Exploiting the structure of furthest neighbor search for fast approximate results

Ryan R. Curtin, Javier Echauz, Andrew B. Gardner

PII: S0306-4379(17)30155-2
DOI: 10.1016/j.is.2017.12.010
Reference: IS 1275

To appear in: *Information Systems*

Received date: 14 March 2017
Revised date: 10 September 2017
Accepted date: 30 December 2017

Please cite this article as: Ryan R. Curtin, Javier Echauz, Andrew B. Gardner, Exploiting the structure of furthest neighbor search for fast approximate results, *Information Systems* (2017), doi: 10.1016/j.is.2017.12.010

Exploiting the structure of furthest neighbor search for fast approximate results

Ryan R. Curtin, Javier Echauz, Andrew B. Gardner

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

Abstract

We present a novel strategy for approximate furthest neighbor search that selects a set of candidate points 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 a study of the behavior of the furthest neighbor search problem, which has significantly different structure than the nearest neighbor search problem, and can be understood with the help of an information-theoretic hardness measure that we introduce. 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, even on the hardest datasets, 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`).

Keywords: furthest neighbor search, farthest neighbor search, hubness and anti-hubness

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, 10, 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, which is an extended version of a previous work [14], we first consider the structure and hardness 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:

- An information-theoretic entropy measure of the hardness of the furthest neighbor search problem.
- `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 8 show that the `DrusillaSelect` algorithm demonstrably outperforms existing solutions for approximate k-furthest-neighbor search, on both easy and hard datasets (according to our information-theoretic entropy measure).

In Section 2, we formally introduce the problem of furthest neighbor search; Section 3 considers related work and other existing algorithms to solve the problem. Section 4 studies the structure of the furthest neighbor search problem, leading us to define a hardness measure in Section 5. This allows us to introduce `DrusillaSelect` in Section 6, and its guaranteed approximation variant `GuaranteedDrusillaSelect` in Section 7. Section 8 details our empirical experiments, with conclusions in Section 9.

2. Notation and formal problem description

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

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

Note that the query set S_q may actually be only a single point (i.e. $|S_q| = 1$). Our notation here considers the batch furthest neighbor search case ($|S_q| > 1$) for completeness, since some algorithms we compare against in our evaluation are built for batch query sets [8].

In this paper, we will consider the ϵ-approximate form of the furthest neighbor search problem, which is often used because of the difficulty of solving the exact problem in reasonable time. This may be defined as a relaxation of the original furthest neighbor search problem:

Given a set of reference points $S_r \subseteq \mathbb{R}^d$ with $|S_r| = n$, a set of query points $S_q \subseteq \mathbb{R}^d$ with $|S_q| = m$, 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.

The brute force algorithm is the simplest possible algorithm to use for exact furthest neighbor search. In this algorithm, we simply loop over all reference points for each query point. With this, approximation is not possible, since every possible pair of points will be compared. There is no preprocessing in this algorithm, and the running time is $O(mn)$. This can be prohibitively expensive to compute for large datasets; hence, faster algorithms have been developed.

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 [15]. 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 [16], 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 [17], 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 [18]. Although this is shown to accelerate nearest neighbor search runtime by a significant amount (depending on the 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 [19]. 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 distribution of furthest neighbors in order to devise a better algorithm.

4. Furthest neighbor point distribution

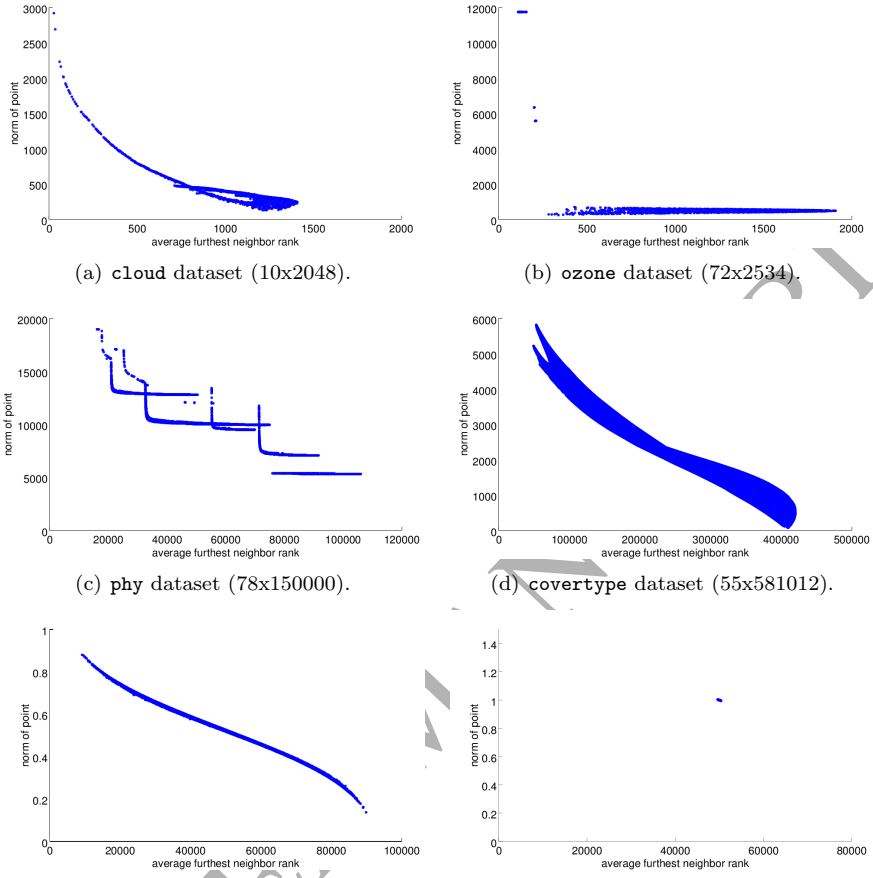
The furthest neighbor problem is quite different from the nearest neighbor problem, which has received significantly more attention [20, 21, 22, 23, 9, 8, 18]. This difference is perhaps counterintuitive, given that the furthest neighbor problem is simply an argmax over S_r, not an argmin. But this change 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 in a Euclidean space[1], the furthest neighbor of every point in S_r can be made to be a single point simply by adding a single point anywhere in the space that is sufficiently far from every other point 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 large S_q, the results may contain the same reference point for many query points. This is easily demonstrated.

