Manual--Setting Up, Using, And Understanding Random Forests V4.0

The V4.0 version of random forests contains some modifications and major additions to Version 3.3. The additions are :

a) replacement of missing values.
b) a method to balance error in class unbalanced data sets.
c) data that can be used to see how the variables relate to the classification.
d) efficient handling of categoricals with a large number of values.

The basics of this program works are in the paper "Random Forests" Its available on the same web page as this manual. It was recently published in the Machine Learning. Journal. Please report bugs either to Leo Breiman (leo@stat.berkeley.edu) or Adele Cutler (adele@sunfs.math.usu.edu)

The program is written in extended Fortran 77 making use of a number of VAX extensions. It runs on Solaris f77 and on Absoft Fortran 77 (Windows and Mac) , the Lahey Windows compiler, and on the free g77 compiler for Linux., but may have hang ups on other f77 compilers. If you find such problems and fixes for them, please let us know.

Features of Random Forests

i) It is an excellent classifier--comparable in accuracy to support vector machines.

ii) It generates an internal unbiased estimate of the generalization error as the forest building progresses.

iii) It has an effective method for estimating missing data and maintains accuracy when up to 80% of the data are missing.

iv) It has a method for balancing error in unbalanced class population data sets.

v) Generated forests can be saved for future use on other data.

vi) It gives estimates of what variables are important in the classification.

vii) Output is generated that gives information about the relation between the variables and the classification

viii) It computes proximities between pairs of cases that can be used in clustering, locating outliers, or by scaling, give interesting views of the data.

ix) The capabilities of vii) above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection. The missing value replacement algorithm also extends to unlabeled data.

The first part of this manual contains instructions on how to set up a run of random forests V4.0. The second part contains the notes on the features of random forests V4.0 and how they work. The appendix has details on how to save forests and run future data down them.

Runs can be set up with no knowledge of FORTRAN 77. The user is required only to set the right switches and give names to input and output files. This is done at the start of the program.

This is what the user sees at the top of the program with the parameters set up for a run on the hepatitus.txt, a data set supplied with v4.

```
c         line #1 information about the data
c         line #2 setting up the run
c         line #3 variable importance options
c         line #4 options using proximities
c       line #5 filling in missing data
c       line #6 setting up parallel coordinates
c       line #7 saving and rerunning the forest
c       line #8 some output controls
```

set all parameters

```
      parameter(
     1 mdim=19,nsample0=155,nclass=2,
     &
     1 maxcat=2,ntest=1,label=0,
     &
     2 jbt=500,  mtry=3,look=10,ndsize=1,iaddcl=0,jclasswt=1,
     &
     3 imp=1,  impfast=0,
     &
     4 ndprox=0,noutlier=0,iscale=0,mdimsc=1,
     &
     5 missquick=1,missright=0,code=-999,
     &
     6 llcoor=0,ncoor=50,
     &
     7 isaverf=0,isavepar=0,irunrf=0,ireadpar=0,
     &
     8 isumout=0,infoutr=0,infouts=0,iproxout=0,iclassout=1,
```

I. ***Setting Parameters***

The first nine lines following the parameter statement need to be filled in by the user.

Line 1 Describing The Data

mdim=number of variables
nsample0=number of cases (examples or instances) in the data
nclass=number of classes
maxcat=the largest number of values assumed by a categorical variable in the data
ntest=the number of cases in the test set. NOTE: Put ntest=1 if there is no test set. Putting ntest=0 may cause compiler complaints.
label=0 if the test set has no class labels, 1 if the test set has class labels.

If their are no categorical variables in the data set maxcat=1. If there are categorical variables, the number of categories assumed by each categorical variable has to be specified in an integer vector called cat, i.e. setting cat(5)=7 implies that the 5th variable is a categorical with 7 values. If maxcat=1, the values of cat are automatically set equal to one. If not, the user must fill in the values of cat in the early lines of code. The categories are set a few lines down--here is the code:

```
c     SET CATEGORICAL VALUES++++++++++++++++++++++++++++++++++++

        do m=1,mdim
       cat(m)=1
      end do

      if(maxcat.ge.2  then
      fill in for all variables with cat(m)>1
      do  m=2,13
       cat(m)=2
       end do
      cat(19)=2
      end  if
```

If you run on a data set with maxcat=1, and, say, 10 variables without commenting out (c in front of) the lines above following if(maxcat.ge.2)--your compiler may generate an array boundary error.

For a J-class problem, random forests expects the classes to be numbered 1,2, ...,J. For an L valued categorical, it expects the values to be numbered 1,2, ... ,L.

A test set can have two purposes--first: to check the accuracy of RF on a test set. The error rate given by the internal estimate will be very close to the test set error unless the test set is drawn from a different distribution. Second: to get predicted classes for a set of data with unknown class labels. In both cases the test set must have the same format as the training set. If there is no class label for the test set, assign each case in the test set label class #1, i.e. put cl(n)=1, and set label=0. Else set label=1.

Line 2 Setting up the run

Line 2 Setting Up the Run

jbt=number of trees to grow
mtry=number of variables randomly selected at each node
look=how often you want to check the prediction error
ndsize=minimum node size
iaddcl=1 adds a synthetic second class
jclasswt=1 assigns weights to different classes

jbt:
this is the number of trees to be grown in the run. Don't be stingy--random forests produces trees very rapidly, and it does not hurt to put in a large number of trees. If you want auxiliary information like variable importance or proximities grow a lot of trees--say a 1000 or more. Sometimes, I run out to 5000 trees if there are many variables and I want the variables importances to be stable.

mtry: this is the only parameter that requires some judgment to set, but forests isn't too sensitive to its value as long as it's in the right ball park. I have found that setting mtry equal to the square root of mdim gives generally near optimum results. My advice is to begin with this value and try a value twice as high and half as low monitoring the results by setting look=1 and checking the internal test set error for a small number of trees. With many noise variables present, mtry has to be set higher.

look: random forests carries along an internal estimate of the test set error as the trees are being grown. This estimate is outputted to the screen every look trees. Setting look=10, for example, gives the internal error output every tenth tree added. If there is a labeled test set, it also gives the test set error. Setting look=jbt+1 eliminates the output. Do not be dismayed to see the error rates fluttering around slightly as more trees are added. Their behavior is analogous to the sequence of averages of the number of heads in tossing a coin.

ndsize: setting this to the value k means that no node with fewer than k cases will be split. The default that always gives good performances is ndsize=1. In large data sets, memory requirements will be less and speed enhanced if ndsize is set larger. Usually, this results in only a small loss of accuracy for large data sets.

iaddcl: If the data has no class labels, addition of a synthetic class enables it to be treated as a two-class problem with nclass=2. Setting iaddcl=1 forms the synthetic class by independent sampling from each of the univariate distributions of the variables in the original data. Keep iaddcl=0 for labeled data.

jclasswt: In some data sets, one class may have a significantly higher error rate than the others. For example, if a class has a population much smaller than the other classes, it will tend to be badly missclassified To remedy, weight up the high error rate class..

Set jclasswt=1 and go down to the code that reads::

```
c       GIVE CLASS WEIGHTS

        do  j=1,nclass
        classwt(j)=1
        end  do

        if(jclasswt.eq.1)  then
c       fill in for each class with weight >1
        classwt(1)=3
        end  if
```

To equalize the error rates, adjust the weights and check the oob error.

Line 3 Options on Variable Importance

i**mp**=1 turns on the variable importances methods described below, and computes and prints the following columns to a file

i) variable number
ii) variable importance computed as: 100* the change in the margins averaged over all cases

impfast=1 computes and prints out only the gini increase by variable for the run This is a very fast option while imp=1 causes considerably more computation.

Line 4 Options based on proximities

iprox=1 turns on the computation of the intrinsic proximity measures between any two cases . This has to be turned on for the following options to work.

noutlier=1 computes an outlyingness measure for all cases in the data. If iaddcl=1 then the outlyingness measure is computed only for the original data. If it exceeds a certain level for a case, the output has the columns :

i) class
ii) case number
iii) measure of outlyingness

iscale=1 computes scaling coordinates based on the proximity matrix. If iaddcl is turned on, then the scaling is outputted only for the original data. The output has the columns:

i) case number
i) true class
iii) predicted class.
iv) 0 if ii)=iii), 1 otherwise
v-v+msdim) scaling coordinates

mdimsc is the number of scaling coordinates to be extracted. Usually 4-5 is sufficient.

Line 5 Replacing Missing Values

To replace missing values there must be a single real or integer code that identifies missing values. This has to be given after the word code. i.e. code=1000.0 implies that all missing values are marked by the value 1000.0.

missquick: this replaces all missing values by the median of the non-missing values in their column, if real, and by the most numerous value in their column if categorical. This is fast and if there are only small amounts of missing data will serve.

missright: this option starts with missfast but then iterates using proximities and does an effective replacement even with large amounts of missing data. The data file with the replaced values is then downloaded to a file (set infoutr=1). If missright is set to 1, then ndprox must also be set to one.

