ABSTRACT

Faster k-means Clustering

Jonathan Drake, M.S.

Mentor: Gregory J. Hamerly, Ph.D.

The popular k-means algorithm is used to discover clusters in vector data automatically. We present three accelerated algorithms that compute exactly the same clusters much faster than the standard method. First, we redesign Hamerly's algorithm to use k heaps to avoid checking distance bounds for all n points, with little empirical gain. Second, we use an adaptive number of distance bounds to avoid redundant calculations (Drake and Hamerly 2012). Experiments show the superior performance of adaptive k-means in medium dimension ($20 \leq d \leq 200$) on uniform random data. Finally, we reformulate the triangle inequality to constrain the search space for a point's nearest center to an annular region centered at the origin. For uniform random data, annulus k-means is competitive with or much faster than other algorithms in low dimension ($d < 20$), and it outperforms other algorithms on five of six naturally-clustered, real-world datasets tested ($d \leq 74$).

Faster k-means Clustering

by

Jonathan Drake, B.A.

A Thesis

Approved by the Department of Computer Science

Gregory D. Speegle, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of

Master of Science

Approved by the Thesis Committee

Gregory J. Hamerly, Ph.D., Chairperson

Gregory D. Speegle, Ph.D.

Ronald B. Morgan, Ph.D.

Accepted by the Graduate School
August 2013

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I want to express my great appreciation to Greg Hamerly for his valuable and constructive suggestions throughout this entire research process. His reliability and willingness to give his time so generously have been as important to this work as his clear insight and standard of excellence.

I would also like to thank my thesis committee members Greg Speegle and Ron Morgan for their support and critical attention. I am grateful to Paul Grabow and Cindy Fry, whose flexibility and enthusiasm made balancing work and research easy. The dependable assistance given by Sharon Humphrey over the years is also much appreciated. Finally, I thank Diane Drake, Thomas Carlson, Ryan Henning, and Amanda Nguyen for asking me what k means and for listening to the answer.

CHAPTER ONE

Introduction

1.1 Overview

As early as the mid-1960s, Forgy, MacQueen, and Lloyd contributed ideas and algorithms which led to the formulation of k-means as a method of clustering vector data (Forgy 1965; MacQueen 1967; Lloyd 1982). Broadly, k-means is an unsupervised machine learning method useful for dividing a dataset into subgroups. Although there are many ways to define a cluster, we generally mean for points within a cluster to be maximally similar and for points in separate clusters to be maximally different. Clustering, and hence k-means, is popular where there are large amounts of multi-dimensional data requiring analysis, e.g. in bioinformatics, cheminformatics, astrophysics, vector quantization, and computer vision.

Today k-means is a common tool for machine learning and data mining. Despite its long history, k-means continues to enjoy widespread relevance (Jain 2010); it has been named among the top 10 algorithms in data mining (Wu et al. 2008). Currently, there are many avenues of ongoing research into accelerating, enhancing, and otherwise improving k-means. Since clustering is often at the heart of larger data mining efforts, new developments in k-means have a wide impact. In Chapter 2, we discuss published discoveries and contributions from researchers worldwide, ranging from theory to practice.

The algorithms introduced in Chapter 3 accelerate Lloyd's standard k-means algorithm by avoiding redundant distance calculations. Embedding additional geometric reasoning into the algorithm allows us to eliminate significant computational overhead without changing the answer, yielding 10- to 40-fold speedups in runtime. The results of our experiments are detailed and analyzed in Chapter 4.

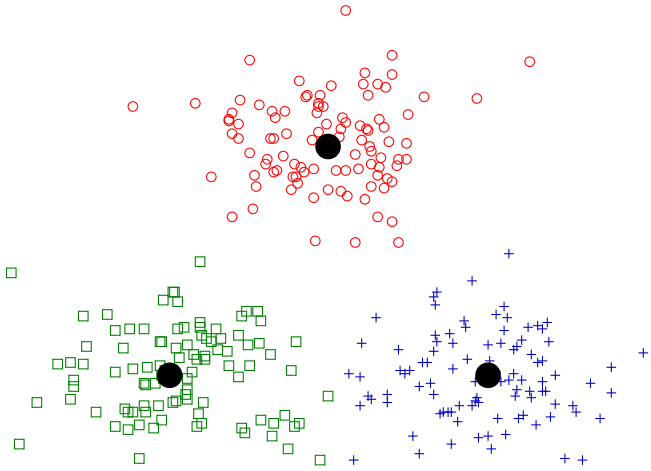

Figure 1.1: A simple clustering of points in the plane with $k = 3$ clusters sampled from the normal distribution; centers are shown as large black dots.

1.1.1 Clustering Vector Data, and the Name "k-means"

Cluster analysis is a fundamental machine learning and data mining technique. When clustering vector data, we partition a dataset into disjoint subsets called clusters, and by convention we say there are k such clusters. We also identify each cluster with a representative center point, where the natural center is the mean of all data points belonging to the cluster. There are k of these means; hence the name k-means clustering. In various contexts, centers may equivalently be called centroids, barycenters, or centers of mass (Kanungo et al. 2002; Arthur and Vassilvitskii 2006). Figure 1.1 shows a simple example of Gaussian data and centers, with $k = 3$ clusters.

1.1.2 Applications of k-means

Clusters form an encoded description or compression of the dataset used to generate them. Moreover, after clustering a dataset of sample observations, we can quickly predict the cluster assignment of a previously-unobserved sample by choosing its closest center. Clustering and hence k-means is relevant within many scientific, engineering, and other technical disciplines.

In the biological sciences, clustering is useful for grouping related organisms into phylogenies, according to whatever features of the organisms are under investigation. As mentioned previously, we can cluster related genes sequences for genotyping or for research into genetic evolution (Celebi 2009). In chemistry, the behaviors and properties of complex molecules can be encoded numerically, clustered, and used to understand which aspects of molecular structure predict molecular properties. k-means can be used to aid in computer vision by clustering the pixels of an image into segments, facilitating border detection and object recognition (Celebi 2011). k-means is also useful in social networking. For example, k-means could be used to find communities within a network of people, or group related videos based on usage logs. In data communications, it is a common method of vector quantization, e.g. for lossy data compression of streaming voice or video signals.

Clustering groups related items. For example, recommender systems might use k-means to help users find items they like. It can also help automatically generate abstracts of certain types of streaming video, which is becoming increasing prevalent on the web (Furini et al. 2008).

1.2 Lloyd's Algorithm

The standard implementation of k-means clustering is a simple iterative process named Lloyd's algorithm; in common usage, it's synonymous with k-means.

1.2.1 Basic Procedure

The strength of Lloyd's algorithm is its simplicity. Given a dataset $\{x_i\}_1^n \subset \mathbb{R}^d$ and a set of initial center locations $\{c_j\}_1^k \subset \mathbb{R}^d$, we merely alternate between the following two steps repeatedly, until center locations converge (Moore 2001):

– Assign each point x_i to its nearest center

– Relocate each center c_j to the centroid of all points assigned to it

1.2.2 *Formal Restatement as Optimization Problem*

Formally, the k-means task is an optimization problem that partitions data into clusters while minimizing the sum of the squared distances between each data point and the center of the cluster it belongs to. Given a set of points $\{x_i\}_1^n \subset \mathbb{R}^d$ and a set of centers $\{c_j\}_1^k \subset \mathbb{R}^d$, we define the minimization objective function $J(x, c)$ as the sum of squared errors given in Equation 1.1.

$$J(x, c) = \sum_{i=1}^{n} \operatorname*{argmin}_j \|x_i - c_j\|^2 \tag{1.1}$$

1.2.3 *Convergence and Termination*

The consequence of alternately assigning points to their nearest center and moving centers to their cluster centroids is that the average distance between each point and its assigned center decreases after each iteration, so the algorithm always makes progress in reducing the sum of squared distances between each point and its assigned center, i.e. the objective function J.

There are k^n ways to partition a set of n points into k possibly-empty clusters. If we insist on strictly non-empty clusters, then the number of ways to partition the set is given by the Stirling numbers of the second kind $\left\{{n \atop k}\right\}$, which we define below in Equation 1.2. In either case the number of configurations is bounded by a finite number, and we always make progress on the objective surface. For the same reason that monotone, bounded sequences converge, the k-means algorithm eventually terminates.

$$\left\{{n \atop k}\right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n \tag{1.2}$$

1.2.4 Limitations

This standard but naive approach has several limitations. First, the clusters Lloyd's algorithm produces may be arbitrarily far from the optimal clustering. Second, the algorithm requires the user to specify k, the number of clusters, and to provide an initial guess of cluster locations. Finally, and having most relevance to this thesis, the standard algorithm takes a long time to run (Kanungo et al. 1999).

While the centers must necessarily converge in finite time, Lloyd's algorithm need not produce the globally best clustering. We want to minimize the sum of squared distances between each point and its assigned center, but k-means merely descends a multidimensional surface to its local minimum, which may or may not be the global minimum. In particular, both the number of iterations required and the actual output clustering are highly sensitive to the choice of initial center locations used to seed the first iteration. Methods of initialization are discussed in Section 2.1. We also consider non-Lloyd k-means heuristics in Section 2.4.

1.3 Distance Metrics

A key assumption underlying the clustering task is the existence of an appropriate distance metric for points in the dataset. Although we have formulated the k-means problem in $\mathbb{R}^d$ using straight-line Euclidean distance for convenience and ease of understanding, our acceleration techniques also apply to more general spaces with suitable distance metrics.

For many purposes, the Euclidean 2-norm is a perfectly satisfactory measure of distance, but depending on context, other metrics may be more useful. For example, the Mahalanobis distance has the advantage of being able to correct for scaling and distribution of data, while the generalized Minkowski norm allows for various

non-spherical cluster shapes. Kernelized variants of k-means are also possible. Moreover, kernel k-means has been shown to be equivalent to kernel principal component analysis (kernel-PCA) (Dhillon et al. 2004).

As long as we have a distance metric d supporting the triangle inequality, i.e. satisfying $\forall x, y, z$

$$d(x, y) \leq d(x, z) + d(z, y), \tag{1.3}$$

we can accelerate k-means considerably. Our work continues a long tradition of triangle-inequality-based geometric reasoning (Drake and Hamerly 2012; Hamerly 2010; Elkan 2003; Huang et al. 1992; Orchard 1991).

1.4 Complexity

1.4.1 Worst-case Analysis

As a decision task, k-means belongs to the class of NP-hard problems, i.e. there is no known polynomial-time way to decide if, given some target distortion D, there exists a set of k centers having distortion less than D. However, for a dataset of n points, we can trivially bound the number of iterations by the number of possible configurations, k^n. Inaba et al. counted the number of possible Voronoi cells to improve this upper bound to $O(n^{kd})$ (Inaba et al. 1994). It is not known whether this bound is tight, i.e. $\Theta(n^{kd})$, as finding lower bounds has proven difficult. Har-Peled and Sadri (2005) showed $\Omega(n)$ worst-case performance, and Arthur and Vassilvitskii (2006) later showed a super-polynomial lower-bound for worst-case complexity of $2^{\Omega(\sqrt{n})}$. Subsequently, Vattani further improved the lower bound by adversarially constructing an instance requiring $2^{\Omega(n)}$ iterations (Vattani 2009).

1.4.2 Smoothed Analysis

Despite its exponential worst-case behavior, k-means has polynomial smoothed complexity, as proved by Arthur et al. (2009). They also mention research in improving the degree of this polynomial, which is currently about 30. Intuitively, this smoothed analysis means that the algorithm's realistic performance feels polynomial and that worst-case behavior is the exception rather than the norm. This theoretical analysis helps to explain the generally good performance of k-means in practice.

1.5 Scalability and Parallelism

Lloyd's algorithm is embarrassingly parallelizable: during a given iteration, we can easily partition a dataset of size n among different processors, replicating the k clusters. Typically $k \ll n$, so k-means can easily handle big data. There exist modern implementations of Lloyd's algorithm based on common parallel programming frameworks such as MPI, OpenMP, and MapReduce (Zhao et al. 2009; Zhang et al. 2011).

Many of the accelerations discussed in this thesis are also parallelizable, so the speedups achieved through geometric reasoning multiply across parallel processing units, allowing scalability to even larger datasets. Alternatively, from the opposite perspective, our improvements allow an already-parallel clustering system to perform the same task in a fraction of the time, or perform the same task in the same time using a fraction of the computational resources.

1.6 Exact Acceleration

We have designed the k-means accelerations presented in this thesis to produce exactly the same centers as Lloyd's algorithm for any given initialization. We intend these fast algorithms to be drop-in replacements wherever standard k-means is used. Even while restricting our discussion to exact methods, there are many ways to make

k-means faster. Some techniques are now obsolete and others remain competitive in certain circumstances. We discuss several noteworthy approaches in greater detail in Chapter 2, including methods based on spatial data structures and others based on geometric reasoning.

1.6.1 Justification of Exactness

Having differentiated between the k-means task and Lloyd's algorithm, we acknowledge that the standard algorithm is already heuristic. Nevertheless, exact acceleration of this heuristic is valuable for several reasons. First, k-means is widely used and widely studied, so acceleration offers immediate benefits. Second, any given non-exact variant of k-means may also be amenable to our triangle-inequality accelerations; even distant cousins like the k-nearest neighbors problem share this property. Finally, maintaining exactness allows researchers and end-users to make clear comparisons between different algorithms.

1.6.2 Spatial Data Structures

In the category of spatial structures, k-d trees tend to work best in very low dimension, e.g. $d \leq 5$ (Pelleg and Moore 1999; Kanungo et al. 1999), Moore's anchors hierarchy tolerates high dimension but achieves only modest speedups (Moore 2000), and the filtering algorithm of Kanungo et al. (2002) is practical only for $10 \leq d \leq 20$.

1.6.3 Geometric Reasoning

These structural approaches are largely made obsolete by a class of algorithms leveraging the triangle inequality. Phillips (2002) adapts Orchard's nearest-neighbors method to k-means and makes major improvements by pre-sorting a table of inter-center distances, yielding competitive speedups over Lloyd's algorithm in medium and low dimension. Elkan's algorithm introduces lower bounds on the distance between

points and centers, a technique which avoids explicitly computing distances across all pairs in every iteration (Elkan 2003). In high dimension, roughly $d > 200$, Elkan's algorithm remains the fastest competitor at the time of this writing.

Simplifications and modifications of Elkan's technique have yielded new algorithms with different characteristics, including Hamerly's low-dimensional algorithm (Hamerly 2010), the medium-dimensional adaptive distance bounds algorithm presented in Chapter 3 of this thesis (Drake and Hamerly 2012), and the annular search algorithm, which is also introduced in this thesis.

1.6.4 Towards a Universal k-means Algorithm

These accelerated methods vary in memory overhead, as well as in their sensitivity to the number of centers k, the dataset's dimension d, and the dataset's level of natural clusterability. Ideally, we want a k-means algorithm performing well for all values of n, k, and d and all datasets. Meanwhile, we can construct a piecewise algorithm that switches to the most appropriate acceleration technique based on the particular conditions of the problem.

CHAPTER TWO

Literature Review

2.1 Initialization Methods

The clustering found by k-means depends on the choice of initial center locations. Since k-means is heuristic, a bad initialization can lead to poor quality results. So what is a good way to choose initial centers? There have been several proposed techniques. We first review basic initialization strategies and then discuss more sophisticated options.

2.1.1 Basic Approaches

The simplest and most obvious way to initialize centers is just to pick k random points from the dataset. Random initialization (Forgy 1965) has several advantages, including fast runtime and being easy to understand and trivial to implement. Random initialization often performs adequately, but arbitrary center selection can lead to arbitrarily high distortion, e.g. if two centers are too close and become trapped together inside one true cluster of points in the dataset. Figure 2.1 gives a simple example of poor initialization.

Since we generally want points from the same cluster to be maximally similar and points from different clusters to be maximally different, it makes sense that centers should not be near each other. This insight motivates an improved technique called FARTHEST-FIRST initialization (Hochbaum and Shmoys 1985). To start off, the first center is initialized randomly, but remaining centers are selected to be those points that maximize the minimum distance to previously-chosen centers. This straightforward method does a better job of distributing centers across the dataset than random initialization, but it has a serious pitfall: farthest-first initialization

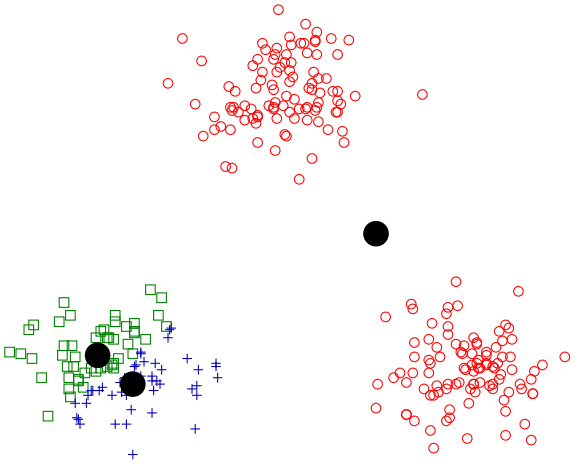

Figure 2.1: Given these initial center locations (black dots), k-means cannot iterate: points are already assigned to their nearest centers, and all $k = 3$ centers are at the mean of their assigned points. Here, the points marked with red $\circ$'s are closest to the center in the space between them, whereas the points marked with green $\square$'s and blue $+$'s belong to separate clusters, according to their nearest center.

preferentially selects distant outliers as centers, as noted by Hamerly (2010). This choice of initial center locations tends to increase the number required iterations and reduce the quality of the final clusters found by k-means.

Turnbull and Elkan (2005) describe a variation of the farthest-first technique using a random *subset* of the data points of size $2k \ln(k)$ rather than the entire dataset. The idea here is to choose a subset containing a better ratio of non-outliers to outliers, such that running farthest-first initialization selects fewer outliers as centers. We call this method SUBSET-FARTHEST-FIRST.

2.1.2 Advanced Methods

The popular k-means++ algorithm (Arthur and Vassilvitskii 2007) is a randomized technique for picking statistically good initial centers from the dataset, also based on the insightful observation that clusters should be far away from each other. Impressively, this recent technique offers theoretical guarantees about the output clustering quality. While k-means itself provides no bound on the distance between

the local minimum it finds and the true optimal clustering, k-means++ initialization leads to solutions $\Theta(\log k)$-competitive with the optimal clustering.

The k-means++ initialization method chooses the first center c_1 at random from the dataset x, but each successive center c_j is selected with probability

$$\frac{D^2(c_j)}{\sum_{x_i \in x} D^2(x_i)}, \tag{2.1}$$

where $D(c_j)$ denotes the minimum distance between c_j and all previously-chosen centers (Arthur and Vassilvitskii 2007). One noteworthy shortcoming of k-means++ initialization is its lack of straightforward parallelizability, however Bahmani et al. (2012) recently developed a highly-scalable variant called k-MEANS|| with similar guarantees about the quality of the resulting clusters.

