
Fast K-Means with Accurate Bounds

James Newling

Idiap Research Institute & EPFL, Switzerland

JAMES.NEWLING@IDIAP.CH

François Fleuret

Idiap Research Institute & EPFL, Switzerland

FRANCOIS.FLEURET@IDIAP.CH

Abstract

We propose a novel accelerated exact k -means algorithm, which outperforms the current state-of-the-art low-dimensional algorithm in 18 of 22 experiments, running up to $3\times$ faster. We also propose a general improvement of existing state-of-the-art accelerated exact k -means algorithms through better estimates of the distance bounds used to reduce the number of distance calculations, obtaining speedups in 36 of 44 experiments, of up to $1.8\times$. We have conducted experiments with our own implementations of existing methods to ensure homogeneous evaluation of performance, and we show that our implementations perform as well or better than existing available implementations. Finally, we propose simplified variants of standard approaches and show that they are faster than their fully-fledged counterparts in 59 of 62 experiments.

1. Introduction

The k -means problem is to compute a set of k centroids to minimise the sum over data-points of the squared distance to the nearest centroid. It is an NP-hard problem for which various effective approximation algorithms exist. The most popular is often referred to as Lloyd’s algorithm, or simply as *the* k -means algorithm. It has applications in data compression, data classification, density estimation and many other areas, and was recognised in Wu et al. (2008) as one of the top-10 algorithms in data mining.

Lloyd’s algorithm relies on a two-step iterative process: In the *assignment* step, each sample is assigned to the cluster whose centroid is nearest. In the *update* step, cluster centroids are updated in accordance with their assigned samples. Lloyd’s algorithm is also called the *exact* k -means

algorithm, as there is no approximation in either of the two steps. This name can lead to confusion as the algorithm does not solve the k -means problem exactly.

The linear dependence on the number of clusters, the number of samples and the dimension of the space, means that Lloyd’s algorithm requires upwards of a billion floating point operations per round on standard datasets such as those used in our experiments (§4). This, coupled with slow convergence and the fact that several runs are often performed to find improved solutions, can make it slow.

Lloyd’s algorithm does not state how the assignment and update steps should be performed, and as such provides a scaffolding on which more elaborate algorithms can be constructed. These more elaborate algorithms, often called *accelerated* exact k -means algorithms, are the primary focus of this paper. They can be dropped-in wherever Lloyd’s algorithm is used.

1.1. Approximate k -means

Alternatives to exact k -means have been proposed. Certain of these rely on a relaxation of the assignment step, for example by only considering certain clusters according to some hierarchical ordering (Nister & Stewenius, 2006), or by using an approximate nearest neighbour search as in Philbin et al. (2007). Others rely on a relaxation of the update step, for example by using only a subset of data to update centroids (Frahling & Sohler, 2006; Sculley, 2010).

When comparing approximate k -means clustering algorithms such as those just mentioned, the two criteria of interest are the quality of the final clustering, and the computational requirements. The two criteria are not independent, making comparison between algorithms difficult and often preventing their adoption. When comparing accelerated exact k -means algorithms on the other hand, all algorithms produce the same final clustering, and so comparisons can be made based on speed alone. Once an accelerated exact k -means algorithm has been confirmed to provide a speed-up, it is rapidly adopted, automatically inheriting the trust which the exact algorithm has gained through its simplicity

and extensive use over several decades.

1.2. Accelerated Exact k -means

The first published accelerated k -means algorithms borrowed techniques used to accelerate the nearest neighbour search. Examples are the adaptation of the algorithm of Orchard (1991) in Phillips (2002), and the use of kd-trees (Bentley, 1975) in Kanungo et al. (2002). These algorithms relied on storing centroids in special data structures, enabling nearest neighbor queries to be processed without computing distances to all k centroids.

The next big acceleration (Elkan, 2003) came about by maintaining bounds on distances between samples and centroids, frequently resulting in more than 90% of distance calculations being avoided. It was later shown (Hamerly, 2010) that in low-dimensions, it is more effective to keep bounds on distances to only the two nearest centroids, and that in general bounding-based algorithms are significantly faster than tree-based ones. Further bounding-based algorithms were proposed by Drake (2013) and Ding et al. (2015), each providing accelerations over their predecessors in certain settings. In this paper, we continue in the same vein.

1.3. Our Contribution

Our first contribution (§3.1) is a new bounding-based accelerated exact k -means algorithm, the Exponion algorithm. Its closest relative is the Annular algorithm (Drake, 2013), which is the current state-of-the-art accelerated exact k -means algorithm in low-dimensions. We show that the Exponion algorithm is significantly faster than the Annular algorithm on a majority of low-dimensional datasets.

Our second contribution (§3.2) is a technique for making bounds tighter, allowing further redundant distance calculations to be eliminated. The technique, illustrated in Figure 1, can be applied to all existing bounding-based k -means algorithms.

Finally, we show how certain of the current state-of-the-art algorithms can be accelerated through strict simplifications (§2.2 and §2.6). Fully parallelised implementations of all algorithms are provided under an open-source license at <https://github.com/idiap/eakmeans>

2. Notation and baselines

We describe four accelerated exact k -means algorithms in order of publication date. For two of these we propose simplified versions which offer natural stepping stones in understanding the full versions, as well being faster (§4.1.2).

Our notation is based on that of Hamerly (2010), and only where necessary is new notation introduced. We use for ex-

ample N for the number of samples and k for the number of clusters. Indices i and j always refer to data and cluster indices respectively, with a sample denoted by $x(i)$ and the index of the cluster to which it is assigned by $a(i)$. A cluster’s centroid is denoted as $c(j)$. We introduce new notation by letting $n_1(i)$ and $n_2(i)$ denote the indices of the clusters whose centroids are the nearest and second nearest to sample i respectively.

