input and adopt the mean misclassification cost as the performance measure to minimize during model creation. Several approaches to achieving cost sensitivity are discussed in Chapter 6.

1.4 Regression

Similar to classification, regression is an inductive learning task that has been extensively studied and can be widely encountered in practical applications. It can be informally characterized as "classification with continuous classes," which means that regression models predict numerical values rather than discrete class labels. This relationship to the classification task makes it possible to describe the regression task by referring to the latter where appropriate and highlighting the differences where necessary.

The term "regression" tends to be sometimes used in a narrow technical meaning referring to statistical algorithms for fitting parametric regression models. We adopt here a broader view in which regression is presented in a completely algorithm-independent way as one of the major data mining tasks, and any algorithm that solves this task will be considered a regression algorithm. This makes regression equivalent to *numerical prediction*, practical instantiations of which are nearly as common as those of classification. In particular, we might want to predict prices, demand, production or sales volumes, resource consumption, physical parameters, etc.

The regression task consists in assigning numerical values to instances from a given domain, described by a set of discrete or continuous-valued attributes. This assignment is supposed to approximate some target function, generally unknown, except for a subset of the domain. This subset can be used to create the regression model, which is a machine-friendly representation of the relationships between the target function and the attributes that makes it possible to predict unknown target function values for any possible instance from the same domain. The regression task adopts therefore the same general scenario of inductive learning that has been presented above for the classification task. In practical applications, the target function represents some interesting property of instances from the domain that is either difficult and costly to determine, or (more typically) that becomes known later than is needed. Subsections below add regression-specific highlights to what has been presented above for inductive learning in general.

1.4.1 Target function

The *target function* $f : X \to \mathcal{R}$ represents the true assignment of numerical values to all instances from the domain. Target function values will briefly be called *target values* or *target labels*.

1.4.2 Training set

The *training set* $T \subseteq D \subset X$ for regression consists of some or all labeled instances for which target function values are available, despite being unknown in general.

Example 1.4.1 As a simple example of a training set for the regression task, consider a modified version of the *weatherc* data, in which the `play` attribute originally representing the target concept is replaced by a new continuous `playability` attribute, which now

represents the target function. This will be referred to as the *weatherr* data. The following R code reads this dataset to an R dataframe and summarizes the attribute distributions.

dmr.data

```
weatherr <- read.table(text="
       outlook temperature humidity   wind playability
   1     sunny          27       80 normal        0.48
   2     sunny          28       65   high        0.46
   3  overcast          29       90 normal        0.68
   4     rainy          21       75 normal        0.52
   5     rainy          17       40 normal        0.54
   6     rainy          15       25   high        0.47
   7  overcast          19       50   high        0.74
   8     sunny          22       95 normal        0.49
   9     sunny          18       45 normal        0.64
  10     rainy          23       30 normal        0.55
  11     sunny          24       55   high        0.57
  12  overcast          25       70   high        0.68
  13  overcast          30       35 normal        0.79
  14     rainy          26       85   high        0.33")

summary(weatherr)
```

1.4.3 Model

A *regression model* $h : X \to \mathcal{R}$ can be used to generate numerical predictions for all instances $x \in X$ and supposed to provide a good approximation of the target function f on the whole domain.

1.4.4 Performance

The exact meaning of "good approximation" is established by regression model performance measures, but – informally – we want the model to usually provide predictions that are not far away from the true target values. One commonly adopted performance measure is the mean sum of squared differences between the true and predicted values, referred to as the *mean square error*. This and other regression performance measures are discussed in Chapter 10.

1.4.5 Generalization

Generalization is no less crucial for regression than for classification. Regression algorithms have to not only *discover* relationships between the target function and attribute values in the training set, but also to *generalize* them so that they can be expected to hold on new data.

1.4.6 Overfitting

Poor generalization leads to overfitting, which is the same serious problem for regression as for the classification and can be defined in the same way. Many regression algorithms include mechanisms supposed to reduce the risk of overfitting.

1.4.7 Algorithms

A *regression algorithm* generates a regression model based on a given training set.

1.5 Clustering

Clustering is an inductive learning task that differs from the classification and regression tasks from the same family by the lack of a predetermined target attribute to be predicted. It can be thought of as classification with autonomously discovered rather than predefined classes, which are based on similarity patterns identified in the data.

The clustering task consists in dividing a set of instances from a given domain, described by a number of discrete or continuous-valued attributes, into a set of clusters based on their similarity, and creating a model that can map arbitrary instances from the same domain to these clusters. This can be considered a superposition of two subtasks:

Cluster formation. The identification of similarity-based groups in the analyzed data.

Cluster modeling. Creating a model for cluster membership prediction.

The latter is clearly a classification task, with clusters identified within the first subtask used as classes. This could be performed, in principle, using any available classification algorithm. It is usually more convenient not to separate these two subtasks, though, and most clustering algorithms handle both cluster formation and cluster modeling. It makes it possible for the criteria used to identify clusters to be subsequently reused for cluster membership prediction.

1.5.1 Motivation

The utility of the clustering task may not be as self-evident as for the classification and regression tasks and deserves some more explanation. Some typical reasons to perform clustering are listed below, along with example applications where they are likely to appear.

- Clustering can provide useful insights about the similarity patterns present in the data, and a clustering model can be considered as knowledge *per se*. Some example applications where this is the case include
  - — customer segmentation,
  - — point of sale segmentation,
  - — document catalog creation.
- Clustering can be performed on a selected subset of *observable* attributes that are easily available for all instances, and used to predict *hidden* attributes that are impossible or difficult to determine for some instances based on cluster membership. This is similar to classification or regression with multiple target attributes with sparingly available values. Such situation occurs in the following example applications:
  - — customer clustering based on socio-demographic attributes used to predict attributes describing purchase behavior,
  - — point-of-sale segmentation based on location, building, and local population features, used to predict attributes describing selling performance.

8

Linear regression

8.1 Introduction

Parametric regression is the most direct instantiation of the idea of a parametric model representation, in which the model is represented by a finite number of parameters with a fixed functional form assumed. This is also the most frequently used approach to the regression task, to which such a representation is particularly well suited. Parametric regression algorithms can deliver successful regression models by themselves or in combination with other techniques, including those borrowed from algorithms used for the classification task.

Linear regression is the simplest approach to the regression task based on a parametric model representation. Despite its obvious and unquestionable linearity limitation (being capable of directly approximating linear target functions only), it deserves particular attention due to its algorithmic and computational advantages. Interestingly, it is possible, at least to some extent, to overcome the limitation while retaining the advantages. This chapter covers both plain linear regression and augmented versions thereof, breaking the linearity limitation. The presented discussion of model representation and creation techniques maintains a higher level of generality whenever possible, presenting the particular linear representation as an instantiation of the more general parametric regression approach. Linear model representation and gradient-based parameter estimation have already appeared in Chapter 5 in the context of the classification task, but it is here where they are thoroughly discussed.

Example 8.1.1 Demonstrating linear regression algorithms with R code examples will require the use of DMR packages providing functions for model evaluation, attribute transformation, and simple utilities. Parameter estimation for linear regression models will be illustrated using the *Boston Housing* data, available in the `mlbench` package. The *weatherr* data will be used to illustrate discrete attribute processing. The packages and the datasets are loaded by the R code presented below. The larger of those is partioned into training and tests subsets, with the fourth column – containing a single discrete attribute – skipped.

`dmr.data`

Data Mining Algorithms: Explained Using R, First Edition. Paweł Cichosz.

```
library(dmr.regeval)
library(dmr.trans)
library(dmr.util)

data(weatherr, package="dmr.data")
data(BostonHousing, package="mlbench")

set.seed(12)
rbh <- runif(nrow(BostonHousing))
bh.train <- BostonHousing[rbh>=0.33,-4]
bh.test <- BostonHousing[rbh<0.33,-4]
```

Additionally a small artificial dataset will be used in several examples. It is generated by the following R code:

```
set.seed(12)

  # generate artificial data
lrdat <- data.frame(a1=floor(runif(400, min=1, max=5)),
                    a2=floor(runif(400, min=1, max=5)),
                    a3=floor(runif(400, min=1, max=5)),
                    a4=floor(runif(400, min=1, max=5)))
lrdat$f1 <- 3*lrdat$a1+4*lrdat$a2-2*lrdat$a3+2*lrdat$a4-3
lrdat$f2 <- tanh(lrdat$f1/10)
lrdat$f3 <- lrdat$a1^2+2*lrdat$a2^2-lrdat$a3^2-2*lrdat$a4^2+
            2*lrdat$a1-3*lrdat$a2+2*lrdat$a3-3*lrdat$a4+1
lrdat$f4 <- 2*tanh(lrdat$a1-2*lrdat$a2+3*lrdat$a3-lrdat$a4+1)-
            3*tanh(-2*lrdat$a1+3*lrdat$a2-2*lrdat$a3+lrdat$a4-1)+2

  # training and test subsets
lrdat.train <- lrdat[1:200,]
lrdat.test <- lrdat[201:400,]
```

It generates a dataset of 300 instances described by four continuous attributes, with integer values drawn uniformly at random from the $[1, 5]$ interval, and four different target functions all of which have known functional relationships to these attributes (with only one being linear). This is, of course, completely unrealistic and serves the illustration purpose only. The dataset is then partitioned into training and test subsets, to evaluate the performance of subsequently created models using the simple hold-out procedure. The partitioning is based on instance numbers, which would be normally unacceptable, but is perfectly fine with a randomly generated dataset. A fixed seed of the random number generator is used to ensure the reproducibility of presented results. The mean square error (MSE), defined in Section 10.2.3, will be used as the performance measure for these demonstrations, calculated using the `mse` function.

Ex. 10.2.3
dmr.regeval

8.2 Linear representation

Linear model representation is a particularly simple and popular instantiation of the more general parametric representation family. While focusing on the former, this will also discuss the more general case, which helps one better understand the advantages and limitations of the linear special case.

8.2.1 Parametric representation

A parametric regression model $h : X \to \mathcal{R}$ is described by the following formula, which specifies how to calculate its prediction for instance $x \in X$:

$$h(x) = F(a_1(x), a_2(x), \ldots, a_n(x), w_1, w_2, \ldots, w_N) \tag{8.1}$$

where $a_1, a_2, \ldots, a_n$ are attributes defined on the domain X, $w_1, w_2, \ldots, w_N$ are *model parameters* (also called weights), and F is a predetermined *representation function* that maps attribute value and parameter vectors to real-valued model predictions. This can also be written in a vector form as

$$h(x) = F(\mathbf{a}(x), \mathbf{w}) \tag{8.2}$$

where $\mathbf{a}(x)$ denotes the vector of attribute values for instance x and $\mathbf{w}$ denote the parameter vector. A parametric model is therefore represented by a hypersurface in an $(n+1)$-dimensional space.

The essential feature of parametric representation is using a predetermined representation function, which reduces model creation to estimating model parameters from data. Model representations for which this property does not hold are considered *nonparametric*. This is not supposed to say that they do not have parameters – since they do – but that their representation function needs to be derived from data as well. This is the case, in particular, for regression trees that will be presented in the next chapter.

Example 8.2.1 To illustrate the parametric representation of regression models, the R code presented below defines a function that applies a parametric regression model to generate predictions for a given dataset. The model is assumed to be represented by a list containing two components, named `repf` and `w`. The former is the model's representation function and the latter is its parameter vector. Setting the `class` attribute of such an object to `par` enables appropriate prediction method dispatching.

The representation function takes an attribute value vector and a parameter vector on input and returns the resulting prediction on output. The particular representation function `repf.perf4` defined in this example can be immediately seen to match the `f4` target function in the artificial dataset generated in the previous example:

$$F_4(\mathbf{a}(x), \mathbf{w}) = w_{2n+3} \tanh\left(\sum_{i=1}^{n} w_i a_i(x) + w_{n+1}\right) + w_{2n+4} \tanh\left(\sum_{i=1}^{n} w_{i+n+1} a_i(x) + w_{2n+2}\right) + w_{2n+5} \tag{8.3}$$

where $n = 4$ is the number of attributes. It can therefore be referred to as the *perfect representation function* for `f4`, which – with appropriate parameters – matches the target function exactly. Indeed, when combined with the same parameters as actually used for target function generation it yields the perfect model, achieving a 0 test set error. Note that the `repf.perf4` function is implemented so that it can be applied not only to single instances, but also to complete datasets as well. This is why it uses the `rowSums` function instead of

the `sum` function. The `cmm` utility function is used to multiply all columns of the dataset by the corresponding elements of the parameter vector. Subsequent examples will similarly define functions capable of handling both single and multiple instances whenever possible.

`dmr.util`

```
## parametric regression prediction for a given model and dataset
predict.par <-  function(model, data) { model$repf(data, model$w) }

  # perfect representation function for f4
repf.perf4 <- function(data, w)
{
  w[2*(n <- ncol(data))+3]*tanh(rowSums(cmm(data, w[1:n]))+w[n+1]) +
    w[2*n+4]*tanh(rowSums(cmm(data, w[(n+2):(2*n+1)]))+w[2*n+2]) + w[2*n+5]
}

  # perfect parameters for f4
w.perf4 <- c(1, -2, 3, -1, 1, -2, 3, -2, 1, -1, 2, -3, 2)
  # perfect model for f4
mod.perf4 <- `class<-`(list(w=w.perf4, repf=repf.perf4), "par")
  # test set error
mse(predict(mod.perf4, lrdat.test[,1:4]), lrdat.test$f4)
```

Of course, it is completely unrealistic to assume that either the true representation function or its true parameters are known, as in this example. This assumption is only adopted to illustrate the parametric representation of regression models.