Set jbt between 100 and 200. The oob estimates will appear at look intervals. At the finish of each forest construction, nrep= -- appears to announce the number of repetitions, and there is a pause while the missing data is being re-estimated. Then the new cycle begins again. There is a stopping rule that allows at most 6 repetitions. Our exp[erience is that the oob error rate at the end overestimates the test set error rate by 10-20%.

comment If there is an unlabeled test set with missing values, at present the way to fill these in effectively is to add a synthetic second class and use missquick, missright on this two class problem. Missing values will be filled in only from the original data.

If missquick is set to one but missright to zero then both the training set and the test set will be filled in by missquick using the medians and most probably values on the test set that were evaluated in the training set.

RF is robust with respect to missing values. If there is less than 20% missing values, the best approach is to use missquick only. Its fast. Missright is relatively slow, requiring up to 6 iterations of forest growing.

Line 6 The Relation between Variables and Classification.

llcoor/ncoor. Putting llcoor=1 and ncoor=50 (for instance) causes a write to file of each variable value for those 50 cases that have the highest votes for class #1. Similarly for class#2, etc. This enables the user to see which values of the variables are most closely associated with the recognition of each class.

The output file has the following format: the columns are of length nclass*mdim. The first column consists of each class label repeated mdim times. The second column has the variable numbers from 1 to mdim repeated nclass times. Column 3 contains the variables values for the first of the ncoor cases for each class--column 4 the variables values for the second of the ncoor cases for each class, and so on until column 3+ncoor. The last three columns hold the 25th, 50th and 75th percentiles for their respective rows.

Line 7 Saving the forest

isaverf=1 saves all the trees in the forest to a file named eg. A.

isavepar=1 creates a file B that contains the parameters used in the run and allows up to 500 characters of text description about the run.

irunrf=1 reads file A and runs new data down the forest.

ireadpar =1 reads file B and prints it to the screen

The files names required for A and B output are entered at the beginning of the program. Similarly, the reading of files of old A,B is done at the beginning of the program. See the appendix for more details.

Line 7 Output Controls

Note: user must supply file names for all output listed below or send it to the screen.

nsumout=1 writes out summary data to the screen. This includes errors rates and the confusion matrix

infoutr=1 prints the following columns to a file
i) case number
ii) true class label
iii) predicted class label
iv) margin=true class prob. minus the max of the other class prob.
v)-v+nclass) proportion of votes for each class

infouts=1 prints the following columns to a file
i) case number in test set
ii) true class (true class=1 if data is unlabeled)
iii) predicted class
iv-iv+nclass) proportion of votes for each class

iproxout=1 prints to file
i) case #1 number
ii) case #2 number
iii) proximity between case #1 and case #2

jclassout: if jclassout=1, the missclassification rates by class are printed out at every look trees. If jclassout=0, only the overall error rate is printed out.

Reading in the Data

Occurs right after the dimensioning of arrays. Here is an example

```
open(9,file='satimage.tra',status='old')
do k=1,nsample
read(9,*) (x(j,k),j=1,mdim),cl(k)
if(cl(k).eq.7) cl(k)=6
end do
close(9)
```

(turning class #7 into #6 was done because there were zero class#6 originally)

Here is an example of reading in test set data:

```
open(7,file='satimage.tes',status='old')
do k=1,ntest
read(7,*) (xts(j,k),j=1,mdim),clts(k)
if(clts(k).eq.7) clts(k)=6
end do
close(7)
```

Note: when reading in data to run down a stored tree, use the training set notation, i,e. x((j,k),cl(k)

Specifying the Names of Files

The user must specify the names of the files containing the data and the names of the files to receive output data. All of these files are listed immediately following the dimensioning of arrays and the reading in of data.. For those that are applicable, remove the comment "c" in front of the line and fill in the name.

Outline Of How Random Forests Works

Usual Tree Construction--Cart

Node=subset of data. The root node contains all data.

At each node, search through all variables to find best split into two children nodes.

Split all the way down and then prune tree up to get minimal test set error.

Random Forests Construction

Root node contains a bootstrap sample of data of same size as original data. A different bootstrap sample for each tree to be grown.

An integer mtry is fixed, mtry<<number of variables. mtry is the **only** parameter that needs to be specified. Default is the square root of number of variables.

At each node, mtry of the variables are selected at random. Only these variables are searched through for the best split. The largest tree possible is grown and is not pruned.

The forest consists of N trees. To classify a new object having coordinates **x**, put **x** down each of the N trees. Each tree gives a classification for **x.**

The forest chooses that classification having the most out of N votes.

remarks: Random forests does not overfit. You can run as many trees as you want. Also, it is fast. Running on a 250mhz machine, the current version with a training set with 800 cases, 8 variables, and mtry=1, constructs each tree in .1 seconds. On a training set with 50000 cases, 100 variables, and mtry=10, each tree is constructed in 12 seconds on an 800mhz machine.

For large data sets, if proximities are not required, the major memory requirement is the storage of the data itself, and the three integer arrays a,at,b. If there are less than 64,000 cases, these latter three may be declared integer*2 (non-negative). Then the total

storage requirement is about three times the size of the data set. If proximities are calculated, storage requirements go up by the square of the number of cases times eight bytes (double precision).

Three Useful Properties

There are three properties that give random forests its variety of tools.

i) The bootstrap training sample on which each tree is grown omits about 1/3rd of the cases. These are called out-of-bag (oob). These oob cases turn out to be useful.

Put back into the associated tree they form a test sample that gives the ongoing oob estimate of test set error. If an individual variable in the oob cases is randomly permuted before being put back into the tree, then the decrease in the estimated margins (see below) is an indication of how inportant that variable is.

ii) After each tree is grown, the entire training set is run down the tree. If two cases k and n wind up in the same terminal node, then their proximity measure prox(k,n) is increased by one. At the end of the forest construction, these are normalized by dividing by the number of trees.

The proximities give an intrinsic measure of similarities between cases. They are used in replacing missing values by estimating each missing value by a proximity weighted sum over the non-missing values. Then using the replaced values, run RF again to get new proximiites and repeat.

The proximities are also used to locate outliers--using the definition of an outlier as a case that only has weak similarities to the other cases.

The most useful property is that 1-prox(k,n) form Euclidean distances in a high dimensional space. They can be projected down onto a low dimensional space using metric scaling. This gives informative views of the data.

iii) For each case n, the proportion of votes that n gets in the forest for class j is q(j,n). The higher that q(j,n) is for a class j, the more "confident" its classification.

Ordinarily, a case is classified into the class j that maximizes q(j,n). The methods of distributing classification errors and increasing coverage of small classes are based on finding suitable thresholdings of the q(j,n)

To understand the differences in variables that drive the classification, we extract for each class j, those ncoor cases having the highest values of q(j,n). These are then contrasted with each other to see the values of the variables that are discriminating between the classes.

Random Forests Tools

The design of random forests is to give the user a good deal of information about the data besides an accurate prediction. The information includes:

a) Test set error rate.

b) Variable importance

c) Intrinsic proximities between cases

d) Scaling coordinates based on the proximities

e) Outlier detection

f) Variable Effect on Classes

The following explains how these work and give applications, both for labeled and unlabeled data.

Test Set Error Rate

In random forests, there is no need for cross-validation or a separate test set to get an unbiased estimate of the test set error. It is gotten internally, during the run, as follows:

Each tree is constructed using a different bootstrap sample from the original data. About one-third of the cases are left out of the bootstrap sample and not used in the construction of the kth tree.

Test Set Error Rate

Put each case left out in the construction of the kth tree down the kth tree to get a classification.

In this way, a test set classification is gotten for each case in about one-third of the trees. Let the final test set classification of the forest be the class having the most votes.

Comparing this classification with the class label present in the data gives an estimate of the test set error.

Class Vote Proportions

At run's end, for each case, the proportion of votes for each class is recorded. For each member of a test set (with or without class labels), these proportions are also computed. They contain useful information about the case. The margin of a case is the proportion of votes for the true class minus the maximum proportion of votes for the other classes. The size of the margin gives a measure of how confident the classification is.

Variable Importance.

Because of the need to know which variables are important in the classification, we have experimented with a number of different ways of measuring importance and settled on the following measure:

To estimated the importance of the mth variable. In the left out cases for the kth tree, randomly permute all values of the mth variable Put these new variable values down the tree and get classifications.

For the nth case in the data, its margin at the end of a run is the proportion of votes for its true class minus the maximum of the proportion of votes for each of the other classes. The measure of importance of the mth variable is the average lowering of the margin across all cases when the mth variable is randomly permuted.

Note that in earlier versions of RF the prime criterion was the rise in the oob error rate when the mth variable was randomly permuted.

This has been dropped because studies showed it was too volatile when there were many variables.