Define the **rank** of a reference point p_r for a query point p_q as the position of p_r in the descending ordered list of distances from p_q. That is, if the rank of p_r for a 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

[1]Similar observations also apply in other spaces, but for simplicity of discussion we only consider the Euclidean space in our discussion right here.



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

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

(e) `randu` dataset (10x100000). (f) `ball` dataset (10x100000).

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

dataset repository [24] as well as two others: `randu`, which is 10-dimensional uniformly distributed random data, and `ball`, which is data uniformly distributed on the surface of a hypersphere in 10 dimensions. Figure 1 contains scatterplots displaying the mean-centered norm of the reference point versus the average rank of a 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. Even for the uniformly random distributed dataset (`randu`) this phenomenon is observed; however, for points distributed on the surface of a unit ball (`ball`) this is not the since the norm for all points is the same.

This correlation is related to the phenomenon of *hubness* in the nearest neighbor search literature [25]; 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 [26]—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 likely candidates for furthest neighbors than low-norm points.*

5. Hardness of furthest neighbor search

The observations about the average rank of furthest neighbor candidates has some implications about the difficulty of the furthest neighbor search task. In Figure 1(b), very good approximate furthest neighbor results could be obtained simply by taking only the small set of high-norm outliers as candidates to search with. But in Figure 1(e), taking only a few high-norm points will not result in very good approximate results; and in Figure 1(f), this is not even possible since all points have identical unit norm. Consider the two following extreme cases:

- **Easy**: all points in S_q lie directly on the surface of the unit ball centered at the origin. All points in S_r also lie within the unit ball, except for one extreme outlier that lies a distance of 10 from the origin. *The furthest neighbor of all points in S_q will be the extreme outlier.*
- **Hard**: all points in S_q lie directly on the surface of the unit ball (that is, all points have a norm of 1) and are uniformly distributed. S_r has an identical distribution. *The furthest neighbor of each point in S_q will be the point with minimum cosine similarity. This will almost certainly be different for just about every point in S_q.*

So, clearly, the hardness of furthest neighbor search depends strongly on the distribution of S_r. But the hardness also depends on the distribution of S_q:

Lemma 1. *Given some reference set S_r, if a query point p_q is such that $\|p_q\| \geq \eta$ where*

$$\eta = \frac{(2+\epsilon)}{\epsilon} \max_{p_r \in S_r} \|p_r\| \tag{3}$$

then an ϵ-approximate furthest neighbor of p_q may be found simply by taking any point from S_r.

Proof. All points in S_r lie within a ball of radius $\max_{p_r \in S_r} \|p_r\|$. Therefore, the maximum absolute error we could possibly encounter for any query point is twice that radius. Since $\|p_q\| \geq \eta$,

$$\begin{aligned}
\frac{d(p_q, p^*_{fn})}{d(p_q, \hat{p}_{fn})} &\leq \frac{\eta + \max_{p_r \in S_r} \|p_r\|}{\eta - \max_{p_r \in S_r} \|p_r\|} && (4)\\
&= 1 + \frac{2 \max_{p_r \in S_r} \|p_r\|}{\eta - \max_{p_r \in S_r} \|p_r\|} && (5)\\
&= 1 + \frac{2 \max_{p_r \in S_r} \|p_r\|}{((2+\epsilon)/\epsilon - 1) \max_{p_r \in S_r} \|p_r\|} && (6)\\
&= 1 + \epsilon && (7)
\end{aligned}$$

and therefore the statement holds. □

We can see that for p_q far enough away from S_r, the furthest neighbor search problem becomes trivial. In addition, when we use different query sets S_q with the same reference set S_r, the hardness may be very different. To demonstrate this, Figure 2 shows the average rank distribution for the same S_r and for multiple different query sets that have different distributions. The reference set used here is the `randu` dataset (10-dimensional random uniformly distributed with 100k points), and we have taken four different random datasets as the query set, each with 100k points:

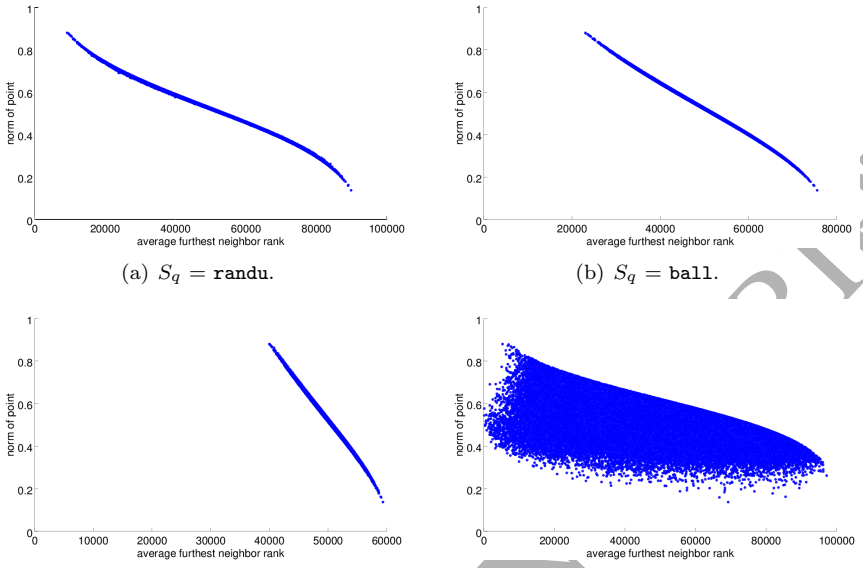
- `randu`: uniform random distribution, each dimension in $[0, 1)$.
- `ball`: uniform distribution on the surface of the unit ball.
- `randn`: Gaussian distribution with identity covariance matrix.
- `randu-2x`: uniform random distribution, each dimension in $[0, 2)$.

