Fast approximate furthest neighbors with data-dependent candidate selection

Ryan R. Curtin and Andrew B. Gardner

Center for Advanced Machine Learning
Symantec Corporation
Atlanta, GA 30338, USA.
ryan@ratml.org, Andrew_Gardner@symantec.com

Abstract. We present a novel strategy for approximate furthest neighbor search that selects a candidate set using the data distribution. This strategy leads to an algorithm, which we call `DrusillaSelect`, that is able to outperform existing approximate furthest neighbor strategies. Our strategy is motivated by an empirical study of the behavior of the furthest neighbor search problem, which lends intuition for where our algorithm is most useful. We also present a variant of the algorithm that gives an absolute approximation guarantee; under some assumptions, the guaranteed approximation can be achieved in provably less time than brute-force search.. Performance studies indicate that `DrusillaSelect` can achieve comparable levels of approximation to other algorithms while giving up to an order of magnitude speedup. An implementation is available in the **mlpack** machine learning library (found at `http://www.mlpack.org`).

1 Introduction

We concern ourselves with the problem of *furthest neighbor search*, which is the logical opposite of the well-known problem of nearest neighbor search. Instead of finding the nearest neighbor of a query point, our goal is to find the furthest neighbor. This problem has applications in recommender systems, where furthest neighbors can increase the diversity of recommendations [1, 2]. Furthest neighbor search is also a component in some nonlinear dimensionality reduction algorithms [3], complete linkage clustering [4, 5] and other clustering applications [6]. Thus, being able to quickly return furthest neighbors is a significant practical concern for many applications.

However, it is in general not feasible to return exact furthest neighbors from large sets of points. Although this is possible with Voronoi diagrams in 2 or 3 dimensions [7], and with single-tree or dual-tree algorithms in higher dimensions [8], these algorithms tend to have long running times in practice. Therefore, approximate algorithms are often considered acceptable in most applications.

For approximate neighbor search algorithms, hashing strategies are a popular option [9–11]. Typically hashing has been applied to the problem of nearest neighbor search, but recently there has been interest in applying hashing

techniques to furthest neighbor search [12, 13]. In general, these techniques are based on random projections, where random unit vectors are chosen as projection bases. This allows probabilistic error guarantees, but the entirely random approach does not use the structure of the dataset.

In this paper, we first consider the structure of the furthest neighbors problem and then conclude that a data-dependent approach can be used to select a small set of candidate points that work for all query points. This allows us to develop:

- `DrusillaSelect`, an algorithm that selects candidate points based on the data distribution and outperforms other approximate furthest neighbors approaches in practice.
- A modified version of `DrusillaSelect` which satisfies rigorous approximation guarantees, and under some assumptions will provably outperform the brute-force approach at search time. However, it is not likely to be useful in practice.

Our empirical results in Section 7 show that the `DrusillaSelect` algorithm demonstrably outperforms existing solutions for approximate k-furthest-neighbor search.

2 Notation and formal problem description

The problem of furthest neighbor search is easily formalized. Given a set of *reference points* $S_r \in \mathcal{R}^{n \times d}$, a set of *query points* $S_q \in \mathcal{R}^{m \times d}$, and a distance metric $d(\cdot, \cdot)$, the problem is to find, for each query point $p_q \in S_q$,

$$\text{argmax}_{p_r \in S_r} d(p_q, p_r). \tag{1}$$

A trivial way to solve this algorithm is by brute-force: for each query point, loop over all reference points and find the furthest one. But this algorithm takes $O(nm)$ time, and does not scale well to large S_r or S_q. In this paper, we will consider the ϵ-approximate form of the furthest neighbor search problem.

Given a set of *reference points* $S_r \in \mathcal{R}^{n \times d}$, a set of *query points* $S_q \in \mathcal{R}^{m \times d}$, an approximation parameter $\epsilon \geq 0$, and a distance metric $d(\cdot, \cdot)$, the ϵ-approximate furthest neighbor problem is to find a furthest neighbor candidate $\hat{p}_{fn}$ for each query point $p_q \in S_q$ such that

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < 1 + \epsilon \tag{2}$$

where p_{fn} is the true furthest neighbor of p_q in S_r. When $\epsilon = 0$, this reduces to the exact furthest neighbor search problem. This form of approximation is also known as relative-value approximation.

3 Related work

There have been a number of improvements over the naive brute-force search algorithm suggested above. Exact techniques based on Voronoi diagrams can solve the furthest neighbor problem. In 1981, Toussaint and Bhattacharya proposed building a furthest-point Voronoi diagram to solve the furthest neighbors

problem in $O(m \log n)$ time [14]. But in high dimensions, Voronoi diagrams are not useful because of their exponential memory dependence on the dimension.