Note that $a(i)$ and $n_1(i)$ are different, with the objective in a round of k -means being to set $a(i)$ to $n_1(i)$. $a(i)$ is a variable maintained by algorithms, changing within loops whenever a better candidate for the nearest centroid is found. On the other hand, $n_1(i)$ is introduced purely to aid in proofs, and is external to any algorithmic details. It can be considered to be the hidden variable which algorithms need to reveal.

All of the algorithms which we consider are elaborations of Lloyd’s algorithm, and thus consist of repeating the assignment step and update step, given respectively as

$$a(i) \leftarrow n_1(i), \quad i \in \{1, \dots, N\} \quad (1)$$

$$c(j) \leftarrow \frac{\sum_{i:a(i)=j} x(i)}{\|i : a(i) = j\|}, \quad j \in \{1, \dots, k\}. \quad (2)$$

These two steps are repeated until there is no change to any $a(i)$, or some other stopping criterion is met. We reiterate that all the algorithms discussed provide the same output at each iteration of the two steps, differing only in how $a(i)$ is computed in (1).

2.1. Standard algorithm (sta)

The Standard algorithm, henceforth `sta`, is the simplest implementation of Lloyd’s algorithm. The only variables kept are $x(i)$ and $a(i)$ for $i \in \{1, \dots, N\}$ and $c(j)$ for $j \in \{1, \dots, k\}$. The assignment step consists of, for each i , calculating the distance from $x(i)$ to all centroids, thus revealing $n_1(i)$.

2.2. Simplified Elkan’s algorithm (selk)

Simplified Elkan’s algorithm, henceforth `selk`, uses a strict subset of the strategies described in Elkan (2003). In addition to $x(i)$, $a(i)$ and $c(j)$, the variables kept are $p(j)$, the distance moved by $c(j)$ in the last update step, and bounds $l(i, j)$ and $u(i)$, maintained to satisfy,

$$l(i, j) \leq \|x(i) - c(j)\|, \quad u(i) \geq \|x(i) - c(a(i))\|.$$

These bounds are used to eliminate unnecessary centroid-data distance calculations using,

$$\begin{aligned} u(i) < l(i, j) &\implies \|x(i) - c(a(i))\| < \|x(i) - c(j)\| \\ &\implies j \neq n_1(i). \end{aligned} \quad (3)$$

We refer to (3) as an *inner* test, as it is performed within a loop over centroids for each sample. This as opposed to an *outer* test which is performed just once per sample, examples of which will be presented later.

To maintain the correctness of the bounds when centroids move, bounds are updated at the beginning of each assignment step with

$$l(i, j) \leftarrow l(i, j) - p(j), \quad u(i) \leftarrow u(i) + p(a(i)). \quad (4)$$

The validity of these updates is a simple consequence of the triangle inequality (proof in B.1). We say that a bound is *tight* if it is known to be equal to the distance it is bounding, a *loose* bound is one which is not tight. For `selk`, bounds are initialised to be tight, and tightening a bound evidently costs one distance calculation.

When in a given round $u(i) \geq l(i, j)$, the test (3) fails. The first time this happens, both $u(i)$ and $l(i, j)$ are loose due to preceding bound updates of the form (4). Tightening either bound may result in the test succeeding. Bound $u(i)$ should be tightened before $l(i, j)$, as it reappears in all tests for sample i and will thus be reused. In the case of a test failure with tight $u(i)$ and loose $l(i, j)$ we tighten $l(i, j)$. A test failure with $u(i)$ and $l(i, j)$ both tight implies that centroid j is nearer to sample i than the currently assigned cluster centroid, and so $a(i) \leftarrow j$ and $u(i) \leftarrow l(i, j)$.

2.3. Elkan’s algorithm (`e1k`)

The fully-fledged algorithm of Elkan (2003), henceforth `e1k`, adds to `selk` an additional strategy for eliminating distance calculations in the assignment step. Two further variables, $cc(j, j')$, the matrix of inter-centroid distances, and $s(j)$, the distance from centroid j to its nearest other centroid, are kept. A simple application of the triangle inequality, shown in B.2, provides the following test,

$$\frac{cc(a(i), j)}{2} > u(i) \implies j \neq n_1(i). \quad (5)$$

`e1k` uses (5) in unison with (3) to obtain an improvement on the test of `e1k`, of the form,

$$\max \left(l(i, j), \frac{cc(a(i), j)}{2} \right) > u(i) \implies j \neq n_1(i). \quad (6)$$

In addition to the inner test (6), `e1k` uses an outer test, whose validity follows from that of (5), given by,

$$\frac{s(a(i))}{2} > u(i) \implies n_1(i) = a(i). \quad (7)$$

If the outer test (7) is successful, one proceeds immediately to the next sample without changing $a(i)$.

2.4. Hamerly’s algorithm (`ham`)

The algorithm of Hamerly (2010), henceforth `ham`, represents a shift of focus from inner to outer tests, completely foregoing the inner test of `e1k`, and providing an improved outer test.

