# Chapter 15

# Model Performance Evaluation

*If a model looks too good to be true, then generally it is.*

The preceding chapters presented a number of algorithms for building descriptive and predictive models. Before we can identify the best from amongst the different models, we must evaluate the performance of the model. This will allow us to understand what to expect when we use the model to score new observations. It can also help identify whether we have made any mistakes in our choice of input variables. A common error is to include as an input variable a variable that directly relates to the outcome (like the amount of rain tomorrow when we are predicting whether it will rain tomorrow). Consequently, this input variable is exceptionally good at predicting the target.

In this chapter, we consider the issue of evaluating the performance of the models that we have built. Essentially, we consider `predict()`, provided by R and accessed through Rattle's Evaluate tab, and the functions that summarise and analyse the results of the predictions.

We will work through each of the approaches for evaluating model performance. We start with a simple table, called a confusion matrix, that compares predictions with actual answers. This also introduces the concepts of true positives, false positives, true negatives, and false negatives. We then explain a risk chart which graphically compares the performance of the model against known outcomes and is used to identify a suitable trade-off between effort and outcomes. Traditional ROC

G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, Use R, DOI 10.1007/978-1-4419-9890-3_15, © Springer Science+Business Media, LLC 2011

curves are then introduced. We finish with a discussion of simply scoring datasets and saving the results to a file.

In applying a model to a new dataset, the new dataset must contain all of the same variables and have the same data types on which the model was built. This is true even if any variables were not used in the final model. If the variable is missing from the new dataset, then generally an error is generated.

## 15.1 The Evaluate Tab: Evaluation Datasets

Rattle's Evaluate tab provides access to a variety of options for evaluating the performance of our models. We can see the options listed in Figure 15.1. We briefly introduce the options here and expand on them in this chapter.

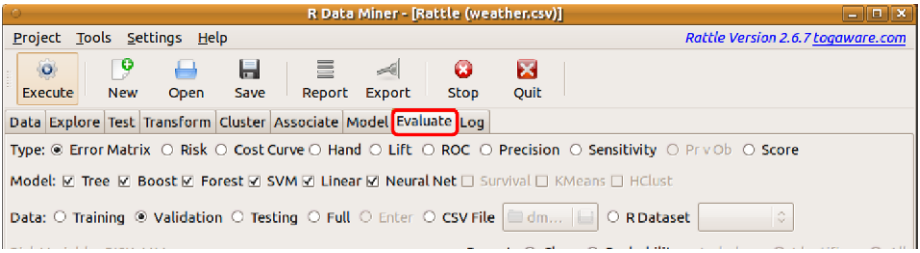


Figure 15.1: The Evaluate tab options.

### Types of Evaluations

The range of different Types of evaluations is presented as a series of radio buttons running from Confusion Matrix to Score. Only one type of evaluation is permitted to be chosen at any time. Each type of evaluation is presented in the following sections of this chapter.

### Models to Evaluate

Below the row of evaluation types is a row of check boxes to choose the models we wish to evaluate. These check boxes are only available once a model has been built. As models are built, the check boxes will become available as options to check.

As we move from the Model tab to this Evaluate tab, the most recently built model will be automatically checked (and any previously checked model choices will be unselected). This corresponds to a common pattern of behaviour in that often we will build and tune a model, then want to explore its performance by moving to this Evaluate tab. If the All option has been chosen within the Model tab, then all models that were successfully built will automatically be checked on the Evaluate tab. This is the case here, where the six predictive models are checked.

**Dataset Used for Evaluation**

To evaluate a model, we need to identify a dataset on which to perform the evaluation. The next row of options within the Rattle interface provides a collection of alternative sources of data.

The first four options for the Data correspond to the partitioning of the dataset specified on the Data tab. The options are Training, Validation, Testing, and Full. The concept of a training/validation/testing dataset partition was discussed in Section 3.1, and we discussed the concept of sampling and associated biases in Section 4.7. We now discuss it further in the context of evaluating the models.

The first option (but not the best option) is to evaluate our model on the Training dataset. This is generally not a good idea, and an information dialogue will be shown to reinforce this.

The problem with evaluating our model on the training dataset is that we have built it on this training dataset. It is often the case that the model will perform very well on that dataset. It should, because we've tried hard to make sure it does. But this does not give us a very good idea of how well the model will perform in general on previously unseen data.

We need a better guide to how well the model will perform in general, that is, how the model will perform on new and previously unseen data. To answer that question, we need to apply the model to such data. In doing so, we will obtain the overall error rate of the model. This is simply the proportion of observations where the model and the actual known outcomes differ. This error rate, and not the error rate from the training dataset, will then be a better estimate of how well the model will perform. It is a less biased estimate of the error.

