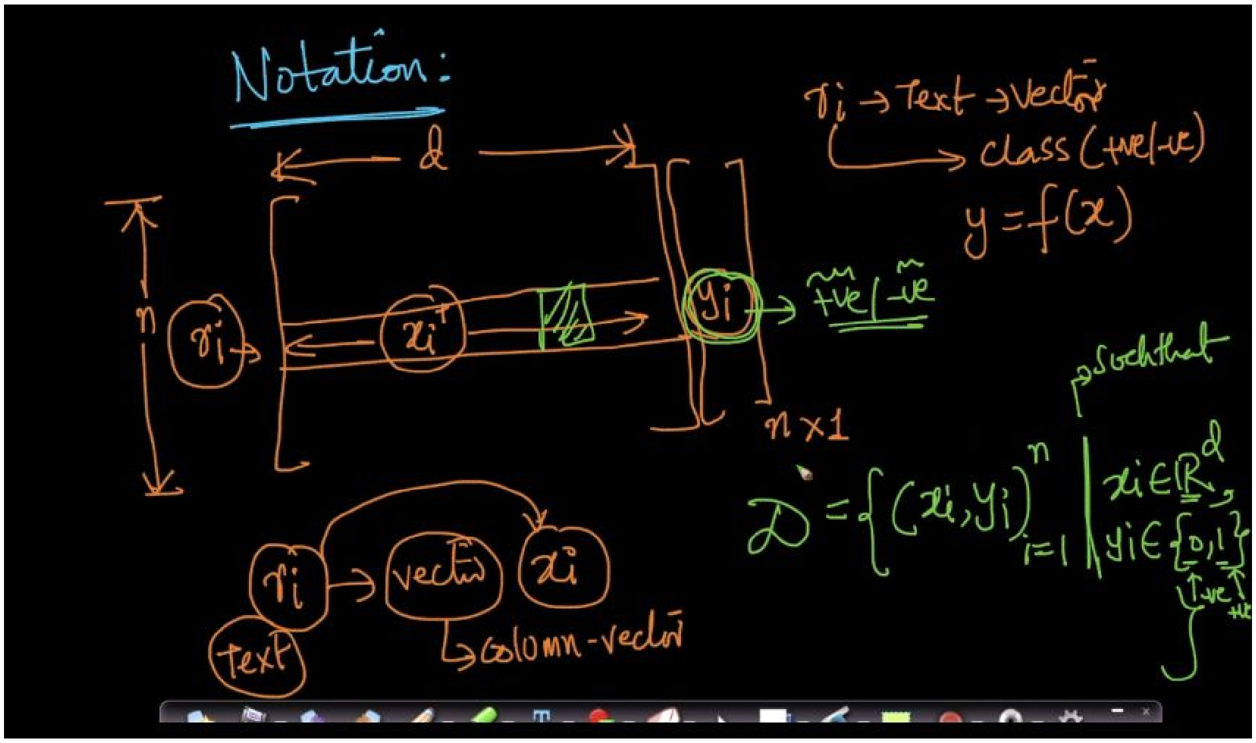
Data Matrix Notation:

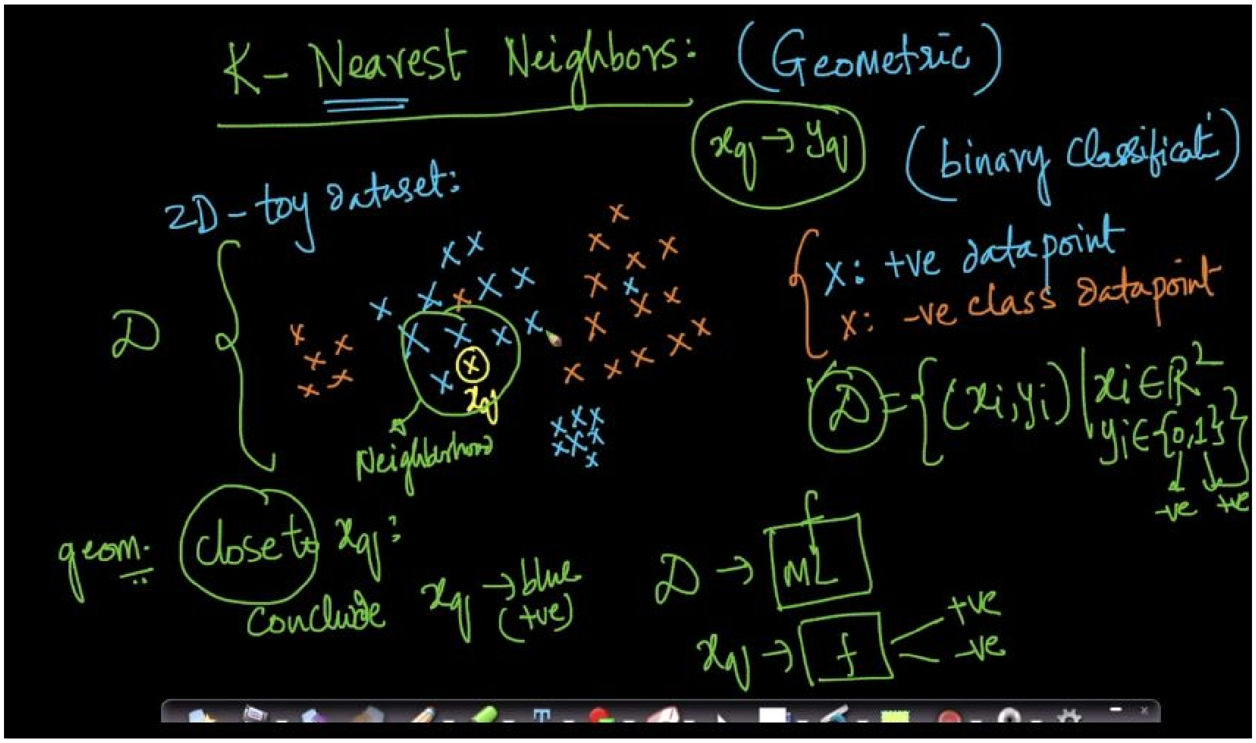
By default, a vector is a column vector.



For every row ri we have a vector representation xi. Since we are denoting ri as a row, the corresponding vector would be xi.T (Since x denotes each feature column).
And for each xi, we have a corresponding class label yi (here the class label is in binary 0,1)

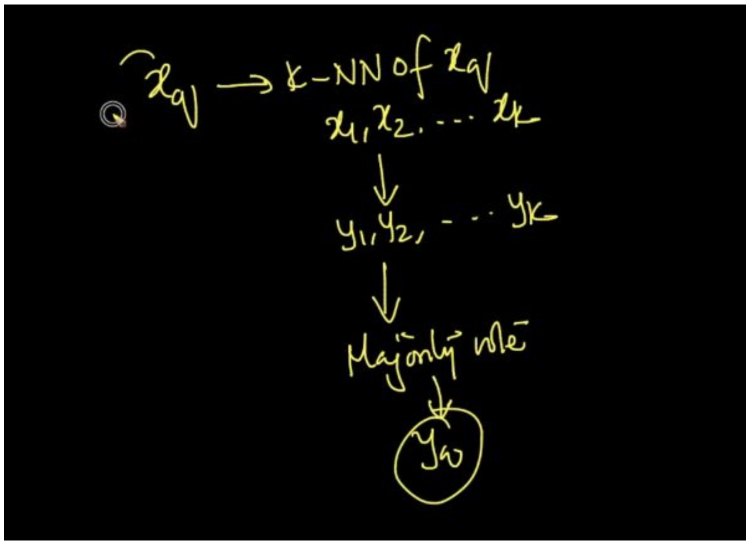
Hence, to sum it up, Dataset (D) = a set of (xi, yi) ranging from 1 to n, such that, xi belongs to R^d (where R is a real number an d is the dimension) and y belongs to {0,1}

K-Nearest Neighbours (KNN)



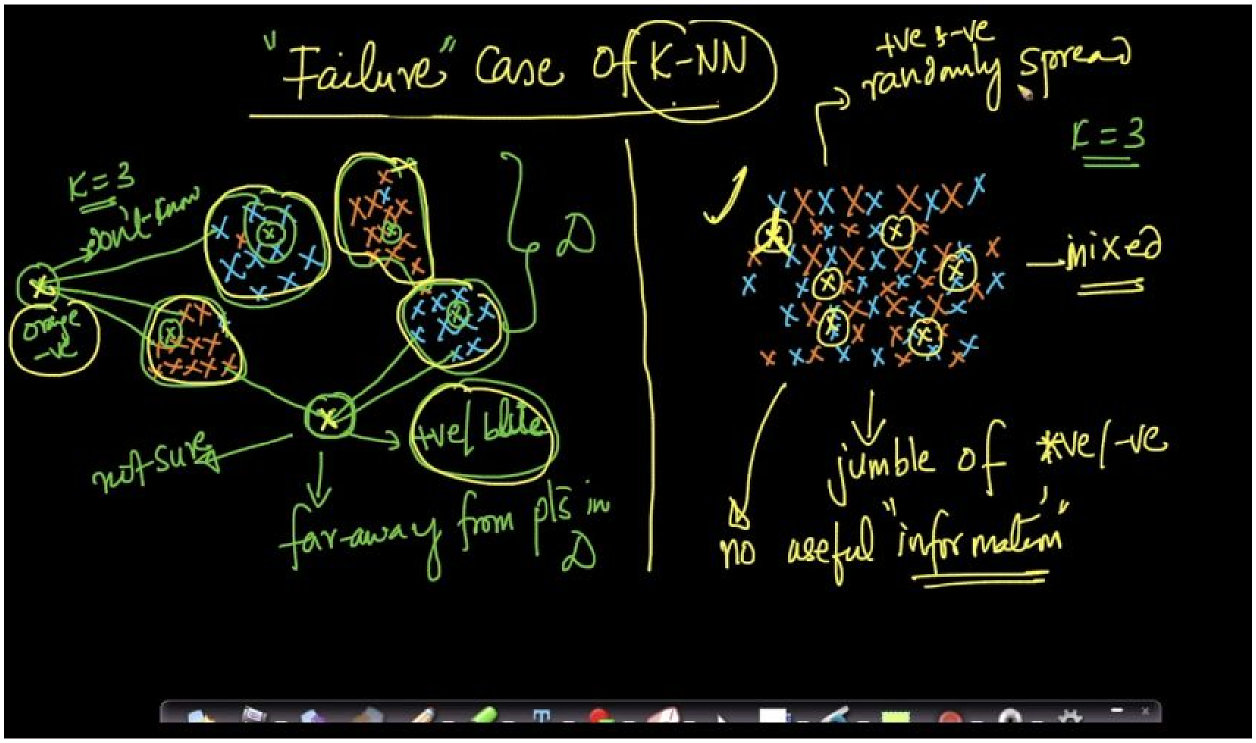
Majority vote is about finding the majority of the class labels among k for a query point xq.

We should choose k as an odd number (2n+1) since in an even k, the majority vote might be inconclusive due to equal number of positive and negative class labels.



To measure the accuracy of KNN:
Accuracy = (Number of accurate predictions in Dtest)/(Total number of points in Dtest)
Hence, Accuracy lies between 0 and 1.

Failure cases of KNN:

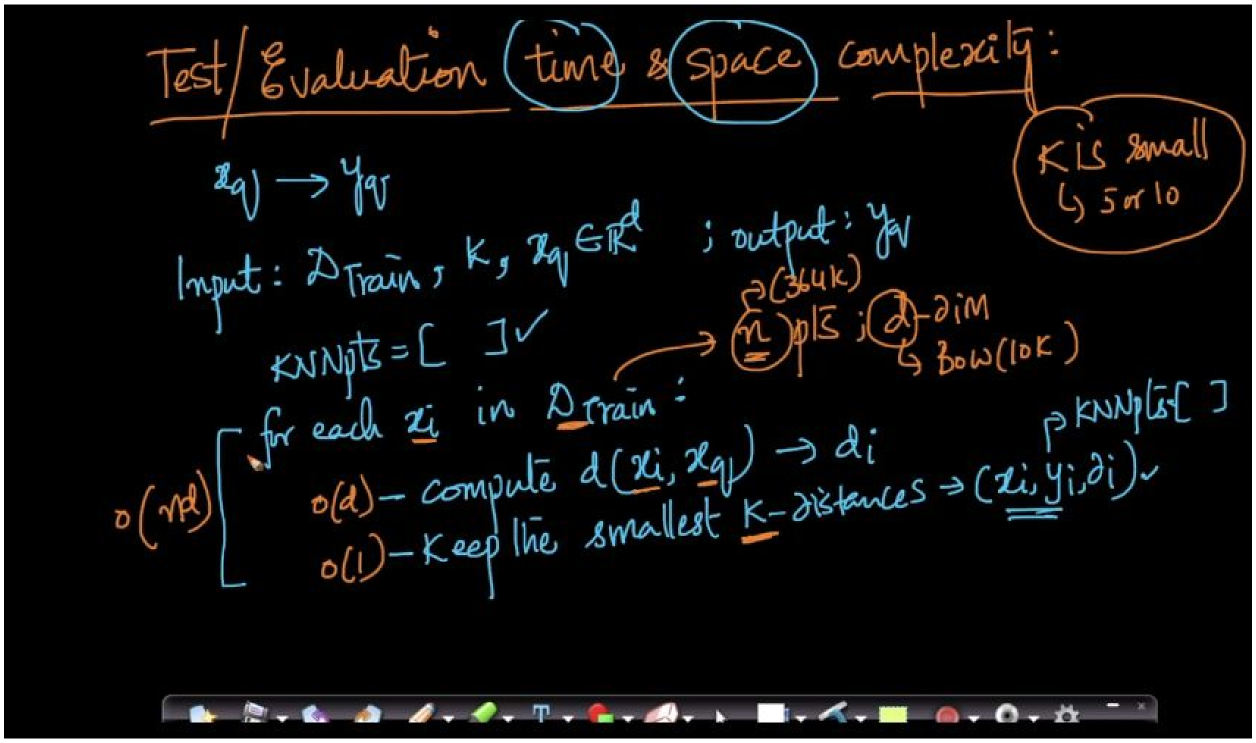


Case 1: If the query point is *far away from the clusters or equidistant from the different clusters of the dataset*, it will will be inaccurate to apply KNN in these cases as the result might be misleading. Eg: The query point might have properties different to the trained clusters.

Case 2: If the data points are randomly spread, then it's quite futile to draw any observable conclusions by applying KNN on a query point.

NB: One of the biggest drawbacks of KNN is the time and space complexity it is required for the algorithm to run which is in the order of O(nd).

KNN Time and Space Complexity:



Test/Evaluation time & space complexity:

K is small ↳ 5 or 10

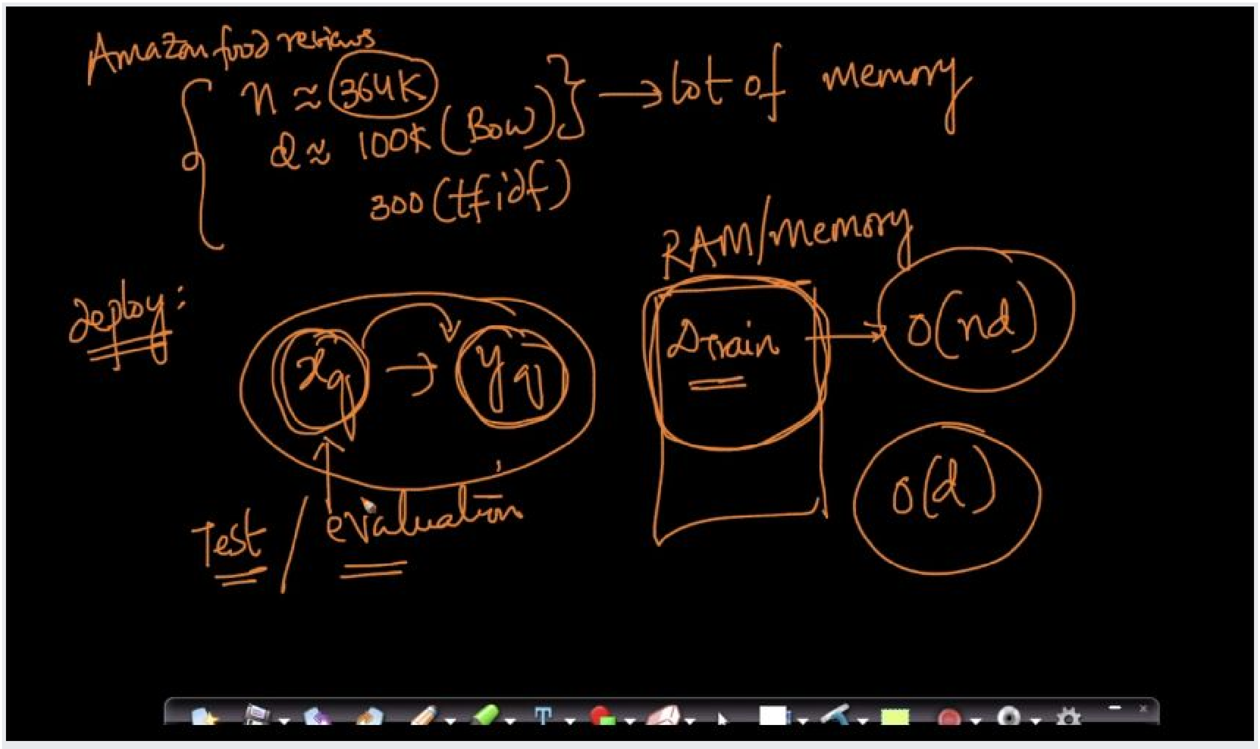
$x_q \rightarrow y_q$

Input: D_{Train}, K, $x_q \in \mathbb{R}^d$; output: y_q

n pts (364K) ; d-dim ↳ BoW(10K)

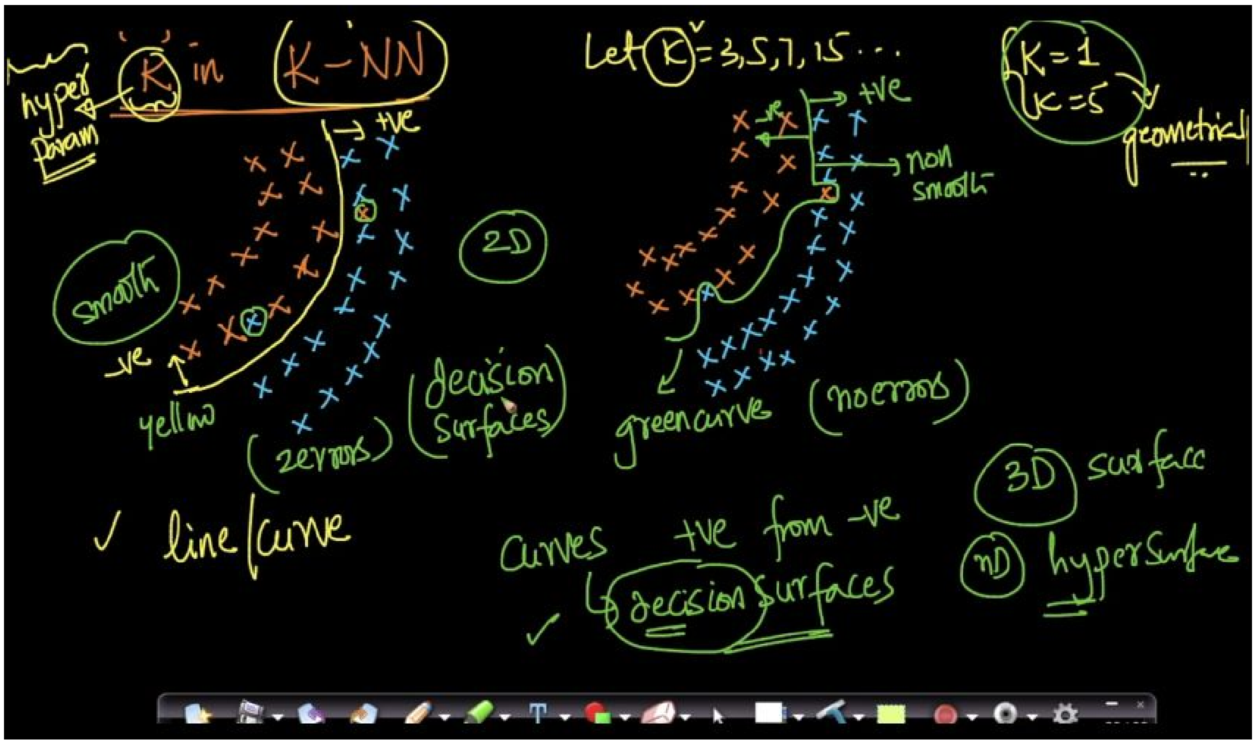
```
KNNpts = [ ] ✓
O(nd) [ for each $x_i$ in $D_{Train}$:
    O(d) - compute $d(x_i, x_q) \rightarrow d_i$
    O(1) - Keep the smallest K-distances $\Rightarrow (x_i, y_i, d_i)$ → KNNpts[ ]
```

Time Complexity:
To run the data set,
- Compute all the n rows
- For each row compute the distance (d)
Hence it takes **O(nd) time complexity**



Space complexity:
The space required in the RAM to predict yq given xq is,
The whole of the training set which is O(nd)

