

# Focus on Classification and Supervised Learning

# 21

by Patrick Boily, with contributions from Olivier Leduc and Shintaro Hagiwara

In Chapter 19 (*Machine Learning 101*), we provided a (mostly) math-free general overview of machine learning. In this chapter, we present an introductory mathematical treatment of the discipline, with a focus on classification, ensemble learning, and non-parametric supervised methods.

Our approach once again borrows heavily from [2, 3]; explanations and examples are also available in [233, 237]. It provides a continuation of the treatment found in Chapter 20 (*Regression and Value Estimation*) and is a companion piece to Chapter 22 (*Focus on Clustering*).

## 21.1 Overview

We will discuss classification in the same context as we discussed regression/value estimation in Section 20.1; the latter should have been read before embarking on this chapter.<sup>1</sup> In particular, it is expected that readers are familiar with training sets  $\text{Tr}$  and testing sets  $\text{Te}$  for a dataset with  $n$  observations  $\mathbf{x}_1, \dots, \mathbf{x}_n$ ,  $p$  predictors  $X_1, \dots, X_p$  and response variable  $Y$  (see Figure 20.4 for details).

One important note about this chapter's notation: in the design matrix  $\mathbf{X}$ , a row corresponds to the signature vector of an observation (the values of the predictors); when we write  $\mathbf{x}$  or  $\boldsymbol{\beta}$ , we typically understand those to be **column vectors**. When in doubt, remember that all matrices/vectors involved must have compatible dimensions when multiplied or compared; that will sometimes mean that vectors must be viewed as row-vectors rather than column vectors, and *vice-versa*, depending on the context.

### 21.1.1 Formalism

In a **classification setting**, the response  $Y$  is **categorical**, which is to say that  $Y \in \mathcal{C}$ , where  $\mathcal{C} = \{C_1, \dots, C_K\}$ , but the supervised learning objectives remain the same:

- build a classifier  $C(\mathbf{x}^*)$  that assigns a label  $C_k \in \mathcal{C}$  to test observations  $\mathbf{x}^*$ ;
- understand the role of the predictors in this assignment, and
- assess the uncertainty and the accuracy of the classifier.

21.1 Overview	961
Formalism	961
Model Evaluation	963
Bias-Variance Trade-Off	963
21.2 Simple Classifiers	966
Logistic Regression	970
Discriminant Analysis	975
ROC Curve	982
21.3 Rare Occurrences	985
21.4 Other Approaches	987
Tree-Based Methods	987
Support Vector Machines	1002
Artificial Neural Networks	1018
Naïve Bayes Classifiers	1043
21.5 Ensemble Learning	1051
Bagging	1052
Random Forests	1056
Boosting	1058
21.6 Exercises	1070

1: Or, at the very least, should be read concurrently.

The main difference with the regression setting (and to be fair, it's a big one) is that we do not have access to an MSE-type metric to evaluate the classifier's performance.

The counterpart of the regression function

$$f(\mathbf{x}) = E[Y \mid \vec{X} = \mathbf{x}]$$

2: In other words, pick the most numerous categorical label of observations for which the signature vector is  $\mathbf{x}$ .

is defined as follows. For  $1 \leq k \leq K$ , let  $p_k(\mathbf{x}) = P(Y = C_k \mid \vec{X} = \mathbf{x})$ ,<sup>2</sup> the **Bayes optimal classifier** at  $\mathbf{x}$  is the function

$$C(\mathbf{x}) = C_j \quad \text{where } p_j(\mathbf{x}) = \max\{p_1(\mathbf{x}), \dots, p_K(\mathbf{x})\}.$$

As was the case or regression, it could be that there are too few observations at  $\vec{X} = \mathbf{x}$  to estimate the probability exactly, in which case we might want to allow for **nearest neighbour averaging**:

$$\hat{C}(\mathbf{x}) = C_j, \quad \text{where } \tilde{p}_j(\mathbf{x}) = \max\{\tilde{p}_1(\mathbf{x}), \dots, \tilde{p}_K(\mathbf{x})\},$$

and  $\tilde{p}_k(\mathbf{x}) = P(Y = C_k \mid \vec{X} \in N(\mathbf{x}))$  and  $N(\mathbf{x})$  is a neighbourhood of  $\mathbf{x}$ .<sup>3</sup>

3: The curse of dimensionality is also in play when  $p$  becomes too large.

The quantity that plays an analogous role to the MSE for  $\hat{C}(\mathbf{x})$  is the **misclassification error rate**:

$$\text{ERR}_{\text{Te}} = \frac{1}{M} \sum_{j=N+1}^M \mathcal{J}[y_j \neq \tilde{C}(\mathbf{x}_j)],$$

where  $\mathcal{J}$  is the **indicator function**

$$\mathcal{J}[\text{condition}] = \begin{cases} 0 & \text{if the condition is false} \\ 1 & \text{otherwise} \end{cases}$$

The Bayes optimal classifier  $C(\mathbf{x})$  is the optimal classifier with respect to  $\text{ERR}_{\text{Te}}$ ; the **Bayes error rate**

$$\eta_{\mathbf{x}} = 1 - E \left[ \max_k P(Y = C_k \mid \vec{X} = \mathbf{x}) \right]$$

corresponds to the irreducible error and provides a lower limit on any classifier's expected error.

Most classifiers build structured models  $\hat{C}(\mathbf{x})$  which **directly** approximate the Bayes optimal classifier  $C(\mathbf{x})$  (such as support vector machines or naïve Bayes classifiers), but some classifiers build structured models  $\hat{p}_k(\mathbf{x})$  for the **conditional probabilities**  $p_k(\mathbf{x})$ ,  $1 \leq k \leq K$ , which are then used to build  $\hat{C}(\mathbf{x})$ , such as logistic regression, generalized additive models, and  $k$ -nearest neighbours.

The latter models are said to be **calibrated** (i.e., the relative values of  $\hat{p}_k(\mathbf{x})$  represent relative probabilities), whereas the former are **non-calibrated**.<sup>4</sup>

4: Only the most likely outcome is provided; it is impossible to say to what extent a given outcome is more likely than another.

### 21.1.2 Model Evaluation

The **confusion matrix** of a classifier on  $\text{Te}$  is a tool to evaluate the model's performance:

		prediction	
		0	1
actual	0	TP	FN
	1	FP	TN

Here, TP stands for **true positive**, FN for **true negative**, FP for **false positive**, and TN for **true negative**. There are various **classifier evaluation metrics**; if the testing set  $\text{Te}$  has  $M$  observations, then:

- **accuracy** measures the correct classification rate  $\frac{TP+TN}{M}$ ;
- **misclassification** is  $\frac{FP+FN}{M} = 1 - \text{accuracy}$ ;
- **false positive rate** (FPR) is  $\frac{FP}{FP+TN}$ ;
- **false negative rate** (FNR) is  $\frac{FN}{TP+FN}$ ;
- **true positive rate** (TPR) is  $\frac{TP}{TP+FN}$ ;
- **true negative rate** (TNR) is  $\frac{TN}{FP+TN}$ ;

There are other measures, including the  $F_1$ -score, the Matthews' correlation coefficient, etc. [238].

One thing to remember is that we should not put all the performance evaluation eggs in the same metric basket!

### 21.1.3 Bias-Variance Trade-Off

The **bias-variance trade-off** (see Section 20.1) is also observed in classifiers, although the decomposition is necessarily different (see [2] for details).

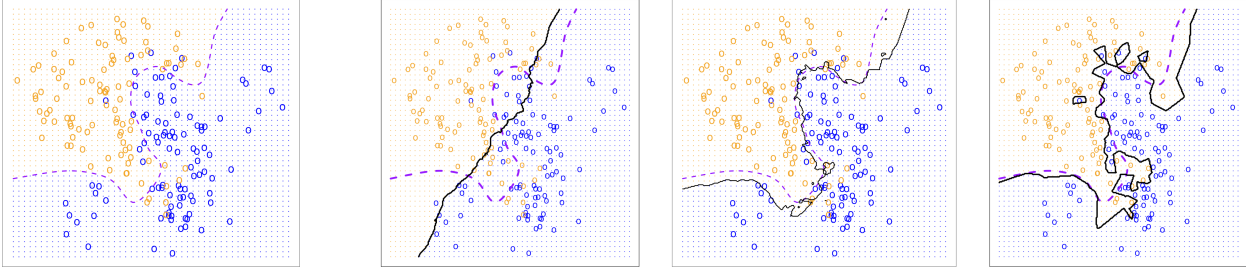
In a  $k$ -**nearest neighbours** classifier, for instance, the prediction for a new observation with predictors  $\mathbf{x}^* \in \text{Te}$  is obtained by finding the most frequent class label of the  $k$  nearest neighbours to  $\mathbf{x}^*$  in  $\text{Tr}$  on which the model  $\hat{C}_{k\text{NN}}(\mathbf{x})$  is built.

As the number of nearest neighbours under consideration increases, the complexity of the model  $\hat{C}_{k\text{NN}}(\mathbf{x})$  decreases, and *vice-versa*.

We would thus expect:

- a model with a large  $k$  to underfit the data;
- a model with a small  $k$  to overfit the data, and
- models in the “Goldilock zone” to strike a balance between **prediction accuracy** and **interpretability of the decision boundary** (see Figures 20.5 and 21.1).

As it happens, the optimal classifier  $Y = C(\vec{X})$  is, in fact, the Bayes optimal classifier.



**Figure 21.1:** Illustration of the accuracy-boundary interpretability trade-off for classifiers on an artificial dataset; Bayes optimal classifier  $C(x)$  (leftmost), underfit  $\hat{C}_{100NN}(x)$  model (2nd leftmost), Goldilocks  $\hat{C}_{10NN}(x)$  model (3rd leftmost), overfit  $\hat{C}_{1NN}(x)$  model (4th leftmost). Notice the interplay between prediction accuracy and complexity of the decision boundary (the dashed curve in the last three graphs shows the Bayes optimal boundary). [2, 3]

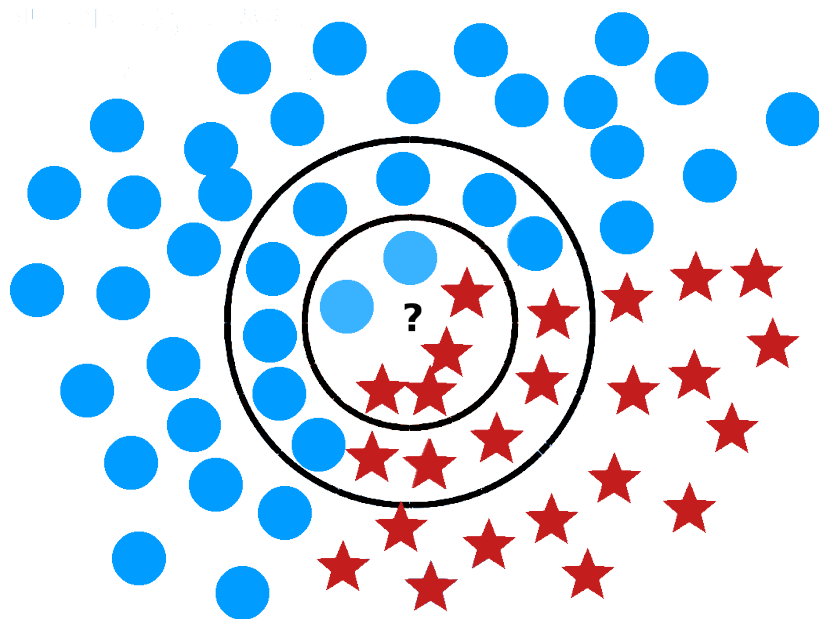
**Comparison Between  $kNN$  and OLS** We are going to try to get a better intuitive sense of the bias-variance trade-off by comparing ordinary least squares (OLS), a rigid yet simple model (as measured by the number of **effective parameters**), with  $k$ -nearest neighbours ( $kNN$ ), a very flexible yet more complex model (again, according to the number of effective parameters).

Given an input vector  $\mathbf{z} \in \mathbb{R}^p$ , the  $k$ -nearest neighbours ( $kNN$ ) model predicts the response  $Y$  as the average

$$\hat{Y} = \text{Avg}\{Y(x) \mid x \in N_k(\mathbf{z})\} = \frac{1}{k} \sum_{x \in N(\mathbf{z})} Y(x),$$

where  $Y(x)$  is the known response for predictor  $x \in \text{Tr}$  and  $N_k(\mathbf{z})$  is the set of the  $k$  training observations **nearest to  $\mathbf{z}$** . Another approach to neighbourhoods, which we will use at a later stage, is that they contain all training observations **within a certain (fixed) distance of  $\mathbf{z}$** .<sup>5</sup>

For classification problems,  $kNN$  models use the mode instead of the average. Of course, the prediction may depend on the value of  $k$ : in the **classification** image below, the 6NN prediction would be a red star, whereas the 19NN model prediction would be a blue disk.



**Figure 21.2:** Illustration of  $kNN$  classifiers.

5: The notion of proximity depends on the distance metric in use; the Euclidean case is the most common, but it does not have to be that one.



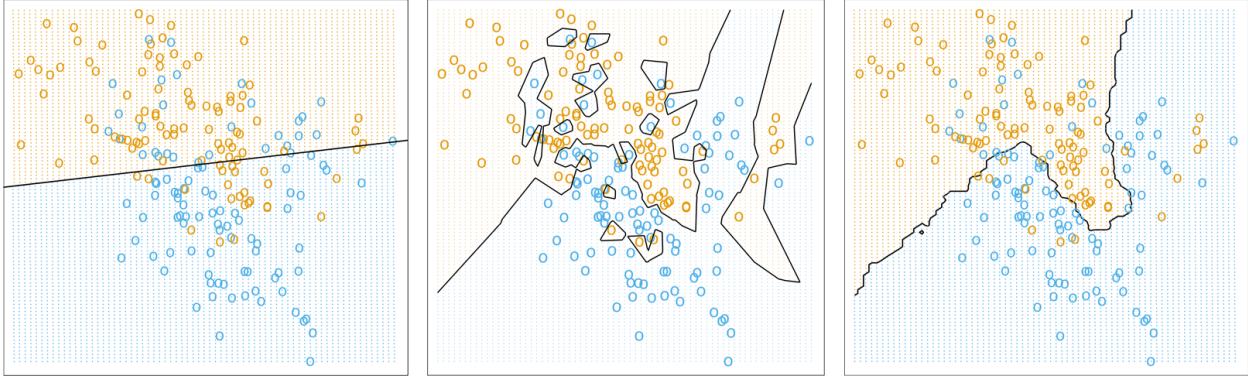


Figure 21.3: Classification based on OLS (left), 1NN (middle), and 15NN (right). [2, 3]

The following classification example (based on [2]) illustrates some of the bias-variance trade-off consequences. Consider a training dataset  $\text{Tr}$  consisting of 200 observations with features  $(x_1, x_2) \in \mathbb{R}^2$  and responses  $y \in \{\text{BLUE}_{(=0)}, \text{ORANGE}_{(=1)}\}$ . Let  $[\cdot] : \mathbb{R} \rightarrow \{\text{BLUE}, \text{ORANGE}\}$  denote the function

$$[w] = \begin{cases} \text{BLUE} & w \leq 0.5 \\ \text{ORANGE} & w > 0.5 \end{cases}$$

**Linear Fit** Fit an OLS model

$$\hat{y}(\mathbf{x}) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2$$

on  $\text{Tr}$ ; the class prediction is  $\hat{g}(\mathbf{x}) = [\hat{y}(\mathbf{x})]$ . The **decision boundary**

$$\partial_{\text{OLS}} = \{(x_1, x_2) \mid \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 = 0.5\}$$

is shown in Figure 21.3 (on the left); it is a straight line which can be described using only 2 **effective parameters**.<sup>6</sup>

There are several **misclassifications** on both sides of  $\partial_{\text{OLS}}$ ; even though errors seem to be unavoidable, the OLS model is likely to be **too rigid**.

**kNN Fit** If  $\hat{y}(\mathbf{x})$  represents the proportion of **ORANGE** points in  $N_k(\mathbf{x})$ , then the class prediction is  $\hat{g}(\mathbf{x}) = [\hat{y}(\mathbf{x})]$ . The decision boundaries  $\partial_{1\text{NN}}$  and  $\partial_{15\text{NN}}$  are displayed in Figure 21.3.

They are both irregular:  $\partial_{1\text{NN}}$  is overfit, whereas  $\partial_{15\text{NN}}$  is less likely to be.<sup>7</sup> The effective parameters are not as obviously defined for this model; one approach is to view  $k\text{NN}$  as a model that fits 1 parameter (a mean) to each **ideal** (non-overlapping) **neighbourhood** in the data, so that the number of effective parameters is roughly equal to the number of such neighbourhoods:

$$\frac{N}{k} \approx \begin{cases} 13 & \text{when } k = 15 \\ 200 & \text{when } k = 1 \end{cases}$$

The  $k\text{NN}$  models are thus fairly complex, in comparison with the OLS model. There are no misclassification for  $k = 1$ , and several in the case  $k = 15$ .<sup>8</sup> The 15NN model seems to strike a balance between various competing properties; it is likely nearer the “sweet spot” of the test error curve.<sup>9</sup>

6: Their number is a measure of a model's **complexity**.

7: Although neither is great for **interpretability**.

8: But not as many as with the OLS model.

9: Remember however that we have not evaluated the performance of the models on a testing set  $\text{Te}$ ; we have only described some of their behaviours on the training set  $\text{Tr}$ .

**Conclusions** The OLS model is **stable** as adding a few training observations is unlikely to alter the fit substantially, but also **biased** since the assumptions of a valid linear fit is questionable; the  $k$ NN models are **unstable** as adding a few training observations is quite likely to alter the fit substantially (especially for small values of  $k$ ), but it is also **unbiased** since no apparent assumptions are made about the data.

So which approach is best? That depends entirely on what the ultimate task is: description, prediction, etc. In predictive data science, machine learning, and artificial intelligence, the validity of modeling assumptions takes a backseat to a model's ability to **make good predictions on new (and unseen) observations**.

Naturally, we would expect that models whose assumptions are met are more likely to make good predictions than models for whom that is not the case, but it does not need to be the case. The theory of linear models is mature and extensive, and we could have discussed a number of their other features and extensions (see Chapter 8 for details).

Keep in mind, then, that machine learning methods are not meant to replace or supplant classical statistical analysis methods, but rather, to **complement them**. They simply provide different approaches to **gain insights from data**.

## 21.2 Simple Classification Methods

Qualitative variables take values in an **unordered** set  $\mathcal{C} = \{C_1, \dots, C_K\}$ . For instance,

- hair colour  $\in \{\text{black, red, blond, grey, other}\}$
- email message  $\in \{\text{ham, spam}\}$
- life expectancy  $\in \{\text{high, low}\}$

For a training set  $\text{Tr}$  with observations  $(\vec{X}, Y) \in \mathbb{R}^p \times \mathcal{C}$ , the **classification problem** is to build a classifier  $\hat{C} : \mathbb{R}^p \rightarrow \mathcal{C}$  to approximate the optimal Bayes classifier  $C : \mathbb{R}^p \rightarrow \mathcal{C}$  (as discussed in the previous section).

In many instances, we might be more interested in the probabilities

$$\pi_k(\mathbf{x}) = P\{\hat{C}(\mathbf{x}) = C_k\}, \quad k = 1, \dots, K$$

than in the classification predictions themselves. Typically, the classifier  $\hat{C}$  is **built** on a training set

$$\text{Tr} = \{(\mathbf{x}_j, y_j)\}_{j=1}^N$$

and **evaluated** on a testing set

$$\text{Te} = \{(\mathbf{x}_i, y_i)\}_{i=N+1}^M.$$

**Example** Let us revisit the [gapminder.csv](#) dataset, again focusing on observations from 2011, with the difference that life expectancy is now recorded as “high” (1) if it falls above 72.45 (the median in 2011), and as “low” (0) otherwise.

## Setting up the Gapminder dataset

```
library(dplyr)
gapminder.ML = read.csv("gapminder.csv",
                        stringsAsFactors=TRUE)
gapminder.ML <- gapminder.ML[complete.cases(gapminder.ML),]
gapminder.ML <- gapminder.ML[,c("country", "year", "region",
                                "continent", "population", "infant_mortality",
                                "fertility", "gdp", "life_expectancy")]

gapminder.2011 <- gapminder.ML |> filter(year==2011) |>
  mutate(LE=as.factor(ifelse(life_expectancy <
                              median(life_expectancy), "low", "high")))
```

The structure and summary are provided below:

```
summary(gapminder.2011[,c("infant_mortality",
                           "fertility", "LE")])
```

infant_mortality	fertility	LE
Min. : 1.800	Min. :1.260	high:83
1st Qu.: 7.275	1st Qu.:1.792	low :83
Median : 16.900	Median :2.420	
Mean : 27.333	Mean :2.931	
3rd Qu.: 41.125	3rd Qu.:3.908	
Max. :106.800	Max. :7.580	

Let us assume that we are interested in modeling the response LE ( $Y$ ) as a linear response of the predictors  $X_1$  (infant mortality) and  $X_2$  (fertility), using ordinary linear regression (OLS).<sup>10</sup>

10: See Chapter 8 for a detailed discussion of such models.

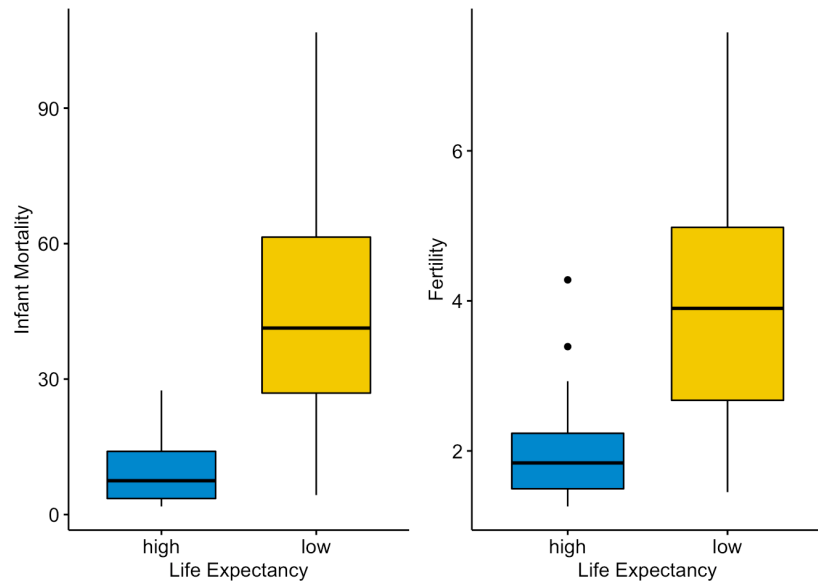
```
p1 <- ggpubr::ggboxplot(gapminder.2011, x = "LE",
                        y = "infant_mortality", fill = "LE", palette = "jco",
                        xlab="Life Expectancy", ylab="Infant Mortality") +
  ggpubr::rremove("legend")

p2 <- ggpubr::ggboxplot(gapminder.2011, x = "LE",
                        y = "fertility", fill = "LE", palette = "jco",
                        xlab="Life Expectancy", ylab="Fertility") +
  ggpubr::rremove("legend")

grid::pushViewport(grid::viewport(
  layout = grid::grid.layout(nrow = 1, ncol = 2)))

# helper function to define a region on the layout
define_region <- function(row, col){
  grid::viewport(layout.pos.row = row, layout.pos.col = col)
}

print(p1, vp = define_region(row = 1, col = 1))
print(p2, vp = define_region(row = 1, col = 2))
```



We run an OLS regression of  $Y$  on  $\vec{X}$  over  $Tr$  and obtain the model

$$\hat{Y} = \hat{\beta}_0 + \hat{\beta}_1 \cdot \text{infant mortality} + \hat{\beta}_2 \cdot \text{fertility},$$

from which we would classify an observation's life expectancy as

$$\hat{C}(\vec{X}) = \begin{cases} \text{high} & \text{if } \hat{Y} > 0.5 \\ \text{low} & \text{else} \end{cases}$$

```
gapminder.2011 <- gapminder.2011 |>
  mutate(LE.resp=ifelse(LE=="high",1,0))
model.class <- lm(LE.resp ~ infant_mortality + fertility,
  data=gapminder.2011)
beta_0=as.numeric(model.class[[1]][1])
beta_1=as.numeric(model.class[[1]][2])
beta_2=as.numeric(model.class[[1]][3])
model.class[[1]]
```

```
(Intercept) infant_mortality    fertility
1.00102979   -0.01188533   -0.06010533
```

Thus,

$$\hat{Y} = 1.001 - 0.012 \cdot \text{infant mortality} - 0.060 \cdot \text{fertility}.$$

11: If the boundary splits the observations at  $\hat{Y} = \gamma \in [0, 1]$ , then it solves

$$\gamma = \beta_0 + \beta_1 X_1 + \beta_2 X_2, \quad \beta_2 \neq 0,$$

so that

$$X_2 = \left( \frac{\gamma - \beta_0}{\beta_2} \right) - \frac{\beta_1}{\beta_2} X_1.$$

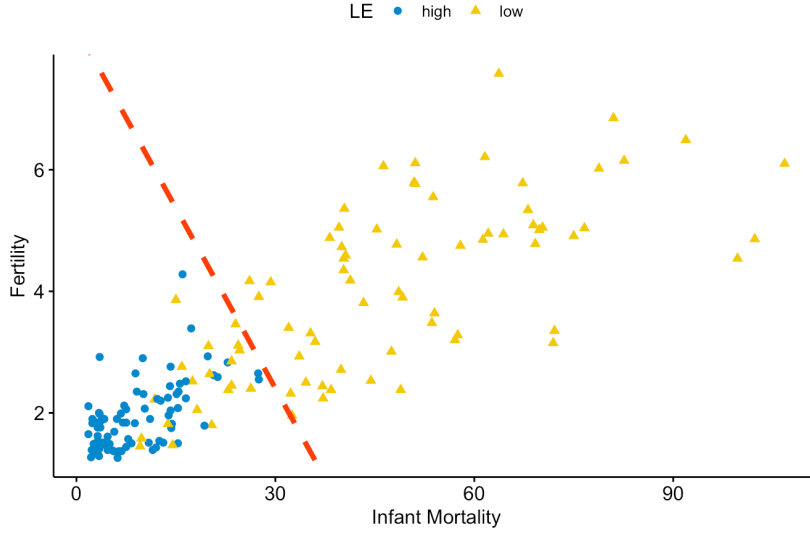
We plot the decision boundary on the scatterplot of the domain:<sup>11</sup>

```
slope = -beta_1/beta_2
intercept = 0.5*(1-2*beta_0)/beta_2

ggpubr::ggscatter(gapminder.2011, x="infant_mortality",
  y="fertility", shape="LE", color="LE", palette="jco",
  size = 2, xlab="Infant Mortality", ylab = "Fertility",
```

```
title = "Gapminder 2011 Data") +
ggplot2::geom_abline(intercept = intercept,
  slope = slope, color="red", linetype="dashed", size=1.5)
```

Gapminder 2011 Data



The OLS approach is likely to do a decent job here since the data is roughly **linearly separable** over the predictors. This will not usually be the case, however.

In the example above, the optimal regression function is

$$f(\mathbf{x}) = E[Y \mid \vec{X} = \mathbf{x}] = P(Y = 1 \mid \vec{X} = \mathbf{x}) = p_1(\mathbf{x})$$

because  $Y$  is a binary variable; this might lead us to believe that  $f(\mathbf{x})$  could also be used to directly classify and determine the class probabilities for the data, in which case there would be no need for a **separate classification apparatus**.<sup>12</sup>

A problem arises if we study the residual situation further. If we model  $Y = \{0, 1\}$  with an OLS regression, we have

$$Y_i = \mathbf{x}_i^\top \boldsymbol{\beta} + \varepsilon_i.$$

Thus

$$\varepsilon_i = Y_i - \mathbf{x}_i^\top \boldsymbol{\beta} = \begin{cases} 1 - \mathbf{x}_i^\top \boldsymbol{\beta} & \text{if } Y_i = 1 \\ -\mathbf{x}_i^\top \boldsymbol{\beta} & \text{if } Y_i = 0 \end{cases}$$

But OLS assumes that  $\varepsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2 I)$ , which is clearly not the case here, as  $\varepsilon_i$  can only take two values. OLS is thus not an appropriate way to model the response.

Furthermore,

$$\text{Var}(Y_i) = p_1(\mathbf{x}_i)(1 - p_1(\mathbf{x}_i)),$$

since  $Y_i$  is a binomial random variable, and

$$\text{Var}(\varepsilon_i) = \text{Var}(Y_i - p_1(\mathbf{x}_i)) = \text{Var}(Y_i) = p_1(\mathbf{x}_i)(1 - p_1(\mathbf{x}_i)),$$

which is not constant as it depends on  $\mathbf{x}_i$ .

12: There is one major drawback with this approach: if linear regression is used to model the data (which is to say, if we assume that  $f(\mathbf{x}) \approx \mathbf{x}^\top \boldsymbol{\beta}$ ), we need to insure that  $\hat{f}_{\text{OLS}}(\mathbf{x}) \in [0, 1]$  for all  $\mathbf{x} \in \mathcal{T}$ . This, in general, cannot be guaranteed.

13: There is another way in which OLS could fail, but it has nothing to do with the OLS assumptions *per se*. When the set of qualitative responses contains more than 2 level (such as  $\mathcal{C} = \{\text{low}, \text{medium}, \text{high}\}$ , for instance), the response is usually encoded using numerals to facilitate the implementation of the analysis:

$$Y = \begin{cases} 0 & \text{if low} \\ 1 & \text{if medium} \\ 2 & \text{if high} \end{cases}$$

This encoding suggests an **ordering** and a **scale** between the levels (for instance, the difference between “high” and “medium” is equal to the difference between “medium” and “low”, and half again as large as the difference between “high” and “low”). OLS is not appropriate in this context.

14: The probit transformation uses  $g_P(y^*) = \Phi(y^*)$ , where  $\Phi$  is the cumulative distribution function of  $\mathcal{N}(0, 1)$ .

The OLS assumptions are thus violated at every turn<sup>13</sup> – OLS is simply not a good fit/modeling approach to estimate

$$p_k(\mathbf{x}) = P(Y = C_k \mid \vec{X} = \mathbf{x}).$$

We start by introducing two simple classification methods (see [3] for more details).

### 21.2.1 Logistic Regression

The problems presented above point to OLS not being an ideal method for classification, but the linear regression still provided a good separator in the Gapminder example. This suggests that we should not automatically reject the possibility of first transforming the data and then seeing if OLS might not provide an appropriate modeling strategy on the transformed data.

**Formulations** In **logistic regression**, we are seeking an invertible transformation  $g : \mathbb{R} \rightarrow [0, 1]$ , with  $g(y^*) = y$  and  $g^{-1}(y) = y^*$ . The variable  $y$  must behave like a probability; in the 2-class setting, we use  $g_L(y^*)$  to approximate the probability

$$p_1(\mathbf{x}) = P(Y = 1 \mid \vec{X} = \mathbf{x}).$$

The idea is to run OLS on a transformed training set

$$\text{Tr}^* = \{(\mathbf{x}_i, y_i^*)\}_{i=1}^N,$$

and to transform the results back using  $y_i = g(y_i^*)$ .

There are many such functions: the **probit** model,<sup>14</sup> which we will not discuss, and the **logit** model regression model are two common approaches.

**Logit Model** The logit model uses the transformation

$$y = g_L(y^*) = \frac{e^{y^*}}{1 + e^{y^*}}.$$

It is such that

$$g_L^{-1}(0) = -\infty, \quad g_L^{-1}(1) = \infty, \quad g_L^{-1}(0.5) = 0, \quad \text{etc.}$$

We solve for  $y^*$  in order to get a transformed response  $y^* \in \mathbb{R}$  (instead of one restricted to  $[0, 1]$ ):

$$p_1(\mathbf{x}) = \frac{e^{y^*}}{1 + e^{y^*}} \iff y^* = g_L^{-1}(y) = \ln \left( \frac{p_1(\mathbf{x})}{1 - p_1(\mathbf{x})} \right).$$

It is the **log-odds** transformed observations that we attempt to fit with an OLS model:

$$\hat{Y}^* = \ln \left( \frac{p_1(\mathbf{x})}{1 - p_1(\mathbf{x})} \right) = \beta_0 + \beta_1 X_1 + \dots + \beta_p X_p = \mathbf{x}^\top \boldsymbol{\beta}.$$

In order to make a prediction for  $p_1(\mathbf{x})$ , we estimate  $y^*$  and use the logit transformation to recover  $y$ . For instance, if  $\mathbf{x}^\top \hat{\boldsymbol{\beta}} = 0.68$ , then

$$\hat{y}^* = \ln \left( \frac{\hat{p}_1(\mathbf{x})}{1 - \hat{p}_1(\mathbf{x})} \right) = 0.68$$

and

$$\hat{p}_1(\mathbf{x}) = \frac{e^{y^*}}{1 + e^{y^*}} = \frac{e^{0.68}}{1 + e^{0.68}} = 0.663.$$

Depending on the **decision rule threshold**  $\gamma$ , we may thus predict that  $\hat{C}(\mathbf{x}) = C_1$  if  $p_1(\mathbf{x}) > \gamma$  or  $\hat{C}(\mathbf{x}) = C_2$ , otherwise.

The technical challenge is in obtaining the coefficients  $\hat{\boldsymbol{\beta}}$ ; they are found by **maximizing the likelihood** (see [41])

$$\begin{aligned} L(\boldsymbol{\beta}) &= \prod_{y_i=1} p_1(\mathbf{x}_i) \prod_{y_i=0} (1 - p_1(\mathbf{x}_i)) \\ &= \prod_{y_i=1} \frac{\exp(\mathbf{x}_i^\top \boldsymbol{\beta})}{1 + \exp(\mathbf{x}_i^\top \boldsymbol{\beta})} \prod_{y_i=0} \frac{1}{1 + \exp(\mathbf{x}_i^\top \boldsymbol{\beta})}, \end{aligned}$$

or, more simply:

$$\begin{aligned} \hat{\boldsymbol{\beta}} &= \arg \max_{\boldsymbol{\beta}} \{L(\boldsymbol{\beta})\} = \arg \max_{\boldsymbol{\beta}} \{\ln L(\boldsymbol{\beta})\} \\ &= \arg \max_{\boldsymbol{\beta}} \left\{ \sum_{y_i=1} \ln p_1(\mathbf{x}_i) + \sum_{y_i=0} \ln(1 - p_1(\mathbf{x}_i)) \right\} \\ &= \dots \text{(terms in } \boldsymbol{\beta} \text{ and the observations } \mathbf{x}_i \text{)}. \end{aligned}$$

The optimizer  $\hat{\boldsymbol{\beta}}$  is then found using numerical methods; in R, the function `glm()` computes the maximum likelihood estimate directly.

**Example** Using the Gapminder data from this section's start, we obtain the following model:

```
model.LR <- glm(LE.resp ~ infant_mortality + fertility,
               data=gapminder.2011, family=binomial)
model.LR
```

Coefficients:

(Intercept)	infant_mortality	fertility
4.58733	-0.22499	-0.06495

Degrees of Freedom: 165 Total (i.e. Null); 163 Residual

Null Deviance: 230.1

Residual Deviance: 78.17 AIC: 84.17

Thus

$$\hat{y}^* = \ln \left( \frac{P(Y = \text{high} \mid \vec{X})}{1 - P(Y = \text{high} \mid \vec{X})} \right) = 4.59 - 0.22X_1 - 0.06X_2.$$

For a decision rule threshold of  $\gamma = 0.5$ , the decision boundary is shown below (compare with the linear regression boundary on page 969).

```

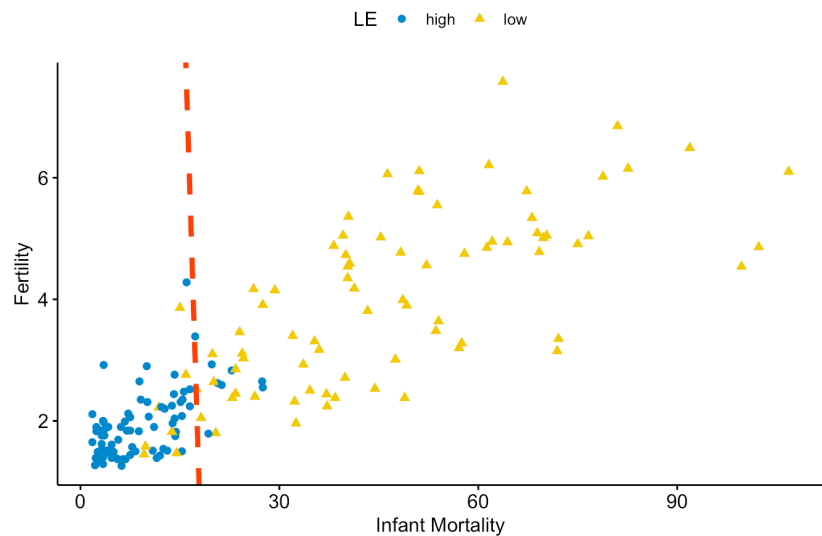
beta_0 = as.numeric(model.LR[[1]][1])
beta_1 = as.numeric(model.LR[[1]][2])
beta_2 = as.numeric(model.LR[[1]][3])

slope = -beta_1/beta_2
intercept = 0.5*(1-2*beta_0)/beta_2

ggpubr::ggscatter(gapminder.2011, x="infant_mortality",
  y="fertility", shape="LE", color="LE", palette="jco",
  size = 2, xlab="Infant Mortality", ylab = "Fertility",
  title = "Gapminder 2011 Data") +
  ggplot2::geom_abline(intercept = intercept,
    slope = slope, color="red", linetype="dashed", size=1.5)

```

Gapminder 2011 Data



What is the estimated probability that the life expectancy is high in a country whose infant mortality is 15 and whose fertility is 4? By construction,

$$\begin{aligned}
 p_1(Y = \text{high} \mid X_1 = 15, X_2 = 4) &\approx g_L([1, 15, 24]^T \hat{\beta}) \\
 &= \frac{\exp(4.59 - 0.22(15) - 0.06(4))}{1 + \exp(4.59 - 0.22(15) - 0.06(4))} \\
 &= \frac{\exp(0.9526322)}{1 + \exp(0.9526322)} = 0.72.
 \end{aligned}$$

How does all of this square up with the statistical learning framework of Sections 19 and 20: no testing set has made an appearance, no misclassification or mean squared error rate has been calculated.

Next, we **randomly** select 116 observations, say, and train a logistic regression model on this training set  $\text{Tr}$  to obtain:

```

set.seed(0)
ind.train = sample(nrow(gapminder.2011),
  round(0.7*nrow(gapminder.2011)), replace=FALSE)

```



```
gapminder.2011.tr = gapminder.2011[ind.train,]
gapminder.2011.te = gapminder.2011[-ind.train,]

model.LR.tr <- glm(LE.resp ~ infant_mortality + fertility,
                  family=binomial, data=gapminder.2011.tr)
model.LR.tr
```

Coefficients:

(Intercept)	infant_mortality	fertility
6.1194	-0.2050	-0.6653

Degrees of Freedom: 115 Total (i.e. Null); 113 Residual

Null Deviance: 159.1

Residual Deviance: 50.83 AIC: 56.83

Thus,

$$\hat{y}^* = 6.12 - 0.21x_1 - 0.67x_2.$$

Now, compute

$$\hat{p}_i = P(Y_i = \text{high} \mid X_1 = x_{1,i}, X_2 = x_{2,i}) = \frac{\exp(\hat{y}_i^*)}{1 + \exp(\hat{y}_i^*)}$$

on the observations in the testing set Te (see below).<sup>15</sup>

15: The observations in the original dataset *not* in Tr.

```
beta_0 = as.numeric(model.LR[[1]][1])
beta_1 = as.numeric(model.LR[[1]][2])
beta_2 = as.numeric(model.LR[[1]][3])

gapminder.2011.te$y.star = beta_0 +
  beta_1*gapminder.2011.te$infant_mortality +
  beta_2*gapminder.2011.te$fertility
gapminder.2011.te$p.1 = exp(gapminder.2011.te$y.star)/
  (1+exp(gapminder.2011.te$y.star))
gapminder.2011.te$MSE = (gapminder.2011.te$p.1 -
  gapminder.2011.te$LE.resp)^2

summary(gapminder.2011.te$LE.resp)
```

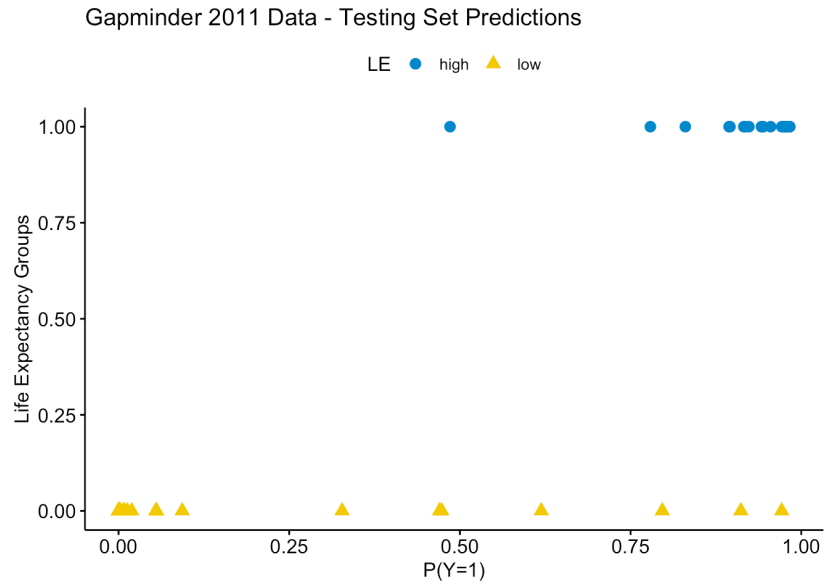
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.00	0.00	0.00	0.36	1.00	1.00

We obtain

$$\text{MSE}_{\text{Te}} = \frac{1}{50} \sum_{i=1}^{50} (\hat{p}_i - \mathcal{I}[Y_i = \text{high}])^2 = 0.075.$$

Is that a good test error? It is difficult to answer without more context. Perhaps a more intuitive way to view the situation is to make actual predictions and to explore their quality.

```
ggpubr::ggscatter(gapminder.2011.te, x="p.1", y="LE.resp",
  shape="LE", color="LE", palette="jco", size = 3,
  xlab="P(Y=1)", ylab = "Life Expectancy Groups",
  title = "Gapminder 2011 Data - Testing Set Predictions")
```



For  $\alpha \in [0, 1]$ , we further define

$$\text{pred}_i(\alpha) = \begin{cases} \text{high} & \text{if } \hat{p}_i > \alpha \\ \text{low} & \text{else} \end{cases}$$

In the specific version of  $\text{Te}$  used in this example, 36% of the nations had a high life expectancy.

```
gapminder.2011.te$pred81 = ifelse(gapminder.2011.te$p.1 > 0.81,
  1, 0)
table(gapminder.2011.te$LE.resp, gapminder.2011.te$pred81)
```

If we set  $\alpha = 0.81$ , then the model predicts that 36% of the test nations will have a high life expectancy, and the **confusion matrix** on  $\text{Te}$  is shown below:

$\alpha = 0.81$		prediction	
		0	1
actual	0	30	2
	1	2	16

16: In a sense, this could prove to be the only rational choice in the absence of information.

