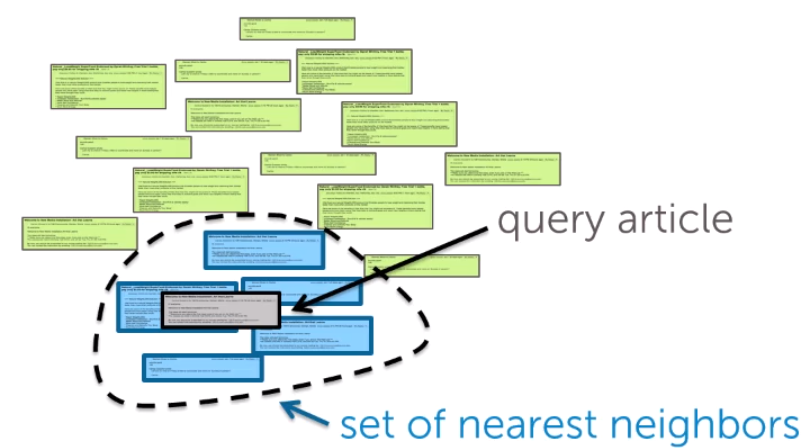
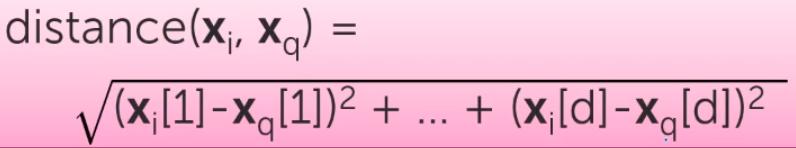
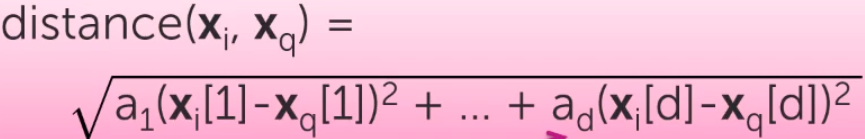
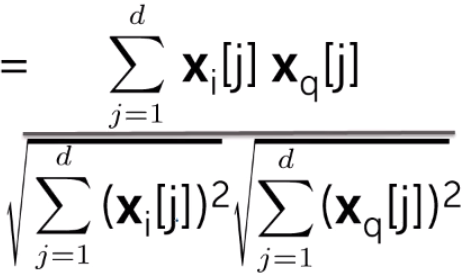
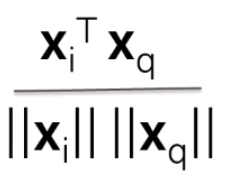
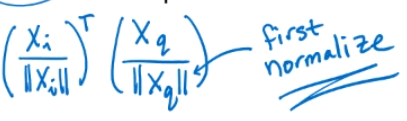
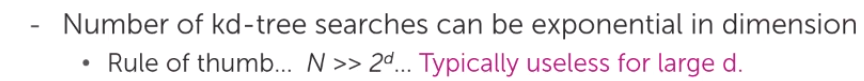
