Geometric methods to accelerate k-means algorithms

Petr Ryšavý
Department of Computer Science
Baylor University
Waco, TX 76798-7356
petr_rysavy@alumni.baylor.edu

Greg Hamerly
Department of Computer Science
Baylor University
Waco, TX 76798-7356
greg_hamerly@baylor.edu

Abstract

The k-means algorithm is popular for data clustering applications. Most implementations use Lloyd's algorithm, which does many unnecessary distance calculations. Several accelerated algorithms (Elkan's, Hamerly's, heap, etc.) have recently been developed which produce exactly the same answer as Lloyd's, only faster. They avoid redundant work using the triangle inequality paired with a set of lower and upper bounds on point-centroid distances. In this paper we propose several novel methods that allow those accelerated algorithms to perform even better, giving up to eight times further speedup. Our methods give tighter lower bound updates, efficiently skip centroids that cannot possibly be close to a set of points, keep extra information about upper bounds to help the heap algorithm avoid more distance computations, and decrease the number of distance calculations that are done in the first iteration.

1 Introduction

Clustering is one of the most frequently-used types of learning algorithms, and is often at the heart of learning systems. Its applications include image segmentation, market studies, and social network analysis. Clustering algorithms should be fast, not only because of their wide applicability, but also because a common practice for finding good clusterings is to randomly restart with different initializations. In this paper we propose four methods of making existing state-of-the-art k-means clustering algorithms even faster, by a factor of up to 8 times in our experiments.

Clusters are typically sets of points that are similar according to some measure. For k-means this measure is the squared distance between the point and its assigned centroid, where the centroids are k artificially introduced points. In k-means each point is assigned to its closest centroid and centroids are placed so as to minimize the *distortion function*

$$J(c) = \sum_{i=1}^{n} \|x_i - c(x_i)\|^2,$$

where function c returns the centroid assigned to point x_i. Finding a global minimum of this function is difficult as it is not convex, but the popular Lloyd's algorithm [9] finds a local minimum using hill-climbing.

Naively implemented, Lloyd's algorithm repeats a lot of work unnecessarily across iterations. The repeated work is computing the distances between each point and all k centroids. While several algorithms use different techniques to eliminate this redundancy [11, 10, 7, 12, 4, 5, 6, 3], in this work we will focus on the subset of Elkan's algorithm [4], Hamerly's algorithm [5], the annulus algorithm [6] and the heap algorithm [6]. Detailed explanations of how those algorithms work is beyond scope of this paper, but [6] provides an overview. We will briefly explain their main features in Section 2.

These algorithms are faster than Lloyd's original algorithm because they avoid unnecessary distance calculations by introducing a set lower and upper bounds on distances between points and selected centroids. They eliminate provably unnecessary distance calculations based on the *triangle inequality*, which is also used for updating the bounds after each iteration using just the movement of the centroids. A main feature of these algorithms is that they produce exactly the same results as Lloyd's algorithm, but faster.

In this paper we propose four improvements:

1. We propose *lower bound updates that are tighter* than those produced by the triangle inequality. That is, we reduce lower bounds by smaller amounts, permitting the bounds to hold for more iterations and resulting in fewer distance calculations. This novel update uses the direction a centroid moves, which previous algorithms did not.

2. We propose the concept of *neighbor centroids* of a centroid, allowing us to avoid examining all k centroids in the innermost loop.

3. We introduce a method by which *the heap algorithm can track the upper bound* for each point,

allowing it to reach the same number of distance calculations as Hamerly's algorithm.

4. We show how to *accelerate the first iteration* of k-means (which typically incurs many distance calculations) by using the initial assignment and the neighbor centroid concept.

In Section 2 we briefly discuss related work on accelerating k-means using distance bounds and the triangle inequality. In Section 3 we present new methods that further speedup these state-of-the-art algorithms. In Section 4 we present experiments that demonstrate the performance of our methods and we conclude in Section 5.

2 Prior algorithms overview

Elkan's algorithm [4] uses one upper bound per point, which represents a bound on the distance from point x to its assigned (closest) centroid. There are also k lower bounds per point on the distance from x to each centroid. Between iterations the bounds are updated based on the triangle inequality. For each lower bound update we assume that the centroid moves toward point x and for the upper bound update we assume that the assigned centroid moves away from x. If the upper bound between point x and its assigned centroid becomes larger than the lower bound on the distance between x and some other centroid, the distance between x and this centroid is recalculated and the assignment changed as appropriate.

Hamerly's algorithm [5] is a simplification of Elkan's algorithm. It has only one lower bound per point, which is a lower bound on the distance between point x and its *second* closest centroid. If the lower bound is smaller than the upper bound for some point x, we have to recalculate k distances. Revisiting all k centroids is more costly than necessary, as this paper shows in Section 3.2, as well as by previous work by the annulus algorithm [6]. Both methods restrict the set of centroids searched to something possibly much smaller than k. The annulus algorithm searches only those inside the origin-centered annulus containing point x, which is often fewer than k. This paper defines 'neighbor' centroids, allowing us to ignore non-neighbors.