But why pick  $\alpha = 0.81$  instead of  $\alpha = 0.5$ , say?<sup>16</sup> In the latter case, 42% of nations are predicted to have high life expectancy, and the confusion matrix on  $\text{Te}$  is as in the next page.

```
gapminder.2011.te$pred50 = round(gapminder.2011.te$p.1, 0)
table(gapminder.2011.te$LE.resp, gapminder.2011.te$pred50)
```

$\alpha = 0.5$		prediction	
		0	1
actual	0	28	4
	1	1	17

We will revisit this question at the end of this section (*ROC Curve*).

### 21.2.2 Discriminant Analysis

In logistic regression, we model  $P(Y = C_k | \mathbf{x})$  directly *via* the logistic function

$$p_1(\mathbf{x}) = \frac{\exp(\mathbf{x}^\top \hat{\boldsymbol{\beta}})}{1 + \exp(\mathbf{x}^\top \hat{\boldsymbol{\beta}})}.$$

We have discussed some of the properties of the process in the previous section, but it should be noted that logistic regression is sometimes contra-indicated:

- when the classes are **well-separated**, the coefficient estimates may be **unstable** (adding as little as one additional point to Tr could change the coefficients substantially);
- when Tr is small and the distribution of the predictors is roughly **Gaussian** in each of the classes  $Y = C_k$ , the coefficient estimates may be unstable too;
- when there are more than 2 response levels, it is not always obvious how to select an extension of logistic regression.

In **discriminant analysis** (DA), we instead model

$$P(\mathbf{x} | Y = C_k),$$

the distribution of the predictors  $\vec{X}$  conditional on the level of  $Y$ , and use Bayes' Theorem to obtain

$$P(Y = C_k | \mathbf{x}),$$

the probability of observing the response conditional on the predictors.

Let  $\mathcal{C} = \{C_1, \dots, C_K\}$  be the  $K$  response levels,  $K \geq 2$ , and denote by  $\pi_k$  the probability that a random observation lies in  $C_k$ , for  $k \in \{1, \dots, K\}$ ;  $\pi_k$  is the **prior**

$$\pi_k = P(Y = C_k) = \frac{|C_k|}{N}.$$

Let  $f_k(\mathbf{x}) = P(\mathbf{x} | Y = C_k)$  be the **conditional density function** of the distribution of  $\vec{X}$  in  $C_k$ ; we would expect  $f_k(\mathbf{x})$  to be large if there is a high probability that an observation in  $C_k$  has a corresponding predictor  $\vec{X} \approx \mathbf{x}$ , and small otherwise.

According to Bayes' Theorem,

$$\begin{aligned}
 p_k(\mathbf{x}) &= P(Y = C_k | \mathbf{x}) \\
 &= \frac{P(\mathbf{x} | Y = C_k) \cdot P(Y = C_k)}{P(\mathbf{x})} \\
 &= \frac{P(\mathbf{x} | Y = C_k) \cdot P(Y = C_k)}{P(\mathbf{x} | Y = C_1) \cdot P(Y = C_1) + \cdots + P(\mathbf{x} | Y = C_K) \cdot P(Y = C_K)} \\
 &= \frac{\pi_k f_k(\mathbf{x})}{\pi_1 f_1(\mathbf{x}) + \cdots + \pi_K f_K(\mathbf{x})}.
 \end{aligned}$$

Given an observation  $\mathbf{x} \in \mathcal{T}_e$ , the DA classifier is

$$\hat{C}_{DA}(\mathbf{x}) = C_{\arg \max_j \{p_j(\mathbf{x})\}}.$$

In order to say more about discriminant analysis, we need to make additional assumptions on the nature of the underlying distributions.

**Linear Discriminant Analysis** If there is only one predictor ( $p = 1$ ), we make the **Gaussian assumption**,

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_k}{\sigma_k} \right)^2 \right],$$

where  $\mu_k$  and  $\sigma_k$  are the mean and the standard deviation, respectively, of the predictor for all observations in class  $C_k$ .<sup>17</sup>

If we further assume that  $\sigma_k \equiv \sigma$  for all  $k$ , then

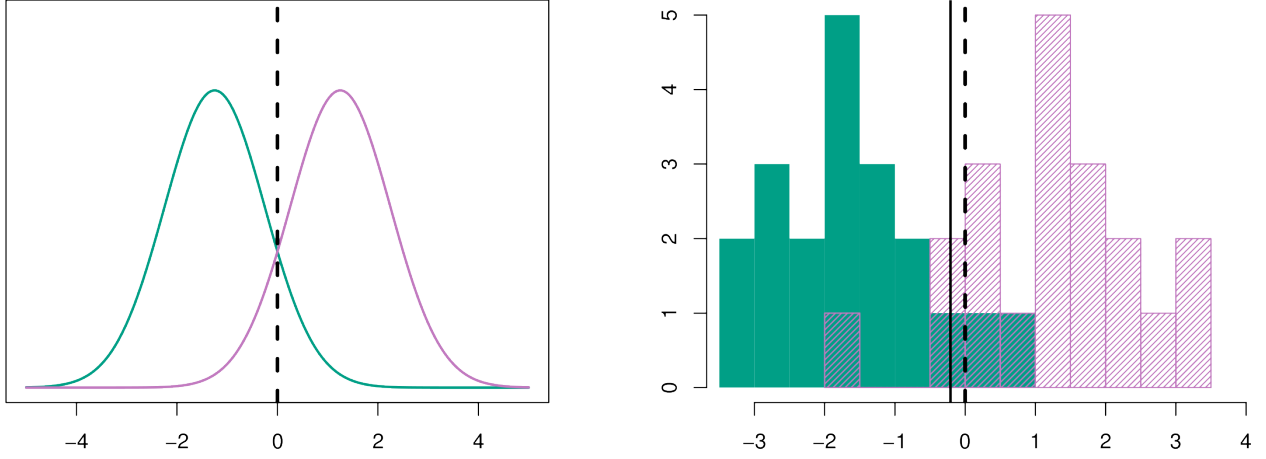
$$\begin{aligned}
 p_k(x) &= \frac{\pi_k \frac{1}{\sqrt{2\pi}\sigma} \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_k}{\sigma} \right)^2 \right]}{\pi_1 \frac{1}{\sqrt{2\pi}\sigma} \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_1}{\sigma} \right)^2 \right] + \cdots + \pi_K \frac{1}{\sqrt{2\pi}\sigma} \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_K}{\sigma} \right)^2 \right]} \\
 &= \frac{\pi_k \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_k}{\sigma} \right)^2 \right]}{\pi_1 \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_1}{\sigma} \right)^2 \right] + \cdots + \pi_K \exp \left[ -\frac{1}{2} \left( \frac{x - \mu_K}{\sigma} \right)^2 \right]} \\
 &= \frac{\pi_k \exp \left[ \frac{\mu_k}{\sigma^2} \left( x - \frac{\mu_k}{2} \right) \right] \exp \left( -\frac{x^2}{2\sigma^2} \right)}{\left\{ \pi_1 \exp \left[ \frac{\mu_1}{\sigma^2} \left( x - \frac{\mu_1}{2} \right) \right] + \cdots + \pi_K \exp \left[ \frac{\mu_K}{\sigma^2} \left( x - \frac{\mu_K}{2} \right) \right] \right\} \exp \left( -\frac{x^2}{2\sigma^2} \right)} \\
 &= \pi_k \exp \left[ \frac{\mu_k}{\sigma^2} \left( x - \frac{\mu_k}{2} \right) \right] \cdot A(x).
 \end{aligned}$$

We do not need to compute the actual probabilities  $p_k(x)$  directly if we are only interested in classification; in that case, the **discriminant score** for each class may be more useful:

$$\delta_k(x) = \ln p_k(x) = \ln \pi_k + x \frac{\mu_k}{\sigma^2} - \frac{\mu_k}{2\sigma^2} + \ln A(x).$$

As  $\ln A(x)$  is the same for all  $k$ , we can drop it from the score as it does not contribute to relative differences in class scores; given an observation

17: Any other predictor distribution could be used if it is more appropriate for Tr, and we could assume that the standard deviations or the means (or both) are identical across classes.



**Figure 21.4:** Midpoint of two theoretical normal distributions (dashed line); midpoint of two empirical normal distributions (solid line). Observations to the left of the decision boundary are classified as green, those to the right as purple. [3]

$x \in \mathcal{T}_e$ , the **linear discriminant analysis** (LDA) classifier with  $p = 1$  is

$$\hat{C}_{\text{LDA}}(\mathbf{x}) = C_{\arg \max_j \{\delta_{j(\mathbf{x})}\}}.$$

Under other assumptions on the density function, the discriminant score formulation may change.

The “linear” in LDA comes from the linearity of the discriminant scores  $\delta_k$ .<sup>18</sup>

If  $K = 2$  and  $\pi_1 = \pi_2 = 0.5$ , the midpoint  $x^* = \frac{1}{2}(\mu_1 + \mu_2)$  of the predictor means in  $C_1$  and  $C_2$  plays a crucial role. Indeed, the discriminant scores  $\delta_1(x)$  and  $\delta_2(x)$  meet when

$$x^* \frac{\mu_1}{\sigma^2} - \frac{\mu_1^2}{2\sigma^2} = x^* \frac{\mu_2}{\sigma^2} - \frac{\mu_2^2}{2\sigma^2} \implies x^* = \frac{\mu_1 + \mu_2}{2},$$

as long as  $\mu_1 \neq \mu_2$ . If  $\mu_1 < \mu_2$ , say, then the decision rule simplifies to

$$\hat{C}(x) = \begin{cases} C_1 & \text{if } x \leq x^* \\ C_2 & \text{if } x > x^* \end{cases}$$

The principle is illustrated in Figure 21.4.

In practice, we estimate  $\pi_k$ ,  $\mu_k$  and  $\sigma$  from  $\mathcal{T}_r$ :

$$\begin{aligned} \hat{\pi}_k &= \frac{N_k}{N}, \quad \hat{\mu}_k = \frac{1}{N_k} \sum_{y_i \in C_k} x_i \\ \hat{\sigma}^2 &= \sum_{k=1}^K \frac{N_k - 1}{N - K} \left( \frac{1}{N_k - 1} \sum_{y_i \in C_k} (x_i - \hat{\mu}_k)^2 \right). \end{aligned}$$

If there are  $p > 1$  predictors, we can still make the Gaussian assumption, but adapted to  $\mathbb{R}^p$ :

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{p/2} |\boldsymbol{\Sigma}_k|^{1/2}} \exp \left[ -\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k) \right],$$

18: After the  $\ln A(x)$  term has been dropped.

where  $\mu_k = (\overline{X_1}, \dots, \overline{X_p})$  and  $\Sigma_k(j, i) = \text{Cov}(X_i, X_j)$  for all  $\vec{X}$  with  $Y = C_k$ .

If we further assume that  $\sigma_k \equiv \Sigma$  for all  $k$ , then we can show that the discriminant score is, again, linear (in  $\mathbf{x}$ ):

$$\delta_{k;\text{LDA}}(\mathbf{x}) = \mathbf{x}^\top \Sigma^{-1} \mu_k - \frac{1}{2} \mu^\top \Sigma^{-1} \mu_k + \ln \pi_k = c_{k,0} + \mathbf{c}_k^\top \mathbf{x}.$$

We can estimate  $\mu_k$  and  $\Sigma$  from the data, from which we can recover the estimates

$$P(Y = C_k | \mathbf{x}) \approx \hat{p}_k(\mathbf{x}) = \frac{\exp(\hat{\delta}_{k;\text{LDA}}(\mathbf{x}))}{\sum_{j=1}^K \exp(\hat{\delta}_{j;\text{LDA}}(\mathbf{x}))}.$$

The decision rule is as before: given an observation  $\mathbf{x} \in \text{Te}$ , the LDA classifier with  $p > 1$  is

$$\hat{C}_{\text{LDA}} = C_{\arg \max_j \{\hat{\delta}_{j;\text{LDA}}(\mathbf{x})\}}.$$

**Quadratic Discriminant Analysis** The assumption that the conditional probability functions be Gaussians with the same covariance in each training class may be a stretch in some situations.

If  $\Sigma_i \neq \Sigma_j$  for at least one pair of classes  $(i, j)$ , then a similar process gives rise to **quadratic discriminant analysis** (QDA), which reduces to discriminant scores

$$\begin{aligned} \delta_{k;\text{QDA}}(\mathbf{x}) &= -\frac{1}{2}(\mathbf{x} - \mu)^\top \Sigma_k^{-1}(\mathbf{x} - \mu) + \ln \pi_k \\ &= -\frac{1}{2} \mathbf{x}^\top \Sigma_k^{-1} \mathbf{x} + \mathbf{x}^\top \Sigma_k^{-1} \mu_k - \frac{1}{2} \mu^\top \Sigma_k^{-1} \mu_k + \ln \pi_k. \end{aligned}$$

To learn the LDA model, we must estimate  $Kp + \frac{p(p+1)}{2}$  parameters from  $\text{Tr}$ ;<sup>19</sup> to learn QDA,  $K \left( p + \frac{p(p+1)}{2} \right)$ .<sup>20</sup> QDA is thus more complex (and more flexible) than LDA.

The latter is recommended if  $\text{Tr}$  is small; the former if  $\text{Tr}$  is large, but LDA will yield high bias if the  $\Sigma_k \equiv \Sigma$  assumption is invalid. Note that LDA gives rise to nearly **linear separating hypersurfaces** and QDA to **quadratic** ones.

**Gaussian Naïve Bayes Classification** If we assume further that each  $\Sigma_k$  is **diagonal** (that is, if we assume that the features are independent from each other in each class), we obtain the **Gaussian naïve Bayes classifier** (GNBC), with discriminant scores given by

$$\delta_{k;\text{GNBC}}(\mathbf{x}) = -\frac{1}{2} \sum_{j=1}^p \frac{(x_j - \mu_{k,j})^2}{\sigma_{k,j}^2} + \ln \pi_k.$$

The classification process continues as before. Note that the assumption of independence is usually not met, hence the “naïve” part in the name.

19:  $p$  parameters for each  $\hat{\mu}_k$  and  $1 + \dots + p$  parameters for  $\hat{\Sigma}$ .

20:  $p$  parameters for each  $\hat{\mu}_k$  and  $1 + \dots + p$  parameters for each  $\hat{\Sigma}_k$ .

In spite of this, GNBC can prove very useful when  $p$  is too large, where both LDA and QDA break down.

Note that this approach can also be used for mixed feature vectors, by using combinations of probability mass functions and probability density functions in  $f_{k,j}(x_j)$ , as required. We will re-visit NBC in Section 21.4 (*Naïve Bayes Classifiers*).

**Logistic Regression (Reprise)** We can also recast the 2-class LDA model as

$$\ln \left( \frac{p_0(\mathbf{x})}{1 - p_0(\mathbf{x})} \right) = \ln(p_0(\mathbf{x})) - \ln(p_1(\mathbf{x})) = \delta_0(\mathbf{x}) - \delta_1(\mathbf{x}) = a_0 + \mathbf{a}^\top \mathbf{x},$$

which has the same form as logistic regression.

It is not the same model, however:

- in **logistic regression**, the parameters are estimated using the maximum likelihood  $P(Y | \mathbf{x})$ ;
- in **LDA**, the parameters are estimated using the full likelihood  $P(\mathbf{x} | Y)P(\mathbf{x}) = P(\mathbf{x}, Y)$ .

**Example** We finish this section by giving an example of LDA and QDA on the 2011 Gapminder data (we will use the same training set  $\text{Tr}$  with  $N = 116$  observations and testing set  $\text{Te}$  with  $M = 50$  observations).

Given an observation  $\mathbf{x} \in \text{Te}$ , we use a decision rule based on the probabilities  $\hat{p}_0(\mathbf{x})$ ,  $\hat{p}_1(\mathbf{x})$  and a decision threshold  $\alpha \in (0, 1)$ . On the training set  $\text{Tr}$ , we find:

```
library(dplyr)
tmp = gapminder.2011.tr |> group_by(LE.resp) |>
  summarise(N.k=n(), mean.im=mean(infant_mortality),
    mean.f=mean(fertility))

N.0 = tmp[[2]][1]
N.1 = tmp[[2]][2]
N = N.0 + N.1

mu.0 = t(matrix(cbind(tmp[[3]][1],tmp[[4]][1])))
mu.1 = t(matrix(cbind(tmp[[3]][2],tmp[[4]][2])))
mu = (N.0*mu.0+N.1*mu.1)/N

tmp <- gapminder.2011.tr |>
  split(gapminder.2011.tr$LE.resp) |>
  purrr::map(select, c("infant_mortality","fertility")) |>
  purrr::map(cov)

Sigma.0 <- tmp[[1]]
Sigma.1 <- tmp[[2]]
Sigma <- cov(gapminder.2011.tr[,c("infant_mortality",
  "fertility")])
```

which yields

$$\begin{aligned} N_0 &= 51, N_1 = 65, \hat{\pi}_0 = 51/116, \hat{\pi}_1 = 65/116, \\ \hat{\mu}_0 &= (45.40, 4.08)^\top, \hat{\mu}_1 = (9.57, 1.92)^\top \\ \Sigma_0 &= \begin{pmatrix} 496.51 & 23.38 \\ 23.38 & 2.17 \end{pmatrix}, \Sigma_1 = \begin{pmatrix} 42.79 & 2.14 \\ 2.14 & 0.31 \end{pmatrix} \\ \hat{\mu} &= (25.30, 2.87)^\top, \Sigma = \begin{pmatrix} 557.89 & 30.51 \\ 30.51 & 2.27 \end{pmatrix} \end{aligned}$$

We invert the matrices  $\Sigma_1, \Sigma_2, \Sigma$  using R's `matlib::inv()`, and plug in the results in the LDA and QDA score formulas to obtain:<sup>21</sup>

21: Is this the best way to store the LDA/QDA functions in R?

```
gapminder.2011.te$d.0.LDA = -4.780979901 -
  0.0633308892*gapminder.2011.te$infant_mortality +
  2.645503201*gapminder.2011.te$fertility

gapminder.2011.te$d.1.LDA = -2.277022950 -
  0.1094188053*gapminder.2011.te$infant_mortality +
  2.313946886*gapminder.2011.te$fertility

gapminder.2011.te$d.0.QDA = -4.657130536 -
  0.002040555604*gapminder.2011.te$infant_mortality^2 +
  0.00614539606*gapminder.2011.te$infant_mortality +
  0.04390614038*gapminder.2011.te$infant_mortality*
  gapminder.2011.te$fertility +
  1.811698768*gapminder.2011.te$fertility -
  0.4663036203*gapminder.2011.te$fertility^2

gapminder.2011.te$d.1.QDA = -6.700309855 -
  0.01775013332*gapminder.2011.te$infant_mortality^2 -
  0.1263473930*gapminder.2011.te$infant_mortality +
  0.2427525754*gapminder.2011.te$infant_mortality*
  gapminder.2011.te$fertility +
  7.005915844*gapminder.2011.te$fertility -
  2.429442185*gapminder.2011.te$fertility^2
```

Thus,

$$\begin{aligned} \hat{\delta}_{0;\text{LDA}} &= -4.78 - 0.06x_1 + 2.65x_2 \\ \hat{\delta}_{1;\text{LDA}} &= -2.28 - 0.11x_1 + 2.31x_2 \\ \hat{\delta}_{0;\text{QDA}} &= -4.66 - 0.002x_1^2 + 0.01x_1 + 0.04x_1x_2 + 1.81x_2 - 0.47x_2^2 \\ \hat{\delta}_{1;\text{QDA}} &= -6.70 - 0.02x_1^2 - 0.13x_1 + 0.24x_1x_2 + 7.01x_2 - 2.43x_2^2 \end{aligned}$$

With the class probability estimates

$$\begin{aligned} \hat{p}_{1;\text{LDA}} &= \frac{\exp(\hat{\delta}_{1;\text{LDA}})}{\exp(\hat{\delta}_{0;\text{LDA}}) + \exp(\hat{\delta}_{1;\text{LDA}})}, \\ \hat{p}_{1;\text{QDA}} &= \frac{\exp(\hat{\delta}_{1;\text{QDA}})}{\exp(\hat{\delta}_{0;\text{QDA}}) + \exp(\hat{\delta}_{1;\text{QDA}})} \end{aligned}$$



```
gapminder.2011.te$p.1.LDA = exp(gapminder.2011.te$d.1.LDA)/
  (exp(gapminder.2011.te$d.0.LDA)+exp(gapminder.2011.te$d.1.LDA))
gapminder.2011.te$p.1.QDA = exp(gapminder.2011.te$d.1.QDA)/
  (exp(gapminder.2011.te$d.0.QDA)+exp(gapminder.2011.te$d.1.QDA))
```

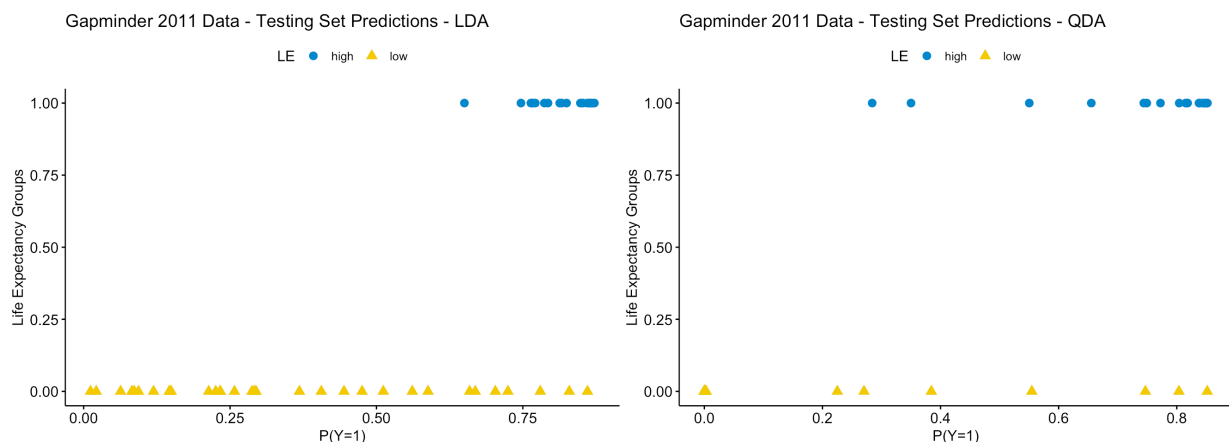
and the decision threshold set at  $\alpha = 0.5$ , the LDA and QDA life expectancy classifiers are defined on Te by

$$\hat{C}_{\alpha;LDA}(\mathbf{x}) = \begin{cases} 1 \text{ (high)} & \text{if } p_{1;LDA}(\mathbf{x}) \geq 0.5 \\ 0 \text{ (low)} & \text{else} \end{cases}$$

$$\hat{C}_{\alpha;QDA}(\mathbf{x}) = \begin{cases} 1 \text{ (high)} & \text{if } p_{1;QDA}(\mathbf{x}) \geq 0.5 \\ 0 \text{ (low)} & \text{else} \end{cases}$$

```
ggpubr::ggscatter(gapminder.2011.te, x="p.1.LDA", y="LE.resp",
  shape="LE", color="LE", palette="jco", size = 3,
  xlab="P(Y=1)", ylab = "Life Expectancy Groups",
  title = "Gapminder 2011 Data - Test Predictions - LDA")

ggpubr::ggscatter(gapminder.2011.te, x="p.1.QDA", y="LE.resp",
  shape="LE", color="LE", palette="jco", size = 3,
  xlab="P(Y=1)", ylab = "Life Expectancy Groups",
  title = "Gapminder 2011 Data - Test Predictions - QDA")
```



The  $\alpha = 0.5$  confusion matrices for the LDA and QDA classifiers are:

```
LDA.table = function(x,alpha){
  tmp = ifelse(x$p.1.LDA > alpha, 1, 0)
  LDA.table = table(factor(x$LE.resp, levels = 0:1),
    factor(tmp, levels = 0:1)) }

QDA.table = function(x,alpha){
  tmp = ifelse(x$p.1.QDA > alpha, 1, 0)
  QDA.table = table(factor(x$LE.resp, levels = 0:1),
    factor(tmp, levels = 0:1)) }
```

```
test.LDA=LDA.table(gapminder.2011.te,0.5)
test.QDA=QDA.table(gapminder.2011.te,0.5)
```

LDA $\alpha = 0.5$	prediction		QDA $\alpha = 0.5$	prediction	
	0	1		0	1
<b>actual</b>	0	22    10	<b>actual</b>	0	28    4
	1	0    18		1	2    16

In the LDA case, we further have

- accuracy =  $\frac{22+18}{22+10+0+18} = 80\%$
- misclassification rate =  $\frac{10+0}{22+10+0+18} = 20\%$
- FPR =  $\frac{0}{0+18} = 0\%$
- FNR =  $\frac{10}{22+10} = 31.25\%$
- TPR =  $\frac{22}{22+10} = 68.75\%$
- TNR =  $\frac{18}{0+18} = 100\%$

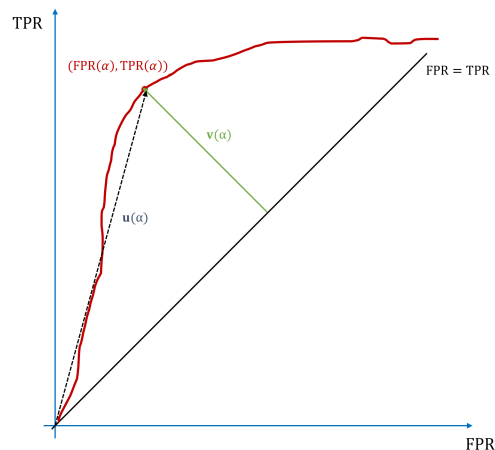
In the QDA case:

- accuracy =  $\frac{28+16}{28+4+2+16} = 88\%$
- misclassification rate =  $\frac{4+2}{28+4+2+16} = 12\%$
- FPR =  $\frac{2}{2+16} = 11.1\%$
- FNR =  $\frac{4}{28+4} = 12.5\%$
- TPR =  $\frac{28}{28+4} = 87.5\%$
- TNR =  $\frac{16}{2+16} = 88.9\%$

At first glance, it would certainly seem that the QDA model performs better (at a decision threshold of  $\alpha = 0.5$ ), but the FPR is not ideal. What would be the ideal value of  $\alpha$ ? How would we find it?

### 21.2.3 ROC Curve

The **receiver operating characteristic** (ROC) curve plots the true positive rate against the false positive rate for classifiers obtained by varying the decision threshold  $\alpha$  in  $[0, 1]$ . The important realization is that a classifier that is completely random would lie on the line  $\text{TPR} = \text{FPR}$ . Thus, the **ideal threshold** would be the one associated with the model which is **farthest** from that line.



**Figure 21.5:** Illustration of ROC curve concepts.

Let  $\mathbf{u}(\alpha)$  be the vector from  $\mathbf{0}$  to the  $(\text{FPR}(\alpha), \text{TPR}(\alpha))$  coordinates of the classifier with threshold  $\alpha$ , and let  $\mathbf{v}(\alpha)$  be the vector through  $(\text{FPR}(\alpha), \text{TPR}(\alpha))$  and perpendicular to the line  $\text{TPR} = \text{FPR}$ . The ideal  $\alpha^*$  satisfies:

$$\begin{aligned}
 \alpha^* &= \arg \max_{\alpha} \{\|\mathbf{v}(\alpha)\|\} \\
 &= \arg \max_{\alpha} \{\|\mathbf{v}(\alpha)\|^2\} \\
 &= \arg \max_{\alpha} \left\{ \left\| \mathbf{u}(\alpha) - \text{proj}_{(1,1)} \mathbf{u}(\alpha) \right\|^2 \right\} \\
 &= \arg \max_{\alpha} \left\{ \left\| (\text{FPR}(\alpha), \text{TPR}(\alpha)) - \text{proj}_{(1,1)} (\text{FPR}(\alpha), \text{TPR}(\alpha)) \right\|^2 \right\} \\
 &= \arg \max_{\alpha} \left\{ \left\| (\text{FPR}(\alpha) - \text{TPR}(\alpha), \text{TPR}(\alpha) - \text{FPR}(\alpha)) \right\|^2 \right\} \\
 &= \arg \max_{\alpha} \{(\text{FPR}(\alpha) - \text{TPR}(\alpha))^2\}.
 \end{aligned}$$

**Example** For the LDA and QDA classifiers built with the 2011 Gapminder data (see preceding section), the false positive rates (FPR), false negative rates (FNR), true positive rates (TPR), true negative rates (TNR), and misclassification rates (MCR) when the decision threshold  $\alpha$  varies from 0.01 to 0.99 by steps of length 0.01 are computed below:

```

fpr=c()
fnr=c()
tpr=c()
tnr=c()
mcr=c()

for(alpha in 1:99){
  tmp=LDA.table(gapminder.2011.te,alpha/100)
  mcr[alpha]=(tmp[1,2]+tmp[2,1])/sum(tmp)
  fpr[alpha]=tmp[2,1]/(tmp[2,1]+tmp[2,2])
  fnr[alpha]=tmp[1,2]/(tmp[1,1]+tmp[1,2])
  tnr[alpha]=tmp[2,2]/(tmp[2,1]+tmp[2,2])
  tpr[alpha]=tmp[1,1]/(tmp[1,1]+tmp[1,2]) }

plot(c(0,fpr),c(0,tpr), type = "b", pch = 21,
     col = "red", xlim=c(0,1), ylim=c(0,1),
     xlab="FPR",ylab="TPR",
     main="Receiver Operating Characteristic Curve - LDA")
abline(0,1)
(index=which((fpr-tpr)^2==max((fpr-tpr)^2)))
abline(fpr[index[1]]+tpr[index[1]],-1, col="green")

for(alpha in 1:99){
  tmp=QDA.table(gapminder.2011.te,alpha/100)
  mcr[alpha]=(tmp[1,2]+tmp[2,1])/sum(tmp)
  fpr[alpha]=tmp[2,1]/(tmp[2,1]+tmp[2,2])
  fnr[alpha]=tmp[1,2]/(tmp[1,1]+tmp[1,2])
  tnr[alpha]=tmp[2,2]/(tmp[2,1]+tmp[2,2])
  tpr[alpha]=tmp[1,1]/(tmp[1,1]+tmp[1,2]) }

```

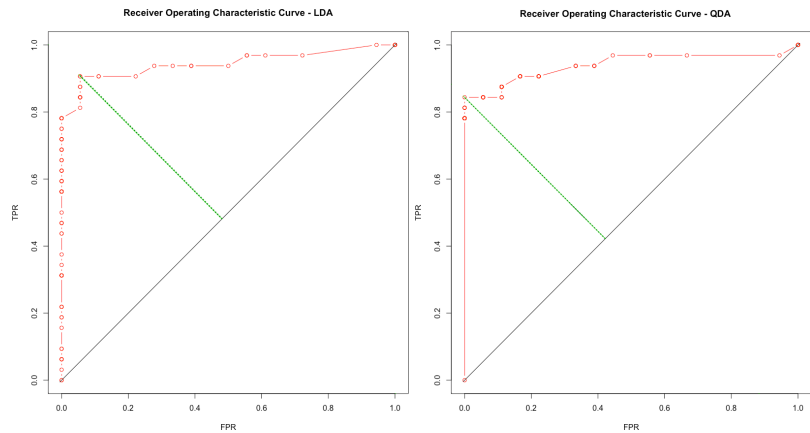
```

plot(c(0,fpr),c(0,tpr), type = "b", pch = 21,
     col = "red", xlim=c(0,1), ylim=c(0,1),
     xlab="FPR",ylab="TPR",
     main="Receiver Operating Characteristic Curve - QDA")
abline(0,1)
(index=which((fpr-tpr)^2==max((fpr-tpr)^2)))
abline(fpr[index[1]]+tpr[index[1]],-1, col="green")

```

```
[1] 73 74
```

```
[1] 28
```



In both frameworks, a number of models have identical (FPR, TPR) coordinates. With the LDA model, the ideal threshold is  $\alpha_{\text{LDA}}^* = 0.73$  (coordinates (0.056, 0.906)); with the QDA model, it is  $\alpha_{\text{QDA}}^* = 0.28$  (coordinates (0, 0.844)).

The corresponding confusion matrices are shown below.

LDA		prediction		QDA		prediction	
$\alpha_{\text{LDA}}^* = 0.73$		0	1	$\alpha_{\text{QDA}}^* = 0.28$		0	1
actual	0	29	3	actual	0	27	5
	1	1	17		1	0	18

Which model is best? It depends on the context of the task, and on the consequences of the choice. What makes the most sense here? Is there a danger of overfitting? Is parameter tuning acceptable, from a data massaging perspective? What effect does the choice of priors have?

While we can find the **optimal**  $\alpha$  according to the procedure highlighted above, there is another aspect of the ROC curve that may be of interest: in general, the larger the area under the curve is, the better the model may behave for non-optimal decision thresholds.

The metric is known as ROC AUC; technically, it varies between 0 and 1, but we since a classifier that is wrong more often than expected indirectly provides a classifier that is right more often than expected,<sup>22</sup> we focus instead on the area between the curve and the line  $\text{TPR} = \text{FPR}$ .

22: Simply predict the opposite of what it predicts.

## 21.3 Rare Occurrences

Before we continue and discuss other supervised approaches, we briefly touch on the problem of **rare occurrences** (or **unbalanced dataset**). Say we are looking to detect fraudulent transactions. We can build classifiers to approach this task using any number of methods, but there is a potential problem. If  $(100 - \varepsilon)\%$  of observations belong to the **normal** category, and  $\varepsilon\%$  to the **special** category, the model that predicts that EVERY observation is normal has  $(100 - \varepsilon)\%$  accuracy.

In practice, the vast majority of transactions are legitimate, so that  $0 < \varepsilon \ll 1$ ; the model in question has tremendous accuracy, even if it misses the point of the exercise altogether.

There are two general approaches to overcome this issue: either we **modify the algorithms** to take into account the asymmetric cost of making a classification error (through so-called **cost-sensitive classifiers** or **one-class models**), or we **modify the training data** to take into account the imbalance in the data. In the former case, we invite you to read the documentation of the methods that interest you to see how this could be achieved.

In the latter case, we could try to obtain more training data. This is the simplest method, but it is not always possible to do so, and it could be that the new data would follow the same pattern as the original data, which would leave us no better off than we were to start with.

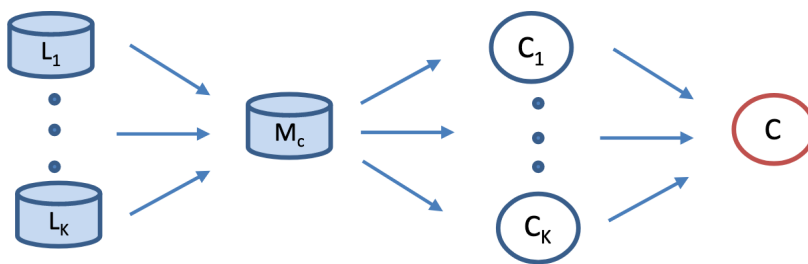
Another alternative is to create a new training set by either **undersampling** the majority class(es) or **oversampling** the under-represented class(es). We assume for now that there are only two classes in the data (the strategies can easily be adapted to multi-class problems).

**Undersampling** Let  $\text{Tr} = L \sqcup M_c$ , where  $L$  consists of all observations in the majority case, and  $M_c$  of all observations in the minority case; by assumption,  $|L| \gg |M_c|$ . Split  $L$  into  $K$  subsets  $L_1, \dots, L_K$ , each roughly of the same size;  $K$  should be selected so that  $|M_c| \ll |L_i|$ , for all  $i$  (in other words, even though  $|L_i|$  could be larger than  $|M_c|$ , it is not going to be substantially so).

We then construct  $K$  training sets

$$\text{Tr}_1 = L_1 \sqcup M_c, \dots, \text{Tr}_K = L_K \sqcup M_c;$$

for all  $1 \leq i \leq K$ , we train a classifier  $C_i$  (using a given algorithm) on  $\text{Tr}_i$ . Once that is done, we combine the predictions using bagging or other ensemble learning methods (see Section 21.5).



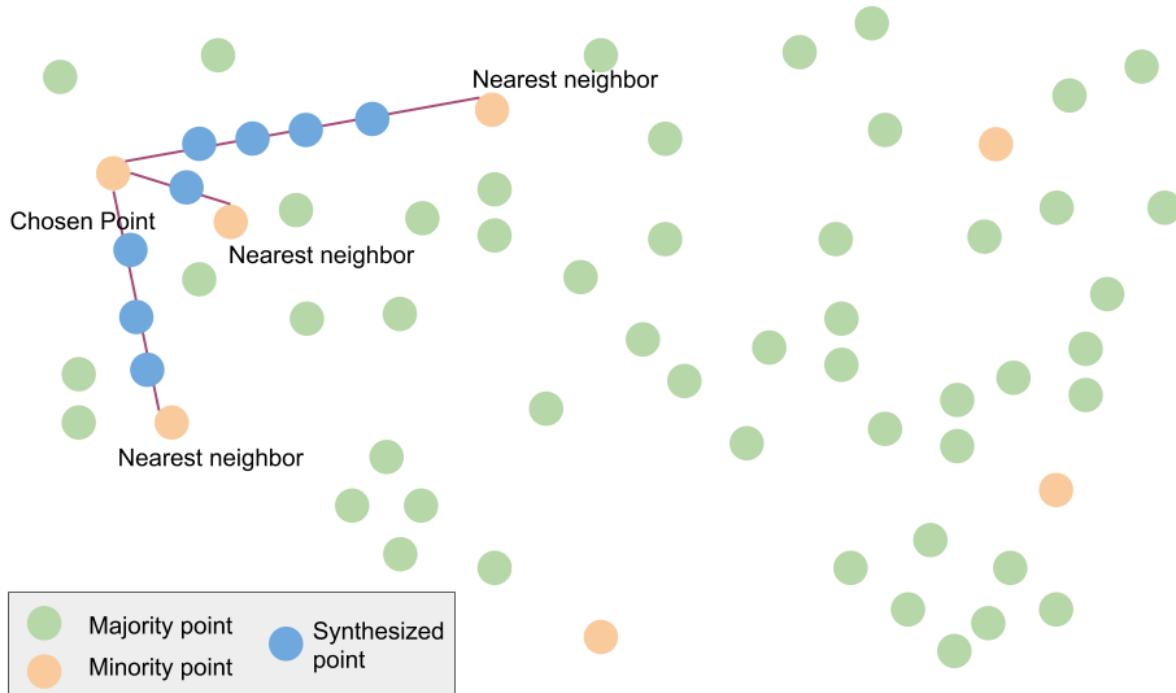


Figure 21.7: Illustration of oversampling (SMOTE) [author unknown].

**Oversampling** We can oversample the minority cases to create balanced datasets, but that introduces a dependency in the data that can have far-reaching effect when it comes to bias and variability.

**Synthetic Minority Oversampling Technique (SMOTE)** is a common approach which creates “synthetic” examples rather than oversampling with replacement – the same idea is used to create samples for handwriting recognition by perturbing training data (e.g., rotating, skewing, etc.) [239]:

1. select random integers  $k \ll \ell$ ;
2. draw a random sample  $\mathcal{V}_\ell$  of size  $\ell$  from the minority class  $M_c$ ;
3. for each  $\mathbf{x} \in \mathcal{V}_\ell$ , find the  $k$  nearest neighbors of  $\mathbf{x}$  in  $M_c$ , say  $\mathbf{z}_{\mathbf{x},1}, \dots, \mathbf{z}_{\mathbf{x},k}$ ;
4. compute the vectors  $\mathbf{v}_{\mathbf{x},1}, \dots, \mathbf{v}_{\mathbf{x},k}$ , originating from  $\mathbf{x}$  and ending at each of the  $\mathbf{z}_{\mathbf{x},1}, \dots, \mathbf{z}_{\mathbf{x},k}$ ;
5. draw random values  $\gamma_1, \dots, \gamma_k \sim \mathcal{U}(0, 1)$ , and multiply  $\mathbf{v}_{\mathbf{x},i}$  by  $\gamma_i$ , for each  $1 \leq i \leq k$ ;
6. the points found at  $\mathbf{x} + \gamma_i \mathbf{v}_{\mathbf{x},i}$ ,  $1 \leq i \leq k$ , are added to the set  $M_c$ .

