K-nearest neighbours

Intro to classification

Given a new data-point, predict which class it belongs to.

$$y = f(x)$$

x = featurization of query data pt.
y = predicted class label.
$f()$ = classifier.

How does it work? algo.

data pt their class labels

$\{(x_i, y_i)\}$ → f

Training set

Algo learns the function f from the training set data.

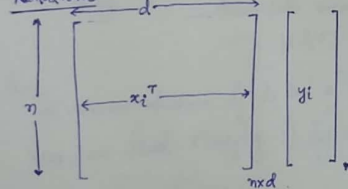
TRAINING STAGE

x_q → f → y_q predicted class label.

Test set

TESTING STATE.

Notations



* By default, every data pt is represented as col. vector. So, transpose taken here.

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

for binary classification.

D is a dataset of pair of values, n-such pairs, where the 1st of pair is a d-dimensional vector & 2nd is either 0 or 1.

Binary classification → $y_i \in \{0, 1\}$ Only 2 classes will there.

Multi-class → more than 2 classes. Ex → Iris dataset, MNIST, etc.

Ex → Amazon food reviews. (+ve & -ve class).

Regression → $y_i \in R$ (real number). Ex → predicting housing prices, etc.

K-nearest neighbour (Geometric intuition).

2D-toy dataset

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

Objective
→ We want to determine class label of query point.

Intuitive Approach → The points that are geometrically close to x_q or say in the neighbourhood of x_q are (+)ve points mostly. Hence, we can conclude that x_q is also (+)ve point. That's what KNN does.

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.

$\downarrow \; \downarrow \; \downarrow$

$y_1 \; y_2 \; y_3$

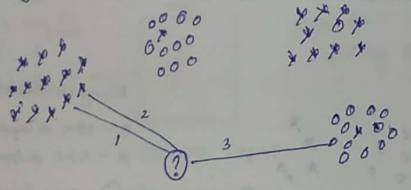
② decide class label of x_q using majority vote of y_i's.

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

that's it

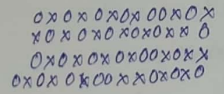
Failure cases of KNN

① query point is far away from clusters.



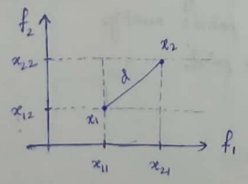
Output label may say ve/s -ve because of majority vote but it doesn't make much sense.
It would be better if the model could o/p - "don't know"

② (+)ve & (−)ve pts are randomly spread.



→ No useful information.

Distance Measures



d = len of shortest line fm x_1 to x_2.

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

Also called Euclidean distance.

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 $x_1 - x_2$

Manhattan distance :-

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

L1 norm of $x_1 - x_2$

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 $p=1$, → manhattan
$p=2$, → euclidean.

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

In general, L_q norm of a vector x is given by. $(x \in R^d)$

$$\|x\|_q = \left(\sum_{i=1}^{d} |x_i|^q\right)^{1/q}.$$

Hamming distance (b/w boolean vectors).

x_1, x_2 → boolean vectors → Binary BOW.

$x_1 = [0, 1, 1, 0, 1, 0, 0, \ldots]$
$x_2 = [1, 0, 1, 1, 0, 1, 0, \ldots]$

Hamming-dist(x_1, x_2) ⇒ # locations where binary vectors differ.

Also, hamming-distance could be used for measuring distances b/w strings.

Cosine-distance and cosine similarity

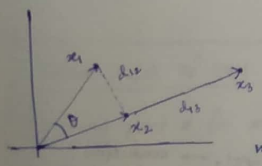
Intuitively, similarity ↑ ⇒ distance ↓
& similarity ↓ ⇒ distance ↑.

So, by definition, $\boxed{\text{cosine-distance}(x_1, x_2) = 1 - \text{cosine-similarity}(x_1, x_2)}$

Just as in Euclidean terms, points which have less distance b/w 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 b/w x_1 & x_2.



