treeClust: An R Package for Tree-Based Clustering Dissimilarities

by Samuel E. Buttrey and Lyn R. Whitaker

Abstract This paper describes **treeClust**, an R package that produces dissimilarities useful for clustering. These dissimilarities arise from a set of classification or regression trees, one with each variable in the data acting in turn as a the response, and all others as predictors. This use of trees produces dissimilarities that are insensitive to scaling, benefit from automatic variable selection, and appear to perform well. The software allows a number of options to be set, affecting the set of objects returned in the call; the user can also specify a clustering algorithm and, optionally, return only the clustering vector. The package can also generate a numeric data set whose inter-point distances relate to the **treeClust** ones; such a numeric data set can be much smaller than the vector of inter-point dissimilarities, a useful feature in big data sets.

Introduction

Clustering is the act of dividing observations into groups. One critical component of clustering is the definition of distance or dissimilarity between two entities (two observations, or two clusters, or a cluster and an observation). We assume the usual case of a rectangular array of data in which p attributes are measured for each of n individuals. The common choice of the usual Euclidean distance needs to be extended when some of the attributes are categorical. A good dissimilarity should be able to incorporate categorical variables, impose appropriate attribute weights (in particular, be insensitive to linear scaling of the attributes), and handle missing values and outliers gracefully. Buttrey and Whitaker (2015) describe an approach which uses a set of classification and regression trees that collectively determine dissimilarities that seems to perform well in both numeric and mixed data, particularly in the presence of noise.

The next section covers the idea behind the tree-based clustering, while the following one ("The treeClust package") describes the software we have developed for this purpose. The software is implemented as an R package, available under the name **treeClust** at the CRAN repository. The section on software also gives some of the attributes of the procedure, like its insensitivity to missing values, and of the software, like the ability to parallelize many of the computations. The "Dissimilarity-preserving data" section describes a feature of the software that has not yet been explored – the ability to generate a new numeric data set whose inter-point distances are similar to our tree-based dissimilarities. This allows a user to handle larger data sets, since in most cases the new data set will have many fewer entries than the vector of all pairwise dissimilarities. In the final section we provide an example using a data set from the UC Irvine repository (Bache and Lichman, 2013); this data set, and a script with the commands used in the example, are also available with the paper or from the authors.

Clustering with trees

The idea of tree-based clustering stems from this premise: objects that are similar tend to land in the same leaves of classification or regression trees. In a clustering problem there is no response variable, so we construct a tree for each variable in turn, using it as the response and all others are potential predictors. Trees are pruned to the "optimal" size (that is, the size for which the cross-validated deviance is smallest). Trees pruned all the way back to the root are discarded.

The surviving set of trees are used to define four different, related measures of dissimilarity. In the simplest, the dissimilarity between two observations i and j, say, $d_1(i,j)$, is given by the proportion of trees in which those two observations land in different leaves. (Our dissimilarities are not strictly distances, since we can have two distinct observations whose dissimilarity is zero.) That is, the dissimilarity between observations associated with a particular tree t is $d_1^t(i,j) = 0$ or 1, depending on whether i and j do, or do not, fall into the same leaf under tree t. Then over a set of T trees, $d_1(i,j) = \sum_t d_1^t(i,j)/T$.

A second dissimilarity, d_2, generalizes the values of 0 and 1 used in d_1. We start by defining a measure of tree "quality," q, given by the proportion of deviance "explained" by the tree. The contribution to the deviance for each leaf is the usual sum of squares from the leaf mean for a numeric response, and the multinomial log-likelihood for a categorical one. Then, for tree t, we have q_t = [(Deviance at root − Deviance in leaves) / Deviance in root]. A "good" tree should have a larger q

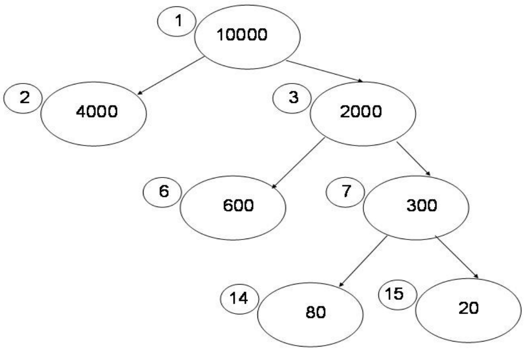

Figure 1: Example tree, showing node numbers (small circles) and deviances, from Buttrey and Whitaker (2015).

and a higher weight in the dissimilarity. So we define $d_2^t(i,j)$ to be 0 if the two observations fall in the same leaf, and otherwise $q_t/\max_k(q_k)$, where the maximum, of course, is taken over the whole set of trees. As before, $d_2(i,j) = \sum_t d_2^t(i,j)/T$.

