A note on quickly finding the nearest neighbour

David Barber
Department of Computer Science
University College London

May 19, 2014

1 Finding your nearest neighbour quickly

Consider that we have a set of datapoints $\mathbf{x}^1, \ldots, \mathbf{x}^N$ and a new query vector $\mathbf{q}$. Our task is to find the nearest neighbour to the query $n^* = \underset{n}{\operatorname{argmin}}\, d(\mathbf{q}, \mathbf{x}^n)$, $n = 1, \ldots, N$. For the Euclidean distance

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^2} = \sqrt{\sum_{i=1}^{D} (x_i - y_i)^2} \tag{1}$$

and D dimensional vectors, it takes $O(D)$ operations to compute this distance. For a set of N vectors, computing the nearest neighbour to $\mathbf{q}$ would take then $O(DN)$ operations. For large datasets this can be prohibitively expensive. Is there a way to avoid calculating all the distances? This is a large research area (see [2] for a review) and we will focus here on first methods that make use of the triangle inequality for metric distances and secondly a KD-trees which form a spatial data structure.

2 Using the triangle inequality to speed up search

2.1 The triangle inequality

For the Euclidean distance we have

$$(\mathbf{x} - \mathbf{y})^2 = (\mathbf{x} - \mathbf{z} + \mathbf{z} - \mathbf{y})^2 = (\mathbf{x} - \mathbf{z})^2 + (\mathbf{z} - \mathbf{y})^2 + 2(\mathbf{x} - \mathbf{z})(\mathbf{z} - \mathbf{y}) \tag{2}$$

Since the scalar product between two vectors $\mathbf{a}$ and $\mathbf{b}$ relates the lengths of the vectors $|\mathbf{a}|$, $|\mathbf{b}|$ and the angle θ between them by $\mathbf{a}^\mathsf{T}\mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos(\theta)$ we have (using $|\mathbf{x}| \equiv \sqrt{\mathbf{x}^\mathsf{T}\mathbf{x}}$)

$$|\mathbf{x} - \mathbf{y}|^2 = |\mathbf{x} - \mathbf{z}|^2 + |\mathbf{z} - \mathbf{y}|^2 + 2\cos(\theta)|\mathbf{x} - \mathbf{z}||\mathbf{z} - \mathbf{y}| \tag{3}$$

Using $\cos(\theta) \leq 1$ we obtain the triangle inequality

$$|\mathbf{x} - \mathbf{y}| \leq |\mathbf{x} - \mathbf{z}| + |\mathbf{z} - \mathbf{y}| \tag{4}$$

Geometrically this simply says that for a triangle formed by the points $\mathbf{x}, \mathbf{y}, \mathbf{z}$, it is shorter to go from $\mathbf{x}$ to $\mathbf{y}$ than to go from $\mathbf{x}$ to an intermediate point $\mathbf{z}$ and then from that point to $\mathbf{y}$.

More generally a distance $d(\mathbf{x}, \mathbf{y})$ satisfies the triangle inequality if it is of the form[1]

$$d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{y}, \mathbf{z}) \tag{5}$$

Formally the distance is a metric if it is symmetric $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$, non negative, $d(\mathbf{x}, \mathbf{y}) \geq 0$ and $d(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$.

[1]Note here that $d(\cdot, \cdot)$ is defined as the Euclidean distance, not the squared Euclidean distance.

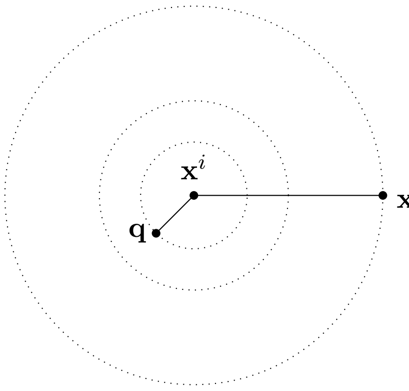


Figure 1: If we know that $\mathbf{x}^i$ is close to $\mathbf{q}$, but that $\mathbf{x}^i$ and $\mathbf{x}^j$ are not close, namely $d(\mathbf{q},\mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i,\mathbf{x}^j)$, then we can infer that $\mathbf{x}^j$ will not be closer to $\mathbf{q}$ than $\mathbf{x}^i$, *i.e.* $d(\mathbf{q},\mathbf{x}^i) \leq d(\mathbf{q},\mathbf{x}^j)$.

