Unsupervised Hierarchical Clustering via a Genetic Algorithm

William A. Greene
Computer Science Department
University of New Orleans
New Orleans, LA 70148
bill@cs.uno.edu

Abstract: We present a clustering algorithm which is unsupervised, incremental, and hierarchical. The algorithm is distance-based and creates centroids. Then we combine the power of evolutionary forces with the clustering algorithm, counting on good clusterings to evolve to yet better ones. We apply our approach to standard data sets, and get very good results. Finally, we use bagging to pool the results of different clustering trials, and again get very good results.

1 Introduction

A clustering algorithm should partition a set of elements in such a way that elements in a same subset are similar or close together, and concomitantly, elements in different subsets are comparably dissimilar or far apart. (Recall a *partition* of a set S is a collection of disjoint subsets of S whose union is all of S.) The algorithm is termed *unsupervised* if its input elements are not pre-classified (or pre-labeled) with respect to membership in some set of categories; the input just exists, without further overt hint about its natural subgroups.

Unsupervised learning algorithms face an especially daunting challenge, and one may fairly question if it is possible to expect much success from them in general. As a counter to such qualms, no less an authority than Duda and Hart, in the recent revision (Duda, Hart, & Stork, 2001) of their 1973 classic, tell us (page 517) there are at least five basic reasons why we should be interested in unsupervised learning and clustering methods. They give the reasons (1) to label a large data set (such as speech patterns) can be very costly, (2) to first cluster unlabeled patterns can give a supervised learner better sets for it to label, (3) patterns can slowly change over time and an unsupervised classifier might track the changes, (4) there are certain unsupervised methods which can find features useful for categorization, and (5) in an investigation, early exploratory data analysis can give insights into the nature or structure of the data.

Results from the area of genetic algorithms, such as (Jones & Beltramo, 1991), (Falkenauer, 1995), and (Greene, 2001), have demonstrated there can be noteworthy successes when evolutionary techniques are used to partition sets. Thus, it is natural to wonder if genetic algorithms can serve the cause of unsupervised hierarchical clustering. Our thought is to generate a population of candidate clusterings, apply evolutionary forces to them -- we have in mind survival of the fittest, mutation, and mating with crossover (whatever crossover may amount to, for partitions) -- and hope that better and better clusterings will evolve.

In this paper we introduce a clustering algorithm which is incremental, hierarchical, and unsupervised. It is distance-based, and creates centroids for its subsets. We then couple the power of evolutionary techniques to our clustering algorithm, and get very good results when using standard data sets from the UC-Irvine repository. Finally, we apply bagging, *à la* Breiman, to coalesce the results of multiple trials, and again get very good results.

Our work holds promise for when there exist a set of data records whose natural subgroups are unknown but desired. Two examples that come to mind are (a) customer records from a retailer's database, and (b) descriptions of diseased tissues.

2 Preliminaries

We suppose that the objects of interest to us can be described by listing the values each exhibits along various featural dimensions. For example, an object might be described in terms of the features of size, color, and shape, as a (small, red, triangle). The latter illustrates a *feature vector*, meaning one whose coordinates are values from corresponding features. The feature of color illustrates a nominal feature. A feature is *nominal* if its values form a simple discrete set without any additional structure.

When all feature values are real numbers, a feature vector becomes a point in a Euclidean vector space, where we have familiar notions of distance and geometry. Also statistical considerations can be exploited. Indeed, given some (training) set of feature vectors which are pre-classified as to belonging to one of several categories, there are many results, posited upon Bayes' Theorem, about ways to learn a predictor for the category membership of novel or unseen feature vectors. The reader interested in further explication is directed to (Duda, Hart, & Stork, 2001).

In artificial intelligence work, features do not so often have numerical values. Those in the machine learning community have investigated how to cluster objects which are described in terms of features which are not necessarily numeric. A feature's value-set might be nominal, or linear (order isomorphic to a set of integers), or numeric (i.e., real-valued) or tree-like.

There is considerable variety in the approaches to be found in the literature of machine learning. Fisher (1987),

Martin and Billman (1994), Mirkin (1999) and Meila and Heckerman (2001) use statistical considerations to build clusters. Hanson and Bauer (1989) use an information-theoretic approach to assessing similarity. Michalski and Stepp (1983), Carpineto and Romano (1996), and Mirkin (1999) accept only clusters that can be described in a concept description language which is based on conjunctions of assertions about feature values. Michalski and Stepp (1983), Fisher (1987), Lebowitz (1987), and Carpineto and Romano (1996) build hierarchies of clusterings; other researchers seek a simple (*flat*) partition. Some approaches allow overlapping subsets (i.e., not true partitions in the mathematical sense). Some are batch approaches and others are incremental. Some are intended only for nominal features, others allow various kinds of features.