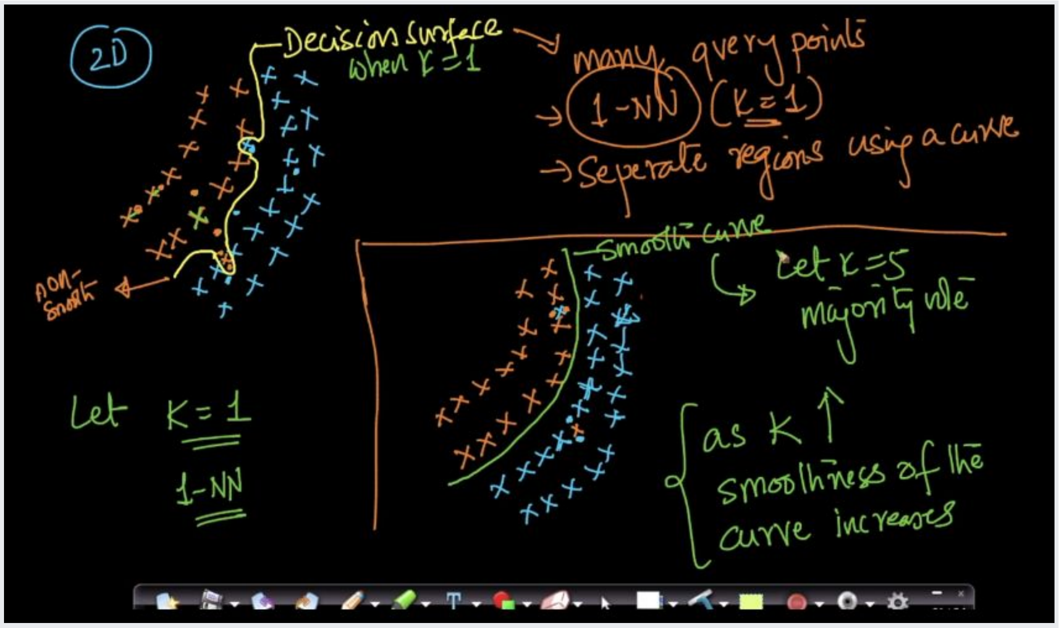
Decision surface for KNN as K changes:



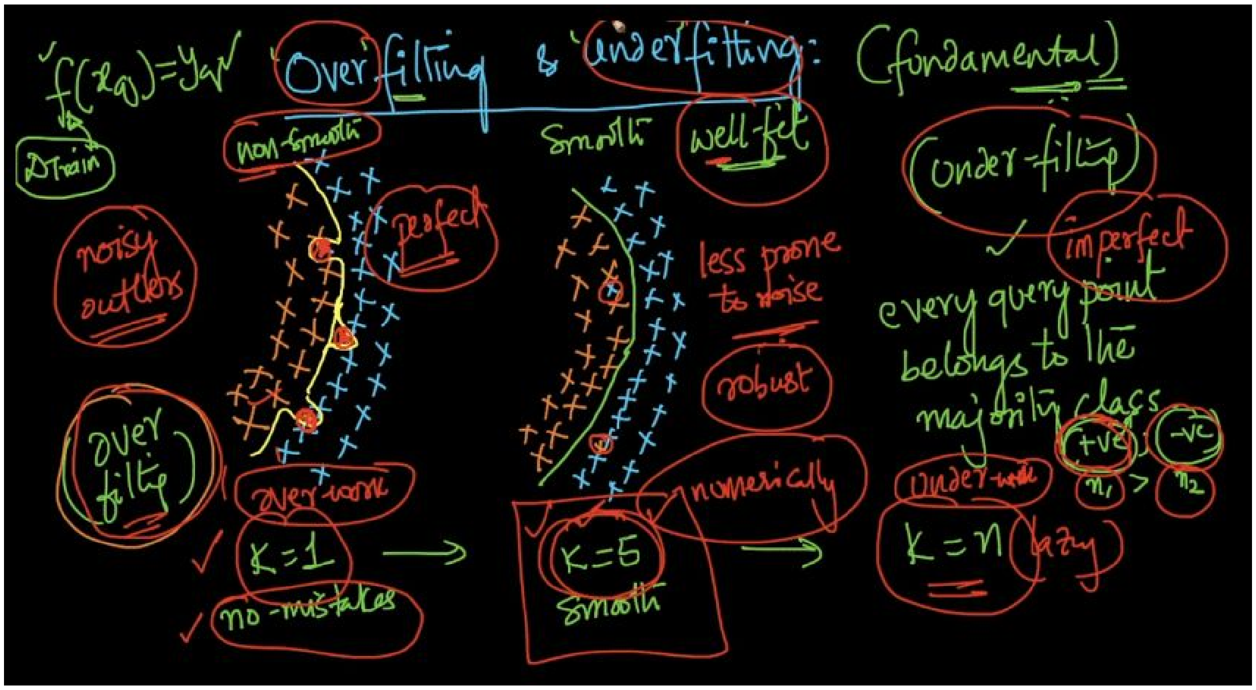
As K increases, the smoothness of the decision surface increases.

If K=1, the decision curve becomes jagged.

If K=n, then the query point will become the majority class.


2D
Decision surface when K=1
many query points
1-NN (K=1)
Seperate regions using a curve
non-smooth
smooth curve
Let K=5
majority vote
Let K=1
1-NN
as K ↑
smoothness of the curve increases

Overfitting and Underfitting:



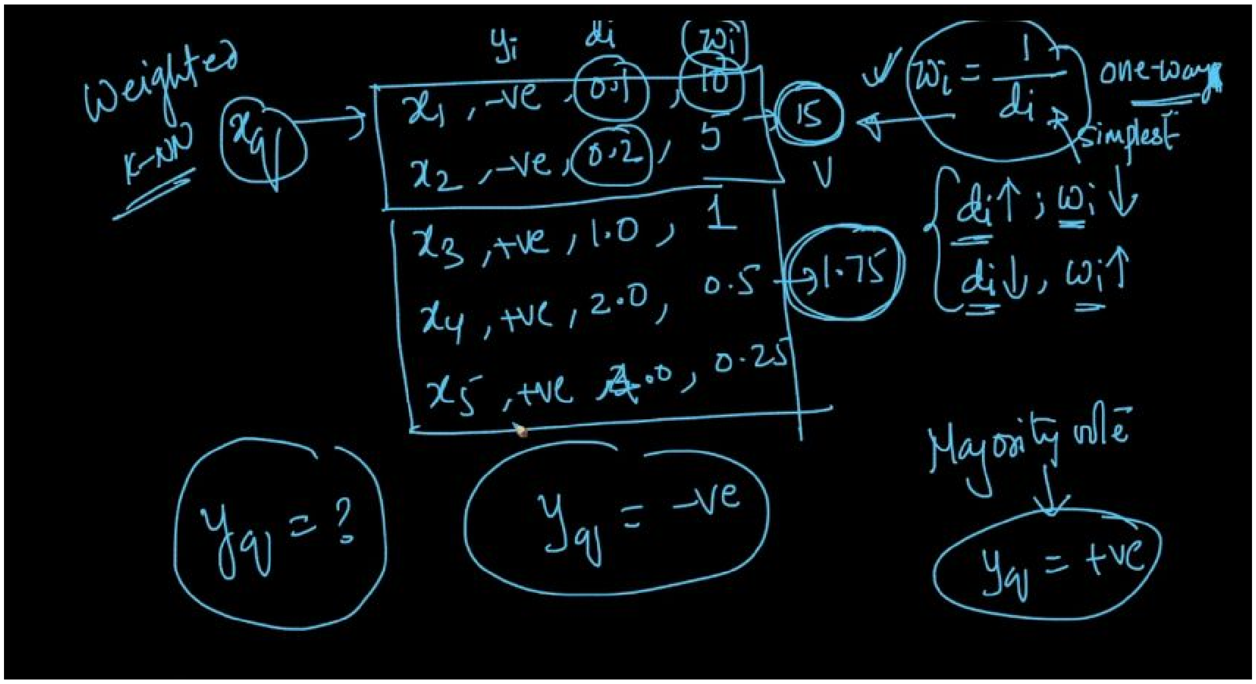
In statistics, a fit refers to how well you approximate a target function. This is good terminology to use in machine learning, because supervised machine learning algorithms seek to approximate the unknown underlying mapping function for the output variables given the input variables.

If k=1, we try to fit a function which tries to overfit for every point in the data set in order to create no mistakes at all.

If k=n, we try to fit a function which tries to underfit and under-work making an incorrect mapping of the query point by taking all the points and hence make incorrect assumptions.

In middle diag, we consider outliers/noise and make it less prone to such and hence make the function more robust.

Weighted KNN:



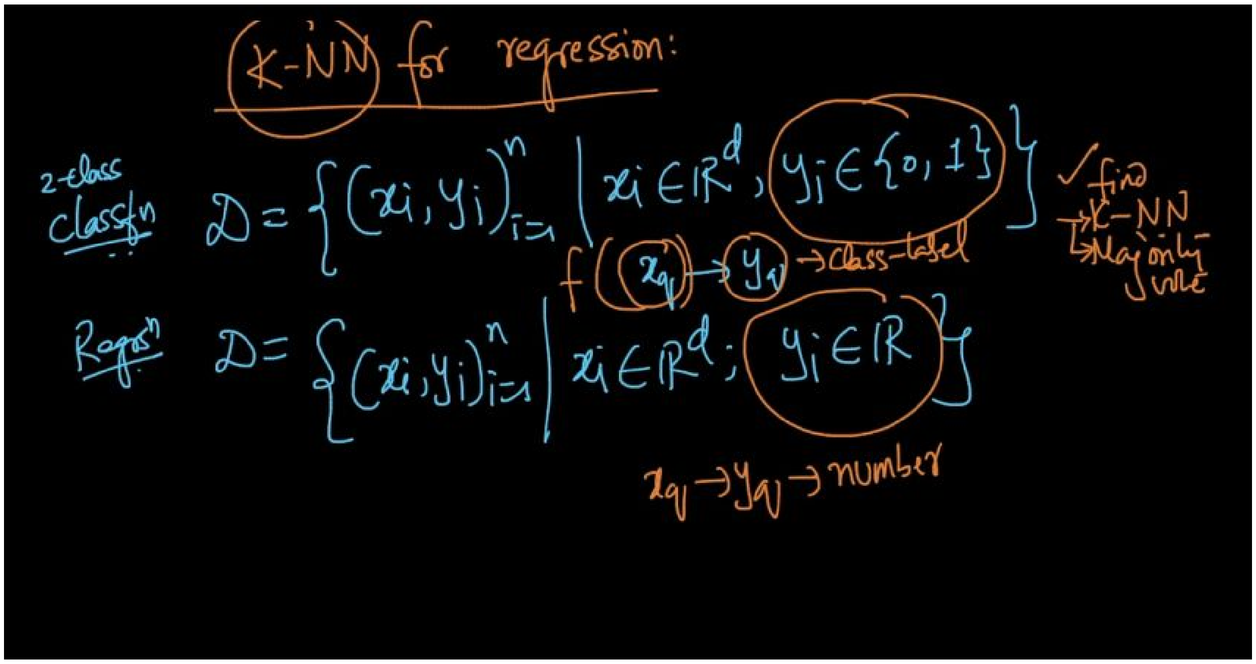
In weighted KNN, we take the distances of all the K points from the xq.

Then we find the **reciprocal of the distances** *(=weight)* of each point from xq.

Then we find the weight-age of each class label by summing up the component labels respective to the class.

Finally, we assign yq to the class which has the maximum weight-age.

KNN for regression:



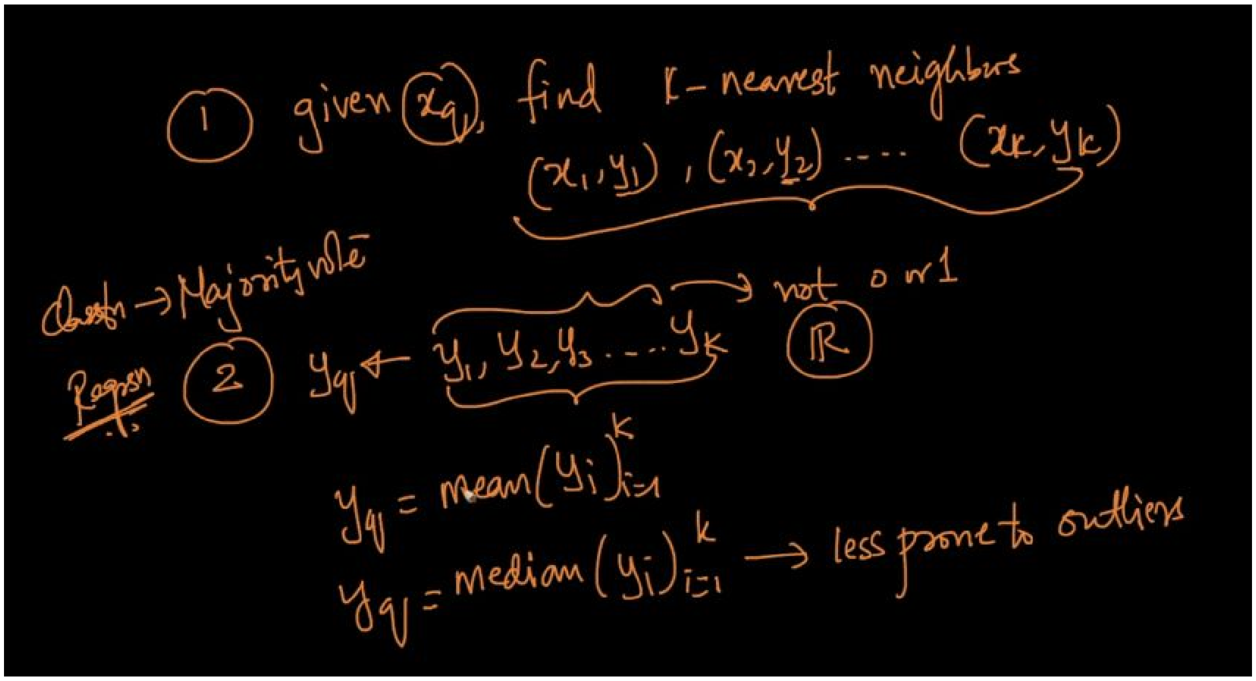
K-NN for regression:

2-class classfn $D = \{(x_i, y_i)_{i=1}^{n} \mid x_i \in \mathbb{R}^d, \boxed{y_i \in \{0,1\}}\}$ ✓ find → K-NN → Majority vote

$f(\boxed{x_q}) \to \boxed{y_q} \to$ class-label

Regresn $D = \{(x_i, y_i)_{i=1}^{n} \mid x_i \in \mathbb{R}^d; \boxed{y_i \in \mathbb{R}}\}$

$x_q \to y_q \to$ number



① given $\boxed{x_q}$, find K-nearest neighbors

$\underbrace{(x_1, y_1), (x_2, y_2) \dots (x_k, y_k)}$

Classfn → Majority vote

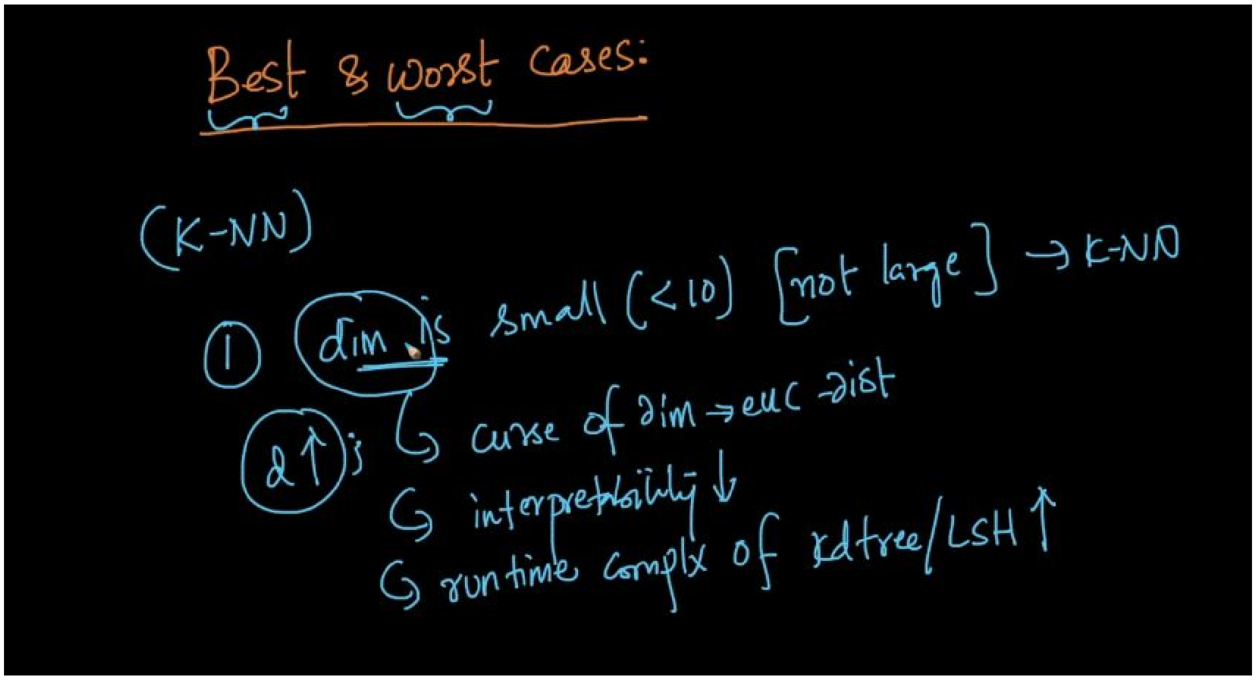
Regresn ② $y_q \leftarrow \underbrace{y_1, y_2, y_3 \dots y_k}$ → not 0 or 1 $\boxed{\mathbb{R}}$

$y_q = \text{mean}(y_i)_{i=1}^{k}$

$y_q = \text{median}(y_i)_{i=1}^{k}$ → less prone to outliers

To apply KNN in regression, which predicts y as a real number, given xq we find the optimal K value and after finding the KNN for the xq, we just take the mean/median of those neighbours.

Best and worst cases of KNN:


Best & worst cases:

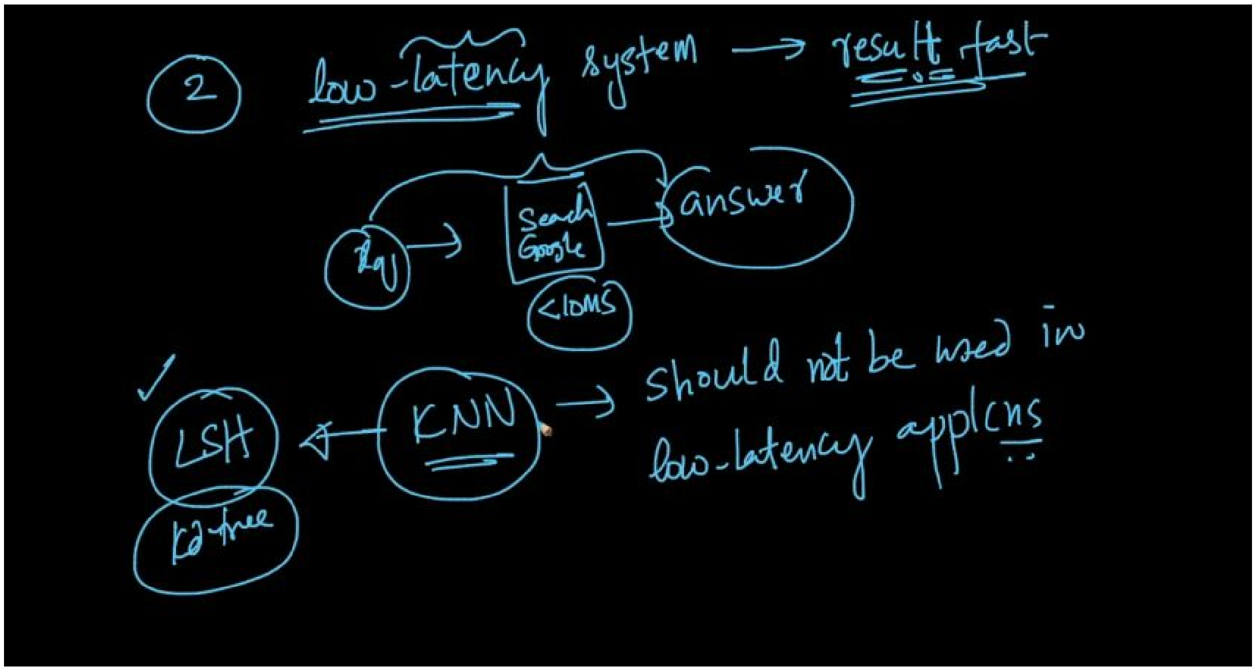
(K-NN)

