Machine Learning Summary

Maximilian Pfundstein

2019-01-03

Contents

1 Terms

1.1 Bagging

Bagging (stands for Bootstrap Aggregating) is a way to decrease the variance of your prediction by generating additional data for training from your original dataset using combinations with repetitions to produce multisets of the same cardinality/size as your original data. By increasing the size of your training set you can't improve the model predictive force, but just decrease the variance, narrowly tuning the prediction to expected outcome.

1.2 Bias-Variance Tradeoff

There's a bias-variance tradeoff between setting the degrees of freedom. The more variables you have, more complex your model is. But it tends to overfit. So there must be a balance in selecting the variables.

1.3 Boosting

Boosting is a two-step approach, where one first uses subsets of the original data to produce a series of averagely performing models and then "boosts" their performance by combining them together using a particular cost function (=majority vote). Unlike bagging, in the classical boosting the subset creation is not random and depends upon the performance of the previous models: every new subsets contains the elements that were (likely to be) misclassified by previous models.

1.4 Curse of Dimensionality

The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience. The expression was coined by Richard E. Bellman when considering problems in dynamic optimization.

Cursed phenomena occur in domains such as numerical analysis, sampling, combinatorics, machine learning, data mining and databases. The common theme of these problems is that when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality.

Also, organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data, however, all objects appear to be sparse and dissimilar in many ways, which prevents common data organization strategies from being efficient

1.5 Degrees of Freedom

Definition: The degrees of freedom is the values in the final calculation of a statistic that are free to vary.

Example: Example: If you know that mean of 4 numbers is 20, and two of those numbers are 40, 50. This means that if you can find out one of the remaining numbers (say x), you can tell with certainty that the last number (say y) will be $y = 20 - ((40 + 50 + x)/3)$ i.e. you have one degree of freedom

- In Machine learning theory, the independent variables on which the target depends on are called the degrees of freedom.
- Need of degree of freedom: It is important to know which variables does the model depend on. This can then be used for feature selection.
- Impact of degrees of freedom on Machine learning model: There's a bias-variance tradeoff between setting the degrees of freedom. The more variables you have, more complex your model is. But it tends to overfit. So there must be a balance in selecting the variables.

1.6 Generative and Discriminative Models

Let's say you have input data x and you want to classify the data into labels y. A generative model learns the joint probability distribution $p(x, y)$ and a discriminative model learns the conditional probability distribution $p(y|x)$ - which you should read as "the probability of y given x".

$p(x, y)$	y=0	y=1
x=1	1/2	0
x=2	1/4	1/4

$p(x|y)$	y=0	y=1
x=1	1	0
x=2	1/2	1/2

If you take a few minutes to stare at those two matrices, you will understand the difference between the two probability distributions.

The distribution $p(y|x)$ is the natural distribution for classifying a given example x into a class y, which is why algorithms that model this directly are called discriminative algorithms. Generative algorithms model $p(x, y)$, which can be transformed into $p(y|x)$ by applying Bayes rule and then used for classification. However, the distribution $p(x, y)$ can also be used for other purposes. For example, you could use $p(x, y)$ to generate likely (x, y) pairs.

From the description above, you might be thinking that generative models are more generally useful and therefore better, but it's not as simple as that. This paper is a very popular reference on the subject of discriminative vs. generative classifiers, but it's pretty heavy going. The overall gist is that discriminative models generally outperform generative models in classification tasks

1.7 Kernel Trick

The kernel trick is based on some concepts: you have a dataset, e.g. two classes of 2D data, represented on a cartesian plane. It is not linearly separable, so for example a SVM could not find a line that separates the two classes. Now, what you can do it project this data into an higher dimension space, for example 3D, where it could be divided linearly by a plane.

Now, a basic concept in ML is the dot product. You often do dot products of the features of a data sample with some weights w, the parameters of your model. Instead of doing explicitly this projection of the data in 3D and then evaluating the dot product, you can find a kernel function that simplifies this job by simply doing the dot product in the projected space for you, without the need to actually compute projections and then the dot product. This allows you to find a complex non linear boundary that is able to separate the classes in the dataset. This is a very intuitive explainatio

1.8 No Free Lunch Theorem

The "No Free Lunch" theorem states that there is no one model that works best for every problem. The assumptions of a great model for one problem may not hold for another problem, so it is common in machine learning to try multiple models and find one that works best for a particular problem. This is especially true in supervised learning; validation or cross-validation is commonly used to assess the predictive accuracies of multiple models of varying complexity to find the best model. A model that works well could also be trained by multiple algorithms - for example, linear regression could be trained by the normal equations or by gradient descent.

1.9 Ordinary Least Squares Regression (OLS)

In statistics, ordinary least squares (OLS) is a type of linear least squares method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by the principle of least squares: minimizing the sum of the squares of the differences between the observed dependent variable (values of the variable being predicted) in the given dataset and those predicted by the linear function.

1.10 Parametric and Non-Parametric Models

Parametric models A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples). No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs.

Some examples of parametric machine learning algorithms are:

- Linear Regression
- Linear Support Vector Machines
- Logistic Regression
- Naive Bayes
- Perceptron

Nonparametric models Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features.

Some examples of nonparametric models are:

- Decision Trees
- K-Nearest Neighbor
- Support Vector Machines with Gaussian Kernels

- Artificial Neural Networks

In conclusion with parametric models to predict new data, you only need to know the parameters of the model. In nonparametric methods are more flexible and for forecasting new data you need to know the parameters of the model and the state of the data that has been observed.

1.11 Stacking

Stacking is a similar to boosting: you also apply several models to your original data. The difference here is, however, that you don't have just an empirical formula for your weight function, rather you introduce a meta-level and use another model/approach to estimate the input together with outputs of every model to estimate the weights or, in other words, to determine what models perform well and what badly given these input data.

1.12 Types of Learning

1.12.1 Active Learning

Active learning is a special case of machine learning in which a learning algorithm is able to interactively query the user (or some other information source) to obtain the desired outputs at new data points. In statistics literature it is sometimes also called optimal experimental design.

In active learning, the algorithm gets a lot of data, but not the labels. The algorithm can then explicitly request labels to individual examples. This can be helpful when we have a large amount of unlabeled data, and we want the examples that we label manually to be as helpful for learning as possible.

Let's say you want to find faces in YouTube videos. The labels for the algorithm would be bounding boxes around the faces in each video frame. However, there are so many videos with so many frames, that we can't label all of them manually.

We could just label 100 videos frame by frame, but that would be a lot of wasted effort. Since following frames are often similar, the algorithm might not benefit very much from each consecutive frames being labeled.

In active learning, the algorithm tries to both learn the task and tell us what labels would be most useful at the current state. We can then label just those frames, so that the manual effort is directed best.

1.12.2 Reinforcement Learning

This method aims at using observations gathered from the interaction with the environment to take actions that would maximize the reward or minimize the risk. Reinforcement learning algorithm (called the agent) continuously learns from the environment in an iterative fashion. In the process, the agent learns from its experiences of the environment until it explores the full range of possible states.

Reinforcement Learning is a type of Machine Learning, and thereby also a branch of Artificial Intelligence. It allows machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize its performance. Simple reward feedback is required for the agent to learn its behavior; this is known as the reinforcement signal.

1.12.3 Semi-supervised

In the previous two types, either there are no labels for all the observation in the dataset or labels are present for all the observations. Semi-supervised learning falls in between these two. In many practical situations, the cost to label is quite high, since it requires skilled human experts to do that. So, in the absence of labels in the majority of the observations but present in few, semi-supervised algorithms are the best candidates for the model building. These methods exploit the idea that even though the group memberships of the unlabeled data are unknown, this data carries important information about the group parameters.

1.12.4 Supervised Learning

- I like to think of supervised learning with the concept of function approximation, where basically we train an algorithm and in the end of the process we pick the function that best describes the input data, the one that for a given X makes the best estimation of y (X -> y). Most of the time we are not able to figure out the true function that always make the correct predictions and other reason is that the algorithm rely upon an assumption made by humans about how the computer should learn and this assumptions introduce a bias, Bias is topic I'll explain in another post.
- Here the human experts acts as the teacher where we feed the computer with training data containing the input/predictors and we show it the correct answers (output) and from the data the computer should be able to learn the patterns.
- Supervised learning algorithms try to model relationships and dependencies between the target prediction output and the input features such that we can predict the output values for new data based on those relationships which it learned from the previous data sets.

1.12.5 Unsupervised Learning

- The computer is trained with unlabeled data.
- Here there's no teacher at all, actually the computer might be able to teach you new things after it learns patterns in data, these algorithms a particularly useful in cases where the human expert doesn't know what to look for in the data.
- It's a family of machine learning algorithms which are mainly used in pattern detection and descriptive modeling. However, there are no output categories or labels here based on which the algorithm can try to model relationships. These algorithms try to use techniques on the input data to mine for rules, detect patterns, and summarize and group the data points which help in deriving meaningful insights and describe the data better to the users.

1.13 Under- and Overfitting

Overfitting in Machine Learning

Overfitting refers to a model that models the training data too well.

Overfitting happens when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. This means that the noise or random fluctuations in the training data is picked up and learned as concepts by the model. The problem is that these concepts do not apply to new data and negatively impact the models ability to generalize.

Overfitting is more likely with nonparametric and nonlinear models that have more flexibility when learning a target function. As such, many nonparametric machine learning algorithms also include parameters or techniques to limit and constrain how much detail the model learns.

For example, decision trees are a nonparametric machine learning algorithm that is very flexible and is subject to overfitting training data. This problem can be addressed by pruning a tree after it has learned in order to remove some of the detail it has picked up.

Underfitting in Machine Learning

Underfitting refers to a model that can neither model the training data nor generalize to new data.

An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training data.

Underfitting is often not discussed as it is easy to detect given a good performance metric. The remedy is to move on and try alternate machine learning algorithms. Nevertheless, it does provide a good contrast to the problem of overfitting.

2 Useful Code Snippets

2.1 Confusion Matrix and Misclassification Rate

```
confusion_matrix_train = as.matrix(table(spambase_predict_train, train$Spam))

error_rate =
  1 - sum(diag(confusion_matrix_train)/sum(confusion_matrix_train))
print(error_rate)
```

2.2 Feature Plot

```
ggplot(statedata, aes(x = MET, y = EX)) +  geom_point() + geom_smooth()
```

```
ggplot(adb_errors) +
  geom_line(aes(x = n, y = error_rate_training,
                colour = "AdaBoost Training"), linetype = "dashed") +
  geom_point(aes(x = n, y = error_rate_training), colour = "orange") +

  geom_line(aes(x = n, y = error_rate_validation,
                colour = "AdaBoost Validation")) +
  geom_point(aes(x = n, y = error_rate_validation), colour = "red") +

  geom_line(aes(x = n, y = error_rate_training,
                colour = "Random Forest Training"),
            data = rf_errors, linetype = "dashed") +
  geom_point(aes(x = n, y = error_rate_training),
             colour = "blue", data = rf_errors) +

  geom_line(aes(x = n, y = error_rate_validation,
                colour = "Random Forest Validation"), data = rf_errors) +
  geom_point(aes(x = n, y = error_rate_validation),
             colour = "steelblue2", data = rf_errors) +
  labs(title = "Random Forest and AdaBoost", y = "Error Rate",
       x = "Number of Forests", color = "Legend") +
  scale_color_manual(values = c("orange", "red", "blue", "steelblue2"))
```

2.3 Histogram

```
hist(pruned_tree_plot_dataframe$residual,
     col="orange", main = "Histogram of the Residuals", xlab = "Residuals", breaks = 20)
```

2.4 Importing Data

2.4.1 Splitting Data

```
data(spam)
n = dim(spam)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.4))
train_spam = spam[id,]
id1 = setdiff(1:n, id)
set.seed(12345)
id2 = sample(id1, floor(n*0.3))
val_spam = spam[id2,]
id3 = setdiff(id1,id2)
test_spam = spam[id3,]
```

2.4.2 .csv

```
emails = read.csv2("data.csv", fileEncoding = "ISO-8859-1", sep = ";")
emails$Conference = as.factor(emails$Conference)
```

```
stations = read.csv("stations.csv", encoding = "UTF-8")
temps = read.csv("temps50k.csv", encoding = "UTF-8")
st = merge(stations, temps, by="station_number")
```

2.4.3 .xls and .xlsx

```
creditscoring = read_excel("creditscoring.xls")
creditscoring$good_bad = as.factor(creditscoring$good_bad)
```

2.5 RMarkdown Template

2.5.1 Header

```
author: "Maximilian Pfundstein"
date: "2019-01-03"
output:
  pdf_document:
    toc: true
    toc_depth: 3
    number_sections: true
  html_document:
    df_print: paged
    toc: true
    toc_float: true
    number_sections: true
```

2.5.2 Setup

```
{r setup, include = FALSE}
knitr::opts_chunk$set(echo = FALSE, cache = TRUE, include = TRUE, eval = FALSE)
```

2.5.3 Source Code as Appendix

```
{r, ref.label=knitr::all_labels(), echo = TRUE, eval = FALSE, results = 'show'}
```

3 Models

3.1 Bayesian Classification

- Prediction $\widehat{Y}(x) = C_l$ and $l = arg_{i\epsilon\{1,\ldots,m\}} maxp(C_i|x)$
- Bayes Theorem $p(C_i|x) = \frac{p(x|C_i)p(C_i)}{p(x)}$
- WE get $p(C_i|x) \propto \frac{K_i}{K}$

3.2 Boosting

3.2.1 AdaBoost

`blackboost()` from package `mboost`.

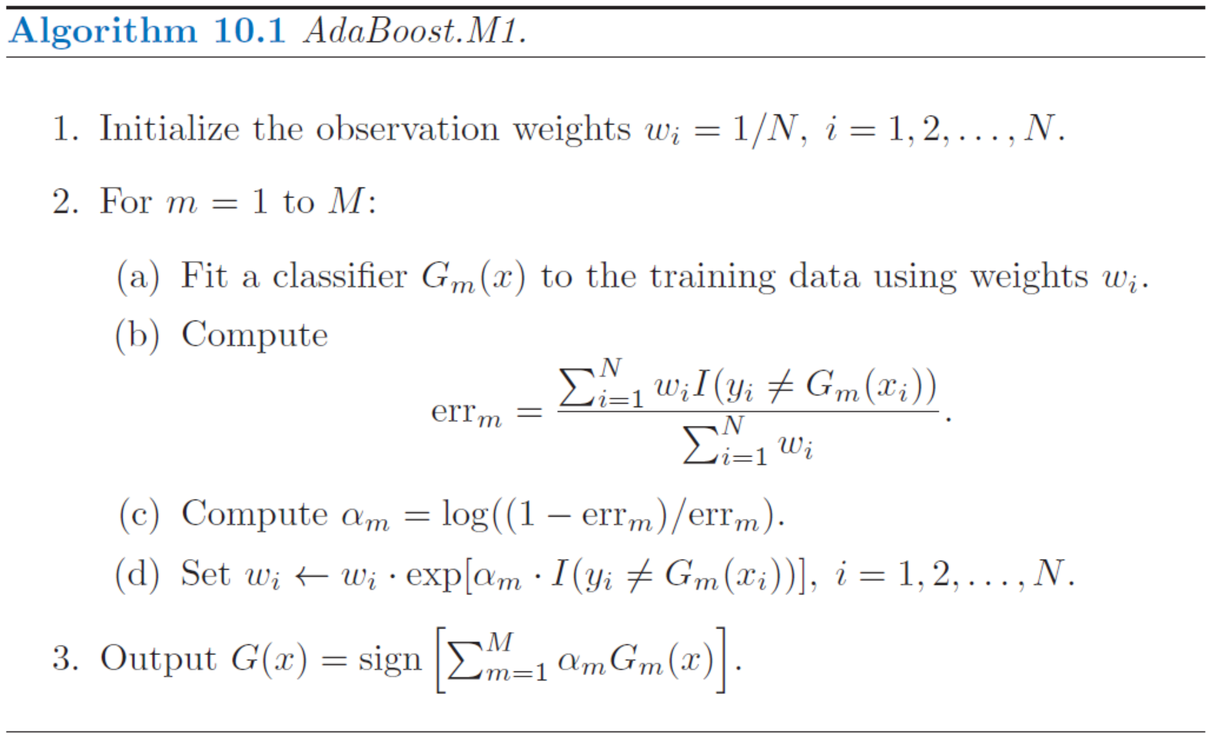
```
c_adaBoost = blackboost(formula = c_formula,
                        data = train_spambase,
                        family = AdaExp(),
                        control=boost_control(mstop=i))
```

