Permutation Search Methods are Efficient, Yet Faster Search is Possible

Bileg (Bilegsaikhan) Naidan[1]   Leo (Leonid) Boytsov[2]   Eric Nyberg[2]

[1] Norwegian University of Science and Technology (NTNU)

[2] Carnegie Mellon University (CMU)

https://github.com/searchivarius/NonMetricSpaceLib

Nearest-neighbor search (NN-search)

Nearest-neighbor search (NN-search)

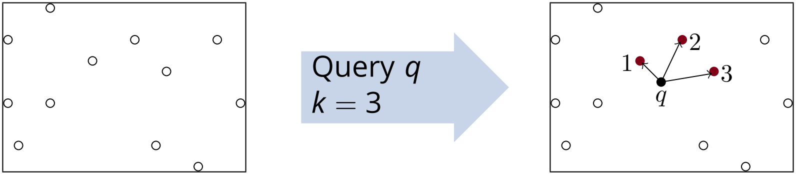
- **Input:** A set of n objects and a distance function $d(x, y)$

Nearest-neighbor search (NN-search)

- **Input:** A set of n objects and a distance function $d(x, y)$
- **Query:** New object q and k

Nearest-neighbor search (NN-search)

- **Input:** A set of n objects and a distance function $d(x, y)$
- **Query:** New object q and k
- **Task:** Quickly find k most similar objects in the database to q



Distance function

Name	$d(x,y)$	Symmetry	Triangle ineq.
Euclidean (L_2)	$\sqrt{\sum(x_i - y_i)^2}$	✓	✓
Cosine distance	$1 - \frac{x \cdot y}{\lvert x\rvert \lvert y\rvert}$	✓	✗
KL-diverg.	$\sum x_i \log \frac{x_i}{y_i}$	✗	✗
JS-diverg.	symmetrized & smoothed KL-diverg.	✓	✗

Distance functions can be **metric** or **non-metric**

How to find similar objects?

How to find similar objects?

- **Brute-force**
  - Exact search
  - Slow: *n* distance computations

How to find similar objects?

- **Brute-force**
  - Exact search
  - Slow: n distance computations
- **Indexing**
  - Exact search is mostly slow in high-dimensions and/or non-metric spaces: $O(n)$ distance computations
  - **Approximate** search can be fast

State-of-the-art approximate search methods

- Locality Sensitivity Hashing (LSH)
- VP-tree/ball-tree (data-dependent tuning)
- Proximity graphs (kNN-graphs)
- **Permutation methods**

Why should we care about permutation methods?

Why should we care about permutation methods?

- Promising **universal** methods for non-metric spaces

Why should we care about permutation methods?

- Promising **universal** methods for non-metric spaces
- Mapping data from “hard” spaces to “easy” spaces (the Euclidean space)

Why should we care about permutation methods?

- Promising **universal** methods for non-metric spaces
- Mapping data from “hard” spaces to “easy” spaces (the Euclidean space)
- **Database-friendly** methods that are easy to implement on top of a database system or Lucene

Research questions

Research questions

- How good are permutation-based projections?

Research questions

- How good are permutation-based projections?
- How well do permutation methods fare against state of the art?

Permutation Methods

- **Filter-and-refine** methods using **pivot-based projection** to the permutation space (L_1 or L_2)

Permutation Methods

- **Filter-and-refine** methods using **pivot-based projection** to the permutation space (L_1 or L_2)
- Select randomly a set of reference points called **pivots**

Permutation Methods

- **Filter-and-refine** methods using **pivot-based projection** to the permutation space (L_1 or L_2)
- Select randomly a set of reference points called **pivots**
- Order **pivots** by their distances to data points to obtain pivot rankings, which we call **permutations**

Permutation Methods

- **Filter-and-refine** methods using **pivot-based projection** to the permutation space (L_1 or L_2)
- Select randomly a set of reference points called **pivots**
- Order **pivots** by their distances to data points to obtain pivot rankings, which we call **permutations**
- **Filter** by comparing **permutations** to obtain candidate points

Permutation Methods

- **Filter-and-refine** methods using **pivot-based projection** to the permutation space (L_1 or L_2)
- Select randomly a set of reference points called **pivots**
- Order **pivots** by their distances to data points to obtain pivot rankings, which we call **permutations**
- **Filter** by comparing **permutations** to obtain candidate points
- **Refine** by comparing candidate points to the query

Permutation Methods

How do we carry out the filtering step?