Another approach to exact furthest neighbor search uses space trees [8]. A tree is built on the reference points S_r, and nodes that cannot contain the furthest neighbor of a given query point are pruned. This is essentially equivalent to many algorithms for nearest neighbor search, such as the algorithm for nearest neighbor search with cover trees [15], but with inequalities reversed (i.e., prune nearby nodes, not faraway nodes). This can be done in a dual-tree setting, by also building a tree on the query points S_q. Dual-tree nearest neighbor search has been proven to scale linearly in the size of the reference set under some conditions [16], but no similar bound has been shown for dual-tree furthest neighbor search. It would be reasonable to expect similar empirical scaling. Unfortunately, tree-based approaches tend to perform poorly in high dimensions, and the tree construction time can cause the algorithm to be undesirably slow.

Further runtime acceleration can be achieved if approximation is allowed. It is easy to modify the single-tree and dual-tree algorithms to support this, in the manner suggested by Curtin for nearest neighbor search [17]. Although this is shown to accelerate nearest neighbor search runtime by a significant amount (depending on the allowed approximation), the setup time of building the trees can still dominate. A similar approach to this strategy is the fair split tree, designed by Bespamyatnikh [18]. But this approach suffers from the same issues.

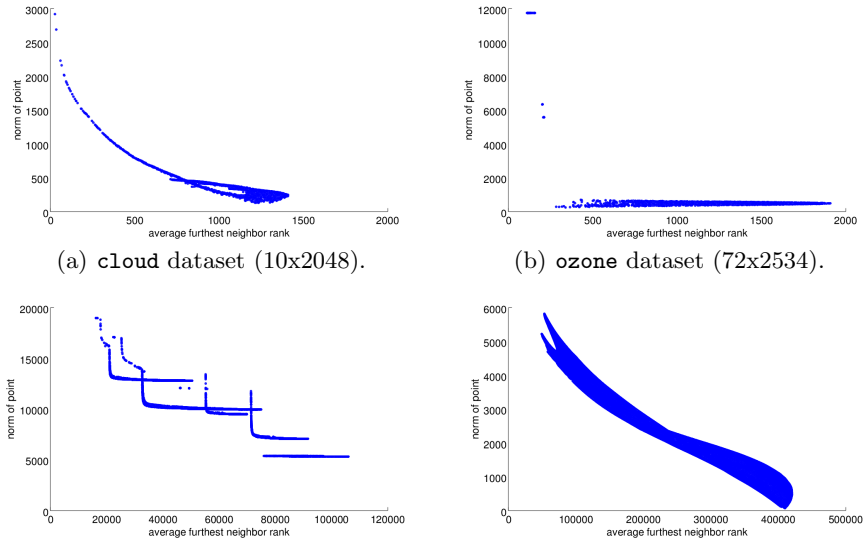
The fastest known algorithms for approximate furthest neighbor search are hashing algorithms. Indyk [13] proposed a hashing algorithm based on random projections that is able to solve a slightly different problem: this algorithm is able to determine (approximately) whether or not there exists a point in S_r farther away than a given distance. This can be reduced to the approximate furthest neighbor problem we are interested in, but this is complex to implement.

Pagh et al. [12] refine this approach to directly solve the approximate furthest neighbor problem; this improves on the runtime of Indyk's algorithm and is easy to implement. This algorithm, called QDAFN ('query-dependent approximate furthest neighbor'), has a guaranteed success probability. A user must specify the number of projections and the number of points stored for each projection; usually, this number is generally low. But in very high-dimensional settings, the random projections can fail to capture important outlying points. This motivates us to investigate the point distribution as a path towards a better algorithm.

4 Furthest neighbor point distribution

The furthest neighbor problem is quite different from the nearest neighbor problem, which has received significantly more attention [19–22, 9, 8, 17]. This difference is perhaps somewhat counterintuitive, given that the furthest neighbor problem is simply an argmax over S_r, not an argmin. But this change causes the problem to have surprisingly different structure with respect to the results.

As a first observation of the differences between the two problems, consider that for any set S_r, the furthest neighbor of every point can be made to be a single point simply by adding a single point sufficiently far from every other point



(a) `cloud` dataset (10x2048). (b) `ozone` dataset (72x2534).

(c) `phy` dataset (78x150000). (d) `covertype` dataset (55x581012).

Fig. 1. Average rank vs. norm for a handful of datasets. Observe that a large norm is correlated with a low rank.

in S_r. There is no analog to this in the nearest neighbor search problem. Indeed, it is often true that for a furthest neighbor query with many query points, the results may contain the same reference point. This is easily demonstrated.

Define the **rank** of a reference point p_r for some query point p_q as the position of p_r in the ordered list of distances from p_q. That is, if the rank of p_r for some query point p_q is k, then p_r is the k-furthest neighbor from p_q.

We can obtain insight into the behavior of furthest neighbor queries by observing the average rank of points on some example datasets from the UCI dataset repository [23]. Figure 1 contains scatterplots displaying the average rank of a reference point versus the mean-centered norm of the reference point for the all-furthest-neighbors problem (that is, each point in the reference set is used as a query point).