In both of the d_1 and d_2 measures, each tree contributes only two values, zero and a constant that does not depend on which two different leaves a pair of observations fell into. The d_3 dissimilarity attempts to address that. Consider the tree in figure 1, taken from Buttrey and Whitaker (2015). In this picture, small numbers in circles label the leaves, while large numbers in ovals give the node-specific deviance values. A pair of observations falling into leaves 14 and 15 on this tree seem closer together than a pair falling into 2 and 15, for example. We measure the distance between two leaves by computing the increase in deviance we would observe, if we were to minimally prune the tree so that the two observations were in the same leaf, compared to the overall decrease in deviance for the full tree. For example, the picture in figure 1 shows a starting deviance (node 1) of 10000 and a final deviance (summing across leaves) of 4700. Now consider two observations landing in leaves 14 and 15. Those two observations would end up in the same leaf if we were to prune leaves 14 and 15 back to node 7. The resulting tree's deviance would be increased to 4900, so we define the dissimilarity between those two leaves – and between the two relevant observations – by $(4900 - 4700)$ / $(10000 - 4700)$ = 0.038. Observations in leaves 6 and 15 would contribute distance $(6000 - 4700)$ / $(10000 - 4700)$ = 0.25, while observations in leaves 2 and 14, say, would have distance 1 for this tree.

The final dissimilarity measure, d_4, combines the d_2-style tree-specific weights with the d_3 dissimilarity, simply by multiplying each d_3 dissimilarity by the tree's $q_t/max_k q_k$. The best choice of dissimilarity measure, and algorithm with which to use it, varies from data set to data set, but our experience suggests that in general, distances d_2 and d_4 paired with with the `pam()` clustering algorithm from the **cluster** (Maechler et al., 2015) package are good choices for data sets with many noise variables.

Other tree-based approaches

Other researchers have used trees to construct dissimilarity measures. One is based on random forests, a technique introduced in (Breiman, 2001) for regression and classification. The user manual (Breiman and Cutler, 2003) describes how this approach might be used to produce a proximity measure for unlabelled data. The original data is augmented with data generated at random with independent columns whose marginal distributions match those of the original. So the original data preserves the dependence structure among the columns, whereas the generated data's columns are independent. The original data is labelled with 1's, and the generated data with 0's; then the random forest is generated with those labels as the response. Proximity is measured by counting the number of times two observations fall in the same trees, dividing this number by twice the number of trees, and setting each observation's proximity with itself as 1. In our experiments we construct dissimilarity by subtracting proximity from 1. The random forest dissimilarity is examined in (Shi and Horvath, 2006), and we use it below to compare to our new approach.

Other clustering approaches based on trees includes the work of Fisher (1987), in which trees are used to categorize items in an approach called "conceptual clustrering." It appears to be intended for categorical predictors in a natural hierarchy. Ooi (2002) uses trees as a way to partition the feature space on the way to density estimation, and Smyth et al. (2006) use multivariate regression trees with principal component and factor scores. Both of these approaches are intended for numeric data.

We thank an anonymous referee for bringing to our attention the similarity between our approach and the extensive literature on building phylogenetic trees. A thread in this literature is to start with a distance matrix and end with a phylogenetic or evolutionary tree, whereas we use the trees to build the distance. Felsenstein (2003) serves as a guide to developing phylogenetic trees, while O'Meara (2015) gives the CRAN Task View summary of R packages for phylogenetics. In the phylogenetic literature, methods for measuring distances and growing trees are tailored to the specific application. For example defining distances might involve informed choices of morphological or behavioral features or on aligning genetic material to capture dissimilarities at a molecular level. Our contribution, on the other hand, is to use trees to automate construction of dissimilarity matrices. Our context is more general in that we do not know *a priori* which variables are more important than others, which are noise, how to scale numeric variables, or how to collapse and combine categorical ones. Methods using application-specific knowledge might be expected to yield superior results, but it would be interesting to see if the ideas behind tree distances can be adapted for use in growing phylogenetic trees.

