Clustering Through Decision Tree Construction

Bing Liu
School of Computing
National University of Singapore
3 Science Drive 2
Singapore 117543
liub@comp.nus.edu.sg

Yiyuan Xia
School of Computing
National University of Singapore
3 Science Drive 2
Singapore 117543
xiayy@comp.nus.edu.sg

Philip S. Yu
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
USA
psyu@watson.ibm.com

ABSTRAT

Clustering aims to find the intrinsic structure of data by organizing data objects into similarity groups or clusters. It is often called unsupervised learning as no class labels denoting an *a priori* partition of the objects are given. This is in contrast with supervised learning (e.g., classification) for which the data objects are already labeled with known classes. Past research in clustering has produced many algorithms. However, these algorithms have some major shortcomings. In this paper, we propose a novel clustering technique, which is based on a supervised learning technique called decision tree construction. The new technique is able to overcome many of these shortcomings. The key idea is to use a decision tree to partition the data space into cluster and empty (sparse) regions at different levels of details. The technique is able to find "natural" clusters in large high dimensional spaces efficiently. It is suitable for clustering in the full dimensional space as well as in subspaces. It also provides comprehensible descriptions of clusters. Experiment results on both synthetic data and real-life data show that the technique is effective and also scales well for large high dimensional datasets.

1. INTRODUCTION

Clustering is an important exploratory data analysis task. It aims to organize objects (data records) into similarity groups or clusters. Clustering is often called unsupervised learning as no classes denoting an *a priori* partition of the objects are known. This is in contrast with supervised learning (e.g., classification), for which the data records are already labeled with known classes.

In this paper, we study clustering in a numerical space. Clusters in such a space are commonly defined as connected regions in the space containing a relatively high *density* of points, separated from other such regions by sparse regions [11].

Clustering has been studied extensively in the past [e.g., 1, 3, 4, 6, 8, 10, 11, 12, 15, 19, 20, 21, 23, 26, 29, 31, 33]. Many algorithms have been reported. However, these algorithms have some major shortcomings. In this paper, we propose a novel clustering technique, which is based on a supervised learning method called decision tree construction [27]. The new technique, called CLTree (*CL*ustering based on decision *Tree*s), is fundamentally different from the existing methods. It is able to overcome many of their shortcomings. To distinguish from normal decision trees, we call the trees produced by CLTree the *cluster trees*.

Decision tree building is a popular technique for classifying data of various classes (at least two classes). Its algorithm uses a *purity function* to partition the data space into different class regions. The technique is not directly applicable to clustering because datasets for clustering have no pre-assigned class labels. We will present a method to solve this problem.

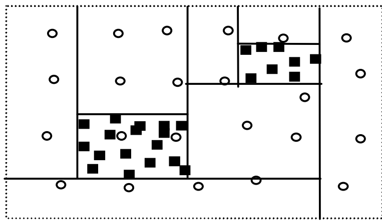
The basic idea is that we regard each data record (or point) in the dataset to have a class *Y*. We then assume that the data space is uniformly distributed with another type of points, called *non-existing points*. We give them the class, *N*. With the *N* points added to the original data space, our problem of partitioning the data space into *data* (*dense*) *regions* and *empty* (*sparse*) *regions* becomes a classification problem. A decision tree algorithm can be applied to solve the problem. However, for the technique to work many important issues have to be addressed (see Section 2).

We now use an example to show the intuition behind the proposed technique. Figure 1(A) gives a 2-dimensional space, which has 24 data (*Y*) points. Two clusters exist in the space. We then add some uniformly distributed *N* points (represented by "**o**") to the data space (Figure 1(B)). With the augmented dataset, we can run a decision tree algorithm to obtain a partitioning of the space (Figure 1(B)). The two clusters are identified.

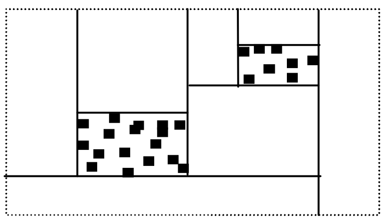
The reason that this technique works is that if there are clusters in the data, the data points cannot be uniformly distributed in the entire space. By adding some uniformly distributed *N* points, we can isolate the clusters because within each cluster region there are more *Y* points than *N* points. The decision tree technique is well known for this task.

We now answer two immediate questions: (1) how many *N* points should we add, and (2) can the same task be performed without physically adding the *N* points to the data? The answer to the first question is that it depends. The number changes as the tree grows (see Section 2.2). The answer to the second question is *yes*. Physically adding *N* points increases the size of the dataset and also the running time. A subtle but important issue is that it is unlikely that we can have points truly uniformly distributed in a very high dimensional space as we would need an exponential number of points [22]. We propose a technique to solve the problem. This is done by not adding any *N* point to the space but computing them when needed. Hence, CLTree is able to produce the partition in Figure 1(C) with no *N* point added to the data.

(A)



(B)



(C)

Figure 1: Clustering using decision trees: an intuitive example

The CLTree technique consists of two steps:

1. Cluster tree construction: This step uses a modified decision tree algorithm with a new purity function to construct a cluster tree to capture the natural distribution of the data without making any prior assumptions.
2. Cluster tree pruning: After the tree is built, an interactive pruning step is performed to simplify the tree to find meaningful/useful clusters. The final clusters are expressed as a list of hyper-rectangular regions.

The rest of the paper develops the idea further. Experiment results on both synthetic data and real-life application data show that the proposed technique is very effective and scales well for large high dimensional datasets.

1.1 Our Contributions

The main contribution of this paper is that it proposes a novel clustering technique, which is based on a supervised learning method [27]. It is fundamentally different from existing clustering techniques. Existing techniques form clusters explicitly by grouping data points using some distance or density measures. The proposed technique, however, finds clusters implicitly by separating data and empty (sparse) regions using a purity function based on the information theory (the detailed comparison with related work appears in Section 6). The new method has many distinctive advantages over the existing methods (although some existing methods also have some of the advantages, there is no system that has all the advantages):

