29.1 How Classification works?

Let us consider the Amazon Fine Food reviews dataset. For each review, we have a text associated with it, and this text is converted into a vector form through one of the vectorization techniques like BOW, TF-IDF, Avg Word2Vec, TF-IDF Weighted Average Word2Vec, etc. Now for each review, we have a vector and also the data that tells us whether the review is positive or negative.

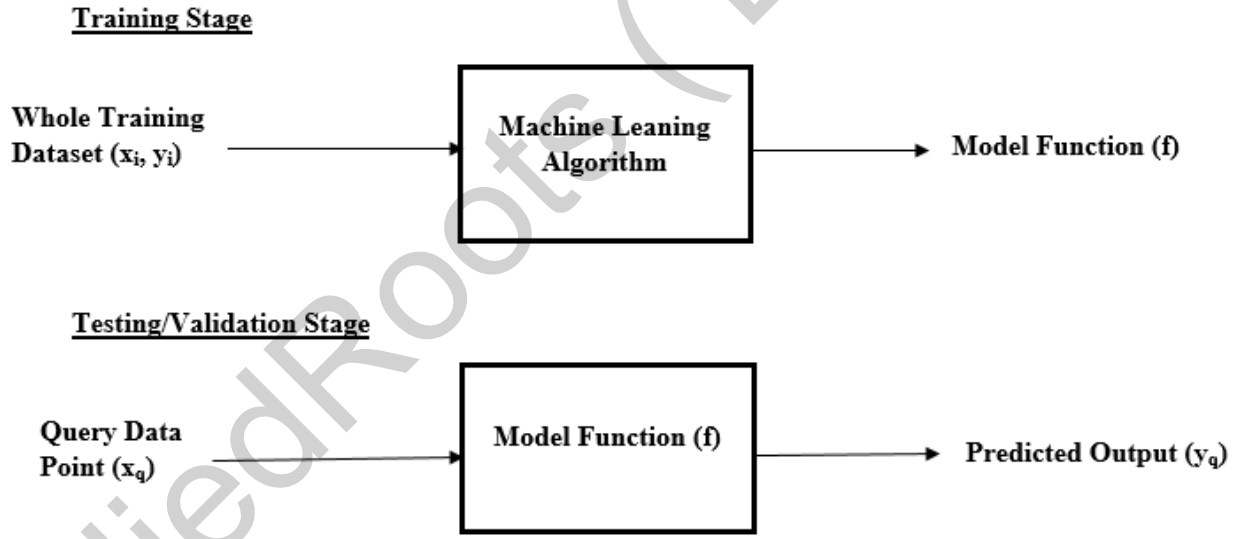
The classification task for this problem is, for a given review, we have to predict if it is positive or negative.

Let us assume each review in our dataset is denoted as 'x_i'. Classification here is given a review 'x_i', we have to find a function 'f' such that if we apply that function on 'x', we get the output 'y' either as 'Positive' or 'Negative'. This is the central concept of classification.

$y_i = f(x_i)$

where $x_i \rightarrow$ query review (in vector form)

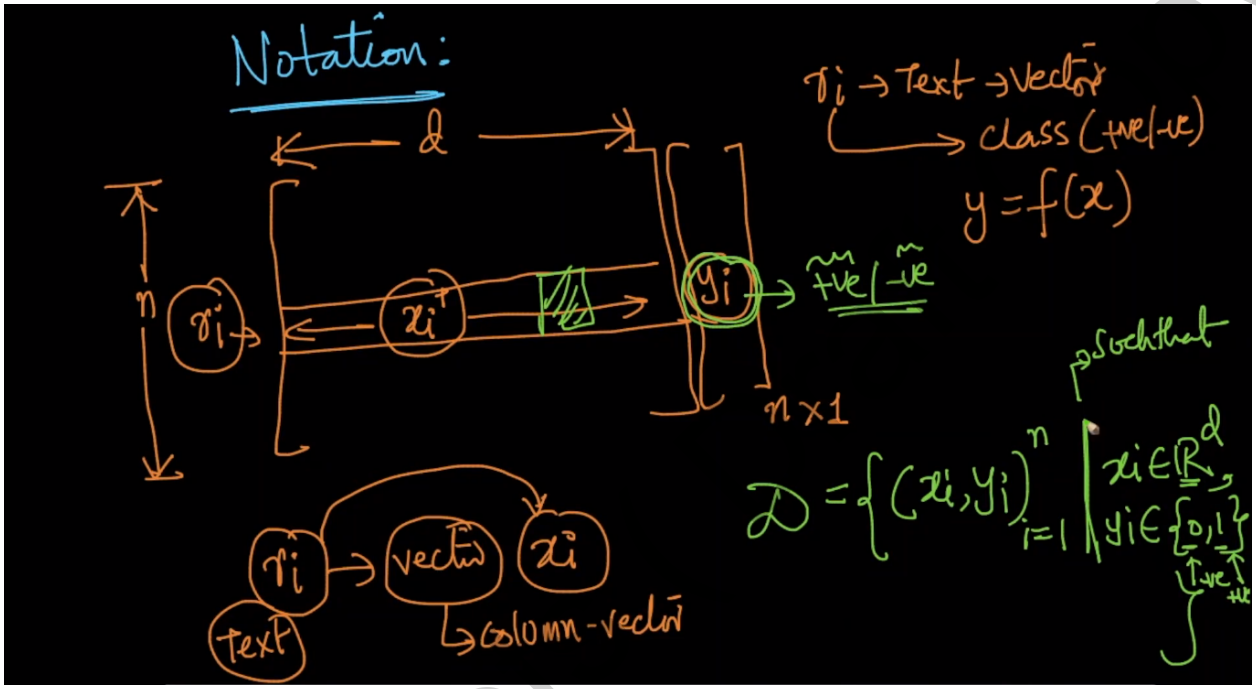
$y_i \rightarrow$ predicted class label



The algorithm gets trained using the training data and computes the function 'f'. After computing the function 'f', in the validation stage, this function is used to predict the output for the test data.

29.2 Data Matrix Notation

We have the reviews in Amazon Fine Food Reviews Dataset and let each review 'r_i' be represented in a vector form 'x_i' and the corresponding output class may be denoted by 'y_i'. Then each data point is represented in data matrix format as shown below.



Here the rows represent the data points and the columns represent the dimensions/features. Let us assume we have 'n' data points, and each data point has components in 'd' dimensions. Then the dataset is represented mathematically in the form of a set as $\mathbf{D = \{(x_i, y_i)_{i=1}^{n} | x_i \in R^d, y_i \in \{0,1\}\}}$

Here we are converting the classes from 'Negative' and 'Positive' format to 0-1 format, because linear algebra couldn't understand the terms 'Positive' and 'Negative'.

Note: As there are 'd' dimensions in every data point 'x_i', we have used the notation $\mathbf{x_i \in R^d}$. As 'y_i' takes only two values (ie., 0 and 1), we have used the notation $\mathbf{y_i \in \{0,1\}}$.As there are only 2 classes in the dataset, we call this problem a **binary classification** problem.

Note: If we consider the MNIST dataset, there are 60000 data points, each data point is a 784-dimensional vector and the class labels are the numbers 0 to 9, then the MNIST dataset is represented mathematically in the form of a set as

$\mathbf{D = \{(x_i, y_i)_{i=1}^{60000} | x_i \in R^{784}, y_i \in \{0,1,2,3,4,5,6,7,8,9\}\}}$

As there are 10 classes in the MNIST dataset, we call this problem a **10-class classification** (or) a **multi-class classification** problem.

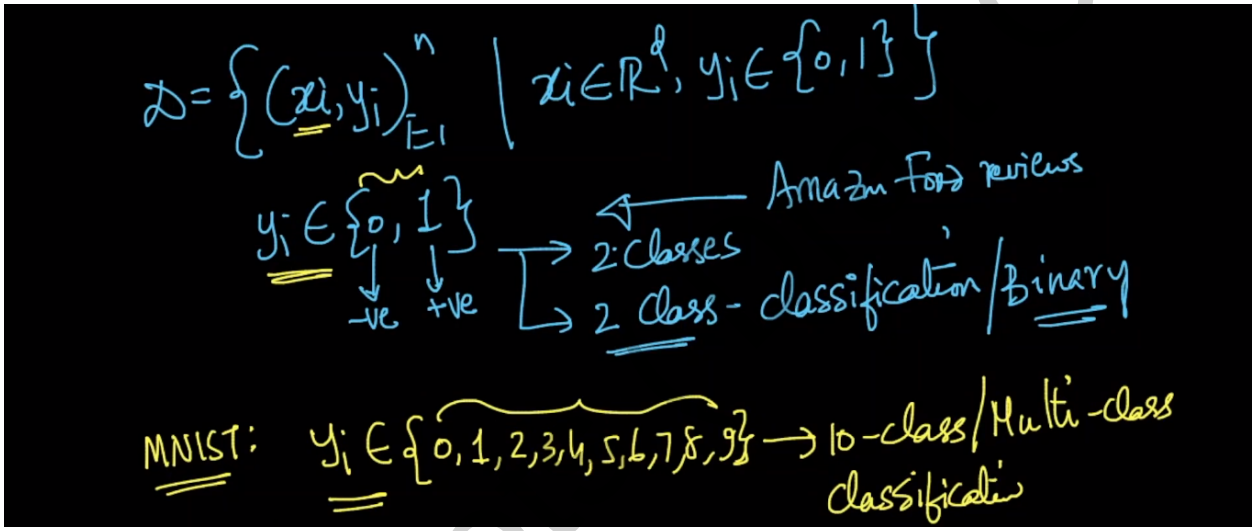
29.3 Classification vs Regression

Classification - Definition

Classification is the problem of identifying to which of a set of categories a new observation belongs to, on the basis of a training set of data containing observations whose category is already known.

Classification deals with predicting a qualitative (or) categorical response.

Below are the examples of Classification problems that were discussed starting from the timestamp 0.03.



In the dataset of Amazon Fine Food Reviews, the class labels are 0 and 1. If any query point is given, its label would be either 0 or 1. As it has only 2 classes, such a classification problem is called a **binary classification** problem.

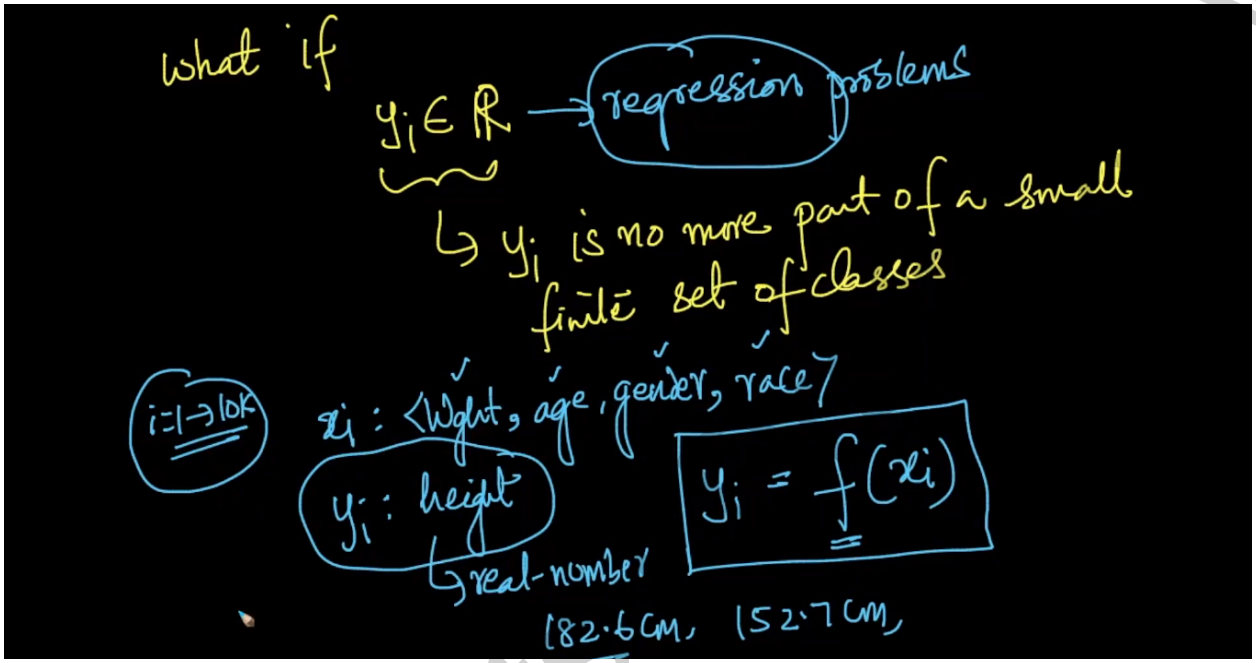
Similarly in the MNIST dataset, the class labels are the numbers from 0 to 9. If any query point is given, its label would be any one value from 0 to 9. As it has only 10 classes, such a classification problem is called a **10-class classification** problem (or) **multi-class classification** problem.

In both the examples mentioned above, the output class label comes from a finite set of values. Hence we call the problem of predicting such an output, a classification problem.

Regression - Definition

Regression is the problem of predicting a value for a new observation, on the basis of a functional relationship between two variables and on the basis of the training set of data containing observations whose value is already known.

Regression deals with predicting a quantitative (or) numerical response.



In the above example, we are predicting the height of an individual. The height value doesn't come from a finite set of values, but it comes from an infinite set of numerical values. As the output comes from an infinite set of numerical values, we call this problem a Regression problem.

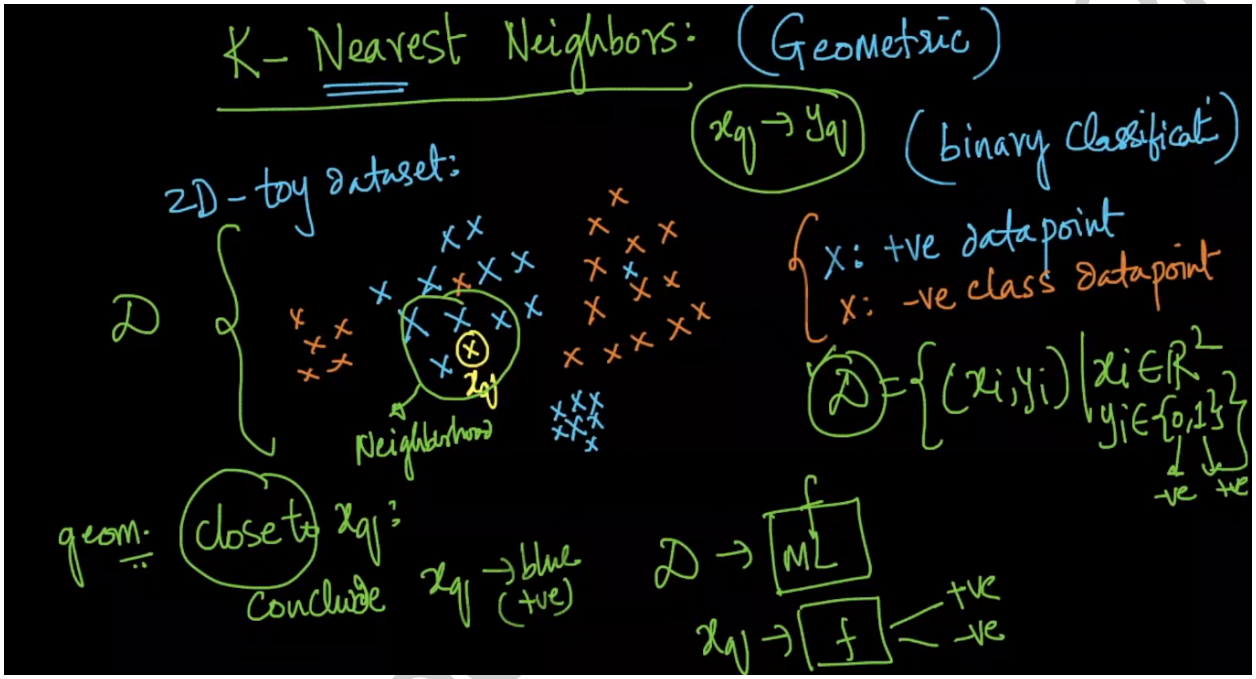
Note: As the output of a regression problem comes from an infinite set of numerical values, we denote it as $\mathbf{y_i \in R}$.

29.4 K-Nearest Neighbors Geometric Intuition with a toy example

Let us assume we are working on a binary classification problem and each observation belongs to either positive class or negative class. Let the dataset be in a 2-dimensional form. Here the dataset is represented as

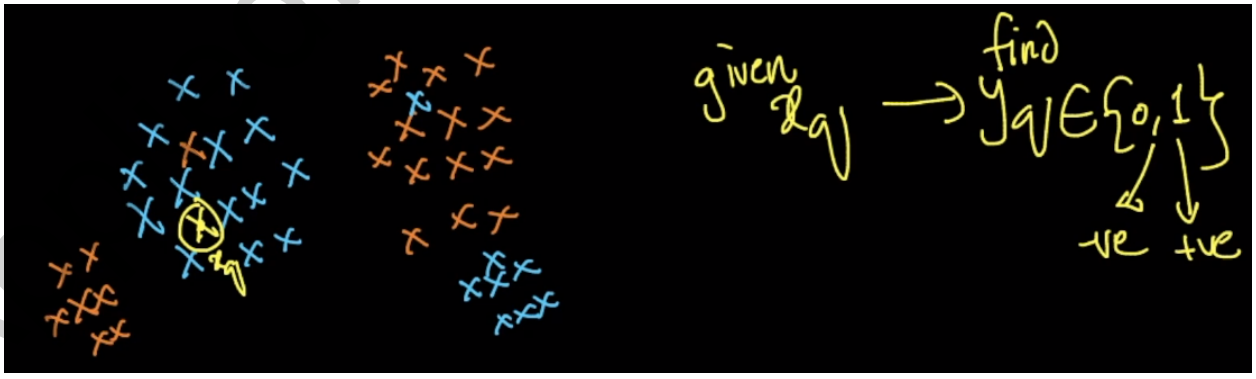
$\mathbf{D = \{(x_i, y_i)_{i=1}^{n} \mid x_i \in R^2, y_i \in \{0,1\}\}}$

The main purpose of classification is given a point 'x_q', we have to predict the value of 'y_q'. Below is an example that was discussed starting from the timestamp 0:18.



Procedure of KNN

Given a query point 'x_q', we have to predict the class label 'y_q'.



1) Compute the distance of the point 'x_q' to all the points in the training dataset.
2) Sort all the distances in the ascending order, and then select the top 'K' nearest points to 'x_q'.

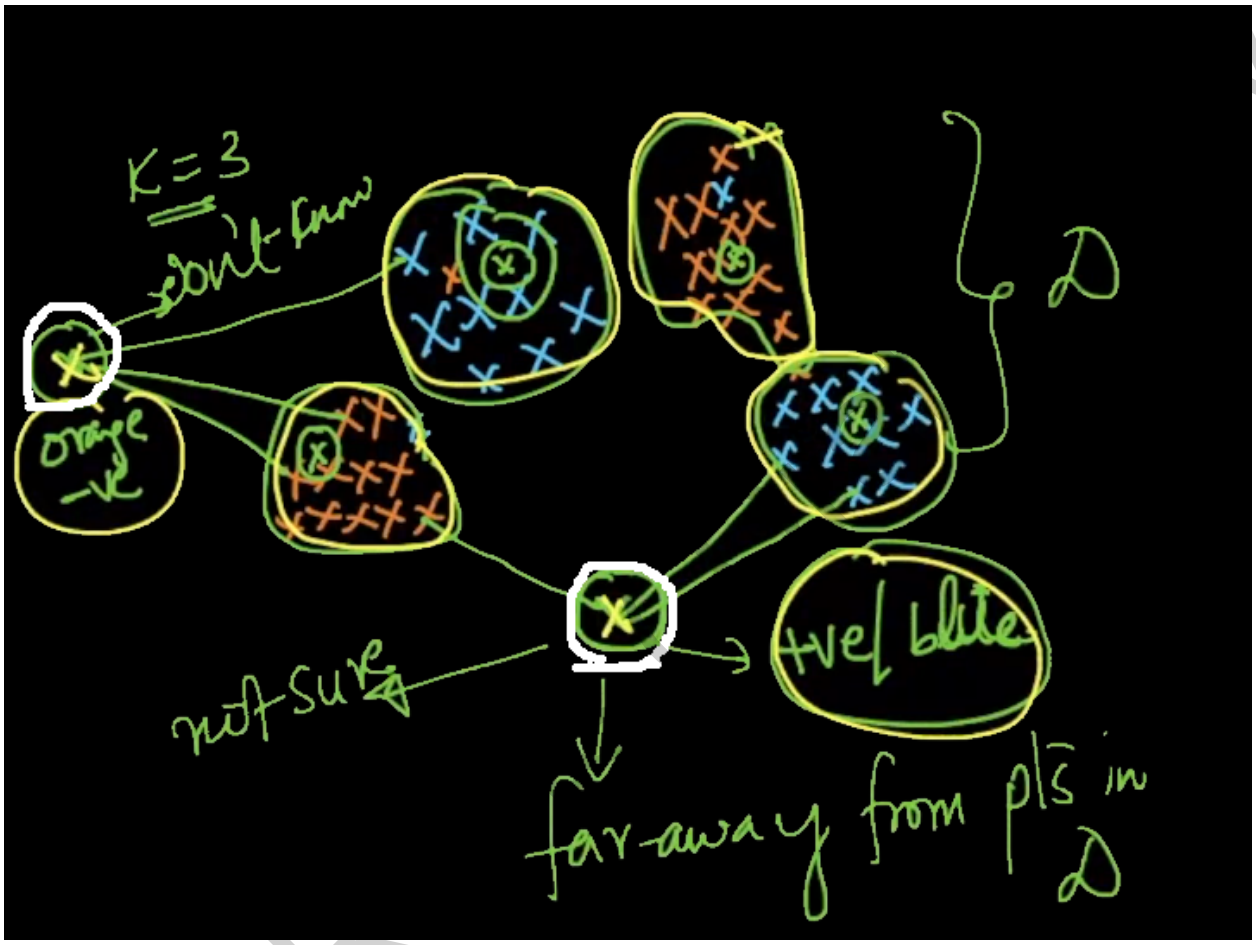
3) Let these 'K' points be $\{x_1, x_2, x_3, ..., x_k\}$ and their corresponding outputs may be $\{y_1, y_2, y_3, ..., y_k\}$.
   Here for the point 'x_q', the class to which majority of the class labels among $\{y_1, y_2, y_3, ..., y_k\}$ belong to, will be predicted as the output for 'x_q'.

a) For example, if K=3 and let $\{y_1, y_2, y_3\}$ = {+ve, +ve, -ve}, then 'x_q' will be assigned to the '+ve' class, as the '+ve' class is in majority.

b) For example, if K=5 and let $\{y_1, y_2, y_3, y_4, y_5\}$ = {+ve, -ve, -ve, -ve, +ve}, then 'x_q' will be assigned to the '-ve' class, as the -'ve' class is in majority.
   Here if 'K' is an even number, and if both the '+ve' and '-ve' class points in the neighborhood are equal in number, then it becomes difficult for the model to decide which class the datapoint 'x_q' should be assigned to. So it is always better to avoid even values of 'K' in K-NN.
   In cases where the 'K' value is even and the number of data points in '+ve' and '-ve' classes equal, then we should better increase the 'K' value, and repeat the same operation. Pick the class label with the majority of the points and assign it to 'x_q'.

29.5 Failure Cases of K-NN

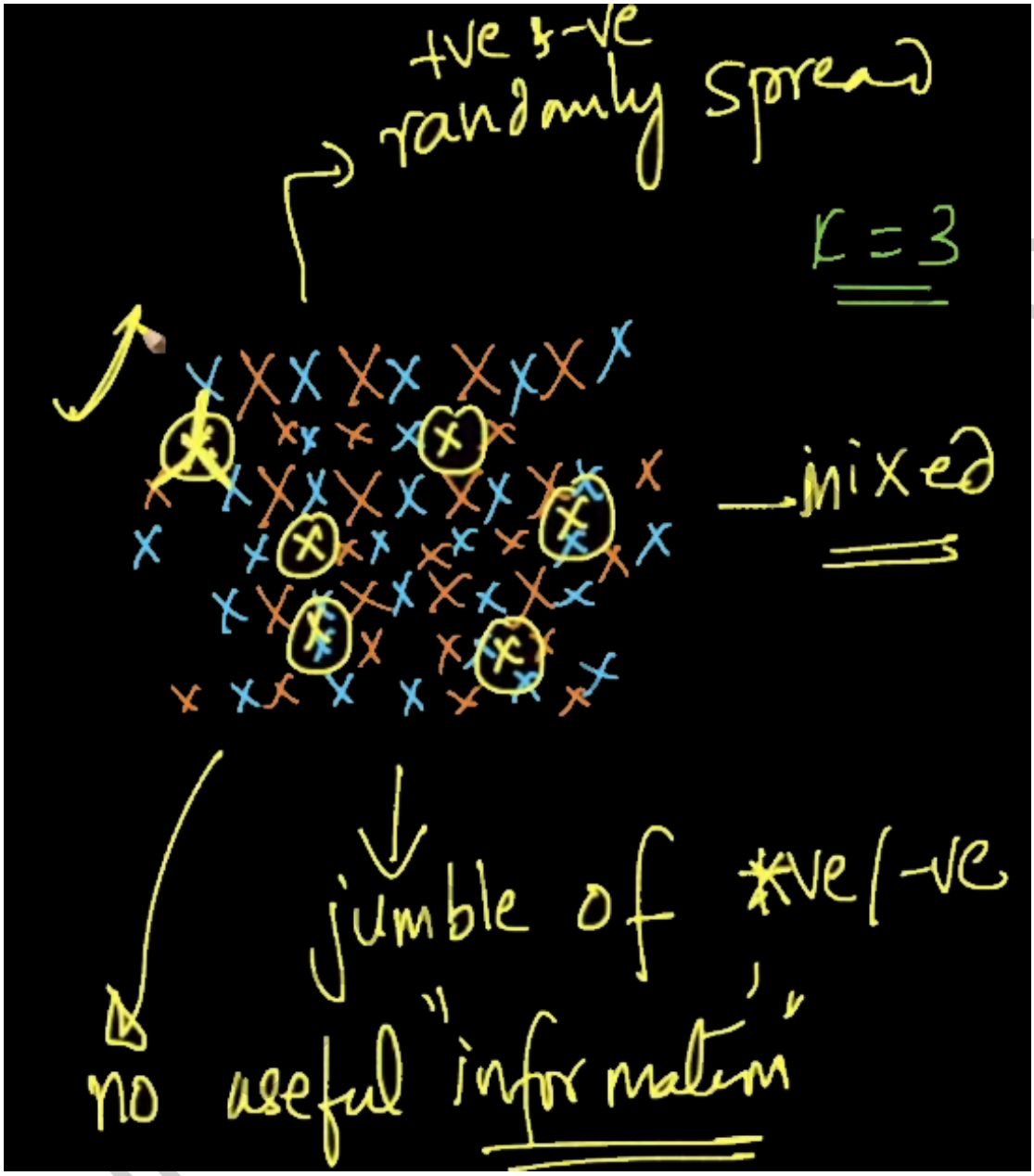
Case 1



If a point 'x_q' is surrounded by (or) nearby more number of '+ve' points cluster or '-ve' points cluster, then we can easily classify this point into one of them, depending on the majority vote of the class labels of the points in the neighborhood.

