Clustering and Retrieval – Week 2 – k Nearest Neighbors Search

Introduction to nearest neighbor search and algorithms

The Query Article (or query document) is the article for which we want to find another similar article in the space of all articles. Articles are organized by similarity of text. We want to compute the 'distance' from the query article to each other article. We then choose the article with the shortest distance (greatest similarity) to the query article. That article is called the nearest neighbor.

In this case, since we are choosing only a single result, this is called 1-nearest neighbor search. If we wanted to find k articles, this would be k-nearest neighbors.

So this problem has two inputs;

x_q - the query article
$x_1, x_2, \ldots, x_n$ - the corpus of N documents to search

The output is the most similar article (the nearest neighbor), x^{NN}

$$x^{NN} = \min_{x_i} distance(x_q, x_i)$$

In pseudo code:

```
Dist2NN = ∞
x^NN = ∅
N = number of documents in the corpus
for i = 1,2,…,N
    δ = distance(x_i, x_q)
    if δ < Dist2NN
        x^NN = x_i
        Dist2NN = δ
return x^NN
```

k-nearest neighbor is setup similarly, but the output is now the set of k documents that are nearest the query document.

The problem has the same two inputs;

x_q - the query article
$x_1, x_2, \ldots, x_n$ - the corpus of N documents to search

The output is a set of k most similar articles such that those articles that are NOT in the similar set all have distances from x_q that are greater than or equal to the maximum distance from x_q of the members in the similar set:

$$X^{NN} = \{x^{NN_1}, \ldots, x^{NN_k}\} \; for \; all \; x_i \; not \; in \; X^{NN},$$
$$where \; distance(x_i, x_q) \geq \max_{x^{NN_j}, j=1,\ldots,k} \text{distance}(x^{NN_j}, x_q)$$

In pseudo code:

- Initialize X^{NN}, the set of similar documents sorted by distance to X_q, to the first k documents in the corpus.
- Initialize Dist2kNN to the sorted distances corresponding to elements in X^{NN}.
- N is the number of documents in the corpus
- for each of the remaining k+1 to N documents in the corpus
  - calculate the distance of the document from x_q
  - if that distance is less than kth distance in Dist2kNN (the maximum distance in the current set of similar documents) then
    - Insert the document into X^{NN} in sorted order by distance from x_q
    - Insert the distance of the document from x_q into Dist2kNN in sorted order
- return the set of similar documents X^{NN}

$x^{NN} = sort(x_i, \ldots, x_k)$
$Dist2kNN = sort(\delta_1, \ldots, \delta_k)$
`for i = k+1,…,N`
    $\delta = distance(x_i, x_q)$
    $if \; \delta < Dist2kNN[k]$
        $x^{NN} = x_i$
        $Dist2NN = \delta$
`return` x^{NN}

The importance of data representations and distance metrics

Bag of Words (Word Count) Model
- ignores the order of the words
- counts the number of instances of each word in the document

So bag of words **uses a vector for each document where each element is the word frequency for a given word**. Then **to calculate the similarity metric** for two documents, we would **multiply the two bag-of-words vectors and sum the resulting values** (this is stated as the inner (dot) product of the transpose of x_i and x_q:

$$bag\ of\ words\ similarity\ =\ {x_i}^T \cdot x_q = \sum_{j=1}^{d} x_i[j] * x_q[j]$$

where:

${x_i}^T$	is the transpose of the i-th document vector
x_q	is the query vector
d	is the number of features (the length of the document and query vectors)
$x_i[j]$	is the j-th element of the i-th document vector
$x_q[j]$	is the j-th element of the query vector

This simple model has a problem, in that it **biases toward long documents**. Long documents have larger word counts, so the resulting similarity metric will typically be larger as the documents become larger. That is not what we want.

To remove the bias of large word counts in long documents, **we can normalize each word count** value by dividing each entry **by the norm of the vector** (the square root of the sum of the squares of each element). This vector of normalized values is then used in the same way as the original word count vector; the query document vector is multiplied by the corpus document vector and the resulting elements are summed to get the similarity metric.

Event this normalized bag of words model has a problem in that **very common words that have little importance can come to dominate the calculation**. It is common for rare words that are idiosyncratic to the meaning of the content to be the most important words from a clustering point of view, but there word counts may be low compared to common words. For instance, ‘the’ is a very common word in English but has basically no value for assessing similarity of documents, where as ‘cerebellum’ is a rare word, but it’s inclusion in a document may mean the document’s content is related to the brain.

To handle this, we use TF-IDF (**Term Frequency – Inverse Document Frequency**), which gives higher weight to words that

- appear frequently in the query document (common in x_q)
- appear rarely in the corpus (rare in x_1 to x_N)

Term Frequency = word count in the document

Inverse Document Frequency = $log\left(\frac{document\ count}{1+number\ of\ documents\ using\ the\ word}\right)$

- you can see that if the word is common in all documents, then the value will approach log(1), which is zero.
- if the word is rarely seen in the corpus, then the value will approach the document count, which is presumable large, so log(large number) is a smaller but still relatively large number (much larger than zero).

So **Inverse Document Frequency discounts words based on how many documents they appear in**.

TF-IDF uses the product of these two terms to calculate a similarity metric that trades off local frequence and global rarity;

Term Frequency * Inverse Document Frequency

So if a word is frequent in the document, but rarely found in the other documents, then both terms of the product will be large and so the resulting weight will be large.