For the research reported in this paper, all features are nominal, but our approach has a natural extension to other varieties of features. In so far as it is distance-based, our approach harkens back to *cluster analysis* and *numerical taxonomy* (we cite Romesburg (1984) as a reference). Our approach creates centroids for its subsets.

Also we note that our approach is related to *k*-means clustering (we recommend section 10.4.3 of Duda, Hart, and Stork (2001) as a reference for the latter), and later we will contrast our approach with *k*-means clustering.

3 Description of the Clustering Algorithm

Our algorithm is to be described as hierarchical, incremental, and unsupervised. It builds a hierarchy of subsets: a partitioning subset may itself be partitioned, and those subsubsets may themselves be partitioned, etc. Making the clustering hierarchical does complicate matters somewhat. On the other hand, sometimes a hierarchical classification is definitely what is desired, as in the classification of living organisms into kingdom, phylum, class, order, family, genus, species. Also, in the evolutionary approach which we practice in this research, the goodness (or, for evolutionary thinking, the *fitness*) of a subset can include a component that reflects the fitnesses of its substructures.

Our algorithm is incremental: having built a clustering of certain feature vectors, when a new feature vector arrives, it can be incorporated into the existent clustering by a modification of the latter and without the need for an entire re-calculation of the cluster structure. Also, in our genetic work for evolving clusterings, mutation will make feature vectors move in and out of subsets; this granted, an incremental approach is better for our work.

Our algorithm is unsupervised: feature vectors are not pre-classified as to membership in one of some known collection of categories. They just exist, and we seek to identify the ways in which they naturally group themselves together into clumps.

In the remainder of this section we give the details of our clustering algorithm. The algorithm is implemented in the language Java. We define a Java class named `Subset`, and each cluster is an instance of it. Also, each instance consists of a subset of the input feature vectors, and a `Subset` is defined recursively to be either unpartitioned, or partitioned into child `Subset`'s of its own elements.

Clusters are formed based upon distances between feature vectors. In the research reported here, all features are nominal. For the distance between two feature vectors, we use Hamming distance: the distance between two values from the same nominal feature (such as color) is 0 or 1 according as the values are the same or different, then the distance between two feature vectors is the sum of the distances between their corresponding coordinates.

A `Subset` has a *centroid*, which acts as a geometric center, or prototypical element. The centroid is defined feature by feature. In a particular nominal feature coordinate *F*, the centroid's value is obtained by majority vote of the feature vectors in the `Subset`. Those feature vectors exhibit some distribution of the values of feature *F*, and the predominating one is taken as the centroid's value for feature *F*. In the case of ties, one of the tying values is chosen at random. A centroid is essentially a feature vector (in our Java implementation, class `Centroid` is an extension of class `FeatureVector`), so we define the distance from a feature vector to a `Subset` to be its distance to the centroid of that subset. Similarly, the distance between two `Subset`'s is the distance between their centroids.

(Our work has a natural extension to non-nominal features. For instance, if feature *F* is order-isomorphic to the integers 1..N, then the distance between values *x* and *y* would be $k\ |x - y|$ for some constant *k*, and majority vote gets replaced by rounded averaging when computing a centroid's *F*-coordinate. But we explore only nominal features in this research.)

A `Subset` includes a list of the elements (feature vectors) in it. A `Subset` has an *actual radius*, which is the maximum distance from any element in it to the centroid. Also a `Subset` has a *maximum radius* (meaning the "biggest" the subset is allowed to expand to) which is fixed when the subset is created. Our clustering is hierarchical and we relate the maximum radii of parent and child by saying the child's is the parent's times a certain fixed fraction (the *child radius factor*, a system parameter), but note we never let a maximum radius get below a certain universal value (for which we used 4.0). The child radius factor influences how many subsets a parent can have. The child maximum radius acts as a cutoff value for determining when partitioning a `Subset` into sub-`Subset`'s is justified: when adding a new feature vector to an unpartitioned `Subset` makes its actual radius exceed the child maximum radius, then we partition the `Subset` (described below) and thereby give it child `Subset`'s.

The output from our algorithm is a hierarchical clustering; the output is an instance of `Subset` and is the root subset of the hierarchy we form from the input. The maximum radius at the root, fixed at creation time, is the maximum distance possible between two feature vectors. (Usually this is simply the number of nominal features at hand.)