Now what if our query points are far away from all the points in the dataset like the two given query points circled in white. When we have a query point that is far away from the rest of the points in the dataset, then we are not sure which class this point has to be classified. Though the majority of the points which are nearer belong to a particular class, we still couldn't say anything about the class accurately, because as this particular point is far away from the regular points, it might exhibit a different behaviour(ie., it doesn't belong to the pattern of the points)

Case 2



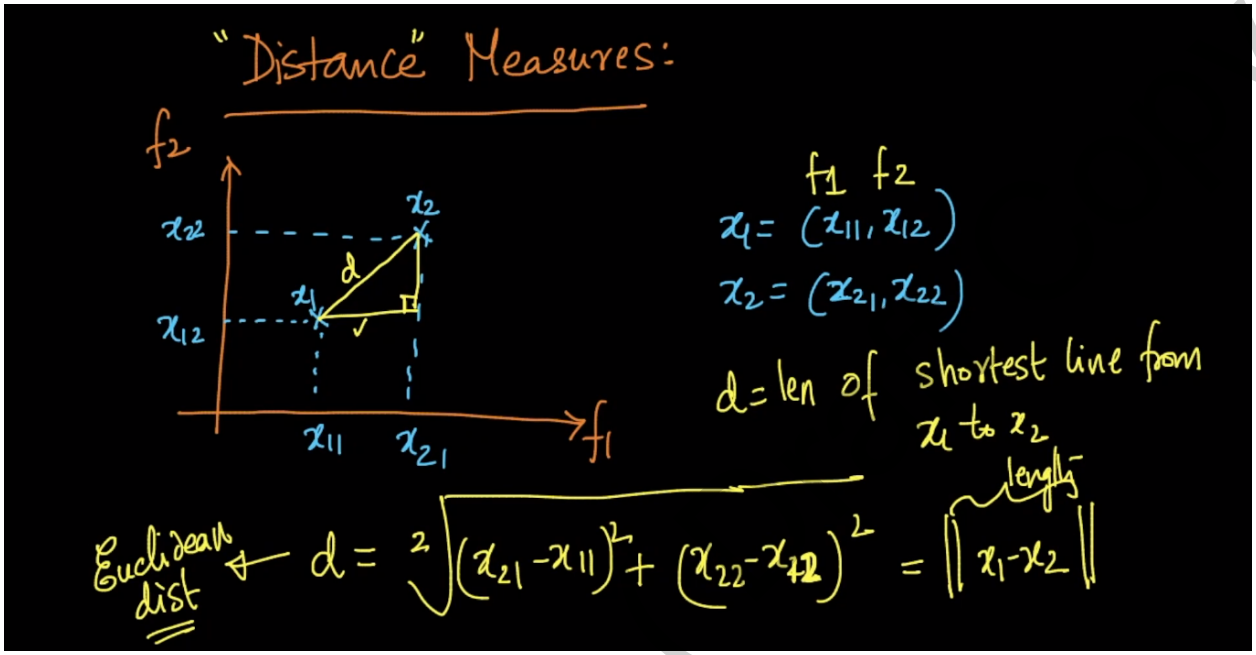
If the '+ve' and the '-ve' points are randomly spread, here we do not have exact and useful information. Here even if we predict the class label for the point 'x_q' as '+ve' (or) '-ve', it will result in an error. In this kind of situation where there is no useful information, KNN fails.

The above 2 mentioned cases are the simplest cases, where KNN fails.

Note: When the data points of different classes are randomly jumbled up, no model can work well. All models will fail. If a real world dataset shows this sort of behavior, it is better to change the feature set, as the current features are useless for classification.

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

Euclidean Distance (L2)



Let us assume, we have a 2-dimensional space and we have two points 'x_1' and 'x_2'.

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

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

d → Length of the shortest line from 'x_1' to 'x_2'

$d = sqrt((x_{21}-x_{11})^2 + (x_{22}-x_{12})^2) = ||x1-x2||$

Here 'd' → euclidean distance between the points 'x_1' and 'x_2'.

Let us assume we are working on a d-dimensional space.

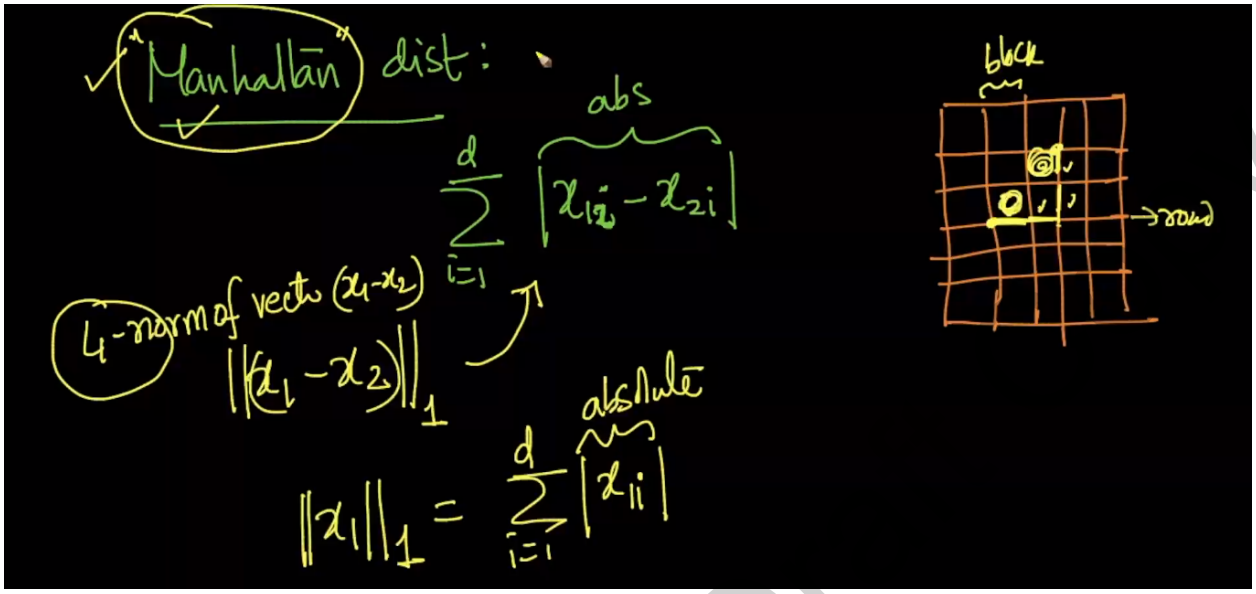
$x_1 \in R^d$, $x_2 \in R^d$

Euclidean Distance = $sqrt(\Sigma_{i=1}^{d} (x_{1i}-x_{2i})^2)$

$||x_1-x_2||_2$ → L2 norm of the vector (x_1-x_2)

$||x_i||$ → Distance of the point 'x_i' from origin = **$sqrt(\Sigma_{i=1}^{d} x_i^2)$**

Manhattan Distance (L1)



The Manhattan distance (d) between $x_1(x_{11}, x_{12})$ and $x_2(x_{21}, x_{22})$ is given as

Manhattan Distance = $\mathbf{\Sigma_{i=1}^{2} |x_{1i}-x_{2i}|}$

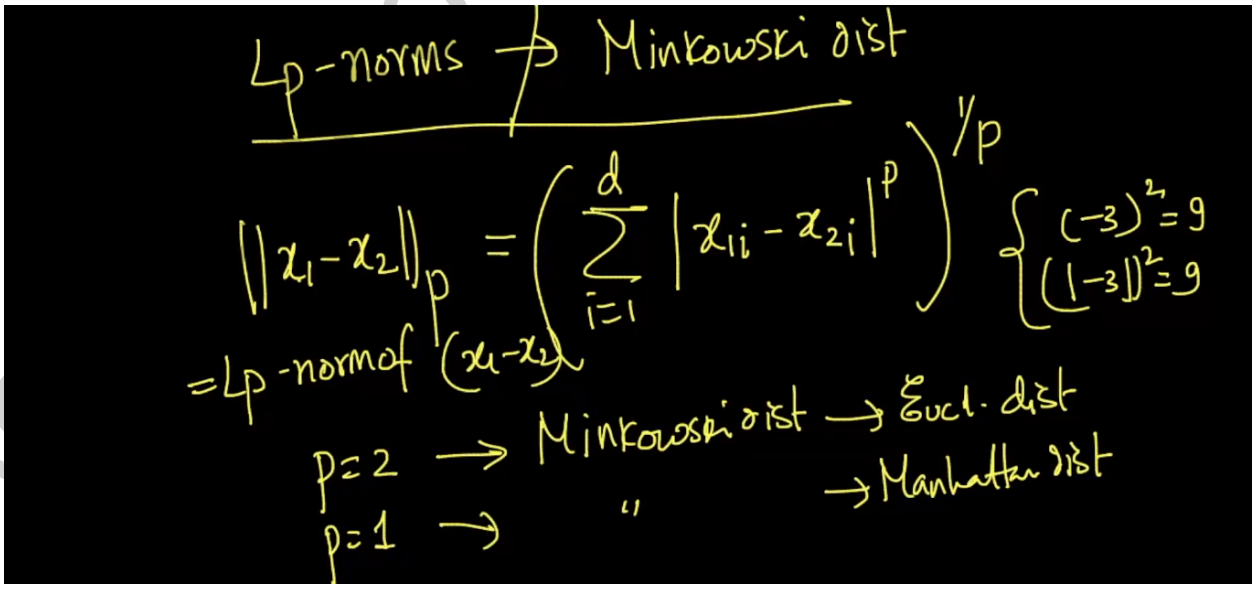
For a d-dimensional space, it is given as

Manhattan Distance = $\mathbf{\Sigma_{i=1}^{d} |x_{1i}-x_{2i}|}$

$||x_1-x_2|| \rightarrow$ L1 Norm of the vector (x_1-x_2)

$||x_1||_1 = \mathbf{\Sigma_{i=1}^{d} |x_{1i}|}$

Minkowski Distance (Lp)



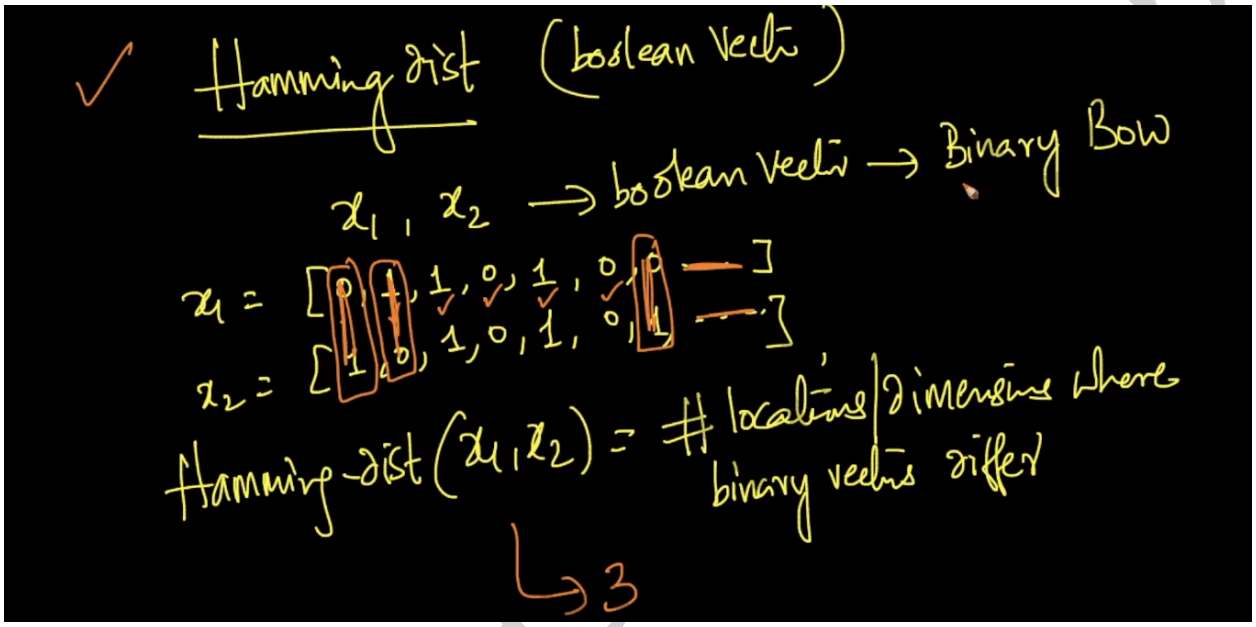
The Minkowski distance is the 'L_p' norm of the vector (x_1-x_2). The Minkowski distance in a d-dimensional space is given as

$||\mathbf{x_1}-\mathbf{x_2}||_p = (\Sigma_{i=1}^{d} |x_{1i}-x_{2i}|_p)^{1/p}$ (Here p>0 always)

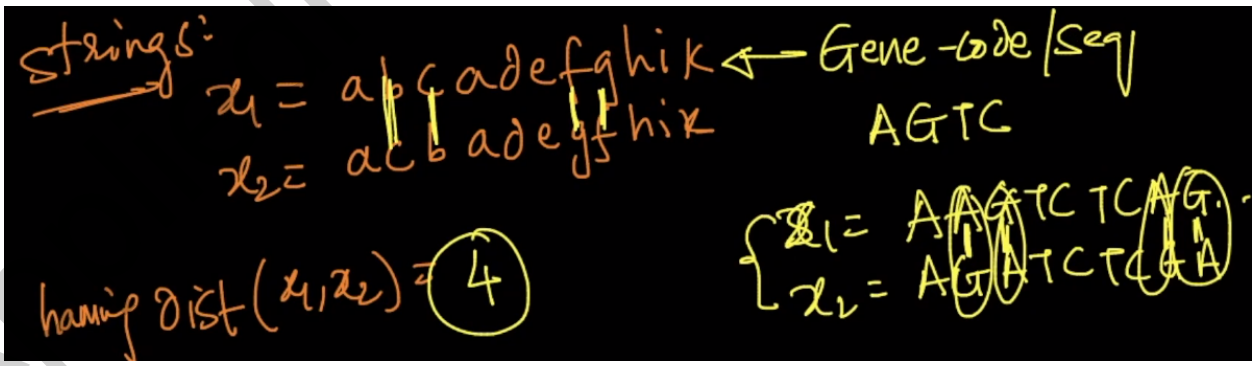
If p = 1, Minkowski Distance → Manhattan Distance

If p = 2, Minkowski Distance → Euclidean Distance

Hamming Distance



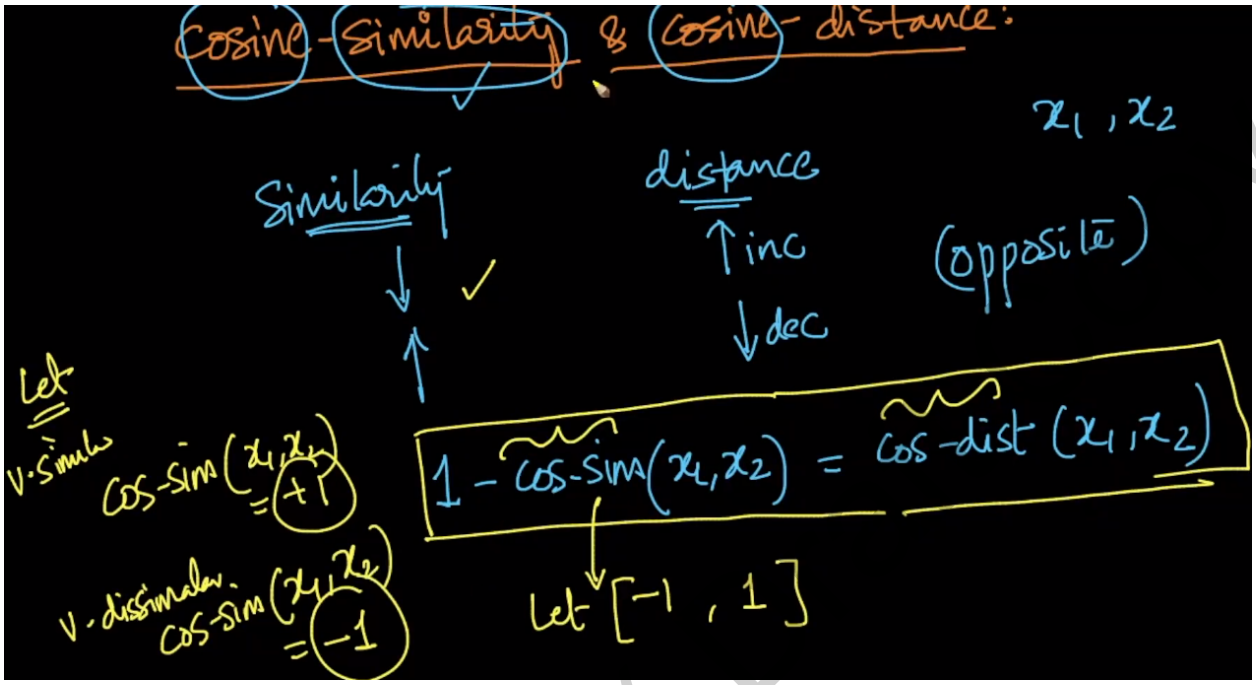
Hamming Distance between the two given vectors is the number of dimensions/features in which the two vectors differ. It is recommended to use Hamming Distance only on Binary Vectors. One best example is to apply Hamming Distance on the Binary BOW data. You also can apply it on Count Bsed BOW, but still you'll get the same result, as that obtained on the BOW data.



Hamming Distance between the two given strings is the total number of differing components in those strings. In the above example, we see there are 4 characters differing(2nd, 3rd, 7th, 8th locations). So the hamming distance is 4.

Note: Hamming Distance is no way related to the Minkowski Distance.

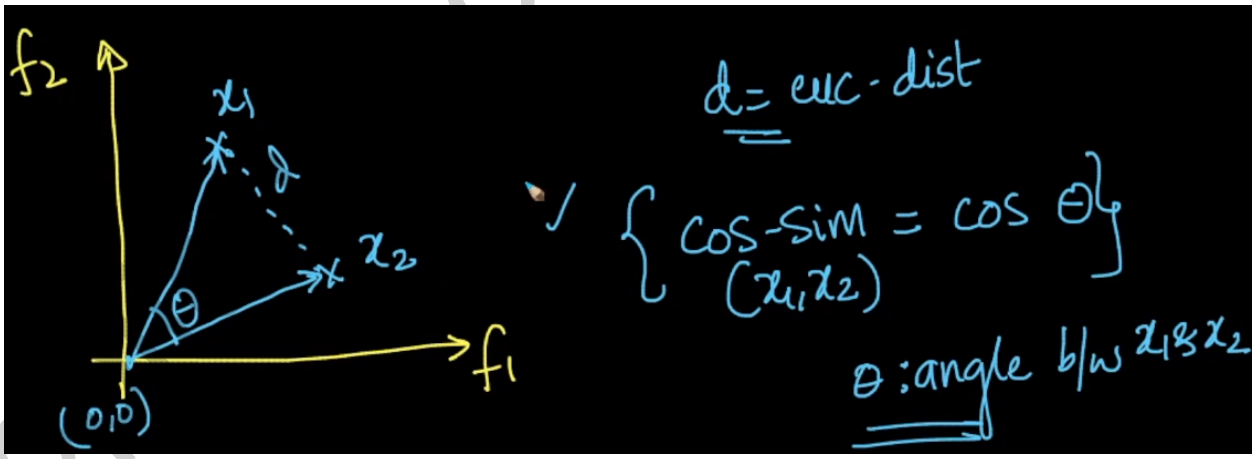
29.7 Cosine Distance & Cosine Similarity



Let us assume there are two vectors 'x_1' and 'x_2'. As the distance between the two points increases, the similarity between them decreases. Similarly if the similarity between two points increases, then the distance between them decreases.

The relationship between cosine similarity and cosine distance is given as

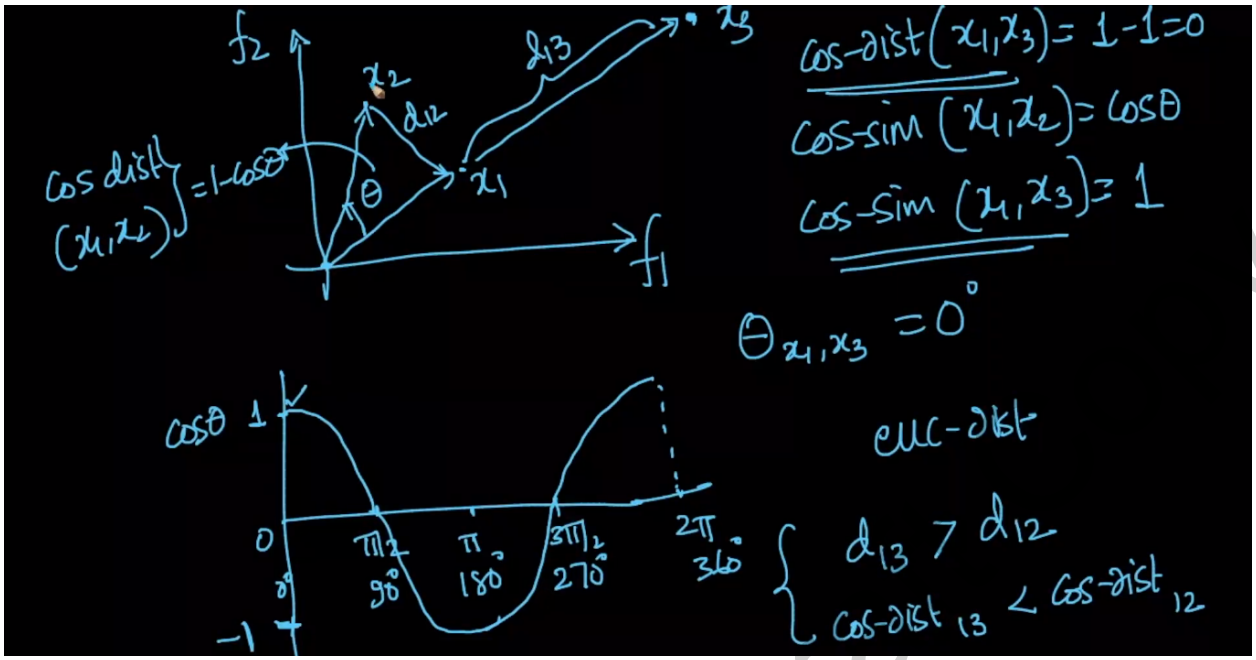
1 - cosine_similarity(x_1, x_2) = cosine_distance(x_1, x_2)



The cosine similarity between 'x_1' and 'x_2' is given by the cosine of the angle between the vectors 'x_1' and 'x_2'.

cosine_similarity(x_1, x_2) = cosθ where 'θ' is the angle between the vectors 'x_1' and 'x_2'.

Cosine Similarity is the measure of the similarity between two given non zero vectors.