Distance metrics: Euclidean and scaled Euclidean

Euclidean distance in 2-space (the Cartesian coordinate system) is the 'straight line' distance between two points and can be calculated using the Pythagorean theorem; given a point p and a point q on 2D plane

$$distance(p,q) = distance(q,p) = |p-q| = \sqrt[2]{(p_x - q_x)^2 + \left(p_y - q_y\right)^2}$$

Euclidean distance in n-space between points p and q;

$$distance(p,q) = distance(q,p) = \sqrt[2]{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2}$$

Scaled Euclidean distance in n-space applies individual scaling factors to each term in the summation, prior to taking the square root.

This is useful for weighting features differently; more important features would be weight more heavily. For instance, the text in the title of a document may be more important that body text.

This can also be used to help handle the different spread in values for different features. This is so that features with large numbers but small spread ranges (max feature – min feature) don't dominate the calculation. For intuition, think about feature-a, it has a large min and a small range, but feature-b has a small max, but a large range. Without scaling, the documents where feature-a is at the min could dominate the calculation even though feature-b might be at the max. We can use scaling to normalize the spread across different features in order to avoid this. There are two ways to do this;

- **scale the value by the inverse of the range**. In this case, the scaling value for feature-j is calculated by finding the min and max of the feature value x_i across all observations (i = 1..N); **a_j = 1 / range(x_i[j]) = 1 / (max(x_i[j]) – min(x_i[n]).** Here x_i[j] is the i-th observation for feature-j. The scaling a_j value would then be used to multiply each feature value x_i in all N observations.
- An alternative to 1 / range(x_i[j]) is to use the variance(x_i[j]) in the denominator, so **scale the value by the inverse of the variance. a_i = 1 / variance(x_i[j])**

Scaled Euclidean distance between a document vector i and the query vector is given by;

$$distance(x_i, x_q) = \sqrt{a_1(x_i[1] - x_q[1])^2 + \cdots + a_d(x_i[d] - x_q[d])^2}$$

where:

d	the number of features (the dimension of the document vector). This has values 1..d.
x_i	the i-th document vector. This has dimension d.
x_q	the query vector. This has dimension d.
x_i[j]	the i-th observation of j-th feature, so x_i[1] is the i-th observation for the first feature and x_i[d] is the i-th observation for the last feature.
a_j	the scaling factor for feature j-th feature. So a_i is the scaling factor for the first feature and a_d is the scaling factor for the last feature.

Note that **if a feature weight has only binary values zero or 1, then this is equivalent to feature selection** where features with value 1 are used in the distance and features with value zero are not used in the distance calculation.

Obviously, the choice of distance metric and any associated feature scaling factors are critical to the algorithm. This is a difficult task in practice.

Non-scaled Euclidean distance as an inner product

We can arrange the distance squared as the inner (dot) product of the difference vector and it's transpose (remember that the dot product is the sum of the products of the corresponding entries of two vectors, see here https://en.wikipedia.org/wiki/Dot_product):

$$distance^2 = (x_i - x_q)^T \cdot (x_i - x_q)$$

$$= \begin{bmatrix}(x_i[1] - x_q[1]) & \dots & (x_i[d] - x_q[d])\end{bmatrix} \cdot \begin{bmatrix}(x_{i[1]} - x_q[1]) \\ \dots \\ (x_i[d] - x_q[d])\end{bmatrix}$$

$$= (x_i[1] - x_q[1])^2 + \cdots + (x_i[d] - x_q[d])^2$$

So:

$$distance(x_i, x_q) = \sqrt{(x_i - x_q)^T \cdot (x_i - x_q)}$$

where:

$x_i - x_q$	is the difference vector
$(x_i - x_q)^T$	is the transpose of the inner product
$(x_i - x_q)^T \cdot (x_i - x_q)$	is the inner product of the difference vector and it's transpose. This results in another vector which is the square of each difference.

We can layer in the scaling factors $a_1..a_d$ by arranging them in a diagonal matrix A where each scaling factor a_j is on the diagonal at A[j,j]

$$A = \begin{bmatrix} a_1 & & \\ & \ddots & \\ & & a_d \end{bmatrix}$$

$$distance^2 = \begin{bmatrix} (x_i[1] - x_q[1]) & \dots & (x_i[d] - x_q[d]) \end{bmatrix} * A * \begin{bmatrix} (x_{i[1]} - x_q[1]) \\ \dots \\ (x_i[d] - x_q[d]) \end{bmatrix}$$

$$distance(x_i, x_q) = \sqrt{a_1(x_i[1] - x_q[1])^2 + \cdots + a_d(x_i[d] - x_q[d])^2}$$
$$= \sqrt{(x_i - x_q)^T A (x_i - x_q)}$$

Distance Metrics: Cosine Similarity

When we first talked about the bag-of-words model, we developed a similarity metric:

$$bag\ of\ words\ similarity\ = {x_i}^T \cdot x_q = \sum_{j=1}^{d} x_i[j] * x_q[j]$$

We simply multiply each element in the document vector against it's associated element in the query vector and sum the results to get a scalar value. The larger the value, the more similar document-i is to the query document.

