Chapter #

TITLE FRACTAL MINING

Subtitle Self Similarity-based Clustering and its Applications

AUTHOR Dr. Daniel Barbara, Dr .Ping Chen

Affiliation George Mason Universit, University of Houston-Downtown

Abstract: Self-similarity is the property of being invariant with respect to the scale used to look at the data set. While fractals are self-similar at every scale used to look at them, many data sets exhibit self-similarity over a range of scales. Self-similarity can be measured using the fractal dimension. Fractal dimension is an important charactaristics for many complex systems and can serve as a powerful representation technique. In this chapter, we present a new clustering algorithm, based on self-similarity properties of the data sets, and also its applications to other fields in data mining, such as projected clustering and trend analysis. Clustering is a widely used knowledge discovery technique. It helps uncovering structures in data that were not previously known. The clustering of large data sets has received a lot of attention in recent years, however, clustering is a still a challenging task since many published algorithms fail to do well in scaling with the size of the data set and the number of dimensions that describe the points, or in finding arbitrary shapes of clusters, or dealing effectively with the presence of noise. The new algorithm which we call Fractal Clustering (FC) places points incrementally in the cluster for which the change in the fractal dimension after adding the point is the least. This is a very natural way of clustering points, since points in the same clusterhave a great degree of self-similarity among them (and much less self-similarity with respect to points in other clusters). FC requires one scan of the data, is suspendable at will, providing the best answer possible at that point, and is incremental. We show via experiments that FC effectively deals with large data sets, high-dimensionality and noise and is capable of recognizing clusters of arbitrary shape.

Key words: self-similarity, clustering, projected clustering, trend analysis

1. INTRODUCTION

Clustering is one of the most widely used techniques in data mining. It is used to reveal structure in data that can be extremely useful to the analyst.

The problem of clustering is to partition a data set consisting of *n* points embedded in a d-dimensional space into k sets or clusters, in such a way that the data points within a cluster are more similar among them than to data points in other clusters.

A precise definition of clusters does not exist. Rather, a set of functional definitions has been adopted. A cluster has been defined [3] as a set of entities which are alike (and different from entities in other clusters), an aggregation of points such that the distance between any point in the cluster is less than the distance to points in other clusters, and as a connected region with a relatively high density of points. Our method adopts the first definition and uses a fractal property to define similarity between points.

The area of clustering has received an enormous attention as of late in the database community. The latest techniques try to address pitfalls in the traditional clustering algorithms (for a good coverage of traditional algorithms see [20]). These pitfalls range from the fact that traditional algorithms favor clusters with spherical shapes (as in the case of the clustering techniques that use centroid-based approaches), are very sensitive to outliers (as in the case of all-points approach to clustering, where all the points within a cluster are used as representative of the cluster), or are not scalable to large data sets (as is the case with all traditional approaches).

New approaches need to satisfy the data mining desiderata[6]:

- Require at most one scan of the data.
- Have on-line behavior: provide the best answer possible at any given time and be suspendable at will.
- Be incremental by incorporating additional data efficiently.

In this chapter we propose a clustering algorithm that follows these desiderata[4], while providing a very natural way of defining clusters that is not restricted to spherical shapes (or any other type of shape). This algorithm is based on self-similarity (namely, a property exhibited by self-similar data sets, i.e., the fractal dimension) and clusters points in such a way that data points in the same cluster are more self-affine among themselves than to points in other clusters.

This chapter is organized as follows. Section 2 offers a brief introduction to the fractal concepts we need to explain the algorithm. Section 3 describes our clustering technique and experimental results. Section 4 discusses its application on projected clustering, and section 5 shows its application on trend analysis. Finally, Section 6 offers conclusions and future work.

2. FRACTAL DIMENSION

Nature is filled with examples of phenomena that exhibit seemingly chaotic behavior, such as air turbulence, forest fires, etc. However, in this behavior it is almost always possible to find *self-similarity*, i.e. an invariance with respect to the scale used. The structures that exhibit self-similarity over every scale are known as *fractals* [24]. On the other hand, many data sets, although not fractal, exhibit self-similarity over a range of scales.

Fractals have been used in numerous disciplines (for a good coverage of the topic of fractals and their applications see [29]). In the database area, fractals have been successfully used to analyze R-trees [13], Quadtrees [14], model distributions of data [15] and selectivity estimation [5].

Self-similarity can be measured using the *fractal dimension*. Loosely speaking, the fractal dimension measures the number of dimensions "filled" by the object represented by the data set. In truth, there exists an infinite family of fractal dimensions. By embedding the data set in an n-dimensional grid which cells have sides of size *r*, we can count the frequency with which data points fall into the *i*-th cell, p_i, and compute D_q, the generalized fractal dimension[17][18], as shown in the equation below.

$$D_q = \begin{cases} \frac{\partial \log \sum_i p_i \log p_i}{\partial \log r} & \text{for } q = 1 \\ \frac{1}{q-1} \frac{\partial \log \sum_i p_i^q}{\partial \log r} & \text{otherwise} \end{cases} \qquad \text{2-1}$$

Among the dimensions described by the above equation, the *Hausdorff Fractal Dimension* (q=0), the *Information Dimension* (q→1), and the *Correlation Dimension* (q=2) are widely used. The Information and Correlation dimensions are particularly useful for data mining, since the numerator of D_1 is Shannon's entropy, and D_2 measures the probability that two points chosen at random will be within a certain distance of each other. Changes in the Information dimension mean changes in the entropy and therefore point to changes in trends. Equally, changes in the Correlation dimension mean changes in the distribution of points in the data set.

The traditional way to compute fractal dimensions is by means of the box-counting plot. For a set of N points, each of D dimensions, one divides the space in grid cells of size r (hypercubes of dimension D). If N(r) is the number of cells occupied by points in the data set, the plot of N(r) versus r in log-log scales is called the *box-counting plot*. The negative value of the slope of that plot corresponds to the Hausdorff fractal dimension D_0. Similar procedures are followed to compute other dimensions, as described in [21].

To clarify the concept of box-counting, let us consider the famous example of George Cantor's dust, constructed in the following manner. Starting with the closed unit interval [0, 1] (a straight-line segment of length

1), we erase the open middle third interval and repeat the process on the remaining two segments, recursively.

