My First Adventures in Similarity Search

Luke Kerbs
Jul 9 · 9 min read

For my first project as an intern at GSI Technology, my mentor assigned me with investigating ***similarity search;*** a general term used to describe the various techniques for finding similar items in very large data sets. To illustrate, take *image similarity search* as an example: if Google has a massive quantity of images and a user queries an image of her labradoodle, how does Google quickly narrow billions of images down to the top ten images resembling her labradoodle? It was interesting to explore the available solutions to such a seemingly complex problem.

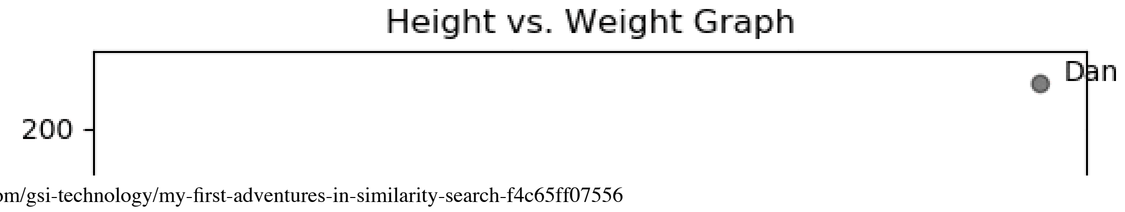
Introduction to Similarity Search

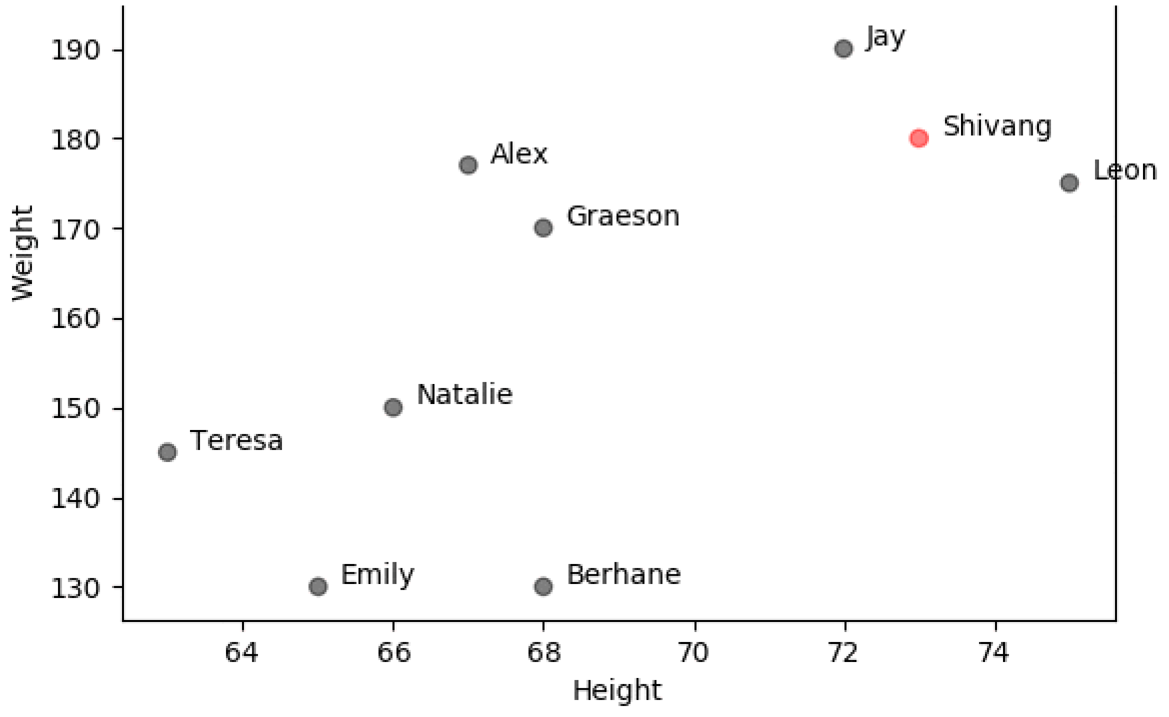
Suppose you are at a coffee shop that has *ten patrons* and you decide to conduct a study — You decide to record two pieces of information, or two *features*, from each of the ten customers:

1. **The height of each customer**
2. **The weight of each customer**

After conducting your survey, you have determined that Shivang is 73 inches tall, weighing in at 185 pounds, Leon is 75 inches tall, weighing in at 175 pounds, etc …

Now that you have collected your data, you can plot out all ten of the data points in a 2-D graph that corresponds with each customer's height and weight information:





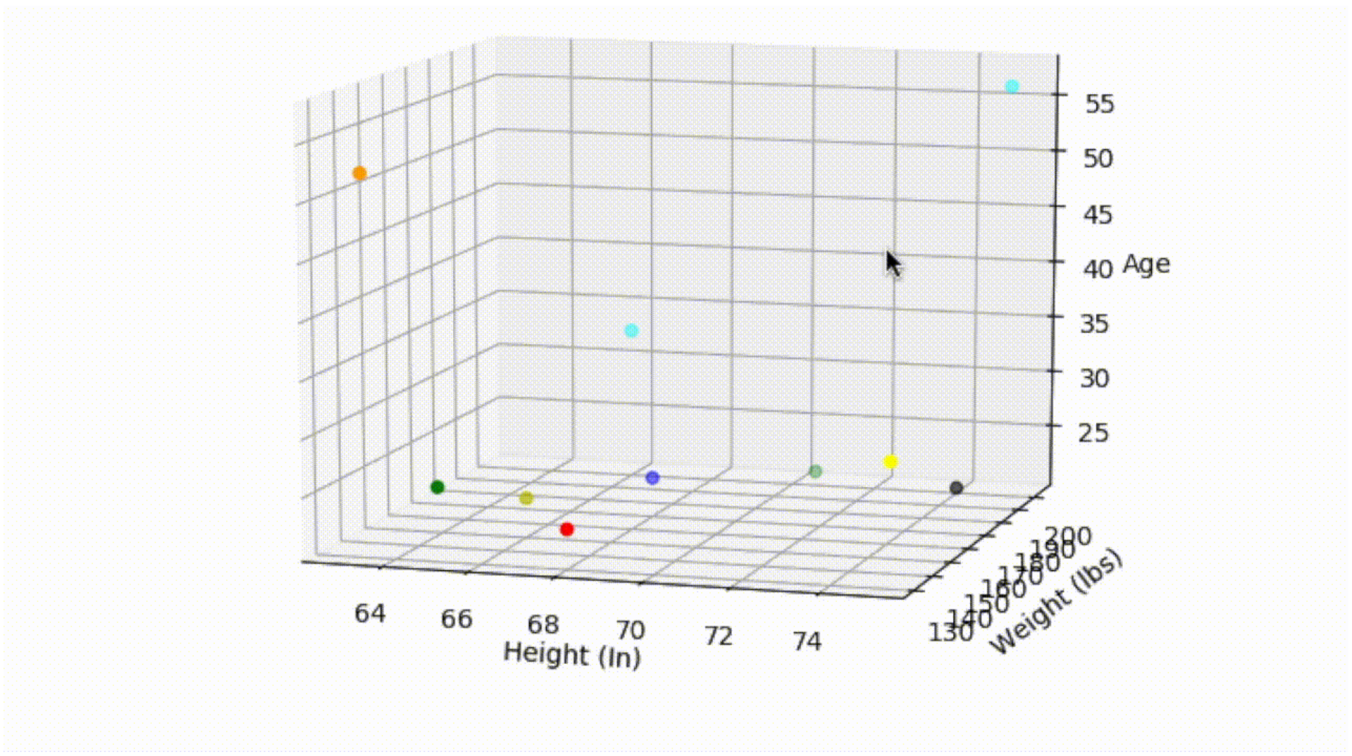
Nearest Neighbors: Height vs Weight Graph

Then, in theory, the *closer* two customers are plotted in this 2-D graph, the more similar they are based on their features. This makes intuitive sense. Take Shivang as an example, represented by the red data point in the graph above.