This procedure is repeated until  $|M_c| \ll |L|$  (see Figure 21.7).

There are variants, where we always use the same  $k, \ell, \mathcal{V}_\ell$ , or where we only pick one of the  $k$  nearest neighbours, etc. In general, SMOTE increases **recall**, but it comes at the cost of lower **precision**.

We will have more to say about rare occurrences in Chapter 27, *Anomaly Detection and Outlier Analysis*.

## 21.4 Other Supervised Approaches

In this section, we present a number of **non-parametric** approaches, with a focus on classification methods (although we will also discuss regression problems):

- **classification and regression trees** (CART) [2–4, 240];
- **support vector machines** (SVW) [2–4, 241, 242];
- **artificial neural networks** (ANN) [177, 237, 243], and
- **naïve Bayes classification** (NBC).

### 21.4.1 Tree-Based Methods

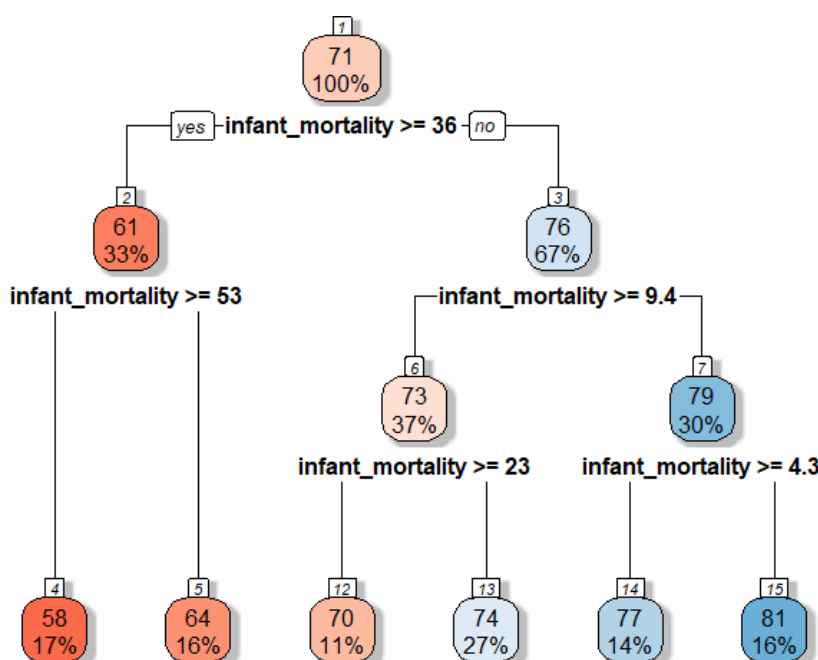
This family of methods involves **stratifying** or **segmenting** the predictor space into a small number of “simple” regions.

The set of **splitting rules** used to segment the space can be summarized using a **tree**, whence their name. Tree-based methods are **simple** and **easy to interpret**; but they don’t tend to be competitive with the best supervised learning methods when it comes to predictive accuracy.

Nevertheless, there are instances when the ease of interpretability overrules the lessened accuracy. Tree-based methods are applicable both to regression and to classification problems.

**Regression Trees** We introduce the important concepts via the 2011 Gapminder dataset.<sup>23</sup> In the figure on page 968, we saw that when  $X_1$  and  $X_2$  are both high,  $Y$  is low, and when  $X_1$  and  $X_2$  are both low,  $Y$  is high. But what is the pattern “in the middle”?

Below, we see a possible **regression tree** for the (full) dataset ( $N = 166$  observations).



23: The response  $Y$  is once again the life expectancy of nations, and the predictors  $X_1$  and  $X_2$  are the fertility rates and infant mortality rates per nation.

The tree can also be displayed as:

- 1) root (166) 70.82349
  - 2) infant\_mortality>=35.65 (54) 60.85370
    - 4) infant\_mortality>=52.9 (28) 58.30714 \*
    - 5) infant\_mortality< 52.9 (26) 63.59615 \*
  - 3) infant\_mortality< 35.65 (112) 75.63036
    - 6) infant\_mortality>=9.35 (62) 72.89516
      - 12) infant\_mortality>=22.85 (18) 69.50000 \*
      - 13) infant\_mortality< 22.85 (44) 74.28409 \*
    - 7) infant\_mortality< 9.35 (50) 79.02200
      - 14) infant\_mortality>=4.25 (23) 76.86087 \*
      - 15) infant\_mortality< 4.25 (27) 80.86296 \*

Node 1 is the tree's **root** (initial node) with 166 (100%) observations; the average life expectancy for these observations is 70.82.

The root is also the tree's first **branching point**, separating the observations into two groups: node 2 with 54 observations (33%), given by "infant mortality  $\geq 35.65$ ", for which the average life expectancy is 60.85, and node 3 with 112 observations (67%), given by "infant mortality  $< 35.65$ ", for which the average life expectancy is 75.63.

Note that  $54 + 112 = 166$  and that

$$\frac{54(60.81) + 112(75.63)}{54 + 112} = 70.82.$$

Node 2 is an **internal node** – it is further split into two groups: node 4 with 28 observations (17%), given with the additional rule "infant mortality  $\geq 52.9$ ", for which the average life expectancy is 58.31, and node 5 with 26 observations (16%), given with the additional rule "infant mortality  $< 52.9$ ", for which the average life expectancy is 63.60.

Note that  $28 + 26 = 54$  and that

$$\frac{28(58.31) + 26(63.60)}{28 + 26} = 60.85.$$

Both nodes 4 and 5 are **leaves** (final nodes, terminal nodes); the tree does not grow any further on that branch.

The tree continues to grow from node 3, eventually leading to 4 leaves on that branch (there are intermediate branching points). There are 6 leaves in total, 5 branching points (including the root) and the tree's **depth** is 3 (excluding the root).

Only one feature is used in the regression tree in this example: to make a prediction for a new observation, only infant mortality is needed. If it was 21, say, the observation's leaf would be node 13 and we would predict that the life expectancy of that nation would be 74.28.

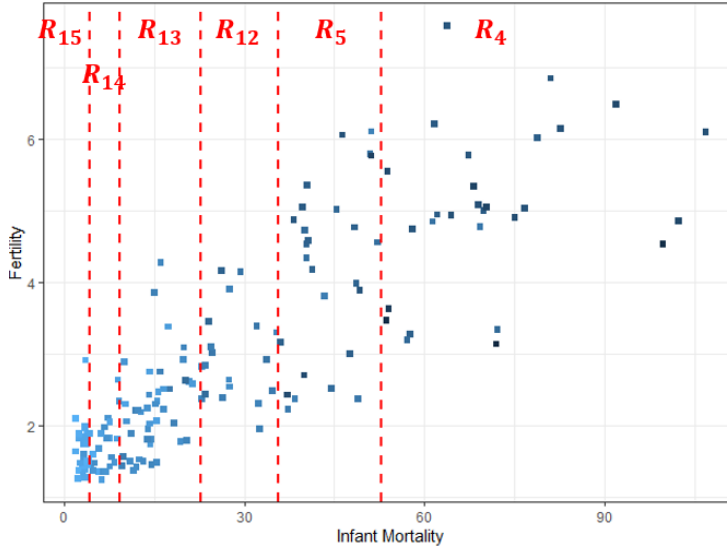
The tree diagram is a useful heuristic, especially since it allows the results to be displayed without resorting to a multi-dimensional chart, but it obscures the **predictor space's stratification**.



We can also write

$$\begin{aligned}
 R_4 &= \{(\text{infant mortality}, \text{fertility}) \mid \text{infant mortality} \geq 52.9\} \\
 R_5 &= \{(\text{infant mortality}, \text{fertility}) \mid 36.65 \leq \text{infant mortality} < 52.9\} \\
 R_{12} &= \{(\text{infant mortality}, \text{fertility}) \mid 22.85 \leq \text{infant mortality} < 35.65\} \\
 R_{13} &= \{(\text{infant mortality}, \text{fertility}) \mid 9.35 \leq \text{infant mortality} < 22.85\} \\
 R_{14} &= \{(\text{infant mortality}, \text{fertility}) \mid 4.25 \leq \text{infant mortality} < 9.35\} \\
 R_{15} &= \{(\text{infant mortality}, \text{fertility}) \mid \text{infant mortality} < 4.25\}
 \end{aligned}$$

It turns out that only infant mortality is involved in the definition of the tree's **terminal nodes**. The regions are shown below:



**Figure 21.8:** Stratification of the predictor space for the 2011 Gapminder data regression tree.

The regression tree model for life expectancy would thus be

$$\hat{y}_i = f(\mathbf{x}_i) = \text{Avg}\{y \mid \mathbf{x} \in R_{j(i)}\} = \begin{cases} 58.3, & j(i) = 4 \\ 63.6, & j(i) = 5 \\ 69.5, & j(i) = 12 \\ 74.3, & j(i) = 13 \\ 76.9, & j(i) = 14 \\ 80.9, & j(i) = 15 \end{cases}$$

where  $R_{j(i)}$  is the region in which  $\mathbf{x}_i$  falls. This tells us that infant mortality is the most important factor in determining life expectancy, with a negative correlation.<sup>24</sup> But it is not the only way to stratify the data: how is it an **optimal** tree?<sup>25</sup>

**Building A Regression Tree** The process is quite simple:

1. divide the predictor space  $\mathcal{X} \subseteq \mathbb{R}^p$  into a disjoint union of  $J$  regions:

$$\mathcal{X} = R_1 \sqcup \cdots \sqcup R_J;$$

2. for any  $\mathbf{x} \in R_j$ ,

$$\hat{y}(\mathbf{x}) = \text{Avg}\{y(\mathbf{z}) \mid \mathbf{z} \in R_j \cap \text{Tr}\}.$$

24: This interpretation is, of course, a coarse oversimplification, but it highlights the advantage of using a regression tree when it comes to **displaying, interpreting, and explaining** the results.

25: Recall that all supervised learning tasks are optimization problems.

26: It would be possible to define trees that are not locally constant, if required.

The second step tells us that trees are **locally constant**.<sup>26</sup>

In theory, the regions  $R_j$  could have any shape as long as they form a **disjoint cover** of  $\mathcal{X}$ ; in practice, we use **hyperboxes** with  $p-1$ -dimensional affine boundaries that are perpendicular/parallel to the  $p$  hyperplanes  $X_1 \dots \hat{X}_k \dots X_p, k = 1, \dots, p$ .

We find the optimal  $(R_1, \dots, R_J)$  by minimizing

$$\text{SSE} = \sum_{j=1}^J \sum_{\mathbf{x}_i \in R_j} (y_i - \hat{y}_{R_j})^2,$$

where  $\hat{y}_{R_j}$  is the mean response of  $y$  in  $R_j \cap \text{Tr}$ . In an ideal world, we would compute SSE for all partitions of  $\mathcal{X}$  into hyperboxes, and pick the one that minimizes SSE, but that is not computationally feasible, in general.

Instead, we use a growth algorithmic approach known as **recursive binary splitting**, which is both **top-down** (starts at the root and successively splits  $\mathcal{X}$  via 2 new branches down the tree) and **greedy** (at each step of the splitting process, the best choice is made there and now, rather than by looking at long-term consequences).

**Regression Tree Algorithm** The algorithm has 10 steps, but it is fairly straightforward.

1. Let  $\hat{y}_0 = \text{Avg}\{y(\mathbf{x}_i) \mid i = 1, \dots, N \text{ and } \mathbf{x}_i \in \text{Tr}\}$ .
2. Set the baseline  $\text{SSE}_0 = \sum_{i=1}^N (y_i - \hat{y}_0)^2$ .
3. For each  $1 \leq k \leq p$ , order the predictor values of  $X_k$  in Tr:  
 $\min_{1 \leq i \leq N} \{x_{i,k}\} = v_{k,1} \leq v_{k,2} \leq \dots \leq v_{k,N} = \max_{1 \leq i \leq N} \{x_{i,k}\}$ .
4. For each  $X_k$ , set  $s_{k,\ell} = \frac{1}{2}(v_{k,\ell} + v_{k,\ell+1}), \ell = 1, \dots, N-1$ .
5. For each  $k = 1, \dots, p, \ell = 1, \dots, N-1$ , define

$$R_1(k, \ell) = \{\vec{X} \in \mathbb{R}^p \mid X_k < s_{k,\ell}\}, \quad R_2(k, \ell) = \{\vec{X} \in \mathbb{R}^p \mid X_k \geq s_{k,\ell}\}.$$

Note that  $\mathcal{X} = R_1(k, \ell) \sqcup R_2(k, \ell)$  for all  $k, \ell$ .

6. For each  $k = 1, \dots, p, \ell = 1, \dots, N-1$ , set

$$\text{SSE}_1^{k,\ell} = \sum_{m=1}^2 \sum_{\vec{X}_i \in R_m(k,\ell)} (y_i - \hat{y}_{R_m(k,\ell)})^2,$$

where  $\hat{y}_{R_m(k,\ell)} = \text{Avg}\{y(\mathbf{x}) \mid \mathbf{x} \in \text{Tr} \cap R_m(k, \ell)\}$ .

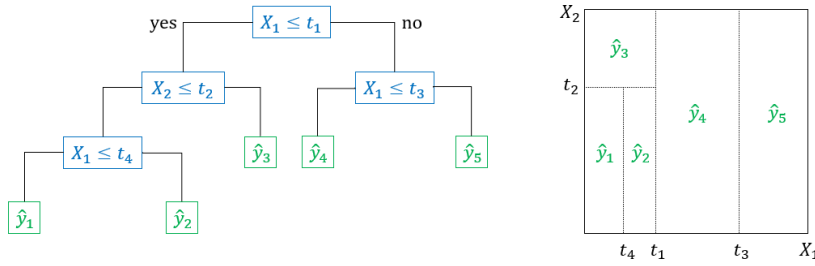
7. Find  $k^*, \ell^*$  for which  $\text{SSE}_1^{k,\ell}$  is **minimized**.
8. Define the **children sets**  $R_1^L = R_1(k^*, \ell^*)$  and  $R_1^R = R_2(k^*, \ell^*)$ .
9. While some children sets  $R_\mu^v$  still do not meet a **stopping criterion**, repeat steps 3 to 8, searching and minimizing SSE over  $\mathcal{X} \cap R_\mu^v$ , and producing a binary split  $R_{\mu+1}^L, R_{\mu+1}^R$ .<sup>27</sup>
10. Once the stopping criterion is met for all children sets, the tree's growth ceases, and  $\mathcal{X}$  has been partitioned into  $J$  regions (renumbering as necessary)

$$\mathcal{X} = R_1 \sqcup \dots \sqcup R_J,$$

on which the regression tree predicts the  $J$  responses  $\{\hat{y}_1, \dots, \hat{y}_J\}$ , according to  $\hat{y}_j = \text{Avg}\{y(\mathbf{x}) \mid \mathbf{x} \in R_j\}$ .

27: Multiple stopping criteria are used in practice, such as insisting that all final nodes contain 10 or fewer observations, etc.

For instance, if the training set was  $\text{Tr} = \{(x_{1,i}, x_{2,i}, y_i)\}_{i=1}^N$ , the algorithm might provide the regression tree in Figure 21.9.



**Figure 21.9:** Generic recursive binary partition regression tree for a two-dimensional predictor space, with 5 leaves.

In R, the recursive binary partition algorithm is implemented in package `rpart`'s function `rpart()`.

**Tree Pruning** Regression trees grown with the algorithm are prone to overfitting; they can provide good predictions on  $\text{Tr}$ , but they usually make shoddy predictions on  $\text{Te}$ .<sup>28</sup>

A smaller tree with fewer splits might lead to lower variance and better interpretability, at the cost of a little bias. Instead of simply growing a tree  $T_0$  until each leaf contains at most  $M$  observations, say,<sup>29</sup> it could be beneficial to **prune** it in order to obtain an **optimal subtree**.

We use **cost complexity pruning** (CCP) to build a sequence of candidate subtrees indexed by the complexity parameter  $\alpha \geq 0$ . For each such  $\alpha$ , find a subtree  $T_\alpha \subseteq T_0$  which minimizes

$$\text{SSE} + \text{complexity penalty} = \sum_{m=1}^{|T|} \sum_{\mathbf{x}_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha |T|,$$

where  $|T|$  represents the number of final nodes in  $T$ ; when  $\alpha$  is large, it is costly to have a complex tree.

This is similar to the bias-variance trade-off or the regularization framework: a good tree balances considerations of fit and complexity.

**Pruning Algorithm** Assume that a recursive binary splitting regression tree  $T_0$  has been grown on  $\text{Tr}$ , using a given stopping criterion:

1. apply CCP to  $T_0$  to obtain a “sequence”  $T_\alpha$  of subtrees of  $T_0$ ;
2. divide  $\text{Tr}$  into  $K$  folds;
3. for all  $k = 1, \dots, K$ , build a regression tree  $T_{\alpha,k}$  on  $\text{Tr} \setminus \text{Fold}_k$  and evaluate

$$\widehat{\text{MSE}}(\alpha) = \text{Avg}_{1 \leq k \leq K} \{\text{MSE}_k(\alpha) \text{ of } T_{\alpha,k} \text{ on } \text{Fold}_k\};$$

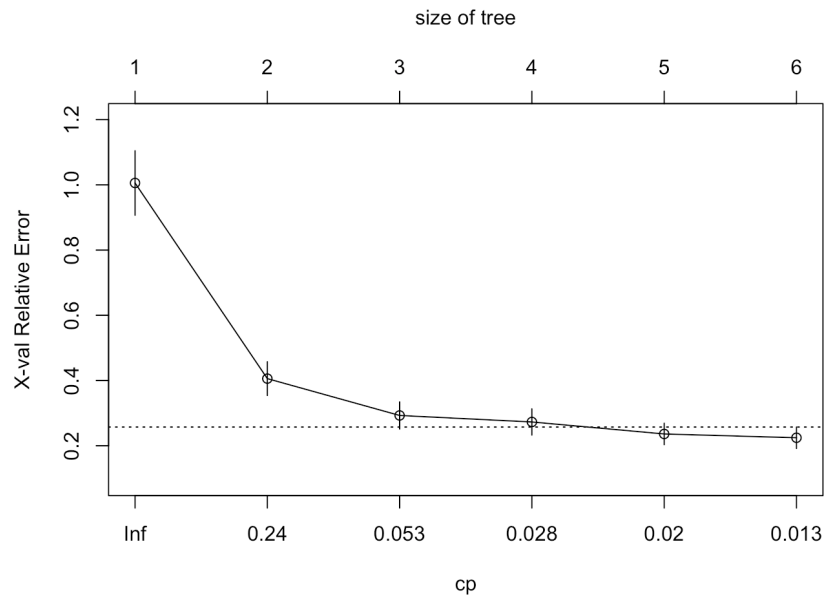
4. return  $T_{\alpha^*}$  from step 1, where  $\alpha^* = \arg \min_{\alpha} \{\widehat{\text{MSE}}(\alpha)\}$ .

The Gapminder 2011 tree is pruned in the Figures below, using the `rpart` functions `plotcp()` (the complexity parameter  $\alpha$  is denoted by `cp` in the code below) and `rpart()`.

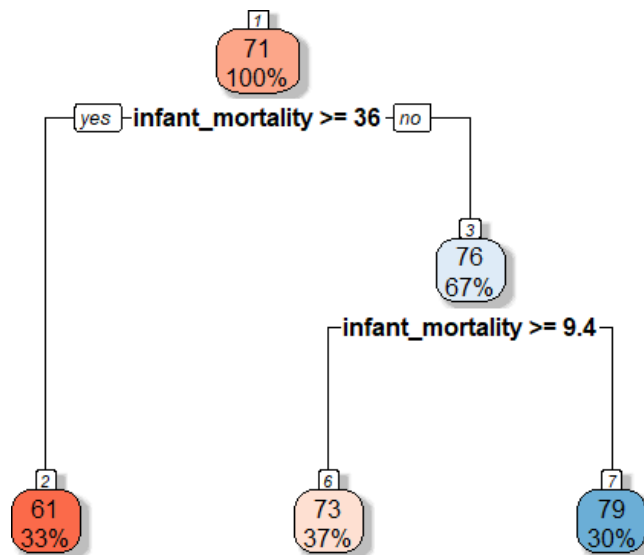
28: Because the resulting tree might be too complex – it captures noise as well as the signal.

29: Or whatever other stopping criterion might be appropriate.

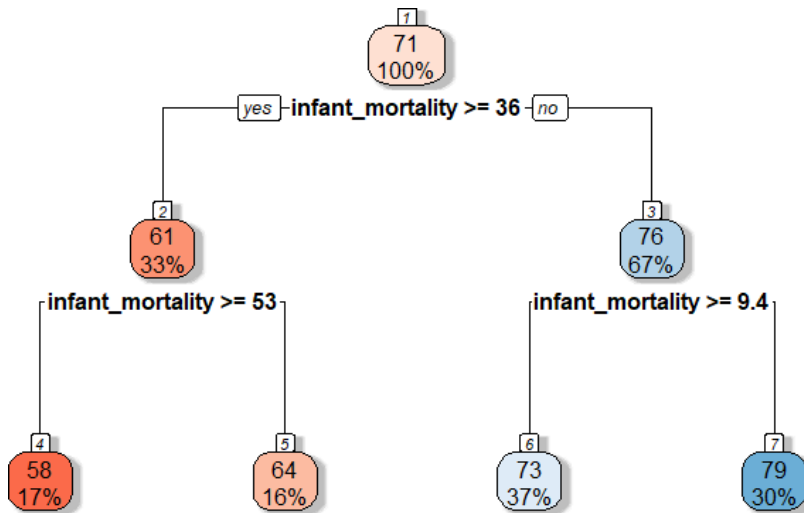
```
rpart::plotcp(reg.tree.1)
```



```
reg.tree.1.pruned.2 <- rpart::rpart(life_expectancy ~
  fertility + infant_mortality,
  data=gapminder.2011, cp=0.06)
rpart.plot::rpart.plot(reg.tree.1.pruned.2,
  box.palette="RdBu", shadow.col="gray", nn=TRUE)
```



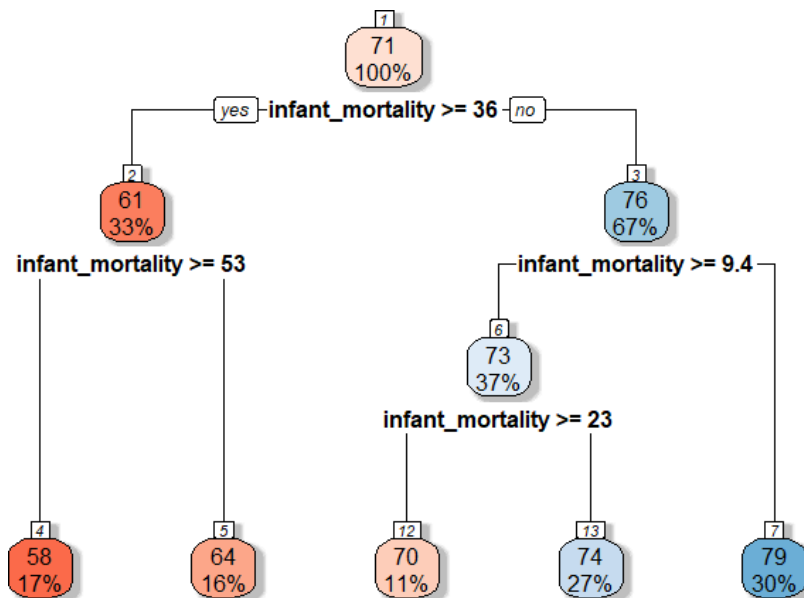
```
reg.tree.1.pruned.3 <- rpart::rpart(life_expectancy ~
  fertility + infant_mortality,
  data=gapminder.2011, cp=0.028)
rpart.plot::rpart.plot(reg.tree.1.pruned.3,
  box.palette="RdBu", shadow.col="gray", nn=TRUE)
```



```

reg.tree.1.pruned.4 <- rpart::rpart(life_expectancy ~
  fertility + infant_mortality,
  data=gapminder.2011, cp=0.02)
rpart.plot::rpart.plot(reg.tree.1.pruned.4,
  box.palette="RdBu", shadow.col="gray", nn=TRUE)

```



We plotted the complexity pruning parameter, and the pruned trees for  $cp = 0.06$ ,  $cp = 0.028$ , and  $cp = 0.02$  in the Gapminder 2011 example. Note that the tree's complexity increases when  $cp$  decreases.

**Classification Trees** The approach for classification is much the same, with a few appropriate substitutions:

1. prediction in a terminal node is either the **class label mode** or the **relative frequency** of the class labels;

2. SSE must be replaced by some other fit measure:

- the **classification error rate**:

$$E = \sum_{j=1}^J (1 - \max_k \{\hat{p}_{j,k}\}),$$

where  $\hat{p}_{j,k}$  is the proportion of  $\text{Tr}$  observations in  $R_j$  of class  $k$  (this measure is not a recommended choice, however);

- the **Gini index**, which measures the total variance across classes

$$G = \sum_{j=1}^J \sum_k \hat{p}_{j,k}(1 - \hat{p}_{j,k}),$$

which should be small when the nodes are **pure** ( $\hat{p}_{j,k} \approx 0$  or  $1$  throughout the regions), and

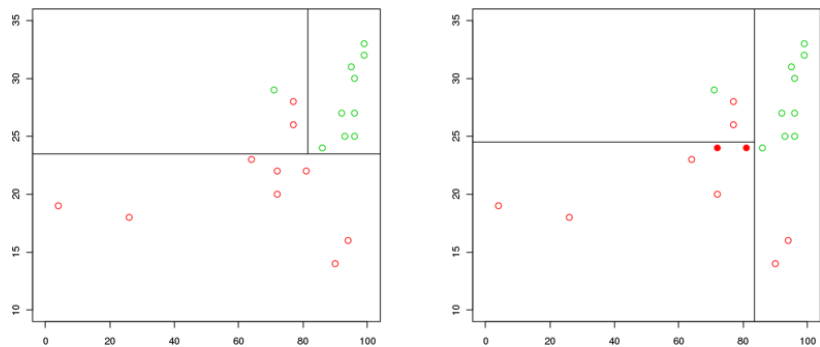
- the **cross-entropy deviance**

$$D = - \sum_{j=1}^J \sum_k \hat{p}_{j,k} \ln \hat{p}_{j,k},$$

which behaves like the Gini index, numerically.

One thing to note is that **classification and regression trees** (jointly known as CART) suffer from high variance and their structure is **unstable** – using different training sets typically gives rise to wildly varying trees.

As an extreme example, simply modifying the level of only one of the predictors in only two observations can yield a tree with a completely different topology, as in Figure 21.10.

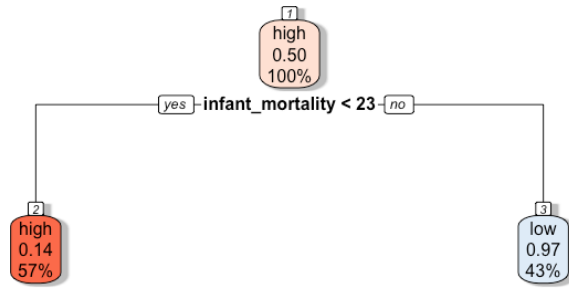


**Figure 21.10:** Different tree topologies with small changes in the training set (data modified from [43]).

This lack of robustness is a definite strike against the use of CART; despite this, the relative ease of their implementation makes them a popular classification tool.

**Examples** A classification tree for the response LE in the Gapminder 2011 dataset is shown below:

```
reg.tree.2 <- rpart::rpart(LE~fertility +
  infant_mortality + gdp, data=gapminder.2011)
rpart.plot::rpart.plot(reg.tree.2, box.palette="RdBu",
  shadow.col="gray", nn=TRUE)
```



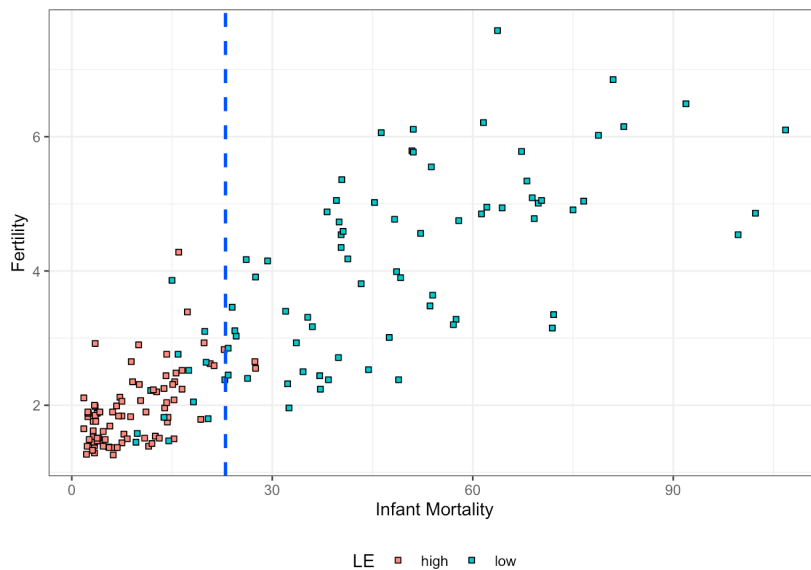
- 1) root (166) high/low (0.5 0.5)
- 2) infant\_mortality < 23 (94) high (0.862 0.138)
- 3) infant\_mortality >= 23 (72) low (0.028 0.972)

The stratification in the scatter plot is shown below.

```

library(ggplot2)
ggplot(gapminder.2011) +
  geom_point(aes(fill=LE, x=infant_mortality, y=fertility),
             pch=22) + theme_bw() +
  theme(legend.position = "bottom") +
  geom_vline(xintercept = c(23), linetype="dashed",
             color = "blue", size=1) + xlab("Infant Mortality") +
  ylab("Fertility")

```



Note that this tree should not be used for predictions as it was not built on a training subset of the data.

We now revisit the Iowa Housing Price ([VE\\_Housing.csv](#) <sup>30</sup>, modified from [233]) example of Section 20.5 (*Splines*). We build a CART for the sale price, requiring at least 5 observations per leaf.<sup>30</sup>

We only keep those columns for which there are no missing values, for simplicity's sake; in real-world applications, this is not usually a reasonable strategy (see Chapter 15, *Data Preparation*).

30: Recall that we had built a training set `dat.train` with  $n = 1160$  observations relating to the selling price `SalePrice` of houses in Ames, Iowa.

n= 1160

node), split, n, deviance, yval  
 \* denotes terminal node

```

1) root 1160 7371073.00 180.1428
  2) OverallQual< 7.5 985 2301952.00 157.4737
    4) Neighborhood=Blueste,BrDale,BrkSide,Edwards,IDOTRR,MeadowV,Mitchel,NAmes,NPkvill,OldTown,
      Sawyer,SWISU 580 672022.40 132.2839
      8) X1stFlrSF< 1050.5 332 240468.90 118.2474 *
      9) X1stFlrSF>=1050.5 248 278574.40 151.0747 *
    5) Neighborhood=Blmngtn,ClearCr,CollgCr,Crawfor,Gilbert,NoRidge,NridgHt,NWAmes,SawyerW,
      Somerst,StoneBr,Timber,Veenker 405 734856.30 193.5480
      10) GrLivArea< 1732.5 281 296100.70 178.2365
      20) GrLivArea< 1204 56 25438.63 143.0286 *
      21) GrLivArea>=1204 225 183967.70 186.9993 *
      11) GrLivArea>=1732.5 124 223588.10 228.2459 *
  3) OverallQual>=7.5 175 1713865.00 307.7376
    6) OverallQual< 8.5 126 498478.90 273.6180
      12) GrLivArea< 1925.5 71 167331.90 246.3000 *
      13) GrLivArea>=1925.5 55 209762.20 308.8831 *
    7) OverallQual>=8.5 49 691521.30 395.4736
      14) Neighborhood=CollgCr,Edwards,Gilbert,NridgHt,Somerst,StoneBr,Timber,Veenker
        44 358089.40 373.9441
        28) Neighborhood=CollgCr,Edwards,Somerst,Timber 11 39962.11 293.0025 *
        29) Neighborhood=Gilbert,NridgHt,StoneBr,Veenker 33 222038.20 400.9246
          58) GrLivArea< 2260 20 42801.90 358.2032 *
          59) GrLivArea>=2260 13 86576.40 466.6498 *
      15) Neighborhood=NoRidge 5 133561.60 584.9336 *
```

```

dat.Housing = read.csv("VE_Housing.csv",
  header=TRUE, stringsAsFactors = TRUE)
missing = attributes(which(apply(is.na(dat.Housing), 2,
  sum) > 0))$names
dat.Housing.new = dat.Housing[,!colnames(dat.Housing) %in%
  missing]
dat.Housing.new = subset(dat.Housing.new, select = -c(Id))

set.seed(1234) # for replicability
n.train = 1160
ind.train = sample(1:nrow(dat.Housing.new), n.train)
dat.train = dat.Housing.new[ind.train,]
dat.test = dat.Housing.new[-ind.train,]
(RT = rpart::rpart(SalePrice ~ ., data=dat.train,
  minbucket=5))
```

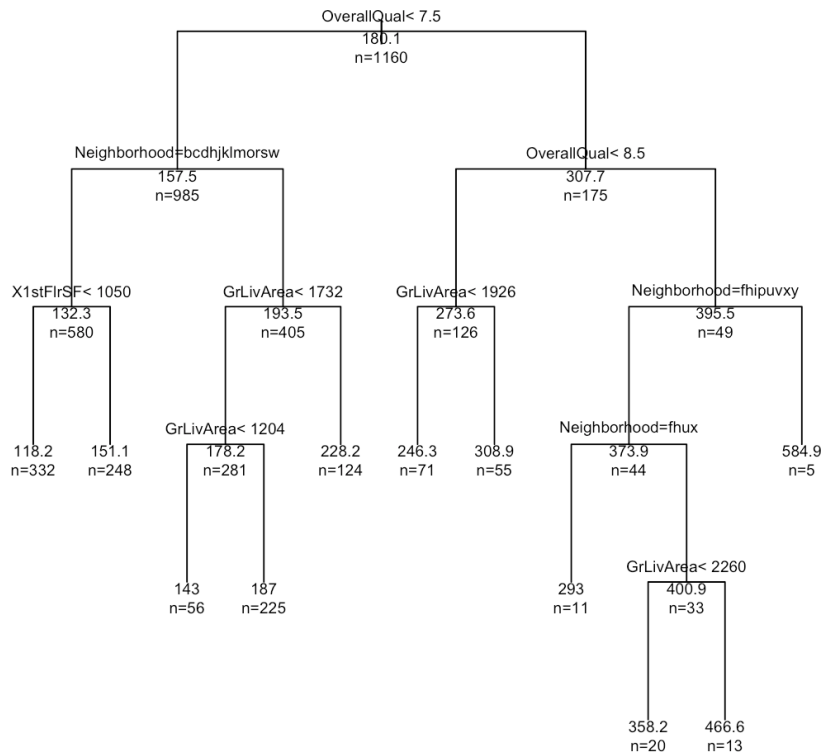
The tree is reasonably complex and fairly difficult to read, especially since there are so many categorical levels in some of the branching nodes.

There are multiple ways to provide a visual display that makes it easier to read the tree.

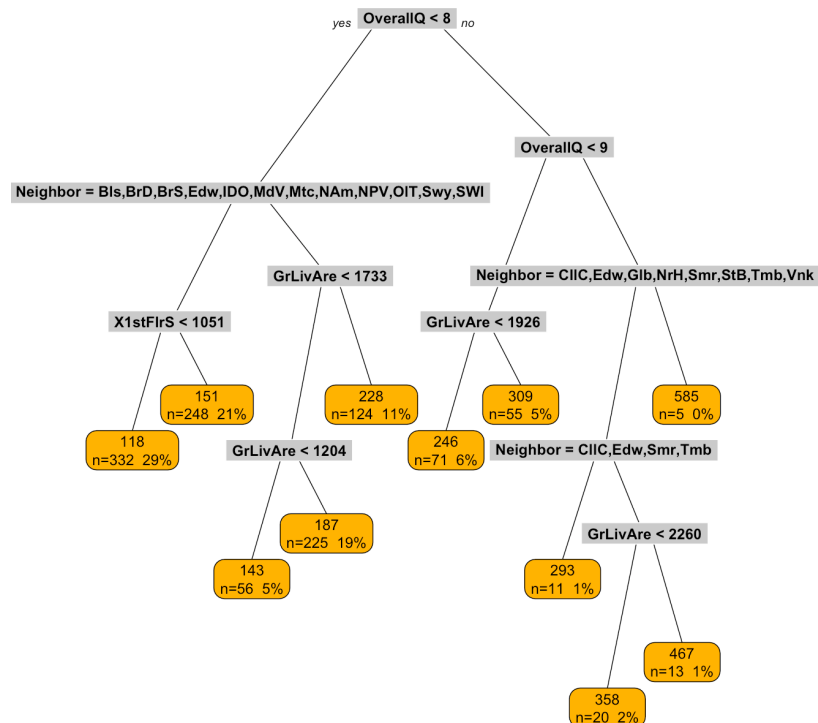
```

plot(RT, margin=0.05, uniform=TRUE)
text(RT, all=TRUE, use.n=TRUE, fancy=FALSE, cex=0.6)
```

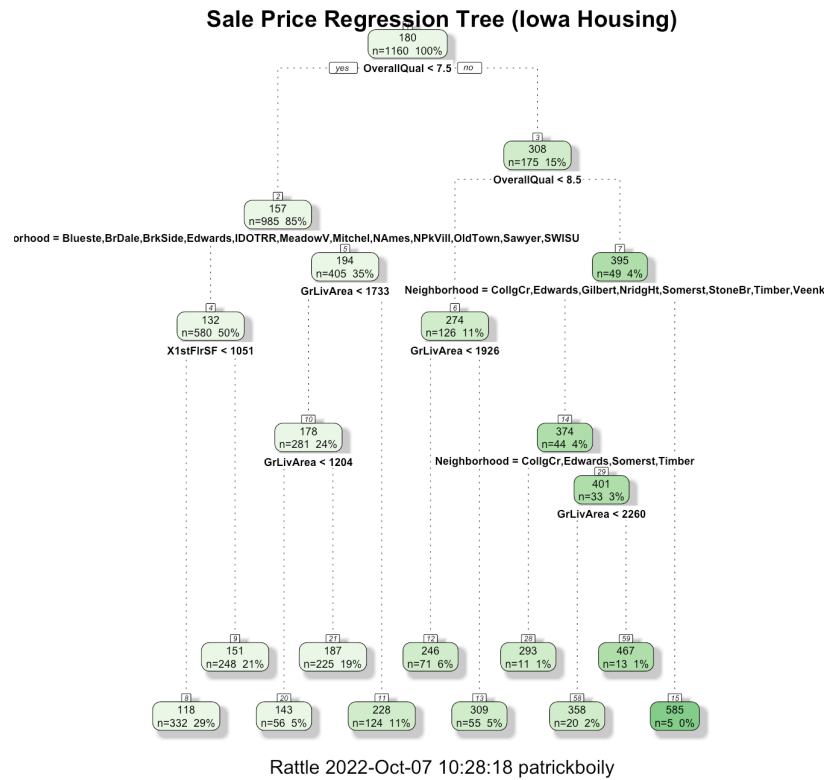




```
rpart.plot::prp(RT,extra=101,
  box.col="orange",split.box.col="gray")
```

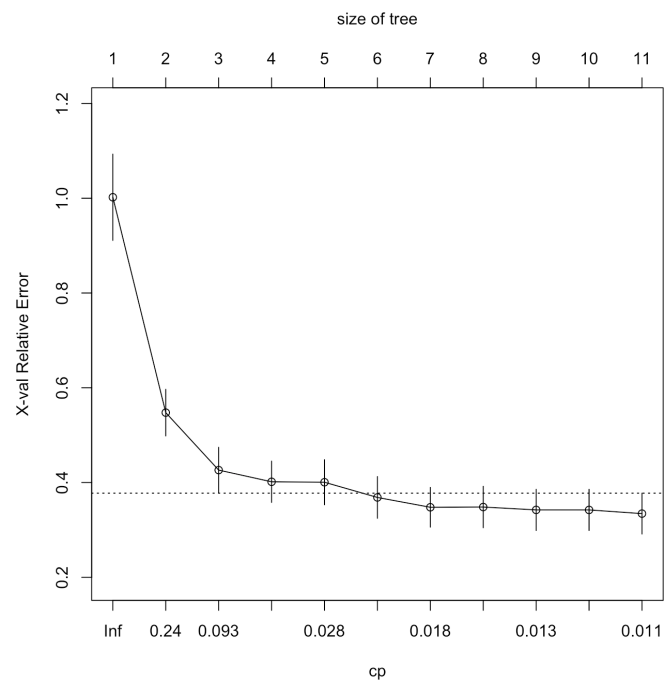


```
rattle::fancyRpartPlot(RT, main="Sale Price Regression Tree
(Iowa Housing)")
```



These are fully grown trees. Next we use `rpart`'s `plotcp()` to determine how to control the tree's growth (i.e., we prune the tree).

```
set.seed(1234) # for replicability
rpart::plotcp(RT)
```



From this plot, we see that the tuning parameter should be around 0.024, so we prune the tree as follows:

```
(RT.p = rpart::prune(RT, cp=0.024))
```