Figure 1 illustrates the procedure. The “dust” has a length measure of zero and yet contains an uncountable number of points. The Hausdorff dimension can be computed the following way:

It is easy to see that for the set obtained after n iterations, we are left with $N=2^n$ pieces, each of length $r = (1/3)^n$. So, using a one-dimensional box size with $r = (1/3)^n$, we find 2^n of the boxes populated with points. If, instead, we use a box size twice as big, i.e., $r=2(1/3)^n$, we get 2^{n-1} populated boxes and so on. The log-log plot of box population vs. r renders a line with slope D_0=-log2 / log3 = -0.63....

The value 0.63 is precisely the fractal dimension of the Cantor's dust.

Figure 1 Construction of Cantor dust. The final set has fractal (Hausdorff) dimension 0.63.

In what follows of this section we present a motivating example that illustrates how the fractal dimension can be a powerful way for driving a clustering algorithm.

Figure 2 shows the effect of superimposing two different Cantor dust sets. After erasing the open middle interval which results of dividing the original line in three intervals, the leftmost interval gets divided in 9 intervals, and only the alternative ones survive (5 in total). The rightmost interval gets divided in three, as before, erasing the open middle interval. The result is that if one considers grid cells of size $1/(3\times9^n)$ at the *n*-th iteration, the number of occupied cells turns out to be 5^n+6^n.

The slope of the log-log plot for this set is $D_0'=\lim_{n\to\infty}\log(5^n+6^n)/\log(3\times9^n)$. It is easy to show that $D_0'>D_0^r$, where $D_0^r=\log2/\log3$ is the fractal dimension of the rightmost part of the data set (the Cantor dust of

Figure 1).

Therefore, one could say that the inclusion of the leftmost part of the data set produces a change in the fractal dimension and this subset is therefore “anomalous” with respect to the rightmost subset (or vice-versa). From the clustering point of view, for a human being it is easy to recognize the two Cantor sets as two different clusters. And, in fact, an algorithm that exploits the fractal dimension (as the one presented in this paper) will indeed separate these two sets as different clusters. Any point in the right Cantor set would change the fractal dimension of the left Cantor set if included in the left

cluster (and vice versa). This fact is exploited by our algorithm (as we shall explain later) to place the points accordingly.

Figure 2 A “hybrid” Cantor dust set. The final set has fractal (Hausdorff) dimension larger than that of the rightmost set (which is the Cantor dust set of
Figure 1).

To further motivate the algorithm, let us consider two of the clusters in

Figure 5 the right-top ring and the left-bottom (square-like) ring. Figure 3 shows two log-log plots of number of occupied boxes against grid size. The first is obtained by using the points of left-bottom ring (except one point). The slope of the plot (in its linear region) is equal to 1.57981, which is the fractal dimension of this object. The second plot, obtained by adding to the data set of points on the left-bottom ring the point (93.285928, 71.373638) -- which naturally corresponds to this cluster-- almost coincides with the first plot, with a slope (in its linear part) of 1.57919.

Figure 4 on the other hand, shows one plot obtained by the data set of points in the right-top ring, and another one obtained by adding to that data set the point (93.285928, 71.373638). The first plot exhibits a slope in its linear portion of 1.08081 (the fractal dimension of the data set of points in the right-top ring); the second plot has a slope of 1.18069 (the fractal dimension after adding the above-mentioned point). While the change in the fractal dimension brought about the point (93.285928, 71.373638) in the bottom-left cluster is 0.00062, the change in the right-top ring data set is 0.09988, more than 3 orders of magnitude bigger than first change. Our algorithm would proceed to place point (93.285928, 71.373638) in left-bottom ring, based on these changes.

Figure 3 and

Figure 4 also illustrate another important point. The "ring" used for the box counting algorithm is not a pure mathematical fractal set, as the Cantor Dust (

Figure 1), or the Sierpinski Triangle [24] are. Yet, this data set exhibits a fractal dimension (or more precisely a linear behavior in the log-log box counting plot) through a (relatively) large range of grid sizes. This fact serves to illustrate the point that our algorithm does not depend on the clusters being "pure" fractals, but rather to have a measurable dimension (i.e., their box count plot has to exhibit linearity over a range of grid sizes). Since we base our definition of cluster in the self-similarity of points within the cluster, this is an easy constraint to meet.

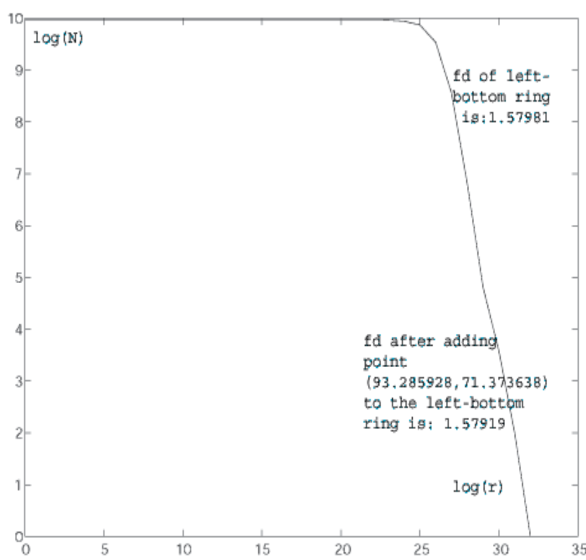


Figure 3 The box-counting plots of the bottom-left ring data set of

Figure 5, before and after the point (93.285928, 71.373638) has been added to the data set. The difference in the slopes of the linear region of the plots is the "fractal impact" (0.00062). (The two plots are so similar that they lie almost on top of each other.)

