Exact Acceleration of K-Means++ and K-Means‖

Edward Raff
Booz Allen Hamilton
University of Maryland, Baltimore County
raff_edward@bah.com, raff.edward@umbc.edu

Abstract

K-Means++ and its distributed variant K-Means‖ have become de facto tools for selecting the initial seeds of K-means. While alternatives have been developed, the effectiveness, ease of implementation, and theoretical grounding of the K-means++ and ‖ methods have made them difficult to "best" from a holistic perspective. By considering the limited opportunities within seed selection to perform pruning, we develop specialized triangle inequality pruning strategies and a dynamic priority queue to show the first acceleration of K-Means++ and K-Means‖ that is faster in run-time while being algorithmicly equivalent. For both algorithms we are able to reduce distance computations by over 500×. For K-means++ this results in up to a 17× speedup in run-time and a 551× speedup for K-means‖. We achieve this with simple, but carefully chosen, modifications to known techniques which makes it easy to integrate our approach into existing implementations of these algorithms.

1 Introduction

Before one can run the K-means algorithm, a prerequisite step is needed to select the initial K-*seeds* to use as the initial estimate of the means. This seed selection step is critical to obtaining high quality results with the K-means algorithm. Selecting better initial centers $m_1, \ldots, m_K$ can improve the quality of the final K-means clustering. A major step in developing better seed selection was the K-means++ algorithm. This was the first to show that the seeds it finds are log-optimal in expectation for solving the K-means problem [Arthur and Vassilvitskii, 2007]. For a dataset with n items K-means++ requires $O(nK)$ distance computations. If P processors are available K-means++ can be done in $O(nK/P)$. However, the amount of communication overhead to do K-means in parallel is significant. To remedy this, Bahmani *et al.* [2012] introduced K-means‖ which retains the $O(nK/P)$ complexity and performs a constant factor more distance computations to significantly reduce the communication overhead while still yielding the same log-optimal results [Bachem *et al.*, 2017]. When working in a distributed environment, where communication must occur over the network, this can lead to large reductions in run-time [Bahmani *et al.*, 2012].

The cost of K-means++ has long been recognized as being an expensive but necessary step for better results, with little progress on improvement. Modern accelerated versions of K-means clustering perform as few as 1.2 total iterations of the dataset [Ryšavý and Hamerly, 2016], making K-means++ seed selection take up to 44% of all distance computations. Outside of exact K-means clustering, faster seed selection can help improve stochastic variants of K-means [Bottou and Bengio, 1995; Sculley, 2010] and is useful for applications like corset construction [Bachem *et al.*, 2015], change detection [Raff *et al.*, 2020], tensor algorithms [Jegelka *et al.*, 2009], clustering with Bergman divergences [Nock *et al.*, 2008], and Jensen divergences [Nielsen and Nock, 2015]. Applications with large K have in particular been neglected, even though $K \geq 20,000$ is useful for scaling kernel methods [Si *et al.*, 2017].

In this work, we seek to accelerate the original K-means++ and K-means‖ algorithms so that we may obtain the same provably good results in less time without compromising on *any* of the desirable qualities of K-means++ or K-means‖. We will review work related to our own in § 2. Since the bottlenecks and approach to accelerating these two algorithms are different we will review their details and our approach to accelerating them sequentially. In respect to K-means++ in § 3, we show how simple application of the triangle inequality plus a novel dynamic priority queue allows us to avoid redundant computations and keep the cost of sampling new means low. In § 4 we address K-means‖ and develop a new *NearestInRange* query that allows us to successfully use a metric index to prune distance computations even though it is restricted to corpora normally too small to be useful with structures like KD-trees. We then perform empirical evaluation of our modifications in § 5 over a larger set of corpora with more diverse properties covering $K \in [32, 4096]$. In doing so, we observe that our accelerated algorithms succeed in requiring either the same or less time across all datasets and all values of K, making it a Pareto improvement. Finally, we will conclude in § 6.

2 Related Work

Many prior works have looked at using the triangle inequality, $d(a, b) + d(b, c) \geq d(a, c)$, to accelerate the K-means algorithm. While the first work along this line was done by Phillips [2002], it was first successfully popular-

ized by Elkan [2003]. Since then, several works have attempted to build faster K-means clustering algorithms with better incorporation or tighter bounds developed through use of the triangle inequality [Hamerly, 2010; Ding *et al.*, 2015; Newling and Fleuret, 2016]. Despite the heavy use of the triangle inequality to accelerate K-means clustering, we are aware of no prior works that apply it to the seed selection step of K-means++ and K-means‖. We belive this is largely because these methods can not accelerate the first iteration of K-means, as they rely on the first iteration's result to accelerate subsequent iterations. Since K-means++ is effectively a single iteration of K-means, their approaches can not be directly applied to the seed selection step.

In our work to accelerate K-means‖ using metric index structures a similar historical theme emerges. Prior works have looked at using index structures like KD-trees [Pelleg and Moore, 1999] and Cover-trees [Curtin, 2017] to accelerate the K-means clustering algorithm, but did not look at the seed selection step. Similarly we will use a metric indices to accelerate K-means‖, but we will develop an enhanced nearest neighbor query that considers a maximum range to meaningfully prune even when using small values of K.

Most work we are aware of focuses on extending or utilizing the K-means++ algorithm with few significant results on improving it. The most significant in this regard is the AFK-MC[Bachem *et al.*, 2016a] algorithm and its predecessor K-MC [Bachem *et al.*, 2016b]. Both can obtain initial seeds with the same quality as K-means++ with less distance computations but scale as $O(n/P + mK^2)$, where m is a budget factor. This makes them less effective when a large number of CPUs P is available or when K is large. Neither work factored in actual run-time. [Newling and Fleuret, 2017] showed that these implementations are actually 3.3× *slower* when overheads are factored in. We consider run-time in our own work to show that our improvements materialize in practice.