We use the Validation dataset to test the performance of a model whilst we are building and fine-tuning it. Thus, after building one deci-

sion tree, we will check its performance against this validation dataset. We might then change some of the tuning options for building a decision tree model. We compare the new model against the old one based on its performance on the validation dataset. In this sense, the validation dataset is used during the modelling process to build the final model. Consequently, we will still have a biased estimate of the final performance of our model if we rely on the validation dataset for this measure.

The Testing dataset is then a "hold-out" dataset that has not been used at all during the model building. Once we have identified our "best" model based on using the validation dataset, the model's performance on the testing dataset is then assessed. This is then an estimate of the expected performance on any new data.

The fourth option uses the Full dataset for evaluating the model (the combined training, validation, and testing datasets). This might be seen to be useful only as a curiosity rather than for accurate performance.

Another option available as a data source is provided through the Enter choice. This is available when Score is chosen as the type of evaluation. In this case, a window will pop up to allow us to directly enter some data and have that "scored" by the model.

The final two options for the data source are a CSV File and an R Dataset. These allow data to be loaded into R from a CSV file, and the model can be evaluated on that dataset. Alternatively, for a data frame already available through R, the R Dataset will allow it to be chosen and the model evaluated on that.

### Risk Variable

The final row of options begins with an informative label that reports on the name of the Risk Variable chosen in the Data tab. The risk variable is used as a measure of how significant each observation is with respect to the target variable. For example, it might record the dollar value of the fraud or the amount of rain received "tomorrow." The risk chart makes use of this variable, if there is one, and it is included in the common area of the Evaluate tab for information purposes only.

### Scoring

The remaining options on the final row of options relate to scoring. Many models can predict an outcome or a probability for a particular outcome.

The Report option allows us to choose which we would like to see in the output when scoring. The Include option indicates whether to include all variables for each observation in the output or just the identifiers (those variables marked as having an Ident role on the Data tab).

### A Note on Cross-Validation

In Section 2.7, we introduced the concept of partitioning our dataset into three samples: the training, validation, and testing datasets. This concept was then further discussed in Section 3.1 and in the section above. In considering each of the modelling algorithms, we also touched on the evaluation of the models, using the validation dataset, as part of the model building process. We have stressed that the testing dataset is used as the final *unbiased* estimate of the performance of a model.

A related paradigm for evaluating the performance of our models is through the use of *cross-validation*. Indeed, some of the algorithms implemented in R will perform cross-validation for us and report a performance measure based on it. The decision tree algorithm using `rpart()` is an example.

Cross-validation is a simple concept. Given a dataset, we partition it into, perhaps, ten random sample subsets. Then we build a model using nine of those subsets, combined to form the training dataset. We can then measure the performance of the resulting model on the *hold-out dataset*. Then we can repeat this by selecting a different nine subsets to use as a training dataset. Once again, the remaining dataset will serve as a testing dataset. This can be repeated ten times to give us a measure of the expected performance of the resulting model.

A related concept, and one that we often find in the context of ensemble models, is the concept of *out-of-bag*, or *OOB*, measures of performance. We saw this concept when building a random forest model in Section 12.4. We might recall that in building a random forest we sample a subset of the full dataset. The subset is used as the training dataset. Thus, the remaining dataset can be used to test the performance of the resulting model.

In those cases where the R implementation of an algorithm provides its own performance measure, using cross-validation or out-of-bag estimates, we might choose not to create a validation dataset in Rattle. Instead, we can rely on the measure supplied by the algorithm as we build and fine-tune our models. The testing dataset remains useful then

to provide an unbiased measure once we have built our best models.

## 15.2 Measure of Performance

Quite a collection of measures has been developed over many years to gauge the performance of a model. The help page for `performance()` of **ROCR** (Sing et al., 2009) in R collects most of them together with a brief description, with 30 other measures listed. To review that list, using the R Console, simply ask for help:

```
> library(ROCR)
> help(performance)
```

We will discuss performance in the context of a binary classification model. This has been our focus with the *weather* dataset, predicting `No` or `Yes` for the variable `RainTomorrow`. For binary classification, we also often identify the predictions as positives or negatives. Thus, in terms of predicting whether it will rain tomorrow, `Yes` is the *positive* class and `No` is the *negative* class.

For an evaluation of a model, we apply the model to a dataset of observations with known actual outcomes (classes). The model will be used to predict the class for each observation. We then compare the predicted class with the actual class.

