Accelerating Exact k-means Algorithms with Geometric Reasoning

Dan Pelleg and Andrew Moore
School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213, (dpelleg,awm)@cs.cmu.edu

Abstract

We present new algorithms for the k-means clustering problem. They use the kd-tree data structure to reduce the large number of nearest-neighbor queries issued by the traditional algorithm. Sufficient statistics are stored in the nodes of the kd-tree. Then, an analysis of the geometry of the current cluster centers results in great reduction of the work needed to update the centers. Our algorithms behave exactly as the traditional k-means algorithm. Proofs of correctness are included. The kd-tree can also be used to initialize the k-means starting centers efficiently. Our algorithms can be easily extended to provide fast ways of computing the error of a given cluster assignment, regardless of the method in which those clusters were obtained. We also show how to use them in a setting which allows approximate clustering results, with the benefit of running faster.

We have implemented and tested our algorithms on both real and simulated data. Results show a speedup factor of up to 170 on real astrophysical data, and superiority over the naive algorithm on simulated data in up to 5 dimensions. Our algorithms scale well with respect to the number of points and number of centers, allowing for clustering with tens of thousands of centers.

1 Introduction

Consider a dataset with R records, each having M attributes. Given a constant k, the *clustering* problem is to partition the data into k subsets such that each subset behaves "well" under some measure. For example, we might want to minimize the squared Euclidean distances between points in any subset and their center of mass. The *k-means* algorithm for clustering finds a local optimum of this measure by keeping track of centroids of the subsets, and issuing a large number of nearest-neighbor queries [7].

A *kd-tree* is a data structure for storing a finite set of points from a finite-dimensional space [1]. Recently, Moore has shown its usage in very fast EM-based Mixture Model Clustering [9]. The need for such a fast algorithm arises when conducting massive-scale model selection, and in datasets with a large number of attributes and records (see also [10]). An extreme example is the data which is gathered in the Sloan Digital Sky Survey (SDSS) [12], where M is about 500 and R is in the millions.

We show that kd-trees can be used to reduce the number of nearest-neighbor queries in k-means by using the fact that their nodes can represent a large number of points. We are frequently able to prove for certain nodes of the kd-tree statements of the form "any point associated with this node must have X as its nearest neighbor" for some center X. This, together with a set of statistics stored in the kd-nodes, allows for great reduction in the number of arithmetic operations needed to update the centroids of the clusters.

We have implemented our algorithms and tested their behavior with respect to variations in the number of points, dimensions, and centers, as measured on synthetic data. We also present results of tests on preliminary SDSS data.

The remainder of this paper is organized as follows. In Section 2 we discuss related work, introduce notation and describe the naive k-means algorithm. In Section 3 we present our algorithms with proofs of correctness.

Section 4 discusses results of experiments on real and simulated data, and Section 5 concludes and suggests ideas for further work.

2 Definitions and Related Work

Throughout this paper, we denote the number of records by R, the number of dimensions by M and the number of centers by k.

We first describe the naive k-means algorithm for producing a clustering of the points in the input into k clusters [5, 2]. It partitions the data-points into k subsets such that all points in a given subset "belong" to some center. The algorithm keeps track of the centroids of the subsets, and proceeds in iterations. We denote the set of centroids after the i-th iteration by $C^{(i)}$. Before the first iteration the centroids are initialized to random values. The algorithm terminates when $C^{(i)}$ and $C^{(i-1)}$ are identical. In each iteration, the following is performed:

1. For each point x, find the center in $C^{(i)}$ which is closest to x. Associate x with this center.
2. Compute $C^{(i+1)}$ by taking, for each center, the center of mass of points associated with this center.

Our algorithms involve modification of just the code within one iteration. We therefore analyze the cost of a single iteration. Naive k-means performs a "nearest-neighbor" query for each of the R points. During such a query the distances in M-space to k centers are calculated. Therefore the cost is $O(kMR)$.