① dim is small (<10) [not large] → K-NN

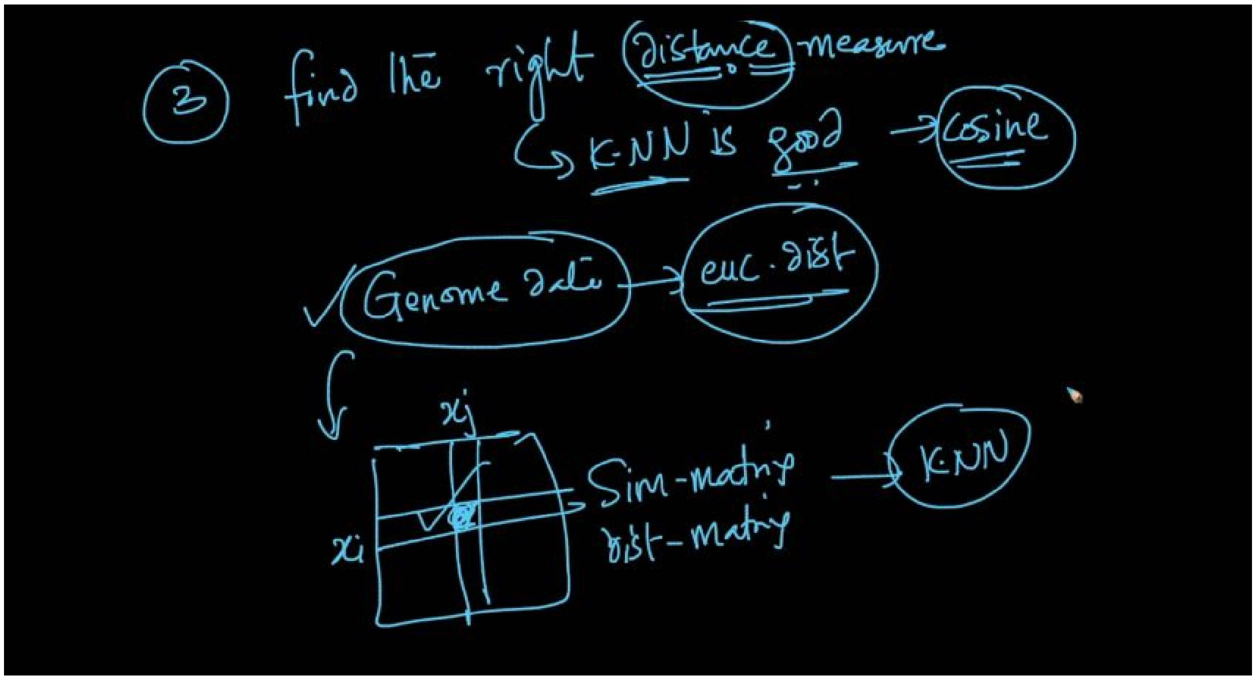
d↑ ; ↳ curse of dim ⇒ euc-dist

↳ interpretability ↓

↳ runtime complx of kdtree/LSH ↑

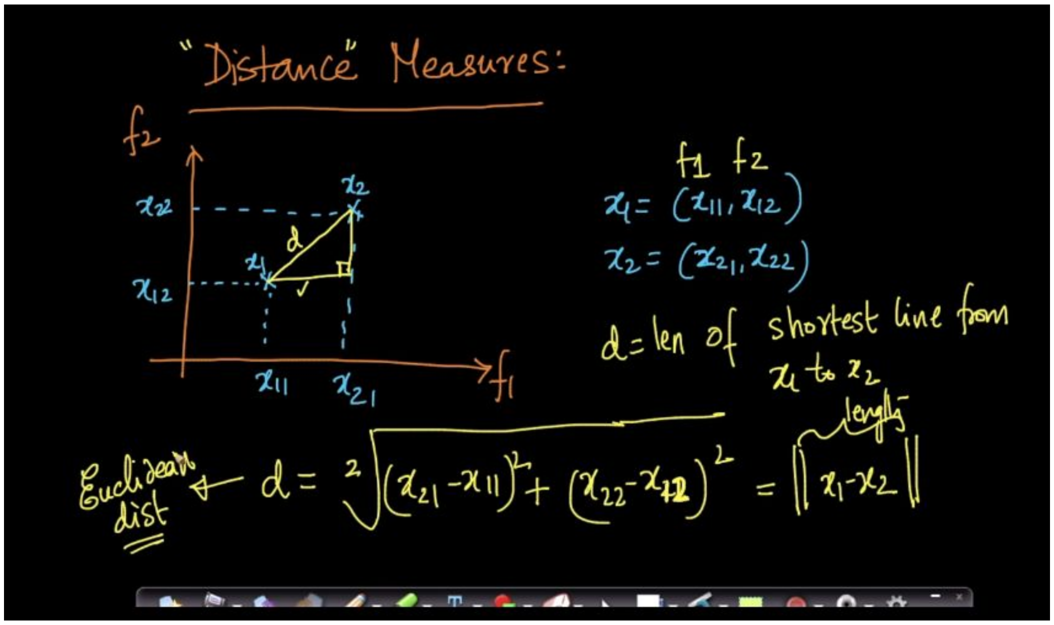

② low-latency system → result fast

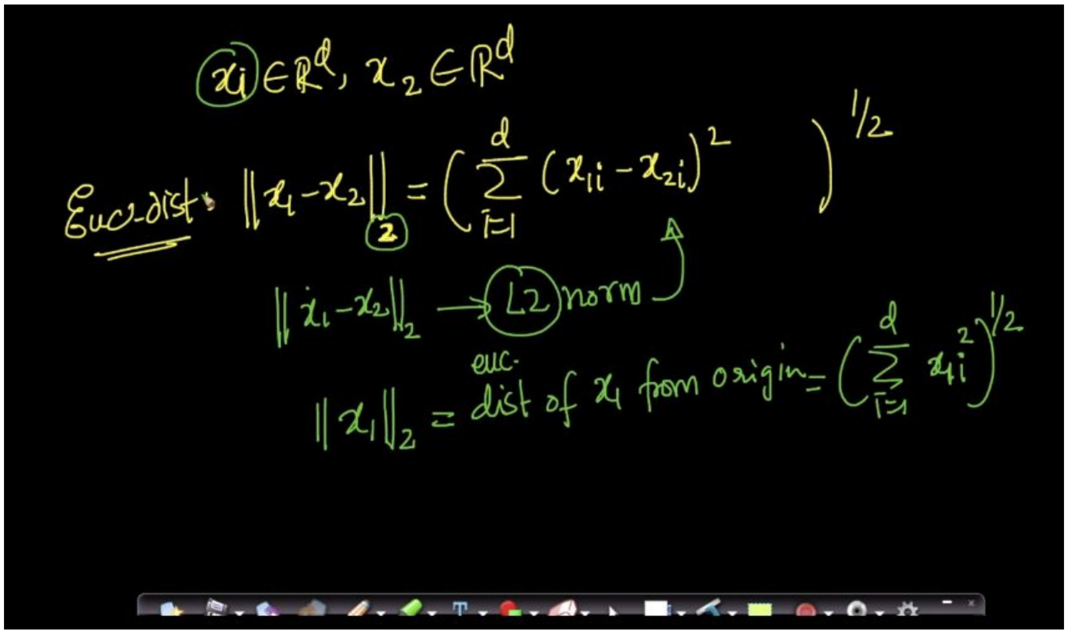
KNN → should not be used in low-latency applcns



Distance Measures:

1. Euclidean Distance:


"Distance" Measures:
f2
f1
x22
x12
x11
x21
x1
x2
d
f1 f2
x1 = (x11, x12)
x2 = (x21, x22)
d = len of shortest line from x1 to x2
Euclidean dist ← d = √((x21 − x11)² + (x22 − x12)²) = ‖x1 − x2‖
length



$x_1 \in \mathbb{R}^d, \; x_2 \in \mathbb{R}^d$

Euc-dist: $\|x_1 - x_2\|_2 = \left(\sum_{i=1}^{d} (x_{1i} - x_{2i})^2 \right)^{1/2}$

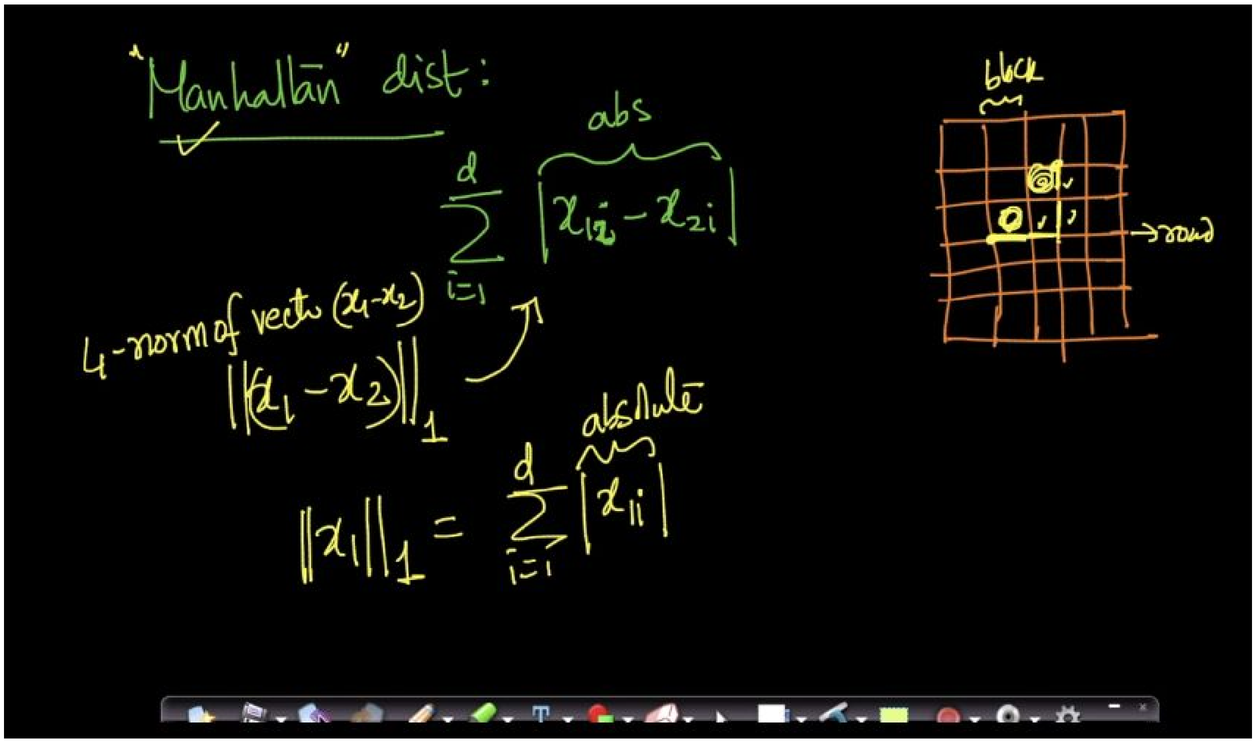
$\|x_1 - x_2\|_2 \rightarrow$ L2 norm

$\|x_1\|_2 =$ euc. dist of x_1 from origin $= \left(\sum_{i=1}^{d} x_{1i}^2 \right)^{1/2}$

The Euclidean distance d is derived by applying Pythagoras' Theorem on the points x1 and x2 on the axis f1 and f2 respectively.

The Euclidean distance is also called the L2 norm distance.

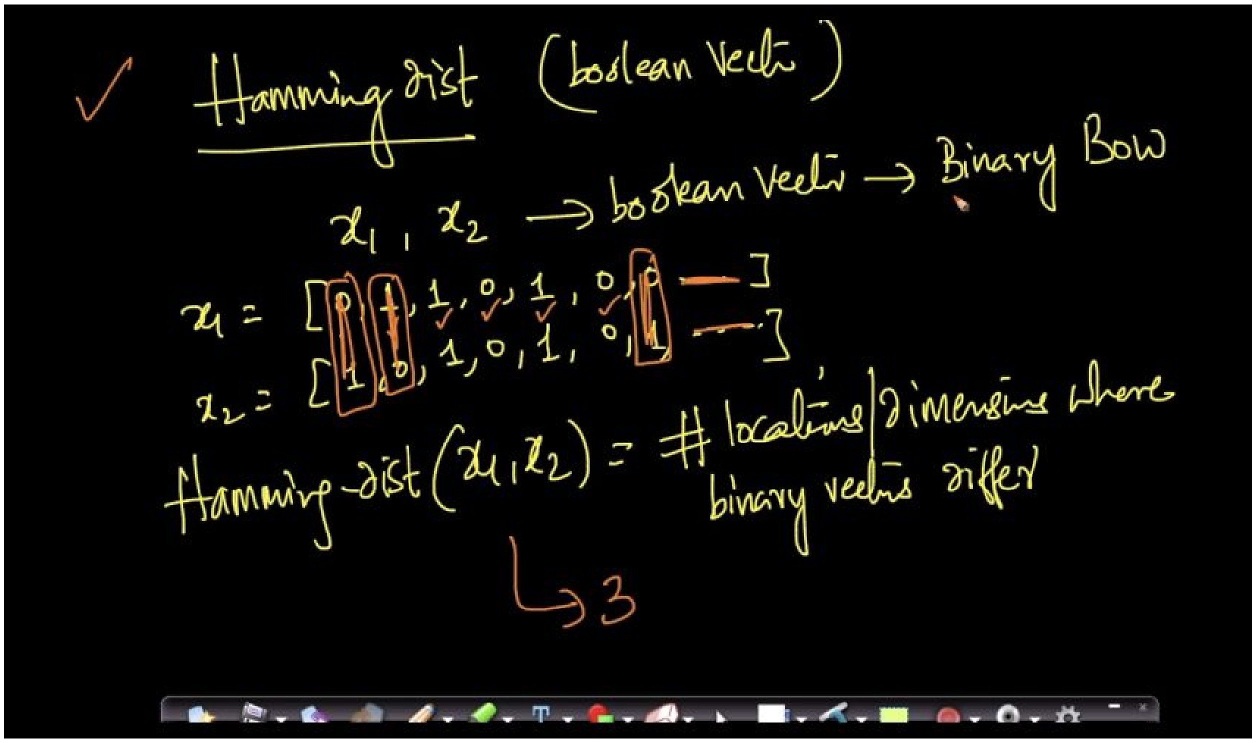
2. Manhattan Distance:


"Manhattan" dist:
abs
$\sum_{i=1}^{d} |x_{1i} - x_{2i}|$
L-norm of vector $(x_1 - x_2)$
$\|(x_1 - x_2)\|_1$
absolute
$\|x_1\|_1 = \sum_{i=1}^{d} |x_{1i}|$
block
road

In a grid like structure, comprising of unit cells, Manhattan distance is the distance between two vertices of cells. It is measured as the shortest distance between the two points along the sides of the cells.

Manhattan distance is also called the L1 norm distance.

3. Hamming Distance:



✓ Hamming dist (boolean vectrs)

$x_1, x_2 \rightarrow$ boolean vectrs $\rightarrow$ Binary BoW

$x_1 = [0, 1, 1, 0, 1, 0, 0 \text{ --- }]$

$x_2 = [1, 0, 1, 0, 1, 0, 1 \text{ --- }]$

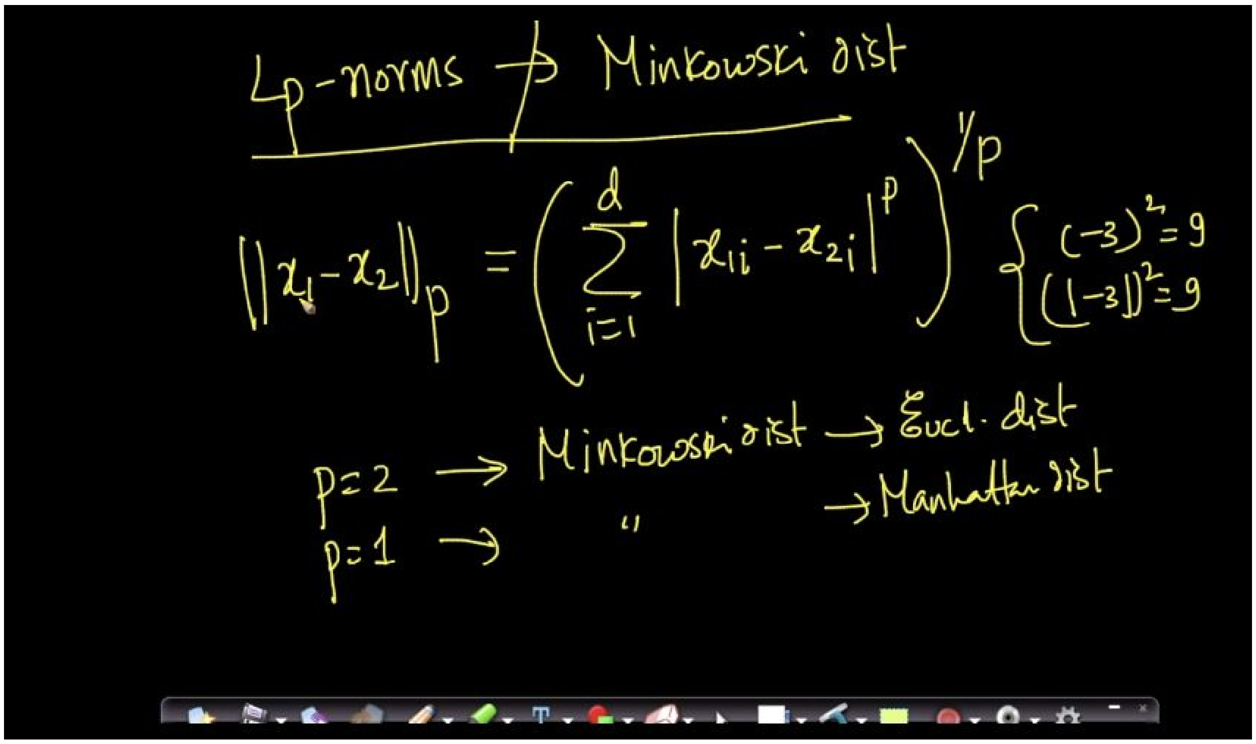
Hamming-dist(x_1, x_2) = # locations/dimensions where binary vectrs differ

↳ 3

Hamming distance between two vectors is the number of locations/dimensions where the vectors differ.

It is useful in cases of finding difference between strings and genetic sequences.

4. Minkowski Distance:



Lp-norms → Minkowski dist

$$\|x_1 - x_2\|_p = \left(\sum_{i=1}^{d} |x_{1i} - x_{2i}|^p\right)^{1/p} \quad \begin{cases} (-3)^2 = 9 \\ (|-3|)^2 = 9 \end{cases}$$

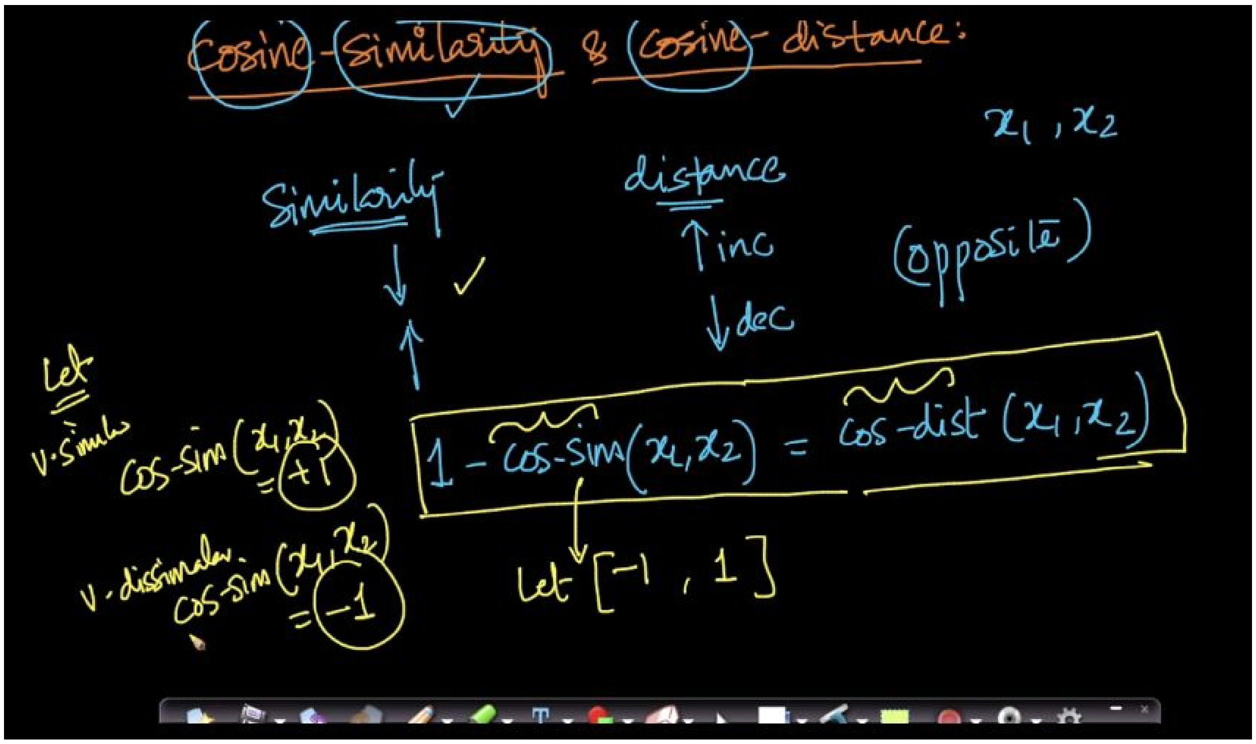
$p=2 \rightarrow$ Minkowski dist $\rightarrow$ Eucl. dist

$p=1 \rightarrow$ " $\rightarrow$ Manhattan dist

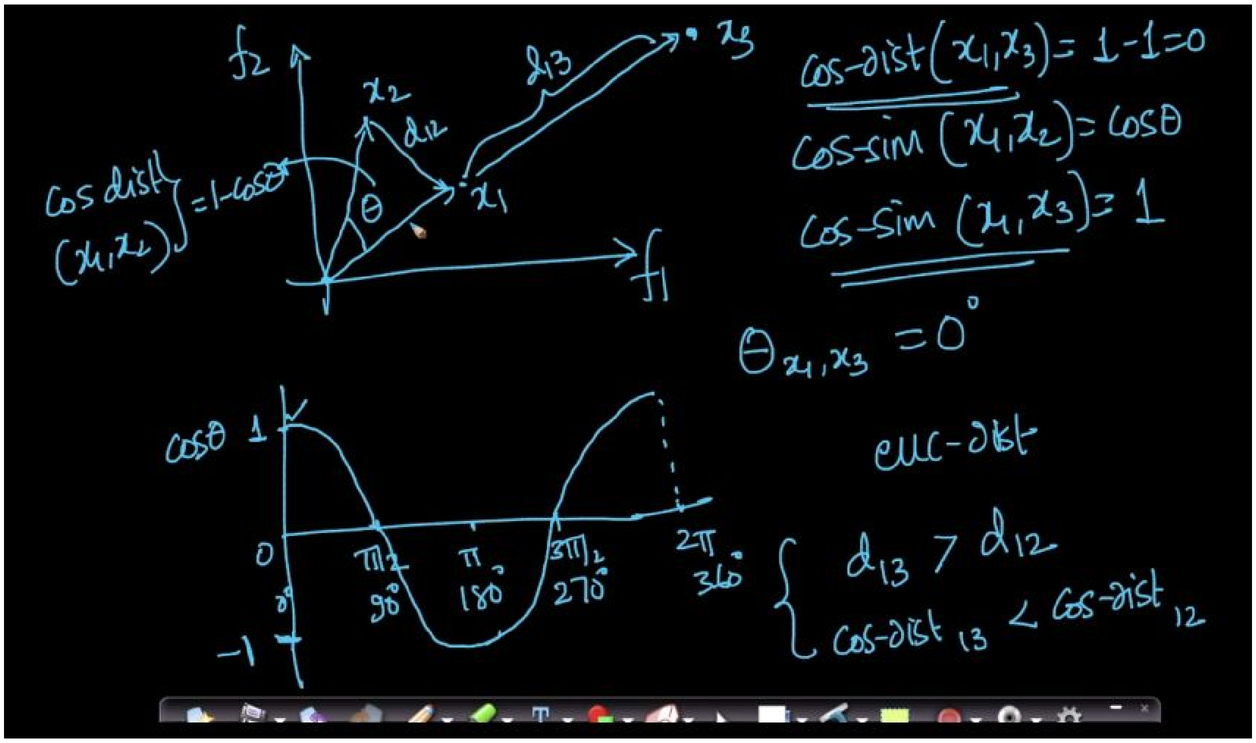
Minkowski distance is a concise way of summarizing both Euclidean and Manhattan distance where p>0 and p!=0.

