Let them have CAKES: A Cutting-Edge Algorithm for Scalable, Efficient, and Exact Search on Big Data[*]

Morgan E. Prior[†], Thomas J. Howard III[‡], Oliver McLaughlin[‡], Terrence Ferguson[‡], Najib Ishaq[‡], and Noah M. Daniels[‡]

Abstract. The Big Data explosion has created a demand for efficient and scalable algorithms for similarity search. While much recent work has focused on *approximate* k-NN search, *exact* k-NN search has not kept up. We present CAKES, a set of three novel algorithms for exact k-NN search. CAKES's algorithms are generic over *any* distance function, and do not scale with the cardinality or embedding dimension of the dataset. Instead, they scale with geometric properties of the dataset–namely, metric entropy and fractal dimension–thus providing immense speed improvements over existing exact k-NN search algorithms when the dataset conforms to the manifold hypothesis. We demonstrate these claims by contrasting performance on a randomly-generated dataset against that on some datasets from the ANN-Benchmarks suite under commonly-used distance functions, a genomic dataset under Levenshtein distance, and a radio-frequency dataset under Dynamic Time Warping distance. CAKES exhibits near-constant running time on data conforming to the manifold hypothesis as cardinality grows, and has perfect recall on data in metric spaces. CAKES also has significantly higher recall than state-of-the-art k-NN search algorithms even when the distance function is not a metric. We conclude that CAKES is a highly efficient and scalable algorithm for exact k-NN search on Big Data. We provide a Rust implementation of CAKES under an MIT license at https://github.com/URI-ABD/clam.

MSC codes. 68P05, 68P10

1. Introduction.

Researchers are collecting data at an unprecedented rate. In many fields, datasets are growing exponentially, and this increase in the rate of data collection outpaces improvements in computing performance as predicted by Moore's Law [19]. This indicates that the performance of computing systems will not keep pace with the growth of data. Often dubbed "the Big Data explosion," this phenomenon has created a need for better algorithms to analyze large datasets.

Examples of large datasets include genomic databases, time-series data such as radio frequency signals, and neural network embeddings. Large language models such as GPT [6, 27] and LLAMA-2 [34], and image embedding models [30, 9] are a common source of neural network embeddings. Among biological datasets, SILVA 18S [29] contains ribosomal DNA sequences of approximately 2.25 million genomes; the longest of these individual sequences is 3,712 amino acids, but in a multiple sequence alignment, the length becomes 50,000 letters. Among time-series datasets, the RadioML dataset [28] contains approximately 2.55 million samples of synthetically-generated signals of different modulation modes.

Many researchers are interested in similarity search on these datasets. Similarity search enables a variety of applications, including recommendation [4] and classification systems [33]. As the cardinalities and dimensionalities of datasets have grown, however, efficient and accurate

similarity search has become challenging; even state-of-the-art algorithms exhibit a steep tradeoff between recall and throughput [24, 18, 4, 2].

Given some measure of similarity between data points, there are two common definitions of similarity search: k-nearest neighbor search (k-NN) and ρ-nearest neighbor search (ρ-NN). k-NN search aims to find the k most similar points to a query, while ρ-NN search aims to find all points within a similarity threshold ρ of a query. Previous works have used the term *approximate* search to refer to ρ-NN search, but in this paper, we reserve the term *approximate* for algorithms that do not exhibit perfect recall. In contrast, an *exact* search algorithm exhibits perfect recall.

k-NN search is one of the most ubiquitous classification and recommendation methods in use [11, 7]. Naïve implementations of k-NN search, whose time complexity is linear in the dataset's cardinality, prove prohibitively slow for large datasets. While fast algorithms for k-NN search on large datasets exist, they are often approximate [12], and while approximate search may be sufficient for some applications, the need for efficient and *exact* search remains [35]. For example, for a majority voting classifier, approximate k-NN search may agree with exact k-NN search for large values of k, but may be sensitive to local perturbations for smaller values of k. This is especially true when classes are not well-separated [38]. Further, there is evidence that distance functions that do not obey the triangle inequality, such as cosine distance, perform poorly for k-NN search in biomedical settings [15]; this suggests that approximate k-NN search could perform poorly in such contexts.

This paper introduces CAKES (CLAM-Accelerated K-NN Entropy-Scaling Search), a set of three novel algorithms for exact k-NN search. We also compare CAKES against several current algorithms; namely, FAISS [18], HNSW [24], and ANNOY [4], on datasets from the ANN-benchmarks suite [2]. We further benchmark CAKES on a large genomic dataset, the SILVA 18S dataset [29], using Levenshtein [23] distance on unaligned genomic sequences, and a radio frequency dataset, RadioML [28], using Dynamic Time Warping (DTW) [13] distance on complex-valued time-series. Finally, we compare real-world datasets to an artificially-generated dataset obeying simple statistical properties.

1.1. Related Works. Recent k-nearest neighbor search algorithms designed to scale with the exponential growth of data include Hierarchical Navigable Small World networks (HNSW) [24], InVerted File indexing (FAISS-IVF) [1], random projection and tree building (ANNOY) [4], and entropy-scaling search [37, 17]. However, some of these algorithms do not provide exact search (as defined in Section 1 above).

Hierarchical Navigable Small World networks [24] rely on navigable small world (NSW) networks [22, 5] and skip lists. HNSW builds a multi-layered graph of the dataset. A query point and each data point are inserted into the graph and joined by an edge to the M nearest nodes in the in the graph, where M is a tunable parameter. The highest layer in which an element can be placed is determined randomly with an exponentially-decaying probability distribution. Search starts at the highest layer and descends to the lowest layer, greedily following a path of edges to the nearest node, until reaching the query point. To improve accuracy, the $efSearch$ hyperparameter can be changed to specify the number of closest nearest neighbors to the query vector to be found at each layer.

InVerted File indexing (IVF) [1, 31, 20] clusters data into high-dimensional Voronoi cells,

82 and whichever cell a query point falls into is then searched exhaustively, similarly to an early
83 precursor to our work [37]. The number of cells used is governed by the n_{list} parameter.
84 Increasing this parameter decreases the number of points being exhaustively searched, so it
85 improves speed at the cost of accuracy. To mitigate accuracy issues caused by a query point
86 falling near a cell boundary, the algorithm has a tunable parameter n_{probe} that specifies the
87 number of additional adjacent or nearby cells to search.
88   ANNOY [4] relies on random projection and tree building for approximate k-NN search.
89 At each intermediate node of the tree, two points are randomly sampled from the space, and
90 the hyperplane equidistant from them is chosen to divide the space into two subspaces. This
91 process iterates to create a forest of trees, and the number of iterations is a tunable parameter.
92 At search time, one can increase the number of trees to be searched to improve recall at the
93 cost of speed.

94   **1.2. Entropy-Scaling Search.** The entropy-scaling search paradigm exploits the geometric
95 and topological structure inherent in large datasets. Importantly, as suggested by their name,
96 entropy-scaling search algorithms exhibit asymptotic complexity that scales with topological
97 properties of the dataset, instead of its cardinality, when the dataset has a manifold structure.
98 In 2019, we introduced CHESS (Clustered Hierarchical Entropy-Scaling Search) [17], which
99 extended entropy-scaling ρ-NN search from a flat clustering approach to a hierarchical clustering
100 approach. CLAM (Clustering, Learning and Approximation with Manifolds), originally devel-
101 oped to allow "manifold mapping" for anomaly detection [16], is a refinement of the clustering
102 algorithm from CHESS. In this paper, we introduce CAKES, a set of three entropy-scaling
103 algorithms for k-NN search, implemented in the Rust programming language.
104   We also provide a theoretical analysis of the time complexity of CAKES's algorithms in
105 Sections 2.2.2 and 2.2.5. These analyses are not worst-case analyses in the traditional sense,
106 as they do not assume the worst possible dataset, namely, a uniform distribution. Given that
107 CAKES's algorithms are intended to be used on datasets with a manifold structure, complexity
108 analysis assuming a uniform distribution of data would be uninformative. As a result, we
109 assume in our analyses that the dataset has a manifold structure, as quantified by low fractal
110 dimension and metric entropy. For our purposes, it makes more sense to analyze the expected
111 performance of CAKES's algorithms under these assumptions.

112   **2. Methods.** In this manuscript, we are primarily concerned with k-NN search in a finite-
113 dimensional space. Given a dataset $\mathbf{X} = \{x_1 \ldots x_n\}$ of cardinality $|\mathbf{X}| = n$, we define a *point*
114 or *datum* $x_i \in \mathbf{X}$ as a singular observation. Examples include the neural-network embedding of
115 an image, the genome of an organism, and a measurement of a radio frequency signal.
116   We define a *distance function* $f : \mathbf{X} \times \mathbf{X} \mapsto \mathbb{R}^+ \cup \{0\}$ which, given two points, de-
117 terministically returns a finite and non-negative real number. A distance value of zero
118 defines an identity among points (i.e., $f(x,y) = 0 \Leftrightarrow x = y$) and larger values indicate
119 greater dissimilarity among points. We also require that the distance function be symmetric
120 (i.e., $f(x,y) = f(y,x) \ \forall \ x, y \in \mathbf{X}$). In addition to these constraints, if the distance function
121 obeys the triangle inequality (i.e., $f(x,y) \leq f(x,z) + f(z,y) \ \forall \ x, y, z \in \mathbf{X}$), then it is also a
122 *distance metric*. Similar to [37], when used with distance metrics, all search algorithms in
123 CAKES are exact. For example, Euclidean, Levenshtein [23] and Dynamic Time Warping
124 (DTW) [25] distances are all distance metrics, while cosine distance is not a metric because it

125 violates the triangle inequality (e.g., consider the points $x = (1,0)$, $y = (0,1)$ and $z = (1,1)$ on
126 the Cartesian plane).
127   The choice of an appropriate distance function varies by dataset and domain. For example,
128 with neural-network embeddings, one could use Euclidean or Cosine distance. With genomic or
129 proteomic sequence data, Levenshtein and Hamming distances are useful. With time-series
130 data, one could use Dynamic Time Warping (DTW) or Wasserstein distance.
131   CAKES derives its advantages from the manifold hypothesis [10], the notion that high-
132 dimensional data collected from constrained generating phenomena typically only occupy a
133 low-dimensional manifold within their embedding space. We say that such data are *manifold-*
134 *constrained* and have low *local fractal dimension* (LFD). In other words, we assume that the
135 dataset is embedded in a D-dimensional space, but that the data only occupy a d-dimensional
136 manifold, where $d \ll D$. While we sometimes use Euclidean notions to describe the geometric
137 and topological properties of the clusters and manifold, CLAM and CAKES do not rely on such
138 notions; they serve merely as convenient and intuitive vocabulary to discuss the underlying
139 mathematics. CAKES exploits the low LFD of such datasets to accelerate search. We define
140 LFD at some length scale around a point in the dataset as:

141 (2.1)
$$\text{LFD}(q, r_1, r_2) = \frac{\log\left(\frac{|B(q,r_1)|}{|B(q,r_2)|}\right)}{\log\left(\frac{r_1}{r_2}\right)}$$

142 where $B(q, r)$ is the set of points contained in the metric ball of radius r centered at a point q
143 in the dataset $\mathbf{X}$.
144   We use a simplified version of Equation 2.1 by using a length scale where $r_1 = 2 \cdot r_2$.

