Ch2: Classification and Regression Models: K-Nearest Neighbours

Video1: Introduction to Classification

Given a new datapoint, predict which class it belongs to

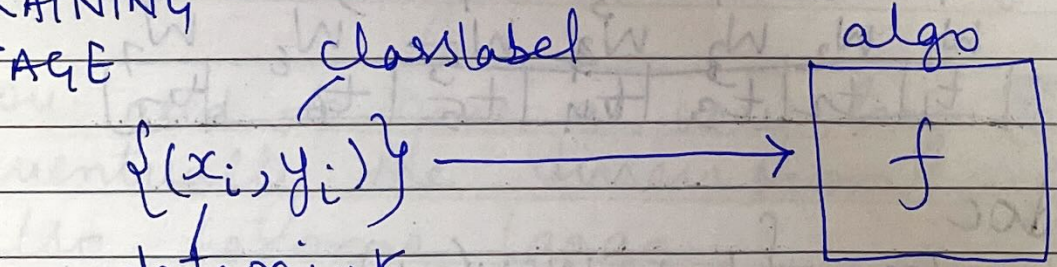
$$y = f(x)$$

x = featurization of query data point.

y = predicted class label

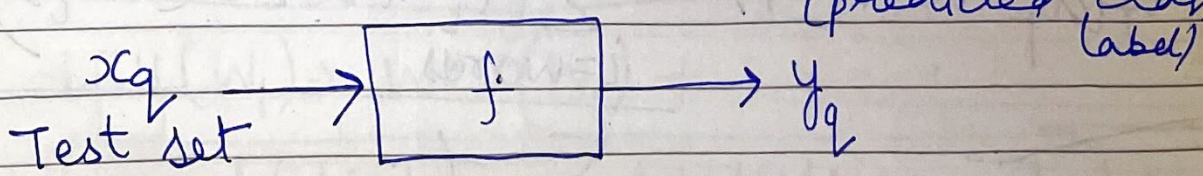
$f()$ = classifier.

TRAINING STAGE

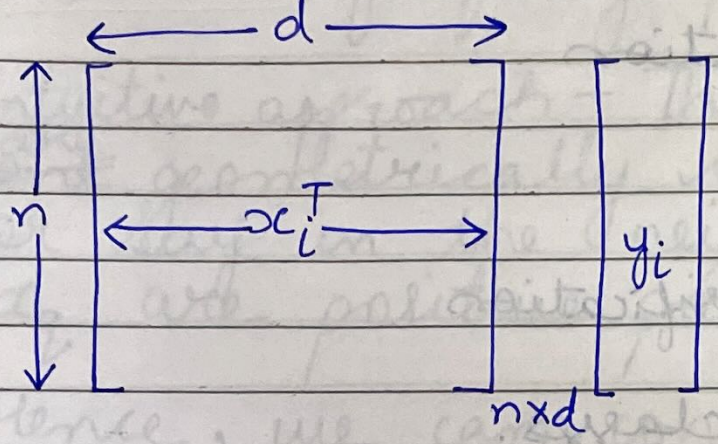


Algo learns from the function & from the training data set.

TESTING STAGE



Video 2 : Notations of Data Matrix



By default, every data point is represented as column vector. So, transpose taken here ∵ we want row vector of it. x_i is column vector so take Transpose for making row

$$D_n = \{(x_i, y_i)_{i=1}^{n} \mid x_i \in \mathbb{R}^d, y_i \in \{0, 1\}\}$$

1 for binary classification

D is a dataset of pair of values, n-such pairs, where the first pair is a d-dimensional vector and second as either 0 or 1.

Video 3: Classification vs Regression

1) Binary classification

$y_i \in \{0, 1\}$

2) Multiclass classification

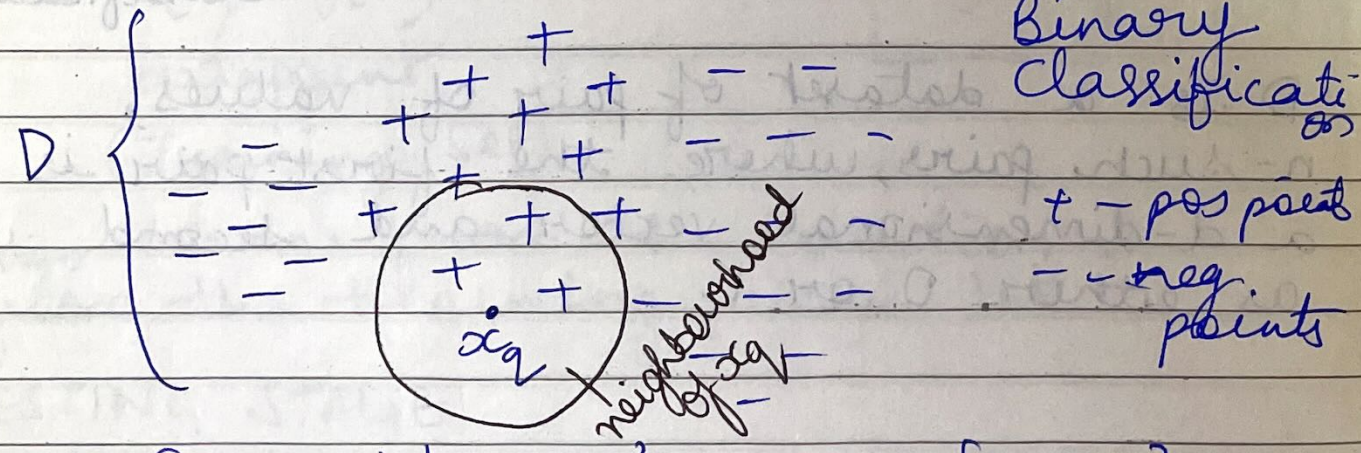
More than 2 classes.

3) Regression

$y_i \in R$ (real value/number)

Video 4: K-Nearest Neighbours Geometric Intuition

2D Toy dataset



$D = \{(x_i, y_i) \mid x_i \in R^2, y_i \in \{0, 1\}\}$

2 in R denotes 2D space.

Objective - To determine class label of query point.

Intuitive approach - The points that ~~are~~ are geometrically close to x_q or say in the neighbourhood of x_q are positive points mostly. Hence, we can conclude that x_q is also positive point.

KNN Algo :-

1) Find K nearest points to x_q in D
Say K = 3 & x_1, x_2 and x_3 are closest to x_q.

$x_1 \rightarrow y_1$
$x_2 \rightarrow y_2$
$x_3 \rightarrow y_3$

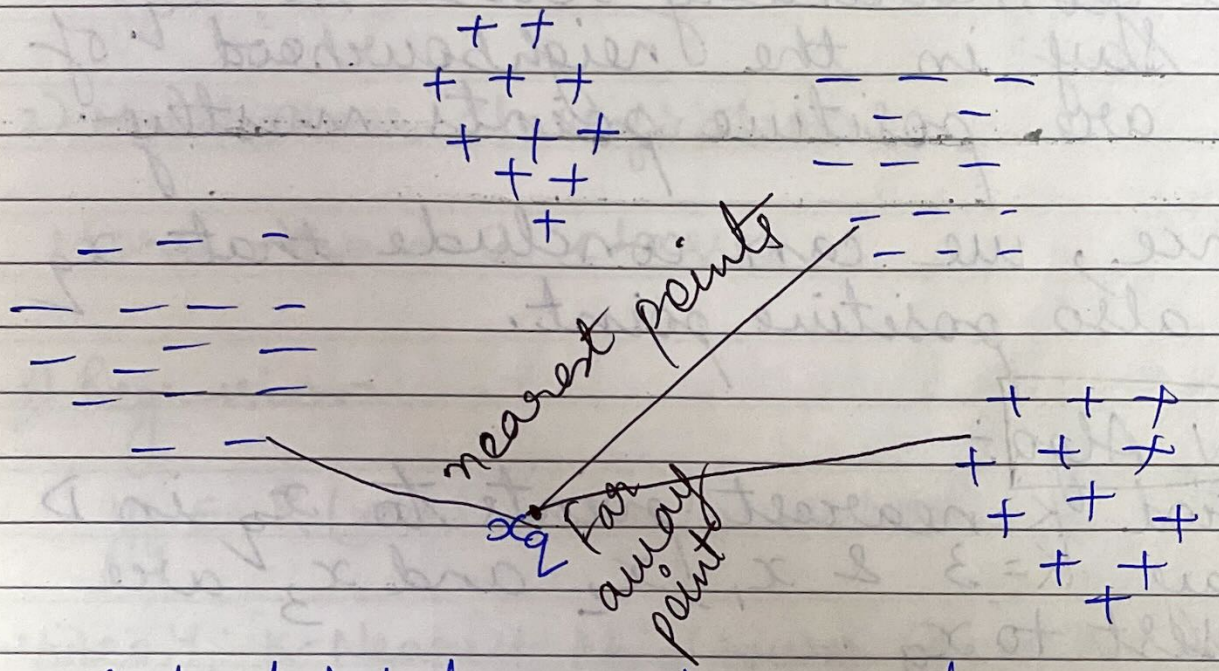
2) Decide class label of x_q using majority vote of y_i's

$\{y_1, y_2, y_3\}$	y_q
+ + +	+
+ + −	+
− + −	−

Note - K should be an odd no.