### Error Rate

The simplest measure of the performance of a model is the error rate. This is calculated as the proportion of observations for which the model incorrectly predicts the class with respect to the actual class. That is, we simply divide the number of misclassifications by the total number of observations.

### True and False Positives and Negatives

If our weather model predicts `Yes` in agreement with the actual outcome, then we refer to this as a **true positive**. Similarly, when they both agree on the negative, we refer to it as a **true negative**. On the other hand, when the model predicts `No` and the actual is `Yes`, then we have a **false negative**. Predicting a `Yes` when it is actually a `No` results in a **false positive**.

Often it is useful to differentiate in this way between the types of misclassification errors that a model makes. For example, in the context of our weather dataset, it makes a difference whether we have a false positive or a false negative. A false positive would predict that it will rain tomorrow when in fact it does not. The consequence is that I might take my umbrella with me but I won't need to use it—only a minor inconvenience.

However, a false negative predicts that it won't rain tomorrow but in fact it does rain. Relying on the model, I would not bother with an umbrella. Consequently, I am caught in the rain and get uncomfortably wet. The consequences of a false negative in this case are more significant for me than they are for a false positive.

Whether false positives or false negatives are more of an issue depends on the application. For medical applications, a false positive (e.g., falsely diagnosed with cancer) may be less of an issue than a false negative (e.g., the diagnosis of cancer being missed). Different model builders can deal with these situations in different ways. The decision tree algorithm, for example, can accept a loss matrix that gives different weights to the different outcomes. This will then bias the model building to avoid one type of error or the other.

Often, we are interested in the ratio of the number of true positives to the number of predicted positives. This is referred to as the true positive rate and similarly for the false positive rate and so on.

### Precision, Recall, Sensitivity, Specificity

The **precision** of a model is the ratio of the number of true positives to the total number of predicted positives (the sum of the true positives and the false positives). It is a measure of how accurate the positive predictions are, or how *precise* the model is in predicting.

The **recall** of a model is just another name for the true positive rate. It is a measure of how many of the actual positives the model can identify, or how much the model can *recall*. The recall is also known as the **sensitivity** of the model.

Another measure that often arises in the context of sensitivity is **specificity**. This is simply another name for the true negative rate.

### Other Measures

We will use and refine the measure we have introduced here in describing the various approaches to evaluating our models in the following sections. As the `help()` for **ROCR** indicates, we have very many to choose from, and which works best for the many different application areas is often determined through trial and error and experience.

## 15.3 Confusion Matrix

A confusion matrix (also known as an error matrix) is appropriate when predicting a categoric target (e.g., in binary classification models). We saw a number of confusion matrices in Chapter 2.

In Rattle, the Confusion Matrix is the default on the Evaluate tab. Clicking the Execute button will run the selected model(s) against the chosen dataset to predict the outcomes for each of the observations in that dataset. The predictions are compared with the actual observations, and the true and false positives and negatives are calculated.

Figure 15.2 illustrates this for the decision tree model using the *weather* dataset. We see in Figure 15.2 that six models have been built, and the Textview will show the confusion matrix for each of the selected models. A quick way to build each type of model is to choose the All option on the Model tab.

The confusion matrix displays the predicted versus the actual results in a table. The first table shows the actual counts, whilst the second table shows the percentages. For the decision tree applied to the validation dataset, there are 5 true positives and 39 true negatives, and so the model is correct for 44 observations out of 54. That is, the overall error rate is 10 out of 54, or 19%.

The false positives and false negatives have the same count. On five days we will get wet and on another five we will carry an umbrella with us unnecessarily.

If we scroll the text view window of the Evaluate tab, we can see the confusion-matrix-based performance measure for other models. The random forest appears to provide a slightly more accurate prediction, as we see in Figure 15.3.[1]

[1] Note that the results vary slightly between different deployments of R, particularly between 64 bit R, as here, and 32 bit R.

