Chapter 11 Black Box Machine-Learning Methods: Neural Networks and Support Vector Machines

In this Chapter, we are going to cover two very powerful machine-learning algorithms. These techniques have complex mathematical formulations; however, efficient algorithms and reliable software packages have been developed to utilize them for various practical applications. We will (1) describe Neural Networks as analogues of biological neurons; (2) develop hands-on a neural net that can be trained to compute the square-root function; (3) describe support vector machine (SVM) classification; and (4) complete several case-studies, including optical character recognition (OCR), the Iris flowers, Google Trends and the Stock Market, and Quality of Life in chronic disease.

Later, in Chap. 23, we will provide more details and additional examples of deep neural network learning. For now, let's start by exploring the *magic* inside the machine learning black box.

11.1 Understanding Neural Networks

11.1.1 From Biological to Artificial Neurons

An `Artificial Neural Network (ANN)` model mimics the biological brain response to multisource inputs, e.g., sensory-motor stimuli. ANNs simulate the brain using networks of interconnected neuron cells to create massively parallel processors. Of course, ANNs use networks of artificial nodes, not brain cells, to train data.

When we have three signals (or inputs) x_1, x_2 and x_3, the first step is weighting the features (w's) according to their importance. Then, the weighted signals are summed by the "neuron cell" and this sum is passed on according to an **activation function** denoted by **f**. The last step is generating an output **y** at the end of the process. A typical output will have the following mathematical relationship to the inputs.

I. D. Dinov, *Data Science and Predictive Analytics*,
https://doi.org/10.1007/978-3-319-72347-1_11

$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i\right).$$

There are three important components for building a neural network:

- **Activation function**: transforms weighted and summed inputs to an output.
- **Network topology**: describes the number of "neuron cells", the number of layers and manner in which the cells are connected.
- **Training algorithm**: how to determine weights w_i.

Let's look at each of these components one by one.

11.1.2 Activation Functions

One of the functions, known as threshold activation function, triggers an output signal once a specified input threshold has been attained (Fig. 11.1).

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}.$$

This is the simplest form for activation functions. It is rarely used in real world situations. The most commonly used alternative is the sigmoid activation function, where $f(x) = \frac{1}{1+e^{-x}}$. Here, e is Euler's natural number, which is also the base of the natural logarithm function. The output signal is no longer binary but can be any real number ranged from 0 to 1 (Fig. 11.2).

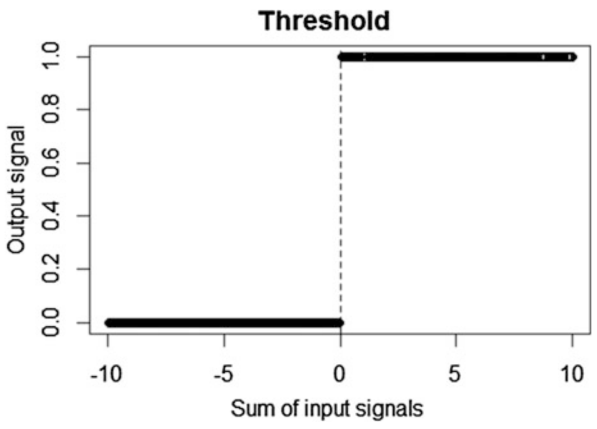

Fig. 11.1 An example of a hard threshold activation function, $f(x)$

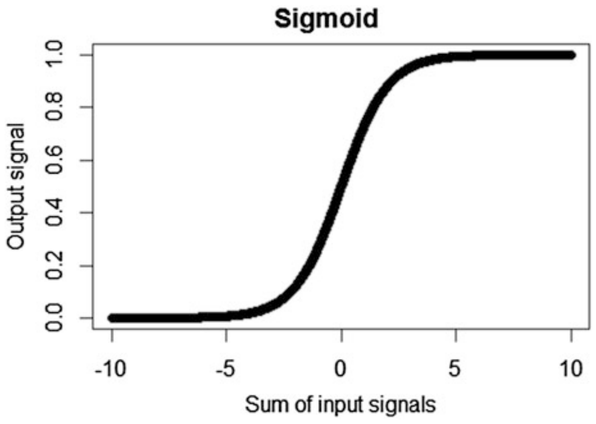

Fig. 11.2 The S-shaped sigmoid activation function

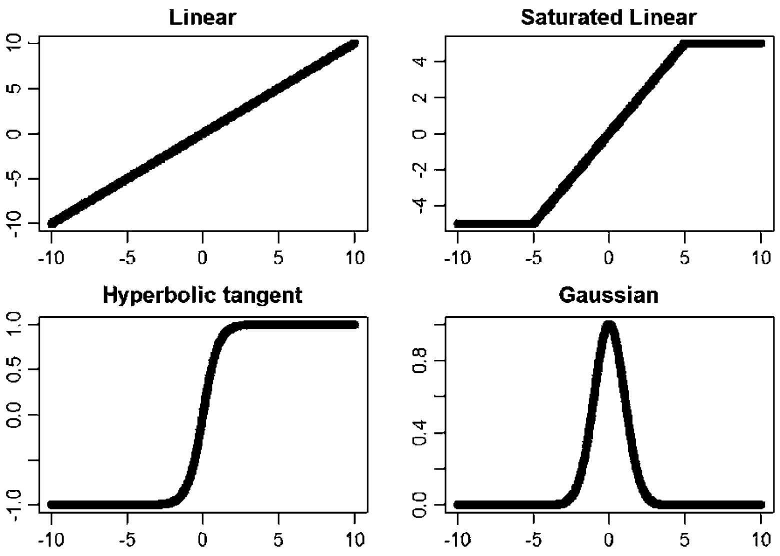

Fig. 11.3 Alternative types of activation functions

Other activation functions might also be useful, Fig. 11.3.

Basically, we can chose a proper activation function based on the corresponding codomain of the function. For example, with hyperbolic tangent activation function, we can only have outputs ranging from -1 to 1 regardless of the input. With linear function we can go from $-\infty$ to $+\infty$. Our Gaussian activation function will give us a model called *Radial Basis Function* network (Fig. 11.3).

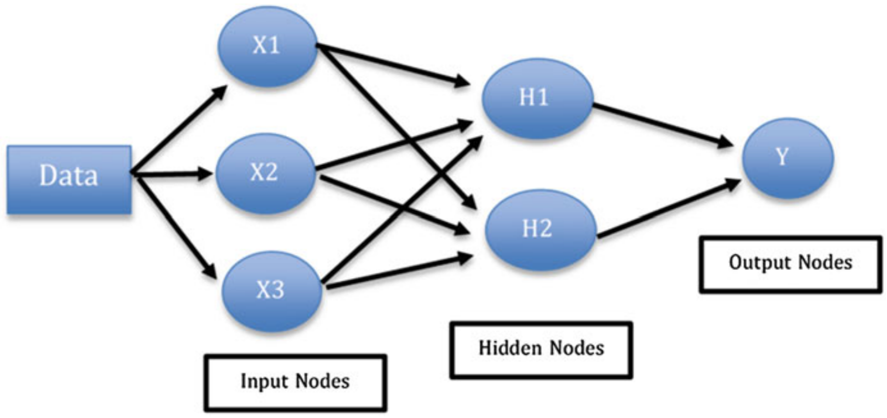

Fig. 11.4 A schematic of a two-layer neural network

11.1.3 Network Topology

Number of layers: The *x*'s or features in the dataset are called **input nodes** while the predicted values are called the **output nodes**. Multilayer networks include multiple hidden layers. Figure 11.4 shows a two layer neural network.

When we have multiple layers, the information flow could be complicated.

11.1.4 The Direction of Information Travel

The arrows in the last graph (with multiple layers) suggest a feed forward network. In such a network, we can also have multiple outcomes modeled simultaneously (Fig. 11.5).

Alternatively, in a recurrent network (feedback network), information can also travel backwards in loops (or delay). This is illustrated in Fig. 11.6, where the short-term memory increases the power of recurrent networks dramatically. However, in practice, recurrent networks are rarely used.

11.1.5 The Number of Nodes in Each Layer

The number of input nodes and output nodes are predetermined by the dataset and the predictive variables. The number we can specify determines the hidden nodes in the model. To simplify the model, our goal is to add the least number of hidden nodes possible when the model performance remains reasonable.

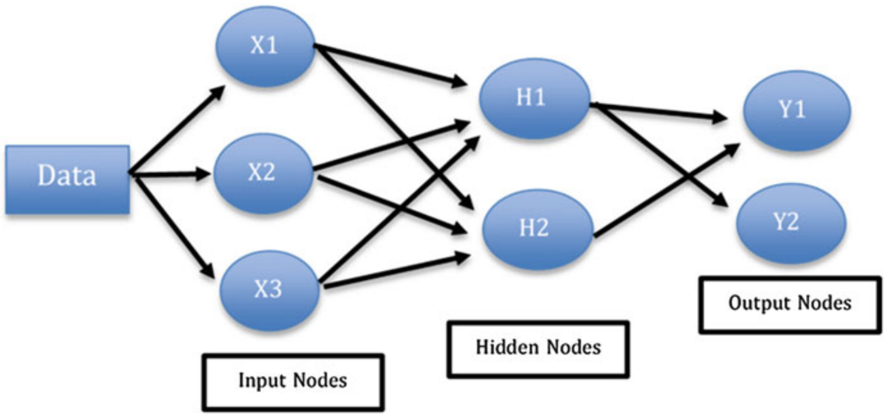

Fig. 11.5 A multi-output neural network example

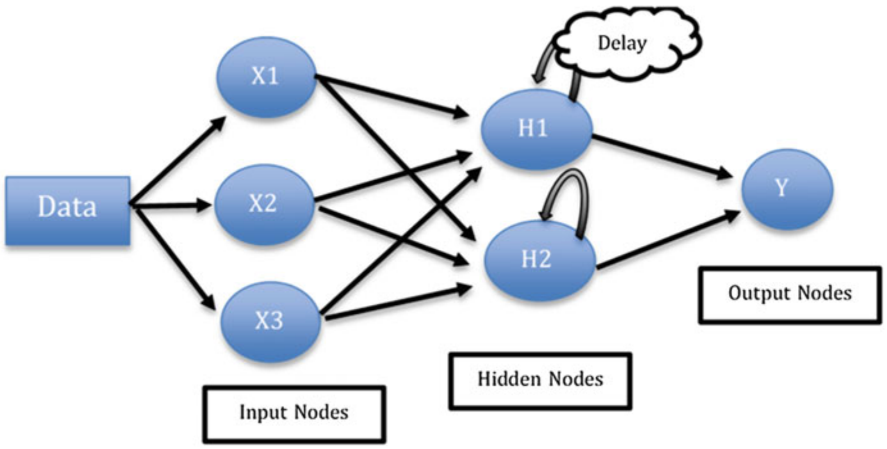

Fig. 11.6 A schematic of a delay (feedback) neural network

11.1.6 Training Neural Networks with Backpropagation

This algorithm could determine the weights in the model using the strategy of back-propagating errors. First, we assign random weights (but all weights must be non-trivial). For example, we can use normal distribution, or any other random process, to assign initial weights. Then we adjust the weights iteratively by repeating the process until certain convergence or stopping criterion is met. Each iteration contains two phases.

- Forward phase: from input layer to output layer using current weights. Outputs are produced at the end of this phase, and
- Backward phase: compare the outputs and true target values. If the difference is significant, we change the weights and go through the forward phase, again.

In the end, we pick a set of weights, which correspond to the least total error, to be the final weights in our network.