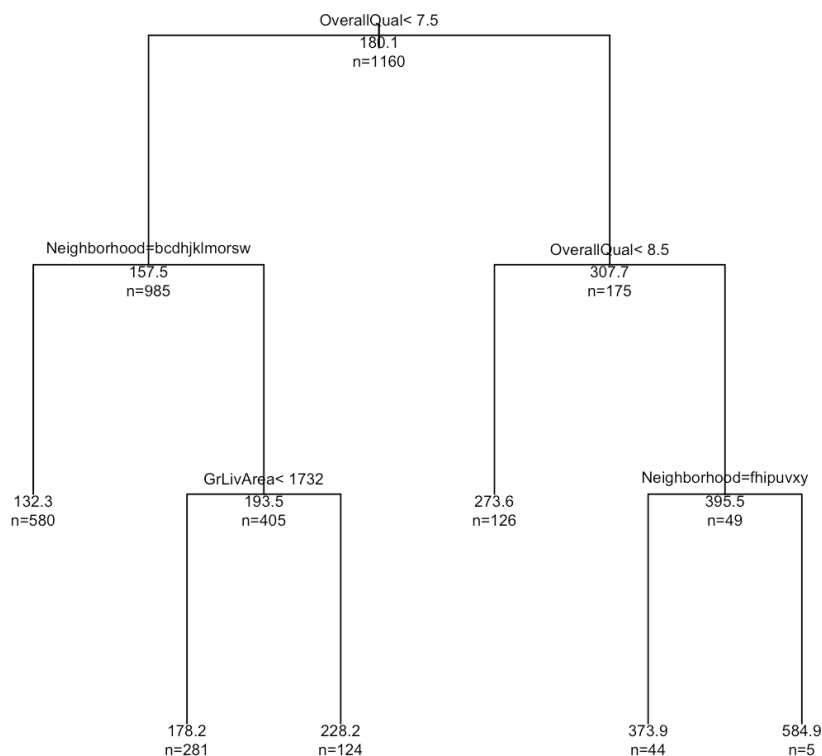
```
n= 1160
```

```
node), split, n, deviance, yval
  * denotes terminal node
```

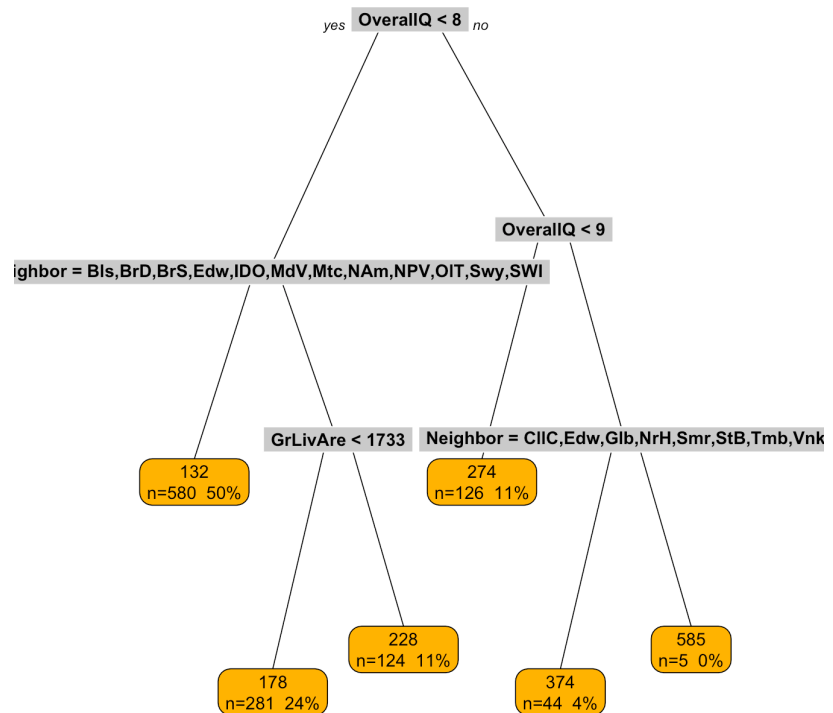
```
1) root 1160 7371073.0 180.1428
  2) OverallQual< 7.5 985 2301952.0 157.4737
    4) Neighborhood=Blueste,BrDale,BrkSide,Edwards,IDOTRR,MeadowV,Mitchel,NAmes,NPkvill,OldTown,
      Sawyer,SWISU 580 672022.4 132.2839 *
    5) Neighborhood=Blmngtn,ClearCr,CollgCr,Crawfor,Gilbert,NoRidge,NridgHt,NWAmes,SawyerW,
      Somerst,StoneBr,Timber,Veenker 405 734856.3 193.5480
      10) GrLivArea< 1732.5 281 296100.7 178.2365 *
      11) GrLivArea>=1732.5 124 223588.1 228.2459 *
  3) OverallQual>=7.5 175 1713865.0 307.7376
    6) OverallQual< 8.5 126 498478.9 273.6180 *
    7) OverallQual>=8.5 49 691521.3 395.4736
      14) Neighborhood=CollgCr,Edwards,Gilbert,NridgHt,Somerst,StoneBr,Timber,Veenker
        44 358089.4 373.9441 *
      15) Neighborhood=NoRidge 5 133561.6 584.9336 *
```

We go from 11 to 6 leaves. The structure of the pruned tree is plotted below.

```
plot(RT.p, margin=0.05, uniform=TRUE)
text(RT.p, all=TRUE, use.n=TRUE, fancy=FALSE, cex=0.6)
```

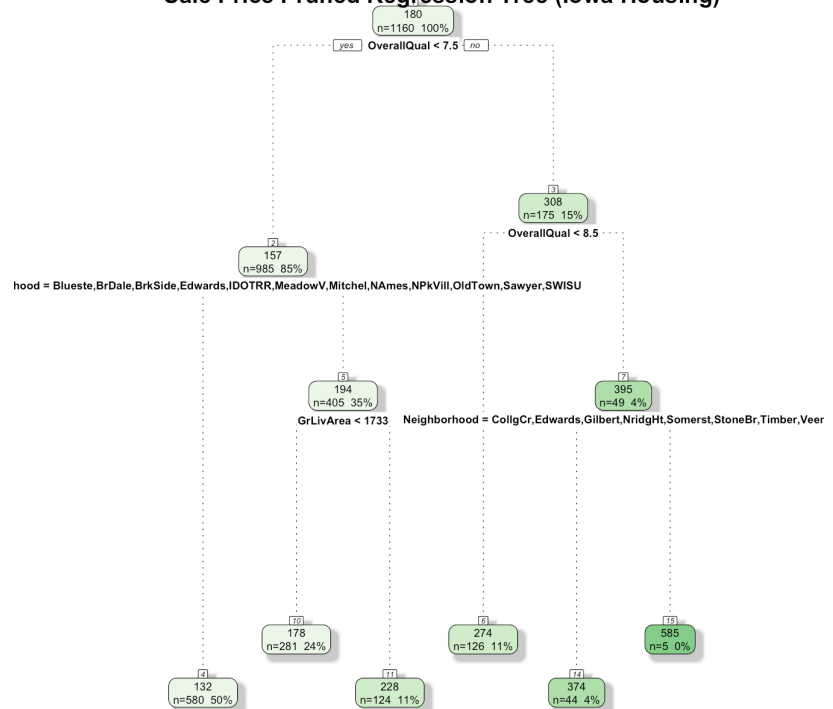


```
rpart.plot::prp(RT.p,extra=101, box.col="orange",
split.box.col="gray")
```



```
rattle::fancyRpartPlot(RT.p, main="Sale Price Pruned
Regression Tree (Iowa Housing)")
```

**Sale Price Pruned Regression Tree (Iowa Housing)**



Rattle 2022-Oct-07 10:28:20 patrickboily

Now that we have a full regression tree and a pruned tree, our last task is to see how well they perform as predictive models on the test data `dat.test`.

For the full tree, we compute the reduction in SSE using the **predictions**:

```
yhat.RT = predict(RT, dat.test)
SSE.RT = sum((yhat.RT-dat.test$SalePrice)^2)
SSE.average = sum((mean(dat.test$SalePrice) -
                    dat.test$SalePrice)^2)
round((1-SSE.RT/SSE.average), digits=3)
```

```
[1] 0.703
```

For the pruned tree, the corresponding reduction is:

```
yhat.RT.p = predict(RT.p, dat.test)
SSE.RT.p = sum((yhat.RT.p-dat.test$SalePrice)^2)
round((1-SSE.RT.p/SSE.average), digits=3)
```

```
[1] 0.646
```

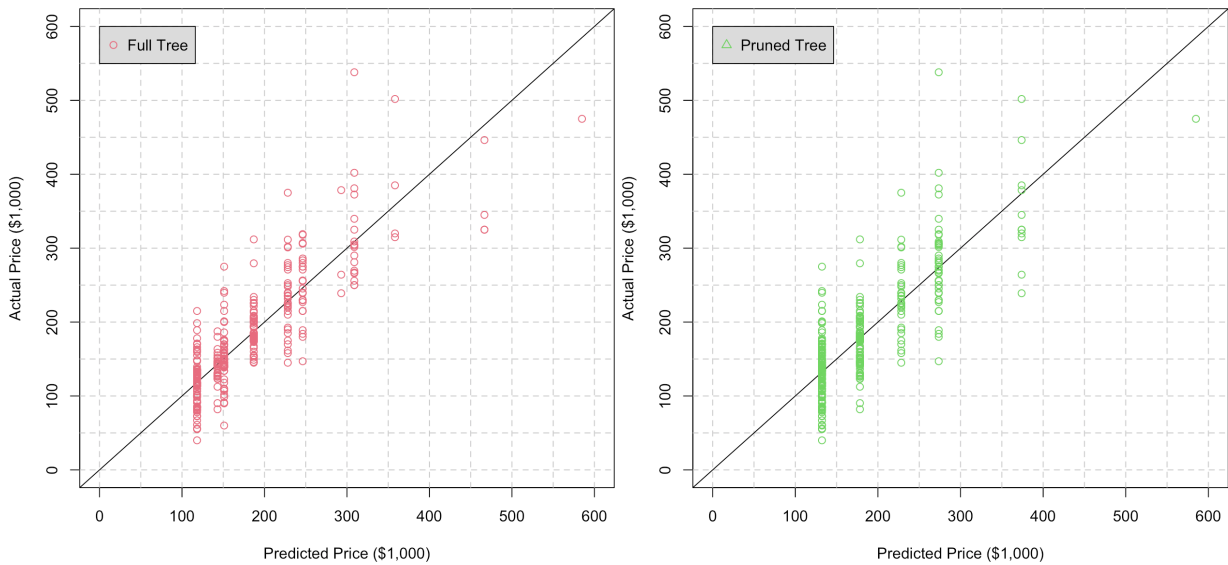
This suggests that pruning is a reasonable approach, in this case (based on `Tr`). The predictions of both trees are plotted against the actual `SalePrice` values in the next plots.

```
xlimit = ylimit = c(0,600)

plot(NA, col=2, xlim=xlimit, ylim=ylim,
     xlab="Predicted Price ($1,000)",
     ylab="Actual Price ($1,000)")
abline(h=seq(0,600, length.out=13), lty=2, col="grey",
       v=seq(0,600, length.out=13))
abline(a=0, b=1)
points(yhat.RT, dat.test$SalePrice, col=2)
legend(0,600, legend=c("Full Tree"), col=c(2),
      pch=rep(1), bg='light grey')

plot(NA, col=2, xlim=xlimit, ylim=ylim,
     xlab="Predicted Price ($1,000)",
     ylab="Actual Price ($1,000)")
abline(h=seq(0,600, length.out=13), lty=2, col="grey",
       v=seq(0,600, length.out=13))
abline(a=0, b=1)
points(yhat.RT.p, dat.test$SalePrice, col=3)
legend(0,600, legend=c("Pruned Tree"), col=c(3),
      pch=rep(2), bg='light grey')
```

Obviously, there are some departures from the actual response values, but given that the regression trees can only predict a small number of selling prices (corresponding to the tree leaves), these predictions are reasonably accurate.



```
cor(yhat.RT, dat.test$SalePrice)
cor(yhat.RT.p, dat.test$SalePrice)
```

```
[1] 0.8412035
```

```
[1] 0.8047534
```

How do these CART predictions compare with the MARS predictions of Section 20.6? The Iowa Housing dataset contains information about sale prices from 2006 to 2010; would you use the model to make predictions about 2022 sale prices?

---

Classification trees work in the same manner, although the evaluation step can be conducted in two ways: we can build trees that predict class membership (`type="class"`) or probability of class membership (`type="prob"`). Examples of how to work with these `predict()` options are provided in Section 19.7, *Classification: Kyphosis Dataset*.

## 21.4.2 Support Vector Machines

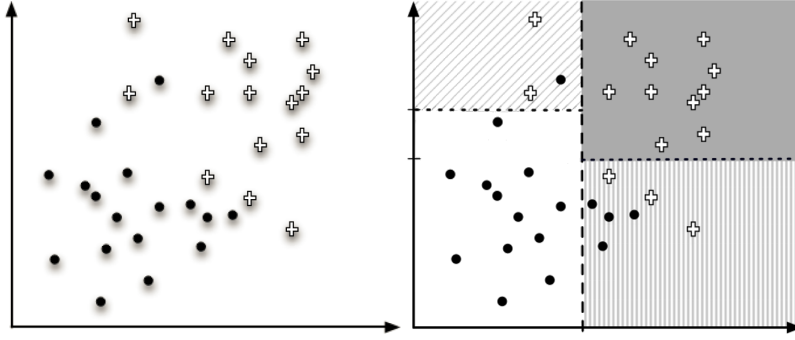
This next classifier is more sophisticated, from a mathematical perspective. It was invented by computer scientists in the 1990s.

**Support vector machines** (SVM) attempt to find hyperplanes that separate the classes in the feature space. On the left in Figure 21.11, we see an artificial data with 3 features:  $X_1$  and  $X_2$  (numerical),  $Y$  (categorical, represented by different symbols).

We grow a classification tree (perhaps the one shown on the right in Figure 21.11): two of the leaves are pure, but the risk of misclassification is fairly large in the other 2 (at least for that tree).<sup>31</sup> Without access to more features, that tree is as good as it gets.<sup>32</sup>

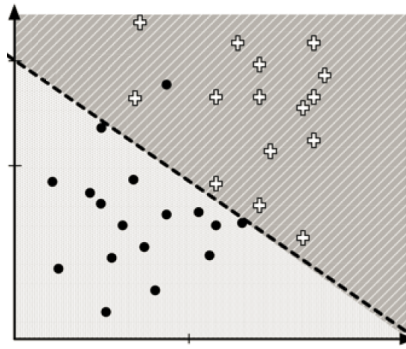
31: The tree is not unique, obviously, but any other tree with separators parallel to the axes will only be marginally better, at best.

32: To be sure, we could create an intricate decision tree with more than  $2^2 = 4$  separating lines, but that is undesirable for a well-fitted tree.



**Figure 21.11:** Two-class artificial dataset (left) and classification tree (right) [168].

But it is easy to draw a **decision curve** which improves on the effectiveness of the decision tree (see Figure 21.12): a single observation is misclassified by this rule.<sup>33</sup>



33: Perfect separation could lead to over-fitting.

**Figure 21.12:** Separating hyperplane on a two-class artificial dataset [168].

Separating hyperplanes do not always exist; we may need to:

- extend our notion of **separability**, and/or
- extend the feature space so separation becomes possible.

A **hyperplane**  $H_{\beta, \beta_0} \subseteq \mathbb{R}^p$  is an affine (“flat”) subset of  $\mathbb{R}^p$ , with

$$\dim(H_{\beta, \beta_0}) = p - 1;$$

in other words, it can be described by

$$H_{\beta, \beta_0} : \beta_0 + \beta^T \mathbf{x} = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p = 0.$$

The vector  $\beta$  is normal to  $H_{\beta, \beta_0}$ ; if  $\beta_0 = 0$ ,  $H_{\beta, \beta_0}$  goes through the origin in  $\mathbb{R}^p$ . Set  $F(\mathbf{x}) = \beta_0 + \beta^T \mathbf{x}$ ; then  $F(\mathbf{x}) > 0$  for points on one “side” of  $H_{\beta, \beta_0}$  and  $F(\mathbf{x}) < 0$  for points on the other.<sup>34</sup>

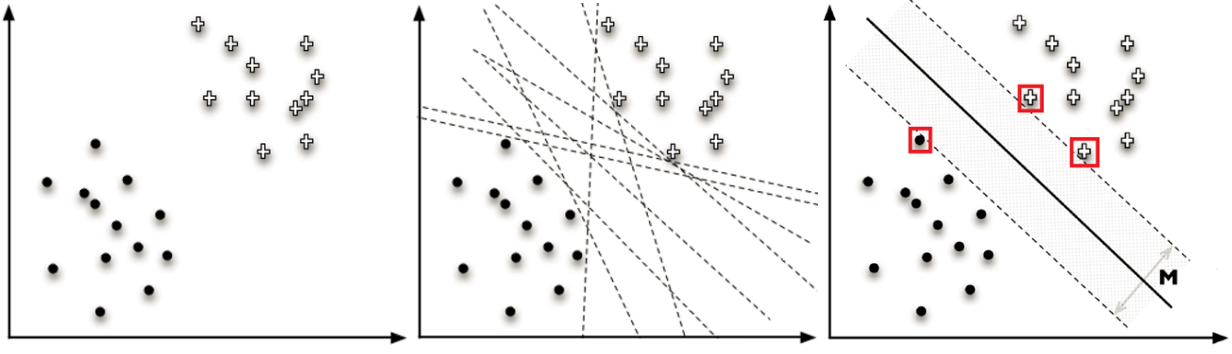
34:  $F(\mathbf{x}) = 0$  for points on  $H_{\beta, \beta_0}$ .

In a binary classification problem with  $\mathcal{C} = \{C_1, C_2\} = \{\pm 1\}$ , if

$$y_i F(\mathbf{x}_i) > 0, \quad \text{for all } (\mathbf{x}_i, y_i) \in \text{Tr}$$

(or,  $y_i F(\mathbf{x}_i) < 0$  for all  $(\mathbf{x}_i, y_i) \in \text{Tr}$ ), then  $F(\mathbf{x}) = 0$  determines a **separating hyperplane** for Tr (which does not need to be unique, see Figure 21.12), and we say that Tr is **linearly separable**.

Among all separating hyperplanes, the one which provides the widest separation between the two classes is the **maximal margin hyperplane** (MMH); training observations on the boundary of the **separating strip** are called the **support vectors** (see observations in Figure 21.13).



**Figure 21.13:** Artificial linearly separable subset of a two-class dataset (left), with separating hyperplanes (centre), maximal margin hyperplane with support vectors (right) [168].

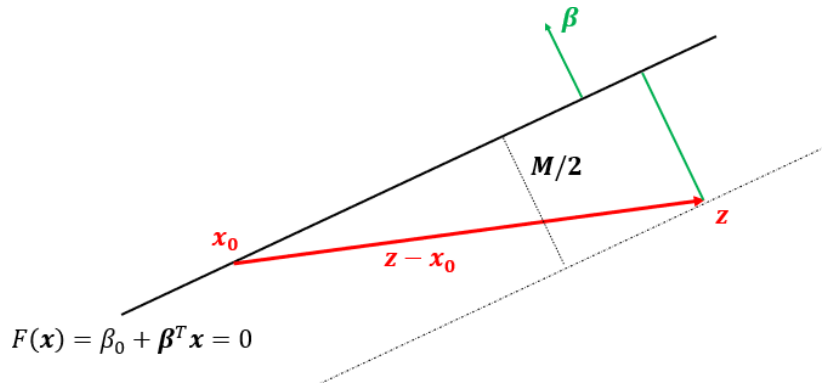
The classification problem simplifies, as always, to a constrained optimization problem:

$$(\beta^*, \beta_0^*) = \arg \max_{(\beta, \beta_0)} \{M_{(\beta, \beta_0)}\} \quad \text{s.t.} \quad y_i(\beta_0 + \beta x_i) \geq M_{(\beta, \beta_0)}$$

for all  $(x_i, y_i) \in \text{Tr}$ , with MMH given by  $F(\mathbf{x}) = \beta_0^* + \beta^* \mathbf{x} = 0$ .

Any hyperplane can be expressed in an uncountable number of ways; the MMH for which  $|F(\mathbf{x}^*)| = 1$  for all support vectors  $\mathbf{x}$  provides a **canonical representation**). From geometry, we know that the distance from the canonical maximal margin hyperplane  $H_{\beta, \beta_0}$  to any point  $\mathbf{z}$  can be computed using vector projections.

Let  $\mathbf{x}_0$  be a point on MMH, i.e.,  $F(\mathbf{x}_0) = \beta_0 + \beta^T \mathbf{x}_0 = 0$ , as shown below:



In particular, note that  $\beta_0 = -\beta^T \mathbf{x}_0$ . Then,

$$\begin{aligned} \frac{M}{2} &= \text{dist}(\mathbf{z}, H_{\beta, \beta_0}) = \left\| \text{proj}_{\beta}(\mathbf{z} - \mathbf{x}_0) \right\| = \left\| \frac{\beta^T (\mathbf{z} - \mathbf{x}_0)}{\|\beta\|^2} \beta \right\| \\ &= \frac{|\beta^T (\mathbf{z} - \mathbf{x}_0)|}{\|\beta\|^2} \|\beta\| = \frac{|\beta^T \mathbf{z} - \beta^T \mathbf{x}_0|}{\|\beta\|} = \frac{|F(\mathbf{z})|}{\|\beta\|}. \end{aligned}$$

If  $\mathbf{z}$  is a support vector, then  $F(\mathbf{z}) = 1$ , and

$$\frac{M}{2} = \text{dist}(\mathbf{z}, H_{\beta, \beta_0}) = \frac{1}{\|\beta\|}.$$



Maximizing the margin  $M$  is thus equivalent to minimizing  $\frac{\|\beta\|}{2}$ , and, since the square function is monotonic,

$$\arg \max_{(\beta, \beta_0)} \{M \mid y_i(\beta_0 + \beta^\top \mathbf{x}_i), \forall \mathbf{x}_i \in \text{Tr}\}$$

is equivalent to

$$\arg \min_{(\beta, \beta_0)} \left\{ \frac{1}{2} \|\beta\|^2 \mid y_i(\beta_0 + \beta^\top \mathbf{x}_i), \forall \mathbf{x}_i \in \text{Tr} \right\}.$$

This constrained quadratic problem (QP) can be solved by Lagrange multipliers (in implementations, it is solved numerically), but a key observation is that it is possible to rewrite

$$\beta = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i, \quad \text{with} \quad \sum_{i=1}^N \alpha_i y_i = 0$$

thanks to the **representer theorem**.<sup>35</sup>

The original QP becomes

$$\arg \min_{(\beta, \beta_0)} \left\{ \frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j \mathbf{x}_i^\top \mathbf{x}_j - \sum_{i=1}^N \alpha_i \mid \sum_{i=1}^N \alpha_i y_i = 0, \forall \mathbf{x}_i, \mathbf{x}_j \in \text{Tr} \right\}.$$

Ultimately, it can be shown that all but  $L$  of the coefficients  $\alpha_i$  are 0, typically,  $L \ll N$ . The **support vectors** are those training observations  $\mathbf{x}_{i_k}$ ,  $k = 1, \dots, L$ , for which  $\alpha_{i_k} \neq 0$ . The **decision function** is defined by

$$T(\mathbf{x}; \alpha) = \sum_{k=1}^L \alpha_{i_k} y_{i_k} \mathbf{x}_{i_k}^\top \mathbf{x} + \beta_0,$$

scaled so that  $T(\mathbf{x}_{i_k}; \alpha) = y_{i_k} = \pm 1$  for each support vector  $\mathbf{x}_{i_k}$ .

The **class assignment** for any  $\mathbf{x} \in \text{Te}$  is thus

$$\text{class}(\mathbf{x}) = \begin{cases} +1 & \text{if } T(\mathbf{x}; \alpha) \geq 0 \\ -1 & \text{if } T(\mathbf{x}; \alpha) < 0 \end{cases}$$

In practice (especially when  $N < p$ ), the data is rarely linearly separable into distinct classes (as below, for instance).

Additionally, even when the classes are linearly separable, the data may be **noisy**, which could lead to overfitting, with technically optimal but practically sub-optimal maximal margin solutions (see [168] for examples).

In applications, **support vector classifiers** optimize instead a **soft margin**, one for which some misclassifications are permitted (as in Figure 21.15).

The soft margin problem can be written as

$$\arg \min_{(\beta, \beta_0)} \left\{ \frac{1}{2} \beta^\top \beta \mid y_i(\beta_0 + \beta^\top \mathbf{x}_i) \geq 1 - \varepsilon_i, \varepsilon_i \geq 0, \forall \mathbf{x}_i \in \text{Tr}, \|\varepsilon\| < C \right\},$$

where  $C$  is a (budget) **tuning parameter**,  $\varepsilon$  is a vector of slack variables, canonically scaled so that  $|F(\mathbf{x}^*)| = |\beta_0 + \beta^\top \mathbf{x}^*| = 1$  for any eventual

35: Technically speaking we do not need to invoke the representer theorem in the linear separable case. At any rate, the result is out-of-scope for this document.

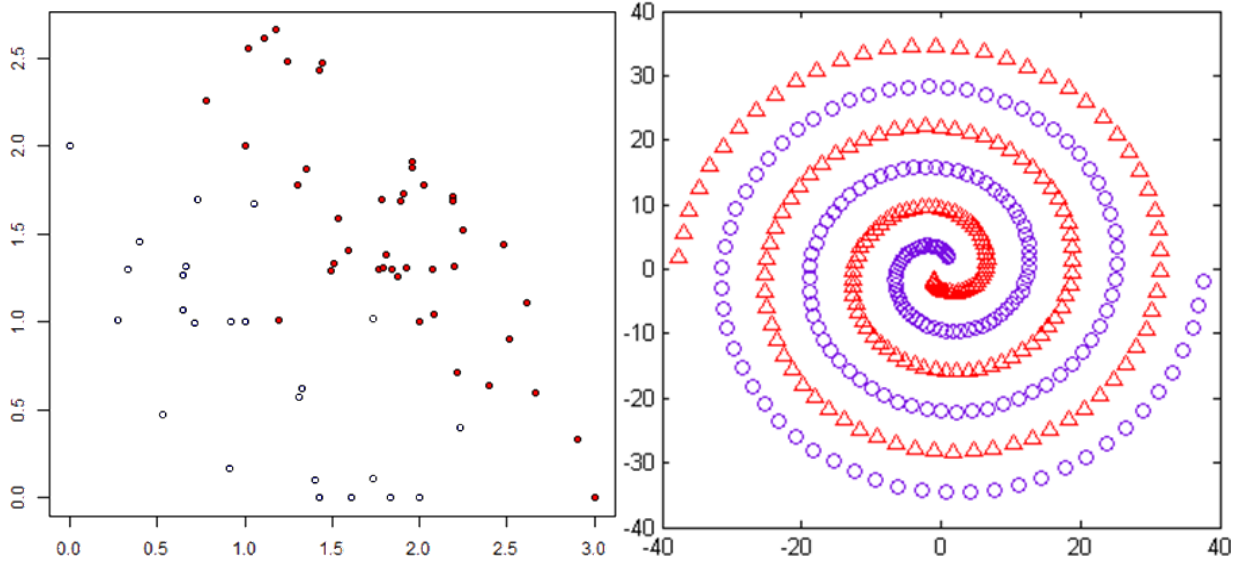


Figure 21.14: Non-linearly separable two-class datasets.

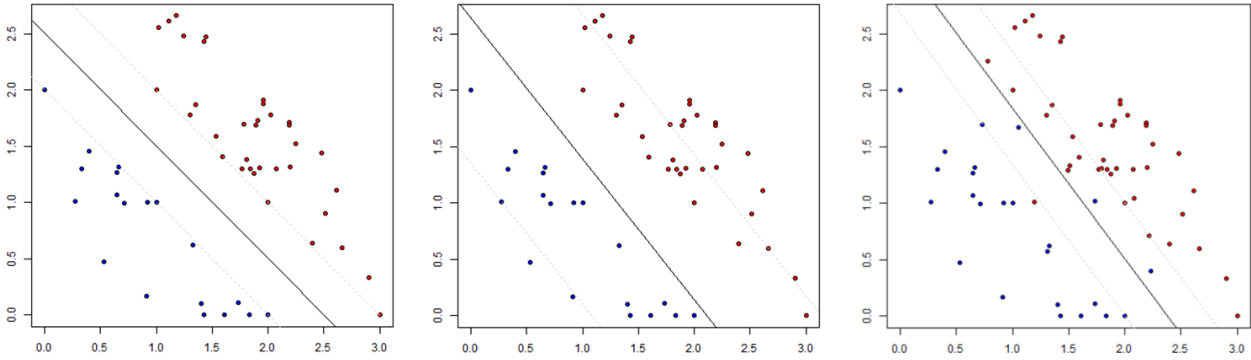


Figure 21.15: Hard margin for a linearly separable classifier (left); soft margin for a linearly separable classifier (middle); soft margin for a non-linearly separable classifier (right).

support vector  $\mathbf{x}^*$ .

Such a model offers greater robustness against unusual observations, while still classifying most training observations correctly:

36: It falls on the correct side of the hyper-plane, and outside the maximum margin.

37: It falls on the correct side of the hyper-plane, but within the margin.

- if  $\varepsilon_i = 0$ , then  $\mathbf{x}_i \in \text{Tr}$  is **correctly classified**,<sup>36</sup>
- if  $0 < \varepsilon_i < 1$ , then  $\mathbf{x}_i \in \text{Tr}$  is **acceptably classified**,<sup>37</sup>
- if  $\varepsilon_i \geq 1$ , it is **incorrectly classified**.

If  $C = 0$ , then no violations are allowed ( $\|\varepsilon\| = 0$ ) and the problem reduces to the hard margin SVM classifier; a solution may not even exist if the data is not linearly separable.

If  $C > 0$  is an integer, no more than  $C$  training observations can be misclassified; indeed, if  $i_1, \dots, i_C$  are the misclassified indices, then  $\varepsilon_{i_1}, \dots, \varepsilon_{i_C} \geq 1$  and

$$C \geq \sum_{i=1}^N \varepsilon_i \geq \sum_{k=1}^C \varepsilon_{i_k} \geq C.$$

As  $C$  increases, tolerance for violations also increases, as does the width of the soft margin;  $C$  plays the role of a **regularization parameter**, and is usually selected *via* cross-validation.

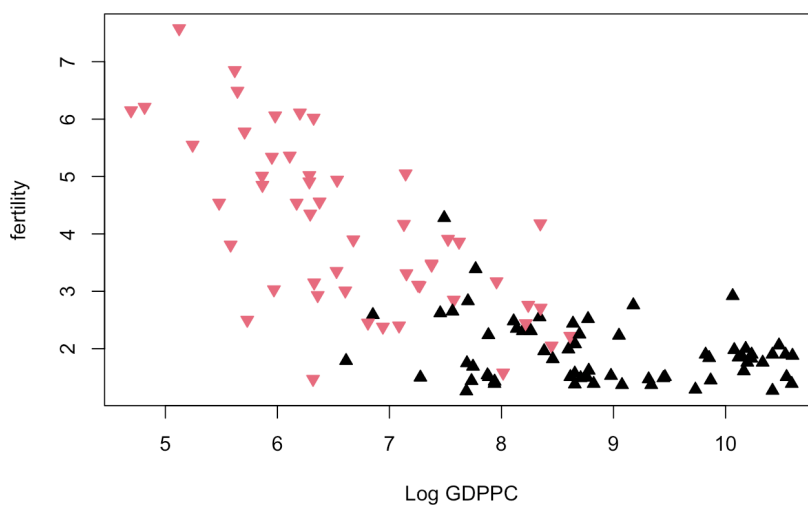
Low values of  $C$  are associated with harder margins, which leads to low bias but high variance (a small change in the data could create qualitatively different margins); large values of  $C$  are associated with wider (softer) margins, leading to more potential misclassifications and higher bias, but also lower variance as small changes in the data are unlikely to change the margin significantly.

We can build a classifier through the representer theorem formulation as before, the only difference being that the **decision function**  $T(\mathbf{x}; \boldsymbol{\alpha})$  is scaled so that  $|T(\mathbf{x}_{i_k}; \boldsymbol{\alpha})| \geq 1 - \varepsilon_{i_k}$  for every support vector  $\mathbf{x}_{i_k}$ . It is difficult to determine what the value of the regularization parameter  $C$  should be at first glance; an optimal value can be obtained *via* a **tuning process**, which tries out various values and identifies the one that produces an optimal model.

**Example** We train a SVM with  $C = 0.1$  (obtained *via* a tuning procedure for  $C$ ) for the 2011 Gapminder dataset to predict the life expectancy class  $Y$  in terms of the fertility rate  $X_1$  and the logarithm of GDP per capita  $X_2$ ;  $n = 116$  observations are used in the training set `gapminder.2011.tr`, the rest are set aside in the test set `gapminder.2011.te`.

```
set.seed(0)
ind.train = sample(nrow(gapminder.2011),
                  round(0.7*nrow(gapminder.2011)),
                  replace=FALSE)
gapminder.2011.tr = gapminder.2011[ind.train,]
gapminder.2011.te = gapminder.2011[-ind.train,]

x <- gapminder.2011.tr[,c("fertility", "gdp", "population")]
w <- log(x[,2]/x[,3])
x <- data.frame(x[,1], w)
y <- gapminder.2011.tr[,c("LE")]
dat = data.frame(x, y)
plot(w, x[,1], col=y, bg=y, pch=(as.numeric(y)+23),
     xlab="Log GDPPC", ylab="fertility")
```



The red triangles represent countries with low life expectancy; the black ones, countries with high life expectancy. Notice the class overlap in the training data.

We run 7 linear SVM models with various cost parameters (through e1071's `tune()` function), the optimal model has  $C = 0.1$ .

```
library(e1071)
tuned.model <- tune(svm, y~., data = dat, kernel = "linear",
  ranges = list(cost = c(0.001, 0.01, 0.1,
    1, 5, 10, 100)))
(best.mod <- tuned.model$best.model)
```

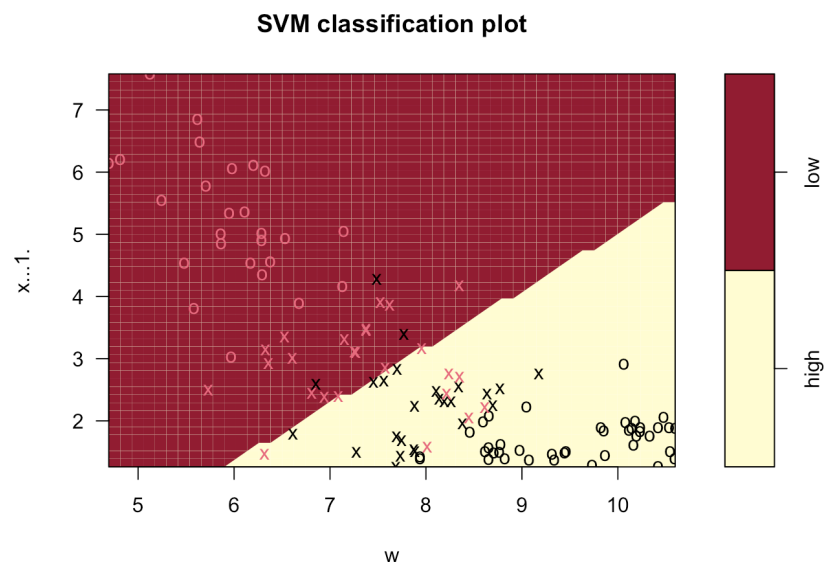
Parameters:

```
SVM-Type: C-classification
SVM-Kernel: linear
cost: 0.1
```

Number of Support Vectors: 50

The corresponding SVM model is obtained *via* the `svm()` function (note the parameters). The SVM decision boundary is shown below:

```
svmfit <- svm(y~., data = dat, kernel = "linear", cost=0.1)
plot(svmfit, dat, main="Linear Kernel")
```



We can evaluate the model's performance on `Te(dat.te)`; the confusion matrix of the model on the test set is:

```
x <- gapminder.2011.te[,c("fertility", "gdp", "population")]
w <- log(x[,2]/x[,3])
x <- data.frame(x[,1], w)
y <- gapminder.2011.te[,c("LE")]

# Test data
```

```

dat.te = data.frame(x,y)
# Class prediction on test data
results = predict(svmfit,dat.te)
# Confusion matrix
table(actual=gapminder.2011.te$LE,pred=results)

```

$\alpha = 0.5$		prediction	
		0	1
actual	0	22	10
	1	1	17

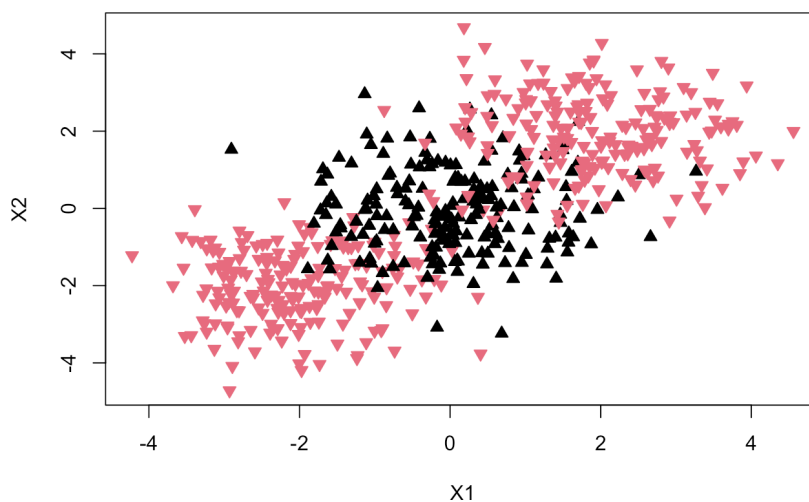
It is not a perfectly accurate model, but it is certainly acceptable given the class overlap in Tr.

**Nonlinear Boundaries** If the boundary between two classes is linear, the SVM classifier of the previous section is a natural way to attempt to separate the classes. In practice, however, the classes are rarely so cleanly separated, as below, say.

```

set.seed(0)
x <- matrix(rnorm(600*2), ncol = 2)
y <- c(rep(-1,200), rep(0,200),rep(1,200))
x[y==1,] <- x[y==1,] + 2
x[y==-1] <- x[y==-1,] - 2
y <- y^2
dat <- data.frame(x=x, y=as.factor(y))
plot(dat[,1], dat[,2], col=dat[,3], bg=dat[,3],
      pch=(as.numeric(dat[,3])+23), xlab="X1", ylab="X2")

```



In both the hard and the soft margin support vector classifiers, the function to optimize takes the form

$$\frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j \mathbf{x}_i^\top \mathbf{x}_j - \sum_{i=1}^N \alpha_i,$$

and the decision function, the form

$$T(\mathbf{x}; \boldsymbol{\alpha}) = \sum_{k=1}^L \alpha_{i_k} y_{i_k} \mathbf{x}_{i_k}^\top \mathbf{x} + \beta_0.$$

However, we do not actually need to know the support vectors  $\mathbf{x}_{i_k}$  (or even the observations  $\mathbf{x}_i$ , for that matter) in order to compute the decision function values – it is sufficient to have access to the **inner products**  $\mathbf{x}_i^\top \mathbf{x}_j$  or  $\mathbf{x}_{i_k}^\top \mathbf{x}$ , which are usually denoted by  $\langle \mathbf{x}_{i_k}, \mathbf{x} \rangle$  or  $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$ .

The objective function and the decision function can thus be written as

$$\frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle - \sum_{i=1}^N \alpha_i, \quad T(\mathbf{x}; \boldsymbol{\alpha}) = \sum_{k=1}^L \alpha_{i_k} y_{i_k} \langle \mathbf{x}_{i_k}, \mathbf{x} \rangle + \beta_0.$$

This seemingly innocuous remark opens the door to the **kernel approach**; we could conceivably replace the inner products  $\langle \mathbf{x}, \mathbf{w} \rangle$  by generalized inner products  $K(\mathbf{x}, \mathbf{w})$ , which provide a measure of similarity between the observations  $\mathbf{x}$  and  $\mathbf{w}$ .

Formally, a **kernel** is a symmetric (semi-)positive definite operator  $K : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}_0^+$ .<sup>38</sup> Common statistical learning kernels include:

- **linear** –  $K(\mathbf{x}, \mathbf{w}) = \mathbf{x}^\top \mathbf{w}$ ;
- **polynomial of degree  $d$**  –  $K_d(\mathbf{x}, \mathbf{w}) = (1 + \mathbf{x}^\top \mathbf{w})^d$ ;
- **Gaussian** (or radial) –  $K_\gamma(\mathbf{x}, \mathbf{w}) = \exp(-\gamma \|\mathbf{x} - \mathbf{w}\|_2^2)$ ,  $\gamma > 0$ ;
- **sigmoid** –  $K_{\kappa, \delta}(\mathbf{x}, \mathbf{w}) = \tanh(\kappa \mathbf{x}^\top \mathbf{w} - \delta)$ , for allowable  $\kappa, \delta$ .

For instance, a linear kernel SVM and a radial kernel SVM with  $\gamma = 1$ ,  $C = 0.5$  yield the following classifications on the previous dataset.<sup>39</sup>

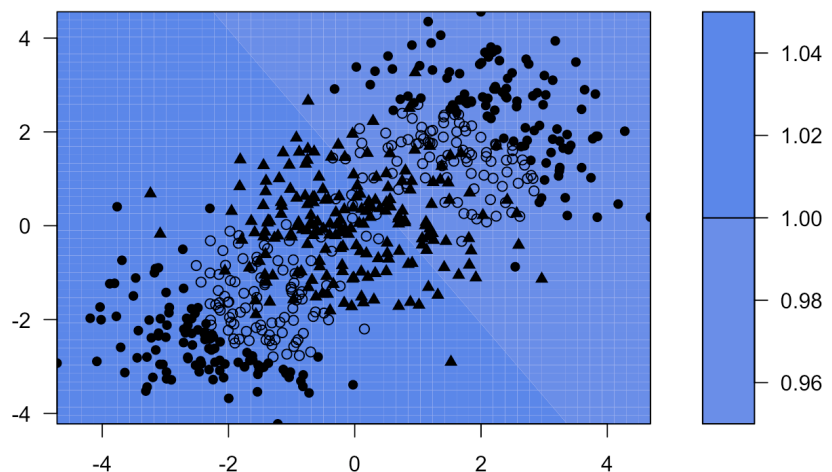
38: By analogy with positive definite square matrices, this means that  $\sum_{i,j=1}^N c_i c_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0 \forall \mathbf{x}_i \in \mathbb{R}^p, c_j \geq 0$ .