The heap algorithm [6] combines the upper bound and Hamerly's lower bound into a single value representing their difference. As bound updates are the same for all points in a cluster whose assignment remains the same, we need to store only one such value per point, and one value containing the accumulated movement of centroids for each cluster. Storing those bound difference values in a heap allows us to consider only those points whose bounds are violated (lower < upper), avoiding iterating over the many points whose bounds are not violated.

3 Proposed methods for speedup of k-means

In this section we discuss four methods for improving accelerated k-means methods: tighter updates for lower bounds, iterating over fewer than k centroids when bounds are violated, maintaining upper bounds in the heap algorithm, and avoiding distance calculations in the first iteration.

3.1 A tighter lower bound update When we use the triangle inequality to update lower (upper) bounds in any of the above algorithms, we assume that the centroid moves directly toward (away from) the point. But this is obviously not always true, making the bound update often too pessimistic, especially in high dimension (since two high dimensional vectors are far more likely to be orthogonal than collinear). If some centroid c_j moves away from centroid c_i, the lower bound for point x (assigned to c_i) can theoretically even grow rather than shrink. Using tighter lower bound updates would allow the bounds to hold for more iterations and avoid more distance calculations.

We propose a tighter update for the lower bound on the distance between point x and centroid c_j. To achieve this goal we maintain $m(c_i)$, which is the radius of a hypersphere centered at centroid c_i that contains all points assigned to c_i. (Note that $m(c_i)$ is easily obtained as the maximum upper-bound of all points in the cluster.)

Assume that we have a lower bound $l(x, c_j) \leq \|x - c_j\|$. Also assume that point x is assigned to centroid c_i. Let c'_j be the new location of centroid c_j in the next iteration. Between the current and next iteration we need to update the lower bound $l(x, c_j)$. Any update $\delta(x, c_j)$ that fulfills the condition

$$\delta(x, c_j) \geq \|x - c_j\| - \|x - c'_j\| \tag{3.1}$$

is valid as it implies

$$\begin{aligned} l(x, c'_j) &= l(x, c_j) - \delta(x, c_j) \\ &\leq \|x - c_j\| - \big(\|x - c_j\| - \|x - c'_j\|\big) = \|x - c'_j\|, \end{aligned}$$

which is the condition we need to maintain across iterations. From now we will refer to the right hand side of Inequality (3.1) as $f(x)$.

For simplicity of explanation we consider the case where dimension is 2; later we will generalize. To simplify the calculations we will assume that $c_j = (0, 1)$ and $c'_j = (0, -1)$. The following lemma (illustrated in Figure 1) gives us an idea of how to find a valid update $\delta(x, c_j)$. First, let us denote $u(x)$ the upper

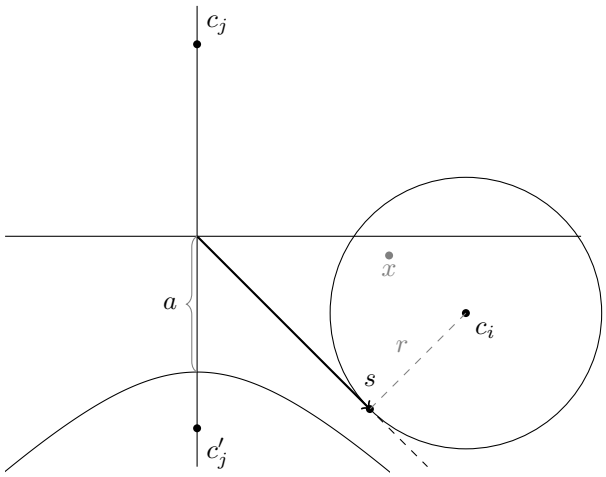

Figure 1: Illustration of Lemma 3.1.

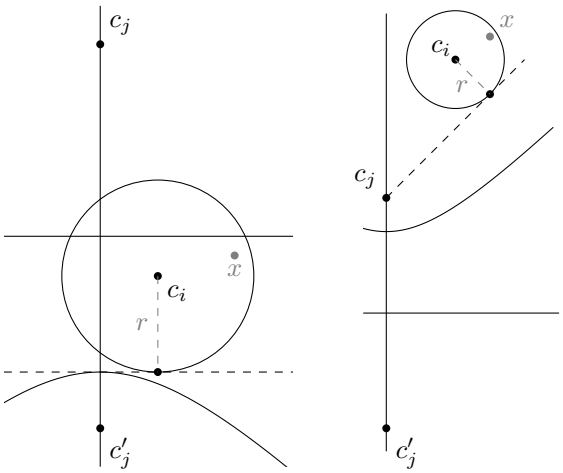

Figure 2: Special cases when conditions of Lemma 3.1 are violated. If the circle intersects the vertical axis, we can bound by a horizontal line that touches the hyperbola and the circle. If the circle lies above the horizontal axis, we can bound by a line that is parallel to the asymptote and goes through c_j.

bound on distance between x and its closest centroid, i.e. $u(x) \geq \|x - c(x)\|$. We assume that the point x is located in a circle centered at c_i with radius r.

