Lecture Notes: From Nearest Neighbor Search to Maximum Inner Product Search

CS 7800/4810: Advanced Data Structures
Northeastern University

1 Introduction: Nearest Neighbor Search

1.1 Problem Definition

The nearest neighbor problem is one of the most fundamental problems in computational geometry and data management. Informally, given a collection of points and a query, we want to find the point in the collection that is closest to the query.

Definition 1.1 (Nearest Neighbor)**.** Given a universe Ω and a distance function $D : \Omega^2 \to \mathbb{R}$, the *nearest neighbor* of a query point $q \in \Omega$ in a finite set of points $P \subseteq \Omega$ is

$$p^* = \arg\min_{p_i \in P} D(p_i, q).$$

For example, when $\Omega = \mathbb{R}^2$ and D is the Euclidean distance, we are finding the closest point in the plane.

1.2 Nearest Neighbor Search (NNS)

The *nearest neighbor search* problem asks us to **preprocess** P so that we can **efficiently** find the nearest neighbor of any query q in P. The goals are:

- **Reasonable preprocessing time:** $\text{poly}(n, d)$, where $n = |P|$ and d is the dimensionality.
- **Fast query time:** $\text{poly}(\log n, d)$.

1.3 Similarity Search

Similarity is an opposing concept to distance: similar elements should have small distance, and dissimilar elements should have large distance.

A common way to convert a distance into a similarity is via a kernel function. For example, the **Gaussian kernel similarity** is defined as:

$$K(p, q) = \exp\left(-\frac{D(p, q)^2}{2\sigma^2}\right).$$

When $D(p, q) = 0$, the kernel evaluates to $e^0 = 1$ (maximum similarity). As $D(p, q) \to \infty$, the kernel approaches 0.

The role of σ. The parameter $\sigma > 0$ is called the *bandwidth* and controls how quickly the similarity decays with distance. It sets the characteristic distance scale at which points transition from "similar" to "dissimilar." To build intuition, consider a few concrete values:

$D(p,q)$	$K(p,q)$
0	1
σ	$e^{-1/2} \approx 0.607$
2σ	$e^{-2} \approx 0.135$
3σ	$e^{-9/2} \approx 0.011$

A **small** σ makes the kernel drop off sharply, so only very close points are considered similar. A **large** σ makes the decay gradual, so even distant points retain appreciable similarity.

The name "Gaussian" comes from the connection to the Gaussian (normal) probability distribution $\mathcal{N}(\mu, \sigma^2)$, whose density is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The kernel $K(p,q)$ is precisely the unnormalized Gaussian density centered at q, evaluated at p (with $D(p,q)$ playing the role of $|x-\mu|$). Just as σ controls the width of the bell curve in the normal distribution, it controls the radius of influence in the kernel: roughly 68% of the probability mass of a Gaussian lies within $\pm\sigma$ of the mean, 95% within $\pm 2\sigma$, and 99.7% within $\pm 3\sigma$.

In practice, σ is a tunable hyperparameter. In kernel methods (e.g., SVMs, kernel density estimation) and spectral clustering, it is often set via cross-validation or heuristics such as the median pairwise distance in the dataset.

Key Insight

Minimizing a distance function is equivalent to maximizing the corresponding similarity function. When the objective is to maximize a similarity function, the problem is called *similarity search*.

1.4 Common Distance and Similarity Functions

For $\mathbb{R}^d$:

L_p distance. The L_p distance is a family of distances parameterized by $p \geq 1$:

$$L_p(x,y) := \left(\sum_{i=1}^{d} |x_i - y_i|^p\right)^{1/p}.$$

The idea is: take the component-wise absolute differences, raise each to the p-th power, sum, and take the p-th root. The parameter p controls how much large differences in a single coordinate are penalized relative to spreading error across many coordinates. The three most common choices are:

- $p = 2$ **(Euclidean distance):** The familiar straight-line distance. In two dimensions this is the Pythagorean theorem: $\sqrt{(x_1-y_1)^2 + (x_2-y_2)^2}$. It treats all directions equally and penalizes large deviations quadratically before taking the square root.

