Classification and Regression by randomForest

Andy Liaw and Matthew Wiener

Introduction

Recently there has been a lot of interest in "ensemble learning" — methods that generate many classifiers and aggregate their results. Two well-known methods are boosting (see, e.g., Shapire et al., 1998) and bagging Breiman (1996) of classification trees. In boosting, successive trees give extra weight to points incorrectly predicted by earlier predictors. In the end, a weighted vote is taken for prediction. In bagging, successive trees do not depend on earlier trees — each is independently constructed using a bootstrap sample of the data set. In the end, a simple majority vote is taken for prediction.

Breiman (2001) proposed random forests, which add an additional layer of randomness to bagging. In addition to constructing each tree using a different bootstrap sample of the data, random forests change how the classification or regression trees are constructed. In standard trees, each node is split using the best split among all variables. In a random forest, each node is split using the best among a subset of predictors randomly chosen at that node. This somewhat counterintuitive strategy turns out to perform very well compared to many other classifiers, including discriminant analysis, support vector machines and neural networks, and is robust against overfitting (Breiman, 2001). In addition, it is very user-friendly in the sense that it has only two parameters (the number of variables in the random subset at each node and the number of trees in the forest), and is usually not very sensitive to their values.

The **randomForest** package provides an R interface to the Fortran programs by Breiman and Cutler (available at `http://www.stat.berkeley.edu/users/breiman/`). This article provides a brief introduction to the usage and features of the R functions.

The algorithm

The random forests algorithm (for both classification and regression) is as follows:

1. Draw $n_{\texttt{tree}}$ bootstrap samples from the original data.
2. For each of the bootstrap samples, grow an *unpruned* classification or regression tree, with the following modification: at each node, rather than choosing the best split among all predictors, randomly sample $m_{\texttt{try}}$ of the predictors and choose the best split from among those variables. (Bagging can be thought of as the special case of random forests obtained when $m_{\texttt{try}} = p$, the number of predictors.)
3. Predict new data by aggregating the predictions of the $n_{\texttt{tree}}$ trees (i.e., majority votes for classification, average for regression).

An estimate of the error rate can be obtained, based on the training data, by the following:

1. At each bootstrap iteration, predict the data not in the bootstrap sample (what Breiman calls "out-of-bag", or OOB, data) using the tree grown with the bootstrap sample.
2. Aggregate the OOB predictions. (On the average, each data point would be out-of-bag around 36% of the times, so aggregate these predictions.) Calcuate the error rate, and call it the OOB estimate of error rate.

Our experience has been that the OOB estimate of error rate is quite accurate, given that enough trees have been grown (otherwise the OOB estimate can bias upward; see Bylander (2002)).

Extra information from Random Forests

The **randomForest** package optionally produces two additional pieces of information: a measure of the importance of the predictor variables, and a measure of the internal structure of the data (the proximity of different data points to one another).

Variable importance This is a difficult concept to define in general, because the importance of a variable may be due to its (possibly complex) interaction with other variables. The random forest algorithm estimates the importance of a variable by looking at how much prediction error increases when (OOB) data for that variable is permuted while all others are left unchanged. The necessary calculations are carried out tree by tree as the random forest is constructed. (There are actually four different measures of variable importance implemented in the classification code. The reader is referred to Breiman (2002) for their definitions.)

proximity measure The (i, j) element of the proximity matrix produced by **randomForest** is the fraction of trees in which elements i and j fall in the same terminal node. The intuition is that "similar" observations should be in the same terminal nodes more often than dissimilar ones. The proximity matrix can be used

to identify structure in the data (see Breiman, 2002) or for unsupervised learning with random forests (see below).

Usage in R

The user interface to random forest is consistent with that of other classification functions such as `nnet()` (in the **nnet** package) and `svm()` (in the **e1071** package). (We actually borrowed some of the interface code from those two functions.) There is a formula interface, and predictors can be specified as a matrix or data frame via the `x` argument, with responses as a vector via the `y` argument. If the response is a factor, `randomForest` performs classification; if the response is continuous (that is, not a factor), `randomForest` performs regression. If the response is unspecified, `randomForest` performs unsupervised learning (see below). Currently `randomForest` does not handle ordinal categorical responses. Note that categorical predictor variables must also be specified as factors (or else they will be wrongly treated as continuous).

The `randomForest` function returns an object of class `"randomForest"`. Details on the components of such an object are provided in the online documentation. Methods provided for the class includes `predict` and `print`.