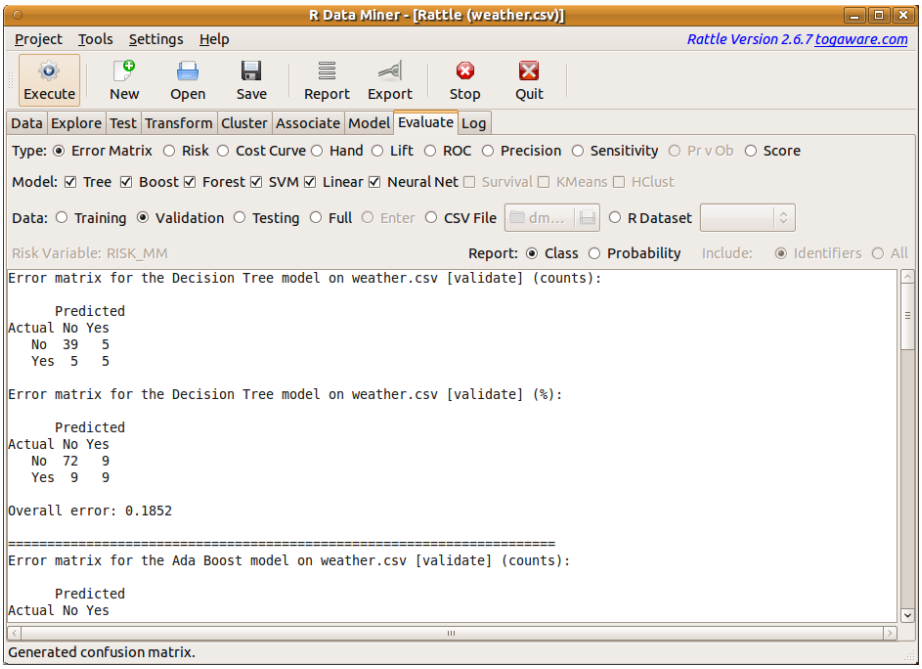


Figure 15.2: The Evaluate tab showing a confusion matrix.

The overall error rate for the random forest is 12%, with 4 true positives and 40 true negatives. Compared with the decision tree, there is one less day when we will get wet and three fewer days when we would unnecessarily carry our umbrella. We might instead look for a model that reduces the false negatives rather than the false positives. (Also remember that we should be careful when comparing such small numbers of observations—the differences won't be significant, though when using very large training datasets, as would be typical for data mining, we are in a better position to compare.)

## 15.4 Risk Charts

A *risk chart*, also known as a *cumulative gain chart*, provides another perspective on the performance of a binary classification model. Such a chart can be displayed by choosing the Risk option on the Evaluate tab. We will explain risk charts here using the *audit* dataset. The use of risk charts to evaluate models of fraud and noncompliance is more logical

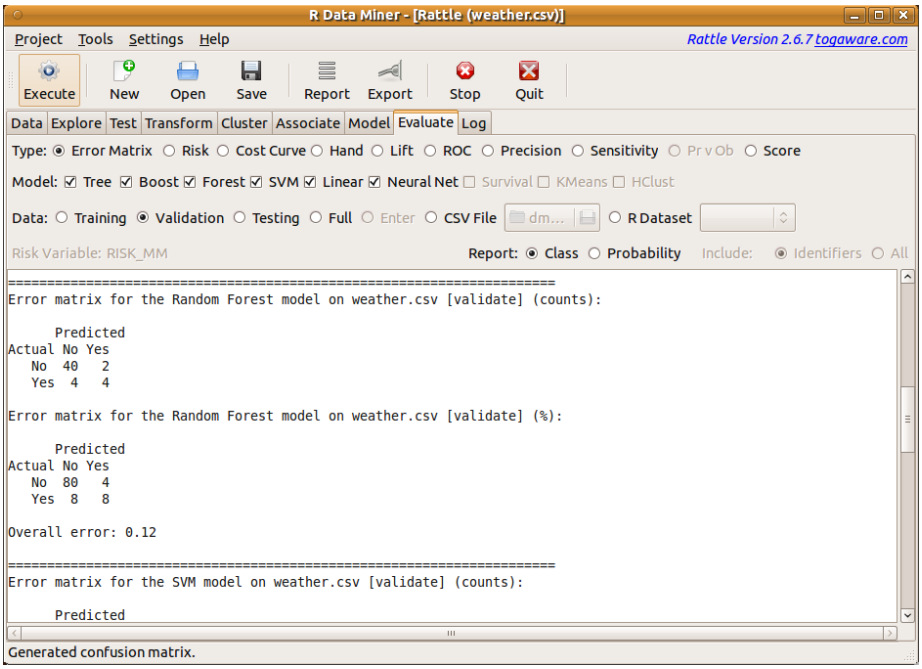


Figure 15.3: Confusion matrix for the random forest model.

than with the application to predicting rain.

The *audit* dataset (Section B.2) contains observations of taxpayers who have been audited together with the outcome of the audit: `No` or `Yes`. A positive outcome indicates that the taxpayer was required to update the tax return because of inaccuracies in the figures reported. A negative outcome indicates that the tax return required no adjustment. For each adjustment we also record its dollar amount (as the risk variable).