145 (2.2)
$$\text{LFD}(q, r) = \log_2\left(\frac{|B(q,r)|}{|B(q, \frac{r}{2})|}\right)$$

146   Intuitively, LFD measures the rate of change in the number of points in a ball of radius
147 r around a point q as r increases. When the vast majority of points in the dataset have low
148 ($\ll D$) LFD, we can simply say that the dataset has low LFD. We stress that this concept
149 differs from the *embedding dimension* of a dataset. To illustrate the difference, consider the
150 SILVA 18S rRNA dataset that contains genomic sequences with unaligned lengths of up to
151 3,712 base pairs and aligned length of 50,000 base pairs. Hence, the *embedding dimension* of
152 this dataset is at least 3,712 and at most 50,000; as used in this paper it is 3,712. However,
153 physical constraints (namely, biological evolution and biochemistry) constrain the data to a
154 lower-dimensional manifold within this space. LFD is an approximation of the dimensionality
155 of that lower-dimensional manifold in the "vicinity" of a given point. Section 4.1 discusses this
156 concept on a variety of datasets, showing how real datasets uphold the manifold hypothesis.
157 For real-world datasets, we expect the LFD to be locally uniform, i.e., when r is small, but
158 potentially highly variable at larger length scales, i.e., when r is large.

159   **2.1. Clustering.** We define a *cluster* as a set of points with a *center* and a *radius*. The
160 center is the geometric median of the points in the cluster, i.e., it is the point that minimizes the
161 sum of distances to all other points in the cluster. In cases where the cardinality of the cluster
162 is large, we take a random subsample of $\sqrt{|C|}$ points and compute the geometric median of

that subsample [17]. The center, therefore, is one of the points in the cluster and is used as a representative of the cluster. The radius is the maximum distance from the center to any point in the cluster. Each non-leaf cluster has two child clusters in much the same way that a node in a binary tree has two child nodes. Note that clusters can have overlapping volumes and, in such cases, points in the overlapping volume are assigned to exactly one of the overlapping clusters. As a consequence, a cluster can be a proper subset of the metric ball at the same center and radius, i.e., $C(c,r) \subset B(c,r)$. We denote the cluster tree by $\mathcal{T}$ and the root by $\mathcal{R}$ (see Section 2.1.1).

Hereafter, when we refer to the LFD of a cluster, it is estimated at the length scale of the cluster radius and half that radius, i.e., using Equation 2.2. We also only use points that are in $C(c,r)$ instead of all those in $B(c,r)$.

The *metric entropy* $\mathcal{N}_r(X)$ for some radius r is the minimum number of clusters of a uniform radius r needed to cover the data [37] for a flat clustering with clusters of radius r. In this paper, where the clustering is hierarchical rather than flat, we define the metric entropy $\mathcal{N}_{\hat{r}}(X)$ as the number of leaf clusters in the tree where $\hat{r}$ is the mean radius of all leaf clusters.

2.1.1. Building the Tree. We start by performing a divisive hierarchical clustering on the dataset using CLAM to obtain a cluster tree $\mathcal{T}$. The procedure is almost identical to that outlined in CHESS [17], but with better selection of poles for partitioning (see Algorithm 2.1). After building $\mathcal{T}$ with CLAM, we also perform a depth-first reordering of the dataset (see Section 2.1.2).

Algorithm 2.1 Partition(C, *criteria*)

Require: $f : X \times X \mapsto \mathbb{R}^+ \cup \{0\}$, a distance function
Require: C, a cluster
Require: *criteria*, user-specified continuation criteria
    $seeds \Leftarrow$ random sample of $\left\lceil \sqrt{|C|} \right\rceil$ points from C
    $c \Leftarrow$ geometric median of *seeds*
    $l \Leftarrow \arg\max f(c,x) \ \forall \ x \in C$
    $r \Leftarrow \arg\max f(l,x) \ \forall \ x \in C$
    $L \Leftarrow \{x \mid x \in C \wedge f(l,x) \leq f(r,x)\}$
    $R \Leftarrow \{x \mid x \in C \wedge f(r,x) < f(l,x)\}$
    if $|L| > 1$ **and** L satisfies *criteria* **then**
        Partition(L, *criteria*)
    end if
    if $|R| > 1$ **and** R satisfies *criteria* **then**
        Partition(R, *criteria*)
    end if

Given a cluster C with $|C|$ points, we define its two children by the following process. We take a random subsample S of $\sqrt{|C|}$ of C's points, and compute pairwise distances between all points in S. Using these distances, we compute the *geometric median* of S; in other words, we find the point that minimizes the sum of distances to all other points in S. We define the *center* of C to be this geometric median.

188   The *radius* of C is the maximum distance from the center to any other point in C. The
189 point that is responsible for that radius (i.e., the furthest point from the center) is designated
190 the *left pole* and the point that is furthest from the left pole is designated the *right pole*. We
191 then partition the cluster into a *left child* and a *right child*, where the left child contains all
192 points in the cluster that are closer to the left pole than to the right pole, and the right child
193 contains all points in the cluster that are closer to the right pole than to the left pole. Without
194 loss of generality, we assign to the left child those points that are equidistant from the two
195 poles. Starting from a root cluster $\mathcal{R}$ containing the entire dataset, we repeat this procedure
196 until each leaf contains only one datum, or we meet some other user-specified stopping criteria
197 (e.g., minimum cluster radius, minimum cluster cardinality, maximum tree depth, etc). This
198 process is described in Algorithm 2.1. During the partitioning process, we also compute (and
199 cache) the LFD of each cluster using Equation 2.2.

200   **2.1.2. Depth-First Reordering.** In CHESS [17], each cluster stored a list of indices into
201 the dataset. This list was used to retrieve the clusters' points during search. Although this
202 approach allowed us to retrieve the points in constant time, its memory cost was prohibitively
203 high. With a dataset of cardinality n and each cluster storing a list of indices for its points,
204 we stored a total of n indices at each depth in the tree $\mathcal{T}$. Assuming $\mathcal{T}$ is balanced, and thus
205 $\mathcal{O}(\log n)$ depth, this approach had a memory overhead of $\mathcal{O}(n \log n)$. In this work, we introduce
206 a new approach wherein, after building $\mathcal{T}$, we reorder the dataset so that points are stored
207 in a depth-first order. Then, within each cluster, we need only store its *cardinality* and an
208 *offset* to access its points from the dataset. The root cluster $\mathcal{R}$ has an offset of zero and a
209 cardinality equal to the number of points in the dataset. A left child has the same offset its
210 parent, and the corresponding right child has an offset equal to the left child's offset plus the
211 left child's cardinality. With no additional memory cost nor time cost for retrieving points
212 during search, depth-first reordering offers the same time complexity as CHESS but with $\mathcal{O}(n)$
213 memory overhead.

214   **2.1.3. Time Complexity.** The asymptotic complexity of Partition is the same as described
215 in [17]. By using an approximate partitioning with a $\sqrt{n}$ sample, we achieve $\mathcal{O}(n)$ cost of
216 partitioning and $\mathcal{O}(n \log n)$ cost of building $\mathcal{T}$. This is a significant improvement over exact
217 partitioning and tree-building, which cost $\mathcal{O}(n^2)$ and $\mathcal{O}(n^2 \log n)$ respectively.
218   This complexity analysis ignores the cost of checking the continuation criteria. While a user
219 could specify criteria of any complexity (our implementation allows for this), for this paper we
220 continue clustering until each cluster is a singleton, i.e., it contains only one point or duplicates
221 of the same point and has a radius of zero. This criterion costs $\mathcal{O}(1)$ to check per cluster, and
222 so does it not affect the overall complexity of the algorithm as used in this paper.
223   We also note here that this $\mathcal{O}(n \log n)$ complexity assumes that $\mathcal{T}$ is balanced. In practice,
224 for real datasets, we expect Algorithm 2.1 to produce anything but a balanced tree; the varying
225 sampling density in different regions of the manifold and the low dimensional "shape" of the
226 manifold itself will cause it to be unbalanced. The only case in which we would expect a
227 balanced tree is if the dataset were uniformly distributed, e.g., in a d-dimensional hyper-cube.
228   When building a tree for a real dataset, the depth would be larger than $\log n$, which would
229 seem to increase the cost of tree-building. However, because of the large numbers of leaf clusters
230 at shallow depths, each subsequent level of $\mathcal{T}$ would also have fewer points, and so the cost

231 of building each subsequent level would be lower. Any analysis of unbalanced trees would be
232 highly dependent on the specific dataset, and so we do not provide one here. We do, however,
233 explore empirical results based on forcing a balanced tree in Section 4.4. In any case, the tree
234 is built only once, so this cost is amortized over all search queries and other applications.
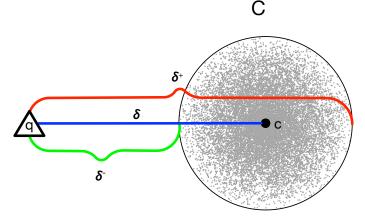

Figure 1: δ, δ^+, and δ^- for a cluster C and a query q. $\delta = f(q, c)$ is the distance from the query to the cluster center c. $\delta^+ = \delta + r$ is the distance from the query to the theoretically farthest point in C. $\delta^- = \max(0, \delta - r)$ is the distance from the query to the theoretically closest point in C.

