Linear regression 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

An example of ordinary least squares regression

```
mydata=read.csv2("Bilexempel.csv")
fit1=lm(Price~Year, data=mydata)
summary(fit1)
fit2=lm(Price~Year+Mileage+Equipment,
data=mydata)
summary(fit2)
```

Response variable:
Requested price of used Porsche cars (1000 SEK)

Inputs:
X_1 = Manufacturing year
X_2 = Milage (km)
X_4 = Equipment (0 or 1)

```
> summary(fit1)

Call:
lm(formula = Price ~ Year, data = mydata)

Residuals:
    Min      1Q  Median      3Q     Max
-167683  -14683   20056   35933   72317

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -78161027    8448038  -9.252 6.00e-13 ***
Year            39246       4226   9.288 5.25e-13 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 57270 on 57 degrees of freedom
Multiple R-squared:  0.6021, Adjusted R-squared:  0.5952
F-statistic: 86.26 on 1 and 57 DF,  p-value: 5.248e-13
```

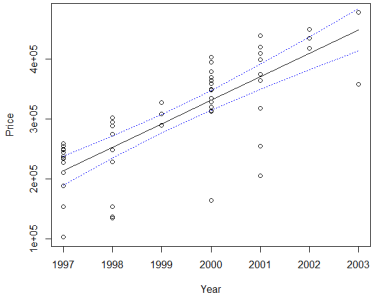

An example of ordinary least squares regression

```
> summary(fit2)

Call:
lm(formula = Price ~ Year + Mileage + Equipment, data = mydata)

Residuals:
   Min     1Q Median     3Q    Max
-66223 -10525   -739  14128  65332

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -2.083e+07  6.309e+06  -3.302  0.00169 **
Year         1.062e+04  3.154e+03   3.366  0.00139 **
Mileage     -2.077e+00  2.022e-01 -10.269 2.14e-14 ***
Equipment    5.790e+04  1.041e+04   5.563 8.08e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 29270 on 55 degrees of freedom
Multiple R-squared:  0.8997, Adjusted R-squared:  0.8942
F-statistic: 164.5 on 3 and 55 DF,  p-value: < 2.2e-16
```


An example of ordinary least squares regression

- Prediction

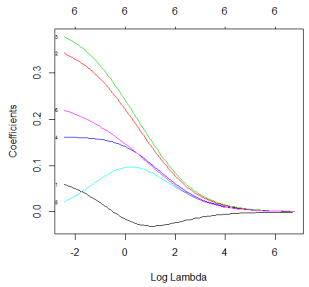
```
fitted <- predict(fit1, interval =
"confidence")

# 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

- 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)
```

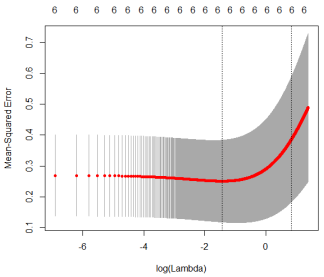


Ridge regression

- 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
[1] 0.046
```


Ridge regression

- 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**
- Equivalently

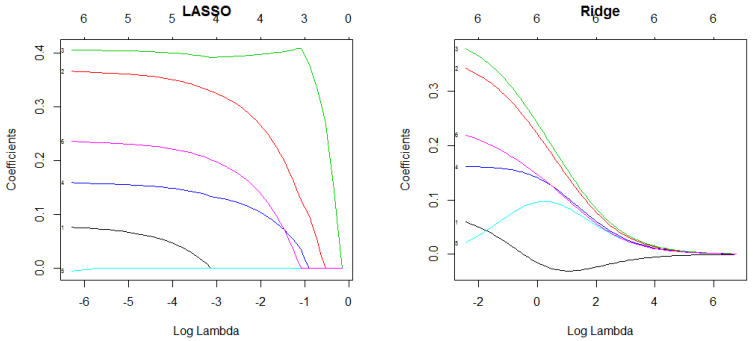
$$\hat{w}^{lasso} = \text{argmin}\sum_{i=1}^{N}(y_i - w_0 - w_1 x_{1j} - \ldots - w_p x_{pj})^2$$

subject to $\sum_{j=1}^{p}|w_i| \le s$

LASSO vs Ridge

- **LASSO yields sparse solutions!**

Example Computer hardware data



LASSO vs Ridge

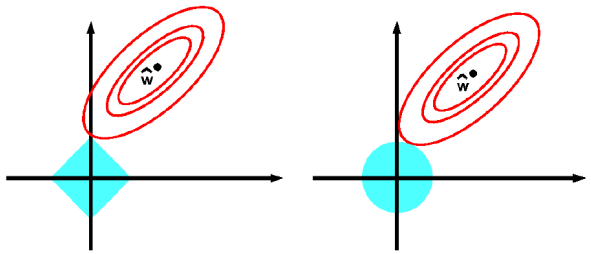
- Only 5 variables selected by LASSO

```
> coef(model, s="lambda.min")
7 x 1 sparse Matrix of class "dgCMatrix"
                      1
(Intercept) -5.091825e-17
V3           6.350488e-02
V4           3.578607e-01
V5           4.033670e-01
V6           1.541329e-01
V7           .
V8           2.287134e-01
> |
```

```
> sum((ynew-mean(y))^2)/sum((y-mean(y))^2)
[1] 0.5826904
> sum((ynew-y)^2)
[1] 16.63756
```


LASSO vs Ridge

- Why Lasso leads to sparse solutions?
  - Feasible area for Ridge is a circle (2D)
  - Feasible area for LASSO is a polygon (2D)



LASSO properies

- **Lasso is widely used when $p \gg n$**
  - Linear regression breaks down when $p > n$
  - Application: DNA sequence analysis, Text Prediction
- When inputs are orthonormal,

$$\hat{w}_i^{\text{lasso}} = sign\left(w_i^{linreg}\right)\left(\left|w_i^{linreg}\right| - \frac{\lambda}{2}\right)_+$$

- No explicit formula for $\hat{\boldsymbol{w}}^{\boldsymbol{lasso}}$
  - Optimization algorithms used

Coding in R: use glmnet() with alpha=1

Variable selection in R

- Use stepAIC() in MASS

```
library(MASS)
fit <- lm(V9~.,data=data.frame(data1))
step <- stepAIC(fit, direction="both")
step$anova
summary(step)
```

```
Call:
lm(formula = V9 ~ V3 + V4 + V5 + V6 + V8, data = data.fra

Residuals:
      Min        1Q    Median        3Q       Max
-1.20232 -0.15512  0.03579  0.16567  2.42280

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -5.783e-17  2.574e-02   0.000   1.0000
V3           7.948e-02  2.826e-02   2.813   0.0054 **
V4           3.661e-01  4.312e-02   8.490 4.34e-15 ***
V5           4.055e-01  4.664e-02   8.695 1.18e-15 ***
V6           1.591e-01  3.394e-02   4.687 5.07e-06 ***
V8           2.360e-01  3.356e-02   7.031 3.06e-11 ***
```

```
> step <- stepAIC(fit, direction="both")
Start:  AIC=-405.35
V9 ~ V3 + V4 + V5 + V6 + V7 + V8

       Df Sum of Sq    RSS     AIC
- V7    1    0.0139 28.117 -407.25
<none>               28.103 -405.35
- V3    1    1.0819 29.185 -399.46
- V6    1    2.9385 31.041 -386.57
- V8    1    6.3150 34.418 -364.99
- V4    1    9.7492 37.852 -345.11
- V5    1   10.4837 38.586 -341.09

Step:  AIC=-407.25
V9 ~ V3 + V4 + V5 + V6 + V8

       Df Sum of Sq    RSS     AIC
<none>               28.117 -407.25
+ V7    1    0.0139 28.103 -405.35
- V3    1    1.0958 29.212 -401.26
- V6    1    3.0431 31.160 -387.77
- V8    1    6.8472 34.964 -363.70
- V4    1    9.9840 38.101 -345.74
- V5    1   10.4713 38.588 -343.08
```


Model selection

Lecture 2b

Holdout in R

- How to partition into train/test?
  - Use set.seed(12345) in the labs to get identical results

```
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.7))
train=data[id,]
test=data[-id,]
```

- How to partition into train/valid/test?

```
n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.4))
train=data[id,]

id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
valid=data[id2,]

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

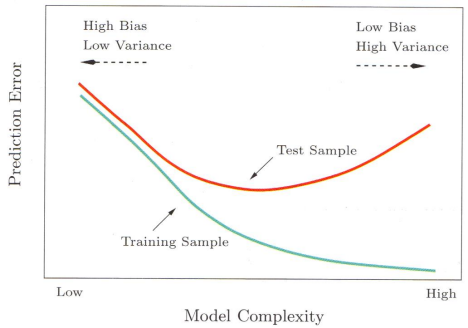

Bias-variance tradeoff

- 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!

Bias-variance tradeoff

- 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-validation

- Compared to holdout method:
  - Why do we use only some portion of data for training- can we use more (increase accuracy)?

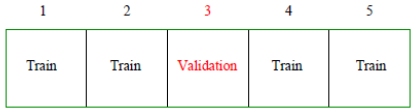
Cross-validation (Estimates Err)

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)

Cross-validation

Cross-validation



Note: if K=N then method is *leave-one-out* cross-validation.

$$\kappa : \{1, \ldots, N\} \mapsto \{1, \ldots, K\}$$

K-fold cross-validation: $CV = \frac{1}{N}\sum_{i=1}^{N} L(Y_i, \hat{y}^{-k(i)}(x_i))$

What to do if N is not a multiple of K?

Cross-validation vs Holdout

- 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

Analytical methods

- Analytical expressions to select models
  - AIC (Akaike's information criterion)

Idea: Instead of $R(Y,\hat{y}) = E[L(Y,\hat{y}(X,D))]$ consider **in-sample** risk (only *Y* in *D* is random):

$$R_{in}(Y,\hat{y}) = \frac{1}{N}\sum_{i=1}^{N} E_{Y_i}\left[L\left(Y_i,\hat{y}(X,D)\right)|D, X \in D\right]$$

Analytical methods

- One can show that

$$R_{in}(Y,\hat{y}) \approx R_{train} + \frac{2}{N}\sum_{i} cov(\hat{y}_i, Y_i)$$

where $R_{train} = \sum_{X_i,Y_i \in T} L(Y_i,\hat{y}_i)$

- Recall, **degrees of freedom** $\mathrm{df}(\mathrm{model}) = \frac{1}{\sigma^2}\sum_i cov(\hat{y}_i, Y_i)$
  - When model is linear, *df* is the number of parameters.

- If loss is defined by minus two loglikelihood,

$$AIC \equiv -2loglik(D) + 2df(model)$$

Model selection

Example Computer Hardware Data Set : performance measured for various processors and also

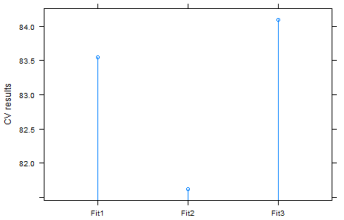
- Cycle time
- Memory
- Channels
- …

Build model predicting performance

Cross-validatation

- 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)
```



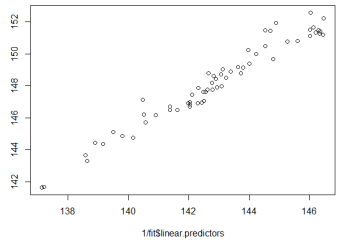
Generalized linear model

Assume

$$High \sim Gamma(1, \frac{1}{w_0 + w_1 Open})$$

What is link function here?

```
Call:
glm(formula = high ~ open, family = Gamma(link = "inverse"),
    data = data)

Deviance Residuals:
      Min         1Q     Median         3Q        Max
-0.0052879  -0.0028896  -0.0006678   0.0016598   0.0148083

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.355e-02  1.962e-04   69.06   <2e-16 ***
open        -4.604e-05  1.379e-06  -33.39   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



New generated data

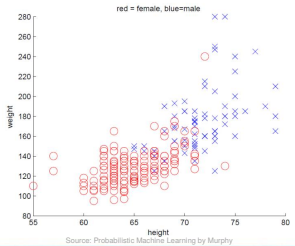
- Has similar pattern as original data!

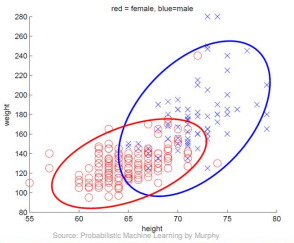
Quadratic discriminant analysis

- Generative classifier
- Main assumptions:
  - $\boldsymbol{x}$ is now **random** as well as y

$$p(\boldsymbol{x}|y = C_i, \theta) = N(\boldsymbol{x}|\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i})$$

Unknown parameters $\boldsymbol{\theta} = \{\boldsymbol{\mu_i}, \boldsymbol{\Sigma_i}\}$

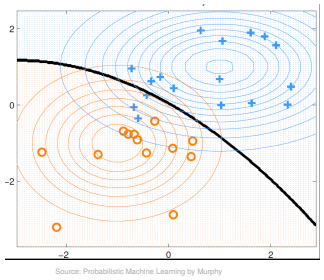




Quadratic discriminant analysis

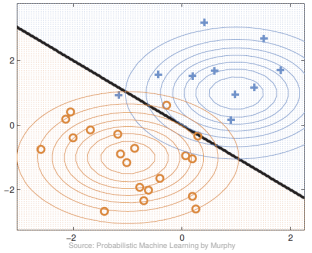
- If parameters are estimated, classify:

$$\hat{y}(\boldsymbol{x}) = \arg\max_c p(y = c|\boldsymbol{x}, \theta)$$



Linear discriminant analysis (LDA)

- Assumtion $\Sigma_i = \Sigma, i = 1, \dots K$
- Then $p(y = c_i|x) = softmax(w_i^T x + w_{0i})$
  →exactly the same form as the logistic regression
  - $w_{0i} = -\frac{1}{2}\boldsymbol{\mu}_i^T \Sigma^{-1} \boldsymbol{\mu}_i + \log \pi_i$
  - $w_i = \boldsymbol{\Sigma}^{-1} \boldsymbol{\mu}_i$



- Decision boundaries are linear
  - Discriminant function:

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T \Sigma^{-1} \mu_k + \log \pi_k$$

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**),

$$\hat{\pi}_c = \frac{N_c}{N}$$

LDA and QDA: code

- Syntax in R, library **MASS**

lda(formula, data, ..., subset, na.action)
- Prior – class probabiliies
- Subset – indices, if training data should be used

qda(formula, data, ..., subset, na.action)

predict(..)

LDA: output

```
resLDA=lda(Equipment~Mileage+Year, data=mydata)
 print(resLDA)
```

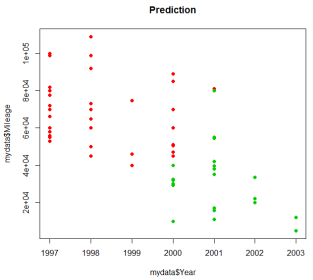
```
> print(resLDA)
Call:
lda(Equipment ~ Mileage + Year, data = mydata)

Prior probabilities of groups:
        0         1
0.6440678 0.3559322

Group means:
   Mileage     Year
0 63539.21 1998.447
1 36857.62 2000.762

Coefficients of linear discriminants:
                LD1
Mileage -1.500069e-05
Year     5.745893e-01
```


LDA: output

- Misclassified items

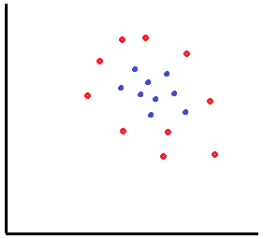
```
plot(mydata$Year, mydata$Mileage,
col=as.double(Pred$class)+1, pch=21,
bg=as.double(Pred$class)+1,
main="Prediction")
```

```
> table(Pred$class, mydata$Equipment)

   0  1
0 31  6
1  7 15
```



LDA versus Logistic regression

- 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 versus Logistic regression

- 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 Σ → 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.

Naive Bayes in R

- naiveBayes in package **e1071**

Example: Satisfaction of householders with their present housing circumstances

```
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)
```

```
> table(Yfit,housing1$Sat)

Yfit      Low Medium High
  Low     294    162  144
  Medium  20      23   20
  High    253    261  504
```


Decision trees in R

- **tree** package
  - Alternative: **rpart**

tree(formula, data, weights, control, split = c("deviance", "gini"), …)

print(), summary(), plot(), text()

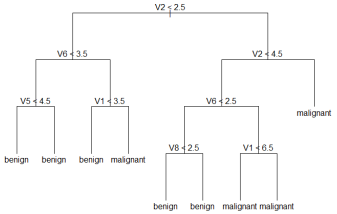
Example: breast cancer as a function av biological measurements

```
library(tree)
n=dim(biopsy)[1]
fit=tree(class~., data=biopsy)
plot(fit)
text(fit, pretty=0)
fit
summary(fit)
```

Decision trees in R

- Adjust the splitting in the tree with *control* parameter (leaf size for ex)

```
> fit
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

 1) root 683 884.400 benign ( 0.650073 0.349927 )
   2) V2 < 2.5 418 108.900 benign ( 0.971292 0.028708 )
     4) V6 < 3.5 395  25.130 benign ( 0.994937 0.005063 )
       8) V5 < 4.5 389   0.000 benign ( 1.000000 0.000000 ) *
       9) V5 > 4.5 6   7.638 benign ( 0.666667 0.333333 ) *
     5) V6 > 3.5 23  31.490 benign ( 0.565217 0.434783 )
      10) V1 < 3.5 11   0.000 benign ( 1.000000 0.000000 ) *
      11) V1 > 3.5 12  10.810 malignant ( 0.166667 0.833333 ) *
   3) V2 > 2.5 265 217.900 malignant ( 0.143396 0.856604 )
     6) V2 < 4.5 90 120.300 malignant ( 0.388889 0.611111 )
      12) V6 < 2.5 30  27.030 benign ( 0.833333 0.166667 )
        24) V8 < 2.5 19   0.000 benign ( 1.000000 0.000000 ) *
        25) V8 > 2.5 11  15.160 benign ( 0.545455 0.454545 ) *
      13) V6 > 2.5 60  54.070 malignant ( 0.166667 0.833333 )
        26) V1 < 6.5 28  35.160 malignant ( 0.321429 0.678571 ) *
        27) V1 > 6.5 32   8.900 malignant ( 0.031250 0.968750 ) *
     7) V2 > 4.5 175  30.350 malignant ( 0.017143 0.982857 ) *
```




```
> summary(fit)

Classification tree:
tree(formula = class ~ ., data = biopsy)
Variables actually used in tree construction:
[1] "V2" "V6" "V5" "V1" "V8"
Number of terminal nodes:  9
Residual mean deviance:  0.1603 = 108 / 674
Misclassification error rate: 0.03221 = 22 / 683
```


Decision trees in R

- Misclassification results

```
Yfit=predict(fit, newdata=biopsy, type="class")
table(biopsy$class,Yfit)
```

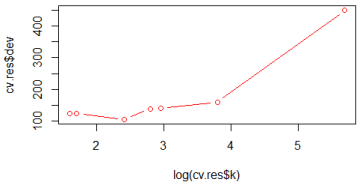
```
> table(biopsy$class,Yfit)
           Yfit
            benign malignant
  benign       440        18
  malignant      7       234
```

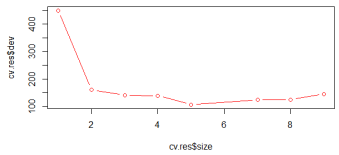

Decision trees in R

- 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?

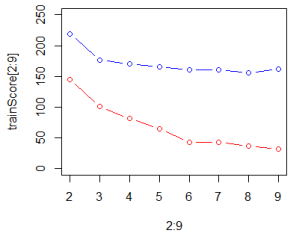




Decision trees in R

- 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")
```



What is optimal number of leaves?

Decision trees in R

- Final tree: 5 leaves

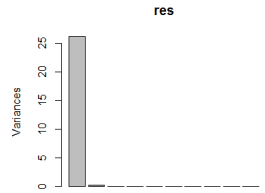
```
finalTree=prune.tree(fit, best=5)
Yfit=predict(finalTree, newdata=valid,
type="class")
table(valid$class,Yfit)
```

```
> table(valid$class,Yfit)
           Yfit
            benign malignant
  benign       222         8
  malignant      6       114
```


PCA in R

- 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)
```




```
> lambda
  [1] 2.612713e+01 2.385369e-01 7.844883e-02 3.018501e-
  [7] 2.052212e-04 1.084213e-04 2.077326e-05 1.150359e-
> sprintf("%2.3f",lambda/sum(lambda)*100)
  [1] "98.679" "0.901"  "0.296"  "0.114"  "0.006
  [9] "0.000"  "0.000"  "0.000"  "0.000"  "0.000
```

Only 1 component captures the 99% of variation!

PCA in R

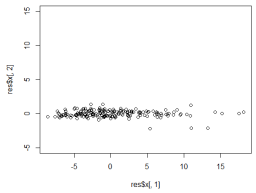
- Principal component **loadings (U)**

```
U=res$rotation
head(U)
```

```
> head(U)
                PC1        PC2        PC3
Channel1 0.07938192 0.1156228 0.08073156 -0.0927
Channel2 0.07987445 0.1170972 0.07887873 -0.0981
Channel3 0.08036498 0.1185571 0.07702127 -0.1031
Channel4 0.08085611 0.1200006 0.07515015 -0.1077
Channel5 0.08135022 0.1214075 0.07323819 -0.1119
Channel6 0.08184896 0.1227491 0.07125048 -0.1156
```

- Data in (PC1, PC2) – **scores (Z)**

plot(res$x[,1], res$x[,2], ylim=c(-5,15))

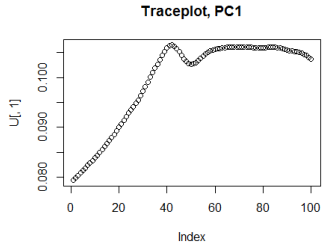


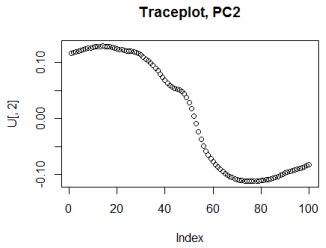
Do we need second dimension?

PCA in R

- Trace plots

```
U=loadings(res)
plot(U[,1], main="Traceplot, PC1")
plot(U[,2],main="Traceplot, PC2")
```





Which components contribute to PC1-2?

