# Chapter 2
# Accelerating Lloyd's Algorithm for *k*-Means Clustering

**Greg Hamerly and Jonathan Drake**

**Abstract** The $k$-means clustering algorithm, a staple of data mining and unsupervised learning, is popular because it is simple to implement, fast, easily parallelized, and offers intuitive results. Lloyd's algorithm is the standard batch, hill-climbing approach for minimizing the $k$-means optimization criterion. It spends a vast majority of its time computing distances between each of the $k$ cluster centers and the $n$ data points. It turns out that much of this work is unnecessary, because points usually stay in the same clusters after the first few iterations. In the last decade researchers have developed a number of optimizations to speed up Lloyd's algorithm for both low- and high-dimensional data.

In this chapter we survey some of these optimizations and present new ones. In particular we focus on those which avoid distance calculations by the triangle inequality. By caching known distances and updating them efficiently with the triangle inequality, these algorithms can provably avoid many unnecessary distance calculations. All the optimizations examined produce the same results as Lloyd's algorithm given the same input and initialization, so are suitable as drop-in replacements. These new algorithms can run many times faster and compute far fewer distances than the standard unoptimized implementation. In our experiments, it is common to see speedups of over 30–50x compared to Lloyd's algorithm. We examine the trade-offs for using these methods with respect to the number of examples $n$, dimensions $d$, clusters $k$, and structure of the data.

**Keywords** k-Means • Triangle inequality • Caching • Accelerate • Lloyd's algorithm • Clustering • Unsupervised learning

G. Hamerly (✉)
Baylor University, 105 Baylor Ave., Waco, TX 76798, USA
e-mail: hamerly@cs.baylor.edu

J. Drake
Hewlett-Packard Company, 14231 Tandem Blvd, Austin, TX 78728, USA
e-mail: jonathan.drake@hp.com

## 2.1 Introduction

The $k$-means clustering algorithm is a very popular tool for data analysis and learning. At its heart is an easily-understood optimization problem: given a set of data points (in some vector space), try to position $k$ other points (called 'centers') at locations that minimize the (squared) distance between each point and its closest center. While it is popular and easy to implement, the naive implementation for solving the problem is inefficient, wasting a lot of processing time on unnecessary and redundant computations.

This chapter discusses simple geometric methods, based on the triangle inequality and keeping cached bounds on computed distances, to reduce wasted computation and build much more efficient algorithms that give exactly the same output. We point out that all the accelerated algorithms we investigate in this chapter give exactly the same answer as Lloyd's standard batch algorithm, given the same initialization. In our experiments, it is common to see speedups of over 30–50x compared to Lloyd's algorithm.

### *2.1.1 Popularity of the k-Means Algorithm*

The $k$-means algorithm is very widely used. In [40], it is chosen as one of the top ten data mining algorithms. It is implemented in many commercial and open-source statistical data analysis software packages, including MATLAB, SAS, Stata, SPSS, R, and Weka to name a few. A simple search for 'k means clustering' on Google Scholar yields over 2.1 million results, which is greater than the number of results for 'neural network', 'support vector machine', 'nearest neighbor', or 'logistic regression'.

Many applications benefit from using $k$-means. Just a few are:

- clustering the pixels of an image for image color quantization [7, 19],
- post-processing to decide the memberships in spectral clustering [29],
- selecting the codewords for vector quantization, enabling lossy compression of audio or image data [23],
- image segmentation [14, 19],
- unsupervised feature learning in single-layer neural networks [9],
- identifying self-similar behaviors in dynamic program execution for structured sampling of the behaviors [36],
- finding a good initialization for a more costly learning method [6], and
- finding good locations for basis functions in a radial basis function network [39].

Such a widely-used algorithm deserves to be well-studied and efficiently implemented.

### 2.1.2 The Standard *k*-Means Algorithm does a Lot of Unnecessary Work

The $k$-means algorithm is popular due to its clarity, simplicity, and intuitive optimization function. As a result, it has been implemented, albeit inefficiently, many times over. We investigated the source code of the $k$-means implementations for the software packages ELKI, graphlab, Mahout, MATLAB, MLPACK, Octave, OpenCV, R, SciPy, Weka, and Yael, and found that none of them use the triangle inequality bound acceleration techniques that we discuss in this chapter, though they would benefit from doing so. Of those, the ones which provide scalable implementations primarily depend on some form of parallel processing, which are compatible with the acceleration methods presented here.

The standard methods for solving the $k$-means optimization problem are Lloyd's algorithm [24] (a batch algorithm, also known as Lloyd-Forgy [13]), as well as MacQueen's algorithm [26]. Each algorithm spends the vast majority of its time computing distances between the clustered points and the current cluster centers. However, much of the time, these distance calculations are unnecessary, wasted computation. In this study we focus on improving Lloyd's algorithm, which is widely used.

The basic reason why the standard batch methods for optimizing $k$-means are inefficient is because in each iteration they must identify the closest center for each clustered point. To do this, the methods naively compute all $nk$ distances between each of the $n$ clustered points and each of the $k$ centers. After each iteration, the centers move and these distances may all change, requiring recomputation. But typically the centers don't move much, especially after the first few iterations. Most of the time the closest center in the previous iteration remains the closest center. Thus, keeping track of the closest center for each clustered point is made much more efficient with some caching. When the closest center doesn't change, ideally we shouldn't need to compute the distance between that point and *any* cluster center. And even when the closest center for a point changes, it might be possible to avoid computing the distance from that point to all centers, instead looking only at a few centers that are guaranteed to be closer to that point than all other centers.

### 2.1.3 Previous Work on *k*-Means Acceleration

There is a healthy line of research on accelerating learning algorithms, which goes hand in hand with accelerating data retrieval algorithms such as $k$-nearest neighbor search. The primary methods of acceleration are: algorithmic improvements, parallelization (including threading, multiprocessing, and distributed computation), and approximation. In this chapter, we focus on algorithmic improvements which give exact answers.

#### 2.1.3.1 Algorithmic Improvements

Pelleg and Moore [31] incorporated the popular $k$-d tree data in the task of accelerating the $k$-means method (note that the $k$ in $k$-d tree is a naming clash with the $k$ in $k$-means). Kanungo et al. [19] developed a similar algorithm. Both algorithms use the $k$-d tree to structure the data to be clustered. By restructuring the search for each point's closest center, many distance calculations can be avoided. While these approaches are excellent in low dimension, $k$-d trees perform poorly in dimensions much greater than 8.

Moore [28] developed the anchors hierarchy, a new type of spatial data structure based on metric trees. Applying the triangle inequality, this structure organizes the data by enclosing it in hierarchically organized anchors with associated radii. The anchors hierarchy allows the $k$-means algorithm to avoid many provably unnecessary distance calculations, even in high dimension. We discuss this structure more in Sect. 2.3.2.

Elkan [12] started a line of research which pairs the triangle inequality directly with cached distance bounds to avoid unnecessary point-center distance calculations. Avoiding complicated hierarchical data structures and preprocessing, his algorithm simply caches $O(nk)$ distance bounds to prune distance calculations. It uses the triangle inequality to efficiently updates the bounds each time centers move. Hamerly [15] reduced this overhead to $O(n)$ bounds, which makes it much faster in practice for low and medium-dimension datasets. Drake [11] bridged the gap between these two approaches by using an adaptive number of distance bounds, $O(nb)$ where $b < k$ and can be learned online.

Other minor algorithmic improvements have also helped accelerate the standard $k$-means algorithm. Because $k$-means repeatedly seeks the minimum distance between a point and all $k$ centers, using partial distortion search (PDS, also called partial distance search) [5] and some loop unrolling permits some distance calculations to be cut short without looking at all dimensions. Mean-distance-ordered partial search (MPS) [33] draws a connection between the squared distances and the squared difference of vector sums to help eliminate candidate k-means centers. Pan et al. [30] eliminates unlikely centers using the first and second moments of a vector and portions of the vector.

DHSS (dynamic hyperplanes shrinking search) [37] eliminates unlikely candidate centers by transforming the input space (e.g. using principal components analysis) and then using the new canonical dimensions to bound the closest centers to a point. It is unclear if this method will work well in higher-dimension spaces.

Several researchers, beginning with Kaukoranta et al., identified that some clusters found by Lloyd's algorithm are 'static' over some iterations. In other words, no points join or leave the cluster from one iteration to the next. These static clusters are easily identified, as their centers do not move. Clusters which do change are called active. This information can be used to reduce the number of candidate centers for some points [20–22]. It's worth noting that the triangle inequality methods which are the focus of this chapter implicitly exploit the same information and enjoy similar benefits.

#### 2.1.3.2 Parallelization

Several machine learning packages have been constructed with the intent of improving the scalability and speed of the learning algorithms, making them applicable to large real-world problems. Packages like graphlab [25], Yael [41], and Mahout [2] provide scalable implementations of many machine learning algorithms, including $k$-means. They use different approaches: Yael focuses on low-level multithreaded optimized implementations, Mahout provides machine learning on a map-reduce infrastructure, and graphlab focuses on graph-structured algorithms and multi-core processing.

#### 2.1.3.3 Alternative Heuristic Methods

Lloyd's algorithm is a gradient descent heuristic algorithm for minimizing $k$-means distortion criterion. While it is popular and is the focus of this chapter, there are alternative methods which also aim to reduce the $k$-means distortion using different heuristics.

Agarwal et al. [1] use subsamples of the whole dataset, called core-sets, and optimize a solution based on the sample. This algorithm can be much faster in practice in low-dimensional data, but can be slow and find poor solutions in higher dimensions.

Hartigan and Wong [17] suggested an algorithm for optimizing the $k$-means distortion which considers the clustered points one by one. Each time a point changes membership, the algorithm updates the affected center locations.

Sculley [35] developed a method that is a hybrid of stochastic gradient descent (as proposed by Bottou and Bengio [6]) and batch $k$-means. It operates on small samples rather than individual examples. The resulting algorithm is less susceptible to noise caused by individual examples, yet still quite fast.

Each of these alternative algorithms can be quite fast, much faster than Lloyd's batch algorithm. The tradeoff is that they tend to produce different results. We view these as suitable alternatives to Lloyd's algorithm, but focus on accelerating Lloyd's algorithm due to its popularity. Some of these algorithms, especially those which repeatedly make nearest-center queries for the same points, may be compatible with the acceleration methods we discuss here.

## 2.2 Cluster Distortion and Lloyd's Algorithm

When clustering with $k$-means, we are trying to minimize the distortion, or sum of squared errors, between the points and their assigned centers. We can improve the distortion in two ways: by changing the points' cluster assignments and moving the cluster centers. Specifically, given a fixed set of points $X$, we are attempting to minimize the distortion function

**Table 2.1** Terms that are used frequently in this chapter

| Name and type | Description |
|---|---|
| $d \in \mathbb{N}$ | Dimension of points to cluster and cluster centers. |
| $n \in \mathbb{N}$ | Number of points to cluster. |
| $k \in \mathbb{N}$ | Number of cluster centers. |
| $X \subset \mathbb{R}^{n \times d}$ | Set of points to cluster, indexed as $x(i)$ for $1 \leq i \leq n$. |
| $C \subset \mathbb{R}^{n \times d}$ | Set of centers, indexed as $c(j)$ for $1 \leq j \leq k$. |
| $n(j) \in \mathbb{R}$ | The number of points that are currently assigned to cluster $j$. |
| $N = \{i \in \mathbb{N} \| 1 \leq i \leq n\}$ | Indexes of the points in $X$. |
| $K = \{j \in \mathbb{N} \| 1 \leq j \leq k\}$ | Indexes of the points in $C$. |
| $a : N \to K$ | Index of assigned (closest) center for each point. |
| $u : N \to \mathbb{R}$ | Upper bound on the distance between each point and its assigned center. |
| $\ell : N \times K \to \mathbb{R}$ | Lower bound on the distance between each point and each center. |
| $s : K \to \mathbb{R}$ | Half the distance between a center and its current closest other center. |

$$J(X, C) = \sum_{i \in N} \|x(i) - c(a(i))\|^2 \tag{2.1}$$

by choosing the best set of clusters $C$ (Table 2.1).

Lloyd's batch algorithm for minimizing the distortion has three basic steps, which are stated here and in slightly more detail in Algorithm 1.

1. Initialize the centers.
2. Until the algorithm converges:
   a. Assign each point to its currently closest cluster center.
   b. Move each center to the mean of its currently-assigned centers.