8.2.2 Linear representation function

Linear regression is based on the following special form of a parametric model representation:

$$h(x) = \sum_{i=1}^{n} w_i a_i(x) + w_{n+1} \tag{8.4}$$

The formula specifies how the prediction of a linear model h is calculated for instance $x \in X$. This implicitly assumes that attributes are continuous (so that they can be used for arithmetics). With such a representation, h is linear with respect to both attribute values and parameters and corresponds to a hyperplane in an $(n + 1)$-dimensional space. Unless combined with some additional enhancements, linear regression models can accurately represent only target functions that are linear with respect to attribute values.

The representation function used for linear regression is defined as a linear combination of attribute values and model parameters, with an additional *intercept* parameter. It is a common and convenient practice to avoid explicitly referring to the latter in equations related to linear regression by assuming that there is an additional a_{n+1} attribute defined, always taking a value of 1, which makes it possible to rewrite the model representation formula as

$$h(x) = \sum_{i=1}^{n+1} w_i a_i(x) \tag{8.5}$$

On several occasions, it is also convenient to adopt a vector notation. With $\mathbf{a}(x)$ denoting the vector of attribute values for instance x and $\mathbf{w}$ denoting the parameter vector, we can present

the linear model representation in each of the following two equivalent vector forms:

$$h(x) = \mathbf{w} \circ \mathbf{a}(x) \tag{8.6}$$

$$h(x) = \mathbf{w}^T \mathbf{a}(x) \tag{8.7}$$

where $\circ$ is the dot product operator and T is the matrix transpose operator. The latter assumes that vectors are treated as one-column matrices.

Example 8.2.2 The following R code defines a linear representation function that takes an attribute value vector and a parameter vector on input and returns a linear combination thereof on output. To make it applicable not only to single instances, but also complete datasets, the `rowSums` function is used instead of `sum` and the `cmm` function is used to multiply all columns of the dataset by the corresponding elements of the parameter vector. This representation function is then combined with an appropriate parameter vector to create the perfect model for the `f1` target function from the artificial dataset created in Example 8.1.1, which was indeed generated as a linear combination of attribute values. The perfect model can be verified to achieve a 0 mean square error.

`dmr.util`

```
## linear representation function
repf.linear <- function(data, w)
{ rowSums(cmm(data, w[1:(n <- ncol(data))])) + w[n+1] }

  # perfect parameter vector for f1
w.perf1 <- c(3, 4, -2, 2, -3)
  # perfect model for f1
mod.perf1 <- 'class<-'(list(w=w.perf1, repf=repf.linear), "par")
  # test set error
mse(predict(mod.perf1, lrdat.test[,1:4]), lrdat.test$f1)
```

Representation functions that are nonlinear with respect to attribute values, but linear with respect to model parameters, yield regression models that are still linear in a modified attribute space. They can be presented as

$$h(x) = \sum_{i=1}^{N} w_i a'_i(x) + w_{N+1} \tag{8.8}$$

where $a'_1, a'_2, \ldots, a'_N$ are new modified attributes that are obtained via some nonlinear transformations of the original attributes. This approach is referred to as an enhanced representation and will be further discussed in Section 8.6.2 as one of the possible ways of using linear regression to approximate nonlinear target functions.

A generalized linear representation assumes that the output of a linear model is transformed nonlinearly to generate predictions. A model of this form, although not intrinsically nonlinear, is therefore capable of approximating some nonlinear target functions, as will be further discussed in Section 8.6.1.

8.2.3 Nonlinear representation functions

Only when the representation function is nonlinear with respect to model parameters and cannot be linearized in a straightforward way, we are faced with intrinsically *nonlinear regression*. Typical nonlinear representation functions include:

- polynomial functions,
- exponential functions,
- logarithmic functions,
- trigonometric functions,

as well as various combinations or superpositions thereof. These are beyond our interest in this chapter.

8.3 Parameter estimation

Parameter estimation is the process of identifying model parameters based on a given training set that are likely to yield good prediction performance. This can be viewed as an optimization process in which the space of possible parameter vectors is searched for one that optimizes an adopted performance measure. In general, several different performance measures and optimization methods could be used for this purpose. The mean square error, defined in Section 10.2.3 as the mean squared difference between true target function values and model predictions, is a particularly convenient performance measure to adopt. In this role, it is also referred to as the quadratic loss. Gradient descent methods belong to the simplest approaches to optimization that can be employed for both linear and (some) nonlinear representations. Most of this section is focused on the combination of these two, which is simple enough to be explained and understood with just elementary maths and illustrate with a plain-vanilla implementation. An alternative least-squares method – similarly simple, easier to use, and usually much more efficient (except for excessively large data), but inapplicable to nonlinear representations – will also be discussed.

8.3.1 Mean square error minimization

The definition of the mean square error of the regression model h with respect to the target function f on the training set T can be rewritten in a slightly modified form as follows:

$$E_{T,f}(h) = \frac{1}{2} \sum_{x \in T} (f(x) - h(x))^2 \tag{8.9}$$

The modification of the original definition from Section 10.2.3 consists in replacing the $\frac{1}{|T|}$ coefficient by $\frac{1}{2}$, which will turn out to serve the purpose of aesthetics only. Clearly, any model that minimizes $E_{T,f}(h)$ does also minimize the training set mean square error, so this modification has no impact on the model that could be identified.

The MSE-like function defined above will serve as the objective function for minimization. It may appear at first unreasonable to optimize the training performance, which could easily lead to overfitting, but we actually have no choice here, since the true performance can only be estimated when a model is already built and not during the training process. An appropriately selected model representation function (e.g., without too many parameters) and an optimization technique will have to take the responsibility for overfitting prevention rather than the objective function. An alternative approach to linear model parameter estimation that relaxes the objective of training set error minimization to increase the resistance to overfitting is presented in Section 16.3.

The idea of gradient descent function minimization is to gradually modify the parameter vector by shifting it in the direction indicated by the negated gradient of the function being minimized with respect to the parameters. In our case, it can be written as follows:

$$\mathbf{w} := \mathbf{w} + \beta \left(-\nabla_{\mathbf{w}} E_{T,f}(h)\right) \tag{8.10}$$

where $\beta > 0$ is a *step-size* parameter that controls the amount of the update performed. The gradient $\nabla_{\mathbf{w}} E_{T,f}(h)$ is the vector of partial derivatives of $E_{T,f}(h)$ with respect to model parameters. The above parameter update rule can therefore be rewritten for a single parameter w_i in the following form:

$$w_i := w_i + \beta \left(-\frac{\partial E_{T,f}(h)}{\partial w_i}\right) \tag{8.11}$$

Both the sign and the size of the update performed for each parameter depend on the corresponding partial derivative. A positive derivative value indicates that increasing the parameter would increase the error, and therefore the parameter should be decreased. A negative derivative value similarly leads to increasing parameter. The smaller the absolute derivative value, which may indicate approaching a local minimum, the smaller the update.

8.3.2 Delta rule

The partial derivative of $E_{T,f}(h)$ with respect to w_i can be calculated as follows:

$$\begin{aligned}\frac{\partial E_{T,f}(h)}{\partial w_i} &= \frac{1}{2}\sum_{x\in T} 2(f(x) - h(x))\left(-\frac{\partial h(x)}{\partial w_i}\right) \\ &= \sum_{x\in T}(f(x) - h(x))\left(-\frac{\partial h(x)}{\partial w_i}\right)\end{aligned} \tag{8.12}$$

since the dependence of $E_{T,f}(h)$ on parameters is through the model representation. This makes it clear why the $\frac{1}{2}$ coefficient rather than $\frac{1}{|T|}$ is used in the definition of $E_{T,f}(h)$ for the sake of aesthetics, since it simplified with the 2 from the derivative. After substituting to the update rule this yields

$$w_i := w_i + \beta \sum_{x\in T}(f(x) - h(x))\frac{\partial h(x)}{\partial w_i} \tag{8.13}$$

or, in the equivalent vector form

$$\mathbf{w} := \mathbf{w} + \beta \sum_{x\in T}(f(x) - h(x))\nabla_{\mathbf{w}} h(x) \tag{8.14}$$

The obtained update rule shows how to modify the parameters of a parametric regression model based on the training set in order to decrease the mean square error. It is often convenient to decompose these updates into contributions of single training instances, which is achieved by simply removing the summation with respect to $x \in T$:

$$\mathbf{w} := \mathbf{w} + \beta(f(x) - h(x))\nabla_{\mathbf{w}} h(x) \tag{8.15}$$

This final update rule shows how to modify the parameters of a parametric regression model based on a single training instance. It is referred to as the *incremental delta rule*, whereas the previous rule given by Equation 8.14, aggregating the contributions of all training instances, will be called the *batch delta rule*. They can be instantiated for particular representation functions by calculating $\nabla_{\mathbf{w}} h(x)$. Whenever the distinction between the batch and incremental formulation is immaterial, we will simply speak of the delta rule.

The $\nabla_{\mathbf{w}} h(x)$ gradient, which is the vector of per-parameter partial derivatives $\frac{\partial h(x)}{\partial w_i}$, is more than straightforward to calculate for the linear case. Clearly we have

$$\frac{\partial h(x)}{\partial w_i} = a_i(x) \tag{8.16}$$

and

$$\nabla_{\mathbf{w}} h(x) = \mathbf{a}(x) \tag{8.17}$$

accordingly. This makes it possible to write down a specialized linear version of the delta rule (assuming the incremental version) as follows:

$$\mathbf{w} := \mathbf{w} + \beta(f(x) - h(x))\mathbf{a}(x) \tag{8.18}$$

The linear delta rule is also referred to as the LMS rule or the Widrow–Hoff rule. Similarly to its general counterpart, it can be applied in either an incremental or batch mode. The former performs the updates based on single training instances immediately after they are calculated, whereas the latter accumulates the updates resulting from all training instances and applies them after the complete training set has been processed. In any case, multiple iterations are needed with an appropriately selected step-size parameter β to approach a minimum of the mean square error. Such iterative incremental or batch updates are performed by the gradient descent algorithm.

Example 8.3.1 Determining the gradient of the linear representation function is implemented and demonstrated by the R code presented below. The `w` argument is not actually used, since the linear representation gradient does not depend on model parameters. It appears on the argument list only to make the call interface of the gradient function ready for nonlinear representations as well.

```
## gradient of the linear representation function
grad.linear <- function(data, w) { cbind(data, 1) }

  # gradient for the first 10 instances
grad.linear(lrdat.train[1:10,1:4], rep(0, 5))
```

Example 8.3.2 The following R code implements the delta rule for mean square error minimization according to Equation 8.18. It takes vectors of true and predicted target function values, the gradient, and the step size value as arguments, and returns the resulting model parameter update vector. Its application to the `f1` target function of the example dataset is demonstrated, using the linear representation gradient implemented in the previous example. The first demonstration call uses the target value predictions produced by the perfect model from Example 8.2.2 and the corresponding parameter vector, which yields parameter updates of 0, as expected. In the second call, the true target function values are modified to force the delta rule to determine nonzero updates.

```
## calculate parameter update based on given true and predicted values,
## gradient, and step-size using the delta rule for MSE minimization
delta.mse <- function(true.y, pred.y, gr, beta)
{ colSums(beta*rmm(gr, (true.y-pred.y))) }

  # parameter updates for the perfect model for f1
delta.mse(lrdat.train$f1, predict(mod.perf1, lrdat.train[,1:4]),
          grad.linear(lrdat.train[,1:4], w.perf1), 0.1)
  # parameter updates for the perfect model for f1
  # with modified target function values
delta.mse(lrdat.train$f1+0.1, predict(mod.perf1, lrdat.train[,1:4]),
          grad.linear(lrdat.train[,1:4], w.perf1), 0.1)
```

8.3.3 Gradient descent

The gradient descent algorithm for training a parametric regression model performs a number of delta rule iterations to reach a parameter vector that yields a sufficiently small training set error. Similarly as for the delta rule, one can consider a batch or incremental version of the algorithm.

The batch gradient descent algorithm can be formulated as presented below.

1: **repeat**
2: **for** $i = 1, 2, ..., N$ **do**
3: $\Delta w_i := 0$;
4: **end for**
5: **for all** training instances $x \in T$ **do**
6: **for** $i = 1, 2, ..., N$ **do**
7: $\Delta w_i := \Delta w_i + \beta(f(x) - h(x))\frac{\partial h(x)}{\partial w_i}$;
8: **end for**
9: **end for**
10: **for** $i = 1, 2, ..., N$ **do**
11: $w_i := w_i + \Delta w_i$;
12: **end for**
13: **until** stop criteria are satisfied;

In each iteration of the algorithm the entire training set is used to calculate the updates for each parameter, which is then applied. The incremental version does not have to accumulate the updates resulting from each instance before applying them to actually modify parameters, so it is even simpler. The incremental gradient descent algorithm is also referred to as *online* or *stochastic gradient descent*.