- $p = 1$ **(Manhattan distance):** The sum of absolute differences: $\sum_{i=1}^{d} |x_i - y_i|$. Named after the grid-like street layout of Manhattan, it measures the distance you would travel if restricted to axis-aligned paths (no diagonal shortcuts). Because it does not square the differences, it is more robust to large deviations in a single coordinate compared to Euclidean distance.
- $p = \infty$ **(Chebyshev distance):** The maximum absolute difference across any single coordinate: $\max_i |x_i - y_i|$. Two points are "far" if they differ substantially in *any* dimension, even if they are close in all others. In chess, this is the number of moves a king needs to travel between two squares (since a king can move diagonally).

To build geometric intuition, consider the *unit ball* (all points at distance ≤ 1 from the origin) in $\mathbb{R}^2$. For L_1 it is a diamond (rotated square), for L_2 it is a circle, and for L_∞ it is an axis-aligned square. As p increases from 1 to ∞, the unit ball "inflates" from diamond to circle to square.

Cosine similarity. Cosine similarity measures the angle between two vectors, ignoring their magnitudes:

$$S_C(p, q) = \frac{p^T q}{\|p\|_2 \, \|q\|_2} = \cos(\theta),$$

where θ is the angle between p and q. It ranges from -1 (vectors pointing in opposite directions) through 0 (orthogonal vectors) to $+1$ (vectors pointing in the same direction). Cosine similarity is widely used for text and embedding similarity because it captures *direction*: two documents about the same topic will point in similar directions in embedding space regardless of whether one document's vector has a larger magnitude than the other's.

Dot product (inner product) similarity. The dot product similarity is simply

$$S_D(p, q) = p^T q = \sum_{i=1}^{d} p_i \, q_i.$$

Unlike cosine similarity, the dot product is sensitive to both direction *and* magnitude. A vector that points in the right direction and has a large norm will score higher than one that points in the right direction but has a small norm. This distinction matters in settings like recommendation systems, where magnitude may encode meaningful information (e.g., the "strength" or "popularity" of a user/item representation).

Relationship to cosine similarity: If all vectors are normalized to unit length ($\|p\| = \|q\| = 1$), then the dot product equals the cosine similarity. So cosine similarity can be viewed as the dot product restricted to the unit sphere.

For strings:

Edit distance (Levenshtein distance). The edit distance between two strings is the minimum number of single-character *insertions*, *deletions*, or *substitutions* required to transform one string into the other. For example, "kitten" $\rightarrow$ "sitting" has edit distance 3 (substitute k$\rightarrow$s, substitute e$\rightarrow$i, insert g). Edit distance is computed via dynamic programming in $O(nm)$ time, where n and m are the lengths of the two strings.

Hamming distance. For two strings of *equal length*, the Hamming distance counts the number of positions where the characters differ. For example, "karolin" vs. "kathrin" has Hamming distance 3. It can be viewed as a special case of edit distance where only substitutions are allowed and the strings must have the same length. This is the distance function used in the LSH example in Section 3.

For the power set of a finite set:

Jaccard similarity and distance. For two sets A and B, the Jaccard similarity measures their overlap:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

It ranges from 0 (disjoint sets) to 1 (identical sets). The Jaccard distance is $1 - J(A, B)$. This is commonly used for comparing documents represented as sets of words (or shingles), for near-duplicate detection, and as the basis for MinHash-based LSH schemes.

For nodes in a graph:

Shortest path distance. The distance between two nodes in a graph is defined as the length (number of edges, or total edge weight in a weighted graph) of the shortest path between them. This is a proper metric on connected graphs—it satisfies non-negativity, symmetry, and the triangle inequality.

