Dual-tree k-means with bounded iteration runtime

Ryan R. Curtin RYAN@RATML.ORG
School of Computational Science and Engineering,
Georgia Institute of Technology, Atlanta, GA 30332 USA

Abstract

k-means is a widely used clustering algorithm, but for k clusters and a dataset size of N, each iteration of Lloyd's algorithm costs $O(kN)$ time. Although there are existing techniques to accelerate single Lloyd iterations, none of these are tailored to the case of large k, which is increasingly common as dataset sizes grow. We propose a dual-tree algorithm that gives the *exact* same results as standard k-means; when using cover trees, we use adaptive analysis techniques to, under some assumptions, bound the single-iteration runtime of the algorithm as $O(N + k \log k)$. To our knowledge these are the first sub-$O(kN)$ bounds for exact Lloyd iterations. We then show that this theoretically favorable algorithm performs competitively in practice, especially for large N and k in low dimensions. Further, the algorithm is tree-independent, so any type of tree may be used.

1. Introduction

Of all the clustering algorithms in use today, among the simplest and most utilized is the venerated k-means clustering algorithm, usually implemented via Lloyd's algorithm: given a dataset S, repeat the following two steps (a 'Lloyd iteration') until the centroids of each of the k clusters converge:

1. Assign each point $p_i \in S$ to the cluster with nearest centroid.
2. Recalculate the centroids for each cluster using the assignments of each point in S.

Clearly, a simple implementation of this algorithm will take $O(kN)$ time where $N = |S|$. However, the number of iterations is not bounded unless the practitioner manually sets a maximum, and k-means is not guaranteed to converge to the global best clustering. Despite these shortcomings, in practice k-means tends to quickly converge to reasonable solutions. Even so, there is no shortage of techniques for improving the clusters k-means converges to: refinement of initial centroids (Bradley & Fayyad, 1998) and weighted sampling of initial centroids (Arthur & Vassilvitskii, 2007) are just two of many popular existing strategies.

There are also a number of methods for accelerating the runtime of a single iteration of k-means. In general, these ideas use the triangle inequality to prune work during the assignments step. Algorithms of this sort include the work of Pelleg and Moore (1999), Elkan (2003), Hamerly (2010), and Ding et al. (2015). However, the scaling of these algorithms can make them problematic for the case of large k and large N.

In this paper, we describe a dual-tree k-means algorithm tailored to the large k and large N case that outperforms all competing algorithms in that setting; this dual-tree algorithm also has bounded single-iteration runtime in some situations (see Section 6). This algorithm, which is our main contribution, has several appealing aspects:

- **Empirical efficiency**. In the large k and large N setting for which this algorithm is designed, it outperforms all other alternatives, and scales better to larger datasets. The algorithm is especially efficient in low dimensionality.
- **Runtime guarantees**. Using adaptive runtime analysis techniques, we bound the single-iteration runtime of our algorithm with respect to the intrinsic dimensionality of the centroids and data, when cover trees are used. This gives theoretical support for the use of our algorithm in large data settings. In addition, the bound is dependent on the intrinsic dimensionality, *not* the extrinsic dimensionality.
- **Generalizability**. We develop our algorithm using a tree-independent dual-tree algorithm abstraction (Curtin et al., 2013b); this means that our algorithm may be used with *any* type of valid tree. This includes not just *kd*-trees but also metric trees, cone trees, octrees, and others. Different trees may be suited to different types of data, and since our algorithm is general, one may use any type of tree as a plug-and-play parameter.

Submitted to the 33rd International Conference on Machine Learning (ICML '16).

Algorithm	Setup	Worst-case	Memory
naive	n/a	$O(kN)$	$O(k+N)$
blacklist	$O(N \log N)$	$O(kN)$	$O(k \log N + N)$
elkan	n/a	$O(k^2 + kN)$	$O(k^2 + kN)$
hamerly	n/a	$O(k^2 + kN)$	$O(k+N)$
yinyang	$O(k^2 + kN)$	$O(kN)$	$O(kN)$
dualtree	$O(N \log N)$	$O(k \log k + N)$[1]	$O(k+N)$

Table 1. Runtime and memory bounds for k-means algorithms.

- **Separation of concerns**. The abstraction we use to develop our algorithm allows us to focus on and formalize each of the pruning rules individually (Section 4). This aids understanding of the algorithm and eases insertion of future improvements and better pruning rules.

Section 2 shows the relevance of the large k case; then, in Section 3, we show that we can build a tree on the k clusters, and then a dual-tree algorithm (Curtin et al., 2013b) can be used to efficiently perform an exact single iteration of k-means clustering. Section 4 details the four pruning strategies used in our algorithm, and Section 5 introduces the algorithm itself. Sections 6 and 7 show the theoretical and empirical results for the algorithm, and finally Section 8 concludes the paper and paints directions for future improvements.

2. Scaling k-means

Although the original publications on k-means only applied the algorithm to a maximum dataset size of 760 points, the half-century of relentless progress since then has seen dataset sizes scale into billions. Due to its simplicity, though, k-means has remained relevant, and is still applied in many large-scale applications.

In cases where N scales but k remains small, a good choice of algorithm is a sampling algorithm, which will return an approximate clustering. One sampling technique, coresets, can produce good clusterings for n in the millions using several hundred or a few thousand points (Frahling & Sohler, 2008). However, for large k, the number of samples required to produce good clusterings can become prohibitive.

For large k, then, we turn to an alternative approach: accelerating exact Lloyd iterations. Existing techniques include the brute-force implementation, the *blacklist* algorithm (Pelleg & Moore, 1999), Elkan's algorithm (2003), and Hamerly's algorithm (2010), as well as the recent Yinyang k-means algorithm (Ding et al., 2015). The blacklist algorithm builds a kd-tree on the dataset and, while the tree is traversed, blacklists individual clusters that cannot be the closest cluster (the *owner*) of any descendant points of a node. Elkan's algorithm maintains an upper bound and a lower bound on the distance between each point and centroid; Hamerly's algorithm is a memory-efficient simplification of this technique. The Yinyang algorithm organizes the centroids into groups of about 10 (depending on algorithm parameters) using 5 iterations of k-means on the centroids followed by a single iteration of standard k-means on the points. Once groups are built, the Yinyang algorithm attempts to prune groups of centroids at a time using rules similar to Elkan and Hamerly's algorithms.

Of these algorithms, only Yinyang k-means considers centroids in groups at all, but it does not consider points in groups. On the other hand, the blacklist algorithm is the only algorithm that builds a tree on the points and is able to assign multiple points to a single cluster at once. So, although each algorithm has its own useful region, none of the four we have considered here are particularly suited to the case of large N **and** large k.

Table 1 shows setup costs, worst-case per-iteration runtimes, and memory usage of each of these algorithms as well as the proposed dual-tree algorithm[1]. The expected runtime of the blacklist algorithm is, under some assumptions, $O(k+k \log N+N)$ per iteration. The expected runtime of Hamerly's and Elkan's algorithm is $O(k^2+\alpha N)$ time, where α is the expected number of clusters visited by each point (in both Elkan and Hamerly's results, α seems to be small).

However, none of these algorithms are specifically tailored to the large k case, and the large k case is common. Pelleg and Moore (1999) report several hundred clusters in a subset of 800k objects from the SDSS dataset. Clusterings for n-body simulations on astronomical data often involve several thousand clusters (Kwon et al., 2010). Csurka et al. (2004) extract vocabularies from image sets using k-means with $k \sim 1000$. Coates et al. (2011) show that k-means can work surprisingly well for unsupervised feature learning for images, using k as large as 4000 on 50000 images. Also, in text mining, datasets can have up to

[1]The dual-tree algorithm worst-case runtime bound also depends on some assumptions on dataset-dependent constants. This is detailed further in Section 6.

18000 unique labels (Bengio et al., 2010). Can and Ozkarahan (1990) suggest that the number of clusters in text data is directly related to the size of the vocabulary, suggesting $k \sim mN/t$ where m is the vocabulary size, n is the number of documents, and t is the number of nonzero entries in the term matrix. Thus, it is important to have an algorithm with favorable scaling properties for both large k and N.

3. Tree-based algorithms

The blacklist algorithm is an example of a *single-tree algorithm*: one tree (the *reference tree*) is built on the dataset, and then that tree is traversed. This approach is applicable to a surprising variety of other problems, too (Bentley, 1975; Moore, 1999; Curtin et al., 2013c). Following the blacklist algorithm, then, it is only natural to build a tree on the data points. Tree-building is (generally) a one-time $O(N \log N)$ cost and for large N or k, the cost of tree building is often negligible compared to the time it takes to perform the clustering.

The speedup of the blacklist algorithm comes from the hierarchical nature of trees: during the algorithm, we may rule out a cluster centroid for *many points at once*. The same reason is responsible for the impressive speedups obtained for other single-tree algorithms, such as nearest neighbor search (Bentley, 1975; Liu et al., 2004). But for nearest neighbor search, the nearest neighbor is often required not just for a query point but instead a *query set*. This observation motivated the development of *dual-tree algorithms*, which also build a tree on the query set (the *query tree*) in order to share work across query points. Both trees are recursed in such a way that combinations of query nodes and reference nodes are visited. Pruning criteria are applied to these node combinations, and if a combination may be pruned, then the recursion does not continue in that direction.

This approach is applicable to k-means with large k: we may build a tree on the k cluster centroids, as well as a tree on the data points, and then we may rule out *many* centroids for *many* points at once.

A recent result generalizes the class of dual-tree algorithms, simplifying their expression and development (Curtin et al., 2013b). Any dual-tree algorithm can be decomposed into three parts: a type of space tree, a pruning dual-tree traversal, and a point-to-point `BaseCase()` function and node-to-node `Score()` function that determines when pruning is possible. Precise definitions and details of the abstraction are given by Curtin et al. (2013b), but for our purposes, this means that we can describe a dual-tree k-means algorithm entirely with a straightforward `BaseCase()` function and `Score()` function. Any tree and any traversal can then be used to create a working dual-tree algorithm.

The two types of trees we will explicitly consider in this paper are the kd-tree and the cover tree (Beygelzimer et al., 2006), but it should be remembered that the algorithm as provided is sufficiently general to work with any other type of tree. Therefore, we standardize notation for trees: a tree is denoted with $\mathscr{T}$, and a node in the tree is denoted by $\mathscr{N}$. Each node in a tree may have children; the set of children of $\mathscr{N}_i$ is denoted $\mathscr{C}_i$. In addition, each node may hold some points; this set of points is denoted $\mathscr{P}_i$. Lastly, the set of *descendant* points of a node $\mathscr{N}_i$ is denoted $\mathscr{D}_i^p$. The descendant points are all points held by descendant nodes, and it is important to note that the set $\mathscr{P}_i$ is *not* equivalent to $\mathscr{D}_i^p$. This notation is taken from Curtin et al. (2013b) and is detailed more comprehensively there. Lastly, we say that a centroid c *owns* a point p if c is the closest centroid to p.

