**CS 7280: Data Str & Alg Scalable Comp** Spring 2025

# Lecture 16 — March 12, 2025

*Prof. Prashant Pandey* *Scribe: Irakli Gurgenidze*

## 1 Overview

In the last lecture, we began studying the probabilistic estimation of set qualities. We covered the MASH (bottom k min-hash sketch) algorithm to efficiently estimate the similarity between two sets, and introduced the concept of cardinality (number of elements) estimation via HyperLogLog.

In this lecture, we expand on the idea of similarity by introducing the concept of Nearest Neighbors, and investigate the utility that similarity estimation techniques offer in practical applications.

## 2 Nearest Neighbors

To frame our discussion, let us first introduce the idea of **Jaccard Similarity**:

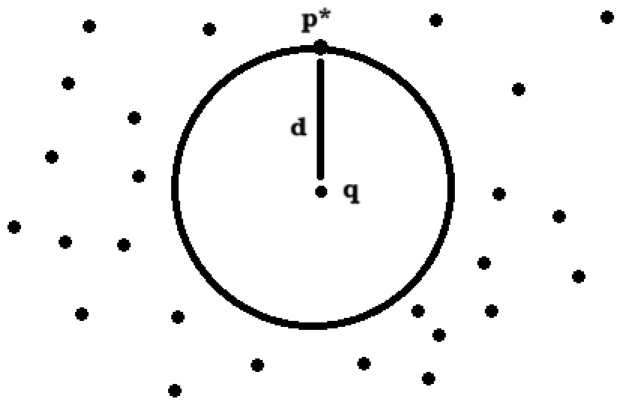
$$J = \frac{|A \cap B|}{|A \cup B|}, \quad J \in [0, 1]$$

In short, the similarity of two sets $A$ and $B$ can be defined by the ratio of overlapping elements to total unique elements. $J(A, A) = 1$.

In modern computing, it is common for databases to have vector keys, sometimes of very high dimensions. Given a large enough vector space, performing an exact search over a database can be extremely expensive. Similar to filters, by relaxing our tolerance for error, we can achieve greater efficiency when working with big data.

**Nearest Neighbor Search** aims to do exactly that - approximately match queries to similar keys:

Given a universe $\Omega$ 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 $P \leq \Omega$ is $p^* = argmin_{p_i \in P}\ D(p_i, q)$.



## 3 Nearest Neighbor Search (NNS)

By preprocessing $P$, we can efficiently find the nearest neighbor of $q$, where $q \in \mathbb{R}^d$:

- poly$(n, d)$ - preprocessing
- poly$(\lg n, d)$ - query

Similarity is the opposite of distance. Similar elements have a small distance, and the opposite is true for very different elements. Minimizing the difference between two elements is equivalent to maximizing the similarity between them. When the objective is maximizing the similarity between two elements, we call that a **similarity search**.

### 3.1 Common Distance Functions

1. For $\mathbb{R}^d$
   - $L_p$ distance: $L_p(P, q) = \left(\sum_{i=1}^{d} |P_i - q_i|^p\right)^{\frac{1}{p}}$ for $p \geq 1$.

   When $p = 2$ we have Euclidean distance. Other common picks are $p = 1$ (Manhattan distance) and $p = \infty$ (Chebyshev distance).

2. For strings
   - Edit distance
   - Hamming distance

3. For sets
   - Jaccard Similarity

4. For nodes in graphs
   - Length of the shortest path

### 3.2 Applications

What can we do with efficient NNS?

1. Learning
   - Pattern recognition
   - Prediction
   - Classification

   → Need to store all data points
   → Model-based methods might be more efficient

2. Matching
   - DNA sequencing
   - Compression
   - Clustering

   → Domain-specific data require domain-specific algorithms

3. Searching
   - Info retrieval
   - Web search
   - Map search
   - Plagarism detection

   → A lot of data
   → Often vector data, of high or low dimensionality

Efficient NNS is crucial across many domains.

### 3.3 Curse of Dimensionality

We will focus on the most common data space - a $d$-dimensional vector $\Omega = \mathbb{R}^d$.

