Nearest Neighbor Search in Hilbert Space

Neel V. Rakholia,[1*] William B. March,[2] George Biros[2]

[1]Department of Applied Mathematics and Applied Physics,
Columbia University, New York, NY 10027, USA
[2]Institute for Computational Engineering and Science,
University of Texas at Austin, Austin, TX 78705, USA

*To whom correspondence should be addressed; E-mail: nvr2105@columbia.edu

Abstract. Significant work has been done on addressing the problem of nearest neighbor (NN) search in Euclidean Space. Notable is the wealth of literature on search techniques involving the use of trees. Kd-trees and Ball trees are among the more commonly used data structures to speed up NN search. Surprisingly little work however has been done on using trees to find nearest neighbors in general metric spaces.

In this paper we undertake a study of vantage point trees (VP-trees), and analyze their effectiveness in finding NN for kernel based distance metrics. We also propose a new algorithm for searching VP-trees: an approximate random tree search algorithm. Previous work on VP-trees has focused on a priority queue based search, which is not very effective for even moderately high dimensional data. Our approach is unique in this regard.

For an RBF-kernel based distance metric, 4,500,000 training points and 18 features, NN search on 500,000 query points using the random trees search method yielded 90 percent accuracy with only about 1 percent of total distance evaluations, an improvement of about 2 orders of magnitude.

Keywords. Nearest Neighbor Algorithms, Tree Codes, Metric Spaces, Data Analysis, VP-Trees, Machine Learning

1 Introduction

Nearest Neighbors and Metric Spaces The nearest neighbor problem refers to finding the set of points P_c in a database of points D that are closest to a query point q. The notion of close and correspondingly distance can be fairly arbitrary as long as it follows the following properties. For a space to be metric, these properties must hold true. [3]

1. Reflectivity: $d(a, a) = 0$
2. Symmetry: $d(a, b) = d(b, a)$

3. Non-Negativity: $d(a, b) > 0$, $a \neq b$

4. Triangle Inequality: $d(a, b) \leq d(a, c) + d(b, c)$

One of the more widely used distance metrics are derived from similarity measures such a kernels. These are of immense practical importance in fields such Computer Vision and Natural Language Processing where the concept of similarity between abstract objects is of importance. A kernel $K : \mathbb{R}^d \times \mathbb{R}^d$ is a similarity function with the property that for any x and y, the distance between x and y increases, $K(x, y)$ decreases. The construction of the kernel distance subsequently involves a transformation from similarities to distances. It can be represented in the following general form. Given two "objects" A and B, and a measure of similarity between them given by $K(A, B)$, then the induced distance between A and B can be defined as the difference between the self-similarities $K(A, A) + K(B, B)$ and the cross-similarity $K(A, B)$. Additionally this distance could be normalized by taking the square root. [7]

$$d(A, B) = \sqrt{K(A, A) + K(B, B) - 2K(A, B)} \tag{1}$$

VP-trees require the use of a bounded distance metric: a metric that yields distance between $[0, 1]$. Any unbounded kernel distance metric can be scaled to be a bounded metric by the following simple transformation: [10]

$$d'(A, B) = \frac{d(A, B)}{1 + d(A, B)} \tag{2}$$

Significance The problem of finding nearest neighbors in Hilbert space[1] is a fundamental problem in many fields. Image analysis, pattern recognition, high dimensional generalized N-body problems, classification, and manifold learning are among the few areas of application. [6] In particular for algorithms like Approximate Skeletonization Kernel-Independent Treecode in High Dimensions (ASKIT) [4], the problem of finding NN in Hilbert space would increase the scope of their application.

Known Approaches and their Limitations Many of the known techniques used to find NN in Hilbert space use kernelized variants of algorithms in Euclidean space.

