A Simple Algorithm for kNN Sampling in General Metrics*

Kirk P. Gardner† and Donald R. Sheehy‡

Abstract

Finding the kth nearest neighbor to a query point is a ubiquitous operation in many types of metric computations, especially those in unsupervised machine learning. In many such cases, the distance to k sample points is used as an estimate of the local density of the sample. In this paper, we give an algorithm that takes a finite metric $(P, \mathbf{d})$ and an integer k and produces a subset $S \subseteq P$ with the property that for any $q \in P$, the distance to the second nearest point of S to q is a constant factor approximation to the distance to the kth nearest point of P to q. Thus, the sample S may be used in lieu of P. In addition to being much smaller than P, the distance queries on S only require finding the second nearest neighbor instead of the kth nearest neighbor. This is a significant improvement, especially because theoretical guarantees on kth nearest neighbor methods often require k to grow as a function of the input size n.

1 Introduction

Subsampling is a fundamental step in many large scale data analysis problems. The goal is that the distribution of the subsample $M \subset P$ resembles the distribution on the whole data set, P. On the one hand, preserving purely statistical properties is often achieved by random sampling. This paper, on the other hand, considers preserving metric properties of a subsample. Specifically, we want a sample where the distance to the second nearest neighbor in M is approximately the distance to the kth nearest neighbor in P. We call this a *kth nearest neighbor sample* and it balances between competing demands of representing the underlying distribution and the underlying metric.

Figure 3 shows a kth nearest neighbor sample of a collection of points in the plane. The points are 5 times denser on the right half and there are correspondingly more points in the sample on that side. Figure 1 shows a random sample of the same point set with the same number of points sampled. The random sample also has more points on the denser half, but it has more variability in the distance between samples; some are virtually on top of each other. The greedy sample (Figure 2) is a standard way to produce a uniform sample at a particular scale, but that scale does not vary with the density. In this sense, the kNN sample achieves a balance between the random sample and the uniform (i.e. greedy) sample.

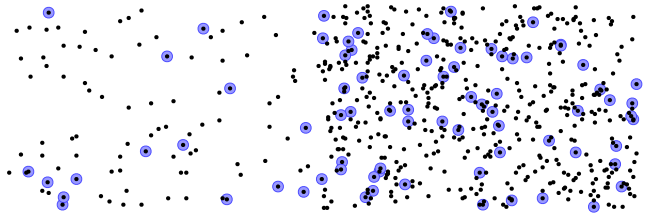

Figure 1: A random sample.

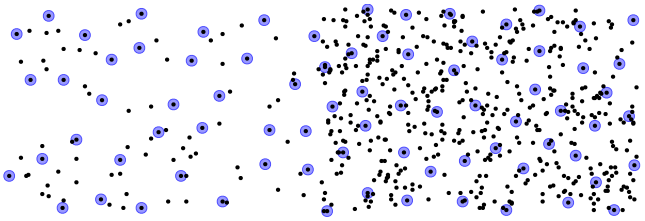

Figure 2: A greedy sample.

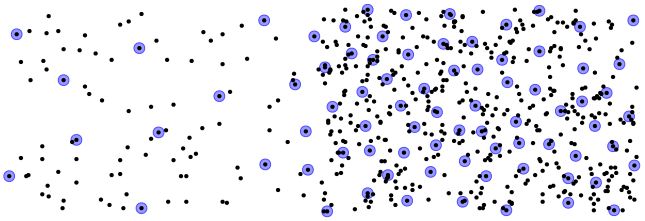

Figure 3: A kth nearest neighbor sample.

In prior work [7], we showed how a variation of Delaunay refinement can be used in the plane to compute such a sample. In this paper, we prove that a simple algorithm can compute a kNN sample in any metric space. Then, we show how data structures and ideas from nearest neighbor search can be adapted to speed up the computation.

The algorithm is presented in its simplest form in Section 3. There, we prove upper and lower bounds on the kth nearest neighbor distance in terms of the

*This work was partially supported by the NSF under grants CCF-1464379, CCF-1525978, and CCF-1652218.