If $d = 1$, we can sort the data with BST, hashtables, VEB Trees, etc. This gives us $O(n)$ space, and $O(\lg n)$ query time.

If $d = 2$, we can represent the data as a Voronoi Diagram [Lipton-Tarjan '80] or a KD Tree [Jon, Louis, Bentley '75]. This too gives us $O(n)$ space, and $O(\lg n)$ query time.

The **curse of dimensionality** states that all known data structures that beat $O(dn)$ linear scan query time require $O(2^d)$ space.

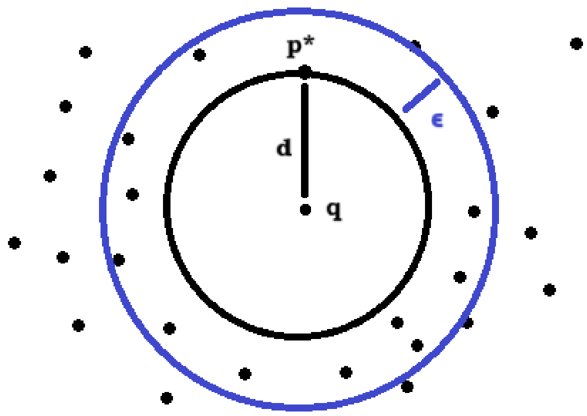
**Intuition:** Splits the data space in each dimension into two halves, end with $2^d$ partitions.

## 4 Approximate Nearest Neighbor

$\epsilon$ **Nearest Neighbor Search:** Given a point set $P$ and a query point $q$, we say $\hat{p} \in P$ is an $\epsilon - NN$ of $q$ in $P$ if $D(\hat{p}, q) \leq (1 + \epsilon) \cdot min_{p \in P}\ D(p, q)$

**$\epsilon$-NN Search:** Given a point set $P$ and $r \in \mathbb{R}^+$, for any query point $q$

- if $min_{p \in P}\ D(p,q) \leq r$, return a point $p'$ such that $D(p',q) \leq (1+\epsilon)r$
- if $(1+\epsilon)r \leq min_{p \in P}\ D(p,q)$, return $\emptyset$
- if $r \leq min_{p \in P}\ D(p,q) \leq (1+\epsilon)r$, return $p'$ or $\emptyset$



## 5 Hashing Recap

$U$ - universe
$T$ - hash table
$H$ - $[h : U \to \{0, T-1\}$

Hashing is useful because it allows us to "map" a small number $N$ of keys from a large universe $U$, to a much smaller universe $U$. A **universal hash function** is a type of hash function that belongs to a family of functions with strong guarantees in minimizing collisions.

Specifically, a **family** of hash functions $H$ is **universal** if, for any two distinct $x \neq y$, the probability that a randomly chosen hash function $h \in H$ maps them to the same value is at most $\frac{1}{|T|}$:

$$Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{|T|}, \quad \forall x, y \in U, x \neq y$$

A good hash function will take some item $x$, project it into a much larger space, and use *mod* of some large prime number to compute the output of $h(x)$. For any set of keys, our goal is to produce a set of hashes that are **uniformly random**. That is, for any item $x \in U$, $Pr(h(x) = m) = \frac{1}{|T|}$, where $m$ is any index in the hash table.

**A bad hash function:** $h(x) = x \ mod \ |T|$. This function preserves the structure in the initial set of keys.

**A good hash function:** $h(x) = ax + b \ mod \ P \ mod \ |T|$, where $1 \leq a \leq P$, $0 \leq b < P$, and $P$ is some very large prime number. If we choose $a$ and $b$ uniformly randomly, we can expect a uniformly random output $h(x) \ \forall x \in U$.

Randomness **is not** encoded in the hash function - it comes from the selection of $a$ and $b$.

## References

[1] Richard Lipton, Robert Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics, Vol 36. Iss. 2*, DOI 10.1137/0136016, 1979.

[2] John Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM Vol. 18, No. 9*, 509-517, 1975.