Chapter 3

Predicting Stock Market Returns

This second case study tries to move a bit further in terms of the use of data mining techniques. We will address some of the difficulties of incorporating data mining tools and techniques into a concrete business problem. The specific domain used to illustrate these problems is that of automatic stock trading systems. We will address the task of building a stock trading system based on prediction models obtained with daily stock quotes data. Several models will be tried with the goal of predicting the future returns of the S&P 500 market index. These predictions will be used together with a trading strategy to reach a decision regarding the market orders to generate. This chapter addresses several new data mining issues, among which are (1) how to use R to analyze data stored in a database, (2) how to handle prediction problems with a time ordering among data observations (also known as time series), and (3) an example of the difficulties of translating model predictions into decisions and actions in real-world applications.

3.1 Problem Description and Objectives

Stock market trading is an application domain with a large potential for data mining. In effect, the existence of an enormous amount of historical data suggests that data mining can provide a competitive advantage over human inspection of these data. On the other hand, there are researchers claiming that the markets adapt so rapidly in terms of price adjustments that there is no space to obtain profits in a consistent way. This is usually known as the *efficient markets hypothesis*. This theory has been successively replaced by more relaxed versions that leave some space for trading opportunities due to temporary market inefficiencies.

The general goal of stock trading is to maintain a portfolio of assets based on buy and sell orders. The long-term objective is to achieve as much profit as possible from these trading actions. In the context of this chapter we will constrain a bit more this general scenario. Namely, we will only "trade" a single security, actually a market index. Given this security and an initial capital, we will try to maximize our profit over a future testing period by means of trading actions (Buy, Sell, Hold). Our trading strategy will use as a basis for decision

making the indications provided by the result of a data mining process. This process will consist of trying to predict the future evolution of the index based on a model obtained with historical quotes data. Thus our prediction model will be incorporated in a trading system that generates its decisions based on the predictions of the model. Our overall evaluation criteria will be the performance of this trading system, that is, the profit/loss resulting from the actions of the system as well as some other statistics that are of interest to investors. This means that our main evaluation criteria will be the operational results of applying the knowledge discovered by our data mining process and not the predictive accuracy of the models developed during this process.

3.2 The Available Data

In our case study we will concentrate on trading the S&P 500 market index. Daily data concerning the quotes of this security are freely available in many places, for example, the Yahoo finance site.[1]

The data we will use is available in the book package. Once again we will explore other means of getting the data as a form of illustrating some of the capabilities of R. Moreover, some of these other alternatives will allow you to apply the concepts learned in this chapter to more recent data than the one packaged at the time of writing this book.

In order to get the data through the book R package, it is enough to issue

```
> library(DMwR)
> data(GSPC)
```

The first statement is only required if you have not issued it before in your R session. The second instruction will load an object, `GSPC`,[2] of class `xts`. We will describe this class of objects in Section 3.2.1, but for now you can manipulate it as if it were a matrix or a data frame (try, for example, `head(GSPC)`).

At the book Web site,[3] you can find these data in two alternative formats. The first is a comma separated values (CSV) file that can be read into R in the same way as the data used in Chapter 2. The other format is a MySQL database dump file that we can use to create a database with the S&P 500 quotes in MySQL. We will illustrate how to load these data into R for these two alternative formats. It is up to you to decide which alternative you will download, or if you prefer the easy path of loading it from the book package. The remainder of the chapter (i.e., the analysis after reading the data) is independent of the storage schema you decide to use.

[1] http://finance.yahoo.com.

[2] `^GSPC` is the ticker ID of S&P 500 at Yahoo finance from where the quotes were obtained.

[3] http://www.liaad.up.pt/~ltorgo/DataMiningWithR.

For the sake of completeness we will also mention yet another way of getting this data into R, which consists of downloading it directly from the Web. If you choose to follow this path, you should remember that you will probably be using a larger dataset than the one used in the analysis carried out in this book.

Whichever source you choose to use, the daily stock quotes data includes information regarding the following properties:

- Date of the stock exchange session
- Open price at the beginning of the session
- Highest price during the session
- Lowest price
- Closing price of the session
- Volume of transactions
- Adjusted close price[4]

3.2.1 Handling Time-Dependent Data in R

The data available for this case study depends on time. This means that each observation of our dataset has a time tag attached to it. This type of data is frequently known as time series data. The main distinguishing feature of this kind of data is that order between cases matters, due to their attached time tags. Generally speaking, a time series is a set of ordered observations of a variable Y:

$$y_1, y_2, \ldots, y_{t-1}, y_t, y_{t+1}, \ldots, y_n \tag{3.1}$$

where y_t is the value of the series variable Y at time t.

The main goal of time series analysis is to obtain a model based on past observations of the variable, $y_1, y_2, \ldots, y_{t-1}, y_t$, which allows us to make predictions regarding future observations of the variable, $y_{t+1}, \ldots, y_n$.

In the case of our stocks data, we have what is usually known as a multivariate time series, because we measure several variables at the same time tags, namely the $Open$, $High$, Low, $Close$, $Volume$, and $AdjClose$.[5]

R has several packages devoted to the analysis of this type of data, and in effect it has special classes of objects that are used to store type-dependent

[4]This is basically the closing price adjusted for stock splits, dividends/distributions, and rights offerings.

[5]Actually, if we wanted to be more precise, we would have to say that we have only two time series ($Price$ and $Volume$) because all quotes are actually the same variable ($Price$) sampled at different times of the day.

data. Moreover, R has many functions tuned for this type of objects, like special plotting functions, etc.

Among the most flexible R packages for handling time-dependent data are `zoo` (Zeileis and Grothendieck, 2005) and `xts` (Ryan and Ulrich, 2010). Both offer similar power, although `xts` provides a set of extra facilities (e.g., in terms of sub-setting using ISO 8601 time strings) to handle this type of data. In technical terms the class `xts` extends the class `zoo`, which means that any `xts` object is also a `zoo` object, and thus we can apply any method designed for `zoo` objects to `xts` objects. We will base our analysis in this chapter primarily on `xts` objects. We start with a few illustrative examples of the creation and use of this type of object. Please note that both `zoo` and `xts` are extra packages (i.e., that do not come with a base installation of R), and that you need to download and install in R (see Section 1.2.1, page 3).

The following examples illustrate how to create objects of class `xts`.

```
> library(xts)
> x1 <- xts(rnorm(100), seq(as.POSIXct("2000-01-01"), len = 100,
+     by = "day"))
> x1[1:5]

                 [,1]
2000-01-01  0.82029230
2000-01-02  0.99165376
2000-01-03  0.05829894
2000-01-04 -0.01566194
2000-01-05  2.02990349

> x2 <- xts(rnorm(100), seq(as.POSIXct("2000-01-01 13:00"),
+     len = 100, by = "min"))
> x2[1:4]

                        [,1]
2000-01-01 13:00:00 1.5638390
2000-01-01 13:01:00 0.7876171
2000-01-01 13:02:00 1.0860185
2000-01-01 13:03:00 1.2332406

> x3 <- xts(rnorm(3), as.Date(c("2005-01-01", "2005-01-10",
+     "2005-01-12")))
> x3

                [,1]
2005-01-01 -0.6733936
2005-01-10 -0.7392344
2005-01-12 -1.2165554
```

The function `xts()` receives the time series data in the first argument. This can either be a vector, or a matrix if we have a multivariate time series.[6]

[6]Note that this means that we cannot have `xts` with mix-mode data, such as in a data frame.

In the latter case each column of the matrix is interpreted as a variable being sampled at each time tag (i.e., each row). The time tags are provided in the second argument. This needs to be a set of time tags in any of the existing time classes in R. In the examples above we have used two of the most common classes to represent time information in R: the `POSIXct/POSIXlt` classes and the `Date` class. There are many functions associated with these objects for manipulating dates information, which you may want to check using the help facilities of R. One such example is the `seq()` function. We have used this function before to generate sequences of numbers. Here we are using it[7] to generate time-based sequences as you see in the example.

As you might observe in the above small examples, the objects may be indexed as if they were "normal" objects without time tags (in this case we see a standard vector sub-setting). Still, we will frequently want to subset these time series objects based on time-related conditions. This can be achieved in several ways with `xts` objects, as the following small examples try to illustrate:

```
> x1[as.POSIXct("2000-01-04")]

                 [,1]
2000-01-04 -0.01566194

> x1["2000-01-05"]

               [,1]
2000-01-05 2.029903

> x1["20000105"]

               [,1]
2000-01-05 2.029903

> x1["2000-04"]

                          [,1]
2000-04-01 01:00:00  0.2379293
2000-04-02 01:00:00 -0.1005608
2000-04-03 01:00:00  1.2982820
2000-04-04 01:00:00 -0.1454789
2000-04-05 01:00:00  1.0436033
2000-04-06 01:00:00 -0.3782062
2000-04-07 01:00:00 -1.4501869
2000-04-08 01:00:00 -1.4123785
2000-04-09 01:00:00  0.7864352

> x1["2000-03-27/"]
```

[7] Actually, it is a specific method of the generic function `seq()` applicable to objects of class `POSIXt`. You may know more about this typing "? seq.POSIXt".

```
                           [,1]
2000-03-27 01:00:00  0.10430346
2000-03-28 01:00:00 -0.53476341
2000-03-29 01:00:00  0.96020129
2000-03-30 01:00:00  0.01450541
2000-03-31 01:00:00 -0.29507179
2000-04-01 01:00:00  0.23792935
2000-04-02 01:00:00 -0.10056077
2000-04-03 01:00:00  1.29828201
2000-04-04 01:00:00 -0.14547894
2000-04-05 01:00:00  1.04360327
2000-04-06 01:00:00 -0.37820617
2000-04-07 01:00:00 -1.45018695
2000-04-08 01:00:00 -1.41237847
2000-04-09 01:00:00  0.78643516
```

```
> x1["2000-02-26/2000-03-03"]
```

```
                  [,1]
2000-02-26  1.77472194
2000-02-27 -0.49498043
2000-02-28  0.78994304
2000-02-29  0.21743473
2000-03-01  0.54130752
2000-03-02 -0.02972957
2000-03-03  0.49330270
```

```
> x1["/20000103"]
```

```
                 [,1]
2000-01-01 0.82029230
2000-01-02 0.99165376
2000-01-03 0.05829894
```

The first statement uses a concrete value of the same class as the object given in the second argument at the time of creation of the **x1** object. The other examples illustrate a powerful indexing schema introduced by the `xts` package, which is one of its advantages over other time series packages in R. This schema implements time tags as strings with the `CCYY-MM-DD HH:MM:SS[.s]` general format. As you can confirm in the examples, separators can be omitted and parts of the time specification left out to include sets of time tags. Moreover, the "/" symbol can be used to specify time intervals that can unspecified on both ends, with the meaning of start or final time tag.

Multiple time series can be created in a similar fashion as illustrated below:

```
> mts.vals <- matrix(round(rnorm(25),2),5,5)
> colnames(mts.vals) <- paste('ts',1:5,sep='')
> mts <- xts(mts.vals,as.POSIXct(c('2003-01-01','2003-01-04',
+                     '2003-01-05','2003-01-06','2003-02-16')))
> mts
```

```
            ts1   ts2   ts3   ts4   ts5
2003-01-01 0.96 -0.16 -1.03  0.17  0.62
2003-01-04 0.10  1.64 -0.83 -0.55  0.49
2003-01-05 0.38  0.03 -0.09 -0.64  1.37
2003-01-06 0.73  0.98 -0.66  0.09 -0.89
2003-02-16 2.68  0.10  1.44  1.37 -1.37
```

```
> mts["2003-01",c("ts2","ts5")]
```

```
             ts2   ts5
2003-01-01 -0.16  0.62
2003-01-04  1.64  0.49
2003-01-05  0.03  1.37
2003-01-06  0.98 -0.89
```

The functions `index()` and `time()` can be used to "extract" the time tags information of any `xts` object, while the `coredata()` function obtains the data values of the time series:

```
> index(mts)
```

```
[1] "2003-01-01 WET" "2003-01-04 WET" "2003-01-05 WET" "2003-01-06 WET"
[5] "2003-02-16 WET"
```

```
> coredata(mts)
```

```
      ts1   ts2   ts3   ts4   ts5
[1,] 0.96 -0.16 -1.03  0.17  0.62
[2,] 0.10  1.64 -0.83 -0.55  0.49
[3,] 0.38  0.03 -0.09 -0.64  1.37
[4,] 0.73  0.98 -0.66  0.09 -0.89
[5,] 2.68  0.10  1.44  1.37 -1.37
```

In summary, `xts` objects are adequate to store stock quotes data, as they allow to store multiple time series with irregular time tags, and provide powerful indexing schemes.

3.2.2 Reading the Data from the CSV File

As we have mentioned before, at the book Web site you can find different sources containing the data to use in this case study. If you decide to use the CSV file, you will download a file whose first lines look like this:

```
"Index"    "Open" "High" "Low" "Close" "Volume" "AdjClose"
1970-01-02 92.06  93.54  91.79 93      8050000  93
1970-01-05 93     94.25  92.53 93.46   11490000 93.46
1970-01-06 93.46  93.81  92.13 92.82   11460000 92.82
1970-01-07 92.82  93.38  91.93 92.63   10010000 92.63
1970-01-08 92.63  93.47  91.99 92.68   10670000 92.68
1970-01-09 92.68  93.25  91.82 92.4    9380000  92.4
1970-01-12 92.4   92.67  91.2  91.7    8900000  91.7
```

Assuming you have downloaded the file and have saved it with the name "sp500.csv" on the current working directory of your R session, you can load it into R and create an `xts` object with the data, as follows:

```
> GSPC <- as.xts(read.zoo("sp500.csv", header = T))
```

The function `read.zoo()` of package `zoo`[8] reads a CSV file and transforms the data into a `zoo` object assuming that the first column contains the time tags. The function `as.xts()` coerces the resulting object into an object of class `xts`.

3.2.3 Getting the Data from the Web

Another alternative way of getting the S&P 500 quotes is to use the free service provided by Yahoo finance, which allows you to download a CSV file with the quotes you want. The `tseries` (Trapletti and Hornik, 2009) R package[9] includes the function `get.hist.quote()` that can be used to download the quotes into a `zoo` object. The following is an example of the use of this function to get the quotes of S&P 500:

```
> library(tseries)
> GSPC <- as.xts(get.hist.quote("^GSPC",start="1970-01-02",
          quote=c("Open", "High", "Low", "Close","Volume","AdjClose")))

...
...

> head(GSPC)

            Open  High   Low Close   Volume AdjClose
1970-01-02 92.06 93.54 91.79 93.00  8050000    93.00
1970-01-05 93.00 94.25 92.53 93.46 11490000    93.46
1970-01-06 93.46 93.81 92.13 92.82 11460000    92.82
1970-01-07 92.82 93.38 91.93 92.63 10010000    92.63
1970-01-08 92.63 93.47 91.99 92.68 10670000    92.68
1970-01-09 92.68 93.25 91.82 92.40  9380000    92.40
```

As the function `get.hist.quote()` returns an object of class `zoo`, we have again used the function `as.xts()` to coerce it to `xts`. We should remark that if you issue these commands, you will get more data than what is provided with the object in the book package. If you want to ensure that you get the same results in future commands in this chapter, you should instead use the command

[8] You may wonder why we did not load the package `zoo` with a call to the `library()` function. The reason is that this was already done when we loaded the package `xts` because it depends on package `zoo`.

[9] Another extra package that needs to be installed.

```
> GSPC <- as.xts(get.hist.quote("^GSPC",
          start="1970-01-02",end='2009-09-15',
          quote=c("Open", "High", "Low", "Close","Volume","AdjClose")))
```

where "2009-09-15" is the last day with quotes in our package `GSPC` object.

Another way of obtaining quotes data from the Web (but not the only, as we will see later), is to use the function `getSymbols()` from package `quantmod` (Ryan, 2009). Again this is an extra package that you should install before using it. It provides several facilities related to financial data analysis that we will use throughout this chapter. Function `getSymbols()` in conjunction with other functions of this package provide a rather simple but powerful way of getting quotes data from different data sources. Let us see some examples of its use:

```
> library(quantmod)
> getSymbols("^GSPC")
```

The function `getSymbols()` receives on the first argument a set of symbol names and will fetch the quotes of these symbols from different Web sources or even local databases, returning by default an `xts` object with the same name as the symbol,[10] which will silently be created in the working environment. The function has many parameters that allow more control over some of these issues. As you can verify, the returned object does not cover the same period as the data coming with our book package, and it has slightly different column names. This can be easily worked around as follows:

```
> getSymbols("^GSPC", from = "1970-01-01", to = "2009-09-15")
> colnames(GSPC) <- c("Open", "High", "Low", "Close", "Volume",
+     "AdjClose")
```

With the framework provided by package `quantmod` you may actually have several symbols with different associated sources of data, each with its own parameters. All these settings can be specified at the start of your R session with the `setSymbolLookup()` function, as you may see in the following simple example:

```
> setSymbolLookup(IBM=list(name='IBM',src='yahoo'),
+                 USDEUR=list(name='USD/EUR',src='oanda'))
> getSymbols(c('IBM','USDEUR'))

> head(IBM)

           IBM.Open IBM.High IBM.Low IBM.Close IBM.Volume IBM.Adjusted
2007-01-03    97.18    98.40   96.26     97.27    9196800        92.01
2007-01-04    97.25    98.79   96.88     98.31   10524500        93.00
2007-01-05    97.60    97.95   96.91     97.42    7221300        92.16
2007-01-08    98.50    99.50   98.35     98.90   10340000        93.56
2007-01-09    99.08   100.33   99.07    100.07   11108200        94.66
2007-01-10    98.50    99.05   97.93     98.89    8744800        93.55
```

[10] Eventually pruned from invalid characters for R object names.

```
> head(USDEUR)

           USDEUR
2009-01-01 0.7123
2009-01-02 0.7159
2009-01-03 0.7183
2009-01-04 0.7187
2009-01-05 0.7188
2009-01-06 0.7271
```

In this code we have specified several settings for getting the quotes from the Web of two different symbols: IBM from Yahoo! finance; and US Dollar—Euro exchange rate from Oanda.[11] This is done through function `setSymbolLookup()`, which ensures any subsequent use of the `getSymbols()` function in the current R session with the identifiers specified in the call, will use the settings we want. In this context, the second statement will fetch the quotes of the two symbols using the information we have specified. Functions `saveSymbolLookup()` and `loadSymbolLookup()` can be used to save and load these settings across different R sessions. Check the help of these functions for further examples and more thorough explanations of the workings behind these handy functions.