3.2.2 Forward Stagewise Additive Modeling

3.2.3 Gradient Boosting

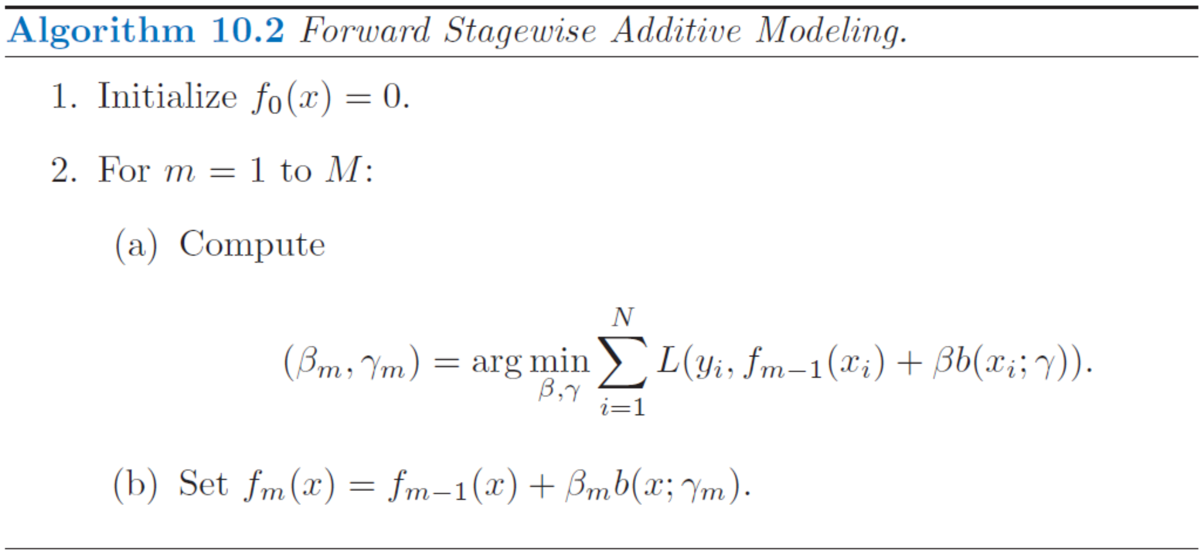
3.3 Elastic Net

```
# Elastic Net

# Predictor Variables. The -1 removes the intercept component
x_test = as.matrix(train_emails[, -ncol(train_emails)])
x_val = as.matrix(val_emails[, -ncol(val_emails)])

# Outcome variable
y_test = train_emails$Conference
y_val = val_emails$Conference

model_elastic_net = cv.glmnet(x = x_test, y = y_test, alpha = 0.5,
                              family = "binomial")

# Support vector machine with "vanilladot" kernel.
# Vanilladot means "Linear Kernel Function"
```


Algorithm 10.1 *AdaBoost.M1.*

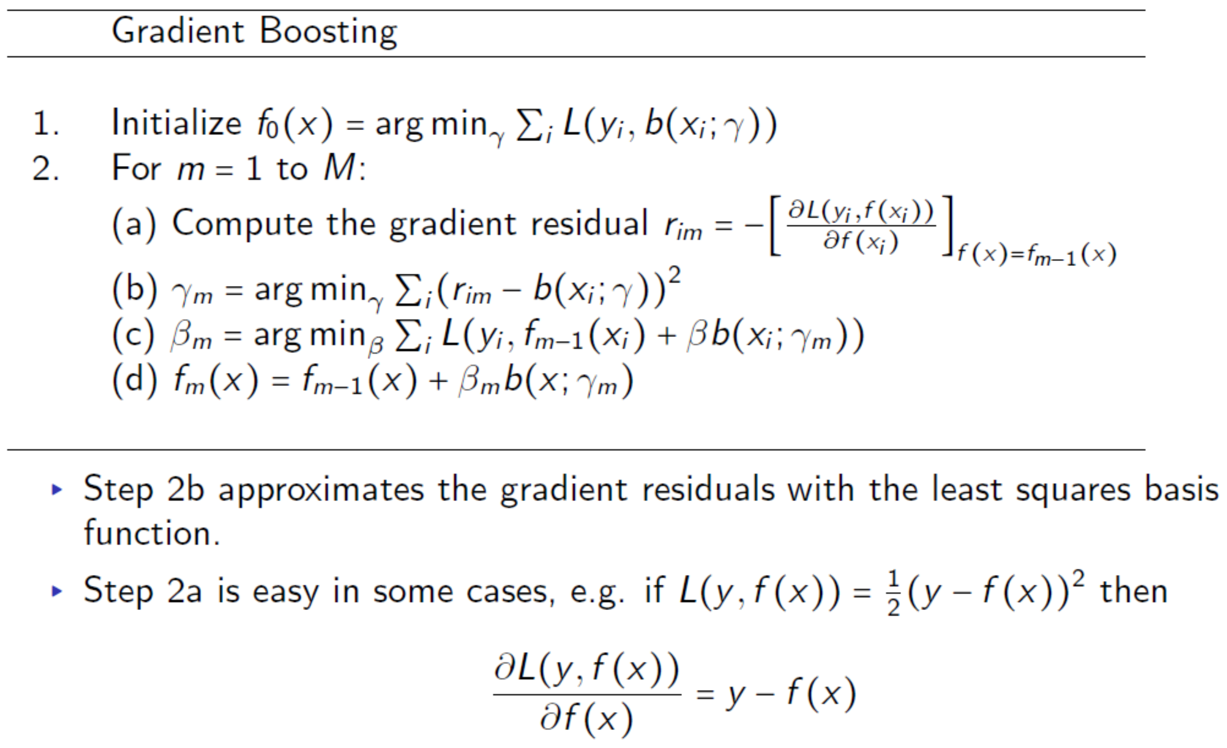
```
1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to $M$:
    (a) Fit a classifier $G_m(x)$ to the training data using weights $w_i$.
    (b) Compute
        $$\text{err}_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}.$$
    (c) Compute $\alpha_m = \log((1 - \text{err}_m)/\text{err}_m)$.
    (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \text{sign}\left[\sum_{m=1}^M \alpha_m G_m(x)\right]$.
```

Figure 1: Ada Boost Algorithm


Algorithm 10.2 *Forward Stagewise Additive Modeling.*

```
1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to $M$:
    (a) Compute
        $$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)).$$
    (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
```

Figure 2: Forward Stagewise Additive Modeling



Gradient Boosting

1. Initialize $f_0(x) = \arg\min_\gamma \sum_i L(y_i, b(x_i; \gamma))$
2. For $m = 1$ to M:

   (a) Compute the gradient residual $r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x) = f_{m-1}(x)}$

   (b) $\gamma_m = \arg\min_\gamma \sum_i (r_{im} - b(x_i; \gamma))^2$

   (c) $\beta_m = \arg\min_\beta \sum_i L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma_m))$

   (d) $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$

- Step 2b approximates the gradient residuals with the least squares basis function.
- Step 2a is easy in some cases, e.g. if $L(y, f(x)) = \frac{1}{2}(y - f(x))^2$ then

$$\frac{\partial L(y, f(x))}{\partial f(x)} = y - f(x)$$

Figure 3: Gradient Boosting Algorithm

```r
# set scale = FALSE to prevent "variable(s) `' constant. Cannot scale data.""
model_svm = ksvm(x = x_test, y = y_test, kernel = "vanilladot", scale = FALSE)

# Predictions
pred_val_elastic = predict(model_elastic_net, newx = x_val, type = "class")
pred_val_svm = predict(model_svm, x_val, type = "response")

matrix_elastic = table(y_val, t(pred_val_elastic))
matrix_svm = table(y_val, pred_val_svm)

kable(matrix_elastic)
kable(matrix_svm)

summary(model_elastic_net)
plot(model_elastic_net)
```

3.4 Generalized Additive Model (GAM)

Somehow uses cross validation.

```r
gam_model = gam(formula = Mortality ~ Year + s(Week,
                k=length(unique(influanza$Week))), family = gaussian(),
                data = influanza, method="GCV.Cp")

print(gam_model)
```

```r
summary(gam_model)
```

Probabilistic Model:

$$Model \sim \mathcal{N}(\beta_{year} * X_{Year} + S_{week} * X_{week} + \alpha, \sigma^2)$$

```
Model \sim \mathcal{N}(\beta_{year} * X_{Year} +S_{week} * X_{week} + \alpha,  \sigma^2)
```

3.4.1 Useful GAM Plots

```r
time = influanza$Time
observed = influanza$Mortality
predicted = gam_model$fitted.values

mortality_values = data.frame(time, observed, predicted)

ggplot(mortality_values) +
  geom_line(aes(x = time, y = observed,
                colour = "Observed Mortality")) +
  geom_line(aes(x = time, y = predicted,
                colour = "Predicted Mortality")) +
  labs(title = "Observed vs Predicted Mortality", y = "Mortality",
       x = "Time", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))

ggplot() +
  geom_line(aes(x=influanza$Week,y=influanza$Mortality,
                         colour=as.factor(influanza$Year))) +
  labs(x='Week', y='Mortality', colour='Year',
       title='Mortality by Week and by Year')
```

3.4.2 Penalty Factor, Deviance, Degress of Freedom

```r
penalties = seq(from = 0 , to = 20, by = 0.1)
index = 1

fitted_gam_with_penalties = data.frame()

for (penalty in penalties) {
  gam_model = gam(formula = Mortality ~ Year +
                    s(Week, sp = penalty, k = length(unique(influanza$Week))),
                  family = gaussian(), data = influanza, method="GCV.Cp")

  fitted_gam_with_penalties = rbind(fitted_gam_with_penalties,
        data.frame(list(penalty = penalty,
                        deviance = gam_model$deviance,
                        df = sum(influence(gam_model)))))
}

## Deviance VS Penalty
```

```r
ggplot(fitted_gam_with_penalties) +
  geom_line(aes(x = fitted_gam_with_penalties$penalty,
                y = fitted_gam_with_penalties$deviance,
                colour = "Deviance/Penalty")) +
  labs(title = "Deviance VS Penalty", y = "Deviance",
       x = "Penalty", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))

## Penalty Factors vs Degrees of Freedom
ggplot(fitted_gam_with_penalties) +
  geom_line(aes(x = fitted_gam_with_penalties$deviance,
                y = fitted_gam_with_penalties$df,
                colour = "Degrees of Freedom/Deviance")) +
  labs(title = "Deviance VS Degrees of Freedom", y = "Degrees of Freedom",
       x = "Deviance/DF", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))

## Create Models for High and Low penalties
gam_model_low_pen = gam(formula = Mortality ~ Year + s(Week,
                  k=length(unique(influanza$Week)), sp = 0.1),
                  family = gaussian(), data = influanza, method="GCV.Cp")
gam_model_high_pen =
  gam(formula = Mortality ~ Year + s(Week, k=length(unique(influanza$Week)),
                                     sp = 20),
                  family = gaussian(), data = influanza, method="GCV.Cp")

high_low_df = data.frame()
high_low_df = rbind(high_low_df,
                    list(mortality = influanza$Mortality,
                          mortality_low = fitted(gam_model_low_pen),
                          mortality_high = fitted(gam_model_high_pen),
                          date = influanza$Time))

ggplot(high_low_df) +
  geom_line(aes(x = date, y = mortality, colour = "Mortality")) +
  geom_line(aes(x = date, y = mortality_low, colour = "Mortality (Low: 0.1)")) +
  geom_line(aes(x = date, y = mortality_high, colour = "Mortality (High: 20)")) +
  labs(title = "Mortalities vs Time", y = "Mortality",
       x = "Time", color = "Legend") +
  scale_color_manual(values = c("blue", "orange", "violet"))

## Observed vs Predicted Mortality
ggplot(mortality_values) +
  geom_line(aes(x = time, y = observed,
                colour = "Observed Mortality")) +
  geom_line(aes(x = time, y = predicted,
                colour = "Predicted Mortality")) +
  labs(title = "Observed vs Predicted Mortality", y = "Mortality",
       x = "Time", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))
```

3.4.3 Residuals

```
gam_model$residuals
```

Family parameter mgcv

3.5 Generalized Linear Model (GLM)

```
spambase_model = glm(Spam ~ ., data = train, family = "binomial")
spambase_predict_train =
  predict(object = spambase_model, newdata = train, type = "response")
spambase_predict_train =
  apply(as.matrix(spambase_predict_train), c(1),
        FUN = function(x) return(x > 0.5))
```

- Response Poisson distributed
- Canonical Link (log) is used for regression
- probabilistic expression for the fitted model

3.6 K-Nearest Neighbour (KNN)

3.6.1 Implementation

```
knearest = function(data, K = 5, newdata) {

  # Internal Function to calculate D
  d = function(X, Y) {

    # Make sure the input in the matrix format
    X = as.matrix(X)
    Y = as.matrix(Y)

    # Calculate D as described in the exercise
    X_hat = X/sqrt(rowSums(X^2))
    Y_hat = Y/sqrt(rowSums(Y^2))
    C = X_hat %*% t(Y_hat)
    D = 1 - C

    return(D)
  }

  # Get D (trim classification column)
  D = d(data[,-49], newdata[-49])

  classificationRate = c()
  classification = c()
  classifiedCorrectly = c()

  # For each data entry
  for (i in 1:nrow(D)) {
    # Sort the distances and also get their index
```

```r
    sortedRow = sort(D[,i], index.return = TRUE)

    # Take the K best guys and save their indexed
    indexesKnn = sortedRow$ix[1:K]

    # Lookup if they're classified as 0 or 1
    classificationRates = data$Spam[indexesKnn]

    # Add the classification
    classificationRate = c(classificationRate,
                           (sum(data$Spam[indexesKnn] / K)))
    temp_classification = sum(data$Spam[indexesKnn]) > (K/2)
    classification = c(classification, temp_classification)
    classifiedCorrectly =
      c(classifiedCorrectly, temp_classification != as.logical(newdata$Spam[i]))
  }

  returnDataFrame = cbind(newdata, classificationRate)
  returnDataFrame = cbind(returnDataFrame, classification)
  returnDataFrame = cbind(returnDataFrame, classifiedCorrectly)

  return(returnDataFrame)
}

# Set seed and import data from excel
set.seed(12345)
spambase = read_excel("spambase.xlsx")

# Shuffle the data
n = dim(spambase)[1]
id = sample(1:n, floor(n*0.5))
c_data = spambase[id,]
c_newdata = spambase[-id,]

# Get the missqualification rates for training and  test
training_classification = knearest(c_data, K = 30, c_data)
test_classification = knearest(c_data, K = 30, c_newdata)

# Get the classification rate
training_rate = sum(training_classification$classifiedCorrectly/
  nrow(training_classification))
test_rate = sum(test_classification$classifiedCorrectly/
  nrow(test_classification))
```

3.6.2 Library

```r
kknn_model_train = kknn(formula = Spam ~ ., train = train, test = train, k = 30)
y_hat_train =
  apply(as.matrix(kknn_model_train$fitted.values), c(1),
        FUN = function(x) return(x > 0.5))
```

3.7 Lasso

Ridge Regression has the problem, that even with high $\lambda's$ the coefficients will never be zero, this would only be the case for $lim_{\lambda \to \infty}$. This time the *Shrinkage Penalty* is

$$\lambda \sum_j |\beta_j|$$

Due to the fact that high $\lambda's$ can set the coefficients to zero, LASSO perform a *Variable Selection.* This is what can bee seen in the plot, with increasing $\ln(\lambda)$ more and more variables are set to 0.

```
covariates_lasso = scale(tecator_data[,2:(ncol(tecator_data)-3)])
response_lasso = tecator_data$Fat

glm_model_lasso = glmnet(as.matrix(covariates_lasso),
                         response_lassoe, alpha = 1, family="gaussian")
plot(glm_model_lasso, xvar="lambda")
```

3.7.1 Cross-Validation

```
lasso_vc = cv.glmnet(y = response_lasso, x = covariates_lasso,
                     alpha = 1, lambda = seq(from = 0, to = 10, by = 0.01))
plot(lasso_vc)
print(paste(sep = "", "Best lambda score: ", lasso_vc$lambda.min))
```

3.8 Least Absolute Deviation Regression

Package `L1pack`.