39: We are using kernlab's `ksvm()` function and display the linear SVM output for comparison, whose performance we expect to be crap-tastic

```
library(kernlab)

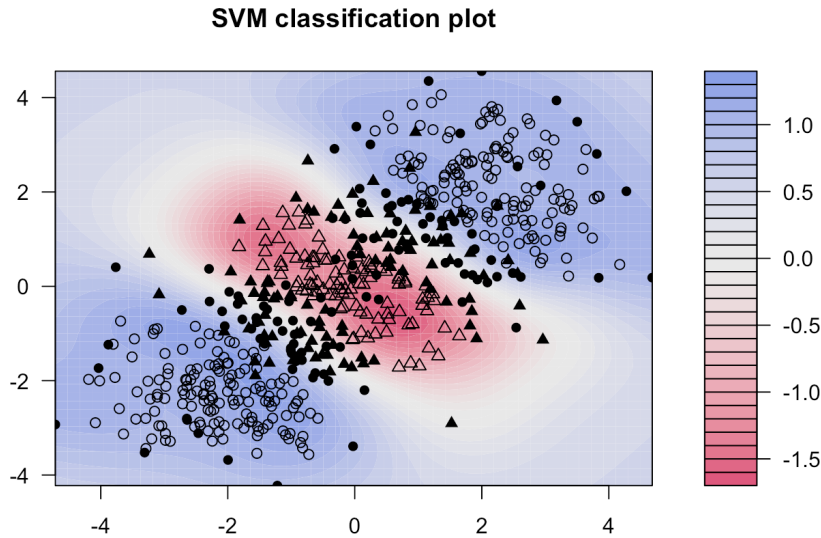
# linear SVM
kernfit.lin <- ksvm(x,y, type = "C-svc",
                   kernel = 'vanilladot', C = 10)
kernlab::plot(kernfit.lin, data=x)
```

SVM classification plot



Not that great, to be honest...

```
# Gaussian SVM
kernfit.rbfc <- ksvm(x,y, type = "C-svc", kernel = 'rbfdot',
                    sigma=1, C = 0.5)
kernlab::plot(kernfit.rbfc, data=x)
```



How is the decision boundary computed? The principle is the same as with linear SVM: the objective function and the decision function are

$$\frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j) - \sum_{i=1}^N \alpha_i, \quad T(\mathbf{x}; \boldsymbol{\alpha}) = \sum_{k=1}^L \alpha_{i_k} y_{i_k} K(\mathbf{x}_{i_k}, \mathbf{x}) + \beta_0.$$

For the radial kernel, for instance, if a test observation  $\mathbf{x}$  is near a training observation  $\mathbf{x}_i$ , then  $\|\mathbf{x} - \mathbf{x}_i\|_2^2$  is small and  $K_\gamma(\mathbf{x}, \mathbf{x}_i) \approx 1$ ; if they are far from one another, then  $\|\mathbf{x} - \mathbf{x}_i\|_2^2$  is large and  $K_\gamma(\mathbf{x}, \mathbf{x}_i) \approx 0$ .

In other words, in the radial kernel framework, only those observations close to a test observation play a role in class prediction.

**Kernel Trick** But why even use kernels in the first place? While the linear kernel is easier to interpret and implement, not all data sets are linearly separable, as we have just seen. Consider the toy classification problem on the left of Figure 21.16 (adapted from an unknown online source).

The optimal margin separating “strip” is obviously not linear. One way out of this problem is to introduce a **transformation**  $\Phi$  from the original  $X$ -feature space to a higher-dimensional (or at least, of the same dimension)  $Z$ -feature space in which the data is linearly separable, and to build a linear SVM on the transformed training observations  $\mathbf{z}_i = \Phi(\mathbf{x})_i$ .<sup>40</sup>

In this example, we use some  $\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ ; the projection of the transformation into the  $Z_1 Z_3$ -plane could be as in Figure 21.16 (right).

40: This might seem to go against reduction strategies used to counter the curse of dimensionality; the added dimensions are needed to “unfurl” the data, so to speak.

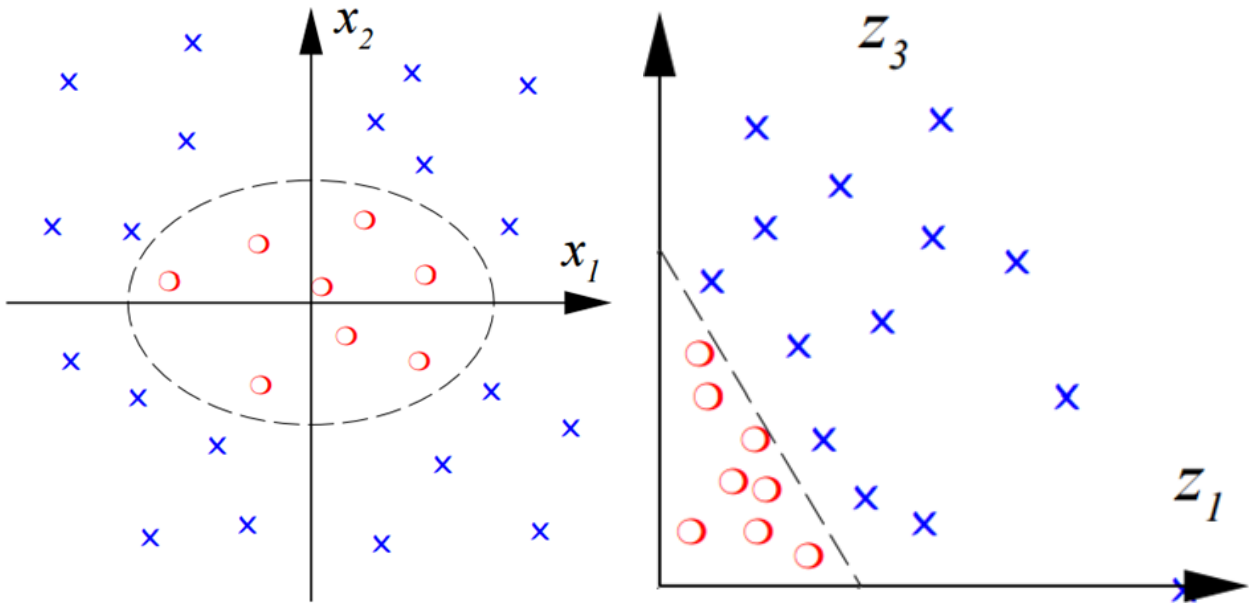


Figure 21.16: Toy classification problem (left); corresponding projection of the linear problem in  $Z$ -space [author unknown].

The objective function and the decision function take the form

$$\frac{1}{2} \sum_{i,j=1}^N \alpha_i \alpha_j \Phi(\mathbf{x}_i)^\top \Phi(\mathbf{x}_j) - \sum_{i=1}^N \alpha_i,$$

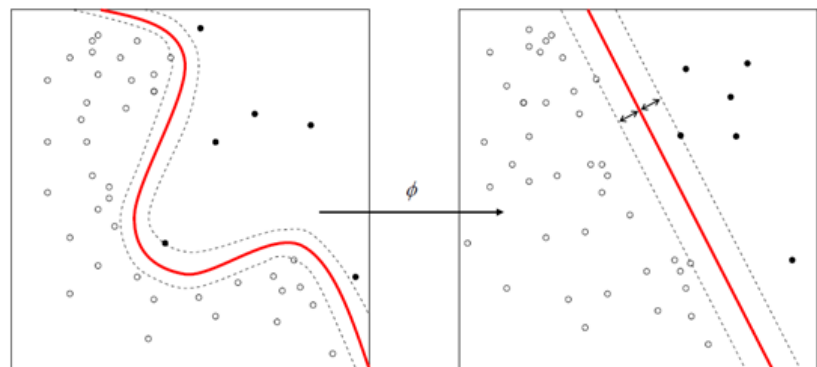
$$T(\mathbf{x}; \boldsymbol{\alpha}) = \sum_{k=1}^L \alpha_{i_k} y_{i_k} \Phi(\mathbf{x}_{i_k})^\top \Phi(\mathbf{x}) + \beta_0,$$

and the linear SVM is built as before (but in  $Z$ -space, not in  $X$ -space).

It sounds straightforward, but it does take a fair amount of experience to recognize that one way to separate the data is to use

$$\mathbf{z} = \Phi(\mathbf{x}) = (x_1^2, \sqrt{2}x_1x_2, x_2^2).$$

And this is one of the easy transformations: what should be used in the case below (image taken from Wikipedia)?



The **kernel trick** simply states that  $\Phi$  can remain unspecified if we replace  $\Phi(\mathbf{x})^\top \Phi(\mathbf{w})$  by a “reasonable” (often radial) kernel  $K(\mathbf{x}, \mathbf{w})$ .



**General Classification** What do we do if the response variable has  $K > 2$  classes? In the **one-versus-all** (OVA) approach, we fit  $K$  different 2-class SVM decision functions  $T_k(\mathbf{x}; \boldsymbol{\alpha})$ ,  $k = 1, \dots, K$ ; in each, one class versus the rest. The test observation  $\mathbf{x}^*$  is assigned to the class for which  $T_k(\mathbf{x}^*; \boldsymbol{\alpha})$  is largest.

In the **one-versus-one** (OVO) approach, we fit all  $\binom{K}{2}$  pairwise 2-class SVM classifiers  $\text{class}_{k,\ell}(\mathbf{x})$ , for training observations with levels  $k, \ell$ , where  $k > \ell = 1, \dots, K - 1$ . The test observation  $\mathbf{x}^*$  is assigned to the class that wins the most pairwise “competitions”.

If  $K$  is large,  $\binom{K}{2}$  might be too large to make OVO computationally efficient; when it is small enough, OVO is the recommended approach.

**Example** The vowel dataset was taken from the *openML website* [41](#). This modified version, by Turney, is based on Robinson’s *Determining Vowel Recognition Data*, which is a speaker-independent recognition of the eleven steady state vowels of British English using a specified training set of lpc-derived log area ratios.<sup>41</sup>

We start by reading in the data and summarizing it – the dataset has  $n = 990$  observations and  $p = 14$  variables.

```
vowel <- read.csv("datasets-uci-vowel.csv", header=TRUE,
                  sep=";", stringsAsFactors=TRUE)
str(vowel)
```

```
'data.frame': 990 obs. of 14 variables:
 $ Train.or.Test: Factor w/ 2 levels "Test","Train": 2 2 2 2 2 2 2 2 2 2 ...
 $ Speaker.Name : Factor w/ 15 levels "Andrew","Bill",...: 1 1 1 1 1 1 1 1 1 1 ...
 $ Speaker.Sex : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
 $ Feature.0 : num -3.64 -3.33 -2.12 -2.29 -2.6 ...
 $ Feature.1 : num 0.418 0.496 0.894 1.809 1.938 ...
 $ Feature.2 : num -0.67 -0.694 -1.576 -1.498 -0.846 ...
 $ Feature.3 : num 1.779 1.365 0.147 1.012 1.062 ...
 $ Feature.4 : num -0.168 -0.265 -0.707 -1.053 -1.633 ...
 $ Feature.5 : num 1.627 1.933 1.559 1.06 0.764 ...
 $ Feature.6 : num -0.388 -0.363 -0.579 -0.567 0.394 0.217 0.322 -0.435 -0.512 -0.466 ...
 $ Feature.7 : num 0.529 0.51 0.676 0.235 -0.15 -0.246 0.45 0.992 0.928 0.702 ...
 $ Feature.8 : num -0.874 -0.621 -0.809 -0.091 0.277 0.238 0.377 0.575 -0.167 0.06 ...
 $ Feature.9 : num -0.814 -0.488 -0.049 -0.795 -0.396 -0.365 -0.366 -0.301 -0.434 -0.836 ...
 $ Class : Factor w/ 11 levels "had","hAd","hed",...: 5 6 4 2 11 1 8 7 10 9 ...
```

41: **Real talk:** we don’t actually know what any of that means. But does it matter? Yes, any conclusion we can draw from this dataset will need to be scrutinized by subject matter experts before we can hope to apply them to real-world situations. On the other hand, data is simply marks on paper (or perhaps electromagnetic patterns on the cloud). We can analyze the data without really knowing what the underlying meaning is. The latter approach is usually sterile, but we can always use it to illustrate basic concepts.

There is some imbalance in the training/testing set-up (especially as it relates to the speaker sex):

```
table(vowel$Train.or.Test, vowel$Speaker.Sex)
```

	Female	Male
Test	198	264
Train	264	264

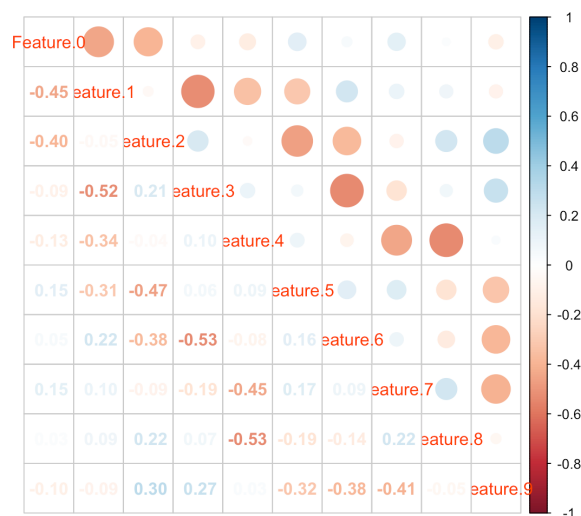
All-in-all, the numerical features seem to be generated from a multivariate normal distribution, with mean vector:

```
colMeans(vowel[,c(4:13)], dims = 1)
```

Feature.0	Feature.1	Feature.2	Feature.3	Feature.4
-3.203740404	1.881763636	-0.507769697	0.515482828	-0.305657576
Feature.5	Feature.6	Feature.7	Feature.8	Feature.9
0.630244444	-0.004364646	0.336552525	-0.302975758	-0.071339394

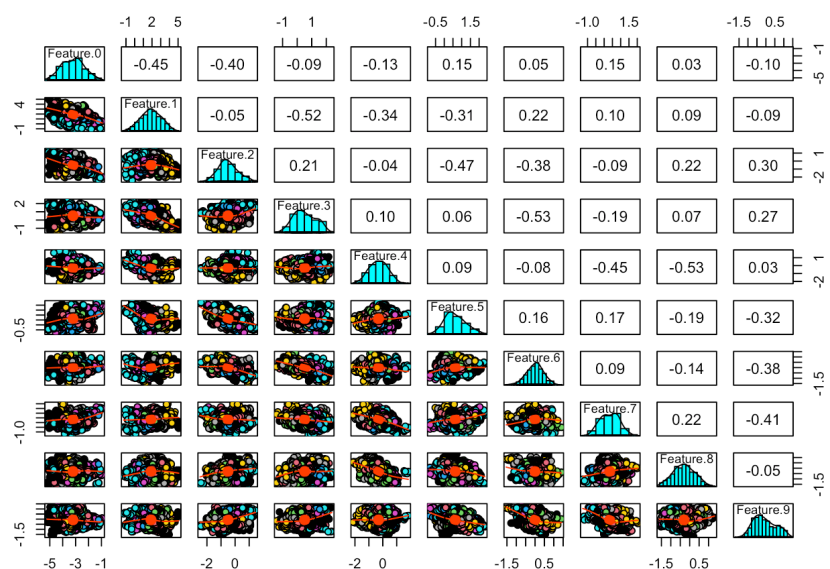
and correlation matrix:

```
corrplot::corrplot.mixed(cor(vowel[,c(4:13)]))
```



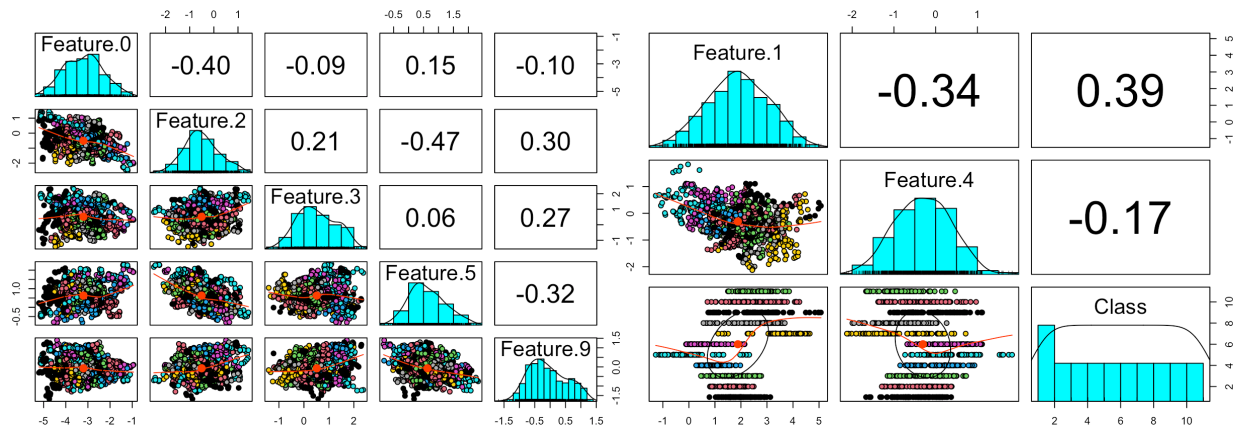
Can we get any information from the paired plots?

```
psych::pairs.panels(vowel[,4:13], pch = 21, bg = vowel$Class)
```



Perhaps if we focus only on certain variables?

```
library(psych)
pairs.panels(vowel[,c(4,6,7,9,13)], pch = 21, bg = vowel$Class)
pairs.panels(vowel[,c(5,8,14)], pch = 21, bg = vowel$Class)
```



The response variable is the `Class`, with 11 levels, and the  $p = 10$  predictors are `Feature.0`,  $\dots$ , `Feature.9`. We train an SVM model on a subset of the vowel dataset. In this instance, we use the training/testing split provided with the data (`Train.or.Test`), but any randomly selected split would be appropriate.

```
training = vowel[vowel$Train.or.Test=="Train",4:14]
testing = vowel[vowel$Train.or.Test=="Test",4:14]
c(nrow(training),nrow(testing)) # training/testing split
```

```
[1] 528 462
```

We use the support vector machine implementation found in the R library `e1071`.

First we tune the hyper-parameters on a subsample of the training data by using the `tune()` function, which selects optimal parameters by carrying out a grid search over the specified parameters (otherwise we might spend a lot of time trying to find a good combination of parameters).

For C-classification with a Gaussian kernel, the parameters are

- $C$ , the cost of constraint violation (which controls the penalty paid by the SVM model for misclassifying a training point), and
- $\gamma$ , the parameter of the Gaussian kernel (used to handle non-linear classification).

If  $C$  is “high”, then misclassification is costly, and *vice-versa*. If  $\gamma$  is “high”, then the Gaussian bump around the points are narrow, and *vice-versa*. Let us run a grid search with  $C$  varying from 0.1 to 100 by powers of 10, and  $\gamma = 0.5, 1, 2$ .

```
vowel.svm.tune.1 <- e1071::tune(e1071::svm,
                              train.x=training[,1:10],
                              train.y=training[,11],
                              kernel="radial",
                              ranges=list(cost=10^(-1:2), gamma=c(.5,1,2)))
print(vowel.svm.tune.1)
```

Parameter tuning of 'e1071::svm':

- sampling method: 10-fold cross validation
- best parameters:
 

cost	gamma
10	0.5
- best performance: 0.007619739

The minimal misclassification error (best performance) in this run is reached when the best parameters have the values listed in the output above. Obviously, that search was fairly coarse: searching at a finer level can be very demanding, time-wise.

For comparison's sake, let us see if tuning with finer intervals and larger ranges gives substantially different results.

```
vowel.svm.tune.2 <- e1071::tune(e1071::svm,
                              train.x=training[,1:10],
                              train.y=training[,11],
                              kernel="radial",
                              ranges=list(cost=10^(-2:2), gamma=1:20*0.1))
print(vowel.svm.tune.2)
```

Parameter tuning of 'e1071::svm':

- sampling method: 10-fold cross validation
- best parameters:
 

cost	gamma
10	0.8
- best performance: 0.003773585

The optimal parameters are sensibly the same, so we might as well stick with the optimal parameters values from the first tuning. Training the model with these values yields:

```
vowel.svm.model = e1071::svm(training[,11] ~ ., data = training,
                             type="C-classification",
                             cost=10, kernel="radial", gamma=0.5)
summary(vowel.svm.model)
```

Parameters:

SVM-Type: C-classification  
SVM-Kernel: radial  
cost: 10

Number of Support Vectors: 351

( 27 32 32 26 30 36 37 29 40 32 30 )

Number of Classes: 11

Levels:

had hAd hed hEd hid hId hod hOd hud hUd hYd

Note the number of support vectors. How accurately can this model predict the class of new observations?

```
predicted = predict(vowel.svm.model, testing)
(confusion.matrix = table(pred = predicted, true = testing[,11]))
e1071::classAgreement(confusion.matrix, match.names=TRUE)
```

	true										
pred	had	hAd	hed	hEd	hid	hId	hod	hOd	hud	hUd	hYd
had	36	0	0	0	0	0	0	0	0	0	0
hAd	0	40	0	0	0	0	0	0	0	0	0
hed	0	0	42	0	0	0	0	0	0	0	0
hEd	0	0	0	37	0	0	0	0	0	0	0
hid	0	0	0	0	39	0	0	0	0	0	0
hId	0	0	0	0	2	42	0	0	1	0	0
hod	0	0	0	0	0	0	36	0	0	0	0
hOd	0	0	0	0	0	0	0	35	0	0	0
hud	0	0	0	0	0	0	0	0	23	0	0
hUd	6	2	0	5	1	0	6	7	18	42	16
hYd	0	0	0	0	0	0	0	0	0	0	26

\$diag

[1] 0.8614719

\$kappa

[1] 0.847619

\$rand

[1] 0.9425585

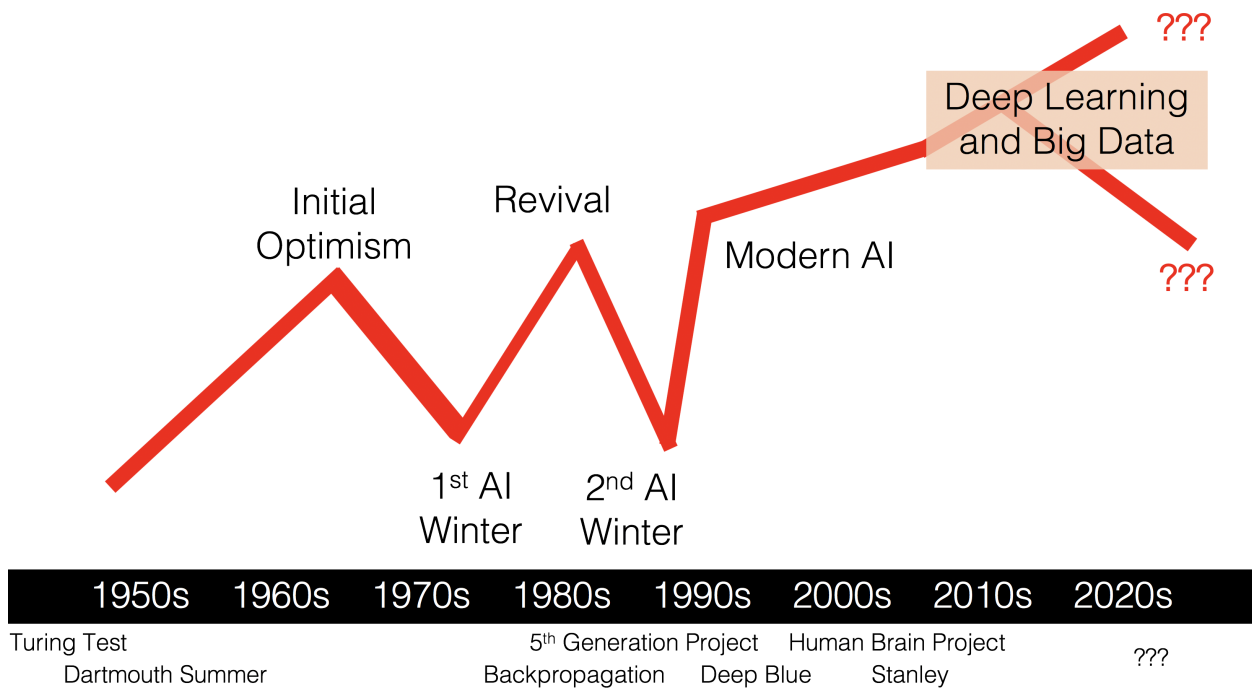
\$crand

[1] 0.6798992

What do you think?

**Final Comments** In practice, it is not always obvious whether one should use SVM, logistic regression, linear discriminant analysis (LDA), decision trees, etc.<sup>42</sup>

42: In Section 21.5, we argue that it is usually preferable to train a variety of models, rather than just the one.



**Figure 21.17:** Conceptual timeline of the interest and optimism regarding AI; important milestones are indicated below the dates.

43: The actual values of  $T(\mathbf{x}; \boldsymbol{\alpha})$  have no intrinsic meaning, other than their relative ordering.

- if classes are (nearly) separable, SVM and LDA are usually preferable to logistic regression;
- otherwise, using logistic regression together with a ridge penalty (see Section 20.2, *Shrinkage Methods*) is roughly equivalent to using SVM;
- if the aim is to estimate class membership probabilities, it is preferable to use logistic regression as SVM is not **calibrated**;<sup>43</sup>
- it is possible to use kernels in the logistic regression and LDA frameworks, but at the cost of increased computational complexity.

All in all, it remains crucial to understand that the *No Free Lunch Theorem* remains in effect [215, 216, 235]. There is simply no magical recipe... although the next technique we discuss is often viewed (and used) as one.

### 21.4.3 Artificial Neural Networks

When practitioners discuss using **Artificial Intelligence** (AI) techniques [166] to solve a problem, the implicit assumption is often (but not always) that a neural network (or some other variant of **deep learning**) will be used, and for good reason: “neural networks blow all previous techniques out of the water in terms of performance” [244]. But there are some skeletons in the closet: “[...] given the existence of adversarial examples, it shows we really don’t understand what’s going on” [244].

At various times since Turing’s seminal 1950 paper (in which he proposed the celebrated **Imitation Game** [245]), complete artificial intelligence has been announced to be “just around the corner” (see Figure 21.17).

With the advent of **deep learning** and Big Data processing, optimism is as high as it’s ever been, but opinions on the topic are varied – to some

commentators, AI is a brilliant success, while to others it is a spectacular failure (see the headlines in Section 14.1.3). So what is really going on?

It is far from trivial to identify the **essential qualities and skills of an intelligence**. There have been multiple attempts to solve the problem by building on Turing's original effort. An early argument by Hofstadter [86] is that any intelligence should:

- provide flexible responses in various scenarios;
- take advantage of lucky circumstances;
- make sense out of contradictory messages;
- recognize the relative importance of a situation's elements;
- find similarities between different situations;
- draw distinctions between similar situations, and
- come up with new ideas from scratch or by re-arranging previous known concepts.

This is not quite the approach taken by modern AI researchers, which define the discipline as the study of **intelligent agents** – any device that perceives its environment and takes actions to maximize its chance of success at some task/goal [246].

Examples include:

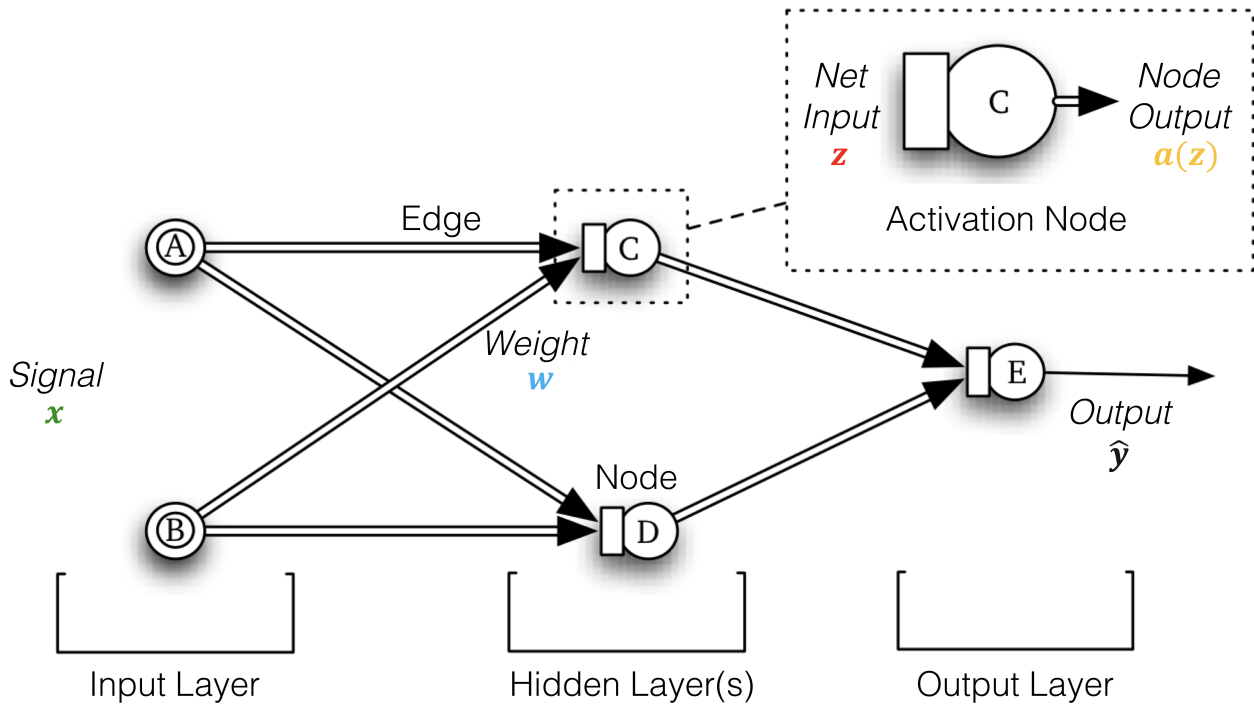
- **expert systems** – TurboTax, WebMD, technical support, insurance claim processing, air traffic control, etc.;
- **decision-making** – Deep Blue, auto-pilot systems, “smart” meters, etc.;
- **natural Language Processing** – machine translation, Siri, named-entity recognition, chatGPT, etc.;
- **recommenders** – Google, Expedia, Facebook, LinkedIn, Netflix, Amazon, etc.;
- **content generators** – music composer, novel writer, animation creator, etc.;
- **classifiers** – facial recognition, object identification, fraud detection, etc.

A trained **artificial neural network** (ANN) is a function that maps inputs to outputs in a useful way: it uses a Swiss-army-knife approach to providing outputs – plenty of options are available in the **architecture**, but it's not always clear which ones should be used.

One of the reasons that ANNs are so popular is that the user does not need to decide much about the function or know much about the problem space in advance – ANNs are **quiet models**.

Algorithms allow ANNs to **learn** (i.e. to generate the function and its internal values) automatically; technically, the only requirement is the user's ability to minimize a cost function (which is to say, to be able to solve optimization problems).

**Overview** The simplest definition of an **artificial neural network** is provided by the inventor of one of the first neuro-computers, R. Hecht-Nielsen, as:



**Figure 21.18:** Artificial neural network topology – conceptual example. The number of hidden layers is arbitrary, as is the size of the signal and output vectors.

“[...] a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs. [247]”

An **artificial neural network** is an interconnected group of nodes, inspired by a simplification of neurons in a brain but on much smaller scales. Neural networks are typically organized in **layers**. Layers are made up of a number of interconnected **nodes** which contain an **activation function**.

A **pattern  $x$**  (input, signal) is presented to the network *via* the **input layer**, which communicates with one or more **hidden layers**, where the actual processing is done *via* a system of weighted **connections  $W$**  (edges).

The hidden layers then link to an **output layer**, which outputs the **predicted response  $\hat{y}$**  (see Figure 21.18).

**Neural Networks Architecture** In order to train a neural network, we need the following objects [237]:

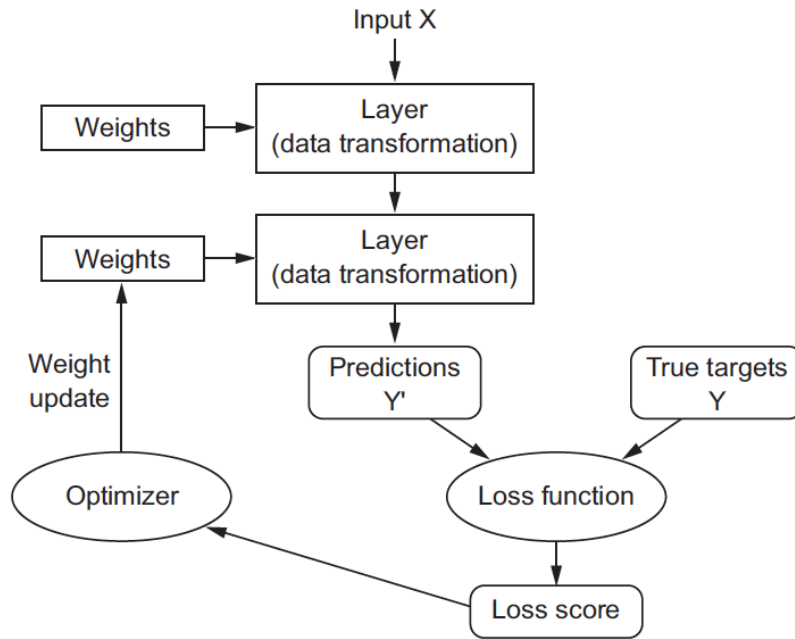
- some **input data**,
- a number of **layers**,
- a **model**, and
- a **learning process** (loss function and optimizer).

The object interactions is visualized in Figure 21.19.

A network (model), which is composed of layers that are chained together, maps the input data into predictions.<sup>44</sup> The loss function then compares these predictions to the targets, producing a loss value: a measure of how

44: In essence, a neural network is a **function**.





**Figure 21.19:** Relationship between the network, layers, loss function, and optimizer [237].

well the network's predictions match what was expected. The optimizer uses this loss value to update the network's weights.

**Input Data** Neural networks start with the **input training data** (and corresponding **targets**) in the form of a **tensor**. Generally speaking, most modern machine learning systems use tensors as their basic data structure. At its core, a tensor is a **container** for data – and it is almost always numerical.

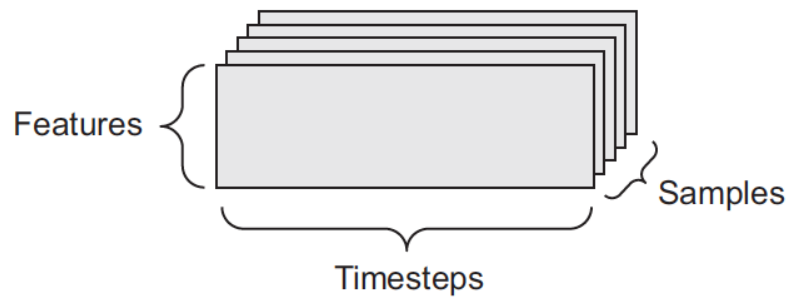
Tensors are defined by three key attributes: their

- **rank** (number of axes) – for instance, a 3D tensor has three axes, while a matrix (2D tensor) has two axes;
- **shape**, a tuple of integers that describes how many dimensions the tensor has along each axis – for instance, a matrix's shape is described using two elements, such as (3, 5), a 3D tensor's shape has three elements, such as (3, 5, 5), a vector (1D tensor)'s shape is given by a single element, such as (5), whereas a scalar has an empty shape, ( );
- **data type** – for instance, a tensor's type could be float32, uint8, float64, etc.

Data tensors almost always fall into one of the following categories:

- the most common case is **vector data**; in such datasets, each single data point can be encoded as a vector, and a batch of data will be encoded as a matrix or 2D tensor of shape (#samples, #features), or more simply, as an array of vectors where the first axis is the samples axis and the second axis is the features axis;
- **time series or sequence data**, whenever the passage of time is crucial to the observations in the dataset (or the notion of sequence order), can be stored in a 3D tensor with an explicit time axis; each sample can be encoded as a sequence of vectors (a 2D tensor), and

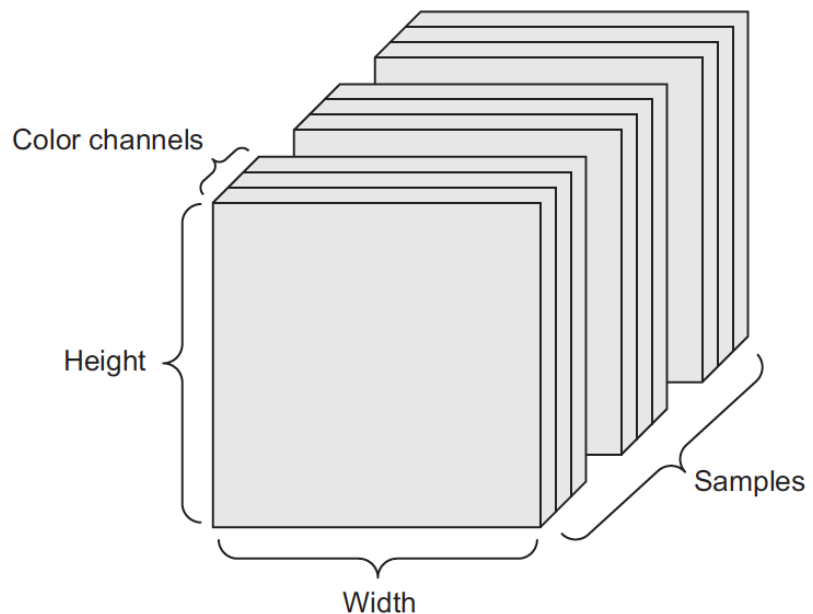
a batch of data will be encoded as a 3D tensor of shape ( $\#samples$ ,  $\#timesteps$ ,  $\#features$ ), as in Figure 21.20;



**Figure 21.20:** A 3D time series data tensor [237].

45: Although grayscale images have only a single colour channel and could thus be stored in 2D tensors, by convention image tensors are always 3D, with a one-dimensional colour channel for grayscale images.

- **images** typically have three dimensions: height, width, and colour depth;<sup>45</sup> a batch of image data could thus be stored in a 4D tensor of shape ( $\#samples$ ,  $\#height$ ,  $\#width$ ,  $\#channels$ ), as in Figure 21.21;



**Figure 21.21:** A 4D image data tensor [237].

- **video data** is one of the few types of real-world data for which 5D tensors are needed – a video can be understood as a sequence of frames, each frame being a colour image; a sequence of frames can be stored in a 4D tensor ( $\#frames$ ,  $\#height$ ,  $\#width$ ,  $\#channels$ ), and so a batch of different videos can be stored in a 5D tensor of shape ( $\#samples$ ,  $\#frames$ ,  $\#height$ ,  $\#width$ ,  $\#channels$ ).

**Layers** The core building block of neural networks is the **layer**, a data-processing module that is, in a sense, a **filter** for data: some data goes into the layer and comes out in a more useful form.

Specifically, layers extract **representations** out of the data fed into them – hopefully, representations that are **more meaningful** for the problem at hand. A layer takes as input 1+ tensors and outputs 1+ tensors. Different

layers are appropriate for different tensor formats and different types of data processing.

For instance, simple vector data, stored in 2D tensors, is often processed by **densely connected** layers, also called **fully connected** or **dense** layers. Sequence data, stored in 3D tensors, is typically processed by **recurrent** layers. Image data, stored in 4D tensors, is usually processed by 2D **convolution** layers.

Most of deep learning consists of chaining together simple layers that will implement a form of **progressive data distillation**. However, to build deep learning models in tensor-based modules like Keras [237], it is important to clip together **compatible** layers to form useful data-transformation pipelines.

The notion of **layer compatibility** refers specifically to the fact that every layer can only accept input tensors of a certain shape and return output tensors of a certain shape.

We will discuss tensors in greater detail in Chapter ??.

**Model: Networks of Layers** An artificial neural network model is essentially a **data processing sieve**, made of a succession of increasingly refined data filters – the **layers**. The most common example of a model is a linear stack of layers, mapping a single input to a single output. Other network topologies include: **two-branch networks**, **multihead networks**, and **inception blocks**. The topology of a network defines a **hypothesis space**.

Since machine learning is basically

“[...] searching for useful representations of some input data, within a predefined space of possibilities, using guidance from a feedback signal [237],”

by choosing a network topology, we constrain the space of possibilities (hypothesis space) to a specific series of tensor operations, mapping input data to output data.

From a ML perspective, what we are searching for is a good set of values for the weight tensors involved in these tensor operations. Picking the right network architecture is more an art than a science; and although there are some best practices and principles we can rely on, practical experience is the main factor in becoming a proper neural network architect.

**Learning Process: Loss Function and Optimizer** After a network architecture has been selected, two other objects need to be chosen:

- the **(objective) loss function** is the quantity that is minimized during training – it represents a measure of success for the task at hand, and
- the **optimizer** determines how the network is updated based on the loss function.

In this context, **learning** means finding a combination of model parameters that minimizes the **loss function** for a given set of training data observations and their corresponding targets.

Learning happens by drawing random batches of data samples and their targets, and computing the **gradient** of the network parameters with respect to the **loss** on the batch. The network parameters are then updated by a small amount (the magnitude of the move is defined by the learning rate) in the opposite direction from the gradient.