Say that you wanted to determine the three *most similar* coffee shop customers to Shivang. To accomplish this, you could compute and record the **Euclidean distances** between Shivang's red data point and the remaining nine customer data points. The smallest three recorded distances correspond with the three most similar customers to Shivang. As shown in the graph above, ***the three most similar customers to Shivang are Jay, Dan, and Leon.***

This method is called an ***exhaustive search*** because the query (Shivang's datapoint) is compared with *every* other datapoint in the data set.

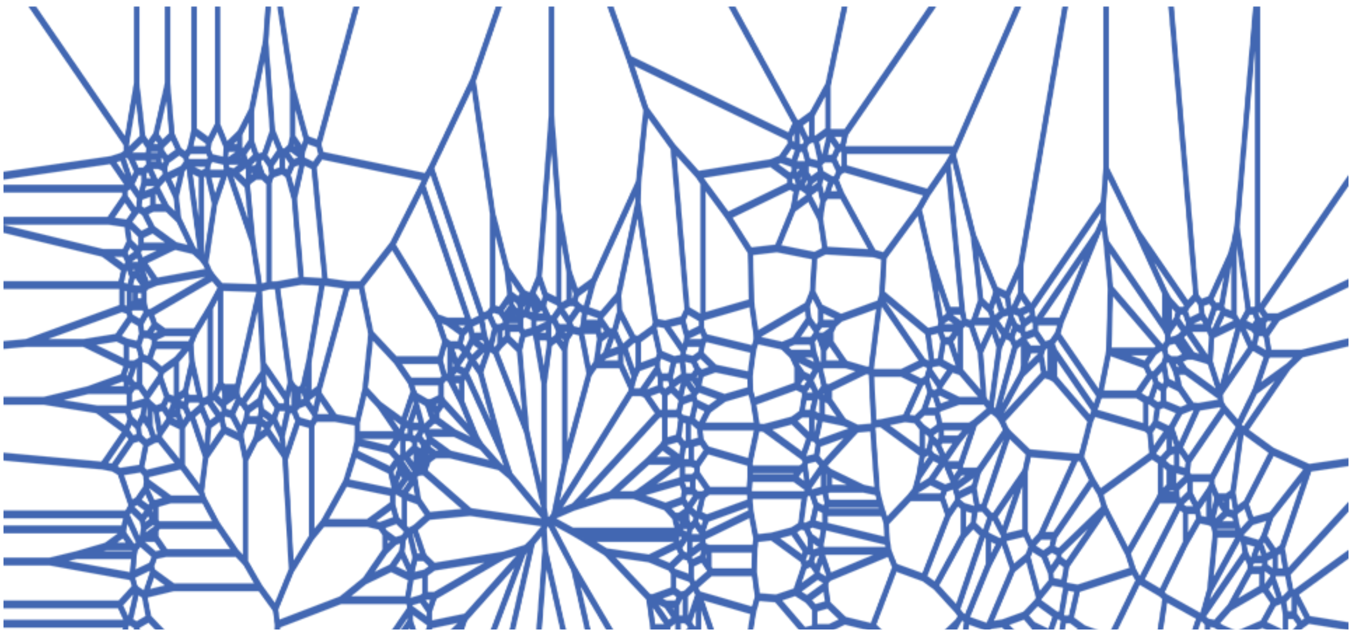
We could extend this two variable case even further — Suppose that in *addition* to height and weight, we recorded the age of each customer and plotted this additional information on a third axis. With the additional information we can now determine the distance between Shivang and any other customer in *3-dimensional space*. And this can be taken as far as you like, by recording as many features from each of the customers as you wish. The data is not so easily visualized once we start going beyond the third dimension, but the principle relationship between distance and similarity remains the same.