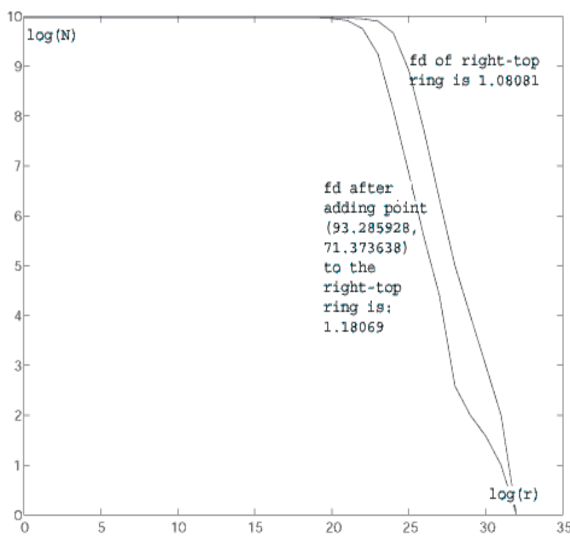


Figure 4 The box-counting plots of the top-right ring data set of Figure 5, before and after the point (93.285928, 71.373638) has been added to the data set. The difference in the slopes of the linear region of the plots is the "fractal impact" (0.09988), much bigger than the corresponding impact shown in Figure 3.

3. CLUSTERING USING THE FRACTAL DIMENSION

Incremental clustering using the fractal dimension, abbreviated as Fractal Clustering, or FC, is a form of grid-based clustering (where the space is divided in cells by a grid; other techniques that use grid-based clustering are STING [32], WaveCluster [31] and Hierarchical Grid Clustering [28]).

The main idea behind FC is to group points in a cluster in such a way that none of the points in the cluster changes the cluster's fractal dimension radically. FC also combines connectness, closeness and data points' position information to pursue high clustering quality.

Our algorithm takes a first step of initializing a set of clusters, and then incrementally adds points to that set. In what follows, we describe the initialization and incremental steps.

3.1 FC initialization step

In clustering algorithms the quality of initial clusters is extremely important, and has direct effect on the final clustering quality. Obviously, before we can apply the main concept of our technique, i.e., adding points incrementally to existing clusters, based on how they affect the clusters' fractal dimension, some initial clusters are needed. In other words, we need to “bootstrap” our algorithm via an initialization procedure that finds a set of clusters, each with sufficient points so its fractal dimension can be computed. We present in this section two choices for the initialization algorithm. The first choice uses fractal concepts to cluster a sample of points taken from the data set. The second uses a more traditional distance-based procedure. Obviously, there are many choices for the initialization step, we present only the results of these two here, but we are considering and experimenting with other options. In any case, if the wrong decisions are made at this step, we will be able to correct them later by reshaping the clusters dynamically.

3.1.1 Initialization algorithm 1

In this first algorithm, the process of initialization is made easy by the fact that we are able to convert a problem of clustering a set of multidimensional data points (which is a subset of the original data set) into a much simpler problem of clustering 1-dimensional points. The problem is further simplified by the fact that the set of data points that we use for the initialization step fits in memory.

The pseudo-code of the initialization step is shown below. Notice that lines 3 and 4 of the code map the points of the initial set into one-dimensional values, by computing the effect that each point has in the fractal dimension of the rest of the set (we could have computed the difference between the fractal dimension of S and that of S minus a point, but the result would have been the same). Line 5 of the code deserves further explanation: in order to cluster the set of Fd_i values, we can use any known algorithm. For instance, we could feed the fractal dimension values Fd_i and a value k to a K-means implementation [16] [30]. Alternatively, we can let a hierarchical clustering algorithm (e.g., CURE [19]) cluster the sequence of Fd_i values. Although, in principle, any of the dimensions in the family described by 2-1 can be used in line 4 of the initialization step, we have found that the best results are achieved by using D_2, i.e., the correlation dimension.

1 Given an initial set S of points $p_1,\ldots, p_M$ that fit in main memory (obtained by sampling the data set).
2 For each i=1 ,…, M

2.1 Define group $G_i = S - p_i$
2.2 Calculate the fractal dimension of the set G_i, Fd_i
3 Cluster the set of Fd_i values (The resulting clusters are the initial clusters.)

3.1.2 Initialization Algorithm 2

In this second choice, we use a more classic distance-based approach. We try to cluster the initial sample of points by taking points at random and finding recursively points that are close to them. When no more close points can be found, a new cluster is initialized, choosing another point at random out of the set of points that have not been clustered yet.

Given a sample of points (which fits in main memory) and a distance threshold κ (Line 1), the algorithm proceeds to build clusters by picking a random yet unclustered point and recursively finding the nearest neighbor in such a way that the distance between the point and the neighbor is less than κ. The neighbor is then included in the cluster, and the search for nearest neighbors continues in depth-first fashion, until no more points can be added to clusters. Notice that we do not try to restrict the clusters by constraining it, as it is customarily done, to have a diameter less than or equal to a certain threshold, since we want to be able to find arbitrarily shaped clusters. The threshold, κ is the product of a predefined, static parameter κ_0, and $\hat{d}$, the average distance between pairs of points already in the cluster. Thus, κ is a dynamic threshold that is updated while points are included in the cluster. Intuitively, we are trying to restrict the membership to an existing cluster to points whose minimum distance to a member of the cluster is similar to the average distance between points already in the cluster.

1.Given an initial set S of points $\{p_1, \cdots, p_M\}$ that fit in main memory (obtained by sampling the data set), and a distance threshold κ. (Initially $\kappa = \kappa_0$.)
2. Mark all points as unclustered, and make $k = 0$
3. Choose a point P (at random) out of the set of unclustered points
    4. Mark P as belonging to cluster C_k
    5. Starting at P and in a recursive, depth-first fashion,
        call $P' = NEAR(P, \kappa)$
    6. If P' is not NULL
        7. Mark P' as belonging to cluster C_k
    8. else backtrack to the previous point in the search.
    9. Update $\hat{d}$, the average distance between pairs of points in C_k
    10. Make $\kappa = \kappa_0 \times \hat{d}$
11. If there are still unclustered points, make $k = k + 1$ and go to 3.

$NEAR(P, \kappa)$
Find the nearest neighbor of P such that $dist(P', P) \leq \kappa$.
If no such P' can be found return NULL
Otherwise return P'.

3.2 Incremental step

After we get the initial clusters, we can proceed to cluster the rest of the data set. Each cluster found by the initialization step is represented by a set of boxes (cells in a grid). Each box in the set records its population of points. Let k be the number of clusters found in the initialization step, and $C = \{C_1, C_2, \ldots, C_k\}$ where C_i is the set of boxes that represent cluster i. Let $F_d(C_i)$ be the fractal dimension of cluster i.