Probabilistic PCA in R

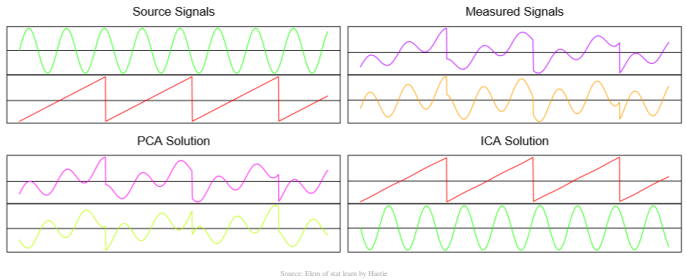
- Use **pcaMethods** from Bioconductor
- Install
  - source("https://bioconductor.org/biocLite.R")
  - biocLite("pcaMethods")

Ppca(data, nPcs,…)

Results: scores, loadings…

ICA

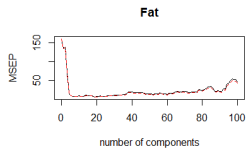
- Example



PCR and PLS: R

- Package **pls**
- PCR: pcr(formula, ncomp, data, scale = FALSE, validation = c("none", "CV", "LOO"…)
- PLS: plsr(…)

```
predictors=paste("Channel", 1:100, sep="")
f=formula(paste("Fat ~ ", paste(predictors,
collapse=" + ")))

set.seed(12345)
pcr.fit=pcr(f, data=train, validation="CV")
summary(pcr.fit)
validationplot(pcr.fit,val.type="MSEP")
```




```
> summary(pcr.fit)
Data:  X dimension: 150 100
 Y dimension: 150 1
Fit method: svdpc
Number of components considered: 100

VALIDATION: RMSEP
Cross-validated using 10 random segments.
       (Intercept)  1 comps  2 comps  3 comps  4 comps
CV           12.68    11.65    11.75    8.506    4.211
adjCV        12.68    11.65    11.74    8.500    4.198

TRAINING: % variance explained
      1 comps  2 comps  3 comps  4 comps  5 comps  6 comp
s
X       98.65    99.59    99.88    99.99   100.00   100.0
0
Fat     16.79    16.98    56.44    89.97    93.62    95.2
6
```


PCR

- Select 3 components

```
pcr.fit1=pcr(f, 3,data=train, validation="none")
summary(pcr.fit1)
coef(pcr.fit1)
scores(pcr.fit1)
l=loadings(pcr.fit1)
print(l,cutoff=0)
Yloadings(pcr.fit1)
plot(pcr.fit1)
```

```
> summary(pcr.fit1)
Data:  X dimension: 150 100
 Y dimension: 150 1
Fit method: svdpc
Number of components considered: 3
TRAINING: % variance explained
     1 comps  2 comps  3 comps
X      98.65    99.59    99.88
Fat    16.79    16.98    56.44
```

Coefficients in the original variables

```
> coef(pcr.fit1)
, , 3 comps

                Fat
Channel1  -2.61109872
Channel2  -2.56607281
Channel3  -2.52081831
Channel4  -2.47510841
Channel5  -2.42831883
Channel6  -2.37920922
Channel7  -2.32674365
Channel8  -2.27010925
Channel9  -2.20867856
Channel10 -2.14358679
```

PCR

```
> l=loadings(pcr.fit1)
> print(l,cutoff=0)

Loadings:
          Comp 1 Comp 2 Comp 3
Channel1   -0.079 -0.106  0.089
Channel2   -0.080 -0.108  0.088
Channel3   -0.080 -0.110  0.086
Channel4   -0.081 -0.112  0.084
Channel5   -0.081 -0.113  0.083
Channel6   -0.082 -0.115  0.081
Channel7   -0.082 -0.117  0.079
Channel8   -0.083 -0.118  0.077
Channel9   -0.083 -0.119  0.075
Channel10  -0.084 -0.121  0.073
Channel11  -0.084 -0.122  0.070
Channel12  -0.085 -0.123  0.068
```

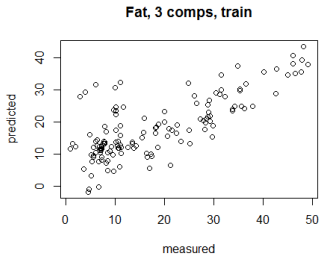
Scores matrix (new coordinates)

```
> scores(pcr.fit1)
           Comp 1       Comp 2       Comp 3
155  -9.66855445  0.092805568 -0.1012873027
188  -1.04084544 -0.307978589 -0.3180672681
163  -1.92212872 -0.243714308  0.0124365801
214   1.27768585 -0.232939083 -0.4315433137
97    2.55797242 -0.741279740  0.1582517775
35  -11.17415228  2.124582502  0.2004058835
68    2.96005563 -0.312953959  0.1945895756
106   3.71331162  0.015766621 -0.1130155332
```


PCR

```
> Yloadings(pcr.fit1)

Loadings:
    Comp 1  Comp 2  Comp 3
Fat  -1.015   1.114 -28.847
```

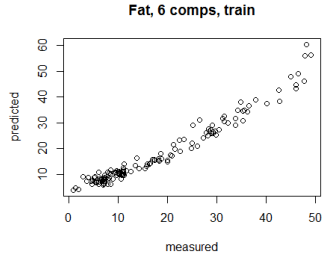
What is the regression equation in the new coordinates?



Is fit OK?

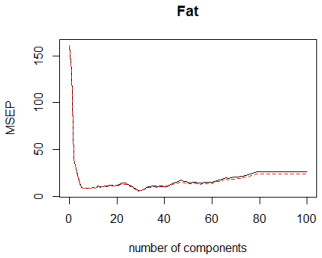
PCR

- Now 6 components




```
> summary(pcr.fit2)
Data:  X dimension: 150 100
 Y dimension: 150 1
Fit method: svdpc
Number of components considered: 6
TRAINING: % variance explained
    1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
X     98.65    99.59    99.88    99.99   100.00   100.00
Fat   16.79    16.98    56.44    89.97    93.62    95.29
```


PLS




```
> summary(pls.fit2)
Data:  X dimension: 150 100
 Y dimension: 150 1
Fit method: kernelpls
Number of components considered: 6
TRAINING: % variance explained
    1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
X     98.65    98.95    99.75    99.99   100.00   100.00
Fat   17.10    77.67    83.32    90.58    94.93    95.46
```

Bootstrap: R

- Package **boot**
  - **Functions:**
    - boot()
    - boot.ci() – 1 parameter
    - envelope() – many parameters
- Random random generation for parametic bootstrap:
  - Rnorm()
  - Runif()
  - …

boot(data, statistic, R, sim = "ordinary", ran.gen = function(d, p) d, mle = NULL,…)

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
```


Bootstrap: R

Parametric bootstrap:

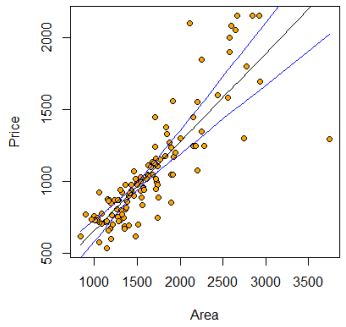
- Compute value *mle* that estimates model parameters from the data
- Write function *ran.gen* that depends on *data* and *mle* and which generates new data
- Write function *statistic* that depend on *data* which will be generated by *ran.gen* and should return the estimator

```
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")
```


Uncertainty estimation: R

- Bootstrap cofidence bands for linear model

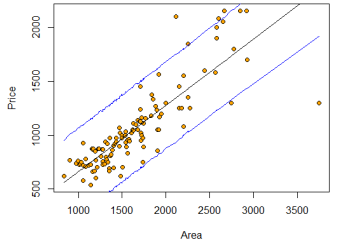
```
e=envelope(res) #compute confidence bands

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")
```



Estimation of the model quality

Example: parametric bootstrap

```
mle=lm(Price~Area, data=data2)

