LSH: Efficiently Finding Similar Objects

Vinicius Vielmo Cogo

Web Applications – May 5, 2017
Locality Sensitive Hashing

LSH is an efficient algorithm to find similar objects using hashes
Relation to Web Applications

- Information retrieval (finding similar objects)
- Improving existent web services with new features
- Using existent data structure as a service (e.g., DynamoDB)
- Creating new data structure as a service (e.g., LSH as a service)
Recommendation Algorithms

Suggest something to users ...
Recommendation Algorithms

Customers who bought this object also bought …
Recommendation Algorithms

- **Facebook** → ... people you may know
- **Tinder** → ... people you may like
- **YouTube** → ... videos you may like
- **Netflix** → ... movies you may like
- **Spotify** → ... music you may like
- **Amazon** → ... products you may like
Recommendation Algorithms

How do they work?
Recommendation Algorithms

Task: Suggest 30 tracks per week for each user
Recommendation Algorithms

General Recommendations
(e.g., popular songs)

≠

Personalized recommendations
Example: Music Recommendation

1) Find **similar users** (with similar preferences)
   Comparing the list of things they like

2) Suggest what one likes and the other doesn’t know yet
Example: Music Recommendation

P1

P2

P3
Example: Music Recommendation

P1

A  B  C  D  E

P2

A  B  D  E  F

P3

A  V  X  Y  Z
Example: Music Recommendation

P1

P2
Example: Music Recommendation

P1

P2
Example: Music Recommendation

How to identify similar users?
Example: Music Recommendation

P1  
A  B  C  D  E

P2  
A  B  D  E  F

P3  
A  V  X  Y  Z

Jaccard Distance*  
Similarity = \frac{|A \cap B|}{|A \cup B|}

*See extra slides for other distances
Example: Music Recommendation

\[ \frac{|P_1 \cap P_2|}{|P_1 \cup P_2|} = \frac{4}{6} = 0.667 \]

\[ \frac{|P_2 \cap P_3|}{|P_2 \cup P_3|} = \frac{1}{8} = 0.125 \]

\[ \frac{|P_1 \cap P_3|}{|P_1 \cup P_3|} = \frac{1}{8} = 0.125 \]
Example: Music Recommendation

\[ \frac{|P_1 \cap P_2|}{|P_1 \cup P_2|} = \frac{4}{6} = 0.667 \]

\[ \frac{|P_2 \cap P_3|}{|P_2 \cup P_3|} = \frac{1}{8} = 0.125 \]

\[ \frac{|P_1 \cap P_3|}{|P_1 \cup P_3|} = \frac{1}{8} = 0.125 \]
Millions of users that listen thousands different songs each
Problem

Users = Objects

Songs = Dimensions*

Millions of users that listen thousands different songs each

* See extra slides for other dimension examples
Problem

Comparing 60M users

Naïve: $60M \times 60M = 3.6 \times 10^{14} \text{ s} = 11M \text{ years}$

Smart: $60M \times \log(60M) = 46M \text{ s} = 14 \text{ years}$
Problem

1000 machines = 1 week

What if fails?

Being more efficient
Does it solve efficiently similarity search with more than 10 dimensions?
Locality Sensitive Hashing

Locality Sensitive Hashing

LSH is an efficient algorithm to find similar objects using hashes
So, imagine millions of buckets
Locality Sensitive Hashing

Given an object (user): hash it, obtain a value, and place the object in this bucket
Locality Sensitive Hashing

In a way that objects in the same bucket have bigger probability of being similar
Locality Sensitive Hashing

Calculate the **distance** between objects **within** the same **bucket** only
Locality Sensitive Hashing

LSH = Hash Function + Hash Tables

MinHash* + MultiMap[]