11.2 Case Study 1: Google Trends and the Stock Market: Regression

11.2.1 Step 1: Collecting Data

In this case study, we are going to use the Google trends and stock market dataset. A doc file with the meta-data and the CSV data are available on the Case-Studies Canvas Site. These daily data (between 2008 and 2009) can be used to examine the associations between Google search trends and the daily marker index - Dow Jones Industrial Average.

Variables

- **Index**: Time Index of the Observation
- **Date**: Date of the observation (Format: YYYY-MM-DD)
- **Unemployment**: The Google Unemployment Index tracks queries related to "unemployment, social, social security, unemployment benefits" and so on.
- **Rental**: The Google Rental Index tracks queries related to “rent, apartments, for rent, rentals,” etc. RealEstate: The Google Real Estate Index tracks queries related to “real estate, mortgage, rent, apartments” and so on.
- **Mortgage**: The Google Mortgage Index tracks queries related to "mortgage, calculator, mortgage calculator, mortgage rates".
- **Jobs**: The Google Jobs Index tracks queries related to "jobs, city, job, resume, career, monster" and so forth.
- **Investing**: The Google Investing Index tracks queries related to "stock, finance, capital, yahoo finance, stocks", etc.
- **DJI_Index**: The Dow Jones Industrial (DJI) index. These data are interpolated from 5 records per week (Dow Jones stocks are traded on week-days only) to 7 days per week to match the constant 7-day records of the Google-Trends data.
- **StdDJI**: The standardized-DJI Index computed by: StdDJI = 3 + (DJI-11,091)/1,501, where m = 11,091 and s = 1,501 are the approximate mean and standard-deviation of the DJI for the period (2005–2011).

- **30-Day Moving Average Data Columns**: The 8 variables below are the 30-day moving averages of the 8 corresponding (raw) variables above.
  – *Unemployment30MA, Rental30MA, RealEstate30MA, Mortgage30MA, Jobs30MA, Investing30MA, DJI_Index30MA, StdDJI_30MA*.
- **180-Day Moving Average Data Columns**: The 8 variables below are the 180-day moving averages of the 8 corresponding (raw) variables.
  – *Unemployment180MA, Rental180MA, RealEstate180MA, Mortgage180MA, Jobs180MA, Investing180MA, DJI_Index180MA, StdDJI_180MA*.

Here we use the `RealEstate` as our dependent variable. Let's see if the Google Real Estate Index could be predicted by other variables in the dataset.

11.2.2 Step 2: Exploring and Preparing the Data

First, we need to load the dataset into R.

```
google<-read.csv("https://umich.instructure.com/files/416274/download?downlo
ad_frd=1", stringsAsFactors = F)
```

Let's delete the first two columns, since the only goal is to predict Google Real Estate Index with other indexes and DJI.

```
google<-google[, -c(1, 2)]
str(google)

## 'data.frame':    731 obs. of  24 variables:
##  $ Unemployment      : num  1.54 1.56 1.59 1.62 1.64 1.64 1.71 1.85 1.82
1.78 ...
##  $ Rental            : num  0.88 0.9 0.92 0.92 0.94 0.96 0.99 1.02 1.02 1
.01 ...
##  $ RealEstate        : num  0.79 0.81 0.82 0.82 0.83 0.84 0.86 0.89 0.89
0.89 ...
##  $ Mortgage          : num  1 1.05 1.07 1.08 1.1 1.11 1.15 1.22 1.23 1.24
...
##  $ Jobs              : num  0.99 1.05 1.1 1.14 1.17 1.2 1.3 1.41 1.43 1.4
4 ...
##  $ Investing         : num  0.92 0.94 0.96 0.98 0.99 0.99 1.02 1.09 1.1 1
.1 ...
##  $ DJI_Index         : num  13044 13044 13057 12800 12827 ...
##  $ StdDJI            : num  4.3 4.3 4.31 4.14 4.16 4.16 4.16 4 4.1 4.17 .
..
##  $ Unemployment_30MA : num  1.37 1.37 1.38 1.38 1.39 1.4 1.4 1.42 1.43 1.
44 ...
##  $ Rental_30MA       : num  0.72 0.72 0.73 0.73 0.74 0.75 0.76 0.77 0.78
0.79 ...
##  $ RealEstate_30MA   : num  0.67 0.67 0.68 0.68 0.68 0.69 0.7 0.7 0.71 0.
72 ...
##  $ Mortgage_30MA     : num  0.98 0.97 0.97 0.97 0.98 0.98 0.98 0.99 0.99
```

```
1 ...
##  $ Jobs_30MA         : num  1.06 1.06 1.05 1.05 1.05 1.05 1.05 1.06 1.07
1.08 ...
##  $ Investing_30MA    : num  0.99 0.98 0.98 0.98 0.98 0.97 0.97 0.97 0.98
0.98 ...
##  $ DJI_Index_30MA    : num  13405 13396 13390 13368 13342 ...
##  $ StdDJI_30MA       : num  4.54 4.54 4.53 4.52 4.5 4.48 4.46 4.44 4.41 4
.4 ...
##  $ Unemployment_180MA: num  1.44 1.44 1.44 1.44 1.44 1.44 1.44 1.44 1.44
1.44 ...
##  $ Rental_180MA      : num  0.87 0.87 0.87 0.87 0.87 0.87 0.86 0.86 0.86
0.86 ...
##  $ RealEstate_180MA  : num  0.89 0.89 0.88 0.88 0.88 0.88 0.88 0.88 0.88
0.87 ...
##  $ Mortgage_180MA    : num  1.18 1.18 1.18 1.18 1.17 1.17 1.17 1.17 1.17
1.17 ...
##  $ Jobs_180MA        : num  1.24 1.24 1.24 1.24 1.24 1.24 1.24 1.24 1.24
1.24 ...
##  $ Investing_180MA   : num  1.04 1.04 1.04 1.04 1.04 1.04 1.04 1.04 1.04
1.04 ...
##  $ DJI_Index_180MA   : num  13493 13492 13489 13486 13482 ...
##  $ StdDJI_180MA      : num  4.6 4.6 4.6 4.6 4.59 4.59 4.59 4.58 4.58 4.58
...
```

As we can see from the structure of the data, these indices and DJI have different ranges. We should `rescale` the data. In Chap. 6, we learned that normalizing these features using our own `normalize()` function provides one solution. We can use `lapply()` to apply the `normalize()` function to each column.

```
normalize <- function(x) {
return((x - min(x)) / (max(x) - min(x)))
}
google_norm<-as.data.frame(lapply(google, normalize))
summary(google_norm$RealEstate)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.0000  0.4615  0.6731  0.6292  0.8077  1.0000
```

Looks like all the vectors are normalized into the [0, 1] range.

The next step would be to split the `google` dataset into training and testing subsets. This time we will use the `sample()` and `floor()` function to separate the training and testing sets. `sample()` is a function to create a set of indicators for row numbers. We can subset the original dataset with random rows using these indicators. `floor()` takes a number x and returns the closest integer to x

```
sample(row, size)
```

- row: rows in the dataset that you want to select from. If you want to select all of the rows, you can use $nrow(data)$ or $1 : nrow(data)$(single number or vector).
- size: how many rows you want for your subset.

```
sub<-sample(nrow(google_norm), floor(nrow(google_norm)*0.75))
google_train<-google_norm[sub, ]
google_test<-google_norm[-sub, ]
```

We are good to go and can move forward to the model training phase.

11.2.3 Step 3: Training a Model on the Data

Here, we use the function `neuralnet()` in package `neuralnet`. `neuralnet` returns a NN object containing:

- call: the matched call.
- response: extracted from the data argument.
- covariate: the variables extracted from the data argument.
- model.list: a list containing the covariates and the response variables extracted from the formula argument.
- err.fct and act.fct: the error and activation functions.
- net.result: a list containing the overall result of the neural network for every repetition.
- weights: a list containing the fitted weights of the neural network for every repetition.
- result.matrix: a matrix containing the reached threshold, needed steps, error, AIC, BIC, and weights for every repetition. Each column represents one repetition.

```
m<-neuralnet(target~predictors, data=mydata, hidden=1),
```
where:

- target: variable we want to predict.
- predictors: predictors we want to use. Note that we cannot use "." to denote all the variables in this function. We have to add all predictors one by one to the model.
- data: training dataset.
- hidden: number of hidden nodes that we want to use in the model. By default, it is set to one.

```
# install.packages("neuralnet")
library(neuralnet)
google_model<-neuralnet(RealEstate~Unemployment+Rental+Mortgage+Jobs+Investi
ng+DJI_Index+StdDJI, data=google_train)
plot(google_model)
```

Figure 11.7 shows that we have only one hidden node. `Error` stands for the sum of squared errors and `Steps` is how many iterations the model had to go through. These outputs could be different when you run the exact same code because the weights are stochastically generated.

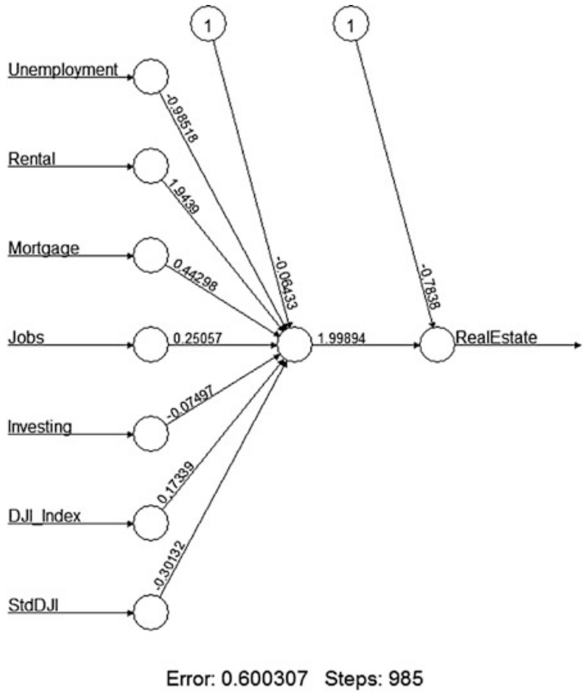

Fig. 11.7 A simple neural network predicting the real estate prices using Google market data

11.2.4 Step 4: Evaluating Model Performance

Similar to the `predict()` function that we have mentioned in previous Chapters, `compute()` is an alternative method that helps with ANN model predictions.

```
p<-compute(m, test)
```

- m: a trained neural networks model.
- test: the test dataset. This dataset should only contain the same type of predictors in the neural network model.

In our model, we picked `Unemployment`, `Rental`, `Mortgage`, `Jobs`, `Investing`, `DJI_Index`, `StdDJI` as our predictors. So, we need to find these corresponding column numbers in the test dataset (1, 2, 4, 5, 6, 7, 8, respectively).

```
google_pred<-compute(google_model, google_test[, c(1:2, 4:8)])
pred_results<-google_pred$net.result
cor(pred_results, google_test$RealEstate)

##              [,1]
## [1,] 0.9653369986
```

As mentioned in Chap. 9, we can still use the correlation between predicted results and observed Real Estate Index to evaluate the data. A correlation over 0.96 is very good for real world datasets. Could this still be improved?

11.2.5 Step 5: Improving Model Performance

This time we will include four hidden nodes in the model. Let's see what results we can get from this more elaborate ANN model (Fig. 11.8).

```
google_model2<-neuralnet(RealEstate~Unemployment+Rental+Mortgage+Jobs+Invest
ing+DJI_Index+StdDJI, data=google_train, hidden = 4)
plot(google_model2)
```

Although the graph looks complicated, we have smaller `Error`, or sum of squared errors. Neural net models may be used for *classification* and *regression*, which we will see in the next part. Let's first try regression tree modeling.