- Model $Y \sim Laplace(w^T X, b)$
- Equivalent to minimizing sum of absolute deviations
- Properties
  - Robust to outliers
  - Sensitive tochangesindata
  - Multiplesolutions possible

3.9 Linear Regression

```
model = lm(formula = c_formula, data = tecator_data)
```

3.10 Log Likelihood

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

```
# Function that computes log-likelihood for given theta and vector x
compute_llh = function(theta, x) {
  p = dexp(x, theta) # vector with densities p(xi | theta) for each i
```

```r
  likelihood = prod(p) # product of all densities in p
  loglikelihood = log(likelihood) # take natural logarithm
}

# Plot that shows the dependency of the log-likelihood on theta
llhs = sapply(seq(0, 10, 0.005), compute_llh, x = df$Length)
plotdata1 = data.frame(loglikelihood = llhs, theta = seq(0, 10, 0.005),
data = "All", type = "Exponential")

ggplot(plotdata1) +
geom_point(aes(x = theta, y = loglikelihood), size = 0.5, color = "#00BFC4") +
labs(title = "Loglikelihood by Theta") + theme_minimal() +
theme(plot.title = element_text(hjust = 0.5)) +
scale_x_continuous(limits = c(0, 10), breaks = seq(0, 10, 0.5))
```

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

```r
llhs = sapply(seq(0, 10, 0.005), compute_llh, x = df$Length[1:6])
plotdata2 = data.frame(loglikelihood = llhs, theta = seq(0, 10, 0.005),
  data = "6 observations", type = "Exponential")
plotdata_aggr = rbind(plotdata2, plotdata1)

p = ggplot(plotdata_aggr) +
  geom_point(aes(x = theta, y = loglikelihood, color = data), size = 0.5) +
  labs(title = "Loglikelihood by Theta", color = "Data") +
  theme_minimal() +
  theme(plot.title = element_text(hjust = 0.5)) +
  scale_x_continuous(limits = c(0, 10), breaks = seq(0, 10, 0.5))

p

# tail(ggplot_build(p)$data[[1]]) # To see which color values are used
# head(ggplot_build(p)$data[[1]]) # To see which color values are used
# For comparison: Just the curve with 6 observations
# p = ggplot(plotdata_aggr %>% filter(data == "6 observations")) +
# geom_point(aes(x = theta, y = loglikelihood), color = "#F8766D", size = 0.5) +
# labs(title = "Loglikelihood by Theta", color = "Data") +
# theme_minimal() +
# theme(plot.title = element_text(hjust = 0.5)) +
# scale_x_continuous(limits = c(0, 10), breaks = seq(0, 10, 0.5))
#
# p
```

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

```r
df = readxl::read_xlsx("machines.xlsx")

compute_l = function(theta, x) {

# Likelihood: p(x|theta) ----------------------------------------------------
p = dexp(x, theta) # vector with densities p(xi | theta) for each i
likelihood = prod(p) # product of all densities in p

# Prior: p(theta) where rate = lambda ----------------------------------------
lambda = 10
prior = dexp(theta, lambda)

# Posterior ------------------------------------------------------------------
posterior = likelihood * prior

# Natural logarithm of posterior ---------------------------------------------
logposterior = log(posterior)
}

# Plot that shows the dependency of the log-likelihood on theta
ls = sapply(seq(0, 10, 0.005), compute_l, x = df$Length)
plotdata3 = data.frame(loglikelihood = ls, theta = seq(0, 10, 0.005),
  data = "All", type = "Bayesian")

plotdata_aggr = rbind(plotdata3, plotdata1)
p = ggplot(plotdata_aggr) +
  geom_point(aes(x = theta, y = loglikelihood, color = type), size = 0.5) +
  labs(title = "Loglikelihood (Exponential) vs. l (Bayesian) by Theta",
  color = "Type") + theme_minimal() +
  theme(plot.title = element_text(hjust = 0.5)) +
  scale_x_continuous(limits = c(0, 10), breaks = seq(0, 10, 0.5))

p
```

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

```r
p1 = ggplot(df) +
  geom_histogram(aes(x = Length), fill = "#00BFC4", alpha = 0.7) +
  theme_minimal() + labs(title = "Original") +
  theme(plot.title = element_text(hjust = 0.5))

# Histogram for simulated data with theta = 1.2
simulated1 = data.frame(Length = rexp(50, rate = 1.2))

p2 = ggplot(simulated1) +
  geom_histogram(aes(x = Length), fill = "#F8766D", alpha = 0.7) +
  theme_minimal() + labs(title = "Simulated (theta = 1.2)") +
  theme(plot.title = element_text(hjust = 0.5))

# Bind Plots together
plot_list = list(p1, p2)
plot_list_arranged = arrangeGrob(grobs = plot_list, ncol = 2)
grid.arrange(plot_list_arranged, top = "Histogram of Original vs. Simulated Lengths",
padding = unit(0.5, "line"))
```

3.11 Logistic Regression

- Equation of decision boundary
- Plot classified data and decision boundary
- GLM
- Custom Classification

3.12 Naive Bayes

3.12.1 Library

Package e1071.

```r
naiveBayesModel = naiveBayes(good_bad ~ ., data = train)
```

3.12.2 Implementation

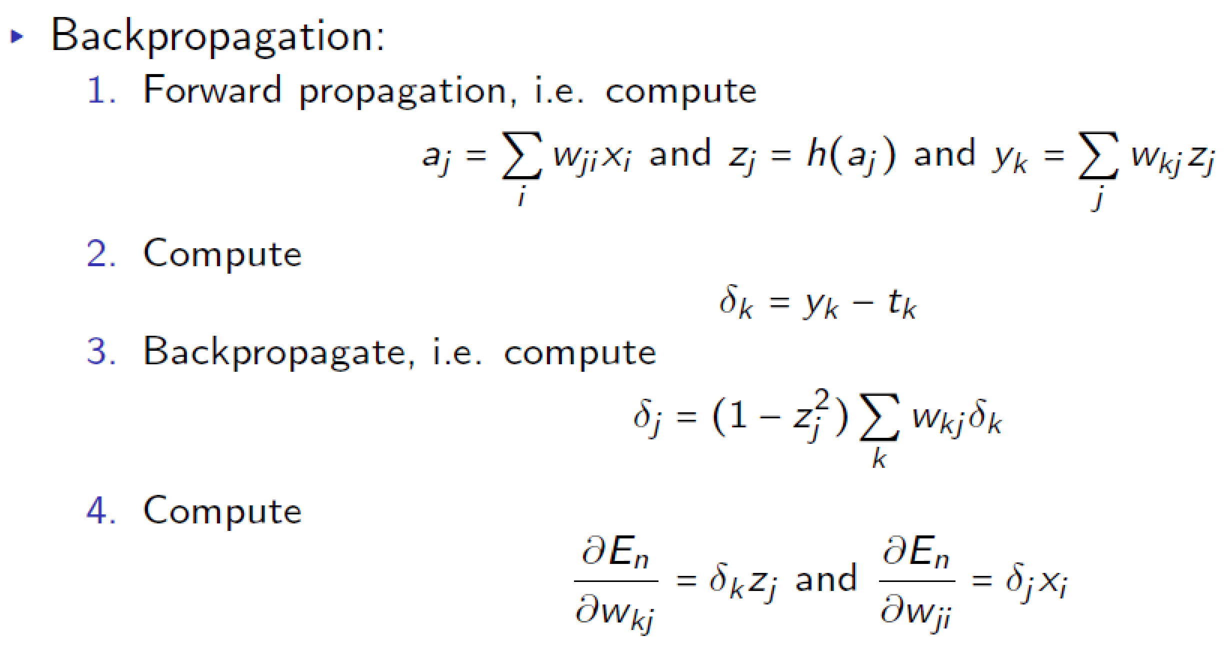
Hint: density() function does not have predict() function but it evaluates predictions on a given grid. To make prediction for a vector of new values, you may call density() several times and specify one prediction point at a time, i.e. interval [a,b]=[x(i),x(i)].

```r
# Transforms a confusion matrix
# to a data frame. It assumes
# a 2 by 2 binary confusion matrix.
prettyfy_cm = function(cm_table) {
  pretty_table = data.frame(list(titles=c("Blue", "Orange"),
  Bad=c(cm_table[1, 1], cm_table[2, 1]),
  Good=c(cm_table[1, 2], cm_table[2, 2])))

  colnames(pretty_table) = c("True / Predicted", "Blue", "Orange")
  return(pretty_table)
}

# Removing variables that doesn't matter.
data$index = NULL
data$sex = NULL

# Traing / test split.
set.seed(12345)
n = dim(data)[1]
id = sample(1:n, floor(n * 0.5))
train = data[id,]
test = data[-id,]
X_train = train[, c('FL', 'RW', 'CL', 'CW', 'BD')]
X_test = test[, c('FL', 'RW', 'CL', 'CW', 'BD')]
Xb_train = train[train$species == 'Blue',
  c('FL', 'RW', 'CL', 'CW', 'BD')]
Xo_train = train[train$species == 'Orange',
  c('FL', 'RW', 'CL', 'CW', 'BD')]
Xb_test = test[test$species == 'Blue',
  c('FL', 'RW', 'CL', 'CW', 'BD')]
Xo_test = test[test$species == 'Orange',
  c('FL', 'RW', 'CL', 'CW', 'BD')]
```

```r
# Estimating the parameters.
p_blue = sum(train$species == 'Blue') / nrow(train)
p_orange = 1 - p_blue

get_prob_vector = function(x, df) {
  prob = 1
  for (col_i in 1:ncol(df)) {
    prob = prob * density(df[, col_i],
    from=as.numeric(x[col_i]),
    to=as.numeric(x[col_i]))$y[1]
  }
  return(prob)
}

# Function that calculates the likelihood
# P(X | y=c).
likelihood = function(data, train) {
  probs = c()
  for (row_i in 1:nrow(data)) {
    probs = c(probs, get_prob_vector(data[row_i,], train))
  }
  return(probs)
}

# Getting the likelihood of
# blue and orange for the train and test.
l_blue_train = likelihood(X_train, Xb_train)
l_orange_train = likelihood(X_train, Xo_train)
l_blue_test = likelihood(X_test, Xb_train)
l_orange_test = likelihood(X_test, Xo_train)

# Calculating the class posterior.
p_blue_x_train = l_blue_train * p_blue /
  ((l_blue_train * p_blue) + (l_orange_train * p_orange))
p_blue_x_test = l_blue_test * p_blue /
  ((l_blue_test * p_blue) + (l_orange_test * p_orange))

# Getting the classes.
yhat_train = as.factor(ifelse(p_blue_x_train > 0.5, 'Blue', 'Organge'))
yhat_test = as.factor(ifelse(p_blue_x_test > 0.5, 'Blue', 'Organge'))

# Getting the confusion matrixes.
cm_train = table(train$species, yhat_train)
cm_test = table(test$species, yhat_test)

# Printing the output.
kable(prettyfy_cm(cm_train),
caption='Confusion matrix for NBC on the train set')
kable(prettyfy_cm(cm_test),
caption='Confusion matrix for NBC on the test set')
kable(get_metrics_cm(cm_train, cm_test),
caption='Metrics for NBC on the train and test set')
```

3.13 Nearest Shrunken Centroid Classification (NSCC)

```r
rownames(train_emails) = 1:nrow(train_emails)
x_train = t(train_emails[,-ncol(train_emails)])
y_train = train_emails[[ncol(train_emails)]]
mydata_train = list(x=x_train ,y=as.factor(y_train), geneid =
                      as.character(1:nrow(x_train)),
              genenames = rownames(x_train))

nsc_model = pamr.train(mydata_train, threshold=seq(0, 4, 0.1))
nsc_model_cv = pamr.cv(nsc_model, mydata_train)

nsc_model_cv$threshold[which.min(nsc_model_cv$error)]

genes = pamr.listgenes(nsc_model, mydata_train, threshold = 1.3)

pamr.plotcen(nsc_model, mydata_train, threshold = 1.3)

print(nsc_model_cv)
pamr.plotcv(nsc_model_cv)

# Amount of features
nrow(genes)

# 10 Most contributing features
kable(colnames(emails)[as.numeric(genes[1:10, 1])])

# Confusion Matrix
rownames(val_emails) = 1:nrow(val_emails)
x_val = t(val_emails[,-ncol(val_emails)])
y_val = val_emails[[ncol(val_emails)]]
#mydata_val = list(x=x_val ,y=as.factor(y_val),
# geneid = as.character(1:nrow(x_val)),
#               genenames = rownames(x_val))

pred_train = pamr.predict(nsc_model, newx = x_train, threshold = 2.5)
pred_val = pamr.predict(nsc_model, newx = x_val, threshold = 2.5)

matrix_train = table(train_emails$Conference, pred_train)
matrix_val = table(val_emails$Conference, pred_val)

kable(matrix_train)
kable(matrix_val)

error_train_nsc = 1 - sum(diag(matrix_train)/sum(matrix_train))
error_val_scn = 1 - sum(diag(matrix_val)/sum(matrix_val))

print(paste("Error Train:", error_train_nsc))
print(paste("Error Validation:", error_val_scn))
```



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

Figure 4: Backpropagartion Algorithm

3.14 Neural Networks (NN)

Limitations and Types

3.14.1 Backpropagation Implementation

```r
# Helper function for the derivative
# of the activation function.
tanh_prime = function(z) {
  return(1 - (tanh(z) ^ 2))
}

# Helper function for the mse.
MSE = function(y_true, y_hat) {
  mse = mean((y_true - y_hat) ^ 2)
  return(mse)
}

# Helper function for the
# forward pass.
forward = function(X, theta_1, b_1, theta_2, b_2) {

  # From input to the first hidden layer.
  Z_1 = X %*% theta_1 + matrix(b_1, ncol=10, nrow=nrow(X), byrow=TRUE)
  h_1 = tanh(Z_1)

  # From the hidden layer to the output layer.
  Z_2 = h_1 %*% theta_2 + matrix(b_2, nrow=nrow(X), ncol=1)
  # Creating a cache for the backward pass.
  cache = list(Z_1=Z_1,  h_1=h_1, Z_2=Z_2)
  return(cache)
}

set.seed(1432425)

# Initialization of the network.

Var = runif(50, 0, 10)
trva = data.frame(Var, Sin=sin(Var))
X_train = matrix(trva[1:25, 1], nrow=25, ncol=1)
y_train = matrix(trva[1:25, 2], nrow=25, ncol=1)
X_val = matrix(trva[26:50, 1], nrow=25, ncol=1)
y_val = matrix(trva[26:50, 2], nrow=25, ncol=1)

# Parameter initialization.
lr = 0.23
epochs = 25000

# Weights initialization.
# Xavier init: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf
theta_1 = matrix(rnorm(10, mean=0, sqrt(2 / 11)), nrow=1, ncol=10)
theta_2 = matrix(rnorm(10, mean=0, sqrt(2 / 11)), nrow=10, ncol=1)
b_1 = matrix(rnorm(10, mean=0, sqrt(2 / 11)), nrow=1, ncol=10)
b_2 = matrix(rnorm(1, mean=0, sqrt(2 / 1)), nrow=1, ncol=1)

# Variables that are going to
# store the losses.
loss_train = c()
loss_val = c()

# Creating the test data.
X_test = matrix(seq(0, 10, 0.1), nrow=101, ncol=1)
y_test = sin(X_test)

# Plotting the output of the neural network
# before training.
# Plotting the nn output.
cache = forward(X_test, theta_1, b_1, theta_2, b_2)
yhat_test = cache[['Z_2']]

p = ggplot() +
geom_line(aes(x=X_test, y=yhat_test, colour='Neural Network')) +
geom_line(aes(x=X_test, y=y_test, colour='Real Values')) +
scale_colour_manual(values=c("#1162bc", "#f83d69")) +
labs(x='x', y='Sin(x)', title='Untrained Neural Network')

print(p)
# Training loop.
for (epoch in 1:epochs) {

  # Selecting a random permutation
  # of the training data.
  rng_idx = sample(1:(nrow(X_train)))

  # Generating the new training sample.
  X_train_i = X_train[rng_idx, , drop=FALSE]
  y_train_i = y_train[rng_idx, , drop=FALSE]

  ################
  # Forward pass #
  ################
  forward_train = forward(X_train_i, theta_1, b_1, theta_2, b_2)
  forward_val = forward(X_val, theta_1, b_1, theta_2, b_2)

  # Calculating the errors on the
  # train and validation set.
  mse_train = MSE(y_train_i, forward_train[['Z_2']])
  mse_val = MSE(y_val, forward_val[['Z_2']])

  # Storing the losses.
  loss_train = c(loss_train, mse_train)
  loss_val = c(loss_val, mse_val)

  #################
  # Backward pass #
  #################

  # Calculating the derivatives.
  delta = (- (1 / nrow(X_train_i)) * (y_train_i - forward_train[['Z_2']]))
  d_theta_2 = t(forward_train[['h_1']]) %*% delta
  d_theta_1 = t(delta) %*% (tanh_prime(forward_train[['Z_1']]) *
    (X_train_i %*% t(theta_2)))

  d_b_2 = mean(delta)
  d_b_1 = t(delta) %*%
  (tanh_prime(forward_train[['Z_1']]) *
  matrix(t(theta_2), ncol=10, nrow=nrow(X_train), byrow=TRUE))

  # Updating the parameters.
  theta_2 = theta_2 - lr * d_theta_2
  theta_1 = theta_1 - lr * d_theta_1
  b_2 = b_2 - lr * d_b_2
  b_1 = b_1 - lr * d_b_1
}

# Plotting the error.
p = ggplot() +
geom_line(aes(x=(1:epochs), y=loss_train, colour='Train')) +
geom_line(aes(x=(1:epochs), y=loss_val, colour='Validation')) +
labs(x='Epoch', y='MSE', colour='Data Set',
title='Error of the neural network') +
scale_colour_manual(values=c("#1162bc", "#f83d69"))
print(p)
```

```r
# Plotting the nn output.
cache = forward(X_test, theta_1, b_1, theta_2, b_2)
yhat_test = cache[['Z_2']]

# Plotting the train / val results.
cache_train = forward(X_train, theta_1, b_1, theta_2, b_2)
cache_val = forward(X_val, theta_1, b_1, theta_2, b_2)
yhat_train = cache_train[['Z_2']]
yhat_val = cache_val[['Z_2']]

# Train.
p = ggplot() +
geom_point(aes(x=X_train, y=y_train, colour='Original')) +
geom_point(aes(x=X_train, y=yhat_train, colour='Predicted')) +
  scale_colour_manual(values=c("#1162bc", "#f83d69")) +
labs(x='x', y='Sin(x)', colour='Type', shape='Type', title='Training data set')
print(p)

# Test.
p = ggplot() +
geom_point(aes(x=X_train, y=y_train, colour='Original')) +
geom_point(aes(x=X_train, y=yhat_train, colour='Predicted')) +
scale_colour_manual(values=c("#1162bc", "#f83d69")) +
labs(x='x', y='Sin(x)', colour='Type', shape='Type', title='Validation data set')
print(p)

# Plotting the whole space [0, 10].
p = ggplot() +
geom_line(aes(x=X_test, y=yhat_test, colour='Neural Network')) +
geom_line(aes(x=X_test, y=y_test, colour='Real Values')) +
scale_colour_manual(values=c("#1162bc", "#f83d69")) +
labs(x='x', y='Sin(x)', title='Trained Neural Network')
print(p)
```