*See extra slides for other hash families
MinHash

- An array with the minimal hashes from all dimensions for each hash function

\[
\text{minHash}(\text{P1})
\]

\[
\begin{bmatrix}
23 & 12 & 5 & 45 & 7 & 34 & 12 & 76 & 87 & \ldots
\end{bmatrix}
\]

\[
\text{minHash(\text{Object o}):}
\]

```python
num_hashes <- 200  //defined based on a similarity threshold
hashes <- new hash[num_hashes]  //200 different hash functions
minHash <- new int[num_hashes]  //MinHash of the object
for i in 0..hashes:  // for each hash function
    for d in object.dimensions:  //for each dimension (music)
        hi <- hashes[i](d)  //calculate the hash of d
        minHash[i] <- min(minHash[i], hi)  // store the min
```

- Converts \textbf{variable} number of dimensions to a \textbf{fixed} configurable number
- Using the \textbf{same order} of hash functions is important to \textbf{find similar} objects
Locality Sensitive Hashing

**break it into b bands and r rows** (based also on the desired similarity threshold)

<table>
<thead>
<tr>
<th>23</th>
<th>12</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>7</td>
<td>34</td>
</tr>
<tr>
<td>12</td>
<td>76</td>
<td>87</td>
</tr>
</tbody>
</table>

Each band of r rows is the key for a different hashtable

**hashTable[0]**

| 23 | 12 | 5 | [P1] |

**hashTable[1]**

| 45 | 7  | 34 | [P1] |

**hashTable[2]**

| 12 | 76 | 87 | [P1] |

LSH Index
Locality Sensitive Hashing

break it into $b$ bands and $r$ rows (based also on the desired similarity threshold)

Each band of $r$ rows is the key for a different hashtable

- **hashTable[0]**  
  - 23 12 5 [P1, P2]
- **hashTable[1]**  
  - 45 7 34 [P1]
  - 73 22 15 [P2]
- **hashTable[2]**  
  - 12 76 87 [P1]
  - 3 28 56 [P2]
Interfaces

distance(Object o1, Object o2)
insert(Object o)
Queries (similarity search):
   nearestNeighbor(Object o)
★★★★ nearNeighbors(Object o, int maxNeighbors)
   clustering(Object o)
Challenges: Implementing

- Generic to any object
- Providing multiple hash function families (generic to all distances)
- Being efficient (space and time)
- Durability
Take outs

- LSH efficiently solves similarity search
- LSH is very useful for several applications
- Similarity search is usually a step to something bigger
- Think what do with the similarity knowledge
Thank you!
# Other Objects and Dimensions

| 1 Dimension | Binary values: 0 or 1  
|             | Numbers: age, height, weight, etc. |
| 2 Dimensions | Cartesian coordinates: \((x, y)\)  
|             | Tuples: \((k, v)\) |
| 3 Dimensions | 3D coordinates: \((x, y, z)\)  
|             | Animation: \((\text{time}, x, y)\) |
| N Dimensions | Characters in a string: “abcdefgh”  
|             | Substrings of a string: “abc”, “bcd”, “cde”…  
|             | Bits in a Byte array: 0011 1101  
|             | Words in a sentence: “Foo bar bar foo”  
|             | Sentences in a document  
|             | Pixels in an image  
|             | Notes in a music  
|             | Music in a playlist  
|             | Properties in an object  
|             | Columns in a DB row  
|             | *Minutiae* of fingerprints |
## Distances and LSH families

<table>
<thead>
<tr>
<th>Distance</th>
<th>Description</th>
<th>LSH family</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euclidean</td>
<td>Distance between two vectors</td>
<td>Random projections</td>
</tr>
<tr>
<td>Jaccard</td>
<td>$\frac{\text{len(intersection)}}{\text{len(union)}}$</td>
<td>MinHash</td>
</tr>
<tr>
<td>Cosine</td>
<td>Angular distance between vectors</td>
<td>SimHash</td>
</tr>
<tr>
<td>Hamming</td>
<td>Number of Substitutions</td>
<td>BitSampling</td>
</tr>
<tr>
<td>Levenshtein</td>
<td>Minimal number of substitutions, insertions and deletions</td>
<td></td>
</tr>
</tbody>
</table>
Who uses LSH for what?

