Making k-means even faster

Greg Hamerly*

Abstract

The k-means algorithm is widely used for clustering, compressing, and summarizing vector data. In this paper, we propose a new acceleration for exact k-means that gives the same answer, but is much faster in practice. Like Elkan's accelerated algorithm [8], our algorithm avoids distance computations using distance bounds and the triangle inequality. Our algorithm uses one novel lower bound for point-center distances, which allows it to eliminate the innermost k-means loop 80% of the time or more in our experiments. On datasets of low and medium dimension (e.g. up to 50 dimensions), our algorithm is much faster than other methods, including methods based on low-dimensional indexes, such as k-d trees. Other advantages are that it is very simple to implement and it has a very small memory overhead, much smaller than other accelerated algorithms.

1 Introduction

The k-means algorithm is a popular method for automatically classifying vector-based data. It's used in various applications such as vector quantization, density estimation, workload behavior characterization, image compression, and automatic topic identification, among many others. It's been identified as one of the top-10 algorithms in data mining [18].

Because it's such a widely used algorithm, k-means is often used as a subroutine of other learning, coding, and compression algorithms. Many proposed methods are wrappers around k-means, running it many times. For example, a practitioner might try many different initializations to find a high-quality solution. Recent stability-based model selection algorithms also call for running k-means many times with different k values [2]. Since it's so widely used, k-means should be as fast as possible.

The most popular algorihtm for k-means is known as Lloyd's algorithm [13]. The two primary steps in this iterative algorithm are:

1. for each data point x
   - for each cluster center c (the 'innermost loop')
     - compute the distance between x and c
   - assign x to its closest cluster center
2. move each center to the mean of its assigned points

These two steps repeat until the algorithm converges. Given an initial set of centers, this algorithm will move toward a (local) minimum of the k-means criterion, which is the sum of squared distances between each point and its assigned center. While this algorithm can be programmed so that the outer loop in step 1 is over data points or over centers, the way we have presented uses the least memory. The innermost loop over centers is where the algorithm spends the majority of its time. The algorithm we present in this paper eliminates the need to execute this loop 80% of the time or more (in our experiments).

This paper's focus is on accelerating Lloyd's algorithm, but there are many other algorithms that try to optimize the k-means criterion, either in conjunction with or as replacements of Lloyd's algorithm. They include random restarts, local search [12, 6], and simulated annealing [12]. Since Lloyd's algorithm is sensitive to its initialization, another line of research focuses on providing an initialization that will lead to a high-quality solution [11, 16, 1].

Some factors that can cause k-means to be slow are: processing large amounts of data, computing many point-center distances, and requiring many iterations to converge. A primary method of accelerating k-means is applying geometric knowledge to avoid computing point-center distances when possible. Elkan's algorithm [8] exploits the triangle inequality to avoid many distance computations, and is the fastest current algorithm for high-dimensional data. Several tree-based accelerations have also been proposed, such as Moore's anchors hierarchy [15], which is effective in high dimensions, and k-d tree algorithms [17, 12], which are very fast on low-dimensional data.

In this paper we present a new method of accelerating Lloyd's algorithm which builds on Elkan's algorithm.

*Department of Computer Science, Baylor University; greg_hamerly@baylor.edu

Our algorithm is significantly faster than any competing algorithm in data of dimensions up to about 50, and we investigate the reasons it is so fast. They are:

- Novel lower bound: the lower bound used in our algorithm usually allows it to eliminate the innermost k-means loop 80% of the time, or more.
- Simplicity: it is simple to code, and therefore easy to optimize.
- Low latency: it does not require the construction of a spatial index, so the algorithm can begin executing quickly.
- Iterative: the algorithm does not rely on recursive function calls, as do many algorithms that use tree indexes, so it saves costly function calls in performance-critical code.
- Small memory footprint: it uses far less memory than existing accelerated algorithms.

As a result, we believe the algorithm proposed in this paper will become the algorithm of choice for fast k-means clustering on low-dimensional data. Further, our algorithm complements Elkan's algorithm, which is very fast for high-dimensional data. In the conclusion, we discuss methods of combining these two algorithms.

2 Related work

Many people have worked on accelerating clustering algorithms. Some of this work has been in proposing new clustering algorithms or approximation techniques for existing algorithms [19, 9]. Our interest is in accelerating Lloyd's algorithm, so that the accelerated version can be used anywhere Lloyd's algorithm can be used. Because k-means is so widely used, improving its efficiency will have a large immediate impact. We briefly review methods of accelerating k-means.

2.1 Low-dimensional acceleration For low-dimensional data, indexing the data to be clustered is an effective way to accelerate k-means. Kanungo et al. [12] and Pelleg and Moore [17] separately came up with similar ways of adapting a standard k-d tree for fast k-means. Pelleg and Moore's 'blacklisting' algorithm operates as follows:

- Construct a k-d tree on the data to be clustered, keeping sufficient statistics at each node about the vector sum and number of all points in that subtree.
- When calculating the assignment of each data point, filter the centers through the tree in a depth-first recursive fashion. During the descent, prune centers that cannot possibly be the closest to the current subtree. If all but one center can be pruned, stop descending that subtree and assign all its points to the remaining center using the sufficient statistics.

