STATS 402 - Interdisciplinary Data Analysis
Enhancing Dense Retrieval Efficiency with Hierarchical Clustering

Kaiyuan Lou
kaiyuan.lou@dukekunshan.edu.cn

Yizhou Bi
yizhoubi@dukekunshan.edu.cn

July 8, 2023

Abstract

1 Introduction

With the improvement of the information extraction ability of language models, dense vector retrieval is playing a more and more important role in the information retrieval field. Nowadays, the adoption of dense retrieval models has become indispensable for successful search engines. Notably, Google has incorporated BERT [1] as the foundation of its information retrieval model, while Facebook relies on the Faiss [2] retrieval model, and Microsoft has introduced MT-DSSM [3]. The dense retrieval model can extract high-level understandings of natural language, which make the search more accurate and easy to use. However, the effectiveness of dense vector retrieval models is impeded by the challenge of sluggishness resulting from the use of high-dimensional embeddings, thereby hindering the implementation of similarity search. Consequently, brute-force techniques are commonly used to traverse every embedding, which makes retrieval speed significantly lag behind that of traditional sparse vector retrieval models.

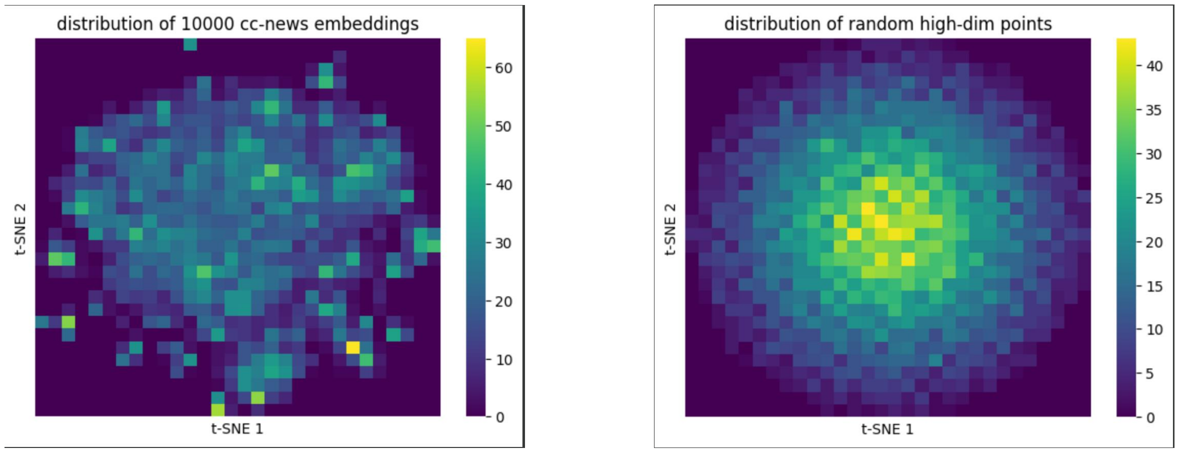
In traditional search algorithms like TF-IDF [4] and BM25 [5], the retrieval targets are the calculated values between searched token and documents, which means only documents containing the searched words are selected and compared. In contrast, the dense vector retrieval model operates differently. Here, each document is accompanied by a high-dimensional dense vector embedding, forcing comparisons to be performed across the entire corpus dataset. It will be super expansive when running a large dataset.

After analyzing the embeddings of documents, we find that documents with similar topics tend to have similar embedding vectors in the high-dimensional space, which makes clustering possible. From the distribution visualization by using t-SNE [6] algorithm over 1000 corpus from our test dataset, we find that the embeddings form little clusters automatically. The comparison of distribution between corpus embeddings and random data is in Figure 1.

In this paper, we are introducing a solution for quicker similarity search over large corpus embeddings by sacrificing a little accuracy. We achieved the speed boost by building a search tree over the entire dataset using a hierarchical clustering strategy.

Our test model is implemented by using Vladimir Karpukhin et al.'s Dense Passage Retrieval model [7], the test dataset is the cc-news dataset [8] from Huggingface, and the Large Language Model (LLM) for generating embeddings is the bert-base model from Huggingface.