1: **repeat**
2: select an instance $x \in T$;
3: **for** $i = 1, 2, ..., N$ **do**
4: $w_i := w_i + \beta(f(x) - h(x))\frac{\partial h(x)}{\partial w_i}$;
5: **end for**
6: **until** stop criteria are satisfied;

Both versions of the algorithm are capable of reaching a (local) minimum of the mean square error if using a sufficiently small (or appropriately adapted) step-size value. The batch

version, processing multiple instances at a time, may be more efficient if implemented using optimized vector and matrix operations. The incremental version may often be more convenient, however, particularly if the training set is not fixed and completely available, but consists of continuously arriving instances, which requires continuous parameter updates. It may also perform better computationally for very large data. It usually behaves similarly well or better than the batch version even when working with fixed and completely available training sets, as long as all training instances keep being selected with equal average frequency, but in varying order. An easy way to achieve this when using a fixed training set is to perform an internal iteration in which all training instances are processed in a randomized order, as in the following alternative formulation of the algorithm.

1: **repeat**
2: **for all** training instances $x \in T$ in randomized order **do**
3: **for** $i = 1, 2, ..., N$ **do**
4: $w_i := w_i + \beta(f(x) - h(x))\frac{\partial h(x)}{\partial w_i}$;
5: **end for**
6: **end for**
7: **until** stop criteria are satisfied;

Unless some background knowledge is available that could recommend a good initial guess of model parameters, the safest approach is to initialize them to small random numbers. The more "obvious" initialization to 0 may be inappropriate for some nonlinear representation functions which may then have a 0 gradient, which would clearly prevent the algorithm from making any parameter update, but is perfectly fine for linear models.

Typical stop criteria used to terminate the gradient descent process include:

Error. Reaching a sufficiently low error level.

Duration. Completing a specified number of iterations.

No improvement. No error improvement observed during a specified number of iterations.

Example 8.3.3 The following R code contains a simple implementation of the gradient descent algorithm for parametric regression models. The `gradient.descent` function receives a formula specifying the attributes and the target function, the training dataset, the initial parameter vector to start with, as well as the representation function and its gradient. Both of these take an attribute value vector or a dataset and a parameter vector on input. One can also specify the step-size value, the training mode, and the stop criterion (via an acceptable mean square error level or a maximum number of iterations). The training mode defaults to incremental (`batch=FALSE`), which processes training instances in randomized order. The latter, almost always desirable behavior, can be turned off by specifying the `randomize=FALSE` argument. The `delta` and `perf` arguments, defaulting to `delta.mse` and `mse`, respectively, specify the functions for model parameter update and performance measure calculation. Two auxiliary functions are used to extract attribute and target function names from the input formula, `x.vars` and `y.var`. The `as.num0` function is used to convert the target attribute values to a numerical representation, which makes it possible to apply the `gradient.descent` function to datasets with discrete target attributes as

`dmr.util`

long as the conversion is successful. The gradient descent algorithm is applied to estimating the parameters of a linear model for the `f1` target function of the example artificial dataset, as well as for the real *Boston Housing* data. The implementations of the linear representation function and its gradient from Examples 8.2.2 and 8.3.1, respectively, are used for this purpose.

```
## perform gradient descent iterative parameter estimation
## for parametric regression models
gradient.descent <- function(formula, data, w, repf, grad, delta=delta.mse, perf=mse,
                             beta=0.001, batch=FALSE, randomize=!batch,
                             eps=0.001, niter=1000)
{
  f <- y.var(formula)
  attributes <- x.vars(formula, data)
  aind <- names(data) %in% attributes
  true.y <- as.num0(data[[f]])
  model <- `class<-`(list(repf=repf, w=w), "par")
  iter <- 0

  repeat
  {
    if (batch)
    {
      pred.y <- predict.par(model, data[,aind])
      model$w <- model$w + delta(true.y, pred.y, grad(data[,aind], model$w), beta)
    }
    else
    {
      pred.y <- numeric(nrow(data))
      xind <- if (randomize) sample.int(nrow(data)) else 1:nrow(data)
      for (i in 1:length(xind))
      {
        av <- data[xind[i], aind]
        pred.y[xind[i]] <- predict.par(model, av)
        model$w <- model$w +
                   delta(true.y[xind[i]], pred.y[xind[i]], grad(av, model$w), beta)
      }
    }
    iter <- iter+1

    cat("iteration ", iter, ":\t", p <- perf(pred.y, true.y), "\n")
    if (p < eps || iter >= niter)
      return(list(model=model, perf=p))
  }
}

  # linear model for f1
gd1 <- gradient.descent(f1~a1+a2+a3+a4, lrdat.train, w=rep(0, 5),
                        repf=repf.linear, grad=grad.linear, beta=0.01, eps=0.0001)

  # linear model for the Boston Housing data
bh.gd <- gradient.descent(medv~., bh.train, w=rep(0, ncol(bh.train)),
                          repf=repf.linear, grad=grad.linear, beta=1e-6, eps=25,
                          niter=5000)

  # test set error
mse(predict(gd1$model, lrdat.test[,1:4]), lrdat.test$f1)
mse(predict(bh.gd$model, bh.test[,-13]), bh.test$medv)
```

When applied to the small artificial data, the algorithm converges quite fast to a low error level. In particular, the 0.0001 mean square error boundary is crossed within less than 50 iterations in most runs. The test set error is marginally above the training error, which is to be expected, and the estimated parameters can be seen to differ from the true ones (used for data generation) only slightly. Even better results would probably be possible with a longer training process or more carefully fine-tuned step-size value. Setting it too large, by the way, results in numerical explosion with model parameters falling out of bounds permitted by the arithmetic precision. The real *Boston Housing* data requires a very small step-size value, which results in slow convergence, but a reasonably good mean square error level is ultimately reached.

8.3.4 Least squares

An alternative approach to training linear regression models is also possible, using the well-known linear algebra *least-squares* method. It treats the parameter estimation task as a linear system solving task rather than an optimization task. Indeed, consider the following equation:

$$a_1(x)w_1 + a_2(x)w_2 + \cdots + a_n(x)w_n + a_{n+1}(x)w_{n+1} = f(x) \tag{8.19}$$

which simply demands that the model's prediction agrees with the true target function value for instance x. By writing such equations for all instances from the training set we receive a linear system, with model parameters playing the role of unknown variables. If $n + 1 < |T|$, which is natural to assume (it is in fact common to request $n \ll |T|$), the system is overdetermined and it may not have an exact solution, but a least-squares solution (i.e., such that it minimizes the mean square error of the predictions made using the obtained parameter vector with respect to the true target function values) can be found using a simple algebraic procedure.

Switching to a matrix notation for compactness and convenience, the linear system under consideration can be presented as

$$\mathbf{a}(T)\mathbf{w} = \mathbf{f}(T) \tag{8.20}$$

where $\mathbf{a}(T)$ is the $|T| \times (n + 1)$ attribute value matrix for the training set (with one row per instance and one column per attribute, including the artificial a_{n+1} attribute) and $\mathbf{f}(T)$ is the target function value vector for the training set, with one value per instance. As before, $\mathbf{w}$ is the model parameter vector and all vectors are treated as single-column matrices. The above matrix equation cannot be directly solved for $\mathbf{w}$ because $\mathbf{a}(T)$ is (usually) not square. It can be made square, however, by multiplying both sides of the equation by its transpose $\mathbf{a}^T(T)$:

$$\mathbf{a}^T(T)\mathbf{a}(T)\mathbf{w} = \mathbf{a}^T(T)\mathbf{f}(T) \tag{8.21}$$

Now, unless the $\mathbf{a}^T(T)\mathbf{a}(T)$ matrix turns out to be singular, it can be inverted to achieve the desired solution:

$$\mathbf{w} = \left(\mathbf{a}^T(T)\mathbf{a}(T)\right)^{-1}\mathbf{a}^T(T)\mathbf{f}(T) \tag{8.22}$$

Otherwise a pseudo-inversion operation can be applied to achieve an approximate solution. Thus, the "multiply-by-transpose" trick turns an undetermined linear system into an ordinary linear system with the same number of unknowns and equations which can be solved in the usual way.

This is rather a conceptual description of the least-squares method than a directly applicable algorithm, since it relies on the matrix inverse operation, which is not necessarily trivial and involves some advanced numerical techniques to be performed accurately and efficiently. Discussing these is beyond the scope of this chapter, though. Assuming a numerically correct and efficient implementation, this approach would be usually preferred to the gradient descent algorithm as more efficient (at least as long as the number of attributes, which determines the dimensions of the matrix that has to be inverted, is not exceedingly large), more reliable (since the optimal parameter vector is obtained directly rather than gradually approached), and easier to use (since there is no need to adjust the step-size parameter). This is actually the most basic form of the least-squares method, often referred to as *ordinary least squares* (OLS), with other versions having been developed for some extensions of the basic linear regression model representation.

Where the gradient descent algorithm wins is the incremental learning capability, which may be important for some applications, where there is a stream of training instances arriving one at a time (or a portion at a time) and the model needs to be continuously updated to incorporate new data.

Example 8.3.4 The R code presented below implements the ordinary least-squares method for linear model parameter estimation. It uses the `x.vars` and `y.var` functions to extract the attribute and target function names from the supplied formula. Then it uses the built-in `solve` function to solve the linear system obtained after applying the "multiply-by-transpose" trick, which takes care of the required matrix inversion. The least squares method is subsequently used to estimate linear model parameters for the `f1` target function.

`dmr.util`

```
## estimate linear model parameters using the OLS method
ols <- function(formula, data)
{
  f <- y.var(formula)
  attributes <- x.vars(formula, data)
  aind <- names(data) %in% attributes

  amat <- cbind(as.matrix(data[,aind]), intercept=rep(1, nrow(data)))
  fvec <- data[[f]]
  'class<-'(list(repf=repf.linear, w=solve(t(amat)%*%amat, t(amat)%*%fvec)), "par")
}

  # linear model for f1
ols1 <- ols(f1~a1+a2+a3+a4, lrdat.train)
  # linear model for the Boston Housing data
bh.ols <- ols(medv~., bh.train)

  # test set error
mse(predict(ols1, lrdat.test[,1:4]), lrdat.test$f1)
mse(predict(bh.ols, bh.test[,-13]), bh.test$medv)
```

The least-squares method is lightning fast compared to the gradient descent algorithm demonstrated in the previous example. This is, in part, due to implementational reasons (the `gradient.descent` function was entirely implemented in R, whereas the most computationally expensive matrix inverse operation of the OLS method is actually performed by an

efficiently implemented R's built-in function), but the latter is inherently more efficient anyway as long as the dimension of the matrix to inverse – i.e., the number of attributes – remains within reasonable limits. For the artificial data the estimated parameters match the true ones exactly, which makes the test set mean square error practically 0. For the real *Boston Housing* data the mean square error level is somewhat less than the one previously achieved with gradient descent.

8.4 Discrete attributes

Unlike for the classification task and many classification algorithms, where it is often convenient to assume that attributes are mostly discrete and it is nearly always necessary to assume that some attributes may be discrete, it is not untypical for regression algorithms to implicitly assume that attributes are continuous. In particular, parametric regression is usually presented with this assumption. Whereas the representation function mapping attribute value and parameter vectors into model predictions can, in general, handle arbitrary attributes, the linear representation function can be directly applied to continuous attributes only. This is also the case for nearly all nonlinear parametric representations that are in common use, which employ representation functions defined via arithmetic operations on attribute values.

To make the linear representation function and other arithmetic representation functions applicable to discrete attributes, the latter can be transformed by the simple binary encoding transformation described in Section 17.3.5. For the linear representation, after a k-valued discrete attribute $a_i : X \to \{v_{i,1}, v_{i,2}, \ldots, v_{i,k}\}$ has been replaced by $k-1$ binary attributes, its contribution to the representation function is

$$a_{i,1}(x)w_{i,1} + a_{i,2}(x)w_{i,2} + \cdots + a_{i,k-1}(x)w_{i,k-1} = \sum_{j=1}^{k-1} a_{i,j}(x)w_{i,j} \tag{8.23}$$

where

$$a_{i,j}(x) = \begin{cases} 1 & \text{if } a_i(x) = v_{i,j} \\ 0 & \text{otherwise} \end{cases} \tag{8.24}$$

for $j = 1, 2, \ldots, k-1$. In practice, any set of two substantially different values can be used for this encoding instead of $\{0, 1\}$, with $\{-1, 1\}$ being the second typical choice.

The gradient descent algorithm can be directly used with discrete attributes as long as this binary encoding is applied when calculating the representation function and its gradient. The least-squares method requires that the encoding be applied when creating the attribute value matrix for the training set. Other than that, both the algorithms can operate without modifications.