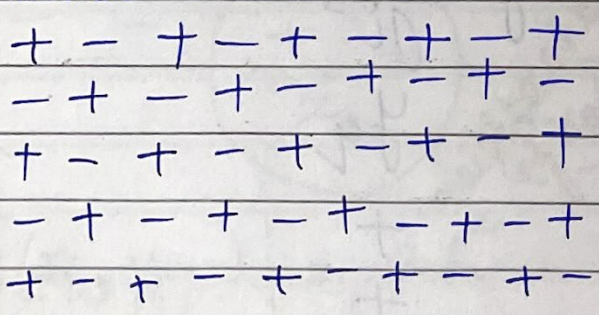
Video 5: Failure cases of KNN

1) query point is far away from clusters.



Output label may say point x_q as -ve ∵ of majority vote but it doesn't make much sense. It would be better if the model could output as - Don't know

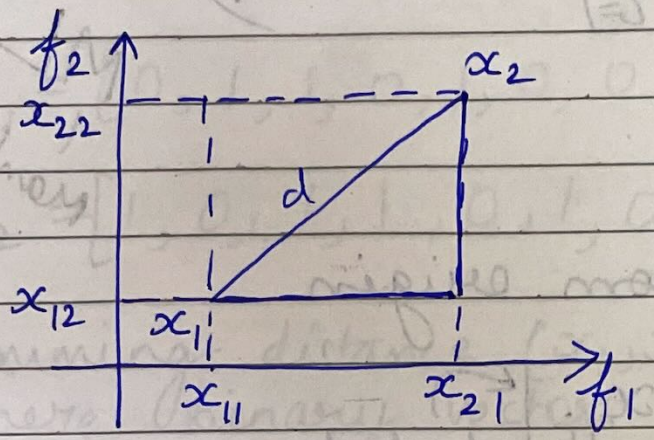
2) +ve and -ve points are randomly spread.



No useful information

Grouped data is useful but random noisy data is not useful.

V6: Distance Measures: Euclidean (L2), Manhattan (L1), Minkowski, Hamming



d = length of shortest line from x_1 to x_2

f_1 f_2

$x_1 = (x_{11}, x_{12})$

$x_2 = (x_{21}, x_{22})$

$$d = \sqrt{(x_{21} - x_{11})^2 + (x_{22} - x_{12})^2}$$

For d-dimensions:-

1) Euclidean distance,

If x_1 & $x_2 \in R^d$

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

L2 norm of $x_1 - x_2$ vector

Distance of x from Origin $= \left(\sum_{i=1}^{d} x_{1i}^2\right)^{1/2}$

Eu L2
Euclidean

absolute value

2) Manhattan distance

$$\|x_1 - x_2\|_1 = \left(\sum_{i=1}^{d} |x_{1i} - x_{2i}|\right)$$

L1 norm of $x_1 - x_2$

ML
|
Manhattan

Distance of x from origin.

$$\|x_1\|_1 = \sum_{i=1}^{d} |x_{1i}|$$ ← absolute

3) Minkowski distance

$$\|x_1 - x_2\|_p = \left(\sum_{i=1}^{d} |x_{1i} - x_{2i}|^p\right)^{1/p}$$

Lp norm of $x_1 - x_2$

For $p \neq 0$ but any no.

For $p=1$ → manhattan
For $p=2$ → Euclidean

Lp norm operates on a single vector whereas distances operate on 2 vectors.

Minkowski – Lp norm
Euclidean – L2 "
Manhattan – L1 "

Distances are b/w 2 pts & Norms are b/w vectors. } Note

4) Hamming distance (b/w boolean vectors)

x_1, x_2 → boolean vectors → like Binary BoW

$x_1 = [0, 1, 1, 0, 1, 0, 0]$

$x_2 = [1, 0, 1, 1, 0, 1, 0]$

Hamming distance (x_1, x_2) ⇒ # of locations where binary vectors differ.

Also, Hamming distance could be used for measuring distance b/w strings.

5th distance after Hamming, Euclidean, manhattan, minkowski.

V7: Cosine distance and Cosine Similarity

Intuitively, when Similarity ↑ distance ↓
& when Similarity ↓ distance ↑

So, by definition,

$$\text{cosine-distance}(x_1, x_2) = 1 - \text{cosine-Similarity}(x_1, x_2)$$

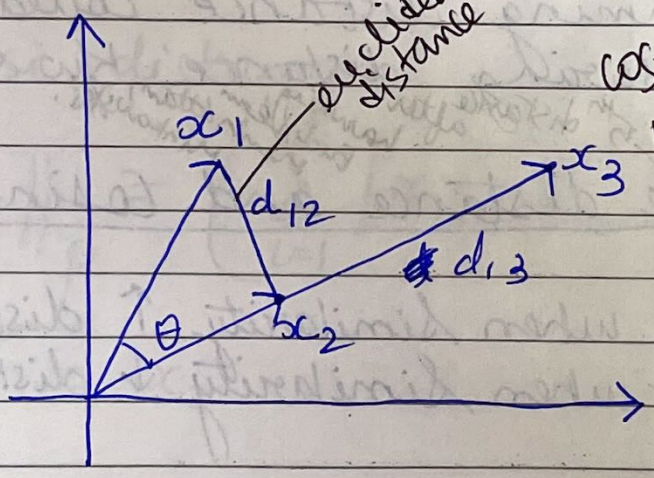
Just as in Euclidean terms, points which have less distance between them are Similar. Similarly, here points that make less angle with each other are Similar.

So, lesser the angle, more should be similarity.

And we know $\cos\theta \propto \frac{1}{\theta}$.

Hence, Similarity $\propto \cos\theta$

So, cos-sim $(x_1, x_2) = \cos\theta$ where θ = angle between x_1 & x_2.



$\cos\text{-sim}(x_2, x_3) = 1$
$\cos\text{-sim}(x_1, x_2) = \cos\theta$

Euclidean $d_{12} < d_{13}$

But $\cos\text{dist}_{13} < \cos\text{dist}_{12}$

So, depending upon problem specification, we may need either of the metrics as the distance measure.

If the problem deals with angular terms, then taking euclidean distance doesn't make

sense and vice-versa.

loop runs, $\cos\theta = \frac{x_1 \cdot x_2}{\|x_1\|_2 \|x_2\|_2}$ default value for any norm is 2. If not put assume 2.

n

if, if x_1 & x_2 are unit vectors,

$\cos\theta = x_1 \cdot x_2$

Relationship between Euclidean and cosine distance (Refer. cosine-similarity wikipedia)

$[euc-dist(x_1, x_2)]^2 = 2 \times cos\,dist(x_1, x_2)$

$= 2 \times (1 - cos\text{-}sim(x_1, x_2))$

if x_1 & x_2 are unit vectors

V8: Measuring Effectiveness of KNN

1) Split D_n into D_{Train} $(n_1) \approx 70\%$

D_{Test} $(n_2) \approx 30\%$

$D_{Train} \cap D_{Test} = \phi$ — null set

$D_{Train} \cup D_{Test} = D_n$

2) Train the KNN model using D_{Train} and measure how well it does using D_{Test}.

Algo:

count = 0
for each point in D_{test}:
  x_q = point
  use D_{train} & KNN to find y_q
  if y_q = y_{point}:
    count = count + 1

$$\text{Accuracy} = \frac{\text{count}}{n_2}$$

where,
count = # pts for which D_{Train} + KNN gave correct class label.

n_2 = # pts. in total D_{test}.

Accuracy is one of the measures. There are many other measures.