Cosine Similarity

As we discussed earlier, the simple bag-of-words similarity over-emphasizes long documents because they are likely to contain more instances of a given word. We can eliminate this bias by normalizing the vectors (dividing by their magnitude). Remember that the norm (magnitude) of a vector is the square root of the sum of the squares of all the elements (it is square root of the inner (dot) product of the vector with itself):

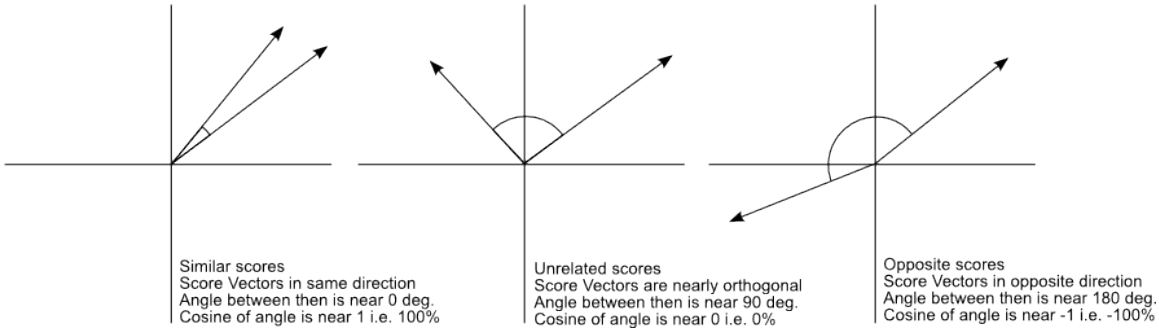
$$\|x\| = \sqrt{x^T \cdot x} = \sqrt{x[1]^2 + \cdots + x[d]^2} = \sqrt{\sum_{j=1}^{d} x[j]^2}$$

So if we modify the bag-of-words similarity so that we normalize both the document vector and the query vector, we get this;

$$similarity = \left(\frac{x_i}{\|x_i\|}\right)^T \cdot \left(\frac{x_q}{\|x_q\|}\right) = \frac{{x_i}^T \cdot x_q}{\|x_i\|\|x_q\|} = \cos\theta$$

This is called cosine similarity because it measures the cosine of the angle between the two vectors (see https://en.wikipedia.org/wiki/Cosine_similarity). The cosine of the angle varies from

Cosine similarity is not a proper distance metric; it fails the triangle test. It could also be referred to as angular similarity, in that it is a measure of how closely the angles of two vectors match. The cosine of 0° is 1, and it is less than 1 for any other angle. It is a measure of orientation: two vectors with the same orientation have a cosine similarity of 1, two vectors at 90° have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude. So cosine ranges from 1 to -1.



Taken from Christian S. Perone

For our purposes, **our features are always zero or positive, so cosine will vary from 0 to 1 for our data**. Given that, can use it to define a 'distance' metric;

$$distance = 1 - similarity$$

To normalize or not and other distance considerations

Remember that the simple bag-of-words similarity (dot product of the document and query vectors) is sensitive to the length of the documents, because long documents will be more likely to have higher word frequency. In fact, this measure, because it involves squaring the product of the elements, has an outsized effect; given a document and query with a similarity s, if we double the length of each document with no other changes, then similarity is s*4.

So it is not always desirable that documents become more similar as they become longer. So we have used the normalized vectors (vector divided by it's magnitude) to adjust for this. In this case, because we are ignoring the magnitude, only the angle between the vectors matters. So **with cosine similarity, we are making the length of a document an invariant**. With cosine similarity, we are focusing exclusively on the content of the documents and not their length.

But what if we do want the length of a document to have an effect? For instance, what if our query article is a very long article? **Because cosine similarity ignores length, then it is just as likely that a simple tweet may be found to be similar to a long query document as it is a document of comparable length**. That may be undesirable.

To make a tradeoff between the similarity of the content and the length, we can cap the word count on documents and NOT normalize. This would then mean that very long documents would not be overly similar.

Other Distance Metrics

This module has focused on distance metrics that are useful for text retrieval. Clustering is using for a lot more than text retrieval and **there are many other distance metrics that are useful in other domains**

- Mahalanobis
- rank-based
- correlation-based
- Manhattan
- Jaccard
- Hamming
- …

In practice, **it is common to apply different distance metrics for different features** of the documents. Sticking with text documents, we could compare the text of the document with Cosine similarity and metadata, such as the number of times the document has been read, with Euclidean distance. These could then be combined using weights (this is where domain knowledge would come in handy).

Scaling-up k-NN using KD-trees

Complexity of brute force search

The complexity of brute-force search, given a query point:

- 1-nearest neighbor
  - for each document in the corpus, calculate the distance to the query document
  - remember the smallest distance
  - return the smallest distance
- k-nearest neighbors
  - for each document in the corpus, calculate the distance to the query document
  - remember the distance in a k-length priority queue
  - return the k-length queue

The effort for **1-nearest neighbor brute force search is proportional to N**, the number of documents in the corpus.

The effort for **k-nearest neighbors**, when using a priority queue for efficiency, **is N log k**, since we need to insert in the priority queue each time. Insert into a priority queue is itself a 0(log N) operation (where N is the length of the queue), see here.

If N is very, very large and/or there will be many, many queries, then we need better performance.

KD-tree representation

KD-tree is also known as K-Dimensional Tree.

From Wikipedia: The k-d tree is a binary tree in which every node is a n-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Every node in the tree is associated with one of the n-dimensions, with the hyperplane perpendicular to that dimension's axis. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree.