The entire learning process is made possible by the fact that under a network disguise, neural networks are simply **chains of differentiable tensor operations**, to which it is possible to apply the chain rule of differentiation to find the gradient function mapping the current parameters and current batch of data to a gradient value.

Choosing the right objective function for a given problem is extremely important: the network is ruthless when it comes to lowering its loss function, and it will take any shortcut it can to achieve that objective. If the latter does not fully correlate with success for the task at hand, the network may face unintended side effects.

Simple guidelines exist to help analysts select an appropriate loss function for common problems such as classification, regression, and sequence prediction. We typically use:

- **binary cross entropy** for a binary classification;
- **categorical cross entropy** for a  $n$ -ary classification;
- **mean squared error** for a regression;
- **connectionist temporal classification (CTC)** for sequence-learning, etc.

In most cases, one of these will do the trick – only when analysts are working on truly new research problems do they have to develop their own objective functions. Let us first illustrate the principles underlying ANNs with a simple example.

We have seen that ANNs are formed from an **input layer** from which the **signal vector**  $\mathbf{x}$  is inputted, an **output layer** which produces an **output vector**  $\hat{\mathbf{y}}$ , and any number of **hidden layers**; each layer consists of a number of **nodes** which are connected to the nodes of other layers *via directed edges* with associated **weights**  $\mathbf{w}$ , as seen below.

Nodes from the hidden and output layers are typically **activation nodes** – the output  $a(\mathbf{z})$  is some function of the input  $\mathbf{z}$ . Signals propagate through the ANN using simple arithmetic, once a set of weights  $\mathbf{w}$  and activation functions  $a(\cdot)$  have been selected (see Figure 21.22).

In a nutshell, at each node, the neural net computes a weighted sum of inputs, applies an activation function, and sends a signal. This is repeated until the various signals reach the final output nodes.

That part is easy – given a signal, an ANN can produce an output, as long as the weights are specified. Matrix notation can simplify the expression for the output  $\hat{\mathbf{y}}$  in terms of the signal  $\mathbf{x}$ , weights  $\mathbf{w}$ , and activation function  $a(\cdot)$ .

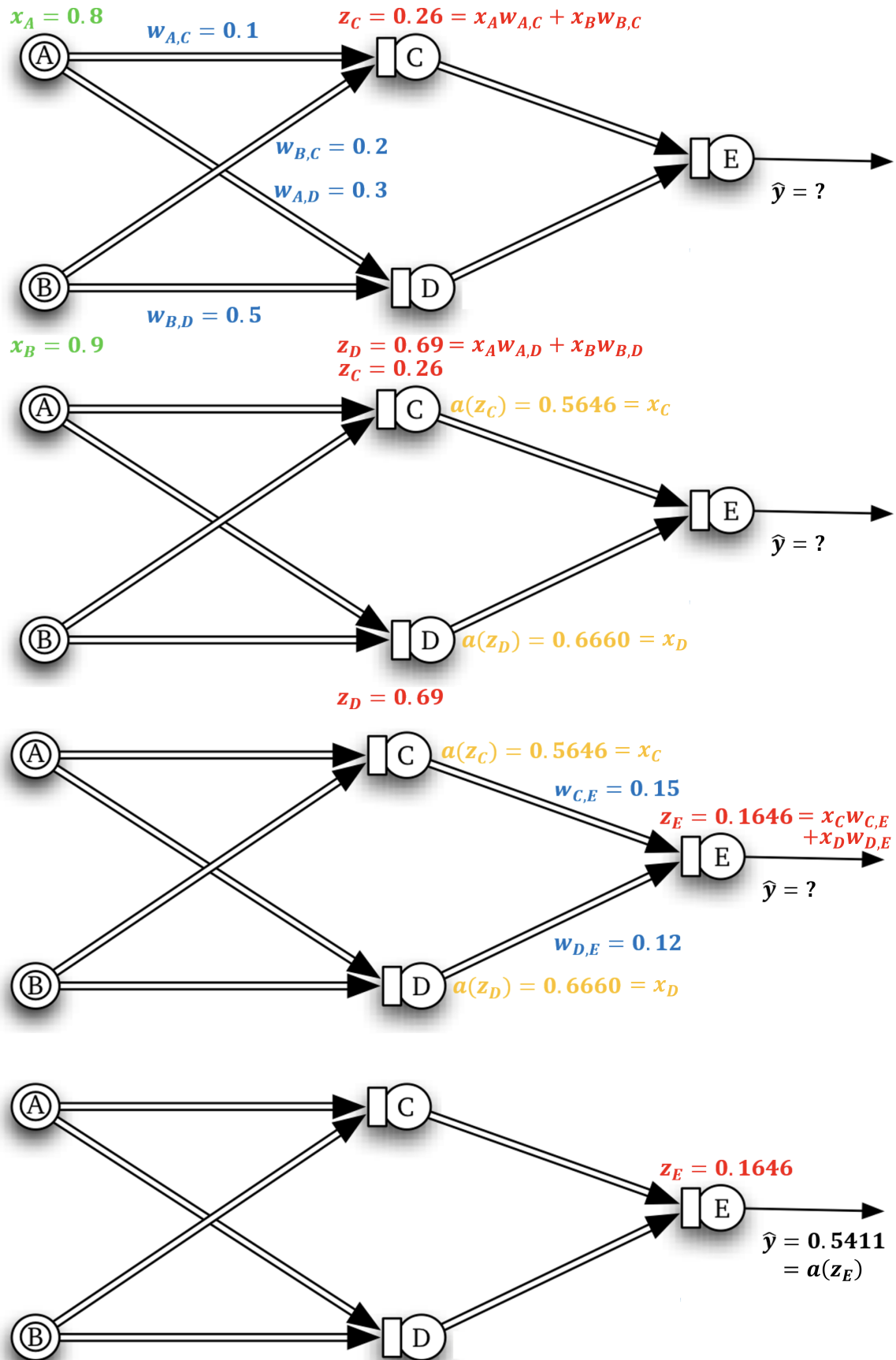


Figure 21.22: Signal propagating forward through an ANN; weights (in blue), activation functions (in yellow), inputs (in green), and output (in black).

For instance, consider the network of Figure 21.22; if

$$a(z) = (1 + \exp(-z))^{-1},$$

the network topology can be re-written as:

- **input layer with  $p$  nodes**

$$\mathbf{X}_{N \times p} = \mathbf{X}_{n \times 2} = \begin{bmatrix} x_{A,1} & x_{B,1} \\ \vdots & \vdots \\ x_{A,N} & x_{B,N} \end{bmatrix};$$

- **weights from input layer to hidden layer with  $M$  nodes**

$$\mathbf{W}_{p \times M}^{(1)} = \mathbf{W}_{2 \times 2}^{(1)} = \begin{bmatrix} w_{AC} & w_{AD} \\ w_{BC} & w_{BD} \end{bmatrix};$$

- **hidden layer with  $M$  nodes**

$$\mathbf{Z}_{N \times M}^{(2)} = \mathbf{Z}_{N \times 2}^{(2)} = \begin{bmatrix} z_{C,1} & z_{D,1} \\ \vdots & \vdots \\ z_{C,N} & z_{D,N} \end{bmatrix} = \mathbf{X} \mathbf{W}^{(1)};$$

- **activation function on hidden layer**

$$\mathbf{a}^{(2)} = \begin{bmatrix} (1 + \exp(-z_{C,1}))^{-1} & (1 + \exp(-z_{D,1}))^{-1} \\ \vdots & \vdots \\ (1 + \exp(-z_{C,N}))^{-1} & (1 + \exp(-z_{D,N}))^{-1} \end{bmatrix} = g(\mathbf{Z}^{(2)});$$

- **weights from hidden layer with  $M$  nodes to output layer with  $K$  nodes**

$$\mathbf{W}_{M \times K}^{(2)} = \mathbf{W}_{2 \times 1}^{(2)} = \begin{bmatrix} w_{CE} \\ w_{DE} \end{bmatrix};$$

- **output layer with  $K$  nodes**

$$\mathbf{Z}_{N \times K}^{(3)} = \mathbf{Z}_{N \times 1}^{(3)} = \begin{bmatrix} z_{E,1} \\ \vdots \\ z_{E,N} \end{bmatrix} = \mathbf{a}^{(2)} \mathbf{W}^{(2)};$$

- **activation function on output layer**

$$\hat{\mathbf{y}} = \mathbf{a}^{(3)} = \begin{bmatrix} (1 + \exp(-z_{E,1}))^{-1} \\ \vdots \\ (1 + \exp(-z_{E,N}))^{-1} \end{bmatrix} = g(\mathbf{Z}^{(3)});$$

The main problem is that unless the weights are judiciously selected, the output that is produced is unlikely to have anything to do with the desired output. For SL tasks (i.e., when an ANN attempts to emulate the results of training examples), there has to be some method to optimize the choice of the weights against an error function

$$R(\mathbf{W}) = \sum_{i=1}^N \sum_{\ell=1}^k (\hat{y}_{i,\ell}(\mathbf{W}) - y_{i,\ell})^2 \quad \text{or} \quad R(\mathbf{W}) = - \sum_{i=1}^N \sum_{\ell=1}^k y_{i,\ell} \ln \hat{y}_{i,\ell}(\mathbf{W})$$

(for value estimation and classification, respectively), where  $N$  is the number of observations in the training set,  $K$  is the number of output nodes in the ANN,  $y_{i,\ell}$  is the known value or class label for the  $\ell^{\text{th}}$  output of the  $i^{\text{th}}$  observation in the training set.

Enter **backpropagation**, which is simply an application of calculus' **chain rule** to  $R(\mathbf{W})$ . Under reasonable regularity condition, the desired **minimizer**  $\mathbf{W}^*$  satisfies  $\nabla R(\mathbf{W}^*) = 0$  and is found using **numerical gradient descent**.

**Gradient-Based Optimization** Initially, the weight matrix  $\mathbf{W}$  is filled with small random values (a step called **random initialization**). The weights are then gradually **trained** (or learned), based on a **feedback signal**. This occurs within a **training loop**, which works as follows:

1. draw a batch of training samples  $\mathbf{x}$  and corresponding targets  $\mathbf{y}$ ;
2. run the network on  $\mathbf{x}$  (the **forward pass**) to obtain predictions  $\hat{\mathbf{y}}$ ;
3. compute the loss of the network on the batch, a measure of the mismatch between  $\hat{\mathbf{y}}$  and  $\mathbf{y}$ ;
4. update all weights of the network in a way that slightly reduces the loss on this batch.

Repeat these steps in a loop, as often as necessary. Hopefully, the process will eventually converge on a network with very low training loss, which is to say that there will be a low mismatch between the predictions  $\hat{\mathbf{y}}$  and the target  $\mathbf{y}$ . In the vernacular, we say that the ANN has **learned** to map its inputs to correct targets.

Step 1 is easy enough. Steps 2 and 3 are simply the application of a handful of tensor operations (or matrix multiplication, as above). Step 4 is more difficult: how do we update the network's weights? Given an individual weight coefficient in the network, how can we compute whether the coefficient should be increased or decreased, and by how much?

One solution is to successively minimize the objective function along coordinate directions to find the minimum of a function; this algorithm is called **coordinate descent** and at each iteration determines a coordinate, then minimizes over the corresponding hyperplane while fixing all other coordinates [237].

It is based on the idea that optimization can be achieved by minimizing along one direction at a time. Coordinate descent is useful in situations where the objective function is not **differentiable**, as is the case for most regularized regression models, say. But this approach would be inefficient in deep learning networks, where there is a large collection of individual weights to update. A smarter approach is use the fact that all operations used to propagate a signal in the network are differentiable, and compute the **gradient** of the objective function (loss) with regard to the network's coefficients.

Following a long-standing principle of calculus, we can decrease the objective function by updating the coefficients in the **opposite direction to the gradient**.<sup>46</sup> For an input vector  $\mathbf{X}$ , a weight matrix  $\mathbf{W}$ , a target  $\mathbf{Y}$ , and a loss function  $L$ , we predict a target candidate  $\hat{\mathbf{Y}}(\mathbf{W})$ , and compute the loss when approximating  $\mathbf{Y}$  by  $\hat{\mathbf{Y}}(\mathbf{W})$ .

46: The gradient is the derivative of a tensor operation; it generalizes the notion of the derivative to functions of multidimensional inputs.

If  $\mathbf{X}$  and  $\mathbf{Y}$  are fixed, the loss function maps weights  $\mathbf{W}$  to loss values:  $f(\mathbf{W}) = L(\hat{\mathbf{Y}}(\mathbf{W}), \mathbf{Y})$ .

In much the same way that the derivative of a univariate function  $f(x)$  at a point  $x_0$  is the slope of the tangent at  $f$  at  $x_0$ , the gradient  $\nabla f(\mathbf{W}_0)$  is the tensor describing the **curvature** of  $f(\mathbf{W})$  around  $\mathbf{W}_0$ . As is the case with the derivative, we can reduce  $f(\mathbf{W})$  by moving  $\mathbf{W}_0$  to

$$\mathbf{W}_1 = \mathbf{W}_0 - s \nabla f(\mathbf{W}_0),$$

where  $s$  is the **learning rate**, a small scalar needed to approximate the curvature of the hypersurface close to  $\mathbf{W}_0$ .

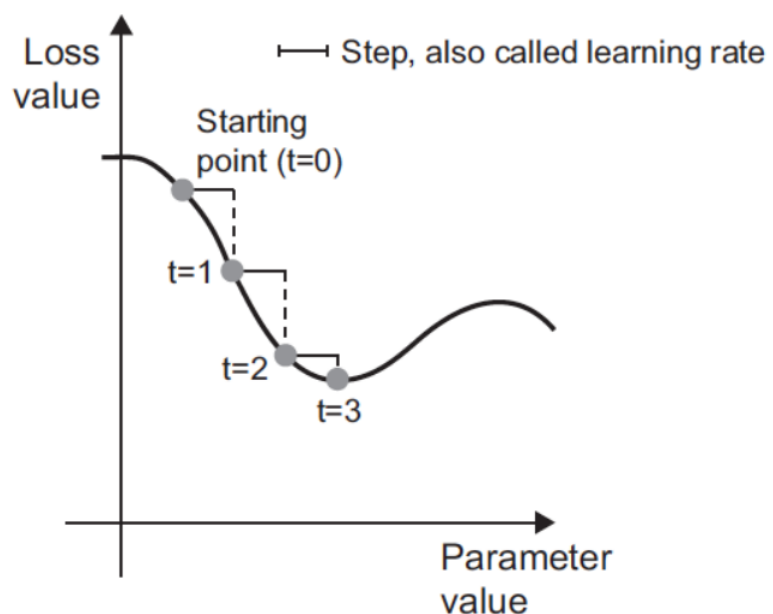
**Stochastic Gradient Descent** When dealing with ANNs, we can take advantage of the differentiability of the gradient by finding its **critical points**  $\nabla f(\mathbf{W}) = 0$  analytically.

If the neural network contains  $Q$  edges, this requires solving a polynomial equation in  $Q$  variables. However, real-world ANNs often have over a few thousand such connections (if not more), and so this analytical approach is not reasonable.

Instead, we modify the parameters slightly based on the current loss value on a random batch of data. Since we are dealing with a differentiable function, we can use a **mini-batch stochastic gradient descent** (minibatch SGD) to update the weights, simply by modifying Step 4 of the gradient descent algorithm as follows:

- 4a. compute the gradient of the loss with regard to the weights (the **backward pass**);
- 4b. update the weights “a little” in the direction opposite the gradient.

Figure 21.23 illustrates how SGD works when the network only has the one parameter to learn, with a single training sample.



**Figure 21.23:** SGD with one parameter [237].



We automatically see why it is important to choose a reasonable learning rate (the step size); too small a value leads to either slow convergence or running the risk of staying stuck at some local minimum; too large a value may send the descent to essentially random locations on the curve and overshooting the global minimum altogether.

**SGD Challenges** The main issue with minibatch SGD is that “good” convergence rates are not guaranteed, but there are other challenges:

- selecting a reasonable learning rate can be difficult. Too small a rate leads to painfully slow convergence, too large a rate can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge [237];
- the same learning rate applies to all parameter updates, which might not be ideal when the data is sparse;
- a key challenge is in minimizing highly non-convex loss functions that commonly occur in ANNs and avoiding getting trapped in sub-optimal local minima or saddle points. It is hard for SGD to escape these sub-optimal local minima and even worse for the saddle points [248].

**SGD Variants** There are several SGD variants that are commonly used by the deep learning community to overcome the aforementioned challenges. They take into account the previous weight updates when computing the next weight update, rather than simply considering the current value of the gradients. Popular **optimizers** include SGD with momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSProp, and many more [249, 250].<sup>47</sup>

ANNs can be quite accurate when making predictions – more than other algorithms, if given a proper set up (but this can be hard to achieve). They degrade gracefully, and they often work when other things fail:

- when the relationship between attributes is **complex**;
- when there are a lot of dependencies/**nonlinear relationships**;
- when the inputs are **messy** and highly-connected (images, text and speech), and
- when dealing with non-linear classification.

But they are relatively slow and prone to overfitting (unless they have access to large and diverse training sets), they are notoriously hard to interpret due to their **blackbox** nature, and there is no algorithm in place to select the optimal network topology.

Finally, even when they do perform better than other options, ANNs may not perform that much better due to the **No Free-Lunch** theorems; and they always remain susceptible to various forms of **adversarial attacks** [251], so they should be used with caution.<sup>48</sup>

**Example: Wine Dataset** In this example, we explore the wine dataset with the ANN architecture implemented in the R package `neuralnet`. We will revisit deep learning networks, and more complicated topologies, in Chapter ??.

47: A beautiful [animation](#) (created by A. Radford) compares the performance of different optimization algorithms and shows that the methods usually take different paths to reach the minimum.

48: For now, at least . . . who knows what the future holds.

```
wine = read.csv("wine.csv", header=TRUE)
wine = as.data.frame(wine)
str(wine)
```

```
'data.frame': 178 obs. of 14 variables:
 $ Class      : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Alcohol    : num  14.2 13.2 13.2 14.4 13.2 ...
 $ Malic.acid : num  1.71 1.78 2.36 1.95 2.59 ...
 $ Ash        : num  2.43 2.14 2.67 2.5 2.87 ...
 $ Alcalinity.of.ash : num  15.6 11.2 18.6 16.8 21 ...
 $ Magnesium  : int  127 100 101 113 118 112 ...
 $ Total.phenols : num  2.8 2.65 2.8 3.85 2.8 ...
 $ Flavanoids  : num  3.06 2.76 3.24 3.49 2.69 ...
 $ Nonflavanoid.phenols: num  0.28 0.26 0.3 0.24 0.39 ...
 $ Proanthocyanins : num  2.29 1.28 2.81 2.18 1.82 ...
 $ Colour.intensity : num  5.64 4.38 5.68 7.8 4.32 ...
 $ Hue         : num  1.04 1.05 1.03 0.86 1.04 ...
 $ OD280.OD315 : num  3.92 3.4 3.17 3.45 2.93 ...
 $ Proline     : int  1065 1050 1185 1480 735 ...
```

```
table(wine$Class)
```

```
1 2 3
59 71 48
```

We notice that there are only 3 classes: 1, 2, 3. These classes will be the level of the categorical response variable for a classification task.

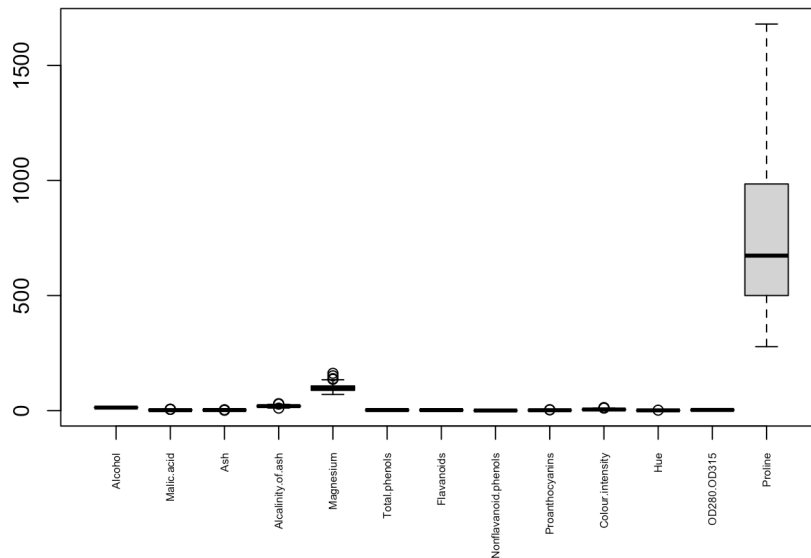
We set-up the model parameters/inputs as follows:

```
# Number of instances
n = nrow(wine)
# Dependent variable - Class
Y = wine$Class
# Independent variables (full)
X = wine[, -1]

# Indices for Class=1,2,3
C1.loc = which(Y==1)
C2.loc = which(Y==2)
C3.loc = which(Y==3)
```

We begin data exploration by taking a look at the variables' boxplots, an excellent way to understand the distribution of each variable.

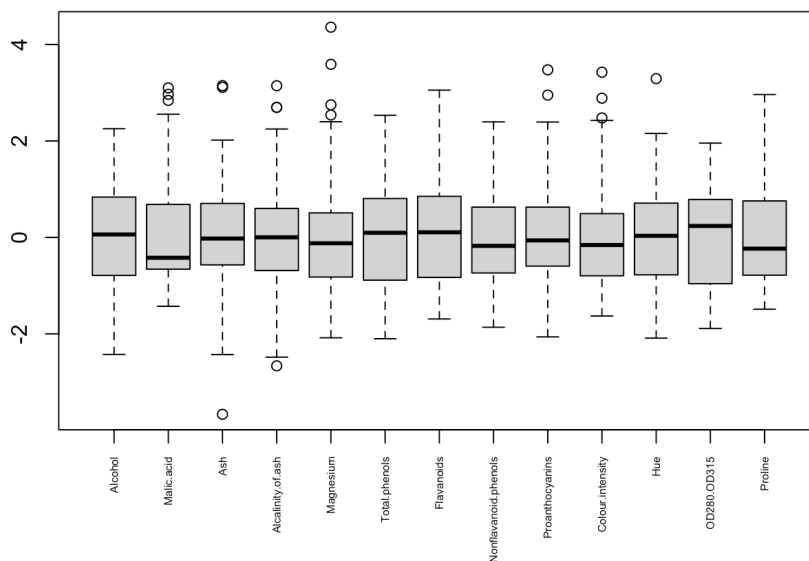
```
plot.title = "Boxplot of Variables in the Wine dataset
              (original scale)"
boxplot(X, main=plot.title, xaxt="n")
axis(1, at=1:dim(X)[2], labels=colnames(X), las=2, cex.axis=0.5)
```

**Boxplot of Variables in the Wine dataset (original scale)**

We clearly see that Proline has higher magnitudes in mean value and variability. This suggests that if we apply reduction techniques like PCA to the wine dataset, the 1st principal component will be based almost entirely on the Proline value. In order to reduce undue effects, we need to standardize the data first (see Chapter 23 for details).

```
# Standardized predictor variables (full)
X.std = scale(X)

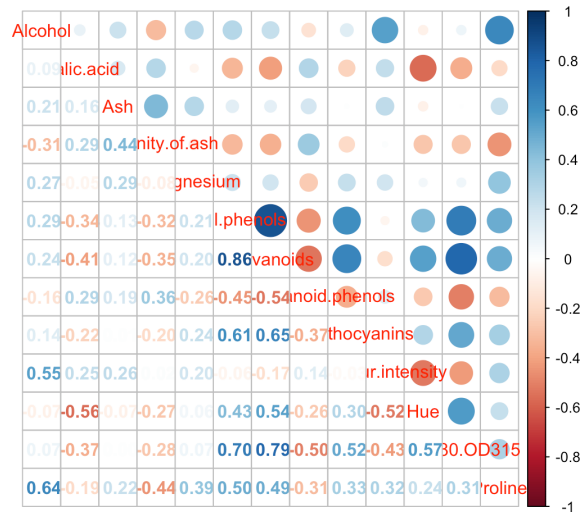
plot.title = "Boxplot of Variables in the Wine dataset
              (standardized)"
boxplot(X.std, main=plot.title, xaxt="n")
axis(1, at=1:dim(X)[2], labels=colnames(X), las=2, cex.axis=0.5)
```

**Boxplot of Variables in the Wine dataset (standardized)**



We can also calculate and display the correlation matrix:

```
X.cor = cor(X.std)
corrplot::corrplot.mixed(X.cor)
```



We write a little function that will compute the VIF of each variable in a design matrix

```
vif <- function(X){
  vif=rep(0,dim(X)[2])
  for (i in 1:dim(X)[2]){
    expl=X[, -c(i)]
    y=lm(X[,i]~expl)
    vif[i]=1/(1-summary(y)$r.squared)}
  return(vif)
}

vif.X = matrix(vif(X.std), nrow=1)
colnames(vif.X) = colnames(X)
rownames(vif.X) = "VIF"
round(t(vif.X),2)
```

	VIF		VIF
Alcohol	2.46	Nonflavanoid.phenols	1.80
Malic.acid	1.66	Proanthocyanins	1.98
Ash	2.19	Colour.intensity	3.03
Alcalinity.of.ash	2.24	Hue	2.55
Magnesium	1.42	OD280.OD315	3.79
Total.phenols	4.33	Proline	2.82
Flavanoids	7.03		

We see that Flavanoids has high multicollinearity with respect to the other variables, as its VIF is greater than 5; as such we have reasonable grounds to remove that variable from further analyses as the other variables can explain how Flavanoids would behave and doing so might reduce the risk of creating **unstable models**.

```
X = X[, -7]
X.std = X.std[, -7]
```

Now we apply a PCA reduction (see Chapter 23 for details) to further reduce the problem complexity. We start by performing principal component analysis on the standardized data:

```
pca.std = prcomp(X.std)
summary(pca.std)
```

Importance of components:

	PC1	PC2	PC3	PC4
Standard deviation	1.9757	1.5802	1.1870	0.94820
Proportion of Variance	0.3253	0.2081	0.1174	0.07492
Cumulative Proportion	0.3253	0.5334	0.6508	0.72569

	PC5	PC6	PC7	PC8
Standard deviation	0.91472	0.80087	0.74082	0.58095
Proportion of Variance	0.06973	0.05345	0.04573	0.02813
Cumulative Proportion	0.79541	0.84886	0.89460	0.92272

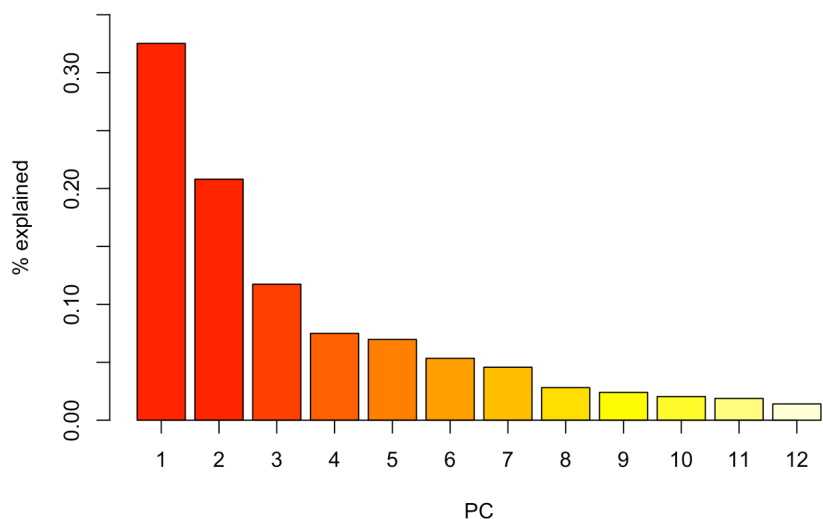
  

	PC9	PC10	PC11	PC12
Standard deviation	0.53687	0.49487	0.4750	0.41059
Proportion of Variance	0.02402	0.02041	0.0188	0.01405
Cumulative Proportion	0.94674	0.96715	0.9859	1.00000

The scree plot for proportions of variance explained by each PC is:

```
scree.y = eigen(t(X.std)%*%X.std)$values /
           sum(eigen(t(X.std)%*%X.std)$values)
barplot(scree.y, main=plot.title, ylim=c(0, 0.35),
        ylab="% explained", xlab="PC", col=heat.colors(12))
test = seq(0.7, 13.9, length.out=12)
axis(1, at=test, labels=1:12)
```

**Scatterplot matrix**

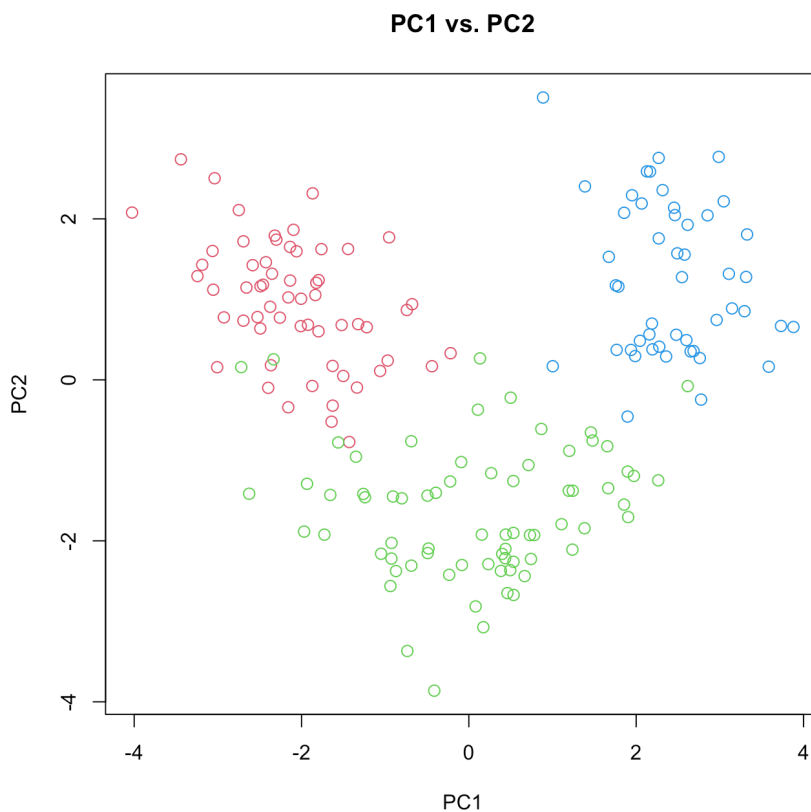


Based on the summary table, the first 6 PC would be required to retain 80% of the explanatory power; the scree plot, on the other hand, shows a knee at the 4th component.

We can explore this further: let us take a look at the scatterplot of the first two PC. We start by transforming `X.std` into its principal coordinates:

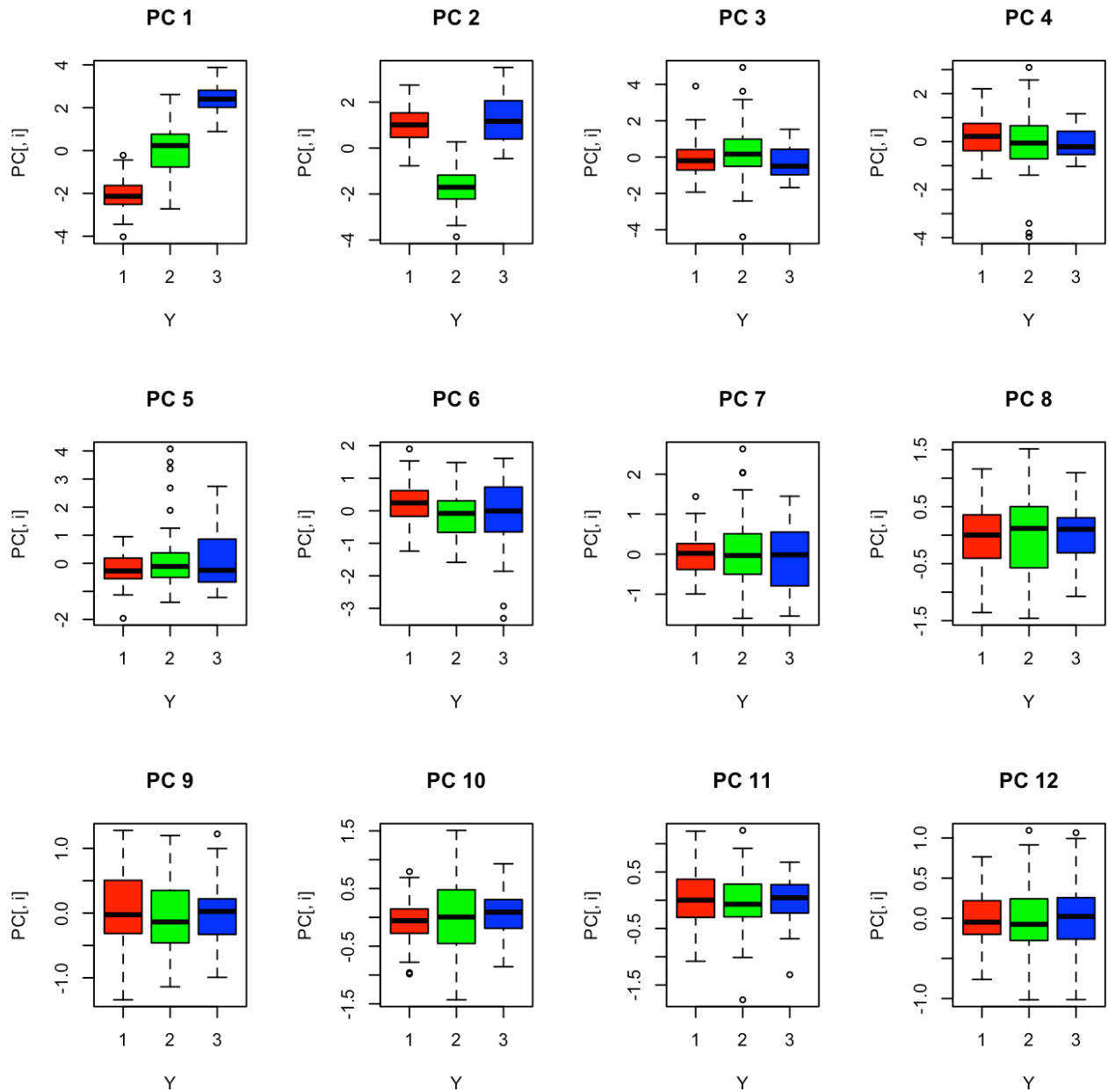
```
PC = X.std %*% pca.std$rotation
PCnames = c("PC1", "PC2", "PC3", "PC4", "PC5", "PC6", "PC7",
            "PC8", "PC9", "PC10", "PC11", "PC12")
colnames(PC) <- PCnames

plot.title = "PC1 vs. PC2"
plot(PC[,1], PC[,2], cex=1.2, main=plot.title, col=Y+1,
     xlab="PC1", ylab="PC2")
```



The scatterplot shows that the three classes are separated reasonably well by PC1 and PC2 (although, not linearly).

```
plot.title = "Boxplots (in Principal Coordinates)"
par(mfrow = c(3,4))
for (i in 1:12){
  plot.title.ind = paste("PC ", i, sep="")
  boxplot(PC[,i]~Y, main=plot.title.ind,
         col=c("red", "green", "blue"))
}
```





The boxplots further leave the impression that PC3 to PC12 do not provide as clear a separation as do PC1 and PC2. We will thus use only the latter to **visually** evaluate the effectiveness of ANN (and its prediction regions).

In order to evaluate the effectiveness of the model (i.e., does it have good predictive power without overfitting the data?), we split the data into training and testing sets.

The model is then developed using the training set (i.e., optimized using a subset of data), and then evaluated for its prediction power using the testing set.

There are  $n = 178$  observations in total: we sample 140 of them for the training set (say, 46 of class 1, 56 of class 2, and 38 of class 3). The remaining 38 observations form the testing set.

```
set.seed(1111) # for replicability
C1.train.loc = sort(sample(C1.loc, size=46))
C2.train.loc = sort(sample(C2.loc, size=56))
C3.train.loc = sort(sample(C3.loc, size=38))
train.loc = c(C1.train.loc, C2.train.loc, C3.train.loc)
test.loc = which(!(1:length(Y) %in% train.loc))

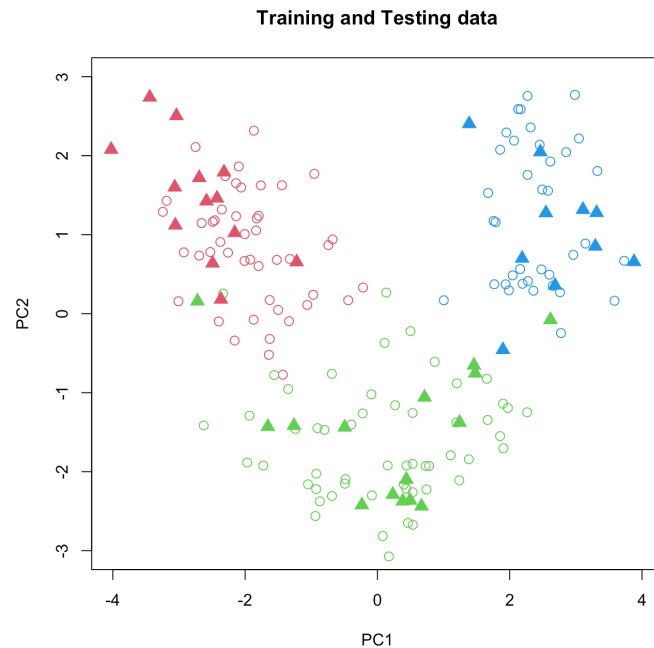
# training data
PC.train = PC[train.loc,]
Y.train = Y[train.loc]
dat.train = as.data.frame(cbind(nnet::class.ind(Y.train),
                                PC.train))
colnames(dat.train)[1:3] = c("C1", "C2", "C3")

# testing data
PC.test = PC[test.loc,]
Y.test = Y[test.loc]
dat.test = as.data.frame(cbind(nnet::class.ind(Y.test),
                                PC.test))
colnames(dat.test)[1:3] = c("C1", "C2", "C3")
```

We display the training dataset (circles) and testing dataset (triangles) on the PC1/PC2 scatterplot.

```
plot.title = "Training and Testing data"
xlimit = c(-4,4)
ylimit = c(-3,3)

plot(dat.train$PC1, dat.train$PC2, cex=1.2, col=Y.train+1,
     main=plot.title, xlab="PC1", ylab="PC2", xlim=xlimit,
     ylim=ylimit)
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5,
       col=Y.test+1)
```



If we only use the PC1/PC2-reduced data, we would expect that at least two of the test observations would be misclassified (the left-most and right-most green triangles, respectively).

We are finally ready to build an ANN *via* the R package `neuralnet` (the main function is also called `neuralnet()`). We run three analyses:

1. using only the first two principal components as inputs;
2. using the first six principal components as inputs, and
3. using all 12 principal components as inputs.

We start by forming a grid in the PC1/PC2 space on which we can colour the prediction regions.

```
predict.region.PC1=seq(-5,5, length.out=100)
predict.region.PC2=seq(-4,4, length.out=100)
predict.region=expand.grid(x=predict.region.PC1,
                           y=predict.region.PC2)
```

We will also use an expanded form of the confusion matrix:

```
# A souped-up version of the confusion matrix
confusion.expand <- function (pred.c, class) {
  temp <- mda::confusion(pred.c, class)
  row.sum <- apply(temp, 1, sum)
  col.sum <- apply(temp, 2, sum)
  t.sum <- sum(col.sum)
  tmp <- rbind(temp, rep("----", dim(temp)[2]), col.sum)
  tmp <- noquote(cbind(tmp, rep("|", dim(tmp)[1]),
                       c(row.sum, "----", t.sum)))
  dimnames(tmp) <- list(object =
    c(dimnames(temp)[[1]], "-----", "Col Sum"),
    true = c(dimnames(temp)[[2]], "|", "Row Sum"))
}
```

```

attr(tmp, "error") <- attr(temp, "error")
attr(tmp, "mismatch") <- attr(temp, "mismatch")
return(tmp)
}

```

In what follows, we build an ANN model using `neuralnet`, with PC1 and PC2 as inputs. The parameter `model.structure`, which defines the number of hidden nodes in each hidden layer, is modifiable:

- a model with no hidden layer would have `model.structure = 0`;
- 1 hidden layer of 3 nodes would require `model.structure = 3`;
- 2 hidden layers of 5 and 10 nodes, respectively, would require `model.structure = c(5,10)`, and so on.

