Estimating Cluster Population

Laxmi Gewali(✉), K.C. Sanjeev, and Henry Selvaraj

University of Nevada, Las Vegas, USA
{laxmi.gewali,henry.selvaraj}@unlv.edu

Abstract. Partitioning a given set of points into clusters is a well-known problem in pattern recognition, data mining, and knowledge discovery. One of the widely used methods for identifying clusters in Euclidean space is the K-mean algorithm. In using K-mean clustering algorithm it is necessary to know the value of *k* (the number of clusters) in advance. We present an efficient algorithm for a good estimation of *k* for points distributed in two dimensions. The techniques we propose is based on bucketing method in which points are examined on the buckets formed by carefully constructed orthogonal grid embedded on input points. We also present experimental results on the performances of bucketing method and K-mean algorithm.

Keywords: Clustering · Data partitioning · Bucketing method

1 Introduction

Clustering is a technique of identifying ‘closely related points’ from a collection of large number of given data points. Closely related points in terms of some distance metric are grouped together as a cluster. In most input data there could be several blocks of clusters. Cluster analysis is extensively used in many fields that include statistics, medicine, social sciences and humanities [1, 6]. In fact, any study that uses collection of data can make productive use of cluster analysis.

Most of the early research on cluster analysis is done by considering point distribution in Euclidean space, where Euclidean metric is used to measure distance between points. In this setting, distance between a pair of points in the same cluster is distinctly smaller than the distance between a pair formed by taking one point from the cluster and the other from outside the cluster. After the advent of computer science, researchers considered the problem of developing efficient algorithms for extracting clusters [3, 4]. K-Mean algorithm and its variations are example of practical algorithms for identifying clusters in Euclidean space. In recent years, there is surge in research interest for identifying clusters in big-data. In the normal data we can assume that all the data is available in the main memory and algorithms are developed by considering standard RAM model. In big-data, not all the data can be stored in RAM. The challenge is to develop cluster identification algorithms when data is available in external memory and the cloud. In some applications, Euclidean metric cannot be used to measure distance between points. For example, the data points could be visitors to Las Vegas entertainment sites and we may be interested to identify cluster of visitors who frequent casino sites and are coming from Hong Kong. Straightforward use of Euclidean metric may not be applicable for such data to extract clusters. We need to come up with appropriate metric other than the Euclidean. In statistics, a widely used technique for cluster analysis is the method of principal component analysis (pca). In this approach orthogonal transformation is performed to obtain linearly uncorrelated data from possibly correlated ones [5].

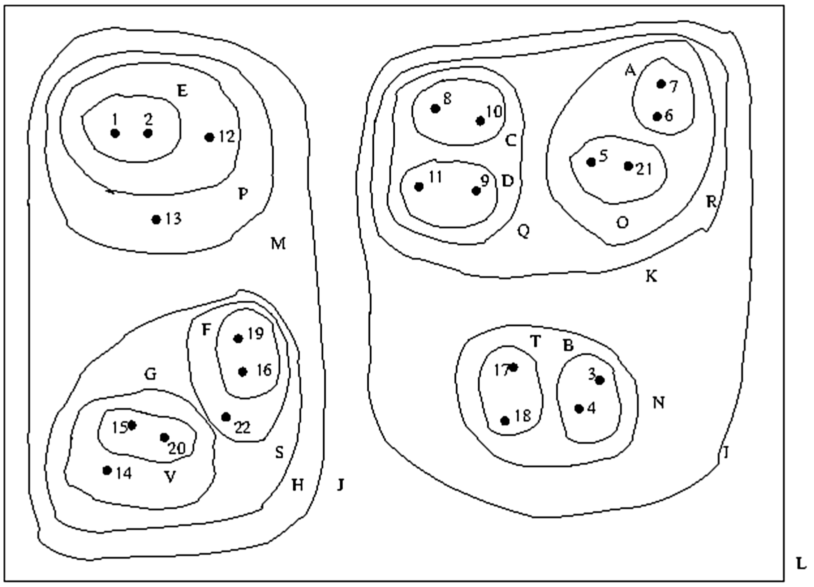
In this paper we address the issues of estimating the number of clusters for points distributed in Euclidean space. In Sect. 2, we present a brief review of the prominent existing methods for extracting clusters. In Sect. 3, we present the main contribution of the paper. We present an efficient algorithm for estimating the number of clusters and the location of their centers. In Sect. 4, we present preliminary results of the experimental investigation of the proposed algorithm.