- CLTree is able to find "natural" or "true" clusters because its tree building process classifies the space into data (dense) and empty (sparse) regions without making any prior assumptions or using any input parameters. Most existing methods require the user to specify the number of clusters to be found and/or density thresholds [e.g., 26, 33, 20, 6, 12, 1, 2, 3, 9, 22]. Such values are normally difficult to provide, and can be quite arbitrary. As a result, the clusters found may not reflect the "true" grouping structure of the data.
- CLTree is able to find clusters in the full dimension space as well as in subspaces. It is noted in [3] that many algorithms that work in the full space do not work well in subspaces of a high dimensional space. The opposite is also true, i.e., existing subspace clustering algorithms only find clusters in low dimension subspaces [1, 2, 3]. Our technique is suitable for both types of clustering because it aims to find simple descriptions of the data (using as fewer dimensions as possible), which may use all the dimensions or any subset.
- It provides descriptions of the resulting clusters in terms of hyper-rectangle regions. Most existing clustering methods only group data points together and give a centroid for each cluster with no detailed description. Since data mining applications typically require descriptions that can be easily assimilated by the user as insight and explanations, interpretability of clustering results is of critical importance.
- It comes with an important by-product, the empty (sparse) regions. Although clusters are important, empty regions can also be very useful. For example, in a marketing application, clusters may represent different segments of existing customers of a company, while the empty regions are the profiles of non-customers. Knowing the profiles of non-customers allows the company to probe into the possibilities of modifying the services or products and/or of doing targeted marketing in order to attract these potential customers.
- It deals with outliers effectively. Outliers are data points in a relatively empty region. CLTree is able to separate outliers from real clusters because it naturally identifies sparse and dense regions. When outliers are concentrated in certain areas, it is possible that they will be identified as small clusters. If such outlier clusters are undesirable, we can use a simple threshold on the size of clusters to remove them. However, sometimes such small clusters can be very useful as they may represent exceptions (or unexpected cases) in the data. The interpretation of these small clusters is dependent on applications.

1.2 Applications of the CLTree Algorithm

Apart from being used as an algorithm for clustering itself, the CLTree algorithm may also be used by other clustering techniques for the following two purposes:

1. Most existing clustering algorithms do not provide interpretable descriptions of the resulting clusters. The CLTree algorithm may be used to generate such a description for each cluster (in terms of hyper-rectangles) after the cluster has been found by an existing clustering algorithm. This can be done by running the CLTree algorithm using the data points within this cluster.
2. Most existing clustering algorithms also require the number of

clusters (k) as the input parameter, and are sensitive to initial seeds. The CLTree algorithm may be used to produce a set of initial seeds and also k as CLTree is able to find the core of each cluster. This will be clear later.

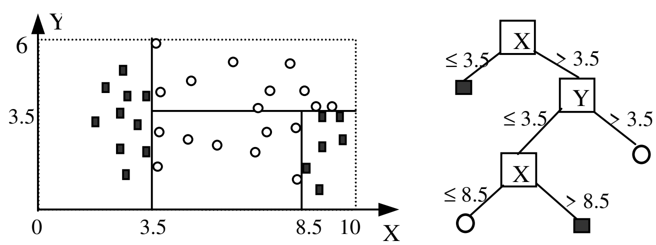
2. BUILDING CLUSTER TREES

This section presents our cluster tree algorithm. Since a cluster tree is basically a decision tree for clustering, we first review the decision tree algorithm in [27]. We then modify the algorithm and its purity function for clustering.

2.1 Decision Tree Construction

Decision tree construction is a classic technique for classification. A database for decision tree classification consists of a set of data records that are pre-classified into q (≥ 2) known classes. The objective of decision tree construction is to partition the data to separate the q classes. A decision tree has two types of nodes, *decision nodes* and *leaf nodes*. A decision node specifies some test on a single attribute. A leaf node indicates the class.

From a geometric point of view, a decision tree represents a partitioning of the data space. A serial of tests (or cuts) from the root node to a leaf node represents a hyper-rectangle. For example, the four hyper-rectangular regions in Figure 2(A) are produced by the tree in Figure 2(B). A region represented by a leaf can also be expressed as a rule, e.g., the upper right region in Figure 2(A) can be represented by $X > 3.5$, $Y > 3.5 \rightarrow$ o, which is also the right most leaf in Figure 2(B). Note that for a numeric attribute, the tree algorithm in [27] performs binary split, i.e., each cut splits the current space into two parts (see Figure 2(B)).



(A) **(B)**

Figure 2. A decision tree example

The algorithm in [27] for building a decision tree uses the divide and conquer strategy to recursively partition the data to produce the tree. Each successive step greedily chooses the best cut to partition the space into two parts in order to obtain purer regions. A commonly used criterion (or *purity function*) for choosing the best cut is the *information gain* (see [27] for the detailed computation of information gain). The criterion selects the test or cut that maximizes the gain in information to partition the current data (or space). The procedure for gain evaluation is given in Figure 3. It evaluates every possible value (or cut point) on all dimensions to find the cut point that gives the best gain.

```
for each attribute A_i ∈ {A_1, A_2, …, A_d} of the dataset D do
    for each value x of A_i in D do
                    /* each value is considered as a possible cut */
        Compute the information gain at x
end
Select the test or cut that gives the best information gain to
partition the space
```

Figure 3. The information gain evaluation

Scale-up decision tree algorithms: Traditionally, a decision tree algorithm requires the whole data to reside in memory. When the dataset is too large, techniques from the database community can be used to scale up the algorithm so that the entire dataset is not required in memory. SPRINT [28] and RainForest [17] propose two scalable techniques for decision tree building. For example, RainForest only keeps an AVC-set (attribute-value, classLabel and count) for each attribute in memory. This is sufficient for tree building and gain evaluation. It eliminates the need to have the entire dataset in memory. BOAT [18] uses statistical techniques to construct the tree based on a small subset of the data, and correct inconsistency due to sampling via a scan over the database.

2.2 Introducing *N* Points

We now present the modifications made to the decision tree algorithm in [27]. This sub-section focuses on introducing N points. The next sub-section discusses two changes that need to be made to the decision tree algorithm. The final sub-section describes the new cut selection criterion or purity function.

As mentioned before, we give each data point in the original dataset the class Y, and introduce some uniformly distributed "non-existing" N points. Note that we do not physically add these N points to the original data, but only assume their existence.

We now determine how many N points to add. We add a different number of N points at each node. The number of N points for the current node E is determined by the following rule (note that at the root node, the number of inherited N points is 0):

```
If  the number of N points inherited from the parent node of E
        is less than the number of Y points in E then
        the number of N points for E is increased to the number of
        Y points in E
else the number of inherited N points is used for E
```

Figure 4 gives an example. The (parent) node P has two children nodes L and R. Assume P has 1000 Y points and thus 1000 N points, stored in $P.Y$ and $P.N$ respectively. Assume after splitting, L has 20 Y points and 500 N points, and R has 980 Y points and 500 N points. According to the above rule, for subsequent partitioning, we increase the number of N points at R to 980. The number of N points at L is unchanged.