†Department of Computer Science, North Carolina State University, kpgardn2@ncsu.edu

‡Department of Computer Science, North Carolina State University, don.r.sheehy@gmail.com

2nd nearest neighbor distance in the sample. Then, in Section 4, we describe a neighborhood graph structure adapted from Clarkson [3] that we use to speed up the local search step of the algorithm. In Section 5, we put all these pieces together to bound the overall running time. We report on our open source implementation and give some demonstrations of the code in action in Section 7. Finally, in Section 8, we propose some open problems that remain.

1.1 Related Work

The distance to the kth nearest neighbor has been used for a long time as a density estimator (see Biau and Devroye [1]). It has also been used in pointwise estimates of the local density in metric measure spaces (see Cutler and Dawson [5] and the survey by Clarkson [4]). There are many data structures that compute kth nearest neighbors efficiently, perhaps the fastest in practice are the Faster Cover Trees of Izbicki and Shelton [11]. There has also been theoretical work by Har-Peled and Raichel on a general framework for computing aggregate statistics like the kth nearest neighbor distance with the so-called Net and Prune paradigm [10]. The first algorithms for computing kNN samples was the work of Gardner and Sheehy using Delaunay refinement, but was limited to Euclidean space [7]. The main algorithmic paradigm we use here is based on greedy orderings, which have been used since the 1980's for approximate k-center clustering [6, 8]. The underlying data structure is a variant of the *sb* structure of Clarkson [3, 2], which was first analyzed for greedy orderings by Har-Peled and Mendel [9].

2 Background

We will deal with finite subsets of metric spaces $(X, \mathbf{d})$, where the X is the set of points and $\mathbf{d}$ is the metric. For $x \in X$ and subsets S of X, we define

$$\mathbf{d}(x, S) := \min_{s \in S} \mathbf{d}(x, s).$$

The minimum distance to k points in S is denoted

$$\mathbf{d}_{S,k}(x) := \min_{U \in \binom{S}{k}} \max_{y \in U} \mathbf{d}(x, y).$$

In particular, $\mathbf{d}_{S,1}(x) = \mathbf{d}(x, S)$. The *Hausdorff distance*, $\mathbf{d}_H$, is defined on subsets of X as

$$\mathbf{d}_H(A, B) := \max\{\max_{a \in A} \mathbf{d}(a, B), \max_{b \in B} \mathbf{d}(b, A)\}.$$

A *metric ball in* S is the set of points of S within a fixed radius of a point in S. A *minimum r-cover* of a set S is the smallest set of centers of metric balls of radius r whose union contains S. Equivalently, it is the smallest subset $C \subset S$ such that $\mathbf{d}_H(S, C) \leq r$.

The *doubling constant* of a metric is the size of the largest minimum r-cover of any ball of radius $2r$. The *doubling dimension* is the base-two logarithm of the doubling constant. A bound on the doubling dimension allows one to apply the kind of packing arguments as are often used in Euclidean space.

The *spread* $\Delta(P)$ of a finite metric P is the ratio of the largest to smallest pairwise distances. The *k-spread* is the ratio

$$\Delta_k(P) := \frac{\max_{p,q \in P} \mathbf{d}(p, q)}{\min_{p \in P} \mathbf{d}_{P,k}(p)}.$$

A set with spread Δ in a metric with doubling dimension d has at most $O(\Delta^d)$ points [9].

An *r-packing* is a set of points for which the minimum pairwise distance is at least r. An r-packing that is also an r-cover is called an *r-net*.

For a metric space X and a subset P, an *(α, β)-kNN sample* of P is a subset $M \subseteq P$ with the property that for all $x \in X$,

$$\alpha \mathbf{d}_{P,k}(x) \leq \mathbf{d}_{M,2}(x) \leq \beta \mathbf{d}_{P,k}(x).$$

We will refer to it simply as a kNN sample when the values of α and β are not important. The algorithm in this paper produces a $(1/5, 2)$-kNN sample.

Let $P = (p_1, \ldots, p_n)$ be an ordered subset of X. Let $P_i = \{p_1, \ldots, p_i\}$ be the ith *prefix*. The ordering is *greedy* if for all $i \in 2, \ldots, n$, we have

$$\mathbf{d}(p_{i+1}, P_i) := \mathbf{d}_H(P, P_i).$$

In other words, every point p_{i+1} is the farthest point from P_i. We will compute kNN-samples in a greedy order. Both the data structure in Section 4 and the algorithm in Section 5 will depend on this ordering for their correctness and efficient running time.