For example, if for a particular split the "x" axis (axis associate with feature 'x') is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the split point.

As this relates to clustering, the kd-tree is used as a structured organization of documents. To build the kd-tree, we recursively partition the n-dimensional points created by a collection of feature vectors of length N into axis-aligned bounding boxes. The vectors (points in n-space) that fall into a given box are found in a leaf node of the tree. For most cases, this allows us to prune the search space, so that we can look at fewer documents when looking for neighbors.

Building the kd-tree is done recursively; a dimension is chosen to split and then a split value for that dimension is chosen. Each recursion creates a split that partitions that space into 2, resulting in a binary tree. Each node holds a split dimension and value. Walking the binary tree creates a bounding box in n-space. Those vectors that are in the same bounding box will be in the same leaf node and the bounds of the 'box' (as determined by the split dimensions and values) are in the nodes that lead from the root of the tree to that leaf node. Each point (each vector) is eventually bounded by one and only one box.

This is reflected in the data structure, which is a binary tree where each vector is associated with one and only one leaf node. The leaf node for a vector can be found by walking the kd-tree's nodes from the root; at each node the child node is chosen by comparing the vector at the split dimension to the split value. Once we get to a leaf node, the vector and it's neighbors in the bounding box are located.

Construction (from Wikipedia)
Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct k-d trees. The canonical method of k-d tree construction has the following constraints:

- As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have z-aligned planes, the root's great-grandchildren would all have x-aligned planes, the root's great-great-grandchildren would all have y-aligned planes, and so on.)
- Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of n points into the algorithm up-front.)

Note that it is not required to select the median point, but this method leads to a balanced k-d tree, in which each leaf node is approximately the same distance from the root. However, balanced trees are not necessarily optimal for all applications.