This method is highly effective for large numbers of data points. However, Pelleg and Moore report that it becomes slow when the dimension increases beyond 8. This is due to the curse of dimensionality – when there are many dimensions, then points and centers tend to be far from one another, and little pruning can be done. There are several overhead costs that must be considered as well. Constructing the k-d tree costs $O(n \log(n))$ for n data points, and roughly doubles the memory cost of the algorithm. An algorithm which traverses a non-linear structure like a tree is likely to have poor spatial locality. Further, if the dataset changes, updating the k-d tree to mirror these updates is not trivial.

2.2 High-dimensional acceleration With high-dimensional data, indexing schemes such as k-d trees do not work well, to the point where examining every data point (i.e. using no special data structure for acceleration) can be much faster than algorithms intended for low-dimensional acceleration. However, several alternative methods have shown promise.

Moore proposed the anchors hierarchy [15], a hierarchical structure for dealing with high-dimensional data that obeys the triangle inequality. Two nice properties of this method are that it does not require a vector embedding of the data, nor the ability to index individual dimensions. The paper showed significant speedups (defined in terms of the number of distance calculations required) on many different datasets, even up to 10,000 dimensions. Like the k-d tree, building and maintaining the hierarchical structure can be complex and have significant time and memory overhead.

The most efficient current method for high-dimensional k-means clustering is Elkan's algorithm [8]. His method does not use any indexing structure, but keeps a number of distance bounds that allow it to avoid unnecessary distance computations. Our work is based on this algorithm, so we discuss it in more detail in the following subsection.

Partial distance search [4, 14] is a way to accelerate algorithms like k-means that need to identify closest

points. It takes advantage of the fact that in such algorithms we don't need to know exact distances, only which distance is the minimum. For example, suppose we have calculated $||x - c||^2$, the squared distance between a point x and a center c. When calculating the distance from x to another center c', the typical calculation will proceed by summing squared distances in each dimension. If this sum ever exceeds $||x - c||^2$, then the distance calculation can stop early, since c' cannot possibly be closer than c. The overhead of partial distance search is usually only cost effective in high dimension.

When dealing with high-dimensional sparse data, it is often convenient to re-cast Euclidean distance computations in terms of inner products. Namely,

$$||x - c||^2 = \langle x - c, x - c\rangle = \langle x, x\rangle - 2\langle x, c\rangle + \langle c, c\rangle$$

If we have pre-calculated $\langle x, x\rangle$ and $\langle c, c\rangle$, then computing the squared distance between x and c requires only a single inner product $\langle x, c\rangle$, combined with the cached $\langle x, x\rangle$ and $\langle c, c\rangle$. Further, if we desire the minimum distance from x to any other c, the term $\langle x, x\rangle$ is constant, and can be ignored until the closest c has been identified.

Rather than clustering directly in high dimension, many practitioners will cluster a low-dimensional projection of the original high-dimensional data. Random linear projection [5, 10] is a popular and simple technique. Classical principal components analysis (PCA) is also popular, though more expensive. Recent work has shown strong connections between PCA dimension reduction and k-means clustering [7]. A consequence of this result is that high-dimensional data can be effectively reduced in dimension before clustering is applied. This suggests that accelerated low-dimensional clustering algorithms will continue to be useful in the face of high-dimensional data, since dimension reduction prior to cluster analysis is both theoretically and practically beneficial.

2.3 Elkan's algorithm Elkan [8] presented a highly effective k-means algorithm which eliminates a large number of distance calculations between points and centers. It does this without indexing the data. Instead, it uses efficiently-updated bounds on center-point distances to determine when exact distance calculations can be avoided. Specifically, for n points and k cluster centers it keeps the following extra information:

- n upper bounds on the distance between each point and its assigned center
- kn lower bounds on each point-center distance
- $O(k^2)$ inter-center distances
- n cluster assignments from the previous iteration

The algorithm proceeds much like Lloyd's algorithm, but before computing the distance between a point and any center, it first determines whether it is possible (via the upper/lower bounds and the inter-center distances) that any other center *could* be closer than the currently assigned center. If it's not possible, then the distance calculation is not performed.

There are several cases that must be checked to determine whether each distance calculation can be avoided. One case involves a point x, its assigned center c, and another center c'. If the upper bound on the distance between x and c is less than the lower bound on the distance between x and c', then c' cannot be the closest center to x. Another case is if the same upper bound is less than half the distance from c to c', then again c' cannot be the closest center to x (due to the triangle inequality). For more details on all the cases and the general algorithm, see [8].

Elkan's algorithm updates the upper and lower bounds each time the centers move. When the bounds update they are no longer tight, but are likely to remain useful for eliminating point-center distance calculations. The upper bound for a point x increases by the distance moved by its assigned center, and the lower bound between a point x and a center c decreases by the distance moved by c. Using this method, the bounds are loose but remain correct.

2.4 Other acceleration techniques Other strategies for accelerating k-means include subsampling large datasets [9, 3], as well as finding initializations that will place centers near their final positions. Peña et al. [16] examined how four initialization techniques affect the quality of the clustering.

Hochbaum and Shmoys [11] proposed their furthest-first algorithm as an approximate solution to the k-center problem. It has been popular to use this algorithm as an initialization technique for k-means. This method chooses a randomly-selected point as the first center, and then repeatedly selects, as the next center, the point that is furthest from any current center. This method has the downside of tending to choose outliers as part of the initial centers.

Similar in spirit to furthest-first initialization, the k-means++ initialization algorithm [1] adds randomness to the process and tends to avoid the outlier problem of furthest-first. Choosing the first center randomly,

each subsequent center is chosen with a probability that is proportional to its squared distance to its closest existing center. This is currently the method of choice for initializing k-means, and it offers statistical guarantees on the quality of clusterings initialized this way.

