EE392A: Undergraduate Research project
Vector Search Algorithms

Sonal Agrawal

Supervisor: Prof. Vipul Arora

April 23, 2023

Contents

Abstract

Vector search algorithms are a class of algorithms used to search and retrieve high-dimensional vectors in a dataset efficiently. These vectors can represent various types of data, such as images, videos, text, or audio. The goal of vector search algorithms is to find the nearest neighbors of a query vector in a dataset, where nearest neighbors are defined as the vectors that are most similar to the query vector.

In this paper, we look at various vector search algorithms. We get a comprehensive overview about their principle, implementation, benefits. The algorithms are evaluated on large scale real-world database. The paper concludes by listing the possible future improvements that can be made in our approach.

1 Introduction

In recent years, the search of audio vectors using vector search algorithms for nearest neighbor search has gained increasing attention. Finding the nearest neighbors of an audio query vector in a high-dimensional dataset is a crucial task for various applications, including speech recognition, music retrieval, and audio recommendation systems. Vector search algorithms provide an efficient way to solve this problem by searching through the dataset and identifying the audio vectors that are closest to the query vector.

In this report, we explore several state-of-the-art vector search algorithms for audio nearest neighbor search, including locality-sensitive hashing (LSH), product quantization (PQ), optimized product quantization (OPQ), hierarchical navigable small-world graphs (HNSW), and scalable nearest neighbor (ScaNN). We evaluate the performance of these algorithms on the SIFT1M dataset and compare their strengths and limitations.

Additionally, we discuss potential future improvements to vector search algorithms for audio search, such as the use of graph based approaches and the incorporation of audio-specific features. Overall, this report provides a comprehensive overview of the current state of vector search algorithms for nearest neighbor search and identifies potential avenues for future research.

2 SIFT1M dataset

The SIFT1M dataset is a widely-used benchmark dataset in the field of vector search algorithms. It consists of one million SIFT descriptors, which are high-dimensional vectors that represent distinctive features of images. The dataset is designed to test the performance of algorithms for nearest neighbor search in a high-dimensional space, where traditional search methods can become prohibitively slow. The SIFT1M dataset is particularly relevant for testing vector search algorithms because it is large enough to pose a significant challenge, but small enough to be stored in memory on a standard computer. Additionally, the SIFT descriptors are widely used in computer vision applications, making the dataset useful for evaluating algorithms for real-world scenarios. The SIFT1M dataset has been used to test a variety of vector search algorithms, including locality-sensitive hashing (LSH), product quantization (PQ), and hierarchical navigable small-world graphs (HNSW), and has become a standard benchmark in the field.

3 Need for vector search algorithms

Specialized vector search algorithms for nearest neighbor search are needed because traditional search techniques can become prohibitively slow in high-dimensional spaces. In traditional search techniques such as linear search or brute-force search, each item in the dataset is compared to the query vector, which becomes increasingly computationally expensive as the dimensionality of the dataset increases. This problem is known as the "curse of dimensionality" and can make traditional search techniques infeasible for large datasets. Vector search algorithms address this problem by using specialized data structures and algorithms that are optimized for high-dimensional spaces.

Algorithm General Vector Search Algorithm

- Load the Vector Database
- Select the choice of Algorithm
- Select the number of closest neighbours to return
- Build the Searcher/Codebook accordingly
- Ask for the Query Vectors
- Implement the Search for each vector
- Calculate the accuracy of the code by matching the cosine similarity between the query vector and neighbour vector

4 Product Quantization

4.1 Understanding the Algorithm

Product quantization is a popular method for approximating vector quantization in high-dimensional space. This algorithm allows us to represent high-dimensional data in a compressed form, reducing storage requirements and computation time.

Benefits of Product Quantization:

1. **Reduced memory consumption:** The product quantization algorithm reduces the storage requirements for high-dimensional data by compressing the data into a small set of integers.
2. **Fast retrieval:** Since the compressed representation of the data can be stored in memory, retrieval of data becomes faster, as we only need to compare the compressed representation of the query with the compressed representation of the stored data.
3. **Scalability:** The product quantization algorithm scales well with the size of the dataset and the number of dimensions of the data.
4. **Improved accuracy:** The product quantization algorithm has been shown to be accurate for a wide range of datasets.

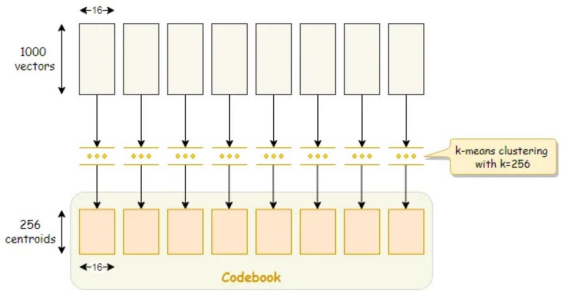
4.2 Terms and Terminologies

1. **D:** The dimensionality of the original high-dimensional vectors.
2. **m:** The number of subvectors into which a high-dimensional vector is partitioned. 'D' must be divisible my 'm'.
3. **k:** The number of codewords or centroids in each codebook. It is generally a power of 2.
4. **Codebook:** A set of discrete values or codewords that are used to represent the original continuous data. In PQ, each subvector is quantized using a separate codebook.
5. **Subvector:** A low-dimensional vector obtained by partitioning a high-dimensional vector into multiple disjoint subvectors. In PQ, a high-dimensional vector is split into multiple subvectors, and each subvector is quantized separately.
6. **Centroid:** The average value of a set of data points in a high-dimensional space. In PQ, the codebook is represented by a set of centroids, and each subvector is quantized to the index of its nearest centroid.
7. **PQ_code:** PQ code is compact binary code table obtained by dividing the high-dimensional vector into multiple low-dimensional subvectors and quantizing each subvector separately using a set of codebooks.
8. **Distance Table:** A precomputed table that stores the distances between subvectors and their corresponding codebook entries. During the encoding process, each subvector is compared to all the codebook entries in its corresponding subcodebook, and the distance between each subvector and its nearest codebook entry is recorded in the distance table.