In an article analyzing the efficient application of k-means to color quantization, Celebi compares random selection, FARTHEST-FIRST, SUBSET-FARTHEST-FIRST, and k-means++ (Celebi 2009). The methods performing most consistently and leading to the highest-quality solutions, i.e. with least distortion, were k-means++ and SUBSET-FARTHEST-FIRST. Another paper comparing additional k-means initialization schemes makes usage recommendations based on time and memory constraints and other problem conditions (Celebi et al. 2013). Notably, there has been promising research into deterministic (non-random) initialization schemes competitive with k-means++, e.g. VAR-PART and PCA-PART (Su and Dy 2007).

For both internal and external consistency and ease of comparison, our accelerated algorithms employ k-means++ to initialize centers owing to its widespread adoption and good performance. Additionally, we perform a number of experiments with random initialization to show our algorithms' speedup is not contingent on a particular initialization method. Although we use k-means++ because it represents best practice, we note that better center initialization means there is relatively less room for subsequent acceleration, which if anything tends to dampen empirical results.

2.2 Spatial Data Structures

Another branch of k-means research seeks to accelerate the process of computing nearest-neighbors by implementing Lloyd's algorithm with efficient structures.

2.2.1 k-d trees

In 1999, Moore used multi-resolution k-d trees to build a fast expectation-maximization algorithm for mixture model clustering (Moore 1999). That same year, Pelleg and Moore (1999) and Kanungo et al. (1999) independently proposed new implementations of k-means that achieved speedups by structuring data points in these multi-dimensional trees.

Kanungo et al. gave no empirical analysis of their proposed algorithm, but the authors did discuss an insightful theoretical motivation for their work. "After an initial phase of rapid movement of the center points, the [standard] algorithm tends to settle into a long phase where the center points move only very slowly. This [observation] suggests that a smart algorithm should attempt to update nearest neighbors incrementally after the centers move, rather than recompute them from scratch each time" (Kanungo et al. 1999). The new algorithm reduces the number of explicit nearest-neighbor computations by storing the dataset in *balanced box-decomposition trees*.

Pelleg and Moore independently developed essentially the same method of acceleration using the familiar name k-d trees. Their experiments showed that k-means is not intractably slow, at least in low dimension: $d \leq 5$. Beyond that, k-d trees rapidly become unwieldy and expensive to maintain. The original paper acknowledges that for $d > 8$, the costs of updating the tree exceed the benefits of the structure, and the algorithm performs "badly" (Pelleg and Moore 1999). Regardless of dimension, faster alternatives now exist, including Hamerly's simplification of Elkan's algorithm (Hamerly 2010) and the annular search variant proposed in this thesis.

2.2.2 *Moore's Anchors Hierarchy*

Moore later published another acceleration technique: the anchors hierarchy (Moore 2000), which tolerates high dimension well, complementing the low-dimensional speedups from k-d trees. In the anchors hierarchy, points are stored in metric trees, improving the efficiency of determining a point's nearest center. Moore also used a novel *middle-out* technique for building these trees, which is faster than top-down or bottom-up construction and leads to faster runtime.

2.2.3 *The Filtering Algorithm*

Later still, Kanungo et al. published a new technique using k-d trees called the filtering algorithm (Kanungo et al. 2002). After indexing the dataset in a tree, the algorithm maintains a set of candidate nearest centers for each node in the tree, and propagates these candidate centers from the root down. The authors gave a data-sensitive analysis describing the theoretical merits of their algorithm, and then gave an empirical analysis which showed the algorithm's good performance for $10 \leq d \leq 20$.

2.3 *Geometric Reasoning: the Triangle Inequality*

Many accelerated algorithms use the triangle inequality, which we defined in Chapter 1 as Inequality 1.3, to decide that a distance calculation is unnecessary.

2.3.1 *Elkan's Algorithm*

Elkan demonstrates how to use the triangle inequality in metric spaces to significantly accelerate k-means (Elkan 2003). His accelerated algorithm maintains n upper bounds on the distance between each point and its assigned center, nk lower bounds on the distances between points and each center, and $O(k^2)$ inter-center distances. For a given data point, as long as the upper bound on the distance to its assigned center is within the lower bound on its distance to some other center, then

we do not need to compute the exact distance to that other center, because it cannot possibly be closer than the currently assigned center.

Kanungo et al. suggested that a good k-means algorithm would exploit the fact that, as centers converge to their ultimate configuration, most points remain assigned to the same center (Kanungo et al. 2000). Elkan observes that his algorithm achieves this goal (Elkan 2003). Keeping distance bounds allows for many unnecessary calculations to be eliminated when points do not frequently change assignment. Intuitively, since centers do not move very much, we can get away with only keeping loose bounds on distances instead of making many exact distance calculations.

Coupled with low overhead compared to indexing algorithms that maintain large tree structures, Elkan's algorithm is exact, correct, and quite fast. Hamerly provides a clear and concise summary of this algorithm in (Hamerly 2010). For full details, see the original paper (Elkan 2003).

2.3.2 Hamerly's Algorithm

Subsequently, Hamerly gives an "even faster" k-means algorithm that simplifies Elkan's algorithm by replacing the lower bounds on point-center distances with a single lower bound between points and the second-closest centers (Hamerly 2010). Whereas Elkan kept nk lower bounds, Hamerly keeps only n lower bounds on the distance between each point and its second-closest center. Now we no longer need to test bounds on all k centers: ideally, the second-closest center is sufficiently far away, and hence all the other centers are as well.

Assuming that converging centers do not move very much, these loose bounds are sufficient to get an exact k-means solution without making many expensive distance calculations. Provided that this assumption holds, Hamerly's algorithm spends less time updating bounds and is much more efficient than Elkan's algorithm and other accelerated algorithms, including indexing methods. However, due to the curse

of dimensionality, centers move greater distances in higher dimension, so we expect speed gains to deteriorate as dimension increases. In fact, Hamerly's algorithm outperforms competitors up to medium dimension ($d \leq 50$), whereas the additional lower bounds in Elkan's algorithm tolerate higher-dimensional spaces (Hamerly 2010).

Notably, Hamerly's algorithm and Elkan's are complementary: one gives superior performance in low-dimension, and the other tolerates high-dimension well. A lesion study of four modifications to the algorithm demonstrated performance tradeoffs depending on dimension and the number of clusters (Hamerly 2010). Attempts to apply the triangle equality to the still-expensive computation of $O(k^2)$ inter-center distances at each iteration were unsuccessful.

Hamerly concludes by orienting his accelerated algorithm as a precursor to an ideal, universal k-means algorithm that automatically selects between different methods of acceleration depending on circumstance. The adaptive distance bounds algorithm presented in this thesis takes the next step toward this goal by keeping a variable number of lower bounds instead of just one or all of them.

2.3.3 Orchard's Method and Sort-means

In an article on fast nearest-neighbor search, Orchard shows how to use the triangle inequality to eliminate codewords from consideration (Orchard 1991). Given a point x and potential nearest-neighbor y, he observes that another point z cannot possibly be nearer to x than y if inequality 2.2 holds. Since Lloyd's algorithm amounts to finding a data point's nearest neighbor among the set of centers, we can easily adapt Orchard's algorithm to k-means, which is exactly what Phillips did in his apparently independent Compare-means algorithm (Phillips 2002).

$$\|z - y\| > 2\|x - y\| \tag{2.2}$$

In fact, Phillips published two k-means algorithms simultaneously: COMPARE-MEANS and SORT-MEANS. True to its name, COMPARE-MEANS compares the distances between centers at each iteration in order to intelligently eliminate point-center computations in the next iteration, via the triangle inequality, exactly as Orchard eliminates candidate nearest-neighbors. This approach yields modest speedups in low dimension; in higher-dimensional spaces, it's often even slower than Lloyd's method due to negligible gains and the overhead of maintaining and testing $\Theta(k^2)$ inter-center distances.

SORT-MEANS improves this process by building a sorted table of inter-center distances, allowing the algorithm to not only skip over centers but to completely stop searching once a distance threshold is crossed. This scheme is quite profitable when k is not too large. On a real-world dataset of BGP updates for internet routers containing roughly 140,000 29-dimensional points, the speedup of the new algorithms was about 11-13 times faster than Lloyd's algorithm for $k = 5{,}000$.

Phillips gives the complexity of standard k-means at each iteration as $O(nkd)$, and claims that his algorithms reduce the per-iteration dependency on k to $O(n\gamma d)$. Here, $\gamma \leq k$ is the average over all points x of the number of centers that are no more than twice as distant from x as x was distant from the center it was assigned to in the previous iteration. Phillips suggested that since Pelleg and Moore's algorithm (Pelleg and Moore 1999) reduced the dependency on n, perhaps a hybrid algorithm could achieve further acceleration relative to both n and k. However, such a combination would suffer from k-d trees' poor performance beyond very low dimension, and we expect the curse of dimensionality to prevent Phillips's γ from being very small when the number of dimensions is large.

2.3.4 *Partial Distance Search*

Partial distance search (PDS) is a method of stopping d-dimensional distance calculations early when we are calculating distance in order to test it against a minimum threshold. If, after considering a subset of the d vector dimensions, we know that the full distance must be sufficiently large, we do not need to calculate the contributions of remaining dimensions. In k-means, we can use PDS to accelerate the key process of calculating a point's closest center.

First, consider the squared distance between two arbitrary vectors $a, b \in \mathbb{R}^d$, as shown in Equation 2.3 below. Given some distance m, suppose we want to know whether $\|a - b\| \leq m$. While summing the right-hand side of Equation 2.3 over each of the d vector dimensions, we can stop early if any partial sum exceeds m^2, because the total sum must also exceed m^2, and hence the distance $\|a - b\| > m$.

$$\|a - b\|^2 = \sum_{p=1}^{d} (a_p - b_p)^2 \tag{2.3}$$

In k-means, we're trying to find a point's nearest center, so we check all centers to see if they're closer than the previously-closest center. While computing point-center distances, we can stop early via PDS if the partial sum exceeds the distance to the closest known center, meaning the center under consideration cannot possibly be the new closest center. Formally, given two vectors $a, b \in \mathbb{R}^d$ and a minimum distance m, PDS looks for minimal d_0 with $1 \leq d_0 < d$ such that inequality 2.4 holds.

$$\sum_{p=1}^{d_0} (a_p - b_p)^2 \geq m \tag{2.4}$$

Savings clearly increase with lower d_0 and lower m. Naively, m begins at ∞ in each k-means iteration and is reduced as additional point-center distances are computed. Note that this algorithm is useless in one dimension, and in general, its benefits are only realized in sufficiently high-dimensional spaces.

Al-Zoubi et al. (2008) propose a better strategy for finding good values of m by persisting its value from one iteration to the next. The authors' empirical analysis

compares Lloyd's algorithm, the naive PDS method, and their modified PDS method, achieving a modest 40-50% time-savings relative to Lloyd's algorithm on datasets from the UCI Repository of Machine Learning Databases. Since this algorithm is easy to implement, PDS may be a useful addition to other k-means implementations with more sophisticated acceleration techniques.

2.4 Alternative k-means Heuristics

There are alternative heuristics for the k-means minimization problem beyond Lloyd's common algorithm. We briefly describe three such methods: core-sets, Hartigan's method, and mini-batch k-means.

2.4.1 Core-sets

Agarwal et al. (2005) describe a core-set as a small subset extracted from a large dataset so that computations may be performed on the subset, giving an approximation for computations on the full set. Unlike previously-discussed approaches, using core-sets constitutes an alternative, non-Lloyd heuristic for the k-means task.

Har-Peled and Kushal (2005) demonstrate the existence of suitable core-sets for the k-means problem. Frahling and Sohler (2006) give an approximation algorithm called CoreMeans, which is competitive with KMhybrid, a simulated annealing variant of Lloyd's algorithm. The core-set acceleration is most pronounced in low dimension, accruing penalties of reduced accuracy in higher dimension. Although this method is inexact, it can run quickly for many different values of k, making it useful for discovering the number of clusters (Frahling and Sohler 2006).

2.4.2 *Hartigan's Method*

Hartigan's method is another non-Lloyd heuristic, which updates centers upon consideration of each point, rather than after each pass over the entire dataset (Hartigan and Wong 1979). More recently, Telgarsky and Vattani resurrected this old approach and show that the set of local optima found by Hartigan's method is a subset of the local optima found by Lloyd's algorithm (Telgarsky and Vattani 2010). In other words, Hartigan's method finds a better, more refined clustering. Practically, experiments by Telgarsky and Vattani reveal a modest 5-10% improvement in the k-means objective function (Equation 1.1).

Essentially, Hartigan's method repeatedly chooses a point's best cluster assignment until a stopping condition is met, where the order of point traversal has little empirical effect (Telgarsky and Vattani 2010). Telgarsky and Vattani derive three formulations of the stopping condition. Holistically, the idea is to calculate the net k-means cost of moving a point from one cluster into another cluster. Relative to Lloyd's algorithm, this expanded stopping criterion means that more points will be reassigned, leading to better clusterings.

Although the authors did not give an empirical runtime comparison, it seems likely that any cluster quality improvement from Hartigan's method is paid for by additional computational expense: Hartigan's method searches a larger space of possible partitions. Telgarsky and Vattani suggest there may be ways to accelerate Hartigan's method, which is still an open question.

2.4.3 *Mini-batch k-means*

While we can make Lloyd's method significantly faster, there are extreme real-world situations where exact acceleration is too conservative, e.g. the sub-second runtime on large datasets required for web-scale clustering at Google (Sculley 2010). Bottou and Bengio (1995) proposed an stochastic gradient descent variant of online

k-means, which computes a descent step one data point at a time. Sculley (2010) revises this approach, simultaneously computing a small set of points: a mini-batch.

Mini-batch k-means exhibits convergence orders of magnitude faster than the original online stochastic gradient descent method. Moreover, the tradeoff between cluster quality and runtime is smoothly adjustable: the more iterations, the better the cluster quality. When extreme speed is more important than cluster quality, mini-batch k-means is a good alternative to accelerated variants of Lloyd's algorithm.

CHAPTER THREE

Methodology

3.1 Adaptive k-means

Our proposed adaptive k-means algorithm combines ideas from Elkan (2003) and Hamerly (2010). As Hamerly notes, these two algorithms have complementary strengths. Elkan primarily achieves speedups by avoiding distance calculations between data points and centers. Hamerly does not avoid as many distance calculations, but he keeps lower bounds on the distance to each point's *second-closest* center; when updating point-center assignments, this lower bound often gives Hamerly's algorithm more opportunities to avoid looping across all centers to determine if one has become the closest, a trade-off which works well in low dimension. If the lower bound holds, then the second-closest center is still further away than the assigned centers, so there is no need to check any of the other centers. Hamerly refers to this process as skipping the innermost loop. However, as dimension increases, the expense of distance calculations begins to outweigh gains from skipping the loop.

A better algorithm would balance these expenses, avoiding many distance calculations and simultaneously avoiding many inner loops. Our adaptive k-means algorithm achieves this goal by keeping a variable number of lower bounds.

3.1.1 Keeping a Variable Number of Bounds

The adaptive k-means algorithm leverages the complementary strengths of Elkan's and Hamerly's algorithms by keeping a *variable* number of lower bounds, which is automatically adjusted at runtime. We never completely skip the innermost loop, but we often break out early when a lower bound holds, giving a partial inner loop. Maintaining a smaller number of lower bounds gives the algorithm potentially

more opportunities to avoid looping, while a greater number of lower bounds gives more opportunities to avoid distance calculations.

For sufficiently large k, this hybrid approach achieves superior efficiency on medium-dimensional data. In particular, we keep the same upper bound on the distance to each point's assigned center, but we track b lower bounds per point on the distance to its b next-closest centers, always ordered by increasing distance, where $1 \leq b \leq k$. In practice, $1 < b < k$, since the mechanism required to maintain a variable number of bounds is more expensive than the mechanism required to maintain all k bounds per point like Elkan or simply 1 bound per point like Hamerly.

3.1.2 Automatically Tuning the Number of Bounds

Keeping b bounds introduces another parameter to k-means. In order to make our algorithm more useful, we built an adaptive tuning mechanism. We generated several training datasets with uniform random distribution and found that runtime is generally good in the interval $\frac{k}{8} \leq b \leq \frac{k}{4}$.

Next, we observed that, for the majority of data points, the first lower bound is enough to avoid distance calculations (line 6 in Algorithm 3.1), the second lower bound is enough for the majority of any remaining points, and so on. This trend becomes stronger as the algorithm progresses, such that in later iterations, some of the outer bounds are *never* used. Figure 3.1 shows the fraction of a particular dataset for which each bound enables us to avoid distance calculations over time.

We use the following tuning strategy. Initially, $b = \frac{k}{4}$. After each iteration, we calculate the number of useful bounds, i.e. bounds that allowed the algorithm to avoid distance calculations, taking the maximum across all data points. We then reduce b to that number of bounds, subject to a minimum of $b = \frac{k}{8}$, saving on subsequent bound-maintenance overhead. Advantageously, this strategy allows us to proceed without knowing the optimal b in advance.

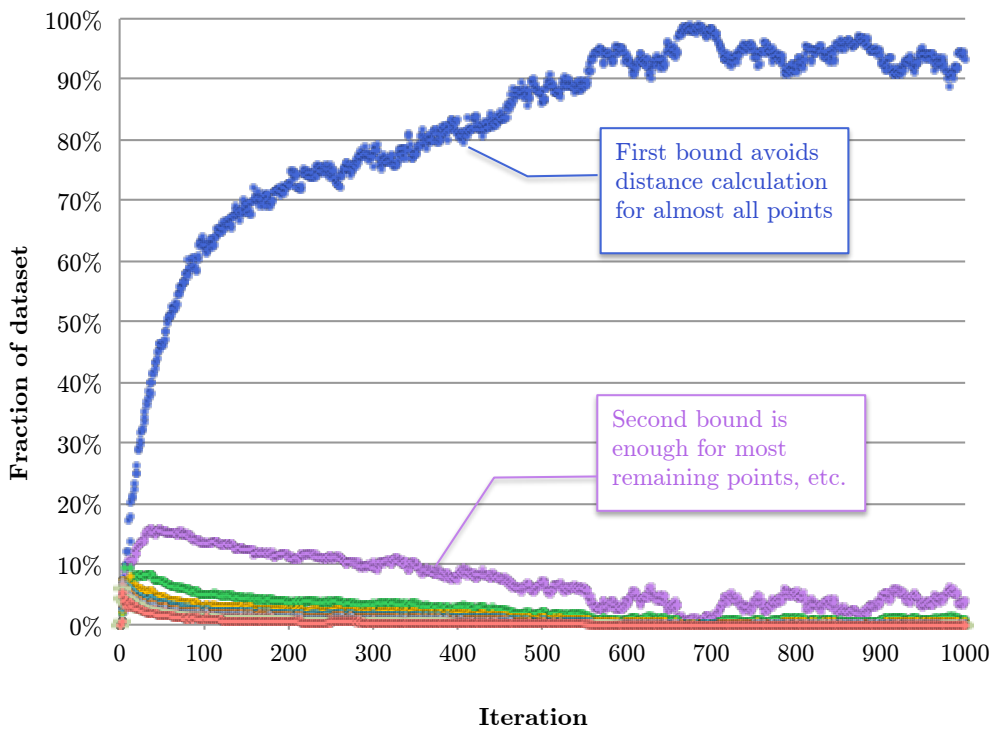

Figure 3.1: Fraction of dataset for which each bound was useful in each iteration on uniform data, with $n = 320{,}000$, $d = 32$, $k = 200$, $b = 50$.