The incremental step brings a new set of points to main memory and proceeds to take each point and add it to each cluster, computing its new fractal dimension. The pseudo-code of this step is shown in the algorithm below. Line 5 computes the fractal dimension for each modified cluster (adding the point to it). Line 6 finds the proper cluster to place the point (the one for which the change in fractal dimension is minimal). We call the value $|F_d'(C_i)-F_d(C_i)|$ the *Fractal Impact* of the point being clustered over cluster i. The quantity $\min_i| F_d'(C_i)-F_d(C_i)|$ is the *Minimum Fractal Impact* of the point. Line 7 is used to discriminate “noise”. If the Minimum Fractal Impact of the point is bigger than a threshold τ, then the point is simply rejected as noise (Line 8). Otherwise, it is included in that cluster. We choose to use the Hausdorff dimension, D_0, for the fractal dimension computation of Line 5 in the incremental step. We chose D_0 since it can be computed faster than the other dimensions and it proves robust enough for the task.

1.Given a batch S of points brought to main memory:
2.For each point $p \in S$:
    3.For $i = 1, \cdots, k$:
        4. Let $C'_i = C_i \bigcup \{p\}$
        5. Compute $F_d(C'_i)$
        6.Find $\hat{i} = min_i(|F_d(C'_i - F_d(C_i)|)$
        7.If $|F_d(C'_{\hat{i}}) - F_d(C_{\hat{i}})| > \tau$
            8.Discard p as noise
        9.else
            10. Place p in cluster $C_{\hat{i}}$

To compute the fractal dimension of the clusters every time a new point is added to them, we keep the cluster information using a series of grid representations, or layers. In each layer, boxes (i.e., grids) have a size that is smaller than in the previous layer. The sizes of the boxes are computed in the following way. For the first layer (largest boxes), we divide the cardinality of each dimension in the data set by 2, for the next layer, we divide the cardinality of each dimension by 4 and so on.

Accordingly, we get 2^D, 2^{2D}, … , 2^{LD} D-dimensional boxes in each layer, where D is the dimensionality of the data set, and L the maximum layer we will store. Then, the information kept is not the actual location of points in the boxes, but rather, the number of points in each box. It is important to remark that the number of boxes in layer L can grow considerably, especially for high-dimensionality data sets. However, we need only to save boxes for which there is any population of points, i.e., empty boxes are not needed. The number of populated boxes at that level is, in practical data sets, considerably smaller (that is precisely why clusters are formed, in the first place). Let us denote by B the number of populated boxes in level L. Notice that, B is likely to remain very stable throughout passes over the incremental step. Every time a point is assigned to a cluster, we register that fact in a table, adding a row that maps the cluster membership to the point identifier (rows of this table are periodically saved to disk, each cluster into a file, freeing the space for new rows). The array of layers is used to drive the computation of the fractal dimension of the cluster, using a box-counting algorithm. In particular, we chose FD3, an implementation of a box counting algorithm based on ideas described in [27].

3.3 Reshaping clusters in mid-flight

It is possible that the number and form of the clusters may change after having processed a set of data points. This may occur because the data used

in the initialization step does not accurately reflect the true distribution of the overall data set or because we are clustering an incoming stream of data, whose distribution changes over time. There are two basic operations that can be performed: splitting a cluster and merging two or more clusters.

A good indication that a cluster may need to be split is given by how much the fractal dimension of the cluster has changed since its inception during the initialization step. (This information is easy to keep and does not occupy much space.) A large change may indicate that the points inside the cluster do not belong together. (Notice that these points were included in that cluster because it was the *best choice* at the time, i.e., it was the cluster for which the points caused the least amount of change on the fractal dimension; but this does not mean this cluster is an ideal choice for the points.) Once the decision of splitting a cluster has been made, the actual procedure is simple. Using the box (finest resolution layer, i.e., the first layer of boxes) population we can run the initialization step.

That will define how many clusters (if more than one) are needed to represent the set of points. Notice that up to that point, there is no need to re-process the actual points that compose the splitting cluster (i.e., no need to bring them to memory). This is true since the initialization step can be run over the box descriptions directly (the box populations represent an approximation of the real set of points, but this approximation is good enough for the purpose). On the other hand, after the new set of clusters has been decided upon, we need to relabel the points and a pass over that portion of the data set is needed (we assume that the points belonging to the splitting cluster can be retrieved from disk without looking at the entire data set: this can be easily accomplish by keeping each cluster in a separate file).

Merging clusters is even simpler. As an indication of the need to merge two clusters, we keep the minimum distance between clusters, defined by the distance between two points P_1 and P_2, such that P_1 belongs to the first cluster and P_2 to the second, and P_1 and P_2 are the closest pair of such points. When this minimum distance is smaller than a threshold, it is time to consider merging the two clusters. The threshold used is the minimum of the $\kappa = \kappa_0 \times đ$ for each of the two clusters. (Recall that đ is the average pairwise distance in the cluster.)

The merging can be done by using box population at the highest level of resolution (smallest box size), for all the clusters that are deemed as too close. To actually decide whether the clusters ought to be merged or not, we perform the initialization algorithm 2, using the center of the populated boxes (at the highest resolution layer) as “points”.

Notice that it is not necessary to bring previously examined points back to memory, since the relabeling can simply be done by equating the labels of the merged clusters at the end. In this sense, merging does not affect the

"one-pass" property of fractal clustering (as splitting does, although only for the points belonging to the splitting cluster).

3.4 Complexity of the algorithm

We assume that the cost of computing the fractal dimension of a set of n points is O(nlogn), as it is the case for the software FD3 that we have chosen for our experiments.

For the first initialization algorithm, the complexity is $O(M^2 \log M)$, where M is the size of the sample of points. This follows from the fact that for each point in the sample, we need to compute the fractal dimension of the rest of the sample set (minus the point), incurring a cost of O(MlogM) per point.

