Vector Similarity Search: A Deeper Dive

Finn Dayton, Rodrigo Nieto, Oscar O'Rahilly, Eric Werner

CS168 Final Report

June 6, 2023

1 Introduction

This writeup is a continuation of the topic of similarity search from Week 2. We want to answer the following question: given a vector v_q and a large corpus of vectors V, how do we find the k "closest" vectors in V to v_q. In many cases, $k = 1$, meaning we just want to find the closest vector.

1.1 Why is this important?

Every company that makes recommendations to users must implement some form of similarity search. Every time Netflix recommends a you a list of movies, or Amazon shows you item it thinks you'll like, it is using some form of vector similarity search. If you've ever used Google's "search by image" option, how do you think Google can compare your provided image (the query) with the trillions of images it has (the database) in under a second? It's certainly not the $O(n)$ solution of comparing your image with each of its 1 trillion one-by-one. Spotify does this too: how can it reccomend the best songs (of millions) to you based on your songs? Indeed, one of the ANN (Approximate Nearest Neighbors) techniques mentioned below , ANNOY 5.2, was created by a Spotify engineer.

With the proliferation of Large Language Models (LLMs) and popularity of Chat-GPT, many of us are asking "How can I search my data more efficiently?", from Notion, Slack, Gmail to PDF documents. Can you ask a question in plain English and get a great answer? Hebbia and Glean are enterprise search companies with +$100mm in funding doing just this. You could fine tune an LLM on your data but this is impractical because your data is constantly changing, which means you'd have to re-train often. Instead, you can compress you data into lower dimensional but rich vector embeddings and then perform similarity search over your data and each new query.

Thus, both these problems–recommendation systems and semantic search–rely on fast similarity search between a query vector and a dataset. The key question is: How can we beat $O(n)$ lookup performance? We discuss sub-$O(n)$ solutions in "More Advanced Techniques" 5

1.2 Outline

We start by briefly rehashing the topics already covered: k-d trees and the curse of dimensionality. We then go into more detail on locality sensitive hashing and cover some of the more recent advancements in similarity search.

2 Measures of Vector Similarity

Given two high dimensional vectors, how can we compare their "likeness"? What do these metrics actually mean intuitively and how are they different?

The good news is the choice of similarity metric does not affect the actual mechanics of the similarity search itself–it only affects the *meaning* of the results we find. For more comprehensive overview of different similarity metrics and their trade offs, see the Week 2 notes for CS168. [1]

2.1 Jaccard Similarity

The Jaccard Similarity between two sets (or binary vectors) A and B is defined as the size of the intersection divided by the size of the union of the two sets. It is often used to compare the similarity and diversity of sample sets. The Jaccard Similarity can be represented as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

Jaccard Similarity is used mainly with sets or binary vectors and is well suited to situations where presence or absence of a feature is more significant than its frequency.

2.2 Cosine Similarity

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. It is given by the formula:

$$cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} \tag{2}$$

This metric is effectively a normalized dot product, and is used often in text analysis and summarizing and other areas dealing with raw frequencies. Intuitively, it measures how similar the directions of two vectors are, with -1 being opposite, 0 being perpendicular, and 1 being parallel.

2.3 L_2 Distance (Euclidean)

The L_2 distance, also known as Euclidean distance, is a specific case of the more general Minkowski (L_p) distance where $p = 2$. It is a measure of the straight line distance between two points in a space. In the context of high-dimensional vectors, it is given by the formula:

$$L_2(A,B) = \sqrt{\sum_i (a_i - b_i)^2} \tag{3}$$

Euclidean distance is widely used in machine learning and data science because of its simplicity and interpretability. It measures the ordinary straight-line distance between two points, making it intuitive and straightforward for most applications.

In summary, Jaccard similarity is best suited to comparing sets or binary vectors where absence or presence of a feature is key, while cosine similarity is often used for frequency data like text. Euclidean distance, unlike Cosine Similarity, takes into account the magnitude of the vectors, meaning that longer documents will have larger Euclidean distances to other documents compared to shorter ones, even if they share the same word usage patterns. Therefore, for tasks where the length of the documents or the absolute frequencies of words is important, Euclidean distance might be more appropriate.