Let us consider the 3 vectors 'x_1', 'x_2' and 'x_3'. According to the figure above,

cosine_similarity(x_1, x_2) = cosθ

cosine_similarity(x_1, x_3) = cos(0) = 1

cosine_distance(x_1, x_3) = 1 - cosine_similarity(x_1, x_3) = 1 - 1 = 0

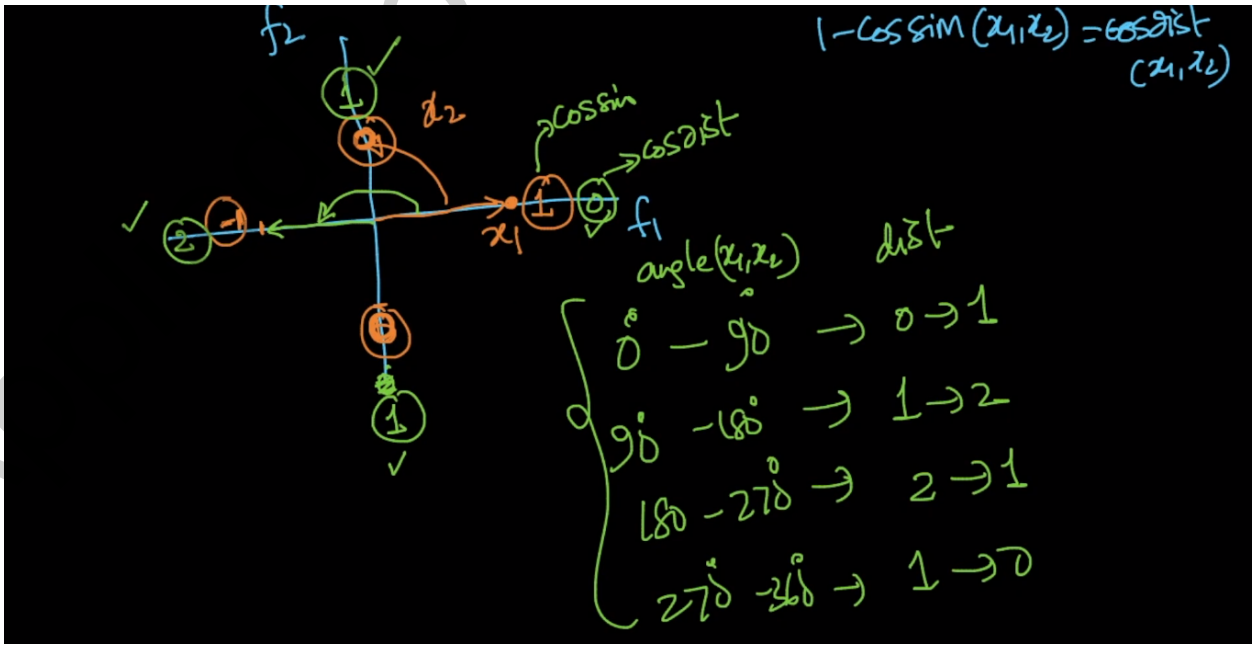
In case, if the angle between the vectors 'x_1' and 'x_2' is known, then

cosine_similarity(x_1, x_2) = cosθ

In case, if the angle between the vectors 'x_1' and 'x_2' is unknown, then

cosine_similarity(x_1, x_2) = $x_1.x_2/(||x_1||_2 * ||x_2||_2)$

If the vectors 'x_1' and 'x_2' are unit vectors, then $||x_1||_2 = ||x_2||_2 = 1$



If the angle between the two given vectors 'x_1' and 'x_2' lies in between 0^0 and 90^0 (or) 270^0 and 360^0(ie., $\theta \in [0, \pi/2]$ or $\theta \in [3\pi/2, 2\pi]$), then the cosine similarity lies in between 0 and 1. The cosine distance also lies in between 0 and 1.

If the angle between the two given vectors 'x_1' and 'x_2' lies in between 90^0 and 270^0((ie., $\theta \in [\pi/2, 3\pi/2]$)), then the cosine similarity lies in between -1 and 0. The cosine distance lies in between 1 and 2.

Note: Cosine Distance measures the angular difference between the vectors 'x_1' and 'x_2'. Cosine Similarity is used as a metric for measuring distance when the magnitude of the vectors does not matter.

Relationship between Euclidean Distance and Cosine Distance

If 'x_1' and 'x_2' are the two given vectors, then

$$||x_1-x_2||^2 = (x_1-x_2)^T(x_1-x_2) = ||x_1^2|| + ||x_2^2|| - 2x_1^Tx_2 \text{ ------- (1)}$$

$x_1^Tx_2$ is nothing but the dot product of the two vectors 'x_1' and 'x_2'.

Formula: *If we have two vectors 'A' and 'B', then*

$A^TB = A.B = ||A||*||B||*cos(\theta)$ *where '*θ*'* is the angle between the vectors *'A' and 'B'*.

So $$x_1^Tx_2 = x_1.x_2 = ||x_1||*||x_2||*\cos(\theta) \text{ -------- (2)}$$

In the above equation, 'θ' is the angle between the vectors 'x_1' and 'x_2'.

After substituting (2) in (1), we get

$$||x_1-x_2||^2 = ||x_1^2|| + ||x_2^2|| - (2*||x_1||*||x_2||*\cos(\theta)) \text{ -------- (3)}$$

Let us assume the given two vectors 'x_1' and 'x_2' are the unit vectors, then

$$||x_1|| = ||x_2|| = 1 \text{ -------- (4)}$$

After substituting (4) in the RHS of (3), we get

$$||x_1-x_2||^2 = 1 + 1 - (2*\cos(\theta))$$

$$||x_1-x_2||^2 = 2 - (2*\cos(\theta)) = 2(1 - \cos(\theta)) \text{ ---- (5)}$$

Here $||x_1-x_2||^2 \rightarrow$ Euclidean Distance between the vectors 'x_1' and 'x_2'.

$\cos(\theta) \rightarrow$ Cosine Similarity between the vectors 'x_1' and 'x_2'.

$1-\cos(\theta) \rightarrow$ Cosine Distance between the vectors 'x_1' and 'x_2'.

So now the equation (5) becomes

$$[\text{euclidean\_distance}(x_1, x_2)]^2 = 2*\text{cosine\_distance}(x_1, x_2)$$

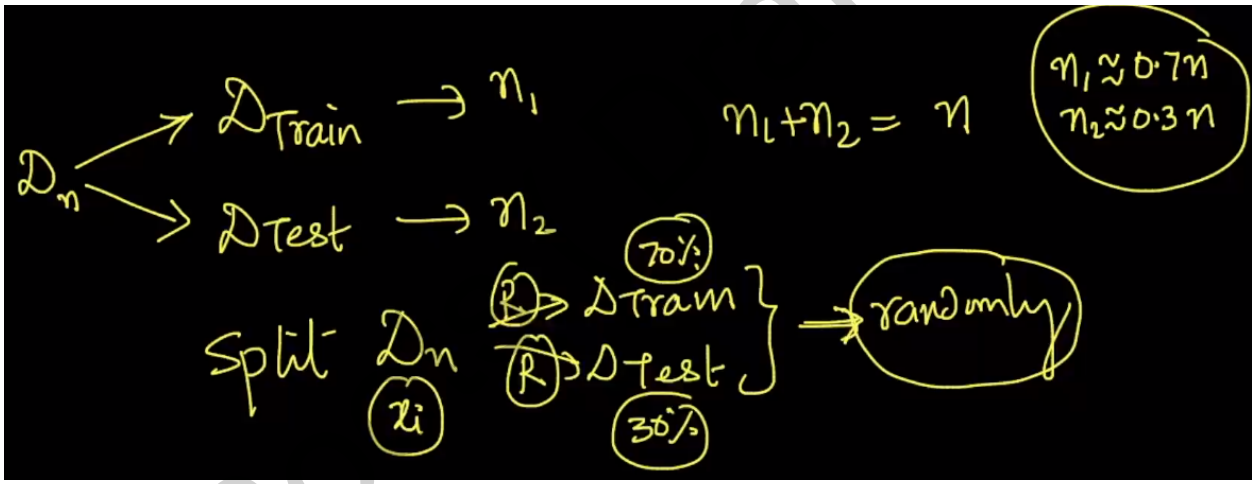
The above derived relationship is valid and applicable only if the two given vectors 'x_1' and 'x_2' are the unit vectors. (ie., $||x_1||_2 = ||x_2||_2 = 1$)

29.8 How to measure the effectiveness of K-NN?

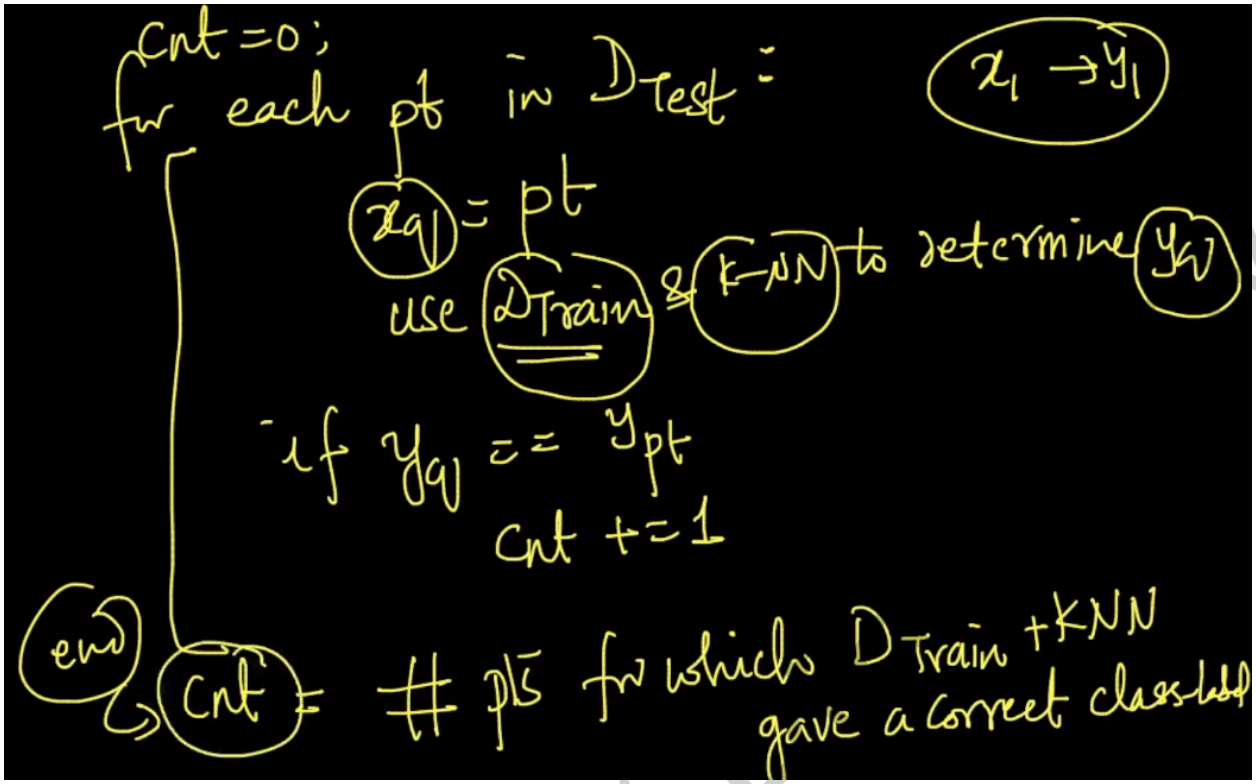
Let us consider the Amazon Fine Food Reviews Dataset which has got 364K reviews(after deduplication). For a given query point 'x_q', we have to predict the class label 'y_q'. Each data point is represented in the form of a numerical vector, and each data point has its own class label.

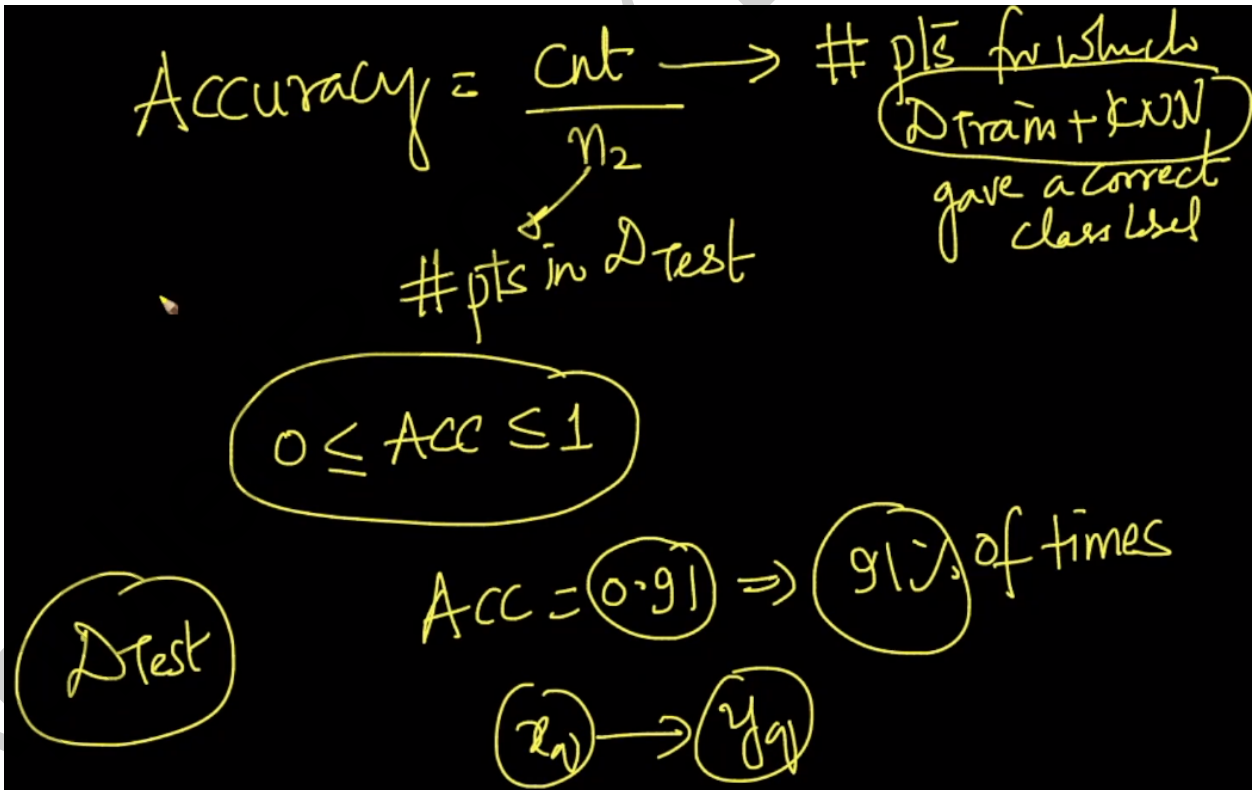
Procedure to measure the effectiveness of K-NN

1) Let us assume we are given a dataset $\{D_n\}$ and our inputs are $\{x_i\}_{i=1}^{n}$ and the outputs are $\{y_i\}_{i=1}^{n}$
2) Divide the dataset $\{D_n\}$ into the training set $\{D_{Train}\}$ and the test set $\{D_{Test}\}$. Let 'n_1' be the number of points in $\{D_{Train}\}$ and 'n_2' be the number of points in $\{D_{Test}\}$.
   ($n_1 + n_2 = n$)



3) Now we have to fit the KNN model on 'D_{Train}', so that the entire 'D_{Train}' gets stored. Then for each point 'x_q' in 'D_{Test}', we have to make predictions using the same KNN model and predict the value of 'y_q'.
4) Let us initialize a variable 'count = 0' and for every data point 'x_q' in 'D_{Test}', if **y_q == y_q'**, then increment the 'count' value by 1.
5) Finally we have to compute the accuracy using the formula
   Accuracy = count/(number of data points in 'D_{Test}') = count/n_2
   Accuracy value typically lies in between 0 and 1.





Note: If accuracy = 0.92, it means in 92% of the cases, using the fit on 'D_{Train}', the model predicts the output labels accurately.

29.9 Test/Evaluation Time and Space Complexity

For a data point 'x_q' in KNN, in order to classify the label for the data points, we also need to take time and space complexities into consideration.

Here the data points are represented as 'x_q' each and we have to predict their corresponding class label 'y_q'.

Inputs in KNN Algorithm: D_{Train}, K, $x_q \in R^d$

Output: y_q

Pseudo Algorithm

```
KNN_points = []
for each_x_i in D_Train:
    # Compute the distance between each_x_i and 'x_q'. (Takes O(d) as we have to compute the difference for all the 'd' dimensions and square it.
    # Keep the smallest 'K' distances and store them in KNN_points. It takes O(K)

# Let us assume 'K' is small. So let's ignore 'K' in the Time Complexity. Then after this,
    count_positive = 0, count_negative = 0
    for each_x_i in KNN_points:
        if y_i is +ve:
            count_positive += 1
        else:
            count_negative += 1
    if count_positive>count_negative:
        return y_q = 1
    else:
        return y_q = 0
```

Here the time complexity for the execution of the portion of code highlighted in yellow color is O(nd).

(It is because the 'for' loop has to traverse through the entire training dataset 'D_{Train}' of 'n' points which takes O(n) and again it has to compute the difference of the components and square them up, for all the 'd' dimensions, which required O(d). So the total time complexity for this task is O(nd))

The time complexity for the execution of the portion of the code highlighted in blue color is O(K). If 'K' is very small, then it would be O(1).

The time complexity for the execution of the portion of the code highlighted in green color is O(1).

So now the total time complexity to run this code = O(nd) + O(1) + O(1) = O(nd)
If d<<<n, then this total time complexity = O(n).

Now we shall check the space complexity. The space complexity is the total number of space locations needed to run the program. For the testing phase of KNN, the total space needed is O(nd).

Conclusion
Test Time Complexity for KNN = O(nd)
Test Space Complexity for KNN = O(nd)

29.10 K-NN Limitations

a) **High Space Complexity**
   Space Complexity of KNN = O(nd)
   Where 'n' → number of data points, 'd' → number of dimensions
   Let us now consider the Amazon Fine Food Reviews Dataset where n = 364K and d = 100K (let us assume, because we use techniques like BOW/TF-IDF)
   Now the data matrix contains 364K * 100K elements.
   So the total space needed = 364K * 100K = ~36GB of RAM is required. So the space complexity is so high.

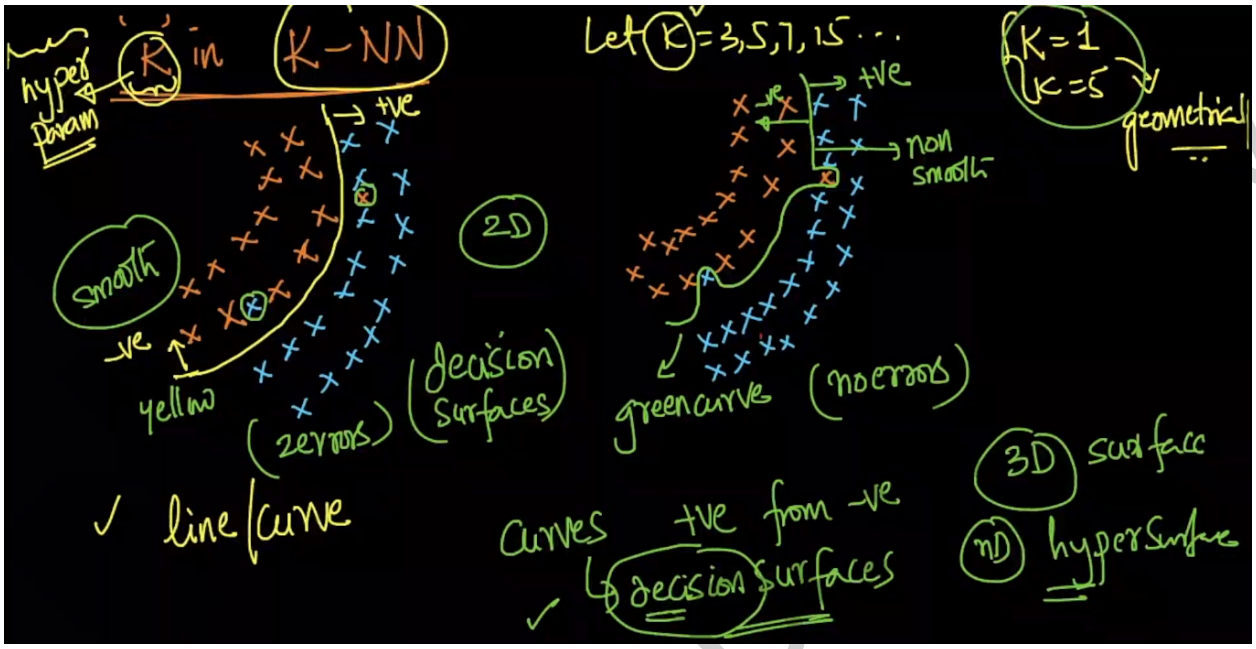
b) **High Time Complexity**
   In Amazon Fine Food Reviews Dataset, if a review is given, the system should be able to give the result within less than a millisecond.
   But here the system has to perform 36 billion computations. So the time complexity is so high.

Solutions to address this problem

a) The simple implementation of KNN has a time complexity of O(nd) and a space complexity of O(nd). So we have to implement KNN with better time and space complexities. **(or)**
b) Just change the algorithm. KNN is simply intuitive and elegant, but the reason for not being used is because of large time and space complexities.

Note: When we process such a huge amount of data on distributed systems, we'll have multiple threads processing the data. But still the time complexity can't be affected by the usage of multiple threads, whereas the total computation time reduces.

29.11 Decision Surface for K-NN as 'K' changes



'K' in K-NN is referred to as a Hyperparameter. Let us assume we have two different datasets and their training points are as shown above.

In the first dataset, we could see a smooth curve separating the '+ve' and the '-ve' classes. There are a few misclassifications, but still the curve is smooth. Whereas in the second dataset, the curve is not smooth, but all the points are classified perfectly. So when the curve is smooth, there are chances for a few classifications in this context, whereas if the curve is non-smooth, there are more chances for proper classifications.

The line/curve that separates the points belonging to two different classes is called a decision surface.

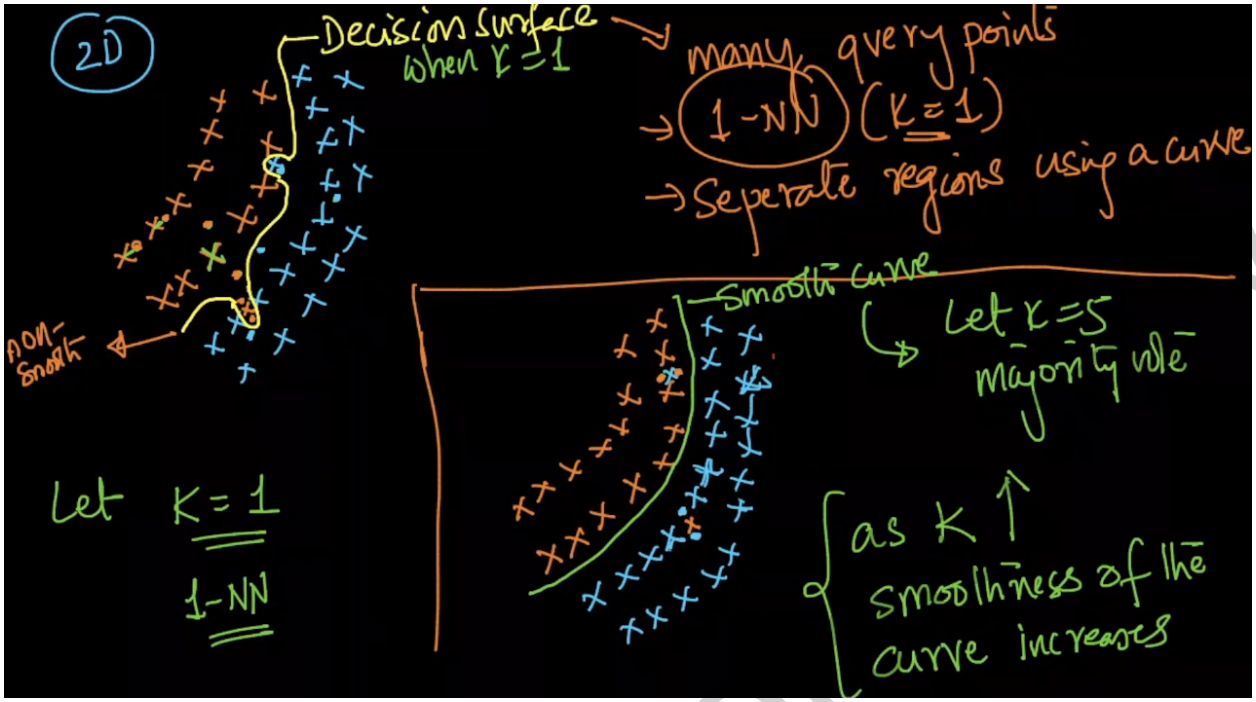
In 2-D space, we call the decision surface a line/curve.

In 3-D space, we call the decision surface a surface.

In n-D space, we call the decision surface a hyper-surface.

Let us examine the datasets in detail. When we look at the dataset, where the decision surface is non-smooth, we see all the points are classified correctly. In this case, if we consider K=1, as we have a '+ve' point lying in between a group of '-ve' points, and also a '-ve' point lying in between a group of '+ve' points.

So when K=1, if any query point has this exceptional blue point(ie., the blue point lying in between the range points) as it's 1-nearest neighbor, then the class assigned for this query point will be the same as that of the exceptional blue point.

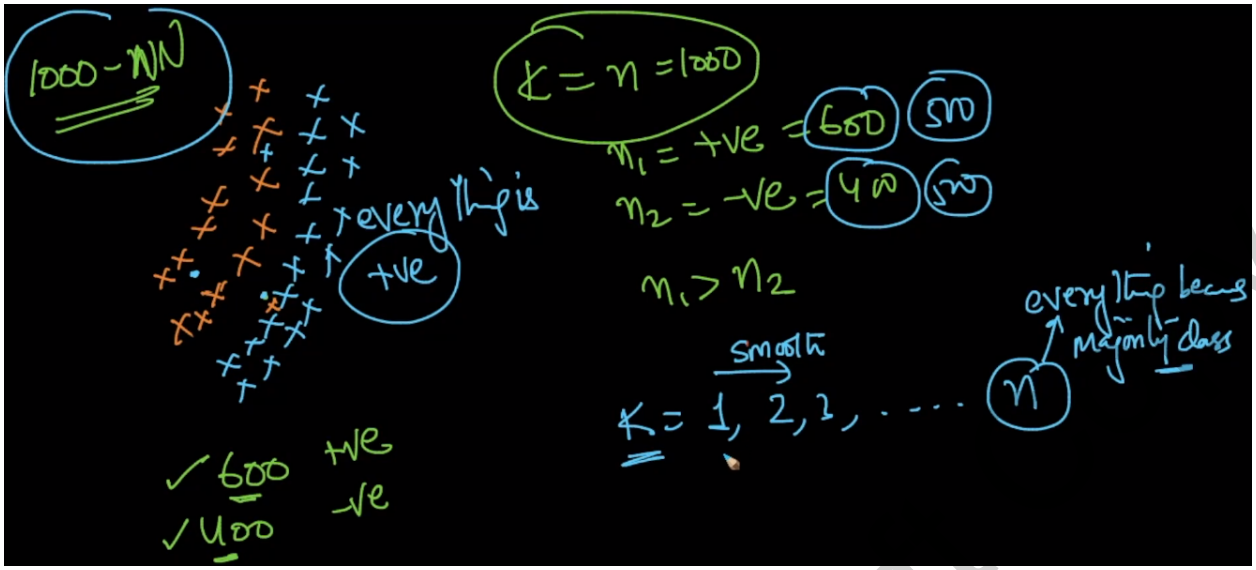


In case, if a new query point arrives with the exceptional orange point(ie.,the orange point lying in between the group of blue points) as its i-nearest neighbor, then the class label assigned for this query point will be the same as this exceptional orange point.