4. Pruning strategies

All of the existing accelerated k-means algorithms operate by avoiding unnecessary work via the use of pruning strategies. Thus, we will pursue four pruning strategies, each based on or related to earlier work (Pelleg & Moore, 1999; Elkan, 2003; Hamerly, 2010).

These pruning strategies are meant to be used during the dual-tree traversal, for which we have built a query tree $\mathscr{T}_q$ on the points and a reference tree $\mathscr{T}_r$ on the centroids. Therefore, these pruning strategies consider not just combinations of single points and centroid p_q and c_i, but the combination of sets of points and sets of centroids, represented by a query tree node $\mathscr{N}_q$ and a centroid tree node $\mathscr{N}_r$. This allows us to prune many centroids for many points simultaneously.

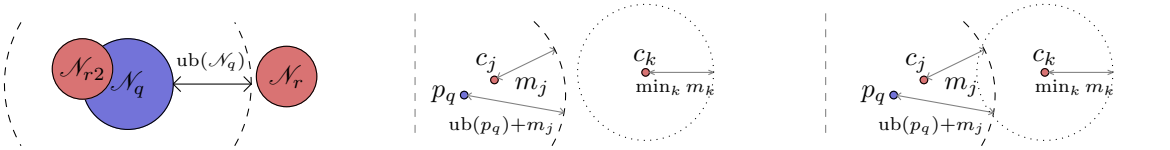
Strategy one. When visiting a particular combination $(\mathscr{N}_q, \mathscr{N}_r)$ (with $\mathscr{N}_q$ holding points in the dataset and $\mathscr{N}_r$ holding centroids), the combination should be pruned if every descendant centroid in $\mathscr{N}_r$ can be shown to own none of the points in $\mathscr{N}_q$. If we have cached an upper bound $\text{ub}(\mathscr{N}_q)$ on the distance between any descendant point of $\mathscr{N}_q$ and its nearest cluster centroid that satisfies

$$\text{ub}(\mathscr{N}_q) \geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_q) \tag{1}$$

where c_q is the cluster centroid nearest to point p_q, then the node $\mathscr{N}_r$ can contain no centroids that own any descendant points of $\mathscr{N}_q$ if

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > \text{ub}(\mathscr{N}_q). \tag{2}$$

This relation bears similarity to the pruning rules for



(a) $\mathscr{N}_r$ can be pruned. (b) p_q's owner cannot change. (c) p_q's owner can change.

Figure 1. Different pruning situations.

nearest neighbor search (Curtin et al., 2013b) and max-kernel search (Curtin & Ram, 2014). Figure 1a shows a situation where $\mathscr{N}_r$ can be pruned; in this case, ball-shaped tree nodes are used, and the upper bound $\text{ub}(\mathscr{N}_q)$ is set to $d_{\max}(\mathscr{N}_q, \mathscr{N}_{r2})$.

Strategy two. The recursion down a particular branch of the query tree should terminate early if we can determine that only one cluster can possibly own all of the descendant points of that branch. This is related to the first strategy. If we have been caching the number of pruned centroids (call this $\text{pruned}(\mathscr{N}_q)$), as well as the identity of any arbitrary non-pruned centroid (call this $\text{closest}(\mathscr{N}_q)$), then if $\text{pruned}(\mathscr{N}_q) = k - 1$, we may conclude that the centroid $\text{closest}(\mathscr{N}_q)$ is the owner of all descendant points of $\mathscr{N}_q$, and there is no need for further recursion in $\mathscr{N}_q$.

Strategy three. The traversal should not visit nodes whose owner could not have possibly changed between iterations; that is, the tree should be coalesced to include only nodes whose owners may have changed.

There are two easy ways to use the triangle inequality to show that the owner of a point cannot change between iterations. Figures 1b and 1c show the first: we have a point p_q with owner c_j and second-closest centroid c_k. Between iterations, each centroid will move when it is recalculated; define the distance that centroid c_i has moved as m_i. Then we bound the distances for the next iteration: $d(p_q, c_j) + m_j$ is an upper bound on the distance from p_q to its owner next iteration, and $d(p_q, c_k) - \max_i m_i$ is a lower bound on the distance from p_q to its second closest centroid next iteration. We may use these bounds to conclude that if

$$d(p_q, c_j) + m_j < d(p_q, c_k) - \max_i m_i, \tag{3}$$

then the owner of p_q next iteration must be c_j. Generalizing from individual points p_q to tree nodes $\mathscr{N}_q$ is easy. This pruning strategy can only be used when all descendant points of $\mathscr{N}_q$ are owned by a single centroid, and in order to perform the prune, we need to establish a lower bound on the distance between any descendant point of the node $\mathscr{N}_q$ and the second closest centroid. Call this bound $\text{lb}(\mathscr{N}_q)$. Remember that $\text{ub}(\mathscr{N}_q)$ provides an upper bound on the distance between any descendant point of $\mathscr{N}_q$ and its nearest centroid. Then, if all descendant points of $\mathscr{N}_q$ are owned by some cluster c_j in one iteration, and

$$\text{ub}(\mathscr{N}_q) + m_j < \text{lb}(\mathscr{N}_q) - \max_i m_i, \tag{4}$$

then $\mathscr{N}_q$ is owned by cluster c_j in the next iteration. Implementationally, it is convenient to have $\text{lb}(\mathscr{N}_q)$ store a lower bound on the distance between any descendant point of $\mathscr{N}_q$ and the nearest pruned centroid. Then, if $\mathscr{N}_r$ is entirely owned by one cluster, all other centroids are pruned, and $\text{lb}(\mathscr{N}_q)$ holds the necessary lower bound for pruning according to the rule above.

The second way to use the triangle inequality to show that an owner cannot change depends on the distances between centroids. Suppose that p_q is owned by c_j at the current iteration; then, if

$$d(p_q, c_j) - m_j < 2\left(\min_{c_i \in C, c_i \neq c_j} d(c_i, c_j)\right) \tag{5}$$

then c_j will own p_q next iteration (Elkan, 2003). We may adapt this rule to tree nodes $\mathscr{N}_q$ in the same way as the previous rule; if $\mathscr{N}_q$ is owned by cluster c_j during this iteration and

$$\text{ub}(\mathscr{N}_q) + m_j < 2\left(\min_{c_i \in C, c_i \neq c_j} d(c_i, c_j)\right) \tag{6}$$

then $\mathscr{N}_q$ is owned by cluster c_j in the next iteration. Note that the above rules do work with individual points p_q instead of nodes $\mathscr{N}_q$ if we have a valid upper bound $\text{ub}(p_q)$ and a valid lower bound $\text{lb}(p_q)$. Any nodes or points that satisfy the above conditions do not need to be visited during the next iteration, and can be removed from the tree for the next iteration.

Strategy four. The traversal should use bounding information from previous iterations; for instance, $\text{ub}(\mathscr{N}_q)$ should not be reset to ∞ at the beginning of each iteration. Between iterations, we may update $\text{ub}(\mathscr{N}_q)$, $\text{ub}(p_q)$, $\text{lb}(\mathscr{N}_q)$, and $\text{lb}(p_q)$ according to the following rules:

$$\text{ub}(\mathscr{N}_q) \leftarrow \begin{cases} \text{ub}(\mathscr{N}_q) + m_j & \text{if } \mathscr{N}_q \text{ is} \\ \quad \text{owned by a single cluster } c_j \\ \text{ub}(\mathscr{N}_q) + \max_i m_i & \text{if } \mathscr{N}_q \text{ is} \\ \quad \text{not owned by a single cluster,} \end{cases} \tag{7}$$

$$\text{ub}(p_q) \leftarrow \text{ub}(p_q) + m_j, \tag{8}$$

$$\text{lb}(\mathscr{N}_q) \leftarrow \text{lb}(\mathscr{N}_q) - \max_i m_i, \tag{9}$$

$$\text{lb}(p_q) \leftarrow \text{lb}(p_q) - \max_i m_i. \tag{10}$$

Special handling is required when descendant points of $\mathscr{N}_q$ are not owned by a single centroid (Equation 7). It is also true that for a child node $\mathscr{N}_c$ of $\mathscr{N}_q$, $\text{ub}(\mathscr{N}_q)$ is a valid upper bound for $\mathscr{N}_c$ and $\text{lb}(\mathscr{N}_q)$ is a valid lower bound for $\mathscr{N}_c$: that is, the upper and lower bounds may be taken from a parent, and they are still valid.

5. The dual-tree k-means algorithm

These four pruning strategies lead to a high-level k-means algorithm, described in Algorithm 1. During the course of this algorithm, to implement each of our pruning strategies, we will need to maintain the following quantities:

- $\text{ub}(\mathscr{N}_q)$: an upper bound on the distance between any descendant point of a node $\mathscr{N}_q$ and the nearest centroid to that point.
- $\text{lb}(\mathscr{N}_q)$: a lower bound on the distance between any descendant point of a node $\mathscr{N}_q$ and the nearest pruned centroid.
- $\text{pruned}(\mathscr{N}_q)$: the number of centroids pruned during traversal for $\mathscr{N}_q$.
- $\text{closest}(\mathscr{N}_q)$: if $\text{pruned}(\mathscr{N}_q) = k-1$, this holds the owner of all descendant points of $\mathscr{N}_q$.
- $\text{canchange}(\mathscr{N}_q)$: whether or not $\mathscr{N}_q$ can change owners next iteration.
- $\text{ub}(p_q)$: an upper bound on the distance between point p_q and its nearest centroid.
- $\text{lb}(p_q)$: a lower bound on the distance between point p_q and its second nearest centroid.
- $\text{closest}(p_q)$: the closest centroid to p_q (this is also the owner of p_q).
- $\text{canchange}(p_q)$: whether or not p_q can change owners next iteration.

At the beginning of the algorithm, each upper bound is initialized to ∞, each lower bound is initialized to ∞, $\text{pruned}(\cdot)$ is initialized to 0 for each node, and $\text{closest}(\cdot)$ is initialized to an invalid centroid for each node and point. $\text{canchange}(\cdot)$ is set to `true` for each node and point. Thus line 6 does nothing on the first iteration.

First, consider the dual-tree algorithm called on line 9. As detailed earlier, we can describe a dual-tree algorithm as a combination of tree type, traversal, and point-to-point `BaseCase()` and node-to-node `Score()` functions. Thus, we need only present `BaseCase()` (Algorithm 2) and `Score()` (Algorithm 3)[2].

[2]In these algorithms, we assume that any point present in a node $\mathscr{N}_i$ will also be present in at least one child $\mathscr{N}_c \in \mathscr{C}_i$. It is possible to fully generalize to any tree type, but the exposition is significantly more complex, and our assumption covers most standard tree types anyway.