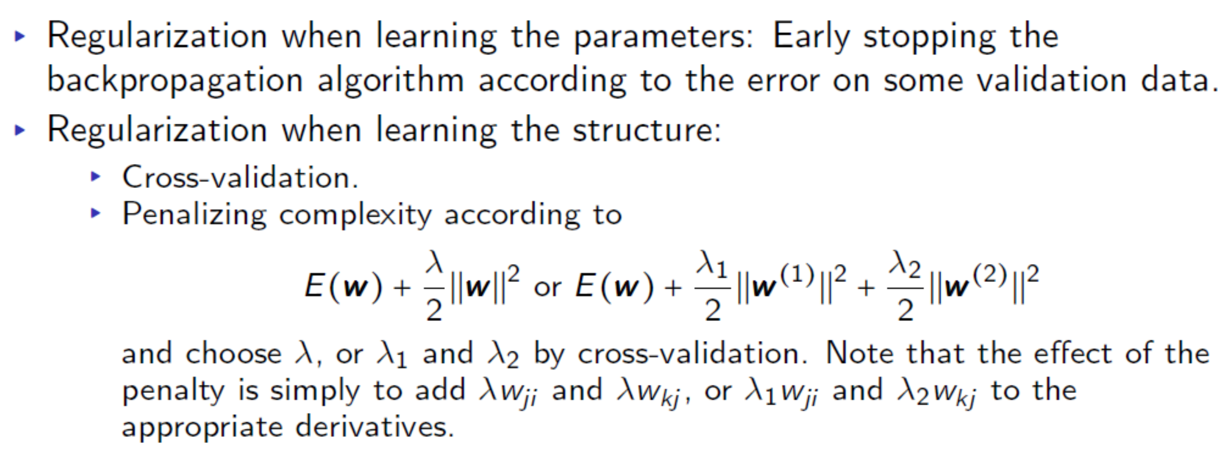
3.14.2 Library

```r
# Generating data
set.seed(1234567890)
Var = runif(50, 0, 10)
trva = data.frame(Var, Sin = sin(Var))

# Training and validation split
tr = trva[1:25, ] # Training
va = trva[26:50, ] # Validation
nn_val_res_df = data.frame()

# Random initialization of the weights in the interval [-1, 1]
set.seed(1234567890)
w_init = runif(31, -1, 1)

for(i in 1:10) {
  print(paste("Running NN: ", i))
  set.seed(1234567890)

  # Training neural network
  nn = neuralnet(Sin ~ Var, data = tr, hidden = 10,
  startweights = w_init, threshold = i / 1000)

  # Predicting values for train and validation
  va_res = neuralnet::compute(nn, va$Var)$net.result
  tr_res = neuralnet::compute(nn, tr$Var)$net.result

  # Computing train and validation MSE
  tr_mse = mean((tr_res - tr$Sin)^2)
  va_mse = mean((va_res - va$Sin)^2)

  # Storing data in data frame
  nn_val_res_df = rbind(nn_val_res_df,
  data.frame(thres_num = i, thres_val = i / 1000,
  val_mse = va_mse, trn_mse = tr_mse))
}

# Plot of MSE vs threshold for train and validation
ggplot(nn_val_res_df) +
  geom_point(aes(x = thres_val, y = val_mse, color = "Validation")) +
  geom_line(aes(x = thres_val, y = val_mse, color = "Validation")) +
  geom_point(aes(x = thres_val, y = trn_mse, color = "Train")) +
  geom_line(aes(x = thres_val, y = trn_mse, color = "Train")) +
  xlab("Threshold") + ylab("MSE") + labs(color = "Data") +
  scale_x_continuous(breaks = (1:10)/1000) +
  ggtitle("Neural Network - MSE vs Threshold for Train and Validation")

# Final neural network
# Best threshold = 0.001
opt_nn = neuralnet(Sin ~ Var, data = tr, hidden = 10,
  startweights = w_init, threshold = 0.001)
  plot(x = opt_nn, rep = "best", information = F)

# Plot of the predictions and the data
nn_pred_df = tr
nn_pred_df$Type = "Training Data"
nn_pred_df = rbind(nn_pred_df, data.frame(Var = va$Var,
  Sin = neuralnet::compute(opt_nn, va$Var)$net.result,
  Type = "NN Prediction \nfor validation"))

ggplot(nn_pred_df, aes(x = Var, y = Sin, color = Type)) + geom_point() +
ggtitle("Comparison of neural network prediction with training data") +
labs(color = "Legend")
```

3.14.3 Regularization

3.15 Partial Least Squares Regression (PLS)

Cannot find the slides to this or from where this is coming from. Maybe an old exam?!



- 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

$$E(\mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||^2 \text{ or } E(\mathbf{w}) + \frac{\lambda_1}{2}||\mathbf{w}^{(1)}||^2 + \frac{\lambda_2}{2}||\mathbf{w}^{(2)}||^2$$

  and choose λ, or λ_1 and λ_2 by cross-validation. Note that the effect of the penalty is simply to add λw_{ji} and λw_{kj}, or $\lambda_1 w_{ji}$ and $\lambda_2 w_{kj}$ to the appropriate derivatives.

Figure 5: Regularization

3.16 Quadratic Discriminant Analysis

Is a generative classifier, main assumptions: x is now **random** as well as y.

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

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

3.17 Ridge Regression

λ, also called the *Shrinkage Parameter*, penalizes the coefficients to decrease the number of coefficients to prevent overfitting. This is due to the fact that the *Shrinkage Penalty*

$$\lambda \sum_j \beta_j^2$$

is added to the term of calculating the estimates $\hat{\beta}^R$. If $\lambda = 0$ we're back to least squares estimates.

```r
covariates_ridge = scale(tecator_data[,2:(ncol(tecator_data)-3)])
response_ridge = tecator_data$Fat

glm_model_ridge = glmnet(as.matrix(covariates_ridge),
                         response_ridge, alpha = 0, family="gaussian")
plot(glm_model_ridge, xvar="lambda")
```

3.18 StepAIC (AIC)

```r
model = lm(formula = c_formula, data = tecator_data)
model.stepAIC = stepAIC(model, direction = c("both"), trace = FALSE)
summary(model.stepAIC)
```

3.19 Support Vector Machines (SVM)

```
model_svm_05 =
  ksvm(type ~ ., train_spam, kernel = "rbfdot",
       kpar = list(sigma = 0.05), C = 0.5)

model_svm_05_prediction = predict(model_svm_05, newdata = val_spam)

cm_svm_05 = table(val_spam$type, model_svm_05_prediction)

error_svm_05 = 1 - sum(diag(cm_svm_05)/sum(cm_svm_05))
```

3.20 Trees

```
decisionTree_deviance = tree(good_bad ~ ., data = train, split = "deviance")
decisionTree_gini = tree(good_bad ~ ., data = train, split = "gini")

prediction_deviance_train =
  predict(decisionTree_deviance, newdata = train, type = "class")
prediction_deviance_test =
  predict(decisionTree_deviance, newdata = test, type = "class")

summary(decisionTree_deviance)
summary(decisionTree_gini)
```

3.20.1 Optimal Tree

```
trainScore = rep(0, 15)
testScore = rep(0, 15)

for(i in 2:15) {
  prunedTree = prune.tree(decisionTree_deviance, best = i)
  pred = predict(prunedTree, newdata = valid, type = "tree")
  trainScore[i] = deviance(prunedTree)
  testScore[i] = deviance(pred)
}

## Add one as we trim the first index
optimalTreeIdx = which.min(testScore[-1]) + 1
optimalTreeScore = min(testScore[-1])

print(optimalTreeIdx)
print(optimalTreeScore)
```

3.20.2 Plot Deviance

```
plot(2:15, trainScore[2:15], type = "b", col = "orange", ylim = c(250,650),
     main = "Tree Depth vs Training/Test Score", ylab = "Deviance",
     xlab = "Number of Leaves")
```

```
points(2:15, testScore[2:15], type = "b", col = "blue")
legend("topright", legend = c("Train (orange)", "Test (blue)"))
```

3.20.3 Prune a tree and print it

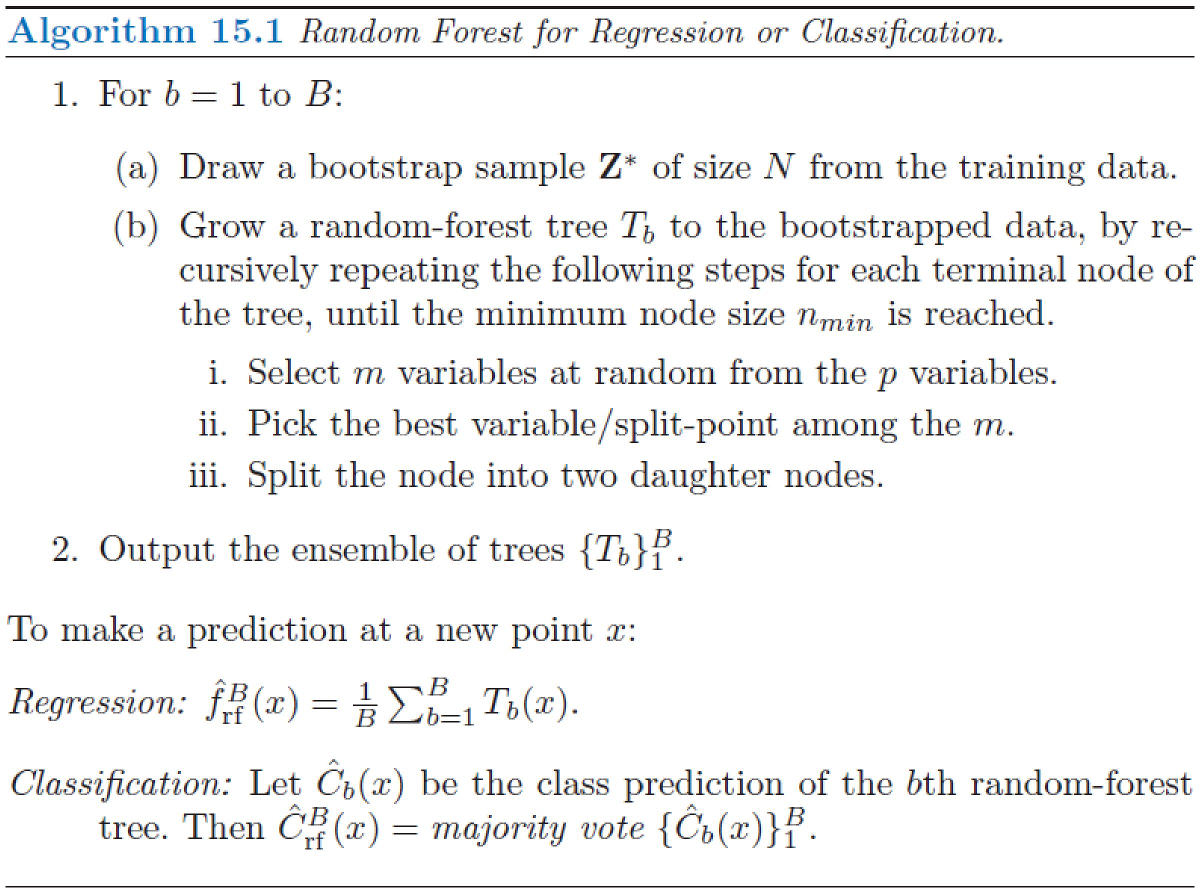
```
optimalTree = prune.tree(decisionTree_deviance, best = optimalTreeIdx)
plot(optimalTree)
text(optimalTree, pretty = 1)
title("Optimal Tree")
```

3.20.4 Regression Tree with Deviance

```
# Create the model
reg_tree = tree(EX ~ MET, data = statedata, control =
                  tree.control(nobs = nrow(statedata), minsize = 8))

# Use cross validation
cross_val_reg_tree = cv.tree(reg_tree)

# Plot the deviance of the sizes
plot(cross_val_reg_tree)
```

3.20.5 Pruned tree with residuals

```
# Let's create the pruned the with best set to 3 and get it's prediction
pruned_tree = prune.tree(reg_tree, best = 3)
pruned_tree_prediction = predict(pruned_tree, newdata = statedata, type = "vector")

# We create a data.frame to save our values to make it easier to plot the data
pruned_tree_plot_dataframe =
  data.frame(statedata$MET, statedata$EX, pruned_tree_prediction,
             pruned_tree_prediction-statedata$EX)
names(pruned_tree_plot_dataframe) = c("met", "orignal_ex", "predicted_ex", "residual")

# Lets first plot the pruned tree
plot(pruned_tree)
text(pruned_tree, pretty = 1)
title("Optimal Tree with best = 3")

# Lets create a plot with the real and predictes values and highlight the
# residuals
ggplot(pruned_tree_plot_dataframe) +
  geom_point(aes(x = pruned_tree_plot_dataframe$met,
                 y = pruned_tree_plot_dataframe$orignal_ex),
             color = "black") +
  geom_point(aes(x = pruned_tree_plot_dataframe$met,
                 y = pruned_tree_plot_dataframe$predicted_ex),
             color = "darkblue") +
  geom_segment(mapping=aes(x=pruned_tree_plot_dataframe$met,
```


Algorithm 15.1 *Random Forest for Regression or Classification.*

```
1. For $b = 1$ to $B$:
    (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size $N$ from the training data.
    (b) Grow a random-forest tree $T_b$ to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size $n_{min}$ is reached.
        i. Select $m$ variables at random from the $p$ variables.
        ii. Pick the best variable/split-point among the $m$.
        iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.
```

To make a prediction at a new point x:

Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$.