3 A new algorithm

Our proposed algorithm is a modification and simplification of Elkan's k-means algorithm. As such, it uses efficiently updated distance bounds and the triangle inequality to avoid point-center distance calculations. Beyond just avoiding individual distance computations, our algorithm is quite effective at regularly eliminating the innermost loop in Lloyd's algorithm. That is, it often is able to skip the loop which iterates over the k centers. This is possible by maintaining two distance bounds per data point for its *two closest centers*. One is an upper bound on the distance to the closest center, and one is a lower bound on the distance to the second-closest center. Unlike Elkan's algorithm, our algorithm does not maintain lower bounds between all point-center combinations, so our lower bound is different.

Our algorithm, as shown in the experiments, outperforms the current best k-means algorithms on data of low and moderate dimension. Elkan's algorithm performs best on high-dimensional data. Thus, the two algorithms are complementary, and one could choose between them based on the conditions of the data.

To introduce our algorithm, there are several definitions we must first review. We assume a distance metric $d(\cdot, \cdot)$ defined on point-center and center-center pairs. Structures that relate to cluster centers are:

- $c(j)$ – cluster center j (where $1 \leq j \leq k$),
- $c'(j)$ – vector sum of all points in cluster j,
- $q(j)$ – number of points assigned to cluster j,
- $p(j)$ – distance that $c(j)$ last moved, and
- $s(j)$ – distance from $c(j)$ to its closest other center.

Structures that relate to data points are:

- $x(i)$ – data point i (where $1 \leq i \leq n$),
- $a(i)$ – index of the center to which $x(i)$ is assigned,
- $u(i)$ – upper bound on the distance between $x(i)$ and its assigned center $c(a(i))$, and
- $l(i)$ – lower bound on the distance between $x(i)$ and its *second* closest center – that is, the closest center to $x(i)$ that is not $c(a(i))$.

Note that $u(i)$ and $l(i)$ are bounds on point-center distances, and that these bounds are not always tight. However, our algorithm guarantees the following:

- $u(i) \geq ||x(i) - c(a(i))||$
- $l(i) \leq \min_{j \neq a(i)} ||x(i) - c(j)||$.

Also note that the lower bound we use is not the same as Elkan's, since we have only one per point, and it is not tied to any particular center. That is, we don't explicitly keep track of *which* center is the second closest to each point. Now we turn to some details of the algorithm.

3.1 Algorithm description The code listing in Algorithm 1 gives a detailed description of the proposed algorithm. The subroutines listed in Algorithms 2 through 5 provide support to the main algorithm. After initializing the upper and lower bounds and the assignments, the algorithm iterates until the centers converge.

Each iteration first updates the values in s. Then the loop beginning at line 5 iterates over every data point $x(i)$, and line 7 determines if the bounds permit skipping the innermost loop over all centers. If not, the algorithm recomputes $u(i)$ in case the bound was too loose, and tests the condition again on line 9. If both tests fail, then it calls the innermost loop over all centers (Point-All-Ctrs, Algorithm 3) which updates $u(i)$, $l(i)$, and $a(i)$ in an $O(kd)$ loop.

While the pseudocode given here is a fairly complete description, for brevity we have left out several of the small optimizations which are possible. These include avoiding square-roots on distance computations when the goal is identifying closest neighbors and avoiding repeated distance computations. Now we explore some of the other algorithmic details, such as efficiently updating the bounds and keeping track of the center locations.

3.2 Using bounds to avoid distance computations Each data point $x(i)$ has one upper bound $u(i)$, which is identical to the upper bound used by Elkan. It bounds the distance between $x(i)$ and its closest center. Rather than keeping k lower bounds for each data point, our algorithm keeps only one per data point, which is $l(i)$. It represents a lower bound on the distance between the data point and its *second-closest* cluster center. As

Algorithm 1 K-means(dataset x, initial centers c)

```
1: INITIALIZE(c, x, q, c', u, l, a)
2: while not converged do
3:    for j = 1 to |c| do                                  {update s}
4:       s(j) ← min_{j'≠j} d(c(j'), c(j))
5:    for i = 1 to |x| do
6:       m ← max(s(a(i))/2, l(i))
7:       if u(i) > m then                                 {first bound test}
8:          u(i) ← d(x(i), c(a(i)))                       {tighten upper bound}
9:          if u(i) > m then                              {second bound test}
10:            a' ← a(i)
11:            POINT-ALL-CTRS(x(i), c, a(i), u(i), l(i))
12:            if a' ≠ a(i) then
13:               update q(a'), q(a(i)), c'(a'), c'(a(i))
14:   MOVE-CENTERS(c', q, c, p)
15:   UPDATE-BOUNDS(p, a, u, l)
```

Algorithm 2 INITIALIZE(c, x, q, c', u, l, a)

```
1: for j = 1 to |c| do
2:    q(j) ← 0
3:    c'(j) ← 0⃗
4: for i = 1 to |x| do
5:    POINT-ALL-CTRS(x(i), c, a(i), u(i), l(i))
6:    q(a(i)) ← q(a(i)) + 1
7:    c'(a(i)) ← c'(a(i)) + x(i)
```

Algorithm 3 POINT-ALL-CTRS($x(i), c, a(i), u(i), l(i)$)

```
1: a(i) ← argmin_j d(x(i), c(j))
2: u(i) ← d(x(i), c(a(i)))
3: l(i) ← min_{j≠a(i)} d(x(i), c(j))
```