3 K-D Trees

Low dimensional (<25 dims) vector similarity search is solved quite handily by k-d trees in sub-$O(n)$ time. Introduced by a Stanford undergrad, Jon Bentley, in 1975, a k-d tree is a clever way to place multidimensional vectors into a binary search tree by looking at one dimension at a time. Each interior node in the tree selects one dimension (often at random) to compare incoming vectors on. The node sets a threshold value based on the median value of that index for the data set. At lookup time, the algorithm finds the leaf node the query vector would have been placed at, but then also checks nearby nodes to see if the true closest vector was placed somewhere else. k-d trees work well on low-dimensional vectors (<25 dims), or when the number of total vectors exceeds 2^d. Lookup time typically scales exponentially with the dimension, however, rendering k-d trees impractical for modern vector similarity search with vectors often exceeding 1000 dimensions.

4 Curse of Dimensionality

The shortcomings of k-d trees highlight a bigger, general issue in vector similarity search called the "curse of dimensionality". Coined by Richard Bellman, the phrase applies to several related fields, including data mining, combinatorics and–in our case–distance functions. Here, it refers to the fact that the number of "close" vectors to a given vector grows exponentially with d, their dimension. The Week 2 notes present an interesting thought experiment: "What is the largest number of points that fit in d-dimensional space, with the property that all pairwise distances are in the interval [0.75, 1]?" [1] For d=1 (a line) it's 2 points. For d=2 (a plane) it's 3 points, forming a triangle. For d=100, the number is in the thousands.

Another intuition: imagine we have 1-d data. Two data points, v_a and v_b. They have just two possible configurations: $v_a <= 4v_b$ or $v_a > v_b$. Now imagine 2-d data. There are now four possible relationships, two for each dimension. For 3-d data, it's 8 possibilities, for 300-d data, it's 2^300, more than the number of atoms in the universe.

No known algorithm can efficiently overcome the curse of dimensionality. In other words, the *exact* similarity search algorithm cannot be improved past $O(n)$ [2]. Luckily, our vectors are not randomly generated, so some dimensions are more important than others. This leads to clustering techniques that drastically reduce run time and storage needs and work very well in practice, though always with some non-zero error.

5 More Advanced Techniques

The following are some some of the techniques used in practice to sidestep the curse of dimensionality and achieve high search times without much of a detriment to search performance.

5.1 Locality-Sensitive Hashing (LSH)

The high level idea behind LSH is to split your data into buckets. A query vector is hashed into one of these buckets, and brute-force similarity search is performed from there. If we have lots of buckets and a method to hash similar vectors into the same bucket, then this strategy works well. There are two main approaches to LSH:

- The traditional approach of shingling, MinHashing and banding [3]
- The more commonly used strategy of random hyperplanes. [4]

5.1.1 Traditional LSH

The traditional approach of shingling, MinHashing and banding is gone over in great depth in [1] and [1]. We present an abbreviated version here.

Given text, we slide a fixed-width window over the characters to create a series of “shingles", shown in Figure 1 for a window of width 2.

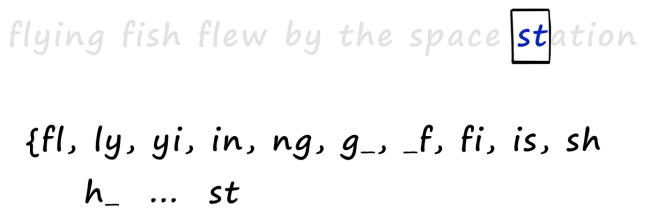
flying fish flew by the space station
st
{fl, ly, yi, in, ng, g_, _f, fi, is, sh
h_ ... st

Figure 1: Singles Example with Sliding Window of Width 2 [4]

We create a set of these shingles (remove duplicates), S, and then use S to produce a one-hot vector encoding of length $|V|$ (vocab size), with a 1 if a word in V is contained in S and a 0 if not.

Next, we need to compress this one-hot vector. To do so, we use a series of minHash functions. A minHash function is a function that assigns a number between 1 and $|V|$ to the one-hot vector. By applying N minHash functions to the one-hot vector, we get a series of N numbers that represent a compressed version of the original text. Our data has gone from a length $|V|$ one-hot vector to a length N "signature". Importantly, these "signatures" of length N preserve the distance properties of the original texts (with some loss) but are much lower dimensional than the original data.