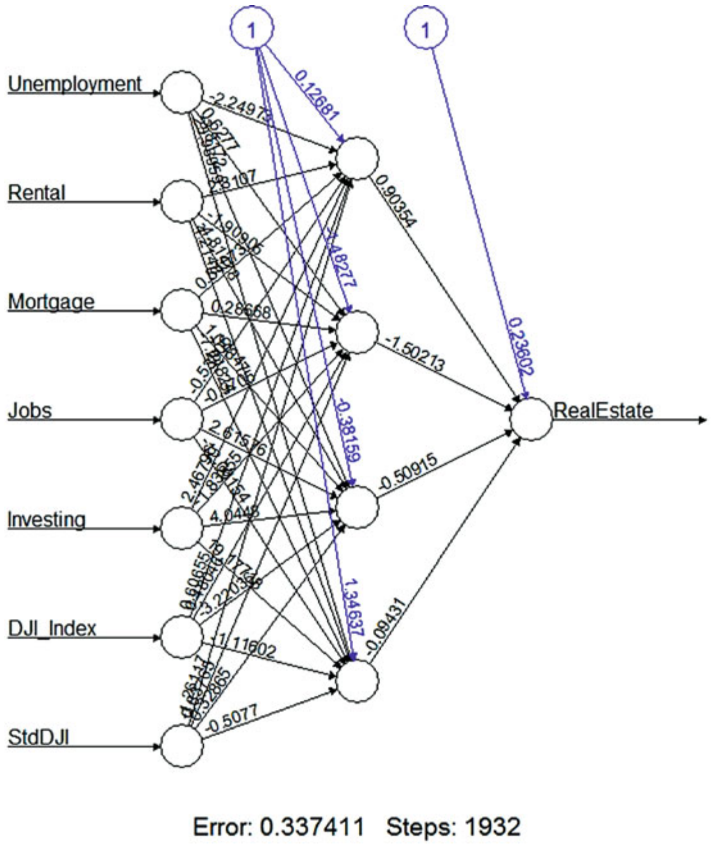

Fig. 11.8 A more elaborate neural network showing decreased prediction error, compare to Fig. 11.7

```
google_pred2<-compute(google_model2, google_test[, c(1:2, 4:8)])
pred_results2<-google_pred2$net.result
cor(pred_results2, google_test$RealEstate)

##              [,1]
## [1,] 0.9869109731
```

We get an even higher correlation. This is almost an ideal result! The predicted and observed indices have a very strong linear relationship. Nevertheless, too many hidden nodes might even decrease the correlation between predicted and true values, which will be examined in the practice problems later in this Chapter.

11.2.6 Step 6: Adding Additional Layers

We observe an even lower `Error` by using three hidden layers with numbers of nodes 4,3,3 within each, respectively.

```
google_model2<-neuralnet(RealEstate~Unemployment+Rental+Mortgage+Jobs+Invest
ing+DJI_Index+StdDJI, data=google_train, hidden = c(4,3,3))
google_pred2<-compute(google_model2, google_test[, c(1:2, 4:8)])
pred_results2<-google_pred2$net.result
cor(pred_results2, google_test$RealEstate)

##              [,1]
## [1,] 0.9853727545
```

11.3 Simple NN Demo: Learning to Compute √

This simple example demonstrates the foundation of the neural network prediction of a basic mathematical function (square-root): $\sqrt{\ } : \Re^+ \to \Re^+$ (Fig. 11.9).

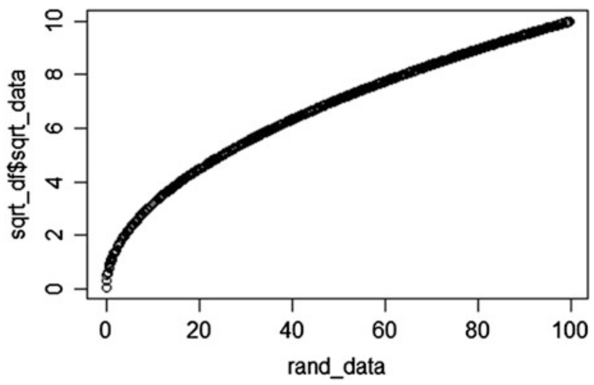

Fig. 11.9 The square-root function evaluated at random Uniform(0,100) values

```
# generate random training data: 1,000 |X_i|, where X_i ~ Uniform (0,10) or
perhaps ~ N(0,1)
rand_data <- abs(runif(1000, 0, 100))

# create a 2 column data-frame (and_data, sqrt_data)
sqrt_df <- data.frame(rand_data, sqrt_data=sqrt(rand_data))
plot(rand_data, sqrt_df$sqrt_data)
```

```
# Train the neural net
net.sqrt <- neuralnet(sqrt_data ~ rand_data,  sqrt_df, hidden=10,
threshold=0.01)

 # report the NN
 # print(net.sqrt)

 # generate testing data seq(from=0.1, to=N, step=0.1)
 N <- 200.0   # out of range [100: 200] is also included in the testing!
 test_data <- seq(0, N, 0.1); test_data_sqrt <- sqrt(test_data)

 # try to predict the square-root values using 10 hidden nodes
 # Compute or predict for test data, test_data
 pred_sqrt <- compute(net.sqrt, test_data)$net.result
```

```
 # compute uses the trained neural net (net.sqrt),
 # to estimate the square-roots of the testing data

 # compare real (test_data_sqrt) and NN-predicted (pred_sqrt) square
roots of test_data
 plot(pred_sqrt, test_data_sqrt, xlim=c(0, 12), ylim=c(0, 12));

 abline(0,1, col="red", lty=2)
 legend("bottomright",  c("Pred vs. Actual SQRT", "Pred=Actual Line"),
cex=0.8, lty=c(1,2), lwd=c(2,2),col=c("black","red"))
```

```
 compare_df <-data.frame(pred_sqrt, test_data_sqrt); # compare_df

 plot(test_data, test_data_sqrt)
 lines(test_data, pred_sqrt, pch=22, col="red", lty=2)
 legend("bottomright",  c("Actual SQRT","Predicted SQRT"), lty=c(1,2), lwd=c
(2,2),col=c("black","red"))
```

We observe that the NN, `net.sqrt` actually learns and predicts the complex square root function with good accuracy, Figs 11.10 and 11.11. Of course, individual results may vary, as we randomly generate the training data (rand_data) and due to the stochastic construction of the ANN.

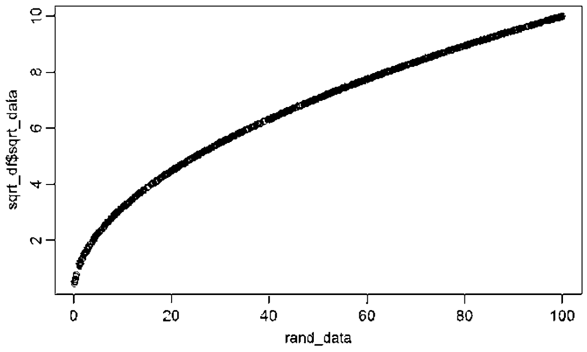

Fig. 11.10 Test data validation of the neural network predicting the behavior of the square-root function

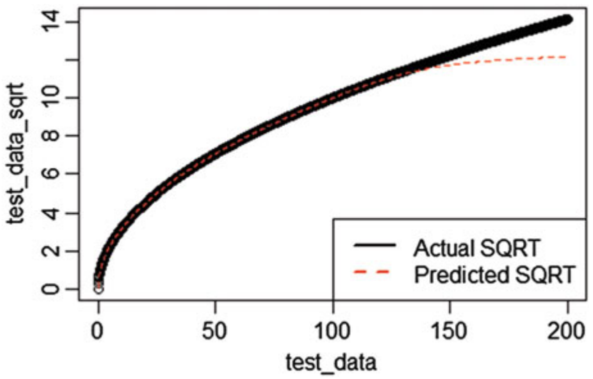

Fig. 11.11 Plots illustrating the agreement of the NN-predicted and the analytical square-root function

11.4 Case Study 2: Google Trends and the Stock Market – Classification

In practice, ANN models are also useful as classifiers. Let's demonstrate this by using again the Stock Market data. We will binarize the samples according to their `RealEstate` values. For those higher than the 75%, we will lable them 0; For those lower than the 25%, we will label them 2; all others will be labeled 1. Even in the classification setting, the response still must be numeric.

```
google_class = google_norm
id1 = which(google_class$RealEstate>quantile(google_class$RealEstate,0.75))
id2 = which(google_class$RealEstate<quantile(google_class$RealEstate,0.25))
id3 = setdiff(1:nrow(google_class),union(id1,id2))
google_class$RealEstate[id1]=0
google_class$RealEstate[id2]=1
google_class$RealEstate[id3]=2
summary(as.factor(google_class$RealEstate))

##   0   1   2
## 179 178 374
```

Here, we divide the data to training and testing sets. We need three more column indicators that correspond to the three derived RealEstate labels.

```
set.seed(2017)
train = sample(1:nrow(google_class),0.7*nrow(google_class))
google_tr = google_class[train,]
google_ts = google_class[-train,]
train_x = google_tr[,c(1:2,4:8)]
train_y = google_tr[,3]
colnames(train_x)

## [1] "Unemployment" "Rental"       "Mortgage"     "Jobs"
## [5] "Investing"    "DJI_Index"    "StdDJI"

test_x = google_ts[,c(1:2,4:8)]
test_y = google_ts[3]
train_y_ind = model.matrix(~factor(train_y)-1)
colnames(train_y_ind) = c("High","Median","Low")
train = cbind(train_x, train_y_ind)
```

We use non-linear output and display every 2,000 iterations.

```
nn_single = neuralnet(High+Median+Low~Unemployment+Rental+Mortgage+Jobs+Inve
sting+DJI_Index+StdDJI,
    data = train,
    hidden=4,
    linear.output=FALSE,
    lifesign='full', lifesign.step=2000)
## hidden: 4    thresh:0.01    rep:1/1    steps:2000  min thresh: 0.13702015
48
##    4000  min thresh: 0.08524054094
##    6000  min thresh: 0.08524054094
##    8000  min thresh: 0.08524054094
##   10000  min thresh: 0.08524054094
…

##   40000  min thresh: 0.02427719823
##   42000  min thresh: 0.02158221449
##   44000  min thresh: 0.01831644589
##   46000  min thresh: 0.01682874388
##   48000  min thresh: 0.01572773551
##   50000  min thresh: 0.01311388938
##   52000  min thresh: 0.01241004281
##   54000  min thresh: 0.01131407008
##   55420  error: 7.01191  time: 19.33 secs
```

Below is the prediction function translating this model to generate forecasting results.

```
pred = function(nn, dat) {
   # compute uses the trained neural net (nn=nn_single), and
   # new testing data (dat=google_ts) to generate predictions (y_hat)
   # compute returns a list containing:
   # (1) neurons: a list of the neurons' output for each layer of the
neural network, and
    # (2) net.result: a matrix containing the overall result of the
neural network.
   yhat = compute(nn, dat)$net.result
   # find the maximum in each row (1) in the net.result matrix
   # to determine the first occurrence of a specific element in each row (1)
   # we can use the apply function with which.max
   yhat = apply(yhat, 1, which.max)-1
   return(yhat)
}
mean(pred(nn_single, google_ts[,c(1:2,4:8)]) != as.factor(google_ts[,3]))

## [1] 0.03181818182
```

Now let’s inspect the structure of the Neural Network.

```
plot(nn_single)
```

Similarly, we can change `hidden` to utilize multiple hidden layers. However, a more complicated model won’t necessarily guarantee an improved performance.

```
nn_single = neuralnet(High+Median+Low~Unemployment+Rental+Mortgage+Jobs+Inve
sting+DJI_Index+StdDJI,
    data = train,
    hidden=c(4,5),
    linear.output=FALSE,
    lifesign='full', lifesign.step=2000)

## hidden: 4, 5    thresh: 0.01    rep:1/1    steps:2000   min thresh: 0.307
##       4000   min thresh: 0.2875517033
##       6000   min thresh: 0.1383720887
##       8000   min thresh: 0.1115440575
##      10000   min thresh: 0.09233958192
##      12000   min thresh: 0.0766173347
##      14000   min thresh: 0.05763223509
##      16000   min thresh: 0.03417989426
##      18000   min thresh: 0.01473872843
##      20000   min thresh: 0.01101646653
##      20741   error: 7.00627  time: 11.3 secs

mean(pred(nn_single, google_ts[,c(1:2,4:8)]) != as.factor(google_ts[,3]))

## [1] 0.03181818182
```

11.5 Support Vector Machines (SVM)

Recall that the Lazy learning methods in Chap. 6 assigned class labels using geometrical distances of different features. In multidimensional spaces (multiple features), we can use spheres with centers determined by the training dataset. Then, we can assign labels to testing data according to their nearest spherical center. Let’s see if we make a choose other hypersurfaces that may separate *nD* data and indice a classification scheme.