One fundamental tool we will use to tackle the problem is the kd-tree data-structure. A thorough discussion is out of the scope of this paper. We just outline its relevant properties, and from this point on will assume that a kd-tree for the input points exists. Further details about kd-trees can be found in [8]. We will use a specialized version of kd-trees called $mrkd$-trees, for "multi-resolution kd-trees" [4]. Their properties are:

- They are binary trees.
- Each node contains information about all points contained in a hyper-rectangle h. The hyper-rectangle is stored at the node as two M-length boundary vectors $h^{\max}$ and $h^{\min}$. At the node are also stored the number, center of mass, and sum of Euclidean norms, of all points within h. All children of the node represent hyper-rectangles which are contained in h.
- Each non-leaf node has a "split dimension" d and a "split value" v. Its children l (resp. r) represent the hyper-rectangles h_l (h_r), both within h, such that all points in h_l (h_r) have their d-th coordinate value smaller than (at least) v.
- The root node represents the hyper-rectangle which encompasses all of the points.
- Leaf nodes store the actual points.

For two points x, y we denote by $d(x, y)$ their Euclidean distance. For a point x and a hyper-rectangle h we define $\text{closest}(x, h)$ to be the point in h which is closest to x. Note that computing $\text{closest}(x, h)$ can be done in time $O(M)$ due to the following facts:

- If $x \in h$, then x is closest.
- Otherwise, $\text{closest}(x, h)$ is on the boundary of h. This boundary point can be found by clipping each coordinate of x, to lie within h, as shown in [8].

We define the distance $d(x, h)$ between a point x and a hyper-rectangle h to be $d(x, \text{closest}(x, h))$. For a hyper-rectangle h we denote by $\text{width}(h)$ the vector $h^{\max} - h^{\min}$.

Given a clustering ϕ, we denote by $\phi(x)$ the centroid this clustering associates with an arbitrary point x (so for k-means, $\phi(x)$ is simply the center closest to x). We then define a measure of quality for ϕ:

$$\text{distortion}_\phi = \frac{1}{R} \cdot \sum_x d^2(x, \phi(x)) \tag{1}$$

where R is the total number of points and x ranges over all input points.

The k-means algorithm is known to converge to a local minimum of the distortion measure. It is also known to be too slow for practical databases. Much of the related work does not attempt to confront the algorithmic issues directly. Instead, different methods of subsampling and approximation are proposed. A way to obtain a small "balanced" sample of points by sampling from the leaves of a $R*$ tree is shown in [6]. In [11], a simulated-annealing approach is suggested to direct the search in the space of possible partitions of the input points. A tree structure with sufficient statistics is presented in [13]. It is used to identify outliers and speed computations. However, the calculated clusters are approximations, and depend on many parameters.

Note that although the starting centers can be selected arbitrarily, k-means is fully deterministic, given the starting centers. A bad choice of initial centers can have a great impact on both performance and distortion. Bradley and Fayyad [3] discuss ways to refine the selection of starting centers through repeated sub-sampling and smoothing.

Originally, kd-trees were used to accelerate nearest-neighbor queries. We could, therefore, use them in the k-means inner loop transparently. For this method to work we will need to store the *centers* in the kd-tree. So whatever savings we achieve, they will be a function of the number of centers, and not of the (necessarily larger) number of points. The number of queries will remain R. Moreover, the centers move between iteration, and a naive implementation would have to rebuild the kd-tree whenever this happens. Our methods, in contrast, store the entire *dataset* in the kd-tree.

3 Algorithms

Our algorithms exploit the fact that instead of updating the centroids point by point, a more efficient approach is to update in bulk. This can be done using the known centers of mass and size of groups of points. Naturally, these groups will correspond to hyper-rectangles in the *kd*-tree. To ensure correctness, we must first make sure that all of the points in a given rectangle indeed "belong" to a specific center before adding their statistics to it. This gives rise to the notion of an *owner*.

Definition 1 *Given a set of centers C and a hyper-rectangle h, we define by $\mathit{owner}_C(h)$ a center $c \in C$ such that any point in h is closer to c than to any other center in C, if such a center exists.*

We will omit the subscript C where it is clear from the context. The rest of this section discusses owners and efficient ways to find them. We start by analyzing a property of owners, which, by listing those centers which do not have it, will help us eliminate non-owners from our set of possibilities. Note that $\mathrm{owner}_C(h)$ is not always defined. For example, when two centers are both *inside* a rectangle, then there exists no unique owner for this rectangle. Therefore the precondition of the following theorem is that there exists a unique owner. The algorithmic consequence is that our method will not always find an owner, and will sometimes be forced to descend the *kd*-tree, thereby splitting the hyper-rectangle in hope to find an owner for the smaller hyper-rectangle.