A useful basic fact that we can deduce for such metric distances is the following: If $d(\mathbf{x},\mathbf{y}) \leq \frac{1}{2}d(\mathbf{z},\mathbf{y})$, then $d(\mathbf{x},\mathbf{y}) \leq d(\mathbf{x},\mathbf{z})$, meaning that we do not need to compute $d(\mathbf{x},\mathbf{z})$. To see this consider

$$d(\mathbf{y},\mathbf{z}) \leq d(\mathbf{y},\mathbf{x}) + d(\mathbf{x},\mathbf{z}) \tag{6}$$

If we are in the situation that $d(\mathbf{x},\mathbf{y}) \leq \frac{1}{2}d(\mathbf{z},\mathbf{y})$, then we can write

$$2d(\mathbf{x},\mathbf{y}) \leq d(\mathbf{y},\mathbf{x}) + d(\mathbf{x},\mathbf{z}) \tag{7}$$

and hence $d(\mathbf{x},\mathbf{y}) \leq d(\mathbf{x},\mathbf{z})$. In the nearest neighbour context, we can infer that if $d(\mathbf{q},\mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i,\mathbf{x}^j)$ then $d(\mathbf{q},\mathbf{x}^i) \leq d(\mathbf{q},\mathbf{x}^j)$, see fig(1).

2.2 Using all datapoint to datapoint distances

One way to use the above result is to first precompute all the distance pairs $d_{ij} \equiv d(\mathbf{x}^i,\mathbf{x}^j)$ in the dataset.

Orchard's algorithm

Given these distances, for each i we can then compute an ordered list $\mathcal{L}^i = \left\{j_1^i, j_2^i, \ldots, j^{i_{N-1}}\right\}$ of those vectors $\mathbf{x}^j$ that are closest to $\mathbf{x}^i$, with $d(\mathbf{x}^i,\mathbf{x}^{j_1^i}) \leq d(\mathbf{x}^i,\mathbf{x}^{j_2^i}) \leq d(\mathbf{x}^i,\mathbf{x}^{j_3^i})\ldots$.

We then start with some vector $\mathbf{x}^i$ as our current best guess for the nearest neighbour to $\mathbf{q}$ and compute $d(\mathbf{q},\mathbf{x}^i)$. We then examine the first element of the list $\mathcal{L}^i$ and consider the following cases:

If $d(\mathbf{q},\mathbf{x}^i) \leq \frac{1}{2}d_{i,j_1^i}$ then j_1^i cannot be closer than $\mathbf{x}^i$ to $\mathbf{q}$; furthermore, neither can any of the other members of this list since they automatically satisfy this bound as well. In this fortunate situation, $\mathbf{x}^i$ must be the nearest neighbour to $\mathbf{q}$.

If $d(\mathbf{q},\mathbf{x}^i) \not\leq \frac{1}{2}d_{i,j_1^i}$ then we move on to the next member of the list, j^{i_2} and compute $d(\mathbf{q},\mathbf{x}^{j_2^i})$. If $d(\mathbf{q},\mathbf{x}^{j_2^i}) < d(\mathbf{q},\mathbf{x}^i)$ we have found a better candidate $i' \equiv j_2^i$ than our current best guess, and we jump to the start of the new list $\mathcal{L}^{i'}$. Otherwise we continue to traverse the current list, checking if $d(\mathbf{q},\mathbf{x}^i) \leq \frac{1}{2}d_{i,j_2^i}$, *etc.* [6].

In summary, this process of traversing a list for the candidate continues until we either: (i) find a better candidate (and jump to the top of its list) and restart traversing the new candidate list (ii) find that the bound criterion is met and declare that the current candidate is then optimal (iii) get to the end of a list, in which case the current candidate is optimal.

See algorithm(1) and `fastnnOrchard.m` for a formal description of the algorithm.

Complexity

Orchard's algorithm requires $O\left(D^2\right)$ storage to precompute the distance sorted lists. In the worst case, it can take $O\left(N\right)$ distance calculations to find the nearest neighbour. To see this, consider a simple one

Algorithm 1 Orchard's Nearest Neighbour Search

```
1: Compute all pairwise distances metric(data{$m$},data{$n$})
2: For each datapoint $n$, compute the list{n} that stores the distance list{n}.distance and index list{n}.index
   of each other datapoint, sorted by increasing distance, list{n}.distance(1)¡list{n}.distance(2)...
3: for all Query points do
4:     cand.index=randi($N$)        ▷ assign first candidate index randomly as one of the $N$ datapoint indices
5:     cand.distance=metric(query,data{cand.index})
6:     Assign all nodes to state not tested
7:     $i = 1$                                                ▷ start at the beginning of the list
8:     while $i \leq N$ and list{cand.index}.distance($i$)<2*cand.distance do
9:         node=list{cand.index}.index($i$)             ▷ get the index of the $i^{th}$ member of the current list
10:        if tested(node)=false then                   ▷ just to avoid computing this distance again
11:            tested(node)=true
12:            querydistance=metric(query,data{node})
13:            if querydistance<cand.distance then                    ▷ found a better candidate
14:                cand.index=node;
15:                cand.distance=querydistance;
16:                $i = 1$                                             ▷ go to start of next list
17:            else
18:                $i = i + 1$                              ▷ continue to traverse the current list
19:            end if
20:            $i = i + 1$
21:        end if
22:    end while
23:    cand.index and cand.distance contain the nearest neighbour and distance thereto
24: end for
```

dimensional dataset:

$$x^{n+1} = x^n + 1, \quad n = 1, \ldots, N \qquad x^1 = 0 \tag{8}$$

If the query point is for example $q = x^N + 1$ and our initial candidate for the nearest neighbour is x^1, then at iteration k, the nearest non-visited datapoint will be x^{k+1}, meaning that we will simply walk through all the data, see fig(2).

