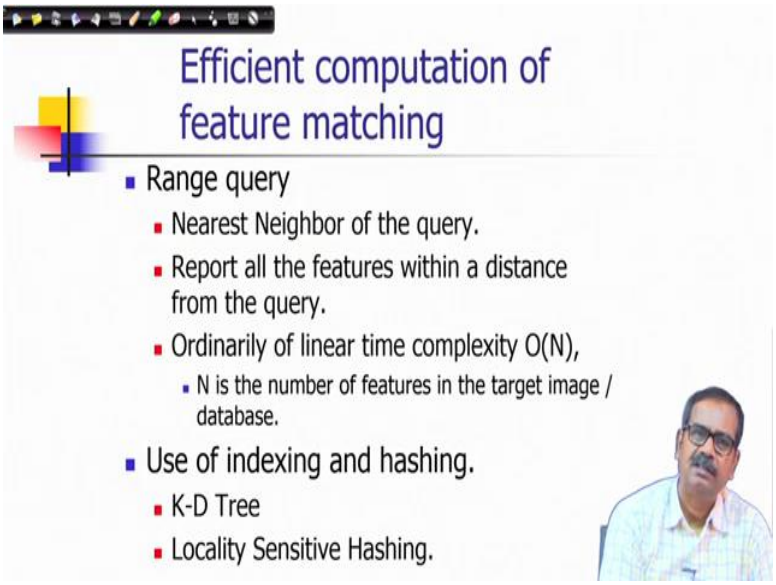
Computer Vision
Prof. Jayanta Mukhopadhyay
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 30
Feature Matching and Model Fitting Part - II

We are discussing about Feature Matching and now we will be considering that how this computation could be made more efficient.

(Refer Slide Time: 00:25)



So, we have seen in the case of feature matching, we required to find out similarities between two feature vectors or you need to compute the distances between a pair of feature vectors. And the nature of this particular computation is that if I give you a feature vector for which you have to find out its corresponding feature vector in a candidate set. It is the kind of range query which means instead of a single feature vector I may get a possible candidate's in its proximity that is the kind of answer I may like to get.

And later on I would like to find out which one of them is more closer or satisfies some other criteria by which I can declare them as the close match. Now so, this is a kind of computational problem. So, it is the nature of you know query. It is not typically I would be interested on a particular feature. I may be interested on a set of features or on a range of or an interval on which this feature vectors should like. A special case is of course know you would be only considering all the nearest neighbor of the query and otherwise you

may as I mentioned for a range query, you can report all the features within a distance from the query as we have seen also that is one of the strategy where you can use a fixed distance threshold. And report all features which are within that region as centering around that query feature vector.

But the question is that now when you are performing this computation, you have to compute the distances with all the feature vectors in that in your candidate set that, so if there are n such feature vectors you have to compute n times those distances. So, that makes your operation as a, you know linear time complexity of order. Ordinarily if I apply this brute force method of computing distances to each of them, then it would take off order in time complexity.

So, our objective is that how we can make this you know computations more efficient. As we have already; you might have known that typically this kind of search problem. If you have a set of you know candidate vectors or candidate elements, then you can keep them in some organized manner, so that your search could be made more efficient and you can perform in sub-linear time complexities in many cases. So, we call actually these operations of organizing the candidate set in a particular form that is that operation is also called as indexing with respect to databases particularly. You might have heard this term.

So, in this case n is a number of features in the target image or database, and that is why to make it efficient as I mentioned that you can use some techniques like similar to indexing. Even you can use techniques like hashing where you can store your candidate set by generating some hash values and based on that hash values, the sets having the same hash values would be kept together.