Step 1 occurs only once, while steps 2(a) and 2(b) alternate until the algorithm converges. Convergence is guaranteed due to the fact that steps 2(a) and 2(b) both reduce $J(X, C)$, and there is a finite number of ways to partition the $n$ points among $k$ clusters [6].

Much has been written on initializing the centers for $k$-means [8]. Researchers have used the first several examples [6], chosen $k$ points from $X$ at random [16], and used the furthest-first method [18]. The most effective current method, theoretically and in practice, is the $k$-means++ initialization [3], which randomly selects a good initialization with high probability, using something akin to furthest-first. In all our experiments, we use $k$-means++ for initialization.

The remainder of this paper is primarily concerned with optimizing step 2(a) of the algorithm (finding the closest center for each point). The naive implementation of Lloyd's algorithm spends the majority of its time here, and much of the computation done here is unnecessary; the information needed for this step can be derived using some caching and geometry.

**Algorithm 1** Lloyd's $k$-means algorithm—the standard algorithm for minimizing $J(X, C)$. Like other algorithms presented in this chapter, this algorithm's pseudocode is presented simply, without details on efficiency optimizations used in real implementations

```
procedure LLOYD($X, C$)
  while not converged do
    for all $i \in N$ do {Find the closest center to each $x(i)$.}
      $a(i) \leftarrow 1$
      for all $j \in K$ do
        if $\|x(i) - c(j)\| < \|x(i) - c(a(i))\|$ then
          $a(i) \leftarrow j$
    for all $j \in K$ do {Move the centers}
      move $c(j)$ to the mean of $\{x(i)|a(i) = j\}$
```

Step 2(b) can be optimized easily by caching sufficient statistics for each cluster: the vector sum of the points assigned to the cluster, and the number of points assigned to the cluster. Keeping these is inexpensive and avoids a sum over all points for each iteration. Each time a point changes cluster membership, the relevant sufficient statistics are updated. After the first few iterations, most points remain in the same cluster for many iterations. Thus these sufficient statistics updates become much cheaper than a sum over all points. All the algorithms we implement for this study use this technique. Thus, it is not explicitly included in algorithmic pseudocode, for clarity.

### *2.2.1 Analysis of Lloyd's Algorithm*

The running time of Lloyd's algorithm for $k$-means is $O(wnkd)$ for $w$ iterations, $k$ centers, and $n$ points in $d$ dimensions. For a fixed dataset and $k$, the number of iterations $w$ will vary depending on the initialization. In fact, $w$ may be superlinear with respect to $n$, even exponential in the worst case [38]. However, when considering less extreme cases, Lloyd's algorithm has polynomial smoothed complexity [4]. That is, $w$ is polynomial in $n$ and $1/\sigma$ (where $\sigma$ is the amount of perturbation allowed on a weakened adversary's challenge dataset).

When viewed as a gradient descent algorithm, Bottou and Bengio [6] showed that (from a given initialization) the distortion converges in Lloyd's batch algorithm at superlinear speed. This is because the variations of the second derivative of the cost function are bounded. Thus, the algorithm is minimizing the distortion in a way that is equivalent to Newton's method.

In this chapter we look at ways to accelerate Lloyd's algorithm. Given that Lloyd's algorithm is so widely used, the goal is to accelerate the exact algorithm (without any approximation), so that the resulting accelerated algorithm can be used anywhere Lloyd's algorithm is used. The accelerations we look at primarily work by avoiding many of the $nk$ interactions between $n$ clustered points and $k$ cluster centers.

### *2.2.2 MacQueen's Algorithm*

MacQueen [26] described a method similar to Lloyd's algorithm, but which updates the location of each affected cluster center whenever a point changes cluster membership. While Lloyd's algorithm moves the centers once per pass over the entire dataset, MacQueen's algorithm will move the centers (by smaller amounts) far more often. Thus, Lloyd's algorithm could be considered a 'batch' algorithm whereas MacQueen's is more 'online'. As Lloyd's algorithm is more popular in practice, and easier to accelerate since the centers move less frequently, we focus primarily on it in this chapter.

## 2.3 Tree Structured Approaches

Tree structures are effective methods for indexing spatial data in retrieval and learning algorithms. Two approaches in particular, $k$-d trees and the anchors hierarchy, have shown success in accelerating the $k$-means algorithm. We review these methods here.

### *2.3.1 Blacklisting and Filtering Centers with k-d Trees*

Pelleg and Moore [31] and Kanungo et al. [19] proposed similar methods for accelerating $k$-means by constructing and using a $k$-d tree.[1] A $k$-d tree is a binary tree that recursively partitions the space of the data it's constructed on using separating hyperplanes (typically axis-aligned). In this discussion, it is applied to the data to be clustered. After construction, the tree's structure plus sufficient statistics kept at each tree node can be used to eliminate point-center calculations. Pelleg and Moore call this approach 'blacklisting' the centers, while Kanungo et al. call it 'filtering' the centers.

Pelleg and Moore's blacklisting algorithm proceeds as shown in Algorithm 2. Initially it constructs a $k$-d tree on the data that is to be clustered. For each iteration

[1]Note that the $k$ in $k$-d trees and the $k$ in $k$-means are two different (clashing) variable names. In $k$-means, the $k$ refers to the number of centers/clusters sought; in $k$-d trees $k$ refers to the dimension of the data the structure is built on.

**Algorithm 2** Pelleg and Moore's Blacklisting $k$-means algorithm

```
procedure BLACKLISTING($X, C$)
    construct $k$-d tree $T$ on $X$
    while not converged do
        Update($T$.root, $C$)
        move centers to the mean of their assigned points
procedure UPDATE($h, C$)
    if $h$ is a leaf then
        for each data point $x$ in $h$ do
            find the closest center to $x$ and update that center's counters
    else
        compute the distance between $h$ and each center in $C$
        remove from $C$ any dominated centers
        if only one center $c'$ remains in $C$ then
            update the counters for $c'$ using the data in $h$
        else
            call Update($h$.left, $C$)
            call Update($h$.right, $C$)
```

of $k$-means, the algorithm performs a traversal of the tree, searching for regions of the tree that are 'owned' by a single center.

The traversal starts with all $k$ centers at the root of the $k$-d tree. Then it recursively descends the tree, and at each node attempts to prove that there is one center that 'dominates' (is closer to) one or more of the other centers, with respect to the hyperrectangle enclosing the $k$-d tree node. If so, it eliminates the dominated center(s). If only the dominating center remains, then the recursion stops and all points below that node in the $k$-d tree are assigned to that center. If multiple centers remain, it recursively continues its search on both child nodes. If it reaches a leaf, it performs a search of the data at the leaf to assign them to the remaining centers.

By using sufficient statistics kept at each internal node of the tree, identifying a dominating center early in the recursion (close to the root) allows the algorithm to skip not only many distance calculations but also many vector additions. For $k$-means, the sufficient statistics for a node are the number of points at that node or below, and the vector sum of those points.

While $k$-d trees can work very well in practice, the following limitations make their use less desirable, especially in high dimension:

- They are efficient only in low dimension—the running time has an exponential dependence on the dimension of the data [27]. If the number of points in the structure are significantly (exponentially) larger than the dimension of the data, $k$-d trees tend to be efficient relative to linear lookups, but for even moderate dimensions in practical applications they become too slow. Empirically they become too slow (compared to simple linear search) somewhere between 8 and 10 dimensions [28].
- They require extra memory for the tree structure and sufficient statistics. The extra memory required is on the order of the original dataset.

- When clustering, we must construct the $k$-d tree in advance, which means that we cannot start clustering until this is done.
- A $k$-d tree is not designed for efficient updates (such as adding new points or removing old points). If many updates are to be done, the tree should be reconstructed.

### *2.3.2 Anchors Hierarchy*

Moore proposed the anchors hierarchy as a tree-like geometric structure for spatial datasets [28]. The tree is built ‘middle-out’ by choosing $\sqrt{n}$ initial points known as anchors, and then merging them (to form the top of the tree) and subdividing them (to form the leaves). Each anchor maintains a list of the points for which it is the closest anchor, sorted by their distances from the anchor. Using the triangle inequality, adding additional anchors can be done efficiently. Moore showed that the anchors hierarchy is effective at eliminating many distance computations in $k$-means. However, Elkan showed it is less effective at eliminating distance computations than his method for even moderate values of $k$ than his method [12].

While tree-structured acceleration methods of the $k$-means algorithm are interesting and often useful, they are often slower and less effective in practice than the methods we turn to now. One reason pointed out by Elkan is that tree-structured methods must build a static structure before clustering begins, without knowledge of the number of clusters. Because the number of clusters is not fixed, and their positions change during clustering, static trees are less able to reduce the number of $k$-means distance calculations.

## 2.4 Triangle Inequality Approaches

The triangle inequality is a simple but very powerful tool from geometry. If $a, b, c \in \mathbb{R}^d$, then the triangle inequality states that

$$\|a - c\| \leq \|a - b\| + \|b - c\| \tag{2.2}$$

for Euclidean vector norm $\|a\| = \sqrt{a^T a}$. Intuitively, this means that the length of line segment $(a, c)$ is at most the sum of the lengths of line segments $(a, b)$ and $(b, c)$. In other words, the shortest path between two points $a$ and $c$ is a straight line; taking a path that goes through an intermediate point $b$ cannot reduce the distance.

The triangle inequality is applicable in the $k$-means algorithm in multiple ways. Generally, we desire to use it to prove that some center must be closer to a point than all other centers. Ideally, we wish to do this with as little computation as possible. For a point $x$ and two centers $c$ and $c'$, here are some of the different ways the triangle inequality can be used in $k$-means:

**Table 2.2** This table shows different acceleration methods that use distance bounds, sorting, and the triangle inequality in various $k$-means algorithms

| | Accelerations used | | | | | |
|---|---|---|---|---|---|---|
| Algorithm | 1 | 2 | 3 | 4 | 5 | 6 |
| Lloyd | | | | | | |
| Compare-means [32] | X | | | | | |
| Sort-means [32] | | X | | | X | |
| Elkan [12] | X | X | X | $k$ | | |
| Hamerly [15] | X | | X | 1 | | |
| Drake [11] | X | | X | $b < k$ | | |
| Annular (this chapter) | X | | X | 1 | | X |
| Heap (this chapter) | X | | X | 0 | | |

1. Distance from a center to its closest other center
2. Distance from a center to all other centers
3. Upper bound on point-center distances
4. Lower bound on point-center distances
5. For each center, sort all centers by distance from it
6. Sort all centers by their vector norm

The column numbers correspond to the list on the right, which lists contexts where the triangle inequality can help $k$-means avoid point-center distance calculations. Please see the algorithm descriptions for more details. Column 4 lists the number of lower bounds used per point

1. To prove that $c'$ is closer to $x$ than $c$, given only distances $\|c' - x\|$ and $\|c' - c\|$.
2. To prove that $c'$ is closer to $x$ than $c$, given only norms $\|x\|$ and $\|c\|$ and distance $\|x - c'\|$.
3. To maintain an upper bound on $\|x - c\|$ when $c$ is moving.
4. To maintain a lower bound on $\|x - c\|$ when $c$ is moving.

Table 2.2 shows the ways the triangle inequality is used in many of the algorithms described in this chapter.

### 2.4.1 *Using the Triangle Inequality for Center-Center and Center-Point Distances*

Phillips [32] demonstrated two ways to use the triangle inequality to accelerate $k$-means. Both of them use the triangle inequality to prove that centers that are far from a point's assigned center are also far from the point, and therefore can be excluded from distance computations with that point. He called his two algorithms compare-means and sort-means.

Compare-means uses the triangle inequality to prove that if center $c'$ is close to point $x$, and some other center $c$ is far away from another center $c'$, then $c'$ must be closer than $c$ to $x$. Given the already-computed distances $\|x - c'\|$ and $\|c - c'\|$, and applying the triangle inequality we can show:

$$\begin{aligned} \|c - c'\| &\le \|x - c\| + \|x - c'\| && \text{By the triangle inequality.} \\ \|c - c'\| - \|x - c'\| &\le \|x - c\| \end{aligned}$$

Thus if we also know that $2\|x - c'\| \le \|c - c'\|$ (which is trivial to calculate given the distances are already known), we can show

$$\begin{aligned} 2\|x - c'\| - \|x - c'\| &\le \|x - c\| \\ \|x - c'\| &\le \|x - c\| \end{aligned}$$

which proves that $c$ is not closer than $c'$ to $x$, without measuring the distance $\|x - c\|$. Phillips' compare-means algorithm uses this inequality in the innermost loop of $k$-means to prove that some point-center distances need not be computed. The algorithm computes and caches the center-center distances each time the centers move (once per iteration).