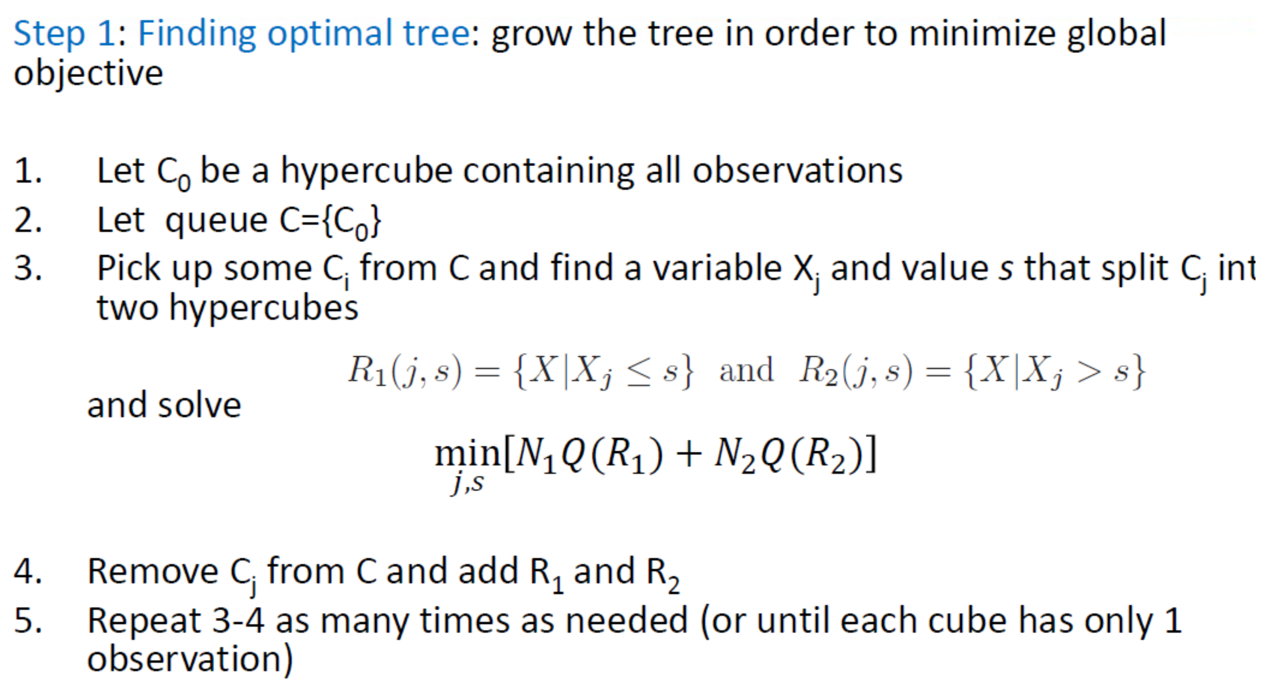
Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

Figure 6: Random Forest Algorithm

```
                             y=pruned_tree_plot_dataframe$orignal_ex,
                             xend=pruned_tree_plot_dataframe$met,
                             yend=pruned_tree_plot_dataframe$predicted_ex),
                 color = "red", linetype = "dotted") +
  labs(title = "Original Data, Fitted Data and Residuals", y = "EX",
       x = "MET", color = "Legend")
```

3.20.6 Random Forest

```
c_randomForest =
    randomForest(formula = c_formula, data = train_spambase, ntree = i)
```



Step 1: Finding optimal tree: grow the tree in order to minimize global objective

1. Let C_0 be a hypercube containing all observations
2. Let queue C={C_0}
3. Pick up some C_i from C and find a variable X_j and value *s* that split C_j int two hypercubes

$$R_1(j,s) = \{X | X_j \leq s\} \text{ and } R_2(j,s) = \{X | X_j > s\}$$

and solve

$$\min_{j,s}[N_1 Q(R_1) + N_2 Q(R_2)]$$

4. Remove C_j from C and add R_1 and R_2
5. Repeat 3-4 as many times as needed (or until each cube has only 1 observation)

Figure 7: Cart

3.20.7 CART

4 Feature Reduction

4.1 Independent Component Analysis (ICA)

```r
set.seed(12345)

ica_model = fastICA(X = nir_spectra_copy, n.comp = 2, alg.typ = "parallel",
                    fun = "logcosh", alpha = 1, method = "R", row.norm = FALSE,
                    maxit = 200, tol = 0.0001, verbose = FALSE)

W_dash = ica_model$K %*% ica_model$W

plot(W_dash[,1], main= "Column 1")
plot(W_dash[,2], main= "Column 2")

ggplot(as.data.frame(W_dash)) +
  geom_point(aes(x = W_dash[,1],
                 y = W_dash[,2]),
             color = "orange") +
    labs(title = "Feature 1 vs Feature 2", y = "Feature 2",
       x = "Feature 1", color = "Legend")
```

4.2 Linear Discriminant Analysis (LDA)

lda() in package mass

4.2.1 Implementation

```r
#Helper function to calculate the
# classification metrics given a
# confusion matrix.
# Rows (0, 1) must be from the classifier
# Columns (0, 1) must be true values.
analyze_cm = function(cm)
{
  cm_df = as.data.frame(cm)
  recall = cm_df[4, "Freq"] / (cm_df[4, "Freq"] + cm_df[2, "Freq"])
  precision = cm_df[4, "Freq"] / (cm_df[4, "Freq"] + cm_df[3, "Freq"])
  accuracy = (cm_df[1, "Freq"] + cm_df[4, "Freq"]) / sum(cm_df[, "Freq"])
  mcr = 1 - accuracy
  fpr = cm_df[3, "Freq"] / (cm_df[1, "Freq"] + cm_df[3, "Freq"])

  return(list(Recall=recall,  Precision=precision,  Missclasification=mcr,
  Accuracy=accuracy,  FPR=fpr))
}

# Helper function to get metrics from a confusion matrix.
get_metrics_cm = function(cm_train, cm_test)
{
  metrics = data.frame(analyze_cm(cm_train))
  metrics = rbind(metrics, analyze_cm(cm_test))
  rownames(metrics) = c("Train", "Test")
  return(t(metrics))
}

# Transforms a confusion matrix
# to a data frame. It assumes
# a 2 by 2 binary confusion matrix.
prettyfy_cm = function(cm_table)
{
  pretty_table = data.frame(list(titles=c("Female", "Male"),
  Bad=c(cm_table[1, 1], cm_table[2, 1]),
  Good=c(cm_table[1, 2], cm_table[2, 2])))
  colnames(pretty_table) = c("True / Predicted", "Female", "Male")
  return(pretty_table)
}

# Reading the data.
data = read.csv("australian-crabs.csv")

# Getting the data.
X = data[, c('RW', 'CL')]
y = data[, 'sex']
X_m = as.matrix(data[data$sex == 'Male', c('RW', 'CL')])
X_f = as.matrix(data[data$sex == 'Female', c('RW', 'CL')])
```

```r
# Setting up the parameters for K=2.
K = 2 # Number of classes.
pi_m = sum(data$sex == 'Male') / nrow(data)
pi_f = 1 - pi_m
mu_m = colMeans(data[data$sex == 'Male', c('RW', 'CL')])
mmu_m = matrix(mu_m, ncol=ncol(X_m), nrow=nrow(X_m),
               byrow=TRUE) # Matrix form of mu_m.
mu_f = colMeans(data[data$sex == 'Female', c('RW', 'CL')])
mmu_f = matrix(mu_f, ncol=ncol(X_f), nrow=nrow(X_f),
               byrow=TRUE) # Matrix form of mu_f.
sigma_m = t(X_m - mmu_m) %*% (X_m - mmu_m)
sigma_f = t(X_f - mmu_f) %*% (X_f - mmu_f)

sigma = (sigma_m + sigma_f) / (nrow(data) - K)
sigma_inv = solve(sigma)
beta_m = sigma_inv %*% mu_m
beta_f = sigma_inv %*% mu_f
gamma_m = -0.5 * (t(mu_m) %*% sigma_inv %*% mu_m) + log(pi_m)
gamma_f = -0.5 * (t(mu_f) %*% sigma_inv %*% mu_f) + log(pi_f)

# Cretating the decision boundary.
# The input should be in the space of RW
# the output is on the space of CL.
boundary = function(x) {
  y = ((gamma_f - gamma_m) - x * (beta_m[1] - beta_f[1])) / (beta_m[2] - beta_f[2])
  return(y)
}

x_boundary = seq(0, 25, 1)
y_boundary = sapply(x_boundary, boundary)
data_boundary = sapply(data$RW, boundary)
classification = ifelse(data$CL > data_boundary, 'Male', 'Female')
p = ggplot() +
geom_point(aes(x=data$RW, y=data$CL, colour=classification)) +
geom_line(aes(x=x_boundary, y=y_boundary), color='red') +
labs(colour='Sex', y='CL', x='RW',
title='Classification of australian crabs given CL and RW')
print(p)

# Creating the confusion matrix and
# getting the metrics of the classification.
cm = table(data$sex, classification)
kable(prettyfy_cm(cm), caption='Confusion matrix for LDA')
df_metrics = as.data.frame(get_metrics_cm(cm, cm)[, 'Train'])
colnames(df_metrics) = 'Train'
kable(df_metrics, caption='Metrics for LDA on the train set')
```

4.2.2 Library

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

```r
# Maybe needed? options(scipen=999)

df = read.csv("australian-crabs.csv")

p_true = ggplot(df, aes(x = RW, y = CL, fill = sex, color = sex)) +
  geom_point(shape = 23, size = 2, alpha = 0.8) +
  scale_x_continuous(limits = c(5, 21), breaks = seq(5, 20, 5)) +
  scale_y_continuous(limits = c(14, 50), breaks = seq(20, 50, 10)) +
  labs(color = "Sex", fill = "Sex", title = "Scatterplot of CL by RW") +
  theme_minimal() + theme(plot.title = element_text(hjust = 0.5))

p_true
```

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

```r
# Conduct lda -----------------------------------------------------------------

# Note: by default, prior is set to the proportions of Male:Female: c(0.5, 0.5)
# To check if the results are the same, add prior = c(0.5, 0.5) as param to lda()

lda.model.train = MASS::lda(sex ~ CL + RW, data = df) # LDA
lda.pred.train = predict(lda.model.train, df) # predict (probs, class etc.)
lda.class.train = lda.pred.train$class # extract class
df$sex_lda = lda.class.train # add class to df

# Compare plots ---------------------------------------------------------------
p_true = p_true + labs(title = "Color = true sex")

p_lda_prop = ggplot(df, aes(x = RW, y = CL, fill = sex_lda, color = sex_lda)) +
  geom_point(shape = 23, size = 2, alpha = 0.8) +
  scale_x_continuous(limits = c(5, 21), breaks = seq(5, 20, 5)) +
  scale_y_continuous(limits = c(14, 50), breaks = seq(20, 50, 10)) +
  labs(color = "Sex", fill = "Sex", title = "Color = classified sex") +
  theme_minimal() + theme(plot.title = element_text(hjust = 0.5))

p = ggpubr::ggarrange(plotlist = list(p_true, p_lda_prop), ncol = 2,
  common.legend = TRUE, legend = "bottom")

ggpubr::annotate_figure(p, top = "Plot of CL by RW [priors: 0.5 Male, 0.5 Female]")
## gridExtra does not allow for common legend unfortunately
# plot_list_arranged = gridExtra::arrangeGrob(grobs = list(p1, p2), ncol = 2)

# gridExtra::grid.arrange(plot_list_arranged, top = "Scatterplot of CL by RW",
# padding = unit(0.5, "line"))

# Misclassification -----------------------------------------------------------
misclassification_cnt = sum(ifelse(df$sex_lda == df$sex, 0, 1))
cat("Misclassification count [LDA]: ", misclassification_cnt, "\n")

misclassification_rate = mean(ifelse(df$sex_lda == df$sex, 0, 1))
cat("Misclassification rate [LDA]: ", misclassification_rate, "\n")
```

```r
table(df$sex, df$sex_lda, dnn = c("True", "Predicted"))
```

Repeat step 2 but use priors $p(Male) = 0.9$ and $p(Female) = 0.1$ instead. How did the classification result change and why?

```r
# Conduct lda -------------------------------------------------------------

# Note: to understand which value is for which class, do table(df$sex)
# The order of the printed categories is the order of the priors to specify.

lda.model.train = MASS::lda(sex ~ CL + RW, data = df, prior = c(0.1, 0.9)) # LDA
lda.pred.train = predict(lda.model.train, df) # predict (probs, class etc.)
lda.class.train = lda.pred.train$class # extract class
df$sex_lda = lda.class.train # add class to df

# Misclassification -------------------------------------------------------
misclassification_cnt = sum(ifelse(df$sex_lda == df$sex, 0, 1))
cat("Misclassification count [LDA]: ", misclassification_cnt, "\n")

misclassification_rate = mean(ifelse(df$sex_lda == df$sex, 0, 1))

cat("Misclassification rate [LDA]: ", misclassification_rate, "\n")
table(df$sex, df$sex_lda, dnn = c("True", "Predicted"))

# Compare plots -----------------------------------------------------------
p_true = p_true + labs(title = "Color = true sex")

p_lda_unprop = ggplot(df, aes(x = RW, y = CL, fill = sex_lda, color = sex_lda)) +
  geom_point(shape = 23, size = 2, alpha = 0.8) +
  scale_x_continuous(limits = c(5, 21), breaks = seq(5, 20, 5)) +
  scale_y_continuous(limits = c(14, 50), breaks = seq(20, 50, 10)) +
  labs(color = "Sex", fill = "Sex", title = "Color = classified sex") +
  theme_minimal() + theme(plot.title = element_text(hjust = 0.5))

p = ggpubr::ggarrange(plotlist = list(p_true, p_lda_unprop), ncol = 2,
  common.legend = TRUE, legend = "bottom")

ggpubr::annotate_figure(p, top = "Plot of CL by RW [priors: 0.9 Male, 0.1 Female]")
```

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

```r
log.model.train = glm(sex ~ CL + RW, data = df, family = "binomial") # logistic
log.pred.train = predict(log.model.train, df, type = "response") # Prediction
log.class.train = ifelse(log.pred.train > 0.5, "Male", "Female") # Classification
df$sex_log = as.factor(log.class.train) # add class to df

# Misclassification -------------------------------------------------------
misclassification_cnt = sum(ifelse(df$sex_log == df$sex, 0, 1))
cat("Misclassification count [Logistic]: ", misclassification_cnt, "\n")

misclassification_rate = mean(ifelse(df$sex_log == df$sex, 0, 1))

cat("Misclassification rate [Logistic]: ", misclassification_rate, "\n")
```

```r
table(df$sex, df$sex_log, dnn = c("True", "Predicted"))

# Comparison LDA vs. Logistic --------------------------------------------------
# Conduct LDA again to explore difference between LDA and logistic
lda.model.train = MASS::lda(sex ~ CL + RW, data = df) # LDA
lda.pred.train = predict(lda.model.train, df) # predict (probs, class etc.)
lda.class.train = lda.pred.train$class # extract class
df$sex_lda = lda.class.train # add class to df
table(df$sex_lda, df$sex_log, dnn = c("LDA", "Log"))

# Compare plots ----------------------------------------------------------------
p_lda_prop = p_lda_prop + labs(title = "LDA (proportional prior)")

p_log = ggplot(df, aes(x = RW, y = CL, fill = sex_log, color = sex_log)) +
geom_point(shape = 23, size = 2, alpha = 0.8) +
  scale_x_continuous(limits = c(5, 21), breaks = seq(5, 20, 5)) +
  scale_y_continuous(limits = c(14, 50), breaks = seq(20, 50, 10)) +
  labs(color = "Sex", fill = "Sex", title = "Logistic regression") +
  theme_minimal() + theme(plot.title = element_text(hjust = 0.5))
  p = ggpubr::ggarrange(plotlist = list(p_lda_prop, p_log), ncol = 2,
  common.legend = TRUE, legend = "bottom")

ggpubr::annotate_figure(p, top = "Plot of CL by RW")

# Decision boundary ------------------------------------------------------------
# Extract and print the boundary
slope = coef(log.model.train)[3]/(-coef(log.model.train)[2])
intercept = coef(log.model.train)[1]/(-coef(log.model.train)[2])

cat("Decision boundary: y_hat = ", intercept, " + x * ", slope, "\n")

db = data.frame(slope = slope, intercept = intercept)

p = ggplot(df, aes(x = RW, y = CL, fill = sex_log, color = sex_log)) +
  geom_point(shape = 23, size = 2, alpha = 0.8) +
  scale_x_continuous(limits = c(5, 21), breaks = seq(5, 20, 5)) +
  scale_y_continuous(limits = c(14, 50), breaks = seq(20, 50, 10)) +
  labs(color = "Sex", fill = "Sex", title = "Logistic regression") +
  theme_minimal() + theme(plot.title = element_text(hjust = 0.5))

p + geom_abline(intercept = eval(intercept), slope = eval(slope),
  color = "steelblue3", linetype = "dashed", size = 1)
```

4.3 Principal Component Analysis (PCA)

```r
# Copy to not modify the original dataset
nir_spectra_copy = nir_spectra
nir_spectra_copy$Viscosity = c()

# PCA
res = prcomp(nir_spectra_copy)
```

```
# Eigenvalues
lambda = res$sdev^2

# Proportion of variation
kable(head(sprintf("%2.3f",lambda/sum(lambda)*100)),
      caption = "Variance for each Feature")

# Plot
screeplot(res, main = "Variances for each Feature")

# PC1 vs PC2
ggplot(as.data.frame(res$x)) +
  geom_point(aes(x = res$x[,1],
                 y = res$x[,2]),
             color = "orange") +
    labs(title = "PC1 vs. PC2", y = "PC2",
       x = "PC1", color = "Legend")
```

4.3.1 Trace Plots

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

4.3.2 Kernel PCA

`kpca` in package `kernlab`.

```
library(kernlab)
K = as.kernelMatrix(crossprod(t(x)))
res = kpca(K)
barplot(res@eig)
plot(res@rotated[,1], res@rotated[,2], xlab="PC1",ylab="PC2")
```

4.4 Regularized Discriminant Analysis

`rda()` in package `klaR`.

4.5 Regularized Logistic Regression

`LiblineaR()` in package `LiblineaR`.