V9: Test Time and Space Complexity of KNN

We need to evaluate:
Given x_q determine y_q.
Input: D_{Train}, K, $x_q \in \mathbb{R}^d$
Output: y_q

pseudocode :-

KNN pts = [] — List

loop runs n times:
- for each x_i in D_{Train} :
  - compute $d(x_i, x_q) \rightarrow d_i$ — O(d)
  - Keep the smallest K distances $\rightarrow (x_i, y_i, d_i)$ to KNN pts tuple — O(K)

when K = small say 1 it becomes O(1)

$K << n$. Hence neglected.

So, $O(n * d)$ for above algo.

O(K) (K = small):
count_pos = 0 ; count_neg = 0.
for each x_i in KNN pts:
  if y_i is +ve:
    count_pos += 1
  else
    count_neg += 1

O(1):
if count_pos > count_neg
  return $y_q = 1$ ($y_q \rightarrow$ +ve)
else
  return $y_q = 0$. ($y_q \rightarrow$ -ve)

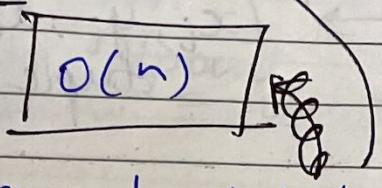
So, total $O(k)$

Overall time complexity

$= O(nd) + O(1) + O(1)$

$= \boxed{O(nd)}$ ✓

If $d \ll n$, $\boxed{O(n)}$



Time takes to compute label of a single query pt.

Space Complexity:

We need to calculate distance for each point in D_{Train}

So, the entire D_{Train} should be present in RAM.

Space $= \boxed{O(nd)}$ ✓

V10 : Limitation of KNN

We need to have a real time system with very low latency:

1) With simple KNN

Time = O(nd)
Space = O(nd)

If $n = 364K$, $d = 100K$ — D_{Train}

nd = 36 GB RAM, means min 64GB RAM needs to be installed.

2) Time Complexity 36 Billion computations

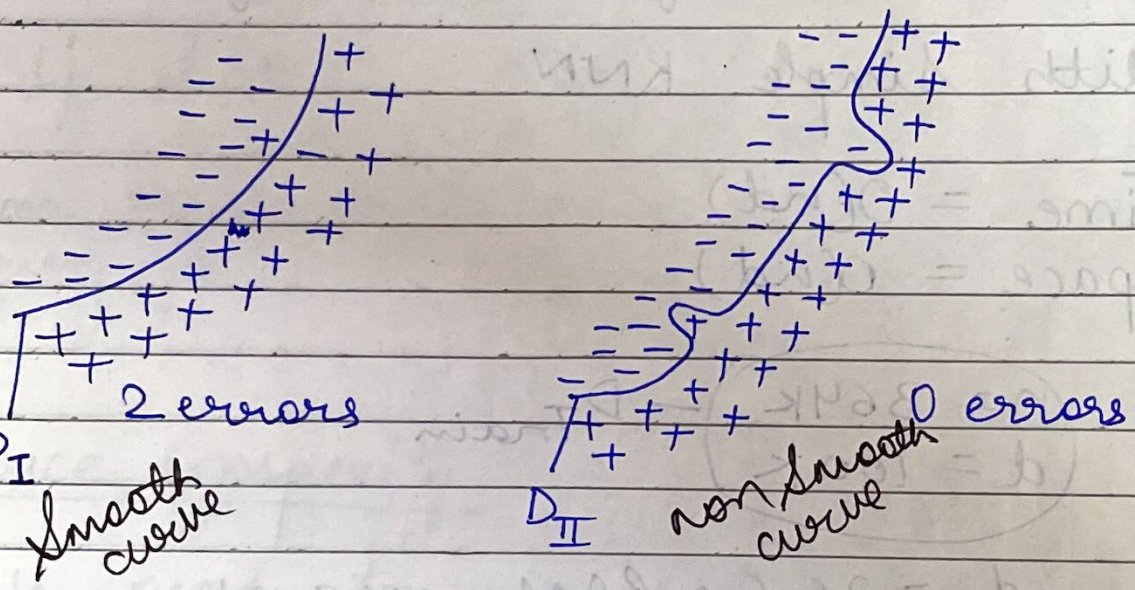
In Internet applications low latency requirement is there.

For trading and finance as much as 1μs delay is permitted only.

Non low latency systems → Medicine etc.
This is one of the main reasons for not implementing KNN in productions and instead of it being so simple algo

VII: Decision Surface for KNN as K changes

K is a hyperparameter



D_I and D_{II} are the decision surfaces. We call them decision surfaces because of generalization of higher dimensions.

The surface decides whether a point is positive or negative depending upon which side of the decision surface the point lies.

But how to algorithmically come up with a decision surface given datapoints?

For KNN decision surface could be found by following:

- Take each point in the space of the datapoints surrounding them.
- Find its k nearest neighbours.
- Find the majority vote and assign that label to that point.
- Now draw a surface that separates both kind of points.

Let k=1
For the given pattern of datapoints distribution as above, if we take a consideration that a point is close to the outlier points, since we are considering 1 closest neighbour, the label of that point will also be same as of outlier point. Thus giving a skewed surface as D_{II}.

But if k=5, even if the sample point is close to outlier and due to majority voting of remaining 4 nearest neighbours that point will be labelled as opposite of outlier giving surface as smooth as D_I.

So, the smoothness of the decision surface increases as k increases.

If say $k = n_1 = 1000$

$n_1 + n_2 = n$

n_1 = no. of +ve points 600

n_2 = " " -ve " 400

and $n_1 > n_2$

Then irrespective of whatever point we take, its label would be positive ∵ of imbalanced dataset.

When K becomes n x_q becomes majority class's y label.

V12: Overfitting & Underfitting

1) Overworking/Overfitting System

If $k=1$

This model overworks to get more complicated decision surface and goes to ensure no errors.

It may take noise as data & classify new points incorrectly.

Overfit

2) Well fit/Robust System

If $k = 5$

It tries to find a balance b/w both extremes.
It is less prone to noise. Makes more sensible decisions.
Appropriately fit function

3) Lazy/Underfitting System

If $k = n$
It checks which class has more data points and assigns that label to the new query point.
Underfit.

V13: Need for Cross Validation

$k = 1, 2, 3, \ldots, n$
overfit — underfit

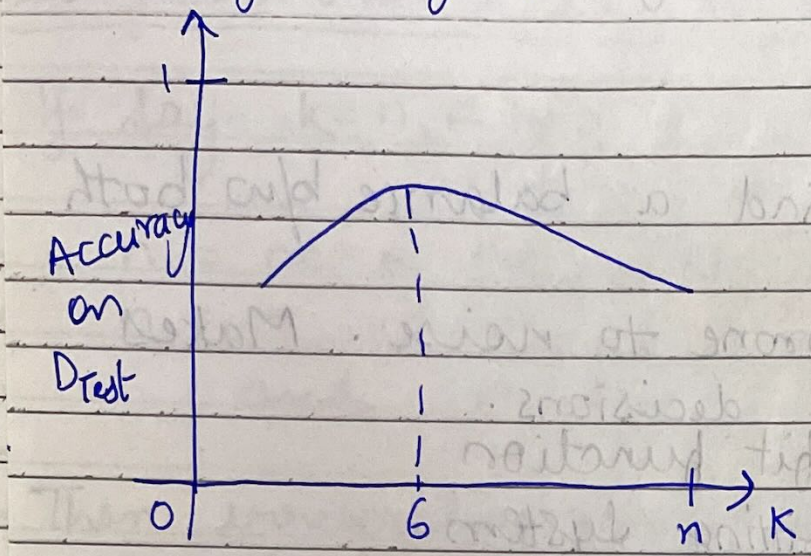
So the optimal value would be somewhere in the middle.

In order to find the right k, we must check the accuracy on D_{Test} and draw graph.

eg.

	Train data	Accuracy on D_{Test}
k = 1	D_{Train}	0.78
k = 2	"	0.82
⋮		

So, a typical graph will look as follows :-



So we conclude that $K=6$ gives the best accuracy on D_{Test} when using D_{Train} as training data.

Using D_{Train} and 6-NN on Amazon Fine Food Reviews accuracy = 96%.

But there is a small problem.
The soul objective of Machine learning is to perform well prediction on unseen data.
We used the whole D_n here to calculate the model function.

D_{Train} gives the Nearest Neighbours.
D_{Test} gives appropriate value of K.