The varying levels of hardness of the furthest neighbor search task motivate us to quantify this hardness. In broad strokes, our observations from the previous investigations suggest that the furthest neighbor search problem is easy when the answer is *predictable*—that is, when the furthest neighbor for all query points is only a couple different points from the reference set. On the other hand, the furthest neighbor search problem is hard when the answer is *unpredictable*, or when the furthest neighbor for each query point is typically a different reference point.

This emphasis on predictability and unpredictability naturally points towards information theory. We can use the simple measure of the Shannon entropy of the furthest neighbors over the query set to quantify hardness. First, define the probability of the furthest neighbor as

$$\mathbf{Pr}[p_{fn} = p_r | S_q] = \frac{1}{|S_q|} \sum_{p_q \in S_q} \mathbb{1}(\arg\max_{p'_r \in S_r} d(p_q, p'_r) = p_r) \quad (8)$$

where $\mathbb{1}(\cdot)$ is the indicator function, taking value 1 if the argument is true and 0 otherwise.



(a) S_q = `randu`. (b) S_q = `ball`.

(c) S_q = `randn`. (d) S_q = `randu-2x`.

Figure 2: Average rank vs. norm for S_r = `randu` and different choices of S_q. Observe that the distribution is different for each S_q.

This quantity is simply the empirical probability that p_r is the furthest neighbor of any point in S_q. If all points in S_q have furthest neighbor p_r, then $\mathbf{Pr}[p_{fn} = p_r|S_q] = 1$. Conversely, if no points in S_q have furthest neighbor p_r, then $\mathbf{Pr}[p_{fn} = p_r|S_q] = 0$. This quantity then allows us to calculate a measure of hardness $h(\cdot)$ of the furthest neighbor problem:

$$h(S_q, S_r) = \sum_{p_r \in S_r} -\mathbf{Pr}[p_{fn} = p_r|S_q] \log_2(\mathbf{Pr}[p_{fn} = p_r|S_q]). \tag{9}$$

This notion gives us a quantifiable way to express the difficulty of the furthest neighbor search problem. Unfortunately, it depends on the true furthest neighbors—meaning that we cannot use $h(\cdot)$ to inform our choice of furthest neighbor algorithm *a priori*. However, it is still a useful measure to describe the difficulty of the problem *a posteriori*.

We can see the difference in hardness as a function of both the query set and the reference set. Table 1 shows $h(\cdot)$ values for the datasets in Figures 1 and 2. In addition, the hardness values are stable with respect to splits of the dataset. For each of the datasets in Figures 1, we calculated the hardness value $h(S_q, S_r)$ by using 10-fold cross-validation to split the dataset into a query set S_q and a reference set S_r. The results are shown in Table 2, showing small variance in hardness values with respect to dataset splits.

Next, we wish to determine that the hardness measure increases as anti-hubs

Figure	S_r	S_q	$h(S_q, S_r)$
1(a)	`cloud`	`cloud`	0.128
1(b)	`ozone`	`ozone`	1.602
1(c)	`phy`	`phy`	3.022
1(d)	`covertype`	`covertype`	1.924
1(e)	`randu`	`randu`	9.249
1(f)	`ball`	`ball`	15.769
2(a)	`randu`	`randu`	9.249
2(b)	`randu`	`ball`	11.136
2(c)	`randu`	`randn`	11.470
2(d)	`randu`	`randu-2x`	6.866

Table 1: Hardness values for datasets from Figures 1 and 2.

Dataset	mean $h(S_q, S_r)$	std. dev. $h(S_q, S_r)$
`cloud`	0.124	0.044
`ozone`	1.591	0.081
`phy`	3.000	0.156
`covertype`	1.882	0.063
`randu`	6.620	0.097
`ball`	13.175	0.005

Table 2: Stability of $h(S_q, S_r)$ over 10 cross-validation trials.

are removed from the data. Therefore, for the `cloud` and `ozone` datasets, we plot the measure $h(S_r, S_r)$ as we remove points with largest mean-centered norm from the dataset. The results are shown in Figure 3, showing that as expected, the hardness increases significantly as we remove anti-hubs. Similarly, when we project the points in these datasets onto the unit ball (so that the mean-centered norms of all points are basically identical), the hardness increases to 5.857 and 2.715 for the `cloud` and `ozone` datasets, respectively. This is in line with what is expected of a hardness measure for furthest neighbor search.

For some applications—for instance, diversifying recommendation systems—a dataset with extremely low hardness is not likely to give good results. After all, this situation is likely to occur when there are very few extreme outliers, and recommendations would not be successfully diversified if all the furthest neighbors are the same. Thus, increasing the hardness $h(\cdot, \cdot)$ of a dataset manually by anti-hub removal or compression could be useful for these types of applications.

