1 Motivation

Similarity search covers a broad range of techniques that allow for searching where objects in the domain can be compared via some metric, termed similarity. Nearest neighbor search, which we will be covering in today’s lecture, is a specific application of similarity search.

In k-Nearest Neighbor Search (k-NNS), you have a set of data and you’d like to find, given a query, which k datapoints in the set are most similar to the query. One practical example is checking document similarity - given a set of documents and a query document, you’d like to find the k most similar documents to the query document by considering the documents mapped in some feature space. Similarity is encoded by some norm (distance measure) on the space. The next section will provide more details on this.

2 k-Nearest Neighbor Search (k-NNS)

k-NNS is a data structure question that has the following definition:

- Dataset: $X \subset (\mathbb{R}^d, ||.||), |X| = n$
- Query: $q \in \mathbb{R}^d$
- Norm: $||.||_2$, where $||x||_2 = \sqrt{\sum_i x_i^2}$ (k-NNS can be defined with an arbitrary norm, and indeed the $\ell_1$ norm is commonly considered, but we will be focusing on the $l_2$ norm for this lecture)
- Goal: Given a query object $q$, retrieve $k$ most similar objects from the database $X$ quickly.

There are two parameters we care about: space (ideally $O(nd)$) and query time (ideally $O(d)$). A naive approach to this problem would be to do a simple linear scan, which achieves our ideal space complexity; however, in terms of query time, we get a bound of $O(nd)$ (since we need to loop through the entire dataset and calculate $l_2$-norm for each data point against the query). Thus, we need a better approach.

The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces. This occur in domains such as computational geometry, machine learning, and databases. As mentioned above, all the known NNS data structures are faster than linear scan, which require $2^{\Omega(d)}$ space, thus it becomes infeasible for $d >> \log(n)$. A data structure with pre-processing time $\text{poly}(n, d)$ and query time $\text{poly}(d) \cdot n^{0.99}$ would refute the Strong Exponential Time Hypothesis (SETH) [1]. The hypothesis states that $k$-SAT cannot be solved in subexponential time in the worst case as $k \to \infty$. 
From the curse of dimensionality, the theory and the “practice” give different but related answers. Theory would give us approximated answers, but it supports arbitrary datasets as it is shown in e.g. [6,7]. On the other hand, in practice we often require exact answers, but only for “nice” instances (e.g. when nearest neighbors are noticeably closer to the query than a typical point). The following graph is an example of a “nice” dataset, where the graph shows the number of words as a function of distance from a fixed word in the GloVe dataset [2]:

![Graph showing number of words as a function of distance](image)

This lecture will focus on practical algorithms that use machine learning.

### 2.1 Pipeline

The diagram above describes the modern pipeline. Start with query point \( q \in \mathbb{R}^d \), pass through an index, or data structure, which looks at the query and “quickly” returns a list of \( T \) candidate points, where \( T << n \) which is then further refined by passing through further sketches. The new list is size \( T' \), where \( T' << T \). Sketches let us give a crude estimate of “distance” between data points (and to query point). The sketching procedure is as follows: Compress \( x \) into \( \text{sk}(x) \) and \( y \) into \( \text{sk}(y) \) such that we can very quickly estimate \( \|x - y\|_2 \). A sketch representation could be a short binary string.

Then, we pass this new candidate list, an output from the sketch, into exact distance measurement to return a final list of \( k \) points. \( k \) points are the fraction from the “true” list of \( k \) nearest neighbors.
Sometimes sketches are not used, and sometimes an index is not used, but often times it helps to have the whole pipeline in place. The performance of this pipe line is determined by averaging over a set of queries $Q$.

Two of the most popular sketching techniques are 1) Hamming distance (very fast - 2 CPU cycles, but usually not very accurate) 2) Quantization (very accurate, but not too fast). We now give more details on these types of sketching techniques. We will first focus on Hamming estimators (like using random projections), then look at some more sophisticated techniques like iterative quantization and product quantization. Finally, we will overview some indexing techniques and discuss new results that involve learning-augmented algorithms.