Algorithm 4 MOVE-CENTERS(c', q, c, p)

```
1: for j = 1 to |c| do
2:    c* ← c(j)
3:    c(j) ← c'(j)/q(j)
4:    p(j) ← d(c*, c(j))
```

Algorithm 5 UPDATE-BOUNDS(p, a, u, l)

```
1: r ← argmax_j p(j)
2: r' ← argmax_{j≠r} p(j)
3: for i = 1 to |u| do
4:    u(i) ← u(i) + p(a(i))
5:    if r = a(i) then
6:       l(i) ← l(i) − p(r')
7:    else
8:       l(i) ← l(i) − p(r)
```

long as $u(i) \leq l(i)$, the center assignment for $x(i)$ cannot change.

Recall that $s(j)$ represents the distance between center j and its closest other center. If $u(i) \leq s(a(i))/2$, then the triangle inequality guarantees that center $a(i)$ is closer to $x(i)$ than any other center.

The primary acceleration in this algorithm comes from exploiting one of these two cases: either $u(i) \leq l(i)$, or $u(i) \leq s(a(i))/2$. Taken together, these are equivalent to $u(i) \leq \max(l(i), s(a(i))/2)$. If this condition is true then the algorithm can avoid the innermost loop which computes distances between $x(i)$ and all the k cluster centers. However, if it is false, then the algorithm must compute distances between $x(i)$ and all centers to determine the correct assignment. Algorithm 3 performs this innermost loop, and at the same time calculates the tight values of $u(i)$ and $l(i)$.

Elkan's algorithm employs only the comparison with s to avoid the innermost loop over all centers. It does not have a lower-bound test for this purpose. Instead, it loops over all centers and use a lower bound for each center to determine whether it was necessary to calculate the distance between $x(i)$ and that center. Thus, our algorithm has more potential opportunities to avoid the innermost loop entirely, while Elkan's algorithm has more potential opportunities to avoid distance calculations. Indeed, experiments show that our algorithm is much more effective at avoiding the innermost loop.

3.2.1 Updating bounds Each time the centers move, the upper and lower bounds of each point need to change to reflect these moves. The subroutine MOVE-CENTERS keeps track of how far each center moves in the structure p.

After this, UPDATE-BOUNDS can update the bounds u and l using p. It first identifies the two furthest-moving centers, according to p. Then each $u(i)$ increases by the distance moved by center $a(i)$ (that is, $p(a(i))$), and each $l(i)$ decreases by the distance moved by the furthest-moving center $p(r)$. One small optimization is if $a(i) = r$ (the furthest-moving center), then it is safe to decrease $l(i)$ by the distance moved by the second-furthest-moving center $p(r')$.

3.2.2 Moving centers A standard way to update center locations is to sum up the points assigned to each center as the assignments are discovered, and compute the average at the end of each k-means iteration. Our

Table 1: This table gives the overhead (in time and memory) for each examined algorithm. Each entry represents the asymptotic overhead spent by that algorithm *beyond* Lloyd's algorithm. The initialization time (column 2) is extra time needed to allocate memory and create data structures. Time/iteration is the extra time spent during each k-means iteration, and memory accounts for all extra memory used.

	init. time	time/iteration	memory
k-d tree	$nd + n\log(n)$	-	nd
elkan	$ndk + dk^2$	dk^2	$nk + k^2$
hamerly	ndk	dk^2	n

algorithm uses an alternative method which we call 'delta updates' that permits the algorithm to avoid adding vectors when assignments don't change. Just after calling Point-All-Ctrs, if the assignment has changed the algorithm adjusts the cluster sums (c') and counts (q) for the two affected centers. It subtracts $x(i)$'s influence from its old assigned center, and adds its influence to the new assigned center. If there are few assignment changes, this method has the potential to reduce computation. If there are many assignment changes, then this method will likely be more costly. Thus, this optimization is likely to be most useful when k is small (relative to n), or during the final stages of k-means when few data points tend to change cluster assignments.

3.2.3 Time and memory analysis The time per k-means iteration for our algorithm is $O(ndk+dk^2)$, where the ndk term comes from Lloyd's algorithm, and the dk^2 term comes from updating s. The time for Initialize is $O(ndk)$. The memory overhead comes from keeping u, l, a, s, and p, giving $3n + 2k$ extra scalar values. This overhead is asymptotically insignificant compared with Lloyd's algorithm, which uses $O(nd+kd)$ memory. Thus, our algorithm should be useful in most situations where it is appropriate to use Lloyd's algorithm, with only slightly higher memory requirements. Indeed, our algorithm uses the least memory of the three accelerated versions examined here, as demonstrated in the experimental results (see Table 4).

Table 1 describes the asymptotic time and memory overheads of each of the three accelerated algorithms considered in this paper. Using a k-d tree nearly doubles the amount of memory needed over Lloyd's algorithm, in order to keep summary statistics at each node, and to store the tree structure. Elkan's algorithm requires an additional $(n + 1)k$ scalar values for the bounds, as well as k^2 scalar values for the inter-center distances. These extra memory requirements become prohibitive for large datasets and large k. One nice feature of Elkan's algorithm is that it is not necessary to store all the $(n + 1)k$ bounds in memory at once; we can simply stream them in while making passes over the dataset. While this would allow us to run the algorithm on very large data sets, it will not make the algorithm faster than storing data in core memory. Our algorithm shares this feature. However, since it has a much smaller memory footprint (especially for large k), it can cluster much larger data sets in memory.