1. Kernel k-means tree: Kernel k-means algorithm assigns points to the closest of k centers by iteratively alternating between selecting centers and assigning points to the centers until neither the centers nor the point partitions change. The resulting structure is equivalent to a Voronoi partition of the points. Unlike k-means however, kernel distances, as defined earlies, are used to measure "closeness". [3] The points in each cluster form the nodes of the tree. Kernel k-means in then recursively applied on each node to get a tree data structure. To search the tree for the NN of a query point, the point's distance is calculated to each cluster center. The node with shortest distance is then recursively traversed to discover the NN. Calculating the cluster centers requires the knowledge of a feature map

[1] The concept of Hilbert Space, for the context of this paper, refers to an inner product space that is complete and separable with respect to the norm defined by the inner product.

however. This is often not present or cannot be calculated accurately.[2] In such cases kernel k-medoids would provide an approximation of the cluster centers.

2. Kernel k-medoids tree: Like Kernel k-means, kernel K-medoids also assigns points to the nearest cluster center. Only data points are selected as cluster points though. Consequently, no additional information is needed about the feature map. Tree construction and search can be performed using the aforementioned algorithm.

3. Backtrack search on ball tree: Ball trees are a type of metric trees. To construct a ball tree, each node's points are assigned to the closest center of the node's two children. The children are chosen to have maximum distance between them, typically using the following construction at each level of the tree. First, the centroid of the points is located, and the point with the greatest distance from this centroid is chosen as the center of the first child. Then, the second child's center is chosen to be the point farthest from the first one. [3] Backtrack on a ball tree is an exact NN search algorithm, always returning the true NN. It starts off with a depth search to reach the node closest to the given query point in Hilbert Space. The tree is then pruned to exclude nodes that do not contain NN of the query point. This approach is a slight variation on the priority queue based algorithm mentioned in [10].

The first two approaches require pairwise distance computation between points at each level of the tree to calculate the clusters. This is in $O(n^2)$ operation, making tree construction expensive. Additionally for data with a large number of features, the curse of dimensionality makes these methods less accurate. Backtrack search on moderately high dimensional data often searches the entire tree. [5]
Note: Kd-trees and other "projective trees" [3] do not work for the problem at hand because the notion of projection is not present in Hilbert space.

Our contribution VP-trees use concentric hyperspheres to partition points into a metric tree. [10]. Construction is an $O(n \cdot log(n))$ operation and search is $O(d \cdot log(n))$. In the paper, we propose a new method for searching VP-point trees:
Random trees search: This is a novel approach that we present to speed up NN search without sacrificing the accuracy. We extrapolate the work done on using randomized kd-trees for NN search in high dimensions [2] to the concept of VP-trees. For this algorithm multiple random VP-trees are iterated over to find the NN for a query point. With each iteration better approximate NN are found.
The following table presents the results of running this algorithm on the SUSY data set. [9]

[2]The feature space for an RBF-kernel in infinite dimensional. Hence the feature map for a vector would also be infinite dimensional.

Property	Value
Number of points	4,500,000
Number of query points	500,000
Number of features	18
Number of nearest neighbors	10
RBF kernel bandwidth	0.15
Max. points per node	8,192
Tree depth	12
Time spent running linear search	114216 s
Time spent running random tree search	**1943 s**
Accuracy	**0.935**
Number of iterations to get 90% accuracy	3
Fraction of total distance evaluations	**0.011**
Average ratio of distance	1.003

Table 1: Running random tree search on SUSY

The proposed method is not only fast, but also accurate. The following section provides additional information about constructing VP-trees, using random trees to search for NN, and experiments to evaluate the performance of this approach.

2 Construction of Vantage Point Trees

Geometry A VP-tree, like a kd-tree, geometrically divides the metric space. The difference arises in nature of space partition. A kd-tree is built by recursively bisecting the database using single coordinate position cuts. For a given coordinate, the database is cut at the median of the distribution generated by projection onto that coordinate. An optimized kd-tree results by choosing the cutting coordinate to be that whose distribution exhibits the most spread. [10] VP-

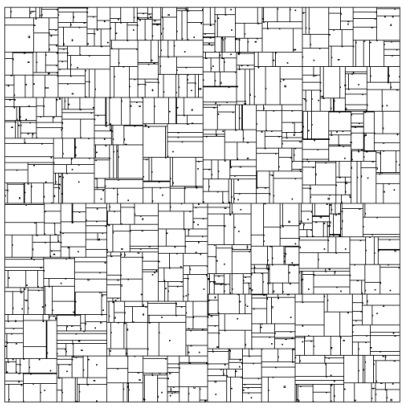

Figure 1: Space partitioning using coordinate split [10]

trees on the other hand, uses distance from a selected vantage point as the criterion for partition. Again, the notion of distance here is arbitrary. Points near the selected point are assigned to the

left child, whereas points further away are assigned to the right child. [10] Proceeding recursively a binary tree is formed. This results in a spherical partition of space.

