CHAPTER 14

Further Discussion on Regression and Classification Trees, Computer Software, and Other Useful Classification Methods

14.1 R PACKAGES FOR TREE CONSTRUCTION

R includes several excellent packages for tree construction, of which the foremost are **tree** and **rpart**. We have explained the package **tree** in the previous chapter. Alternatively, one can use the package **rpart** (rpart stands for recursive partitioning). The methodologies and the outputs of the two routines are very similar, and so are the solutions that are found by them. You can experiment with either package and compare the output. The following discussion may help you understand the slight differences in the results.

The deviance node impurity is used as the default in tree(); it coincides exactly with the criteria that have been covered in Chapter 13: the sum of squares criterion for regression trees and the classification deviance criterion for classification trees. The parameters in the tree.control() function—mincut (minimum number of items in a node, default 5), minsize (minimum number of items in a node before it is being considered for a cut; default 10), and mindev (the required change in deviance before a split is being carried out)—determine the forward motion of growing a tree. Forward growth stops when one of these stopping parameters is reached. The cross-validation routine cv.tree() and the pruning routine prune.tree() in the package **tree** use the deviance measure in the calculation of the complexity parameters. The cost complexity tells us about the required change in the deviance per number of cut leaves that is needed before a subtree can be pruned; if the cost complexity is smaller than the change in deviance, we simplify (i.e., prune) the tree. Note that the R package **tree** considers absolute, not relative changes. The output of the cross-validation routine cv.tree() lists the average cross-validation deviance for various tree sizes. One locates the size of tree that minimizes the average deviance. But one should not always insist on the absolute minimum. One usually scans the average deviances to find the

Data Mining and Business Analytics with R, First Edition. Johannes Ledolter.

smallest tree with an average deviance that is not much larger than the minimum deviance.

The Gini node impurity measure is used as the default in rpart(); the deviance impurity measure is used if it is specified through parms=list(split="information"). The forward growth of the tree is controlled through the parameters in the rpart.control function: minsplit (same as minsize in tree; the minimum number of observations that must exist in a node for a split to be attempted with default 20), minbucket (same as mincut in tree; number of observations in a terminal node with default round(minsplit/3)), and cp (the required percentage change in node impurity before a split is being carried out with default 0.01). Forward growth stops whenever one of the growth thresholds is reached. The complexity parameters in rtree() are defined proportionally relative to the root node (this is different from tree()), and they are defined in terms of the reduction in the number of misclassifications, and not in terms of the reduction of node impurity (the criterion used to grow the tree). Also, rpart() uses the number of misclassifications for the calculation of the errors (which are expressed relative to the number of errors at the root node). For given size of the tree (defined as number of splits, which is the size of the tree minus 1), rpart() calculates the number of errors in the training set, as well as the average and the standard deviation of the errors from the cross-validation of a tree of that size. Again one looks for the number of splits for which the average error from cross-validation has flattened out; it has been recommended that one adds one standard deviation to the minimal error and uses that as the cutoff. Also note that the cross-validation output from rpart() may list results for #splits=0 and #splits=2, but not for #splits=1. This happens when the reduction in the number of misclassification errors from #splits=0 to #splits=1 is smaller than half of the reduction in the number of misclassification errors from #splits=0 to #splits=2.

14.2 CHI-SQUARE AUTOMATIC INTERACTION DETECTION (CHAID)

CHAID or *chi-square automatic interaction detection* is a tree-structured classification procedure developed by Kass (1980) and further improved by Biggs et al. (1991). It is part of the SAS Enterprise Miner software and the SPSS data mining software AnswerTree.

Our previous discussion emphasized that classification and regression trees are useful for extracting structure from large multivariate data sets. These techniques partition a data set into mutually exclusive, exhaustive subsets that "best" describe a given response variable. Classification/regression trees are hierarchical displays that result from a series of questions about the outcomes of the predictor variables on each unit in the data set. At the end of the process, one knows the most likely response value or class membership of each unit. One speaks of a classification tree if the response variable is categorical, and of a regression tree if the response variable is continuous.