6. 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 pro-

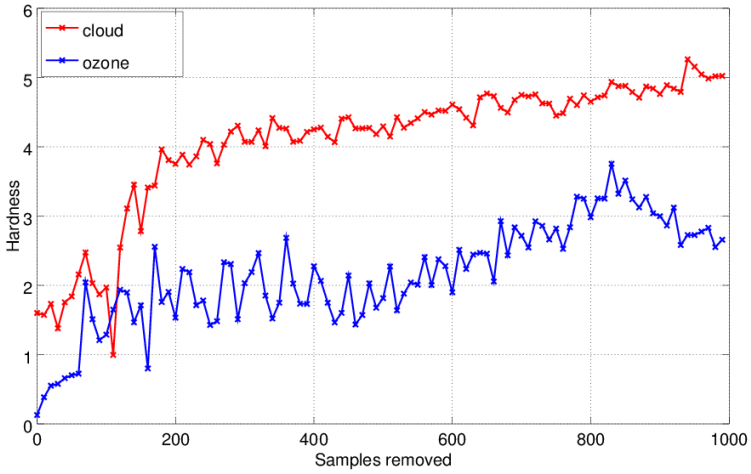


Figure 3: Hardness increases as we remove points with the largest mean-centered norm (i.e. anti-hubs).

jection bases from the data points with largest norm.[2] 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 candidate 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 the 1-furthest-neighbor search problem for most datasets, with approximation levels between $\epsilon = 0.01$ and $\epsilon = 1.1$.

Typically, a good choice for m would be at least the maximum number of furthest neighbor desired. For instance, if k-furthest-neighbors is to be run with $k = 5$, then `DrusillaSelect` will perform best when $m \geq 5$. This is because all k true furthest neighbors of a query point p_q may lie in the same direction; therefore, if $m < k$, then it is possible that not enough candidates will be available from the correct direction for p_q, and the approximation quality may suffer significantly.

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

[2]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.

1: **Input:** reference set S_r, query set S_q, number of neighbors k, number of projections l, set size m
2: **Output:** array of furthest neighbors $N[]$
3: {Pre-processing: mean-center data.}
4: $\mu \leftarrow \frac{1}{n} \sum_{p_r \in S_r} p_r$
5: $S_r \leftarrow S_r - \mu$; $S_q \leftarrow S_q - \mu$
6: {Pre-processing: build DrusillaSelect sets.}
7: **for all** $p_r \in S_r$ **do** $n[p_r] \leftarrow \|p_r\|$ {Initialize norms of points.}
8: **for all** $i \in \{1, \ldots, l\}$u **do**
9: $p_i \leftarrow \text{argmax}_{p_r \in S_r} n[p_r]$ {Take next unused point with largest norm.}
10: $v_i \leftarrow p_i / \|p_i\|$
11: {Calculate distortions and offsets.}
12: **for all** $p_r \in S_r$ such that $n[p_r] \neq 0$ **do**
13: $O[p_r] \leftarrow p_r^T v_i$
14: $D[p_r] \leftarrow \|p_r - O[p_r] v_i\|$
15: $s[p_r] \leftarrow |O[p_r]| - D[p_r]$
16: {Collect points that are well-represented by p_i.}
17: $R_i \leftarrow$ points corresponding to largest m elements of $s[\cdot]$
18: **for all** $p_r \in R_i$ **do** $n[p_r] = 0$ {Mark point as used.}
19: **for all** $p_r \in S_r$ such that $\text{atan}(D[p_r]/|O[p_r]|) \leq \pi/8$ **do**
20: $n[p_r] = 0$ {Mark point as used.[3]}
21: {Search for furthest neighbors.}
22: **for all** $p_q \in S_q$ **do**
23: **for all** $i \in 1, \ldots, l$ **do**
24: **for all** $p_r \in R_i$ **do**
25: **if** $d(p_q, p_r) > N_k[p_q]$ **then**
26: update results $N[p_q]$ for p_q with p_r

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 (because they were not in the top m scores in this projection) are ignored for future projections (line 18).

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 4 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.

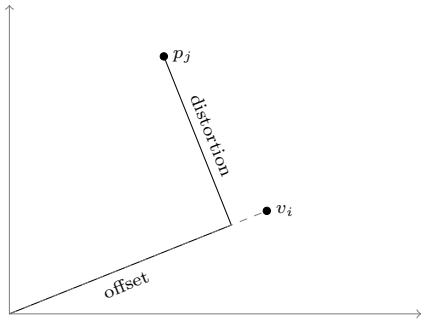


Figure 4: Distortion and offset for p_j with base vector v_i.

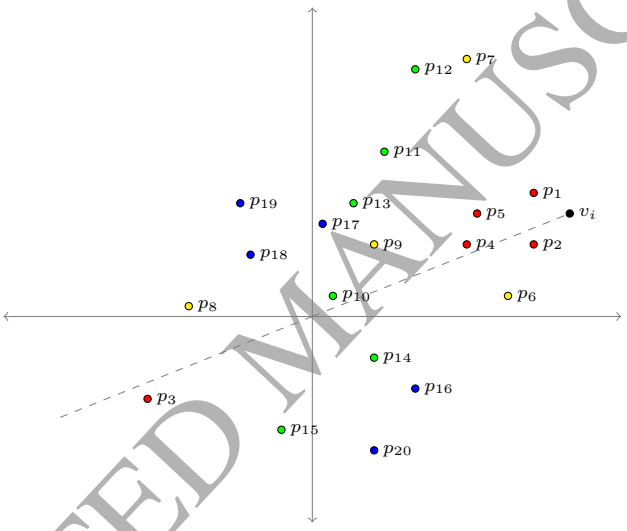


Figure 5: Example scores for a set of points with `jet` colormap representing ordering of scores; red: highest scores, blue: lowest scores.

- 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 5 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 p_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 p_{i+1} because it is the point with largest norm that has not been selected (line 9).