3.2.4 Reading the Data from a MySQL Database

Another alternative form of storing the data used in this case study is in a MySQL database. At the book Web site there is a file containing SQL statements that can be downloaded and executed within MySQL to upload S&P 500 quotes into a database table. Information on the use and creation of MySQL databases can be found in Section 1.3 (page 35).

After creating a database to store the stock quotes, we are ready to execute the SQL statements of the file downloaded from the book site. Assuming that this file is in the same directory from where you have entered MySQL, and that the database you have created is named `Quotes`, you can log in to MySQL and then type

```
mysql> use Quotes;
mysql> source sp500.sql;
```

The SQL statements contained in the file "`sp500.sql`" (the file downloaded from the book Web site) will create a table named "gspc" and insert several records in this table containing the data available for this case study. You can confirm that everything is OK by executing the following statements at the MySQL prompt:

```
mysql> show tables;
```

[11] http://www.oanda.com.

```
+------------------+
| Tables_in_Quotes |
+------------------+
| gspc             |
+------------------+
1 row in set (0.00 sec)
```

```
mysql> select * from gspc;
```

The last SQL statement should print a large set of records, namely the quotes of S&P 500. If you want to limit this output, simply add `limit 10` at the end of the statement.

There are essentially two paths to communicate with databases in R. One based on the ODBC protocol and the other is based on the general interface provided by package `DBI` (R Special Interest Group on Databases, 2009) together with specific packages for each database management system (DBMS).

If you decide to use the ODBC protocol, you need to ensure that you are able to communicate with your DBMS using this protocol. This may involve installing some drivers on the DBMS side. From the side of R, you only need to install package `RODBC`.

Package `DBI` implements a series of database interface functions. These functions are independent of the database server that is actually used to store the data. The user only needs to indicate which communication interface he will use at the first step when he establishes a connection to the database. This means that if you change your DBMS, you will only need to change a single instruction (the one that specifies the DBMS you wish to communicate with). In order to achieve this independence the user also needs to install other packages that take care of the communication details for each different DBMS. R has many DBMS-specific packages for major DBMSs. Specifically, for communication with a MySQL database stored in some server, you have the package `RMySQL` (James and DebRoy, 2009).

3.2.4.1 Loading the Data into R Running on Windows

If you are running R on Windows, independently of whether the MySQL database server resides on that same PC or in another computer (eventually running other operating system), the simplest way to connect to the database from R is through the ODBC protocol. In order to use this protocol in R, you need to install the `RODBC` package.

Before you are able to connect to any MySQL database for the first time using the ODBC protocol, a few extra steps are necessary. Namely, you need also to install the MySQL ODBC driver on your Windows system, which is called "`myodbc`" and can be downloaded from the MySQL site. This only needs to be done the first time you use ODBC to connect to MySQL. After installing this driver, you can create ODBC connections to MySQL databases residing on your computer or any other system to which you have access through your

local network. According to the ODBC protocol, every database connection you create has a name (the *Data Source Name*, or DSN according to ODBC jargon). This name will be used to access the MySQL database from R. To create an ODBC connection on a Windows PC, you must use a program called "ODBC data sources", available at the Windows control panel. After running this program you have to create a new User Data Source using the MySQL ODBC driver (`myodbc`) that you are supposed to have previously installed. During this creation process, you will be asked several things, such as the MySQL server address (`localhost` if it is your own computer, or e.g., `myserver.xpto.pt` if it is a remote server), the name of the database to which you want to establish a connection (`Quotes` in our previous example), and the name you wish to give to this connection (the DSN). Once you have completed this process, which you only have to do for the first time, you are ready to connect to this MySQL database from R.

The following R code establishes a connection to the `Quotes` database from R, and loads the S&P 500 quotes data into a data frame,

```
> library(RODBC)
> ch <- odbcConnect("QuotesDSN",uid="myusername",pwd="mypassword")
> allQuotes <- sqlFetch(ch,"gspc")
> GSPC <- xts(allQuotes[,-1],order.by=as.Date(allQuotes[,1]))
> head(GSPC)

            Open  High   Low Close   Volume AdjClose
1970-01-02 92.06 93.54 91.79 93.00  8050000    93.00
1970-01-05 93.00 94.25 92.53 93.46 11490000    93.46
1970-01-06 93.46 93.81 92.13 92.82 11460000    92.82
1970-01-07 92.82 93.38 91.93 92.63 10010000    92.63
1970-01-08 92.63 93.47 91.99 92.68 10670000    92.68
1970-01-09 92.68 93.25 91.82 92.40  9380000    92.40

> odbcClose(ch)
```

After loading the `RODBC` package, we establish a connection with our database using the previously created DSN,[12] using the function `odbcConnect()`. We then use one of the functions available to query a table, in this case the `sqlFetch()` function, which obtains all rows of a table and returns them as a data frame object. The next step is to create an `xts` object from this data frame using the date information and the quotes. Finally, we close the connection to the database with the `odbcClose()` function.

A brief note on working with extremely large databases: If your query generates a result too large to fit in your computer main memory, then you have to use some other strategy. If that is feasible for your analysis, you can try to handle the data in chunks, and this can be achieved with the parameter `max` of the functions `sqlFecth()` and `sqlFecthMore()`. Other alternatives/approaches

[12] Here you should substitute whichever DSN name you have used when creating the data source in the Windows control panel, and also your MySQL username and password.

can be found in the High-Performance and Parallel Computing task view,[13] for instance, through the package `ff` (Adler et al., 2010).

3.2.4.2 Loading the Data into R Running on Linux

In case you are running R from a Unix-type box the easiest way to communicate to your MySQL database is probably through the package `DBI` in conjunction with the package `RMySQL`. Still, the ODBC protocol is also available for these operating systems. With the `RMySQL` package you do not need any preparatory stages as with `RODBC`. After installing the package you can start using it as shown by the following example.

```
> library(DBI)
> library(RMySQL)
> drv <- dbDriver("MySQL")
> ch <- dbConnect(drv,dbname="Quotes","myusername","mypassword")
> allQuotes <- dbGetQuery(ch,"select * from gspc")
> GSPC <- xts(allQuotes[,-1],order.by=as.Date(allQuotes[,1]))
> head(GSPC)

            Open  High   Low Close   Volume AdjClose
1970-01-02 92.06 93.54 91.79 93.00  8050000    93.00
1970-01-05 93.00 94.25 92.53 93.46 11490000    93.46
1970-01-06 93.46 93.81 92.13 92.82 11460000    92.82
1970-01-07 92.82 93.38 91.93 92.63 10010000    92.63
1970-01-08 92.63 93.47 91.99 92.68 10670000    92.68
1970-01-09 92.68 93.25 91.82 92.40  9380000    92.40

> dbDisconnect(ch)

[1] TRUE

> dbUnloadDriver(drv)
```

After loading the packages, we open the connection with the database using the functions `dbDriver()` and `dbConnect()`, with obvious semantics. The function `dbGetQuery()` allows us to send an SQL query to the database and receive the result as a data frame. After the usual conversion to an `xts` object, we close the database connection using the `dbDisconnect()` and `dbUnloadDriver()`. Further functions, including functions to obtain partial chunks of queries, also exist in the package `DBI` and may be consulted in the package documentation.

Another possibility regarding the use of data in a MySQL database is to use the infrastructure provided by the `quantmod` package that we described in Section 3.2.3. In effect, the function `getSymbols()` can use as source a MySQL database. The following is a simple illustration of its use assuming a database as the one described above:

[13] http://cran.at.r-project.org/web/views/HighPerformanceComputing.html.

```
> setSymbolLookup(GSPC=list(name='gspc',src='mysql',
+  db.fields=c('Index','Open','High','Low','Close','Volume','AdjClose'),
+  user='xpto',password='ypto',dbname='Quotes'))
> getSymbols('GSPC')

[1] "GSPC"
```

3.3 Defining the Prediction Tasks

Generally speaking, our goal is to have good forecasts of the future price of the S&P 500 index so that profitable orders can be placed on time. This general goal should allow us to easily define what to predict with our models—it should resort to forecast the future values of the price time series. However, it is easy to see that even with this simple task we immediately face several questions, namely, (1) which of the daily quotes? or (2) for which time in the future? Answering these questions may not be easy and usually depends on how the predictions will be used for generating trading orders.

3.3.1 What to Predict?

The trading strategies we will describe in Section 3.5 assume that we obtain a prediction of the tendency of the market in the next few days. Based on this prediction, we will place orders that will be profitable if the tendency is confirmed in the future.

Let us assume that if the prices vary more than $p\%$, we consider this worthwhile in terms of trading (e.g., covering transaction costs). In this context, we want our prediction models to forecast whether this margin is attainable in the next k days.[14] Please note that within these k days we can actually observe prices both above and below this percentage. This means that predicting a particular quote for a specific future time $t+k$ might not be the best idea. In effect, what we want is to have a prediction of the overall dynamics of the price in the next k days, and this is not captured by a particular price at a specific time. For instance, the closing price at time $t+k$ may represent a variation much lower than $p\%$, but it could have been preceded by a period of prices representing variations much higher than $p\%$ within the window $t \cdots t+k$. So, what we want in effect is to have a good prediction of the overall tendency of the prices in the next k days.

We will describe a variable, calculated with the quotes data, that can be seen as an indicator (a value) of the tendency in the next k days. The value of this indicator should be related to the confidence we have that the target margin p will be attainable in the next k days. At this stage it is important

[14] We obviously do not want to be waiting years to obtain the profit margin.

to note that when we mention a variation in $p\%$, we mean above or below the current price. The idea is that positive variations will lead us to buy, while negative variations will trigger sell actions. The indicator we are proposing resumes the tendency as a single value, positive for upward tendencies, and negative for downward price tendencies.

Let the daily average price be approximated by

$$\bar{P}_i = \frac{C_i + H_i + L_i}{3} \tag{3.2}$$

where C_i, H_i and L_i are the close, high, and low quotes for day i, respectively.

Let V_i be the set of k percentage variations of today's close to the following k days average prices (often called arithmetic returns):

$$V_i = \left\{ \frac{\bar{P}_{i+j} - C_i}{C_i} \right\}_{j=1}^{k} \tag{3.3}$$

Our indicator variable is the total sum of the variations whose absolute value is above our target margin $p\%$:

$$T_i = \sum_v \{v \in V_i : v > p\% \vee v < -p\%\} \tag{3.4}$$

The general idea of the variable T is to signal k-days periods that have several days with average daily prices clearly above the target variation. High positive values of T mean that there are several average daily prices that are $p\%$ higher than today's close. Such situations are good indications of potential opportunities to issue a buy order, as we have good expectations that the prices will rise. On the other hand, highly negative values of T suggest sell actions, given the prices will probably decline. Values around zero can be caused by periods with "flat" prices or by conflicting positive and negative variations that cancel each other.

The following function implements this simple indicator:

```
> T.ind <- function(quotes, tgt.margin = 0.025, n.days = 10) {
+     v <- apply(HLC(quotes), 1, mean)
+     r <- matrix(NA, ncol = n.days, nrow = NROW(quotes))
+     for (x in 1:n.days) r[, x] <- Next(Delt(v, k = x), x)
+     x <- apply(r, 1, function(x) sum(x[x > tgt.margin | x <
+         -tgt.margin]))
+     if (is.xts(quotes))
+         xts(x, time(quotes))
+     else x
+ }
```

The function starts by obtaining the average price calculated according to Equation 3.2. The function `HLC()` extracts the High, Low, and Close quotes from a quotes object. We then obtain the returns of the next `n.days` days with respect to the current close price. The `Next()` function allows one to

shift the values of a time series in time (both forward or backward). The `Delt()` function can be used to calculate percentage or log returns of a series of prices. Finally, the `T.ind()` function sums up the large absolute returns, that is, returns above the target variation margin, which we have set by default to 2.5%.

We can get a better idea of the behavior of this indicator in Figure 3.1, which was produced with the following code:

```
> candleChart(last(GSPC, "3 months"), theme = "white", TA = NULL)
> avgPrice <- function(p) apply(HLC(p), 1, mean)
> addAvgPrice <- newTA(FUN = avgPrice, col = 1, legend = "AvgPrice")
> addT.ind <- newTA(FUN = T.ind, col = "red", legend = "tgtRet")
> addAvgPrice(on = 1)
> addT.ind()
```

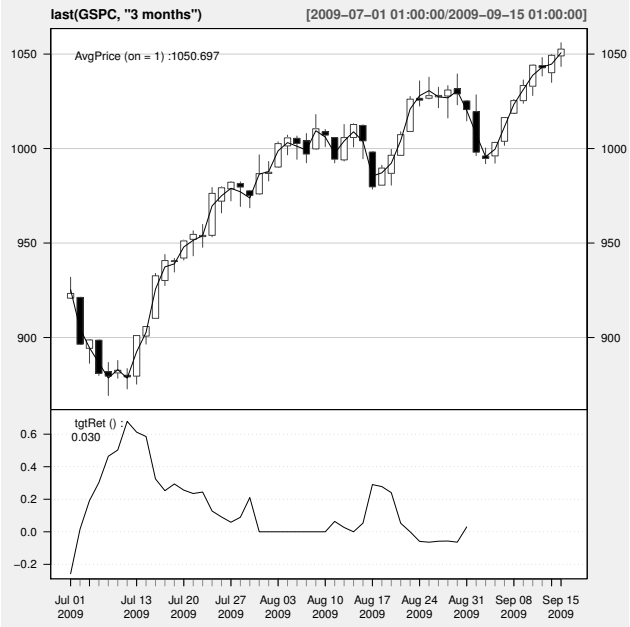

FIGURE 3.1: S&P500 on the last 3 months and our indicator.

The function `candleChart()` draws candlestick graphs of stock quotes. These graphs represent the daily quotes by a colored box and a vertical bar. The bar represents the High and Low prices of the day, while the box represents the Open-Close amplitude. The color of the box indicates if the top of the box is the Open or the Close price, that is, if the prices declined (black in Figure 3.1, orange in an interactive R session) or rose (white in our graphs, green in R sessions) across the daily session. We have added to the candlestick graph two indicators: the average price (on the same graph as the candlesticks)

and our T indicator (below). The function `newTA()` can be used to create new plotting functions for indicators that we wish to include in candlestick graphs. The return value of this function is a plotting function![15] This means that the objects `addT.ind` and `addAvgPrice` can be called like any other R function. This is done on the last two instructions. Each of them adds an indicator to the initial graph produced by the `candleChart()` function. The function `addAvgPrice()` was called with the parameter set to 1, which means that the indicator will be plotted on the first graph window; that is, the graph where the candlesticks are. The function `addT.ind()` was not called with this argument, leading to a new graph below the candlesticks. This is what makes sense in the case of our indicator, given the completely different scale of values.

As you can observe in Figure 3.1, the T indicator achieves the highest values when there is a subsequent period of positive variations. Obviously, to obtain the value of the indicator for time i, we need to have the quotes for the following 10 days, so we are not saying that T anticipates these movements. This is not the goal of the indicator. Its goal is to summarize the observed future behavior of the prices into a single value and not to predict this behavior!

In our approach to this problem we will assume that the correct trading action at time t is related to what our expectations are concerning the evolution of prices in the next k days. Moreover, we will describe this future evolution of the prices by our indicator T. The correct trading signal at time t will be "buy" if the T score is higher than a certain threshold, and will be "sell" if the score is below another threshold. In all other cases, the correct signal will be do nothing (i.e., "hold"). In summary, we want to be able to predict the correct signal for time t. On historical data we will fill in the correct signal for each day by calculating the respective T scores and using the thresholding method just outlined above.

3.3.2 Which Predictors?

We have defined an indicator (T) that summarizes the behavior of the price time series in the next k days. Our data mining goal will be to predict this behavior. The main assumption behind trying to forecast the future behavior of financial markets is that it is possible to do so by observing the past behavior of the market. More precisely, we are assuming that if in the past a certain behavior p was followed by another behavior f, and if that causal chain happened frequently, then it is plausible to assume that this will occur again in the future; and thus if we observe p now, we predict that we will observe f next. We are approximating the future behavior (f), by our indicator T. We now have to decide on how we will describe the recent prices pattern (p in the description above). Instead of using again a single indicator to de-

[15] You can confirm that by issuing `class(addT.ind)` or by typing the name of the object to obtain its contents.

scribe these recent dynamics, we will use several indicators, trying to capture different properties of the price time series to facilitate the forecasting task.

The simplest type of information we can use to describe the past are the recent observed prices. Informally, that is the type of approach followed in several standard time series modeling approaches. These approaches develop models that describe the relationship between future values of a time series and a window of past q observations of this time series. We will try to enrich our description of the current dynamics of the time series by adding further features to this window of recent prices.

Technical indicators are numeric summaries that reflect some properties of the price time series. Despite their debatable use as tools for deciding when to trade, they can nevertheless provide interesting summaries of the dynamics of a price time series. The amount of technical indicators available can be overwhelming. In R we can find a very good sample of them, thanks to package `TTR` (Ulrich, 2009).

