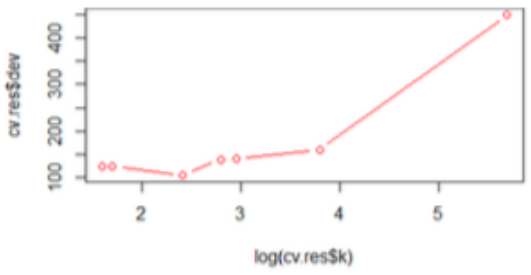
```
library(tree)
n=dim(biopsy)[1]
fit=tree(class~., data=biopsy)
plot(fit)
text(fit, pretty=0)
fit
summary(fit)
```

Misclassification results

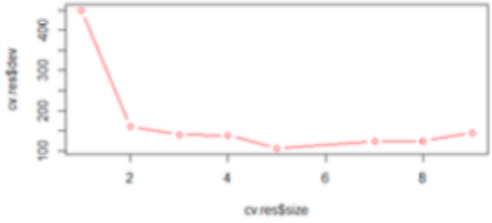
```
Yfit=predict(fit, newdata=biopsy, type="class")
table(biopsy$class,Yfit)
```

- Selecting optimal tree by penalizing
  - Cv.tree()

```
set.seed(12345)
ind=sample(1:n, floor(0.5*n))
train=biopsy[ind,]
valid=biopsy[-ind,]

fit=tree(class~., data=train)
set.seed(12345)
cv.res=cv.tree(fit)
plot(cv.res$size, cv.res$dev, type="b",
col="red")
plot(log(cv.res$k), cv.res$dev,
type="b", col="red")
```

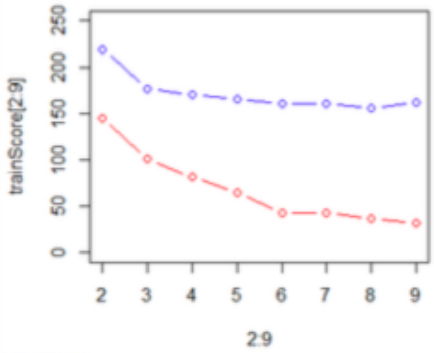


What is optimal number of leaves?



- Selecting optimal tree by train/validation

```
fit=tree(class~., data=train)

trainScore=rep(0,9)
testScore=rep(0,9)

for(i in 2:9) {
  prunedTree=prune.tree(fit,best=i)
  pred=predict(prunedTree, newdata=valid,
type="tree")
  trainScore[i]=deviance(prunedTree)
  testScore[i]=deviance(pred)
}
plot(2:9, trainScore[2:9], type="b", col="red",
ylim=c(0,250))
points(2:9, testScore[2:9], type="b", col="blue")
```



- Final tree: 5 leaves

```
finalTree=prune.tree(fit, best=5)
Yfit=predict(finalTree, newdata=valid,
type="class")
table(valid$class,Yfit)
```

Nonparametric bootstrap:

Given estimator $\widehat{w} = \hat{f}(D)$

Assume $X \sim F(X, w)$, F and w are unknown

1. Estimate $\widehat{w}$ from data **D**=(X_1,…X_n)
2. Generate $\mathbf{D_1}$ =(X^*_1,…X^*_n) by sampling with replacement
3. Repeat step 2 B times
4. The distribution of w is given by $\hat{f}(D_1), \ldots \hat{f}(D_B)$

Nonparametric bootstrap can be applied to any deterministic estimator, distribution-free

Nonparametric bootstrap:

- Write a function *statistic* that depends on *dataframe* and *index* and returns the estimator

```
library(boot)
data2=data[order(data$Area),]#reordering data according to Area

# computing bootstrap samples
f=function(data, ind){
  data1=data[ind,]# extract bootstrap sample
  res=lm(Price~Area, data=data1) #fit linear model
  #predict values for all Area values from the original data
  priceP=predict(res,newdata=data2)
  return(priceP)
}
res=boot(data2, f, R=1000) #make bootstrap
```

Parametric bootstrap:

Given estimator $\hat{w} = \hat{f}(D)$

Assume $X \sim F(X, w)$, F is known and w is unknown

1. Estimate $\hat{w}$ from data **D**=(X_1,...X_n)
2. Generate **D_1** =(X^*_1,...X^*_n) by generating from $F(X, \hat{w})$
3. Repeat step 2 B times
4. The distribution of w is given by $\hat{f}(D_1), \ldots \hat{f}(D_B)$

Parametric bootstrap is **more** precise if the distribution form is correct

Parametric bootstrap:

- Compute value *mle* that estimates model parameters from the data
- Write function *ran.gen* that depends on *data* and *mle* and which generates new data
- Write function *statistic* that depend on *data* which will be generated by *ran.gen* and should return the estimator

```
mle=lm(Price~Area, data=data2)

rng=function(data, mle) {
  data1=data.frame(Price=data$Price, Area=data$Area)
  n=length(data$Price)
#generate new Price
  data1$Price=rnorm(n,predict(mle, newdata=data1),sd(mle$residuals))
  return(data1)
}

f1=function(data1){
  res=lm(Price~Area, data=data1) #fit linear model
  #predict values for all Area values from the original data
  priceP=predict(res,newdata=data2)
  return(priceP)
}

res=boot(data2, statistic=f1, R=1000, mle=mle,ran.gen=rng, sim="parametric")
```

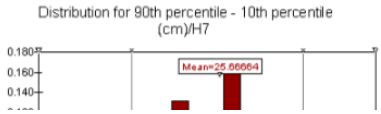
Confidence Intervals:

- To estimate 100(1-α) confidence interval for w

Bootstrap percentile method

1. Using bootstrap, compute $\hat{f}(D_1), \ldots \hat{f}(D_B)$, sort in ascending order, get $w_1 \ldots w_B$
2. Define A_1=ceil(B α/2), A_2=floor(B-B α/2)
3. Confidence interval is given by

$$\left(w_{A_1}, w_{A_2}\right)$$



Confidence Intervals in regression:

Estimation

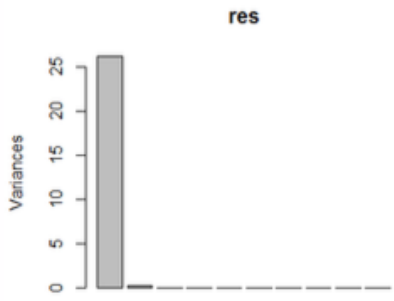
1. Compute $D_1, \dots D_B$ using a bootstrap
2. Fit model to $D_1, \dots D_B$ →estimate $\widehat{w}_1, \dots \widehat{w}_B$
3. For a given X, compute $f(X, \widehat{w}_1), \dots f(X, \widehat{w}_B)$ and estimate confidence interval by (percentile method)
4. Combine confidence intervals in a band

```
e=envelope(res) #compute confidence bands

fit=lm(Price~Area, data=data2)
priceP=predict(fit)

plot(Area, Price, pch=21, bg="orange")
points(data2$Area,priceP,type="l") #plot fitted line

#plot cofidence bands
points(data2$Area,e$point[2,], type="l", col="blue")
points(data2$Area,e$point[1,], type="l", col="blue")
```

Principal Components Analysis:

- *PCA* is a technique for reducing the complexity of high dimensional data
- It can be used to approximate high dimensional data with a few dimensions (latent features) –> much less data to store
- New variables might have a special interpretation
- The first principal component (PC1) is the direction that maximizes the variance of the projected data
- The second principal component (PC2) is the direction that maximizes the variance of the projected data after the variation along PC1 has been removed
- Prcomp(), biplot(), screeplot()

```
mydata=read.csv2("tecator.csv")
data1=mydata
data1$Fat=c()
res=prcomp(data1)
lambda=res$sdev^2
#eigenvalues
lambda
#proportion of variation
sprintf("%2.3f",lambda/sum(lambda)*100)
screeplot(res)
```



- Principal component **loadings (U)**

```
U=res$rotation
head(U)
```

- Trace plots

```
U= res$rotation
plot(U[,1], main="Traceplot, PC1")
plot(U[,2],main="Traceplot, PC2")
```

Probabilistic PCA:

- z_i-latent variables, x_i- observed variables

$$\boldsymbol{z} \sim \boldsymbol{N}(0, \boldsymbol{I})$$
$$\boldsymbol{x}|\boldsymbol{z} \sim \boldsymbol{N}(\boldsymbol{x}|\boldsymbol{W}\boldsymbol{z} + \boldsymbol{\mu}, \sigma^2\boldsymbol{I})$$

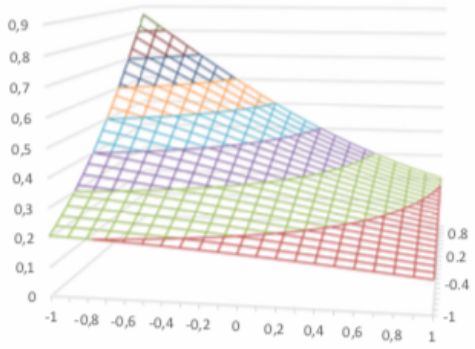
- Alternatively

$$\boldsymbol{z} \sim \boldsymbol{N}(0, \boldsymbol{I}), \boldsymbol{x} = \boldsymbol{\mu} + \boldsymbol{W}\boldsymbol{z} + \boldsymbol{\epsilon}, \boldsymbol{\epsilon} \sim \boldsymbol{N}(\boldsymbol{0}, \boldsymbol{\sigma^2}\boldsymbol{I})$$

- **Interpretation**: Observed data (X) is obtained by rotation, scaling and translation of standard normal distribution (Z) and adding some noise.

Independent Component Analysis (ICA):

- Probabilistic PCA does not capture latent factors
  - Rotation invariance
- Let's choose distribution which is not rotation invariant→will get unique latent factors
- Choose non-Gaussian $p(z_i)$
- Assuming latent features are **independent**



$$p(z) = \prod_{i=1}^{M} p(z_i) \qquad p(z_i) = \frac{2}{\pi(e^{z_i} + e^{-z_i})}$$

- Model

$$x = \mu + Wz + \epsilon, \qquad \epsilon \sim \boldsymbol{N}(0, \Sigma)$$

- **Estimation : Maximum likelihood** $(V = W^{-1})$
  - Assuming noise-free x

$$\max_V \sum_{i=1}^{n} \sum_{j=1}^{p} \log\left(p_j\left(v_j^T x_i\right)\right)$$

Subject to $||v_i|| = 1$

R package: fastICA

```
S <- cbind(sin((1:1000)/20), rep((((1:200)-100)/100), 5))
A <- matrix(c(0.291, 0.6557, -0.5439, 0.5572), 2, 2)
X <- S %*% A #mixing signals
a <- fastICA(X, 2) #now separate them
```

3. Kernel methods and support vector machines. Neural networks and deep learning.

Kernel Classification

- The moving window rule gives equal weight to all the points in the ball, which may be counterintuitive. Then,

$$y_k(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \sum_n \mathbf{1}_{\{t_n=1\}} k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) \leq \sum_n \mathbf{1}_{\{t_n=0\}} k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) \\ 1 & \text{otherwise} \end{cases}$$

where $k : \mathbb{R}^D \to \mathbb{R}$ is a kernel function, which is usually non-negative and monotone decreasing along rays starting from the origin. The parameter h is called smoothing factor or width.

