Introduction to Machine Learning
Individual Laboration Report –1–

Erik Sven Vasconcelos Jansson
`erija578@student.liu.se`
Linköping University, Sweden

November 27, 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 3, 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 2, 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.

Below follow *confusion matrices* & *ROC curves*, and it seems *knearest* gives better results than *kknn*. The *first confusion matrix* below belongs to the *testing data set* while the *second* one belongs to the *training data set*, and finally *testing* for *kknn*. The misslassification rates for each respective *confusion matrix* is: $(0.202, 0.347)$ for *training*, $(0.317, 0.347)$ for *testing* and finally $(0.345, 0.345)$ for *kknn*. Notice how predictions are more "accurate" in *training*.

k=5	**false**	**true**
false	695	193
true	242	240

k=1	**false**	**true**
false	639	178
true	298	255

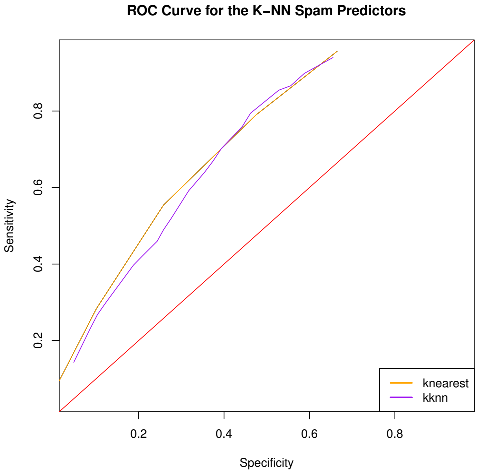
k=5	**false**	**true**
false	787	119
true	158	306

k=1	**false**	**true**
false	939	2
true	6	423

k=5	**false**	**true**
false	640	177
true	297	256

k=1	**false**	**true**
false	640	177
true	297	256

The reason why this happens is because *training* is being used by the *predictor*, and therefore (especially when $k = 1$) will give very "accurate" predictions. This becomes a bit less apparent when we enforce majority voting (taking into account near neighbors), since the predictor doesn't become as biased towards the *training* data set as before (however it still seems $k = 5$ predicts better than *testing* $k = 5$). Note in the ROC curve below that *knearest* performs better than the *kknn*, at least in this case.



Assignment 2

Knowing how to *infer* what parameter θ most likely produced a already given data vector $\boldsymbol{x}$ is useful to gain more information about underlying processes. In this case, that is *expected lifetime of machines*, which has been assumed to follow $p(\boldsymbol{x}|\theta) = \theta e^{-\theta x}$, where $\boldsymbol{x}$ are the *expected lifetimes* of n machines, which are independent and identically distributed. The P.D.F. $\theta e^{-\theta x}$ is from a *exponential distribution*, as seen in *Jonsson et al.* [JN99]: $X \sim \text{Exp}(\mu, \sigma^2)$...

Estimation of the parameter θ can be done with *MLE (Maximum Likelihood Estimation)*, which is usually done by using the *log-likelihood* of θ for a given data vector $\boldsymbol{x}$. The formula for *log-likelihood* is shown in Equation 3, where the parameter θ can then be estimated with $\hat{\theta}_{mle}$ by selecting the most probable θ from a given set of thetas $\Theta, \theta_i \in \Theta$, by *maximizing* the *average log-likelihood*, as in Eq. 4. This is seen in e.g. *Myung* [Myu03] and *Wikipedia*.

$$\mathcal{L}(\theta; \boldsymbol{x}) = \ln p(\boldsymbol{x}|\theta) = \sum_{i=1}^{n} \ln p(x_i|\theta) \tag{3}$$

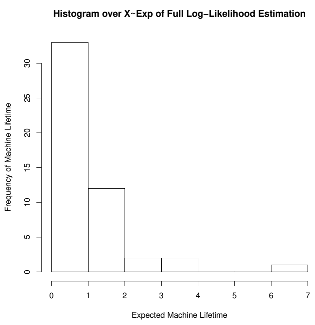
$$\hat{\theta}_{mle} = \max_{\forall\theta} \hat{\mathcal{L}}(\theta; \boldsymbol{x}),\ \hat{\mathcal{L}} = \frac{\ln p(\boldsymbol{x}|\theta)}{n} \tag{4}$$

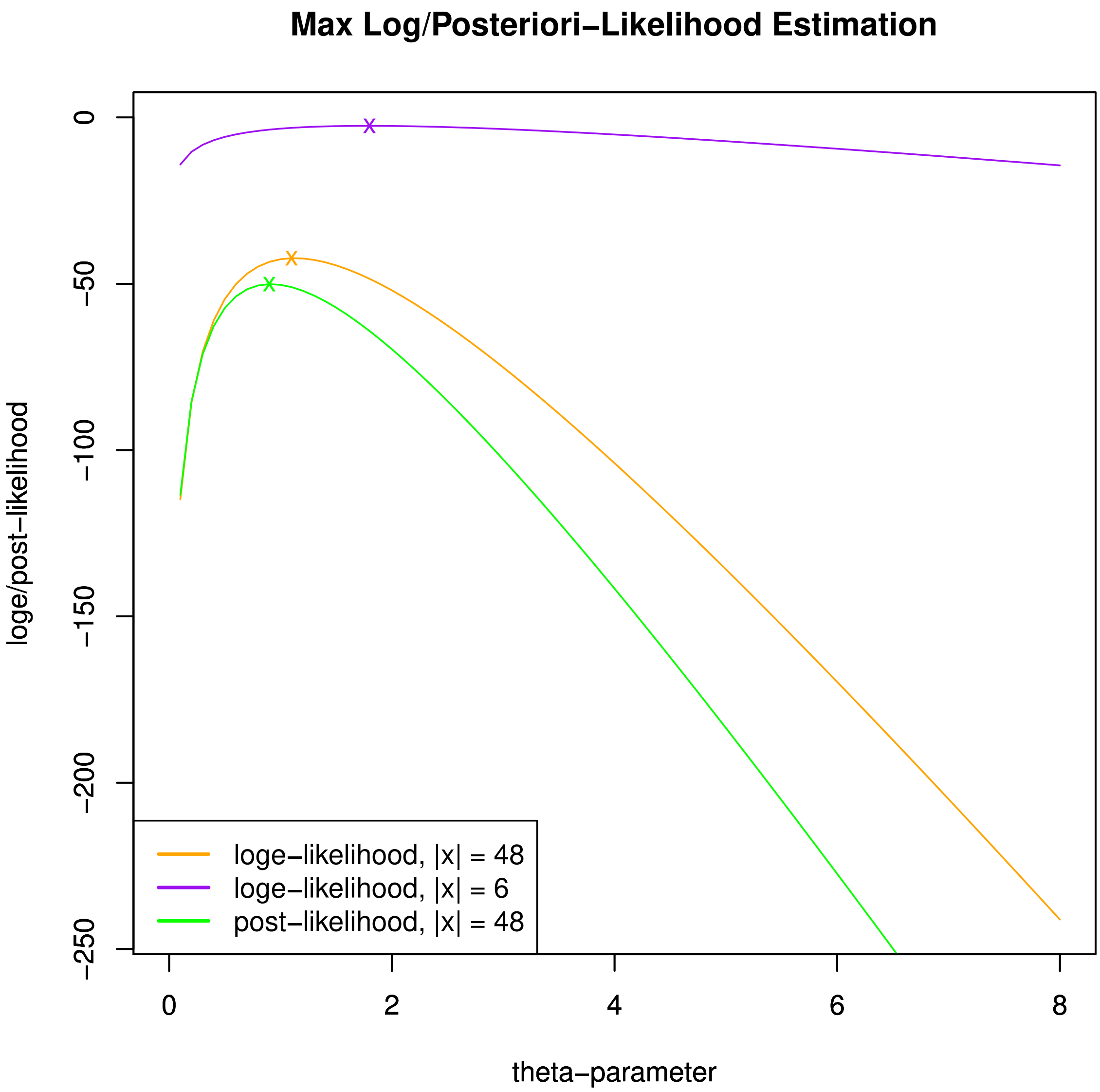
The actual implementation of these is found in Listing 5, in the *R* functions `lnlikelihood` and `distribution` (gives $\theta e^{-\theta x_i}, \forall x_i \in \boldsymbol{x}$). For the MLE, lines `7-9` in Listing 4 calculate the *average log-likelihood* and then picks the *most probable* θ_i. The MLE for the dataset shown in Listing 6 for when the *entire dataset* is used, and when only $|\boldsymbol{x}| = 6$ is taken, is shown in the table to the right. As can be seen in the graph to the right, the $\hat{\theta}_{mle}$ for when $|\boldsymbol{x}| = 6$ will overshoot the value for when the actual full dataset is used, therefore, it is less reliable then when $|\boldsymbol{x}| = 48$, not a good estimator.

Exp. Distribution	**MLE/MAP**
$p(\boldsymbol{x}\|\theta), \|\boldsymbol{x}\| = 48$	1.1
$p(\boldsymbol{x}\|\theta), \|\boldsymbol{x}\| = 6$	1.8
$p(\boldsymbol{x}\|\theta)p(\theta), \|\boldsymbol{x}\| = 48$	0.9

If additional information is know about the distribution, in this case $p(\theta) = \lambda e^{-\lambda\theta}$ where $\lambda = 10$, the *MAP (Maximum a Posteriori)* estimation can be used instead. The implementation can be found in Listing 5, in the *R* function `polikelihood`. The MAP for the $\boldsymbol{x}$ dataset is shown in the table, and the graph for it is the green line to the right. Finally, taking random samples from the distribution with $\hat{\theta}_{mle}$ produces the histogram to the right.

In conclusion, the parameter $\hat{\theta}_{mle}$ is a very good estimator for $\boldsymbol{x}$ given the Exp. distribution $p(\boldsymbol{x}|\theta)$.




Max Log/Posteriori−Likelihood Estimation
loge/post−likelihood
0
−50
−100
−150
−200
−250
0
2
4
6
8
theta−parameter
loge−likelihood, |x| = 48
loge−likelihood, |x| = 6
post−likelihood, |x| = 48

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.

[JN99] Dag Jonsson and Lennart Norell. *Ett stycke statistik.* Studentlitteratur, 1999.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT univeristy press, 1st edition, 2012.

[Myu03] In Jae Myung. Tutorial on maximum likelihood estimation. *Journal of mathematical Psychology*, 47(1):90–100, 2003.

Appendix

Listing 1: Spam Prediction Script

```r
library("kknn")
source("distance.r")
source("knearest.r")

sensitivity <- function(x, y) {
    tp <- sum(x == 1 & y == 1)
    fn <- sum(x == 0 & y == 1)
    return(tp / (tp + fn))
}

specificity <- function(x, y) {
    tn <- sum(x == 0 & y == 0)
    fp <- sum(x == 1 & y == 0)
    return(tn / (tn + fp))
}

set.seed(12345) # For debugging.
data <- read.csv("spambase.csv")
# Pick randomly around half of the rows in dataset.
samples <- sample(1:nrow(data), floor(0.5*nrow(data)))
# Split given dataset evenly for training and tests.
learning <- data.matrix(data[samples,]) # Training.
testing <- data.matrix(data[-samples,]) # Testing.

# Predict spam for learning data K=5.
cat("\nknearest: predicting k = 5 (learning)\n")
k5 <- knearest(learning, 5, learning)
kr5 <- round(k5) # Classify >0.5 -> 1|0.
# Generate the confusion matrix for K=5.
cm5 <- table(kr5, learning[,ncol(learning)])
# Calculate given missclassification.
mc5 <- 1 - sum(diag(cm5)) / sum(cm5)
# Report confusion matrix and error.
print(cm5) ; print(mc5)

# Predict spam for learning data K=1.
cat("\nknearest: predicting k = 1 (learning)\n")
k1 <- knearest(learning, 1, learning)
kr1 <- round(k1) # Classify >0.5 -> 1|0.
# Generate the confusion matrix for K=1.
cm1 <- table(kr1, learning[,ncol(learning)])
# Calculate given missclassification.
mc1 <- 1 - sum(diag(cm1)) / sum(cm1)
# Report confusion matrix and error.
print(cm1) ; print(mc1)

# Predict spam for testing data K=5.
cat("\nknearest: predicting k = 5 (testing)\n")
k5 <- knearest(learning, 5, testing)
kr5 <- round(k5) # Classify >0.5 -> 1|0.
# Generate the confusion matrix for K=5.
cm5 <- table(kr5, testing[,ncol(testing)])
# Calculate given missclassification.
mc5 <- 1 - sum(diag(cm5)) / sum(cm5)
# Report confusion matrix and error.
print(cm5) ; print(mc5)