Minkowski distance is also called the Lp norm of vector(x1-x2).

5. Cosine distance and cosine similarity:



In general, if similarity increases, distance decreases between two points.



$\underline{\text{cos-dist}(x_1,x_3)} = 1-1=0$

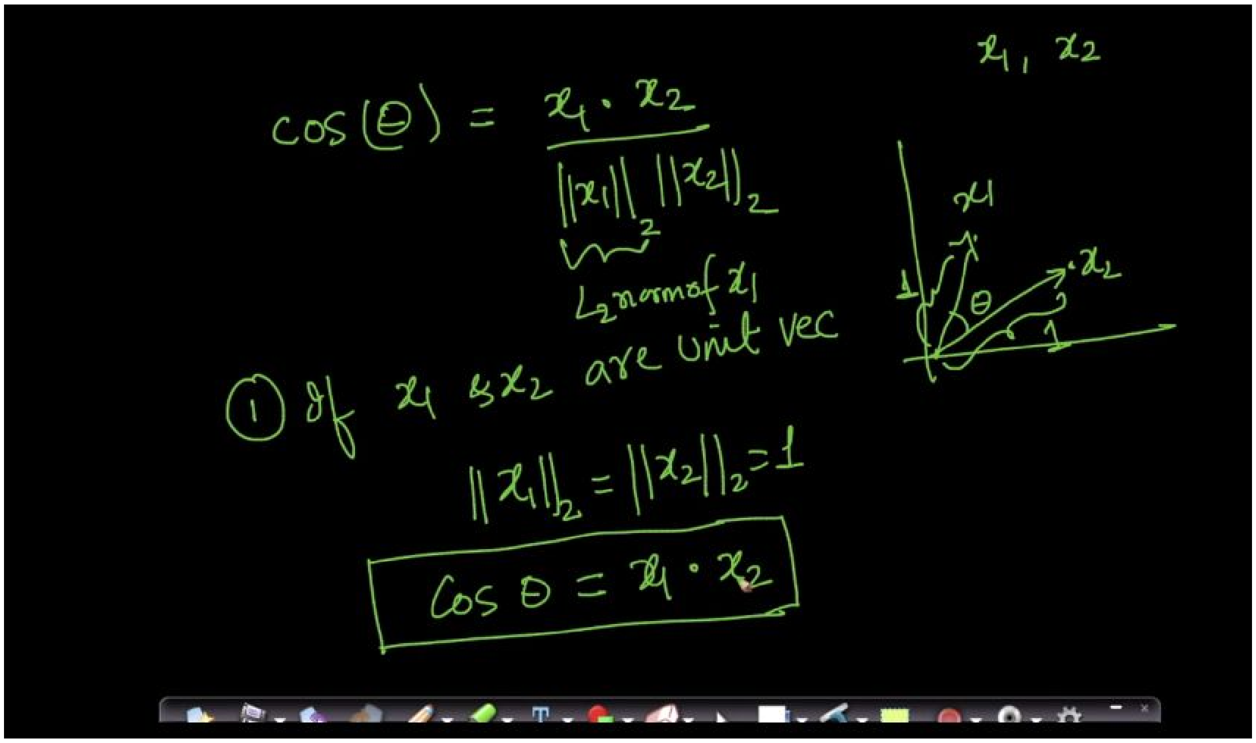
$\text{cos-sim}(x_1,x_2) = \cos\theta$

$\underline{\text{cos-sim}(x_1,x_3) = 1}$

$\theta_{x_1,x_3} = 0°$

euc-dist

$$\begin{cases} d_{13} > d_{12} \\ \text{cos-dist}_{13} < \text{cos-dist}_{12} \end{cases}$$



$x_1,\ x_2$

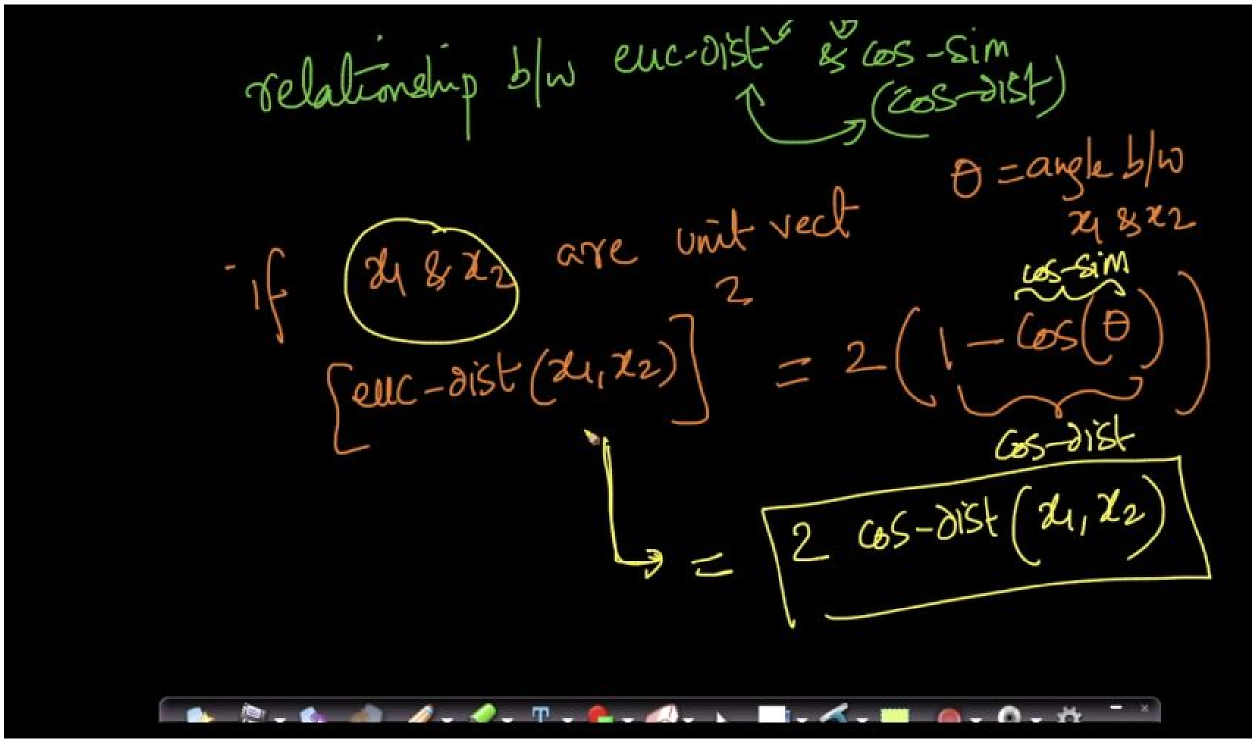
$$\cos(\theta) = \frac{x_1 \cdot x_2}{\underbrace{\|x_1\|_2}_{L_2\text{ norm of } x_1} \|x_2\|_2}$$

① If x_1 & x_2 are unit vec

$\|x_1\|_2 = \|x_2\|_2 = 1$

$$\boxed{\cos\theta = x_1 \cdot x_2}$$

Relationship between euclidean distance and cosine distance:



relationship b/w euc-dist & cos-sim (cos-dist)

θ = angle b/w x_1 & x_2

- if x_1 & x_2 are unit vect

$$[\text{euc-dist}(x_1,x_2)]^2 = 2\Big(\underbrace{1-\overbrace{\cos(\theta)}^{\text{cos-sim}}}_{\text{cos-dist}}\Big)$$

$$= \boxed{2\ \text{cos-dist}(x_1,x_2)}$$

It is the angular distance measure between two points.

We measure the distance, cos(theta) = (x1.x2)/(|x1|.|x2|)   [Refer to Linear Algebra]

In L2 norm, x1=x2=1, i.e., if they are unit vectors, then the distance would be equal to x1.x2.

NB: If x1 and x2 are unit vectors or normalized to unit length, then their relationship with euclidean distance is as follows:
[euclidean distance between x1 and x2] = 2*(cosine distance)
The above relation is derived using the law of cosines between two vectors a and b, which states that, c^2 = a^2 + b^2 + 2ab*cos(theta).
Therefore, the resultant vector,
c = a-b
The dot product of c with itself is,
c.c = (a-b)^2
= a^2 +b^2 -2ab*cos(theta)
= **2*(1- cos(theta))**   [*Since a and b are unit vectors*]
= **2cos-distance(x1,x2)**

Need for cross-validation:

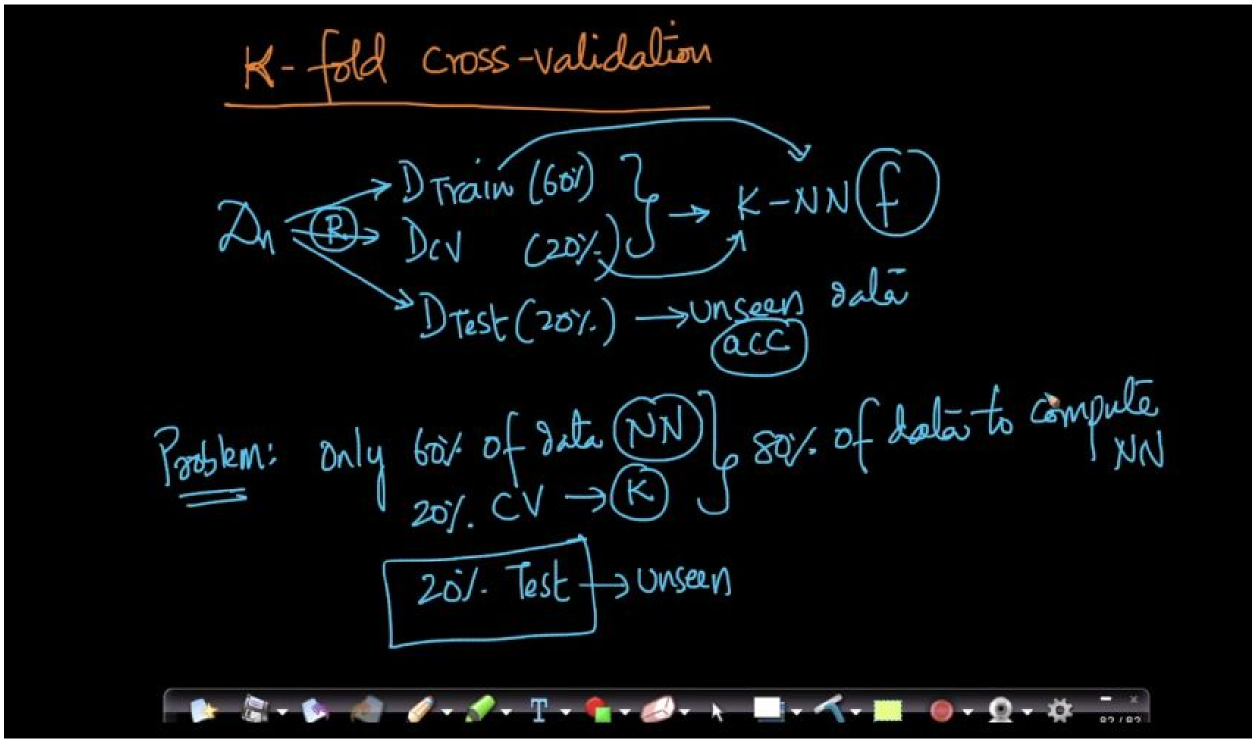
When an algorithm does very well on predicting future unseen points, it is called **generalization**.

Typically, on splitting a Data set into Train and Test set, a problem arises, where we cannot validate the accuracy of unseen points.

Hence to resolve that problem, we split the Dataset into Train, Cross Validation (CV) and Test set, where we primarily train the dataset before cross validating it with the CV dataset and finally we measure the accuracy of the algorithm on the Test set. We call the error percentage of this Test data, generalization error.

K-fold cross validation:

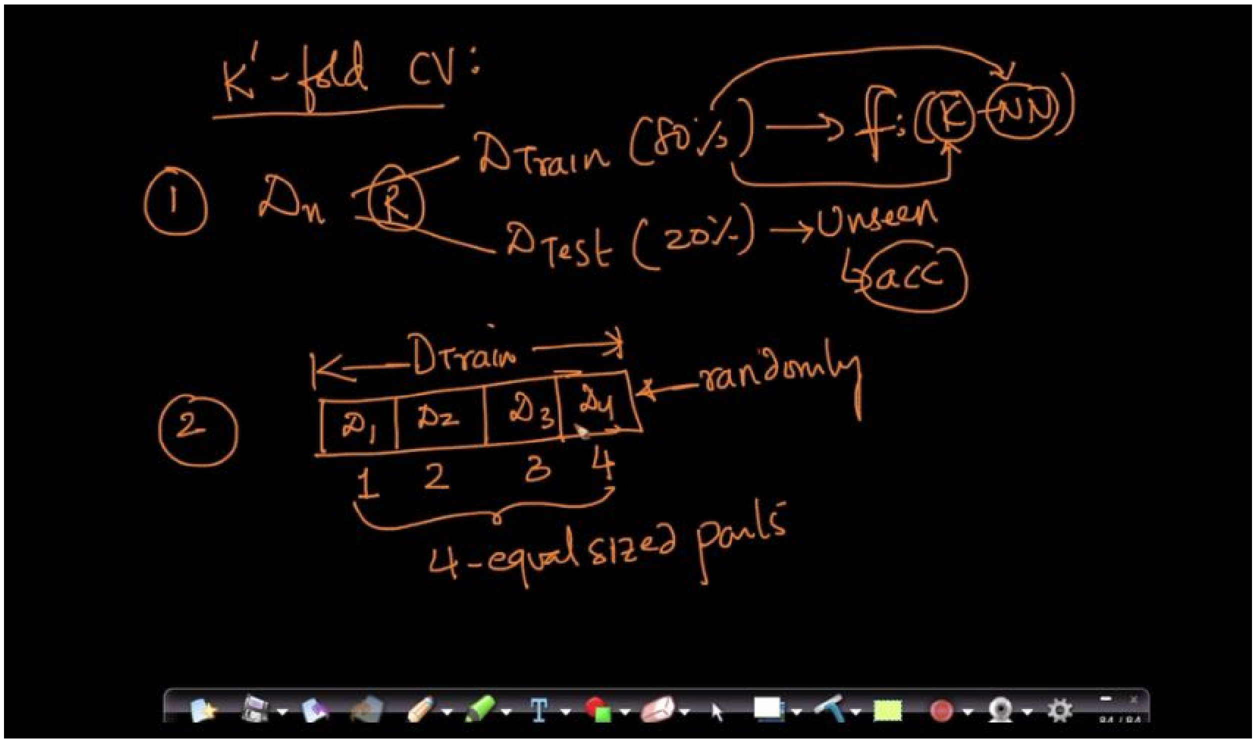
If we split the dataset into Train, CV and Test datasets, we lose a lot of points for training. More the points in the training dataset, the better are the chances of an accurate prediction.

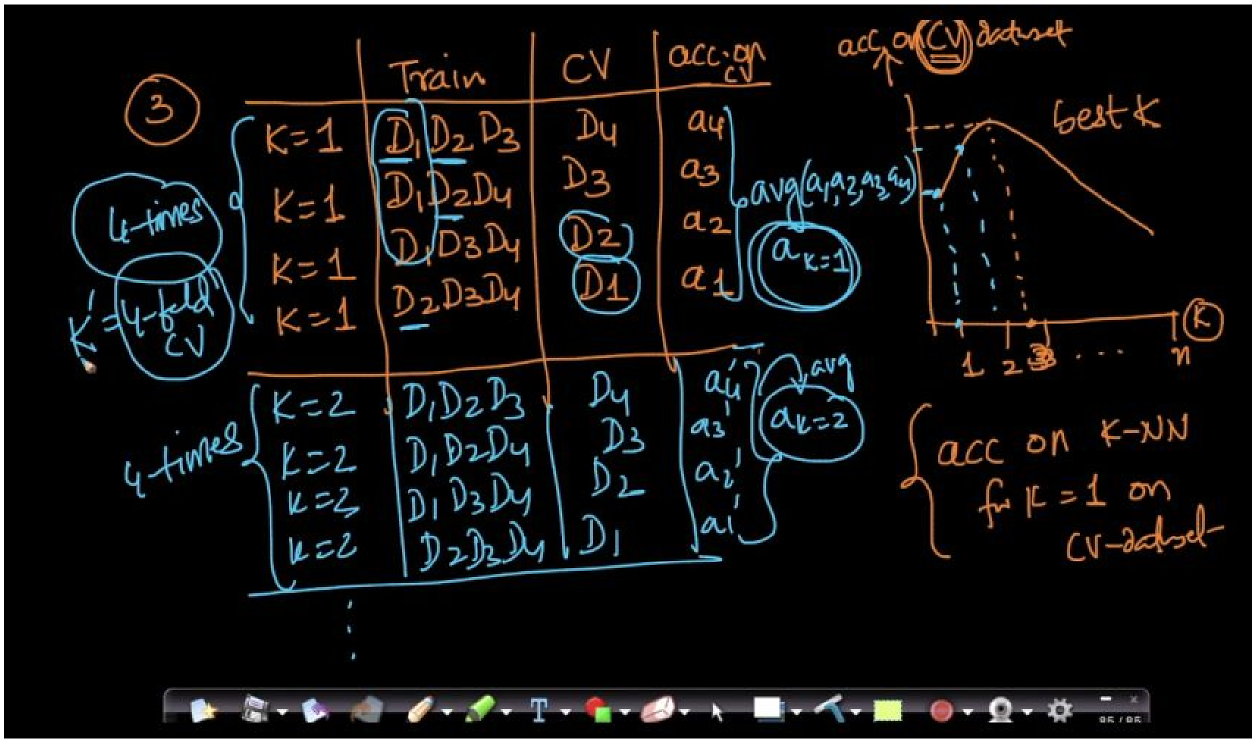


Hence, to counter that problem, we split the dataset into Train and Test.
Then we divide the Train Dataset into K' folds or K' equal halves selecting data randomly.
Then, for each K-value, we take one fold of the Train data for cross validation and the remaining data for training and we permute the process till each of the folds are used as CV.

NB: As a rule of thumb, we select the K' to be 10, or in other words we divide the initial Train data into 10 equal random folds.

The time required to compute the optimal K in KNN increases by K' times if we use K' fold CV.