A classification example

The Forensic Glass data set was used in Chapter 12 of MASS4 (Venables and Ripley, 2002) to illustrate various classification algorithms. We use it here to show how random forests work:

```
> library(randomForest)
> library(MASS)
> data(fgl)
> set.seed(17)
> fgl.rf <- randomForest(type ~ ., data = fgl,
+    mtry = 2, importance = TRUE,
+    do.trace = 100)
 100: OOB error rate=20.56%
 200: OOB error rate=21.03%
 300: OOB error rate=19.63%
 400: OOB error rate=19.63%
 500: OOB error rate=19.16%
> print(fgl.rf)
Call:
 randomForest.formula(formula = type ~ .,
   data = fgl, mtry = 2, importance = TRUE,
   do.trace = 100)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 19.16%
Confusion matrix:
      WinF WinNF Veh Con Tabl Head class.error
WinF    63     6   1   0    0    0   0.1000000
WinNF    9    62   1   2    2    0   0.1842105
Veh      7     4   6   0    0    0   0.6470588
Con      0     2   0  10    0    1   0.2307692
Tabl     0     2   0   0    7    0   0.2222222
Head     1     2   0   1    0   25   0.1379310
```

We can compare random forests with support vector machines by doing ten repetitions of 10-fold cross-validation, using the `errorest` functions in the **ipred** package:

```
> library(ipred)
> set.seed(131)
> error.RF <- numeric(10)
> for(i in 1:10) error.RF[i] <-
+     errorest(type ~ ., data = fgl,
+     model = randomForest, mtry = 2)$error
> summary(error.RF)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.1869  0.1974  0.2009  0.2009  0.2044  0.2103
> library(e1071)
> set.seed(563)
> error.SVM <- numeric(10)
> for (i in 1:10) error.SVM[i] <-
+     errorest(type ~ ., data = fgl,
+     model = svm, cost = 10, gamma = 1.5)$error
> summary(error.SVM)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.2430  0.2453  0.2523  0.2561  0.2664  0.2710
```

We see that the random forest compares quite favorably with SVM.

We have found that the variable importance measures produced by random forests can sometimes be useful for model reduction (e.g., use the "important" variables to build simpler, more readily interpretable models). Figure 1 shows the variable importance of the Forensic Glass data set, based on the `fgl.rf` object created above. Roughly, it is created by

```
> par(mfrow = c(2, 2))
> for (i in 1:4)
+   plot(sort(fgl.rf$importance[,i], dec = TRUE),
+   type = "h", main = paste("Measure", i))
```

We can see that measure 1 most clearly differentiates the variables. If we run random forest again dropping `Na`, `K`, and `Fe` from the model, the error rate remains below 20%.

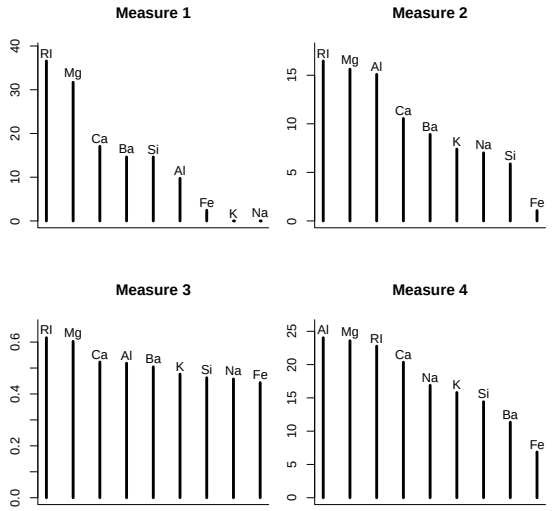

Figure 1: Variable importance for the Forensic Glass data.

The gain can be far more dramatic when there are more predictors. In a data set with thousands of predictors, we used the variable importance measures to select only dozens of predictors, and we were able to retain essentially the same prediction accuracy. For a simulated data set with 1,000 variables that we constructed, random forest, with the default $m_{\texttt{try}}$, we were able to clearly identify the only two informative variables and totally ignore the other 998 noise variables.

A regression example

We use the Boston Housing data (available in the **MASS** package) as an example for regression by random forest. Note a few differences between classification and regression random forests:

- The default $m_{\texttt{try}}$ is $p/3$, as opposed to $p^{1/2}$ for classification, where p is the number of predictors.
- The default `nodesize` is 5, as opposed to 1 for classification. (In the tree building algorithm, nodes with fewer than `nodesize` observations are not splitted.)
- There is only one measure of variable importance, instead of four.

```
> data(Boston)
> set.seed(1341)
> BH.rf <- randomForest(medv ~ ., Boston)
> print(BH.rf)
Call:
 randomForest.formula(formula = medv ~ .,
     data = Boston)
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 4

          Mean of squared residuals: 10.64615
                    % Var explained: 87.39
```

The "mean of squared residuals" is computed as

$$\text{MSE}_{\text{OOB}} = n^{-1} \sum_{1}^{n} \{y_i - \hat{y}_i^{\text{OOB}}\}^2,$$

where $\hat{y}_i^{\text{OOB}}$ is the average of the OOB predictions for the ith observation. The "percent variance explained" is computed as

$$1 - \frac{\text{MSE}_{\text{OOB}}}{\hat{\sigma}_y^2},$$

where $\hat{\sigma}_y^2$ is computed with n as divisor (rather than $n-1$).