For point sets:

Kernel distance. The kernel distance measures the discrepancy between two point sets using a kernel function (such as the Gaussian kernel discussed in Section 1.3). It compares the "shapes" of two distributions and is used in two-sample testing and distribution comparison.

Gromov–Hausdorff distance. The Gromov–Hausdorff distance measures how far two metric spaces are from being isometric. It is used in shape matching and computational topology to compare geometric objects up to rigid transformations.

1.5 Applications of NNS

Efficient nearest neighbor search has broad applications:

- **Learning:** Pattern recognition, prediction, classification (e.g., k-NN classifiers).
  - *Drawback:* Requires storing all data points. Model-based learning methods are often preferable for pure classification tasks.
- **Matching:** DNA sequencing, point cloud registration, compression, clustering.
  - These are general NNS problems, but domain-specific data often calls for specialized algorithms.
- **Searching:** Information retrieval, web search, map search, recommendation systems, plagiarism detection.

– These involve large volumes of vector data (low or high dimensional) that must be stored regardless, making efficient NNS systems crucial.

Remark 1.2. The remainder of these notes focuses on the most common data space: d-dimensional real vectors, i.e., $\Omega = \mathbb{R}^d$.

2 Low-Dimensional Data

In low dimensions, exact and efficient NNS is achievable.

2.1 Dimension $d = 1$

For one-dimensional data, the problem reduces to predecessor/successor search:

- **Sorted array + binary search:** $O(n)$ space, $O(\log n)$ query time.
- **Balanced binary search tree:** $O(n)$ space, $O(\log n)$ query time.
- **van Emde Boas tree / x-fast trie:** $O(n)$ space, $O(\log \log u)$ query time (where u is the universe size).

2.2 Dimension $d = 2$

For two-dimensional data, geometric data structures provide efficient solutions:

- **Voronoi diagram:** The Voronoi diagram partitions the plane so that each cell contains exactly those points closer to one data point than to any other. Combined with a planar point location structure:
  - $O(n)$ space, $O(\log n)$ query time [Lipton–Tarjan '80].
- **kd-tree:** A binary space-partitioning tree that alternately splits along each coordinate axis:
  - $O(n)$ space, $O(\log n)$ query time on average.

3 High-Dimensional Data: The Curse of Dimensionality

3.1 Why High Dimensions are Hard

The low-dimensional techniques described above do not scale well to high dimensions. This breakdown is collectively known as the *curse of dimensionality*.

Exponential space partitioning. The strategy behind efficient NNS in low dimensions is *space partitioning*: divide the space into cells, preprocess each cell, and at query time only examine a small number of cells. The problem is that the number of cells grows exponentially with dimension.

Observation 3.1 (Curse of Dimensionality — Space)**.** All known exact NNS data structures that achieve sub-linear query time (i.e., beat $O(dn)$ linear scan) require $O(2^d)$ space. Intuitively, splitting the data space in half along each of d dimensions yields 2^d partitions.

Concentration of distances. A deeper geometric issue is that in high dimensions, all pairwise distances tend to become nearly equal. Consider n points drawn uniformly from $[-1, 1]^d$. The expected squared Euclidean distance between any two points is

$$\mathbb{E}\left[\|p - q\|^2\right] = \sum_{i=1}^{d} \mathbb{E}\left[(p_i - q_i)^2\right] = d \cdot \frac{2}{3},$$

since each coordinate difference has variance $2/3$. By the law of large numbers, as d grows, the actual distance $\|p-q\|^2$ concentrates tightly around $2d/3$ for *every* pair. The standard deviation of $\|p-q\|^2$ grows as $O(\sqrt{d})$, so the *relative* spread (ratio of standard deviation to mean) shrinks as $O(1/\sqrt{d})$. In practical terms: when $d = 1000$, the nearest neighbor and the farthest neighbor are nearly the same distance from the query, making the concept of "nearest" almost meaningless for uniformly distributed data.