3 A Simple Algorithm

We present a simple algorithm that generates an (α, β)-kNN-sample as a subset of the input. It iteratively builds M from P by adding a point p of P to M as long as $\mathbf{d}_{P,k}(p) \leq 2\mathbf{d}_{M,1}(p)$. In this section, we prove that any such algorithm produces a $(1/5, 2)$-kNN-sample.

Algorithm 1 kNNSample(P, k)

```
M ← ∅
while ∃p ∈ P such that d_{P,k}(p) ≤ 2d_{M,1}(p) do
    Add p to M
return M
```

Theorem 1 *Let P be a subset of a metric space X. Let $M =$ kNNSample(P, k) for some $k \geq 2$. Then,*

$$\frac{1}{5}\mathbf{d}_{P,k} \leq \mathbf{d}_{M,2} \leq 2\mathbf{d}_{P,k}.$$

Proof. We first prove the lower bound on $\mathbf{d}_{M,2}$. Let $x \in P$ be any point. Let v_1 and v_2 be the two nearest points to x in M, so,

$$\mathbf{d}_{M,1}(x) = \mathbf{d}(x, v_1) \leq \mathbf{d}(x, v_2) = \mathbf{d}_{M,2}(x). \tag{1}$$

Let $v \in \{v_1, v_2\}$ be whichever point was added later by the algorithm, which guarantees that

$$\mathbf{d}_{P,k}(v) \leq 2\mathbf{d}(v_1, v_2). \tag{2}$$

We can now bound $\mathbf{d}_{P,k}(x)$ as follows.

$$\begin{aligned}
\mathbf{d}_{P,k}(x) &\leq \mathbf{d}_{P,k}(v) + \mathbf{d}(v, x) && \left[\mathbf{d}_{P,k} \text{ is 1-Lipschitz}\right] \\
&\leq 2\mathbf{d}(v_1, v_2) + \mathbf{d}(v, x) && \left[\text{by (2)}\right] \\
&\leq 2(\mathbf{d}(v_1, x) + \mathbf{d}(x, v_2)) + \mathbf{d}(v, x) && \left[\text{triangle inequality}\right] \\
&\leq 5\mathbf{d}_{M,2}(x) && \left[\text{by (1)}\right].
\end{aligned}$$

Now, we prove the upper bound on $\mathbf{d}_{M,2}$. Suppose for contradiction that there exists a point $x \in X$ such that $\mathbf{d}_{M,2}(x) > 2\mathbf{d}_{P,k}(x)$. Let S be the k closest points in P to x. Let r be the radius of the minimum enclosing ball of S and note that $r \leq \mathbf{d}_{P,k}(x)$. If $\mathbf{d}_{M,1}(s) < r$ for some $s \in S$, then let $m \in M$ be the nearest neighbor of s in M. It follows from the triangle inequality that $\mathbf{d}(x, m) \leq \mathbf{d}(x, s) + \mathbf{d}(s, m) \leq 2\mathbf{d}_{P,k}(x) < \mathbf{d}_{M,2}(x)$. There can be only one point $m \in M$ whose distance to x is less than $\mathbf{d}_{M,2}(x)$. However, if $\mathbf{d}_{M,1}(s) < r$ for all $s \in S$, then there would be a smaller minimum enclosing ball for S centered at m, contradicting our choice of r. So, for at least one point $s_* \in S$, we have $\mathbf{d}_{M,1}(s_*) \geq r$. Therefore, by the triangle inequality, $\mathbf{d}_{P,k}(s_*) \leq 2r \leq 2\mathbf{d}_{M,1}(s_*)$. The existence of such a point would cause the algorithm to add s_* to M, contradicting the assumption that $M = \textsc{kNNSample}(P, k)$. Thus, we conclude that no such x exists and indeed $\mathbf{d}_{M,2} \leq 2\mathbf{d}_{P,k}$. □

In Sections 5 and 6, we explain how to efficiently test this condition and bound the running time. Efficiency is achieved by constructing the sample in a greedy order. Note that the simplified algorithm does not rely on a particular ordering.