```

```
# Predict spam for testing data K=1.
cat("\nknearest: predicting k = 1 (testing)\n")
k1 <- knearest(learning, 1, testing)
kr1 <- round(k1) # Classify >0.5 -> 1|0.
# Generate the confusion matrix for K=1.
cm1 <- table(kr1, testing[,ncol(testing)])
# Calculate given missclassification.
mc1 <- 1 - sum(diag(cm1)) / sum(cm1)
# Report confusion matrix and error.
print(cm1) ; print(mc1)

cat("\nkknn: training and predicing with k = 1, 5\n")
m5 <- train.kknn(Spam ~ ., data = data.frame(learning), ks = c(5))
p5 <- predict(m5, data.frame(testing)) # Predict spam with k = 5.
pr5 <- round(p5) # Classify with the function >0.5 -> 1 else 0.
cm5 <- table(pr5, testing[,ncol(testing)])
# Calculate given missclassification.
mc5 <- 1 - sum(diag(cm5)) / sum(cm5)
# Report confusion matrix and error.
print(cm5) ; print(mc5)

cat("\nkknn: training and predicing with k = 1\n")
m1 <- train.kknn(Spam ~ ., data = data.frame(learning), ks = c(1))
p1 <- predict(m1, data.frame(testing)) # Predict spam with k = 1.
pr1 <- round(p1) # Classify with the function >0.5 -> 1 else 0.
cm1 <- table(pr1, testing[,ncol(testing)])
# Calculate given missclassification.
mc1 <- 1 - sum(diag(cm1)) / sum(cm1)
# Report confusion matrix and error.
print(cm1) ; print(mc1)

# Classify dataset by 0.05 steps...
response <- testing[,ncol(testing)]
classify <- seq(0.05, 0.95, by=0.05)
# Apply the classification rule for all.
kc5 <- sapply(k5, function(x) x > classify)
pc5 <- sapply(p5, function(x) x > classify)

# Find the sensitivity and specificity of knearest.
ksensitivity <- apply(kc5, 1, sensitivity, response)
kspecificity <- apply(kc5, 1, specificity, response)
psensitivity <- apply(pc5, 1, sensitivity, response)
pspecificity <- apply(pc5, 1, specificity, response)

plot(1 - kspecificity, ksensitivity, xlim=c(0.05,0.95), ylim=c(0.05,0.95), xlab="Specificity", ylab="Sensitivity", type='l')
lines(1 - kspecificity, ksensitivity, col="Orange") ; lines(1 - pspecificity, psensitivity, col="Purple")
legend(x = "bottomright", c("knearest", "kknn"), lty = c(1,1), lwd = c(2,2), col=c("Orange", "Purple"))
lines(0:1, 0:1, col="Red", xlim=c(0.05, 0.95), ylim=c(0.05,0.95))
title("ROC Curve for the K-NN Spam Predictors")
```

Listing 2: 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 3: 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 4: Inference Script for Machine Lifetime

```
source("likelihood.r") # sum of ln(p(x|theta)).
lifetimes <- read.csv("machines.csv") # Matrix?
parameter <- seq(0.1, 8.0, by=0.1) # Testing...
# Apply each parameter theta individually gives
# the log-likelihood for each of the parameters
p <- sapply(parameter,lnlikelihood,x=lifetimes)
average_lnlikelihoods <- p / dim(lifetimes)[1];
mle <- order(average_lnlikelihoods)[length(p)];
mle <- mle * 0.1 ; cat("MLE(theta): ",mle,"\n")

# Plot the relation between theta = logl.
plot(parameter,p, xlab="theta-parameter",
     ylab = "loge/post-likelihood", type = "l",
     xlim=c(0,8), ylim=c(-242,-2), col="orange")
points(mle, lnlikelihood(lifetimes, mle),
       col="orange", lwd=c(2, 2), pch="x");
title("Max Log/Posteriori-Likelihood Estimation")

lifetimes6 <- t(data.matrix(lifetimes)[1:6])
p6 <- sapply(parameter,lnlikelihood,x=lifetimes6)
average_lnlikelihoods6 <- p6 / dim(lifetimes6)[1]
mle6 <- order(average_lnlikelihoods6)[length(p6)]
mle6 <- mle6 * 0.1 ; cat("MLE(theta): ",mle6,"\n")

# Plot the relation between theta = ln-l.
par(new=TRUE) # Seems a little bit hacky.
plot(parameter,p6, xlab="theta-parameter",
     ylab = "loge/post-likelihood", type = "l",
     xlim=c(0,8), ylim=c(-242, -2), col="purple")
points(mle6, lnlikelihood(lifetimes6, mle6),
       col="purple", lwd=c(2, 2), pch="x")

po <- sapply(parameter, polikelihood, x=lifetimes)
average_polikelihoods <- po / dim(lifetimes)[1]
mpe <- order(average_polikelihoods)[length(po)]
mpe <- mpe * 0.1 ; cat("MPE(theta): ",mpe,"\n")

# Plot the relation between theta = po-li.
par(new=TRUE) # Seems a little bit hacky.
plot(parameter,po, xlab="theta-parameter",
     ylab = "loge/post-likelihood",type="l",
     xlim=c(0,8), ylim=c(-242, -2),
     col = "Green")
points(mpe, polikelihood(lifetimes, mpe),
       col="Green", lwd=c(2, 2), pch="x")
legend(x = "bottomleft", c("loge-likelihood, |x| = 48",
                           "loge-likelihood, |x| = 6",
                           "post-likelihood, |x| = 48"),
       lty = c(1,1), lwd = c(2,2),
       col=c("Orange", "Purple", "Green"))

random_exponential <- rexp(50, mle)
hist(random_exponential, main="Histogram over X~Exp of Full Log-Likelihood Estimation",
     xlab = "Expected Machine Lifetime", ylab="Frequency of Machine Lifetime")
```

Listing 5: Max Log- and Posteriori-Likelihood Estimation Formula

```
distribution <- function(x, theta) {
    # An exponential distribution.
    exponential <- exp((-theta)*x)
    return(theta*exponential)
}

lnlikelihood <- function(x, theta) {
    p <- log(distribution(x, theta))
    return(sum(p)) # log-likelihood.
}

polikelihood <- function(x, theta) {
    jp<-prod(distribution(x, theta))
    posterior <- 10*exp(-10*theta)
    return(log(jp*posterior))
}
```

Listing 6: The Given Machine Lifetime CSV Dataset (Excerpt)

```
Length
0.394761404022574
1.17680263974688
0.768466353536656
0.126129498627658
0.053941871673606
0.839961192069958
2.83505971839929
1.2602572965046
4.41829496504448
0.737917928761447
0.282883279724047
0.405573239549994
1.38489578934096
1.41224693846584
1.81130825686632
1.58639749349854
1.0564000190196
```

Introduction to Machine Learning
Individual Laboration Report –2–

Erik Sven Vasconcelos Jansson
erija578@student.liu.se
Linköping University, Sweden

November 15, 2016

Assignment 1

Reducing the amount of relevant features by using *feature selection* is an important task in *supervised machine learning*. Since many features $\mathcal{F}_i$ are more relevant than others, producing the optimal feature set $\hat{\mathcal{F}} \subseteq \mathcal{F}$, reducing $|\mathcal{F}|$, while also $\hat{\varepsilon}(\hat{\mathbf{y}}, \mathbf{y})$, the *Mean Squared Error (M.S.E)*. Here, we build a *brute-force feature selection* for *linear models*, using *k-folds cross validation* for ranking feature subsets.

First, every possible feature combination $\mathcal{F}_i \subseteq \mathcal{F}$ is generated. Thereafter, each $\mathcal{F}_i$ is tested through *k-fold cross-validation*, giving the *mean* $\hat{\varepsilon}_i(\hat{\mathbf{y}}, \mathbf{y})$ of the feature subset $\mathcal{F}_i$. By picking the feature subset $\mathcal{F}_i$ which produces $\min_i \hat{\varepsilon}_i(\hat{\mathbf{y}}, \mathbf{y})$, the best features are picked. In Listing 3 line `26` we generate all $\mathcal{F}_i$, which are then cross-validated in lines `30-31`, then in lines `34-43` the best feature subset $\hat{\mathcal{F}}$ is given by evaluating the errors $\min_i \hat{\varepsilon}_i(\hat{\mathbf{y}}, \mathbf{y})$, where $\hat{\mathcal{F}} = \mathcal{F}_i$.

Now, how *k-fold cross validation* works is shown. Roughly, Algorithm 1 demonstrates these steps, by giving each individual $\mathcal{F}_i$ and respective $X_{\mathcal{F}_i}$, $\mathbf{y}_{\mathcal{F}_i}$ as arguments, the *feature matrix* and *target vector*. For each $\mathcal{F}_i$ and *k-fold iteration*, a *linear hypothesis function* is trained, predicting $\hat{\mathbf{y}}$ using Equations 1. Thereafter, the *M.S.E* of the prediction $\hat{\varepsilon}_i(\hat{\mathbf{y}}, \mathbf{y})$, is calculated by using Equation 3. Finally, the *mean* of these $\hat{\varepsilon}_i(\hat{\mathbf{y}}, \mathbf{y})$ is the result of $\mathcal{F}_i$ cross-validation.

$$\hat{\mathbf{w}}_t = (X_t^\intercal X_t)^{-1} X_t^\intercal \mathbf{y}_t \tag{1}$$

$$\hat{\mathbf{y}}_v = X_v \hat{\mathbf{w}}_t \tag{2}$$

$$\hat{\varepsilon}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{3}$$

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

The implementation of *feature selection* is found under Listing 3, while *k-fold cross-validation* should be found in Listing 2, *linear regression* in Listing 1. Most information was derived from *Andrew Ng's Regularization & Model Selection Handouts* [Ng16].

Finally, testing the *swiss dataset* on our *feature selection* implementation where $\mathcal{F} = \mathcal{U} -$ Fertility, gives $\hat{\mathcal{F}} = \{3, 4, 5\}$, also shown in the Table 1 below. Our $\hat{\mathcal{F}}$ seems reasonable, being *Catholic* is usually attributed with *low abortion rate*, *Infant Mortality* is directly linearly related as can be seen in the plot.

Feature Selection	**M.S.E.**
$\{3, 4, 5\}$	90.6202
$\{1, 3, 4, 5\}$	111.411
$\{1, 2, 3, 5\}$	117.092

Table 1: where 1 → *Agriculture*, 2 → *Examination*, 3 → *Education*, 4 → *Catholic*, 5 → *Infant Mortality.*

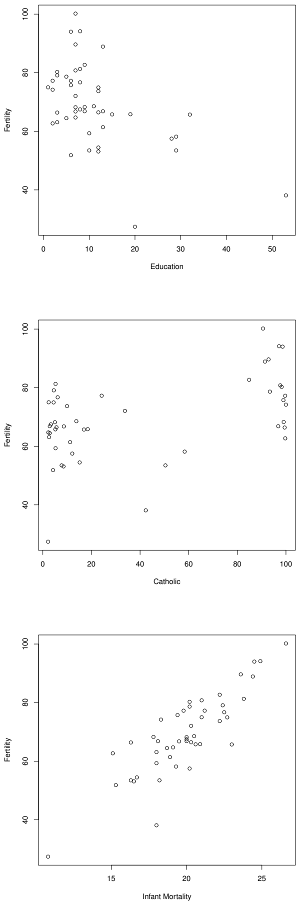
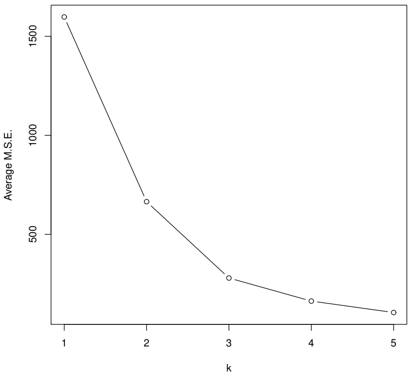


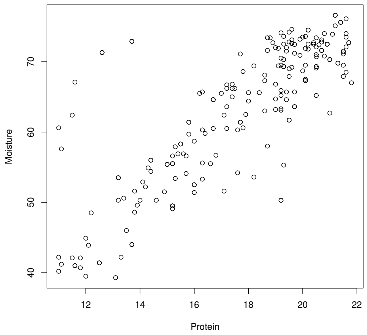
Figure 1: displays the relationships between feature subsets of *Size K* and their *Average M.S.E.* Notice that the *expected error decreases* by *increases in k.*



Assignment 2

By using the *tecator dataset*, produced from observations on *predicting fat content* simply by using an *infrared absorbance spectrum* (w. several channels), we try to find relationships between their features.