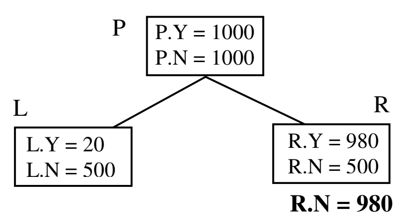


Figure 4. Distributing *N* points

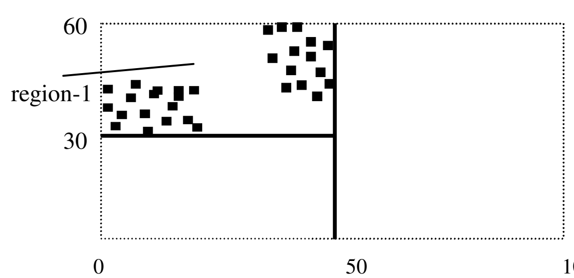


Figure 5. The effect of using a fixed number of *N* points

The basic idea is that we use an equal number of *N* points to the number of *Y* (data) points (in fact 1:1 ratio is not necessary, see [25]). This is natural because it allows us to isolate those regions that are densely populated with data points. The reason that we increase the number of *N* points of a node (line 2) if it has more inherited *Y* points than *N* points is to avoid the situation where there may be too few *N* points left after some cuts or splits. If we fix the number of *N* points in the entire space to be the number of *Y* points in the original data, the number of *N* points at a later node can easily drop to a very small number for a high dimensional space. If there are too few *N* points, further splits become difficult, but such splits may still be necessary. Figure 5 gives an example. In Figure 5, the original space contains 32 data (*Y*) points. According to the above rule, it also has 32 *N* points. After two cuts, we are left with a smaller region (region 1). All the *Y* points are in this region. If we do not increase the number of *N* points for the region, we are left with only $32/2^2 = 8$ *N* points in region 1. This is not so bad because the space has only two dimensions. If we have a very high dimensional space, the number of *N* points will drop drastically (close to 0) after some splits (as the number of *N* points drops exponentially).

The number of *N* points is not reduced if the current node is an *N* node (*an N node has more N points than Y points*) (line 3). A reduction may cause outlier *Y* points to form *Y* nodes (*a Y node has an equal number of Y points as N points or more*). Then cluster regions and non-cluster regions may not be separated.

2.3 Two Modifications to the Decision Tree Algorithm

Since the *N* points are not physically added to the data, we need to make two modifications to the decision tree algorithm in [27]:

Compute the number of N points on the fly: The information gain evaluation in [27] only needs the frequency or the number of points of each class on each side of a possible cut (or split). Since we do not have the *N* points in the data, we need to compute them. This is simple because we assume that the *N* points are uniformly distributed in the space. Figure 6 shows an example. The space has 25 data (*Y*) points and 25 *N* points. Assume the system is evaluating a possible cut *P*. The number of *N* points on the left-hand-side of *P* is 25 * 4/10 = 10. The number of *Y* points is 3. Likewise, the number of *N* points on the right-hand-side of *P* is 15 (= 25 - 10), and the number of *Y* points is 22. With these numbers, the information gain at *P* can be computed. Note that by computing the number of *N* points, we essentially guarantee their uniform distribution in the current space.

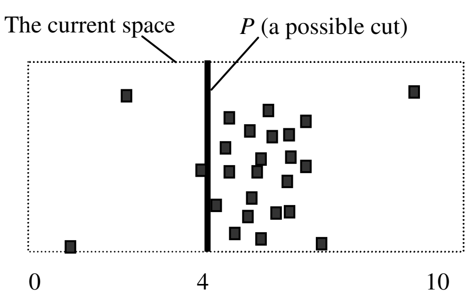


Figure 6. Computing the number of *N* points

Evaluate on both sides of data points: In the standard decision tree building, cuts only occur on one side of data points [27]. However, for our purpose, this is not adequate as the example in Figure 7 shows. Figure 7 gives 3 possible cuts. cut_1 and cut_3 are on the right-hand-side of some data points, while cut_2 is on the left-hand-side. If we only allow cuts on the right-hand-side of data points, we will not be able to obtain a good cluster description. If we use cut_1, our cluster will contain a large empty region. If we use cut_3, we lose many data points. In this case, cut_2 is the best. It cuts on the left-hand-side of the data points.

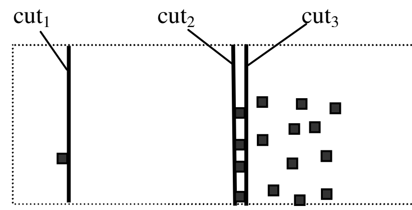


Figure 7. Cutting on either side of data points

2.4 The New Criterion

Decision tree building for classification uses the gain criterion [27] to select the best cut. For clustering, this is not satisfactory. The cut that gives the best gain may not be the best cut for clustering. There are two main problems with the gain criterion:

1. The cut with the best information gain tends to cut into clusters. This results in severe loss of data points in clusters.
2. The gain criterion does not look ahead in deciding the best cut.

Let us see an example. Figure 8(A) shows a space with two clusters, which illustrates the first problem. Through gain computation, we find the best cuts for dimension 1 (d_1_cut), and for dimension 2 (d_2_cut) respectively. Clearly, both cuts are undesirable because they cut into clusters. Assume d_1_cut gives a better information gain than d_2_cut. We will use d_1_cut to partition the space. The cluster points on the right of d_1_cut from both clusters are lost.

This problem occurs because at cluster boundaries there is normally a higher proportion of *N* points than that of cluster centers for clusters whose data points follow a normal-like distribution (cluster centers are much denser than boundaries). The gain criterion will find a balanced point for partitioning, which tends to be somewhere inside the clusters.

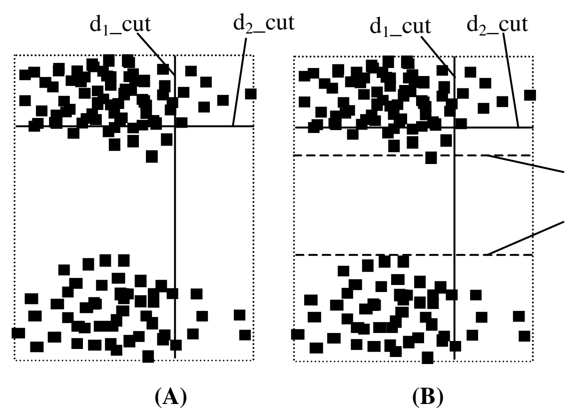


Figure 8. Problems with the gain criterion

Next, we look at the second problem using Figure 8(B). Ideally, in this situation, we should cut at d_2_cut2 or d_2_cut3, rather than

d_1_*cut* (although it gives the highest gain). However, using the gain criterion, we are unable to obtain d_2_*cut2* or d_2_*cut3* because the gain criterion does not look ahead to find better cuts. There is a piece of important information that the gain criterion is unable to capture, the empty region between d_2_*cut2* and d_2_*cut3*. Recognizing the empty region is very important for clustering.

The two problems result in severe fragmentation of clusters (each cluster is cut into many pieces) and loss of data points. Our first implementation was based on the information gain criterion alone, and experiments showed that it is unsatisfactory.

To overcome these problems, we designed a new criterion, which still uses information gain as the basis, but adds to it the ability to look ahead. We call the new criterion the *lookahead gain criterion*. For the example in Figure 8(B), we aim to find a cut that is very close to d_2_*cut2* or d_2_*cut3*.