Figure 1 shows that there is a clear and unmistakable correlation between the norm of a point and its average rank for the all-k-furthest-neighbor problem. For the `ozone` dataset, we can see that there are only a few points with high norm, and all of these have much lower average rank than the rest of the points.

This correlation is related to the phenomenon of *hubness* in the nearest neighbor search literature [24]; specifically, points with low average rank may be seen to be related to *anti-hubs* and distance-based outliers. In higher dimensions, more anti-hubs may be expected [25]—thus we may conclude that high-norm points (which have low average rank and are related to anti-hubs) are increasingly important in high-dimensional settings. Therefore, an effective furthest neighbors algorithm for high-dimensional data should take this structure into account: *high-norm points are more important than low-norm points.*

5 The algorithm: `DrusillaSelect`

Our collective observations motivate an algorithm for approximate furthest neighbor search, which we introduce as `DrusillaSelect` in Algorithm 1. The algorithm constructs a small collection of points by repeatedly choosing projection bases from the data points with largest norm.[1] Then, the other points in the dataset are projected onto the basis and are selected if they are good candidates. After this collection is built, each query point is simply compared with all points in the collection in order to determine a good furthest neighbor candidate.

`DrusillaSelect` depends on two parameters: l, the number of projections, and m, the number of points taken for each projection. Empirically we observe that values in the range of $l \in [2, 15]$ and $m \in [1, 5]$ produce acceptably good approximations for most datasets, with approximation levels between $\epsilon = 0.01$ and $\epsilon = 1.1$.

[1] This is where the algorithm gets its name; the first author's cat displays the same behavior when selecting a food bowl to eat from.

Algorithm 1 `DrusillaSelect`: fast approximate k-furthest neighbor search.

Input: reference set S_r, query set S_q, number of neighbors k, number of projections l, set size m
Output: array of furthest neighbors $N[]$

{Pre-processing: mean-center data.}
$m \leftarrow \frac{1}{n} \sum_{p_r \in S_r} p_r$
$S_r \leftarrow S_r - m$; $S_q \leftarrow S_q - m$

{Pre-processing: build DrusillaSelect sets.}
for all $p_r \in S_r$ **do** $n[p_r] \leftarrow \|p_r\|$ {Initialize norms of points.}
for all $i \in \{0, 1, \ldots, l\}$ **do**
  $p_i \leftarrow \text{argmax}_{p_r \in S_r}\, n[p_r]$ {Take next point with largest norm.}
  $v_i \leftarrow p_i / \|p_i\|$

  {Calculate distortions and offsets.}
  for all $p_r \in S_r$ such that $n[p_r] \neq 0$ **do**
    $O[p_r] \leftarrow p_r^T v_i$
    $D[p_r] \leftarrow \|p_r - O[p_r] v_i\|$
    $s[p_r] \leftarrow |O[p_r]| - D[p_r]$

  {Collect points that are well-represented by p_i.}
  $R_i \leftarrow$ points corresponding to largest m elements of $s[\cdot]$
  for all $p_r \in R_i$ **do** $n[p_r] = 0$ {Mark point as used.}
  for all $p_r \in S_r$ such that $\text{atan}(D[p_r]/O[p_r]) \geq \pi/8$ **do**
    $n[p_r] = 0$ {Mark point as used.}

{Search for furthest neighbors.}
for all $p_q \in S_q$ **do**
  for all $R_i \in R$ **do**
    for all $p_r \in R_i$ **do**
      if $d(p_q, p_r) > N_k[p_q]$ **then**
        update results $N[p_q]$ for p_q with p_r

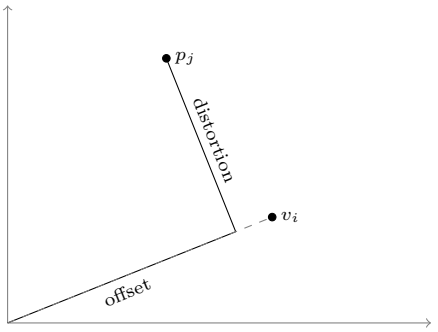


Fig. 2. Distortion and offset for p_j with base vector v_i.

The primary intuition of the algorithm is that we want to collect points in the sets R_i that are likely to be furthest neighbors of any query point. We know from our earlier experiments that points with high mean-centered norms are likely to be good furthest neighbor candidates. Thus, we start by selecting the highest-norm mean-centered point p_i as the primary point of the set R_i, and collect m points that are not too distorted by a projection onto the unit vector v_i which points in the direction of p_i. Any points that are not too distorted by this projection but not collected are ignored for future projections (line 18). In addition, points that lie within a cone pointing in the direction of v_i are also ignored (line 20). The value of $\pi/8$ was chosen for its decent empirical performance, but it would be reasonable to select different values.