We can build a random forest model using this dataset, but we first need to load it into Rattle. To do so, go back to the Data tab and after loading **rattle**'s *weather* dataset click on the Filename chooser. We can then select the file `audit.csv`. Click on Execute to have the new dataset loaded. Then, from Rattle's Model tab, build a Forest and then request a Risk Chart from the Evaluate tab. The resulting risk chart is shown in Figure 15.4. To read the risk chart, we will pick a particular point and consider a specific scenario. The scenario is that of auditing taxpayers. Suppose we normally audit 100,000 taxpayers each year. Of those, only 24,000, let's say, end up requiring an adjustment to their tax return. We

call this the *strike rate*. That is, we strike 24,000 out of the 100,000 as being of interest—a strike rate of 24%.

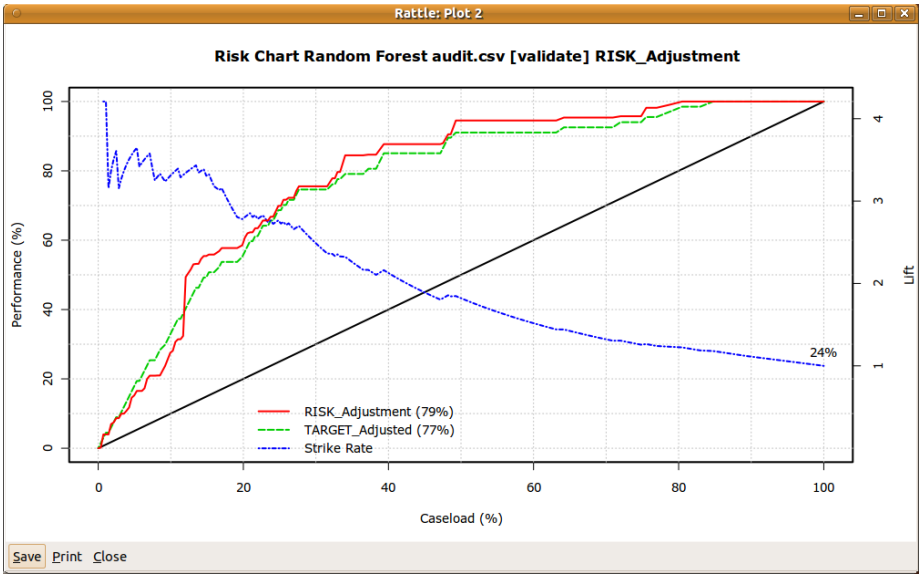


Figure 15.4: A risk chart for a random forest on the *audit* dataset.

Suppose our funding now allows us to audit only 50,000 taxpayers. If we were to randomly select 50% from the 100,000 taxpayers, then we would expect to identify just 50% of the actual taxpayers whose tax returns required an adjustment. That is, we would identify only 12,000 of the 24,000 tax returns requiring an adjustment from amongst the 50,000 taxpayers randomly selected. This random selection is represented by the diagonal line in the plot. A random 50% caseload (i.e., 50,000 cases) will deliver a 50% performance (i.e., only half of the known cases of interest will be found). We can think of this as the baseline—this is what the situation would be if we used random selection and no other model.

We now introduce our random forest model, which predicts the likelihood of a taxpayer's tax return requiring an adjustment. For each taxpayer, the model provides a score—the probability of the taxpayer's tax return requiring an adjustment. We can now prioritise our audits of taxpayers based on these scores so that taxpayers with a higher score are audited before taxpayers with a lower score. Once again, but now using this priority, we choose to audit only 50,000 taxpayers, but we select the 50,000 that have the highest risk scores.

The dashed green line of the plot indicates the performance achieved when using the model to prioritise the audits. For a 50% caseload, the performance is approximately 90%. That is, we expect to identify 90% of the tax returns requiring an adjustment. So 21,600 of the 24,000 known adjustments, from amongst the 50,000 taxpayers chosen, are expected to be identified. That is a significant improvement over the 12,000 from the 50,000 selected randomly. Indeed, as the blue line in the plot indicates, that provides a *lift* in performance of almost 2. That is, we are identifying almost twice as many tax returns requiring adjustment than we would expect if we were simply selecting taxpayers randomly.

In this light, the model provides quite a significant benefit. Note that we are not particularly concentrating on error rates as such but on the benefit we achieve in using the model to rank or prioritise our business processes. Whilst a lot of attention is often paid to simplistic measures of model performance, other factors usually come into play in deciding which model performs best.

