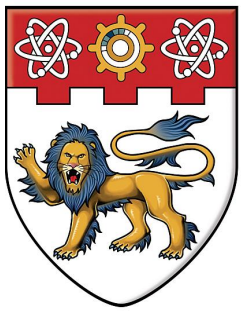

NANYANG TECHNOLOGICAL UNIVERSITY

SC4020 Data Mining Project

Group 07

Lewis Chng Qihon
Bhat Sachin

Abstract

This paper presents an exploration and comparison of various Approximate Nearest Neighbor Search (ANNS) algorithms, namely Locality-Sensitive Hashing with Random Projections (LSH-RP), K-Decision trees (KD-Trees), and Anisotropic Vector Quantization (AVQ). We systematically examined the impact of key parameters within each algorithm, such as hash value length for LSH, tree-related parameters for KD-Trees, and indexing structure parameters for AVQ. Our findings highlight the various effects of parameter optimization in influencing the time taken for similarity search processes, with implications for the performance of these algorithms in handling datasets of varying sizes and dimensionalities. We highlight the need for further investigations, emphasising the potential use of advanced techniques like k-fold cross-validation to fine tune parameters and achieve a balance between recall and resource utilisation, enhancing the practical applicability of these ANNS algorithms.

Introduction

Similarity search is the process of finding the most similar items given an input vector of features, commonly known as the Nearest Neighbour Search (NNS) or Approximate NNS (ANNS) problems. Naturally, many classes of algorithms have been implemented to solve this problem, each using unique methods. In this paper, we aim to analyse three types of algorithms, namely: Locality-Sensitive Hashing (LSH), Vector Quantization (VQ) and Tree-based approaches (e.g. K-Decision trees (KD-Trees). Throughout this paper, we aim to evaluate the performance and results of these algorithms, and understand how changes in their parameters influence the search process.

Methods

In this section, we introduced each algorithm we have chosen, and explained the methods by which these algorithms perform their search. Furthermore, we provided an outline of our methods to compare and contrast the various algorithms and their corresponding parameters on our chosen dataset.

The family of LSH algorithms function by hashing entries in such a way that similar data points are more likely to get mapped to the same hash value, allowing for more efficient similarity search by both reducing the dimensionality of input features and mapping similar points to the same hash. In particular, the algorithm we use in this exploration is LSH-Random Projection (LSH-RP), where k amounts of random and normally distributed vectors are used to construct an equal number of hash functions projecting each input into a binary hash, and are further combined into a k-bit digest of the original vector (random k-bit binary projection). Contrary to conventional cryptographic hash functions, LSH digests aim to maximise the potential collisions between adjacent or similar data points. It is these collisions that form *hash buckets* that serve to group colliding points, which should be similar to each other, together for searching. Given a query, the LSH algorithm generates a digest of the query vector using the same hash functions, and performs a linear search within the query's corresponding hash bucket for the top-k nearest neighbours using any specified similarity metric.

VQ algorithms process inputs by quantizing each dimension to centroids or *codebooks* that serve to cluster similar values, improving search efficiency by reducing the complexity required for *approximate* distance calculations, instead of exact distances. An alternate implementation of this is Anisotropic VQ (AVQ), in which Guo et al. [1] introduces a variation of quantizing vectors that attempts to minimise the difference in inner product between the original and quantized vectors, leading to errors parallel to the vector being more heavily penalised than errors in other direction, notably for errors perpendicular to the original vector. By approximating distances between clusters of similar points, implementations of VQ algorithms can be made hardware-efficient for parallelized processing, or be further compressed with techniques such as Product Quantization (PQ), all of which further improve scalability with increasing dimensionality and size of the dataset.

Tree-based algorithms, such as those using KD-Trees, recursively partition the vector space along a specified axis according to specified values to decide which subtrees data points will be categorised under. This process is repeated recursively for each feature until a criterion is met, which may be a set depth or amount of leaf nodes. The search process involves comparing the root node's value with the query's corresponding dimension value, and traversing the tree structure comparing the dimension components until a leaf node is reached [2]. Top-k results can be implemented by introducing a priority queue that stores potential nearest neighbours during the initial traversal, and sorted based on shortest distance between the current query point. and the traversal repeats until k leaf nodes corresponding to the top-k nearest neighbours have been reached.