Theorem 2 *Let C be a set of centers, and h a hyper-rectangle. Let $c \in C$ be $\mathit{owner}_C(h)$. Then:*

$$d(c, h) = \min_{c' \in C} d(c', h) \ .$$

Proof: Assume, for the purpose of contradiction, that $c \neq \arg\min_{c' \in C} d(c', h) \equiv c'$. Then there exists a point in h (namely $\mathrm{closest}(c', h)$) which is closer to c'

than to c. A contradiction to the definition of c as owner(h). □

Equivalently, we can say that when looking for owner(h), we should only consider centers with shortest (as opposed to "minimal") distance $d(c,h)$. Suppose that two (or more) centers share the minimal distance to h. Then neither can claim to be an owner.

Theorem 2 narrows down the number of possible owners to either one (if there exists a shortest distance center) or zero (otherwise). In the latter case, our algorithm will proceed by splitting the hyper-rectangle. In the former case, we still have to check if this candidate is an owner of the hyper-rectangle in question. As will become clear from the following discussion, this will not always be the case. Let us begin by defining a restricted form of ownership, where just two centers are involved.

Definition 3 *Given a hyper-rectangle h, and two centers c^1 and c^2 such that $d(c^1,h) < d(c^2,h)$, we say that c^1* dominates *c^2 with respect to h if every point in h is closer to c^1 than it is to c^2.*

Observe that if some $c \in C$ dominates *all* other centers with respect to some h, then $c =$ owner(h). A possible (albeit inefficient) way of finding owner(h) if one exists would be to scan all possible pairs of centers. However, using theorem 2, we can reduce the number of pairs to scan since c^1 is fixed. To prove this approach feasible we need to show that the domination decision problem can be solved efficiently.

Lemma 4 *Given two centers c^1, c^2, and a hyper-rectangle h such that $d(c^1,h) < d(c^2,h)$, the decision problem "does c^1 dominate c^2 with respect to h?" can be answered in $O(M)$ time.*

Proof: Observe the decision line L composed of all points which are equidistant to c^1 and c^2 (see Figure 1). If c^1 and h are both fully contained in one half-space defined by L, then c^1 dominates c^2. The converse is also true; if there exists a point $x \in h$ such that it is not in the same half-space of L as c^1, then $d(c^1, x) > d(c^2, x)$ and c^1 does not dominate c^2. It is left to show that finding whether c^1 and h are contained in the same half-space of L can be done efficiently. Consider the vector $\vec{v} \equiv c^2 - c^1$. Let p be a point in h which maximizes the value of the inner product $\langle v, p \rangle$. This is the extreme point in h in the direction $\vec{v}$. Note that $\vec{v}$ is perpendicular to $\vec{L}$. If p is closer to c^1 than it is to c^2, then so is any point in h (p is the closest one can get to L, within h). If not, p is a proof that c^1 does not dominate c^2.

Furthermore, the linear program "maximize $\langle v, p \rangle$ such that $p \in h$" can be solved in time $O(M)$. Again we notice the extreme point is a corner of h. For each coordinate i, we choose p_i to be $h_i^{\max}$ if $c_i^2 > c_i^1$, and $h_i^{\min}$ otherwise. $\square$

3.1 The Simple Algorithm

We now describe a procedure to update the centroids in $C^{(i)}$. It will take into consideration an additional parameter, a hyper-rectangle h such that all points in h affect the new centroids. The procedure is recursive, with the initial value of h being the hyper-rectangle with all of the input points in it. If the procedure can find owner(h), it updates its counters using the center of mass and number of points which are stored in the kd-node corresponding to h (we will frequently interchange h with the corresponding kd-node). Otherwise, it splits h by recursively calling itself with the children of h. The proof of correctness follows from the discussion above.