The k lower bounds per sample of `e1k` are replaced by a single lower bound on all centroids other than the one assigned,

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

The variables $p(j)$ and $u(i)$ used in `e1k` have the same definition for `ham`. The test for a sample i is

$$\max \left(l(i), \frac{s(a(i))}{2} \right) > u(i) \implies n_1(i) = a(i), \quad (8)$$

with the proof of correctness being essentially the same as that for the inner test of `e1k`. If test (8) fails, then $u(i)$ is made tight. If test (8) fails with $u(i)$ tight, then the distances from sample i to all centroids are computed, revealing $n_1(i)$ and $n_2(i)$, allowing the updates $a(i) \leftarrow n_1(i)$, $u(i) \leftarrow \|x(i) - c(n_1(i))\|$ and $l(i) \leftarrow \|x(i) - c(n_2(i))\|$. As with `e1k`, at the start of the assignment step, bounds need to be adjusted to ensure their correctness following the update step. This is done via,

$$l(i) \leftarrow l(i) - \arg \max_{j \neq a(i)} p(a(i)), \quad u(i) \leftarrow u(i) + p(a(i)).$$

2.5. Annular algorithm (`ann`)

The Annular algorithm of Drake (2013), henceforth `ann`, is a strict extension of `ham`, adding one novel test. In addition to the variables used in `ham`, one new variable $b(i)$ is required, which roughly speaking is to $n_2(i)$ what $a(i)$ is to $n_1(i)$. Also, the centroid norms $\|c(j)\|$ should be computed and sorted in each round.

Upon failure of test (8) with tight bounds in `ham`, $\|x(i) - c(j)\|$ is computed for all $j \in \{1, \dots, k\}$ to reveal $n_1(i)$ and $n_2(i)$. With `ann`, certain of these k calculations can be eliminated. Define the radius, and corresponding set of cluster indices,

$$R(i) = \max(u(i), \|x(i) - c(b(i))\|), \quad \mathcal{J}(i) = \{j : \|\|c(j)\| - \|x(i)\|\| \leq R(i)\}. \quad (9)$$

The following implication (proved in B.3) is used

$$j \notin \mathcal{J}(i) \implies j \notin \{n_1(i), n_2(i)\}.$$

Thus only distances from sample i to centroids of the clusters whose indices are in $\mathcal{J}(i)$ need to be calculated for $n_1(i)$ and $n_2(i)$ to be revealed. Once $n_1(i)$ and $n_2(i)$ revealed, $a(i)$, $u(i)$ and $l(i)$ are updated as per `ham`, and $b(i) \leftarrow n_2(i)$.

Note that by keeping an ordering of $\|c(j)\|$ the set $\mathcal{J}(i)$ can be determined in $\Theta(\log(K))$ operations with two binary searches, one for each of the inner and outer radii of $\mathcal{J}(i)$.

2.6. Simplified Yinyang (**syin**) and Yinyang (**yin**) algorithms

The basic idea with the Yinyang algorithm (Ding et al., 2015) and the Simplified Yinyang algorithm, henceforth **yin** and **syin** respectively, is to maintain consistent lower bounds for groups of clusters as a compromise between the $k - 1$ lower bounds of **elk** and the single lower bound of **ham**. In Ding et al. (2015) the number of groups is fixed at one tenth the number of centroids. The groupings are determined and fixed by an initial clustering of the centroids. The algorithm appearing in the literature most similar to **yin** is Drake’s algorithm of (Drake & Hamerly, 2012), not to be confused with **ann**. According to Ding et al. (2015), Drake’s algorithm does not perform as well as **yin**, and we thus choose not to consider it in this paper.

Denote by G the number of groups of clusters. Variables required in addition to those used in **sta** are $p(j)$ and $u(i)$, as per **elk**, $\mathcal{G}(f)$, the set of indices of clusters belonging to the f ’th group, $g(i)$, the group to which cluster $a(i)$ belongs, $q(f) = \max_{j \in \mathcal{G}} p(j)$, and bound $l(i, f)$, maintained to satisfy,

$$l(i, f) \leq \arg \min_{j \in \mathcal{G}(f) \setminus \{a(i)\}} \|x(i) - c(j)\|.$$

For both **syin** and **yin**, both an outer test and group tests are used. To these, **yin** adds an inner test. The outer test is

$$\min_{f \in \{1, \dots, G\}} l(i, f) > u(i) \implies a(i) = n_1(i). \quad (10)$$

If and when test (10) fails, group tests of the form

$$l(i, f) > u(i) \implies a(i) \notin \mathcal{G}(f), \quad (11)$$

are performed. As with **elk** and **ham**, if test (11) fails with $u(i)$ loose, $u(i)$ is made tight and the test reperformed.

The difference between **syin** and **yin** arises when (11) fails with $u(i)$ tight. With **syin**, the simple approach of computing distances from $x(i)$ to all centroids in $\mathcal{G}(f)$, then updating $l(i, f)$, $l(i, g(i))$, $u(i)$, $a(i)$ and $g(i)$ as necessary, is taken. With **yin** a final effort at eliminating distance calculations by the use of a local test is made, as described in C.1. As will be shown (§4.1.2), it is not clear that the local test of **yin** makes it any faster. Finally, we mention how $u(i)$ and $l(i, f)$ are updated at the beginning of the assignment step for the **syin** and **yin**,

$$\begin{aligned} l(i, f) &\leftarrow l(i, f) - \arg \max_{j \in \mathcal{G}(f)} p(a(i)), \\ u(i) &\leftarrow u(i) + p(a(i)). \end{aligned}$$

3. Contributions and New Algorithms

We first present (§3.1) an algorithm which we call **Exponion**, and then (§3.2) an improved bounding approach.

3.1. Exponion algorithm (**exp**)

Like **ann**, **exp** is an extension of **ham** which adds a test to filter out $j \notin \{n_1(i), n_2(i)\}$ when test (8) fails. Unlike **ann**, where the filter is an origin-centered annulus, **exp** has as filter a ball centred on centroid $a(i)$. This change is motivated by the ratio of volumes of an annulus of width r at radius w and a ball of radius r from the origin, which is $d \left(\frac{w}{r}\right)^{d-1}$ in \mathbb{R}^d . We expect r to be greater than w , whence the expected improvement. Define,

$$\begin{aligned} R(i) &= 2u(i) + s(a(i)), \\ \mathcal{J}(i) &= \{j : \|c(j) - c(a(i))\| \leq R(i)\}. \end{aligned} \quad (12)$$

The underlying test used (proof in B.4) is

$$j \notin \mathcal{J}(i) \implies j \notin \{n_1(i), n_2(i)\}.$$

In moving from **ann** to **exp**, the decentralisation from the origin to the centroids incurs two costs, one which can be explained algorithmically, the other is related to cache memory.