The indicators usually try to capture some properties of the prices series, such as if they are varying too much, or following some specific trend, etc. In our approach to this problem, we will not carry out an exhaustive search for the indicators that are most adequate to our task. Still, this is a relevant research question, and not only for this particular application. It is usually known as the feature selection problem, and can informally be defined as the task of finding the most adequate subset of available input variables for a modeling task. The existing approaches to this problem can usually be cast in two groups: (1) feature filters and (2) feature wrappers. The former are independent of the modeling tool that will be used after the feature selection phase. They basically try to use some statistical properties of the features (e.g., correlation) to select the final set of features. The wrapper approaches include the modeling tool in the selection process. They carry out an iterative search process where at each step a candidate set of features is tried with the modeling tool and the respective results are recorded. Based on these results, new tentative sets are generated using some search operators, and the process is repeated until some convergence criteria are met that will define the final set.

We will use a simple approach to select the features to include in our model. The idea is to illustrate this process with a concrete example and not to find the best possible solution to this problem, which would require other time and computational resources. We will define an initial set of features and then use a technique to estimate the importance of each of these features. Based on these estimates we will select the most relevant features.

We will center our analysis on the Close quote, as our buy/sell decisions will be made at the end of each daily session. The initial set of features will be formed by several past returns on the Close price. The h-days (arithmetic) returns,[16] or percentage variations, can be calculated as

[16] Log returns are defined as $\log(C_i/C_{i-h})$.

$$R_{i-h} = \frac{C_i - C_{i-h}}{C_{i-h}} \tag{3.5}$$

where C_i is the Close price at session i.

We have included in the set of candidate features ten of these returns by varying h from 1 to 10. Next, we have selected a representative set of technical indicators, from those available in package `TTR`—namely, the Average True Range (ATR), which is an indicator of the volatility of the series; the Stochastic Momentum Index (SMI), which is a momentum indicator; the Welles Wilder's Directional Movement Index (ADX); the Aroon indicator that tries to identify starting trends; the Bollinger Bands that compare the volatility over a period of time; the Chaikin Volatility; the Close Location Value (CLV) that relates the session Close to its trading range; the Arms' Ease of Movement Value (EMV); the MACD oscillator; the Money Flow Index (MFI); the Parabolic Stop-and-Reverse; and the Volatility indicator. More details and references on these and other indicators can be found in the respective help pages of the functions implementing them in package `TTR`. Most of these indicators produce several values that together are used for making trading decisions. As mentioned before, we do not plan to use these indicators for trading. As such, we have carried out some post-processing of the output of the `TTR` functions to obtain a single value for each one. The following functions implement this process:

```
> myATR <- function(x) ATR(HLC(x))[, "atr"]
> mySMI <- function(x) SMI(HLC(x))[, "SMI"]
> myADX <- function(x) ADX(HLC(x))[, "ADX"]
> myAroon <- function(x) aroon(x[, c("High", "Low")])$oscillator
> myBB <- function(x) BBands(HLC(x))[, "pctB"]
> myChaikinVol <- function(x) Delt(chaikinVolatility(x[, c("High",
+     "Low")]))[, 1]
> myCLV <- function(x) EMA(CLV(HLC(x)))[, 1]
> myEMV <- function(x) EMV(x[, c("High", "Low")], x[, "Volume"])[,
+     2]
> myMACD <- function(x) MACD(Cl(x))[, 2]
> myMFI <- function(x) MFI(x[, c("High", "Low", "Close")],
+     x[, "Volume"])
> mySAR <- function(x) SAR(x[, c("High", "Close")])[, 1]
> myVolat <- function(x) volatility(OHLC(x), calc = "garman")[,
+     1]
```

The variables we have just described form our initial set of predictors for the task of forecasting the future value of the T indicator. We will try to reduce this set of 22 variables using a feature selection method. Random forests (Breiman, 2001) were used in Section 2.7 to obtain predictions of algae occurrences. Random forests can also be used to estimate the importance of the variables involved in a prediction task. Informally, this importance can be estimated by calculating the percentage increase in the error of the random

forest if we remove each variable in turn. In a certain way this resembles the idea of wrapper filters as it includes a modeling tool in the process of selecting the features. However, this is not an iterative search process and moreover, we will use other predictive models to forecast T, which means that the set of variables selected by this process is not optimized for these other models, and in this sense this method is used more like a filter approach.

In our approach to this application, we will split the available data into two separate sets: (1) one used for constructing the trading system; and (2) other to test it. The first set will be formed by the first 30 years of quotes of S&P 500. We will leave the remaining data (around 9 years) for the final test of our trading system. In this context, we must leave this final test set out of this feature selection process to ensure unbiased results.

We first build a random forest using the data available for training:

```
> data(GSPC)
> library(randomForest)
> data.model <- specifyModel(T.ind(GSPC) ~ Delt(Cl(GSPC),k=1:10) +
+         myATR(GSPC) + mySMI(GSPC) + myADX(GSPC) + myAroon(GSPC) +
+         myBB(GSPC)  + myChaikinVol(GSPC) + myCLV(GSPC) +
+         CMO(Cl(GSPC)) + EMA(Delt(Cl(GSPC))) + myEMV(GSPC) +
+         myVolat(GSPC)  + myMACD(GSPC) + myMFI(GSPC) + RSI(Cl(GSPC)) +
+         mySAR(GSPC) + runMean(Cl(GSPC)) + runSD(Cl(GSPC)))
> set.seed(1234)
> rf <- buildModel(data.model,method='randomForest',
+               training.per=c(start(GSPC),index(GSPC["1999-12-31"])),
+               ntree=50, importance=T)
```

The code given above starts by specifying and obtaining the data to be used for modeling using the function `specifyModel()`. This function creates a `quantmod` object that contains the specification of a certain abstract model (described by a formula). This specification may refer to data coming from different types of sources, some of which may even not be currently in the memory of the computer. The function will take care of these cases using `getSymbols()` to obtain the necessary data. This results in a very handy form of specifying and getting the data necessary for your subsequent modeling stages. Moreover, for symbols whose source is the Web, you can later use the obtained object (`data.model` in our case) as an argument to the function `getModelData()`, to obtain a refresh of the object including any new quotes that may be available at that time. Again, this is quite convenient if you want to maintain a trading system that should be updated with new quotes information.

The function `buildModel()` uses the resulting model specification and obtains a model with the corresponding data. Through, parameter `training.per`, you can specify the data that should be used to obtain the model (we are using the first 30 years). This function currently contains wrap-

pers for several modeling tools,[17] among which are random forests. In case you wish to use a model not contemplated by `buildModel()`, you may obtain the data using the function `modelData()`, and use it with your favorite modeling function, as shown in the following illustrative example:

```
> ex.model <- specifyModel(T.ind(IBM) ~ Delt(Cl(IBM), k = 1:3))
> data <- modelData(ex.model, data.window = c("2009-01-01",
+     "2009-08-10"))
```

The obtained `data` object is a standard `zoo` object, which can be easily cast into a matrix or data frame, for use as a parameter of any modeling function, as the following artificial[18] example illustrates:

```
> m <- myFavouriteModellingTool(ex.model@model.formula,
+                               as.data.frame(data))
```

Notice how we have indicated the model formula. The "real" formula is not exactly the same as the one provided in the argument of function `specifyModel()`. This latter formula is used to fetch the data, but the "real" formula should use whichever columns and respective names the `specifyModel()` call has generated. This information is contained in the slot `model.formula` of the `quantmod` object generated by the function.

Notice that on this small artificial example we have mentioned a ticker (IBM) for which we currently had no data in memory. The `specifyModel()` function takes care of that by silently fetching the quotes data from the Web using the `getSymbols()` function. All this is done in a transparent way to the user and you may even include symbols in your model specification that are obtained from different sources (see, for instance, the examples in Section 3.2.3 with the function `setSymbolLookup()`).

Returning to our feature selection problem, notice that we have included the parameter `importance=TRUE` so that the random forest estimates the variable importance. For regression problems, the R implementation of random forests estimates variable importance with two alternative scores. The first is the percentage increase in the error of the forest if we remove each variable in turn. This is measured by calculating the increase in the mean squared error of each tree on an out-of-bag sample when each variable is removed. This increase is averaged over all trees in the forest and normalized with the standard error. The second score has to do with the decrease in node impurity that is accountable with each variable, again averaged over all trees. We will use the first score as it is the one mentioned in the original paper on random forests (Breiman, 2001). After obtaining the model, we can check the importance of the variables as follows:

```
> varImpPlot(rf@fitted.model, type = 1)
```

[17] Check its help page to know which ones.
[18] Do not run it as this is a "fake" modeling tool.

The result of this function call is given in Figure 3.2. The arguments to the `varImpPlot()` function are the random forest and the score we wish to plot (if ommited both are plotted). The generic function `buildModel()` returns the obtained model as a slot (`fitted.model`) of the `quantmod` object it produces as a result.

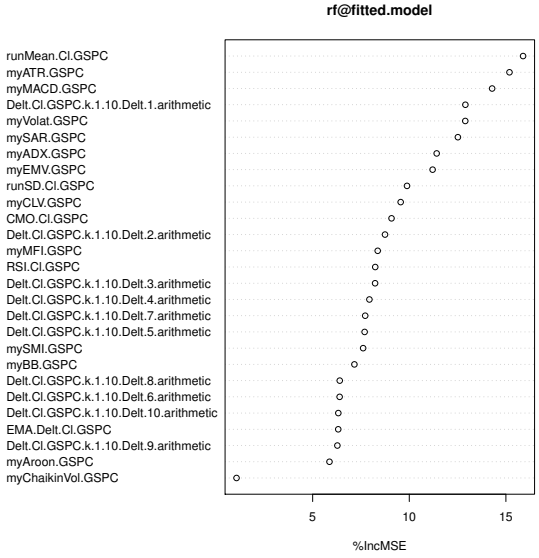

FIGURE 3.2: Variable importance according to the random forest.

At this stage we need to decide on a threshold on the importance score to select only a subset of the features. Looking at the results on the figure and given that this is a simple illustration of the concept of using random forests for selecting features, we will use the value of 10 as the threshold:

```
> imp <- importance(rf@fitted.model, type = 1)
> rownames(imp)[which(imp > 10)]

[1] "Delt.Cl.GSPC.k.1.10.Delt.1.arithmetic"
[2] "myATR.GSPC"
[3] "myADX.GSPC"
[4] "myEMV.GSPC"
[5] "myVolat.GSPC"
[6] "myMACD.GSPC"
[7] "mySAR.GSPC"
[8] "runMean.Cl.GSPC"
```

The function `importance()` obtains the concrete scores (in this case the first score) for each variable, which we then filter with our threshold to obtain

the names of the variables that we will use in our modeling attempts. Using this information we can obtain our final data set as follows:

```
> data.model <- specifyModel(T.ind(GSPC) ~ Delt(Cl(GSPC), k = 1) +
+     myATR(GSPC) + myADX(GSPC) + myEMV(GSPC) + myVolat(GSPC) +
+     myMACD(GSPC) + mySAR(GSPC) + runMean(Cl(GSPC)))
```

3.3.3 The Prediction Tasks

In the previous section we have obtained a `quantmod` object (`data.model`) containing the data we plan to use with our predictive models. This data has as a target the value of the T indicator and as predictors a series of other variables that resulted from a feature selection process. We have seen in Section 3.3.1 that our real goal is to predict the correct trading signal at any time t. How can we do that, given the data we have generated in the previous section? We will explore two paths to obtain predictions for the correct trading signal.

The first alternative is to use the T value as the target variable and try to obtain models that forecast this value using the predictors information. This is a multiple regression task similar to the ones we considered in the previous chapter. If we follow this path, we will then have to "translate" our model predictions into trading signals. This means to decide upon the thresholds on the predicted T values that will lead to either of the three possible trading actions. We will carry out this transformation using the following values:

$$signal = \begin{cases} sell & \text{if } T < -0.1 \\ hold & \text{if } -0.1 \leq T \leq 0.1 \\ buy & \text{if } T > 0.1 \end{cases} \tag{3.6}$$

The selection of the values 0.1 and -0.1 is purely heuristic and we can also use other thresholds. Still, these values mean that during the 10 day-period used to generate the T values, there were at least four average daily prices that are 2.5% above the current close ($4 \times 0.025 = 0.1$). If you decide to use other values, you should consider that too high absolute values will originate fewer signals, while too small values may lead us to trade on too small variations of the market, thus incurring a larger risk. Function `trading.signals()`, available in the book package, can carry out this transformation of the numeric T values into a factor with three possible values: "s", "h", and "b", for sell, hold and buy actions, respectively.

The second alternative prediction task we consider consists of predicting the signals directly. This means to use as a target variable the "correct" signal for day d. How do we obtain these correct signals? Again using the T indicator and the same thresholds used in Equation 3.6. For the available historical data, we obtain the signal of each day by calculating the T value using the following 10 days and using the thresholds in Equation 3.6 to decide on the signal. The target variable in this second task is nominal. This type of prediction problem

is known as a classification task.[19] The main distinction between classification and regression tasks is thus the type of the target variable. Regression tasks have a numeric target variable (e.g., our T indicator), while classification tasks use a nominal target variable, that is, with a finite set of possible values. Different approaches and techniques are used for these two types of problems.

The `xts` package infrastructure is geared toward numeric data. The data slots of `xts` objects must be either vectors or matrices, thus single mode data. This means it is not possible to have one of the columns of our training data as a nominal variable (a factor in R), together with all the numeric predictors. We will overcome this difficulty by carrying out all modeling steps outside the `xts` framework. This is easy and not limiting, as we will see. The infrastructure provided by `xts` is mostly used for data sub-setting and plotting, but the modeling stages do not need these facilities.

The following code creates all the data structures that we will use in the subsequent sections for obtaining predictive models for the two tasks.

```
> Tdata.train <- as.data.frame(modelData(data.model,
+                        data.window=c('1970-01-02','1999-12-31')))
> Tdata.eval <- na.omit(as.data.frame(modelData(data.model,
+                        data.window=c('2000-01-01','2009-09-15'))))
> Tform <- as.formula('T.ind.GSPC ~ .')
```

The `Tdata.train` and `Tdata.eval` are data frames with the data to be used for the training and evaluation periods, respectively. We have used data frames as the basic data structures to allow for mixed mode data that will be required in the classification tasks. For these tasks we will replace the target value column with the corresponding signals that will be generated using the `trading.signals()` function. The `Tdata.eval` data frame will be left out of all model selection and comparison processes we carry out. It will be used in the final evaluation of the "best" models we select. The call to `na.omit()` is necessary to avoid `NA`s at the end of the data frame caused by lack of future data to calculate the T indicator.

3.3.4 Evaluation Criteria

The prediction tasks described in the previous section can be used to obtain models that will output some form of indication regarding the future market direction. This indication will be a number in the case of the regression tasks (the predicted value of T), or a direct signal in the case of classification tasks. Even in the case of regression tasks, we have seen that we will cast this number into a signal by a thresholding mechanism. In Section 3.5 we will describe several trading strategies that use these predicted signals to act on the market.

In this section we will address the question of how to evaluate the signal predictions of our models. We will not consider the evaluation of the numeric

[19] Some statistics schools prefer the term "discrimination tasks".

predictions of the T indicator. Due to the way we are using these numeric predictions, this evaluation is a bit irrelevant. One might even question whether it makes sense to have these regression tasks, given that we are only interested in the trading signals. We have decided to maintain these numeric tasks because different trading strategies could take advantage of the numeric predictions, for instance, to decide which amount of money to invest when opening a position. For example, T values much higher than our thresholds for acting ($T > 0.1$ for buying and $T < -0.1$ for selling) could lead to stronger investments.

The evaluation of the signal predictions could be carried out by measuring the error rate, defined as

$$error.rate = \frac{1}{N}\sum_{i=1}^{N} L_{0/1}(y_i, \hat{y}_i) \tag{3.7}$$

where $\hat{y}_i$ is the prediction of the model for test case i, which has true class label y_i, and $L_{0/1}$ is known as the 0/1 loss function:

$$L_{0/1}(y_i, \hat{y}_i) = \begin{cases} 1 & \text{if } \hat{y}_i \neq y_i \\ 0 & \text{if } \hat{y}_i = y_i \end{cases} \tag{3.8}$$

One often uses the complement of this measure, known as *accuracy*, given by $1 - error.rate$.

These two statistics basically compare the model predictions to what really happened to the markets in the k future days.

The problem with accuracy (or error rate) is that it turns out not to be a good measure for this type of problem. In effect, there will be a very strong imbalance between the three possible outcomes, with a strong prevalence of hold signals over the other two, as big movements in prices are rare phenomena in financial markets.[20] This means that the accuracy scores will be dominated by the performance of the models on the most frequent outcome that is *hold*. This is not very interesting for trading. We want to have models that are accurate at the rare signals (*buy* and *sell*). These are the ones that lead to market actions and thus potential profit—the final goal of this application.

Financial markets forecasting is an example of an application driven by rare events. Event-based prediction tasks are usually evaluated by the precision and recall metrics that focus the evaluation on the events, disregarding the performance of the common situations (in our case, the hold signals). *Precision* can be informally defined as the proportion of event signals produced by the models that are correct. *Recall* is defined as the proportion of events occurring in the domain that is signaled as such by the models. These metrics can be easily calculated with the help of confusion matrices that sum up the results of a model in terms of the comparison between its predictions and the true values for a particular test set. Table 3.1 shows an example of a confusion matrix for our domain.

[20] This obviously depends on the target profit margin you establish; but to cover the trading costs, this margin should be large enough, and this rarity will be a fact.