Sort-means computes a $k \times k$ matrix of center-center distances each time the centers move, and then sorts each row of matrix by distance. This gives each center a ranking of the other centers by distance. Whenever the algorithm wants to find the closest center for some point $x$, it searches the centers in the order of increasing distance from its currently-assigned center $c$. Then, if it can ever prove that the distance $\|c - c'\|$ to some other center $c'$ is greater than twice the distance $\|x - c\|$, it can stop searching. Thus sort-means uses the same inequality as compare-means, but it searches the centers in a different order. Because of the search order it may avoid examining some far-away centers. Of course, sort-means has the extra overhead of sorting the center-center distance matrix each time the centers move.

### *2.4.2 Maintaining Distance Bounds with the Triangle Inequality*

We can use the triangle inequality to cheaply maintain an upper bound on the distance between points, after one point has moved. Suppose $x, c, c' \in \mathbb{R}^d$, where $x$ is a point to cluster, $c$ is a cluster center and $c'$ is its new position after an iteration of $k$-means. If we know $\|x - c\|$ (from a previous iteration of $k$-means) and $\|c - c'\|$ (calculated when we move the cluster centers), we can provide an upper bound on $\|x - c'\|$ without explicitly calculating its exact value:

$$\|x - c'\| \le \|x - c\| + \|c - c'\|. \tag{2.3}$$

Intuitively, this upper bound assumes that in the worst case, $c$ moved directly away from $x$ a distance of $\|c - c'\|$, along the vector $x - c$.

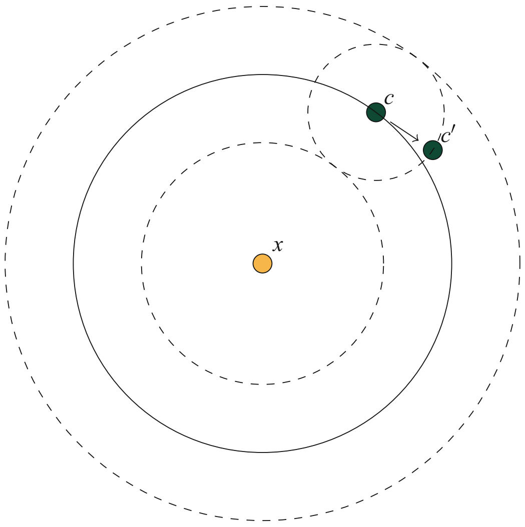

**Fig. 2.1** Using the triangle inequality to bound the distance between $x$ and $c'$, the new location of the center $c$. Assume the distance $\|x - c\|$ has been calculated. It is illustrated by the *solid circle* centered on $x$ and going through $c$. After $c$ moves to $c'$, we measure $\|c - c'\|$ (illustrated by the *dashed circle* centered on $c$). The upper and lower bounds on $\|x - c'\|$ are then given by $\|x - c\| - \|c - c'\| \leq \|x - c'\| \leq \|x - c\| + \|c - c'\|$, which are illustrated by the *two dashed circles* centered on $x$. Thus, $c'$ must be inside the region bounded by these *two dashed circles*

Another way to apply the triangle inequality is to form a lower bound on the distance between two points. Again considering $x, c$, and $c'$ as a point to cluster and the old and new positions of a center, we can form the lower bound on $\|x - c'\|$:

$$\|x - c\| \leq \|x - c'\| + \|c - c'\| \tag{2.4}$$

$$\|x - c\| - \|c - c'\| \leq \|x - c'\|. \tag{2.5}$$

Similar to the upper bound, this distance bound provides a lower bound that assumes the worst case—that $c$ moved directly toward $x$ a distance of $\|c - c'\|$, along the vector $x - c$.

Further, the upper (lower) bound can be updated correctly and efficiently in subsequent $k$-means iterations by adding (subtracting) the distance moved by a center each time it moves. This allows us to maintain both upper and lower distance bounds between a point and a moving center without explicitly calculating distances (Fig. 2.1).

### *2.4.3 Elkan's Algorithm: k Lower Bounds, $k^2$ Center-Center Distances*

Elkan [12] introduced an algorithm which uses the triangle inequality multiple ways to avoid distance calculations in the $k$-means algorithm (see Algorithm 3 for pseudocode). For each clustered point $x(i)$, the algorithm employs one upper bound and $k$ lower bounds. The upper bound is on the distance between $x(i)$ and its closest center $c(a(i))$; that is, $u(i) \geq \|x(i) - c(a(i))\|$. Each lower bound $\ell(i, j) \leq \|x(i) - c(j)\|$ bounds the distance between $x(i)$ and center $c(j)$. The upper (lower) bounds may be efficiently updated by adding (subtracting) the distance moved by each center after each $k$-means iteration.

Each time the centers move, Elkan's algorithm calculates and caches the distance between each pair of centers, as well as half the distance between each center $c(j)$ and its closest other center as $s(j)$. When it is true, the test $u(i) \leq s(a(i))$ allows Elkan's algorithm to avoid the innermost loop for $x(i)$. This is because no other center could possibly be closer to $x(i)$ than its currently-assigned center.

When Elkan's algorithm reaches the innermost loop, it may want to determine whether $c(j)$ is closer to $x(i)$ than the currently assigned center. However, if either $u(i) \leq \ell(i, j)$ or $u(i) \leq \|c(a(i)) - c(j)\|/2$, then this calculation is unnecessary, because it is not possible for $c(j)$ to be the closest center. The latter test uses the cached center-center distances.

### *2.4.4 Hamerly's Algorithm: 1 Lower Bound*

Hamerly [15] altered Elkan's algorithm by reducing the number of bounds used (see Algorithm 4). Hamerly's algorithm uses the same upper bound $u(i)$ for each point $x(i)$—for the distance between that point and its closest center $c(a(i))$. But instead of $k$ lower bounds, it uses only one lower bound per point, $\ell(i)$. This lower bound does not bound the distance from $x(i)$ to any particular cluster center. Instead, it represents the minimum distance that any center—except for the closest—can be to that point.

Consider the case where $u(i) \leq \ell(i)$. If this is true, it is not possible for any center to be closer to $x(i)$ than its assigned center. Thus, determining the assignment for $x(i)$ does not require knowing any exact distances, and the algorithm can skip the innermost loop that computes the distances between $x(i)$ and the $k$ centers.

However, if $\ell(i) < u(i)$ then it might be that the closest center for $x(i)$ has changed. In this case, Hamerly's algorithm first tightens the upper bound by computing the exact distance $u(i) \leftarrow \|x(i) - c(a(i))\|$. If this reduces $u(i)$ significantly, then possibly $u(i) \leq \ell(i)$ and the algorithm can skip the innermost loop. If not, then it must compute the distances between $x(i)$ and all $k$ cluster centers. Keeping track of the closest and second-closest allows the algorithm to

**Algorithm 3** Elkan's algorithm—using $k$ lower bounds per point and $k^2$ center-center distances

```
procedure ELKAN($X, C$)
    $a(i) \leftarrow 1, u(i) \leftarrow \infty, \forall i \in N$ {Initialize invalid bounds, all in one cluster.}
    $\ell(i, j) \leftarrow 0, \forall i \in N, j \in K$
    while not converged do
        compute $\|c(j) - c(j')\|, \forall j, j' \in K$
        compute $s(j) \leftarrow \min_{j' \neq j} \|c(j) - c(j')\|/2, \forall j \in K$
        for all $i \in N$ do
            if $u(i) \leq s(a(i))$ then continue with next $i$
            $r \leftarrow$ True
            for all $j \in K$ do
                $z \leftarrow \max(\ell(i, j), \|c(a(i)) - c(j)\|/2)$
                if $j = a(i)$ or $u(i) \leq z$ then continue with next $j$
                if $r$ then
                    $u(i) \leftarrow \|x(i) - c(a(i))\|$
                    $r \leftarrow$ False
                    if $u(i) \leq z$ then continue with next $j$
                $\ell(i, j) \leftarrow \|x(i) - c(j)\|$
                if $\ell(i, j) < u(i)$ then $a(i) \leftarrow j$
        for all $j \in K$ do {Move the centers and track their movement}
            move $c(j)$ to its new location
            let $\delta(j)$ be the distance moved by $c(j)$
        for all $i \in N$ do {Update the upper and lower distance bounds}
            $u(i) \leftarrow u(i) + \delta(a(i))$
            for all $j \in K$ do
                $\ell(i, j) \leftarrow \ell(i, j) - \delta(j)$
```

find the correct assignment, compute a tight $\ell(i)$ (for the second-closest center), and possibly tighten $u(i)$ (if the assignment happens to change).

Since $u(i)$ is the same as in Elkan's algorithm, it is updated the same way whenever the centers move. As the lower bound $\ell(i)$ is different, it is updated differently. When the centers move, the algorithm also tracks the maximum distance $\delta$ moved by any center. Then by the triangle inequality, the lower bound for each point can be updated as $\ell(i) \leftarrow \ell(i) - \delta$. This is correct since $\ell(i)$ represents the closest distance between any (non-assigned) center and $x(i)$, and no center could have moved closer toward $x(i)$ than the distance $\delta$. In fact, a small optimization is possible. For those points assigned to the furthest-moving center, their lower bounds may be reduced by the distance moved by the second-furthest-moving center.

Why does Hamerly's algorithm not track the identity of the second-closest center? Knowing its identity would allow the algorithm to more efficiently tighten $\ell(i)$ (avoiding the innermost loop over all $k$ centers). The reason is that the second-closest center's identity can change as the centers move, and without looking at all centers it can't be proved that the second-closest center remains the same over multiple iterations.

Hamerly's algorithm has several efficiency tradeoffs compared with Elkan's algorithm. With fewer lower bounds, Hamerly's algorithm uses less memory.

**Algorithm 4** Hamerly's algorithm—using 1 lower bound per point

```
    procedure HAMERLY($X, C$)
        $a(i) \leftarrow 1, u(i) \leftarrow \infty, \ell(i) \leftarrow 0, \forall i \in N$ {Initialize invalid bounds, all in one cluster.}
        while not converged do
            compute $s(j) \leftarrow \min_{j' \neq j} \|c(j) - c(j')\|/2, \forall j \in K$
5:          for all $i \in N$ do
                $z \leftarrow \max(\ell(i), s(a(i)))$
                if $u(i) \leq z$ then continue with next $i$
                $u(i) \leftarrow \|x(i) - c(a(i))\|$ {Tighten the upper bound}
                if $u(i) \leq z$ then continue with next $i$
10:             Find $c(j)$ and $c(j')$, the two closest centers to $x(i)$, as well as the distances to each.
                if $j \neq a(i)$ then
                    $a(i) \leftarrow j$
                    $u(i) \leftarrow \|x(i) - c(a(i))\|$
                $\ell(i) \leftarrow \|x(i) - c(j')\|$
15:         for all $j \in K$ do {Move the centers and track their movement}
                move $c(j)$ to its new location
                let $\delta(j)$ be the distance moved by $c(j)$
            $\delta' \leftarrow \max_{j \in K} \delta(j)$
            for all $i \in N$ do {Update the upper and lower distance bounds}
20:             $u(i) \leftarrow u(i) + \delta(a(i))$
                $\ell(i) \leftarrow \ell(i) - \delta'$
```

It spends less time checking bounds (in the innermost loop) and updating bounds (when centers move). Having the single lower bound allows it to avoid entering the innermost loop more often than Elkan's algorithm. On the other hand, Elkan's algorithm computes fewer distances than Hamerly's, since Elkan's has more bounds to prune the required distance calculations. Also, Hamerly's algorithm works better in low dimension than in high dimension. Its single lower bound reduces by the maximum distance moved by any center, and in high dimension all centers tend to move a lot due to the curse of dimensionality.

### 2.4.5 *Drake's Algorithm:* $1 < b < k$ *Lower Bounds*

Elkan's and Hamerly's algorithms keep, respectively, $k$ bounds and one lower bound per clustered point. Drake and Hamerly [11] bridged the gap between these two extreme values by using $1 < b < k$ lower bounds on the $b$ closest centers to each point. The value of $b$ can be selected in advance or adaptively learned while the algorithm runs. Drake's algorithm uses one upper bound per clustered point. Thus, Drake's algorithm uses $(b + 1)n$ total distance bounds.

