# Nearest Neighbor Search

Nearest Neighbor Search is a problem in which we try to find one or many points which are near to a given point. Another form in which the problem presents itself is if two points are similar or not. Nearest Neighbor search has a wide variety of application across varied domains ranging form:

1. Data Mining
2. Machine Learning
3. Pattern recognition
4. Cluster Analysis
5. Recommendation
6. Coding theory
7. DNA Sequencing etc.

In this notebook i will discuss some high level approaches of solving this problem and some available libraries which helps us implement them.

## Complexity Metrics

While analyzing various methods of solving this problem we will also talk about **Space** and **time** complexity associated with these methods. These are the two most competing metrics we need to consider while deciding which approach to take while solving given problem. **Space** complexity will measure how space requirement increases with increase in data size and similarly **time** complexity measures the increase in the time requirements.

## List of Few Approaches to solve this problem

**Exact Methods:**

1. Linear Search(Naive Approach)
2. Spatial decomposition

**Approximate Methods**

1. Locality sensitive hashing
2. ANNOY
3. Compression/clustering based search
4. Projected radial search

*Note: The approaches are also dependent on the distance metric being used and may change from one metric to other. We will discuss high level approaches used to solve such problems.*

### 1. Linear Search:

The simplest solution to the NNS problem is to compute the distance from the query point to every other point in the database, keeping track of the "best so far". This algorithm, sometimes referred to as the naive approach, has a running time of O(dN) where N is the number of data points in training set and d refers to the dimension of each data point. There are no search data structures to maintain, so linear search has no space complexity beyond the storage of the database.

**Space Complexity:** O(dN) which is lowest
**Time Complexity:** O(dN) which is linear in N

#### Advantages

1. It does not require any search data structure to store the data.
2. It performs better for high dimensional data compared to Spatial Decomposition methods.

#### Dis-advantages

1. The search time is linear and becomes very slow for large data sets.

Find below the code for Linear Search. For better understanding refer to the kNN Notebook (https://github.com/jyotipmahes/Implementation-of-ML-algos-in-Python/blob/master/k-NN%20.ipynb)

```
In [1]: def get_neighbors_optimized(train_set, test_set, k):

            # calculate euclidean distance
            euc_distance = np.sqrt(np.sum((train_set - test_set)**2 , axis=1))
            # return the index of nearest neighbour
            return np.argsort(euc_distance)[0:k]
```

Above code helps us find k-nearest neighbors to a test_set from a given train_set. In above code we use **Euclidean distance** as the distance metric.

## 2. Spatial decomposition:

In this approach we iteratively try to divide the search space into small spaces and store this into tree form so that we do not have to search the whole data spaces to find nearest neighbors. This approach helps in reducing the time complexity to sub-linear form but does increase the space complexity a bit depending on the algorithm. For kNN problems, Scikit-Learn has 2 algorithms which we can use:

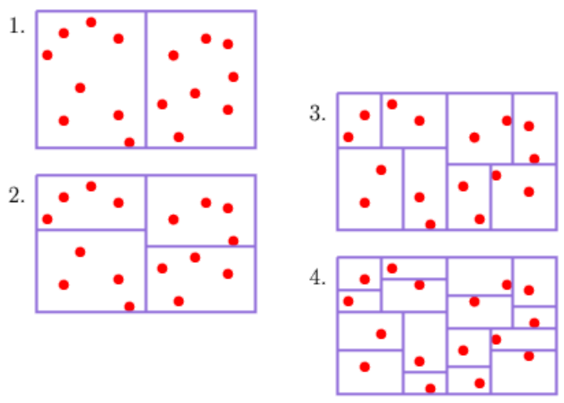
1. kd tree
2. Ball tree

**1. kd tree:**

kd tree is a binary tree which helps in recursively divide the given data space into subsections and store the data so that searching is much more optimal during prediction/searching.