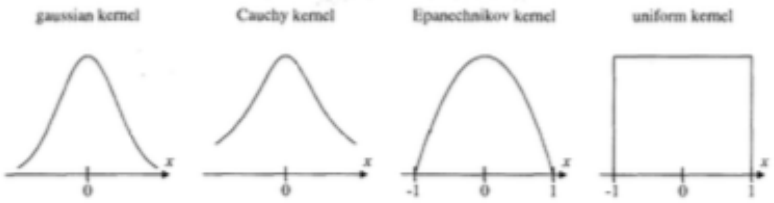


FIGURE 10.3. *Various kernels on $\mathcal{R}$.*

- Gaussian kernel: $k(u) = exp(-||u||^2)$ where $||\cdot||$ is the Euclidean norm.
- Cauchy kernel: $k(u) = 1/(1 + ||u||^{D+1})$
- Epanechnikov kernel: $k(u) = (1 - ||u||^2)\mathbf{1}_{\{||u||\leq 1\}}$
- Moving window kernel: $k(u) = \mathbf{1}_{\{u \in S(0,1)\}}$

- Consider regressing an unidimensional continuous random variable on a D-dimensional continuous random variable.
- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, t_n)\}$ available, then we average over the points in $C(\boldsymbol{x}, h)$ or $S(\boldsymbol{x}, h)$, or kernel-weighted average over all the points.
- In other words,

$$y_C(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{x}_n \in C(\boldsymbol{x},h)} t_n}{|\{\boldsymbol{x}_n \in C(\boldsymbol{x}, h)\}|}$$

or

$$y_S(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{x}_n \in S(\boldsymbol{x},h)} t_n}{|\{\boldsymbol{x}_n \in S(\boldsymbol{x}, h)\}|}$$

or

$$y_k(\boldsymbol{x}) = \frac{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) t_n}{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right)}$$

- Consider density estimation for a D-dimensional continuous random variable.
- Let $R \subseteq \mathbb{R}^D$ and $\boldsymbol{x} \in R$. Then,

$$P = \int_R p(\boldsymbol{x}) d\boldsymbol{x} \simeq p(\boldsymbol{x})\, Volume(R)$$

and the number of the N training points $\{\boldsymbol{x}_n\}$ that fall inside R is

$$|\{\boldsymbol{x}_n \in R\}| \simeq P\, N$$

and thus

$$p(\boldsymbol{x}) \simeq \frac{|\{\boldsymbol{x}_n \in R\}|}{N\, Volume(R)}$$

- Then,

$$p_C(\boldsymbol{x}) = \frac{|\{\boldsymbol{x}_n \in C(\boldsymbol{x}, h)\}|}{N\, Volume(C(\boldsymbol{x}, h))}$$

or

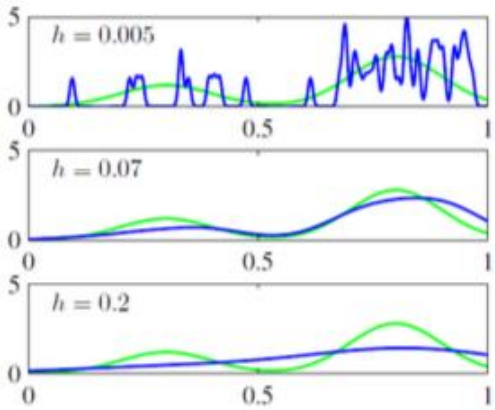
$$p_S(\boldsymbol{x}) = \frac{|\{\boldsymbol{x}_n \in S(\boldsymbol{x}, h)\}|}{N\, Volume(S(\boldsymbol{x}, h))}$$

or

$$p_k(\boldsymbol{x}) = \frac{1}{N} \sum_n k\left(\frac{\boldsymbol{x} - \boldsymbol{x}_n}{h}\right)$$

assuming that $k(u) \geq 0$ for all u and $\int k(u) du = 1$.

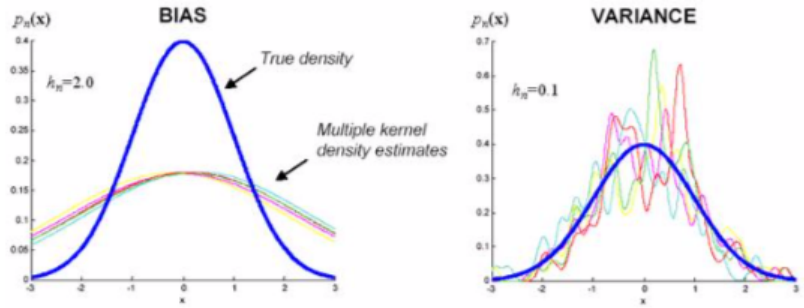
Figure 2.25 Illustration of the kernel density model (2.250) applied to the same data set used to demonstrate the histogram approach in Figure 2.24. We see that h acts as a smoothing parameter and that if it is set too small (top panel), the result is a very noisy density model, whereas if it is set too large (bottom panel), then the bimodal nature of the underlying distribution from which the data is generated (shown by the green curve) is washed out. The best density model is obtained for some intermediate value of h (middle panel).



- From kernel density estimation to kernel classification:
  1. Estimate $p(\boldsymbol{x}|y = 0)$ and $p(\boldsymbol{x}|y = 1)$ using the methods just seen.
  2. Estimate $p(y)$ as class proportions.
  3. Compute $p(y|\boldsymbol{x}) \propto p(\boldsymbol{x}|y)p(y)$ by Bayes theorem.

Kernel Selection

- How to choose the right kernel and width ? E.g., by cross-validation.
- What does "right" mean ? E.g., minimize loss function.
- Note that the width of the kernel corresponds to a bias-variance trade-off.



- Small width implies considering few points. So, the variance will be large (similar to the variance of a single point). The bias will be small since the points considered are close to $\boldsymbol{x}$.
- Large width implies considering many points. So, the variance will be small and the bias will be large.

Kernel Selection

- Recall the following from previous lectures.
- Cross-validation is a technique to estimate the prediction error of a model.

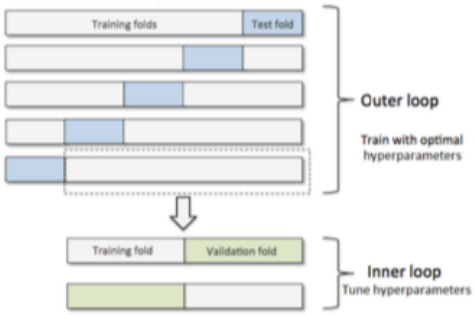
- If the training set contains N points, note that cross-validation estimates the prediction error when the model is trained on $N - N/K$ points.
- Note that the model returned is trained on N points. So, cross-validation overestimates the prediction error of the model returned.
- This seems to suggest that a large K should be preferred. However, this typically implies a large variance of the error estimate, since there are only N/K test points.
- Typically, $K = 5, 10$ works well.

Kernel Selection

- Model: For example, ridge regression with a given value for the penalty factor λ. Only the parameters (weights) need to be determined (closed-form solution).
- Model selection: For example, determine the value for the penalty factor λ. Another example, determine the kernel and width for kernel classification, regression or density estimation. In either case, we do not have a continuous criterion to optimize. Solution: **Nested** cross-validation.

Cross-validation for estimating model prediction error

Nested cross-validation for estimating model **selection** prediction error



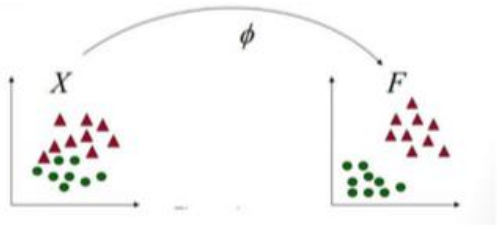
- Error overestimation may not be a concern for model selection. So, $K = 2$ may suffice in the inner loop.
- Which is the fitted model returned by nested cross-validation ?

Kernel Trick

- The kernel function $k\left(\frac{x-x'}{h}\right)$ is invariant to translations, and it can be generalized as $k(\boldsymbol{x}, \boldsymbol{x}')$. For instance,
  - Polynomial kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = (\boldsymbol{x}^T \boldsymbol{x}' + c)^M$
  - Gaussian kernel: $k(\boldsymbol{x}, \boldsymbol{x}') = exp(-||\boldsymbol{x} - \boldsymbol{x}'||^2/2\sigma^2)$
- If the matrix

$$\begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \dots & \vdots \\ k(\boldsymbol{x}_N, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix}$$

is symmetric and positive semi-definite for all choices of $\{\boldsymbol{x}_n\}$, then $k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T \phi(\boldsymbol{x}')$ where $\phi(\cdot)$ is a mapping from the input space to the feature space.



- The feature space may be non-linear and even infinite dimensional. For instance,

$$\phi(\boldsymbol{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2, \sqrt{2c}x_1, \sqrt{2c}x_2, c)$$

for the polynomial kernel with $M = D = 2$.

Support Vector Machines (SVM) for Classification:

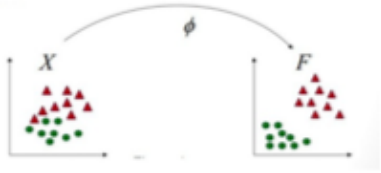
Support Vector Machines for Classification

- Consider binary classification with input space $\mathbb{R}^D$.
- Consider a training set $\{(\boldsymbol{x}_n, t_n)\}$ where $t_n \in \{-1, +1\}$.
- Consider using the linear model

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + b$$

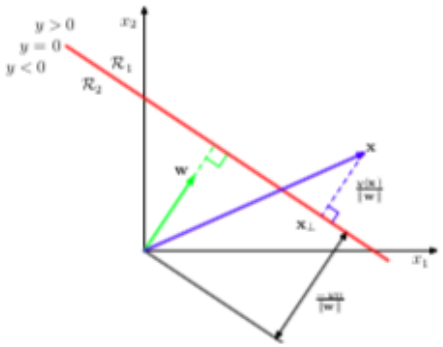
so that a new point $\boldsymbol{x}$ is classified according to the sign of $y(\boldsymbol{x})$.

- Assume that the training set is linearly separable in the feature space (but not necessarily in the input space), i.e. $t_n y(\boldsymbol{x}_n) > 0$ for all n.



- Aim for the separating hyperplane that maximizes the margin (i.e. the smallest perpendicular distance from any point to the hyperplane) so as to minimize the generalization error.

Support Vector Machines for Classification



- The perpendicular distance from any point to the hyperplane is given by

$$\frac{t_n y(\boldsymbol{x}_n)}{\|\boldsymbol{w}\|} = \frac{t_n(\boldsymbol{w}^T \phi(\boldsymbol{x}_n) + b)}{\|\boldsymbol{w}\|}$$

- Then, the maximum margin separating hyperplane is given by

$$\arg\max_{\boldsymbol{w}, b} \left(\min_n \frac{t_n(\boldsymbol{w}^T \phi(\boldsymbol{x}_n) + b)}{\|\boldsymbol{w}\|} \right)$$

- Multiply $\boldsymbol{w}$ and b by κ so that $t_n(\boldsymbol{w}^T \phi(\boldsymbol{x}_n) + b) = 1$ for the point closest to the hyperplane. Note that $t_n(\boldsymbol{w}^T \phi(\boldsymbol{x}_n) + b)/\|\boldsymbol{w}\|$ does not change.

- Then, the maximum margin separating hyperplane is given by

$$\arg\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2$$

subject to $t_n(\boldsymbol{w}^T\phi(\boldsymbol{x}_n)+b) \geq 1$ for all n.

- To minimize the previous expression, we minimize

$$\frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_n a_n\big(t_n(\boldsymbol{w}^T\phi(\boldsymbol{x}_n)+b)-1\big)$$

where $a_n \geq 0$ are called Lagrange multipliers.