Note also from the plot in Figure 15.4 that after we have audited about 85% of the cases (i.e., at a caseload of 85) the model achieves 100% performance. That is, the model has ensured that all tax returns requiring adjustment have been identified by the time we have audited 85,000 taxpayers. A conservative use of the model would then ensure nearly all required audits (i.e., 24,000) are performed, yet saving 15% of the effort previously required to identify all of the required audits. We also note that out of the 85,000 audits we are still unnecessarily auditing 61,000 taxpayers.

The solid red line of the risk chart often follows a path similar to that of the green line. It provides an indication of the measure of the size of the risk covered by the model. It is based on the variable identified as having a role as a Risk variable on the Data tab. In our case, it is the variable `RISK_Adjustment` and records the dollar amount of any adjustment made to a tax return. In that sense, it is a measure of the size of the risk.

The "risk" performance line is included for information. It has not been used in the modelling at all (though it could have been). Empirically, we often note that it sits near or above the "target" performance line. If it sits high above the target line, then the model is fortuitously identifying higher-risk cases earlier in the process, which is a useful outcome.

A risk chart can be displayed for any binary classification model built using Rattle. In comparing risk charts for different models, we are looking for a larger area under the curve. This generally means that the curve "closer" to the top left of the risk chart identifies a better-performing model than a curve that is closer to the baseline (diagonal line). Figure 15.5 illustrates the output when multiple models are selected, so that performances can be directly compared.

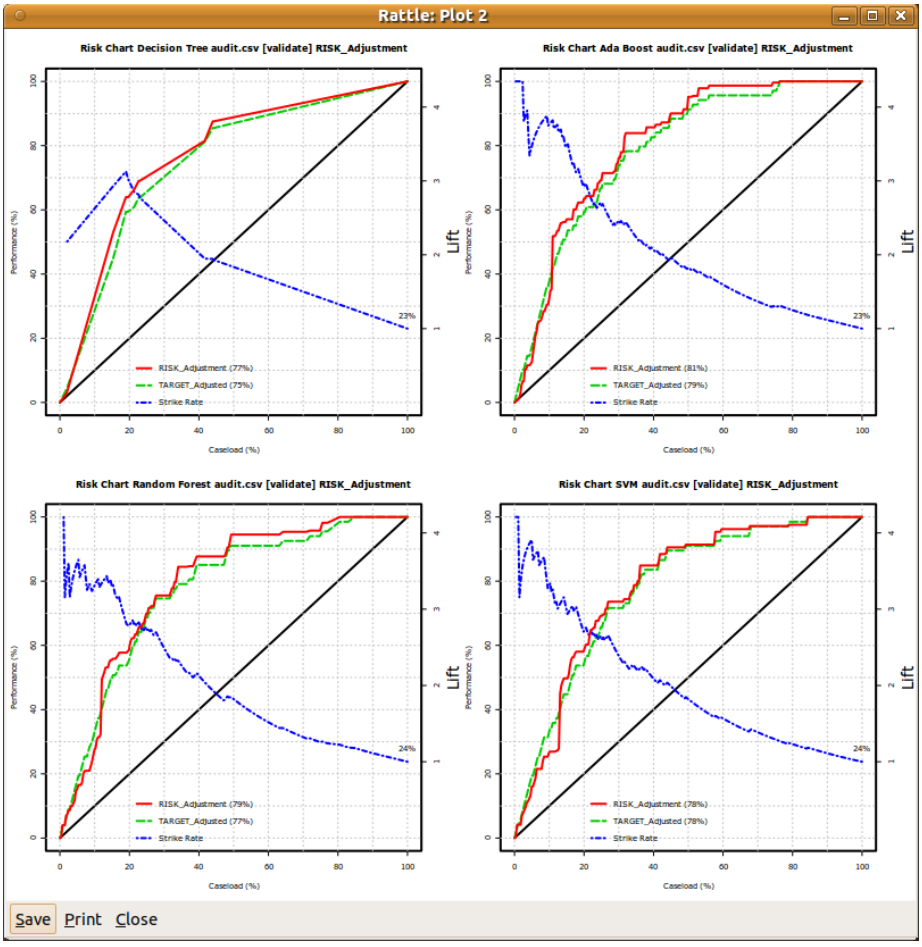


Figure 15.5: Four risk charts displayed to compare performances of multiple model builders on the *audit* dataset.

The plots generated by Rattle include a measure of the area under the curve in the legend of the plot. For Figure 15.4, the area under the

target line is 77%. That is, the line splits the plot into two regions, with 77% of the region below the line and 23% of the region above the line.