In both the above mentioned scenarios, if these exceptional points are not the 1-nearest neighbors, then the assigned class labels would definitely change. So here we can say that, for small values of 'K', the class labels of the points keep changing as we get different points into the neighborhood, and also the decision surface is non-smooth.

In the second example, we see the decision surface is smooth. Let us assume the 'K' value as 5, then even if we have one or two exceptional points, we do not see much changes in the polarities of those points which have these exceptional points as their neighbors. Because having these one or two couldn't show much impact when we consider 5 nearest neighbors.

So we can say, as the 'K' value keeps increasing, the decision surface becomes smoother.



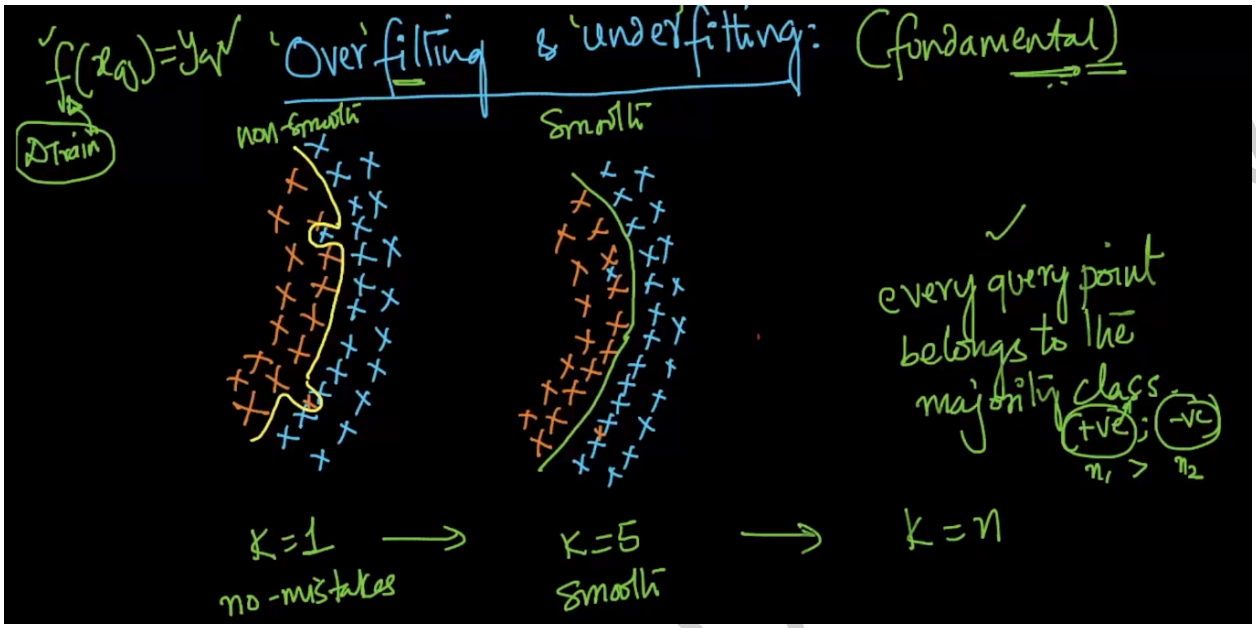
Let us now consider the case, where K=n(where 'n' is the total number of points in the training dataset). Here we do not see the model putting much effort for predictions. It blindly goes with the majority class.

For example, if our dataset has 1000 data points, out of which 600 belong to the '+ve' class, and the remaining 400 belong to the 've' class, then for every query point, the KNN model assigns the majority class label. In this case, all the query points will be assigned with the '+ve' class label. This happens in case of an imbalance dataset(where the number of points belonging to a particular class differ from that of another classes)

For example, if our dataset has 1000 data points, out of which 600 belong to the '+ve' class, and the remaining 400 belong to the 've' class, then for every query point, the KNN model assigns the majority class label. In this case, all the query points will be assigned with the '+ve' class label.

For example, if our dataset has 1000 data points, out of which 500 belong to the '+ve' class, and the remaining 500 belong to the 've' class, then the model becomes indecisive and couldn't assign the accurate class label. It picks one of the labels randomly and assigns it to the query point.

29.12 Overfitting and Underfitting



In the above scenarios, we can clearly observe that the decision surface becomes complex, if the 'K' value is low. It is because the KNN tries to take the least number of neighbors into consideration while predicting the class label of a query point. As we are taking only a small number of points in the neighborhood for consideration, the prediction on the query point can easily get affected.

For K=1, we take the nearest point into consideration. If we slightly increase the 'K' value (say K=3), then if the other class points(not the class label that was assigned when K=1) are in majority, then the class label prediction changes. So for small 'K' values, the decision surface keeps changing easily. Hence we see a complex decision surface. As the value of 'K' keeps increasing, the decision surface keeps getting smoother.

For example, if we consider K=5, then we could see the decision surface has become smoother when compared to that of K=1. Here the prediction we make would be of more confidence when compared to K=1. So as the 'K' value keeps increasing, we can give our predicted result with more confidence.

Similarly, if K = n (where 'n' is the total number of points in the training dataset), then it always predicts the majority class as the class label for a given query point. In this scenario, the model just gives the predictions blindly.

In overfitting, the decision surface does its maximum job to classify all the points accurately and hence it becomes non-smooth and more complex. In the case above where K=1, the exceptional points may be noise/outliers, but still the decision surface tries to classify them as accurately as possible. Hence the decision surface is working

extremely hard. In such a case, a small change in the 'K' value will affect the decision surface severely.

In underfitting, the decision surface is underworking and blindly gives the majority class label as the prediction. Hence even if you make a small change in the value of 'K', you do not find much difference in the predictions.

In case of a well-fit or best-fit, the decision surface is smoother and tries to classify the maximum number of points correctly (except the noise/outliers). Well-fit (or) Best-fit creates a balance between overfitting and underfitting.

Note:
If 'K' is small (say K=1), then the KNN model overfits
If 'K' is large (say K=n), then the KNN model underfits.
If 'K' is an intermediate value, then the KNN model fits perfectly.

	Train Accuracy	**Test Accuracy**	**Variance**	**Bias**
Overfitting	High	Low	High	Low
Best/Well Fit	Moderate/High	Moderate/High	Moderate	Moderate
Underfitting	Low	Low	Low	High

Note: In overfitting, if those exceptional points are noise/outliers, then the model makes a lot of errors because it has taken noise as the data, thereby creating a decision surface that is far from the truth. This decision surface is more prone(easily affected) to the outliers/noise.

In case of a well-fit model, those exceptional points are considered as noise and are ignored while building the decision surface. This decision surface is less prone to the outliers/noise when compared to the decision surface obtained with overfitting.

Q) When the model in overfitting is working more and harder to give perfect results, why don't we consider that?
Ans) It is because in overfitting, the model will work hard to give perfect results on the training data, but not on the test data. We actually care for the performance on the test data, rather than the performance on the training data. When the model overfits, it predicts well on the training data, but not on the test data.

When we have noise/outliers in our training data, if we consider overfitting, the model fits very well to all the points in the training data including the noise/outliers, but does not fit well on the test data points. Hence we ignore the models that overfit.

29.13 Need for Cross-Validation

The main purpose of performing cross-validation is to find out the optimal hyperparameter 'K' value for the K-NN model. Let our given dataset be represented mathematically in the form of a set as

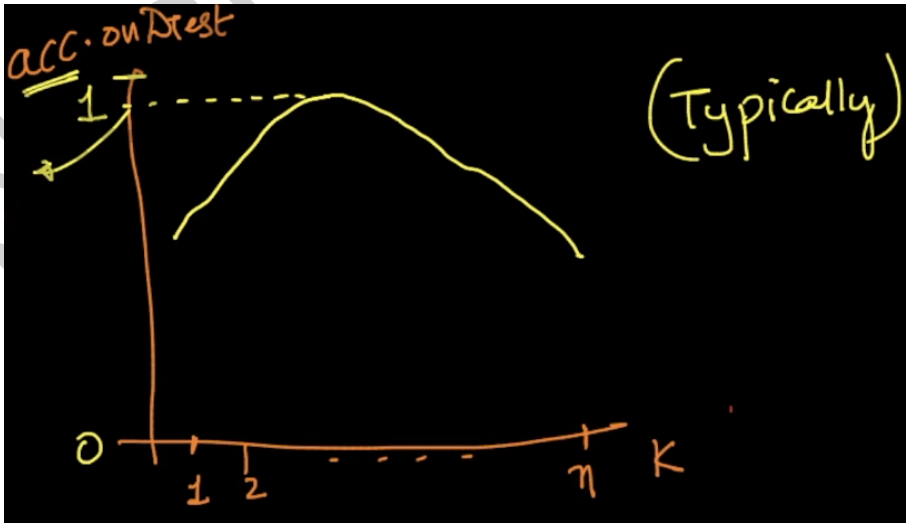
$$\mathbf{D = \{(x_i, y_i) | x_i \in R^d, y_i \in \{0,1\}\}}$$

How to choose the 'K' value?

1) Divide the dataset 'D' into the training set (D_{Train}) and the test set (D_{Test}).
2) For different values of 'K', fit the K-NN model on 'D_{Train}' and make predictions on 'D_{Test}' and then compute the test accuracy.

K	Accuracy on D_{Test}
K=1 K=2 . . K=5 . . .	$accuracy_1$ $accuracy_2$ . . $accuracy_5$ . . .

3) Build a 2D line plot with all the 'K' values on the 'X' axis and the corresponding Test 'Accuracy' scores on the 'Y' axis, as shown below. Whichever value of 'K' yields the highest test accuracy score, that would be chosen as the optimal value.



But here we have an issue. The main goal of machine learning is to learn a function 'f' and give the best accuracy on the unseen data. If the model doesn't give the best accuracy on the unseen data, then the model itself is useless.

Here we are using 'D_{Train}' to learn the function of finding out the nearest neighbors, 'D_{Test}' to find out the optimal 'K' value. There needs to be an unseen dataset on which we can measure the accuracy(Here 'unseen' in the sense, it is not used either for finding out the nearest neighbors or for finding out the optimal 'K'). So this concept of splitting the dataset into two parts(ie., D_{Train} and D_{Test}) fails, as this is not serving the actual purpose of Machine Learning.

For example, if we got an accuracy of 0.96 for K=6 on 'D_{Test}', it means that our model predicts the results correctly on 'D_{Test}' in 96% of the cases, when K=6. But as 'D_{Test}' has been used/seen by the model for finding out the optimal 'K', we couldn't say for sure that the same model could give the same accuracy on some unseen data.

When an algorithm works well for unseen future data, we call it **generalization**. The accuracy computed from the predictions made on such unseen data is called Generalization Accuracy. So we should split the dataset in such a way that we also get an unseen set of data points, to compute the generalization accuracy.

So we split the dataset into 3 parts. They are the training dataset (D_{Train}), cross-validation dataset (D_{cv}) and test dataset(D_{Test}).

D_{Train} → Used for learning the function 'f' to obtain the nearest neighbors

D_{cv} → Used for finding out the optimal 'K' value

D_{Test} → The unseen data used for computing the generalization accuracy.

Procedure for Cross-Validation (Simple Cross-Validation) on K-NN using D_{Train}, D_{cv} and D_{Test}

1) For different values of 'K', fit the K-NN model on 'D_{Train}'.
2) For every value of 'K', after fitting the model on 'D_{Train}', make predictions on 'D_{cv}' and compute the accuracy score for the predictions made on 'D_{cv}'.
3) Build a 2D line plot with all the 'K' values on the 'X' axis, and their corresponding cross-validation scores(ie., accuracies computed on 'D_{cv}') on the 'Y' axis.
4) Pick the value of 'K' which yields the highest cross-validation accuracy, and that will be your optimal 'K' value.
5) After obtaining the optimal 'K' value, we have to fit the K-NN model on 'D_{Train}' using the optimal 'K' value, and then make predictions on 'D_{Test}'(unseen data). Compute the accuracy for the predictions made on 'D_{Test}' and this accuracy is called 'Generalization Accuracy.'

Q) **Why do we need cross-validation?**

Ans) Cross-Validation is used to assess the predictive performance of the models and to judge how well they perform on unseen data.

The motivation to use the cross-validation mechanism is that when we fit a model, we are fitting it on the training dataset(D_{Train}). Without cross-validation, we only have the information on how our model performs on the seen data.

Ideally we would like to see how does the model perform when we have new data(ie., unseen data), in terms of accuracy of its predictions.

Q) **Which value of 'K' should be chosen, if multiple 'K' values give the highest cross-validation accuracy?**

Ans) If we get the same highest accuracy for multiple values of 'K', then if we need the test time to obtain the nearest neighbors to be low, we should go for the least value of 'K' among those which give the highest cross-validation accuracy.

If we have lots of data, and.test time is of no concern, then picking the largest value of 'K' is preferable, as we can have more trust in the decision, as there are more points supporting our decision/class label.

29.14 k'-Fold Cross-Validation

So far we have seen the dataset 'D' being split into 3 parts. They are D_{Train}(60%), D_{cv}(20%) and D_{Test}(20%). We have been using 'D_{Train}' to find out the nearest neighbors and 'D_{cv}' to find out the optimal 'K' value, but because of this, after obtaining the optimal 'K', we are training the final optimal model on 'D_{Train}' and making predictions on 'D_{Test}'. The 'D_{cv}' is not at all used anywhere after getting the optimal 'K'. So it goes to waste.

In order to make the maximum usage of our data and not letting any subset of data go to waste, we have come up with a strategy called K-fold Cross-Validation. For this, we have to split the dataset 'D' only into two parts. They are 'D_{Train}'(80%) and 'D_{Test}'(20%). The 'D_{cv}' gets created internally during the cross-validation. More the data used for the training, the more the model's predictive power would be. But we cannot skip the test data(D_{Test}). So the 'D_{cv}' would be created from D_{Train}.

Procedure for k'-Fold Cross-Validation

1) The dataset 'D' is split into 'D_{Train}' and 'D_{Test}'.
   D_{Train} → Finding out the Nearest Neighbors and obtaining the optimal 'K'(hyperparameter of K-NN) value
   D_{Test} → Unseen data on which the final model has to run and compute the accuracy.
2) The training data (D_{Train}) is divided into k' parts(let us assume k'=4 for now).
   So the 4 parts would be 'D_1', 'D_2', 'D_3' and 'D_4'.

D_{Train}

D_1	D_2	D_3	D_4

In k'-fold CV, the number of parts into which 'D_{Train}' is divided is equal to **k'**.

3)

	D_{Train}	D_{cv}	Accuracy on D_{cv}
K=1	$D_1D_2D_3$	D_4	$a_4^{(1)}$
K=1	$D_1D_2D_4$	D_3	$a_3^{(1)}$
K=1	$D_1D_3D_4$	D_2	$a_2^{(1)}$
K=1	$D_2D_3D_4$	D_1	$a_1^{(1)}$
K=2	$D_1D_2D_3$	D_4	$a_4^{(2)}$
K=2	$D_1D_2D_4$	D_3	$a_3^{(2)}$
K=2	$D_1D_3D_4$	D_2	$a_2^{(2)}$
K=2	$D_2D_3D_4$	D_1	$a_1^{(2)}$
.	.	.	.

Avg. Accuracy Score for K=1 → $\mathbf{(a_1^{(1)}+a_2^{(1)}+a_3^{(1)}+a_4^{(1)})/4}$. Let us denote this as '$\mathbf{a_{k=1}}$'.

Avg. Accuracy Score for K=2 → $\mathbf{(a_1^{(2)}+a_2^{(2)}+a_3^{(2)}+a_4^{(2)})/4}$. Let us denote this as '$\mathbf{a_{k=2}}$'. Like this, we have to compute the average CV scores for all the values of 'K'. (Here 'K' is the hyperparameter in K-NN)

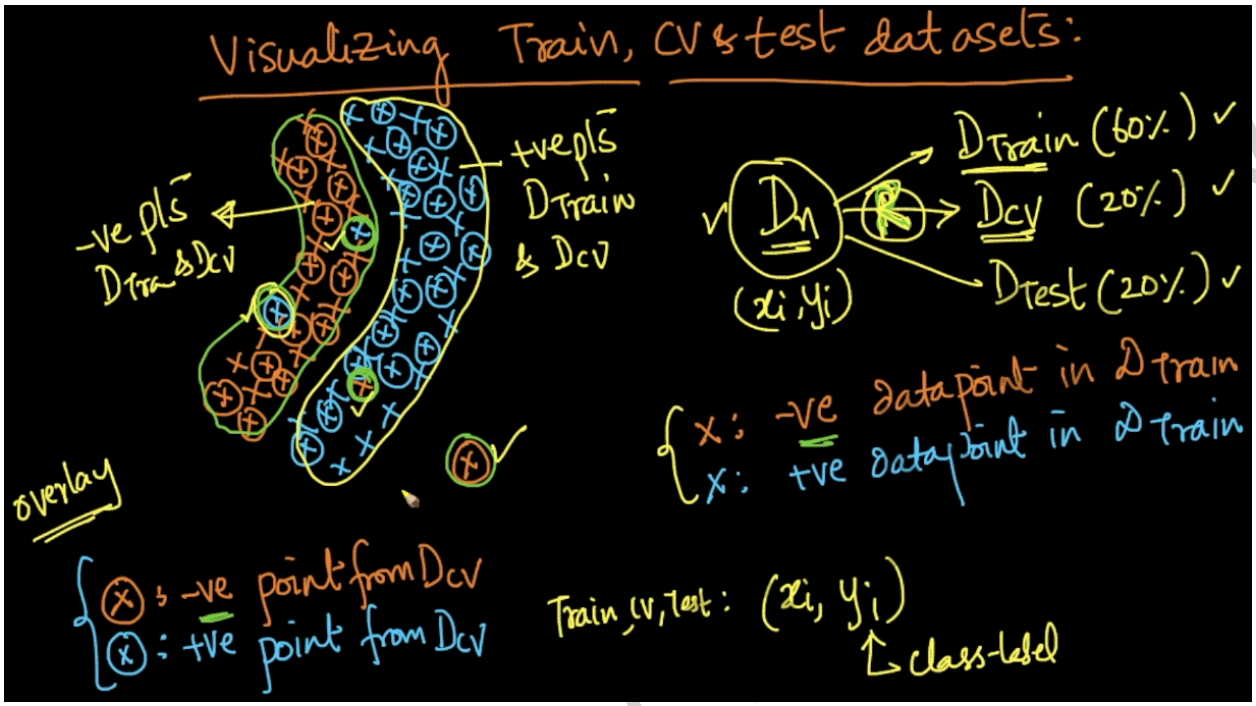
4) We should now plot a 2D line plot, with the 'K'(hyperparameter in K-NN) values on the 'X' axis, and their corresponding average CV accuracies on the 'Y' axis.

   Whichever value of 'K' gives the highest average CV accuracy score, that would be considered as the optimal 'K' value.
5) Fit the final K-NN models using the obtained optimal 'K' value on the whole 'D_{Train}', and make predictions on 'D_{Test}'(unseen data), and compute the final test accuracy score.

Note: In k'-fold cross-validation, the training data 'D_{Train}' is divided into k' folds and each fold is used in both the training and the cross-validation phases. The most typically used values for k' are 3 (or) 5.

K'-fold cross-validation gives the optimal value of the hyperparameter such that the results obtained on the unseen data are more generalized.

29.15 Visualizing train, validation and test datasets



For now we shall split the dataset 'D_n' into 3 parts. They are 'D_{Train}'(60%), 'D_{cv}'(20%) and 'D_{Test}(20%)'. Every data point (irrespective of whether it is present in 'D_{Train}', 'D_{cv}' and 'D_{Test}') is represented as (x_i, y_i).

Observations:

1) 'D_{Train}' and 'D_{cv}' do not overlap perfectly.
2) If there are many +ve/-ve points from 'D_{Train}' in a region, then it is highly likely to find many +ve/-ve points from 'D_{cv}' in that region.
3) If there are a few +ve/-ve points in a region from 'D_{Train}', then it is very unlikely to find +ve/-ve points from 'D_{cv}' in that region. Such points are noise/outliers.

All the above 3 observations are True, as long as 'D_{Train}' and 'D_{cv}' are randomly sampled. We also could see the above observations between 'D_{Train}' and 'D_{Test}', if they both are randomly sampled.

29.16 How to determine overfitting and underfitting?

Either by using Simple cross-validation (or) k'-fold cross-validation, we get the best value of 'K'(here hyperparameter in K-NN) for the model, which neither leads the model to overfit nor to underfit.

Accuracy = (Number of points correctly classified)/(Total number of points)

Error = 1 - Accuracy

Our main aim is always to maximize the accuracy and minimize the error.

For now, we shall consider the case of simple cross-validation, for that we have to split the dataset 'D_n' into 'D_{Train}', 'D_{cv}' and 'D_{Test}'.

In simple cross-validation, we use the 'D_{Train}' to find the nearest neighbors, 'D_{cv}' to find the optimal 'K' value and 'D_{Test}' to find out the model performance at prediction on the unseen data.

Training Error

We have to fit the model on 'D_{Train}' and make predictions on the same 'D_{Train}'. Here we come across a few misclassifications while predicting the class labels of 'D_{Train}'. This error obtained is called the **Training Error**.

Cross-Validation Error

We have to fit the model on 'D_{Train}' and make predictions on 'D_{cv}'. Here we come across a few misclassifications while predicting the class labels of 'D_{cv}'. This error obtained is called the **Cross-Validation Error**.

Test Error

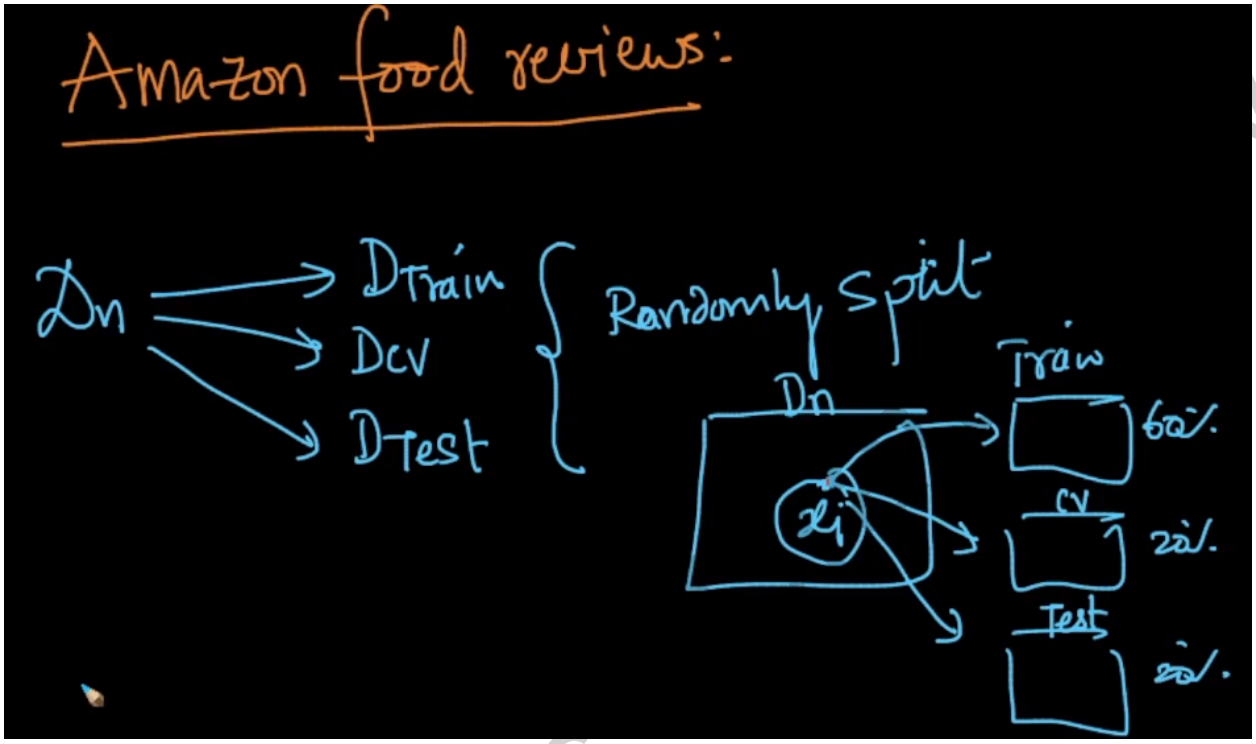
We have to fit the model on 'D_{Train}' and make predictions on the same 'D_{Test}'. Here we come across a few misclassifications while predicting the class labels of 'D_{Test}'. This error obtained is called the **Test Error**.