Given a list of n points, the following algorithm uses a median-finding sort to construct a balanced k-d tree containing those points.

```
function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtree
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```

How do we choose which dimension?
This is generally done with a heuristic of some sort, here are some examples:

- Choose the dimension with the largest range (the ‘widest’ dimension): max – min.
- or Cycle through all dimensions, choosing each in turn (wrapping around until done splitting). This ignores the data in the box, but can be faster because it avoids having to determine the min and max of the data, which is potentially costly if the data is very large.
- etc.

How do we choose what value in the dimension where we split?

- split at median of observations in the box. This has the advantage of creating a balanced kd-tree (all leaf nodes have the same number of observations +/- 1)
- or Choose the center of the observations. If you've already chosen the split dimension based on range, then this is a straightforward calculation: min + (max - min) / 2.
- or Choose the center of the box (based on range of that dimension is the parent node). This ignores the data in the box, which can be faster because it avoids potentially costly operations to find the min, max or median of the observations.
- or Select a random subset of observations and use that sample's median as an estimate of the median (this leads to nearly balanced trees in practice).
- etc.

When do we stop splitting?

- Stop splitting if there are fewer that a given number of points in the box
- or Stop splitting if the box hits a minimum width. Again, this ignores the data and so can be very fast.

What do we save at each split node?

- The split dimension
- The split value
- The observations that fall into the left node
- The observations that fall into the right node
- Optionally, the minimum bounding box for the actual observations in the node. This is an optimization that allows for efficient pruning when we do queries on the kd-tree.

NN search with KD-trees

Given a query vector, q, find the nearest neighbor in the corpus using a kd-tree build from the corpus.

1. Traverse the tree, moving from the root node to the leaf node that contains the query point.
    a. At each node, compare the query point at the split dimension to the split value.
    b. If it is less than the split value take the left node
    c. If it is greater than the split value, then take the right node (Note that if it is equal, then some tie breaker must be used. It could be as simple as >=, or it could alternate with each dimension, etc.)
    d. Once we are on a leaf node, we have located the query point's node (we have found the bounding box that the query point falls within).
2. At the query point's leaf node, search for nearest neighbors.
    a. compute the distance between the query point and each point in the leaf node.
    b. Record the minimum distance, r, and the associated nearest neighbor, nn.
3. Backtrack to the parent node and completely traverse the other (sibling) subtree, looking for nearest neighbor.
    a. At each node, check the bounding box (or the minimum bounding box for the actual observations, if that is being tracked).
        i. If the distance to the bounding box is greater than or equal the current nearest neighbor distance, r, then none of the points in the bounding box can be nearer, so we can stop looking.
    b. At each sibling leaf node, search for nearest neighbors
        i. compute the distance between the query point and each point in the leaf node.
        ii. Record the minimum distance, r, and the associated nearest neighbor, nn.
4. Continue backtracking in order to check the rest of the kd-tree in the same way. Efficient pruning will mean that most nodes will be skipped, so this happens very efficiently.

Note that **if the kd-tree will be used for many, many searches, then it is worth the computational cost to use efficient metrics for choosing split dimension and split value**. For instance, if we use the dimension range for choosing dimension, then we must track min and max at each node. However, then are useful for tracking the minimum bounding box, which maximizes pruning during queries. Tracking the median at each node and using it as the split value

results in a balanced tree that minimizes traversal depth during queries; essentially making it close to a standard binary search.

Complexity of NN search with KD-trees

Assuming that we used the range and median metrics for splits, then we would end up with a nearly balanced binary tree.

- Construction complexity
  - Size is 2N – 1 nodes if we have one value per leaf in a nearly balanced binary tree. This changes if we allow more values per leaf node, but in any case the complexity is still related to the number of observations, N. So O(N) complexity in regards to the size of the tree.
  - The Depth of the tree, again assuming a nearly balance binary tree, is log(N). So the complexity component associated with depth is 0(log N).