Algorithm 2.2 tree-search(C, q, r)

Require: f, a distance function
Require: C, a cluster
Require: q, a query
Require: r, a search radius
235   **if** $\delta_C^+ \leq r$ **then**
        return $\{C\}$
    else
        $[L, R] \Leftarrow children$ of C
        return tree-search(L, q, r)
                $\cup$ tree-search(R, q, r)
    end if

Algorithm 2.3 leaf-search(Q, q, r)

Require: f, a distance function
Require: Q, a set of clusters
Require: q, a query
Require: r, a search radius
    $H \Leftarrow \emptyset$
    for $C \in Q$ **do**
        if $\delta_C^+ \leq r$ **then**
            $H \Leftarrow H \cup C$
        else
            for $p \in C$ **do**
                if $f(p, q) \leq r$ **then**
                    $H \Leftarrow H \cup \{p\}$
                end if
            end for
        end if
    end for
    return H

---

236   **2.2. k-Nearest Neighbors Search.** Given a dataset $\mathbf{X}$, a distance function f and the root
237 cluster $\mathcal{R}$ of the tree $\mathcal{T}$ (constructed by Algorithm 2.1), we can now formally pose the k-NN
238 search problem. Given a user specified integer k and query q, k-NN search aims to find the k
239 closest points to q in $\mathbf{X}$. In other words, k-NN search aims to find the set H such that $|H| = k$
240 and $H = B_X(q, \rho_k)$ where $\rho_k = \max\{f(q, p) \,\forall\, p \in H\}$ is the distance from q to the kth nearest

241 neighbor in **X**.
242   In this section, we present three novel algorithms for exact k-NN search: Repeated ρ-NN,
243 Breadth-First Sieve, and Depth-First Sieve. In these algorithms, we use H, for *hits*, to refer to
244 the data structure that stores the closest points to the query found so far, and Q to refer to
245 the data structure that stores the clusters and points that are still in contention for being one
246 of the k nearest neighbors. These algorithms also use some terminology defined and illustrated
247 in Figure 1.

Algorithm 2.4 Repeated ρ-NN($\mathcal{R}$, q, k)

Require: $\mathcal{R}$, the root cluster
Require: q, a query
Require: k, the number of neighbors to find
  $H \Leftarrow [\ ]$, a max-heap by δ of size k
  $r \Leftarrow radius$ of $\mathcal{R}$
  $r \Leftarrow \frac{r}{|\mathcal{R}|}$
  $Q \Leftarrow$ tree-search($\mathcal{R}$, q, r)
  while $\sum_{C \in Q} |C| < k$ **do**
    if $Q = \emptyset$ **then**
      $r \Leftarrow 2 \cdot r$
    else if $\sum_{C \in Q} |C| >= k$ **then**
      $\mu \Leftarrow \frac{1}{|Q|} \cdot \sum_{C \in Q} \left(LFD(C)^{-1} \right)$
      $r \Leftarrow r \cdot \min \left(2, \left(\frac{k}{\sum_{C \in Q} |C|} \right)^{\mu} \right)$
    end if
    $Q \Leftarrow$ tree-search($\mathcal{R}$, q, r)
  end while
  $H \Leftarrow$ leaf-search(Q, q, r)
  return H as a list

---

248   **2.2.1. Repeated ρ-NN.** This algorithm relies on the *tree-search* (Algorithm 2.2) and
249 *leaf-search* (Algorithm 2.3) as described in [17] and reproduced here for completeness.
250   For Repeated ρ-NN search (Algorithm 2.4), we begin by performing *tree-search* with a
251 search radius r equal to the radius of the root $\mathcal{R}$ divided by the cardinality of the dataset. If
252 no clusters are found, then we double r and perform tree-search again, repeating until we find
253 at least one cluster.
254   Now, so long as $\sum_{C \in Q} |C| < k$, we continue to perform tree-search, but instead of doubling
255 r on each iteration, we multiply it by a factor determined by the LFD in the vicinity of the
256 query ball. In particular, we increase the radius by a factor of

257 (2.3)
$$\min \left(2, \left(\frac{k}{\sum_{C \in Q} |C|} \right)^{\mu} \right)$$

258 where μ is the multiplicative inverse of the harmonic mean of the LFD of the clusters in Q,
259 i.e., $\mu = \frac{1}{|Q|} \cdot \sum_{C \in Q} \left(LFD(C)^{-1} \right)$. We use the harmonic mean to ensure that μ is not dominated

by outlier clusters with very high LFD. We cap the radial increase at 2 to ensure that we do not increase the radius too quickly in any single iteration.

Intuitively, the factor by which we increase the radius should be *inversely* related to the number of points found so far. When the LFD at the radius scale from the previous iteration is high, this suggests that the data are densely populated in that region. Thus, a small increase in the radius would likely encounter many more points, so a smaller radial increase would suffice to find k neighbors. Conversely, when the LFD at the radius scale from the previous iteration is low, this suggests that the data are sparsely populated in that region. In such a region, a small increase in the radius would likely encounter vacant space, so a larger radial increase is needed. Thus, the factor of radius increase should also be *inversely* related to the LFD. However, we should not increase the radius too drastically with any one iteration because we assume that the LFD is only *locally* uniform. A large increase in the radius would likely break out of the local region and potentially encounter too many new clusters, which would make the subsequent step computationally expensive.

Once $\sum_{C \in Q} |C| \geq k$, we are guaranteed to have found at least k neighbors, and so we perform *leaf-search* to find the k nearest neighbors among the points in the clusters found after the last tree-search.

2.2.2. Complexity of Repeated ρ-NN.

Theorem 2.1. *Let X be a dataset and q a query sampled from the same distribution (i.e., arising from the same generative process) as X. Then time complexity of performing Repeated ρ-NN search on X with query q is*

$$(2.4) \qquad \mathcal{O}\left(\underbrace{\log \overbrace{\mathcal{N}_{\hat{r}}(X)}^{\text{metric entropy}}}_{\text{tree-search}} + \underbrace{\overbrace{k}^{\text{output size}} \cdot \overbrace{\left(1 + 2 \cdot \left(\frac{|\hat{C}|}{k} \right)^{d-1} \right)^{d}}^{\text{scaling factor}}}_{\text{leaf-search}} \right)$$

where $\mathcal{N}_{\hat{r}}(X)$ is the metric entropy of the dataset, d is the LFD of the dataset, and k is the number of nearest neighbors.

Proof. We consider the *tree-search* and *leaf-search* stages of search separately. Tree-search refers to the process of identifying clusters that overlap with the query ball, or in other words, clusters that might contain one of the k nearest neighbors by doing repeated iterations of the ρ-NN search Algorithm 2.2. In [17], we showed that the complexity of ρ-NN search is

$$(2.5) \qquad \mathcal{O}\left(\underbrace{\log \overbrace{\mathcal{N}_{\hat{r}}(X)}^{\text{metric entropy}}}_{\text{tree-search}} + \underbrace{\overbrace{|B(q, \rho)|}^{\text{output size}} \overbrace{\left(\frac{\rho + 2 \cdot \hat{r}}{\rho} \right)^{d}}^{\text{scaling factor}}}_{\text{leaf-search}} \right)$$

where $\hat{r}$ is the *mean* radius of leaf clusters, $\mathcal{N}_{\hat{r}}(X)$ is the metric entropy at that radius, $B(q, \rho)$ is a ball of radius ρ around the query q, and d is the LFD around the query at the length scale of ρ and $\rho + 2 \cdot \hat{r}$.

To extend Equation 2.5 to Repeated ρ-NN, we must first estimate the number of iterations of tree-search (Algorithm 2.2) needed to find a radius that guarantees at least k neighbors. Since q is sampled from the same distribution as X, the LFD near q should not differ significantly from the (harmonic) mean of the LFDs of clusters near q at the scale of the distance from the query to the kth nearest neighbor. Given that LFD near q does not differ significantly from that of nearby clusters, Equation 2.3 suggests that in the expected case, we need only two iterations of tree-search to find k neighbors: one iteration to find at least one cluster, and one more to find enough clusters to guarantee k neighbors. Since this is a constant factor, complexity of tree-search for Repeated ρ-NN is the same as that of ρ-NN search, i.e., $\mathcal{O}\big(\log \mathcal{N}_{\hat{r}}(X)\big)$.

We proceed to determine the complexity of leaf-search. Let Q be the set of clusters returned by tree-search. We must estimate $\sum_{C \in Q} |C|$, the total cardinality of the clusters returned by tree-search; since we must examine every point in each such cluster, time complexity of leaf-search is linear in this quantity. Let ρ_k be the distance from the query to the kth nearest neighbor. Then, we see that Q is expected to be the set of clusters that overlap with a ball of radius ρ_k around the query. We can estimate this region as a ball of radius $\rho_k + 2\hat{r}$, where $\hat{r}$ is the mean radius of the clusters in Q.

The work in [37] showed that $\sum_{C \in S} |C| \leq \gamma \, |B(q, \rho_k)| \left(\frac{\rho_k + 2 \cdot \hat{r}}{\rho_k}\right)^d$, where γ is a constant. By definition of ρ_k, we have that $|B(q, \rho_k)| = k$. Thus, $\sum_{C \in S} |C| \leq \gamma k \left(1 + 2 \cdot \frac{\hat{r}}{\rho_k}\right)^d$. It remains to determine an estimate for ρ_k. We let $\hat{d}$ be the (harmonic) mean LFD of the clusters in Q. While ordinarily we compute LFD by comparing cardinalities of two balls with two different radii centered at *the same* point, in order to estimate ρ_k, we instead compare the cardinality of a ball *around the query* of radius ρ_k to the mean cardinality, $|\hat{C}|$, of clusters in Q at a radius equal to the mean of their radii, $\hat{r}$. Since q is from the same distribution as X, the LFD at q should not be significantly different from that at the center of a cluster in Q. By Equation 2.1, $\hat{d} = \frac{\log \frac{|\hat{C}|}{k}}{\log \frac{\hat{r}}{\rho_k}}$. We can rearrange this equation to get $\frac{\hat{r}}{\rho_k} = \left(\frac{|\hat{C}|}{k}\right)^{\hat{d}^{-1}}$. Using this to simplify the term for leaf-search in Equation 2.5, we get:

$$k \left(1 + 2 \cdot \left(\frac{|\hat{C}|}{k}\right)^{\hat{d}^{-1}}\right)^d$$

Again using the assumption that since q is from the same distribution as X, the LFD at q should not be significantly different from that at a cluster in Q, we have that $\hat{d} \approx d$. By combining the bounds for tree-search and leaf-search, we see that Repeated ρ-NN has time complexity as claimed in Equation 2.4. ∎

We remark that the scaling factor in Equation 2.4 should be close to 1 unless LFD is highly variable in the region around the query (i.e., if $\hat{d}$ differs significantly from d).

2.2.3. Breadth-First Sieve. This algorithm (described in Algorithm 2.5) performs a breadth-first traversal of the tree $\mathcal{T}$, pruning clusters by using a modified version of the QuickSelect algorithm [14] at each level.