5 Miscellaneous

5.1 Benjamin-Hochberg Algorithm

```r
# Orignal Data Set: emails
# H0: Feature has effect on Conference
# Ha: Feature has no effect on Conference

# 1) Calculate p-values
# 2) Assign ranks and sort
# 3) BH Critical Value
# 4) Find critical value and select all p < score

q_value = 0.05

## 1)

feature_names = c()
p_values = c()

# For each data point calculate the p_value

for (i in 1:(ncol(emails)-1)) {
  colname = colnames(emails)[i]
  c_formula = paste(sep = "", colname, " ~ Conference")
  p_value = t.test(as.formula(c_formula), data = emails,
                   alternative="two.sided")

  feature_names = c(feature_names, colname)
  p_values = c(p_values, p_value$p.value)
}

## 2)

# Sorting
bh_entries = data.frame(feature_names, p_values)
bh_entries = bh_entries[order(bh_entries$p_values, decreasing = FALSE),]
rownames(bh_entries) = NULL

## 3)

# Define the function for the CV-Score
getCV_BH = function(i, m, Q) {
  return(i/m*Q)
}

cv_scores = c()

for (j in 1:nrow(bh_entries)) {
  current_val = getCV_BH(as.numeric(rownames(bh_entries)[j]),
                         nrow(bh_entries), q_value)
  cv_scores = c(cv_scores, current_val)
}

#cv_col = apply(bh_entries, 1, function(x)
#getCV_BH(as.numeric(rownames(bh_entries)), nrow(bh_entries), q_value))
bh_entries = data.frame(bh_entries, cv_scores)
```

```r
## 4)
## Find all rows where p_values < cv_scores
bh_entries = bh_entries[bh_entries$p_values < bh_entries$cv_scores,]

kable(bh_entries)
```

5.2 Bootstrapping

5.2.1 Non-Parametric Bootstrap with Confidence Bands

```r
# We take the function given from the slides and adjust to the tree
# computing bootstrap samples
f_non_p_bootstrap = function(data, ind) {

  # First take the subsample
  data1 = data[ind,]

  # Now create a tree with the same hyperparameters from that subsample
  tree_model = tree(EX ~ MET, data = data1,
                    control = tree.control(nrow(data),minsize = 8))
  tree_model_pruned = prune.tree(tree_model, best = 3)

  # Use that model to predict on the real data
  prediction = predict(tree_model_pruned, newdata = data)
  return(prediction)
}

# Lets create the Bootstrap (again taken from slides)
res = boot(statedata, f_non_p_bootstrap, R = 1000)

# Confidence Bands using envelope
ci_non_p_bootstrap = envelope(res)
ci_non_p_bootstrap_df = as.data.frame(t(ci_non_p_bootstrap$point))
names(ci_non_p_bootstrap_df) = c("upper_bound", "lower_bound")
pruned_tree_plot_dataframe =
  data.frame(pruned_tree_plot_dataframe, ci_non_p_bootstrap_df)

# Plot the data
ggplot(pruned_tree_plot_dataframe) +
  geom_point(aes(x = pruned_tree_plot_dataframe$met,
                 y = pruned_tree_plot_dataframe$orignal_ex),
             color = "black") +
  geom_ribbon(aes(x = pruned_tree_plot_dataframe$met,
                  ymin = ci_non_p_bootstrap_df$lower_bound,
                  ymax = ci_non_p_bootstrap_df$upper_bound),
              alpha = 0.4, fill = "orange", color = "orange3") +
    labs(title = "Confidence Bands (non-parametric)", y = "EX",
       x = "MET", color = "Legend")
```

5.2.2 Parametric Bootstrap with Confidence Bands

```r
# Again we take the sample from the slides and adjust it to our needs
# 1) Compute value mle
# 2) Write function ran.gen that depends on data and mle and which generates
# new data
# 3) Write function statistic that depend on data which will be generated by
# ran.gen and should return the estimator

## 1)
mle = pruned_tree

## 2)
rng = function(data, mle) {
  data1 = data.frame(EX=data$EX, MET=data$MET)
  n = length(data$EX)
  #generate new Price
  # summary needed to access the residuals
  data1$EX = rnorm(n, predict(mle, newdata=data1), sd(summary(mle)$residuals))
  return(data1)
}

## 3) f_non_p_bootstrap + distribution N
f_p_bootstrap = function(data) {

  # The index is not needed any more as we don't take a sub-sample

  # Now create a tree with the same hyperparameters from that subsample
  tree_model = tree(EX ~ MET, data = data,
                    control = tree.control(nrow(data), minsize = 8))
  tree_model_pruned = prune.tree(tree_model, best = 3)

  # Use that model to predict on the real data
  prediction = predict(tree_model_pruned, newdata = data)

  return(prediction)
}

# Bootstrap
res2 = boot(statedata,
            statistic = f_p_bootstrap, R=1000, mle=mle,
            ran.gen=rng, sim="parametric")

# Confidence Bands using envolope
ci_p_bootstrap = envelope(res2)
ci_p_bootstrap_df = as.data.frame(t(ci_p_bootstrap$point))
names(ci_p_bootstrap_df) = c("upper_bound", "lower_bound")
pruned_tree_plot_dataframe_p =
  data.frame(pruned_tree_plot_dataframe, ci_p_bootstrap_df)

# Plot the data
ggplot(pruned_tree_plot_dataframe_p) +
  geom_point(aes(x = pruned_tree_plot_dataframe_p$met,
                 y = pruned_tree_plot_dataframe_p$orignal_ex),
             color = "black") +
  geom_ribbon(aes(x = pruned_tree_plot_dataframe_p$met,
                  ymin = ci_p_bootstrap_df$lower_bound,
                  ymax = ci_p_bootstrap_df$upper_bound),
              alpha = 0.4, fill = "orange", color = "orange3") +
    labs(title = "Confidence Bands (parametric)", y = "EX",
       x = "MET", color = "Legend")
```

5.2.3 Parametric Bootstrap with Prediction Bands

```
# Prediction Bands

# from slides
f_p_bootstrap_pb = function(data) {

  # The index is not needed any more as we don't take a sub-sample

  # Now create a tree with the same hyperparameters from that subsample
  tree_model = tree(EX ~ MET, data = data,
                    control = tree.control(nrow(data), minsize = 8))
  tree_model_pruned = prune.tree(tree_model, best = 3)

  # Use that model to predict on the real data
  prediction = predict(tree_model_pruned, newdata = data)

  # Add the rnrom to the prediction
  prediction_normal = rnorm(nrow(data), prediction, sd(summary(mle)$residual))

  return(prediction_normal)
}

# Bootstrap
res3 = boot(statedata, statistic = f_p_bootstrap_pb,
            R=1000, mle=mle, ran.gen=rng, sim="parametric")

# Confidence Bands using envolope
pb_p_bootstrap = envelope(res3)
pb_p_bootstrap_df = as.data.frame(t(pb_p_bootstrap$point))
names(pb_p_bootstrap_df) = c("upper_bound", "lower_bound")
pruned_tree_plot_dataframe_p_pb =
  data.frame(pruned_tree_plot_dataframe, pb_p_bootstrap_df)

# Plot the data
ggplot(pruned_tree_plot_dataframe_p_pb) +
  geom_point(aes(x = pruned_tree_plot_dataframe_p_pb$met,
                 y = pruned_tree_plot_dataframe_p_pb$orignal_ex),
             color = "black") +
  geom_ribbon(aes(x = pruned_tree_plot_dataframe_p_pb$met,
                  ymin = pb_p_bootstrap_df$lower_bound,
                  ymax = pb_p_bootstrap_df$upper_bound), alpha = 0.4,
              fill = "orange", color = "orange3") +
    labs(title = "Prediction Bands (parametric)", y = "EX",
       x = "MET", color = "Legend")
```

5.3 Cross-Validation

5.3.1 Cross Validation Plot

**This was aksed somewhere (maybe exam?), so clarify what's meant here!

5.3.2 K-Fold

```
c_cross_validation = function(k = 5, Y, X) {

  if (!is.numeric(X) && ncol(X) == 0) {
    y_hat = mean(Y)
    return(mean((y_hat-Y)^2))
  }

  Y = as.matrix(Y)
  X = as.matrix(X)
  X = cbind(1, X)

  # Create a list of 5 matrices with the appropiate
  # size (these will hold the subsets)
  X_subsets = list()
  Y_subsets = list()

  # We fill the list entries with the subsets
  for (i in 1:k) {
    percentage_marker = nrow(X)/k
    start = floor(percentage_marker*(i-1)+1)
    end = floor(percentage_marker*i)
    X_subsets[[i]] = X[start:end,]
    Y_subsets[[i]] = Y[start:end]
  }

  # Now we take one matrix at a time for training and
  # everything else as the testing
  scores = 0
  for (i in 1:k) {

    ## Initial
    X_train = matrix(0, ncol = ncol(X))
    Y_train = c()

    # Get validation and training data
    current_subset_X = X_subsets[-i]
    current_subset_Y = Y_subsets[-i]
    for (j in 1:(length(X_subsets)-1)) {
      X_train = rbind(X_train, current_subset_X[[j]])
      Y_train = c(Y_train, current_subset_Y[[j]])
    }

    # Because of R
    X_train = X_train[-1,]

    # Model
    betas =
      as.matrix((solve(t(X_train) %*% X_train)) %*% t(X_train) %*% Y_train)

    ## Select the training data and transform them to
    # one matrix X_test and one vector Y_test
    X_val = X_subsets[[i]]
    Y_val = Y_subsets[[i]]

    ## Now we get our y_hat and y_real. y_real is
    # only used to clarify the meaning
    y_hat = as.vector(X_val %*% betas)
    y_real = Y_val

    ## Get MSE and add to the scores list
    scores = c(scores, mean((y_hat - y_real)^2))

  }
  # Return the mean of our scores
  scores = scores[-1]
  return(mean(scores))
}

c_best_subset_selection = function(Y, X) {

  # Shuffle X and Y via indexes
  ids = sample(x = 1:nrow(X), nrow(X))
  X = X[ids,]
  Y = Y[ids]

  # Get all combinations
  comb_matrix = matrix(0, ncol = ncol(X))
  for (i in c(1:(2^(ncol(X))-1))) {
    comb_matrix =
      rbind(comb_matrix, tail(rev(as.numeric(intToBits(i))), ncol(X)))
  }

  results = c()

  # Do cross validation for each feature set
  for (j in 1:nrow(comb_matrix)) {
    comb = as.logical(comb_matrix[j,])
    feature_select = X[,comb]
    res = c_cross_validation(5, Y, feature_select)
    results = c(results, res)
  }
  models = matrix(results, ncol = 1)
  models = cbind(models, comb_matrix)
```

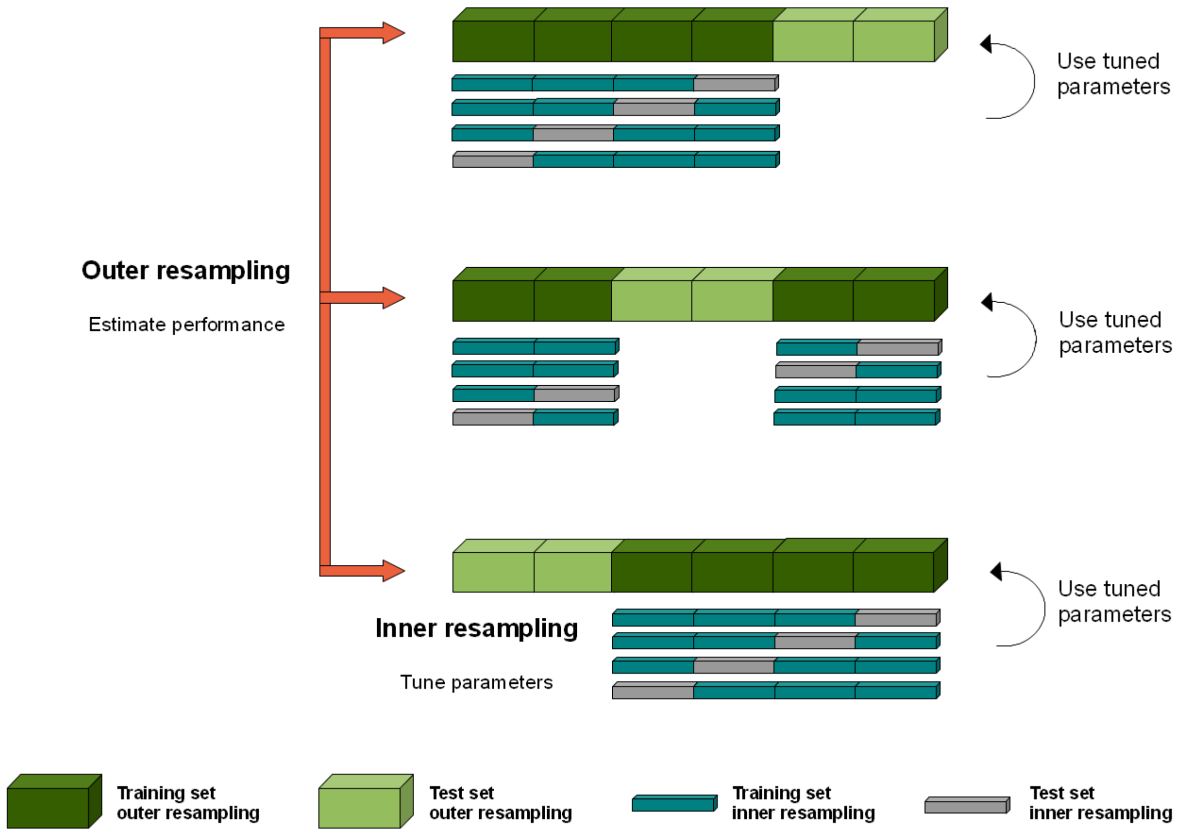


Figure 8: Nested Cross Validation

```
  # Add column with the sum of the features for plotting
  feature_sum = c()
  for (k in 1:nrow(comb_matrix)) {
    row_sum = sum(comb_matrix[k,])
    feature_sum = c(feature_sum, row_sum)
  }
  models = as.data.frame(cbind(feature_sum, models))
  colnames(models)[1:2] = c("Sum", "Score")
  print(ggplot(models, aes(x = Sum, y = Score, colour = factor(feature_sum))) +
    geom_point())
   stat_summary(fun.y = min, colour = "red", geom = "point", size = 5)
  return(models[min(models[,2]) == models[,2],])
}
```

5.3.3 Nested Cross Validation

- Divide the dataset into K cross-validation folds at random.
- For each fold $k = 1, 2, ., K$: outer loop for evaluation of the model with selected hyperparameter
  - Let `test` be fold k
  - Let `trainval` be all the data except those in fold k
  - Randomly split `trainval` into L folds

    - For each fold $l = 1, 2, ., L$: inner loop for hyperparameter tuning
        * Let `val` be fold l
        * Let `train` be all the data except those in `test` or `val`
        * Train with each hyperparameter on `train`, and evaluate it on `val`. Keep track of the performance metrics
    - For each hyperparameter setting, calculate the average metrics score over the L folds, and choose the best hyperparameter setting.
    - Train a model with the best hyperparameter on `trainval`. Evaluate its performance on `test` and save the score for fold k.
- Calculate the mean score over all K folds, and report as the generalization error.

5.4 EM-Algorithm

Let's have a look at the mathematical equations and how we can derive our formulas for the matrix multiplication from that. Formulas without a source are either taken from the lecture slides or derived by previous formulas.

The first step is to calculate Z. We will divide that in first calculating Bx which contains $Bernoulli(x|\mu_k)$ and afterwards calculating $p(x)$ which we can use to calculate Z. The formulas (left side) will be assigned to a letter which you will find in the source code.

$$Bernoulli(x|\mu_k) = B_x = \prod_{i} \mu^{x_i}_{k_i} (1 - \mu_{k_i})^{1-x_i}$$

`Bernoulli(x|\mu_k) = B_x = \prod_{i} \mu^{x_i}_{k_i} (1 - \mu_{k_i})^{1-x_i}`

For using matrix multiplication we need to get rid of the product and the exponents, so we use ln on both sides:

$$\ln(B) = \sum \ln (\mu^{x_i}_{k_i}) x_i + \sum \ln (1-\mu_{x_i}) (1 - x_i)$$

`\ln(B) = \sum \ln (\mu^{x_i}_{k_i}) x_i + \sum \ln (1-\mu_{x_i}) (1 - x_i)`

Now let's get rid of the ln on the left side:

$$B_x = e^{\sum \ln (\mu^{x_i}_{k_i}) x_i + \sum \ln (1-\mu_{x_i}) (1 - x_i)}$$

`B_x = e^{\sum \ln (\mu^{x_i}_{k_i}) x_i + \sum \ln (1-\mu_{x_i}) (1 - x_i)}`

This craves to be put into a neat matrix multiplication. Okay, let's look for $p(x)$.

$$p(x) = P_x =\sum_k \pi_k Bernoulli(x|\mu_k) = \sum_k \pi_k B$$

`p(x) = P_x =\sum_k \pi_k Bernoulli(x|\mu_k) = \sum_k \pi_k B`

We will use $P(x)$ later to calculate the likelihood L but for now let's calculate Z:

$$P(z_{nk}|x_n,\mu,\pi) = Z = \frac{\pi_k p(x_n|\mu_k)}{\sum_k p(x_n|\mu_k)}$$

`P(z_{nk}|x_n,\mu,\pi) = Z = \frac{\pi_k p(x_n|\mu_k)}{\sum_k p(x_n|\mu_k)}`

The likelihood L is given by the following equation which can be found in Pattern Recognition on page 433 equation 9.14.

$$\ln p(X|\pi, \mu, \Sigma) = L = \sum_{n=1}^{N} ln\{ \sum_{k=1}^{K}\pi_k N(x_n|\mu_k,\Sigma_k) \}$$

\ln p(X|\pi,\mu,\Sigma) = L = \sum_{n=1}^{N} ln\{ \sum_{k=1}^{K}\pi_k N(x_n|\mu_k,\Sigma_k) \}

As we already have the value inside the curly braces ($P(x)$) it's basically just the sum of the logarithms over n.