4.3 How does PQ work?

The product quantization algorithm can be divided into three main steps:

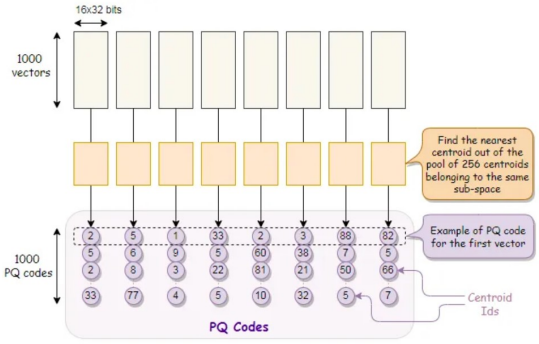
1. **Splitting the input vector and quantization of sub-vectors:** We first start by dividing our D dimensional vector into m subvectors. Let us name the chunks from 0 to $m-1$. Let us take the first subvector of each vector and call it codebook C_0. Now, we will use $k-means$ clustering to find k clusters for this set of subvectors. We will add these set of centroids (also called reproduction/reconstruction values) to the codebook C_0 and name each centroid as $C_{0,0}, C_{0,1}, .., C_{0,k-1}$. This process would be repeated for each subvector $i \in [m]$. The centroids are represented by $C_{i,j}$ where i is our subvector identifier, and j identifies the chosen cluster.



2. **Encoding:** Next, we encode each of the vectors with a different code (cluster index, i.e. 0 to $k-1$) by looking for the closest centroid in the codebook for each of the subvector, j. Here, we use L2-norm to calculate the distances.

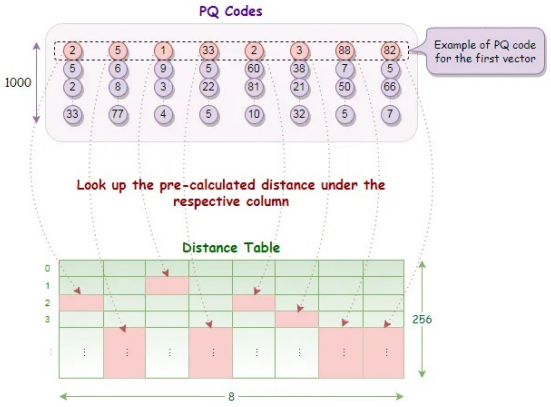
$$\mathbf{k} \sum_{i=1}^{D^*} [x_j^i - C_{j,k}^i]^2$$

Now, our D dimensional vector has been reduced to just m dimensions where each dimension represents the cluster number to which that subvector belongs. We repeat this for each vector to build a code library called PQ_Code.



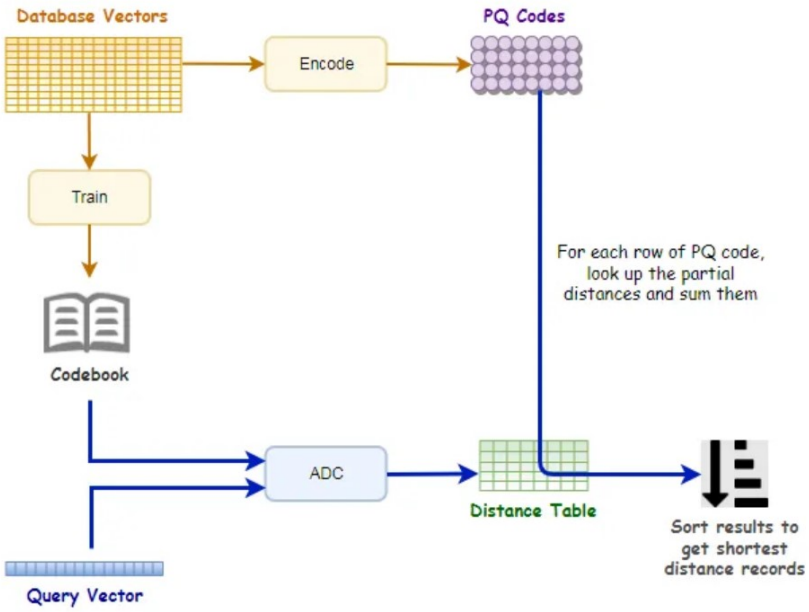
3. **Query Vector Search:** The final step is to search a query vector using this method. We take a query vector and divide it into similar subvectors, and for each subvector, we pre-calculate the partial squared Euclidean distance with all the centroids of the same subvector from the codebook. We store these results into a matrix, M of size $k \times m$. Now, we can obtain the distance from each vector by summing up the partial distances of each row according to the PQ_Codes. The distance of $(p+1)^{th}$ vector from the query vector is given by,

