Smart-Sample: An Efficient Algorithm for Clustering Large High-Dimensional Datasets

Dudu Lazarov, Gil David, Amir Averbuch

School of Computer Science, Tel-Aviv University

Tel-Aviv 69978, Israel

Abstract

Finding useful related patterns in a dataset is an important task in many interesting applications. In particular, one common need in many algorithms, is the ability to separate a given dataset into a small number of *clusters*. Each cluster represents a subset of *data-points* from the dataset, which are considered similar. In some cases, it is also necessary to distinguish data points that are not part of a pattern from the other data-points.

This paper introduces a new data clustering method named *smart-sample* and compares its performance to several clustering methodologies. We show that smart-sample clusters successfully large high-dimensional datasets. In addition, smart-sample outperforms other methodologies in terms of running-time.

A variation of the smart-sample algorithm, which guarantees efficiency in terms of I/O, is also presented. We describe how to achieve an approximation of the in-memory smart-sample algorithm using a constant number of scans with a single sort operation on the disk.

1 Introduction

Clustering in data-mining methodologies identifies related patterns in the underlying data. A common formulation of the problem is: given a dataset S of n data-points in $\mathbb{R}^d$ and an integer k, the goal is to partition the data-points into k clusters that optimize a certain criterion function with respect to a distance measure. Each data-point is an array of d components (features). It is a common practice to assume that the dataset was numerically processed and thus a data-point is a point in $\mathbb{R}^d$.

The partition includes k pairwise-disjoint subsets of S, denoted as $\{S_i\}_{i=1}^k$, and a set of k representative points $\{C_i\}_{i=1}^k$ which may or may not be part of the original dataset S. In some cases, we also allow to classify a data-point in S as an *outlier*, which means that the data-point is an anomaly. In some applications, outliers can be discarded, while other applications may look for these outliers.

The most commonly used criterion function for data clustering is the square-error:

$$\phi = \sum_{i=1}^{k} \sum_{x \in S_i} \| x - C_i \|^2 \tag{1.1}$$

where in many cases C_i is the *centroid* of S_i, i.e. the mean of the cluster S_i is

$$mean(S_i) = \frac{\sum_{x \in S_i} x}{| S_i |}. \tag{1.2}$$

Finding an exact solution to this problem is NP-hard even for a fixed small k. Therefore, clustering algorithms try to approximate the optimal solution, denoted ϕ_{opt}, to a given instance of the problem. In section 2, we present examples for several practical and efficient methods to achieve it. However, measuring the quality of a chosen method is not a simple task. In addition to accuracy and computational efficiency, clustering algorithms should satisfy the following scenarios:

1. In many cases, the size of the set S is very large. Therefore, it cannot be loaded at once into the main memory. It is known that an expensive I/O performance of a very computationally-efficient algorithm still yields a slow algorithm;

2. From the definition of the clustering problem (Eq. 1.1), it is obvious that many solutions strongly rely on Euclidean distance between points. It is reasonable in low-dimensional spaces, while using Euclidean distance to cluster points in high-dimensional spaces is useless [1].

We present a new clustering method, named **smart-sample**, that has two variations:

1. An in-memory algorithm, which loads the entire dataset to memory;
2. A modified algorithm that is suitable for a bounded memory environments.

We demonstrate the performance of the algorithms on real data and compare it with existing methods. We find that smart-sample is at least as accurate as the other methods. In addition, it is usually faster and scalable with high-dimensional data.

This paper is organized as follows. In section 2, we describe various clustering algorithms by listing their advantages and disadvantages. The smart-sample algorithm is introduced in section 3. Experimental results are presented in section 4. It includes run-time and solution accuracy comparison between smart-sample methodology and current clustering methods.

2 Clustering algorithms: related work

Traditional clustering algorithms are often categorized into two main classes: *partitional* and *hierarchical*. A partitional clustering algorithm usually works in phases. In each phase, it tries to determine at once a partition of S into k subsets, such that a criterion function ϕ is improved compared to the previous phase. An hierarchical clustering algorithm usually works in steps, where in each step it defines a new set of clusters using previously established clusters. An *agglomerative* algorithm for hierarchical clustering begins with each point from a separate cluster and successively merges clusters that it considers closest with respect to some criterion. A *divisive* algorithm begins with the entire dataset S as a single cluster and successively splits it into smaller clusters.
In sections 2.1-2.3, we describe in details the algorithms studied in this paper.

2.1 k-means and its derivatives

The k-means algorithm [2] is probably the most popular clustering algorithm used today. Although it only guarantees convergence towards a local optimum, its simplicity and speed makes it attractive for various applications.
The k-means algorithm works as follows:

1. Initialize an arbitrary set $\{C_i^0\}_{i=1}^k$ of k centers. Typically, these centers are chosen at random with uniform probability from a dataset S.

2. Assign each point $x \in S$ to its closest center. Let S_i^j be the set of all points in S assigned to the center C_i^j in phase j.

3. Recompute the centers $C_i^{j+1} = mean(S_i^j)$.

4. Repeat the last two steps until convergence. It is guaranteed that ϕ in Eq. 1.1 is monotonically decreasing.

The k-means algorithm may yield poor results since $\frac{\phi}{\phi_{opt}}$ is not guaranteed to be bounded. However, if a bounded error is wanted, it is possible to use other initialization strategies instead of step (1). In other words, we can *seed* the k-means algorithm with an initial set $\{C_i^0\}_{i=1}^k$ that was chosen more carefully than just sampling S at random.

k-means++ [3] is such a method. Instead of choosing $\{C_i^0\}_{i=1}^k$ at random, in k-means++ the following strategy is used:

1. Select C_1^0 uniformly at random from S.
2. For $i \in \{2, \ldots, k\}$, select $C_i^0 = x'$ at random from S with probability $\frac{D_i(x')^2}{\sum_{x \in S} D_i(x)^2}$, where $D_i(x) = \min_{i \in \{1,\ldots,i-1\}} \parallel x - C_i \parallel_2$.