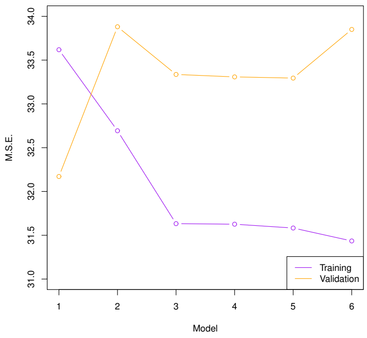
Figure 2: relationship seems very close to linear. *Protein* and *Moisture* seem to be related somehow, at least in the provided meat observation datasets.



Now we consider models M_i where *Moisture* is *normally distributed* and related to *Protein* with a *polynomial function* of steadily increasing degree. Equation 4 displays this relation, MSE in Figure 3.

$$\text{Moisture} \sim \mathcal{N}(w_0 + \sum_{i=1}^{6}(w_i^i \cdot \text{Protein}_i), \sigma^2) \quad (4)$$

Figure 3: increasing complexity for the i^{th} *Model* seems to generate *bias* towards the *training dataset*, motivating the higher *validation dataset* error rate.



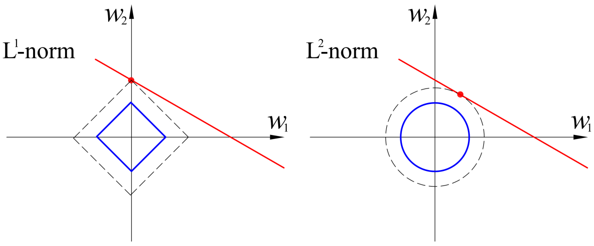
It seems the model *overfits* with more *complexity*. The models M_i and their *M.S.E.* are calculated in lines `24-39` in Listing 4, the assignment 2 scripts. Also, according to the plot, the best model M_i is located somewhere between $i = 1$ and 2, where the lines intersect, producing the lowest M.S.E. for *both datasets*, meaning there isn't *bias* towards either D.

Now we perform *feature/variable selection* on the *features Channel1-100* and *target/response* of *Fat*. By using *stepAIC* on a linear model generated with *R's lm*, a total of *64 variables* were selected here. This is done in lines `66-67` in Listing 4 (*w. MASS*).

By using the *Ridge regression model* on the same *features* and *response variables* with *glmnet library*. Figure 5 shows how *coefficients* relate to the *log* λ, *the log penalty.* Notice how all coefficients *converge uniformly* as *higher penalty* is added to the model. This is being done in line `72` with $\lambda = 0.0$ (Ridge).

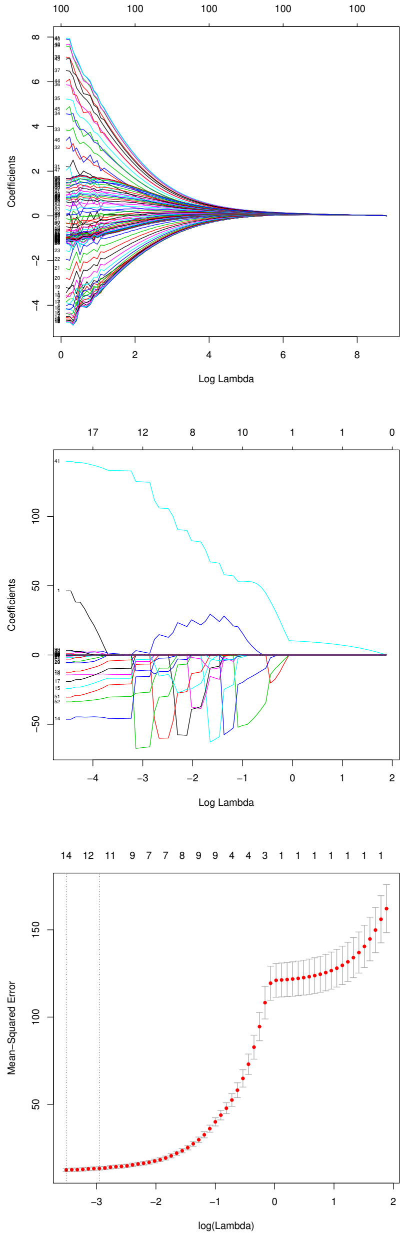
Thereafter, we use the *LASSO regression model* with the same *features and response variables.* See in Figure 5 how the *coefficients* converge *iteratively* instead of *simultaneously*, comparing toward *Ridge*. This can be explained graphically within Figure 4.

Figure 4: on the left we have *Lasso* and right *Ridge*. Notice, that *Lasso* can set $w_1 \rightarrow$ zero much faster. Distributed under CC by Rezamohammadighazi.



Finally, we *cross-validate* the *Lasso model* with *k-fold cross-validation* using the *glmnet library*. By using $k = 20$ and using *M.S.E.* as the metric, we find the last plot in Figure 5 which shows that the chosen features (the dotted interval) are those that produce the lowest M.S.E. Which are the 14 selected features. This is shown in the lines `96-97` of Listing 4. Lastly, we compare these results with those found with *stepAIC*. Notice how the *Lasso model* with *k-fold C.V.* produced a much lower number of selected features, 14, while *stepAIC()* produced a total of 64. *Lasso* gives us less features.

Figure 5: below are the relationships between the *coefficients* and the $log(\lambda)$ *penalty* for *Ridge* and *Lasso regression* (shown below in that exact order). Notice how *Ridge* converges all of the *coefficients simultaneously* while *Lasso* does this step *iteratively*, therefore, *Lasso regression* should converge "faster". Finally, the bottom plot displays how a *Lasso C.V.* relates to the increasing of *penalty factor* of $log(\lambda)$.



References

[HQ14] Trevor Hastie and Junyang Qian. *Glmnet Vignette*. `stanford.edu/~hastie/glmnet/`, 2014. [Online; on 15/11/2016].

[Ng16] Andrew Ng. *Handouts: Regularization and Model Selection*. Stanford University, 2016.

Appendix

Listing 1: Estimation of the Linear Regression Model with a Hat Matrix

```
# ------linrhat(X, y)------
# Predicts the parameters w
# for the given features X,
# and the targets y through
# the use of an hat matrix.
linrhat <- function(X, y) {
    # Nice hat good sir...
    return(solve(t(X)%*%X)
           %*% t(X)%*%y)
}
```

Listing 2: Implementation of a K-Fold Cross-Validation Algorithm for $\mathcal{M}$

```
source("linrhat.r")

# ------disjoin(X, k)------
# Produce a k disjoint sets
# of the training matrix X.
# Useful for kfoldcv below.
# Note: only returns index.
disjoin <- function(X, k) {
    row <- 1:nrow(X)
    folds <- nrow(X) / k
    # Resulting disjoints.
    S <- matrix(, k, folds)
    U <- c() # Picked sets.
    for (fold_row in 1:k) {
        D <- setdiff(row, U)
        Si <- sample(D, folds)
        S[fold_row,] <- Si
        U <- union(U, Si)
    }

    return(S)
}

# ------egerror(x, y)------
# Locate the generalization
# error within in our model
# by comparing the results:
# x and y targets. Returns:
# the difference for x & y.
egerror <- function(x, y) {
    targets <- length(x)
    # Using good old MSE...
    sdiff <- sum((x - y)^2)
    return(sdiff / targets)
}

# ------kfoldcv(X, y, k)-------
# Returns the estimated genera-
# lization error of the feature
# set according to a linear mo-
# del. It does this by applying
# an k-folding cross validation
# method, splitting X randomly,
# gives k disjoint subets of X.
kfoldcv <- function(X, y, k)  {
    kfolding <- 1:k
    sets <- disjoin(X, k)
    ege <- c() # Empty set of errors.
    for (i in kfolding) { # Every set.
        kset <- sets[-i,] # Remove ’i’
        iset <- sets[i,]  # Only  ’i’.
        # Pick dataset for all but ’i’
        Xi <- X[kset,] ; yi <- y[kset]
        # Estimated parameters w. Xi.
        hypothesis <- linrhat(Xi, yi)
        # Predict for the unused ’i’.
        p <- X[iset,]%*%hypothesis
        # Esimate error of this.
        e <- egerror(p, y[iset])
        # Add to list of these.
        ege <- c(ege, abs(e))
    }

    # Oh yea baby,
    # average errors.
    return(mean(ege))
}
```

Listing 3: Brute-Force Feature Selection to Find Lowest M.S.E. Subset

```
library("ggplot2")
source("kfoldcv.r")

# ------kfold(i, X, y, k)------
# Wrapper for applying kfoldcv,
# since it needs to apply rows.
kfold <- function(i, X, y, k) {
    Xi <- data.matrix(X[,i]);
    return(kfoldcv(Xi, y, k))
}

# ------featsel(X, y, k)------
# Applies feature selection to
# X, giving the best subset of
# X which minimizes the error.
# This is done by applying the
# k-fold cross validation, for
# each possible subset of X’s.
featsel <- function(X, y, k) {
    features <- c() # Nothing.
    lowerror <- Inf # Not good
    relation <- data.frame(K   = 1:k,
                           MSE = c(0))
    for (i in 1:k) { # Wrapper
        # Produce all combins.
        fi <- t(combn(1:k, i))
        # Apply combinations for
        # each of the featureset
        # giving the error value
        ei <- apply(fi, 1, kfold,
                    X, y, k)

        # Order by error...
        err <- ei[order(ei)]
        fea <- fi[order(ei),]
        fea<-data.matrix(fea)

        relation$MSE[i]<-mean(err)
        # Update best estimates.
        if (lowerror > err[1]) {
            lowerror <- err[1];
            features <- fea[1,]
        }
    }

    setEPS()
    postscript("kvsmse.eps")
    # Plot relation k v.s. M.S.E.
    plot(relation$K, relation$MSE,
         type="both", xlab="k",
         ylab = "Average M.S.E.")
    dev.off() # Write...
    # Best features.
    return(features)
}

set.seed(12345)
X <- data.matrix(swiss[,-1])
y <- data.matrix(swiss[,1]);
features <- featsel(X, y, 5)
X <- data.matrix(X[,features])
what <- linrhat(X, y)
yhat <- X %*% what

graph <- data.frame(X)
graph$Fertility <- yhat

setEPS()
postscript("education.eps")
plot(graph$Education, graph$Fertility,
   xlab="Education", ylab="Fertility")
dev.off()

postscript("catholic.eps")
plot(graph$Catholic, graph$Fertility,
   xlab="Catholic", ylab="Fertility")
dev.off()

postscript("mortality.eps")
plot(graph$Infant.Mortality, graph$Fertility,
   xlab="Infant Mortality", ylab="Fertility")
dev.off()
```

