The Analysis of a Simple k-Means Clustering Algorithm

Tapas Kanungo* David M. Mount† Nathan S. Netanyahu‡ Christine Piatko§
Ruth Silverman¶ Angela Y. Wu‖

ABSTRACT

K-means clustering is a very popular clustering technique, which is used in numerous applications. Given a set of n data points in $\mathbf{R}^d$ and an integer k, the problem is to determine a set of k points $\mathbf{R}^d$, called *centers*, so as to minimize the mean squared distance from each data point to its nearest center. A popular heuristic for k-means clustering is Lloyd's algorithm. In this paper we present a simple and efficient implementation of Lloyd's k-means clustering algorithm, which we call the filtering algorithm. This algorithm is very easy to implement. It differs from most other approaches in that it precomputes a kd-tree data structure for the data points rather than the center points. We establish the practical efficiency of the filtering algorithm in two ways. First, we present a data-sensitive analysis of the algorithm's running time. Second, we have implemented the algorithm and performed a number of empirical studies, both on synthetically generated data and on real data from applications in color quantization, compression, and segmentation.

*Center for Automation Research, University of Maryland College Park, Maryland. Email: `kanungo@cfar.umd.edu`.

†Department of Computer Science, University of Maryland, College Park, Maryland. Email: `mount@cs.umd.edu`. This research was partially supported by the NSF under grant CCR–9712379.

‡Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel and Center for Automation Research, University of Maryland, College Park, Maryland 20742. Email: `nathan@macs.biu.ac.il`.

§The Johns Hopkins University Applied Physics Laboratory, Laurel, Maryland. Email: `christine.piatko@jhuapl.edu`.

¶Department of Computer Science, University of the District of Columbia, Washington, DC, and Center for Automation Research, University of Maryland, College Park, Maryland. Email: `ruth@cfar.umd.edu`.

‖Department of Computer Science and Information Systems, The American University, Washington, DC. Email: `awu@american.edu`.

1. INTRODUCTION

Clustering problems arise in many different applications, such as data mining and knowledge discovery [16], data compression and vector quantization [21], and pattern recognition and pattern classification [13]. One important class of clustering problems, sometimes called *k-means clustering*, is the following: Given a set of n data points in real d-dimensional space, $\mathbf{R}^d$, and an integer k, determine a set of k points in $\mathbf{R}^d$, called *centers*, so as to minimize the mean squared distance from each data point to its nearest center. This measure is often called the squared-error distortion [24, 21], and this type of clustering falls into the general category of variance-based clustering [23, 22].

The k-means problem is closely related to a number of other clustering problems, such as the Euclidean *k-medians* problem [2, 26], in which the objective is to minimize the sum of distances, and the geometric *k-center* problem [1], in which the objective is to minimize the maximum distance. Also see Capolyeas *et al.* [9] and Jain and Dubes [24] for other generalizations and issues in geometric clustering problems.

An asymptotically efficient approximation for the k-means clustering problem has been presented by Matoušek [31], but the very large constant factors involved with the construction suggest that it is not a good candidate for implementation. One of the most popular heuristics for solving the k-means problem is based on a simple iterative scheme for finding a locally minimal solution. This algorithm is often called the *k-means algorithm* [18, 28]. There are a number of variants of this algorithm, so to clarify which version we are using, we will refer to it as *Lloyd's algorithm.* (To be even more accurate, it would be better to call it the *generalized Lloyd's algorithm*, since Lloyd's original result was for scalar data [27].)

Here is how Lloyd's algorithm works. Given any set of k centers, for each center c_i, let V_i denote the set of data points for which c_i is the nearest neighbor. (Equivalently, V_i is the set of data points lying in the Voronoi cell of c_i relative to the set of centers.) For the next iteration of the algorithm, replace c_i with the centroid of V_i and update V_i accordingly. These two steps are repeated until some convergence conditions have been met. See Faber [15] for descriptions of other variants of this algorithm.

For points in general position, the algorithm will eventually converge to a point that is a local minimum. This is true because any local minimum for this problem corresponds to a centroidal Voronoi configuration (see [15, 12]). However, the result is not necessarily a global minimum. See [7, 30, 33, 35] for further discussion of its statistical and convergence properties. Because of its simplicity and flexibility it is a very popular algorithm and is widely used in statistical analysis, in spite of its apparent limitations. For example, it can be used in conjunction with a global algorithm for generating clusters as a post-processing stage to improve the quality of the final results.

One problem shared by all the various k-means algorithms is that they take a long time to run. There are two reasons for this. First, they are often applied in moderate to high dimensional spaces, and computing nearest neighbors efficiently in such spaces (without resorting to brute-force search) is not a trivial problem. The second reason is that often many iterations are needed until the algorithm's termination conditions are satisfied. (In fact it can be shown that when the algorithm is sufficiently close to the local minimum, the algorithm converges at a linear rate [12].) The most common way to improve the efficiency of Lloyd's algorithm is to compute nearest neighbors more efficiently, say by preprocessing the center points at the start of each stage into a data structure for answering nearest neighbor queries.

In this paper we present a simple and efficient implementation of Lloyd's algorithm, which we call the *filtering algorithm*. This algorithm was described briefly as part of a more complex kinetic-based k-means algorithm [25]. Unlike the kinetic algorithm, this algorithm is very easy to implement, and only requires that a kd-tree be built for the data points. We present the algorithm and the underlying data structures in Section 2. The filtering algorithm differs from most other approaches in that it precomputes a data structure for the data points, rather than the center points. Intuitively, this is a smart thing to do because (1) the data points do not vary throughout the computation, and hence this data structure does not need to be recomputed at each stage, and (2) there are typically many more data points than query points, and hence the relative advantages provided by preprocessing is greater. We should emphasize that this is not a different clustering method from Lloyd's algorithm, but rather a more efficient implementation of the algorithm.