It is called a *tree* because the resulting display resembles an upside down tree, with a root on top (the entire data set), a series of branches connecting nodes, and leaves at the bottom. At each node (initially starting with the entire data set), a question about one of the predictor variables is posed; the branch taken at that node depends on the answer to this question. The order in which questions are asked and the rules about node splits are important, as they determine the structure of the tree. Many different algorithms have been proposed, and a detailed discussion of algorithms that split nodes of regression and classification trees on deviances has been given in the previous chapter. A general principle in tree construction focuses on "node purity" at each node-splitting juncture. The tree is built by repeatedly splitting subsets of the data to create further subsets, which are as homogeneous as possible with respect to the response variable. CART, the popular tree-based approach for continuous and categorical response variables that has been discussed in the previous chapter, generates binary trees. At each stage, it splits subsets of the data into just two child nodes of one of the predictor variables. See Breiman et al. (1984) for further discussion.

CHAID is designed to handle categorical response and categorical predictor variables. For example, the response may be the degree of success of an organization. Degree of success may be measured by a dichotomous (successful/not successful) variable, or success may be given as a nominal-scaled categorical variable with more than two outcomes. The predictor variables may be the size of the organization (small, medium, large), whether the organization is a multinational corporation or not, and the degree to which new management techniques are being adopted (on an integer scale from 1 to 10). A CHAID decision tree is constructed by splitting the units on one of the feature (explanatory) variables into two *or more* nodes; splitting the data set into more than two nodes makes this technique different from CART, which considers binary splits exclusively.

The basic idea behind CHAID is as follows. Assume a nominal response variable with d possible outcomes and a nominal explanatory variable with c outcomes; for the time being, consider just one explanatory variable; more explanatory variables will be introduced shortly. The Pearson chi-square statistic is used to assess the relationship among the two categorical variables. CHAID maximizes the Pearson chi-square statistic between the d outcomes on the response variable and all possible groupings of the c outcomes on the explanatory variable. For an explanatory variable with say $c = 7$ outcomes, the grouped table of frequencies with outcomes 1 and 2 on the explanatory variable in one group, outcomes 3 and 6 in a second group, and outcomes 4, 5, and 7 in the third group may maximize the Pearson chi-square statistics. Assuming that the chi-square statistic for this 3 by d table is significant, CHAID would then split the data into these three groups. The tree would show three leaves, with each leaf listing proportions of the d response categories calculated from the units in that particular leaf. This is straightforward, but the search over all possible groupings of the categorical explanatory variable is time consuming (especially as this has to be repeated for several categorical variables and for many nodes). Hence, the following "merge" strategy is suggested.

For a given node (at the beginning we start with all data) and for a given predictor variable, the procedure merges a pair of predictor variable outcomes if there is no statistically significant association between them and the outcomes on the response variable; the probability value of the chi-square statistic from the 2 by d table of frequencies is compared to a given alpha-to-merge value. The merge step is then repeated, and the procedure finds the next pair of categories to merge, which may now include previously merged categories. The merging step stops if the statistical significance for the last pair of predictor categories to be merged is significant. At that step the procedure will have found the best split for that predictor variable.

The next step in CHAID is to select the split variable. The predictor variable with the smallest p-value among the chi-square statistics testing the association between the response outcomes and the optimally merged outcomes on the predictor variables is selected for the split. That is, the predictor variable that yields the most significant split is selected for the split. If the smallest p-value is greater than some alpha-to-split value, then no further splits from the present node will be performed, and the present node is a terminal node. The process is repeated at each node until one of the stopping rules is triggered; that is, either a further split would violate user-specified minimum data requirements, or no more significant splits can be found.

Of course, such a search procedure may not find the split that is best at each node. Only an exhaustive search procedure will do that. The modifications by Biggs et al. (1991) enhance the search procedure.

14.3 ENSEMBLE METHODS: BAGGING, BOOSTING, AND RANDOM FORESTS

Do not put all your eggs into one basket. Because if all your eggs are in one basket and you drop the basket, you lose everything. Similarly, do not put all your trust in a single classification or prediction method. Use the information from several alternative methods if more than one method is available.

The advantage of aggregating several methods tends to be greatest if the methods are unrelated (independent). The advantage of aggregation disappears if the methods are identical (i.e., strongly correlated). In forecasting, there is a large literature on the benefits of combining forecasts of quantitative information such as sales, going back to the paper by Granger and Bates (1969). Suppose the head of a forecasting division has two sources of forecasts for the company's sales, one source being the forecasts developed by the division's econometrics group using a time series model and the other source being the aggregated forecasts of the regional sales managers of the company. Suppose that the forecast horizon is $h = 1$. Represent the one-step-ahead forecast of the econometrics group made at time t of the observation y_{t+1} by f^1_{t+1} and that of the managers by f^2_{t+1}, and consider the combination forecast $f^c_{t+1} = \omega f^1_{t+1} + (1 - \omega)\ f^2_{t+1}$. Assume that either forecast is unbiased (i.e., $E(y_{t+1} - f^1_{t+1}) = E(y_{t+1} - f^2_{t+1}) = 0$) and assume that