In addition, after we select m points for the projection basis v_i, we choose to ignore points that lie within a cone pointing in the direction of v_i in future iterations (line 20). The value of $\pi/8$ was chosen for its good empirical performance, but it would be reasonable to select different values. This is in order to handle cases where there are many large-norm points in a single direction. Sup-

Algorithm	Setup time	Search time
`DrusillaSelect`	$O(ld\|S_r\| \log \|S_r\|)$	$O(\|S_q\|dlm)$
QDAFN [12]	$O(ld\|S_r\| \log \|S_r\|)$	$O(\|S_q\|d(l \log l + m \log l))$
Indyk [13]	$O(ld\|S_r\| \log \|S_r\|)$	$O(l\|S_q\|(d + \log \|S_r\|) \log d \log \log d)$
Dual-tree [8]	$\sim O(\|S_r\| \log \|S_r\| + \|S_q\| \log \|S_q\|)$	$\sim O(\|S_r\|)$
Single-tree [8]	$\sim O(\|S_r\| \log \|S_r\|)$	$\sim O(\|S_q\| \log \|S_r\|)$
Brute-force	none	$O(\|S_q\|\|S_r\|)$

Table 3: Runtimes of approximate furthest neighbor algorithms. All runtimes are guarantees except the tree-based algorithms, which are expected or typically observed scalings.

pose that more than $2m$ points have high mean-centered norm and lie (roughly) in one direction, v_i. Then, we will end up with at least two projection bases that have almost identical directions close to v_i. This defeats our objective, which is to choose *different* directions that capture a diverse set of points with large mean-centered norm. Hence, any points with direction 'too close' to v_i are ignored in future iterations.

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 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 3 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.

7. 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 contained 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

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}

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 2. *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

$$d(p_q, \underset{p_r \in S_r}{\operatorname{argmax}}\|p_r\|) \geq \max_{p_r \in S_r}\|p_r\| - \frac{1}{3}\max_{p_r \in S_r}\|p_r\| = \frac{2}{3}\max_{p_r \in S_r}\|p_r\|. \tag{10}$$

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*

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

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 2 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:

$$d(p_q, p_{fn}) \leq \|p_q\| + \|p_{fn}\| < \|p_q\| + \delta \max_{p_r \in S_r}\|p_r\|. \tag{12}$$

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:

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < \frac{\|p_q\| + \delta \max_{p_r \in S_r}\|p_r\|}{\|p_q\| - \delta \max_{p_r \in S_r}\|p_r\|}. \tag{13}$$

Now, define the convenience quantity α as

$$\alpha = \frac{\max_{p_r \in S_r} \|p_r\|}{\|p_q\|}. \tag{14}$$

Because of our assumptions on p_q, we know that $\alpha < 3$. Using these inequalities, we may further simplify Equation 13.

$$\begin{aligned} \frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} &< \frac{1+\delta\alpha}{1-\delta\alpha} && (15)\\ &= 1 + \frac{2\delta\alpha}{1-\delta\alpha} && (16)\\ &< 1 + \frac{6\delta}{1-3\delta} && (17) \end{aligned}$$

and because $\delta = \frac{\epsilon}{6+3\epsilon}$, Equation 17 simplifies to the result,

$$\frac{d(p_q, p_{fn})}{d(p_q, \hat{p}_{fn})} < 1 + \epsilon \tag{18}$$

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.

			QDAFN params		DS params	
Dataset	n	d	l	m	l	m
cloud	2048	10	30	60	2	1
nips	5810	11463	18	18	2	2
isolet	7797	617	40	40	2	1
gisette	12.5k	5000	40	40	2	2
ujindoorloc	21.0k	526	50	50	2	1
corel	37.7k	32	5	5	2	1
p53	48.1k	5409	25	25	3	2
mnist	70k	784	44	20	2	1
randu	100k	10	15	15	5	2
ball	100k	10	150	40	50	22
randn	100k	10	30	30	5	2
miniboone	130k	50	125	200	2	1
phy	150k	78	12	12	4	1
covertype	581k	55	15	20	6	2
pokerhand	1M	10	15	50	7	2
susy	5M	18	18	18	2	2
higgs	11M	28	32	32	2	2

Table 4: Datasets and parameters.

8. Experiments

Next, we investigate the empirical performance of the `DrusillaSelect` algorithm, comparing with brute-force search, QDAFN [12], and single-tree 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 available in **mlpack** [27]. We test the algorithms on a variety of datasets from the UCI dataset repository and the previously-described random `randu`, `randn`, and `ball` datasets. These datasets and their properties are given in Table 4.

Running `DrusillaSelect` with **mlpack** is straightforward:

```
mlpack_approx_kfn -q query_set.csv -r ref_set.csv -k $k -a ds
```

Additional options exist to use the QDAFN algorithm and set the number of tables (l) and number of projections (m), and the `mlpack_kfn` program may be used to obtain exact results with the single-tree, dual-tree, or brute-force algorithms.

8.1. Comparison across datasets

First, we compare runtimes across all five 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 4). Table 5 shows the average runtimes of each of the five algorithms on each dataset across

Dataset	$h(S_q, S_r)$	b.f.	s.t.	d.t.	QDAFN	DS
cloud	0.127	0.036s	0.002s	0.003s	0.012s	**0.001s**
nips	0.482	68.056s	224.277s	211.488s	3.052s	**0.454s**
isolet	3.913	6.753s	9.276s	6.989s	0.154s	**0.028s**
gisette	2.278	138.026s	272.846s	158.591s	2.437s	**0.280s**
ujindoorloc	1.381	42.216s	28.557s	18.018s	0.426s	**0.077s**
corel	1.367	9.639s	5.849s	5.264s	0.021s	**0.015s**
p53	1.168	4375.694s	1143.218s	489.037s	3.475s	**2.734s**
mnist	4.760	684.711s	815.647s	576.537s	0.940s	**0.361s**
randu	6.580	38.989s	12.329s	16.912s	0.145s	**0.072s**
ball	14.472	40.078s	86.613s	43.640s	1.210s	**1.025s**
randn	5.804	37.923s	16.347s	20.533s	0.267s	**0.076s**
miniboone	0.001	354.051s	0.689s	1.033s	2.184s	**0.053s**
phy	2.891	370.061s	1.419s	1.250s	0.238s	**0.148s**
covertype	1.891	4077.922s	3.127s	4.165s	3.155s	**0.486s**
pokerhand	6.722	–	4.812s	85.942s	1.904s	**0.809s**
susy	0.070	–	23.798s	19.792s	1.902s	**1.546s**
higgs	0.001	–	146.413s	156.578s	55.175s	**4.896s**