For calculating π we use the following.

$$\pi_{k}^{ML} = pi = \frac{\sum_k p(z_{nk|x_n|\mu|\pi})}{N}$$

\pi_{k}^{ML} = pi = \frac{\sum_k p(z_{nk|x_n|\mu|\pi})}{N}

And finally we use the following for calculating μ. Note that the nominator of π and the denominator of μ are the same.

$$\mu_{k}^{ML} = mu = \frac{\pi_k p(z_{nk|x_n|\mu|\pi})}{\sum_k \pi_k p(z_{nk|x_n|\mu|\pi})}$$

\mu_{k}^{ML} = mu = \frac{\pi_k p(z_{nk|x_n|\mu|\pi})}{\sum_k \pi_k p(z_{nk|x_n|\mu|\pi})}

Voila we're done, now the coding is actually just a few lines of code, you'll find it in the appendix.

```r
set.seed(1234567890)
max_it = 100 # max number of EM iterations
min_change = 0.1 # min change in log likelihood between two consecutive EM iterations
N=1000 # number of training points
D=10 # number of dimensions
x = matrix(nrow=N, ncol=D) # training data
true_pi = vector(length = 3) # true mixing coefficients
true_mu = matrix(nrow=3, ncol=D) # true conditional distributions

true_pi=c(1/3, 1/3, 1/3)
true_mu[1,]=c(0.5,0.6,0.4,0.7,0.3,0.8,0.2,0.9,0.1,1)
true_mu[2,]=c(0.5,0.4,0.6,0.3,0.7,0.2,0.8,0.1,0.9,0)
true_mu[3,]=c(0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5)

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

plot(true_mu[1,], type="o", col="blue", ylim=c(0,1))
points(true_mu[2,], type="o", col="red")
points(true_mu[3,], type="o", col="green")

kable(true_pi, caption = "true_pi")
kable(true_mu, caption = "true_mu")

################################################################################
```

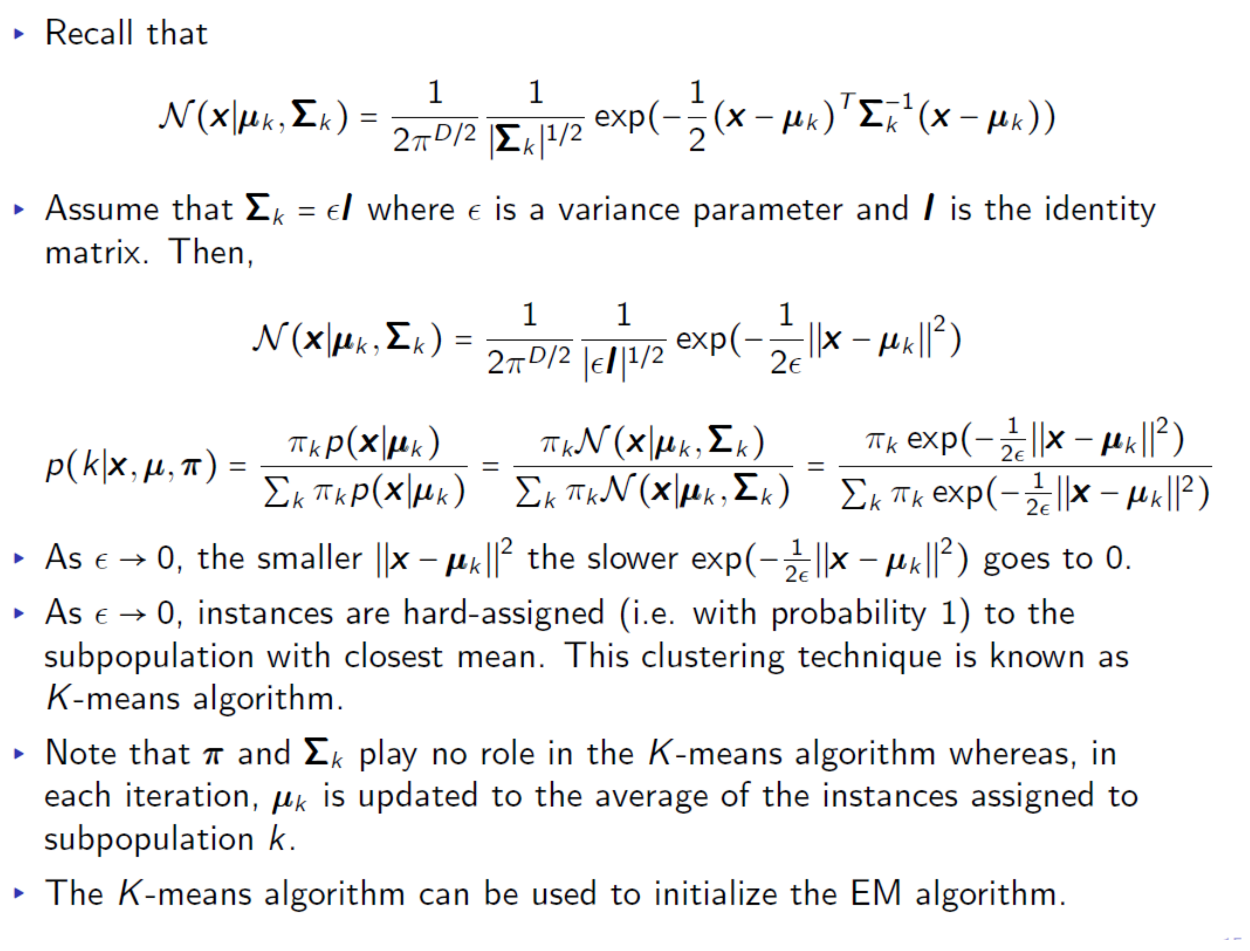
```r
# K = 3
################################################################################

set.seed(1234567890)
K = 3 # number of guessed components
z = matrix(nrow=N, ncol=K) # fractional component assignments
pi = vector(length = K) # mixing coefficients
mu = matrix(nrow=K, ncol=D) # conditional distributions
llik = vector(length = max_it) # log likelihood of the EM iterations
# Random initialization of the paramters
pi = runif(K,0.49,0.51)
pi = pi / sum(pi)
for(k in 1:K) {
  mu[k,] = runif(D,0.49,0.51)
}

for(it in 1:max_it) {
  # E-step: Computation of the fractional component assignments
  Bx = exp(x %*% log(t(mu)) + (1-x) %*% log(t(1-mu)))
  Px = Bx * rep(pi, nrow(Bx))
  Z = Px / rowSums(Px)
  #Log likelihood computation.
  L = sum(log(rowSums(Px)))
  llik[it] = L

  # Stop if the lok likelihood has not changed significantly
  if (it > 1 && abs(llik[it-1] - llik[it]) < min_change) break

  #M-step: ML parameter estimation from the data and fractional component assignments
  pi = colSums(Z) / N
  mu = (t(Z) %*% x) / colSums(Z)
}

kable(pi, caption = "pi")
kable(mu, caption = "mu")
kable(it, caption = "Number of Iterations")
kable(llik[it], caption = "Ln-Likelihood")
plot(mu[1,], type="o", col="blue", ylim=c(0,1))
points(mu[2,], type="o", col="red")
points(mu[3,], type="o", col="green")
plot(llik[1:it], type="o")
```

5.5 Holdout-Principle

Divide into training, validation and test set. Code is found in the snippets section.

5.6 Hypothesis Testing

```r
res = t.test(MFCC_2nd.coef~Quality,data=data, alternative="two.sided")
res$p.value
```



- Recall that

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k) = \frac{1}{2\pi^{D/2}}\frac{1}{|\boldsymbol{\Sigma}_k|^{1/2}}\exp(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_k)^T\boldsymbol{\Sigma}_k^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_k))$$

- Assume that $\boldsymbol{\Sigma}_k = \epsilon\boldsymbol{I}$ where ϵ is a variance parameter and $\boldsymbol{I}$ is the identity matrix. Then,

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k) = \frac{1}{2\pi^{D/2}}\frac{1}{|\epsilon\boldsymbol{I}|^{1/2}}\exp(-\frac{1}{2\epsilon}||\boldsymbol{x}-\boldsymbol{\mu}_k||^2)$$

$$p(k|\boldsymbol{x},\boldsymbol{\mu},\boldsymbol{\pi}) = \frac{\pi_k p(\boldsymbol{x}|\boldsymbol{\mu}_k)}{\sum_k \pi_k p(\boldsymbol{x}|\boldsymbol{\mu}_k)} = \frac{\pi_k\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)}{\sum_k \pi_k\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)} = \frac{\pi_k \exp(-\frac{1}{2\epsilon}||\boldsymbol{x}-\boldsymbol{\mu}_k||^2)}{\sum_k \pi_k \exp(-\frac{1}{2\epsilon}||\boldsymbol{x}-\boldsymbol{\mu}_k||^2)}$$

- As $\epsilon \to 0$, the smaller $||\boldsymbol{x}-\boldsymbol{\mu}_k||^2$ the slower $\exp(-\frac{1}{2\epsilon}||\boldsymbol{x}-\boldsymbol{\mu}_k||^2)$ goes to 0.
- As $\epsilon \to 0$, instances are hard-assigned (i.e. with probability 1) to the subpopulation with closest mean. This clustering technique is known as K-means algorithm.
- Note that $\boldsymbol{\pi}$ and $\boldsymbol{\Sigma}_k$ play no role in the K-means algorithm whereas, in each iteration, $\boldsymbol{\mu}_k$ is updated to the average of the instances assigned to subpopulation k.
- The K-means algorithm can be used to initialize the EM algorithm.

Figure 9: K-Means Algorithm

```
res = oneway_test(MFCC_2nd.coef~as.factor(Quality), data=data,paired=FALSE)
pvalue(res)
```

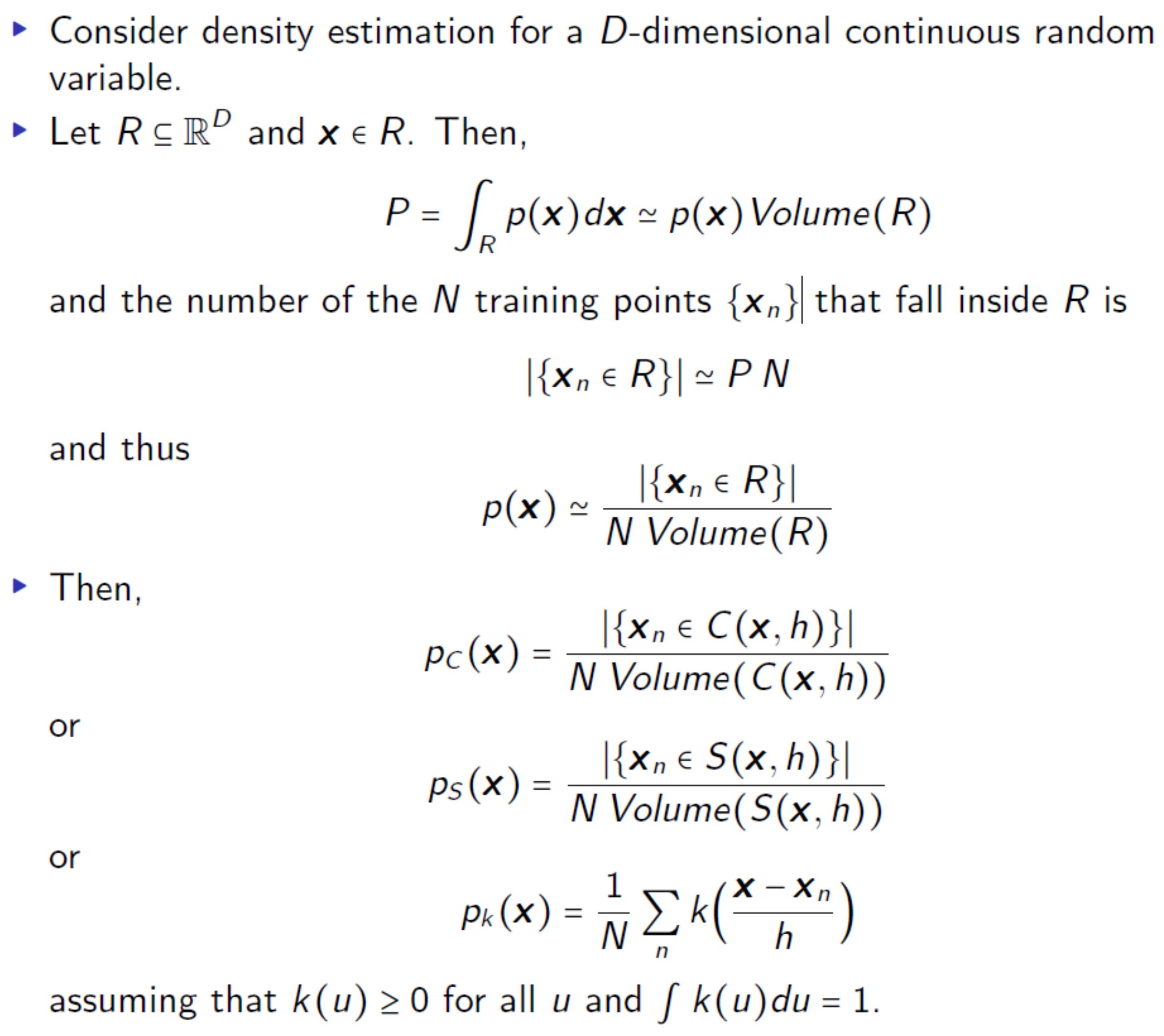
5.7 K-Means Algorithm

1. Assign each point to a cluster (a.k.a subpopulation) at random
2. Compute the cluster centroids as the averages of the points assigned to each cluster
3. Repeat until the centroids do not change
4. Assign each point to the cluster with the closest centroid
5. Update the cluster centroids as the averages of the points assigned to each cluster

5.8 Histogram, Moving Window and Kernel Density Estimation

5.8.1 Density Estimation

```
density_estimation = function(data, K = 6, X) {

  N = length(data)
```



- 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 $\int k(u) du = 1$.

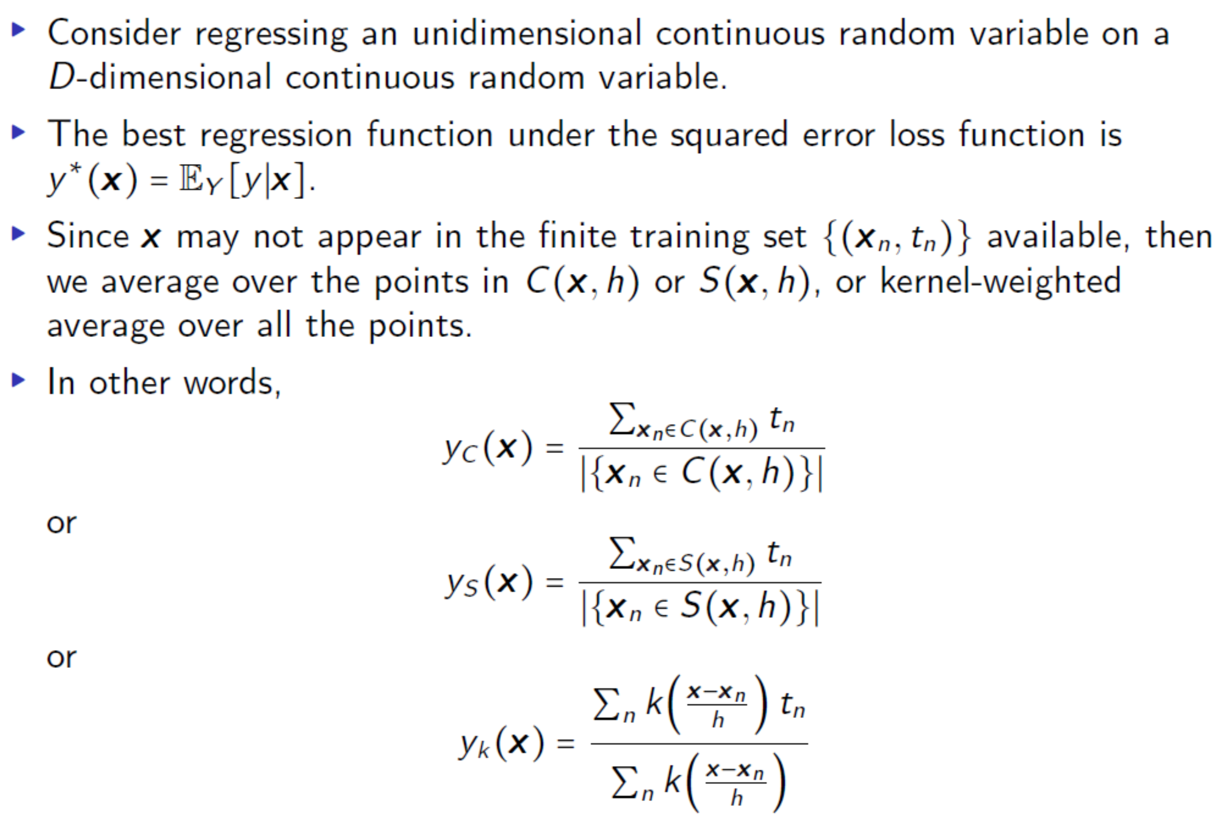
Figure 10: Histogram, Moving Window and Kernel Density Estimation

```r
  S = data

  # V needs to be calculated, first get all distances
  distances = abs(X - S)

  # Sort them the get the K nearest and get their indexes
  sorted_distances = sort(distances, index.return = TRUE)
  idx_knearest = sorted_distances$ix[1:K]

  # We take the point which is furthest away and take it as the "radius" to get
  # the V (linear)
  V = 2 * max(abs(data[idx_knearest[K]] - X))

  # Returns based on formula from slides
  return(K/(N*V))
}

# Get the min and max from the dataset and define a stepsize
min_speed = min(cars$speed)
max_speed = max(cars$speed)
steps = 1000

# X-Values are a sequence from min_speed to max_speed
x_values =   seq(min_speed, max_speed, by = (max_speed - min_speed)/steps)

# Y-Values are the density estimation at each point
y_values = unlist(lapply(x_values,
                         function(x) density_estimation(cars$speed, K = 6, x)))

# Put them into a dataframe
density_data_frame = data.frame(x_values, y_values)
colnames(density_data_frame) = c("X", "Density")

# Plot the data
print(ggplot(density_data_frame) +
  geom_histogram(data = cars, bins = 21) + aes(x = speed, y = ..density..) +
  geom_line(aes(x = X, y = Density, colour = "Density Function")) +
  labs(title="Density of cars$speed", y="Density", x="X", color = "Legend") +
  scale_color_manual(values = c("orange")))
```