$$distance[p] = \sum_{j=0}^{m-1} M[\text{PQ\_Code}[p, j], j]$$



Algorithm 2 Logic behind Product Quantization

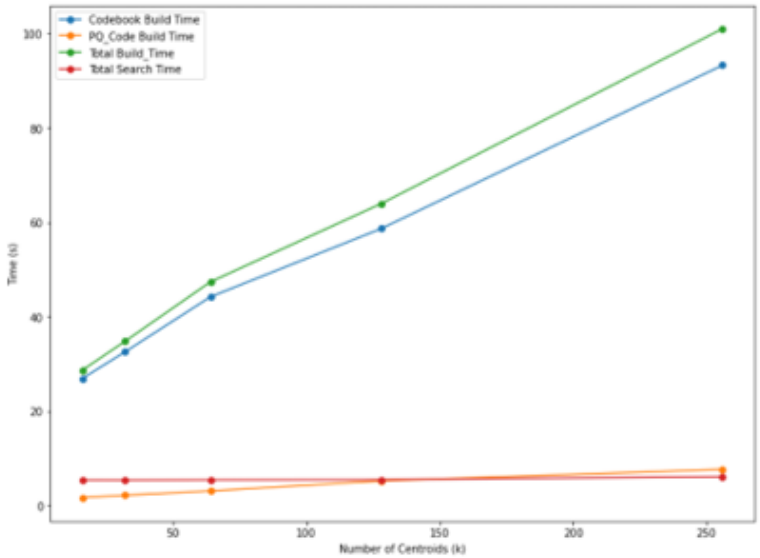
- Divide each training vector into m chunks of equal size D/m
- Run a separate K-means algorithm for every chunk and get a k centroids for each chunk and store in in Codebook $C_{j,k}$
- Assign each block to its closest cluster for each chunk. This is a symbolic representation of the training matrix and stored as PQ_Code
- Divide the query vector into equal m segments and assign each segment to a centroid according to the smallest Euclidean distance
- Create a distance map for each chunk
- Compare the query vector with all the training vectors using the distance maps and store their distances
- Once you have the summed distance, choose the top-k indices that have the lowest summed distance



4.4 Evaluation and Analysis of PQ

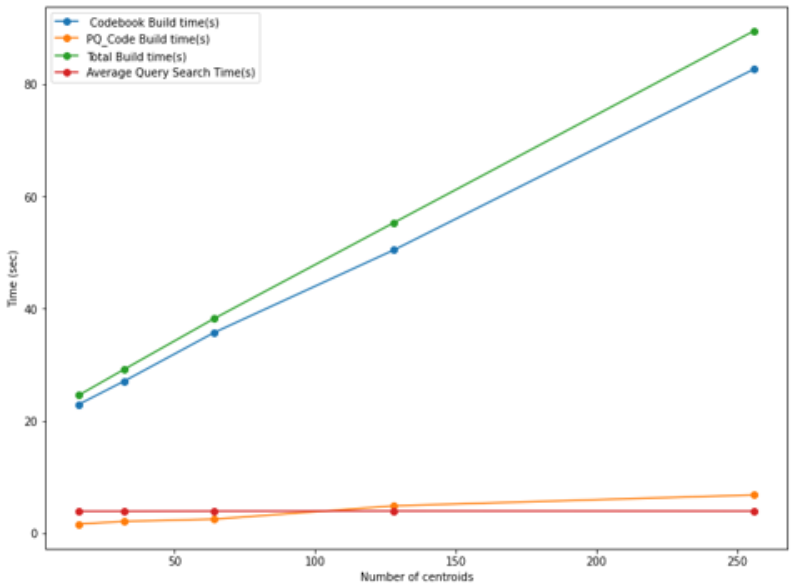
Exact Match Query

Value of k	$k = 16$	$k = 32$	$k = 64$	$k = 128$	$k = 256$
Codebook Build Time (s)	26.99	32.62	44.26	58.70	93.31
PQ_Code Build Time (s)	1.76	2.24	3.17	5.29	7.73
Total Build Time (s)	28.75	34.86	47.43	63.99	101.04
Average Query Search Time (s)	5.48	5.46	5.53	5.56	6.20



Perturbed Query(Gaussian Noise, sigma=0.1), M=4

Value of k	$k = 16$	$k = 32$	$k = 64$	$k = 128$	$k = 256$
Codebook Build Time (s)	22.960	27.059	35.706	50.436	82.629
PQ_Code Build Time (s)	1.633	2.097	2.492	4.862	6.805
Total Build Time (s)	24.593	29.156	38.198	55.298	89.434
Average Query Search Time (s)	3.906	3.906	3.932	3.955	3.936
Accuracy	0.1	0.2	0.6	1	1



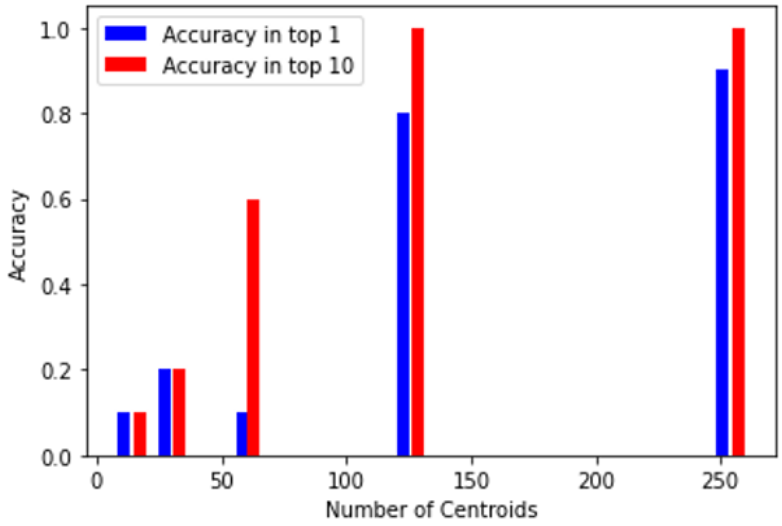


Figure 1: Accuracy for perturbed query increases with increase in k

5 Locality Sensitive Hashing

5.1 What is LSH?

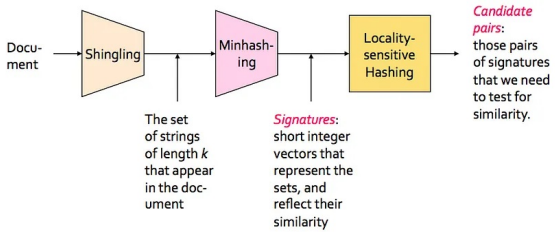
Locality Sensitive Hashing (LSH) is a technique for solving the nearest neighbor search problem in high-dimensional data. The key idea is to hash the points using several hash functions to ensure that for each function the probability of collision is much higher for objects that are close to each other than for those that are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point.

