Locality Sensitive Hashing

Clément King

15 août 2017

Summary

Problem and Motivations (1)

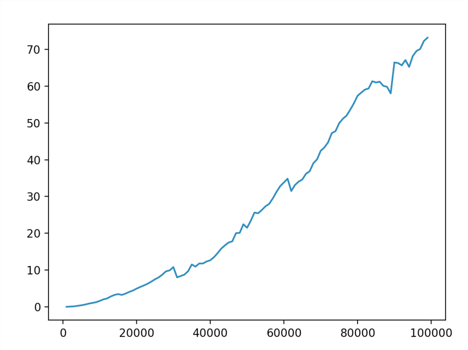
Nearest Neighbor (NN) problem : given a collection of n points, build a data structure which, given a query point, reports the point that is the closest to the queried one.

In general points are represented as vectors in $\mathbb{R}^d$.

Issue : Naive solutions suffer from the "curse of dimensionality". Brute force linear search is too long for big database. To compute all the distances between n points, one needs to compute $\frac{n(n-1)}{2}$ distances.

Problem and Motivation (2)

y-axis : computation time, x-axis : number of points



Alternative to brute force ? Near Neighbor search !

Summary

Locality Sensitive Hashing

The algorithm relies on the existence of hash functions (function that map data of arbitrary size to a fixed size subspace).

Definition : A family $\mathbb{H}$ is said (R, cR, p_1, p_2) - sensitive if for any points p and q in $\mathbb{R}^d$, and any hash function h in $\mathbb{H}$:

- $P_{h\in\mathbb{H}}(h(p) = h(q)) \geq p_1$ if $d(p, q) \leq R$
- $P_{h\in\mathbb{H}}(h(p) = h(q)) \leq p_2$ if $d(p, q) \geq cR$

where $c > 1$, $p_1 > p_2$, R is some radius and d is some distance in $\mathbb{R}^d$.

What does that mean ? We want the probability of collision (p_1) to be higher for objects that are close to each other $(d(p, q) \leq R)$ than for those that are far apart $(d(p, q) \geq cR)$.

Example

Suppose our data points are binary (10001, 11011, ...), and let d be the Hamming distance (i.e $d(101, 100) = 1$, the number of bits that are different).

Now, if $\mathbb{H} = \{h_i : \{0,1\}^d \rightarrow \{0,1\} | h_i(p) = p_i \}$, choosing a random h in if $\mathbb{H}$ means that $h(p)$ returns a random coordinate of $p \in \mathbb{R}^d$.

Is that a "sensitive" family ? We have that $P_{h \in \mathbb{H}}(h(q) = h(q))$ is equal to the number of coordinates on which p and q agree divided by the total number of coordinates.

Hence $p_1 = 1-$ proportion of bits that differ $= 1 - \frac{R}{d}$ and similarly, $p_2 = 1 - \frac{cR}{d}$.

$\Rightarrow$ As long as $c > 1$, $p_1 > p_2$.

Summary

Amplification (1)

Since p_1 and p_2 may be close to each other, we cannot use a hash function by itself.

Consider k and $B \in \mathbb{N}$, and define :

$g_i(q) = (h_{1,i}(q), ..., h_{k,i(q)})$ for all $1 \leq i \leq B$,

where each $h_{j,i}$ is chosen independently in $\mathbb{H}$.

$g_i : \mathbb{R}^d \to \mathbb{Z}^k$ (we created a hash function using k random hash functions) and so each query point q is placed into B different buckets.

To process a query point q we look at the B buckets q is in and compute the distances between the points of those buckets and q.