The Gini Measure

The splitting criterion used in RF is the gini criterion--also used in CART. At every split one of the mtry variables is used to form the split and there is a resulting decrease in the gini. The sum of all decreases in the forest due to a given variable, normalized by the number of trees, forms the Gini measure. This measure is not as reliable as the margin measure above but it is automatically computed in every run of random forests.

We illustrate the use of this information by some examples.

An Example--Hepatitis Data

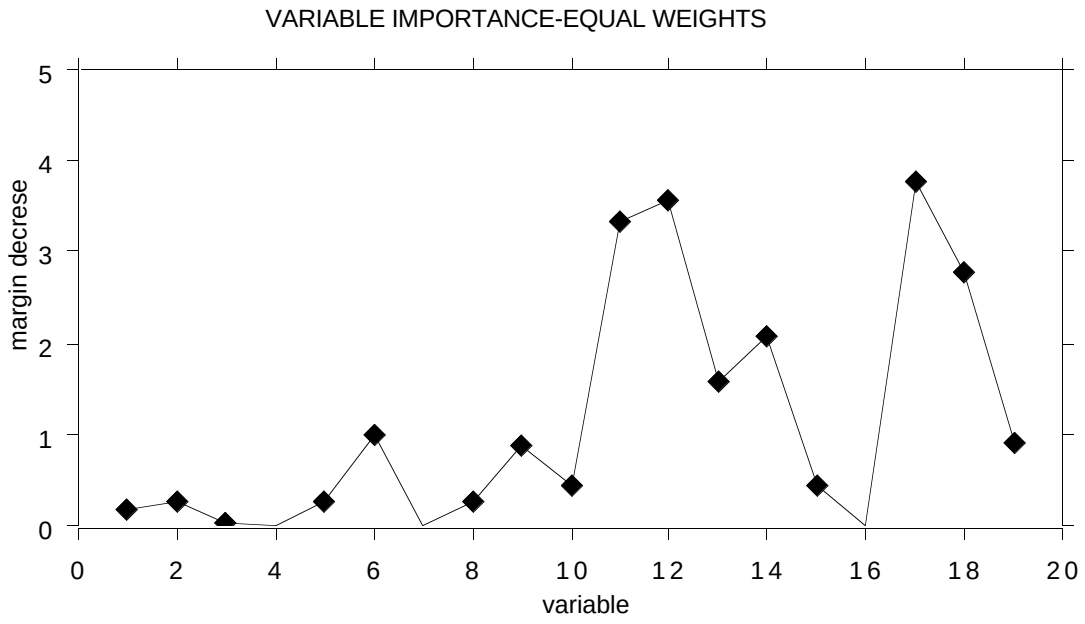
Data: Survival (123) or non-survival (32) of 155 hepatitis patients with 19 covariates. Analyzed by Diaconis and Efron in 1983 Scientific American. The original Stanford Medical School analysis concluded that the important variables were numbers 6, 12, 14, 19. Error rate for logistic regression is 17.4%.

Efron and Diaconis drew 500 bootstrap samples from the original data set and used a similar procedure, including logistic regression, to isolate the important variables in each bootstrapped data set.

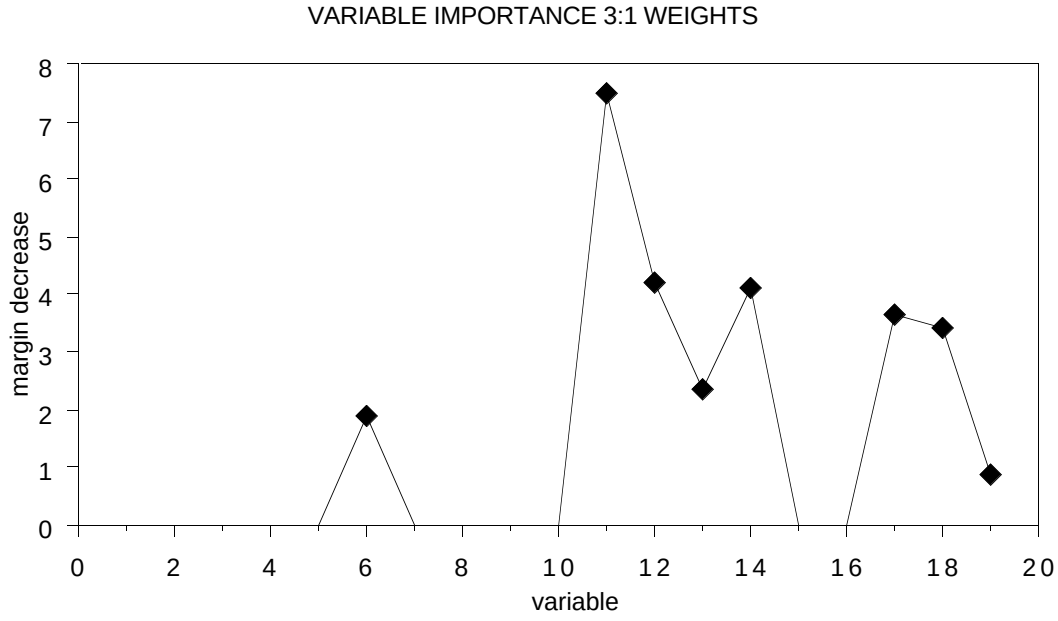
Their conclusion , "Of the four variables originally selected not one was selected in more than 60 percent of the samples. Hence the variables identified in the original analysis cannot be taken too seriously."

The parameters are set up for the analysis of the data hepatitus.txt available with FRv4. The is a small amount of missing data coded as -999, so missquick is set equal to 1. Originally, jclasswt is set equal to zero, and jclassout set to 1. Mtry is set equal to 3 which gave slightly lower error rate than 4.

The overall error rate is 14.2%. There is a 53% error in class 1, and 4% in class 2. The variable importances are graphed below:



The three most important variables are 11,12,17. Since the class of interest is non-survival which, with equal weights, has a high error rate, jclasswt is set to 1 and the classwt of class 1 to 3. The run gave an overall error rate of 22%, ,with class 1 error19% and 23% for class 2. The variable importances for this run are graphed below:



Variable 11 is the most important variable in separating non-survival from survival.

The standard procedure when fitting data models such as logistic regression is to delete variables; Diaconis and Efron (1983) state , ".statistical experience suggests that it is unwise to fit a model that depends on 19 variables with only 155 data points available."

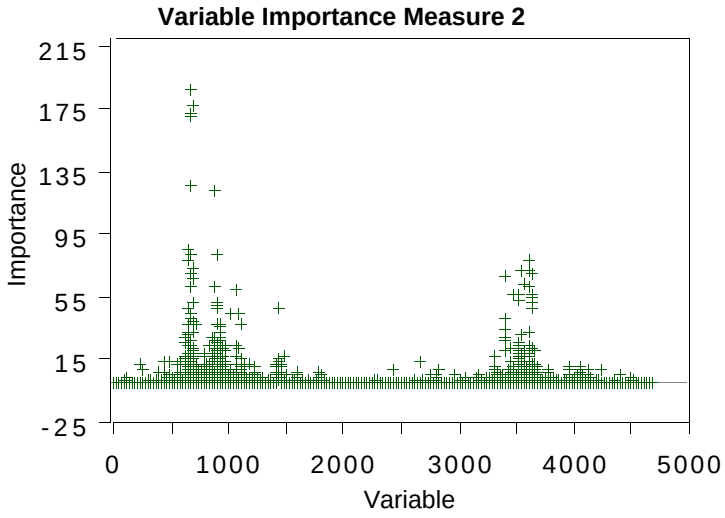
Newer methods in Machine Learning thrive on variables--the more the better. The next example is an illustration.

Microarray Analysis

Random forests was run on a microarray lymphoma data set with three classes, sample size of 81 and 4682 variables (genes) without any variable selection. The error rate was low (1.2%) using mtry=150.

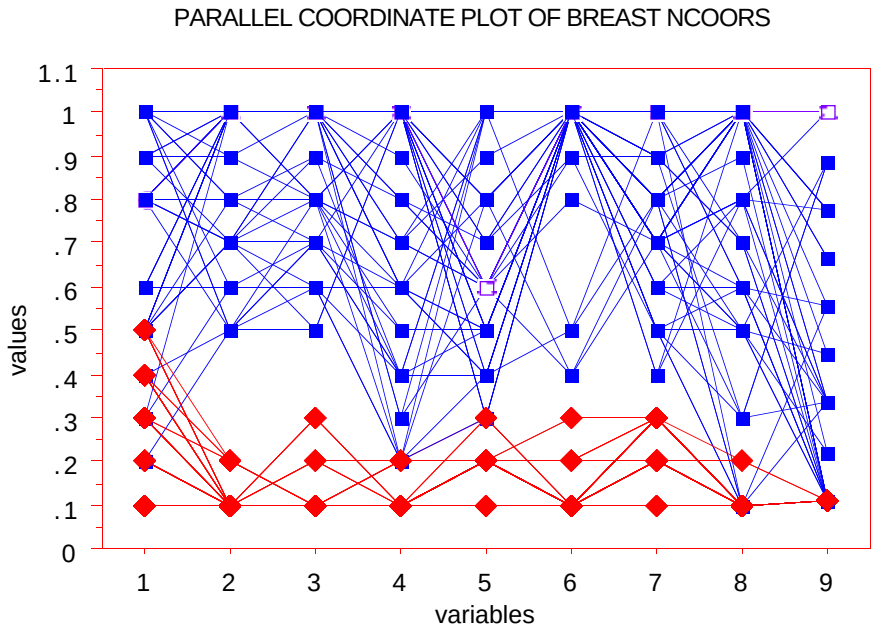
What was also interesting from a scientific viewpoint was an estimate of the importance of each of the 4682 genes.