Our algorithm is incremental, so we must answer, how is a new feature vector incorporated into an existent clus-

tering? This is answered by explaining how a new feature vector is added to a Subset. First let us note it turns out a feature vector is added only when its distance to the subset is within the maximum radius of that subset. First the new element is added to the list of the elements in this Subset. Next, the feature values of the new element are incorporated into the structure of votes cast for the feature values of the centroid of this subset. The centroid and the actual radius of the subset are re-calculated. If the subset is not already partitioned but its actual radius has now grown greater than its child maximum radius, then structure is added to this Subset by partitioning its elements into two (sub) Subset's. The latter is accomplished by identifying the two furthest apart elements in the subset, starting singleton Subset's on them, then (recursively) adding randomly chosen remaining elements of the subset to whichever of the two sub-subsets they are closer to (for ties, we flip a coin).

On the other hand, the Subset may already be partitioned, in which case matters are a little more complicated. The newly calculated centroid and actual radius may make the latter fall below the cutoff value which justifies partitioning. In this case, we simply discard all substructure and make this Subset become unpartitioned. Otherwise, the actual radius remains large enough to justify the existent substructure, so we incorporate the new feature vector into the existent substructure. The new element is "close enough" to an existent sub-Subset if its distance to (the centroid of) such is within the maximum radius of that sub-subset. We add the new element to the closest sub-subset to which it is close enough, but if it is not close enough to any sub-subset, then we start a new singleton sub-Subset on the new feature vector. The overview pseudo-code for adding a new individual to a Subset is given next.

```
/* Add a new individual to this subset */
public void addIndividual(Individual indivl){
    this.addToRoster(indivl);
    this.acceptCentroidVotingFrom(indivl);
    this.calculateCentroid();
    this.calculateActualRadius();
    this.calculateFitness();
    if (this.isPartitioned())
        if (this.actualRadius() <=
                this.childMaxRadius())
            //lost need to partition
            this.discardSubSubsetStructure();
        else  //stay partitioned and...
            addIndividualToSomeSubSubset(indivl);
    else //this subset was not partitioned
        if (this.actualRadius() >
                this.childMaxRadius())
            partitionThisSubsetInTwo();
}//end command addIndividual
```

In summary, a new feature vector gets added to a Subset as follows. The new element is added into a simple list of pointers to the members of this Subset, then percolates down through substructures of partitionings into sub-subsets if such exist. The result of multiple additions will of course depend on the order in which new elements are added, but this artifact will be largely overcome under our genetic algorithm, as will be seen.

Here we examine the cost of incorporating a new feature vector into an existent Subset, as a function of n = the number of elements in the Subset. Incorporating the new element's feature values into the centroidal voting structure does not depend on n, and similarly for re-calculating the centroid. Re-calculation of the actual radius takes $n+1$ distance calculations, so has cost $O(n)$. If the Subset is unpartitioned and now merits partitioning, we calculate the distances between all pairs of members, at cost $O(n^2)$, but note we can expect a constant cost when adding each element to one of the two new sub-subsets, whence cost $O(n^2)$ in this case. Moreover, we can assume this case occurs only when n is fairly small. On the other hand, when the Subset is partitioned, we make the simplifying assumption that there are k sub-Subset's at each non-leaf level of the hierarchy. The cost function then satisfies the recurrence relation $f(n) = c_0 + c_1 n + c_2 k + f(n/k)$ (where c's are constants), which when solved shows $f(n)$ is $O(n)$. That is, $f(n)$ is $O(n)$ except for smaller n.

In our approach, mutation will remove a feature vector from a Subset. The algorithmics of removal use reasoning analogous to that for adding a feature vector.

Here let us note the fluidity of the hierarchy that is formed when feature vectors are added or removed. Addition of a feature vector to a Subset can cause a partitioning into sub-Subset's to come into existence, or stay in existence, but also can cause such a partitioning to get erased. Similar behavior holds for removals.

Now that we have completed our description of how basic clustering is done, we next turn to a description of the evolutionary aspects of our work.

4 Description of the Genetic Algorithm

Given some problem, a genetic algorithm seeks to find solutions to it, as follows. Start with an initial population of candidate solutions to the problem, then subject this population to evolutionary forces such as survival of the fittest, mating with crossover, and mutation. The hope is that, over time, better and better candidate solutions will surface in the population. Use the best solution ever found as the working solution. Genetic algorithms were devised by (Holland, 1975) and popularized by (Goldberg, 1989).

Research from the literature, such as (Jones & Beltramo, 1991), (Falkenauer, 1995), and (Greene, 2001), show that genetic algorithms can have notable success when devising set partitions. Our clustering work forms partitions, at various levels. We have combined geneticism with our basic clustering algorithm, in the hope that evolutionary forces might cause good clusterings to produce yet better ones. The genetic operators used in this research derive from those of (Greene, 2001) and in particular we have in mind the greedy crossover operator found there. In

the remainder of this section we describe the genetic components of the current research.

