Algorithm Engineering for High-Dimensional Similarity Search Problems

Martin Aumüller

IT University of Copenhagen
Roadmap

01
Similarity Search in High-Dimensions: Setup/Experimental Approach

02
Survey of state-of-the-art Nearest Neighbor Search algorithms

03
Similarity Search on the GPU, in external memory, and in distributed settings
1. Similarity Search in High-Dimensions: Setup/Experimental Approach
$k$-Nearest Neighbor Problem

- **Preprocessing**: Build DS for set $S$ of $n$ data points
- **Task**: Given query point $q$, return $k$ closest points to $q$ in $S$
Nearest neighbor search on words

• GloVe: learning algorithm to find vector representations for words
• *GloVe.twitter* dataset: **1.2M words**, vectors trained from **2B tweets**, **100 dimensions**
• Semantically similar words: nearest neighbor search on vectors


https://nlp.stanford.edu/projects/glove/
GloVe Examples

$ grep -n "sicily" glove.twitter.27B.100d.txt
118340:sicily -0.43731 -1.1003 0.93183 0.13311 0.17207 ...

"sicily"
- sardinia
- tuscany
- dubrovnik
- liguria
- naples

"algorithm"
- algorithms
- optimization
- approximation
- iterative
- computation

"engineering"
- engineer
- accounting
- research
- science
- development
Basic Setup

• Data is described by **high-dimensional feature vectors**

• **Exact similarity search is difficult** in high dimensions

• data structures and algorithms suffer
  
  • **exponential dependence** on dimensionality
  
  • **in time, space, or both**
Why is Exact NN difficult?

• Choose $n$ random points from $N(0, 1/d)^d$, for large $d$

• Choose a random query point

• nearest and furthest neighbor basically at same distance
Performance on GloVe

Recall-Queries per second (1/s) tradeoff - up and to the right is better

10^10 queries
Difficulty measure for queries

• Given query $q$ and distances $r_1, \ldots, r_k$ to $k$ nearest neighbors, define

$$D(q) = -\left(\frac{1}{k} \sum \ln \frac{r_i}{r_k}\right)^{-1}$$

Based on the concept of local intrinsic dimensionality [Houle, 2013] and its MLE estimator [Amsaleg et al., 2015]
LID Distribution

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data Points</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT [9]</td>
<td>1 000 000</td>
<td>128</td>
</tr>
<tr>
<td>MNIST</td>
<td>65 000</td>
<td>784</td>
</tr>
<tr>
<td>Fashion-MNIST [19]</td>
<td>65 000</td>
<td>784</td>
</tr>
<tr>
<td>GLOVE [17]</td>
<td>1 183 514</td>
<td>100</td>
</tr>
<tr>
<td>GLOVE-2M [17]</td>
<td>2 196 018</td>
<td>300</td>
</tr>
<tr>
<td>GNEWS [16]</td>
<td>3 000 000</td>
<td>300</td>
</tr>
</tbody>
</table>
Results (GloVe, 10-NN, 1.2M points)

http://ann-benchmarks.com/sisap19/faiss-ivf.html
2. STATE-OF-THE-ART NEAREST NEIGHBOR SEARCH
General Pipeline

Index generates candidates

Brute-force search on candidates
Brute-force search

GloVe: 1.2 M points, inner product as distance measure

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$...$</th>
<th>$p_n$</th>
</tr>
</thead>
</table>

- 400 byte

```c
inline float dot_naive(const float* x, const float* y, int f) {
    float result = 0;
    for (int i = 0; i < f; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

Automatically SIMD vectorized with clang –O3: [https://godbolt.org/z/TJX68s](https://godbolt.org/z/TJX68s)

- 100ms per scan
- 4.2 GB/s throughput
- CPU-bound
Manual vectorization (256 bit registers)

\[
x \quad \cdots \quad \cdots
\]
\[
y \quad \cdots \quad \cdots
\]
\[
0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0
\]

- Parallel multiply
- Parallel add to result register
- Horizontal sum and cast to float

- 25 ms per query
- 16 GB/s
- 16.5 GB/s single-thread max on my laptop
- Memory-bound

https://gist.github.com/maumueller/720d0f71664bef694bd56b2aeff80b17
Brute-force on bit vectors

• Another popular distance measure is Hamming distance
  • Number of positions in which two bit strings differ

• Can be nicely packed into 64-bit words
• Hamming distance of two words is just bitcount of the XOR