  - If using the median heuristic, so that we get a nearly balanced tree, at each level of the tree, we are splitting N values into some number of groups, so the complexity of making splits at each level of the tree is O(N)
  - Putting these components together, we have O(N) per level of the tree and log N levels of the tree, so the overall complexity for construct a tree with N observations is O(N logN)
- Query complexity for 1-NN
  - We do a search down the tree (which is binary) so that is O(log N).
  - We then backtrack. In the worst possible case, we would search all bins, and so need to do a distance calculation for each point, or N points, so that component is O(N).
  - So in the best case, we find the nearest neighbor in the first search and we cull all the other bins in backtracking; so that is O(log N) best case.
  - In the worst case, we cannot cull any bins in backtracking, so that is O(N).

Locality sensitive hashing for approximate NN search

Queries require traversing down the tree to the staring point, then backtrack.

- The initial traversal to find the query point's leaf node will require searching log N nodes (the depth of the nearly balanced binary tree). If we were to find the point in that node and know it, then that would be the minimum complexity.
- Backtracking, in the worst possible case, could then require us to search every point in every leaf node (so N points) if pruning doesn't help us. So the maximum complexity is the same as a brute force search of the points.
- So the complexity of a query has a range

    - Best case is O(log N)
    - Worst case is O(N)
- However, even with well distributed points, the number of comparisons becomes exponential in the dimension, d, of the feature vectors. So in high dimensional spaces, kd-trees don't work as well as other techniques.

Visualizing scaling behavior of KD-trees

For example, if we want to take each document in the corpus and find it's nearest neighbor in the corpus; so we have N documents so we are making N queries to find the 1-NN for each document.

- For brute force, we woud do N queries and each would be O(N) complexity, so the overall complexity for N queries is $O(N^2)$
    - for instance, given $N = 10^6$, $N^2 = 10^{12}$
- In the worst case, the query kd-tree may also by O(N), so it would be no better than brute force. However, if we have an efficient kd-tree, then each query is O(logN) so the overall complexity for N queries is O(N logN). For very large numbers of documents, that is a considerable improvement over $O(N^2)$.
    - for instance, given $N = 10^6$, $N*\log2(N) \sim= 2*10^7$

We can show this artificially by organizing our data in a circle; that will result in a search that have to visit almost every node in the tree.

Limitations of KD-trees

Even for the best case, in very high-dimensional spaces, we end up with many nodes in the tree and so many more checks must be made; the number of the checked nodes rises as power of 2 factor with the space dimension. Also, Computing the Euclidean distance between two d-dimensional points is an O(d) operation, so distance calculations become slower as the dimension increases.

From Wikipedia:
k-d trees are not suitable for efficiently finding the nearest neighbor in high-dimensional spaces. **As a general rule, if the dimensionality is d, the number of points in the data, N, should be $N >> 2^d$**. As d approaches log(n), the investment in using kd-trees begins to diminish until the resulting performance is no better than O(n). Otherwise, **when k-d trees are used with high-dimensional data, most of the points in the tree will be evaluated and the efficiency is no better than exhaustive search**. This comes down to the two things; in high dimensional space, most points are far away from each other (so r is large) and many or most of the hypercubes overlap in at least one dimension and so cannot be pruned.

- In high dimensional space it is unlikely to have any data points close to the query point unless $N >> 2^d$

- Since the 'nearest' neighbor candidate if generally far away from the query point, the search radius, r, is likely to intersect many hypercubes in at least one dimension.
- This means that no many nodes can be pruned
- I can be shown that under some conditions that a search visits at least 2^d nodes.

Andrew Moore: An introductory tutorial on kd-trees, explains the issues of dimensionality vs N when searching a kd-tree and was used as some of the source material for this module's video.

Extension to k-NN

This is very direct. Rather than keep just the a single neighbor and it's distance, r, to the query document, we keep a list of k nearest neighbors, sorted by distance to the query vector. Then 'r' is the distance to the last neighbor in the list; the k-th neighbor (the most distant of the nearest neighbors).

- When we add the first neighbor, then r is the distance to that neighbor.
- For each subsequent k-1 potential neighbors (up until we fill the list of k-nearest neighbors), we add the neighbor into the list in sorted order by distance to the query vector.
