Permutation Search Methods are Efficient, Yet Faster Search is Possible

Bileg (Bilegsaikhan) Naidan\textsuperscript{1} \quad Leo (Leonid) Boytsov\textsuperscript{2} \quad Eric Nyberg\textsuperscript{2}

\textsuperscript{1}Norwegian University of Science and Technology (NTNU)
\textsuperscript{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 $q$

![Diagram showing nearest-neighbor search](image)
# Distance function

<table>
<thead>
<tr>
<th>Name</th>
<th>$d(x, y)$</th>
<th>Symmetry</th>
<th>Triangle ineq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euclidean ($L_2$)</td>
<td>$\sqrt{\sum (x_i - y_i)^2}$</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cosine distance</td>
<td>$1 - \frac{x \cdot y}{</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>KL-diverg.</td>
<td>$\sum x_i \log \frac{x_i}{y_i}$</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>JS-diverg.</td>
<td>symmetrized &amp; smoothed KL-diverg.</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

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

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

![Graph showing improvement in efficiency over recall for various methods including VP-tree, MPLSH, kNN-graph (SW), and NAPP.]
Experiments: Efficiency vs Accuracy

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

[Graph showing the comparison of different methods with labels on the y-axis: Improv. in efficiency (log. scale) and Recall. The x-axis shows Recall values from 0.6 to 1.0. The methods compared are VP-tree, kNN-graph (NN-desc), brute-force filt. bin., and NAPP. The graph illustrates the improvement in efficiency for different recall values, with higher efficiency values indicating better performance.]

- VP-tree
- kNN-graph (NN-desc)
- brute-force filt. bin.
- NAPP
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 datasets, 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 $\pi_i$.

<table>
<thead>
<tr>
<th>Point</th>
<th>Pivot Order</th>
<th>Permutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$(\pi_1, \pi_2, \pi_3, \pi_4)$</td>
<td>$(1, 2, 3, 4)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$(\pi_1, \pi_2, \pi_4, \pi_3)$</td>
<td>$(1, 2, 4, 3)$</td>
</tr>
<tr>
<td>$c$</td>
<td>$(\pi_3, \pi_1, \pi_2, \pi_4)$</td>
<td>$(2, 3, 1, 4)$</td>
</tr>
<tr>
<td>$d$</td>
<td>$(\pi_4, \pi_2, \pi_1, \pi_3)$</td>
<td>$(3, 2, 4, 1)$</td>
</tr>
</tbody>
</table>

Position of $\pi_4$ is 1
Permutation Methods

The data points are \( a, b, c, d \) in \( \mathbb{R}^2 \). The Voronoi diagram produced by 4 pivots \( \pi_i \).

**Permutation is a fancy word for a pivot ranking!**

<table>
<thead>
<tr>
<th>Point</th>
<th>Pivot Order</th>
<th>Permutations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>((\pi_1, \pi_2, \pi_3, \pi_4))</td>
<td>(1, 2, 3, 4)</td>
</tr>
<tr>
<td>( b )</td>
<td>((\pi_1, \pi_2, \pi_4, \pi_3))</td>
<td>(1, 2, 4, 3)</td>
</tr>
<tr>
<td>( c )</td>
<td>((\pi_3, \pi_1, \pi_2, \pi_4))</td>
<td>(2, 3, 1, 4)</td>
</tr>
<tr>
<td>( d )</td>
<td>((\pi_4, \pi_2, \pi_1, \pi_3))</td>
<td>(3, 2, 4, 1)</td>
</tr>
</tbody>
</table>

Position of \( \pi_4 \) is 1
Permutation Methods

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

- 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.
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]

<table>
<thead>
<tr>
<th>Point</th>
<th>Pivot Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$(\pi_1, \pi_2, \pi_3, \pi_4)$</td>
</tr>
<tr>
<td>b</td>
<td>$(\pi_1, \pi_2, \pi_4, \pi_3)$</td>
</tr>
<tr>
<td>c</td>
<td>$(\pi_3, \pi_1, \pi_2, \pi_4)$</td>
</tr>
<tr>
<td>d</td>
<td>$(\pi_4, \pi_2, \pi_1, \pi_3)$</td>
</tr>
</tbody>
</table>

![Pivot Order Tree](image)
# Permutation Methods

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

<table>
<thead>
<tr>
<th>Point</th>
<th>Pivot Order</th>
<th>Posting Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$(\pi_1, \pi_2, \pi_3, \pi_4)$</td>
<td>1 → $(a, 1), (b, 1), (c, 2)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$(\pi_1, \pi_2, \pi_4, \pi_3)$</td>
<td>2 → $(a, 2), (b, 2), (d, 2)$</td>
</tr>
<tr>
<td>$c$</td>
<td>$(\pi_3, \pi_1, \pi_2, \pi_4)$</td>
<td>3 → $(c, 1)$</td>
</tr>
<tr>
<td>$d$</td>
<td>$(\pi_4, \pi_2, \pi_1, \pi_3)$</td>
<td>4 → $(d, 1)$</td>
</tr>
</tbody>
</table>
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

- 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:

<table>
<thead>
<tr>
<th></th>
<th>VP-tree</th>
<th>NAPP</th>
<th>MPLSH</th>
<th>Brute-force filt.</th>
<th>kNN graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIFT</td>
<td>0.4</td>
<td>5</td>
<td>18.4</td>
<td></td>
<td>52.2</td>
</tr>
<tr>
<td>ImageNet</td>
<td>4.4</td>
<td>33</td>
<td>32.3</td>
<td>32.3</td>
<td>127.6</td>
</tr>
<tr>
<td>Wiki-sparse</td>
<td>7.9</td>
<td></td>
<td></td>
<td></td>
<td>231.2</td>
</tr>
<tr>
<td>Wiki-128</td>
<td>1.2</td>
<td></td>
<td>36.6</td>
<td>15.6</td>
<td>36.1</td>
</tr>
<tr>
<td>DNA</td>
<td>0.9</td>
<td>15.9</td>
<td></td>
<td></td>
<td>88</td>
</tr>
</tbody>
</table>
Experiments: Efficiency vs Accuracy

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

- 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.)**
  - kNN-graph (SW)
  - NAPP

- **Norm. Levenshtein**
  - VP-tree
  - kNN-graph (NN-desc)
  - brute-force filt. bin.
  - NAPP

- 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