In the following sections, we will first review the current algorithms for similarity search over high-dimensional data. Subsequently, we will explicate our algorithm in a meticulous manner while undertaking a thorough analysis of the associated hyper-parameters and their respective influences. Furthermore, we will benchmark our test implementation and establish a comparative evaluation against existing strategies. Ultimately, we will explore additional potentialities for our strategies that we haven't implemented yet.



(a) corpus distribution

(b) normal distribution

Figure 1: distribution of high dimensional data points

2 Literature review

2.1 Existing methods

The target of vector retrieval is, given a vector, an algorithm needs to find the nearest neighbor(s) of that vector. Many space partitioning methods, of which main goal is to build some structure in the vector space, like Grid File [9], KD Tree [10], K-D-B Tree [11], M Tree [12], Ball Tree [13], have been studied to improve the speed of vector retrieval speed. The difficulty this algorithm faces is that the vectors are naturally unstructured data and the curse of dimensionality– their performance drastically decline with the increase of the dimensionality, because of the dimensionality the algorithm introduced increases with the increase of dimension space. [14] The difficulties can be understood from two perspective. The first perspective is that the increase in dimensionality will make the possible partition of the space increase exponentially. Or, another perspective is that the tree structure created by partitioning the space will become very empty, and deep, greatly limiting the efficiency of the retrieval process. Another direction of efficient vector retrieval is to make local sensitivity hashing. To counter the disadvantage introduced by high dimensionality, some studies suggest the usage of hashing, like Local Sensitivity Hashing (LSH) [15]. One great advantage of LSH is that when indexing the vectors, the complexity of the index does not directly correspond to the dimensionality of the original vectors. However, to achieve accurate approximation of the nearest neighbour in high-dimension space, many more tables and longer hash is required, negatively influencing the performance. [16] However, it is still very promising and is widely adopted, like in Spotify's Approximate Nearest Neighbour Oh Yeah (ANNOY) library. Despite the ANNOY utilizing the tree structure or forest structure, it resembles the LSH by adopting the concept of predefined amount of hyperplane to slice the space, essentially creating a hash-like bucket. Besides tree and hashing, another potential solution is the graph. The primitive graph algorithm construct a directional linked node, using an edge that contains vector info. The vector info describes the direction and distance between two vector in the space, essentially linking two nodes (vectors) together. When conducting the search, the algorithm will greedily follow the path that minimize the distance between input vector and indexed vectors in the database until the distance no longer decrease. This primitive algorithm is not good for clustered data and may yield bad accuracy because of local minimum, rather than the global one, is reached. To counter this problem, the current state-of-art vector search algorithm is to use a hierarchical directional graph. Hierarchical Navigable Small Worlds (HNSW) [17] is applied by Facebook (Meta)'s vector search library, Faiss. The original idea of HNSW is from skip list data structure [18]. Skip list essentially is a probabilistic data structure that consists of a hierarchy of linked lists with nodes that contain forward pointers, allowing for efficient navigation between elements. The height of each node is determined randomly, with higher nodes representing fewer elements and enabling faster searches. 2 Hierarchical Navigable Small Worlds, unlike a 1-dimensional skip list, can more efficiently deal with high dimensional vectors, promising better overall performance than existing algorithms.

Besides trying to make the search itself more efficient, a previous study also suggests making the vector itself more efficient, by doing product quantization [19]. Product quantization is a technique used for compressing