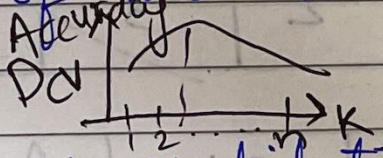
If a model works on future unseen data it has gener

Find D_{Train} = NN
D_{CV} = Right K value
D_{Test} = Accuracy after applying the function

Cross Validation

Divide D_n into 3 parts (D_{Train} to find nearest neighbour) (D_{CV} to be used to determine right K value) (D_{Test} to apply function to get accuracy on unseen data)

Now, if D_{Test} gives 93% accuracy on the optimal value of K found on $D_{Cross\ validation}$, we can say the accuracy of model is 93% on unseen data.



generalization accuracy
1% generalization error

V14: K-fold Cross Validation

Let's suppose we divide our D_n dataset into 3 parts D_{Train}, D_{CV} and D_{Test} into 60%, 20% and 20% division.
D_{Test} should be unseen data and accuracy must be calculated on that.
But the main problem is that we get only 60% of our data as train data.

So there must be a solution as more the training data, better is the algorithm. So, we should somehow try to get whole 80% ~~D_{Test}~~ D_{Train} + D_{CV} as

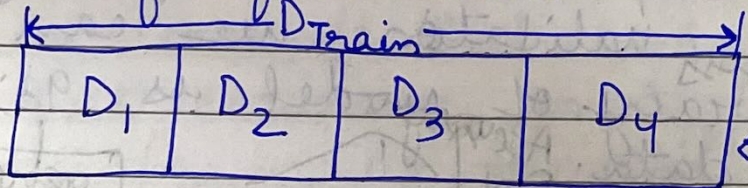
training data — different from K in KNN

Now we use K'-Fold Cross Validation to solve the above problem.

① D_n → D_{Train} (80%) → f_1 : (KNN)

D_n → D_{test} (20%) → unseen → accuracy.

② Say $K' = 4$, so we divide D_{Train} to 4 parts of equal size.



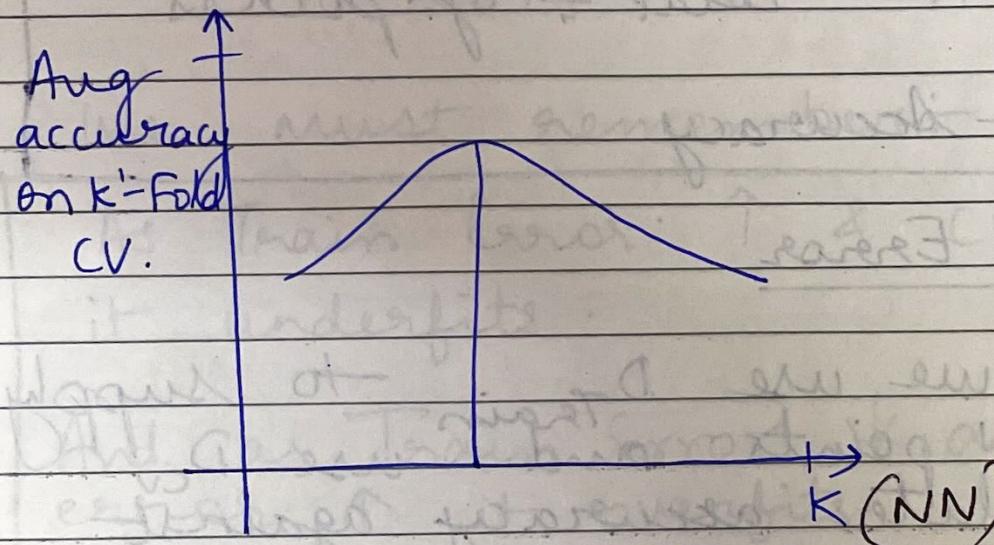
← random partition

K=NN not K'	Train	CV	Acc. on CV	
K=1	D_1, D_2, D_3	D_4	a_4	average $a_{K=1}$
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	average $a_{K=2}$
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	

gener

These average accuracy score for each k' must be plotted on Accuracy vs. K plot.

So, for arriving on conclusion of accuracy for a particular 'K', all the train is used as training data Also, more cross-validation is done as we get 4 D_{CV} sets.



But what should be value of k'

k' = 10 Rule of Thumb

10 fold Cross validation

Time taken to compute optimal 'K' using k' Fold cross validation increases k' times as compared to simple cross validation set.

V16: How to determine overfitting and Underfitting?

Given a value of 'K', how to determine whether it underfits or overfits? We use Graphical method for that.

$$\text{Accuracy} = \frac{\text{Total \# of points correctly classified}}{\text{Total \# of points}}$$

$$\text{Error} = 1 - \text{Accuracy}$$

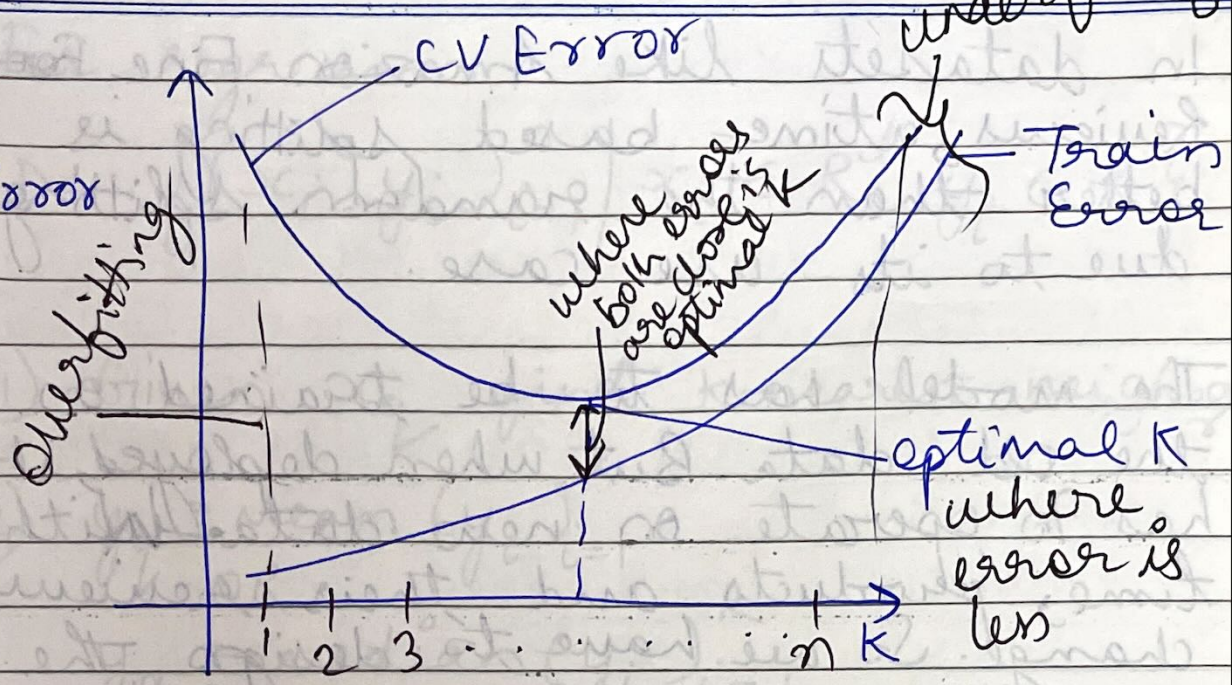
Training Error

Usually, we use D_{Train} to supply the NN points and use D_{CV} to calculate accuracy against different values of K.

The corresponding loss of this accuracy is called Cross Validation loss.

Instead of using D_{CV} points if we use D_{Train} points itself to calculate accuracy, the corresponding loss would be training loss.

There is no test error. we are calculating now.



We must remember :-

As Train Error ↑ & CV error ↑ it underfits.

And as Train Error ↓ and CV error ↑ it overfits.

V17 : Time based Splitting

It works only, if we have timestamp data.

1) Sort D_n in increasing order of time.

$D_n \xrightarrow{\text{Sort}} D_n$ → D_{Train} (top 60%.)
→ D_{CV} (next 20%.)
→ D_{Test} (last 20%.)

In datasets like Amazon Fine Food Reviews, a time based splitting is better than the random splitting due to its use case.

The model has to be trained on the old data. But when deployed, it has to operate on new data. With time, products and their reviews change. So we have to design the model in such a way that it takes more older data and gets good predictions on less older data.

This can be achieved by splitting in the above mentioned way. Merely doing random splitting won't give this ability to the model.

V18: K-NN for regression

For 2-class classification

$$D = \{(x_i, y_i)_{i=1}^{n} \mid x_i \in R^d, y_i \in \{0, 1\}\}$$

For regression

$$D = \left\{ (x_i, y_i)_{i=1}^{n} \mid x_i \in R^d, y_i \in R \right\}$$

1) Given x_q find k-nearest neighbours

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

OR

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

> The median formula is less prone to outliers.