Listing 4: Script for Assignment 2 on Linear Models, Ridge, Lasso and Cross-Validation

```
library("MASS")
library("glmnet")
data <- read.csv("tecator.csv");
data_matrix <- data.matrix(data)

setEPS()
postscript("linear.eps")
plot(data$Protein, data$Moisture,
     xlab="Protein", ylab="Moisture")
```

```
dev.off()

# Do you think these data are described well by a linear model?
# Answer: yes, definitely. Both data are rising similar ratios.
# Moisture ~ N(mu = Xw, sigma^2), a linear model should work...
# Moisture_hat ~ Mi = polynomial function dependent on Protein:
#                     w0 + x1*w1 + x2*w2^2 + ... xi*wi^i = f(x)

set.seed(12345) # Set seed for getting same results...
indices <- sample(1:nrow(data), floor(nrow(data)*0.5))
training <- data[indices,] # Subset for training data.
validation <- data[-indices,] # Subset for validation.
validation <- validation[-nrow(validation),] # Shhh...

polynomials <- 1:6
imse <- data.frame(Model = polynomials,
                   Training.MSE = c(0.0),
                   Validation.MSE = c(0))
for (degree in polynomials) {
    model <- lm(Moisture ~ poly(Protein, degree),
                training)
    tprediction <- predict(model, training)
    vprediction <- predict(model, validation)
    vmse <- (vprediction - validation$Moisture)^2
    tmse <- (tprediction - training$Moisture)^2
    vmse <- mean(vmse) # MSE for validation set.
    tmse <- mean(tmse) # MSE for training set...
    imse$Validation.MSE[degree] = vmse # For i.
    imse$Training.MSE[degree] = tmse # For i.
}

setEPS()
postscript("depends.eps")
plot(imse$Model, imse$Training.MSE,
     xlab="Model", ylab="M.S.E.", "b", col="purple",
     ylim=c(31.0, 34.0))
points(imse$Model, imse$Validation.MSE, col="orange",
     type = "b", ylim=c(31.0, 34.0))
legend("bottomright", legend=c("Training", "Validation"),
       col=c("purple", "orange"), lty=1)
dev.off()

# Which model is best according to the plot?
# Answer: model where i = 1, since the validation predictions
# are becoming more erronous as i gets larger, while training
# becomes more accurate (this is caused by overfitting data).
# How do the M.S.E. values change and why?
# Answer: the error for the training set seems to become less
# as i gets larger while validation becomes more error prone.
# This is caused because the model overfits,  with a bias to-
# wards the training set (since the model is based on these).
# Ppecifically, it overfits:  the models become more complex.
# Interpret this picture in terms of bias-variance tradeoffs.
# Answer: it seems to be biased towards the training dataset.
# Since the E[yhat(x0) - f(x)] isn't close to zero, biased...

model <- lm(Fat ~ . - Moisture - Protein, data)
feature_selection <- stepAIC(model, direction="both")
# How many were selected? Answer: 64,  coeffiecients.

y <- data_matrix[,102] # Select Fat as the only response...
X <- data_matrix[,2:101] # Select Channel1-Channel-100 features.
```

```
ridge <- glmnet(X, y, alpha = 0) # Fit with the Ridge regression.
lasso <- glmnet(X, y, alpha = 1) # Fit with the Lasso regression.

setEPS()
postscript("ridge.eps")
# Explicit about x-axis variable.
plot(ridge, xvar="lambda", label=TRUE)
dev.off()

# Report on how the coeffiecients change with lambda.
# Answer: ridge penalizes all features equally with lambda.
# Therefore, it will take longer to coverge all features...

setEPS()
postscript("lasso.eps")
# Explicit about x-axis variable.
plot(lasso, xvar="lambda", label=TRUE)
dev.off()

# Conclusions on the Ridge vs Lasso resulting plots?
# Answer: the Lasso regression seems to converge individual
# features, therefore converging faster, towards the values
# of some features. While Ridge converges simultaniously...

kfoldcv <- cv.glmnet(X, y,  type.measure="mse", nfolds=20)
feature_selection <- coef(kfoldcv, s = "lambda.min")

# Report the optimal lambda and how many variables were selected.
# Answer: the optimal lambda was 0.02985605 and 14 were selected.
# Conclusions: the interval area shown in the graph shows optimal
# number of feature selections,  basically with the lowest M.S.E.

setEPS()
postscript("kfold.eps")
plot(kfoldcv)
dev.off()

# Compare the results from steps (4) and (7). Basically, compare
# stepAIC() and the glmnet CV (using k-fold).    Answer: stepAIC
# gives a lot of features, 64 for the selection, while glmnet cv
# for Lasso gives 13 features. Therefore, stepAIC() seems to not
# penalize the features very well, and therefore chooses more...
```

Introduction to Machine Learning Individual Laboration Report –3–

Erik Sven Vasconcelos Jansson
erija578@student.liu.se
Linköping University, Sweden

November 27, 2016

Assignment 1

Now that *linear regression* and *cross-validation* have been studied for *regression problems* (involving continuous target variables), the question is how to apply the same concept to *classification problems* (which involve a discrete amount of target variables). Both *Linear Discriminant Analysis (LDA)* and *Logistic Regression* can be used to achieve this, and also to draw *decision boundaries.*

In this task the goal is to *classify* the *Sex* of *Australian Crabs* by using their *Carapace Length* and *Rear Width* using *LDA* and thereafter plotting the *decision boundary.* Plotting the data gives Figure 1. As can be seen, a decision boundary seems to exist.

Firstly, the *linear parameters* w_0 and $\boldsymbol{w}$ need to be found for each class k, essentially creating our *target hypothesis function* for each one of these k. Finding these parameters is done with Equations 1, essentially computing the mean $\hat{\boldsymbol{\mu}}_k$ for each class k which is used to derive the *covariance matrices* Σ_k.

$$\begin{aligned} \hat{\pi}_k &= \frac{N_k}{N} \\ \hat{\mu}_k &= \frac{1}{N_k}\sum_{i \in k} \boldsymbol{x}_i \\ \Sigma_k &= \frac{1}{N_k}\sum_{i \in k} (\hat{\boldsymbol{\mu}}_k - \boldsymbol{x}_i)(\hat{\boldsymbol{\mu}}_k - \boldsymbol{x}_i)^{\intercal} \\ \hat{\Sigma} &= \frac{1}{N}\sum_{k \in K} N_k \Sigma_k = \sum_{k \in K} \frac{\operatorname{cov} X_k}{|X_k|} \end{aligned} \tag{1}$$

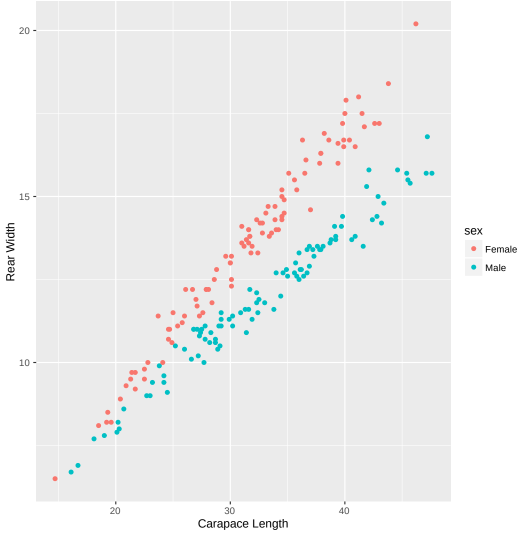
$$\begin{aligned} w_k &= -\frac{\hat{\boldsymbol{\mu}}_k^{\intercal}}{2}\hat{\Sigma}^{-1}\hat{\boldsymbol{\mu}}_k + \log \hat{\pi}_k \\ \boldsymbol{w}_k &= \hat{\Sigma}^{-1}\hat{\boldsymbol{\mu}}_k \end{aligned} \tag{2}$$

Finally, computing $\hat{\Sigma}$ is done by using these Σ_k, which is thereafter used to calculate w_0 and $\boldsymbol{w}$, $\forall k$. Implementation of these formulae in *R* can be found in Listing 1 under lines `15-39` in the `lda` function. All equations are in *Friedman et al's* [FHT09] book.

$$\delta_k(\boldsymbol{x}) = w_0 + \boldsymbol{w} \tag{3}$$

Using the *Discriminant Function* in Equation 3 above, one can plot the line through each class k. The *decision boundary* is in-between these 2 lines. In lines `58-61` we calculate the *intercept* and the *slope* of the *decision boundary*, which gives us all the information needed to produce a new Figure 2. The fit seems perfect, exactly as previous Figure 1.

Figure 1: Sex Classifications of Australian Crabs



Thereafter, by using *Logistic Regression* instead, Figure 3 can be obtained, which misses predictions.

Class	$\boldsymbol{w_0}$	$\boldsymbol{w}$
Female	-22.42	(-2.161, 8.248)
Male	-12.42	(-0.213, 2.565)

Table 1: Predicted Parameters for Australian Crab

Figure 2: Crab Sex Decision Boundary using LDA

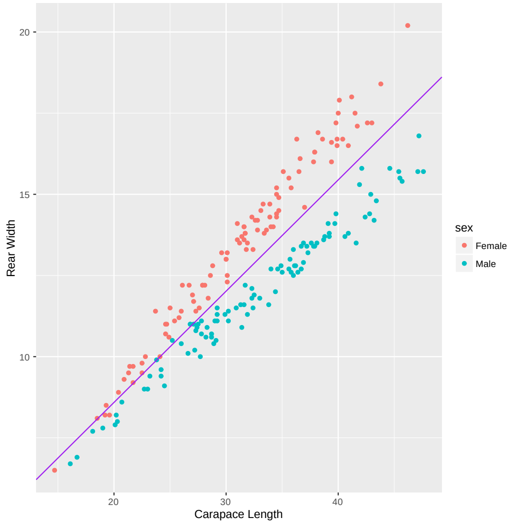
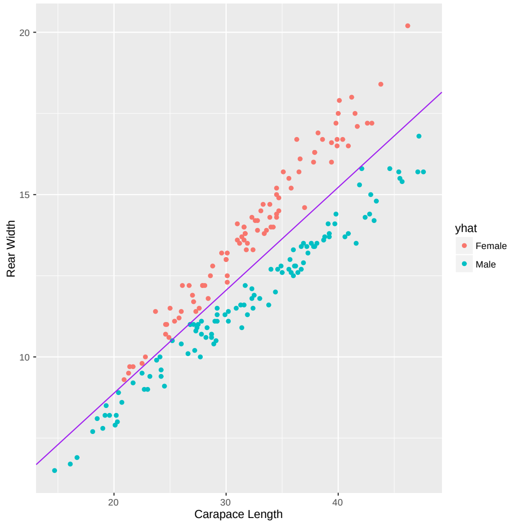


Figure 3: Crab Sex Decision Boundary using LR



Assignment 2

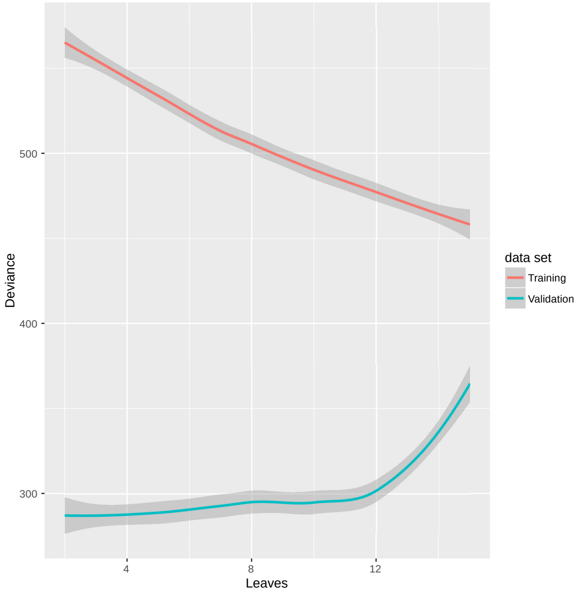
Given several *observations* from different *customers* with certain *features* and *classifications* of whether they have *managed their loans good* or *bad*, we are tasked to *predict* if a new customer with certain set of features, will pay back their loans on time or not.

By using *decision tree learning*, which maps observations (the branches) to their conclusions (the leaves), we can predict this aforementioned model. In Listing 2 under the lines `26-45`, we use `tree` to fit our model for usage on *training* & *testing* sets. These models can be fit using different measures of *impurity*, where we only consider *deviance* and the *gini index* here. In Table 2 are the results for these. It seems that *gini and/or deviance* classifies better.

Data Set	**Impurity Measure**	**Miss Rate**
Training	Deviance	0.212
Testing	Deviance	0.236
Training	Gini	0.230
Testing	Gini	0.282
Training	Deviance & Gini	0.212
Testing	Deviance & Gini	0.236

Table 2: Decision Tree Missclassification Rates

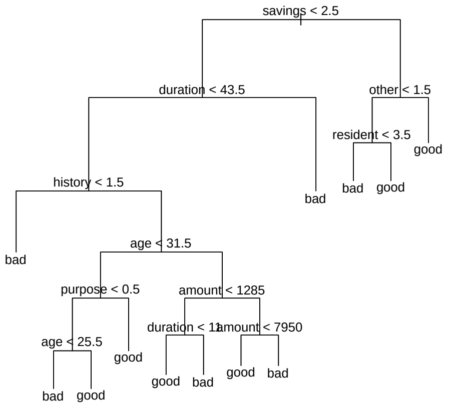
Figure 4: Training and Validation Deviance Values



Afterwards, we use the *training* and *validation data sets* to choose the *optimal tree depth.* Following lines `47-66`, we iteratively *prune* the decision tree, essentially adding more *leaves*, and thereafter plot the dependence on the *number of leaves* and the *estimated deviance of the model.* This plot can be seen in Figure 4. As can be seen, when more than 12 leaves are selected, the *validation set* has a pretty large increase in *deviance*, which isn't the desired behaviour. On the other hand, the *training data set* has a constant decrease in deviance as the number of leaves increase. Therefore, the best number of leaves seems to be 12, giving best of both worlds. Additionally, the depth of this optimal tree is 6 and has a misclassification rate of 0.24 with the following features selected: *savings, duration, history, age, purpose, amount, other and resident*, all of which can be found easy with `summary(tree)`.