3.3 Parallelism and large datasets Our algorithm parallelizes easily for multi-threaded or multi-core processing. Structures related to the dataset (x, u, l, and a) would be partitioned across the processors. Structures related to the centers (c, c', q, p, and s) would be replicated across the processors and synchronized at each call to Move-Centers. We expect the potential for large real-time speedups using parallel processing this way, since the majority of time would be spent in independent computation.

Datasets which are too large to fit in a single processor's memory could easily be partitioned this way across multiple processors. For massive datasets which do not fit in memory, the dataset-related structures could be streamed from disk. Since the algorithm iterates over the dataset in order, a fast external storage device should be able to quickly stream the dataset and bounds to memory.

3.4 Other discussion One of the benefits of our algorithm over indexed methods (like k-d trees) is that it is fast despite the fact that it iterates over the entire dataset. In some applications like constraint-based clustering, it is conceivable that one cannot use indexes easily. For example, it may be that a nearby pair of points might be constrained to belong to separate clusters. Situations like this are easier to deal with when visiting every data point explicitly. Generalizing somewhat, any adaptation of k-means which cannot simply group points based on proximity will be easier to implement with our algorithm than with an index.

4 Experiments

We compare Lloyd's k-means algorithm with three accelerated versions: the k-d tree algorithm (also called the blacklisting or filtering algorithm) [17, 12], Elkan's

triangle inequality algorithm [8], and the algorithm proposed in this paper. All were implemented on a common code base in single-threaded C++. Each algorithm chooses the same set of initial centers using k-means++ [1].

Moore [15] and Elkan [8] reported results in terms of the number of distance computations performed (or avoided). However, we are interested in real-world performance. Therefore, we report the time it takes to cluster and the memory footprint during execution. The code was compiled with `g++` version 4.1.2, and used 64-bit double-precision primitives. The experiments ran on Linux 2.6.18 on 8-core Intel Xeon 2.66 GHz computers with 16 GB of memory, but only a single core was used for each experiment. For each experiment we measure CPU time using `getrusage()` and memory footprint using the file `/proc/[PID]/statm`.

4.1 Data sets We perform tests on several datasets, both synthetic and from other applications. The synthetic datasets are generated from uniform hypercube distributions of varying dimension. Uniform data provides perhaps the worst case for most accelerated k-means algorithms, because there is little structure in the data. This lack of structure means that most points are likely to be on the edges of clusters, rather than close to cluster centers. Distance pruning methods in k-means work best when points are much closer to their assigned centers than to other centers, so we expect pruning methods have the most difficulty with these datasets.

We also compare the performance of these algorithms on several datasets which were used in Elkan's paper. We did not use the high-dimensional datasets 'mnist784' and 'random,' because Elkan's algorithm more effectively reduces the number of distance computations, which are more expensive in high dimension. However, our algorithm does dramatically outperform both Lloyd's algorithm and the k-d tree algorithm on such high-dimensional data.

4.2 Timing results For each experiment we measure total time and time per iteration. Both are user CPU time measurements. The total time includes three phases: loading data from disk, initializing any necessary data structures (e.g. a k-d tree), and clustering. The time per iteration is the time spent in only the last phase (clustering) divided by the number of iterations. Since all these algorithms are exact k-means implementations, they all perform the same number of iterations for the same dataset and initialization. The time per iteration allows us to compare algorithms across different datasets and numbers of clusters.

Table 2 shows the performance of each algorithm. Our algorithm is often more than 10 times faster in total time than Lloyd's algorithm. It's fastest out of all the algorithms in most experiments, including all 8-dimensional and 32-dimensional experiments. For the uniform dataset in 8 dimensions with 500 clusters, our algorithm is at least 8 times faster than any of the other algorithms. On 2-dimensional datasets, our algorithm is usually fastest, but k-d trees perform very well here too. However, k-d tree performance degrades quickly as the dimension increases. For the three datasets that have 50 to 56 dimensions (covtype, kddcup, mnist50), our algorithm performs best when k is small, and is comparable to Elkan's algorithm for large k.

On per-iteration time, over all experiments, our algorithm is an average of 6.1 times faster than Lloyd's algorithm, 2.8 times faster than the k-d tree algorithm, and 2.4 times faster than Elkan's algorithm. Two experiments where our algorithm performed particularly well were for the uniform random dataset with 8 dimensions and $k = 100$ or $k = 500$.

4.3 Avoiding the innermost loop One of the ways that both our algorithm and Elkan's avoid computation is by skipping the innermost loop over the k centers. Elkan's algorithm skips this loop if the test $u(i) \leq s(a(i))/2$ is true. Our algorithm skips this loop if that condition is true, or if $u(i) \leq l(i)$. Further, our algorithm checks these conditions twice when necessary, after lowering $u(i)$ to be a tight bound. Thus, our algorithm has potentially more opportunities to avoid the innermost loop.

The innermost loops of these two algorithms are different. Our algorithm simply computes the point-center distance for each center (except the assigned center, since that distance was just calculated when updating $u(i)$). Elkan's algorithm performs further checks against lower bounds for each center to determine if each point-center distance computation is necessary.