$d_{12} < d_{13}$.

but cos-dist$_{13}$ < cos-dist$_{12}$

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

If the problem deals with angular stuff, then taking euclidean distance doesn't make much sense and vice-versa.

* Also, $\cos\theta = \frac{x_1 \cdot x_2}{\|x_1\|_2 \, \|x_2\|_2}$ if x_1 & x_2 are unit vectors, $\cos\theta = x_1 \cdot x_2$

Relationship between Euclidean & cosine distance.
(See cosine similarity with page for details.)

$[\text{euc-dist}(x_1, x_2)]^2 = 2 \times \text{cos-dist}(x_1, x_2)$. if x_1 & x_2 are unit vectors.

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

Measuring effectiveness of KNN

① Split D — D_{train} (n_1) ≈ 70%. $D_{train} \cap D_{test} = \phi$
D_{test} (n_2) ≈ 30%. $D_{train} \cup D_{test} = D$

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

Algo:-

```
count ← 0
for each pt in D_test:
    x_q ← pt
    use D_train & KNN to determine y_q
    if y_q == y_qt:
        count ← count + 1
```

In the end, count = # pts for which D_{train} + KNN gave correct class label

$$\text{Accuracy} = \frac{\text{count}}{n_2} \qquad 0 \le \text{accuracy} \le 1$$

Accuracy is one of the measures. There are many others that'll come.

Time and Space complexity of KNN

Evaluation:- given x_q. Need to determine y_q.

① KNNpts = [].
for each x_i in D_{train}: → O(n).
- compute d(x_i, x_q) → d_i → O(d)
- Keep the smallest k-distances (x_i, y_i, d_i) in KNNpts. → O(k).

$k << n$, hence, neglected.
So, O(n*d) for above logic.

② cnt_pos = 0 ; cnt_neg = 0.
for each x_i in KNNpts: — O(k).
if y_i is +ve: — O(1).
cnt_pos += 1
else
cnt_neg += 1
return (cnt_pos ≥ cnt_neg)

So, total O(k). So, overall time complexity = O(nd) + O(k). = O(nd)
If d << n, O(n)
Time taken to compute label of a single query pt.

Space complexity - We need to calculate distance fm each pt in D_{train}

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

Space = $O(n \times d)$ → Not good. Very high.

Limitation of KNN (simple implementation).

Consider amazon food reviews example.

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

With simple KNN,

Time = $O(nd)$

Space = $O(nd)$.

$n \approx 364K$, $d \approx 100K$ → D_{train}.

$nd \approx 36$ GiB RAM, means min. 64 GiB RAM needs to be installed.

(a) Time complexity - 36 Billion computations.

Review → (1 ms) → (+)ve / (-)ve label.

Internet services

This should be the case. If latency ↑, bad user experience.

low latency

for Trading & finance, as much as 1 μs delay is permitted only.

Non-low latency systems → in medicines recommendation, etc.

This is one of the main reason of not implementing KNN in production much instead of it being so simple & intuitive.

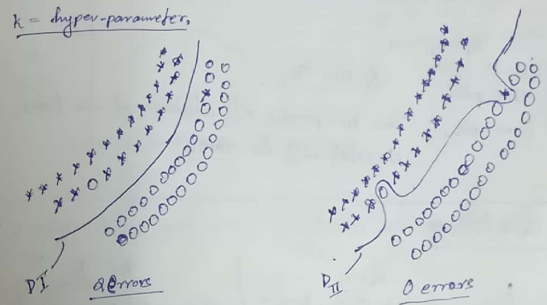
Some modifications to KNN later to be covered -

① Kd-tree

② LSH (Locality sensitive hashing).

Decision surface of KNN as K-changes.

K = hyper-parameter.



D_I & D_{II} are the decision surfaces

└ because, of generalization of higher dimensions.

this surface decides whether a point is positive or negative depending upon which side of the surface the pt. 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 the K-nearest neighbours.
- → find the majority vote & 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 consider a point in close proximity of 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, but due to majority voting of remaining 4 nearest neighbours, that pt will be labelled the opposite of outlier, giving smooth surface as D_I.