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)
  n=length(data2$Price)
  predictedP=rnorm(n,priceP, sd(mle$residuals))
  return(predictedP)
}
res=boot(data2, statistic=f1, R=10000,
mle=mle,ran.gen=rng, sim="parametric")
```



Why wider band?

732A95 Introduction to Machine Learning

Lecture 5a: Kernel Methods

Jose M. Peña
IDA, Linköping University, Sweden

Contents

Literature

- Main source
  - Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006. Sections 2.5 and 6.1-6.2.
- Additional source
  - Devroye, L., Györfi, L. and Lugosi, G. *A Probabilistic Theory of Pattern Recognition*. Springer, 1996. Sections 6.4 and 10.0.
  - Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning*. Springer, 2009. Chapter 6.

Histogram Classification

- Consider binary classification with input space $\mathbb{R}^D$.
- The best classifier under the 0-1 loss function is $y^*(\boldsymbol{x}) = \arg\max_y p(y|\boldsymbol{x})$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, t_n)\}$ available, then
  - divide the input space into D-dimensional cubes of side h, and
  - classify according to majority vote in the cube $C(\boldsymbol{x}, h)$ that contains $\boldsymbol{x}$.

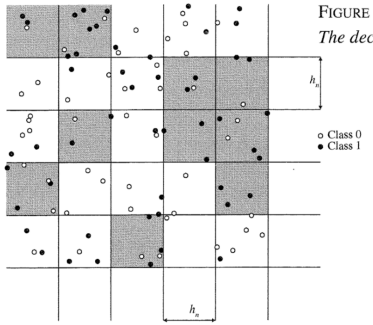


FIGURE 6.1. *A cubic histogram rule: The decision is* 1 *in the shaded area.*

- In other words,

$$y_C(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \sum_n \mathbf{1}_{\{t_n=1, \boldsymbol{x}_n \in C(\boldsymbol{x},h)\}} \leq \sum_n \mathbf{1}_{\{t_n=0, \boldsymbol{x}_n \in C(\boldsymbol{x},h)\}} \\ 1 & \text{otherwise} \end{cases}$$

Moving Window Classification

- The histogram rule is less accurate at the borders of the cube, because those points are not as well represented by the cube as the ones near the center. Then,
  - consider the points within a certain distance to the point to classify, and
  - classify the point according to majority vote.

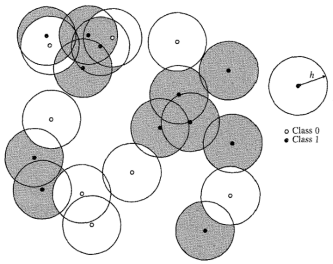


FIGURE 10.1. *The moving window rule in $\mathcal{R}^2$. The decision is 1 in the shaded area.*

- In other words,

$$y_S(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \sum_n \mathbf{1}_{\{t_n=1, \boldsymbol{x}_n \in S(\boldsymbol{x},h)\}} \leq \sum_n \mathbf{1}_{\{t_n=0, \boldsymbol{x}_n \in S(\boldsymbol{x},h)\}} \\ 1 & \text{otherwise} \end{cases}$$

where $S(\boldsymbol{x}, h)$ is a D-dimensional closed ball of radius h centered at $\boldsymbol{x}$.

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.

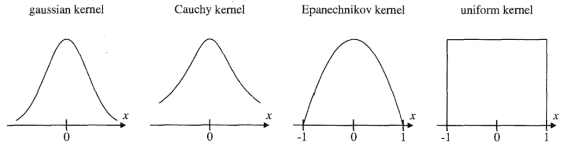


FIGURE 10.3. *Various kernels on $\mathcal{R}$.*

- 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: $k(u) = \mathbf{1}_{\{u \in S(0,1)\}}$

Kernel Classification

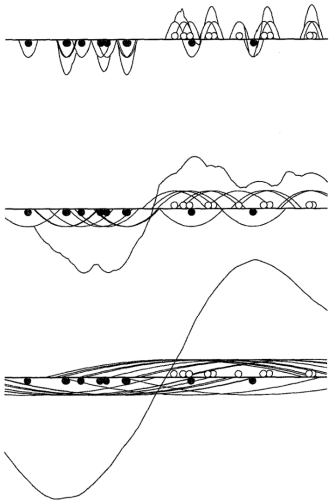


FIGURE 10.2. *Kernel rule on the real line. The figure shows $\sum_{i=1}^{n}(2Y_i - 1)K((x - X_i)/h)$ for $n = 20$, $K(u) = (1 - u^2)I_{\{|u| \leq 1\}}$ (the Epanechnikov kernel), and three smoothing factors h. One definitely undersmooths and one oversmooths. We took $p = 1/2$, and the class-conditional densities are $f_0(x) = 2(1 - x)$ and $f_1(x) = 2x$ on $[0, 1]$.*

Histogram, Moving Window, and Kernel Regression

- 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

$$y_k(\boldsymbol{x}) = \frac{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right) t_n}{\sum_n k\left(\frac{\boldsymbol{x}-\boldsymbol{x}_n}{h}\right)}$$

Histogram, Moving Window, and Kernel Density Estimation

- 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

$$p_k(\boldsymbol{x}) = \frac{1}{N} \sum_n k\Big(\frac{\boldsymbol{x} - \boldsymbol{x}_n}{h}\Big)$$

assuming that $k(u) \geq 0$ for all u and $\int k(u) du = 1$.

Histogram, Moving Window, and Kernel Density Estimation

Figure 2.24 An illustration of the histogram approach to density estimation, in which a data set of 50 data points is generated from the distribution shown by the green curve. Histogram density estimates, based on (2.241), with a common bin width Δ are shown for various values of Δ.

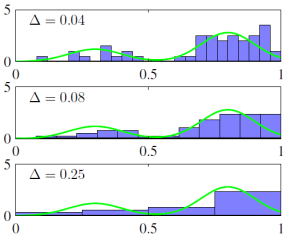
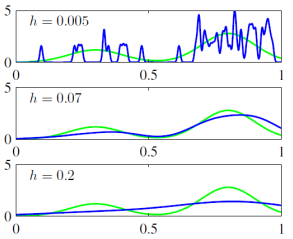


Figure 2.25 Illustration of the kernel density model (2.250) applied to the same data set used to demonstrate the histogram approach in Figure 2.24. We see that h acts as a smoothing parameter and that if it is set too small (top panel), the result is a very noisy density model, whereas if it is set 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).



Histogram, Moving Window, and Kernel Density Estimation

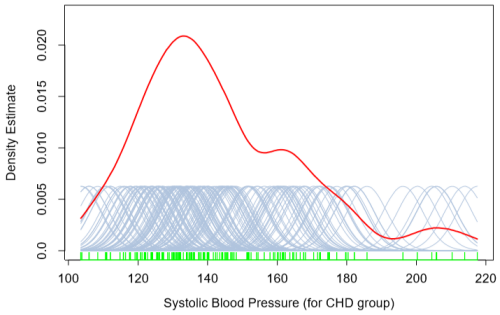


FIGURE 6.13. *A kernel density estimate for systolic blood pressure (for the CHD group). The density estimate at each point is the average contribution from each of the kernels at that point. We have scaled the kernels down by a factor of 10 to make the graph readable.*

- 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.

Histogram, Moving Window, and Kernel Density Estimation

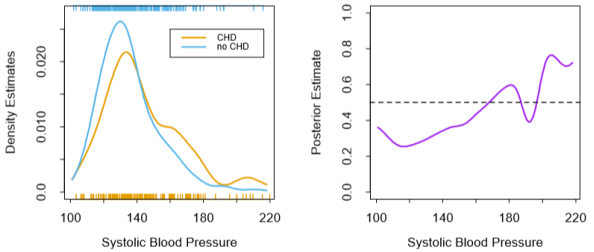
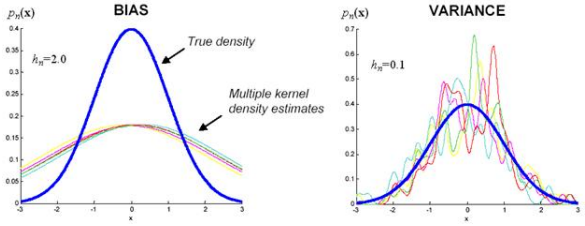


FIGURE 6.14. *The left panel shows the two separate density estimates for systolic blood pressure in the CHD versus no-CHD groups, using a Gaussian kernel density estimate in each. The right panel shows the estimated posterior probabilities for CHD, using (6.25).*

Kernel Selection

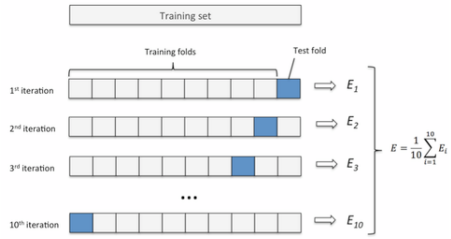
- How to choose the right kernel and width ? E.g., by cross-validation.
- What does "right" mean ? E.g., minimize loss function.
- Note that the width of the kernel corresponds to a bias-variance trade-off.



- Small width implies considering few points. So, the variance will the 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.



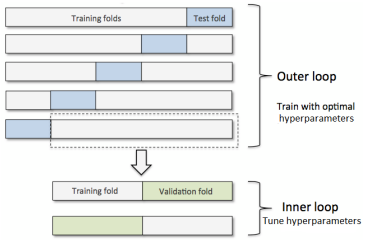
- If the training set contains N points, note that cross-validation estimates the prediction error when the model is trained on $N - N/K$ points.
- Note that the model returned is trained on N points. So, cross-validation overestimates the prediction error of the model returned.
- This seems to suggest 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



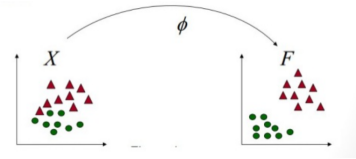
- Error overestimation may not be a concern for model selection. So, $K = 2$ may suffice in the inner loop.
- Which is the fitted model returned by nested cross-validation ?

Kernel Trick

- The kernel function $k\left(\frac{\boldsymbol{x}-\boldsymbol{x}'}{h}\right)$ 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 $M = D = 2$.

Kernel Trick

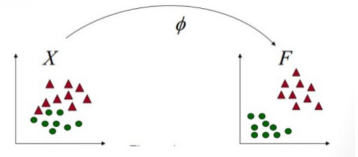
- Consider again moving window classification, regression, and density estimation.
- Note that $\boldsymbol{x}_n \in S(\boldsymbol{x}, h)$ if and only if $||\boldsymbol{x} - \boldsymbol{x}_n|| \leq h$.
- Note that

$$||\boldsymbol{x} - \boldsymbol{x}_n|| = (\boldsymbol{x} - \boldsymbol{x}_n)^T(\boldsymbol{x} - \boldsymbol{x}_n) = \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}_n^T\boldsymbol{x}_n - 2\boldsymbol{x}^T\boldsymbol{x}_n$$

- Then,

$$\begin{aligned} ||\phi(\boldsymbol{x}) - \phi(\boldsymbol{x}_n)|| &= \phi(\boldsymbol{x}^T)\phi(\boldsymbol{x}) + \phi(\boldsymbol{x}_n^T)\phi(\boldsymbol{x}_n) - 2\phi(\boldsymbol{x}^T)\phi(\boldsymbol{x}_n) \\ &= k(\boldsymbol{x}, \boldsymbol{x}) + k(\boldsymbol{x}_n, \boldsymbol{x}_n) - 2k(\boldsymbol{x}, \boldsymbol{x}_n) \end{aligned}$$

- So, the distance is now computed in a (hopefully) more convenient space.



- Note that we do not need to compute $\phi(\boldsymbol{x})$ and $\phi(\boldsymbol{x}_n)$.

Kernel Trick

- Two alternatives for building $k(\boldsymbol{x}, \boldsymbol{x}')$:
  - Choose a convenient $\phi(\boldsymbol{x})$ and let $k(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x})^T\phi(\boldsymbol{x}')$.
  - Build it from existing kernel functions as follows.

Techniques for Constructing New Kernels.

Given valid kernels $k_1(\mathbf{x}, \mathbf{x}')$ and $k_2(\mathbf{x}, \mathbf{x}')$, the following new kernels will also be valid:

$$\begin{aligned}
k(\mathbf{x}, \mathbf{x}') &= ck_1(\mathbf{x}, \mathbf{x}') && (6.13)\\
k(\mathbf{x}, \mathbf{x}') &= f(\mathbf{x})k_1(\mathbf{x}, \mathbf{x}')f(\mathbf{x}') && (6.14)\\
k(\mathbf{x}, \mathbf{x}') &= q\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.15)\\
k(\mathbf{x}, \mathbf{x}') &= \exp\left(k_1(\mathbf{x}, \mathbf{x}')\right) && (6.16)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}') + k_2(\mathbf{x}, \mathbf{x}') && (6.17)\\
k(\mathbf{x}, \mathbf{x}') &= k_1(\mathbf{x}, \mathbf{x}')k_2(\mathbf{x}, \mathbf{x}') && (6.18)\\
k(\mathbf{x}, \mathbf{x}') &= k_3\left(\boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\phi}(\mathbf{x}')\right) && (6.19)\\
k(\mathbf{x}, \mathbf{x}') &= \mathbf{x}^{\mathrm{T}}\mathbf{A}\mathbf{x}' && (6.20)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a) + k_b(\mathbf{x}_b, \mathbf{x}'_b) && (6.21)\\
k(\mathbf{x}, \mathbf{x}') &= k_a(\mathbf{x}_a, \mathbf{x}'_a)k_b(\mathbf{x}_b, \mathbf{x}'_b) && (6.22)
\end{aligned}$$

where $c > 0$ is a constant, $f(\cdot)$ is any function, $q(\cdot)$ is a polynomial with nonnegative coefficients, $\boldsymbol{\phi}(\mathbf{x})$ is a function from $\mathbf{x}$ to $\mathbb{R}^M$, $k_3(\cdot, \cdot)$ is a valid kernel in $\mathbb{R}^M$, $\mathbf{A}$ is a symmetric positive semidefinite matrix, $\mathbf{x}_a$ and $\mathbf{x}_b$ are variables (not necessarily disjoint) with $\mathbf{x} = (\mathbf{x}_a, \mathbf{x}_b)$, and k_a and k_b are valid kernel functions over their respective spaces.

Summary

- Kernel methods: Smoothing models.
- Model selection: Nested cross-validation.
- Kernel trick: It allows to work in the feature space without constructing it.

732A95 Introduction to Machine Learning

Lecture 5b: Support Vector Machines

Jose M. Peña
IDA, Linköping University, Sweden

Contents

Literature

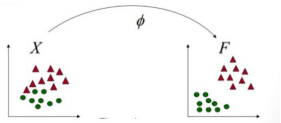
- Main source
  - Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006. Section 7.1.

Support Vector Machines for Classification

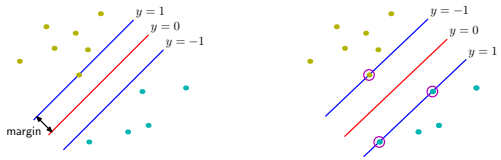
- Consider binary classification with input space $\mathbb{R}^D$.
- Consider a training set $\{(\boldsymbol{x}_n, t_n)\}$ where $t_n \in \{-1, +1\}$.
- Consider using the linear model

$$y(\boldsymbol{x}) = \boldsymbol{w}^T \phi(\boldsymbol{x}) + b$$

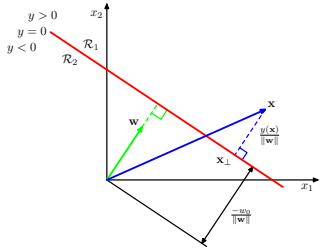
  so that a new point $\boldsymbol{x}$ is classified according to the sign of $y(\boldsymbol{x})$.
- Assume that the training set is linearly separable in the feature space (but not necessarily in the input space), i.e. $t_n y(\boldsymbol{x}_n) > 0$ for all n.



- 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.

Support Vector Machines for Classification

- 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

$$\boldsymbol{w} = \sum_n a_n t_n \phi(\boldsymbol{x}_n)$$

$$0 = \sum_n a_n t_n$$

Support Vector Machines for Classification

- 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.

Support Vector Machines for Classification

- When the Lagrangian function is maximized, the Karush-Kuhn-Tucker condition holds for all n:

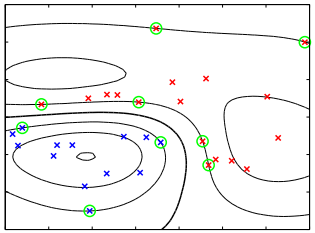
$$a_n(t_n y(\boldsymbol{x}_n) - 1) = 0$$

- 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.



Support Vector Machines for Classification

- To find b, consider any support vector $\boldsymbol{x}_n$. Then,

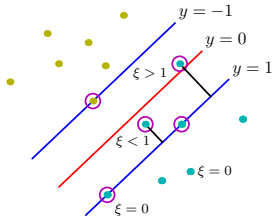
$$1 = t_n y(\boldsymbol{x}_n) = t_n\Big(\sum_{m\in\mathcal{S}} a_m t_m k(\boldsymbol{x}_n,\boldsymbol{x}_m) + b\Big)$$

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

$$\xi_n = \begin{cases} 0 & \text{if } t_n y(\boldsymbol{x}_n) \geq 1 \\ |t_n - y(\boldsymbol{x}_n)| & \text{otherwise} \end{cases}$$

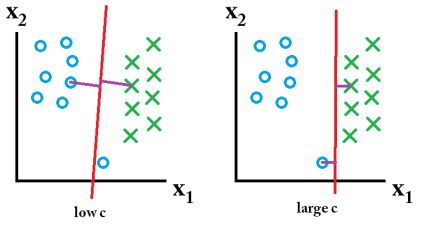


Support Vector Machines for Classification

- The optimal separating hyperplane is given by

$$\arg\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_n \xi_n$$

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.

Support Vector Machines for Classification

- 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 $\mu_n \geq 0$.

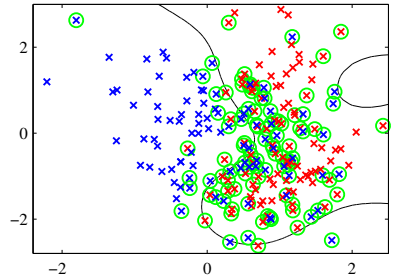
- When the Lagrangian function is maximized, the Karush-Kuhn-Tucker conditions hold for all n:

$$a_n(t_n y(\boldsymbol{x}_n) - 1 + \xi_n) = 0$$
$$\mu_n\xi_n = 0$$

- 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 for Classification

- Since the optimal $\boldsymbol{w}$ takes the same form as in the linearly separable case, classifying a new point is done the same as before. Finding b is done the same as before by considering any support vector $\boldsymbol{x}_n$ with $0 < a_n < C$.



- Not covered topics:
  - Classifying into more than two classes.
  - Returning class posterior probabilities.

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

$$y(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) + b$$

- Instead of minimizing the classical regularized error function

$$\frac{1}{2}\sum_n (y(\boldsymbol{x}_n) - t_n)^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

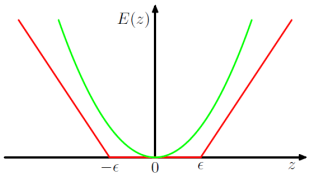
consider minimizing the ϵ-insensitive regularized error function

$$C\sum_n E_\epsilon(y(\boldsymbol{x}_n) - t_n) + \frac{1}{2}\|\boldsymbol{w}\|^2$$

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).



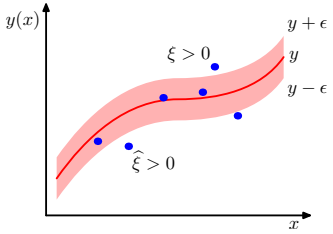
Support Vector Machines for Regression

- The values of C and ϵ can be decided by cross-validation.
- Consider the slack variables $\xi \geq 0$ 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

$$\widehat{\xi}_n = \begin{cases} y(\boldsymbol{x}_n) + \epsilon - t_n & \text{if } t_n < y(\boldsymbol{x}_n) + \epsilon \\ 0 & \text{otherwise} \end{cases}$$



Support Vector Machines for Regression

- The optimal regression curve is given by

$$\arg\min_{\boldsymbol{w}} C \sum_n (\xi_n + \widehat{\xi}_n) + \frac{1}{2}\|\boldsymbol{w}\|^2$$

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

$$\boldsymbol{w} = \sum_n (a_n - \widehat{a}_n)\phi(\boldsymbol{x}_n)$$

$$0 = \sum_n (a_n - \widehat{a}_n)$$

$$C = a_n + \mu_n$$

$$C = \widehat{a}_n + \widehat{\mu}_n$$

Support Vector Machines for Regression

- 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:

$$a_n(y(\boldsymbol{x}_n) + \epsilon + \xi_n - t_n) = 0$$

$$\widehat{a}_n(t_n - y(\boldsymbol{x}_n) + \epsilon + \widehat{\xi}_n) = 0$$

$$\mu_n \xi_n = 0$$

$$\widehat{\mu}_n \widehat{\xi}_n = 0$$

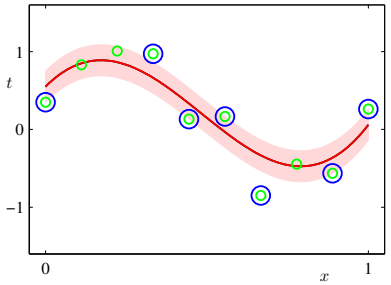
- Then, $a_n > 0$ if and only if $y(\boldsymbol{x}_n) + \epsilon + \xi_n - t_n = 0$, which implies that $\boldsymbol{x}_n$ lies on or above the upper margin of the ϵ-tube or above it. Similarly for $\widehat{a}_n > 0$.

Support Vector Machines for Regression

- 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.



- 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,

$$b = t_n - \epsilon - \sum_{m \in \mathcal{S}} (a_m - \widehat{a}_m) k(\boldsymbol{x}_n, \boldsymbol{x}_m)$$

Summary

- Kernel trick: It allows to work in the feature space without constructing it.
- Quadratic objective function: It allows to obtain the global optimum for a given kernel and C/ϵ (which are obtained by cross-validation).
- Sparse model: Only the support vectors are needed for classification/regression (compare with kernel models).

732A95 Introduction to Machine Learning

Lecture 6a: Neural Networks

Jose M. Peña
IDA, Linköping University, Sweden

Contents

Literature

- Main source
  - Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006. Sections 5.1-5.3.3 and 5.5.3.
- Additional source
  - Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning*. Springer, 2009. Chapter 11.

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

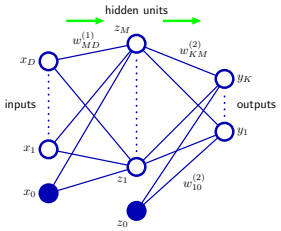
$$y(\boldsymbol{x}) = \text{sgn}\Big(\sum_{m\in\mathcal{S}} a_m t_m k(\boldsymbol{x}, \boldsymbol{x}_m) + b\Big)$$

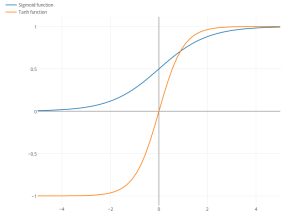
- Consider regressing an unidimensional continuous random variable on a D-dimensional continuous random variable. Consider a training set $\{(\boldsymbol{x}_n, t_n)\}$
- For a new point $\boldsymbol{x}$, SVMs predict

$$y(\boldsymbol{x}) = \sum_{m\in\mathcal{S}} (a_n - \widehat{a}_n) k(\boldsymbol{x}, \boldsymbol{x}_m) + b$$

- SVMs imply data-selected user-defined basis functions.
- NNs imply a user-defined number of data-selected basis functions.

Neural Networks





- Activations: $a_j = \sum_i w^{(1)}_{ji} x_i + w^{(1)}_{j0}$
- Hidden units and activation function: $z_j = h(a_j)$
- Output activations: $a_k = \sum_j w^{(2)}_{kj} z_j + w^{(2)}_{k0}$
- 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:

$$y_k(\boldsymbol{x}) = \sigma\Big(\sum_j w^{(2)}_{kj} h\Big(\sum_i w^{(1)}_{ji} x_i + w^{(1)}_{j0}\Big) + w^{(2)}_{k0}\Big)$$

- 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.

Neural Networks

- For a large variety of activation functions, the two-layer NN can uniformly approximate any continuous function to arbitrary accuracy provided enough hidden units. Easy to fit the parameters ? Overfitting ?!

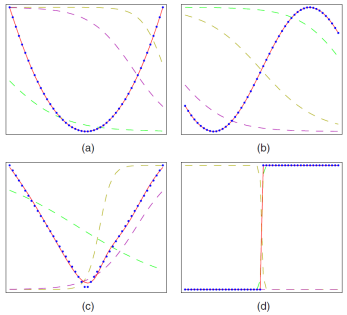


Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

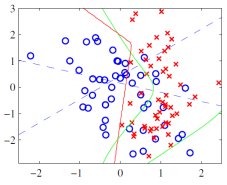
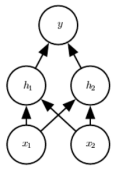
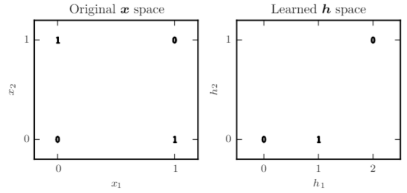


Figure 5.4 Example of the solution of a simple two-class classification problem involving synthetic data using a neural network having two inputs, two hidden units with 'tanh' activation functions, and a single output having a logistic sigmoid activation function. The dashed blue lines show the $z = 0.5$ contours for each of the hidden units, and the red line shows the $y = 0.5$ decision surface for the network. For comparison, the green line denotes the optimal decision boundary computed from the distributions used to generate the data.

Neural Networks

- Solving the XOR problem with NNs.
- No line shatters the points in the original space.
- The NN represents a mapping of the input space to an alternative space where a line can shatter the points. Note that the points (0,1) and (1,0) are mapped both to the point (1,0).
- It resembles SVMs.





$w^{(1)}_{11} = w^{(1)}_{12} = w^{(1)}_{21} = w^{(1)}_{22} = 1$
$w^{(1)}_{10} = 0,\ w^{(1)}_{20} = -1$
$h_j = z_j = h(a_j) = \max\{0, a_j\}$
$w^{(2)}_{11} = 1,\ w^{(2)}_{12} = -2$
$w^{(2)}_{10} = 0$
$y = y_k = a_k$

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 error function

$$E(\boldsymbol{w}^t) = \sum_n E_n(\boldsymbol{w}^t) = \sum_n \frac{1}{2}\|\boldsymbol{y}(\boldsymbol{x}_n) - \boldsymbol{t}_n\|^2 = \sum_n \sum_k \frac{1}{2}(y_k(\boldsymbol{x}_n) - t_{nk})^2$$

- The weight space is highly multimodal and, thus, we have to resort to approximate iterative methods to minimize the previous expression.
- Batch gradient descent

$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_n \nabla E(\boldsymbol{w}^t)$$

where $\eta_n > 0$ is the learning rate ($\sum_n \eta_n = \infty$ and $\sum_n \eta_n^2 < \infty$ to ensure convergence, e.g. $\eta_n = 1/n$).

- Sequential, stochastic or online gradient descent

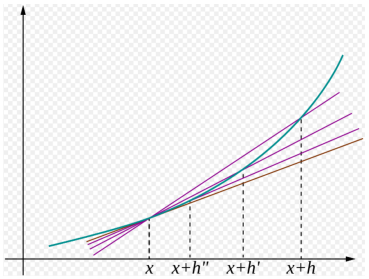
$$\boldsymbol{w}^{t+1} = \boldsymbol{w}^t - \eta_n \nabla E_n(\boldsymbol{w}^t)$$

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.

Backpropagation Algorithm

- Recall that $f'(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$



- Recall that $\nabla E_n(\boldsymbol{w}^t)$ is a vector whose components are the partial derivatives of $E_n(\boldsymbol{w}^t)$.

Backpropagation Algorithm

- 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, then

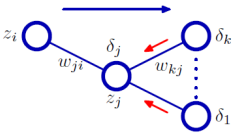
$$\delta_k = \frac{\partial E_n}{\partial a_k} = y_k - t_k$$

- 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.

Backpropagation Algorithm

- 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.

Figure 5.7 Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

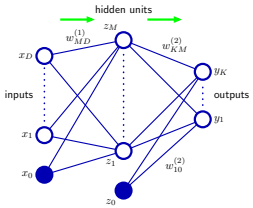


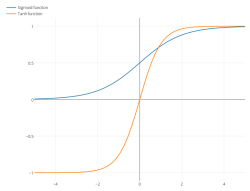
- Since $y_k = \sigma(a_k)$ for classification, then

$$\delta_k = \frac{\partial E_n}{\partial a_k} = \sigma(a_k)(1 - \sigma(a_k))$$

- This is an example of embarrassingly parallel algorithm.

Backpropagation Algorithm





- 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
  $$\frac{\partial E_n}{\partial w_{kj}} = \delta_k z_j \text{ and } \frac{\partial E_n}{\partial w_{ji}} = \delta_j x_i$$

Backpropagation Algorithm

- 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

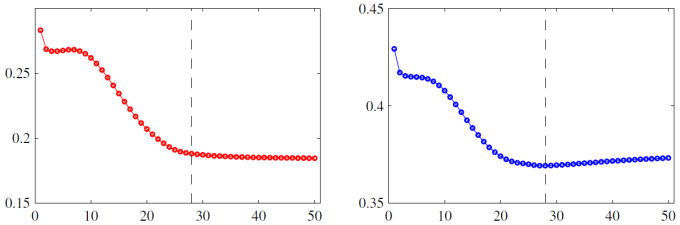


Figure 5.12 An illustration of the behaviour of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set. The goal of achieving the best generalization performance suggests that training should be stopped 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
  $$E(\boldsymbol{w}) + \frac{\lambda_1}{2}||\boldsymbol{w}^{(1)}||^2 + \frac{\lambda_2}{2}||\boldsymbol{w}^{(2)}||^2$$
  instead of the classical
  $$E(\boldsymbol{w}) + \frac{\lambda}{2}||\boldsymbol{w}||^2$$
  and choose λ_1 and λ_2 by cross-validation.

Regularization

- To see why, let $z_j = h(\sum_i w_{ji} x_i + w_{j0})$ and $y_k = \sum_j w_{kj} z_j + w_{k0}$.
- Consider the linear transformation $x_i \to a x_i + b$.
- Transform $w_{j0} \to w_{j0} - \frac{b}{a}\sum_i w_{ji}$ and $w_{ji} \to \frac{1}{a} w_{ji}$.
- Both NNs define the same mapping, but they may receive different penalty for complexity $||\boldsymbol{w}||^2$.
- Solution: Penalize according to $\frac{\lambda_1}{2}||\boldsymbol{w}^{(1)}||^2 + \frac{\lambda_2}{2}||\boldsymbol{w}^{(2)}||^2$ and let $\lambda_1 \to a^{1/2}\lambda_1$.
- Finally, note that the effect of the penalty is simply to add $\lambda_1 w_{ji}$ and $\lambda_2 w_{kj}$ to the appropriate derivatives.

Summary

- NNs: Nonlinear mapping from input to output.
- Extremely expressive.
- Training: Backpropagation algorithm, and regularization.

732A95 Introduction to Machine Learning

Lecture 6b: Deep Learning

Jose M. Peña
IDA, Linköping University, Sweden

Contents

Literature

- Main sources
  - Bengio, Y. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning*, 2:1-127, 2009. Chapters 1-3, 4.2, 4.5-4.6, 6.2-6.3.
  - Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2006. Section 5.5.6.
  - LeCun, Y., Bengio, Y. and Hinton, G. Deep Learning. *Nature*, 521:436-44, 2015.
- Additional source
  - Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. Book in preparation for MIT Press, 2016. Available at `www.deeplearningbook.org`. Chapters 1, 6.

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 $C^{(k)}$ contain all classifiers defined by NNs of one hidden layer with k hidden units and the sigmoid activation function. Then, for any distribution $p(\mathbf{x}, t)$,

$$\lim_{k \to \infty} \inf_{y \in C^{(k)}} E_X[y(\mathbf{x})] - E_X[p(t|\mathbf{x})] = 0$$

- 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.

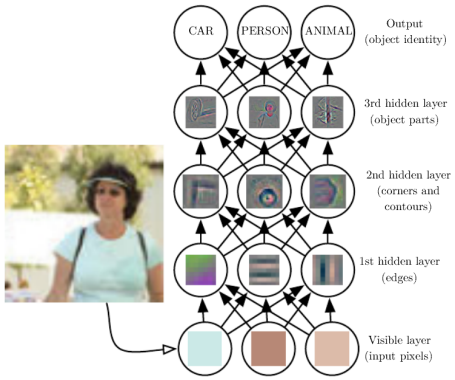
Limitations of Neural Networks

- 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.

Deep Neural Networks

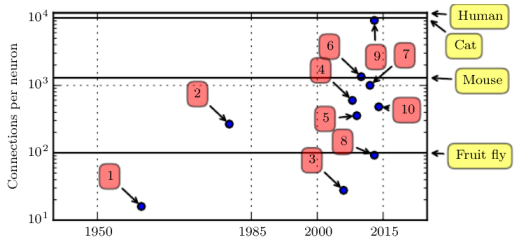


Figure 1.10: Initially, the number of connections between neurons in artificial neural networks was limited by hardware capabilities. Today, the number of connections between neurons is mostly a design consideration. Some artificial neural networks have nearly as many connections per neuron as a cat, and it is quite common for other neural networks to have as many connections per neuron as smaller mammals like mice. Even the human brain does not have an exorbitant amount of connections per neuron. Biological neural network sizes from Wikipedia (2015).

1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett *et al.*, 2009)
6. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
7. Distributed autoencoder (Le *et al.*, 2012)
8. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
9. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
10. GoogLeNet (Szegedy *et al.*, 2014a)

Deep Neural Networks

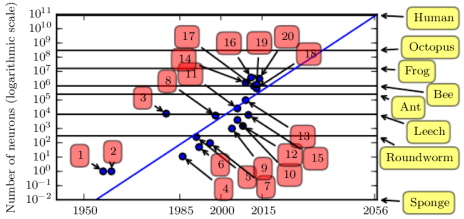


Figure 1.11: Since the introduction of hidden units, artificial neural networks have doubled in size roughly every 2.4 years. Biological neural network sizes from Wikipedia (2015).

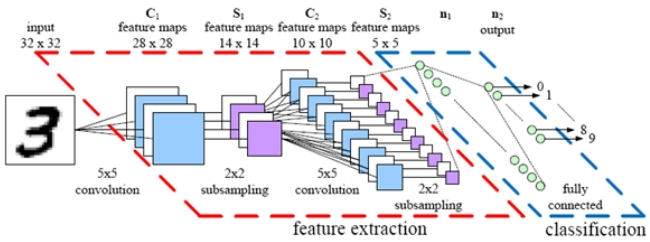
1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart *et al.*, 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio *et al.*, 1991)
7. Mean field sigmoid belief network (Saul *et al.*, 1996)
8. LeNet-5 (LeCun *et al.*, 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton *et al.*, 2006)
11. GPU-accelerated convolutional network (Chellapilla *et al.*, 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina *et al.*, 2009)
14. Unsupervised convolutional network (Jarrett *et al.*, 2009)
15. GPU-accelerated multilayer perceptron (Ciresan *et al.*, 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le *et al.*, 2012)
18. Multi-GPU convolutional network (Krizhevsky *et al.*, 2012)
19. COTS HPC unsupervised convolutional network (Coates *et al.*, 2013)
20. GoogLeNet (Szegedy *et al.*, 2014a)

22 layers DNN, but
12 times fewer weights
than DNN 19

Deep Neural Networks

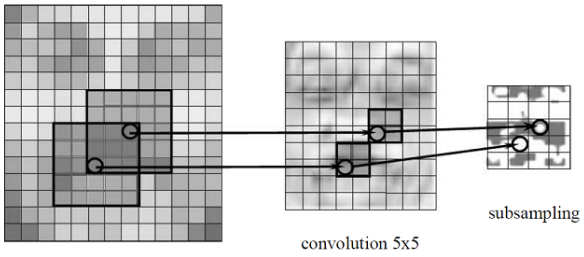
- Training DNNs is difficult:
  - Typically, poorer generalization than (shallow) NNs.
  - The gradient may vanish/explode as we move away from the output layer, due to multiplying small/big quantities. E.g. the gradient of σ and tanh is in $[0, 1)$. So, they may only suffer the gradient vanishing problem. 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".

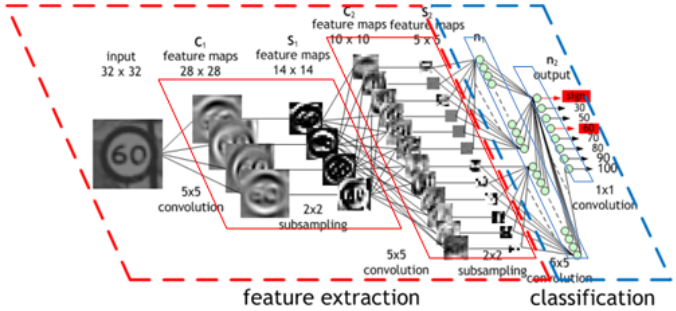
Convolutional Networks



- 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.

Convolutional Networks





Convolutional Networks

- 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

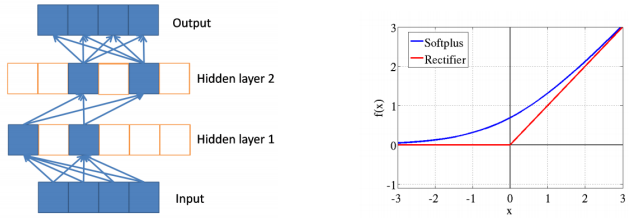


Figure 2: *Left:* **Sparse propagation of activations and gradients in a network of rectifier units.** The input selects a subset of active neurons and computation is linear in this subset. *Right:* **Rectifier and softplus activation functions.** The second one is a smooth version of the first.

- $rectifier(x) = \max\{0, x\}$, 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.

Rectifier Activation Function

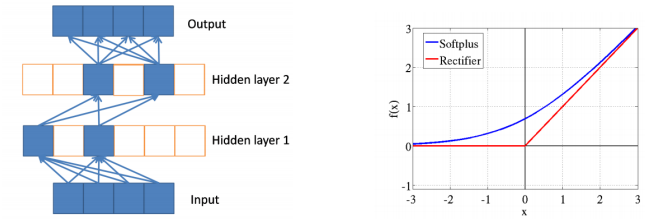


Figure 2: *Left:* **Sparse propagation of activations and gradients in a network of rectifier units.** The input selects a subset of active neurons and computation is linear in this subset. *Right:* **Rectifier and softplus activation functions.** The second one is a smooth version of the first.

- 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.
- 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.

Layer-Wise Pre-Training

- Supervised version:
  1. Train each layer of the DNN as if it was the hidden layer in a depth-two NN. As input, use the output of the last of the previously trained layers.
  2. Run the backpropagation algorithm to fine-tune the weights.
- The pre-training aims to find a good starting point for the subsequent run of the backpropagation algorithm.
- Unsupervised version: Similar to the supervised one but the hidden layers (except the last one) are trained to learn an encoding of the output of the previous layer, instead of the original classification or regression function.

Summary

- 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.

TDDE01 – Machine Learning
Group 9 Laboration Report 1

Martin Estgren <mares480>
Björn Jansson <bjoja408>
Erik S. V. Jansson <erija578>
Sebastian Maghsoudi <sebma654>

Linköping University (LiU), Sweden

November 9, 2016

Assignment 1

Nobody likes *e-mail spam*, therefore methods for autonomously *predicting* if a given e-mail is probably *spam* or *not spam* is an important task. This is a classic example where *machine learning* is useful; given a set of *training data* and *testing data*, can we predict what is *spam* and *not spam* in the *testing set* (without knowing the answer) by deriving a *hypothesis function* built from the *training data*?

By using *k-nearest neighbor classification*, one can derive if an e-mail is spam or not by simply looking at *similar e-mails/messages*, and picking the most likely solution by doing a *"majority vote"*. First, a *distance function* needs to be implemented, which is the *cosine distance function* in Equation 1, whose implementation can be found in Listing 2, but with a optimized solution using only matrices.

$$d(X,Y) = 1 - \frac{X^T Y}{\sqrt{\sum_i X_i^2}\sqrt{\sum_i Y_i^2}} \qquad (1)$$

After defining the distance function $d(X,Y)$, one can find the *e-mail/message distance* for each Y_j in respect to each X_i. Where X is the *testing set* and Y the *training set*. Each row of the resulting matrix contains the relative distance between X_i and $\forall Y_j$. Therefore, sorting each row X_i and picking the first K elements gives the K *closest messages* from the *training set* in respect to each *testing element*. By using this, the *k-nearest neighbors* can be found, and the prediction of $\hat{Y}$ (spam, not spam) is done by using Equation 2, where K_i classify as being C_i.

$$\hat{Y} = \max_{\forall C_i} p(C_i|\boldsymbol{x}),\ p(C_i|\boldsymbol{x}) \propto K_i \div K \qquad (2)$$

The *k-nearest neighbor algorithm* is implemented in Listing 1, in the function `knearest(t,k,t')`. It works as previously described, where line `20` is calculating the *distance matrix* and line `21` sorting each row, so that all Y distances are relative to X_i. Thereafter, in line `26-27` the classification is found for the K-nearest neighbors of X_i. The *mean* value is then taken, which is equivalent to $K_i \div K$ since only two classifications exist (spam and not spam), following a *Cover et al.* [CH67] K-NN descriptions.

From the *knearest* function we get a vector containing all the predicted classifications for the data set. First we set a probability threshold $p(\mathbf{x}|\theta) > 0.5$ and compare the predicted classifications with the true classifications in the data set. Where the threshholds are done in lines `49-50` in the main spam classification script found under a Listing 3.

The result for $k = 5$, $k = 1$ can be observed in the following tables, for both *kknn* and *knearest.r*:

```
          knearest, k = 5
observations   0   1
           0 682 263
           1 186 239
misclassification rate = 0.3277372
```

These results indicate a "hit ratio" of around 68%.