Example 8.4.1 The R code presented below defines a wrapper that takes a representation function on input and returns its version equipped with the discrete attribute handling capability. This is achieved by applying the `discode` encoding function that leaves continuous attribute values unchanged, but replaces discrete attribute values with appropriate binary sequences, as discussed above. An analogous wrapper is also defined for the representation function gradient. This makes it possible to directly apply

Ex. 17.3.5
`dmr.trans`

the gradient descent algorithm to estimate linear model parameters for data with discrete attributes. Discrete attribute-enabled implementation of the least-squares regression algorithm is also presented that apply the same encoding to the training set when creating the attribute value matrix. Then the two algorithms are applied to estimate linear model parameters for the *weatherr* data. For the gradient descent algorithm, the last (intercept) parameter is initialized to 1 rather than to 0, which appears to be reasonable given the fact that all target function values are positive numbers below 1. Note that there are six linear model parameters: two for the originally continuous attributes `temperature` and `humidity`, two for the transformed `outlook` attribute (which has originally three discrete values), and one for the transformed `windy` attribute (which has originally two discrete values).

```
## representation function wrapper to handle discrete attributes
repf.disc <- function(repf)
{ function(data, w) { repf(discode(~., data, b=c(-1,1)), w) } }

## representation function gradient wrapper to handle discrete attributes
grad.disc <- function(grad)
{ function(data, w) { grad(discode(~., data, b=c(-1,1)), w) } }

## estimate linear model parameters using the OLS method
## with discrete attributes
ols.disc <- function(formula, data)
{
  f <- y.var(formula)
  attributes <- x.vars(formula, data)
  aind <- names(data) %in% attributes

  amat <- cbind(as.matrix(discode(~., data[,aind], b=c(-1,1))),
                intercept=rep(1, nrow(data)))
  fvec <- data[[f]]
  `class<-`(list(repf=repf.disc(repf.linear),
                 w=solve(t(amat)%*%amat, t(amat)%*%fvec)), "par")
}

  # gradient descent for the weatherr data
w.gdl <- gradient.descent(playability~., weatherr, w=c(rep(0, 5), 1),
                          repf=repf.disc(repf.linear), grad=grad.disc(grad.linear),
                          beta=0.0001, eps=0.005)
mse(weatherr$playability, predict(w.gdl$model, weatherr[,1:4]))

  # OLS for the weatherr data
w.ols <- ols.disc(playability~., weatherr)
mse(predict(w.ols, weatherr[,1:4]), weatherr$playability)
```

Both the algorithms work with a discrete attribute as expected. Not surprisingly, the least-squares algorithm beats gradient descent both with respect to speed and model accuracy.

8.5 Advantages of linear models

There is only one disadvantage of linear models, which is both very obvious and very severe: they cannot directly represent nonlinear relationships and hence approximate nonlinear target functions. Before discussing whether and how this major limitation could be overcome, it is worthwhile to consider the advantages of linear models that justify that effort.

One advantage that may sometimes be important is the computational efficiency of creating and using linear models. The linear representation function is inexpensive to calculate, and even more so is its derivative, which makes the cost of both prediction and gradient descent training relatively small. Moreover, the least-squares method makes it possible to calculate the optimal parameter vector directly, which may often take less time than required by the gradient descent algorithm to converge. However, with the increasing computational power available, this advantage tends to become less important.

What remains more unquestionable is the benefits resulting from the shape of the mean square error function, which – under a linear representation – is quadratic with respect to model parameters. This makes the optimization process easy and free of misleading local optima at which the gradient descent algorithm could get stuck with a nonlinear representation. Linear models are therefore not only more efficient to create and use, but also much more easy to fit accurately to the data, as long as the relationship between the target function and attributes is indeed linear.

Last but not least, the representation simplicity makes linear models much easier to understand, explain, and verify than most nonlinear parametric models. The parameter corresponding to each attribute directly reflects its contribution to the model's predictions.

All this justifies the interest that linear regression receives and the efforts made toward enhancing its capabilities toward modeling nonlinear relationships.

8.6 Beyond linearity

To retain the advantages of linear models while overcoming the linearity limitation one should seek for regression models that use the same representation and parameter estimation algorithms as linear models internally, but are wrapped with some add-on techniques that make it possible to represent nonlinear relationships. One natural way to achieve this is to transform the output of a plain linear model nonlinearly, i.e., adopting a generalized linear representation. This form of introducing nonlinearity may often be insufficient, though. In such cases two other strategies can be employed, enhanced representation and piecewise-linear regression.

8.6.1 Generalized linear representation

A *generalized linear representation* assumes that there is a nonlinear *link function* that transforms the linear combination of attribute values and model parameters into the final model prediction. This is written as

$$h(x) = L^{-1}\left(\sum_{i=1}^{n} w_i a_i(x) + w_{n+1}\right) = L^{-1}(g(x)) \tag{8.25}$$

The resulting representation function is a composite of the linear representation function (as the inner function):

$$g(x) = \sum_{i=1}^{n} w_i a_i(x) + w_{n+1} = \mathbf{w} \circ \mathbf{a}(x) \tag{8.26}$$

and the nonlinear inverse link function L^{-1} (as the outer function). The link function applied to model predictions makes them linear:

$$L(h(x)) = \mathbf{w} \circ \mathbf{a}(x) \tag{8.27}$$

It is noteworthy that the linear threshold and logit representations for linear classification presented in Sections 5.2.3 and 5.2.4 are instantiations of the generalized linear representation.

Using a generalized linear representation is one of the features of *generalized linear models*, which also include other important enhancements over plain linear models. In particular, they can incorporate a specified target function probability distribution and a performance measure to be optimized. For some specific instantiations, dedicated parameter estimation algorithms have been developed. All these issues are beyond the scope of this chapter.

8.6.1.1 Delta rule for generalized linear representation

The delta rule for the generalized linear representation with a differentiable link function can be directly derived from Equation 8.15 by calculating the gradient

$$\nabla_{\mathbf{w}} h(x) = (L^{-1})'(\mathbf{w} \circ \mathbf{a}(x))\mathbf{a}(x) \tag{8.28}$$

where $(L^{-1})'$ is the derivative of the inverse link function. This yields the following model parameter update:

$$\mathbf{w} := \mathbf{w} + \beta(f(x) - h(x))(L^{-1})'(\mathbf{w} \circ \mathbf{a}(x))\mathbf{a}(x) \tag{8.29}$$

which makes it possible to apply the gradient descent descent algorithm.

Example 8.6.1 The following R code defines the `repf.gen` and `grad.gen` functions that create the generalized linear representation function and its gradient for a given link function. These are wrappers around `repf.linear` and `grad.linear` (although another inner representation function and gradient can be specified). The inverse link function has to be supplied as an argument to `repf.gen` and its derivative as an argument to `grad.gen`. Such a pair of functions are then defined to exactly match the `f2` target function, which cannot be expected in practice for unknown target functions, but serves well the illustration purpose. This makes it possible to estimate generalized linear model parameters for `f2` using the gradient descent algorithm.

```
## generalized representation function
repf.gen <- function(link.inv, repf=repf.linear)
{ function(data, w) { link.inv(repf(data, w)) } }

## generalized representation function gradient
grad.gen <- function(link.inv.deriv, repf=repf.linear, grad=grad.linear)
{ function(data, w) { rmm(grad(data, w), link.inv.deriv(repf(data, w))) } }

  # perfect inverse link function for f2
link2.inv <- function(v) { tanh(v/10) }
  # and its derivative
link2.inv.deriv <- function(v) { (1-tanh(v/10)^2)/10 }

  # perfect generalized linear representation function for f2
repf.gen2 <- repf.gen(link2.inv)
  # and its gradient
grad.gen2 <- grad.gen(link2.inv.deriv)

  # gradient descent estimation of generalized linear model parameters for f2
gd2g <- gradient.descent(f2~a1+a2+a3+a4, lrdat.train, w=rep(0, 5),
                         repf=repf.gen2, grad=grad.gen2,
                         beta=0.5, eps=0.0001)
  # test set error
mse(predict(gd2g$model, lrdat.test[,1:4]), lrdat.test$f2)
```

The gradient descent algorithms works as expected, arriving at a low error. Interestingly, unlike in the plain linear case, a large step-size value can be used without running at numerical explosion problems. This is a consequence of the particular link function, with values bound to the $(-1, 1)$ interval.

8.6.1.2 Least squares for generalized linear representation

Unfortunately there is no simple modification of the OLS algorithm that would be it applicable to estimating generalized linear representation parameters, because there is no closed-form solution of the mean square error minimization problem for arbitrary nonlinear link functions. A naïve approach that may give reasonable results could be applying the link function to the target values in the dataset and minimizing the mean square error of the linear inner representation function with respect to such transformed target values:

$$\frac{1}{|T|}\sum_{x\in T}(L(f(x)) - g(x))^2 \tag{8.30}$$

Clearly the resulting parameter vector is not guaranteed to minimize the mean square error of h with respect to f in general. It may still be quite good, however, particularly if the link function is monotonic.

Example 8.6.2 The naïve application of the ordinary least-squares method to parameter estimation for a generalized linear representation is implemented and demonstrated by the following R code. The specified link function – matching the `f2` target function – is internally used to transform true target function values. The corresponding inverse link function from the previous example is used for prediction.

```
## a naive application of OLS to a generalized linear representation
ols.gen <- function(formula, data, link=function(v) v, link.inv=function(v) v)
{
  f <- y.var(formula)
  attributes <- x.vars(formula, data)
  aind <- names(data) %in% attributes

  amat <- cbind(as.matrix(data[,aind]), intercept=rep(1, nrow(data)))
  fvec <- link(data[[f]])
  `class<-`(list(repf=repf.gen(link.inv), w=solve(t(amat)%*%amat, t(amat)%*%fvec)),
            "par")
}

  # perfect link function for f2
link2 <- function(v) { 10*atanh(v) }

  # estimate of generalized linear model parameters for f2
ols2g <- ols.gen(f2~a1+a2+a3+a4, lrdat.train, link=link2, link.inv=link2.inv)
  # test set error
mse(predict(ols2g, lrdat.test[,1:4]), lrdat.test$f2)
```

Since the link function applied in this illustration matches the target function perfectly (which is completely unrealistic), a parameter vector is found that achieves a near-zero mean square error of the inner linear representation function with respect to the transformed target function. It should therefore be not surprising that the mean square error of the inversely

transformed model predictions with respect to the target function is near-zero as well. Unfortunately such successful performance cannot be expected in practice.

8.6.2 Enhanced representation

The idea of enhanced representation is to replace the original attributes $a_1, a_2, \ldots, a_n$ defined on the domain by new attributes $a'_1, a'_2, \ldots, a'_N$ related to them nonlinearly. More exactly, these new enhanced attributes are defined by nonlinear functions of single or multiple original attributes, and typically (but not necessarily) $N \gg n$. The nonlinear relationship between the target function and the original attributes is expected to become linear in the enhanced representation.

Once the transformation that generates new attribute values has been determined, no changes to parameter estimation algorithms are required. They can operate in the usual way, just using the enhanced set of attributes. The definitions of enhanced attributes have to be retained with the model, since new data has to be transformed to the same representation before the model is applied for prediction.

There are many possible approaches to defining an enhanced representation, some of which yield sophisticated and powerful regression algorithms. Sometimes sufficient background knowledge may be available about the domain and the target function to directly suggest appropriate enhanced attribute definitions. Otherwise one of the general-purpose enhanced representations can be employed, such as

- randomized representation, in which enhanced attributes are defined using nonlinear random transformations,
- tile coding, which is based on a series of grids partitioning the domain into overlapping boxes,
- kernel methods, using an implicit nonlinear transformation based on attribute vector dot products.

Of those, the last approach is the most interesting and widely used, and will be presented in Chapter 16.

Example 8.6.3 To illustrate the basic idea of enhanced representation, the following R code defines a general enhanced representation function and its gradient, which can be used to apply an enhancement transformation performed by a function specified as the `enhance` argument to an arbitrary base representation function specified by the `repf` argument and its gradient specified by the `grad` argument. An enhanced representation function and its gradient are sufficient to use the gradient descent algorithm, which requires no changes by itself. The least squares method needs a slightly modified implementation, provided by the `ols.enh` function, that applies the enhancement internally when creating the attribute value matrix. Then a representation enhancement function is defined that implicitly creates a new set of attributes, consisting of both the original four attributes and their squares. This quite trivial enhanced representation can be immediately seen to match the `f3` target function and can therefore be considered the perfect enhanced representation for this target function. This is what could be hardly possible in practice with an unknown target function. Linear model parameters for the `f3` target function with this enhanced representation are then estimated using both the gradient descent algorithm and the least-squares method.

```
## enhanced representation function
repf.enh <- function(enhance, repf=repf.linear)
{ function(data, w) { repf(enhance(data), w) } }

## enhanced representation function gradient
grad.enh <- function(enhance, grad=grad.linear)
{ function(data, w) { grad(enhance(data), w) } }

## estimate linear model parameters using the OLS method
## with enhanced representation
ols.enh <- function(formula, data, enhance=function(data) data)
{
  f <- y.var(formula)
  attributes <- x.vars(formula, data)
  aind <- names(data) %in% attributes

  amat <- cbind(as.matrix(enhance(data[,aind])), intercept=rep(1, nrow(data)))
  fvec <- data[[f]]
  `class<-`(list(repf=repf.enh(enhance), w=solve(t(amat)%*%amat, t(amat)%*%fvec)),
            "par")
}

  # perfect representation enhancement for f3
enhance3 <- function(data) { cbind(data, sq=data^2) }

  # gradient descent estimation for f3
gd3e <- gradient.descent(f3~a1+a2+a3+a4, lrdat.train, w=rep(0, 9),
                         repf=repf.enh(enhance3), grad=grad.enh(enhance3),
                         beta=0.001, eps=0.005)
  # test set error
mse(predict(gd3e$model, lrdat.test[,1:4]), lrdat.test$f3)

  # ols estimation for f3
ols3e <- ols.enh(f3~a1+a2+a3+a4, lrdat.train, enhance3)
  # test set error
mse(predict(ols3e, lrdat.test[,1:4]), lrdat.test$f3)
```

The gradient descent algorithm turns out to perform noticeably worse in the enhanced representation than observed before in the original four-attribute representation for the `f1` target function, which was indeed linear. Additional attributes not only increase the computational time, but also appear to make the optimization task more complex, a smaller step-size value has to be used, and several hundred iterations are required to reach a rather unimpressive error level of 0.01. Still, the algorithm does work and a longer training process would likely lead to a better estimated parameter vector. The OLS method remains lightning fast and accurate.