Algorithm 1 High-level outline of dual-tree k-means.

```
1: Input: dataset S ∈ R^{N×d}, initial centroids C ∈ R^{k×d}.
2: Output: converged centroids C.
3: T ← a tree built on S
4: while centroids C not converged do
5:     {Remove nodes in the tree if possible.}
6:     T ← CoalesceNodes(T)
7:     T_c ← a tree built on C
8:     {Call dual-tree algorithm.}
9:     Perform a dual-tree recursion with T, T_c, BaseCase(), and Score().
10:    {Restore the tree to its non-coalesced form.}
11:    T ← DecoalesceNodes(T)
12:    {Update centroids and bounding information.}
13:    C ← UpdateCentroids(T)
14:    T ← UpdateTree(T)
15: return C
```

The `BaseCase()` function is simple: given a point p_q and a centroid c_r, the distance $d(p_q, c_r)$ is calculated; $\text{ub}(p_q)$, $\text{lb}(p_q)$, and $\text{closest}(p_q)$ are updated if needed.

`Score()` is more complex. The first stanza (lines 4–6) takes the values of $\text{pruned}(\cdot)$ and $\text{lb}(\cdot)$ from the parent node of $\mathscr{N}_q$; this is necessary to prevent $\text{pruned}(\cdot)$ from undercounting. Next, we prune if the owner of $\mathscr{N}_q$ is already known (line 7). If the minimum distance between any descendant point of $\mathscr{N}_q$ and any descendant centroid of $\mathscr{N}_r$ is greater than $\text{ub}(\mathscr{N}_q)$, then we may prune the combination (line 16). In that case we may also improve the lower bound (line 14). Note the special handling in line 15: our definition of tree allows points to be held in more than one node; thus, we must avoid double-counting clusters that we prune.[3]. If the node combination cannot be pruned in this way, an attempt is made to update the upper bound (lines 17–20). Instead of using $d_{\max}(\mathscr{N}_q, \mathscr{N}_r)$, we may use a tighter upper bound: select any descendant centroid c from $\mathscr{N}_r$ and use $d_{\max}(\mathscr{N}_q, c)$. This still provides a valid upper bound, and in practice is generally smaller than $d_{\max}(\mathscr{N}_q, \mathscr{N}_r)$. We simply set $\text{closest}(\mathscr{N}_q)$ to c (line 20); $\text{closest}(\mathscr{N}_q)$ only holds the owner of $\mathscr{N}_q$ if all centroids except one are pruned—in which case the owner *must* be c.

Thus, at the end of the dual-tree algorithm, we know the owner of every node (if it exists) via $\text{closest}(\cdot)$ and $\text{pruned}(\cdot)$, and we know the owner of every point

[3]For trees like the kd-tree and the metric tree, which do not hold points in more than one node, no special handling is required: we will never prune a cluster twice for a given query node $\mathscr{N}_q$.

Algorithm 2 `BaseCase()` for dual-tree k-means.

Input: query point p_q, reference centroid c_r
Output: distance between p_q and c_r

if $d(p_q, c_r) < \text{ub}(p_q)$ **then**
  $\text{lb}(p_q) \leftarrow \text{ub}(p_q)$
  $\text{ub}(p_q) \leftarrow d(p_q, c_r)$
  $\text{closest}(p_q) \leftarrow c_r$
else if $d(p_q, c_r) < \text{lb}(p_q)$ **then**
  $\text{lb}(p_q) \leftarrow d(p_q, c_r)$
return $d(p_q, c_r)$

Algorithm 3 `Score()` for dual-tree k-means.

Input: query node $\mathscr{N}_q$, reference node $\mathscr{N}_r$
Output: score for node combination $(\mathscr{N}_q, \mathscr{N}_r)$, or ∞ if the combination can be pruned

{Update the number of pruned nodes, if needed.}
if $\mathscr{N}_q$ not yet visited and is not the root node **then**
  $\text{pruned}(\mathscr{N}_q) \leftarrow \text{parent}(\mathscr{N}_q)$
  $\text{lb}(\mathscr{N}_q) \leftarrow \text{lb}(\text{parent}(\mathscr{N}_q))$
if $\text{pruned}(\mathscr{N}_q) = k - 1$ **then return** ∞
$s \leftarrow d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
$c \leftarrow$ any descendant cluster centroid of $\mathscr{N}_r$
if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) > \text{ub}(\mathscr{N}_q)$ **then**
  {This cluster node owns no descendant points.}
  if $d_{\min}(\mathscr{N}_q, \mathscr{N}_r) < \text{lb}(\mathscr{N}_q)$ **then**
    {Improve the lower bound for pruned nodes.}
    $\text{lb}(\mathscr{N}_q) \leftarrow d_{\min}(\mathscr{N}_q, \mathscr{N}_r)$
  $\text{pruned}(\mathscr{N}_q) \mathrel{+}= |\mathscr{D}_r^p \setminus \{\text{clusters not pruned}\}|$
  $s \leftarrow \infty$
else if $d_{\max}(\mathscr{N}_q, c) < \text{ub}(\mathscr{N}_q)$ **then**
  {We may improve the upper bound.}
  $\text{ub}(\mathscr{N}_q) \leftarrow d_{\max}(\mathscr{N}_q, \mathscr{N}_r)$
  $\text{closest}(\mathscr{N}_q) \leftarrow c$
{Check if all clusters (except one) are pruned.}
if $\text{pruned}(\mathscr{N}_q) = k - 1$ **then return** ∞
return s

via closest($\cdot$). A simple algorithm to do this is given here as Algorithm 4 (`UpdateCentroids()`); it is a depth-first recursion through the tree that terminates a branch when a node is owned by a single cluster.

Next is updating the bounds in the tree and determining if nodes and points can change owners next iteration; this work is encapsulated in the `UpdateTree()` algorithm, which is an implementation of strategies 3 and 4 (see the appendix for details). Once `UpdateTree()` sets the correct value of canchange($\cdot$) for every point and node, we coalesce the tree for the next iteration with the `CoalesceTree()` function. Coalescing the tree is straightforward: we simply remove any nodes from the tree where canchange($\cdot$) is `false`. This leaves a smaller tree with no nodes where canchange($\cdot$) is `false`. Decoalescing the tree (`DecoalesceTree()`) is done by restoring the tree to its original state. See the appendix for more details.

Algorithm 4 `UpdateCentroids()`.

Input: tree $\mathscr{T}$ built on dataset S
Output: new centroids C

$C := \{c_0, \ldots, c_{k-1}\} \leftarrow \mathbf{0}^{k \times d}$; $n = \mathbf{0}^k$
{s is a stack.}
$s \leftarrow \{\text{root}(\mathscr{T})\}$
while $|s| > 0$ **do**
  $\mathscr{N}_i \leftarrow s.\texttt{pop}()$
  if $\text{pruned}(\mathscr{N}_i) = k - 1$ **then**
    {The node is entirely owned by a cluster.}
    $j \leftarrow$ index of $\text{closest}(\mathscr{N}_i)$
    $c_j \leftarrow c_j + |\mathscr{D}_i^p| \, \text{centroid}(\mathscr{N}_i)$
    $n_j \leftarrow n_j + |\mathscr{D}_i^p|$
  else
    {The node is not entirely owned by a cluster.}
    if $|\mathscr{C}_i| > 0$ **then** $s.\texttt{push}(\mathscr{C}_i)$
    else
      for $p_i \in \mathscr{P}_i$ not yet considered
        $j \leftarrow$ index of $\text{closest}(p_i)$
        $c_j \leftarrow c_j + p_i$; $n_j \leftarrow n_j + 1$
for $c_i \in C$, **if** $n_i > 0$ **then** $c_i \leftarrow c_i / n_i$
return C

6. Theoretical results

Space constraints allow us to only provide proof sketches for the first two theorems here. Detailed proofs are given in the appendix.

Theorem 1. *A single iteration of dual-tree k-means as given in Algorithm 1 will produce exactly the same results as the brute-force $O(kN)$ implementation.*

Proof. (Sketch.) First, we show that the dual-tree algorithm (line 9) produces correct results for ub($\cdot$), lb($\cdot$), pruned($\cdot$), and closest($\cdot$) for every point and node. Next, we show that `UpdateTree()` maintains the correctness of those four quantities and only marks canchange($\cdot$) to `false` when the node or point truly cannot change owner. Next, it is easily shown that `CoalesceTree()` and `DecoalesceTree()` do not affect the results of the dual-tree algorithm because the only nodes and points removed are those where canchange($\cdot$) = `false`. Lastly, we show that `UpdateCentroids()` produces centroids correctly. □

Next, we consider the runtime of the algorithm. Our results are with respect to the *expansion constant* c_k of the centroids (Beygelzimer et al., 2006), which is a measure of intrinsic dimension. c_{qk} is a related quan-

tity: the largest expansion constant of C plus any point in the dataset. Our results also depend on the imbalance of the tree $i_t(\mathscr{T})$, which in practice generally scales linearly in N (Curtin et al., 2015). As with the other theoretical results, more detail on each of these quantities is available in the appendix.

Theorem 2. *When cover trees are used, a single iteration of dual-tree k-means as in Algorithm 1 can be performed in* $O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})) + c_k^9 k \log k)$ *time.*

Proof. (Sketch.) Cover trees have $O(N)$ nodes (Beygelzimer et al., 2006); because `CoalesceTree()`, `DecoalesceTree()`, `UpdateCentroids()`, and `UpdateTree()` can be performed in one pass of the tree, these steps may each be completed in $O(N)$ time. Building a tree on the centroids takes $O(c_k^6 k \log k)$ time, where c_k is the expansion constant of the centroids. Recent results show that dual-tree algorithms that use the cover tree may have their runtime easily bounded (Curtin et al., 2015). We may observe that our pruning rules are at least as tight as nearest neighbor search; this means that the dual-tree algorithm (line 11) may be performed in $O(c_{kr}^9 (N + i_t(\mathscr{T})))$ time. Also, we must perform nearest neighbor search on the centroids, which costs $O(c_k^9 (k + i_t(\mathscr{T})))$ time. This gives a total per-iteration runtime of $O(c_{kr}^9 (N + i_t(\mathscr{T})) + c_k^6 k \log k + c_k^9 i_t(\mathscr{T}_k))$. □

This result holds intuitively. By building a tree on the centroids, we are able to prune many centroids at once, and as a result the amortized cost of finding the nearest centroid to a point is $O(1)$. This meshes with earlier theoretical results (Beygelzimer et al., 2006; Curtin et al., 2015; Ram et al., 2009a) and earlier empirical results (Gray & Moore, 2003; 2001) that suggest that an answer can be obtained for a single query point in $O(1)$ time. Note that this worst-case bound depends on the intrinsic dimension (the expansion constant) of the centroids, c_k, and the related quantity c_{qk}. If the intrinsic dimension of the centroids is low—that is, if the centroids are distributed favorably—the dual-tree algorithm will be more efficient.

However, this bound is generally quite loose in practice. First, runtime bounds for cover trees are known to be loose (Curtin et al., 2015). Second, this particular bound does not consider the effect of coalescing the tree. In any given iteration, especially toward the end of the k-means clustering, most points will have canchange$(\cdot)$ = `false` and thus the coalesced tree will be far smaller than the full tree built on all N points.