The incremental step is executed O(N) times, where N is the size of the data set. The complexity of the incremental step is O(nlogn) where n is the number of points involved in the computation of the fractal dimension. Now, since we do not use the point information, but rather the box population to drive the computation of the fractal dimension, we can claim that n is O(B) (the number of populated boxes in the highest layer). Now, since B<<N, it follows that the incremental part of FC will take time linear with respect to the size of the data set.

For small data sets, the first initialization algorithm time becomes dominant in FC. However, for large data sets, i.e., when M<<N, the cost of the incremental step dominates, making FC linear in the size of the data set.

The second initialization algorithm exhibits different complexity. A naïve implementation of NEAR will do a pass over the sample data every time it tries to find the nearest neighbor of a given point, given a complexity O(M), and therefore a worst-case total complexity $O(M^2)$. Clever data structures for the nearest neighbor problem have been studied [26] and can be used to improve the running time of the initialization algorithm. In practice, however, we have found that running a naive implementation of NEAR is still much more time-efficient than running the first initialization algorithm. Following the same reasoning we used for the first initialization algorithm, we can conclude that for large data sets, the cost of the incremental step dominates, making FC linear in the size of the data set.

3.5 Confidence Bounds

One question we need to settle is how to determine if we are placing points as outliers correctly. A point is deemed an outlier in the test of Line 7, in incremental algorithm, when the Minimum Fractal Impact of the point exceeds a threshold τ.

To add confidence to the stability of the clusters that are defined by this step, we can use the Chernoff bound [8] and the concept of adaptive sampling [22][23][10][11][12], to find the minimum number of points that must be successfully clustered after the initialization algorithm in order to guarantee with a high probability that our clustering decisions are correct.

Consider the situation immediately after the initial clusters have been found, and we start clustering points using FC. Let us define a random variable X_i, whose value is 1 if the *i*-th point to be clustered by FC has a "Minimum Fractal Impact" which is less than τ, and 0 otherwise. Using Chernoff's inequality one can bound the expectation of the sum of the X_i's, $X = \Sigma_i^n X_i$, which is another random variable whose expected value is *np*, where $p = Pr[X_i = 1]$, and n is the number of points clustered. The bound is shown in 3-1, where ε is a small constant.

$$Pr[\ X/n\ >\ (1+\epsilon)p\ ]\ \leq\ exp(-pn\epsilon^2/3) \quad \text{3-1)}$$

Notice that we really do not know p, but rather have an estimated value of it, namely p^, given by the number of times that X_i is 1 divided by n. (I.e., the number of times we can successfully cluster a point divided by the total number of times we try.)

In order that the estimated value of p, p^ obeys Equation 3-2, which bounds the estimate close to the real value with an arbitrarily large probability (controlled by δ), on e needs to use a sample of n points, with n satisfying the inequality shown in Equation 3-3.

$$Pr[\ \|\hat{p}-p\|\ ]\ >\ 1-\delta \quad \text{3-2)}$$

$$n\ >\ \frac{3}{p\epsilon^2}ln(\frac{2}{\delta}) \quad \text{3-3)}$$

By using adaptive sampling, one can keep bringing points to cluster until obtaining at least a number of successful events (points whose minimum fractal impact is less than τ) equal to s. It can be proven that in adaptive sampling [33], one needs to have s bound by the inequality shown in Equation 3-4, in order for Equation 3-2 to hold.

Moreover, with probability greater than 1-δ/2, the sample size (number of points processed) n, would be bound by the inequality of Equation 3-4. (Notice that the bound of Equation 3-4 and that of Equation 3-5 are very close; The difference is that the bound of Equation 3-5 is achieved without knowing p in advance.)

$$s\ >\ \frac{3(1+\epsilon)}{\epsilon^2}ln(\frac{2}{\delta}) \quad \text{3-4)}$$

$$n\ \leq\ \frac{3(1+\epsilon)}{(1-\epsilon)\epsilon^2 p}ln(\frac{2}{\delta})$$

(3-5)

Therefore, after seeing s positive results, while processing n points where n is bounded by Equation 3-4 one can be confident that the clusters will be stable and the probability of successfully clustering a point is the expected value of the random variable X divided by n (the total number of points that we attempted to cluster).

3.6 Memory management

FC is very space-efficient, by the virtue of requiring memory just to hold the boxes population at any given time during its execution. This fact makes FC scale very well with the size of the set. Notice that if the initialization sample is a good representative of the rest of the data, the initial clusters are going to remain intact (just containing large populations in the boxes). In that case, the memory used during the entire clustering task remains stable. However, there are cases in which we will have demands beyond the available memory. Mainly, there are two cases where this can happen. If the sample is not a good representative (or the data changes with time in an incoming stream) we will be forced to change the number and structure of the clusters, possibly requiring more space. The other case arises when we deal with high dimensional sets, where the number of boxes needed to describe the space may exceed the available memory. For these cases, we have devised a series of memory reduction techniques that aim to achieve reasonable trade-offs between the memory used and the performance of the algorithm, both in terms of running time and quality of uncovered clusters.

3.6.1 Memory reduction technique 1

In this technique, we cache boxes in memory, while keeping others swapped out to the disk, replacing the ones in memory on demand. Our experience shows that the boxes of smallest size consume 75% of all memory. So, we share the cache only amongst the smallest boxes, keeping the other layers always in memory. Of course, we cluster the boxes in pages, and use the pages as a caching unit. This reduction technique affects the running time but not the clustering quality.

3.6.2 Memory reduction technique 2

A way of requiring less memory is to ignore boxes with very few points. While this method can, in principle, affect the quality of clusters, it may actually be a good way to eliminate noise from the data set.

3.7 Projected clustering

Most clustering algorithms do not work efficiently in higher dimensional spaces because of the inherent sparsity of the data. In high dimensional applications, it is likely that for any given pair of points there exist at least a few dimensions on which the points are far apart from one another. So a clustering algorithm is often preceded by feature selection [1][2]. The goal is to find the particular dimensions on which the points in the data are correlated. Pruning away the remaining dimensions reduces the noise in the data. The problem of using traditional feature selection algorithms is that picking certain dimensions in advance can lead to a loss of information. Furthermore, in many real data examples, some points are correlated with respect to a given set of dimensions and others are correlated with respect to different dimensions. Thus it may not always be feasible to prune off too many dimensions without at the same time incurring a substantial loss of information [7].