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

But, say $k = n$. $\quad n_1 + n_2 = n$.

n_1 = no of (+)ve pts $\quad$ & $n_1 > n_2$.

n_2 = no of (-)ve pts. $\quad$ Then irrespective of whatever pt we take, its label will be (+)ve.

Overfitting & Underfitting

$k = n$ ✗

→ Lazy model.

Checks which class has more data pts & assigns that label to new query pt.

↔ Underfit model.

$k = 5$ ✓

→ Tries to find a balance b/w both extremes

→ Less prone to noise. Makes more sensible decision.

$k = 1$ ✗

→ This model overworks to get more complicated decision surface & so as to ensure no errors.

→ It may take noise as data & classify new pts incorrectly.

Determining optimal value of k. (Need for cross validation).

$k = 1, 2, 3, \ldots n$

↓ overfit $\qquad$ ↓ underfit.

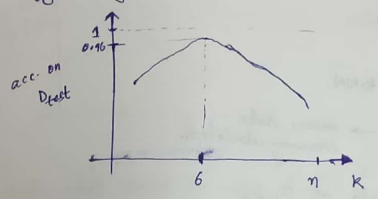
So, the optimal value will be somewhere middle.

Initial thoughts

Trying out diff value of k & training on D_{train} and checking the accuracy on D_{test}.

	train data	acc on D_{test}
$k=1$	D_{train}	0.78
$k=2$	"	0.82
$k=3$	"	0.85.
⋮	⋮	⋮

A typical graph will look as follows.



So, we can conclude $k=6$ gives the best accuracy on D_{test} when using D_{train} as training data.

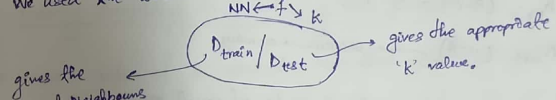
✗ using D_{train} and 6-NN on Amazon reviews dataset, accuracy = 96%.

✓

Small problem.

The sole objective of ML is to perform well prediction on unseen data. (generalization).

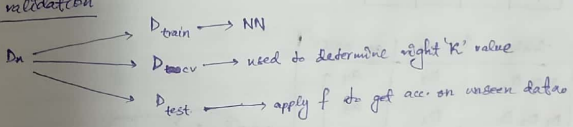
We used the whole D_n here to calculate the model & found



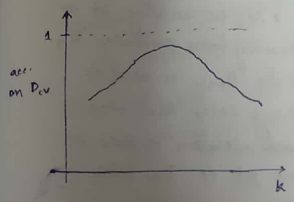
So, we can't say here the accuracy of model is 96% on unseen data.

(typical dissatisfaction (20%))

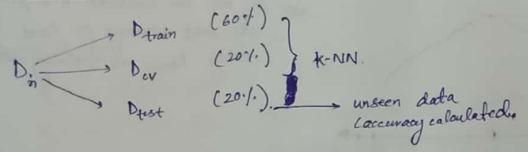
Cross validation



Now, if D_{test} gives 93% accuracy on the optimal value of 'k' found on D_{cv} cross-validation, we can say the accuracy of model as 93% on unseen data.



K'-fold cross validation



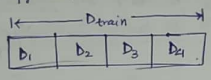
Problem :- we get only 60% of data as D_{train}.

* More the training data, better is the algorithm. So, we should somehow try to get whole 80% ($D_{test} + D_{cv}$) as training data.

K'-fold cross validation does this very smartly.

① D_n → D_{train} (80%) → f: (K-NN).
D_{test} (20%) → unseen → accuracy

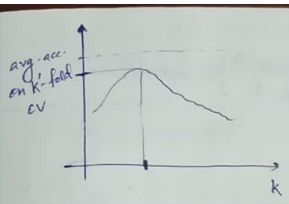
② say K' = 4. So, we divide D_{train} to 4 parts of equal size.



	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	
				avg → $a_{K=1}$
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 → $a_{K=2}$

The points to be plotted on Accuracy Vs K plot

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



So, as we wanted, all the 80% of data used as training data.