The basic evolutionary progression is generational, in the sense that the population *P(t)* at time *t* is transformed into and replaced by a successor population *P(t+1)* at time *t+1*. There is an initial population *P(0)* which is formed from the input of feature vectors. Each individual of the initial population is formed as follows. First, the input is scrambled into a random order. We start a first `Subset` on the first feature vector, then we add each remaining feature vector to it (in the order of the scrambling), as described in the preceding section on basic clustering. Each initial population member is created from a distinct scrambling, so the initial population is a variety of sensible clusterings. It is the use of a population which overcomes to a large degree the fact that the result of building a `Subset` depends on the order in which elements are added.

For any genetic approach, we must have some measure of an individual's fitness. In our case, each population member is a `Subset`. We define the fitness of a `Subset` recursively. If the subset is partitioned (has acquired the structure of sub-`Subset`'s), then we define fitness to be the sum of (a) the average fitness of its sub-subsets, and (b) the average distance between the (centroids of the) sub-subsets. Note for partitioned `Subset`'s this favors one which consists of fitter sub-subsets which moreover are rather far apart from one another. On the other hand, if the `Subset` is not partitioned, we define its fitness to be sum of (a) the number of elements in it, and (b) the thinness rating of the subset, which we define as its maximum radius divided by its actual radius. Note for unpartitioned `Subset`'s this favors one which is a tight grouping of many elements.

Parents are chosen for mating using a standard *weighted roulette wheel* approach (a form of survival of the fittest): The fitness values exhibited by the current population are first linearly transformed into the real interval [1, 4]. (Such scaling is known to decrease problems of premature convergence to local but not global optima in the fitness landscape). Denoting the (transformed) fitness of population member m_i as f_i, then the *k*-th member m_k is chosen with probability $f_k/(\Sigma_i\ f_i)$.

Now we will describe how mating with crossover produces a child from two parents in the population. Invariably in our work, a parent `Subset` in the population has become partitioned (because it contains all the diverse input feature vectors at hand). Our approach to crossover is greedy, and is driven by the fitnesses of the sub-`Subset`'s (the ones on level one) of the root `Subset` that is a population individual. Denote by *j* the number of sub-subsets on level one of the first parent, and likewise denote by *k* the number on level one of the second parent. Let *n* be the average of *j* and *k*. Collect the *n* fittest distinct level-one subsets which can be found among those of the two parents. These are the initial level-one subsets of the child of crossover. We note that these subsets, as mathematical subsets, may overlap, and may fail to contain every feature vector in the input. Hence, repair is needed if we are to obtain a true partition of all the input. A feature vector from the input might belong to two of the child's initial level-one subsets, but no more than two, since each parent does partition the input. Alternatively, a feature vector from the input might belong to one or none of the child's initial level-one subsets. If a feature vector belongs to two of the initial level-one subsets of the child, we remove it from the less fit one (we flip a coin in the case of a tie). If a feature vector belongs to exactly one child level-one subset, that is as desired and we leave it alone. Finally we round up all the feature vectors which as yet belong to no level-one subset of the child, scramble them, then add them to the child by using the algorithm given earlier for basic clustering. After these steps, the child's level-one subsets do form a partition of the input feature vectors.

Now we describe how mutation of a `Subset` is performed. A single mutation step consists of removing a randomly chosen feature vector from the root `Subset` (which removes it from all levels it appears in), then adding it back in again. Of course, this can make the structure of the `Subset` change, which is the point of mutation. Moreover, we practice mutation which is stochastic and graduated. It is stochastic, in that when a mutation is attempted, it succeeds and results in a single mutation step only with a 50-50 chance. Our mutation is graduated, in that less fit members of the population are subjected to greater numbers of mutation attempts. Specifically, there is some maximum number (we used the number 40) of mutation attempts which the least fit population member is subjected to (note we expect only about 20 of these attempts to succeed and result in single mutation steps). More fit population members are subjected to proportionately fewer mutation attempts, and in our approach we spare the most fit two members from any mutation at all.

Now we are able to describe how we derive the population *P(t+1)* at time *t+1* from the population *P(t)* at time *t*. Each generation of the population will contain the same number of individuals, denote it *K*. First, the fittest two members of *P(t)* automatically survive unchanged into the next generation *P(t+1)*; these are termed the *elite* survivors. (Thus it is guaranteed that the fittest member of *P(t+1)* is at least as fit as the fittest member of *P(t)*.) Next, *K*-2 children are produced by mating with crossover, choosing parents within *P(t)* by weighted roulette wheel. After the *K*-2 children have been collected, they are subjected to graduated stochastic mutation. Then, because it is reasonable to discourage the presence of level-one subsets which are too small, each child which has no level-one subsets that are too small gets its fitness boosted upward modestly (we added a boost of 2; what too small amounts to is problem dependent). This completes the construction of *P(t+1)*.