```
            knearest, k = 1
observations   0   1
           0 650 295
           1 174 251
misclassification rate = 0.3423358
```

These results indicate a "hit ratio" of around 66% for the results when *knearest* has $k = 1$ neighbors.

The result indicates that our classifier is fairly good for the given data set but to get a better perspective we compare it to a built in classifier called *Weighted K-nearest neighbor* (kknn) on the same training and testing data set. The result can be observed bellow, for both $k = 1$ and $k = 5$ values:

```
            kknn, k = 5, k = 1
observations   0   1
           0 626 319
           1 184 241
misclassification rate = 0.3671533
```

The biggest difference from out own implementation is that the hit rate and confusion matrix are identical for both $K = 1$ and $K = 5$. The hit rate is also slightly lower that ours at about 0.63.

We then proceed to plot the classifiers in a *ROC - curve* for the probability threshold $p(x) > 0.05, ..., 0.95$ with steps of 0.01. Where the *x-axis* shows the *sensitivity/true positive rate* and the *y-axis* shows the *false negative rate*. The *true positive rate* is calculated as

$$\frac{Tp}{Tp + Fn} \tag{3}$$

where Tp stands for the *true positives* i.e. the cases where both the prediction and observations classified the case as *true*, and Fn stands for the *false negatives* i.e. the cases where the prediction was *false* but the observation was *true*.

The *true negative rate* is calculated as $1 - specificity$ where *specificity* is equal to:

$$\frac{Tn}{Tn + Fp} \tag{4}$$

where Tn stands for the *true negatives* i.e. the cases where both the prediction and observations classified the case as *false*, and Fp stands for the *false positives* i.e. the cases where the prediction was *true* but the observation was *false*.

These operations can be found in the lines `95-97` for *knearest* and `109-111` for *kknn* in Listing 3 `spam.r`.

The resulting *ROC - curve* can be observed in the column to the right, that is in the Figure 1.

Blue curve is *knearest* and green curve is *kknn*. The red line is a reference line for a random classifier. The result indicates as with the confusion matrices that our *knearest* implementation slightly out performs *kknn* on some threshold variables.

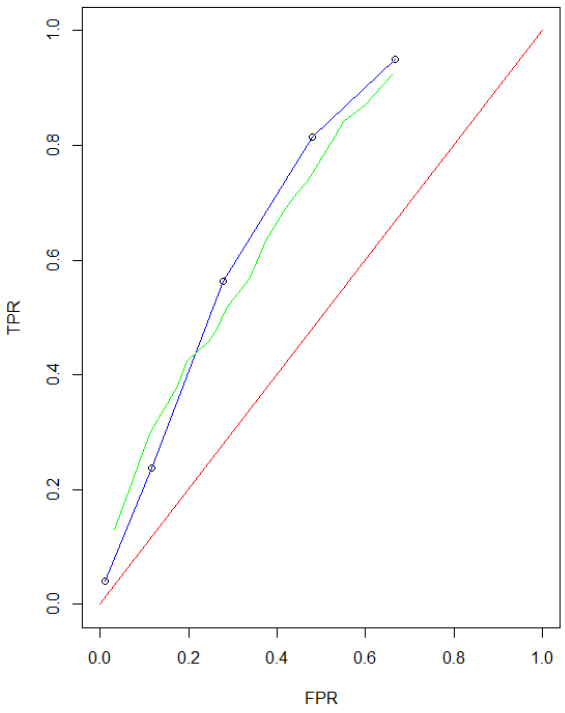


Figure 1: ROC - curve of *knearest* and *kknn*.

Assignment 2

The first part of the assignment was completed by importing the data from a csv-file and assuming that the probability is exponentially distributed. Because $L(\theta|x) = p(\theta|x)$ [1], one can derive the log-likelihood in the following steps:

$$L(\theta|x) = \prod_{i=0}^{n} p(x_i|\theta)$$
$$= \prod_{i=0}^{n} \theta e^{-\theta x} = \theta^n exp(-\theta \sum_{i=0}^{n} x_i)$$
$$\implies lnL(\theta|x)) = nln(\theta) - \theta \sum_{i=0}^{n} x_i$$

Having this formulae, the sum of observations is constant as well as the length (n) of the data. Thus, one can simply plot the return value of the function for all reasonable values (0.05-4) for θ. In order to use the maximum-likelihood method to estimate θ, one simply picks the theta that is most likely to produce the observed data i.e. the θ that produced the highest value of $ln(L(\theta|x))$. In order to investigate how the size of the observed data affect the estimation of θ, one might simply use less of the data.

The posterior probability[2] is also called the reverse probability i.e. $p(\theta|x)$ meaning the probability for θ given observed data x.This can be calculated in this context with $p(x|\theta)p(\theta)$ where $p(\theta)$ is given in the assignment description as $10e^{-10\theta}$. Also here the ln version of the functions may be used since the functionality is already defined earlier. This would not change the estimation of the optimal θ since the maximum value for ln(f(x)) and f(x) dosn't change [1]. In order to use already defined functions, one can make the following derivations:

$$ln(p(x|\theta)p(\theta)) = lnL\theta|x)p(\theta))$$
$$= lnL) + ln(p(\theta))$$

In order to find the most optimal estimation of θ one simply elects the theta that produces the highest value of said function.

Given that we use all the data and apply said data on said log-likelihood function previously described, the output is persented in figure 2. The maximum likelihood method gives us an estimation of $\theta = 1.1262$. When using less of the data we produce another value $\theta = 1.7857$ however as can be seen in figure 2, on the blue curve most values are not significantly different from the maximum value. This means one can not be certain when estimating θ since other options are also viable. The black curve in figure 2, the estimated θ

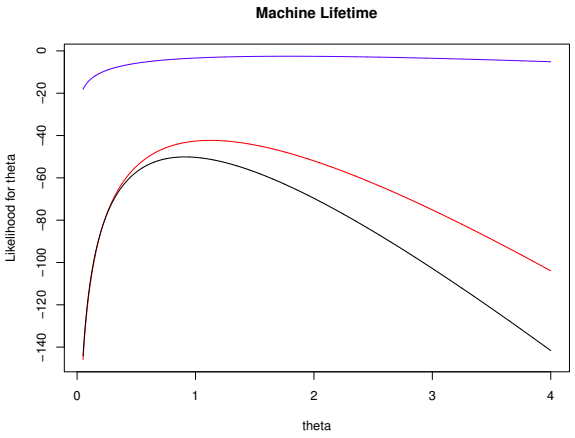


Figure 2: Log-likelihood & postiori curves.

The presentation of the distribution of the observed data can be viewed in figure 3, in the left histogram. The right histogram however, a randomly generated data set of 50 values is generated. The data is not completely random, but generated using an exponential distribution given the *theta* estimated from the 48 given data values. This of course would imply that the data sets look similar when plotted in a histogram. However this does not necessarily imply that they are representative of the entire population since the posteroriprobability was neglected for this measure.

Contributions

- **Erik S. V. Jansson:** wrote the initial section in *Assignment 1* regarding on how the *k-nearest neighbor algorithm* works, and also provided the `knearest` scripts in Listings 1,2.
- **Martin Estgren:** wrote the sections on generating *confusion matrices* and the *missclassification ratio* of *knearest* and *kknn*. Additionally, on how to generate *ROC-curves* and the plot itself. Finally, he provided Listing 3 which wraps the script functions in assignment 1 up. Additionally, he also extended the formula description on *ROC-curves* for full completeness.

- **Bjorn Jansson:** Wrote the third section.
- **Sebastian Maghsoudi:** Wrote the first, second and fourth section.

References

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

[CH67] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

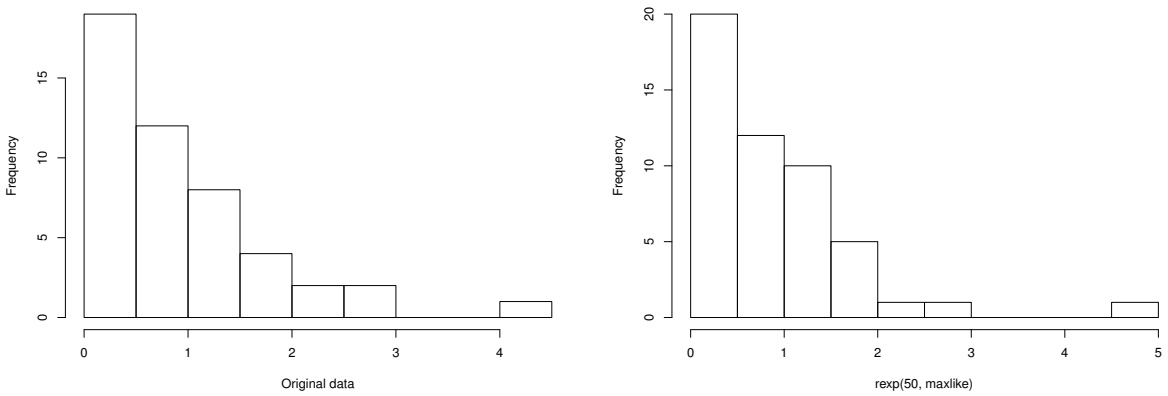


Figure 3: Histogram for original and generated data.

Appendix

Listing 1: K-Nearest Neighbor Algorithm Implementation

```
source("distance.r") # cos-distance d.

# spam(i, t) - gets spam vector in i.
spam <- function(indices, training) {
    spamid <- ncol(training) #  last.
    return(mean(training[indices, spamid]))
}

# knearest(t, k, t') - predicts values
# given in t', for training data in t.
# Done with using k-nearest neighbors.
knearest <- function(train, k, test) {
    # Don't include the 'Spam' column, not feature.
    test_features <- data.matrix(test[,-ncol(test)])
    train_features <- data.matrix(train[,-ncol(train)])

    # Compute the distance matrix between train and test
    # using the cosine distance formula (see distance.r)
    # which is then sorted, for picking the k-neighbors.
    distances <- distance(train_features, test_features)
    sorted_distance_ids <- as.matrix(t(apply(distances, 2, order))[,1:k])

    # Finally, retrieve if the training data is spam or
    # not, selecting the k-closest classifications, for
    # later determining the most likely classification.
    spamv <- apply(sorted_distance_ids, 1, spam, train)
    kspam_vector <- spamv # Select only first K.
    mean_spam <- data.matrix(kspam_vector)
    # Still need to classify data by e.g. >0.5 -> 1.
    return(mean_spam) # This step is done in spam.r.
}
```

Listing 2: Cosine Cost/Distance Formula

```
# distance(X, Y) - distances between X, Y.
# Uses the usual cosine distance function.
# Batch operation into a matrix -> fast...
distance <- function(matrix_x, matrix_y) {
    x_squared_sum <- rowSums(matrix_x^2)
    y_squared_sum <- rowSums(matrix_y^2)
    x_prime <- matrix_x / sqrt(x_squared_sum)
    y_prime <- matrix_y / sqrt(y_squared_sum)
    similarity_matrix <- x_prime %*% t(y_prime)
    distance_matrix <- 1.0 - similarity_matrix
    return(distance_matrix)
}
```

Listing 3: Main Spam Prediction Script

```
library(readxl)
library(kknn)

# Import data
```

```
spambase <- read_excel("spambase.xlsx")

# Get number of observations
n = dim(spambase)[1]

# Set psuedo random seed
set.seed(12345)

# Get half of the indexes
id = sample(1:n, floor(n * 0.5))

# Assign 50% of the observations as traning data
train = spambase[id, ]

# Assign 50% of the observations as test data
test = spambase[-id, ]

# Function to get the spam classification of a given index
spam_lookup <- function(indexes, lookup_table) {
  return (mean(lookup_table[indexes, ncol(lookup_table)]))
}

# Returns if a specific arrays elements are over a specific threshold
threshold <- function(threshold, data) {
  return (as.numeric(data > threshold))
}

#Function to calcuate the sensitivity of a data set
sensitivity <- function(observations, predictions) {

  result = sum((observations == 1 & predictions == 1)) / (sum((observations == 1 &predictions
    == 1)) + sum(observations == 1 & predictions == 0))
  return(result)

}

#Function to calcuate the specificity of a data set
specificity <- function(observations, predictions) {
  result = sum((observations == 0 &
    predictions == 0)) / (sum((observations == 0 &
    predictions == 0)) + sum(observations == 0 & predictions == 1))
  return(result)
}

observations = test$Spam
knearest_k5 = as.numeric(knearest(as.matrix(train), 5, as.matrix(test)) > 0.5)
knearest_k1 = as.numeric(knearest(as.matrix(train), 1, as.matrix(test)) > 0.5)

mst_k5 = table(observations, knearest_k5)
mst_k1 = table(observations, knearest_k1)
print(mst_k5)
print(mst_k1)

print(1 - sum(diag(mst_k5) / sum(mst_k5)))
print(1 - sum(diag(mst_k1) / sum(mst_k1)))

# Check with build-in functions
kknn_predictions_k5 = kknn(
  formula = Spam ~ .,
  train = train,
  test = test ,
  k = 5
```

```
)
kknn_predictions_k1 = kknn(
  formula = Spam ~ .,
  train = train,
  test = test,
  k = 1
)

kknn_predictions_k5 = as.numeric(fitted.values(kknn_predictions_k5) > 0.5)
kknn_predictions_k1 = as.numeric(fitted.values(kknn_predictions_k1) > 0.5)

print(table(observations, kknn_predictions_k5))
print(table(observations, kknn_predictions_k1))

print(mean(kknn_predictions_k5 != observations))
print(mean(kknn_predictions_k1 != observations))

# Check with other prediction thresholds

# Converts to matrix for apply operations
thresholds = seq(from = 0.05, to = 0.95, by = 0.05)
thresholds = matrix(thresholds, length(thresholds), 1)

# Get the knearest predictions of a data set
knearest_predictions = knearest(as.matrix(train), 5, as.matrix(test))

# Apply the threshold function to get the predictions
knearest_outcomes = t(apply(thresholds, 1, threshold, data = knearest_predictions))

# Calculate the specificity and the sensitivty of said predictions
knearest_sensitivity = apply(knearest_outcomes, 1, sensitivity, observations = observations)
knearest_specificity = apply(knearest_outcomes, 1, specificity, observations = observations)


# Calcualte the kknn predictions
kknn_predictions = kknn(
  formula = Spam ~ .,
  train = train,
  test = test ,
  k = 5
)
kknn_outcomes = t(apply(thresholds, 1, threshold, data = fitted.values(kknn_predictions)))

# Calculate the sensitivty and specificty of the kknn predictions
kknn_sensitivity = apply(kknn_outcomes, 1, sensitivity, observations = observations)
kknn_specificity = apply(kknn_outcomes, 1, specificity, observations = observations)

# Plot the values as ROC curves
plot(
  1 - knearest_specificity,
  knearest_sensitivity,
  xlim = c(0, 1),
  ylim = c(0, 1),
  xlab = "FPR",
  ylab = "TPR"
)
lines(1 - knearest_specificity, knearest_sensitivity, col = "Blue")
lines(1 - kknn_specificity, kknn_sensitivity, col = "Green")
lines(c(0, 1), c(0, 1), col = "Red")
```

TDDE01 – Machine Learning
Group 9 Laboration Report 2

Martin Estgren <mares480>
Erik S. V. Jansson <erija578>
Sebastian Maghsoudi <sebma654>

Linköping University (LiU), Sweden

November 16, 2016

Assignment 1

In this assignment we were tasked with finding the best *feature selection* using *k-fold cross-validation* with *linear regression* as estimator.

Feature Selection

The purpose of this algorithm is to find the subset of features from a given dataset that yields the lowest *cross-validation score*. This is performed by iterating through all possible combinations of features, in this case, *brute forcing* through all possibilities.

$$s \in \{1, ..., n\} \tag{1}$$

$$\mathbf{A} = \{s_1, s_2, ..., s_{2^n-1}\} \tag{2}$$

$$\bigcap_{i=1}^{|\mathbf{A}|} (s_i) = \emptyset \tag{3}$$

K-Fold Cross-Validation

The algorithm (as can be seen in 1) splits the given dataset into K disjoint subsets of equal size, K is arbitrarily defined as 5 for this assignment. The algorithm then iterates through all the subsets (folds) using the current fold as validation data and all the other folds as training data for the linear regression model.

Algorithm 1 K-Fold Cross-Validation (Linear $\mathcal{M}$)

Require: feature matrix $X_\mathcal{F}$ and target vector $\mathbf{y}_\mathcal{F}$, given a feature selection $\mathcal{F}$ with cardinality $|\mathcal{F}|$.

1: $(X_i, \mathbf{y}_i) \leftarrow \text{split}(X_\mathcal{F}, \mathbf{y}_\mathcal{F}, k)$ {Equally $|X_\mathcal{F}| \div k$}
2: **for** $i \leftarrow 1$ **to** k **do** {Attempts every of k-folds}
3: $\quad X_t \leftarrow X_1 \cup \cdots \cup X_k - X_i$ {Except fold i}
4: $\quad \mathbf{y}_t \leftarrow \mathbf{y}_1 \cup \cdots \cup \mathbf{y}_k - \mathbf{y}_i$ {Except fold i}
5: $\quad \hat{\mathbf{w}}_t \leftarrow (X_t^\intercal X_t)^{-1} X_t^\intercal \mathbf{y}_t$ {Train model}
6: $\quad \hat{\mathbf{y}}_i \leftarrow X_i \hat{\mathbf{w}}_t$ {Predict target vector}
7: $\quad \hat{\varepsilon}_i(\hat{\mathbf{y}}_i, \mathbf{y}_i) \leftarrow \frac{1}{n} \sum_{j=1}^{n} (\hat{y}_j - y_j)^2$
8: **end for**
9: **return** $(\sum_{i=1}^{k} \hat{\varepsilon}_i(\hat{\mathbf{y}}_i, \mathbf{y}_i)) \div k$

Linear Regression

In our implementation we use the *ordinary least squares* estimator as regression model. *sum of square error* (SSE) is used as metric for error rate, which can be calculated with $\sum_i (\hat{y}_i - y_i)^2$.

Ordinary Least Squares

$\hat{\mathbf{w}} = (X^T X)^{-1} X^T Y$, where X is a predictor matrix and Y is the response vector and $\hat{\mathbf{w}}$ the estimated parameters for all predictors.

This is used when predicting the response vector $\hat{\mathbf{y}} = X\hat{\mathbf{w}}$, which is used in Algorithm 1 in line 6.

Analysis

Given the previously described methods, one can now apply these to any dataset. Here, the *swiss* set is used with *Fertility* as the target variable and

the other five features as predictors. We calculate the mean error rate for each number of features in the combination, the results can be observed in the following graph below.

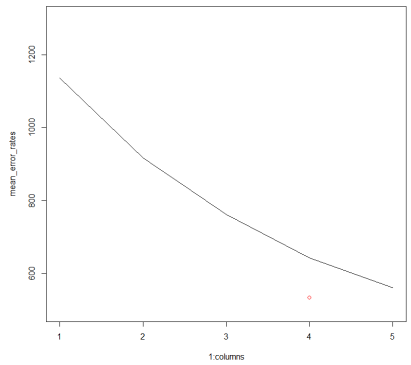
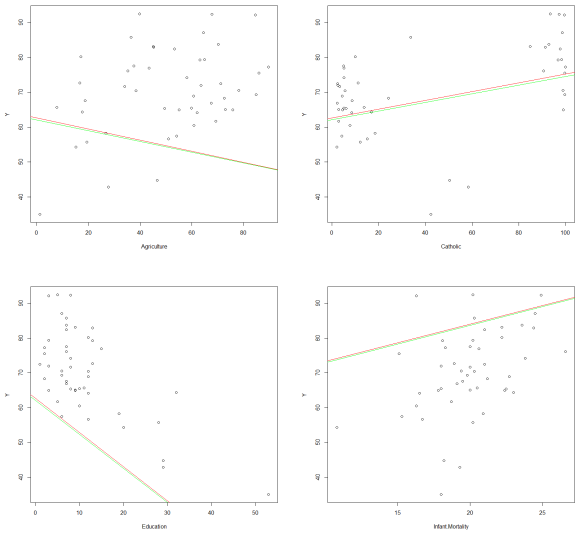


Figure 1: The mean error rate for different number of feature combinations.

We can observe that the error rate decreases with the number of features selected, and the best possible chosen feature set contains: *Agriculture, Education, Catholic, Infant Mortality*. For the next set of graphs shown below portrays the linear estimation of the features in the best feature subset. The model manages to approximate what humans would consider as the correct estimation.



Assignment 2

It turns out moisture in meat might be correlated to the concentration of protein. The data given in the assignment yields a plot which has points densely located around a line. Meaning, it is rather clear that said statement is true since the points seem distributed linearly. The plot for this can be found in the Appendix. Assuming there is some linear model M that fits the data sufficiently well, one can might suspect that there is a model of higher polynomial order that fits it even better. The model can be described in the following manner: $M_i \sim N(\sum_{j=0}^{i} w_j x^i, \sigma^2)$. The assignment specifies that observations above polynomial order 6 is completely inadvisable in this context (only $M_{1 \text{ to } 6}$).

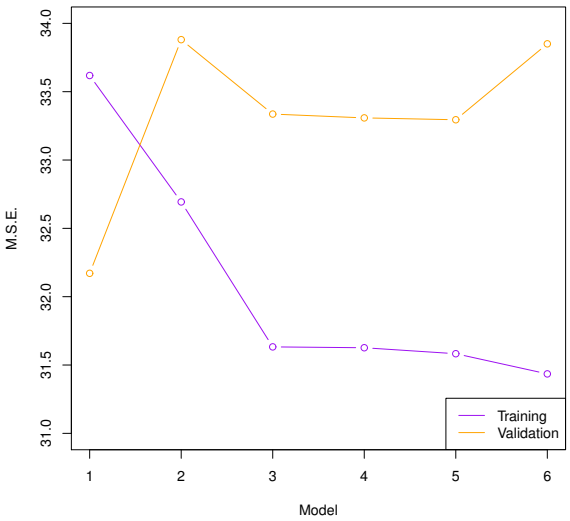


Figure 2: The mean error rate for different number of feature combinations.

From the plot in figure 2 the reader can observe that increasing the polynomial order leads to overfitting. This event is caused because though MSE for the training set decreases, the test set MSE is increasing. This means that any model M_i where $i > 1$, probably leads to worse predictions for the rest of the observation (not part of training data).

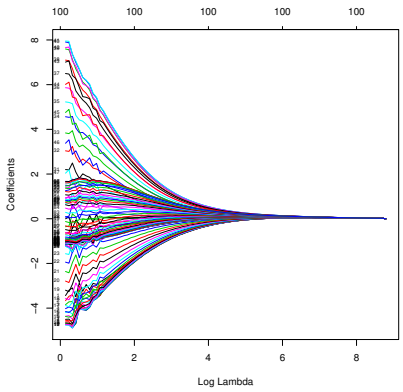
The following explanations concerns how the input variables for Channels 1 - 100 correlates

to Fat contents by doing variable selection using `stepAIC()`, which is a built-in *R* function for doing this. `stepAIC()` works by constructing a model from removing and adding features to it. It starts the algorithm with the entire feature set, then removes one feature depending on a quantified representation of the model that takes into consideration its prediction accuracy as well as its necessity. One can define the direction of the algorithm, meaning it will only remove variables if one defines the direction as *backwards*, and adding if it is defined as *forward*, but using the *both* keyword, they can be applied tougher. The *both* keyword will result in a "brute force" solution which should elect the optimal set of features. By applying `stepAIC` to the dataset, where the feature set are *Channel 1 - 100* and the response *Fat*, the results were a staggering 64 features, where most were selected in the backwards part of the algorithm.

Ridge regression works by taking into consideration the size of the parameters, this results in less variance for parameters since there are less options to choose from. By adding a representation of the size and factorize it by a *penalty rate* (λ in this case), one can control the importance of the size of parameters, meaning by increasing λ, it is more desirable for the algorithm to select such small parameters. This is done by adding the term $\sum_i \mathbf{w}^2$ to the usual *OLS formula*. The library *glmnet* provides such functionality. By setting the parameter $\lambda = 0.0$ (in *glmnet()* that is), one can use *Ridge regression* on the existing dataset. This yields Figure 3, as astute reader can observe, the coefficients converge uniformly towards zero, which is of course implied from the previously described explanation of Ridge (and also more graphically from Figure 4).

LASSO works similarly to Ridge, except that it represents the size of the parameters differently. Rather than using $\sum_i \mathbf{w}^2$ is uses $\sum_i |\mathbf{w}|$ instead. See Figure 4. As can be seen, the edges are linear and there are four points where the distance from the origin is as large as possible, making at least one parameter zero. Because of this, LASSO converges individual parameters to zero faster than Ridge (instead of uniformly), this is because a tangent has to be parallel with the line from one extreme point to another (created from a feature), if it doesn't, the intersection will converge to an extreme point primarily. Thus, it excludes features in the model by assigning zero as a parameter. Practically, this

Figure 3: Ridge regression penalization.



can also be implemented by using *glmnet*, this time by setting $\lambda = 1.0$. Figure 5 confirms the previous assertions, the coefficients converge to zero not as uniformly as Ridge, in fact, they seem to converge individually (in an iterative fashion, after another).

Figure 4: LASSO on the left, Ridge on the right. Distributed under the CC-license, created by *Rezamohammadighazi* in LASSO on Wikipedia.

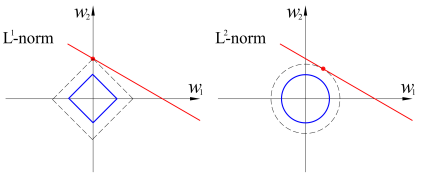
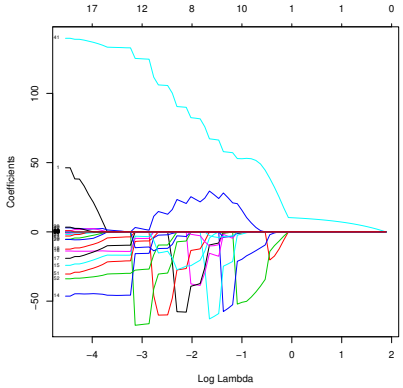


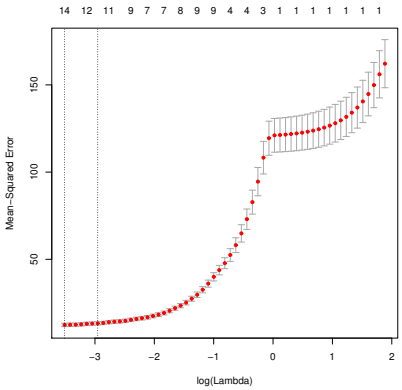
Figure 5: LASSO regression penalization.



By using the built in *k-fold cross-validation* al-

gorithm in *glmnet* for the *Lasso model*, the best feature selection can be found. How *k-fold cross-validation* work is explained back in Assignment 1. The best λ has been reported to be around 0.029 after running `cv.glmnet` for the same feature and response variables. Also, after plotting the *log penalty* against the *mean square error*, Figure 6 is produced. As can be seen, the marked interval displays the best set of selected features, with their respective penalty values. The algorithm has selected 13 features, quite a few less than *stepAIC*.

Figure 6: K-Fold Cross-Validation



The results produced from `stepAIC()` gave 64 selected features, while *k-fold cross-validation with the LASSO regression* produced only a set of 13 features. LASSO is more efficient when larger quantities of data are to be used when training a model, while `stepAIC()` will take longer to process but will in most cases result in a lower error rate value.

Contributions

The written text has been contributed by all members of the group, where it was re-written from scratch together. However, certain plots, scripts and also algorithms were taken from certain individual reports. *Erik* contributed with *Algorithm 1* and some of the plots for *Ridge* and *Lasso* while *Martin* contributed with most of the plots in Assignment 1 and also was chosen to provide the scripts for both assignments (to not have to piece together scripts, which was a hassle in past report). *Sebastian* provided explanations on *Ridge* & *Lasso*.

Appendix

Listing 1: Script for Assignment 1