Failure of spatial data structures. These geometric phenomena explain why data structures like kd-trees degrade in high dimensions. A kd-tree prunes search by determining that the query's distance to a splitting hyperplane exceeds its current best distance, allowing it to skip an entire subtree. In high dimensions, the distance to a splitting hyperplane (which involves only one coordinate) is almost always much smaller than the distance to any data point (which involves all d coordinates). As a result, the pruning condition is rarely satisfied, and the kd-tree degenerates to a linear scan. Empirically, kd-trees become slower than brute force for $d \gtrsim 20$.

3.2 Approximate NNS and the PLEB Problem

Since exact NNS is intractable in high dimensions (without exponential space), we relax the problem to allow approximate answers.

PLEB stands for Point Location in Equal Balls, and it's a decision-problem relaxation of approximate nearest neighbor search that makes the problem easier to reason about and build algorithms for. The idea is this: instead of asking "find the closest point to my query," PLEB asks a simpler yes/no-style question: "is there a point within radius r of my query?", but with an approximation gap built in.

Definition 3.2 (ε-Approximate Nearest Neighbor (ε-NN))**.** Given a point set P and a query point q, a point $\hat{p} \in P$ is an ε-NN of q in P if

$$D(\hat{p}, q) \leq (1+\varepsilon) \min_{p \in P} D(p, q).$$

A key intermediate problem that enables efficient approximate NNS is:

Definition 3.3 (ε-Point Location in Equal Balls (ε-PLEB))**.** Given a point set P and a radius $r \in \mathbb{R}^+$, for any query point q:

1. If $\min_{p \in P} D(p, q) \leq r$, return a point p' such that $D(p', q) \leq (1+\varepsilon)r$.
2. If $(1+\varepsilon)r \leq \min_{p \in P} D(p, q)$, return $\emptyset$.
3. If $r \leq \min_{p \in P} D(p, q) \leq (1+\varepsilon)r$, return either p' or $\emptyset$ (either answer is acceptable).

The PLEB formulation is useful because it separates "near" points (within radius r) from "far" points (beyond radius $(1+\varepsilon)r$), with a gap zone where any answer is valid. An ε-NN data structure can be built from an ε-PLEB data structure by searching over $O(\log n)$ scales of r.

3.3 Locality-Sensitive Hashing (LSH)

The breakthrough approach for approximate NNS in high dimensions is **Locality-Sensitive Hashing** (LSH), introduced by Indyk and Motwani (1998).

Definition 3.4 (Locality-Sensitive Hashing)**.** A family $\mathcal{H} = \{h : \Omega \to S\}$ is called (r_1, r_2, p_1, p_2)-sensitive for a distance function D if for any $q, p \in \Omega$:

- If $D(p, q) \leq r_1$, then $\Pr_{h \sim \mathcal{H}}[h(q) = h(p)] \geq p_1$.

- If $D(p,q) \geq r_2$, then $\Pr_{h\sim\mathcal{H}}[h(q) = h(p)] \leq p_2$.

where $p_1 > p_2$ and $r_1 < r_2$.

The key property is that nearby points hash to the same bucket with higher probability than far-away points. By amplifying this gap (using multiple hash functions and multiple hash tables), we can build an efficient approximate NNS data structure.

Example: Hamming space. For the d-dimensional Hamming space $\Omega = \{0,1\}^d$ with Hamming distance $D(p,q) = \sum_{i=1}^{d} \mathbf{1}(p_i \neq q_i)$, the family

$$\mathcal{H} = \{h_i : h_i(p) := p_i,\ i = 1, \ldots, d\}$$

(i.e., each hash function simply selects one coordinate) is $(r,\ r(1+\varepsilon),\ 1-r/d,\ 1-r(1+\varepsilon)/d)$-sensitive.