At this point we will compare our approach to *k*-means clustering, which also is a centroidal approach (confer section 10.4.3 of (Duda, Hart, & Stork, 2001)). Imagine a set *V* of vectors; assume there is a notion of distance between vectors. In *k*-means clustering, the number of clusters is

chosen in advance, namely, k. Choose k initial centroids c_i, say, any k elements of V. Loop on the steps...

(1) partition V into k subsets V_i by associating each vector in V with the c_i it is closest to;

(2) re-calculate centroids: c_i := the distance-implied centroid of V_i;

...until there is no change in the centroids.

Two obvious differences between our approach and standard k-means clustering are that ours does not pre-choose the number of clusters, and also forms hierarchical clusters. K-means clustering includes no genetic component. K-means clustering uses a loop to jostle V into good subsets, as centroids migrate until quiescence. Our approach may do a little bit of jostling of input feature vectors when building the elements of *P(0)* (substructures may come into or go out of existence), but more heavily depends on the jostling that accompanies evolutionary operators.

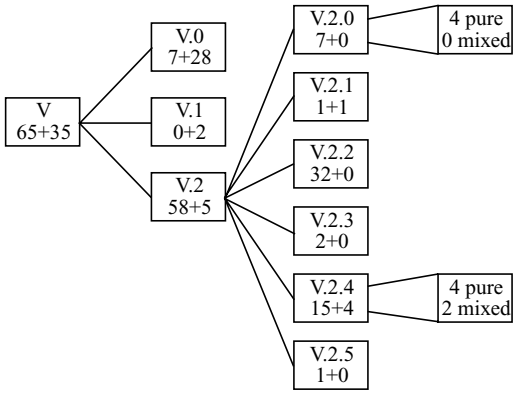

Figure 1: Votes; Dem + Rep
PopSize = 30
Radius at level-0 = 16
Child radius factor = 0.66
Min level-1 subset size = 5

5 Clustering Experiments

We applied our clustering approach to three standard data sets which can be found in the UC-Irvine repository of data sets for machine learning (`www.ics.uci.edu/~mlearn/MLRepository.html`). The data sets we use are, in fact, pre-classified with regard to membership in certain categories. We exclude the known classification as a feature, of course. After clustering, we can then examine how our clusters correlate with the known classification.

One such data set consists of the voting records from 1984 of all 435 US Congressional House members on 16 issues indicative of political philosophy (such as providing aid to handicapped infants, and putting a freeze on physician fees). Each vote has one of 3 values: yea, nay, or "other" (such as did not vote, or voted merely "present" in order to avoid a conflict of interest). We randomly culled out 100 of these 435 records. Our subset turned out to contain 65 Democrats and 35 Republicans, but, to repeat ourselves, we do not use party affiliation as a feature.

Naturally, we expect our clustering algorithm to group conservative Congressmen together, and, moreover, to tend to group Democrats together, etc. Here we report a typical trial run. The trial ran to 25 generations (as did every trial reported in this research). Figure 1 depicts the tree structure of the resulting clustering. In Figure 1, the annotation "7+28" for level-one subset V.0 means this subset consists of 7 Democrats plus 28 Republicans. The inflated box to the right of that for V.2.4 signifies that this subset was further partitioned, in this case into 4 subsets comprised purely of one party, plus 2 subsets with mixed party affiliation.

Analysis of the tree structure shows the clustering is very sensible. There are 3 level-one subsets: V.0, which is predominantly Republican, V.2, which is predominantly Democratic, and V.1, which consists of two outlying Republicans whose votes were "other" on almost all 16 votes. We can use a subset's centroid, which is prototypical for the subset, as an indication of voting tendency. So examining the centroids (not presented here) of V.0 and V.2, we find they are diametrically opposite, yea versus nay, on all votes except for agreement on votes 6 and 16. Note that subset V.0 is not further partitioned; it is a tight grouping of conservative voters (though V.0 is allowed, it turns out, a maximum radius of 10.56, its actual radius is 6). On the other hand, subset V.2 is further partitioned into six subsets. The largest two of these six are V.2.2 (not further partitioned) and V.2.4 (partitioned one level deeper into 6 subsets). The centroids (not shown here) for V.2.2 and V.2.4 reveal that these groups tended to vote oppositely on votes 1, 2, 6, 10, 11, and 14, and vote the same on the remaining 10 votes. Considering the issues at vote, one might say the members of V.2.2 tend to be liberal on social issues but conservative on monetary ones.

