Recommendation systems
and the LSH algorithm

Vinicius Vielmo Cogo
http://lasige.di.fc.ul.pt/~vielmo
vielmo@lasige.di.fc.ul.pt

12º SeTEIC – IFF São Vicente – August 22, 2018
Outline

Big Data

Locality-Sensitive Hashing
Big Data

Locality-Sensitive Hashing
World’s Data Capacity

Source: www.martinhilbert.net/worldinfocapacity
Trending Topic

Special theme: Big Data
Characteristics

5 Vs

• **Volume** (quantity or size)
• **Variety** (type and nature)
• **Velocity** (speed it is generated)
• **Variability** (inconsistency in datasets)
• **Veracity** (quality)
Big Data ...

• ... is relative

• ... is not defined by a specific number of TB, PB, EB

• ... is when it becomes big for you

• ... is when your solutions become inefficient

• ... is when traditional processing becomes impractical
Data Structures and Big Data

- **Traditional DSs** are subject to the **same problems**
  - e.g., lists, trees

- Requires ...
  
  ... **distributing your data** (e.g., YARN, Spark)
  or
  ... **using auxiliary data structures** (e.g., index, metadata)
  or
  ... **trading precision for feasibility and utility**
  (e.g., probabilistic approach)
Probabilistic Data Structures

... trading precision for feasibility and utility

Precision:
- Rounded values (e.g., float with reduced precision)
- Ranges instead of values (e.g., location, ages)
- False positives (e.g., system incorrectly triggers something)

Recall:
- False negatives (e.g., system fails to find any solution)
- Similar objects instead of the nearest (e.g., approximate solutions)
- Good path instead of the optimal (e.g., suboptimal)
Outline

Big Data

Locality-Sensitive Hashing
Locality Sensitive Hashing

LSH is an efficient algorithm to find similar objects using hashes
Recommendation Algorithms

Suggest something to users…
Recommendation Algorithms

Customers who bought this object also bought…

![Snake Oil](image1)
Price: 50€
Rating: 5/5

![Snake Oil](image2)
Price: 35€
Rating: 4/5

![Snake Oil](image3)
Price: 20€
Rating: 4/5
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?
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

A  B  C  D  E

P2

A  B  D  E  F

P3

A  V  X  Y  Z
Example: Music Recommendation

P1

P2

P3
Example: Music Recommendation

P1

A  B  C  D  E

P2

A  B  D  E  F
Example: Music Recommendation
Example: Music Recommendation

How to identify similar users?
Example: Music Recommendation

P1

A B C D E

Jaccard Distance*

P2

A B D E F

Similarity = \frac{|A \cap B|}{|A \cup B|}

P3

A V X Y Z

LSH

24/50
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

P1

P2

P3

\[
\frac{|P1 \cap P2|}{|P1 \cup P2|} = \frac{4}{6} = 0.667
\]

\[
\frac{|P2 \cap P3|}{|P2 \cup P3|} = \frac{1}{8} = 0.125
\]

\[
\frac{|P1 \cap P3|}{|P1 \cup P3|} = \frac{1}{8} = 0.125
\]
Problem

Millions of users that listen thousands different songs each
Problem

Users = Objects

Songs = Dimensions*

Millions of users that listen thousands different songs each
Problem

Comparing 60M users

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

Smart: $60M \times \log(60M) = 46M$ s = 14 years
Being more efficient
Does it solve efficiently similarity search with more than 10 dimensions?
Locality Sensitive Hashing

Locality Sensitive Hashing

Locality Sensitive Hashing

LSH is an efficient algorithm to find similar objects using hashes
So, imagine **millions of buckets**
Algorithm – Insert

Given an object (user): hash it, obtain a value, and place the object in this bucket
In a way that objects in the **same bucket** have **bigger probability** of being similar
Algorithm – Query

Calculate the **distance** between objects *within* the same *bucket* only
Components

$\text{LSH} = \text{Hash Function} + \text{Hash Tables}$

$\text{MinHash}^* + \text{MultiMap}[]$
MinHash

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

```
minHash(Object o):
    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 variable number of dimensions to a fixed configurable number
- Using the same order of hash functions is important to find similar objects
Example

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

<table>
<thead>
<tr>
<th>hashTable[0]</th>
<th>hashTable[1]</th>
<th>hashTable[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 12 5 [P1]</td>
<td>45 7 34 [P1]</td>
<td>12 76 87 [P1]</td>
</tr>
</tbody>
</table>

minHash(P1)

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

```
minHash(P2)
23 12 5 73 22 15 3 28 56 ...
```

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

```
23 12 5
73 22 15
3 28 56 ...
```

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)

Nearest neighbor  Near neighbors  Clustering
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
• Deduplicating similar genomic and quality portions to compress data
• Pointer + modifications

Entries: > 1B
Size: 75% from the best
Throughput: > 200MB/s reads
Available 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))
Challenges: Implementing

- Generic to any **object**
- Providing multiple hash function families (generic to all **distances**)
- Being **efficient** (space and time)
- Durability
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)
Challenges: Scaling out

- 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]`)
Final Remarks

- **Big Data** is real and requires efficient solutions
- **Probabilistic** algorithms are feasible and useful
- **Locality-Sensitive Hashing** → efficient similarity search
Final Remarks

• These algorithms are usually a step to something bigger

• What to do with them?

• Where are they useful?

• There are opportunities of enhancements and applications on them
# Objects and Dimensions

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Examples</th>
</tr>
</thead>
</table>
| 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: \((x, y, z)\)  
2D 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>$\text{len(intersection)}/\text{len(union)}$</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>
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