3.1.3 Data Structures

The new algorithm achieves speedups by storing additional information beyond the requirements of Lloyd's algorithm. Given a dataset $\{x_i\}_1^n$ and initial centers $\{c_j\}_1^k$, the algorithm maintains the structures shown in Table 3.1, where the number of lower bounds b is adjusted automatically from $\frac{k}{4}$ to $\frac{k}{8}$.

3.1.4 Pseudocode

We give pseudocode for the new algorithm, omitting minor details and optimizations for clarity. To improve reading ease, in pseudocode we render subscripted parameters using parentheses instead: for example, we display point x_i as $x(i)$. Given a dataset x, initial centers c, and a number of lower bounds b to keep, the algorithm proceeds as described in Algorithm 3.1, which calls the subroutines in Algorithm 3.2

Table 3.1: Data structures maintained by adaptive k-means

Structure	Cardinality	Description
y_i	n	index of point x_i's assigned closet center
u_i	n	upper bound on the distance $\|x_i - c_{y_i}\|$
$a_{i,z}$	$n \times b$	index of point x_i's $(z+1)^{\text{th}}$ closest center
$l_{i,z}$	$n \times b$	lower bound on the distance $\|x_i - c_{a_{i,z}}\|$
t_j	k	distance that center c_j travelled in the last iteration

and Algorithm 3.3. Naturally, Algorithm 3.1 is similar to Hamerly's and Elkan's algorithm. In our innermost loop (lines 5-10), we are hoping the upper bound $u(i)$ satisfies one of our b lower bounds. If the z^{th} lower bound works, we only need to re-sort the first $(z+1)$ closest centers rather than the entire set c.

Algorithm 3.1 ADAPTIVE-k-MEANS(x, b, c)

```
1: for i = 1 to |x| do
2:     SORT-CENTERS(x(i), b, c)
3: end for
4: while not converged do
5:     for i = 1 to |x| do
6:         for z = 1 to b do
7:             {Check if this lower bound holds}
8:             if u(i) ≤ l(i, z) then
9:                 {Re-sort only assigned center y and z next-closest centers}
10:                r ← {y_i, a(i, 1), ..., a(i, z)}
11:                SORT-CENTERS(x(i), z, r)
12:                Skip to next iteration of for i loop
13:            end if
14:        end for
15:        {All lower bounds failed, so sort all centers}
16:        SORT-CENTERS(x(i), b, c)
17:    end for
18:    Move centers to centroid of assigned points, updating t
19:    UPDATE-BOUNDS()
20: end while
```

Algorithm 3.2 SORT-CENTERS$(x(i), q, r)$

```
1: Sort r by increasing distance from x(i)
2: y ← r(1)
3: u(i) ← ||x(i) − y||
4: for z = 1 to q do
5:       a(i, z) ← r(z + 1)
6:       l(i, z) ← d(x(i), c(a(i, z)))
7: end for
```

Algorithm 3.3 UPDATE-BOUNDS()

```
1: m ← max_{1≤j≤k} t(j)
2: for i = 1 to |x| do
3:       u(i) ← u(i) + t(y)
4:       l(i, b) ← l(i, b) − m
5:       for z = b − 1 to 1 do
6:             l(i, z) ← l(i, z) − t(a(i, z))
7:             if l(i, z) > l(i, z + 1) then
8:                   l(i, z) ← l(i, z + 1)
9:             end if
10:      end for
11: end for
```

3.1.5 Updating Distance Bounds

Algorithm 3.3 is more subtle. We pre-compute m (line 1) in order to update the outermost lower bound (line 4). Lower bounds shrink by the distance moved by their corresponding center just as in Elkan's algorithm, but the outermost bound must also account for the movement of the $(k - b - 1)$ farthest centers not tracked by our algorithm. Computing this bound tightly would be expensive, so we settle for m, which we only need to compute once per iteration. Finally, we sacrifice additional tightness in order to force the bounds to stay in increasing order (lines 6-8), which enables us to search the b bounds efficiently in subsequent iterations.

3.2 Annulus k-means

Our proposed annulus k-means algorithm is a variant of Hamerly's algorithm (Hamerly 2010), which keeps an upper bound on the distance between each data point and its assigned center, plus a single lower bound on the distance to the data point's second-closest center. As long as the upper bound on the closest center is less than the lower bound on the second-closest center, the data point's center assignment cannot change, avoiding distance calculations. However, when the bounds overlap for

a particular data point, Hamerly's algorithm must compute distances from that point to all centers in the innermost loop to find the center with minimum distance.

3.2.1 Reducing the Search Space

Annulus k-means efficiently prunes the search space of this innermost loop over all centers, by considering only those centers c' that satisfy Inequality 3.1, where x is a given data point and c is its previously assigned closest center. Except in high dimension, annular search avoids about 50-90% of the distance calculations made by Hamerly's algorithm.

$$\big|\, \|x\| - \|c'\| \,\big| \leq \|x - c\| \tag{3.1}$$

This inequality defines an annular region centered at the origin, which must contain any centers c' closer to x than c. If Inequality 3.1 is not satisfied, then

$$\|x - c\| < \big|\, \|x\| - \|c'\| \,\big| \tag{3.2}$$

$$\leq \|x - c'\|. \tag{3.3}$$

Here we have used the reverse triangle inequality to show centers c' that fail to satisfy Inequality 3.1 cannot possibly be closer to x than c. Therefore, we do not need to calculate the explicit distance $\|x - c'\|$. Figure 3.2 illustrates how the annular region can be used to exclude centers from consideration as the closest to x.

Huang et al. applied this same geometric reasoning to nearest neighbor calculations for vector quantization (Huang et al. 1992), constructing an annulus that avoids explicit distance calculations. However, the annulus has limited usefulness for finding nearest neighbors, because it depends on the estimate c used to construct it.

The success of the approach of Huang et al. relies on their specific application: image compression. To estimate a point's nearest neighbor, their algorithm supposes that one point is like the next, i.e. that one pixel is likely similar to adjacent pixels

in the image. However, this convenient structure is not present in arbitrary datasets, and the algorithm fails to provide speedup for general nearest neighbor problems.

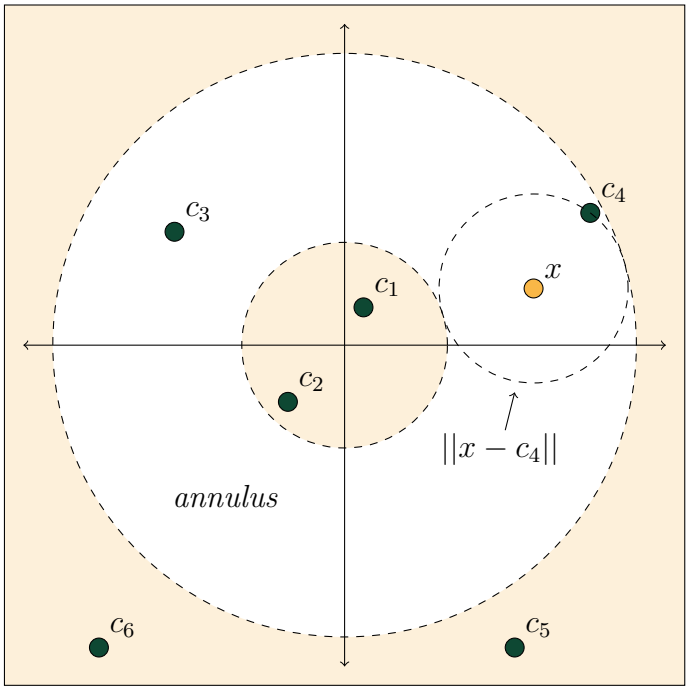

Figure 3.2: The annular region (white ring centered at origin) bounds where the closest center for x might be. Centers c_j are numbered by their distance from the origin. Point x has c_4 as its previously-closest center, so the width of the annulus is $2||x - c_4||$ (dashed circle centered at x).

3.2.2 Algorithm Description

Since $\|x\|$ is fixed and $\|c\|$ changes once per iteration, we can leverage this information efficiently in k-means: each iteration, we keep an auxiliary list of centers sorted by their norms. Whenever Hamerly's distance bounds fail to avoid a distance calculation, we use binary search on the sorted list of center norms to find a set of candidate centers satisfying Inequality 3.1.

3.2.3 Pseudocode

We give pseudocode for our proposed k-means variant with annular search in Algorithm 3.4. Where possible, we maintain consistency with Hamerly (2010). Likewise, we assume a distance metric $d(\cdot, \cdot)$ and use the important data structures from Hamerly's algorithm shown in Table 3.2, introducing the new structure b_i.

Table 3.2: Data structures maintained by annulus k-means

Structure	Cardinality	Description
a_i	n	index of point x_i's assigned closet center
u_i	n	upper bound on the distance $\|x_i - c_{a_i}\|$
b_i	n	index of point x_i's second-closet center (when l_i tight)
l_i	n	lower bound on the distance $\|x_i - c_{b_i}\|$
s_j	k	distance from center c_j to its closest other center

As in the pseudocode for adaptive k-means, to improve reading ease, we render subscripted parameters in our pseudocode using parentheses instead: for example, we display bound u_i as $u(i)$ and distance s_j as $s(j)$.

We also omit minor implementation details for clarity. For example, we make comparisons using squared distances to avoid taking roots where possible. Also, finding the set J from Line 12 in Algorithm 3.4 is implemented as a binary search over the sorted centers from Line 4. Additionally, to ensure that our algorithm produces *identical* clusterings to Lloyd's algorithm, we explicitly break ties in closest-center calculations. For example, points are occasionally the *same* distance from two would-be-closest centers; here we must choose the same center as Lloyd, namely the one with the lower index.

Algorithm 3.4 ANNULUS-k-MEANS(x, c)

Compute $\|x(i)\|$ for all i
while not converged **do**
  Compute $s(j)$ and $\|c(j)\|$ for all j
  Sort centers by increasing norm
  for $i = 1$ **to** $|x|$ **do**
    $m \leftarrow \max(s(a(i))/2, l(i))$
    if $u(i) \leq m$ **then** continue to next i
    $u(i) \leftarrow d(x(i), c(a(i)))$
    if $u(i) \leq m$ **then** continue to next i
    $l(i) \leftarrow d(x(i), c(b(i)))$
    $r \leftarrow \max(l(i), u(i))$
    $J \leftarrow \left\{ j \mid \left| \|x(i)\| - \|c(j)\| \right| \leq r \right\}$
    for all $j \in J$ **do** {Search the annulus}
      if $d(x(i), c(j)) < u(i)$ **then**
        {New closest center found}
        $l(i) \leftarrow u(i)$
        $b(i) \leftarrow a(i)$
        $u(i) \leftarrow d(x(i), c(j))$
        $a(i) \leftarrow j$
      else if $d(x(i), c(j)) < l(i)$ **then**
        {New second-closest center found}
        $l(i) \leftarrow d(x(i), c(j))$
        $b(i) \leftarrow j$
      end if
    end for
  end for
  Move each center to its centroid
  Update upper and lower bounds
end while

3.3 Theoretical Analysis of Annulus k-means

We can derive theoretical results about the effectiveness of annular search in d-dimensional space. If we make the simplifying assumption that data is uniformly distributed in the unit d-ball centered at the origin, then we can obtain an analytic solution for the expected volume of the data space eliminated from the nearest-center search, the predictions of which we confirm by Monte Carlo simulation. If our data is instead uniformly distributed throughout the unit hypercube $[-0.5, 0.5]^d$ centered at the origin, we lack an analytic solution for all cases, so we give only results from Monte Carlo simulation for that scenario.

3.3.1 Clustering in a Spherical Space

Assuming data and k centers are uniformly distributed in the unit d-ball centered at the origin, namely the set of points $\{p \in \mathbb{R}^d \mid \|p\| \leq 1\}$, we can determine the expected portion of the dataset eliminated by the annulus in Line 12 of Algorithm 3.4, giving an estimate of our algorithm's performance relative to Hamerly's algorithm.

Now, the volume of a ball in d dimensions with radius r is

$$V_{\text{ball}}(r) = \frac{r^d \pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)}, \tag{3.4}$$

where $\Gamma(z)$ is the extended factorial function, defined by $\Gamma(z+1) = \int_0^\infty t^z e^{-t} \mathrm{d}t$. Ideally, we want to divide this volume into k equal partitions, each with volume $\frac{1}{k} V_{\text{ball}}(1)$. Since this fractional volume corresponds naturally to a ball of radius

$$u = k^{-\frac{1}{d}} \tag{3.5}$$

we make the simplifying assumption that we can somehow divide the ball into k smaller balls, each of radius u, giving the same total volume. This assumption enables us to easily propose an upper bound on the expected distance u from a point

to its nearest center: the radius $u = k^{-\frac{1}{d}}$ of each of the k balls. In fact, as dimension increases, the upper bound becomes an increasingly good estimate of expected distance to a point's nearest center.

3.3.2 Constructing the Annular Search Region

We now show how to calculate the volume of the annular search region for an arbitrary point x. In general, the volume of an annulus in d dimensions with outer radius r and inner radius s is given by

$$V_{\text{annulus}}(s, r) = \frac{(r^d - s^d)\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2} + 1\right)}. \tag{3.6}$$

Assuming the expected distance from x to its nearest center is fixed at $u = k^{-\frac{1}{d}}$ allows us to explicitly construct the annular search region for an arbitrary point x in the ball. To satisfy Inequality 3.1, the annulus must have an outer radius of $\|x\| + u$ and an inner radius of $\|x\| - u$.

Balls conveniently exhibit radial symmetry, so to determine the expected volume of the annular search region over all points x in the d-dimensional ball, we need only consider the radius $r = \|x\|$. Therefore, the expected volume of the annular search region (ASR) can be expressed as

$$V_{\text{ASR}} = \int_0^1 v(r)\rho(r)\mathrm{d}r \tag{3.7}$$

where we have combined the volume of annulus $v(r)$ at radius r with the probability density $\rho(r)$ of a point with radius r within the ball.

What is the probability density $\rho(r)$ in the d-ball? It is easy to show that points in the ball are distributed proportionally to r^{d-1} by calculating the surface area of a sphere of radius r, which is intuitively the first derivative of the d-ball's volume. This multi-dimensional surface area calculates the volume of points in the d-ball at a given radius, i.e. on the sphere surface.

Thus we have $\rho(r) \propto r^{d-1}$, but for $\rho(r)$ to be a valid probability density function we also need to satisfy the definite integral

$$\int_0^1 \rho(r)\mathrm{d}r = 1, \tag{3.8}$$

so we multiply by the constant d to get the probability density function

$$\rho(r) = dr^{d-1}. \tag{3.9}$$

Now we have expressed the probability density for points of radius r, but we still need to compute the associated volume $v(r)$ of the annular search region at r. However, computing $v(r)$ is somewhat tricky, so we split into cases. At its simplest,

$$v(r) = \frac{[(r+u)^d - (r-u)^d]\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)}. \tag{3.10}$$

For $r \in [u, 1-u]$, the annulus is fully formed, and so the above formula is valid. However, we must deal with two degenerate cases, namely when the outer radius exceeds 1 and when the inner radius collapses to 0.

For $r \in [0, u]$, the inner radius $(r-u)$ collapses to zero, so we must adjust the volume calculation accordingly, giving $\tilde{v}(r)$:

$$\tilde{v}(r) = \frac{(r+u)^d \pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)} \tag{3.11}$$

For $r \in [1-u, 1]$, the outer radius must not exceed the unit ball, so we modify the volume function again, giving $\hat{v}(r)$:

$$\hat{v}(r) = \frac{[1 - (r-u)^d]\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)} \tag{3.12}$$

We now have the expected volume of the ASR in three pieces:

$$V_{\mathrm{ASR}} = \int_0^u \tilde{v}(r)\rho(r)\mathrm{d}r + \int_u^{1-u} v(r)\rho(r)\mathrm{d}r + \int_{1-u}^1 \hat{v}(r)\rho(r)\mathrm{d}r \tag{3.13}$$

These integrals can be evaluated analytically in terms of the Gaussian hypergeometric function ${}_2F_1$ in the general case. Alternatively, for fixed d, the underlying

indefinite integrals are polynomial in r and can be expressed in closed form. For our purposes, numerical integration is satisfactory.

Finally, we are trying to prune the overall search space, so we report the fraction α of the ball's volume avoided by the annular search, namely the region outside the annulus, giving a measure of expected performance:

$$\alpha = 1 - \frac{V_{\text{ASR}}}{V_{\text{ball}}(1)} \tag{3.14}$$

The results in Table 3.3 show that annulus k-means has the best chance of speedups in low dimension, under the pessimistic assumption that points are uniformly distributed throughout a spherical space.

Table 3.3: Expected fraction of the unit ball eliminated by the annulus, according to numerical integration of our analytic result for $k = 64$ and $1 \leq d \leq 10$. An avoidance fraction of $\alpha = 1$ would be ideal: eliminating the entire search region.

Dimension	Fraction of search space avoided (α)
2	0.6978
3	0.3500
4	0.1322
5	0.0399
6	0.0100
7	0.0022
8	0.0004
9	0.0001
10	0.0000

3.3.3 Monte Carlo Simulation

To confirm our theoretical results in spherical space, we ran Monte Carlo simulations and calculated the average fraction of distance calculations avoided by the annulus. The results of our simulations very closely match and confirm the validity of our analytic results. Figure 3.3 compares analytic and simulated results for $k = 16$ and $k = 256$. Since the expected distance to a point's nearest center decreases with larger k, more calculations are avoided for $k = 256$.

We also performed Monte Carlo simulations in the unit hypercube $[-0.5, 0.5]^d$, giving somewhat different results but producing the same exponential performance decay as in the spherical case, as displayed in Figure 3.4. We lack an analytic solution because of the difficulty of calculating the degenerate annular volumes $\tilde{v}$ and $\hat{v}$ in a hypercube and subsequently integrating over all points. However, Monte Carlo results in the hypercube are not much different from results in the hypersphere, so we believe our current analytic solution reasonably explains the algorithm's performance.

3.4 Heap k-means

We now discuss a variant of Hamerly's algorithm that attempts to improve the algorithm's dependence on n by restructuring the algorithm within a given iteration to use k heaps. The basic structure of Hamerly's algorithm is a loop over all points. For each point, we perform a distance bound check. If the check fails, then we need a loop over all centers to find the nearest one.