- Note that any stationary point of the Lagrangian function is a stationary point of the original function subject to the constraints. Moreover, the Lagrangian function is a quadratic function subject to linear inequality constraints. Then, it is concave, actually concave up because of the $+1/2$ and, thus, "easy" to minimize.
- Note that we are now minimizing with respect to $\boldsymbol{w}$ and b, and maximizing with respect to a_n.
- Setting its derivatives with respect to $\boldsymbol{w}$ and b to zero gives

$$\boldsymbol{w} = \sum_n a_n t_n \phi(\boldsymbol{x}_n)$$

$$0 = \sum_n a_n t_n$$

- Replacing the previous expressions in the Lagrangian function gives the dual representation of the problem, in which we maximize

$$\sum_n a_n - \frac{1}{2}\sum_n\sum_m a_n a_m t_n t_m \phi(\boldsymbol{x}_n)^T\phi(\boldsymbol{x}_m) = \sum_n a_n - \frac{1}{2}\sum_n\sum_m a_n a_m t_n t_m k(\boldsymbol{x}_n,\boldsymbol{x}_m)$$

subject to $a_n \geq 0$ for all n, and $\sum_n a_n t_n = 0$.

- Again, this "easy" to maximize.
- Note that the dual representation makes use of the kernel trick, i.e. it allows working in a more convenient feature space without constructing it.

- When the Lagrangian function is maximized, the Karush-Kuhn-Tucker condition holds for all n:

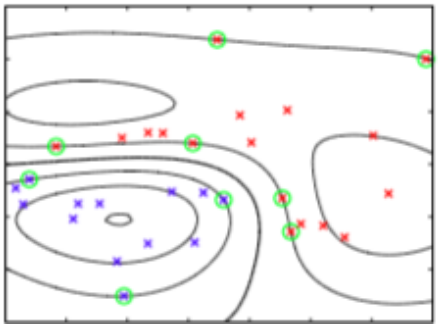
$$a_n(t_n y(\boldsymbol{x}_n) - 1) = 0$$

- Then, $a_n > 0$ if and only if $t_n y(\boldsymbol{x}_n) = 1$. The points with $a_n > 0$ are called support vectors and they lie on the margin boundaries.
- A new point $\boldsymbol{x}$ is classified according to the sign of

$$y(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) + b = \sum_n a_n t_n \phi(\boldsymbol{x}_n)^T\phi(\boldsymbol{x}) + b = \sum_n a_n t_n k(\boldsymbol{x},\boldsymbol{x}_n) + b$$

$$= \sum_{m\in\mathcal{S}} a_m t_m k(\boldsymbol{x},\boldsymbol{x}_m) + b$$

where $\mathcal{S}$ are the indexes of the support vectors. Sparse solution!



Support Vector Machines for Classification

- To find b, consider any support vector $\boldsymbol{x}_n$. Then,

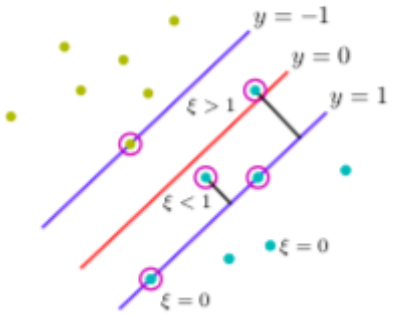
$$1 = t_n y(\boldsymbol{x}_n) = t_n\Big(\sum_{m\in\mathcal{S}} a_m t_m k(\boldsymbol{x}_n, \boldsymbol{x}_m) + b\Big)$$

and multiplying both sides by t_n, we have that

$$b = t_n - \sum_{m\in\mathcal{S}} a_m t_m k(\boldsymbol{x}_n, \boldsymbol{x}_m)$$

- We now drop the assumption of linear separability in the feature space, e.g. to avoid overfitting. We do so by introducing the slack variables $\xi_n \geq 0$ to penalize (almost-)misclassified points as

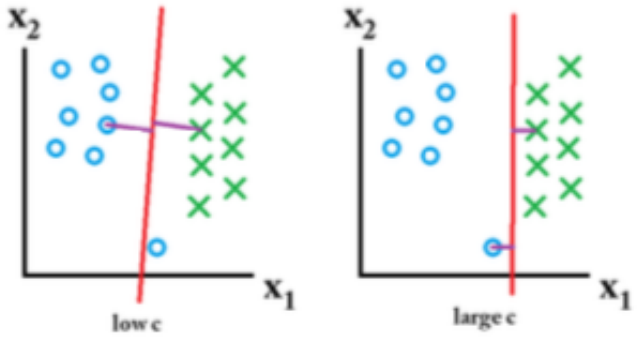
$$\xi_n = \begin{cases} 0 & \text{if } t_n y(\boldsymbol{x}_n) \geq 1 \\ |t_n - y(\boldsymbol{x}_n)| & \text{otherwise} \end{cases}$$



- The optimal separating hyperplane is given by

$$\arg\min_{\boldsymbol{w},b,\{\xi_n\}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_n \xi_n$$

subject to $t_n y(\boldsymbol{x}_n) \geq 1 - \xi_n$ and $\xi_n \geq 0$ for all n, and where $C > 0$ controls regularization. Its value can be decided by cross-validation. Note that the number of misclassified points is upper bounded by $\sum_n \xi_n$.



- To minimize the previous expression, we minimize

$$\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_n \xi_n - \sum_n a_n\big(t_n(\boldsymbol{w}^T\phi(\boldsymbol{x}_n) + b) - 1 + \xi_n\big) - \sum_n \mu_n\xi_n$$

where $a_n \geq 0$ and $\mu_n \geq 0$ are Lagrange multipliers.

- Setting its derivatives with respect to $\boldsymbol{w}$, b and ξ_n to zero gives

$$\boldsymbol{w} = \sum_n a_n t_n \phi(\boldsymbol{x}_n)$$
$$0 = \sum_n a_n t_n$$
$$a_n = C - \mu_n$$

- Replacing these in the Lagrangian function gives the dual representation of the problem, in which we maximize

$$\sum_n a_n - \frac{1}{2}\sum_n \sum_m a_n a_m t_n t_m k(\boldsymbol{x}_n, \boldsymbol{x}_m)$$

subject to $a_n \geq 0$ and $a_n \leq C$ for all n, because $\mu_n \geq 0$.

- When the Lagrangian function is maximized, the Karush-Kuhn-Tucker conditions hold for all n:

$$a_n(t_n y(\boldsymbol{x}_n) - 1 + \xi_n) = 0$$
$$\mu_n \xi_n = 0$$

- Then, $a_n > 0$ if and only if $t_n y(\boldsymbol{x}_n) = 1 - \xi_n$ for all n. The points with $a_n > 0$ are called support vectors and they lie
  - on the margin if $a_n < C$, because then $\mu_n > 0$ and thus $\xi_n = 0$, or
  - inside the margin (even on the wrong side of the decision boundary) if $a_n = C$, because then $\mu_n = 0$ and thus ξ_n is unconstrained.

Support Vector Machines (SVM) for Regression:

Support Vector Machines for Regression

- Consider regressing an unidimensional continuous random variable on a D-dimensional continuous random variable.
- Consider a training set $\{(\boldsymbol{x}_n, t_n)\}$. Consider using the linear model

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + b$$

- To get a sparse solution, instead of minimizing the classical regularized error function

$$\frac{1}{2}\sum_n (y(\boldsymbol{x}_n) - t_n)^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

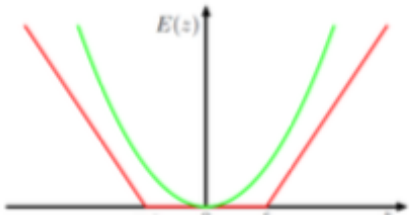
consider minimizing the ϵ-insensitive regularized error function

$$C\sum_n E_\epsilon(y(\boldsymbol{x}_n) - t_n) + \frac{1}{2}\|\boldsymbol{w}\|^2$$

where $C > 0$ controls regularization and

$$E_\epsilon(y(\boldsymbol{x}) - t) = \begin{cases} 0 & \text{if } |y(\boldsymbol{x}) - t| < \epsilon \\ |y(\boldsymbol{x}) - t| - \epsilon & \text{otherwise} \end{cases}$$

Figure 7.6 Plot of an ϵ-insensitive error function (in red) in which the error increases linearly with distance beyond the insensitive region. Also shown for comparison is the quadratic error function (in green).

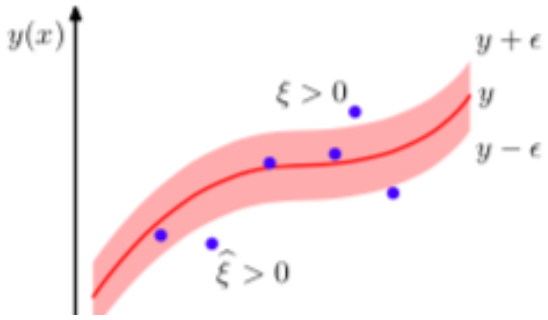


- The values of C and ϵ can be decided by cross-validation.
- Consider the slack variables $\xi_n \geq 0$ and $\widehat{\xi}_n \geq 0$ such that

$$\xi_n = \begin{cases} t_n - y(\boldsymbol{x}_n) - \epsilon & \text{if } t_n > y(\boldsymbol{x}_n) + \epsilon \\ 0 & \text{otherwise} \end{cases}$$

and

$$\widehat{\xi}_n = \begin{cases} y(\boldsymbol{x}_n) - \epsilon - t_n & \text{if } t_n < y(\boldsymbol{x}_n) - \epsilon \\ 0 & \text{otherwise} \end{cases}$$



- The optimal regression curve is given by

$$\underset{\boldsymbol{w}, b, \{\xi_n\}, \{\widehat{\xi}_n\}}{\arg\min} \; C \sum_n (\xi_n + \widehat{\xi}_n) + \frac{1}{2}\|\boldsymbol{w}\|^2$$

subject to $\xi \geq 0$, $\widehat{\xi}_n \geq 0$, $t_n \leq y(\boldsymbol{x}_n) + \epsilon + \xi_n$ and $t_n \geq y(\boldsymbol{x}_n) - \epsilon - \widehat{\xi}_n$.

- To minimize the previous expression, we minimize

$$C \sum_n (\xi_n + \widehat{\xi}_n) + \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_n (\mu_n \xi_n + \widehat{\mu}_n \widehat{\xi}_n)$$

$$- \sum_n a_n (y(\boldsymbol{x}_n) + \epsilon + \xi_n - t_n) - \sum_n \widehat{a}_n (t_n - y(\boldsymbol{x}_n) + \epsilon + \widehat{\xi}_n)$$

where $\mu_n \geq 0$, $\widehat{\mu}_n \geq 0$, $a_n \geq 0$ and $\widehat{a}_n \geq 0$ are Lagrange multipliers.

- Setting its derivatives with respect to $\boldsymbol{w}$, b, ξ_n and $\widehat{\xi}_n$ to zero gives

$$\boldsymbol{w} = \sum_n (a_n - \widehat{a}_n) \phi(\boldsymbol{x}_n)$$

$$0 = \sum_n (a_n - \widehat{a}_n)$$

$$C = a_n + \mu_n$$

$$C = \widehat{a}_n + \widehat{\mu}_n$$

- Replacing these in the Lagrangian function gives the dual representation of the problem, in which we maximize

$$\frac{1}{2}\sum_n\sum_m(a_n-\widehat{a}_n)(a_m-\widehat{a}_m)k(\boldsymbol{x}_n,\boldsymbol{x}_m)-\epsilon\sum_n(a_n+\widehat{a}_n)+\sum_n(a_n-\widehat{a}_n)t_n$$

subject to $a_n \geq 0$ and $a_n \leq C$ for all n, because $\mu_n \geq 0$. Similarly for $\widehat{a}_n$.