Recall that **ann** sorts $\|c(j)\|$ in each round, thus guaranteeing that the set of candidate centroids (9) can be obtained in $O(\log(C))$ operations. To guarantee that the set of candidate centroids (12) can be obtained with $O(\log(C))$ operations requires that $\|c(j) - c(a(i))\|$ be sorted. For this to be true for all samples requires sorting $\|c(j) - c(j')\|$ for all $j \in \{1, \dots, k\}$, increasing the overhead of sorting from $O(k \log k)$ to $O(k^2 \log k)$.

The cache related cost is that, unless samples are ordered by $a(i)$, the bisection search performed to obtain $\mathcal{J}(i)$ is done with a different row of $c(j, j')$ for each sample, resulting in cache memory misses.

To offset these costs, we replace the exact sorting of cc with a partial sorting, paying for this approximation with additional distance calculations. We maintain, for each centroid, $\lceil \log_2 k \rceil$ concentric annuli, each successive annulus containing twice as many centroids as the one interior to it. For cluster j , annulus $f \in \{1, \dots, \lceil \log_2 k \rceil\}$ is defined by inner and outer radii $e(j, f - 1)$ and $e(j, f)$, and a list of indices $w(j, f)$ with $|w(j, f)| = 2^f$, where

$$w(j, f) = \{j' : e(j, f - 1) < \|c(j') - c(j)\| \leq e(j, f)\}.$$

Note that $w(j, f)$ is not an ordered set, but there is an ordering between sets,

$$\begin{aligned} j' \in w(j, f), j'' \in w(j, f + 1) &\implies \\ \|c(j') - c(j)\| &< \|c(j'') - c(j)\|. \end{aligned}$$

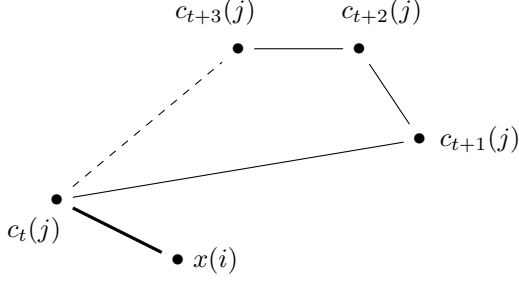


Figure 1. The classical sn-bound is the sum of the last known distance between the sample to a previous position of the centroid (thick solid line), with all the distances between successive positions of the centroid since then (thin solid lines). The ns-bound we propose uses the actual distance between that previous location of the centroid and its current one (dashed line).

Given a search radius $R(i)$, without a complete ordering of $c(j, j')$ we cannot obtain $\mathcal{J}(i)$ in $O(\log(k))$ operations, but we can obtain a slightly larger set $\mathcal{J}^*(i)$ defined by

$$f^*(i) = \min\{f : e(a(i), f) \geq R(i)\},$$

$$\mathcal{J}^*(i) = \bigcup_{f \leq f^*(i)} w(j, f),$$

in $\log \log(k)$ operations. It is easy to see that $|\mathcal{J}^*(i)| \leq 2|\mathcal{J}(i)|$, and so using the partial sorting cannot cost more than twice the number of distance calculations.

3.2. Improving bounds (sn to ns)

In all the algorithms presented so far, upper bounds (lower bounds) are updated in each round with increments (decrements) of norms of displacements. If tests are repeatedly successful, these increments (decrements) accumulate. Consider for example the upper bound update,

$$u_{t_0+1}(i) \leftarrow u_{t_0}(i) + p_{t_0}(a(i)),$$

where subscripts denote rounds. The upper bound after δt such updates without bound tightening is

$$u_{t_0+\delta t}(i) = u_{t_0}(i) + \sum_{t'=t_0}^{t_0+\delta t-1} p_{t'}(a(i)). \quad (13)$$

The summation term is a (s)um of (n)orms of displacement, thus we refer to it as an sn-bound and to an algorithm using only such an update scheme as an sn-algorithm. An alternative upper bound at round $t_0 + \delta t$ is,

$$u_{t_0+\delta t}(i) = u_{t_0}(i) + \left\| \sum_{t'=t_0}^{t_0+\delta t-1} c_{t'+1}(i) - c_{t'}(i) \right\|,$$

$$= u_{t_0}(i) + \|c_{t_0+\delta t}(i) - c_{t_0}(i)\|. \quad (14)$$

Bound (14) derives from the (n)orm of a (s)um, and hence we refer to it as an ns-bound. An ns-bound is guaranteed to be tighter than its equivalent sn-bound by a trivial application of the triangle inequality (proved in B.5). We have presented an upper ns-bound, but lower ns-bound formulations are similar. In fact, for cases where lower bounds apply to several distances simultaneously, due to the additional operation of taking a group maximum, there are three possible ways to compute a lower bound, details in §C.2.