the one-step-ahead forecast errors have variances σ_{11} and σ_{22}, and covariance σ_{12}. It is straightforward to show that the combined forecast is also unbiased, and that its variance is given by $E(y_{t+1} - f^c_{t+1})^2 = \omega^2\sigma_{11} + (1-\omega)^2\sigma_{22} + 2\omega(1-\omega)\sigma_{12}$. This variance is smallest for $\omega = (\sigma_{22} - \sigma_{12})/(\sigma_{11} + \sigma_{22} - 2\sigma_{12})$. We can show this by taking the first derivative with respect to ω and setting it equal to zero. For forecasts with the same precision ($\sigma^2 = \sigma_{11} = \sigma_{22}$), the individual forecasts should be averaged ($\omega = 0.5$), and this is true for any value of the covariance σ_{12}. Then the forecast error of the resulting combined (averaged) forecast has variance $E(y_{t+1} - f^c_{t+1})^2 = \sigma^2(1+\rho)/2 \leq \sigma^2$, which is never larger than the variance of an individual method. The benefit of averaging is actually largest if the correlation $\rho = \sigma_{12}/\sigma^2$ between the two unbiased forecast errors is negative, as then the two errors compensate each other.

The same idea can be applied to the classification of a new case into one of $m \geq 2$ different outcome categories. Assume that results from an ensemble of k different base classifiers are available. The ensemble returns a class prediction that is based on the votes of the base classifiers. The combined (ensemble) classifier is given by the outcome that occurs most often. An ensemble classifier tends to be more accurate than its components. Each base classifier in a binary classification may make mistakes, but the ensemble classifier will lead to an error only if over half of the base classifiers are incorrect. Ensemble methods work well if there is little correlation among the classifiers and if each classifier is better than random guessing. Combining classifiers that are all alike brings little benefit. Combining classifiers that resemble random guessing does not lead to an advantage either, as the distribution of the classifiers of the ensemble is again uniform across the possible outcome groups.

There are various ways of creating ensembles. The classifiers could come from different methods, such as logistic regression, nearest neighbor methods, classification trees, naïve Bayesian methods, or discriminant methods. Or, the classifiers can come from different subsets of the training part of the data. *Bootstrap* aggregation, or *bagging*, combines classifiers across different subsets of the training data. Assume that the training data set consists of d cases, each case being described by its attributes and the true classification. Bagging selects a new training set of d cases by selecting d of the cases through sampling *with* replacement (a certain case could come up multiple times in the new training set), constructs the model from this new training set, and uses the fitted model to classify the new case. This process is repeated k times, resulting in k classifications of the new test case. The ensemble classifier assigns to the new test case the most frequent classification. The bootstrap (or bagged) classifier is often considerably better than a single classifier that is derived from the original training set. It certainly will not be much worse, and it has been shown to be robust to overfitting and to noisy data.

Boosting is similar, except that there the ensemble classifier assigns weights to the individual base classifiers when combining them. *Adaptive boosting*, for example, works as follows. We start, as in bagging, constructing a new training set of d cases from the original training set by selecting d of the cases through sampling *with* replacement, constructing the model (classifier) from this

new training set, and using the fitted model to classify the new case. In the beginning, each case of the original training set is assigned weight $1/d$. But now, we assess the internal performance of the resulting classifier on the new training set that has been used for its construction, and we revise the weights for drawing the next training sample. If a case in the training sample has been classified incorrectly, we increase its weight; if the case has been classified correctly, we reduce its weight. The revised weights reflect how difficult it is to classify each case. The revised weights are used to draw the next bootstrap sample, construct a new classifier, obtain the classification of the test case, assess the internal performance of the classifier, and revise the weights for the draw of the next training sample, and so on. Note that each new training sample increases the focus on misclassified cases. The boosting strategy tends to build a series of classifiers that complement each other. Once we have obtained k training sets and k classifiers, adaptive boosting combines the results, and it does so by assigning a weight to each classifier's vote that reflects how well this classifier has performed internally. Strategies for revising the weights from one draw to the next and for weighting the votes of the classifiers are discussed in the literature, and programs for implementing this approach are available. In general, boosting tends to increase the accuracy.