5.2 How does LSH work?

There are some basic principles on which LSH works:

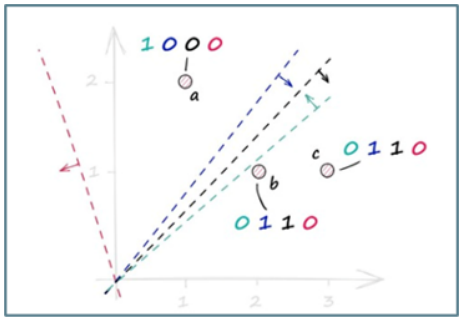
1. Hashing Techniques for High-Dimensional Vector Search: Hashing techniques for high-dimensional vector search aim to convert high-dimensional vectors into low-dimensional binary codes, such that vectors that are close to each other in the original space have a higher probability of being mapped to the same code. These techniques enable fast and efficient search in large databases of high-dimensional vectors.

2. LSH Algorithm: Principle and Methodology: The LSH algorithm uses a family of hash functions to map high-dimensional vectors to binary codes. The hash functions are chosen in such a way that the probability of collision between the codes is higher for vectors that are close to each other in the original space. The LSH algorithm then uses these codes to search for the nearest neighbors of a given query vector in a database of high-dimensional vectors.



Hyperplanes: A hyperplane is a flat subspace of a high-dimensional space that divides the space into two parts. In two dimensions, a hyperplane is simply a line. In three dimensions, it is a plane. In higher dimensions, it is a higher-dimensional flat subspace. Hyperplanes can be defined by a normal vector, which is a vector perpendicular to the hyperplane.

Shingling - Hashing a vector: It works by randomly selecting a set of hyperplanes to partition the high-dimensional space into buckets. The mapping of a vector to a bucket is done by computing the dot product between the vector and each hyperplane. If the dot product is positive, the vector is assigned to one bucket; if it is negative, the vector is assigned to the other bucket.



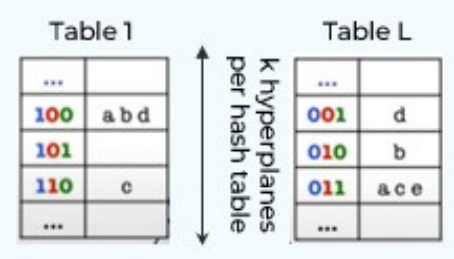
Parameters that affect the quality of LSH:
k_vals: numbers of hashes to concatenate in each hash function to try in grid search, basically number of hyperplanes
L_vals: numbers of hash functions/tables to try in grid search

One application of LSH is for text data represented as "shingles," which are contiguous subsequences of words. The LSH algorithm can be used to find similar documents by projecting the shingle space into a lower-dimensional space, and using a hash function to map each shingle to a bucket. Documents that have many shingles that map to the same bucket are likely to be similar.

Minhashing: Another key component of LSH is min-hashing, which is used to estimate the similarity of sets. Given two sets A and B, the min-hash value of a set is defined as the minimum hash value of any element in the set. By comparing the min-hash values of two sets, we can estimate their Jaccard similarity, which is the size of the intersection of the sets divided by the size of their union.

In LSH, we use min-hashing to estimate the similarity of points in the projected space. Each hash function used by LSH is defined as a linear combination of the dimensions in the projected space, followed by taking the minimum value. By comparing the hash values of two points, we can estimate their similarity in the original high-dimensional space.

Query Search: Create L hash tables, with k hyperplanes each. Bucket vectors with same code in one row. Query vector is hashed similar to the dataset vectors. Similarity is judged on the Jaccard score Each vector in the final bucket is individually matched through the chosen distance metric.



3. LSH Parameters and Configuration: The performance of the LSH algorithm depends on several parameters, such as the number of hash functions, the number of bits per code, and the threshold value for collision. The optimal values of these parameters depend on the specific dataset and the desired search performance. The LSH algorithm can be configured by tuning these parameters to achieve the desired search performance.

Applications:

1. Image and video search
2. Nearest neighbor search in large-scale databases
3. Recommendation systems
4. Collaborative filtering
5. Natural language processing

Advantages:

1. It is computationally efficient, especially when working with high-dimensional data.
2. It is scalable, making it suitable for large-scale databases.
3. It can be used with multiple hash tables to improve accuracy.

Disadvantages:

1. The quality of the LSH depends on the choice of hyperplanes and the number of hash tables used.
2. It can suffer from false positives, where vectors that are not similar in high-dimensional space are assigned to the same bucket.
3. The accuracy of LSH decreases as the dimensionality of the data increases.

5.3 Impact of LSH Parameters on Search Performance

The performance of the LSH algorithm is sensitive to the choice of parameters, such as the number of hash functions and the number of bits per code. Increasing the number of hash functions or the number of bits per code can improve search accuracy, but also increases the computational cost and memory usage. The optimal values of these parameters depend on the specific dataset and the desired search performance.

For exact match query:

Parameter	Number of Points Searched	Build Time	Search Time	Accuracy
k ↑	Decreases	Increases	Decreases	Decreases
L ↑	Increases	Increases linearly	Increases	Increases