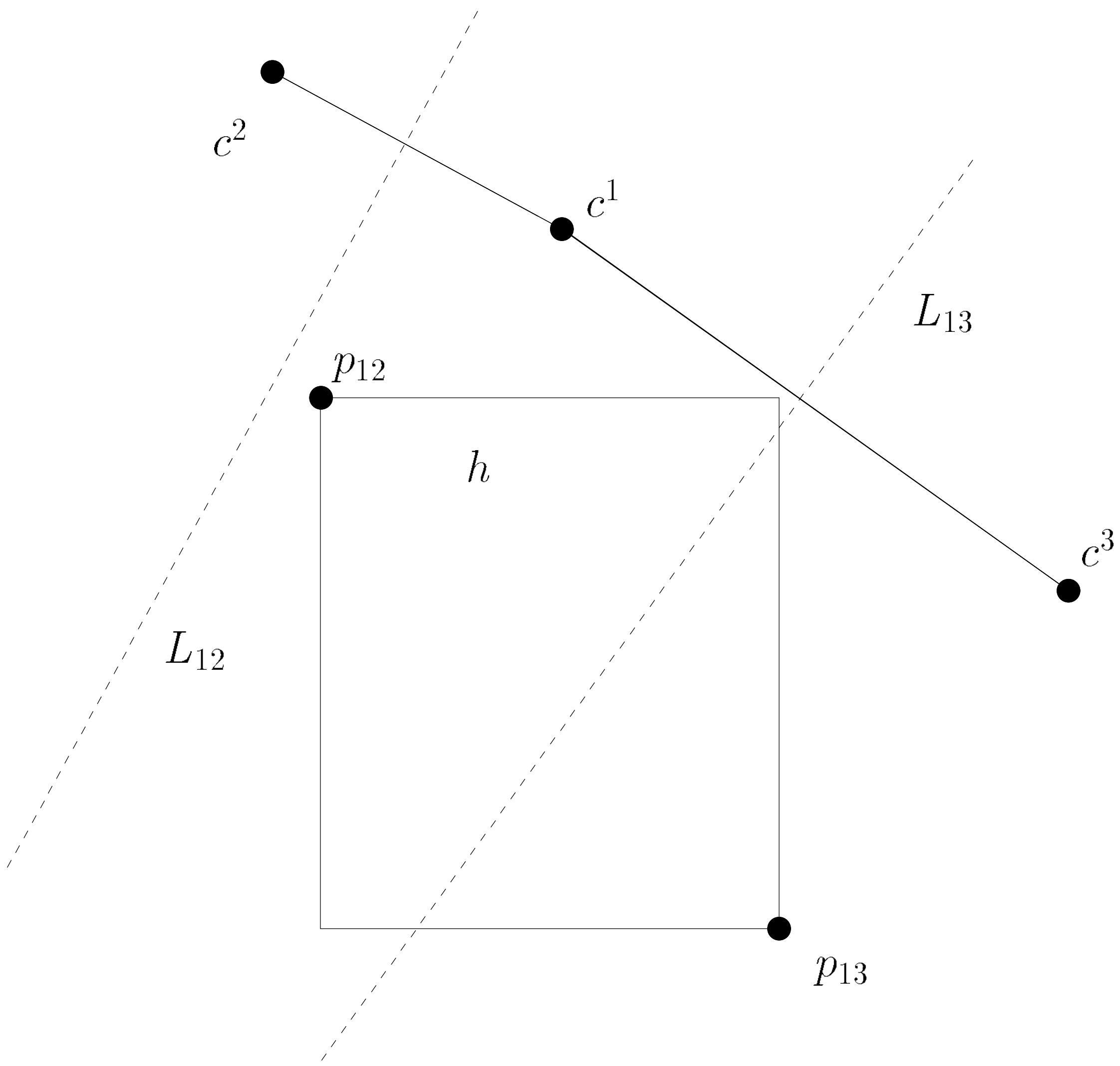

Figure 1: Domination with respect to a hyper-rectangle.
L_{12} is the decision line between centers c^1 and c^2. Similarly, L_{13} is the decision line between c^1 and c^3. p_{12} is the extreme point in h in the direction $c^2 - c^1$, and p_{13} is the extreme point in h in the direction $c^3 - c^1$. Since p_{12} is on the same side of L_{12} as c^1, c^1 dominates c^2 with respect to the hyper-rectangle h. Since p_{13} is *not* on the same side of L_{13} as c^1, c^1 does not dominate c^3.

Update(h, C):

1. If h is a leaf:
   (a) For each data point in h, find the closest center to it and update that center's counters.
   (b) Return.
2. Compute $d(c, h)$ for all centers c. If there exists one center c with shortest distance:

   If for all other centers c', c dominates c' with respect to h (so we have established $c = \text{owner}(h)$):

   (a) Update c's counters using the data in h.
   (b) Return.
3. Call Update(h_l, C).
4. Call Update(h_r, C).