The use of this approach to seed the k-means algorithm guarantees that the expectation of $\frac{\phi}{\phi_{opt}}$ is bounded (see [3]).

2.2 Hierarchical Clustering

A fundamental algorithm for hierarchical clustering is described in [4]. The process begins by assigning each point to a separate cluster and successively merges the two closest clusters, until only k clusters remain. There are three main approaches to measure the distance between clusters U and V in hierarchical clustering:

$$\begin{aligned} dist_{single-linkage}(U, V) &= \min_{p \in U \wedge q \in V} \parallel p - q \parallel, \\ dist_{complete-linkage}(U, V) &= \max_{p \in U \wedge q \in V} \parallel p - q \parallel, \\ dist_{average-linkage}(U, V) &= \frac{\sum_{p \in U \wedge q \in V} \|p - q\|}{|U||V|}. \end{aligned}$$

The main weakness of these methods is that they do not scale well. The time complexity is $\Omega(n^2)$, where n is the size of the dataset. This is due to the fact that we need to calculate a whole matrix of distances between any pair of points.

A relaxed method can be considered to estimate the distance between clusters by taking into account only the centroid of each cluster. However, it is obvious that a straightforward implementation of this approach also yields an $\Omega(n^2)$ algorithm.

2.3 Clustering of large datasets

2.3.1 BIRCH

BIRCH [5] is a clustering method that handles large datasets. At any time, only a compact representation, whose maximal size is configurable, is stored in memory. This is achieved by using a designated data-structure called **CF-tree**. The CF-tree is a balanced search-tree whose internal nodes are structures called **Clustering Features** (**CF**).

A CF summarizes the information we have about a certain subset of S. If S' is a subset of S, $CF(S')$ is defined to be (N, LS, SS), where N is the number of data-points in S', LS and SS are their linear sum ($\sum_{x \in S'} x$) and squared sum ($\sum_{x \in S'} x^2$), respectively. It is easy to see that if $S'_1, S'_2 \subseteq S$ and $S'_1 \cap S'_2 = \emptyset$, then $CF(S'_1 \oplus S'_2) = CF(S'_1) + CF(S'_2)$.

Each internal node in the CF-tree has at most B child nodes. It sums their CF structures. This way, any internal node represents the union of all the data-points that are represented by its child nodes. Each leaf node contains at most L leaves, where a leaf is also a CF with the requirement that all the data points it represents are within a radius T. The leaf nodes are linked together so they can be efficiently scanned.

The BIRCH clustering algorithm has the following phases:

Phase 1: The dataset S is scanned and each data-point is inserted into the CF-tree. Insertion of a data-point x to a CF-tree is similar to insertion into a B-tree: starting from the root, the closest child node is chosen at each level. The node's centroid (as in Eq. 1.2) can be computed from its CF. If x is within a radius T of the corresponding leaf l, l absorbs x by an update of its CF and then the insertion is completed. Otherwise, a new leaf is created. If the parent of l has more than L leaves, then it should be split. Splitting of a node in a CF-tree is accomplished by taking the farthest pair of child nodes while redistributing the remaining entries so that each entry is assigned to a node closest to it. As in B-tree, node splitting may cascade up to the root.

Phase 2: A global clustering algorithm such as k-means or hierarchical clustering is applied to the centroids of the leaves to get the desired number of clusters.

Phase 3: Each data-point is reassigned to a cluster closest to it by using another scan of S. If the distance of a data-point from the closest cluster is more than twice the radius of the cluster,then this data-point is classified as an outlier.

Phase 4: (Optional) k-means iterations are used (as was described in section 2.1) to refine the result. Each iteration requires another scan of S.

As described before, BIRCH is used in cases where the memory is smaller than the size of the dataset. The main drawback of BIRCH is that it has many “free” parameters which are not easily tuned. BIRCH contains a method to dynamically adjust the value of T so that at any time the CF-tree is of bounded size, but the tree is not more compact than needed. It is done by starting with $T = 0$. After each insertion to the tree, its size is calculated. If it exceeds the size of available memory then T is increased and the CF-tree is rebuilt.

Roughly speaking, BIRCH proposes a practical way to perform clustering with a bounded memory size using a fixed number of scans of the input dataset.

2.3.2 CURE

The algorithms, which we surveyed in sections 2.1 - 2.3.1, are centroid-based: they usually take the mean of a cluster as a single representative of the cluster and make decisions accordingly. CURE [6] claims that centroid-based clustering algorithms do not behave well on data that is not sphere shaped, thus, a new method is proposed.

The core of CURE is an hierarchical clustering algorithm as was described in section 2.2. However, in order to compute the distance between two clusters, it considers a subset of (at most) c well-scattered representative points within each cluster instead of just considering the mean. In order to do it efficiently, CURE makes use of two data-structures: a kd-tree [9], which stores the representative points in each cluster, and a priority queue, which maintains the currently existing clusters with distance from the closest cluster as the sorting keys. As in the original agglomerative algorithm for hierarchical clustering, we begin with each point as a separate cluster. At each merging step, the cluster with minimal priority is extracted from the priority queue and merged with the cluster closest to it, which in turn is also deleted from the priority queue. By construction, these two clusters are the closest available clusters, where the distance measure is defined by

$$dist(U, V) = \min_{p \in Rep(U) \wedge q \in Rep(V)} \parallel p - q \parallel, \tag{2.1}$$

U and V are clusters, $Rep(U)$ and $Rep(V)$ are the sets of representative points in U and V, respectively.

While merging two clusters, we need to maintain the set of representative points and update the data structures. This is accomplished as follows:

1. Let W be the cluster obtained by merging the clusters U and V. The mean of W is calculated by $mean(W) = \frac{|W|mean(U)+|V|mean|V|}{|U|+|V|}$ (Eq. 1.2).

2. $L = \emptyset$. Repeat the following c times (c is a parameter):

   (a) Select the point $p \in W$ whose minimal distance (Eq. 2.1) from some other point $q \in L$ is maximal (if L is empty, use $mean(W)$ as if it was in L).