2 Review of Existing Approaches

Clustering algorithms have been reported in Engineering and Statistics literature for almost last one hundred years [6, 7]. Most of the clustering algorithms are developed by using some variations of the following two general methods. In *hierarchical scheme*, each of the point p_i in the input data is considered as a cluster. Each cluster is associated with its centroid point which is taken as the arithmetic mean of the coordinates of the points in the cluster. Two clusters are picked to combine by formulating some metric. One simple way of combining clusters is to pick a pair of clusters whose centroids are closest. Another way to combine clusters is to consider the smallest distance between nodes from one cluster to the other. When a new cluster is formed by combining two smaller clusters, the corresponding centroid is also computed. The process of combining two clusters is continued until all points are grouped into one cluster. In some sense the hierarchical clustering scheme works by following the spirit similar as the construction of minimum spanning tree by using Kruskals' algorithm [2]. We can illustrate this strategy by an example shown in Fig. 1, where the top part shows the recursive process of cluster combination and the bottom part shows the implied tree.

In the *point assignment* strategy, clustering algorithms are developed by making an initial estimate of the number of clusters and approximate locations of their centers. The K-mean algorithm described next is an example of this strategy. The K-Mean Algorithm was first formally introduced by Stuart Lloyd [7] in connection with its application to pulse code modulation at Bell Lab. This algorithm is perhaps the most widely referred clustering algorithm for almost 35 years. The algorithm works for points distributed in Euclidean space. The algorithm assumes the number of clusters as a part of the input. The location of the initial k points is also specified by the user of the algorithm. The algorithm grows clusters by adding carefully selected nodes to the partially constructed clusters.

Initially, the k clusters have one node each. The locations of the initial single member in the clusters are taken as their centroids. The algorithm progresses through a series of steps to grow clusters by adding one node at a time. The nodes outside the clusters are unprocessed nodes. The algorithm examines an unprocessed node p_i as the next candidate point. The candidate point p_i is added to the cluster whose center is closest to p_i. This process of "adding a candidate point" is continued until all nodes are



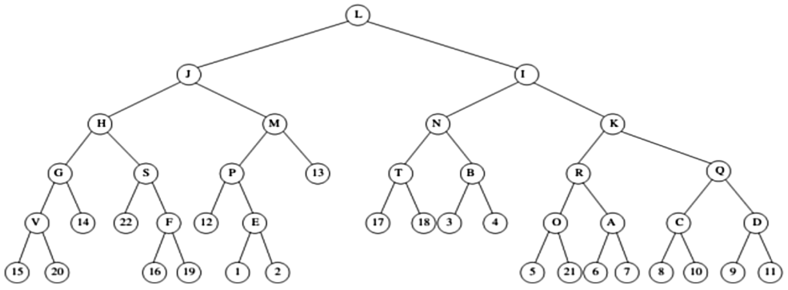


Fig. 1. Illustration of hierarchical clustering

processed. When all points are processed, one pass of the "clusters construction" is completed. After the completion of a pass the centroids are recomputed. The updated centroid of a cluster C_i is the centroid of all points included in it. A new pass of computation starts with respect to the newly updated centroids. In each pass the estimation of centroids and their memberships are updated. The initiation of the next pass stops when cluster members do not change or the change in the location of centroids is below a certain predetermined threshold value.

3 Estimation of Cluster Population

3.1 Preliminaries

As mentioned in the previous section, one of the most popular methods for constructing clusters from a given set of points distributed in Euclidean space is the *K*-mean algorithm [7]. This algorithm assumes that the number of clusters *k (cluster population)* is known in advance. If the value of *k* is not given as a part of the input then we need to estimate it 'somehow'. One straightforward technique would be to repeat the execution of the algorithm for several values of *k* and evaluate the quality of resulting solutions. The value of *k* that corresponds to the best value of cluster quality is the desired answer. A brute-force method is to try all values of k = 2,3,4,… n. A faster method based on the binary search technique has been suggested [6], for searching the value of *k*. Obviously the binary search technique is only effective where the quality of cluster as a function of *k* is a monotone function. An exhaustive searching approach has several demerits: (i) executing the clustering algorithm repeatedly is time consuming, (ii) measuring the quality of a candidate solution is not precise, and (iii) locating the cluster center for a given value of *k* is itself a difficult problem. We present next an innovative approach for estimating the value of *k* and the locations (co-ordinates) of cluster centers for points distributed in the Euclidean plane.