3.3. Simplified Elkan's algorithm-ns (selk-ns)

In transforming an sn-algorithm into an ns-algorithm, additional variables need to be maintained. These include a record of previous centroids C , where $C(j, t) = c_t(j)$, and displacement of $c(j)$ with respect to previous centroids, $P(j, t) = \|c(j) - c_t(j)\|$. We no longer keep rolling bounds for each sample, instead we keep a record of when most recently bounds were made tight and the distances then calculated. For Simplified Elkan's Algorithm-ns, henceforth selk-ns, we define $T(i, j)$ to be the last time $\|x(i) - c(j)\|$ was calculated, with corresponding distance $l(i, j) = \|x(i) - c_{T(i,j)}(j)\|$. We emphasise that $l(i, j)$ is defined differently here to in selk, with $u(i)$ similarly redefined as $u(i) = \|x(i) - c_{T(i,a(i))}(a(i))\|$.

The underlying test is

$$u(i) + P(a(i), T(i, a(i))) < l(i, j) - P(j, T(i, j))$$

$$\implies j \neq n_1(i).$$

As with selk, the first bound failure for sample i results in $u(i)$ being updated, with subsequent failures resulting in $l(i, j)$ being updated to the current distance. In addition, when $u(i)$ ($l(i, j)$) is updated, $T(i, a(i))$ ($T(i, j)$) is set to the current round.

Due to the additional variables C , P and T , the memory requirement imposed is larger with selk-ns than with selk-sn. Ignoring constants, in round t the memory requirement assuming samples of size $O(d)$ is,

$$\text{mem}_{ns} = O(Nd + Nk + ktd),$$

where x , l and C are the principal contributors to the above three respective terms. selk consists of only the first two terms, and so when $t > N/\min(k, d)$, the dominant memory consumer in selk-ns is the new variable C . To guarantee that C does not dominate memory consumption, an sn-like reset is performed in rounds $\{t : t \equiv 0 \pmod{N/\min(k, d)}\}$, consisting of the following updates,

$$u(i) \leftarrow u(i) + P(a(i), T(i, a(i))),$$

$$l(i, j) \leftarrow l(i, j) - P(j, T(i, j)),$$

$$T(i, j) \leftarrow t,$$

and finally the clearing of C .

3.4. Changing Bounds for Other Algorithms

All sn- to ns- conversions are much the same as that described in Section 3.3. We have implemented versions of `elk`, `syn` and `exp` using ns-bounds, which we refer to as `elk-ns`, `syn-ns` and `exp-ns` respectively.

4. Experiments and Results

Our first set of experiments are conducted using a single core. We first establish that our implementations of baseline algorithms are as fast or faster than existing implementations. Having done this, we consider the effects of the novel algorithmic contributions presented, simplification, the Exponion algorithm, and ns-bounding. The final set of experiments are conducted on multiple cores, and illustrate how all algorithms presented parallelise gracefully.

We compare 23 k -means implementations, including our own implementations of all algorithms described, original implementations accompanying the papers (Hamerly, 2010; Drake, 2013; Ding et al., 2015), and implementations in two popular machine learning libraries, VLFeat and mlpack. We use the following notation to refer to implementations: `{codesource-algorithm}`, where `codesource` is one of `bay` (Hamerly, 2015), `mlp` (Curtin et al., 2013), `pow` (Low et al., 2010), `vlf` (Vedaldi & Fulkerson, 2008) and `own` (our own code), and `algorithm` is one of the algorithms described.

Unless otherwise stated, times are wall times excluding data loading. We impose a time limit of 40 minutes and a memory limit of 4 GB on all `{dataset, implementation, k, seed}` runs. If a run fails to complete in 40 minutes, the corresponding table entry is ‘t’. Similarly, failure to execute with 4GB of memory results in a table entry ‘m’. We confirm that for all `{dataset, k, seed}` triplets, all implementations which complete within the time and memory constraint take the same number of iterations to converge to a common local minimum, as expected.

The implementations are compared over the 22 datasets presented in Table 1, for $k \in \{100, 1000\}$, with 10 distinct centroid initialisations (seeds). For all `{dataset, k, seed}` triplets, the 23 implementations are run serially on a machine with an Intel i7 processor and 8MB of cache memory. All experiments are performed using double precision floating point numbers.

Findings in Drake (2013) suggest that the best algorithm to use for a dataset depends primarily on dimension, where in low-dimensions, `ham` and `ann` are fastest, in high-dimensions `elk` is fastest, and in intermediate dimensions an approach maintaining a fractional number of bounds, Drake’s algorithm, is fastest. Our findings corroborate these on real datasets, although the lines separating the

	d	N		d	N		d	N		d	N
i	2	100k	vi	4	200k	xii	28	66k	xvii	68	2.6m
ii	2	169k	vii	4	200k	xiii	30	1m	xviii	74	146k
iii	2	1m	viii	9	68k	xiv	50	60k	xix	108	1m
iv	3	165k	ix	11	41k	xv	50	130k	xx	128	14k
v	3	164k	x	15	166k	xvi	55	581k	xxi	310	95k
			xi	17	23k				xxii	784	60k

Table 1. The 22 datasets used in experiments, ranging in dimension from 2 to 784. The datasets come from: the UCI, KDD and KEEL repositories (11,2,2), MNIST and STL-10 image databases (2,1), random (2), European Bioinformatics Institute (1) and Joensuu University (1). Full names and further details in D.

three groups are blurry. In presenting our results we prefer to consider a partitioning of the datasets into just two groups about the dimension $d = 20$. `ham` and its derivatives are considered for $d < 20$, `elk` and its derivatives for $d \geq 20$, and `syn` and `yn` for all d .

4.1. Single core experiments

A complete presentation of wall times and number of iterations for all `{dataset, implementation, k}` triplets is presented over two pages in Tables 2 and 3 (§D). Here we attempt to summarise our findings. We first compare implementations of published algorithms (§4.1.1), and then show how `selk` and `syn` often outperform their more complex counterparts (§4.1.2). We show that `exp` is in general much faster than `ann` (§4.1.3), and finally show how using ns-bounds can accelerate algorithms (§4.1.4).