3 Accelerating K-Means++

We start with the K-means++ algorithm where we present detailed pseudo-code in Algorithm 1. We detail the method and how it works when each data point x_i has with it an associated weight w_i, as this is required later on. The algorithm begins by selecting an initial seed at random, and then assigning a new weight β_i to each data point x_i, based on the squared distance of x_i to the closest existing seed. At each iteration, we select a new seed to the set based on these weights and return once we have k total seeds. This requires k iterations through the dataset or size n resulting in $O(n \cdot k)$ distance computations. Note that we cache the distance between each point x_i and it's closest mean into the variable α_i. We will maintain this notation throughout the paper and use α_i as shorthand.

The first step toward improving the K-means++ algorithm is to filter out redundant distance computations. To do this, we note that at each iteration we compare the distance of each point x_i to the newest mean m_k against the previous closest mean m_j, where $1 \leq j < k$. That is, we need to determine if $d(x_i, m_k) < d(x_i, m_j)$. To do this, we can use Lemma 1 as introduced and proven by Elkan [2003],

Algorithm 1 K-Means++

Require: Desired number of seeds K, data points $x_1, \ldots, x_n$, data weights $w_1, \ldots, w_n$ such that $w_i \geq 0$

1: $\beta_i \leftarrow w_i / \sum_{z=1}^{n} w_z, \forall i \in [1, n]$
2: $m_1 \leftarrow x_i$, where i is selected with probability β_i
3: $k \leftarrow 1, \boldsymbol{\alpha} \leftarrow \vec{\infty}$
4: **while** $k < K$ **do**
5: **for** $i \in [1, n]$ **do**
6: $\alpha_i \leftarrow \min(\alpha_i, d(m_k, x_i))$
7: **for** $i \in [1, n]$ **do**
8: $\beta_i \leftarrow w_i \cdot \alpha_i^2 / \left(\sum_{z=1}^{n} w_z \cdot \alpha_z^2\right)$
9: $k \leftarrow k + 1$
10: $m_k \leftarrow x_i$, where i is selected with probability β_i
11: **return** initial means $m_1, \ldots, m_K$

Lemma 1. *Let x be a point and let b and c be centers. If $d(b, c) \geq 2d(x, b)$ then $d(x, c) \geq d(x, b)$*

3.1 Applying the Triangle Inequality

We can use the distance between m_p and m_j to determine if computing $d(x_i, m_k)$ is a fruitless effort by checking if $d(m_j, m_k) > d(x_i, m_j)$. This is already available in the form of α_i as presented in Algorithm 1. We then only need to compute $d(m_j, m_k) \forall j < k$, of which there is intrinsically less than k unique values at each iteration. Thus, we can compute $\gamma_j = d(m_j, m_k)$ once at the start of each loop, and we can re-use these k values for all $n - k$ distance comparisons.

Applying this bound we can avoid many redundant computations. As there are still K total iterations to select K means, each iteration will perform k comparisons to previous means and $n - k$, we get at most n distance comparisons per iteration making the worst case still $O(nk)$ distance computations for the K-means++ algorithm.

3.2 Avoiding Subnomral Slowdowns

A non-trivial cost exists in lines 7-10 of Algorithm 1 where we must compute the probability of selecting each point as the next mean and then perform the selection. This requires at least $3 \cdot n$ floating point multiplications which can be a bottleneck in low dimensional problems. This can be exasperated because squared distance to the closest center α_i^2 naturally becomes very small as k increases resulting in subnormalized floating point values. Subnormals (also called denormal) attempt to extend the precision of IEEE floats, but can cause 100× slowdowns in computation [Dooley and Kale, 2006]. Depending on hardware, subnormals can also interfere with pipelining behavior and out-of-order execution, making a single subnormal computation highly detrimental to performance [Fog, 2016]. This is particularly problematic because pruning based on the triangle inequality works best on low dimensional problems, and the normalization step prevents us from realizing speedups in terms of total run-time.

To circumvent this bottleneck, we develop a simple approach to create a *dynamic* priority queue that allows us to sample the next mean accurately without having to interact with most of the samples per iteration. We start with the elegant sampling without replacement strategy introduced

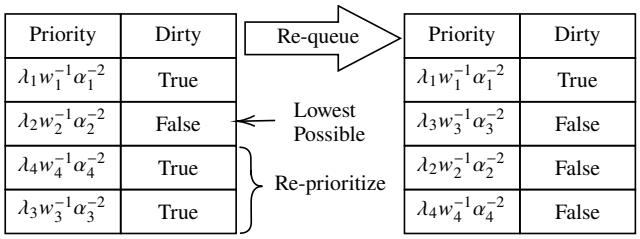

Figure 1: Example of priority re-queueing strategy for $n = 4$ items. Initially, it is not clear if items 2, 3, or 4 are the next to sample. All dirty items are removed from the queue until we reach a clean item and then re-inserted after fixing their priorities. We do not need to consider any item after the first clean item.

by Efraimidis and Spirakis [2006]. Given n items $1, \ldots, n$ with weighted probabilities $w_1, \ldots, w_n$, it works by assigning each item i a priority $\lambda_i w_i^{-1}$ where λ_i is sampled from the Exponential distribution with $\lambda = 1$ (i.e., $\lambda_i \sim \text{Exponential}(1)$). To select K items without replacement, one selects the K values with highest priority (smallest $\lambda_i w_i^{-1}$ values). This can normally be done with the quick-select algorithm in $O(n)$ time.