```
library(readxl)

# folding indexes, returns start indexes for each fold
indexes <- function(n,k){
  s = floor(n/k)
  indexes = matrix(1,k,2)
  for(i in 1:k){
    indexes[i,1] = (i-1) * s
    indexes[i,2] = i * s -1
  }
  indexes = indexes +1
  indexes[k,2] = n
  return(indexes)
}

# Deprecated because of reasons
binary_permutations <- function(n){
  indexes = matrix(0,2^n-1,n)
  for(i in 1:2^n-1){
    indexes[i,] = as.numeric(intToBits(i))[1:n]
  }
  return(indexes)
}

k_fold <- function(X,Y,K){
  n = nrow(X)
  fold_errors = matrix(0,K,1)
  fold_weights = matrix(0,K,(ncol(X)+1))
  indexes = indexes(n,K)
  for( i in 1:K){
    test_fold_indexes = indexes[i,1]:indexes[i,2]
    train_fold_indexes = (1:n)[-test_fold_indexes]

    test_y = Y[test_fold_indexes]
    test_x = X[test_fold_indexes,]
    train_x = X[train_fold_indexes,]
    train_y = Y[train_fold_indexes]
    result = linear_regression(as.matrix(train_x), train_y,as.matrix(test_x),test_y)
    fold_errors[i,] = result$err
    fold_weights[i,] = result$param
  }

  return (list(weights=colMeans(fold_weights), err=fold_errors))
}
best_subset <- function(X,Y,K){
  n = nrow(X)
  columns = ncol(X)
  print(columns)

  errors = matrix(0,columns,2^columns)
  mean_error_rates = matrix(0,columns,1)
  best_error = Inf
  best_indexes = c()
  best_weights = c()
  for(n_features in 1:columns){
    binary_permutations = binary_permutations(ncol(X))
    binary_permutations = t(combn(1:columns,n_features))
```

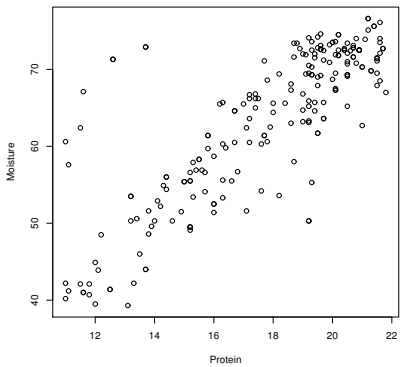
```
    n_combinations = nrow(binary_permutations)

    combination_errors = matrix(Inf,n_combinations,1)

    for(combination in 1:n_combinations){
      current_features = binary_permutations[combination,]
      filtered_x = as.matrix(X[current_features])
      k_fold = k_fold(filtered_x,Y,K)
      combination_errors[combination,] = mean(k_fold$err)
      best_combination_index = which.min(combination_errors)

      if(combination_errors[best_combination_index,] < best_error){
        best_weights = k_fold$weights
        best_indexes = binary_permutations[combination,]
        best_error = combination_errors[best_combination_index,]
      }

    }
    mean_error_rates[n_features,] = mean(combination_errors)

  }
  plot(1:columns,mean_error_rates, type = "l", ylim = c(500,1300))
  points(length(best_indexes),best_error, col="Red")
  print(best_error)
  return(list(indexes=best_indexes, weights=best_weights, err=best_error))
}

 # Linear regression between two samples, one as x-values and one as y-values
 linear_regression <- function(x_train,y_train,x_test,y_test){
    x_train = cbind(matrix(1,nrow(x_train),1),x_train)
    x_test = cbind(matrix(1,nrow(x_test),1),x_test)
    w = solve(t(x_train) %*% x_train) %*% t(x_train) %*% y_train
    w = as.vector(w)
    pY = x_test %*% w
    errors = sum((y_test - pY)^2)
   return(list(param=w, pred=pY, err=errors))
 }


data = swiss
set.seed(12345)
s = sample(1:nrow(data),nrow(data))
data = data[s,]
# ============
#fold_indexes(nrow(data),5)

X = data[,2:ncol(data)]
Y = as.matrix(data$Fertility)
best_features = best_subset(X,Y,5)
print(best_features)
X = X[,best_features$indexes]
print(X)
model = linear_regression(x_train = as.matrix(X),y_train = Y,x_test = as.matrix(X),y_test = Y)
for(i in 1:ncol(X)){
  plot(X[,i],Y, xlab = colnames(X)[i])
  # From best_features
  abline(best_features$weights[1],best_features$weights[i+1],col="Red")
  # From new regression
  abline(model$param[1],model$param[i+1],col="Green")

}
```

Listing 2: Script for Assignment 2

```
library(MASS)
library(readxl)
library(Matrix)
library(glmnet)

# task 1
# load data and show plot, do you think the data can be represented by a linear model?
data = read_excel("tecator.xlsx")
plot(data$Protein,data$Moisture)
# answer: yes

# task 2
# Find a probailist model explaining Mi. where M is a polynomial model of the protein up to
    power i
# Why is it important to use mean squared error whne fitting model?

# We approximate the model as w0 + w1x^1 + w2x^2 ...
# Use MSE instead of SSE because MSE is less dependent on

# task 3
# Divide the data into traning and test set 50/50.
# Fit models Mi, i = 1:6
#   Record MSE for traning and validation,
#   Show a plot of how traning and validation depends on i
set.seed(12345)
n = nrow(data)
train_indexes = sample(1:n,floor(n*0.5))
train_data = data[train_indexes,]
test_data = data[-train_indexes,]
power = 6
train_error = matrix(0,power,1)
test_error = matrix(0,power,1)

for(i in 1:power) {
  model = lm(Moisture ~ poly(Protein,i), data=train_data)
  train_predictions = predict(model,train_data)
  test_predictions = predict(model,test_data)

  train_error[i,] = mean((train_data$Moisture - train_predictions)^2)
  test_error[i,] = mean((test_data$Moisture - test_predictions)^2)
}

ylim = c(min(rbind(train_error,test_error)),max(rbind(train_error,test_error)))
plot(1:power,train_error, col="Green", ylim=ylim)
lines(1:power,train_error, col="Green")
points(1:power,test_error,col="Red")
lines(1:power,test_error, col="Red")

# Observation of the plot indicates that as we increase the polynomial level the functiuon
    becomes more fitted for the training data and results in increasingly worse fit for the
    test data

# task 4
# Use stepAIC to perform variable selection over a linear model with channel1-100 as
    predictors and Fat as response
#   Comment on how many predictors were selected
# Remove eveyrting except Fat and channels
# Perform feature slection and record number of features

model = lm(Fat ~ . - Protein - Moisture - Sample, data=data)
```

```
steps = stepAIC(model,direction="both", trace=FALSE)
coeff_aics = steps$coefficients
n_coeff_aics = length(coeff_aics)
print(n_coeff_aics)
# 64 columns were selected

# task 5
#  Fit a ridge regresion model to the same predictor/response varaibles
#  Present a plot on how the coefficients depend on the log of the penalty factor lamba.
#  Report how the coefficients change depending on lamda
data_y = data$Fat

data = data[,-which(colnames(data) == "Sample")]
data = data[,-which(colnames(data) == "Fat")]
data = data[,-which(colnames(data) == "Protein")]
data = as.matrix(data[,-which(colnames(data) == "Moisture")])

ridge = glmnet(x=data, y=data_y, alpha=0)
plot(ridge, xvar="lambda")

# task 6
# Do the same stuff but with LASSO, compare with ridge
lasso = glmnet(x=data, y=data_y, alpha=1)
plot(lasso, xvar="lambda")

# task 7
#   CV to find the optimal LASSO, report the optimal lambda and how many variables were chosen
     by the model '
#   make conculsions
#   show a plot of CV scores in comparasion to lambda

lasso_cv = cv.glmnet(data,data_y, alpha=1)
lasso_cv_lambda = min(lasso_cv$lambda)
n_lass_cv = sum(coef(lasso_cv) != 0)
print(lasso_cv_lambda)
print(n_lass_cv)
plot(lasso_cv)
# task 8
# compare result from 4 and 7
```

Figure 7: Moisture versus Protein



TDDE01 – Machine Learning
Individual Lab Report 3

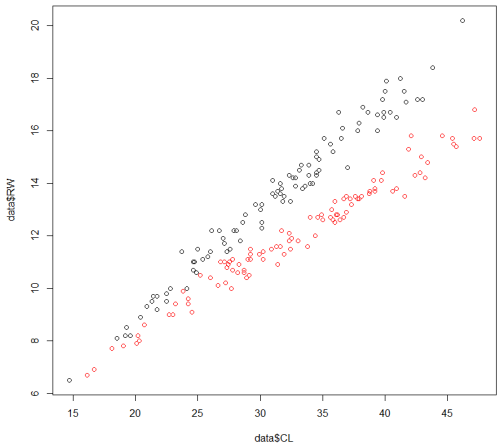
Martin Estgren <mares480>

9 december 2016

1 Assignment 1

This assignment involves creating an implementation of the *linear discriminant analysis* (LDA) with *maximum likelihood estimation* (MLE) as *discrimination function*. The LDA function will then be used to classify the *sex* of Chicago crabs using the *carapace length* and the *rear width* from the data set given for the assignment.

The first step involves visual inspection of the data set to determine if a *linear discrimination function* would be a good fit for the data set. The result of the *response variable sex* is ploted as the color of each point with the predictors *carapace length* and *rear width* as X and Y dimensions.



Figur 1: Raw plot of the data set.

Visual inspection of the data set indicates that a linear classification model would be suitable.

In order to implement the LDA function we start by implementing the MLA and return the weights for each classification category.

$$w_{1i} = -\frac{1}{2}\mu_i^T \Sigma^{-1} \mu_i + log\pi_i \tag{1}$$

where $i \in \{CL, RW\}$

$$b_{1i} = \Sigma^{-1}\mu_i \tag{2}$$

The classification function can be derived form these variables by combining them into:

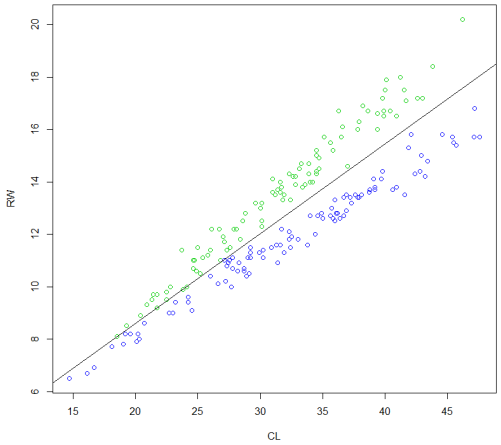
$$d_i(x) = X^T \Sigma^{-1}\mu_i - \frac{1}{2}\mu_i^T \Sigma^{-1}\mu_i + log\pi_i \tag{3}$$

In our case we have two categories for the classifier: $\{Male, Female\}$ which means that we need to combine the two different $d_i(x)$ functions with $d(x) = d_{Male}(x) - d_{Female}(x)$. We are in addition to classifying the data points interested in drawing the LDA line in the plot. In order to do this the intercept and slope of the LDA function is calculated from the $d(x)$ in the following way:

$$intercept = \frac{-b_1}{w_{1RW}} \tag{4}$$

$$slope = \frac{-w_{1CL}}{w_{1RW}} \tag{5}$$

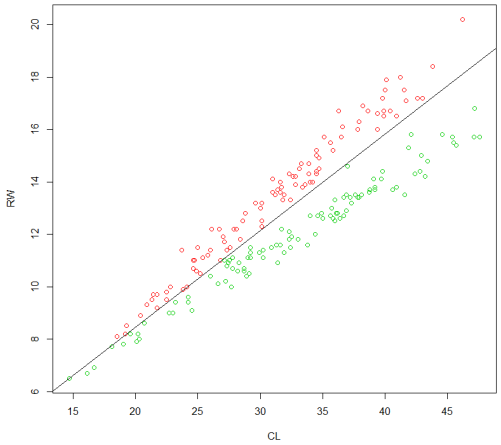
The resulting plot can be seen below



Figur 2: Plot of the raw data classified with LDA.

The LDA classifier managed to predict close to all of the observations correctly. As observed in above, the LDA managed to perform a perfect classification. The next task involves using the built in *logistic regression* classifier and examine its performance on the same data set.

Once again we extract the coefficients from the discriminant model and plot both the predicted result together with the *discrimination bound*.



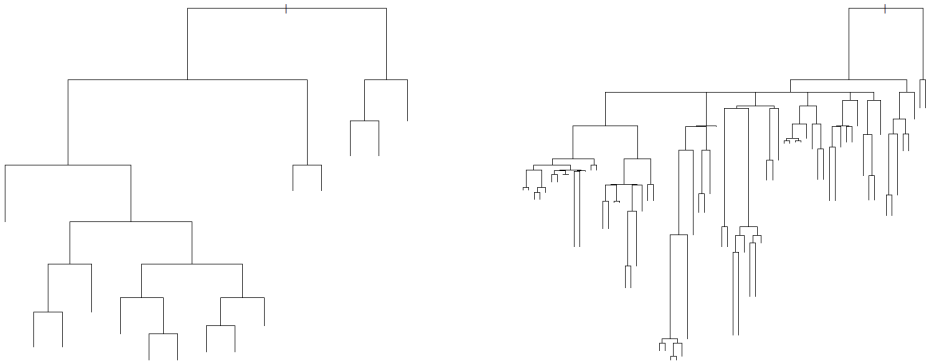
Figur 3: Plot of the raw data classified with logistic regression.

The *logistic regression* also managed a similar classification of the given data set with.

2 Assignment 2

In this assignment we are tasked with finding a classification model to find potential good customers who can manage their loans in a goodåay. To achieve this we use a data set with observations about how previous customers has manged their loans given a set of predictors. We start by splitting the observations into 50,25,25 percent parts and try fitting a decision tree model based on *deviance* and *gini index*. Below are the confusion matrices and misclassification rates for the testing and training observations.

Below the two different types of description trees can be observed followed by the confusion matrices and miss classification rate.

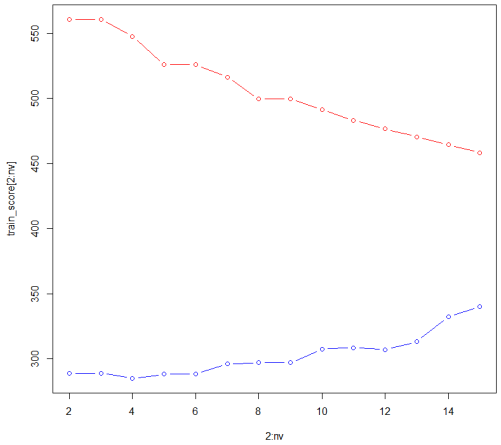


The *deviance* metric created a much less complex tree compared to the *gini index*. This is mirrored in the misclassification ate where the *deviance* creates the best result.

```
deviance model fitness
       predicted
       bad good
  bad   29   17
  good  45  159
misclassification rate: 0.248
```

```
gini model fitness
       predicted
       bad good
  bad   24   26
  good  50  150
misclassification rate: 0.304
```

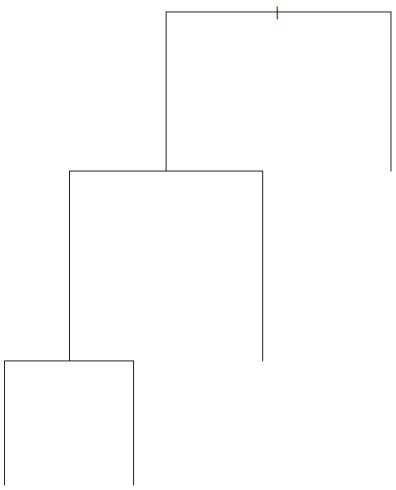
In order to determine the best max tree depth of the *deviance* model we iterates through all depths between 2 and max size of the model.



Figur 4: deviance for different max tree depths.

The *red* line indicates deviance of the training data and the *blue* line, the validation data. Tree depth of 4 results in the least deviance for the validation data.

```
optimal depth deviance fitness
       predicted
       bad good
  bad   20   59
  good   9  162
misclassification rate: 0.272
```



Figur 5: Optimal tree model.

The following confusion matrix represents the same data set used on a *naive bayesian* model instead of a decision tree.

```
model fitness of the training set
      predicted
      bad good
bad   95   98
good  52  255
misclassification rate: 0.3

model fitness of the testing set
      predicted
      bad good
bad   52   53
good  22  123
misclassification rate: 0.3
```

The misclassification rate is around the same values as the decision tree model. The training data predictions have about the same level of misclassification as the testing set but with a higher level of true positives.

We now apply a *loss matrix* to the predictions of the model and compare the result to the result above. The loss matrix can be observed below:

$$\begin{bmatrix} 0 & 1 \\ 10 & 0 \end{bmatrix} \tag{6}$$

The loss matrix will make *false positives* be classified much harsher compared to *false negatives*.

```
model fitness of the training set
      predicted
truth bad good
bad   137 263
good  10  90
misclassification rate: 0.546

model fitness of the testing set
      predicted
truth bad good
bad   66  134
good  8   42
misclassification rate: 0.568
```

Applying the loss matrix resulted in a higher misclassification rate with the number of false positives drastically reduced at the cost of false negatives increasing. The reason for this is that we consider false positives to be much worse compared the false negatives. In the group report we flipped the loss matrix which resulted in a lower misclassification score, which was incorrect.

3 Appendix: A - Code assignment 1

Listing 1: Code for assignment 1

```
library(readxl)

data = read.csv("australian-crabs.csv")
set.seed(12345)

plot(data$CL,data$RW, col=data$sex)
# Is the data easy to calssify be linear
   discriminant analysis?
# : Yes, very easy
```

```
X = cbind(data$CL,data$RW)
Y = data$sex
#ASSIGNMENT 1.2

disc_fun=function(label, S){
  X1=X[Y==label,]
  #MISSING: compute LDA parameters w1 (vector with 2
      values) and w0 (denoted here as b1)
  estimated_prob =  nrow(X1) / nrow(X)
  estimated_mean = colMeans(X1)
  b1 = -0.5*t(estimated_mean)%*%solve(S)%*%estimated
     _mean+log(estimated_prob)
  w1 = solve(S)%*%estimated_mean
    return(c(w1[1],w1[2],b1[1,1]))
}

X1=X[Y=="Male",]
X2=X[Y=="Female",]

S=cov(X1)*dim(X1)[1]+cov(X2)*dim(X2)[1]
S=S/dim(X)[1]

#discriminant function coefficients
res1=disc_fun("Male",S)
res2=disc_fun("Female",S)
print(res1)
print(res2)
# MISSING: use these to derive  decision boundary
   coefficients 'res'
res = res1-res2
intercept = -res[3] / res[2]
slope = -res[1]/res[2]
print(intercept)
print(slope)
# classification
d=res[1]*X[,1]+res[2]*X[,2]+res[3]
Yfit=(d>0)
plot(X[,1], X[,2], col=Yfit+3, xlab="CL", ylab="RW")
#MISSING: use 'res' to plot decision boundary.
abline(intercept,slope)
```

```
model = glm( sex ~ CL + RW, data = data, family = "
   binomial")
res_log = coefficients(model)
d_log = res_log[2]*X[,1]+res_log[3]*X[,2]+res_log[1]
Yfit_log=(d_log>0)
plot(X[,1], X[,2], col=Yfit_log+2, xlab="CL", ylab="
   RW")
print(res_log)
intercept_log = -res_log[1]/res_log[3]
slope_log = -res_log[2]/res_log[3]
abline(intercept_log,slope_log)
```

4 Appendix: B - Code assignment 2

Listing 2: Code for assignment2

```
library(readxl)
library(tree)
library(e1071)
library(rpart)

data = read.csv("creditscoring.csv")
#data$good_bad = as.characterdata$good_bad == "good"
n = nrow(data)
set.seed(12345)

indexes = sample(1:n,n)
end_traning = floor(n*0.5)
end_validation = end_traning + floor(n*0.25)

traning_indexes = indexes[1:end_traning]
validation_indexes = indexes[(end_traning+1):end_
   validation]
testing_indxes = indexes[(end_validation+1):n]

train = data[traning_indexes,]
validation = data[validation_indexes,]
testing = data[testing_indxes,]

dtreefit <- tree(as.factor(good_bad) ~ ., data=train
   , split = c("deviance"))
```

```
gtreefit <- tree(as.factor(good_bad) ~ ., data=train
   , split = c("gini"))

d_yfit = predict(dtreefit, newdata = testing,type="
   class")
g_yfit = predict(gtreefit, newdata = testing,type="
   class")
plot(dtreefit)
plot(gtreefit)

d_table = table(d_yfit,testing$good_bad)
g_table = table(g_yfit,testing$good_bad)

print(d_table)
print(1-sum(diag(d_table))/sum(d_table))
print(g_table)
print(1-sum(diag(g_table))/sum(g_table))

nv = summary(dtreefit)[4]$size
train_score = rep(0,nv)
test_score = rep(0,nv)
for(i in 2:nv){
  pruned=prune.tree(dtreefit,best=i)
  pred=predict(pruned, newdata=validation, type="
     tree")
  train_score[i] = deviance(pruned)
  test_score[i] = deviance(pred)
}
plot(2:nv,train_score[2:nv], col="Red",type = "b",
   ylim=c(min(test_score[2:nv]),max(train_score)))
points(2:nv,test_score[2:nv],col="Blue",type="b")

final = prune.tree(dtreefit,best=4)
yfit = predict(final,newdata=validation,type="class"
   )
f_table = table(validation$good_bad,yfit)
print(f_table)
print(1-sum(diag(f_table))/sum(f_table))
plot(final)

# Naive bayes ????
```

```
bayes_model = naiveBayes(good_bad ~., data=train)

test_yfit = predict(bayes_model, testing[,-ncol(
   testing)], type = "class")
train_yfit = predict(bayes_model, train[,-ncol(train
   )])

naive_table = table(test_yfit,testing$good_bad)
naive_table_train = table(train_yfit,train$good_bad)


print(naive_table_train)
print(1-sum(diag(naive_table_train))/sum(naive_table
   _train))

print(naive_table)
print(1-sum(diag(naive_table))/sum(naive_table))