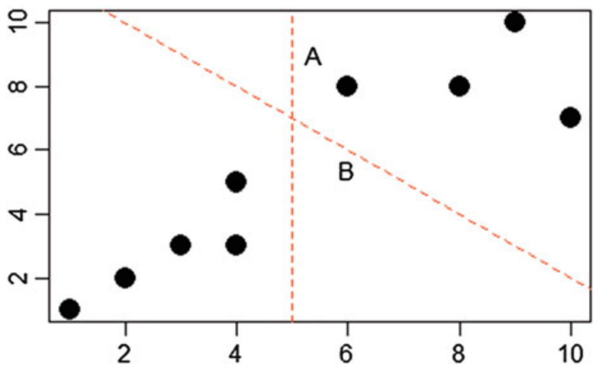

Fig. 11.12 Schematic representation of linear-kernel SVM classification

11.5.1 *Classification with Hyperplanes*

The easiest shape would be a plane. Support Vector Machine (SVM) can use hyperplanes to separate data into several groups or classes. This is used for datasets that are linearly separable. Assume that we have only two features, will you use the A or B hyperplane to separate the data on Fig. 11.12? Perhaps even another hyperplane, C?

Finding the Maximum Margin

To answer the above question, we need to search for the **Maximum Margin Hyperplane (MMH)**. That is the hyperplane that creates the greatest separation between the two closest observations.

We define support vectors as the points from each class that are closest to the MMH. Each class must have at least one observation as a support vector.

Using support vectors alone is not sufficient for finding the MMH. Although tricky mathematical calculations are involved, the fundamental process is fairly simple. Let's look at linearly separable data and non-linearly separable data individually.

Linearly Separable Data

If the dataset is linearly separable, we can find the outer boundaries of our two groups of data points. These boundaries are called convex hull (red lines in the following graph). The MMH (black solid line) is the line that is perpendicular to the shortest line between the two convex hulls (Fig. 11.13).

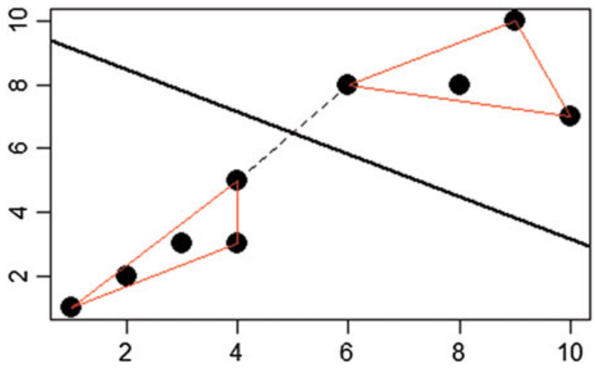

Fig. 11.13 Convex hulls of the linearly separable groups of points

An alternative way would be picking two parallel planes that can separate the data into two groups while the distance between two planes is as far as possible.

To mathematically define a plane, we need to use vectors. In n-dimensional spaces planes could be expressed by the following equation:

$$\vec{w} \cdot \vec{x} + b = 0,$$

where $\vec{w}$ (weights) and $\vec{x}$ both have n coordinates, and b is a single number known as the *bias*.

To clarify this notation, let's look at the situation in a 3D space where we can express (embed) 2D Euclidean planes given a point $((x_o, y_o, z_o))$ and a normal-vector $((a, b, c))$ form. This is just a linear equation, where $d = -(ax_o + by_o + cz_0)$:

$$ax + by + cz + d = 0,$$

or equivalently

$$w_1x_1 + w_2x_2 + w_3x_3 + b = 0.$$

We can see that it is equivalent to the vector notation.

Using the vector notation, we can specify two hyperplanes as follows:

$$\vec{w} \cdot \vec{x} + b \geq +1$$

and

$$\vec{w} \cdot \vec{x} + \mathrm{b} \leq -1.$$

We require that all class 1 observations fall above the first plane and all observations in the other class fall below the second plane.

The distance between two planes is calculated as:

$$\frac{2}{\|\vec{w}\|},$$

where $\|\vec{w}\|$ is the Euclidean norm. To maximize the distance, we need to minimize the Euclidean norm.

To sum up we are going to find $min\frac{\|\vec{w}\|}{2}$ with the following constrain:

$$y_i\left(\vec{w}\cdot\vec{x}-b\right)\geq 1, \forall\vec{x}_i,$$

where $\forall$ means "for all".

For each nonlinear programming problem, called the **primal problem**, there is a related nonlinear programming problem, called the **Lagrangian dual problem**. Under certain assumptions for convexity and suitable constraints, the primal and dual problems have equal optimal objective values. Primal optimization problems are typically described as:

$$\underset{x}{min} f(x)$$

subject to

$$\left|\begin{array}{l} g_i(x)\leq 0 \\ h_j(x)=0 \end{array}\right..$$

The Lagrangian dual problem is defined as a parallel nonlinear programming problem:

$$\min_{u,v}\ (\theta(u,v))$$
$$\text{subject to } u\geq 0,$$

where

$$\theta(u,v)=\inf_x\left(f(x)+\sum_i u_i g_i(x)+\sum_j v_j h_j(x)\right).$$

Chapter 21 provides additional technical details about optimization duality.

Suppose the Lagrange primal is:

$$L_p=\frac{1}{2}\|w\|^2-\sum_{i=1}^n\alpha_i\left[y_i\left(w_0+x_i^T w\right)-1\right], where\ \alpha_i\geq 0.$$

To optimize that objective function, we can set the partial derivatives equal to zero:

$$\frac{\partial}{\partial w}: w=\sum_{i=1}^n\alpha_i y_i x_i$$

$$\frac{\partial}{\partial b}: 0=\sum_{i=1}^n\alpha_i y_i.$$

Substituting into the Lagrange primal, we obtain the Lagrange dual:

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \alpha_i \alpha_i' y_i y_i' x_i^T x_i'.$$

We maximize $\mathrm{L_D}$ subject to $\alpha_i \geq 0$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$.

The Karush-Kuhn-Tucker optimization conditions suggest that we have $\hat{\alpha}\left[y_i\left(\hat{b} + x_i^T \hat{w}\right) - 1\right] = 0$.

Which implies that if $y_i \hat{f}(x_i) > 1$, then $\hat{\alpha}_i = 0$. The **support** of a function (f) is the smallest subset of the domain containing only arguments (x) which are not mapped to zero $(f(x) \neq 0)$. In our case, the solution $\hat{w}$ is defined in terms of a linear combination of the **support points**:

$$\hat{f}(x) = w^T x = \sum_{i=1}^{n} \alpha_i y_i x_i.$$

That's where the name of Support Vector Machines (SVM) comes from.

Non-linearly Separable Data

For non-linearly separable data, we need to use a small trick. We still use a plane, but allow some of the points to be misclassified into the wrong class. To penalize for that, we add a cost term after the Euclidean norm function that we need to minimize.

Therefore, the solution changes to:

$$\min\left(\frac{\|\vec{w}\|}{2}\right) + C\sum_{i=1}^{n} \xi_i$$
$$s.t. y_i\left(\vec{w} \cdot \vec{x} - b\right) \geq 1, \forall \vec{x}_i, \xi_i \geq 0,$$

where C is the cost and ξ_i is the distance between the misclassified observation i and the plane.

We have Lagrange dual problem:

$$L_p = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i \left[y_i\left(b + x_i^T w\right) - (1 - \xi_i)\right] - \sum_{i=1}^{n} \gamma_i \xi_i,$$

where

$$\alpha_i, \gamma_i \geq 0.$$

Similar to what we did above for the separable case, we can use the derivatives of the primal problem to solve the dual problem.

Notice the inner product in the final expression. We can replace this inner product with a kernel function that maps the feature space into a higher dimensional space (e.g., using a polynomial kernel) or an infinite dimensional space (e.g., using a Gaussian kernel).

Using Kernels for Non-linear Spaces

An alternative way to solve for the non-linear separable is called the kernel trick. That is to add some dimensions (or features) to make these non-linear separable data to be separable in a higher dimensional space.

How can we do that? We transform our data using kernel functions. A general form for kernel functions would be:

$$K\left(\vec{x}_i, \vec{x}_j\right) = \phi\left(\vec{x}_i\right) \cdot \phi\left(\vec{x}_j\right)$$

The kernel is a mapping of the data into another space.

The linear kernel would be the simplest one that is just the dot product of the features.

$$K\left(\vec{x}_i, \vec{x}_j\right) = \vec{x}_i \cdot \vec{x}_j.$$

The polynomial kernel of degree *d* transforms the data by adding a simple non-linear transformation of the data.

$$K\left(\vec{x}_i, \vec{x}_j\right) = \left(\vec{x}_i \cdot \vec{x}_j + 1\right)^d.$$

The sigmoid kernel is very similar to neural network. It uses a sigmoid activation function.

$$K\left(\vec{x}_i, \vec{x}_j\right) = tanh\left(k\vec{x}_i \cdot \vec{x}_j - \delta\right).$$

The Gaussian RBF kernel is similar to RBF neural network and is a good place to start investigating a dataset.

$$K\left(\vec{x}_i, \vec{x}_j\right) = exp\left(\frac{- \parallel \vec{x}_i - \vec{x}_j \|^2)}{2\sigma^2}\right).$$

11.6 Case Study 3: Optical Character Recognition (OCR)

This example illustrates interpreting, processing and recognizing handwritten notes (text). Specifically, we will convert handwritten characters (unstructured image data) to printed text (typeset characters).