But, what should be value of K' ?

↳ K' = 10 — Rule of Thumb.

* Time taken to compute optimal 'K' using K'-fold cross validation increases K' times as compared to simple cross-validation set.

Knowing overfitting or Underfitting from given 'K'

* Given a value of 'K', how to determine whether it underfits or overfits?

→ graphical method.

Some terminologies—

$$\text{Accuracy} = \frac{\#\text{pts correctly classified}}{\text{Total } \#\text{pts.}}$$

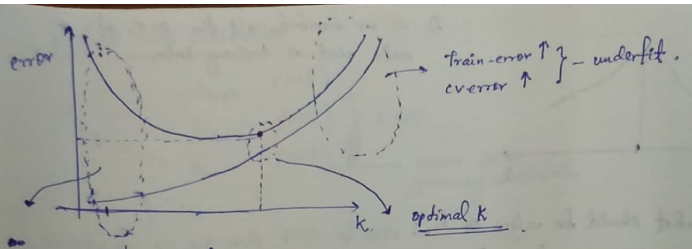
$$\text{error} = 1 - \text{Accuracy}$$

Training error :-

Usually we use D_{train} to supply the NN pts. 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} pts, if we use D_{train} pts itself to calculate accuracy, the corresponding loss would be training loss.



Time-based splitting

↳ works only if we have timestamp data.

① sort D_n in ascending order of time.

D_n → sort → D_n → D_{train} (top 60%)
→ D_{cv} (next 20%).
→ D_{test} (last 20%).

In datasets like Amazon food reviews, a time-based splitting is better than random splitting due to ~~this~~ its use case.

* The model has to be trained on the old data we have. (no other option).
But when deployed, it has ~~operate~~ to operate on new data.
~~Sorting~~ * With time, products and their reviews change.
So, we have to design the model in such a way that it takes ~~older~~ (or train)
more older data & gets good predictions on less older (more recent) data.
This can be achieved via by splitting in the above mentioned way.
Random splitting won't give this ability to the model.

So, whenever time data available and behaviour/data changes over time, time based splitting preferable.

K-NN for regression

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

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

① given x_q, find k-nearest neighbours $(x_1,y_1), (x_2,y_2) \ldots$

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

weighted K-NN

The closer pts among the KNN pts are given more weight than farther pts.
Now, while voting, these weights are added for each class & majority value wins.

One of the w_i can be $\boxed{w_i = \frac{1}{d_i}}$ ← there can be many more as well.

Voronoi diagram

↳ follow similar procedure procedure as while trying to draw decision boundary. & take $K=1$ (one nearest neighbour).
Colour each region with different colour.

Efficient implementation of K-NN

Simple implementation — K-NN — $O(n)$ time, $O(n)$ space } K & d are small.

We can't do much about space complexity since all pts needs to be present for calculation.
We can improve time complexity using data-structure called kd-tree

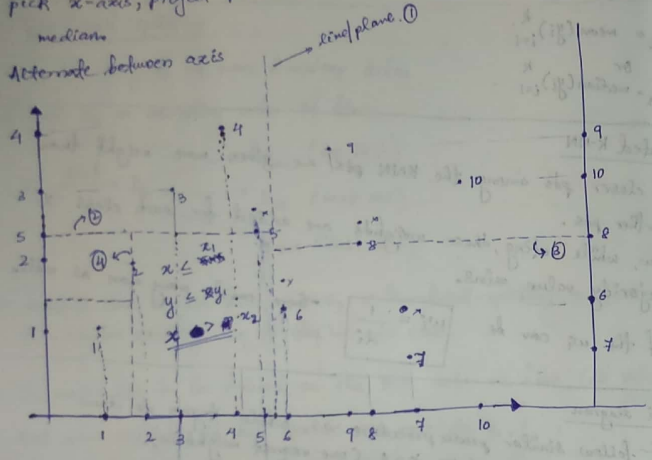
kd-tree — makes time complexity O(log n).
└ very popularly used in computer graphics.