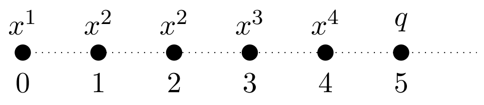


Figure 2: A worst-case scenario for Orchard's algorithm. If the initial candidate is x^1, then x^2 becomes the next best candidate, and subsequenly x^3 *etc.*

Approximating and Eliminating Search Algorithm (AESA)

The triangle inequality can be used to form a lower bound

$$d(\mathbf{q}, \mathbf{x}^j) \geq d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{x}^j) \tag{9}$$

For datapoints $\mathbf{x}^i$, $i \in \mathcal{I}$ for which $d(\mathbf{q}, \mathbf{x}^i)$ has already been computed, one can then maximise the lower bounds to find the tightest lower bound on all other $d(\mathbf{q}, \mathbf{x}^j)$[2]:

$$d(\mathbf{q}, \mathbf{x}^j) \geq \max_{i \in \mathcal{I}} d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{x}^j) \equiv L_j \tag{10}$$

All datapoints $\mathbf{x}^j$ whose lower bound is greater than the current best nearest neighbour distance can then be eliminated, see fig(3). One may then select the next (non-eliminated) candidate datapoint $\mathbf{x}^j$ corresponding to the lowest bound and continue, updating the bound and eliminating [7], see algorithm(2) and `fastnnAESA.m`.

[2]In the classic AESA one only retains the best current nearest neighbour $\mathcal{I} = best$.

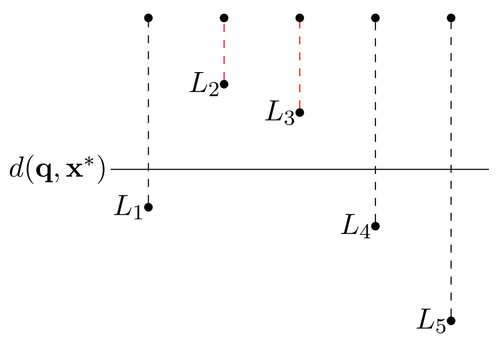


Figure 3: We can eliminate datapoints $\mathbf{x}^2$ and $\mathbf{x}^3$ since their distance to the query is greater than the current best candidate distance $d(\mathbf{q}, \mathbf{x}^*)$. After eliminating these points, we use the datapoint with the lowest bound to suggest the next candidate, in this case $\mathbf{x}^5$.

Algorithm 2 AESA Nearest Neighbour Search

```
1: For all datapoints compute and store the distances $d(\mathbf{x}^i, \mathbf{x}^j)$
2: for all Queries do
3:     best.dist=$\infty$
4:     $\mathcal{I} = \emptyset$                                                        ▷ Datapoints examined
5:     $\mathcal{J} = \{1, \ldots, N\}$                                            ▷ Datapoints not examined
6:     $L(n) = \infty, \quad n = 1, \ldots, N$
7:     while $\mathcal{J}$ is not empty do
8:         cand.ind=$\arg\min_{j \in \mathcal{J}} bound(j)$                  ▷ select candidate based on lowest bound
9:         cand.dist=metric(query,data{cand.ind})
10:        distQueryData(cand.ind)=cand.dist                                   ▷ store computed distances
11:        $\mathcal{I} = \mathcal{I} \cup$ cand.ind                          ▷ Add candidate to examined list
12:        if cand.dist<best.dist then                                 ▷ If candidate is nearer than current best
13:            best.dist=cand.dist
14:            best.ind=cand.ind
15:        end if
16:        $L(j) = \max_{i \in \mathcal{I}}$ distQueryData$(i) - d(\mathbf{x}^i, \mathbf{x}^j), \quad j \in \mathcal{J} \setminus$ cand.ind     ▷ lower bound
17:        $\mathcal{J} = \{j$ such that $L(j) <$ best.dist$\}$                     ▷ eliminate
18:    end while
19: end for
```

Complexity

As for Orchard's algorithm, AESE requires $O\left(D^2\right)$ storage to precompute the distance matrix $d(\mathbf{x}^i, \mathbf{x}^j)$. During the first lower bound computation, when no datapoints have yet been eliminated, AESE needs to compute all the $N-1$ lower bounds for datapoints other than the first candidate examined. Whilst each lower bound is fast to compute, this still requires an $O(N)$ computation. A similar computation is required to update the bounds at later stages, with the worst case being that there are $N/2$ datapoints left to examine, meaning that computing the optimal lower bound scales $O\left(N^2\right)$ in this case. One can limit this complexity by restricting the elements of $\mathcal{I}$ in the max operation to compute the bound; however this may result in more iterations being required since the bound is then potentially inferior.