TABLE 3.1: A Confusion Matrix for the Prediction of Trading Signals

		Predictions			
		sell	hold	buy	
True Values	sell	$n_{s,s}$	$n_{s,h}$	$n_{s,b}$	$N_{s,.}$
	hold	$n_{h,s}$	$n_{h,h}$	$n_{h,b}$	$N_{h,.}$
	buy	$n_{b,s}$	$n_{b,h}$	$n_{b,b}$	$N_{b,.}$
		$N_{.,s}$	$N_{.,h}$	$N_{.,b}$	N

With the help of Table 3.1 we can formalize the notions of precision and recall for this problem, as follows:

$$Prec = \frac{n_{s,s} + n_{b,b}}{N_{.,s} + N_{.,b}} \tag{3.9}$$

$$Rec = \frac{n_{s,s} + n_{b,b}}{N_{s,.} + N_{b,.}} \tag{3.10}$$

We can also calculate these statistics for particular signals by obtaining the precision and recall for sell and buy signals, independently; for example,

$$Prec_b = \frac{n_{b,b}}{N_{.,b}} \tag{3.11}$$

$$Rec_b = \frac{n_{b,b}}{N_{b,.}} \tag{3.12}$$

Precision and recall are often "merged" into a single statistic, called the $F - measure$ (Rijsbergen, 1979), given by

$$F = \frac{\left(\beta^2 + 1\right) \cdot Prec \cdot Rec}{\beta^2 \cdot Prec + Rec} \tag{3.13}$$

where $0 \leq \beta \leq 1$, controls the relative importance of recall to precision.

3.4 The Prediction Models

In this section we will explore some models that can be used to address the prediction tasks defined in the previous section. The selection of models was mainly guided by the fact that these techniques are well known by their ability to handle highly nonlinear regression problems. That is the case in our problem. Still, many other methods could have been applied to this problem. Any thorough approach to this domain would necessarily require a larger comparison of more alternatives. In the context of this book, such exploration does not make sense due to its costs in terms of space and computation power required.

3.4.1 How Will the Training Data Be Used?

Complex time series problems frequently exhibit different regimes, such as periods with strong variability followed by more "stable" periods, or periods with some form of systematic tendency. These types of phenomena are often called non-stationarities and can cause serious problems to several modeling techniques due to their underlying assumptions. It is reasonably easy to see, for instance by plotting the price time series, that this is the case for our data. There are several strategies we can follow to try to overcome the negative impact of these effects. For instance, several transformation techniques can be applied to the original time series to eliminate some of the effects. The use of percentage variations (returns) instead of the original absolute price values is such an example. Other approaches include using the available data in a more selective way. Let us suppose we are given the task of obtaining a model using a certain period of training data and then testing it in a subsequent period. The standard approach would use the training data to develop the model that would then be applied to obtain predictions for the testing period. If we have strong reason to believe that there are regime shifts, using the same model on all testing periods may not be the best idea, particularly if during this period there is some regime change that can seriously damage the performance of the model. In these cases it is often better to change or adapt the model using more recent data that better captures the current regime of the data.

In time series problems there is an implicit (time) ordering among the test cases. In this context, it makes sense to assume that when we are obtaining a prediction for time i, all test cases with time tag $k < i$ already belong to the past. This means that it is safe to assume that we already know the true value of the target variable of these past test cases and, moreover, that we can safely use this information. So, if at some time m of the testing period we are confident that there is a regime shift in the time series, then we can incorporate the information of all test cases occurring before m into the initial training data, and with this refreshed training set that contains observations of the "new" regime, somehow update our predictive model to improve the performance on future test cases. One form of updating the model could be to change it in order to take into account the new training cases. These approaches are usually known as incremental learners as they adapt the current model to new evidence instead of starting from scratch. There are not so many modeling techniques that can be used in this way, particularly in R. In this context, we will follow the other approach to the updating problem, which consists of re-learning a new model with the new updated training set. This is obviously more expensive in computational terms and may even be inadequate for applications where the data arrives at a very fast pace and for which models and decisions are required almost in real-time. This is rather frequent in applications addressed in a research area usually known as data streams. In our application, we are making decisions on a daily basis after

the market closes, so speed is not a key issue.[21] Assuming that we will use a re-learn approach, we have essentially two forms of incorporating the new cases into our training set. The growing window approach simply adds them to the current training set, thus constantly increasing the size of this set. The eventual problem of this approach lies in the fact that as we are assuming that more recent data is going to be helpful in producing better models, we may also consider whether the oldest part of our training data may already be too outdated and in effect, contributing to decreasing the accuracy of the models. Based on these considerations, the sliding window approach deletes the oldest data of the training set at the same time it incorporates the fresher observations, thus maintaining a training set of constant size.

Both the growing and the sliding window approaches involve a key decision: when to change or adapt the model by incorporating fresher data. There are essentially two ways of answering this question. The first involves estimating this time by checking if the performance of our current model is starting to degrade. If we observe a sudden decrease in this performance, then we can take this as a good indication of some form of regime shift. The main challenge of these approaches lies in developing proper estimates of these changes in performance. We want to detect the change as soon as possible but we do not want to overreact to some spurious test case that our model missed. Another simpler approach consists of updating the model on a regular time basis, that is, every w test case, we obtain a new model with fresher data. In this case study we follow this simpler method.

Summarizing, for each model that we will consider, we will apply it using three different approaches: (1) single model for all test period, (2) growing window with a fixed updating step of w days, and (3) sliding window with the same updating step w. Figure 3.3 illustrates the three approaches.

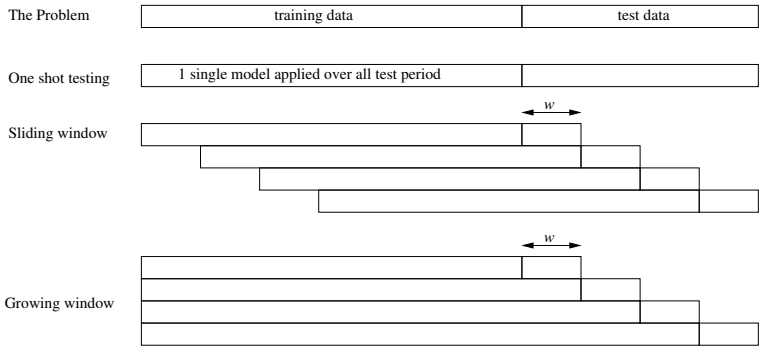

FIGURE 3.3: Three forms of obtaining predictions for a test period.

[21] It could be if we were trading in real-time, that is, intra-day trading.

Further readings on regime changes

The problem of detecting changes of regime in time series data is a subject studied for a long time in an area known as statistic process control (e.g., Oakland, 2007), which use techniques like control charts to detect break points in the data. This subject has been witnessing an increased interest with the impact of data streams (e.g., Gama and Gaber, 2007) in the data mining field. Several works (e.g., Gama et al., 2004; Kifer et al., 2004; Klinkenberg, 2004) have addressed the issues of how to detect the changes of regime and also how to learn models in the presence of these changes.

3.4.2 The Modeling Tools

In this section we briefly describe the modeling techniques we will use to address our prediction tasks and illustrate how to use them in R.

3.4.2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are frequently used in financial forecasting (e.g., Deboeck, 1994) because of their ability to deal with highly nonlinear problems. The package `nnet` implements feed-forward neural nets in R. This type of neural networks is among the most used and also what we will be applying.

ANNs are formed by a set of computing units (the neurons) linked to each other. Each neuron executes two consecutive calculations: a linear combination of its inputs, followed by a nonlinear computation of the result to obtain its output value that is then fed to other neurons in the network. Each of the neuron connections has an associated weight. Constructing an artificial neural network consists of establishing an architecture for the network and then using an algorithm to find the weights of the connections between the neurons.

Feed-forward artificial neural networks have their neurons organized in layers. The first layer contains the input neurons of the network. The training observations of the problem are presented to the network through these input neurons. The final layer contains the predictions of the neural network for any case presented at its input neurons. In between, we usually have one or more "hidden" layers of neurons. The weight updating algorithms, such as the back-propagation method, try to obtain the connection weights that optimize a certain error criterion, that is, trying to ensure that the network outputs are in accordance with the cases presented to the model. This is accomplished by an iterative process of presenting several times the training cases at the input nodes of the network, and after obtaining the prediction of the network at the output nodes and calculating the respective prediction error, updating the weights in the network to try to improve its prediction error. This iterative process is repeated until some convergence criterion is met.

Feed-forward ANNs with one hidden layer can be easily obtained in R using a function of the package `nnet` (Venables and Ripley, 2002). The networks obtained by this function can be used for both classification and regression problems and thus are applicable to both our prediction tasks (see Section 3.3.3).

ANNs are known to be sensitive to different scales of the variables used in a prediction problem. In this context, it makes sense to transform the data before giving them to the network, in order to avoid eventual negative impacts on the performance. In our case we will normalize the data with the goal of making all variables have a mean value of zero and a standard deviation of one. This can be easily accomplished by the following transformation applied to each column of our data set:

$$y_i = \frac{x_i - \bar{x}}{\sigma_x} \tag{3.14}$$

where $\bar{x}$ is the mean value of the original variable X, and σ_x its standard deviation.

The function `scale()` can be used to carry out this transformation for our data. In the book package you can also find the function `unscale()` that inverts the normalization process putting the values back on the original scale. Below you can find a very simple illustration of how to obtain and use this type of ANN in R:

```
> set.seed(1234)
> library(nnet)
> norm.data <- scale(Tdata.train)
> nn <- nnet(Tform, norm.data[1:1000, ], size = 10, decay = 0.01,
+     maxit = 1000, linout = T, trace = F)
> norm.preds <- predict(nn, norm.data[1001:2000, ])
> preds <- unscale(norm.preds, norm.data)
```

By default, the function `nnet()` sets the initial weights of the links between nodes with random values in the interval $[-0.5 \cdots 0.5]$. This means that two successive runs of the function with exactly the same arguments can actually lead to different solutions. To ensure you get the same results as we present below, we have added a call to the function `set.seed()` that initializes the random number generator to some seed number. This ensures that you will get exactly the same ANN as the one we report here. In this illustrative example we have used the first 1,000 cases to obtain the network and tested the model on the following 1,000. After normalizing our training data, we call the function `nnet()` to obtain the model. The first two parameters are the usual of any modeling function in R: the functional form of the model specified by a formula, and the training sample used to obtain the model. We have also used some of the parameters of the `nnet()` function. Namely, the parameter `size` allows us to specify how many nodes the hidden layer will have. There is no magic recipe on which value to use here. One usually tries several values to observe the network behavior. Still, it is reasonable to assume it should be smaller than the number of predictors of the problem. The parameter `decay` controls the weight updating rate of the back-propagation algorithm. Again, trial and error is your best friend here. Finally, the parameter `maxit` controls the maximum number of iterations the weight convergence process is allowed

to use, while the `linout=T` setting tells the function that we are handling a regression problem. The `trace=F` is used to avoid some of the output of the function regarding the optimization process.

The function `predict()` can be used to obtain the predictions of the neural network for a set of test data. After obtaining these predictions, we convert them back to the original scale using the function `unscale()` provided by our package. This function receives in the first argument the values, and on the second argument the object with the normalized data. This latter object is necessary because it is within that object that the averages and standard deviations that were used to normalize the data are stored,[22] and these are required to invert the normalization.

Let us evaluate the results of the ANN for predicting the correct signals for the test set. We do this by transforming the numeric predictions into signals and then evaluate them using the statistics presented in Section 3.3.4.

```
> sigs.nn <- trading.signals(preds, 0.1, -0.1)
> true.sigs <- trading.signals(Tdata.train[1001:2000, "T.ind.GSPC"],
+     0.1, -0.1)
> sigs.PR(sigs.nn, true.sigs)

    precision    recall
s   0.2101911 0.1885714
b   0.2919255 0.5911950
s+b 0.2651357 0.3802395
```

Function `trading.signals()` transforms numeric predictions into signals, given the buy and sell thresholds, respectively. The function `sigs.PR()` obtains a matrix with the precision and recall scores of the two types of events, and overall. These scores show that the performance of the ANN is not brilliant. In effect, you get rather low precision scores, and also not so interesting recall values. The latter are not so serious as they basically mean lost opportunities and not costs. On the contrary, low precision scores mean that the model gave wrong signals rather frequently. If these signals are used for trading, this may lead to serious losses of money.

ANNs can also be used for classification tasks. For these problems the main difference in terms of network topology is that instead of a single output unit, we will have as many output units as there are values of the target variable (sometimes known as the class variable). Each of these output units will produce a probability estimate of the respective class value. This means that for each test case, an ANN can produce a set of probability values, one for each possible class value.

The use of the `nnet()` function for these tasks is very similar to its use for regression problems. The following code illustrates this, using our training data:

[22] As object attributes.

```
> set.seed(1234)
> library(nnet)
> signals <- trading.signals(Tdata.train[, "T.ind.GSPC"], 0.1,
+     -0.1)
> norm.data <- data.frame(signals = signals, scale(Tdata.train[,
+     -1]))
> nn <- nnet(signals ~ ., norm.data[1:1000, ], size = 10, decay = 0.01,
+     maxit = 1000, trace = F)
> preds <- predict(nn, norm.data[1001:2000, ], type = "class")
```

The `type="class"` argument is used to obtain a single class label for each test case instead of a set of probability estimates. With the network predictions we can calculate the model precision and recall as follows:

```
> sigs.PR(preds, norm.data[1001:2000, 1])

    precision    recall
s   0.2838710 0.2514286
b   0.3333333 0.2264151
s+b 0.2775665 0.2185629
```

Both the precision and recall scores are higher than the ones obtained in the regression task, although still low values.

Further readings on neural networks

The book by Rojas (1996) is a reasonable general reference on neural networks. For more financially oriented readings, the book by Zirilli (1997) is a good and easy reading book. The collection of papers entitled "Artificial Neural Networks Forecasting Time Series" (Rogers and Vemuri, 1994) is another example of a good source of references. Part I of the book by Deboeck (1994) provides several chapters devoted to the application of neural networks to trading. The work of McCulloch and Pitts (1943) presents the first model of an artificial neuron. This work was generalized by Ronsenblatt (1958) and Minsky and Papert (1969). The back-propagation algorithm, the most frequently used weight updating method, although frequently attributed to Rumelhart et al. (1986), was, according to Rojas (1996), invented by Werbos (1974, 1996).

3.4.2.2 Support Vector Machines

Support vector machines (SMVs)[23] are modeling tools that, as ANNs, can be applied to both regression and classification tasks. SVMs have been witnessing increased attention from different research communities based on their successful application to several domains and also their strong theoretical background. Vapnik (1995, 1998) and Shawe-Taylor and Cristianini (2000) are two of the essential references for SVMs. Smola and Schölkopf (2004, 1998) published an excellent tutorial giving an overview of the basic ideas underlying SVMs for regression. In R we have several implementations of SMVs available, among which we can refer to the package `kernlab` by Karatzoglou

[23] Extensive information on this class of models can be obtained at http://www.kernel-machines.org.

et al. (2004) with several functionalities available, and also the function `svm()` on package `e1071` by Dimitriadou et al. (2009).

The basic idea behind SVMs is that of mapping the original data into a new, high-dimensional space, where it is possible to apply linear models to obtain a separating hyper plane, for example, separating the classes of the problem, in the case of classification tasks. The mapping of the original data into this new space is carried out with the help of the so-called kernel functions. SMVs are linear machines operating on this dual representation induced by kernel functions.

The hyper plane separation in the new dual representation is frequently done by maximizing a separation margin between cases belonging to different classes; see Figure 3.4. This is an optimization problem often solved with quadratic programming methods. Soft margin methods allow for a small proportion of cases to be on the "wrong" side of the margin, each of these leading to a certain "cost".

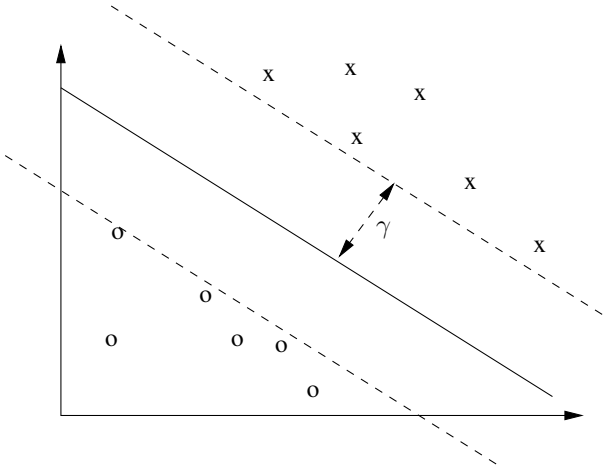

FIGURE 3.4: The margin maximization in SVMs.

In support of vector regression, the process is similar, with the main difference being on the form the errors and associated costs are calculated. This resorts usually to the use of the so-called ϵ-*insensitive* loss function $|\ \xi\ |_\epsilon$ given by

$$|\ \xi\ |_\epsilon = \begin{cases} 0 & \text{if } |\ \xi\ | \leq \epsilon \\ |\ \xi\ | - \epsilon & \text{otherwise} \end{cases} \tag{3.15}$$

We will now provide very simple examples of the use of this type of models in R. We start with the regression task for which we will use the function provided in the package `e1071`:

```
> library(e1071)
```

```
> sv <- svm(Tform, Tdata.train[1:1000, ], gamma = 0.001, cost = 100)
> s.preds <- predict(sv, Tdata.train[1001:2000, ])
> sigs.svm <- trading.signals(s.preds, 0.1, -0.1)
> true.sigs <- trading.signals(Tdata.train[1001:2000, "T.ind.GSPC"],
+     0.1, -0.1)
> sigs.PR(sigs.svm, true.sigs)

    precision     recall
s   0.4285714 0.03428571
b   0.3333333 0.01257862
s+b 0.4000000 0.02395210
```

In this example we have used the `svm()` function with most of its default parameters with the exception of the parameters `gamma` and `cost`. In this context, the function uses a radial basis kernel function

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\gamma \times \|\mathbf{x} - \mathbf{y}\|^2\right) \tag{3.16}$$

where γ is a user parameter that in our call we have set to 0.001 (function `svm()` uses as default `1/ncol(data)`).