3.2 Adaptive Bucketing

Without loss of generality we can assume that the input point-sites $S = p_0, p_1, \ldots p_{N-1}$ are inside a rectangular box *R* of height = *h* and width = *w*. The box *R* can be divided into *n* by *m* orthogonal buckets. The value of bucket size *m* can be pre-determined by examining the distribution of the nearest neighbor distance distribution for *n* input points. An example of partitioning the bounding box *R* into orthogonal buckets is shown in Fig. 2 (top part).

A straight-forward approach for counting the points in each bucket is to check for point inclusion in each bucket. The bucket that returns 'true' for point inclusion is the bucket containing the point. Since the buckets are disjoint, only one bucket will return true for inclusion for a given point. To implement this approach we maintain a count array *cnt[]* whose entries are initialized to zeros. An array *bx[]* holds the coordinates of the top left corner of buckets. If the inclusion test for point $p_i(x_i, y_i)$ against bucket *bx[j]* returns true then *cnt[bx[j]]* is increased by 1. When this check is repeated for all points, point counts for all buckets are complete. A formal sketch of the algorithm based on this approach is listed as Straightforward Count Algorithm (Algorithm 1). It is easily seen that he time complexity of Algorithm 1 is $O(Nnm)$. If $n*m$ is comparable to N then the time complexity becomes $O(N^2)$ which is rather high.

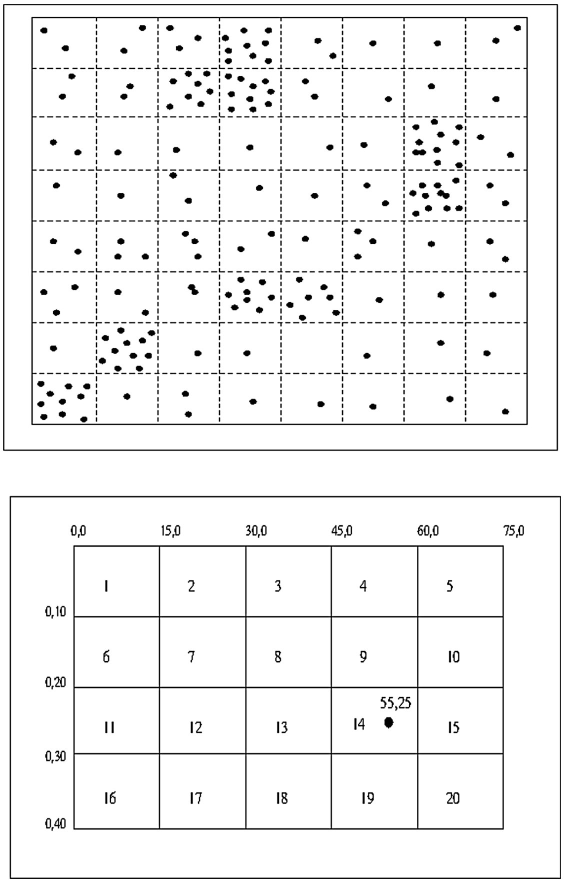


Fig. 2. Illustrating grid embedding and index mapping

Algorithm 1. Straightforward Count Algorithm

```
Input: (i) p[N]; // Input points in 2D
       (ii) int n, m; // Number of bucket rows and columns
       (iii) int bx[n,m]; //To hold top left corner of buckets
       (iv) int kv, kh; // length and width of each bucket
Output: cnt[]; // Array to hold count of bucket
Step 1: read p[N], n, m ,kv,kh
Step 2: for (int i = 0; i < n*m; i++)
            cnt[i] = 0;
Step 3: for (int i = 0; i <N; i++) f
            for (int j = 0; j < n*m; j++) f
                if (inside(bx[j], p[i]))   cnt[bx[j]]++;
Step 4: Output cnt[];
```