The words "not too distorted" deserve some elaboration: we wish to find high-norm points that are well-represented by p_i, but we do not wish to find high-norm points that are *not* well-represented by p_i. Ideally, those points will be selected as the primary point of another set R_j. Therefore, for each point p_j, we calculate the offset $O[p_j]$; this is the norm of the projection of p_j onto v_i. Similarly, we calculate the distortion $D[p_j]$. Figure 2 displays a simple example of offset and distortion.

Our goal is to balance two objectives in selecting points for R_i:

- Select high-norm points.
- Select points that are well-represented by v_i.

The solution we have used here is to construct a score $s[p_j]$ which is just the distortion subtracted from the offset (see line 15). Figure 3 displays an example v_i with 20 points; each point is indexed by its position in the ordered score set $s[\cdot]$. In the context of `DrusillaSelect`, if we took $m = 6$ (so, 6 points were selected for each v_i), then v_i and the five red points p_1 through p_5 would be selected to make up the set R_i. Then, p_7 would be chosen as v_{i+1} because it is the point with largest norm that has not been selected (line 9).

Once we have constructed the sets R_i, then our actual search is a simple brute-force search over every point contained in each set R_i. Because the total number of points in R is only lm, brute-force scan is sufficient.

`DrusillaSelect` has a somewhat similar structure to the QDAFN algorithm [12]; except for three important differences: *(i)* the vectors v_i are drawn using

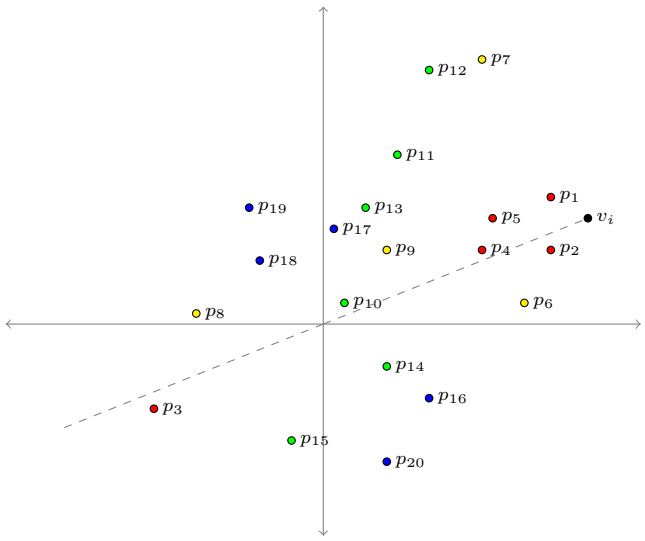


Fig. 3. Example scores for a set of points; red: highest scores, blue: lowest scores.

properties of the reference set, *(ii)* there is no priority queue structure when scanning the sets, and *(iii)* the projection bases chosen cannot be too similar. Although `DrusillaSelect` can involve more setup time, our empirical simulations show it is able to provide better results with fewer sets and points in each sets, resulting in better overall performance for a given level of approximation.

Table 1 gives a comparison of the runtimes of different approximate furthest neighbor algorithms. Note that `DrusillaSelect` and QDAFN have the same asymptotic setup time for the same l and m; but in practice, the overhead of `DrusillaSelect` setup time is higher than QDAFN for equivalent l and m. But again it must be noted that to provide the same results accuracy, l and m may generally be set smaller with `DrusillaSelect` than QDAFN.

Algorithm	Setup time	Search time
`DrusillaHash`	$O(ld\|S_r\|\log\|S_r\|)$	$O(\|S_q\|dlm)$
QDAFN [12]	$O(ld\|S_r\|\log\|S_r\|)$	$O(\|S_q\|d(l\log l + m\log l))$
Indyk [13]	$O(ld\|S_r\|\log\|S_r\|)$	$O(l\|S_q\|(d+\log\|S_r\|)\log d\log\log d$
Brute-force	none	$O(\|S_q\|\|S_r\|)$

Table 1. Runtimes of approximate furthest neighbor algorithms.

6 Guaranteed approximation

Next, we wish to consider the problem of an absolute approximation guarantee: in what situations can we ensure that the furthest neighbor returned is an ϵ-approximate furthest neighbor?

It turns out that this is possible with a modification of `DrusillaSelect`, given in Algorithm 2 as `GuaranteedDrusillaSelect`. This algorithm, instead of taking a number of projections l, takes an acceptable approximation level ϵ. The algorithm uses a utility quantity, $\delta = \epsilon/(6+3\epsilon)$.

The algorithm is roughly the same as `DrusillaSelect`, except for that more sets are added until all points with norm greater than $\delta \max_{p_r \in S_r} \|p_r\|$ are con-

Algorithm 2 `GuaranteedDrusillaSelect`: guaranteed approximate k-furthest neighbor search.