We establish the practical efficiency of the filtering algorithm in two ways. First, we present a *data-sensitive* analysis of the algorithm's running time. The algorithm may perform as poorly as the simple brute-force algorithm in the worst case, but we claim that in the typical situations where the algorithm will be used (in particular where clustering is present in the data) the algorithm runs quite efficiently. Furthermore, the efficiency improves as the clusters become more isolated from each other. These results are presented in Section 3. Second, we have implemented the algorithm and performed a number of empirical studies, both on synthetically generated data and on real data from applications in color quantization, compression, and segmentation, as described in Section 4.

2. THE FILTERING ALGORITHM

In this section we describe the filtering algorithm. We begin with a review of the data structure that the algorithm uses. The structure is a *balanced box-decomposition tree* (or *BBD-tree*) for the point set. The BBD-tree [4] is a balanced variant of a number of well-known data structures based on hierarchical subdivision of space into rectilinear regions. Examples of this class of structures include point quadtrees [34] and variants [6], k-d trees [5], and the (unbalanced) box-decomposition tree (also called a fair-split tree) [8, 10, 36]. A related structure is the BAR tree [14], which partitions space into regions of bounded aspect ratio whose regions need not be rectilinear. For completeness we review the BBD-tree, emphasizing the elements that are important for this application. The special properties guaranteed by the BBD-tree are important for our theoretical analysis. In practice, the simpler kd-tree is usually good enough, and was used for our experiments.

We begin with a few definitions. By a *rectangle* in $\mathbf{R}^d$ we mean a d-fold product of closed intervals on the coordinate axes. The *size* of a rectangle is the length of its longest side. We define a *box* to be a rectangle such that the ratio of its longest to shortest side, called its *aspect ratio*, is bounded by some constant, which for concreteness we will take to be 2.

Each node of the BBD-tree is associated with a region of space called a cell. In particular, define a *cell* to be either a box or the set-theoretic difference of two boxes, one enclosed within the other. Thus each cell is defined by an *outer box* and an optional *inner box*. Each cell is associated with the set of data points lying within the cell. Cells are considered to be closed, and hence points which lie on the boundary between cells may be assigned to either cell. The *size* of a cell is the size of its outer box.

The following two lemmas summarize what we need to know about the BBD-tree and its properties. See [4] for proofs.

Lemma 1. *Given a set of n data points P in $\mathbf{R}^d$ and a bounding hypercube C for the points, in $O(dn \log n)$ time it is possible to construct a binary tree representing a hierarchical decomposition of C into cells of complexity $O(d)$ such that*

(i) The tree has $O(n)$ nodes and depth $O(\log n)$.

(ii) The cells have bounded aspect ratio, and with every $2d$ levels of descent in the tree, the sizes of the associated cells decrease by at least a factor of $1/2$.

Lemma 2. *(Packing Constraint) Consider any set $\mathcal{C}$ of cells of the BBD-tree with pairwise disjoint interiors, each of size at least s, that intersect a ball of radius r. The size of such a set is at most $\left(1 + \lceil \frac{4r}{s} \rceil\right)^d$.*

Now we describe how the filtering algorithm implements Lloyd's algorithm. Recall our objective. For each of the k centers, we need to compute the centroid of the set of data points for which this center is closest. After this, we move this center to this centroid, and proceed to the next stage. The algorithm begins by building a BBD-tree for the data points. This is done only once (not repeated at each stage) since the data points do not change throughout the computation. For each node u in the tree, we compute the centroid of its associated data points, which is then weighted (multiplied) by the number of such points. It is easy to modify the BBD-tree construction to compute this additional information in the same time bounds.

Next we consider the processing for each individual stage. For each node of the BBD-tree we will maintain a set of *candidate centers*. These are defined to be the center points that might serve as the nearest neighbor for some point lying within the associated cell. These candidates are computed as follows. The candidate centers for the root consist of all k centers. We then filter the candidates down using the following recursive pruning algorithm. For each node u let $C = \mathrm{cell}(u)$ denote its cell and let $Z = \mathrm{cand}(u)$ denote its set of candidates. We compute the candidate $z^* \in Z$ that is closest to the center of C. Then for each candidate $z \in Z - \{z^*\}$, if no point of C is closer to z than it is to z^*, we can infer that z is not the nearest center to any data point associated with u, and hence we can eliminate z from the list of candidates. If u is associated with a single candidate (which must be z^*) then z^* is the nearest neighbor of all its data points. We can assign them to z^* by adding the associated weighted centroid to an accumulator associated with z^*. Otherwise, if u is an internal node, we recurse on its children. If u is a leaf node, we compute the distances from its associated data point to all the candidates in Z, and assign the data point to its nearest center. To aid in the process, we make use of the following utility procedure, which can be computed easily in $O(d)$ time. Given points z^*, z and cell C, closer(z^*, z, C) returns false if if any part of C is closer to z than to z^*.

```
Filter(u, Z) {
    C = cell(u);
    z* = the closest point in Z to C's center;
    for each z ∈ Z − {z*} {
        if closer(z*, z, C) { Z −= {z}; }
        if ((|Z| == 1) ∨ (u is a leaf))
            z*.accum += u.centroid;
        else {
            Filter(u.left, Z);
            Filter(u.right, Z);
        }
    }
}
```

Figure 1: Filtering Algorithm.

Figure 1 presents a more detailed description of the recursive procedure, called Filter(u, Z). Here u is the current node and Z are the candidates passed in from u's parent. The initial call is to Filter(r, Z_0), where r is the root of the tree and Z_0 is the set of all centers.

3. DATA SENSITIVE ANALYSIS

In this section we present an analysis of the time spent in each stage of the filtering algorithm. Traditional worst-case analysis is not appropriate here, since it is not difficult to concoct scenarios in which virtually all of the nodes in the BBD-tree are visited, and in which every node is associated with all k center points as candidates. Thus the running time is $O(kn)$ in the worst case, which is no better than the simple brute-force algorithm.