Overfitting (vs) Underfitting

If the **Training Error is high** and the **Cross-Validation Error is high**, then we call it **Underfitting**.
If the **Training Error is low**, but the **Cross-Validation Error is high**, then we call it Overfitting.
If the **Training Error is moderate** and the **Cross-Validation Error is moderate**, then we call it the **Best Fit**. (Here both the errors are close enough)

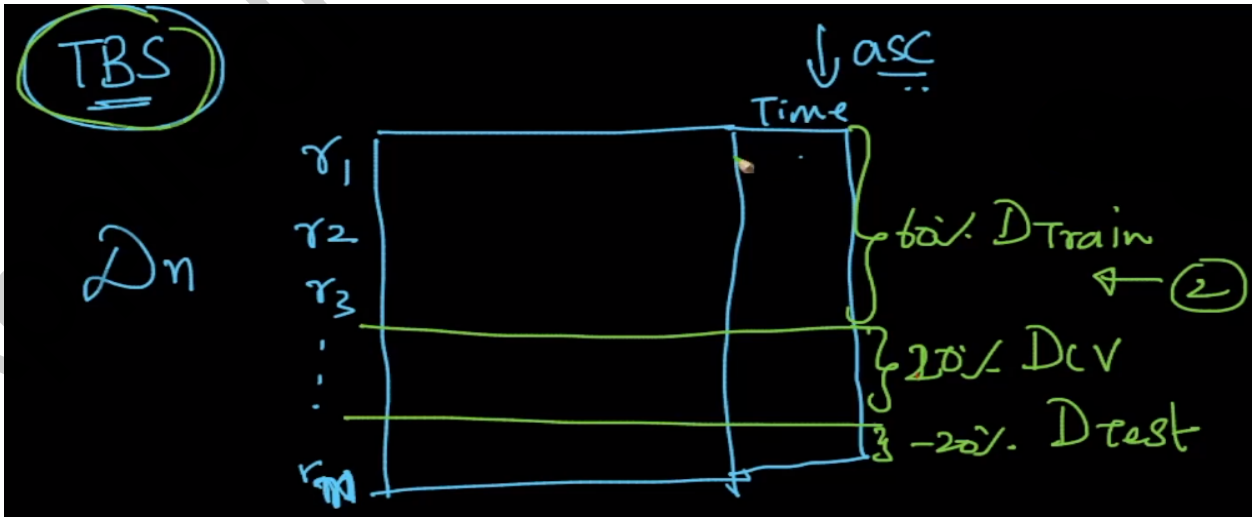
29.17 Time Based Splitting

So far we have split the dataset into D_{Train}, D_{cv} and D_{Test} using random splitting.



We also do have another strategy of splitting the dataset which is known as Time Based Splitting. There are certain problems where Random Splitting is better than Time Based Splitting, and in some problems, Time Based Splitting is better than Random Splitting.

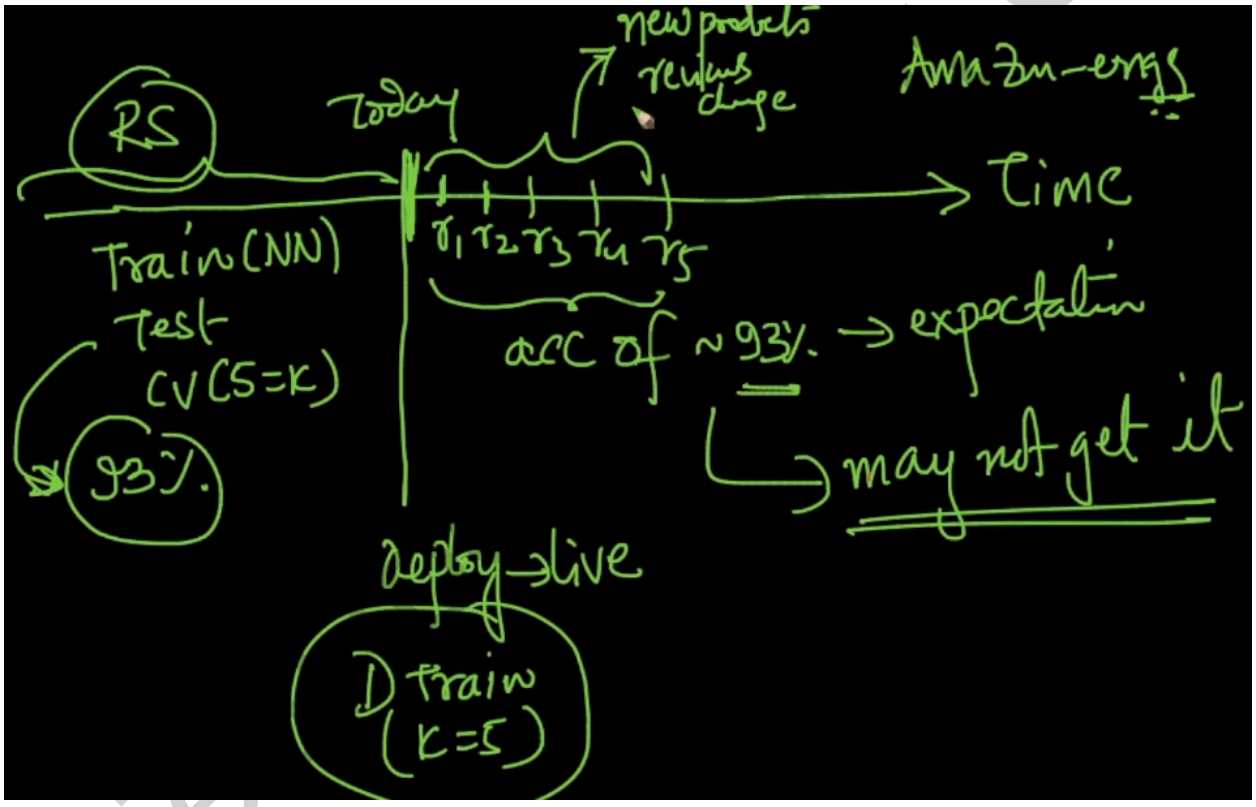
In order to perform Time Based Splitting, we need to have the timestamp column in the dataset.



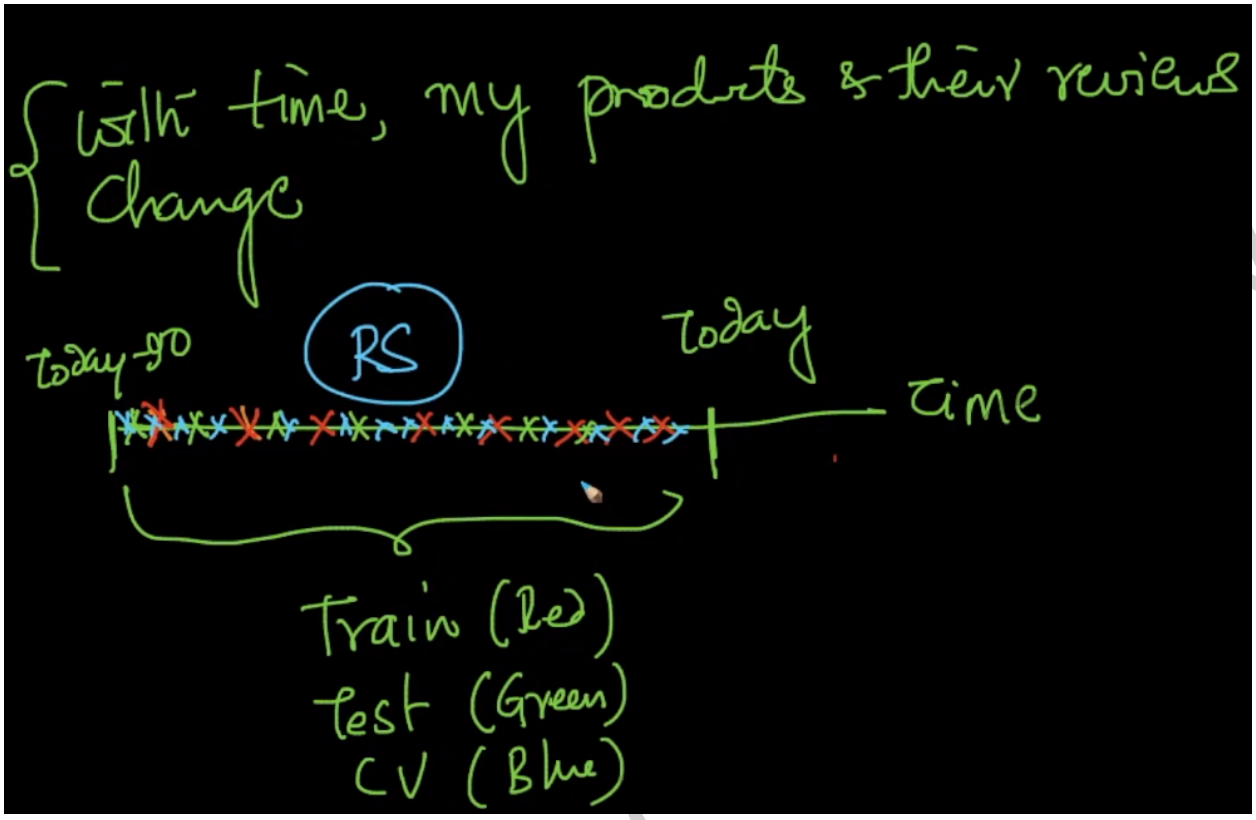
Procedure to perform Time Based Splitting

1) Sort the dataset 'D_n' in the ascending order of the time column values.
2) Now we split the dataset with the first 60% of the points into 'D_{Train}', the next 20% of the points into 'D_{cv}', and the last 20% of the points into 'D_{Test}'.

In case of Random Splitting, we had
D_{Train} → For finding the nearest neighbors
D_{cv} → For choosing the optimal 'K' value
D_{Test} → For measuring the accuracy



The above figure shows applying random splitting on a real-time dataset. Here for the products, the reviews would change as the time keeps progressing. It doesn't mean the already existing reviews would change, but as the time keeps progressing, we do get many more reviews on the similar products purchased, and the reviews/feedback would change. In such a case, if we perform a random splitting, there are chances for a few of the latest reviews to go into the D_{Train} and D_{cv}, and a few of the older reviews to go into D_{Test}. Due to this, the model once trained in the past and gave a good score on D_{Test}, if it is deployed, might not give a similar score on the latest data. It is because as the patterns in the data keep changing, the model performance also keeps changing.

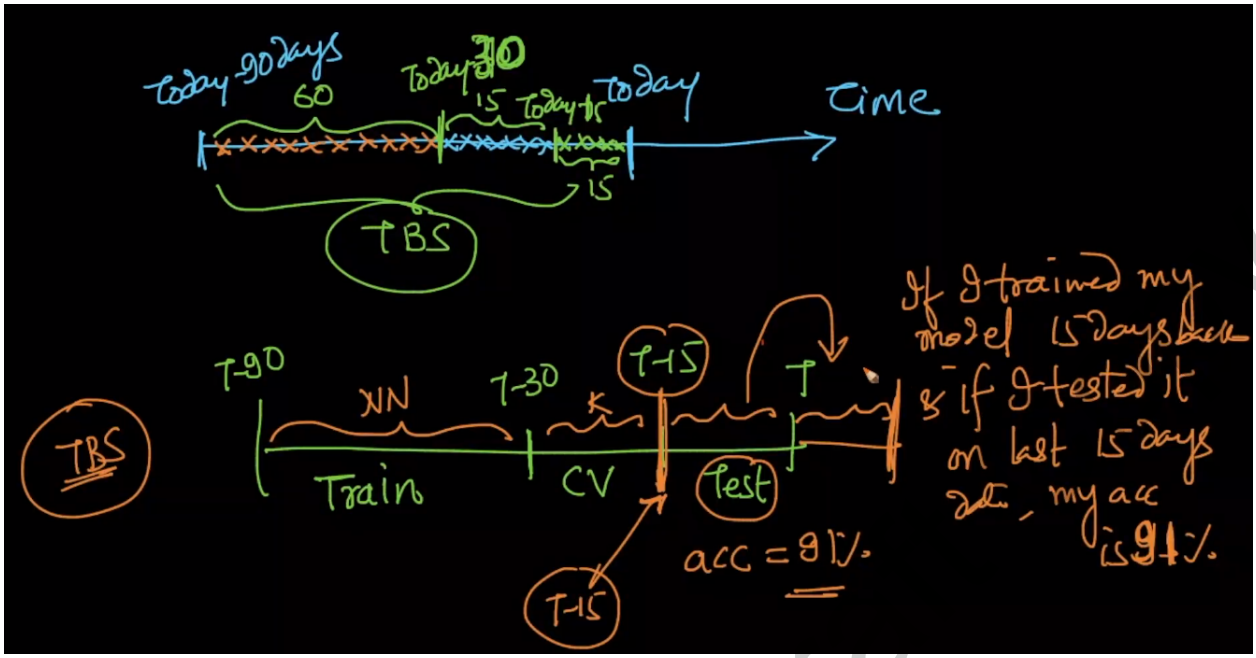


Also there are chances for new products to get added, and also the older products to disappear. In case, if a new product gets added today on the website, and if there is no data about similar products previously, then we couldn't make a valid prediction. Similarly, if we have a product that is no more available for sale, then it is always better to keep it in D_{Train} because, if it comes in D_{Test}, then making predictions again on it, is just a waste of time and meaningless, as that product is never going to be added again for sale. Hence in order to get rid of all these kinds of problems, we go for Time Based Splitting.

In case of the time based splitting, we split the older data into D_{Train} and D_{cv}, and then the future data would be D_{Test}.

Whenever we apply Random Splitting, if we train, test and deploy the model, and if it gives an accuracy of say x%, then in future the same model might not give similar accuracy for the future data, whereas whenever we apply time-based splitting, if we train a model with a particular set of data belonging to a particular interval, cross-validate on a particular interval and test the model on a particular interval of data and finally deploy it, then the accuracy obtained on the test data will be the almost the same for the future data as well.

Note: People wrongly understand that the consistency in the accuracy as mentioned above is due to the presence of the 'Time' column in our dataset. It is not because of the presence of the 'Time' column, but due to the way in which we split the dataset.



Note: Whenever the 'Time' column is available in the dataset, and if things/behavior of the data changes over the time, then time-based splitting is preferred over random splitting.

In case, if the 'Time' column is not present in the dataset, then we have to go for Random Splitting, as we have no other option.

29.18 K-NN for Regression

The dataset for a binary classification task is represented as

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

The dataset for a regression task is represented as

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

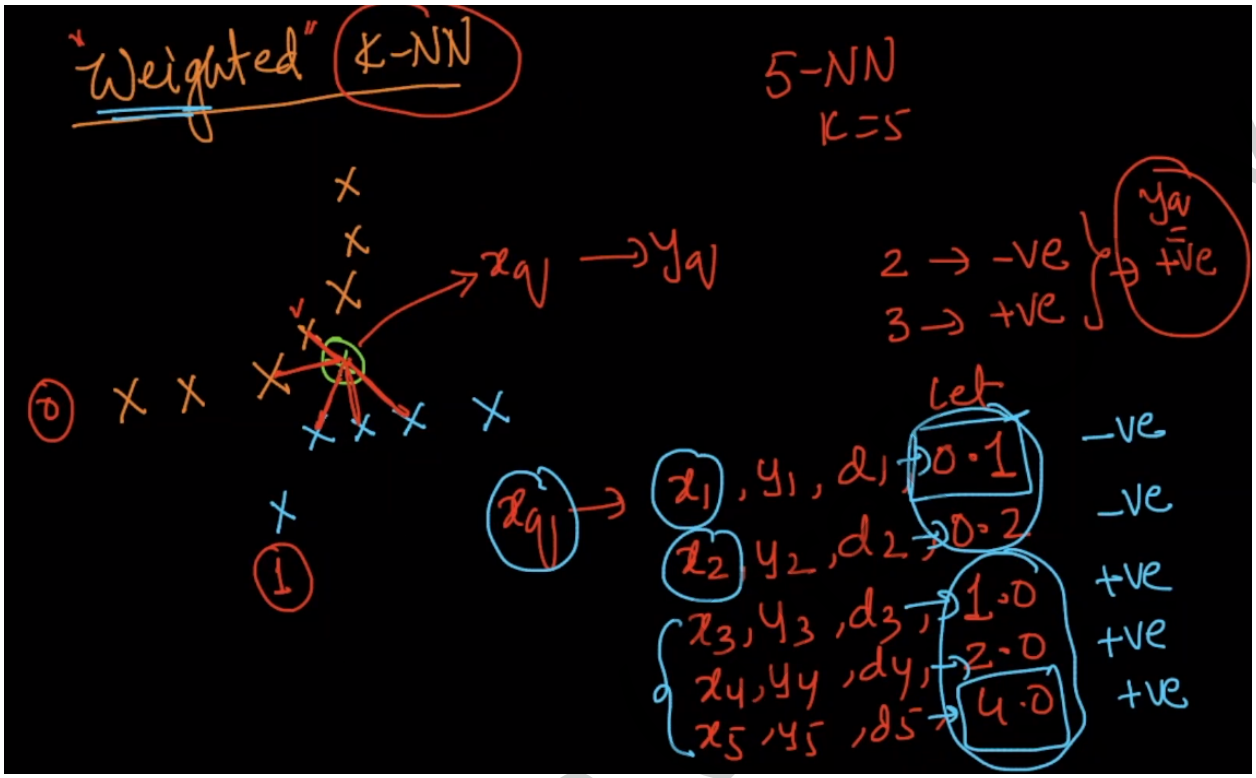
Procedure of K-NN for Regression

1) Given '$\mathbf{x_q}$', find the 'K' nearest neighbors. Let them be
   $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots\ldots\ldots, (x_k, y_k)$
2) Find $\mathbf{y_q}$' from $y_1, y_2, y_3, \ldots., y_k$ using the formula
   $\mathbf{y_q' = mean(y_i)_{i=1}^k}$ **(or)** $\mathbf{y_q' = median(y_i)_{i=1}^k}$
   (The median is less prone to the outliers when compared to the mean)

Note: In classification using K-NN, we go with the majority vote, whereas in regression using K-NN, we go with either the mean or the median of the output values of the 'K' nearest neighbors.

It is recommended to go with the median, because median is less prone to the outliers/noise when compared to the mean.

29.19 Weighted K-NN



So far we have seen simple K-NN. Let us now look at the weighted K-NN. Let us assure we are working on a 5-NN problem. So let us assume we have to predict the class label for 'x_q'.

Let 'd_1', 'd_2', 'd_3', 'd_4' and 'd_5' be the distances of 'x_q' from the 5 nearest neighbors 'x_1', 'x_2', 'x_3', 'x_4' and 'x_5' respectively. Let 'x_1' and 'x_2' be the negative points and 'x_3', 'x_4' and 'x_5' be the positive points. Let us assume

Distance between 'x_1' and 'x_q' = d_1 = 0.1
Distance between 'x_2' and 'x_q' = d_2 = 0.2
Distance between 'x_3' and 'x_q' = d_3 = 1
Distance between 'x_4' and 'x_q' = d_4 = 2
Distance between 'x_5' and 'x_q' = d_5 = 3

So now we need to calculate the weights. Let us consider the weighted function for the a point 'x_i' as $\mathbf{w_i = 1/d_i}$

As the points 'x_1' and 'x_2' are the negative points, $w_1 + w_2 = (1/0.1) + (1/0.2) = 10 + 5 = 15$

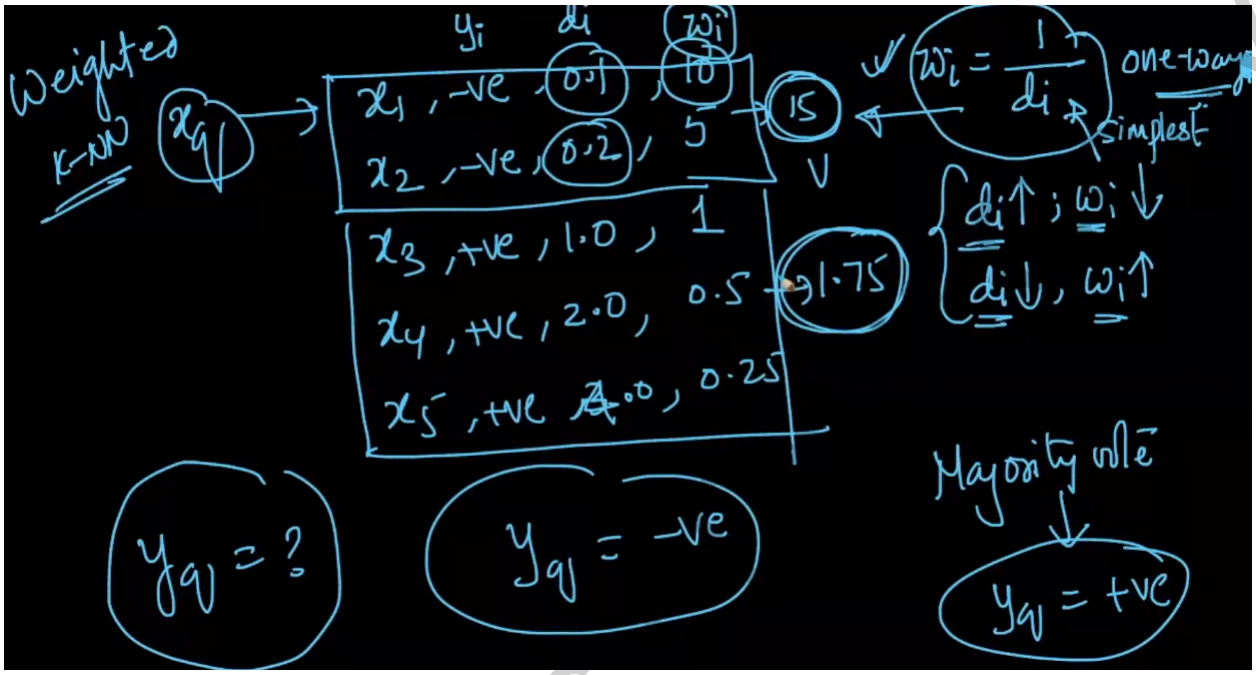
As the points 'x_3', 'x_4' and 'x_5' are the positive points, $w_1 + w_2 + w_3 = (1/1) + (1/2) + (1/3) = 1 + 0.5 + 0.33 = 1.83$

So in simple K-NN, we just go blindly with majority voting. So 'x_q' will be classified as 'positive' (as the majority count is 'positive')

In the weights K-NN, we go with the weights of the classes. So

Total Weight of the 'negative' class = $w_1 + w_2 = 15$
Total Weight of the 'positive' class = $w_3 + w_4 + w4 = 1.83$

So in the neighborhood of 'x_q', the total weight of the 'negative' class points is more than the total weight of the 'positive' class points. Hence the point 'x_q' is classified as 'negative' by weighted K-NN.



Q) When should we choose Weighted K-NN over Simple K-NN?

Ans) Weighted K-NN is used when we want to prefer closer points over farther points among the 'K' neighbors, in our decision making.

Whenever there is a requirement such that the nearest points among the nearest neighbors should contribute more in predicting the class label, then we have to go for Weighted K-NN.

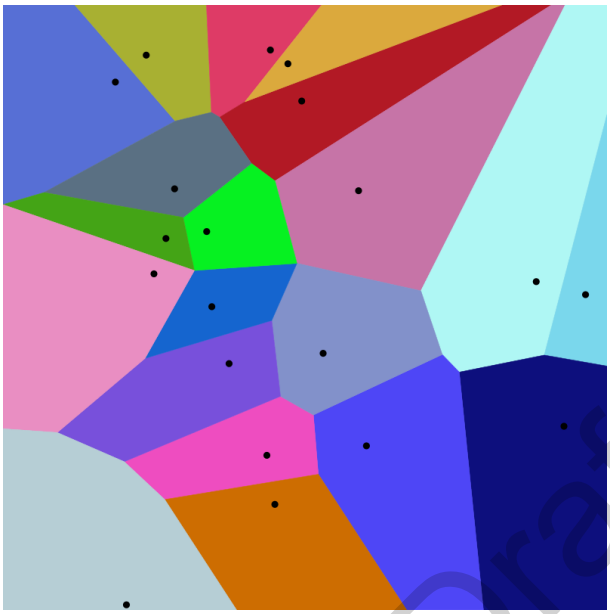
The main disadvantage with Weighted K-NN is we give too much importance to the nearby points, which could result in errors in some instances. Weighted K-NN is problem specific.

Effect of Outliers on Weighted K-NN

Even the weighted K-NN gets easily affected by the presence of the outliers. Hence it is recommended to remove the outliers before applying any model.

Along with the outliers, the scale of the features also easily affects the Weighted KNN model. So we should also make sure all the features are scaled properly, before applying any model.

20.20 Voronoi Diagram



Voronoi diagram is a partitioning of a plane into the regions based on distance to the points in a specific subset of the plane. For each seed(point), there is a corresponding region and throughout that region, if any new point falls, that seed will be the nearest point to the newly arrived point.

In 1-NN, each point has only one region, These regions are called Voronoi cells.

Note: Mostly Voronoi diagrams are not used in practice as they become very complex in high dimensional space. Voronoi diagrams are mostly used in theoretical evaluation of K-NN and in a sub-area of computer science called Computational Geometry.

The partitions dividing two regions are equidistant from the points. All the vertices are equidistant from the points. The Voronoi diagram is related to K-NN with K=1. Just like Logistic Regression has a hyperplane as the decision surface, the decision surface for 1-NN is a voronoi diagram. But in ML, voronoi diagrams aren't much used in practice.