The experiments in `demofastnn.m` also suggest that the AESA method typically requires less distance calculations than Orchard's approach. However, there remains at least an $O(N)$ calculation required to compute the bounds in AESA which may be prohibitive, depending on the application.

2.3 Using the datapoints to 'buoys' distances

Both Orchard's algorithm and AESA can significantly reduce the number of distance calculations required. However, we pay an $O\left(N^2\right)$ storage cost. For very large datasets, this storage cost is likely to be prohibitive. Given the difficulty in storing $d_{i,j}$, an alternative is to consider the distances between the training points and a smaller number of strategically placed 'buoys'[3], $\mathbf{b}^1, \ldots, \mathbf{b}^B$, $B < N$. These buoys can be either a subset of the original datapoints, or new positions.

[3]Also called 'pivots' or 'basis' vectors by other authors.

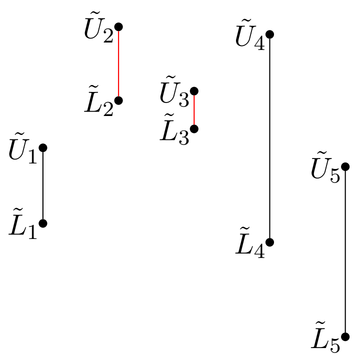


Figure 4: We can eliminate datapoint $\mathbf{x}^2$ since there is another datapoint (either $\mathbf{x}^1$ or $\mathbf{x}^5$) that has an upper bound $\tilde{U}$ that is lower than $\tilde{L}_2$. We can similarly eliminate $\mathbf{x}^3$.

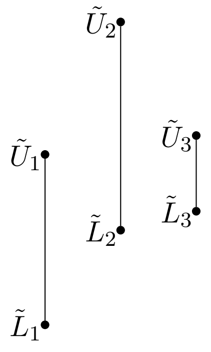


Figure 5: The lower bounds of non-eliminated datapoints from fig(4) relabelled such that $L_1 \leq L_2 \leq \ldots$. In 'linear' AESA we use these lower bounds to order the search for the nearest neighbour, starting with $\mathbf{x}^1$. If we get to a bound where L_m is greater than our current best distance, then all remaining distances must be greater than our current best distance, and the algorithm terminates.

Pre-elimination

Given the buoys, the triangle inequality gives the following upper and lower bounds on the distance from the query to each datapoint:

$$d(\mathbf{q}, \mathbf{x}^n) \geq \max_m d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{b}^m, \mathbf{x}^n) \equiv \tilde{L}_n \tag{11}$$

$$d(\mathbf{q}, \mathbf{x}^n) \leq \min_m d(\mathbf{q}, \mathbf{b}^m) + d(\mathbf{b}^m, \mathbf{x}^n) \equiv \tilde{U}_n \tag{12}$$

We can then immediately eliminate any m for which there is some $n \neq m$ with $\tilde{L}(m) \geq \tilde{U}(n)$, see fig(4). This enables one to 'pre-eliminate' datapoints, at a cost of B distance calculations, see `fastnnBuoysElim.m`. The remaining candidates can then be used in the Orchard or AESA algorithms (either the standard ones described above or the buoy variants described below).

AESA with buoys

In place of the exact distances to the datapoints, an alternative is to relabel the datapoints according to $\tilde{L}_n$, with lowest distance first $\tilde{L}_1 \leq \tilde{L}_2, \ldots \leq \tilde{L}_n$. We can then compute the distance $d(\mathbf{q}, \mathbf{x}^1)$ and compare this to $\tilde{L}_2$. If $d(\mathbf{q}, \mathbf{x}^1) \leq \tilde{L}_2$ then $\mathbf{x}^1$ must be the nearest neighbour, and the algorithm terminates. Otherwise we move on to the next candidate $\mathbf{x}^2$. If this datapoint has a lower distance than our current best guess, we update our current best guess accordingly. We then move on to the next candidate in the list and continue. If we reach a candidate in the list for which $d(\mathbf{q}, \mathbf{x}^{best}) \leq \tilde{L}_m$ the algorithm terminates, see fig(5). This algorithm is also called 'linear' AESA [5], see `fastnnLAESA.m`.

The gain here is that the storage costs are reduced to $O(NB)$ since we only now need to pre-compute the distances between the buoys and the dataset vectors. By choosing $B \ll N$, this can be a significant saving. The loss is that, since we are now not using the true distance but a bound, we may need more distance calculations $d(\mathbf{q}, \mathbf{x}^i)$.