The treeClust package

Dependencies

This paper describes version 1.1.1 of the **treeClust** package. This package relies on the **rpart** package (Therneau et al., 2015) to build the necessary classification and regression trees. This package is "recommended" in R, meaning that users who download binary distributions of R will have this package automatically. **treeClust** also requires the **cluster** package for some distance computations; this, too, is "recommended." Finally, the **treeClust** package allows some of the computations to be performed in parallel, using R's **parallel** package, which is one of the so-called "base" packages that can be expected to be in every distribution of R. See "Parallel processing" for more notes on parallelization.

Functions

In this section we describe the functions that perform most of the work in **treeClust**, with particular focus on `treeClust()`, which is the function users would normally call, and on `treeClust.control()`, which provides a mechanism to supply options to `treeClust()`.

The `treeClust()` function

The `treeClust()` function takes several arguments. First among these, of course, is the data set, in the form of an R "data.frame" object. This rectangular object will have one row per observation and one column per attribute; those attributes can be categorical (including binary) or numeric. (For the moment there is no explicit support for ordered categoricals.) The user may also supply a choice of tree-based dissimilarity measure, indicated by an integer from 1 to 4. By default, the algorithm builds one tree for each column, but the `col.range` argument specifies that only a subset of trees is to be built. This might be useful when using parallel processing in a distributed computing architecture. Our current support for parallelization uses multiple nodes on a single machine; see "Parallel processing" below.

If the user plans to perform the clustering, rather than just compute the dissimilarities, he or she may specify the algorithm to be used in the clustering as argument `final.algorithm`. Choices include `"kmeans"` from base R, and `"pam"`, `"agnes"` and `"clara"` from the **cluster** package. (The first of these is an analog to `"kmeans"` using medoids, which are centrally located observations; the next two are implementations of hierarchical clustering, respectively agglomerative and divisive.) If any of these is specified, the number of clusters k must also be specified. By default, the entire output from the clustering algorithm is preserved, but the user can ask for only the actual vector of cluster membership. This allows users easy access to the most important part of the clustering and obviates the need to store all of the intermediate objects, like the trees, that go into the clustering. This election is made through the `control` argument to argument to `treeClust()`, which is described next.

The `treeClust.control()` function

The `control` argument allows the user to modify some of the parameters to the algorithm, and particularly to determine which results should be returned. The value passed in this argument is normally the output of a function named `treeClust.control()`, which follows the convention of R functions like glm.control() and rpart.control(). That is, it accepts a set of input names and values, and returns a list of control values in which the user's choices supersede default values.

The primary motivation for the control is to produce objects of manageable size. The user may elect to keep the entire set of trees, but there may be dozens or even hundreds of these, so by default the value of the treeClust.control()'s return.trees member is FALSE. Similarly, the dissimilarity object for a data set with n observations has $n(n-1)/2$ entries, so by default return.dists is FALSE. The matrix of leaf membership is approximately the same size as the original data – it has one column for each tree, but not every column's tree is preserved. The return.mat entry defaults to TRUE, since we expect our users will often want this object. The return.newdata entry describes whether the user is asking for newdata, which is the numeric data set of observations in which the inter-point dissimilarities are preserved. This is described in the section on "Dissimilarity-preserving data" below. By default, this, too, is FALSE. It is also through treeClust.control() that the user may specify cluster.only = TRUE. In this case, the return value from the function is simply the numeric n-vector of cluster membership.

The serule argument descibes how the pruning is to be done. By default, trees are pruned to the size for which the cross-validated deviance is minimized. When the serule argument is supplied, trees are pruned to the smallest size for which the cross-validated deviance is no bigger than the minimum deviance plus serule standard errors. So, for example, when serule $= 1$, the one-SE rule of Breiman et al. (1984) is implemented.

An argument named DevRatThreshold implements an experimental approach to detecting and removing redundant variables. By redundant, we mean variables that are very closely associated with other variables in the data set, perhaps because one was constructed from the other through transformation or binning, or perhaps because two variables describing the same measurement were acquired from two different sources. Let the sum of deviances across a tree's leaves be d_L, and the deviance at the root be d_R. Then for each tree we can measure the proportional reduction in deviance, $(d_R - d_L)/d_R$. A tree whose deviance reduction exceeds the value of DevRatThreshold is presumed to have arisen because of redundant variables. We exclude the predictor used at the root split from the set of permitted predictor variables for that tree and re-fit until the trees's deviance reduction does not exceed the threshold, or until there are no remaining predictors. (We thank our student Yoav Shaham for developing this idea.)