③

4-times

K' = 4-fold CV

	Train	CV	acc. on CV
K=1	$D_1 D_2 D_3$	D_4	a_4
K=1	$D_1 D_2 D_4$	D_3	a_3
K=1	$D_1 D_3 D_4$	D_2	a_2
K=1	$D_2 D_3 D_4$	D_1	a_1
K=2	$D_1 D_2 D_3$	D_4	a_4'
K=2	$D_1 D_2 D_4$	D_3	a_3'
K=2	$D_1 D_3 D_4$	D_2	a_2'
K=2	$D_2 D_3 D_4$	D_1	a_1'

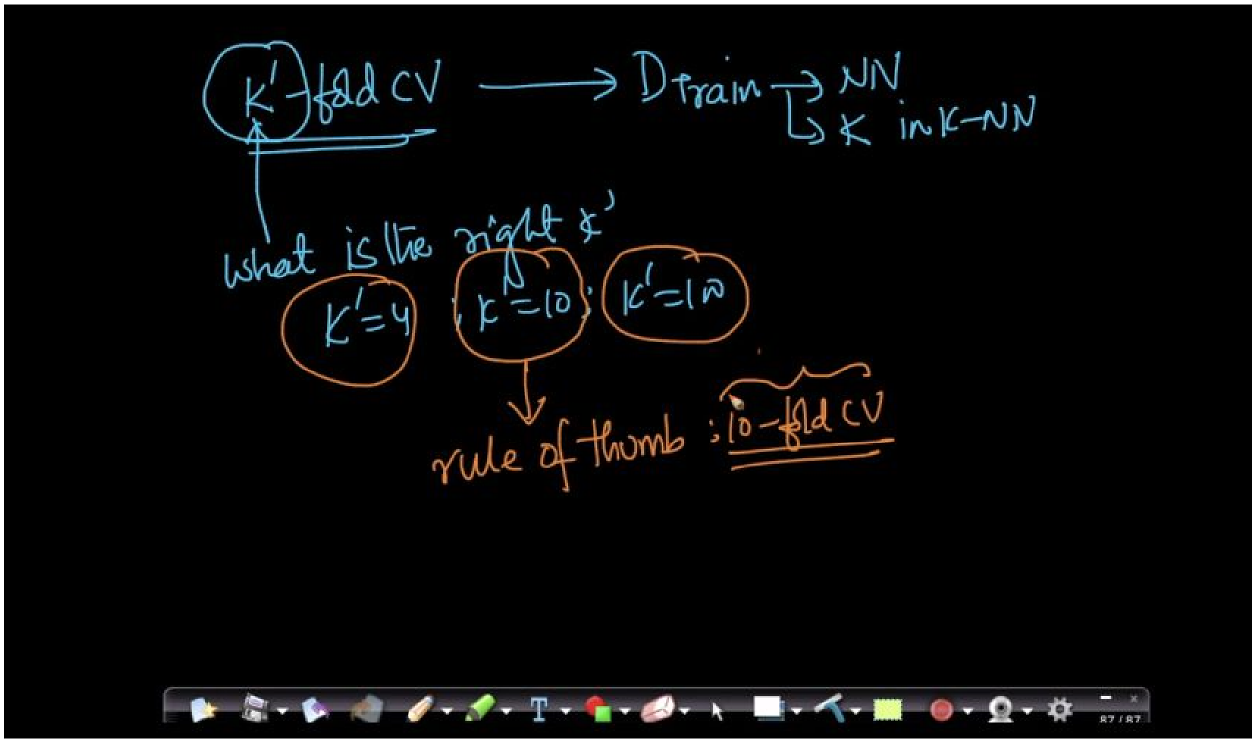
avg(a_1, a_2, a_3, a_4) → $a_{K=1}$

avg → $a_{K=2}$

4-times

⋮

{ acc on K-NN for K = 1 on CV-dataset

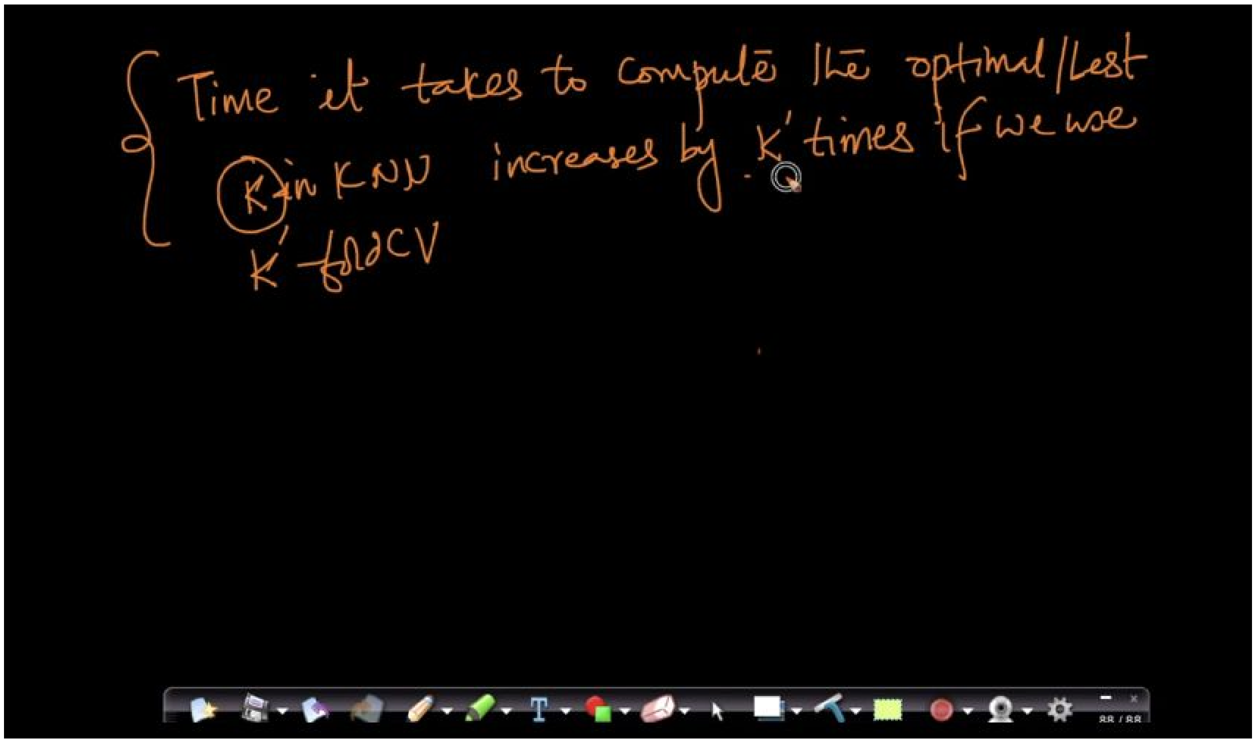


K'-fold CV → D_{train} → NN
↳ K in K-NN

what is the right K'

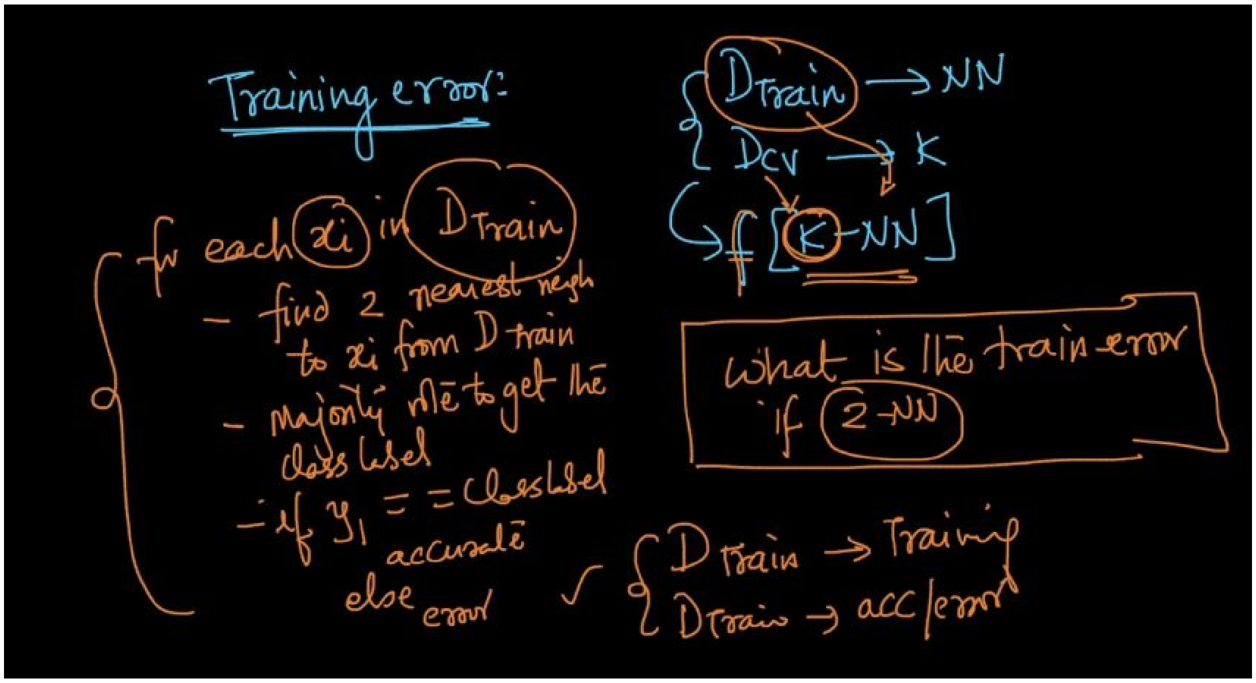
K' = 4 ; K' = 10 ; K' = 100

rule of thumb : 10-fold CV



{ Time it takes to compute the optimal/best K in KNN increases by K' times if we use K'-fold CV

How to Determine Overfitting and Underfitting?

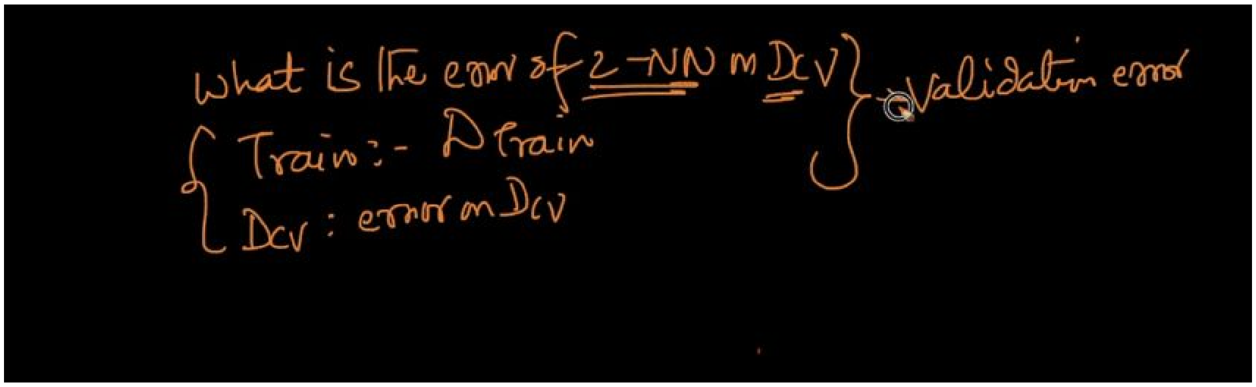

Training error:

for each x_i in Dtrain
- find 2 nearest neigh to x_i from D train
- majority vote to get the class label
- if y_i == class label
  accurate
  else error

✓ { D train → Training
D train → acc / error }

{ Dtrain → NN
Dcv → K }
f [K-NN]

What is the train error if 2-NN

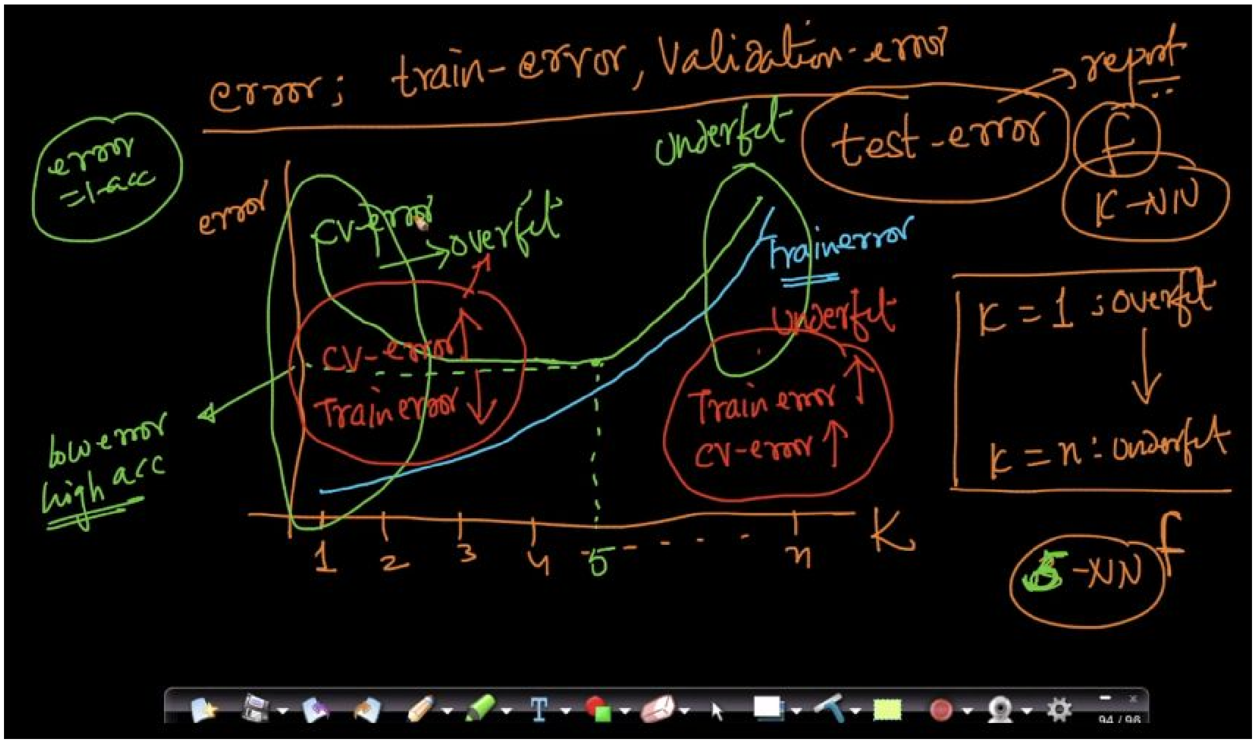

What is the error of 2-NN on Dcv } Validation error

{ Train :- D train
Dcv : error on Dcv }

First, we evaluate the Training error.

Then, we evaluate the CV error.

Then from the K vs error graph, we determine the point where the curve of the CV and train are closest and choose that as the optimal K value.

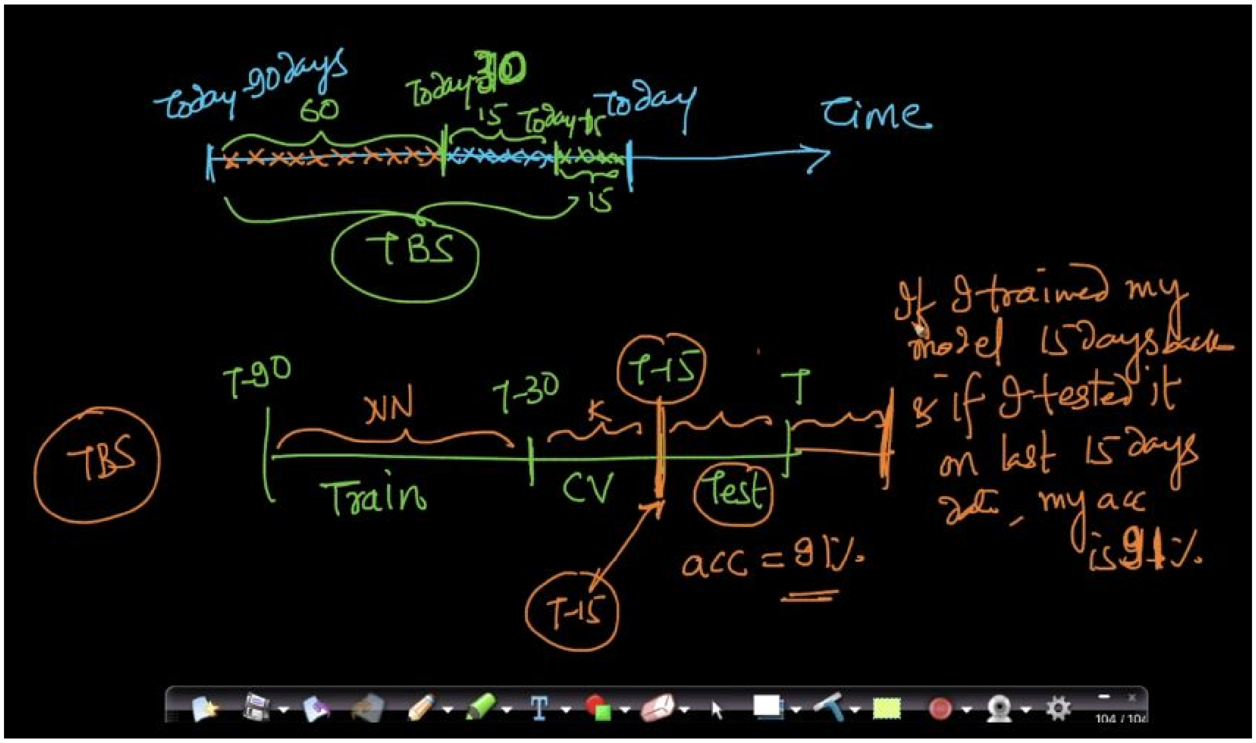


Referring to the graph in Fig 3, the region where Training error is low and the CV error is high, the values of K are overfitted,

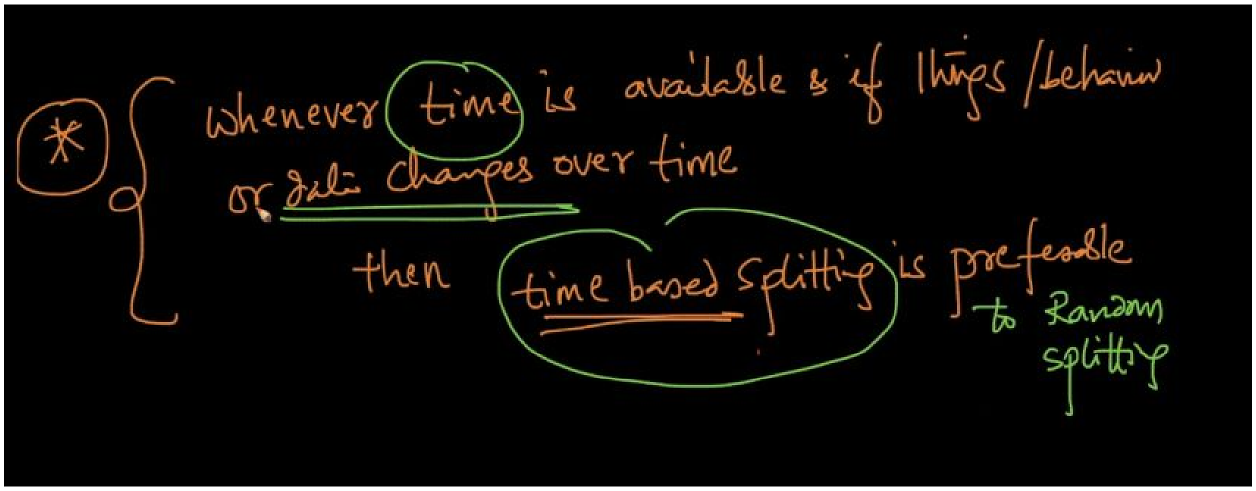
and the region where the Training error is high and so is the CV error, the values of K are underfitted.

Time Based Splitting:

For Time based splitting, the parameter time is a mandatory field.



If I trained my model 15 days back & if I tested it on last 15 days data, my acc is 91%.



Whenever time is available & if things/behaviour or data changes over time then time based splitting is preferable to Random splitting

Primarily, we need to sort the time column in ascending order.

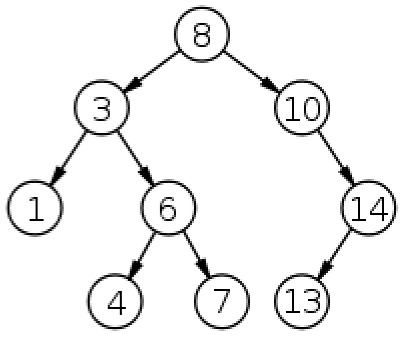
*We use Time based splitting in scenarios like Amazon Fine Food Reviews, **where the review of a product changes with time**. Hence, if we use Random split in circumstances like this, our train, test and CV points can come from anywhere and might cause an inaccurate accuracy when it comes to predicting unseen data points.*

Vornoi Diagram:

In Vornoi diagram, for 1-NN, for a particular point in a certain shade, any point in that shade in comparison to other shades is it's 1-NN.

Refer: https://en.wikipedia.org/wiki/Voronoi_diagram

Binary Search Tree:

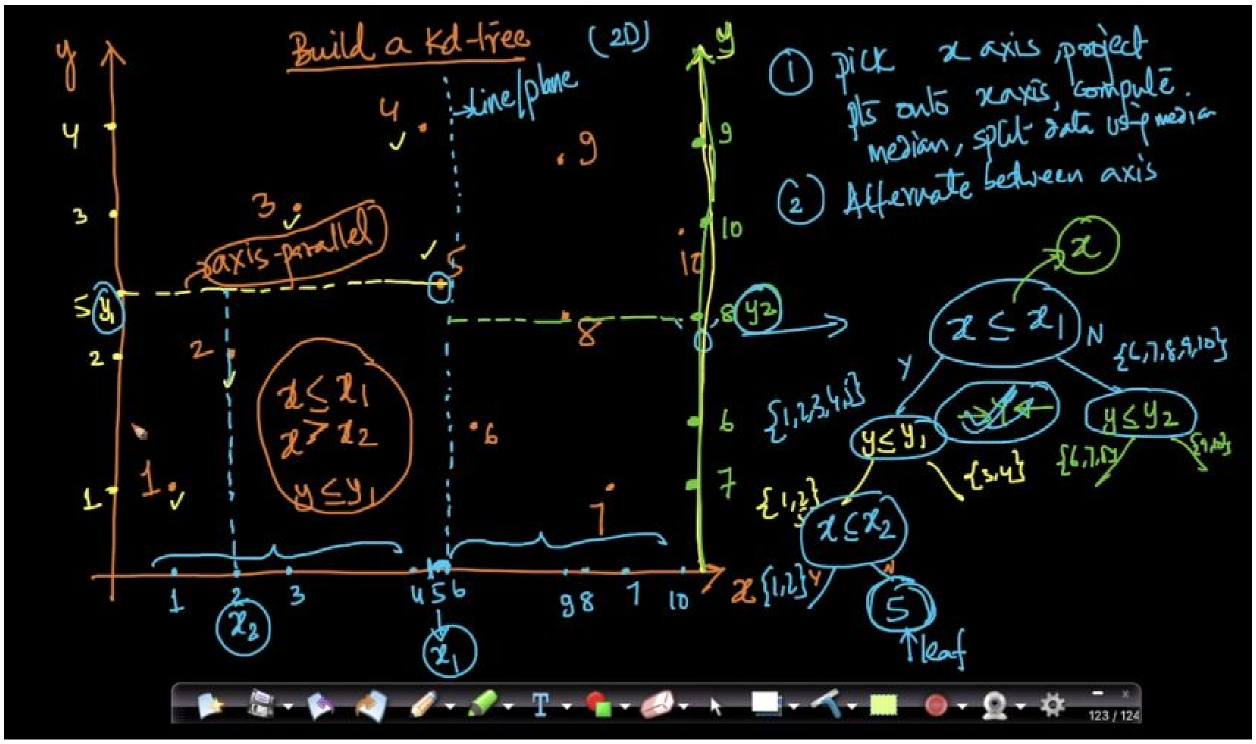


Binary Search Tree or BST, is a Data Structure in CS.
The time complexity of BST is O(logn) while the space complexity is O(n).