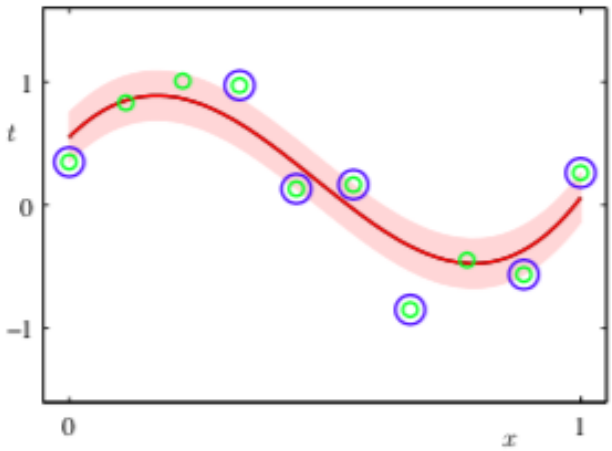
- When the Lagrangian function is maximized, the Karush-Kuhn-Tucker conditions hold for all n:

$$\begin{aligned} a_n(y(\boldsymbol{x}_n)+\epsilon+\xi_n-t_n) &= 0 \\ \widehat{a}_n(t_n-y(\boldsymbol{x}_n)+\epsilon+\widehat{\xi}_n) &= 0 \\ \mu_n\xi_n &= 0 \\ \widehat{\mu}_n\widehat{\xi}_n &= 0 \end{aligned}$$

- Then, $a_n > 0$ if and only if $y(\boldsymbol{x}_n)+\epsilon+\xi_n-t_n=0$, which implies that $\boldsymbol{x}_n$ lies on or above the upper margin of the ϵ-tube. Similarly for $\widehat{a}_n > 0$.
- The prediction for a new point $\boldsymbol{x}$ is made according to

$$y(\boldsymbol{x})=\sum_{m\in\mathcal{S}}(a_m-\widehat{a}_m)k(\boldsymbol{x},\boldsymbol{x}_m)+b$$

where $\mathcal{S}$ are the indexes of the support vectors. Sparse solution!



- To find b, consider any support vector $\boldsymbol{x}_n$ with $0 < a_n < C$. Then, $\mu_n > 0$ and thus $\xi_n = 0$ and thus $0 = t_n - \epsilon - y(\boldsymbol{x}_n)$. Then,

$$b=t_n-\epsilon-\sum_{m\in\mathcal{S}}(a_m-\widehat{a}_m)k(\boldsymbol{x}_n,\boldsymbol{x}_m)$$

Neural Networks:

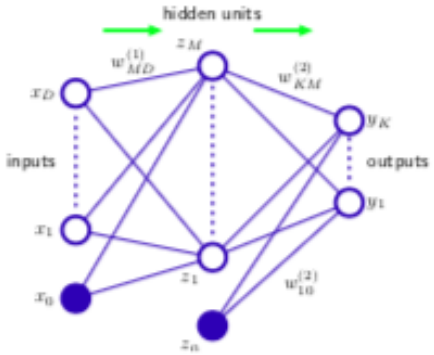
- Consider binary classification with input space $\mathbb{R}^D$. Consider a training set $\{(\boldsymbol{x}_n, t_n)\}$ where $t_n \in \{-1, +1\}$.
- SVMs classify a new point $\boldsymbol{x}$ according to

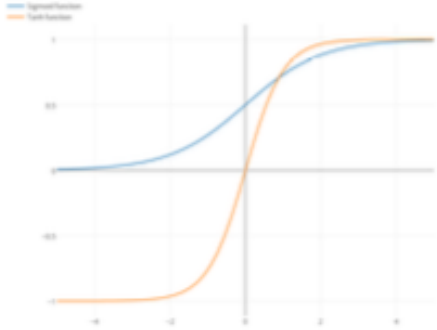
$$y(\boldsymbol{x}) = \text{sgn}\left(\sum_{m \in \mathcal{S}} a_m t_m k(\boldsymbol{x}, \boldsymbol{x}_m) + b\right)$$

- Consider regressing an unidimensional continuous random variable on a D-dimensional continuous random variable. Consider a training set $\{(\boldsymbol{x}_n, t_n)\}$
- For a new point $\boldsymbol{x}$, SVMs predict

$$y(\boldsymbol{x}) = \sum_{m \in \mathcal{S}} (a_n - \widehat{a}_n) k(\boldsymbol{x}, \boldsymbol{x}_m) + b$$

- SVMs imply data-selected user-defined basis functions.
- NNs imply a user-defined number of data-selected basis functions.





- Activations: $a_j = \sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)}$
- Hidden units and activation function: $z_j = h(a_j)$
- Output activations: $a_k = \sum_j w_{kj}^{(2)} z_j + w_{k0}^{(2)}$
- Output activation function for regression: $y_k(\boldsymbol{x}) = a_k$
- Output activation function for classification: $y_k(\boldsymbol{x}) = \sigma(a_k)$
- Sigmoid function: $\sigma(a) = \frac{1}{1+\exp(-a)}$
- Two-layer NN:

$$y_k(\boldsymbol{x}) = \sigma\left(\sum_j w_{kj}^{(2)} h\left(\sum_i w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

- Evaluating the previous expression is known as forward propagation. The NN is said to have a feed-forward architecture.
- All the previous is, of course, generalizable to more layers.

Backpropagation Algorithm

- Consider regressing an K-dimensional continuous random variable on a D-dimensional continuous random variable.
- Consider a training set $\{(\boldsymbol{x}_n, \boldsymbol{t}_n)\}$. Consider minimizing the sum-of-squares error function

$$E(\boldsymbol{w}) = \sum_n E_n(\boldsymbol{w}) = \sum_n \frac{1}{2}||\boldsymbol{y}(\boldsymbol{x}_n) - \boldsymbol{t}_n||^2 = \sum_n \sum_k \frac{1}{2}(y_k(\boldsymbol{x}_n) - t_{nk})^2$$

- This error function can be justified from a maximum likelihood approach to learning $\boldsymbol{w}$. To see it, **assume** that

$$p(t_k|\boldsymbol{x}, \boldsymbol{w}, \sigma) = \mathcal{N}(t_k|y_k(\boldsymbol{x}), \sigma)$$

- Then, the likelihood function is

$$p(\{\boldsymbol{t}_n\}|\{\boldsymbol{x}_n\}, \boldsymbol{w}, \sigma) = \prod_n \prod_k \mathcal{N}(t_{nk}|y_k(\boldsymbol{x}_n), \sigma) = \prod_n \prod_k \frac{1}{(2\pi\sigma^2)^{1/2}} e^{-\frac{1}{2\sigma^2}(t_{nk} - y_k(\boldsymbol{x}_n))^2}$$

and thus

$$-\ln p(\{\boldsymbol{t}_n\}|\{\boldsymbol{x}_n\}, \boldsymbol{w}, \sigma) = \sum_n \sum_k \frac{1}{2\sigma^2}(t_{nk} - y_k(\boldsymbol{x}_n))^2 + \frac{N}{2}\ln\sigma^2 + \frac{N}{2}\ln 2\pi$$

which is equivalent to the sum-of-squares error function for a given σ.

- If σ is not given, then we can find the ML estimates of $\boldsymbol{w}$, plug them into the log likelihood function, and maximize it with respect to σ.

- The weight space is highly multimodal and, thus, we have to resort to approximate iterative methods to minimize the previous expression.
- Batch gradient descent

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t \nabla E(\boldsymbol{w}^t)$$

where $\eta_t > 0$ is the learning rate ($\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$ to ensure convergence, e.g. $\eta_t = 1/t$).

- Sequential, stochastic or online gradient descent

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_t \nabla E_n(\boldsymbol{w}^t)$$

where n is chosen randomly or sequentially.

- Sequential gradient descent is less affected by the multimodality problem, as a local minimum of the whole data will not be generally a local minimum of each individual point.
- Since E_n depends on w_{ji} only via a_j, and $a_j = \sum_i w_{ji} x_i$, then

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} x_i = \delta_j x_i$$

- Since E_n depends on a_j only via a_k, then

$$\delta_j = \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial a_j}$$

- Since $a_k = \sum_j w_{kj} z_j$ and $z_j = h(a_j)$, then

$$\frac{\partial a_k}{\partial a_j} = h'(a_j) w_{kj}$$

- Putting all together, we have that

$$\delta_j = h'(a_j) \sum_k \delta_k w_{kj}$$

- Since $y_k = a_k$ for regression and $a_k = \sum_j w_{kj} z_j$, then

$$\frac{\partial E_n}{\partial w_{kj}} = \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial w_{kj}} = \delta_k z_j \text{ and } \delta_k = \frac{\partial E_n}{\partial a_k} = y_k - t_k$$

- Backpropagation algorithm:
  1. Forward propagate to compute activations, and hidden and output units.
  2. Compute δ_k for the output units.
  3. Backpropagate the δ's, i.e. evaluate δ_j for the hidden units recursively.
  4. Compute the required derivatives.

- For classification, we minimize the negative log likelihood function, a.k.a. cross-entropy error function:

$$E_n(\boldsymbol{w}) = -\sum_k [t_{nk} \ln y_k(\boldsymbol{x}_n) + (1 - t_{nk}) \ln(1 - y_k(\boldsymbol{x}_n))]$$

  with $t_{nk} \in \{0, 1\}$ and $y_k(\boldsymbol{x}_n) = \sigma(a_k)$. Then, again

$$\frac{\partial E_n}{\partial w_{kj}} = \delta_k z_j \text{ and } \delta_k = \frac{\partial E_n}{\partial a_k} = y_k - t_k$$

- Example: $y_k = a_k$, and $z_j = h(a_j) = \tanh(a_j)$ where $\tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$.
- Note that $h'(a) = 1 - h(a)^2$.
- Backpropagation:
  1. Forward propagation, i.e. compute
$$a_j = \sum_i w_{ji} x_i \text{ and } z_j = h(a_j) \text{ and } y_k = \sum_j w_{kj} z_j$$
  2. Compute
$$\delta_k = y_k - t_k$$
  3. Backpropagate, i.e. compute
$$\delta_j = (1 - z_j^2) \sum_k w_{kj} \delta_k$$
  4. Compute
$$\frac{\partial E_n}{\partial w_{kj}} = \delta_k z_j \text{ and } \frac{\partial E_n}{\partial w_{ji}} = \delta_j x_i$$
- The weight space is non-convex and has many symmetries, plateaus and local minima. So, the initialization of the weights in the backpropagation algorithm is crucial.
- Hints based on experimental rather than theoretical analysis:
  - Initialize the weights to different values, otherwise they would be updated in the same way because the algorithm is deterministic, and so creating redundant hidden units.
  - Initialize the weights at random, but
    - too small magnitude values may cause losing signal in the forward or backward passes, and
    - too big magnitude values may cause the activation function to saturate and lose gradient.
  - Initialize the weights according to prior knowledge: Almost-zero for hidden units that are unlikely to interact, and bigger magnitude values for the rest.
  - Initialize the weights to almost-zero values so that the initial model is almost-linear, i.e. the sigmoid function is almost-linear around the zero. Let the algorithm to introduce non-linearities where needed.
    - Note however that this initialization makes the sigmoid function take a value around half its saturation level. That is why the hyperbolic tangent function is sometimes preferred in practice.

Regularization

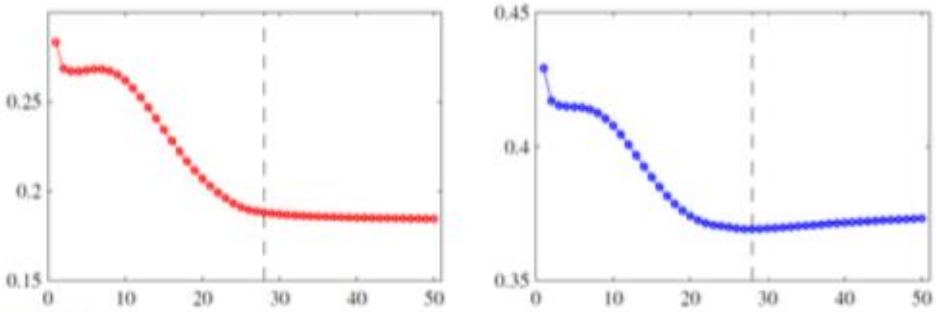


Figure 5.12 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

- Regularization when learning the parameters: Early stopping the backpropagation algorithm according to the error on some validation data.
- Regularization when learning the structure:
  - Cross-validation.
  - Penalizing complexity according to

$$E(\boldsymbol{w}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \text{ or } E(\boldsymbol{w}) + \frac{\lambda_1}{2}\|\boldsymbol{w}^{(1)}\|^2 + \frac{\lambda_2}{2}\|\boldsymbol{w}^{(2)}\|^2$$

  and choose λ, or λ_1 and λ_2 by cross-validation. Note that the effect of the penalty is simply to add λw_{ji} and λw_{kj}, or $\lambda_1 w_{ji}$ and $\lambda_2 w_{kj}$ to the appropriate derivatives.