The final choice of the tree.control() is the parallelnodes argument, which allows trees to be grown in parallel. When this number is greater than 1, a set of nodes on the local machine is created for use in growing trees. (This set of nodes is usually referred to as a "cluster," but we want to reserve that word for the output of clustering.) Finally, additional arguments can be passed down to the underlying clustering routine, if one was specified, using R's ... mechanism.

Return value

If cluster.only has not been set to TRUE, the result of a call to treeClust() will be an object of class "treeClust" and contain at least five entries. Two of these are a copy of the call that created the object and the d_num argument supplied by the user. A "treeClust" object also contains a matrix named tbl with one row for each of the trees that was kept, and two columns. These are DevRat, which gives the tree's strength (q above), and Size, which gives the number of leaves in that tree. Finally, every "treeClust" object has a final.algorithm element, which may have the value "None", and a final.clust element, which may be NULL.

Dissimilarity functions

There are two functions that are needed to compute inter-point dissimilarities. The first of these, tcdist() (for "tree clust distance") is called by treeClust(). However, it can also be called from the command line if the user supplies the result of an earlier call. Unless the user has asked for cluster.only = TRUE, the return value of a call to treeClust() is an object of class "treeClust", and an object of this sort can be passed to tcdist() in order to compute the dissimilarity matrix. Note that certain elements will need to be present in the "treeClust" object, the specifics depending on the choice of dissimilarity measure. For example, most "treeClust" objects have the leaf membership matrix, which is required in order to compute dissimilarities d_1 and d_2. For d_3 and d_4 the set of trees is required.

The tcdist() function makes use of another function, d3.dist(), when computing d_3 or d_4 distances. This function establishes all the pairwise distances among leaves in a particular tree, and then computes the pairwise dissimilarities among all all observations by extracting the relevant leaf-wise distances. An unweighted sum of those dissimilarities across trees produces the final d_3, and a weighted one produces d_4. The d3.dist() function's argument return.pd defaults to FALSE; when set to TRUE, the function returns the matrix of pairwise leaf distances. Those leaf distances are used by the tcnewdata() function in order to produce a matrix called newdata, which has the property that the inter-point distances in this matrix mirror the treeClust() dissimilarities. Users might create this newdata matrix in large samples, where the vector of all pairwise distances is too large to be readily handled. This feature is described under "Dissimilarity-preserving data" below.

Support functions

A number of support functions are also necessary. We describe them here. Two of these arise from quirks in the way rpart() works. In a regression tree, an "rpart" object's frame element contains a column named dev which holds the deviance associated with each node in the tree. However, in a classification tree, this column holds the number of misclassifications. So our function rp.deviance() computes the deviance at each node. We declined to name this function deviance.rpart, because we felt that users would expect it to operate analgously to the deviance method, deviance.tree(), for objects of class "tree" in the **tree** package (Ripley, 2015). However, that function produces a single number for the whole tree, whereas our function produces a vector of deviances, one for each node.

A second unexpected feature of rpart() is that there is no predict method which will name the leaf into which an observation falls. The where element gives this information for observations used to fit the tree, but we need it also for those that were omitted from the fitting because they were missing the value of the response (see section "Handling missing values" below). Our function rpart.predict.leaves() produces the leaf number (actually, the row of the "rpart" object's frame element) into which each observation falls. For the idea of how to do this, we are grateful to Yuji (2015).

The support function treeClust.rpart() is not intended to be called directly by users. This function holds the code for the building of a single tree, together with the pruning and, if necessary, the re-building when the user has elected to exclude redundant variables (see DevRatThreshold above). It is useful to encapsulate this code in a stand-alone function because it allows that function to be called once for each column either serially or in parallel as desired.

The make.leaf.paths() function makes it easy to find the common ancestors of two leaves. Recall that under the usual leaf-naming convention, the children of leaf m have numbers $2m$ and $2m + 1$. The make.leaf.paths() function takes an integer argument up.to giving the largest row number; the result is a matrix with up.to rows, with row i giving the ancestors of i. For example, the ancestors of 61 are 1, 3, 7, 15, 30 and 61 itself. The number of columns in the output is j when $2^j \leq \texttt{up.to} < 2^{j+1}$, and unneeded entries are set to zero.

Finally we provide a function cramer() that computes the value of Cramér's V for a two-way table. This helps users evaluate the quality of their clustering solutions.

Print, plot and summary methods

The objects returned from treeClust() can be large, so we supply a print() method by which they can be displayed. For the moment, the set of trees, if present, is ignored, and the print() method shows a couple of facts about the call, and then displays the table of deviances, with one row for each tree that was retained. Even this output can be too large to digest easily, and subsequent releases may try to produce a more parsimonious and informative representation. The summary() method produces the output from the print() method, only it excludes the table of deviances, so its output is always small.