V19 : Weighted KNN

The closer points among the KNN points are given more weight than further points.

~~How while voting~~, while these weights are added for each class, whereas ~~and~~ majority value wins. ~~###~~ in majority voting strategy.

> One of the ~~way~~ weightage ~~decided~~ can be $w_i = \frac{1}{d_i}$

where d_i = distance.

V20: Voronoi Diagram

It follows similar procedure while trying to draw decision boundary and take K=1 (one nearest neighbour) It colours each region with different colours. In each cell there is only 1 nearest neighbour.

V21: Binary Search Tree

Suppose we have numbers :-

V22: How to build a Kd Tree

It is a simple implementation.

KNN Time Complexity = $O(n)$
Space " = $O(n)$

We can't do much about space complexity since all points made to be present for calculation.

We can improve time complexity using data structure called Kd tree.
It makes time complexity = $O(\lg n)$

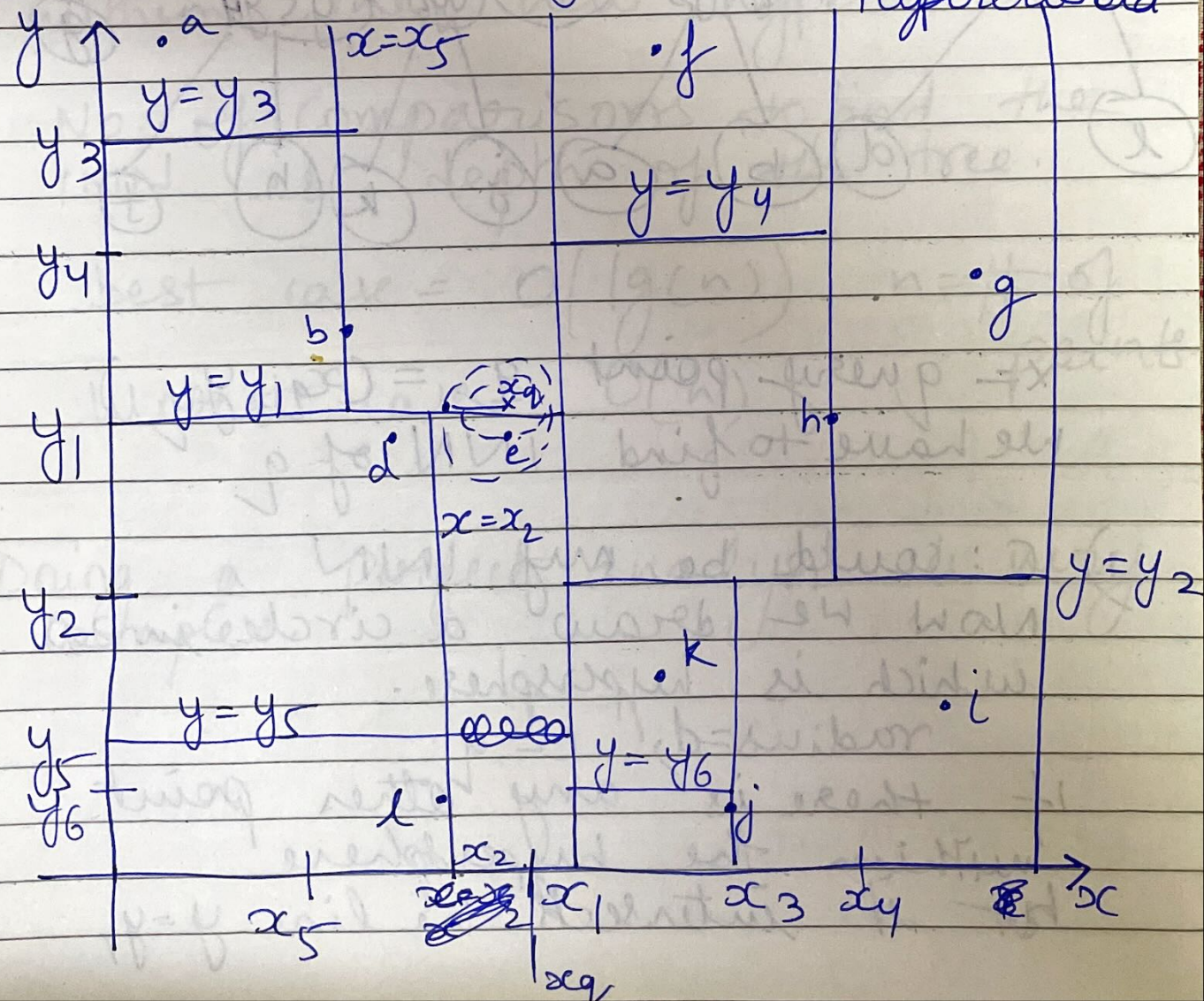
It is very popular and is used in Computer Graphics.

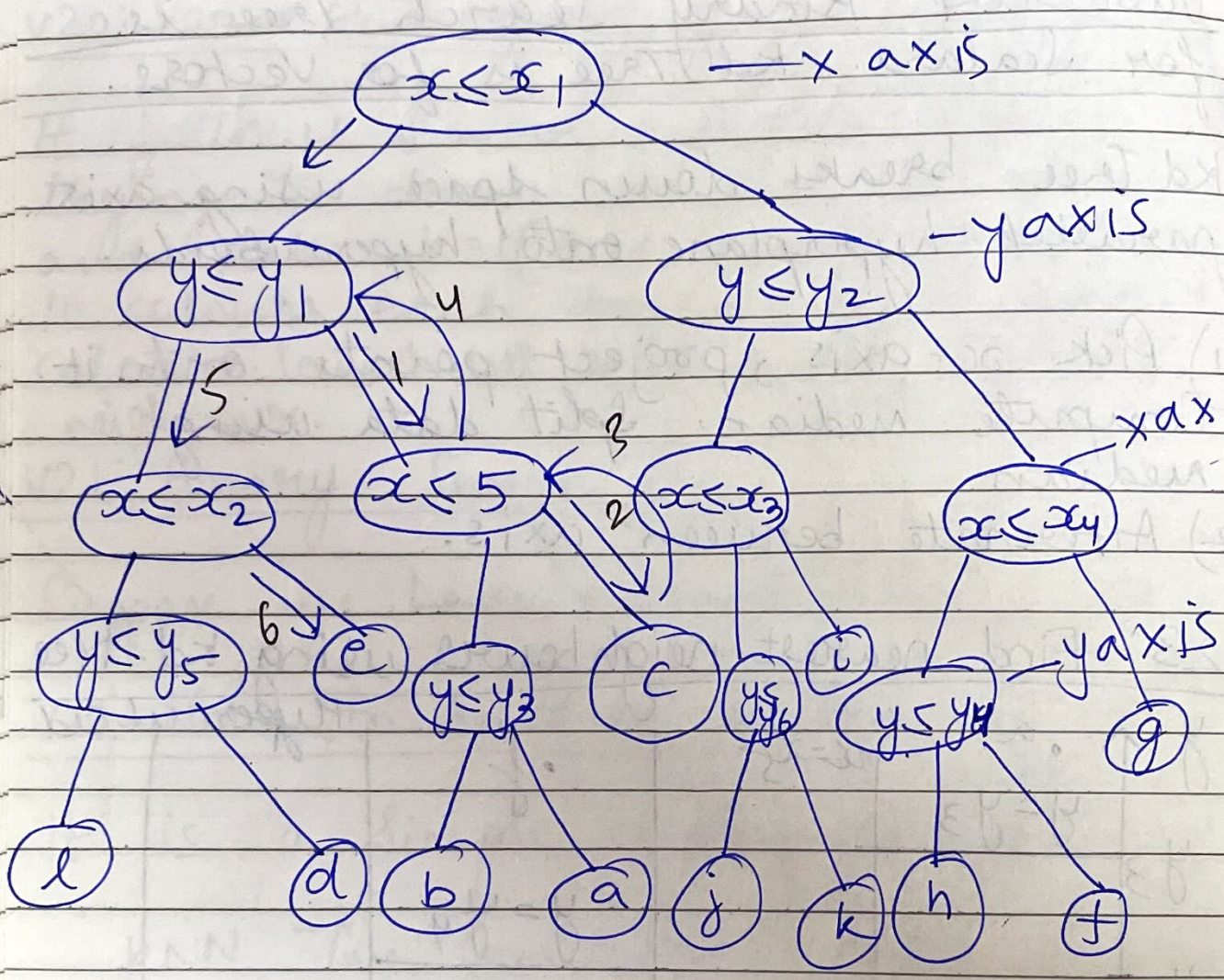
Just like Binary Search Tree is for Scalars. Kd Tree is for Vectors.