Theorem 3. *Algorithm 1 uses no more than* $O(N+k)$ *memory when cover trees are used.*

Proof. This proof is straightforward. A cover tree on N points takes $O(N)$ space. So the trees and associated bounds take $O(N)$ and $O(k)$ space. Also, the dataset and centroids take $O(N)$ and $O(k)$ space. □

Dataset	N	d	tree build time	
			kd-tree	cover tree
cloud	2048	10	0.001s	0.005s
cup98b	95413	56	1.640s	32.41s
birch3	100000	2	0.037s	2.125s
phy	150000	78	4.138s	22.99s
power	2075259	7	7.342s	1388s
lcdm	6000000	3	4.345s	6214s

Table 2. Dataset information.

7. Experiments

The next thing to consider is the empirical performance of the algorithm. We use the publicly available `kmeans` program in **mlpack** (Curtin et al., 2013a); in our experiments, we run it as follows:

```
$ kmeans -i dataset.csv -I centroids.csv -c
    $k -v -e -a $algorithm
```

where `$k` is the number of clusters and `$algorithm` is the algorithm to be used. Each algorithm is implemented in C++. For the `yinyang` algorithm, we use the authors' implementation. We use a variety of k values on mostly real-world datasets; details are shown in Table 2 (Lichman, 2013; Zhang et al., 1997; Lupton et al., 2001). The table also contains the time taken to build a kd-tree (for `blacklist` and `dualtree-kd`) and a cover tree (for `dualtree-ct`). Cover trees are far more complex to build than kd-trees; this explains the long cover tree build time. Even so, the tree only needs to be built once during the k-means run. If results are required for multiple values of k—such as in the X-means algorithm (Pelleg & Moore, 2000)—then the tree built on the points may be re-used.

Clusters were initialized using the Bradley-Fayyad refined start procedure (1998); however, this was too slow for the very large datasets, so in those cases points were randomly sampled as the initial centroids. k-means was then run until convergence on each dataset. These simulations were performed on a modest consumer desktop with an Intel i5 with 16GB RAM, using **mlpack**'s benchmarking system (Edel et al., 2014).

Average runtime per iteration results are shown in Table 3. The amount of work that is being pruned away is somewhat unclear from the runtime results, because the `elkan` and `hamerly` algorithms access points linearly and thus benefit from cache effects; this is not true of the tree-based algorithms. Therefore, the average number of distance calculations per iteration are also included in the results.

It is immediately clear that for large datasets, `dualtree-kd` is fastest, and `dualtree-ct` is almost

			avg. per-iteration runtime (distance calculations)					
dataset	k	**iter.**	`elkan`	`hamerly`	`yinyang`	`blacklist`	`dualtree-kd`	`dualtree-ct`
cloud	3	8	1.50e-4s (867)	1.11e-4s (1.01k)	1.11e-1s (2.00k)	**4.68e-5s** (302)	1.27e-4s (**278**)	2.77e-4s (443)
cloud	10	14	2.09e-4s (**1.52k**)	1.92e-4s (4.32k)	7.66e-2s (9.55k)	**1.55e-4s** (2.02k)	3.69e-4s (1.72k)	5.36e-4s (2.90k)
cloud	50	19	5.87e-4s (**2.57k**)	**5.30e-4s** (21.8k)	9.66e-3s (15.6k)	8.20e-4s (12.6k)	1.23e-3s (5.02k)	1.09e-3s (9.84k)
cup98b	50	224	0.0445s (**25.9k**)	0.0557s (962k)	0.0465s (313k)	**0.0409s** (277k)	0.0955s (254k)	0.1089s (436k)
cup98b	250	168	0.1972s (**96.8k**)	0.4448s (8.40M)	**0.1417s** (898k)	0.2033s (1.36M)	0.4585s (1.38M)	0.3237s (2.73M)
cup98b	750	116	1.1719s (**373k**)	1.8778s (36.2M)	**0.2653s** (1.26M)	0.6365s (4.11M)	1.2847s (4.16M)	0.8056s (81.4M)
birch3	50	129	0.0194s (**24.2k**)	0.0093s (566k)	0.0378s (399k)	**0.0030s** (42.7k)	0.0082s (37.4k)	0.0378s (67.9k)
birch3	250	812	0.0895s (**42.8k**)	0.0314s (2.59M)	0.0711s (239k)	**0.0164s** (165k)	0.0183s (79.7k)	0.0485s (140k)
birch3	750	373	0.3253s (292k)	0.0972s (8.58M)	0.1423s (476k)	0.0554s (450k)	**0.02989s** (**126k**)	0.0581s (235k)
phy	50	34	0.0668s (82.3k)	0.1064s (1.38M)	0.1072s (808k)	**0.0081s** (**33.0k**)	0.02689s (67.8k)	0.0945s (188k)
phy	250	38	0.1627s (121k)	0.4634s (6.83M)	0.2469s (2.39M)	**0.0249s** (104k)	0.0398s (**90.4k**)	0.1023s (168k)
phy	750	35	0.7760s (**410k**)	2.9192s (43.8M)	0.6418s (5.61M)	**0.2478s** (1.19M)	0.2939s (1.10M)	0.3330s (1.84M)
power	25	4	0.3872s (2.98M)	0.2880s (12.9M)	1.1257s (33.5M)	**0.0301s** (216k)	0.0950s (**87.4k**)	0.6658s (179k)
power	250	101	2.6532s (425k)	0.1868s (7.83M)	1.2684s (10.3M)	0.1504s (1.13M)	**0.1354s** (**192k**)	0.6405s (263k)
power	1000	870	*out of memory*	6.2407s (389M)	4.4261s (9.41M)	0.6657s (2.98M)	**0.4115s** (**1.57M**)	1.1799s (4.81M)
power	5000	504	*out of memory*	29.816s (1.87B)	22.7550s (58.6M)	4.1597s (11.7M)	**1.0580s** (**3.85M**)	1.7070s (12.3M)
power	15000	301	*out of memory*	111.74s (6.99B)	*out of memory*	*out of memory*	**2.3708s** (**8.65M**)	2.9472s (30.9M)
lcdm	500	507	*out of memory*	6.4084s (536M)	8.8926s (44.5M)	0.9347s (4.20M)	**0.7574s** (**3.68M**)	2.9428s (7.03M)
lcdm	1000	537	*out of memory*	16.071s (1.31B)	18.004s (74.7M)	2.0345s (5.93M)	**0.9827s** (**5.11M**)	3.3482s (10.0M)
lcdm	5000	218	*out of memory*	64.895s (5.38B)	*out of memory*	12.909s (16.2M)	**1.8972s** (**8.54M**)	3.9110s (19.0M)
lcdm	20000	108	*out of memory*	298.55s (24.7B)	*out of memory*	*out of memory*	**4.1911s** (**17.8M**)	5.5771s (43.2M)

Table 3. Empirical results for k-means.

as fast. The `elkan` algorithm, because it holds kN bounds, is able to prune away a huge amount of work and is very fast for small datasets; however, maintaining all of these bounds becomes prohibitive with large k and the algorithm exhausts all available memory. The `blacklist` algorithm has the same issue: on the largest datasets, with the largest k values, the space required to maintain all the blacklists is too much. This is also true of the `yinyang` algorithm, which must maintain bounds between each point and each group of centroids. For large k, this burden becomes too much and the algorithm fails. The `hamerly` and dual-tree algorithms, on the other hand, are the best-behaved with memory usage and do not have any issues with large N or large k; however, the `hamerly` algorithm is very slow on large datasets because it is not able to prune many points at once.

Similar to the observations about the `blacklist` algorithm, the tree-based approaches are less effective in higher dimensions (Pelleg & Moore, 1999). This is an important point: the performance of tree-based approaches suffer in high dimensions in part because the bound $d_{\min}(\cdot, \cdot)$ generally becomes looser as dimension increases. This is partly because the volume of nodes in high dimensions is much higher; consider that a ball has volume that is exponential in the dimension.

Even so, in our results, we see speedup in reasonable dimensions (for example, the `phy` dataset has 78 dimensions). Further, because our algorithm is tree-independent, we may use tree structures that are tailored to high-dimensional data (Arya et al., 1998)—including ones that have not yet been developed. From our results we believe as a rule of thumb that the dual-tree k-means algorithm can be effective up to a hundred dimensions or more.

Another clear observation is that when k is scaled on a single dataset, the `dualtree-kd` and `dualtree-ct` algorithms nearly always scale better (in terms of runtime) than the other algorithms. These results show that our algorithm satisfies its original goals: to be able to scale effectively to large k and N.

8. Conclusion and future directions

Using four pruning strategies, we have developed a flexible, tree-independent dual-tree k-means algorithm that is the best-performing algorithm for large datasets and large k in small-to-medium dimensions. It is theoretically favorable, has a small memory footprint, and may be used in conjunction with initial point selection and approximation schemes for additional speedup.

There are still interesting future directions to pursue, though. The first direction is parallelism: because our dual-tree algorithm is agnostic to the type of traversal used, we may use a parallel traversal (Curtin et al., 2013b), such as an adapted version of a recent parallel dual-tree algorithm (Lee et al., 2012). The second direction is kernel k-means and other spectral clustering techniques: our algorithm may be merged with the ideas of Curtin & Ram (2014) to perform kernel k-means. The third direction is theoretical. Recently, more general notions of intrinsic dimensionality have been proposed (Houle, 2013; Amsaleg et al., 2015); these may enable tighter and more descriptive runtime bounds. Our work thus provides a useful and fast k-means algorithm and also opens promising avenues to further accelerated clustering algorithms.

References

Amsaleg, L., Chelly, O., Furon, T., Girard, S., Houle, M.E., Kawarabayashi, K., and Nett, M. Estimating local intrinsic dimensionality. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*, pp. 29–38, 2015.

Arthur, D. and Vassilvitskii, S. k-means++: The advantages of careful seeding. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1027–1035, 2007.

Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., and Wu, A.Y. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

Bengio, S., Weston, J., and Grangier, D. Label embedding trees for large multi-class tasks. In *Advances in Neural Information Processing Systems 23 (NIPS '10)*, volume 23, pp. 3, 2010.

Bentley, J.L. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Beygelzimer, A., Kakade, S.M., and Langford, J. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–104, 2006.

Bradley, P.S. and Fayyad, U.M. Refining initial points for k-means clustering. In *Proceedings of the 15th International Conference on Machine Learning (ICML '98)*, pp. 91–99, 1998.

Can, F. and Ozkarahan, E.A. Concepts and effectiveness of the cover-coefficient-based clustering methodology for text databases. *ACM Transactions on Database Systems*, 15(4):483–517, December 1990. ISSN 0362-5915. doi: 10.1145/99935.99938. URL http://doi.acm.org/10.1145/99935.99938.

Coates, A., Ng, A.Y., and Lee, H. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of AISTATS*, pp. 215–223, 2011.

Csurka, G., Dance, C., Fan, L., Willamowski, J., and Bray, C. Visual categorization with bags of keypoints. In *Workshop on Statistical Learning in Computer Vision, ECCV*, volume 1, pp. 1–16, 2004.

Curtin, R.R. and Ram, P. Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining*, 7(4):229–253, 2014. ISSN 1932-1872. doi: 10.1002/sam.11218. URL http://dx.doi.org/10.1002/sam.11218.