4.1.1. COMPARING IMPLEMENTATIONS OF BASELINES

There are algorithmic techniques which can speedup all k -means algorithms discussed in this paper, we mention a few which we use. One is pre-computing the squares of norms of all samples just once, and those of centroids once per round. Another, first suggested in Hamerly (2010), is to update the sum of samples by considering only those samples whose assignment changed in the previous round. A third optimisation technique is to decompose while-loops which contain inner branchings dependant on the tightness of upper bounds into separate while-loops, eliminating unnecessary comparisons. Finally, while there are no large matrix operations with bounding-based algorithms, in high-dimensions distance calculations can be accelerated by the use of SSE, as in VLFeat, or by fast implementations of BLAS, such as OpenBLAS (Xianyi, 2016).

Our careful attention to optimisation is reflected in Table 7 (§A), where implementations of `elk`, `ham`, `ann` and `yn` are compared. The values shown are ratios of mean run-times using another implementation (column) and our own implementation of the same algorithm, on a given dataset

(row). Our implementations are faster in all but 4 comparisons.

4.1.2. BENEFITS OF SIMPLIFICATION

We compare published algorithms `elk` and `yin` with their simplified counterparts `selk` and `syin`. The values in Table 2 are ratios of mean runtimes using simplified and original algorithms, values less than 1 mean that the simplified version is faster. We observe that `selk` is faster than `elk` in 16 of 18 experiments, and `syin` is faster than `yin` in 43 of 44 experiments, often dramatically so.

It is interesting to ask why the inventors of `elk` and `yin` did not instead settle on algorithms `selk` and `syin` respectively. A partial answer might relate to the use of BLAS, as the speedup obtained by simplifying `yin` to `syin` never exceeds more than 10% when BLAS is deactivated. `syin` is more responsive to BLAS than `yin` as it has larger matrix multiplications due to it not having a final filter.

own-yin → own-syin				own-elk → own-selk				
	100	1000		100	1000		100	1000
i	0.96	0.90	xii	0.58	0.76	xii	0.85	<i>1.05</i>
ii	<i>1.03</i>	0.86	xiii	0.66	0.61	xiii	0.97	m
iii	0.88	0.92	xiv	0.50	0.55	xiv	0.84	0.57
iv	0.94	0.87	xv	0.49	0.58	xv	0.54	0.49
v	0.93	0.88	xvi	0.49	0.66	xvi	0.92	m
vi	0.91	0.87	xvii	0.44	0.58	xvii	0.75	m
vii	0.96	0.90	xviii	0.42	0.47	xviii	0.86	0.66
viii	0.79	0.80	xix	0.36	0.42	xix	0.72	m
ix	0.77	0.80	xx	0.38	0.60	xx	<i>1.12</i>	0.74
x	0.72	0.73	xxi	0.32	0.36	xxi	0.89	0.73
xi	0.64	0.71	xxii	0.36	0.38	xxii	0.99	0.89

Table 2. Comparing `yin` and `elk` to simplified versions `syin` and `selk`. Values are ratios of mean runtimes of simplified versions to their originals, for different low-dimensional datasets (rows) and k (columns). Values less than 1 mean that the simplified version is faster. In all but 3 of 62 cases (italicised), simplification results in speedup, by as much as $3\times$.

4.1.3. FROM ANNULAR TO EXPONION

We compare the Annular algorithm (`ann`) with the Exponion algorithm (`exp`). The values in Table 3 are ratios of mean runtimes (columns q_t) and of mean number of distance calculations (columns q_{au}). Values less than 1 denote better performance with `exp`. We observe that `exp` is markedly faster than `ann` on most low-dimensional datasets, reducing by more than 30% the mean runtime in 17 of 22 experiments. The primary reason for the speedup is the reduced number of distance calculations.

Table 4 summarises how many times each of the sn-algorithms is fastest on the 44 {dataset, k } experiments, ns-algorithms excluded. The 13 experiments on which `exp` is fastest are all very low-dimensional ($d < 5$), the 24 on which `syin` is fastest are intermediate ($8 < d < 69$) and `selk` or `elk` are fastest in very high dimensions ($d > 73$). For a detailed comparison across all algorithms, consult Tables 2 and 3 (§D).

own-ann → own-exp									
	100		1000			100		1000	
	q_t	q_{au}	q_t	q_{au}		q_t	q_{au}	q_t	q_{au}
i	0.48	0.52	0.72	0.61	vii	0.71	0.80	0.36	0.32
ii	0.54	0.80	0.58	0.50	viii	<i>1.12</i>	<i>1.24</i>	<i>1.02</i>	<i>0.93</i>
iii	0.53	0.58	0.48	0.44	ix	0.96	0.99	0.73	0.64
iv	0.63	0.80	0.36	0.33	x	0.67	0.65	0.55	0.41
v	0.63	0.80	0.37	0.34	xi	<i>1.24</i>	<i>1.43</i>	<i>1.30</i>	<i>1.16</i>
vi	0.62	0.73	0.42	0.38					

Table 3. Ratios of mean runtimes (q_t) and mean number of distance calculations (q_{au}) using the Exponion (`own-exp`) and Annular (`own-ann`) algorithms, on datasets with $d < 20$. Exponion is faster in all but the four italicised cases. The speedup is primarily due to the reduced number of distance calculations.

ham	ann	exp	syin	yin	selk	elk
0	0	13	24	0	6	1

Table 4. Number of times each sn-algorithm is fastest, over the 44 {dataset, k } experiments, ns-algorithms not considered here.