3D Graph: Height by Weight by Age

That being said, the above example of measuring height, weight, and age is somewhat contrived and is mainly useful for highlighting the relationship between *similarity* and *distance* between points in a dataset. In modern day applications, the dimension size will be very large and the features will usually be *latent* — meaning they are inferred and not directly observable. See **feature learning**.

This basic concept is applied on massive datasets in dimensions of hundreds or thousands. Performing an exhaustive search on massive datasets is expensive, time consuming, and impractical for most use cases. Luckily, there exists a tool to speed up the search process …



FAISS

Facebook Artificial Intelligence Similarity Search

Facebook Artificial Intelligence Similarity Search (FAISS) is a C++ / Python library developed by Facebook Research that provides several built-in functions for implementing fast similarity search on CPU or GPU memory.

Most of the similarity search methods from FAISS use a compressed representation of the data in order to speed up the search process. While this can lead to less precise results, the resulting payoff in memory storage and time saved can greatly out-weigh a marginal loss in search precision.

If you want to try using FAISS, first make sure you have Anaconda installed on your computer, the rest is simple. Use the following command in terminal:

```
conda install faiss-cpu -c pytorch
```

FAISS is relatively easy to use. Simply load up your dataset, choose an index, run a training phase on your data, and add your data to the index. Your data is now ready for fast search. This can all be accomplished in as little as ten lines of Python code:

```
import faiss

data = read_dataset('data_set.fvec')                    # dataset
queries = read_queries('query_set.fvec')                # queries
d = 128                                                 # dimension
k = 100                                                 # k-NN size
nlist = 100                                             # clusters

quantizer = faiss.IndexFlatL2(d)                        # quantizer
index = faiss.IndexIVFFlat(quantizer, d, nlist)         # index
index.tran(data)                                        # train
index.add(data)                                         # add dataset

D, I = index.search(queries, k)                         # search
```

For detailed instructions on how to use FAISS, please visit Facebook Research's Github: **Tutorial.**

Approximate Nearest Neighbor Search

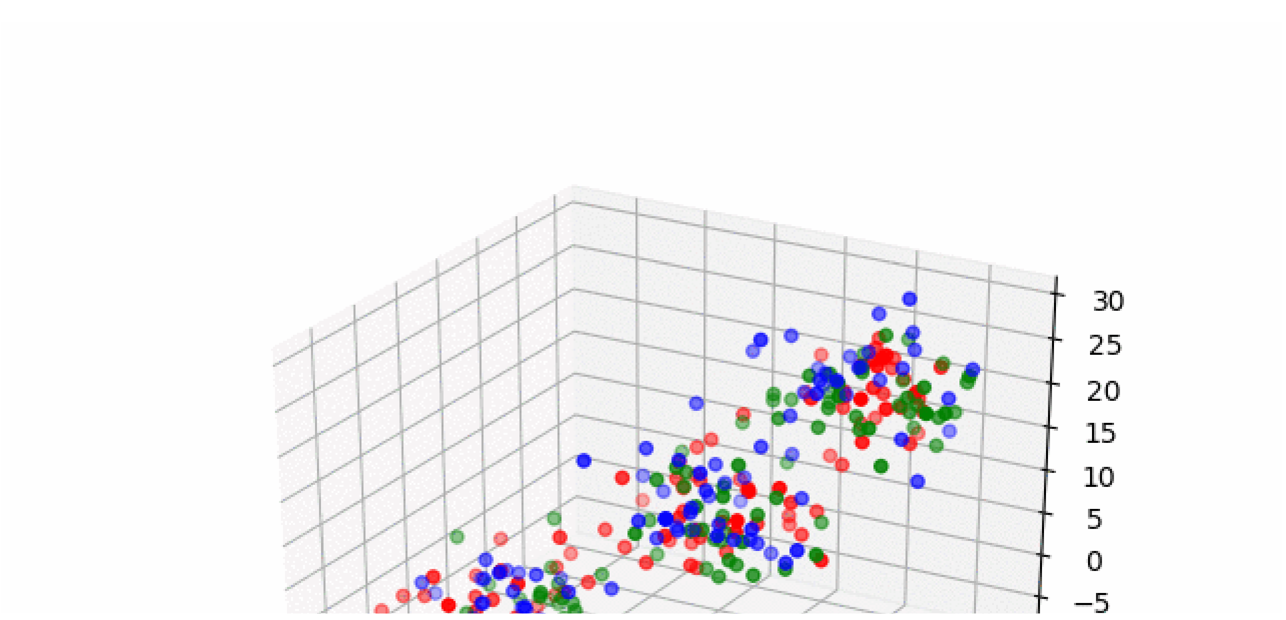
In the coffee shop example from above, we used a **k-nearest neighbors** exhaustive search to determine the k=3 most similar customers to Shivang. As alluded to earlier, an exhaustive search on the entire dataset is not always desirable given time and memory constraints.