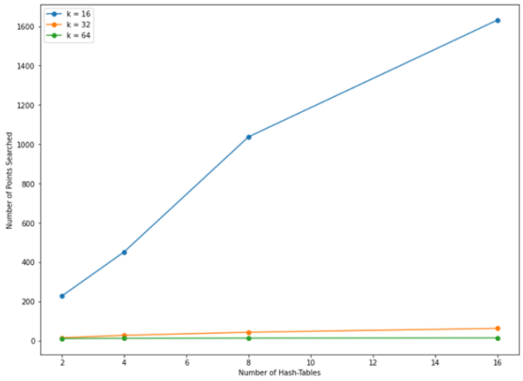


Figure 2: No. of Points Searched vs Number of Hash Tables

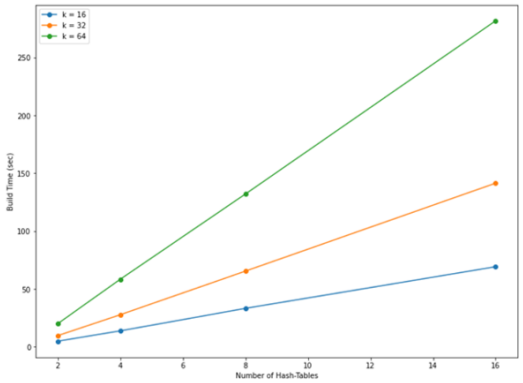


Figure 3: Build time vs Number of Hash Tables

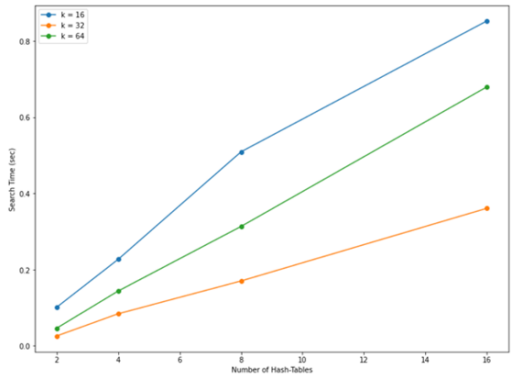


Figure 4: Search time vs Number of Hash Tables

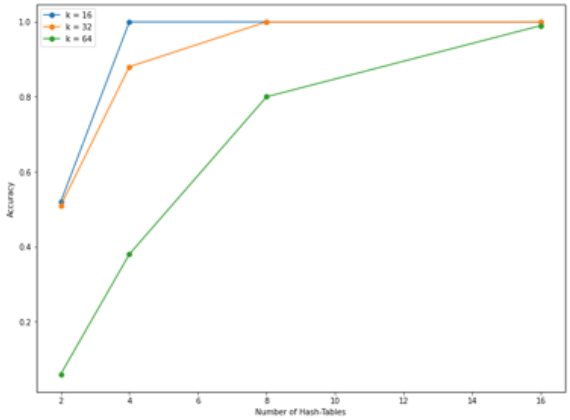


Figure 5: Accuracy vs Number of Hash Tables

For perturbed query(Gaussian Noise, sigma = 0.1)

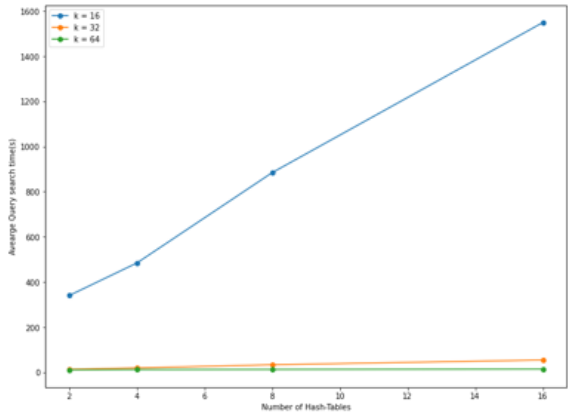


Figure 6: No. of Points Searched vs Number of Hash Tables

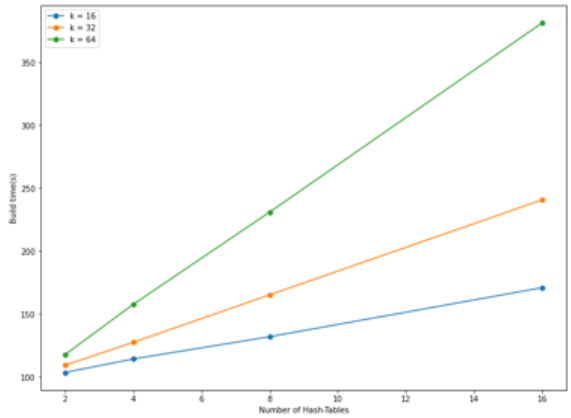


Figure 7: Build time vs Number of Hash Tables

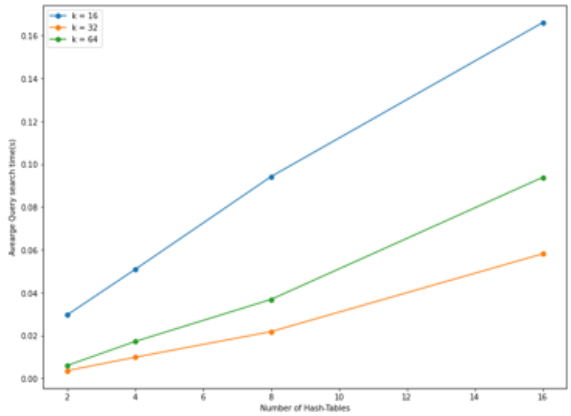


Figure 8: Search time vs Number of Hash Tables

Inferences made:

- The time graphs and number of points searched vs number of hash tables for perturbed query, follow a same relation as that of exact match query
- Accuracy for all cases was 1. Metric used: cosine (a.b <0.05)

6 Hierarchical Navigable Small Worlds

6.1 Idea behind HNSW

It slots into the graph category of ANN algorithm. The basic idea behind HNSW is to create a graph-based structure for the data points, where each node in the graph represents a data point and the edges represent the relationships between the points. The structure is organized into multiple levels, with each level having a smaller number of nodes than the previous level. When a query is made, HNSW starts at the highest level of the graph and uses ANN to find the best match, it then proceeds to send the match to the next level until the last layer is reached.