LEMMA 3.1. *Suppose that $x \in \mathbb{R}^2$, $u(x) \leq r \in \mathbb{R}_0^+$ and $c_i = (c_{ix}, c_{iy})$, where $c_{ix} > r$ and $c_{iy} \leq r$. Let $c_j = (0, 1)$ and $c'_j = (0, -1)$. Then*

$$\delta(x, c_j) = 2\frac{c_{ix}r - c_{iy}\sqrt{\|c_i\|^2 - r^2}}{\|c_i\|^2} \tag{3.2}$$

is a valid update of lower bound in the meaning of Inequality (3.1).

Proof. Refer to Figure 1. The conditions say that point x lies somewhere in the circle with center c_i and radius r. This circle does not intersect the vertical axis and there is at least one point on the circle that has nonpositive y coordinate.

Now consider the curve $\{x \mid f(x) = z\}$ for some fixed $0 \leq z \leq 2$. The curve is a hyperbola with focal points c_j and c'_j,[1] and z is the difference in old and new distances from c_j and c'_j to x. The range $0 \leq z \leq 2$ indicates that the hyperbola lies below the horizontal axis. When $f(x) = 0$, the old and new distances are the same (c_j moved no closer, thus the lower bound need not be reduced), and if $f(x) = 2$, then c_j moved directly toward x (so the lower bound needs to be reduced). If we consider any point x on *or above* the hyperbola (i.e. a point x for that $f(x) \leq z$), we see that $\delta(x, c_j) = z$ is a proper update of $l(x, c_j)$ in the meaning of Inequality (3.1), as $\delta(x, c_j) = z \geq f(x)$.

[1]Recall that one of the ways how to define a hyperbola is as a set of points in the plane that have the same difference of the distances to two focal points.

We could find the value of z that causes the hyperbola to just touch the circle containing x, but we use a simpler approach that produces a slightly less-tight but still valid bound update. Notice that for positive z the hyperbola lies below its asymptote. If the asymptote touches the circle containing x and the whole circle lies above the asymptote, the whole circle lies also above the hyperbola. From the previous paragraph we know that $\delta(x, c_j) = z$ is a valid update of $l(x, c_j)$ for any point x on or above the hyperbola.

Denote by s the point where the asymptote touches the circle. It must be true that

$$\|c_i - s\|^2 = r^2, \tag{3.3}$$

$$(s - c_i)^T s = 0. \tag{3.4}$$

From (3.4) we see that $\|s\|^2 = c_i^T s$. Plugging this into (3.3) leads to

$$\|s\|^2 = \|c_i\|^2 - r^2. \tag{3.5}$$

Equation (3.4) leads to $s_x = \frac{\|s\|^2 - c_{iy}s_y}{c_{ix}}$. Plugging this into $\|s\|^2 = s_x^2 + s_y^2$ leads to a quadratic equation w.r.t. s_y:

$$s_y^2\|c_i\|^2 \; - \; s_y(2c_{iy}\|s\|^2) \; + \; (\|s\|^4 - \|s\|^2 c_{ix}^2) = 0.$$

We are interested in the solution with the smaller y

coordinate, which is

$$s_y = \frac{c_{iy}\|s\|^2 - \sqrt{c_{iy}^2\|s\|^4 - \|c_i\|^2(\|s\|^4 - \|s\|^2 c_{ix}^2)}}{\|c_i\|^2} = \frac{c_{iy}\|s\|^2 - c_{ix}\|s\|r}{\|c_i\|^2}.$$

The last step is based on Equation (3.5). Consider the top point of the hyperbola and denote a as its distance to the origin. The distance from this point to c_j is $1+a$ and the distance to c'_j is $1-a$. Therefore the $f(x)$ is equal to $1+a-(1-a) = 2a$ for each point on the hyperbola. This means that the update will be equal to $2a$. As the distance from each focal point to the origin is one, the properties of a hyperbola imply that

$$\delta(x, c_j) = f(x) = 2a = -2\frac{s_y}{\|s\|} = 2\frac{c_{ix}r - c_{iy}\|s\|}{\|c_i\|^2},$$

which is equivalent to Equation (3.2), which we had to prove. ■

If condition $c_{ix} > r$ is violated, it means that the circle intersects the vertical axis. Then the update computed using the methods just described does not work. The whole circle lies above a horizontal line with y coordinate $c_{iy} - r$. If $c_{iy} - r < -1$ we use $\delta(x, c_j) = 2$, which comes from the triangle inequality. If $c_{iy} - r \in [-1, 0]$, we can consider a hyperbola with $a = r - c_{iy}$ and use update $\delta(x, c_j) = 2(r - c_{iy})$.

If condition $c_{iy} \leq r$ is violated, then the whole circle lies above the horizontal axis. Therefore we can either use $\delta(x, c_j) = 0$ or we can repeat all the calculations, but we bound with line that is parallel to the asymptote and goes through point c_j. If both conditions are violated we use $\delta(x, c_j) = 0$. Those cases are illustrated in Figure 2.

We have to deal with higher dimensions than two. The following theorem provides a tool that allows us to solve the problem in higher dimension using the simplified two-dimensional case. This means that we can restrict ourselves to just the 2-dimensional plane defined by points c_i, c_j and c'_j.