The parameter `cost` indicates the cost of the violations of the margin. You may wish to explore the help page of the function to learn more details on these and other parameters.

As we can observe, the SVM model achieves a considerably better score than the ANN in terms of precision, although with a much lower recall.

Next, we consider the classification task, this time using the `kernlab` package:

```
> library(kernlab)
> data <- cbind(signals = signals, Tdata.train[, -1])
> ksv <- ksvm(signals ~ ., data[1:1000, ], C = 10)

Using automatic sigma estimation (sigest) for RBF or laplace kernel

> ks.preds <- predict(ksv, data[1001:2000, ])
> sigs.PR(ks.preds, data[1001:2000, 1])

    precision    recall
s   0.1935484 0.2742857
b   0.2688172 0.1572327
s+b 0.2140762 0.2185629
```

We have used the `C` parameter of the `ksvm()` function of package `kernlab`, to specify a different cost of constraints violations, which by default is 1. Apart from this we have used the default parameter values, which for classification involves, for instance, using the radial basis kernel. Once again, more details can be obtained in the help pages of the `ksvm()` function.

The results of this SVM are not as interesting as the SVM obtained with the regression data. We should remark that by no means do we want to claim

that these are the best scores we can obtain with these techniques. These are just simple illustrative examples of how to use these modeling techniques in R.

3.4.2.3 Multivariate Adaptive Regression Splines

Multivariate adaptive regression splines (Friedman, 1991) are an example of an additive regression model (Hastie and Tibshirani, 1990). A MARS model has the following general form:

$$mars\,(\mathbf{x}) = c_0 + \sum_{i=1}^{k} c_i B_i(\mathbf{x}) \tag{3.17}$$

where the c_is are constants and the B_is are basis functions.

The basis functions can take several forms, from simple constants to functions modeling the interaction between two or more variables. Still, the most common basis functions are the so-called *hinge* functions that have the form

$$H[-(x_i - t)] = \max(0, t - x_i) \qquad H[+(x_i - t)] = \max(0, x_i - t)$$

where x_i is a predictor and t a threshold value on this predictor. Figure 3.5 shows an example of two of these functions.

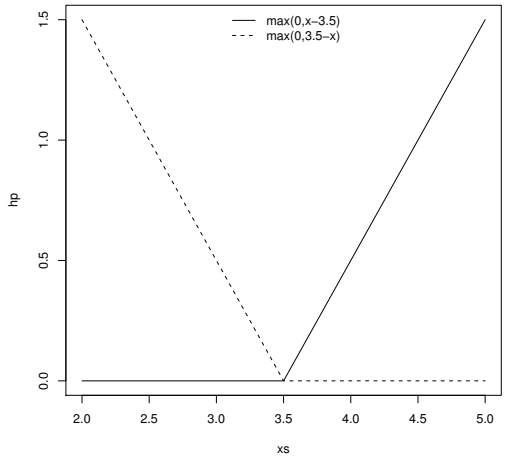

FIGURE 3.5: An example of two hinge functions with the same threshold.

MARS models have been implemented in at least two packages within R. Package `mda` (Leisch et al., 2009) contains the function `mars()` that implements this method. Package `earth` (Milborrow, 2009) has the function `earth()` that also implements this methodology. This latter function has the

advantage of following a more standard R schema in terms of modeling functions, by providing a formula-based interface. It also implements several other facilities not present in the other package and thus it will be our selection.

The following code applies the function `earth()` to the regression task

```
> library(earth)
> e <- earth(Tform, Tdata.train[1:1000, ])
> e.preds <- predict(e, Tdata.train[1001:2000, ])
> sigs.e <- trading.signals(e.preds, 0.1, -0.1)
> true.sigs <- trading.signals(Tdata.train[1001:2000, "T.ind.GSPC"],
+     0.1, -0.1)
> sigs.PR(sigs.e, true.sigs)

    precision    recall
s   0.2785714 0.2228571
b   0.4029851 0.1698113
s+b 0.3188406 0.1976048
```

The results are comparable to the ones obtained with SVMs for classification, with precision scores around 30%, although with lower recall.

MARS is only applicable to regression problems so we do not show any example for the classification task.

Further readings on multivariate adaptive regression splines

The definitive reference on MARS is the original journal article by Friedman (1991). This is a very well-written article providing all details concerning the motivation for the development of MARS as well as the techniques used in the system. The article also includes quite an interesting discussion section by other scientists that provides other views of this work.

3.5 From Predictions into Actions

This section will address the issue of how will we use the signal predictions obtained with the modeling techniques described previously. Given a set of signals output by some model there are many ways we can use them to act on the market.

3.5.1 How Will the Predictions Be Used?

In our case study we will assume we will be trading in future markets. These markets are based on contracts to buy or sell a commodity on a certain date in the future at the price determined by the market at that future time. The technical details of these contracts are beyond the scope of this manuscript. Still, in objective terms, this means that our trading system will be able to

open two types of trading positions: long and short. Long positions are opened by buying a commodity at time t and price p, and selling it at a later time $t + x$. It makes sense for the trader to open such positions when he has the expectation that the price will rise in the future, thus allowing him to make some profit with that transaction. On short positions, the trader sells the security at time t with price p with the obligation of buying it in the future. This is possible thanks to a borrowing schema whose details you can find in appropriate documents (e.g., Wikipedia). These types of positions allows the trader to make profit when the prices decline as he/she will buy the security at a time later than t. Informally, we can say that we will open short positions when we believe the prices are going down, and open long positions when we believe the prices are going up.

Given a set of signals, there are many ways we can use them to trade in future markets. We will describe a few plausible trading strategies that we will be using and comparing in our experiments with the models. Due to space and time constraints, it is not possible to explore this important issue further. Still, the reader is left with some plausible strategies and with the means to develop and try other possibilities.

The mechanics of the first trading strategy we are going to use are the following. First, all decisions will be taken at the end of the day, that is, after knowing all daily quotes of the current session. Suppose that at the end of day t, our models provide evidence that the prices are going down, that is, predicting a low value of T or a sell signal. If we already have a position opened, the indication of the model will be ignored. If we currently do not hold any opened position, we will open a short position by issuing a sell order. When this order is carried out by the market at a price pr sometime in the future, we will immediately post two other orders. The first is a buy limit order with a limit price of $pr - p\%$, where $p\%$ is a target profit margin. This type of order is carried out only if the market price reaches the target limit price or below. This order expresses what our target profit is for the short position just opened. We will wait 10 days for this target to be reached. If the order is not carried out by this deadline, we will buy at the closing price of the 10th day. The second order is a buy stop order with a price limit $pr + l\%$. This order is placed with the goal of limiting our eventual losses with this position. The order will be executed if the market reaches the price $pr + l\%$, thus limiting our possible losses to $l\%$.

If our models provide indications that the prices will rise in the near future, with high predicted T values or buy signals, we will consider opening a long position. This position will only be opened if we are currently out of the market. With this purpose we will post a buy order that will be accomplished at a time t and price pr. As before, we will immediately post two new orders. The first will be a sell limit order with a target price of $pr + p\%$, which will only be executed if the market reaches a price of $pr + p\%$ or above. This sell limit order will have a deadline of 10 days, as before.The second order is a sell stop order with price $pr - l\%$, which will again limit our eventual losses to $l\%$.

This first strategy can be seen as a bit conservative as it will only have a single position opened at any time. Moreover, after 10 days of waiting for the target profit, the positions are immediately closed. We will also consider a more "risky" trading strategy. This other strategy is similar to the previous one, with the exception that we will always open new positions if there are signals with that indication, and if we have sufficient money for that. Moreover, we will wait forever for the positions to reach either the target profit or the maximum allowed loss.

We will only consider these two main trading strategies with slight variations on the used parameters (e.g., holding time, expected profit margin, or amount of money invested on each position). As mentioned, these are simply chosen for illustrative purposes.

3.5.2 Trading-Related Evaluation Criteria

The metrics described in Section 3.3.4 do not translate directly to the overall goal of this application, which has to do with economic performance. Factors like the economic results and the risk exposure of some financial instrument or tool are of key importance in this context. This is an area that alone could easily fill this chapter. The R package `PerformanceAnalytics` (Carl and Peterson, 2009) implements many of the existing financial metrics for analyzing the returns of some trading algorithm as the one we are proposing in this chapter. We will use some of the functions provided by this package to collect information on the economic performance of our proposals. Our evaluation will be focused on the overall results of the methods, on their risk exposure, and on the average results of each position hold by the models. In the final evaluation of our proposed system to be described in Section 3.7, we will carry out a more in-depth analysis of its performance using tools provided by this package.

With respect to the overall results, we will use (1) the simple net balance between the initial capital and the capital at the end of the testing period (sometimes called the profit/loss), (2) the percentage return that this net balance represents, and (3) the excess return over the buy and hold strategy. This strategy consists of opening a long position at the beginning of the testing period and waiting until the end to close it. The return over the buy and hold measures the difference between the return of our trading strategy and this simple strategy.

Regarding risk-related measures, we will use the Sharpe ratio coefficient, which measures the return per unit of risk, the latter being measured as the standard deviation of the returns. We will also calculate the maximum draw-down, which measures the maximum cumulative successive loss of a model. This is an important risk measure for traders, as any system that goes over a serious draw-down is probably doomed to be without money to run, as investors will most surely be scared by these successive losses and redraw their money.

Finally, the performance of the positions hold during the test period will be evaluated by their number, the average return per position, and the percentage of profitable positions, as well as other less relevant metrics.

3.5.3 Putting Everything Together: A Simulated Trader

This section describes how to implement the ideas we have sketched regarding trading with the signals of our models. Our book package provides the function `trading.simulator()`, which can be used to put all these ideas together by carrying out a trading simulation with the signals of any model. The main parameters of this function are the market quotes for the simulation period and the model signals for this period. Two other parameters are the name of the user-defined trading policy function and its list of parameters. Finally, we can also specify the cost of each transaction and the initial capital available for the trader. The simulator will call the user-provided trading policy function at the end of each daily section, and the function should return the orders that it wants the simulator to carry out. The simulator carries out these orders on the market and records all activity on several data structures. The result of the simulator is an object of class `tradeRecord` containing the information of this simulation. This object can then be used in other functions to obtain economic evaluation metrics or graphs of the trading activity, as we will see.

Before proceeding with an example of this type of simulation, we need to provide further details on the trading policy functions that the user needs to supply to the simulator. These functions should be written using a certain protocol, that is, they should be aware of how the simulator will call them, and should return the information this simulator is expecting.

At the end of each daily session d, the simulator calls the trading policy function with four main arguments plus any other parameters the user has provided in the call to the simulator. These four arguments are (1) a vector with the predicted signals until day d, (2) the market quotes (up to d), (3) the currently opened positions, and (4) the money currently available to the trader. The current position is a matrix with as many rows as there are open positions at the end of day d. This matrix has four columns: "pos.type" that can be 1 for a long position or -1 for a short position; "N.stocks", which is the number of stocks of the position; "Odate", which is the day on which the position was opened (a number between 1 and d); and "Oprice", which is the price at which the position was opened. The row names of this matrix contain the IDs of the positions that are relevant when we want to indicate the simulator that a certain position is to be closed.

All this information is provided by the simulator to ensure the user can define a broad set of trading policy functions. The user-defined functions should return a data frame with a set of orders that the simulator should carry out. This data frame should include the following information (columns): "order", which should be 1 for buy orders and -1 for sell orders; "order.type", which should be 1 for market orders that are to be carried out immediately (ac-

tually at next day open price), 2 for limit orders or 3 for stop orders; "val", which should be the quantity of stocks to trade for opening market orders, NA for closing market orders, or a target price for limit and stop orders; "action", which should be "open" for orders that are opening a new position or "close" for orders closing an existing position; and finally, "posID", which should contain the ID of the position that is being closed, if applicable.

The following is an illustration of a user-defined trading policy function:

```
> policy.1 <- function(signals,market,opened.pos,money,
+                      bet=0.2,hold.time=10,
+                      exp.prof=0.025, max.loss= 0.05
+                      )
+   {
+     d <- NROW(market) # this is the ID of today
+     orders <- NULL
+     nOs <- NROW(opened.pos)
+     # nothing to do!
+     if (!nOs && signals[d] == 'h') return(orders)
+
+     # First lets check if we can open new positions
+     # i) long positions
+     if (signals[d] == 'b' && !nOs) {
+       quant <- round(bet*money/market[d,'Close'],0)
+       if (quant > 0)
+         orders <- rbind(orders,
+               data.frame(order=c(1,-1,-1),order.type=c(1,2,3),
+                          val = c(quant,
+                                  market[d,'Close']*(1+exp.prof),
+                                  market[d,'Close']*(1-max.loss)
+                                 ),
+                          action = c('open','close','close'),
+                          posID = c(NA,NA,NA)
+                         )
+                        )
+
+     # ii) short positions
+     } else if (signals[d] == 's' && !nOs) {
+       # this is the nr of stocks we already need to buy
+       # because of currently opened short positions
+       need2buy <- sum(opened.pos[opened.pos[,'pos.type']==-1,
+                                  "N.stocks"])*market[d,'Close']
+       quant <- round(bet*(money-need2buy)/market[d,'Close'],0)
+       if (quant > 0)
+         orders <- rbind(orders,
+               data.frame(order=c(-1,1,1),order.type=c(1,2,3),
+                          val = c(quant,
+                                  market[d,'Close']*(1-exp.prof),
+                                  market[d,'Close']*(1+max.loss)
+                                 ),
```

```
+                          action = c('open','close','close'),
+                          posID = c(NA,NA,NA)
+                         )
+                        )
+     }
+
+     # Now lets check if we need to close positions
+     # because their holding time is over
+     if (nOs)
+       for(i in 1:nOs) {
+         if (d - opened.pos[i,'Odate'] >= hold.time)
+           orders <- rbind(orders,
+                 data.frame(order=-opened.pos[i,'pos.type'],
+                            order.type=1,
+                            val = NA,
+                            action = 'close',
+                            posID = rownames(opened.pos)[i]
+                           )
+                          )
+       }
+
+     orders
+   }
```

This `policy.1()` function implements the first trading strategy we described in Section 3.5.1. The function has four parameters that we can use to tune this strategy. These are the `bet` parameter, which specifies the percentage of our current money, that we will invest each time we open a new position; the `exp.prof` parameter, which indicates the profit margin we wish for our positions and is used when posting the limit orders; the `max.loss`, which indicates the maximum loss we are willing to admit before we close the position, and is used in stop orders; and the `hold.time` parameter, which indicates the number of days we are willing to wait to reach the profit margin. If the holding time is reached without achieving the wanted margin, the positions are closed.

Notice that whenever we open a new position, we send three orders back to the simulator: a market order to open the position, a limit order to specify our target profit margin, and a stop order to limit our losses.

Equivalently, the following function implements our second trading strategy:

```
> policy.2 <- function(signals,market,opened.pos,money,
+                      bet=0.2,exp.prof=0.025, max.loss= 0.05
+                     )
+   {
+     d <- NROW(market) # this is the ID of today
+     orders <- NULL
+     nOs <- NROW(opened.pos)
```

```
+     # nothing to do!
+     if (!nOs && signals[d] == 'h') return(orders)
+
+     # First lets check if we can open new positions
+     # i) long positions
+     if (signals[d] == 'b') {
+       quant <- round(bet*money/market[d,'Close'],0)
+       if (quant > 0)
+         orders <- rbind(orders,
+               data.frame(order=c(1,-1,-1),order.type=c(1,2,3),
+                          val = c(quant,
+                                  market[d,'Close']*(1+exp.prof),
+                                  market[d,'Close']*(1-max.loss)
+                                 ),
+                          action = c('open','close','close'),
+                          posID = c(NA,NA,NA)
+                         )
+                        )
+
+     # ii) short positions
+     } else if (signals[d] == 's') {
+       # this is the money already committed to buy stocks
+       # because of currently opened short positions
+       need2buy <- sum(opened.pos[opened.pos[,'pos.type']==-1,
+                                  "N.stocks"])*market[d,'Close']
+       quant <- round(bet*(money-need2buy)/market[d,'Close'],0)
+       if (quant > 0)
+         orders <- rbind(orders,
+               data.frame(order=c(-1,1,1),order.type=c(1,2,3),
+                          val = c(quant,
+                                  market[d,'Close']*(1-exp.prof),
+                                  market[d,'Close']*(1+max.loss)
+                                 ),
+                          action = c('open','close','close'),
+                          posID = c(NA,NA,NA)
+                         )
+                        )
+     }
+
+     orders
+   }
```

This function is very similar to the previous one. The main difference lies in the fact that in this trading policy we allow for more than one position to be opened at the same time, and also there is no aging limit for closing the positions.

Having defined the trading policy functions, we are ready to try our trading simulator. For illustration purposes we will select a small sample of our data to obtain an SVM, which is then used to obtain predictions for a subsequent

period. We call our trading simulator with these predictions to obtain the results of trading using the signals of the SVM in the context of a certain trading policy.

```
> # Train and test periods
> start <- 1
> len.tr <- 1000
> len.ts <- 500
> tr <- start:(start+len.tr-1)
> ts <- (start+len.tr):(start+len.tr+len.ts-1)
> # getting the quotes for the testing period
> data(GSPC)
> date <- rownames(Tdata.train[start+len.tr,])
> market <- GSPC[paste(date,'/',sep='')][1:len.ts]
> # learning the model and obtaining its signal predictions
> library(e1071)
> s <- svm(Tform,Tdata.train[tr,],cost=10,gamma=0.01)
> p <- predict(s,Tdata.train[ts,])
> sig <- trading.signals(p,0.1,-0.1)
> # now using the simulated trader
> t1 <- trading.simulator(market,sig,
+              'policy.1',list(exp.prof=0.05,bet=0.2,hold.time=30))
```

Please note that for this code to work, you have to previously create the objects with the data for modeling, using the instructions given in Section 3.3.3.