6.2 Terms and Terminologies

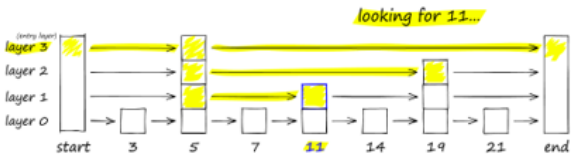
1. **Degree:** Number of links of a Vertex
2. **Level Multiplier m_L:** Normalization constant for probability function of vector insertion
3. **ef_Search:** The number of nearest neighbours to search while proceeding to the next layer
4. **ef_Construction:** The number of nearest neighbours to return while inserting the new element
5. **M:** The number of links to be returned at each layer while insertion
6. **M_max:** Maximum number of neighbours a vertex can have at a layer, usually = M
7. **M_maxO:** Maximum number of neighbours a vertex can have at layer 0, usually = 2*M
8. **Zoom-Out / Zoom-In:** Search Phase where we pass through low-degree/high-degree vertices

6.3 How does HNSW work?

There are two fundamental techniques that have contributed heavily to HNSW:

1. **Probability Skip Lists:** Probability skip lists are a probabilistic variation of skip lists that were introduced to reduce the search time complexity in hierarchical navigable small world (HNSW) graphs.

   In probability skip lists, each node is assigned a probability value p, which is a number between 0 and 1. When we traverse the skip list, we use the probability value of each node to determine whether or not to make a jump to the next node. Specifically, at each node, we generate a random number r between 0 and 1. If $r \leq p$, we make a jump to the next node at the next level. Otherwise, we stay at the current node and continue to the next node at the current level.



   In HNSW graphs, we use probability skip lists to store the neighbors of each point in the graph. Each point is represented as a node in the skip list, and the neighbors of each point are stored as nodes that are linked to the corresponding node in the skip list. The probability values of the nodes in the skip list are chosen in such a way that the resulting graph has a small world property, i.e., each node has a small number of short-range edges and a few long-range edges that connect it to other parts of the graph.

   During the search process, we start from a query point and traverse the skip list to find the nearest neighbors of the query point. The probability skip list allows us to quickly skip over parts of the graph that are far away from the query point and focus our search on the parts of the graph that are likely to contain the nearest neighbors. This makes the search process much faster than a brute-force search over all the points in the graph.

2. **Navigable Small World Graphs:** Navigable Small World (NSW) graphs are a type of graph structure that have been designed for efficient nearest neighbor search in high-dimensional spaces. NSW graphs combine the advantages of two other types of graphs: small-world graphs and navigable graphs.

   Small-world graphs are characterized by a high clustering coefficient (i.e., neighbors of a node tend to be connected to each other as well) and a low characteristic path length (i.e., the average number of steps it takes to go from one node to another). These properties make small-world graphs efficient for local search, but they may not be optimal for global search.

   Navigable graphs, on the other hand, are designed to optimize global search. They achieve this by embedding high-dimensional data into a lower-dimensional space, using techniques such as locality-sensitive hashing or principal component analysis. Once the data is embedded, a graph is constructed on top of the lower-dimensional representation, with the goal of preserving the pairwise distances between data points. This graph can then be searched efficiently using standard graph algorithms.

   They are constructed by first creating a small-world graph and then augmenting it with long-range edges that are added based on the pairwise distances between data points. The resulting graph has a small characteristic path length, making it efficient for local search, but it also has long-range edges that help to maintain the global structure of the data.

   The search algorithm used in NSW graphs is based on a variant of the breadth-first search algorithm, which is used to explore the graph in a way that minimizes the number of distance computations. The algorithm starts at the query point and traverses the graph in a series of hops, using a priority queue to keep track of the next node to visit. At each step, the algorithm selects the node with the smallest estimated distance to the query point, and then visits its neighbors to update their estimates. The algorithm terminates when the desired number of nearest neighbors has been found or when the search has reached a maximum depth.

6.4 How do the parameters affect HNSW?

Here are some of the results based on the implementation of HNSW. Parameters that affect performances: M, efConstruction, efSearch.