The intuition behind our analysis is based on the observation that a great deal of the running time of Lloyd's algorithm is spent in the later stages of the algorithm, when the center points are close to their final locations, but the algorithm has not yet converged. Because of the method's relatively slow convergence rate, a significant fraction of the method's overall running time is spent in this mode. Our analysis will be based on the assumption that the data set can indeed be clustered into k natural clusters, and that the current centers are located close to the true cluster centers. These are admitedly strong assumptions, but our experimental results will bear out that the algorithm is quite efficient even when these assumptions are not met.

We say that a node is *visited* if the filter procedure is invoked on it. An internal node is *expanded* if its children are visited. A nonexpanded internal node or a leaf node is a *terminal node*. Under what circumstances is a node a terminal node? Consider a cell of the BBD-tree and the set of candidate centers that are associated with this cell. Let z^* denote the closest candidate to the center of the cell. If no point of the cell is closer to another candidate center than it is to z^*, then the algorithm can infer that all the data points contained within the cell can be assigned to z^*, and hence it will be a terminal node. If this condition is never satisfied, then the decomposition continues until the leaves are reached, at which point the process will certainly terminate. Thus, a nonleaf node whose cell intersects the Voronoi diagram cannot be a terminal node.

For example, in Fig. 2 the node with cell a is terminal because it lies entirely within the Voronoi cell of its nearest center. Cell b is terminal because it is a leaf. However, cell c is not terminal because it is not a leaf, and it intersects the Voronoi diagram. Observe that in this rather typical example if the clusters are well-separated, or *isolated*, and the center points are close to the true cluster centers, then a relatively small fraction of the data set lies near the Voronoi diagram of the centers. The more isolated the clusters are, the smaller this fraction will be, and hence fewer cells of the search structure will need to be visited by the algorithm. Our analysis will show that in such circumstances, the filtering search algorithm performs very well.

In our analysis we model this sort of situation as follows. We assume that the data points are generated independently from k different multivariate distributions, i.e, probability distributions in $\mathbf{R}^d$. Consider any one such distribution. Let $\mathbf{X} = (x_1, x_2, \ldots, x_d)$ denote a random vector from this distribution.

Let $\mu \in \mathbf{R}^d$ denote the mean point of this distribution and let Σ denote the $d \times d$ *covariance matrix* for the distribution [20], that is

$$\Sigma = E((\mathbf{X} - \mu)(\mathbf{X} - \mu)^T).$$

Observe that the diagonal elements of Σ are the variances of the random variables that are associated, respectively, with the individual coordinates. Let $\mathrm{tr}(\Sigma)$ denote the *trace* of Σ,

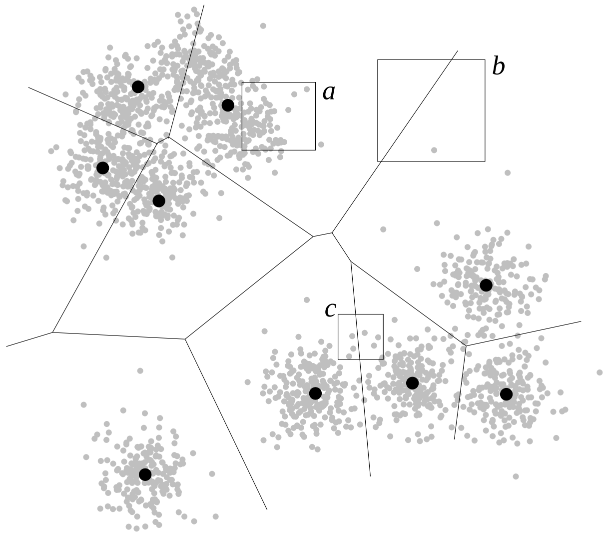

Figure 2: Filtering algorithm analysis. Note that the density of points near the edges of the Voronoi diagram is relatively low.

that is, the sum of its diagonal elements. We will measure the *dispersion* of the distribution by defining $\sigma = \sqrt{\mathrm{tr}(\Sigma)}$. This provides a generalization of the notion of standard deviation for univariate distributions.

We will characterize the degree of isolation of the clusters by two parameters. Let $\mu^{(i)}$ and $\sigma^{(i)}$ denote the mean and dispersion of the ith cluster distribution. Let

$$r_{\min} = \frac{1}{2}\min_{i\neq j}|\mu^{(i)} - \mu^{(j)}| \qquad \text{and} \qquad \sigma_{\max} = \max_i \sigma^{(i)},$$

where $|q-p|$ denotes the Euclidean distance between points q and p. The former quantity is half the minimum distance between any two cluster centers, and the latter is the maximum dispersion. Intuitively, isolated clusters of points should have a large value of $r_{\min}$ and a small value of $\sigma_{\max}$. We define the *cluster isolation* of the point distribution to be

$$\rho = \frac{r_{\min}}{\sigma_{\max}}.$$

This measure is similar to other known measures of separatedness or isolation. For example, Coggins and Jain [11] defined a measure of isolation for a single cluster i as the ratio $\min_{j\neq i}|\mu^{(i)} - \mu^{(j)}|/\sigma^{(i)}$.

We will show that, assuming that the candidate centers are relatively close to the cluster means, $\mu^{(i)}$, then as ρ increases (as clusters are more isolated) the algorithm's running time improves. Our proof makes use of the following straightforward generalization of Chebyshev's inequality (see [17]) to multivariate distributions. The proof has been omitted from this version.