**Steps of Constructing kd tree:**
The tree is formed by splitting on various features(dimensions) present in the data. Since there are many possible ways to choose the splitting planes, there are many different ways to construct kd trees. The canonical method of kd tree construction has the following steps:

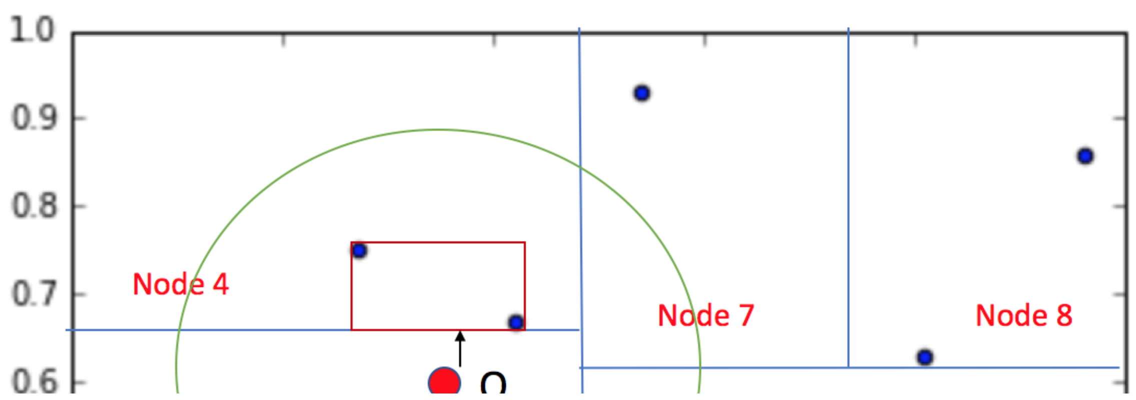
1. We alternatively select the various features/dimensions present in the data for binary split. Ex: If we have a 2 dimension data, we can start by splitting based on 'X' value, followed by 'Y' values alternatively.
2. We normally make use of the median value of the features/dimension to find the splitting point so that we end up with a balanced tree. Other strategies may also be applied here. Also, the the point corresponding to median can be treated separately or assigned to one of the partition.
3. We stop splitting based on thresholds relating to number of points in the node or depth of the tree based on our requirement. Normally we split till we have 9 data points at leaves.

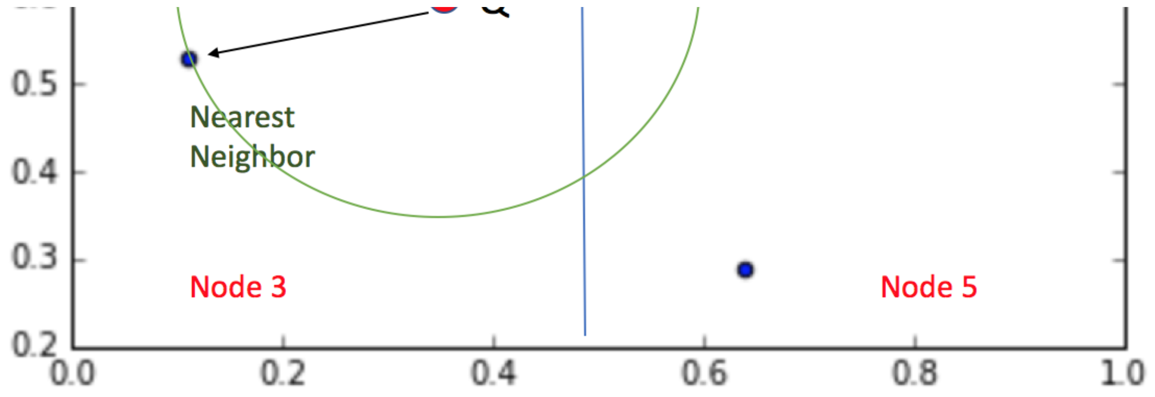


**Finding Nearest Neighbors in kd tree**

1. Let's say we are given a point Q for which we need to find the nearest neighbor, we transverse through the tree to find the leaf node which will contain point Q.
2. We now use the leaf node data points to find nearest neighbor(through linear search) and minimum distance **d** corresponding to it.
3. Now we move one level up lets say **A** in the tree as leaf node corresponding to point Q may not contain the nearest neighbor. We can verify this by finding the distance between point Q and the split boundary and see if it is greater than **d** or not. If true, we discard the other side of the tree and move a level up. If no, we use the other side leaf nodes of **A** to find the nearest neighbor to Q.