We begin by letting Q be a set of 3-tuples (p, δ_p^+, m), where p is either a cluster or a point,

δ_p^+ is the δ^+ of p as illustrated in Figure 1, and m is the multiplicity of p in Q. During the breadth-first traversal, for every cluster C we encounter, we add $(C, \delta_C^+, |C| - 1)$ and $(c, \delta_c, 1)$ to Q, where c is the center of C. Recall that by the definitions of δ and δ^+ given in Section 2.2, since c is a point, $\delta_C = \delta_c = \delta_c^+ = \delta_c^-$.

Algorithm 2.5 Breadth-First Sieve($\mathcal{R}, q, k$)

Require: $\mathcal{R}$, the root cluster
Require: q, a query
Require: k, the number of neighbors to find
  $c \Leftarrow center$ of $\mathcal{R}$
  $Q \Leftarrow \{ (\mathcal{R}, \delta_\mathcal{R}^+, |\mathcal{R}| - 1), (c, \delta_\mathcal{R}, 1) \}$
  while $\sum_{(\_,\_,m) \in Q} m \neq k$ **do**
    $\tau \Leftarrow$ QuickSelect(Q, k)
    $Q' \Leftarrow \emptyset$
    for $(C, \_, \_) \in Q$ **do**
      if $\delta_C^- \leq \tau$ **then**
        if C is a point **then**
          $Q' \Leftarrow Q' \cup \{(C, \delta_C, 1)\}$
        else if C is a leaf **then**
          $Q' \Leftarrow Q' \cup \{(p, \delta_p, 1)$ for $p \in C\}$
        else
          $[L, R] \Leftarrow$ children of C
          $l, r \Leftarrow$ centers of L, R
          $Q' \Leftarrow Q' \cup \{(L, \delta_L^+, |L| - 1), (l, \delta_L, 1)\}$
          $Q' \Leftarrow Q' \cup \{(R, \delta_R^+, |R| - 1), (l, \delta_R, 1)\}$
        end if
      end if
    end for
    $Q \Leftarrow Q'$
  end while
  QuickSelect(Q, k)
  return The first k points in Q

Algorithm 2.6 Depth-First Sieve($\mathcal{R}, q, k$)

Require: $\mathcal{R}$, the root cluster
Require: q, a query
Require: k, the number of neighbors to find
  $Q \Leftarrow [\mathcal{R}]$, a min-heap by δ^-
  $H \Leftarrow [\,]$, a max-heap by δ of size k
  while $|H| < k$ **or** $H.peek.\delta \geq Q.peek.\delta^-$ **do**
    while $Q.peek$ is not a leaf **do**
      $C \Leftarrow Q.pop$, the closest cluster
      $[L, R] \Leftarrow$ children of C
      $Q.push(L)$
      $Q.push(R)$
    end while
    $leaf \Leftarrow Q.pop$
    for $p \in leaf$ **do**
      $H.push(p)$
    end for
  end while
  return H

We then use the QuickSelect algorithm, modified to account for multiplicities and to reorder Q in-place, to find the element in Q with the kth smallest δ^+; in other words, we find τ, the smallest δ^+ in Q such that $|B(q, \tau)| \geq k$. Since this step may require a binary search for the correct pivot element to find τ and reordering with a new pivot takes linear time in the size of the input list, this version of QuickSelect has $\mathcal{O}(|Q| \log |Q|)$ time complexity.

Next, we go over all items in Q, skipping over any for which $\delta^- > \tau$ because such elements cannot contain (or be) one of the k nearest neighbors. If the item corresponds to a point, we keep it. If the item corresponds to a leaf cluster, we add all its points to Q with a multiplicity of 1 each. If the item corresponds to a non-leaf cluster, we to Q the pairs of 3-tuples corresponding to its child clusters. We continue this process until the sum of multiplicities in Q is exactly k. We then use the QuickSelect algorithm one last time to reorder Q and return the k nearest neighbors.

2.2.4. Depth-First Sieve. This algorithm (described in Algorithm 2.6) is quite similar to a depth-first traversal of the tree $\mathcal{T}$, using two priority queues to track clusters and hits, and to prioritize which branch of $\mathcal{T}$ to explore next.

Let Q be a min-queue of clusters prioritized by δ^- and H be a max-queue (with capacity k and the ability to maintain its maximum size) of points prioritized by δ. Q starts containing only the root cluster $\mathcal{R}$ while H starts empty. So long as H is not full or the top element in H has δ greater than or equal to the top element in Q, we take the following steps:

- While the top-priority element is not a leaf, remove it from Q and add its children to Q.
- Remove the top-priority element (a leaf) from Q and add all its points to H.

This process terminates when H is full and the top element in H has δ less than the top element in Q, i.e., the theoretically closest point left to be considered in Q is farther from the query than the kth nearest neighbor in H. This leaves H containing exactly the k nearest neighbors to the query.

Note that this algorithm is not truly a depth-first traversal of $\mathcal{T}$ in the classical sense, because we use Q to prioritize which branch of $\mathcal{T}$ we descend into. Indeed, we expect this algorithm to often switch which branch of $\mathcal{T}$ is being explored at greater depth.

2.2.5. Complexity of Sieve Methods. Due to their similarity, we combine the complexity analyses of both Sieve methods. For these methods we again use the terminology of tree-search and leaf-search. Tree-search navigates the cluster tree and adds clusters to Q. Leaf-search exhaustively searches *some* of the clusters in Q to find the k nearest neighbors.

Theorem 2.2. Let X be a dataset and q a query sampled from the same distribution (i.e., arising from the same generative process) as X. Let $T := \mathcal{O}\left(\lceil d \rceil \cdot \log \mathcal{N}_{\hat{r}}(X)\right)$ and $L :=$
$$\mathcal{O}\left(k \cdot \left(1 + 2 \cdot \left(\frac{|\hat{C}|}{k}\right)^{d-1}\right)^d\right), \text{ where } \mathcal{N}_{\hat{r}}(X) \text{ is the metric entropy of } X, \, d \text{ is the LFD of } X, \, |\hat{C}|$$
is the mean cardinality of clusters overlapping the query ball, and k is the number of nearest neighbors. Then, for dataset X and query q, the time complexity of performing Breadth-First Sieve search is

$$(2.6) \qquad\qquad \mathcal{O}\Big((T + L)\log(T + L)\Big)$$

and the the time complexity of performing Depth-First Sieve search is

$$(2.7) \qquad\qquad \mathcal{O}\Big(T \log T + L \log k\Big).$$

Proof. Since q is sampled from the same distribution as X, the LFD near q should not differ significantly from the LFDs of clusters near q at the scale of the distance from the query to the kth nearest neighbor.

Consider leaf clusters with cardinalities near k. Let d be the LFD in this region, and Q denote the set of candidate points and clusters. Then the number of leaf-clusters in Q is bounded above by $2d$, where the bound is achieved if we have a cluster overlapping the query ball at each end of each of $\lceil d \rceil$ mutually-orthogonal axes. In the worst-case scenario for tree-search, these leaf clusters would all come from different branches of the tree, and so tree-search looks

at $2 \cdot \lceil d \rceil \cdot \log \mathcal{N}_{\hat{r}}(X)$ clusters. Thus, the asymptotic complexity is $T := \mathcal{O}(\lceil d \rceil \cdot \log \mathcal{N}_{\hat{r}}(X))$. For leaf-search, the output size and scaling factor are the same as in Repeated ρ-NN, and so

the asymptotic complexity is $L := \mathcal{O}\left(k \cdot \left(1 + 2 \cdot \left(\frac{|\hat{C}|}{k} \right)^{d-1} \right)^d \right)$.

The asymptotic complexity of Breadth-First Sieve is dominated by the QuickSelect algorithm to calculate τ. Since this method is log-linear in the length of Q, and Q contains the clusters from tree-search and the points from leaf-search, we see that the asymptotic complexity is as claimed in Equation 2.6.

For Depth-First Sieve, since we use two priority queues, the asymptotic complexity is as claimed in Equation 2.7.   ■

2.3. Auto-Tuning. We perform some simple auto-tuning to select the optimal k-NN algorithm to use with a given dataset. We start by taking the center of every cluster at a low depth (e.g., 10) in $\mathcal{T}$ as a query. This gives us a small, representative sample of the dataset. Using these clusters' centers as queries, and a user-specified value of k, we record the time taken for k-NN search on the sample using each of the three algorithms described in Section 2.2. We select the fastest algorithm over all the queries as the optimal algorithm for that dataset and value of k. Note that even though we select the optimal algorithm based on use with some user-specified value of k, we still allow search with any value of k.

2.4. Synthetic Data. Based on our asymptotic complexity analyses, we expect CAKES to perform well on datasets with low LFD, and for its performance to scale sub-linearly with the cardinality of the dataset. To test this hypothesis, we use some datasets from the ANN-benchmarks suite [2] and synthetically augment them to generate similar datasets with exponentially larger cardinalities. We do the same with a large random dataset of uniformly distributed points in a hypercube. We then compare the performance of CAKES to that of other algorithms on the original datasets and the synthetically augmented datasets.

To elaborate on the augmentation process, we start with an original dataset from the ANN-benchmarks suite. Let X be the dataset, d be its dimensionality, ϵ be a user-specified noise level, and m be a user-specified integer multiplier. For each datum $\mathbf{x} \in X$, we create $m-1$ new data points within a distance ϵ of $\mathbf{x}$. We construct a random vector $\mathbf{r}$ of d dimensions in the hyper-sphere of radius ϵ centered at the origin. We then add $\mathbf{r}$ to $\mathbf{x}$ to get a new point $\mathbf{x}'$. Since $||\mathbf{r}|| \leq \epsilon$, we have that $||\mathbf{x} - \mathbf{x}'|| \leq \epsilon$ (i.e., $\mathbf{x}'$ is within a distance ϵ of $\mathbf{x}$). This produces a new dataset X' with $|X'| = m \cdot |X|$. This augmentation process does not add to the overall topological structure of the dataset, but it does increase its cardinality by a factor of m. This allows us to isolate the effect of cardinality on search performance from that of other factors such as dimensionality, choice of metric, or the topological structure of the dataset.

3. Datasets And Benchmarks.

3.1. ANN-Benchmark Datasets. We benchmark on a variety of datasets from the ANN-benchmarks suite [2]. Table 1 summarizes these datasets. All benchmarks were conducted on an Intel Xeon E5-2690 v4 CPU @ 2.60GHz with 512GB RAM. The OS kernel was Manjaro Linux 5.15.164-1-MANJARO. The Rust compiler was Rust 1.83.0, and the Python interpreter version was 3.9.18.

Table 1: Datasets used in benchmarks.