Limitations of Neural Networks:

Theorem (Universal approximation theorem)

For every continuous function $f : [a, b]^D \to \mathbb{R}$ and for every $\epsilon > 0$, there exists a NN with one hidden layer such that

$$\sup_{\mathbf{x}\in[a,b]^D} |f(\mathbf{x}) - y(\mathbf{x})| < \epsilon$$

Theorem (Universal classification theorem)

Let $\mathcal{C}^{(k)}$ contain all classifiers defined by NNs of one hidden layer with k hidden units and the sigmoid activation function. Then, for any distribution $p(\mathbf{x}, t)$,

$$\lim_{k\to\infty} \inf_{y\in\mathcal{C}^{(k)}} L(y(\mathbf{x})) - L(p(t|\mathbf{x})) = 0$$

where $L()$ is the 0/1 loss function.

- How many hidden units has such a NN ?
- How much data do we need to learn such a NN (and avoid overfitting) via the backpropagation algorithm ?
- How fast does the backpropagation algorithm converge to such a NN ? Assuming that it does not get trapped in a local minimum...
- The answer to the last two questions depends on the first: More hidden units implies more training time and higher generalization error.

- How many hidden units does the NN need ?
- Any Boolean function can be written in disjunctive normal form (OR of ANDs) or conjunctive normal form (AND of ORs). This is a depth-two logical circuit.
- For most Boolean functions, the size of the circuit is exponential in the size of the input.
- However, there are Boolean functions that have a polynomial-size circuit of depth k and an exponential-size circuit of depth $k-1$.
- Then, there is no universally right depth. Ideally, we should let the data determine the right depth.

Theorem (No free lunch theorem)

For any algorithm, good performance on some problems comes at the expense of bad performance on some others.

Deep Neural networks:

- A deep NN is a function that maps input to output.
- The mapping is formed by composing many simpler functions.
- Each layer provides a new representation of the input, i.e. complex concepts are built from simpler ones.
- The representation is learned automatically from data.
- Training DNNs is difficult:
  - Typically, poorer generalization than (shallow) NNs.
  - The gradient may vanish/explode as we move away from the output layer, due to multiplying small/big quantities. E.g. the gradient of σ and tanh is in $[0, 1)$. So, they may only suffer the gradient vanishing problem. Other activations functions may suffer the gradient exploding problem.
  - There may be larger plateaus and many more local minima than with NNs.
- Training DNNs is doable:
  - Convolutional networks, particularly suitable for image processing.
  - Rectifier activation function, a new activation function.
  - Layer-wise pre-training, to find a good starting point for training.
- In addition to performance, the computational demands of the training must be considered, e.g. CPU, GPU, memory, parallelism, etc.
  - The authors state that GoogLeNet was trained "using modest amount of model and data-parallelism. Although we used a CPU based implementation only, a rough estimate suggests that the GoogLeNet network could be trained to convergence using few high-end GPUs within a week, the main limitation being the memory usage".

Convolutional Neural Networks:

- DNNs suitable for image recognition, since they exhibit invariance to translation, scaling, rotations, and warping.
- Convolution: Detection of local features, e.g. a_j is computed from a 5x5 pixel patch of the image.
- To achieve invariance, the units in the convolution layer share the same activation function and weights.
- Subsampling: Combination of local features into higher-order features, e.g. a_k is compute from a 2x2 pixel patch of the convoluted image.
- There are several feature maps in each layer, to compensate the reduction in resolution by increasing in the number of features being detected.
- The final layer is a regular NN for classification.
- DNNs allow increased depth because
  - they are sparse, which allows the gradient to propagate further, and
  - they have relatively few weights to fit due to feature locality and weight sharing.
- The backpropagation algorithm needs to be adapted, by modifying the derivatives with respect to the weights in each convolution layer m.
- Since E_n depends on $w_i^{(m)}$ only via $a_j^{(m)}$, and $a_j^{(m)} = \sum_{i \in L_j^{(m)}} w_i^{(m)} z_i^{(m-1)}$ where $L_j^{(m)}$ is the set of indexes of the input units, then

$$\frac{\partial E_n}{\partial w_i^{(m)}} = \sum_j \frac{\partial E_n}{\partial a_j^{(m)}} \frac{\partial a_j^{(m)}}{\partial w_i^{(m)}} = \sum_j \delta_j^{(m)} z_i^{(m-1)}$$

- Note that $w_i^{(m)}$ does not depend on j by weight sharing, whereas $i \in L_j^{(m)}$ by feature locality.

Rectifier Activation Function:

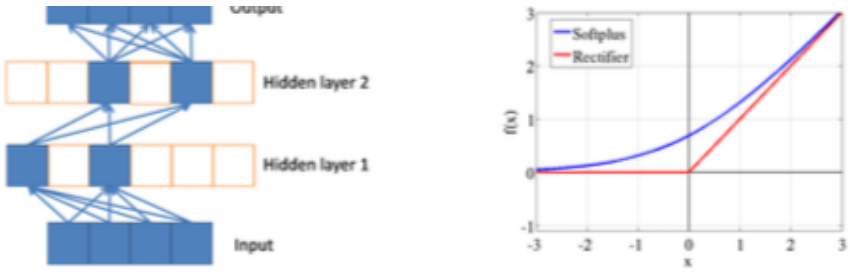


Figure 2: *Left:* **Sparse propagation of activations and gradients in a network of rectifier units.** The input selects a subset of active neurons and computation is linear in this subset. *Right:* **Rectifier and softplus activation functions.** The second one is a smooth version of the first.

- $rectifier(x) = \max\{0, x\}$, i.e. hidden units are off or operating in a linear regime.
- The most popular choice nowadays.
- Sparsity promoting: Uniform initialization of the weights implies that around 50 % of the hidden units are off.
- Piece-wise linear mapping: The input selects which hidden units are active, and the output is a liner function of the input in the selected hidden units.

- It simplifies the backpropagation algorithm as $h'(a_j) = 1$ for the selected units. So, there is no gradient vanishing on the paths of selected units. Compare with the sigmoid or hyperbolic tangent, for which
  - the gradient is smaller than one, or
  - even zero due to saturation.
- Note that $h'(0)$ does not exist since $h'_+(0) \neq h'_-(0)$. We can get around this problem by simply returning one of two one-sided derivatives. Or using a generalization of the rectifier function.
- Regularization is typically added to prevent numerical problems due to the activation being unbounded, e.g. when forward propagating.

Summary of Neural Networks:

- Direct application of the backpropagation algorithm to DNNs produces poor results.
- Convolutional networks: It makes the backpropagation algorithm more efficient by using local features and weight sharing. This also achieves invariance, which is particularly important for image processing.
- Rectifier activation function: Free of gradient vanishing problem and it simplifies the backpropagation algorithm.
- Layer-wise pre-training: Heuristic weight initialization to alleviate the local optima problem.

BLOC 2

1. Ensemble methods and mixture models. Online Learning.

Random Forest:

- Random forest = decision trees + bagging + decorrelation.

Algorithm 15.1 *Random Forest for Regression or Classification.*

1. For $b = 1$ to B:
   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data.
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the p variables.
      ii. Pick the best variable/split-point among the m.
      iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point x:

Regression: $\hat{f}_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$.

Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{rf}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

- Note that step 1ai aims to decorrelate the individual decision trees.

Boosting:

- Boosting is a technique to combine (weak) classifiers and so produce a more accurate (committee) classifier. It can also be applied to regression.
- Main steps:
  - Run the original classification algorithm on the original/modified training data.
  - Modify the training data by giving
    - more weight to the erroneously classified points, and
    - less weight to the correctly classified points.
  - Iterate through the previous steps a number of times.
  - Return a weighted average of the classifiers obtained.

Adaboost:

Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N$, $i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\text{err}_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))]$, $i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$.

Gradient Boosting:

Gradient Boosting

1. Initialize $f_0(x) = \arg\min_\gamma \sum_i L(y_i, b(x_i; \gamma))$
2. For $m = 1$ to M:
   (a) Compute the gradient residual $r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$
   (b) $\gamma_m = \arg\min_\gamma \sum_i (r_{im} - b(x_i; \gamma))^2$
   (c) $\beta_m = \arg\min_\beta \sum_i L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma_m))$
   (d) $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$

- Step 2b approximates the gradient residuals with the least squares basis function.
- Step 2a is easy in some cases, e.g. if $L(y, f(x)) = \frac{1}{2}(y - f(x))^2$ then

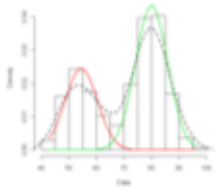
$$\frac{\partial L(y, f(x))}{\partial f(x)} = y - f(x)$$

Mixture Models:

- Sometimes the data do not follow any known probability distribution but a mixture of known distributions such as

$$p(\boldsymbol{x}) = \sum_{k=1}^{K} p(k)p(\boldsymbol{x}|k)$$

where $p(\boldsymbol{x}|k)$ are called mixture components and $p(k)$ are called mixing coefficients, which are usually denoted by π_k and $0 \leq \pi_k \leq 1$ and $\sum_k \pi_k = 1$.



- We can also see a mixture model as an ensemble model of a population with subpopulations:
  1. Choose a subpopulation according to $Multinomial(k|\pi_1, \ldots, \pi_K)$.
  2. Sample an instance from the chosen subpopulation according to $p(\boldsymbol{x}|k)$.
- Mixture of multivariate Gaussian distributions:

$$p(\boldsymbol{x}) = \sum_k \pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \text{ and } \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{2\pi^{D/2}} \frac{1}{|\boldsymbol{\Sigma}_k|^{1/2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k)}$$

- Note that a mixture model defines a proper probability distribution:

$$0 \leq p(\boldsymbol{x}) \leq 1 \text{ and } \int p(\boldsymbol{x})\, d\boldsymbol{x} = 1$$

- Mixture of multivariate Bernoulli distributions:

$$p(\boldsymbol{x}) = \sum_k \pi_k Bernoulli(\boldsymbol{x}|\boldsymbol{\mu}_k)$$

where we assume that

$$Bernoulli(\boldsymbol{x}|\boldsymbol{\mu}_k) = \prod_i Bernoulli(x_i|\mu_{ki}) = \prod_i \mu_{ki}^{x_i}(1-\mu_{ki})^{(1-x_i)}$$

Maximum Likelihood

- Given a sample $\{\boldsymbol{x}_n, k_n\}$ of size N from a mixture of multivariate Bernoulli distributions, rewrite it as $\{\boldsymbol{x}_n, \boldsymbol{z}_n\}$ where $\boldsymbol{z}_n$ is a K-dimensional binary vector having only the k_n-th element equal to 1.
- The log likelihood function is

$$\log p(\{\boldsymbol{x}_n, \boldsymbol{z}_n\}|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_n \log p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\mu}, \boldsymbol{\pi}) = \sum_n \log \prod_k [\pi_k \prod_i \mu_{ki}^{x_{ni}}(1-\mu_{ki})^{(1-x_{ni})}]^{z_{nk}}$$

$$= \sum_n \sum_k z_{nk} [\log \pi_k + \sum_i [x_{ni} \log \mu_{ki} + (1-x_{ni}) \log(1-\mu_{ki})]]$$