To compare the performances for the algorithms above, we used the SIFT1M synthetic dataset [3] which consists of pre-trained feature embedding vectors with 128 dimensions, designed to evaluate the quality of ANNS algorithms using Euclidean (L2) distance as a similarity metric. The dataset consists of 1 million base vectors and 10,000 query vectors, in which the query set is used to find similar items in the base dataset. For the purposes of our exploration, we used a subset of 100 query vectors to evaluate and visualise the algorithms' search performance.

Additionally, for the three algorithms selected, we compared the top 10 nearest neighbours returned from the base dataset given a database of query vectors, as well as the time taken and memory required between all algorithms. Both time and space measures will be in terms of the time (in milliseconds) taken for the algorithms to search the subset of query vectors, and the final file size for each algorithm's runtime respectively. Lastly, for each algorithm, we investigated how variations in certain parameters affect results given the same subset of query vectors, and visualised the time taken to search the query subset as a given parameter changes.

Experiment & Discussion of results

We identified Python's NearPy library [4] as a suitable implementation for LSH, as it streamlines the classic LSH-RP algorithm as a pipeline or *engine* with minimal manual input. By

setting the baseline hash value size to be 10 bits, we instantiated the search engine to construct the hash table using the base dataset, and searched for nearest neighbours given the set of query vectors. The time taken to search the database given 100 queries was 7.548 seconds, and storing all data required 9.68 kilobytes of memory. While LSH search time is $O(n)$ for each query, its space complexity is $O(lnk)$, for l hash tables with n points and with k-dimensional vectors. Therefore, the space needed to store the hash table scales with the size and dimensionality of the dataset, which does not scale well for large and highly dimensional datasets.

Next, we investigated the Fast Library for Approximate Nearest Neighbours (FLANN) [5], which incorporates several ANNS algorithms such as KD-trees and hierarchical K-means trees and has grown to be a popular library for NNS problems and machine learning frameworks. The implementation we used was the Pyflann library, which serves as a Python wrapper around FLANN's C++ code, and we selected the standard KD-Tree approach for our dataset. Memory required for data storage was 5.16 kilobytes, and the time taken for search was 0.075 seconds. Compared to LSH-RP, the search time was significantly faster for 100 queries, in addition to requiring less memory to store data. In terms of analytical time complexity, searching for nearest neighbours using KD tree algorithms perform in $O(log(n))$ time on average, given the tree's branching structure. However, KD-Tree performance has been found to degrade rapidly if applied on high-dimensionality datasets, due to the amount of dimensions the tree builder will have to create subtrees on. There are extensions based on KD trees that introduce randomness (e.g. random projection trees) that aim to reduce dimensionality for ANNS problems, in order to minimise computational complexity. The space complexity is $O(kn)$ for n amounts of k -dimensional vectors, since the data is stored usually as indices in trees for efficient lookup.

Moving on to AVQ, we used the Scalable Nearest Neighbours (ScaNN) library [6] by Google Research, which implements their findings in [1]. ScaNN implements a tree-based index structure using the anisotropic loss function defined in their paper to quantize vectors. The search on the query set was completed in 1.427 seconds, while data storage took 16.6 kilobytes, which , but requires more memory to store data than the former two. According to page 5 in [1], AVQ achieves a search time complexity of $O(kd + n)$, by constructing a lookup table using the d-dimensional query vector's inner product with k amounts of codewords in $O(kd)$ time, and performing lookup table search for n query points. While it is slower than FLANN, the experimental recall is consistently high given