   (b) $L = L \cup p$.

3. The points $\{p + \alpha \cdot (mean(W) - p) \mid p \in L\}$ are taken as the representatives for W, where α is a parameter.

4. The representatives of U and V are deleted from the kd-tree data-structure. The representatives of W are inserted instead.

5. The keys of the clusters, which are stored in the priority queue, are updated by scanning it while comparing the current key to the distance from W. The current key of a cluster X should be set to $dist(X, W)$ if $dist(X, W) \leq key[X]$. Otherwise, if the current key of X corresponds to either U or V, then the new key can be calculated using the kd-tree structure: the closest neighbor to each representative of X is computed, and the distance from the closest one is chosen to be the key.
   During this process, the key of W can also be calculated by keeping the minimal value of $dist(W, X)$.

In the end of the process, clusters of very small size (e.g. 1 or 2) can be discarded since they are probably can be considered as outliers.

The worst-case time complexity of the algorithm is $O(n^2 \log n)$, where n is the size of the dataset. This makes it impractical to handle large datasets. Furthermore, it requires the entire dataset S to be in-memory, which is again impractical. Therefore, CURE uses two methods in order to reduce the time and space complexities:

Random Sampling: The task of generating a random sample from a file in one pass is well studied, and can be utilized in order to reduce the size of the dataset on which we apply the hierarchical clustering algorithm. For example, two such methods were described in section 2.1.

Partitioning: The basic idea is to partition the dataset into p partitions of size $\frac{n}{p}$, where n is the size of the dataset. Now, each partition is clustered until its size decreases to $\frac{n}{pq}$ for some constant $q > 1$. Then, the $\frac{n}{q}$ partial clusters are clustered in order to get the final clustering. This strategy reduces the run-time to $O(\frac{n^2}{p}(\frac{q-1}{q}) \log \frac{n}{p} + \frac{n^2}{q^2} \log \frac{n}{q})$ ([6]).

If random sampling is used, the final clusters will not include the unsampled points from S. Thus, in order to assign each point to a cluster, an additional scan of S is needed, in which each point is assigned to its closest cluster. To speedup the this computation, only a random subset of representatives are considered.

2.3.3 Locality-Sensitive Hashing (LSH)

LSH [7] is a randomized algorithm that was originally used for the approximated nearest neighbor (ANN) search for points in an Euclidean space. LSH defines a set of hash functions which store close points in the same bucket of some hash-table with high probability, and distant points are stored in the same bucket with low probability. Thus, when searching for the nearest neighbor of a query point q, only the points within the same bucket are considered. The LSH algorithm, which searches for the approximated nearest neighbors, is described in next in **ANN Search using LSH** and data clustering using LSH is described **Data-Clustering using LSH**.

ANN Search using LSH Given a set of points S in $\mathbb{R}^d$. It is transformed into a set P where all the components of all points are positive integers. This is done by translating the points so that all its components are positive. Then, they are multiplied by a large number and rounded to the nearest integer. Obviously, the error induced by this rounding procedure can be made arbitrarily small by choosing a sufficiently large number.

Let C be the maximal component value of all points. l sets of b distinct numbers each are selected from $\{1, 2 \ldots, Cd\}$ uniformly and randomly, where l and b are parameters of the algorithm. Assume I was generated this way. The corresponding hash function $g_I(p)$ is defined as follows: each number in I corresponds to a component of the point p. For $i \in I$, let

$$B_i(p) \triangleq \begin{cases} 1 & \text{if } (i-1 \bmod C)+1 \leq p(\lceil \frac{i}{C} \rceil) \\ 0 & \text{otherwise} \end{cases}.$$

$g_I(p)$ is defined to be the concatenation of the bits $B_i(p)$, $i \in I$. Obviously, if b is bigger then the range of g_I becomes wider. Thus, another (standard) hash function is used in order to map it to a more compact range.

When the hash functions were chosen, l hash values are computed for each point in P. As a result, l hash tables are created. In order to perform a query q, l hash values are computed for q and the nearest point is determined by measuring the actual distance to q among all the points stored in the buckets that were computed for q.

Data-Clustering using LSH In this section, we describe an agglomerative hierarchical clustering algorithm [8], which uses LSH to approximate the single-linkage hierarchical algorithm described in section 2.2.
As always, we begin by assuming that each point is in a separate cluster. Then, we proceed as follows:

1. P is constructed from S as was described above.
2. l hash-functions are generated as was described above, using b numbers each.
3. For each point $p \in P$, l hash values $\{h_i(p)\}_{i=1}^{l}$ are calculated. p is stored in the i-th hash table using the hash-function $h_i(p)$. If a corresponding bucket in some hash-table already contains a point that belongs to the same cluster as p then we do not insert p to this bucket.
4. For a point $p \in P$, let $N(p)$ be the set of all points which are stored in the same bucket as p within any hash table. $N(p)$ and the distances between p and the points in $N(p)$ are calculated for each point $p \in P$. If the distance of the closest point is less than r (described below), the corresponding clusters are merged.
5. r is increased and b is decreased.
6. Steps 1-4 are repeated until we are left with no more than k clusters.

It is suggested in [8] to take $r = \frac{dC}{k+d}\sqrt{d}$. In step 4, we can choose a constant ratio α, then r is increased and b is decreased by α.

The run-time complexity of the LSH clustering method is $O(nB)$ in the worst case, where B is the maximal size of any bucket in the hash-tables ([8]). Thus, if we limit the bucket size as suggested in [7], we get a worst-case result. In addition, if the hash-tables are sufficiently big, B can be smaller than n, which yields an almost linear time complexity algorithm.

3 The smart-sample clustering algorithm

In this section, we introduce an heuristic approach for partitional clustering that we call smart-sample. Our approach is similar in many ways to other methods such as the BIRCH algorithm. However, it has many benefits over other methods since it overcomes successfully the problems that were described in section 1. Smart-Sample is easy to implement, it is efficient and its output is at least as good as the other methods.