3.4 The Indyk–Motwani LSH Algorithm

Algorithm 1 LSH Preprocessing

Require: Point set $P \subset \{0,1\}^d$, number of hash tables ℓ, hash length k

Draw $\ell \times k$ hash functions i.i.d. from $\mathcal{H}$: $H \leftarrow \mathcal{H}^{\ell \times k}$
Initialize ℓ hash tables: $T_1, \ldots, T_\ell \leftarrow \{\}$
for each $p \in P$ **do**
    for $j = 1$ to ℓ **do**
        $T_j[(H_{j,1}(p), \ldots, H_{j,k}(p))] \leftarrow p$
    end for
end for
return T, H

Algorithm 2 LSH Query

Require: Hash tables $T_1, \ldots, T_\ell$, hash functions H, query point $q \in \{0,1\}^d$

for $j = 1$ to ℓ **do**
    $p \leftarrow T_j[(H_{j,1}(q), \ldots, H_{j,k}(q))]$
    if $D(q,p) \leq (1+\varepsilon)r$ **then**
        return p
    end if
end for
return $\emptyset$

Parameter settings and complexity. With $k = O(\log n)$ and $\ell = O(n^{1/\varepsilon})$:

- **Space:** $O(dn + n^{1+1/\varepsilon})$.
- **Query time:** $O(d \cdot n^{1/\varepsilon})$.

Theorem 3.5 (Indyk–Motwani '98)**.** For any $\varepsilon > 1$, there exists an algorithm for ε-PLEB in $\{0,1\}^d$ using $O(dn + n^{1+1/\varepsilon})$ space and $O(dn^{1/\varepsilon})$ query time.

Intuition Behind LSH

Each hash function h provides a weak signal about whether two points are close. By concatenating k hash functions (forming a compound key), we reduce false positives (far points colliding). By using ℓ independent hash tables, we ensure that true near neighbors are found with high probability.

4 Maximum Inner Product Search (MIPS)

4.1 Why Not Euclidean Distance in High Dimensions?

In high-dimensional spaces, Euclidean distance becomes a poor discriminator of similarity. This phenomenon is a manifestation of the curse of dimensionality.

Concentration of distances. Consider $n = 1000$ points sampled uniformly from $[-1, 1]^d$. As d grows, the distribution of pairwise Euclidean distances (normalized by the maximum distance) concentrates sharply:

Dimension d	**Behavior**
$d = 3$	Distances spread broadly across $[0, 1]$.
$d = 10$	Distribution begins to concentrate.
$d = 100$	Narrow peak; most distances are nearly identical.
$d = 1000$	Extremely concentrated; a small additive error includes most points.

At $d = 1000$, an error of $2/24$ of the range already encompasses more than 50% of all points, and an error of $3/24$ includes nearly all points. This makes Euclidean nearest neighbor search almost meaningless for uniformly distributed high-dimensional data.

Hyper-cube and hyper-sphere range queries. Another way to see the curse: the probability that a uniformly random point in $[0, 1]^d$ falls within the largest inscribed ball (radius 0.5) drops super-exponentially with d:

d	$\Pr[R \in \text{Ball}(Q, 0.5)]$
2	0.785
4	0.308
10	0.002
20	2.461×10^{-8}
40	3.278×10^{-21}
100	1.868×10^{-70}

In contrast, **cosine similarity** remains a meaningful measure even in high dimensions. While the distribution of pairwise cosine similarities also concentrates around zero as d grows, it retains discriminative power for data that has meaningful angular structure (as is the case for learned embeddings).

4.2 Embedded Data

In modern machine learning, virtually all data modalities—text, images, audio—are represented as high-dimensional vectors (embeddings). These embeddings are learned by minimizing loss functions that are often defined in terms of cosine similarity or dot product:

- Google's Universal Sentence Encoder: $d = 512$.
- GPT-3 embeddings: $d \in [1024, 12288]$.

Because these embeddings are trained to encode semantic similarity via the inner product or cosine similarity, the natural search problem becomes **Maximum Inner Product Search (MIPS)**: given a query q, find $p^* = \arg\max_{p_i \in P} p_i^T q$.