8.6.3 Polynomial regression

A popular simple special case of enhanced representation is obtained if new attributes are polynomial functions of the original attributes. The resulting representation function can be presented as

$$h(x) = F(\mathbf{a}(x), w) = \sum_{j=1}^{p} \sum_{i=1}^{n} w_{i+(j-1)n} a_i^j(x) + w_{pn+1} \tag{8.31}$$

where p is the maximum degree of polynomials used. Models using such a representation are called *polynomial regression* models.

Example 8.6.4 The `repf.enh` and `grad.enh` functions from the previous example can be used to generate polynomial regression representation functions and gradients. This is demonstrated by the following R code, which defines the `enhance.poly` function for polynomial representation enhancement, as well as the `repf.poly` and `grad.poly` functions calculating the polynomial representation function and its gradient. They are illustrated by reproducing the gradient descent and OLS parameter estimation process from the previous example.

```
## polynomial representation enhancement
enhance.poly <- function(data, p=2)
{ do.call(cbind, lapply(1:p, function(j) data^j)) }

## polynomial regression representation function
repf.poly <- function(p=2)
{ repf.enh(function(data) enhance.poly(data, p), repf.linear) }

## polynomial regression representation function gradient
grad.poly <- function(p=2)
{ grad.enh(function(data) enhance.poly(data, p), grad.linear) }

  # gradient descent polynomial regression estimation for f3
gd3p <- gradient.descent(f3~a1+a2+a3+a4, lrdat.train, w=rep(0, 9),
                         repf=repf.poly(p=2), grad=grad.poly(p=2),
                         beta=0.001, eps=0.005)
  # test set error
mse(predict(gd3p$model, lrdat.test[,1:4]), lrdat.test$f3)

  # OLS polynomial regression estimation for f3
ols3p <- ols.enh(f3~a1+a2+a3+a4, lrdat.train, enhance.poly)
  # test set error
mse(predict(ols3p, lrdat.test[,1:4]), lrdat.test$f3)
```

8.6.4 Piecewise-linear regression

Piecewise-linear regression is based on partitioning the domain into disjoint regions such that the target function can be sufficiently well approximated by a linear model in each of them. The overall regression model therefore consists of multiple linear models, each applicable in an appropriate region, and the description of the underlying partitioning that can be applied to appropriately select a linear model for any instance for which prediction would be requested. These individual linear models can be created in the usual way, using subsets of the training set consisting of instances from the corresponding regions.

There are several possible ways of decomposing the domain into regions and describing the obtained partitioning. Not surprisingly, some of them borrow their essential ideas from classification and clustering algorithms, which explicitly or implicitly partition the domain into regions corresponding to different classes or clusters. Arguably the simplest approach

is to use a clustering algorithm, such as k-means or another member of the k-centers family presented in Chapter 12, to create a clustering model and then create separate linear regression models for each cluster. This would be based on the hope that the relationship between the target function and attributes, nonlinear in general, would be linear within similarity-based clusters. More refined approaches include:

- model trees, which apply the hierarchical partitioning idea borrowed from decision trees to the regression task,
- local regression, which can be considered an extension of nearest-neighbor regression.

The former is described separately in Section 9.8 and the latter is beyond the scope of this book.

8.7 Conclusion

Parametric regression is the most commonly applied approach to the regression task and, at the same time, the most common instantiation of the parametric model representation. It is not surprising given the fact that regression models predict continuous values based on usually also continuous attributes. It is much more likely to encounter discrete attributes in classification tasks, and whenever all or most attributes are discrete, the parametric model representation becomes considerably less natural and convenient.

A variety of diverse regression algorithms belong to the parametric regression family. The purpose of this chapter was to provide a common background for them and to present more details on those that employ a linear model representation. This helps us to understand various practical parametric regression algorithms and makes it possible to train domain-specific models based on custom representation functions, suggested by the available background knowledge.

Linear regression, either in its plain form or equipped with enhancements that help it to overcome the linearity limitation (generalized linear representation, enhanced representation, piecewise-linear representation), is the most commonly used instantiation of parametric regression for data mining applications. Whereas nonlinear parametric models have also gained considerable popularity and proved successful whenever model accuracy is of extreme importance (with neural networks being the most typical example), linear models and their enhancements remain unbeatable whenever the computation time needed to create the model matters or large volumes of data have to be used. This is because of the efficiency of their parameter estimation algorithms and their robustness against local error minima. They are also relatively easy to interpret, which may be important in some application areas where regression models have to be compared to or combined with existing background expert knowledge.

8.8 Further readings

Similar to linear classification, linear regression has its roots in both machine learning and statistics, with the former traditionally being associated with the gradient descent approach to parameter estimation and the latter with least squares and other more advanced methods not covered in this chapter. This tradition distinction tends to disappear, by the way, with linear regression algorithms usually presented in a similar way by contemporary textbooks in both

these areas. From the large set of available machine learning and data mining books, Hand *et al*. (2001) and Bishop (2007) may be particularly worthwhile to consult for a much broader review of linear regression parameter estimation techniques and their theoretical foundations. For an even greater scope and depth, dedicated books on regression and statistical modeling can be referred to (e.g., Draper and Smith 1998; Freedman 2009; Glantz and Slinker 2000). Faraway (2004, 2005) complements an extensive discussion of linear regression by R language demonstrations.

The gradient-based delta or LMS parameter update rule for linear regression models was presented by Widrow and Hoff (1960). Related nonlinear regression algorithms studied in the field of neural networks (e.g., Hertz *et al*. 1991), such as error backpropagation for multilayer nonlinear perceptrons, are also covered by some data mining and machine learning books (Bishop 2007; Hand *et al*. 2001; Mitchell 1997; Tan *et al*. 2013). More refined optimization methods that can be applied to parameter estimation, particularly in the nonlinear case, include the Newton–Raphson, Gauss–Newton, and conjugate gradient algorithms (e.g., Björck 1996; Snyman 2005). Different variations of the online (stochastic) gradient descent algorithm can be seen as instantiations of stochastic approximation, initialized by Robbins and Monro (1951). Bottou (1998) discussed links between these related families of algorithms and showed how the theory of stochastic approximation can be used to prove the convergence of online gradient descent.

The ordinary least-squares method belongs to the oldest modeling algorithms still in widespread use, with its first description published nearly 200 years ago (Legendre 1805). There are of course several contemporary reviews, covering different variations of least-squares parameter estimation (e.g. Hansen *et al*. 2012; Lawson and Hanson 1987).

Generalized linear models (GLM) introduced by Nelder and Wedderburn (1972) are extensively described by McCullagh and Nelder (1989). This discussion covers, in particular, a general approach to parameter estimation and a variety of different link functions and target distributions that yield specific GLM instantiations. Two specific types of enhanced linear representations mentioned in this chapter, but not discussed in the book, are tile coding and random representation. The former was proposed by Albus (1975a,b) as the CMAC function approximator (*cerebellar model articulation controller*) and the latter by Sutton and Whitehead (1993), inspired by the earlier work of Kanerva (1988).

References

Albus JS 1975a Data storage in the cerebellar model articulation controller (CMAC). *Transactions of the ASME Journal of Dynamic Systems, Measurement, and Control* **97**, 228–233.

Albus JS 1975b New approach to manipulator control: The cerebellar model articulation controller (CMAC). *Transactions of the ASME Journal of Dynamic Systems, Measurement, and Control* **97**, 220–227.

Bishop CM 2007 *Pattern Recognition and Machine Learning*. Springer.

Björck A 1996 *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics.

Bottou L 1998 Online algorithms and stochastic approximations In *Online Learning and Neural Networks* (ed. Saad D) Cambridge University Press.

Draper NR and Smith H 1998 *Applied Regression Analysis* 3rd edn. Wiley.

Faraway JJ 2004 *Linear Models in R*. Chapman and Hall.

Faraway JJ 2005 *Extending the Linear Model with R: Generalized Linear, Mixed Effects, and Nonparametric Regression Models*. Chapman and Hall.

Freedman DA 2009 *Statistical Models: Theory and Practice*. Cambridge University Press.

Glantz SA and Slinker BS 2000 *Primer of Applied Regression and Analysis of Variance* 2nd edn. McGraw-Hill.

Hand DJ, Mannila H and Smyth P 2001 *Principles of Data Mining*. MIT Press.

Hansen PC, Pereyra V and G. S 2012 *Least Squares Data Fitting with Applications Hardcover*. John Hopkins University Press.

Hertz J, Krogh A and Palmer RG 1991 *Introduction to the Theory of Neural Computation*. Addison-Wesley.

Kanerva P 1988 *Sparse Distributed Memory*. MIT Press.

Lawson CL and Hanson RJ 1987 *Solving Least Squares Problems*. Society for Industrial and Applied Mathematics.

Legendre AM 1805 *Nouvelles méthodes pour la détermination des orbites des comètes*. Didot.

McCullagh P and Nelder JA 1989 *Generalized Linear Models* 2nd edn. Chapman and Hall.

Mitchell T 1997 *Machine Learning*. McGraw-Hill.

Nelder J and Wedderburn R 1972 Generalized linear models. *Journal of the Royal Statistical Society A* **135**, 370–384.

Robbins H and Monro S 1951 A stochastic approximation method. *The Annals of Mathematical Statistics* **22**, 400–407.

Snyman JA 2005 *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Springer.

Sutton RS and Whitehead SD 1993 Online learning with random representations *Proceedings of the Tenth International Conference on Machine Learning (ICML-93)*. Morgan Kaufmann.

Tan PN, Steinbach M and Kumar V 2013 *Introduction to Data Mining* 2nd edn. Addison-Wesley.

Widrow B and Hoff ME 1960 Adaptive switching circuits *Western Electronic Show and Convention (WesCon-60)*, vol. 4. Institute of Radio Engineers.

10

Regression model evaluation

10.1 Introduction

Just like for classification model evaluation addressed in Chapter 7, the evaluation of a regression model is intended to provide a reliable assessment of its *predictive performance*, i.e., the quality of the target function's approximation it represents. There are several regression performance measures calculated by comparing the model's predictions and true target function values on a particular dataset. These are not only the direct indicators of *dataset performance*, but – under some conditions – can also serve as estimators of *true performance*, i.e., their expected values on the whole domain.

10.1.1 Dataset performance

Dataset performance, obtained by calculating one or more selected performance measures on a particular dataset, represents the degree of match between model predictions and target function values on this dataset.

10.1.2 Training performance

Performance measures calculated for a model on the training set used to create the model represent its training performance. It may be useful for diagnostic purposes, but does not provide information on the actual predictive utility of the model.

10.1.3 True performance

The actual predictive power of a model is reflected by its expected performance (with respect to one or more selected performance measures) on the whole domain, which is the model's true performance. Since target function values are generally unavailable, true performance always remains unknown and has to be estimated by dataset performance. The challenge of reliably estimating the unknown values of the adopted performance measures on the whole domain

Data Mining Algorithms: Explained Using R, First Edition. Paweł Cichosz.

is addressed by the same *evaluation procedures* as presented in Section 7.3 for classification models and will therefore be discussed only very briefly. Regression model performance measures, however, have to take into account the specificity of numeric predictions and will be presented more extensively.

Example 10.1.1 The regression model performance measures and evaluation procedures presented in this chapter will be illustrated in R by applying them to the evaluation of a regression tree model created using the `rpart` package for the *Boston Housing* dataset, available in the `mlbench` package. The following R code prepares the demonstration by loading the packages and the dataset, splitting the dataset randomly into training and test subsets, and creating a model based on the training set. The random generator seed is explicitly initialized to make the results which are presented for some of the forthcoming examples easily reproducible.

```
library(dmr.claseval)
library(rpart)

data(BostonHousing, package="mlbench")

set.seed(12)
rbh <- runif(nrow(BostonHousing))
bh.train <- BostonHousing[rbh>=0.33,]
bh.test <- BostonHousing[rbh<0.33,]

bh.tree <- rpart(medv~., bh.train)
```

10.2 Performance measures

Regression performance measures share the same basic principle with classifcation performance measures: they compare the predictions generated by the model on a dataset S with the true target function values for the instances from this dataset. What has to be different, due to the specificity of the regression task, is the exact way of making the comparison.