DATASET	DISTANCE FUNCTION	CARDINALITY	DIMENSIONALITY
FASHION-MNIST	EUCLIDEAN	60,000	784
GLOVE-25	COSINE	1,183,514	25
SIFT	EUCLIDEAN	1,000,000	128
RANDOM	EUCLIDEAN	1,000,000	128
SILVA	LEVENSHTEIN	2,224,640	3,712
RADIOML	DYNAMIC TIME WARPING	97,920	1,024

3.2. Random Datasets and Synthetic Augmentation. In addition to benchmarks on datasets from the ANN-Benchmarks suite, we also benchmarked on synthetic augmentations of these real datasets, using the process described in 2.4. In particular, we use a noise tolerance $\epsilon = 0.01$ and explore the scaling behavior as the cardinality multiplier (referred to as "Mult." in Tables 2, 3, 4 and 5) increases. We also benchmarked on purely randomly-generated datasets of various cardinalities. For this, we used a base cardinality of 1,000,000 and a dimensionality of 128 to match the Sift dataset; hereafter, we refer to the random dataset with cardinality 1,000,000 and dimensionality 128 as "Random." This benchmark allows us to isolate the effect of a manifold structure (which we expect to be absent in a purely random dataset) on the performance of the CAKES' algorithms.

In order to calculate recall on the augmented datasets, we perform linear search in Rust and save the results to disk. We verified that linear search on the original dataset produced neighbors with perfect recall compared to those provided by the ANN-Benchmarks suite. We then use the results of linear search on the augmented datasets as ground truth for calculating recall of CAKES's algorithms in Rust and read them in Python for calculating recall of HNSW, ANNOY, and FAISS-IVF.

3.3. SILVA 18S. To demonstrate CAKES with a more exotic distance function, we also benchmarked on the SILVA 18S ribosomal RNA dataset [29]. This dataset contains ribosomal RNA sequences of 2,224,640 genomes, with the longest sequence having 3,712 letters. We held out a set of 1,000 random sequences from the dataset to use as queries for benchmarking. We use Levenshtein [23] distance on the unaligned sequences to build the tree and to perform k-NN search. We note that the sequences in this dataset were provided in a multiple sequence alignment with a width of 50,000 characters. We could have used Hamming distance on the aligned sequences, but we chose to use Levenshtein distance on the unaligned sequences to help demonstrate the flexibility of CAKES in handling exotic distance functions. Under Hamming distance, we could consider this dataset to have an embedding dimension of 50,000, but for Levenshtein distance, we state the dimensionality as the length of the longest sequence, i.e. 3,712.

3.4. Radio ML. As another example of an exotic distance function, we benchmarked CAKES on the Radio-ML dataset [28] under Dynamic Time Warping [25] as the distance function. This dataset contains samples of synthetically generated signal captures of different modulation modes over a range of signal-to-noise ratio (SNR) levels. Specifically, it comprises

24 modulation modes at 26 different SNRs ranging from -20 dB to 30 dB, with 4,096 samples at each combination of modulation mode and SNR level. Thus, it contains $24 \cdot 26 \cdot 4096 = 2,555,504$ samples in total. Each sample is a 1,024-dimensional complex-valued vector, representing a signal capture (a time-series of complex-valued numbers). We used a subset of this dataset, containing $97,304$ samples at 10dB SNR and used another $1,000$ samples at the same SNR as a hold-out set of queries.

3.5. Other Algorithms. We benchmarked the three CAKES algorithms against a naïve linear search implementation in Rust. We also benchmarked against three state-of-the-art similarity search algorithms: HNSW, ANNOY, and FAISS-IVF in Python. We verified that our implementation of linear search produces the same neighbors as provided by the ANN-Benchmarks suite for Fashion-MNIST, Glove-25 and Sift datasets. We then used this linear search implementation to find and store the ground-truth for the augmented versions of the datasets. We used this ground-truth to calculate recall for CAKES's algorithms in Rust and for HNSW, ANNOY, and FAISS-IVF in Python. We plot the results of these benchmarks in Figure 3.

4. Results. The algorithms presented in this paper rely heavily on the local fractal dimension (LFD) being much smaller than than the embedding dimension of the dataset. As such, we begin by examining our assumptions about the LFD of real-world datasets (see Section 4.1 and Figure 2).

We continue by comparing the performance of the CAKES algorithms against HNSW, ANNOY and FAISS-IVF on the Fashion-MNIST, Glove-25, Sift and Random datasets. We also compare how this performance scales to the augmented versions of these datasets. We also report the performance of CAKES on the Silva and RadioML datasets (see Section 4.3, Figure 3, and Tables 2, 3, 4 and 5).

Finally, we test our intuitions about unbalanced clustering being better for search than balanced clustering (see Section 4.4 and Figure 4).

4.1. Local Fractal Dimension of Datasets. Since the time complexity of CAKES algorithms scales with the LFD of the dataset, we examine the LFD of each dataset we used for benchmarks. Figure 2 illustrates the trends in LFD for Fashion-MNIST, Glove-25, Sift, Random, Silva 18S, and Radio-ML. In this section, when we discuss trends in LFD, unless otherwise noted, we are referring to the 95th percentile of LFD.

The Fashion-MNIST dataset has an embedding dimension of 784 and uses the Euclidean distance metric. In Figure 2a we observe that until approximately depth 5, Fashion-MNIST's LFD is low (i.e., less than 4). It then starts increasing, reaching a peak of about 6 near depth 20, before decreasing to 1 at the maximum depth.

The Glove-25 dataset has an embedding dimension of 25 and uses the cosine distance function which, notably, is not a metric. Relative to Fashion-MNIST, Glove-25 has low LFD, as shown in Figure 2b. All percentile lines for Glove-25 are flatter and lower, indicating that the LFD is lower across the entire dataset, and that the LFD does not vary as much by depth. In particular, Glove-25's LFD is less than 3 for all depths.

The Sift dataset has an embedding dimension of 128 and uses the Euclidean distance metric. Figure 2c shows the LFD by depth for Sift, which has higher LFD relative to Fashion-MNIST

and Glove-25. It increases sharply to a peak of 9 around a depth of 10. It then decreases smoothly until reaching the deepest leaves in the tree.

We generated the Random dataset to have the same cardinality and dimensionality as Sift. We used a uniform distribution in a 128-dimensional unit-hypercube to generate the points and the Euclidean metric to measure distances among them. Figure 2d shows that the character of this dataset is significantly different from the others. The LFD starts at 20 at depth 0 and all percentile lines decrease linearly with depth until reaching the leaves of the tree. The spread in LFD starts very small for the first few clusters and increases as depth increases. The LFD of approximately 20 for the root cluster $\mathcal{R}$ is what we expect for this random dataset. To elaborate, the distribution of points in such a dataset should reflect the curse of dimensionality, i.e., the fact that in high dimensional spaces, the minimum and maximum pairwise distances between any two points are approximately equal. As a result, $\mathcal{R}$'s radius r, which reflects the maximum distance between the center c and any other point, should not differ significantly from the distance between the center and its closest point. A consequence of this is that, with high probability, for every point in $\mathcal{R}$, its distance from c is greater than $\frac{r}{2}$; in other words, $B(c, \frac{r}{2})$ contains only c while $B(c, r)$ contains the entire dataset. Given our definition of LFD in Equation 2.2, this means that the LFD of $\mathcal{R}$ is approximately $\log_2(\frac{|X|}{1}) = \log_2(1,000,000) \approx 20$, which is what we observe in Figure 2d. Theoretically, the LFD of this dataset should be 128, i.e. it should be the same as the embedding dimension. This reflects the difference between how we empirically measure the LFD and the value we would expect. With sample sizes larger than 1,000,000, we would expect the LFD to approach 128 until we have sampled 2^{128} points, at which point the LFD would be 128 and would stay at 128 for even larger sample sizes. Unfortunately, such a large sample is practically impossible to generate.

The Silva-18S dataset consists of genomic sequences whose unaligned lengths are at-most 3,712. As such the embedding dimension is 3,712, though as discussed previously, the embedding dimension would be 50,000 in a multiple sequence alignment. We use the Levenshtein edit distance (a metric) to measure distances between sequences. This dataset, as shown in Figure 2e, exhibits consistently low LFD. In particular, LFD is less than 3 for all depths, hovering near 1 for clusters at depth 40 and deeper.

The Radio-ML dataset consists of measurements of radio-frequency signals using 1,024 dimensional complex-valued vectors. We use the Dynamic Time Warping distance metric on this dataset. This dataset is synthetic [28] but uses a far more elaborate generation process than our Random dataset. The LFD values show three distinct peaks around an LFD of 12 at or near depths of 8, 25 and 50. Each peak is followed by a linear decrease until encountering a sharp spike for the next peak. Within each of the tree portions, this dataset has a character very similar to that of the Random dataset. This suggests that the dataset obeys the manifold hypothesis at some scales, but that it is not "scale free," as the LFD varies significantly by depth. This is likely the result of a piecewise uniform sampling strategy used to generate the different modulation modes present in the dataset.

4.2. Indexing and Tuning Time. For each of the ANN-benchmark datasets and the Random dataset, we report the time taken for each algorithm to build the index and to tune the hyper-parameters for these indices to achieve the highest possible recall. For the sake of brevity, these results are reported in the supplement.

(a) Fashion-MNIST

(b) Glove-25

(c) Sift

(d) A random dataset

(e) Silva 18S

(f) RadioML

minimum — 5th percentile — 25th percentile
median — 95th percentile — 75th percentile
maximum
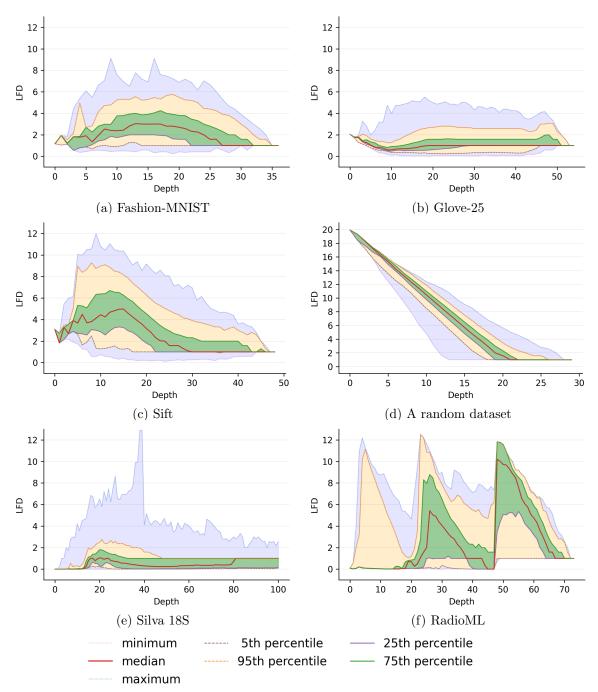
Figure 2: Local fractal dimension vs. cluster depth across six datasets. The 'random' dataset is randomly generated according to the procedure in Section 3.2; note that the y-axis is different for this dataset. In each plot, the horizontal axis denotes depth in the cluster tree, and the vertical axis denotes the LFD of clusters at that depth. We show lines for the 5th, 25th, 50th, 75th and 95th percentiles of LFD, as well as the minimum and maximum LFD at each depth. So that plots best reflect the distribution of LFDs across the entire *dataset*, we count each cluster as many times as its cardinality. For example, if, for some dataset, the 95th percentile of LFD at depth 40 is 3, this means that 95% of the points in clusters at depth 40 belong to a cluster whose LFD is at most 3.