In our call to the trading simulator we have selected the first trading policy and have provided some different values for some of its parameters. We have used the default values for transaction costs (five monetary units) and for the initial capital (1 million monetary units). The result of the call is an object of class `tradeRecord`. We can check its contents as follows:

```
> t1

Object of class tradeRecord with slots:

         trading: <xts object with a numeric  500 x 5  matrix>
         positions: <numeric  16 x 7  matrix>
         init.cap :  1e+06
         trans.cost :  5
         policy.func :  policy.1
         policy.pars : <list with  3  elements>

> summary(t1)

== Summary of a Trading Simulation with  500  days ==

Trading policy function :  policy.1
Policy function parameters:
         exp.prof  =  0.05
```

```
        bet  =  0.2
        hold.time  =  30

Transaction costs :  5
Initial Equity    :  1e+06
Final Equity      :  997211.9   Return :  -0.28 %
Number of trading positions:  16

Use function "tradingEvaluation()" for further stats on this simulation.
```

The function `tradingEvaluation()` can be used to obtain a series of economic indicators of the performance during this simulation period:

```
> tradingEvaluation(t1)

    NTrades       NProf    PercProf          PL         Ret   RetOverBH
      16.00        8.00       50.00    -2788.09       -0.28       -7.13
      MaxDD SharpeRatio     AvgProf     AvgLoss       AvgPL     MaxProf
   59693.15        0.00        4.97       -4.91        0.03        5.26
    MaxLoss
      -5.00
```

We can also obtain a graphical overview of the performance of the trader using the function `plot()` as follows:

```
> plot(t1, market, theme = "white", name = "SP500")
```

The result of this command is shown on Figure 3.6.

The results of this trader are bad, with a negative return. Would the scenario be different if we used the second trading policy? Let us see:

```
> t2 <- trading.simulator(market, sig, "policy.2", list(exp.prof = 0.05,
+     bet = 0.3))
> summary(t2)

== Summary of a Trading Simulation with  500  days ==

Trading policy function :  policy.2
Policy function parameters:
        exp.prof  =  0.05
        bet  =  0.3

Transaction costs :  5
Initial Equity    :  1e+06
Final Equity      :  961552.5   Return :  -3.84 %
Number of trading positions:  29

Use function "tradingEvaluation()" for further stats on this simulation.

> tradingEvaluation(t2)
```

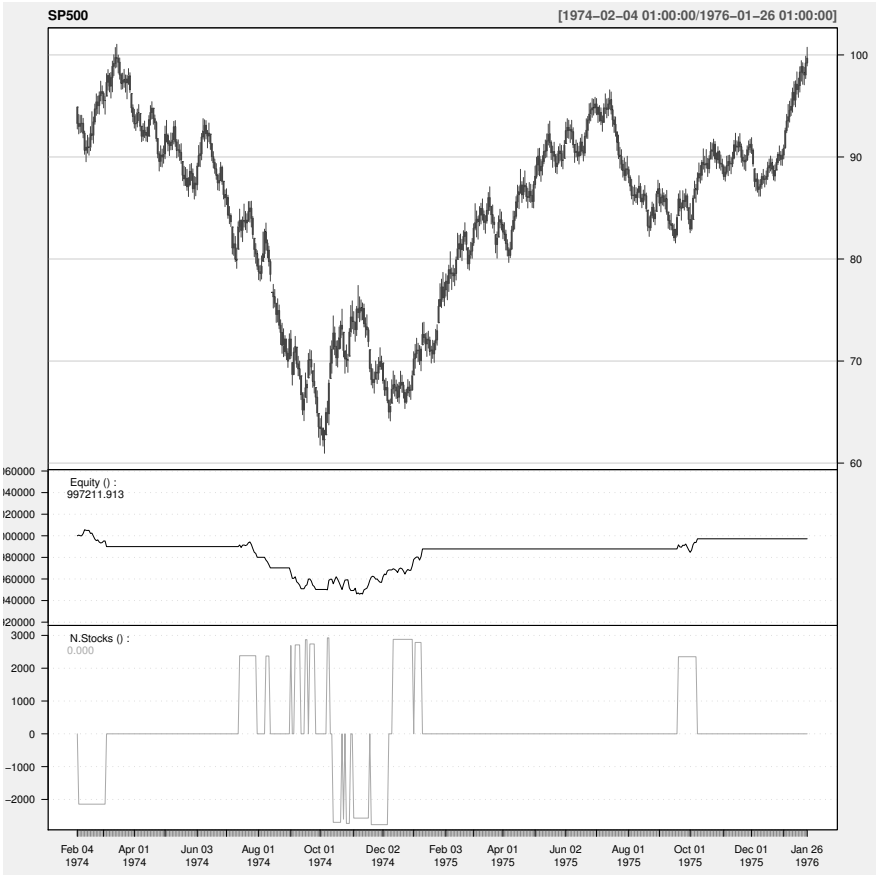

FIGURE 3.6: The results of trading using Policy 1 based on the signals of an SVM.

```
    NTrades       NProf    PercProf          PL         Ret   RetOverBH
      29.00       14.00       48.28   -38447.49       -3.84      -10.69
      MaxDD SharpeRatio     AvgProf     AvgLoss       AvgPL     MaxProf
  156535.05       -0.02        4.99       -4.84       -0.10        5.26
    MaxLoss
      -5.00
```

Using the same signals but with a different trading policy the return decreased from -0.27% to -2.86%. Let us repeat the experiment with a different training and testing period:

```
> start <- 2000
> len.tr <- 1000
> len.ts <- 500
> tr <- start:(start + len.tr - 1)
> ts <- (start + len.tr):(start + len.tr + len.ts - 1)
> s <- svm(Tform, Tdata.train[tr, ], cost = 10, gamma = 0.01)
> p <- predict(s, Tdata.train[ts, ])
> sig <- trading.signals(p, 0.1, -0.1)
> t2 <- trading.simulator(market, sig, "policy.2", list(exp.prof = 0.05,
+     bet = 0.3))
> summary(t2)

== Summary of a Trading Simulation with  500  days ==

Trading policy function :  policy.2
Policy function parameters:
         exp.prof  =  0.05
         bet  =  0.3

Transaction costs :  5
Initial Equity    :  1e+06
Final Equity      :  107376.3   Return :  -89.26 %
Number of trading positions:  229

Use function "tradingEvaluation()" for further stats on this simulation.

> tradingEvaluation(t2)

    NTrades       NProf    PercProf          PL         Ret   RetOverBH
     229.00       67.00       29.26  -892623.73      -89.26      -96.11
      MaxDD SharpeRatio     AvgProf     AvgLoss       AvgPL     MaxProf
  959624.80       -0.08        5.26       -4.50       -1.65        5.26
    MaxLoss
      -5.90
```

This trader, obtained by the same modeling technique and using the same trading strategy, obtained a considerable worse result. The major lesson to be learned here is: reliable statistical estimates. Do not be fooled by a few repetitions of some experiments, even if it includes a 2-year testing period.

We need more repetitions under different conditions to ensure some statistical reliability of our results. This is particularly true for time series models that have to handle different regimes (e.g., periods with rather different volatility or trend). This is the topic of the next section.

3.6 Model Evaluation and Selection

In this section we will consider how to obtain reliable estimates of the selected evaluation criteria. These estimates will allow us to properly compare and select among different alternative trading systems.

3.6.1 Monte Carlo Estimates

Time series problems like the one we are addressing bring new challenges in terms of obtaining reliable estimates of our evaluation metrics. This is caused by the fact that all data observations have an attached time tag that imposes an ordering among them. This ordering should be respected with the risk of obtaining estimates that are not reliable. In Chapter 2 we used the cross-validation method to obtain reliable estimates of evaluation statistics. This methodology includes a random re-sampling step that changes the original ordering of the observations. This means that cross-validation should not be applied to time series problems. Applying this method could mean to test models on observations that are older than the ones used to obtain them. This is not feasible in reality, and thus the estimates obtained by this process are unreliable and possibly overly optimistic, as it is easier to predict the past given the future than the opposite.

Any estimation process using time series data should ensure that the models are always tested on data that is more recent than the data used to obtain the models. This means no random re-sampling of the observations or any other process that changes the time ordering of the given data. However, any proper estimation process should include some random choices to ensure the statistical reliability of the obtained estimates. This involves repeating the estimation process several times under different conditions, preferably randomly selected. Given a time series dataset spanning from time t to time $t + N$, how can we ensure this? First, we have to choose the train+test setup for which we want to obtain estimates. This means deciding what is the size of both the train and test sets to be used in the estimation process. The sum of these two sizes should be smaller than N to ensure that we are able to randomly generate different experimental scenarios with the data that was provided to us. However, if we select a too small training size, we may seriously impair the performance of our models. Similarly, small test sets will also be less reliable,

particularly if we suspect there are regime shifts in our problem and we wish to test the models under these circumstances.

Our dataset includes roughly 30 years of daily quotes. We will evaluate all alternatives by estimating their performance on a test set of 5 years of quotes, when given 10 years of training data. This ensures train and test sizes that are sufficiently large; and, moreover, it leaves space for different repetitions of this testing process as we have 30 years of data.

In terms of experimental methodology, we will use a Monte Carlo experiment to obtain reliable estimates of our evaluation metrics. Monte Carlo methods rely on random sampling to obtain their results. We are going to use this sampling process to choose a set of R points in our 30-year period of quotes. For each randomly selected time point r, we will use the previous 10 years of quotes to obtain the models and the subsequent 5 years to test them. At the end of these R iterations we will have R estimates for each of our evaluation metrics. Each of these estimates is obtained on a randomly selected window of 15 years of data, the first 10 years used for training and the remaining 5 years for testing. This ensures that our experiments always respect the time ordering of the time series data. Repeating the process R times will ensure sufficient variability on the train+test conditions, which increases the reliability of our estimates. Moreover, if we use the same set of R randomly selected points for evaluating different alternatives, we can carry out paired comparisons to obtain statistical confidence levels on the observed differences of mean performance. Figure 3.7 summarizes the Monte Carlo experimental method. Notice that as we have to ensure that for every random point r there are 10 years of data before and 5 years after, this eliminates some of the data from the random selection of the R points.

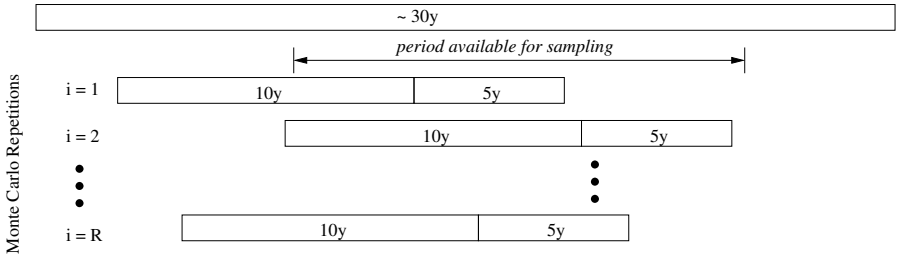

FIGURE 3.7: The Monte Carlo experimental process.

The function `experimentalComparison()`, which was used in Chapter 2 for carrying out k-fold cross-validation experiments, can also be used for Monte Carlo experiments. In the next section we will use it to obtain reliable estimates of the selected evaluation metrics for several alternative trading systems.

3.6.2 Experimental Comparisons

This section describes a set of Monte Carlo experiments designed to obtain reliable estimates of the evaluation criteria mentioned in Sections 3.3.4 and 3.5.2. The base data used in these experiments are the datasets created at the end of Section 3.3.3.

Each of the alternative predictive models considered on these experiments will be used in three different model updating setups. These were already described in Section 3.4.1 and consist of using a single model for all 5-year testing periods, using a sliding window or a growing window. The book package contains two functions that help in the use of any model with these windowing schemes. Functions `slidingWindow()` and `growingWindow()` have five main arguments. The first is an object of class `learner` that we have used before to hold all details on a learning system (function name and parameter values). The second argument is the formula describing the prediction task, while the third and fourth include the train and test datasets, respectively. The final argument is the re-learning step to use in the windowing schema. After the number of test cases is specified in this argument, the model is re-relearned, either by sliding or growing the training data used to obtain the previous model. Both functions return the predictions of the model for the provided test set using the respective windowing schema.

The following code creates a set of functions that will be used to carry out a full train+test+evaluate cycle of the different trading systems we will compare. These functions will be called from within the Monte Carlo routines for different train+test periods according to the schema described in Figure 3.7.

```
> MC.svmR <- function(form, train, test, b.t = 0.1, s.t = -0.1,
+     ...) {
+     require(e1071)
+     t <- svm(form, train, ...)
+     p <- predict(t, test)
+     trading.signals(p, b.t, s.t)
+ }
> MC.svmC <- function(form, train, test, b.t = 0.1, s.t = -0.1,
+     ...) {
+     require(e1071)
+     tgtName <- all.vars(form)[1]
+     train[, tgtName] <- trading.signals(train[, tgtName],
+         b.t, s.t)
+     t <- svm(form, train, ...)
+     p <- predict(t, test)
+     factor(p, levels = c("s", "h", "b"))
+ }
> MC.nnetR <- function(form, train, test, b.t = 0.1, s.t = -0.1,
+     ...) {
+     require(nnet)
+     t <- nnet(form, train, ...)
+     p <- predict(t, test)
```

```
+     trading.signals(p, b.t, s.t)
+ }
> MC.nnetC <- function(form, train, test, b.t = 0.1, s.t = -0.1,
+     ...) {
+     require(nnet)
+     tgtName <- all.vars(form)[1]
+     train[, tgtName] <- trading.signals(train[, tgtName],
+         b.t, s.t)
+     t <- nnet(form, train, ...)
+     p <- predict(t, test, type = "class")
+     factor(p, levels = c("s", "h", "b"))
+ }
> MC.earth <- function(form, train, test, b.t = 0.1, s.t = -0.1,
+     ...) {
+     require(earth)
+     t <- earth(form, train, ...)
+     p <- predict(t, test)
+     trading.signals(p, b.t, s.t)
+ }
> single <- function(form, train, test, learner, policy.func,
+     ...) {
+     p <- do.call(paste("MC", learner, sep = "."), list(form,
+         train, test, ...))
+     eval.stats(form, train, test, p, policy.func = policy.func)
+ }
> slide <- function(form, train, test, learner, relearn.step,
+     policy.func, ...) {
+     real.learner <- learner(paste("MC", learner, sep = "."),
+         pars = list(...))
+     p <- slidingWindowTest(real.learner, form, train, test,
+         relearn.step)
+     p <- factor(p, levels = 1:3, labels = c("s", "h", "b"))
+     eval.stats(form, train, test, p, policy.func = policy.func)
+ }
> grow <- function(form, train, test, learner, relearn.step,
+     policy.func, ...) {
+     real.learner <- learner(paste("MC", learner, sep = "."),
+         pars = list(...))
+     p <- growingWindowTest(real.learner, form, train, test,
+         relearn.step)
+     p <- factor(p, levels = 1:3, labels = c("s", "h", "b"))
+     eval.stats(form, train, test, p, policy.func = policy.func)
+ }
```

The functions `MC.`x`()` obtain different models using the provided formula and training set, and then test them on the given test set, returning the predictions. When appropriate, we have a version for the regression task (name ending in "R") and another for the classification tasks (name ending in "C"). Note that both these alternatives follow different pre- and post-processing

steps to get to the final result that is a set of predicted signals. These functions are called from the `single()`, `slide()`, and `grow()` functions. These three functions obtain the predictions for the test set using the model specified in the parameter `learner`, using the respective model updating mechanism. After obtaining the predictions, these functions collect the evaluation statistics we want to estimate with a call to the function `eval.stats()` that is given below.

```
> eval.stats <- function(form,train,test,preds,b.t=0.1,s.t=-0.1,...) {
+   # Signals evaluation
+   tgtName <- all.vars(form)[1]
+   test[,tgtName] <- trading.signals(test[,tgtName],b.t,s.t)
+   st <- sigs.PR(preds,test[,tgtName])
+   dim(st) <- NULL
+   names(st) <- paste(rep(c('prec','rec'),each=3),
+                      c('s','b','sb'),sep='.')
+
+   # Trading evaluation
+   date <- rownames(test)[1]
+   market <- GSPC[paste(date,"/",sep='')][1:length(preds),]
+   trade.res <- trading.simulator(market,preds,...)
+
+   c(st,tradingEvaluation(trade.res))
+ }
```

The function `eval.stats()` uses two other functions to collect the precision and recall of the signals, and several economic evaluation metrics. Function `sigs.PR()` receives as arguments the predicted and true signals, and calculates precision and recall for the sell, buy, and sell+buy signals. The other function is `tradingEvaluation()`, which obtains the economic metrics of a given trading record. This trading record is obtained with the function `trading.simulator()`, which can be used to simulate acting on the market with the model signals. All these function were fully described and exemplified in Section 3.5.3.