Permutation Methods

How do we carry out the filtering step?

- Brute force searching

Permutation Methods

How do we carry out the filtering step?

- Brute force searching
- Indexing of permutations

Permutation Methods

How do we carry out the filtering step?

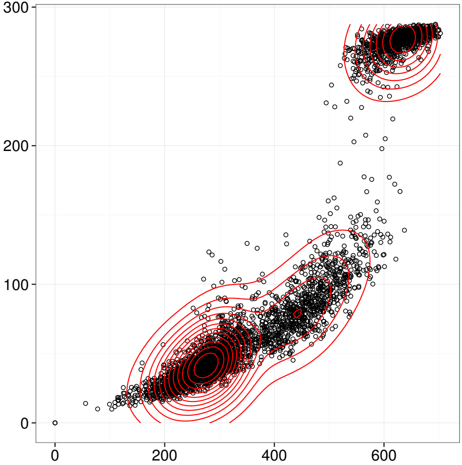
- Brute force searching
- Indexing of permutations
  - Neighborhood APProximation Index (NAPP) is the best approach

Experiments: Datasets

Name	Distance function	Number of points	Brute-force (sec.)	Dimens.
		Metric Data		
CoPhIR	L_2	$5 \cdot 10^6$	0.6	282
SIFT	L_2	$5 \cdot 10^6$	0.3	128
ImageNet	SQFD	$1 \cdot 10^6$	**4.1**	N/A
		Non-Metric Data		
Wiki-sparse	Cosine sim.	$4 \cdot 10^6$	1.9	10^5
Wiki-8	KL-div/JS-div	$2 \cdot 10^6$	0.045/0.28	8
Wiki-128	KL-div/JS-div	$2 \cdot 10^6$	0.22/**4**	128
DNA	Norm. Leven.	$1 \cdot 10^6$	**3.5**	N/A

Experiments: Projection Quality

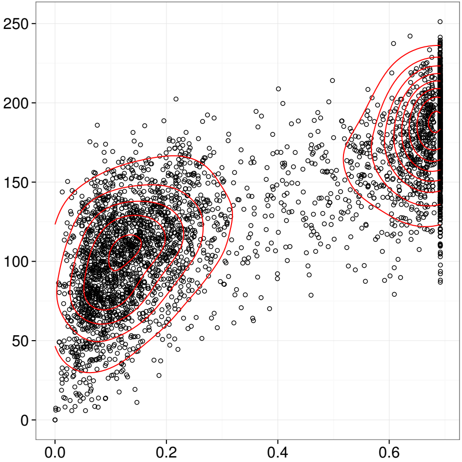
Distance in the original space vs. distance in the projected space.
The closer to a **monotonic** mapping, the **better**:



Good projection (original distance: L_2)

Experiments: Projection Quality

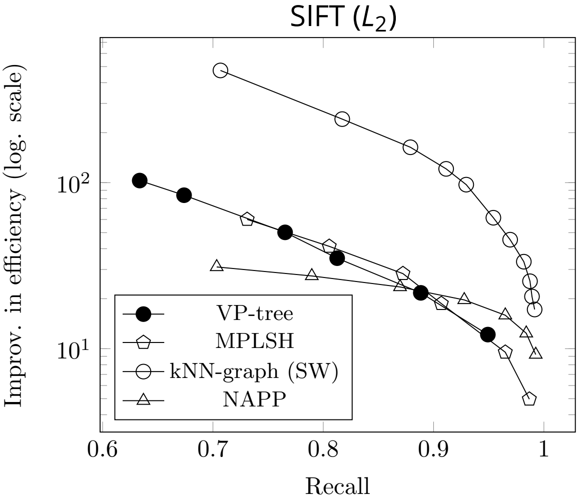
Distance in the original space vs. distance in the projected space.
The closer to a monotonic mapping, the **better**:



Bad projection (original distance: JS-div.)