- Let $x'_{ni} = 1 - x_{ni}$ and $\mu'_{ki} = 1 - \mu_{ki}$. To maximize the log likelihood function subject to the constraints $\sum_k \pi_k = 1$ and $\mu_{ki} + \mu'_{ki} = 1$, we maximize

$$\sum_n \sum_k z_{nk} [\log \pi_k + \sum_i [x_{ni} \log \mu_{ki} + x'_{ni} \log \mu'_{ki}]] + \lambda(\sum_k \pi_k - 1) + \sum_k \sum_i \lambda_{ki}(\mu_{ki} + \mu'_{ki} - 1)$$

where λ and λ_{ki} are called Lagrange multipliers.[1]

- Setting to zero the derivatives with respect to π_k, μ_{ki} and μ'_{ki} gives

$$\pi_k = -\sum_n z_{nk}/\lambda \text{ and } \mu_{ki} = -\sum_n z_{nk} x_{ni}/\lambda_{ki} \text{ and } \mu'_{ki} = -\sum_n z_{nk} x'_{ni}/\lambda_{ki}$$

- Replacing this into the constraint gives $\lambda = -N$ and $\lambda_{ki} = -\sum_n z_{nk}$ and, thus,

$$\pi_k^{ML} = \frac{\sum_n z_{nk}}{N} \text{ and } \mu_{ki}^{ML} = \frac{\sum_n z_{nk} x_{ni}}{\sum_n z_{nk}}$$

[1] Any stationary point of the Lagrangian function is a stationary point of the original function subject to the constraints. Unfortunately, the log likelihood function is typically not concave.

- Given a sample $\{\boldsymbol{x}_n\}$ of size N from a mixture of multivariate Bernoulli distributions, the expected log likelihood function is

$$\mathbb{E}_{\boldsymbol{Z}}[\log p(\{\boldsymbol{x}_n, \boldsymbol{z}_n\}|\boldsymbol{\mu}, \boldsymbol{\pi})] = \sum_n \sum_{\boldsymbol{z}_n} p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}) \log p(\boldsymbol{x}_n, \boldsymbol{z}_n|\boldsymbol{\mu}, \boldsymbol{\pi})$$

$$= \sum_n \sum_{\boldsymbol{z}_n} p(\boldsymbol{z}_n|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}) \sum_k z_{nk}[\log \pi_k + \sum_i [x_{ni} \log \mu_{ki} + (1 - x_{ni}) \log(1 - \mu_{ki})]]$$

$$= \sum_n \sum_k p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})[\log \pi_k + \sum_i [x_{ni} \log \mu_{ki} + (1 - x_{ni}) \log(1 - \mu_{ki})]]$$

- Following a reasoning analogous to the complete-data case, we obtain that

$$\pi_k^{ML} = \frac{\sum_n p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{N}$$

$$\mu_{ki}^{ML} = \frac{\sum_n x_{ni} p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_n p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$$

- This is not a closed form solution because

$$p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}) = \frac{p(z_{nk}, \boldsymbol{x}_n|\boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_k p(z_{nk}, \boldsymbol{x}_n|\boldsymbol{\mu}, \boldsymbol{\pi})} = \frac{\pi_k p(\boldsymbol{x}_n|\boldsymbol{\mu}_k)}{\sum_k \pi_k p(\boldsymbol{x}_n|\boldsymbol{\mu}_k)}$$

but it suggests the following algorithm.

EM Algorithm:

EM algorithm

Set $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ to some initial values
Repeat until $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ do not change
  Compute $p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})$ for all k and n /* E step */
  Set π_k to π_k^{ML}, and μ_{ki} to μ_{ki}^{ML} for all k and i /* M step */

- Note that $p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})$ is computed for all k and n in each iteration:

$$p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi}) = \frac{p(z_{nk}, \boldsymbol{x}_n|\boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_k p(z_{nk}, \boldsymbol{x}_n|\boldsymbol{\mu}, \boldsymbol{\pi})} = \frac{\pi_k p(\boldsymbol{x}_n|\boldsymbol{\mu}_k)}{\sum_k \pi_k p(\boldsymbol{x}_n|\boldsymbol{\mu}_k)}$$

- The difficulty of maximizing the expected log likelihood function is not only that no closed form solution exists, but also that the landscape has typically many local optima. As a result, the EM algorithm is very sensitive to initialization.
- The EM algorithm can also be obtained by maximizing $\log p(\{\boldsymbol{x}_n\}|\boldsymbol{\mu}, \boldsymbol{\pi})$.
- The EM algorithm is guaranteed to increase $\log p(\{\boldsymbol{x}_n\}|\boldsymbol{\mu}, \boldsymbol{\pi})$ in each iteration until a local maximum is reached. So, the algorithm aims for the ML estimates.

- We can derive the EM algorithm for mixtures of multivariate Gaussian distributions in much the same way. Simply,

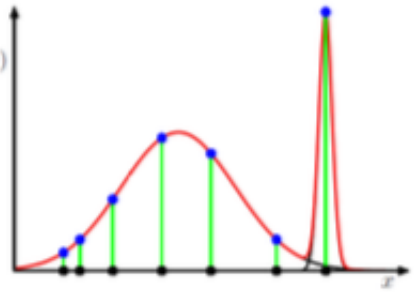
$$\pi_k^{ML} = \frac{\sum_n p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{N}$$

$$\boldsymbol{\mu}_k^{ML} = \frac{\sum_n \boldsymbol{x}_n p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_n p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$$

$$\boldsymbol{\Sigma}_k^{ML} = \frac{\sum_n (\boldsymbol{x}_n - \boldsymbol{\mu}_k^{ML})(\boldsymbol{x}_n - \boldsymbol{\mu}_k^{ML})^T p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_n p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$$

- Unlike in the case of mixture of multivariate Bernoulli distributions, there can be singularities, i.e. the log likelihood function goes to infinity when a component of the mixture collapses onto a single data point.

Figure 9.7 Illustration of how singularities in the likelihood function arise with mixtures of Gaussians. This should be compared with the case of a single Gaussian shown in Figure 1.14 for which no singularities arise.



- Solution: Reset the mean and covariance of the component to random and large values, respectively. Or adopt a Bayesian approach.

Number of Mixture Components

- Too few/many components will result in underfitting/overfitting.
- We can perform a search over the number of components by scoring each number with, for instance, the Bayesian information criterion (BIC):

$$\log p(\{\boldsymbol{x}_n\}|\boldsymbol{\mu}^{ML}, \boldsymbol{\pi}^{ML}) - \frac{M}{2}\log N$$

  where M is the number of free parameters in the mixture model. Note that the EM algorithm has to be run for each candidate number.
- Under some conditions, the score above can be seen as an approximation of the Bayesian score for a given number of components:

$$\log p(\{\boldsymbol{x}_n\}) = \int\int \log p(\{\boldsymbol{x}_n\}|\boldsymbol{\mu}, \boldsymbol{\pi}) p(\boldsymbol{\mu}, \boldsymbol{\pi}) d\boldsymbol{\mu} d\boldsymbol{\pi}$$

- There also exist algorithms that iteratively split and merge components according to the scores above until no further improvement occurs.
- Nested cross-validations is also an option.

K-means Algorithm:

- Recall that

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{2\pi^{D/2}} \frac{1}{|\boldsymbol{\Sigma}_k|^{1/2}} \exp(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x} - \boldsymbol{\mu}_k))$$

- Assume that $\boldsymbol{\Sigma}_k = \epsilon \boldsymbol{I}$ where ϵ is a variance parameter and $\boldsymbol{I}$ is the identity matrix. Then,

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{2\pi^{D/2}} \frac{1}{|\epsilon \boldsymbol{I}|^{1/2}} \exp(-\frac{1}{2\epsilon}||\boldsymbol{x} - \boldsymbol{\mu}_k||^2)$$

$$p(k|\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\pi}) = \frac{\pi_k p(\boldsymbol{x}|\boldsymbol{\mu}_k)}{\sum_k \pi_k p(\boldsymbol{x}|\boldsymbol{\mu}_k)} = \frac{\pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_k \pi_k \mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} = \frac{\pi_k \exp(-\frac{1}{2\epsilon}||\boldsymbol{x} - \boldsymbol{\mu}_k||^2)}{\sum_k \pi_k \exp(-\frac{1}{2\epsilon}||\boldsymbol{x} - \boldsymbol{\mu}_k||^2)}$$

- As $\epsilon \to 0$, the smaller $||\boldsymbol{x} - \boldsymbol{\mu}_k||^2$ the slower $\exp(-\frac{1}{2\epsilon}||\boldsymbol{x} - \boldsymbol{\mu}_k||^2)$ goes to 0.
- As $\epsilon \to 0$, instances are hard-assigned (i.e. with probability 1) to the subpopulation with closest mean. This clustering technique is known as K-means algorithm.
- Note that $\boldsymbol{\pi}$ and $\boldsymbol{\Sigma}_k$ play no role in the K-means algorithm whereas, in each iteration, $\boldsymbol{\mu}_k$ is updated to the average of the instances assigned to subpopulation k.
- The K-means algorithm can be used to initialize the EM algorithm.

Summary:

- Mixture models: To model complex distributions by linearly combining simple distributions.
- EM algorithm: To estimate the ML parameters of mixture models. It converges to a local maximum of the log likelihood of the observed data.
- We can see mixture models as model-based clustering, and the K-means algorithm as a limit case thereof.
- The EM algorithm can be used to estimate the ML parameters from data with any pattern of missing (at random) entries, i.e. not only one latent variable.

Online Learning:

- Offline learning: Batch of data used to optimize a function such as

$$\sum_n L(t_n, y(\boldsymbol{x}_n, \boldsymbol{\theta}))$$

where $L(\cdot)$ is log loss, squared error, 0/1 loss, etc.

- Online learning: Streaming data so we cannot wait until the end to start processing the data. Instead, we update our estimate with each new data point. It may even be an interesting option if the batch of data does not fit in main memory, e.g. scarce resources.

Online learning
for $n = 1, 2, \ldots$
Receive question $\boldsymbol{x}_n$
Predict $y(\boldsymbol{x}_n, \boldsymbol{\theta}_n)$
Receive true answer t_n
Suffer loss $L(t_n, y(\boldsymbol{x}_n, \boldsymbol{\theta}_n))$

- Examples: Weather prediction, spam filtering, small devices.
- Goal: Minimize the cumulative loss suffered by updating the predictor.

- Assume that $|\Theta| < \infty$ and $t_n = y(\boldsymbol{x}_n, \boldsymbol{\theta}^*)$ for all n with $\boldsymbol{\theta}^* \in \Theta$. The latter is also known as realizability.

Consistent

$\Theta_1 = \Theta$
for $n = 1, 2, \ldots$
  Receive question $\boldsymbol{x}_n$
  Choose any $\boldsymbol{\theta}_n \in \Theta_n$
  Predict $y(\boldsymbol{x}_n, \boldsymbol{\theta}_n)$
  Receive true answer t_n
  Update $\Theta_{n+1} = \{\boldsymbol{\theta} \in \Theta_n : y(\boldsymbol{x}_n, \boldsymbol{\theta}) = t_n\}$

- The algorithm above makes at most $|\Theta| - 1$ errors.