We ran our clustering algorithm using two other data sets from the UC-Irvine repository. The second data set used is mushroom data. This very large data set consists of descriptions, in terms of 22 nominal features, of 8124 hypothetical mushroom samples. The samples correspond to 23 species of gilled mushrooms in the *Agaricus* and *Lepiota* families, and are derived from the 1981 *Audubon Society Field Guide to North American Mushrooms*. Each feature vector is pre-classified as to being edible or poisonous (again, not used as a feature in our algorithm). Some feature vectors have missing values. Among those with complete featural descriptions, we culled out 200 at random; of these 128 are edible and 72 are poisonous. Of course, we will be interested in how our algorithm separates edible mushrooms from poisonous ones.

Figure 2 shows the tree structure produced on a typical run. On level one, we see two subsets M.1 and M.2 which consist of exclusively poisonous mushrooms, and moreover M.2 is not further partitioned. Subset M.0 is predominantly edible mushrooms, and gets partitioned into 9 subsets, of varying sizes, each of which is purely edible or purely poisonous examples. Subset M.3 consists of 84 edi-

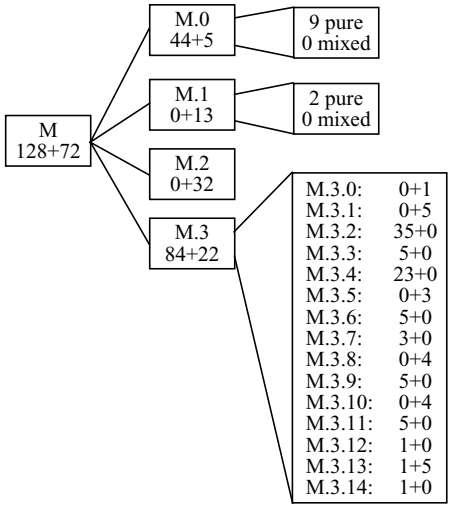

Figure 2: Mushrooms; edible + poisonous
PopSize = 20
Radius at level-0 = 22
Child radius factor = 0.5
Min level-1 subset size = 10

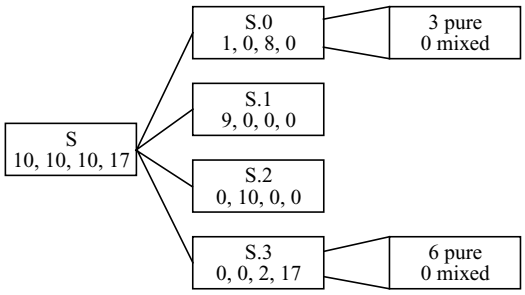

Figure 3: Soy; diseases 1, 2, 3, 4
PopSize = 25
Radius at level-0 = 21
Child radius factor = 0.5
Min level-1 subset size = 4

ble and 22 poisonous mushrooms; this group must be the one with characteristics most confusing to mushroom hunters. This subset is further partitioned in 15 subsets, each of which, with but the one exception of M.3.13, consists of purely edible or purely poisonous mushrooms. We consider this to be an excellent result! The level-one subsets in the hierarchy have not separated the edible mushrooms from the poisonous ones, but the level-two subsets have (with one exception).

The third data set used from the UC-Irvine repository concerns diseased soybean plants. The plants are described in terms of 35 nominal features. The full data set has instances from 19 diseases. We use the repository's small soybean data set; it concerns just four diseases, and has, respectively, 10, 10, 10, and 17 instances of these four diseases. Each of these 47 feature vectors comes pre-classified as to disease (not used as a feature in our algorithm). (These 47 feature vectors have identical values for 14 of the 35 features, hence, can differ at most on the remaining 21 features. Consequently, when we create a root Subset, we let its maximum radius be 21.) Trial runs on this data set had an undesirable property. Although the first two diseases' examples tended to wind up in distinct level-one subsets, the last two diseases were routinely lumped together. They became separated on lower levels, but this gave us the following thought.

We might reward clusterings that had a desired number of level-one subsets. This is not so unreasonable. A data analyst may have some insight into how many principal subsets ought to appear in a clustering. So, we modified the algorithm as follows. When creating generation *P(t+1)* from *P(t)*, a child can get two boosts to its fitness. Not only does a child get a boost for having no level-one subsets which are too small, but also gets a boost (we added 2) for having a prescribed number (here, four) of level-one subsets. Results of a typical run under this alternative regime are shown in Figure 3. There are now four level-one subsets. Subset S.0 is mixed, with one instance of disease-1 and eight instances of disease-3, but these get separated on the next level. Subset S.1 consists only of disease-1 instances (but not quite all 10), and S.2 consists of exactly the 10 instances of disease-2. Finally we see S.3 is mixed, with 2 instances of disease-3 and all 17 instances of disease-4, and these get separated on the next level. Here again our approach is quite good at finding the natural subgroups of the input.