Based on the complexity analysis and our experimental findings, both LSH-RP and KD-Trees seem to be better suited for low-dimensional data with less features, while KD-Trees is more efficient in terms of memory required than LSH-RP and AVQ when performing searches large datasets with many features. Looking at time complexity, LSH-RP took significantly more time than KD-Trees and AVQ, KD-Trees had the fastest search time in the range of milliseconds compared to AVQ with 1.4 seconds. However, the time complexity for tree implementations do not scale well with increasing data dimensionality, due to the algorithm having to traverse an exponentially increasing amount of subtrees as mentioned previously, to which [7] coins the

phrase “curse of dimensionality”. It is this issue of scalability that, for high dimensional datasets with many features, exact distances can be expensive to compute. Therefore, ANNS methods are preferred to simply compute approximate near neighbours in these types of datasets in an efficient manner. If we consider increasing the amount of vector features for an equally large dataset, the KD-Tree method would suffer the most performance loss compared to LSH and AVQ, with the latter only facing potential memory storage issues for larger datasets. Even so, the data compression aspect found in most vector quantization techniques will benefit searches on extremely large datasets, allowing more data to be stored in running memory.

Now that we have compared the results between the different algorithms, we proceeded with tuning parameters within each algorithm to identify and illustrate any potential variation in the similarity search process. The main parameter of interest for LSH algorithms would be the size or length of the hash value. For a fixed size dataset, longer hash values would necessitate a decreased probability of collisions for less similar data points, resulting in hash buckets containing lesser similar points. We experimented with varying the hash lengths from 1 bits to 20 bits, and recorded the time taken to search all queries in the subset. A plot of the time taken against hash length can be seen in Figure 1 below.

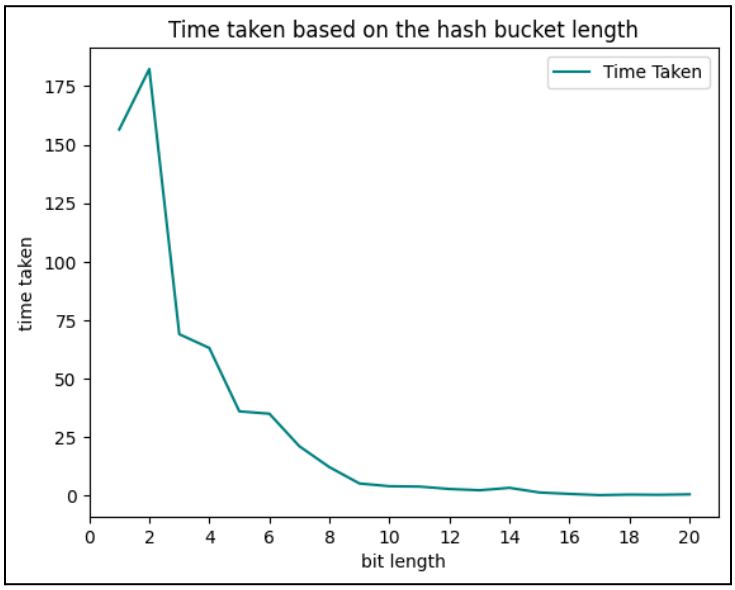

Figure 1: Plot of search time against length of hash bucket

We observed that the search time for increasing hash bit lengths were shorter, and this may be explained due to the decreasing amount of data points colliding in the same hash bucket, resulting in less candidate points to calculate distances with. At 20-bit hash length, on average there will be 2^20 possible combinations, which is just over 1 million unique hash buckets (1,048,576 to be precise). At this point, it is increasingly unlikely to have colliding hashes for our dataset, since it has only 1 million base vectors. Therefore, for top-k ANNS problems, there should be a minimal hash bucket length to which the average value in each bucket is at least $$k$$; For our dataset, the value would be a 16-bit hash length (comprising 65,536 buckets) with 17 candidate nearest neighbours per hash bucket on average.

For FLANN's KD-Trees approach, there are many more parameters to vary in the tree's construction and searching method. In our implementation, we used the standard KD-Tree implementations with the amount of trees, maximum leaf node count across the entire tree, and the branching factor of the tree as our parameters of interest. We only varied one variable at a time while maintaining the rest at fixed values, as we considered investigating all combinations to be redundant for the purposes of our exploration. Our results for the three parameters can be seen in Figures 2, 3, and 4.