Input: reference set S_r, query set S_q, number of neighbors k, acceptable approximation level ϵ, set size m
Output: array of furthest neighbors $N[]$

{Pre-processing: mean-center data.}
$m \leftarrow \frac{1}{n} \sum_{p_r \in S_r} p_r$; $S_r \leftarrow S_r - m$; $S_q \leftarrow S_q - m$

{Pre-processing: build GuaranteedDrusillaSelect sets.}
for all $p_r \in S_r$ **do** $n[p_r] \leftarrow \|p_r\|$ {Initialize norms of points.}
$\delta \leftarrow \frac{\epsilon}{6+3\epsilon}$
while $\max_{p_r \in S_r} n[p_r] > \delta \max_{p_r \in S_r} \|p_r\|$ **do**
  $p_i \leftarrow \text{argmax}_{p_r \in S_r} n[p_r]$ {Take next point with largest norm.}
  $v_i \leftarrow p_i / \|p_i\|$

  {Calculate distortions and offsets.}
  for all $p_r \in S_r$ such that $n[p_r] \neq 0$ **do**
    $O[p_r] \leftarrow p_r^T v_i$
    $D[p_r] \leftarrow \|p_r - O[p_r] v_i\|$
    $s[p_r] \leftarrow |O[p_r]| - D[p_r]$

  {Collect points that are well-represented by p_i.}
  $R_i \leftarrow$ points corresponding to largest m elements of $s[\cdot]$
  for all $p_r \in R_i$ **do** $n[p_r] = 0$ {Mark point as used.}

{Set shrug point (if we can).}
$p_{sh} \leftarrow \emptyset$
if there is any point such that $n[p_r] \neq 0$ **then**
  $p_{sh} \leftarrow$ some point such that $n[p_r] \neq 0$

{Search for furthest neighbors.}
for all $p_q \in S_q$ **do**
  for all $R_i \in R$ **do**
    for all $p_r \in R_i$ **do**
      if $d(p_q, p_r) > N_k[p_q]$ **then**
        update results $N[p_q]$ for p_q with p_r
  if $p_{sh} \neq \emptyset$ and $d(p_q, p_{sh}) > N_k[p_q]$ **then**
    update results $N[p_q]$ for p_q with p_{sh}

tained in some set R_i, and an extra point called the *shrug point* is held. The shrug point is set to be any point within the small zero-centered ball of radius $\delta \max_{p_r \in S_r} \|p_r\|$. This is needed to catch situations where p_q is close to every point in some R_i, and serves to provide a "good enough" result to satisfy the approximation guarantee.

Because `GuaranteedDrusillaSelect` collects potentially huge numbers of sets that may contain most of the points in S_r, the algorithm is primarily of theoretical interest. Although the algorithm will outperform brute-force search as long as the sets do not contain nearly all of the points in S_r, it is not likely to be practical for large S_r.

Now we may present our theoretical result. First, we need a utility lemma.

Lemma 1. *Given a mean-centered set S_r and a query point p_q with true furthest neighbor p_{fn}, if $\|p_q\| \leq \frac{1}{3}\max_{p_r \in S_r}\|p_r\|$, then $\|p_{fn}\| \geq \frac{1}{3}\max_{p_r \in S_r}\|p_r\|$.*

Proof. This is a simple proof by contradiction: suppose $\|p_{fn}\| < \frac{1}{3}\max_{p_r \in S_r}\|p_r\|$. Then, the maximum possible distance between p_q and p_{fn} is bounded above as $d(p_q, p_{fn}) < \frac{2}{3}\max_{p_r \in S_r}\|p_r\|$. But the minimum possible distance between p_q and the largest point in S_r is bounded below as

$$d(p_q, \operatorname*{argmax}_{p_r \in S_r}\|p_r\|) \geq \max_{p_r \in S_r}\|p_r\| - \frac{1}{3}\max_{p_r \in S_r}\|p_r\| = \frac{2}{3}\max_{p_r \in S_r}\|p_r\|. \tag{3}$$

This means that the largest point in S_r is a further neighbor than p_{fn}, which is a contradiction. □

We may now prove the main result.

Theorem 1 *Given a set S_r and an approximation parameter $\epsilon < 1$ and any set size $m > 0$,* `GuaranteedDrusillaSelect` *will return, for each query point p_q, a furthest neighbor $\hat{p}_{fn}$ such that*

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < 1 + \epsilon \tag{4}$$

where p_{fn} is the true furthest neighbor of p_q in S_r. That is, $\hat{p}_{fn}$ is an ϵ-approximate furthest neighbor of p_q.

Proof. We know from Lemma 1 that if the norm of p_q is less than or equal to 1/3 of the maximum norm of any point in S_r, then the true furthest neighbor must have norm greater than or equal to 1/3 of the maximum norm of any point in S_r. Since δ is always less than 1/3 in Algorithm 2, we know that any such point will be contained in some set R_i, and thus the algorithm will return the exact furthest neighbor in this case.