For K-means++ seeding we want to instead use the priority $\lambda_i w_i^{-1}\alpha_i^{-2}$ in order to produce random samples. The term $w_i^{-1}\alpha_i^{-2}$ acts as the weight for datum i being selected, and it is a combination of the original relative weight of the datum w_i and the squared distance to the nearest seed α_i^2. At the start we sample $\lambda_i \sim \text{Exponential}(1)$ once. During each round, we update all α_i values and leave λ_i fixed. It is trivial to see that this does not alter the expectation of any point being selected conditioned on the point i already being removed. This is because all λ_i are sampled independently, and so the removal of any λ_i does not impact the relative weights of any other point. Thus, we can use the weighted sampling without replacement strategy of Efraimidis and Spirakis [2006] to select the seeds. We performed a sanity check by implementing this naive approach and making no other changes. This resulted in the same quality solutions over many trials with the same statistical mean and variance.

At first glance, this strategy obtains no benefit as the value of α_i will change on each iteration. Each value of α_i changing means that the relative ordering of all remaining priorities $\lambda_i w_i^{-1}\alpha_i^{-2}$ will also change. This requires a full quick-select run on each iteration to discover the new maximum priority item. However, we note that α_i can only decrease with each iteration, and thus the priority of any given sample either remains constant or decreases. Our first contribution is the realization that this property can be exploited to reduce the cost of sampling so that only a subset of priorities need to be considered to sample the next point.

We can instead create a priority queue using a standard binary heap to select the next smallest value of $\lambda_i w_i^{-1}\alpha_i^{-2}$ and maintain a marker if the priority of an item i has become *dirty*. An item is dirty if and only if the item has a higher priority than it actually should. If there is a clean item z in the queue, then all items with a lower apparent priority than z must have a true priority that is still lower than z. Thus, we need only fix the priority of items higher than z.

See Figure 1 for an example of this queue for a dataset of $n = 4$ items. Item 2 is clean, and all items with a higher priority (3 and 4) are dirty. That means item 2 has the lowest possible priority that *could* be the next true sample because it is possible the values of items 3 and 4 will become larger (read, lower priority) once the updated values of α_3 and α_4 are computed. Thus, we can remove all items in the queue until we reach item 2 and then re-insert them into the queue with their correct priorities. In this hypothetical example, item 4 still had a lower priority after updating, and so will become the next mean when we them remove it from the queue. Item 1 occurred after item 2 because it had a lower priority. Even though item 1 was dirty, we did not need to consider it because its priority can only decrease once α_1 is updated. Because Item 2 was clean, its priority will not change, and there is no possibility of item 1 being selected.

3.3 Accelerated K-Means++

Algorithm 2 Our Accelerated K-Means++

Require: Desired number of seeds K, data points $x_1, \ldots, x_n$, data weights $w_1, \ldots, w_n$ such that $w_i \geq 0$

1: $\lambda_i \sim \text{Exponential}(1), \forall i \in [1, n]$
2: Priority Queue Q with each index i given priority λ_i / w_i
3: $\text{dirty}_i \leftarrow \text{False}$
4: $m_1 \leftarrow x_{Q.\text{POP}()}$
5: $\boldsymbol{\alpha} \leftarrow \vec{\infty}, k \leftarrow 1, \phi_i \leftarrow 0$
6: **for** $k \in [1, K)$ **do** ▷ For each new center k
7: **for** $j \in [1, k)$ **do** ▷ Get distance to previous centers
8: $\gamma_j \leftarrow d(m_k, m_j)$
9: **for** $i \in [1, n]$ **do**
10: **if** $\frac{1}{2}\gamma_{\phi_i} \geq \alpha_i$ **then**
11: **continue** ▷ Pruned by Lemma 1
12: **if** $d(m_k, x_i) < \alpha_i$ **then**
13: $\alpha_i \leftarrow d(m_k, x_i), \phi_i \leftarrow k$
14: $\text{dirty}_i \leftarrow \text{True}$ ▷ Priority may now be too high
15: Create new stack S
16: **while** $\text{dirty}_{Q.\text{PEEK}()}$ **do** ▷ All items that *could* be selected
17: $i \leftarrow Q.\text{POP}()$
18: $S.\text{PUSH}(i)$
19: **for** $i \in S$ **do** ▷ Update true priority
20: $Q.\text{PUSH}(i, \lambda_i / (w_i \cdot \alpha_i^2))$
21: $\text{dirty}_i \leftarrow \text{False}$
22: $m_k \leftarrow x_{Q.\text{POP}()}$ ▷ Select new mean by clean top priority
23: **return** initial means $m_1, \ldots, m_K$

The final algorithm that performs the accelerated computation is given in Algorithm 2. Lines 7-11 take care to avoid redundant distance computations, and lines 14-21 ensure that the dynamic priority queue allows us to select the next mean without considering all $n - k$ remaining candidates. Combined, we are able to regularly gain reductions both in terms of total time taken as well as the number of distance computations required. Through the use of our dynamic priority queue we find that we regularly consider less than 1% of total remaining $n - k$ items. This is important when we work with low-dimension datasets. When the dimension is very

small (e.g., $d = 2$ for longitude/latitude data is a common use case), there is little computational cost in the distance computations themselves, and so much of the bottleneck in runtime is contained within the sampling process. Our dynamic queue avoids this bottleneck allowing us to realize the benefits of reduced distance computations.