We would not expect our Update procedure to prune in the case that h is the universal set of all input points (since all centers are contained in it, and therefore no shortest-distance center exists). We also notice that if the hyper-rectangles were split again and again so that the procedure is dealing just with leaves, this method would be identical to the original k-means. In fact, this implementation will be much more expensive because of the redundant overhead. Therefore our hope is that large enough hyper-rectangles will be owned by a single center to make this approach worthwhile. See Figure 2 for a visualization of the procedure operation.

3.2 The "Blacklisting" Algorithm

Our next algorithm is a refinement of the simple algorithm. The idea is to identify those centers which will definitely *not* be owners of the hyper-rectangle h. If we can show this is true for some center c, there is no point

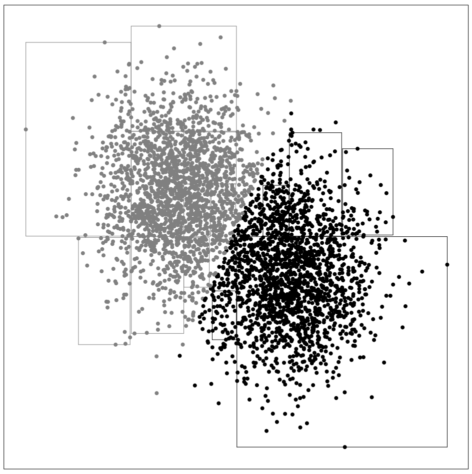

Figure 2: Visualization of the hyper-rectangles owned by centers. The entire two-dimensional dataset is drawn as points in the plane. All points that "belong" to a specific center are colored the same color (here, K=2). The rectangles for which it was possible to prove that belong to specific centers are also drawn. Points outside of rectangles had to be determined in the slow method (by scanning each center). Points within rectangles were not considered by the algorithm. Instead, their number and center of mass are stored together with the rectangle and are used to update the center coordinates.

in checking c for any of the descendants of h, hence the term "blacklisting". Let c^1 be a minimal-distance center to h, and let c^2 be any center such that $d(c^2, h) > d(c^1, h)$. If c^1 dominates c^2 with respect to h, we have two possibilities. One, that $c^1 = \text{owner}(h)$. This is the good case since we do not need any more computation. The other option is that we have no owner for this node. The slow algorithm would have given up at this point and restarted a computation for the children of h. The blacklisting version notices that c^1 dominates c^2 with respect to h' for any h' *contained* in h. This is true by definition. Now, since the descendants of h in the kd-tree are all contained in h, we can eliminate c^2 from the list of possible centers at this point for all descendants. Thus the list of prospective owners shrinks until it reaches a size of 1. At this point we declare the only remaining center the owner of the current node h. Again, we hope this happens before h is a leaf node, otherwise our overhead is wasted. For a typical run with 30000 points and 100 centers, the blacklisting algorithm calculates distances from points to centers about 270000 times each iteration. This, plus the overhead, is to be compared with the 3 million distances the naive algorithm has to calculate.

3.3 Efficiently Computing Goodness-Of-Fit Statistics

As an added bonus, the "ownership" property can help accelerate other computations. With the small price of storing, in each kd-node, the sum of the squared norms of all points of this node, one can use the exact same algorithm to compute the distortion measure defined in Equation 1. We omit the straightforward algebra. For other obtainable statistics see [13].

4 Experimental Results

We have conducted experiments on both real and randomly-generated data. The real data is preliminary SDSS data with some 400,000 celestial objects. The synthetic data covers a wide range of parameters that might affect the performance of the algorithms. Some of the measures are comparative, and measure the performance of our algorithms against the naive algorithm, and against the BIRCH [13] algorithm. Others simply test our fast algorithms' behavior on different inputs.

The real data is a two-dimensional data-set, containing the X and Y coordinates of objects observed by a telescope. There were 433,208 such objects in our data-set. Note that "clustering" in this domain has a well-known astrophysical meaning of clusters of galaxies, etc. Such clustering, however, is somewhat insignificant in a two-dimensional domain since the physical placement of the objects is in 3-space. The results are shown in Table 1. The main conclusion is that the blacklisting algorithm executes 25 to 176 times faster than the naive algorithm, depending on the number of points.

