Case Study 2: Document Retrieval

Task Description: Finding Similar Documents
1-Nearest Neighbor

- **Articles**
  \[ X = \{ x^1, \ldots, x^n \} \times \mathbb{R}^d \]

- **Query:**
  \[ x \]

- **1-NN**
  - **Goal:** Find an article in \( X \) "closest" to \( x \)
  - **Formulation:**
    \[ x^{NN} = \arg \min_{x \in X} d(x, x^i) \]
Issues with Search Techniques

- Naïve approach: **Brute force search**
  - Given a query point $x$
  - Scan through each point $x^i$
  - $O(N)$ distance computations per 1-NN query!
  - $O(N\log k)$ per $k$-NN query!

- What if $N$ is huge??? (and many queries)
Think about Web Search/Image Search

- How big is $N$?
  $$N = \# \text{ web pages}$$

- How fast do we desire to do recall?
Smarter approach: *kd-trees*

- Structured organization of documents
  - Recursively partitions points into axis aligned boxes.
- Enables more efficient pruning of search space
  - Examine nearby points first.
  - Ignore any points that are further than the nearest point found so far.

*kd-trees* work “well” in “low-medium” dimensions

- We’ll get back to this...
Keep one additional piece of information at each node:
- The (tight) bounds of the points at or below this node.
Traverse the tree looking for the nearest neighbor of the query point.
Examine nearby points first:

- Explore branch of tree closest to the query point first.
Examine nearby points first:
- Explore branch of tree closest to the query point first.
When we reach a leaf node:

- Compute the distance to each point in the node.
When we reach a leaf node:

- Compute the distance to each point in the node.
Then backtrack and try the other branch at each node visited
Nearest Neighbor with KD Trees

- Using the distance bound and bounding box of each node:
  - Prune parts of the tree that could NOT include the nearest neighbor
Complexity

- For (nearly) balanced, binary trees...

Construction
- Size: \( O(N) \)
- Depth: \( O(\log N) \) (under some assumptions)
- Median + send points left right:
- Construction time: \( O(N\log N) \)

1-NN query
- Traverse down tree to starting point: \( O(\log N) \)
- Maximum backtrack and traverse: \( O(N) \)
- Complexity range: \( O(\log N) \) for worst case traversal

- Under some assumptions on distribution of points, we get \( O(\log N) \) but exponential in \( d \) (see citations in reading)
Cover trees (+ ball trees)

- What about exact NNs searches in high dimensions?
- Idea: utilize triangle inequality of metric (so allow for arbitrary metric)
- cover-tree guarantees:
What you need to know

- Document retrieval task
  - Document representation (bag of words)
  - \( \text{tf-idf} \)
- Nearest neighbor search
  - Formulation
  - Different distance metrics and sensitivity to choice
  - Challenges with large \( N \)
- \( \text{kd-trees} \) for nearest neighbor search
  - Construction of tree
  - \( \text{NN} \) search algorithm using tree
  - Complexity of construction and query
  - Challenges with large \( d \)
Case Study 2: Document Retrieval

Locality-Sensitive Hashing
Random Projections for NN Search

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Sham Kakade
April 14th, 2016
Using Hashing to Find Neighbors

• KD-trees are cool, but...
  – Non-trivial to implement efficiently
  – Problems with high-dimensional data

• Approximate neighbor finding...
  – Don’t find exact neighbor, but that’s OK for many apps, especially with Big Data

• What if we could use hash functions:
  – Hash elements into buckets:
    – Look for neighbors that fall in same bucket as x:

• But, by design...
What to hash?

- Before: we were hashing ‘words’/strings

- Remember, we can think of hash functions abstractly:

- Idea of LSH: try to hash similar items into same buckets and different items into different buckets
Locality Sensitive Hashing (LSH)

• A LSH function $h$ satisfies (for example), for some similarity function $d$, for $r > 0$, $\alpha > 1$, $1 > P_1, P_2 > 0$:
  – $d(x, x') \leq r$, then $P(h(x) = h(x'))$ is high, with prob $> P_1$
  – $d(x, x') > \alpha r$, then $P(h(x) = h(x'))$ is low, with prob $< P_2$
  – (in between, not sure about probability)
LSH: basic paradigm

• Step 0: pick a ‘simple’ way to construct LSH functions
• Step 1: (amplification) make another hash function by repeating this construction

• Step 2: the output of this function specifies the index to a bucket.