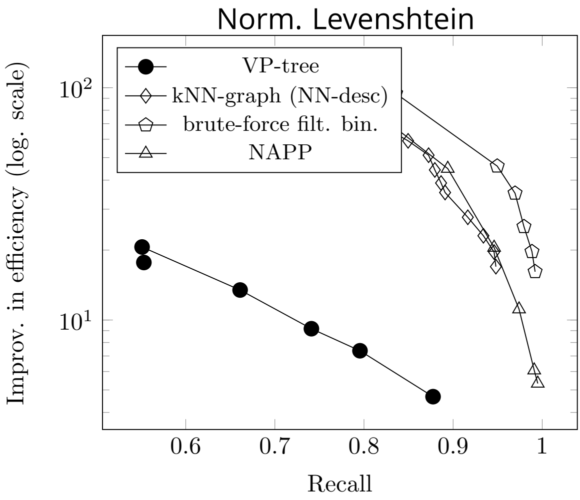
Experiments: Efficiency vs Accuracy

Improvement in efficiency over brute-force search vs. accuracy.
Higher and to the right is **better**:



Experiments: Efficiency vs Accuracy

Improvement in efficiency over brute-force search vs. accuracy.
Higher and to the right is **better**:



Conclusions

- Permutation methods beat state-of-the-art methods (VP-trees, kNN-graphs and Multiprobe LSH) for **some data sets**, in particular, when the distance function is expensive

Conclusions

- Permutation methods beat state-of-the-art methods (VP-trees, kNN-graphs and Multiprobe LSH) for **some data sets**, in particular, when the distance function is expensive
- The quality of permutation-based projection **can be both good and poor**: it appears to be better when the space is metric and/or dimensionality is low

Poster Session Discussion Points

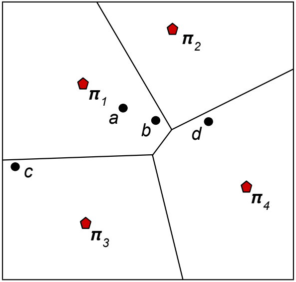
What makes a good, amenable, non-metric space?

Thank you for your attention!

Some technical details

Permutation Methods

The data points are *a*, *b*, *c*, *d* in 2-dim. Euclidean space (L_2).
The Voronoi diagram produced by 4 pivots π_i.



Point	Pivot Order	Permutations
a	$(\pi_1, \pi_2, \pi_3, \pi_4)$	$(1, 2, 3, 4)$
b	$(\pi_1, \pi_2, \pi_4, \pi_3)$	$(1, 2, 4, 3)$
c	$(\pi_3, \pi_1, \pi_2, \pi_4)$	$(2, 3, 1, 4)$
d	$(\pi_4, \pi_2, \pi_1, \pi_3)$	$(3, 2, 4, 1)$

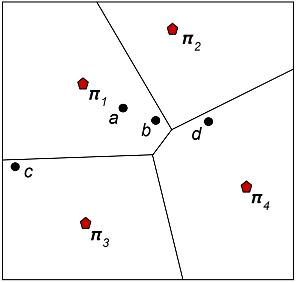
Similar

Position of π_4 is 1

Permutation Methods

Permutation is
a fancy word for
a pivot ranking!

The data points are *a*, *b*, *c*, *d* in …).
The Voronoi diagram produced by 4 pivots π_i.



