Review: K-D tree

- Spatial partitioning hierarchy
- \( K = \) dimensionality of space (below: \( K = 2 \))

Counts of points in leaf nodes
Nearest neighbor search with K-D tree

Step 1: traverse to leaf cell containing query: compute closest point in this cell to the query.

Query point

Closest so far: A (at distance $d$)
Nearest neighbor search with K-D tree

Step 2: backtrack: if distance to other cells is closer than distance to closest point found so far, must check points in this cell

Query point

Closest so far: B (at distance $d'$)
Nearest neighbor search with K-D tree

Step 2: backtrack: if distance to other cells is closer than distance to closest point found so far, must check points in this cell.

Nearest neighbor result: B (at distance $d'$)
(Visited nodes during query shown in pink)
Approximate nearest neighbor (ANN) search

One simple answer: just take closest point in leaf node containing query

Approximate nearest neighbor: A (at distance $d$)
(nodes visited during query shown in pink)
Approximate nearest neighbor search

Improvement: place nodes in priority queue during downward traversal
Resume downward traversal from closest N nodes to query
Basic K-D tree build

- To find a partition for a node:
  - Partition axis for which the variance of current data points is the highest
  - Split at the median of the current data points
Randomized K-D tree

To find a partition for a node:

- Randomly choose axis to partition
  - Draw from distribution weighted proportionally with variance of current data points is the highest
  - Simple solution: pick partition axis by uniformly sampling from top $N$ axes with highest variance

- Randomly choose partition point
  - Draw from distribution heavily weighted at the median of the current data points (make it likely to split near the median of the data points)
ANN search using a forest of randomized K-D trees

- Construct a set ("forest") of random K-D trees
- For each tree, find NN in leaf cell containing query
  - Add all nodes (across all trees) traversed along the way to a priority queue (node priority = distance from query to node)
- Take closest of all answers across all trees as an initial ANN
- For top D nodes in queue, resume downward search from that node (D = 5 in figure [Muja et al. 2009])
- Solution for approximate k-NN as well
K-D search application: feature correspondence

- Example: SIFT descriptor, $K=128$
- For all descriptors in image 1, find nearest neighbor in image 2
Application: approximate K-means clustering *

- Assign \( N \) points to one of \( K \) clusters, subject to minimizing distance of points to their cluster centers

\[
\text{argmin} \sum_{i=1}^{k} \sum_{j \in S_i} \| p_j - \mu_i \|^2
\]

(for \( p_j \) in set of points in cluster \( i \) \( S_i \) and cluster center positions \( \mu_i \))

- Basic algorithm: \( O(kN) \) per iteration
  
  randomly initialize cluster means
  while (assignment of points to clusters continues to change)
  for each point \( p \):
    for each cluster \( c \):
      compute distance between \( p \) and mean(\( c \))
      assign \( p \) to closest cluster
    recompute cluster means

- Recall: clustering used to compute vocabulary for bag-of-words representation
  (given all features in database, assign each feature to one of \( K \)-clusters)

* On this slide: \( K \) is the number of clusters, not the dimensionality of the points!!!
Size matters: large vocabularies yield better retrieval performance

Consider bag of words implementation:

- $K = 100,000$ to $1,000,000$ words
- $N \sim 10$'s of millions when generating datasets for large vocabularies (train on sampling of descriptors from millions of images)

<table>
<thead>
<tr>
<th>Vocab Size</th>
<th>Bag of words</th>
<th>Spatial</th>
</tr>
</thead>
<tbody>
<tr>
<td>50K</td>
<td>0.473</td>
<td>0.599</td>
</tr>
<tr>
<td>100K</td>
<td>0.535</td>
<td>0.597</td>
</tr>
<tr>
<td>250K</td>
<td>0.598</td>
<td>0.633</td>
</tr>
<tr>
<td>500K</td>
<td>0.606</td>
<td>0.642</td>
</tr>
<tr>
<td>750K</td>
<td>0.609</td>
<td>0.630</td>
</tr>
<tr>
<td>1M</td>
<td><strong>0.618</strong></td>
<td><strong>0.645</strong></td>
</tr>
<tr>
<td>1.25M</td>
<td>0.602</td>
<td>0.625</td>
</tr>
</tbody>
</table>

Results from object retrieval task ($N = 16.7M$ for 1M word vocabulary)
mAP = mean average precision (average precision is precision averaged over all recall values)

[Philbin et al. CVPR 2007]
Basic K-means algorithm does not scale to large K

- Consider bag of words implementation:
  - $K = 100,000$ to $1,000,000$ words
  - $N \sim 1M$ (sampling of descriptors from millions of images)