THEOREM 3.1. *Suppose that we project all points x and all centroids onto the plane defined by points c_i, c_j and c'_j, so that the requirements of Lemma 3.1 hold. Then the lower bound update calculated using Lemma 3.1 is correct but for the scale given by the projection.*

Proof. In higher dimension the circle around c_i becomes a hypersphere and the hyperbola with focal points c_j and c'_j becomes a rotational hyperbola with the same focal points. In other words it is rotated around the axis defined by points c_j and c'_j. The asymptote is a multidimensional cone instead of a line.

Suppose that $\delta(x, c_j)$ is the update calculated by Lemma 3.1. From proof of this lemma it follows that the update is valid for any point that lies outside the rotated hyperbola defined by the update value. For points that are inside the rotated hyperbola this update is too low. The points for which we calculate the update lie in the hypersphere centered at c_i.

Consider a hyperplane tangent at point s to the hypersphere centered at c_i. This hyperplane contains vector s and is orthogonal to the plane defined by c_i, c_j and c'_j. The hypersphere and the cone lie on the opposite sides of the hyperplane. This follows that there cannot be any point x for that the update $\delta(x, c_j)$ would be invalid (i.e. too low) as it would require the point to be on both sides of the hyperplane. ■

To obtain the correct solution in higher dimension, we calculate coordinates of c_i using its distance from the line defined by c_j and c'_j and using its projection to that line. The projection can be calculated as

$$P(c_i) = c_j + \frac{(c_i - c_j)^T(c'_j - c_j)}{\|c'_j - c_j\|^2}(c'_j - c_j). \tag{3.6}$$

We scale the coordinates so that they match the conditions of Lemma 3.1. Therefore we need to scale in both dimensions by dividing by $\frac{\|c_j - c_j\|}{2}$. The x coordinate of the transformed point c_i (c_{ix}) is the distance of c_i from the line through c_j and c'_j. The y coordinate (c_{iy}) can be calculated from the projection. Also we need to project the radius of the sphere by dividing by $\frac{\|c'_j - c_j\|}{2}$. This follows that the new coordinates are

$$c_{ix} = 2\frac{\|P(c_i) - c_i\|}{\|c'_j - c_j\|}, \tag{3.7}$$

$$c_{iy} = 1 - 2\frac{(c_i - c_j)^T(c'_j - c_j)}{\|c'_j - c_j\|^2}, \tag{3.8}$$

$$r = 2\frac{m(c_i)}{\|c'_j - c_j\|}. \tag{3.9}$$

Now we can calculate the update using Lemma 3.1. Finally we project back to get the correct update by multiplying with $\frac{\|c'_j - c_j\|}{2}$. For better performance we pre-calculate once per iteration $\|c_i\|^2$, $\|c'_i\|^2$ and $c_i^T c'_i$ as those are used repeatedly during projection calculation.

Calculating such a lower bound update for each point x would be too costly. But we can use the common radius r for all points assigned to centroid c_i, which is the maximum upper bound $m(c_i)$. As a result we do all the calculations once for each pair of centroids each iteration.

To Elkan's algorithm we can apply the change immediately as we do the calculation once per bound.

For the Hamerly, heap, and annulus algorithms we can take (as the update of the lower bound for points assigned to c_i) the maximum of updates if we consider as c_j each other centroid. But we can do better if we consider only *neighbors* of the cluster (see Section 3.2). We can also skip centroids that have moved less than the current maximum. This condition is more useful if we sort the centroids by movement. Algorithms 1, 2 and 3 show all the parts of this proposal connected together.

Algorithm 1 Algorithm for update of $l(x, c_j)$ in the simplified two dimensional case, where $c(x) = c_i$.

Input: Centroid $c_i \in \mathbb{R}^2$, radius r of the cluster c_i
Output: $\delta(x, c_j)$ for all x assigned to c_i
if $c_{ix} \leq r$ **then return** $\max\{0, \min\{2, 2(r - c_{iy})\}\}$
▷ when line through c_j and c'_j intersects the sphere
if $c_{iy} > r$ **then**
    $c_{iy} \leftarrow c_{iy} - 1$ ▷ handle negative update
end if
return $2\frac{c_{ix}r - c_{iy}\sqrt{\|c_i\|^2 - r^2}}{\|c_i\|^2}$ ▷ (3.2)

Algorithm 2 Algorithm for update of $l(x, c_j)$ in the multidimensional case, where $c(x) = c_i$.