4 A Cluster Graph

In this section we will define a graph on the a subset of points that can be used to rapidly shrink the search space when computing $\mathbf{d}_{P,k}$ in the middle of our algorithm. It is a variation on a data structure used in the construction of the *sb* data structure of Clarkson [3]. Each vertex in the graph will be a point that we have already added to our sample. Moreover, each vertex will track the uninserted points that are closest to it. Every time a new point is considered for addition, we use the vertices adjacent to its nearest neighbor to search for nearby points. The formal definition is given below.

Let P be a finite metric space, and let $M \subset P$ be any subset. The *cluster* of $x \in M$ is the set of points C_x in P that are closer to x than to any other point in M. We break ties arbitrarily but consistently. The clusters are discrete Voronoi cells.

The *cluster graph* G_M on vertex set M has points a and b adjacent if there exist $a' \in C_a$ and $b' \in C_b$ such that

$$\mathbf{d}(a', b') \leq 2\max\{\mathbf{d}(a', a), \mathbf{d}(b', b)\}.$$

If a and b are adjacent, we denote this as $a \sim b$. The graph has self loops at every vertex. Because $a' \in C_a$ and $b' \in C_b$, the adjacency condition is equivalent to

$$\mathbf{d}(a', b') \leq \max\{2\mathbf{d}_{M,1}(a'), 2\mathbf{d}_{M,1}(b')\}.$$

So, if we wanted to try to add a' to M, we could find all the points within distance $2\mathbf{d}_{M,1}(a')$ among the the clusters of the vertices adjacent to a.

Moreover, we use the cluster graph to efficiently maintain itself under insertions. That is, we can use the graph to quickly find the edges incident to a newly inserted point.

Lemma 2 *Let $M \subset P$ and let $M' = M \cup \{a'\}$ for $a' \in C_a$ and $a \in M$. If $a' \sim c$ in $G_{M'}$, then there exists $b \in M$ such that $a \sim b \sim c$ in G_M. In other words, the neighbors of a' will be found among the neighbors of neighbors of a.*

Proof. By the definition of adjacency in $G_{M'}$ there exist points $b' \in C_{a'}$ and $c' \in C_c$ such that

$$\begin{aligned}
\mathbf{d}(b', c') &\leq 2\max\{\mathbf{d}(b', a'), \mathbf{d}(c', c)\} \\
&\leq 2\max\{\mathbf{d}(b', b), \mathbf{d}(c', c)\},
\end{aligned}$$

where b is the nearest point to b' in M as illustrated in Figure 4. From this, it follows that $b \sim c$ in G_M. Next, observe that

$$\mathbf{d}(a', b') \leq \mathbf{d}(b, b') \leq 2\max\{\mathbf{d}(b, b'), \mathbf{d}(a, a')\}.$$

So, it immediately follows that $a \sim b$ in G_M. □

Lastly, we would prefer to avoid the extensive checking required to see if two points are adjacent. At first sight, it seems to require searching for a pair that are particularly close. Instead, we follow the example of Clarkson [3] and prove a sufficient condition for bounding the neighbors just by using the radii of the clusters and the triangle inequality.

For a point $x \in M$, define its radius to be

$$\text{rad}(x) = \max_{y \in C_x} \mathbf{d}(y, x).$$

In other words, it is the distance from x to the farthest point in its cluster.

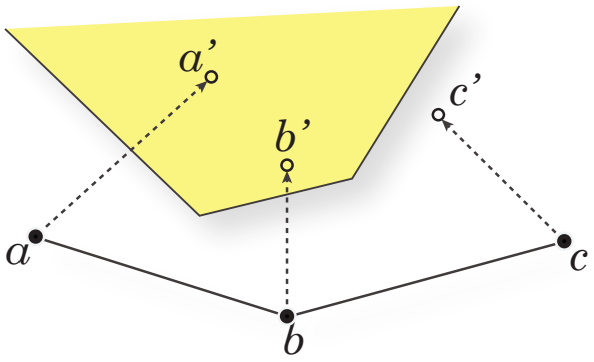

Figure 4: Here, $a' \in C_a$, $b' \in C_b$, and $c' \in C_c$. If adding a' would require creating an edge from a' to c, we will find c among the neighbors of neighbors of a. The cluster of a' is shown. The same pair (b', c') that witnesses a cluster graph edge from a' to c also guarantees the existence of the edge from b to c.