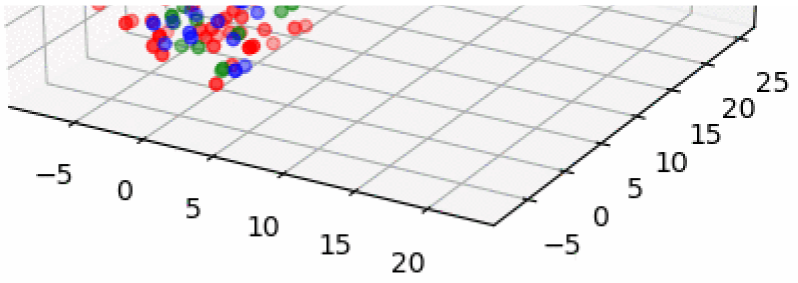
The solution: *Approximate nearest neighbor search.*

What if instead of comparing Shivang to all nine of the remaining customers, there was a way to narrow down your search to only a small portion of the customers where the k-nearest neighbors are most likely to be found? This is called an **approximate nearest neighbors search (A-NN)**, and FAISS has several ways of implementing it.

FAISS has built-in *indexing* methods that can preprocess your data in such a way that makes search more efficient. The basic idea is simple: if you have a list of names, for example, you could *index* them in alphabetical order to make searching for an individual name faster. If you want to find *Jessica's* folder, you would look in the file cabinet labeled *J*. Instead of names, however, FAISS has built-in techniques for indexing data points from a data set in such a way that it makes similarity look ups faster. How is this accomplished?

The code example from earlier uses FAISS' **IVFFlat index**, a method that implements the **k-means algorithm** to partition the data set into k clusters of data points as seen in the following illustration:





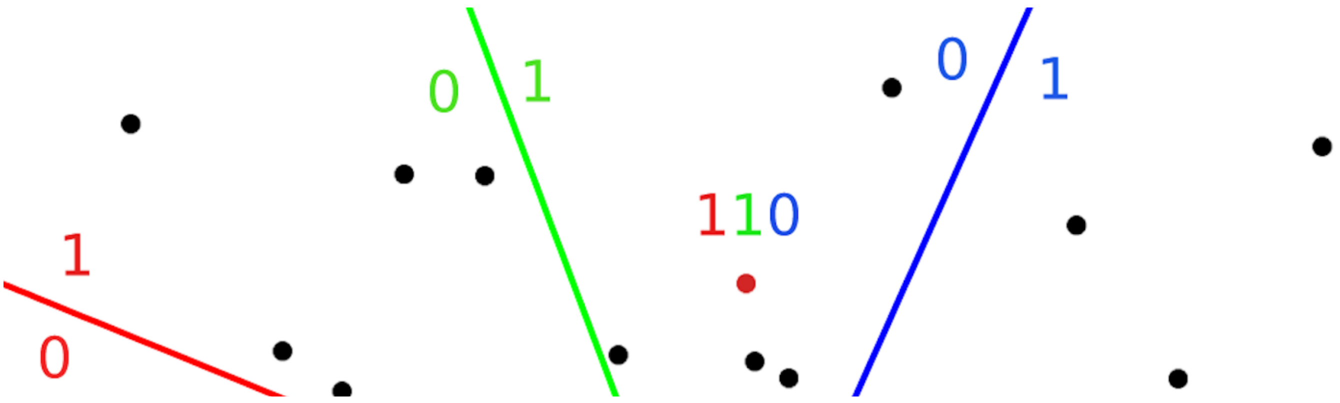
K-Means Algorithm

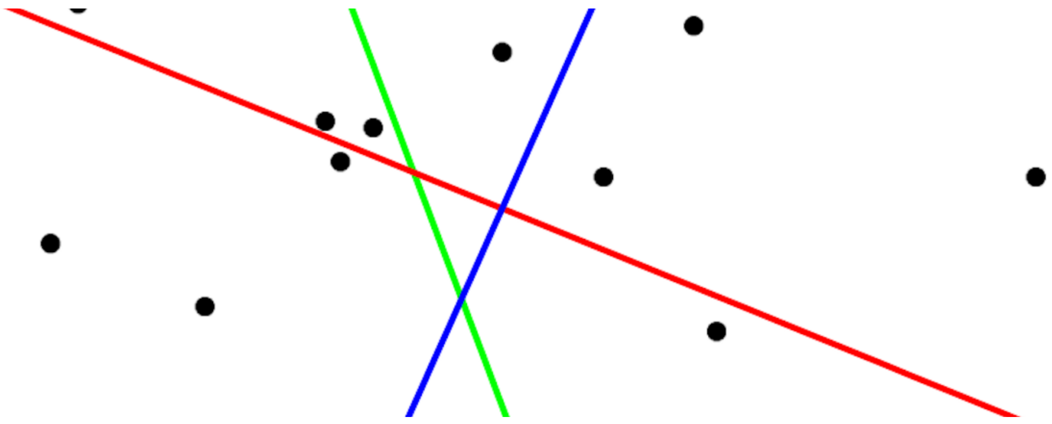
Here, the data has been indexed into three clusters: *red*, *blue*, and *green*. Now let's suppose that Shivang's data point fell in the *green* cluster from the above illustration. Instead of calculating and comparing all the distances across the entire data set to Shivang, you could narrow your similarity search to only the green cluster since it is likely to contain the nearest neighbors. Assuming an even distribution of data points, you have narrowed down your search by 66 percent! This is the magic of the indexer.

Of course, this method is not 100 percent accurate, this is why it is called an *approximate* search. The IVFFlat index has a parameter called *n-probe* that allows you to adjust the number of nearest clusters to *probe* for a potential nearest neighbor. Although in three dimensions it may appear that a single n-probe would be sufficient to achieve high retrieval performance, this is usually not always the case when we start to work in higher dimensions; a perplexity of the **curse of dimensionality**.

Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is another index provided by FAISS to solve approximate nearest neighbor searches in high dimensional spaces. This technique assigns similar data points into the same categories with high probability. Since similar data points end up in the same category, LSH can be used for data clustering. This technique compresses high dimensional vectors into vectors that are much smaller and require less computational work to calculate relative distances.





Local Sensitive Hashing (LSH)

To illustrate how LSH works, consider the scatter plot above. In this simple example we randomly *slice* up the data in three ways as represented by the *red*, *green*, and *blue* lines. The opposite sides of each line are then assigned a 1 and a 0 respectively. Now consider the red data point: it is labeled *110* because it lies on the *1-side* of the red line, the *1-side* of the green line, and the *0-side* of the blue line. Then while searching for nearest neighbors, we will only compare our query with vectors that have the same binary encoding, in this case: 110.

Every data point in the graph can be assigned its own three value binary encoding based on its location just like the red data point. This same principle can be extended to data in higher dimensions. FAISS includes other indexing methods other than IVFFlat and LSH, but these are the two that I happened to look at in detail.