## 15.5 ROC Charts

Another set of common performance charts used in data mining are the ROC chart, sensitivity/specificity chart, lift chart, and precision/recall chart. The ROC is probably the most popular, but each can provide some different insights based essentially on the same performance data. The ROC has a form and interpretation similar to the risk chart, though it plots different measures on the axes. **ROCR** is used by Rattle to generate these charts.

We can differentiate each chart by the measures used on the two axes. An ROC chart plots the true positive rate against the false positive rate. The sensitivity/specificity chart plots the true positive rate against the true negative rate. The lift chart plots the relative increase in predictive performance against the rate of positive predictions. The precision/recall chart plots the proportion of true positives out of the positive predictions against the true positive rate.

## 15.6 Other Charts

A variety of other charts are also supported by Rattle. Some are experimental and implemented directly within Rattle rather than being available in other packages. This includes the Hand chart, which plots a number of measures proposed by David Hand, a senior statistician with a lot of credibility in the data mining community. Cost curves are another measure with quite a long history but have not become particularly popular.

The other performance chart, the Pr v Ob chart, is suitable for evaluating the performance of regression models. Such models predict a continuous outcome, like the dollar amount of a risk, rather than whether or not there is a risk. The Pr v Ob option produces a plot of the actual or observed values on the x-axis with the model-predicted values on the y-axis. A linear model is also fit to the predicted value, based on the actual value, and is displayed in the plot generated by Rattle. A diagonal line (predicted=observed) provides a benchmark as the perfect model

(i.e., perfect correlation between the predicted values and the observed values).

## 15.7 Scoring

The Evaluate tab also provides a Score option. Rather than running a model over some observations and generating a performance measure, the score option allows the predictions to be saved to a file so that we can perform our own analyses of the model using any other tools. Also, it allows us to effectively deploy the model so that we might score new data, save it to a file, and forward the results to the appropriate teams.

Rattle's Score option of the Evaluate tab, when Executed, will apply the selected model(s) to the specified dataset (training/validation/testing/full/CSV file/R dataset), saving the results to a CSV file. Once run, the scores (either the class, probability, or predicted value), together with any Ident variables (or optionally all variables), will be saved to a CSV file for action or further processing with other tools, as desired. A window will pop up to indicate the location and name of the file to which the scores have been saved.

The dataset that is scored must have exactly the same format as the dataset loaded for training the model. Rattle assumes the columns will be in the same order, and we might expect them to have the same names (noting that R is case-sensitive).

# Chapter 16

# Deployment

Once a model is developed and evaluated, and we have determined it to be suitable, we then need to deploy it. This is an often-overlooked issue in many data mining projects. It also seems to receive little attention when setting up a data mining capability within an organisation. Yet it is an important issue, as we need to ensure we obtain the benefit from the model.

In this chapter, we briefly consider a number of deployment options. We begin with considering a deployment in R. We also consider the conversion of our models into the Predictive Modelling Markup Language (PMML). This allows us to export our model to other systems, which includes systems that can convert the model into C code that can run as a stand-alone module.

## 16.1 Deploying an R Model

A simple approach to deployment is to use `predict()` to apply the model to a new dataset. We often refer to this as "scoring." Rattle's evaluation tab supports scoring with the Score option of the Evaluate tab. There are a number of options available. The first is whether to score the training dataset, the validation dataset, the testing dataset, or some dataset loaded from a CSV file (which must contain the exact same variables). Any number of models can be selected, and the results (either as predicted values or probabilities) are written to a CSV file.

Often, we will want to score a new dataset regularly as new observations become available. In this case, we will save the model for later use.

G. Williams, *Data Mining with Rattle and R: The Art of Excavating Data for Knowledge Discovery*, Use R, DOI 10.1007/978-1-4419-9890-3_16, © Springer Science+Business Media, LLC 2011

The Rattle concept of a project, as discussed in Section 2.8, is useful in such a circumstance. This will save the current state of Rattle (including the actual data and models built during a session). At a later time, this project can be loaded into a new instance of Rattle (running on the same host or even a different host and operating system). A new dataset can then be scored using the saved model. To do so, we do not need to start up the Rattle GUI but simply access the relevant model (e.g., `crs$rf`) and apply it to some new data using `predict()`.

As we see in Rattle's Log tab, below the surface, when we save and load projects, we are simply using `save()` and `load()`. These create a binary representation of the R objects, saving them to a file and then loading them into R.

A Rattle project can get quite large, particularly with large datasets. Larger files take longer to load, and for deploying a model it is often not necessary to keep the original data. So as we get serious about deployment, we might save just the model we wish to deploy. This is done using `save()` to save the actual model.