Similarly as for classification, some of regression performance measures serve as implicit or explicit optimization criteria for regression algorithms and, in this role, they are sometimes referred to as *loss functions*.

10.2.1 Residuals

The most common regression performance measures are based on the differences between true and predicted function values. Such differences are called model *residuals*. For any instance x, the difference $f(x) - h(x)$ is the model's residual for instance x.

Apart from being used for calculating performance measures, model residuals are often analyzed statistically using distribution description and visualization techniques such as presented in Section 2.4. In particular, one simple and commonly used visual tool for residual analysis is a *residual plot*, i.e., a plot of model residuals vs. true target function values.

Example 10.2.1 The following R code produces the distribution summary of the test set residuals (as well as their absolute values) of the regression tree created for the *Boston Housing*

data, as well as produces a boxplot, a histogram, and a residual plot thereof, using a simple function that calculates model residuals by subtracting model predictions from the true target function values.

```
res <- function(pred.y, true.y) { true.y-pred.y }

bh.res <- res(predict(bh.tree, bh.test), bh.test$medv)
summary(bh.res)
summary(abs(bh.res))

boxplot(bh.res, main="Residual boxplot")
hist(bh.res, main="Residual histogram")
plot(bh.test$medv, bh.res, main="Residual plot")
```

The plots are presented in Figure 10.1. Although more than a half of residuals are in the $(-3, 3)$ interval, and the absolute values of more than 75% of them do not exceed 4.5, there are some much larger residuals as well. The piecewise-constant regression tree model representation makes the residual plot contain multiple linear segments.

10.2.2 Mean absolute error

Of several types of residual-based performance measures, the *mean absolute error* (MAE) is the most straightforward. It is calculated for model $h : X \to \mathcal{R}$ with respect to target function $f : X \to \mathcal{R}$ on dataset $S \subset X$ as the mean absolute residual for instances from S:

$$\text{mae}_{f,S}(h) = \frac{1}{|S|} \sum_{x \in S} |f(x) - h(x)| \tag{10.1}$$

This makes the contribution of each residual proportional to its absolute value.

The mean absolute error is also referred to as the *absolute loss*.

Example 10.2.2 The following R code defines a function for calculating the mean absolute error for given vectors of predicted and true target function values and demonstrates its application for the *Boston Housing* dataset.

```
mae <- function(pred.y, true.y) { mean(abs(true.y-pred.y)) }

mae(predict(bh.tree, bh.test), bh.test$medv)
```

10.2.3 Mean square error

The most widely employed performance measure for regression models is the *mean square error* (MSE). For model $h : X \to \mathcal{R}$ and target function $f : X \to \mathcal{R}$ it is calculated on dataset $S \subset X$ as the model's mean squared residual on this dataset:

$$\text{mse}_{f,S}(h) = \frac{1}{|S|} \sum_{x \in S} (f(x) - h(x))^2 \tag{10.2}$$

Compared to the mean absolute error, the mean square error more severely "punishes" large residuals. A model with a small number of large residuals and a large number of small residuals might appear good according to the mean absolute error, but poor according to the

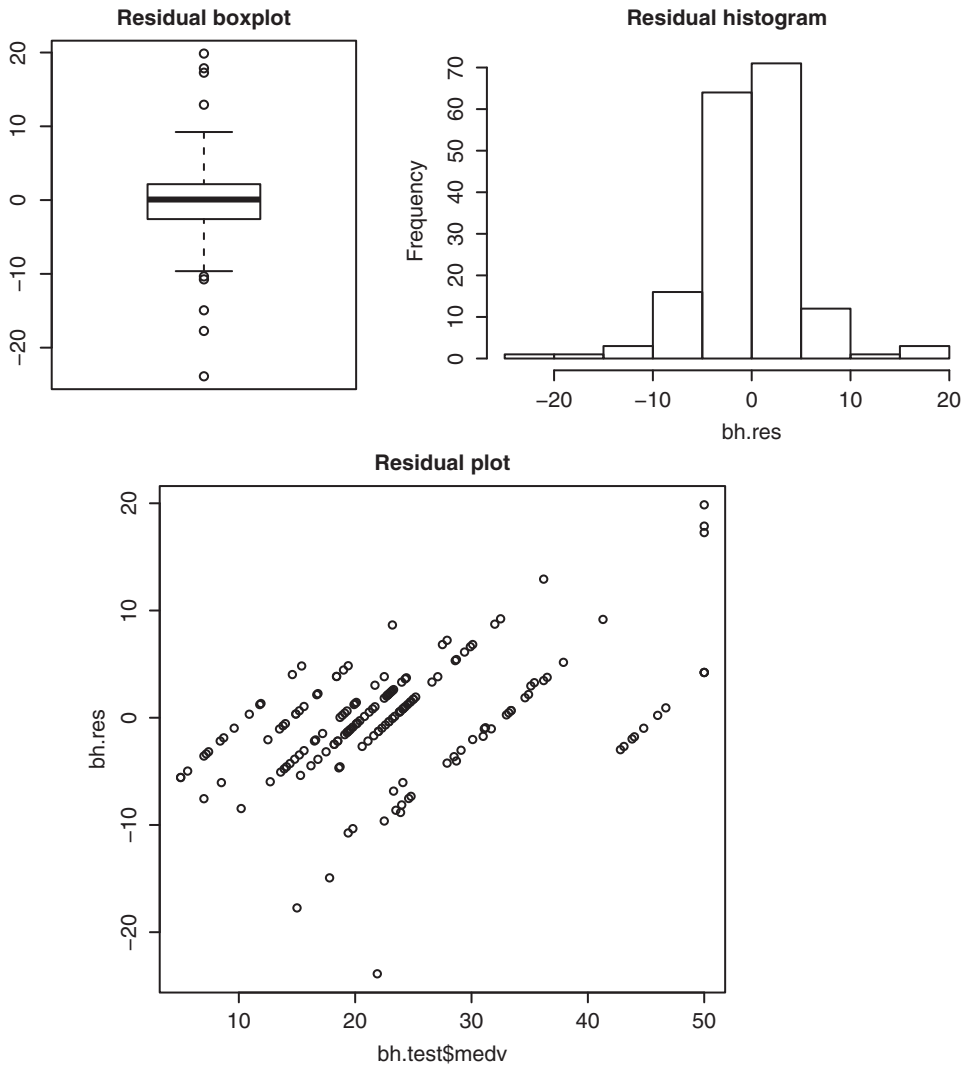

Figure 10.1 Visualization of model residuals for the *Boston Housing* data.

mean square error. Since large residuals are generally undesired, this can be considered one reason of the popularity of the mean square error. Another benefit is related to the analytical advantage of the square function over the absolute function, with the former being differentiable. This is important, in particular, for parametric regression, where gradient-based parameter estimation methods are available to minimize the mean square error on the training set, as discussed in Section 8.3.

The mean square error is the most commonly used type of loss function for regression, referred to as the *quadratic loss*.

Example 10.2.3 The following R code implements mean square error calculation and demonstrates it on the *Boston Housing* dataset.

```
mse <- function(pred.y, true.y) { mean((true.y-pred.y)^2) }

mse(predict(bh.tree, bh.test), bh.test$medv)
```

10.2.4 Root mean square error

A minor practical disadvantage of the mean square error is the effect of changed scale due to squaring, which makes the interpretation of error values harder, particularly if the target function represents quantities in some meaningful units of measurement (such as currency or physical units). This easily solved without loosing the benefits of this performance measure by applying the square root, which yields the *root mean square error* (RMSE):

$$\mathrm{rmse}_{f,S}(h) = \sqrt{\mathrm{mse}_{f,S}(h)} \tag{10.3}$$

Due to the strict monotonicity of the root square function the preference for models implied by the root mean square error is exactly the same as that of the mean square error (i.e., any model minimizing one of these measures also minimizes the other). They only differ in interpretation convenience, which makes the root mean square error popular for presenting the quality of obtained regression models in reports, particularly addressed to business-oriented recipients.

Example 10.2.4 The function defined by the following R code calculates the root mean square error. Like in the previous examples, it is demonstrated in application to the *Boston Housing* dataset.

```
rmse <- function(pred.y, true.y) { sqrt(mse(pred.y, true.y)) }

rmse(predict(bh.tree, bh.test), bh.test$medv)
```

Notice that the root mean square error, although expressed in the same scale and units as the mean absolute error calculated in Example 10.2.2, is considerably larger than the latter, which results from a small number of large residuals being more severely punished.

10.2.5 Relative absolute error

The performance measures presented above, based on absolute or squared residuals, can be directly used for comparing different models, but in order to assess a model's practical utility for a given application they have to be accompanied by some description of the target function distribution (observed on the dataset used for the evaluation), making it possible to judge whether model residuals are sufficiently small. On some occasions it may be interesting and useful to relate model residuals to the values or variability of the target function itself directly within a performance measure. This is accomplished, in particular, by the *relative absolute error* (RAE), defined as follows:

$$\mathrm{rae}_{f,S}(h) = \frac{\sum_{x\in S} |f(x) - h(x)|}{\sum_{x\in S} |f(x) - m_S(f)|} = \frac{\mathrm{mae}_{f,S}(h)}{\frac{1}{|S|}\sum_{x\in S} |f(x) - m_S(f)|} \tag{10.4}$$

where $m_S(f)$ is the mean target function value on S.

The relative absolute error indicates how the mean residual relates to the mean deviation of the target function from its mean. The latter can be thought of as the mean absolute error of a trivial mean value prediction model. The relative absolute error should be clearly less than 1 for any reasonable models, and preferably close to 0.

Example 10.2.5 The following R code defines a function for calculating the relative absolute error and demonstrates its application to the *Boston Housing* dataset.

```
rae <- function(pred.y, true.y)
{ mae(pred.y, true.y)/mean(abs(true.y-mean(true.y))) }

rae(predict(bh.tree, bh.test), bh.test$medv)
```

The obtained value of about 0.5 suggests that the quality of the evaluated model leaves somewhat to be desired, but is substantially better than simple mean value prediction.

10.2.6 Coefficient of determination

A more commonly used performance measure that is similarly motivated as the relative absolute error is the *coefficient of determination*, also referred to as R^2 or R-squared. Consider a quantity that relates to the mean square error in the same way as the relative absolute error relates to the mean absolute error. This could be called the *relative square error* (RSE) and defined as

$$\mathrm{rse}_{f,S}(h) = \frac{\sum_{x\in S}(f(x)-h(x))^2}{\sum_{x\in S}(f(x)-m_S(f))^2} = \frac{|S|\mathrm{mse}_{f,S}(h)}{(|S|-1)s_S^2(f)} \tag{10.5}$$

where $s_S^2(f)$ is the variance of the target function on S. The coefficient of determination is then obtained as 1's complement of the relative square error:

$$R^2_{f,S}(h) = 1 - \frac{\sum_{x\in S}(f(x)-h(x))^2}{\sum_{x\in S}(f(x)-m_f(S))^2} \tag{10.6}$$

This performance measure is typically interpreted as the fraction of the target function's variance explained by the evaluated model. If approaching 1, it indicates a nearly perfect model. Negative values indicate a totally useless model.

Example 10.2.6 The R code presented below defines a function that implements the calculation of the coefficient of determination and demonstrates its application to the *Boston Housing* dataset.

```
r2 <- function(pred.y, true.y)
{ 1 - length(true.y)*mse(pred.y, true.y)/((length(true.y)-1)*var(true.y)) }

r2(predict(bh.tree, bh.test), bh.test$medv)
```

The obtained value shows that the model managed to explain about 70% of the target function's variance, which may be not enough to consider a model fully satisfactory, but it is definitely useful.

10.2.7 Correlation

Residual-based performance measures are not necessarily well suited to some applications of regression models, where even large differences between predicted and true target function values may be acceptable as long as they exhibit roughly the same monotonic behavior with respect to attribute values. In such applications one requires the model's predictions to react to changes in attribute values in a way that most closely mimics the reaction of the target function: whenever the latter changes slightly the former should also change slightly and whenever the latter changes vastly the former should also change vastly, with the direction of the change preserved. This is required, in particular, whenever a regression model is used to support a decision making or optimization process, where the effects of several alternative decisions or solutions need to be predicted so that the most promising choice can be made. The error of predictions may be then of less importance than their utility for distinguishing between good and poor decisions or ordering candidate decisions in the order of preference. In such cases the linear or rank correlation of predicted and true target function values becomes a natural performance measure.

Example 10.2.7 The following R code demonstrates how the model created for the *Boston Housing* dataset can be evaluated using the linear and rank correlation.

```
cor(predict(bh.tree, bh.test), bh.test$medv, method="pearson")
cor(predict(bh.tree, bh.test), bh.test$medv, method="spearman")
```

The model's predictions turn out to correlate with the true target function values on the test set quite well, with both correlation coefficients approaching 0.85.