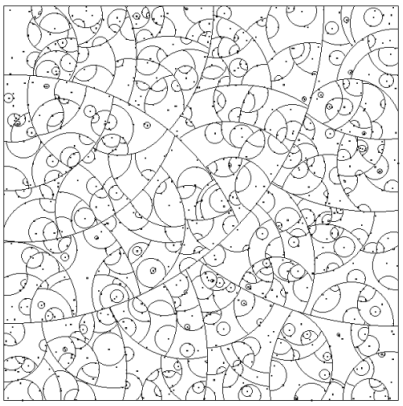

Figure 2: Space partitioning using vantage points [10]

Conventional construction algorithm The simplest algorithm for constructing a VP-tree follows a recursive approach. The root of the tree corresponds to the entire metric space. A function then selects, based on some pre-defined criteria, a vantage point for the root. This "distinguished" point is then used to partition the space into left and right subspaces as previously mentioned. This is repeated on the left and right child to get a binary tree. In the original algorithm outlined in [10] the function that selects a vantage point, basis this selection on the variance in distance. The point that maximizes the variance in distance between itself and other points in the node is selected as the vantage point.

Algorithm 1 VP-tree construction [10]

procedure BSTTREEVP(P)
  if $P = \phi$ **then return** ϕ
  end if
  $new(node)$
  $node.vp \leftarrow selectvp(P)$
  $node.center \leftarrow median_{(p\ \epsilon\ P)}\ d(vp, p)$
  $ldata \leftarrow \{p\ \epsilon\ P \mid d(vp, p) < node.center\}$
  $rdata \leftarrow \{p\ \epsilon\ P \mid d(vp, p) \geq node.center\}$
  $node.left \leftarrow bsttreevp(ldata)$
  $node.right \leftarrow bsttreevp(rdata)$
  return node
end procedure
procedure SELECTVP(P)
  $samp \leftarrow sample(P)$
  $bestvar \leftarrow 0$

```
    for i ϵ samp do
        lsamp ← sample(P)
        dist ← {d(i, j) | j ϵ lsamp}
        var ← variance(dist)
        if var > bestvar then
            bestvar ← var
            vp ← i
        end if
    end for
    return vp
end procedure
```

Introducing randomness Depending on the number points in the samples selected, the trees constructed by Algorithm 1 may or may not be deterministic. Consider the extreme case in which we evaluate all pair wise distances. In this case, we would almost always construct the same tree. For data with a large number of features and for arbitrary metric spaces, such a construction would not always yield the correct NN. Traversing to the appropriate node gives only a fraction of the NN. Despite all the effort put into constructing an optimal tree, we are limited by the constraints of dimensionality. We propose two modifications to the aforementioned VP-tree construction to make it more viable.

1. Truncation: Instead of subdividing the metric space continuously, we divide it until the number of points in a node falls below a threshold value. Alternatively, we impose a limit on the depth of the tree. For large data sets, this would significantly reduce the cost of construction.

2. Randomness: We seek to introduce randomness into the process of VP-tree construction. It is known that random selection of vantage points is only slightly less optimal than the procedure described in Algorithm 1. [10] By constructing multiple random trees, we hope to partially overcome the constraints that dimensionality puts on NN search using VP-trees.