4.3. Scaling Behavior and Recall. We benchmark the CAKES algorithms, naïve linear search, HNSW, ANNOY, and FAISS-IVF on the Fashion-MNIST, Glove-25, Sift, and Random datasets. We augment these datasets with synthetic points to examine how performance scales with cardinality. We also benchmark the CAKES algorithms on the Silva and RadioML datasets, and we subsample these datasets instead of augmenting them to examine how performance scales with cardinality. For each dataset, we use the corresponding distance function (see Table 1). For HNSW, ANNOY, and FAISS-IVF, we allow for a hyper-parameter search to tune their index for maximum recall. For CAKES, we build the tree and use our auto-tuning approach (see Section 2.3) to select the fastest algorithm for each dataset and cardinality. We show results for $k = 10$ in the main text, but results for $k = 100$ can be found in the Supplement. Figure 3 shows the scaling behavior of these algorithms on these datasets while Tables 2, 3, 4 and 5 report the throughput and recall of CAKES's algorithms at each augmented cardinality for Fashion-MNIST, Glove-25, Sift, and Random respectively.

On the Fashion-MNIST, Glove-25, and Sift datasets (in Figures 3a, 3b, and 3c respectively), we observe that as cardinality increases, the Depth-First Sieve algorithm is consistently the fastest CAKES algorithm with a throughput that is constant in the cardinality of the dataset. The Breadth-First Sieve algorithm is the usually the second fastest, also with a nearly constant throughput across all cardinalities. The Repeated ρ-NN algorithm, however, falls off in throughput as cardinality increases. All three CAKES algorithms exhibit perfect recall on the Fashion-MNIST and Sift datasets, and near-perfect recall on the Glove-25 dataset (which uses cosine distance). In contrast, HNSW and ANNOY are faster than CAKES's algorithms for all cardinalities and have near constant throughput as cardinality increases, but their recall degrades quickly as cardinality increases. FAISS-IVF exhibits linearly decreasing throughput as cardinality increases, which is expected from the algorithm given that we tune the hyper-parameters to maximize recall.

On the Random dataset, HNSW and ANNOY are still the fastest algorithms but exhibit recall values near 0. CAKES's algorithms show linearly decreasing throughput as cardinality increases and are also slower than naïve linear search.

Tables 2, 3, 4 and 5 show the throughput and recall of CAKES's algorithms at each augmented cardinality for Fashion-MNIST, Glove-25, Sift, and Random respectively. Though the plots in Figure 3 present results for each of CAKES's three algorithms separately, the results in the CAKES column in these tables represent the fastest CAKES algorithm at that dataset and cardinality only. We used our auto-tuning approach (see Section 2.3) for each new tree (i.e., at each cardinality), and this approach was always able to select the fastest algorithm for each dataset at each cardinality. Since we also allow for tuning hyper-parameters for the other algorithms, and we allow for different sets of hyper-parameters at each cardinality, it is a fair comparison for these tables to only list the performance of the fastest CAKES algorithm. When reporting recall, we use 1.000∗ to denote that the recall is imperfect, but rounds to 1.000.

For Silva and RadioML, we benchmarked only CAKES's algorithms because HNSW, ANNOY and FAISS support neither the required distance functions nor, in the case of RadioML, complex-valued data. Due to the massive sizes of these datasets and challenges in generating plausible augmentations, we took random sub-samples ranging up to the entirety of the dataset—rather than augmented versions of the dataset—to examine how performance scales with cardinality. For the Silva dataset (see Figure 3e), we observe that throughput initially

Table 2: Throughput (QPS) and Recall on the Fashion-MNIST dataset. A recall value of 1.000∗ denotes imperfect recall that rounds to 1.000. While recall for CAKES's does *not* degrade with cardinality, recall for HNSW and ANNOY degrades with cardinality. In contrast, CAKES has perfect recall on Fashion-MNIST, since it used with Euclidean distance, which is a metric.

Mult.	HNSW		ANNOY		FAISS-IVF		CAKES	
	QPS	Recall	QPS	Recall	QPS	Recall	QPS	Recall
1	1.33×10^4	0.954	2.19×10^3	0.950	2.01×10^3	1.000∗	3.46×10^3	1.000
2	1.38×10^4	0.803	2.12×10^3	0.927	9.39×10^2	1.000∗	3.68×10^3	1.000
4	1.66×10^4	0.681	2.04×10^3	0.898	4.61×10^2	0.997	3.44×10^3	1.000
8	1.68×10^4	0.525	1.93×10^3	0.857	2.26×10^2	0.995	3.30×10^3	1.000
16	1.87×10^4	0.494	1.84×10^3	0.862	1.17×10^2	0.991	3.34×10^3	1.000
32	1.56×10^4	0.542	1.85×10^3	0.775	5.91×10^1	0.985	2.96×10^3	1.000
64	1.50×10^4	0.378	1.78×10^3	0.677	2.61×10^1	0.968	3.25×10^3	1.000
128	1.49×10^4	0.357	1.66×10^3	0.538	1.33×10^1	0.964	2.96×10^3	1.000
256	–	–	1.60×10^3	0.592	6.65	0.962	2.79×10^3	1.000
512	–	–	1.83×10^3	0.581	3.56	0.949	2.84×10^3	1.000

Table 3: Throughput (QPS) and Recall on the Glove-25 dataset. A recall value of 1.000∗ denotes imperfect recall that rounds to 1.000. While recall for CAKES's does *not* degrade with cardinality, recall for HNSW and ANNOY degrades with cardinality. In contrast, CAKES has near-perfect recall on Glove-25, since it used with cosine distance, which is not a metric.

Mult.	HNSW		ANNOY		FAISS-IVF		CAKES	
	QPS	Recall	QPS	Recall	QPS	Recall	QPS	Recall
1	2.28×10^4	0.801	2.83×10^3	0.835	2.38×10^3	1.000∗	1.54×10^3	1.000∗
2	2.38×10^4	0.607	2.70×10^3	0.832	1.19×10^3	1.000∗	1.49×10^3	1.000∗
4	2.50×10^4	0.443	2.61×10^3	0.839	6.19×10^2	1.000∗	1.28×10^3	1.000∗
8	2.78×10^4	0.294	2.51×10^3	0.834	3.03×10^2	1.000∗	1.30×10^3	1.000∗
16	3.11×10^4	0.213	2.23×10^3	0.885	1.51×10^2	1.000∗	1.14×10^3	1.000∗
32	3.24×10^4	0.178	2.01×10^3	0.764	7.40×10^1	0.999	1.05×10^3	1.000∗
64	–	–	1.99×10^3	0.631	3.77×10^1	0.997	1.07×10^3	1.000∗
128	–	–	–	–	1.90×10^1	0.998	8.92×10^2	1.000∗
256	–	–	–	–	9.47	0.998	8.91×10^2	1.000∗

decreases linearly as cardinality increases, but levels off at higher cardinalities. Depth-First Sieve is, once again, the fastest CAKES algorithm. For Radio-ML (see Figure 3f), however, we see that throughput declines nearly linearly with cardinality, and that the three CAKES algorithms are, again, slower that naïve linear search.

4.4. Clustering Strategies and Number of Distance Computations. In addition to the scaling experiments, we also explore how four different clustering strategies affect the performance of search. These strategies are the Cartesian product of balanced vs. unbalanced clustering, and the presence vs. absence of depth-first reordering as described in Section 2.1.2. To help with this analysis, we also added some instrumentation to our implementation of
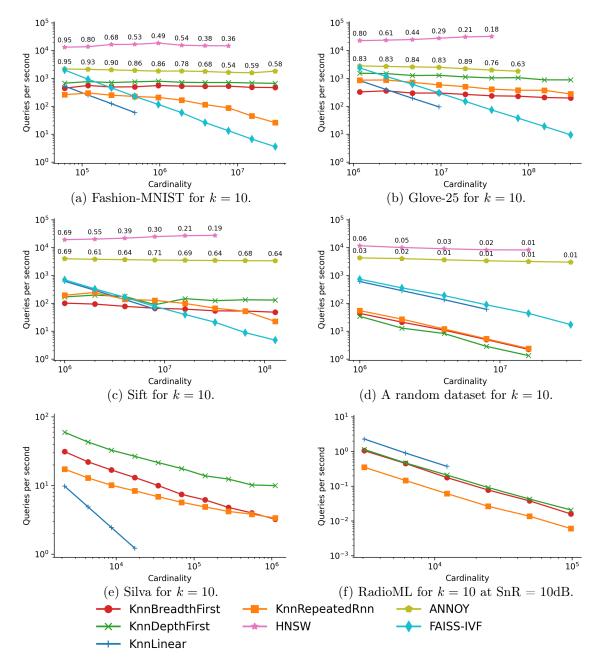
Figure 3: Throughput across six datasets, including a randomly-generated dataset. In each plot, the horizontal axis represents increasing cardinality of the dataset, while the vertical axis represents the throughput in queries per second (higher is better). For Fashion-MNIST, Glove-25, and Sift, as cardinality increases, the CAKES algorithms become faster than linear search but the cardinality at which this occurs differs by dataset. For Fashion-MNIST and Glove-25, Depth-First Sieve is consistently fastest, while for Sift, Repeated ρ-NN is the fastest for smaller cardinalities and Depth-First Sieve is the fastest for larger cardinalities. With Silva, we observe that for all algorithms, throughput initially seems to linearly decrease as cardinality increases, but that it starts to level off at higher cardinalities. Depth-First Sieve is consistently the fastest algorithm on the Silva dataset. For Radio-ML and Random, we see that all three CAKES algorithms are slower than naïve linear search, and that their throughput decreases linearly with cardinality. HNSW and ANNOY are the fastest algorithms on all four datasets we benchmarked them on, but their recall degrades quickly as cardinality increases on all datasets. In particular, on the Random dataset, HNSW and ANNOY have near-zero recall.