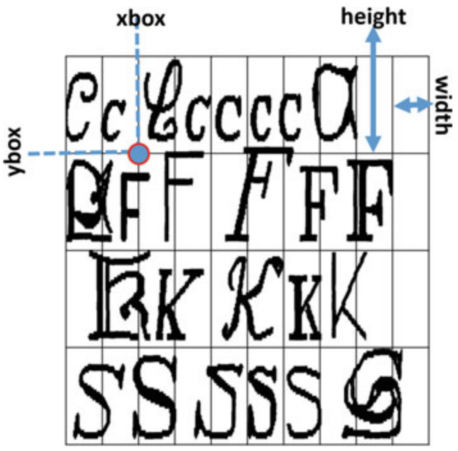

Fig. 11.14 Example of the preprocessed gridded handwritten letters

Protocol:

- Divide the image (typically optical image of handwritten notes on paper) into a fine grid where each cell contains one glyph (symbol, letter, number).
- Match the glyph in each cell to one of the possible characters in a dictionary.
- Combine individual characters together into words to reconstitute the digital representation of the optical image of the handwritten notes.

In this example, we use an optical document image (data) that has already been pre-partitioned into rectangular grid cells containing one character of the 26 English letters, A through Z.

The resulting gridded dataset is distributed by the UCI Machine Learning Data Repository. The dataset contains 20,000 examples of 26 English capital letters printed using 20 different randomly reshaped and morphed fonts (Fig. 11.14).

11.6.1 Step 1: Prepare and Explore the Data

```
# read in data and examine structure
hand_letters <- read.csv("https://umich.instructure.com/files/2837863/downlo
ad?download_frd=1", header = T)
str(hand_letters)

## 'data.frame':    20000 obs. of  17 variables:
##  $ letter: Factor w/ 26 levels "A","B","C","D",..: 20 9 4 14 7 19 2 1 10
13 ...
##  $ xbox  : int  2 5 4 7 2 4 4 1 2 11 ...
##  $ ybox  : int  8 12 11 11 1 11 2 1 2 15 ...
##  $ width : int  3 3 6 6 3 5 5 3 4 13 ...
##  $ height: int  5 7 8 6 1 8 4 2 4 9 ...
##  $ onpix : int  1 2 6 3 1 3 4 1 2 7 ...
##  $ xbar  : int  8 10 10 5 8 8 8 8 10 13 ...
##  $ ybar  : int  13 5 6 9 6 8 7 2 6 2 ...
##  $ x2bar : int  0 5 2 4 6 6 6 2 2 6 ...
##  $ y2bar : int  6 4 6 6 6 9 6 2 6 2 ...
##  $ xybar : int  6 13 10 4 6 5 7 8 12 12 ...
##  $ x2ybar: int  10 3 3 4 5 6 6 2 4 1 ...
##  $ xy2bar: int  8 9 7 10 9 6 6 8 8 9 ...
##  $ xedge : int  0 2 3 6 1 0 2 1 1 8 ...
##  $ xedgey: int  8 8 7 10 7 8 8 6 6 1 ...
##  $ yedge : int  0 4 3 2 5 9 7 2 1 1 ...
##  $ yedgex: int  8 10 9 8 10 7 10 7 7 8 ...

# divide into training (3/4) and testing (1/4) data
hand_letters_train <- hand_letters[1:15000, ]
hand_letters_test  <- hand_letters[15001:20000, ]
```

11.6.2 Step 2: Training an SVM Model

We can specify `vanilladot` as a linear kernel, or alternatively:

- `rbfdot` Radial Basis kernel i.e., “Gaussian”
- `polydot` Polynomial kernel
- `tanhdot` Hyperbolic tangent kernel
- `laplacedot` Laplacian kernel
- `besseldot` Bessel kernel
- `anovadot` ANOVA RBF kernel
- `splinedot` Spline kernel
- `stringdot` String kernel

```
# begin by training a simple linear SVM
library(kernlab)
set.seed(123)
hand_letter_classifier <- ksvm(letter ~ ., data = hand_letters_train,
kernel = "vanilladot")

##  Setting default kernel parameters

# look at basic information about the model
hand_letter_classifier

## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 6618
##
```

```
## Objective Function Value : -13.2947 -19.6051 -20.8982 -5.6651 -7.2092 -31
.5151 -48.3253 -17.6236 -57.0476 -30.532 -15.7162 -31.49 -28.2706 -45.741 -1
1.7891 -33.3161 -28.2251 -16.5347 -13.2693 -30.88 -29.4259 -7.7099 -11.1685
-29.4289 -13.0857 -9.2631 -144.1105 -52.7747 -71.052 -109.7783 -158.3152 -51
.2839 -39.6499 -67.0061 -23.8637 -27.6083 -26.3461 -35.2626 -38.6346 -116.89
67 -173.8336 -214.2196 -20.7925 -10.3812 -53.1156 -12.228 -46.6132 -8.6867 -
18.9108 -11.0535 -94.5751 -26.5689 -224.0215 -70.5714 -8.3232 -4.5265 -132.5
431 -74.6876 -19.5742 -12.7352 -81.7894 -11.6983 -25.4835 -17.582 -23.934 -2
7.022 -50.7092 -10.9228 -4.3852 -13.7216 -3.8547 -3.5723 -8.419 -36.9773 -47
.1418 -172.6874 -42.457 -44.0342 -42.7695 -13.0527 -16.7534 -78.7849 -101.81
46 -32.1141 -30.3349 -104.0695 -32.1258 -24.6301 -32.6087 -17.0808 -5.1347 -
40.5505 -6.684 -16.2962 -56.364 -147.3669 -49.0907 -37.8334 -32.8068 -73.248
-127.7819 -10.5342 -5.2495 -11.9568 -30.1631 -135.5915 -51.521 -176.2669 -99
.0973 -10.295 -14.5906 -3.7822 -64.1452 -7.4813 -84.9109 -40.9146 -87.2437 -
66.8629 -69.9932 -20.5294 -12.7577 -7.0328 -22.9219 -12.3975 -223.9411 -29.9
969 -24.0552 -132.6252 -133.7033 -9.2959 -33.1873 -5.8016 -57.3392 -60.9046
-27.1766 -200.8554 -29.9334 -15.9359 -130.0183 -154.4587 -43.5779 -24.4852 -
135.7896 -74.1531 -303.5043 -131.4741 -149.5403 -30.4917 -29.8086 -47.3454 -
24.6204 -44.2792 -6.2064 -8.6708 -36.4412 -68.712 -179.7303 -44.7489 -84.860
8 -136.6786 -569.3398 -113.0779 -138.435 -303.8556 -32.8011 -60.4546 -139.35
25 -108.9841 -34.277 -64.9071 -38.6148 -7.5086 -204.222 -12.9572 -29.0252 -2
.0352 -5.9916 -14.3706 -21.5773 -57.0064 -19.6546 -178.0543 -19.812 -4.145 -
4.5318 -0.8101 -116.8649 -7.8269 -53.3445 -21.4812 -13.5066 -5.3881 -15.1061
-27.6061 -18.9239 -68.8104 -26.1223 -93.0231 -15.1693 -9.7999 -7.6137 -1.530
1 -84.9531 -5.4551 -93.187 -93.4153 -43.8334 -23.6706 -59.1468 -22.0933 -47.
8381 -219.9936 -39.5596 -47.2643 -34.0752 -20.2532 -11.239 -118.4152 -6.4126
-5.1846 -8.7272 -9.4584 -20.8522 -22.0878 -113.0806 -29.0912 -80.397 -29.620
6 -13.7422 -8.9416 -3.0785 -79.842 -6.1869 -13.9663 -63.3665 -93.2067 -11.55
93 -13.0449 -48.2558 -2.9343 -8.25 -76.4361 -33.5374 -109.112 -4.1731 -6.197
8 -1.2664 -84.1287 -18.3054 -7.2209 -45.5509 -3.3567 -16.8612 -60.5094 -43.9
956 -53.0592 -6.1407 -17.4499 -2.3741 -65.023 -102.1593 -103.4312 -23.1318 -
17.3394 -50.6654 -31.4407 -57.6065 -19.6857 -5.2667 -4.1767 -55.8445 -30.92
-57.2396 -30.1101 -7.611 -47.7711 -12.1616 -19.1572 -53.5364 -3.8024 -53.124
-225.6075 -12.6791 -11.5852 -16.6614 -9.7186 -65.824 -16.3897 -42.3931 -50.5
13 -24.752 -14.513 -40.495 -16.5124 -57.1813 -4.7974 -5.2949 -81.7477 -3.272
-6.3448 -1.1259 -114.3256 -22.3232 -339.8619 -31.0491 -31.3872 -4.9625 -82.4
936 -123.6225 -72.8463 -23.4836 -33.1608 -11.7133 -19.7607 -1.8599 -50.1148
-8.2868 -143.3592 -1.8508 -1.9699 -9.4175 -0.5202 -25.0654 -30.0489 -5.6248
## Training error : 0.129733
```

11.6.3 Step 3: Evaluating Model Performance

Let’s assess the SVM prediction using the testing data.

```
# predictions on testing dataset
hand_letter_predictions<- predict(hand_letter_classifier, hand_letters_test)

head(hand_letter_predictions)

## [1] C U K U E I
## Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

table(hand_letter_predictions, hand_letters_test$letter)

##
```

```
## hand_letter_predictions   A   B   C   D   E   F   G   H   I   J   K   L
##                        A 191   0   1   0   0   0   0   0   0   1   0   0
##                        B   0 157   0   9   2   0   1   3   0   0   1   0
##                        C   0   0 142   0   5   0  14   3   2   0   2   4
##                        D   1   1   0 196   0   1   4  12   5   3   4   4
##                        E   0   0   8   0 164   2   1   1   0   0   3   5
##                        F   0   0   0   0   0 171   4   2   8   2   0   0
##                        G   1   1   4   1  10   3 150   2   0   0   1   2
##                        H   0   3   0   1   0   2   2 122   0   2   4   2
##                        I   0   0   0   0   0   0   0   0 175  10   0   0
##                        J   2   2   0   0   0   3   0   2   7 158   0   0
##                        K   2   1  11   0   0   0   4   6   0   0 148   0
##                        L   0   0   0   0   1   0   1   1   0   0   0 176
##                        M   0   0   1   1   0   0   1   2   0   0   0   0
##                        N   0   0   0   1   0   1   0   1   0   0   0   0
##                        O   0   0   1   2   0   0   2   1   0   2   0   0
##                        P   0   0   0   1   0   3   1   0   0   0   0   0
##                        Q   0   0   0   0   0   0   9   3   0   0   0   3
##                        R   2   5   0   1   1   0   2   9   0   0  11   0
##                        S   1   2   0   0   1   1   5   0   2   2   0   3
##                        T   0   0   0   0   3   6   0   1   0   0   1   0
##                        U   1   0   3   3   0   0   0   2   0   0   0   0
##                        V   0   0   0   0   0   1   6   3   0   0   0   0
##                        W   0   0   0   0   0   0   1   0   0   0   0   0
##                        X   0   1   0   0   2   0   0   1   3   0   2   6
##                        Y   3   0   0   0   0   0   0   1   0   0   0   0
##                        Z   2   0   0   0   2   0   0   0   3   3   0   0
##
## hand_letter_predictions   M   N   O   P   Q   R   S   T   U   V   W   X
##                        A   1   2   2   0   5   0   2   1   1   0   1   0
##                        B   3   0   0   2   4   8   5   0   0   3   0   1
##                        C   0   0   2   0   0   0   0   0   0   0   0   0
##                        D   0   6   5   3   1   4   0   0   0   0   0   5
##                        E   0   0   0   0   6   0  10   0   0   0   0   4
##                        F   0   0   0  18   0   0   5   2   0   0   0   1
##                        G   1   0   0   2  11   2   5   3   0   0   0   1
##                        H   2   5  23   0   2   6   0   4   1   4   0   0
##                        I   0   0   0   1   0   0   3   0   0   0   0   4
##                        J   0   0   1   1   4   0   1   0   0   0   0   2
##                        K   0   2   0   1   1   7   0   1   3   0   0   4
##                        L   0   0   0   0   1   0   4   0   0   0   0   1
##                        M 177   5   1   0   0   0   0   0   4   0   8   0
##                        N   0 172   0   0   0   3   0   0   1   0   2   0
##                        O   0   1 132   2   4   0   0   0   3   0   0   0
##                        P   0   0   3 168   1   0   0   1   0   0   0   0
##                        Q   0   0   5   1 163   0   5   0   0   0   0   0
##                        R   1   1   1   1   0 176   0   1   0   2   0   0
##                        S   0   0   0   0  11   0 135   2   0   0   0   2
##                        T   0   0   0   0   0   0   3 163   1   0   0   0
##                        U   0   1   0   1   0   0   0   0 197   0   1   1
##                        V   0   3   1   0   2   1   0   0   0 152   1   0
##                        W   2   0   4   0   0   0   0   0   4   7 154   0
##                        X   0   0   1   0   0   1   2   0   0   0   0 160
##                        Y   0   0   0   6   0   0   0   3   0   0   0   0
##                        Z   0   0   0   0   1   0  18   3   0   0   0   0
##
```

```
## hand_letter_predictions   Y   Z
##                       A   0   0
##                       B   0   0
##                       C   0   0
##                       D   3   1
##                       E   0   3
##                       F   3   0
##                       G   0   0
##                       H   3   0
##                       I   1   1
##                       J   0  11
##                       K   0   0
##                       L   0   1
##                       M   0   0
##                       N   0   0
##                       O   0   0
##                       P   1   0
##                       Q   3   0
##                       R   0   0
##                       S   0  10
##                       T   5   2
##                       U   1   0
##                       V   5   0
##                       W   0   0
##                       X   1   1
##                       Y 157   0
##                       Z   0 164

# look only at agreement vs. non-agreement
# construct a vector of TRUE/FALSE indicating correct/incorrect predictions
agreement <- hand_letter_predictions == hand_letters_test$letter

# check if characters agree
table(agreement)

## agreement
## FALSE  TRUE
##   780  4220

prop.table(table(agreement))

## agreement
## FALSE  TRUE
## 0.156 0.844
```