Algorithm 2 Random VP-tree construction

```
MAXLVL ← ML
MAXPNTS ← MP
lvl ← 0
procedure RANDBSTTREEVP(P, lvl)
    if P = ϕ or |P| < MAXPNTS or lvl > MAXLVL then
        node.data ← P
        return node
    end if
    new(node)
```

10: $node.vp \leftarrow randselectvp(P)$
11: $node.center \leftarrow median_{(p\ \epsilon\ P)}\ d(vp, p)$
12: $ldata \leftarrow \{p\ \epsilon\ P \mid d(vp, p) < node.center\}$
13: $rdata \leftarrow \{p\ \epsilon\ P \mid d(vp, p) \geq node.center\}$
14: $node.left \leftarrow randbsttreevp(ldata, lvl + 1)$
15: $node.right \leftarrow randbsttreevp(rdata, lvl + 1)$
16: **return** node
17: **end procedure**
18: **procedure** RANDSELECTVP(P)
19: $vp \leftarrow random(P)$
20: **return** vp
21: **end procedure**

3 Searching Random VP-Trees

Psuedocode The algorithm that we propose is based on the construction of a number of random VP-trees and searching each tree iteratively to find the NN. It is an extension of the approach mentioned in [2] and [6]. Presented below is a brief outline of the algorithm and the psuedocode.

1. Construct a random VP-tree using given points
2. Perform depth first search on the tree to get a set of NN candidates for a query.
3. Linearly search the the candidates for the best NN.
4. If there exists a list of other NN candidates from the previous iteration, merge the lists to make a note of best NN and store them.
5. Go to step 1 and repeat until the desired accuracy is achieved.

Algorithm 3 Random VP-tree search

1: $MAXLVL \leftarrow ML$
2: $MAXPNTS \leftarrow MP$
3: $lvl \leftarrow 0$
4: $prevnn \leftarrow \phi$
5: $NTREE \leftarrow NT$
6: **procedure** RANDVPSEARCH(points, query, maxnn)
7: **for** $i \leftarrow 1 : NTREE$ **do**
8: $tree \leftarrow randbsttreevp(points, lvl)$
9: $nn \leftarrow search(tree, query, maxnn)$
10: $bestnn \leftarrow merge(nn, prevnn)$
11: $prevnn \leftarrow nn$
12: **end for**

```
    return node
end procedure
procedure SEARCH(tree, query, maxnn)
    if isleaf(tree) then
        nn ← linearnnsearch(tree.data, query, maxnn)
        return nn
    end if
    dist = d(query, tree.vp)
    if dist < tree.center then
        return search(tree.left, query, maxnn)
    else
        return search(tree.right, query, maxnn)
    end if
end procedure
```

Merging Notice the merge function defined in algorithm 3 is fairly abstract. This is because there are several different ways in which knowledge about previous NN can be incorporated into the NN search for the current iteration. We consider two separate techniques.
Problem: Given n queries $q_1, q_2, ..., q_n$ and $nnmax$ neighbors $k_1^{i-1}, k_2^{i-1}, ...k_{nnmax}^{i-1}$ for each query from the previous iteration, we want to find the best $nnmax$ NN for the current iteration, $k_1^i, k_2^i, ...k_{nnmax}^i$ for each query.

1. Horizontal Merge: For every iteration, we have a transient set of NN that is given by line 8 in Algorithm 3. Therefore there are two matrices:

$$nntrans = \begin{bmatrix} nn_{1,1} & nn_{1,2} & \dots & nn_{1,nnmax} \\ nn_{2,1} & nn_{2,2} & \dots & nn_{2,nnmax} \\ \vdots & \vdots & \ddots & \vdots \\ nn_{n,1} & nn_{n,2} & \dots & nn_{n,nnmax} \end{bmatrix} k_{i-1} = \begin{bmatrix} k_{1,1}^{i-1} & k_{1,2}^{i-1} & \dots & k_{1,nnmax}^{i-1} \\ k_{2,1}^{i-1} & k_{2,2}^{i-1} & \dots & k_{2,nnmax}^{i-1} \\ \vdots & \vdots & \ddots & \vdots \\ k_{n,1}^{i-1} & k_{n,2}^{i-1} & \dots & k_{n,nnmax}^{i-1} \end{bmatrix} \quad (3)$$