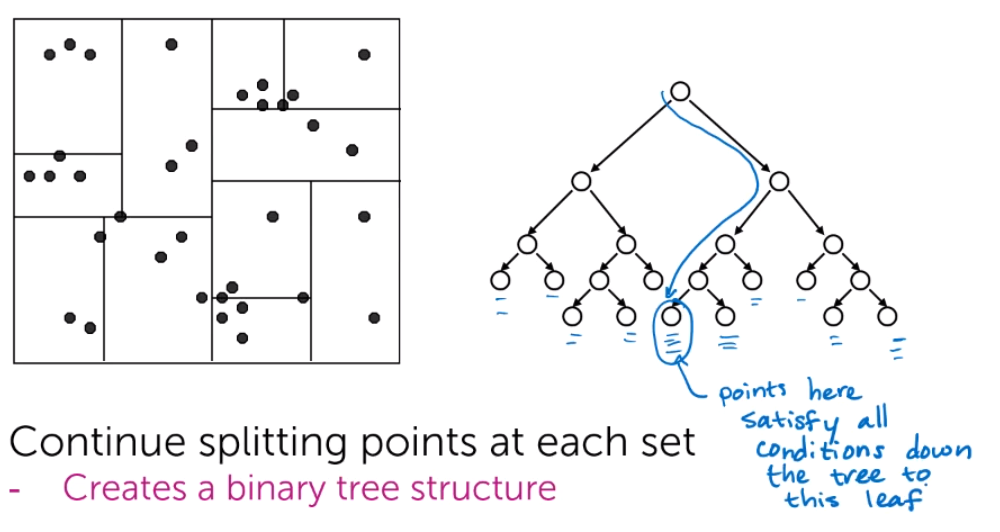
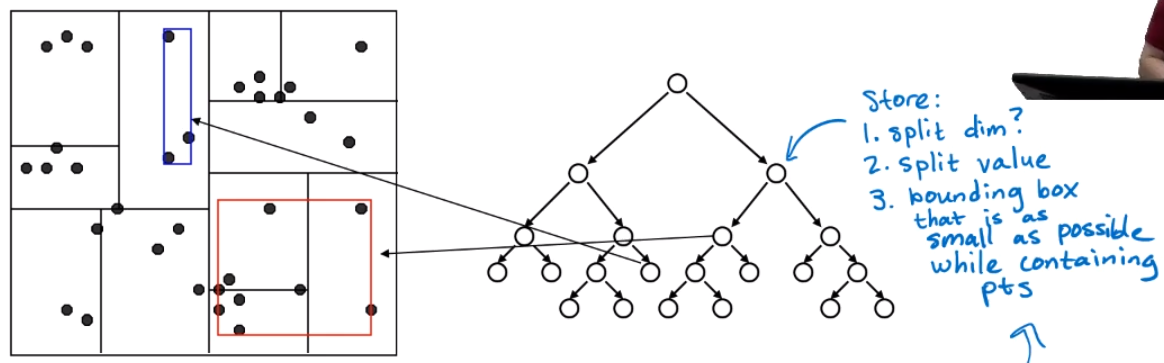
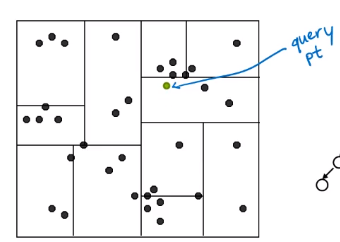
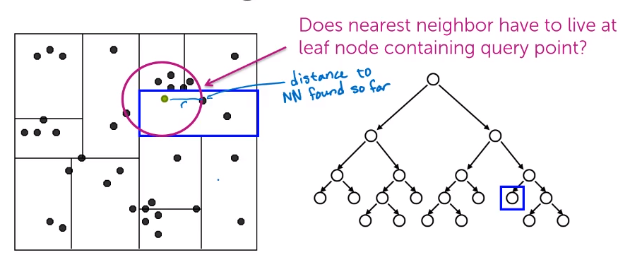
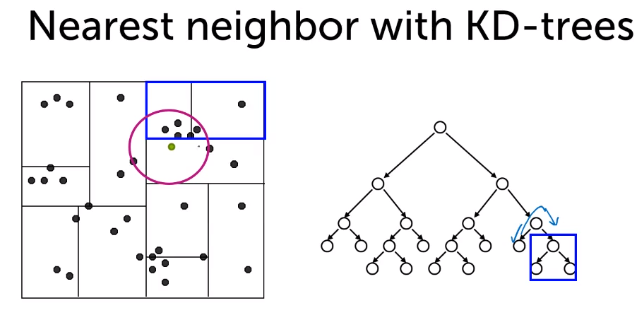
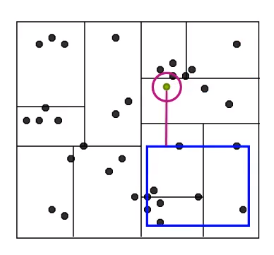