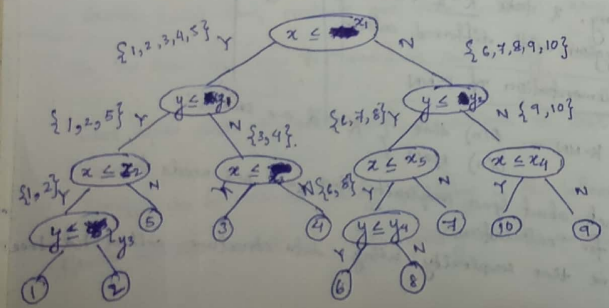
Just like BST used for numbers (scalars), when that concept extended to vectors, its called kd-tree.

kd-tree — breaks down space using axis parallel hyperplane into hypercuboids.

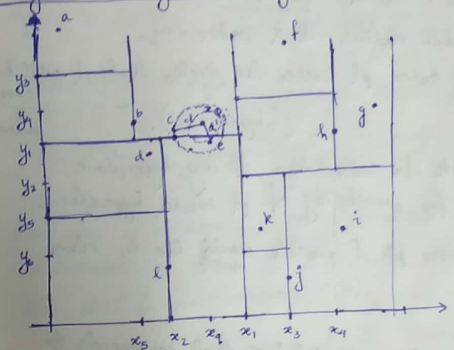
① pick x-axis, project pts onto it, compute median, split data using median.

② Alternate between axis

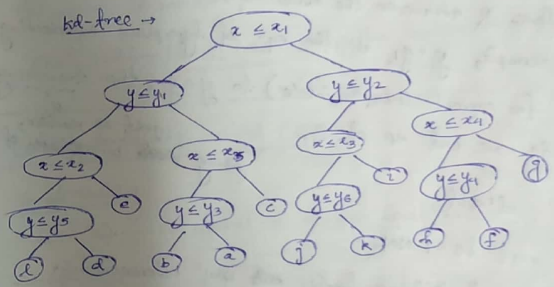




Finding nearest neighbours using kd-tree.



kd-tree →



x_q: query point.

We'll find 1-NN of x_q in the following, which could be extended to get K-NN.

① Parse the tree for x_q (query pt). This will give the hyper cuboid to which x_q belongs to.
(∵ each hypercuboid has a single pt in it).
Since x_q & c belong to same hypercuboid, 'c' potentially could be the 1-NN.

② Join line joining x_q & c. Let distance = d.

③ Draw a hypersphere with radius = d & centre = x_q.
If there will be any other pt inside the circle, that pt will be 1-NN probably.
(hypersphere).

④ We algorithmically check for intersection of any hyperplane with the hypersphere for possibility of pt inside hypersphere.
That is because, any closer pt if possible must lie in other hypercuboids.

⑤ Track back to the condition in kd-tree that corresponds to that hyper-plane & traverse the other side.
Here, in this example, $y = y_1$ is the eqn of hyperplane intersecting.
Tracking back frm current locatn ('one' 'c') to $y \leq y_1$, then traversing the other side, we end up at 'e'.
just as earlier, using co-ordinates of x_q

⑥ $x_q - e = d'$. $d' \leq d$
Hence 'c' can't be 1-NN.
goto ② & repeat till no further 'new' intersections.

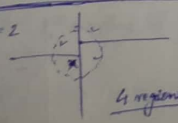
Best case :- $O(\log n)$.
worst case :- $O(n)$.

Extending 1-NN to kNN,
Best-case - $O(k \times \log n)$
worst case - $O(k \times n)$.

All these if d is 'small'

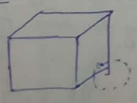
Limitations of kd-tree

d = 2



4 regions

d = 3



8 regions.

So, in worst-case, for d-dim data, hypersphere intersection may result in looking upto '2^d' regions. ⟶ exponential.

If d is large, Time complexity = $O(2^d \log n)$.
If $2^d > n$, Time complexity = $O(n \log n)$ ⟶ worse than brute-force.