In section 3.1, we describe the in-memory version of the algorithm and show how to use local PCA computations in order to efficiently cluster high-dimensional data. In section 3.2, we show how to approximate the in-memory algorithm in order to get an efficient algorithm in terms of I/O.

3.1 In-memory algorithm

The basic idea behind the smart-sample algorithm is to perform some type of pre-clustering on the data. Then, traditional clustering methods are used in order to derive the final clustering. The goals of the pre-clustering phase are:

1. The dimension of the dataset is reduced using local PCA computations. The quality of the clustering is believed to be affected by the dimensionality of the data. However, application of global dimensionality reduction methods to a dataset before performing clustering does not usually improve the result (it even makes it worse).

2. The size of the dataset on which we apply the final clustering phase is reduced. This is done in order to fit it in memory and to speed the computation.

A flow of the in-memory version of the smart-sample algorithm is given in Fig. 1.

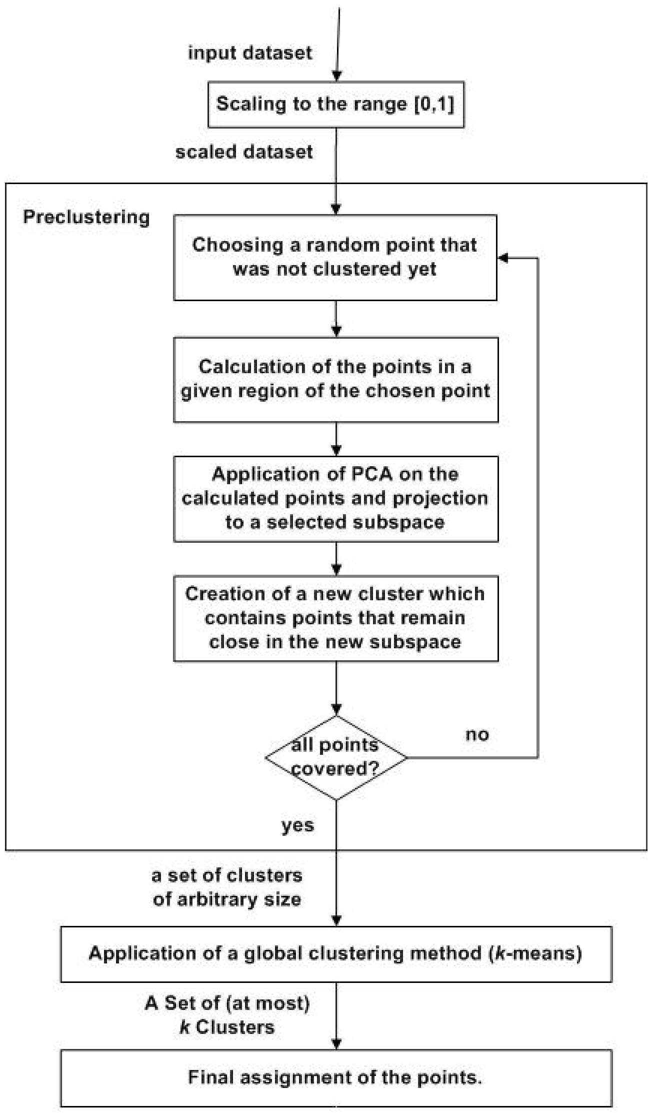

input dataset
Scaling to the range [0,1]
scaled dataset
Preclustering
Choosing a random point that was not clustered yet
Calculation of the points in a given region of the chosen point
Application of PCA on the calculated points and projection to a selected subspace
Creation of a new cluster which contains points that remain close in the new subspace
all points covered?
no
yes
a set of clusters of arbitrary size
Application of a global clustering method (k-means)
A Set of (at most) k Clusters
Final assignment of the points.

Figure 3.1: Overview of the in-memory smart-sample algorithm

In addition to the dataset S and the known in-advance number of clusters, the algorithm uses two additional parameters: the radius $R \in \mathbb{R}$ and the number $P_C \leq d$ of eigenvectors. At each step in the pre-clustering phase, an arbitrary data-point $p \in S$ is chosen. Then, all the data-points that are included in a ball of radius R around p are calculated. All the points in this ball are projected into a lower-dimension subspace that is spanned by at most P_C eigenvectors which are derived from the application of PCA to the points. We then create a new cluster that contains all the points around p which remain relatively close to it in the lower-dimension subspace. We repeat this process until all the points are taken into consideration. This is described in Algorithm 1.

```
Algorithm 1: Smart-Sample Pre-clustering
Input: $S \subset \mathbb{R}^d$, $k$, $R$, $P_C$.
Initialization: $U = \emptyset$.
Repeat until $U = S$:
    1. Select a point $p \in S \setminus U$ at random;
    2. The set $\texttt{Ball}(p) = \{x - p \mid x \in S \wedge \parallel x - p \parallel < R\}$ is calculated;
       If $\texttt{Ball}(p) = \{\overrightarrow{0}\}$ then $U = U \cup \{p\}$. Go to step 1;
    3. If $P_C < d$ then apply PCA to $\texttt{Ball}(p)$ and reduce its dimension to $P_C$. This is achieved
       by projection into the subspace that is spanned by the $P_C$ eigenvectors with the largest
       eigenvalues. Otherwise, $\texttt{Ball}(p)$ is kept as is. Let $\texttt{Ball}_{P_C}(p)$ be the output from this
       step
    4. Calculate the diameter $\overrightarrow{D}$ of the projected points along each direction: $\overrightarrow{D}(i) =$
       $\max_{x \in \texttt{Ball}_{P_C}(p)} x(i) - \min_{x \in \texttt{Ball}_{P_C}(p)} x(i)$ , $i \in \{1, \ldots, P_C\}$;
    5. The set $\texttt{Ellipse}(p) = \{x \in \texttt{Ball}_{P_C}(p) \mid \sum_{i=1}^{P_C} \frac{x(i)}{\overrightarrow{D}(i)} \leq 1\}$ is calculated;
       If $\texttt{Ellipse}(p) = \{\overrightarrow{0}\}$ then $U = U \cup \{p\}$. Go to step 1;
    6. A new cluster $C$ is created. $C$ contains the points in $S$ that correspond to the points
       in $\texttt{Ellipse}(p)$. $mean(C)$, as was defined in Eq. 1.2, is chosen as its center;
    7. $U = U \cup C$. Go to step 1.
```

At the end of this pre-clustering phase, we can have more than k clusters. We proceed by running a traditional clustering algorithm such as k-means on the set of the derived centers. We get a new set of k centers, where each of them can be associated with few 'sub-centers'. We associate each point to a corresponding center to get the final clustering result.