4 Accelerating K-Means||

Now we turn our attention to the K-means|| algorithm detailed in Algorithm 3. While K-means|| requires more distance computations, it is preferred in distributed environments because it requires less communication which is a significant bottleneck for K-means++ [Bahmani *et al.*, 2012]. It works by reducing the K rounds of communication to a fixed number of $R \ll K$ rounds, yet still obtains the log-optimal results of K-means++ [Bachem *et al.*, 2017]. In each of the rounds, ℓ new means are sampled based on the weighted unnormalized probability $\ell w_i \alpha_i^2$. With the standard defaults of $R = 5$ and $\ell = 2K$, we end up with an expected $R2K > K$ total means. These $R \cdot \ell$ potential means are weighted by the number of points that they are closest to and then passed to the K-means++ algorithm to reduce them to a final set of K means, which produces the final result. Note this last step requires $O(K^2)$ distance computations when naively using Algorithm 1, making it necessary to accelerate the K-means++ algorithm in order to effectively accelerate K-means|| for datasets with large K.

Algorithm 3 K-Means||

Require: Desired number of seeds K, $x_1, \ldots, x_n$, data weights $w_1, \ldots, w_n$, rounds R, oversampling factor ℓ

1: $\beta_i \leftarrow w_i / \sum_{j=1}^{n} w_j, \forall i \in [1, n]$
2: $c_1 \leftarrow x_i$, where i is selected with probability β_i
3: $k \leftarrow 1$, $k_{prev} \leftarrow 0$, $\boldsymbol{\alpha} \leftarrow \vec{\infty}$
4: **for** $r \in [1, R]$ **do**
5: **for** $i \in [1, n]$ **do**
6: **for** $j \in (k_{prev}, k]$ **do**
7: $\alpha_i \leftarrow \min\left(\alpha_i, d\left(c_j, x_i\right)\right)$
8: $k_{prev} \leftarrow k$
9: $Z \leftarrow \sum_{i=1}^{n} w_i \cdot \alpha_i^2$
10: **for** $i \in [1, n]$ **do**
11: **if** $p \sim \text{Ber}(\min(1, \ell \cdot w_i \cdot \alpha_i^2 / z))$ is true **then**
12: $k \leftarrow k + 1$, $c_k \leftarrow x_i$, $\alpha_i \leftarrow 0$
13: Let $w'_i \leftarrow \sum_{j=1}^{n} w_j \cdot \mathbb{1}[d(c_i, x_j) = \alpha_j]$ ▷ Weight set to number of points closest to center c_i
14: **return** K-Means++$(K, c_1, \ldots, c_k, w'_1, \ldots, w'_k)$ ▷ Run Algorithm 1

Since $K < R \cdot \ell \ll n$, the final step of running K-means++ is not overbearing to run on a single compute node, and the sampling procedure is no longer a bottleneck that requires subversion. In a distributed setting, the ℓ new means selected are broadcast out to all worker nodes, which is possible because $\ell \ll n$, and thus requires limited communication overhead. However, the ability to use the triangle inequality becomes less obvious. Using the same approach as before, similar to Elkan [2003], would require $O(K^2)$ pairwise distances computations between the new and old means, and more book-keeping overhead that would reduce the effectiveness of avoiding distance computations.

Another strategy uses an algorithm like the Cover-Tree that accelerates nearest neighbor searches and supports the removal of data points from the index [Beygelzimer *et al.*, 2006; Izbicki and Shelton, 2015]. Then, we could perform an all-points nearest neighbor search [Curtin *et al.*, 2013]. However, we are unaware of any approach that has produced a distributed cover-tree algorithm that would not run into the same communication overheads that prevents the standard K-means++ from working in this scenario. As such, it does not appear to be a worth while strategy.

4.1 Nearest In Range Queries

Another approach would be to fit an index structure $\mathscr{C}$ to only the ℓ new points, and for each non-mean x_i find its nearest potentially new assignment by querying $\mathscr{C}$. Since ℓ is $O(K)$ this is too small a dataset for pruning to be effective with current methods.

To remedy this, we note that we have additional information available to perform the search. The value α_i which indicates the distance of point x_i to its closest current mean. As such, we introduce a *NearestInRange* search that returns the nearest neighbor to a query point q against an index $\mathscr{C}$ if it is within a radius of r to the query. Since most points x_i will not change ownership in a given iteration, a NearestInRange search could be able to prune out the entire search tree, and it will increase its effectiveness even if K is small.

To do this, we use the Vantage Point tree (VP) algorithm [Yianilos, 1993] because it is fast to construct, has low overhead which makes it competitive with other algorithms such as KD-trees and Cover-trees [Raff and Nicholas, 2018], and simple to augment with our new NearestInRange search. The pseudo-code for the standard VP search is given in Algorithm 4, where *GetChild*, *Search*, and *Best* are auxiliary functions used by the *Nearest* function to implement a standard nearest neighbor search. The VP has a left and right child, and it uses a value τ to keep track of the distance to the nearest neighbor found. It also maintains two pairs of bounds, $near_{low}, near_{high}$ indicating the shortest and farthest distance to the points in the left child and far_{low}, far_{high} do the same for the right child.

A standard Nearest Neighbor search calls the Nearest function with $\tau = \infty$, and the bound is updated as the search progresses when it fails to prune a branch. Our contribution is simple. The *NearestInRange* function instead sets $\tau = r$, the minimum viable radius. It is easy to verify that this can only monotonically improve the pruning rate of each search. Since τ bounds the distance to the nearest neighbor, and we know from the $\boldsymbol{\alpha}$ values an upper-bound on the distance to the nearest neighbor, the modification remains correct. The rest of the algorithm remains unaltered, and can simply terminate the search faster due to a meaningful initial bound.