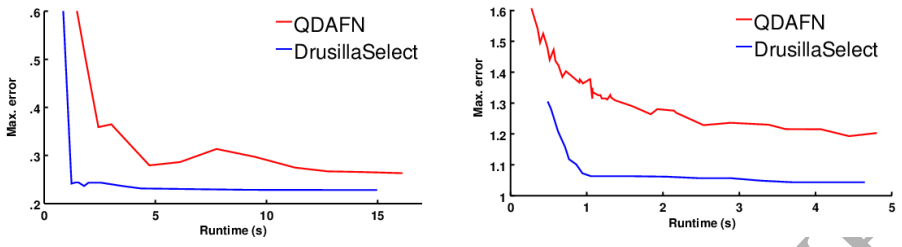
Table 5: Runtimes for $\epsilon = 0.05$-approximate furthest neighbor search. 'b.f.': brute-force; 's.t.': single-tree; 'd.t.': dual-tree; DS: `DrusillaSelect`.

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). Also shown is the average hardness $h(S_q, S_r)$ of the problem. The variance of the runtimes was negligible on our system.

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 `ball` 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 a few extreme examples: the `miniboone` and `higgs` datasets, where the hardness of the problem is extremely low, and the random datasets (`randu`, `ball`, and `randn`), which have the highest hardness values by far.

For the `miniboone` dataset, which lies on a low-dimensional manifold in a high-dimensional space, `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 with QDAFN until using as many as 125 projection bases. But because the hardness of the problem is so low with the `miniboone` and `higgs` datasets, very few points need to be selected to provide good approximation. `DrusillaSelect` excels in this situation, since it chooses points based on the data distribution. A similar effect, where QDAFN



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

Figure 6: Maximum error for QDAFN and `DrusillaSelect` as a function of runtime.

needed many projection bases to get a reliably good approximation, was also observed with the `covertype` and `gisette` datasets.

At the other end of the hardness spectrum are the random datasets. Of these, the `ball` dataset is the hardest, since for each query point, the furthest neighbor will generally be a point directly across the ball. This means, to achieve a good approximation for every single point, we must capture points from virtually every direction in the space. This worst-case situation washes away any advantage the selection procedure of `DrusillaSelect` might typically have, and QDAFN is the best choice here. Note that the `DrusillaSelect` algorithm specifically ensures that projection bases are not too similar (see lines 18–20); this allows the algorithm to achieve reasonable performance in spite of the dataset.

QDAFN is able to search far fewer points (40) than `DrusillaSelect` (1100) to achieve the same average approximation on the `ball` dataset; but the runtime for QDAFN is still greater as a result of the priority queue structure it must maintain. As dimensionality increases, QDAFN is likely to perform increasingly better than `DrusillaSelect`.

For the `randu` and `randn` datasets, there is still structure, since the norm of every point is not 1. Even though these problems are still difficult, `DrusillaSelect` is able to exploit this structure.

8.2. Maximum error

Another important property of `DrusillaSelect` is that it gives a small maximum error compared to QDAFN. Figure 6 shows the maximum error of each approach as the number of points scanned increase on the `covertype` and `gisette` datasets. 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.

We can see that `DrusillaSelect` is able to very quickly decrease the maximum approximation error with very few candidate points.

8.3. Effect of increasing entropy

In this section, we wish to characterize the performance of `DrusillaSelect` as compared to QDAFN when the entropy of a dataset is increased by artificial means—specifically, by compressing the mean-centered norms of the points.

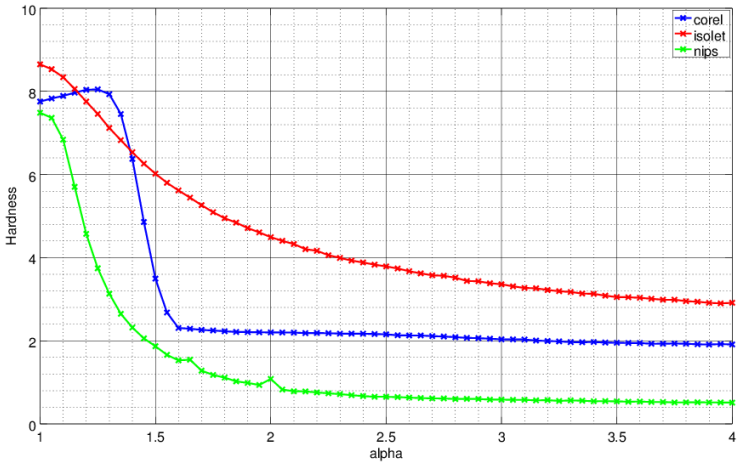


Figure 7: Average hardness values $h(\cdot, \cdot)$ as α is swept from low to high. Higher α implies easier furthest neighbor searches.

This can be understood as an alternate approach to techniques like mutual proximity or SimHub [28, 29], which aim to reduce the hubness of the data.

Given some parameter $\alpha \in [1, \infty)$, we wish that the ratio of the maximum mean-centered norm to the minimum mean-centered norm be equal to α. That is, given some mean-centered dataset S, we wish to transform our data such that

$$\frac{\max_{p_i \in S} \|p_i\|}{\min_{p_i \in S} \|p_i\|} = \alpha. \tag{19}$$

Thus, when $\alpha = 1$, the points all lie on the surface of the unit ball. This transform can be performed using the following formula:

$$\hat{p}_i = \left(\frac{\|p_i\| / \min_{p \in S} \|p\| - 1}{\|p_i\| / \max_{p \in S} \|p\| - 1} (\alpha - 1) + 1 \right) \frac{p_i}{\|p_i\|}. \tag{20}$$

The goal of this transformation is to make the furthest neighbor search task more difficult as $\alpha \to 1$. As α gets larger, the distribution of norms grows larger, and we can expect that some points in the dataset will become more important than others for furthest neighbor search results.