Obviously, the optimal value of the parameter R is input-dependent and difficult to determine. Thus, the input dataset is scaled so that all the values are in the range $[0, 1]$.

Then, R is also in the range $[0, 1]$. A low value of R yields an algorithm that is similar to the traditional k-means, while an higher value of R yields a smaller set of candidate clusters in the final stage where we apply the k-means algorithm. This idea is summarized in Algorithm 2.

A major drawback of the PCA approach is that it is hard to determine which eigenvectors to choose. Empirical tests show that taking those which correspond to largest eigenvalues does not necessarily produce the optimal solution. Thus, a possible variation of Algorithm 1 is to try exhaustedly all the possible subsets of eigenvectors. The one, which provides the best result with respect to some criterion such as maximal cluster size or minimal average distance from the center, is chosen. If there are too many options, it is possible to check a small number of these subsets which are selected at random.

Algorithm 2: `In-Memory Smart-Sample`

1. The input dataset S is scaled so that all values are in the range $[0, 1]$;
2. Pre-clustering using Algorithm 1 is applied to S;
3. k-means is applied to the set of centers in order to get a new set of centers of size at most k;
4. Each point in S is associated with the corresponding center.

3.2 I/O efficient algorithm

Smart-sample, as was described in section 3.1, is difficult to implement on external memory (disk) since in each iteration we need to apply the PCA to a set of points of unknown size. In addition, finding this set of points is not obvious since a naive exact calculation of "balls" of a given radius requires a scan of the entire dataset.

In this section, we show how to use the LSH method (section 2.3.3) in order to get an efficient I/O solution to approximate the in-memory version of the smart-sample algorithm. An overview of the I/O efficient version of the algorithm is given in Fig. 2.

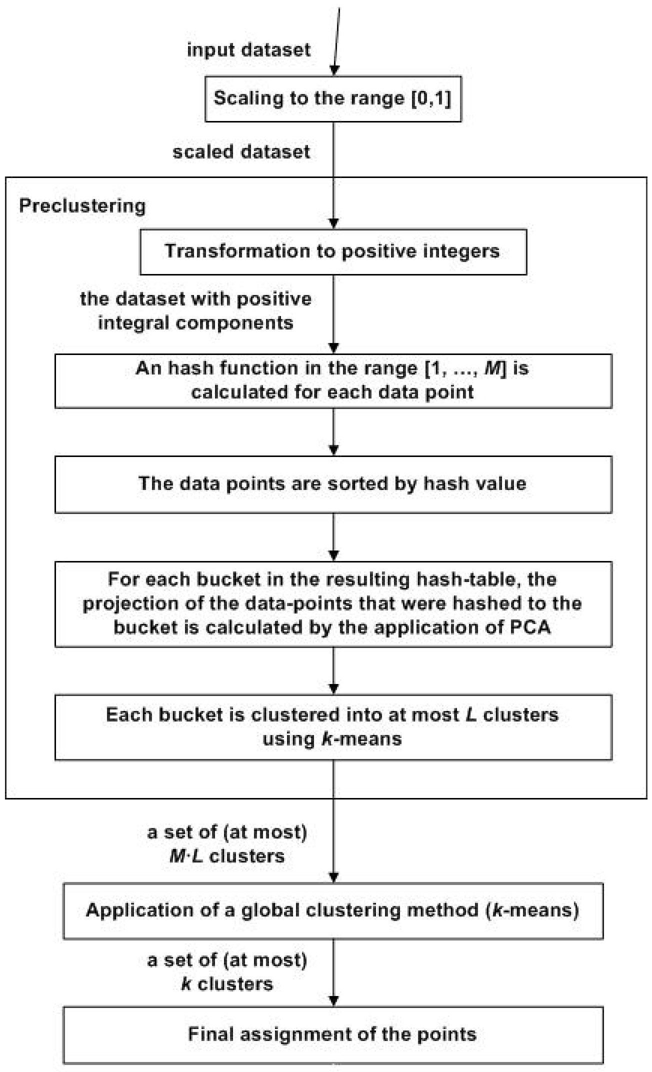

input dataset
Scaling to the range [0,1]
scaled dataset
Preclustering
Transformation to positive integers
the dataset with positive integral components
An hash function in the range [1, ..., M] is calculated for each data point
The data points are sorted by hash value
For each bucket in the resulting hash-table, the projection of the data-points that were hashed to the bucket is calculated by the application of PCA
Each bucket is clustered into at most L clusters using k-means
a set of (at most) M·L clusters
Application of a global clustering method (k-means)
a set of (at most) k clusters
Final assignment of the points

Figure 3.2: Overview of the I/O-efficient smart-sample algorithm

We use the LSH methodology in order to get an initial estimate of the distance between data-points. Thus, as required by LSH, we have to scale the input dataset S so that all its components are positive integers. This requires a scan of the input in order to find the minimal value and the maximal value C of all the components of the data-points in S. Then, by using another scan, S is shifted so that all its components are positive and then scaled so that all components are positive integers within a known range. This results in a new dataset P that was described in section 2.3.3.

As in the original LSH, we have to calculate a set I of b distinct numbers. By using another scan of the input, we calculate the hash value of each data-point in P. Each point with its hash value is saved on a disk. We can bound the hash-value range by a large constant M using another standard hash function. Now, the dataset is sorted according to hash-values using a standard I/O efficient sorting method (see [10]).