Amplification (2)

Two strategies :

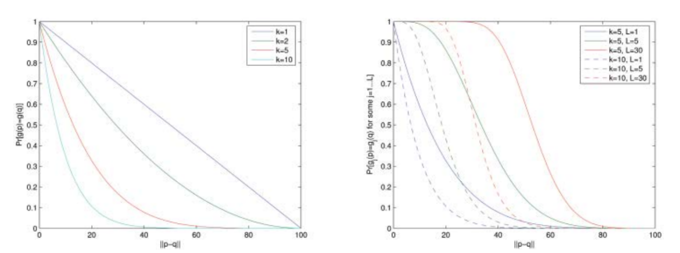
- Stopping the search in the buckets after scanning at most $3B$ collisions (see Indyk-Motwani).
- Scan all the collisions (that's what we do) : Let p be any point in the $R-\textit{neighborhood}$ of q.
  For any g_i, $P(g_i(p) = g_i(q)) \geq p_1^k$ and so,
  $P(g_i(p) = g_i(q)$ for some i)
  $= P(\exists$ i such that $g_i(p) = g_i(q))$
  $= 1 - P(\exists!$ i such that $g_i(p) = g_i(q))$
  $= 1 - P(\{g_i(p) \neq g_i(q)\}$ for all i)
  $\geq 1 - (1 - p_1^k)^B$

Algorithm

- Init
  - Choose B functions $g_i : \mathbb{R}^d \to \mathbb{Z}^k$
  - Construct B hash tables where the i^{th} table contains the points hashed with the $g_i^t h$ function
- Algo
  - Input : a query point q
  - For $i = 1, ..., B$
    - Retrieve points from the bucket $g_i(q)$ in the i^{th} hash table
    - Compute the distances between all the points and q

Trade-off

Large values of k lead to larger gap between p_1 and p_2 which causes the hashing to be selective (good !). However, if k is large p_1^k is small (very bad ! !), and so we need a big B to compensate.



Left : proba that $g_i(p) = g_i(q)$ for some i for different values of k (the larger k is, the more we partition our space)

Right : proba that $g_i(p) = g_i(q)$ for any i for different values of k (the larger B is, the more buckets we're scanning).

Summary

Random Binary Projections (1)

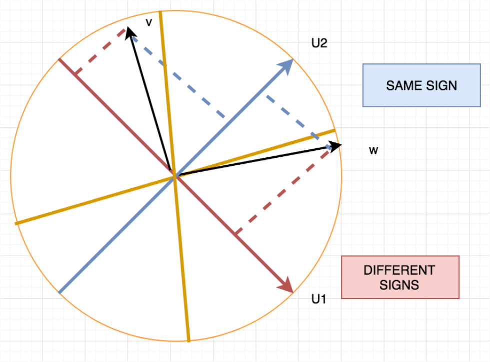
Let our space be the unit sphere in $\mathbb{R}^d$, say $\mathbb{S}$.

- For i from 1 to k
  - Choose a unit vector u_i at random in $\mathbb{S}$
  - For each point v we have, $g_i(v) = sgn(u_i \cdot v)$.

We get a signature matrix M of size k x (number of points v) such that $M(i, v) = sgn(u_i \cdot v)_{i \leq k}$.

Hence, the candidates to be nearest neighbors of a given input vector w are the vectors for which the column of the matrix are identical to the hash representation of w.

Random Binary Projection (2)



It happens that $P(sgn(u \cdot v) = sgn(u \cdot w)) = 1 - \frac{arccos(v \cdot w)}{\pi}$

Summary

Code(1)

```
# https://github.com/pixelogik/NearPy
from nearpy import Engine
from nearpy.distances import CosineDistance
from nearpy.hashes import RandomBinaryProjections

# Let matrix be the scoring vector matrix, i.e each line
# corresponds to a score vector

lsh = RandomBinaryProjections("rbp0", nb_random_vectors)
engine = Engine(matrix.shape[1],
    lshashes=[lsh], distance=CosineDistance())
```

Code (2)

```
mapping = defaultdict(list)
for i in xrange(matrix.shape[0]):
    scoring_vector = matrix[i, :]
    hash = lsh.hash_vector(scoring_vector[0])
    mapping[hash_v].append(i)

# output is like:
# {1001: [5785, 485795, 585, ...],
#  0010: [ 27402, 1957, ...],
#  ... }
```

Bibliography

Andoni, A., Indyk, P., *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions*

Indyk, P., Motwani, R., ANN : Towards removing the curse of dimensionality