We can compare the result with the actual data, as well as fitted values from a linear model, shown in Figure 2.

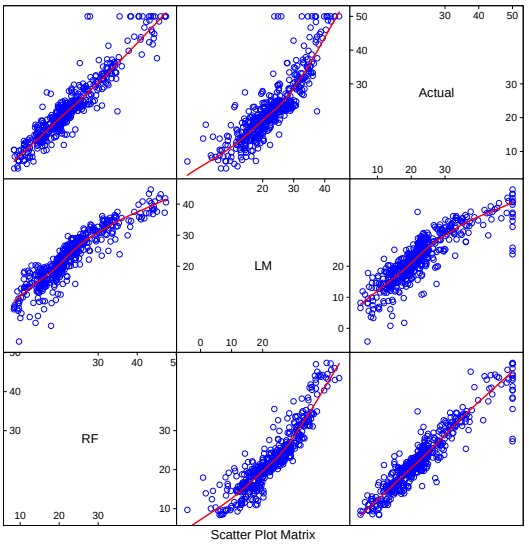

Figure 2: Comparison of the predictions from random forest and a linear model with the actual response of the Boston Housing data.

An unsupervised learning example

Because random forests are collections of classification or regression trees, it is not immediately apparent how they can be used for unsupervised learning. The "trick" is to call the data "class 1" and construct a "class 2" synthetic data, then try to classify the combined data with a random forest. There are two ways to simulate the "class 2" data:

1. The "class 2" data are sampled from the product of the marginal distributions of the variables (by independent bootstrap of each variable separately).

2. The "class 2" data are sampled uniformly from the hypercube containing the data (by sampling uniformly within the range of each variables).

The idea is that real data points that are similar to one another will frequently end up in the same terminal node of a tree — exactly what is measured by the proximity matrix that can be returned using the `proximity=TRUE` option of `randomForest`. Thus the proximity matrix can be taken as a similarity measure, and clustering or multi-dimensional scaling using this similarity can be used to divide the original data points into groups for visual exploration.

We use the `crabs` data in MASS4 to demonstrate the unsupervised learning mode of `randomForest`. We scaled the data as suggested on pages 308–309 of MASS4 (also found in lines 28–29 and 63–68 in '$R_HOME/library/MASS/scripts/ch11.R'), resulting the the `dslcrab` data frame below. Then run `randomForest` to get the proximity matrix. We can then use `cmdscale()` (in package **mva**) to visualize the 1−proximity, as shown in Figure 3. As can be seen in the figure, the two color forms are fairly well separated.

```
> library(mva)
> set.seed(131)
> crabs.prox <- randomForest(dslcrabs,
+   ntree = 1000, proximity = TRUE)$proximity
> crabs.mds <- cmdscale(1 - crabs.prox)
> plot(crabs.mds, col = c("blue",
+     "orange")[codes(crabs$sp)], pch = c(1,
+     16)[codes(crabs$sex)], xlab="", ylab="")
```

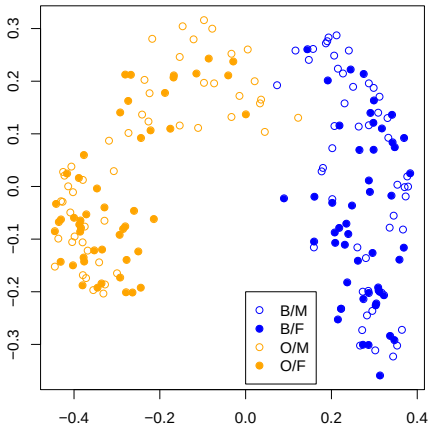

Figure 3: The metric multi-dimensional scaling representation for the proximity matrix of the `crabs` data.

There is also an `outscale` option in `randomForest`, which, if set to `TRUE`, returns a measure of "outlyingness" for each observation in the data set. This measure of outlyingness for the jth observation is calculated as the reciprocal of the sum of squared proximities between that observation and all other observations *in the same class*. The Example section of the help page for `randomForest` shows the measure of outlyingness for the Iris data (assuming they are unlabelled).