Epanechnikov kernel Exam 2015 2.3

5.9 Kernel Methods

```r
cl = makeCluster(detectCores())

min_distance = function(a, b, ringSize) {

  boundOne = parSapply(cl, b, FUN = function(x) abs(a - x))
  boundTwo = parSapply(cl, b, FUN = function(x) abs(a - ringSize - x))
  boundThree = parSapply(cl, b, FUN = function(x) abs(a + ringSize - x))
  return(pmin(boundOne, boundTwo, boundThree))
}

## Kernels
## The definition for the Guassian Kernel is taken from the slides, page 6.

kernel_gauss_distance = function(pointA, pointB, smoothing) {

  # Use distHaversine() as a help
  m = cbind(pointB$longitude, pointB$latitude)
  u = distHaversine(m, c(pointA$longitude, pointA$latitude))
  u = u / smoothing
  return(exp(-(u^2)))
}

kernel_gauss_day = function(dayA, dayB, smoothing) {

  dayA_in_days = as.numeric(strftime(as.Date(dayA$date), '%j'))
  dayB_in_days = as.numeric(strftime(as.Date(dayB$date), '%j'))

  u = min_distance(dayA_in_days, dayB_in_days, 365)
  u = u / smoothing
  return(exp(-(u^2)))
}

kernel_gauss_hour = function (hourA, hourB, smoothing) {

  hourA_in_h = parSapply(cl, hourA$time, FUN = function(x)
    as.numeric(difftime(strptime(x, format = "%H:%M:%S"),
                        strptime("00:00:00", format = "%H:%M:%S"))))

  hourB_in_h = parSapply(cl, hourB$time, FUN = function(x)
    as.numeric(difftime(strptime(x, format = "%H:%M:%S"),
                        strptime("00:00:00", format = "%H:%M:%S"))))

  u = min_distance(hourA_in_h, hourB_in_h, 24)
  u = u / smoothing
  return(exp(-(u^2)))
}

# Students' code here
pred_temp_sum = predict_weather(latitude, longitude, date)

plot_data_frame =
  data.frame(as.POSIXct(paste(date ,pred_temp_sum$times),
                        format = "%Y-%m-%d %H:%M"), pred_temp_sum[2:3])
colnames(plot_data_frame) = c("hour", "predWithSum", "predWithProd")

ggplot(plot_data_frame) +
  geom_line(aes(x = hour, y = predWithSum, group = 1,
                colour = "Kernel Regression (Sum)")) +
  geom_point(aes(x = hour, y = predWithSum), colour = "orange") +
  geom_line(aes(x = hour, y = predWithProd, group = 1,
```



- Consider regressing an unidimensional continuous random variable on a D-dimensional continuous random variable.
- The best regression function under the squared error loss function is $y^*(\boldsymbol{x}) = \mathbb{E}_Y[y|\boldsymbol{x}]$.
- Since $\boldsymbol{x}$ may not appear in the finite training set $\{(\boldsymbol{x}_n, t_n)\}$ available, then we average over the points in $C(\boldsymbol{x}, h)$ or $S(\boldsymbol{x}, h)$, or kernel-weighted average over all the points.
- In other words,

$$y_C(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{x}_n \in C(\boldsymbol{x},h)} t_n}{|\{\boldsymbol{x}_n \in C(\boldsymbol{x}, h)\}|}$$

or

$$y_S(\boldsymbol{x}) = \frac{\sum_{\boldsymbol{x}_n \in S(\boldsymbol{x},h)} t_n}{|\{\boldsymbol{x}_n \in S(\boldsymbol{x}, h)\}|}$$

or

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

Figure 11: Histogram, Moving Window and Kernel Regression

```r
                colour = "Kernel Regression (Prod)")) +
  geom_point(aes(x = hour, y = predWithProd), colour = "blue") +
  labs(title = "Temperature Prediction with Kernel Regression (Sum and Prod)",
       y = "Temperature", x = "Hour of the Day", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))
```

5.9.1 Histogram, Moving Window and Kernel Regression

5.9.2 Histogram Classification

5.9.3 Moving Window Classification

5.9.4 Kernel Classification

5.10 Loss-Matrix

```r
L = matrix(c(0, 10, 1, 0), nrow = 2)
colnames(L) = c("Predicted", "Predicted")
rownames(L) = c("good", "bad")
kable(L)
```

- 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 $\boldsymbol{x}$.

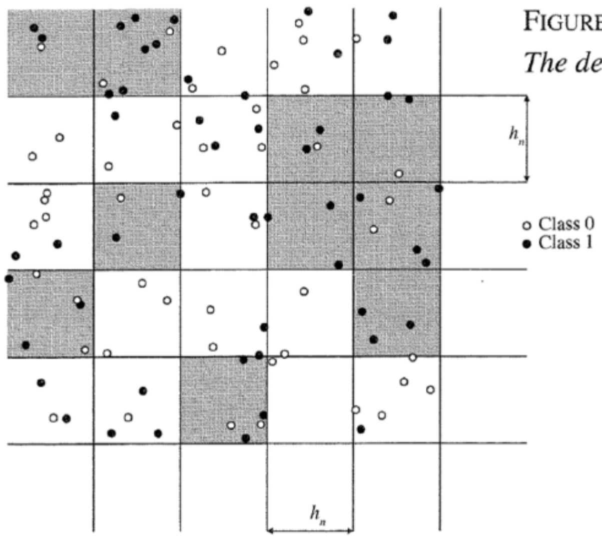


FIGURE 6.1. *A cubic histogram rule: The decision is* 1 *in the shaded area.*

- In other words,

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

Figure 12: Histogram 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.

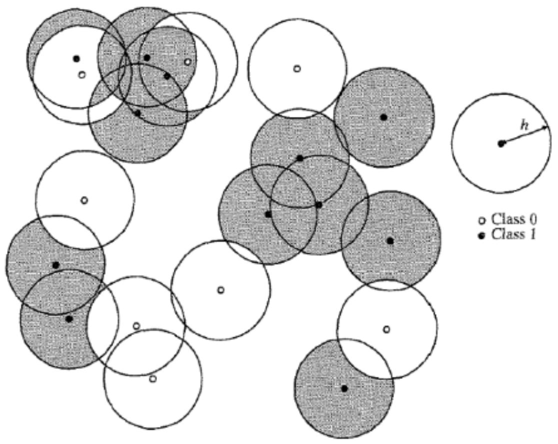


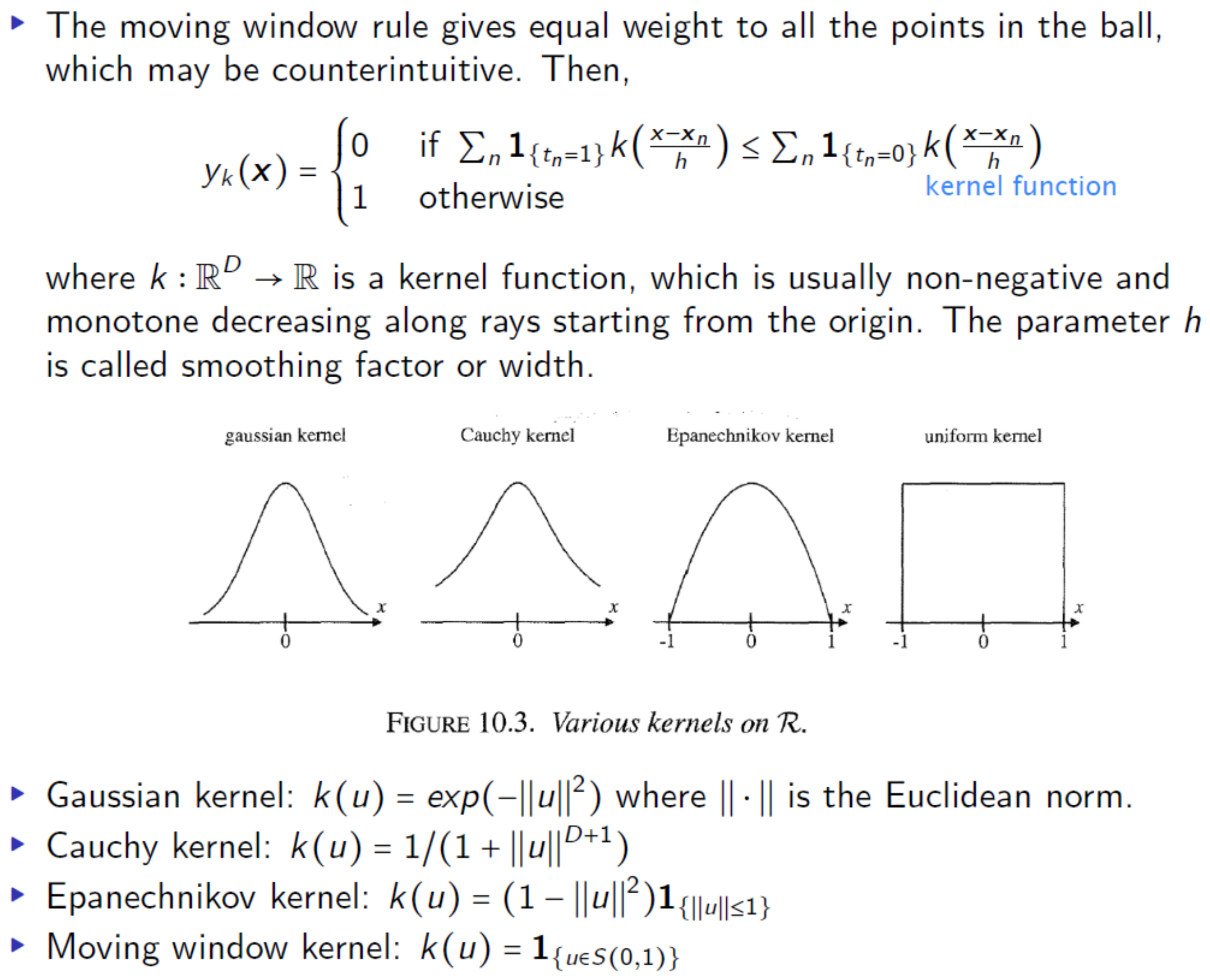
FIGURE 10.1. *The moving window rule in $\mathcal{R}^2$. The decision is* 1 *in the shaded area.*

- 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 $\boldsymbol{x}$.

Figure 13: Moving Window Classification



- The moving window rule gives equal weight to all the points in the ball, which may be counterintuitive. Then,

$$y_k(\mathbf{x}) = \begin{cases} 0 & \text{if } \sum_n \mathbf{1}_{\{t_n=1\}} k\left(\frac{\mathbf{x}-\mathbf{x}_n}{h}\right) \leq \sum_n \mathbf{1}_{\{t_n=0\}} k\left(\frac{\mathbf{x}-\mathbf{x}_n}{h}\right) \\ 1 & \text{otherwise} \end{cases}$$

kernel function

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.

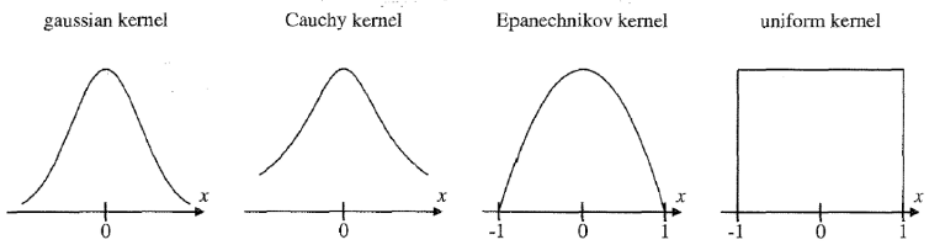


FIGURE 10.3. *Various kernels on $\mathcal{R}$.*

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

Figure 14: Kernel Classification

```r
# Prediction
prediction_bayes_train_raw =
  predict(naiveBayesModel, newdata = train, type = "raw")
prediction_bayes_test_raw =
  predict(naiveBayesModel, newdata = test, type = "raw")

confusion_matrix_bayes_train =
  table(prediction_bayes_train_raw[,2]/
          prediction_bayes_train_raw[,1] > 10, train$good_bad)
confusion_matrix_bayes_test =
  table(prediction_bayes_test_raw[,2]/
          prediction_bayes_test_raw[,1] > 10, test$good_bad)

error_bayes_train_raw = 1 - sum(diag(confusion_matrix_bayes_train)/
                              sum(confusion_matrix_bayes_train))
error_bayes_test_raw =  1 - sum(diag(confusion_matrix_bayes_test)/
                              sum(confusion_matrix_bayes_test))
```

5.11 ROC-Curve

5.11.1 Calculate ROC

```r
# prediction optimal tree
prediction_optimalTree_test_p =
  predict(optimalTree, newdata = test, type = "vector")
# prediction naive bayes
prediction_bayes_test_p =
  predict(naiveBayesModel, newdata = test, type = "raw")

pi = seq(from = 0.00, to = 1.0, by = 0.05)
fprs_tree = c()
tprs_tree = c()
fprs_bayes = c()
tprs_bayes = c()

for (i in pi) {
  current_tree_pi_confusion =
    table(test$good_bad, factor(prediction_optimalTree_test_p[,2] > i,
                                lev=c(TRUE, FALSE)))
  current_bayes_pi_confusion =
    table(test$good_bad, factor(prediction_bayes_test_p[,2] > i,
                                lev=c(TRUE, FALSE)))

  # FPR = FP / N-
  # TPR = TP / N+
  fprs_tree = c(fprs_tree, current_tree_pi_confusion[1,1]/
                 sum(current_tree_pi_confusion[1,]))
  tprs_tree = c(tprs_tree, current_tree_pi_confusion[2,1]/
                  sum(current_tree_pi_confusion[2,]))

  fprs_bayes = c(fprs_bayes, current_bayes_pi_confusion[1,1]/
                   sum(current_bayes_pi_confusion[1,]))
  tprs_bayes = c(tprs_bayes, current_bayes_pi_confusion[2,1]/
                   sum(current_bayes_pi_confusion[2,]))
}

roc_values = data.frame(fprs_tree, tprs_tree, fprs_bayes, tprs_bayes)
```

5.11.2 Print ROC

```
ggplot(roc_values) +
  geom_line(aes(x = fprs_tree, y = tprs_tree,
                colour = "ROC Optimized Tree")) +
  geom_point(aes(x = fprs_tree, y = tprs_tree), colour = "orange") +

  geom_line(aes(x = fprs_bayes, y = tprs_bayes,
                colour = "ROC Naive Bayes")) +
  geom_point(aes(x = fprs_bayes, y = tprs_bayes), colour = "blue") +

  geom_abline(slope=1, intercept=0, linetype="dotted") +

  labs(title = "ROC for Optimized Tree and Naive Bayes", y = "TPR",
       x = "FPR", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))
```

5.12 Splines

Some sample code using GAM.

```
gam_model_additive = gam(formula =
                         Mortality ~ s(Year, k=length(unique(influanza$Year))) +
                         s(Week, k=length(unique(influanza$Week))) +
                         s(Influenza, k=length(unique(influanza$Influenza))),
                         family = gaussian(), data = influanza, method="GCV.Cp")

summary(gam_model_additive)
plot(gam_model_additive)

time = influanza$Time
observed = influanza$Mortality
predicted = fitted(gam_model_additive)

mortality_values = data.frame(time, observed, predicted)

ggplot(mortality_values) +
  geom_line(aes(x = time, y = observed,
                colour = "Observed Mortality")) +
  geom_line(aes(x = time, y = predicted,
                colour = "Predicted Mortality")) +
  labs(title = "Observed vs Predicted Mortality", y = "Mortality",
       x = "Time", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))
```