```
inline uint64_t distance(const uint64_t* x, const uint64_t* y, int f) {
    uint64_t res = 0;
    for (int i = 0; i < f; i++) {
        res += __builtin_popcountll(x[i] ^ y[i]);
    }
    return res;
}
```

• 1.3 ms per query (128 bits)
• 6 GB/s throughput
Sketching to avoid distance computations

- Distance computations on bit vectors faster than Euclidean distance/inner product
- Their number can be reduced by storing compact sketch representations

Can distance computation be avoided?

Set $\tau$ such that with probability at least $1 - \varepsilon$ we don’t disregard point that could be among NN.

Easy to analyze: Sum of Bernoulli trials of $\Pr(X = 1) = f(dist(q, x))$

Sketch representation

$q$

1011100101

x

0101101011

At least $\tau$ collisions?

Yes

No

compute $dist(q, x)$

skip

SimHash [Charikar, 2002]
1-BitMinHash [König-Li, 2010]

[Christiani, 2019]
General Pipeline

Index generates candidates

Brute-force search on candidates
PUFFINN

PARAMETERLESS AND UNIVERSALLY FAST FINDING OF NEAREST NEIGHBORS

[A., Christiani, Pagh, Vesterli, 2019]

https://github.com/puffinn/puffinn

Credit: Richard Bartz
How does it work?

Locality-Sensitive Hashing (LSH) [Indyk-Motwani, 1998]

A family $\mathcal{H}$ of hash functions is **locality-sensitive**, if the collision probability of two points is decreasing with their distance to each other.

$h(p) = h_1(p) \circ h_2(p) \circ h_3(p) \in \{0,1\}^3$
Solving $k$-NN using LSH (with failure prob. $\delta$)

Termination: If $(1 - p)^j \leq \delta$, report current top-$k$.

Not terminated? Decrease $K$!
The Data Structure

Theoretical
• **LSH Forest**: Each repetition is a Trie build from LSH hash values
  [Bawa et al., 2005]

Practical
• Store indices of data set points sorted by hash code
• “Traversing the Trie” by binary search
• use lookup table for first levels
Overall System Design

Hashing

- \( q \)
- \( h_1 \)
- \( h_2 \)
- \( \vdots \)
- \( h_L \)

 Filtering

- 1
- \( s_1(p_1) \)
- \( \cdots \)
- \( s_1(p_n) \)
- \( s_M(p_1) \)
- \( \cdots \)
- \( s_M(p_n) \)

- dist(\( s'(p), s'(q) \)) \leq \tau ?

- insert into buffer
- compute distance

 Accumulation

- top-\( k \)
- buffer

- update top-\( k \) if buffer is full
- deduplicate

- \( p_1 \)
- \( \cdots \)
- \( p_n \)

- \( S \)
Running time (Glove 100d, 1.2M, 10-NN)
A difficult (?) data set in $\mathbb{R}^{3d}$

$n$ data points

\[ x_1 = (0^d, y_1, z_1) \]
\[ \vdots \]
\[ x_{n-1} = (0^d, y_{n-1}, z_{n-1}) \]
\[ x_n = (v, w, 0^d) \]

$m$ query points

\[ q_1 = (v, 0^d, r_1) \]
\[ \vdots \]
\[ q_m = (v, 0^d, r_m) \]

$y_i, z_i, v, w, r_i \sim \mathcal{N}^d \left( 0, \frac{1}{2d} \right)$
Running time ("Difficult", 1M, 10-NN)
Graph-based Similarity Search
Building a Small World Graph
Refining a Small World Graph

**Goal**: Keep out-degree as small as possible (while maintaining “large-enough” in-degree)!

HNSW/ONNG: [Malkov et al., 2020], [Iwasaki et al., 2018]
Running time (Glove 100d, 1.2M, 10-NN)
Open Problems Nearest Neighbor Search

• Data-dependent LSH with guarantees?
• Theoretical sound Small-World Graphs?
• Multi-core implementations
  • Good? [Malkov et al., 2020]
• Alternative ways of sketching data?
3. Similarity Search on the GPU, in External Memory, and in Distributed Settings
Nearest Neighbors on the GPU: FAISS
[Johnson et al., 2017] https://github.com/facebookresearch/faiss

• GPU setting