We evaluate the frequency that each algorithm is able to skip the innermost loop. The results are in Table 3. Our algorithm is clearly much more effective at avoiding the innermost loop. It's interesting that even though our algorithm is considerably slower in high dimension than Elkan's algorithm (due to the many more distance computations), we are still very effective at skipping the innermost loop for 128-dimensional data.

Table 2: This table shows the time (user CPU seconds) each algorithm uses on several datasets. We report total time first, followed by per-iteration time in parentheses. The per-iteration time includes only the time when the algorithm is running, and excludes time to load data and initialize any data structures (e.g. constructing a k-d tree). The first three datasets are synthetically generated; the remaining four are those used by Elkan. Bold indicates the lowest time among the algorithms. Our algorithm competitive with or much faster than other algorithms for 2 to 32 dimensions. It also performs competitively on up to 50 dimensions.

		Total user CPU Seconds (User CPU seconds per iteration)			
Dataset		$k = 3$	$k = 20$	$k = 100$	$k = 500$
uniform random	iterations	44	227	298	710
$n = 1250000$	lloyd	4.0 (0.058)	61.4 (0.264)	320.2 (1.070)	3486.9 (4.909)
$d = 2$	kd-tree	3.5 (**0.006**)	**11.8** (**0.035**)	34.6 (0.102)	338.8 (0.471)
	elkan	7.2 (0.133)	75.2 (0.325)	353.1 (1.180)	2771.8 (3.902)
	hamerly	**2.7** (0.031)	14.6 (0.058)	**28.2** (**0.090**)	**204.2** (**0.286**)
uniform random	iterations	121	353	312	1405
$n = 1250000$	lloyd	21.8 (0.134)	178.9 (0.491)	660.7 (2.100)	13854.4 (9.857)
$d = 8$	kd-tree	117.5 (0.886)	622.6 (1.740)	2390.8 (7.633)	46731.5 (33.254)
	elkan	14.1 (0.071)	130.6 (0.354)	591.8 (1.879)	11827.9 (8.414)
	hamerly	**10.9** (**0.045**)	**40.4** (**0.099**)	**169.8** (**0.527**)	**1395.6** (**0.989**)
uniform random	iterations	137	4120	2096	2408
$n = 1250000$	lloyd	66.4 (0.323)	5479.5 (1.325)	12543.8 (5.974)	68967.3 (28.632)
$d = 32$	kd-tree	208.4 (1.324)	29719.6 (7.207)	74181.3 (35.380)	425513.0 (176.697)
	elkan	48.1 (0.189)	1370.1 (0.327)	2624.9 (1.242)	14245.9 (5.907)
	hamerly	**46.9** (**0.180**)	**446.4** (**0.103**)	**1238.9** (**0.581**)	**9886.9** (**4.097**)
birch	iterations	52	179	110	99
$n = 100000$	lloyd	0.53 (0.004)	4.60 (0.024)	11.80 (0.104)	48.87 (0.490)
$d = 2$	kd-tree	**0.41** (<**0.001**)	0.96 (**0.003**)	2.67 (0.021)	17.68 (0.173)
	elkan	0.58 (0.005)	4.35 (0.023)	11.80 (0.104)	54.28 (0.545)
	hamerly	0.44 (0.002)	**0.90** (**0.003**)	**1.86** (**0.014**)	**7.81** (**0.075**)
covtype	iterations	19	204	320	111
$n = 150000$	lloyd	3.52 (0.048)	48.02 (0.222)	322.25 (0.999)	564.05 (5.058)
$d = 54$	kd-tree	6.65 (0.205)	266.65 (1.293)	2014.03 (6.285)	3303.27 (29.734)
	elkan	3.07 (0.022)	11.58 (0.044)	70.45 (0.212)	**152.15** (**1.347**)
	hamerly	**2.95** (**0.019**)	**7.40** (**0.024**)	**42.83** (**0.126**)	169.53 (1.505)
kddcup	iterations	39	55	169	142
$n = 95412$	lloyd	4.74 (0.032)	12.35 (0.159)	116.63 (0.669)	464.22 (3.244)
$d = 56$	kd-tree	9.68 (0.156)	58.55 (0.996)	839.31 (4.945)	3349.47 (23.562)
	elkan	4.13 (0.012)	6.24 (0.049)	32.27 (0.169)	**132.39** (**0.907**)
	hamerly	**3.95** (**0.011**)	**5.87** (**0.042**)	**28.39** (**0.147**)	197.26 (1.364)
mnist50	iterations	37	249	190	81
$n = 60000$	lloyd	2.92 (0.018)	23.18 (0.084)	75.82 (0.387)	162.09 (1.974)
$d = 50$	kd-tree	4.90 (0.069)	100.09 (0.393)	371.57 (1.943)	794.51 (9.780)
	elkan	2.42 (0.005)	7.02 (0.019)	**21.58** (**0.101**)	**55.61** (**0.660**)
	hamerly	**2.41** (**0.004**)	**4.54** (**0.009**)	21.95 (0.104)	77.34 (0.928)

Table 3: These results show the fraction of times that each algorithm was able to skip the innermost loop on data of different dimensions (values closer to 1 are better). These results are averaged over runs using k = 3, 20, 100, and 500 (one run for each k). The randX datasets are uniform random hypercube data with X dimensions.

dataset	rand2	rand8	rand32	rand128
elkan	0.56	0.01	0.00	0.00
hamerly	0.97	0.88	0.91	0.83
dataset	birch	covtype	kddcup	mnist50
elkan	0.52	0.34	0.18	0.22
hamerly	0.94	0.89	0.82	0.82

For Elkan's algorithm, the performance difference is striking between the random hypercube datasets (top rows of Table 3) and the application datasets (bottom rows). We believe this is due to the natural clustered nature of the application datasets, and the lack of structure in the random datasets. In clustered data, it is likely that points will be much closer to their assigned centers than to other centers. Our algorithm performs well regardless of the clustered nature of the data.