The last step is hashing the signatures into buckets. Unlike your typical hash function, we want to *increase* the chance of a collision, so that similar signatures hash to the *same* bucket. Instead of seeing if the two signatures are equal, we break each signature into subarrays called "bands", then comparing each signature band-wise. If at least one pair of bands match, the signatures (and the corresponding original vectors), are hashed to the same bucket. More bands (smaller band size) increase the false positive rate, i.e. hashing dissimilar signatures to the same bucket. On the other hand, fewer bands (large band size) increase false negatives, i.e. similar signatures are hashed to different buckets. As the user, you must play around with band size to best fit your dataset.

So how does querying work? A query vector is assigned to a bucket by the same process. It's then compared to the each of the vectors in that bucket via the choice of similarity metric (e.g. Cosine or Euclidean) and the closest match is found.

5.1.2 LSH with Random Hyperplanes

This method is more commonly used in practice and is slightly simpler. The goal is the same: represent high-dimensional vectors using lower dimensional vectors in buckets to reduce search time.

In this method, we generate n hyper planes randomly to split up the data. Figure 2 give an illustration of several random hyperplanes created to divide the data.

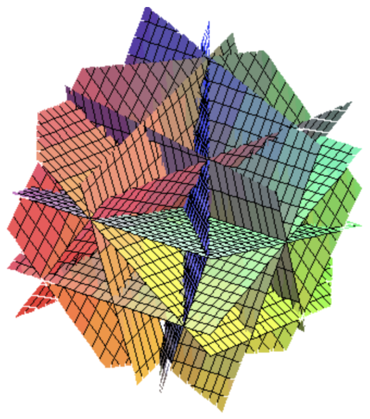

Figure 2: Random Hyperplanes to Divide the Data. [5]

Each vector lies either on one side or the other of a hyperplane. If we label the hyperplanes $1, 2, ...N$, we have 2^N places (hint: buckets) a vector can lie in. For example, say we have 4 hyperplanes, and our data is in 2d space. We have the vectors $v_1 = [1, 2]$, $v_2 = [2, 3]$ and $v_3 = [0, -1]$ and say these map to $[0, 0, 1, 1]$, $[1, 0, 1, 0]$ and $[0, 0, 1, 1]$, respectively. We build up a Python-like dictionary:

```
{
    [0, 0, 1, 1] -> {v_1, v_3},
    [1, 0, 1, 0] -> {v_2}
}
```

These binary vectors *are* the bucket ids for an input vector. At query time, a vector is converted into the binary representation of its location among the hyperplanes. If there are no vectors in that bucket, the query vector is matched with the closest other bucket by the Hamming distance:

Hamming distance, denoted as d_H, is a measure used to quantify the difference between two binary vectors. It is defined as the number of positions at which the corresponding values are different. For two binary vectors u and v, the Hamming distance can be calculated as follows:

$$d_H(u, v) = \sum_{i=1}^{n} |u_i - v_i| \tag{4}$$

Where n is the length of the binary vectors. As an example, consider the binary vectors $u = 1010$ and $v = 1001$. In this case, the Hamming distance $d_H(u, v)$ is equal to 2, because the two vectors differ in the second and fourth positions.

It should be clear that increasing the number of hyperplanes increases the granularity for the compressed vectors and achieves better quality results. The limitation of arbitrarily increasing the number of hyperplanes, however, is that the size of the dictionary grows untenable and approaches the original dataset size both in number of entries and in the dimension of the binary vectors.

5.2 Annoy (Approximate Nearest Neighbors Oh Yeah)

We now transition away from a pure theoretical discussion of approximate nearest neighnor search and look at some existing libraries that implement it. The first is Approximate Nearest Neighbors Oh Yeah (ANNOY), an efficient library for similarity search in high-dimensional spaces implemented in C++. The core idea behind ANNOY, like other Approximate Nearest Neighbor (ANN) methods, is to initiate the search over a narrowed subset of candidate points.

ANNOY was developed by Erik Bernhardsson, an engineer at Spotify, in 2015. It implements a strategy of random projections and leverages tree-based data structures to efficiently partition the data space and facilitate rapid querying. Uniquely, the choice of the hyperplane at each node in the tree is determined by randomly sampling two points from the subset of points within that node and opting for the hyperplane that is equidistant to these two points. This approach produces a complex tree structure that makes lookups lighting fast. Figure 3 illustrate the division of space by the hyperplanes, resembling that of the LSH hyperplanes.