In this context we now define what we call a projected cluster. Consider a set of data points in some multidimensional space. A projected cluster is a subset C of data points together with a subset D of dimensions such that the points in C are closely clustered in the subspace of dimensions D.

Here is our projected clustering algorithm using fractal dimension.

1. sample the original data set D and get a sample set S
2. run FC initialization algorithm shown above on S and get initial clusters C_i (i=1,…, k, k is the number clusters found)
3. compute C_i's fractal dimension f_i
4. run SVD analysis on C_i, and keep only n_i dimensions of C_i (n_i is decided by f_i), prune off unimportant dimensions, these ni dimensions of C_i is stored in FD_i
5. for all points in D
    a. input a point p
    b. for i=1,…, k
        i. prune p according to FD_i, put p into C_i
        ii. compute C_i's fractal dimension change fdc_i
    c. end
    d. compare fdc_i (i=1,…,k), put p into C_i with the smallest fdc_i
6. end

Projected clustering with the use of fractal dimension has two main advantages compared with FC:

1) Prune off possible noise
2) Running FC with lower dimensions may save memory and make its performance even better.

3.8 Experimental results

In this section we will show the results of using FC to cluster a series of data sets. Each data set aims to test how well FC does in each of the issues we have discussed in the Section 1. For each one of the experiments we have used a value of $\tau = 0.03$ (the threshold used to decide if a point is noise or it really belongs to a cluster).

We performed the experiments in a Sun Ultra2 with 500 MB of RAM, running Solaris 2.5. When using the first initialization algorithm, we have used K-means to cluster the one-dimensional vector of effects. In each of the experiments, the points are distributed equally among the clusters (i.e., each cluster has the same number of points). After we run FC, for each cluster found, we count the number of points that were placed in that cluster and that also belonged there. The accuracy of FC is then measured for each cluster as the percentage of points correctly placed there. (We know, for each data set, the membership of each point; in one of the data sets we spread the space with outliers: in that case, the outliers are considered as belonging to an extra "cluster".)

3.8.1 Scalability

In this subsection we show experimental results of running time and cluster quality using a range of data sets of increasing sizes and a high-dimensional data set. First, we use data sets whose distribution follows the one shown in

Figure 5 for scalability experiments. We use a complex set of clusters in this experiment, in order to show how FC can deal with arbitrarily shaped clusters. (Not only do we have a square-shaped cluster, but also one of the clusters resides inside of another one.) We vary the total number of points in the data set to measure the performance of our clustering algorithm. In every case, we pick a sample of 600 points to run the initialization step. The results are summarized in Figure 6. The first number in the column "time" is the running time when using the first initialization algorithm, while the second number corresponds to the usage of the second initialization algorithm. As it can be seen, the running time (when using the first initialization algorithm)

increases with the size of the data set in an almost linear fashion, going from 53 seconds (48 seconds for initialization and 5 seconds for the incremental step) in the case of the 30,000 points data set to 5,028 seconds for the set with 30 million points. When using the second initialization algorithm, the running time goes from 12 sec. (7 sec for initialization and 5 for the incremental step) for the 30,000 points data set to 4,987 sec. (4,980 sec. for the incremental step and 7 seconds for initialization) in the case of 30 million points. The incremental step grows linearly from 5 seconds in the 30,000 points case to 4,980 seconds in a 30-million-point data set. The figure shows that the memory taken by our algorithm is constant for the entire range of data sets (64 Kbytes). Finally, the figure shows the composition of the clusters found by the algorithm, indicating the number of points and their precedence: whether they actually belong to cluster1, cluster2 or cluster3. (Since we know to which one of the rings of

Figure 5 each point really belongs). Because both of cluster 2 and 3 are rings, they have close fractal dimension, and some points of cluster 3 are misplaced. These figures are a measure of the quality of clusters found by FC. The quality of clusters found by two initialization algorithms is very similar, so we report it only once.

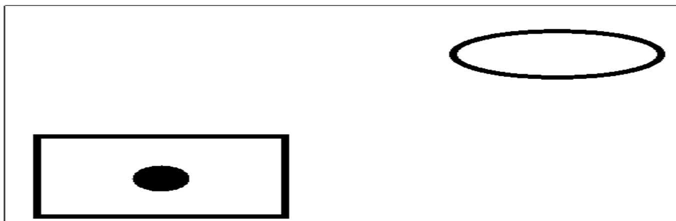


Figure 5 A data set with three clusters

N	time (s)	memory	Clusters found	Assigned to	Coming from Cluster1	cluster2	cluster3	Accuracy (%)
30K	53 12	64KB.	1	10326	9972	0	354	99.72
			2	11751	0	10000	1751	100
			3	7923	28	0	7895	78.95
300K	91 56	64KB.	1	103331	99868	0	3463	99.86
			2	117297	0	100000	17297	100

			3	79372	132	0	79240	79.24
3M	526 485	64KB.	1	1033795	998632	0	35163	99.86
			2	1172895	0	999999	173896	99.99
			3	793310	1368	0	791942	79.19
30M	5028 4987	64KB.	1	10335024	9986110	22	348897	99.86
			2	11722887	0	9999970	1722917	99.99
			3	7942084	13890	8	7928186	79.28

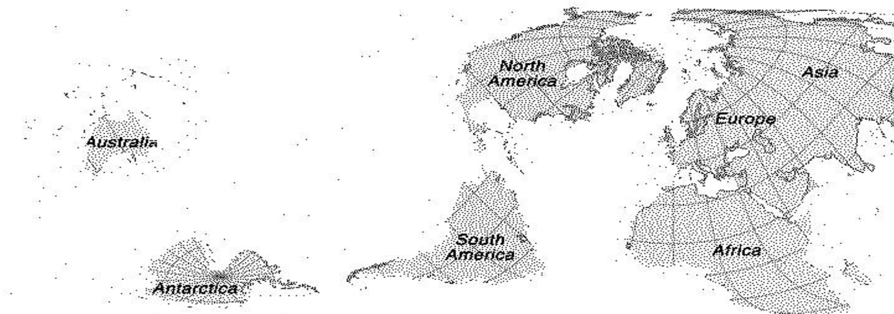
Figure 6 Scalability experiment result

To test FC on a noisy dataset, we added "white noise" to the data set shown in

Figure 5, which has 300,000 points with 5% "noisy" points, i.e., 285,000 points belong into the clusters, while 15,000 points are outliers. FC places 286,242 points into the clusters (i.e., 1242 outliers are incorrectly placed into clusters) and discards 13,758 points (91.72% of the noise). The τ used for this table was 0.003.