29.21 Binary Search Tree

In simple K-NN implementation,

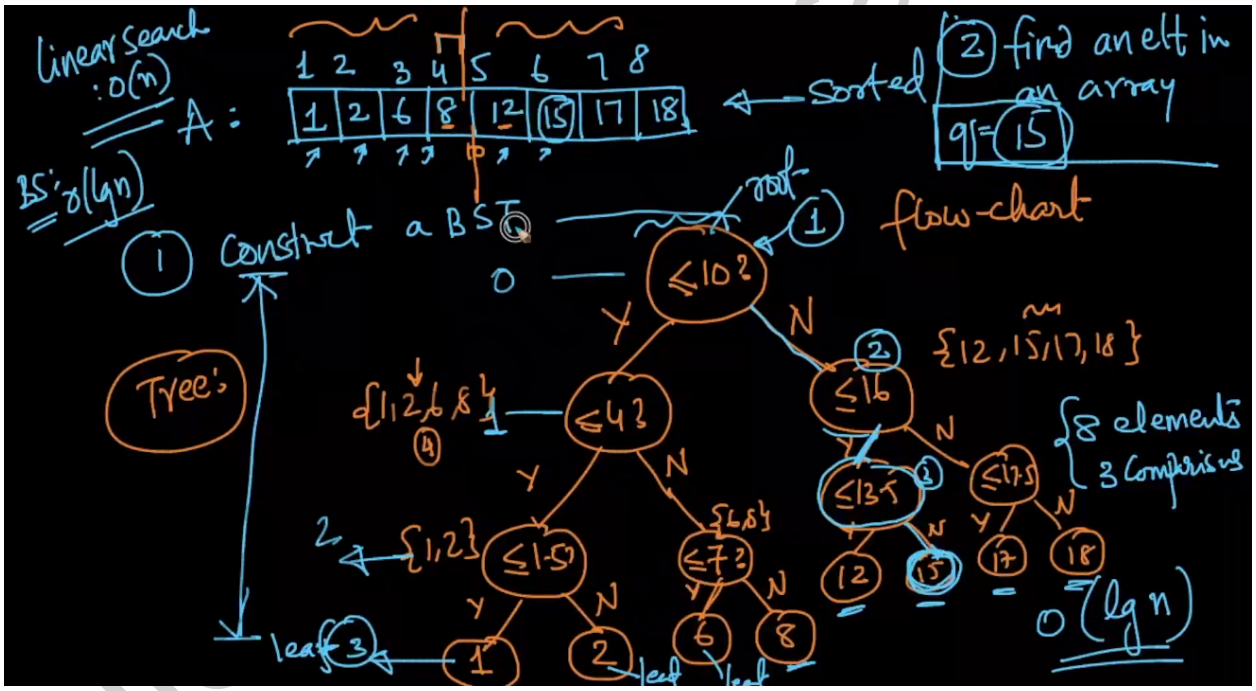
Time Complexity = O(nd) ~ O(n) (if both 'K' and 'd' are small)

Space Complexity = O(nd) ~ O(n) (if 'd' is small)

We could not reduce the space complexity of KNN, but can reduce the time complexity using a technique called KD-Tree. The time complexity reduces from O(n) to O(log(n)). For example, if there are 1024 computations to be performed, then the time complexity of simple KNN would be O(1024) whereas in case of a KD-Tree, the time complexity reduces to O(log(n)) = O(log(1024)) = O(10)

KD-Tree works similar to the way how a Binary Search Tree (BST) Works. Let us now look at how a Binary Search Tree works.

Construction and Working of a Binary Search Tree



Steps of Construction

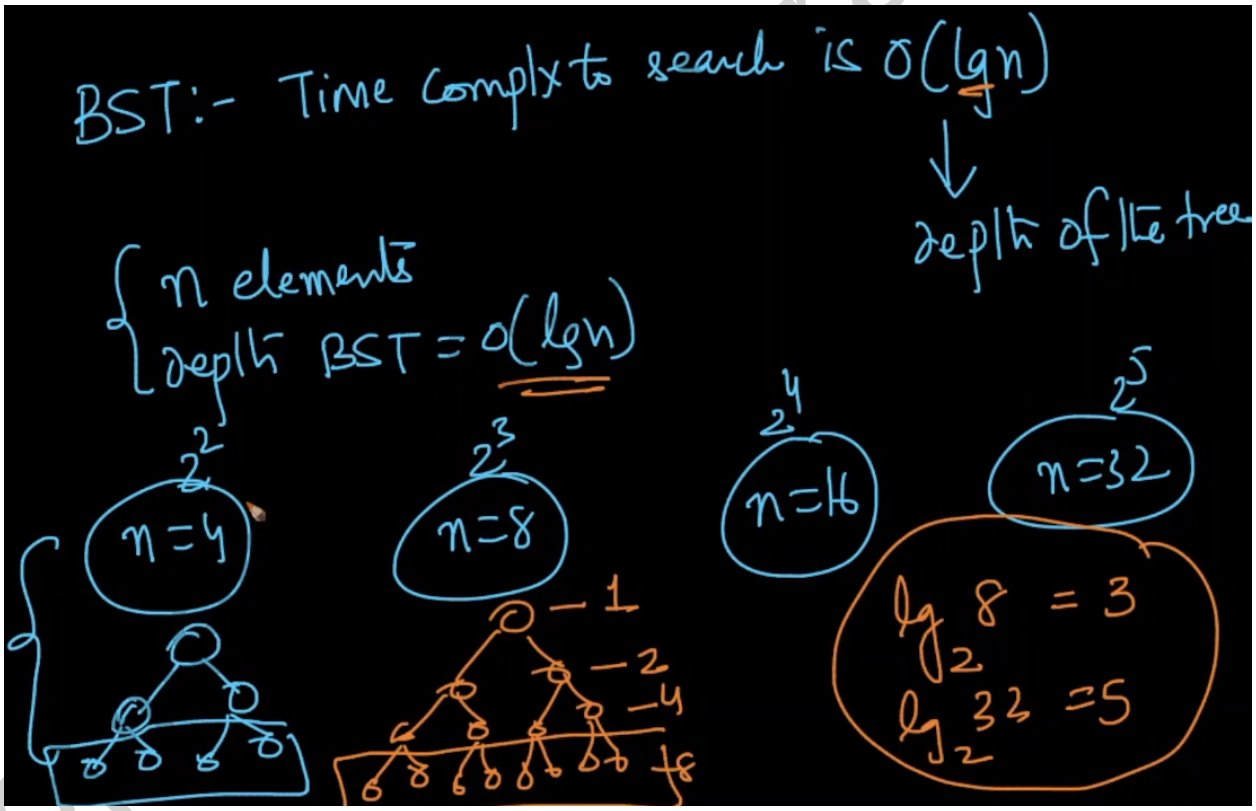
1) We have to first sort the given array/list of values in ascending order.
2) Find out the median of the values and make it a root.
3) All the values that are less than the median should go to the left side(ie., left subtree) of the root, and all the values that are greater than the median should go to the right side(ie., right subtree) of the root.

4) Repeat steps 2 and 3, until we are left with a single value in each node. All these nodes present in the bottom layer without any child nodes, are called the leaf nodes.

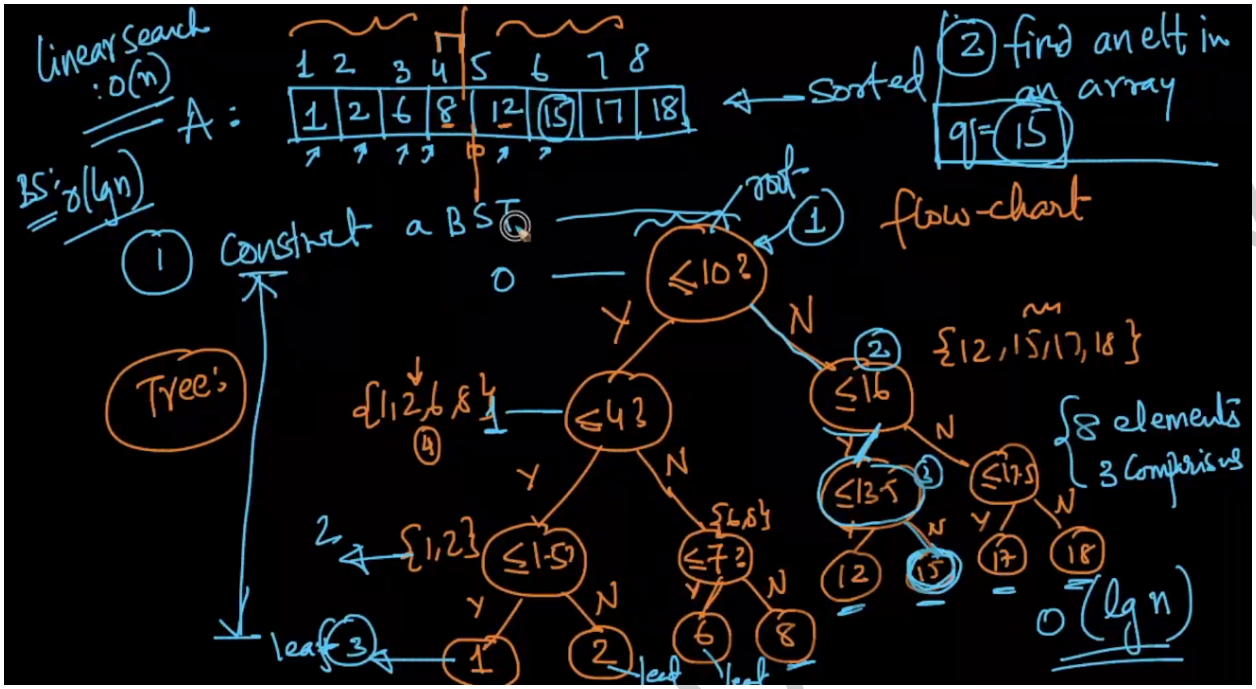
Steps for Searching an element

1) If an element is given, we have to compare it with the value in the root node first.
2) If the element is the same as the value in the root node, return True. If the element is less than the root node value, then move into the left subtree. Otherwise move into the right subtree.
3) Repeat steps 1 and 2, either until the given element is found or all the leaf nodes in the traversed path are also checked. If the given element is not found, return False.

Here as we are first sorting the values and then checking only a portion of the tree, rather than the entire tree. As we are just checking only a half portion in every subtree, the time complexity reduces from O(n) to O(log(n).



Let us check if the numbers 2, 5, 6, 16, 19 are present in the Binary Search Tree.



Checking for Number 2

a) The root value is 10. Comparing '2' with '10'. As 2<10, we have to check the left subtree, with '4' as the root.
b) The root value is now 4. Comparing '2' with '4'. As 2<4, we have to check the left subtree, with '1.5' as the root.
c) Now the root value is 1.5. Comparing '2' with '1.5'. As 2>1.5, we have to check the right subtree, with '2' as the root.
d) Now the root value is 2. As 2==2 returns **True**, the whole result turns out to be **True**. It means the specified element '2' is present in the Binary Search Tree.

Checking for Number 5

a) The root value is 10. Comparing '5' with '10'. As 5<10, we have to check the left subtree, with '4' as the root.
b) The root value is now 4. Comparing '5' with '4'. As 5>4, we have to check the right subtree, with '7' as the root.
c) The root value is now 7. Comparing '5' with '7'. As 5<7, we have to check the left subtree, with '6' as the root.
d) The root value is now 6. Comparing '5' with '6'. As 5<6, we have to check the left subtree. But here '6' is a leaf node and it has no more subtrees/branches. So as we couldn't find the element '5', it returns **False**.

Checking for Number 6

a) The root value is 10. Comparing ‘6’ with ‘10’. As 6<10, we have to check the left subtree, with ‘4’ as the root.
b) The root value is now 4. Comparing ‘6’ with ‘4’. As 6>4, we have to check the right subtree, with ‘7’ as the root.
c) The root value is now 7. Comparing ‘6’ with ‘7’. As 6<7, we have to check the left subtree, with ‘6’ as the root.
d) Now the root value is 6. As 6==6 returns **True**, the whole result turns out to be **True**. It means the specified element ‘6’ is present in the Binary Search Tree.

Checking for Number 16

a) The root value is 10. Comparing ‘16’ with ‘10’. As 16>10, we have to check the right subtree, with ‘16’ as the root.
b) Now the root value is 16. As 16==16 then we move to the left subtree of node 16. As the root node value of the left subtree is 13.5 which is less than 16. So we return **False.**

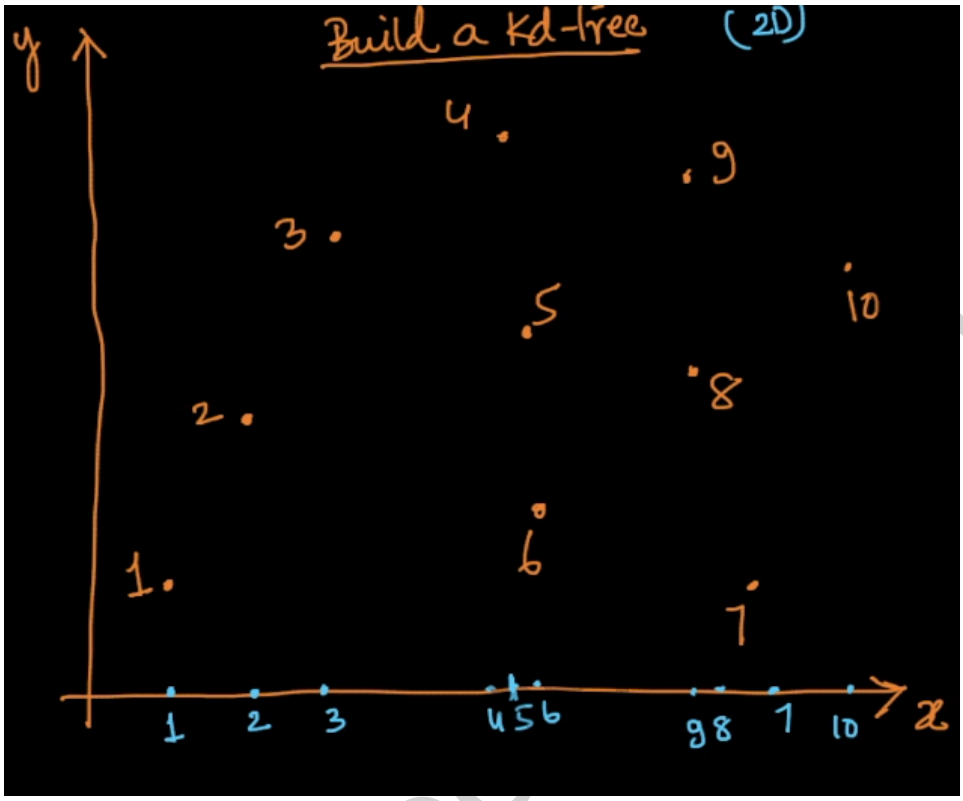
Checking for Number 19

a) The root value is 10. Comparing ‘19’ with ‘10’. As 19>10, we have to check the right subtree, with ‘16’ as the root.
b) The root value is now 16. Comparing ‘19’ with ‘16’. As 19>16, we have to check the right subtree, with ‘17.5’ as the root.
c) The root value is now 17.5. Comparing ‘19’ with ‘17.5’. As 19>17.5, we have to check the right subtree, with ‘18’ as the root. But here ‘18’ is a leaf node and it has no more subtrees/branches. So as we couldn’t find the element ‘19’, it returns **False**.

29.22 How to build a KD-Tree

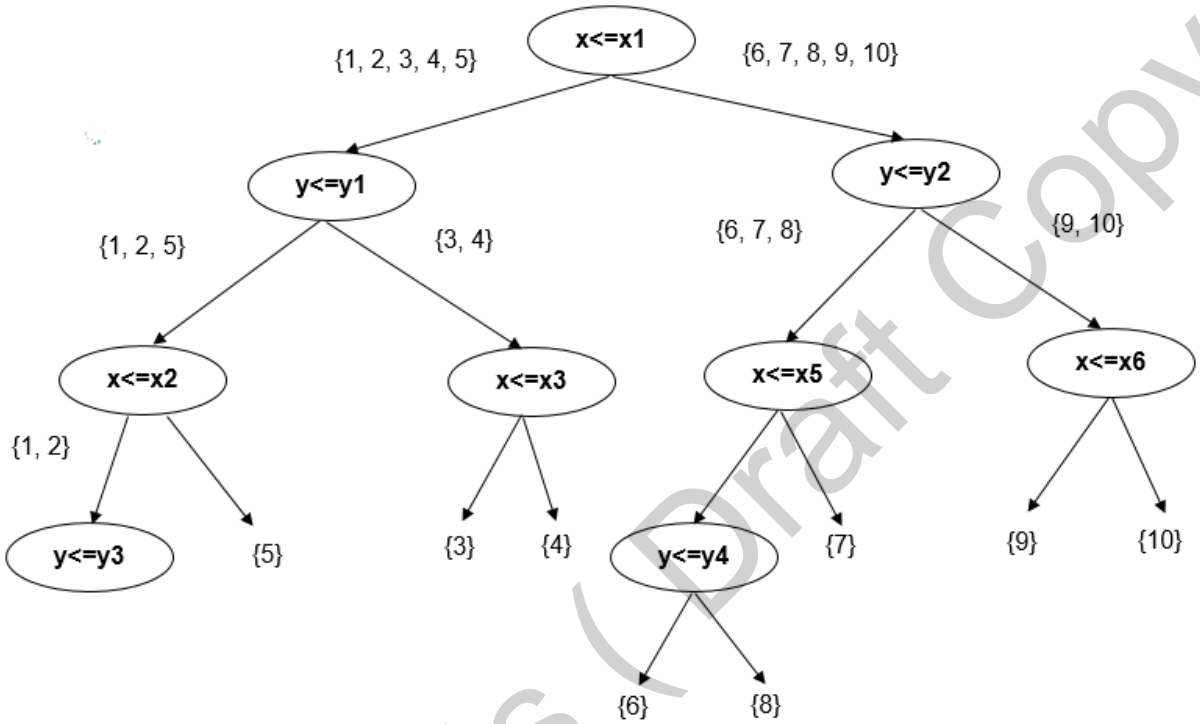
Procedure to construct a KD-Tree

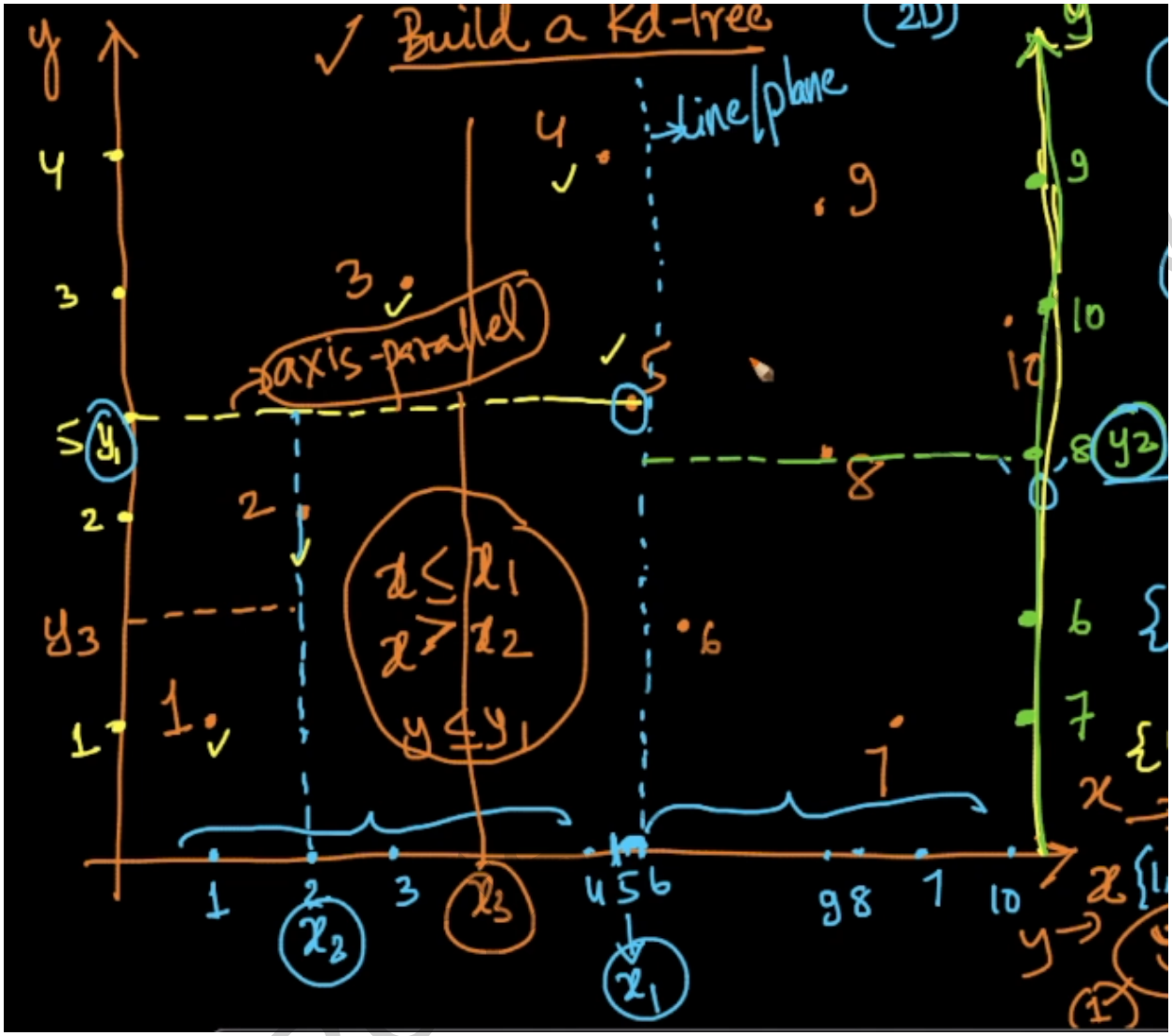
Let the given points be projected in a 2D space as shown below



1) Pick the 'X' axis, project all the points on the 'X' axis. Compute the median. Split the data on the basis of the median 'x_1'.
   All the values that are less than or equal to 'x_1' should go into the left subtree(ie., 1,2,3,4,5), and those values which are greater than 'x_1'(ie., 6,7,8,9,10) should go into the right subtree. 'x_1' would be the root of the tree.
2) Pick the 'Y' axis, project all these points on the 'Y' axis. Compute the median. Split the data on the basis of the median 'y_1'. Now 'y_1' is the root node of the left subtree.
   All the values that are less than or equal to 'y_1'(ie.,1,2,5) should go into the left subtree, and those values which are greater than 'y_1'(ie., 3,4) should go into the right subtree.
   Compute the median of the values {6,7,8,9,10} and it should be the root node of the right subtree. Let it be 'y_2'. All the values that are less than or equal to 'y_2' will now go into the left subtree and the remaining values should now go into the right subtree.

This way we have to keep alternating between the axes. The final KD-Tree looks like below



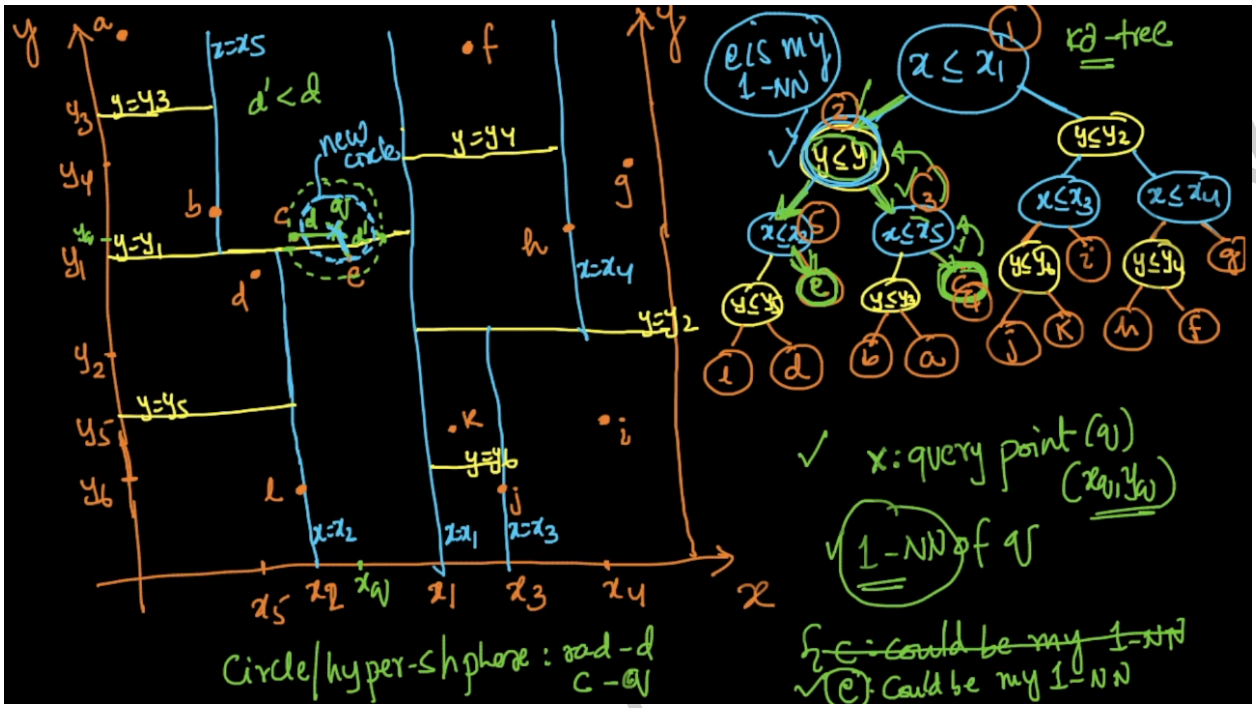


Note:

In KD-Tree, we are breaking the space using axis-parallel lines/planes into rectangles(in 2-D)/ cuboids(in 3-D)/ hyper cuboids (in n-D).

We have to switch the axes alternatively till we reach the leaf nodes. We can build the KD-Tree not only with the 'median' statistic, but also with the 'mean' statistic.' But if we go with the 'mean' statistic, we should make sure, there are no outliers in our data.

29.23 Finding Nearest Neighbors using KD-Tree



Let us assume we have to find out only the 1 nearest neighbor to a given point (x_q, y_q). In the same way as we did in the previous topic, here instead of using 'x', we have to use 'x_q' and instead of using 'y', we have to use 'y_q'.

We have to keep traversing the tree in the direction of the results of the conditions at the nodes, until we reach a leaf node. Here the first lead node value we reach is 'c'. So this tree helps us in finding in which cuboid our point (x_q, y_q) lies in. So we can say that 'c' is the 1st nearest neighbor, but we are still not sure.