The graph below were produced by a run of random forests. (measure 2 is the margin measure)

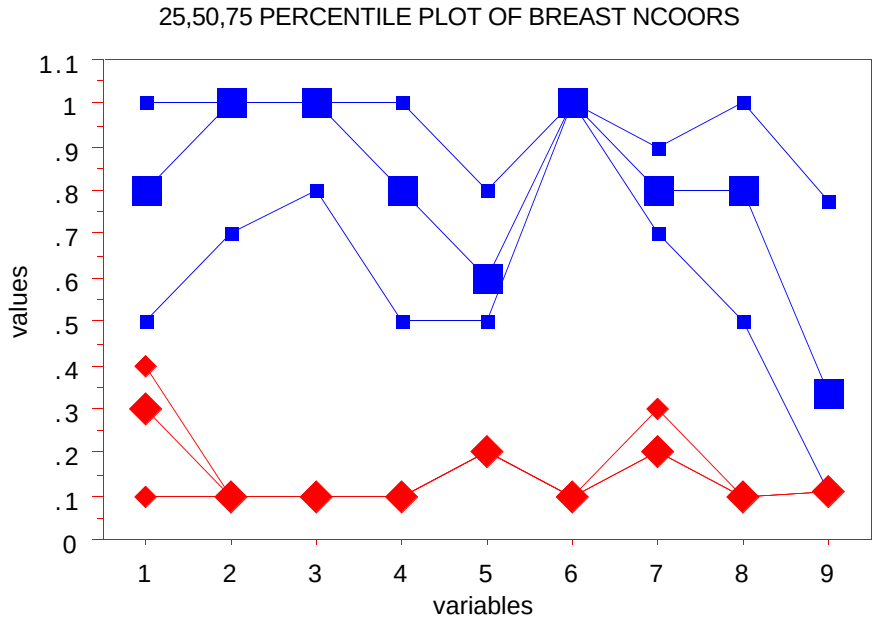


Effects of variables on predictions

If llcoor=1 and ncoor=50 from each class j, 50 of the cases having the highest q(j,n) values are extracted. This was done for the Breast Cancer data with 699 cases, 9 variables and 2 classes. First all variables were normalized so that each variable in the training set had an approximately uniform [0,1] distribution. Then the coordinates of the 100 ncoor cases were displayed-each thread in the display corresponds to one case, blue for class #2, red for #1.



Low values of all variables are associated with class #1 To simplify the quartiles (25%,50%,75%) of the above parallel coordinate plot can be displayed.



The medians are the large symbols. Above and below them are the 75th and 25th percentiles.

An intrinsic proximity measure

Since an individual tree is unpruned, the terminal nodes will contain only a small number of instances. Run all cases in the training set down the tree. If case i and case j both land in the same terminal node. increase the proximity between i and j by one. At the end of the run, the proximities are divided by twice the number of trees in the run and proximity between a case and itself set equal to one.

To cluster-use the above proximity measures.

Example-Bupa Liver Disorders

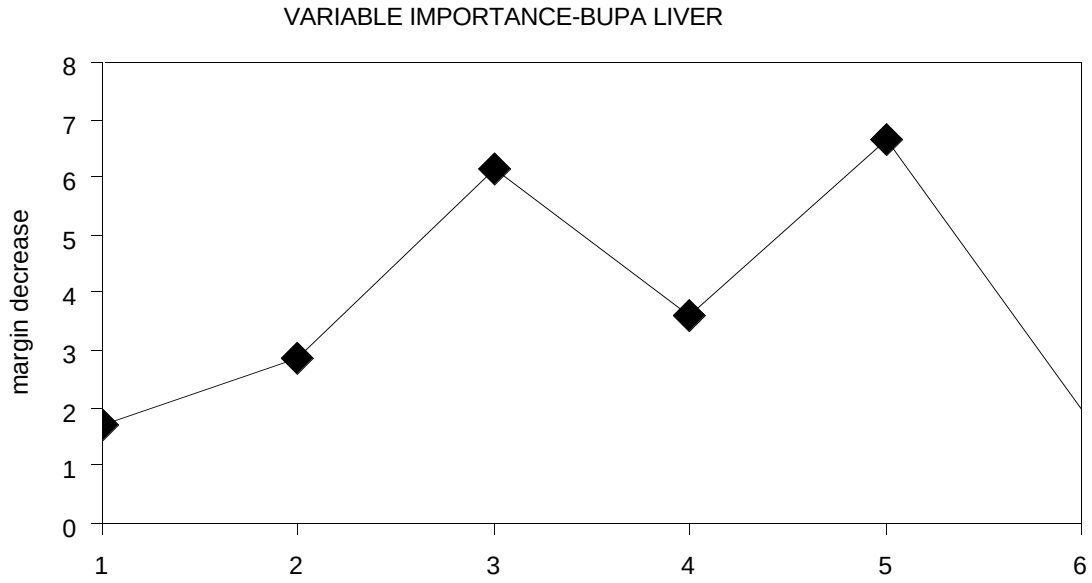
This is a two-class biomedical data set consisting of the covariates

1. mcv	mean corpuscular volume
2. alkphos	alkaline phosphotase
3. sgpt	alamine aminotransferase
4. sgot	aspartate aminotransferase
5. gammagt	gamma-glutamyl transpeptidase
6. drinks	number of half-pint equivalents of alcoholic beverage drunk per day

The first five attributes are the results of blood tests thought to be related to liver functioning. The 345 patients are classified into two classes by the severity of their liver disorders. The class populations are 145 and 200(severe).

The misclassification error rate is 28% in a Random Forests run. Class 1 has a 50% error rate with a rate of 12% for class 2. Setting the weight of class 1 to 1.4 gives an overall rate of 30% with rates 28% and 31% for classes 1 and 2.

Variable Importance



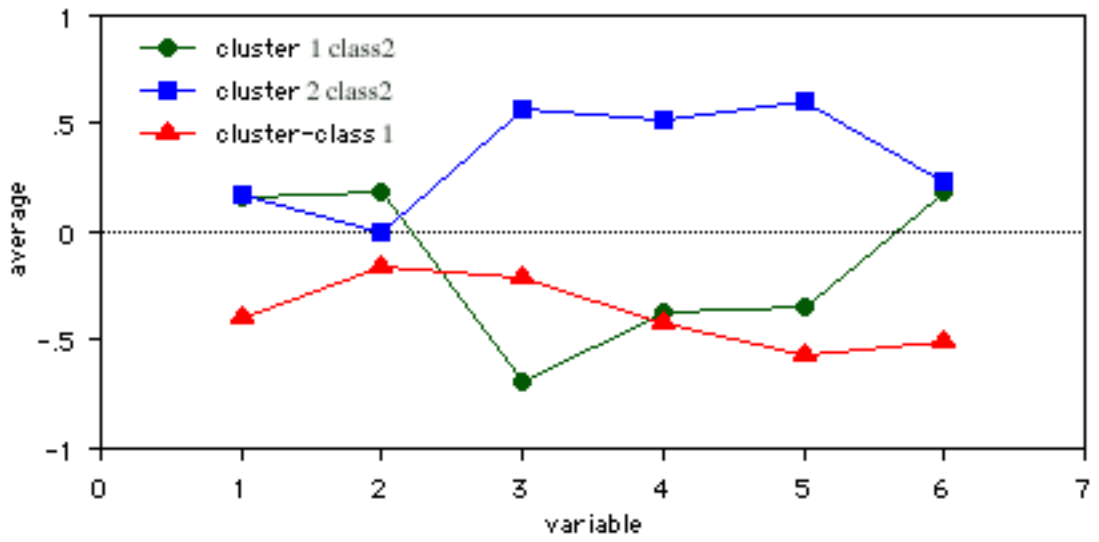
Blood tests 3 and 5 are the most important, followed by test 4.

B) ***Clustering***

Using the proximity measure outputted by Random Forests to cluster, there are two class #2 clusters.

In each of these clusters, the average of each variable is computed and plotted:

FIGURE 3 CLUSTER VARIABLE AVERAGES



Something interesting emerges. The class two subjects consist of two distinct groups: Those that have high scores on blood tests 3, 4, and 5 Those that have low scores on those tests. We will revisit this example below.