Lemma 3 *If $a \sim b$ in G_M, then*

$$\mathbf{d}(a,b) \leq \text{rad}(a) + \text{rad}(b) + 2\max\{\text{rad}(a), \text{rad}(b)\}.$$

Proof. If $a \sim b$, then there exist points $a' \in C_a$ and $b' \in C_b$ such that

$$\mathbf{d}(a',b') \leq 2\max\{\mathbf{d}(a,a'), \mathbf{d}(b,b')\}.$$

We simply observe that $\mathbf{d}(a,a') \leq \text{rad}(a)$ and $\mathbf{d}(b,b') \leq \text{rad}(b)$, so the result follows from the triangle inequality. □

Using the lemma above, we can quickly check if an edge ought to be removed from our cluster graph as new points are added and radii decrease. It does mean that the graph we store will contain more edges than required. This distance condition is precisely what is needed to bound the space usage as long as the points are added in a greedy order.

Lemma 4 *If M is a prefix of a greedy ordering of $S \subseteq P$, then the cluster graph on M has maximum degree γ^3, where γ is the doubling constant of the metric.*

Proof. Let r be the maximum radius among all the clusters in the cluster graph. By Lemma 3, if $a \sim b$ in G_M, then $\mathbf{d}(a,b) \leq r_a + r_b + 2\max\{r_a, r_b\} \leq 4r$. So, the neighbors of any point $a \in M$ are contained in $\texttt{ball}(a, 4r)$. Because the points are added in a greedy order, no two points of M have distance less than r. By the definition of γ, the doubling constant, $\texttt{ball}(a, 4r)$ can be covered by γ^3 balls of radius $r/2$. Each such ball contains at most one point of M. Therefore, there are at most γ^3 neighbors. □

Corollary 5 *Updating the neighbors of the vertices in a cluster graph takes constant time per insertion in doubling metrics if one adds points in a greedy order.*

This last lemma and its corollary show the importance of using the greedy order. It makes the search for nearby neighbors efficient. The algorithm described in the following section will construct M in a greedy order. This greedy order is also important to the efficiency of the algorithm in other ways as we will see in the analysis.

5 Efficiently Computing the GreedykNN Algorithm

The biggest challenge in implementing the simple algorithm of Section 3 is that it requires computing or at least bounding $\mathbf{d}_{M,2}$ and $\mathbf{d}_{P,k}$ for every point.

As explained in Section 4, the main data structure is a kind of discrete Voronoi diagram. For each inserted point p, it has two parts: first, it stores the cluster C_p; second, it stores the neighbors of p, the other inserted points within some distance. This data structure will be used to guarantee that the points in the output are discovered in a greedy ordering.

The efficiency improvements in the algorithm over a linear search are achieved by only searching locally in the cluster graph. Thus, to prove the algorithm is correct, we need to show that it is sufficient to only search the neighborhood graph in order to bound $\mathbf{d}_{P,k}$ in terms of $\mathbf{d}_{M,1}$.

After i points have been added, the inserted points are denoted M_i and the cluster graph is denoted G_i. The neighbors of a point $q \in M_i$ in G_i are denoted $N_i(q)$. We include q itself in $N_i(q)$ for all $q \in M_i$.

If we consider adding a point p, but do not add it because $\mathbf{d}_{P,k}(p) > 2\mathbf{d}_{M,1}(p)$, then this condition will continue to hold as we add more points to M. That is, if we decide not to add p at time i, there will not later come a time when we do want to add it. So, we can safely remove this point from the data structure. We say such a point is marked. In addition to the cluster graph, we store for each point, a list of potentially near points that have been marked. For a point p, we call this list the *nearby marked list of p* and denote it $\texttt{markedpts}(p)$. These lists are used to correctly bound $\mathbf{d}_{P,k}$ for new points considered later in the algorithm (as explained below).

The algorithm works as follows. Start by adding any point to M. All points start unmarked. At step $i+1$, consider adding the farthest unmarked point p to any point in M_i. A heap in the cluster graph makes it easy to find this point. Let $q = \text{NN}_{M_i}(p)$ so $p \in C_q$. Count the number of points in the ball $B = \texttt{ball}(p, 2\mathbf{d}_{M_i,1}(p))$ by iterating over the clusters of q and its neighbors. We also check the nearby marked list of p to count the marked points in B. If there are at least k points in B, then we add p to our kNN sample and continue.