Google
Detect near-duplicate web pages
Detecting Near-Duplicates for Web Crawling
Google News recommendations
Google News Personalization: Scalable Online Collaborative Filtering

UBER
Detect very similar routes
https://spark-summit.org/2016/events/locality-sensitive-hashing-by-spark/

Eventbrite
Detect spam and malicious messages for event organizers

Facebook
Clustering People

Spotify
Spotify recommender system
LSH forest - ANNOY

And others
Available LSH implementations

• **OpenLSH** ([https://github.com/singhj/locality-sensitive-hashing](https://github.com/singhj/locality-sensitive-hashing))

• **Datasketch** ([https://github.com/ekzhu/datasketch](https://github.com/ekzhu/datasketch))

• **TarsosLSH** ([https://github.com/JorenSix/TarsosLSH](https://github.com/JorenSix/TarsosLSH))

• **E2LSH** ([https://github.com/JorenSix/TarsosLSH](https://github.com/JorenSix/TarsosLSH))

• Many others
Some LSH papers

- Similarity Search in High Dimensions via Hashing
- Locality-Preserving Hashing in Multidimensional Spaces
- Approximate Nearest Neighbors: Towards Removing the Curse of dimensionality
- Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions
- Fast Search in Hamming Space with Multi-Index Hashing
- b-Bit Minwise Hashing
- LSH forest: self-tuning indexes for similarity search
Challenges: Scaling up

- MultiMaps (1:n)
- Off-heap implementation (avoid garbage collection)
- Bigger than memory (e.g., using RAM + SSD disk space)
- Multi-threaded (fine-grain locks or non-blocking)
- Using primitives (avoid space overhead)
• Distributing hash tables in several machines
  
  hashTable[0] -> s1
  hashTable[1] -> s2
  hashTable[2] -> s3
  hashTable[3] -> s4
  hashTable[4] -> s5

• Partitioning keys (require to inform hashTable number)
  
  Keys [0 – 1,000,000] -> s1 (hashTable[0-4])
  Keys [1,000,000–2,000,000] -> s2 (hashTable[0-4])
  Keys [2,000,000–3,000,000] -> s3 (hashTable[0-4])
  Keys [3,000,000–4,000,000] -> s4 (hashTable[0-4])
Take outs - Engineering

- **Implementing** basic specific cases is **simple**
- **Being generic** is **not**
- **Scaling** requires **good engineering** and optimizations
- Take time to **experiment the best parameters** to your case
LSH Configurations
Links to used resources

- **Presentations:**
  - http://www.slideshare.net/j_singh/open-lsh-a-framework-for-locality-sensitive-hashing-45912645
  - http://www.slideshare.net/SparkSummit/locality-sensitive-hashing-by-spark (Uber on similar routes)
  - http://www.slideshare.net/SameeraHorawalavithana/locality-sensitive-hashing (Tree on taxonomy)
  - http://www.slideshare.net/DmitriySelivanov/finding-similar-items-in-high-dimensional-spaces-locality-sensitive-hashing
  - http://www.slideshare.net/jsuchal/minhashing-fast-similarity-search
  - http://www.slideshare.net/huitseeker/a-gentle-introduction-to-locality-sensitive-hashing-with-apache-spark

- **Blog posts:**

- **Videos:**
  - https://www.youtube.com/watch?v=dgH0NP8Qxa8
  - https://www.youtube.com/watch?v=bQAYY8INBxg
  - https://www.youtube.com/watch?v=Arni-zkqMBA
  - https://www.youtube.com/watch?v=t_8SpFVOI7A
  - https://www.youtube.com/watch?v=LqcwaW2YE_c
  - https://www.youtube.com/watch?v=Ha7_Vf2eZvQ
  - https://www.youtube.com/watch?v=Dkomk2wPaoc