3.4.1 Avoiding the Outer Loop Over All n Points

Using the naming conventions for data structures as in Table 3.2, Hamerly's primary distance bound check for point x_i asks whether $u_i \leq l_i$. If so, then x_i is already assigned to its closest center, so we can skip any further computation for this point. If we rephrase that inequality as $0 \leq (l_i - u_i)$ and somehow have the points sorted by increasing value of $(l_i - u_i)$, then as soon as we discover a point passes the bounds check, we can ignore the rest of the dataset.

Sorting all n points once per iteration is prohibitively expensive, and even if it weren't, we would still need to update all n bounds l_i and u_i, regardless of whether the assignment changed or not, so we have not yet improved the dependency on n.

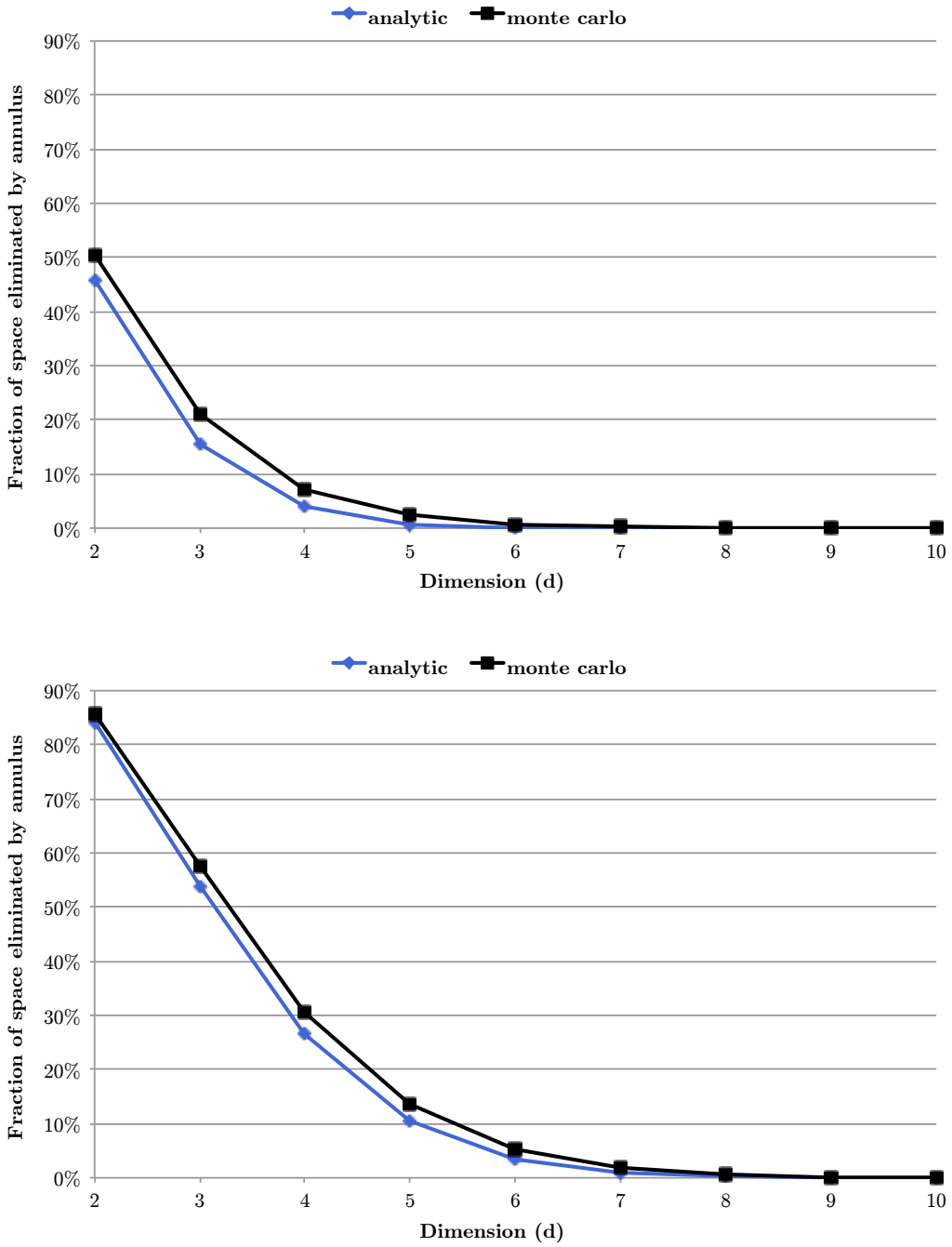

Figure 3.3: Theoretical fraction of search space avoided by the annulus given by our analytic solution in the unit ball, and average results from 1000 Monte Carlo simulations of 1000 data points each for various dimension. The top graph shows $k = 16$, and the bottom shows $k = 256$.

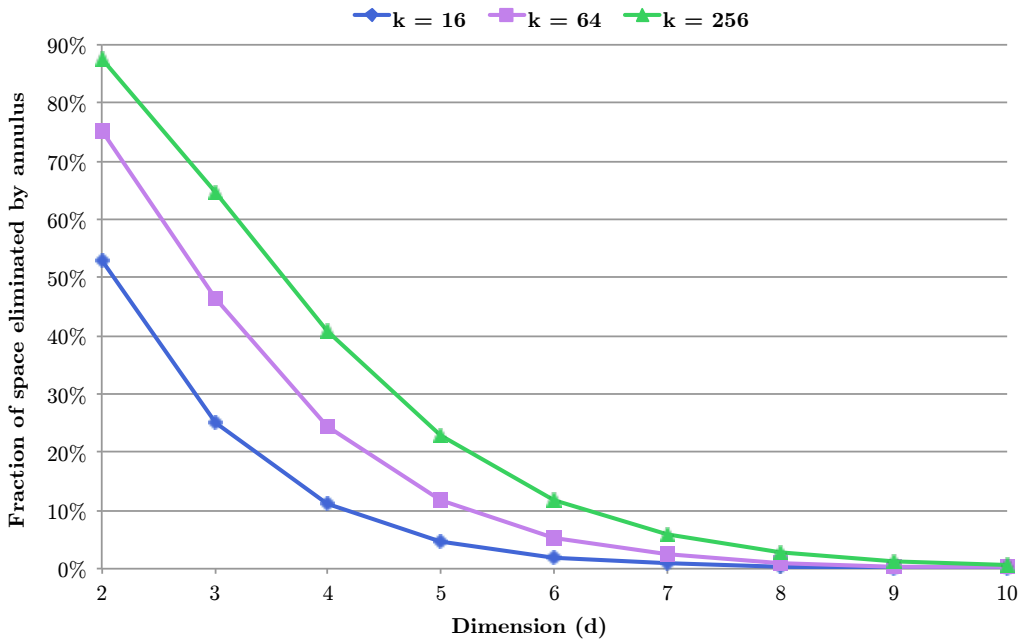

Figure 3.4: Expected fraction of search space eliminated by the annulus in the unit hypercube $[-0.5, 0.5]^d$, averaged over 1000 Monte Carlo simulations of 1000 data points each for various dimension, with $k = 16, 64, 256$.

3.4.2 One Heap Per Cluster

Our proposed solution to this problem is to shift the responsibility for maintaing distance bounds from points to centers. We create k min-heaps: one for each center. Initially, we push the difference $(l_i - u_i)$ for each point onto the heap corresponding to its assigned closest center. In the first iteration, we handle each center by popping values off its heap until $(l_i - u_i) \geq 0$, at which point we know we can skip any remaining assigned points.

From one iteration to the next, however, we must update distance bounds. Instead of an $\Theta(n)$ pass over all points, we combine the updates for l_i and u_i into a single update that depends only on point x_i's assigned center. To update an upper bound in Hamerly's algorithm, we add the distance moved by the assigned closest center. To update a lower bound in Hamerly's algorithm, we subtract the greatest distance moved by any other center. Since these updates depend only on centers and

not on x_i, we can efficiently combine them by defining a new structure $\{h_j\}_1^k$, where h_j represents a distance bound associated with the heap for center c_j.

Initially, $h_j = 0$. After each iteration, we update this distance bound by adding the distance moved by c_j *and* the greatest distance moved by any other center. When popping values of the form $(l_i - u_i)$ off a given heap, we check to see if $h_j \leq (l_i - u_i)$. Ideally, this inequality holds and we can skip the rest of the points on this heap.

When the bound fails, we tighten the distances u_i and l_i and assign the point x_i to its new closest center c_{a_i} and push it onto the corresponding heap, with value $v = (l_i - u_i) + h_{a_i}$. Note that we have added the current heap bound h_{a_i} for the newly assigned closest center, since that bound is updated after each iteration. This way, when we pop value v off the heap in the next iteration and perform the comparison $h_j \leq v$, we are essentially checking $0 \leq (l_i - u_i)$ as if l_i and u_i had been updated independently, like in Hamerly's algorithm. In this heap-based variant, we have consolidated the upper and lower updates into h_j, hoping that the reduced dependency on n will be worth the overhead of keeping k heaps.

3.4.3 Pseudocode

We give pseudocode for this heap-based version of k-means in Algorithm 3.5. As in previous pseudocode listings, we render subscripted parameters in our pseudocode using parentheses instead: for example, we display bound h_j as $h(j)$.

Line 21 explicitly breaks ties in nearest-center calculations to guarantee consistency with Lloyd's algorithm. Usually we omit this type of logic from pseudocode (adaptive k-means and annulus k-means also break ties), but for heap-based k-means explicit tie-breaking is critical: without it, a data point can enter an infinite reassignment loop, passing from one heap to another and back repeatedly because it is equidistant from both centers.

Algorithm 3.5 HEAP-k-MEANS(x, c)

```
1: while not converged do
2:   for j = 1 to k do
3:     while heap j is not empty do
4:       Pop off heap j's minimum value v {associated with point x(i)}
5:       if h(j) ≤ v then
6:         break
7:       end if
8:       u ← d(x(i), c(j)) {tighten upper bound}
9:       l ← ∞
10:      for q = 1 to k do
11:        if d(x(i), c(q)) < u then
12:          {New closest center found}
13:          l ← u
14:          u ← d(x(i), c(q))
15:          a(i) ← j
16:        else if d(x(i), c(q)) < l then
17:          {New second-closest center found}
18:          l ← d(x(i), c(q))
19:        end if
20:      end for
21:      if l = u then
22:        {Tie for closest center}
23:        Choose the center with least index as a(i)
24:      end if
25:      v ← h(a(i)) + (l − u)
26:      Push point x(i) onto heap a(i) with value v
27:    end while
28:  end for
29:  Move each center to its centroid
30:  for j = 1 to k do
31:    {Update heap bound}
32:    Add to h_j the distance moved by center c_j
33:    Add to h_j the greatest distance moved by any other center
34:  end for
35: end while
```

3.5 Experimental Design

There are many factors affecting k-means runtime; to give a clear picture of our algorithm's performance, we ran clustering experiments to address the resilience of our accelerated methods' performance in several orthogonal domains: separability, cardinality, dimensionality, number of clusters, and initialization scheme.

3.5.1 Speedup

We report the performance of an accelerated algorithm as *speedup* relative to Lloyd's algorithm, i.e. the runtime of Lloyd's algorithm divided by the runtime of the accelerated algorithm, on the same task. Higher speedup means better performance; an accelerated algorithm seeks to maximize speedup.

In a given experiment, we supply the same initial center locations to each k-means algorithm; therefore, they all compute the exact same clustering as Lloyd's algorithm, in the same number of iterations. However, *different experiments* may run for different numbers of iterations, which makes it difficult to compare raw runtime. In addition to measuring an algorithm's level of acceleration, reporting our results as a speedup ratio conveniently compensates for the natural variability in iterations.

3.5.2 Experiments

In the best case, the dataset we are attempting to cluster is well-separated into natural clusters. Accelerated algorithms typically exploit geometric reasoning about the structure in a dataset, so we expect more speedup on more clustered, real-world datasets. In the worst case, a uniform random dataset exhibits essentially no structure, where we do not quickly converge on final center assignments, giving a useful performance baseline.

Since we are accelerating k-means, we want our algorithms to perform well on datasets of all sizes but especially on larger ones. Ideally, the relative performance

of an accelerated method compared to Lloyd's algorithm will increase as the dataset size n increases (due to diminishing relative overhead), so we perform experiments on datasets of varying size to confirm this expected good behavior.

We also want fast k-means clustering in both low- and high-dimensional spaces. In general, the curse of dimensionality makes accelerated clustering in high-dimension difficult. Nevertheless, most accelerated algorithms tend to perform best within a certain dimensional range, so it is important to use datasets of varying dimension. Notably, many high-dimensional datasets can be clustered effectively using low-dimensional means after preprocessing via an inexpensive dimension reduction technique, such as random projection (Dasgupta 2000). Consequently, fast low-dimensional algorithms are quite useful in practice, even if the original space of the dataset is very high-dimensional.

Additionally, we compare how different algorithms fare for different settings of k. The speedup of a well-behaved acceleration would not deteriorate as the number of centers grows large. Nevertheless, some algorithms tend to perform better for small k and others for large k, complicating comparisons between accelerated methods.

Finally, we compare the performance of our algorithms using k-means++ to determine initial centers versus using random selection. Since k-means++ represents a best-practice initialization method, it is appropriate to judge competing acceleration techniques using this practical initialization method. Still, we want to confirm that if using random initialization has any effect on algorithm speedup, it typically results in a perceived improvement compared to Lloyd, because k-means++ tends to reduce the necessary amount of clustering work in all cases.

3.5.3 Execution Environment

Experiments were conducted on the same Intel Xeon 3.0GHz machine running Linux 2.6.9 with 8GB of memory. Runtime was measured using `getrusage()` and

memory use was measured from `/proc/[PID]/statm`. Except where we explicitly tested the effects of random initialization, initial center locations were chosen using k-means++. The same initial locations were then used to seed each algorithm tested. Raw runtime in seconds was post-processed into speedup; tables and graphs are then automatically generated from this data.

3.5.4 Algorithms and Datasets

We compared our accelerated algorithms to Lloyd's algorithm and to several competing accelerated methods, testing a total of eight algorithms. Table 3.4 lists the algorithms used in our experiments.

Table 3.4: Algorithms used during experiments

Algorithm	Description
annulus	our annular search method
adaptive	our adaptive distance bounds method (Drake and Hamerly 2012)
orchard	Orchard's method (Orchard 1991)
sort	Sort-means (Phillips 2002)
heap	our heap-based variant of Hamerly's algorithm
hamerly	Hamerly's algorithm (Hamerly 2010)
elkan	Elkan's algorithm (Elkan 2003)
lloyd	Lloyd's algorithm: standard, naive k-means (Lloyd 1982)

To test these algorithms, we used uniform random datasets and naturally-clustered datasets. We generated uniform random datasets varying in cardinality n and dimension d. With three values of n and nine values of d, there are total of 27 uniform datasets, as outlined in Table 3.5.

For consistency and ease of comparison, we tested our algorithms on the same datasets used in other works (Drake and Hamerly 2012; Hamerly 2010; Elkan 2003; Phillips 2002). Table 3.6 describes the six naturally-clustered datasets we used.

Table 3.5: Uniform random datasets used in experiments, showing three values of cardinality (n) and nine values of dimension (d), for a total of 27 datasets.

Parameter	Values
n	100,000; 200,000; 400,000
d	2, 4, 8, 16, 32, 64, 128, 256, 512

Table 3.6: Clustered datasets used in experiments, ordered by increasing dimension

Name	n	d	Description
birch	100,000	2	10x10 grid of Gaussian clusters
mnist50	60,000	50	random projection of mnist784
covtype	581,012	54	soil cover measurements
kddcup98	95,412	56	fundraising response rates
kddcup04	139,658	74	protein homology
mnist784	60,000	784	handwritten digits training set

CHAPTER FOUR

Results and Analysis

We now present the empirical results of our clustering experiments. Several trends emerge. In high dimension, approximately $d > 200$, Elkan's algorithm remains the dominant performer. In medium dimension, roughly $20 \leq d \leq 200$, both our annulus k-means and adaptive k-means algorithms perform very well, with annulus k-means performing best on well-separated, naturally-clustered data. In low dimension, annulus k-means competes with Hamerly's algorithm and Sort-means, depending on the value of k and dataset separability. Our heap k-means algorithm turns out to be slightly slower than Hamerly's algorithm in most cases. Finally, Orchard's method performs poorly overall, and like Sort-means, it often performs *worse* than Lloyd's algorithm for high k or in high dimension.

The remainder of this chapter is a performance breakdown for various clustering conditions. We discuss the negligible impact of initialization scheme on our algorithms' performance; speedup improves slightly with greater dataset cardinality; the effects of different k and d vary from algorithm to algorithm; and speedup tends to improve on better-separated datasets. We also analyze time-memory tradeoffs: Elkan's algorithm uses more memory than adaptive k-means, which in turn uses more memory than annulus k-means, Hamerly's algorithm, and Sort-means.

4.1 Initialization Scheme

Poor initialization of centers usually leads to long runtime and inferior cluster quality, so any practical k-means user will select an appropriate mechanism for choosing initial center locations. Since its introduction in 2007, k-means++ initialization has become standard practice. For this reason alone, it makes sense to compare our

algorithms' performance using k-means++ to initialize centers. However, to assuage any doubt that k-means++ affords our methods some unfair advantage, Figure 4.1 plots algorithm speedup using random center initialization relative to speedup using the k-means++ approach.

Overall, we observe little difference in speedup. If anything, we would expect random initialization to force Lloyd's algorithm to be inefficient, thus inflating our speedup measurements. k-means++ usually does a good job picking centers, leaving less work for our accelerated algorithms to do, making it relatively *more difficult* for them to gain traction over the standard algorithm. Consequently, we are not surprised to see that random initialization generally improves perceived speedup relative to k-means++, i.e. giving ratios ≥ 1.0 in Figure 4.1 below.