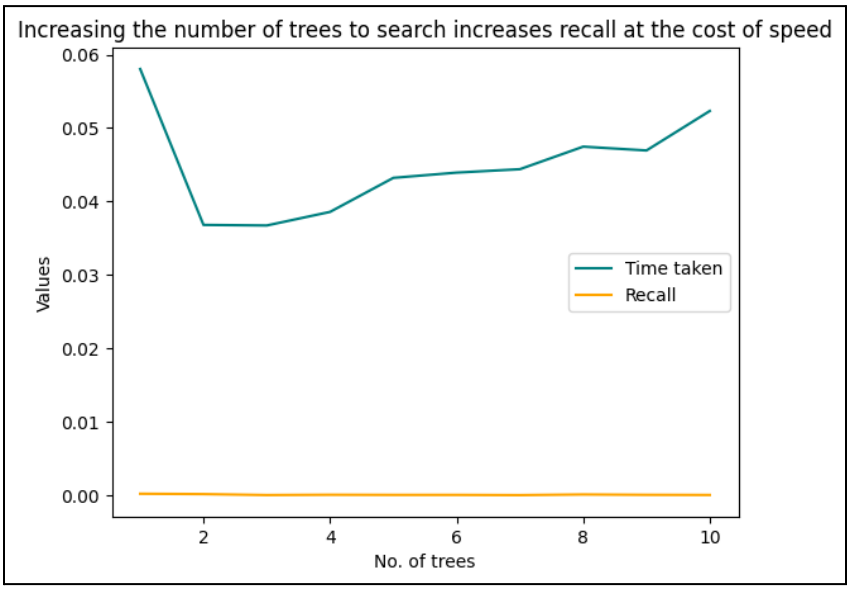

Figure 2: Plot of search time against the number of trees constructed.

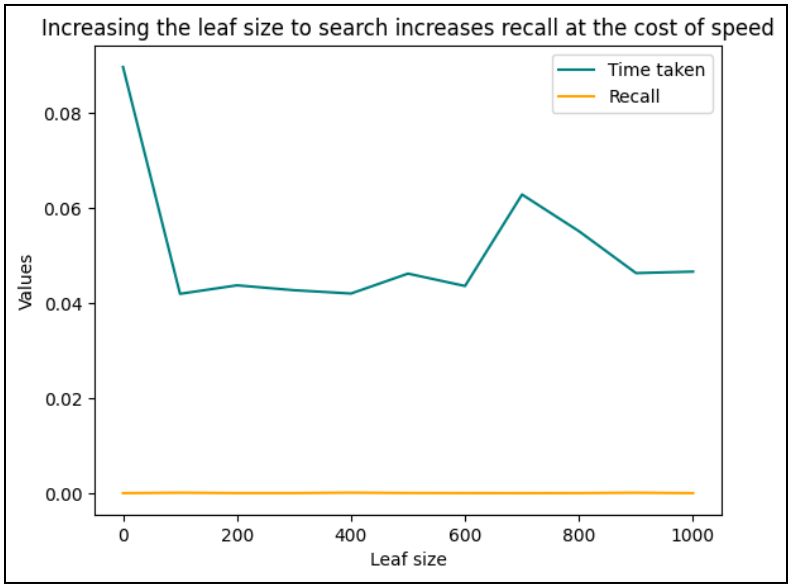

Figure 3: Plot of time taken for search against increasing amount of leaves per tree.