The functions `single()`, `slide()`, and `grow()` are called from the Monte Carlo routines with the proper parameters filled in so that we obtain the models we want to compare. Below we describe how to set up a loop that goes over a set of alternative trading systems and calls these functions to obtain estimates of their performance. Each trading system is formed by some learning model with some specific learning parameters, plus a trading strategy that specifies how the model predictions are used for trading. With respect to trading policies, we will consider three variants that derive from the policies specified in Section 3.5.3 (functions `policy.1()` and `policy.2()`). The following functions implement these three variants:

```
> pol1 <- function(signals,market,op,money)
+   policy.1(signals,market,op,money,
+            bet=0.2,exp.prof=0.025,max.loss=0.05,hold.time=10)
```

```
> pol2 <- function(signals,market,op,money)
+   policy.1(signals,market,op,money,
+            bet=0.2,exp.prof=0.05,max.loss=0.05,hold.time=20)
> pol3 <- function(signals,market,op,money)
+   policy.2(signals,market,op,money,
+            bet=0.5,exp.prof=0.05,max.loss=0.05)
```

The following code runs the Monte Carlo experiments. We recommend that you think twice before running this code. Even on rather fast computers, it will take several days to complete. On the book Web page we provide the objects resulting from running the experiments so that you can replicate the result analysis that will follow, without having to run these experiments on your computer.

```
> # The list of learners we will use
> TODO <- c('svmR','svmC','earth','nnetR','nnetC')
> # The datasets used in the comparison
> DSs <- list(dataset(Tform,Tdata.train,'SP500'))
> # Monte Carlo (MC) settings used
> MCsetts <- mcSettings(20,     # 20 repetitions of the MC exps
+                       2540,   # ~ 10 years for training
+                       1270,   # ~ 5 years for testing
+                       1234)   # random number generator seed
> # Variants to try for all learners
> VARS <- list()
> VARS$svmR   <- list(cost=c(10,150),gamma=c(0.01,0.001),
+                     policy.func=c('pol1','pol2','pol3'))
> VARS$svmC   <- list(cost=c(10,150),gamma=c(0.01,0.001),
+                     policy.func=c('pol1','pol2','pol3'))
> VARS$earth <- list(nk=c(10,17),degree=c(1,2),thresh=c(0.01,0.001),
+                    policy.func=c('pol1','pol2','pol3'))
> VARS$nnetR  <- list(linout=T,maxit=750,size=c(5,10),
+                     decay=c(0.001,0.01),
+                     policy.func=c('pol1','pol2','pol3'))
> VARS$nnetC  <- list(maxit=750,size=c(5,10),decay=c(0.001,0.01),
+                     policy.func=c('pol1','pol2','pol3'))
> # main loop
> for(td in TODO) {
+   assign(td,
+          experimentalComparison(
+            DSs,
+            c(
+              do.call('variants',
+                      c(list('single',learner=td),VARS[[td]],
+                        varsRootName=paste('single',td,sep='.'))),
+              do.call('variants',
+                      c(list('slide',learner=td,
+                             relearn.step=c(60,120)),
+                        VARS[[td]],
```

```
+                         varsRootName=paste('slide',td,sep='.'))),
+              do.call('variants',
+                      c(list('grow',learner=td,
+                             relearn.step=c(60,120)),
+                        VARS[[td]],
+                        varsRootName=paste('single',td,sep='.')))
+              ),
+             MCsetts)
+          )
+
+   # save the results
+   save(list=td,file=paste(td,'Rdata',sep='.'))
+ }
```

The `MCsetts` object controls the general parameters of the experiment that specify the number of repetitions (20), the size of the training sets (2,540 $\sim$ 10 years), the size of the test sets (1,270 $\sim$ 5 years), and the random number generator seed to use.

The `VARS` list contains all parameter variants we want to try for each learner. The variants consist of all possible combinations of the values we indicate for the parameters in the list. Each of these variants will then be run in three different model updating "modes": single, sliding window, and growing window. Moreover, we will try for the two latter modes two re-learn steps: 60 and 120 days.

For the `svm` models we tried four learning parameter variants together with three different trading policies, that is, 12 variants. For `earth` we tried 24 variants and for `nnet` another 12. Each of these variants were tried in single mode and on the four windowing schemes (two strategies with two different re-learn steps). This obviously results in a lot of experiments being carried out. Namely, there will be 60 ($= 12 + 24 + 24$) `svm` variants, 120 ($= 24 + 48 + 48$) `earth` variants, and 60 `nnet` variants. Each of them will be executed 20 times with a training set of 10 years and a test set of 5 years. This is why we mentioned that it would take a long time to run the experiments. However, we should remark that this is a tiny sample of all possibilities of tuning that we have mentioned during the description of our approach to this problem. There were far too many "small" decisions where we could have followed other paths (e.g., the buy/sell thresholds, other learning systems, etc.). This means that any serious attempt at this domain of application will require massive computation resources to carry out a proper model selection. This is clearly outside the scope of this book. Our aim here is to provide the reader with proper methodological guidance and not to help find the best trading system for this particular data.

3.6.3 Results Analysis

The code provided in the previous section generates five data files with the objects containing the results of all variants involving the five learning systems we have tried. These data files are named "svmR.Rdata", "svmC.Rdata", "earth.Rdata", "nnetR.Rdata", and "nnetC.Rdata". Each of them contains an object with the same name as the file, except the extension. These objects are of class `compExp`, and our package contains several methods that can be used to explore the results they store.

Because you probably did not run the experiments yourself, you can find the files on the book Web page. Download them to your computer and then use the following commands to load the objects into R:

```
> load("svmR.Rdata")
> load("svmC.Rdata")
> load("earth.Rdata")
> load("nnetR.Rdata")
> load("nnetC.Rdata")
```

For each trading system variant, we have measured several statistics of performance. Some are related to the performance in terms of predicting the correct signals, while others are related to the economic performance when using these signals to trade. Deciding which are the best models according to our experiments involves a balance between all these scores. The selected model(s) may vary depending on which criteria we value the most.

Despite the diversity of evaluation scores we can still identify some of them as being more relevant. Among the signal prediction statistics, precision is clearly more important than recall for this application. In effect, precision has to do with the predicted signals, and these drive the trading activity as they are the causes for opening positions. Low precision scores are caused by wrong signals, which means opening positions at the wrong timings. This will most surely lead to high losses. Recall does not have this cost potential. Recall measures the ability of the models to capture trading opportunities. If this score is low, it means lost opportunities, but not high costs. In this context, we will be particularly interested in the scores of the models at the statistic "prec.sb", which measures the precision of the buy and sell signals.

In terms of trading performance, the return of the systems is important (statistic "Ret" in our experiments), as well as the return over the buy and hold strategy ("RetOverBH" in our experiments). Also important is the percentage of profitable trades, which should be clearly above 50% (statistic "PercProf"). In terms of risk analysis, it is relevant to look at both the value of the Sharpe ratio and the Maximum Draw-Down ("MaxDD").

The function `summary()` can be applied to our loaded `compExp` objects. However, given the number of variants and performance statistics, the output can be overwhelming in this case.

An alternative is to use the function `rankSystems()` provided by our pack-

age. With this function we can obtain a top chart for the evaluation statistics in which we are interested, indicating the best models and their scores:

```
> tgtStats <- c('prec.sb','Ret','PercProf',
+               'MaxDD','SharpeRatio')
> allSysRes <- join(subset(svmR,stats=tgtStats),
+                   subset(svmC,stats=tgtStats),
+                   subset(nnetR,stats=tgtStats),
+                   subset(nnetC,stats=tgtStats),
+                   subset(earth,stats=tgtStats),
+                   by = 'variants')
> rankSystems(allSysRes,5,maxs=c(T,T,T,F,T))

$SP500
$SP500$prec.sb
          system score
1  slide.svmC.v5     1
2  slide.svmC.v6     1
3 slide.svmC.v13     1
4 slide.svmC.v14     1
5 slide.svmC.v21     1

$SP500$Ret
            system   score
1 single.nnetR.v12 97.4240
2  single.svmR.v11  3.4960
3  slide.nnetR.v15  2.6230
4  single.svmC.v12  0.7875
5   single.svmR.v8  0.6115

$SP500$PercProf
         system   score
1 grow.nnetR.v5 60.4160
2 grow.nnetR.v6 60.3640
3 slide.svmR.v3 60.3615
4  grow.svmR.v3 59.8710
5 grow.nnetC.v1 59.8615

$SP500$MaxDD
          system    score
1  slide.svmC.v5 197.3945
2  slide.svmC.v6 197.3945
3   grow.svmC.v5 197.3945
4   grow.svmC.v6 197.3945
5 slide.svmC.v13 399.2800

$SP500$SharpeRatio
          system score
1  slide.svmC.v5  0.02
2  slide.svmC.v6  0.02
```

```
3 slide.svmC.v13  0.02
4 slide.svmC.v14  0.02
5 slide.svmC.v21  0.02
```

The function `subset()` can be applied to `compExps` objects to select a part of the information stored in these objects. In this case we are selecting only a subset of the estimated statistics. Then we put all trading variants together in a single `compExp` object, using the function `join()`. This function can join `compExp` objects along different dimensions. In this case it makes sense to join then by system variants, as all other experimental conditions are the same. Finally, we use the function `rankSystems()` to obtain the top five scores among all trading systems for the statistics we have selected. The notion of best score varies with each metric. Sometimes we want the largest values, while for others we want the lowest values. This can be set up by the parameter `maxs` of function `rankSystems()`, which lets you specify the statistics that are to be maximized.

The first thing we notice when looking at these top five results is that all of them involve either the `svm` or `nnet` algorithm. Another noticeable pattern is that almost all these variants use some windowing mechanism. This provides some evidence of the advantages of these alternatives over the single model approaches, which can be regarded as a confirmation of regime change effects on these data. We can also observe several remarkable (and suspicious) scores, namely in terms of the precision of the buy/sell signals. Obtaining 100% precision seems strange. A closer inspection of the results of these systems will reveal that this score is achieved thanks to a very small number of signals during the 5-year testing period,

```
> summary(subset(svmC,
+                stats=c('Ret','RetOverBH','PercProf','NTrades'),
+                vars=c('slide.svmC.v5','slide.svmC.v6')))

== Summary of a  Monte Carlo  Experiment ==

 20  repetitions Monte Carlo Simulation using:
         seed =  1234
         train size =  2540  cases
         test size =  1270  cases

* Datasets ::  SP500
* Learners  ::  slide.svmC.v5, slide.svmC.v6

* Summary of Experiment Results:


-> Datataset:  SP500

        *Learner: slide.svmC.v5
              Ret  RetOverBH  PercProf   NTrades
```

```
avg     0.0250000  -77.10350   5.00000 0.0500000
std     0.1118034   33.12111  22.36068 0.2236068
min     0.0000000 -128.01000   0.00000 0.0000000
max     0.5000000  -33.77000 100.00000 1.0000000
invalid 0.0000000    0.00000   0.00000 0.0000000

        *Learner: slide.svmC.v6
              Ret  RetOverBH  PercProf   NTrades
avg     0.0250000  -77.10350   5.00000 0.0500000
std     0.1118034   33.12111  22.36068 0.2236068
min     0.0000000 -128.01000   0.00000 0.0000000
max     0.5000000  -33.77000 100.00000 1.0000000
invalid 0.0000000    0.00000   0.00000 0.0000000
```

In effect, at most these methods made a single trade over the testing period with an average return of 0.25%, which is −77.1% below the naive buy and hold strategy. These are clearly useless models.

A final remark on the global rankings is that the results in terms of maximum draw-down cannot be considered as too bad, while the Sharpe ratio scores are definitely disappointing.

In order to reach some conclusions on the value of all these variants, we need to add some constraints on some of the statistics. Let us assume the following minimal values: we want (1) a reasonable number of average trades, say more than 20; (2) an average return that should at least be greater than 0.5% (given the generally low scores of these systems); (3) and also a percentage of profitable trades higher than 40%. We will now see if there are some trading systems that satisfy these constraints.

```
> fullResults <- join(svmR, svmC, earth, nnetC, nnetR, by = "variants")
> nt <- statScores(fullResults, "NTrades")[[1]]
> rt <- statScores(fullResults, "Ret")[[1]]
> pp <- statScores(fullResults, "PercProf")[[1]]
> s1 <- names(nt)[which(nt > 20)]
> s2 <- names(rt)[which(rt > 0.5)]
> s3 <- names(pp)[which(pp > 40)]
> namesBest <- intersect(intersect(s1, s2), s3)

> summary(subset(fullResults,
                 stats=tgtStats,
                 vars=namesBest))


== Summary of a  Monte Carlo  Experiment ==

 20  repetitions Monte Carlo Simulation using:
         seed =  1234
         train size =  2540  cases
         test size =  1270  cases
```

```
* Datasets ::  SP500
* Learners  ::  single.nnetR.v12, slide.nnetR.v15, grow.nnetR.v12

* Summary of Experiment Results:


-> Dataset:  SP500

        *Learner: single.nnetR.v12
           prec.sb       Ret PercProf      MaxDD SharpeRatio
avg     0.12893147   97.4240 45.88600  1595761.4 -0.01300000
std     0.06766129  650.8639 14.04880  2205913.7  0.03798892
min     0.02580645 -160.4200 21.50000   257067.4 -0.08000000
max     0.28695652 2849.8500 73.08000 10142084.7  0.04000000
invalid 0.00000000    0.0000  0.00000        0.0  0.00000000

        *Learner: slide.nnetR.v15
           prec.sb      Ret  PercProf    MaxDD SharpeRatio
avg     0.14028491  2.62300 54.360500 46786.28  0.01500000
std     0.05111339  4.93178  8.339434 23526.07  0.03052178
min     0.03030303 -7.03000 38.890000 18453.94 -0.04000000
max     0.22047244  9.85000 68.970000 99458.44  0.05000000
invalid 0.00000000  0.00000  0.000000     0.00  0.00000000

        *Learner: grow.nnetR.v12
           prec.sb        Ret PercProf     MaxDD SharpeRatio
avg     0.18774920   0.544500 52.66200  41998.26  0.00600000
std     0.07964205   4.334151 11.60824  28252.05  0.03408967
min     0.04411765 -10.760000 22.22000  18144.11 -0.09000000
max     0.33076923   5.330000 72.73000 121886.17  0.05000000
invalid 0.00000000   0.000000  0.00000      0.00  0.00000000
```

In order to obtain the names of the trading variants satisfying the constraints, we have used the `statScores()` function available in our package. This function receives a `compExp` object and the name of a statistic and, by default, provides the average scores of all systems on this statistic. The result is a list with as many components as there are datasets in the experiments (in our case, this is a single dataset). The user can specify a function on the third optional argument to obtain another numeric summary instead of the average. Using the results of this function, we have obtained the names of the variants satisfying each of the constraints. We finally obtained the names of the variants that satisfy all constraints using the `intersect()` function, which obtains the intersection between sets of values.

As we can see, only three of the 240 trading variants that were compared satisfy these minimal constraints. All of them use a regression task and all are based on neural networks. The three use the training data differently. The "single.nnetR.v12" method does not use any windowing schema and achieves

an impressive 97.4% average return. However, if we look more closely at the results of this system, we see that at the same time on one of the iterations it achieved a return of −160.4%. This is clearly a system with a rather marked instability of the results obtained, as we can confirm by the standard deviation of the return (650.86%). The other two systems achieve rather similar scores. The following code carries out a statistical significance analysis of the results using the function `compAnalysis()`:

```
> compAnalysis(subset(fullResults,
+                     stats=tgtStats,
+                     vars=namesBest))


== Statistical Significance Analysis of Comparison Results ==

Baseline Learner::         single.nnetR.v12  (Learn.1)

** Evaluation Metric::         prec.sb

- Dataset: SP500
       Learn.1    Learn.2 sig.2    Learn.3 sig.3
AVG 0.12893147 0.14028491       0.18774920    +
STD 0.06766129 0.05111339       0.07964205

** Evaluation Metric::         Ret

- Dataset: SP500
     Learn.1 Learn.2 sig.2  Learn.3 sig.3
AVG  97.4240 2.62300    -  0.544500    -
STD 650.8639 4.93178       4.334151

** Evaluation Metric::         PercProf

- Dataset: SP500
     Learn.1   Learn.2 sig.2  Learn.3 sig.3
AVG 45.88600 54.360500    +  52.66200
STD 14.04880  8.339434       11.60824

** Evaluation Metric::         MaxDD

- Dataset: SP500
    Learn.1  Learn.2 sig.2  Learn.3 sig.3
AVG 1595761 46786.28    -- 41998.26    --
STD 2205914 23526.07       28252.05

** Evaluation Metric::         SharpeRatio

- Dataset: SP500
```

```
        Learn.1    Learn.2 sig.2    Learn.3 sig.3
AVG -0.01300000 0.01500000    +  0.00600000
STD  0.03798892 0.03052178       0.03408967

Legends:
Learners -> Learn.1 = single.nnetR.v12 ; Learn.2 = slide.nnetR.v15 ;
            Learn.3 = grow.nnetR.v12 ;
Signif. Codes -> 0 '++' or '--' 0.001 '+' or '-' 0.05 ' ' 1
```

Note that the above code can generate some warnings caused by the fact that some systems do not obtain a valid score on some of the statistics (e.g., no buy or sell signals lead to invalid precision scores).

Despite the variability of the results, the above Wilcoxon significance test tells us that the average return of "single.nnetR.v12" is higher than those of the other systems with 95% confidence. Yet, with respect to the other statistics, this variant is clearly worse.

We may have a better idea of the distribution of the scores on some of these statistics across all 20 repetitions by plotting the `compExp` object:

```
> plot(subset(fullResults,
+             stats=c('Ret','PercProf','MaxDD'),
+             vars=namesBest))
```

The result of this code is shown in Figure 3.8.