  • Data structure is held in GPU memory
  • Queries come in batches of say 10,000 queries per time

• Results:
  • http://ann-benchmarks.com/sift-128-euclidean_10_euclidean-batch.html
FAISS/2

• Data structure
  • Run k-means with large number of centroids
  • Each data point is associated with closest centroid

• Query
  • Find \( L \) closest centroids
  • Return \( k \) closest points found in points associated with these centroids

Nearest Neighbors on the GPU: GGNN

[Groh et al., 2019]
Nearest Neighbors in External Memory
[Subramanya et al., 2019]

RAM

\[ \hat{x}_1 \]

\[ \vdots \]

\[ \hat{x}_n \]

Compressed vectors (32 byte per vector)

Product Quantization

SSD

\[ x_1 \]

\[ \dotsc \]

\[ x_n \]

Original vectors (~400 byte per vector)
Distributed Setting: Similarity Join

• Problem
  • given sets $R$ and $S$ of size $n$,
  • and similarity threshold $\lambda$, compute
  $R \bowtie_\lambda S = \{(x, y) \in R \times S \mid sim(x, y) \geq \lambda\}$

• Similarity measures
  • Jaccard similarity
  • Cosine similarity

• Naive: $O(n^2)$ distance computations
Map-Reduce-based Similarity Join

Single Core on Xeon E5-2630v2 (2.60 GHz)  
Hadoop cluster (12 nodes, 24 HT per node)

Scalability! But at what COST? [McSherry et al., 2015]

[LIVEJ] PPJ 345  
PJL 88.9  
PJL 12.0  
PJL 6.52  
PJL 3.50  
PJL 1.88  
PJL 1.02  
PJL  

[NETFLIX] ALL 123  
PJL 494  
PJL 146  
PJL 76.4  
PJL 36.6  
PJL 15.6  
PJL 4.73  
PJL 0.894  
PJL  

[ORKUT] PPJ 213  
PJL 79.4  
PJL 33.4  
PJL 21.0  
PJL 12.9  
PJL 7.69  
PJL 4.28  
PJL 2.06  
PJL  

[LIVE] 313  
VJ 285  
VJ 278  
VJ 254  
VJ 243  

[NETF] T  
T 527  
VJ 215  
VJ 161  
VJ  

[ORKU] T 1592  
MG 941  
VJ 761  
VJ 681  
VJ  

[Mann et al., 2016]  
[Fier et al., 2018]
Solved almost-optimally in the MPC model

[Hu et al., 2019]

\[ (x, h_i(x)) \]

\[ (y, h_i(y)) \]

\[ (x, y, h_i(x)) \]

\[ (x, y) \]

\[ R \]

\[ S \]

Hash using LSH

Join on hash

Similarity at least \( \lambda \)?

Emit

\( O(n^2) \) local work for distance computations!
Another approach: DANNY


Implementation in Rust using timely dataflow

https://github.com/TimelyDataflow/timely-dataflow

LSH + Sketching, candidate verification locally
Results
Roadmap

01  Similarity Search in High-Dimensions: Setup/Experimental Approach

02  Survey of state-of-the-art Nearest Neighbor Search algorithms

03  Similarity Search on the GPU, in external memory, and in distributed settings
References

- [Christiani, 2019]: Fast locality-sensitive hashing frameworks for approximate near neighbor search
- [Houle, 2013]: Dimensionality, discriminability, density and distance distributions. ICDMW 2013.
- [Iwasaki, Miyazaki, 2018]: Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data, https://arxiv.org/abs/1810.07355
- [Subramanya, Devvrit, Kadekodi, Krishnaswamy, Simhadri, 2019]: DiskANN: Fast accurate billion-point nearest neighbor search on a single node. NeurIPS 2019
Extra slides
PUFFINN: Fast Hash Function Evaluation

- **Main Bottleneck:** Computation of Hash Values
- Adapt the “pooling” technique of [Dahlgaard et al., 2017] and [Christiani, 2019]

\[ K \cdot m \text{ independent hash functions from LSH family, } m \ll L. \]

Pick \( K \) hash functions in repetition \( j \) using universal hash functions in each column.

Analysis using Cantelli’s inequality → Requires different stopping criteria (factor 2 slowdown)