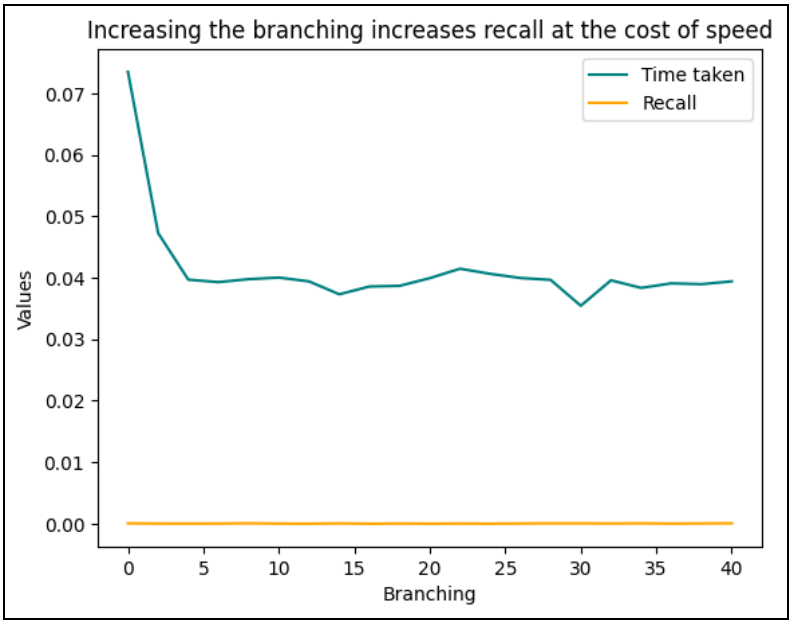

Figure 4: Plot of search time against branching factor of the trees.

From the figures above, it is apparent that increasing the parameter values contribute to decreased search times. The optimal number of trees as seen in Figure 2 seems to be around 4 (which was already set as the default value), though increasing the amount of trees contributes to more variables to search. Likewise, Figure 3 demonstrates a similar trend of decrease with a spike at 700 leaves per tree, which could potentially be explained by methodological error from an inconsistency in measuring system time. Meanwhile, the search time taken with increasing branching factor seems to plateau at half the search time using a branching factor of 1, meaning 1 branch per node which is equivalent to a linear search. The lowest search time appears to be around 30-32, though the value plateaus at around half of linear search. For optimal use, a variant of k-fold cross validation for hyperparameter tuning may be used to find a balance between recall and parameter used, and a better method of calculating recall given the returned neighbours should be implemented.

Last but not least, ScaNN's indexing structure uses a tree approach, therefore the tunable parameters involve the index tree's construction as well. The parameters we have identified are the number of leaves to search, and the maximum number of neighbours to search prior to reordering, which is . Our findings for both variables can be seen in Figure 5, 6, and 7 below.