We will use 2 hidden layers of 10 nodes each.

```

model.structure = c(10,10)
modell <- neuralnet::neuralnet(C1 + C2 + C3 ~ PC1 + PC2,
                             data = dat.train, hidden = model.structure,
                             err.fct = "ce", linear.output = FALSE)
prob.modell.test <- neuralnet::compute(modell, PC.test[,1:2])
predict.modell.test = max.col(prob.modell.test$net.result)
print(paste("Confusion matrix (testing) for model = ",
            list(model.structure)[1], sep=""))
(conf.test=confusion.expand(predict.modell.test, Y.test))

```

```
[1] "Confusion matrix (testing) for model = c(10, 10)"
```

	true			
object	1	2	3	Row Sum
1	13	1	0	14
2	0	13	1	14
3	0	1	9	10
-----	-----	-----	-----	-----
Col Sum	13	15	10	38

```

attr("error")
[1] 0.07894737

```

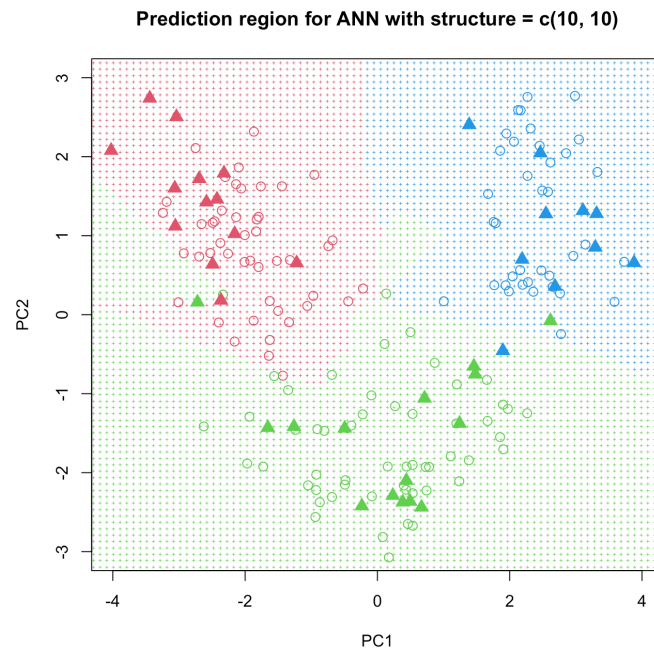
We can compute the prediction region for the two-input model and display it as follows:

```

prob.modell.region <- neuralnet::compute(modell,
                                         predict.region[,1:2])
predict.modell.region = max.col(prob.modell.region$net.result)
plot.title=paste("Prediction region for ANN with structure = ",
                list(model.structure)[1], sep="")
plot(predict.region[,1], predict.region[,2],
     main=plot.title, xlim=xlimit, ylim=ylim,
     xlab="PC1", ylab="PC2",
     col=predict.modell.region+1, pch="+", cex=0.4)
points(dat.train$PC1, dat.train$PC2, cex=1.2,
       col=Y.train+1)

```

```
points(dat.test$PC1, dat.test$PC2, pch=17, cex=1.5,
       col=Y.test+1)
```



Note the complex decision boundary.

Since the error function we seek to minimize (e.g., SSE) is **non-convex**, it is possible for ANN to get stuck at local minima, rather than converge to the global minimum. We can run the model a number of times (say, 50 replicates) and find the average prediction.<sup>49</sup>

49: **Note:** this code may produce an error saying that ANN has issues converging (especially for simpler models). If this happens, the simple solution is to re-run the code again or change the seed.

```
model.structure = c(10,10)
n.j = 50
conf.train.vector = conf.test.vector = NULL

for (j in 1:n.j){
  model1 <- neuralnet::neuralnet(C1 + C2 + C3 ~ PC1 + PC2,
    data = dat.train, hidden = model.structure,
    err.fct = "ce", linear.output = FALSE)

  prob.model1.test <- neuralnet::compute(model1,
                                          PC.test[,1:2])
  predict.model1.test = max.col(prob.model1.test$net.result)
  conf.test = confusion.expand(predict.model1.test,
                               Y.test)
  conf.test.vector=c(conf.test.vector,
                    attributes(conf.test)$error)
}

# number of misclassifications
conf.test.vector = round(conf.test.vector*length(Y.test))

print(paste("Summary of number of misclassifications
in testing data out of", n.j, "trials", sep=" "))
```

```
round(summary(conf.test.vector), digits=2)
```

```
[1] "Summary of number of misclassifications in testing data out of 50 trials"
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2.00	3.00	3.00	2.92	3.00	4.00

We build ANN for the PCA-reduced 6-input dataset.<sup>50</sup> For each ANN topology, we replicate the process 25 times. It should be noted that prediction regions are not computed, as our input is in more than 2 dimensional space.

50: We will try 11 different topologies: no hidden layer; 1 hidden layer with 2, 6, 10, and 30 nodes; 2-hidden layers with (6,6), (10,10), (30,30) nodes, and 30-hidden layers with (6,6,6), (10,10,10) and (30,30,30) nodes.

```
model.structure = list(0,          # no hidden layer
  2, 6, 10, 30,                  # 1 hidden layer
  rep(6,2), rep(10,2), rep(30,2), # 2 hidden layers
  rep(6,3), rep(10,3), rep(30,3)) # 3 hidden layers

set.seed(1)
results = NULL
n.loop = length(model.structure)
n.j = 25

for (i in 1:n.loop){
  conf.train.vector = conf.test.vector = NULL

  for (j in 1:n.j){
    modell <- neuralnet::neuralnet(C1 + C2 + C3 ~
      PC1 + PC2 + PC3 + PC4 + PC5 + PC6,
      data = dat.train, hidden = model.structure[[i]],
      err.fct = "ce", linear.output = FALSE)

    prob.model1.test <- neuralnet::compute(modell,
      PC.test[,1:6])
    predict.model1.test = max.col(prob.model1.test$net.result)
    conf.test=confusion.expand(predict.model1.test,
      Y.test)
    conf.test.vector=c(conf.test.vector,
      attributes(conf.test)$error)
  }

  results[[i]] = summary(round(conf.test.vector*length(Y.test)))
}

results = as.data.frame(dplyr::bind_rows(results,
  .id = "column_label"))
colnames(results) <- c("hidden", "min", "Q1", "med", "mean",
  "Q3", "max")
results$hidden <- model.structure
```

hidden	min	Q1	med	mean	Q3	max
0	2	2	2	2.00	2	2
2	1	2	2	1.92	2	3
6	1	2	2	1.92	2	2

10	1	2	2	1.96	2	2
30	2	2	2	2.00	2	2
6, 6	1	2	2	1.96	2	2
10, 10	1	2	2	1.88	2	2
30, 30	2	2	2	2.00	2	2
6, 6, 6	1	2	2	1.88	2	2
10, 10, 10	1	2	2	1.88	2	2
30, 30, 30	1	2	2	1.96	2	2

We can repeat this process once more, using all 12 PC.

```
for (i in 1:n.loop){
  conf.train.vector = conf.test.vector = NULL

  for (j in 1:n.j){
    modell <- neuralnet::neuralnet(C1 + C2 + C3 ~ .,
                                   data = dat.train,
                                   hidden = model.structure[[i]],
                                   err.fct = "ce", linear.output = FALSE)

    prob.modell.test <- neuralnet::compute(modell,
                                           PC.test[,1:12])
    predict.modell.test = max.col(prob.modell.test$net.result)
    conf.test=confusion.expand(predict.modell.test,
                               Y.test)
    conf.test.vector=c(conf.test.vector,
                      attributes(conf.test)$error)
  }

  results[[i]] = summary(round(conf.test.vector*length(Y.test)))
}

results = as.data.frame(dplyr::bind_rows(results,
                                           .id = "column_label"))
colnames(results) <- c("hidden", "min", "Q1", "med", "mean",
                      "Q3", "max")
results$hidden <- model.structure
```

hidden	min	Q1	med	mean	Q3	max
0	2	2	2	2.00	2	2
2	1	1	1	1.80	3	4
6	1	2	2	2.04	2	3
10	1	2	2	1.96	2	2
30	2	2	2	2.00	2	2
6, 6	1	2	2	2.00	2	3
10, 10	1	2	2	1.92	2	3
30, 30	1	2	2	1.84	2	2
6, 6, 6	1	1	2	1.76	2	3
10, 10, 10	1	2	2	1.84	2	2
30, 30, 30	1	1	2	1.68	2	2

Comparing the mean number of misclassifications, what can we conclude?

### 21.4.4 Naïve Bayes Classifiers

In classical statistics, model parameters (such as  $\mu$  and  $\sigma$ , say) are treated as constants; **Bayesian statistics**, on the other hand assume that model parameters are **random variables**.

**Bayes' Theorem** lies at the foundation of this approach:

$$P(\text{hypothesis} \mid \text{data}) = \frac{P(\text{data} \mid \text{hypothesis}) \times P(\text{hypothesis})}{P(\text{data})},$$

or simply

$$P(H \mid D) = \frac{P(D \mid H) \times P(H)}{P(D)}.$$

This is sometimes written in shorthand as  $P(H \mid D) \propto P(D \mid H) \times P(H)$ ; in other words, our **degree of belief in a hypothesis should be updated by the evidence provided by the data**.<sup>51</sup>

**Naïve Bayes Classification for Tumor Diagnoses** Suppose we are interested in diagnosing whether a tumor is benign or malignant, based on several measurements obtained from video imaging.

Bayes' Theorem can be recast as

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{evidence}} \propto \text{likelihood} \times \text{prior},$$

where


- **posterior:**  $P(H \mid D)$  = based on collected data, how likely is a given tumor to be benign (or malignant)?
- **prior:**  $P(H)$  = in what proportion are tumors benign (or malignant) in general?
- **likelihood:**  $P(D \mid H)$  = knowing a tumor is benign (or malignant), how likely is it that these particular measurements would have been observed?
- **evidence:**  $P(D)$  = regardless of a tumor being benign or malignant, what is the chance that a tumor has the observed characteristics?

The **naïve Bayes classifiers** (NBC) procedure is straightforward.

1. **Objective function:** a simple way to determine whether a tumor is benign or malignant is to compare **posterior probabilities** and choose the one with highest probability. That is, we diagnose a tumor as **malignant** if

$$\frac{P(\text{malignant} \mid D)}{P(\text{benign} \mid D)} = \frac{P(D \mid \text{malignant}) \times P(\text{malignant})}{P(D \mid \text{benign}) \times P(\text{benign})} > 1,$$

and as **benign** otherwise.

2. **Dataset:** there are  $n = 699$  observations (from the [in]famous [BreastCancer-Wisconsin.csv](#)  dataset) with nine predictor measurements, each scored on a scale of 1 to 10, a score of 0 being reserved for missing values. The predictors include items such as `Clump_Thickness` and `Bare_Nuclei`; the categorical response variable is the `Class` (Benign, Malignant).

51: Nobody disputes the validity of Bayes' Theorem, and it has proven to be a useful component in various models and algorithms, such as email spam filters, and the following example, but the **use** of Bayesian statistics is controversial in many quarters. We will discuss Bayesian data analysis in depth in Chapter 26.

```
dat.BC = read.csv("BreastCancer-Wisconsin.csv",
                  header=TRUE, stringsAsFactors = TRUE)
str(dat.BC)
```

```
'data.frame': 699 obs. of 10 variables:
 $ Clump_Thickness      : int  5 5 3 6 4 8 1 2 2 4 ...
 $ Uniformity_of_Cell_Size : int  1 4 1 8 1 10 1 1 1 2 ...
 $ Uniformity_of_Cell_Shape : int  1 4 1 8 1 10 1 2 1 1 ...
 $ Marginal_Adhesion    : int  1 5 1 1 3 8 1 1 1 1 ...
 $ Single_Epithelial_Cell_Size: int  2 7 2 3 2 7 2 2 2 2 ...
 $ Bare_Nuclei          : int  1 10 2 4 1 10 10 1 1 1 ...
 $ Bland_Chromatin       : int  3 3 3 3 3 9 3 3 1 2 ...
 $ Normal_Nucleoli       : int  1 2 1 7 1 7 1 1 1 1 ...
 $ Mitoses               : int  1 1 1 1 1 1 1 1 5 1 ...
 $ Class                 : Factor w/ 2 levels "Benign","Malignant": 1 1 1 1 1 2 1 1 1 1 ...
```

In table layout, the first 6 observations look like:

```
head(dat.BC)
```

Clump_Thickness	Uniformity_of_Cell_Size	Uniformity_of_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bare_Nuclei	Bland_Chromatin	Normal_Nucleoli	Mitoses	Class
5	1	1	1	2	1	3	1	1	Benign
5	4	4	5	7	10	3	2	1	Benign
3	1	1	1	2	2	3	1	1	Benign
6	8	8	1	3	4	3	7	1	Benign
4	1	1	3	2	1	3	1	1	Benign
8	10	10	8	7	10	9	7	1	Malignant

We create a training/testing split for the data, by selecting roughly 80%/20% of the observations.

```
set.seed(1234) # for reproducibility
ind = 1:nrow(dat.BC)
prop.train = 0.8
n.train = floor(nrow(dat.BC)*prop.train)

# indices of training observations
loc.train = sort(sample(ind, n.train, replace=FALSE))
# indices of testing observations
loc.test = ind[-which(ind %in% loc.train)]
# training data
dat.BC.train = dat.BC[loc.train,]
# test data
dat.BC.test = dat.BC[loc.test,]
```

We separate the Benign and Malignant subsets of the training data for graphing purposes.

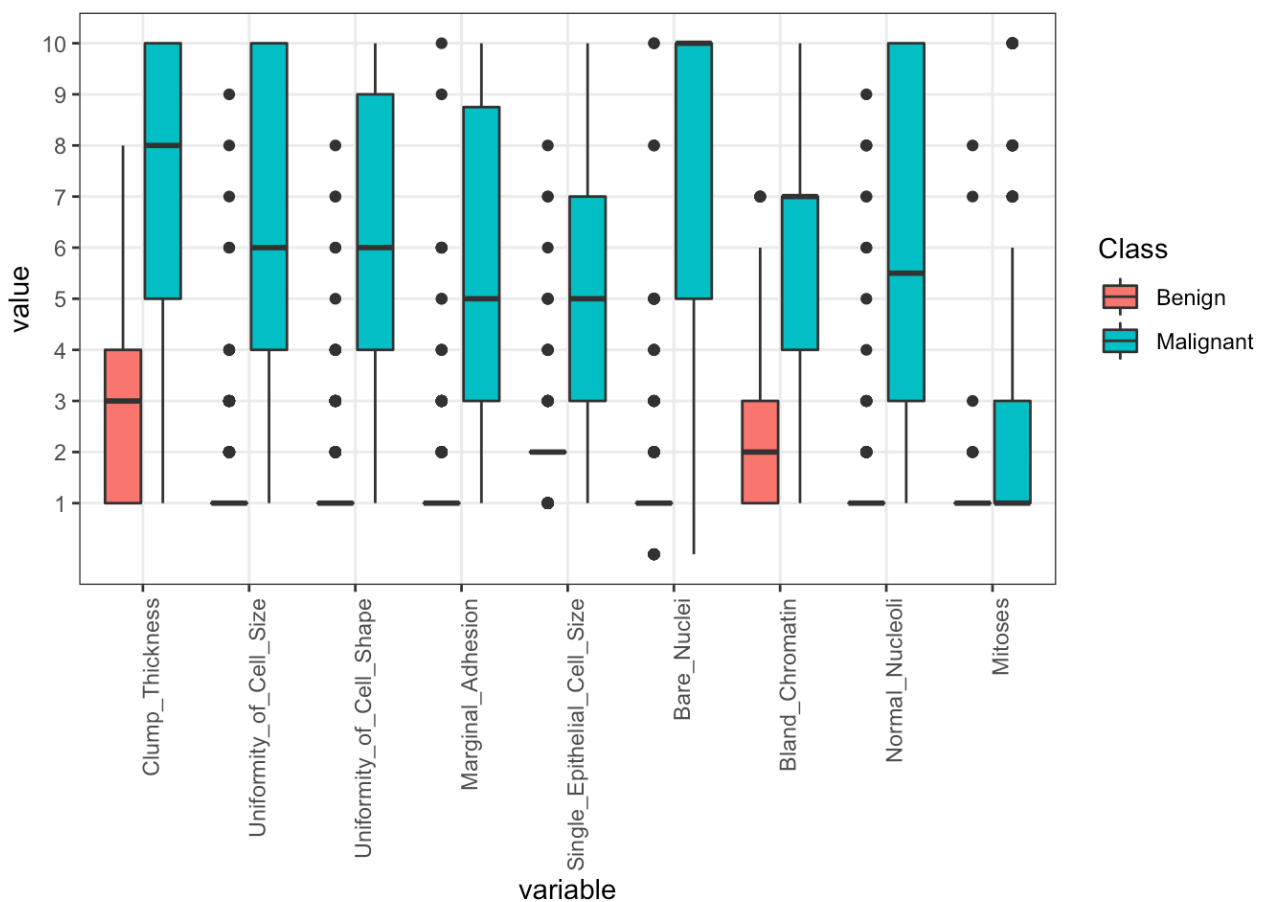
```
location.Benign = which(dat.BC.train$Class=="Benign")
location.Malignant = which(!(1:nrow(dat.BC.train)
                             %in% location.Benign))
cols_remove = c("Class")
```



```
dat.Benign=dat.BC.train[location.Benign,!colnames(dat.BC)
  %in% cols_remove]
dat.Malignant=dat.BC.train[location.Malignant,
  !colnames(dat.BC) %in% cols_remove]
```

The boxplots of the training measurements are shown below.

```
library(ggplot2)
dat.BC2 = reshape2::melt(dat.BC.train, id.var="Class")
ggplot(data = dat.BC2, aes(x=variable, y=value)) +
  geom_boxplot(aes(fill=Class)) +
  scale_y_discrete(limits = 1:10) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



From these plots, we learn that benign tumors have lower scores on average, while malignant tumors have much higher scores and variabilities.<sup>52</sup>

3. **Assumptions:** we assume that the scores of each measurement in a class are independent of one another (hence the **naïve** qualifier); this assumption reduces the likelihood to

$$\begin{aligned}
 P(H \mid D) &= P(H \mid x_1, x_2, \dots, x_9) \propto P(x_1, x_2, \dots, x_9 \mid H) \times P(H) \\
 &= P(x_1 \mid H) \times \dots \times P(x_9 \mid H) \times P(H).
 \end{aligned}$$

Note that this assumption of **conditional independence** is not usually satisfied.

52: In what follows, we treat the test observations as **undiagnosed cases**.

53: In situations where we have no knowledge about the distribution of priors, we may simply assume a **non-informative prior**. In this case, the prevalence rates would be the same for both responses.

4. **Prior distribution:** we can ask subject matter experts to provide a rough estimate for the general ratio of benign to malignant tumors, or use the proportion of benign tumors in the sample as our prior.<sup>53</sup> In this example, we will assume that the training data represents the tumor population adequately, and we use the observed proportions as estimated prior probabilities.

```
n.Benign.train = nrow(dat.Benign)
n.Malignant.train = nrow(dat.Malignant)
(prior.Benign = n.Benign.train/(n.Benign.train +
                               n.Malignant.train))
(prior.Malignant = 1 - prior.Benign)
```

```
[1] 0.6529517
[1] 0.3470483
```

54: In other problems, the predictors could be continuous rather than discrete, in which case we would use continuous distributions instead; even in discrete case, the multinomial assumption might not be appropriate.

5. **Computation of likelihoods:** under conditional independence, each measurement is assumed to follow a multinomial distribution (since scores are provided on a 1 to 10 scale): for each predictor, for each class, we must estimate  $p_1, \dots, p_{10}$ , with  $p_1 + \dots + p_{10} = 1$ .<sup>54</sup> The best estimates are thus

$$\hat{p}_{\ell, \text{pred}} = \frac{\text{\# of training cells in the class with pred score } \ell}{\text{\# of training cells in the class}}, \quad \ell = 1, \dots, 10.$$

This is done in the code below; note that `count.xyz` is a count matrix, while `freq.xyz` is a frequency matrix.

```
# Benign cells
count.Benign = freq.Benign = NULL
for (i in 1:(ncol(dat.BC.train)-1)){
  test.count = table(c(dat.Benign[,i],0:10))-1
  test.freq = test.count/sum(test.count)
  count.Benign = cbind(count.Benign, test.count)
  freq.Benign = cbind(freq.Benign, test.freq)
}
colnames(count.Benign) = colnames(freq.Benign)
                        = colnames(dat.Benign)
p.Benign = freq.Benign
p.Benign[1,] = 1

# Malignant cells
count.Malignant = freq.Malignant = NULL
for (i in 1:(ncol(dat.BC.train)-1)){
  test.count = table(c(dat.Malignant[,i],0:10))-1
  test.freq = test.count/sum(test.count)
  count.Malignant = cbind(count.Malignant, test.count)
  freq.Malignant = cbind(freq.Malignant, test.freq)
}
colnames(count.Malignant) = colnames(freq.Malignant)
                        = colnames(dat.Malignant)
p.Malignant = freq.Malignant
p.Malignant[1,] = 1
```

These are then the best estimates for the multinomial parameters, for the benign tumors:

```
table(p.Benign)
```

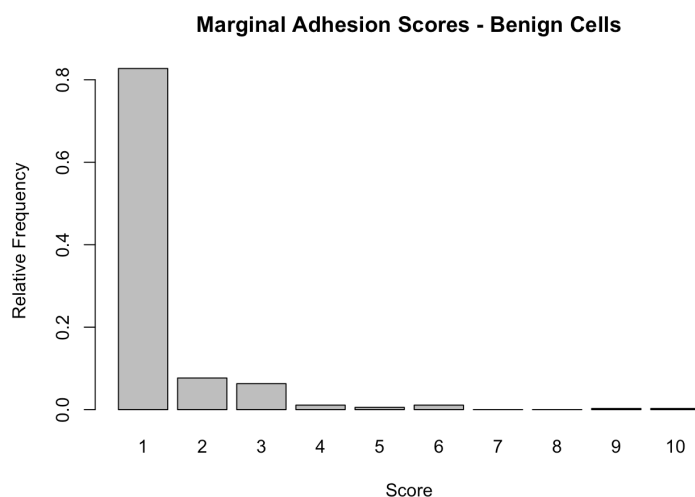
	Clump_Thickness	Uniformity_of_Cell_Size	Uniformity_of_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bare_Nuclei	Bland_Chromatin	Normal_Nucleoli	Mitoses
0	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000
1	0.3342466	0.8356164	0.7616438	0.8273973	0.1123288	0.8383562	0.3397260	0.8931507	0.9780822
2	0.0986301	0.0821918	0.1178082	0.0767123	0.7863014	0.0465753	0.3589041	0.0575342	0.0136986
3	0.2000000	0.0575342	0.0821918	0.0630137	0.0630137	0.0356164	0.2657534	0.0219178	0.0027397
4	0.1315068	0.0109589	0.0219178	0.0109589	0.0164384	0.0136986	0.0136986	0.0027397	0.0000000
5	0.1945205	0.0000000	0.0027397	0.0054795	0.0109589	0.0219178	0.0082192	0.0027397	0.0000000
6	0.0356164	0.0054795	0.0054795	0.0109589	0.0027397	0.0000000	0.0027397	0.0054795	0.0000000
7	0.0027397	0.0027397	0.0054795	0.0000000	0.0054795	0.0000000	0.0109589	0.0054795	0.0027397
8	0.0027397	0.0027397	0.0027397	0.0000000	0.0027397	0.0054795	0.0000000	0.0082192	0.0027397
9	0.0000000	0.0027397	0.0000000	0.0027397	0.0000000	0.0000000	0.0000000	0.0027397	0.0000000
10	0.0000000	0.0000000	0.0000000	0.0027397	0.0000000	0.0054795	0.0000000	0.0000000	0.0000000

For the malignant tumors, we have:

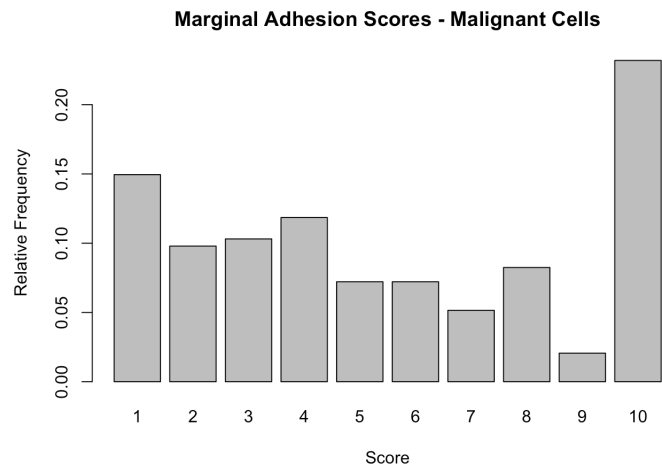
```
table(p.Malignant)
```

	Clump_Thickness	Uniformity_of_Cell_Size	Uniformity_of_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bare_Nuclei	Bland_Chromatin	Normal_Nucleoli	Mitoses
0	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000
1	0.0154639	0.0103093	0.0103093	0.1494845	0.0051546	0.0463918	0.0051546	0.1752577	0.5257732
2	0.0154639	0.0360825	0.0257732	0.0979381	0.0927835	0.0257732	0.0309278	0.0309278	0.1082474
3	0.0567010	0.1134021	0.0979381	0.1030928	0.2010309	0.0721649	0.1546392	0.1391753	0.1340206
4	0.0567010	0.1391753	0.1546392	0.1185567	0.1391753	0.0567010	0.1185567	0.0773196	0.0515464
5	0.1649485	0.1185567	0.1185567	0.0721649	0.1494845	0.0721649	0.1340206	0.0773196	0.0257732
6	0.0618557	0.1030928	0.1030928	0.0721649	0.1546392	0.0206186	0.0360825	0.0721649	0.0154639
7	0.0927835	0.0824742	0.1082474	0.0515464	0.0360825	0.0309278	0.2783505	0.0463918	0.0412371
8	0.1701031	0.1134021	0.1082474	0.0824742	0.0824742	0.0876289	0.1082474	0.0773196	0.0309278
9	0.0618557	0.0154639	0.0360825	0.0206186	0.0103093	0.0360825	0.0463918	0.0412371	0.0000000
10	0.3041237	0.2680412	0.2371134	0.2319588	0.1288660	0.5463918	0.0876289	0.2628866	0.0670103

```
barplot(p.Benign[2:11,4],
        xlab = "Score", ylab = "Relative Frequency",
        main = "Marginal Adhesion Scores - Benign Cells")
```



```
barplot(p.Malignant[2:11,4],
       xlab = "Score", ylab = "Relative Frequency",
       main = "Marginal Adhesion Scores - Malignant Cells")
```



Multiplying probabilities across predictors from each multinomial distribution (one each for both classes) provides the overall likelihoods for benign and malignant tumors, respectively. For instance, if the signature of an undiagnosed case was

$$(1, 1, 3, 1, 0, 2, 3, 2, 2),$$

then we would multiply the probabilities corresponding to each score across predictors, once assuming that the cell was benign, and once assuming it was malignant:

$$\begin{aligned} P(D | H) &= P(\mathbf{x} = (1, 1, 3, 1, 0, 2, 3, 2, 2) | H) \\ &= P(x_1 = 1 | H) \times P(x_2 = 1 | H) \times P(x_3 = 3 | H) \\ &\quad \times P(x_4 = 1 | H) \times P(x_5 = 0 | H) \times P(x_6 = 2 | H) \\ &\quad \times P(x_7 = 3 | H) \times P(x_8 = 2 | H) \times P(x_9 = 2 | H). \end{aligned}$$

We can extract the signature vector of probabilities as follows:

```
x = c(1,1,3,1,0,2,3,2,2) + 1
y = c(1,2,3,4,5,6,7,8,9)
```

For the benign class, we have:

```
p.Benign[as.matrix(data.frame(x,y))]
```

```
[1] 0.33424658 0.83561644 0.08219178 0.82739726
[5] 1.00000000 0.04657534 0.26575342
[8] 0.05753425 0.01369863
```

```
(l.Benign = prod(p.Benign[as.matrix(data.frame(x,y))]))
```

```
[1] 1.852912e-07
```

For the malignant class, we have:

```
p.Malignant[as.matrix(data.frame(x,y))]
```

```
[1] 0.01546392 0.01030928 0.09793814 0.14948454
[5] 1.00000000 0.02577320 0.15463918
[8] 0.03092784 0.10824742
```

```
(l.Malignant = prod(p.Malignant[as.matrix(data.frame(x,y))]))
```

```
[1] 3.114231e-11
```

Based on the multinomial probabilities given in `p.Benign` and `p.Malignant`, the (naïve) likelihood of the undiagnosed case being a benign tumor would thus be  $1.86 \times 10^{-7}$ , while the likelihood of it being a malignant tumor would be  $3.11 \times 10^{-11}$ .<sup>55</sup>

6. **Computation of posterior:** multiplying the corresponding prior probabilities and likelihoods, we get a quantity that is proportional to the respective posterior probabilities:

55: **Careful!** These are the likelihoods, not the posteriors.

$$P(H \mid \mathbf{x}) \propto P(H) \times P(\mathbf{x} \mid H) \approx P(H) \times \prod_{j=1}^9 P(x_j \mid H).$$

The “likelihoods” can be computed as follows:

```
test.Benign = test.Malignant = NULL
likelihood.Benign = likelihood.Malignant = NULL

for (i in 1:nrow(dat.BC.test)){
  location = rapply(dat.BC.test[i,-10]+1,c)
  for(j in 1:length(location)){
    test.Benign[j] = p.Benign[location[j],j]
    test.Malignant[j] = p.Malignant[location[j],j]
  }

  likelihood.Benign.i = prod(test.Benign)
  likelihood.Malignant.i = prod(test.Malignant)

  likelihood.Benign = c(likelihood.Benign,
                        likelihood.Benign.i)
  likelihood.Malignant = c(likelihood.Malignant,
                           likelihood.Malignant.i)
}
```

The “posteriors” can then be computed as follows:

```
posteriors=cbind(likelihood.Benign*prior.Benign,
                 likelihood.Malignant*prior.Malignant)
```

For the undiagnosed case  $\mathbf{x} = (1, 1, 3, 1, 0, 2, 3, 2, 2)$ , we obtain:

```
l.Benign*prior.Benign
l.Malignant*prior.Malignant
```

```
[1] 1.209862e-07
[1] 1.080789e-11
```

Comparing the posteriors

$$P(\text{Malignant} \mid \mathbf{x}) < P(\text{Benign} \mid \mathbf{x}),$$

we conclude that the tumor in the undiagnosed case is **likely benign**.<sup>56</sup> We can complete the procedure for all observations in the test set:

56: Note that we have no measurement on how much more likely it is to be benign than to be malignant – the classifier is **not calibrated**.

```
n.test=nrow(dat.BC.test)
prediction=rep(NA, n.test)
prediction[which(posterior[1]>posterior[2])]="Benign"
prediction[which(posterior[1]<posterior[2])]="Malignant"
prediction=as.factor(prediction)
table(prediction)
```

```
prediction
  Benign Malignant
      85         55
```

Since we actually know the true outcome for the test subjects, we can take a look at the NBC's performance on the data.

```
table(dat.BC.test$Class,prediction)
```

		prediction	
		Benign	Malignant
actual	Benign	85	8
	Malignant	0	47

Let's take a look at cases where NBC misclassified, and their corresponding posteriors:

```
dat.misclassified = dat.BC.test[
  which(dat.BC.test$Class!=prediction),]
missed.class = prediction[
  which(dat.BC.test$Class!=prediction)]
wrong.classifications = cbind(posterior[
  which(dat.BC.test$Class!=prediction),])
colnames(wrong.classifications) =
  c("Posterior.Benign", "Posterior.Malignant")
wrong.classifications =
  cbind(dat.misclassified, wrong.classifications)
table(wrong.classifications)
```

The confusion matrix tells us that 8 out of 140 cases (5.7%) are misclassified. A closer look at misclassified cases reveals that 3 of the 8 false positives are a result of a posterior probability being 0 (a score level that was not observed in the training set). Taking a close look at ID 130, for instance, all but `Single_Epithelial_Cell_Size`

	Clump_Thickness	Uniformity_of_Cell_Size	Uniformity_of_Cell_Shape	Marginal_Adhesion	Single_Epithelial_Cell_Size	Bare_Nuclei	Bland_Chromatin	Normal_Nucleoli	Mitoses	Class	Posterior.Benign	Posterior.Malignant
9	2	1	1	1	2	1	1	1	5	Benign	0	0
130	1	1	1	1	10	1	1	1	1	Benign	0	0
197	8	4	4	5	4	7	7	8	2	Benign	0	0
233	8	4	6	3	3	1	4	3	1	Benign	0	0
253	6	3	3	5	3	10	3	5	3	Benign	0	0
320	4	4	4	4	6	5	7	3	1	Benign	0	0
353	3	4	5	3	7	3	4	6	1	Benign	0	0
658	5	4	5	1	8	1	3	6	1	Benign	0	0

have scores of 1, strongly indicating that tumor is likely benign (perhaps the 10 is a typo?).

Can we prevent misclassification similar to case in ID 130? One way to do so is to replace all 0 probabilities in the likelihood matrices by a small  $\varepsilon$  (obtained by multiplying a base probability with the smallest non-zero probability), and by re-scaling the columns so that they all add up to 1 (excluding the missing values from the process). If  $\varepsilon$  is small enough, the larger probabilities will not be affected, in practice.<sup>57</sup>

57: Using a base probability of  $10^{-8}\%$ , for instance, would reduce the misclassification rate on the test data to 6/140 (4.3%).

**Notes and Comments** In practice, various prior distributions or conditional distributions (for the features) can be used; domain matter expertise can come in handy during these steps:

- the naive assumption is made out of **convenience**, as it renders the computation of the likelihood much simpler;
- the variables in a dataset are not typically independent of one another, but NBC still works well with test data (usually) – the method **seems to be robust** against departure from the independence assumption;
- dependency among variables may change the true posterior values, but the class with maximum posterior probabilities is often left **unchanged**;
- in the classification context, we typically get more insight from independent/correlated data than from correlated data;
- NBC works best for independent cases, but optimality can also be reached when dependency among variables inconsistently support one class over another;
- the **choice of a prior** may have a great effect on the classification predictions, as can the presence of outlying observations, especially when  $|\text{Tr}|$  is small);
- it is not practical to conduct NBC manually, as we have done in this section – a complete implementation can be called by using the method `naiveBayes()` from the R package `e1071` (make sure to read the documentation first!), and
- a final reminder that, like the SVM models, NBC is **not calibrated** and should not be used to estimate probabilities.

## 21.5 Ensemble Learning

In practice, individual learners are often **weak** – they perform better than random guessing would, but not necessarily that much better, or

sufficiently so for specific analytical purposes. In the late 80's, Kearns and Valiant asked the following question: can a set of weak learners be used to create a strong learner? The answer, as it turns out, is yes – *via ensemble learning* methods.

As an example, scientists trained 16 pigeons (weak learners, one would assume) to identify pictures of magnified biopsies of possible breast cancers. On average, each pigeon had an accuracy of about 85%, but when the most popular answer among the group was selected, the accuracy jumped to 99% [252].<sup>58</sup>

58: The material of this section closely follows [3].

### 21.5.1 Bagging

**Bootstrap aggregation** (also known as **bagging**) is an extension of bootstrapping. Originally, bootstrapping was used in situations where it is nearly impossible to compute the variance of a quantity of interest by exact means (see Section 20.3.2).

But it can also be used to improve the performance of various statistical learners, especially those that exhibit **high variance** (such as CART).<sup>59</sup> Given a learning method, bagging can be used to **reduce the variance** of that method.

59: Low variance methods, in comparison, are those for which the results, structure, predictions, etc. remain roughly similar when using different training sets, such as OLS when  $N/p \gg 1$ , and are less likely to benefit from the use of ensemble learning.

If  $Z_1, \dots, Z_B$  are **independent** predictions at  $\mathbf{x} \in \text{Te}$ , say, with

$$\text{Cov}(Z_i, Z_j) = \begin{cases} \sigma^2 & \text{if } i = j \\ 0 & \text{else} \end{cases}$$

the central limit theorem states that

$$\begin{aligned} \text{Var}(\bar{Z}) &= \text{Var}\left(\frac{Z_1 + \dots + Z_B}{B}\right) = \frac{1}{B^2} \text{Var}(Z_1 + \dots + Z_B) \\ &= \frac{1}{B^2} \sum_{i,j=1}^B \text{Cov}(Z_i, Z_j) = \frac{1}{B^2} \sum_{k=1}^B \text{Var}(Z_k) = \frac{\sigma^2}{B}. \end{aligned}$$

In other words, averaging a set of observations reduces the variance as  $\sigma^2 \geq \frac{\sigma^2}{B}$  for all  $B \in \mathbb{N}$ . In practice, this conclusion seems, at first, not to be as interesting as originally intended since we do not usually have access to multiple training sets. However, resampling methods can be used to generate multiple **bagging training sets** from the original training set  $\text{Tr}$ .

Let  $B > 1$  be an integer. We generate  $B$  bootstrapped training sets from  $\text{Tr}$  by sampling  $N = |\text{Tr}|$  observations from  $\text{Tr}$ , with replacement, to yield

$$\text{Tr}_1, \dots, \text{Tr}_B,$$

and train a model  $\hat{f}_i$  (for regression) or  $\hat{C}_i$  (for classification) on each  $\text{Tr}_i$ ,  $i = 1, \dots, B$ ; for each  $\mathbf{x}^* \in \text{Te}$ , we then have  $B$  predictions

$$\begin{aligned} \hat{f}_1(\mathbf{x}^*), \dots, \hat{f}_B(\mathbf{x}^*) & \quad (\text{for regression}) \\ \hat{C}_1(\mathbf{x}^*), \dots, \hat{C}_B(\mathbf{x}^*) & \quad (\text{for classification}). \end{aligned}$$



The **bagging prediction** at  $\mathbf{x}^* \in \text{Te}$  is the average of all predictions

$$\hat{f}_{\text{Bag}}(\mathbf{x}^*) = \frac{1}{B} \sum_{i=1}^B \hat{f}_i(\mathbf{x}^*) \quad (\text{for regression}),$$

or the most frequent prediction

$$\hat{C}_{\text{Bag}}(\mathbf{x}^*) = \text{Mode}\{\hat{C}_1(\mathbf{x}^*), \dots, \hat{C}_B(\mathbf{x}^*)\} \quad (\text{for classification}).$$

Bagging is particularly helpful in the CART framework; to take full advantage of bagging, however, the trees should be grown **deep** (i.e., no pruning), as their complexity will lead to high variance but low bias (thanks to the bias-variance trade-off).

In practice, the bagged tree predictions would also have low bias, but the variance will be reduced by the bagging process; bagging with 100s/1000s of trees typically produces greatly improved predictions (at the cost of interpretability, however).

**Out-of-Bag Error Estimation** As is usually the case with supervised models, we will need to estimate the test error for a bagged model. There is an easy way to provide the estimate without relying on cross-validation, which is computationally expensive when  $N$  is large.

The  $j$ th model is fit to the bootstrapped training set  $\text{Tr}_j$ ,  $j = 1, \dots, B$ . We can show that, on average, each of the  $\text{Tr}_j$  contains  $\approx 2/3$  distinct observations of  $\text{Tr}$ , which means that  $\approx 1/3$  of the training observations are not used to build the model (we refer to those observations are **out-of-bag (OOB) observations**).

We can then predict the response  $y_i$  for the  $i$ th observation in  $\text{Tr}$  using only those models for which  $\mathbf{x}_i$  was OOB; there should be about  $B/3$  such predictions, and

$$\hat{y}_i = \text{Avg}\{\hat{f}_j(\mathbf{x}_i) \mid \mathbf{x}_i \in \text{OOB}(\text{Tr}_j) = \text{Tr} \setminus \text{Tr}_j\} \quad (\text{for regression})$$

$$\hat{y}_i = \text{Mode}\{\hat{C}_j(\mathbf{x}_i) \mid \mathbf{x}_i \in \text{OOB}(\text{Tr}_j)\} \quad (\text{for classification}).$$

The OOB MSE (or the OOB misclassification rate) are thus good  $\text{Te}$  error estimates since none of the predictions are given by models that used the test observations in their training.

**Variable Importance Measure** Bagging improves the accuracy of stand-alone models, but such an improvement comes at the cost of **reduced interpretability**, especially in the case of CART: the bagged tree predictions cannot, in general, be expressed with the help of a single tree. In such a tree, the relative importance of the features is linked to the **hierarchy of splits**.<sup>60</sup>

For bagged **regression trees**, a measure such as the total amount in decreased SSE due to splits over a given predictor, in which we compare SSE in trees with these splits against SSE in trees without these splits, averaged over the  $B$  bagged trees, provides a summary of the importance of each variable (large scores indicate important variables). For bagged **classification trees**, we would replace SSE with the **Gini index**, instead.

60: Namely, the most “important” variables appear in the earlier splits.

Another approach might be to weigh the importance of a factor **inversely proportionally** to the level in which it appears (if at all) in each bagging tree and to average over all bagging trees.

For instance, if predictor  $X_1$  appears in the 1st split level of bagged tree 1, the 4th split level of bagged tree 2, and the 3rd split level of bagged tree 5, whereas predictor  $X_2$  appears in the 2nd, 2nd, 3rd, and 5th split levels of bagged trees 2, 3, 4, 5 (resp.), then the relative importance of each predictor over the 5 bagged trees is

$$X_1 : (1 + 1/4 + 0 + 0 + 1/3) \cdot 1/5 = 19/60 = 0.32$$

$$X_2 : (0 + 1/2 + 1/2 + 1/3 + 1/5) \cdot 1/5 = 23/75 = 0.31;$$

the first variable is nominally more important than the second.