11.6.4 Step 4: Improving Model Performance

Replacing the `vanilladot` linear kernel with `rbfdot` Radial Basis Function kernel, i.e., “Gaussian” kernel, may improve the OCR prediction.

```
hand_letter_classifier_rbf <- ksvm(letter ~ ., data = hand_letters_train,
kernel = "rbfdot")
hand_letter_predictions_rbf <- predict(hand_letter_classifier_rbf,
hand_letters_test)

agreement_rbf <- hand_letter_predictions_rbf == hand_letters_test$letter
table(agreement_rbf)
```

```
## agreement_rbf
## FALSE  TRUE
##   360  4640

prop.table(table(agreement_rbf))

## agreement_rbf
## FALSE  TRUE
## 0.072 0.928
```

Note the improvement of the automated (SVM) classification accuracy (0.928) for `rbfdot` compared to the previous (`vanilladot`) result (0.844).

11.7 Case Study 4: Iris Flowers

Let's have another look at the *iris data* that we saw in Chap. 2.

11.7.1 Step 1: Collecting Data

SVM requires all features to be numeric, and each feature has to be scaled into a relative small interval. We are using the Edgar Anderson's Iris Data in R for this case study. This dataset measures the length and width of sepals and petals from three Iris flower species.

11.7.2 Step 2: Exploring and Preparing the Data

Let's load the data first. In this case study we want to explore the variable `Species`.

```
data(iris)
str(iris)

## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
1 1 1 1 ...

table(iris$Species)

##
##     setosa versicolor  virginica
##         50         50         50
```

The data look good but we still can normalize the features either by hand or using an R function.

Next, we can separate the training and testing datasets using 75%–25% rule.

```
sub<-sample(nrow(iris), floor(nrow(iris)*0.75))
iris_train<-iris[sub, ]
iris_test<-iris[-sub, ]
```

We can try the linear and non-linear kernels on the iris data (Figs. 11.15 and 11.16).

```
require(e1071)

iris.svm_1 <- svm(Species~Petal.Length+Petal.Width, data=iris_train,
                  kernel="linear", cost=1)
iris.svm_2 <- svm(Species~Petal.Length+Petal.Width, data=iris_train,
                  kernel="radial", cost=1)
par(mfrow=c(2,1))
plot(iris.svm_1, iris[,c(5,3,4)]); legend("center", "Linear")
```

```
plot(iris.svm_2, iris[,c(5,3,4)]); legend("center", "Radial")
```

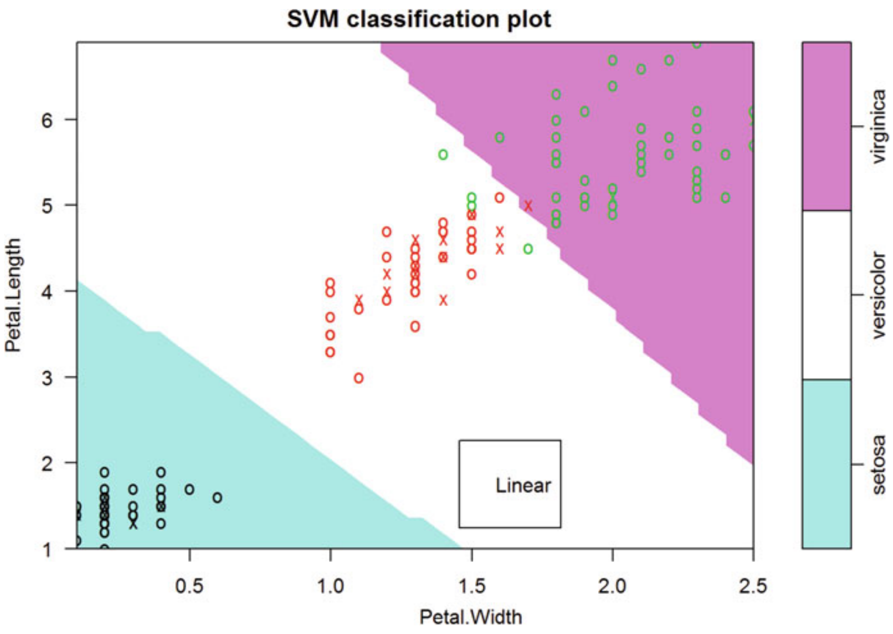

Fig. 11.15 Linear-SVM kernel classification of the iris flowers dataset

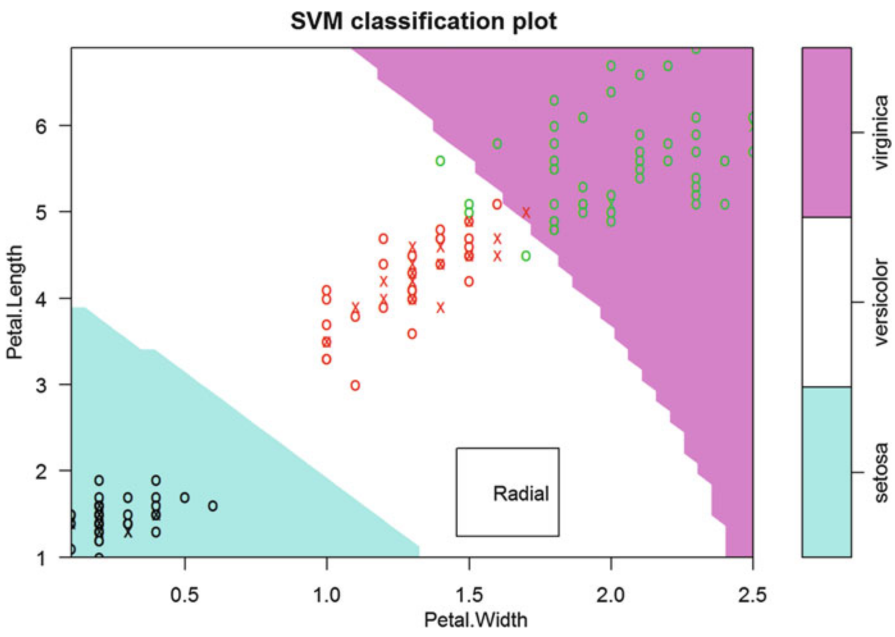

Fig. 11.16 Radial-SVM kernel classification of the iris flowers dataset

11.7.3 *Step 3: Training a Model on the Data*

We are going to use `kernlab` for this case study. However other packages like `e1071` and `klaR` are available if you are quite familiar with C++.

Let's break down the function `ksvm()`

```
m <-ksvm(target~predictors, data = mydata, ker-
nel = "rbfdot", c = 1)
```

- target: the outcome variable that we want to predict.
- predictors: features that the prediction based on. In this function we can use the "." to represent all the variables in the dataset again.
- data: the training dataset that the *target* and *predictors* can be find.
- kernel: is the kernel mapping we want to use. By default it is the radio basis function (`rbfdot`).
- C is a number that specifies the cost of misclassification.

Let’s install the package and play with the data now.

```
# install.packages("kernlab")
library(kernlab)
iris_clas<-ksvm(Species~., data=iris_train, kernel="vanilladot")

##  Setting default kernel parameters

iris_clas

## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 24
##
## Objective Function Value : -1.0066 -0.3309 -13.8658
## Training error : 0.026786
```

Here, we used all the variables other than the `Species` in the dataset as predictors. We also used kernel `vanilladot`, which is the linear kernel in this model. We get a training error that is less than 0.02.

11.7.4 *Step 4: Evaluating Model Performance*

Again, the `predict()` function is used to forecast the species for a test data. Here, we have a factor outcome, so we need the command `table()` to show us how well do the predictions and actual data match.

```
iris.pred<-predict(iris_clas, iris_test)
table(iris.pred, iris_test$Species)

##
## iris.pred    setosa versicolor virginica
##   setosa         13          0         0
##   versicolor      0         14         0
##   virginica       0          1        10
```

We can see a single case of Iris virginica *misclassified* as Iris versicolor. The taxa of all other flowers are correctly predicted.

To see the results more clearly, we can use the proportional table to show the agreements of the categories.

```
agreement<-iris.pred==iris_test$Species
prop.table(table(agreement))

## agreement
##        FALSE         TRUE
## 0.02631578947 0.97368421053
```

Here $==$ means “equal to”. Over 97% of predictions are correct. Nevertheless, is there any chance that we can improve the outcome? What if we try a Gaussian kernel?

11.7.5 Step 5: RBF Kernel Function

Linear kernel is the simplest one but usually not the best one. Let’s try the **RBF (Radial Basis “Gaussian” Function)** kernel instead.

```
iris_clas1<-ksvm(Species~., data=iris_train, kernel="rbfdot")
iris_clas1

## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.877982617394805
##
## Number of Support Vectors : 52
##
## Objective Function Value : -4.6939 -5.1534 -16.2297
## Training error : 0.017857

iris.pred1<-predict(iris_clas1, iris_test)
table(iris.pred1, iris_test$Species)

##
## iris.pred1   setosa versicolor virginica
##   setosa         13          0         0
##   versicolor      0         14         2
##   virginica       0          1         8

agreement<-iris.pred1==iris_test$Species
prop.table(table(agreement))

## agreement
##         FALSE          TRUE
## 0.07894736842 0.92105263158
```

Unfortunately, the model performance is actually worse than the previous one (you might get slightly different results). This is because this Iris dataset has a linear feature. In practice, we could try some alternative kernel functions and see which one fits the dataset the best.