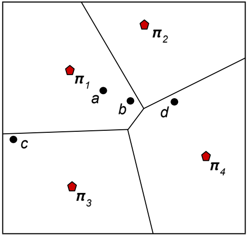
Point	Pivot Order	Permutations
a	$(\pi_1, \pi_2, \pi_3, \pi_4)$	$(1, 2, 3, 4)$
b	$(\pi_1, \pi_2, \pi_4, \pi_3)$	$(1, 2, 4, 3)$
c	$(\pi_3, \pi_1, \pi_2, \pi_4)$	$(2, 3, 1, 4)$
d	$(\pi_4, \pi_2, \pi_1, \pi_3)$	$(3, 2, 4, 1)$

Similar

Position of π_4 is 1

Permutation Methods

- Filtering step - **compare permutations** instead of original data points to obtain γ candidate points
  - Footrule distance$(x, y) = \sum_i |x_i - y_i|$ (same as L_1)
  - Spearman's rho distance (same as L_2)



Point	Footrule$(a, \bullet)$
b	$\lvert 1-1\rvert + \lvert 2-2\rvert + \lvert 3-4\rvert + \lvert 4-3\rvert = 2$
c	$\lvert 1-2\rvert + \lvert 2-3\rvert + \lvert 3-1\rvert + \lvert 4-4\rvert = 4$
d	$\lvert 1-3\rvert + \lvert 2-2\rvert + \lvert 3-4\rvert + \lvert 4-1\rvert = 6$

candidate points

- Refinement step - **apply** $d(q, \bullet)$ for the candidate points (in our example, $\gamma = 2$, $q = a$, $d(q, b)$ and $d(q, c)$)

Permutation Methods

Filtering step:

- **Naive approach** - Brute force searching
  - using a priority queue
  - incremental sorting [Gonzales 2008] ($\times$2 faster than the priority queue approach)
  - binarized permutations (select a threshold *b* and use the Hamming distance)
  - **Brute force** in the permutation space is efficient if the distance is expensive.

Permutation Methods

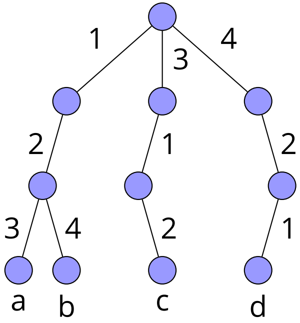
To reduce the cost of **the filtering stage**, three types of indices were proposed:

- use the existing methods for metric spaces [Figueroa 2009]
- the Permutation Prefix Index (PP-Index) [Esuli 2009]
- the Metric Inverted File (MI-file) [Amato et al. 2008]

Permutation Methods

Permutation Prefix Index (PP-index) [Esuli 2009]

Point	Pivot Order
a	$(\pi_1, \pi_2, \pi_3, \pi_4)$
b	$(\pi_1, \pi_2, \pi_4, \pi_3)$
c	$(\pi_3, \pi_1, \pi_2, \pi_4)$
d	$(\pi_4, \pi_2, \pi_1, \pi_3)$



Permutation Methods

Metric Inverted File (MI-file) [Amato et al. 2008]

Point	Pivot Order
a	$(\pi_1, \pi_2, \pi_3, \pi_4)$
b	$(\pi_1, \pi_2, \pi_4, \pi_3)$
c	$(\pi_3, \pi_1, \pi_2, \pi_4)$
d	$(\pi_4, \pi_2, \pi_1, \pi_3)$

	Posting Lists
$1 \rightarrow$	$(a, 1), (b, 1), (c, 2)$
$2 \rightarrow$	$(a, 2), (b, 2), (d, 2)$
$3 \rightarrow$	$(c, 1)$
$4 \rightarrow$	$(d, 1)$

Permutation Methods

Neighborhood Approximation Index (NAPP) [Tellez et al. 2013]