   *For detailed explanation follow this [blog (https://www.analyticsvidhya.com/blog/2017/11/information-retrieval-using-kdtree/)](https://www.analyticsvidhya.com/blog/2017/11/information-retrieval-using-kdtree/) or watch this [video (https://www.youtube.com/watch?v=E1_WCdUAtyE)](https://www.youtube.com/watch?v=E1_WCdUAtyE)*





**Space Complexity:** O(dNlogN) for pre-processing and and O(dN) for storage
**Time Complexity:** Ranges form O(dlog(N)) to O(dN). But on average O(dlog(N)). Worst case is when we need to explore all nodes

**Advantages**

1. Search time is sub-linear

**Dis-Advantages**

1. Not simple to implement
2. As dimension increases, the performance gets even worser than Linear search as there are a lot of node overlap in high dimension and lack of vectorization in distance calculation.

**As a general rule, if the dimensionality is k, the number of points in the data, N, should be N ≫ 2k. Otherwise, when k-d trees are used with high-dimensional data, most of the points in the tree will be evaluated and the efficiency is no better than exhaustive search, and other methods such as approximate nearest-neighbor are used instead.**

**2. Ball Tree:**

Another data structure to speed up discovery of neighborhood points is ball-tree data-structure. Ball tree data structure is very efficient especially in situations when number of dimensions is very large. A ball tree is also a binary tree with a hierarchical (binary) structure. To start with two clusters (each resembling a ball) are created. As it is a multidimensional space, each ball may be appropriately called a hypersphere. Any point in n-dimensional space will belong to either cluster but not to both. It will belong to the cluster from whose centroid its distance is less. If the distance of this point from the centroids of both the balls is same, it may be included in any one of the clusters. It is possible that both (virtual) hyper spheres may intersect but the points will belong to only one of the two. Next, each of the balls is again subdivided into two sub-clusters, again each resembling a ball; meaning thereby that in these sub-clusters again there are two centroids and membership of the point to a ball is decided based upon its distance from the centroid of the sub-cluster. We again sub-divide each of these sub-sub balls and so on up till certain depth.

An unclassified (target) point must fall within any one of the nested balls. Points within this nested ball are expected to be nearest to target point. Points in other nearby balls (or enveloping balls) may also be nearer to it (for example, this point may be at the boundary of one of the balls.) Nevertheless, one need not calculate the distance of this unclassified point from all the points in the n-dimensional space. This hastens up the classification process. Ball tree formation initially requires a lot of time and memory but once nested hyper-spheres are created and placed in memory discovery of nearest points becomes easier.

**Steps of Constructing Ball Tree**

1. Find a direction in the data point space which has maximum spread. We can do this in following ways:
   - a) Use PCA and find the most important component but it will be a overkill.
   - b) Choose any random point and find the farthest point from it. Now, use that point and find its farthest point. These two points give us the vector for maximum spread.
2. Find the projection of all points on this vector and find the median. Separate the data as left and right side by using 2 spheres.
3. The centroid of the left and right side data becomes of the center for the sphere and the distance of these centroids from the farthest points becomes the radius. we want the volume of sphere to be minimum.
4. Now recursively keep splitting till certain end condition is not met.

**Finding nearest neighbors**
Finding nearest neighbors is very similar to kd tree but now we have sphere instead of partition walls to calculate distance from.

**Space Complexity:** O(dN) for storage
**Time Complexity:** Ranges form O(dlog(N)) to O(dN). But on average O(dlog(N)). Worst case is when we need to explore all nodes

**Advantages**

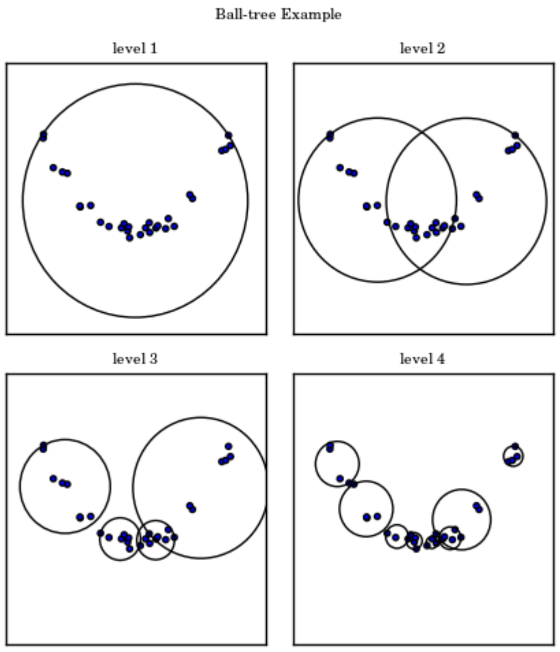
1. Search time is sub-linear
2. It works well even for high dimensions unlike kd tree

**Dis-Advantages**

1. Not simple to implement, more complex compared to kd tree and linear search
2. As dimension increases, the performance gets even worser than Linear search as there are a lot of node overlap in high dimension and lack of vectorization in distance calculation.