# With loss matrix
bayes_model = naiveBayes( good_bad ~ ., data = train
   )

test_yfit = predict(bayes_model, testing[,-ncol(
   testing)],type="raw")
train_yfit = predict(bayes_model, train[,-ncol(train
   )], type="raw")

test_yfit =  (test_yfit[, 2] / test_yfit[, 1]) > 10
train_yfit =  (train_yfit[, 2] / train_yfit[, 1]) >
   10

naive_table = table(test_yfit,testing$good_bad)
naive_table_train = table(train_yfit,train$good_bad)

print(naive_table_train)
print(1-sum(diag(naive_table_train))/sum(naive_table
   _train))

print(naive_table)
print(1-sum(diag(naive_table))/sum(naive_table))
```

TDDE01 – Machine Learning
Group 9 Laboration Report 4

Martin Estgren <mares480>
Erik S. V. Jansson <erija578>
Sebastian Maghsoudi <sebma654>

Linköping University (LiU), Sweden

December 4, 2016

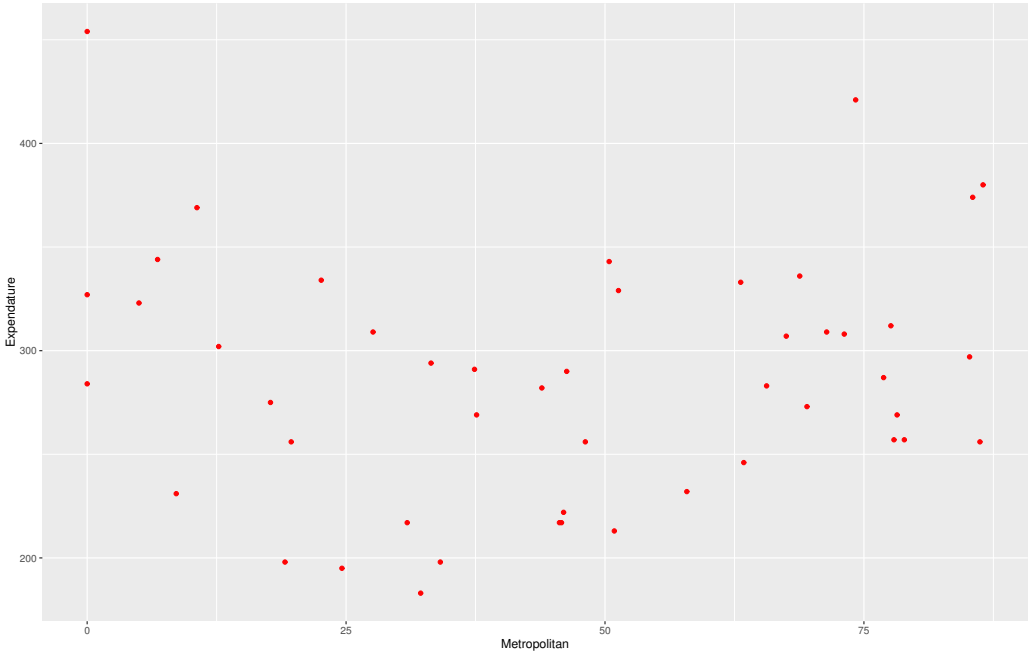
Assignment 1

This assignment involves examining a given data set called *State*, which contains the observations about the *population* & *economy* in different *states*. We are primarily interested in the relationship between the *metropolitan habitation rate (MET)* and the *public expenditure per capita (EX)* for a state.

Raw Data Analysis

We first analyze the data by plotting the *EX* target as a function of *MET*. These results can be observed in the plotted Figure 1, notice the spread of data.

Figure 1: Plot of Metropolitan Rate & Expenditure



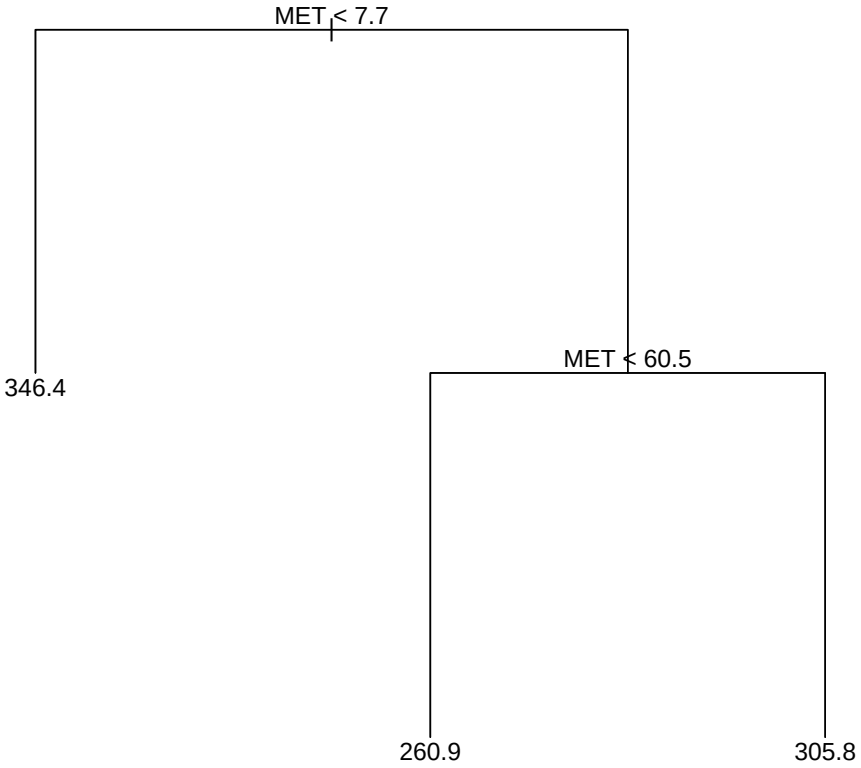
The figure indicates that a linear model isn't suitable for predicting the target in this data set, no easily visible pattern can be observed in this plot. Yet it would be bold to say that there is no correlation between the two, one simply has to use a different scale to get a better perspective. Since we are tasked with using *regression trees* in this assignment, the first step is to fit such a model with it, then finding the optimal number of leaves.

Regression Tree Analysis

We have examined how a *regression tree model* fits the data set using *cross-validation* for finding the optimal number of *leaves.* The plotted *decision tree* from the fitted model can be seen in Figure 2. This model was fitted with the following piece of *R* code:

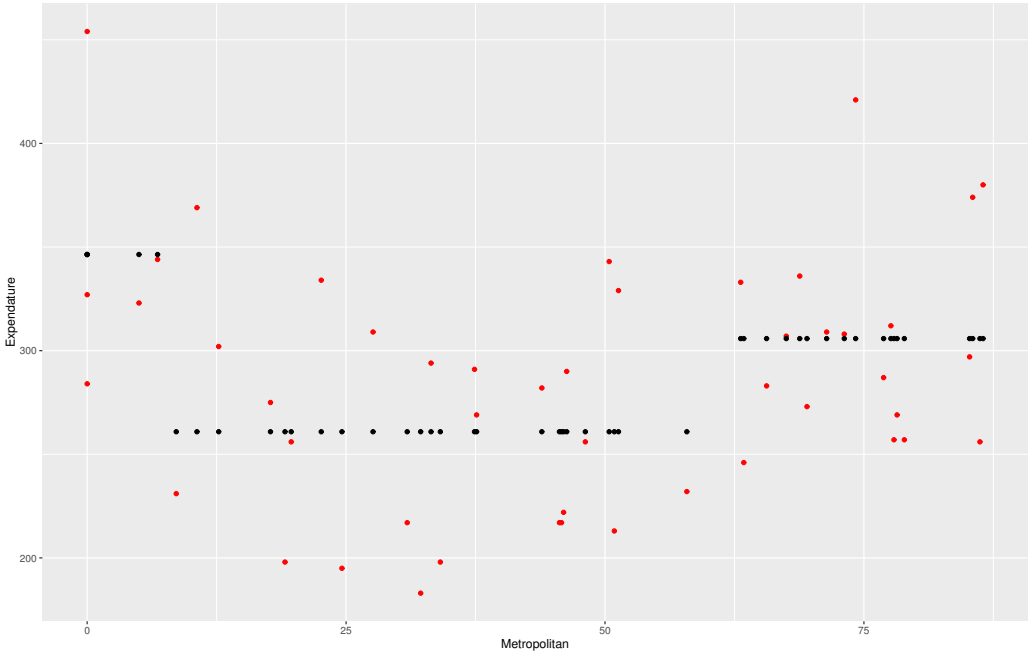
```
control = tree.control(nrow(data), minsize=8)
fit = tree(EX~MET, data, control=control)
fit.cv = cv.tree(fit)
best_k = fit.cv$size[which.min(fit.cv$dev)]
optimal_tree = prune.tree(fit, best=best_k)
```

Figure 2: Plotted Decision Tree for Model



According to the cross-validation, the optimal number of leaves is 3, which we use to prune the full tree model and get the best optimal model. These predicted results of the best model can be seen in Figure 3.

Figure 3: Plot of Metropolitan Rate & Predict Ex

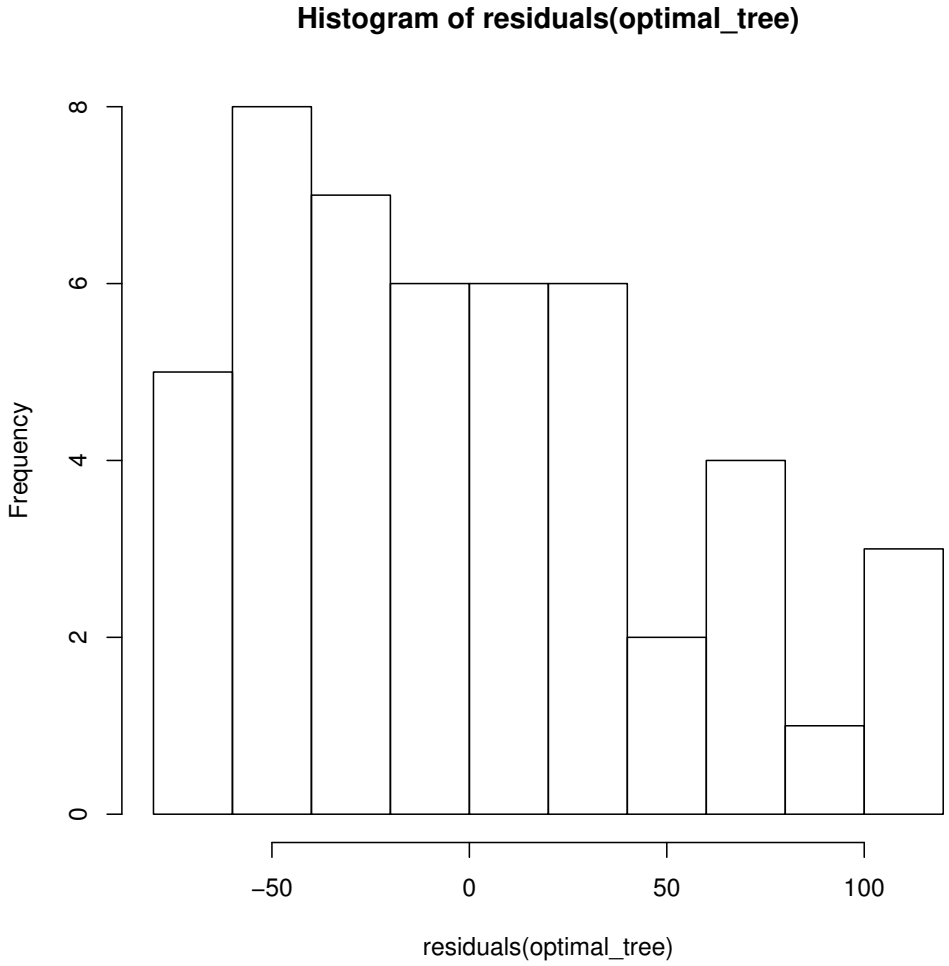


Notice how the above results indicate that the predictions (black) have produced less labels than the original data (in red). These correspond to the three resulting leaves derived from cross-validation. Also, each label is roughly located around the mean of the raw data (that has been divided in "buckets").

Finally, the frequency of the model's residuals are displayed in the Figure 4 below, derived by using:

```
hist(residuals(optimal_tree))
```

Figure 4: Histogram of the Residuals



Preferably, the above distribution should be a *bell curve* around 0, meaning predictions weren't good.

Non-Parametric Bootstrap

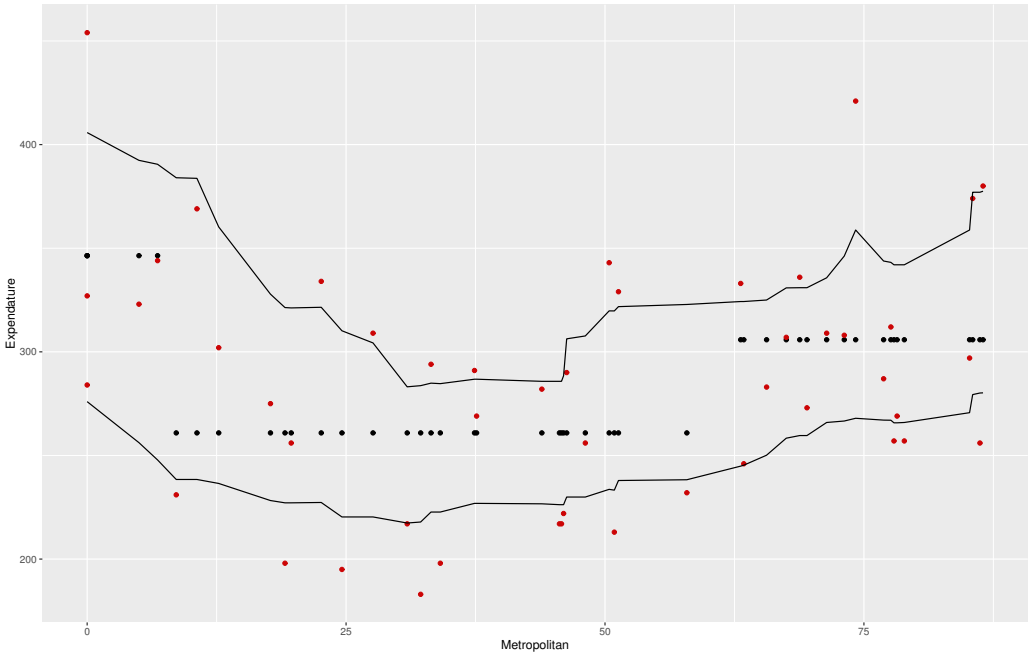
We explore the same data set as before, still being modeled with a *regression tree model* using the *non-parametric bootstrap*. *Bootstrapping* is used to estimate the properties of an estimator by sampling from an approximated distribution. In the case of *non-parametric bootstrapping*, the underlying distribution is assumed to *not* be known, and the data is instead re-sampled with replacement. The estimated distribution is then given by

$\hat{f}(D_1), \hat{f}(D_2), ..., \hat{f}(D_B)$, estimator $\hat{f}$ and data D_i.

For our data set, we re-sample 1000 times using the *non-parametric bootstrapping* function `boot`, for a confidence level of 95 % (which is an $\alpha = 5\%$). By retrieving the *confidence intervals* for each x_i and plotting them, we get the *confidence bands* of the model. Which can be seen in a Figure 5 below. The confidence bands is a calculation of the interval where the best label is with a 95 % probability. Listing 1 does the *non-parametric bootstrapping* in:

```
nonparama = function(data,index) {
    sample = data[index,]
    control = tree.control(nrow(sample), minsize = 8)
    fit = tree( EX ~ MET, data=sample, control = control)
    optimal_tree = prune.tree(fit, best=best_k)
    return(predict(optimal_tree, newdata=data))
}

nonparam_boot = boot(data, statistic = nonparama, R=1000)
confidence_bound = envelope(nonparam_boot,level=0.95)
```

Figure 5: Confidence Bands of Regression Tree with non-parametric bootstrap



Notice how several observations are outside the confidence band derived from our regression tree model.

Parametric Bootstrap

Now the preconditions have changed as opposed to the *non-parametric bootstrap*, since the underlying distribution is now "known" to be: $Y \sim \mathcal{N}(\mu_i, \sigma^2)$. Having the distribution allows us to sample from it, giving us the concept of *parametric bootstrapping*. Similarly, the number of samples to be taken into consideration has been chosen to be 1000 again, with a confidence level of 95 %. Plotting the confidence and prediction bands using *parametric bootstrap* gives Figure 6. The "tighter" band is the *confidence band* while the more relaxed one is the *prediction band*. The following R source produces this:

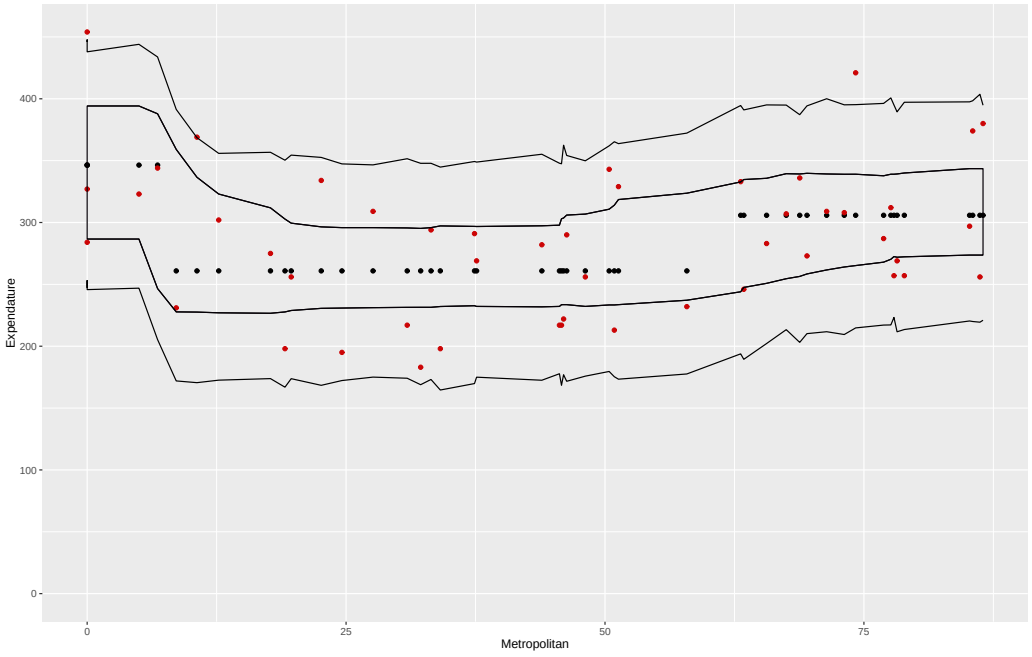
```
parama_conf = function(data){
  control = tree.control(nrow(data), minsize = 8)
  fit = tree( EX ~ MET, data=data, control = control)
  optimal_tree = prune.tree(fit, best=best_k)
  return(predict(optimal_tree, newdata=data))
}
```

```
parama_predic = function(data){
  control = tree.control(nrow(data), minsize = 8)
  fit = tree( EX ~ MET, data=data, control = control)
  optimal_tree = prune.tree(fit, best=best_k)
  predictions = predict(optimal_tree, newdata=data)
  return(rnorm(nrow(data),predictions,sd(resid(fit))))
}
```

```
random_samples = function(data, model){
  sample = data.frame(MET=data$MET, EX=data$EX)
  sample$EX = rnorm(nrow(data), predict(model,newdata=data),sd(
    resid(model)))
  return(sample)
}
```

```
param_boot_conf = boot(data, statistic = parama_conf, R=1000,
    mle = optimal_tree, ran.gen = random_samples, sim = "
    parametric")
confidence_bound_param = envelope(param_boot_conf, level=0.95)
param_boot_pred = boot(data, statistic = parama_predic, R=1000,
    mle = optimal_tree, ran.gen = random_samples, sim = "
    parametric")
prediction_bound_param = envelope(param_boot_pred, level=0.95)
```

Figure 6: Confidence/Prediction Bands Regression the Tree with parametric bootstrap



As can be noted in the figure above, there are a couple of observations outside the prediction band. This is caused because of the 95% level of confidence imposed on the prediction band. It seems reasonable that 5% is outside the prediction band. However, this tells us that our model predicts 95% of the data correctly, which seems a bit convenient. When using this model to predict new observation would be risky since there does not seem to be any radical outliers in the observed space. By referring back to the residual histogram, the assumption that the data is normally distributed is a bit of a stretch, at least given the current data set. *Non-parametric bootstrapping* seems more suitable.

Assignment 2

In this assignment we are tasked with analyzing a data set containing observations regarding different levels of *viscosity* and *near-infrared spectra* for many different *diesel fuels* using *PCA* and *ICA* (*Component Analysis Functions*).

Principal Component Analysis

Principal Component Analysis (PCA) is used to reduce the number of dimensions in a data set by analyzing the variance that each feature contributes to the distribution. We conduct a PCA on the given data set, this gives us the histogram in Figure 7. The R code responsible:

```
spectra <- read.csv2("NIRSpectra.csv")
xspectra <- scale(spectra[,-ncol(spectra)])
yspectra <- spectra[,ncol(spectra)]
principal_comp <- prcomp(xspectra)
lambda <- principal_comp$sdev^2
```

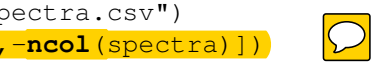
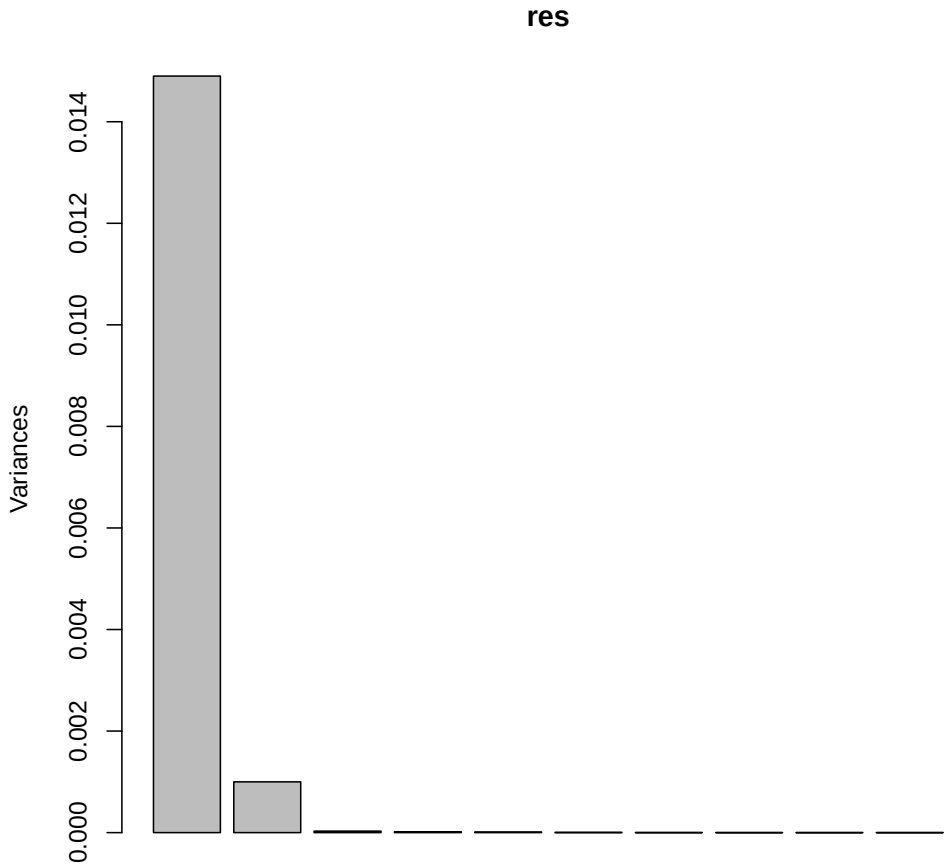
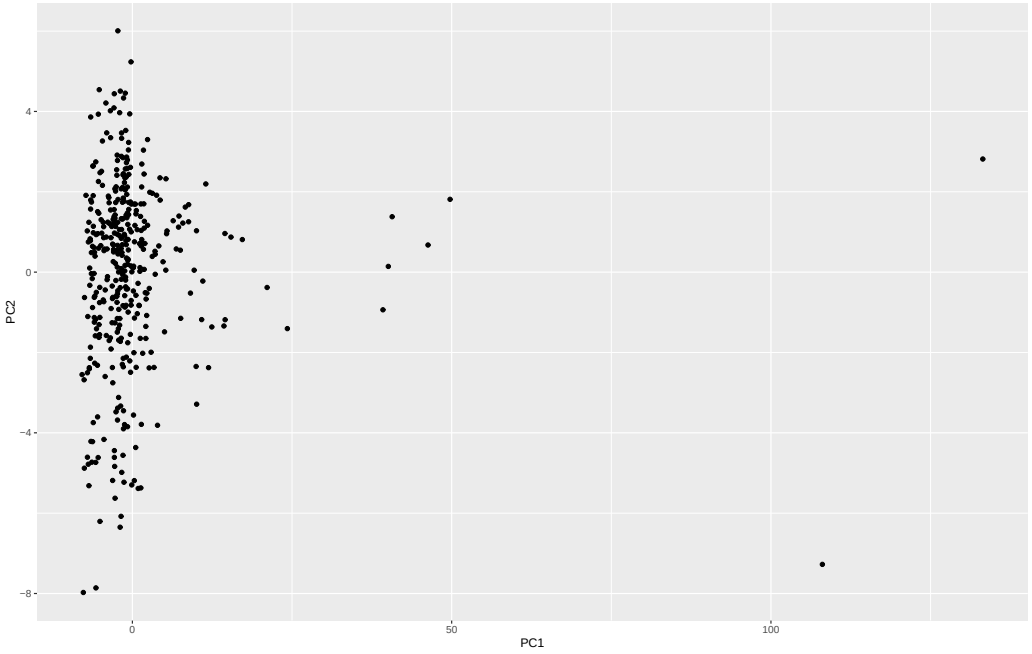


Figure 7: PCA histogram of variance



We observe that only two components are required to reach 99 % cumulative variance, which are *X750 (PC1)* and *X752 (PC2)*. Afterwards, we plot the scores of the chosen principal components in Figure 8.

Figure 8: PCA score distribution



As can be seen, there is a large amount of independence between these two components. The outliers indicates unusual diesel fuels. Now we plot the loadings, which indicates the correlation between components through proportional variance. This is done in Figures 9, 10.

Figure 9: Trace plot of PC1

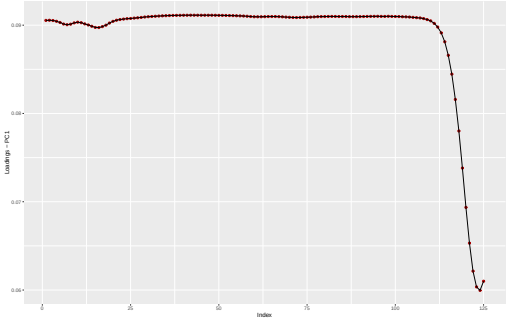
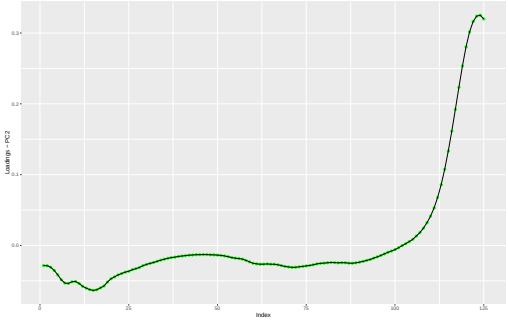


Figure 10: Trace plot of PC2



Note the high correlation values in the same index on both plots. High value indicates higher correlation with the selected components. There are

a couple of components which are relatively highly explained by the components extracted from the previous analysis. PC1 correlates highly with the first components and PC2 with the last ones.

Independent Component Analysis

We perform the same analysis again, this time using the *Independent component analysis* (ICA). In contrast to to PCA, where PCA strives to find correlation, ICA tries to find hidden variables that describes the data but are completely independent of each other. This would require the a new feature space.

The loadings are calculated with the function $\hat{W} = K\dot{W}$. We plot the traces for each column (the chosen components), the result can observed in Figures 11, 12.