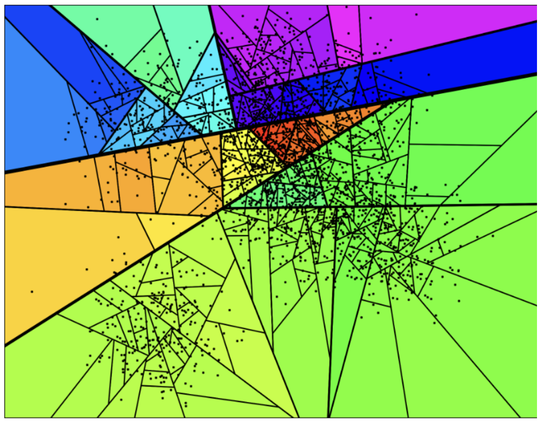

Figure 3: ANNOY Partitioning Scheme [6]

To bolster redundancy and consequently enhance the robustness of the search process, ANNOY constructs multiple such trees. It exhibits high performance on vectors spanning dimensions from as low as 100 to as high as 1000. [7]

As a C++ library, ANNOY strikes an impressive balance between speed and precision, competing favorably with some of the fastest libraries in the field while maintaining a high degree of accuracy. ANNOY supports Euclidean distance, Manhattan distance, cosine distance, Hamming distance, and Dot (Inner) Product distance. Further, ANNOY builds its index on disk, allowing it to index massive datasets that exceed the capacity of memory, a feature indispensable in handling large-scale real-world datasets.

ANNOY provides critical parameters for tuning performance according to specific requirements: N_trees and Search_k [7]. N_trees controls the number of trees to be built; increasing this number augments the index size and consequently the storage space. Search_k dictates the number of nodes to inspect during search; a larger Search_k enhances the accuracy of the search, while a smaller one accelerates the search process. These tunable parameters offer users the flexibility to optimize the trade-off between accuracy and speed as per their specific use-case demands.

5.3 FAISS (Facebook AI Similarity Search)

FAISS is an library for efficient similarity search and clustering of dense vectors released by researchers at Facebook in 2017. [8] It takes the strategies discussed earlier in this paper as well as many more and implements them in efficent C++. The value FAISS is not in theoretical breakthroughs in similarity search, but instead in extremely efficient code and multi-GPU support to speed up search. Faiss GPU is typically 5-10x faster on a single GPU than the corresponding Faiss CPU implementations. [8]

Without diving into too much detail (see the full paper [8]), FAISS uses optimized versions of fundamental techniques such as multi-threading, BLAS libraries for efficient exact distance computations, machine SIMD vectorization, and popcount.

FAISS offers a variety of index options, the following give an overview. For more details, go to [9].

5.3.1 HNSW (Hierarchical Navigable Small World Graphs)

HNSW is an extension of navigable small world graphs [10] (NSW), a graph structure where vertices are linked to their nearest neighbors via edges. The parameters for HNSW setup include M, which denotes the number of connections each vertex will possess, ef_search, which represents the depth of layers explored during search, and ef_construction, indicating the depth of layers explored during index construction. [11] Notably, ef_search and ef_construction parameters do not influence the index memory footprint.

5.3.2 Inverted File Index (IVF)

Inverted File Index (IVF) partitions the data into specific regions called Voronoi Cells, a method extremely similar to LSH shown above. Figure 4 shows a visualization of this.

5.3.3 Product quantization

Product Quantization is another vector compression technique, similar to that of LSH. The implementation by FAISS is grounded in Optimized Product Quantization by [12] Figure 4 shows how product quantization can be combined with IVF for fast lookup times. A query vector is matched with the closest Voronoi cell centroids. The number of centroids compared with is controlled by the *nprobe* parameter to the FAISS index. Figure 4 shows a setup with $nprobe = 2$.