We test on the `corel`, `isolet`, and `nips` datasets (low, medium, and high-dimensional). Without modification, these datasets have α values of 18.142, 1.437, and 3.975, respectively. In Figure 7, the average hardness (using 10-fold cross-validation) is given for the three datasets modified with different α values. We can see that, as we expect, the hardness increases as α goes to 1 (that is, as the points lie closer to the surface of the unit ball).

For a first experiment, we wish to see how many points the QDAFN and `DrusillaSelect` algorithms each need to use to provide (on average) $\epsilon = 0.05$-approximate furthest neighbor search results as α is varied. As in the previous

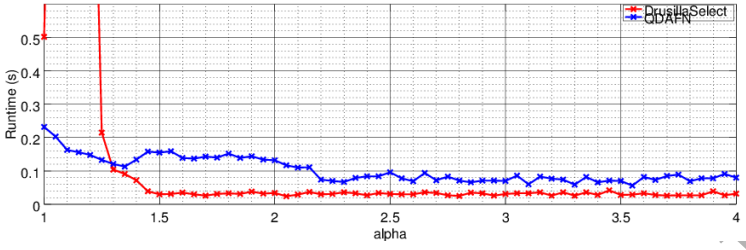
			QDAFN params		DS params		Runtime (s)	
Dataset	α	$h(\cdot)$	l	m	l	m	QDAFN	DS
corel	1.00	7.755	40	42	11	45	0.232s	0.502s
corel	1.25	8.048	18	24	1	31	0.133s	0.215s
corel	1.50	3.495	28	29	1	2	0.155s	0.030s
corel	1.75	2.247	25	26	1	2	0.140s	0.031s
corel	2.00	2.201	15	22	1	2	0.132s	0.034s
corel	3.00	2.034	8	11	1	2	0.070s	0.031s
corel	4.00	1.912	8	11	1	2	0.080s	0.032s
isolet	1.00	8.651	19	24	11	45	0.091s	0.796s
isolet	1.25	7.457	28	33	1	43	0.127s	0.329s
isolet	1.50	6.019	38	38	1	15	0.153s	0.135s
isolet	1.75	5.093	34	39	1	8	0.160s	0.082s
isolet	2.00	4.492	43	38	1	5	0.162s	0.065s
isolet	3.00	3.360	51	54	1	4	0.196s	0.055s
isolet	4.00	2.913	52	53	1	9	0.199s	0.099s
nips	1.00	7.486	3	4	1	5	1.961s	0.883s
nips	1.25	3.744	9	12	1	2	2.542s	0.785s
nips	1.50	1.868	33	39	1	2	4.365s	0.628s
nips	1.75	1.178	20	26	1	2	3.738s	0.637s
nips	2.00	1.082	14	21	1	2	3.332s	0.637s
nips	3.00	0.584	11	12	1	2	2.795s	0.658s
nips	4.00	0.512	9	12	1	2	2.908s	0.654s

Table 6: QDAFN and `DrusillaSelect` comparison for varying values of α; smaller α indicates more distance compression.

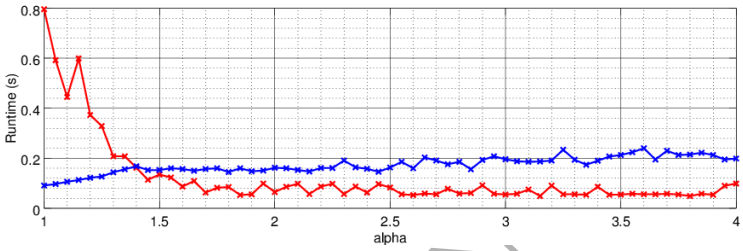
experiments, the dataset is split 70% into a reference set and 30% into a query set, and this is done ten times. The average results are shown. Table 6 and Figure 8 show the results.

It is clear that as α increases, `DrusillaSelect` is able to quickly exploit the structure of the dataset and return adequate results with very few points, whereas QDAFN, since it chooses points randomly, cannot do the same. For the high-dimensional `nips` dataset, `DrusillaSelect` is always able to select few points that give good average error, even for very small α.

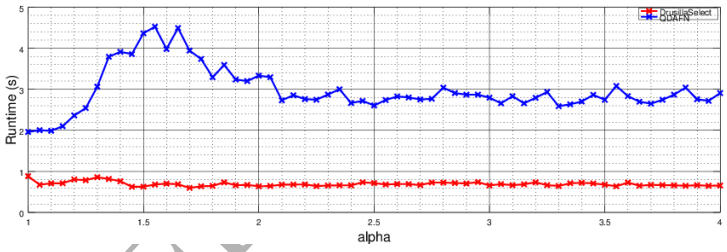
When α is very close to 1, the dataset has little structure that `DrusillaSelect` can exploit. This means that the random projections used for QDAFN are roughly equally effective, and `DrusillaSelect` has to collect almost as many points as QDAFN to give good results. For instance, with the `corel` dataset and $\alpha = 1$, `DrusillaSelect` must collect $11 \cdot 45 = 495$ candidates, and QDAFN must collect $40 \cdot 42 = 1680$ candidates. However, the priority queue used in QDAFN allows that algorithm to have much faster search time, since only 40 candidates are inspected instead of the full 1680 candidates. Were `DrusillaSelect` modified to also use a priority queue to search through its candidates, the runtimes would likely be much more equal; however, in most situations we have investigated, `DrusillaSelect` needs to collect so few points that a priority queue



(a) `corel` dataset.



(b) `isolet` dataset.