4.3 Approaches to High-Dimensional MIPS

Building an efficient MIPS system involves two complementary tasks:

1. **Reducing the number of candidates** (via space/data partitioning):
   - **Tree-based methods:** Randomized partition trees, spill trees [Muja–Lowe '14; Dasgupta–Freund '08].
   - **Locality-sensitive hashing:** Cross-polytope LSH, asymmetric LSH for MIPS [Shrivastava–Li '14; Neyshabur–Srebro '15; Andoni–Indyk–Laarhoven–Razenshteyn–Schmidt '15].
   - **Graph-based search:** Navigable small-world graphs, e.g., HNSW [Malkov–Yashunin '16; Harwood–Drummond '16].

2. **Speeding up distance/similarity evaluation:**
   - **Quantization:** Product quantization (PQ), optimized product quantization (OPQ), and related techniques compress vectors so that approximate distances can be computed from compact codes.
   - **Random projection / dimension reduction:** The Johnson–Lindenstrauss lemma guarantees that random linear projections approximately preserve distances, enabling faster evaluation in reduced dimensions [Ailon–Liberty '13].

4.4 Application Spotlight: MIPS in LLM Inference

One of the most important modern applications of nearest neighbor search—and MIPS in particular—arises in **large language model (LLM) inference**, specifically in the attention mechanism and its interaction with the key-value (KV) cache.

Background: attention as inner product search. The core operation in transformer-based LLMs is *scaled dot-product attention*. Given a query vector $q \in \mathbb{R}^{d_k}$ (derived from the current token being generated) and a set of key vectors $K = \{k_1, \ldots, k_m\} \subset \mathbb{R}^{d_k}$ (derived from all preceding tokens), the attention mechanism computes:

$$\text{Attention}(q, K, V) = \text{softmax}\left(\frac{qK^T}{\sqrt{d_k}}\right) V,$$

where V is the corresponding matrix of value vectors. The softmax operation is dominated by the tokens with the *largest inner products* $q^T k_i$. In other words, attention is implicitly performing a soft version of MIPS: it identifies which keys are most relevant to the current query and concentrates the output on their associated values.

The KV cache scaling problem. During autoregressive generation, each new token must attend over all previously generated tokens. The key and value vectors for past tokens are stored in a *KV cache* to avoid redundant recomputation. For a model with L layers, H attention heads, head dimension d_h, and a context of m tokens, the KV cache consumes $O(L \cdot H \cdot m \cdot d_h)$ memory. For modern LLMs (e.g., LLaMA with 128K context), this can reach tens of gigabytes per request.

The computational cost is equally problematic. The prefill phase (processing a prompt of m tokens) requires computing self-attention among all m tokens, which costs $O(m^2 \cdot d_h)$ per head per layer. This quadratic scaling in context length is the dominant bottleneck for long-context inference.

Approximate attention via ANN search. The connection to NNS/MIPS suggests a natural optimization: since attention weights are concentrated on the keys with the highest inner products, we can use *approximate nearest neighbor* (ANN) methods to identify the top-k most relevant keys for each query, and compute attention only over those k keys. This reduces the per-query attention cost from $O(m \cdot d_h)$ to $O(k \cdot d_h)$, where $k \ll m$.

Several lines of work exploit this idea:

- **Sparse attention patterns** (e.g., Reformer, Routing Transformer) use LSH or learned routing to select a subset of keys for each query.
- **KV cache compression** methods evict or merge low-importance KV entries based on attention score estimates, which are themselves inner product computations.

MIPS as the Core Primitive

Attention *is* MIPS: the softmax over qK^T finds the keys most aligned with the query. Every technique for scaling long-context LLM inference—sparse attention, KV cache eviction, memory injection, retrieval-augmented generation—ultimately reduces to performing approximate MIPS more efficiently over the stored key vectors. The data structures and algorithms discussed in these notes (LSH, graph-based search, quantization) are the building blocks that make these optimizations possible.