One could add our new lower bound to Elkan's algorithm. This would allow Elkan's algorithm to avoid the innermost loop more frequently. However, this approach does not solve the efficiency problem entirely, since after each k-means iteration Elkan's algorithm must loop over and update all nk lower bounds, which is effectively repeating the same innermost loop that it (may have) just avoided. Further, it adds the overhead of keeping this extra bound.

4.4 Memory use A big advantage of our algorithm is its small memory footprint, especially compared with other accelerated algorithms. Table 4 shows the memory footprints of the compared algorithms for various datasets and k. Our algorithm consistently has the lowest memory use of the accelerated algorithms, and is usually close to the memory used by Lloyd's algorithm.

If memory resources are constrained, then our algorithm is clearly the one to choose for k-means clustering. Using very little additional memory, our algorithm is much faster than Lloyd's algorithm. Compared with Elkan's algorithm, which has memory requirements that scale as $O(k^2)$, our algorithm will have much lower memory footprint for large numbers of clusters. The k-d tree implementation uses a little less than twice the memory our algorithm uses. A shared-memory parallel computer should be able to run many copies of our algorithm simultaneously on different large datasets, while the same may not be true for algorithms with significantly larger memory requirements.

Table 4: Memory use of each algorithm on various datasets. Note that our algorithm ('hamerly') consistently uses the least amount of memory among the three accelerated algorithms, and is usually very close to the memory used by Lloyd's algorithm.

		Megabytes			
Dataset	Algorithm	k=3	k=20	k=100	k=500
uniform	lloyd	7.5	7.5	7.5	7.5
random	kd-tree	32.1	32.1	32.1	32.1
n=1.25M	elkan	19.8	60.3	251.0	1205.2
d=2	hamerly	14.7	14.7	14.7	14.7
uniform	lloyd	21.9	21.9	21.9	21.9
random	kd-tree	54.8	54.8	54.8	54.8
n=1.25M	elkan	34.1	74.6	265.3	1219.5
d=8	hamerly	29.0	29.0	29.0	29.0
uniform	lloyd	79.1	79.1	79.1	79.1
random	kd-tree	145.2	145.2	145.2	145.3
n=1.25M	elkan	91.3	131.8	322.6	1276.8
d=32	hamerly	86.2	86.2	86.2	86.3
birch	lloyd	1.4	1.1	1.1	1.3
n=100K	kd-tree	2.9	2.9	2.8	2.7
d=2	elkan	2.1	5.2	20.6	97.3
	hamerly	1.5	1.7	1.6	1.5
covtype	lloyd	16.2	16.2	16.1	16.4
n=150K	kd-tree	27.2	27.2	27.2	27.3
d=54	elkan	17.4	22.5	45.3	160.4
	hamerly	17.0	17.0	16.8	17.2
kddcup	lloyd	10.9	10.8	11.1	11.2
n=95412	kd-tree	18.8	18.9	19.1	19.0
d=56	elkan	11.9	15.1	29.6	103.1
	hamerly	11.6	11.6	11.3	11.7
mnist50	lloyd	6.3	6.6	6.4	6.8
n=60K	kd-tree	10.5	10.4	10.6	10.7
d=50	elkan	7.0	9.1	18.4	64.8
	hamerly	6.9	6.9	6.9	6.8

4.5 Lesion study We want to discover which parts of our algorithm provide the greatest speedup, so we perform a lesion study where we remove different parts of the algorithm in order to find out how they affect the running time. We consider the following modifications:

- **No-S**: Removing the structure s, which represents the distance between each center and its closest other center. In this version, the algorithm uses only upper and lower bounds for each point to prune distance calculations. Note that this also

removes the $O(k^2)$ term from the per-iteration runtime.

- **No-Lower**: Removing the lower bound for each data point. Here, the algorithm compares each upper bound only with s to prune distance calculations.
- **No-Upper**: Removing u, and removing the first bound comparison at line 7 in Algorithm 1.
- **No-Delta**: Removing the incremental updates of center locations, instead recalculating center locations from scratch at each iteration.

The clearest evidence from the lesion study is that the lower bound is essential to a fast runtime. Removing the lower bound dramatically slows our algorithm, while removing the other parts did not. The remaining modified algorithms all performed reasonably well, sometimes better than our original algorithm, but on the large majority of experiments the original algorithm performed best. Each modification changed the algorithm to perform better in certain situations:

- **No-S** performed better with high dimensional data, for two reasons. First, it did not perform $O(k^2)$ center-center distance computations each iteration, and distance computations are expensive in high dimension. Second, in high dimension s is less effective at avoiding the innermost loop, due to the curse of dimensionality.
- **No-Lower** did not perform well, especially in high dimension. The lower bound on the distance to each point's second-closest center is the key to high performance.
- **No-Upper** performed better with low dimensional data. Removing the upper bound requires computing the exact distance to the assigned cluster for each data point every time, which is more expensive in high dimension.
- **No-Delta** performed better with a large number of clusters. When there are many clusters, we expect that points will change assignments frequently, so the delta method will actually incur greater overhead than simply calculating the centers from every assignment each iteration.