## 2.2 Hamming Estimators

Hamming estimators map $x \in \mathbb{R}^d$ to $r \in \{0, 1\}^s$ via methods that will be described in the following subsections. As stated earlier, Hamming estimators are fast, but usually not very accurate.

### 2.2.1 Random Projections

The definition of “random” projections include a vector $x \in \mathbb{R}^d$ and a $d \times d$ random orthogonal matrix $A$. In order to sample a uniformly random orthogonal matrix, we sample a random Gaussian matrix and compute its QR decomposition, in which a matrix $A$ is decomposed into an orthogonal matrix $Q$ and an upper triangular matrix $R$.

The encoding procedure will be following: $x \mapsto \text{sgn}(Ax) \in \{0, 1\}^d$. Then, we use Hamming Distance on the resulting sketches as an estimator for distance in the dataset. Before we do so, we need to center our dataset to increase the quality because random projections is not a shift-invariant key. Next, we want more than $d$ bits to repeat to improve the accuracy. Finally, use the fast Fourier transform to compute matrix multiplications, such as $Ax$, faster, in less than $O(d^2)$ time.

### 2.2.2 ITQ (Iterative Quantization) [3]

Instead of random rotation like $A$, the objective is to design a rotation that is better. In ITQ, the authors start with PCA-projected data. The authors note that although random orthogonal rotations do quite well (referencing a past paper), they are going to propose an alternating minimization approach that further reduces the quantization error.

**Setup:**

- $X \subset \mathbb{R}^d$
- $A$: orthogonal matrix
- $f: X \mapsto \{-1, 1\}^d$
- Optimization prob: $\min_{x \sim X} E_x ||f(x) - Ax||_2$

An alternating minimization technique is used due to the fact that our mapping $f$ is neither continuous nor convex, rendering traditional gradient-based techniques on the objective unhelpful.
Sketches: experimental results

GloVe dataset, 1.2M points, 100 dimensions (3200 bits)

**Quality:** how many candidates we need to filter to get accuracy 0.9 on 10-NN

![Graph](image)

Figure 1: Blue line in plot: Using simple random projection method from 2.2.1. Red line in plot: modification is that we take \( x \) and \( \text{proj}_V x \), where \( V = \text{span} \langle \text{top few PCA dirs} \rangle \) (largest variance). Green line in plot: Uses product quantization (covered later in the notes).

Thus, the hope for the following technique is that we converge to a fixed point (in matrix space) \( A \) that is better than simply using a random rotation.

**Proposed Approach:**

- Fix \( A \) and minimize \( f \) by taking signs
- Fix \( f \) and minimize \( A \) among orthogonal matrices (Orthogonal Procrustes using SVD: Orthogonal Procrustes Problem)

We can imagine the first step from above as picking the vertex on the binary hypercube \( \{-1, 1\}^d \) that is closest to the datapoint. The second step is, given this vertex assignment, adjust the orthogonal matrix rotation to minimize the optimization problem defined in the setup. This optimization problem, given a fixed binary hypercube vertex, is actually a well-known problem called the Orthogonal Procrustes problem. The solution this problem is explained in the Wikipedia link in the proposed approach.

### 2.3 Quantization

K-means clustering can be used as a quantization technique. K-means is an unsupervised technique that groups the dataset of points \( X \) into \( K \) clusters, each with a corresponding mean that every point in the cluster is closest to. If we denote the mean of the cluster that a data point \( x \) belongs to as \( c_x \), we can use the following approximation in our quantization sketch:

\[
||q - x||^2_2 \approx ||q - c_x||^2_2
\]
2.3.1 Product Quantization [4]

One issue with $K$-means clustering is that the query time scales with the number of centers; hence, if we want a large number of centers to get high accuracy, then we need to accept a large query time. Product quantization tries to solve this problem, and it proceeds as follows:

- Partition $x$ into $B$ blocks; each block has $\frac{d}{B}$ points; compute k-means in each block; we now have $k^B$ possible centers.
- Encoding: $x \mapsto (c_1, c_2, ..., c_B)$
- Given query point $q$, produce a table of $B$ rows and $k$ columns where an entry has $\ell_2$-norm of $(q \text{ block } i) - \text{center}(j \text{ in block } i)$. We can now try to find the index of the table with the smallest $\ell_2$-norm to find a nearest neighbor.