10.2.8 Weighted performance measures

Similarly as weight-sensitive algorithms use weighted training instances, one can use a set of weighted instances for model evaluation. Assuming a weight w_x is assigned to each $x \in S$, the definitions of the residual-based performance measures presented above can be rewritten as follows:

$$\mathrm{mae}_{f,S,w}(h) = \frac{\sum_{x \in S} w_x |f(x) - h(x)|}{\sum_{x \in S} w_x} \tag{10.7}$$

$$\mathrm{mse}_{f,S,w}(h) = \frac{\sum_{x \in S} w_x (f(x) - h(x))^2}{\sum_{x \in S} w_x} \tag{10.8}$$

$$\mathrm{rmse}_{f,S,w}(h) = \sqrt{\mathrm{mse}_{f,S,w}(h)} \tag{10.9}$$

$$\mathrm{rae}_{f,S,w}(h) = \frac{\sum_{x \in S} w_x |f(x) - h(x)|}{\sum_{x \in S} w_x |f(x) - m_S(f)|} \tag{10.10}$$

$$R^2_{f,S,w}(h) = 1 - \frac{\sum_{x \in S} w_x (f(x) - h(x))^2}{\sum_{x \in S} w_x (f(x) - m_S(f))^2} \tag{10.11}$$

These are weight-sensitive versions of the mean absolute error, the mean square error, the root mean square error, the relative absolute error, and the coefficient of determination, respectively.

Example 10.2.8 The following R code defines modified versions of the functions from the previous examples that optionally accept a weight vector as an additional argument and calculate weighted performance measures. When called without specifying weights, they behave as their weight-insensitive counterparts. The functions are applied to evaluate the performance of the model created for the *Boston Housing* dataset on the test subset, using a weight vector that doubles the importance of instances with the values of the target function above 25.

```
wmae <- function(pred.y, true.y, w=rep(1, length(true.y)))
{ weighted.mean(abs(true.y-pred.y), w) }

wmse <- function(pred.y, true.y, w=rep(1, length(true.y)))
{ weighted.mean((true.y-pred.y)^2, w) }

wrmse <- function(pred.y, true.y, w=rep(1, length(true.y)))
{ sqrt(wmse(pred.y, true.y, w)) }

wrae <- function(pred.y, true.y, w=rep(1, length(true.y)))
{ wmae(pred.y, true.y, w)/weighted.mean(abs(true.y-weighted.mean(true.y, w)), w) }

wr2 <- function(pred.y, true.y, w=rep(1, length(true.y)))
{
  1-weighted.mean((true.y-pred.y)^2, w)/
      weighted.mean((true.y-weighted.mean(true.y, w))^2, w)
}

  # double weight for medv>25
bh.wtest <- ifelse(bh.test$medv>25, 2, 1)

wmae(predict(bh.tree, bh.test), bh.test$medv, bh.wtest)
wmse(predict(bh.tree, bh.test), bh.test$medv, bh.wtest)
wrmse(predict(bh.tree, bh.test), bh.test$medv, bh.wtest)
wrae(predict(bh.tree, bh.test), bh.test$medv, bh.wtest)
wr2(predict(bh.tree, bh.test), bh.test$medv, bh.wtest)
```

10.2.9 Loss functions

The mean absolute error and the mean square error differ only in the function applied to model residuals before averaging them, which is the absolute value function for the former and the quadratic function of the latter. These two are the most common examples of *loss* functions used to measure the performance of regression models.

In general, a loss function is any function $\mathcal{L} : \mathcal{R}^2 \to \mathcal{R}$ that maps a pair consisting of a predicted and true target function value into a real number, representing the associated cost or regret, that should be considered for performance evaluation. The absolute and quadratic loss functions are simply defined as follows:

$$\mathcal{L}_{|}(f(x), h(x)) = |f(x) - h(x)| \tag{10.12}$$

$$\mathcal{L}_{2}(f(x), f(x)) = (f(x) - h(x))^2 \tag{10.13}$$

For the model h, target function f, and loss function $\mathcal{L}$ the mean loss on dataset S is then calculated as

$$\text{mls}_{f,S,\mathcal{L}}(h) = \frac{1}{|S|} \sum_{x \in S} \mathcal{L}(f(x), h(x)) \tag{10.14}$$

In general, arbitrarily selected loss functions can be employed for model evaluation, incorporating task-specific requirements or preferences. In particular, some loss functions can be *asymmetric*, i.e., assigning different costs to residuals with the same absolute value, but different signs, or *ϵ-insensitive*, i.e., assigning zero costs to residuals below some tolerance threshold.

Example 10.2.9 The following R code defines a function to calculate the mean loss as well as three loss functions: the absolute loss, the quadratic loss, and a simple loss function wrapper that can make an arbitrary loss function asymmetric by permitting specifying different multipliers applied for the positive and negative residuals. They are all demonstrated in application to the regression tree models for the *Boston Housing* dataset.

```
mls <- function(pred.y, true.y, loss) { mean(loss(pred.y, true.y)) }

loss.abs <- function(pred.y, true.y) { abs(true.y-pred.y) }

loss.square <- function(pred.y, true.y) { (true.y-pred.y)^2 }

loss.asymmetric <- function(loss, p=1, n=1)
{
  function(pred.y, true.y)
  {
    ifelse(res(pred.y, true.y)>0, p*loss(pred.y, true.y), n*loss(pred.y, true.y))
  }
}

mls(predict(bh.tree, bh.test), bh.test$medv, loss.abs)
mls(predict(bh.tree, bh.test), bh.test$medv, loss.square)
mls(predict(bh.tree, bh.test), bh.test$medv, loss.asymmetric(loss.abs, 2, 1))
```

10.3 Evaluation procedures

Evaluation procedures for regression models address the same challenge of reliably assessing a model's expected performance on new data as discussed in Section 7.3 for classification models, which requires the separation of the validation or test set from the training set without degrading the model quality due to insufficient training data. It also involves the same difficulties and essentially the same techniques for overcoming them. The extensive discussion of those presented in Section 7.3 will not be repeated here, but it fully applies to regression model evaluation as well. In particular, it is important to keep in mind the following:

- The purpose of evaluation procedures is to evaluate *modeling procedures* (consisting of a regression algorithm, its parameters, applied data transformations and anything else other than the data itself that affects the created model) rather than individual models.
- One or more models have to be created using a given modeling procedure and their performance measured to evaluate the modeling procedure.
- The evaluation should reliably estimate the true performance of the model created using the same modeling procedure on the whole available dataset.
- The evaluation reliability may suffer from *bias* (resulting from evaluating models created on subsets of the available data, which may reduce their quality compared

to all-data models) and *variance* (resulting from using limited randomly selected test/validation subsets).

The remaining contents of this section are limited to a very concise review of the same set of evaluation procedures, with examples of their application to regression models. These procedures can all be used to generate vectors of predicted and true target function values, making it possible to calculate arbitrary performance measures based on these vectors.

10.3.1 Hold-out

The hold-out evaluation procedure separates training and validation or test data in the simplest possible way: a subset of the available labeled dataset is selected randomly as the training set and the remaining instances are *held out* for the purpose of model evaluation. It does not handle the bias vs. variance tradeoff very well: with sufficiently many instances left for low-variance evaluation there may be too little training instances to ensure adequate model quality and one may end up with a considerable pessimistic bias resulting from measuring the performance of a poor model.

Example 10.3.1 Notice that all the examples of performance measures presented above used the hold-out procedure, randomly dividing the *Boston Housing* dataset into training and test subsets. By repeating the random partitioning used for these examples several times and recalculating the performance measures, we would likely observe considerably different results, due to the high variance of this evaluation procedure. The following R code uses the `holdout` function to perform an automated hold-out procedure that repeats the random dataset partitioning, model building, and prediction several times, and collects the observed predicted and true target function values. This procedure is applied to perform 10-times repeated hold-out evaluation of regression tree models for the *Boston Housing* dataset (2/3 of which is used for training with the remaining 1/3 used for testing). All previously discusses performance measures are applied to the generated predictions.

Ex. 7.3.1
`dmr.claseval`

```
  # hold-out regression tree evaluation for the Boston Housing data
bh.ho <- holdout(rpart, medv~., BostonHousing, n=10)
mae(bh.ho$pred, bh.ho$true)
mse(bh.ho$pred, bh.ho$true)
rmse(bh.ho$pred, bh.ho$true)
rae(bh.ho$pred, bh.ho$true)
r2(bh.ho$pred, bh.ho$true)
cor(bh.ho$pred, bh.ho$true, method="pearson")
cor(bh.ho$pred, bh.ho$true, method="spearman")
```

The `holdout` function, originally developed for classification, handles regression model evaluation as well. The same applies to other implementations of model evaluation procedures demonstrated in subsequent examples.

10.3.2 Cross-validation

A more refined evaluation procedure that better handles the bias vs. variance tradeoff is k-fold cross-validation that consists in splitting the available dataset at random into k

disjoint equal-size subsets $D_1, D_2, \ldots, D_k$ and then iterating over these subsets. On the ith iteration, a model is built using $T_i = \bigcup_{j \neq i} D_j$ as the training set, and applied to generate predictions on $Q_i = D_i$. For typical values of k (between 5 and 20) this requires substantially more computation than the simple hold-out procedure, but helps to reduce the bias while keeping the variance under control. It can be repeated several times for even further variance reduction. By choosing k one can tradeoff the bias, variance, and computational expense of the cross-validation procedure, as discussed in Section 7.3.4.

Example 10.3.2 The following R code applies the `crossval` function to evaluate predictions of regression tree models for the *Boston Housing* dataset using k-fold cross-validation with a few different k values. Unlike in the previous example, only a single performance measure – the mean square error – is calculated, but any other performance indicator can be used instead.

Ex. 7.3.2
`dmr.claseval`

```
  # regression tree cross-validation for the BostonHousing data
bh.cv3 <- crossval(rpart, medv~., BostonHousing, k=3)
mse(bh.cv3$pred, bh.cv3$true)
bh.cv5 <- crossval(rpart, medv~., BostonHousing, k=5)
mse(bh.cv5$pred, bh.cv5$true)
bh.cv10 <- crossval(rpart, medv~., BostonHousing, k=10)
mse(bh.cv10$pred, bh.cv10$true)
bh.cv20 <- crossval(rpart, medv~., BostonHousing, k=20)
mse(bh.cv20$pred, bh.cv20$true)
```

10.3.3 Leave-one-out

The leave-one-out validation procedure is an extreme form of k-fold cross-validation in which k is set to the number of instances in the dataset. The procedure iterates over all instances, using the model built on the dataset with one instance removed to make prediction for this instance. This is hardly applicable to large datasets due to the computational expense of building as many models as instances available, but it may be a reasonable evaluation procedure for very small datasets. It has no pessimistic bias, but its variance is high due to using single instances for evaluation.

Example 10.3.3 The following R code uses the `leave1out` function to perform the leave-one-out evaluation procedure for the *Boston Housing* data.

Ex. 7.3.3
`dmr.claseval`

```
  # leave-one-out regression tree evaluation for the BostonHousing data
bh.l1o <- leave1out(rpart, medv~., BostonHousing)
mse(bh.l1o$pred, bh.l1o$true)
```

10.3.4 Bootstrapping

Bootstrapping estimation techniques are based on drawing multiple *bootstrap samples* from the data. Each bootstrap sample, typically of the same size as the original dataset, is drawn uniformly at random *with replacement*. A single bootstrap sample can be expected to contain

about 63.2% of instances from the original dataset, some replicated. The missing instances, also called *out-of-bag* (OOB) instances, stand for about 36.8% of the dataset. The idea of bootstrapping is to use a bootstrap sample as the training set and to evaluate the resulting model on the OOB instances.

Plain bootstrap performance estimators, obtained on OOB instances only, have low variance, but are usually pessimistically biased, since only about 63.2% of available instances are used for training. This is compensated by the .632 bootstrap procedure which produces the final performance estimator as a weighted average of the (overly pessimistic) estimator obtained on OOB instances and the (overly optimistic) estimator that can be obtained by training and evaluating a model on the full dataset. Using the mean square error as the performance measure, this can be presented as follows:

$$\text{mse}_D^{.632} = 0.632 \frac{1}{M} \sum_{i=1}^{M} \text{mse}_{f, D_i'}(h_i) + 0.368 \text{mse}_{f, D}(h) \tag{10.15}$$

where D is the available dataset from which M bootstrap samples $D_1, D_2, \ldots, D_M$ are drawn, h_i is the model built using D_i as the training set, D_i' is the corresponding set of OOB instances on which the model is evaluated, and h is the model built on the full dataset D.

The .632 bootstrap estimator has been most often employed for classification models, but it can be prove similarly useful for regression models. It can be expected to work best with regression algorithms that do not heavily overfit. For models that fit the training set to a great extent the estimator tends to be optimistically biased.

Similarly as for other evaluation procedures, we can use bootstrapping to generate vectors of predicted and true target function values and then calculate arbitrary performance measures based on these vectors. This is straightforward with just one caveat: to apply the .632 bootstrap or another similar weighting scheme, the vectors of predictions and true target function values must be accompanied by a vector of weights, and the calculation of selected performance indicators must incorporate these weights. This was more extensively discussed in Section 7.3.6.