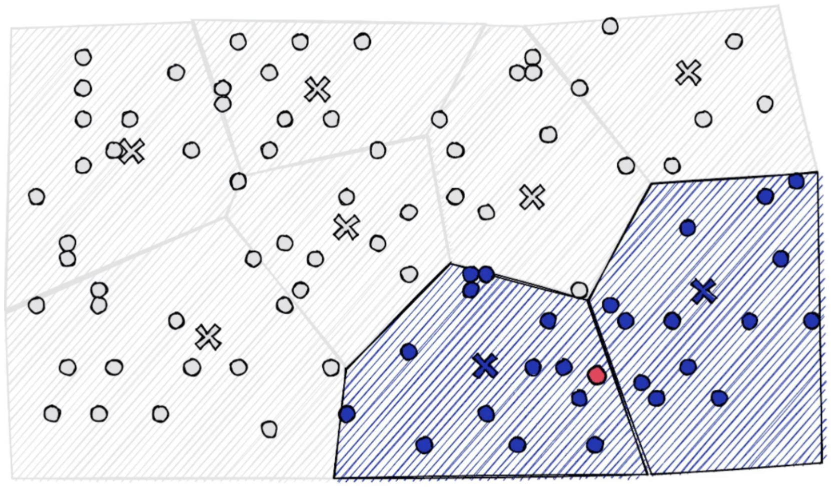

Figure 4: Voronoi Partioning + Product Quantization [13]

PQ is a clustering technique to reduce the storage footprint of your vectors. It produces centroids by chopping the vectors into subarrays (similar to bands) and then assigning each band to its own, lower dimensional cluster. Each cluster has an id, and this id replaces the entire band in final array. This produces massive space savings, while preserving accuracy of similarity comparisons.

Whereas the memory complexity of k-means is kD, where k is the number of clusters and D is the dimension of the input data, product quantization can do it in $k^{1/m}D$, where m is number of subvectors that we split our vectors into. [13]

6 Current Frontier Of Research

Now that we've seen some existing techniques and libraries for similarity search, we briefly highlight the current frontier of research. This can be broken broadly into two categories: software and hardware. The former includes designing transformers to make lower-dimensional embeddings of the sentence data. This is the approach taken by SBERT, a sentence transformer library. [14]. This library produces rich embeddings that can then be paired with some form of bucketing to get sub $O(n)$ lookup times. As far as we know, LSH, and variations of it, is still the best sub-$O(n)$ lookup scheme.

On the hardware side, optimizing these bucketing / clustering algorithms to run on GPUs and multi threading is name of the game. FAISS was a huge leap forward and is currently (2023) the largest open-source library for efficient hardware implementation of similarity search [8]. Private companies, like Pinecone and Weaviate, however, have their own internal algorithms available only via their APIs.

The future winners in vector similarity search will combine novel theoretical algorithms—likely improving on an LSH strategy—with extremely efficient GPU implementations.

References

[1] Tim Roughgarden and Gregory Valiant. Cs168: The modern algorithmic toolbox lecture #3: Similarity metrics and kd-trees, 2023. Accessed: 2023-06-05.

[2] Joachim Wolff. Approximate nearest neighbor query methods for large scale structured datasets. Master's thesis, University of Freiburg, 2016.

[3] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Stanford University, California, 2020. Chapter 3: Finding Similar Items.

[4] Pinecone. Random projection for locality sensitive hashing. Accessed: 2023-06-05.

[5] Louisiana State University. Hyperplanes. Accessed: 2023-06-05.

[6] Erik Bernhardsson. Nearest neighbors and vector models – part 2 – algorithms and data structures, 2015. Accessed: 2023-06-05.

[7] Erik Bernhardsson. spotify/annoy. `https://github.com/spotify/annoy`. Apache-2.0 License.

[8] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus, 2017.

[9] Jeff Johnson, Matthijs Douze, and Hervé Jégou. facebookresearch/faiss. `https://github.com/facebookresearch/faiss`. MIT License.

[10] Yu. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs, 2018.

[11] Pinecone. Hierarchical navigable small worlds (hnsw). Accessed: 2023-06-05.

[12] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence*, 36, 12 2013.

[13] Pinecone. Product quantization: Compressing high-dimensional vectors by 97%. Accessed: 2023-06-05.

[14] Nils Reimers. Sentence transformers: Multilingual sentence, paragraph, and image embeddings using bert co. https://github.com/UKPLab/sentence-transformers, 2019. Apache 2.0.