For a given point, the first $b-1$ lower bounds represent the minimal distance from the point to its $b - 1$ closest centers, excluding the currently assigned center. The last ($b^{th}$) lower bound is treated specially, and we will discuss it later. Since $k$-means is only concerned with the closest center, we can avoid distance calculations to far-away centers if we can use the bounds to prove that the closest is one of the $b$ closest (the current assigned center plus the $b - 1$ next closest centers).

There are two minor complications that arise from keeping $b$ bounds per point:

- For each of the $b$ lower bound we must keep the identity of the associated center. In Hamerly's algorithm, the identity of the lower-bound center is not kept. Elkan's algorithm keeps the lower bounds in the same order as center indexes, implicitly giving the bound-center association. Keeping a center label for each bound increases the algorithm's memory footprint.
- In order to make the algorithm efficient, the lower bounds should be kept in sorted order by distance from the point. Sorting incurs overhead each time the centers move.

Nevertheless, Drake and Hamerly show that this algorithm works well in practice and can be faster than both Elkan and Hamerly's algorithms under certain conditions.

The first $b - 1$ lower bounds for a point represent the lower bounds to the associated points that are ranked 2 through $b$ in increasing distance from the point. The last lower bound (number $b$, furthest from the center) represents something a bit different. Instead of being associated with one particular center, it represents the lower bound on all the furthest $k - b$ centers. This is much like Hamerly's one lower bound, but only for the outermost centers.

When searching for the closest center to a given point, we only need to search the centers whose corresponding lower bounds are less than the upper bound for that point. Moving from the center with the smallest lower bound outward, we can hopefully stop searching after just a few comparisons. For each lower-upper bound comparison that fails, we tighten the lower (and upper) bounds as necessary. If we reach the last bound and still have not proven that any of the $b$ tracked centers are the closest, then we must search all remaining $k - b$ centers.

At the end of each $k$-means iteration, we must adjust the lower bounds. Each is reduced by the amount its associated center moved. The outermost bound should be reduced by the maximum amount moved by any of the $k - b$ outermost centers. However, in practice it is far more efficient, though not as tight, to reduce the last bound by the largest distance moved by *any* center, not just the furthest $k - b$ centers. When doing this, some bound might 'collapse' on (i.e. become smaller than) a lower bound for a supposedly closer center. When this happens, we reduce each bound for closer centers so they are at most the value that is collapsing.

The number of lower bounds $b$ used by Drake's algorithm represents a tradeoff. Increasing $b$ incurs more computational overhead to update the bound values and sort each point's closest centers by their bound, but it is also more likely that one of the bounds will prevent searching over all $k$ centers. Drake's algorithm uses a simple way to determine a 'good' value for $b$ adaptively. Starting with a large value for $b$, it may choose to reduce $b$ each iteration based on which bounds are being

**Algorithm 5** Drake's algorithm—using $b$ lower bounds per point

```
procedure DRAKE($X, C, b$)
    $a(i) \leftarrow 1, u(i) \leftarrow \infty, \forall i \in N$ {Initialize invalid bounds, all in one cluster.}
    $\ell(i, j) \leftarrow 0, \forall i \in N, j \in \{1, \ldots, b\}$
    while not converged do
5:      $m \leftarrow b$
        for all $i \in N$ do
            $j \leftarrow \arg\max_{1 \le j' \le b} u(i) \le \ell(i, j')$
            if $j < b$ then {The bounds pruned the outer centers.}
                compute distances and reorder the $j$ centers closest to $x(i)$
10:         else if $j = b$ or $\ell(i, b) < u(i)$ then {Bounds were ineffective.}
                compute distances from $x(i)$ to all centers and sort the $b$ closest
            $m \leftarrow \max(m, j)$
        $b \leftarrow \max(k/8, m)$ {Reduce $b$ if possible}
        for all $j \in K$ do {Move the centers and track their movement}
15:         move $c(j)$ to its new location
            let $\delta(j)$ be the distance moved by $c(j)$
        $\delta' \leftarrow \max_{j \in K} \delta(j)$
        for all $i \in N$ do {Update the upper and lower distance bounds}
            $u(i) \leftarrow u(i) + \delta(a(i))$
20:         $\ell(i, b) \leftarrow \ell(i, b) - \delta'$
            for $j = b - 1$ down to 1 do
                let $c(z)$ be the center that is the $j$th closest to $x(i)$
                $\ell(i, j) \leftarrow \min(\ell(i, j) - \delta(z), \ell(i, j + 1))$
```

used. If the algorithm is able to stop searching after only $b' < b$ lower bounds (over the entire dataset), then it reduces $b$ down to $b'$. Experimentally, Drake determined that for $k > 8$, $k/8$ is a good floor for $b$.

### *2.4.6 Annular Algorithm: Sorting the Centers by Norm*

Hamerly's algorithm employing one lower bound is effective at avoiding many distance computations in low dimensional spaces. But whenever the lower and upper bounds for a point cross (i.e. $\ell(i) < u(i)$), the algorithm must compute the distance between the point and all $k$ centers. This search determines the two closest centers and tightens the upper and lower bounds. Similarly, when the last lower bound in Drake's algorithm fails to prune the search, it must search over all centers (it has already searched over $b$ centers, and it must continue searching over the remaining $k - b$ centers represented by the last lower bound).

In this subsection we describe an efficient method that can prune such searches between a point $x$ and all $k$ centers. By sorting the centers by their vector norms, we can eliminate from consideration many centers whose norms are too large or too small to be closest to $x$. We can do this with only the knowledge of the norm of $x$, the norms of all $k$ centers, and (an upper bound on) the distance between $x$ and a (hopefully closeby) center.

Consider ordering the $k$ cluster centers by their vector norms, $\|\cdot\|$. This order may change at most once per iteration of $k$-means, and is inexpensive to compute and maintain. This ordering affords a novel way to structure the search for a point's closest center and potentially avoid examining all centers.

For a point $x$ having norm $\|x\|$, we can use the norm-ordering of the centers and the triangle inequality to prune the search over all centers. Assume that we know the exact distance $\|x - c'\|$ between $x$ and a reasonably close center $c'$ (such as its currently assigned center). Then consider a center $c$ that is actually closer to $x$. Starting with two different statements from the triangle inequality, we have

$$\begin{aligned} \|c\| &\le \|x - c\| + \|x\|, \\ \|x\| &\le \|x - c\| + \|c\|. \end{aligned} \tag{2.6}$$

Then we can combine these to get

$$\begin{aligned} \|c\| - \|x\| &\le \|x - c\|, \\ \|x\| - \|c\| &\le \|x - c\|; \end{aligned} \tag{2.7}$$

$$\big|\, \|x\| - \|c\| \,\big| \le \|x - c\| \qquad \text{combining the two results in (2.7).} \tag{2.8}$$

Because $c$ is closer than $c'$ to $x$, we can deduce

$$\big|\, \|x\| - \|c\| \,\big| \le \|x - c\| \le \|x - c'\|. \tag{2.9}$$

Thus, any center $c$ that is closer to $x$ than $c'$ must satisfy Inequality (2.9). From a different perspective, consider that $c$ is actually *farther* from $x$ than $c'$. Then $c$ must violate this inequality—i.e. $\big|\, \|x\| - \|c\| \,\big| > \|x - c'\|$. In this case, $c$ can be eliminated from consideration because $c'$ must be closer to $x$. Note that while the above discussion relied on knowing an *exact* distance between $x$ and some close center $c'$, the same derivation can be done with an upper bound on this distance.

We can use this knowledge to prune the search for the closest center. Given (an upper bound on) a distance $\|x - c'\|$ between $x$ and some center $c'$ (e.g. the currently assigned center), we can eliminate centers $c$ where

$$\big|\, \|x\| - \|c\| \,\big| > \|x - c'\|.$$

And since we have ordered the centers by their norm, we only need to compute the distances between $x$ and those centers $c$ whose norm falls in the range

$$\|x - c'\| - \|x\| \le \|c\| \le \|x - c'\| + \|x\|$$

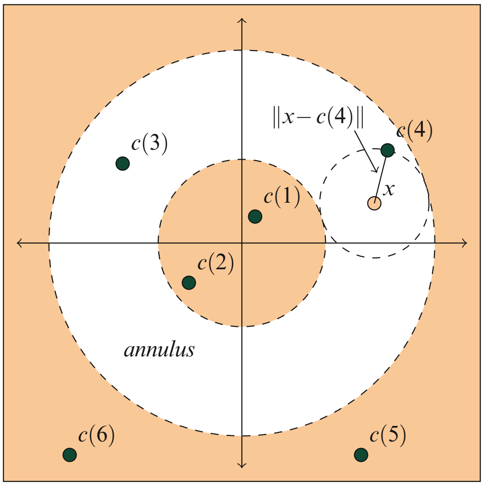

**Fig. 2.2** The annular region (*white ring* centered at origin) bounds where the closest center for $x$ might be. Centers $c(j)$ are numbered by their distance from the origin. Point $x$ has $c(4)$ as its previously-closest center, so the width of the annulus is $2\|x - c(4)\|$ (*dashed circle* centered at $x$)

These bounds form an annular region centered on the origin in which potentially closer centers to $x$ may lie. We can use binary search on the centers ordered by their norms to locate the smallest-norm center that fits this inequality, and then compute the distance between $x$ and each center within the annulus. Figure 2.2 gives a graphical representation of this search space.

We have implemented this annular search in conjunction with Hamerly's algorithm, with one additional change. Each time Hamerly's algorithm searches over all centers, it needs to discover not just the closest center, but also the second-closest center (to tighten the lower bound). Thus, when constructing the annulus for $x$, we use twice the distance between $x$ and its second-closest center as the annulus width. But since Hamerly's algorithm does not explicitly track the second-closest center, the augmented annular search algorithm attempts to do so. As mentioned above, though, the second-closest center can change. However, the annular search does not need to know which is the actual second-closest center—it only needs the index of a center which is likely to be close to $x$ to form the search annulus, and is farther from $x$ than the assigned center. Thus when it does find the actual second-closest center, it caches its identity for constructing the annulus later.

## 2.4.7 Kernelized k-Means with Distance Bounds

As with many distance-based algorithms, $k$-means can be 'kernelized' by applying the kernel trick [10, 34]. Starting from the definition of squared Euclidean distance which can be written using inner products,

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

we can kernelize the distance by replacing each inner product with a call to a kernel function $K(x, y) = \langle \phi(x), \phi(y) \rangle$ which represents the inner product of $x$ and $y$ after they have been transformed to a new space, within the range of $\phi$. Here $\phi$ typically represents a function that yields a higher-dimensional vector. Thus,

$$\begin{aligned}\|\phi(x) - \phi(y)\|^2 &= \langle \phi(x), \phi(x) \rangle - 2\langle \phi(x), \phi(y) \rangle + \langle \phi(y), \phi(y) \rangle \\ &= K(x, x) - 2K(x, y) + K(y, y)\end{aligned}$$

and if $K$ is easy to compute (without explicitly using $\phi$), then using a kernel is a relatively efficient way to perform $k$-means implicitly in the space of $\phi$.

We can apply the triangle inequality algorithms to kernelized $k$-means. However, we must be careful to avoid inefficiencies that come from kernelizing the algorithm. In particular, since the centers live implicitly in the high-dimensional range of $\phi$, and we don't represent high-dimensional feature vectors explicitly, any time we want to use a center, we instead use the kernel applied to all of the points that are in its cluster. That is, in kernel $k$-means we define center $c(j)$ by its member points:

$$c(j) = \frac{1}{n(j)} \sum_{i|a(i)=j} \phi(x(i))$$

so that when we want to know $\|x - c(j)\|^2$, we compute

$$\begin{aligned}\|\phi(x) - c(j)\|^2 &= K(x, x) - 2\langle \phi(x), c(j) \rangle + \langle c(j), c(j) \rangle \\ &= K(x, x) - 2\left\langle \phi(x), \frac{1}{n(j)} \sum_{i|a(i)=j} \phi(x(i)) \right\rangle \\ &\quad + \left\langle \frac{1}{n(j)} \sum_{i|a(i)=j} \phi(x(i)), \frac{1}{n(j)} \sum_{i|a(i)=j} \phi(x(i)) \right\rangle \\ &= K(x, x) - \frac{2}{n(j)} \sum_{i|a(i)=j} K(x, x(i)) \\ &\quad + \frac{1}{n(j)^2} \sum_{i|a(i)=j} \sum_{i'|a(i')=j} K(x(i), x(i'))\end{aligned}$$

Note that, naively, this single distance computation, which is the core of $k$-means, has a runtime of $O(n(j)^2)$ (ignoring the cost of computing the kernel). Under the reasonable assumption that clusters are roughly equal size, this is $O(n^2/k^2)$. Thus naive point-center distance computations are quite costly in kernelized $k$-means, especially when compared with the runtime of the non-kernelized version of $k$-means. However, simply caching the inner product $\langle c(j), c(j) \rangle$ for each cluster center at the beginning of each $k$-means iteration allows us to bring the cost down to a much better, though still very costly, $O(n(j))$ (or $O(n/k)$ under the assumption that all clusters are equal size). Either way, in kernelized $k$-means it is even more appealing to accelerate the algorithm by a method which avoids distance calculations altogether. We can directly apply any of the triangle inequality algorithms to kernelized $k$-means.