Kd Tree breaks down space using axis parallel hyperplane onto hypercuboids.

1) Pick x-axis, project points onto it. Compute median. Split data using median.
2) Alternate between axis.

V23: Find nearest neighbours using kd-tree





x = query point $q = (x_q, y_q)$
We have to find INN of q

1) C : could be my INN
Now we draw a circle in 2D which is hypersphere.
radius = d, c = q
If there is any other point within the hypersphere but it intersects the line $y = y_1$

So we track back so we reach another condition on $y = y$, and we check the other condition and go down there

2) So we say e could be INN is So we discard the previous hypersphere and draw another with less radius.

And there is no other point within the new hypersphere.

No. of comparisons to find the INN is depth of the tree.

best case = $O(\lg(n))$ n = # of points

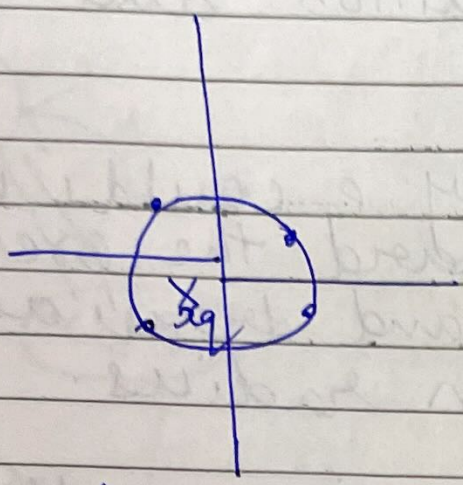
Worst case = $O(n)$

Doing a perfect analysis is very complex.

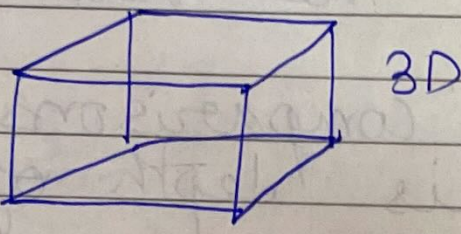
V24: Limitations of kd-tree

KNN

Q.



I have to check 4 regions.



I have to check 8 regions.

In nD, I have to check 2^d

Disadvantages:-

So, (1) when d is not small.

$d = 10$; $2^d = 1024$.

$d = 20$; $2^d = 10^6$ (Close to million)

As d ↑ in worst case, # of adj. cells 2^d ↑

when d is not small

So, n = 1 million; d = 20.

$2^d \approx n$.

So time complexity for NN = ~~$O(2^d \log n)$~~

$O(2^d * \log n)$

In Brute Force $O(2^d \log n)$ ← $O(n)$.

So, since we have many more dimensions kd tree becomes useless.

(2) $O(\log n)$ when d is small.

when data is uniformly distributed.

But non uniformly distributed in real world. So $O(\log n)$ no more valid.

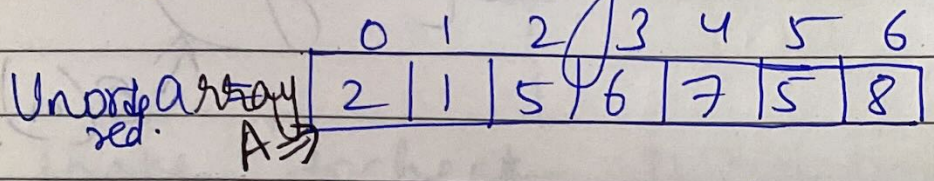
It becomes $O(n)$ in non-uniform distribution.

Kd tree was not invented for KNN but for computer graphic since they require 2D or 3D data.

V26: Hashing vs. LSH

LSH = locality Sensitive Hashing.

Normal hashing



We can store a in hashtable which is dictionary in python.

In normal hashing, there is hash function as hash(x) which returns memory location, basically returns a no. which is interpreted as address.

If empty then data can be stored there, else a linked list like structure formed & its pointer is held by this memory location, in addition to data which it already had.

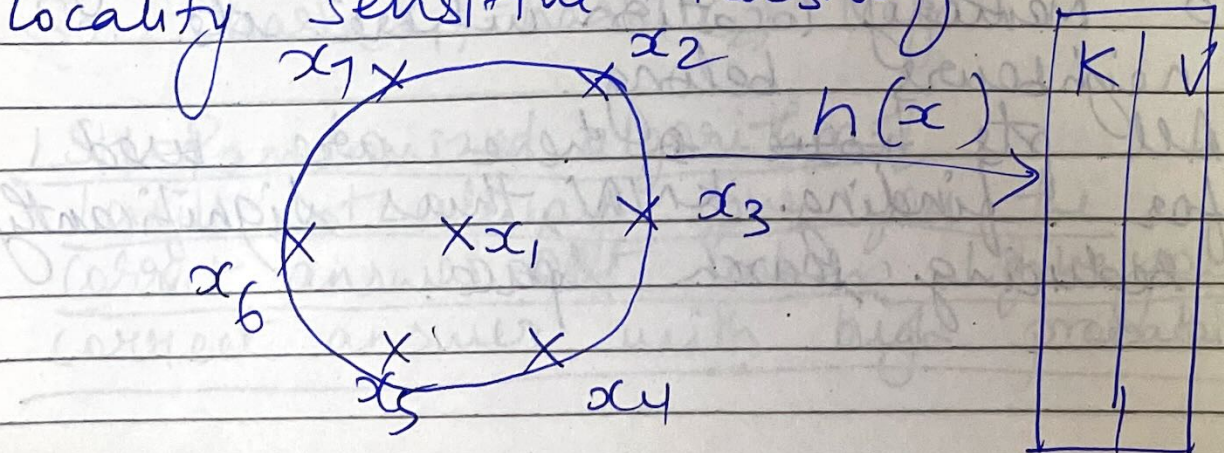
So hashing, we can go to memory location corresponding to an i/p.
Time complexity = $O(1)$.
Say an array is

0	1	2	3	4	5	6
2	1	5	6	7	5	8

We want to store the indexes of each no. using hash table

x	hash(x) loc.
2	0
1	1
5	2 — 5
6	3
7	4
8	6
8	

Locality Sensitive Hashing



goto each hashtable (total L tables) & take union of all the sections to which

It tries to achieve — all points that belong to the same neighbourhood hash to the same memory location.

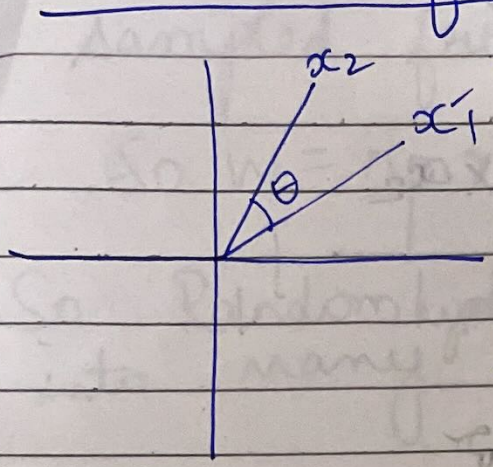
So, here the hash(x) takes a pt. x as i/p and returns memory location.

$$hash(x_1) = hash(x_2) = hash(x_3) = \ldots$$ for ideal LSH.

In the hash address, data pts are stored.

When x_q comes, hash x_q will go to memory location where its neighbours belong.
All pts ~~have~~ ~~0~~ there are used for finding kNN, thus significantly reducing search space.

V27: LSH for Cosine Similarity



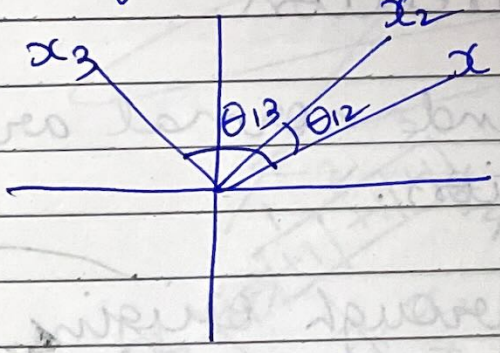
$\cos\text{-}\mathrm{sim}(x_1, x_2) = \cos\theta$

$\cos\text{-}\mathrm{sim}(x_1, x_3) = \cos(0) = 1$

As $\theta \uparrow$, x_1 & x_2 are angularly separated. $\uparrow$ distance $\uparrow$ cosine $\downarrow$ similarity $\downarrow$

Cosine Sim is all about angular similarity.