It is now possible to scan the dataset with an increasing order of its hash-values. Each time, the points, which were hashed to the same bucket, are loaded into memory. Note that if there are more points in the bucket than what can be loaded into memory, then we have to discard some of these points. However, choosing a sufficiently large value of M should make this scenario rare. To each bucket, we apply a PCA and project the points into a subspace of dimension P_C. In contrast to the in-memory version of the algorithm, this time we expect the bucket to contain several different clusters due to the fact that the hashing method is probabilistic and due to the loss of accuracy caused by the discretization of the dataset. We also use a second hash function in order to limit the range of the hash values to M. Thus, we apply k-means to the points in S which correspond to points that are included in the bucket and request from it to generate at most L clusters, where L is a constant input parameter.

We get a set of at most $M \cdot L$ centroids. Obviously, the values of L and M should be chosen so that this set can be loaded at once into memory. As before, we apply k-means to the centroids. By using another scan, we can associate each data-point to its closest centroid. Note that the in-memory smart-sample algorithm can also be used instead of k-means. However, smart-sample is intended to be used on large datasets. $M \cdot L$ should be a relatively small number. Therefore, in this case we believe that k-means should be used.

In Algorithm 3, we summarize this method, which approximates the in-memory smart-sample algorithm using a constant number of scans of the input and a single external memory sorting operation.

Algorithm 3: `I/O Efficient Smart-Sample`

Input: $S \subset \mathbb{R}^d$, k, M, L, P_C.

1. Transform the input dataset S so that all its values are positive integers. Let P be the output of this step and let C be the maximal value for all its components;

2. Select a set I of b distinct numbers chosen uniformly and randomly from $\{1, 2, \ldots, C \cdot d\}$;

3. For each point in P, its hash-value (in the range $[1, \ldots, M]$) is calculated using the hash function described in section 2.3.3;

4. P is sorted according to the calculated hash values;

5. For each bucket, PCA is applied to the corresponding points in S that were hashed into the same bucket. Their projections into a P_C-dimensional subspace are computed. At most L centroids of the points are computed by using a standard k-means;

6. The output of step 5 is a set of at most $M \cdot L$ centroids. The k-means algorithm is applied again to this set of centroids. It finds the k most "significant" clusters, i.e. the set of centroids that minimizes the potential ϕ (Eq. 1.1) as far as k-means guarantees;

7. Each point in S is associated with the center that is closest to it.

As in the in-memory case (Algorithm 1), the projection in step 5 of Algorithm 3 can be applied by checking different subsets of eigenvectors of size P_C. The one, which minimizes the potential ϕ with respect to the points and the centers contained in the bucket, is chosen. Then, the projection of the relevant points in S into the subspace, which is spanned by the P_C eigenvectors that are contained in the chosen subset, is computed.

4 Experimental results

Performance comparisons among all the methods, that were described in sections 2 and 3, were done. All the methods were implemented from scratch using Matlab, and tested on a dual-core Pentium 4 PC machine equipped with 2GB of memory.

4.1 The dataset

We experimented with a dataset that was captured by a network sniffer. Each data-point in the dataset includes 17 features from the sniffed data. Each sample was tagged with the relevant protocol that generated it. The dataset contains samples that were generated by 19 different protocols. However, we do not know exactly how many clusters we expect to get, since different samples from the same protocol can look very different. The dataset was randomly divided into a training set which contains about 5,000 samples (data-points) and a test set which contains about 3,500 samples. Each sample (data-point) contains 17 features. The goal is to classify different protocols into different clusters.

4.2 Algorithms

During the testing procedure it became clear that traditional hierarchical clustering is impractical due to its long running time. Therefore, it was omitted from our experiments. The algorithms that were compared are: smart-sample (the in-memory and the I/O efficient versions), k-means and k-means++, LSH, BIRCH and CURE. In order to compare between our approach and global dimensionality reduction, k-means and k-means++ were applied to a 10-dimensional version of the dataset, in addition to its application to the original dataset. The reduced dataset was obtained by the application of a global PCA to the original dataset.

4.3 Testing methodology

During the training phase, we apply a clustering algorithm to a dataset. The number of clusters (k) varied between 10 and 100. When clustering algorithms, which require more parameters, are applied. then these parameters provided in several variations. The output of each clustering algorithm is a set of at most k centroids and an association between each data-point and its centroid (which is a representative of a cluster). For each cluster, the tags of the data-points, associated with this cluster, are checked in order to verify whether they were generated by the same protocol. Let C_i be the i-th cluster and $C_i^M \subseteq C_i$ be a subset of maximal size in C_i which contains points that are identically tagged. For example, if an optimal solution is given, we get that $C_i = C_i^M$ for $i = 1, \ldots, k$. In practice, we do not expect to get the optimal solution. Therefore, we assign a score for each clustering. If $|C_i^M|/|C_i| \leq \frac{1}{2}$, we consider the entire cluster as an error. Otherwise, we consider the points in C_i^M as if they were clustered correctly. The points in $C_i \setminus C_i^M$ are classified as an error. Thus, the total score for the training phase is

$$score(C_1, \ldots, C_k) = \frac{\sum_{\{i|1\leq i\leq k \wedge |C_i^M|/|C_i|>\frac{1}{2}\}} |C_i^M|}{\mid S \mid}. \tag{4.1}$$

The training phase is repeated 10 times. After each iteration, the score is calculated. We keep the clusters with the maximal score from each algorithm. Each cluster has a representative and a tag associated with it. In the testing phase, each data-point in the test-set is associated with the closest representative. The score for the testing phase is the ratio between the number of points associated with representative points, which are similarly tagged, and the total number of points in the testset.

In order to guarantee the testing procedure integrity, we included a "sanity" test where the clusters are randomly chosen. Indeed, in this case, the training score, which was defined in Eq. 4.1, was close to zero.