The basic idea is as follows: For each dimension i, based on the first cut found using the gain criterion, we look ahead (at most 2 steps) to find a better cut c_i that cuts less into cluster regions, and to find an associated region r_i that is relatively empty (measured by *relative density*, see below). c_i of the dimension i whose r_i has the lowest relative density is selected as the best cut. The intuition behind this modified criterion is clear. It tries to find the emptiest region along each dimension to separate clusters.

Definition (relative density): The *relative density* of a region r is computed with $r.Y / r.N$, where $r.Y$ and $r.N$ are the number of Y points and the number of N points in r respectively.

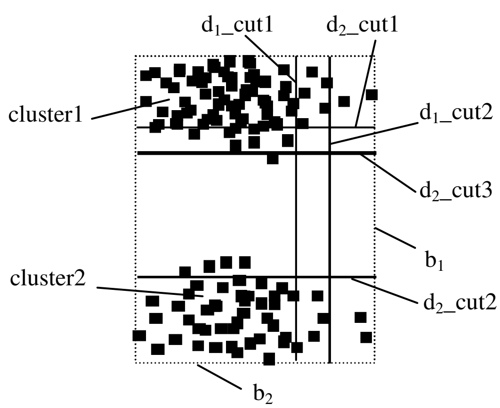


Figure 9. Determining the best cut

We use the example in Figure 9 (a reproduction of Figure 8(A)) to introduce the lookahead gain criterion. The algorithm is given in Figure 10. The new criterion consists of 3 steps:

1. **Find the initial cuts** (line 2, Figure 10): For each dimension i, we use the gain criterion to find the first best cut point d_i_*cut1*. For example, in Figure 9, for dimension 1 and 2, we find d_1_*cut1*, and d_2_*cut1* respectively. If we cannot find d_i_*cut1* with any gain for a dimension, we ignore this dimension subsequently.
2. **Look ahead to find better cuts** (lines 3 and 6, Figure 10): Based on the first cut, we find a better cut on each dimension by further gain evaluation. Our objectives are to find:
   (1) a cut that cuts less into clusters (to reduce the number of lost points), and
   (2) an associated region with a low relative density (*relatively empty*).

Algorithm *evaluateCut*(D)
1 **for** each attribute $A_i \in \{A_1, A_2, \ldots, A_d\}$ **do**
2 d_i_*cut1* = the value (cut) of A_i that gives the best gain on dimension i;
3 d_i_*cut2* = the value (cut) of A_i that gives the best gain in the L_i region produced by d_i_*cut1*;
4 **if** the relative density between d_i_*cut1* and d_i_*cut2* is higher than that between d_i_*cut2* and b_i **then**
5 *r_density*$_i$ = y_i / n_i, where y_i and n_i are the numbers of Y points and N points between d_i_*cut2* and b_i
6 **else** d_i_*cut3* = the value (cut) that gives the best gain in the L_i region produced by d_i_*cut2*;
/* L_i here is the region between d_i_*cut1* and d_i_*cut2* */
7 *r_density*$_i$ = y_i/n_i, where y_i and n_i are the numbers of Y points and N points in the region between d_i_*cut1* and d_i_*cut3* or d_i_*cut2* and d_i_*cut3* that has a lower proportion of Y points (or a lower relative density).
8 **end**
9 **end**
10 *bestCut* = d_i_*cut3* (or d_i_*cut2* if there is no d_i_*cut3*) of dimension i whose *r_density*$_i$ is the minimal among the d dimensions.

Figure 10. Determining the best cut in CLTree

Let us denote the two regions separated by d_i_*cut1* along dimension i as L_i and H_i, where L_i has a lower relative density than H_i. d_i_*cut1* forms one boundary of L_i along dimension i. We use b_i to denote the other boundary of L_i. To achieve both (1) and (2), we only find more cuts (at most two) in L_i. We do not go to H_i because it is unlikely that we can find better cuts there to achieve our objectives (since H_i is denser). This step goes as follows:

Along each dimension, we find another cut (d_i_*cut2*) in L_i that gives the best gain. After d_i_*cut2* is found, if the relative density of the region between d_i_*cut1* and d_i_*cut2* is higher than that between d_i_*cut2* and b_i, we stop because both objectives are achieved. If not, we find the third cut (d_i_*cut3*) by applying the same strategy. We seek the additional cut in this case because if the region between d_i_*cut2* and b_i is denser, it means that there may be clusters in that region. Then, d_i_*cut2* is likely to cut into these clusters.

For example, in Figure 9, we first obtain d_1_*cut2* and d_2_*cut2*. Since the relative density of the region between d_1_*cut1* and d_1_*cut2* is higher than that between d_1_*cut2* and the boundary on the right (b_1), we stop for dimension 1. We have found a better cut d_1_*cut2* and also a low density region between d_1_*cut2* and the right space boundary (b_1).

However, for dimension 2, we now obtain a situation (in the region between d_2_*cut1* and the bottom space boundary, b_2) like that for dimension 1 before d_1_*cut2* is found. d_2_*cut2* cuts into another cluster. We then apply the same method to the data points and the region between d_2_*cut1* and d_2_*cut2* since the relative density is lower between them, another local best cut d_2_*cut3* is found, which is a much better cut, i.e., cutting almost at the cluster boundary. We now have two good cuts d_1_*cut2* and d_2_*cut3* for dimension 1 and 2 respectively. We also found two low density regions associated with the cuts, i.e., the region between d_1_*cut2* and the right space boundary (b_1) for dimension 1, and the region between d_2_*cut2* and d_2_*cut3* for dimension 2.

3. **Select the overall best cut** (line 5, 7 and 10): We compare the relative densities ($r\_density_i$) of the low density regions identified in step 2 of all dimensions. The best cut in the dimension that gives the lowest $r\_density_i$ value is chosen as the best cut overall. In our example, the relative density between d_2_cut2 and d_2_cut3 is clearly lower than that between d_1_cut2 and the right space boundary, thus d_2_cut3 is the overall best cut.

The reason that we use relative density to select the overall best cut is because it is desirable to split at the cut point that may result in a big empty (*N*) region (e.g., between d_2_cut2 and d_2_cut3), which is more likely to separate clusters.

Our algorithm can also be scaled up using the existing decision tree scale-up techniques in [17, 18, 28] since the essential computation here is the same as that in decision tree building, i.e., the gain evaluation. Our new criterion simply performs the gain evaluation more than once. See [25] for details on the scale-up.

3. USER-ORIENTED PRUNING OF CLUSTER TREES

The recursive partitioning method of building cluster trees will divide the data space until each partition contains only points of a single class, or until no test (or cut) offers any improvement. The result is often a very complex tree that partitions the space more than necessary. This problem is the same as that in classification [27]. We need to prune the tree to produce meaningful clusters.

Pruning methods used for classification, however, cannot be applied here because clustering, to certain extent, is a subjective task. Whether a clustering is good or bad depends on the application and the user's subjective judgment of its usefulness [8, 23]. Thus, we use a subjective measure for pruning. We use the example in Figure 11 to explain.