The only other case to consider, then, is when the norm of the query point is large: $\|p_q\| > \frac{1}{3}\max_{p_r \in S_r}\|p_r\|$. But we already know due to the way the algorithm works, that if $\|p_{fn}\| \geq \delta \max_{p_r \in S_r}\|p_r\|$, then p_{fn} will be contained in some set R_i and the algorithm will return p_{fn}, satisfying the approximation guarantee.

But what about when $\|p_{fn}\|$ is smaller? We must consider the case where $\|p_{fn}\| < \delta \max_{p_r \in S_r}\|p_r\|$. Here we may place an upper bound on the distance between the query point and its furthest neighbor:

$$d(p_q, p_{fn}) \leq \|p_q\| + \|p_{fn}\| < \|p_q\| + \delta \max_{p_r \in S_r}\|p_r\|. \tag{5}$$

We may also place a lower bound on the distance between the query point and its returned furthest neighbor using the shrug point p_{sh}. The distance between p_q and p_{sh} is easily lower bounded: $d(p_q, p_{sh}) \geq \|p_q\| - \delta \max_{p_r \in S_r}\|p_r\| > 0$. This is also a lower bound on $d(p_q, \hat{p}_{fn})$. We may combine these bounds:

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < \frac{\|p_q\| + \delta \max_{p_r \in S_r}\|p_r\|}{\|p_q\| - \delta \max_{p_r \in S_r}\|p_r\|}. \tag{6}$$

Now, define the convenience quantity α as

$$\alpha = \frac{\max_{p_r \in S_r} \|p_r\|}{\|p_q\|}. \tag{7}$$

Because of our assumptions on p_q, we know that $\alpha < 3$. Using these inequalities, we may further simplify Equation 6.

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < \frac{1 + \delta\alpha}{1 - \delta\alpha} \tag{8}$$

$$= 1 + \frac{2\delta\alpha}{1 - \delta\alpha} \tag{9}$$

$$< 1 + \frac{6\delta}{1 - 3\delta} \tag{10}$$

and because $\delta = \frac{\epsilon}{6+3\epsilon}$, Equation 10 simplifies to the result,

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < 1 + \epsilon \tag{11}$$

and therefore the theorem holds. □

Note that the theorem holds if we set δ to the simpler quantity of $\epsilon/9$; but the quantity $(\epsilon/(6 + 3\epsilon))$ provides a tighter bound.

Although `GuaranteedDrusillaSelect` does not guarantee better search time than brute force under all conditions, it does in most conditions. As one example, consider a large dataset where the norms of points in the centered dataset are uniformly distributed. Some of these points will have norm less than $(\epsilon/15) \max_{p_r \in S_r} \|p_r\|$. These points (except the shrug point p_{sh}) will not be considered by the `GuaranteedDrusillaSelect` algorithm, and this means that the `GuaranteedDrusillaSelect` algorithm will inspect fewer points at search time than the brute-force algorithm.

Next, consider the extreme case, where there exists one outlier p_o with extremely large norm, such that the next largest point has norm smaller than $(\epsilon/(6 + 3\epsilon))\|p_o\|$. Here, `GuaranteedDrusillaSelect` with $m = 1$ will only need to inspect two points: the extreme outlier, and the shrug point p_{sh}.

On the other hand, there do exist cases where `GuaranteedDrusillaSelect` gives no improvement over brute-force search, and every point must be inspected. If the dataset is such that all points have norm greater than $(\epsilon/(6 + 3\epsilon)) \max_{p_r \in S_r} \|p_r\|$, then the sets R_i will contain every single point in the dataset.

These theoretical results show that it is possible to give a guaranteed ϵ-approximate furthest neighbor in less time than brute-force search, if the distribution of norms of S_r are not worst-case. But due to the algorithm's storage requirement, it is not likely to perform well in practice and so we do not investigate its empirical performance.

Dataset	n	d	QDAFN params		`DrusillaSelect` params	
			l	m	l	m
cloud	2048	10	30	60	2	1
isolet	7797	617	40	40	2	1
gisette	12500	5000	40	40	2	2
corel	37749	32	5	5	2	1
p53	48192	5409	25	25	3	2
randu	100000	10	15	15	5	2
miniboone	130064	50	125	200	2	1
phy	150000	78	12	12	4	1
covertype	581012	55	15	20	6	2
pokerhand	1000000	10	15	50	50	8
susy	5000000	18	18	18	2	2
higgs	11000000	28	32	32	2	2

Table 2. Datasets and parameters.