3.8.2 An experiment on a real data set

We performed an experiment using our fractal clustering algorithm to cluster points in a real data set. The data set used was a picture of the world map in black and white, where the black pixels represent land and the white pixels water. The data set contains 3,319,530 pixels or points. With the second initialization algorithm the running time was 589 sec (358 seconds for the incremental step and 31 seconds for the initialization). The quality of the clusters is extremely good, totally five clusters were found. Cluster 0 spans the European, Asian and African continents (these continents are very close, so the algorithm did not separate them and we did not run the split technique for the cluster), Cluster 1 corresponds to the North American continent, Cluster 2 corresponds to the South American continent; Cluster 3 corresponds to Australia, and finally Cluster 4 shows Antarctica.



4. TRACKING CLUSTERS

Organizations today accumulate data at an astonishing rate. This fact brings new challenges for data mining. For instance, finding out when patterns change in the data opens the possibility of making better decisions and discovering new interesting facts. The challenge is to design algorithms that can track changes in an incremental way and without making growing demands on memory.

In this section we present a technique to track changes in cluster models. Our technique helps in discovering the points in the data stream in which the cluster structure is changing drastically from the current structure. Finding changes in clusters as new data is collected can prove fruitful in scenarios like the following:

1. Tracking the evolution of the spread of illnesses. As new cases are reported, finding out how clusters evolve can prove crucial in identifying sources responsible for the spread of the illness.
2. Tracking the evolution of workload in an e-commerce server (clustering has already been successfully used to characterize e-commerce workloads [25]), which can help in dynamically fine tune the server to obtain better performance.
3. Tracking meteorological data, such as temperatures registered throughout a region, by observing how clusters of spatial-meteorological points evolve in time.

The idea is to track the number of outliers that the next batch of points produce with respect to the current clusters, and with the help of analytical bounds decide if we are in the presence of data that does not follow the patterns (clusters) found so far. If that is the case, we proceed to re-cluster the points to find the new model.

As we get a new batch of points to be clustered, we can ask ourselves if these points can be adequately clustered using the models we have so far. The key to answer this question is to count the number of outliers in this batch of points. A point is deemed an outlier when the MFI of the point exceeds a threshold τ. We can use the Chernoff bound [8] and the concept of adaptive sampling [22][23][10][11][12], to find the minimum number of points that must be successfully clustered after the initialization algorithm in

order to guarantee with a high probability that our clustering decisions are correct.

After seeing s positive results, while processing n points where n is bounded by Equation 3-4 one can be confident that the clusters will be stable and the probability of successfully clustering a point is the expected value of the random variable X divided by n (the total number of points that we attempted to cluster).

These bounds can be used to drive our tracking algorithm. Essentially, the algorithm takes n new points (where n is given by the lower bound of Equation 3-4) and checks how many of them can be successfully clustered by FC, using the current set of clusters. (Recall that if a point has a MFI bigger than τ, it is deemed an outlier.) If after attempting to cluster the n points, one finds too many outliers (tested in Line 9, by comparing the successful count r, with the computed bound s, given by Equation 3-1), then we call this a turning point and proceed to redefine the clusters. This is done by throwing away all the information of the previous clusters and clustering the n points of the current batch. After each iteration, the value of p is re-estimated as the ratio of successfully clustered points divided by total number of points tried.

0. Initialize the count of successfully clustered points, i.e., r=0
1. Given a batch S of n points, where n is computed as the lower bound of Equation 3-4, using the estimated p from previous round of points
2. For each point in S:
3. Use FC to cluster the point.
4. If the point is not an outlier
5. Increase the count of successfully clustered points, i.e., r=r+1
6. Compute s as the lower bound of Equation 3-1
7. If r<s, flag this batch of points S as a turning point and use S to find the new clusters.
8. Else re-estimate p=r/n

4.1 An experiment on a real data set

We describe in this section the result of two experiments using our tracking algorithm. We performed the experiments in a Sun Ultra2 with 500 Mb. of RAM, running Solaris 2.5.

Our experiment used data from the U.S. Historical Climatology Network [9], which contains (among other types of data) data sets with the average temperature per month, for several years measured in many meteorological stations throughout the United States. We chose the data for the years 1990 to 1994 for the state of Virginia for this experiment (the data comes from 19 stations throughout the state). We organized the data as follows. First we

feed the algorithm with the data of the month of January for all the years 1990-1994, since (we were interested in finding how the average temperature changes throughout the months of the year, during those 5 years. Our clustering algorithm found initially a single cluster for points throughout the region in the month of January. This cluster contained 1,716 data points. Using δ=0.15, and ε=0.1, and with the estimate of p=0.9 (given by the number of initial points that were successfully clustered), we get a window n = 1055, and a value of s, the minimum number of points that need to be clustered successfully, of 855. (Which means that if we find more than 1055-855 = 200 outliers, we will declare the need to re-cluster.)

We proceeded to feed the data corresponding to the next month (February for the years 1990-1994) in chunks of 1055 points, always finding less than 200 outliers per window. With the March data, we found a window with more than 2000 outliers and decided to re-cluster the data points (using only that window of data).

After that, with the data corresponding to April, fed to the algorithm in chunks of n points (p stays roughly the same, so n and s remain stable at 1055 and 255, respectively) we did not find any window with more than 200 outliers. The next window that prompts re-clustering comes within the May data (for which we reclustered). After that, re-clustering became necessary for windows in the months of July, October and December. The τ used throughout the algorithm was 0.001. The total running time was 1 second, and the total number of data points processed was 20,000.