11.7.6 Parameter Tuning

We can tune the SVM using the `tune.svm` function in the package `e1071` (Fig. 11.17).

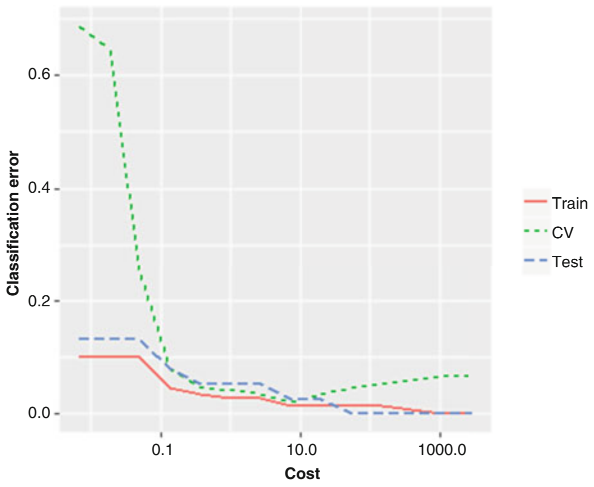

Fig. 11.17 Training data, cross-validation, and testing data SVM classification errors of the iris flowers

```
costs = exp(-5:8)
tune.svm(Species~., kernel = "radial", data = iris_train, cost = costs)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##   cost
##      1
##
## - best performance: 0.03636363636
```

Further, we can draw a **Cross-Validation** (**CV**) plot to gauge the model performance, see cross-validation details in Chap. 21:

```
set.seed(2017)
require(sparsediscrim); require (reshape); require(ggplot2)

folds = cv_partition(iris$Species, num_folds = 5)
train_cv_error_svm = function(costC) {
  #Train
  ir.svm = svm(Species~., data=iris,
                 kernel="radial", cost=costC)
  train_error = sum(ir.svm$fitted != iris$Species) / nrow(iris)
  #Test
  test_error = sum(predict(ir.svm, iris_test) != iris_test$Species) / nrow(i
ris_test)
  #CV error
  ire.cverr = sapply(folds, function(fold) {
    svmcv = svm(Species~.,data = iris, kernel="radial", cost=costC, subset =
fold$training)
    svmpred = predict(svmcv, iris[fold$test,])
    return(sum(svmpred != iris$Species[fold$test]) / length(fold$test))
  })
  cv_error = mean(ire.cverr)
  return(c(train_error, cv_error, test_error))
}

costs = exp(-5:8)
ir_cost_errors = sapply(costs, function(cost) train_cv_error_svm(cost))
df_errs = data.frame(t(ir_cost_errors), costs)
colnames(df_errs) = c('Train', 'CV', 'Test', 'Logcost')
dataL <- melt(df_errs, id="Logcost")
ggplot(dataL, aes_string(x="Logcost", y="value", colour="variable",
       group="variable", linetype="variable", shape="variable")) +
  geom_line(size=1) + labs(x = "Cost",
       y = "Classification error", colour="",group="",
       linetype="",shape="") + scale_x_log10()
```

11.7.7 Improving the Performance of Gaussian Kernels

Now, let's attempt to improve the performance of a Gaussian kernel by tuning:

```
set.seed(2020)
gammas = exp(-5:5)
tune_g = tune.svm(Species~., kernel = "radial", data = iris_train, cost = co
sts,gamma = gammas)
tune_g

## Parameter tuning of 'svm':
## - sampling method: 10-fold cross validation
## - best parameters:
##         gamma        cost
##  0.01831563889 20.08553692
## - best performance: 0.025
```

We observe that the model achieves a better prediction now.

```
iris.svm_g <- svm(Species~., data=iris_train,
                kernel="radial", gamma=0.0183, cost=20)
table(iris_test$Species, predict(iris.svm_g, iris_test))

##
##              setosa versicolor virginica
##   setosa         13          0         0
##   versicolor      0         14         1
##   virginica       0          0        10

agreement<-predict(iris.svm_g, iris_test)==iris_test$Species
prop.table(table(agreement))

## agreement
##         FALSE          TRUE
## 0.02631578947 0.97368421053
```

Chapter 23 provides more details about prediction and classification using neural networks and deep learning.

11.8 Practice

11.8.1 Problem 1 Google Trends and the Stock Market

Use the Google trend data. Fit a neural network model with the same training data as case study 1. This time, use `Investing` as target and `Unemployment, Rental, RealEstate, Mortgage, Jobs, DJI_Index, StdDJI` as predictors. Use three hidden nodes. **Note**: remember to change the columns you want to include in the test dataset when predicting.

The following number is the correlation between predicted and observed values.

```
##                [,1]
## [1,] 0.8845711444
```

You might get a slightly different results since the weights are generated randomly.

11.8.2 Problem 2: Quality of Life and Chronic Disease

Use the same data in Chap. 8 – Quality of life and chronic disease (dataset and metadata doc).

Let's load the data first. In this case study, we want to use the variable `CHARLSONSCORE` as our target variable.

```
qol<-read.csv("https://umich.instructure.com/files/481332/download?download_
frd=1")
str(qol)

## 'data.frame':    2356 obs. of  41 variables:
##  $ ID                 : int  171 171 172 179 180 180 181 182 183 186 ...
##  $ INTERVIEWDATE      : int  0 427 0 0 0 42 0 0 0 0 ...
##  $ LANGUAGE           : int  1 1 1 1 1 1 1 1 1 2 ...
##  $ AGE                : int  49 49 62 44 64 64 52 48 49 78 ...
##  $ RACE_ETHNICITY     : int  3 3 3 7 3 3 3 3 3 4 ...
##  $ SEX                : int  2 2 2 2 1 1 2 1 1 1 ...
##  $ QOL_Q_01           : int  4 4 3 6 3 3 4 2 3 5 ...
##  $ QOL_Q_02           : int  4 3 3 6 2 5 4 1 4 6 ...
##  $ QOL_Q_03           : int  4 4 4 6 3 6 4 3 4 4 ...
##  $ QOL_Q_04           : int  4 4 2 6 3 6 2 2 5 2 ...
##  $ QOL_Q_05           : int  1 5 4 6 2 6 4 3 4 3 ...
##  $ QOL_Q_06           : int  4 4 2 6 1 2 4 1 2 4 ...
##  $ QOL_Q_07           : int  1 2 5 -1 0 5 8 4 3 7 ...
##  $ QOL_Q_08           : int  6 1 3 6 6 6 3 1 2 4 ...
##  $ QOL_Q_09           : int  3 4 3 6 2 2 4 2 2 4 ...
##  $ QOL_Q_10           : int  3 1 3 6 3 6 3 2 4 3 ...
##  $ MSA_Q_01           : int  1 3 2 6 2 3 4 1 1 2 ...
##  $ MSA_Q_02          : int  1 1 2 6 1 6 4 3 2 4 ...
##  $ MSA_Q_03          : int  2 1 2 6 1 2 3 3 1 2 ...
##  $ MSA_Q_04          : int  1 3 2 6 1 2 1 4 1 5 ...
##  $ MSA_Q_05          : int  1 1 1 6 1 2 1 6 3 2 ...
##  $ MSA_Q_06          : int  1 2 2 6 1 2 1 1 2 2 ...
##  $ MSA_Q_07          : int  2 1 3 6 1 1 1 1 1 5 ...
##  $ MSA_Q_08          : int  1 1 1 6 1 1 1 1 2 1 ...
##  $ MSA_Q_09          : int  1 1 1 6 2 2 4 6 2 1 ...
##  $ MSA_Q_10          : int  1 1 1 6 1 1 1 1 1 3 ...
##  $ MSA_Q_11          : int  2 3 2 6 1 1 2 1 1 5 ...
##  $ MSA_Q_12          : int  1 1 2 6 1 1 2 6 1 3 ...
##  $ MSA_Q_13          : int  1 1 1 6 1 6 2 1 4 2 ...
##  $ MSA_Q_14          : int  1 1 1 6 1 2 1 1 3 1 ...
##  $ MSA_Q_15          : int  2 1 1 6 1 1 3 2 1 3 ...
##  $ MSA_Q_16          : int  2 3 5 6 1 2 1 2 1 2 ...
##  $ MSA_Q_17          : int  2 1 1 6 1 1 1 1 1 3 ...
##  $ PH2_Q_01          : int  3 2 1 5 1 1 3 1 2 3 ...
##  $ PH2_Q_02          : int  4 4 1 5 1 2 1 1 4 2 ...
##  $ TOS_Q_01          : int  2 2 2 4 1 1 2 2 1 1 ...
##  $ TOS_Q_02          : int  1 1 1 4 4 4 1 2 4 4 ...
##  $ TOS_Q_03          : int  4 4 4 4 4 4 4 4 4 4 ...
##  $ TOS_Q_04          : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ CHARLSONSCORE     : int  2 2 3 1 0 0 2 8 0 1 ...
##  $ CHRONICDISEASESCORE: num  1.6 1.6 1.54 2.97 1.28 1.28 1.31 1.67 2.21 2
.51 ...
```

Remove the first two columns (we don't need ID variables) and rows that have missing `CHARLSONSCORE` values (e.g., `CHARLSONSCORE` equal to "-9"). Note that, ! `qol$CHARLSONSCORE==` -9, implies that we only select the rows that have CHARLSONSCORE not equal to -9. The exclamation sign indicates "exclude". Also, we need to convert our categorical variable `CHARLSONSCORE` into a factor.

```
qol<-qol[!qol$CHARLSONSCORE==-9 , -c(1, 2)]
qol$CHARLSONSCORE<-as.factor(qol$CHARLSONSCORE)
str(qol)
```

```
## 'data.frame':    2328 obs. of  39 variables:
##  $ LANGUAGE          : int  1 1 1 1 1 1 1 1 1 2 ...
##  $ AGE               : int  49 49 62 44 64 64 52 48 49 78 ...
##  $ RACE_ETHNICITY    : int  3 3 3 7 3 3 3 3 3 4 ...
##  $ SEX               : int  2 2 2 2 1 1 2 1 1 1 ...
##  $ QOL_Q_01          : int  4 4 3 6 3 3 4 2 3 5 ...
##  $ QOL_Q_02          : int  4 3 3 6 2 5 4 1 4 6 ...
##  $ QOL_Q_03          : int  4 4 4 6 3 6 4 3 4 4 ...
##  $ QOL_Q_04          : int  4 4 2 6 3 6 2 2 5 2 ...
##  $ QOL_Q_05          : int  1 5 4 6 2 6 4 3 4 3 ...
##  $ QOL_Q_06          : int  4 4 2 6 1 2 4 1 2 4 ...
##  $ QOL_Q_07          : int  1 2 5 -1 0 5 8 4 3 7 ...
##  $ QOL_Q_08          : int  6 1 3 6 6 6 3 1 2 4 ...
##  $ QOL_Q_09          : int  3 4 3 6 2 2 4 2 2 4 ...
##  $ QOL_Q_10          : int  3 1 3 6 3 6 3 2 4 3 ...
##  $ MSA_Q_01          : int  1 3 2 6 2 3 4 1 1 2 ...
##  $ MSA_Q_02          : int  1 1 2 6 1 6 4 3 2 4 ...
##  $ MSA_Q_03          : int  2 1 2 6 1 2 3 3 1 2 ...
##  $ MSA_Q_04          : int  1 3 2 6 1 2 1 4 1 5 ...
##  $ MSA_Q_05          : int  1 1 1 6 1 2 1 6 3 2 ...
##  $ MSA_Q_06          : int  1 2 2 6 1 2 1 1 2 2 ...
##  $ MSA_Q_07          : int  2 1 3 6 1 1 1 1 1 5 ...
##  $ MSA_Q_08          : int  1 1 1 6 1 1 1 1 2 1 ...
##  $ MSA_Q_09          : int  1 1 1 6 2 2 4 6 2 1 ...
##  $ MSA_Q_10          : int  1 1 1 6 1 1 1 1 1 3 ...
##  $ MSA_Q_11          : int  2 3 2 6 1 1 2 1 1 5 ...
##  $ MSA_Q_12          : int  1 1 2 6 1 1 2 6 1 3 ...
##  $ MSA_Q_13          : int  1 1 1 6 1 6 2 1 4 2 ...
##  $ MSA_Q_14          : int  1 1 1 6 1 2 1 1 3 1 ...
##  $ MSA_Q_15          : int  2 1 1 6 1 1 3 2 1 3 ...
##  $ MSA_Q_16          : int  2 3 5 6 1 2 1 2 1 2 ...
##  $ MSA_Q_17          : int  2 1 1 6 1 1 1 1 1 3 ...
##  $ PH2_Q_01          : int  3 2 1 5 1 1 3 1 2 3 ...
##  $ PH2_Q_02          : int  4 4 1 5 1 2 1 1 4 2 ...
##  $ TOS_Q_01          : int  2 2 2 4 1 1 2 2 1 1 ...
##  $ TOS_Q_02          : int  1 1 1 4 4 4 1 2 4 4 ...
##  $ TOS_Q_03          : int  4 4 4 4 4 4 4 4 4 4 ...
##  $ TOS_Q_04          : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ CHARLSONSCORE      : Factor w/ 11 levels "0","1","2","3",..: 3 3 4 2 1
1 3 9 1 2 ...
##  $ CHRONICDISEASESCORE: num 1.6 1.6 1.54 2.97 1.28 1.28 1.31 1.67 2.21
2.51 ...
```