Scaling Coordinates

The proximities between cases n and k form a matrix {prox(n,k)}. From their definition, it is easy to show that this matrix is symmetric, positive definite and bounded above by 1, with the diagonal elements equal to 1. It follows that the values 1-prox(n,k) are squared distances in a Euclidean space of dimension not greater than the number of cases. For more background on scaling see "Multidimensional Scaling" by T.F. Cox and M.A. Cox

Let prox(n,-) be the average of prox(n,k) over the 2nd coordinate. and prox(-,-) the average over both coordinates. Then the matrix:

cv((n,k)=.5*(prox(n,k)-prox(n,-)-prox(k,-)+prox(-,-))

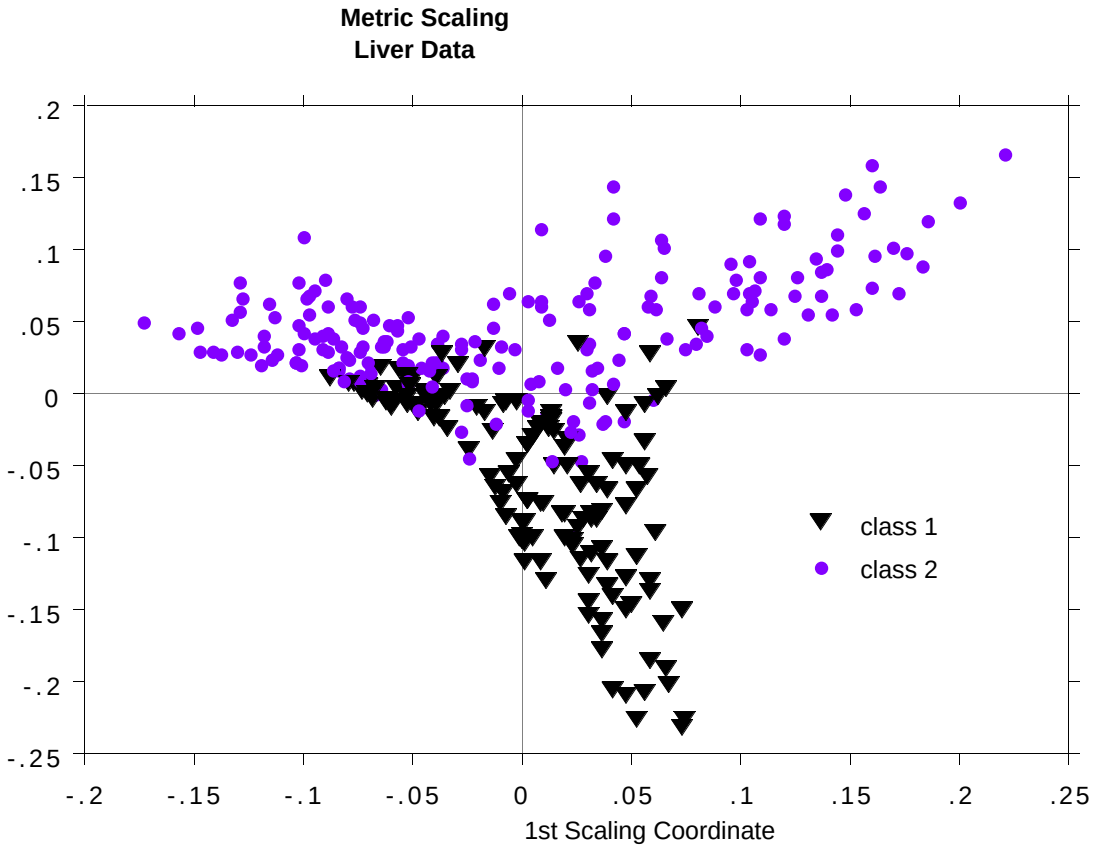
is the matrix of inner products of the distances and is also positive definte symmetric. Let the eigenvalues of cv be $\lambda(l)$_and the eigenvectors $v_l(n)$_Then the vectors

$$x(n) = (\sqrt{\lambda(1)}v_1(n), \sqrt{\lambda(2)}v_{2,}(n), \ldots)$$

have squared distances between them equal to 1-prox(n,k). We refer to the values of $\sqrt{\lambda(j)}v_j(n)$ as the jth scaling coordinate.

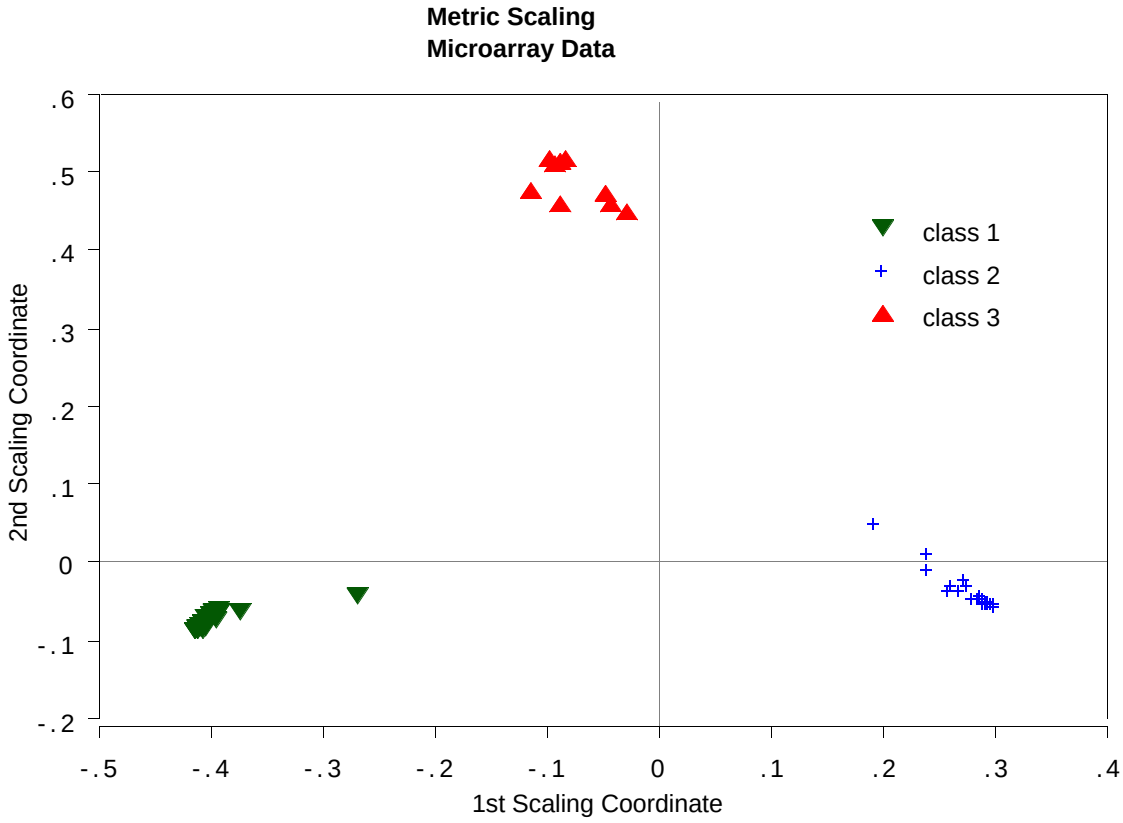
In metric scaling, the idea is to approximate the vectors **x**(n) by the first few scaling coordinates. This is done in random forests by extracting the number msdim of the largest eigenvalues and corresponding eigenvectors of the cv matrix. The two dimensional plots of the ith scaling coordinate vs. the jth often gives useful information about the data. The most useful is usually the graph of the 2nd vs. the 1st.