4.4 Accuracy comparison

Tables 4.1 and 4.2 summarize the scores for the training and for the testing phases, respectively. The scores are given in percentage, i.e. they were calculated by multiplying Eq. 4.1 by 100. A score indicates the percentage of the samples that were clustered correctly. The tables contain the best score for each algorithm that we achieved while checking various configurations of the input parameters. We can see in Table 4.1 that for all values of k no algorithm outperformed all the other algorithms consistently. However, the scores in Table 4.2 are more interesting. An algorithm can perfectly learn a relatively small dataset, but it is more interesting to check how it performs on new data.

In Table 4.2, the best score for each k is denoted by bold. Here we can see that smart-sample, as was described in Algorithm 2, outperforms the other algorithms for most of the values of k. However, the gap between the best two scores for each value of k is less than 1%.

k	Smart-Sample		k-means		k-means++		LSH	BIRCH	CURE
	Alg. 2	Alg. 3	$d = 17$	$d = 10$	$d = 17$	$d = 10$			
10	65.14	65.33	64.73	64.20	62.24	60.94	4.98	64.01	39.79
20	70.85	66.69	69.79	69.13	70.98	68.90	11.37	69.99	56.71
30	71.19	68.26	71.81	69.56	71.40	70.78	12.28	72.14	66.51
40	71.58	72.08	72.39	73.07	72.69	73.54	12.65	73.85	70.49
50	72.19	72.30	73.75	72.16	72.94	73.81	12.96	74.10	72.26
60	73.43	71.99	73.31	72.61	73.64	73.56	13.23	73.89	72.53
70	73.18	72.22	73.07	73.31	73.83	73.66	13.64	74.43	74.01
80	73.49	72.92	73.15	73.25	73.99	73.42	13.99	74.37	73.66
90	73.41	72.69	73.64	73.38	74.51	73.99	14.69	74.63	73.85
100	73.82	73.75	73.29	73.46	73.77	74.10	14.86	74.47	74.43

Table 4.1: The scores for the training phase (in percentage). The training dataset contains 5,000 samples. k is the requested number of clusters and d is the number and d is the number of dimensions (features)

k	Smart-Sample		k-means		k-means++		LSH	BIRCH	CURE
	Alg. 2	Alg. 3	$d = 17$	$d = 10$	$d = 17$	$d = 10$			
10	**68.25**	67.17	66.04	66.16	66.51	66.74	42.57	67.32	42.45
20	**72.55**	71.73	71.70	71.21	71.56	71.38	45.38	71.79	66.71
30	**74.03**	72.28	73.15	73.47	73.41	73.44	53.39	73.38	70.63
40	74.72	73.88	73.76	73.91	73.65	74.40	43.47	**75.07**	71.15
50	**75.10**	73.85	74.92	74.08	74.57	74.26	44.13	74.92	73.21
60	**75.54**	74.46	75.50	74.63	74.84	74.89	38.62	74.98	73.04
70	**75.71**	74.63	75.47	75.39	75.68	75.27	36.85	75.65	74.55
80	**76.12**	75.10	75.44	75.85	75.88	75.53	41.67	75.97	74.37
90	**76.00**	75.30	75.91	75.59	75.65	75.82	39.81	75.85	74.72
100	76.12	75.68	75.59	74.98	76.14	**76.20**	43.61	75.91	75.04

Table 4.2: The scores for the testing phase (in percentage). The testing dataset contains 1,500 samples. k is the requested number of clusters and d is the number of dimensions (features)

4.5 Run-time comparison

For most of the algorithms, the scores of the training and the testing phases are more or less the same. Thus, we compare between the run-time of these algorithms. There are two different classes of algorithms:

1. k-means and its variations and the in-memory version of smart-sample algorithms are not suitable to handle very large datasets.

2. LSH, BIRCH, CURE and the I/O efficient version of smart-sample can handle large datasets.

The average run-time for each algorithm is summarized in Table 4.3. It is obvious from Table 4.3 that smart-sample scales much better to different values of k than the other algorithms.

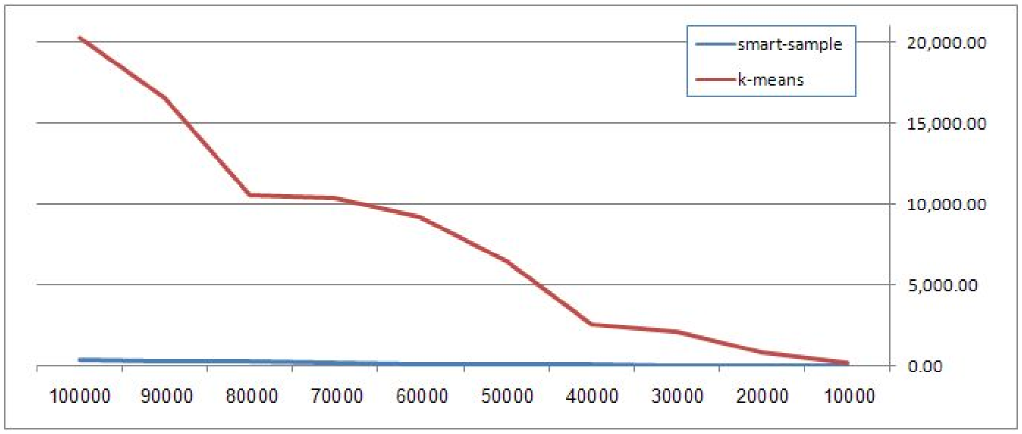
To see it even more clearly, we created a 20-dimensional random datasets of different sizes and repeated the training and testing procedures. Although this does not indicate the run-time of the algorithms on real data, it still shows how bad the algorithms can behave when applied to wrong datasets. The results are summarized in Table 4.4 and Fig. 4.1.