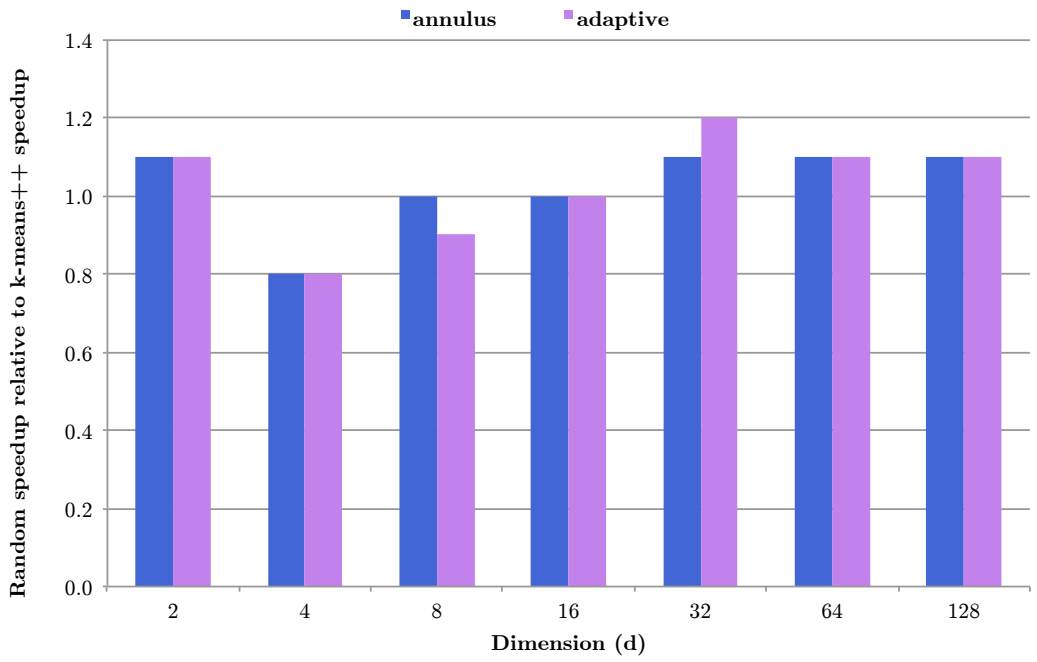

Figure 4.1: Speedup using random selection of initial centers relative to speedup using k-means++ on uniform random data of varying dimension, with fixed $k = 16$ and $n = 400{,}000$. Since k-means++ does such a good job at initialization, it makes sense for this ratio to be greater than 1 most of the time. Results are similar for different k, d, and n.

4.2 Cardinality

While increased dataset cardinality clearly increases runtime, a somewhat more interesting question asks whether cardinality positively affects *speedup*. Since our accelerated methods incur overhead costs for maintaining their various data structures, we would like to see a relative improvement in speedup for larger datasets. Figure 4.2 plots speedup for random uniform data, $d = 32$ and $k = 64$, at three different levels of n. Overall, speedup improves gradually with larger n, except for Orchard's method and SORT-MEANS. However, the overhead for these two algorithms is dependent solely on k, so it makes sense for their speedup to be invariant to n.

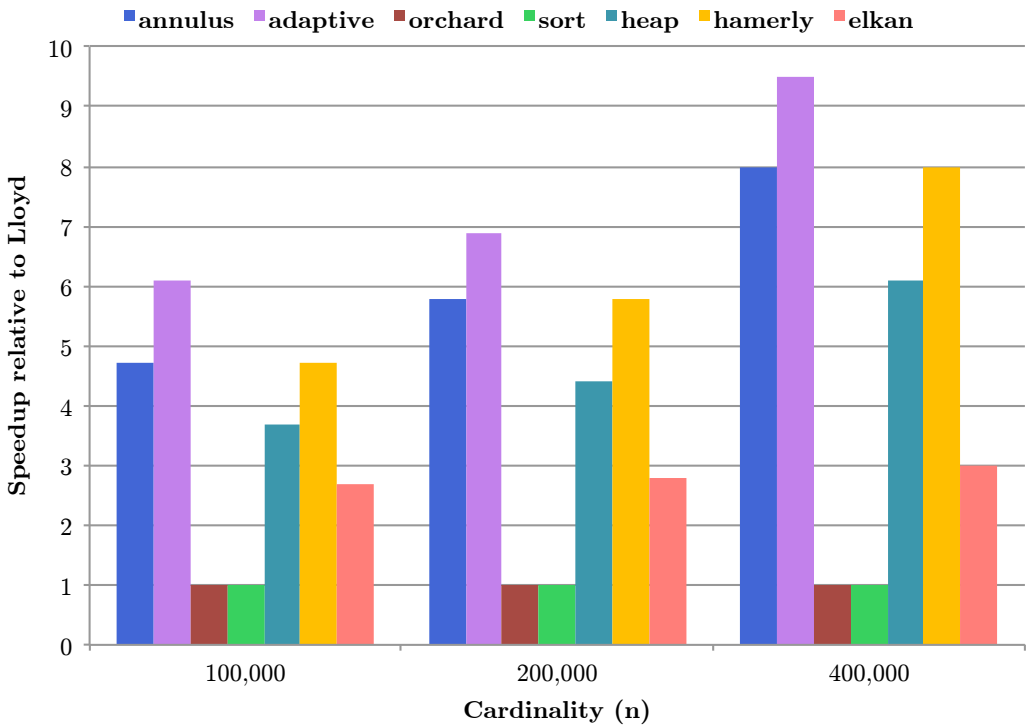

Figure 4.2: Speedup relative to Lloyd on uniform random data for three different cardinalities n, with fixed $d = 32$ and $k = 64$. As cardinality increases, accelerated algorithms tend to do better due to diminishing relative overhead. In each case presented here, both Orchard's method and SORT-MEANS show a speedup factor of 1.0, accurate to the nearest tenth.

4.3 Dimension and Number of Centers

The effects of dimensionality and the number of centers vary from algorithm to algorithm, creating niches of d and k where one algorithm dominates others. Since this landscape is complex, we divide our discussion into three parts: uniform random data, clustered data, and algorithm-specific performance.

4.3.1 Uniform Random Data

Uniform random data is effectively worst-case input for k-means clustering: there really aren't any meaningful clusters to be found (Moore 2000). However, this type of data makes for a useful baseline comparison of exact accelerated algorithms. Since it often takes many k-means iterations to converge on a final clustering, uniform data forces the algorithm to work hard. Moreover, we can easily generate datasets of different dimension d and cardinality n.

We display experimental results both graphically and in tabular format. Since there are many results to show, we have split tables by dimension. Table 4.1 lists algorithm speedup relative to Lloyd's algorithm for $2 \leq d \leq 16$, and Table 4.2 gives speedup for $32 \leq d \leq 512$. In general, Elkan's algorithm dominates high dimension, adaptive k-means dominates medium dimension, and annulus k-means competes with Hamerly's algorithm and Sort-means in low dimension, depending on k.

The performance of our heap k-means method is consistently underwhelming compared to Hamerly's algorithm. Orchard k-means does poorly in general, and although Sort-means slightly outpaces annulus k-means in very low dimension, Sort-means deteriorates severely as dimension increases, performing worse than Lloyd's algorithm in many cases. Annulus k-means does not share this deficiency.

Since performance depends on both d and k, we show the regions of algorithm superiority in Figure 4.3. This landscape provides a rough guide to the fastest modern k-means algorithms, *on worst-case data*.

Table 4.1: Speedup of accelerated algorithms relative to Lloyd's algorithm while clustering uniform random data for various k and dimension $2 \leq d \leq 16$, with fixed dataset size $n = 400{,}000$. Bold values show the best speedup in each experiment.

Dimension	Algorithm	Number of centers (k)					
		16	32	64	128	256	512
$d = 2$	annulus	5.08	8.57	15.70	28.77	**48.88**	**67.88**
	adaptive	2.17	2.79	3.96	4.75	5.52	5.25
	orchard	1.58	1.66	1.62	1.48	1.48	0.96
	sort	5.25	**9.11**	**16.71**	**29.96**	48.38	53.77
	heap	3.60	5.55	8.28	10.64	11.47	9.66
	hamerly	**5.54**	8.18	11.96	15.91	17.45	15.19
	elkan	0.86	0.82	0.81	0.84	0.87	0.83
$d = 4$	annulus	5.03	**6.65**	**8.98**	**12.92**	**20.23**	22.31
	adaptive	2.74	3.01	3.49	4.81	6.42	6.87
	orchard	1.19	1.40	1.55	1.66	1.78	1.03
	sort	2.20	3.45	6.06	10.44	18.63	**22.80**
	heap	4.31	4.74	5.37	6.36	8.23	7.25
	hamerly	**5.67**	6.65	7.47	8.33	10.34	8.86
	elkan	0.91	0.74	0.73	0.75	0.74	0.74
$d = 8$	annulus	3.45	6.10	**7.25**	**4.78**	**6.54**	**7.32**
	adaptive	1.94	3.67	4.76	3.80	5.24	7.04
	orchard	0.57	0.60	0.70	1.28	1.19	1.20
	sort	0.88	0.97	1.25	2.79	3.09	4.21
	heap	2.23	4.70	4.56	3.49	4.57	4.50
	hamerly	**3.72**	**6.31**	7.00	4.29	5.56	6.03
	elkan	1.17	0.83	1.07	1.24	0.82	0.79
$d = 16$	annulus	6.57	8.22	7.40	7.47	6.42	4.29
	adaptive	4.40	5.70	6.36	**8.03**	**9.49**	**8.13**
	orchard	0.97	0.93	0.88	0.86	0.86	0.78
	sort	0.96	0.91	0.87	0.82	0.78	0.78
	heap	6.94	6.84	5.83	5.98	5.50	4.10
	hamerly	**7.77**	**8.68**	**7.56**	7.37	6.40	4.52
	elkan	1.85	1.81	1.80	1.78	1.78	1.74

Table 4.2: Speedup of accelerated algorithms relative to Lloyd's algorithm while clustering uniform random data for various k and dimension $32 \leq d \leq 512$, with fixed $n = 400{,}000$. Bold values show the best speedup in each experiment. *Elkan's memory requirement exceeds 3GB per-process limit on our test machine.

Dimension	Algorithm	Number of centers (k)					
		16	32	64	128	256	512
$d = 32$	annulus	7.20	9.92	8.02	6.64	4.72	3.55
	adaptive	6.15	9.00	**9.54**	**10.28**	**9.89**	**8.53**
	orchard	1.07	1.02	0.99	0.94	0.90	0.83
	sort	1.08	1.00	0.96	0.89	0.84	0.80
	heap	5.88	8.07	6.10	5.22	4.14	3.28
	hamerly	**7.40**	**10.08**	8.03	6.61	4.99	3.79
	elkan	2.87	3.00	3.00	2.89	2.83	2.72
$d = 64$	annulus	6.10	6.02	6.65	4.52	3.58	2.66
	adaptive	**6.96**	**7.60**	**11.86**	**10.49**	**10.37**	**7.71**
	orchard	1.01	0.97	0.91	0.94	0.93	0.76
	sort	1.01	0.97	0.92	0.92	0.90	0.86
	heap	4.84	4.70	5.19	3.70	3.23	2.41
	hamerly	6.54	6.43	6.48	4.72	3.86	2.81
	elkan	4.39	4.77	5.30	5.15	5.25	4.82
$d = 128$	annulus	6.02	5.72	4.74	3.80	3.26	2.36
	adaptive	**8.30**	**9.98**	**10.70**	**10.05**	**10.57**	6.97
	orchard	1.00	1.00	0.96	0.97	0.94	0.71
	sort	1.03	1.00	0.97	0.95	0.94	0.94
	heap	4.69	4.32	3.75	3.09	2.85	2.18
	hamerly	6.35	5.91	4.96	3.95	3.33	2.46
	elkan	6.29	6.91	7.46	7.52	4.79	**7.30**
$d = 256$	annulus	4.39	4.05	3.54	3.34	2.43	2.04
	adaptive	**6.59**	**7.86**	8.35	9.50	7.20	5.01
	orchard	1.05	0.99	0.97	0.98	0.98	1.01
	sort	1.02	0.99	0.98	0.97	0.96	1.04
	heap	3.19	3.17	2.80	2.85	2.20	1.88
	hamerly	4.05	4.28	3.70	3.37	2.54	1.86
	elkan	6.07	7.47	**8.81**	**10.69**	**9.85**	**7.93**
$d = 512$	annulus	4.41	3.17	2.24	2.19	1.73	1.40
	adaptive	**7.44**	6.15	4.66	5.07	3.86	2.32
	orchard	1.08	0.99	0.98	1.02	0.96	0.72
	sort	1.08	1.00	0.98	1.01	0.97	0.92
	heap	3.48	2.52	1.79	1.86	1.55	1.41
	hamerly	4.45	3.32	2.32	2.17	1.73	1.47
	elkan	7.24	**7.47**	**6.86**	**8.09**	**7.22**	*

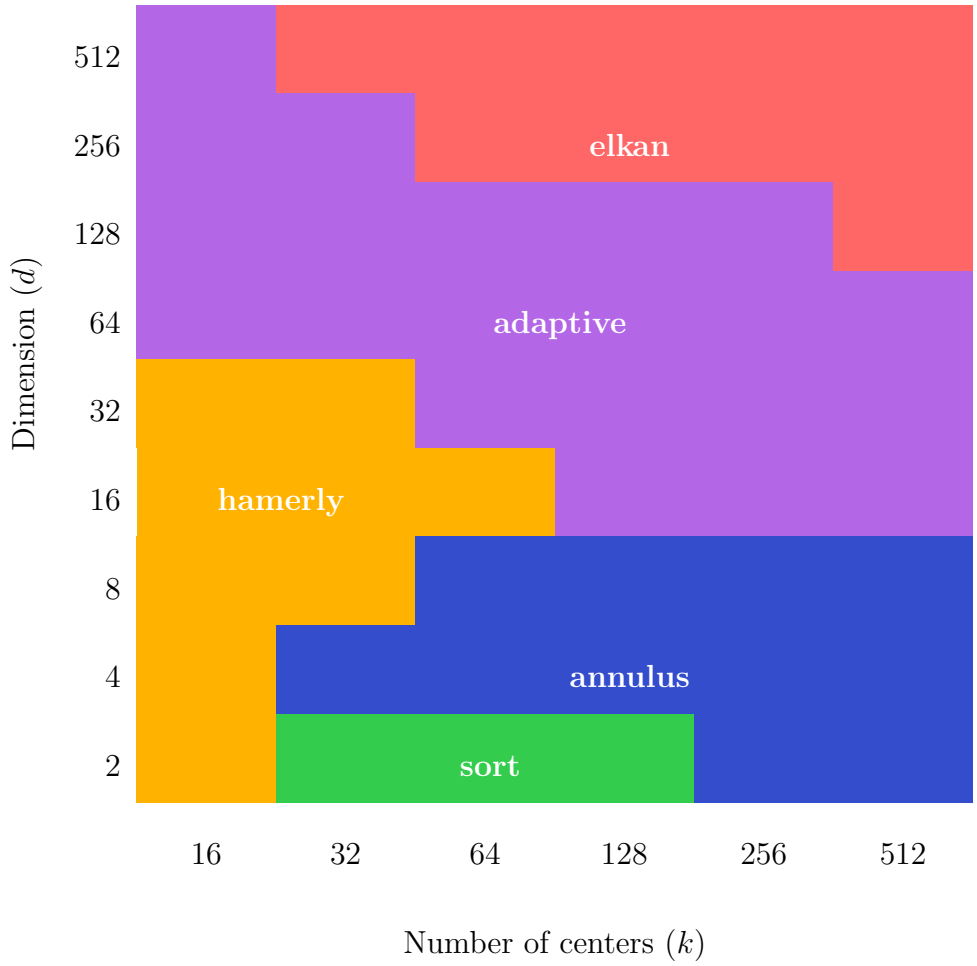

Figure 4.3: Landscape of algorithm superiority on uniform random data for varying dimension d and number of centers of k, with fixed dataset size $n = 400{,}000$. Shaded areas show which algorithm gives the best speedup for particular values of d and k.

4.3.2 Clustered Data

Clustering on uniform random data produces lots of useful data for making comparisons, but there is little practical value in running k-means on non-clustered data. Testing our algorithms on naturally-clustered datasets offers a much clearer impression of their expected performance in a real-world clustering setting. For consistency with other authors, we test on common datasets whenever possible; the characteristics of particular datasets we used are described in Table 3.6 of Chapter 3. Dimension varies from 2 to 74, and cardinality varies from 60,000 to 581,012.

As in our treatment of uniform data, we present results for clustered data in both graphical and tabular format. Table 4.3 gives results for the birch, mnist50, covtype, kddcup98, and kddcup04 datasets. Table 4.4 lists results for the high dimensional mnist784 dataset. Finally, we plot speedup for each algorithm on each dataset, for several fixed k. Figures 4.4, 4.5, 4.6, and 4.7 show results for $k = 16$, 32, 64, and 512, respectively.

Except for very large k, annulus k-means is the clear winner on all tested datasets except the high-dimensional ($d = 784$) dataset mnist784, where Elkan's algorithm dominates. Holding all else constant, larger settings of k tend to increase the effectiveness of SORT-MEANS, which becomes competitive with annulus k-means around $k = 256$, winning on some datasets and losing on others.

4.3.3 Algorithm-Specific Performance

In addition to comparing performance between multiple algorithms, we show how an algorithm performs compared to itself under various settings of k and d. Figures 4.8 through 4.12 show the individual performance characteristics of annulus k-means, adaptive k-means, SORT-MEANS, Hamerly's algorithm, and Elkan's algorithm on uniform random datasets.

Figure 4.8 shows our annulus k-means algorithm has a clear preference for low d and high k, achieving a maximum speedup near 70 for $d = 2$ and $k = 512$. As predicted by our theoretical results, performance decays rapidly with increasing dimension. Performance as a function of k is smoother but depends on dimension. In low dimension, larger k decreases the expected distance to a point's nearest center, so the annular search region shrinks, eliminating more distance calculations. However, in high dimension, this trend reverses: we observe a slight decrease in speedup as k increases. Since the expected distance to a point's closest center varies approximately as $k^{-\frac{1}{d}}$, as discussed in Chapter 3, increasing k causes a negligible change in expected

Table 4.3: Speedup of accelerated algorithms relative to Lloyd's algorithm while clustering five medium-dimensional real-world datasets for various k. Bold values show the best speedup in each experiment.