**Skelearn defaults**
In Sklearn, Currently, algorithm = 'auto' selects 'kd_tree' if k < N/2 and the 'effective*metric*' is in the 'VALID_METRICS' list of 'kd_tree'. It selects 'ball_tree' if k < N/2 and the 'effective*metric*' is not in the 'VALID_METRICS' list of 'kd_tree'. It selects 'brute' if k >= N/2. This choice is based on the assumption that the number of query points is at least the same order as the number of training points, and that leaf_size is close to its default value of 30.

*For detailed explanation watch this video (https://www.youtube.com/watch?v=E1_WCdUAtyE)*



## Approximate Methods (ANN):

An approximate nearest neighbor search algorithm is allowed to return points, whose distance from the query is at most c times the distance from the query to its nearest points. The appeal of this approach is that, in many cases, an approximate nearest neighbor is almost as good as the exact one. So we are compromising slight accuracy for great improvement in performance. There are a lot of ANN methods and various libraries which helps us implement them, but we will only discuss 2 such methods here and basic idea behind these algorithms. For detailed understanding various other methods follow this link (https://github.com/erikbern/ann-benchmarks).
We will discuss the following algorithms in this notebook:

1. Locality Sensitive Hashing (LSH)
2. Approximate Nearest Neighbors Oh Yeah(ANNOY)

**1. Locality Sensitive Hashing (LSH):**

LSH works on the principle that if there are two points in feature space closer to each other, they are very likely to have same hash (reduced representation of data). LSH primarily differs from conventional hashing (aka cryptographic) in the sense that cryptographic hashing tries to avoid collisions but LSH aims to maximize collisions for similar points. In cryptographic hashing a slight changes in the input can alter the hash significantly but in LSH, slight distortions would be ignored so that the main content can be identified easily. The hash collisions make it possible for similar items to have a high probability of having the same hash value. LSH helps in solving both the issue of High Dimensionality and large Linear search space by:

1. Reducing the high dimensional features to smaller dimensions while preserving the differentiability
2. Grouping similar objects (songs in this case) into same buckets with high probability

A hash function h is Locality Sensitive if for given two points a, b in a high dimensional feature space,

**Pr(h(a) == h(b)) is high if a and b are near**
**Pr(h(a) == h(b)) is low if a and b are far**

**Time complexity** to identify close objects is sub-linear

Some popular approaches to construct LSH are

1. Min-wise independent permutations (https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134)
2. Nilsimsa Hash (Anti-Spam focused) (https://wikivisually.com/wiki/Nilsimsa_Hash)
3. TLSH (For security and digital forensic applications) (https://github.com/trendmicro/tlsh/blob/master/TLSH_CTC_final.pdf)

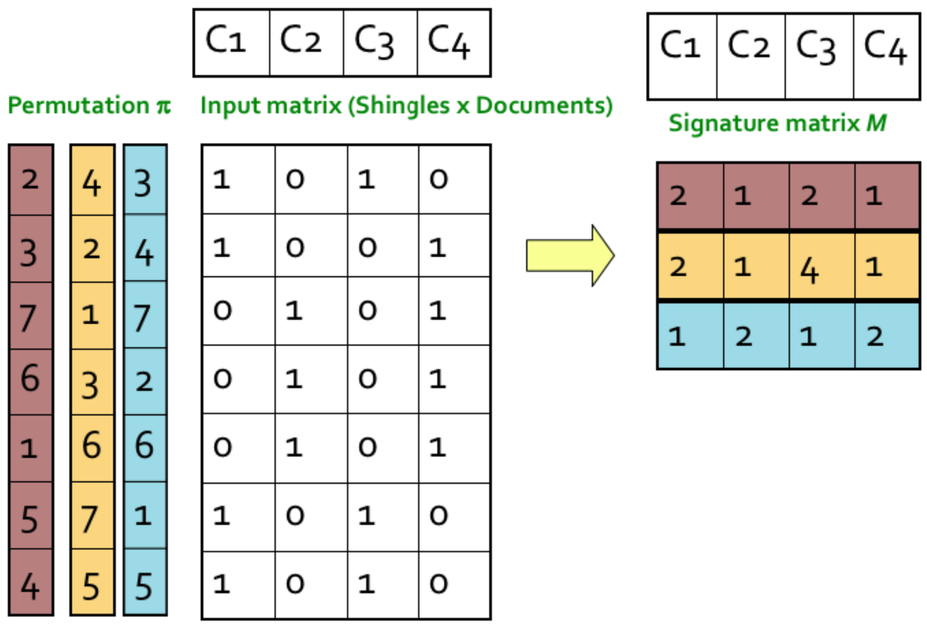
4. Random Projection aka SimHash (https://github.com/santhoshhari/Locality-Sensitive-Hashing)