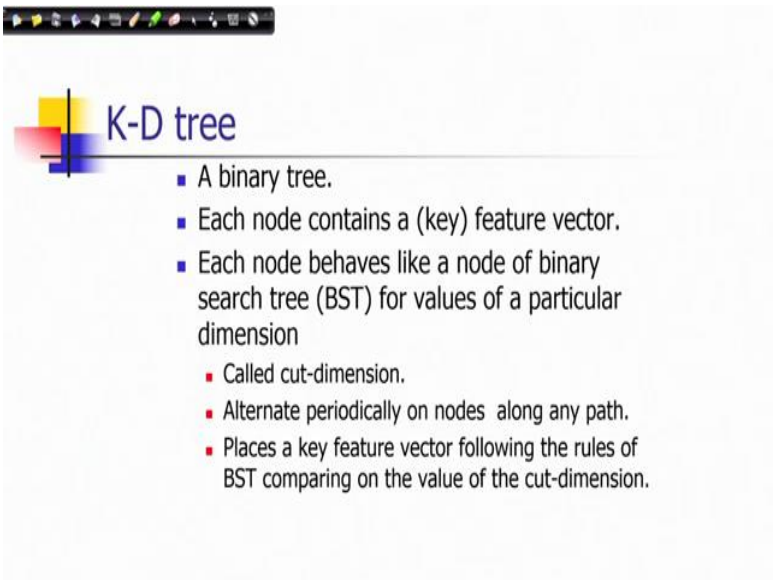
So, your idea is that for a query image it is expected that it should have a similar hash value and it should go to that particular group where your candidate set is expected to be there, and then you can compute the distances of similarities between them. In this particular context there are two techniques which are very popular for this kind of feature matching.

Because you can see that this feature vectors they are multi-dimensional representation and you can use a concept called K-Dimensional tree, K-D tree where K is the known dimension that is representing of the future vector. So if the feature vector is of two dimension we call it 2D tree, if it is three dimensional it is 3D tree, if it is one dimension

it is 1D tree which is which happens to be binary search tree. So, K-D Tree is a is the concept which is extended from binary search tree and that concept is used for searching over a multi-dimensional feature representation. And we will have some discussion, the principal of representation using K-D tree and also the search that we perform over K-D tree. And for hashing there is a technique called Locality Sensitive Hashing, as you can see that in this case the Geometry is also important.

Because know you want to get feature vectors in certain locality of your query feature vector. So, there the geometry concept of geometry space those are to be considered. In a general hashing, it is simply it is a matching of a random point my random element. So, you are randomly you are grouping different elements by the same hash value, but here we need to also consider that the in the group the corresponding candidate sets should have similar, no they should be closely spaced, they should have neighboring properties. So, to ensure that property there is a kind of; there is a technique which is very popular and this technique is locality sensitive hashing and we will also discuss about this technique.

(Refer Slide Time: 06:45)



So, let us consider first the principles behind the K-D tree, as I mentioned K-D Tree is a kind of extension of concept of binary search tree for multi-dimensional feature search. So, it is effectively it is a binary tree and every node it contains a key feature vector like what we have what we know about binary search tree. So, every node contains a key feature vector instead of a single key value for a one-dimensional tree, one dimensional no

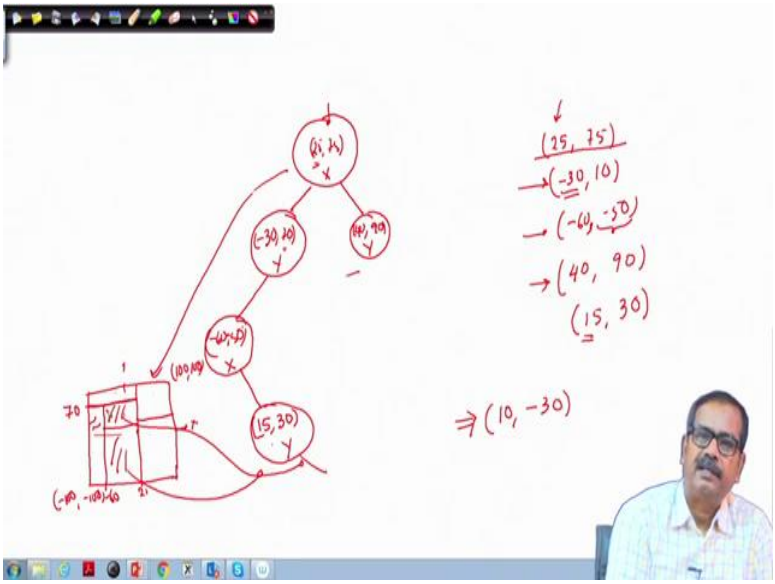
feature representations using binary search tree. And then as I mentioned each node behaves like a node of binary search tree for values of a particular dimension.

So, that is predefined for that position of the node say the, you are starting from the root say root, in the root we will be using only one dimension. Suppose you have a two dimensional feature representation. So, if I consider the dimension 1 is x and dimension 2 is y.

If I represent those dimensions, then say root will be we will consider the you know the rule that would be used for partitioning the values or partitioning the space based on only one dimension same of dimension x. I will explain it in the subsequently. So, we call this dimension as cut dimension. Just referred to this that dimension based on which this partitioning takes place and it alternate periodically on nodes along any path. So, that is how a key value is placed.