Input: Two (different) centroids c_i and c_j
Output: $\delta(x, c_j)$ for all x assigned to c_i
$t \leftarrow \frac{(c_i - c_j)^T(c'_j - c_j)}{\|c'_j - c_j\|^2}$ ▷ (3.6)
$dist \leftarrow \|c_j + t \cdot (c'_j - c_j) - c_i\|$ ▷ $\|P(c_i) - c_i\|$
$c_{ix} \leftarrow dist \cdot \frac{2}{\|c'_j - c_j\|}$ ▷ (3.7)
$c_{iy} \leftarrow 1 - 2t$ ▷ (3.8)
$r \leftarrow m(c_i) \cdot \frac{2}{\|c'_j - c_j\|}$ ▷ (3.9)
return update calculated by Algorithm 1 (using r and $c_i = (c_{ix}, c_{iy})$) multiplied by $\frac{\|c'_j - c_j\|}{2}$

3.2 Iteration over neighboring centroids Sort-means and compare-means [12] exploit conditions that eliminate distance calculation between a point and some centroids in the innermost loop. However, the Hamerly and heap algorithms have to recalculate all k distances when bounds tell us that an assignment may have changed. One straightforward speedup is introducing a condition from compare-means to skip centroids in the algorithm's innermost loop. Compare-means skips the distance calculation between point x and centroid c_i if there is some other centroid c_j that must be closer to x than c_i. This requires knowledge of centroid-centroid distances and distance between c_i and x.

The idea of this proposal is to reduce the number of centroids considered in the innermost loop. Compare-means has to evaluate the condition for each centroid and therefore requires $\mathcal{O}(k)$ work. Sort-means can break out of the innermost loop, but still has to evaluate a condition per visited centroid. We propose a method that allows Hamerly's algorithm or the heap algorithm to iterate only over selected centroids, named neighbors, but without introducing any additional work to the innermost loop.

Algorithm 3 Algorithm for update of $l(x)$ in Hamerly's, heap or annular algorithm, where $c(x) = c_i$.

Input: Centroid c_i
Output: $l(x)$ update for all x assigned to c_i
$update \leftarrow -\infty$
for each c_j **in** centroids that fulfill (3.10) in decreasing order of $\|c'_j - c_j\|$ **do**
    if $\|c'_j - c_j\| \leq update$ **then break**
    $update \leftarrow \max\{$update calculated by Algorithm 2 using c_i and c_j, $update\}$
end for
return $update$

DEFINITION 1. *A* neighbor *of a centroid c_i is any centroid c_j that is closest or second closest to any point assigned to c_i.*

It is clear that we can iterate over only neighbor centroids of c_i in the innermost loop of Hamerly's algorithm or the heap algorithm as both algorithms need the distance to the closest and second closest centroid. To identify which are the neighbors, we would have to do the same work as the original algorithm. But without influencing the solution we can iterate over any superset of the set of neighbors. To get such a set of centroids, we need three things:

1. the set of centroid-centroid distances,
2. the distance between each centroid and its closest other centroid, denoted $2s(c_i)$ (Elkan introduced $s(c_i)$ as half the distance between centroid c_i and its closest other centroid), and
3. the maximum upper bound per cluster, denoted $m(c_i)$.

In Hamerly's algorithm we can identify the maximum upper bound simply by iterating over all points of the cluster and finding the maximum of upper bounds. But for the heap algorithm this is not straightforward and we present a new way to do this in Section 3.3.

THEOREM 3.2. *Any neighbor c_j of centroid c_i must fulfill the inequality*

$$m(c_i) + s(c_i) \geq \frac{1}{2}\|c_i - c_j\|. \tag{3.10}$$

Proof. Suppose that c_c is the closest other centroid to c_i. In the proof we will show an equivalent claim, i.e. that if c_j fulfills the condition $m(c_i)+s(c_i) < \frac{1}{2}\|c_i-c_j\|$, then c_j cannot be a neighbor of c_i because for all points x assigned to centroid c_i it holds that c_i and c_c are closer to x than c_j. We know that $\|x - c_i\| \leq u(x)$, so $\|x - c_i\| \leq \max_{y|c(y)=c_i} u(y) = m(c_i)$.

For the distance between c_i and x it holds that

$$\begin{aligned}\|x - c_i\| \leq m(c_i) \leq m(c_i) + s(c_i) < \frac{1}{2}\|c_i - c_j\| \\ \leq \frac{1}{2}\left(\|x - c_i\| + \|x - c_j\|\right),\end{aligned}$$

from which it immediately follows that $\|x - c_i\| < \|x - c_j\|$.

Similarly we prove that c_c is closer to x than c_j

$$\begin{aligned}\|x - c_c\| &\leq \|x - c_i\| + \|c_i - c_c\| \leq m(c_i) + 2s(c_i) \\ &= 2\left(m(c_i) + s(c_i)\right) - m(c_i) \\ &< \|c_i - c_j\| - m(c_i) \\ &\leq \|x - c_i\| + \|x - c_j\| - m(c_i) \\ &\leq m(c_i) + \|x - c_j\| - m(c_i) = \|x - c_j\|.\end{aligned}$$

Therefore c_i and c_c are both closer to any x (assigned to c_i) than c_j, which means that c_j cannot be a neighbor of c_i. ∎

In the implementation we need to make sure that the upper bound is valid. The appropriate place to calculate the maximum upper bound is before any centroids move. Therefore after moving centroids we need to increase the maximum upper bound by the movement of the cluster's centroid.

We note that Inequality (3.10) is independent of the point x. Therefore we need to evaluate it only once per iteration. Then in the inner loop we iterate over centroids that fulfill Inequality (3.10).

For Elkan's algorithm, instead of Inequality (3.10), we can use the stronger condition $m(c_i) \geq \frac{1}{2}\|c_i - c_j\|$ as we need to know only the distance to the closest centroid.