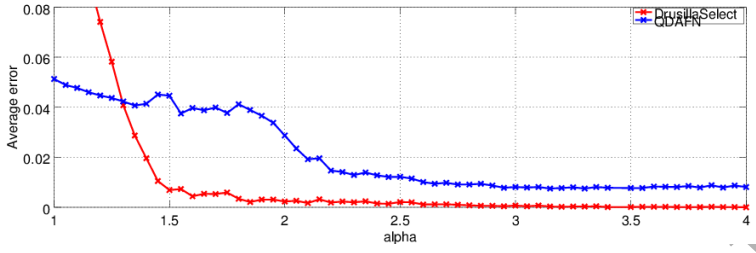
(c) `nips` dataset.

Figure 8: QDAFN and `DrusillaSelect` comparison for varying α: runtime (in seconds) vs. α value to produce 1.05-approximate results.

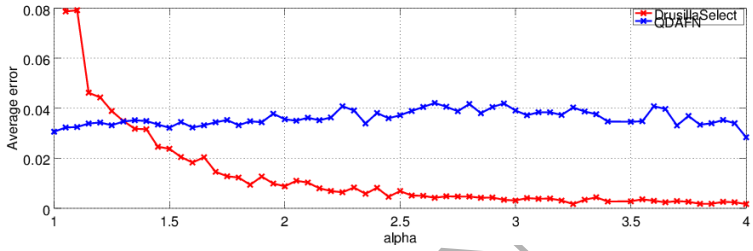
search is not worth the overhead.

Next, we wish to see what the approximation accuracy is when we allow each algorithm a fixed runtime, which we enforce by choosing specific l and m values for each algorithm. For `DrusillaSelect`, we use $l = 5$ and $m = 20$, and for QDAFN, we tune so that the algorithm takes the same amount of time. For the `corel` dataset, the settings for QDAFN are $l = 35$, $m = 35$; for `isolet`, $l = 75$, $m = 75$; and for `nips`, $l = 16$, $m = 16$. For each of the three datasets `corel`, `isolet`, and `nips`, we sweep α from 1 to 4 and plot the average approximation error returned. Figure 9 shows the results.

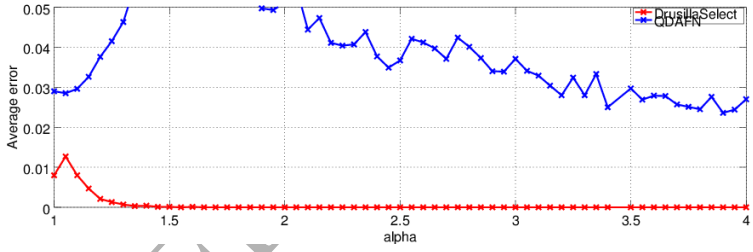
For very small α, random projection bases give better overall error, so QDAFN has better error numbers, but taking the data distribution into account as α gets larger is very important. As a result, `DrusillaSelect` significantly outperforms QDAFN as α increases. For the high-dimensional `nips` dataset,



(a) `corel` dataset.



(b) `isolet` dataset.



(c) `nips` dataset.

Figure 9: Average error as a function of α for fixed l and m parameters for `DrusillaSelect` and QDAFN.

QDAFN is not able to choose good projection bases for any value of α, whereas for $\alpha > 1.5$, `DrusillaSelect` is able to achieve near-zero average approximation error.

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) and the tree-based algorithms also suffer in high-dimensional settings, `DrusillaSelect` effectively captures the low-dimensional structure and exploits the furthest neighbor search problem's properties with few projection bases. Even for difficult datasets with high hardness measure, when there is structure to exploit, `DrusillaSelect` is able to do this effectively.

9. Conclusion

After an in-depth study of the properties of the furthest neighbor search problem, we have proposed an information theoretic measure of hardness, $h(S_q, S_r)$, that captures the difficulty of the furthest neighbor search problem.

Using the intuitions from that study, we have proposed an algorithm called `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 [30, 31]. 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 *Proceedings of the 8th International Conference on Similarity Search and Applications (SISAP), Glasgow, Scotland*, volume 9371, pages 3–14. Springer LNCS, Oct. 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] Ryan R. Curtin and Andrew B. Gardner. Fast approximate furthest neighbors with data-dependent candidate selection. In *Proceedings of the 8th International Conference on Similarity Search and Applications (SISAP), Tokyo, Japan*, volume 9939, pages 221–235. Springer LNCS, Oct. 2016.

[15] 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.

[16] 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.

[17] 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.

[18] R.R. Curtin. Faster dual-tree traversal for nearest neighbor search. In *Proceedings of the 8th International Conference on Similarity Search and Applications (SISAP), Glasgow, Scotland*, volume 9371, pages 77–89. Springer LNCS, Oct. 2015.

[19] S. Bespamyatnikh. Dynamic algorithms for approximate neighbor searching. In *Proceedings of the 8th Canadian Conference on Computational Geometry (CCCG'96)*, pages 252–257, 1996.

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

[21] 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.

[22] 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.

[23] 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.

[24] M. Lichman. UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml`, University of California Irvine, School of Information and Computer Sciences, 2013.

[25] 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.

[26] N. Tomašev, M. Radovanović, D. Mladenić, and M. Ivanović. The role of hubness in clustering high-dimensional data. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):739–751, 2014.

[27] 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.

[28] D. Schnitzer, A. Flexer, M. Schedl, and G. Widmer. Using mutual proximity to improve content-based audio similarity. In *Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR)*, volume 12, pages 79–84, 2011.

[29] N. Tomašev and D. Mladenić. Hubness-aware shared neighbor distances for high-dimensional k-nearest neighbor classification. *Knowledge and Information Systems*, 39(1):89–122, 2014.

[30] 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.

[31] R.R. Curtin and P. Ram. Dual-tree fast exact max-kernel search. *Statistical Analysis and Data Mining*, 7(4):229–253, 2014.