Table 4: Throughput (QPS) and Recall on the Sift dataset. A recall value of 1.000∗ denotes imperfect recall that rounds to 1.000. While recall for CAKES's does *not* degrade with cardinality, recall for HNSW and ANNOY degrades with cardinality. In contrast, CAKES has perfect recall on Sift, since it is used with Euclidean distance, which is a metric.

Mult.	HNSW		ANNOY		FAISS-IVF		CAKES	
	QPS	Recall	QPS	Recall	QPS	Recall	QPS	Recall
1	1.93×10^4	0.782	3.98×10^3	0.686	6.98×10^2	1.000*	6.20×10^2	1.000
2	2.03×10^4	0.552	3.80×10^3	0.614	3.30×10^2	1.000*	2.95×10^2	1.000
4	2.18×10^4	0.394	3.69×10^3	0.637	1.65×10^2	1.000*	1.76×10^2	1.000
8	2.48×10^4	0.298	3.58×10^3	0.710	7.72×10^1	1.000*	1.27×10^2	1.000
16	2.68×10^4	0.210	3.50×10^3	0.690	3.98×10^1	1.000*	1.47×10^2	1.000
32	2.75×10^4	0.193	3.44×10^3	0.639	2.09×10^1	0.999	1.24×10^2	1.000
64	–	–	3.39×10^3	0.678	8.87	0.997	1.34×10^2	1.000
128	–	–	3.36×10^3	0.643	4.78	0.993	1.31×10^2	1.000

Table 5: Throughput (QPS) and Recall on the Random dataset. A recall value of 1.000∗ denotes imperfect recall that rounds to 1.000. In contrast with the results on the ANN Benchmark datasets reported above, with the Random dataset, we observe that CAKES's algorithms perform quite slowly. As with the real datasets, HNSW and ANNOY are the fastest algorithms, and CAKES exhibits perfect recall at all cardinalities. HNSW and ANNOY exhibit *much* lower recall on this random dataset than on any of the ANN benchmark datasets.

Mult.	HNSW		ANNOY		FAISS-IVF		CAKES	
	QPS	Recall	QPS	Recall	QPS	Recall	QPS	Recall
1	1.17×10^4	0.060	4.28×10^3	0.028	7.342	1.000*	6.06×10^2	1.000
2	1.01×10^4	0.048	4.04×10^3	0.021	3.582	1.000*	2.75×10^2	1.000
4	9.12×10^3	0.031	3.64×10^3	0.014	1.902	1.000*	1.35×10^2	1.000
8	8.35×10^3	0.022	3.37×10^3	0.013	8.841	1.000*	6.13×10^1	1.000
16	8.25×10^3	0.008	3.17×10^3	0.006	4.361	1.000*	2.82×10^1	1.000
32	–	–	3.01×10^3	0.007	1.721	1.000*	1.31×10^1	1.000

CAKES to count the number of distance computations performed during search. We note that this instrumentation significantly slows down the wall-clock time of search, so it would not be used in a real-world application.

These four comparisons (referred to as Ball, BalancedBall, PermutedBall, and, Permuted-BalancedBall, where depth-first-reordering is referred to as Permuted for brevity) for all three search algorithms on the Fashion-MNIST dataset are shown in Figure 4. Notably, performance is not only strictly worse for the balanced approaches, but the asymptotic behavior is significantly worse; the algorithms (such as Depth-First Sieve) that seemed to exhibit constant-time behavior in Figure 3, which shows results using PermutedBall, no longer do so under a balanced clustering. We note that on this dataset, depth-first reordering (the Permuted variants) seems to have little effect on throughput; depth-first reordering is primarily intended to improve space complexity.

(a) Repeated ρ-NN

(b) Repeated ρ-NN

(c) Breadth First Sieve

(d) Breadth First Sieve

(e) Depth First Sieve

(f) Depth First Sieve

BalancedBall   PermutedBalancedBall
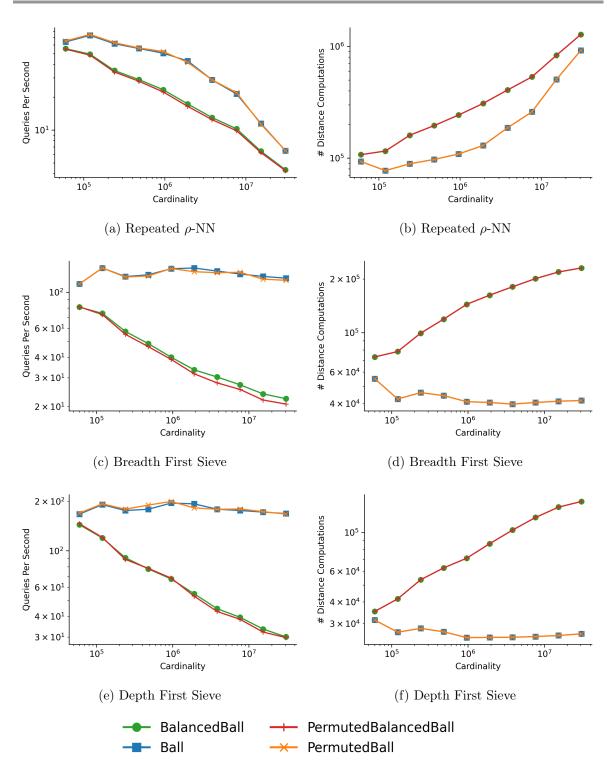Ball   PermutedBall

Figure 4: Number of distance computations across four clustering strategies and three search algorithms on the Fashion-MNIST dataset. Adding the instrumentation to count the number of distance computations had the side-effect of significantly slowing down the search algorithms compared to those reported in Figure 3. The left column shows the throughput in queries per second, while the right column shows the mean number of distance computations per query. The x-axis represents increasing cardinality of the dataset.

5. Discussion and Future Work. We have presented CAKES, a set of three algorithms for fast k-NN search that are generic over a variety of distance functions. CAKES's algorithms are exact when the distance function in use is a metric (see definition in Section 2). Even under cosine distance, which is not a metric, CAKES's algorithms exhibit nearly perfect recall. CAKES's algorithms are designed to be most effective when the data uphold the manifold hypothesis, or in other words, when the data are constrained to a low-dimensional manifold even when embedded in a high-dimensional space. CAKES uses an unbalanced clustering algorithm to build a binary tree that better captures the manifold structure of the data and provides for faster search performance. As a consequence, these algorithms do not perform well on data with random distributions, because of the absence of a manifold structure. In Figure 2, we show the extent to which each dataset we tested exhibits a manifold structure, as quantified by the percentiles of LFD.

5.1. Performance. As expected, our results from Figure 3 indicate that CAKES's algorithms scale sub-linearly with cardinality on real-world datasets with low LFD, and scale linearly on synthetic datasets with high LFD. CAKES's performance on Sift against that on the Random dataset illustrates this phenomenon well. As seen in Figures 2d and 2c, the LFD of the Random dataset is much higher than that of Sift, even though both datasets have the same cardinality and dimensionality. This is unsurprising, given that the Random dataset is uniformly distributed, while Sift is not. On the Random dataset, CAKES's speed decreases linearly as the multiplier increases, while with Sift, Depth-First Sieve and Breadth-First Sieve both exhibit nearly constant throughput. While Repeated ρ-NN does not exhibit near-constant throughput with Sift, throughput decreases much more slowly than it does with the Random dataset. Since these two datasets have the same cardinality and embedding dimension, the aforementioned discrepancies highlight how manifold structure affects algorithm performance.

We emphasize that although CAKES's algorithms are slower on the Random dataset, their recalls remain perfect. Figure 3 also shows that HNSW and ANNOY have near-constant throughput as the multiplier increases, but their recall continues to decrease as the multiplier increases. This, coupled with the increasing time and space cost of building the indices for HNSW and ANNOY, makes these algorithms unsuitable for real-world applications in which the datasets are growing exponentially. CAKES's tree is orders of magnitude faster and less memory-intensive to build, and CAKES's algorithms, while slower than HNSW and ANNOY, still show near-constant scaling as well as perfect recall.

When written with the same notation as used in Section 2.2.5, we see that the time complexity of Repeated ρ-NN is $\mathcal{O}(\frac{T}{\lceil d \rceil} + L)$, where T is the time complexity of tree-search and L is the time complexity of leaf-search. Even though Repeated ρ-NN has the least time cost (compared to $\mathcal{O}\left((T + L)\log{(T + L)}\right)$ for Breadth-First Sieve and $\mathcal{O}(T \log T + L \log k)$ for Depth-First Sieve) of the three CAKES algorithms, it is not always the fastest algorithm empirically. We believe that some of this discrepancy can be explained by the fact that Repeated ρ-NN can significantly "overshoots" the correct radius for k hits (ρ_k) during tree-search. This causes leaf-search to exhaustively search many more clusters than necessary, rendering the true scaling factor higher than that in Equation **??**. This overshot can occur when the LFDs of clusters near the query are not concentrated around their expectation. For example, if most clusters near the query have very low LFD except for one anomaly with very high LFD, the

harmonic mean LFD μ can still be low, so the factor of radial increase in (Equation) 2.3 may be much larger than necessary for guaranteeing k hits. This suggests that rather than using the reciprocal of the harmonic mean LFD in 2.3, we may achieve better results with a mean that is more sensitive to high outliers, such as the geometric mean. We leave it as an avenue for future work to characterize when Repeated ρ-NN significantly overestimates the correct radius ρ_k and to improve upon the factor of radial increase in Equation 2.3 so that this occurs less frequently and with less severity.

While the fastest CAKES algorithm can vary by dataset and cardinality, we note that on all three ANN-benchmark datasets (Fashion-MNIST, Glove-25, and Sift), Depth-First Sieve and Breadth-First Sieve show nearly constant throughput as the cardinality increases. This observation warrants further investigation, but it is especially promising given that both algorithms consistently outperform linear search for high cardinalities. Finally, we emphasize that the variation in performance of our algorithms across different datasets and cardinalities supports our use of an auto-tuning function (as discussed in Section 2.3) to select the best algorithm for a given dataset. Future work is needed to better characterize those datasets for which each algorithm performs best, but our results suggest that a more sophisticated auto-tuning function could be developed to select the fastest algorithm based on the properties of the dataset.