Curtin, R.R., Cline, J.R., Slagle, N.P., March, W.B., Ram, P., Mehta, N.A., and Gray, A.G. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013a.

Curtin, R.R., March, W.B., Ram, P., Anderson, D.V., Gray, A.G., and Isbell Jr, C.L. Tree-independent dual-tree algorithms. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, pp. 1435–1443, 2013b.

Curtin, R.R., Ram, P., and Gray, A.G. Fast exact max-kernel search. In *Proceedings of SIAM International Conference on Data Mining 2013 (SDM '13)*, pp. 1–9, 2013c.

Curtin, R.R., Lee, D., March, W.B., and Ram, P. Plug-and-play dual-tree algorithm runtime analysis. *arXiv preprint arXiv:1501.05222*, 2015.

Ding, Y., Zhao, Y., Shen, X., Musuvathi, M., and Mytkowicz, T. 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 '15)*, pp. 579–587, 2015.

Edel, M., Soni, A., and Curtin, R.R. An automatic benchmarking system. In *Proceedings of the NIPS 2014 Workshop on Software Engineering for Machine Learning*, 2014.

Elkan, C. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML '03)*, volume 3, pp. 147–153, 2003.

Frahling, G. and Sohler, C. A fast k-means implementation using coresets. *International Journal of Computational Geometry & Applications*, 18(06): 605–625, 2008.

Gray, A.G. and Moore, A.W. 'N-Body' problems in statistical learning. In *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, volume 4, pp. 521–527, 2001.

Gray, A.G. and Moore, A.W. Nonparametric density estimation: Toward computational tractability. In *SIAM International Conference on Data Mining (SDM)*, pp. 203–211, 2003.

Hamerly, G. Making k-means even faster. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pp. 130–140, 2010.

Houle, M.E. Dimensionality, discriminability, density and distance distributions. In *2013 IEEE 13th International Conference on Data Mining Workshops (ICDMW)*, pp. 468–473, 2013.

Karger, D.R. and Ruhl, M. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC 2002)*, pp. 741–750, 2002.

Kwon, Y.C., Nunley, D., Gardner, J.P., Balazinska, M., Howe, B., and Loebman, S. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. In *Scientific and Statistical Database Management*, pp. 132–150. Springer, 2010.

Lee, D., Vuduc, R.W., and Gray, A.G. A distributed kernel summation framework for general-dimension machine learning. In *Proceedings of the 2012 SIAM International Conference on Data Mining (SDM '12)*, pp. 391–402, 2012.

Lichman, M. UCI machine learning repository, 2013. URL `http://archive.ics.uci.edu/ml`.

Liu, T., Moore, A.W., Yang, K., and Gray, A.G. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems 18 (NIPS '04)*, pp. 825–832, 2004.

Lupton, R., Gunn, J.E., Ivezic, Z., Knapp, G.R., and Kent, S. The SDSS imaging pipelines. In *Astronomical Data Analysis Software and Systems X*, volume 238, pp. 269, 2001.

March, W.B., Ram, P., and Gray, A.G. Fast Euclidean minimum spanning tree: algorithm, analysis, and applications. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pp. 603–612, 2010.

Moore, A.W. Very fast em-based mixture model clustering using multiresolution kd-trees. *Advances in Neural Information Processing Systems*, pp. 543–549, 1999.

Pelleg, D. and Moore, A.W. Accelerating exact k-means algorithms with geometric reasoning. In *Proceedings of KDD '99*, pp. 277–281. ACM, 1999.

Pelleg, D. and Moore, A.W. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML '00)*, pp. 727–734, 2000.

Ram, P., Lee, D., March, W.B., and Gray, A.G. Linear-time algorithms for pairwise statistical problems. *Advances in Neural Information Processing Systems 22 (NIPS 2009)*, pp. 1527–1535, 2009a.

Ram, P., Lee, D., Ouyang, H., and Gray, A.G. Rank-approximate nearest neighbor search: Retaining meaning and speed in high dimensions. *Advances in Neural Information Processing Systems*, 22, 2009b.

Zhang, T., Ramakrishnan, R., and Livny, M. BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1(2): 141–182, 1997.

A. Supplementary material

Unfortunately, space constraints prevent adequate explanation of each of the points in the main paper. This supplementary material is meant to clarify all of the parts of the dual-tree k-means algorithm that space did not permit in the main paper.

A.1. Updating the tree

In addition to updating the centroids, the bounding information contained within the tree must be updated according to pruning strategies 3 and 4. Unfortunately, this yields a particularly complex recursive algorithm, given in Algorithm 5.

The first if statement (lines 4–10) catches the case where the parent cannot change owner next iteration; in this case, the parent's upper bound and lower bound can be taken as valid bounds. In addition, the upper and lower bounds are adjusted to account for cluster movement between iterations, so that the bounds are valid for next iteration.

If the node $\mathscr{N}_i$ has an owner, the algorithm then attempts to use the pruning rules established in Equations 4 and 6 in the main paper, to determine if the owner of $\mathscr{N}_i$ can change next iteration. If not, canchange($\mathscr{N}_i$) is set to `false` (line 18). On the other hand, if the pruning check fails, the upper bound is tightened and the pruning check is performed a second time. It is worth noting that $d_{\max}(\mathscr{N}_i, c_j)$ may not actually be less than the current value of $\mathrm{ub}(\mathscr{N}_i)$, which is why the min is necessary.

After recursing into the children of $\mathscr{N}_i$, if $\mathscr{N}_i$ could have an owner change, each point is individually checked using the same approach (lines 31–45). However, there is a slight difference: if a point's owner can change, the upper and lower bounds must be set to ∞ (lines 44–45). This is only necessary with points; `BaseCase()`

Algorithm 5 `UpdateTree()` for dual-tree k-means.

Input: node $\mathcal{N}_i$, $\text{ub}(\cdot)$, $\text{lb}(\cdot)$, $\text{pruned}(\cdot)$, $\text{closest}(\cdot)$, $\text{canchange}(\cdot)$, centroid movements m
Output: updated $\text{ub}(\cdot)$, $\text{lb}(\cdot)$, $\text{pruned}(\cdot)$, $\text{canchange}(\cdot)$

$\text{canchange}(\mathcal{N}_i) \leftarrow$ `true`
if $\mathcal{N}_i$ has a parent and $\text{canchange}(\text{parent}(\mathcal{N}_i)) =$ `false` **then**
  {Use the parent's bounds.}
  $\text{closest}(\mathcal{N}_i) \leftarrow \text{closest}(\text{parent}(\mathcal{N}_i))$
  $j \leftarrow$ index of $\text{closest}(\mathcal{N}_i)$
  $\text{ub}(\mathcal{N}_i) \leftarrow \text{ub}(\mathcal{N}_i) + m_j$
  $\text{lb}(\mathcal{N}_i) \leftarrow \text{lb}(\mathcal{N}_i) + \max_i m_i$
  $\text{canchange}(\mathcal{N}_i) \leftarrow$ `false`
else if $\text{pruned}(\mathcal{N}_i) = k - 1$ **then**
  {$\mathcal{N}_i$ is owned by a single cluster. Can that owner change next iteration?}
  $j \leftarrow$ index of $\text{closest}(\mathcal{N}_i)$
  $\text{ub}(\mathcal{N}_i) \leftarrow \text{ub}(\mathcal{N}_i) + m_j$
  $\text{lb}(\mathcal{N}_i) \leftarrow \max\left(\text{lb}(\mathcal{N}_i) - \max_i m_i, \min_{k \neq j} d(c_k, c_j)/2\right)$
  if $\text{ub}(\mathcal{N}_i) < \text{lb}(\mathcal{N}_i)$ **then**
    {The owner cannot change next iteration.}
    $\text{canchange}(\mathcal{N}_i) \leftarrow$ `false`
  else
    {Tighten the upper bound and try to prune again.}
    $\text{ub}(\mathcal{N}_i) \leftarrow \min\left(\text{ub}(\mathcal{N}_i), d_{\max}(\mathcal{N}_i, c_j)\right)$
    if $\text{ub}(\mathcal{N}_i) < \text{lb}(\mathcal{N}_i)$ **then** $\text{canchange}(\mathcal{N}_i) \leftarrow$ `false`
else
  $j \leftarrow$ index of $\text{closest}(\mathcal{N}_i)$
  $\text{ub}(\mathcal{N}_i) \leftarrow \text{ub}(\mathcal{N}_i) + m_j$
  $\text{lb}(\mathcal{N}_i) \leftarrow \text{lb}(\mathcal{N}_i) - \max_k m_k$
{Recurse into each child.}
for each child $\mathcal{N}_c$ of $\mathcal{N}_i$**, call** `UpdateTree(`$\mathcal{N}_c$`)`
{Try to determine points whose owner cannot change if $\mathcal{N}_i$ can change owners.}
if $\text{canchange}(\mathcal{N}_i) =$ `true` **then**
  for $p_i \in \mathcal{P}_i$ **do**
    $j \leftarrow$ index of $\text{closest}(p_i)$
    $\text{ub}(p_i) \leftarrow \text{ub}(p_i) + m_j$
    $\text{lb}(p_i) \leftarrow \min\left(\text{lb}(p_i) - \max_k m_k, \min_{k \neq j} d(c_k, c_j)/2\right)$
    if $\text{ub}(p_i) < \text{lb}(p_i)$ **then**
      $\text{canchange}(p_i) \leftarrow$ `false`
    else
      {Tighten the upper bound and try again.}
      $\text{ub}(p_i) \leftarrow \min\left(\text{ub}(p_i), d(p_i, c_j)\right)$
      if $\text{ub}(p_i) < \text{lb}(p_i)$ **then**
        $\text{canchange}(p_i) \leftarrow$ `false`
      else
        {Point cannot be pruned.}
        $\text{ub}(p_i) \leftarrow \infty$
        $\text{lb}(p_i) \leftarrow \infty$
else
  for $p_i \in \mathcal{P}_i$ where $\text{canchange}(p_i) =$ `false` **do**
    {Maintain upper and lower bounds for points whose owner cannot change.}
    $j \leftarrow$ index of $\text{closest}(p_i)$
    $\text{ub}(p_i) \leftarrow \text{ub}(p_i) + m_j$
    $\text{lb}(p_i) \leftarrow \text{lb}(p_i) - \max_k m_k$
if $\text{canchange}(\cdot) =$ `false` for all children $\mathcal{N}_c$ of $\mathcal{N}_i$ and all points $p_i \in \mathcal{P}_i$ **then**
  $\text{canchange}(\mathcal{N}_i) \leftarrow$ `false`
if $\text{canchange}(\mathcal{N}_i) =$ `true` **then**
  $\text{pruned}(\mathcal{N}_i) \leftarrow 0$

Algorithm 6 `CoalesceTree()` for dual-tree k-means.