6 Bagging Experiments

The experimental results reported in the previous section were typical for these data sets. For instance, the general clumpiness shown in Figure 2 for the mushroom data is typical of numerous runs. Nonetheless, on different runs, somewhat different partitionings are produced. The same can be said for the voting data and the soybean disease data. We wondered if we might obtain an answer in which we could have greater confidence if we “bagged” the results of multiple trials. Breiman (1996) has focused attention on the idea of bagging. The notion is that, given several predictors, let them pool their predictions, such as by majority vote.

Our algorithm produces a hierarchy of partitions of its input, and in particular, the level-one subsets which result may be considered as the principal natural subsets of the input. We may take the groupings at level one as a judgment about which elements should be grouped with which other elements, at the highest level of analysis. So let us focus on the level-one subsets. If we make *N* trial runs on the same input, we get *N* judgments about which elements should be grouped with which other elements.

How can we combine these judgments? We know of no obvious way to do so that is mathematically airtight. Consider the following. If element *x* is paired with element *y* on 66% of the *N* trials, we are inclined to say they should be grouped together. If additionally *y* is paired with *z* on 66% of the *N* trials, then likewise we are inclined to group *y* and *z* together. But now note that it is possible that *x* and *z* are paired on only 32% of the *N* trials. Why should *x* and *z* be grouped together?

Nonetheless we will follow this heuristic approach. Let *M* be the number of feature vectors in the input. Run *N* trials. Afterwards, for each *x* in input, make *M* counts, one for each input *y*, where the *y*-th count is the number of the *N* trials on which *x* and *y* are in the same level-one subset. (The *y*-th count will be in the range 0 to *N*.) Now manufacture a partition (which is flat, i.e., no subpartitioning) of the input using the following simple algorithm:

```
for each input numbered x = 1 to M do
    if the x-th input has not been grouped yet, then {
        start a new group as the singleton x;
        for each input numbered y = x+1 to M do
            if (y has not been grouped yet) &&
                (y has been paired with x on more
                        than half the N trials), then {
                insert y into x's group;
                mark y as grouped;
            }//end: if (y has not been ...
    }//end: if the x-th input ...
```

This approach is only heuristic, and moreover it loses the hierarchical structure since it builds only a flat partition of the input. Nonetheless, it gives satisfying results for our data sets. Retaining the parameter settings from the earlier experiments, for each of the three data sets we bagged the level-one results after N = 15 trials. Figures 4, 5, and 6 show the results for, respectively, the voting, mushroom, and soybean disease data sets.

The flat partition seen in Figure 4 for the voting data is very similar to the level-one groupings in Figure 1. The mushroom results show greater change under bagging. Examination of the actual feature vectors in the sets (not shown here) reveals that, Figure 2's mixed subsets M.0 and M.3 have exchanged small numbers of elements to become sets MB.0 and MB.1 in Figure 5. On the other hand, the two purely poisonous sets M.1 and M.2 in Figure 2 do not so closely resemble the purely poisonous subsets MB.2-5 in Figure 5. Apparently the grouping of poisonous mushrooms in Figure 2 was a bit atypical.

The grouping seen in Figure 6 for the soybean data has ironed out the mild eccentricities seen in Figure 3. Interestingly, bagging proves one feature vector for disease-1 to be a true outlier, in Figure 6.

The bagging results are quite satisfying. One trial of our genetic hierarchical clustering algorithm gives a sensible analysis of the data into natural groupings. Bagging on multiple trials improves the quality of the level-one groupings. Bagging does lose sub-partitioning. Of course, we could take a level-one subset as produced by bagging, then run the genetic hierarchical clustering algorithm on it, followed by bagging there, etc., but we have not pursued such an extension in this research.

7 Comparison to Other Work

Our distance-based, centroidal approach to clustering is doing commendably well. In the present research, it is used for clustering feature vectors of nominal features, though it has a natural extension to linear and numeric features, certainly. Although also k-means clustering is distance-based and uses centroids, in general it has not been used to create hierarchical clusters and moreover it is typically applied to exclusively numeric features. Layering evolution and bagging atop the clustering is certainly novel, to our knowledge.

Other researchers have used the same or similar data in their various clustering approaches. It is generally hard to compare our results to those of others. Sometimes this is because their approaches and goals are too different. Sometimes it is because the descriptions of results in the literature do not have the kinds of details that would permit a fine comparison. Nonetheless, the results we have obtained appear to be at least as good as the ones in the literature.