We also supply a plot method for objects of class "treeClust". This function produces a plot showing the values of q (that is, the DevRat column in the tbl entry) on the vertical axis, scaled so that the maximum value is 1. The horizontal axis simply counts the trees starting at one. Each point on the graph is represented by a digit (or sometimes two) showing the size of the tree. For example, figure 2 shows the plot generated from a call to plot() passing a "treeClust" object generated from the splice data from the UCI Data Repository (Bache and Lichman, 2013) (see also the example below). The data has 60 columns (horizontal axis); this run of the function generated 59 trees (the "1" at $x = 32$ shows that the tree for the 32^{nd} variable was dropped). The "best" tree is number 30, and it had three leaves (the digit "3" seen at coordinates (30, 1)).

Handling missing values

The treeClust() methodology uses the attributes of rpart() to handle missing values simply and efficiency. There are two ways a missing value can impact a tree (as part of a response or as a predictor)

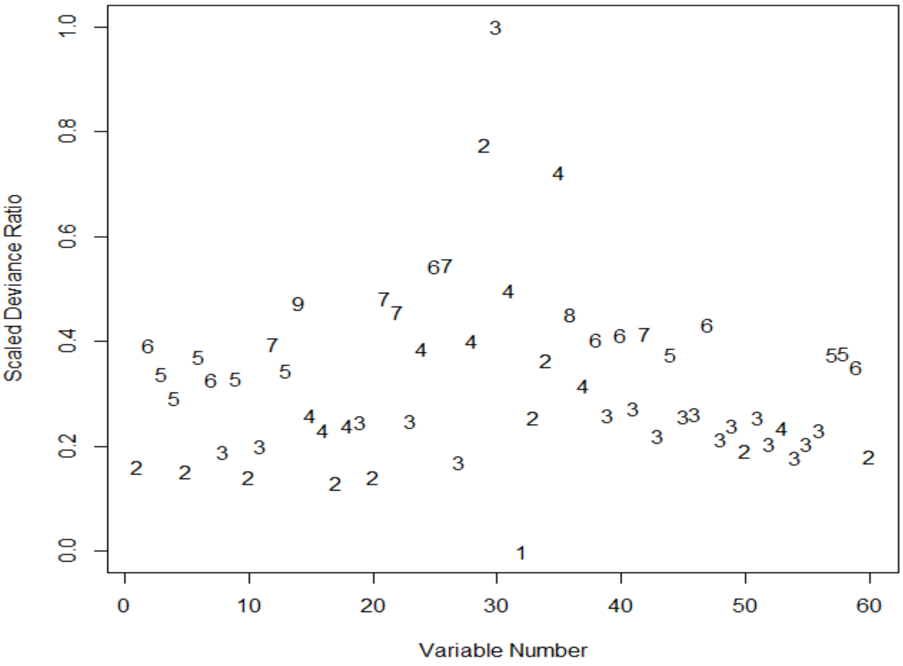

Figure 2: Example of plotting a "treeClust" object. This object contained 59 trees, the first of which had two leaves (digit "2" at (1, 0.16)).

and two times the impact can take place (when the tree is being built, and when observations are being assigned to leaves).

When a value is missing in a predictor, the tree-building routine simply ignored that entry, adjusting its splitting criterion to account for sample size. This is a reasonable and automatic approach when missing values are rare. At prediction time, an observation that is missing a value at a split is adjudicated using a surrogate split (which is an "extra" split, highly concordant with the best split, built for this purpose), or a default rule if no surrogate split is available. When there are large numbers of missing values, however, this approach may be inadequate and we are considering alternatives.

When an observation is missing a value of the response, that row is omitted in the building of the tree, but we then use the usual prediction routines to deduce the leaf into which that observation should fall, based on its predictors.

Parallel processing

The computations required by `treeClust()` can be substantial for large data sets with large numbers of columns, or with categorical variables with large numbers of levels. The current version of the package allows the user to require paralellization explicitly, by setting the `parallelnodes` argument to `treeClust.control()` to be greater than one. In that case, a set of nodes of that size is created using the `makePSOCKcluster()` function from R's **parallel** package, and the `clusterApplyLB()` function from the same package invoked. The parallel nodes then execute the `treeClust.rpart()` tree-building function once for each variable. The time savings is not substantial for experiments on the splice data, experiments which we describe below, because even in serial mode the computations require only 30 seconds or so to build the trees. However, on a dataset with many more columns the savings can be significant. For example, on the Internet Ads data (Bache and Lichman, 2013), which contains 1,559 columns, building the trees required about 57 minutes in serial on our machine, and only about 8 minutes when a set of 16 nodes was used in parallel.