**Example** We once again revisit the Iowa Housing Price example of Sections 20.5.2 and 21.4.1. Recall that we had built a training set `dat.train` with  $n = 1160$  observations relating to the selling price `SalePrice` of houses in Ames, Iowa.

```
dat.Housing = read.csv("VE_Housing.csv", header=TRUE,
                      stringsAsFactors = TRUE)
missing = attributes(which(apply(is.na(dat.Housing), 2,
                                sum)>0))$names
dat.Housing.new = dat.Housing[,!colnames(dat.Housing)
                              %in% missing]
dat.Housing.new = subset(dat.Housing.new, select = -c(Id))
set.seed(1234) # for replicability
n.train = 1160
ind.train = sample(1:nrow(dat.Housing.new), n.train)
dat.train = dat.Housing.new[ind.train,]
dat.test = dat.Housing.new[-ind.train,]
```

We build a regression tree bagging model using the R package `ipred`, with 150 bags, and using an OOB error estimate.

```
set.seed(1234)
library(ipred)
(bag <- bagging(formula = SalePrice ~ ., data = dat.train,
                nbagg = 150, coob = TRUE, control =
                  rpart::rpart.control(minsplit = 5, cp = 0),
                importance = TRUE))
```

Bagging regression trees with 150 bootstrap replications  
Out-of-bag estimate of root mean squared error: 29.2526

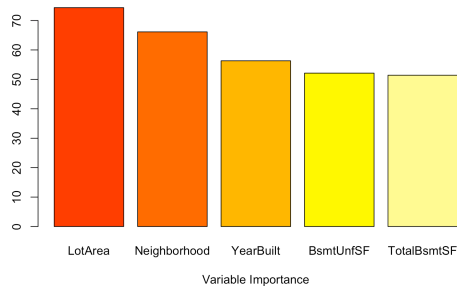
We can display the 5 most important variables:

```
p=ncol(dat.train)-1
vim <- data.frame(var=names(dat.train[,1:p]),
                  imp=caret::varImp(bag))
```

```

vim.plot <- vim[order(vim$Overall, decreasing=TRUE),]
vim.plot <- vim.plot[1:5,]
barplot(vim.plot$Overall, names.arg=rownames(vim.plot),
        col=heat.colors(5), xlab='Variable Importance')

```



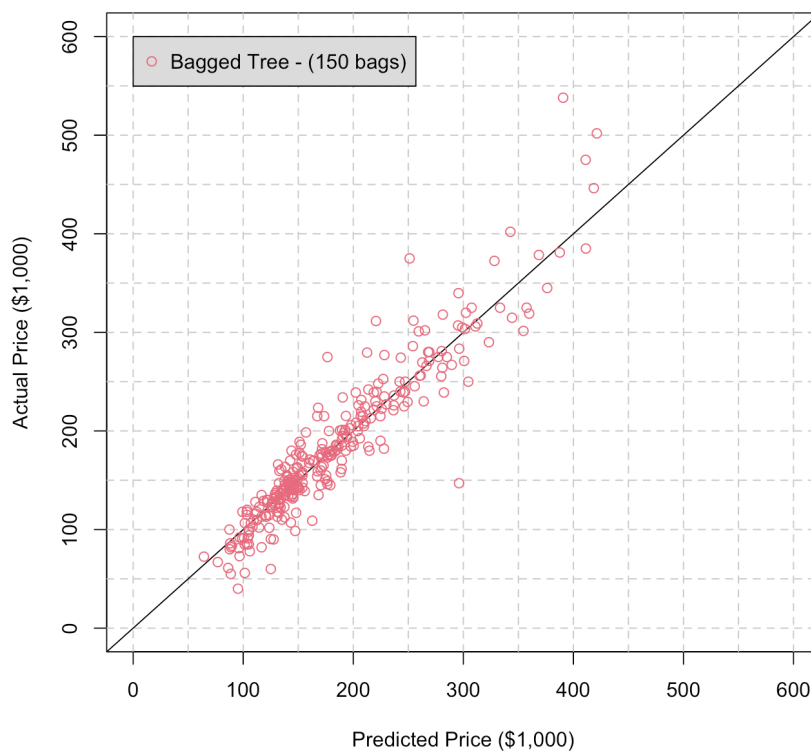
The predictions on the testing data are obtained (and plotted) as follows:

```

yhat.bag = predict(bag, newdata=dat.test)

xlimit = ylimit = c(0,600)
plot(NA, col=2, xlim=xlimit, ylim=ylim,
     xlab="Predicted Price ($1,000)",
     ylab="Actual Price ($1,000)")
abline(h=seq(0,600, length.out=13), lty=2, col="grey",
       v=seq(0,600, length.out=13))
abline(a=0, b=1)
points(yhat.bag, dat.test$SalePrice, col=2)
legend(0,600, legend=c("Bagged Tree - (150 bags)"),
      col=c(2), pch=rep(1), bg='light grey')

```



The correlation measure between predicted and actual sale prices on the `dat.test` is:

```
cor(yhat.bag, dat.test$SalePrice)
```

```
[1] 0.9413044
```

How does that compare to the previous MARS and CART models?

### 21.5.2 Random Forests

In a bagging procedure, we fit models on various training sets, and we use the central limit theorem, assuming independence of the models, to reduce the variance.

In practice, however, the independence assumption is rarely met: if there are only a few strong predictors in  $Tr$ , each of the bagged models (built on the bootstrapped training sets  $Tr_i$ ) is likely to be similar to the others, and the various bagging predictions are unlikely to be un-correlated, so that

$$\text{Var}(\hat{y}_{\text{Bag}}^*) \neq \frac{\sigma^2}{B}, \quad \mathbf{x}^* \in Te;$$

averaging highly correlated quantities does not reduce the variance significantly.<sup>61</sup>

61: The central limit theorem assumption of independence of observations is necessary.

With a small tweak, however, we can **decorrelate the bagged models**, leading to variance reduction when we average the (bagged) predictions. **Random forests** also build models on  $B$  bootstrapped training samples, but each model is built out of a **random subset** of predictors.

For decision trees, every time a split is considered, the set of allowable predictors is selected from a random subset of  $m$  predictors out of the full  $p$  predictors. By selecting predictors randomly for each model, we lose out on building the best possible model on each training sample, but we also reduce the chance of them being correlated. For a test observation  $\mathbf{x}^*$ , the  $B$  predictions are combined as in bagging to yield the **random forest prediction**.

If  $m = p$ , random forests reduce to bagged models; in practice we use  $m \approx \sqrt{p}$  for classification and  $m \approx p/3$  for regression. When the predictors are highly correlated, however, smaller values of  $m$  are recommended.

**Example** We revisit the Wine example of Section 21.4.3, using the R package `randomForest`.

```
wine = read.csv("wine.csv", header=TRUE,
               stringsAsFactors = TRUE)
wine$Class = as.factor(wine$Class)
```

Let's implement a 70%/30% training/testing split:

```
set.seed(1111)
ind.train <- sample(nrow(wine), 0.8*nrow(wine),
                    replace = FALSE)
dat.train <- wine[ind.train,]
dat.test  <- wine[-ind.train,]
```

There are  $p = 13$  predictors in the dataset, so we should use  $m \approx \sqrt{13} \approx 4$  predictors at each split. Keep in mind, however, that we have seen that some of the variables are correlated, so we will try models for  $m = 1$ ,  $m = 2$ ,  $m = 3$ , and  $m = 4$ .

```
wine.rf.1 <- randomForest::randomForest(Class ~ .,
    data = dat.train, ntree = 500, mtry = 1,
    importance = TRUE)
wine.rf.1
```

```
No. of variables tried at each split: 1
      OOB estimate of  error rate: 0.7%
Confusion matrix:
      1  2  3 class.error
1 47  0  0  0.00000000
2  0 54  1  0.01818182
3  0  0 40  0.00000000
```

```
wine.rf.2 <- randomForest::randomForest(Class ~ .,
    data = dat.train, ntree = 500, mtry = 2,
    importance = TRUE)
wine.rf.2
```

```
No. of variables tried at each split: 2
      OOB estimate of  error rate: 0.7%
Confusion matrix:
      1  2  3 class.error
1 47  0  0  0.00000000
2  0 54  1  0.01818182
3  0  0 40  0.00000000
```

```
wine.rf.3 <- randomForest::randomForest(Class ~ .,
    data = dat.train, ntree = 500, mtry = 3,
    importance = TRUE)
wine.rf.3
```

```
No. of variables tried at each split: 3
      OOB estimate of  error rate: 1.41%
Confusion matrix:
      1  2  3 class.error
1 46  1  0  0.02127660
2  0 54  1  0.01818182
3  0  0 40  0.00000000
```

```
wine.rf.4 <- randomForest::randomForest(Class ~ .,
  data = dat.train, ntree = 500, mtry = 4,
  importance = TRUE)
wine.rf.4
```

```
No. of variables tried at each split: 4
      OOB estimate of error rate: 1.41%
Confusion matrix:
      1  2  3 class.error
1 46  1  0  0.02127660
2  0 54  1  0.01818182
3  0  0 40  0.00000000
```

In this example, then, the choice of  $m$  only introduces slight differences. Obviously, this will not always be the case.

### 21.5.3 Boosting

Another general approach to improving prediction results for statistical learners involves creating a sequence of models, each improving over the previous model in the series. **Boosting** does not involve bootstrap sampling; instead, it fits models on a hierarchical sequence of **residuals**, but it does so in a **slow manner**.

For regression problems, we proceed as follows:

1. set  $\hat{f}(\mathbf{x}) = 0$  and  $r_i = y_i$  for all  $\mathbf{x}_i \in \text{Tr}$ ;
2. for  $b = 1, 2, \dots, B$ :
  - i. fit a model  $\hat{f}^b$  to the training set  $(\mathbf{X}, \mathbf{r})$ ;
  - ii. update the regression function  $\hat{f} := \hat{f} + \lambda \hat{f}^b$ ;
  - iii. update the residuals  $r_i := r_i - \lambda \hat{f}^b(\mathbf{x}_i)$  for all  $\mathbf{x}_i \in \text{Tr}$ ;
3. output the boosted model  $\hat{f}(\mathbf{x}) = \lambda(\hat{f}^1(\mathbf{x}) + \dots + \hat{f}^B(\mathbf{x}))$ .

In this version of the algorithm, boosting requires three tuning parameters:

- the **number of models**  $B$ , which can be selected through cross-validation (boosting can overfit if  $B$  is too large);
- the **shrinkage parameter**  $\lambda$  (typically,  $0 < \lambda \ll 1$ ), which controls the **boosting learning rate** (a small  $\lambda$  needs a large  $B$ , in general); the optimal  $\lambda$  and  $B$  can be found *via* cross-validation, and
- although not explicitly stated, we also need the learning models to reach some **complexity threshold**.

Variants of the boosting algorithm allowing for classification and for varying weights depending on performance regions in predictor space also exist and are quite popular. While the various *No Free Lunch* theorems guarantee that no supervised learning algorithm is always best regardless of context/data, the combination of AdaBoost with weak CART learners is seen by many as the best “out-of-the-box” classifier.

**Example** Consider the [Credit.csv](#) dataset [3]; the task is to determine the credit card balance based on a number of other factors.

```
str(Credit)
```

```
'data.frame':  400 obs. of  12 variables:
 $ X          : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Income     : num  14.9 106 104.6 148.9 55.9 ...
 $ Limit      : int  3606 6645 7075 9504 4897 8047 3388 7114 3300 6819 ...
 $ Rating     : int  283 483 514 681 357 569 259 512 266 491 ...
 $ Cards      : int  2 3 4 3 2 4 2 2 5 3 ...
 $ Age        : int  34 82 71 36 68 77 37 87 66 41 ...
 $ Education  : int  11 15 11 11 16 10 12 9 13 19 ...
 $ Gender     : Factor w/ 2 levels "Female","Male": 2 1 2 1 2 2 1 2 1 1 ...
 $ Student    : Factor w/ 2 levels "No","Yes": 1 2 1 1 1 1 1 1 2 ...
 $ Married    : Factor w/ 2 levels "No","Yes": 2 2 1 1 2 1 1 1 1 2 ...
 $ Ethnicity  : Factor w/ 3 levels "African American",...: 3 2 2 2 3 3 1 2 3 1 ...
 $ Balance    : int  333 903 580 964 331 1151 203 872 279 1350 ...
```

We remove the index variable, and create binary variables for all categorical levels in the data.

```
Credit <- Credit[,-c(1)]
Credit$Gender.dummy <- ifelse(
  Credit$Gender == "Female",1,0)
Credit$Student.dummy <- ifelse(
  Credit$Student == "Yes",1,0)
Credit$Married.dummy <- ifelse(
  Credit$Married == "Yes",1,0)
Credit$Ethnicity.AA.dummy <- ifelse(
  Credit$Ethnicity == "African American",1,0)
Credit$Ethnicity.A.dummy <- ifelse(
  Credit$Ethnicity == "Asian",1,0)
```

The dataset under consideration will then have the following shape:

```
Credit <- Credit[,c(1:6,12:16,11)]
str(Credit)
```

```
'data.frame':  400 obs. of  12 variables:
 $ Income      : num  14.9 106 104.6 148.9 55.9 ...
 $ Limit       : int  3606 6645 7075 9504 4897 8047 3388 7114 3300 6819 ...
 $ Rating      : int  283 483 514 681 357 569 259 512 266 491 ...
 $ Cards       : int  2 3 4 3 2 4 2 2 5 3 ...
 $ Age         : int  34 82 71 36 68 77 37 87 66 41 ...
 $ Education   : int  11 15 11 11 16 10 12 9 13 19 ...
 $ Gender.dummy : num  0 1 0 1 0 0 1 0 1 1 ...
 $ Student.dummy : num  0 1 0 0 0 0 0 0 0 1 ...
 $ Married.dummy : num  1 1 0 0 1 0 0 0 0 1 ...
 $ Ethnicity.AA.dummy : num  0 0 0 0 0 0 1 0 0 1 ...
 $ Ethnicity.A.dummy : num  0 1 1 1 0 0 0 1 0 0 ...
 $ Balance     : int  333 903 580 964 331 1151 203 872 279 1350 ...
```

We pick 300 of the 400 observations to be part of the training set:

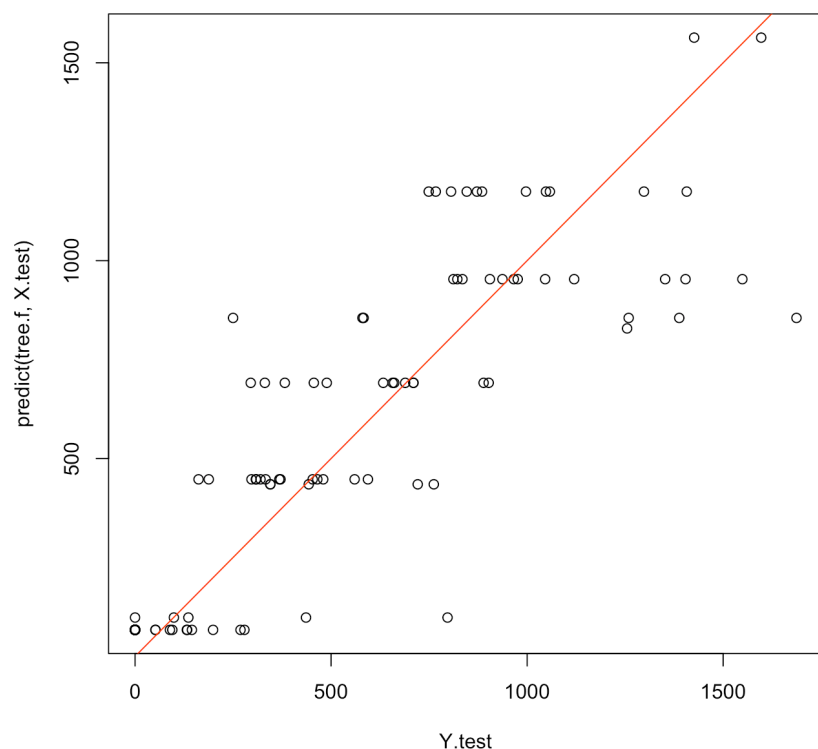
```
set.seed(1234)
ind = sample(1:nrow(Credit), size = 300)
Credit.train = Credit[ind,]
Credit.test = Credit[-ind,]
```

We use  $\lambda = 0.005$  as a shrinkage parameter and  $B = 2000$  models.

```
lambda = 0.005
X <- Credit.train[,1:11] # predictors
r <- Credit.train[,12]   # response
X.test <- Credit.test[,1:11]
Y.test = Credit.test[,12]
```

We start by building the first iteration of the boosting model, using R's tree package, and look at its predictions on the test data:

```
tree.f <- tree::tree(r ~ ., data = data.frame(cbind(X,r)),
                    na.action=na.pass)
r <- r - lambda * predict(tree.f, X)
plot(Y.test, predict(tree.f, X.test))
abline(0,1,col="red")
```



Visually, the performance seems middling, which is born by the correlation metric between the actual test observations and the predicted test observations.



```
cor(Y.test,predict(tree.f, X.test))
```

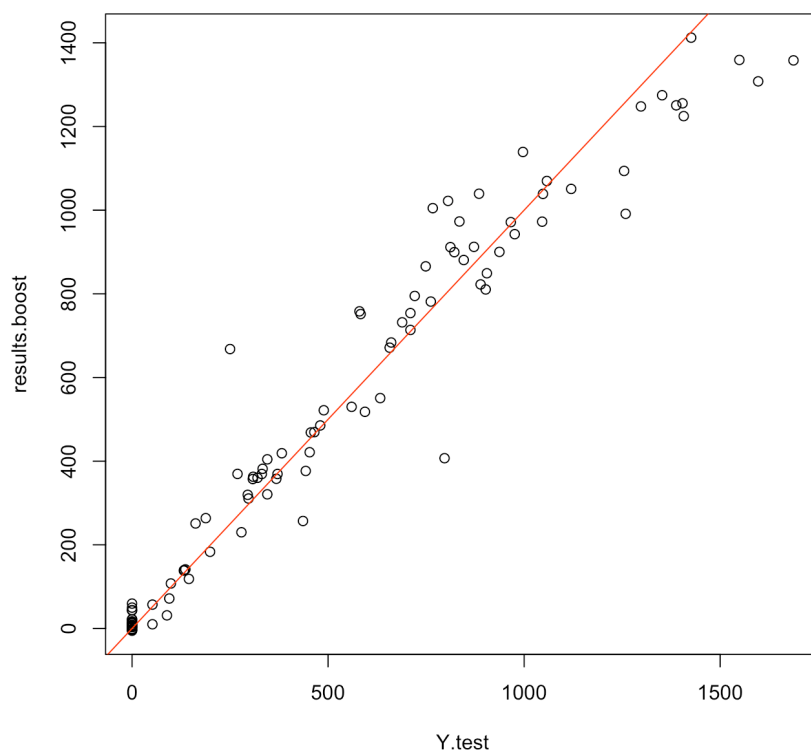
```
[1] 0.8640351
```

Let us compare with the results of boosting with 2000 models.

```
results.boost <- 0*Y.test
B=2000
tree.full <- c()
tree.snipped <- c()

for(b in 1:B){
  tree.full[[b]] <- tree::tree(r ~ .,
    data = data.frame(cbind(X,r)), na.action=na.pass)
  tree.snipped[[b]] <- tree::tree(r ~ .,
    data = data.frame(cbind(X,r)), na.action=na.pass)
  r <- r - lambda * predict(tree.snipped[[b]], X)
  results.boost = results.boost + lambda *
    predict(tree.snipped[[b]], X.test)
}

plot(Y.test,results.boost)
abline(0,1,col="red")
```



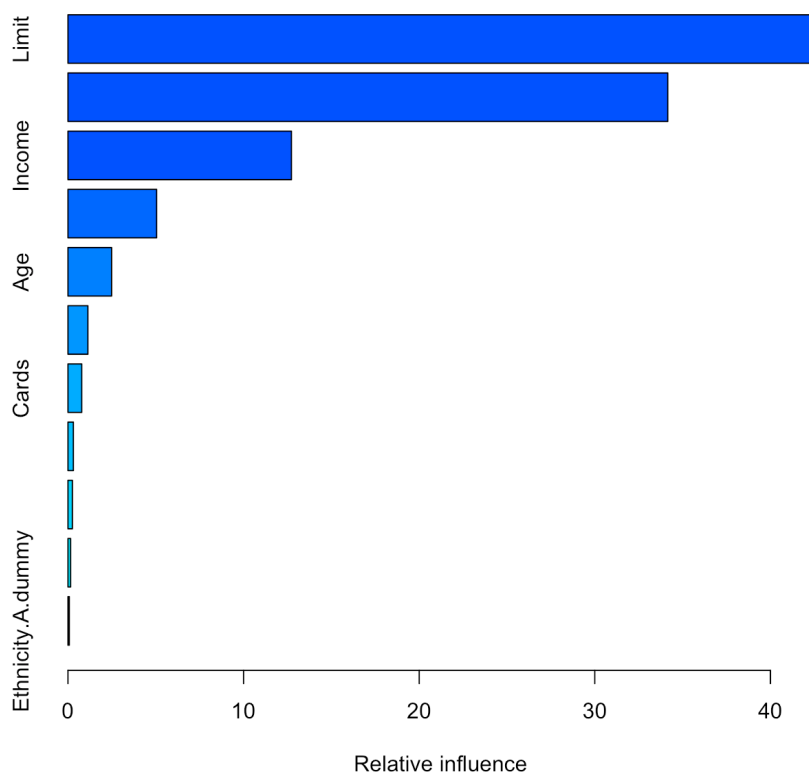
Visually, the performance is much improved; the correlation metric also agrees:

```
cor(Y.test, results.boost)
```

```
[1] 0.9734103
```

We can also use the pre-built `gbm` package to achieve sensibly the same results, and get the influential predictors as a bonus:

```
boost.credit = gbm::gbm(Balance~., data = Credit.train,
                        distribution = "gaussian", n.trees = 10000,
                        shrinkage = 0.01, interaction.depth = 4)
```



```
summary(boost.credit)
```

```

      var      rel.inf
    Limit 42.87367610
    Rating 34.15766697
    Income 12.72692829
 Student.dummy 5.04913001
      Age  2.48572012
 Education 1.13432102
    Cards 0.78545388
Ethnicity.AA.dummy 0.30692238
Married.dummy 0.25299822
  Gender.dummy 0.15135520
Ethnicity.A.dummy 0.07582781

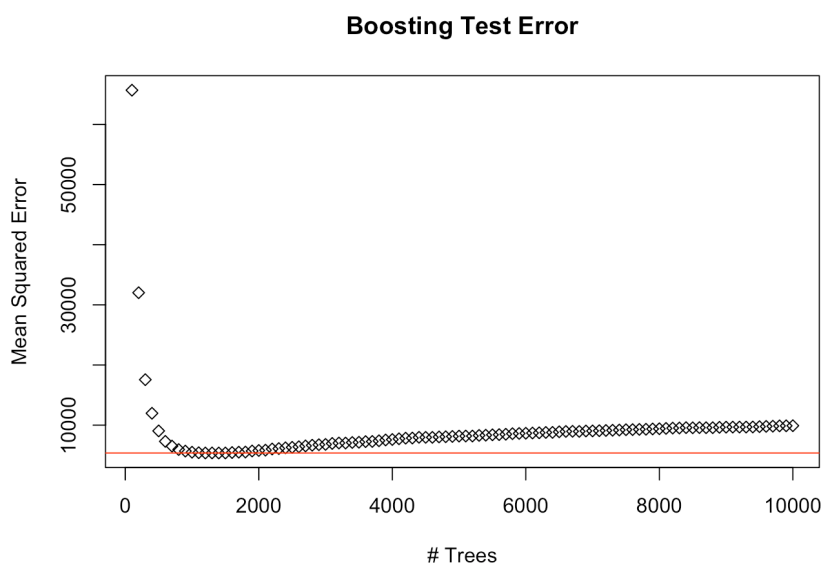
```

Not surprisingly, ethnicity and gender have very little influence on the model.

In order to determine the optimal number of models  $B$  to use,<sup>62</sup> we seek to minimize the prediction MSE:

62: Is it really necessary to run 10,000 models?

```
n.trees = seq(from = 100, to = 10000, by = 100)
predmat = predict(boost.credit, newdata = Credit.test,
                  n.trees = n.trees)
boost.err = with(Credit.test,
                apply( (predmat - Balance)^2, 2, mean) )
plot(n.trees, boost.err, pch = 23, xlab = "# Trees",
     ylab = "MSE", main = "Boosting Test Error")
abline(h = min(boost.err), col = "red")
```



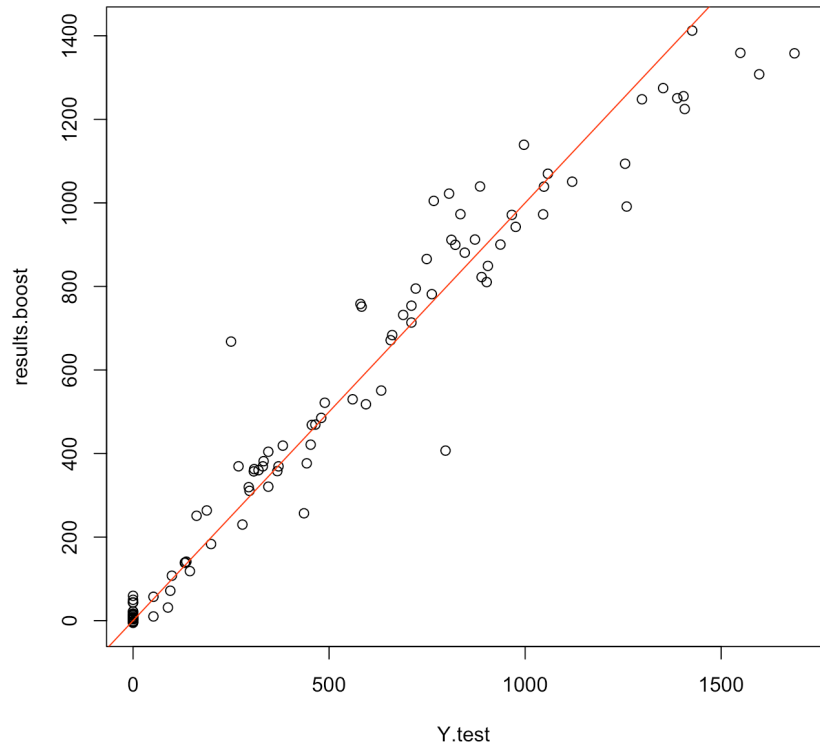
```
which(boost.err==min(boost.err))
```

1200

12

The optimal gbm boosted model (with parameters as in the `gbm()` call above) is thus:

```
results.boost.gbm = predict(boost.credit,
                           newdata = Credit.test, n.trees = 1200)
plot(Y.test, results.boost)
abline(0,1,col="red")
```



```
cor(Y.test, results.boost)
```

```
[1] 0.9734103
```

63: Such as “stubby” trees or “coarse” linear models.

**AdaBoost Adaptive Boosting** (AdaBoost) adapts dynamic boosting to a set of models in order to minimize the error made by the individual weak models.<sup>63</sup> The “adaptive” part means that any new weak learner that is added to the boosted model is **modified to improve the predictive power** on instances which were “mis-predicted” by the (previous) boosted model.

The main idea behind **dynamic boosting** is to consider a **weighted sum** of weak learners whose weights are adjusted so that the prediction error is minimized.

Consider a binary classification context, where

$$\text{Tr} = \{(\mathbf{x}_i, y_i) \mid i = 1, \dots, N\}, \quad \text{and } y_i \in \{-1, +1\} \quad \forall i = 1, \dots, N.$$

The **boosted classifier** is a function

$$F(\mathbf{x}) = \sum_{b=1}^B c_b f_b(\mathbf{x}),$$

where  $f_b(\mathbf{x}) \in \{-1, +1\}$  and  $c_b \in \mathbb{R}$  for all  $b$  and all  $\mathbf{x}$ . The **class prediction** at  $\mathbf{x}$  is simply  $\text{sign}(F(\mathbf{x}))$ .

The **AdaBoost contribution** comes in at the modeling stage, where, for each  $\mu \in \{1, \dots, B\}$  the weak learner  $f_\mu$  is trained on a weighted version

of the original training data, with observations that are misclassified by the partial boosted model

$$F_{\mu}(\mathbf{x}) = \sum_{b=1}^{\mu-1} c_b f_b(\mathbf{x})$$

given larger weights; AdaBoost estimates the weights  $w_i$ ,  $i = 1, \dots, N$  at each of the boosting steps  $b = 1, \dots, B$ .

**Real AdaBoost** [253] is a generalization of this approach which does away with the constants  $c_b$ :

1. Initialize the weights  $\mathbf{w}$ , with  $w_i = 1/N$ , for  $1 \leq i \leq N$ ;
2. For  $b = 1, \dots, B$ , repeat the following steps:
  - i. fit the class probability estimate  $p_m(\mathbf{x}) = P(y = 1 \mid \mathbf{x}, \mathbf{w})$ , using the weak learner algorithm of interest;
  - ii. define  $f_b(\mathbf{x}) = \frac{1}{2} \log \frac{p_b(\mathbf{x})}{1-p_b(\mathbf{x})}$ ;
  - iii. set  $w_i \leftarrow w_i \exp\{-y_i f(\mathbf{x}_i)\}$ , for  $1 \leq i \leq N$ ;
  - iv. renormalize so that  $\|\mathbf{w}\|_1 = 1$ ;
3. Output the classifier

$$F(\mathbf{x}) = \text{sign} \left\{ \sum_{b=1}^B f_b(\mathbf{x}) \right\}.$$

For regression tasks, this procedure must be modified to some extent (in particular, the equivalent task of assigning larger weights to currently misclassified observations at a given step is to train the model to predict (shrunk) residuals at a given step... more or less).

Since boosting is susceptible to overfitting (unlike bagging and random forests), the optimal number of boosting steps  $B$  should be derived from cross-validation.

**Example** The Python library `scikit-learn` provides a useful implementation of AdaBoost. In order to use it, a base estimator (that is to say, a weak learner) must first be selected.

In what follows, we will use a decision tree classifier. Once this is achieved, a `AdaBoostClassifier` object is created, to which is fed the weak learner, the **number of estimators** and the **learning rate** (a.k.a. the shrinkage parameter, which we have seen is a small positive number).

In general, small learning rates require a large number of estimators to provide adequate performance. By default, `scikit-learn`'s implementation uses 50 estimators with a learning rate of 1.

We use the classic *Two-Moons* dataset consisting of two interleaving half circles with added noise, in order to test and compare classification results for AdaBoost (and eventually Gradient Boosting, see below).

This dataset is conveniently built into `scikit-learn` and accessible *via* `make_moons()`, which returns a data matrix  $X$  and a label vector  $y$ . We can treat the dataset as a complete training set as we eventually use **cross-validation** to estimate the test error.<sup>64</sup>

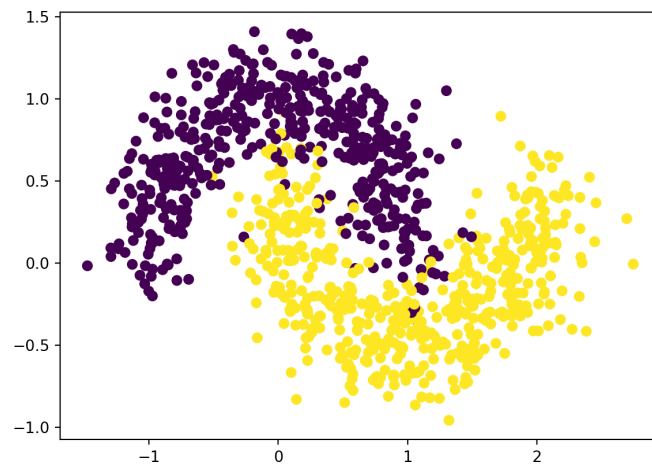
64: The AdaBoost code on the Two-Moons dataset was lifted from an online source whose location cannot be recovered at the moment.

```

from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_moons
N = 1000
X,Y = make_moons(N,noise=0.2)
plt.scatter(X[:,0],X[:,1], c=Y)
plt.show()

```



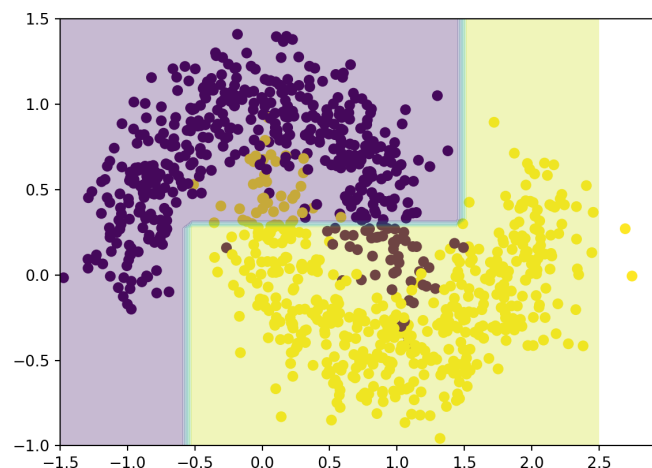
65: This same structure will later be used for our weak learners.

We first attempt to classify the data using `DecisionTreeClassifier()` with a maximum depth of 3.<sup>65</sup>

```

clf = DecisionTreeClassifier(max_depth=3)
clf.fit(X,Y)
xx,yy = np.meshgrid(np.linspace(-1.5,2.5,50),
                    np.linspace(-1,1.5,50))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.scatter(X[:,0],X[:,1], c = Y)
plt.contourf(xx,yy,Z,alpha=0.3)
plt.show()

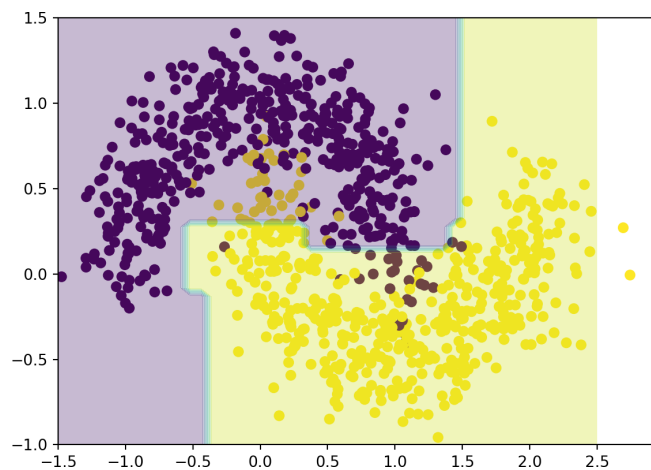
```



As can be seen from the display, this single decision tree does not provide the best of fits.

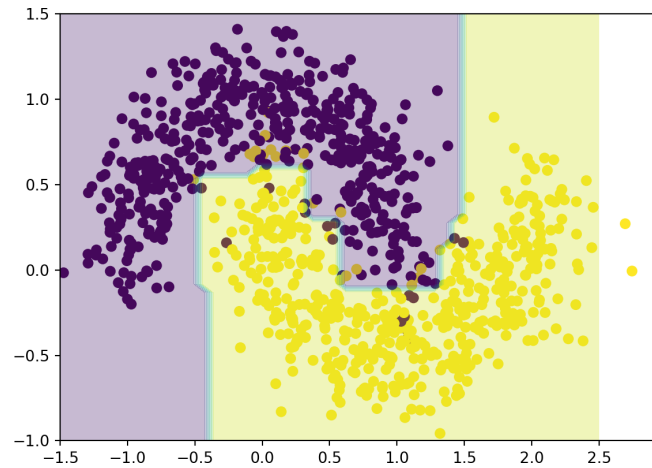
Next, we build an AdaBoost classifier. We first consider a model with  $B = 5$  decision trees, and with a learning rate  $\lambda = 1/10$ .

```
ada = AdaBoostClassifier(clf, n_estimators=5,
                        learning_rate=0.1)
ada.fit(X,Y)
xx,yy = np.meshgrid(np.linspace(-1.5,2.5,50),
                    np.linspace(-1,1.5,50))
Z = ada.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.scatter(X[:,0],X[:,1], c = Y)
plt.contourf(xx,yy,Z,alpha=0.3)
plt.show()
```



Finally, we build an AdaBoost classifier with  $B = 10$  decision trees and with a learning rate  $\lambda = 1/10$ .

```
ada = AdaBoostClassifier(clf, n_estimators=10,
                        learning_rate=0.1)
ada.fit(X,Y)
xx,yy = np.meshgrid(np.linspace(-1.5,2.5,50),
                    np.linspace(-1,1.5,50))
Z = ada.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.scatter(X[:,0],X[:,1], c = Y)
plt.contourf(xx,yy,Z,alpha=0.3)
plt.show()
```



The AdaBoosted tree is better at capturing the dataset's structure. Of course, until we evaluate the performance on an independent test set, this could simply be a consequence of overfitting (one of AdaBoost's main weaknesses, as the procedure is sensitive to outliers and noise). We can guard against this eventuality by adjusting the learning rate (which provides a step size for the algorithm's iterations).

To find the optimal value for the learning rate and the number of estimators, one can use the `GridSearchCV` method from `sklearn.model_selection`, which implements cross-validation on a grid of parameters. It can also be parallelized, in case the efficiency of the algorithm should ever need improving (that is not necessary on such a small dataset, but could prove useful with larger datasets).

```
import random
random.seed(10)

from sklearn.model_selection import GridSearchCV
params = {
    'n_estimators': np.arange(10,300,10),
    'learning_rate': [0.01, 0.05, 0.1, 1],
}

grid_cv = GridSearchCV(AdaBoostClassifier(),
                        param_grid= params, cv=5, n_jobs=1)
grid_cv.fit(X,Y)
grid_cv.best_params_
```

```
{'learning_rate': 0.05, 'n_estimators': 200}
```

The results show that, given the selected grid search ranges, the optimal parameters (those that provide the best cross-validation fit for the data) are 200 estimators with a learning rate of 0.05 (these parameters change if `cv` and `n_jobs` are modified, and with different random seeds).

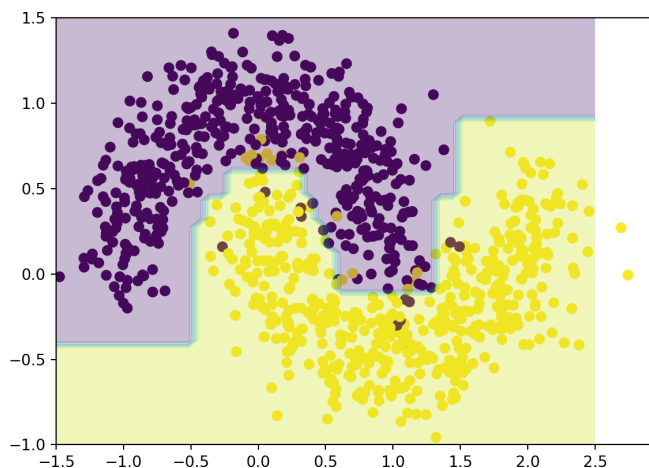
The plot of the model with these parameters indeed shows that the fit looks quite acceptable.



```

ada = grid_cv.best_estimator_
xx,yy = np.meshgrid(np.linspace(-1.5,2.5,50),
                    np.linspace(-1,1.5,50))
Z = ada.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.scatter(X[:,0],X[:,1], c = Y)
plt.contourf(xx,yy,Z,alpha=0.3)
plt.show()

```



**Gradient Boosting** The implementation of **gradient boosting** is simpler than that of AdaBoost. The idea is to first fit a model, then to compute the residuals generated by this model. Next, a new model is trained, but on the residuals instead of on the original response. The resulting model is added to the first one. Those steps are repeated a sufficient number of times. The final model will be a sum of the initial model and of the subsequent models trained on the chain of residuals.

We will not go into the nitty-gritty of gradient boosting here (see [211] for details), but we showcase how it would be applied on the Two-Moons dataset.<sup>66</sup>

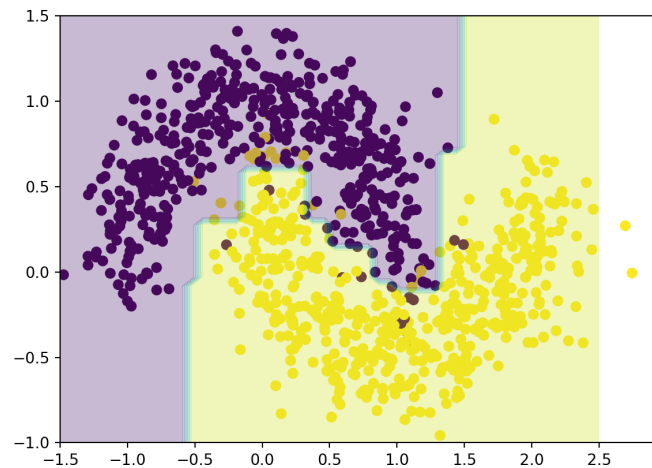
```

from sklearn.ensemble import GradientBoostingClassifier
gbc = GradientBoostingClassifier(n_estimators=9,
                                learning_rate=0.5)

gbc.fit(X,Y)
xx,yy = np.meshgrid(np.linspace(-1.5,2.5,50),
                    np.linspace(-1,1.5,50))
Z = gbc.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.scatter(X[:,0],X[:,1], c = Y)
plt.contourf(xx,yy,Z,alpha=0.3)
plt.show()

```

<sup>66</sup>: Remember that without an estimate of the test error, we cannot use these classifiers for predictive purposes due to avoid overfitting issues.



## 21.6 Exercises

1. Repeat the vowel classification example on PCA-reduced data.
2. Conduct a pre-analysis exploration as in the Wine example to remove variables in the 2011 Gapminder, the Iowa Housing, and the Vowel datasets before conducting the analyses, as in the examples.
3. Construct and evaluate naïve Bayes classifiers for the Wine and for the 2011 Gapminder dataset.
4. Construct and evaluate CART models for the Wine and for the Wisconsin Breast Cancer datasets.
5. Construct and evaluate ANN models for the 2011 Gapminder, for the Iowa Housing, for the Vowel, and for the Wisconsin Breast Cancer datasets.
6. Re-run the ANN models incorporating 10 hidden layers with 30 nodes. How much more time does it take to run a “bigger” neural network on the Wine dataset?
7. Build bagging models for the 2011 Gapminder, Wisconsin Breast Cancer, and Wine datasets.
8. Build random forest models for the 2011 Gapminder, Wisconsin Breast Cancer, and Iowa Housing datasets.
9. Build boosted models for the 2011 Gapminder, Wisconsin Breast Cancer, Wine, and Iowa Housing datasets.
10. Build classification models for the datasets
  - [GlobalCitiesPBI.csv](#)
  - [2016collisionsfinal.csv](#)
  - [polls\\_us\\_election\\_2016.csv](#)
  - [HR\\_2016\\_Census\\_simple.xlsx](#)
  - [UniversalBank.csv](#) .

and/or any other datasets of interest. You may need to identify/define a categorical response variable first.