Lemma 3. *Let* $\mathbf{X}$ *be a random vector in* $\mathbf{R}^d$ *drawn from a distribution with mean* μ *and dispersion* σ *(the square root of the trace of the covariance matrix). For all positive* t

$$\Pr(|\mathbf{X} - \mu| > t\sigma) \le \frac{d}{t^2}.$$

For $\delta \ge 0$, we say that a set of candidates is δ*-close* with respect to a given set of clusters, if for each center $c^{(i)}$ there is an associated cluster mean $\mu^{(i)}$ within distance at most $\delta r_{\min}$ and vice versa. Here is the main result of this section.

Theorem 1. *Consider a set of* n *points in* $\mathbf{R}^d$ *drawn from a collection of cluster distributions with cluster isolation* $\rho = r_{\min}/\sigma_{\max}$*, and consider a set of* k *candidate centers that are* δ*-close to the cluster means, for some* $\delta < 1$*. Then for any* $0 < \epsilon < 1-\delta$ *and for some constant* c*, the expected number of nodes visited by the filtering algorithm is*

$$O\left(k\left(\frac{c\sqrt{d}}{\epsilon}\right)^d + 2^d k \log n + \frac{dn}{\rho^2(1-\epsilon)^2}\right).$$

Before giving the proof let us make a few observations about this result. The result bounds the number of nodes visited. The time needed to process each node is proportional to the number of candidates for the node, which is at most k. Thus the total running time of the algorithm is larger by at most a factor of k. (Our experiments indicate that the average number of candidates per node is typically a small constant, though.) Also, the total running time includes an additive contribution of $O(dn\log n)$ time needed to build the initial BBD-tree.

We note that for fixed d, ρ, and ϵ (bounded away from 0 and $1-\delta$), the number of nodes visited as a function of n is $O(n)$ (since the last term in the analysis dominates). Thus the overall running time is $O(kn)$, which seems to be no better than the brute-force algorithm. The important observation is that as clustering isolation (ρ) increases, the running time is $O(kn/\rho^2)$, and thus the algorithm's efficiency increases rapidly as cluster isolation increases. In Section 4 we will see that the algorithm is quite efficient, even if cluster isolation is not particularly strong.

Proof. Consider the ith cluster distribution. Let $c^{(i)}$ denote a candidate center that is within distance $\delta r_{\min}$ from its mean $\mu^{(i)}$. Let $\sigma^{(i)}$ denote its dispersion. Let $B^{(i)}$ denote a ball of radius $r_{\min}$ centered at $\mu^{(i)}$. Observe that because no cluster centers are closer than $2r_{\min}$, these balls have pairwise disjoint interiors. Let $b^{(i)}$ denote a ball of radius $r_{\min}(1-\epsilon-\delta)$ centered at $c^{(i)}$. Because $c^{(i)}$ is δ-close to μ_i, $b^{(i)}$ is contained within a ball of radius $r_{\min}(1-\epsilon)$ centered at $\mu^{(i)}$, and hence is contained within $B^{(i)}$. Moreover, the distance between the boundaries of $B^{(i)}$ and $b^{(i)}$ is at least $\epsilon r_{\min}$.

Consider a visited node and let C denote its associated cell in the BBD-tree. We consider two cases. First, suppose that for some distribution i, $C \cap b_i \neq \emptyset$. We call C a *close node.* Otherwise, if the cell does not intersect any of the b_i balls it is a *distant node.*

First we bound the number of distant visited nodes. Consider the subtree of the BBD-tree induced by these nodes. If a node is distant, then its children are both distant, implying that this induced subtree is a full binary tree (that is, each nonleaf node has exactly two children). It follows that the total number of nodes in the induced subtree is not more than twice the number of leaves in the tree, which we shall bound next.

The data points associated with all the leaves of the induced subtree cannot exceed the number of data points that lie outside of b. As observed above, for any cluster distribution i, a data point of this distribution that lies outside of $b^{(i)}$ is at distance from $\mu^{(i)}$ of at least

$$r_{\min}(1-\epsilon) \;=\; \frac{r_{\min}}{\sigma^{(i)}}(1-\epsilon)\sigma^{(i)} \;\geq\; \rho(1-\epsilon)\sigma^{(i)}.$$

By Lemma 3 the probability of this occuring is at most $d/(\rho(1-\epsilon))^2$. Thus, the expected number of such data points is at most $dn/(\rho(1-\epsilon))^2$. Since each leaf of the BBD-tree contains at least one data point, and each point can be associated with at most one leaf, the number of leaves in this induced subtree is bounded by the same quantity, and by the above observation, the total number of distant nodes is at most twice as large.

Next we bound the number of close visited nodes. A visited node is said to be *expanded* if the algorithm visits its children. Clearly the number of close visited nodes is proportional to the number of close expanded nodes. For each cluster distribution i, consider the leaves of the induced subtree consisting of close expanded nodes that intersect $b^{(i)}$. The total number of close expanded nodes will be larger by a factor of k. To bound the number of such nodes, we further classify them by the size of the associated cell. An expanded node v whose size is at least $4r_{\min}$ is *large* and otherwise it is *small*. We will show that the number of large expanded nodes is bounded by $O(2^d \log n)$ and the number of small expanded nodes is bounded by $O((c\sqrt{d}/\epsilon)^d)$, for some constant c.

We first show the bound on the number of large expanded nodes. In the descent through the BBD-tree, the sizes of the nodes decrease monotonically. Consider the set of all expanded nodes of size greater than $4r_{\min}$. These nodes induce a subtree in the BBD-tree. Let L denote the leaves of this tree. The cells associated with the elements of L have pairwise disjoint interiors and they intersect $b^{(i)}$ and hence they intersect $B^{(i)}$. It follows from Lemma 2 (applied to $B^{(i)}$ and the cells associated with L) that there are at most $(1+\lceil 4r_{\min}/(4r_{\min})\rceil)^d = 2^d$ such cells. By Lemma 1 the depth of the tree is $O(\log n)$, and hence the total number of expanded large nodes is $O(2^d \log n)$, as desired.