② Even when d is small, $O(\log n)$ holds when data uniformly distributed in space.
Else, it'll be more towards $O(n)$.

* The whole kd-tree was designed for computer graphics (2-D & 3-D) for which it works pretty well.

Read wiki — / k-d_tree — python implementation, variations / ball tree, etc.

Locality Sensitivity Hashing (LSH)

Normal Hashing

hash(x) ⟶ returns a memory locatn {basically returns a no. which is interpreted as address}
i/p no.

If empty, then data can be stored there,
else a linked-list like structure formed & its ptr is held by this memory locn (in addition to data which it already has).

* So, by hashing, we can go to a memory locatn corresponding to an i/p in $O(1)$.

Say, an array →

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

we want to store the indexes of each number using hash table.

x	hash(x) location	
2	0	
1	1	
5	2	→ 5
6	3	
7	4	
8	6	

Locality Sensitive Hashing

↳ It tries to achieves — all pts that belong to the same neighbourhood hash to the same memory locations.

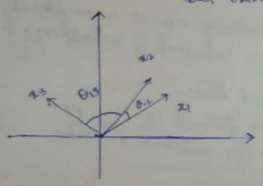
So, here the hash(x) takes a pt 'x' as i/p & returns memory locations.

$hash(x_1) = hash(x_2) = hash(x_3)$ ------ for ideal LSH

* In the hash address, data pts are stored.
* When x_q comes, $hash(x_q)$ will go to memory locat^n where its neighbours belong.
* All pts there are used for finding K-NN, thus significantly reducing search space.

LSH for cosine similarity (Implementation).

↳ Randomized algo → doesn't guarantee to always give the right ans, but there is a high probability of getting correct ans.



$\theta_{12} \sim 0$ (small) $\Rightarrow \cos\theta \approx 1 \Rightarrow$ highly similar

$\text{cosine-similarity} = \cos\theta$ (by def^n).

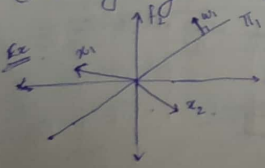
So,

key	value
α	x_1, x_2

$h(x_1) = h(x_2) = \alpha$

This is what we need to implement → same entry for x_1 & x_2.

① We angularly break-up the whole space using random hyperplanes.



π_1 → random hyperplane. w_1 — normal to π_1.

Now, $w_1^T x_1$ (or $\vec{w_1} \cdot \vec{x_1}$) > 0.

$w_1^T x_2 < 0$.

So, the sign of $w_1^T x$ gives an idea about which side of the plane x lies.

↓

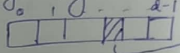
core idea of the implementation.

* Generating random hyperplane

$w^T x = 0$

w = d-dim vector —

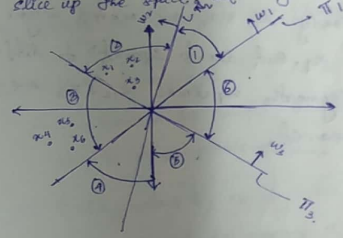
So, generating random d-dim vector ⇒ generating a random hyperplane.



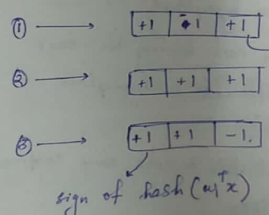
each of the values sampled from N(0, 1).

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

-So, random hyperplanes will slice up the space into regions.



- In the given fig., 3 planes divide the space into 6 regions each region can be uniquely identified via the (sign of (hash of (pts in that region)))



x_1 belongs to ①.

sign of $hash(w_3^T x)$

& so on.

sign of $hash(w_1^T x)$ $x \in$ ③

So, all the pts belonging to same region will give same hash value, just as we wanted.

So, m-hyperplanes → 'key' in hash-table is m-dim vector.

key	value
[1, 1, 1]	$[x_1, x_2, x_3]$
[1, 1, -1]	$[x_4, x_5, x_6]$

Space:- $O(n \times d)$. — dimension (can be large)

Time:- $O(n \times m \times d)$ — training.