Orchard with buoys

For Orchard's algorithm we can also use buoys to construct bounds on $d_{i,j}$ on the fly. Consider an arbitrary vector $\mathbf{b}$, then,

$$d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b}) \leq d(\mathbf{x}^i, \mathbf{x}^j) \tag{13}$$

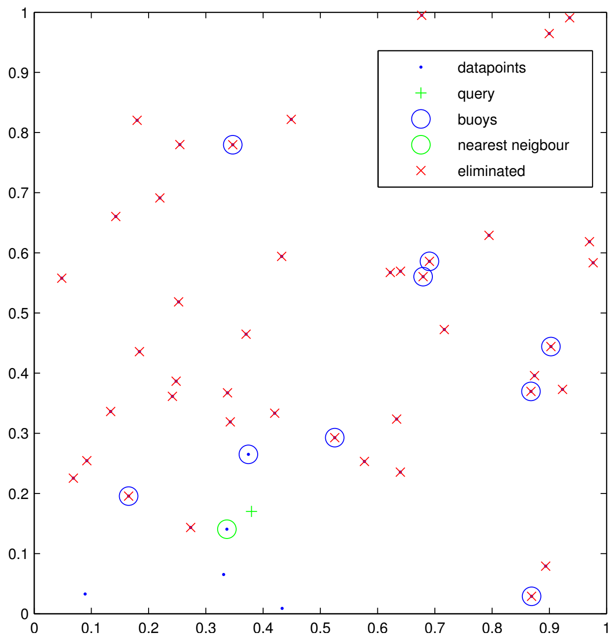


Figure 6: Example of elimination using buoys. All points expect for the query are datapoints. Using buoys, we can pre-eliminate (crossed datapoints) a large number of the datapoints from further consideration. See `demofastnn.m`.

We can use this in Orchard's approach since if there is a vector $\mathbf{b}$ such that

$$2d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b}) \tag{14}$$

then it follows that $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$. This suggests that we can replace using the exact distance $d(\mathbf{x}^i, \mathbf{x}^j)$ with an upper bound $d(\mathbf{x}^i, \mathbf{b}) - d(\mathbf{x}^j, \mathbf{b})$. If we have a set of such buoys $\mathbf{b}^1, \ldots, \mathbf{b}^B$, we want to use the highest lower bound approximation to the true distance $d(\mathbf{x}^i, \mathbf{x}^j)$:

$$2d(\mathbf{q}, \mathbf{x}^i) \leq \max_m d(\mathbf{x}^i, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \equiv \tilde{d}_{i,j} \tag{15}$$

These surrogate distances $\tilde{d}_{i,j}$ can then be used as in the standard Orchard algorithm to form (on the fly) a list $\mathcal{L}^i$ of closest vectors $\mathbf{x}^j$ to the current candidate $\mathbf{x}^i$, sorted according to this surrogate distance. The algorithm then proceeds as before, see `fastnnOrchardBuoys.m`.

Whilst this may seem useful, one can show that AESA-buoys dominates Orchard-buoys. In AESA-buoys the lower bound on $d(\mathbf{q}, \mathbf{x}^i)$ is given by

$$\tilde{L}_i = \max_m d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) \tag{16}$$

Let m be the optimal buoy index for i, namely

$$\tilde{L}_i = d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) \tag{17}$$

Furthermore

$$\tilde{L}_j = \max_m d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \tag{18}$$

If we have reached the Orchard-buoys termination criterion

$$-d(\mathbf{x}^j, \mathbf{b}^m) \geq 2d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{b}^m) \quad \text{for all } j > i \tag{19}$$

then datapoint $\mathbf{x}^i$ is the nearest neighbour. Hence, if Orchard-buoys terminates for $\mathbf{x}^i$, we must have, for $j > i$:

$$\tilde{L}_j \geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) + 2d(\mathbf{q}, \mathbf{x}^i) \geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^i, \mathbf{b}^m) = \tilde{L}_i \quad \text{for all } j > i \tag{20}$$

Furthermore,

$$\begin{aligned} \tilde{L}_j &\geq d(\mathbf{q}, \mathbf{b}^m) - d(\mathbf{x}^j, \mathbf{b}^m) \\ &\geq d(\mathbf{q}, \mathbf{b}^m) + d(\mathbf{q}, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{b}^m) + d(\mathbf{q}, \mathbf{x}^i) \\ &\geq \underbrace{d(\mathbf{b}^m, \mathbf{x}^i) - d(\mathbf{x}^i, \mathbf{b}^m)}_{0} + d(\mathbf{q}, \mathbf{x}^i) \quad \text{for all } j > i \end{aligned} \tag{21}$$

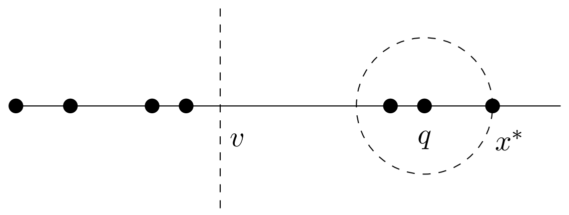


Figure 7: Consider one dimensional data in which the datapoints are partioned into those that lie to the left of v and those to the right. If the current best candidate x^* has a distance to the query q that is less than the distance of the query q to v, then none of the points to the left of v can be the nearest neighbour.