Mapping Count Approach: This approach is used to directly map p_i to the bucket $b[j]$ where it falls. Since the size of buckets are the same and rectangular, the index of the bucket where point p_i falls can be computed in term of the row number, column number, width, and height of the bucket. It is given that the outer rectangle R bounding the input points is partitioned into n columns and m rows of buckets, each of size k_v*k_h. Here k_v is the vertical extent of the bucket and k_h its horizontal width. For a given point $p_i(x_i, y_i)$, its row number r_n is given by $rn = y_i/k_v + 1$ and column number $c_n = x_i / k_h + 1$. We can index buckets left to right and top to bottom as 1, 2,……, $n*m$ as shown in Fig. 2 (bottom part). Then the bucket index corresponding to point $p_i(x_i, y_i)$ is $(r_n - 1) * n + c_n$. As an example, point $p_1(55, 25)$ is mapped bucket $(3-1)*5 + 4 = 14$.

Based on this mapping, the following is a faster algorithm (Algorithm 2) called Mapping Count Algorithm. The time complexity of Algorithm 2 is $O(N + nm)$. This time complexity is optimal in the sense that it takes $O(N)$ time to read the points and nm is at most N

```
Algorithm 2. Mapping Count Algorithm
    Input: (i) p[N]; // Input points in 2D
           (ii) int n;m; // Number of rows and columns
           (iii) int bx[n;m]; //To hold top left of buckets
           (iv) int $k_v$,$k_h$; // length and width of each bucket
    Step 1: read p[N], n, m, $k_v$, $k_h$ // Read input
    Step 2: for(int i = 0; i <n*m; i++)
                cnt[i] = 0;
    Step 3: for(int i = 0; i < N; i++)
                $r_n$ = yi/kv + 1;
                $c_n$ = xi/kh + 1;
                j = ($r_n$ - 1) * n + $c_n$
                cnt[bx[j]]++;
    Step 4: Output cnt[]
```

Remark 1 (*Number of buckets*): The very purpose of using buckets fails if there are too many buckets. For making the bucketing approach valid we do not want to have many empty buckets. At the same time to identify the boundaries of clusters we should have enough buckets. A good upper bound for the number of rows and columns in bucket partitioning is $N^{1/2}$. In some applications, the number of rows and columns is much smaller than $N^{1/2}$, and in some cases it is even constant.

3.3 Aggregating Bucket Clusters

After identifying buckets containing a high concentration of points, it is now necessary to aggregate buckets together belonging to the same cluster. We can clarify this with the following example in Fig. 3.

In this example there are two clusters $C1$ and $C2$. Cluster $C1$ has 10 buckets $b1$, $b2$, $b3$, $b4$, $b5$, $b6$, $b7$, $b8$, $b11$, $b12$ and cluster $C2$ has five buckets $b9$, $b10$, $b13$, $b14$, $b15$. Suppose the starting bucket is $b6$. The algorithm proceeds by initializing a queue Qb by

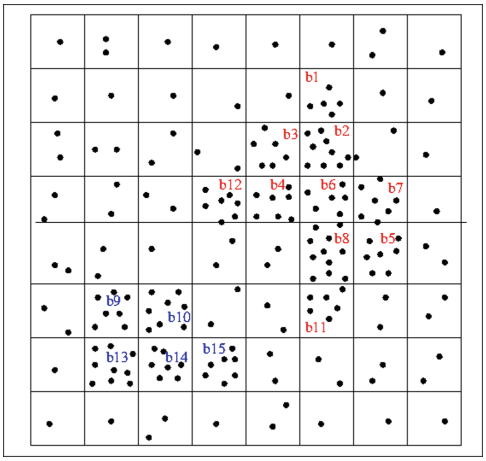


Fig. 3. Aggregating 'H'-buckets

inserting the starting bucket $b6$ to Qb. The algorithm then repeats the following generic task until all buckets of the cluster are aggregated.

Generic Task: Pick the bucket b_j from the front of the queue Qb and inset into queue all four connected neighbors of b_i that are marked H. Bucket b_j is pushed onto stack S_b and b_j is marked processed.