- Simplified version of MI-file
- Main differences:
  - Posting lists contain only object identifiers (no positions of pivots in permutations)
  - Not possible to compute the Footrule distance
  - The number of most closest *common* pivots is used to sort candidate objects

Indexing of permutations

Neighborhood APProximation index (NAPP) appears to be the best indexing approach:

Indexing of permutations

Neighborhood APProximation index (NAPP) appears to be the best indexing approach:

- Neighboring points should share some closest pivots

Indexing of permutations

Neighborhood APProximation index (NAPP) appears to be the best indexing approach:

- Neighboring points should share some closest pivots
- Index k closest pivots using an inverted file

Indexing of permutations

Neighborhood APProximation index (NAPP) appears to be the best indexing approach:

- Neighboring points should share some closest pivots
- Index k closest pivots using an inverted file
- Retrieve candidate points that share $m \leq k$ closest pivots with the query

Experimental settings

[noframenumbering]

- Our program is written in C++ and compiled in GCC 4.8 with the option `-Ofast`
- Linux Intel Xeon server (3.60 GHz, 32GB memory) in a single threaded mode using the Non-Metric Space Library
- Quality measure - Recall
- Performance measure - $\text{Improvement in Efficiency} = \frac{\text{time needed for brute force search}}{\text{time needed for approximate search}}$

Experiments: Indexing time

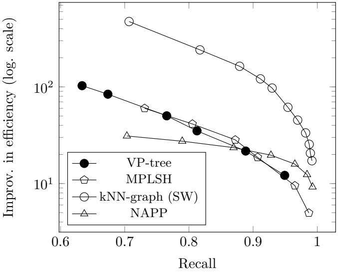
Indexing time in minutes:

	VP-tree	NAPP	MPLSH	Brute-force filt.	kNN graph
SIFT	0.4	5	18.4		**52.2**
ImageNet	4.4	33		32.3	**127.6**
Wiki-sparse		7.9			**231.2**
Wiki-128	1.2	**36.6**			**36.1**
DNA	0.9	15.9		15.6	**88**

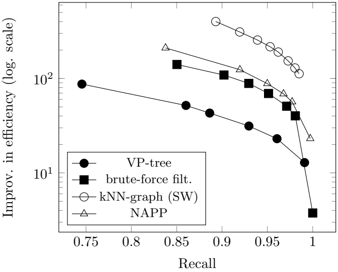
Experiments: Efficiency vs Accuracy

Improvement in efficiency over brute-force search vs. accuracy. Higher and to the right is **better**:

SIFT (L_2)



ImageNet (SQFD)

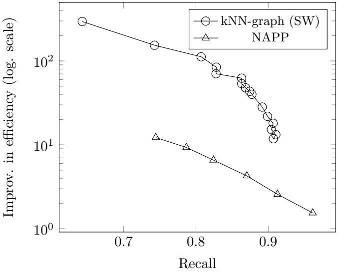


- NAPP beats MPLSH & VP-tree for SIFT, as well as VP-tree for Wiki-128
- kNN graph is the best for SIFT, Wiki-128, and ImageNet

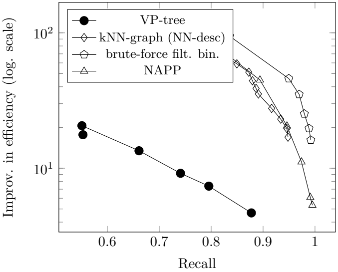
Experiments: Efficiency vs Accuracy

Improvement in efficiency over brute-force search vs. accuracy. Higher and to the right is **better**:

Wiki-sparse (cosine dist.)



Norm. Levenshtein



- kNN graph is the best for Wiki-sparse
- Brute force filtering beats all methods including kNN graphs for Norm. Levenshtein

Some Applications

NN-search is a core primitive in machine learning, vision and language processing.

Some Applications

NN-search is a core primitive in machine learning, vision and language processing.

- Query by image content
- Classification
- Entity detection
- Spell-checking