which is precisely the AESA-bouys termination criterion. Hence Orchard-buoys terminating implies that AESA-buoys terminates. Since AESA-buoys has a different termination criterion to Orchard-buoys, AESA-buoys has the opportunity to terminate before Orchard-buoys. This explains why Orchard-buoys cannot outperform AESA-buoys in terms of the number of distance calculations $d(\mathbf{q}, \mathbf{x}^i)$. This observation is also borne out in `demofastnn.m`, see also fig(6) for a $D = 2$ demonstration.

2.4 Other uses of nearest neighbours

Note also that nearest neighbour calculations are required in other applications, for example k-means clustering – see [4] for a fast algorithm based on similar applications of the triangle inequality.

3 KD trees

K-dimensional trees [1] are a way to form a partition of the space that can be used to help speed up search. Before introducing the tree, we'll discuss the basic idea on which the potential speed-up is based.

3.1 Basic idea in one-dimension

If we consider first one-dimensional data $x^n, n = 1, \ldots, N$ we can partition the data into points that have value less than a chosen value v, and those with a value greater than this, see fig(7). If the distance of the current best candidate x^* to the query point q is smaller than the distance of the query to v, then points to the left of v cannot be the nearest neighbour. To see this, consider a query point that is in the right space, $q > v$ and a candidate x that is in the left space, $x < v$, then

$$(x-q)^2 = (x-v+v-q)^2 = (x-v)^2 + 2\underbrace{(x-v)}_{\leq 0}\underbrace{(v-q)}_{\leq 0} + (v-q)^2 \geq (v-q)^2 \tag{22}$$

Let the distance of the current best candidate to the query be $\delta^2 \equiv (x^* - q)^2$. Then if $(v-q)^2 \geq \delta^2$ it follows that all points in the left space are further from q than x^*.

In the more general K dimensional case, consider a query vector $\mathbf{q}$. Let's imagine that we have partitioned the datapoints into those with first dimension x_1 less than a defined value v (to its 'left'), and those with a value greater or equal to v (to its 'right'):

$$\mathcal{L} = \{\mathbf{x}^n : x_1^n < v\}, \qquad \mathcal{R} = \{\mathbf{x}^n : x_1^n \geq v\} \tag{23}$$

Let's also say that our current best nearest neighbour candidate has squared Euclidean distance $\delta^2 = \left(\mathbf{q} - \mathbf{x}^i\right)^2$ from $\mathbf{q}$ and that $q_1 \geq v$. The squared Euclidean distance of any datapoint $\mathbf{x} \in \mathcal{L}$ to the query is

$$(\mathbf{x}-\mathbf{q})^2 = \sum_k (x_k - q_k)^2 \geq (x_1 - q_1)^2 \geq (v - q_1)^2 \tag{24}$$

If $(v-q_1)^2 > \delta^2$, then $(\mathbf{x}-\mathbf{q})^2 > \delta^2$. That is, all points in $\mathcal{L}$ must be further from $\mathbf{q}$ than the current best point $\mathbf{x}^i$. On the other hand, if $(v-q_1)^2 \leq \delta^2$, then it is possible that some point in $\mathcal{L}$ might be closer to $\mathbf{q}$ than our current best nearest neighbour candidate, and we need to check these points.

The KD-tree approach essentially is a recursive application of the above intuition.

3.2 Constructing the tree

For N datapoints, the tree consists of N nodes. Each node contains a datapoint, along with the axis along which the data is split.

We first need to define a routine `[middata leftdata rightdata]=splitdata(x,axis)` that splits data along a specified dimension of the data. Whilst not necessary, it is customary to use the median value along the split dimension to partition. The routine should return :

`middata` We first form the set of scalar values that correspond to the `axis` components of `x`. These are sorted and the `middata` is the datapoint close to the median of the data.

`leftdata` These are the datapoints 'to the left' of the `middata` datapoint.

`rightdata` These are the datapoints 'to the right' of the `middata` datapoint.