3.3 Explicit upper bounds for the heap algorithm One problem (previously noted by [6]) of combining upper and lower bound into one value representing their difference (as in the heap algorithm) is the fact that when the bounds are violated, such an algorithm can't tighten the upper bound and possibly avoid calculating k distances. We propose a method that fixes this problem without affecting the time complexity of the heap algorithm and at a memory cost of $k + n$ distance values stored. As a result, the heap algorithm can achieve the same number of distance calculations as Hamerly's algorithm.

The first step is to introduce an array with upper bound for each point. This also allows us to get the lower bound using the key of the heap. Then after tightening the upper bound, we might be able to avoid calculation of distances to the $k-1$ centroids as it is done in Hamerly's algorithm. But a naïve implementation of this optimization would mean iteration over n points to update the upper bounds, which ruins the time complexity of the heap algorithm.

Therefore we propose the same method used for the heap key. Instead of storing value $u(x)$, we store $u(x) - p_t(c(x))$, where $p_t(c_i)$ is the length of the path travelled by centroid c_i up to iteration t. Using this value allows us to update only k values per iteration and change the upper bound $u(x)$ for one particular point x only when the assignment of x changes.

When we need to find the maximum upper bound for a cluster, we cannot iterate over all points as it violates the time complexity. But we can introduce a max heap with key $u(x) - p_t(c(x))$, which allows us to get the maximum upper bound for each cluster. Knowledge of the maximum upper bound is required for other optimizations presented in this paper.

3.4 Finding neighbors in the first iteration The last modification we propose is applicable only if the initial assignment is of high quality, i.e. it forms tight clusters without outliers in the area occupied by other clusters. For example k-means++ [1] produces a good assignment. Our experiments have shown that for some datasets, accelerated algorithms make more than 80% of their distance calculations during the first iteration. If this is the case, the runtime can be further improved by the following procedure.

As the distance between a point and its assigned centroid is always needed, we can calculate it separately and find the maximum upper bound per cluster. This allows us to use the approach in Section 3.2 to eliminate some of the distance calculations. Also we can initialize the lower bound to $2s(c_i) - u(x)$ when point x assigned to c_i fulfills the condition $u(x) \leq s(c_i)$.

4 Experiments and Discussion

We run our experiments on several types of datasets. For each algorithm selected we run its original version together with one or more versions modified using the

proposals from Section 3. Table 1 shows the variants of algorithms that we test. For each algorithm we measured time (CPU seconds) needed to obtain the result and also the architecture-independent number of distance calculations. We separately measured the number of additional inner products needed to obtain distances and projections in Section 3.1. We also compared the results with Lloyd's original algorithm.

All algorithms have taken as an input the same initialization produced by k-means++ [1]. All implementations were written in C++ with shared common code, which is available on `http://cs.ecs.baylor.edu/~hamerly/software/kmeans.php`. The tests ran on a 24-core 2.3 GHz 64-bit Linux Intel platform with 16 GB RAM. All implementations were single threaded, though the code base supports multithreading. Table 3 and Figures 3 and 4 show the results. The runtime results are averaged over 10 repeats of the algorithms with the same initializations.

Table 2 shows the synthetic and real-world datasets we used. For space reasons we show results in this paper on only four representative datasets, namely Clustered-5, Uniform-2, BIRCH and MNIST-50. The remaining may be found at the URL given above.

Table 3 shows relative speedup with respect to Lloyd's algorithm. While the proposed changes have minimal or even negative effects on Elkan's algorithm, they speed up the heap algorithm over eight times on the BIRCH dataset. Similarly, Hamerly's algorithm becomes more than seven times faster. On the clustered dataset we have seen speedups around three times in the case of Hamerly's algorithm and four times in the case of the heap algorithm. This ratio gets smaller with increasing dimension; the changes have hardly any effect on the higher-dimension MNIST-784 and Covertype datasets. Those contain discrete or binary features which may partly cause the difference as Euclidean distance is not as appropriate a similarity measure. On uniform datasets the improvements are smaller than on clustered datasets; however clustering data with uniform distribution is less interesting from a practical perspective.

Figure 3 shows the number of distances (point-centroid or centroid-centroid) that are calculated by the algorithms. Again in the case of Elkan's algorithm the difference is not big, but in the case of Hamerly's algorithm and the heap algorithm we are able to save more than 90 % of distance calculations (on the BIRCH dataset).

Of interest is the number of inner products made by the method presented in 3.1. This number is low relative to the number of distance calculations. Therefore the overhead of the proposed method is low and the runtime does not increase much.

Figure 4 shows average lower bound updates per bound. As this value is data dependent, we divided each update by the maximal update found in each dataset. As expected, Hamerly's algorithm, the heap algorithm and the annulus algorithm require the same updates. Elkan's algorithm has more lower bounds, so its average update is smaller. After modifications the average update decreased to approximately one fourth in the case of the clustered and BIRCH datasets. In the case of the MNIST dataset the average update decreased by half. This proves that our modifications were correct and lower bound changed more slowly than in the original algorithms.