Finally, the decision tree is plotted in Figure 5. Most of the branches and their outcomes seem quite reasonable (with common sense), a person that has very few savings and has a history of usually delays loans, will probably be a bad customer who doesn't pay back them. While a person which has savings and is a resident will probably be more responsible and pay back his loans. It seems that the tree has been able to select the most important features as the branches of the decision tree, which are highly correlated with either classification result (the leaves). The depth can be seen here too.

Figure 5: Visualization of the Decision Tree



Now we proceed to fit another model with the same situation, but using *Naïve Bayes* instead, which basically assumes that all *features* are independent, like so: $p(C_k|x_1, x_2, ..., x_n) \propto p(x_i|C_k)$.

Under lines `82-90` we fit the model with `e1071`'s `naiveBayes` *classification*, and produce the *confusion matrices* and *missclassifications* below for both the *training* and the *testing data sets.*

Missclassification: train = 0.300 & *test* = 0.306.

train	**bad**	**good**
bad	95	98
good	52	255

test	**bad**	**good**
bad	142	147
good	83	378

Table 3: Normal Naïve Bayes Confusion Matrix

It seems that in this case, *Naïve Bayes* performs worse than *decision trees* since the missclassification rate is quite a bit higher. Now, let's assume that we are given a *loss matrix*, a way of *penalizing* the *predictor* to apply weight to certain negatives, which in this case is $L = \left(\begin{smallmatrix} 0 & 1 \\ 10 & 0 \end{smallmatrix}\right)$. Doing this in lines `92-109` results in a prediction giving the *confusion matrix* below and missclassifications of train = 0.274 and test = 0.278. These missclassifications are a lot lower than those in the normal "lossless" model. The reason why this happens is because the cost of doing a faulty classification is highly penalized in one case (while the other isn't penalized as much), this is a underlying reason for these results.

train	**bad**	**good**
bad	27	17
good	120	336

test	**bad**	**good**
bad	40	24
good	185	501

Table 4: "Lossy" Naïve Bayes Confusion Matrix

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: Linear Discriminant Analysis Assignment

```
library("glmnet")
library("ggplot2")
library("grDevices")

mu <- function(X)  { return(colMeans(X)) }
softmax <- function(X, wi, wj) {
    X <- data.matrix(X)
    ihypothesis <- exp(X %*% wi[-1] + wi[1])
    jhypothesis <- exp(X %*% wj[-1] + wj[1])
    jhypothesis <- jhypothesis +
                   exp(X %*% wi[-1] + wi[1])
    return(ihypothesis / jhypothesis)
}

lda <- function(X, y) {
    classes <- levels(y) # Only c = 2
    class1  <- which(y == classes[1])
    class2  <- which(y == classes[2])
    X1     <- data.matrix(X[class1,])
    y1     <- data.matrix(y[class1]);
    X2     <- data.matrix(X[class2,])
    y2     <- data.matrix(y[class2]);

    mu1 <- mu(X1) ; mu2 <- mu(X2)
    pi1 <- length(y1) / length(y)
    pi2 <- length(y2) / length(y)
    sigma <- cov(X1)*nrow(X1)+
             cov(X2)*nrow(X2)
    sigma <- sigma/nrow(X)

    w01 <- -0.5 * mu1 %*% solve(sigma) %*% mu1 + log(pi1)
    wx1 <- solve(sigma) %*% mu1 # Some sort of weird magic.
    w1 <- matrix(c(w01, wx1), 1, 3)

    w02 <- -0.5 * mu2 %*% solve(sigma) %*% mu2 + log(pi2)
    wx2 <- solve(sigma) %*% mu2 # Some more magic here too.
    w2 <- matrix(c(w02, wx2), 1, 3)
    return(rbind(w1, w2)) # w1, w2.
}

classify <- function(X, d) {
    return(d[1] + d[2]*X[,1] +
                  d[3]*X[,2])
}

crabs <- read.csv("crabs.csv")
X <- crabs[,c("CL", "RW")]
y <- crabs[,c("sex")]

setEPS()
cairo_ps("crabs.eps")
print(qplot(CL, RW, data = crabs, color = sex,
      geom = c("point"),
      xlab = "Carapace Length",
      ylab = "Rear Width"))
dev.off()

```

```
parameters <- lda(X, y)
difference <- parameters[1,]-parameters[2,]
intercept <- difference[1] / difference[3]
slope <- difference[2] / difference[3]

sex <- classify(X, difference) > 0.0
sex[sex == FALSE] = "Female"
sex[sex == TRUE] = "Male"

setEPS()
cairo_ps("boundarylda.eps")
print(qplot(CL, RW, data = crabs, color = sex,
      geom = c("point"),
      xlab = "Carapace Length",
      ylab = "Rear Width") +
      geom_abline(intercept = -intercept,
                  slope = -slope, colour=’purple’))
dev.off()

fit <- cv.glmnet(data.matrix(X), data.matrix(y),
    family = "binomial", type.measure = "class")
yhat <- predict(fit, data.matrix(X), type="class")

setEPS()
cairo_ps("boundarylr.eps")
print(qplot(X$CL, X$RW, color = yhat,
      geom = c("point"),
      xlab = "Carapace Length",
      ylab = "Rear Width") +
      geom_abline(intercept = -coef(fit)[1] / coef(fit)[3],
                  slope = -coef(fit)[2] / coef(fit)[3],
                  colour=’purple’))
dev.off()

cat("Decision boundary with linear discriminant analysis:",
    -intercept, "+", -slope, "* k\n")
cat("Decision boundary with linear regression:",
    -coef(fit)[1] / coef(fit)[3], "+",
    -coef(fit)[2] / coef(fit)[3], "* k\n")
```

Listing 2: Decision Trees and Naïve Bayes Assignment

```
library("tree")
library("ggplot2")
library("reshape2")
library("grDevices")
library("e1071")

set.seed(12345) # As always.....
scores <- read.csv("scores.csv")
n <- nrow(scores) # Observation.
samples <- sample(1:n,  n / 2.0)
others <- setdiff(1:n, samples)
halves <- sample(others, n/4.0)

training <- scores[samples,]
trainingX <- training[,-ncol(training)]
trainingy <- training[,ncol(training)]

validation <- scores[halves,]
validationX <- validation[,-ncol(validation)]
```

```r
validationy <- validation[,ncol(validation)]

testing  <- scores[-halves,]
testingX <- testing[,-ncol(testing)]
testingy <- testing[,ncol(testing)]

fit <- tree(good_bad ~ ., data = training, split = c("deviance"))
training_prediction <- predict(fit, trainingX, type= "class")
testing_prediction <- predict(fit, testingX, type="class")
cat("Missclassifications only with deviance impurity: (",
    mean(training_prediction != trainingy), "," ,
    mean(testing_prediction != testingy), ")\n")

fit <- tree(good_bad ~ ., data = training, split = c("gini"))
training_prediction <- predict(fit, trainingX, type= "class")
testing_prediction <- predict(fit, testingX, type="class")
cat("Missclassifications only with the gini impurity: (",
    mean(training_prediction != trainingy), "," ,
    mean(testing_prediction != testingy), ")\n")

fit <- tree(good_bad ~ ., data = training, split = c("deviance", "gini"))
training_prediction <- predict(fit, trainingX, type= "class")
testing_prediction <- predict(fit, testingX, type="class")
cat("Missclassifications only with deviance and gini: (",
    mean(training_prediction != trainingy), "," ,
    mean(testing_prediction != testingy), ")\n")

max_depth <- 15
training_deviance <- rep(0, max_depth)
validation_deviance <- rep(0, max_depth)
for (depth_level in 2:max_depth) {
    pruned <- prune.tree(fit, best = depth_level)
    pred <- predict(pruned, validation, type="tree")
    training_deviance[depth_level] <- deviance(pruned)
    validation_deviance[depth_level] <- deviance(pred)
}

deviances <- data.frame(2:max_depth, training_deviance[-1], validation_deviance[-1])
colnames(deviances) <- c("Leaves", "Training", "Validation")
collapsed_deviances <- melt(deviances, id="Leaves")

setEPS()
cairo_ps("deviance.eps")
# It seems depth 12 is good, since validation goes hayware after that...
print(ggplot(data=collapsed_deviances, aes(x=Leaves, y=value, color=variable)) +
      geom_smooth() + labs(x="Leaves", y="Deviance", color="data set"))
dev.off()

# The final tree has depth 6, see output of `final_tree`.
# The parameters chosen are: savings, duration, history, age,
# purpose, amount, other, resident, in `summary(final_tree)`.
final_tree <- prune.tree(fit, best = 12) # The best choice...
prediction <- predict(final_tree, testing, type = "class")
cat("Missclassification for the optimal tree depth: (",
    mean(prediction != testingy), ")\n")

fit <- naiveBayes(good_bad ~ ., data = training)
training_prediction <- predict(fit, training, type = "class")
testing_prediction <- predict(fit, testing, type = "class")
cat("Missclassifications using Naive Bayes method:  (",
    mean(training_prediction != trainingy), "," ,
    mean(testing_prediction != testingy), ")\n")
```

```
cat("\nConfusion matrices for using Naive Bayes:\n")
print(table(training_prediction, trainingy))
print(table(testing_prediction, testingy))
```

Introduction to Machine Learning
Individual Laboration Report –4–

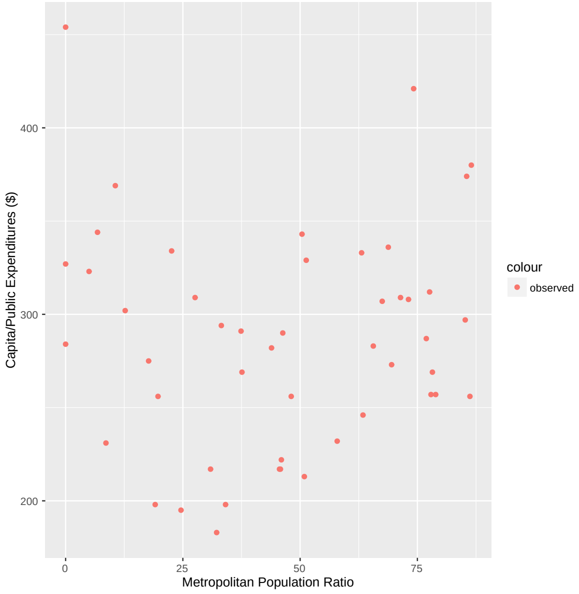
Erik Sven Vasconcelos Jansson
erija578@student.liu.se
Linköping University, Sweden

December 4, 2016

Assignment 1

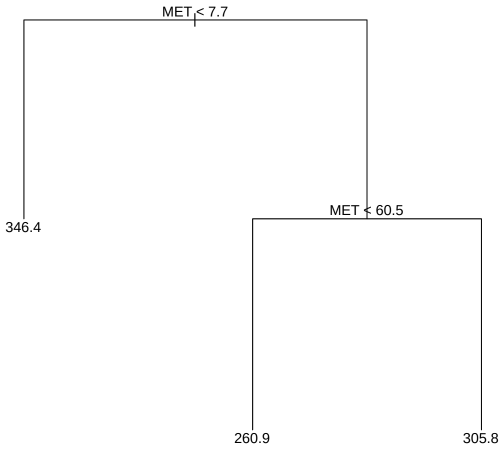
We are given the data set *State*, where observations regarding the *metropolitan ratio* and the *local public expenditure* for several *states* exist. Plotting these against each other gives Figure 1. Notice that the data is quite spread out, and that there doesn't seem to be any easily visible pattern unfortunately.

Figure 1: Metropolitan Ratio vs Expenditures ($)



For this task we are recommended to use *regression trees* as the *predictor* for the above situation. We fit the model using all observations, and thereafter *cross-validate* the model to find the optimal number of leaves for the *decision tree*. See Figure 2.