For example, if the datapoints are $(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)$ and we split along dimension 1, then we would have:

```
>>x =

     2     5     9     4     8     7
     3     4     6     7     1     2


>> [middata leftdata rightdata]=splitdata(x,1)

middata =

     7
     2


leftdata =

     2     4     5
     3     7     4


rightdata =

     8     9
     1     6
```

The tree can be constructed recursively as follows. We start with `node(1)` and create a temporary storage `node(1).data` that contains the complete dataset. We then call

```
[middata leftdata rightdata]=splitdata(node(1).data,1)
```

forming `node(1).x=middata`. We now form two child nodes and populate them with data `node(2).data=leftdata; node(3).data=rightdata`. We store also in node(1).axis which axis was used to split the data.

$$\begin{tikzcd}
& 1 \arrow[dl] \arrow[dr] & \\
2 & & 3
\end{tikzcd}$$

The temporary data `node(1).data` can now be removed. We now move down to the second layer, and split along the next dimension for nodes in this layer. We then go through each of the nodes and split the corresponding data, forming a layer beneath. If `leftdata` is empty, then the corresponding child node is not formed, and similarly if `rightdata` is empty:

Algorithm 3 KD tree construction

```
function [node A]=KDTreeMake(x)
[D N]=size(x);
A=sparse(N,N);
node(1).data=x;
for level=1:1+floor(log2(N)) % generate the binary tree:
    split_dimension=rem(level-1,D)+1; % cycle over split dimensions
    if level>1
        pa=layer{level-1}; % parents
        ch=children(A,pa); % children
    else
        ch=1;
    end
    idx=max(ch);
    layer{level}=ch;
    for i=ch
        [node(i).x leftdata rightdata]=splitdata(node(i).data,split_dimension);
        node(i).split_dimension=split_dimension;
        if ~isempty(leftdata); idx=idx+1; node(idx).data=leftdata; A(i,idx)=1; end
        if ~isempty(rightdata); idx=idx+1; node(idx).data=rightdata; A(i,idx)=1; end
        node(i).data=[]; % remove to save storage
    end
end
```

$$\begin{tikzcd}
 & & 1 \arrow[dl] \arrow[drr] & & \\
 & 2 \arrow[dl] \arrow[dr] & & & 3 \arrow[dl] \\
4 & & 5 & 6 &
\end{tikzcd}$$

In this way, the top layer split the data along axis 1, and then the nodes 2 and 3 in the second layer split the data along dimension 2. The nodes in the third layer don't require any more splitting since they contain single datapoints. Recursive programming is a natural way to construct the tree. Alternatively, one can avoid this by explicitly constructing the tree layer by layer. See algorithm(3) and `KDTreeMake.m` for the full algorithm details in MATLAB. We can also depict the datapoints that each node represents:

$$\begin{tikzcd}
 & & 1:(7,2) \arrow[dl] \arrow[drr] & & \\
 & 2:(5,4) \arrow[dl] \arrow[dr] & & & 3:(9,6) \arrow[dl] \\
4:(2,3) & & 5:(4,7) & 6:(8,1) &
\end{tikzcd}$$

The hierarchical partitioning of the space that this tree represents can also be visualised, see fig(8).
There are many extensions of the KD tree, for example to search for the nearest K neighbours, or search for datapoints in a particular range. Different hierarchical partitioning strategies, including non-axis aligned partitions can also be considered. See [3] for further discussion.

Complexity

Building a KD tree has $O(N \log N)$ time complexity and $O(KN)$ space complexity.

3.3 Nearest Neighbour search

To search we can make recursive use of our simple observation in section(3.1). For a query point $\mathbf{q}$ we first find the leaf node of the tree by traversing the tree from the root, and seeing if the corresponding components of $\mathbf{q}$ is 'to the left' or 'to the right' of the current tree node. For the above tree, for the query $\mathbf{q} = (9, 2)$, we would first consider the value 9 (since the first layer splits along dimension 1). Since 9 is 'to

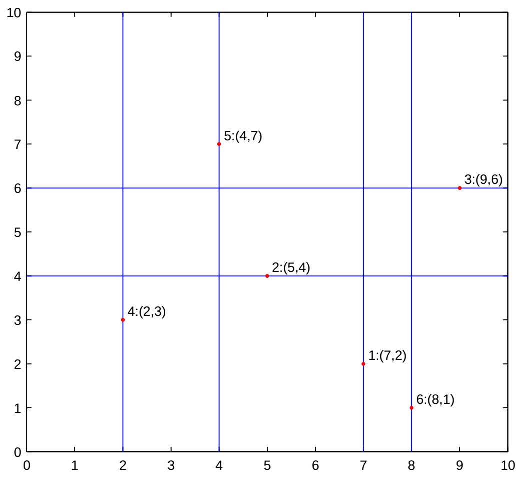


Figure 8: KD tree. The data is plotted along with its node index, and corresponding split dimension (either horiztonal or vertical). The KD tree partitions the space into hyperrectangles.

the right' of 7 (the first dimension of node 1), we go now to node 3. In this case, there is a unique child of node 3, so we continue to the leaf, node 6.