To create a BST, we first sort an array into ascending order.
1. Compute the median of the array and use a conditional to divide the elements into two halves.
2. Compute the median of the daughter elements and use a conditional to divide the elements into two halves until each element is resolved into leaves.

The apex of the whole tree is called the root while the resolved elements as the bottom are called leaves.

KD-Tree:

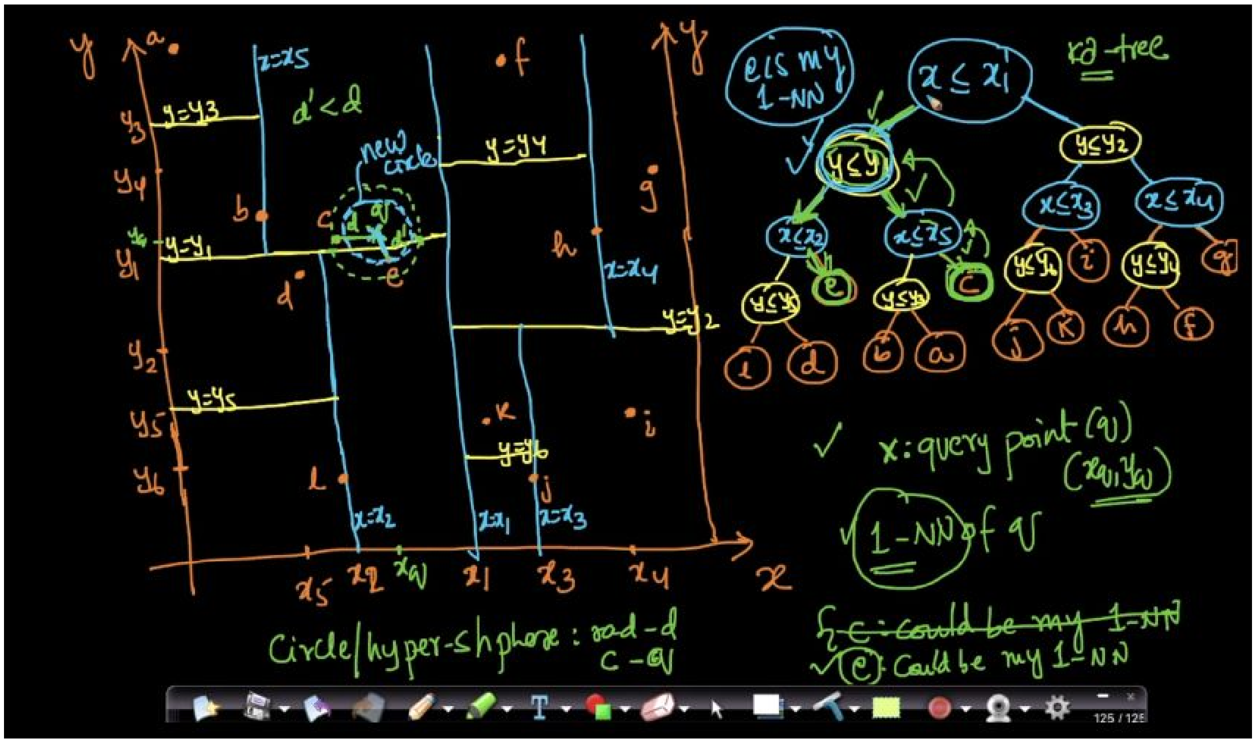


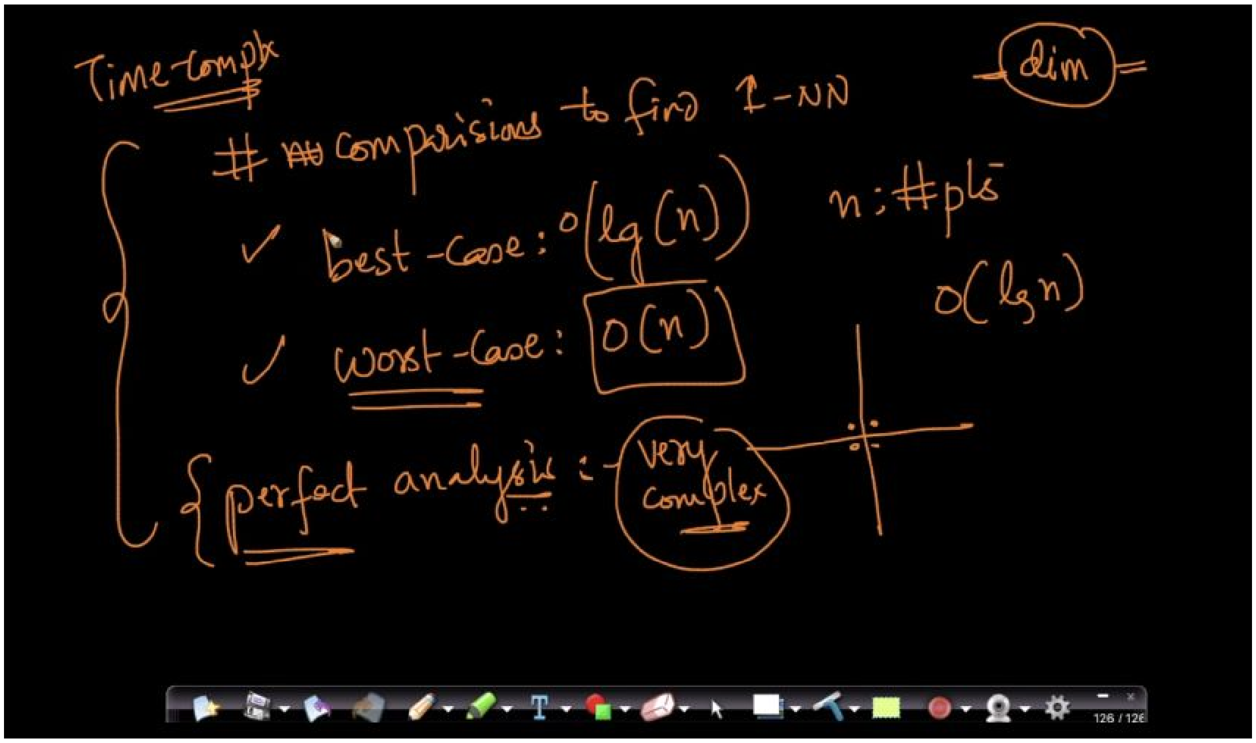
The K in kd-tree refers to k dimensions.
It is a simple technique to split a space into regions fulfilling certain conditions.
To build a kd tree-
1. We take the principle axis (here x).
2. We project our points onto this axis
3. We take the median of the projected points and divide the axis into two regions fulfilling a condition
4. We make the condition the root (or, parent node) of the kd-tree and fill the branches accordingly.
5. Then we take the second axis
6. Repeat steps 2-4
7. We repeat the whole process from 1-6 until the leaf nodes are formed or, in case of higher dimensions, move to the next axis and alternate the whole process.

kd-tree splits a space into regions(cubes, cuboids, hyper-cuboids) which adheres to the conditions of all the successive parent nodes of the region.

https://en.wikipedia.org/wiki/K-d_tree

Find nearest neighbours using kd-tree:





Time-complx

~~no~~ comparisions to find 1-NN

dim

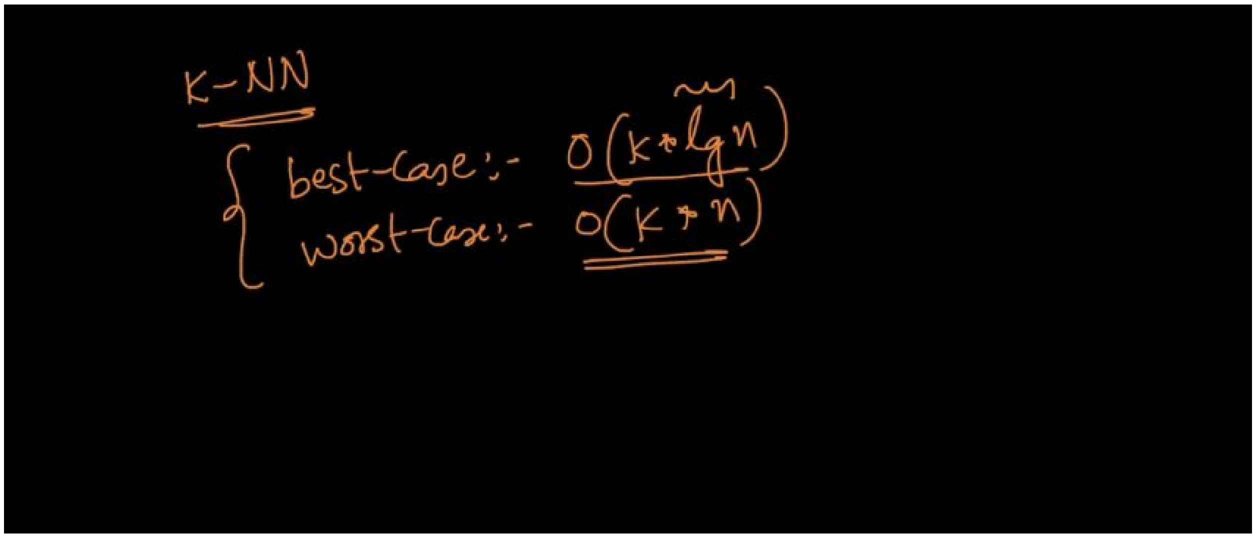
n : # pts

✓ best-case : $O(\lg(n))$

✓ worst-case : $\boxed{O(n)}$

$O(\lg n)$

{ perfect analysis : very complex



K-NN

best-case :- $O(k * \lg n)$

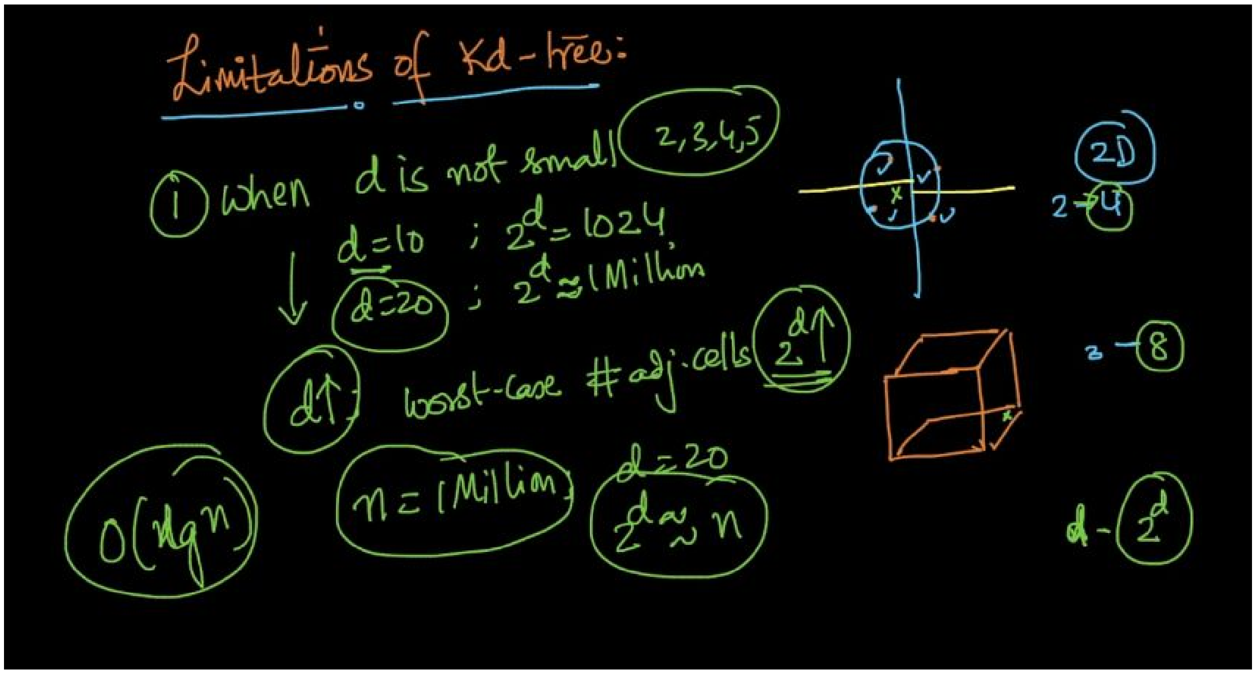
worst-case :- $O(k * n)$

To find the nearest neighbour using kd-tree:

1. Form the kd-tree and it's respective geometrical representation (for visualization)
2. Plot the query point xq on the figure.
3. Iterate through the nodes for which xq satisfies the conditions until a leaf node is found.
4. Save the found node as a possible answer.
5. Draw a circle (or, hyper-sphere) with xq as the centre and the distance d' from xq to the saved node as the radius.
6. If the circle is intercepted by any axis, revert back up the tree till the conditional statement for that axis is met.
7. Repeat steps 3-5.
8. If the distance of the new point is less than d', reject the previous possible solution and accept the new point as the nearest neighbour.
9. Repeat steps 5-8 (if applicable).

The time complexity of finding KNN with kd-tree:
Best case = O(k*log(n))
Worst case = O(k*n)

Limitations of kd-tree:



Limitations of Kd-tree:

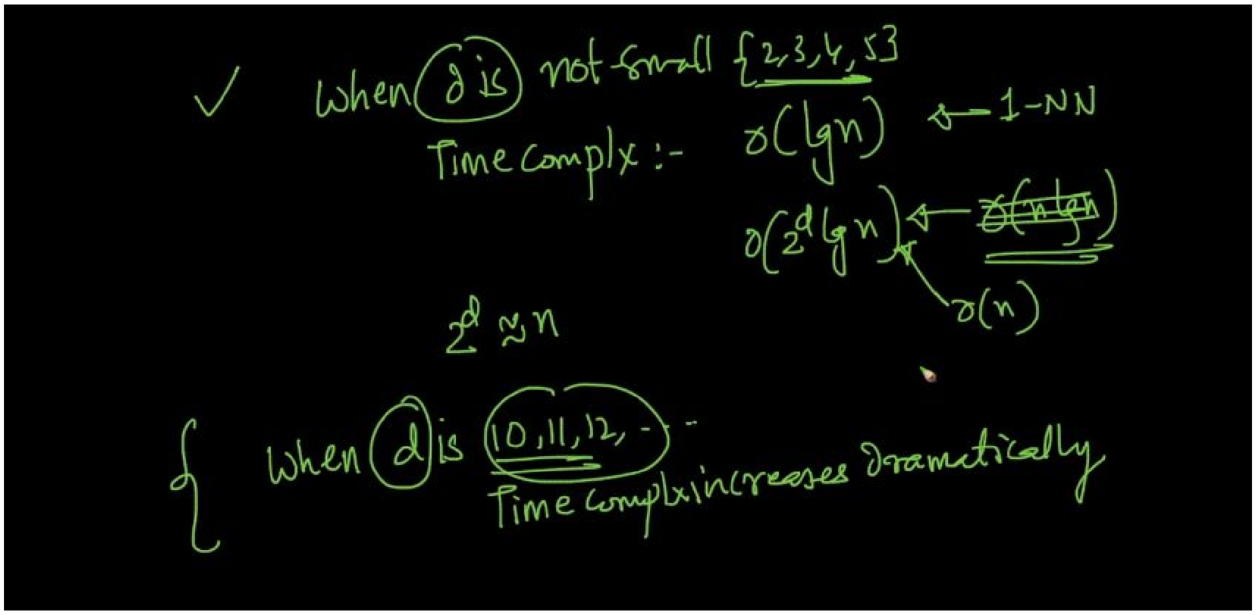
① When d is not small (2,3,4,5)

d=10 ; $2^d = 1024$

↓ d=20 ; $2^d \approx 1$ Million

d↑ ; worst-case # adj. cells 2^d↑

O(lg n) n = 1 Million d=20 $2^d \approx n$



✓ When d is not small {2,3,4,5}

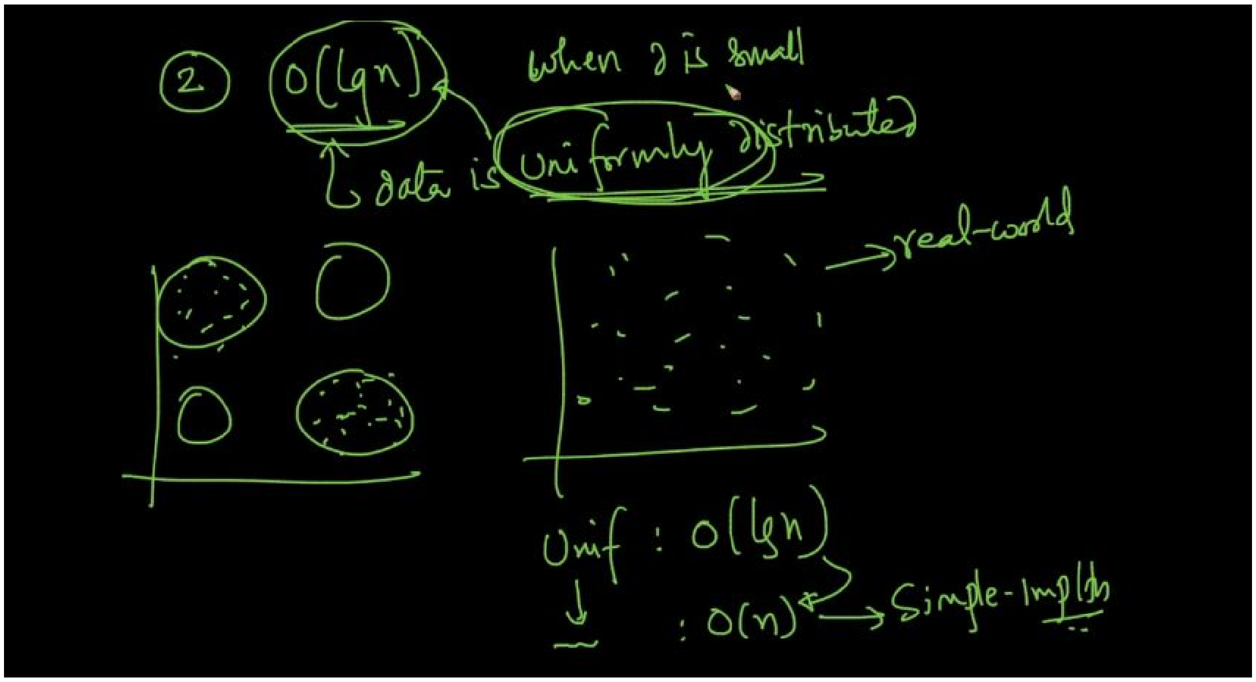
Time complx :- O(lg n) ← 1-NN

O(2^d lg n) ← ~~O(n lg n)~~

← O(n)

$2^d \approx n$

{ When d is 10, 11, 12, ... Time complx increases dramatically



The limitations of kd-tree are as follows:

1. When the dimension (d) of each data point is medium to large, the worst case scenario of the time complexity escalates very quickly to O(2^d * log(n)) as d becomes an exponent of 2. In general ML problems, the value to d is significantly large compared to the permissible limit for kd-tree

2. kd-tree works optimally only if the data points are uniformly distributed. In real world, however that is seldom the case. If the data is not uniformly distributed, the time complexity becomes same as that og general KNN function.