The *randomForest* method is another way to combine information across an ensemble. Assume that each of the k classifiers in the ensemble is a decision tree for classifying a new element into one of m possible outcome groups. At each node, an individual decision tree determines the split on the basis of a smaller, random selection of attributes, and not from the set of all attributes. Each tree in the forest of trees (hence the method's name) then votes on the classification of a new item, and the most popular class is returned as the ensemble solution.

Random attribute selection can be combined with bagging. There a training sample of the d elements from the initial training set of d elements is selected with replacement. At each node of each tree, a randomly selected set of A attributes (selected from the usually much larger set of all attributes) is used to determine the split. Each tree is grown to maximum size and is not pruned. Each tree votes, and the most popular class becomes the ensemble classification. Random forests formed with random attribute input selection are referred to as *Forest-RI*. Several modifications of this basic strategy have been developed, and computer software for their implementation is available.

When constructing a single tree one notices quite often that there is little difference between choosing one splitting variable or several others. One or more attribute variables usually have the same ability to partition the training data set into homogeneous groups, and in such cases, luck or small changes in the training data set determine whether the algorithm prefers one splitting variable over the others. Random forest techniques tend to do better. The randomness of variable selection delivers robustness to noise and the presence of attribute variables that have weak relationships with the target variable, to outliers, and to small changes in the training data set. These conditions usually have little impact on the final

decisions made by the ensemble classifier. A random forest technique also handles underrepresented classes quite well. In addition, it has computational efficiencies as it restricts the search.

Software for carrying out this methodology is available in R; the package **randomForest** a good place to start. Typically, one constructs 500 random trees, with each tree constructed on training data that has been randomly selected from the original training data set (say 70% of the complete data). Randomization enters twice: through the randomly selected training data set (each tree uses different training data) and through the random selection of a few of the attributes (four or so) at each of the node splits. These two random selections make random forests robust to noisy observations and weak attributes. The randomForest() command, through its control parameter replace=TRUE/FALSE, achieves the randomization of the tree-specific training sets through two very similar sampling procedures. Under replace=TRUE, the training set is sampled with replacement from the number of cases that have been established as the training set; typically the sample size is the same as the size of the training set. Because of sampling with replacement, not all elements of the original training set are selected (on average, roughly 1/3 of the elements are not chosen) and a few elements are selected more than once. Under replace=FALSE, the sampling is without replacement, and in this case, the program samples 63.2% of the elements from the training set. Again, roughly one-third of the cases are not selected.

The randomForest package evaluates the classification procedure internally by computing classification errors on only those elements of the training set that have been left out of the training data for at least one of the constructed trees. Elements that are included in the training data for all of the constructed trees are not part of this evaluation. The program refers to the resulting error as the "out-of-bag" misclassification error; the expression "out of bag" comes from the bootstrap aggregation or bagging of the training data. Misclassification error rates for evaluation and test data sets can also be obtained, and they tend to give a more accurate picture of the out-of-sample performance.

The randomForest package also allows the user to specify the number of elements that should be included in each tree's training data set (typically the sample size is the same as the size of the training set). Furthermore, the program allows the user to specify the number of elements that should come from each of the outcome groups; for example, the statement sampsize=c(30,30) forces the selection of 30 samples from each of two groups. This can be an advantage in situations where the groups are not balanced, and one group is considerably more prevalent than the other. Without the sampsize argument, the program selects the cases from the training set at random, and trees constructed on unbalanced training data will have difficulties identifying members of the rare group. By forcing the program to select the same number of training elements from each strata, the trees are given a better chance to identify the rare group. The randomForest software package also includes several ways of assessing the importance of the attributes. The simplest way to determine importance is to count the number of trees in which a certain attribute is present for making a decision.

14.4 SUPPORT VECTOR MACHINES (SVM)

Support vector machines (SVM) comprise another very general group of classification methods, and they cover both linear and nonlinear classifiers. Assume that we want to classify two-dimensional training data into two groups. The information we have available can be shown through a scatter plot of the two attributes, and the plotting positions can be represented with two different labels or colors that identify the two classes. Suppose we are fortunate enough that the data are linearly separable, which means that a straight line through the data set separates the two classes exactly. Quite often there are several lines with different intercepts and slopes that can achieve the separation, and the objective is to find the line that achieves the best possible one. What do we mean by the best separation? Find a separation line and add two parallel lines of equal distance to that line such that each parallel line passes through the data point that is closest to the line of separation. The distance between the two parallel lines, also called the *margin*, measures how good the separation really is. If the distance between these two parallel lines is large, then the two groups are far apart and we have found an excellent classifier. The objective is to find the classifier with the best margin of separation. In two-dimensional space where data can be graphed on a scatter plot, it is fairly straightforward to find the separation line, the two parallel lines, and the margin. In high dimensional space, with more than two attributes and more than two groups, this becomes a complicated problem. Nevertheless mathematical tools are readily available to find the hyperplane that separates the groups with the largest margin. SVM achieves this by finding "support vectors" that identify the data points on the hyperplanes with the largest margins.