Input: tree $\mathscr{T}$
Output: coalesced tree $\mathscr{T}$
{A depth-first recursion to hide nodes where canchange$(\cdot)$ is `false`.}
$s \leftarrow \{\text{root}(\mathscr{T})\}$
while $|s| > 0$ **do**
  $\mathscr{N}_i \leftarrow s.\texttt{pop}()$
  {Special handling is required for leaf nodes and the root node.}
  if $|\mathscr{C}_i| = 0$ **then**
    continue
  else if $\mathscr{N}_i$ is the root node **then**
    for $\mathscr{N}_c \in \mathscr{C}_i$ **do**
      $s.\texttt{push}(\mathscr{N}_c)$
  {See if children can be removed.}
  for $\mathscr{N}_c \in \mathscr{C}_i$ **do**
    if canchange$(\mathscr{N}_c) =$ `false` **then**
      remove child $\mathscr{N}_c$
    else
      $s.\texttt{push}(\mathscr{N}_c)$
  {If only one child is left, then this node is unnecessary.}
  if $|\mathscr{C}_i| = 1$ **then**
    add child to parent$(\mathscr{N}_i)$
    remove $\mathscr{N}_i$ from parent$(\mathscr{N}_i)$'s children
return $\mathscr{T}$

does not take bounding information from previous iterations into account, because no work can be avoided in that way.

Then, we may set canchange$(\mathscr{N}_i)$ to `false` if every point in $\mathscr{N}_i$ and every child of $\mathscr{N}_i$ cannot change owners (and the points and nodes do not necessarily have to have the same owner). Otherwise, we must set pruned$(\mathscr{N}_i)$ to 0 for the next iteration.

A.2. Coalescing the tree

After `UpdateTree()` is called, the tree must be coalesced to remove any nodes where canchange$(\cdot)$ = `false`. This can be accomplished via a single pass over the tree. A simple implementation is given in Algorithm 6. `DecoalesceTree()` may be implemented by simply restoring a pristine copy of the tree which was cached right before `CoalesceTree()` is called.

A.3. Correctness proof

As mentioned in the main document, a correctness proof is possible but difficult. We will individually prove the correctness of various pieces of the dual-tree k-means algorithm, and then we will prove the main correctness result. For precision, we must introduce the exact definition of a space tree and a pruning dual-tree traversal, as given by Curtin et al. (Curtin et al., 2013b).

Definition 1. *A **space tree** on a dataset $S \in \Re^{N \times D}$ is an undirected, connected, acyclic, rooted simple graph with the following properties:*

- *Each node (or vertex), holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).*
- *There is one node in every space tree with no parent; this is the root node of the tree.*
- *Each point in S is contained in at least one node.*
- *Each node $\mathscr{N}$ has a convex subset of $\Re^D$ containing each point in that node and also the convex subsets represented by each child of the node.*

Definition 2. *A pruning dual-tree traversal is a process that, given two space trees $\mathscr{T}_q$ (the query tree, built on the query set S_q) and $\mathscr{T}_r$ (the reference tree, built on the reference set S_r), will visit combinations of nodes $(\mathscr{N}_q, \mathscr{N}_r)$ such that $\mathscr{N}_q \in \mathscr{T}_q$ and $\mathscr{N}_r \in \mathscr{T}_r$ no more than once, and call a function `Score(`$\mathscr{N}_q$`,` $\mathscr{N}_r$`)` to assign a score to that node. If the score is ∞ (or above some bound), the combination is pruned and no combinations $(\mathscr{N}_{qc}, \mathscr{N}_{rc})$ such that $\mathscr{N}_{qc} \in \mathscr{D}_q^n$ and $\mathscr{N}_{rc} \in \mathscr{D}_r^n$ are visited. Otherwise, for every combination of points (p_q, p_r) such that $p_q \in \mathscr{P}_q$ and $p_r \in \mathscr{P}_r$, a function `BaseCase(`p_q`,` p_r`)` is called. If no node combinations are pruned during the traversal, `BaseCase(`p_q`,` p_r`)` is called at least once on each combination of $p_q \in S_q$ and $p_r \in S_r$.*

For more description and clarity on these definitions, refer to (Curtin et al., 2013b).

Lemma 1. *A pruning dual-tree traversal which uses `BaseCase()` as given in Algorithm 2 in the main paper and `Score()` as given in Algorithm 3 in the main paper which starts with valid* ub$(\cdot)$, lb$(\cdot)$, pruned$(\cdot)$, *and* closest$(\cdot)$ *for each node $\mathscr{N}_i \in \mathscr{T}$, and* ub$(p_q) =$ lb$(p_q) = \infty$ *for each point $p_q \in S$, will satisfy the following conditions upon completion:*

- *For every $p_q \in S$ that is a descendant of a node $\mathscr{N}_i$ that has been pruned (*pruned$(\mathscr{N}_i) = k - 1$*), *ub$(\mathscr{N}_i)$ *is an upper bound on the distance between p_q and its closest centroid, and* closest$(\mathscr{N}_i)$ *is the owner of p_q.*

- *For every $p_q \in S$ that is not a descendant of any node that has been pruned,* $\text{ub}(p_q)$ *is an upper bound on the distance between p_q and its closest centroid, and* $\text{closest}(p_q)$ *is the owner of p_q.*
- *For every $p_q \in S$ that is a descendant of a node $\mathscr{N}_i$ that has been pruned (*$\text{pruned}(\mathscr{N}_i) = k - 1$*),* $\text{lb}(\mathscr{N}_i)$ *is a lower bound on the distance between p_q and its second closest centroid.*
- *For every $p_q \in S$ that is not a descendant of any node that has been pruned,* $\min(\text{lb}(p_q), \text{lb}(\mathscr{N}_q))$ *where $\mathscr{N}_q$ is a node such that $p_q \in \mathscr{P}_q$ is a lower bound on the distance between p_q and its second closest centroid.*

Proof. It is easiest to consider each condition individually. Thus, we will first consider the upper bound on the distance to the closest cluster centroid. Consider some p_q and suppose that the closest cluster centroid to p_q is c^*.

Now, suppose first that the point p_q is a descendant point of a node $\mathscr{N}_q$ that has been pruned. We must show, then, that c^* is $\text{closest}(\mathscr{N}_q)$. Take $R = \{\mathscr{N}_{r0}, \mathscr{N}_{r1}, \ldots, \mathscr{N}_{rj}\}$ to be the set of reference nodes visited during the traversal with $\mathscr{N}_q$ as a query node; that is, the combinations $(\mathscr{N}_q, \mathscr{N}_{ri})$ were visited for all $\mathscr{N}_{ri} \in R$. Any $\mathscr{N}_{ri}$ is pruned only if

$$d_{\min}(\mathscr{N}_q, \mathscr{N}_{ri}) > \text{ub}(\mathscr{N}_i) \tag{11}$$

according to line 10 of `Score()`. Thus, as long as $\text{ub}(\mathscr{N}_i)$ is a valid upper bound on the closest cluster distance for every descendant point in $\mathscr{N}_q$, then no nodes are incorrectly pruned. It is easy to see that the upper bound is valid: initially, it is valid by assumption; each time the bound is updated with some node $\mathscr{N}_{ri}$ (on lines 19 and 20), it is set to $d_{\max}(\mathscr{N}_i, c)$ where c is some descendant centroid of $\mathscr{N}_{ri}$. This is clearly a valid upper bound, since c cannot be any closer to any descendant point of $\mathscr{N}_i$ than c^*. We may thus conclude that no node is incorrectly pruned from R; we may apply this reasoning recursively to the $\mathscr{N}_q$'s ancestors to see that no reference node is incorrectly pruned.

When a node is pruned from R, the number of pruned clusters for $\mathscr{N}_q$ is updated: the count of all clusters not previously pruned by $\mathscr{N}_q$ (or its ancestors) is added. We cannot double-count the pruning of a cluster; thus the only way that $\text{pruned}(\mathscr{N}_q)$ can be equal to $k - 1$ is if every centroid except one is pruned. The centroid which is not pruned will be the nearest centroid c^*, regardless of if $\text{closest}(\mathscr{N}_q)$ was set during this traversal or still holds its initial value, and therefore it must be true that $\text{ub}(\mathscr{N}_q)$ is an upper bound on the distance between p_q and c^*, and $\text{closest}(\mathscr{N}_q) = c^*$.

This allows us to finally conclude that if p_q is a descendant of a node $\mathscr{N}_q$ that has been pruned, then $\text{ub}(\mathscr{N}_q)$ contains a valid upper bound on the distance between p_q and its closest cluster centroid, and $\text{closest}(\mathscr{N}_q)$ is that closest cluster centroid.

Now, consider the other case, where p_q is not a descendant of any node that has been pruned. Take $\mathscr{N}_i$ to be any node containing p_q[4]. We have already reasoned that any cluster centroid node that could possibly contain the closest cluster centroid to p_q cannot have been pruned; therefore, by the definition of pruning dual-tree traversal, we are guaranteed that `BaseCase()` will be called with p_q as the query point and the closest cluster centroid as the reference point. This will then cause $\text{ub}(p_q)$ to hold the distance to the closest cluster centroid—assuming $\text{ub}(p_q)$ is always valid, which it is even at the beginning of the traversal because it is initialized to ∞—and $\text{closest}(p_q)$ to hold the closest cluster centroid.

Therefore, the first two conditions are proven. The third and fourth conditions, for the lower bounds, require a slightly different strategy.

There are two ways $\text{lb}(\mathscr{N}_q)$ is modified: first, at line 14, when a node combination is pruned, and second, at line 6 when the lower bound is taken from the parent. Again, consider the set $R = \{\mathscr{N}_{r0}, \mathscr{N}_{r1}, \ldots, \mathscr{N}_{rj}\}$ which is the set of reference nodes visited during the traversal with $\mathscr{N}_q$ as a query node. Call the set of reference nodes that were pruned R^p. At the end of the traversal, then,

$$\begin{aligned} \text{lb}(\mathscr{N}_q) &\leq \min_{\mathscr{N}_{ri} \in R^p} d_{\min}(\mathscr{N}_q, \mathscr{N}_{ri}) && (12) \\ &\leq \min_{c_k \in C^p} d_{\min}(\mathscr{N}_q, c_k) && (13) \end{aligned}$$

where C^p is the set of centroids that are descendants of nodes in R^p. Applying this reasoning recursively to the ancestors of $\mathscr{N}_q$ shows that at the end of the dual-tree traversal, $\text{lb}(\mathscr{N}_q)$ will contain a lower bound on the distance between any descendant point of $\mathscr{N}_q$ and any pruned centroid. Thus, if $\text{pruned}(\mathscr{N}_q) = k-1$, then $\text{lb}(\mathscr{N}_q)$ will contain a lower bound on the distance between any descendant point in $\mathscr{N}_q$ and its second closest centroid. So if we consider some point p_q which is a descendant of $\mathscr{N}_q$ and $\mathscr{N}_q$ is pruned ($\text{pruned}(\mathscr{N}_q) = k - 1$), then $\text{lb}(\mathscr{N}_q)$ is indeed a lower bound on the

[4]Note that the meaning here is not that p_q is a descendant of $\mathscr{N}_i$ ($p_i \in \mathscr{D}_i^p$), but instead that p_q is held directly in $\mathscr{N}_i$: $p_q \in \mathscr{P}_i$.

distance between p_q and its second closest centroid.