In addition, we have conducted experiments with the BIRCH algorithm [13]. It is similar to our approach in that it keeps a tree of nodes representing sets of data-points, together with sufficient statistics. The experiment was conducted as follows. We let BIRCH run through phases 1 through 4 and

points	blacklisting	naive	speedup
50000	2.02	52.22	25.9
100000	2.16	134.82	62.3
200000	2.97	223.84	75.3
300000	1.87	328.80	176.3
433208	3.41	465.24	136.6

Table 1: Comparative results on real data.

Run-times of the naive and blacklisting algorithm, in seconds per iteration. Run-times of the naive algorithms also shown as their ratio to the running time of the blacklisting algorithm, and as a function of number of points. Results were obtained on random samples from the 2-D "petro" file using 5000 centers.

terminate, measuring the total run-time. Then we ran our k-means for as many iterations as possible, given this time limit. We then measured the distortion of the clustering output by both algorithms. The results are in Table 2. In seven experiments out of ten, the blacklisting algorithm produced better (i.e., lower distortion) results. These experiments include randomly generated data files originally used as a test-case in [13], random data files generated by us, and real data.

The synthetic experiments were conducted in the following manner. First, a data-set was generated using 72 randomly-selected points (class centers). For each data-point, a class was first selected at random. Then, the point coordinates were chosen independently under a Gaussian distribution with mean at the class center, and deviation σ equal to the number of dimensions times 0.025. One data-set contained $20,000$ points drawn this way. The naive, "slow", and blacklisting algorithms were run on this data-set and measured for speed. The CPU time measured was then divided by the number of iterations to get a time-per-iteration estimate. Notice that all three algorithms generate exactly the same set of centroids at each iteration, so the number of iterations for all three of them is identical, given the data-set. This experiment was repeated 30 times and averages were taken. The number of

dataset	form	points	K	blacklisting distortion	BIRCH distortion	BIRCH, relative
1	grid	100000	100	1.85	1.76	0.95
2	sine	100000	100	2.44	1.99	0.82
3	random	100000	100	6.98	8.98	1.29
4	random	200000	250	7.94e-04	9.78e-04	1.23
5	random	200000	250	8.03e-04	1.01e-03	1.25
6	random	200000	250	7.91e-04	1.00e-03	1.27
7	real	100000	1000	3.59e-02	3.17e-02	0.88
8	real	200000	1000	3.40e-02	3.51e-02	1.03
9	real	300000	1000	3.73e-02	4.19e-02	1.12
10	real	433208	1000	3.37e-02	4.08e-02	1.21

Table 2: Comparison against BIRCH

The distortion for the blacklisting and BIRCH algorithms, given equal run-time, is shown. Six of the datasets are simulated and 4 are real ("petro" data). Datasets 1-3 are as published in [13]. Datasets 4-6 were generated randomly. For generated datasets, the number of classes in the original distribution is also the number of centers reported to both algorithms. The last column shows the BIRCH output distortion divided by the blacklisting output distortion.

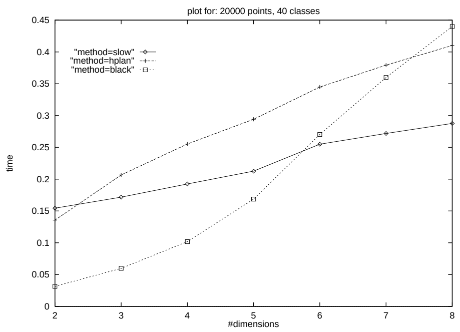

Figure 3: Comparative results on simulated data. Running time, in seconds per iteration, is shown as the number of dimensions varies. Each line stands for a different algorithm: the naive algorithm ("slow"), our simple algorithm ("hplan"), and the blacklisting algorithm ("black").

dimensions varied from 2 to 8. The number of clusters each algorithm was requested to find was 40. The results are shown in Figure 3. The main consequence is that in 2 to 5 dimensions, the blacklisting algorithm is faster than the naive approach, with speedup of up to three-fold in two dimensions. In higher dimensions it is slower. Our simple algorithm is almost always slower than the naive approach.

Another interesting experiment to perform is to measure the sensitivity of our algorithms to changes in the number of points, centers, and dimensions. It is known, by direct analysis, that the naive algorithm has linear dependence on these. It is also known that kd-trees tend to suffer from high dimensionality. In fact, we have just established that in the comparison to the naive algorithm. See [8] as well. To this end, another set of experiments was performed. The experiments used generated data as described earlier (only with 30000 points). But, only the blacklisting algorithm was used to cluster the data-points and the running time was measured. In this experiment-set, the number of dimensions varied from 1 to 8 and the number of centers the program was requested to generate varied from 10 to 80 in steps of 10. The results are shown in Figure 4. The number of dimensions seems to have a super-linear, possibly exponential, effect on the blacklisting algorithm. This worsens as the number of centers increases.