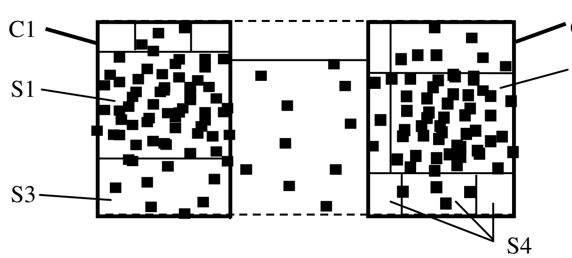


Figure 11. How many clusters are there?

The original space in Figure 11 is partitioned into 14 regions by the cluster tree. It is not clear whether we should report one cluster (the whole space) or two clusters. If we are to report two clusters, should we report the regions C1 and C2, or S1 and S2? The answers to these questions depend on the specific application.

We propose two interactive approaches to allow the user to explore the cluster tree to find meaningful/useful clusters.

***Browsing*:** The user simply explores the tree him/herself to find meaningful clusters (prune the rest). This is not a difficult task as the major clusters are identified at the top levels of the tree. A user interface has been built to facilitate this exploration.

***User-oriented pruning*:** The tree is pruned using two user-specify parameters. After pruning, we summarize the clusters by extracting only those *Y* leaves and express them with hyper-rectangular regions (each *Y* leaf naturally forms a region or a rule (see Section 2.1)). The user can then view and study them.

The two parameters used in pruning are as follows:

min_y: It specifies the minimal number of *Y* points that a region must contain (to be considered interesting). *min_y* is expressed as a percentage of $|D|$. That is, a node with fewer than $min\_y * |D|$ number of *Y* points is not interesting. For example, in Figure 11, if $min\_y * |D| = 6$, the number of *Y* points (which is 4) in S4 (before it is cut into three smaller regions) is too few. Thus, further cuts will not be considered, i.e., the two cuts in S4 are pruned. However, S4 may join S2 to form a bigger cluster.

min_rd: It specifies whether an *N* region (node) *E* should join an adjacent *Y* region *F* to form a bigger cluster region. If the relative density of *E*, *E.Y/E.N*, is greater than *min_rd*, where *E.Y* (or *E.N*) gives the number of *Y* (or *N*) points contained in *E*, then *E* and *F* should be combined to form a bigger cluster. For example, the *min_rd* value will decide whether S3 should join S1 to form a bigger cluster. If so, more data points are included in the cluster.

The pruning algorithm is given in Figure 12. We recursively descends down the tree in a depth first manner and then backs up level-by-level to determine whether a cluster should be formed by a *Y* node alone or by joining it with the neighboring node. If the two subtrees below a node can be pruned, the algorithm assigns TRUE to the *Stop* field (*node.Stop*) of the node data structure. Otherwise, it assigns FALSE to the field. Once the algorithm is completed, we simply descend down the tree again along each branch to find the first *Y* node whose *Stop* field is TRUE (not shown in Figure 12). These nodes are the clusters.

```
Algorithm evaluatePrune(Node, min_y, min_rd)
  if Node is a leaf then  Node.Stop = TRUE
  else  LeftChild = Node.left;
        RightChild = Node.right;
        if LeftChild.Y < min_y * |D|   then
             LeftChild.Stop = TRUE
        else   evaluatePrune(LeftChild, min_y, min_rd);
        end
        if   RightChild.Y < min_y * |D|  then
             RightChild.Stop = TRUE
        else   evaluatePrune(RightChild, min_y, min_rd);
        end
     if LeftChild.Stop = TRUE then
            /* We assume that the relative density of LeftChild is
            always higher than that of RightChild  */
        if  RightChild.Stop = TRUE then
          if RightChild.Y / RightChild.N > min_rd then
             Node.Stop = TRUE
                /* We can prune from Node either because we can
                join or because both children are N nodes. */
          elseif   LeftChild is an N node then
                Node.Stop = TRUE
          else  Node.Stop = FALSE
          end
        else     Node.Stop = FALSE
        end
     else        Node.Stop = FALSE
     end
  end
```

Figure 12: The cluster tree pruning algorithm

The evaluatePrune algorithm is linear to the number of nodes in the tree as it traverses the tree only once.

4 MERGING ADJACENT *Y* REGIONS AND SIMPLIFYING CLUSTERS

The cluster tree partitions the space into *Y* and *N* regions (represented by *Y* and *N* nodes). In a complex situation, it is possible that a cluster is split into several *Y* regions either because the cluster is of irregular shape or because in order to isolate another cluster it is cut into more than one piece. Figure 13 shows an example. The space contains two clusters. We see that cluster-1 is cut into three pieces. A post-processing step is needed to merge them to produce only one cluster. The idea is that two *Y* regions should merge if they *touch* each other.

Definition: A *Y* region, Y_1, is said to *touch* another *Y* region, Y_2, on the *i*th dimension on the lower bounding surface (or the upper bounding surface), if they meet on the *i*th dimension and intersect on all other dimensions. That is,

(1). $max_i(Y_1) = min_i(Y_2)$ (or $max_i(Y_2) = min_i(Y_1)$) and

(2). $min_j(Y_1) < max_j(Y_2)$ and $max_j(Y_1) > min_j(Y_2)$ for all $j \neq i$.

The worse case time complexity of merging is $O(r^2)$, where *r* is the number of *Y* regions found in the tree. Since *r* is very small after pruning, the computation is not a problem.

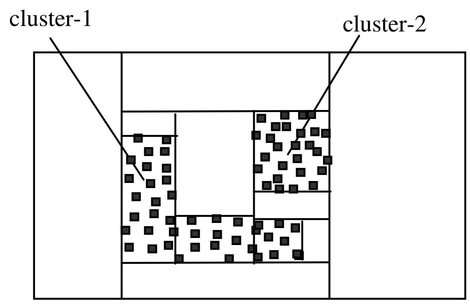


Figure 13. Fragmentation of clusters

Finding simpler descriptions of clusters: In Figure 13, we see that cluster-1 consists of three rectangles after merging. However, two rectangles are sufficient to describe this cluster. If the user is interested in finding a simpler description, we can *re-apply* the CLTree algorithm using only the data points in this cluster.

For example, cluster-1 in Figure 13 is described with only two regions in Figure 14. CLTree is able to perform this simplification task because the evaluation criterion for splitting in tree building aims to produce a simple description (although it does not guarantee to produce the minimal description).