So, like binary search tree is the order of values by which a tree structure is determined, here also as and when the feature vectors are coming you are keeping them in to in this structure and when you are expanding a note, when you are placing it you need to understand, find out that you know which dimension to be used for placing it into a particular sub-tree or in a particular node. So, that is what places a key feature vector following the rules of binary search tree comparing on the value of the cut dimension. Let me explain the formation of a K-D tree.

(Refer Slide Time: 09:29)



Let us consider a set of values say we assume that we have say values say 25, 75; we assume your feature dimension is a two-dimensional feature vector. Some values let us consider say minus 30, 10 minus 60, minus 50 40, 90 take it as 15, 30. So, I am assuming that I have five feature vectors forming a candidate set. So, I have to organize this feature vectors into a K-D tree one by one I have to insert an element. So, I will start from the root and in the root I will be considering the dimension x so, this element would be inserted in the root and the dimension. So based on these value only my next so, here the x-dimension should be compared when we consider insertion of the, so insertion of the next element.

So, the next value is minus 30, 10 so, as minus 30 is less than, so this is the dimension we will be considering here. So, minus 30 is less than 25 according to the binary search tree rule so, I have to place it here. And next is minus 60 minus 50 so, if I go here, so first I have to compare minus 60 and since this is this dimension x, so minus 60 it should go there.

Now, minus 50 would be compared because now the y-dimension in this node, the y dimension has to be compared. So, minus 50 is less than 70 so, it should go in this node. So, this is minus 60 minus 50, here y-dimension is compared; consider so, in this case we need to compare the x-dimension, right. So, consider the next element so 40.

So, first we compared from the root 40; so, 40 means it should come to this place so, this is vacant. So, it will be placed here itself 40, 90 and this node the cut dimension y to compare, next is 15, 30 so, if I come here, so 15 it will be compared with 15 so, it will be coming to this node. Now it will be compared with 30, it will be coming to this node because 30 is less than 70. So, now again the x will be compared, it is 15; 15 is more than minus 60. So, this value is 15, 30 and here the cut dimension is y. So, in this way as you can see a tree is built. So, as you go on including more number of elements your tree will be increasing its levels and also it will be expanding its nodes.

And in this particular so, any query image suppose you have you have a query say 10 minus 3 and we would like to get the nearest neighbor from this query. So, what we can do that it is not simply like you can apply the binary search tree rule because even if you move to certain sub-tree, the other sub-tree also may contain the nearest neighbor.

Because if I note how the partitions of this space is made in this fashion, say let me consider a space where you have if the values are ranging from minus 100, minus 100 to say 100,

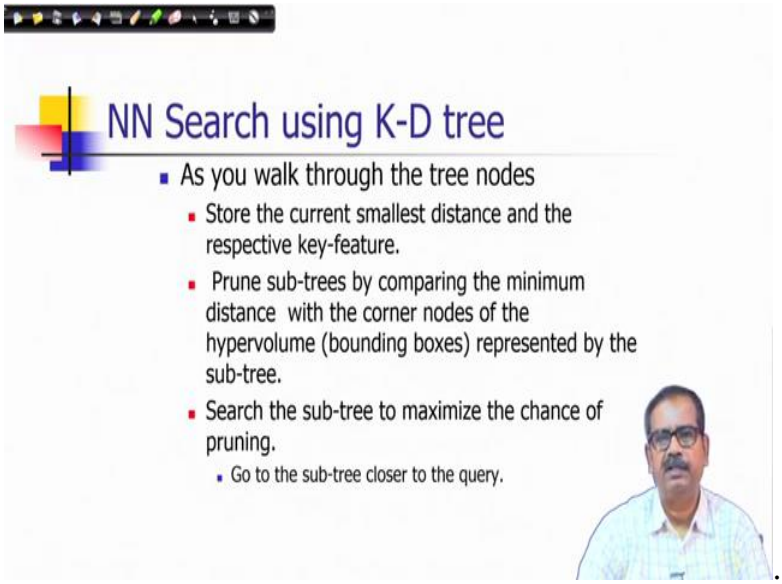
100. And in the first case as you can see that when we are in the root, it is 25 sets. So, you are using 25 so, it is minus 100 is this so, 25 maybe this location for the x-dimension so this is x, this is y.