If there are fewer than k points in B, then we mark p and remove it from its cluster. For each of the points

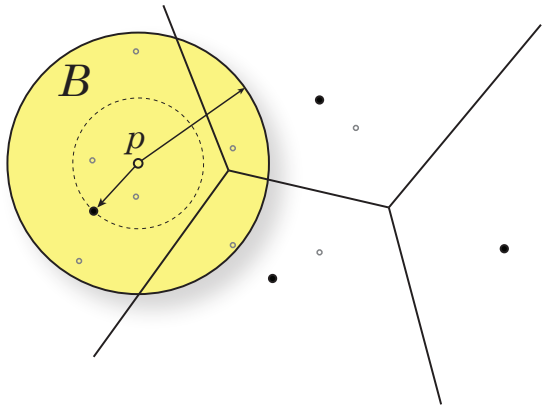

Figure 5: When considering the addition of point p, we count points in $B = \texttt{ball}(p, 2\mathbf{d}_{M,1}(p))$. These are all among the cluster adjacent to the cluster containing p and the list $\texttt{markedpts}(p)$. The clusters are shown as Voronoi cells as a visual aid.

in B that have not yet been added, they may have p nearby when they are considered for insertion later. We add p to the nearby marked list of each point in B. As the following lemma shows, this combination of cluster graph plus nearby marked lists is sufficient to enumerate B.

Lemma 6 *When considering the addition of a point $p \in C_a$ into M_j, the set of points P in $B = \texttt{ball}(p, 2\mathbf{d}_{M_j,1}(p))$ is contained in*

$$\left(\bigcup_{b \sim a} C_b\right) \cup \texttt{markedpts}(p).$$

Proof. By the definition of the cluster graph, the clusters adjacent to a contain all the points that can be within $2\mathbf{d}_{M_j,1}(p)$ of p. However, some points q are removed from the clusters if at some time $i < j$, they were marked because $\mathbf{d}_{P,k}(q) > 2\mathbf{d}_{M_i,1}(q)$. Because the points are considered for insertion in a greedy ordering, we have that if $q \in B$, then

$$\mathbf{d}(p, q) \leq 2\mathbf{d}_{M_j,1}(p) \leq 2\mathbf{d}_{M_i,1}(q).$$

Therefore, q would have been added to $\texttt{markedpts}(p)$ at the time q was marked. □

6 Analysis

In this section we will show that the algorithm of Section 5 computes a kNN sample in $O(kn \log \Delta)$ time for doubling metrics.

The key step of the analysis is to show that each point is touched at most $O(k)$ times before the insertion radius goes down by a constant factor. This is similar to Har-Peled and Mendel's analysis [9] of Clarkson's algorithm except that in our case a point may be considered but not added. The volume packing argument is easily adapted to this case with the loss of a factor of k in the running time.

A point q is *touched* when considering insertion of a point p if we compute $\mathbf{d}(p, q)$. The analysis depends on counting these touches.

Lemma 7 *A point b is touched at most $O(k \log \Delta_k(P))$ times when computing the kNN sample of a doubling metric P.*

Proof. Partition the set of points that touch b into sets A_i where the maximum radius in the cluster graph is in the interval $[2^i, 2^{i+1})$ at the time of a touch from $a \in A_i$. It will suffice to show that there are only $O(k)$ points in each A_i, because at most $O(\log(\Delta_k(P)))$ sets A_i are nonempty. Let S_i be a 2^{i-1} covering of A_i. We have $|A_i| \leq k|S_i|$, because if any point of S_i had more than k points of A_i in its ball of radius 2^{i-1}, one of them would have been inserted and therefore another of them would have been considered for insertion at a time when the radius is less than 2^i, contradicting the assumption that the point is in A_i.