But, what if we are not so fortunate and our data is not linearly separable? Then SVM methods transform the data into a higher dimensional space in which such complete separation is possible. With an appropriate nonlinear mapping into a sufficiently high dimension, such separation is always possible, and the methods that we outlined for linearly separable data can be applied to the transforms. SVM tries several nonlinear transformations, including the one that considers powers and cross products of the attributes.

SVMs are useful, and programs are available in R. However, in this text, we do not discuss SVM in detail, as its full explanation requires advanced tools from mathematical optimization. For reference, you may want to look at the article "Support vector machines in R," by Karatzoglou et al. (2006), the command ksvm() in the R package **kernlab**, and the command svm() in the R package **e1071**.

14.5 NEURAL NETWORKS

Neural networks comprise yet another group of nonlinear procedures for prediction and classification. Neural networks are computational analogs of the processes that describe the working of neurons. Neural nets consist of connected input and output layers, each containing numerous units, where the connections among the units of

the layers have weights associated with them. A multilayer neural network consists of an input layer of distinct units (usually input units represent the attributes of the case that needs to be classified), one or more hidden layers, and an output layer with several units that represent the class that needs to be predicted. Each layer is made up of several units. The inputs from the units of the first layer are weighted and fed simultaneously to a second layer of neuron-like units, known as the *hidden layer*. Each of its units takes as its input a weighted sum of the outputs of the previous layer and applies a nonlinear activation function to determine its output. The activation function is usually a logistic function that transforms the output to a number that is between 0 and 1; for this reason, it is often referred to as the *squashing* function. The output units may already be the actual output of the system (in which case we talk about a single hidden layer), or the outputs of the first hidden layer can be fed as inputs to another hidden layer (in which case we have two hidden layers), and so on if additional hidden layers are involved.

One needs to decide on the number of hidden layers and the number of units in the various layers, as well as the weights that connect the inputs and outputs of these layers. Neural nets are very general, and because of the nonlinear activation functions neural nets are able to approximate most nonlinear functional relationships very well. Of course, the weights in these systems have to be estimated by training the system on actual data. The iterative method of back-propagation can be used to determine the empirical weights that lead to the best fit on a given training sample. For that, one needs to specify a learning parameter that controls the speed of convergence of this iterative estimation method.

Neural nets are very general and can approximate complicated relationships. But they also come with disadvantages. One disadvantage of neural nets is that the approximating models relating inputs and outputs are purely "black box" models, and they provide very little insight into what these models really do. Also, the user of neural nets must make many modeling assumptions, such as the number of hidden layers and the number of units in each hidden layer, and usually there is little guidance on how to do this. It takes considerable experience to find the most appropriate representation. Furthermore, back-propagation can be quite slow if the learning constant is not chosen correctly.

In this chapter, we do not emphasize neural networks. Software for carrying out this methodology is available in R; the packages **nnet** and **neuralnet** are good places to start from.

14.6 THE R PACKAGE RATTLE: A USEFUL GRAPHICAL USER INTERFACE FOR DATA MINING

A graphical user interface (GUI) allows people to interact with software in more ways than just by typing text. The Microsoft Office products are well-known GUIs. Take Microsoft Excel, for example, which allows users to simply point a cursor at a certain icon and click. Excel and other spreadsheet programs, such as Minitab and SPSS for statistical applications, have become popular because they rely on a

simple "point and click" interface. R, on the other hand, is not that user-friendly because one must type commands into an R console. However, more and more R GUIs are being developed. **Rcmdr** (R commander) is one of them, and it provides an excellent GUI especially for statistical analysis. Typing onto the R console the statement "library(Rcmdr)" opens up a graphical interface, and from there numerous statistical functions are just a mouse-click away.