Big data and efficiency considerations

Not only can our approach be computationally expensive, but, if the user elects to preserve the set of trees and also the inter-point dissimilarities, the resulting objects can be quite large. For example, the object built using the splice data in the example in the example below is around 60 MB in size.

Of this, about 40 MB is devoted to the vector of dissimilarities and almost all of the rest to the set of trees. For larger data sets the resulting "treeClust" objects can be very much larger. The d_4 method, in particular, requires that all the trees be kept, at least long enough to compute the dissimilarities, since the necessary weightings are not known until the final tree is produced. In constrast the d_3 dissimilarities can be produced using only one tree at a time. Both of these two dissimilarities require more computation and storage than the d_1 and d_2 ones.

Some clustering techniques require the vector of dissimilarities be computed ahead of time when the data contains categorical variables, and this large vector is produced by `treeClust()` in the normal course of events. When the sample size is large, a more computationally efficient approach is to produce a new data set which has the attribute that its inter-point dissimilarities are the same as, or similar to, the `treeClust()` distances. That new data set can then serve as the input to a clustering routine like, say, *k*-means, which does not require that all the inter-point distances be computed and stored. The next section describes how the `treeClust()` function implements this idea.

Dissimilarity-preserving data

In this section we describe one more output from the `treeClust()` function that we believe will be useful to researchers. As we noted above, the act of computing and storing all $n(n-1)$ pairwise distances is an expensive one. The well-known *k*-means algorithm stays away from this requirement, but is implemented in R only for numeric data sets. Researchers have taken various approaches to modifying the algorithm to accept categorical data, for example, Ahmad and Dey (2007). Our approach is to produce a data set in which the inter-point distances are similar to the ones computed by the `treeClust()` function, but arrayed as a numeric data set suitable for the R implementation of *k*-means or for `clara()`, the large-sample version of `pam()`. We call such a data set `newdata`.

As background, we remind the reader of the so-called Gower dissimilarity (Gower, 1971). Given two vectors of observations, say x_i and x_j, the Gower dissimilarity between those two vectors associated with a numeric column m is $g_m(i,j) = |x_{im} - x_{jm}| / \text{range } x_m$, where the denominator is the range of all the values in that column. That is, the absolute difference is scaled to fall in $[0,1]$. Then, in a data set with all numeric columns, the overall Gower distance $g(i,j)$ is a weighted sum, $g(i,j) = \sum_m w_m g_m(i,j) / \sum_m w_m$, where the w_m are taken to be identically 1, and therefore the Gower dissimilarity between any two observations is also in $[0,1]$. There is a straightforward extension to handle missing values we will ignore here.

The construction of `newdata` is straightforward for the d_1 and d_2 distances. When a tree has, say, l leaves, we construct a set of l 0-1 variables, where 1 indicates leaf membership. The resulting data set has Manhattan (sum of absolute values) distances that are exact multiples of the Gower dissimilarities we would have constructed from the matrix of leaf membership. (The Manhattan ones are larger by a factor of (2 × number of trees)). However, the `newdata` data set can be much smaller than the set of pairwise dissimilarities. In the splice data, the sample size $n = 3{,}190$, so the set of pairwise dissimilarities has a little more than 5,000,000 entries. The corresponding `newdata` object has 3,190 rows and 279 columns, so it requires under 900,000 entries. As sample sizes get larger the tractability of the `newdata` approach grows.

For d_2, we replace the values of 1 that indicate leaf membership with the weights $q_t / \max_k(q_k)$ described earlier, so that a tree t with l leaves contributes l columns, and every entry in any of those l columns is either a 0 or the value $q_t / \max_k(q_k)$. In this case, as with d_1, the Manhattan distance between any two rows is a multiple of the Gower dissimilarity that we would have computed with the leaf membership matrix, setting the weights $w_m = q_m / max_k q_k$.

We note that, for d_1 and d_2, the `newdata` items preserve Gower distances, not Euclidean ones. In the d_1 case, for example, the Euclidean distances will be the square roots of the Manhattan ones, and the square root of a constant multiple of the Gower ones. Applying the *k*-means algorithm to a `newdata` object will apply that Euclidean-based algorithm to these data points that have been constructed with the Gower metric. We speculate that this combination will produce good results, but research continues in this area.