```
independent_comp <- fastICA(xspectra, 2)
W <- independent_comp$K %*% independent_comp$W
x750whitening <- W[,1] # Un-mixed and whitened
x752whitening <- W[,2] # Un-mixed and whitened
```

Figure 11: Trace plot of IC1

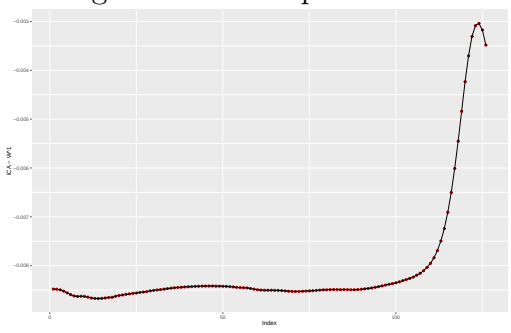
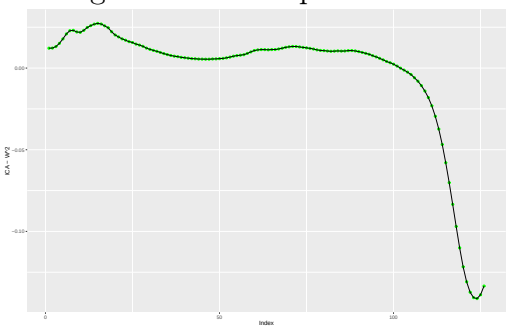
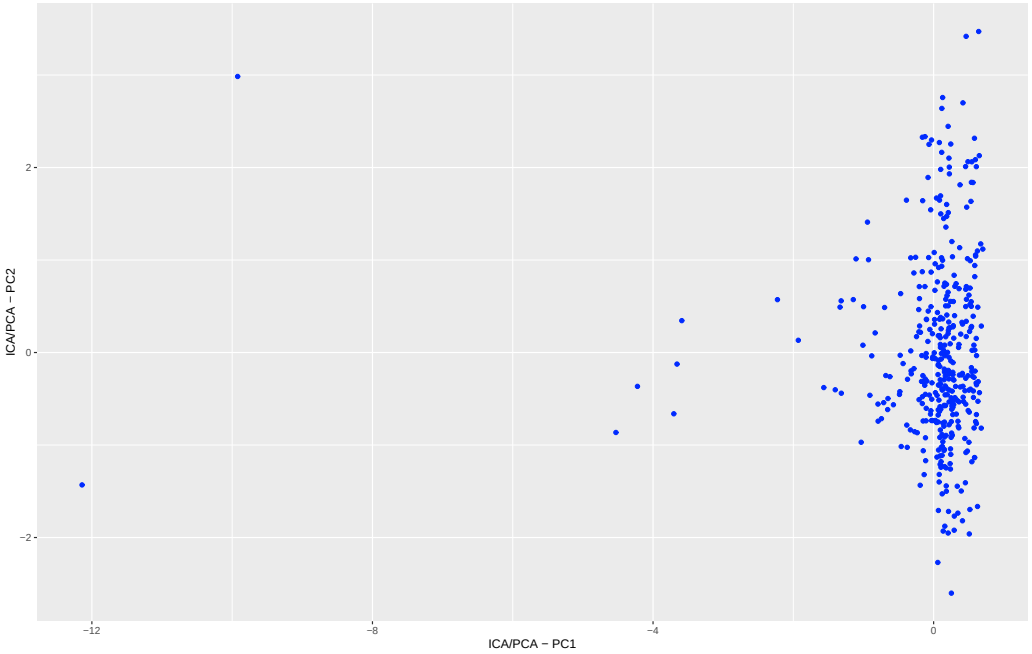


Figure 12: Trace plot of IC2



The results are quite similar to those found in PCA, but inverted. Now the IC1 correlates with the last couple of components while IC2 correlates with the first ones. The reason it is inverted is that we now use a different feature space. We now plot the scores found by doing ICA, which can be seen in Figure 13.

Figure 13: ICA score distribution



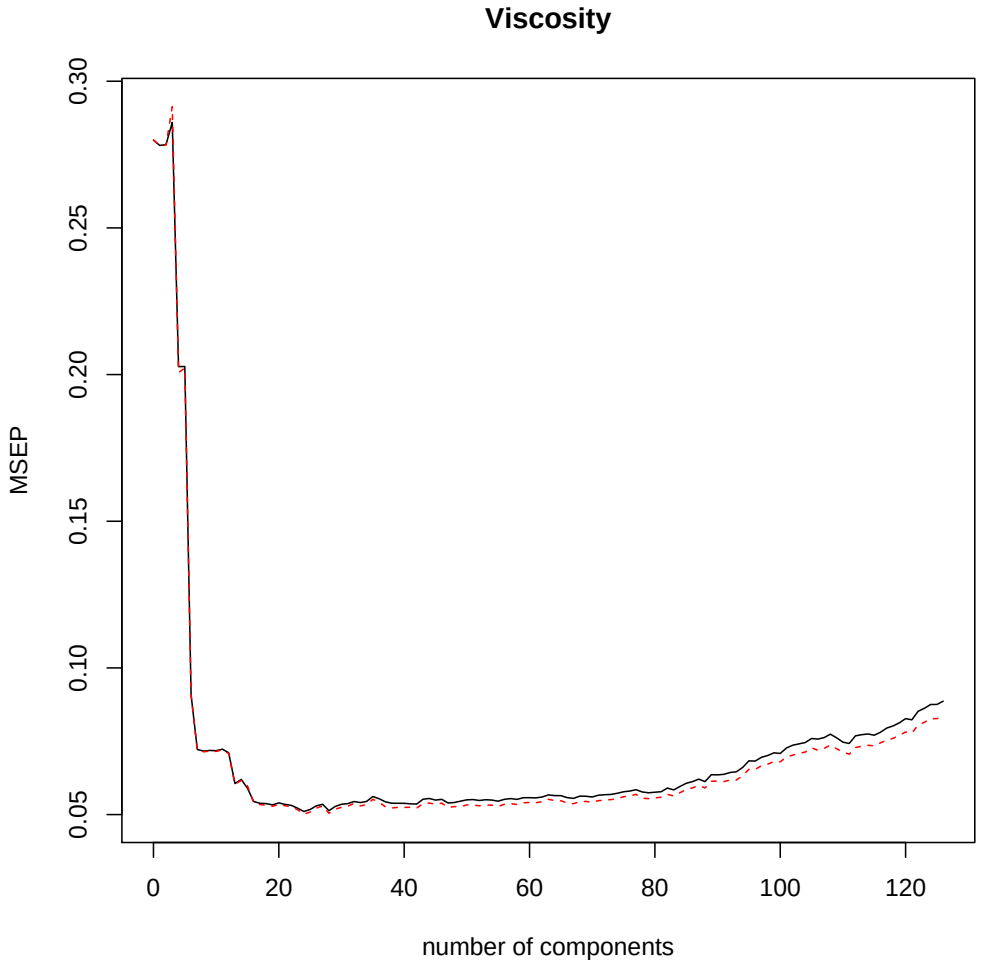
The score distribution has a much higher magnitude, again because we use a new feature space, compared to PCA and the distribution is mirrored in the Y-axis. Otherwise they look very similar. The PCA score distribution can be observed in figure 8.

PCA Cross-Validation

We perform a *Principal component regression* analysis with *cross validation* in order to examine the number of components that should be selected. Below is the resulting plot (13) of *predicted mean-squared error* in relation to the number of components selected for the model.

```
principal_compcv <- pcr(Viscosity ~ ., data = spectra,
                        validation = "CV")
```

Figure 14: Mean squared predicted error over number of components



We can observe from the figure how the MSEP is high when few components are selected and how the MSEP drastically decreases around 8 components. The number of components with the lowest MSEP is around 20 and after that it slowly increases. The optimal number of components appears to be around 20.

Contributions

Most of the text contained in this report was written in conjunction with the entire group, therefore the main body of the text is the work of everyone.

- **Martin Estgren:** provided all the figures, the code for assignment 1, and wrote a lot of the analysis of assignment 2.
- **Erik S. V. Jansson:** provided the script for assignment 2, and polished several sections found in assignment 1.
- **Sebastian Maghsoudi:** provided some of the theoretical reasoning found in chapter 1. In chapter 2 most of the reasoning can be found in the ICA section.

References

[FHT09] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*. Springer series in statistics, Berlin, second (11th) edition, 2009.

Appendix

Listing 1: Script for Assignment 1 on Bootstrapping

```
library(tree)
library(boot)
library(ggplot2)

set.seed(12345)
data = read.csv2("State.csv", header = TRUE)
data = data[order(data$MET),]

control = tree.control(nrow(data), minsize=8)
fit = tree(EX~MET, data, control=control)
fit.cv = cv.tree(fit)
best_k = fit.cv$size[which.min(fit.cv$dev)]
optimal_tree = prune.tree(fit, best=best_k)

predictions = predict(optimal_tree, newdata=data)

fig_data = data.frame(x = data$MET, pred = predictions, orig =
   data$EX)
fig = ggplot(fig_data, aes(x, pred, orig) , xlab = "Metropolitan
   " , ylab = "Expendature")
fig = fig + geom_point(aes(x,orig), colour = "#FF1111") + geom_
   point(aes(x, pred))
print(fig)

hist(residuals(optimal_tree))

 set.seed(12345)
 nonparama = function(data,index) {
     sample = data[index,]
     control = tree.control(nrow(sample), minsize = 8)
     fit = tree( EX ~ MET, data=sample, control = control)
     optimal_tree = prune.tree(fit, best=best_k)
     return(predict(optimal_tree, newdata=data))
 }

 nonparam_boot = boot(data, statistic = nonparama, R=1000)
 confidence_bound = envelope(nonparam_boot,level=0.95)
 predictions = predict(optimal_tree,data)


plot(nonparam_boot)

fig_data = data.frame(orig = data$EX, x=data$MET, pred=
   predictions, upper=confidence_bound$point[1,], lower=
```

```
    confidence_bound$point[2,])
fig = ggplot(fig_data, aes(x,predictions,upper,lower), xlab = "
    Metropolitan" , ylab = "Predicted Expendature")
fig = fig +
     geom_point(aes(x, pred)) +
     geom_point(aes(x, orig),colour="#CC1111") +
     geom_line(aes(x,upper)) +
     geom_line(aes(x,lower)) +
     geom_ribbon(aes(x = x, ymin=lower, ymax=upper), alpha=0.05)
print(fig)

set.seed(12345)
parama_conf = function(data){
  control = tree.control(nrow(data), minsize = 8)
  fit = tree( EX ~ MET, data=data, control = control)
  optimal_tree = prune.tree(fit, best=best_k)
  return(predict(optimal_tree, newdata=data))
}

parama_predic = function(data){
  control = tree.control(nrow(data), minsize = 8)
  fit = tree( EX ~ MET, data=data, control = control)
  optimal_tree = prune.tree(fit, best=best_k)
  predictions = predict(optimal_tree, newdata=data)
  return(rnorm(nrow(data),predictions,sd(resid(fit))))
}

random_samples = function(data, model){
  sample = data.frame(MET=data$MET, EX=data$EX)
  sample$EX = rnorm(nrow(data), predict(model,newdata=data),sd(
    resid(model)))
  return(sample)
}

param_boot_conf = boot(data, statistic = parama_conf, R=1000,
    mle = optimal_tree, ran.gen = random_samples, sim = "
    parametric")
confidence_bound_param = envelope(param_boot_conf, level=0.95)
param_boot_pred = boot(data, statistic = parama_predic, R=1000,
    mle = optimal_tree, ran.gen = random_samples, sim = "
    parametric")
prediction_bound_param = envelope(param_boot_pred, level=0.95)

predictions = predict(optimal_tree,data)
fig_data = data.frame(orig = data$EX, x=data$MET, pred=
    predictions, upper_c=confidence_bound_param$point[1,], lower_
    c=confidence_bound_param$point[2,], upper_p=prediction_bound_
    param$point[1,], lower_p=prediction_bound_param$point[2,])
```

```
  fig = ggplot(fig_data, aes(orig,x,pred,upper_c,lower_c, upper_
   p, lower_p), xlab = "Metropolitan" , ylab = "Predicted
   Expendature")
  fig = fig +
    geom_point(aes(x, pred)) +
    geom_point(aes(x, orig),colour="#CC1111") +
    geom_line(aes(x,upper_c)) +
    geom_line(aes(x,lower_c)) +
    geom_ribbon(aes(x = x, ymin=lower_c, ymax=upper_c), alpha
   =0.05, colour = "#110011")+
    geom_line(aes(x,upper_p)) +
    geom_line(aes(x,lower_p))+
    geom_ribbon(aes(x = x, ymin=lower_p, ymax=upper_p), alpha
   =0.05)

  print(fig)
```

Listing 2: Script for Assignment 2 on Component Analysis

```
library("pls")
library("ggplot2")
library("fastICA")

setEPS() # Enables saving EPS format.
spectra <- read.csv2("NIRSpectra.csv")
xspectra <- scale(spectra[,-ncol(spectra)])
yspectra <- spectra[,ncol(spectra)]
principal_comp <- prcomp(xspectra)
lambda <- principal_comp$sdev^2

cairo_ps("screeplot.eps")
screeplot(principal_comp,
          ncol(xspectra))
dev.off()
cairo_ps("biplot.eps")
biplot(principal_comp)
dev.off()

cairo_ps("score.eps")
print(qplot(principal_comp$x[,1],
            principal_comp$x[,2],
            xlab = "X750",
            ylab = "X752"))
dev.off()

x750loadings <- principal_comp$rotation[,1]
x752loadings <- principal_comp$rotation[,2]
```

```
cairo_ps("x750loadings.eps")
print(qplot(1:length(x750loadings),
            x750loadings, xlab="i",
            ylab="X750 Loadings"))
dev.off()
cairo_ps("x752loadings.eps")
print(qplot(1:length(x752loadings),
            x752loadings, xlab="i",
            ylab="X752 Loadings"))
dev.off()

set.seed(12345)
independent_comp <- fastICA(xspectra, 2)
W <- independent_comp$K %*% independent_comp$W
x750whitening <- W[,1] # Un-mixed and whitened
x752whitening <- W[,2] # Un-mixed and whitened

cairo_ps("x750traceplot.eps")
print(qplot(1:length(x750whitening),
            x750whitening, xlab="i",
            ylab="X750 Inverse Loadings"))
dev.off()
cairo_ps("x752traceplot.eps")
print(qplot(1:length(x752whitening),
            x752whitening, xlab="i",
            ylab="X752 Inverse Loadings"))
dev.off()

cairo_ps("icascore.eps")
print(qplot(independent_comp$S[,1],
            independent_comp$S[,2],
            xlab = "X750",
            ylab = "X752"))
dev.off()

set.seed(12345)
principal_compcv <- pcr(Viscosity ~ ., data = spectra,
                        validation = "CV")
cairo_ps("pcacv.eps")
validationplot(principal_compcv,
               val.type = "MS")
dev.off()
```

TDDE01 – Machine Learning
Group 9 Laboration Report 5

Martin Estgren <mares480>
Erik S. V. Jansson <erija578>
Sebastian Maghsoudi <sebma654>

Linköping University (LiU), Sweden

December 11, 2016

Increasing the accuracy of *weather forecasts* is an important task. We propose an estimator which produces the *air temperature forecast* in *Sweden*, given a *latitude/longitude coordinate* and also *date*. Some observations by *SMHI*, taken from weather stations, have been given for training our estimator.

By using a *Nadaraya–Watson regression kernel*, we can estimate the temperatures $\boldsymbol{y'}$. This is done by taking the *kernels* $k_\sigma(\boldsymbol{x}^{(i)}, \boldsymbol{x'})$ for each i^{th} data from the training set and using it as a *weight* when considering the response variable $\boldsymbol{y}^{(i)}$. Essentially, the kernel $k_\sigma(\boldsymbol{x}^{(i)}, \boldsymbol{x'})$ will reduce $\boldsymbol{y}^{(i)}$'s significance in the *total contribution* by giving less weight when the $\boldsymbol{x^{(i)}}$ and $\boldsymbol{x'}$ are further away (in some measure).

We have used a *Gaussian Radial Basis Function* as our *kernel*, which is defined in Equation 1 below. Note the parameter σ, which can be considered as the *spread* or *width* of the kernel, and also $\boldsymbol{x^{(i)}} - \boldsymbol{x'}$ which is the *distance function*; giving our kernel the property of a *similarity function* (because of $e^{(\cdots)}$).

By using $k_\sigma(\boldsymbol{x}^{(i)}, \boldsymbol{x'})$ in *Nadaraya–Watson's* $\boldsymbol{y'}$ estimator, shown in Equation 2, we are essentially *weighing* how important the contributions from $\boldsymbol{y}^{(i)}$ are to $\boldsymbol{y'}$, because *similar* $\boldsymbol{x}^{(i)}$ will give higher k_σ.

$$k_\sigma(\boldsymbol{x}, \boldsymbol{x'}) = \exp\left(\frac{-\left\|(\boldsymbol{x} - \boldsymbol{x'})\right\|^2}{2\sigma^2 \ \{\sigma \approx h\}}\right) \tag{1}$$

$$\boldsymbol{y'}(\boldsymbol{x}, \boldsymbol{x'}) = \frac{\sum_n \boldsymbol{y}^{(i)} k_\sigma(\boldsymbol{x}^{(i)}, \boldsymbol{x'})}{\sum_n k_\sigma(\boldsymbol{x}^{(i)}, \boldsymbol{x'})} \tag{2}$$

```
gaussian = function(v){
  return(exp(-v^2))
}
```

We now describe the functions which were used to determine the distance/similarity for the kernels.

Below follows the applied *distance functions*, which give the measured distance between a pair of *locations*, *times of the day*, and also *dates of year*. These distance and kernel functions can be found in Listing 1, where they follow similarly as described.

$$d_l = r\,\mathrm{hav}^{-1}(h),\ \mathrm{hav}(\varphi) = \frac{1 - \cos\varphi}{2}$$

```
kernel.distance = function(data_pos, obs_pos){
  # Use haversine distance
  dist = distHaversine(obs_pos,data_pos)
  return(gaussian(dist/ h_distance))
}
```

$$d_t = \begin{cases} |x - y| & |x - y| < (x + y) \bmod 24 \\ (x + y) \bmod 24 & |x - y| \geq (x + y) \bmod 24 \end{cases}$$

```
kernel.time_distance = function(data_time,obs_
    time){
  # Use diff in hours
  dist = as.numeric(difftime(obs_time, data_
    time, units = "hours"))
  dist = abs(dist)
  dist[dist > 12] = 24 - dist[dist > 12]
  return(gaussian(dist/ h_time))
}
```

$$d_d = \begin{cases} |x - y| & |x - y| < (x + y) \bmod 365 \\ (x + y) \bmod 365 & |x - y| \geq (x + y) \bmod 365 \end{cases}$$

```
kernel.date_distance = function(data_date, obs
    _date){
  # Use diff in days
  dist = as.numeric(difftime(obs_date, data_
    date, units = "days"))
  dist = abs(dist)
  dist[dist > 182] = 365 - dist[dist > 182]
  #print(dist)
  return(gaussian(dist  / h_date))
}
```

Finally, we have the *three different kernels* which contribute in unison to the final *forecast kernel*. Each of these use Equation 1 with their respective *distance functions* and also *kernel width* h (σ here).