$O(m \times d + n' d)$ — Runtime — n' — no of pts in each section.

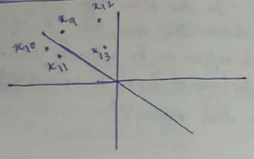
↳ small logically.

So, $O(md)$.

& $m = \log(n)$. — usually taken.

Hence, runtime → $O(d \log n)$.

Problem with LSH



Clearly x_{10} & x_{11} are nearest neighs of x_q. But due to separatⁿ by diff segments, x_q thinks x_{12} & x_{13} are nearest.

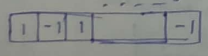
How to overcome?

↳ Instead of having a single set of m-hyperplanes, we'll have L-sets of m-hyperplane each, each having its own hash table. ↳ {each set having different m hyperplane than other sets}.

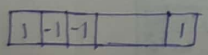
If ~~other~~ two points are close, then they'll fall in the same section in atleast 1 of the hyperplane divisions.

So, for each point x_0,

① calculate which section it belongs to in each of the L sets.

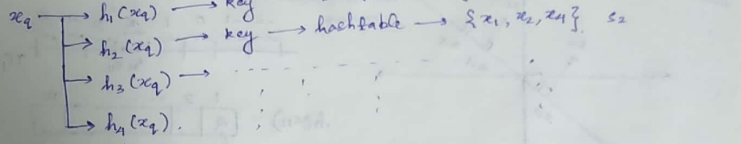
$hash_1(x_0)$ = 

→ m-dimensional

② $hash_2(x_0)$ = 

goto each hash table (total L tables), & take union of all the sections to which x_0 belongs to.

Now find KNN from this set of pts.



KNN searched from $S_1 \cup S_2 \cup S_3 \cdots S_p$

time complexity → $O(mdL)$ → {query pt}.

* As no of hyperplanes ↑, no. of slices ↑, #pts per slice ↓.
So, we need to be careful, because if #pts in a slice < K, then it can be problematic.

* But if #hyperplane ↓, no of slices ↓, #pts per slice ↑.
Hence, we may need to compare a lot of pts.

Thus, we need to find proper balance. Bcz both extremes. usually $[m = \log(n)]$

LSH for euclidean-distance (implementation).

↳ Similar to cosine-similarity with simple extension.

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

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

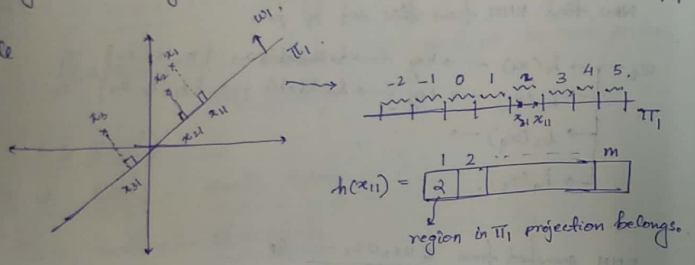
② each hyperplane is divided into say, 'a' regions.

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

④ the $hash(x_q)$ becomes thus 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.

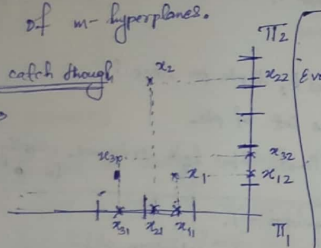
Example



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

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

One catch though



Even if $x_1x_3 << x_1x_2$ 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 same region there.

* LSH used extensively used in computer vision applications.

Probabilistic Class label

say K=7 in KNN.

x_q: 4 -ve pts & 3 +ve pts → y_q = (4/7)ve. } less informative.

x_q': 7 -ve pts → y_q' = 1(-)ve }

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

But the certainity doesn't reflect in the O/P.

Quantifying the certainity → giving probabilistic class label.

$P(y_q = -ve) = 4/7$. } more informative.
$P(y_{q'} = -ve) = 7/7$ } Makes models more interpretable.