Comparing Search Performance

I tested FAISS on my laptop CPU with the **Sift128 dataset** of 10,000 128-dimensional vectors to get a sense for how well the LSH and IVFFlat indexes performed. There are two important things you want to look at when evaluating similarity search techniques:

1. **The time it takes to complete a search**

2. **The accuracy of the search**

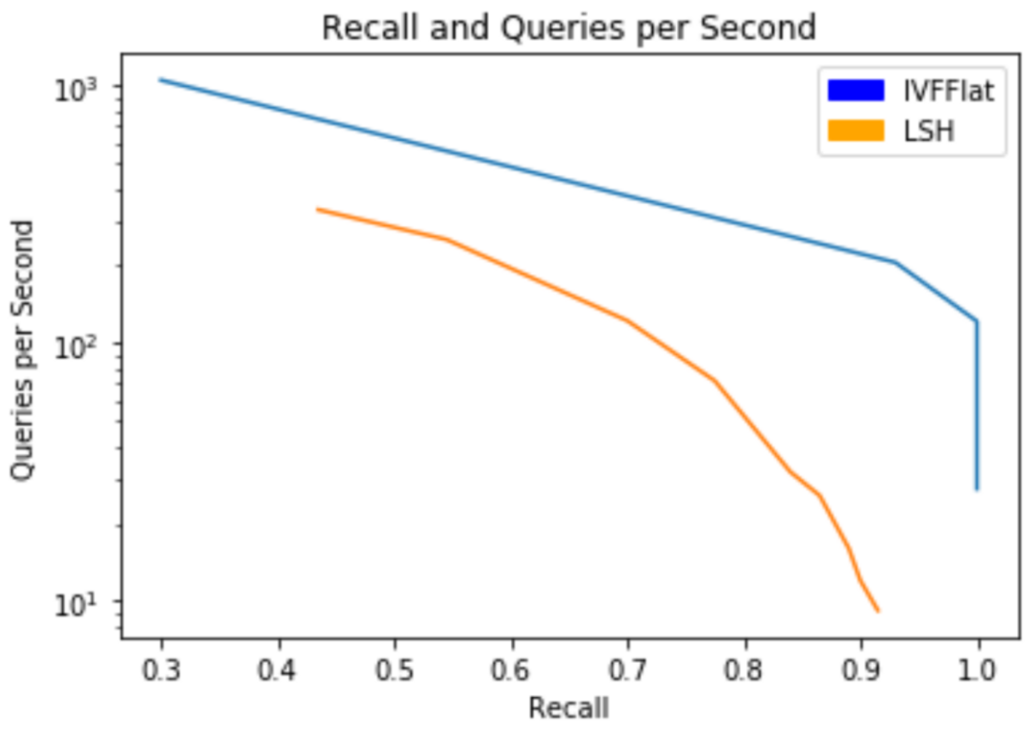
The way we measure the speed of a search is by the number of **queries completed per second**. This is a helpful metric for large scale applications where throughput efficiency is crucial. See **latency and throughput**.

There are several ways to gauge the accuracy of a similarity search technique. One popular metric for search algorithm retrieval performance is called ***recall:***

Recall = True Positives / (True Positives + False Negatives)

In the context of similarity search, a *true positive* is a correctly identified nearest neighbor and a *false negative* is a nearest neighbor that was not correctly identified as a nearest neighbor.

Here is a graph that compares the performance of the LSH and IVFFlat indexing techniques from my laptop experiment with FAISS. We measure retrieval performance in terms of *recall* and *queries per second.*