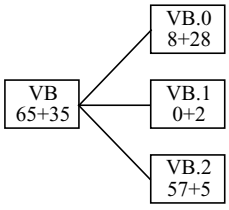

Figure 4: Votes Bagged
Dem + Rep

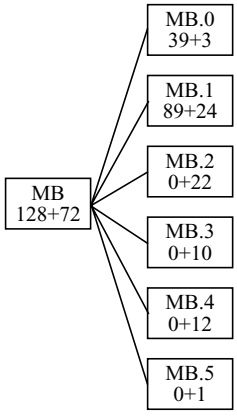

Figure 5: Mushrooms Bagged
Edible + Poisonous

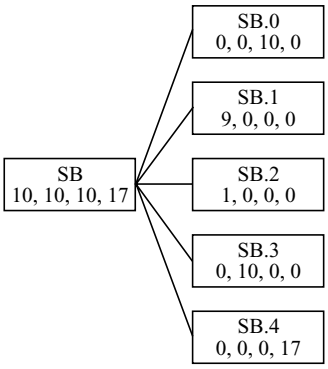

Figure 6: Soy Diseases Bagged
Diseases 1, 2, 3, 4

8 Conclusion

We have presented an incremental unsupervised hierarchical clustering algorithm, which is distance-based and relies on centroids. To our basic clustering approach we have wedded the benefits of evolution. Running on standard data sets, our genetic clustering produced quite good results, ones apparently at least as good as those in the literature. Finally, we have added the benefits of bagging; the results so produced are quite good and ones in which we can have yet more confidence.

This research considered only nominal features. Our work extends in a natural way to other varieties of features. The extension centers on the notions of distance and "value voting" which would be appropriate to such features. Explorations of such features will be a topic of future research.

References

Breiman, L. (1996). Bagging predictors. *Machine Learning* 24, pp. 123-140.

Carpineto, C., & Romano, G. (1996). A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning* 24, pp. 95-122.

Duda, R. O., Hart, P. E., & Stork, D. G. (2001). *Pattern Classification* (2-nd ed.). New York, NY: John Wiley & Sons.

Falkenauer, E. (1995). Solving equal piles with the Grouping Genetic Algorithm. In Eshelman, L. J. (Ed.), *Proceedings of the Sixth International Conference on Genetic Algorithms* (pp 492-497). San Francisco, CA: Morgan Kaufmann Publishing.

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning* 2, pp. 139-172.

Fisher, D. H. (1989). Noise-tolerant conceptual clustering. In Sridharan, N. S. (Ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (1989). San Mateo, CA: Morgan Kaufmann Publishing.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley Publishing.

Greene, W. A. (2001). Genetic algorithms for partitioning sets. *International Journal on Artificial Intelligence Tools* 10, pp. 225-241.

Hanson, S. J., & Bauer, M. (1989). Conceptual clustering, categorization, and polymorphy. *Machine Learning* 3, pp. 343-372.

Hadzikadic, M. & Yun, D. Y. Y. (1989). Concept formation by incremental conceptual clustering. In Sridharan, N. S. (Ed.), *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann Publishing.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Jones, D. R., & Beltramo, M. A. (1991). Solving partitioning problems with genetic algorithms. In Belew, K. R. & Booker, L. B. (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp 442-449. San Francisco, CA: Morgan Kaufmann Publishing.

Langley, P. (1987). Machine learning and concept formation. *Machine Learning* 2, pp. 99-102.

Lebowitz, M. (1987). Experiments with incremental concept formation: UNIMEM. *Machine Learning* 2, pp. 103-138.

Martin, J. D., & Billman, D. O. (1994). Acquiring and combining overlapping concepts. *Machine Learning* 16, pp. 121-155.

Meila, M., & Heckerman, D. (2001). An experimental comparison of model-based clustering methods. *Machine Learning* 42, pp. 9-29.

Pitt, L., & Reinke, R. (1988). Criteria for polynomial-time (conceptual) clustering. *Machine Learning* 2, pp. 371-396.

Michalski, R. S., & Stepp, R. E. (1983). Learning from observation: conceptual clustering. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine Learning: an Artificial Intelligence Approach*. Los Altos, CA: Morgan Kaufmann Publishing.

Mirkin, B. (1999). Concept learning and feature selection based on square-error clustering. *Machine Learning* 35, pp. 25-39.

Romesburg, H. (1984). *Cluster Analysis for Researchers*. Belmont, CA: Lifetime Learning.

Stepp, R. E., & Michalski, R. S. (1986). Conceptual clustering: inventing goal-directed classifications of structure objects. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine Learning: an Artificial Intelligence Approach, Vol. 2*. Los Altos, CA: Morgan Kaufmann Publishing.