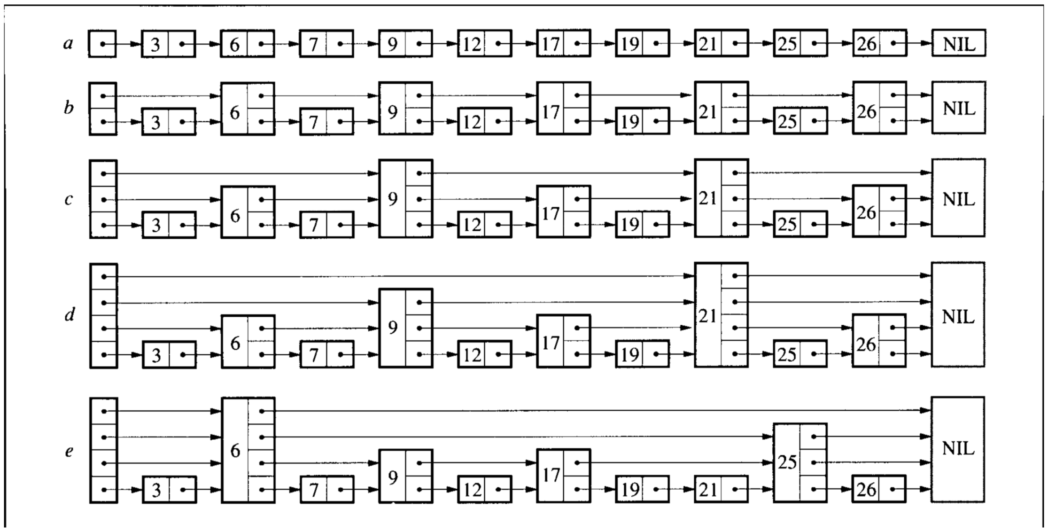


Figure 2: skip link example

data. It involves dividing the data into smaller subspaces or ”cells” and quantizing each subspace separately. The resulting quantized vectors can then be indexed and searched efficiently, reducing the storage and computation requirements. However, unlike principle component analysis, product quantization does not compress the dimension of the vector, instead, it compresses the scope of the vectors. Consequently, the algorithm is also capable of accelerating vector retrieval at the expanse of accuracy.

2.2 Our approach

In conclusion, KD-tree is suitable for low to medium-dimensional spaces but struggles with high-dimensional data. ANNOY and HNSW, on the other hand, sacrifice accuracy for speed. ANNOY constructs a forest of random projection trees, while HNSW builds a hierarchical graph structure. In the case of DPR, the large dataset makes ANNOY struggle with hashing conflict and HNSW can not handle dynamic data properly and consume huge memory.

Our approach is to build a tree by using hierarchical clustering, which can give a speed boost by sacrificing accuracy. Unlike KD-tree, the hierarchical clustering tree's performance does not relate to the dimensionality of the data, making it suitable for embeddings generated by BERT. Also, the ANNOY's drawback of dealing with a large dataset does not exist in the traditional search tree structure we are using. Finally, by maintaining a tree structure, our approach has the capability of dealing with dynamic data without re-initializing unless the change is not too big.

3 Methodology

In order to realize faster information retrieval, we build a search tree over the entire dataset by using hierarchical clustering. The clustering process was proceeding top-down over the dataset recursively and finally constructed a tree structure that can be used for faster retrieval.

Constructing the search tree may compromise the accuracy of the similarity search because the primary objective is to identify similar instances rather than exact matches. While a search tree ensures the retrieval of similar results, it cannot guarantee the absence of superior results in alternative branches. However, a search tree can significantly decrease the time complexity, and what's more, usually the approximate results are enough for search engines, which means the loss of accuracy is acceptable if it can really accelerate the search.

Then, how to build the tree, and what should the tree look like? We will explain the following from the 4 key hyper-parameters in our algorithm: similarity measurement, K-choosing strategy, clustering algorithm, and termination strategy. (see in Figure 3)

