Fast Nearest Neighbor Methods

Thomas Breuel

UniKL

```
1 import cv2
```

Fast Nearest Neighbor and Range Search Methods

Fast Nearest Neighbor Methods

practical:

- FLANN

approaches:

- k-D tree (and R-trees)
- locality sensitive hashing
- tree VQ
- Hamming embedding

Nearest Neighbor vs Range Query

Database:

- collection of points $\{x_1, ..., x_N\}$
- preprocessing allowed

Range Query:

- given a query x and a range ϵ , find $\{i | d(x, x_i)\epsilon\}$

Nearest Neighbor Query:

- given a query x and a k , find the k nearest neighbors of x according to some metric d

Dimensionality Problem

As we saw at the beginning, in high dimensions, all distances become very similar to each other *for data with an intrinsic high dimension*.

Finding the exact nearest neighbor reduces to linear search for all known algorithms.

Solutions to Dimensionality Problem

- perform dimensionality reduction, then perform nearest neighbor search in lower-dimensional space
- use a nearest neighbor search method that takes advantage of low intrinsic dimensionality

Purpose of Nearest Neighbor

Almost always, the purpose of k -nearest neighbor searches is *classification*.

The classification problem may be implicit or dynamic, but it still relies on the relationship between nonparametric density estimation and nearest neighbors.

Methods for Nearest Neighbor Searches

Linear Search

- just compute $d(x, x_i)$ for every x_i in the database
- takes $O(Nd)$ time

possible speedups:

- reduce dimensionality with PCA
- use $||\cdot||_1$ instead of $||\cdot||_2$ (no multiplication)
- stop accumulation inside $||\cdot||$ when a value larger than the best distance so far has been found
- take advantage of sparsity when available

Range Query with Infinity Metric

In the ∞ -metric, the distance between two points is the maximum distance in any coordinate.

For a range query, we require that every coordinate of a match is within the given ϵ of the query point.

We can execute this query coordinate by coordinate, even in a relational database:

select * from t where abs(t.x1-1.7)¡0.1 and abs(t.x2-3.9)¡0.1 and abs(t.x3-0.2)¡1

Internally, this can be executed using inverted sorted indexes and intersections.

kD-Trees

kD-Tree

construction:

- given the point set, cycle through the axes
- for each axis, find a splitting point by selecting the median of the points along the axis
- recurse on each side of the splitting point until a minimum size is reached

retrieval:

- find the leaf containing the query point x , find the nearest neighbor y , let $r = d(y, x)$ and return that
- on return from a child, check the sibling node
- if the node does not overlap the ball $B_r(x)$, return y, r
- if the node does overlap the ball $B_r(x)$, search it for a better neighbor
- return the best y, r found in both child nodes

kD-tree complexity

- asymptotic complexity for nearest neighbor queries is $O(\log N)$ (great!)
- asymptotic complexity is only reached for $N >> 2^k$ (not so great)

In practice, kD-trees do not work for searching in high dimensions.

Locality Sensitive Hashing

Locality Sensitive Hashing

Assume a family of hash functions F , a distance d , a distance threshold $R > 0$, and an approximation factor $c > 1$.

For every $h \in F$, and with $p_1 p_2$:

- If $d(x, y)R$ then $P(h(p) = h(q))p_1$
- If $d(x, y)cR$ then $P(h(p) = h(q))p_2$

We call this an (R, cR, p_1, p_2) -sensitive family of hash functions.

Amplification

AND-construction:

Given a (d_1, d_2, p_1, p_2) -sensitive family of hash functions, choose k random hash functions $h_1, ..., h_k$ and define a new hash function h that has to agree on all of them. This gives rise to a (d_1, d_2, p_1^r, p_2^r) -sensitive family.

OR-construction:

Given a (d_1, d_2, p_1, p_2) -sensitive family of hash functions, choose k random hash functions $h_1, ..., h_k$ and define a new hash function h that has to agree on any of them. This gives rise to a $(d_1, d_2, 1-(1-p_1)^r, 1-(1-p_2)^r)$ -sensitive family.

Approximate nearest neighbor search using LSH

Assume a locality sensitive family F of hash functions, and parameters k and L .

During preprocessing:

- construct new hash functions g by concatenating k hash functions from h
- hash each point into L hash tables using g

During query:

- try to look up the query point x in each of the L hash tables
- for each bucket found in a hash table, return the first point for which the distance is less than cR
- if none is found move to the next hash table
- if all hash tables are exhausted, fail

Complexity:

$\rho = \log P_2 / \log P_1$, assuming constant time hash functions; finds approximate nearest neighbors.

Which random hash functions?

- bit sampling (sample bits from the coordinates)
- quantization of coordinates
- random projections

FLANN

Hierarchical k-Means

Construction:

- take the original data points, perform k -means clustering
- assign each data point to its cluster
- recursively repeat on each cluster until a minimum cluster size has been reached

Lookup:

- traverse the tree in a best-bin-first manner
- maintain a priority queue of branches with closest centers to the query point
- stop the search after a pre-determined number of nodes have been examined

Notes:

- hierarchical k-Means is usually called tree-VQ (hierarchical VQ is multiple resolutions)
- smarter exploration on search doesn't seem to help

```
1 data = array(randn(1000,10),'f')
```

```
1 flann = cv2.flann_Index(data,dict(algorithm=1,trees=4))
```

```
1 index,dist = flann.knnSearch(data,5,params={})
```

```
1 print index[:10]
```

```
[[  0 638 899 940 525]
 [  1 944 749 230 497]
 [  2 430 409 933 928]
 [  3  79 248 788 951]
 [  4 103  58 507 480]
...
 [  6 588 981 990  28]
 [  7 361 780 722 107]
 [  8 722 707 666 685]
 [  9 287 163 125 272]]
```

```
1 print dist[:10]
```

```
[[  0.          2.46911836   2.62438083   3.03617215   3.18475151]
 [  0.          4.27013683   4.83362865   6.21951389   6.43545055]
 [  0.          4.56065941   4.59481716   8.74168491   9.18355083]
 [  0.          5.56775475   6.28863955   6.48139763   6.63649178]
 [  0.          4.99117756   5.16120672   5.90600395   6.00450516]
...
 [  0.          4.98363495   5.5444622    5.70466614   6.22260761]
 [  0.          9.51819229  11.93401909  12.31844997  12.35668182]
 [  0.          4.21605206   4.53588057   4.59677792   4.67584229]
 [  0.          4.1922102    4.80881023   4.81959963   5.81781578]]
```