In our running example, bucket $b6$ is removed from the front of the queue Q_b and its 'H' marked neighbors that have not been processed yet ($b2$, $b4$, $b8$ and $b7$) are inserted into queued onto queue Q_b. Bucket $b6$ is marked processed. Next bucket $b2$ is removed from the queue and its unprocessed neighbors that are marked 'H' ($b1$ and $b3$) are inserted into queue onto the queue.

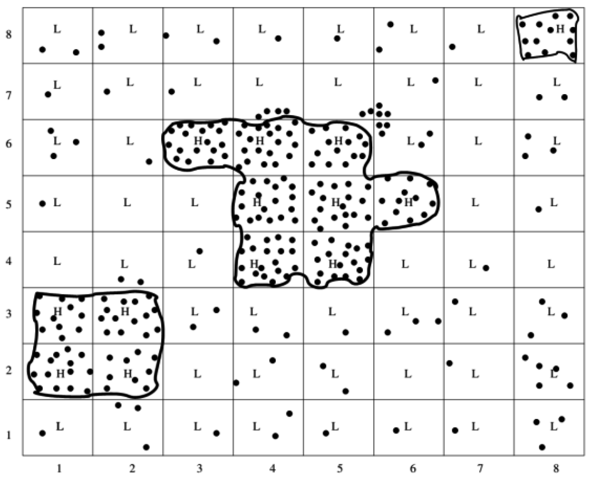
Bucket $b2$ is marked processed. These operations on stack and queue are repeated until the queue is empty. When the queue is empty all the buckets of the cluster in the context are present in the stack. A formal sketch of the algorithm which we refer to as Bucket Clustering Algorithm is listed as Algorithm 3.

```
Algorithm 3. Bucket Clustering Algorithm
    Input: (i) An array b[] of size m * n representing
                    the top left co-ordinates of buckets
           (ii) A given starting bucket index q that
                    belongs to current cluster
    Output: A stack containing the buckets
                representing the cluster counting b[q]
    Step 1: Q = b[q]; // Initialize queue Q
    // Initialize stack Sb to be empty
    Step 2: while (Q is not empty)
                a. Px = Q.delete();
                b. Let Rc be set of unprocessed
                        4-neighbors of Px
                c. Insert points in Rc into Q
                d. Push Px into stack Sb
                e. Mark points in Rc ‘processed'
    Step 3: Output Sb
```

3.4 Nudging

A straightforward application of the bucketing technique aggregates high count buckets (H-buckets) to extract a cluster. We refer to the clusters constructed in this way as *coarse clusters* and their boundaries as *coarse boundaries*. Some points in L-clusters adjacent to coarse boundaries are not included in the cluster even if they are very close to the fence of a H-bucket. Of course, points in L-buckets adjacent to a coarse boundary should not be included in the cluster if such points are farther away from the boundary and appear disconnected to the cluster.

In Fig. 4 (top part), the cluster at the center is formed by aggregating 8 buckets [4, 4], [5,4], [4,5], [5,5], [6,5], [3,6], [4,6] and [5,6]. However, boundary points in low count buckets [4, 7], [5, 7], [6, 7] and [6, 6] should be included in the cluster. When such boundary points are included in the cluster we get better estimation of the cluster as shown in Fig. 4 (bottom part). Now we describe a formal way of identifying points near the coarse boundary that can be included in the cluster. Our approach is to nudge coarse boundaries to capture proximity points in the corresponding cluster. Consider a H-bucket adjacent to a coarse boundary as shown in Fig. 5. If a L-bucket shares an edge with a H-bucket, then we can inspect points inside a rectangle of size $l \times l/4$ (strip rectangle) as shown in Fig. 5 to possibly include in the cluster, where l is the side length of the bucket. This is called strip nudging. If a L-bucket is adjacent to a corner of a H-bucket then we should inspect points inside an arc of radius $l/4$ and angle $3\pi/4$, as shown in the lower left of Fig. 5. This technique is called arc nudging.