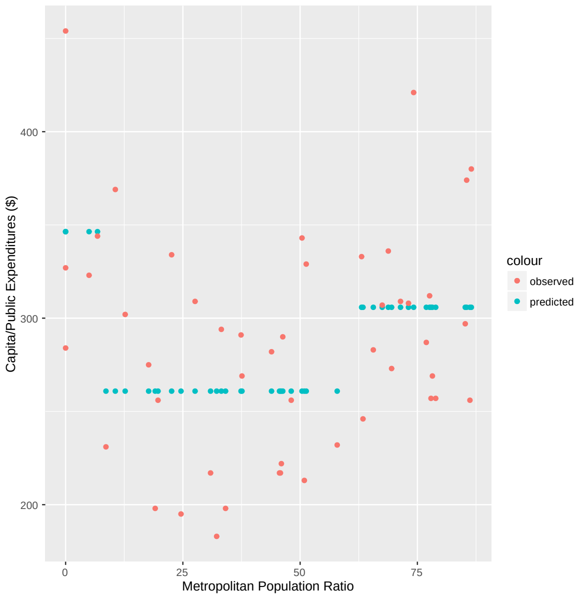
Figure 2: Optimal Regression Tree Predictor



Notice that *three leaves* have been selected, therefore, the *optimal regression tree* is found by *pruning* the original tree and obtaining the above. This is done in Listing 1 under source lines `20-30`.

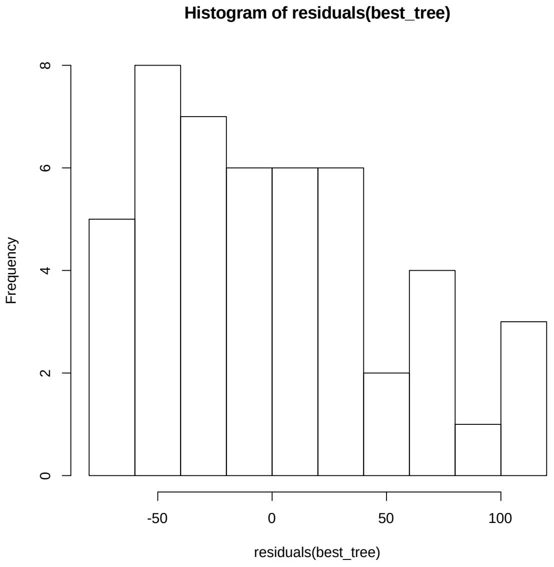
Thereafter, we predict the observed data using our optimal regression tree. Plotting these against the original observations gives Figure 3. Notice how the predictor has given estimates that are roughly the *mean* of each "bucket" selected by the regression tree. Therefore, the selected model is not expected to perform very good since it does not account for the noise present in the model very well, but will predict some correctly, since it's around the mean.

Figure 3: Regression Tree Predicted Expenditure



Finally, we plot the *residuals* from the *regression tree model*, which are basically the *distance* between the *observed responses* and the *predicted responses*. See Figure 4, where we have plotted a histogram of the residuals. Notice that it *doesn't* resemble a *bell curve*, meaning the *error isn't normally distributed*.

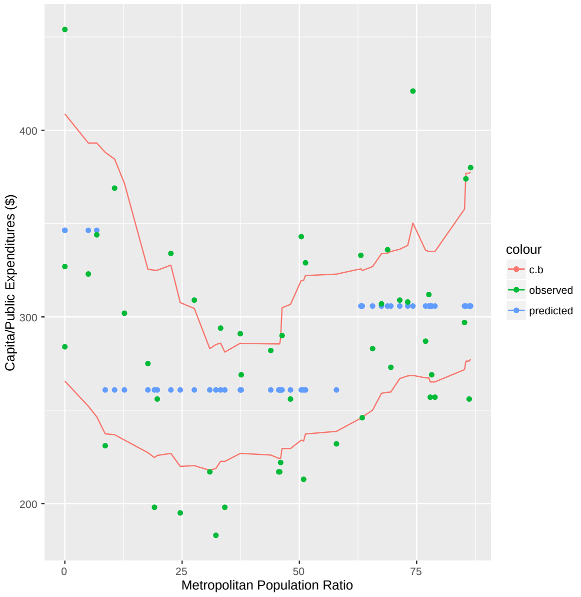
Figure 4: Histogram of Prediction Residuals



Since the chosen model doesn't seem particularly good, at least for these observations, we want to find *more information* about our estimator. Using *bootstrap* seems like a good idea. *Bootstrapping* is used to *estimate the properties* of an estimator by *re-sampling* from a given *approximated distribution*.

Because we don't know the underlying distribution, we want to use *non-parametric bootstrapping*. It works by *re-sampling observations* with *replacement*, the distribution is then calculated by using: $\hat{f}(D_1), \hat{f}(D_2), ..., \hat{f}(D_B)$, where $\hat{f}$ is our estimator. We use the `boot` function in *R*, re-sampling 1024 times. See Listing 1 lines `49-64` for the algorithm.

Figure 5: Non-Parametric Bootstrap C.B.

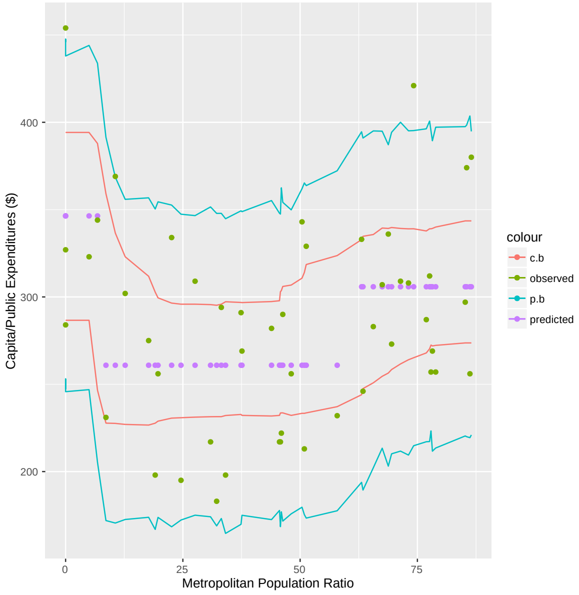


According to Figure 5 which displays the plot of the *confidence bands* of our estimator, several of our observations fall outside the 95% confidence level, which means that our estimator isn't good at all. The curve also seems to be quite bumpy, because the predictor only jumps between the three leaves.

Now, we are given a precondition: $Y \sim \mathcal{N}(\mu_i, \sigma^2)$ which means the *target* is now *normally distributed*. In light of this, we can use *parametric bootstrapping*, since the distribution has now been assumed to be known. It functions by re-sampling from the given distribution, which is different from before since we were only taking samples from the original data set, while we now generate entirely new, fresh, samples.

By doing this in Listing 1 under lines `82-114` we retrieve both the *confidence* and *prediction bands*. Both of these bands are plotted in Figure 6 below.

Figure 6: Parametric Bootstrap C.B. + P.B.



Notice, that the *prediction band* covers most of the observations, as opposed to the *confidence band*. *Prediction bands* account for the *noise* in the data. It seems reasonable that around 5% of the observations fall outside the *prediction band*, since we are targeting a 95% level of confidence (or by $\alpha = 5\%$). In this case, the model which we have predicted seems reasonable, since most predictions are right. *However*, this is probably not true, since Figure 4 shows a distribution that is most likely not normal, therefore *non-parametric bootstrapping* seems more accurate in this particular case (also by intuition).

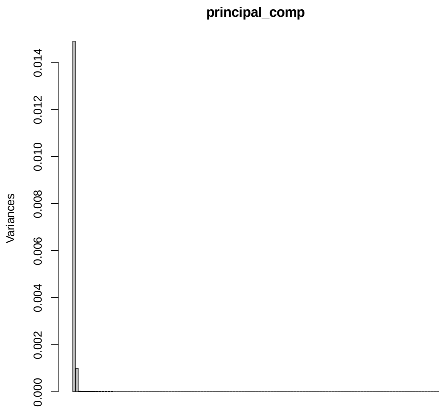
Assignment 2

Here, we are given a data set containing measures of *near-infrared spectra* and their *viscosity level* for a collection of *diesel fuels*. There are a lot of *features*.

Principal Component Analysis (PCA henceforth) is a *dimensionality reduction* technique, where the goal is to find a set of *principal components* where the given *observations* are might to be correlated. Our task is to find the *principal components* of the given data set accounting for 99% of the variance.

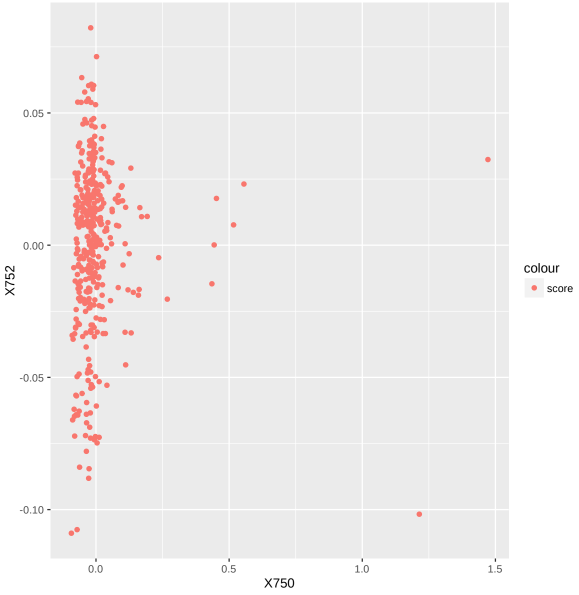
By using the built-in `prcomp` function, we apply *PCA* in Listing 2 under lines `7-17`. Afterwards in Figure 7 we produce a *scree plot*, which tells us that *two features* seem to account for all of the variance. More accurately, a bit above 99% of the variance...

Figure 7: Scree Plot for PCA



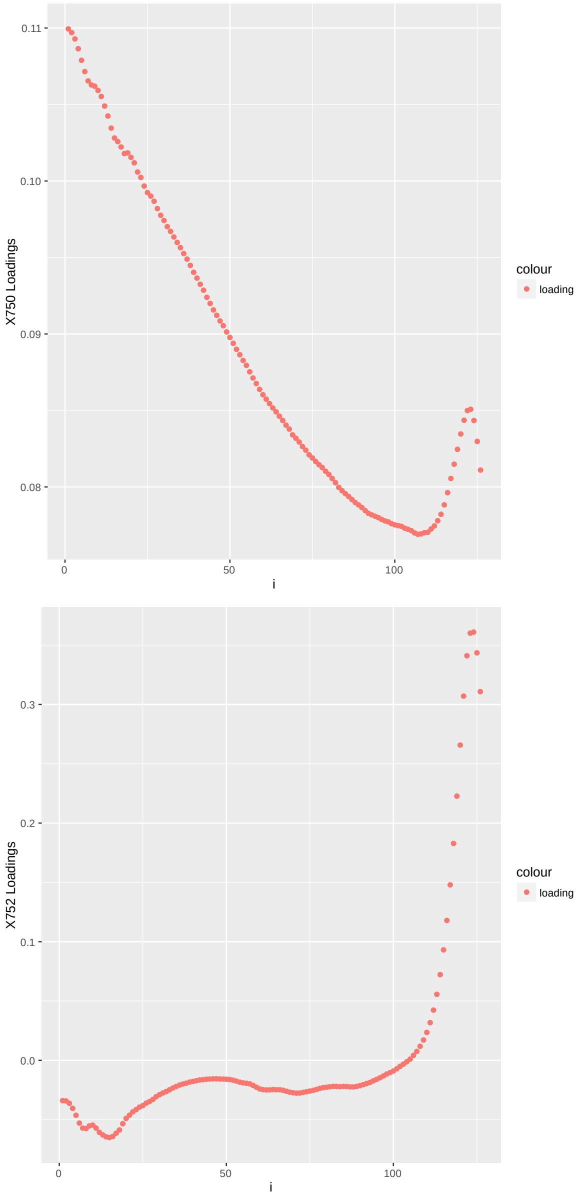
Additionally, the chosen features *X750* and *X752* are plotted against each other, giving the Figure 8. Notice that there are a couple of outliers in [1.0, 1.5] which can be classified as being *unusual diesel fuels*.