We illustrate with three examples. The first is the graph of 2nd vs. 1st scaling coordinates for the liver data



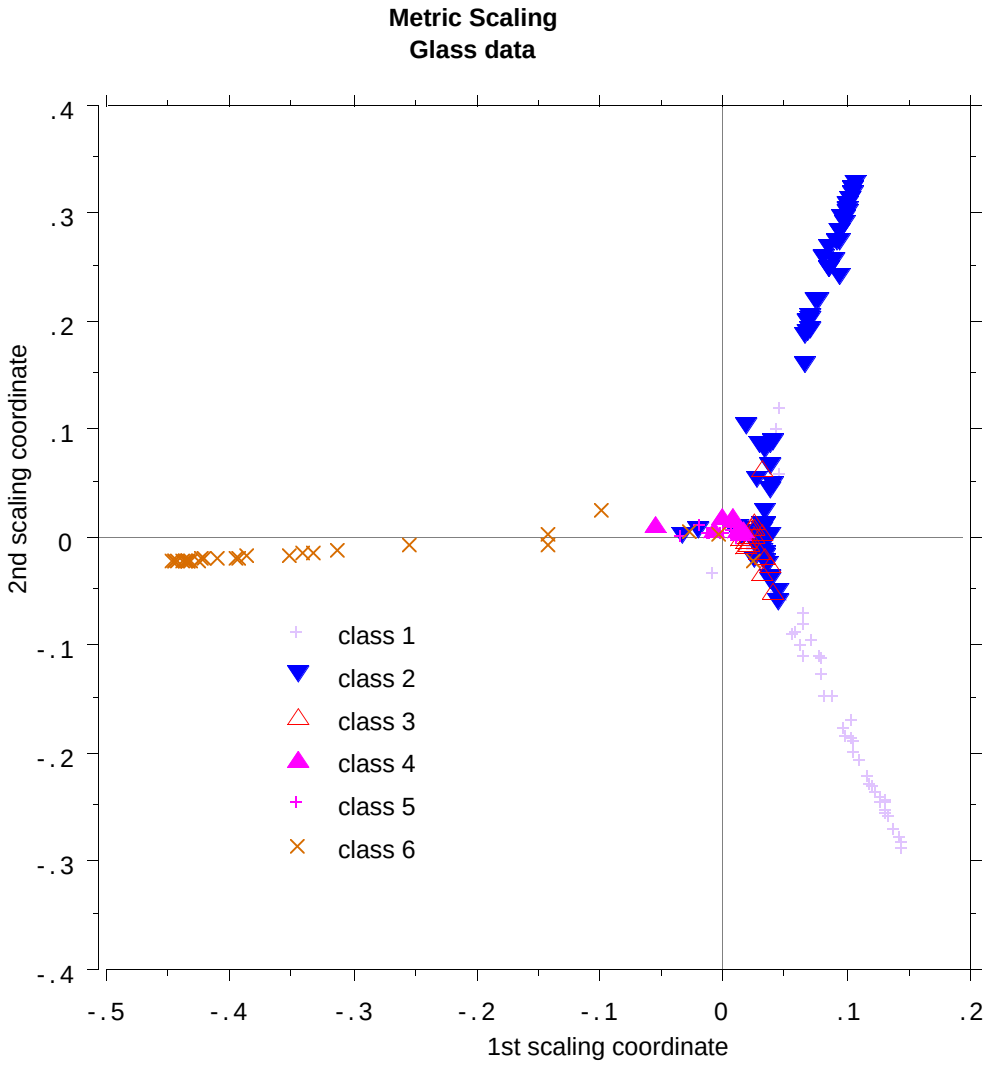
The two arms of the class #2 data in this picture correspond to the two clusters found and discussed above.

The next example uses the microarray data. With 4682 variables, it is difficult to see how to cluster this data. Using proximities and the first two scaling coordinates gives this picture:



Random forests misclassifies one case. This case is represented by the isolated point in the lower left hand corner of the plot.

The third example is glass data with 214 cases, 9 variables and 6 classes. This data set has been extensively analyzed (see Pattern Recognition and Neural Networkks-by B.D Ripley). Here is a plot of the 2nd vs. the 1st scaling coordinates.:

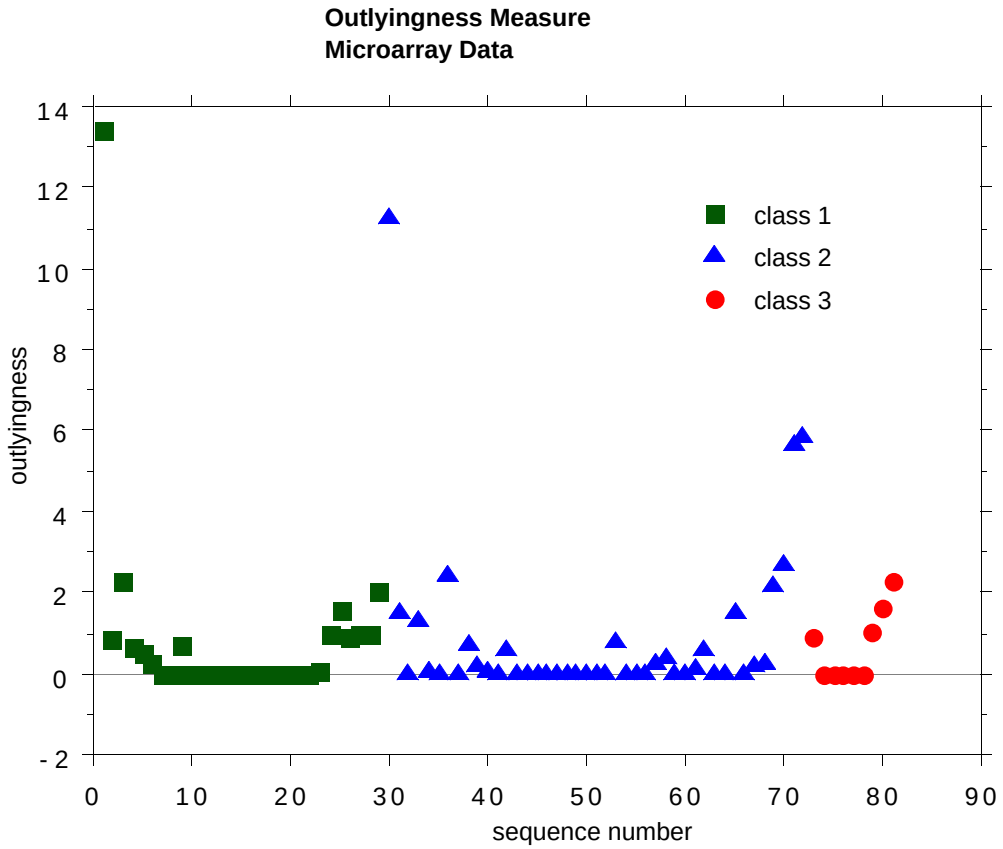


None of the analyses to data have picked up this interesting and revealing structure of the data--compare the plots in Ripley's book.

Outlier Location

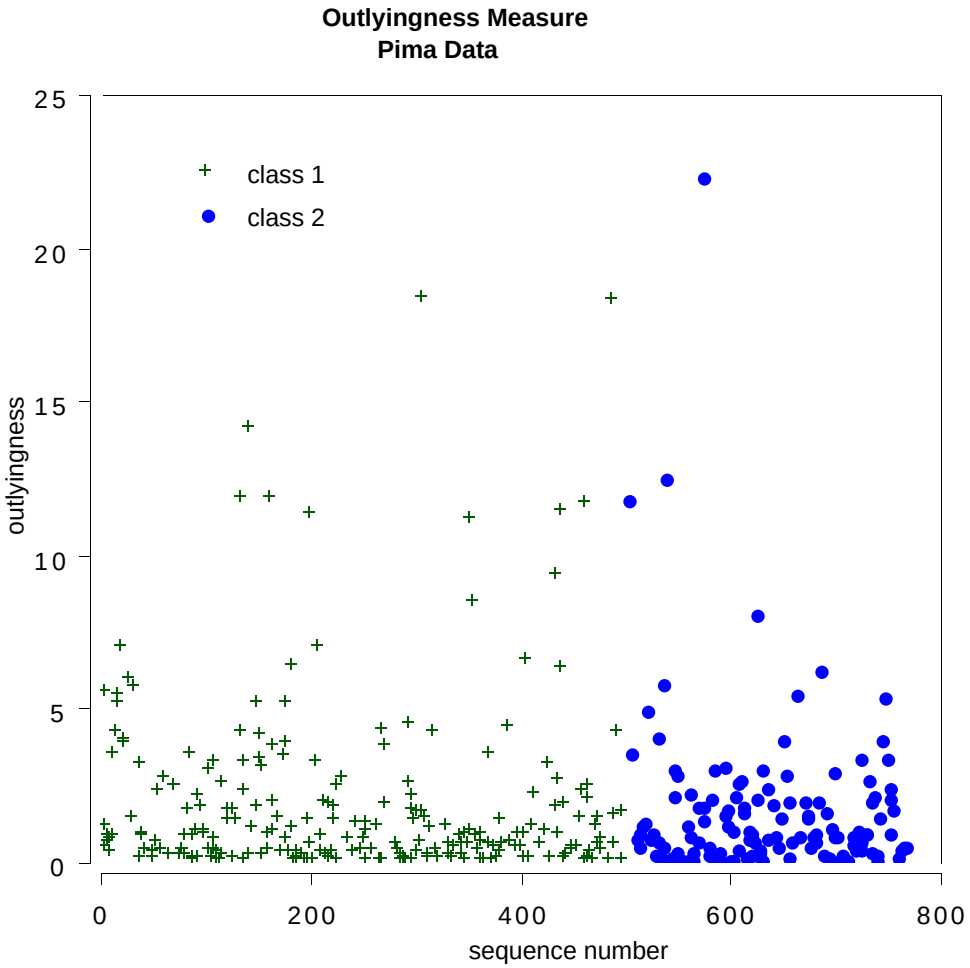
Outliers are defined as cases having small proximities to all other cases. Since the data in some classes is more spread out than others, outlyingness is defined only with respect to other data in the same class as the given case. To define a measure of outlyingness, we first compute, for a case n, the sum of the squares of prox(n,k) for all k in the same class as case n. Take the inverse of this sum--it will be large if the proximities prox(n,k) from n to the other cases k in the same class are generally small. Denote this quantity by out(n).