So, all the values which are more than 25, so this is a subject of the space at the root location; from the root location. So, all the values; all the points two dimensional points which are which will be in the sub-tree should lie in this particular subspace, a box and all other values, all these sub-trees that should lie in this space. So, that is idea and then you consider minus 30, 70 and that is in this case you have to use the y coordinate. So, 70 would be used for; so, you are now in this book sub-tree left sub-tree so, 70 could be somewhere here. So, this is a region which will be representing this particular sub-tree minus 30, 70; so, this is 70 so, this is 25 x direction and say this is 70, if I consider 40, 90 so, that should be also in the y region rather 40, 90 is a point. So, these are the two regions.

So, this should be the left sub-tree of this one and this should be the you know right sub-tree. Similarly this partition, this is the left partition of this node and this is the right partitions. So, when you have minus 60 minus 50 at as it is in the left notation, so you should consider here and it is the dimension x. So, it will be something like this. So, this is minus 60; one second this is a left and this is right and when it is 15, 30 that is from the y, so it should be something like here. So, you can see that for this node, your this node is the right sub-tree and this node is the left sub-tree.

So, in this way you are subdividing a region into different cells and your query point should be expected your if you would like to place a point, it should be placed on in that region and it is expected that your search space, your neighboring point should be in that region. But the problem is that some of the points even in this bordering region they could be also neighbors. So it is not sufficient to search just this cell, you have to search also the neighboring searches. So, you have to follow certain search strategy which is not exactly the same as binary search strategy to find the nearest neighbor. So, let us discuss that also here.

(Refer Slide Time: 18:25)



So, we will be considering a nearest search using K-D tree as we have seen that how K-D tree partitions the space and trying to locate a query into a particular cell, though its neighboring cells are also to be considered for finding the neighboring regions. So, the strategy is that as you walk through the tree nodes, you should always know the current smallest distance and the respective key feature.

You should not ignore any sub-tree when while traversing this node for searching a feature, key feature, tou should not ignore it unless you make sure that all the values, all the feature vectors or all the key vectors within that sub-tree they are at a distance which is greater than the minimum distance what has been found so far among the features space. So, that is why you have to store the current smallest distance and the respective key feature, then you can prune sub-trees by comparing the minimum distance with the corner nodes of the hyper volume, the here the bounding boxes what you have shown.

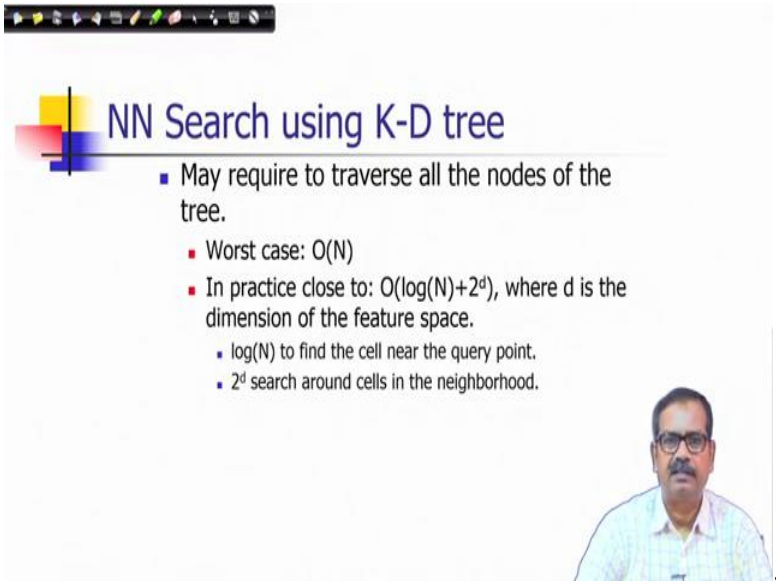
So, you can also note down their corner points and you can compare them these distances. If you find that one of these corners, see all the distances of this corner points they are greater than the minimum distance, then you need not go through. You should not go through that searching of any key values in that region. So, in this way you can prove.

So, while taking a decision in the sub-tree where you should move next, you should search the sub-tree to maximize the chance of pruning which means go to the sub-tree which is closer to the query. So, you are not no, you are not you are not immediately taking any

decision of removing any sub-tree from your search space as we do in the binary search tree., at every note we take a decision and we go either left or right, but in a K-D tree we cannot do that.

