High-dimensional visual similarity search: k-d Generalized Randomized Forests

Y. Avrithis, I. Z. Emiris, G. Samaras

National & Kapodistrian University of Athens

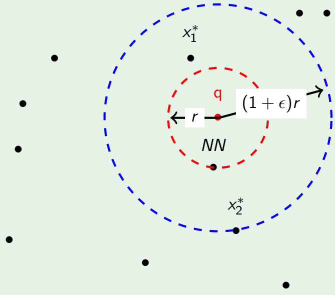
CGI '16, Heraklion, Greece

Problem definition

Definition (approximate-Nearest Neighbour Search)

Given a finite dataset $X \subset R^d$ *and real* $\epsilon > 0$, $x^* \in X$ *is an* ϵ*-approximate nearest neighbour of query* $q \in R^d$, *if* $dist(q, x^*) \leq (1+\epsilon)dist(q, x)$ *for all* $x \in X$. *For* $\epsilon = 0$, *this reduces to exact NNS.*

Example



Here, dist(q, NN) $= r$, but we may return points lying in the $(1+\epsilon)r$ circle, i.e. x_1^* or x_2^*, which are not exact NNs of the query point q, but *approximate* ones.

Brute force search: $O(n)$, where $n = |X|$.

Previous Work

Software

- Balanced Box Decomposition (BBD) tree [Mount]. Subdivides space into axis-aligned hyper rectangles.
- *k*-d trees, forming small forests, up to 6 trees with one point per leaf. FLANN [Muja et al].
- Locality Sensitive Hashing (LSH) [Indyk et al]. Hash the points, so that the similar lie into the same bucket.

Encodings

- SIFT is a 128-dimensional vector that describes a local image patch by histograms of local gradient orientations. [Lowe]
- GIST is a 960-dimensional vector that describes globally an entire image. [Oliva et al]
- CroW is a $\{128, 256, 512\}$-dimensional vector that is derived after such cross-dimensional weighting and pooling, utilizing deep learning methods. That vector is an image feature. [Kalantidis et al]

k-d GeRaF(kd-Generalized Randomized Forests)

Contribution

- Forest of randomized k-d trees (overcome limitations of one k-d tree)
- Every tree has its own structure (independent overall search)
- Several parameters, manual or auto configuration
- Simultaneous search, with no backtracking. Nodes from all trees are visited in an order.
- Accelerated/caching of distance computations
- Public domain C++ software and WebApp (no install required)
- Good performance (fastest building) and scalability

Randomization

Techniques

Rotation

Every tree is based on a rotated pointset, thus based on a different set of dimensions.

Split dimension

Pick the t dimensions of highest variance. Choose one randomly at every node while building the trees.

Split Value

Equal to the median of X in split dimension plus δ, a term uniformly distributed in $[\frac{-3\Delta}{\sqrt{d}}, \frac{3\Delta}{\sqrt{d}}]$, where Δ is the diameter of the current pointset.

Shuffling

The split value may be witnessed in several points, instead of picking always the same point, shuffle them to break ties.

Algorithm 1: *k*-d GeRaF: building

```
input  : pointset X, #trees m, #split-dimensions t, max #points per leaf
         p
output: randomized k-d forest F
begin
    V ← ⟨VARIANCE of X in every dimension⟩
    D ← ⟨t dimensions of maximum variance V⟩
    F ← ∅                                              ▷ forest
    for i ← 1 to m do
        f ← ⟨random transformation⟩                    ▷ rotation, shuffling
        F ← F ∪ (f, BUILD(f(X)))                       ▷ build on transformed X, store f
    return F

function BUILD(X)                                      ▷ recursively build tree (node/leaf)
    if |X| ≤ p then                                    ▷ termination reached
        return leaf(X)
    else                                               ▷ split points and recurse
        s ← ⟨one of dimensions D at random⟩
        v ← ⟨MEDIAN of X in dimension s⟩
        (L, R) ← ⟨SPLIT of X in dimension s at value v⟩
        return node(c, v, BUILD(L), BUILD(R))          ▷ build children on L, R
```

Searching

- Descend every tree, store branch in min-priority queue Q
- Descend again, starting from each node in Q, until $\frac{c}{1+\epsilon}$ leaves are checked, where c is a user parameter
- On descend of the $i - th$ tree:
- ...if leaf, insert distance(point, query) into a min-heap
- ...if node, check if query is in the negative half-space of node
- .... insert right child to Q and descend to left child
- .... vice versa if query in positive half-space

Implementation

Parameters

- m Number of trees in forest (points are stored once)
- t Number of dimensions used for splits
- p Maximum number of points per leaf
- c Maximum number of leaves to be checked during search
- ϵ Determine search accuracy
- k Number of Neighbours to be returned

Remarks

- Quickselect algorithm to find median in $O(n)$
- Reduce distance computation to dot product
- Parallel execution for building process

Building

ϵ	0	0.1	0.5	0.9
BBD	187.5	184.3	185.1	185.6
LSH	1.47	69.76	48.47	14.35
FLANN	244.6	217.2	157.3	142.0
GeRaF	8.167	8.567	8.579	8.565

Table : Build time for MNIST; $n = 60\text{k}, d = 784$

Search Oxford, 5062 images, dim = 512 (CroW features)

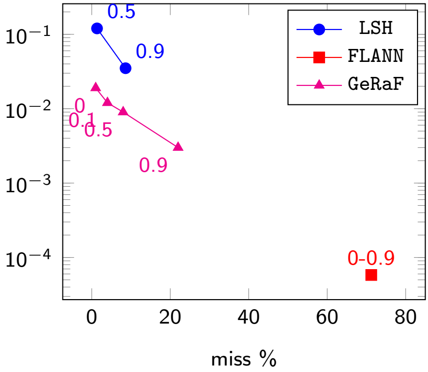
Brute force took 5.22 seconds. Build takes 2 sec with `GeRaF`.

points_per_leaf	*trees_no*	*t*	*max_leaf_check*	*miss*(%)	*time*(*milli*)
1	1	4	2	4	0.2
1	1	4	4	0	0.3
1	4	4	4	0	0.5

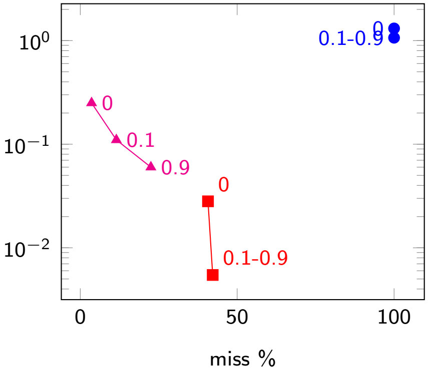
Table : "Noisy" queries

points_per_leaf	*trees_no*	*t*	*max_leaf_check*	*miss*(%)	*time*(*sec*)
16	8	32	32	3.6	0.01
16	32	64	64	0	0.03
16	64	64	4	0	0.02

Table : Oxford queries



(c) SIFT $n = 10^6, d = 128$



(d) GIST $n = 10^6, d = 960$

Figure : Search accuracy (miss rates) and runtimes (sec) on real datasets. Numbers over points are the values of ϵ. In both cases, BBD is out of memory and FLANN does not preprocess after 4 hr for any ϵ, thus we configured its parameters manually.

Conclusion

- Efficient implementation, `GeRaF`, competitive against state-of-the-art methods.
- Automatic parameter configuration that yields the fastest preprocessing, including both configuration and building, as well as a successful trade-off between accuracy and speed.
- Most competing methods have difficulties, namely they suffer from running out of memory at large scale, slow or non-terminating parameter configuration, or unstable search behaviour.