5. CONCLUSIONS

In this chapter we presented a new clustering algorithm based on the usage of the fractal dimension and its applications in other fields of data mining. This algorithm clusters points according to the effect they have on the fractal dimension of the clusters that have been found so far. The algorithm is, by design, incremental and its complexity is O(N), where N is the size of the data set (thus, it requires only one pass over the data, with the exception of cases in which splitting of clusters needs to be done). Our experiments have proven that the algorithm has very desirable properties. It is resistant to noise, capable of finding clusters of arbitrary shape and capable of dealing with points of high dimensionality.

We applied FC to track changes in cluster models for evolving data sets. This problem becomes important as organizations accumulate new data and are interested in analyzing how the patterns in the data change over time. Our algorithm was able to track perfectly changes on the two datasets we experimented with.

We plan to continue the evaluation of our algorithm and apply it to more data analysis fields.

6. REFERENCE

[1]. Charu C. Aggarwal, Cecilia Procopiuc. Fast Algorithms for Projected Clustering In Proceedings of the ACM SIGMOD Conference on Management of Data, 1999.

[2]. Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, Prabhakar Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In Proceedings of the ACM SIGMOD Conference on Management of Data, 1998.

[3]. E. Backer. Computer-Assisted Reasoning in Cluster Analysis. Prentice Hall, 1995.

[4]. Daniel Barbara, and Ping Chen. Using the Fractal Dimension to Cluster Datasets. In Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, MA, 2000.

[5]. A. Belussi and C. Faloutsos. Estimating the Selectivity of Spatial Queries Using the Correlation Fractal Dimension. In Proceedings of the International Conference on Very Large Data Bases, pages 299-310, September 1995.

[6]. P.S. Bradley, U. Fayyad, and C. Reina. Scaling Clustering Algorithms to Large Databases (Extended Abstract). In Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, June 1998.

[7]. Kaushik Chakrabarti, Sharad Mehrotra. Local Dimensionality Reduction: A New Approach to Indexing High Dimensional Spaces. In Proceedings of the 26th VLDB Conference, 2000.

[8]. H. Chernoff. A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations. Annals of Mathematical Statistics, pages 493-509, 1952.

[9]. D. I. A. Us historical climatology network data, http://cdiac.esd.ornl.gov//epubs/ndp/ushcn_r.html

[10]. C. Domingo, R. Gavald, and O. Watanabe. Practical Algorithms for Online Selection. In Proceedings of the first International Conference on Discovery Science, 1998.

[11]. C. Domingo, R. Gavald, and O. Watanabe. Adaptive Sampling Algorithms for Scaling Up Knowledge Discovery Algorithms. In Proceedings of the second International Conference on Discovery Science, 2000.

[12]. P. Domingos and G. Hulten. Mining High-Speed Data Streams. In Proceedings of the Sixth ACM-SIGKDD International Conference on Knowledge Discovery and Data Mining, Boston, MA, 2000.

[13]. C. Faloutsos and V. Gaede. Analysis of the Z-ordering Method Using the Hausdor Fractal Dimension. In Proceedings of the International Conference on Very Large Data Bases, pages 40-50, September 1996.

[14]. C. Faloutsos and I. Kamel. Relaxing the Uniformity and Independence Assumptions, Using the Concept of Fractal Dimensions. Journal of Computer and System Sciences, 55(2):229-240, 1997.

[15]. C. Faloutsos, Y. Matias, and A. Silberschatz. Modeling Skewed Distributions Using Multifractals and the 80-20 law. In Proceedings of the International Conference on Very Large Data Bases, pages 307-317, September 1996.

[16]. K. Fukunaga. Introduction to Statistical Pattern Recognition. Academic Press, San Diego, California, 1990.

[17]. P. Grassberger. Generalized Dimensions of Strange Attractors. Physics Letters, 97A:227-230, 1983.

[18]. P. Grassberger and I. Procaccia. Characterization of Strange Attractors. Physical Review Letters, 50(5):346-349, 1983.

[19]. S. Guha, R. Rastogi, and K. Shim. CURE: An E_cient Clustering Algorithm for Large Databases. In Proceedings of the ACM SIGMOD Conference on Management of Data, Seattle, Washington, pages 73-84, 1998.

[20]. A. Jain and R. C. Dubes. Algorithms for Clustering Data. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[21]. L.S. Liebovitch and T. Toth. A Fast Algorithm to Determine Fractal Dimensions by Box Countig. Physics Letters, 141A(8), 1989.

[22]. R.J. Lipton and J.F Naughton. Query Size Estimation by Adaptive Sampling. Journal of Computer Systems Science, pages 18-25, 1995.

[23]. R.J. Lipton, J.F. Naughton, D.A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. Theoretical Computer Science, pages 195-226, 1993.

[24]. B.B. Mandelbrot. The Fractal Geometry of Nature. W.H. Freeman, New York, 1983.

[25]. D. Menasce, V. Almeida, R. Fonseca, and M. Mendes, A Methodology for Workload Characterizatoin for E-commerce Servers. In Proceedings of the ACM Conference in Electronic Commerce, Denver, CO.

[26]. H. Samet. Applications of Spatial Data Structures. Addison-Wesley, 1990.

[27]. John Sarraille and P. DiFalco. FD3. http://tori.postech.ac.kr/softwares/.

[28]. E. Schikuta. Grid clustering: An efficient hierarchical method for very large data sets. In Proceedings of the 13th Conference on Pattern Recognition, IEEE Computer Society Press, pages 101-105, 1996.

[29]. M. Schroeder. Fractals, Chaos, Power Laws: Minutes from an Infinite Paradise. W.H. Freeman, New York, 1991.

[30]. S.Z. Selim and M.A. Ismail. K-Means-Type Algorithms: A Generalized Convergence Theorem and Characterization of Local Optimality. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-6(1), 1984.

[31]. G. Sheikholeslami, S. Chatterjee, and A. Zhang. WaveCluster: A Multi-Resolution Clustering Approach for Very Large Spatial Databases. In Proceedings of the 24th Very Large Data Bases Conference, pages 428-439, 1998.

[32]. W. Wang, J. Yand, and R. Muntz. STING: A statistical information grid approach to spatial data mining. In Proceedings of the 23rd Very Large Data Bases Conference, pages 186-195, 1997.

[33]. O. Watanabe. Simple Sampling Techniques for Discovery Science. IEEE Transactions on Information and Systems, January 2000.