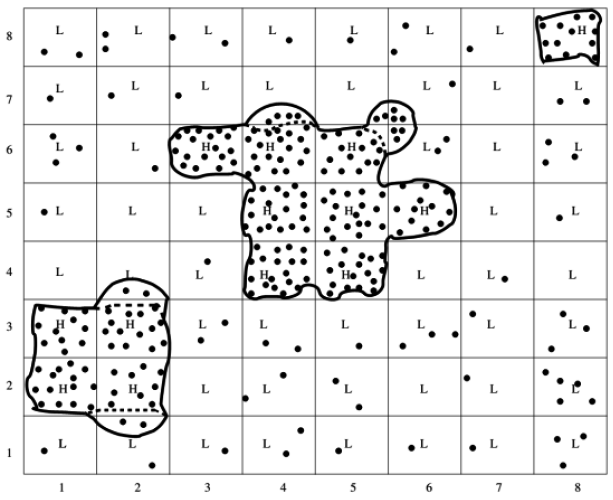


Fig. 4. Aggregation and nudging

4 Results and Discussion

We generated different examples with varying number of clusters and initial representative points to test the performance of the bucketing algorithm. To measure the quality of the solution obtained using the bucket clustering algorithm, we used the *sum of squared error (SSE)* as our objective function [3, 9].

We first calculate the squared error of each point to its closest centroid and computed the total sum of the squared errors for the clusters. A small SSE means the generated clusters better represent the points in the cluster. We calculated SSE for clusters generated using both the standard K-means algorithm and bucketing algorithm. The results are tabulated in Table 1.

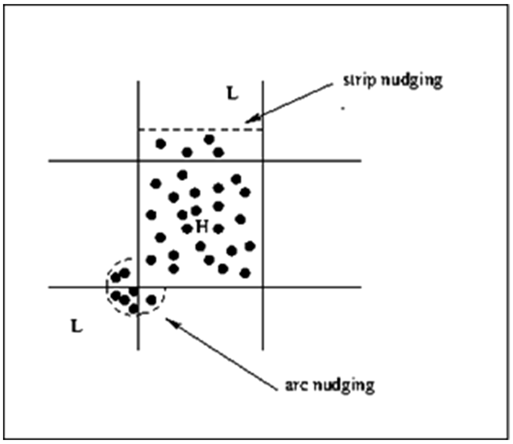


Fig. 5. Strip nudging and arc nudging

Table 1. Experimental Results

Method	Avg. SSE	Total clusters	Threshold points
K-mean (k = 4) Set-1	901703.25	4	N/A
K-mean (k = 4) Set-2	907126.25	4	N/A
Bucketing	24114140	1	4
Bucketing and Nudging	24566045	1	4
Bucketing	1279606	4	5
Bucketing and Nudging	1595448	4	5
Bucketing	599789	4	9
Bucketing and Nudging	881676	4	9
Bucketing	373290	3	13
Bucketing and Nudging	592832	3	13
Bucketing	122360	2	15
Bucketing and Nudging	282545	2	15

From the experimental results it is clear that when the threshold points are carefully selected, the clusters obtained using the bucketing algorithm, in most cases, have either less or almost equal SSE compared to the standard K-means algorithm. Due to the wrong selection of threshold points, in some cases, the SSE obtained from the bucketing algorithm is higher than the standard K-means. Overall, the bucketing algorithm provides almost the same or better SSE compared to original K-means. In addition, bucketing algorithm removes the necessity of providing the number of clusters at the beginning.

References

1. Berg, M., Krevald, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications, 2nd edn. Springer, Heidelberg (2000)
2. Cormen, T.H., Lieserson, C.E., Ronald, L., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
3. Guha, S., Rastogi, R., Shim, K.: CURE: An efficient clustering algorithm for large databases. Inf. Syst. **26**(2), 35–58 (2009). MIT Press
4. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of STOC, pp. 604–13 (1998)
5. Jolliffe, I.T.: Principal Component Analysis, 2nd edn. Springer, New York (2002)
6. Leskovec, J., Ullman, J.D., Rajaraman, A.: Mining of Massive Datasets. Cambridge University Press, New York (2014)
7. Lloyd, S.P.: Least square quantization in PCM. IEEE Trans. Inf. Theory **28**(2), 129–137 (1982)
8. O'Rourke, J.: Computational Geometry in C, 2nd edn. Cambridge University Press, Cambridge (1998)
9. Tan, P., Steinbach, M., Kumar, V.: Introduction to Data Mining. Pearson Addison Wesley, Boston (2005)