The scores of the two systems using windowing schemas are very similar, making it difficult to distinguish among them. On the contrary, the results of "single.nnetR.v12" are clearly distinct. We can observe that the high average return is achieved thanks to a clearly abnormal (around 2800%) return in one of the iterations of the Monte Carlo experiment. The remainder of the scores for this system seem clearly inferior to the scores of the other two. Just out of curiosity, we can check the configuration of this particular trading system using the function `getVariant()`:

```
> getVariant("single.nnetR.v12", nnetR)

Learner::  "single"

Parameter values
         learner  =  "nnetR"
         linout  =  TRUE
         trace  =  FALSE
         maxit  =  750
         size  =  10
         decay  =  0.01
         policy.func  =  "pol3"
```

As you can observe, it uses the trading policy "pol3" and learns a neural network with ten hidden units with a learning decay rate of 0.01.

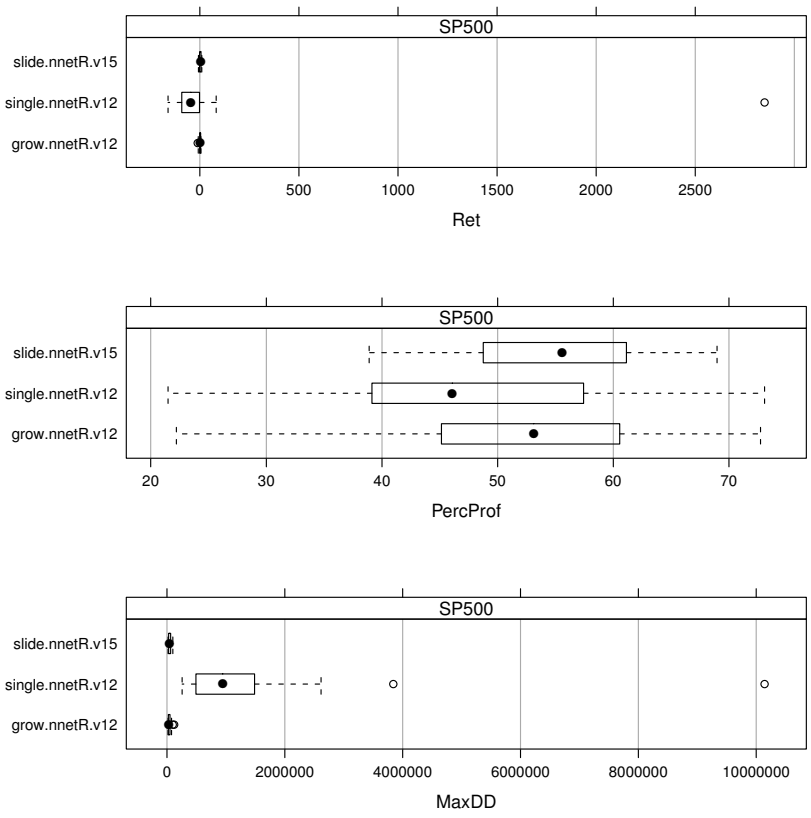

FIGURE 3.8: The scores of the best traders on the 20 repetitions.

In summary, given these results, if we were to select any of the considered alternatives, we would probably skip the "single.nnetR.v12", given its instability. Nevertheless, in the next section we will apply our three best trading systems on the final 9 years of data that were left out for the final evaluation of the best systems.

3.7 The Trading System

This section presents the results obtained by the "best" models in the final evaluation period, which was left out of the model comparison and selection stages. This period is formed by 9 years of quotes, and we will apply the five selected systems to trade during this period using our simulator.

3.7.1 Evaluation of the Final Test Data

In order to apply any of the selected systems to the evaluation period, we need the last 10 years before this evaluation period. The models will be obtained with these 10 years of data and then will be asked to make their signal predictions for the 9 years of the evaluation period. These predictions may actually involve obtaining more models in the case of the systems using windowing schemes.

The following code obtains the evaluation statistics of these systems on the 9-year test period,

```
> data <- tail(Tdata.train, 2540)
> results <- list()
> for (name in namesBest) {
+     sys <- getVariant(name, fullResults)
+     results[[name]] <- runLearner(sys, Tform, data, Tdata.eval)
+ }
> results <- t(as.data.frame(results))
```

We cycle over the three best models, obtaining their predictions by calling them with the initial training data (10 years) and with the evaluation period as test data. These calls involve the use of the functions `single()`, `slide()`, and `grow()` that we have defined before. The result of these functions is a set of evaluation metrics produced by the `eval.stats()` function that we have seen before. At the end of the loop, we transform the obtained list of results into a more appropriate table-like format.

Let us inspect the values of some of the main statistics:

```
> results[, c("Ret", "RetOverBH", "MaxDD", "SharpeRatio", "NTrades",
+     "PercProf")]
```

```
                    Ret RetOverBH      MaxDD SharpeRatio NTrades PercProf
single.nnetR.v12 -91.13    -61.26 1256121.55       -0.03     759    44.66
slide.nnetR.v15   -6.16     23.71  107188.96       -0.01     132    48.48
grow.nnetR.v12     1.47     31.34   84881.25        0.00      89    53.93
```

As you can confirm, only one of the three trading systems achieves positive results in this 9-year period. All others lose money, with the "single.nnetR.v12" system confirming its instability with a very low score of -91.13% return. Among the other two, the "grow.nnetR.v12" method seems clearly better with not only a positive return but also a smaller draw-down and a percentage of profitable trades above 50%. Still, these two systems are clearly above the market in this testing period with returns over the buy and hold of 23.7% and 31.4%.

The best model has the following characteristics:

```
> getVariant("grow.nnetR.v12", fullResults)

Learner::  "grow"

Parameter values
         learner  =  "nnetR"
         relearn.step  =  120
         linout  =  TRUE
         trace  =  FALSE
         maxit  =  750
         size  =  10
         decay  =  0.001
         policy.func  =  "pol2"
```

We now proceed with a deeper analysis of the performance of this best trading system across the evaluation period. For this to be possible, we need to obtain the trading record of the system during this period. The function `grow()` does not return this object, so we need to obtain it by other means:

```
> model <- learner("MC.nnetR", list(maxit = 750, linout = T,
+     trace = F, size = 10, decay = 0.001))
> preds <- growingWindowTest(model, Tform, data, Tdata.eval,
+     relearn.step = 120)
> signals <- factor(preds, levels = 1:3, labels = c("s", "h",
+     "b"))
> date <- rownames(Tdata.eval)[1]
> market <- GSPC[paste(date, "/", sep = "")][1:length(signals),
+     ]
> trade.res <- trading.simulator(market, signals, policy.func = "pol2")
```

Figure 3.9 plots the trading record of the system, and was obtained as follows:

```
> plot(trade.res, market, theme = "white", name = "SP500 - final test")
```

The analysis of Figure 3.9 reveals that the system went through a long period with almost no trading activity, namely since mid-2003 until mid-2007. This is rather surprising because it was a period of significant gain in the market. This somehow shows that the system is not behaving as well as it could, despite the global results observed. It is also noteworthy that the system survived remarkably well during the generally downward tendency in the period from 2000 until 2003, and also during the 2007–2009 financial crisis.

FIGURE 3.9: The results of the final evaluation period of the "grow.nnetR.v12" system.

Package `PerformanceAnalytics` provides an overwhelming set of tools for analyzing the performance of any trading system. Here we provide a glance at some of these tools to obtain better insight into the performance of our trading system. The tools of this package work on the returns of the strategy under evaluation. The returns of our strategy can be obtained as follows:

```
> library(PerformanceAnalytics)
> rets <- Return.calculate(trade.res@trading$Equity)
```

Please note that the function `Return.calculate()` does not calculate the percentage returns we have been using up to now, yet these returns are equivalent to ours by a factor of 100.

Figure 3.10 shows the cumulative returns of the strategy across all testing periods. To obtain such a figure, it is sufficient to run the following code:

```
> chart.CumReturns(rets, main = "Cumulative returns of the strategy",
+     ylab = "returns")
```

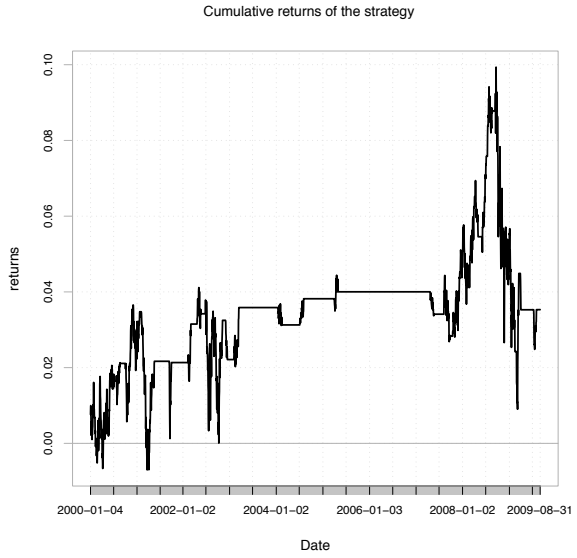

FIGURE 3.10: The cumulative returns on the final evaluation period of the "grow.nnetR.v12" system.

For most of the period, the system is on the positive side, having reached a peak of 10% return around mid-2008.

It is frequently useful to obtain information regarding the returns on an annual or even monthly basis. The package `PerformanceAnalytics` provides some tools to help with this type of analysis, namely, the function `yearlyReturn()`:

```
> yearlyReturn(trade.res@trading$Equity)

           yearly.returns
2000-12-29    0.028890251
```

```
2001-12-31   -0.005992597
2002-12-31    0.001692791
2003-12-31    0.013515207
2004-12-31    0.002289826
2005-12-30    0.001798355
2006-12-29    0.000000000
2007-12-31    0.007843569
2008-12-31    0.005444369
2009-08-31   -0.014785914
```

Figure 3.11 presents this information graphically and we can observe that there were only 2 years with negative returns.

```
> plot(100*yearlyReturn(trade.res@trading$Equity),
+      main='Yearly percentage returns of the trading system')
> abline(h=0,lty=2)
```

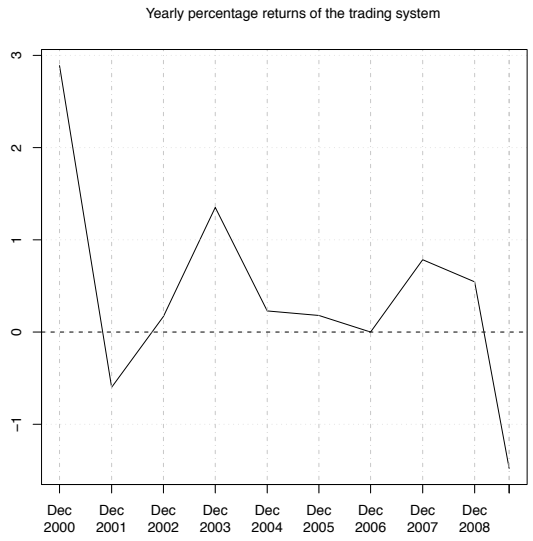

FIGURE 3.11: Yearly percentage returns of "grow.nnetR.v12" system.

The function `table.CalendarReturns()` provides even more detailed information with a table of the percentage monthly returns of a strategy (the last column is the sum over the year):

```
> table.CalendarReturns(rets)
```

```
      Jan  Feb  Mar  Apr May Jun Jul  Aug  Sep  Oct Nov  Dec Equity
2000 -0.5  0.3  0.1  0.2   0 0.2 0.2  0.0  0.0  0.4 0.4 -0.2    1.0
```

```
2001  0.0 -0.3  0.2 -0.1   0 0.0 0.0  0.0  0.4  0.0 0.0  0.0   0.3
2002  0.0 -0.1  0.0 -0.2   0 0.0 0.2  0.0 -0.3 -0.1 0.0  0.0  -0.5
2003  0.0 -0.1  0.0  0.0   0 0.0 0.0  0.0  0.0  0.0 0.0  0.0  -0.1
2004  0.1  0.0  0.0  0.0   0 0.0 0.0  0.0  0.0  0.0 0.0  0.0   0.0
2005  0.0  0.0  0.0 -0.2   0 0.0 0.0  0.0  0.0  0.0 0.0  0.0  -0.2
2006  0.0  0.0  0.0  0.0   0 0.0 0.0  0.0  0.0  0.0 0.0  0.0    NA
2007  0.0  0.0  0.0  0.2   0 0.0 0.0 -0.2  0.0 -0.2 0.2  0.1   0.0
2008 -0.3  0.5  0.1  0.1   0 0.0 0.3  0.0  0.9  0.3 0.2  0.3   2.3
2009 -0.5  0.0 -0.2  0.0   0 0.0 0.0  0.0   NA   NA  NA   NA  -0.6
```

This table clearly shows the long period of inactivity of the system, with too many zero returns.

Finally, we present an illustration of some of the tools provided by the package `PerformanceAnalytics` to obtain information concerning the risk analysis of the strategy using the function `table.DownsideRisk()`:

```
> table.DownsideRisk(rets)

                                Equity
Semi Deviation                  0.0017
Gain Deviation                  0.0022
Loss Deviation                  0.0024
Downside Deviation (MAR=210%)   0.0086
Downside Deviation (Rf=0%)      0.0034
Downside Deviation (0%)         0.0034
Maximum Drawdown               -0.0822
Historical VaR (95%)           -0.0036
Historical ES (95%)            -0.0056
Modified VaR (95%)             -0.0032
Modified ES (95%)              -0.0051
```

This function gives information on several risk measures, among which we find the percentage maximum draw-down, and also the semi-deviation that is currently accepted as a better risk measure than the more frequent Sharpe ratio. More information on these statistics can be found on the help pages of the package `PerformanceAnalytics`.

Overall, the analysis we have carried out shows that the "grow.nnetR.v12" trading system obtained a small return with a large risk in the 9-year testing period. Despite being clearly above the naive buy and hold strategy, this system is not ready for managing your money! Still, we must say that this was expected. This is a rather difficult problem with far too many variants/possibilities, some of which we have illustrated across this chapter. It would be rather surprising if the small set of possibilities we have tried lead to a highly successful trading system.[24] This was not the goal of this case study. Our goal was to provide the reader with procedures that are methodologically sound, and not to carry out an in-depth search for the best trading system using these methodologies.

[24] And it would also be surprising if we were to publish such a system!

3.7.2 An Online Trading System

Let us suppose we are happy with the trading system we have developed. How could we use it in real-time to trade on the market? In this section we present a brief sketch of a system with this functionality.

The mechanics of the system we are proposing here are the following. At the end of each day, the system will be automatically called. The system should (1) obtain whichever new data is available to it, (2) carry out any modeling steps that it may require, and (3) generate a set of orders as output of its call.

Let us assume that the code of the system we want to develop is to be stored on a file named "trader.R". The solution to call this program at the end of each day depends on the operating system you are using. On Unix-based systems there is usually a table named "crontab" to which we can add entries with programs that should be run on a regular basis by the operating system. Editing this table can be done at the command line by issuing the command:

```
shell> crontab -e
```

The syntax of the entries in this table is reasonably simple and is formed by a set of fields that describe the periodicity and finally the command to run. Below you can find an example that should run our "trader.R" program every weekday by 19:00:

```
0 19 * * 1-5 /usr/bin/R --vanilla --quiet < /home/xpto/trader.R
```

The first two entries represent the minute and the hour. The third and fourth are the day of the month and month, respectively, and an asterisk means that the program should be run for all instances of these fields. The fifth entry is the weekday, with a 1 representing Mondays, and the '-' allowing for the specification of intervals. Finally, we have the program to be run that in this case is a call to R with the source code of our trader.

The general algorithm to be implemented in the "trader.R" program is the following:

```
- Read in the current state of the trader
- Get all new data available
- Check if it is necessary to re-learn the model
- Obtain the predicted signal for today
- With this signal, call the policy function to obtain the orders
- Output the orders of today
```

The current state of the trader should be a set of data structures that stores information that is required to be memorized across the daily runs of the trader. In our case this should include the current NNET model, the learning parameters used to obtain it, the training data used to obtain the model and the associated data model specification, the "age" of the model (important to know when to re-learn it), and the information on the trading

record of the system until today and its current open positions. Ideally, this information should be in a database and the trader would look for it using the interface of R with these systems (see Section 3.2.4). Please note that the information on the open positions needs to be updated from outside the system as it is the market that drives the timings for opening and closing positions, contrary to our simulator where we assumed that all orders are accomplished at the beginning of the next day.

Getting the new available data is easy if we have the data model specification. Function `getModelData()` can be used to refresh our dataset with the most recent quotes, as mentioned in Section 3.3.2.

The model will need to be re-learned if the age goes above the `relearn.step` parameter that should be memorized in conjunction with all model parameters. If that is the case, then we should call the `MC.nnetR()` function to obtain the new model with the current window of data. As our best trader uses a growing window strategy, the training dataset will constantly grow, which might start to become a problem if it gets too big to fit in the computer memory. If that occurs, we can consider forgetting the too old data, thereby pruning back the training set to an acceptable size.

Finally, we have to get a prediction for the signal of today. This means calling the `predict()` function with the current model to obtain a prediction for the last row of the training set, that is, today. Having this prediction, we can call the trading policy function with the proper parameters to obtain the set of orders to output for today. This should be the final result of the program.

This brief sketch should provide you with sufficient information for implementing such an online trading system.

3.8 Summary

The main goal of this chapter was to introduce the reader to a more real application of data mining. The concrete application that was described involved several new challenges, namely, (1) handling time series data, (2) dealing with a very dynamic system with possible changes of regime, and (3) moving from model predictions into concrete actions in the application domain.

In methodological terms we have introduced you to a few new topics:

- Time series modeling
- Handling regime shifts with windowing mechanisms
- Artificial neural networks
- Support vector machines

- Multivariate adaptive regression splines
- Evaluating time series models with the Monte Carlo method
- Several new evaluation statistics related either to the prediction of rare events or with financial trading performance

From the perspective of learning R we have illustrated

- How to handle time series data
- How to read data from different sources, such as data bases
- How to obtain several new types of models (SVMs, ANNs, and MARS)
- How to use several packages specifically dedicated to financial modeling