Dataset	Algorithm	Number of centers (k)					
		16	32	64	128	256	512
birch	annulus	5.56	**10.26**	19.70	20.03	26.63	**38.45**
$n = 100{,}000$	adaptive	2.17	3.28	4.80	3.78	4.06	4.41
$d = 2$	orchard	1.58	1.63	1.24	1.44	1.43	1.12
	sort	5.41	9.39	**20.58**	**25.97**	**32.12**	26.96
	heap	4.29	8.19	10.84	5.67	5.24	4.98
	hamerly	**5.83**	9.69	15.08	8.96	8.31	8.01
	elkan	0.58	0.81	0.92	0.79	0.76	0.78
mnist50	annulus	**17.40**	**17.27**	**21.43**	**31.04**	**21.24**	**16.65**
$n = 60{,}000$	adaptive	14.89	12.42	11.19	15.23	9.86	7.35
$d = 50$	orchard	4.63	5.10	4.40	4.06	3.67	3.37
	sort	5.88	7.72	8.03	7.86	7.32	7.66
	heap	11.76	4.79	4.26	4.58	2.74	2.07
	hamerly	12.06	5.26	4.64	4.97	2.95	2.15
	elkan	4.23	3.70	3.58	3.26	3.02	2.92
covtype	annulus	**16.35**	**23.58**	**24.49**	**38.68**	**32.60**	31.65
$n = 581{,}012$	adaptive	10.61	14.23	15.82	24.47	24.75	23.07
$d = 54$	orchard	3.85	5.13	5.24	6.47	7.44	6.61
	sort	5.03	7.64	11.60	18.96	30.35	**49.73**
	heap	9.44	10.37	7.77	9.78	6.77	5.35
	hamerly	12.89	13.38	9.46	11.02	7.55	5.97
	elkan	4.74	4.57	3.96	3.91	3.87	3.87
kddcup98	annulus	**6.34**	5.18	9.49	11.24	7.52	8.38
$n = 95{,}412$	adaptive	5.53	3.74	6.23	9.67	7.82	7.98
$d = 56$	orchard	3.72	4.53	4.77	5.54	6.19	5.30
	sort	4.72	**6.97**	**9.76**	**13.28**	**17.94**	**19.54**
	heap	3.79	2.33	3.93	4.09	2.44	2.51
	hamerly	5.45	3.32	4.99	4.63	2.26	2.65
	elkan	4.01	3.73	3.64	3.61	3.63	3.36
kddcup04	annulus	**7.80**	**12.74**	**12.92**	**18.16**	**17.65**	**13.82**
$n = 139{,}658$	adaptive	4.58	7.07	9.39	12.58	10.54	9.08
$d = 74$	orchard	2.47	2.70	3.02	3.36	3.66	3.96
	sort	3.08	3.43	4.07	4.66	5.41	6.71
	heap	3.50	4.36	3.66	4.40	3.93	2.93
	hamerly	4.40	5.05	3.93	4.73	4.13	3.02
	elkan	4.65	4.59	4.69	4.66	4.62	4.55

Table 4.4: Speedup of accelerated algorithms relative to Lloyd's algorithm on the high-dimensional mnist784 dataset for various k. Bold values show the best speedup in each experiment.

Dataset	Algorithm	Number of centers (k)					
		16	32	64	128	256	512
mnist784	annulus	4.06	4.75	3.32	2.80	1.94	2.40
$n = 60{,}000$	adaptive	7.23	12.07	9.91	8.98	7.16	7.01
$d = 784$	orchard	1.05	1.06	1.08	1.11	0.84	1.34
	sort	1.03	1.06	1.10	1.12	1.18	1.55
	heap	3.93	4.49	3.08	2.58	2.17	2.26
	hamerly	4.17	4.95	3.36	2.81	2.32	2.21
	elkan	**9.68**	**17.04**	**16.52**	**15.56**	**17.07**	**19.68**

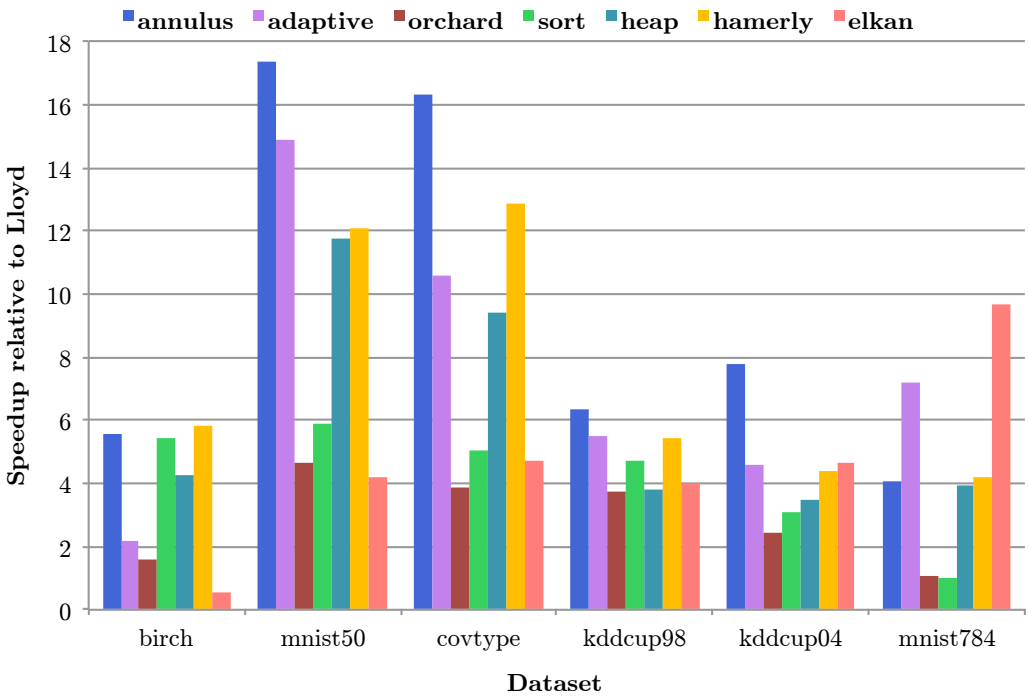

Figure 4.4: Speedup relative to Lloyd on clustered datasets, $k = 16$

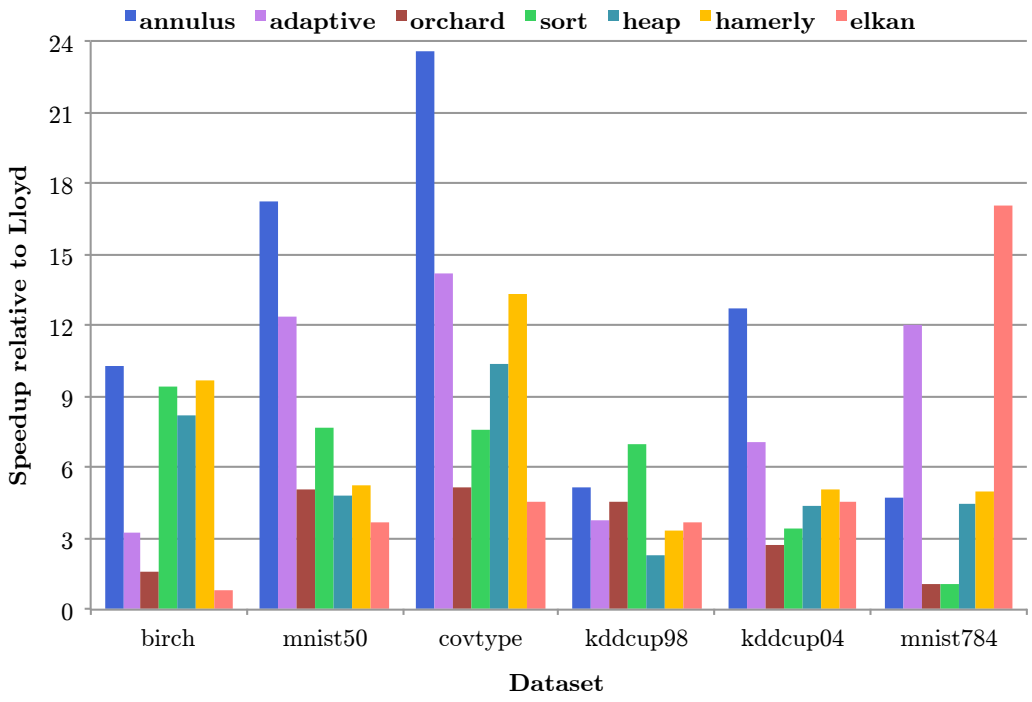

Figure 4.5: Speedup relative to Lloyd on clustered datasets, $k = 32$

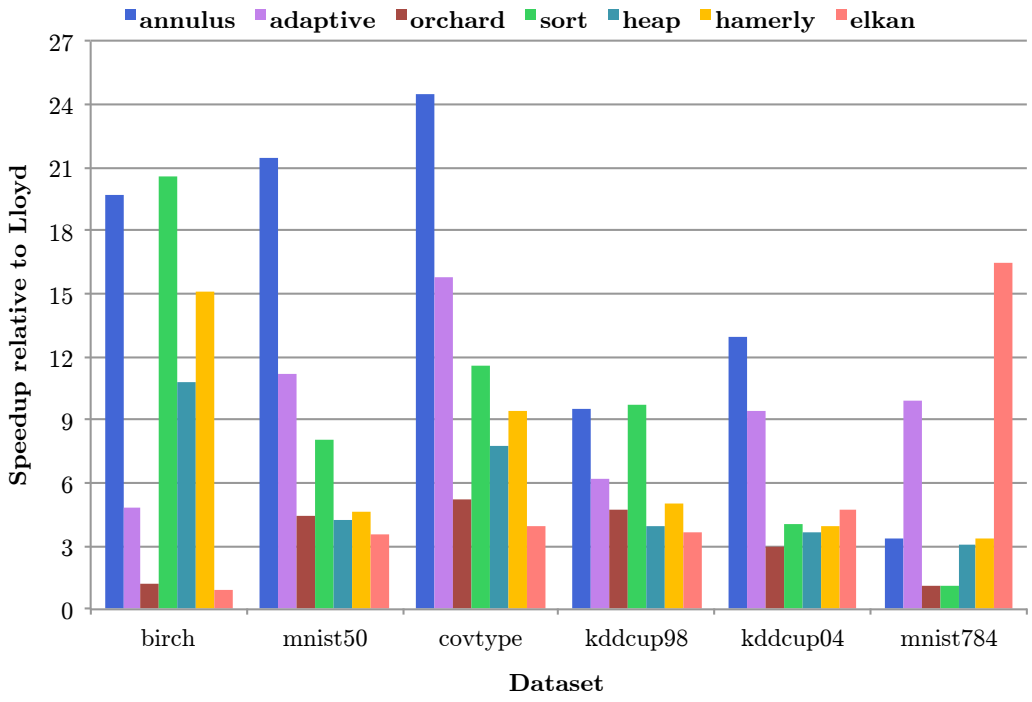

Figure 4.6: Speedup relative to Lloyd on clustered datasets, $k = 64$

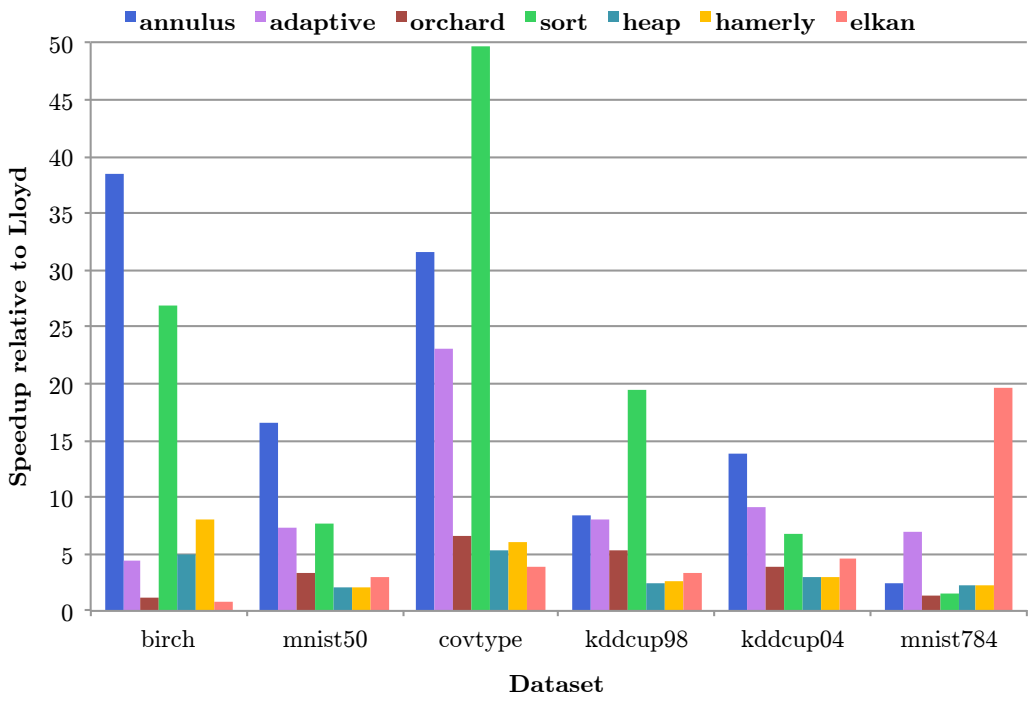

Figure 4.7: Speedup relative to Lloyd on clustered datasets, $k = 512$

distance at $d = 512$. The annulus is ineffective for such a high dimension anyway, so our algorithm wastes time trying to prune centers, an overhead that grows as the number of centers becomes larger.

Figure 4.9 shows the medium-dimensional superiority we expect from our hybrid of Hamerly's low-dimensional algorithm and Elkan's high-dimensional algorithm. Performance generally improves with larger k, although eventually our automatic bounds adjustment fails to select an optimal b. Deviation from the optimum becomes relatively more problematic as k gets large.

Figure 4.10 plots the performance characteristics of Phillips's SORT-MEANS technique, which greatly resembles that of annulus k-means. Overall, SORT-MEANS performs poorly except in very low d and prefers larger k. Speedup degrades quickly with increasing dimension, dropping *below* 1.0 around $d \geq 16$, where Phillips's very conservative inequality fails to discriminate between high-dimensional distances.

Figure 4.11 shows the behavior of Hamerly's algorithm. Like annulus k-means and SORT-MEANS, Hamerly's algorithm prefers low dimension and high k, but never exceeds a speedup of about 20 in our tests, compared to annulus k-means's peak around 70 and SORT-MEANS's peak near 60. In general, the single lower bound on the distance to each point's second-closest center becomes less effective in higher dimension (Hamerly 2010).

Figure 4.12 describes the increasingly good performance of Elkan's algorithm in higher-dimensional spaces. In low dimension, keeping $\Theta(nk)$ distance bounds dwarfs the $\Theta(nd)$ size of the dataset itself, a totally unprofitable overhead. However, in large dimension, two factors contribute to Elkan's success: first, the relative cost of the $\Theta(nk)$ distance bounds diminishes; second, the expense of distance calculations becomes increasingly dominant. Having many distance bounds allows Elkan to avoid many of these now-expensive distance calculations, resulting in impressive high-dimensional speedup. Compared to other methods, speedup in Elkan's algorithm is relatively insensitive to the number of centers k on uniform data.