Clearly, each optimization specializes under different circumstances, and can be added or removed depending on application-specific requirements.

5 Conclusion

We present a simple algorithm which is a drop-in replacement for the standard k-means algorithm (Lloyd's algorithm), but is much faster. In low-dimensional settings (e.g. up to 50 dimensions), our algorithm is as fast or much faster than current accelerated algorithms, including methods based on indexing the data.

The new algorithm is derived from Elkan's accelerated algorithm, and as such it uses several efficiently-updated bounds to avoid unnecessary distance computations. However, it uses only one novel lower bound per data point on point-center distances, instead of k lower bounds per data point. This lower bound is on the distance between the point and its second-closest center.

There are two primary reasons our algorithm outperforms the current best algorithms. First is that it has very low overhead – it uses far less memory than other accelerated algorithms, and has far fewer bounds to maintain compared with Elkan's algorithm. Second, its novel lower bound allows frequent eliminations of the innermost loop (usually more than 80% of the time) over point-center distance calculations.

While our algorithm is always faster than the standard algorithm, it is much slower than Elkan's algorithm in high dimensions. However, we view our algorithm and Elkan's as complementary – they each excel in different domains. A single algorithm built from both could select the appropriate algorithm depending on the dimension.

Recent work [7] has shown that there is a strong connection between low-dimensional data representation (via principal components analysis) and k-means clustering. Thus, we still need algorithms that are efficient in low dimension.

Our future work includes constructing a hybrid algorithm where more than one but far fewer than k lower bounds are kept per data point. This combines the benefits of our algorithm and Elkan's – low overhead and good high-dimensional performance. This method could select the appropriate number of lower bounds based on k and the number of dimensions. Further work is necessary to determine the best method of choosing this number of bounds.

Each iteration of our algorithm requires computing $O(k^2)$ inter-center distances. We have made attempts at reducing this computational cost using the triangle inequality, but without much practical benefit. It remains to be seen if approximations to the inter-center distances could provide additional benefit. Our

lesion study shows that eliminating these inter-center distances completely can have practical benefit in high-dimensional settings.

Acceleration methods based on k-d trees are usually much slower than our algorithm, especially above 2 dimensions. However, in very low dimensions, there is potential for further acceleration by combining tree structures with distance bounds. For example, we might keep distance bounds between each tree node and center.

The proposed algorithm should work well with massive datasets that are too large to fit in memory. Since the algorithm can stream in the data points and bounds, this provides a good access pattern for disks. Further, it would only need to load the points for which the bounds indicate distance calculations are required. Thus, we expect our algorithm will show additional speedup compared with the standard algorithm on massive disk-based datasets.

Our work provides a very fast k-means algorithm, but it is also the basis for a faster universal k-means algorithm. Such an algorithm will select various optimizations like tree indexing, delta-updates, keeping inter-center distances, and the number of lower bounds depending on the dimension, number of clusters, dataset size, and memory constraints.

References

[1] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *SODA*, pages 1027–1035. SIAM, 2007.

[2] Shai Ben-David, Dávid Pál, and Hans Ulrich Simon. Stability of k-means clustering. *Lecture Notes in Computer Science*, 4539:20–34, 2007.

[3] Paul S. Bradley and Usama M. Fayyad. Refining initial points for k-means clustering. In Jude W. Shavlik, editor, *ICML*, pages 91–99. Morgan Kaufmann, 1998.

[4] D. Cheng, A. Gersho, B. Ramamurthi, and Y. Shoham. Fast search algorithms for vector quantization and pattern matching. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 372–375, 1984.

[5] Sanjoy Dasgupta. Experiments with random projection. In Craig Boutilier and Moisés Goldszmidt, editors, *UAI*, pages 143–151. Morgan Kaufmann, 2000.

[6] Inderjit S. Dhillon, Yuqiang Guan, and J. Kogan. Iterative clustering of high dimensional text data augmented by local search. In *ICDM*, pages 131–138, 2002.

[7] Chris H. Q. Ding and Xiaofeng He. k-means clustering via principal component analysis. In *ICML*, pages 225–232, 2004.

[8] Charles Elkan. Using the triangle inequality to accelerate k-means. In Tom Fawcett and Nina Mishra, editors, *ICML*, pages 147–153. AAAI Press, 2003.

[9] Fredrik Farnstrom, James Lewis, and Charles Elkan. Scalability for clustering algorithms revisited. *SIGKDD Explorations*, 2(1):51–57, 2000.

[10] Greg Hamerly, Erez Perelman, Jeremy Lau, Brad Calder, and Timothy Sherwood. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research*, 7:343–378, 2006.

[11] Dorit S. Hochbaum and David B. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, May 1985.

[12] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, 2002.

[13] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[14] J. McNames. Rotated partial distance search for faster vector quantization encoding. *IEEE Signal Processing Letters*, 7(9), 2000.

[15] Andrew Moore. The anchors hierarchy: Using the triangle inequality to survive high-dimensional data.

[16] J. M. Peña, J. A. Lozano, and P. Larrañaga. An empirical comparison of four initialization methods for the k-means algorithm.

[17] Dan Pelleg and Andrew W. Moore. Accelerating exact k-means algorithms with geometric reasoning. In *KDD*, pages 277–281, 1999.

[18] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus F. M. Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, 2008.

[19] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Data Min. Knowl. Discov.*, 1(2):141–182, 1997.