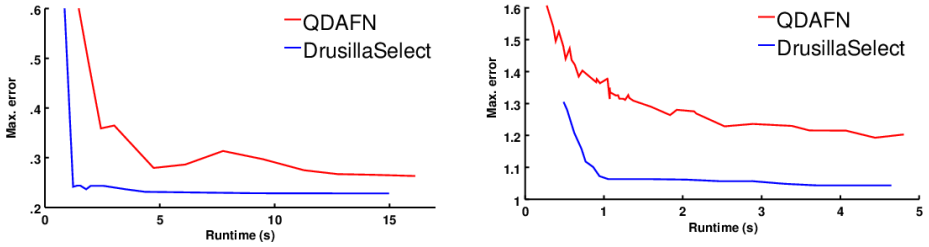
Dataset	brute-force	dual-tree	QDAFN	`DrusillaSelect`
cloud	0.039s	0.040s	0.011s	**0.001s**
isolet	6.754s	7.706s	0.165s	**0.041s**
gisette	141.923s	141.963s	1.875s	**0.549s**
corel	10.292s	1.030s	0.021s	**0.021s**
p53	2258.331s	270.341s	3.475s	**2.734s**
randu	42.392s	28.004s	0.316s	**0.0619s**
miniboone	187.262s	4.105s	2.165s	**0.104s**
phy	370.061s	58.720s	0.203s	**0.189s**
covertype	4077.922s	144.993s	1.244s	**0.203s**
randu	–	16.715s	0.069s	**0.043s**
pokerhand	–	852.001s	11.749s	**8.035s**
susy	–	88.295s	21.678s	**2.4467s**
higgs	–	425.053s	56.094s	**12.694s**

Table 3. Runtimes for $\epsilon = 0.05$-approximate furthest neighbor search.

7 Experiments

Next, we investigate the empirical performance of the `DrusillaSelect` algorithm, comparing with brute-force search, QDAFN [12], and dual-tree exact furthest neighbor search as described by Curtin et al. [8]. Note that both brute-force search and the dual-tree algorithm return exact furthest neighbors; QDAFN and `DrusillaSelect` return approximations. Each implementation is either from **mlpack** [26] or is built using **mlpack**. We test the algorithms on a variety of datasets from the UCI dataset repository and `randu`, which is uniformly randomly distributed. These datasets and their properties are given in Table 2.

First, we compare runtimes across all four algorithms. The approximate algorithms are tuned to return, on average across the query set, $\epsilon = 0.05$-approximate furthest neighbors (using the parameters from Table 2). Table 3 shows the average runtimes of each of the four algorithms on each dataset across ten trials with the dataset randomly split into 30% query set, 70% reference set. I/O times are not included; the runtime only includes the time for the search itself, including preprocessing time (building hash tables, sets, or trees).



(a) `covertype` dataset. (b) `gisette` dataset.

Fig. 4. Maximum error for QDAFN and `DrusillaSelect` as a function of runtime.

The `DrusillaSelect` algorithm provides average $\epsilon = 0.05$-approximate furthest neighbors up to an order of magnitude faster than any other competing algorithm, and it also needs to inspect fewer points to return an accurate approximate furthest neighbor (with the exception of the `pokerhand` dataset). In many cases, `DrusillaSelect` only needs to inspect fewer than 10 points to find good furthest neighbor approximations, whereas QDAFN must inspect 50 or more.

Our datasets have two extreme examples: the `miniboone` dataset, where the data lies on a low-dimensional manifold, and the `randu` dataset.

For the `miniboone` dataset, `DrusillaSelect` is able to easily recover only four points that provide average 1.05-approximate furthest neighbors. But because QDAFN chooses random projection bases, it takes very many to have a high probability of recovering good furthest neighbors. In our experiments, we were not able to achieve good approximation reliably until using as many as 125 projection bases. This effect was also observed with the `covertype` dataset.

`DrusillaSelect` also outperforms other approaches on the `randu` dataset, despite there being no structure for `DrusillaSelect` to exploit. But the algorithm is still able to outperform others; this is because the algorithm specifically ensures that projection bases are not too similar (see lines 18–20).

Another important property of `DrusillaSelect` is that it gives a small maximum error compared to QDAFN. Figure 4 shows the maximum error of each approach as the number of points scanned increase on the `covertype` dataset. For QDAFN, we have swept with $l = m$ from $l = 20$ to $l = 250$, and for `DrusillaSelect`, we have set $m = l/3$ and swept l from 6 to 60.

Our experimental results have shown that `DrusillaSelect` gives excellent approximation while only needing to scan few points. Whereas QDAFN seems to perform poorly in high-dimensional settings where the data lie on a low-dimensional manifold (because projection bases are random), `DrusillaSelect` effectively captures the low-dimensional structure with few projection bases.

8 Conclusion

We have proposed an algorithm, `DrusillaSelect`, that builds a candidate set for approximate furthest neighbor search by using the properties of the dataset. This algorithm design is motivated by our empirical analysis of the structure of the approximate furthest neighbor search problem, and the algorithm performs quite compellingly in practice. It scales better with dataset size than other techniques.

We have also proposed a variant, `GuaranteedDrusillaSelect`, which is able to give an absolute approximation guarantee. Under some assumptions, this algorithm will provably outperform the brute-force approach at search time. This is a benefit that no other furthest neighbor search scheme is able to provide. However, this variant is not likely to be useful in practice due to the large number of points it must search to satisfy the guarantee.