Figure 8: Score for PC1 and PC2



Plotting the so called *loadings* of these principal components gives Figure 9. A high *loading* implies that there is a *strong correlation* for a given feature.

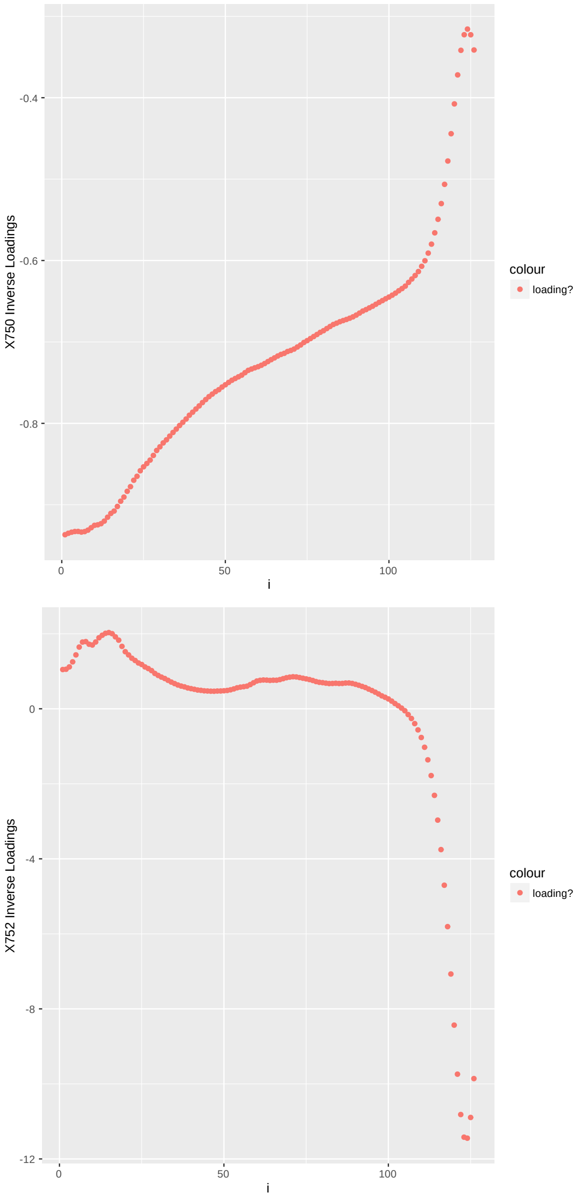
Figure 9: Trace Plot of PCA Loadings



Notice that both PC1 and PC2 seem to have a correlated feature(s) that spike in loading amount, while the rest seem to less significant in comparison. This might be a good candidate for a third principal component if we required to have higher variance.

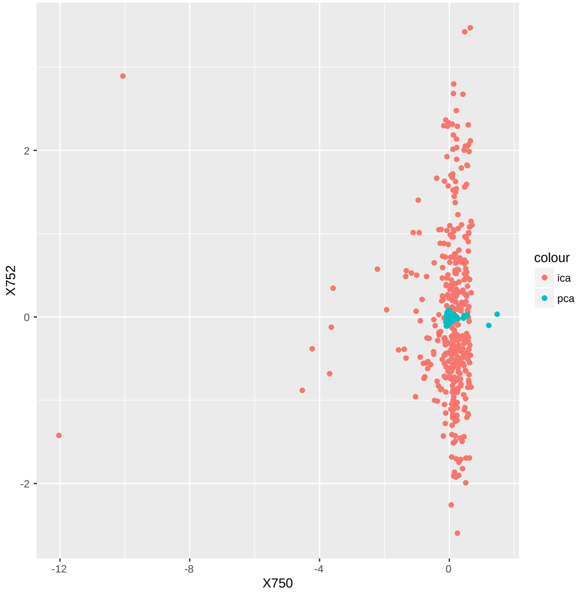
Now we apply *Independent Component Analysis* instead, which assumes the given components are statistically independent. We do this in Listing 2 under lines `45−68`, and produce the *trace plot* for these in Figure 10. The contents of these plots are similar to those in Figure 9, which are *loadings*, but are inverted, since we are measuring *independence* instead of *correlation*, the opposite PCA measure. Finally, we plot the score(s) of these in Figure 11.

Figure 10: Trace Plot of ICA Loadings



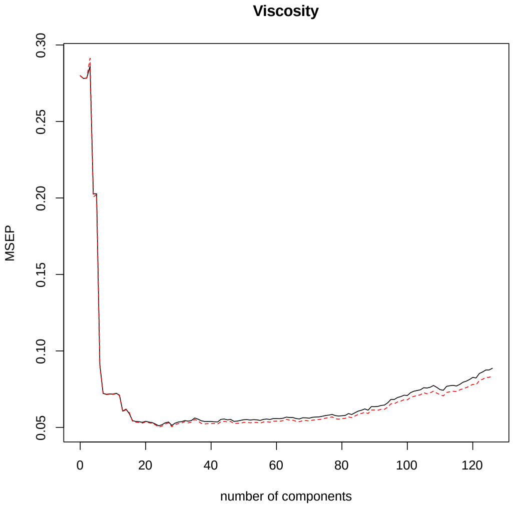
Notice that the direction of the score has been swapped, the underlying reason for this is similar to those previously mentioned regarding loadings. It's interesting to note that the units are different.

Figure 11: Score for PCA and ICA



Finally, we apply *cross-validation* in Figure 11, and note that the ranges of $[10, 18]$ p.c. components produces least *MSE*, in relation to the p.c. amount.

Figure 12: Cross-Validation for PCA



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 Regression Tree Model

```
library("tree")
library("boot")
library("ggplot2")
library("grDevices")

setEPS() # Enable save to eps.
state <- read.csv2("state.csv")
# Reorder data according to $MET.
state <- state[order(state$MET),]

cairo_ps("state.eps")
# Plotting MET vs EX. See file.
print(qplot(MET, EX, data = state,
    xlab = "Metropolitan Population Ratio",
    ylab = "Capita/Public Expenditures ($)",
    geom = c("point")))
dev.off()

set.seed(12345) # Required for cross-validation.
# Make sure that there are at least 8 leaves now.
control <- tree.control(nrow(state), minsize = 8)
# Fit our model by using regression trees (8 leaves).
fit <- tree(EX ~ MET, data = state, control = control)
optimal <- cv.tree(fit) # Do k-fold cross-validation.
least_deviance_index <- which.min(optimal$dev)
leaves <- optimal$size[least_deviance_index]
best_tree <- prune.tree(fit, best = leaves)
# Gives "best" tree according to k-fold cv.
yhat <- predict(best_tree, newdata = state)
# Predict EX by using best regression tree.

cairo_ps("tree.eps")
plot(best_tree)
text(best_tree)
dev.off()

cairo_ps("predicted_state.eps")
# Plotting MET vs EX. See the file.
print(qplot(MET, yhat, data = state,
    xlab = "Metropolitan Population Ratio",
    ylab = "Capita/Public Expenditures ($)",
    geom = c("point")))
dev.off()

cairo_ps("histogram.eps")
hist(residuals(best_tree))
dev.off()

bootstrap_predictor <- function(data, indices) {
    sample <- data[indices,] # Pick a subset of data.
    control <- tree.control(nrow(sample), minsize = 8)
    # Fit our model by using regression trees (8 leaves).
    fit <- tree(EX ~ MET, data = sample, control = control)
    leaves <- optimal$size[least_deviance_index]
    best_tree <- prune.tree(fit, best = leaves)
    # Gives "best" tree according to k-fold cv.
    yhat <- predict(best_tree, newdata = data)
    return(yhat) # Prediction from subset.
}

set.seed(12345) # Required for the bootstrapping.
# Apply non-parametric bootstrap to our regression tree
# model, picking out 1024 different indices from state.
bootstrap <- boot(state, bootstrap_predictor, R = 1024)

cairo_ps("bootstrap.eps")
plot(bootstrap)
dev.off()

# Find the confidence bands.
bands <- envelope(bootstrap)

cairo_ps("bands.eps")
# Plotting MET vs EX. See the file.
print(qplot(MET, yhat, data = state,
    xlab = "Metropolitan Population Ratio",
    ylab = "Capita/Public Expenditures ($)",
    geom = c("point")) + geom_line(data = state, aes(x = MET, y = bands$point[1,], col = "c.b")) +
                         geom_line(data = state, aes(x = MET, y = bands$point[2,], col = "c.b")))
dev.off()

bootstrap_prediction <- function(data) {
    control <- tree.control(nrow(data), minsize = 8)
    # Fit our model by using regression trees (8 leaves).
    fit <- tree(EX ~ MET, data = data, control = control)
    leaves <- optimal$size[least_deviance_index]
    best_tree <- prune.tree(fit, best = leaves)
    # Gives "best" tree according to k-fold cv.
    yhat <- predict(best_tree, newdata = data)
    sample <- rnorm(nrow(data), yhat, sd(resid(fit)))
    return(yhat) # Prediction from random subset.
}

bootstrap_confidence <- function(data) {
    control <- tree.control(nrow(data), minsize = 8)
    # Fit our model by using regression trees (8 leaves).
    fit <- tree(EX ~ MET, data = data, control = control)
    leaves <- optimal$size[least_deviance_index]
    best_tree <- prune.tree(fit, best = leaves)
    # Gives "best" tree according to k-fold cv.
    yhat <- predict(best_tree, newdata = data)
    return(yhat) # Prediction from subset.
}

bootstrap_random <- function(data, mle) {
    data$EX <- rnorm(nrow(data), predict(mle, newdata = data), sd(resid(mle)))
    return(data) # MLE is basically the best tree model from the prediction.
}

set.seed(12345) # Required for the paramatric bootstrapping using confidence bands instead.
bootstrapc <- boot(state, bootstrap_confidence, R = 1024, mle = best_tree, ran.gen = bootstrap_random, sim = "parametric")
set.seed(12345) # Required for the paramatric bootstrapping using prediction bands instead.
bootstrapp <- boot(state, bootstrap_prediction, R = 1024, mle = best_tree, ran.gen = bootstrap_random, sim = "parametric")
confidence_bands <- envelope(bootstrapc) ; prediction_bands <- envelope(bootstrapp) # Bands for the Parametric Boostrap...
```

```

cairo_ps("npcbootstrap.eps")
plot(bootstrapc)
dev.off()
cairo_ps("nppbootstrap.eps")
plot(bootstrapp)
dev.off()

cairo_ps("npbands.eps")
# Plotting MET vs EX. See the file.
print(qplot(MET, yhat, data = state,
    xlab = "Metropolitan Population Ratio",
    ylab = "Capita/Public Expenditures ($)",
    color = "predicted",
    geom = c("point")) + geom_line(data = state, aes(x = MET, y = prediction_bands$point[1,], col = "p.b")) +
                         geom_line(data = state, aes(x = MET, y = prediction_bands$point[2,], col = "p.b")) +
                         geom_line(data = state, aes(x = MET, y = confidence_bands$point[1,], col = "c.b")) +
                         geom_line(data = state, aes(x = MET, y = confidence_bands$point[2,], col = "c.b")) +
                         geom_point(data = state, aes(x = MET, y = EX, color="real")))
dev.off()
```

Listing 2: Script for Assignment 2 on Principal/Individual Component Analysis

```
library("pls")
library("ggplot2")
library("fastICA")
library("reshape2")

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

# Notice both X750, X752.
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) # But WHY?!?!?!??!?!?!?!
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()
```

Introduction to Machine Learning
Individual Laboration Report –5–

Erik Sven Vasconcelos Jansson
erija578@student.liu.se
Linköping University, Sweden

December 12, 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.

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

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

Practically, the *kernel* is calculated in Listing 1 under `gaussian_kernel` and the *estimation* is being done in the function `forecast`. However, note that the final contributions use `forecast_kernel`, which will be described now.

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 are used in `forecast_kernel` for each respective `gaussian_kernel` invocation. Additionally, these distances are *normalized* to range in-between 0.0 - 1.0. See Listing 1 for these values.