Moreover, for all $a \in A_i$, $\mathbf{d}(a, b) \leq 6 \cdot 2^{i+1}$ by the triangle inequality and the definition of edges in the cluster graph. By the usual packing argument (see Har-Peled and Mendel [9]), this implies that

$$|A_i| \leq k|S_i| \leq k\gamma^{\lceil \log_2 24 \rceil} = O(k).$$

□

Theorem 8 *Let P be a finite metric space with size n, doubling dimension d, and spread Δ. The Greedy kNN sampling algorithm runs time $O(kn \log \Delta)$.*

Proof. The algorithm has several different pieces that must be analyzed separately. In the main loop, there are some heap operations to find the next point to consider for addition. This requires $O(\log n)$ time per point. The local search requires touching all the points in a cluster and its neighbors. This requires $O(k \log \Delta_k(P))$ time per point according to Lemma 7. It also requires touching all the points in the nearby marked list. As each point is added to at most $k - 1$ such lists, there are $O(kn)$ touches of this type. By Corollary 5, updating the cluster graph requires only constant time per vertex. Because $n = O(\Delta^d)$, we have $\log n = O(\log \Delta)$. Using the fact that $\Delta_k \leq \Delta$, the total running time is $O(kn \log \Delta)$. □

7 Software

We have implemented the GreedykNN algorithm and integrated it into the `greedypermutations` python

package. The code can be accessed at `https://github.com/donsheehy/greedypermutation` and the documentation at `https://donsheehy.github.io/greedypermutation/`. The code can be installed with `pip` by running the following from a command line.

```
pip install greedypermutation
```

Here are several examples of the code in use. We start with an example of exponentially-spaced points.

```
from greedypermutation.knnsample import knnsample

P = [Point(1.2**i, 5) for i in range(10, 100)]
S_10 = list(knnsample(P,10))
```

Next, we show uniform points with $k = 10$.

```
P = [Point(i, 5) for i in range(10, 600, 10)]
S_10 = list(knnsample(P,10))
```

The next instance, with $k = 20$ is predictably twice as sparse.

```
P = [Point(i, 5) for i in range(10, 600, 10)]
S_20 = list(knnsample(P,20))
```


8 Conclusion

This paper presented a simple greedy approach to computing kNN samples that have distances between the points bounded within a constant times the distance to k points locally. This balances the desire to have a geometrically nice sample, but also one whose density varies with the underlying distribution.

There are several open problems that remain. First, it is not known whether there are bounds on α and β that are necessary for an (α, β)-kNN sample to exist. From the results in this paper, we know that $(1/5, 2)$-kNN samples exist for any finite metric space, but nothing is known for larger α and smaller β. Also open is whether one can efficiently compute a variation where one uses $\mathbf{d}_{M,k'}$ for some $k' < k$ other than 2. It may be that one can give a tighter approximation, but this question is also open.

Another open question is whether or not there are more efficient algorithms for implementing the simple heuristic of Section 3. In particular, it may be possible to cut out or reduce the factor of k. The reason this seems possible is that it comes from enumerating rather than just counting the points in the metric ball around each point that are considered for addition.

References

[1] G. Biau and L. Devroye. *Lectures on the Nearest Neighbor Method*. Springer Series in the Data Sciences. Springer, 2015.

[2] K. L. Clarkson. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.

[3] K. L. Clarkson. Nearest neighbor searching in metric spaces: Experimental results for 'sb(s)'. Preliminary version presented at ALENEX99, 2003.

[4] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In G. Shakhnarovich, T. Darrell, and P. Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006.

[5] C. D. Cutler and D. A. Dawson. Estimation of dimension for spatially distributed data and related limit theorems. *Journal of Multivariate Analysis*, 28(1):115–148, 1989.

[6] M. Dyer and A. Frieze. A simple heuristic for the p-centre problem. *Operations Research Letters*, 3(6):285–288, 1985.

[7] K. Gardner and D. Sheehy. kth nearest neighbor sampling in the plane. In *Proceedings of the Canadian Conference on Computational Geometry*, 2016.

[8] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38:293–306, 1985.

[9] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

[10] S. Har-Peled and B. Raichel. Net and prune: A linear time algorithm for euclidean distance problems. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, 2013.

[11] M. Izbicki and C. R. Shelton. Faster cover trees. In *Proceedings of the Thirty-Second International Conference on Machine Learning*, 2015.