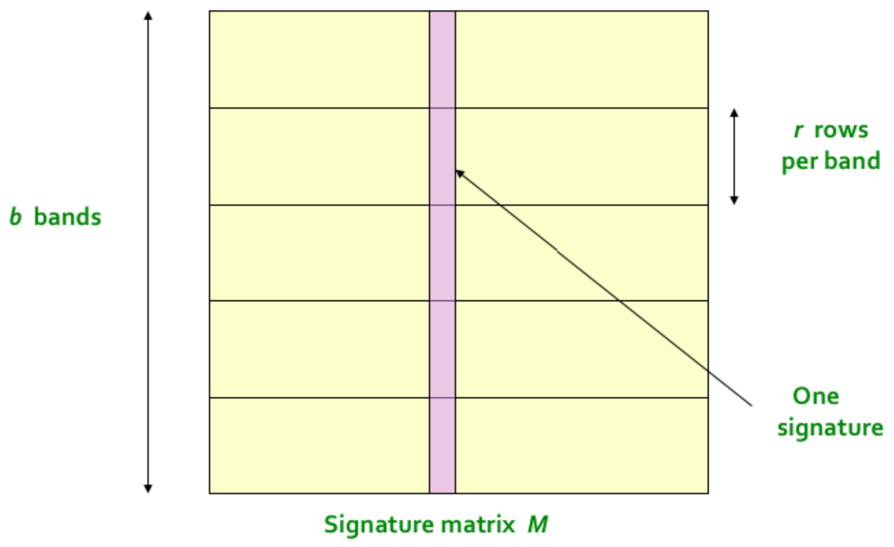
I will briefly discuss the basic idea behind these hashing techniques. For detailed understanding, please follow the links.

**1. Min-wise independent permutations:** The idea of hashing is to convert each data point to a small signature using a hashing function H. For Jaccard similarity the appropriate hashing function is min-hashing. Here is the basic steps involved in Min- Hashing if we have to compare documents:

1. Convert the documents into vectors either by one hot encoding across work corpus or build n-gram representation either at word or character levels. The words/n-grams becomes features and 1 or 0 specify if it is present or absent from a document. Hence we get a spare matrix representation for each document across the features we choose.
2. Create a matrix of the data point with Features/dimensions as rows and different data points as rows i.e features(d)*data points(N)
3. Randomly shuffle the rows of the above matrix i.e we are effectively shuffling the feature/dimension orders for the data points.
4. Now collect the minimum row index for which each columns has a value 1. We get a N dimension vector where each column has the minimum row index value which has a value 1 in the original feature*document metrics. Please refer to the link below for detailed understanding if it is not clear.
5. Repeat the shuffling k times and collect k such vectors using the same technique explained on **step 4.**
6. Now we are left with a k*N dimension matrix which is called signature matrix. So using min-hashing we have solved the problem of space complexity by eliminating the sparseness and at the same time preserving the similarity.
7. Divide the signature matrix into b bands, each band having r rows
8. For each band, hash its portion of each column to a hash table with k buckets
9. If 2 documents hash into same bucket for at least one of the hash function we can take the 2 documents as similar
10. Tune b and r to catch most similar pairs but few non similar pairs.

This may be a lot to process. Please follow this blog (https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134) for a detailed and clear understanding.