Thus, to build an accelerated K-means|| we build an index $\mathscr{C}$ on the newly selected means. We do comparisons against that filtered with our *NearestInRange* search, as detailed in Algorithm 5. For the first iteration, the loop on lines 6-10 will be fast with only c_1 to determine the initial distribution,

Algorithm 4 Nearest Neighbor Search in VP Tree

function GETCHILD(low)
  if $low = true$ **then**
    return left child
  return right child ▷ Else, return other
function SEARCH(r, τ, low)
  if $low = true$ **then**
    $a \leftarrow near_{low}, b \leftarrow near_{high}$
  else
    $a \leftarrow far_{low}, b \leftarrow far_{high}$
  return $a - \tau < r < b + \tau$ ▷ i.e., is this True or False?
function BEST(τ, τ', ID, ID')
  if $\tau < \tau'$ **then**
    return τ, ID
  return τ', ID' ▷ Else, return other
function NEAREST(q, τ, ID)
  $r \leftarrow d(p, q)$
  $\tau, ID \leftarrow$ BEST(τ, r, ID, p)
  $m \leftarrow \frac{near_{high}+far_{low}}{2}$
  $lf \leftarrow r < m$ ▷ True/False, search near/left child first?
  if SEARCH(r, τ, lf) **then**
    $\tau', ID' \leftarrow$GETCHILD(lf).NEAREST(q, τ, ID)
    $\tau, ID \leftarrow$ BEST(τ, τ', ID, ID')
  if SEARCH($r, \tau, \neg lf$) **then**
    $\tau', ID' \leftarrow$GETCHILD($\neg lf$).NEAREST(q, τ, ID)
    $\tau, ID \leftarrow$ BEST(τ, τ', ID, ID')
  return τ, ID
function NEARESTINRANGE($q, maxRange$)
  return NEAREST($q, maxRange, -1$) ▷ This simple function, used in-place of Nearest, is our contribution.

Algorithm 5 Our Accelerated K-Means||

Require: Desired number of seeds K, $x_1, \ldots, x_n$, data weights $w_1, \ldots, w_n$, rounds R, oversampling factor ℓ
$\beta_i \leftarrow w_i / \sum_{j=1}^{n} w_j, \forall i \in [1, n]$
$c_1 \leftarrow x_i$, where i is selected with probability β_i
$\boldsymbol{\alpha} \leftarrow \vec{\infty}$, $k_{prev} \leftarrow 0$, $k \leftarrow 1$
for $r \in [1, R]$ **do**
  $\mathscr{C} \leftarrow$ new index built from $\{c_{k_{prev}}, \ldots, c_k\}$
  for $i \in [1, n]$ **do**
    $j \leftarrow \mathscr{C}$.NEARESTINRANGE(x_i, α_i)
    if $j \geq 0$ **then**
      $\alpha_i \leftarrow d(c_j, x_i)$
  $k_{prev} \leftarrow k$, $Z \leftarrow \sum_{i=1}^{n} w_i \cdot \alpha_i^2$
  for $i \in [1, n]$ **do**
    if $p \sim \text{Ber}(\min(1, \ell \cdot w_i \cdot \alpha_i^2 / Z))$ is true **then**
      $k \leftarrow k + 1$, $c_k \leftarrow x_i$, $\alpha_i \leftarrow 0$
Let $w'_i \leftarrow \sum_{j=1}^{n} w_j \cdot \mathbb{1}[d(c_i, x_j) = \alpha_j]$ ▷ Weight set to number of points closest to center c_i
return K-Means++($K, c_1, \ldots, c_k, w'_1, \ldots, w'_k$) ▷ Run Algorithm 2

and on every subsequent round we have a meaningful value of α_i that can be used to accelerate the search. If none of the ℓ new candidates $c_{k_{prev}}, \ldots, c_k$ are within a distance of α_i to each point x_i, then the *NearestInRange* function will return a negative index which can be skipped.

In addition, we use our accelerated K-means++ algorithm Algorithm 2 in the final step rather than the standard algorithm. This allows us to accelerate all parts of the K-means|| method while also keeping the simplicity and low communication cost of the original design. The Vantage Point tree is a small index since it is built upon a small dataset of ℓ points, and the index can be sent to every worker node in a cluster in the exact same manner.

5 Experimental Results

Now that we have detailed the methods by which we accelerate the K-means++ and K-means|| algorithms, we will evaluate their effectiveness. The two measures we are concerned with are the following: 1) reducing the total number of distance computations and 2) the total run-time spent. Measuring distance computations gives us an upper-bound on potential effectiveness of our algorithm, and allows us to compare approaches in an implementation and hardware independent manner. Measuring the run-time gives us information about the ultimate goal, which is to reduce the time it takes to obtain K seeds. However, it is sensitive to the hardware in use, the language the approach is implemented in, and the relative skills of program authors. For this work we used the JSAT library [Raff, 2017]. The K-means++ algorithm was provided by this framework, and we implemented the K-means|| and accelerated versions of both algorithms using JSAT. This way all comparisons with respect to run-time and the K-means++ and || algorithms presented are directly comparable. Our implementations have been contributed to the JSAT library for public use.

Prior works that have investigated alternatives to K-means++ have generally explored only a few datasets with $D < 100$ features and less than 4 values of K, sometimes testing only one value of K per dataset [Bachem *et al.*, 2016b]. For example, while MNIST is regularly tested in seed selection, it is usually projected down to 50 dimensions first [Hamerly, 2010] due to being difficult to accelerate.

Since our goal is to produce accelerated versions of these algorithms that are uniformly better, we attempt to test over a wide selection of reasonable scenarios. In Table 1 we show the 11 datasets we use, with $D \in [3, 780]$, and n covering four orders of magnitude. We will test $K \in [32, 4096]$ covering each power of two so that we may understand the behavior as K changes and to make sure we produce an improvement even when K is small. To the best of our knowledge, this is a larger number of datasets, range and values of K, and range and values of D to be tested compared to prior work[1].