We shall calculate the distance between 'c' and the point (x_q, y_q). Let the distance be 'd'. As 'c' is on the same side as (x_q, y_q), we try to draw a hyperplane with centre at (x_q, y_q) and radius 'D'.

If we find any point inside this hypersphere, then we can say that point is the 1st nearest neighbor of (x_q, y_q). We looked at only one side of the line $y=y_1$, as 'c' and (x_q, y_q) are on the same side. We shall now look at the other side of $y=y_1$, to find out the nearest neighbor for (x_q, y_q). For that we need to backtrack till we reach y<=y1, and then as we have already traversed in the right subtree, we then have to traverse in the left subtree, according to the results of condition checking and reach the left node 'e'. Now we can say 'e' also could be our nearest neighbor.

Let us now calculate the distance between (x_q, y_q) and 'e' and let this distance be D'.

If D<D', we can ignore 'e' and say that 'c' is the 1st nearest neighbor.
If D'<D, we can ignore 'c' and say that 'e' is the 1st nearest neighbor.

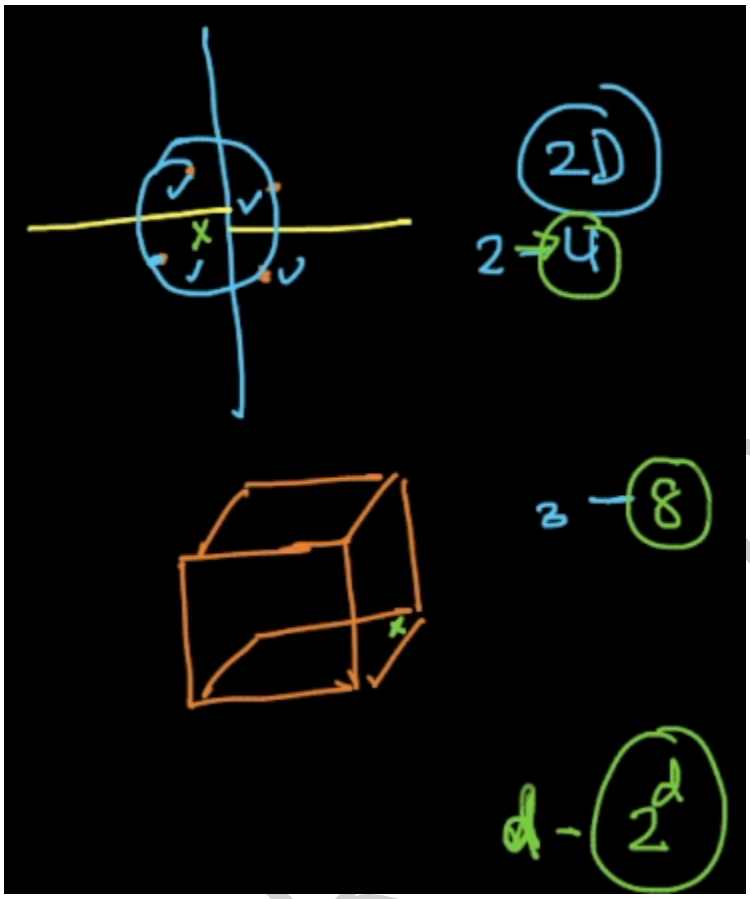
From the figure, it is clear that D'<D, so we can ignore 'c' and declare 'e' as the 1st nearest neighbor.
But again we have to draw a circle/sphere with the centre at (x_q, y_q) and radius D'. Now we'll check if the new circle intersects any other lines other than $y=y_1$. As we don't find any, we can confirm that 'e' is the 1st nearest neighbor.

Note: Number of Comparisons while searching for an element in KD-Tree is
Best Case: O(log(n))
Worst Case: O(n)

It means if there are no intersections of the planes, then the number of comparisons would be O(log(n)). In the presence of intersections, then the number of comparisons would increase.

29.24 Limitations of KD-Tree

So far we have seen that if 'd' is small,
Best Case Time Complexity for KNN = O(k*log(n))
Worst Case Time Complexity for KNN = O(k*n)



So far, we have seen when our data consists of 2-dimensions, we have to look up $2^2 = 4$ adjoining cells.

If the data is in 3-dimensional form, then the number of adjoining cells to look up $= 2^3 = 8$.

So if the data is in d-dimensional form, then the number of adjoining cells to look up $= 2^d$
For example, if the number of dimensions = 20, then
Number of adjoining cells to look up = 2^{20}
Due to this, as the 'd' value is not small, the time complexity is no more O(log(n)).
The time complexity changes to $O(2^d*\log(n))$.
If $2^d = n$, then the time complexity is O(n*log(n)).
As the number of dimensions increases, the number of adjoining cells also increases, thereby increasing the time complexity drastically. In such a case, KD-Tree is useless.

The Space Complexity of KD-Tree = O(n)
Even when the dimensionality is small, O(log(n)) time complexity holds good only if our data is uniformly distributed in space. If the data is not uniformly distributed, the time complexity changes to O(n). (same as that of brute force KNN).

Note: When compared to the other models, KNN is often used as a baseline model, when we are operating with small data. KD-Tree is not developed to find the nearest neighbor in Machine Learning, but was developed to find the nearest neighbors in the computer graphics.

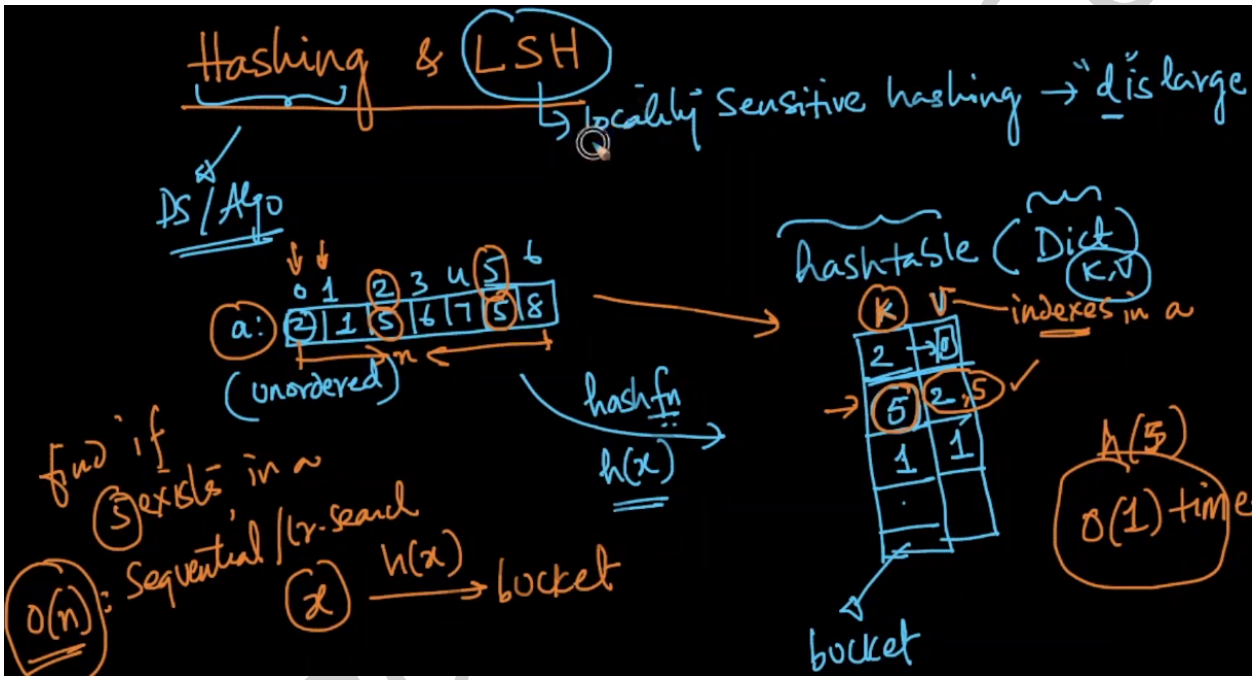
29.25 Extensions

Refer to the wikipedia article link given below, as this video lecture is purely theory based, and also the content in it was taught from the below mentioned link only.
Wikipedia Link: https://en.wikipedia.org/wiki/K-d_tree#See_also
For any queries regarding this topic, please feel free to post them in the comments section below the video lecture.

29.26 Hashing vs LSH(Locality Sensitive Hashing)

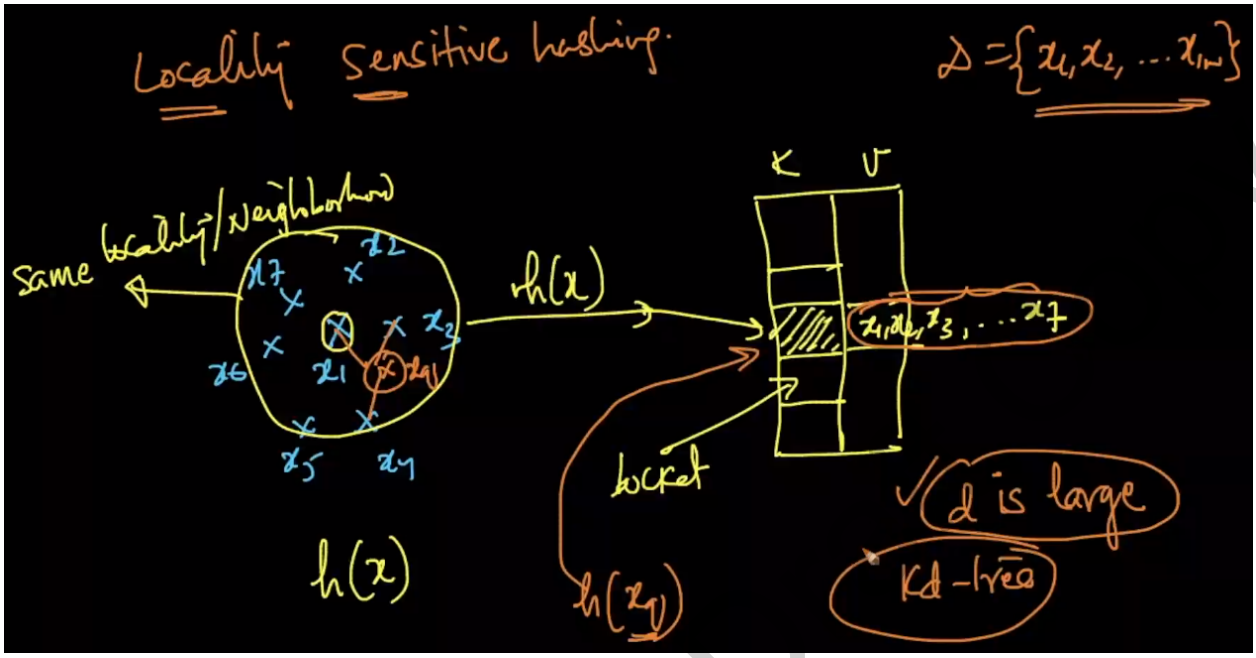
Locality Sensitive Hashing is a powerful technique when compared to KD-Tree and also works well when the dimensionality 'd' is large.

Before we look into LSH, we shall learn about Hashing. In general data structures, we have a data structure called Hashtable. The Hash function is the core concept on which a Hashtable works. One Hashtable will have only one hash function. There are no chances for one hash table to have multiple hash functions at a time. In python, Dictionary is the same as a hashtable.



The data in a hash table is stored in key-value pair format. If a value is given along with a key, then we can directly store that key-value pair in the hashtable. If a value is given without a key, then that value is passed through the hash function. The output of the hash function for a given point is the key, and each key is associated with a bucket in the hashtable. After we get the 'key' for a given value from the output of the hash function, then that particular value is stored in the bucket associated with the obtained key. So whenever we want to retrieve the value from the bucket, if we already know the 'key', then we can directly access it. Otherwise, we again have to pass the same value through the same hash function, we get the same key and then we can access it with that key. All the keys in a hashtable are unique, there is no duplication of keys.

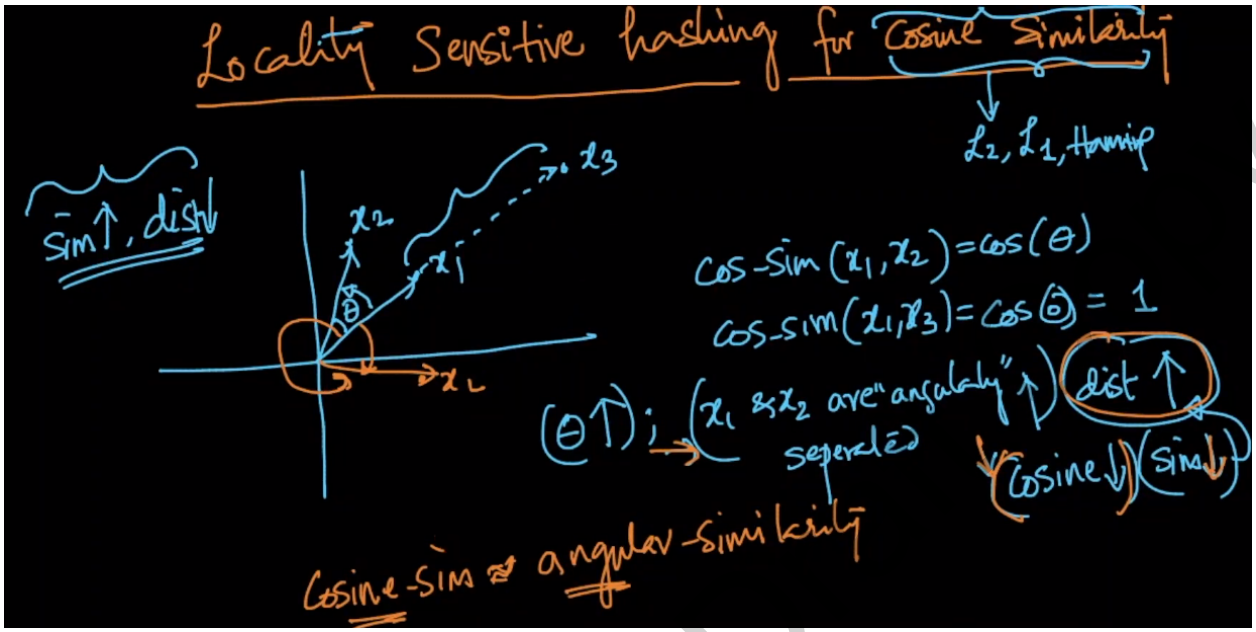
Let us now look at the functioning of Locality Sensitive Hashing. Let h(x) be the hash function that is generated.



All the points here are in the neighborhood. For any given point 'x_i', in such a way that all the points that are nearby go to the same bucket in the hashtable. For the point 'x_i', first it has to go through the hash function, and a hash value is generated. This hash value will be the 'key' and 'x_i' will be the value. These two are going to be stored in the key-value pair format.

So here, all the points that lie in a neighborhood are stored in the cell associated with the same bucket. So we can easily pull out the 'n' nearest neighbors using LSH, as all these values are stored in the same cell.

29.27 LSH for Cosine Similarity



We have 3 vectors 'x_1', 'x_2' and 'x_3'.The vectors 'x_1' and 'x_3' are in the same direction and the angle between them is 0^0. Whereas the vectors 'x_1' and 'x_2' are separated by an angle 'θ'. So we can say the vectors 'x_1' and 'x_2' are angularly separated. If the angle 'θ' increases, the cosine similarity decreases, and the cosine distance increases.

cosine_similarity(x_1, x_2) = cosθ

cosine_similarity(x_1, x_3) = cos(0^0) = 1

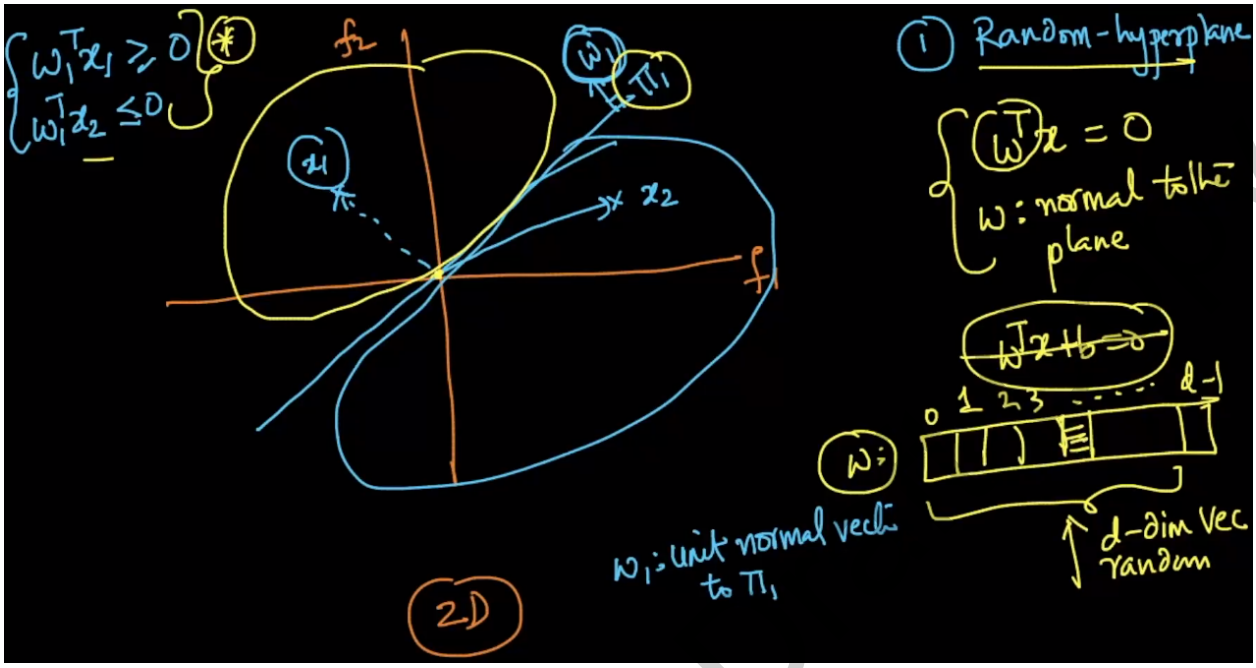
As 'θ' increases, 'x_1' and 'x_2' are more angularly separated. Cosine Similarity is all about angular similarity, but not the geometrical distance. So in LSH, all the points that are more similar/close should go to the same bucket.

So if 'x_1' and 'x_2' are similar, then the hash function associated with these two points will become the key.

Key = h(x_1) = h(x_2)

LSH is a randomized algorithm. LSH says it always doesn't give the correct answer. But it always says it will give the correct answer with high probability. Every time LSH might not yield the nearest neighbors accurately, but everytime whatever results are obtained, are given with high probabilities.

Let us consider a 2-D plane and divide it into two parts using a line/plane/hyperplane(here it is a random plane), as shown below.

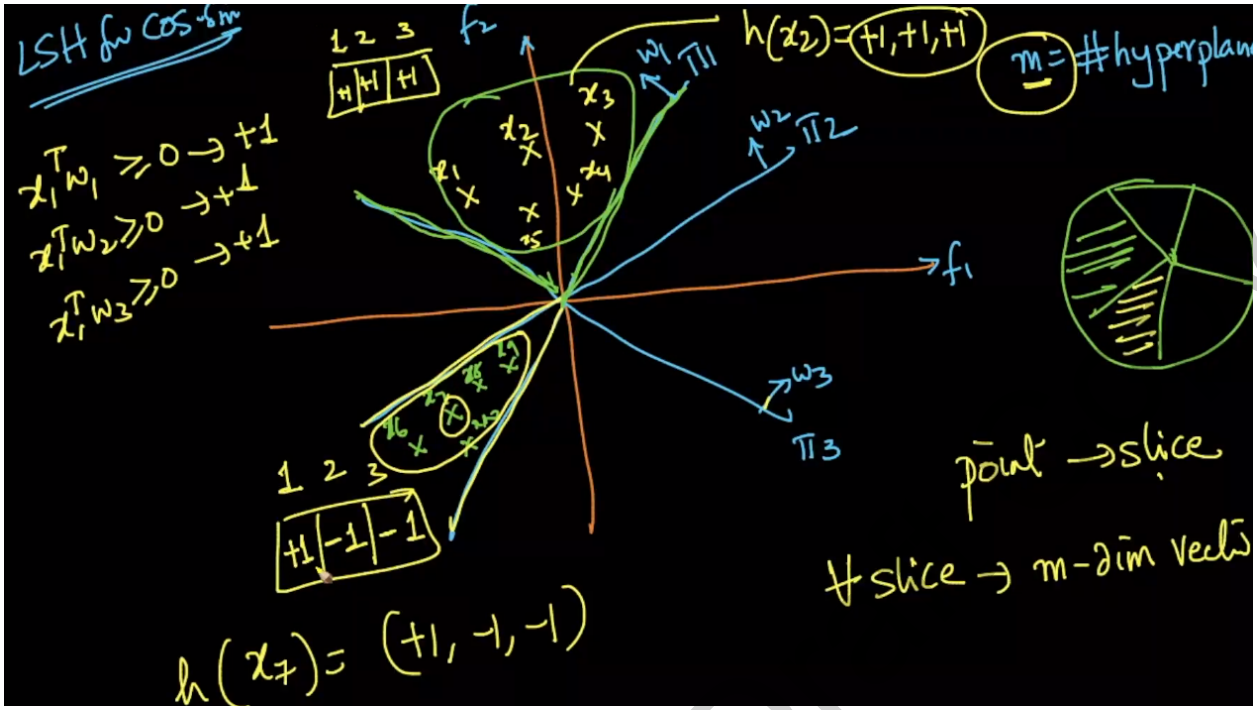


This plane is obtained by randomly generating the values for the unit normal vector 'w_1'. Since 'x_1' is present on the same side as 'w_1', $\mathbf{w_1^T x_1 >= 0}$.

But the point 'x_2' is present on the opposite side of the vector 'w_1'. So $\mathbf{w_1^T x_2 <= 0}$. Here 'w_1' is a d-dimensional random vector. If we are able to determine 'w_1', then we can easily find out the equation of the hyperplane 'π_1'.

If a hyperplane passes through the origin, then the equation of the hyperplane is given as $w^T.x = 0$. If the hyperplane doesn't pass through the origin, then the equation of the hyperplane is $w^T.x+b=0$. (Here 'w' is a unit vector normal to the hyperplane and 'b' is the intercept)

In 'w_1', every component is a random number sampled from Normal distribution. With the help of 'w_1', we can generate a random number hyperplane. Let us take a random hyperplane and understand how it works.



Let us consider 3 random hyperplanes and perform Locality Sensitive Hashing using Cosine Similarity.

If we multiply any point 'x_i' in slice 1 with 'w_1', then $w_1^T.x_i>0$.

So we shall compute the hash key for the points in slice 1.

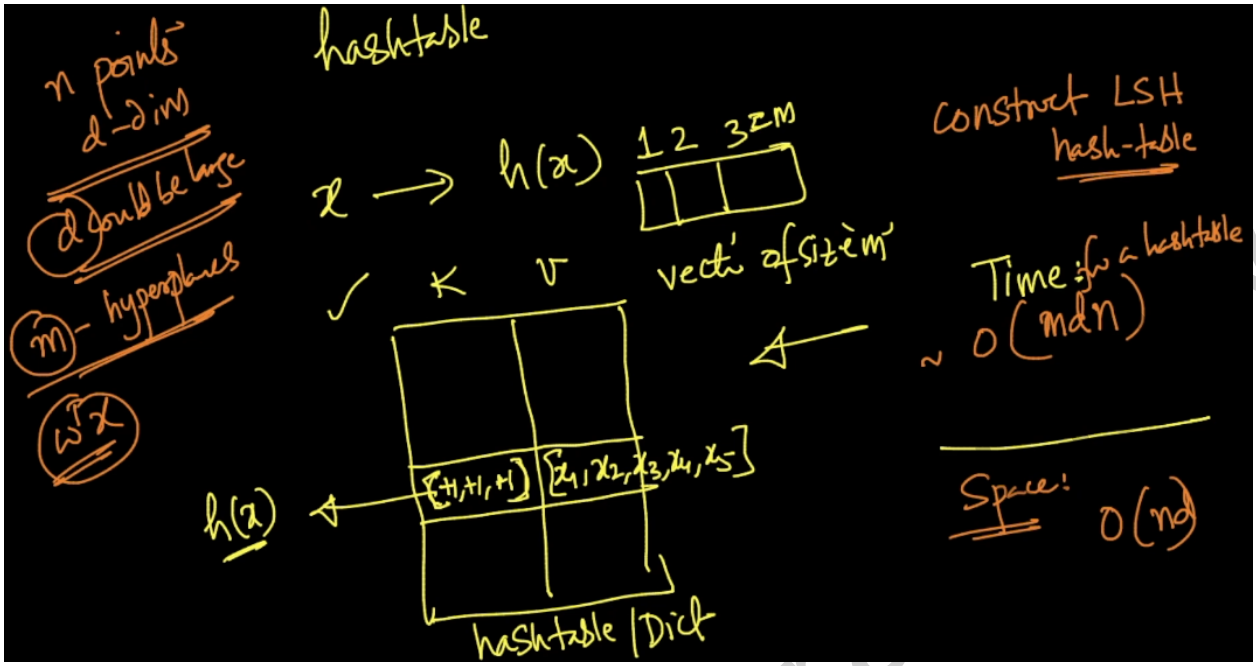
hash(x_1) = {+1, +1, +1} (These 3 values denote the signs of $w_1^T.x_1$, $w_2^T.x_1$ and $w_3^T.x_1$ respectively)

hash(x_2) = hash(x_3) = hash(x_4) = hash(x_5) = hash(x_6) = {+1, +1, +1}

As we got the keys using the hash function, we now have to construct the hash table.

Key (Hash Function Value)	Value
. . . {+1, +1, +1} . . .	. . . [x_1, x_2, x_3, x_4, x_5, x_6] . . .

Here we are taking the vectors as the key, and the points with these vectors, as the values. We have 'n' data points and each data point is 'd' dimensional.