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

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

$$d_d = \begin{cases} |x - y| & |x - y| < (x + y) \bmod 365 \\ (x + y) \bmod 24 & |x - y| \geq (x + y) \bmod 365 \end{cases}$$

Therefore, the final `forecast_kernel` is being calculated as seen below, where k_l uses the *location distance*, k_d the *date distance* and k_t *time distance.*

$$k_f(\boldsymbol{x}, \boldsymbol{x'}) = k_l(\boldsymbol{x}, \boldsymbol{x'}) + k_d(\boldsymbol{x}, \boldsymbol{x'}) + k_t(\boldsymbol{x}, \boldsymbol{x'})$$

Within Table 1 are our chosen σ/h *spread/width.*

Feature	**Spread**
Location	0.192
Day	0.256
Time	0.256

Table 1: Kernel Width

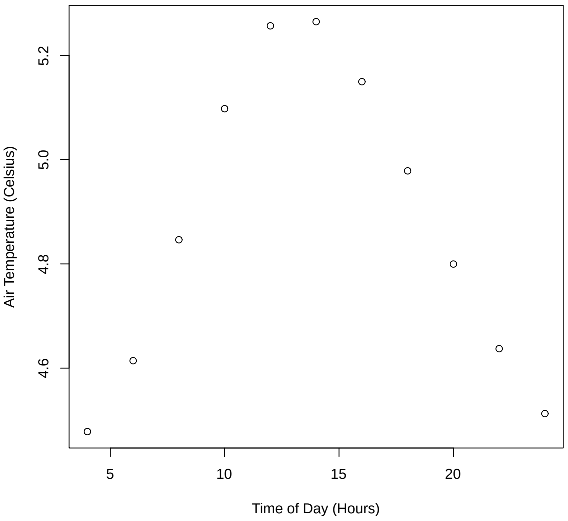
These have been chosen to decrease contribution, so for example *locations* are not easily influenced...

Finally, we estimate $\boldsymbol{y}$ for *2013-10-04* during the entire day in *58.4274 latitude* and *14.826 longitude*. See Table 2 and Figure 1 for the predicted results.

Time	**Temperature (°C)**
04:00	4.478177
06:00	4.614255
08:00	4.846348
10:00	5.097822
12:00	5.256865
14:00	5.264849
16:00	5.149687
18:00	4.978578
20:00	4.799737
22:00	4.637349
24:00	4.512750

Table 2: Forecast for 2013-11-04

Figure 1: Air Temperature Forecast Graphs



Notice how the plot above produces a *bell curve*, which is to be expected from a temperature forecast according to previous data given by *SMHI*. Also, these values don't seem to be that far off the truth, however they seem to be slightly colder than usual. One possible cause for this can be motivated by the *independence* of each *kernel*, outweighing the other.

For example, assume both *location* and *time of day* are highly correlated to our $\boldsymbol{x'}$, therefore, they will *contribute highly* with their $\boldsymbol{y^{(i)}}$. Now, for the sake of argument, assume *day of the year* is *not highly correlated* with $\boldsymbol{x'}$, therefore, the contribution $\boldsymbol{y^{(i)}}$ will not be significant, at least not compared to *location* and *time of day*. Therefore, even if we are taking an observation which is far away from the requested date, the contribution will still be high, which leads to most predictions being influenced with the "mean" temperatures in Sweden. Therefore, our hypothesis why most predictions are colder than expected is because the three kernels are being accounted independently of each other...

Additionally, some words need to be said regarding our choice for the *kernel spread/width* values in Table 1. These were chosen on the assumption that *locations* further away than *350 kilometers* are not very good contributors, as are *dates* with a distance further away than *45 days* and between ≈*5 hours*. The σ where chosen such that these values were reached, and only correlated less than 10%, thereafter proceeding with normal Gaussian falloff.

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: Nadaraya–Watson Gaussian Radial Basis Function Kernel Forecast Estimator

```r
library("ggplot2")
library("reshape2")
library("grDevices")
library("geosphere")

# Fulhax for finding day diff.
epoch <- as.Date("1970-01-01")
epocht <- as.POSIXlt(paste(epoch,
                     "00:00:00"))
stations <- read.csv("stations.csv")
temps50k <- read.csv("temps50k.csv")
weather <- merge(stations, temps50k,
              by = "station_number")

# Language R has a very shitty library for doing dates/times.
weather$time <- as.POSIXlt(paste("1970-01-01", weather$time))
weather$hour <- round(difftime(weather$time, epocht,
                               units="hours"))
weather$day <- sub("^\\d{4}", "1970", weather$date)
weather$day <- as.Date(weather$day) - epoch
weather$day <- as.numeric(weather$day)

# Find faulty convert (because of leap year).
working <- which(complete.cases(weather$day))
weather <- weather[working,] # Remove errors.
indx <- sample(1:nrow(weather),nrow(weather))

gaussian_kernel <- function(distance, sdspread)  {
    # The Gaussian Radial Basis Function.
    gamma_spread   <-  1 / (2*sdspread^2)
    return(exp(-gamma_spread*distance^2))
}

date_time_modulo_distance <- function(x, y, n) {
    numeric_distance <- abs(x - y)
    modulo_distance <- n - numeric_distance
    if (numeric_distance < modulo_distance)
        return(numeric_distance)
    else return(modulo_distance)
}

forecast_kernel <- function(longitudes, latitudes,
                            days,   time_points) {
    time_points <- as.numeric(time_points) # I dunno why. FUCKING R.
    location_distance <- distHaversine(c(longitudes[1], latitudes[1]),
                                       c(longitudes[2], latitudes[2]))
    location_distance <- (location_distance / 1000) # Kilometers.
    day_distance <- date_time_modulo_distance(days[1], days[2], 365)
    day_kernel <- gaussian_kernel(as.numeric(day_distance / 182.5), 0.256)
    time_point_distance <- date_time_modulo_distance(time_points[1], time_points[2], 24)
    time_point_kernel <- gaussian_kernel(as.numeric(time_point_distance / 12), 0.256)
    location_kernel <- gaussian_kernel(as.numeric(location_distance / 1572), 0.192)
    # cat("(", location_distance, day_distance, time_point_distance, ")",
    #     "=> (", location_kernel, day_kernel, time_point_kernel, ") \n")
    return((day_kernel + location_kernel + time_point_kernel))
}

```

```
forecast <- function(date, location, weather) {
    clock <- paste0(seq(4,24,by=2))
    date <- sub("^\\d{4}", "1970", date)
    date <- as.Date(date) # Kill me.....
    day <- abs(as.numeric(date - epoch))
    # See lecture slides for predicting yhat-kernels.
    total_kernel_sum <- vector(length = length(clock))
    weighted_temperatures <- vector(length = length(clock))
    similarities <- matrix(0, nrow(weather), length(clock))
    stp <- 512 # Print steps for showing the progress...

    # Loop through each observation and desired time.
    for (i in 1:nrow(weather)) { # Shitty apply fails...
        if (i%%stp==0) cat("Progress",(i/nrow(weather))*100,"%\n")
        for (j in 1:length(clock)) { # Need to predict for each...
            # Calculate the similarity for this observation and our desired response
            similarities[i, j] <- forecast_kernel(c(location[1], weather[i,]$longitude),
                                                  c(location[2], weather[i,]$latitude),
                                 c(day, weather[i,]$day), c(clock[j], weather[i,]$hour))
            total_kernel_sum[j] <- total_kernel_sum[j] + similarities[i, j]
            weighted_temperature <- similarities[i, j] * weather[i,]$air_temperature
            weighted_temperatures[j] <- weighted_temperatures[j] + weighted_temperature
        }
        # Apply the Nadaraya-Watson kernel regression.
    } ; return(weighted_temperatures / total_kernel_sum)
}

args <- commandArgs(TRUE)
day <- if (!is.na(args[1])) as.Date(args[1]) else as.Date("2013-11-04")
latitude <- if (!is.na(args[2])) as.numeric(args[2]) else 58.4274
longitude <- if (!is.na(args[3])) as.numeric(args[3]) else 14.826
temperatures <- forecast(day, c(longitude, latitude),
                         weather[indx,]) # Kill me...
cat("Forecast (in oC) for the", as.character(day),
    "at 04:00:00 - 24:00:00 in", longitude, latitude,
    "(longitude, latitude)\n")
cat(temperatures, "\n")

setEPS()
cairo_ps("forecast")
plot(clock, temperatures, xlab="Time of Day",
                          ylab="Temperature Celsius")
dev.off()
```

Introduction to Machine Learning
Individual Laboration Report –6–

Erik Sven Vasconcelos Jansson
erija578@student.liu.se
Linköping University, Sweden

December 20, 2016

Finally, the last machine learning topic covered are *artificial neural networks*. These estimators are very flexible, such that even a *single layer feed-forward neural network* complies with the *universal approximation theorem*, presented by *Csáji* [Csá01]:

Theorem 1 (Universal Approximation Theorem). *Any artificial feed-forward neural network with a single hidden layer, containing a finite amount of neurons, can approximate any continuous functions on the compact subset $\mathcal{R}^n$ (with restrictions on σ).*

Proof. *Csáji's* [Csá01] derivation of Theorem 1. □

Basically, the theorem states that even simple neural networks can represent interesting functions, given some suitable subset of *activation functions*. For this assignment, we want to approximate $\sin x$, where we are given *25 observation* for *training set*. Also, we are given a *validation set* of length *25* for checking if our neural network is under/overfitting. We are using the *R package* `neuralnet` for our fit with *10 hidden units* in a *single hidden layer*, also initialized with *random weights* in $[-1, 1]$ interval. See Listing 1 for the entire assignment source code.

For the curious, Equations 1, 2, and 3 are given:

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

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

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

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.

By using a *threshold* for the *gradient descent* we can stop the *neural network* from either *overfitting* or *underfitting.* This simply done by increasing the threshold iteratively and taking the validation set's:

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

After finding the "optimal threshold" of *0.004* by picking the 4^{th} iteration (where $i = 4$ that is) which gives the least amount of error for a validation set, we plot the best neural network in Figure 1, and also the predictions in Figure 2 for a *sine function.* Notice that the fit is pretty good, and the estimator gives a pretty "spot on" prediction for the function. It seems *neural networks* are incredibly powerful, but take time to train and are harder to reason about (for example, how do we choose the number of hidden layers and units? How long will it take?)

References

[Csá01] Balázs Csanád Csáji. Approximation with Artificial Neural Networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[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.

Figure 1: Neural Network

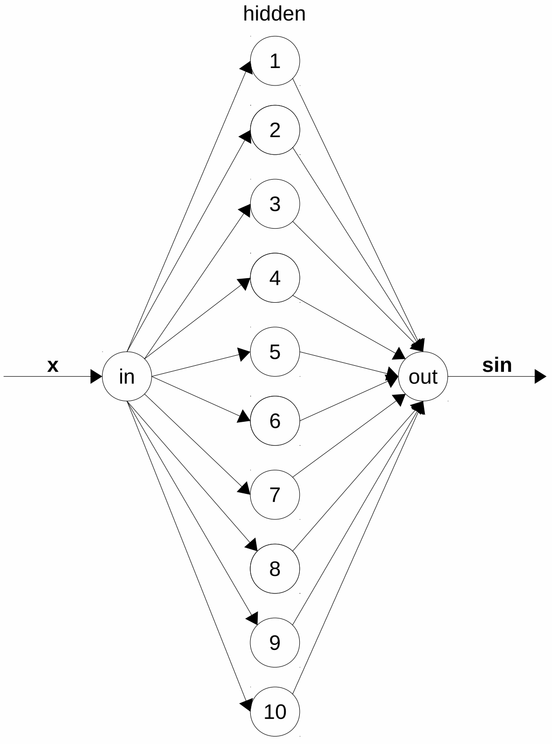
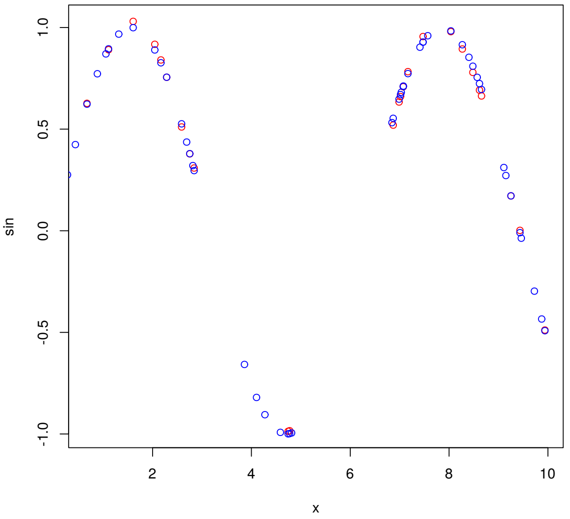


Figure 2: Neural Network's Produced Predictions (in the graph are raw values and predicted values).



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"
```