Now, consider the case where p_q is not a descendant of any node that has been pruned, and take $\mathscr{N}_q$ to be some node that owns p_q (that is, $p_q \in \mathscr{P}_q$). In this case, `BaseCase()` will be called with every centroid that has not been pruned. So $\text{lb}(\mathscr{N}_q)$ is a lower bound on the distance between p_q and every pruned centroid, and $\text{lb}(p_q)$ will be a lower bound on the distance between p_q and the second-closest non-pruned centroid, due to the structure of the `BaseCase()` function. Therefore, $\min(\text{lb}(p_q), \text{lb}(\mathscr{N}_q))$ must be a lower bound on the distance between p_q and its second closest centroid.

Finally, we may conclude that each item in the theorem holds. □

Next, we must prove that `UpdateTree()` functions correctly.

Lemma 2. *In the context of Algorithm 1 in the main paper, given a tree $\mathscr{T}$ with all associated bounds* $\text{ub}(\cdot)$ *and* $\text{lb}(\cdot)$ *and information* $\text{pruned}(\cdot)$, $\text{closest}(\cdot)$, *and* $\text{canchange}(\cdot)$, *a run of `UpdateTree()` as given in Algorithm 5 will have the following effects:*

- *For every node $\mathscr{N}_i$,* $\text{ub}(\mathscr{N}_i)$ *will be a valid upper bound on the distance between any descendant point of $\mathscr{N}_i$ and its nearest centroid next iteration.*
- *For every node $\mathscr{N}_i$,* $\text{lb}(\mathscr{N}_i)$ *will be a valid lower bound on the distance between any descendant point of $\mathscr{N}_i$ and any pruned centroid next iteration.*
- *A node $\mathscr{N}_i$ will only have* $\text{canchange}(\mathscr{N}_i) =$ `false` *if the owner of any descendant point of $\mathscr{N}_i$ cannot change next iteration.*
- *A point p_i will only have* $\text{canchange}(p_i) =$ `false` *if the owner of p_i cannot change next iteration.*
- *Any point p_i with* $\text{canchange}(p_i) =$ `true` *that does not belong to any node $\mathscr{N}_i$ with* $\text{canchange}(\mathscr{N}_i) =$ `false` *will have* $\text{ub}(p_i) = \text{lb}(p_i) = \infty$, *as required by the dual-tree traversal.*
- *Any node $\mathscr{N}_i$ with* $\text{canchange}(\mathscr{N}_i) =$ `false` *at the end of `UpdateTree()` will have* $\text{pruned}(\mathscr{N}_i) = 0$.

Proof. Each point is best considered individually. It is important to remember during this proof that the centroids have been updated, but the bounds have not. So any cluster centroid c_i is already set for next iteration. Take c_i^l to mean the cluster centroid c_i *before* adjustment (that is, the old centroid). Also take $\text{ub}^l(\cdot)$, $\text{lb}^l(\cdot)$, $\text{pruned}^l(\cdot)$, and $\text{canchange}^l(\cdot)$ to be the values at the time `UpdateTree()` is called, before any of those values are changed. Due to the assumptions in the statement of the lemma, each of these quantities is valid.

Suppose that for some node $\mathscr{N}_i$, $\text{closest}(\mathscr{N}_i)$ is some cluster c_j. For $\text{ub}(\mathscr{N}_i)$ to be valid for next iteration, we must guarantee that $\text{ub}(\mathscr{N}_i) \geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j)$ at the end of `UpdateTree()`. There are four ways $\text{ub}(\mathscr{N}_i)$ is updated: it may be taken from the parent and adjusted (line 8), it may be adjusted before a prune attempt (line 14), it may be tightened after a failed prune attempt (line 21), or it may be adjusted without a prune attempt (line 25). If we can show that each of these four ways always results in $\text{ub}(\mathscr{N}_i)$ being valid, then the first condition of the theorem holds.

If $\text{ub}(\mathscr{N}_i)$ is adjusted in line 14 or 25, the resulting value of $\text{ub}(\mathscr{N}_i)$, assuming $\text{closest}(\mathscr{N}_i) = c_j$, is

$$\begin{aligned} \text{ub}(\mathscr{N}_i) &= \text{ub}^l(\mathscr{N}_i) + m_j & (14)\\ &\geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j^l) + m_j & (15)\\ &\geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j) & (16) \end{aligned}$$

where the last step follows by the triangle inequality: $d(c_j, c_j^l) = m_j$. Therefore those two updates to $\text{ub}(\mathscr{N}_i)$ result in valid upper bounds for next iteration. If $\text{ub}(\mathscr{N}_i)$ is recalculated, in line 21, then we are guaranteed that $\text{ub}(\mathscr{N}_i)$ is valid because

$$d_{\max}(\mathscr{N}_i, c_j) \geq \max_{p_q \in \mathscr{D}_q^p} d(p_q, c_j). \quad (17)$$

We may therefore conclude that $\text{ub}(\mathscr{N}_i)$ is correct for the root of the tree, because line 8 can never be reached. Reasoning recursively, we can see that any upper bound passed from the parent must be valid. Therefore, the first item of the lemma holds.

Next, we will consider the lower bound, using a similar strategy. We must show that

$$\text{lb}(\mathscr{N}_i) \leq \min_{p_q \in \mathscr{D}_q^p} \min_{c_p \in C_p} d(p_q, c_p) \quad (18)$$

where C_p is the set of centroids pruned by $\mathscr{N}_i$ and ancestors during the last dual-tree traversal. The lower bound can be taken from the parent in line 9 and adjusted, it can be adjusted before a prune attempt in line 15 or in a similar way without a prune attempt in line 26. The last adjustment can easily be shown to be valid:

$$\begin{aligned} \text{lb}(\mathscr{N}_i) &= \text{lb}^l(\mathscr{N}_i) - \max_k m_k & (19) \\ &\leq \left(\min_{p_q \in \mathscr{D}_q^p} \min_{c_p \in C_p} d(p_q, c_p^l) \right) - \max_k m_k & (20) \\ &\leq \min_{p_q \in \mathscr{D}_q^p} \min_{c_p \in C_p} d(p_q, c_p) & (21) \end{aligned}$$

which follows by the triangle inequality: $d(c_p^l, c_p) \leq \max_k m_k$. Line 15 is slightly more complex; we must also consider the term $\min_{k \neq j} d(c_k, c_j)/2$. Suppose that

$$\min_{k \neq j} d(c_k, c_j)/2 > \text{lb}^l(\mathscr{N}_i) + \max_k m_k. \quad (22)$$

We may use the triangle inequality ($d(p_q, c_k) \leq d(c_j, c_k) + d(p_q, c_j)$) to show that if this is true, the second closest centroid c_k is such that $d(p_q, c_k) > 2d(c_k, c_j)$ and therefore $\min_{k \neq j} d(c_k, c_j)/2$ is also a valid lower bound. We can lastly use the same recursive argument from the upper bound case to show that the second item of the lemma holds.

Showing the correctness of canchange($\mathscr{N}_i$) is straightforward: we know that ub($\mathscr{N}_i$) and lb($\mathscr{N}_i$) are valid for next iteration by the time any checks to set canchange($\mathscr{N}_i$) to `false` happens, due to the discussion above. The situations where canchange($\mathscr{N}_i$) is set to `false`, in line 18 and 22, are simply applications of Equations 4 and 6 in the main paper, and are therefore valid. There are two other ways canchange($\mathscr{N}_i$) can be set to `false`. The first is on line 10, and this is easily shown to be valid: if a parent's owner cannot change, then a child's owner cannot change either. The other way to set canchange($\mathscr{N}_i$) to `false` is in line 53. This is only possible if all points in $\mathscr{P}_i$ and all children of $\mathscr{N}_i$ have canchange($\cdot$) set to `false`; thus, no descendant point of $\mathscr{N}_i$ can change owner next iteration, and we may set canchange($\mathscr{N}_i$) to `false`.

Next, we must show that canchange(p_i) = `false` only if the owner of p_i cannot change next iteration. If canchange$^l(p_i)$ = `true`, then due to Lemma 1, ub$^l(p_i)$ and lb$^l(p_i)$ will be valid bounds. In this case, we may use similar reasoning to show that ub(p_i) and lb(p_i) are valid, and then we may see that the pruning attempts at line 35 and 40 are valid. Now, consider the other case, where canchange$^l(p_i)$ = `false`. Then, ub$^l(p_i)$ and lb$^l(p_i)$ will not have been modified by the dual-tree traversal, and will hold the values set in the previous run of `UpdateTree()`. As long as those values are valid, then the fourth item holds.

The checks to see if canchange(p_i) can be set to `false` (from lines 31 to 45) are only reached if canchange($\mathscr{N}_i$) is `true`. We already have shown that ub(p_i) and lb(p_i) are set correctly in that stanza. The other case is if canchange($\mathscr{N}_i$) is `false`. In this case, lines 47 to 51 are reached. It is easy to see using similar reasoning to all previous cases that these lines result in valid ub(p_i) and lb(p_i). Therefore, the fourth item does hold.

The fifth item is taken care of in line 44 and 45. Given some point p_i with canchange(p_i) = `true`, and where p_i does not belong to any node $\mathscr{N}_i$ where canchange($\mathscr{N}_i$) = `false`, these two lines must be reached, and therefore the fifth item holds.

The last item holds trivially—any node $\mathscr{N}_i$ where canchange($\mathscr{N}_i$) = `true` has pruned($\mathscr{N}_i$) set to 0 on line 55. □

Showing that `CoalesceTree()`, `DecoalesceTree()`, and `UpdateCentroids()` function correctly follows directly from the algorithm descriptions. Therefore, we are ready to show the main correctness result.

Theorem 4. *A single iteration of dual-tree k-means as given in Algorithm 1 in the main paper will produce exactly the same results as the standard brute-force $O(kN)$ implementation.*

Proof. We may use the previous lemmas to flesh out our earlier proof sketch.

First, we know that the dual-tree algorithm (line 9) produces correct results for ub($\cdot$), lb($\cdot$), pruned($\cdot$), and closest($\cdot$) for every point and node, due to Lemma 1. Next, we know that `UpdateTree()` maintains the correctness of those four quantities and only marks canchange($\cdot$) to `false` when the node or point truly cannot change owner, due to Lemma 2. Next, we know from earlier discussion that `CoalesceTree()` and `DecoalesceTree()` do not affect the results of the dual-tree algorithm because the only nodes and points removed are those where canchange($\cdot$) = `false`. We also know that `UpdateCentroids()` produces centroids correctly. Therefore, the results from Algorithm 1 in the main paper are identical to those of a brute-force $O(kN)$ k-means implementation. □

A.4. Runtime bound proof

We can use adaptive algorithm analysis techniques in order to bound the running time of Algorithm 1 in the main paper, based on (Curtin et al., 2015) and (Beygelzimer et al., 2006). This analysis depends on the *expansion constant*, which is a measure of intrinsic dimension defined below, originally from (Karger & Ruhl, 2002).