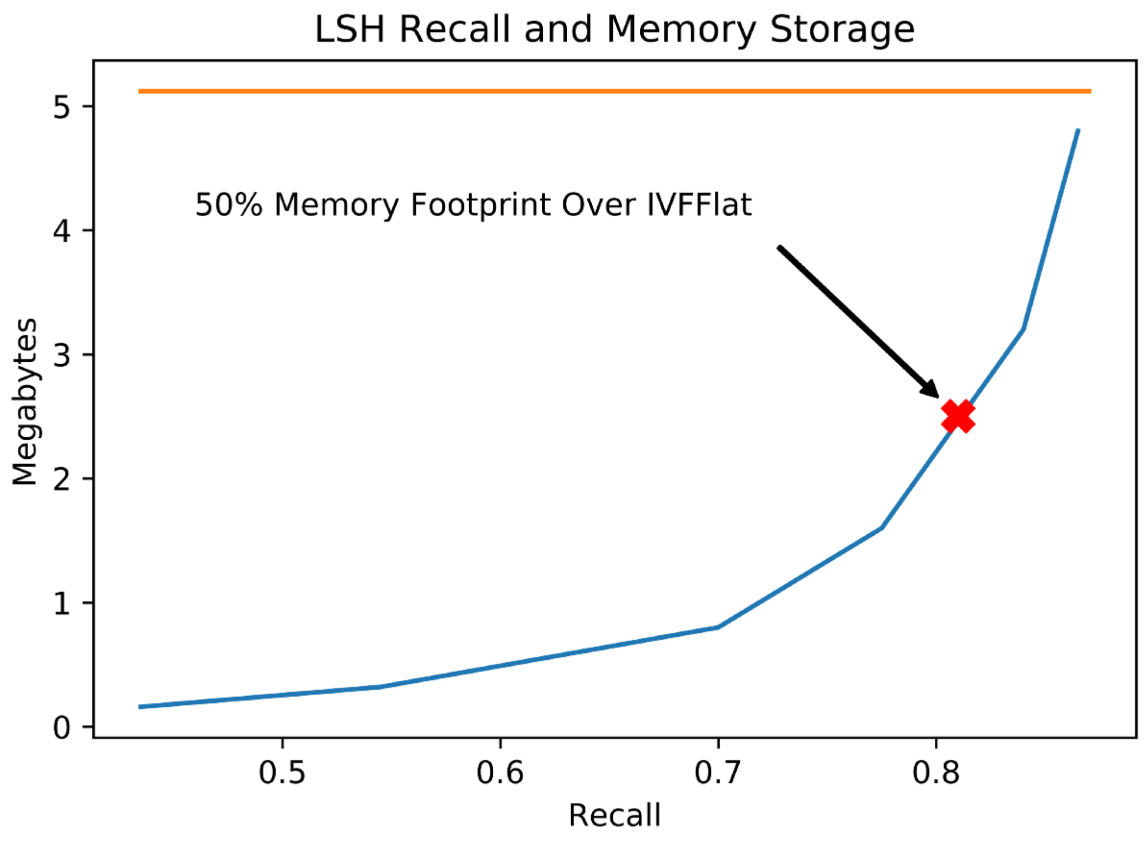


Comparison of IVFFlat and LSH on my laptop

Observe the similar trend between both indexes: as recall *increases* the number of queries made per second *decreases*. In other words, there is a trade-off between search speed and search recall. It will depend on the particular use-case when deciding where to balance the trade-off between speed and retrieval quality.

It is important to note that different indexes may produce different performance results depending on the data volume, processor, and general testing environment. After running the IVFFlat and LSH indexes on the Sift128 dataset with a query set of 100 vectors, it became clear that IVFFlat out-performs LSH when it comes to speed, for this smaller dataset. For any given recall, IVFFlat is able to perform more queries per second. Of course, there are other performance metrics that one should consider when comparing index performance, these being training time and memory storage among others.

In terms of memory storage FAISS only supports reading from RAM, so the size of the database plays an important role in choosing the right indexing technique. LSH is one of several indexes included by FAISS that has a built in compression algorithm, allowing the user to search through larger databases than previously possible.



This Graph Shows the Potential Savings in Memory of LSH over IVFFlat

Consider the graph above which displays the results from another FAISS experiment I ran on my laptop. The blue line represents the relationship between LSH recall and memory and the orange line represents the relationship between IVFFlat recall and memory. Notice that LSH provides an opportunity for large savings in memory costs if very high recall is not required.

In my next blog we will explore these trade-offs even further. I will also explore other performance metrics like training time. And we will perform these experiments on more powerful hardware.

Data Science Similarity Search Lsh Faiss Python