Unless stated otherwise, all experiments were done with a single CPU core from an iMac with a 3.5 GHz Intel i5 CPU with 64 GB of RAM. The phishing dataset is only tested up

[1]We are aware of no prior work in this space that has considered $D > 1024$, where pruning methods are unlikely to succeed due to the curse of dimensionality. We consider this reasonable and beyond scope, as such scenarios are usually sparse and best handled by topic models like LDA.

Dataset	n	D
Phishing	11055	68
cod-rna	59535	8
MNIST	60000	780
aloi	108000	128
Range-Queries	200000	8
Skin/NoSkin	245057	3
covertype	581012	54
SUSY	5000000	18
Activity Rec.	33741500	5
HIGGS	11000000	28
Web[2]	45811883	5

Table 1: Datasets used. Left is the dataset, ordered by number of samples (n). Right most column indicates the number of features D.

to $K = 2048$, because at $K = 4096$ we would be selecting over 1/4 of the dataset as means, at which point the purpose of K-means++ style seeding is being defeated by selecting too large a portion of the corpus. All results are averaged over 5 runs, and took four months to complete in our compute environment.

5.1 K-Means++ Results

We start with the K-means++ results with the reduction in distance computations shown in Figure 2. In the worst case for $K = 32$ on the MNIST dataset, we still have to do 98% of the distance computations as the standard algorithm, but this improved to only 63% by $K = 4096$. The best case is observed with the Web dataset, starting out with only 15% of the distance computations at $K = 32$ and only 0.1% by $K = 4096$, a 739× improvement.

Across all the datasets, we see that the factor reduction in distance computations is a monotonic improvement for K-means++. We never see any case where our accelerated approach performs more distance computations than the naive approach. This confirms our decision to do an extra $k - 1$ distance computation between the newest mean m_k and the previous means $m_1, \ldots, m_{k-1}$.

As we noted in the design of our accelerated variant, we must avoid over-emphasising the performance of just reduced distance computations as the cost of re-normalizing the distribution to sample the next mean is a non-trivial cost. This is especially true when we are able to reduce the distance computations by $\geq 16\times$ for several of our datasets. The results showing the run-time reduction are presented in Figure 3.

In all cases, our accelerated version of K-means++ is always faster than the standard algorithm. As expected, MNIST has the lowest speedup based on the number of distance computations avoided. At $K = 32$ we achieved only a 3.4% reduction in time but was 1.5× faster by $K = 4096$.

Dynamic Priority Impact

Since the normalization step is non-trivial, especially when D is small, we see that the actual speedup in run-time is not as strongly correlated with the dimension D. The Covertype dataset ($D = 54$) had the 4th largest reduction in distance computations, but it had the largest reduction in run-time with a 17× improvement at $K = 4096$. Our ability to still obtain

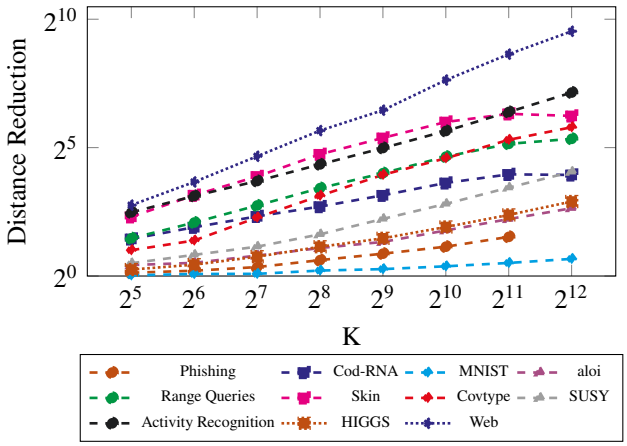

Figure 2: Factor reduction in distance computations for our accelerated K-means++ algorithm compared to original.

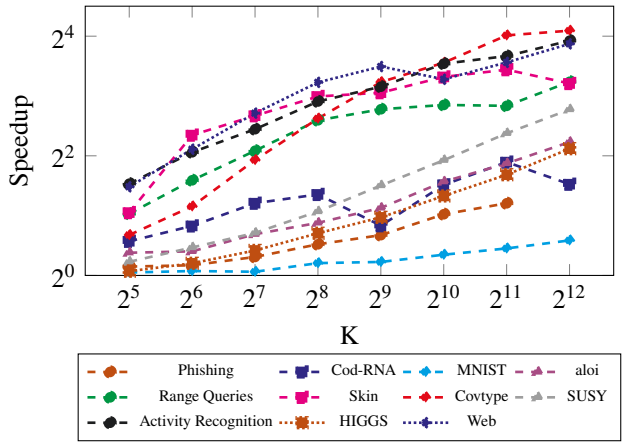

Figure 3: Run-time Speedup for our accelerated K-means++ algorithm compared to the standard algorithm.

real speedups on these datasets is because our dynamic priority queue allows us to consider only a small subset of the dataset to accurately select the next weighted random mean. This can be seen in Figure 4, where a subset of the datasets are shown with the fraction of the corpus examined on the y-axis. As the datasets get larger our dynamic queue generally becomes more effective, thus reducing the number of points that need to be checked to $\leq 1\%$.