The methods proposed in Sections 3.1 and 3.2 provide similar speedups. The method of 3.2 provides better results on most of the datasets. This change is also simpler to implement. These two methods are to some degree independent, which means that implementing both methods leads to better results than implementing only one of the improvements. From the experimental results we see that those two proposals work best if the data are clustered, which was our assumption when we bounded each cluster in a hypersphere. The main advantage of the method in Section 3.3 is that it makes the heap algorithm competitive with Hamerly's algorithm by dropping the number of distance calculations to the same level. The method in Section 3.4 provides the least speedup.

We can explicitly evaluate memory overhead of the proposed methods. Section 3.1 requires an additional $k \cdot d + 6k$ values to be stored in the case of Hamerly's, the annulus, or Elkan's algorithms. The memory is used to save old centroid locations and to cache norms and inner products in order to avoid repeated calculations. The heap algorithm requires the changes in Section 3.3 to be implemented for an effective way of obtaining $m(c_i)$. The method of Section 3.2 requires an additional $k^2 + 2k$ values, or only k^2 if Section 3.1 is already implemented. Section 3.3 requires an additional $n + k$ values and also k heaps in the case that we need to calculate $m(c_i)$. The method of Section 3.4 does not require any additional memory if we implement Section 3.1.

5 Conclusion

Clustering algorithms are used in many machine learning systems, so they should be fast – especially since standard practice for finding good clusterings is to try multiple runs with multiple initializations. In this paper we have shown four ways to further accelerate state-of-the-art k-means clustering algorithms, without significant increases in code complexity or memory costs. The calculations needed for changes in Sections 3.1 and 3.2

Table 1: Algorithm versions and their modifications.

Proposal	Lloyd	Elkan	ElkanA	ElkanB	Hamerly	HamerlyA	HamerlyB	HamerlyC	Heap	HeapA	HeapB	Annulus	AnnulusA
3.1			✓	✓		✓	✓	✓		✓	✓		✓
3.2				✓			✓	✓			✓		
3.3										✓	✓		
3.4								✓					

Table 2: Datasets used in the experiments.

Name	Description	Points n	Dimension d	k
Uniform-d	Synthetic data, uniform distribution	1 000 000	2/3/5/7/10/15	50
Clustered-d	Synthetic data, 50 clusters of similar size	1 000 000	2/3/5/7/10/15	50
BIRCH [13]	Synthetic grid 10×10 Gaussian clusters	100 000	2	100
Covertype [2]	Forest CoverType dataset	581 012	54	8
MNIST-784 [8]	Handwritten digit images 28×28 px	60 000	784	10
MNIST-50	Random linear projection of MNIST-784	60 000	50	10

Table 3: Relative speedup with respect to Lloyd's algorithm.

Dataset	Elkan	ElkanA	ElkanB	Hamerly	HamerlyA	HamerlyB	HamerlyC	Heap	HeapA	HeapB	Annulus	AnnulusA
Clustered-2	6.3	7	*9*	26.6	42.7	77	**79.3**	12.9	33.3	*47.9*	54	*69.3*
Clustered-5	10.3	10.2	*11.6*	107.4	147.3	193.5	*205.1*	78	228.5	**319.4**	91.7	*140*
Uniform-2	5.3	5	*7.1*	31.1	42.6	79	**80**	19.8	28.9	*39*	60.6	*69.5*
BIRCH	7.8	7.5	*10.5*	14.5	30.4	77.7	**104.7**	6.4	27.5	*56.9*	49.9	*70.4*
Covertype	**8.8**	8.8	8.7	5.4	5.4	*5.5*	5.5	4.1	4.2	*4.2*	*2.9*	2.9
MNIST-50	29.9	29.2	*38.6*	25.2	34.5	46.8	**47.2**	15.8	28.7	*36.2*	41.9	*46.4*