Interesting future directions for this line of research may include combining a random projection approach with the approach outlined here. It would also be possible to generalize our approach to arbitrary distance metrics, including those where the points lie in an unrepresentable space. This could be done using techniques similar to some that have been used for max-kernel search [27, 28]. Lastly, we have focused on high-norm points as 'important'; but a study connecting hubness (or anti-hubness) to the average furthest-neighbor rank would be enlightening and may potentially guide future improvements to this approach.

References

1. A. Said, B. Kille, B.J. Jain, and S. Albayrak. Increasing diversity through furthest neighbor-based recommendation. *Proceedings of the Fifth International Conference on Web Search and Data Mining (WSDM 2012)*, 12, 2012.
2. A. Said, B. Fields, B.J. Jain, and S. Albayrak. User-centric evaluation of a k-furthest neighbor collaborative filtering recommender algorithm. In *Proceedings of the 2013 conference on Computer Supported Cooperative Work*, pages 1399–1408. ACM, 2013.
3. N. Vasiloglou, A.G. Gray, and D.V. Anderson. Scalable semidefinite manifold learning. In *Proceedings of the 2008 IEEE Workshop on Machine Learning for Signal Processing, 2008 (MLSP 2008)*, pages 368–373. IEEE, 2008.
4. D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.
5. P.D. Schloss, S.L. Westcott, T. Ryabin, J.R. Hall, M. Hartmann, E.B. Hollister, R.A. Lesniewski, B.B. Oakley, D.H. Parks, C.J. Robinson, J.W. Sahl, B. Stres, G.G. Thallinger, D.J. Van Horn, and C.F. Weber. Introducing mothur: open-source, platform-independent, community-supported software for describing and comparing microbial communities. *Applied and Environmental Microbiology*, 75(23):7537–7541, 2009.
6. C.J. Veenman, M.J.T. Reinders, and E. Backer. A maximum variance cluster algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(9):1273–1280, 2002.
7. O. Cheong, C.-S. Shin, and A. Vigneron. Computing farthest neighbors on a convex polytope. *Theoretical Computer Science*, 296(1):47–58, 2003.
8. R.R. Curtin, W.B. March, P. Ram, D.V. Anderson, A.G. Gray, and C.L. Isbell Jr. Tree-independent dual-tree algorithms. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, 2013.
9. M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry (SoCG '04)*, pages 253–262. ACM, 2004.
10. P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98)*, pages 604–613. ACM, 1998.

11. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS '06)*, pages 459–468. IEEE, 2006.
12. R. Pagh, F. Silvestri, J. Sivertsen, and M. Skala. Approximate furthest neighbor in high dimensions. In *Similarity Search and Applications*, pages 3–14. Springer, 2015.
13. P. Indyk. Better algorithms for high-dimensional proximity problems via asymmetric embeddings. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 539–545. Society for Industrial and Applied Mathematics, 2003.
14. G.T. Toussaint and B.K. Bhattacharya. On geometric algorithms that use the furthest-point voronoi diagram. *School of Computer Science, McGill University, Tech. Rept. No. 81.3*, 1981.
15. A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 97–104. ACM, 2006.
16. R.R. Curtin, D. Lee, W.B. March, and P. Ram. Plug-and-play dual-tree algorithm runtime analysis. *Journal of Machine Learning Research*, 16:3269–3297, 2015.
17. R.R. Curtin. Faster dual-tree traversal for nearest neighbor search. In *Similarity Search and Applications*, pages 77–89. Springer, 2015.
18. S. Bespamyatnikh. Dynamic algorithms for approximate neighbor searching. In *Proceedings of the 8th Canadian Conference on Computational Geometry (CCCG'96)*, pages 252–257, 1996.
19. J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
20. S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.
21. A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *Proceedings of the Twenty-Fifth International Conference on Very Large Data Bases (VLDB '99)*, volume 99, pages 518–529, 1999.
22. A.G. Gray and A.W. Moore. 'N-Body' problems in statistical learning. In *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, volume 4, pages 521–527, 2001.
23. M. Lichman. UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml`, University of California Irvine, School of Information and Computer Sciences, 2013.
24. M. Radovanoić, A. Nanopoulos, and C. Ivanović. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research*, 11(Sep):2487–2531, 2010.
25. N. Tomasev, M. Radovanović, D. Mladenic, and M. Ivanović. The role of hubness in clustering high-dimensional data. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):739–751, 2014.
26. R.R. Curtin, J.R. Cline, N.P. Slagle, W.B. March, P. Ram, N.A. Mehta, and A.G. Gray. mlpack: A scalable C++ machine learning library. *The Journal of Machine Learning Research*, 14(1):801–805, 2013.
27. R.R. Curtin, P. Ram, and A.G. Gray. Fast exact max-kernel search. In *Proceedings of the 2013 SIAM International Conference on Data Mining (SDM '13)*, pages 1–9. SIAM, 2013.
28. R.R. Curtin and P. Ram. Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining*, 7(4):229–253, 2014.