Halving

$\Theta_1 = \Theta$
for $n = 1, 2, \ldots$
  Receive question $\boldsymbol{x}_n$
  Predict $\arg\max_{r \in \{0,1\}} |\{\boldsymbol{\theta} \in \Theta_n : y(\boldsymbol{x}_n, \boldsymbol{\theta}) = r\}|$
  Receive true answer t_n
  Update $\Theta_{n+1} = \{\boldsymbol{\theta} \in \Theta_n : y(\boldsymbol{x}_n, \boldsymbol{\theta}) = t_n\}$

- The algorithm above makes at most $\log_2 |\Theta|$ errors.
- If we drop the realizability assumption, then our goal may be restated as minimizing the regret

$$\sum_n L(t_n, y(\boldsymbol{x}_n, \boldsymbol{\theta}_n)) - \min_{\boldsymbol{\theta}^* \in \Theta} \sum_n L(t_n, y(\boldsymbol{x}_n, \boldsymbol{\theta}^*))$$

- Note that the second term above is the loss of the batch solution.
- We may also be satisfied if we find a predictor whose regret grows sub-linearly with respect to the number of training points N.
- These goals seem hopeless anyway, as the true answers may come from an adversary who waits for our prediction and then answers the opposite label (*regret* $= N - N/2$ at least, e.g. if the batch solution is majority voting).
- Solution: Mild change of game. The adversary has to decide the true label before we make a prediction. Moreover, we are allowed to randomize our predictions, to cope with non-realizability.

Weighted majority (input: $\eta \in (0,1)$)

$\boldsymbol{w}_1 = (1/M, \ldots, 1/M)$ where $M = |\Theta|$

for $n = 1, 2, \ldots$

Choose $m \sim \boldsymbol{w}_n$ and predict $y(\boldsymbol{x}_n, \boldsymbol{\theta}_m)$

Compute cost $c_{nm} = |y(\boldsymbol{x}_n, \boldsymbol{\theta}_m) - t_n|$ for all m

Update $w_{n+1,i} = \frac{w_{ni} \exp(-\eta c_{ni})}{\sum_j w_{nj} \exp(-\eta c_{nj})}$ for all i

- The algorithm predicts the label 1 (vs. 0) with probability

$$p_n = \sum_m w_{nm} y(\boldsymbol{x}_n, \boldsymbol{\theta}_m)$$

- The expected 0-1 loss on round n can be written as

$$\sum_m w_{nm} |y(\boldsymbol{x}_n, \boldsymbol{\theta}_m) - t_n| = |\sum_m w_{nm} y(\boldsymbol{x}_n, \boldsymbol{\theta}_m) - t_n| = |p_n - t_n|$$

- Moreover, if $\eta = \sqrt{N \log M}$ then

$$\sum_n |p_n - t_n| \leq \min_{1 \leq m \leq M} \sum_n c_{nm} + 2\sqrt{N \log M}$$

which implies a sub-linear regret wrt N, i.e. $\frac{regret}{N} \to 0$ as $N \to \infty$.

- Moreover, if $\eta = 1/2$ then

$$\sum_n |p_n - t_n| \leq 2 \min_{1 \leq m \leq M} \sum_n c_{nm} + 4 \log M$$

which reduces to $4 \log M$ for the realizable case (similar to Halving).

- If we drop the assumption that $|\Theta| < \infty$, minimizing the regret may be achieved by online gradient descent, which updates the parameters as

$$\boldsymbol{\theta}_{n+1} = proj_{\Theta}(\boldsymbol{\theta}_n - \eta_n \nabla L(t_n, y(\boldsymbol{x}_n, \boldsymbol{\theta}_n)))$$

where $proj_{\Theta}(\boldsymbol{v}) = \arg\min_{\boldsymbol{\theta} \in \Theta} ||\boldsymbol{v} - \boldsymbol{\theta}||$, and it is needed only if Θ is a subset of $\mathbb{R}^D$.

- To ensure that stochastic gradient descent converges, we need to check that

$$\sum_{n=1}^{\infty} \eta_n = \infty \text{ and } \sum_{n=1}^{\infty} \eta_n^2 < \infty$$

e.g. $\eta_n = 1/n$.

- Convergence vs adaptation or concept drift, e.g. $\eta_n = 1/2$.

Online EM Algorithm:

EM algorithm

Set $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ to some initial values
Repeat until $\boldsymbol{\pi}$ and $\boldsymbol{\mu}$ do not change
  Compute $p(z_{nk}|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})$ for all n /* E step */
  Set π_k to π_k^{ML}, and μ_{ki} to μ_{ki}^{ML} for all k and i /* M step */

Mixture of Bernoullis	Mixture of Gaussians
$\pi_k^{ML} = \frac{\sum_n p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{N}$	$\pi_k^{ML} = \frac{\sum_n p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{N}$
$\mu_{ki}^{ML} = \frac{\sum_n x_{ni} p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_n p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$	$\boldsymbol{\mu}_k^{ML} = \frac{\sum_n \boldsymbol{x}_n p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_n p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$
	$\boldsymbol{\Sigma}_k^{ML} = \frac{\sum_n (\boldsymbol{x}_n - \boldsymbol{\mu}_k^{ML})(\boldsymbol{x}_n - \boldsymbol{\mu}_k^{ML})^T p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}{\sum_n p(z_{nk}\|\boldsymbol{x}_n, \boldsymbol{\mu}, \boldsymbol{\pi})}$

- Online EM algorithm: Replace the E step with a stochastic E step such as

$$S_{1:n} = (1 - \eta_n) S_{1:n-1} + \eta_n \mathbb{E}_{\boldsymbol{Z}}[S_n|\boldsymbol{\mu}_n, \boldsymbol{\pi}_n]$$

  where $\{\eta_n\}$ is the learning rate, e.g. $\eta_n = 1/n$. Similarly for the mixture of Gaussians.
- Convergent under mild conditions. Adaptative too.

Online Kernel Classifier:

- Kernel classifier:

$$y(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \sum_n \mathbf{1}_{\{t_n=1\}} k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) \leq \sum_n \mathbf{1}_{\{t_n=0\}} k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) \\ 1 & \text{otherwise} \end{cases}$$

- The kernel classifier implies no learning and, thus, it can be applied to online learning directly. Deploying it may however become computationally demanding (specially with scarce resources), since it sums over all the training points.

Online Support Vector Machines:

- Support vector machines:

$$y(\boldsymbol{x}) = \sum_{m \in S} a_m t_m k(\boldsymbol{x}, \boldsymbol{x}_m) + b$$

- Training a support vector machine needs to be adapted for online learning, i.e. retraining is not an option as the training data may grow very fast. Deploying it is feasible due to its sparseness, even with scarce resources.

	Online SVM
1	$\mathcal{S} = \emptyset$
2	$b = 0$
3	Receive a random example $(\boldsymbol{x}_i, t_i)$
4	Compute $y(\boldsymbol{x}_i) = \sum_{m \in \mathcal{S}} a_m t_m k(\boldsymbol{x}_i, \boldsymbol{x}_m) + b$
5	If $t_i y(\boldsymbol{x}_i) \leq 0$ then
6	$\mathcal{S} = \mathcal{S} \cup \{i\}$
7	$a_i = 1$
8	Go to step 3

- It converges to a separating hyperplane in the separable case.
- It does not attempt to maximize the margin.
- It only adds support vectors.
- Therefore, it may overfit the training data (and increase computational cost for deploying it).

	Budget online SVM (input: β and M)
1	$\mathcal{S} = \emptyset$
2	$b = 0$
3	Receive a random example $(\boldsymbol{x}_i, t_i)$
4	Compute $y(\boldsymbol{x}_i) = \sum_{m \in \mathcal{S}} a_m t_m k(\boldsymbol{x}_i, \boldsymbol{x}_m) + b$
5	If $t_i y(\boldsymbol{x}_i) \leq \beta$ then
6	$\mathcal{S} = \mathcal{S} \cup \{i\}$
7	$a_i = 1$
8	If $\lvert\mathcal{S}\rvert > M$ then $\mathcal{S} = \mathcal{S} \setminus \{\arg\max_{m \in \mathcal{S}} t_m (y(\boldsymbol{x}_m) - a_m t_m k(\boldsymbol{x}_m, \boldsymbol{x}_m))\}$
9	Go to step 3

- **Quiz**: Why does the removal criterion in step 8 make sense ?
- It tolerates mild noisy data but does not maximize the margin.
- It may still overfit the training data (and increase computational cost for deploying it) due to its myopic nature.
- More sophisticated addition and removal criteria are needed, e.g. LASVM which handles noisy data (slack variables) and converges to the batch solution.

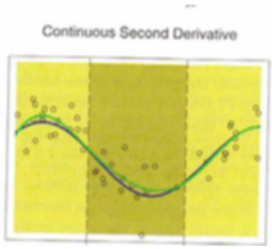
2. Splines and additive models. High-dimensional problems.

Splines:

- A piecewise polynomial is called an **order-M** (or degree *M*-1) **spline** if it is continuous and has continuous derivatives up to order *M*-2 at the knots.
- Equivalent: An order-*M* spline with *K* knots:

$$h_j(X) = X^{j-1},\ j = 1, \ldots, M$$
$$h_{M+l}(X) = (X - \xi_l)_+^{M-1},\ l = 1, \ldots, K$$

- An order-4 (degree-3) spline is called a **cubic spline**

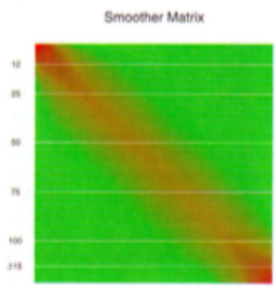


In cubic splines, knot discontinuity is not visible

$$\mathbf{S}_\lambda \mathbf{y} = \sum_{k=1}^{N} \mathbf{u}_k \rho_k(\lambda) \mathbf{u}_k^{\mathrm{T}} \mathbf{y}$$

- Smoothing spline decomposes vector **y** with respect to basis of eigenvectors and shrinks respective contributions

- The eigenvectors ordered by ρ increase in complexity. The higher the complexity, the more the contribution is shrunk.
- $df_\lambda = trace(\mathbf{S}_\lambda) \rightarrow \quad df_\lambda = \sum_{k=1}^{N} \frac{1}{1+\lambda d_k}$



- λ increase → df_λ decrease
- higher λ → higher penalization.
- Smoother matrix is has banded nature → local fitting method

How to fit data smoothly in higher dimensions?