- Approximate K-means:
  - Replace inner loop on previous slide with ANN search using K-D tree
    - randomly initialize cluster means
    - while (assignment of points to clusters continues to change)
      - construct K-D tree from cluster means
      - for each point $p$:
        - use approximate NN search to find closest cluster center
        - assign $P$ to closest cluster
        - recompute cluster means
  - Per-iteration run time: $O(N \lg k)$
  - Enables construction of much larger vocabularies ($\sim 1M$)
Approx. k-NN application to image retrieval

- Full representation of database
  - Search based on actual descriptor values, not quantized values

- Database:
  - K-D tree of features appearing in database images
  - e.g., SIFT descriptor: $K = 128$

- Search procedure:
  - Compute SIFT features for query image
  - For each descriptor
    - Find ANN descriptor in database (or k-NN)
    - Add “vote” for image containing feature (e.g., vote weighted by distance)
  - Rank database images by final score
Nearest neighbor image retrieval

- **Good:** no quantization of features like in bag of words
  - Common problem: how many visual words to create?
  - Active research area is design of good vocabulary

- **Cost:**
  - Storage of K-D tree is much larger than inverted index
    - Must store descriptor values, not just a weight (tf-idf) for each descriptor
    - Also store tree structure itself, but this is much less (unless forest gets large)
  - 1 million images, \( \sim 1,000 \) descriptors per image, 128 bytes = 128 bytes per descriptor → **128 GB database!!!**
Distributing a search tree

- Simple solution:
  - Partition dataset into chunks of data points that fit in memory on a node
  - Build K-D trees independently and in parallel on all nodes
  - For each query:
    - Broadcast query to all N nodes
    - Run N independent k-NN searches in parallel
    - Broadcast results to a master node
    - Master sorts results to produce overall k-NN

- Problems:
  - Lack of parallelism in the combine results stage
  - Less efficient structure
    - N independent K-D tree lookups
    - Search through single, large K-D tree would visit fewer nodes

[Figure credit: Aly et al. VISAPP 2011]
Distributing a search tree

- Idea: Store top part of tree in master, bottom parts of tree are distributed across nodes
- Tree construction:
  - Build top subtree using sampling of entire dataset that fits in memory
  - Top subtree height must be at least \( \lg(N) \) (to generate \( N \) leaf trees for \( N \) machines)
  - For each remaining datapoint:
    - Use search to determine which subtree data belongs to
    - Build leaf trees in parallel on respective nodes

[Figure credit: Aly et al. VISAPP 2011]
Distributing a search tree

- For each query:
  - Compute features, for each feature:
    - Search top of tree, find all leaf nodes within distance $d$ to query
    - Send query to these leaf nodes
    - All leaf nodes carry out search in parallel
    - Send k-NN results back to master for combination

- Good:
  - Efficacy similar to single big tree (each node contains an actual subtree, not a random sampling of data points)

- Bad: serialization of work at root

- Optimizations:
  - Replicate root tree to increase overall system throughput (but not individual query latency)
Computational characteristics

- **Inverted index**
  - **Computation:**
    - K-d tree lookup to quantize features into words (tree holds cluster centers)
    - Sparse dot products to compute image distances
  - **Storage:**
    - For each word, maintain list of documents and word TFIDF weight for word in each document: 4 to 8 bytes per descriptor

- **Full representation, approx k-NN search**
  - **Computation:**
    - K-d tree lookup to find k-NN
    - Dense dot products (e.g., 128-element vector) at the leaves
  - **Storage:**
    - Must store full descriptor representation (128 bytes for SIFT) for each occurrence
    - Also store tree structure (increasingly significant with a forest of trees)
Locality sensitive hashing

- Basic intuition:
  - Hash points into buckets, such that points nearby in space are likely to fall into the same (or nearby) buckets

- Given $x_1$ and $x_2$ and distance $r$
  - If $d(x_1, x_2) < r$, then $P(h(x_1) = h(x_2))$ is high
  - If $d(x_1, x_2) > \alpha r$, then $P(h(x_1) = h(x_2))$ is low
Locality sensitive hashing

- Example: pick \( m \) random projections
  - For each input query, hash into \( m \) different hash keys (associated with \( m \) different hash tables)
  - Union of data points from matching bins is candidate nearest neighbor set
    - Compute full distance function on these points
Locality sensitive hashing (as an embedding)

- Example: pick $m$ random projections
  - For each input query, compute 1 bit per projection
  - Query now reduced to $m$-bit string
  - 1 hash table containing ($m$-bit keys)
  - Check all hash bins with hamming distance similar to query!

Note: much better ways to determine set of hash functions than random projections (Learn them from the data)
Image retrieval summary

- Key issues at scale:
  - Quality of results
  - Speed of query
  - Space footprint of index

```
image → Compute queries/keys (SIFT, BOW, embedding, hash functions) → Index lookup → Filter → results
```