   For horizontal merge, we linearly search for $nnmax$ NN for each query q_i by searching over the i^{th} row in $nntrans$ and k_{i-1}. Therefore, the optimal set of NN after iteration i is given by:

$$k_i = \begin{bmatrix} linearnnsearch([nn_{1,1}, \dots, nn_{1,nnmax}, k_{1,1}^{i-1}, \dots, k_{1,nnmax}^{i-1}], q_1, nnmax) \\ linearnnsearch([nn_{2,1}, \dots, nn_{2,nnmax}, k_{2,1}^{i-1}, \dots, k_{2,nnmax}^{i-1}], q_2, nnmax) \\ \vdots \\ linearnnsearch([nn_{n,1}, \dots, nn_{n,nnmax}, k_{n,1}^{i-1}, \dots, k_{n,nnmax}^{i-1}], q_n, nnmax) \end{bmatrix} \quad (4)$$

$$k_i = \begin{bmatrix} k_{1,1}^i & k_{1,2}^i & \dots & k_{1,nnmax}^i \\ k_{2,1}^i & k_{2,2}^i & \dots & k_{2,nnmax}^i \\ \vdots & \vdots & \ddots & \vdots \\ k_{n,1}^i & k_{n,2}^i & \dots & k_{n,nnmax}^i \end{bmatrix} \quad (5)$$

This method often requires less total distance evaluations to converge to a particular accuracy valuation. However, it requires more number of iterations and correspondingly more tree constructions. It is most suited for problems for which tree construction is less expensive.

2. Proximity Merge: In only considering the prior NN for a query q_i to compute the NN for the current iteration, we are inherently losing some information present. Consider depth first search on a random VP-tree for three query points: q_i, q_j, q_k. Suppose the NN for all of them come from the same leaf of the tree. Then in searching for NN of q_i, we should not only look at prior neighbors of q_i, but also previous neighbors of q_j, q_k. We use the knowledge about proximity of different query points to optimize the search. Let Q_i be the set of query points close to q_i. If K_i is a cumulative list of all previous NN for query points in Q_i, k_i can be computed as follows:

$$k_i = \begin{bmatrix} linearnnsearch([nn_{1,1}, \ldots, nn_{1,nnmax}, k_{1,1}^{i-1}, \ldots, k_{1,nnmax}^{i-1}, K_1], q_1, nnmax) \\ linearnnsearch([nn_{2,1}, \ldots, nn_{2,nnmax}, k_{2,1}^{i-1}, \ldots, k_{2,nnmax}^{i-1}, K_2], q_2, nnmax) \\ \vdots \\ linearnnsearch([nn_{n,1}, \ldots, nn_{n,nnmax}, k_{n,1}^{i-1}, \ldots, k_{n,nnmax}^{i-1}, K_n], q_n, nnmax) \end{bmatrix} \tag{6}$$

$$k_i = \begin{bmatrix} k_{1,1}^i & k_{1,2}^i & \cdots & k_{1,nnmax}^i \\ k_{2,1}^i & k_{2,2}^i & \cdots & k_{2,nnmax}^i \\ \vdots & \vdots & \ddots & \vdots \\ k_{n,1}^i & k_{n,2}^i & \cdots & k_{n,nnmax}^i \end{bmatrix} \tag{7}$$

This algorithm is useful when tree construction is expensive. Although we end up evaluating more distances, the number of iterations taken is less. In this paper, we predominantly adopt this approach.

4 Results

RBF kernel In sections 2 and 3, we have developed a general algorithm for constructing and searching random VP-trees, ignoring specifics about metric spaces. To test the method's effectiveness, we use a RBF-kernel based distance metric. Because of its ubiquitous use, the performance of the algorithm on this distance metric would be a useful parameter for measuring its success. A Gaussian radius basis function (RBF) is defined as follows [8]:

$$k(x, x') = exp\left(\frac{||x - x'||_2^2}{2\sigma^2}\right) \tag{8}$$

Using equations 1 and 2, we can transform this function so that it is a valid bounded distance metric.

$$d_{RBF}(x, x') = \frac{\sqrt{2 - 2 \cdot exp\left(\frac{||x-x'||_2^2}{2\sigma^2}\right)}}{1 + \sqrt{2 - 2 \cdot exp\left(\frac{||x-x'||_2^2}{2\sigma^2}\right)}} \tag{9}$$

This is the distance measure used for testing the algorithm on different data sets. Depending on the nature of the data set, different values for the bandwidth parameter, sigma, would have to be used so that the data spread in Hilbert space is optimal.