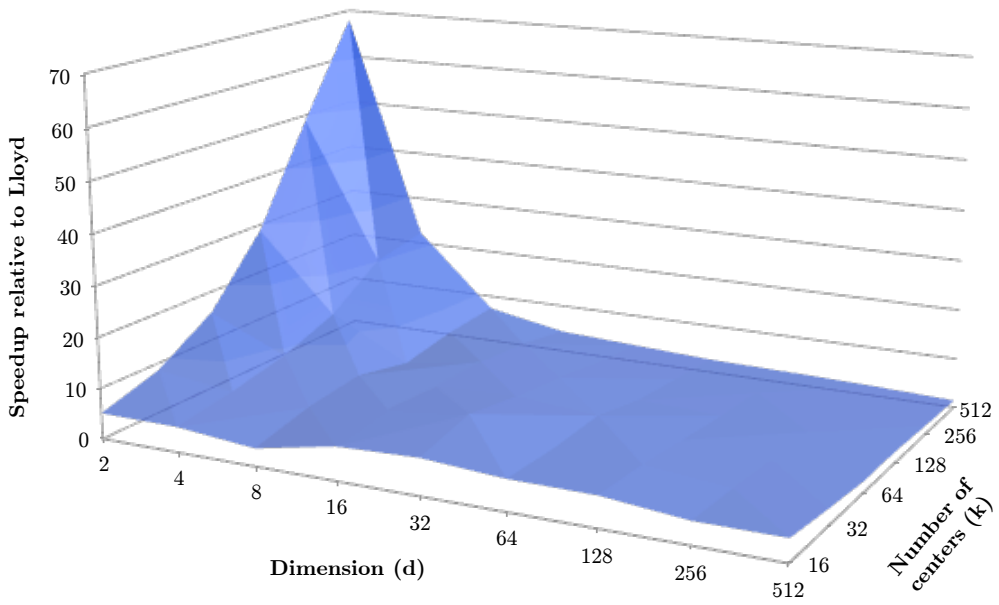

Figure 4.8: Speedup of annulus k-means relative to Lloyd for various k and d

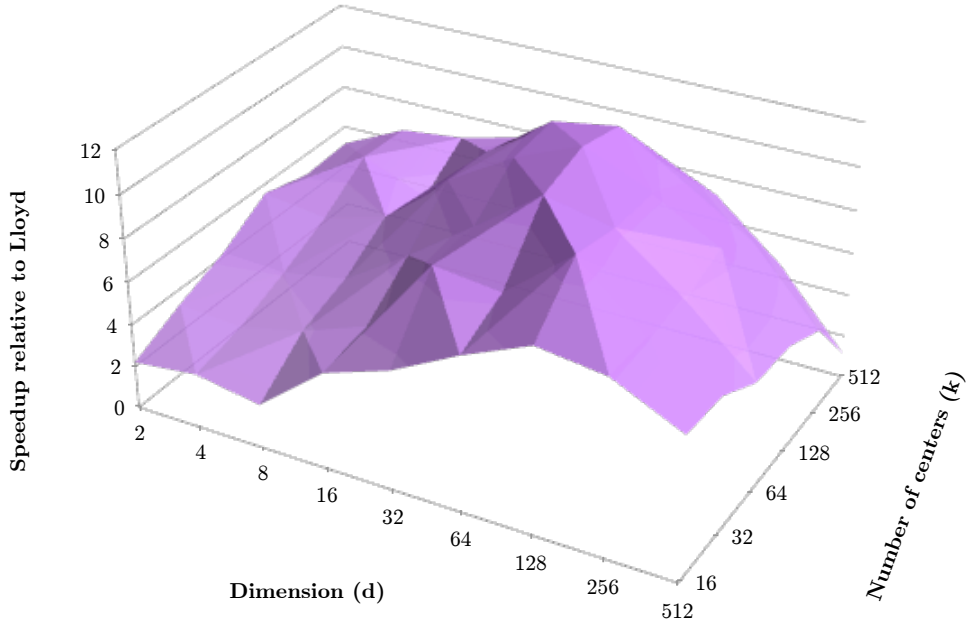

Figure 4.9: Speedup of adaptive k-means relative to Lloyd for various k and d

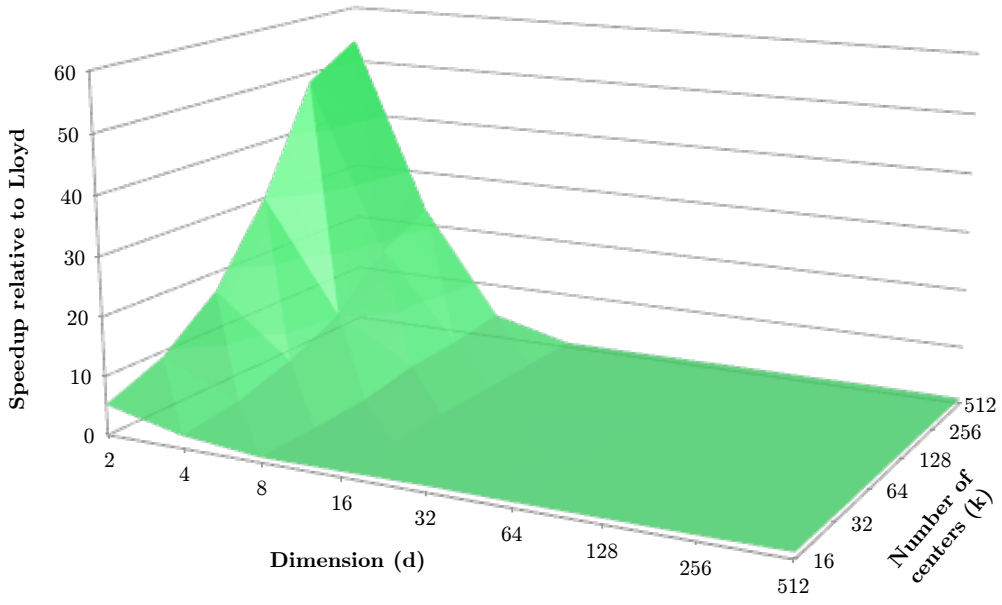

Figure 4.10: Speedup of SORT-MEANS relative to Lloyd for various k and d

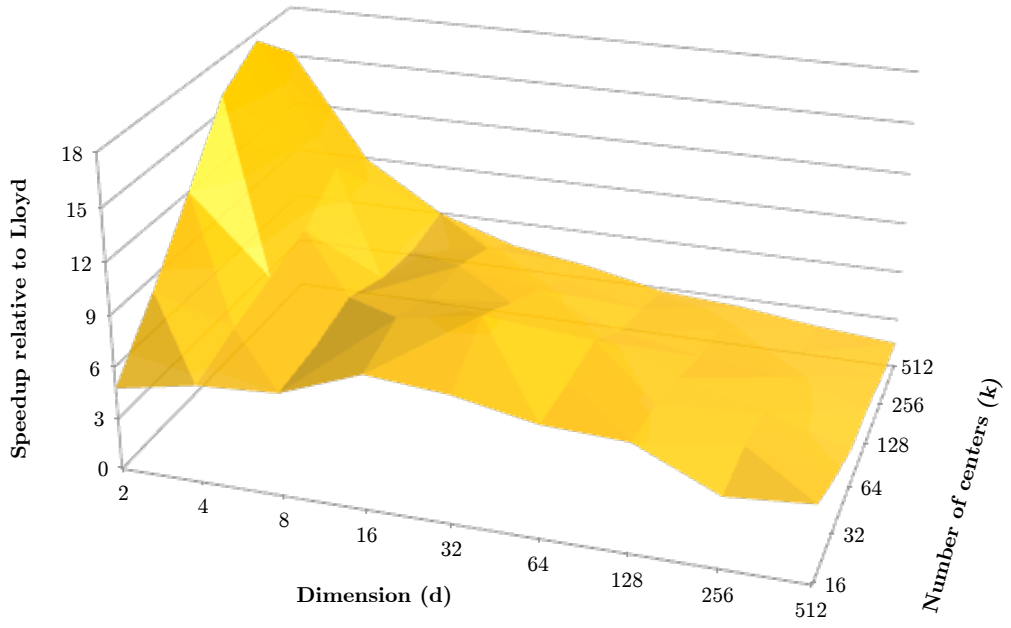

Figure 4.11: Speedup of Hamerly's algorithm relative to Lloyd for various k and d

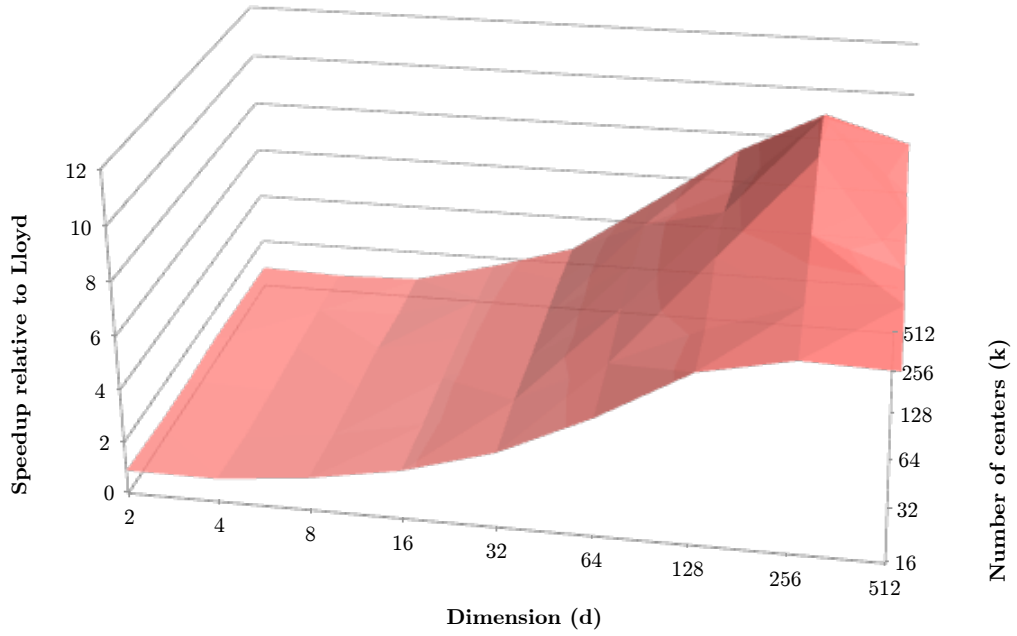

Figure 4.12: Speedup of Elkan's algorithm relative to Lloyd for various k and d

4.4 Separability

In the paper on his anchors hierarchy, Andrew Moore writes, "If there is no underlying structure in the data (e.g. if it is uniformly distributed) there will be little or no acceleration in high dimensions no matter what we do. This gloomy view [. . .] means that we can only accelerate datasets that have interesting internal structure" (Moore 2000). Rephrased, a dataset with more structure will see more acceleration. One way to increase structure is by random projection, since projection makes clusters more Gaussian (Dasgupta 2000).

The success of random projection means that algorithms like annulus k-means, which thrives in lower dimension, may function well despite high dimensionality by first projecting data into a smaller space. As observed in (Hamerly 2010), random projection is both theoretically and practically beneficial.

We illustrate the validity of these theoretical observations with a small experiment in $d = 3$ dimensions. Starting from a $4 \times 4 \times 4$ unit lattice of points in 3-space, we generate different random datasets by creating Gaussian clusters at the lattice points of varying standard deviation σ. On each axis, clusters are separated by a unit distance of 1.0, so setting $\sigma = 0.25$ places the midpoint between clusters within two standard deviations of one another, creating noticeable overlap.

For several values of σ, we plot the speedup of annulus k-means, Sort-means, and Hamerly's algorithm in Figure 4.13, using the known value $k = 64$. Performance declines sharply as σ increases, then eventually stabilizes around $\sigma = 0.5$. Effectively, we have created a smooth transition from well-separated data to poorly-separated data. Our real-world datasets like covtype and mnist50 behave like datasets here with low deviation, and our uniform random datasets resemble the datasets here with high deviation. Notably, Sort-means tends to take slightly greater advantage of structure in this very low-dimensional dataset than annulus k-means and Hamerly's algorithm, which are however more resilient to overlapping clusters than Sort-means.

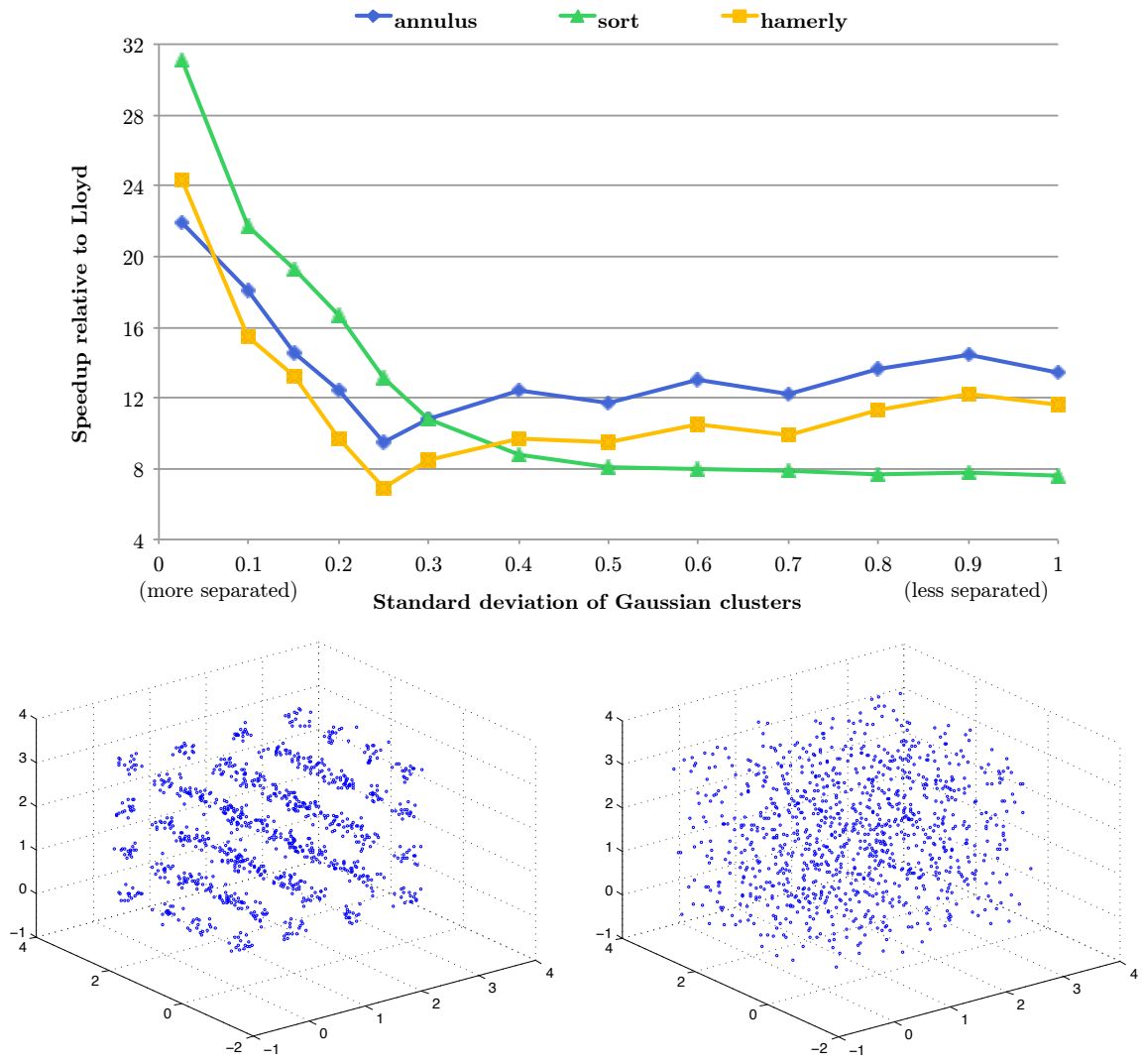

Figure 4.13: The top graph shows speedup relative to Lloyd on a regular three-dimensional unit lattice of $k = 4 \times 4 \times 4 = 64$ Gaussian clusters of varying standard deviation, with fixed $n = 400{,}000$. The bottom left graph illustrates the lattice with low deviation (more separation) and the bottom right graph shows a lattice with high deviation (less separation).

4.5 Memory Use

All of the accelerated algorithms in this paper gain speed in exchange for additional memory beyond that used in Lloyd's algorithm, which is dominated by the size n of the dataset and k centers $\Theta(nd+kd)$. For example, the dominant additional expense in Elkan's algorithm is $\Theta(nk)$ lower bounds. For $k = d = 256$, this means that Elkan's algorithm uses roughly twice the memory of Lloyd's algorithm. Adaptive k-means keeps a variable number b of lower bounds, reducing the additional memory complexity to $\Theta(nb)$. For the same case $k = d = 256$, adaptive k-means keeps about 1.3 times the memory of Lloyd's algorithm. Table 4.5 shows the asymptotic requirements for each algorithm.

Table 4.5: Asymptotic memory requirements for each algorithm in addition to the baseline $\Theta(nd + kd)$ bytes used by Lloyd to store the dataset and centers.

Algorithm(s)	Additional Memory	Total Memory
orchard, sort	$\Theta(k^2)$	$\Theta(nd + kd + k^2)$
annulus, hamerly, heap	$\Theta(n + k)$	$\Theta(nd + kd)$
adaptive	$\Theta(nb)$	$\Theta(nd + kd + nb)$
elkan	$\Theta(nk)$	$\Theta(nd + kd + nk)$

Hamerly's algorithm keeps a single lower bound, reducing the additional memory complexity to $\Theta(n)$. Annulus k-means additionally stores $\Theta(n+k)$ pre-computed norms, which is a minor increase in overall memory overhead. The memory overhead of Sort-means and Orchard's method depends only on k, where $k \ll n$ naturally.

Depending on a user's specific application needs, it may be necessary to sacrifice the runtime performance of Elkan's algorithm or adaptive k-means for an algorithm with better memory characteristics. In general, the good runtime performance and small memory overhead of annulus k-means makes it a powerful replacement for Lloyd's algorithm in a wide variety of circumstances.

Table 4.6: Memory use relative to Lloyd's algorithm while clustering uniform random data for various k and dimension $2 \leq d \leq 16$, with fixed dataset size $n = 400{,}000$.

Dimension	Algorithm	Number of centers (k)					
		16	32	64	128	256	512
$d = 2$	annulus	2.2	2.2	2.2	2.2	2.2	2.2
	adaptive	4.3	6.2	9.9	17.5	32.5	62.7
	orchard	3.9	1.0	1.0	1.0	1.0	1.2
	sort	3.9	1.0	1.0	1.0	1.1	1.3
	heap	2.2	2.3	2.4	2.4	2.4	2.5
	hamerly	1.8	1.8	1.8	1.8	1.8	1.8
	elkan	7.4	13.5	25.5	49.7	97.9	194.6
$d = 4$	annulus	1.7	1.7	1.7	1.7	1.7	1.7
	adaptive	2.9	3.9	6.1	10.4	19.0	36.2
	orchard	2.7	1.0	1.0	1.0	1.0	1.1
	sort	2.7	1.0	1.0	1.0	1.0	1.2
	heap	1.7	1.8	1.8	1.8	1.8	1.8
	hamerly	1.4	1.4	1.4	1.4	1.4	1.4
	elkan	4.7	8.1	15.0	28.8	56.3	111.4
$d = 8$	annulus	1.4	1.4	1.4	1.4	1.4	1.4
	adaptive	2.0	2.6	3.7	6.1	10.7	19.9
	orchard	1.9	1.0	1.0	1.0	1.0	1.1
	sort	1.9	1.0	1.0	1.0	1.0	1.1
	heap	1.4	1.4	1.4	1.4	1.5	1.5
	hamerly	1.2	1.2	1.2	1.2	1.2	1.2
	elkan	3.0	4.8	8.5	15.9	30.7	60.4
$d = 16$	annulus	1.2	1.2	1.2	1.2	1.2	1.2
	adaptive	1.5	1.8	2.4	3.6	6.0	10.8
	orchard	1.5	1.0	1.0	1.0	1.0	1.0
	sort	1.5	1.0	1.0	1.0	1.0	1.1
	heap	1.2	1.2	1.2	1.2	1.2	1.2
	hamerly	1.1	1.1	1.1	1.1	1.1	1.1
	elkan	2.0	3.0	4.9	8.8	16.5	31.9

Table 4.7: Memory use relative to Lloyd's algorithm while clustering uniform random data for various k and dimension $32 \leq d \leq 512$, with fixed $n = 400{,}000$. *Elkan's memory requirement exceeds 3GB per-process limit on our test machine.

Dimension	Algorithm	Number of centers (k)					
		16	32	64	128	256	512
$d = 32$	annulus	1.1	1.1	1.1	1.1	1.1	1.1
	adaptive	1.3	1.4	1.7	2.3	3.6	6.0
	orchard	1.2	1.0	1.0	1.0	1.0	1.0
	sort	1.2	1.0	1.0	1.0	1.0	1.0
	heap	1.1	1.1	1.1	1.1	1.1	1.1
	hamerly	1.1	1.1	1.1	1.1	1.1	1.1
	elkan	1.5	2.0	3.0	5.0	8.9	16.7
$d = 64$	annulus	1.0	1.0	1.1	1.1	1.1	1.1
	adaptive	1.1	1.2	1.4	1.7	2.3	3.5
	orchard	1.1	1.0	1.0	1.0	1.0	1.0
	sort	1.1	1.0	1.0	1.0	1.0	1.0
	heap	1.1	1.1	1.1	1.1	1.1	1.1
	hamerly	1.0	1.0	1.0	1.0	1.0	1.0
	elkan	1.3	1.5	2.0	3.0	5.0	8.9
$d = 128$	annulus	1.0	1.0	1.0	1.0	1.0	1.0
	adaptive	1.1	1.1	1.2	1.3	1.7	2.3
	orchard	1.1	1.0	1.0	1.0	1.0	1.0
	sort	1.1	1.0	1.0	1.0	1.0	1.0
	heap	1.0	1.0	1.0	1.0	1.0	1.0
	hamerly	1.0	1.0	1.0	1.0	1.0	1.0
	elkan	1.1	1.3	1.5	2.0	3.0	5.0
$d = 256$	annulus	1.0	1.0	1.0	1.0	1.0	1.0
	adaptive	1.0	1.1	1.1	1.2	1.3	1.6
	orchard	1.0	1.0	1.0	1.0	1.0	1.0
	sort	1.0	1.0	1.0	1.0	1.0	1.0
	heap	1.0	1.0	1.0	1.0	1.0	1.0
	hamerly	1.0	1.0	1.0	1.0	1.0	1.0
	elkan	1.1	1.1	1.3	1.5	2.0	3.0
$d = 512$	annulus	1.0	1.0	1.0	1.0	1.0	1.0
	adaptive	1.0	1.0	1.0	1.1	1.2	1.3
	orchard	1.0	1.0	1.0	1.0	1.0	1.0
	sort	1.0	1.0	1.0	1.0	1.0	1.0
	heap	1.0	1.0	1.0	1.0	1.0	1.0
	hamerly	1.0	1.0	1.0	1.0	1.0	1.0
	elkan	1.0	1.1	1.1	1.3	1.5	*

4.6 Distance Calculations

Finally, we compare the raw number of distance calculations computed by a family of four related algorithms: annulus k-means, adaptive k-means, Hamerly's algorithm, and Elkan's algorithm. Since Elkan's algorithm keeps distance bounds for all centers, we expect for Elkan to avoid the largest number of distance calculations (in exchange for larger overhead). Hamerly's algorithm is simple and fast, but avoids fewer distance calculations, which makes it less suitable for high-dimensional spaces where distance calculations are critically expensive.