When applying an acceleration method such as Elkan's algorithm to kernelized $k$-means, we must additionally compute the center movement at each iteration. This is no more costly than computing the inner product for each center with itself, which is already performed each iteration as discussed previously.

## 2.5 Heap-Ordered $k$-Means: Inverting the Innermost Loops

Next we turn to a way of restructuring Hamerly's algorithm. We motivate the next algorithm in three ways:

1. it's desirable to reduce the memory use of the accelerated algorithm—in other words the number of bounds kept per point;
2. since $k < n$, the memory-efficient way of searching all point-center distances is to have a nested loop over all $n$ on the outside and all $k$ on the inside, but we may wish to invert this loop structure so that the outer loop is over $k$; and
3. thus far, the algorithms that are accelerated by the triangle inequality examine all $n$ points every iteration, but we would like an algorithm which only investigates those points whose triangle inequality bounds have been violated.

For all these reasons, we consider an algorithm that orders all the points by their likelihood of needing cluster reassignment—in other words, those for which $\ell(i) - u(i)$ is smallest (perhaps even negative).

### 2.5.1 *Reducing the Number of Bounds Kept*

This new algorithm, Heap-ordered $k$-means, replaces each pair of bounds kept for each point $(u(i), \ell(i))$ by Hamerly's algorithm with a single value representing their difference, $\ell u(i) = \ell(i) - u(i)$. The reasoning is that the bounds avoid distance calculations whenever

$$\begin{aligned} u(i) &\leq \ell(i) \\ 0 &\leq \ell(i) - u(i) \\ 0 &\leq \ell u(i). \end{aligned}$$

Thus, we can reduce by half the number of distance bounds used by Hamerly's algorithm simply by replacing the upper and lower bounds by their difference. Whenever $\ell u(i) < 0$, the distance bound for $x(i)$ has been violated and we need to tighten its bound (possibly reassigning the point to another cluster in the process).

Each update to $\ell u(i)$ is done as follows. If in Hamerly's algorithm we would increment $u(i)$ by $a$ and decrement $\ell(i)$ by $b$, then $\ell u(i)$ is decremented by $b + a$. Thus, updates to this single bound are simple.

### *2.5.2 Cost of Combining Distance Bounds*

There is a cost of replacing two bounds with one. In the innermost loop of Hamerly's algorithm, the bounds fail whenever $u(i) > \ell(i)$ and it may have to examine all $k$ point-center distances involving $x$. However, before that happens it tightens the upper bound as $u(i) = \|x(i) - c(a(i))\|$. If tightening $u(i)$ causes the bounds to become ordered again $(u(i) \leq \ell(i))$, then it has avoided $k - 1$ distance calculations. However, when the new bound $\ell u(i) < 0$, we cannot tighten it by looking at just the two centers defining the bound (the first- and second-closest centers), because we do not know the identity of the second-closest center. So when $\ell u(i) < 0$, we must examine all $k$ point-center distances for $x$ to determine whether $a(i)$ is still its closest center. We might reduce the search over all $k$ by using an annular search approach (see Sect. 2.4.6).

### *2.5.3 Inverting the Loops Over n and k*

Usually, the innermost loops in Lloyd's algorithm loop over all $n$ points, and for each point over all $k$ cluster centers to find its closest. We could invert these two loops, searching over each cluster and within that each point. But naively doing so requires maintaining an extra distance for each of the $n$ points indicating its distance to the closest of the centers examined so far.

Whichever way we structure the nesting of the these two loops, the naive strategy examines all $n$ points, and typically all $nk$ point-center pairs. It would be an advantage to examine only those points which could possibly change cluster membership, avoiding completely those points whose assignments are provably 'safe' (via distance bounds). It turns out that we can achieve this goal by using a single (combined) lower-upper distance bound, a heap for each cluster, and making the outer loop be over the $k$ clusters.

### *2.5.4 Heap-Structured Bounds*

For each cluster we construct a min-heap of the points assigned to that cluster, ordered by an estimate of their distance from the cluster center. For now assume

**Algorithm 6** Heap-ordered algorithm—inefficient version

```
    procedure HEAPKMEANS-INEFFICIENT($X, C$)
        construct $k$ min-heaps: $h(j)$ for each $j \in K$
        insert $(-1, x(i))$ into $h(1)$ for each $i \in N$ {put all in the first cluster, with violated bounds}
        while not converged do
5:          for all $j \in K$ do
                while $h(j)$ is not empty and $(\ell u(i), i)$ at the top of $h(j)$ has $\ell u(i) < 0$ do
                    remove $(\ell u(i), x(i))$ from $h(j)$
                    find $c(j')$ and $c(j'')$, the two closest centers for $x(i)$
                    compute $\ell u(i) = \|x(i) - c(j'')\| - \|x(i) - c(j')\|$ {tighten the bound}
10:                 put $(\ell u(i), i)$ into $h(j')$
            move each center to the mean of its assigned points
            update $\ell u(i)$ for each $i \in N$, restructuring each heap as necessary
```

each heap entry for $x(i)$ is a pair $(lu(i), i)$. (This is not exactly the case, shortly we will adjust this definition.) But this suffices to show that the point at the top of the heap is the one whose bound is closest to failing, or has already failed (if $\ell u(i) < 0$).

This basic approach leads to Algorithm 6 for examining only those points whose bounds have failed. While this is a reasonable algorithm, it is hampered by the fact that it must visit every point to update each $\ell u(i)$, restructuring the heap as it goes. This violates a primary goal we have for this algorithm: to avoid the $O(n)$ factor of considering every point in each iteration of $k$-means.

We can improve this algorithm by removing the per-iteration update for $\ell u(i)$. This is possible by changing the key for the heap from $\ell u(i)$ to be a related, but static value, and keeping the updates for $\ell u(i)$ external to the heap. First we will need some new notation.

Let $\ell u(i, t)$ be the value of $\ell u(i)$ at $k$-means iteration $t$. Instead of updating $\ell u(i)$ at each iteration, we keep track of the cumulative *updates* for $\ell u(i)$, which turn out to be the same for all points assigned to the same center. Suppose $x(i)$ is assigned to center $c(j)$. Let $c(j)_t$ be its location at $k$-means iteration $t$. At iteration $t$, we compute the current value

$$\ell u(i, t) \leftarrow \ell u(i, t-1) - \|c(j)_t - c(j)_{t-1}\| - m(t), \qquad \text{where} \tag{2.10}$$

$$m(t) \leftarrow \max_{j' \in K} \|c(j')_t - c(j')_{t-1}\| \tag{2.11}$$

is the distance moved by the furthest-moving center for that iteration. Then for each center $c(j)$ we maintain the following structure

$$z(j, t) = m(t) + \sum_{p=1}^{t} \|c(j)_p - c(j)_{p-1}\|, \tag{2.12}$$

where $c(j)_0$ is the center's initial position. Then $z(j, t)$ is the distance center $c(j)$ has traveled since the beginning of the algorithm plus the furthest distance any center has traveled, up through iteration $t$. This can be computed efficiently each iteration, taking $O(kd)$ time for all centers.

Assume that at iteration $t$, point $x(i)$ becomes newly assigned to center $c(j)$, and has second-closest center $c(j')$. Then we can compute the (tight) value of $\ell u(i,t) = \|x(i) - c(j')_t\| - \|x(i) - c(j)_t\|$. Into heap $h(j)$ we place the pair

$$(\ell u(i,t) + z(j,t), i). \tag{2.13}$$

In other words, we order the heap by $\ell u(i,t)$ offset by the current value of $z(j,t)$. Consider now the value of $\ell u(i,t')$ at some later iteration (i.e. $t' > t$):

$$\begin{aligned}
\ell u(i,t') &= \ell u(i,t) - \|c(j)_{t+1} - c(j)_t\| - m(t+1) - \ldots - \|c(j)_{t'} - c(j)_{t'-1}\| - m(t') \\
&= \ell u(i,t) - \sum_{p=t+1}^{t'} \|c(j)_p - c(j)_{p-1}\| + m(p) \\
&= \ell u(i,t) + \sum_{p=1}^{t} \|c(j)_p - c(j)_{p-1}\| + m(p) - \sum_{p=1}^{t'} \|c(j)_p - c(j)_{p-1}\| + m(p) \\
&= \ell u(i,t) + z(j,t) - z(j,t')
\end{aligned} \tag{2.14}$$

In Algorithm 6 at iteration $t' > t$ we would check if the top of the heap has $\ell u(i,t') < 0$. Starting with this and using Eq. (2.14) and the new heap structure, we find the equivalent test

$$\begin{aligned}
\ell u(i,t') &< 0 \\
\ell u(i,t) + z(j,t) - z(j,t') &< 0 \\
\ell u(i,t) + z(j,t) &< z(j,t'),
\end{aligned} \tag{2.15}$$

which is exactly what we used as the distance key on the heap (i.e. $\ell u(i,t) + z(j,t)$) and have been updating in the intervening iterations (i.e. $z(j,t')$).

Thus, we can keep a heap structure that does not require updating as $\ell u(i)$ changes, instead accumulating updates for each center external to the heap structure, and achieve equivalent tests for $\ell u(i) < 0$. Note that we do not need to keep the history of $z(j,t)$ for all iterations; we only ever need the value for the current iteration. This leads us to the more efficient Algorithm 7.

### 2.5.5 Analysis of Heap-Structured k-Means

To analyze Algorithm 7, we assume a simple binary-heap implementation that takes $O(\log(n))$ time to insert and remove, and introduce two new terms. The number

**Algorithm 7** Heap-ordered algorithm—using 1 bound per point, and one per cluster

```
procedure HeapKMeans($X, C$)
    construct min-heap $h(j)$ for each $j \in K$
    let $z(j) \leftarrow 0$ for each $j \in K$
    insert $(-1, i)$ into $h(1)$ for each $i \in N$ {put all in the first cluster, with violated bounds}
    while not converged do
        for all $j \in K$ do
            while $h(j)$ is not empty and $(y, i)$ at the top of $h(j)$ has $y < z(j)$ do
                remove $(y, i)$ from $h(j)$
                compute the distance from $x(i)$ to each center
                let $c(j')$ and $c(j'')$ be its closest and second-closest centers
                insert $(\|x(i) - c(j'')\| - \|x(i) - c(j')\| + z(j'), i)$ into $h(j')$
        for all $j \in K$ do
            move $c(j)$ to the average of its assigned points
            calculate $\delta(j)$ as the distance $c(j)$ moved
        compute $\delta' = \max_{j \in K} \delta(j)$
        for all $j \in K$ do
            update $z(j) \leftarrow z(j) + \delta(j) + \delta'$
```

of iterations performed by $k$-means is $w$, and the number of bound violations per iteration is $v$. In other words, $v$ is the number of points that must be removed from any heap in one iteration of $k$-means. Then the running time is

$$O(n + wv(\log(n) + kd)). \tag{2.16}$$

The most important thing to notice about this analysis is the lack of a $wn$ term, which does occur in other algorithms based on the triangle inequality. While there is a term $wv$, and $v$ depends on $n$, in general $v < n$ and highly clustered data will have $v \ll n$.

## 2.6 Parallelization

There are multiple ways to parallelize the $k$-means algorithm. While the purpose of our study is to improve the core $k$-means algorithm, we also want to show that such improvements are suitable for parallelization. In particular, we consider the simplest case of parallelizing the algorithm over a shared-memory, multicore machine.