Measures for success We use three different parameters for evaluating how well random VP-trees behave on different data sets.
Accuracy: Given a query point q_i, its true NN[3] $k_i = \{k_1, k_2, ..., k_{nnmax}\}$, and its approximate NN $m_i = \{m_1, m_2, ..., m_{nnmax}\}$, the accuracy of the method for the single query point is given by:

$$acc_i = \frac{|k_i \cap m_i|}{|k_i|} \tag{10}$$

The overall accuracy for n query points is the average of the accuracy for all query points:

$$acc = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(\frac{|k_i \cap m_i|}{|k_i|} \right) \tag{11}$$

Fraction of total distance evaluations: Given m database points and n query points, the total number of distance evaluations calculated by the linear brute force method is $m \times n$. For the random VP-tree algorithm suppose we construct a total of $ntree$ trees and this requires t distance evaluations. Additionally, assume that searching all the trees requires calculating s distances. The fraction of total distance evaluations is then given by:

$$frac = \frac{t + s}{mn} \tag{12}$$

Average ratio of distance: Given a query point q_i, its true NN $k_i = \{k_1, k_2, ..., k_{nnmax}\}$, and its approximate NN $m_i = \{m_1, m_2, ..., m_{nnmax}\}$, the average ratio of distance between its approximate neighbors and true neighbors is calculated as follows:

$$avgd_i = \frac{1}{nnmax} \cdot \sum_{j=1}^{nnmax} \left(\frac{d_{RBF}(q_i, m_{i,j})}{d_{RBF}(q, k_{i,j})} \right) \tag{13}$$

$avgd_i$ averaged over all n queries gives the desired value:

$$avgd = \frac{1}{n} \cdot \sum_{i=1}^{n} avgd_i \tag{14}$$

Tests on cover type dataset The forest cover type dataset is a geological dataset available on University of California, Irvine (UCI) Machine Learning Repository. [1]. We want to measure how NN search for RBF kernel distances compares to NN search for Euclidean distances. To make this comparison, we first sample a few points from the cover type data set. We then run a state-of-the-art kd-tree search (refer to random projection in [6]) on the sample followed by

[3]These are calculated by the brute force search algorithm

the proposed random VP-tree based search. For VP-trees a RBF-kernel distance metric is used with a specified value of sigma (0.22 here). Kd-trees use the L2 norm to evaluate distance pairs. Presented below is a brief summary of the test data set and the trees used.

Property	Value
Number of points	5,000
Number of query points	400
Number of features	54
Number of nearest neighbors	100
Max. points per node	256
Tree depth	12

Table 2: Running random tree search on Covtype

Both methods are iterative in nature and both find a greater percentage of NN with each iteration. Correspondingly accuracy, as defined in equation 11, would be a good measure of comparison. Following is the graph of accuracy of both the methods as a function of iteration number.

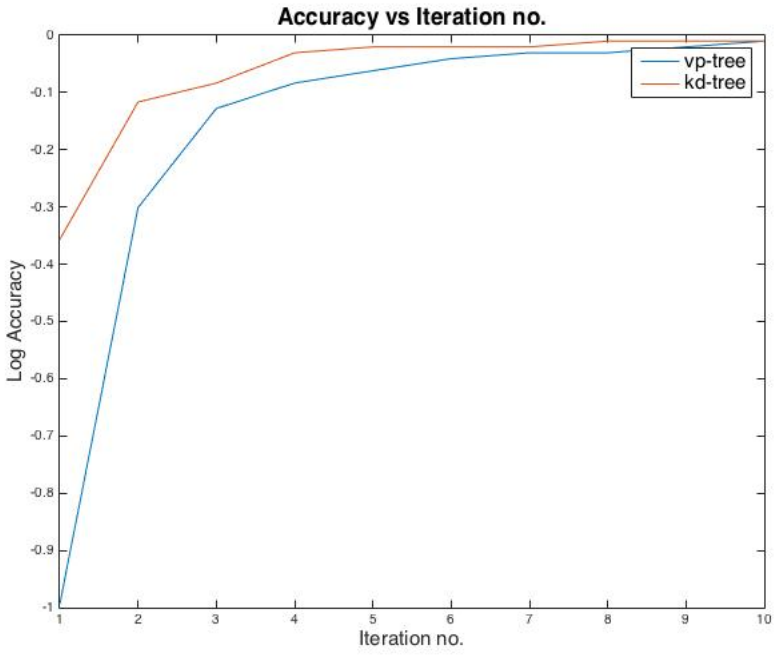

Figure 3: Accuracy vs Iteration no. for kd-trees and VP-trees

It is evident that kd-trees for Euclidean distances converge to a a larger accuracy value faster for the cover type dataset. However, VP-trees are not significantly worse. In fact, for a sufficiently large iteration no., both methods give the same accuracy value. Consider the values after 15 iterations for both the methods.