We also stress that CAKES's algorithms are designed to work well for *big* data, and our results support this claim; we observe that while linear search can outperform CAKES on datasets with small cardinality, CAKES always overtakes linear search at a large enough cardinality for each of the ANN-benchmark datasets we tested. Moreover, we observe that CAKES's algorithms outperform linear search at a lower cardinality when the dataset has a lower LFD. For example, Sift has much higher LFD than Fashion-Mnist and Glove-25, and we observe that CAKES overtakes linear search at a cardinality of 10^7 for Sift, as opposed to about 10^5 and 10^6 for Fashion-Mnist and Glove-25, respectively. For the Random dataset, which has the highest LFD, CAKES's algorithms *never* outperform linear search. These observations support our claim that CAKES performs well on datasets that arise from constrained generating phenomena, even as the cardinalities of these datasets grow exponentially.

5.2. Applicability. We note that despite the genericity of CAKES, effectively applying it to real-world datasets may require domain-specific knowledge. As an example, consider the domain of protein sequence search; protein sequences are of highly variable length (30–1000 amino acids are typical, with some outliers) and typically comprise a 20-letter alphabet. There have been numerous protein sequence search algorithms [21, 8, 37, 32], and the one thing they have in common besides the application domain is that their development required deep domain-specific knowledge. While simple edit distance may be a poor choice for clustering when sequence lengths exhibit such variance, other choices of distance metrics and preprocessing could be used by CAKES, such as representing each sequence as a set of k-mers (short substrings of length k) and choosing Jaccard distance on these sets as the metric; this approach was chosen by [21]. If the distance function uses a global sequence alignment algorithm, such as Needleman-Wunsch [26], the dataset may also be preprocessed into a number of mutually exclusive subsets based on the lengths of the sequences, and a separate CAKES tree may be built for each subset. At search time, the query sequence would be assigned to the appropriate

tree based on its length, and the search would be performed on that tree, and on a small number of trees on adjacent lengths' bins.

5.3. Future Work. The questions raised by this study suggest several additional avenues for future work. A comparison across more datasets is in order, as is further analysis with other distance functions that existing methods, such as FAISS, HNSW and ANNOY, do not support. These include Wasserstein distance [36] for probability distributions (particularly for high dimensional distributions) and Tanimoto distance [3] for comparing molecular structures by their maximal common subgraphs. Incorporating new distance functions in CAKES requires only that a Rust implementation of the distance function be provided.

Additionally, we intend to improve the data augmentation process described in Section 2.4 to better preserve the topological structure of the underlying dataset. In particular, we expect that favoring augmentation along the first few principal components of the local manifold may allow for more realistic data augmentation.

We also plan to investigate hierarchical data compression by representing differences at each level of the binary tree, particularly for string or genomic data, where all differences are discrete. Such an encoding would allow us to store the data in a compressed format, and to perform search on the compressed data without decompressing it in its entirety. This would allow us to perform fast search on datasets that are otherwise too large to fit in memory. An exploration of compression and compressed search is another avenue for future work.

We also would like to explore the use of CAKES in a streaming environment. This would require the ability to perform "online" updates to the tree as points are added to or deleted from the dataset. Such online updates would take advantage of the fast search algorithms provided by CAKES. CAKES can also be used to extend anomaly detection in CHAODA [16]. We could add to CHAODA's ensemble of graph-based anomaly detection methods by using the distribution of distances among the k nearest neighbors of cluster centers.

5.4. Availability. CLAM and CAKES are implemented in the Rust programming language, and the source code is available under an MIT license at https://github.com/URI-ABD/clam.

REFERENCES

[1] *Faiss-ivf.* https://github.com/facebookresearch/faiss/wiki/Faiss-indexes, 2016.

[2] M. Aumüller, E. Bernhardsson, and A. Faithfull, *Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms*, Information Systems, 87 (2020), p. 101374.

[3] D. Bajusz, A. Rácz, and K. Héberger, *Why is tanimoto index an appropriate choice for fingerprint-based similarity calculations?*, Journal of cheminformatics, 7 (2015), pp. 1–13.

[4] E. Bernhardsson, *Annoy.* https://github.com/spotify/annoy, 2015.

[5] M. Boguna, D. Krioukov, and K. C. Claffy, *Navigability of complex networks*, Nature Physics, 5 (2009), pp. 74–80.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language Models are Few-Shot Learners*, arXiv e-prints, (2020), https://doi.org/10.48550/arXiv.2005.14165.

[7] T. M. Cover, P. Hart, et al., *Nearest neighbor pattern classification*, IEEE Transactions on Information Theory, 13 (1967), pp. 21–27.

[8] N. M. Daniels, A. Gallant, J. Peng, L. J. Cowen, M. Baym, and B. Berger, *Compressive genomics for protein databases*, Bioinformatics, 29 (2013), pp. i283–i290.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al., *An image is worth 16x16 words: Transformers for image recognition at scale*, arXiv preprint arXiv:2010.11929, (2020).

[10] C. Fefferman, S. Mitter, and H. Narayanan, *Testing the manifold hypothesis*, Journal of the American Mathematical Society, 29 (2016), pp. 983–1049.

[11] E. Fix and J. L. Hodges Jr, *Discriminatory analysis-nonparametric discrimination: Small sample performance*, tech. report, California Univ Berkeley, 1952.

[12] J. Gao and C. Long, *High-dimensional approximate nearest neighbor search: with reliable and efficient distance comparison operations*, Proceedings of the ACM on Management of Data, 1 (2023), pp. 1–27.

[13] O. Gold and M. Sharir, *Dynamic time warping and geometric edit distance: Breaking the quadratic barrier*, ACM Transactions on Algorithms (TALG), 14 (2018), pp. 1–17.

[14] C. A. Hoare, *Algorithm 65: find*, Communications of the ACM, 4 (1961), pp. 321–322.

[15] L.-Y. Hu, M.-W. Huang, S.-W. Ke, and C.-F. Tsai, *The distance function effect on k-nearest neighbor classification for medical datasets*, SpringerPlus, 5 (2016), pp. 1–9.

[16] N. Ishaq, T. J. Howard, and N. M. Daniels, *Clustered hierarchical anomaly and outlier detection algorithms*, in 2021 IEEE International Conference on Big Data (Big Data), IEEE, 2021, pp. 5163–5174.

[17] N. Ishaq, G. Student, and N. M. Daniels, *Clustered hierarchical entropy-scaling search of astronomical and biological data*, in 2019 IEEE International Conference on Big Data (Big Data), IEEE, 2019, pp. 780–789.

[18] J. Johnson, M. Douze, and H. Jégou, *Billion-scale similarity search with GPUs*, IEEE Transactions on Big Data, 7 (2019), pp. 535–547.

[19] S. D. Kahn, *On the future of genomic data*, Science, 331 (2011), pp. 728–729.

[20] A. Kent, R. Sacks-Davis, and K. Ramamohanarao, *A signature file scheme based on multiple organizations for indexing very large text databases*, Journal of the american society for information science, 41 (1990), pp. 508–534.

[21] Y. Kim, Z. Guo, J. A. Robertson, B. Reidys, Z. Zhang, and L. S. Heath, *Entrance: Exploration of entropy scaling ball cover search in protein sequences*, bioRxiv, (2021), https://doi.org/10.1101/2021.05.31.446458, https://www.biorxiv.org/content/early/2021/05/31/2021.05.31.446458, https://arxiv.org/abs/https://www.biorxiv.org/content/early/2021/05/31/2021.05.31.446458.full.pdf.

[22] J. M. Kleinberg, *Navigation in a small world*, Nature, 406 (2000), pp. 845–845.

[23] V. I. Levenshtein et al., *Binary codes capable of correcting deletions, insertions, and reversals*, in Soviet physics doklady, vol. 10, Soviet Union, 1966, pp. 707–710.

[24] Y. Malkov and D. A. Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 42 (2016), pp. 824–836, https://api.semanticscholar.org/CorpusID:8915893.

[25] M. Müller, *Dynamic time warping*, Information retrieval for music and motion, (2007), pp. 69–84.

[26] S. B. Needleman and C. D. Wunsch, *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of molecular biology, 48 (1970), pp. 443–453.

[27] OpenAI, *Gpt-4 technical report*, ArXiv, abs/2303.08774 (2023).

[28] T. J. O'Shea, T. Roy, and T. C. Clancy, *Over-the-air deep learning based radio signal classification*, IEEE Journal of Selected Topics in Signal Processing, 12 (2018), pp. 168–179, https://doi.org/10.1109/JSTSP.2018.2797022.

[29] C. Quast, E. Pruesse, P. Yilmaz, J. Gerken, T. Schweer, P. Yarza, J. Peplies, and F. O. Glöckner, *The SILVA ribosomal RNA gene database project: improved data processing and web-based tools*, Nucleic Acids Research, 41 (2012), pp. D590–D596, https://doi.org/10.1093/nar/gks1219.

[30] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al., *Learning transferable visual models from natural language supervision*, in International Conference on Machine Learning, PMLR, 2021, pp. 8748–8763.

[31] R. Sacks-Davis, A. Kent, and K. Ramamohanarao, *Multikey access methods based on superimposed coding techniques*, ACM Transactions on Database Systems (TODS), 12 (1987), pp. 655–696.

[32] M. Steinegger and J. Söding, *Clustering huge protein sequence sets in linear time*, Nature communications, 9 (2018), p. 2542.

[33] S. Suyanto, P. E. Yunanto, T. Wahyuningrum, and S. Khomsah, *A multi-voter multi-commission nearest neighbor classifier*, Journal of King Saud University - Computer and Information Sciences, 34 (2022), pp. 6292–6302, https://doi.org/10.1016/j.jksuci.2022.01.018.

[34] H. Touvron, L. Martin, K. R. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. M. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. S. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. M. Kloumann, A. V. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, *Llama 2: Open foundation and fine-tuned chat models*, ArXiv, abs/2307.09288 (2023), https://api.semanticscholar.org/CorpusID:259950998.

[35] N. Ukey, Z. Yang, B. Li, G. Zhang, Y. Hu, and W. Zhang, *Survey on exact knn queries over high-dimensional data space*, Sensors, 23 (2023), p. 629.

[36] S. Vallender, *Calculation of the wasserstein distance between probability distributions on the line*, Theory of Probability & Its Applications, 18 (1974), pp. 784–786.

[37] Y. W. Yu, N. Daniels, D. C. Danko, and B. Berger, *Entropy-scaling search of massive biological data*, Cell Systems, 1 (2015), pp. 130–140.

[38] J. Zhang, T. Wang, W. W. Ng, and W. Pedrycz, *Ensembling perturbation-based oversamplers for imbalanced datasets*, Neurocomput., 479 (2022), p. 1–11, https://doi.org/10.1016/j.neucom.2022.01.049.