Extensions of KD-tree:

Close variations:

- implicit *k*-d tree, a *k*-d tree defined by an implicit splitting function rather than an explicitly-stored set of splits
- min/max *k*-d tree, a *k*-d tree that associates a minimum and maximum value with each of its nodes
- Relaxed *k*-d tree, a *k*-d tree that the discriminants in each node are arbitrary

Related variations:

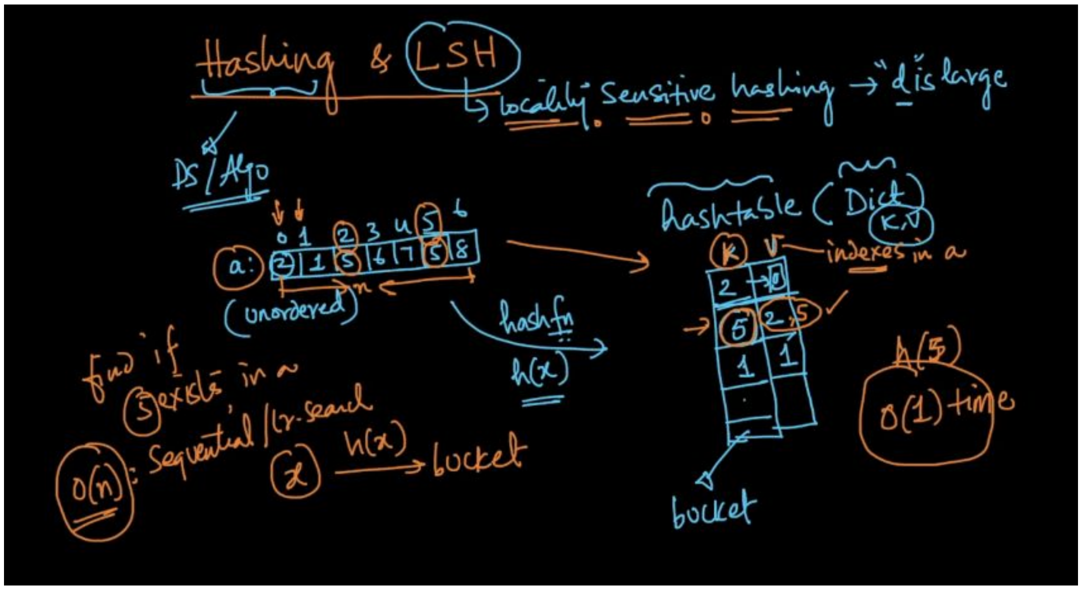
- Quadtree, a space-partitioning structure that splits in two dimensions simultaneously, so that each node has 4 children
- Octree, a space-partitioning structure that splits in three dimensions simultaneously, so that each node has 8 children
- Ball tree, a multi-dimensional space partitioning useful for nearest neighbor search
- R-tree and bounding interval hierarchy, structure for partitioning objects rather than points, with overlapping regions
- Vantage-point tree, a variant of a *k*-d tree that uses hyperspheres instead of hyperplanes to partition the data

Problems that can be addressed with *k*-d trees:

- Recursive partitioning, a technique for constructing statistical decision trees that are similar to *k*-d trees
- Klee's measure problem, a problem of computing the area of a union of rectangles, solvable using *k*-d trees
- Guillotine problem, a problem of finding a *k*-d tree whose cells are large enough to contain a given set of rectangles

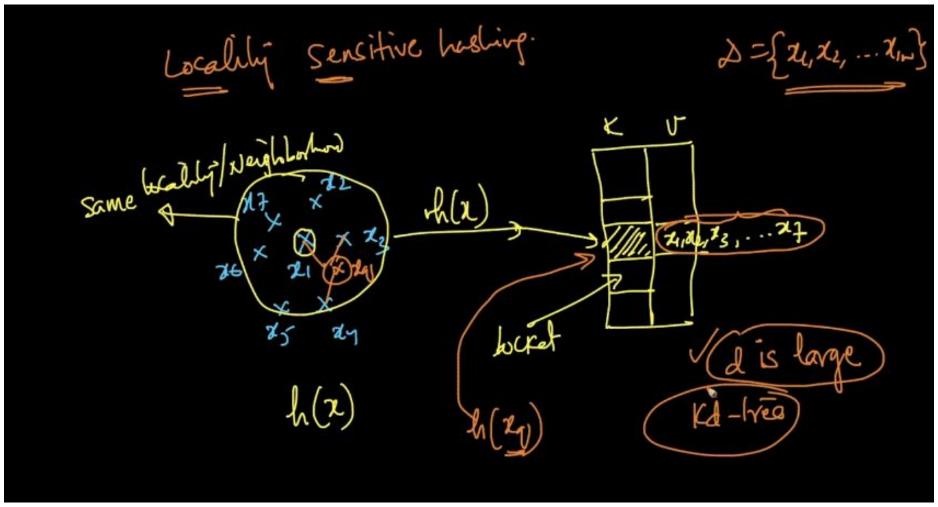
Refer: https://en.wikipedia.org/wiki/K-d_tree

Hashing vs Locality Sensitive Hashing (LSH):



How hashing works:
1. Let us consider an array on n elements
2. Apply a hash function h(x)on x, where x is each element of the array. The hashing function maps the element to an unit cell called bucket which can be accessed by the key(x) and stores the corresponding index of the element from the array as it's value.
3. Whenever an element needs to be accessed, the value of x is passed as the key to the hashing function and the corresponding index of the element is retrieved.
4. The time complexity of using hash tables (or Dict in python) is O(1).

LSH works in a similar way and it applies the hash function to a point and stores it as the key in the bucket while storing the corresponding local nearest neighbours as values.

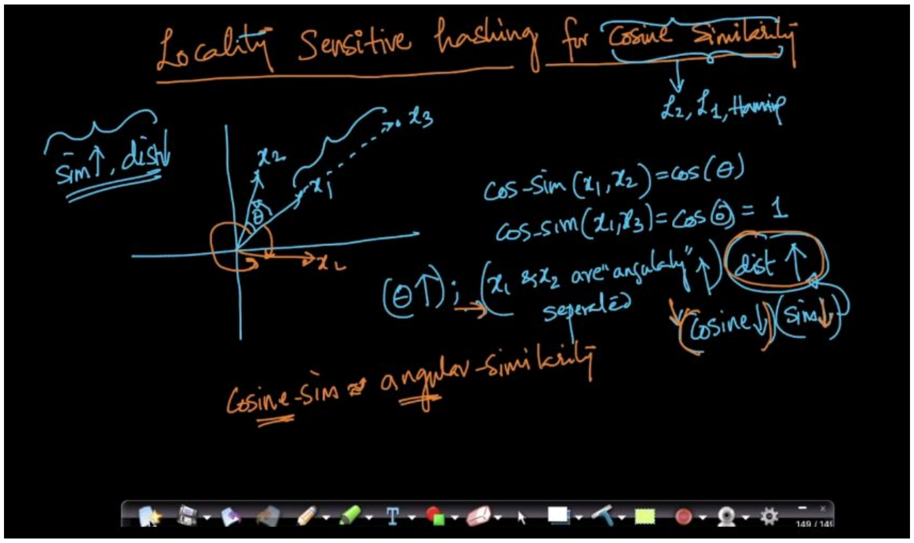


LSH for cosine similarity:

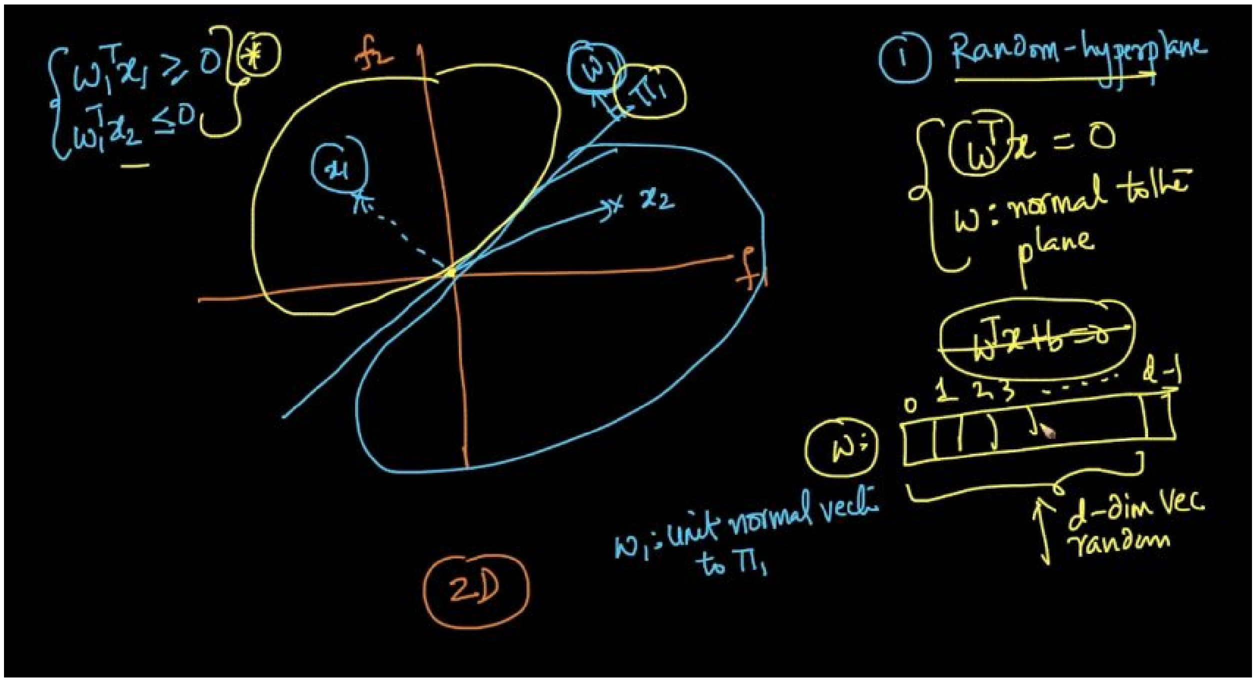
Procedure of performing LSH for cosine similarity:

1. LSH is a randomized algorithm with high probabilistic output but not the exact same for different runs.
2. We start by generating a vector w of d dimensions by randomly sampling values from a normal distribution with mean=0 and std deviation=1 which normally dissects the space.
3. We generate m such hyperplanes and divide the space like pieces of a pie.
4. Then we apply the hash function on each points such that h(x) is of consists of m elements where each element is the position of it respective to the individual planes. We apply the signs of their positions and form the hash value.
5. Points belonging to the same region are put under the same hash value/bucket in a dictionary/hash table.
6. Typically select the hyperplanes as the log function of x, st, number of hyperplanes = log(x)
7. Create multiple (L) such instances of dissecting the space with hyperplanes and creating hash value as key and the corresponding elements as the corresponding value.
8. Make a set union of all these hash values as there will be repetitions/duplicates.

NB: As the number of points increase, slices increase and so does the number of points per slice.



Locality Sensitive hashing for Cosine Similarity
L_2, L_1, Hamming
sim↑, dist↓
x_3
x_2
x_1
θ
$\cos\text{-Sim}(x_1, x_2) = \cos(\theta)$
$\cos\text{-sim}(x_1, x_3) = \cos(0) = 1$
(θ↑); (x_1 & x_2 are "angularly" separated) ↑ dist ↑
Cosine↓ Sim↓
Cosine-Sim ≈ angular-similarity



$$\left\{\begin{array}{l} w_1^T x_1 \geq 0 \\ w_1^T x_2 \leq 0 \end{array}\right\} \boxed{*}$$

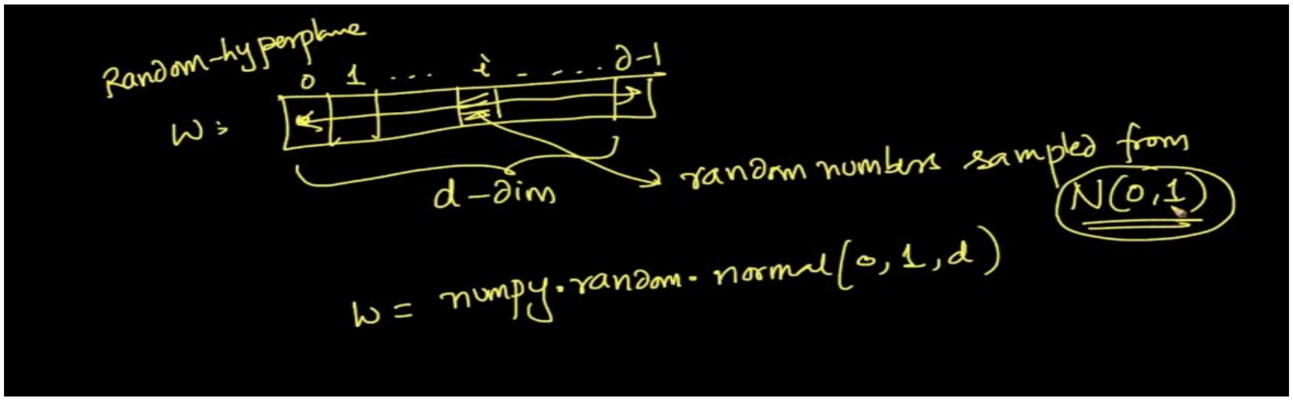
① Random-hyperplane

$$\left\{\begin{array}{l} w^T x = 0 \\ w: \text{normal to the plane} \end{array}\right.$$

~~$w^T x + b = 0$~~

w: d-dim vec random

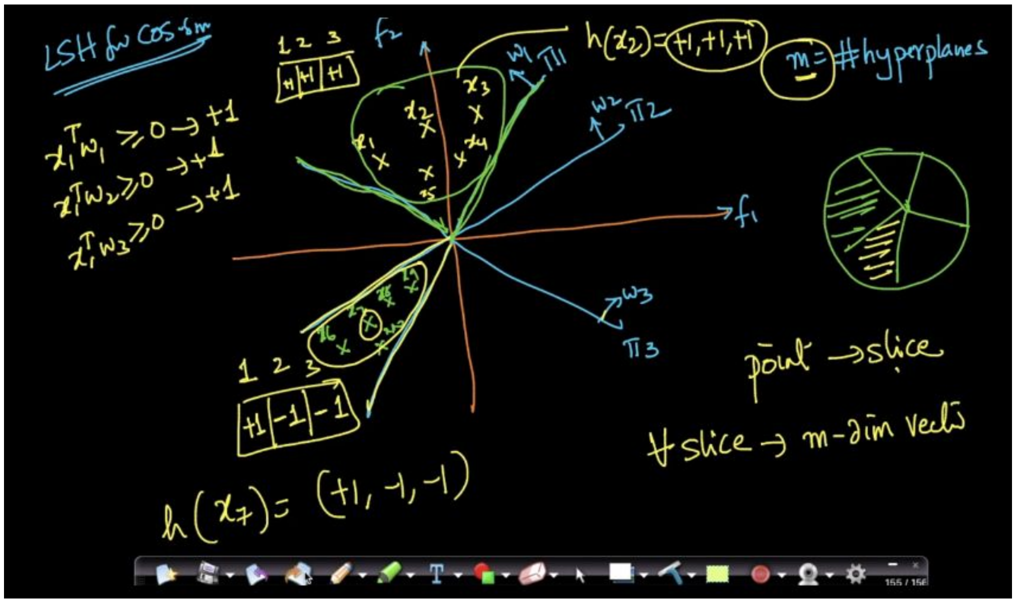
w_1: unit normal vec to π_1



Random-hyperplane

w: d-dim → random numbers sampled from $\underline{N(0,1)}$

w = numpy.random.normal(0, 1, d)



LSH for cos-sim

$x_1^T w_1 \geq 0 \rightarrow +1$

$x_1^T w_2 \geq 0 \rightarrow +1$

$x_1^T w_3 \geq 0 \rightarrow +1$

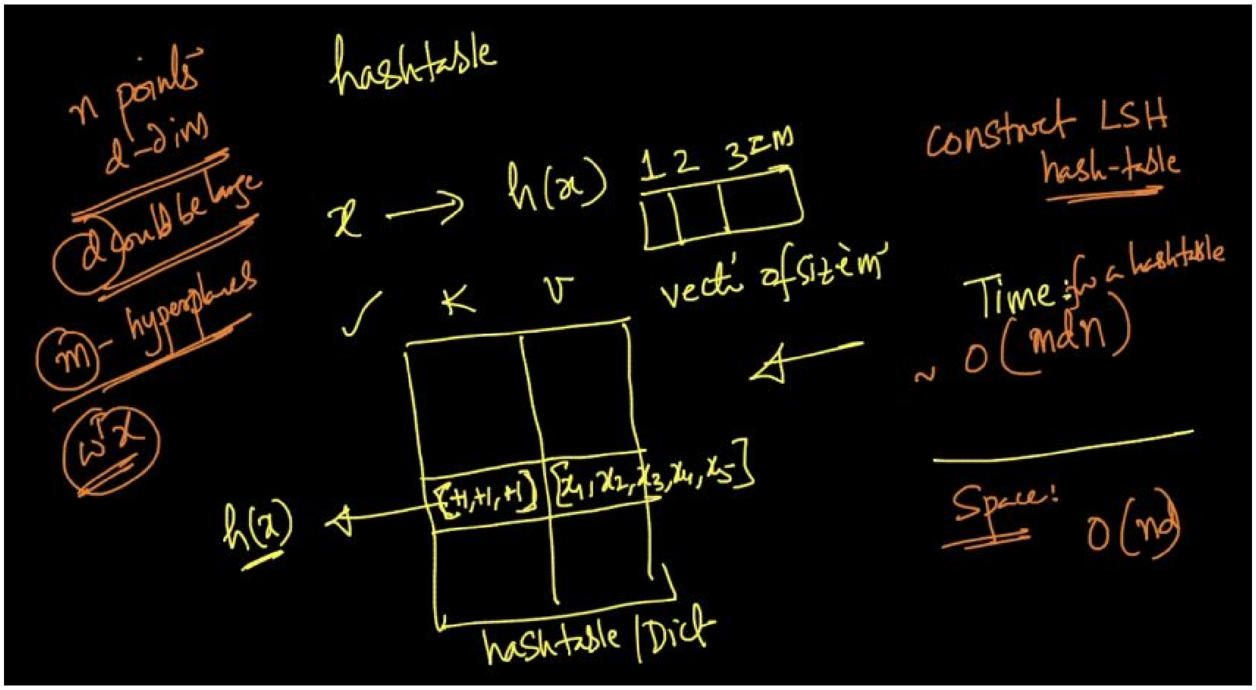
$h(x_2) = (+1, +1, +1)$

$\underline{m}$ = #hyperplanes

point → slice

∀ slice → m-dim vec

$h(x_7) = (+1, -1, -1)$



hashtable

n points

d-dim

d could be large

m - hyperplanes

$w^T x$

$x \longrightarrow h(x)$ 1 2 3 ≈ m

vector of size 'm'

K V

$[+1,+1,+1]$ $[x_1, x_2, x_3, x_4, x_5]$

$h(\underline{x})$

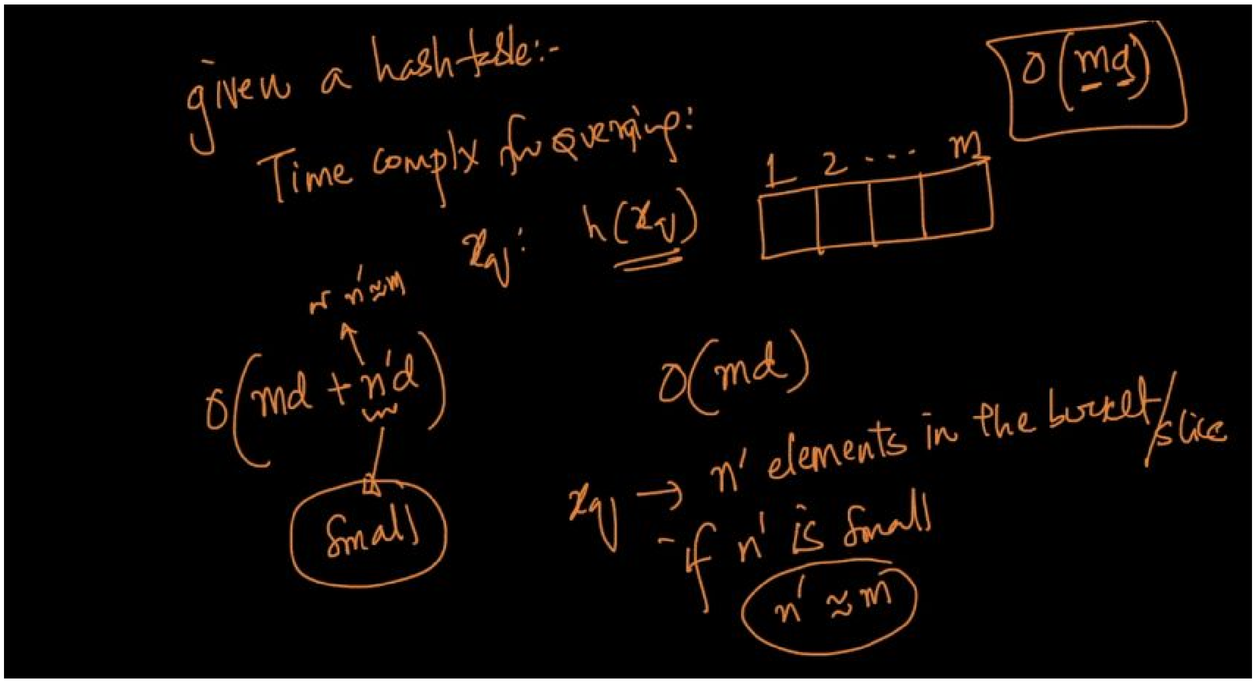
hashtable / Dict

construct LSH hash-table

Time: for a hashtable

$\sim O(mdn)$

Space: $O(nd)$



given a hashtable:-

Time comply for querying:

x_q: $\underline{h(x_q)}$ 1 2 ... m

$\boxed{O(\underline{md})}$

$n' \approx m$

$O(md + \underbrace{n'd}_{})$

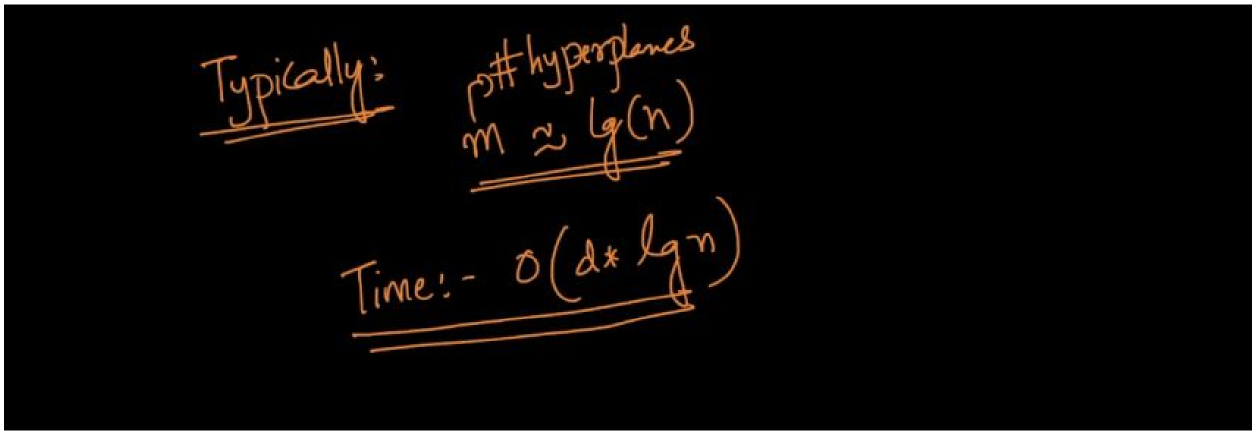
small

$O(md)$

$x_q \rightarrow n'$ elements in the bucket/slice

- if n' is small

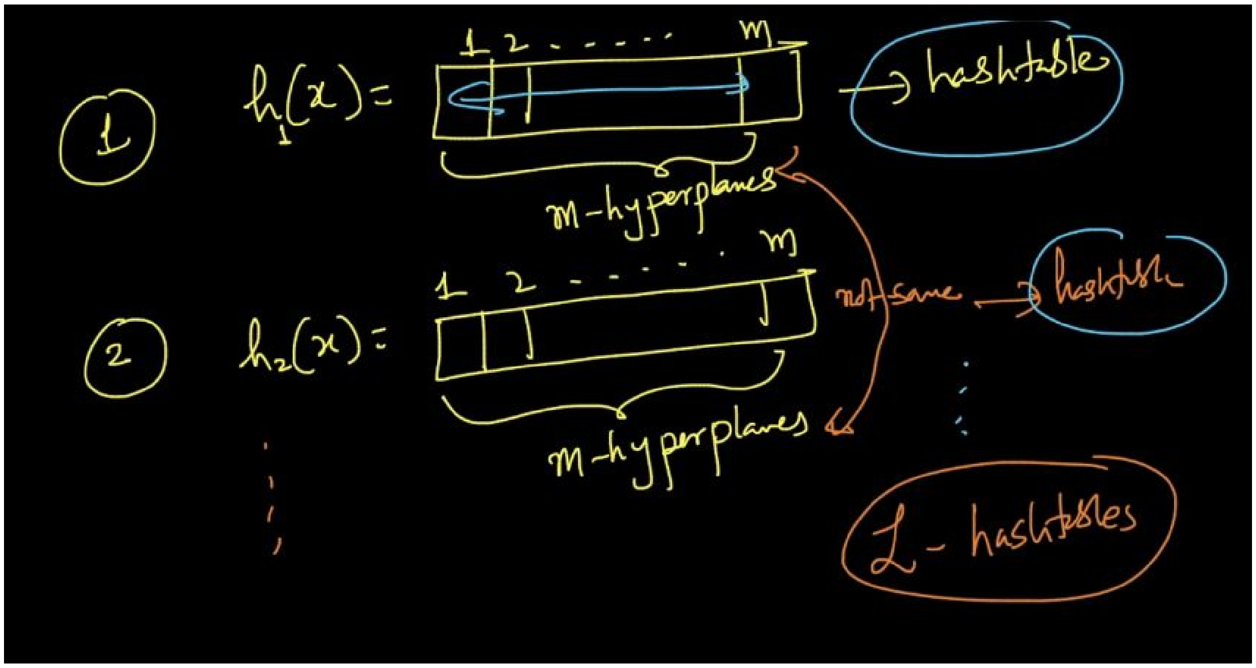
$n' \approx m$

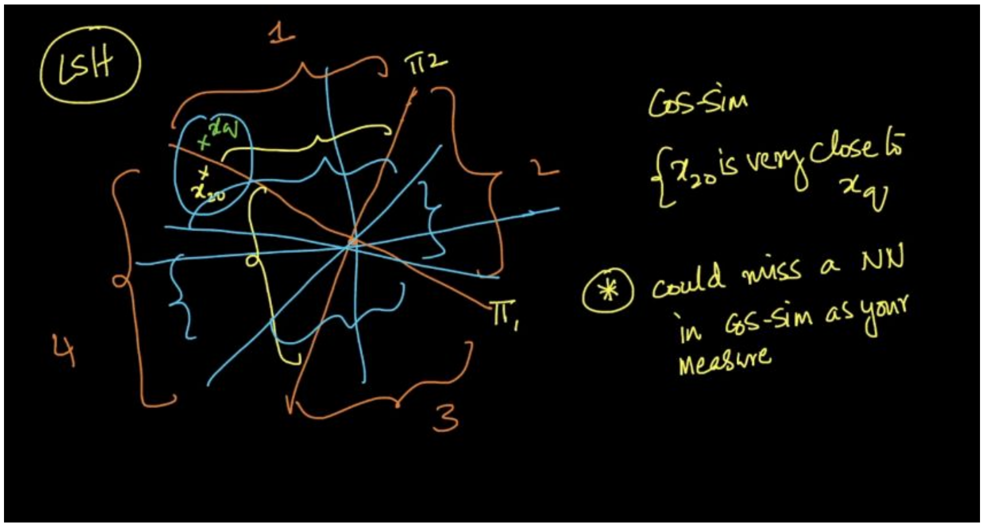


Typically: #hyperplanes

$m \approx \lg(n)$

Time:- $O(d * \lg n)$

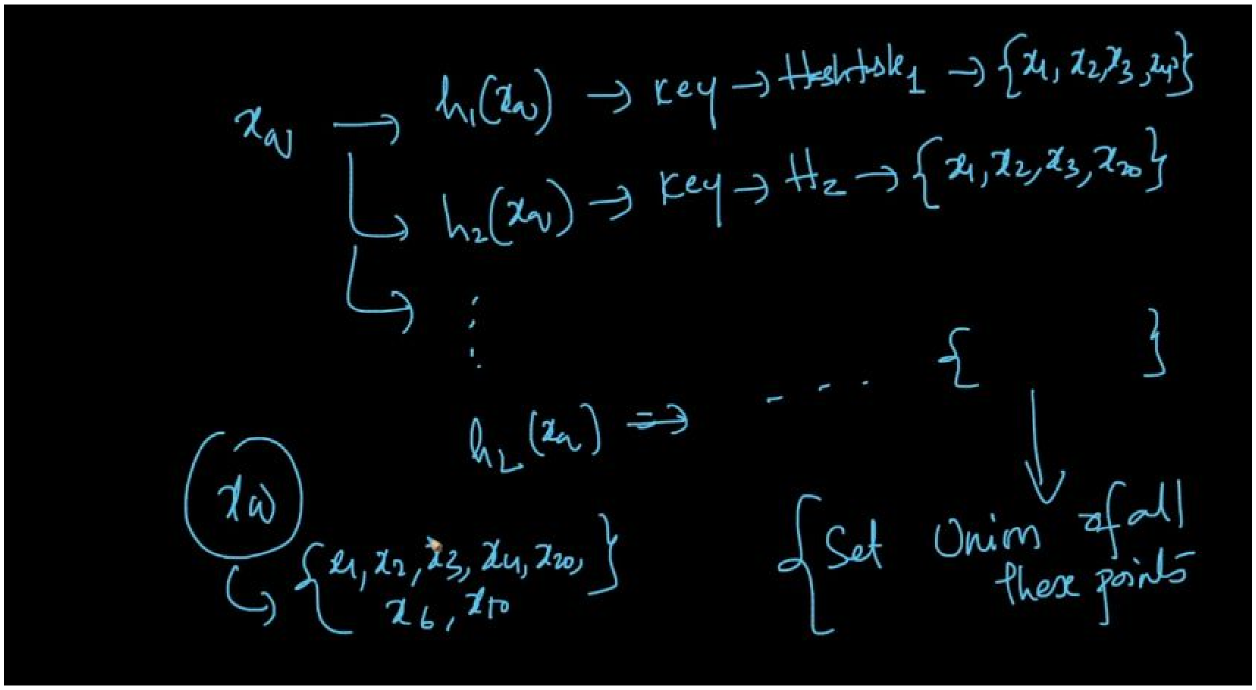




Cos-Sim

$\{x_{20}$ is very close to $x_q\}$

⊛ could miss a NN in cos-sim as your measure



$x_q \rightarrow h_1(x_q) \rightarrow \text{key} \rightarrow \text{Hashtable}_1 \rightarrow \{x_1, x_2, x_3, x_4\}$

$\hookrightarrow h_2(x_q) \rightarrow \text{key} \rightarrow H_2 \rightarrow \{x_1, x_2, x_3, x_{20}\}$

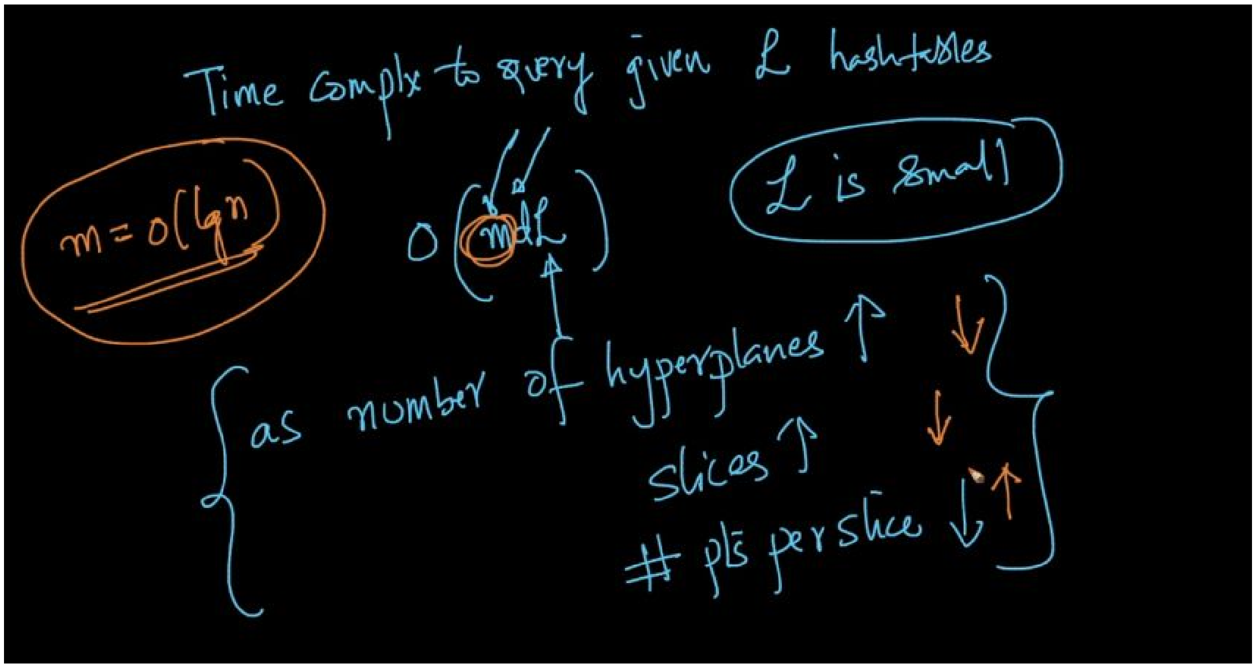
$\hookrightarrow \vdots$

$h_L(x_q) \Rightarrow \ldots \{ \quad \}$

$x_q \hookrightarrow \{x_1, x_2, x_3, x_4, x_{20}, x_6, x_{10}\}$

{Set Union of all these points

Time complexity for given LSH:



Time complx to query given L hashtables

$m = O(\lg n)$

$O(mdL)$

L is small

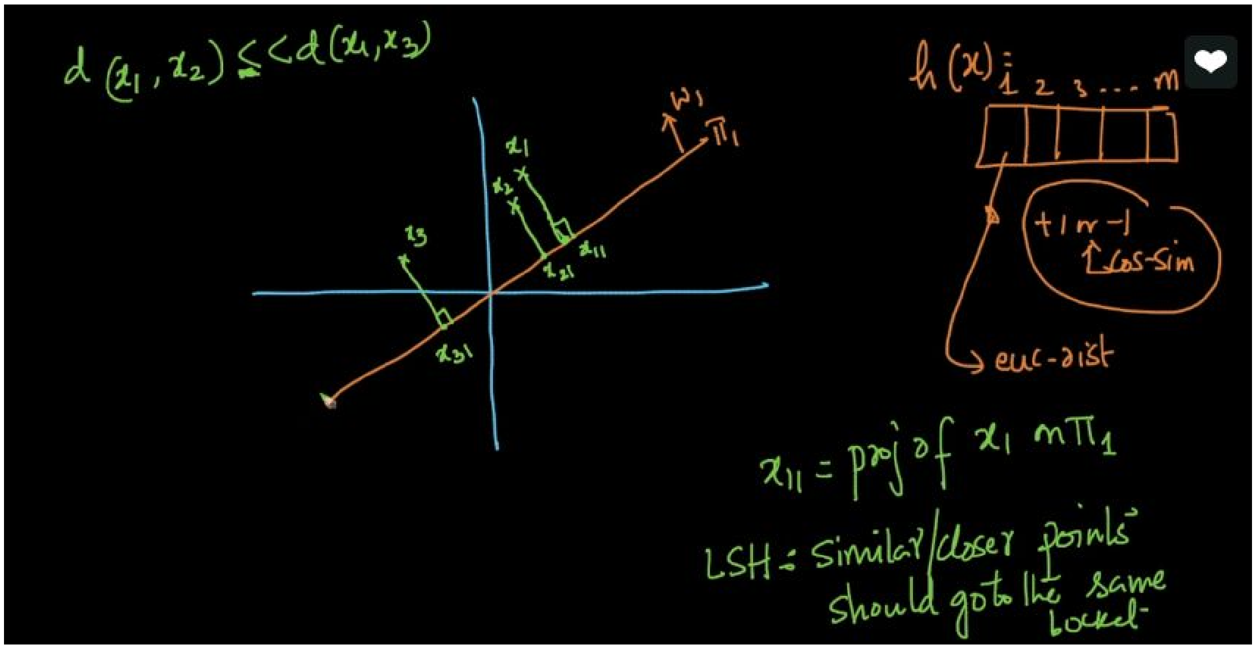
{as number of hyperplanes ↑ ↓
slices ↑ ↓
# pts per slice ↓ ↑}

LSH for Euclidean distance:

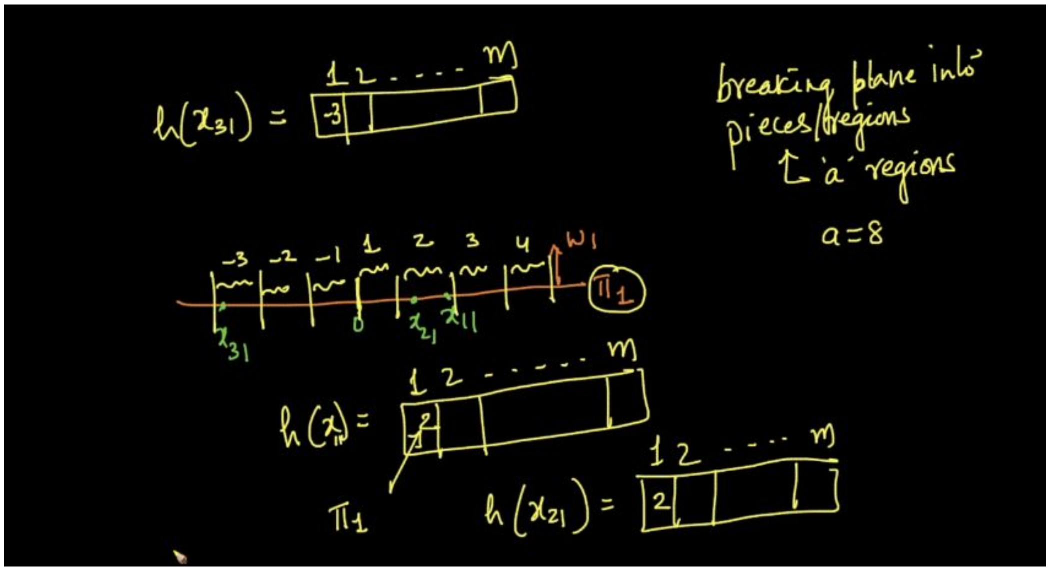
[Refer to LSH for cosine similarity for better understanding.]

Procedure:

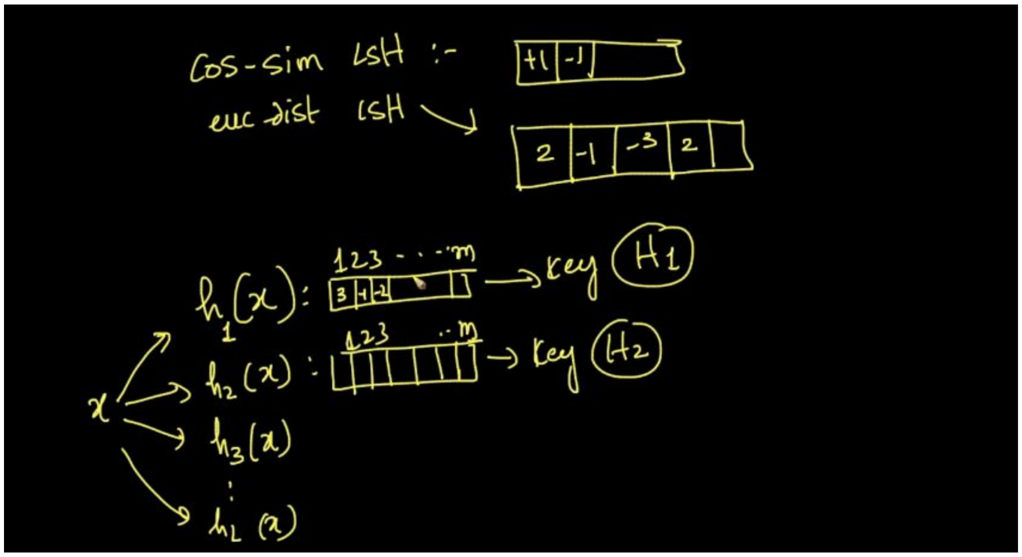
1. LSH is a randomized algorithm with high probabilistic output but not the exact same for different runs.
2. We start by generating a vector w of d dimensions by randomly sampling values from a normal distribution with mean=0 and std deviation=1 which normally dissects the space.
3. Then we project the points on the planes
4. We divide the planes into regions and generate the hash value of a point in regards to the position of the region and we do this for the the respective points of all the planes and derive the hash value for the bucket and the corresponding points as the values of the Hash-table/Dictionary.
5. We do multiple instances of creating planes and take a set union of all such instances of the key, value pair.



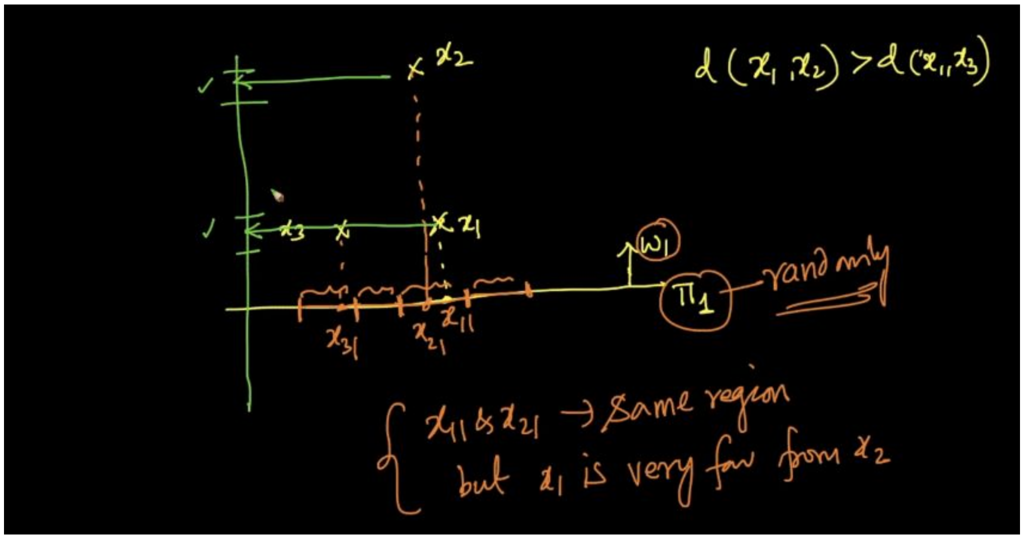
$d(x_1, x_2) \leq\, < d(x_1, x_3)$
w_1
π_1
x_1
x_2
x_3
x_{11}
x_{21}
x_{31}
$h(x)$: 1 2 3 ... m
+1 or -1
cos-sim
euc-dist
x_{11} = proj of x_1 on π_1
LSH: Similar/closer points should go to the same bucket



$h(x_{31}) = $ [-3 | | … |]
1 2 … m
breaking plane into pieces/regions
↳ 'a' regions
a=8
-3 -2 -1 1 2 3 4
w_1
π_1
0
x_{31}
x_{21} x_{11}
$h(x_{11}) = $ [2 | | … |]
1 2 … m
π_1
$h(x_{21}) = $ [2 | | … |]
1 2 … m



cos-sim LSH :- [+1 | -1 |]
euc dist LSH → [2 | -1 | -3 | 2 |]
$h_1(x)$: [3 | +1 | -2 | … |] → key H_1
1 2 3 … m
$h_2(x)$: [| | | | |] → key H_2
1 2 3 … m
x
$h_3(x)$
⋮
$h_L(x)$



$d(x_1, x_2) > d(x_1, x_3)$
x_2
x_3
x_1
w_1
π_1
randomly
x_{31}
x_{21} x_{11}
x_{11} & x_{21} → same region
but x_1 is very far from x_2