The d_3 and d_4 dissimilarities require some more thought. With these dissimilarities, each tree of l leaves gives rise to a set of $l(l-1)/2$ distances between pairs of leaves. We construct the $l \times l$ symmetric matrix whose i,j-th entry is the distance between leaves i and j. Then we use multidimensional scaling, as implemented in the built-in R function `cmdscale()`, to convert these distances into a $l \times (l-1)$ matrix of coordinates, such that the Euclidean distance between rows i and j in the matrix is exactly the value in the (i,j)-th entry of the distance matrix. Each observation that fell into leaf i of the tree is assigned that vector of $l-1$ coordinates. In the case of the d_4 dissimilarity, each element in the distance matrix for tree t is multiplied by the scaling value $q_t / \max_k q_k$.

The use of multidimensional scaling is convenient, but it produces a slight complication. When

d_3 or d_4 are used to construct a newdata object, each tree contributes a set of coordinates such that, separately, Euclidean distances among those points match the d_3 or d_4 dissimilarities for that tree. However, across all coordinates the newdata distances are not generally the same as the d_3 or d_4 ones. In fact the d_3 distances are Manhattan-type sums of Euclidean distacnes. The properties of this hybrid-type distance have yet to be determined.

An example

In this section we give an example of clustering with treeClust(). This code is also included in the 'treeClustExamples.R' script that is included with the paper. We use the splice data mentioned above (and this data set is also included with the paper.) This data starts with 3,190 rows and 63 columns, but after removing the class label and two useless columns, we are left with a data set named splice that is 3,190 × 60. The class label is stored in an item named splice.class. (A shorter version of the data set and class vector, from which 16 rows with ambiguous values have been removed, is also constructed in the script.) At the end of the section we compare the performance of the treeClust() approach with the random forest dissimilarity approach.

Our goal in this example is to cluster the data, without using the class label, to see whether the clusters correspond to classes. We select distance d_4 and algorithm "pam", and elect to use six clusters for this example. Then (after we seed the random seed) the "treeClust" object is produced with a command like the one below. Here we are keeping the trees, as well as the dissimilarities, for later use:

```
set.seed(965)
splice.tc <- treeClust(splice, d.num = 4,
    control = treeClust.control(return.trees = TRUE, return.dists = TRUE),
    final.algorithm = "pam", k = 6)
```

This operation takes about 90 seconds on a powerful desktop running Windows; quite a lot of that time is used in the "pam" portion (the call with no final algorithm requires about 30 seconds). The resulting object can be plotted using plot(), and a short summary printed using summary():

```
plot(splice.tc); summary(splice.tc)
```

The splice.tc object is quite large, since it contains both the set of trees used to construct the distance and the set of dissimilarities. Had the call to treeClust.control() included cluster.only = TRUE, the function would have computed the clustering internally and returned only the vector of cluster membership. As it is, since a final.algorithm was specified, splice.tc contains an object called final.clust which is the output from the clustering algorithm – in this case, pam(). Therefore the final cluster membership can be found as a vector named splice.tc$final.clust$clustering.

We might evaluate how well the clustering represents the underlying class structure by building the two-way table of cluster membership versus class membership, as shown here. The outer pair of parentheses ensures both that the assignment is performed and also that its result be displayed.

```
(tc.tbl <- table(splice.tc$final.clust$clustering, splice.class))
# with result as seen here
   splice.class
     EI  IE  N,
  1 275  67  53
  2 246   0 246
  3 235 170  72
  4   6 529 198
  5   3   1 597
  6   2   1 489
```

Cluster six (bottom row) is almost entirely made up of items of class "N,". Others are more mixed, but the treeClust() dissimilarity compares favorably in this example to some of its competitors (Buttrey and Whitaker, 2015). For example, we compute the random forest dissimilarities in this way:

```
require(randomForest)
set.seed(965)
splice.rfd  <- as.dist(1 - randomForest(~ ., data = splice,
    proximity = TRUE)$proximity)
splice.pam.6 <- pam(splice.rfd, k = 6)
(rf.tbl <- table(splice.pam.6$cluster, splice.class))
```

That produces this two way-table:

```
  splice.class
     EI   IE   N,
 1   92   65  192
 2   19    3    7
 3  518  559 1035
 4   59   57  216
 5   75   69  195
 6    4   15   10
```