After building a random forest model in Rattle using the *weather* dataset, we might save the model to a file by using `save()` in the R Console:

```
> myrf <- crs$rf
> save(myrf, file="model01_110501.RData")
```

A warning message may be shown just to suggest that when reloading the binary file into a new instance of R, **rattle** might not have been loaded, and it is perhaps a good idea to do so. This is to ensure the objects that are saved are correctly interpreted by R.

We now want to simulate the application of the model to a new dataset. To do so, we might simply save the current dataset to a CSV file using `write.csv()`:

```
> write.csv(crs$dataset, file="cases_110601.csv")
```

We can then load the model into a different instance of R at a later time using `load()` and apply (i.e., use `predict()` on) the model (using a script based on the commands shown in Rattle's Log tab) to a new dataset. In this case, we then also write the results to another CSV file using `write.csv()`:

```
> library(randomForest)
> (load("model01_110501.RData"))

[1] "myrf"

> dataset <- read.csv("cases_110601.csv")
> pr <- predict(myrf, dataset, type="prob")[,2]
> write.csv(cbind(dataset, pr),
            file="scores_110601.csv",
            row.names=FALSE)
> head(cbind(Actual=dataset$TARGET_Adjusted, Predicted=pr))

  Predicted
1     0.688
2     0.712
3     0.916
4     0.164
5     0.052
6     0.016
```

We can see that the random forest model is doing okay on these few observations.

In practise, once model deployment has been approved, the model is deployed into a secure environment. We can then schedule the model to be applied regularly to a new dataset using a script that is very similar to that above. The dataset can be obtained from a data warehouse, for example, and the results populated back into the data warehouse. Other processes might then come into play to make use of the scores, perhaps to identify clients who need to be audited or to communicate to the weather forecaster the predicted likelihood of rain tomorrow.

## 16.2 Converting to PMML

An alternative approach to direct deployment within R is to export the model in some form so that it might be imported into other software for predictions on new data. Exporting a model to an open standard format facilitates this process. A model represented using this open standard representation can then be exported to a variety of other software or languages.

The Predictive Model Markup Language (Guazzelli et al., 2010, 2009) (PMML) provides such a standard language for representing data mining

models. PMML is an XML-based standard that is supported, to some extent, by the major commercial data mining vendors and many open source data mining tools.

The **pmml** package for R is responsible for exporting models from R to PMML. PMML models generated by Rattle using `pmml()` can be imported into a number of other products. The Export button (whilst displaying a model within the Model tab) will export a model as PMML.

We illustrate here the form that the PMML export takes. First, we again create a dataset object:

```
> library(rattle)
> weatherDS <- new.env()
> evalq({
    data <- weather
    nobs <- nrow(data)
    vars <- -grep('^(Date|Locat|RISK)', names(weather))
    set.seed(42)
    train <- sample(nobs, 0.7*nobs)
    form <- formula(RainTomorrow ~ .)
  }, weatherDS)
```

Next, we build the decision tree model:

```
> library(rpart)
> weatherRPART <- new.env(parent=weatherDS)
> evalq({
    model <- rpart(formula=form, data=data[train, vars])
  }, weatherRPART)
```

Now we can generate the PMML representation using `pmml()`. We then print some rows from the PMML representation of the model:

```
> library(pmml)
> p <- pmml(weatherRPART$model)
> r <- c(1:4, 7, 12, 35, 36, 69, 71, 137:139)
> cat(paste(strsplit(toString(p), "\n")[[1]][r],
            collapse="\n"))
<PMML version="3.2" xmlns="http://www.dmg.org/PMML-3_2"
    xmlns=...>
 <Header copyright="Copyright (c) 2011 gjw"
     description="RPart Decision Tree Model">
  <Extension name="user" value="gjw" extender="Rattle"/>
  <Application name="Rattle/PMML" version="1.2.27"/>
 <DataDictionary numberOfFields="21">
  <DataField name="MinTemp" optype="continuous" .../>
 </DataDictionary>
 <TreeModel modelName="RPart_Model" ...>
   <Node id="2" score="No" recordCount="204" ...>
     <SimplePredicate field="Pressure3pm"
     operator="greaterOrEqual" value="1011.9"/>
  </Node>
 </TreeModel>
</PMML>
```

## 16.3 Command Summary

This chapter has referenced the following R packages, commands, functions, and datasets:

| | | |
|---|---|---|
| `load()` | command | Load R objects from a file. |
| `pmml()` | function | Convert a model to PMML. |
| **pmml** | package | Supports conversion of many models. |
| `predict()` | function | Score a dataset using a model. |
| `save()` | command | Save R objects to a binary file. |