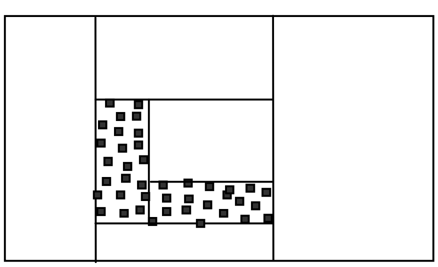


Figure 14. Generating a simpler description

5 PERFORMANCE EXPERIMENTS

We have evaluated CLTree using synthetic as well as real-life datasets. However, due to space limitations, we could only report part of the results using synthetic data. For other results with synthetic data and real-life data, see [25]. Our experiments aim to establish the following:

- Efficiency: Determine how the execution time scales with, dimensionality of clusters, size of datasets, dimensionality of datasets, and number of clusters in the data space.
- Accuracy: Test if CLTree finds known clusters in subspaces as well as in the full space of a high dimensional space. Since CLTree provides descriptions of clusters, we test how accurate the descriptions are with different pruning parameters.

  Note that we also tested how the ratio of the number of *N* and *Y* points, and the noise level affect the accuracy. Due to space limitations, they are not given here (see [25] for details).

All the experiments were run on SUN E450 with one 250MHZ cpu and 512MB memory.

5.1. Synthetic Data Generation

We implemented two data generators for our experiments. One generates datasets following a normal distribution, and the other generates datasets following a uniform distribution. For both data generators, all data points have coordinates in the range [0, 100] on each dimension. The percentage of noise or outliers (*noise level*) is a parameter.

Normal distribution: The first data generator generates data points in each cluster following a normal distribution. Each cluster is described by the subset of dimensions, the number of data points, and the value range along each cluster dimension. The data points for a given cluster are generated as follows: The coordinates of the data points on the non-cluster dimensions are generated uniformly at random. For a cluster dimension, the coordinates of the data points projected onto the dimension follow a normal distribution. The data generator is able to generate clusters of elliptical shape and also has the flexibility to generate clusters of different sizes.

Uniform distribution: The second data generator generates data points in each cluster following a uniform distribution. The clusters are hyper-rectangles in a subset of the dimensions. The surfaces of such a cluster are parallel to axes. The data points for a given cluster are generated as follows: The coordinates of the data points on non-cluster dimensions are generated uniformly at random over the entire value ranges of the dimensions. For a cluster dimension in the subspace in which the cluster is embedded, the value is drawn at random from a uniform distribution within the specified value range.

5.2. Scalability Results

For our experiments reported below, the noise level is set at 10%. The execution times (in sec.) do not include the time for pruning, but only tree building. Pruning is very efficient because we only need to traverse the tree once. The datasets reside in memory. The results with the data on disk are reported in [25]. When the data is on disk, the slowdown is about 30%.

Dimensionality of hidden clusters: CLTree can be used for finding clusters in the full dimensional space as well as in any subspaces. Figure 15 shows the scalability as the dimensionality of the clusters is increased from 2 to 20 in a 20-dimensional space. In each case, 5 clusters are embedded in different subspaces of the 20-dimensional space. In the last case, the clusters are in the full space. Each dataset has 100,000 records. From the figure, we see that when the clusters are hyper-rectangles (in which the data points are uniformly distributed), CLTree takes less time to build the tree. This is because CLTree can naturally find hyper-rectangular clusters, and thus tree building stopped earlier. For both normal and uniform distribution data, we obtain better than linear scale-up.

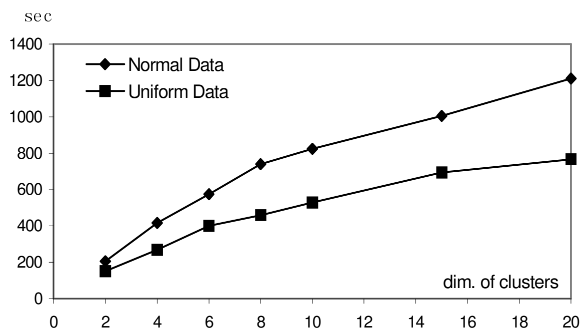


Figure 15. Scalability with the dimensionality of hidden clusters

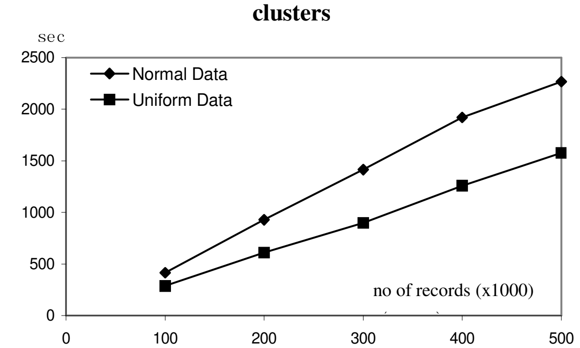


Figure 16. Scalability with the dataset size

Dataset size: Figure 16 shows the scalability as the size of the dataset is increased from 100,000 to 500,000 records. The data space has 20 dimensions, and 5 hidden clusters, each in a different 5-dimensional subspace. The execution time scales up linearly.

The scalability results with the dimensionality (linear scale-up), and the number of clusters (execution times do not vary a great deal as the number of clusters increases) are given in [25].

5.3. Accuracy and Sensitivity Results

In all the above experiments, CLTree recovers all the original clusters embedded in the full dimensional space and subspaces. All cluster dimensions and their boundaries are found without including any extra dimension. For pruning, we use CLTree's default settings of *min_y* = 1% and *min_rd* = 10% (see below).

Since CLTree provides precise cluster descriptions, which are represented by hyper-rectangles and the number of data (*Y*) points contained in each of them, we show the percentage of data points recovered in the clusters using various *min_y* and *min_rd* values. Two sets of experiments are conducted. In both sets, each data space has 20 dimensions, and 100,000 data points. In the first set, the number of clusters is 5, and in the other it is 10. Each cluster is in a different 5-dimensional subspace.

min_y: We vary the value of *min_y* from 0.05% to 5%, and set *min_rd* = 10% and noise level = 10%. Figure 17 gives the results. For uniform distribution, even *min_y* is very low, all the data points in the clusters are found for both 5 and 10 cluster cases. All the clusters and their dimensions are also recovered. For normal distribution, the percentage of data points found in the clusters is relatively low when *min_y* is very small (0.05%, 0.1% or 0.3%). It increases dramatically and stabilizes after *min_y* passes 0.5%. From *min_y* = 0.5, the percentages of data points (outliers are not counted) recovered are very high, around 95%.