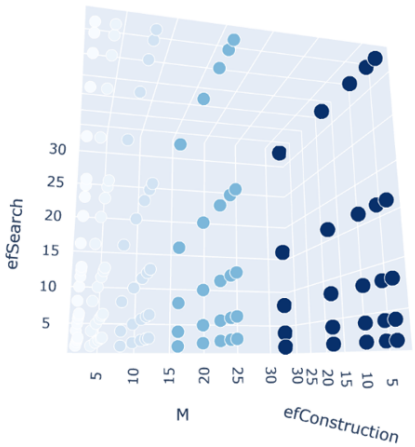


Figure 9: Memory vs parameters

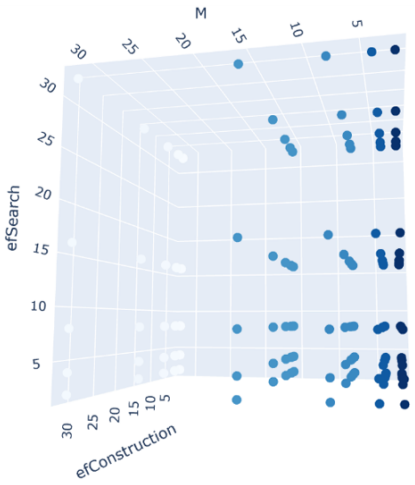


Figure 10: Build time vs parameters

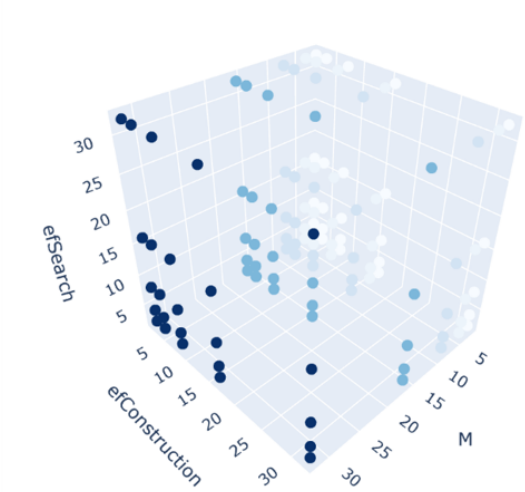


Figure 11: Search time vs parameters

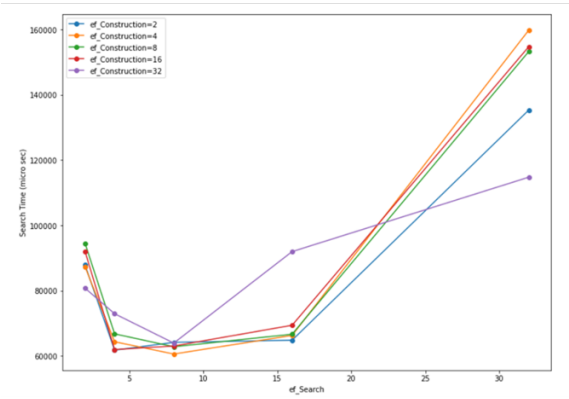


Figure 12: Search time vs parameters

Parameter	Recall	Build Time	Search Time	Memory
M ↑	Increases	Increases	Increases	Increases
efSearch ↑	Almost same	Almost same	Decreases initially then increases	Almost same
efConstruction ↑	Almost same: highest for efCon =16	Initially increases then decreases	Randomized but very close	Almost same

Figure 13: Inferences

6.5 What can we infer from the analysis?

1. Memory usage only depends on M and depends on it linearly.
2. Build_time depends on M and ef_Construction
3. Increasing M increases the recall as well as search_time, build_time and memory usage
4. No effect of ef_Search, ef_Construction on recall, only search time is affected
5. Recall increased from 913 to 941 for when number of nodes to search were increased from 1 to 10 (M=64, ef_Search = 2, ef_Construction = 32)

7 Scalable Nearest Neighbors

7.1 Introduction to ScaNN

Scalable Nearest Neighbors (ScaNN) is an algorithm for fast and efficient approximate nearest neighbor search on large-scale datasets. It is designed to handle high-dimensional vectors and can be used in a variety of applications, such as recommendation systems and image search.

7.2 ScaNN Algorithm: Principle and Methodology:

The ScaNN algorithm consists of three main steps: clustering, indexing, and querying.

1. Clustering: In the first step, ScaNN partitions the dataset into a set of clusters, each containing a small subset of the total data points. The number of clusters is determined by a user-defined parameter, and the clustering algorithm used can vary (e.g., K-means, Hierarchical clustering, etc.).
2. Indexing: Once the clusters are created, ScaNN constructs an indexing structure that allows for efficient nearest neighbor searches. Specifically, it builds a hierarchical tree of clusters using the clustering labels, with each level of the tree representing a different scale of the dataset. At each level, the clustering labels are used to split the dataset into non-overlapping sub-regions, with each sub-region corresponding to a cluster. ScaNN constructs an inverted index for each sub-region that maps each vector to a unique identifier.
3. Querying: When a query vector is received, ScaNN traverses the hierarchical tree and identifies a small set of candidate clusters that are likely to contain nearest neighbors of the query. The query vector is then compared to the inverted indices of the candidate clusters, and a small set of nearest neighbors is identified. The number of neighbors returned can be adjusted by the user-defined parameters.

ScaNN is an approximate algorithm, which means that it can sacrifice some accuracy to achieve higher search speed. Specifically, ScaNN aims to return a set of points that are near-neighbors to the query, but not necessarily the exact nearest neighbors. The degree of approximation can be adjusted by changing the clustering parameters, indexing structure, or the number of neighbors returned.

Overall, ScaNN is designed to handle high-dimensional vectors and large-scale datasets, and has been shown to be significantly faster than other state-of-the-art algorithms while maintaining comparable accuracy.

7.3 How well did ScaNN perform?

Exact Query:

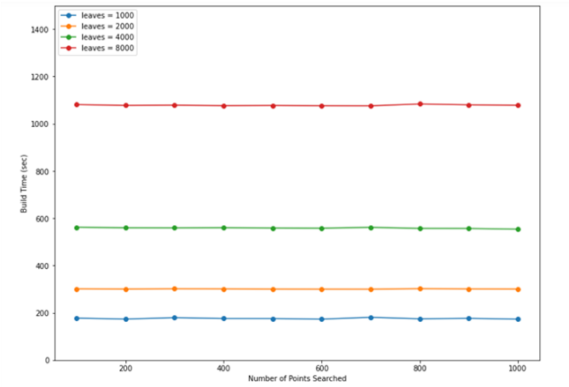


Figure 14: Build time vs Number of points searched

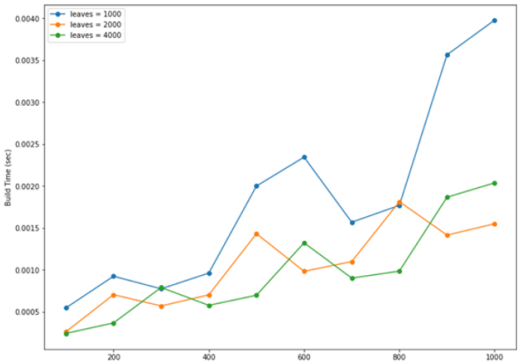


Figure 15: Search time vs Number of points searched

Inferences:

- The search_time is almost constant with change in number of neighbours to return or number of leaves to search.
- It is linearly increasing with increase in number of leaves of each node.
- ScaNN was the fastest and most accurate among the tried algorithms.

Perturbed Query: Gaussian Noise: sigma=[0.1, 0.5, 2]

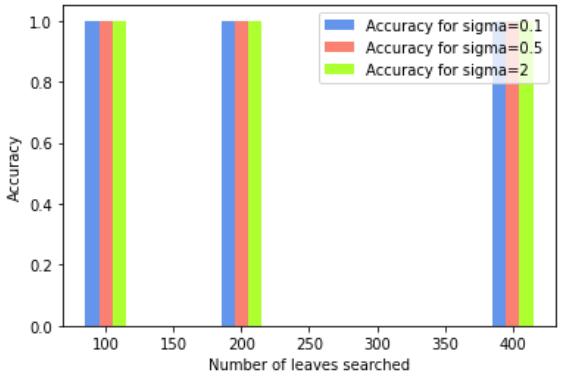


Figure 16: Accuracy

8 Optimized Product Quantization

8.1 What is the need for Optimization?

The need for optimization in Product Quantization (PQ) arises from the fact that the standard PQ algorithm has limitations in terms of its accuracy and search efficiency. Some of the key limitations include:

1. Suboptimal codebooks: The standard PQ algorithm uses K-means clustering to generate codebooks for each subvector. However, K-means can result in suboptimal codebooks that may not capture the true distribution of the data. This can lead to lower accuracy and higher search times.