The R library **rattle**, developed by Williams and discussed in his book "Data Mining with Rattle and R: the Art of Excavating Data for Knowledge Discovery" (Williams, 2011), provides a very useful GUI to important data-mining libraries in R. Typing onto the R console the statement "library(rattle)" and then "rattle()" into the next line opens up a graphical interface to R. Once the data is loaded into rattle, the analysis can be carried out by entering answers to simple queries and mouse clicks. This makes the R analysis simple and easy, as data mining is now carried out in a spreadsheet environment that is familiar to users of programs such as Excel or Minitab. By clicking on various window tabs, the user of rattle can execute data-mining tasks such as constructing regression/classification trees discussed in this and the previous chapter, clustering (forthcoming Chapter 15), and association analysis (forthcoming Chapter 16), and can do so without having to write out detailed R instructions. In addition, rattle makes it convenient to preprocess the data. Once the data has been loaded into rattle, it is easy to specify the data type and declare the data as continuous or categorical (rattle infers the data type automatically; while it is actually very good at this, the user can always overwrite its selection). It is also easy to get information on missing data and identify outliers, and simple methods for adjusting outliers and imputing missing observations are available. Rattle provides useful graphical displays for one or more variables and calculates standard summary statistics such as means, standard deviations, and correlation coefficients, and it does so without the user having to write out R instructions. Rattle reduces much of the programming pain, as it becomes unnecessary to write out R instructions. The user does not need to remember the R code; information is simply entered into various windows and the information is being translated into R code. An added benefit of rattle is that the generated code is visible to the user, who then can change the code for more sophisticated analyses. Experiment with this R package by loading the library rattle, and start it by typing rattle().

Rattle makes it very easy to split the data into training, evaluation, and test data sets, and to evaluate the models that are fitted to one set of data on other data sets. The evaluation of the predictive performance of a model on the very same data that has been used for its estimation is most likely too optimistic. Just think of regression models that can be made to fit better and better just by increasing the number of estimated coefficients, but at the expense of their forecasting performance. It is important that the predictive performance of models and methods is being assessed on new observations that are independent of those that were used at the model estimation/fitting stage. Typically, one divides the data set into two, or sometimes three nonoverlapping parts: the training, evaluation, and test data sets. The training data set is used to build and estimate the model, and the test data set

is used to evaluate the out-of-sample performance of the model (which can result in a prediction of a response variable or a classification of a new case). Sometimes a third data set, called the *evaluation data set*, is present. Consider LASSO regression for predicting the outcome of a certain response. The implementation of LASSO regression for prediction requires the knowledge of a penalty parameter λ, which usually is determined through cross-validation on an independent data set. This is where the evaluation data set comes in. For each of several choices of λ, penalized regression estimates are obtained from the training set; these estimates are then used to determine the predictions for cases in the evaluation data set. The resulting prediction errors are calculated, and the penalty parameter is estimated by minimizing the sum of the squared prediction errors from the evaluation data set. Once the penalty parameter is found, one can use the LASSO estimates from the training data, using the penalty that was found on the evaluation data set, to evaluate the model performance on the test data set. Rattle has easy ways of splitting the data into these three parts, and they use splits, such as 70/15/15. In certain other modeling situations, just two independent data sets are needed: the training data set and the test data set. Then we use the terms "test" and "evaluation" data sets interchangeably to refer to the same data set. With just two data sets, it is common to split the complete data set into two equal parts (50/50 split).

Rattle covers several important data-mining tasks, but not all methods discussed in this book are covered; it does not address regression and logistic regression, LASSO and false discovery rates, nearest neighbor methods, and principal components and partial least squares, network analysis and the analysis of text information (topics that are being discussed in subsequent chapters).

REFERENCES

Biggs, D., DeVille, B., and Suen, E.: A method of choosing multiway partitions for classification and decision trees. *Journal of Applied Statistics*, Vol. 18 (1991), 49–62.

Breiman, L., Friedman, J., Ohlsen, R., and Stone, C.: *Classification and Regression Trees*. Pacific Grove: Wadsworth, 1984.

Granger, C.W.J and Bates, J.: The combination of forecasts. *Operations Research Quarterly*, Vol. 20 (1969), 451–468.

Karatzoglou, A., Meyer, D., and Hornik, K: Support vector machines in R. *Journal of Statistical Software*, Vol. 15 (2006), No 1, 1–28.

Kass, G.V.: An exploratory technique for investigating large quantities of categorical data. *Applied Statistics*, Vol. 29 (1980), 119–127.

Williams, G.: *Data Mining with Rattle and R: the Art of Excavating Data for Knowledge Discovery*. New York: Springer, 2011.