Appendix

9 How small is the sample?

Ideally, one hopes that the sample M is much smaller than P. If P has n points, then $2n/k$ points is the goal, having 2 points in the sample for k points in the input. Allowing some constant factors we want $|M|$ to be $O(n/k)$. There are clearly cases where this is achieved. For example, points spaced uniformly on a line will have a kNN sample that is uniformly spaced and has size $O(n/k)$.

Unfortunately, there are also simple examples where a kNN sample will be large. One such example is exponentially spaced points on a line, such as $\{10^i \mid i = 1, \ldots, n\}$.

The size decrease is best understood in terms of the measure induced by a kth nearest neighbor density estimate. Let's continue with one dimensional examples. Let $q : [0, 1] \to \mathbb{R}$ be a density function for a measure

$$\mu(B) := \int_{x \in B} q(x)dx.$$

with total mass n, i.e. $\mu([0, 1]) = n$.

The k-th nearest neighbor density estimate constructed from P induces the following approximation to this measure.

$$\mu_{P,k}(B) := \int_{x \in B} \frac{k}{\mathbf{d}_{P,k}(x)}dx.$$

Similarly, we get an estimate from M.

$$\mu_{M,2}(B) := \int_{x \in B} \frac{2}{\mathbf{d}_{M,2}(x)}dx.$$

If M satisfies the kNN sampling lower bound $\alpha\mathbf{d}_{P,k} \leq \mathbf{d}_{M,2}$, then we can relate the total mass of these measures

$$\begin{aligned}\mu_{M,2}([0, 1]) &= \int_0^1 \frac{2}{\mathbf{d}_{M,2}(x)}dx \\ &\leq \frac{2}{k}\int_0^1 \frac{k}{\alpha\mathbf{d}_{P,k}(x)}dx \\ &= \frac{2}{k\alpha}\mu_{p,k}([0, 1])\end{aligned}$$

So, the total mass of the measure $\mu_{M,2}$ is $O(1/k)$ times the total mass of $\mu_{P,k}$. There is nothing special about the line in this example and the same argument holds for other measures induced by densities.

The the total mass of $\mu_{M,2}$ gives an upper bound on the number of points, i.e., $|M| = O(\mu_{M,2}([0, 1]))$. So, the number of points in a kNN sample is $O(1/k)$ times the total mass of $\mu_{P,k}$. This means that if the kNN sample is large, then the kNN density estimate was a bad approximation to the true density. If that is the case, then using kth nearest neighbors may have been a poor choice in the first place.

The moral of this story is that the kNN sample will have size $O(n/k)$ whenever the k-th nearest neighbor density estimate was a good approximation to the underlying measure from which P was sampled.

10 A Python Implementation

The main kNNSample algorithm is described in the prose of the paper. However, the algorithm is sufficiently simple that we can include here the code from the Python implementation.

The cluster graph implementation is straightforward and not shown here. The clusters are iterable and provide `pop` method that returns their farthest point. The cluster graph stores the clusters in a heap, ordered by their radius. It provides a `nbrs_of_nbrs` method that allows one to iterate over all clusters within two hops of a given `cluster`. Because the graph has self loops, this set includes `cluster` and its immediate neighbors. The `knnsample` function is a generator, so it `yield`s points as they are added to the sample rather than returning the final set.

Several small optimizations appear in the official release of the code that have been removed here to show the essentials of the algorithm. That said, the code below is complete and has been tested.

```
from collections import defaultdict
from greedypermutation.clustergraph import ClusterGraph

def knnsample(M, k, seed = None):
    G = ClusterGraph(M, nbrconstant = 2, moveconstant = 1)
    markedpts = defaultdict(set)

    # Yield the first point.
    yield G.heap.findmax().center

    for i in range(1, len(M)):
        cluster = G.heap.findmax()
        point = cluster.pop()
        G.heap.changepriority(cluster)
        radius = 2 * point.dist(cluster.center)

        nearbypts = {q for nbr in G.nbrs(cluster)
                       for q in nbr
                       if q.dist(point) <= radius
                    }

        nearbymarkedpts = {q for q in markedpts[point]
                             if q.dist(point) <= radius
                           }

        if len(nearbypts) + len(nearbymarkedpts) < k:
            for p in nearbypts:
                markedpts[p].add(point)
        else:
            G.addcluster(point, cluster)
            yield point
```