In a shared-memory context with $p$ processors, the most straightforward way to parallelize the batch $k$-means algorithm is to partition the $n$ data points to be clustered into $p$ subsets each of size $n/p$. The cluster centers are replicated across (or shared by) all processors.

During each iteration, each processor assigns each point in its partition to the center nearest that point. After the assignment step, each processor computes for its partition the (partial) sufficient statistics required to compute the new center locations. In particular, for each center each processor must compute the vector sum of the points assigned to that center, as well as the number of points assigned to it.

Using these partial results from all processors at the end of the iteration, the new cluster centers can be computed and shared with all processors. Thus the algorithm is embarrassingly parallel within each iteration, but requires synchronization of all processors between iterations. Note that any $k$-means optimization algorithm which is not batch but stochastic in nature will not be able to directly apply this type of parallelization and maintain identical output.

A few details are worth noting. For heap-based $k$-means presented in this chapter, the algorithm can be parallelized this way by constructing $k$ heaps for each processor (resulting in $pk$ total heaps). For Drake's algorithm which may reduce the number of lower bounds used as the algorithm proceeds, each thread can adaptively reduce the number of bounds that it uses without affecting other threads, allowing the potential for optimization within smaller regions of the dataset.

We have implemented multithreaded versions of the algorithms we use in experiments, and perform experiments to see how well they scale with an increasing number of available processors. Please see Sect. 2.7 for the experimental results.

## 2.7 Experiments and Discussion

In this section we describe the experimental results the algorithms discussed in this chapter on several real-world and synthetic datasets. Table 2.3 describes the datasets. Table 2.4 describes the algorithms tested.

**Table 2.3** A list of the datasets used in the experiments

| Name | Description | Number of points $n$ | Dimension $d$ |
|---|---|---|---|
| Uniform-2/8/32 | Synthetic, uniform distribution | 1,000,000 | 2/8/32 |
| Clustered-2/8/32 | Synthetic, 50 separated spherical Gaussian clusters | 1,000,000 | 2/8/32 |
| BIRCH | 10 × 10 grid of Gaussian clusters | 100,000 | 2 |
| MNIST-50 | Random projection from mnist784 | 60,000 | 50 |
| Covertype | Soil cover measurements | 581,012 | 54 |
| KDD Cup 1998 | Response rates for fundraising campaign | 95,412 | 56 |
| MNIST-784 | Raster images of handwritten digits | 60,000 | 784 |

**Table 2.4** Algorithms tested. Given the same initialization, all algorithms produce the same result

| Algorithm | Type of acceleration | Unique features |
|---|---|---|
| Lloyd | None | Baseline algorithm for batch $k$-means. |
| Compare-means [32] | Triangle inequality | avoid innermost loop when closest other center is far away. |
| Sort-means [32] | Triangle inequality, sorting centers | Search over centers in increasing distance from closest center. |
| Elkan [12] | Triangle inequality, distance bounds | 1 upper bound, $k$ lower bounds per point. |
| Hamerly [15] | Triangle inequality, distance bounds | 1 upper bound, 1 lower bound per point. |
| Drake (adaptive version) [11] | Triangle inequality, distance bounds, sorting bounds | 1 upper bound, $b$ lower bounds per point; $b$ is chosen adaptively. |
| Annular (this chapter) | Triangle inequality, distance bounds, sorting centers | Like Hamerly, but with norm-ordered centers. |
| Heap (this chapter) | Triangle inequality, distance bounds | Uses $k$ heaps of assigned points, ordered by bounded distance from center. Upper and lower bounds are combined into one value. |
| Kernelized Lloyd | None | Lloyd's algorithm with kernels. |
| Kernelized Elkan (this chapter) | Triangle inequality, bounds | Applying Elkan's algorithm to kernelized $k$-means. |

### 2.7.1 Testing Platforms

We ran our tests on two Linux 64-bit Intel platforms. One is a 128-node parallel machine with 8 processors and 16 GB of RAM per node. We used up to 8 simultaneous threads on this machine. The other is a more recent 12-core computer with 16 GB of RAM which we used for testing up to 12 simultaneous threads. We implemented all of the algorithms tested in C++ and Pthreads. For algorithms that used similar structures, we tried to use common code wherever possible to minimize differences due to implementation.

### 2.7.2 Speedup Relative to the Naive Algorithm

Figures 2.3 and 2.4 show the speedup of the accelerated algorithms relative to the naive algorithm. Speedup is defined as the time for the naive algorithm divided by the time for the accelerated algorithm. For each dataset, we run it with multiple $k$ values. Speedups of up to 50x are observed, with the largest accelerations being for low-dimensional, naturally-clustered data. This is an important case for user-facing applications.

The results show a general improvement in speedup for most accelerated algorithms as the number of clusters increases. This makes sense because as the number of total possible distance calculations rises with $k$, so does the number of 'far away' centers that can be pruned using the acceleration techniques in this chapter. The speedup curves are not monotonic because the number of iterations varies (depending on $k$ and the initialization), and when $k$-means performs very few iterations, all the algorithms take roughly the same amount of time. One reason is because the accelerations using distance bounds provide the most benefit when the centers are moving very little.

We observe generally that not all of the accelerated algorithms always outperform the naive algorithm. For example, Elkan's algorithm shows very little if any improvement when the dimension is 2. On the other end of the dimension spectrum, sort-means and compare-means perform about the same as the naive algorithm when the data is unstructured (uniform random) and of dimension 8 or 32.

In the highest dimension dataset, MNIST-784, Elkan's algorithm is the clear winner. It benefits in this high-dimension space by being the best at avoiding distance calculations, where distance calculations are very expensive. Drake's algorithm is second-best, since it uses fewer bounds than Elkan's and is unable to avoid as many distance calculations. Generally, as dimension increases the algorithm gains more benefit from caching additional (lower) distance bounds.

### *2.7.3 Parallelism*

We implemented and tested multithreaded versions of each algorithm we investigate. Here we look at how well each is able to use additional computation resources, in terms of speedup and efficiency. While we try to parallelize all parts of each algorithm, the different steps of each algorithm require different amounts of thread synchronization in each iteration, and some parts are not easily parallelized (e.g. a single sort that occurs each iteration whose input depends on data from all threads, and whose output must be shared to all threads, as happens in the Annular algorithm).

Figure 2.5 shows the speedup of each algorithm with respect to the number of threads. Each algorithm's speedup is computed relative to using only one thread with that algorithm. We measure wall-clock time for using one thread and for using $t$ threads, and divide the former by the latter to obtain the speedup. All versions of an algorithm (e.g. single-threaded versus multithreaded), given the same initialization, produce the same sequence of $k$-means iterations and final clustering.

Figure 2.6 shows the efficiency of each algorithm with respect to the number of threads. We define efficiency as speedup divided by the number of threads. So a perfect efficiency would be a line fixed at 1.0, and a program which cannot use multiple threads would have an efficiency curve of $1/t$ where $t$ is the number of threads.

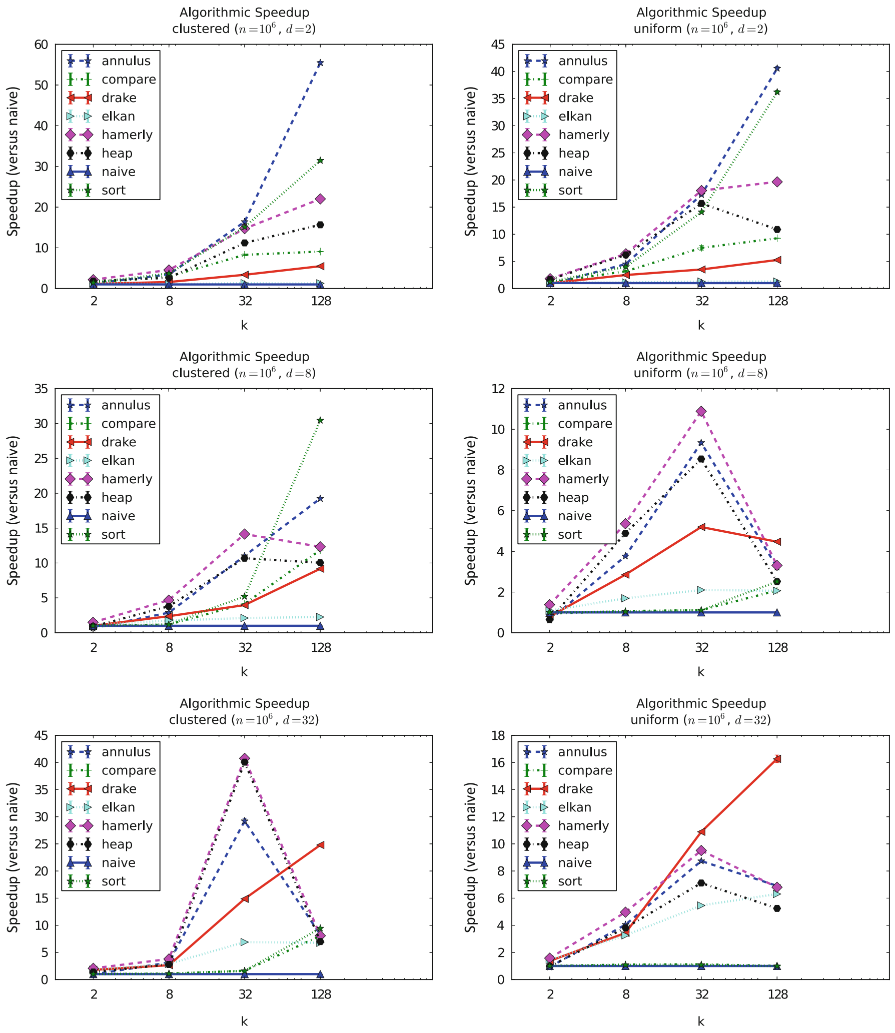

**Fig. 2.3** Speedup relative to the naive algorithm for synthetic datasets (clustered and uniform). Speedup is defined as time(naive)/time(accelerated)

It is clear that not all algorithms use the additional available computation equally well. The algorithm that benefits the most from additional threads is the non-accelerated (naive) Lloyd's algorithm, which obtains nearly linear speedups. This is likely for two reasons: it has the least synchronization between threads, and its per-thread behavior is the most predictable (each thread will do approximately equal work). Accelerated algorithms require more synchronization since there is more information kept and shared between threads. For per-thread behavior, it's possible

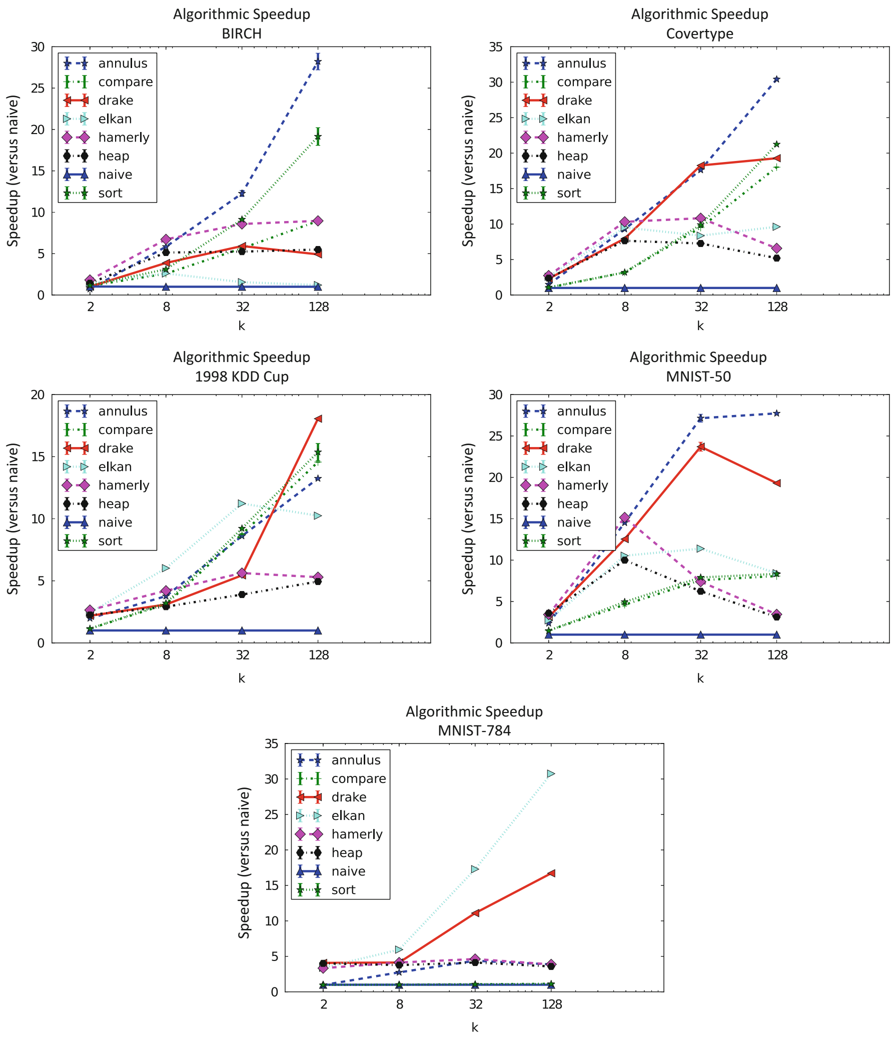

**Fig. 2.4** Speedup relative to the naive algorithm for other datasets. Speedup is defined as time(naive)/time(accelerated)

that one thread will have more work to do than another due to, e.g., distance bounds being more effective for the data assigned to the thread.