Shown in Figure 5 is the effect of the number of centers on the algorithm. The run-time was measured for the blacklisting algorithm clustering

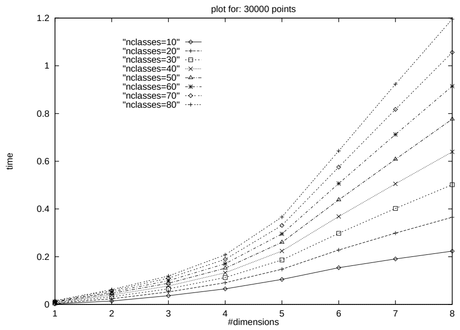

Figure 4: Effect of dimensionality on the blacklisting algorithm. Running time, in seconds per iteration, is shown as the number of dimensions varies. Each line shows results for a different number of classes (centers).

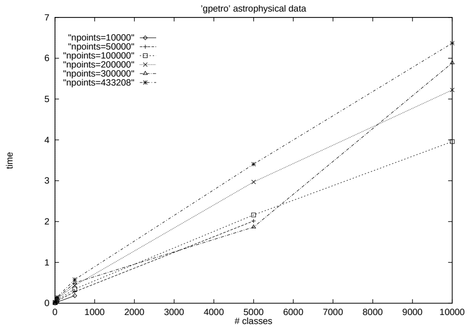

Figure 5: Effect of number of centers on the blacklisting algorithm. Running time, in seconds per iteration, is shown as the number of classes (centers) varies. Each line shows results for a different number of random points from the original file.

random subsets of varying size from the astronomical data, with 50, 500, and 5000 centers. We see that the number of centers has a linear effect on the algorithm. This result was confirmed on simulated data (data not shown).

In Figure 6 the same results are shown, now using the number of points for the X axis. We see a very small increase in run-time as the number of points increases.

4.1 Approximate Clustering

Another way to accelerate clustering is to prune the search when only small error is likely to be incurred. We do this by not descending down the kd-tree when a "small-error" criterion holds for a specific node. We then assume that the points of this node (and its hyper-rectangle h) are divided evenly among all current competitors. For each such competing center c, we update its location as if the relative number of points are all located at closest(c, h).

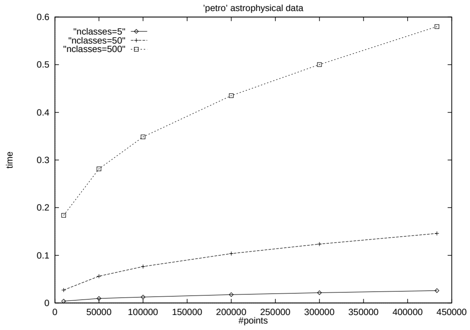

Figure 6: Effect of number of points on the blacklisting algorithm. Running time, in seconds per iteration, is shown as the number of points varies. Each line shows results for a different number of classes.

Our pruning criterion is:

$$n \cdot \sum_{j=1}^{M} \left(\frac{\text{width}(h)_j}{\text{width}(U)_j} \right)^2 \leq d^i$$

where n denotes the number of points in h, U is the "universal" hyper-rectangle bounding all of the input points, i is the iteration number, and d is a constant, typically set to 0.8.

We have conducted experiments with approximate clustering using simulated data. Again, the results shown are averages over 30 random datasets. Figure 7 shows the effect approximate clustering has on the run-time of the algorithm. We notice it runs faster than the blacklisting algorithm, with larger speedups as the number of points increases. It is about 25% faster for 10,000 points, and twice as fast $50,000$ points or more. As for the quality of the output, Figure 8 shows the distortion of the clustering of both algorithms. The distortion of the approximate method is at most 1% more than the blacklisting clustering distortion (which is exact).