For all n in the same class, compute the median of the out(n), and then the mean absolute deviation from the median. Subtract the median from each out(n) and divide by the deviation to give a normalized measure of outlyingness. The values less than zero are set to zero. Generally, a value above 10 is reason to suspect the case of being outlying. Here is a graph of outlyingness for the microarray data



There are two possible outliers--one is the first case in class 1, the second is the first case in class 2.

As a second example, we plot the outlyingness for the Pima Indians hepatitis data. This data set has 768 cases, 8 variables and 2 classes. It has been used often as an example in Machine Learning research but has been suspected of containing a number of outliers.



If 10 is used as a cutoff point, there are 12 cases suspected of being outliers.

Analyzing Unlabeled Data

Unlabeled date consists of N vectors {**x**(n)} in M dimensions. Using the iaddcl option in random forests, these vectors are assigned class label 1. Another set of N vectors is created and assigned class label 2. The second synthetic set is created by independent sampling from the one-dimensional marginal distributions of the original data.

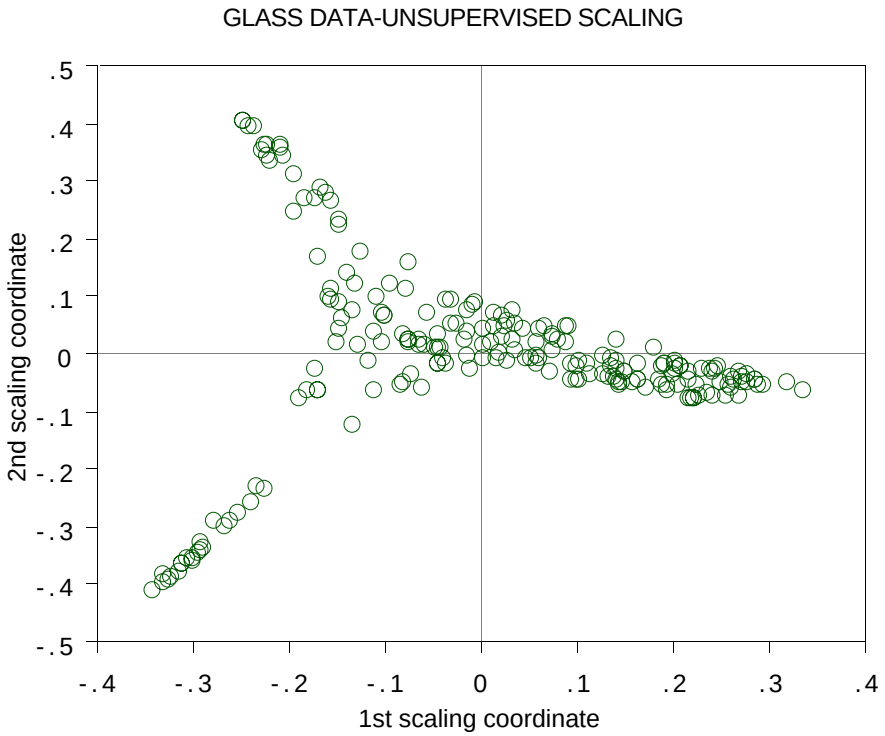
For example, if the value of the mth coordinate of the original data for the nth case is x(m,n), then a case in the synthetic data is constructed as follows: its first coordinate is sampled at random from the N values x(1,n), its second coordinate is sampled at random from

the N values x(2,n), and so on. Thus the synthetic data set can be considered to have the distribution of M independent variables where the distribution of the mth variable is the same as the univariate distribution of the mth variable in the original data.

When this two class data is run through random forests a high misclassification rate--say over 40%, implies that there is not much dependence structure in the original data. That is, that its structure is largely that of M independent variables--not a very interesting distribution. But if there is a strong dependence structure between the variables in the original data, the error rate will be low. In this situation, the output of random forests can be used to learn something about the structure of the data. Following are some examples.

Application to the Glass Data

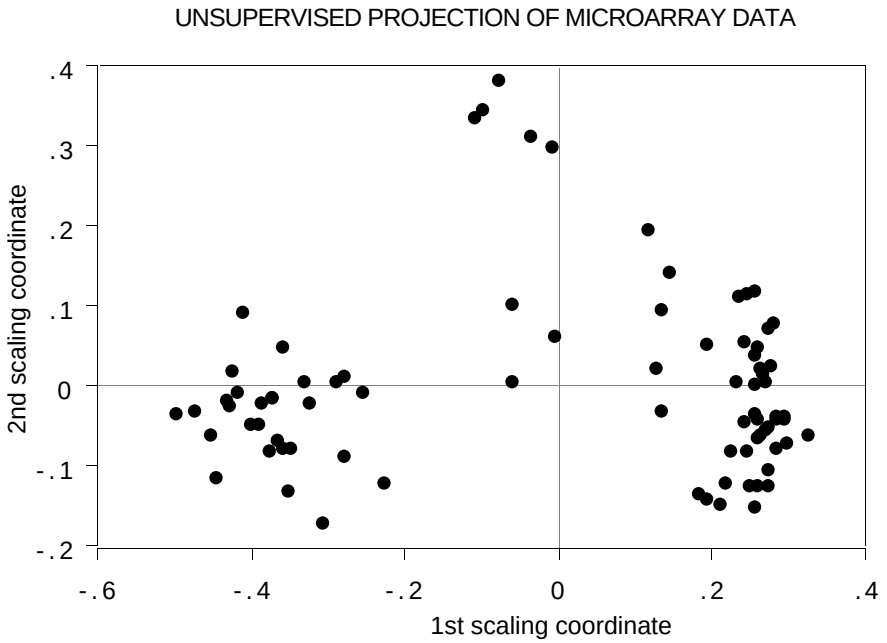
Recall that the scaling view of the labelled glass data was in a three armed starfish configuration. In this experiment, we labelled all the glass data class#1, set up a second synthetic data set labeled class #2, and used scaling coordinates to project class #1 onto two dimensions. Here is the outcome:



This is a good replica of the original projection.

` *Application to the Microarray Data*

Recall that the scaling plot of the microarray data showed three clusters--two larger ones in the lower left hand and right hand corners and a smaller one in the top middle. Again, we erased labels from the data and projected down an unsupervised view:

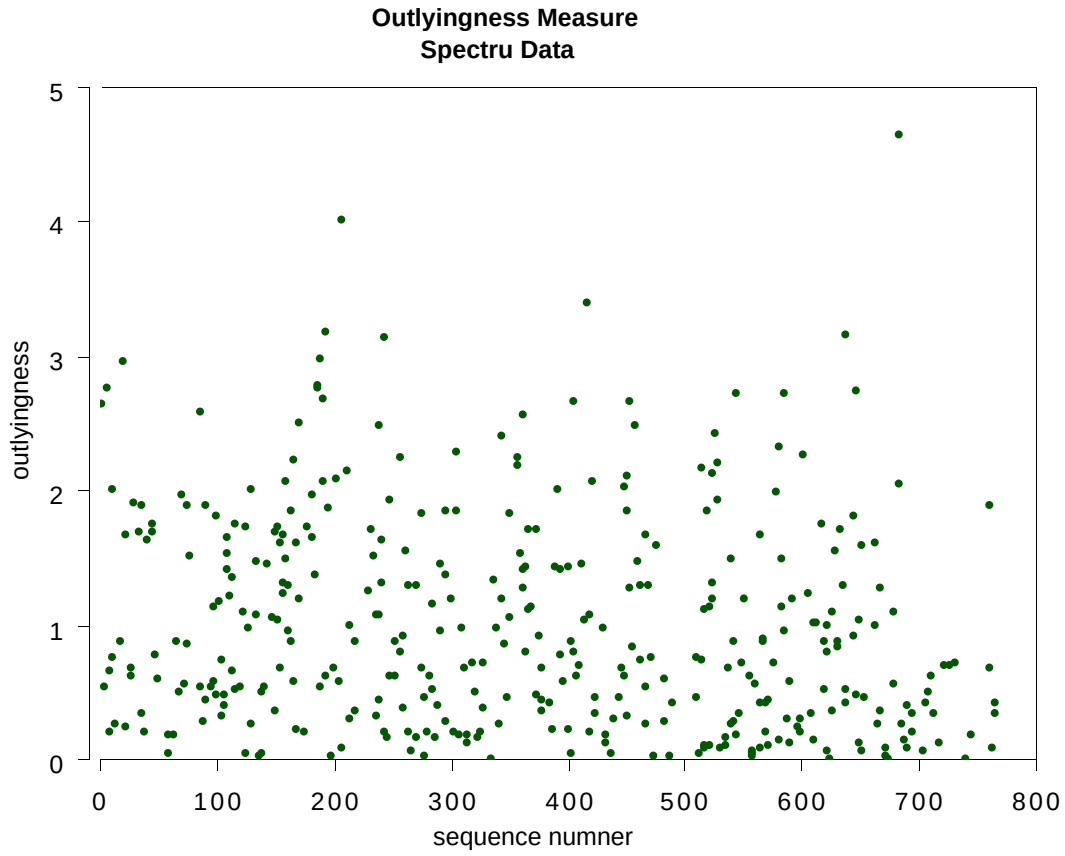


The three clusters are more diffuse but still apparent.