4.1.4. FROM SN TO NS BOUNDING

For each of the 44 {dataset, k } experiments, we compare the fastest sn-algorithm with its ns-variant. The results are presented in Table 5. Columns ‘x’ denote the fastest sn-algorithm. Values are ratios of means over runs of some quantity using the ns- and sn- variants. The ratios are q_t (runtimes), q_a (number of distance calculations in the assignment step) and q_{au} (total number of distance calculations).

In all but 8 of 44 experiments (italicised), we observe a speedup using ns-bounding, by up to 45%. As expected, the number of distance calculations in the assignment step is never greater when using ns-bounds, however the total number of distance calculations is occasionally be increased due to initial variables being maintained.

4.2. Multicore experiments

We have implemented parallelised versions of all algorithms described in this paper using the C++11 thread support library. To measure the speedup using multiple cores, we compare the runtime using four threads to that using one thread on a non-hyperthreading four core machine.

own-x \rightarrow own-x-ns								
	100				1000			
	x	q_t	q_a	q_{au}	x	q_t	q_a	q_{au}
i	exp	0.96	0.97	0.99	exp	0.99	0.98	1.00
ii	exp	0.94	0.97	0.97	exp	0.99	0.99	1.00
iii	exp	0.95	0.97	0.98	exp	0.98	0.96	0.99
iv	exp	0.97	0.97	0.97	exp	0.99	0.97	0.98
v	exp	0.96	0.97	0.97	exp	0.98	0.96	0.98
vi	exp	0.95	0.96	0.97	exp	0.97	0.96	0.98
vii	syin	0.98	0.82	0.86	exp	0.98	0.96	0.99
viii	syin	0.98	0.86	0.88	syin	0.87	0.44	0.65
ix	syin	0.98	0.83	0.86	syin	0.83	0.32	0.66
x	syin	<i>1.03</i>	<i>0.91</i>	<i>0.92</i>	syin	<i>1.11</i>	<i>0.72</i>	<i>0.80</i>
xi	selk	0.92	0.80	0.84	syin	0.81	0.56	0.69
xii	syin	<i>1.00</i>	<i>0.86</i>	<i>0.88</i>	syin	0.96	0.51	0.85
xiii	syin	0.96	0.84	0.84	syin	0.87	0.58	0.61
xiv	syin	0.99	0.86	0.87	syin	0.74	0.51	0.63
xv	syin	<i>1.06</i>	<i>0.93</i>	<i>0.94</i>	syin	0.94	0.58	0.69
xvi	syin	<i>1.04</i>	<i>0.91</i>	<i>0.93</i>	syin	0.98	0.61	0.79
xvii	syin	1.00	0.87	0.89	syin	m	m	m
xviii	selk	0.89	0.81	0.82	syin	0.75	0.64	0.68
xix	selk	0.88	0.84	0.85	syin	0.91	0.75	0.77
xx	elk	<i>1.02</i>	<i>0.96</i>	<i>1.02</i>	selk	<i>1.06</i>	<i>0.99</i>	<i>1.00</i>
xxi	selk	0.85	0.81	0.82	selk	0.72	0.72	0.73
xxii	selk	0.80	0.77	0.78	selk	0.55	0.68	0.69

Table 5. The effect of using ns-bounds. Columns ‘x’ denotes the fastest sn-algorithm for a particular {dataset, k } experiment. Columns ‘ q_t ’ denote the ratio of mean runtimes of ns- and sn-variants of x. Italicised values are cases where using ns-bounding results in a slow down ($q_t > 1$), in the majority of cases there is a speedup. ‘ q_a ’ and ‘ q_{au} ’ denote ratios of ns- to sn- mean number of distance calculations in the assignment step (a) and in total (au). ‘m’ described in paragraph 3 of §4.

The results are summarised in Table 6, where near fourfold speedups are observed.

i-xi		xii-xxii	
100	1000	100	1000
own-exp-ns	0.29 0.31	own-selk-ns	0.33 0.30
own-syin-ns	0.31 0.29	own-elk-ns	0.30 0.28
		own-syin-ns	0.27 0.27

Table 6. The median speedup using four cores. The median is over i-xi on the left and xii-xxii on the right.

5. Conclusion and future work

The experimental results presented show that the ns-bounding scheme makes exact k -means algorithms faster, and that our Exponion algorithm is significantly faster than existing state-of-the-art algorithms in low-dimensions. Both can be seen as good default choices for k -means clus-

tering on large data-sets.

The main practical weakness that remains is the necessary prior selection of which algorithm to use, depending on the dimensionality of the problem at hand. This should be addressed through an adaptive procedure able to select automatically the optimal algorithm through an efficient exploration/exploitation strategy. The second and more prospective direction of work will be to introduce a sharing of information between samples, instead of processing them independently.

Acknowledgements

James Newling was funded by the Hasler Foundation under the grant 13018 MASH2.