k	Smart-Sample		k-means		k-means++		LSH	BIRCH	CURE
	Alg. 2	Alg. 3	$d = 17$	$d = 10$	$d = 17$	$d = 10$			
10	0.52	6.15	0.84	0.35	4.24	4.43	401.31	19.09	822.25
20	0.54	6.2	2.56	2.11	9.65	7.35	377.42	20.02	766.59
30	0.57	6.25	2.98	1.29	13.35	12.35	347.38	20.63	773.71
40	0.62	6.31	5.60	1.32	25.22	18.40	349.28	21.64	814.32
50	0.62	6.31	11.01	2.01	25.29	22.17	337.78	23.50	808.84
60	0.66	6.25	11.38	2.45	37.99	29.51	339.28	26.10	765.81
70	0.68	6.31	13.03	2.97	39.14	35.15	321.33	26.99	787.84
80	0.70	6.47	18.13	5.37	51.61	49.21	328.5	27.19	791.25
90	0.66	6.48	18.49	6.28	64.56	51.09	323.38	30.78	795.59
100	0.67	6.48	19.91	10.27	64.50	56.73	328.53	32.86	781.09

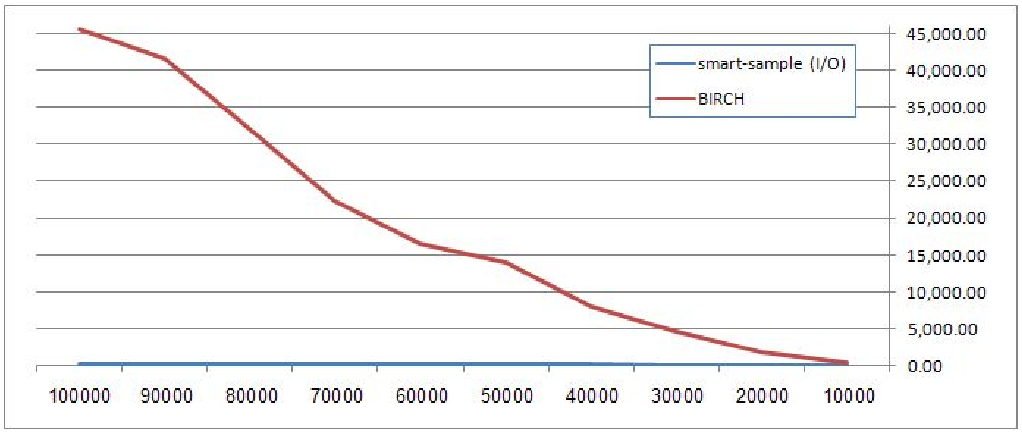
Table 4.3: Average run-time during the training phase on a dataset which contains 5,000 samples (in seconds). k is the number of requested clusters and d is the number of dimensions (features)

Number of samples	Smart-Sample		k-means	BIRCH
10,000	3.84	29.13	176.80	516.84
20,000	14.46	69.53	783.37	1867
30,000	31.42	135.89	2080.40	4650.2
40,000	57.83	174.90	2541.7	8040.4
50,000	85.97	182.55	6485.4	14050
60,000	123.21	305.92	9184.8	16574
70,000	168.11	301.96	10391	22425
80,000	221.30	218.04	10569	32140
90,000	281.85	217.19	16630	41547
100,000	349.18	236.14	20339	45589

Table 4.4: Average run-time on random data (in seconds) where $k = 10$. and d is the number of dimensions (features).



(a)



(b)

Figure 4.1: (a) Run-time in seconds of memory-based algorithms. (b) Run-time in seconds of disk-based algorithms.

5 Conclusion and future work

We presented a new method for clustering large, high-dimensional datasets, which is significantly faster than the currently used algorithms. It also generates accurate results. Smart-sample was tested also on other datasets by using the same parameters that were used in section 4.

Several open questions still remain:

1. It is clear that the LSH method did not handle our datasets very well. We believe that the hash-function methodology should be explored more for the case where the dataset contains non-integral values. This issue probably decreases the performance of the efficient I/O version of smart-sample as well.

2. The issue of choosing the right subset of eigenvectors, which are derived from the local PCA computations, should be investigated. It is not obvious how many eigenvectors to choose and which. In our experiments, we used the largest eigenvalues. Trying all the possibilities was too expensive thus the application of other methodologies should be explored.

3. How to handle outliers? The in-memory version of smart-sample, as suggested in this paper, can have many data-points that are classified as outliers. This can be handled by associating each point to the closest center. However, this leads back to the issue of how to measure a distance in high-dimensional space. We may consider the option to keep for each cluster the sub-space calculated for it and then comparing each point within this subspace. This strategy may also be used in the testing phase described in section 4.

References

[1] A. Hinneburg, C. C. Aggarwal, and D. A. Keim, What is the nearest neighbor in high dimensional spaces?, Proc. 26th Intl. Conf. Very Large DataBases, 2000, pp. 506-515.

[2] Stuart P. Lloyd. Least squares quantization in PCM. IEEE Transactions on Information Theory, 28(2), 1982, pp. 129-136.

[3] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In Proceedings of SODA. 2007, pp. 1027-1035.

[4] S. C. Johnson. Hierarchical Clustering Schemes. Psychometrika, 32, 1967, pp. 241–254

[5] T. Zhang, R. Ramakrishnan and M. Livny. BIRCH: An efficient data clustering method for very large databases. In Proceedings of the ACM SIGMOD Conference on Management of Data, 1996, pp. 103-114.

[6] S. Guha, R. Rastogi and K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. In Proceedings of SIGMOD Conference, 1998, pp. 3-84.

[7] A. Gionis, P. Indyk and R. Motwani. Similarity Search in High Dimensions via Hashing. Proc. of the 25th VLDB Conference, 1999, pp. 518-528.

[8] H. Koga, T. Ishibashi and T. Watanabe. Fast Hierarchical Clustering Algorithm Using Locality-Sensitive Hashing. Knowledge and Information Systems, Vol. 12, 2007, pp. 25-53

[9] H. Samet. The Design and Analysis of Spatial Data Structures. Addison-Wesley Publishing Company, Inc., New York, 1990.

[10] A. Aggarwal and J. S. Vitter, The input/output complexity of sorting and related problems, Comm. ACM, 31, 1988, pp. 1116-1127.