```
  dist_pos = kernel.distance(obs_pos,data_pos)
  dist_date = kernel.date_distance(obs_date,
    data_date)
  dist_time = kernel.time_distance(obs_time,
    data_time)
  dist = dist_pos + dist_date + dist_time
```

An obvious question the reader has by now is:

What values should be assigned to $2\sigma^2$*?*

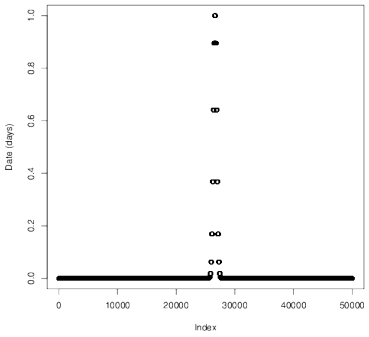
If said reader does not wonder about this, said reader needs to be more intellectually involved in this papers reasoning in the future. However, the answer to the question has to do with the scale of the range of values that the distance can have. One could assign 1 to as a weight, and it might be sufficient, especially if the values are not very large for relatively close observations. This means it is completely related to the context. When in the the context of physical distance it is reasonable to use 1000000 as h since if the point of interest is only a few km away from an already existing observation, the kernel value will return close to 0 values. For date and time, the values already seems to be good enough and only needs to be modified with very small factors. Date uses 3 for h and time uses 2. The intuition seems to be confirmed when observing the result from the kernels presented in Figures 3, 1 and 2, which are described in detail.

The date and time formats for the data set required pre-processing before they could be used in the above mentioned functions. This is done with:

```
fix_time = function(data){
  data$time = as.POSIXct(data$time,format="%H
    :%M:%S")
  data$date = sub('\\d{4}(?=-)', '2016', data$
    date, perl=TRUE)
  return(data)
}
```

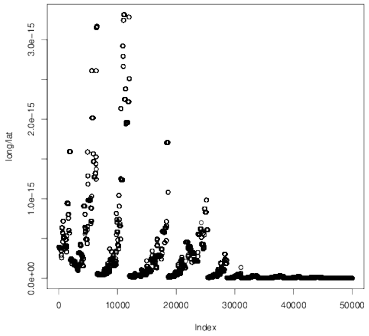
The plots in Figure 1 shows the date-kernel distance for each of the observations. It's harder to detect the trend in the date feature but we can observe a spike each year. For the purpose of processing, all years are set to 2016.

Figure 1: Date distance plot for the observations



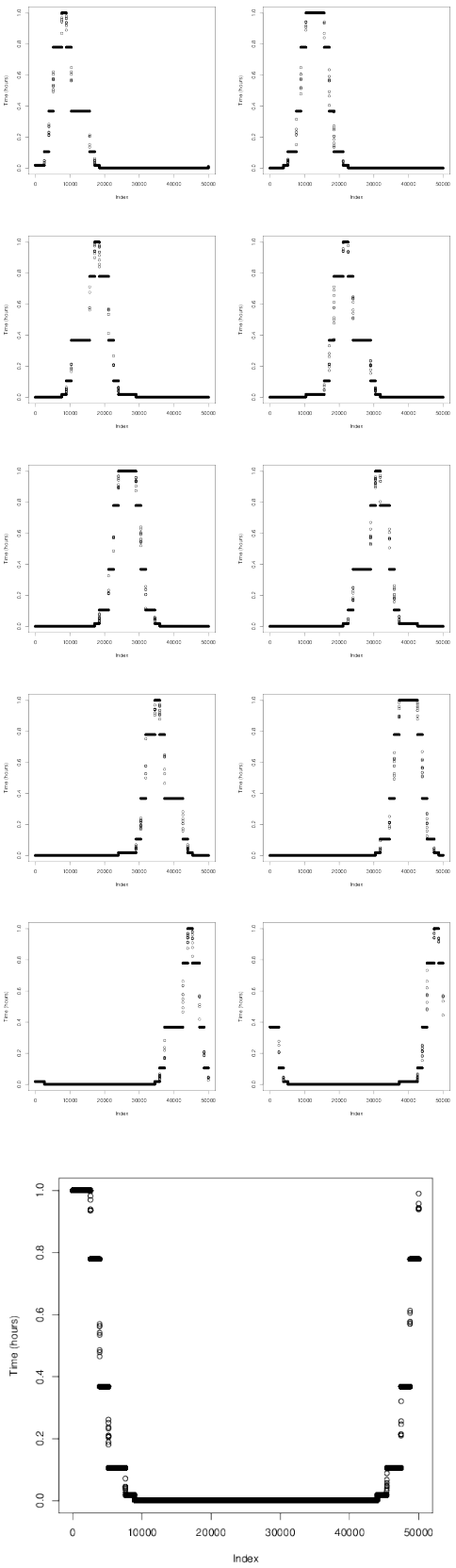
The plots in Figures 2 shows the longitude/latitude-kernel distance for each of the observations. The data is ordered by station number. As expected we can observe a specific group of stations closer with a higher kernel value that others. Given that there's a relation between the station number and location this is expected.

Figure 2: Station distance plot for the observations



The plots in Figures 3 shows the time-kernel distance for each of the observations.

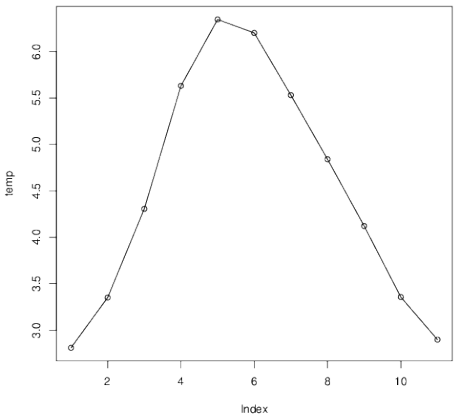
Figure 3: Distance plot (time of day) for the observations (Ordered by time of day)



The figures would indicate that the close a given data sample (time of day) is to the observation the greater the value of the kernel. In the last couple of figures we can observe how the kernel value loops around the clock, as would be expected.

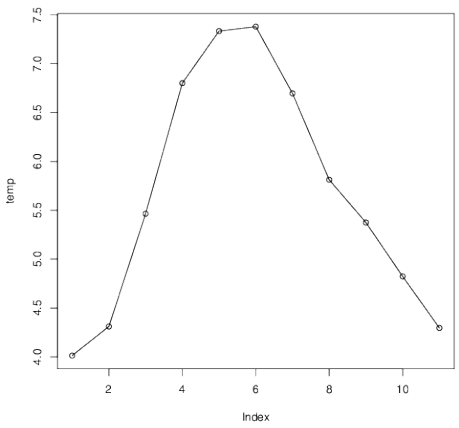
The plots in Figures 4 shows the predicted *air temperature* for each of the observations.

Figure 4: Predicted temperature for the day 2013-04-12 in the interval 04:00-24:00 (ordered by hour)



In Figure 4 the observer might wonder the following question: *Why are the temperatures this low?* The obvious answer is that the point of interest is in Sweden. But the actual answer might be that, since the kernel feature are independent from one another, two "good" values might counter one "bad" value. Meaning that two of the kernels might return very high values since they almost completely match with the point of interest. One value however might be very small, e.g. the date is completely in the wrong season. This issue will pull the predicted temperature to the sample average. However, there is a solution to this problem. In order to avoid that irrelevant observations affects the result, one might exclude such observation. In truth could only include a number of observations (100) that are reasonably close to the point of interest. This will make the result much more reasonable in this aspect. There is however a limit to where the data stops being reasonable because of spikes in temperature change. This happens because to few observations are used.

Figure 5: Predicted temperature for the day 2013-07-12 in the interval 04:00-24:00 (ordered by hour)



In Figure 5 the observations for the date 2013-07-12 can be observed. As described above, the mean temperature of the year means that even in one of the warmest periods where the temperature is expected to be close to 20° we get degrees in the ones of digits.

Contributions

- **Martin Estgren:** Provided the all the figures and code listings, also wrote some of the result analysis.
- **Erik S. V. Jansson:** Gave a general introduction to the theory relevant in this assignment, such as the *gaussian kernel* and *Nadaraya–Watson kernel regression equations.*
- **Sebastian Maghsoudi:** Answered questions specified in the assignment,(width and odd temperatures).

References

[FHT09] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning.* Springer series in statistics, Berlin, second (11th) edition, 2009.

Appendix

Listing 1: Script for Assignment

```
set.seed(1234567890)
library(geosphere)
library(ggplot2)

# Read the data files
stations <- read.csv("stations.csv", stringsAsFactors=FALSE)
temps <- read.csv("temps50k.csv", stringsAsFactors=FALSE)
st <- merge(stations,temps,by="station_number")

# Filter only interesting features
st = st[,c("longitude", "latitude" , "date", "time", "air_temperature")]

# To reduce time
ind <- sample(1:50000, 5000)
st <- st[ind,]

# Weights for ech of the kernels
h_distance = 1000000
h_time = 2
h_date = 3

# Gaussian functiondistance function
gaussian = function(v){
  return(exp(-v^2))
}

# Distance kernel for long and lat
kernel.distance = function(data_pos, obs_pos){
  # Use haversine distance
  dist = distHaversine(obs_pos,data_pos)
  return(gaussian(dist/ h_distance))
}

# Distance kernel for date
kernel.date_distance = function(data_date, obs_date){
  # Use diff in days
  dist = as.numeric(difftime(obs_date, data_date, units = "days"))
  dist = abs(dist)
  dist[dist > 182] = 365 - dist[dist > 182]
  #print(dist)
  return(gaussian(dist  / h_date))
}

# Distance kernel for time
kernel.time_distance = function(data_time,obs_time){
  # Use diff in hours
  dist = as.numeric(difftime(obs_time, data_time, units = "hours"))
  dist = abs(dist)
  dist[dist > 12] = 24 - dist[dist > 12]
  return(gaussian(dist/ h_time))
}

# Kernel function, serves as wrapper for distance kernels
kernel = function(data, observation, index){
  data_fixed = fix_time(data)
  observation = fix_time(observation)
```

```
  # Etract feature vectors
  data_pos  = data_fixed[,c("longitude", "latitude")]
  obs_pos   = c(observation$longitude, observation$latitude)
  data_date = data_fixed$date
  obs_date  = observation$date
  data_time = data_fixed$time
  obs_time  = observation$time

  # Calcualte kernels
  dist_pos = kernel.distance(obs_pos,data_pos)
  dist_date = kernel.date_distance(obs_date,data_date)
  dist_time = kernel.time_distance(obs_time,data_time)
  dist = dist_pos + dist_date + dist_time
  data$distance = dist
  data$dist_pos = dist_pos
  data$dist_date = dist_date
  data$dist_time = dist_time

  # Plot distance kernels
  setEPS()
  postscript(paste(index,sep="","_dist.eps"))
  plot_sample = data$dist_pos
  plot(1:length(plot_sample),plot_sample, ylab = "long/lat", xlab = "Index")
  dev.off()

  setEPS()
  postscript(paste(index,sep="","_date.eps"))
  plot_sample = data[order(data$date),]$dist_date
  plot(1:length(plot_sample),plot_sample, ylab = "Date (days)", xlab = "Index")
  dev.off()

  setEPS()
  postscript(paste(index,sep="","_time.eps"))
  plot_sample = data[order(data$time),]$dist_time
  plot(1:length(plot_sample),plot_sample, ylab = "Time (hours)", xlab = "Index")
  dev.off()

  # Pick the N best observations
  #n = nrow(data)
  selection = data#data[order(data$distance, decreasing = TRUE),][n,]

  # Return the mean temperature over the picked obsrvations
  return(sum(selection$distance * selection$air_temperature) / sum(selection$distance))
}

# Make usre time and date are in propper formats
fix_time = function(data){
  data$time = as.POSIXct(data$time,format="%H:%M:%S")
  data$date = sub('\\d{4}(?=-)', '2016', data$date, perl=TRUE)
  return(data)
}

# Set observation featuers
a <- 58.4274
b <- 14.826
date <- c("2013-04-12")
times <- 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")
n = length(time)
temp <- vector(length=length(date))

# # Students' code here
```

```
data = data.frame(date=rep(date,n/length(date)), time=rep(times,n/length(date)), longitude=rep
    (a,n), latitude=rep(b,n))
for(i in 1:nrow(data)){
  temp[i]  = kernel(st, data[i,],i)
}

print(temp)
setEPS()
postscript(paste("result",sep=".","eps"))
plot(temp, type="o")
dev.off()
```

TDDE01 – Machine Learning
Group 9 Laboration Report 6

Martin Estgren <mares480>
Erik S. V. Jansson <erija578>
Sebastian Maghsoudi <sebma654>

Linköping University (LiU), Sweden

December 18, 2016

In this assignment we are tasked with approximating a sinus function using a neural net with one hidden layer of 10 nodes. The activation function we will use is the sigmoid function with batched gradient descent and resilient back propagation.

```
variable <- runif(50, 0, 10)
sine <- data.frame(x=variable, sin=sin(
    variable))
training <- sine[1:25,] ; testing <- sine
    [26:50,]
```

First a set of 50 randomly generated observations from the sinus function and split in two parts of equal size, one for training and one for validation. We create 10 neural nets, each with a different cut-off threshold for the gradient descent. We consider the thresholds 0.001, ... , 0.01 in steps of 0.001.

```
    nn <- neuralnet(sin~x, training, units,
                    startweights = weights,
                    threshold = thresholdi)
```

$$\sigma(u) = \frac{1}{1 + e^{-u}} \tag{1}$$

$$\boldsymbol{w}_{(i)} = \boldsymbol{w}_{(i-1)} - \eta_k \nabla E(\boldsymbol{w}_{(i-1)}) \tag{2}$$

$$\hat{y}_j(\boldsymbol{x}) = \sigma(w_0 + \sum_{h=1}^{H} \sigma(w_{0h} + \boldsymbol{w}_h^\intercal \boldsymbol{x})) \tag{3}$$

1. **Sigmoid Activation Function:** "S"-shaped function which converges $\sigma(u) = 1$ as $u \to \infty$ and $\sigma(u) = 0$ as $u \to -\infty$. Used in Equation 3.
2. **Batch Gradient Descent:** finds the "step" in the right direction for *minimizing error* E. This is achieved with the *gradient* of E given in respect to the weights $\boldsymbol{w}$; giving a *hyperplane*.
3. **Single-Layer Neural Network Estimator:** uses Equations 1 and 2 to find $\hat{y}_j$ by finding the parameters $\boldsymbol{w}$ in each layer (a linear equation) by means of *gradient descent* and producing a *non-linear result in subsequent layers* by the *activation function*. This is the primary reason why neural networks are so flexible & general.

We calculate the S.S.E of each of the nets using the validation set and pick the one with the lowest error-rate. In the list below the S.S.E for each of the thresholds can be observed.

```
    predicted <- compute(nn, testing$x)
    error <- sum((testing$sin - predicted$net.
    result)^2)
    cat("NN Threshold", thresholdi, "->",
    error, "SSE \n")
    if (error < candidate_error) {
        candidate_error = error
        candidate_threshold = thresholdi
    }
```

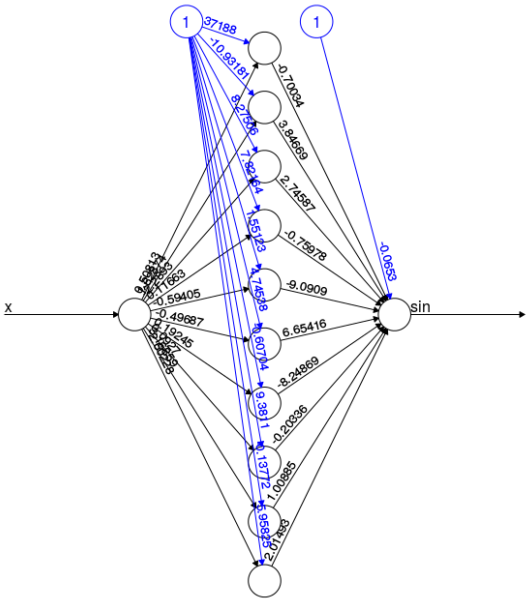
The best threshold was observed to be 0.004.

We recreate the neural net with the threshold 0.004 and generate predictions for the full data set of 50 observations. The result can be seen in figure 2. The figure above shows a graph representation of the neural net. The black numbers are the individual weights and the blue ones are the intersections for each node.

Threshold	S.S.E.
0.001	0.01367691527
0.002	0.01262419958
0.003	0.00988418900
0.004	0.00850089424
0.005	0.00955545744
0.006	0.00974372099
0.007	0.01583926857
0.008	0.01649252416
0.009	0.02112490377
0.010	0.02735909554

Table 1: Neural Network Values

Figure 1: Neural Network



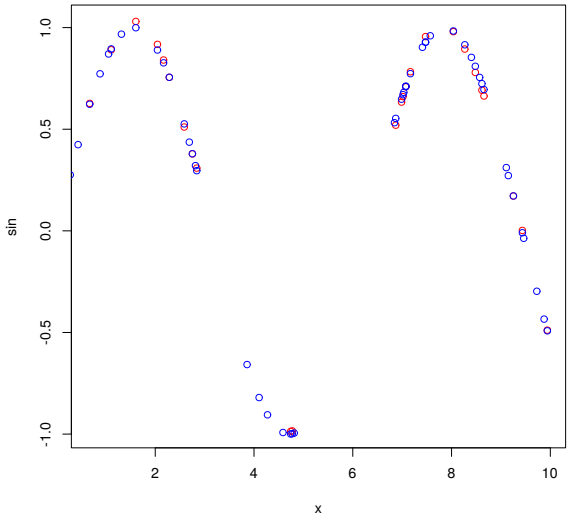
As we can observe in the figure above. The neural net produces a good estimate for the sinus function. Given the sample data.

Contributions

This report was a joint effort between all group members, and as such, everyone has contributed equally.

- **Martin Estgren:** all explanatory text in the report and most of the analysis.

Figure 2: Neural Network's Produced Predictions (in the graph are raw values and predicted values).



- **Erik S. V. Jansson:** provided the listings and the figures.
- **Sebastian Maghsoudi:** in depth analysis of the report and results.

References

[FHT09] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*. Springer series in statistics, Berlin, second (11th) edition, 2009.

[GF10] Frauke Günther and Stefan Fritsch. neuralnet: Training of Neural Networks. *The R Journal*, 2(1):30–38, 2010.

Appendix

Listing 1: Feed-Forward Backpropagating Neural Network Sine Estimator Script

```
library("ggplot2")
library("reshape2")
library("neuralnet")
library("grDevices")
set.seed(1234567890)

variable <- runif(50, 0, 10)
sine <- data.frame(x=variable, sin=sin(variable))
training <- sine[1:25,] ; testing <- sine[26:50,]

candidate_error <- Inf
units <- 10 # Hidden baby!
candidate_threshold <- Inf
weights <- runif(50, -1, +1)

for (threshold_attempt in 1:10) {
    thresholdi <- threshold_attempt / 1000
    nn <- neuralnet(sin~x, training, units,
                    startweights = weights,
                    threshold = thresholdi)

    predicted <- compute(nn, testing$x)
    error <- sum((testing$sin - predicted$net.result)^2)
    cat("NN Threshold", thresholdi, "->", error, "SSE \n")
    if (error < candidate_error) {
        candidate_error = error
        candidate_threshold = thresholdi
    }
}

nn <- neuralnet(sin~x, training, units,
                   candidate_threshold,
                startweights = weights)
predicted <- compute(nn, testing$x)

plot(nn)
setEPS()
cairo_ps("predictions.eps")
plot(testing$x, predicted$net.result, col = "red",
     xlab = "x", ylab = "sin")
points(sine, col = "blue")
dev.off()
```

Listing 2: Output About the Produced Neural Network in the Assignment

```
$response
               sin
1   0.31115890803
2  -0.65787112371
3   0.85356988285
4   0.92820698816
5   0.71194544538
6   0.95969186755
7   0.27531467859
8  -0.03662256168
```

```
9  -0.29718457265
10 -0.43427724087
11  0.27176755816
12  0.96762993527
13  0.87023024548
14  0.90319426880
15  0.53211225475
16 -0.90515370065
17 -0.99209419164
18  0.75493516282
19  0.43639270658
20  0.42400734122
21  0.77254200174
22  0.68138797265
23  0.32070401674
24 -0.99484612705
25 -0.82027249428

$covariate
              [,1]
 [1,] 9.1083657276
 [2,] 3.8595812093
 [3,] 8.4019783861
 [4,] 7.4727497180
 [5,] 7.0754500036
 [6,] 7.5690891198
 [7,] 0.2789170155
 [8,] 9.4614087138
 [9,] 9.7265206091
[10,] 9.8740137112
[11,] 9.1495487187
[12,] 1.3156641065
[13,] 1.0556694935
[14,] 7.4103393406
[15,] 6.8442786904
[16,] 4.2733338126
[17,] 4.5865617390
[18,] 8.5692227143
[19,] 2.6900070859
[20,] 0.4378655413
[21,] 0.8828348410
[22,] 7.0328426105
[23,] 2.8151199827
[24,] 4.8139597056
[25,] 4.1034799209

$err.fct
function (x, y)
{
    1/2 * (y - x)^2
}
<environment: 0x339a758>
attr(,"type")
[1] "sse"

$act.fct
function (x)
{
    1/(1 + exp(-x))
}
<environment: 0x339a758>
attr(,"type")
```

```
[1] "logistic"

$linear.output
[1] TRUE

$data
              x            sin
1  9.1083657276  0.31115890803
2  3.8595812093 -0.65787112371
3  8.4019783861  0.85356988285
4  7.4727497180  0.92820698816
5  7.0754500036  0.71194544538
6  7.5690891198  0.95969186755
7  0.2789170155  0.27531467859
8  9.4614087138 -0.03662256168
9  9.7265206091 -0.29718457265
10 9.8740137112 -0.43427724087
11 9.1495487187  0.27176755816
12 1.3156641065  0.96762993527
13 1.0556694935  0.87023024548
14 7.4103393406  0.90319426880
15 6.8442786904  0.53211225475
16 4.2733338126 -0.90515370065
17 4.5865617390 -0.99209419164
18 8.5692227143  0.75493516282
19 2.6900070859  0.43639270658
20 0.4378655413  0.42400734122
21 0.8828348410  0.77254200174
22 7.0328426105  0.68138797265
23 2.8151199827  0.32070401674
24 4.8139597056 -0.99484612705
25 4.1034799209 -0.82027249428

$net.result
$net.result[[1]]
             [,1]
1   0.30018554412
2  -0.64466078637
3   0.82577426828
4   0.95531707389
5   0.71041413678
6   0.98604140527
7   0.27220011825
8  -0.02393016651
9  -0.27977135370
10 -0.42452441939
11  0.26376340300
12  0.97423750701
13  0.86489547875
14  0.92926095080
15  0.49438343511
16 -0.91926885456
17 -0.98908629285
18  0.72304892189
19  0.42816650637
20  0.43013044493
21  0.76891173652
22  0.67393025023
23  0.32904062832
24 -0.97934291894
25 -0.83193245662
```

```
$weights
$weights[[1]]
$weights[[1]][[1]]
             [,1]          [,2]         [,3]         [,4]         [,5]
[1,] 0.3718763846 -10.931812117  8.275060257 7.8216380497  1.551228805
[2,] 0.5081317757   1.628735531 -2.289303563 0.1166254588 -0.594052483
             [,6]          [,7]          [,8]          [,9]        [,10]
[1,]  4.7453831203 -0.6070429987 9.38110372186 -0.1377222063  5.958245683
[2,] -0.4968713811  0.1924524647 0.09270470267  3.1685937697 -2.602280174

$weights[[1]][[2]]
                [,1]
 [1,] -0.06529714022
 [2,] -0.70033511720
 [3,]  3.84669442716
 [4,]  2.74586539382
 [5,] -0.75978348453
 [6,] -9.09090264177
 [7,]  6.65416477397
 [8,] -8.24869453812
 [9,] -0.20335986240
[10,]  1.00884789102
[11,]  2.01492912933



$startweights
$startweights[[1]]
$startweights[[1]][[1]]
             [,1]          [,2]          [,3]         [,4]          [,5]
[1,] 0.4591262657 -0.5031667114  0.9014065554 0.9589186665  0.3389983536
[2,] 0.5853618421  0.4307389595 -0.8788680169 0.4978831881 -0.5358421607
             [,6]          [,7]         [,8]           [,9]         [,10]
[1,]  0.6293357364 -0.4028865024 0.5968314419 -0.07648990629  0.7421438890
[2,] -0.4464168334 -0.3094406570 0.9041418806 -0.17025629710 -0.8588890089

$startweights[[1]][[2]]
              [,1]
 [1,] -0.2698646844
 [2,] -0.9254528601
 [3,]  0.3498687637
 [4,] -0.7230960354
 [5,] -0.9836924160
 [6,] -0.6452815034
 [7,]  0.8491520174
 [8,]  0.6410233583
 [9,] -0.4020887311
[10,]  0.9406797229
[11,]  0.2142777084



$generalized.weights
$generalized.weights[[1]]
             [,1]
1  -4.18437409317
2   0.84720544205
3  -3.90202008113
4   8.87306916010
5   4.06386829427
6  18.83336456829
```

```
7   5.28857232424
8  38.75962317972
9   2.72921588482
10  1.62852618574
11 -4.58128912031
12 13.07325336492
13  4.31497701909
14  6.93919826366
15  4.07243773606
16  0.23010347764
17  0.02853630559
18 -3.31477743585
19 -3.26159533671
20  3.80902182260
21  3.41516125086
22  3.98677345674
23 -3.57280260613
24 -0.06878886818
25  0.40865817424
```

$result.**matrix**

```
                                  1
error                  0.003576080337
reached.threshold      0.003929680826
steps              23174.000000000000
Intercept.to.1layhid1  0.371876384634
x.to.1layhid1          0.508131775660
Intercept.to.1layhid2 -10.931812117300
x.to.1layhid2          1.628735530657
Intercept.to.1layhid3  8.275060257474
x.to.1layhid3         -2.289303563425
Intercept.to.1layhid4  7.821638049688
x.to.1layhid4          0.116625458773
Intercept.to.1layhid5  1.551228805249
x.to.1layhid5         -0.594052483023
Intercept.to.1layhid6  4.745383120333
x.to.1layhid6         -0.496871381057
Intercept.to.1layhid7 -0.607042998673
x.to.1layhid7          0.192452464714
Intercept.to.1layhid8  9.381103721859
x.to.1layhid8          0.092704702673
Intercept.to.1layhid9 -0.137722206303
x.to.1layhid9          3.168593769723
Intercept.to.1layhid10 5.958245682591
x.to.1layhid10        -2.602280174422
Intercept.to.sin      -0.065297140222
1layhid.1.to.sin      -0.700335117198
1layhid.2.to.sin       3.846694427157
1layhid.3.to.sin       2.745865393824
1layhid.4.to.sin      -0.759783484527
1layhid.5.to.sin      -9.090902641770
1layhid.6.to.sin       6.654164773966
1layhid.7.to.sin      -8.248694538124
1layhid.8.to.sin      -0.203359862396
1layhid.9.to.sin       1.008847891021
1layhid.10.to.sin      2.014929129330
```

attr(,"class")
[1] "nn"