The advantage of using Product Quantization over naive k-means clustering is that we are given a much larger set of "centers" to choose from in the former ($k^B$ for Product Quantization vs $k$ for k-means clustering) without trading for much in the way of query or space complexity. To calculate the query time, the time it takes to compute the $\ell_2$ distance between the query point and $k$ centers in $\frac{d}{B}$-dimensional space is $O(\frac{d}{B} \cdot k)$. If we sum this time bound over all blocks $B$, we get $O(kd)$ query time per point.

2.4 Indexing

Now we discuss indexing. We will focus on partition-based indexing. The goal is to partition our underlying space so that, when we receive a query point, we have an indication as to which of our dataset points to look at (for example, those in the same part as our query or those in nearby parts). One example of such a partition is a Voronoi diagram:

![Voronoi Diagram](image)

Figure 2: Voronoi Diagram: Decomposition of plane of points into regions such that all points within a region are closest to the representative point in the region and farther away from all other points in the dataset.

5
Computing a Voronoi diagram can be expensive. So let us recap what we are looking for in a partition. The idea is that we can make a partition that is more crude, but still has the properties we want for nearest neighbor search.

In particular, we want the following properties for a P-partition of $\mathbb{R}^d$: 1) we want to look up few points, 2) we want to hit many nearest neighbors in the same or closeby parts to the query point, and 3) be algorithmically simple for a query. In particular, to get Property 1) we will want the partition to be approximately balanced with respect to the dataset (so that there are no parts with overwhelmingly many dataset points). Possible solutions include:

1) Use a decision tree where each node represents a hyperplane — if there exist $t$ levels, then we have $2^t$ parts by only computing $t$ dot products. Then, for a query procedure for partition is as follows: (1) start in the root and decide which side of the hyperplane you are (2) depending on which side determines right or left child to traverse and (3) continue this process.

2) Locality-sensitive hashing is a way to produce partitions of $\mathbb{R}^d$ to satisfy three conditions above. It has some probabilistic guarantees, but it is data-oblivious. This is similar to a universal hashing because it works for all kinds of data, but it does not tailor to any specific data set.

3) Use k-means to partition the space. Given a query, compute the distance from each candidate point. This approach minimizes the sum of squared errors.

4) PCA tree; compute top PCA direction of the dataset, and cut it in the middle orthogonally. Because the dataset is spread, the first time will not split many nearest neighbors. Thus, continue to recurse on each half.

5) New partitioning framework [5]: Use a heuristic algorithm (KaHIP, see [8]) to partition the k-Nearest Neighbor graph on the dataset in a balanced way, but extend this partitioning to all of $\mathbb{R}^d$ in order to answer arbitrary queries. Supervised learning is used where the training data is their dataset, in order to extend their partition to $\mathbb{R}^d$. This ends up creating a partition that optimizes the objective function they want to directly (i.e. approximately balanced wrt dataset, so you look up just a few points, and most nearest neighbors are in the same or one of the close parts), and uses learning for a classical algorithms problem in a meaningful way. The paper has two approaches: using a logistic regression and a small neural network. Also, find a balanced and unsupervised partition is important. The followings are the experimental results:

![Graph 1](image1.png)

For GloVe, Regression LSH significantly outperforms 2-means, while for SIFT, Regression LSH essentially matches 2-means in terms of the average number of candidates, but shows a noticeable advantage in terms of the 0.95-percentiles. In both instances, Regression LSH significantly outperforms PCA tree, and all of the above methods dramatically improve upon random projections.
3 References