• Step 3: use multiple hash tables. for recall, search for similar items in the same buckets.
• Pick a random vector $\nu$:
  – Independent Gaussian coordinates

• Preserves separability for most vectors
  – Gets better with more random vectors
Multiple Random Projections: Approximating Dot Products

- Pick $m$ random vectors $v(i)$:
  - Independent Gaussian coordinates

- Approximate dot products:
  - Cheaper, e.g., learn in smaller $m$ dimensional space

- Only need logarithmic number of dimensions!
  - $N$ data points, approximate dot-product within $\varepsilon > 0$:

\[ m = \mathcal{O}\left(\frac{\log N}{\varepsilon^2}\right) \]

- But all sparsity is lost
LSH Example function: Sparser Random Projection for Dot Products

- Pick random vector $v$
- Simple 0/1 projection: $h(x) =$

- Now, each vector is approximated by a single bit

- This is an LSH function, though with poor $\alpha$ and $P2$
LSH Example continued: Amplification with multiple projections

- Pick random vectors $v^{(i)}$
- Simple 0/1 projection: $\phi_i(x) = \ldots$

- Now, each vector is approximated by a bit-vector

- Dot-product approximation:
LSH for Approximate Neighbor Finding

• Very similar elements fall in exactly same bin:
  
• And, nearby bins are also nearby:
  
• Simple neighbor finding with LSH:
  – For bins $b$ of increasing hamming distance to $\phi(x)$:
    • Look for neighbors of $x$ in bin $b$
  
    – Stop when run out of time
  
• Pick $m$ such that $N/2^m$ is “smallish” + use multiple tables
LSH: using multiple tables
<table>
<thead>
<tr>
<th></th>
<th>Query time</th>
<th>Space used</th>
<th>Preprocessing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vornoi</td>
<td>$O(2^d \log n)$</td>
<td>$O(n^{d/2})$</td>
<td>$O(n^{d/2})$</td>
</tr>
<tr>
<td>Kd-tree</td>
<td>$O(2^d \log n)$</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>LSH</td>
<td>$O(n^\rho \log n)$</td>
<td>$O(n^{1+\rho})$</td>
<td>$O(n^{1+\rho} \log n)$</td>
</tr>
</tbody>
</table>
Two big problems with random projections:
- Data is sparse, but random projection can be a lot less sparse
- You have to sample m huge random projection vectors
  - And, we still have the problem with new dimensions, e.g., new words

Hash Kernels: Very simple, but powerful idea: combine sketching for learning with random projections

Pick 2 hash functions:
- $h$: Just like in Count-Min hashing
- $\xi$: Sign hash function
  - Removes the bias found in Count-Min hashing (see homework)

Define a “kernel”, a projection $\phi$ for $x$: 
Hash Kernels, Random Projections and Sparsity

\[ \phi_i(x) = \sum_{j : h(j) = i} \xi(j)x_j \]

- Hash Kernel as a random projection:
  - What is the random projection vector for coordinate \( i \) of \( \phi_i \):

- Implicitly define projection by \( h \) and \( \xi \), so no need to compute apriori and automatically deals with new dimensions
- Sparsity of \( \phi \), if \( x \) has \( s \) non-zero coordinates:
What you need to know

• **Locality-Sensitive Hashing (LSH):** nearby points hash to the same or nearby bins
• LSH uses **random projections**
  – Only $O(\log N/\varepsilon^2)$ vectors needed
  – But vectors and results are not sparse
• Use LSH for nearest neighbors by mapping elements into bins
  – Bin index is defined by bit vector from LSH
  – Find nearest neighbors by going through bins
• **Hash kernels:**
  – Sparse representation for feature vectors
  – Very simple, use two hash functions
    • Can even use one hash function, and take least significant bit to define $\xi$
  – Quickly generate projection $\phi(x)$
  – Learn in projected space