To confirm that our dynamic priority queue's results are meaningful, we perform an ablation of Algorithm 2 where the dynamic priority queue on lines 15-21 are replaced with the standard sampling code from Algorithm 1. We run both versions and record the speedup when our dynamic queue is used in Table 2 for $K = 4096$. Here we can see that with the exception of the cod-rna dataset, where there is a $< 2\%$ slowdown (on the fastest dataset to run), our approach gives a $5\% - 231\%$ speedup in all other cases with a median improvement of 20%. We also note that for all $K < 4096$ we still observe benefits to our queue, but the variance does increase to the degree of speedup. We did not observe any

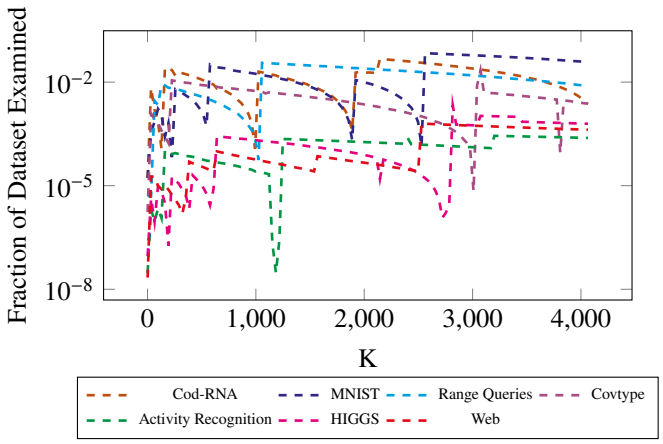

Figure 4: The fraction of the remaining candidates that need to be examined (y-axis, log scale) to select the k'th mean (x-axis, linear scale) using our dynamic priority queue.

Dataset	Speedup
cod-rna	0.983
Phishing	2.313
MNIST	1.059
aloi	1.059
Range-Queries	1.342
Skin/NoSkin	1.217
covtype	1.877
SUSY	1.259
Activity Recognition	1.207
HIGGS	1.279
Web	1.725

Table 2: Ablation testing of speedup from using our new dynamic priority queue to perform seed selection at every iteration. Positive values indicate faster results using our dynamic queue, where our pruning from Algorithm 2 was used with/without the dynamic queue.

performance regressions larger than 3% in extended testing.

5.2 K-Means|| Results

In Figure 5 we show the factor reduction in distance computations, which mirrors the overal trends of Figure 2. The results have improved by an additional $\approx 2-4\times$ with a 579× reduction in distance computations on the Activity Recognition dataset. The MNIST dataset still had the least improvement, but still obtained a more significant 88% reduction in distance computations at $K = 32$.

The $\approx 4\times$ improvement in distance computations also carries over to the total run-time, as shown in Figure 6. We observe a more consistent behavior because the cost of normalizing and sampling the new means is reduced to only $R = 5$ rounds of sampling. Where our accelerated K-means++ had the relative improvement drop significantly for small $D < 10$ datasets due to this overhead, our accelerated K-means|| algorithm sees the ordering remain relatively stable. For example, the Activity Recognition dataset enjoys the greatest reduction in distance computations as well as run-time, and the 579× reduction in distance computations closely matches the 551× reduction in run-time. The HIGGS dataset has the lowest improvement in run-time with a 1.02× speedup at $K = 32$ and 2.9× at $K = 4096$. We also note that the NearestInRange query provided an additional $1.5-4\times$ speedup in most cases, but was highly dependent on the dataset and value of K.

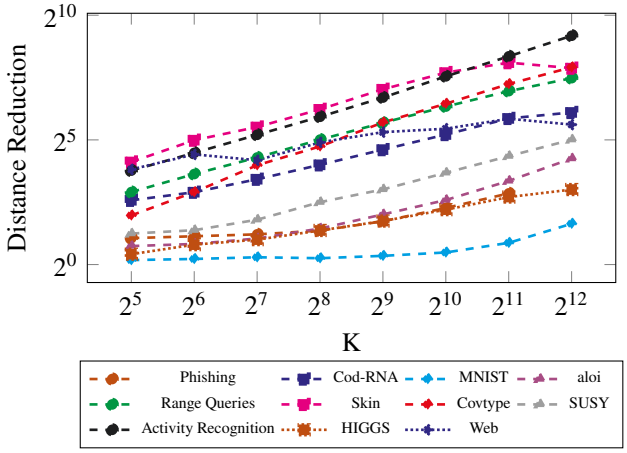

Figure 5: Factor Reduction in Distance Computations needed to perform K-means|| seed selection. Larger is better.

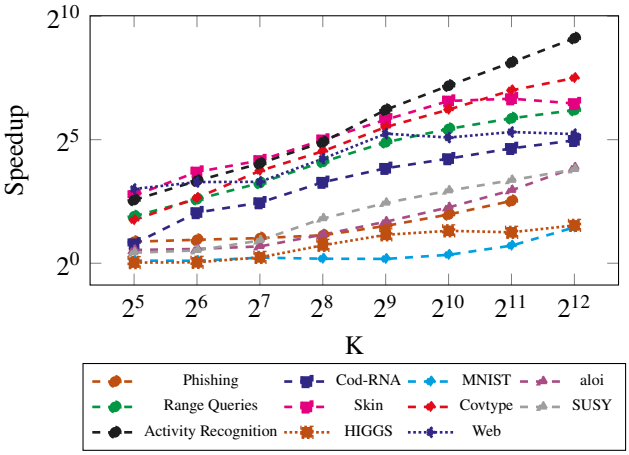

Figure 6: Run-time speedup for our accelerated K-means|| algorithm compared to the standard algorithm. Larger is better.

6 Conclusion

Leveraging simple modifications and a novel priority queue, we show the first method that delivers equal or better run-time in theory (less distance computations) and practice (less run-time). None of our changes impact the function of K-means++ or K-means||, allowing us to retain existing properties.

Acknowledgements

I would like to thank Ashley Klein, Drew Farris, and Frank Ferraro for reviewing early drafts and their valuable feedback.