## *2.7.4 Number of Distance Calculations*

The number of distance calculations performed by $k$-means for several datasets is shown in Figs. 2.7 and 2.8 (for clustered and uniform synthetic datasets). While the datasets for the two figures are comparable in terms of the number

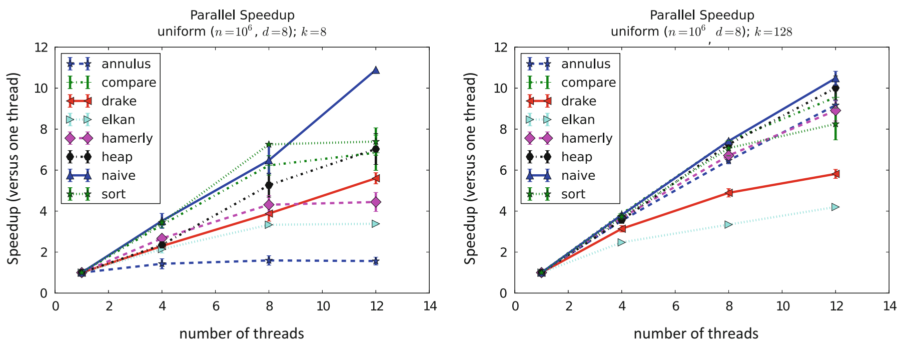

**Fig. 2.5** Speedup for each algorithm as a function of the number of threads. Speedup for $t$ threads is defined as time(single-thread)/time($t$ threads)

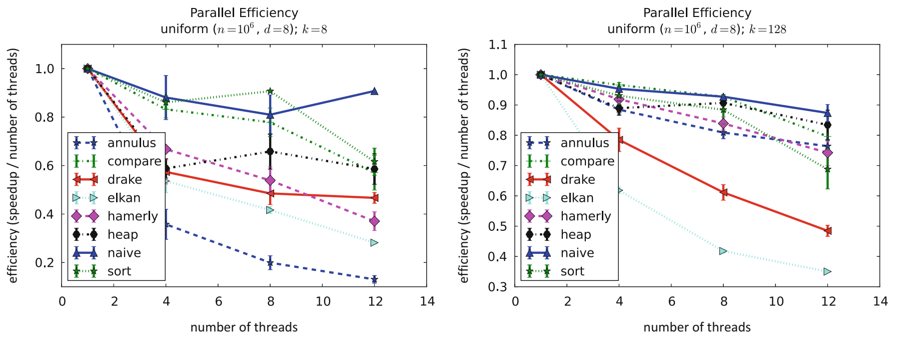

**Fig. 2.6** Parallel efficiency for each algorithm as a function of the number of threads. Efficiency for $t$ threads is defined the speedup($t$ threads)/$t$. Perfect efficiency is 1.0, and higher is better

of points, dimensions, and cluster centers used, they differ in structure (clustered versus uniform). For clustered data, all accelerated algorithms appear to compute dramatically fewer distances than the naive algorithm. However, there is a stark difference in the uniform datasets. Those accelerated algorithms that use some kind of distance bounds (Elkan, Hamerly, Annular, Drake, and Heap) all do much better than those algorithms which do not (Compare-means and Sort-means), when the dimension is 8 or higher. Thus, the distance bounds seem to be a key part of reducing distance computations in $k$-means.

As point-center distance calculations are especially expensive in kernelized $k$-means algorithms, we tested the effectiveness of Elkan's algorithm on this algorithm. Table 2.5 shows the number of distances calculated by both naive kernel $k$-means and Elkan's version on a small dataset. It is clear that Elkan's algorithm saves a dramatic number of distance calculations even in kernel spaces.

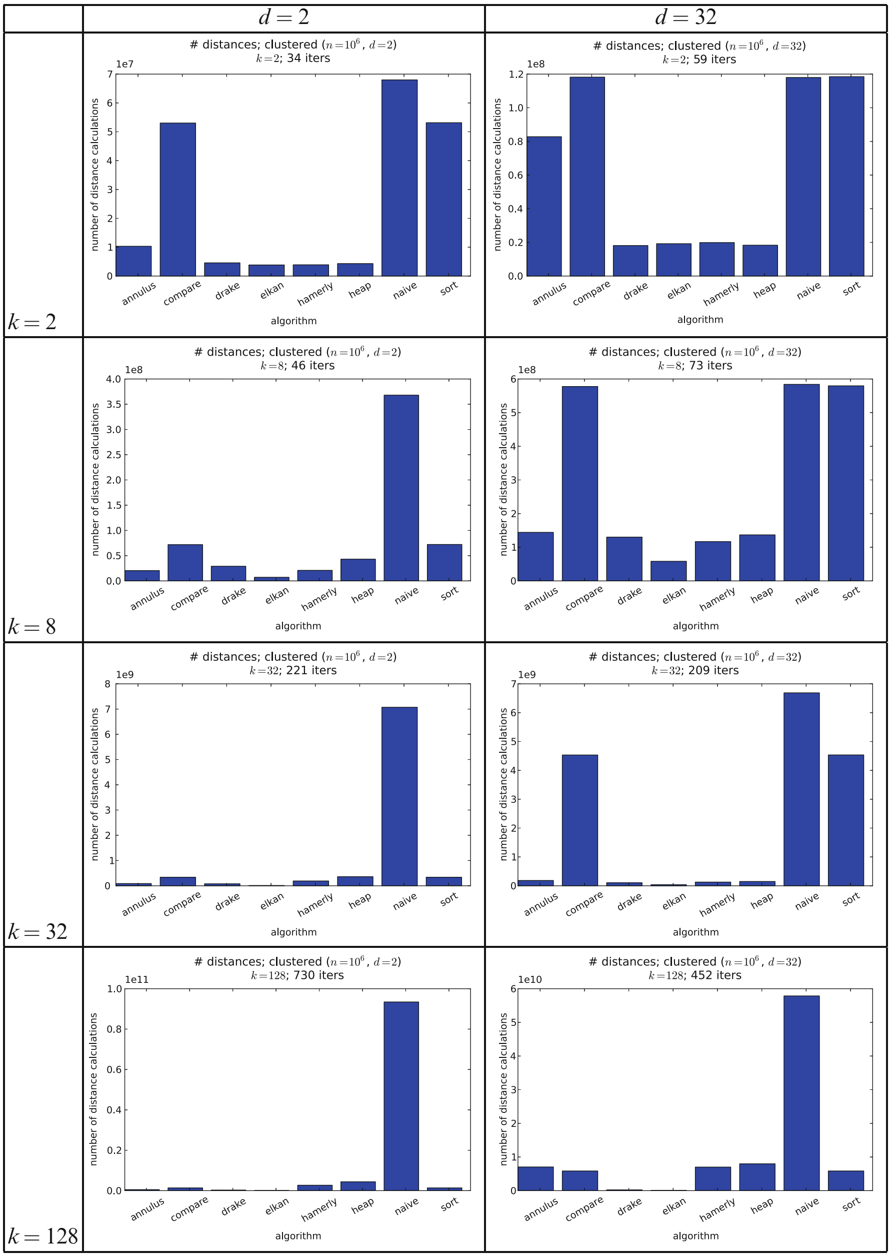

**Fig. 2.7** Number of distance calculations performed for synthetic clustered data in 2 and 32 dimensions, over $k = 2$, 8, 32, and 128

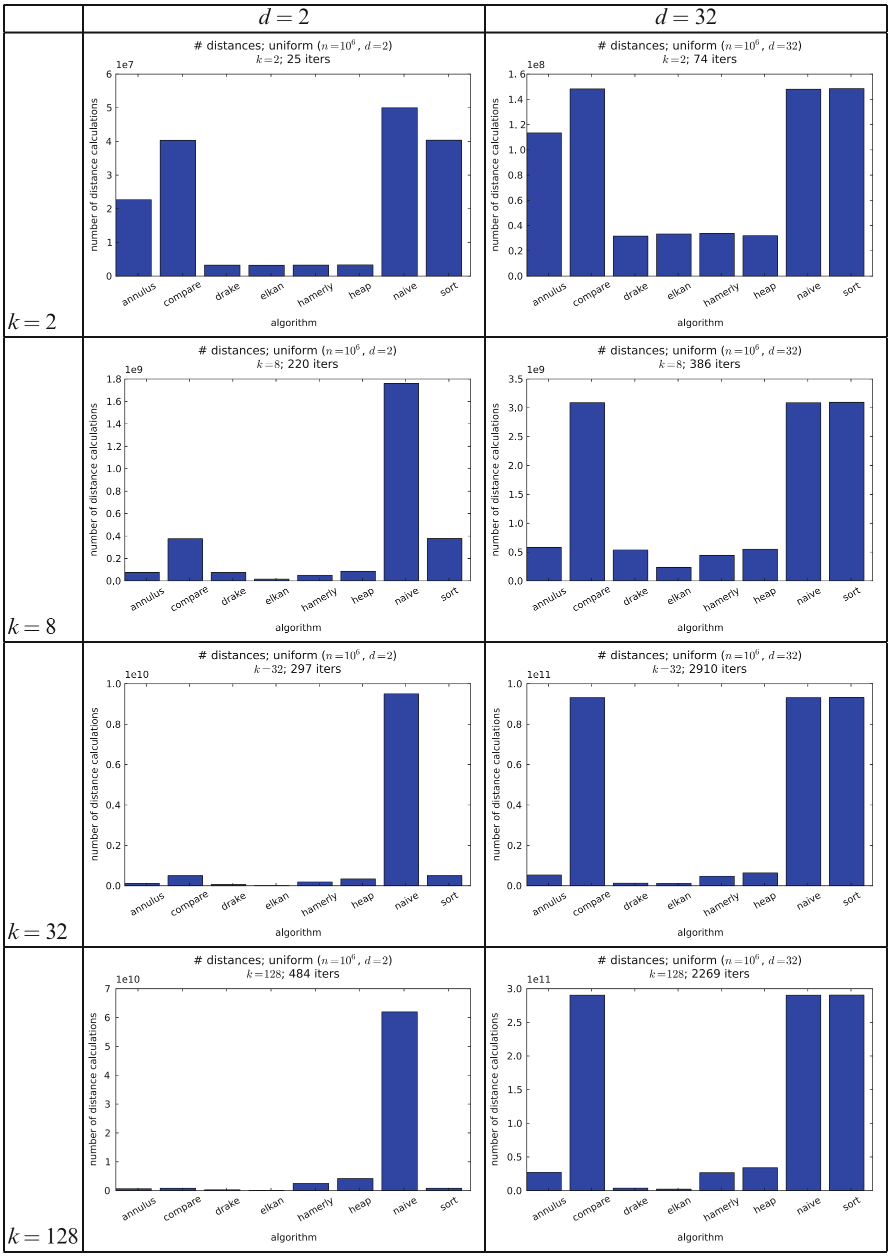

**Fig. 2.8** Number of distance calculations performed for synthetic uniform data in 2, 8, and 32 dimensions, over $k = 2$, 8, 32, and 128

**Table 2.5** The number of distances computed by unaccelerated kernel $k$-means and Elkan's kernel $k$-means

| | | Number of distance calculations | |
|---|---|---|---|
| $k$ | Iterations | Naive kernel $k$-means | Elkan kernel $k$-means |
| 2 | 24 | 96,467 | 26,766 |
| 8 | 39 | 624,750 | 155,130 |
| 32 | 43 | 2,752,810 | 708,491 |
| 128 | 14 | 3,584,506 | 628,437 |