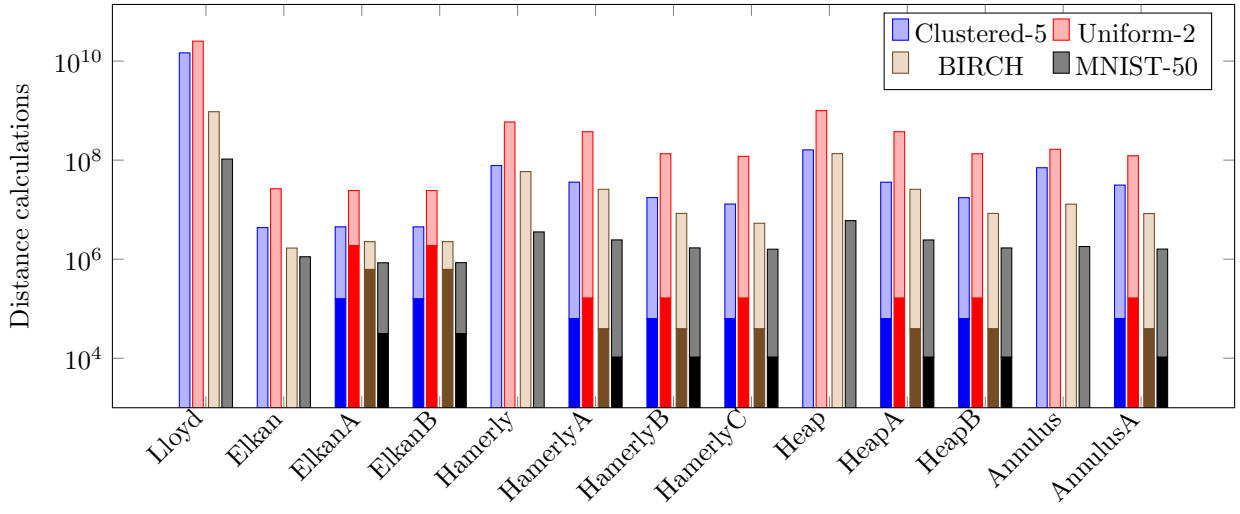

Figure 3: Number of distance calculations made by different algorithms. Note that the vertical axis uses a logarithmic scale. The bottom stacks of the bars show the number of inner products needed by proposals presented in this paper.

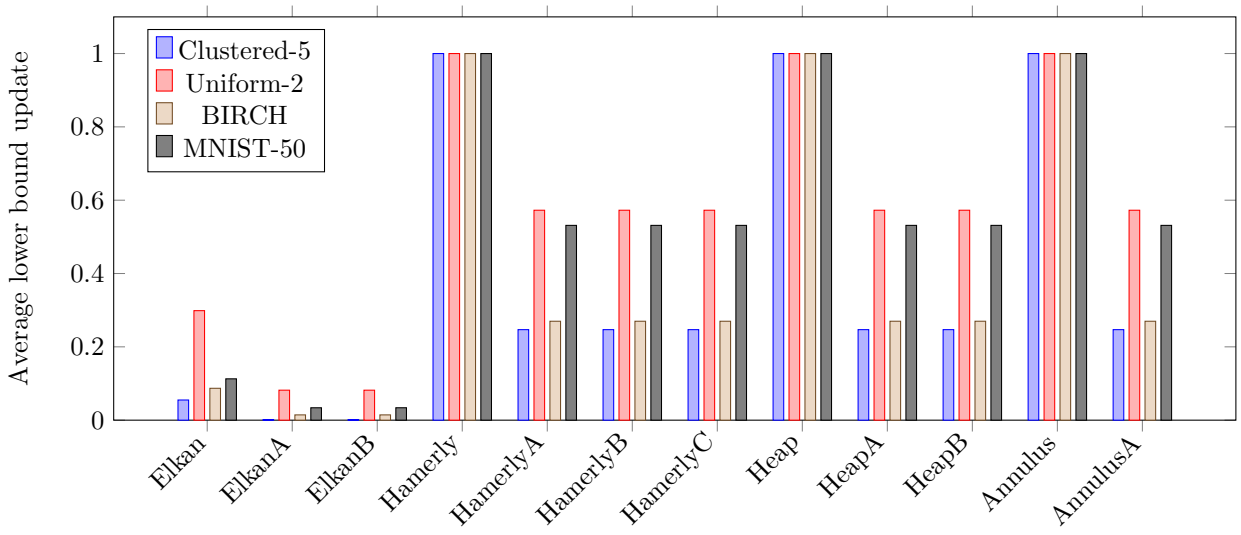

Figure 4: Average lower bound update per bound. The values depend on magnitude of data and are therefore normalized.

are done only once per iteration. Therefore the overhead of those methods is low, while they may save a significant amount of time and distance calculations. We also show how to make change of Section 3.2 applicable to the first iteration.

Our changes work well on clearly clustered datasets. The speedup relative to the original versions of (already accelerated) algorithms was typically three or four times for the heap and Hamerly algorithms. Speedup of the annulus algorithm was less than two, but on Elkan's algorithm the impact was small. The proposed changes did not improve runtime on high dimension datasets and datasets without a natural clustering.

References

[1] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[2] Jock A. Blackard, Denis J. Dean, and Charles W. Anderson. Covertype data set. `https://archive.ics.uci.edu/ml/datasets/Covertype`, 1998.

[3] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 579–587, 2015.

[4] Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning*, volume 3, pages 147–153, 2003.

[5] Greg Hamerly. Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining (SDM 2010)*, pages 130–140, 2010.

[6] Greg Hamerly and Jonathan Drake. Accelerating Lloyds algorithm for k-means clustering. In *Partitional Clustering Algorithms*, pages 41–78. Springer, 2015.

[7] 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. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892, 2002.

[8] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. Mnist handwritten digit database. `http://yann.lecun.com/exdb/mnist/`, 1998.

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

[10] Andrew W Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 397–405. Morgan Kaufmann Publishers Inc., 2000.

[11] Dan Pelleg and Andrew Moore. Accelerating exact k-means algorithms with geometric reasoning. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 277–281. ACM, 1999.

[12] Steven J Phillips. Acceleration of k-means and related clustering algorithms. In *Algorithm Engineering and Experiments*, pages 166–177. Springer, 2002.

[13] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH clustering datasets. `http://cs.joensuu.fi/sipu/datasets/`, 1997.