2. Codebook size: PQ requires a large number of codewords in the codebook to accurately represent the subvectors. However, a large codebook size can lead to higher memory usage and slower search times.

3. Query time complexity: The standard PQ algorithm has a query time complexity that is linear in the number of subvectors, which can be computationally expensive for large datasets.

4. Search accuracy: The primary goal of vector search algorithms is to find the most similar vectors to a given query vector. However, if the quantization boundaries and codebooks used for subvector quantization are suboptimal, the search accuracy may be reduced. Optimization techniques such as OPQ can help to improve search accuracy by finding better quantization boundaries and codebooks.

5. Generalization: Vector search algorithms may perform well on certain types of datasets but may not generalize well to other datasets. Optimization techniques can help to improve the generalization by finding better solutions that work well across different datasets and vector types.

To address these limitations, various optimization techniques have been proposed for PQ, such as Optimized Product Quantization (OPQ). OPQ improves upon the standard PQ algorithm by optimizing the codebook generation process and reducing the codebook size. It also introduces a rotation matrix to align the subvectors, which can improve the accuracy of the algorithm. By optimizing PQ, it is possible to achieve better search efficiency and accuracy, making it a more effective vector search algorithm.

8.2 How do we Optimize PQ?

Optimized Product Quantization (OPQ) overcomes the challenges of traditional Product Quantization (PQ) by optimizing the codebooks used in the subvector quantization process. Here are some ways in which OPQ achieves this:

1. Rotation matrix: In OPQ, an additional rotation matrix is introduced that rotates the input vectors to a new coordinate system. Let X be the original input matrix and R be the rotation matrix. Then, the rotated input matrix X' can be obtained as X' = XR. The rotation matrix R is optimized to minimize the distortion between the input vectors and their corresponding quantized representations. This is achieved by solving a matrix optimization problem that minimizes the sum of squared errors between X' and its quantized representation.

2. Learning: The rotation matrix in OPQ is learned using a gradient descent algorithm that minimizes the distortion between X' and its quantized representation. Let B be the codebook matrix and Q be the quantization function that maps X' to its quantized representation. Then, the distortion between X' and its quantized representation can be measured as the sum of squared errors between X' and BQ(X'). The gradient descent algorithm updates the rotation matrix R iteratively by computing the gradient of the distortion with respect to R and adjusting R in the direction of the negative gradient.

3. Fine-tuning: OPQ allows for fine-tuning of the codebooks by adjusting the quantization boundaries based on the distribution of the data. Let P be the probability distribution of the input vectors and B be the codebook matrix. Then, the optimal quantization boundaries

can be obtained by solving a maximum likelihood estimation problem that maximizes the likelihood of the data given the codebook. This can be achieved using an iterative algorithm that adjusts the quantization boundaries based on the current estimates of P and B.

4. Compression: OPQ can be combined with compression techniques such as product hashing or vector quantization to reduce the memory requirements of the codebooks. Product hashing involves hashing the subvectors of the input vectors to a small number of hash tables, and then using the hash tables to lookup the corresponding codebook indices during quantization. Vector quantization involves clustering the subvectors of the input vectors into a small number of centroids, and then using the centroids as the codebook. Both techniques can significantly reduce the memory requirements of the codebooks without sacrificing search accuracy.

   Overall, OPQ overcomes the challenges of traditional PQ by optimizing the codebooks used in the subvector quantization process. This can lead to significant improvements in search accuracy, query speed, storage efficiency, and generalization.

9 Comparison & Scalability of different Algorithms

- **Total build time:** LSH(30-250s) > ScaNN(200-1000s) > HNSW(70-400s) > PQ(30-100s)
- **Average Query Search time:** ScaNN(0.5-3.5ms) < HNSW(0.08s-0.16s) < LSH(0.1-0.8s) < PQ (4-5s)
- **Accuracy for exact matches:** ScaNN>LSH>HNSW>PQ
- **Accuracy for perturbed query:** ScaNN LSH > HNSW PQ

10 Results

- ScaNN proves to be the best algorithm for our use.
- We can also see that there is a time vs accuracy trade-off for the algorithms.
- Although Build time for ScaNN is much higher compared to other algorithms but it's average query search time is much lower.
- Accuracy is also best for both exact matches and perturbed query.

11 Future Prospects

In this project, I have worked mainly on the sift1m dataset which is a publicly available image dataset. We can also implement these algorithms on real audio datasets and test their efficiencies. Further, we can take combinations of many vectors as the audio sound could be bigger. There are many scopes of experiments and I would like to keep on exploring these ideas.

References

[1] https://github.com/gamboviol/lsh/blob/master/lsh.py
[2] https://www.analyticsvidhya.com/blog/2022/08/product-quantization-nearest-neighbor-search/
[3] https://www.pinecone.io/learn/product-quantization/s
[4] https://medium.com/dotstar/understanding-faiss-part-2-79d90b1e5388
[5] https://www.pinecone.io/learn/hnsw/
[6] https://faiss.ai/index.html
[7] https://youtube.com/playlistlist=PLBv09BD7ez_6xoNh_luPdBmDCIHOQ3j7F
[8] https://cloud.google.com/blog/topics/developers-practitioners/find-anything-blazingly-fast-googles-vector-search-technology
[9] hhttps://github.com/google-research/google-research/tree/master/scann/