LSH for Cos-Sim is:-



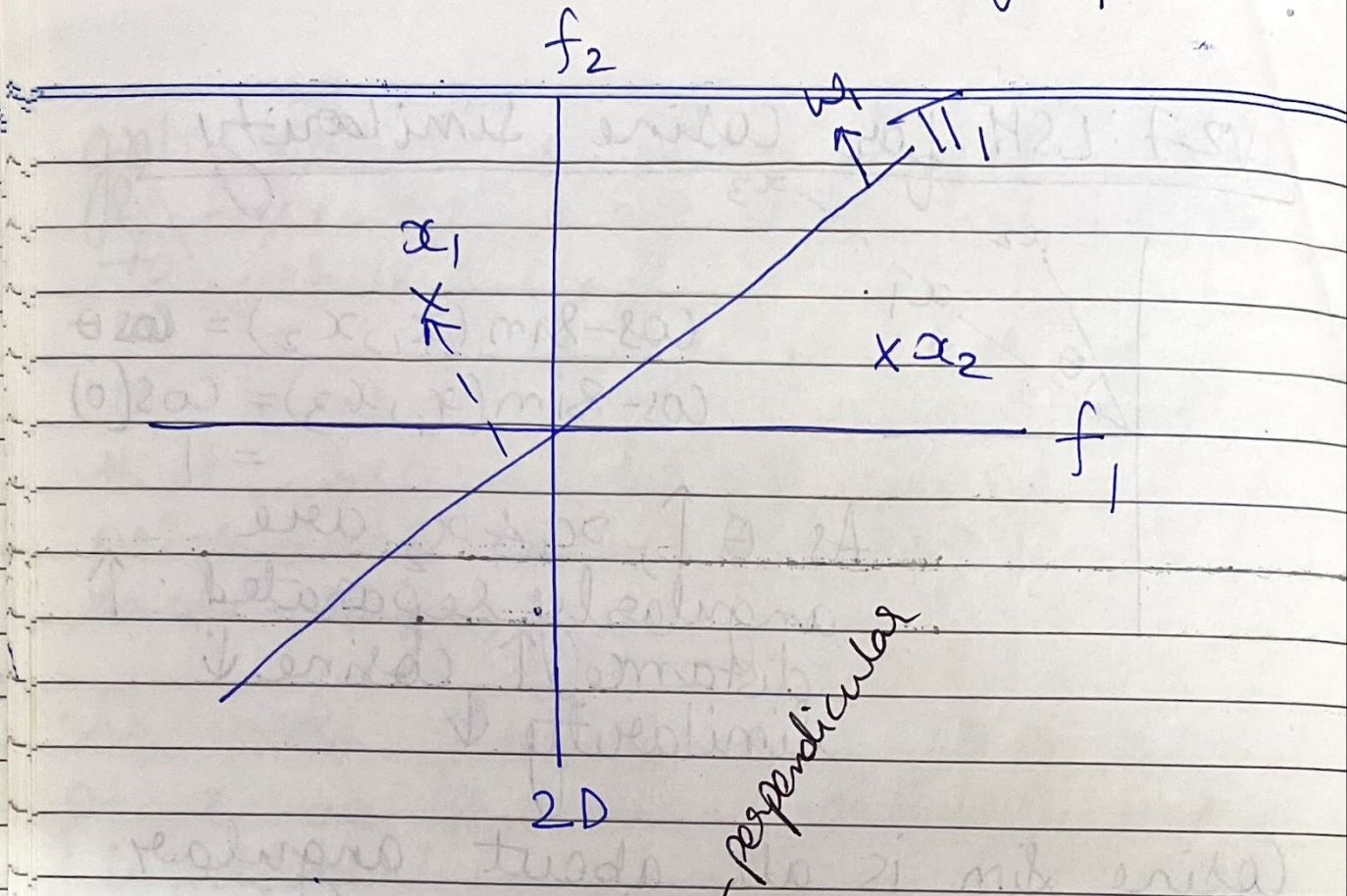
$\theta_{12} \approx 0 \Rightarrow x_1$ & x_2 are similar

Small θ pts which are close, goto same buket in hashtable.

So. $\mathrm{hash}(x_1) = \mathrm{hash}(x_2) = \mathrm{key}$.

LSH = Randomized algorithm:
I will not always give you the correct answer but when I give you correct answer with high probability

Random Hyperplane

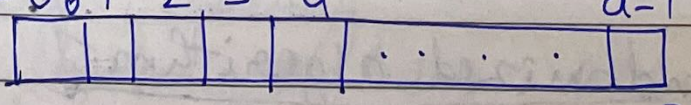


w_1 is unit normal vector to Π_1

$w_1^T x_1 \geq 0$ Since x_1 and normal w_1 are in same direction.

$w_1^T x_2 \leq 0$ Since x_2 and normal are in opposite direction.

Since plane passes through origin $w^T x = 0$

So, $w =$ 

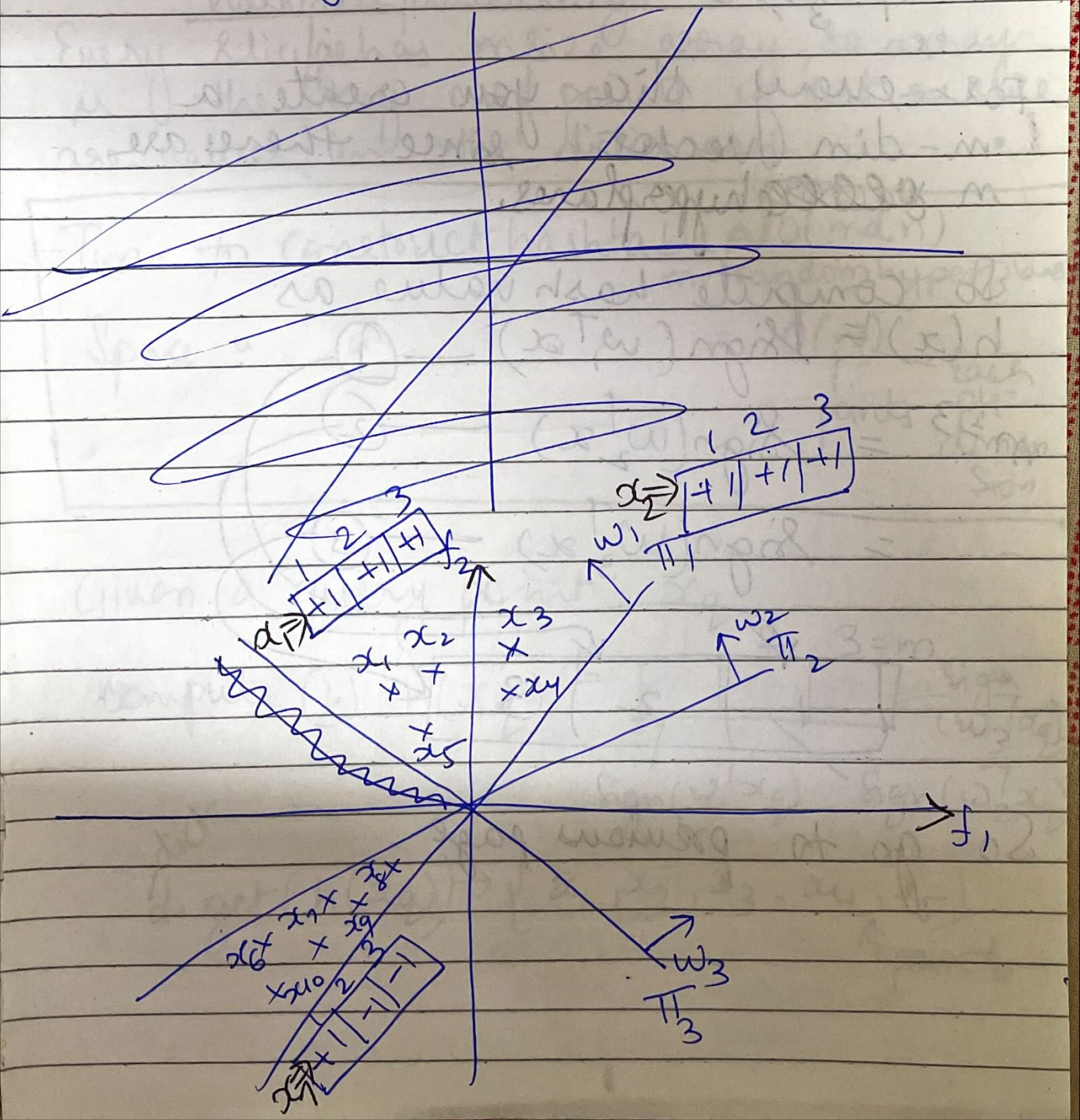
d-dim random vector.