We have may traverse along both the sub-trees unless you make sure that the sub-tree, the cell represented by the sub-tree, all the corner points they are at the distance greater than the smallest distance what you have in your current execution and then you can prune that. So, in this way you go and finally, whatever sub-trees you are having at the end you have to compare all the feature vectors in those sub-trees to find out the neighboring vectors.

(Refer Slide Time: 21:11)

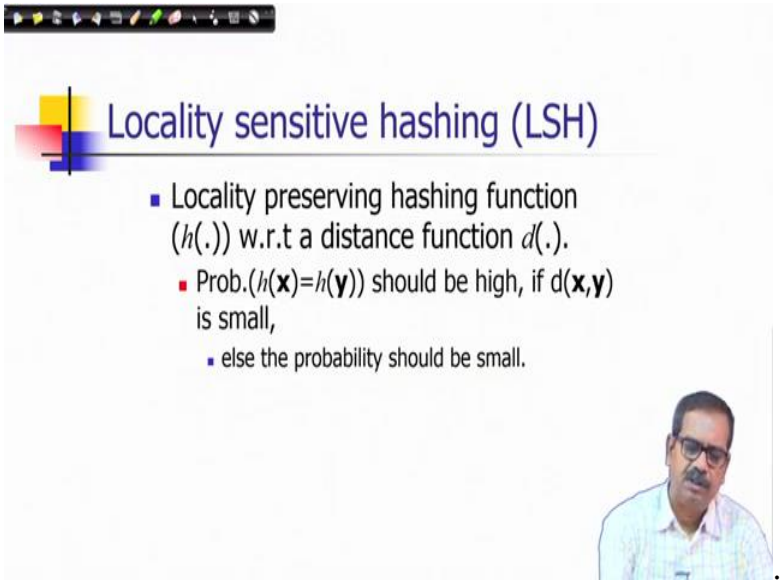


So, the time complexity if we consider like even in binary search tree also worst case space time complexity, still linear in K-D Tree also it would be linear, but in practice you may get

$$O(\log(N) + 2^d), where\ d\ is\ the\ dimension\ of\ the\ feature\ space$$

So, log(N) to find the cell near the query point and 2^d for searching around the cells in the neighborhood. As I mentioned that it is just not that cell, you have to consider also neighboring cells.

(Refer Slide Time: 21:45)

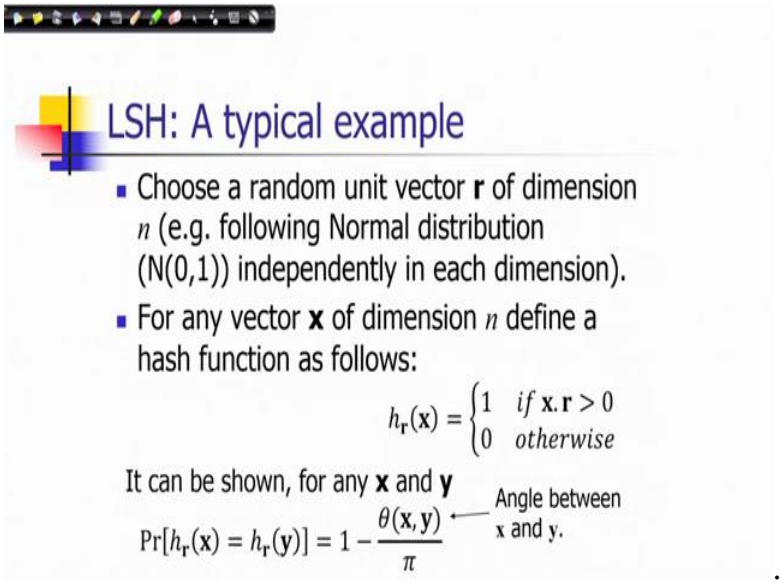


The other technique for performing this efficient search is locality sensitive hashing technique and I would also elaborate a bit more about this technique. I will provide you the principles without going into much details. So, the property here is that you have to use an hash function which should preserve the locality of feature vectors with respect to distance function which means that if I consider a hash function say h, and if I consider two feature vectors x and y which are being shown here by bold fonts.

So, the both the hash functions if they are same, so their probability that both of them would be same would be very high. If the distance is between them is also small, that means they are very closely spaced. So, if I can ensure this property of a hash function, then those hash functions will be useful in placing similar features into the same group, same bucket.

So, that is a principle we will be using. So, let us find; let us consider an example of a hash function as it is true that know this should be the probability should be high when their distances are small. Similarly probability of hash function should be different when the distances are large. So, it should satisfy that property also.