Property	Value
Number of iterations	15
Accuracy	**0.997**
Time taken to run	45 s

Table 3: Running kd-tree search on Covtype

Property	Value
Number of iterations	15
RBF kernel bandwidth	0.22
Accuracy	**0.993**
Fraction of total distance evaluations	2.11
Average ratio of distance	1.00
Time taken to run	1.23 s

Table 4: Running VP-tree search on Covtype

While the absence of a vector space is a constraint on the convergence of random VP-tree based NN search, it is not significantly worse. Runs on SUSY dataset show similar trends.

Tests on SUSY dataset SUSY is a physical data set also available on University of California, Irvine (UCI) Machine Learning Repository. [9] A similar set of runs as before, but using a much larger set of points helps establish the observations stated previously. Consider the following properties of the points and trees.

Property	Value
Number of points	500,000
Number of query points	100
Number of features	18
Number of nearest neighbors	100
Max. points per node	2,048
Tree depth	12

Table 5: Running random tree search on SUSY

Accuracy values for different iteration show slightly exaggerated differences in convergence rates compared to those shown in figure 3.

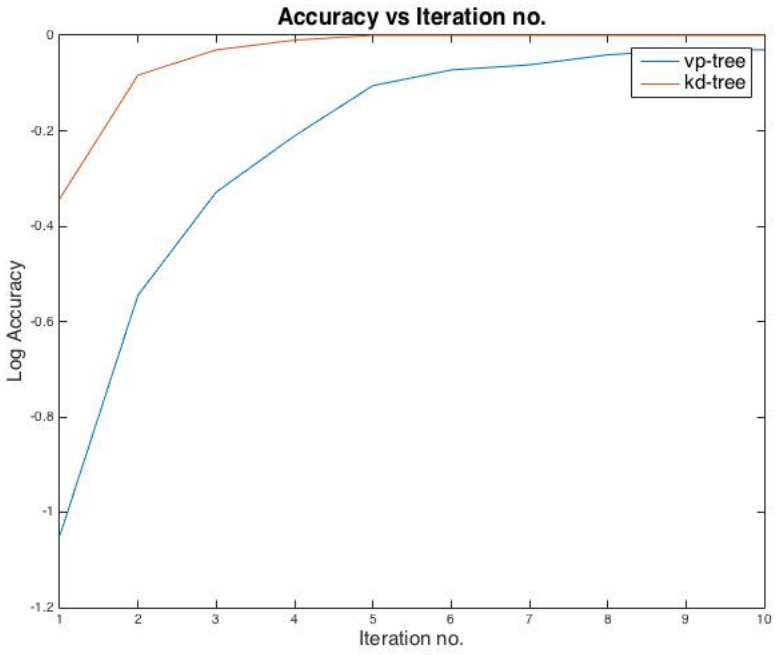

Figure 4: Accuracy vs Iteration no. for kd-trees and VP-trees

This disparity in convergence rates though is significant only if a large accuracy value is desired. Random VP-trees reach 90 percent accuracy is 5 iterations or less, making them viable for applications where there is room for some error. Additionally, for a sufficiently large iteration no., the accuracy values do converge to similar values.

Property	Value
Number of iterations	20
Accuracy	**1.00**
Time taken to run	471 s

Table 6: Running kd-tree search on SUSY

Property	Value
Number of iterations	20
RBF kernel bandwidth	0.15
Accuracy	**0.994**
Fraction of total distance evaluations	0.30
Average ratio of distance	1.00
Time taken to run	40 s

Table 7: Running VP-tree search on SUSY

Tests on large datasets So far VP-trees have been tested on a small number of data points. In evaluating their usefulness, it would be worthwhile to note how they scale with data volume. Presented below is a summary of runs on a large sample of points from cover type and SUSY.