- After the list is full, when we compare a neighbor, we compare it's distance against the k-th neighbor (the most distant of the nearest neighbors); if it is closer, then the k-th neighbor is removed and the new neighbor is inserted in sorted order by distance from the query vector.

Approximate k-NN search using KD-trees

Sometimes (in fact, often) it is not necessary to find the exactly nearest neighbor (or k nearest neighbors). In fact, because creation of models is an inexact process, based as much on domain knowledge as math, the calculation of distance cannot be thought of as exact, as it might be when measuring things in the physical world. So with that in mind, even an exactly nearest neighbor search produces something that is still only 'exact' with the accuracy of the model we have created. An approximation in many cases is just as good as an exhaustively exact value.

This is becomes more true as the number of dimensions (features) increases, because some features are less relevant than others. As we add features, we are more likely to be including irrelevant features, which add noise (random distance in this case) to the model.

Efficiently choosing and weighting (learning) features is again seen as critical to creating a useful model, so that the distance calculation is done of relevant features.

A simple way to increase efficiency is to increase the level of pruning. The algorithm described previously using the distance to the bounding box to decide if it should look at the points inside the bounding box. In the algorithm, we prune if the distance from the query point to the bounding box is >= r, where r is the current distance to the nearest neighbor. If we include a factor in the distance, α, then we can modify this test such that we prune the node if the distance to the bounding box is >= r/ α. When α > 1, this then decreases the distance (the bounding box must be closer to be included in the search) and so increases the number of nodes that will be pruned. When α=1, this acts just like the 'default' nearest neighbor search.

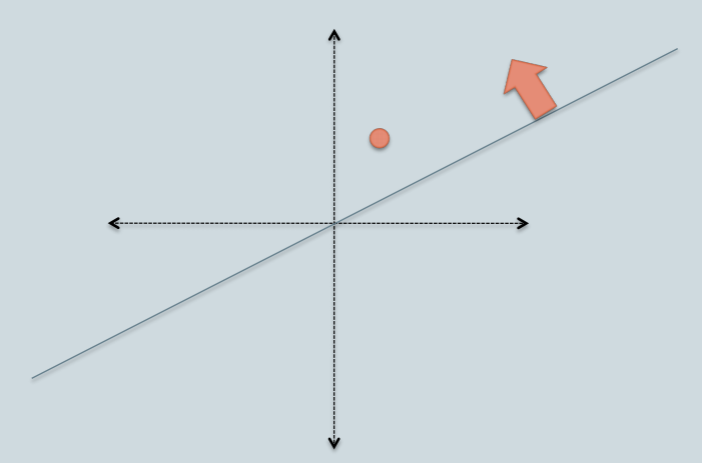
Other kd-trees

The kd-trees talked about here use axis-aligned splits, resulting in series of nested, axis-aligned bounding boxes (nested hyperrectangles). However, other shapes can be used to bound the points in each node. If nested hyperspheres are used, the resulting tree is called a ball-tree or metric-tree (see Wikipedia).

Locality sensitive hashing for approximate NN search

In this section we're going to return to an example that we used in the classification course, which is imagine we have reviews of restaurants. We have text from those reviews. And we're just going to count how many times we see the word awesome versus the word awful. And when we're thinking about retrieving similar types of reviews we're just going to look at these counts.

So we are going to calculate a decision boundary (for 2d data, a line), like we did in classification; this is called the Score(). Points below the line have a positive score, points above the line have a negative score. If the sign of the score is negative, points fall into the zero bin. If the sign is positive, the point is assign to bin 1.



Taken from cs-stackenchange

To simplify things, the boundary always goes through the origin. We use cosine similarity as our distance measure.

When we query, we calculate the Sign(Score(x)) for the query point, then we only search for the nearest neighbor in the resulting bin.

In practice, we use a hash table to maintain the list of points in each bin. Each bin is an entry in the hash table and each entry is a list of points in that bin.

This process does not deliver an exact nearest neighbor. If the query point is close to the boundary, then there may be another point close to the boundary on the other side that is the actual nearest neighbor; but will will not find it because we don't look in that bin.

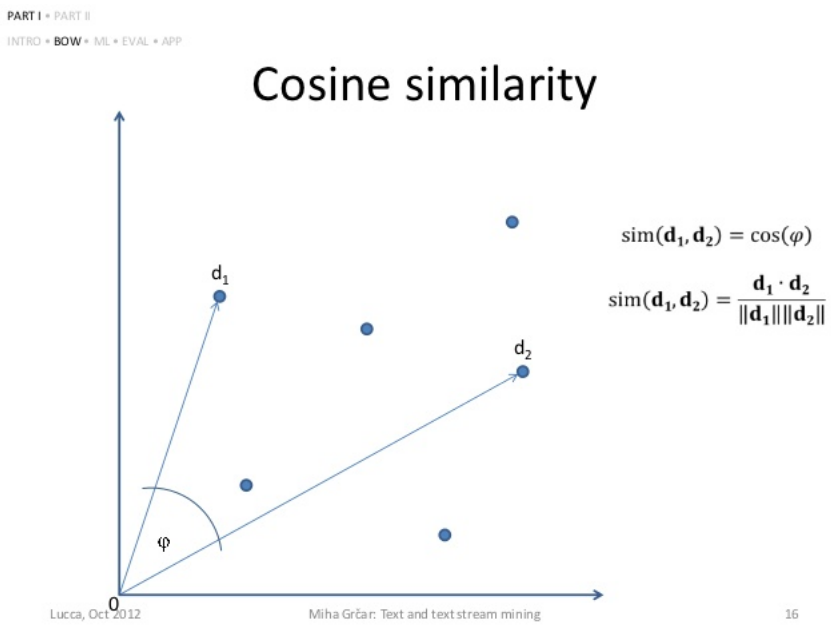
Using random lines to partition points

At a high level, **if two points are 'near' then we expect them to be in the same bin**. So we want an algorithm that puts them in the same bin.

We use cosine similarity to decide what is near (see a nice discussion on cosine similarity by Christian Perone). **So, given two points x and y, the cosine of the angle between them is θ_{xy}.**

What if we just pick a random line to partition the data? When, if the two points are close together, based on the cosine of the angle between them, then the chance of picking a random line that falls between them is very low. Conversely, if the two points are far apart, then the chance of picking a line that falls between them is high.

- **If θ_{xy} is small, then there is only a small probability that a random line will fall between them**.
- **If θ_{xy} is large, then there is a large probability that a random line will fall between them**.



Taken from presentation by Miha Grcar

So a random line provides good performance for providing a probabilistic query.

However, splitting points into two bins does not improve our search efficiency very much; each bin still has a lot of candidate points; we still must measure the distance between these points and the query point.