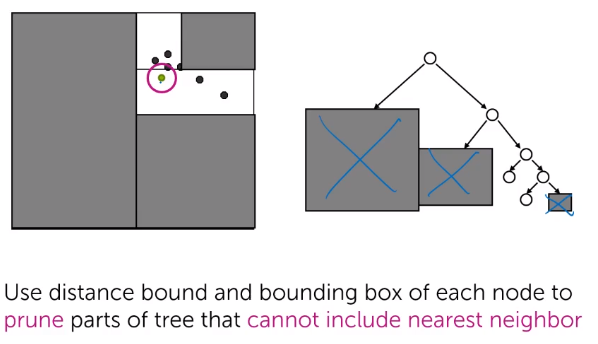
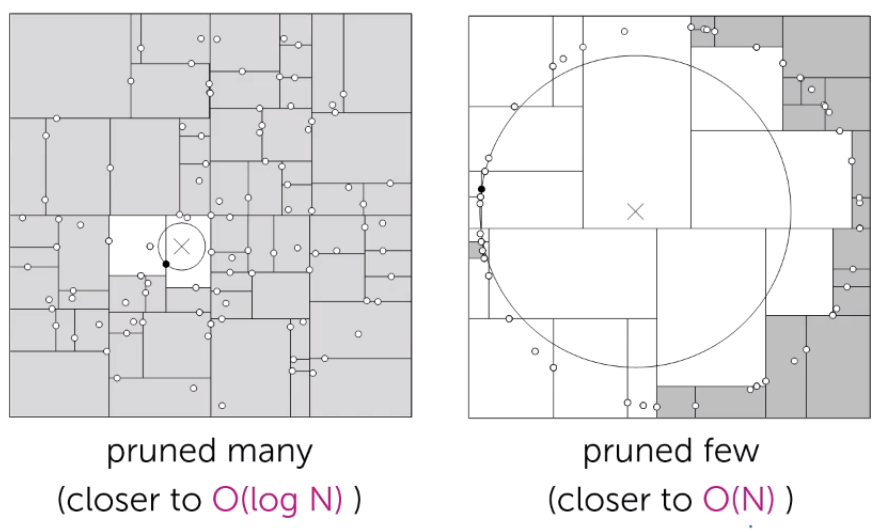
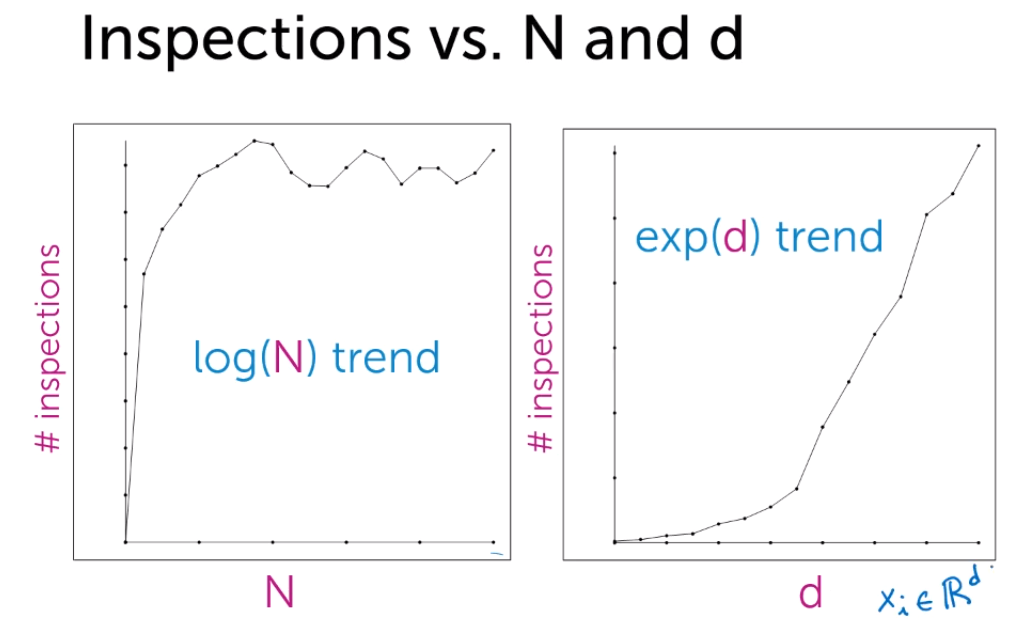
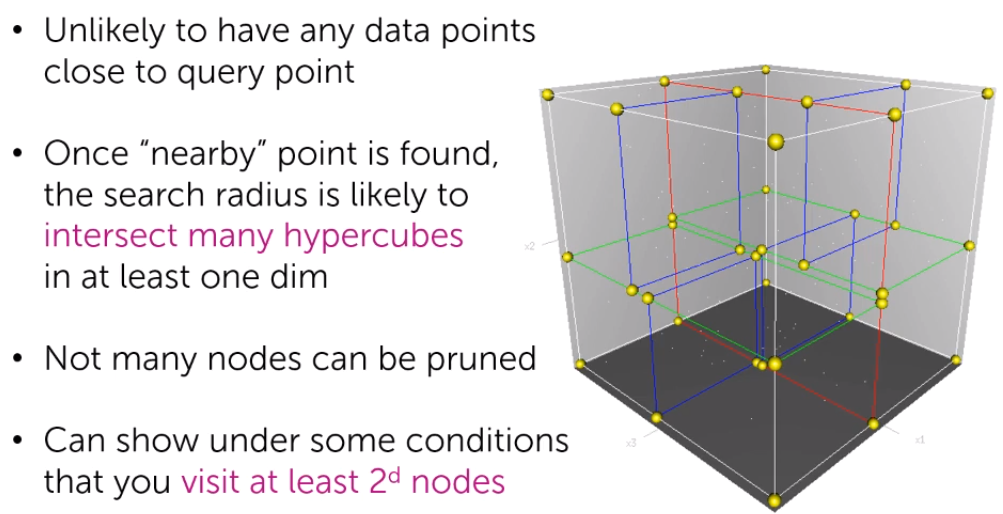
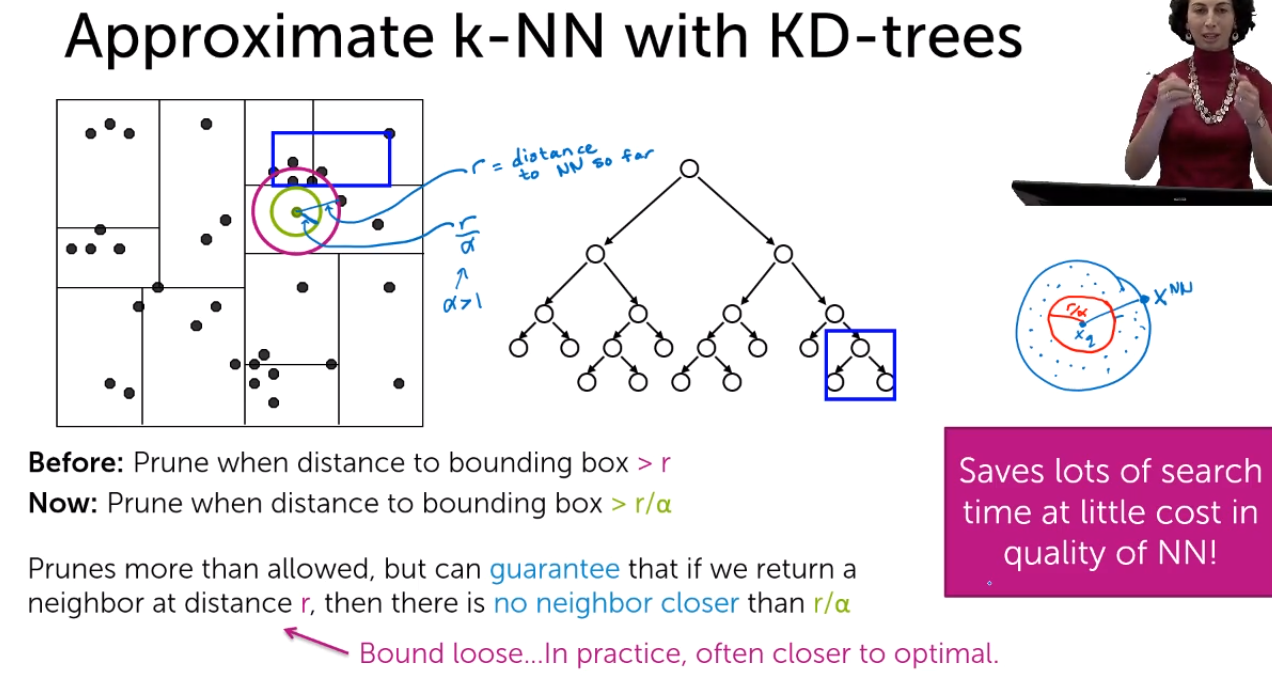
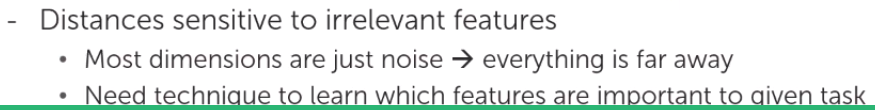
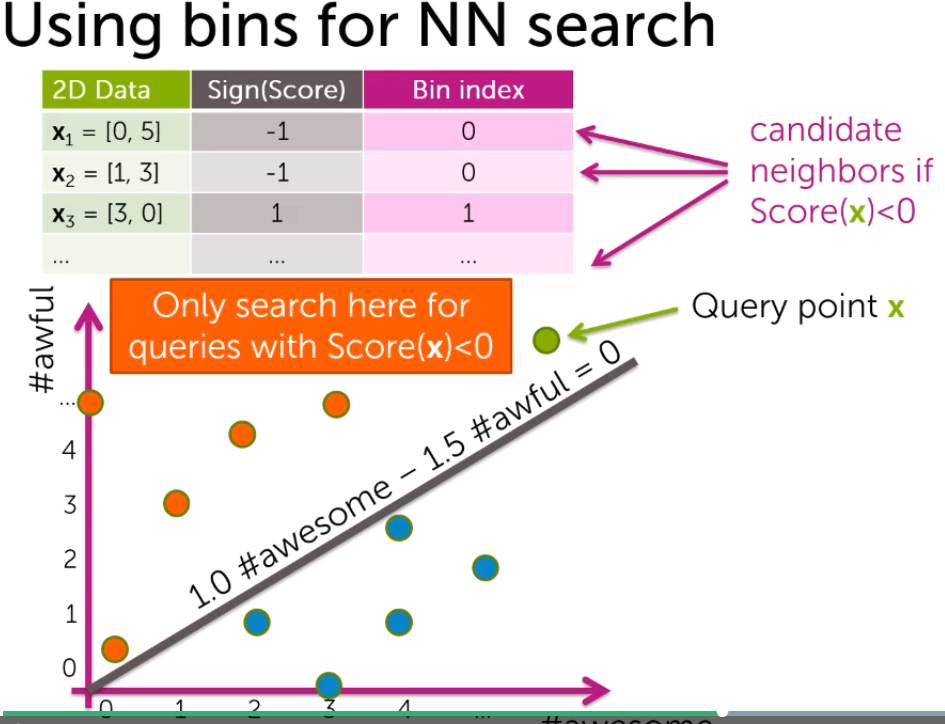