A. Table Comparing Implementations

		bay-ham	mlp-ham	bay-ann	pow-yin		pow-yin	bay-elk	mlp-elk	vlf-elk
100	i	1.23	1.04	<i>0.78</i>	7.52	xii	2.51	2.69	1.87	2.48
	ii	1.28	<i>0.99</i>	<i>0.86</i>	4.91	xiii	3.84	1.36	1.56	t
	iii	1.19	1.27	<i>0.88</i>	9.84	xiv	1.98	1.75	1.53	2.72
	iv	1.59	1.15	1.24	6.21	xv	2.21	1.48	1.48	2.01
	v	1.59	1.20	1.24	6.01	xvi	2.30	1.68	2.00	2.85
	vi	1.78	1.27	1.32	6.41	xvii	m	1.79	1.88	2.61
	vii	1.78	1.17	1.48	5.63	xviii	1.69	1.91	1.46	2.68
	viii	2.67	1.38	2.38	3.99	xix	1.49	1.64	1.74	2.44
	ix	2.93	1.51	2.90	3.65	xx	1.35	2.53	2.21	2.41
	x	3.59	1.75	2.67	3.28	xxi	1.24	2.35	1.57	1.81
	xi	3.89	2.04	3.18	2.17	xxii	1.16	2.86	1.43	1.35
1000	i	1.51	1.03	1.06	7.57	xii	3.37	6.21	3.20	2.44
	ii	1.52	1.04	1.17	8.03	xiii	t	m	m	m
	iii	1.47	1.05	1.04	8.57	xiv	2.09	1.89	1.86	2.07
	iv	1.77	1.09	1.59	6.98	xv	3.14	1.43	2.76	1.80
	v	1.77	1.09	1.59	7.01	xvi	3.98	m	m	m
	vi	2.07	1.17	1.79	7.23	xvii	m	m	m	m
	vii	1.99	1.17	1.73	6.57	xviii	1.82	1.78	1.40	1.92
	viii	3.01	1.38	2.97	4.63	xix	t	m	m	m
	ix	3.28	1.58	3.34	4.06	xx	2.06	6.17	2.60	1.72
	x	3.92	1.76	3.57	5.08	xxi	1.32	2.88	1.80	1.51
	xi	4.08	1.99	4.03	2.89	xxii	1.17	4.82	1.74	1.28

Table 7. Comparing implementations. For 100 (above) and 1000 (below) clusters, and in low- (left) and high- (right) dimensions. Existing implementations (columns) of ham, ann, yin and elk are compared to our implementations as a ratio of mean runtimes, with the mean runtime of our implementation in the denominator. Values greater than 1 mean our implementation runs faster. ‘t’ and ‘m’ are described in paragraph 3 of §4.

References

- Bentley, Jon Louis. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9): 509–517, September 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL <http://doi.acm.org/10.1145/361002.361007>.
- Curtin, Ryan R., Cline, James R., Slagle, Neil P., March, William B., Ram, P., Mehta, Nishant A., and Gray, Alexander G. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14: 801–805, 2013.
- Ding, Yufei, Zhao, Yue, Shen, Xipeng, Musuvathi, Madanlal, and Mytkowicz, Todd. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pp. 579–587, 2015. URL <http://jmlr.org/proceedings/papers/v37/ding15.html>.
- Drake, Jonathan. Faster k-means clustering, 2013. Accessed online 19 August 2015.
- Drake, Jonathan and Hamerly, Greg. Accelerated k-means with adaptive distance bounds. In *5th NIPS Workshop on Optimization for Machine Learning, 2012*, pp. 42–53, 2012. URL http://opt-ml.org/oldopt/papers/opt2012_paper_13.pdf.
- Elkan, Charles. Using the triangle inequality to accelerate k-means. In *Machine Learning, Proceedings of the Twentieth International Conference (ICML 2003), August 21-24, 2003, Washington, DC, USA*, pp. 147–153, 2003. URL <http://www.aaai.org/Library/ICML/2003/icml03-022.php>.
- Frahling, Gereon and Sohler, Christian. A fast k-means implementation using coresets. In *Proceedings of the Twenty-second Annual Symposium on Computational Geometry, SCG '06*, pp. 135–143, New York, NY, USA, 2006. ACM. doi: 10.1145/1137856.1137879. URL <http://doi.acm.org/10.1145/1137856.1137879>.
- Hamerly, Greg. Making k-means even faster. In *SDM*, pp. 130–140, 2010. doi: http://www.siam.org/proceedings/datamining/2010/dm10_012_hamerlyg.pdf.
- Hamerly, Greg. baylorml. <https://github.com/BaylorCS/baylorml.git>, 2015.
- Kanungo, Tapas, Mount, D.M., Netanyahu, N.S., Piatko, C.D., Silverman, R., and Wu, A.Y. An efficient k-means clustering algorithm: analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892, Jul 2002. ISSN 0162-8828. doi: 10.1109/TPAMI.2002.1017616.
- Low, Yucheng, Gonzalez, Joseph, Kyrola, Aapo, Bickson, Danny, Guestrin, Carlos, and Hellerstein, Joseph M. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.
- Nister, David and Stewenius, Henrik. Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2, CVPR '06*, pp. 2161–2168, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2597-0. doi: 10.1109/CVPR.2006.264. URL <http://dx.doi.org/10.1109/CVPR.2006.264>.
- Orchard, M.T. A fast nearest-neighbor search algorithm. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pp. 2297–2300 vol.4, Apr 1991. doi: 10.1109/ICASSP.1991.150755.
- Philbin, J., Chum, O., Isard, M., Sivic, J., and Zisserman, A. Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2007*.
- Phillips, S.J. Acceleration of k-means and related clustering algorithms. volume 2409 of *Lecture Notes in Computer Science*. Springer, 2002.
- Sculley, D. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web, WWW '10*, pp. 1177–1178, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772862. URL <http://doi.acm.org/10.1145/1772690.1772862>.
- Vedaldi, A. and Fulkerson, B. VLFeat: An open and portable library of computer vision algorithms. <http://www.vlfeat.org/>, 2008.
- Wu, Xindong, Kumar, Vipin, Quinlan, J. Ross, Ghosh, Joydeep, Yang, Qiang, Motoda, Hiroshi, McLachlan, Geoffrey, Ng, Angus, Liu, Bing, Yu, Philip, Zhou, Zhi-Hua, Steinbach, Michael, Hand, David, and Steinberg, Dan. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, January 2008. ISSN 0219-1377. URL <http://dx.doi.org/10.1007/s10115-007-0114-2>.
- Xianyi, Zhang. OpenBLAS. <http://www.openblas.net>, 2016.