- Formulate a new problem

$$\min \sum_i (y_i - f(x_i))^2 + \lambda J[f]$$

- The solution is **thin-plate splines**

- The solution in 2 dimensions is essentially sum of radial basis functions

$$f(x) = \beta_0 + \beta^T x + \sum \alpha_j \eta(\|x - x_j\|)$$

- Smoothing splines : smooth.spline()
- Natural cubic splines: ns() in **splines**
- Thin plate splines: Tps() in **fields**

```
res1=smooth.spline(data$Time,data$RSS_anchor2,df=10)
predict(res1,x=data$Time)$y
```

Generalized Additive Models:

- Model

$$Y \sim EF(\mu, \ldots)$$

where

  - $g(\mu) = \alpha + s_1(X_1) + s_2(X_2) + s_p(X_p)$
  - $s_i(X)$ - smoothers, normally splines
  - EF – distribution from exponential family
  - g – Link function

- Often linear terms are often included separately

$$EY = \alpha + s_1(X_1) + \ldots + s_p(X_p) + \sum_{j=1}^{q} \beta_j X_{p+j}$$

Example: EF= normal, EF=Bernoulli (logistic)

- Sometimes even higher orders are included (thin-plate splines)

$$g(\mu) = \alpha + s_1(X_1) + \ldots + s_p(X_p) + \sum_{j=1}^{q} \beta_j X_{p+j} + s_{12}(X_1, X_2)$$

- Method is reasonable to apply when additivity is observed or admissble

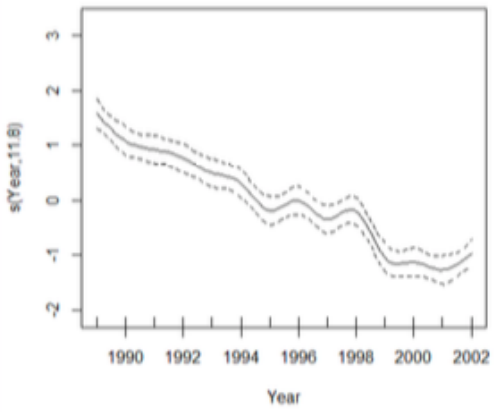
Example: Total Nitrogen level in Rhine river

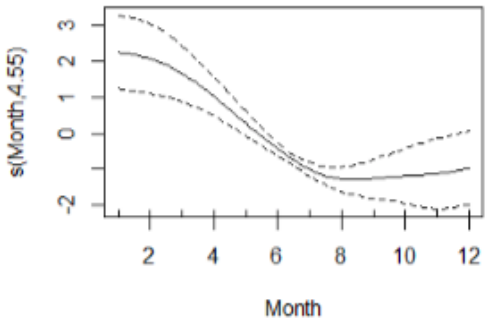
- Example: Modelling the concentration of total nitrogen at Lobith on the Rhine
  - There are seasonal trends (GAM reasonable)
  - Variables
    - Nitrogen level
    - Year
    - Month
- R: package **mgcv** (also package **gam**)
  - gam(formula, family,data,select, method)
    - Select allows for term (variable) selection
  - predict(), plot(), summary()...
  - s(k, sp)
    - k should be the same as the amount of **unique values** of this variable in **smoothing splines**
    - sp - smoothing penalty.

```
library(mgcv)
library(akima)
library(plotly)
river=read.csv2("Rhine.csv")
res=gam(TotN_conc~Year+Month+s(Year,
k=length(unique(river$Year)))+
          s(Month, k=length(unique(river$Month))),
data=river)

s=interp(river$Year,river$Month, fitted(res))
print(res)
summary(res)
res$sp
plot_ly(x=~s$x, y=~s$y, z=~s$z, type="surface")
```

- Seeing trend and seasonal pattern

plot(res)





High dimensional problems (wide data):

- **Wide data** $p \gg n$. Many variables, few data points.
  - **Genomics**
  - **Text**

- **Tall data**: $p \ll n$. Few variables, many data points.

Most of applications

- Economics, for ex. Currency exchange rates vs time
- Industry, Car performance characteristics vs probability of malfunctioning
- Surveys, customer satisfaction vs survey answers

- Tall and Wide. Supermarket scanners. Many purchases, many products.

- Linear regression $\mu = w^T x, Y \sim N(\mu, \sigma^2)$
- ML solution $\hat{w} = (X^T X)^{-1} X^T Y$
  - X is $n \times p$, has rank n
  - $X^T X$ is $p \times p$, has rank n
  - → $X^T X$ is not invertible!
- Solutions:
  - **Dimensionality reduction**: PCA, PCR
  - **Shrinkage**: Lasso, Ridge, Elastic network
  - **Forward variable selection**

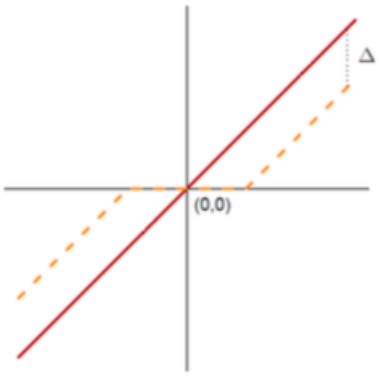
Nearest Shrunken Centroids

Nearest Shrunken Centroids

- Idea: Shrink classwise means towards overall mean

1. Compute $d_{kj} = \frac{\bar{x}_{kj} - \bar{x}_j}{m_k(s_j + s_0)}$
2. Shrink $d'_{kj} = sign(d_{kj})(|d_{kj}| - \Delta)_+$
3. Set $x'_{kj} = \bar{x}_j + m_k(s_j + s_0)d'_{kj}$

Only features with nonzero d'_{kj} contribute to classification!→insignificant features are shrunk!



Define Δ by CV

- Package **pamr**
  - pamr.train()
  - pamr.cv

```
data0=read.csv2("voice.csv")
data=data0
data=as.data.frame(scale(data))
data$Quality=as.factor(data0$Quality)
library(pamr)
rownames(data)=1:nrow(data)
x=t(data[,-311])
y=data[[311]]
mydata=list(x=x,y=as.factor(y),geneid=as.character(1:nrow(x)), genenames=rownames(x))
model=pamr.train(mydata,threshold=seq(0,4, 0.1))
pamr.plotcen(model, mydata, threshold=1)
pamr.plotcen(model, mydata, threshold=2.5)

a=pamr.listgenes(model,mydata,threshold=2.5)
cat( paste( colnames(data)[as.numeric(a[,1])], collapse='\n' ) )

cvmodel=pamr.cv(model,mydata)
print(cvmodel)
pamr.plotcv(cvmodel)
```

Regularized Discriminant Analysis:

- Another way of solving singularity of Σ
  - γ is some constant

$$\hat{\Sigma}(\gamma) = \gamma\hat{\Sigma} + (1-\gamma)diag(\hat{\Sigma})$$

- $\gamma = 0$→diagonal-covariance LDA
- γ is chosen by CV
- R: rda() in **klaR**

Regularized Logistic Regression:

- Usual logistic regression

$$p(Y = C_i|x) = \frac{e^{w_{i0}+\boldsymbol{w_i^T x}}}{\sum_{j=1}^{K} e^{w_{j0}+\boldsymbol{w_j^T x}}} = softmax(w_{i0} + \boldsymbol{w_i^T x})$$

- Lp -Regularization:

$$\max_{w} \sum_{i=1}^{n} \log p(Y_i|x_i) - \frac{\lambda}{2}\sum_{k=1}^{K} \|\boldsymbol{w}_i\|^p$$

- Parameter redunancy is solved
- **L1 regularization:** some w are shrunk to 0
- Numerical optimization is used to solve
- R: LiblineaR() in package **LiblineaR**
- Support Vector Machine do not suffer from $p \gg n$ problem
  - Largest margin can be found even if the data is perfectly separable

Support Vector Machine with wide data:

- SVD decomposition $X = UDV^T = RV^T$
- If model is linear in parameters and has quadratic penalties:
  - Transform data observations from X into R
  - Minimize loss (minus log likelihood) with R instead of X and get $\boldsymbol{\theta}$
  - Original parameters $\boldsymbol{w} = V\boldsymbol{\theta}$
- Can be applied to many methods

- Example: ridge regression

Elastic net:

- L1 regularization

$$\min_w -\log p(D|\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_1$$

$$\|w\|_1 = \sum_i |w_i|$$

- For p>n, LASSO can extract at most n nonzero components
  - Severe regularization if $p \gg n$
- L1 regularization→selects some feature among the correlated ones
- L2 regularization→ w's of the correlated variables are shrunk towards each other are nonzero

Combine L1 and L2 to diminish effect of L1 regularization.

- Elastic net regularization:

$$\min_w -\log p(D|\boldsymbol{w}) + \lambda(\alpha\|\boldsymbol{w}\|_1 + (1-\alpha)\|w\|_2)$$

- α is set ad hoc or chosen by CV
- Elastic net may select more than n features

- R: glmnet() in **glmnet** package
  - Specify "family" for classification or regression

Kernel PCA (when features are not available we can use K instead of X):

- Usual PCA
  - Center $\boldsymbol{X}$
  - Find $\boldsymbol{S}\boldsymbol{u}_i = \lambda_i \boldsymbol{u}_i, \boldsymbol{S} = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}, \boldsymbol{S} = [p \times p]$
    - $\boldsymbol{u}_i$ has dimension p
  - Project data on PCs: $Z = X\,U$

- Problems: X is unknown, and it can be p can be very large

- Kernel PCA: Equivalent formulation

1. Solve $\boldsymbol{K'a_i} = \lambda_i' \boldsymbol{a_i}, i = 1,..M$
   - $\boldsymbol{K} = \|K(\boldsymbol{x}_i, \boldsymbol{x}_j), i, j = 1,..n\|$
   - Centering $\boldsymbol{K'} = \boldsymbol{K} - \boldsymbol{1_n K} - \boldsymbol{K 1_n} + \boldsymbol{1_n K 1_n}$
   - $\lambda_i = \lambda_i'/n$
2. Scores for PC_i: $z_i(\boldsymbol{x}) = \sum_{i=1}^{n} a_{in} K(\boldsymbol{x}, \boldsymbol{x}_n)$

- There are at most *n* eigenvectors even if *p>>n*

- **Use kpca() in kernlab**

```
library(kernlab)
K <- as.kernelMatrix(crossprod(t(x)))
res=kpca(K)
barplot(res@eig)
plot(res@rotated[,1], res@rotated[,2], xlab="PC1",
ylab="PC2")
```

- Features assessment:
  - Individual gene: t-test

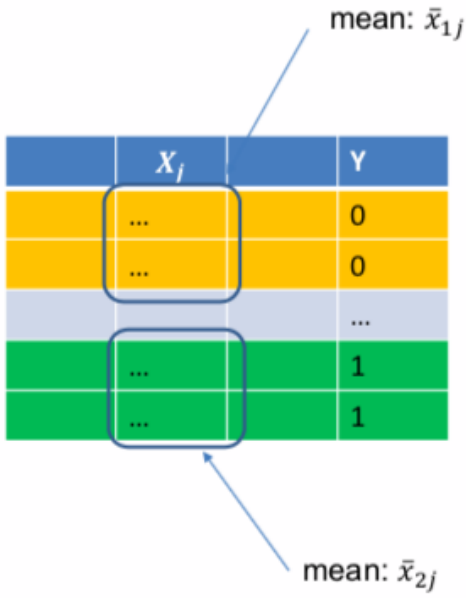
    H_{0j}: *treatment has no effect on gene j*

    H_{1j}: *treatment has an effect on gene j*

    $$t = \frac{\bar{x}_{2j} - \bar{x}_{1j}}{se_j}$$

  - Alternatively, nonparametric tests (permutation tests) can be used to compare two populations
  - Testing hypothesis for all genes?→multiple hypothesis testing
  - Control family-wise error rate
    - Bonferroni correction: $\alpha' = \alpha/M$
    - Ex: α=0.05, M=12000→$\alpha' \approx 10^{-6}$

  In practice, no genes with such small p-values



- Hypothesis testing Voice Rehabilitation
  - Feature ”MFCC_2nd.coef”

```
res=t.test(MFCC_2nd.coef~Quality,data=data,
alternative="two.sided")
res$p.value
```

```
> res$p.value
[1] 1.21246e-11
```

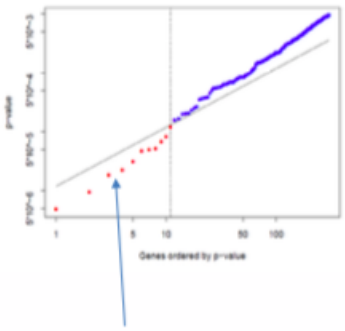
```
res=oneway_test(MFCC_2nd.coef~as.factor(Qu
ality), data=data,paired=FALSE)
pvalue(res)
```

```
> pvalue(res)
[1] 3.166942e-09
```

- Benjamini-Hochberg method (BH method)
  - Shown that $FDR(BH) < \alpha$ for independent hypotheses
  - →we can control FDR!

Algorithm 18.2 *Benjamini–Hochberg (BH) Method.*

1. Fix the false discovery rate α and let $p_{(1)} \leq p_{(2)} \leq \cdots \leq p_{(M)}$ denote the ordered p-values
2. Define
$$L = \max\left\{j : p_{(j)} < \alpha \cdot \frac{j}{M}\right\}.$$
3. Reject all hypotheses H_{0j} for which $p_j \leq p_{(L)}$, the BH rejection threshold.



Rejected hypotheses