# Using the Triangle Inequality to Accelerate $k$-Means

Nipun Gupta

May 3, 2015

**Abstract**

The $k$-means algorithm is the most widely used algorithm for discovering clusters in the data. Here, we first show Elkan's algorithm [1], which accelerates $k$-means by avoiding redundant distance calculations. This is done by using triangle inequality to keep track of lower and upper bounds for distances between points and centers. Further, we discuss another algorithm by Hamerly [2], which builds on Elkan's algorithm, and uses a novel lower bound for point-center distances to give a much faster algorithm for datasets of low and medium dimensions (e.g. up to 50 dimensions).

## 1 Lloyd's algorithm

This is the most popular algorithm used for $k$-means clustering. The two primary steps in this iterative algorithm are:

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

The number of distance calculations in this algorithm is $nke$ where $n$ is the number of data points, $k$ is the number of clusters to be found, and $e$ is the number of iterations required. Empirically, $e$ grows sublinearly with $k$, $n$, and the dimensionality $d$ of the data. Elkan's algorithm reduces the number of distance calculations in practice closer to $n$.

**Why is Lloyd's algorithm inefficient?**
Most of the distance calculations in the algorithm are redundant. It is not necessary to calculate exact distance between the point and the center in order to say whether the point should be assigned to that center or not. This is where the bounds on the distances come into play. Also, towards the end of algorithm during convergence, there are very small changes in the center-assignments of the data points. This fact is not exploited in the standard algorithm as no information is passed in the subsequent iterations. Elkan's optimized algorithm recognizes these short-comings and improves on them. For accelerated $k$-means algorithm to be usable wherever the standard algorithm is used, it needs to satisfy some properties.

**Properties that accelerated $k$-means algorithm should satisfy**
1. It should be able to start with any initial centers, so that all existing initialization methods can be used.
2. It should be able to use any black-box distance metric, so it should not rely for example on optimizations specific to Euclidean distance.
3. Given the same initial centers, it should always produce the same final centers as the standard algorithm. In fact, Elkan's algorithm produces the same set of center locations as the standard $k$-means algorithm after each iteration.

## 2 Applying the triangle inequality

The only "black box" property that all distance metrics $d$ possess is the triangle inequality: for any three points $x$, $y$, and $z$, $d(x,z) \leq d(x,y) + d(y,z)$. However, the triangle inequality gives only upper bounds. So, in order to get the lower bounds, we use the following two lemmas.

**Lemma 1:** For any three points $x$, $b$, and $c$, if $d(b,c) \geq 2d(x,b)$ then $d(x,c) \geq d(x,b)$.

**Proof:**

$$\begin{aligned} \text{We know that, } & d(b,c) \leq d(b,x) + d(x,c) \\ & \Rightarrow d(b,c) - d(b,x) \leq d(x,c) \\ & \Rightarrow 2d(x,b) - d(x,b) \leq d(b,c) - d(b,x) \leq d(x,c) \\ & \Rightarrow d(x,b) \leq d(x,c) \end{aligned} \tag{1}$$

**Lemma 2:** For any three points $x$, $b$, and $c$, $d(x,c) \geq \max\{0, d(x,b) - d(x,c)\}$

**Proof:**

$$\begin{aligned} \text{We know that, } & d(x,b) \leq d(x,x) + d(b,c) \\ & \Rightarrow d(x,c) \geq d(x,b) - d(b,c) \\ \text{Also, } & d(x,c) \geq 0 \end{aligned} \tag{2}$$

Now, we will describe how to use Lemma-1 and Lemma-2 in order to avoid distance calculations and to find upper and lower bounds.

**1. How to use Lemma-1**
Let $x$ be the data point, $c$ be the center to which $x$ is currently assigned, and $c'$ be any other center.

(a) If $d(x,c) \leq \frac{1}{2}d(c,c') \Rightarrow d(x,c) \leq d(x,c')$. So, we don't have to calculate $d(x,c')$.

(b) If $d(x,c)$ is not known exactly, but we know an upper bound $d(x,c) \leq u$. If $u \leq \frac{1}{2}d(c,c') \Rightarrow d(x,c) \leq d(x,c')$. Again, no need to calculate $d(x,c')$.

(c) if $u \leq \frac{1}{2}\min_{c' \neq c}\{d(c,c')\} \Rightarrow x$ remains assigned to $c$ and there is no need for any distance calculation of $x$.

**2. How to use Lemma-2**
Let $x$ be the data point, $b$ be any $j^{th}$ center, and $b'$ be the previous version of $j^{th}$ center.

(a) Let $l'$ be the lower bound of the previous iteration such that $d(x,b') \geq l'$. Then the lower bound for the current iteration, $l$, can be inferred as $d(x,b) \geq \max\{0, d(x,b') - d(b,b')\} \geq \max\{0, l' - d(b,b')\} = l$.

(b) Now we show how this lower bound is used. Let $x$ be assigned to center $c$ and $c'$ be any other center. Let $d(x,c) \leq u(x)$ and $d(x,c') \geq l(x,c')$, where $u(x)$ is the upper bound on the distance between $x$ and the center to which it is assigned, and $l(x,c')$ is the lower bound on the distance between point $x$ and center $c'$. If $u(x) \leq l(x,c') \Rightarrow d(x,c) \leq u(x) \leq l(x,c') \leq d(x,c')$. So, we need to calculate neither $d(x,c)$ nor $d(x,c')$.