References

[Arthur and Vassilvitskii, 2007] David Arthur and Sergei Vassilvitskii. k-means++: The Advantages of Careful Seeding. In *ACM-SIAM symposium on Discrete algorithms*, volume 8, pages 1027–1035, 2007.

[Bachem *et al.*, 2015] Olivier Bachem, Mario Lucic, and Andreas Krause. Coresets for Nonparametric Estimation - the Case of DP-Means. In *ICML*, volume 37, pages 209–217, 2015.

[Bachem *et al.*, 2016a] Olivier Bachem, Mario Lucic, Hamed Hassani, and Andreas Krause. Fast and Provably Good Seedings for k-Means. In *NeurIPS*, pages 55–63. 2016.

[Bachem *et al.*, 2016b] Olivier Bachem, Mario Lucic, S Hamed Hassani, and Andreas Krause. Approximate K-means++ in Sublinear Time. In *AAAI*, pages 1459–1467, 2016.

[Bachem *et al.*, 2017] Olivier Bachem, Mario Lucic, and Andreas Krause. Distributed and Provably Good Seedings for k-Means in Constant Rounds. In *ICML*, volume 70, pages 292–300, 2017.

[Bahmani *et al.*, 2012] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable K-means++. *VLDB Endowment*, 5(7):622–633, mar 2012.

[Beygelzimer *et al.*, 2006] Alina Beygelzimer, S Kakade, and John Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[Bottou and Bengio, 1995] Léon Bottou and Yoshua Bengio. Convergence Properties of the K-Means Algorithms. In *NeurIPS*, pages 585–592, 1995.

[Curtin *et al.*, 2013] Ryan R. Curtin, William B. March, Parikshit Ram, David V. Anderson, Alexander G. Gray, and Charles L. Isbell. Tree-Independent Dual-Tree Algorithms. In *ICML*, volume 28, pages 1435–1443, 2013.

[Curtin, 2017] Ryan R Curtin. A Dual-Tree Algorithm for Fast k -means Clustering With Large k. In *SDM*, pages 300–308. jun 2017.

[Ding *et al.*, 2015] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang K-means: A Drop-in Replacement of the Classic K-means with Consistent Speedup. In *ICML*, pages 579–587, 2015.

[Dooley and Kale, 2006] Isaac Dooley and Laxmikant Kale. Quantifying the Interference Caused by Subnormal Floating-Point Values. In *Proceedings of the Workshop on Operating System Interference in High Performance Applications*, sep 2006.

[Efraimidis and Spirakis, 2006] Pavlos S. Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97(5):181–185, mar 2006.

[Elkan, 2003] Charles Elkan. Using the Triangle Inequality to Accelerate k-Means. In *ICML*, pages 147–153, 2003.

[Fog, 2016] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. 2016.

[Hamerly, 2010] Greg Hamerly. Making k-means even faster. In *SDM*, pages 130–140, 2010.

[Izbicki and Shelton, 2015] Mike Izbicki and Christian R Shelton. Faster Cover Trees. In *ICML*, volume 37, 2015.

[Jegelka *et al.*, 2009] Stefanie Jegelka, Suvrit Sra, and Arindam Banerjee. Approximation Algorithms for Tensor Clustering. In *ATL*, pages 368–383, 2009.

[Newling and Fleuret, 2016] James Newling and François Fleuret. Fast k-means with accurate bounds. In *ICML*, pages 936–944, 2016.

[Newling and Fleuret, 2017] James Newling and François Fleuret. K-Medoids For K-Means Seeding. In *NeurIPS*, pages 5195–5203. 2017.

[Nielsen and Nock, 2015] Frank Nielsen and Richard Nock. Total Jensen divergences: Definition, properties and clustering. In *ICASSP*, pages 2016–2020, apr 2015.

[Nock *et al.*, 2008] Richard Nock, Panu Luosto, and Jyrki Kivinen. Mixed Bregman Clustering with Approximation Guarantees. In *ECMLPKDD*, pages 154–169, 2008.

[Pelleg and Moore, 1999] Dan Pelleg and Andrew Moore. Accelerating Exact K-means Algorithms with Geometric Reasoning. In *KDD*, pages 277–281, 1999.

[Phillips, 2002] Steven J. Phillips. Acceleration of K-Means and Related Clustering Algorithms. In *Proc. 4th Int. Workshop on Algorithm Engineering and Experiments (ALENEX 2002)*, pages 166–177, 2002.

[Raff and Nicholas, 2018] Edward Raff and Charles Nicholas. Toward Metric Indexes for Incremental Insertion and Querying. *arXiv*, 2018.

[Raff *et al.*, 2020] Edward Raff, Bobby Filar, and James Holt. Getting Passive Aggressive About False Positives: Patching Deployed Malware Detectors. In *2020 International Conference on Data Mining Workshops (ICDMW)*, pages 506–515. IEEE, nov 2020.

[Raff, 2017] Edward Raff. JSAT: Java Statistical Analysis Tool, a Library for Machine Learning. *JMLR*, 18(23):1–5, 2017.

[Ryšavý and Hamerly, 2016] Petr Ryšavý and Greg Hamerly. Geometric methods to accelerate k-means algorithms. In *SDM*, pages 324–332, jun 2016.

[Sculley, 2010] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th international conference on World wide web*, pages 1177–1178, 2010.

[Si *et al.*, 2017] Si Si, Cho-Jui Hsieh, and Inderjit S Dhillon. Memory Efficient Kernel Approximation. *JMLR*, 18(20):1–32, 2017.

[Yianilos, 1993] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete algorithms*, pages 311–321, 1993.