Next we consider the small expanded nodes. We assert that the diameter of the cell corresponding to any such node is at least $\epsilon r_{\min}$. If not, then this cell would lie entirely within a disk of radius $r_{\min}(1-\delta)$ of $c^{(i)}$. Since the cell was expanded, we know that there must be a point in this cell that is closer to some other center c_j. This implies that the distance between $c^{(i)}$ and $c^{(j)}$ is less than $2r_{\min}(1-\delta)$. Since the candidates are δ-close, it follows that there are two cluster means $\mu^{(i)}$ and $\mu^{(j)}$ that are closer than $2r_{\min}(1-\delta)+2\delta r_{\min} = 2r_{\min}$. However, this would contradict the definition of $r_{\min}$.

Armed with this observation, we apply an argument similar to the one used by Arya and Mount [3] to bound the complexity of approximate range searching. A cell in dimension d of diameter x has longest side length of at least $x/\sqrt{d}$. Thus the size of each such cell is at least $\epsilon r_{\min}/\sqrt{d}$. It suffices to count the number of expanded nodes of sizes from $4r_{\min}$ down to $\epsilon r_{\min}/\sqrt{d}$. To do this we partition nodes into groups according to their size. For $i \geq 0$, define *size group* i to be the set of nodes whose cell size is in the interval $(1/2^{i+1}, 1/2^i]$. Since small nodes are of size less than $4r_{\min}$, the first size group of interest is $a+1$, where $1/2^{a+1} < 4r_{\min} \leq 1/2^a$, and hence $a = \lfloor -\lg 4r_{\min} \rfloor$. Since nodes that are smaller than $\epsilon r_{\min}/\sqrt{d}$ are not expanded, the last size group of interest is b, where $1/2^{b+1} < \epsilon r_{\min}/\sqrt{d} \leq 1/2^b$, and hence $b = \left\lfloor -\lg(\epsilon r_{\min}/\sqrt{d}) \right\rfloor$. Because a node and its child may have the same size, we cannot apply the packing lemma directly to each size group. Define the *base group* for the ith size group to be the subset of nodes in the size group that are leaves or whose children are both in the next smaller size group. The cells corresponding to the nodes in a base group have pairwise disjoint interiors, since none of their descendants can be in the same base group. Applying Lemma 2, it follows that the number of nodes in the ith base group is at most

$$\left(1+\left\lceil \frac{4r_{\min}}{1/2^i} \right\rceil\right)^d = \left(1+\left\lceil r_{\min} 2^{i+2} \right\rceil\right)^d.$$

From claim (ii) of Lemma 1 we know that at most $2d$ levels of ancestors above the base group can be in the same size group, and thus the number of nodes in any size group is at most $2d$ times the above quantity.

Thus, the total number of expanded nodes over all of the base groups is

$$E_d(r_{\min},\epsilon) \leq \sum_{i=a+1}^{b} \left(1+\left\lceil r_{\min} 2^{i+2} \right\rceil\right)^d.$$

Solving this geometric series, we have

$$E_d(r_{\min},\epsilon) \;\leq\; \left(\frac{c\sqrt{d}}{\epsilon}\right)^d,$$

for some constant c. This completes the analysis of the number of small expanded close nodes. Summing this bound over all k clusters provides the desired result.

□

4. EMPIRICAL ANALYSIS

To establish the practical efficiency of the filtering algorithm we implemented it and tested it on a number of data sets, both synthetically generated and from real applications. We implemented a simpler variant of the BBD-tree, which constructs a kd-tree using the sliding midpoint rule [29]. This decomposition method was chosen because our studies have shown that it performs better than the standard kd-tree decomposition rule for clustered data sets. Data points were chosen from a clustered Gaussian distribution. In particular, a given number of cluster centers were sampled from a uniform distribution over the hypercube $[-1,1]^d$. A Gaussian distribution was generated around each center, in which each coordinate was generated independently from a univariate Gaussian with a given standard deviation.

We wanted to compare our algorithm against existing implementations of k-means algorithms, and Lloyd's algorithm in particular. We found this difficult to do for a number of reasons. Many algorithms come as part of a large software package for data analysis, and it is not easy to modify the software to add the desired instrumentation. Also many of these packages do not provide the user with the ability to control termination conditions or to determine the initial placements of centers. Because of the slow convergence of k-means, many implementations seek to speed up the process or improve its clustering performance for the application of interest by implementing an algorithm that is different from Lloyd's algorithm. This might involve modifying the metric, by computing approximate nearest neighbors, or by applying different update strategies. Since the algorithm's behavior is very sensitive to the placement of the centers, any deviation from Lloyd's algorithm could result in significantly different performance on a given data set.

Lloyd's algorithm involves two basic components, computing nearest neighbors for the data points relative to the centers, and moving the center points to the centroids of their neighborhoods. Other than ours, all implementations that we have found that are true to this method differ essentially only in how the nearest neighbors are computed. For our experiments, we implemented two common methods for computing nearest neighbors. The first is by simple *brute force*, computing the distance from each data point to every center. The second is by building a nearest neighbor search structure for the centers. We did this by computing a kd-tree for the centers [19], which we call the *kd-center* algorithm. This was done using the ANN library [32], using the same decomposition method mentioned above. Our studies compared these two methods to the filtering algorithm. We performed two sets of experiments, one involving synthetic data and the other data derived from applications in image segmentation and compression.

Each experiment involved running all three algorithms: the brute-force algorithm, the kd-center algorithm, and our filtering algorithm. Even for the same data set, different starting configurations of centers result in different numbers of stages. However, given the same data set and the same starting centers, the number of stages is identical for all three algorithms. Because we were interested in comparing the relative running times of the three algorithms, we chose to factor out the number of stages, by dividing the total time by the number of stages. For the brute-force and kd-center algorithms, this is quite reasonable since the processing at each stage is virtually identical. However, this introduces a bias for the filtering algorithm. Since the total running time includes the one-time cost of building the kd-tree for the data points, by averaging over the stages, we effectively amortize this one-time cost over the stages. If the algorithm converges after only a small number of stages, then the cost per stage is artificially increased as a result.