## 3 Elkan's accelerated $k$-means algorithm

**Initialization:** – Pick initial centers and set $l(x,c) = 0\ \forall x \forall c$.

- Assign $c(x) = \text{argmin}_c\, d(x,c)$, using 1(a) and $d(c,c')$ from Step-1.
- Every time we compute $d(x,c)$, set $l(x,c) = d(x,c)$.
- Assign $u(x) = \min_c d(x,c) = d(x,c(x))$.

**Repeat till convergence:** 1. Compute $d(c,c')\ \forall c,c'$. Compute $s(c) = \frac{1}{2}\min_{c' \neq c} d(c,c')\ \forall c$.

2. Identify $x$ such that $u(x) \leq s(c(x))$. According to result 1(c), we can avoid calculations for all such $x$ beyond this point.
3. For all remaining $x$ and $c$ such that

(i) $c \neq c(x)$
(ii) $u(x) > l(x, c)$ [Condition 2(b)]
(iii) $u(x) > \frac{1}{2}d(c(x), c)$ [Condition 1(b)]:

(a) Compute $d(x, c(x))$ and update $u(x) = d(x, c(x))$.
(b) If $d(x, c(x)) > l(x, c)$ or $d(x, c(x)) > \frac{1}{2}d(c(x), c)$, then compute $d(x, c)$. Now, if $d(x, c) < d(x, c(x))$, then assign $c(x) = c$ and update $u(x) = d(x, c(x))$.

4. Let $m(c)$ be the mean of points assigned to center $c$.
5. Assign $l(x, c) = \max\{l(x, c) - d(c, m(c)), 0\}\ \forall x, c$
6. $\forall x$, assign: $u(x) = u(x) + d(m(c(x)), c(x))$.
7. Replace $c$ by $m(c)$.

The following points are to be noted about the algorithm:

- at the starting of each iteration, the upper bounds $u(x)$ and lower bounds $l(x, c)$ are tight for most points $x$ and centers $c$, which in effect reduces the number of distance calculations in that iteration.
- There are many distance calculations in the initialization, but the number reduces thereafter.
- Each time $d(x, c)$ is calculated, update $l(x, c) = d(x, c)$. Also, whenever $c(x)$ is changed or $d(x, c(x))$ is computed, update $u(x) = d(x, c(x))$.
- Step-3(b) repeats checks (ii) and (iii) in order to avoid computing $d(x, c)$ if possible.

### 3.1 Limitations

During each iteration of the algorithm, the lower bounds $l(x, c)$ are updated for all points $x$ and centers $c$. These updates take $O(nk)$ time, so the complexity of the algorithm remains at least $O(nke)$, even though the number of distance calculations is roughly $O(n)$ only.

Also, the amount of memory overhead becomes large for large values of $n$ and $k$ since we need to maintain $n$ upper bounds, $kn$ lower bounds, and $O(k^2)$ inter-center distances.

## 4 Brief Overview of Hamerly Algorithm

Hamerly discusses a new algorithm which is built on Elkan's algorithm. On datasets of low and medium dimensions, this algorithm performs much better than other algorithms. It eliminates the need to execute the innermost loop of Lloyd's algorithm 80% of the times or more.

The new algorithm maintains two distance bounds per data point for its *two closest centers*. One is an upper bound on the distance to the closest center, and one is the lower bound on the distance to the second-closest center. Unlike Elkan's algorithm, this algorithm does not maintain lower bounds between all point-center combinations, so the lower bound is different. This algortihm uses the same triangle inequality results as Elkan to skip the entire loop completely.

## 5 Results

Without going into the details of Hamerly algorithm, some of the comparative study of the two algorithms is presented here.

Table-1 describes the asymptotic time and memory overheads of three accelerated algorithms. The $k$-d tree algorithm is the one that was discussed by Madhur in the class during his presentation. Using a $k$-d tree nearly doubles the amount of memory needed over Lloyd's algorithm, in order to keep summary statistics at each node, and to store the tree structure. Elkan's algorithm requires an additional $(n + 1)k$ scalar values for the bounds, as well as $k^2$ scalar values for the inter-center distances. These extra memory requirements become prohibitive for large datasets and large $k$.

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

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

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

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

A big advantage of Hamerly algorithm is its small memory footprint, especially compared with other accelerated algorithms. Hamerly algorithm consistently has the lowest memory use out of all the accelerated algorithms, and is usually close to the memory used by Lloyd's algorithm. Compared with Elkan's algorithm, which has memory requirements that scale as $O(k^2)$, Hamerly algorithm will have much lower memory footprint for large numbers of clusters. The $k$-d tree implementation uses a little less than twice the memory Hamerly algorithm uses.

Table-2 shows the performance of each algorithm. Hamerly algorithm is often more than 10 times faster in total time than Lloyd's algorithm. Its fastest out of all the algorithms in most experiments, including all 8-dimensional and 32-dimensional experiments. On 2-dimensional datasets, $k$-d trees perform very well. However, $k$-d tree performance degrades quickly as the dimension increases. For the three datasets that have 50 to 56 dimensions, Hamerly algorithm performs best when $k$ is small, and is comparable to Elkan's algorithm for large $k$.

## References

[1] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means.

[2] Greg Hamerly. 2010. Making k-means even faster.