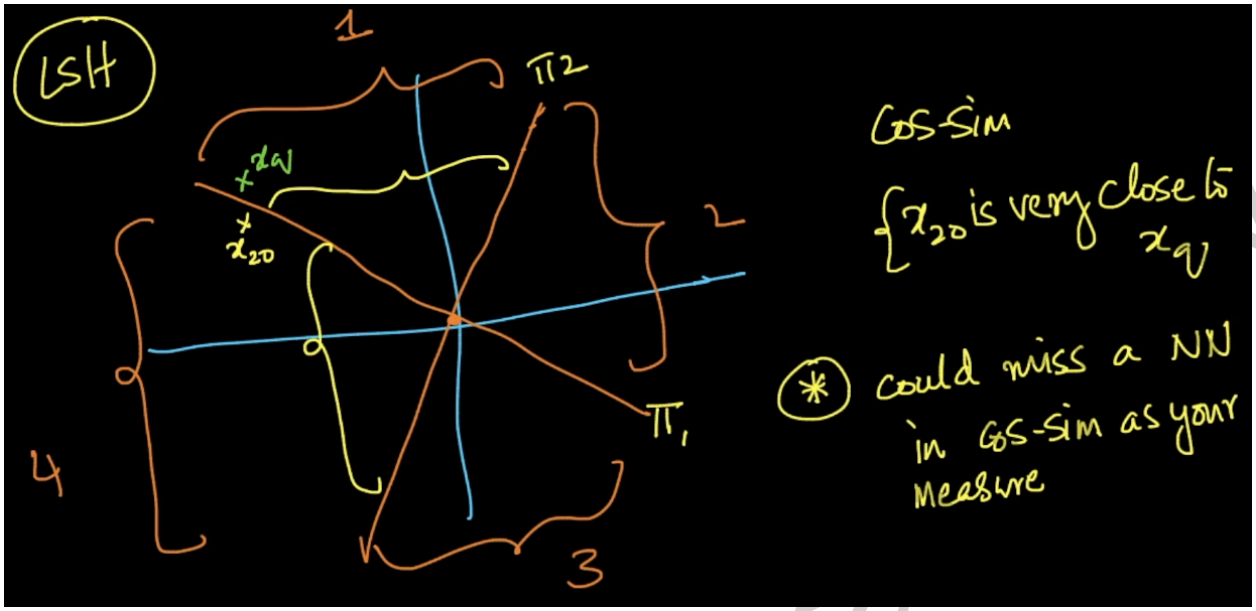


So space complexity → O(nd) (To compute a hashtable)
Let us assume we have 'm' hyperplanes.
Time Complexity to compute one key-value pair = O(m*d)
Time Complexity to compute all the key-value pairs = O(m*d*n)

So in order to find out the nearest neighbors for the point 'x_q', we need to compute the hash function for the point 'x_q', and pull out the value associated with this hash function. In case if we want 'K' nearest neighbors, then after pulling out the list of the values from the hashtable, we have to compute the cosine similarity scores of 'x_q' with all the points extracted from the hashtable. Those 'K' points with the top Cosine Similarity scores are chosen.

Time Complexity for querying a point = O(m*d)
In case, if there are n' elements in that bucket, then the total time complexity = O(md + n'd)
The ideal value for 'm' = log(n)
So the time complexity = O(d*log(n))

But here we come across a special case. So far, we have seen the points lying in the same bucket. But let us look at the figure below.



We have the point 'x_{20}' in a different bucket/slice when compared to 'x_q'. So here even if 'x_{20}' is a nearest neighbor of the query point 'x_q', it will not be considered in the same bucket. So we would definitely miss this point, if we use cosine distance as the metric.

In order to get such points under the same bucket, we need to

a) Take 'm' hyperplanes and create a hash function. Let it be $h_1(x_q)$.
b) Take another 'm' hyperplanes and create another hash function. Let it be $h_2(x_q)$.
In this way, we have to keep doing it. At one hash function value, we get both these points in the same bucket.

So now, for every data point 'x_i', we have to find out the hash value by passing it through all those hash functions, obtaining the nearest neighbors from every hash function, and performing a set union operation. Out of all the nearest neighbors, we have to pick our 'K' nearest neighbors, using the cosine distance.

So the total time complexity to query 'L' hash tables = O(m*d*L)

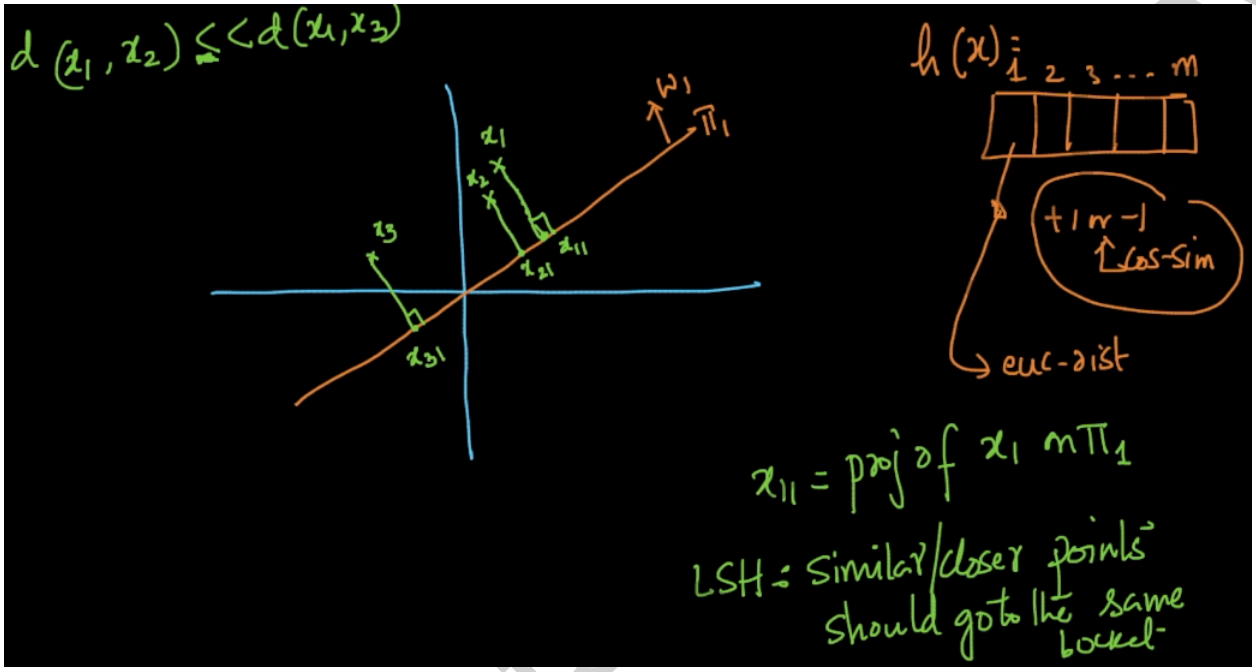
As the number of hyperplanes increases, the number of slices/buckets increase,and the number of points in each slice decreases.

As the number of hyperplanes decreases, the number of slices/buckets decreases, and the number of points in each slice increases, thereby the time complexity becomes worse.

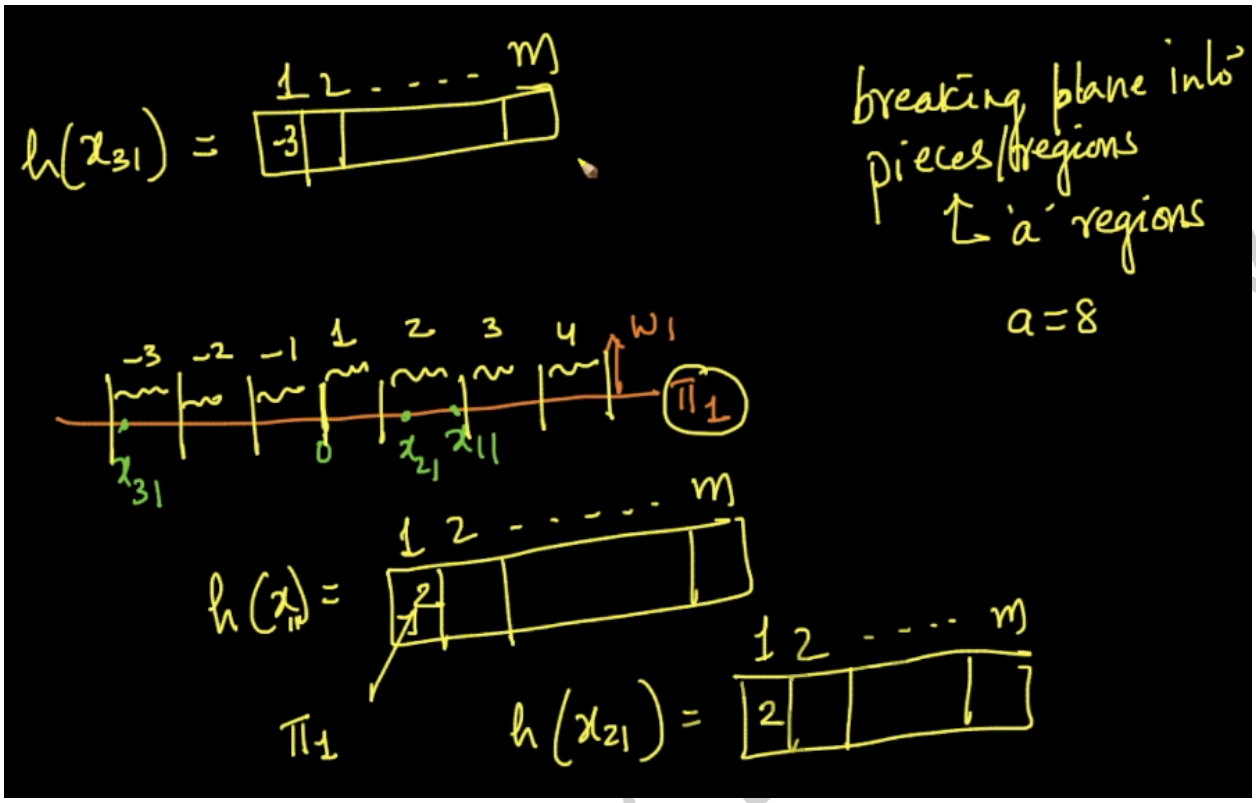
29.28 LSH for Euclidean Distance

So far we have seen that if a point 'x_i' belongs to the region same as 'w1', then we use +1, and if it is present on the opposite side, we use -1 in the hash key.

Now we shall draw the perpendiculars of all the points onto the hyperplane 'π', as shown below.



We now have to divide this hyperplane into the regions on the basis of the projections of the points on it. We shall give labels to each region as shown below.



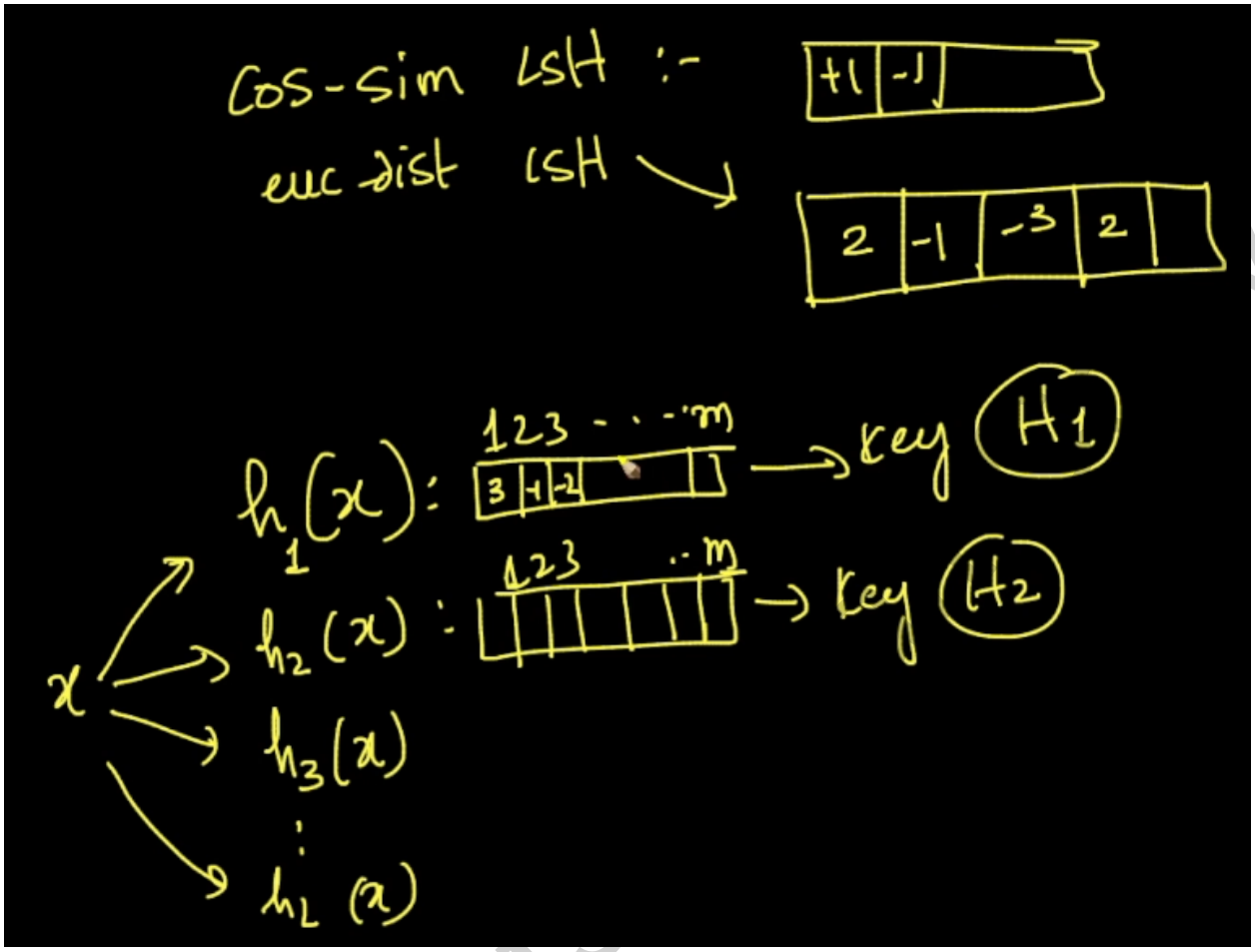
Now we have to create the hash key. If there are 'm' hyperplanes, then the hash key would be a 'm' dimensional vector.

The hash key for the point 'x_{11}' consists of '2' for the component '1'. (It is because this point is having a projection onto the hyperplane 'π_1'. So the 1st component of the 'm' dimensional vector would be filled with the value. This value here would be the region number in which this point is present). We can see the point 'x_{11}' lies in region '2' of the hyperplane 'π_1', so the first component of the hash key would be '2'.

When it comes to the hash key of 'x_{21}', even here this point lies in the 2nd region of the hyperplane 'π_1'. So the first component of the hash key should be '2'.

If we look at the hash key of the point 'x_{31}', as this point is present in the region '-3', we fill the first component of the hash key with '-3'.

This is the difference between LSH for Cosine Similarity and LSH for Cosine Distance. The hash values in LSH for Cosine Similarity contain +1 or -1, whereas in LSH for euclidean distance, the value can be anything(the region number associated with the hyperplane).



Let us assume, here is LSH for euclidean distance, we have 'L' hashtables, then for every point 'x', we have to generate 'L' hash keys, as shown above. Each of these hash keys is 'm' dimensional.

$h_1(x)$ → Hash key of the point 'x' in Hashtable 1

$h_2(x)$ → Hash key of the point 'x' in Hashtable 2

.

.

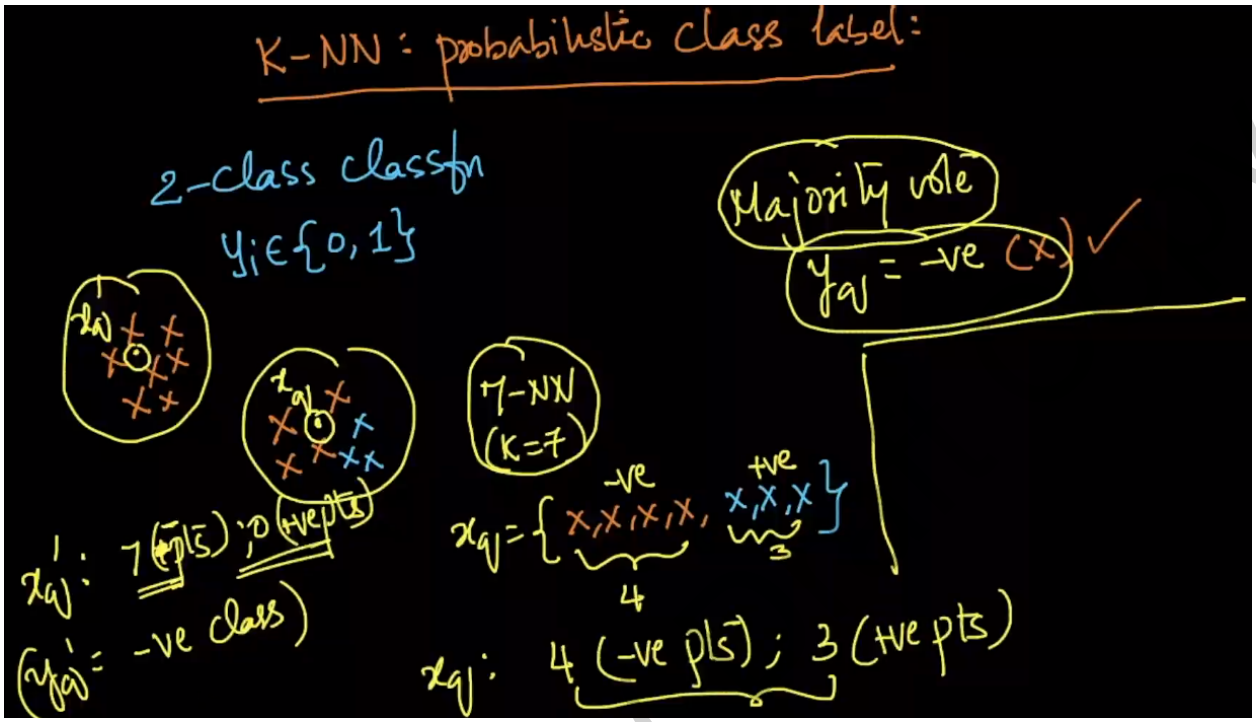
$h_L(x)$ → Hash key of the point 'x' in Hashtable L

The points that have the same region number tend to be closer. If 2 points have the same region value of a hyperplane 'π_1', then the distance between them will be closer.

But again as these hyperplanes are formed randomly, they can be formed in any direction, and due to this sometimes the points may fall in different regions.

So LSH is not perfect all the time. It is a probabilistic/randomized algorithm and is still used extensively.

29.29 Probabilistic Class Label



Let us consider a binary classification problem. (ie., yi∈{0,1}). Let us assume we have the points as shown above, and are working on a 7-NN problem. (K=7)
So the 7 nearest neighbors of 'x_q' are

x_q = {-ve, -ve, -ve, -ve, +ve, +ve, +ve}

Here if we go with the majority vote count, we get **y_q = -ve**.

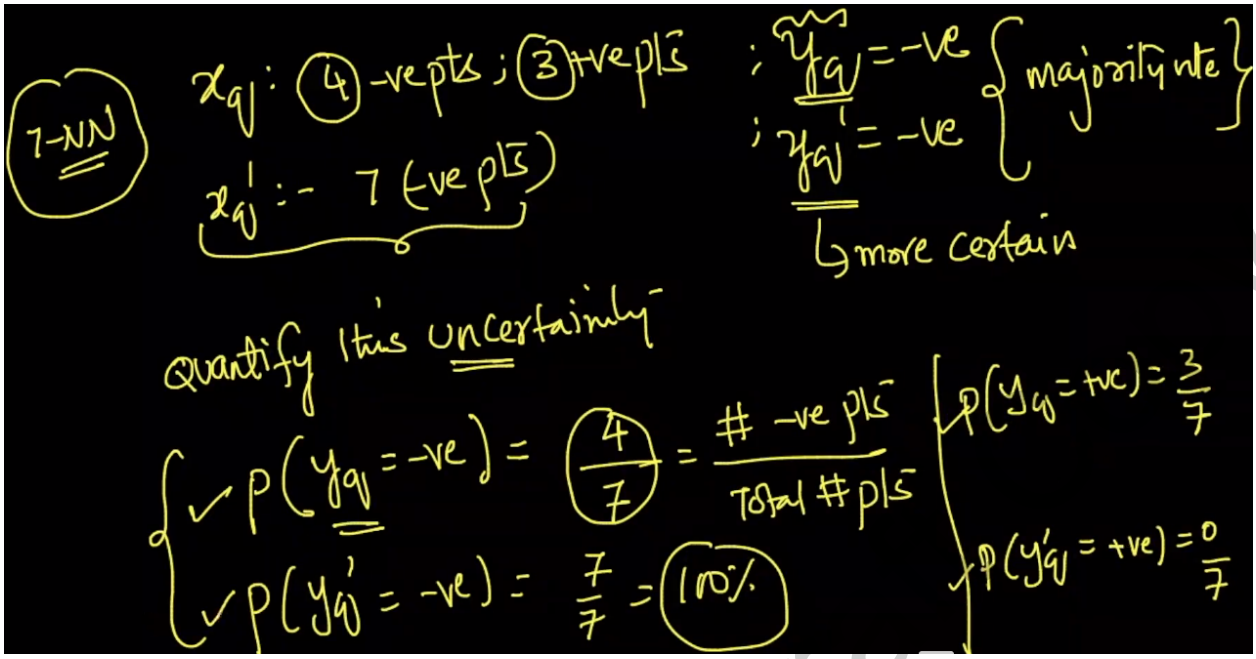
Let us assume we have another point '**x_q**'' and its 7 nearest neighbors are

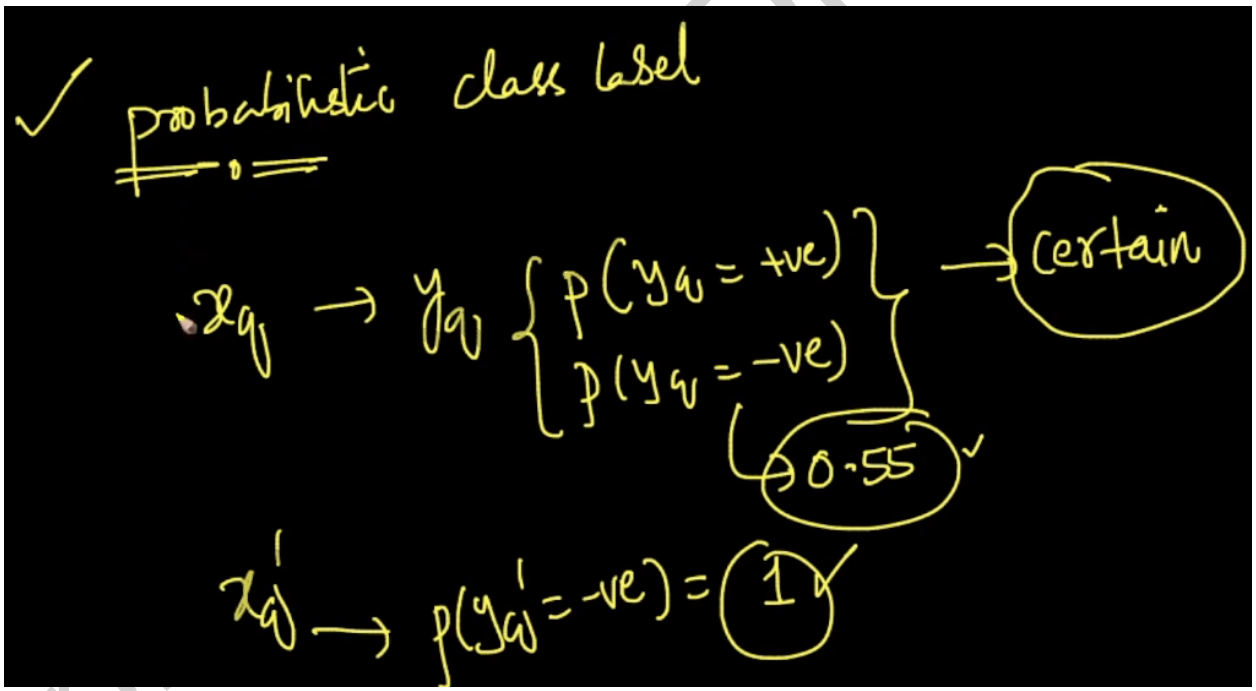
x_q' = {-ve, -ve, -ve, -ve, -ve, -ve, -ve}

Here if we go with the majority vote count, we get **y_q' = -ve**.

P(yq = -ve) = (Number of '-ve' neighbors of x_q)/(Total number of points in the neighborhood of x_q) = 4/7
P(yq = +ve) = (Number of '+ve' neighbors of x_q)/(Total number of points in the neighborhood of x_q) = 3/7

P(yq' = -ve) = (Number of '-ve' neighbors of x_q')/(Total number of points in the neighborhood of x_q') = 7/7 = 1
P(yq' = +ve) = (Number of '+ve' neighbors of x_q')/(Total number of points in the neighborhood of x_q') = 0/7 = 0





Probabilistic class labels are mostly used in classification tasks. Instead of giving deterministic class labels, we are giving probabilistic class labels. Sometimes giving probabilistic class labels is better than giving deterministic class labels.

Probabilistic result is a mathematical way to show how certain we are in predicting the class labels.

Note: We are not giving any notes for the video lectures 29.30 and 29.31 as they both are of only the code discussions.
You can download the ipython notebooks from the link given below.
https://drive.google.com/drive/folders/1tMYRWzbrSMxQ7aQ5mc8Qf4190gPSt5fy

For any queries, please feel free to post them in the comments section below the video lecture.