Generate d dim vector s.t.,

Each value in w are random values (nos) sampled from $N(0,1)$

So $w = np.random.normal(0, 1, d)$

So Random hyperplanes can cut hyperspace into many regions.



Each point gets assigned to each slice/pc

So, $x_1^T w_1 \geq 0 \rightarrow +1$ if it is +ve we assign +1 ie same direction of normal.

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

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

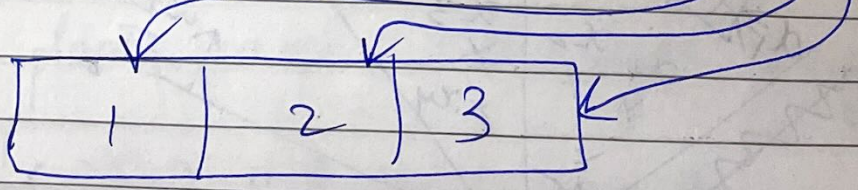
For every slice you create a m-dim vector since there are n hyperplanes.

So compute hash value as

$h(x) = \text{Sign}(w_1^T x)$ — (1)

$m=3 = \text{Sign}(w_2^T x)$ — (2)

$= \text{Sign}(w_3^T x)$ — (3)



So go to previous page

K	V
type 1: +1,-1,-1	points
type 2: +1,+1,-1	points

hashtable/dictionary

Every slice/pie has m size array, so array is the key and points in that slice are values in the dictionary.

Time to construct hashtable = $O(mdn)$ (m = random hyperplanes)

Space " " " = $O(nd)$ (n points, each pt's dimension)

Given a query point x_q,

compute (1.) $h(x_q)$ = [1 | 2 | 3=m] → sign $(w_3^T x_q)$

sign $(w_1^T x_q)$ sign $(w_2^T x_q)$

dict

d.get($h(x_q)$) → $\{x_1, x_2, x_3, x_4, x_5\}$ ← points

So given a hash table

Time Complexity for querying

$x_q = h(x_q) = O(md)$

If n' elements in slice then compare x_q So total time

$O(md + n'd)$

↓ if n' is small then

$O(md)$

So typically m is set to be $\lg(n)$

So, Time $O(d * \lg n)$ ← For querying time complexity.

Problem if point lies in adjacent slice, one could miss NN in cosine similarity as a measure of similarity

So, we do it this way :-

Create different hash function for each set of type.

1) $h_1(x) =$ m hyperplanes (different) ↕ different

2) $h_2(x) =$ m hyperplanes (different)

Different hyperplanes create different hashtable

Do repeatedly. Do it multiple times.

Your x_q will fall into same slice hence x_q and another point will fall in same slice.
After computing $h(x)$ of every set, take set union of all the points, you get a big set.

Compare x_q with all pts in big set.

Final Time Complexity to query
L hashtables $O(mdL)$

L - hashtables (typically small)
m = random hyperplanes
d - Each pts dimension.

$m = O(\lg n) \rightarrow$ No. of hyperplanes choice

As # of hyperplanes ↑
Slice ↑
pts per slice ↓

ty. Q
plane

So we need to be careful: if # of pts in a slice $< k$, it can be problem.

Similar or closer points should go to same bucket in hashtable

V28: LSH for Euclidean Distance

It is Similar to cosine-Similarity with Simple extension.

As in previous case, here also space is divided by different hyperplanes.

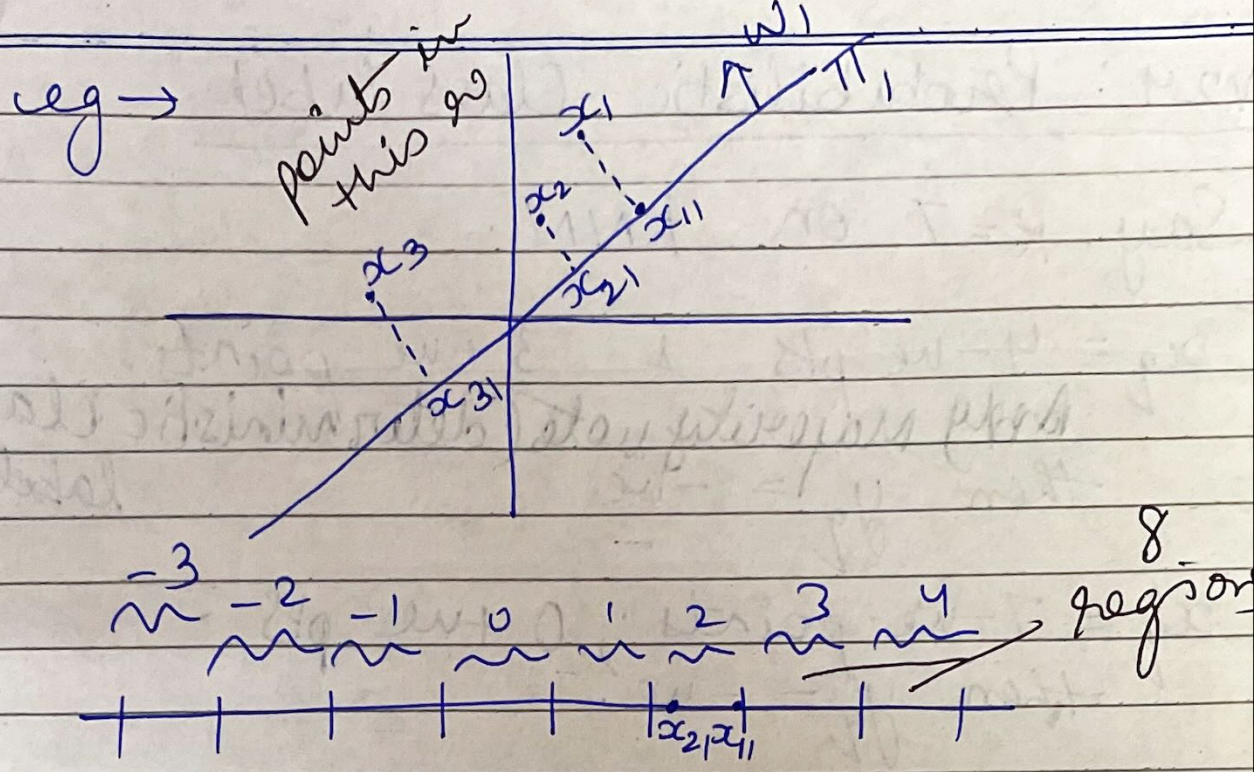
1) For each hyperplane, take projections of all data points on that hyperplane.

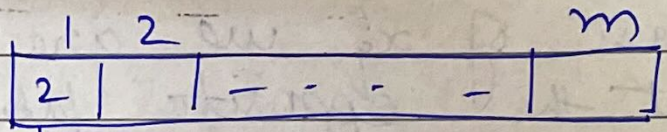
2) Each hyperplane is divided into ~~some~~ regions.

3) The projections belonging to same region would be closer.

4) The hash(x_q) becomes the m-dimensional vector, where each component of the vector say 5th Component represents the region of 5th hyperplane where the projection of that point belongs to.

LSH is not perfect/randomized algo.



$hash(x_{11}) =$ 

region in Π_1 projection belongs

So, in euclidean distance h(x) vector can have any numeric values unlike cosine similarity which only had +1 & -1

Just like cosine, here also we have L-different sets of m-hyperplanes.

Even if $x_1 x_2 \ll x_1 x_3$ still x_1 & x_2 are in same region & x_1 & x_3 in different for Π_1.

But since planes generated randomly Π_2 also may occur & x_1 & x_3 will be in

v29: Probabilistic Class label

Say $K = 7$ on KNN.

x_q = 4 -ve pts & 3 +ve points
Apply majority vote (deterministic class labelling)
then y_q = -ve

x'_q = 7 -ve points ; 0 +ve pts.
then y'_q = -ve.

In case of x'_q we are more certain about the opposite label as compared to x_q.

But the certainty doesn't reflect in the o/p.

Quantifying the certainty giving probabilistic class label.

1) $P(y_q = -ve) = 4/7$
2) $P(y'_q = -ve) = 7/7$

} more informative
makes model more interpretable.