An Application to Chemical Spectra

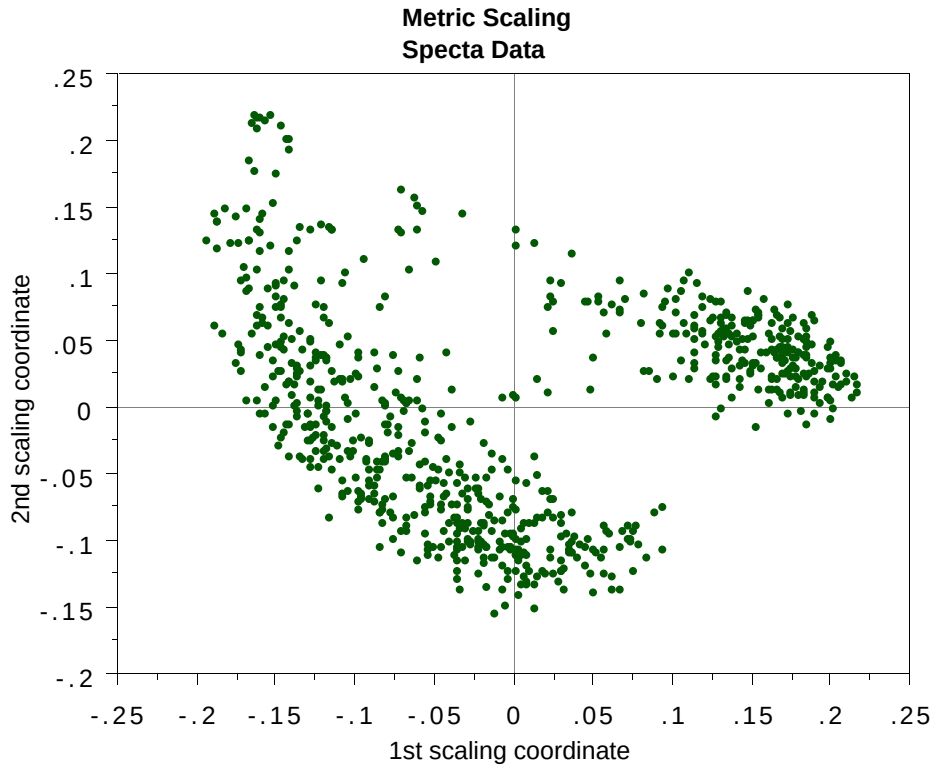
Data graciously supplied by Merck consists of the first 468 spectral intensities in the spectrums of 764 compounds. The challenge presented by Merck was to find small cohesive groups of outlying cases in this data. Using the iaddcl option, there was excellent separation between the two classes, with an error rate of 0.5%, indicating strong dependencies in the original data.

We looked at outliers and generated this plot.

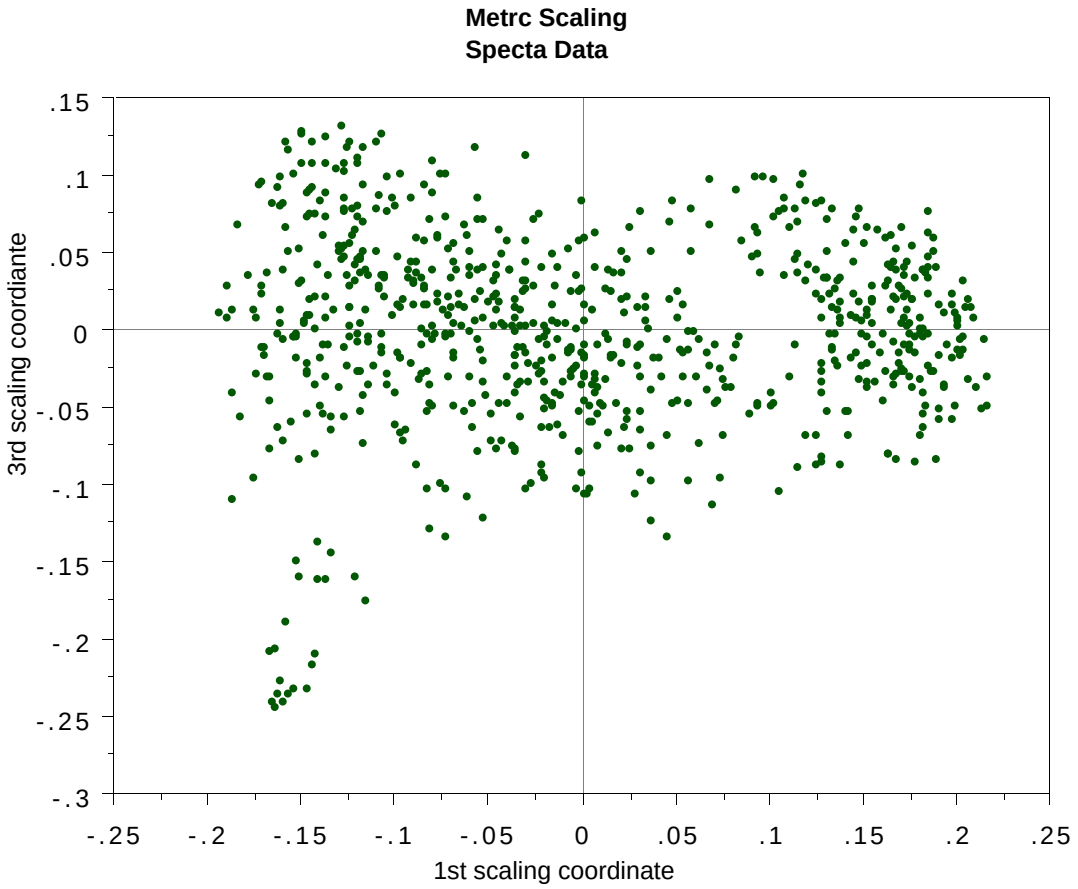


This plot gives no indication of outliers. But outliers must be fairly isolated to show up in the outlier display.

To search for outlying groups scaling coordinates were computed. The plot of the 2nd vs. the 1st is below:



This shows, first, that the spectra fall into two main clusters. There is a possiblity of a small outlying group in the upper left hand corner. To get another picture, the 3rd scaling coordinate is plotted vs. the 1st.



The group in question is now in the lower left hand corner and its separation from the body of the spectra has become more apparent.

Appendix: Saving Forests and Running New Data Down Them

If the data set is large with many variables, a run growing 100 trees may take awhile. If there is another set of data with the same parameters except for sample size, the user may want to run this 2nd set down the forest either to get classifications or to use the data as a test set. In this case put isaverf=1 and isavepar=1.

When isaverf is on, the variable values saved to file (which the user must name-say 'forest55') are enough to reconstruct the forest. If irunrf =1, then read in the new data from file. The statement open(1, file='forest55',status='old') runs the new data down the saved 'forest55'.

If label is set =1, that implies that the new data has labels, and the program will output to screen the overall error rate and the class error rate at the end of the run.. If label=0, then the new data has no class labels. But data must still be read in as though there were values of the labels. Simply assign each label the value 1. As soon as these values are filled in and the new data read in, it goes through the forest. If infoutr=1, then at the end of the run, all predicted class labels will be saved to a file.

The user knows nsample0--the sample size of the data to run down the reconstituted forest. But may not remember the other values that need to be put in the parameter statement. However, if when doing the initial run, isavepar was put equal to 1 and a filename given to store the information, then all the needed parameters will be saved to the file as well as a textual description of up to 500 characters. The only thing that the user provides is the file name and the textual description. To recover this information, put ireadpar=1, other option values do not matter, give the file name containing the parameters and compile.

For runs on an old forest with new data modify the parameter statement as follows.

line1) set nsample0=sample size of the new data.
set label=0 if the new data has no labels.
line2) only jbt has to be the same as original
line3) same as original
line4) set options to zero
line5) set options to zero except mdimsc=1
line6) same as original
line7) set options to zero
line8) first two are zero, irunrf=1,i readpar=0.
line9) all zero except perhaps infoutr

replace the line:
nrnodes=2*(nsample/ndsize)+1
with
nrnodes=(original value of nrnodes as given in the parameter file)

If you want the same output from the new run as infoutr provides on the original run (except for margins) set infoutr=1 and give a file name to receive the information (output file #3)

If you used class weights in the original run, the same weights will be applied to the new run. The programs expects the same missing value code applies to any new data and files in missing values using the original fillins from missquick.