5 Conclusion

The main message of this paper is that the well-known k-means algorithm need not necessarily be considered an impractically slow algorithm, even with many records. We have described, analyzed and given empirical results for

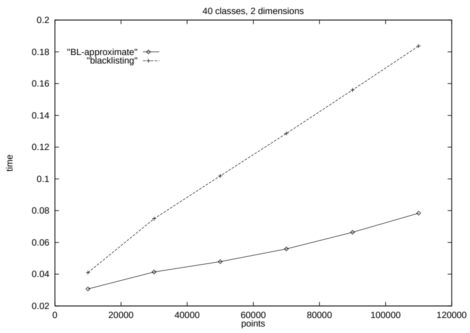

Figure 7: Runtime of approximate clustering. Running time, in seconds per iteration, is shown as the number of points varies. Each line stands for a different algorithm.

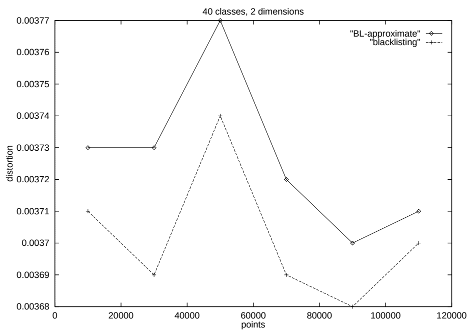

Figure 8: Distortion of approximate clustering. Distortion for approximate and exact clustering. Each line stands for a different algorithm.

a new fast implementation of k-means. We have shown how a kd-tree of all the datapoints, decorated with extra statistics, can be traversed with a new, extremely cheap, pruning test at each node. Another new technique—blacklisting—gives a many-fold additional speed-up, both in theory and empirically.

For datasets too large to fit in main memory, the same traversal and black-listing approaches could be applied to an on-disk structure such as an R-tree, permitting exact k-means to be tractable even for many billions of records.

This method performs badly in high (> 8) dimensions: it is not a clustering panacea, but possibly a worthy problem-specific tool for domains in which there is massively large amounts of low-dimensional data (e.g. astrophysics, geo-spatial-data, and controls). We are also investigating whether AD-trees [10] could be used to give similar speed-ups on categorical data: an advantage of AD-trees is that, subject to many caveats, they remain efficient up to hundreds of dimensions.

Unlike previous approaches (such as the *mrkd*-trees for EM in [9]) this new algorithm scales very well with the number of centers, permitting clustering with tens of thousands of centers.

Why would we care about making exact k-means fast? Why not just use a fast non-k-means approximate clusterer? First, exact k-means is a well-established algorithm that has prospered for many years as a clustering algorithm workhorse. Second, it is often used to help find starting clusters for more sophisticated iterative methods such as mixture models. Third, running k-means on an in-memory sample of the points is a popular approximate clustering algorithm for monstrously large datasets. The techniques in this paper can make such preprocessing steps efficient. Finally, with fast k-means, we can afford to run the algorithm many times in the time it would usually take to run it once. This allows automatic selection of k, or subsets of attributes upon which to cluster, to become a tractable, real-time operation.

References

[1] J. L. Bentley. Multidimensional Divide and Conquer. *Communications of the ACM*, 23(4):214—229, 1980.

[2] C. M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[3] P. S. Bradley and Usama M. Fayyad. Refining initial points for K-Means clustering. In *Proc. 15th International Conf. on Machine Learning*, pages 91–99. Morgan Kaufmann, San Francisco, CA, 1998.

[4] K. Deng and A. W. Moore. Multiresolution instance-based learning. In *The Proceedings of IJCAI-95*, pages 1233–1242. Morgan Kaufmann, 1995.

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

[6] M. Ester, H.-P. Kriegel, and Xiaowei Xu. A database interface for clustering in large spatial databases. In *Proceedings of First International Conference on Knowledge Discovery and Data Mining*. AAAI; Menlo Park, CA, USA, 1995.

[7] A. Gersho and R. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers; Dordrecht, Netherlands, 1992.

[8] Andrew W. Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, 1991. Technical Report 209, Computer Laboratory, University of Cambridge.

[9] Andrew W. Moore. Very fast EM-based mixture model clustering using multiresolution kd-trees. In *Neural Information Processing Systems Conference*, 1998.

[10] Andrew W. Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.

[11] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining,. In *Proc. of VLDB,*, 1994.

[12] SDSS. *The Sloan Digital Survey*. http://www.sdss.org.

[13] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases,. In *to appear on Proc. of ACM SIGMOD Conf.*, pages 103–114, 1995.