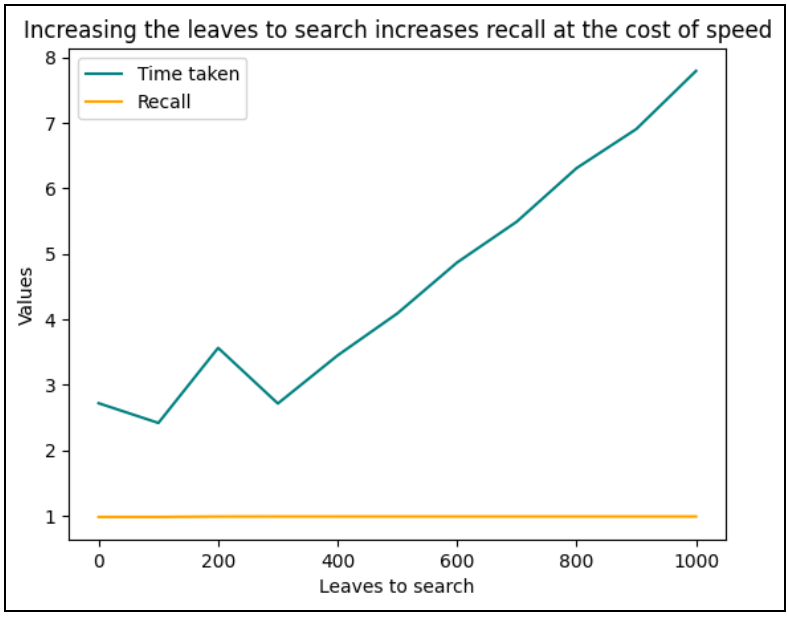

Figure 5: Plot of search time against amount of leaves searched

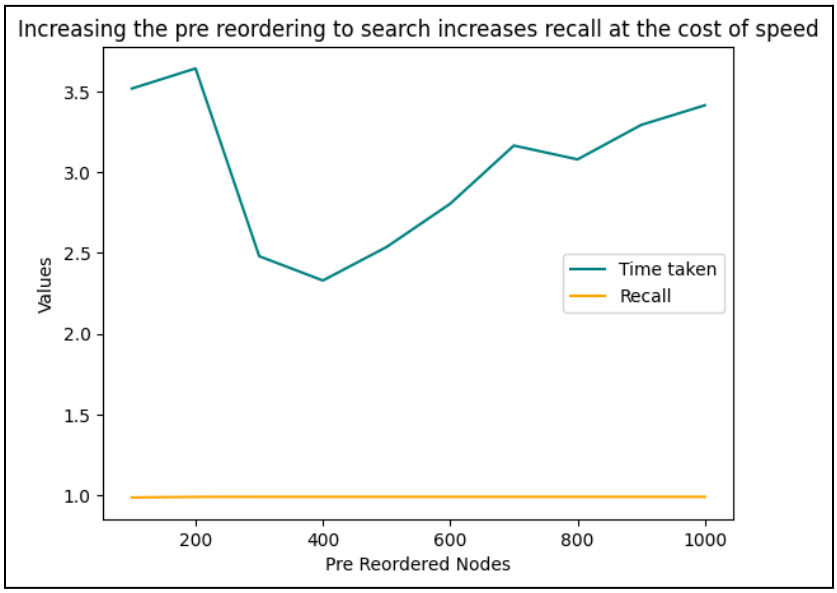

Figure 6: Plot of search time against amount of nodes in the tree prior to tree reordering

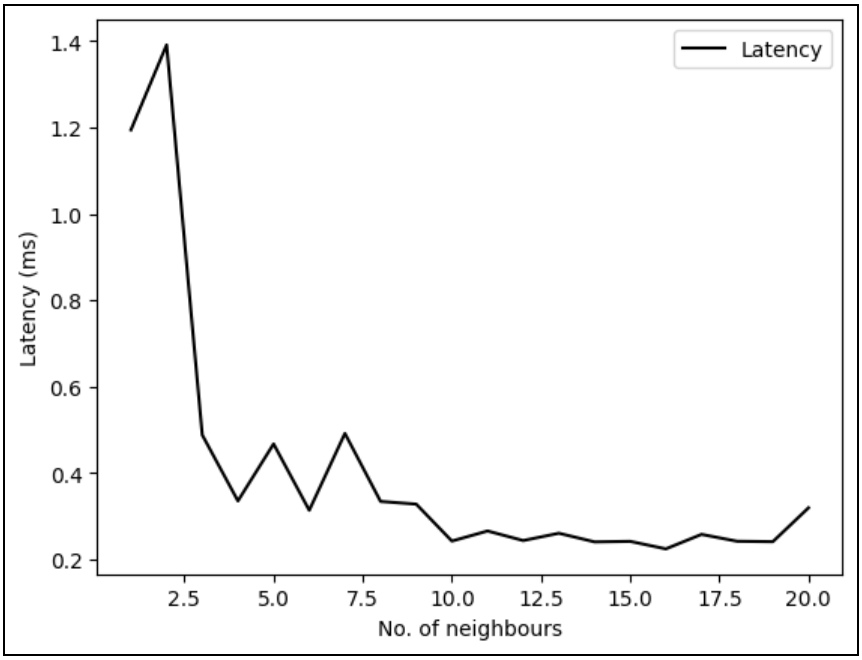

Figure 7: Plot of search time against number of nearest neighbours returned

By observing the results above, Figure 5 demonstrates that increasing the total amount of leaves to search in the tree increases the search time linearly, which supports the algorithm's linear search method to find nearest neighbours in a lookup table. The tree reordering refers to sorting the data points to minimise the number of distance computations required in order to reduce search times. From Figure 6, there appears to be a local minimum at 400 nodes pre-reordering, performing better at 2.32 seconds compared to the more extreme values. Search performance appears to drop off as the number of nodes increases, likely due to the increased amount of nodes to process in the reordering step. Lastly, increasing the amount of neighbours returned shows a significant decrease in search time, as seen in Figure 7. The high latency for lesser amounts of neighbours can be explained due to the algorithm having to search for the highest and/or second nearest neighbours, which necessitates more distance calculations to search for one specific point, whereas with high values the algorithm may more easily approximate the top-k nearest neighbours more efficiently, though further in-depth study of the algorithm may be required to support this view.