**2.Nilsimsa Hash:**

Nilsimsa is an anti-spam focused locality-sensitive hashing algorithm. The goal of Nilsimsa is to generate a hash digest of an email message such that the digests of two similar messages are similar to each other. In comparison with cryptographic hash functions such as SHA-1 or MD5, making a small modification to a document does not substantially change the resulting hash of the document. The paper suggests that the Nilsimsa satisfies three requirements:

1. The digest identifying each message should not vary significantly (sic) for changes that can be produced automatically.
2. The encoding must be robust against intentional attacks.
3. The encoding should support an extremely low risk of false positives.

Nilsimsa uses a 5-byte fixed-size sliding window that analyses the input on a byte-by-byte and produces trigrams of possible combinations of the input characters.

The trigrams map into a 256-bit array (known as the accumulator) to create the hash, and every time a given position is accessed, its value is incremented. At the end of the processing, if the values are above a certain threshold, the value is set to a 1, else it will be zero. This produces a 32-byte digest

To compare two hashes, the method checks the number of identical bits red to the the same position. This produces a score from 0 (dissimilar objects) to 128 (identical or very similar objects).

*For detailed understanding, follow the paper (https://wikivisually.com/wiki/Nilsimsa_Hash) or blog (https://asecuritysite.com/encryption/nil)

**3. TLSH.**

TLSH provides an algorithms for evaluating and comparing hash values and provide a reference to its open source code. It is very similar to Nilsimsa in its core idea but different in implementation. The TLSH digest of the byte string is evaluated as the following steps

1. Process the byte string using a sliding window of size 5 to populate an array of bucket counts
2. Calculate the quartile points, q1, q2 and q3
3. Construct the digest header values
4. Construct the digest body by processing the bucket array

The final TLSH digest constructed from the Byte string is the concatenation of:

- the hexadecimal representation of the digest header values from step 3, and
- the hexadecimal representation of the binary string from step 4.

*For a detailed explanation please refer to the paper (https://github.com/trendmicro/tlsh/blob/master/TLSH_CTC_final.pdf)

**4. Random Projection aka SimHash:** Random projection is a technique for representing high-dimensional data in low-dimensional feature space (dimensionality reduction). It gained traction for its ability to approximately preserve relations (pairwise distance or cosine similarity) in low-dimensional space while being computationally less expensive.
In this LSH implementation, we construct a table of all possible bins where each bin is made up of similar items. Each bin can be represented by a bitwise hash value, which is a number made up of a sequence of 1's and 0's (Ex: 110110, 111001). In this representation, two observations with same bitwise hash values are more likely to be similar than those with different hashes. Basic algorithm to generate a bitwise hash table is

1. Create k random vectors of length d each, where k is the size of bitwise hash value and d is the dimension of the feature vector.
2. For each random vector, compute the dot product of the random vector and the observation. If the result of the dot product is positive, assign the bit value as 1 else 0
3. Concatenate all the bit values computed for k dot products
4. Repeat the above two steps for all observations to compute hash values for all observations
5. Group observations with same hash values together to create a LSH table

In addition, because of the randomness, it is not likely that all similar items are grouped correctly. To overcome this limitaion a common practice is to create multiple hash tables and consider an observation a to be similar to b, if they are in same bin in atleast one of the tables. It is also worth noting that multiple tables generalize the high dimensional space better and amortize the contribution of bad random vectors.

*For detailed explanation follow the article (https://github.com/santhoshhari/Locality-Sensitive-Hashing)

## 2. Approximate Nearest Neighbors Oh Yeah(ANNOY):

Annoy (Approximate Nearest Neighbors Oh Yeah) is a C++ library with Python bindings to search for points in space that are close to a given query point. It also creates large read-only file-based data structures that are mmapped into memory so that many processes may share the same data. It was developed at Spotify for music recommendations. Basic understanding of how ANNOY works:

1. Data space is divided randomly on the features by creating a hyperplane. This hyperplane is chosen by sampling two points from the subset and taking the hyperplane equidistant from them. Tree splitting stops when we have some k elements are left at the leaf node.
2. This is done n_tree times so that we get a forest of trees. n_trees has to be tuned to our need, by looking at what trade off we have between precision and performance.
3. Once we have n_trees, we find all the unique nearest neighbor candidates buy using our usual process of tree traversal and pruning.
4. Once we have all the unique probable candidates collected form different trees, we do a linear search on them to find best neighbors.

*For detailed explanation follow this blog (https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html) or watch this video (https://www.youtube.com/watch?v=QkCCyLW0ehU&t=2190s)