Example 10.3.4 To illustrate the bootstrapping approach to model evaluation, the following R code uses the `bootstrap` function. It draws a specified number of bootstrap samples from the provided dataset and uses them to build models which it subsequently applies to generate predictions for OOB instances. If the `w` parameter is set to a value less than 1, it also creates a full-data model and applies it to generate predictions on the full dataset. These receive a weight of $1 - w$, whereas the former receive a weight of w, divided by the number of bootstrap samples. For $w = 0.632$ this is equivalent to the .632 bootstrap procedure. The procedure returns a data frame with predictions, target function values, and the corresponding weights. The function is applied to evaluate the regression tree model for the *Boston Housing* data, with `w` set to 1 (plain bootstrap, likely to be pessimistically biased) and to 0.632. A relatively small number of 20 bootstrap samples is used. The produced output is used to calculate the mean square error.

Ex. 7.3.4
`dmr.claseval`

```
  # 20x bootstrap regression tree evaluation for the BostonHousing data
bh.bs20 <- bootstrap(rpart, medv~., BostonHousing, w=1, m=20)
mse(bh.bs20$pred, bh.bs20$true)

  # 20x .632 bootstrap regression tree evaluation for the BostonHousing data
bh.632bs20 <- bootstrap(rpart, medv~., BostonHousing, m=20)
wmse(bh.632bs20$pred, bh.632bs20$true, bh.632bs20$w)
```

The .632 bootstrap performance estimators suggest a better performance level than the plain bootstrap ones (with $w = 1$), which agrees with the expectation of the pessimistic bias of the latter. The former may appear overoptimistic, though, given the error estimates previously obtained with the cross-validation and leave-one-out procedures.

10.3.5 Choosing the right procedure

As discussed in Section 7.3.7, the choice of the right evaluation procedure for a given application depends on the accepted level of the bias vs. variance tradeoff, the size of the dataset, the regression algorithm, and the available computational resources. Most evidence suggests that the bias vs. variance tradeoff is best handled by the *k*-fold cross-validation procedure with *k* set to 10 or 20, particularly repeated several times. Whereas the .632 bootstrap procedure should also yield nearly unbiased and low-variance performance estimators for algorithms that are not prone to overfitting, in practice it is often hardly possible to completely eliminate the risk of overfitting *a priori*, which result in .632 bootstrap performance estimates being overly optimistic. The leave-one-out procedure should be rather avoided, except for small datasets where all other evaluation procedures would be considerably biased. The hold-out procedure is, conversely, best suited to very large datasets, for which other evaluation procedures would be too expensive and for which considerably smaller training samples would have to be used anyway due to computational constraints.

Example 10.3.5 To illustrate the properties of the different evaluation procedures discussed above, the following R code applies the `eval.bias.var` function to perform a simple experiment to observe their bias and variance. This basically reproduces Example 7.3.5 in the regression context. A random $2/3$ subset is drawn from the provided dataset and considered a simulated "available" dataset, with the remaining $1/3$ subset considered a simulated "new" dataset. The "new" dataset is used to calculate an estimate of the true performance of the model built on the "available" dataset (the mean square error in this case). The hold-out, cross-validation, leave-one-out, and bootstrap evaluation procedures are then run on the "available" dataset to produce their performance estimates. This experiment is repeated a number of times with all the obtained results collected, to finally calculate the estimated bias and variance of each evaluation procedure.

Ex. 7.3.5
dmr.claseval

The function is applied to observe the bias and variance of different evaluation procedures when applied to evaluate regression tree models for the *Boston Housing* dataset. The results are used to produce a boxplot of the error estimates produced by particular evaluation procedures, with a horizontal line designating the mean true error estimated on the "new" dataset. Barplots of the bias and variance of all the evaluation procedures are also produced. The plots presented in Figure 10.2 are based on 200 evaluation repetitions, which takes some considerable time. The line that runs this full experiment is commented out and another one, that runs a 10-times repeated evaluation experiment, is recommended instead for a quick illustration.

```
  # the commented line runs a 200-repetition experiment, which takes a long time
#bh.ebv <- eval.bias.var(rpart, medv~., BostonHousing, perf=mse, wperf=wmse, n=200)
  # this can be used for a quick illustration
bh.ebv <- eval.bias.var(rpart, medv~., BostonHousing, perf=mse, wperf=wmse, n=10)

boxplot(bh.ebv$performance[,-1], main="Error", las=2)
lines(c(0, 13), rep(mean(bh.ebv$performance[,1]), 2), lty=2)
barplot(bh.ebv$bias, main="Bias", las=2)
barplot(bh.ebv$variance, main="Variance", las=2)
```

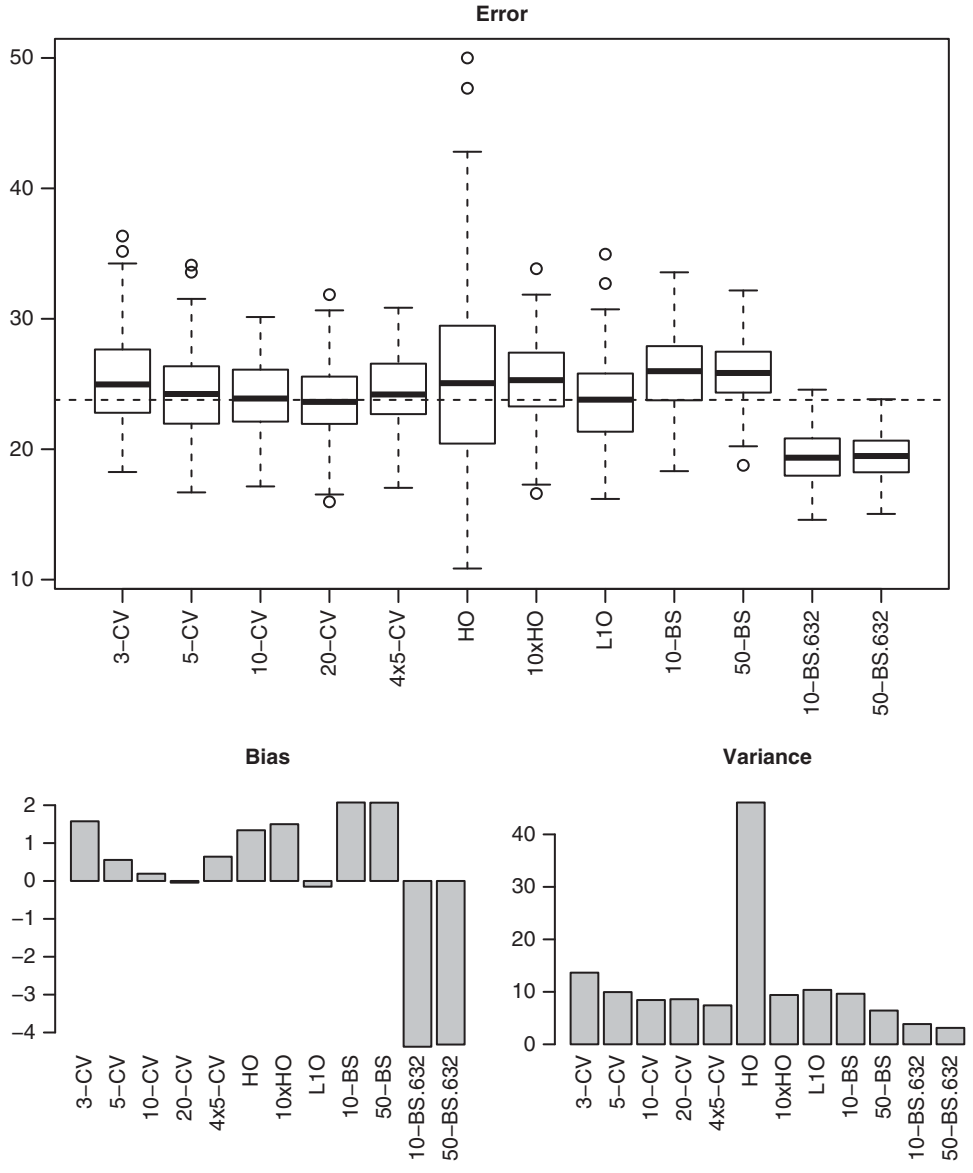

Figure 10.2 The mean square distribution, bias, and variance for different evaluation procedures.

The results agree with expectations and are consistent with those observed in Example 7.3.5, confirming the high bias of hold-out (both single and repeated) and 3-fold cross-validation, the very small bias of 20-fold cross-validation and leave-one-out, the pessimistic bias of the plain bootstrap procedure, and the optimistic bias of the .632 bootstrap procedure. With respect to variance, 4×5-fold cross-validation is the best of the cross-validation procedures, the .632 bootstrap is clearly better, even with just 10 bootstrap

samples, and the single hold-out is by far the worst. The repetition reduces the variance of hold-out considerably, as expected. Overall, the 4×5-fold cross-validation procedure appears to achieve a good compromise between bias and variance.

10.4 Conclusion

Performance measures used to assess the quality of regression models differ from those used for classification models for obvious reasons. With a numeric target function the quality of model predictions is no longer assessed by counting the number of mistakes, but rather calculating the differences or correlations between them. The former leads to the most common residual-based performance measures: the mean absolute error and the mean square error, as well as their more easily interpretable counterparts (the relative absolute error, the root mean square error, and the coefficient of determination). The latter may be preferred in some special model application scenarios where capturing the monotonic behavior of the target function with respect to attribute values is more important than achieving small differences.

Despite different performance measures, the purpose and overall methodology of regression model evaluation remains the same as for classification models. The same evaluation procedures are used, facing the same challenge of controlling evaluation bias and variance. And as for classification, the biggest risk of failing to reliably evaluate regression models is not associated with choosing inadequate performance measures or evaluation procedures, but with insufficiently careful separation of the data used to create the model and the data on which the evaluation is performed. The former should be understood broadly, including all data subsets on which any decisions that may impact the final model are based. These are, in particular, data transformation, parameter tuning, and attribute selection decisions.

10.5 Further readings

With evaluation procedures for regression models being the same as for classification models, the same references as provided in Chapter 7 also apply here, including both books (e.g., Abu-Mostafa *et al*. 2012; Cios *et al*. 2007; Han *et al*. 2011; Hand *et al*. 2001; Tan *et al*. 2013; Witten *et al*. 2011) and research articles (e.g., Arlot and Celisse 2010; Efron 1983; Efron and Tibshirani 1997). Some of the former also present basic regression-specific predictive performance measures (e.g., Hand *et al*. 2001; Witten *et al*. 2011), but the statistical literature on regression modeling gives a much wider and more in-depth coverage of those (e.g., Draper and Smith 1998; Freedman 2009; Glantz and Slinker 2000). They discuss the issue of assessing the performance of regression models – referred to as goodness of fit in standard statistical terminology – much more extensively than this chapter. Even if they are often presented – as this term suggests – in application to evaluating the training performance only, they can be clearly applied to nontraining data and used, in combination with evaluation procedures, to assess the true performance. For historical reasons, this may be not such a well-established practice as for classification model evaluation, though (Picard and Cook 1984; Snee 1977).

It is worthwhile to mention the close relationship between the evaluation of regression models and that of measurements in experimental physics or engineering. The latter was extensively discussed by Taylor (1996). In particular, popular types of residual-based performance measures are common for these two domains. Evaluating the predictive performance

(or the goodness of fit) of regression models is, however, only one of the several aspects of statistical regression model diagnostics, which also include statistics and tests for verifying the model's underlying assumptions or quantifying the impact of particular attributes and training instances on model parameters and predictions (Cook and Weisberg 1982; Davison and Tsai 1992).

References

Abu-Mostafa YS, Magdon-Ismail M and Lin HT 2012 *Learning from Data*. AMLBook.

Arlot S and Celisse A 2010 A survey of cross-validation procedures for model selection. *Statistics Surveys* **4**, 40–79.

Cios KJ, Pedrycz W, Swiniarski RW and Kurgan L 2007 *Data Mining: A Knowledge Discovery Approach*. Springer.

Cook RD and Weisberg S 1982 *Residuals and Influence in Regression*. Chapman and Hall.

Davison AC and Tsai CL 1992 Regression model diagnostics. *International Statistical Review* **60**, 337–353.

Draper NR and Smith H 1998 *Applied Regression Analysis* 3rd edn. Wiley.

Efron B 1983 Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association* **78**, 316–331.

Efron B and Tibshirani R 1997 Improvements on cross-validation: The .632+ bootstrap method. *Journal of the American Statistical Association* **92**, 548–560.

Freedman DA 2009 *Statistical Models: Theory and Practice*. Cambridge University Press.

Glantz SA and Slinker BS 2000 *Primer of Applied Regression and Analysis of Variance* 2nd edn. McGraw-Hill.

Han J, Kamber M and Pei J 2011 *Data Mining: Concepts and Techniques* 3rd edn. Morgan Kaufmann.

Hand DJ, Mannila H and Smyth P 2001 *Principles of Data Mining*. MIT Press.

Picard RR and Cook RD 1984 Cross-validation of regression models. *Journal of the American Statistical Association* **79**, 575–583.

Snee RD 1977 Validation of regression models: Methods and examples. *Technometrics* **19**, 415–428.

Tan PN, Steinbach M and Kumar V 2013 *Introduction to Data Mining* 2nd edn. Addison-Wesley.

Taylor JR 1996 *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements* 2nd edn. University Science Books.

Witten IH, Frank E and Hall MA 2011 *Data Mining: Practical Machine Learning Tools and Techniques* 3rd edn. Morgan Kaufmann.