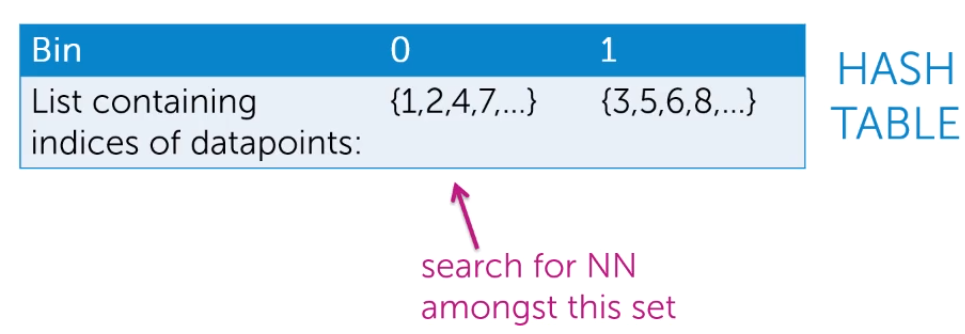
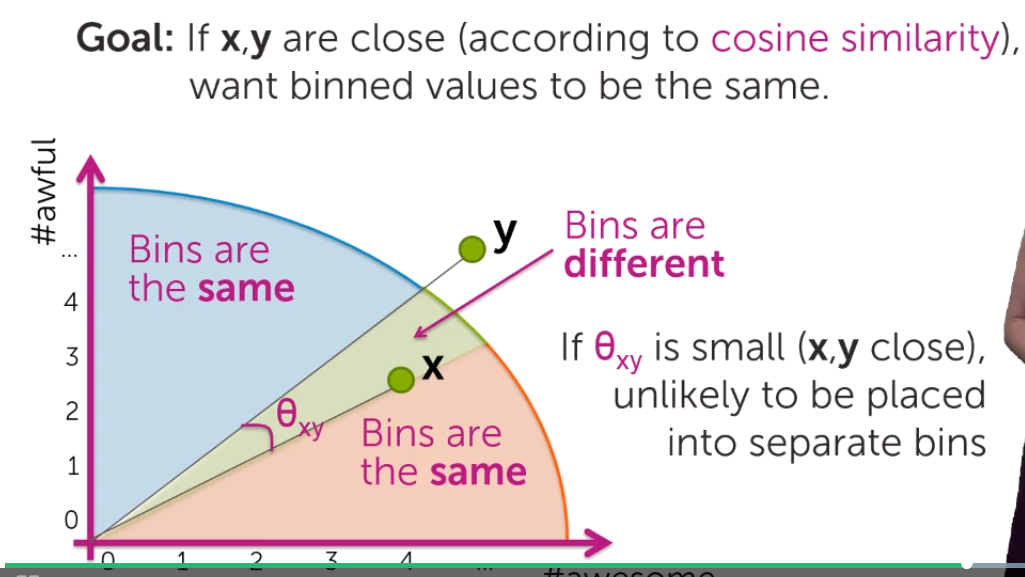
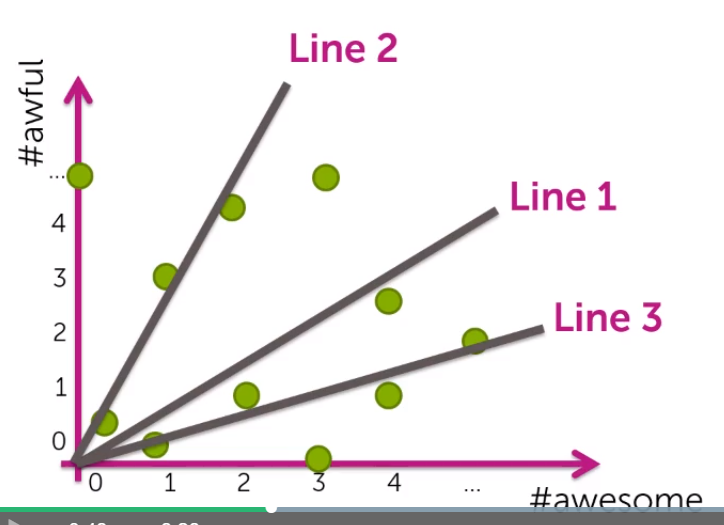
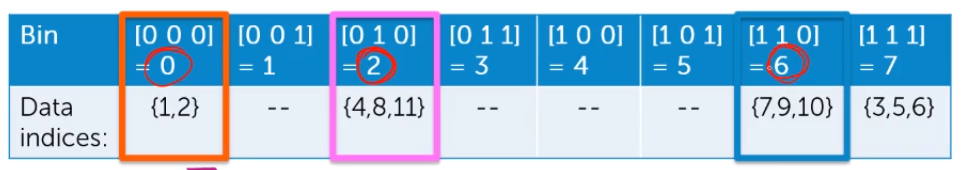
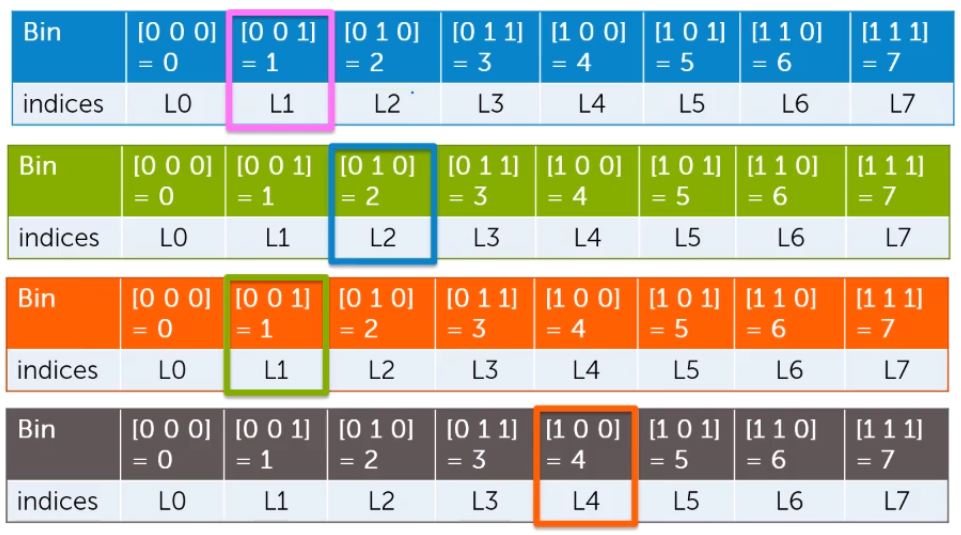
# Intro

* Retrieval – get similar products/documents to the one chosen.
* 
* Clustering – group products/documents in groups.
  + Often it could be an unsupervised learning in contrast to regression and classification

# Nearest Neighbour Search (k-Nearest Neighbours (k-NN algrorithm))

* We define similarity across articales. We compute distances between the article of interest and other articles. We pick the one (ones) that have the smallest distance. In k-NN, we pick k documents with lowest distances
* Document representation
  + Bag of words model – count all word. Problem is that focus will be on common words like ‘the’, ‘in’ and rare words like ‘volcano’ would be lost
  + TF-IDF – term frequency – inverse document frequency. Local frequency (word count in a document) / global frequency (word count in all documents). Thus, unique words will become more important.