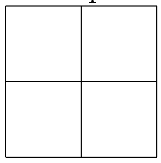
Node 6 now represents our first guess for the nearest neighbour. This has distance $(9-8)^2+(2-1)^2 = 2 \equiv \delta^2$ from the query. We set this to our current best guess of the nearest neighbour and corresponding distance. We then go up the tree, to the parent, node 3. We check if this is a better neighbour. It has distance $(9-9)^2 + (2-6)^2 = 16$, so this is worse. Since there are no other children to consider, we move up another level, this time to node 1. This has distance $(9-7)^2 + (2-2)^2 = 4$, which is also worse than our current best. We now need to consider if there are any nodes in the other branch of the tree containing nodes 2,4,5, that could be better than our current best. We know that for all nodes 2,4,5 they have first dimensions with value less than 7. We can then check if the corresponding datapoints are necessarily further than our current best guess by checking if $(v-q_1)^2 > \delta^2$, namely if $(7-9)^2 > 2$. Since this is true, it must be that all the points in nodes 2,4,5 are further away from the query than our current best guess. At this point the algorithm terminates, having examined only 3 datapoints in which to find the nearest neighbour, rather than all 6. The full procedure is given in MATLAB in algorithm(4) `KDTreeNN.m` (again using non-recursive programming). See also `demoKDTree.m` for a demonstration.

Complexity

Searching a KD tree has in the worst case $O(N)$ time complexity. For 'dense' and non-pathological data, typically the algorithm is much more efficient and $O(\log N)$ time complexity can be achieved.

4 Curse of Dimensionality

Whilst KD-trees and Orchard's approach can work well when the data is reasonably uniformly distributed over the space, their efficacy can reduce dramatically when the space is not well covered by the data.
If we split each of the D dimensions into 2 parts, for example in 2 dimensions:



we will have 2^D partitions of the data. For data to be uniformly distributed, we need a dataset to occupy each of these partitions. This means that we need at least 2^D datapoints in order to begin to see reasonable coverage of the space. Without this exponentially large number of datapoints, most of these partitions will be empty, with the effectively small number of datapoints very sparsely scattered throughout the space. In this case there is little speed up that can be expected based on either triangle or KD tree methods.

It is also instructive to understand that in high dimensions, two datapoints will typically be far apart. To

Algorithm 4 KD tree nearest neighbour search

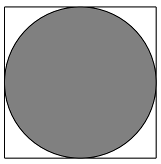
```
function [bestx bestdist]=KDTreeNN(q,x)
[node A]=KDTreeMake(x);
bestdist=realmax; N=size(x,2); tested=false(1,N);
treenode=1; % start at the top of the tree
for loop=1:N % maximum possible number of loops
    % assign query point to a leaf node (in the remaining tree):
    for level=1:1+floor(log2(N))
        ch=children(A,treenode);
        if isempty(ch); break; end % hit a leaf
        if length(ch)==1
            treenode=ch(1);
        else
            if q(node(treenode).split_dimension)<node(treenode).x(node(treenode).split_dimension);
                treenode=ch(1);
            else
                treenode=ch(2);
            end
        end
    end

    % check if leaf is closer than current best:
    if ~tested(treenode)
        testx=node(treenode).x; dist=sum((q-testx).^2);
        tested(treenode)=true;
        if dist<bestdist; bestdist=dist; bestx=testx; end
    end

    parentnode=parents(A,treenode);
    if isempty(parentnode); break; end % finished searching all nodes
    A(parentnode,treenode)=0; % remove child from tree to stop searching

    % first check if this is a better node:
    if ~tested(parentnode)
        testx=node(parentnode).x; dist=sum((q-testx).^2);
        if dist<bestdist; bestdist=dist; bestx=testx; end
        tested(parentnode)=true;
    end
    % see if could be points closer on the other branch:
    if (node(parentnode).x(node(parentnode).split_dimension) ...
     - q(node(parentnode).split_dimension))^2>bestdist
        A(parentnode,children(A,parentnode))=0; % if not then remove this branch
    end
    treenode=parentnode; % move up to parent on tree
end
```

see this, consider the volume of a hypershere of radius r in D dimensions. This is given by the expression

$$V = \frac{\pi^{D/2} r^D}{(D/2)!} \tag{25}$$

The fraction of volume that a unit hypersphere occupies when inscribed by a unit hypercube is

$$\frac{\pi^{D/2}}{2^D (D/2)!} \tag{26}$$



This value drops exponentially quickly with the dimension of the space D, meaning that nearly all points lie in the 'corners' of the hypercube. Hence two randomly chosen points will typically be in different corners and far apart – nearest neighbours are likely to be a long distance away. The triangle and KD tree approaches are effective in cases where data is locally concentrated, a situation highly unlikely to occur in high dimensional data.

References

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

[2] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In *In Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*. MIT Press, 2006.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry (Algorithms and Applications)*. Springer, 1998.

[4] C. Elkan. Using the Triangle Inequality to Accelerate k-Means. In T. Fawcett and N. Mishra, editors, *International Conference on Machine Learning*, pages 147–153, Menlo Park, CA, 2003. AAAI Press.

[5] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. 15(1):9–17, 1994.

[6] M. T. Orchard. A fast nearest-neighbor search algorithm. *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4:2297—3000, 1991.

[7] E. Vidal. An algorithm for finnding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145–157, 1986.