We measured running time in two different ways. We measured both the CPU time (using the standard `clock()` function), and a second quantity called the number of *node-candidate pairs*. The latter quantity is a machine-independent statistic of the algorithm's complexity. Intuitively, it measures the number of interactions between a node of the kd-tree (or data point in the case of brute force) and a candidate center. For the brute-force algorithm, this quantity is always kn. For the kd-center algorithm, for each data point we count the number of nodes that were accessed in the kd-tree of the centers for computing its nearest center and sum this over all data points. For the filtering algorithm, we computed a sum of the number of candidates associated with every visited node of the tree. In particular, for each call to Filter(u, Z), the cardinality of the set Z of candidate centers is accumulated. The one-time cost of building the kd-tree is not included in this measure (and hence the amortization issue discussed above is not a problem here).

Note that the three algorithms are functionally equivalent. They all produce the same final result, and so there is no issue regarding the quality of the final output. The issue of interest for us is the efficiency of the computation.

4.1 Synthetic Data

We ran three experiments to determine the variations in running time as a function of cluster isolation and data set size. The first two experiments were designed to test the validity of Theorem 1. We generated $n = 10,000$ data points in $\mathbf{R}^3$ distributed evenly among 50 clusters from a clustered Gaussian distribution. The standard deviation varied from 0.01 (very well isolated) up to 0.7 (virtually no isolation). Because the same distribution is used for cluster centers throughout, the expected distances between cluster centers will be constant throughout. Thus the expected value of the cluster isolation parameter ρ varies inversely with the standard deviation. For the first experiment we chose $k = 50$ centers for each run and for the second we chose $k = 20$. The initial centers were chosen by taking a random sample of data points.

For each standard deviation we ran each of the algorithms (brute-force, kd-center, and filter) three times. For all runs the same data points were used. For each of the three runs a new set of initial center points was generated, and all three algorithms were run using the same data and initial center points. The algorithm ran for a maximum of 30 stages or until convergence. The reason that we felt it safe to limit the number of stages is that the running times tend to remain constant after the first few stages.

The average CPU time per stage and for all three methods are shown in Fig. 3, for $k = 50$ and $k = 20$. As mentioned above, if the algorithm runs for only a small number of stages, a significant bias is introduced into the filtering algorithm to account for the time needed to build the initial kd-tree. This bias was quite noticeable for highly clustered instances (small standard deviation). In both graphs we show the average running time for the filtering algorithm, both with and without the initialization bias included. For the purposes of comparison with the results of Theorem 1 the times without initialization are the more relevant.

The results of these experiments show that the filtering algorithm runs significantly faster than the other two algorithms. Ignoring the initialization bias, its running time improves when the clusters are more isolated. These experiments also show that as the standard deviation decreases (ρ increases) the running time for the filtering algorithm decreases rapidly.

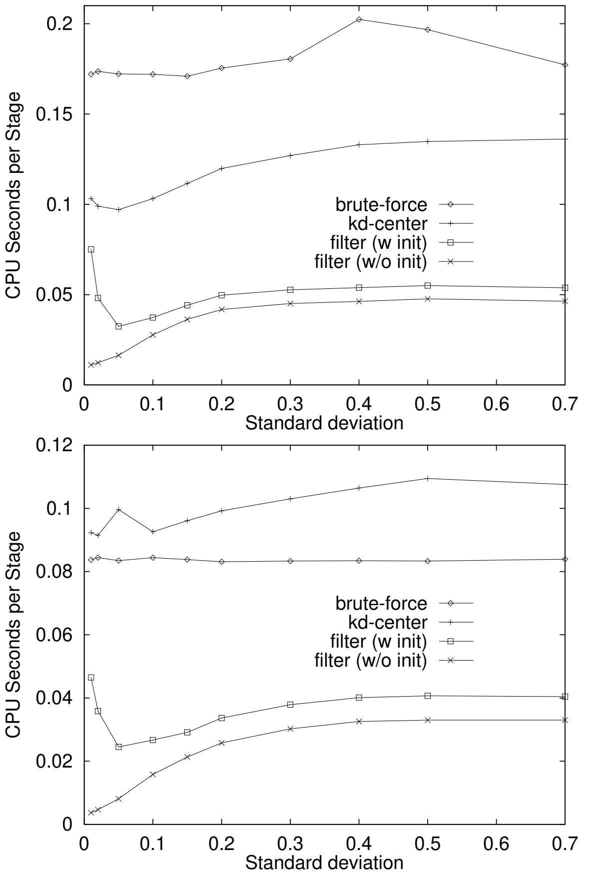

Figure 3: Average CPU time per stage versus cluster standard deviation, for $n = 10,000$ and $k = 50$ (above) and $k = 20$ (below).

The objective of the third experiment was to study the effects of data size on the running time. We generated data points of various sizes in a clustered Gaussian distribution, again with 50 clusters and with $k = 50$. The standard deviation was fixed at 0.10. The data size varied from $n = 1,000$ to $n = 20,000$. Again, the searches were terminated when stability was achieved or after 30 stages, and in each case we made three runs with different starting centers and averaged the results. The CPU times per stage of all three methods are shown in Fig. 4.

The results of the third experiment show that for fixed k and large n, all three algorithms have running times that vary linearly with n, with the filtering algorithm doing the best.