Adaptive k-means hybridizes Elkan's and Hamerly's approaches, so it should avoid more distance calculations than Hamerly but fewer than Elkan's. Annulus k-means is similar to Hamerly's algorithm, but the annular structure allows it to avoid more distance calculations, so we again expect a number between Hamerly's and Elkan's. Nota bene: we do not show results for the heap-based variant of Hamerly's algorithm, because it performs the very same computations as Hamerly, but by a different mechanism.

Figure 4.14 shows the cumulative number of distance calculations computed by each algorithm as k-means iterates on the mnist50 dataset. Figure 4.15 shows the cumulative number of distance calculations computed by each algorithm as k-means iterates on a uniform random dataset with $d = 2$ and $n = 400{,}000$.

To further compare annulus k-means with Hamerly's algorithm, Figures 4.16 and 4.17 show the number of distance calculations computed in each iteration relative to the number computed by Lloyd. The battle is decided in early iterations, where centers assignments change more dramatically: here, the annulus clearly avoids more computations than Hamerly's algorithm alone.

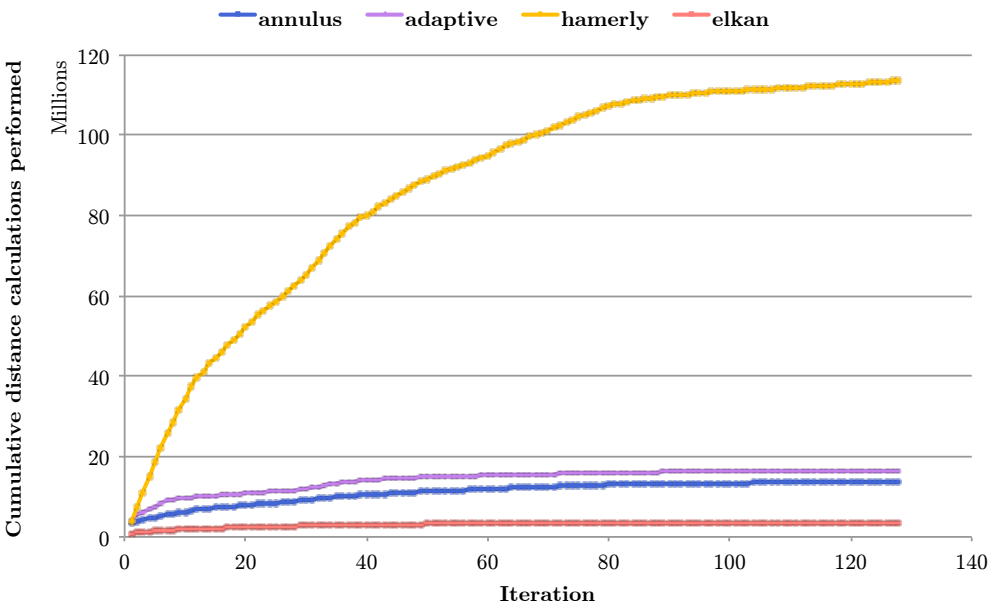

Figure 4.14: Cumulative number of distance calculations performed over time on the mnist50 dataset with $k = 64$. This clustering took 128 iterations.

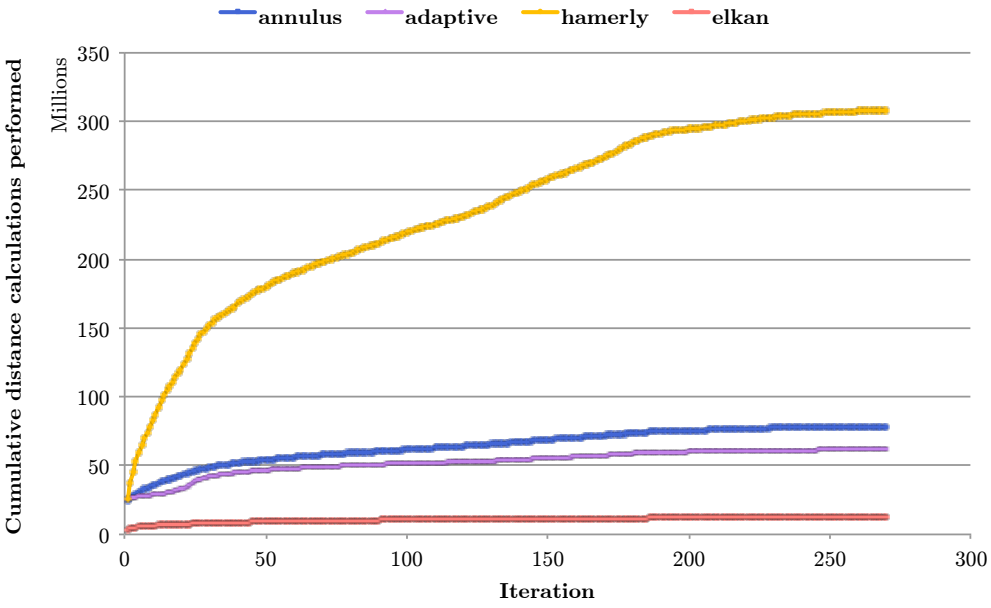

Figure 4.15: Cumulative number of distance calculations performed over time on a uniform random dataset of $n = 400{,}000$ points with $d = 2$ and $k = 64$. This clustering took 270 iterations.

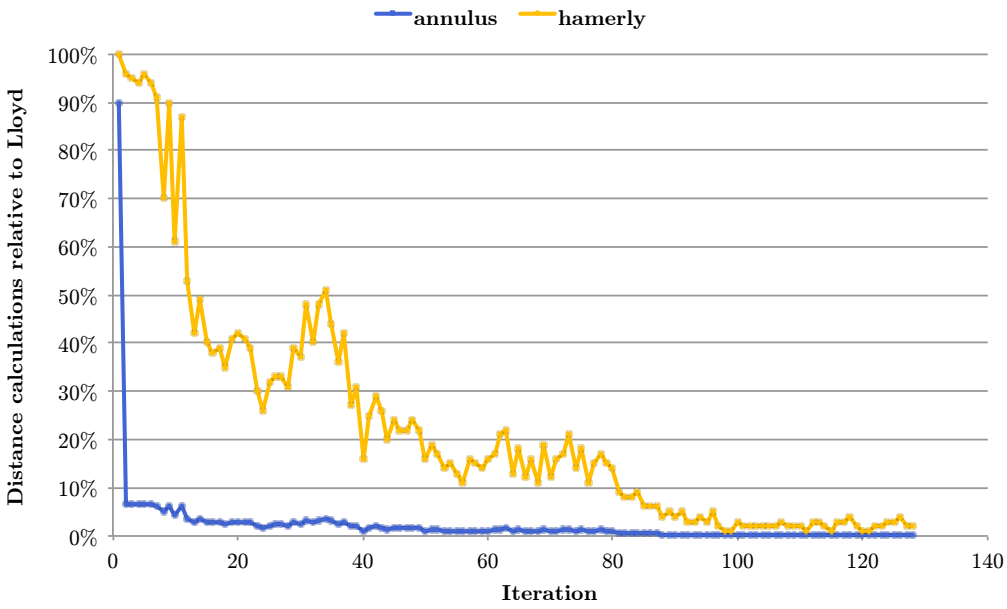

Figure 4.16: Distance calculations relative to Lloyd on the 50-dimensional, naturally-clustered mnist50 dataset with $k = 64$. This clustering took 128 iterations.

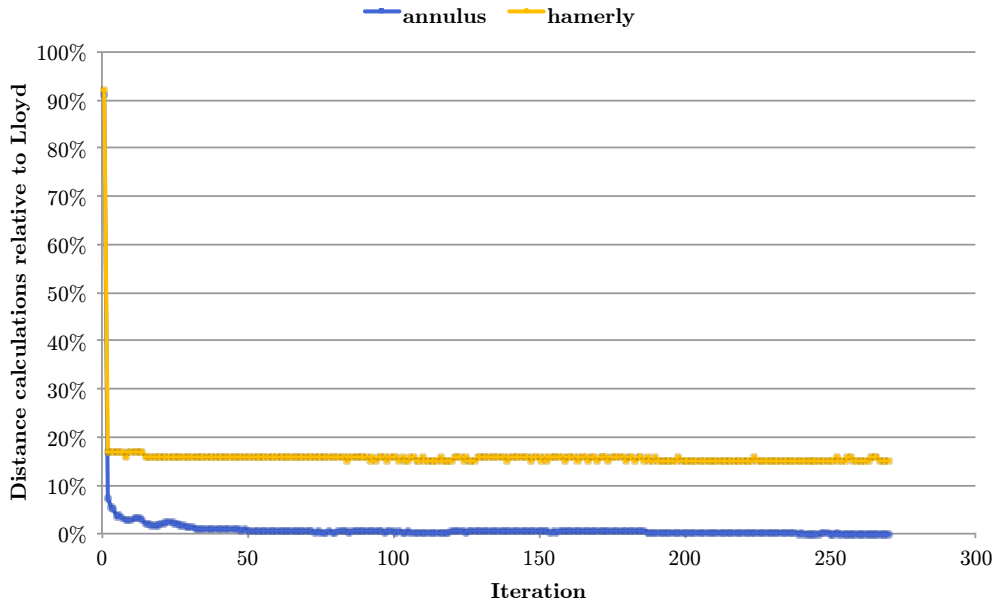

Figure 4.17: Distance calculations relative to Lloyd on a uniform random dataset of $n = 400{,}000$ points with $d = 2$ and $k = 64$. This clustering took 270 iterations.

CHAPTER FIVE

Conclusion

5.1 Summary of Research

This thesis introduces three accelerated k-means algorithms: heap k-means, adaptive k-means, and annulus k-means. While heap k-means tends to be less effective than other methods, adaptive k-means and annulus k-means are competitive with or much faster than other accelerated algorithms, except in very high dimension, where Elkan's algorithm remains dominant (Elkan 2003).

First, we redesign Hamerly's algorithm (Hamerly 2010) to use k min-heaps to avoid necessarily looping over all n data points when testing upper and lower bounds, but the computational overhead of the required heap mechanisms outweighs any reduction in the number of required distance bound checks.

Second, we present our adaptive k-means algorithm (Drake and Hamerly 2012), which keeps an adaptive number of distance bounds to avoid redundant distance calculations, combining the low-dimensional strength of Hamerly's algorithm and the high-dimensional strength Elkan's algorithm. Adaptive k-means shows superior performance in medium dimension (approximately $20 \leq d \leq 200$) in our empirical tests on uniform random data, and beats Hamerly's and Elkan's algorithms on all clustered datasets except the 784-dimensional dataset mnist784.

Finally, we introduce annulus k-means: a variant of Hamerly's algorithm that improves the search for a point's nearest center by cheaply constructing an annular search region derived from the triangle inequality. Any center outside the annular region is provably not the point's closest center, giving dramatic speedups in low dimension, especially for larger k. In high dimension, the annulus becomes less effective, but the algorithm still outperforms competitors on each clustered dataset we

tested except mnist784. Annulus k-means has good runtime performance and small memory overhead, making it a powerful replacement for Lloyd's algorithm.

5.2 Limitations and Extensions

We test a wide variety of accelerated algorithms against Lloyd's standard method on uniform random data and on naturally-clustered data. Depending on the dimension d and number of clusters k used in experiments on uniform data, different algorithms have different regions of superiority. In high dimension, Elkan's algorithm is the clear winner. In medium dimension, adaptive k-means is more efficient than Elkan, and in low dimension, annulus k-means, Hamerly's algorithm, and Sort-means perform well. On clustered data, annulus k-means outperforms competitors, except for very high k, where the performance of Sort-means tends to match that of annulus k-means.

Attempts to combine the strengths of annular search with adaptive k-means or Elkan's algorithm proved unprofitable; in the high-dimensional setting where these latter algorithms excel the annulus is ineffective and merely creates a small overhead. Similarly, hybrids with Sort-means also resulted in lackluster performance. Ideally, we would like an algorithm performing well under all conditions: low and high dimension, clustered and uniform, many and few centers. Meanwhile, the algorithms we have developed and the results of our experiments guide current practical k-means use, showing how and when to replace Lloyd's algorithm with equivalent but more efficient methods.

BIBLIOGRAPHY

Agarwal, P. K., S. Har-peled, and K. R. Varadarajan (2005). Geometric approximation via coresets. In *Combinatorial and Computational Geometry, MSRI*, pp. 1–30. University Press.

Al-Zoubi, M., A. Hudaib, A. Huneiti, and B. Hammo (2008). New efficient strategy to accelerate k-means clustering algorithm. *American Journal of Applied Sciences 5*, 1247–1250.

Arthur, D., B. Manthey, and H. Roeglin (2009). k-means has polynomial smoothed complexity. In *50th Symposium on Foundations of Computer Science*.

Arthur, D. and S. Vassilvitskii (2006). How slow is the k-means method? In *22nd Annual Symposium on Computational Geometry*.

Arthur, D. and S. Vassilvitskii (2007). kmeans++: The advantages of careful seeding. In *ACM-SIAM Symposium on Discrete Algorithms*, pp. 1027–1035.

Bahmani, B., B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii (2012). Scalable k-means++. *Proceedings of the VLDB Endowment 5*(7), 622–633.

Bottou, L. and Y. Bengio (1995). Convergence properties of the k-means algorithms. *Advances in Neural Information Processing Systems 7*(7), 585–592.

Celebi, M. E. (2009). Effective initialization of k-means for color quantization. In *Proceedings of the 16th IEEE international conference on Image processing*, Piscataway, NJ, pp. 1629–1632. IEEE Press.

Celebi, M. E. (2011, March). Improving the performance of k-means for color quantization. *Image Vision Comput. 29*(4), 260–271.

Celebi, M. E., H. A. Kingravi, and P. A. Vela (2013, January). A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Syst. Appl. 40*(1), 200–210.

Dasgupta, S. (2000). Experiments with random projection. In *UAI*, pp. 143–151.

Dhillon, I. S., Y. Guan, and B. Kulis (2004). Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, New York, NY, USA, pp. 551–556. ACM.

Drake, J. and G. Hamerly (2012). Accelerated k-means with adaptive distance bounds. In *5th NIPS Workshop on Optimization for Machine Learning*.

Elkan, C. (2003). Using the triangle inequality to accelerate k-means. In *ICML*, pp. 147–153.

Forgy, E. (1965). Cluster analysis of multivariate data: efficiency versus interpretability of classifications. In *Biometric Society Meeting*, Riverside, CA.

Frahling, G. and C. Sohler (2006). A fast k-means implementation using coresets. In *22nd Annual Symposium on Computational Geometry*, pp. 135–143.

Furini, M., F. Geraci, M. Montangero, and M. Pellegrini (2008). On using clustering algorithms to produce video abstracts for the web scenario. In *Consumer Communications and Networking Conference, 2008. 5th IEEE*, pp. 1112–1116.

Hamerly, G. (2010). Making k-means even faster. In *2010 SIAM international conference on data mining*.

Har-Peled, S. and A. Kushal (2005). Smaller coresets for k-median and k-means clustering. In *SOCG 2005*, pp. 126–134.

Har-Peled, S. and B. Sadri (2005). How fast is the k-means method? In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, pp. 877–885. Society for Industrial and Applied Mathematics.

Hartigan, J. A. and M. A. Wong (1979). Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics 28*(1), 100–108.

Hochbaum, D. S. and D. B. Shmoys (1985). A best possible heuristic for the k-center problem. *MATHEMATICS OF OPERATIONS RESEARCH 10*(2), 180–184.

Huang, C.-M., Q. Bi, G. Stiles, and R. Harris (1992). Fast full search equivalent encoding algorithms for image compression using vector quantization. *IEEE Transactions on Image Processing 1*(3), 413–416.

Inaba, M., N. Katoh, and H. Imai (1994). Applications of weighted voronoi diagrams and randomization to variance-based k-clustering. In *Proceedings of the tenth annual symposium on Computational Geometry*, New York, pp. 332–339. ACM.

Jain, A. K. (2010, June). Data clustering: 50 years beyond k-means. *Pattern Recogn. Lett. 31*(8), 651–666.

Kanungo, T., D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu (1999). Computing nearest neighbors for moving points and applications to clustering.

Kanungo, T., D. M. Mount, N. S. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu (2000). The analysis of a simple k-means clustering algorithm. In *Proceedings of the sixteenth annual symposium on Computational geometry*, SCG '00, New York, pp. 100–109. ACM.

Kanungo, T., D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu (2002, July). An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 881–892.

Lloyd, S. (1982). Least squares quantization in pcm. In *IEEE Transactions Information Theory*, 28, pp. 129–137.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, 1, Berkeley, pp. 281–297. University of California Press.

Moore, A. (1999, April). Very fast em-based mixture model clustering using multiresolution kd-trees. In M. Kearns and D. Cohn (Eds.), *Advances in Neural Information Processing Systems*, pp. 543–549. Morgan Kaufman.

Moore, A. (2000). The anchors hierarchy: Using the triangle inequality to survive high-dimensional data. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, pp. 397–405. AAAI Press.

Moore, A. (2001). k-means and hierarchical clustering. `http://www.cs.cmu.edu/~awm/tutorials/kmeans.html`.

Orchard, M. (1991). A fast nearest-neighbor search algorithm. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pp. 2297–2300 vol.4.

Pelleg, D. and A. Moore (1999). Accelerating exact k-means algorithms with geometric reasoning. In *ACM SIGKDD fifth international conference on knowledge discovery and data mining*, pp. 277–281.

Phillips, S. J. (2002). Acceleration of k-means and related clustering algorithms. In D. Mount and C. Stein (Eds.), *Algorithm Engineering and Experiments*, Volume 2409 of *Lecture Notes in Computer Science*, pp. 61–62. Springer Berlin-Heidelberg.

Sculley, D. (2010). Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, WWW '10, New York, NY, USA, pp. 1177–1178. ACM.

Su, T. and J. G. Dy (2007, December). In search of deterministic methods for initializing k-means and gaussian mixture clustering. *Intell. Data Anal. 11*(4), 319–338.

Telgarsky, M. and A. Vattani (2010). Hartigans method: k-means clustering without voronoi. *Journal of Machine Learning Research - Proceedings Track 9*, 820–827.

Turnbull, D. and C. Elkan (2005, April). Fast recognition of musical genres using rbf networks. *IEEE Trans. on Knowl. and Data Eng. 17*, 580–584.

Vattani, A. (2009). k-means requires exponentially many iterations even in the plane. In *Proceedings of the 25th Annual Symposium on Computational Geometry*, New York, pp. 324–332. ACM.

Wu, X., V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. F. M. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg (2008). Top 10 algorithms in data mining. *Knowl. Inf. Syst. 14*(1), 1–37.

Zhang, J., G. Wu, X. Hu, S. Li, and S. Hao (2011). A parallel k-means clustering algorithm with mpi. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pp. 60–64.

Zhao, W., H. Ma, and Q. He (2009). Parallel k-means clustering based on mapreduce. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, Berlin, Heidelberg, pp. 674–679. Springer-Verlag.