We used a small synthetic dataset: a uniform random distribution, with 2,000 points and 8 dimensions. We use a Gaussian kernel with bandwidth $\tau = 10{,}000$. The number of iterations ranged from 14 to 43

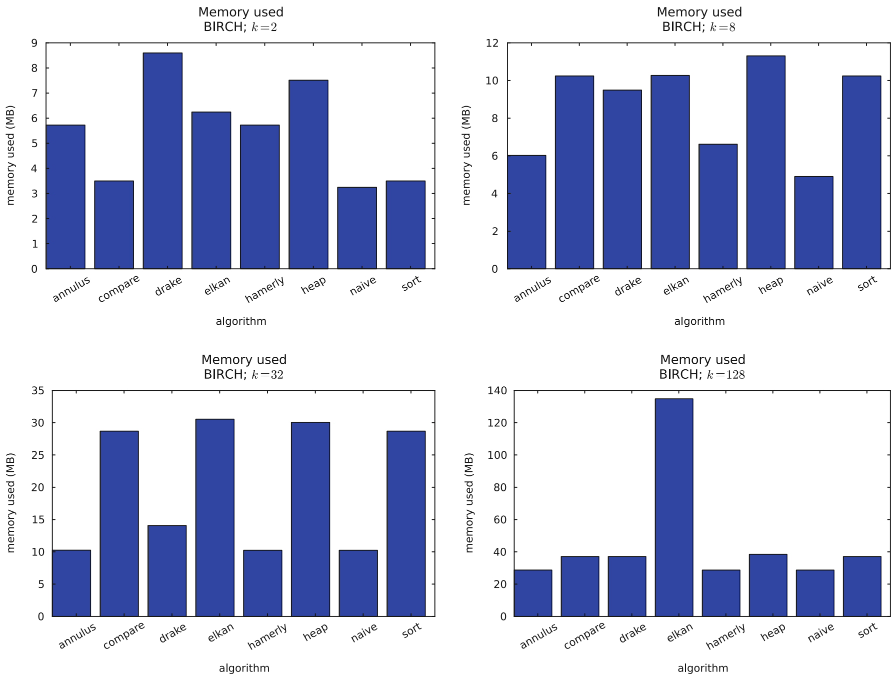

**Fig. 2.9** The amount of memory used (in megabytes) for the BIRCH, over $k = 2, 8, 32$, and 128

## 2.7.5 Memory Use

Figures 2.9 and 2.10 show the amount of memory used by $k$-means for the small BIRCH dataset ($n$ =100,000 and $d = 2$) and a large synthetic uniform dataset ($n$ =1,000,000 and $d = 32$). As opposed to the amount of time used, the amount of memory used is a function of just the number of points, dimension, and number of clusters. It's clear that when $k$ is small, the algorithms all use about the same amount

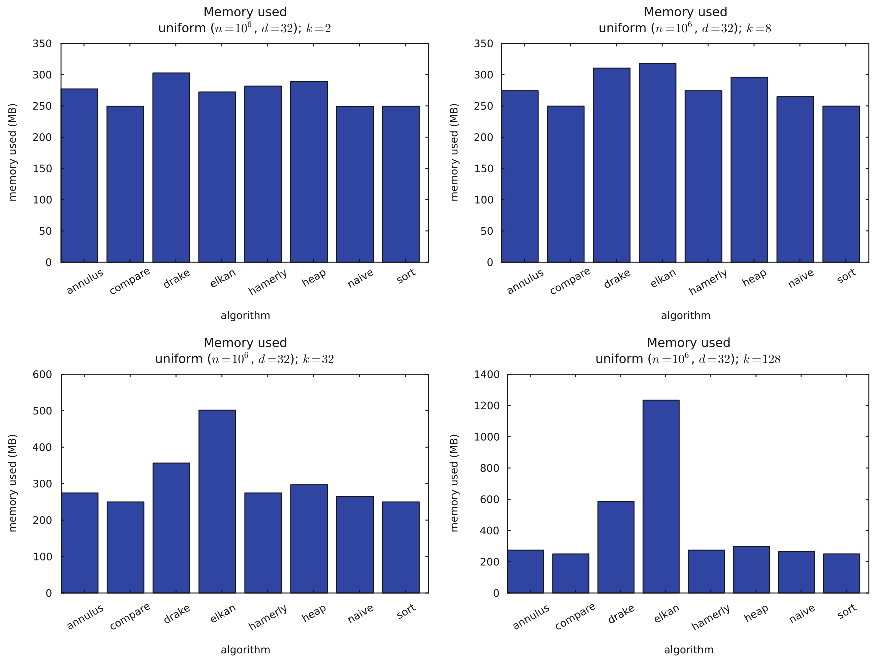

**Fig. 2.10** The amount of memory used (in megabytes) for synthetic uniform data in 32 dimensions, over $k$ = 2, 8, 32, and 128

of memory. When $k$ is large, however, the large number of bounds used by Elkan's and Drake's algorithms, begin to set them apart as using a lot more memory. It is nice that for a small relative increase in memory footprint, many of these algorithms afford significant speedups.

## 2.8 Conclusion

This chapter presents a number of alternatives to Lloyd's very popular and widely used batch $k$-means algorithm. All those presented aim to provide exactly the same answer as Lloyd's (given the same initialization), but faster. Some algorithms are from the literature of the last decade (Compare-means, Sort-means; Elkan's, Hamerly's, and Drake's algorithms), and some are new (Annular, Heap).

The algorithms studied here rely on the geometric triangle inequality to avoid unnecessary and costly distance calculations. This proves to be simple to implement and can provide dramatic speedups of up to 40x in our tests. There are multiple ways to apply the triangle inequality to speed up $k$-means. Practically, using the triangle inequality to inexpensively maintain a set of distance bounds between points and centers is the idea with the greatest benefit.

## References

1. Agarwal PK, Har-Peled S, Varadarajan KR (2005) Geometric approximation via coresets. Comb Comput Geom 52:1–30
2. Apache Mahout http://mahout.apache.org/. Version 0.8, Accessed 24 Jan 2014
3. Arthur D, Vassilvitskii S (2007) kmeans++: the advantages of careful seeding. In: ACM-SIAM symposium on discrete algorithms, pp 1027–1035
4. Arthur D, Manthey B, Röglin H (2011) Smoothed analysis of the k-means method. J ACM 58(5):19
5. Bei C-D, Gray RM (1985) An improvement of the minimum distortion encoding algorithm for vector quantization. IEEE Trans Commun 33(10):1121–1133
6. Bottou L, Bengio Y (1995) Convergence properties of the k-means algorithms. In: Advances in neural information processing systems, vol 7. MIT Press, Cambridge, 585–592
7. Celebi ME (2011) Improving the performance of k-means for color quantization. Image Vis Comput 29(4):260–271
8. Celebi ME, Kingravi HA, Vela PA (2013) A comparative study of efficient initialization methods for the k-means clustering algorithm. Expert Syst Appl 40(1):200–210
9. Coates A, Ng AY, Lee H (2011) An analysis of single-layer networks in unsupervised feature learning. In: International conference on artificial intelligence and statistics, pp 215–223
10. Dhillon I, Guan Y, Kulis B (2005) A unified view of kernel k-means, spectral clustering and graph cuts. Technical Report TR-04-25, University of Texas at Austin
11. Drake J, Hamerly G (2012) Accelerated k-means with adaptive distance bounds. In: 5th NIPS workshop on optimization for machine learning
12. Elkan C (2003) Using the triangle inequality to accelerate k-means. In: Proceedings of the twentieth international conference on machine learning (ICML), pp 147–153
13. Forgy EW (1965) Cluster analysis of multivariate data: efficiency versus interpretability of classifications. In: Biometric society meeting, Riverside
14. Fu K-S, Mui JK (1981) A survey on image segmentation. Pattern Recognit 13(1):3–16
15. Hamerly G (2010) Making *k*-means even faster. In: SIAM international conference on data mining
16. Hamerly G, Elkan C (2002) Alternatives to the k-means algorithm that find better clusterings. In: Proceedings of the eleventh international conference on Information and knowledge management, pp 600–607. ACM, New York
17. Hartigan JA, Wong MA (1979) Algorithm as 136: a k-means clustering algorithm. J R Stat Soc Ser C Appl Stat 28(1):100–108
18. Hochbaum DS, Shmoys DB (1985) A best possible heuristic for the k-center problem. Math Oper Res 10(2):180–184
19. Kanungo T, Mount DM, Netanyahu NS, Piatko CD, Silverman R, Wu AY (2002) An efficient *k*-means clustering algorithm: analysis and implementation. IEEE Trans Pattern Anal Mach Intell 24:881–892
20. Kaukoranta T, Franti P, Nevalainen O (2000) A fast exact gla based on code vector activity detection. IEEE Trans Image Process 9(8):1337–1342
21. Lai JZC, Liaw Y-C (2008) Improvement of the k-means clustering filtering algorithm. Pattern Recognit 41(12):3677–3681
22. Lai JZC, Liaw Y-C, Liu J (2008) A fast vq codebook generation algorithm using codeword displacement. Pattern Recognit 41(1):315–319
23. Linde Y, Buzo A, Gray R (1980) An algorithm for vector quantizer design. IEEE Trans Commun 28(1):84–95
24. Lloyd S (1982) Least squares quantization in PCM. IEEE Trans Inf Theory 28:129–137
25. Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2010) Graphlab: a new parallel framework for machine learning. In: Conference on uncertainty in artificial intelligence (UAI)

26. MacQueen JB (1967) Some methods for classification and analysis of multivariate observations. In: 5th Berkeley symposium on mathematical statistics and probability, vol 1. University of California Press, Berkeley, pp 281–297
27. Moore AW (1991) An introductory tutorial on kd-trees. Technical Report 209, Carnegie Mellon University
28. Moore AW (2000) The anchors hierarchy: using the triangle inequality to survive high dimensional data. In: twelfth conference on uncertainty in artificial intelligence. AAAI Press, Stanford, CA, pp 397–405
29. Ng AY, Jordan MI, Weiss Y et al (2002) On spectral clustering: analysis and an algorithm. Adv Neural Inf Process Syst 2:849–856
30. Pan J-S, Lu Z-M, Sun S-H (2003) An efficient encoding algorithm for vector quantization based on subvector technique. IEEE Trans Image Process 12(3):265–270
31. Pelleg D, Moore A (1999) Accelerating exact $k$-means algorithms with geometric reasoning. In: ACM SIGKDD fifth international conference on knowledge discovery and data mining, pp 277–281
32. Phillips SJ (2002) Acceleration of k-means and related clustering algorithms. In: Mount D, Stein C (eds) Algorithm engineering and experiments. Lecture notes in computer science, vol 2409. Springer, Berlin, Heidelberg, pp 61–62
33. Ra S-W, Kim JK (1993) A fast mean-distance-ordered partial codebook search algorithm for image vector quantization. IEEE Trans Circuits Syst II 40(9):576–579
34. Schölkopf B, Smola A, Müller K-R (1998) Nonlinear component analysis as a kernel eigenvalue problem. Neural Comput 10(5):1299–1319
35. Sculley D (2010) Web-scale k-means clustering. In: Proceedings of the 19th international conference on World Wide Web. ACM, New York, pp 1177–1178
36. Sherwood T, Perelman E, Hamerly G, Calder B (2002) Automatically characterizing large scale program behavior. SIGOPS Oper Syst Rev 36(5):45–57
37. Tai S-C, Lai CC, Lin Y-C (1996) Two fast nearest neighbor searching algorithms for image vector quantization. IEEE Trans Commun 44(12):1623–1628
38. Vattani A (2011) $k$-means requires exponentially many iterations even in the plane. Discrete Comput Geom 45(4):596–616
39. Wettschereck D, Dietterich T (1991) Improving the performance of radial basis function networks by learning center locations. In Neural Inf Process Syst 4:1133–1140
40. Wu X, Kumar V, Quinlan JR, Ghosh J, Yang Q, Motoda H, McLachlan GJ, Ng AFM, Liu B, Yu PS, Zhou Z-H, Steinbach M, Hand DJ, Steinberg D (2008) Top 10 algorithms in data mining. Knowl Inf Syst 14(1):1–37
41. Yael https://gforge.inria.fr/projects/yael/. Version v1845, Accessed 24 Jan 2014