4.2 Real Data

To understand the relative efficiency of this algorithm under more practical circumstances, we ran a number of experiments on data sets derived from actual applications in image processing and compression. Each experiment involved solving a k-means problem on a set of points in $\mathbf{R}^d$ for various d's. In each case we applied all three algorithms: brute force (Brute), kd-center (KDCen), and filtering (Filter). In

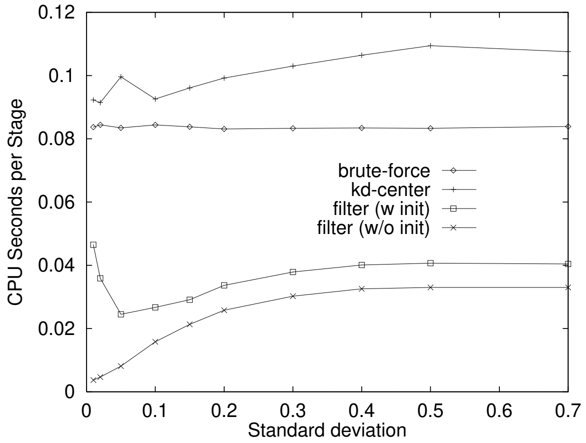

Figure 4: CPU time versus data size, for $k = 50$ and $\sigma = 0.10$.

each case we computed the average CPU time per stage in seconds (T-) and the average number of node-candidate pairs (NC-). This was repeated for values of k chosen from $\{8, 64, 256\}$. The results are shown in Table 1.

The experiments are grouped into three general categories. The first involve a color quantization application. A color image is given and a number of pixels are sampled from the image (10,000 for this experiment). Each pixel is represented as a triple consisting of red, green and blue components, each in the range $[0, 255]$, and hence is a point in $\mathbf{R}^3$. The input images were chosen from a number standard images for this application. The results are shown in the upper half of Table 1, under the names "balls," "kiss," ... , "ball1_s." Some of the images are shown in Figure 5.

The next experiment involved a vector quantization application for data compression. Here the images are gray-scale images with pixel values in the range $[0, 255]$. Each 2×2 subarray of pixels is selected and mapped to a 4-element vector, and the k-means algorithm is run on the resulting set of vectors. The results are shown in Table 1 under the names "couple,"... , "woman2." Some of the images are shown in Figure 5.

The final experiment involved an image segmentation problem. A 512×512 LandSat image of Israel consisting of 7 spectral bands was used. The resulting 7-element vectors in the range $[0, 255]$ were presented to the algorithms. The results are shown in Table 1 under the name "Israel." The image is shown in Figure 5.

An inspection of the results shows that the filtering algorithm significantly outperformed the other two algorithms in all cases. Plots of the underlying point distributions showed that most of these data sets were really not well clustered. Thus the filtering algorithm is quite efficient, even when the conditions of Theorem 1 are not satisfied.

5. CONCLUDING REMARKS

We have presented an efficient implementation of Lloyd's k-means clustering algorithm, called the filtering algorithm.

Image	Dim	Size	k	T-Brute	T-KDCen	T-Filter	NC-Brute	NC-KDCen	NC-Filter
balls	3	10000	8	0.0542857	0.0804762	0.0157143	80	68.13	3.678
			64	0.20875	0.1515	0.01875	640	160.9	17.65
			256	0.742424	0.17875	0.0375758	2560	180	49.77
kiss	3	10000	8	0.053	0.08225	0.0165	80	71.92	10.07
			64	0.209	0.14875	0.0435	640	170.4	41.76
			256	0.7455	0.218	0.089	2560	243.9	111
Lena	3	10000	8	0.0538889	0.0797222	0.0163889	80	65.18	9.767
			64	0.20925	0.14575	0.046	640	164.7	43.72
			256	0.75025	0.218	0.0915	2560	236.8	118.2
ball1_h	3	10000	8	0.0535	0.0755	0.0095	80	57.27	3.859
			64	0.207	0.119118	0.0245	640	117.8	20.98
			256	0.749231	0.16575	0.0584615	2560	167.2	69.42
ball1_3	3	10000	8	0.0535	0.07625	0.0095	80	58.88	4.073
			64	0.208571	0.11641	0.0251429	640	115.9	20.69
			256	0.748846	0.166364	0.0553846	2560	167.8	65.83
ball1_5	3	10000	8	0.0541667	0.08	0.0225	80	62.72	2.712
			64	0.214688	0.125278	0.029375	640	122.4	23.16
			256	0.74775	0.178	0.059	2560	175.5	71.89
ball1_p	3	10000	8	0.0535714	0.0778571	0.0207143	80	64.37	3.12
			64	0.2095	0.122	0.0285	640	122.2	23.02
			256	0.743043	0.17	0.0608333	2560	174.2	72.93
ball1_s	3	10000	8	0.053	0.0771429	0.015	80	57.95	3.497
			64	0.20875	0.123939	0.0275	640	123.5	24.04
			256	0.745	0.17225	0.059	2560	176	74.3
couple	4	65536	8	0.37475	0.5575	0.15825	524.3	482.3	74.6
			64	1.616	1.21925	0.3535	4194	1452	285.8
			256	5.501	1.732	0.658	1.678e+04	2375	739.6
crowd	4	65536	8	0.36875	0.57375	0.146	524.3	497.7	62.35
			64	1.60975	1.304	0.32175	4194	1512	244.9
			256	5.49975	1.716	0.535	1.678e+04	2347	567.3
lax	4	65536	8	0.36675	0.59225	0.18475	524.3	558.4	99.04
			64	1.6085	1.3725	0.42325	4194	1772	359.7
			256	5.495	1.86175	0.7055	1.678e+04	2776	790.6
lena	4	65536	8	0.373	0.55475	0.123	524.3	471.5	50.83
			64	1.615	1.19375	0.34475	4194	1339	266.9
			256	5.50475	1.649	0.618	1.678e+04	2200	677.1
man	4	65536	8	0.3685	0.5505	0.13925	524.3	465.9	63.16
			64	1.609	1.20275	0.35375	4194	1392	278.1
			256	5.49975	1.66125	0.60675	1.678e+04	2275	665.4
woman1	4	65536	8	0.370937	0.563125	0.176875	524.3	483.4	73.65
			64	1.612	1.28275	0.3805	4194	1500	304.2
			256	5.523	1.7945	0.64325	1.678e+04	2450	706.8
woman2	4	65536	8	0.37175	0.544	0.1125	524.3	433.3	35.46
			64	1.615	1.20075	0.30775	4194	1277	214.2
			256	5.51675	1.64675	0.519	1.678e+04	2133	531.4
Israel	7	262144	8	1.7945	2.739	1.26775	2097	2170	356.2
			64	8.0195	6.2145	4.78825	1.678e+04	7520	1581
			256	28.5027	9.76025	7.324	6.711e+04	1.367e+04	4434

Table 1: Running times for various test inputs.

balls kiss Lena ball1_h

couple lax woman2 Israel

Figure 5: Samples of some of the images used in our experiments.