Clearly, in this example, the `treeClust()` dissimilarity has out-performed the random forest one. The `cramer()` function computes the value of V for these two-way tables, as shown here:

```
round(c(cramer(tc.tbl), cramer(rf.tbl)), 3)
[1] 0.679 0.113
```

If we prefer the d_3 dissimilarity, it can be computed with another call to `treeClust()`, or, in this case, directly from the `slice.tc` object, since the latter has all the necessary trees stored in it. The d_3 dissimilarity would be produced by this command:

```
splice.tc.d3 <- tcdist(splice.tc, d.num = 3)
```

As a final example, we can compute the `newdata` object whose inter-point distances reflect the d_4 distances with a call to `tcnewdist()`. This object can then be used as the input to, for example, `kmeans()`.

```
splice.d4.newdata <- tcnewdata(splice.tc, d.num = 4)
```

In this example the `splice.d4.newdata` object is about 1/7 the size of the vector of d_4 dissimilarities.

We note that the results of a call to `treeClust()` are random, because random numbers are used in the cross-validation process to prine trees. Therefore users might find slightly different results than those shown above.

Conclusion

This paper introduces the R package **treeClust**, which produces inter-point dissimilarities based on classification and regression trees. These dissimilarities can then be used in clustering. Four dissimilarities are available; in the simplest, the dissimilarity between two points is proportional to the number of trees in which they fall in different leaves. Exploiting the properties of trees, the dissimilarities are resistant to missing values and outliers and unaffected by linear scaling. They also benefit from automatic variable selection and are resistant, in some sense, to non-linear monotonic transformations of the variables.

For large data sets, the trees can be built in parallel to save time. Furthermore the package can generate numeric data sets in which the inter-point distances are related to the ones computed from the trees. This last attribute, while still experimental, may hold promise for the clustering of large data sets.

Bibliography

A. Ahmad and L. Dey. A k-mean clustering algorithm for mixed numeric and categorical data. *Data and Knowledge Engineering*, 63:503–527, 2007. [p233]

K. Bache and M. Lichman. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`. [p227, 231, 232]

L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001. [p228]

L. Breiman and A. Cutler. *Manual–Setting Up, Using, and Understanding Random Forests v4.0*, 2003. URL `https://www.stat.berkeley.edu/~breiman/Using_random_forests_v4.0.pdf`. [p228]

L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984. [p230]

S. E. Buttrey and L. R. Whitaker. A scale-independent, noise-resistant dissimilarity for tree-based clustering of mixed data. In submission, Feb. 2015. [p227, 228, 234]

J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Sunderland, MA, 2003. [p229]

D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987. [p228]

J. C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, 27(4):857–871, 1971. [p233]

M. Maechler, P. Rousseeuw, A. Struyf, M. Hubert, and K. Hornik. *cluster: Cluster Analysis Basics and Extensions*, 2015. URL `https://CRAN.R-project.org/package=cluster`. R package version 2.0.3. [p134, 228]

B. O'Meara. CRAN task view: Phylogenetics, especially comparative methods, 2015. URL `https://CRAN.R-project.org/view=Phylogenetics`. [p229]

H. Ooi. Density visualization and mode hunting using trees. *Journal of Computational and Graphical Statistics*, 11(2):328–347, 2002. [p228]

B. Ripley. *tree: Classification and regression trees*, 2015. URL `https://CRAN.R-project.org/package=tree`. R package version 1.0-36. [p231]

T. Shi and S. Horvath. Unsupervised learning with random forest predictors. *Journal of Computational and Graphical Statistics*, 15(1):118–138, 2006. [p228]

C. Smyth, D. Coomans, and Y. Everingham. Clustering noisy data in a reduced dimension space via multivariate regression trees. *Pattern Recognition*, 39:424–431, 2006. [p228]

T. Therneau, B. Atkinson, and B. Ripley. *rpart: Recursive Partitioning and Regression Trees*, 2015. URL `https://CRAN.R-project.org/package=rpart`. R package version 4.1-10. [p229]

Yuji. Search for corresponding node in a regression tree using rpart, 2015. URL `http://stackoverflow.com/questions/5102754/search-for-corresponding-node-in-a-regression-tree-using-rpart`. [p231]

Samuel E. Buttrey
Department of Operations Research
Naval Postgraduate School
USA
`buttrey@nps.edu`

Lyn R. Whitaker
Department of Operations Research
Naval Postgraduate School
USA
`whitaker@nps.edu`