(Refer Slide Time: 23:19)



So, a typical example as we can consider that, consider a random unit vector r of dimension small n. And for example, you can form this vector from using a normal distribution independently generating the values following this distribution in each dimension. That would give you n times if you do, then that would give you a feature vector of random feature vector of dimension n. So for any vector x of dimension n, you can define a hash function.

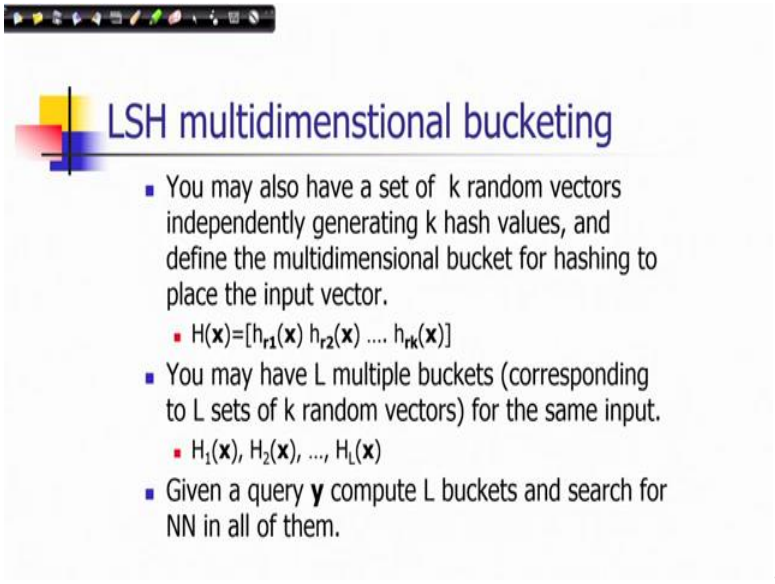
$$h_r(x) = \begin{cases} 1\ if\ x.r > 0 \\ 0\ otherwise \end{cases}$$

So, it is a two valued hash function. So, there are only two vectors either 1 or 0, but the interesting fact is that this would it can be shown that for any two vectors x and y, then probability that two hash values would be equal is given by this fact.

$$\Pr[h_r(x) = h_r(y)] = 1 - \frac{\theta(x,y)}{\pi}$$

So, you can express this angle also in terms of radian, you know that this angle is expressed in terms of radian. In fact, this function is closely similar to cos cosine functions, so which means the angle is very small then those feature vectors are quite nearer and then there is hash values probability of having the same hash value which is equal to 1 in this case. So, any hash value as either both of them are 0 or both of them are 1. So, that would be quite high. So, that is you know particular feature particular property of this or significance of this property.

(Refer Slide Time: 25:29)



So, using this property you can design a scheme. So, you can design a scheme of multi-dimensional bucketing. So, you consider that instead of having a single vector you have a k random vectors and so, independently you can generate k hash values. So, each will give K-dimensional binary representation and that itself can be an index of a bucket, multi dimensional bucket and you can place your input vector in one of these buckets. So, if you have k hash values as if it is a binary representation, there could be 2^K buckets and in one of these buckets.

So, whenever an input vector has the same bucket number, you will be placing them in that bucket. So, in this way you can perform in a hashing. So, this is how a particular bucket is characterized. Now you can further know extend that instead of having a single no single multi-dimensional bucketing scheme, you can repeat this process. You can have L multiple buckets. So, instead of keeping a feature vector only in one bucket, you consider there are L such family of hash functions and independently you are keeping L multiple instances by doing you are making the chance of getting neighboring feature vector, the probability of neighboring feature vector would be high that is a chance by doing it multiple times.

So, you are repeating this process L times and for example, you can generate such you know multi-dimensional vectors hx like L times. So, you have L such buckets and that is the scheme and then when you are performing a nearest neighbor search what you can do

that given a query, you have to compute L buckets and search for nearest neighbor in all of them. So, that makes your locality sensitive hashing more efficient and chance of missing a nearest neighbor will be less.

So, with this let me stop this lecture at this point and we will continue this discussion. In fact, in this topic we have these two parts; one is matching the other part is model fitting which is related as I mentioned because after matching the points, then know using those points you are trying to derive a model which explains the data. So, we will be discussing some of these techniques of model fitting in the continuing lecture.

Thank you very much.

Keywords: K-D tree, nearest search, locality sensitive hashing, bucketing