Definition 3. *Let $B_S(p, \Delta)$ be the set of points in S*

within a closed ball of radius Δ around some $p \in S$ with respect to a metric d:

$$B_S(p, \Delta) = \{r \in S : d(p, r) \leq \Delta\}. \quad (23)$$

Then, the **expansion constant** *of S with respect to the metric d is the smallest $c \geq 2$ such that*

$$|B_S(p, 2\Delta)| \leq c|B_S(p, \Delta)| \ \forall \ p \in S, \ \forall \ \Delta > 0. \quad (24)$$

The expansion constant is a bound on the number of points which fall into balls of increasing sizes. A low expansion constant generally means that search tasks like nearest neighbor search can be performed quickly with trees, whereas a high expansion constant implies a difficult dataset. Thus, if we assume a bounded expansion constant like in previous theoretical works (Beygelzimer et al., 2006; Ram et al., 2009a; Karger & Ruhl, 2002; Curtin & Ram, 2014; Curtin et al., 2015), we may assemble a runtime bound that reflects the difficulty of the dataset.

Our theoretical analysis will concern the cover tree in particular. The cover tree is a complex data structure with appealing theoretical properties. We will only summarize the relevant properties here. Interested readers should consult the original cover tree paper (Beygelzimer et al., 2006) and later analyses (Ram et al., 2009a; Curtin et al., 2015) for a complete understanding.

A cover tree is a leveled tree; that is, each cover tree node $\mathscr{N}_i$ is associated with an integer scale s_i. The node with largest scale is the root of the tree; each node's scale is greater than its children's. Each node $\mathscr{N}_i$ holds one point p_i, and every descendant point of $\mathscr{N}_i$ is contained in the ball centered at p_i with radius 2^{s_r+1}. Further, every cover tree satisfies the following three invariants (Beygelzimer et al., 2006):

- *(Nesting.)* When a point p_i is held in a node at some scale s_i, then each smaller scale will also have a node containing p_i.
- *(Covering tree.)* For every point p_i held in a node $\mathscr{N}_i$ at scale s_i, there exists a node with point p_j and scale $s_i + 1$ which is the parent of $\mathscr{N}_i$, and $d(p_i, p_j) < 2^{s_i+1}$.
- *(Separation.)* Given distinct nodes $\mathscr{N}_i$ holding p_i and $\mathscr{N}_j$ holding p_j both at scale s_i, $d(p_i, p_j) > 2^{s_i}$.

A useful result shows there are $O(N)$ points in a cover tree (Theorem 1, (Beygelzimer et al., 2006)). Another measure of importance of a cover tree is the *cover tree imbalance*, which aims to capture how well the data is distributed throughout the cover tree. For instance, consider a tree where the root, with scale s_r, has two nodes; one node corresponds to a single point and has scale $-\infty$, and the other node has scale $s_r - 1$ and contains every other point in the dataset as a descendant. This is very imbalanced, and a tree with many situations like this will not perform well for search tasks. Below, we reiterate the definition of cover tree imbalance from (Curtin et al., 2015).

Definition 4. *The cover node imbalance $i_n(\mathscr{N}_i)$ for a cover tree node $\mathscr{N}_i$ with scale s_i in the cover tree $\mathscr{T}$ is defined as the cumulative number of missing levels between the node and its parent $\mathscr{N}_p$ (which has scale s_p). If the node is a leaf child (that is, $s_i = -\infty$), then number of missing levels is defined as the difference between s_p and $s_{\min} - 1$ where $s_{\min}$ is the smallest scale of a non-leaf node in $\mathscr{T}$. If $\mathscr{N}_i$ is the root of the tree, then the cover node imbalance is 0. Explicitly written, this calculation is*

$$i_n(\mathscr{N}_i) = \begin{cases} s_p - s_i - 1 & \text{if } \mathscr{N}_i \text{ is not a leaf and not the root node} \\ \max(s_p - s_{\min} - 1, \ 0) & \text{if } \mathscr{N}_i \text{ is a leaf} \\ 0 & \text{if } \mathscr{N}_i \text{ is the root.} \end{cases} \quad (25)$$

This simple definition of cover node imbalance is easy to calculate, and using it, we can generalize to a measure of imbalance for the full tree.

Definition 5. *The cover tree imbalance $i_t(\mathscr{T})$ for a cover tree $\mathscr{T}$ is defined as the cumulative number of missing levels in the tree. This can be expressed as a function of cover node imbalances easily:*

$$i_t(\mathscr{T}) = \sum_{\mathscr{N}_i \in \mathscr{T}} i_n(\mathscr{N}_i). \quad (26)$$

Bounding $i_t(\mathscr{T})$ is non-trivial, but empirical results suggest that imbalance scales linearly with the size of the dataset, when the expansion constant is well-behaved. A bound on $i_t(\mathscr{T})$ is still an open problem at the time of this writing.

With these terms introduced, we may introduce a slightly adapted result from (Curtin et al., 2015), which bounds the running time of nearest neighbor search.

Theorem 5. *(Theorem 2, (Curtin et al., 2015).) Using cover trees, the standard cover tree pruning dual-tree traversal, and the nearest neighbor search*

`BaseCase()` and `Score()` as given in Algorithms 2 and 3 of (Curtin et al., 2015), respectively, and also given a reference set S_r with expansion constant c_r, and a query set S_q, where the range of pairwise distances in S_r is completely contained in the range of pairwise distances in S_q, the running time of nearest neighbor search is bounded by $O(c_r^4 c_{qr}^5 (N + i_t(\mathscr{T}_q)))$, where $c_{qr} = \max((\max_{p_q \in S_q} c'_r), c_r)$, where c'_r is the expansion constant of the set $S_r \cup \{p_q\}$.

Now, we may adapt this result slightly.

Theorem 6. *The dual-tree k-means algorithm with `BaseCase()` as in Algorithm 2 in the main paper and `Score()` as in Algorithm 3 in the main paper, with a point set S_q that has expansion constant c_q and size N, and k centroids C with expansion constant c_k, takes no more than $O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T}_q)))$ time.*

Proof. Both `Score()` and `BaseCase()` for dual-tree k-means can be performed in $O(1)$ time. In addition, the pruning of `Score()` for dual-tree k-means is at least as tight as `Score()` for nearest neighbor search: the pruning rule in Equation 2 in the main paper is equivalent to the pruning rule for nearest neighbor search. Therefore, dual-tree k-means can visit no more nodes than nearest neighbor search would with query set S_q and reference set C. Lastly, note that the range of pairwise distances of C will be entirely contained in the range of pairwise distances in S_q, to see that we can use the result of Theorem 5. Adapting that result, then, yields the statement of the algorithm. □

The expansion constant of the centroids, c_k, may be understood as the intrinsic dimensionality of the centroids C. During each iteration, the centroids change, so those iterations that have centroids with high intrinsic dimensionality cannot be bounded as tightly. More general measures of intrinsic dimensionality, such as those recently proposed by Houle (Houle, 2013), may make the connection between c_q and c_k clear.

Next, we turn to bounding the entire algorithm.

Theorem 7. *A single iteration of the dual-tree k-means algorithm on a dataset S_q using the cover tree $\mathscr{T}$, the standard cover tree pruning dual-tree traversal, `BaseCase()` as given in Algorithm 2 in the main paper, `Score()` as given in Algorithm 3 in the main paper, will take no more than*

$$O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})) + c_k^9 k \log k) \tag{27}$$

time, where c_k is the expansion constant of the centroids, c_{qk} is defined as in Theorem 6, and $i_t(\mathscr{T})$ is the imbalance of the tree as defined in Definition 5.

Proof. Consider each of the steps of the algorithm individually:

- `CoalesceNodes()` can be performed in a single pass of the cover tree $\mathscr{N}$, which takes $O(N)$ time.
- Building a tree on the centroids ($\mathscr{T}_c$) takes $O(c_k^6 k \log k)$ time due to the result for cover tree construction time (Beygelzimer et al., 2006).
- The dual-tree algorithm takes $O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})))$ time due to Theorem 6.
- `DecoalesceNodes()` can be performed in a single pass of the cover tree $\mathscr{N}$, which takes $O(N)$ time.
- `UpdateCentroids()` can be performed in a single pass of the cover tree $\mathscr{N}$, so it also takes $O(N)$ time.
- `UpdateTree()` depends on the calculation of how much each centroid has moved; this costs $O(k)$ time. In addition, we must find the nearest centroid of every centroid; this is nearest neighbor search, and we may use the runtime bound for monochromatic nearest neighbor search for cover trees from (Ram et al., 2009a), so this costs $O(c_k^9 k)$ time. Lastly, the actual tree update visits each node once and iterates over each point in the node. Cover tree nodes only hold one point, so each visit costs $O(1)$ time, and with $O(N)$ nodes, the entire update process costs $O(N)$ time. When we consider the preprocessing cost too, the total cost of `UpdateTree()` per iteration is $O(c_k^9 k + N)$.

We may combine these into a final result:

$$O(N) + O(c_k^6 k \log k) + O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T}))) + O(N) + O(N) + O(c_k^9 k + N) \tag{28}$$

and after simplification, we get the statement of the theorem:

$$O(c_k^4 c_{qk}^5 (N + i_t(\mathscr{T})) + c_k^9 k \log k). \tag{29}$$

□

Therefore, we see that under some assumptions on the data, we can bound the runtime of the dual-tree k-means algorithm to something tighter than $O(kN)$ per iteration. As expected, we are able to amortize the cost of k across all N nodes, giving amortized $O(1)$ search for the nearest centroid per point in the dataset. This is similar to the results for nearest neighbor search, which obtain amortized $O(1)$ search for a

single query point. Also similar to the results for nearest neighbor search is that the search time may, in the worst case, degenerate to $O(kN + k^2)$ when the assumptions on the dataset are not satisfied. However, empirical results (Ram et al., 2009b; Gray & Moore, 2001; March et al., 2010; Beygelzimer et al., 2006) show that well-behaved datasets are common in the real world, and thus degeneracy of the search time is uncommon.

Comparing this bound with the bounds for other algorithms is somewhat difficult; first, none of the other algorithms have bounds which are adaptive to the characteristics of the dataset. It is possible that the blacklist algorithm could be refactored to use the cover tree, but even if that was done it is not completely clear how the running time could be bounded. How to apply the expansion constant to an analysis of Hamerly's algorithm and Elkan's algorithm is also unclear at the time of this writing.

Lastly, the bound we have shown above is potentially loose. We have reduced dual-tree k-means to the problem of nearest neighbor search, but our pruning rules are tighter. Dual-tree nearest neighbor search assumes that every query node will be visited (this is where the $O(N)$ in the bound comes from), but dual-tree k-means can prune a query node entirely if all but one cluster is pruned (Strategy 2). These bounds do not take this pruning strategy into account, and they also do not consider the fact that coalescing the tree can greatly reduce its size. These would be interesting directions for future theoretical work.