The dataset is now ready for processing. First, separate the dataset into training and test datasets using the 75%–25% rule. Then, build a SVM model using all other variables in the dataset to be predictor variables. Try to add different cost of misclassification to the model. Rather than the default `C = 1`, we will explore the behavior of the model for `C = 2` and `C = 3` utilizing the radial basis kernel.

The output for `C = 2` is included below.

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 2
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.0174510649312293
##
## Number of Support Vectors : 1703
##
## Objective Function Value : -1798.9778 -666.9432 -352.2265 -46.2968 -15.92
36 -9.2176 -7.1853 -27.9366 -16.3096 -3.5681 -697.4275 -362.6579 -47.0801 -1
6.3701 -9.6556 -6.9882 -28.2074 -16.4556 -3.5121 -321.0676 -44.7405 -15.8416
-9.1439 -6.8161 -26.7174 -15.4833 -3.3944 -43.1026 -15.2923 -7.994 -6.58 -24
.8459 -14.6379 -3.4484 -13.9377 -5.2876 -5.6728 -15.2542 -9.8408 -3.255 -4.6
982 -4.8924 -9.2482 -6.5144 -2.9608 -2.7409 -6.2056 -6.0476 -2.0833 -6.1775
-4.919 -2.7715 -10.5691 -3.0835 -2.566
## Training error : 0.310997

##
## qol.pred2   0   1   2   3   4   5   6   7   8   9  10
##         0 126  76  24   5   1   0   1   0   3   0   1
##         1  88 170  47  19   9   2   1   0   4   3   0
##          2    1    0    0    1    0    0    0    0    0    0    0
##          3    0    0    0    0    0    0    0    0    0    0    0
##          4    0    0    0    0    0    0    0    0    0    0    0
##          5    0    0    0    0    0    0    0    0    0    0    0
##          6    0    0    0    0    0    0    0    0    0    0    0
##          7    0    0    0    0    0    0    0    0    0    0    0
##          8    0    0    0    0    0    0    0    0    0    0    0
##          9    0    0    0    0    0    0    0    0    0    0    0
##          10   0    0    0    0    0    0    0    0    0    0    0

## agreement
##         FALSE         TRUE
## 0.4914089347 0.5085910653
```

The output for C = 3 is included below.

```
## Support Vector Machine object of class "ksvm"
## SV type: C-svc  (classification)
##  parameter : cost C = 3
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.0168577510531693
##
## Number of Support Vectors : 1695
##
## Objective Function Value : -2440.0638 -915.9967 -492.6748 -63.2895 -21.09
29 -11.9108 -10.2404 -39.1843 -21.976 -5.0624 -970.6173 -514.9584 -64.7791 -
22.0947 -12.8987 -9.8114 -39.7908 -22.2957 -4.9403 -431.5178 -59.9296 -20.94
08 -11.7468 -9.4269 -36.602 -20.1783 -4.6829 -56.9469 -19.7357 -9.238 -8.904
7 -32.6121 -18.4667 -4.8007 -17.3102 -5.4133 -6.9733 -17.2097 -10.3016 -4.37
39 -4.7816 -5.7083 -9.7236 -6.6365 -3.723 -2.7726 -6.4151 -6.4453 -2.1222 -8
.03 -5.411 -3.3088 -11.9186 -3.996 -2.8572
## Training error : 0.266896

##
## qol.pred3    0    1    2    3    4    5    6    7    8    9   10
##         0  131   79   24    6    2    0    1    0    4    1    1
##         1   83  165   47   18    8    2    1    0    3    2    0
##         2    1    2    0    1    0    0    0    0    0    0    0
##         3    0    0    0    0    0    0    0    0    0    0    0
##         4    0    0    0    0    0    0    0    0    0    0    0
##         5    0    0    0    0    0    0    0    0    0    0    0
##         6    0    0    0    0    0    0    0    0    0    0    0
##         7    0    0    0    0    0    0    0    0    0    0    0
##         8    0    0    0    0    0    0    0    0    0    0    0
##         9    0    0    0    0    0    0    0    0    0    0    0
##         10   0    0    0    0    0    0    0    0    0    0    0

## agreement
##         FALSE         TRUE
## 0.4914089347 0.5085910653
```

Can you reproduce (approximately) these results?

11.9 Appendix

Below is some additional R code demonstrating the various results reported in this **Chapter**.

```
#Picture 1
x<-runif(1000, -10, 10)
y<-ifelse(x>=0, 1, 0)
plot(x, y, xlab = "Sum of input signals", ylab = "Output signal", main = "Th
reshold")
abline(v=0, lty=2)
#Picture 2
x<-runif(100000, -10, 10)
y<-1/(1+exp(-x))
plot(x, y, xlab = "Sum of input signals", ylab = "Output signal", main = "Si
gmoid")
#Picture 3
x<-runif(100000, -10, 10)
y1<-x
y2<-ifelse(x<=-5, -5, ifelse(x>=5, 5, x))
y3<-(exp(x)-exp(-x))/(exp(x)+exp(-x))
y4<-exp(-x^2/2)
par(mfrow=c(2, 2))
plot(x, y1, main="Linear", xlab="", ylab="")
plot(x, y2, main="Saturated Linear", xlab="", ylab="")
plot(x, y3, main="Hyperbolic tangent", xlab="", ylab="")
plot(x, y4, main = "Gaussian", xlab="", ylab="")
#Picture 4
A<-c(1, 4, 3, 2, 4, 8, 6, 10, 9)
B<-c(1, 5, 3, 2, 3, 8, 8, 7, 10)
plot(A, B, xlab="", ylab="", pch=16, cex=2)
abline(v=5, col="red", lty=2)
text(5.4, 9, labels="A")
abline(12, -1, col="red", lty=2)
text(6, 5.4, labels="B")
#Picture 5
plot(A, B, xlab="", ylab="", pch=16, cex=2)
segments(1, 1, 4, 5, lwd=1, col = "red")
segments(1, 1, 4, 3, lwd = 1, col = "red")
segments(4, 3, 4, 5, lwd = 1, col = "red")
segments(6, 8, 10, 7, lwd = 1, col = "red")
segments(6, 8, 9, 10, lwd = 1, col = "red")
segments(10, 7, 9, 10, lwd = 1, col = "red")
segments(6, 8, 4, 5, lwd = 1, lty=2)
abline(9.833, -2/3, lwd=2)
```

Try to replicate these results with other data from the list of our Case-Studies.

11.10 Assignments: 11. Black Box Machine-Learning Methods: Neural Networks and Support Vector Machines

11.10.1 Learn and Predict a Power-Function

In Chap. 11, we learned about predicting the square-root function. It's just one instance of the power-function.

- Why did we observe a decrease in the accuracy of the NN prediction of the square-root outside the interval [0, 1] (note we trained inside [0, 1])? How can you improve on the prediction of the square-root network?
- Can you design a more generic NN network that can learn and predict a power-function for a given power ($\lambda \in \Re$)?

11.10.2 Pediatric Schizophrenia Study

Use the SOCR Normal and Schizophrenia pediatric neuroimaging study data to complete the following tasks:

- Conduct some initial data visualization and exploration.
- Use derived neuroimaging biomarkers (e.g., *Age*, *FS_IQ*, *TBV*, *GMV*, *WMV*, *CSF*, *Background*, *L_superior_frontal_gyrus*, *R_superior_frontal_gyrus*, ..., *brainstem*) to train a `NN` model and predict *DX* (Normals = 1; Schizophrenia = 2).
- Try one hidden layer with a different number of nodes.
- Try multiple hidden layers and compare the results to the single layer. Which model is better?
- Compare the type I (false-positive) and type II (false-negative) errors for the alternative methods.
- Train separate models to predict *DX* (diagnosis) for the *Male* and *Female* cohorts, respectively. Explain your findings.
- Train an *SVM*, using `ksvm` and `svm` in `e1071`, for *Age*, *FS_IQ*, *TBV*, *GMV*, *WMV*, *CSF*, *Background* to predict *DX*. Compare the results of linear, Gaussian, and polynomial SVM kernels.
- Add *Sex* to your models and see if this makes a difference.
- Expand the model by training on all derived neuroimaging biomarkers and re-train the SVM using *Age*, *FS_IQ*, *TBV*, *GMV*, *WMV*, *CSF*, *Background*, *L_superior_frontal_gyrus*, *R_superior_frontal_gyrus*, ..., *brainstem*. Again, try linear, Gaussian, and polynomial kernels. Compare the results.
- Are there differences between the alternative kernels?
- For *Age*, *FS_IQ*, *TBV*, *GMV*, *WMV*, *CSF*, and *Background*, tune parameters for Gaussian and polynomial kernels.

- Generate a CV (cross-validation) plot and interpret the resulting graph.
- Use different random seeds and repeat the experiment five times. Are the results stable?
- Inspecting the results above, explain why it makes sense to set a tune over a range such as $exp(-5:8)$.
- How can we design alternative tuning strategies other than greedy search?

References

Schölkopf, B, Smola, AJ. (2002) *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond Adaptive computation and machine learning*, MIT Press, ISBN 0262194759, 9780262194754.

Du, K-L, Swamy, MNS. (2013) *Neural Networks and Statistical Learning*, SpringerLink: Bücher, ISBN 1447155718, 9781447155713.

Abe, S. (2010) *Support Vector Machines for Pattern Classification*, Advances in Computer Vision and Pattern Recognition, Springer Science & Business Media, ISBN 1849960984, 9781849960984.