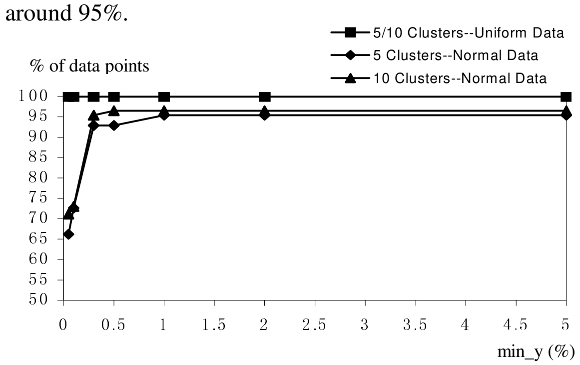


Figure 17. Percentage of cluster data points found with *min_y*

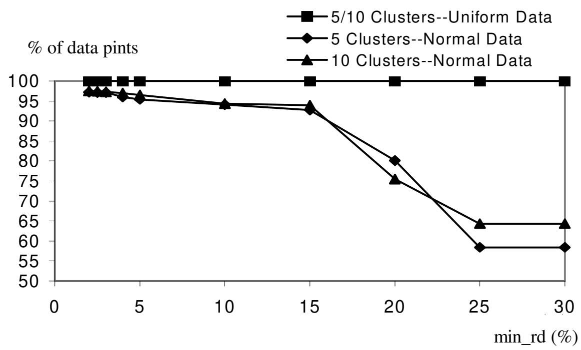


Figure 18. Percentage of cluster data points found with *min_rd*

min_rd: Figure 18 shows the percentages of data points found in the clusters with different values of *min_rd*. The noise level is set at 10% and *min_y* at 1%. The *min_rd* values in the range of 2-30% do not affect the number of cluster data points found in the uniform distribution datasets. The reason is that clusters in these datasets do not have the problem of low-density regions around the cluster boundaries as in normal distribution.

For the normal distribution datasets, when *min_rd* is small, more data points are found, 95% or more. When *min_rd* is too small, i.e., below 2% (not shown in Figure 18), some clusters are merged, which is not desirable. When *min_rd* is more than 15%, the number of data points recovered in the clusters starts to drop and reaches 58-64% when *min_rd* is 25% and 30%. That is, when the *min_rd* value is very high, we only find the core regions of the clusters. In all these experiments (except those with *min_rd* below 2%), pruning finds the correct number of clusters, and also the cluster dimensions. Only the number of data points contained in each cluster region changes with different *min_rd* values. The results shown in Figure 18 can be explained using Figure 11 (Section 3). If *min_rd* is set low, we will find C1 and C2. If *min_rd* is set too low we will find the whole space as one cluster. If it is set very high, we only find S1 and S2.

min_y and min_rd in applications: From the above experiment results, we see that the clustering results are not very sensitive to *min_y* and *min_rd* values. Although when *min_rd* is too high, we may lose many data points, but we can still find the core regions of the clusters. Thus, in a real-life application, it is rather safe to give both parameters high values, e.g., *min_y* = 1%-5% and *min_rd* = 10%-30%. After we have found the core of each cluster, we can lower down the values to find bigger cluster regions. Alternatively, we can explore the cluster tree ourselves from those nodes representing the cluster cores. As discussed in Section 1.2, the core may also be used as the initial seeds for other clustering algorithms.

6 RELATED WORK

6.1. Clustering Techniques

Traditional clustering techniques can be broadly categorized into *partitional clustering* and *hierarchical clustering* [23, 11]. Partitional clustering determines a partitioning of the data records into *k* clusters such that the data records in a cluster are nearer to one another than the records in different clusters [23, 11]. The main problem with partitional techniques is that they are often very sensitive to the initial seeds, and outliers [23, 6, 12]. CLTree does not have these problems.

Hierarchical clustering is a nested sequence of partitions. A clustering can be created by building a tree either from leaves to the root (*agglomerative approach*) or from the root down to the leaves (*divisive approach*). CLTree is different from partitional clustering because it does not explicitly group data points using distance comparison. It is different from hierarchical clustering because it does not merge the closest (or split the farthest) groups of records to form clusters. Instead, CLTree performs clustering by classifying data regions (*Y* regions) and empty regions (*N* regions) in the space.

Most traditional clustering methods have very high computational complexities. They are not suitable for clustering of large high dimensional datasets. In the past few years, a number of studies were made to scale up these algorithms (e.g., CLARANS [26], BIRCH [33], [6], DENCLUE [21], CURE [20]). These works are different from ours as our objective is not to scale up an existing algorithm, but to propose a new clustering technique that aims to overcome many problems with the existing methods.

Recently, some clustering algorithms based on local density comparisons and/or grids were reported, e.g., DBSCAN [9], DBCLASD [32], STING [31], WaveCluster [29] and DENCLUE [21]. Density-based approaches, however, are not effective in a high dimensional space because the space is too sparsely filled. These methods also cannot be used to find subspace clusters.

OptiGrid [22] finds clusters in high dimension spaces by projecting the data onto each axis and then partitioning the data using cutting planes at low-density points. The approach does not work effectively in situations where some well-separated clusters in the full space may overlap when they are projected onto each axis. For example in Figure 13, the two clusters overlap when they are projected onto either axis. They cannot be separated by any axis parallel cutting plane. Merging (as in Section 4) could not be done in OptiGrid because it does not have the concept of sparse regions that separate dense regions and it does not provide a understandable description of each cluster. OptiGrid also cannot find subspace clusters. These are not problems for CLTree.

CLIQUE [3] is a subspace clustering algorithm. It finds dense regions in each subspace of a high dimensional space. The algorithm uses equal-size cells and cell densities to determine clustering. The user needs to provide the cell size and the density threshold. This approach does not work effectively for clusters that involve many dimensions. According to the results reported in [3], the highest dimensionality of subspace clusters is only 10. Furthermore, CLIQUE does not produce disjoint clusters as normal clustering algorithms do. Dense regions at different subspaces typically overlap. This is due to the fact that for a given dense region all its projections on lower dimensionality subspaces are also dense, and get reported. [7] presents a system that uses the same approach as CLIQUE, but has a different measurement of good clustering. CLTree is different from this grid and density based approach because its cluster tree building does not depend on any input parameter. It is also able to find disjoint clusters of any dimensionality.

[1, 2] studies projected clustering, which is related to subspace clustering in [3], but find disjoint clusters as traditional clustering algorithms do. Unlike traditional algorithms, it is able to find clusters that use only a subset of the dimensions. The algorithm ORCLUS [2] is based on hierarchical merging clustering. In each clustering iteration, it reduces the number of clusters (by merging) and also reduces the number of dimensions. The algorithm assumes that the number of clusters and the number of projected dimensions are given beforehand. CLTree does not need such parameters. CLTree also does not require all projected clusters to have the same number of dimensions, i.e., different clusters may involve different numbers of dimensions (see also [25]).

In [24], we proposed a method to find sparse and data regions using decision trees. The method first divides the data space into equal-size grids. Those empty (low density) grids represent non-existing data points. However, as discussed earlier, this grid-based method does not work for high dimensional spaces. It is also hard to decide the size of the grids and the density threshold.

[14] proposes a technique for finding combinations of values of input variables (or regions in input space) that imply large (or small) values of an output (or target) variable. It uses a covering method to find a region at a time and then remove the data points in the region before finding the next region. The technique for finding each region is based on peeling. At each step, peeling removes a small region from the lower or upper boundary of a input variable (or dimension) that gives the highest average value of the target variable. It stops when a user-specified minimum support (the number of data points in the resulting region) is reached. [14] mentions that this method may be used indirectly for clustering, but no results are reported. We believe the method will fragment clusters and find many small (or fragmented) regions due to its stopping criteria and its covering method. The CLTree method is different. Its method is based on partitioning rather than covering. Great care has been taken to ensure that clusters are not unnecessarily fragmented due to partitioning. CLTree can also run with data residing on disk (see [25]).