Property	Value
Number of points	500,000
Number of query points	50,000
Number of features	54
Number of nearest neighbors	1000
RBF kernel bandwidth	0.22
Max. points per node	2,048
Tree depth	12
Time spent running linear search	1553 s
Time spent running random tree search	**434 s**
Accuracy	**0.934**
Number of iterations to get 90% accuracy	3
Fraction of total distance evaluations	**0.11**
Average ratio of distance	1.004

Table 8: Running random tree search on cover type

Property	Value
Number of points	4,500,000
Number of query points	500,000
Number of features	18
Number of nearest neighbors	10
RBF kernel bandwidth	0.15
Max. points per node	8,192
Tree depth	12
Time spent running linear search	114216 s
Time spent running random tree search	**1943 s**
Accuracy	**0.935**
Number of iterations to get 90% accuracy	3
Fraction of total distance evaluations	**0.011**
Average ratio of distance	1.003

Table 9: Running random VP-tree search on SUSY

Notice that VP-trees, for both datasets, require less time and computational resources than a brute force search. For the larger data set SUSY, we achieve a speed up of almost two orders of magnitude: the time required is about a hundred times less and the number of distance evaluations is also a reduced by a factor of 100. Similarly for cover type we achieve a four fold reduction in time and ten fold decrease in the number of distance evaluations. There is some evidence that increasing the number of points makes the use of VP-trees even more efficient. [3]

5 Conclusion

Summary In this paper we have have explored VP-trees as instruments for speeding up NN search in Hilbert Space. In particular, we have applied the concept of random tree constructions to VP-trees for RBF-kernel distance metric. We have discovered that while random VP-trees do not converge as fast as random kd-trees, they are not significantly worse. On large data sets such as cover type and SUSY, they provide considerable speed up.

Limitations and future work Despite tangible results, there is considerable potential for improvement and exploration in the algorithms presented in the paper. All the results in mentioned use an RBF-kernel distance metric. For this metric space, VP-trees behave well. However, they may not work as efficiently for other kernel distance functions. A wide array of kernels would have to studied to truly evaluate the effectiveness of random VP-trees in NN search in Hilbert Space. Additionally, there is scope for implementing a parallel version of algorithm 3. This might reduce the time taken for tree construction.

References

[1] Jock A. Blackard. UCI machine learning repository, 1998.

[2] Sanjoy Dasgupta and Kaushik Sinha. Randomized partition trees for nearest neighbor search. *Algorithmica*, 72(1):237–263, 2015.

[3] Neeraj Kumar, Li Zhang, and Shree Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In David Forsyth, Philip Torr, and Andrew Zisserman, editors, *Computer Vision – ECCV 2008*, volume 5303 of *Lecture Notes in Computer Science*, pages 364–378. Springer Berlin Heidelberg, 2008.

[4] William B. March, Bo Xiao, and George Biros. ASKIT: approximate skeletonization kernel-independent treecode in high dimensions. *CoRR*, abs/1410.0260, 2014.

[5] R. B. Marimont and M. B. Shapiro. Nearest neighbour searches and the curse of dimensionality. *IMA Journal of Applied Mathematics*, 24(1):59–70, 1979.

[6] Logan Moon, Daniel Long, Shreyas Joshi, Vyomkesh Tripathi, Bo Xiao, and George Biros. Poster: parallel algorithms for clustering and nearest neighbor search problems in high dimensions. In *Conference on High Performance Computing Networking, Storage and Analysis - Companion Volume, SC 2011, Seattle, WA, USA, November 12-18, 2011*, pages 57–58, 2011.

[7] Jeff M. Phillips and Suresh Venkatasubramanian. A gentle introduction to the kernel distance. *CoRR*, abs/1103.1625, 2011.

[8] Koji Tsuda and Bernhard Schölkopf. A primer on kernel methods. In *in Kernel Methods in Computational*, pages 35–70. MIT Press, 2004.

[9] Daniel Whiteson. UCI machine learning repository, 2014.

[10] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.