Some notes for practical use

- The number of trees necessary for good performance grows with the number of predictors. The best way to determine how many trees are necessary is to compare predictions made by a forest to predictions made by a subset of a forest. When the subsets work as well as the full forest, you have enough trees.

- For selecting $m_{\texttt{try}}$, Prof. Breiman suggests trying the default, half of the default, and twice the default, and pick the best. In our experience, the results generally do not change dramatically. Even $m_{\texttt{try}} = 1$ can give very good performance for some data! If one has a very large number of variables but expects only very few to be "important", using larger $m_{\texttt{try}}$ may give better performance.

- A lot of trees are necessary to get stable estimates of variable importance and proximity. However, our experience has been that even though the variable importance measures may vary from run to run, the ranking of the importances is quite stable.

- For classification problems where the class frequencies are extremely unbalanced (e.g., 99% class 1 and 1% class 2), it may be necessary to change the prediction rule to other than majority votes. For example, in a two-class problem with 99% class 1 and 1% class 2, one may want to predict the 1% of the observations with largest class 2 probabilities as class 2, and use the smallest of those probabilities as threshold for prediction of test data (i.e., use the `type='prob'` argument in the `predict` method and threshold the second column of the output). We have routinely done this to get ROC curves. Prof. Breiman is working on a similar enhancement for his next version of random forest.

- By default, the entire forest is contained in the `forest` component of the `randomForest` object. It can take up quite a bit of memory for a large data set or large number of trees. If prediction of test data is not needed, set the argument `keep.forest=FALSE` when running `randomForest`. This way, only one tree is kept in memory at any time, and thus lots of

memory (and potentially execution time) can be saved.

- Since the algorithm falls into the "embarrassingly parallel" category, one can run several random forests on different machines and then aggregate the `votes` component to get the final result.

Acknowledgment

We would like to express our sincere gratitute to Prof. Breiman for making the Fortran code available, and answering many of our questions. We also thank the reviewer for very helpful comments, and pointing out the reference Bylander (2002).

Bibliography

L. Breiman. Bagging predictors. *Machine Learning*, 24 (2):123–140, 1996. 18

L. Breiman. Random forests. *Machine Learning*, 45(1): 5–32, 2001. 18

L. Breiman. Manual on setting up, using, and understanding random forests v3.1, 2002. http://oz.berkeley.edu/users/breiman/Using_random_forests_V3.1.pdf. 18, 19

T. Bylander. Estimating generalization error on two-class datasets using out-of-bag estimates. *Machine Learning*, 48:287–297, 2002. 18, 22

R. Shapire, Y. Freund, P. Bartlett, and W. Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *Annals of Statistics*, 26 (5):1651–1686, 1998. 18

W. N. Venables and B. D. Ripley. *Modern Applied Statistics in S*. Springer, 4th edition, 2002. 19

Andy Liaw
Matthew Wiener
Merck Research Laboratories
andy_liaw@merck.com
matthew_wiener@merck.com

Some Strategies for Dealing with Genomic Data

by R. Gentleman

Introduction

Recent advances in molecular biology have enabled the exploration of many different organisms at the molecular level. These technologies are being employed in a very large number of experiments. In this article we consider some of the problems that arise in the design and implementation of software that associates biological meta-data with the experimentally obtained data. The software is being developed as part of the Bioconductor project www.bioconductor.org.

Perhaps the most common experiment of this type examines a single species and assays samples using a single common instrument. The samples are usually homogeneous collection of a particular type of cell. A specific example is the study of mRNA expression in a sample of leukemia patients using the Affymetrix U95A v2 chips Affymetrix (2001). In this case a single type of human cell is being studied using a common instrument.

These experiments provide estimates for thousands (or tens of thousands) of sample specific features. In the Affymetrix experiment described previously data on mRNA expression for approximately 10,000 genes (there are 12,600 probe sets but these correspond to roughly 10,000 genes). The experimental data, while valuable and interesting require additional biological meta-data to be correctly interpreted. Considering once again the example we see that knowledge of chromosomal location, sequence, participation in different pathways and so on provide substantial interpretive benefits.

Meta-data is not a new concept for statisticians. However, the scale of the meta-data in genomic experiments is. In many cases the meta-data are larger and more complex than the experimental data! Hence, new tools and strategies for dealing with meta-data are going to be needed. The design of software to help manage and manipulate biological annotation and to relate it to the experimental data will be of some importance. As part of the Bioconductor project we have made some preliminary studies and implemented some software designed to address some of these issues. Some aspects of our investigations are considered here.