Another body of existing work is on clustering of categorical data [30, 19, 13]. Since this paper focuses on clustering in a numerical space, we will not discuss these works further.

6.2. Input Parameters

Most existing cluster methods critically depend on input parameters, e.g., the number of clusters [e.g., 26, 33, 1, 2, 6], the size and density of grid cells [e.g., 21, 24, 29, 3, 7], and density thresholds [e.g., 9, 22]. Different parameter settings often result in completely different clustering. CLTree does not need any input parameter in its main clustering process, i.e., cluster tree building. It uses two parameters only in pruning. However, these parameters are quite different in nature from the parameters used in the existing algorithms. They only facilitate the user to explore the space of clusters in the cluster tree to find useful clusters.

In traditional hierarchical clustering, one can also save which clusters are merged and how far apart they are at each clustering step in a tree form. This information can be used by the user in deciding which level of clustering to make use of. However, as we discussed above, distance in a high dimensional space is misleading. Furthermore, traditional hierarchical clustering methods do not give a precise description of each cluster. It is thus hard for the user to interpret the saved information.

7 CONCLUSION

In this paper, we proposed a novel clustering technique, called CLTree, which is based on decision trees in classification research. CLTree performs clustering by partitioning the data space into dense and sparse regions. To make the decision tree algorithm work for clustering, we have devised a technique to introduce *non-existing* points to the data space, and designed a new purity function that looks ahead in determining the best partitioning. CLTree has many advantages over existing clustering methods (see Section 1.1). Extensive experiments have been conducted with the proposed technique. The results show that it is both effective and efficient.

ACHKNOWLEDGEMENT

We would like to thank Charu Aggarwal from IBM T. J Watson Research Center for many helpful discussions. The project is funded by National Science and Technology Board, and National University of Singapore under RP3981678.

REFERENCES

[1] C. Aggarwal, C. Propiuc, J. L. Wolf, P. S. Yu, and J. S. Park. "Fast algorithms for projected clustering." *SIGMOD-99*, 1999.

[2] C. Aggarwal, and P. S. Yu. "Finding generalized projected clusters in high dimensional spaces." *SIGMOD-00*, 2000.

[3] R. Agrawal, J. Gehrke, D. Gunopulos and P. Raghavan. "Automatic subspace clustering for high dimensional data for data mining applications." *SIGMOD-98*, 1998.

[4] P. Arabie and L. J. Hubert. "An overview of combinatorial data analysis." In P. Arabie, L. Hubert, and G.D. Soets, editors, *Clustering and Classification*, pages 5-63, 1996.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan and U. Shaft. "When is nearest neighbor meaningful?" *Proc.7th Int. Conf. on Database Theory (ICDT)*, 1999.

[6] P. Bradley, U. Fayyad and C. Reina. "Scaling clustering algorithms to large databases." *KDD-98*, 1998.

[7] C. H. Cheng, A. W. Fu and Y Zhang. "Entropy-based subspace clustering for mining numerical data." *KDD-99*.

[8] R. Dubes and A. K. Jain, "Clustering techniques: the user's dilemma." *Pattern Recognition*, 8:247-260, 1976.

[9] M. Ester, H.-P. Kriegal, J. Sander and X. Xu. "A density-based algorithm for discovering clusters in large spatial databases with noise." *KDD-96*, 1996.

[10] M. Ester, H.-P. Kriegel and X. Xu. "A database interface for clustering in large spatial data bases." *KDD-95*, 1995.

[11] B. S. Everitt. *Cluster analysis*. Heinemann, London, 1974.

[12] U. Fayyad, C. Reina, and P. S. Bradley, "Initialization of iterative refinement clustering algorithms." *KDD-98*, 1998.

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

[14] J. H. Friedman and N. I. Fisher "Bump hunting in high-dimensional data." To appear: *Statistics and Computation*, 1998.

[15] K. Fukunaga. *Introduction to statistical pattern recognition*. Academic Press, 1990.

[16] V. Ganti, J. Gehrke, and R. Ramakrishnan, "CACTUS-Clustering categorical data using summaries." *KDD-99*.

[17] J. Gehrke, R. Ramakrishnan, V. Ganti. "RainForest - A framework for fast decision tree construction of large datasets." *VLDB-98*, 1998.

[18] J. Gehrke, V. Ganti, R. Ramakrishnan & W-Y. Loh, "BOAT – Optimistic decision tree construction." *SIGMOD-99*, 1999.

[19] S. Guha, R. Rastogi, and K. Shim. "ROCK: a robust clustering algorithm for categorical attributes." *ICDE-99*.

[20] S. Guha, R. Rastogi, and K. Shim. "CURE: an efficient clustering algorithm for large databases." *SIGMOD-98*.

[21] A. Hinneburg and D. A. Keim. "An efficient approach to clustering in large multimedia databases with noise." *KDD-98*, 1998.

[22] A. Hinneburg and D. A. Keim. "An optimal grid-clustering: towards breaking the curse of dimensionality in high-dimensional clustering." *VLDB-99*, 1999.

[23] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice Hall, 1988.

[24] B. Liu, K. Wang, L. Mun and X. Qi. "Using decision tree induction for discovering holes in data." *PRICAI-98*, 1998.

[25] B. Liu, Y. Xia, and P. S. Yu. "CLTree - Clustering through decision tree construction." *Forthcoming*, an earlier version appeared as an IBM Research Report RC 21695, 20/3/2000.

[26] R. Ng and J. Han. "Efficient and effective clustering methods for spatial data mining." *VLDB-94*.

[27] J. R. Quinlan. *C4.5*: *program for machine learning*. Morgan Kaufmann, 1992.

[28] J.C. Shafer, R. Agrawal, & M. Mehta. "SPRINT: A scalable parallel classifier for data mining." *VLDB-96*, 1996.

[29] G. Sheikholeslami, S. Chatterjee and A. Zhang. "WaveCluster: a multi-resolution clustering Approach for very large spatial databases." *VLDB-98*, 1998.

[30] K. Wang, C. Xu, and B Liu. "Clustering transactions using large items." *CIKM-99*, 1999.

[31] W. Wang, J. Yang and R. Muntz. "STING: A statistical information grid approach to spatial data mining." *VLDB-97*, 1997.

[32] X. Xu, M. Ester, H-P. Kriegel and J. Sander. "A non-parametric clustering algorithm for knowledge discovery in large spatial databases." *ICDE-98*, 1998.

[33] T. Zhang, R. Ramakrishnan and M. Linvy. "BIRCH: an efficient data clustering method for very large databases." *SIGMOD-96*, 1996.