Overall, the parameters we explored can be tuned with further experimentation to optimise resource consumption against recall. Further exploration into this area may involve a wider array of variables, metrics, algorithms or data set types to test again, such as recall over queries per second. [8] investigates several other indexing methods including graph-based techniques and Deep Quantization Networks (DQN) as an extension of PQ on neural embeddings, similar to the dataset we utilised in our exploration. Other vector feature datasets ,such as the Global Vectors for Word Representation dataset (GloVE) [9], may be used to benchmark these algorithms' performances against different types of datasets to investigate performance variations. Some improvements to the experimental methods may also include

running multiple runs and obtaining average results, though one should be mindful of the available computational resources as using a larger dataset will significantly increase runtime.

Naturally, there exist some potential improvements to the methods we use to collect experimental data. Since the method we used to measure time taken for a block of code to run was using Python's time library, any form of background process may influence the CPU's performance on processing our intended code, therefore introducing unwanted variability into the modules we use to record time. Additionally, the memory taken according to file size includes all text outputs for all blocks of code within the Jupyter Notebook environment. For future explorations, some possible suggestions to address both issues could be using an external server with minimal background activity to enable Python's time module to accurately measure the time taken, as well as to implement some form of memory access or data serialisation (including but not limited to Python's pickling functions to serialise data structures), since the data structures built by the algorithms are stored in random access memory and are lost upon termination of the kernel.

Conclusion

To conclude our exploration, we have investigated three common families of ANNS algorithms, namely LSH,-RP, KD-Trees and AVQ, in performing similarity search on high-dimensional feature vector embeddings. Our investigation revealed that LSH-RP and KD-Trees are better suited for low-dimensional datasets with fewer features, with KD-Trees demonstrating more efficient memory usage compared to LSH-RP and AVQ for larger datasets. Consequently, for datasets with high dimensions and numerous features, approximate methods such as LSH-RP and AVQ are preferable, providing efficient and effective solutions for identifying approximate nearest neighbours. Moreover, we presented the various effects of each algorithms' parameters on the time taken to search a fixed-size subset of query vectors. Further explorations using advanced techniques, such as k-fold cross-validation for hyperparameter tuning, would be instrumental to achieve a balance between recall and parameter utilisation, thereby advancing the efficiency of these algorithms in practical applications.

References

[1] R. Guo et al., “Accelerating large-scale inference with anisotropic vector quantization,” arXiv.org, https://arxiv.org/abs/1908.10396

[2] H. Hristov, “Introduction to K-D trees,” Baeldung on Computer Science, https://www.baeldung.com/cs/k-d-trees

[3] “Datasets for approximate nearest neighbour search,” Evaluation of Approximate nearest neighbours: large datasets, http://corpus-texmex.irisa.fr

[4] NearPy - ANN in Python, http://pixelogik.github.io/NearPy

[5] M. Muja and D. G. Lowe, “Scalable nearest neighbour algorithms for high dimensional data”, https://ieeexplore.ieee.org/document/6809191

[6] ScaNN: Scalable Nearest Neighbors, https://github.com/google-research/google-research/tree/master/scann

[7] R. Bellman, “Dynamic programming”, https://www.science.org/doi/10.1126/science.153.3731.34

[8] Y. Wang, A survey on Efficient Processing of Similarity Queries over Neural Embeddings, https://arxiv.org/pdf/2204.07922.pdf

[9] J. Pennington, R. Socher, and C. D. Manning, Glove: Global Vectors for Word Representation, https://nlp.stanford.edu/projects/glove