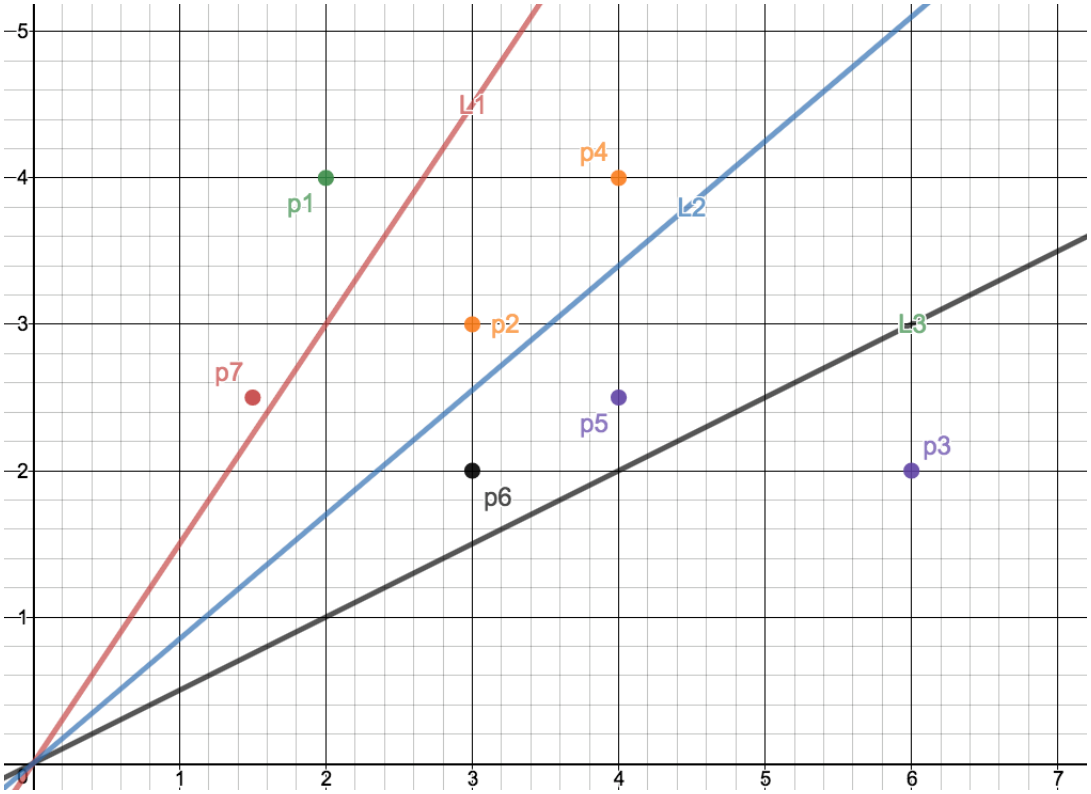
Defining more bins

So a single random line creates two bins; we only have to test, on the average, half the data, which is better than all the data, but is still not great.

What is we generate more random lines, and so more bins?
So we will need to test each point against each of these lines. As before, each of these tests yields a 0 or 1. We then concatenate these values into a binary number; that binary number is the hash which is used to lookup the bin in the hash table. Any point that 'hashes' to the same binary number (so all points that have the same Sign(Score(x)) for each of the random lines) are put in the same bin.

Our rule is that points that lie above the line are given a zero and points that lie below are line are given a 1 (note that this is arbitrary and it could be done in the opposite way, as is done in the diagram below). If we have 3 random lines, then there will be 3 scores that we concatenate, so the hash is a 3 digit binary number.

Note that all our data is non-negative (it is word counts), so we are only working in the upper-right quadrant. Given that, any point on the vertical axis (0, y) has hash [0,0,0] because it must be above all 3 lines. Conversely, if any point on the horizontal axis (x, 0) has hash [1,1,1] because it must fall below all 3 lines.



Created with Desmos

	Sign(Score1)	hash-digit 1	Sign(Score2)	hash-digit 2	Sign(Score3)	hash-digit 3	full hash
p1	-1	0	-1	0	-1	0	[0,0,0]
p2	1	1	-1	0	-1	0	[1,0.0]
p3	1	1	1	1	1	1	[1,1,1]
p4	1	1	-1	0	-1	0	[1,0,0]
p5	1	1	1	1	-1	0	[1,1,0]
p6	1	1	1	1	-1	0	[1,1,0]
p7	-1	0	-1	0	-1	0	[0,0,0]

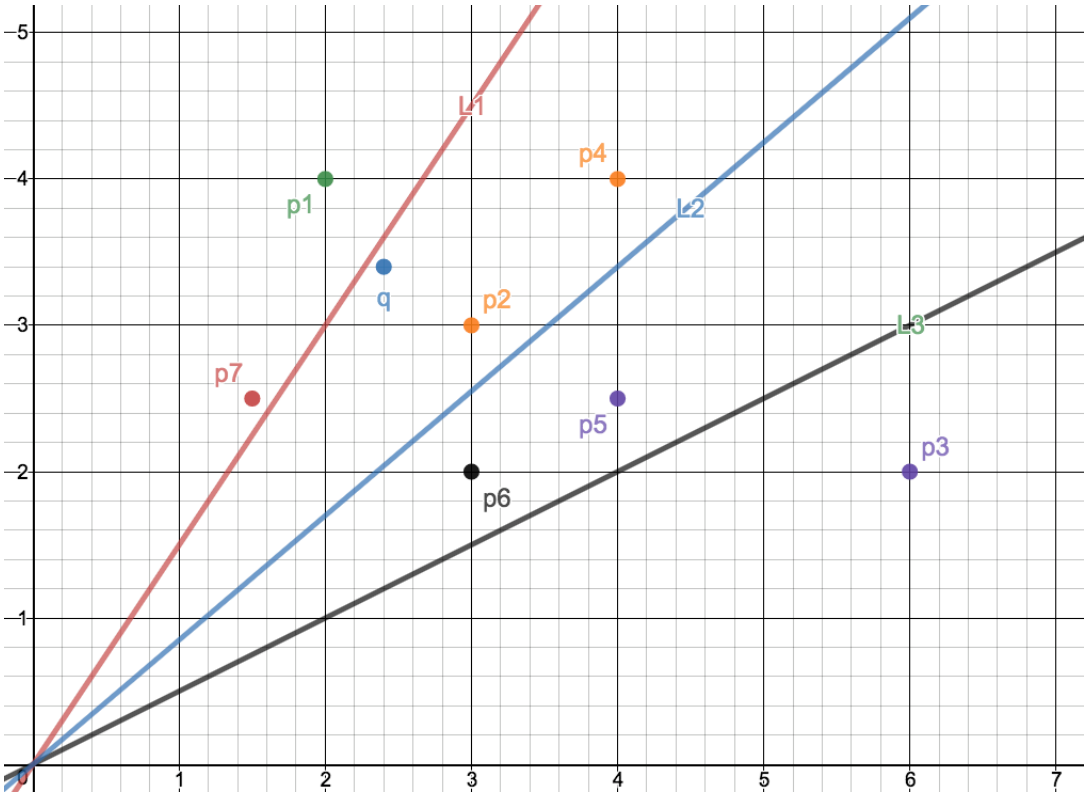
bin(0,0,0) [p1, p7]
bin(0,0,1)
bin(0,1,0)
bin(0,1,1)
bin(1,0,0) [p2,p4]
bin(1,0,1)
bin(1,1,0) [p5,p6]
bin(1,1,1) [p3]

Searching neighboring bins

Searching is done by applying the hashing algorithm to the query point and coming up with a bin. The points in the bin are then compared against the query point to find the nearest neighbor.

We have already said that LSH using cosine distance is an approximation because there may be a point that is closest to the query point but that is separated from it by a partition, and so that point is not considered. This becomes more likely as we add more lines, because it is more likely that we will create a line that separates near points simply because there are more lines. We are, in effect, trading off accuracy for speed when we add more lines.

Consider the bins we created in the prior illustration and the query point q = (2.4, 3.4).



In this case, q will hash into bin(1,0,0), so only p2 and p4 will be considered for nearest neighbor. In fact, p7 is the closest (based on Cosine distance), but is it not considered because it is in bin(0,0,0).

The good news is that we can 'tweek' the tradeoff between quality and speed even more if we have multiple lines; we can choose to search more bins. We are more likely to find a nearest neighbor by looking in 'nearby' bins. At a high level, this is done by 'flipping' bits in the hash to find a nearby bin.

We can limit the extra searching in order to fine-tune the trade-off between quality and efficiency. In practice, we can set some sort of computational budget (a maximum number of bins, or a maximum time, for instance) and stop looking in bins when we hit our budget limit. We can also set a metric for 'good enough' NN and stop when we find a good enough approximation. We can combine these approaches as well. In these ways; adding more lines and creating a metric that allows us to search more bins, we can improve accuracy and manage efficiency.

We see a trade-off between quality and performance as the number of random vectors increases: the query time goes down as each bin contains fewer documents on average, but on average the neighbors are likewise placed farther from the query, so more neighboring bins must be searched to find a good nearest neighbor. On the other hand, when using a small enough number of random vectors, LSH becomes very similar brute-force search: many documents appear in a single bin, so searching the query bin alone covers a lot of the corpus; including neighboring bins might result in searching all documents, just as in the brute-force approach.

LSH in higher dimensions

In higher dimensions we partition with hyperplanes (rather than lines, as in 2d space). So if instead of just #awful and #awsome as our vector elements, we might add a third word, #great. Now we would have feature vectors with 3 dimensions. As we add features, we add dimensions.

When we calculate Score(x), we need d multiplies to determine an index per partition (the hash-digit associated with each hyperplane partitition). In some applications (not all), this is a sparse vector and so a sparse multiplication, so we end up doing less than d multiplies per partition. In any case, this upfront cost does not need to be repeated; once the data points are assigned to bins, we don't need to recalculate the bin when doing a query.