* Distances:
  + Euclidean distance
    - Both magnitude and direction are taken into account
  + Weigth features differently -> e.g. abstract and title are more important than body
  + Scaled euclidean distance 
  + Cosine similarity = cos(theta) =  =  = 
    - Vectors are normalized – their length is made to 1
    - Cosine similarity looks only on direction (co-direction) of vectors, but ignores their magnitude (ignores length of documents)
    - Cosine similarity can make 1 tweet similar to 1 long article. Thus, in practice it is common to compare items (documents) of similar length and not normalizing them
    - In general cosine similarity = [-1,0,1] where 1 is maximum similarity, where -1 means opposite directions and 0 means orthogonality. For only positive features (like tf-idf) cosine similarity = (0,1)
  + Different metrics should be used for different purposes. For the contents of a short text, can use cosine similarity. But for numberical features use euclidean distance
* k-NN complexity:
  + brute force is very computationally expensive, especially if we often send querries
* KD-trees:
  + KD-trees allow splitting data in groups that would make search much faster
  + KD-trees work fine when we have low or medium dimensionality 
  + Grouping data points in boxes and storing this infor in a decision tree
  + 
  + 
  + Options
    - Which dimension to split on?
      * The widest
    - What value split at?
      * Median or center of the box
    - When do we stop?
      * Fewer than m points left OR box hits minimum width
  + Algorithm:
    - 1) Find our query point: Essentially we store all points in the leafs or a tree. To find the query point, we start from top of the tree and find where our query point is located.
    - 
    - 2) Looking for nearest neighbour in the same node/box/leaf
    - 
    - 3) Looking for closest NNs in other leafs nearby
    - 
    - To optimize search here, we calculate the distance to boundary box. If this distance is higher than the smallest distance so far, than we ignore it (prune it). Otherwise, we will start computing distances to the points in that box:
    - 
    - 4) We look to other boxes, and can prune them effectively:
    - 
  + Complexity:
    - O(log N) – O(N)
    - In general much more efficient than brute. Unless we are very unlucky, and that the structure of KD-trees is really bad:
    - 
    - 
    - When dimensions (D) is high => KD-trees not very effective
  + KD tree limitations:
    - In general it is hard to implement them efficiently
    - Ineffective when we have many dimensions:
    - 
* Optimization / k-NN search Approximation:
  + Finding AN approximate nearest neighbour vs THE nearest neighbour can increase efficiency a lot! In practice should be good enough and there is no need to look for the closest neighbour
  + 
  + 
    - Dimensionality reduction would really help (e.g. L1 regularization)
* Locality sensitive hashing (LSH) as an alternative to KD-trees
  + Calculate scores and put point in bin 0 with score <0 and to bin 1 with scores >=0. Then we will look only for the NNs in the same bin
  + 
  + We store points in a Hash table
  + 
  + But it will provide only approximate search. We will miss the closest NN if it is in another bin
  + Challenges:
    - 1. How to find good line(s)?
    - 2. How to prevent close points to be separated in separate bins?
    - 3. Bin can contain a lot of points, so still a lot of brute force there
  + Solutions:
    - Draw line randomly! (covers Challenge 1 and 2). Actually it is very unlikely that a random line would separate close NNs into separate bins
    - 
    - We can draw more lines and separate points in more bins! (thus cover cahllange 3)
    - It is more likely that 2 NNs will be split in separate bins. However, we can look not just in one bin, but also in several bins nearby. (Still could be a problem if there are many lines separating 2NNs)
    - 
    - We will search until good approximation is found or while our computational budget is reached
    - We encode each bin and thus can effectively loop over those that are nearby
    - Instead of comparing various bins, we can make several independent hashtables and look at 1 bin in each. This can be computationally more effective in most cases.
    - 
  + When we will do many queries, we should not worry much about the cost of building a LHS table or KD-tree, we should worry about the time to execute query
  + Many dimensions:
    - We will draw planes(hyperplane):
    - 