The algorithm is easy to implement and only requires that a kd-tree be built once for the data points. Efficiency is achieved because the data points do not vary throughout the computation, and hence this data structure does not need to be recomputed at each stage, and there are typically many more data points than query points, and hence the relative advantage provided by preprocessing are greater. This algorithm differs from most other algorithms only in how nearest centers are computed, so it could be applied to the many of the variants of Lloyd's algorithm.

The algorithm has been implemented and the source code is available on request. We have demonstrated the practical efficiency of this algorithm both theoretically through a data sensitive analysis and empirically through experiments on both synthetically generated and real data sets. The data sensitive analysis shows that the more well-isolated are the clusters, the faster the algorithm runs. This is subject to the assumption that the center points are indeed close to the cluster centers. Although this assumption is rather strong, our empirical analysis on synthetic data indicates that the algorithm's running time does improve dramatically as cluster isolation increases. These results for both synthetic and real data sets indicate that the filtering algorithm is significantly more efficient than the other two methods that were tested.

A natural question is whether the filtering algorithm can be improved. The most obvious source of inefficiency in the algorithm is that it passes no information from one stage to the next. Presumably in the later stages of Lloyd's algorithm, as the centers are converging to their final positions, one would expect that the vast majority of the data points have the same closest center from one stage to the next. A good algorithm would exploit this coherence to improve running time. A kinetic method along these lines was proposed in [25], but this algorithm is quite complex, and does not provide significantly faster running time in practice. The development of a simple and practical algorithm which combines the best elements of the kinetic and filtering approaches would be a significant contribution.

6. REFERENCES

[1] P. K. Agarwal and C. M. Procopiuc. Exact and approximation algorithms for clustering. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 658–667, 1998.

[2] S. Arora, P. Raghavan, and S. Rao. Approximation schemes for Euclidean k-median and related problems. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 106–113, 1998.

[3] S. Arya and D. M. Mount. Approximate range searching. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 172–181, 1995.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *J. ACM*, 45:891–923, 1998.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, 1975.

[6] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.

[7] L. Bottou and Y. Bengio. Convergence properties of the k-means algorithms. In G. Tesauro and D. Touretzky, editors, *Advances in Neural Information Processing Systems 7*, pages 585–592. MIT Press, 1995.

[8] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.

[9] V. Capoyleas, Günter Rote, and G. Woeginger. Geometric clusterings. *J. Algorithms*, 12:341–356, 1991.

[10] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 226–232, 1983.

[11] J. M. Coggins and A. K. Jain. A spatial filtering approach to texture analysis. *Pattern Recognition Letters*, 3:195–203, 1985.

[12] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tesselations: Applications and algorithms. *SIAM Review*, 41:637–676, 1999.

[13] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, New York, NY, 1973.

[14] C. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 300–309, 1999.

[15] V. Faber. Clustering and the continuous k-means algorithm. *Los Alamos Science*, 22:138–144, 1994.

[16] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[17] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, New York, NY, 3rd edition, 1968.

[18] E. Forgey. Cluster analysis of multivariate data: Efficiency vs. interpretability of classification. *Biometrics*, 21:768, 1965.

[19] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3:209–226, 1977.

[20] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, Boston, MA, 1990.

[21] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic, Boston, MA, 1992.

[22] M. Inaba, H. Imai, and N. Katoh. Experimental results of a randomized clustering algorithm. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages C1–C2, 1996.

[23] M. Inaba, N. Katoh, and H. Imai. Applications of weighted Voronoi diagrams and randomization to variance-based k-clustering. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 332–339, 1994.

[24] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[25] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu. Computing nearest neighbors for moving points and applications to clustering. In *10th Ann. ACM-SIAM Symposium on Discrete Algorithms*, pages S931–S932, 1999.

[26] S. Kolliopoulos and S. Rao. A nearly linear-time approximation scheme for the Euclidean k-median problem. In *Proc. 7th Annual European Symp. on Algorithms*, 1999.

[27] S. P. Lloyd. Least squares quantization in PCM. *IEEE Trans. on Inf. Theory*, 28:129–137, 1982.

[28] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the Fifth Berkeley Symposium on Math. Stat. and Prob.*, volume 1, pages 281–296, 1967.

[29] S. Maneewongvatana and D. M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In *ALENEX*, 1999.

[30] O. L. Mangasarian. Mathematical programming in data mining. *Data Mining and Knowledge Discovery*, 1, 1997.

[31] J. Matoušek. On approximate geometric k-clustering. Manuscript, 1999.

[32] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. Center for Geometric Computing 2nd Annual Fall Workshop on Computational Geometry, URL: `http://www.cs.umd.edu/~mount/ANN.`, 1997.

[33] D. Pollard. A centeral limit theorem for k-means clustering. *Annals of Probability*, 10:919–926, 1982.

[34] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.

[35] S. Z. Selim and M. A. Ismail. K-means-type algorithms: a generalized convergence theorem and characterization of local optimality. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 6:81–87, 1984.

[36] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.