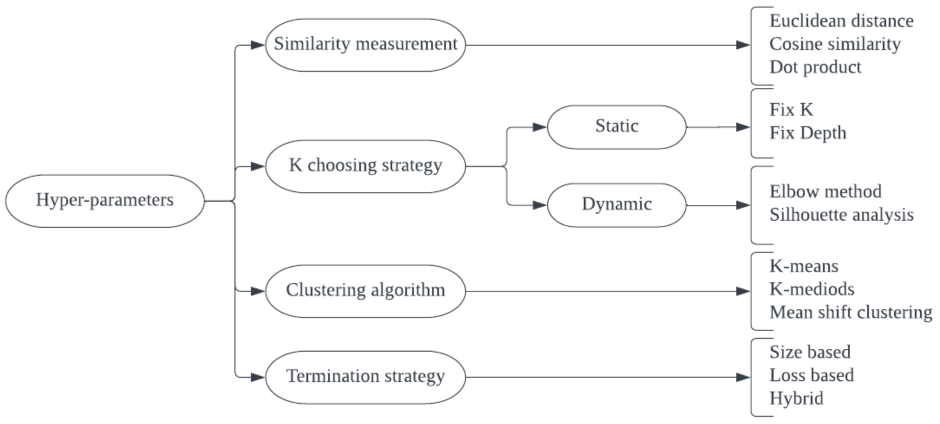


Figure 3: Hyper-parameters of the retrieval model

3.1 Similarity measurement

Similarity measurement is one of the most important hyper-parameters of the algorithm. For a retrieval model aimed at locating similar embeddings, the fundamental aspect lies in establishing the criteria for gauging the similarity between embeddings.

There are several options for this:

- **Cosine similarity:** Cosine similarity is an important and commonly used measurement for similarity between vectors, it provides a normalization to focusing on the direction of vectors.
- **Euclidean distance:** Euclidean distance is the most intuitive similarity between data points, simply by calculating the distance.
- **Dot product:** Dot product is hard to be thought of in similarity measurement, but it turns out to be a good choice after the experiment and mathematical analysis.

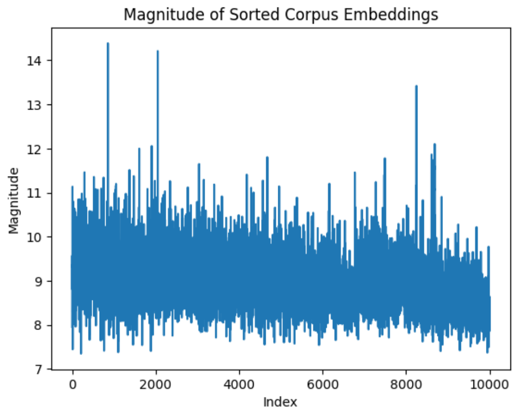
We did several rounds of experiments on all of those options. The experiment shows that both dot-product and cosine similarity gives relatively good search results. The detailed result can be found in the milestone 3 report.

Also, after mathematical analysis, we finally chose the dot product for our demo implementation, and here are two main reasons for not choosing cosine similarity.

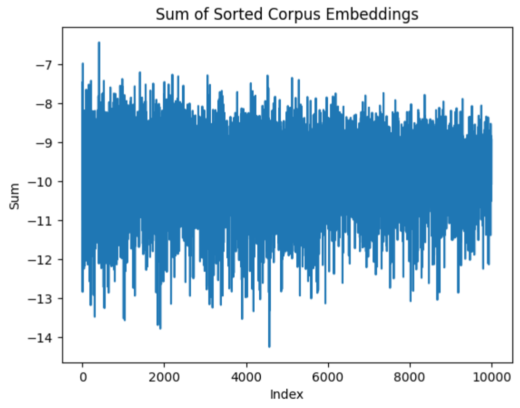
- The cosine distance normalizes the vectors to make the algorithm only sensitive to the direction of vectors instead of the magnitude of vectors. So, it will remove one important dimension of data if the magnitudes of embeddings are not noises. According to the paper of BERT network [1], it is possible that the network is more tends to output larger embeddings if the input is longer since the cumulation of attention vectors. We sorted 10000 random embeddings from the cc-news dataset based on the length of the input string (Figure 4a) and found that this influence is slight. So, in this case, introducing cosine distance might have more drawbacks than benefits.
- The cosine distance will be less efficient if the vectors are all positive/negative or almost all positive/negative. It will make vectors closer to the axis less likely to be searched, which means it introduces bias to the retrieval model. From Figure 4b, we can find that the sums of embeddings are all negative, which means the features inside embeddings are more likely to be negative numbers, which will potentially increase the bias caused by cosine similarity.

3.2 K choosing strategy & Alpha

When mentioning clustering, the K value is always the important value since it decides the number of cluster. What's more, in our tree structure, the K value also correspond to the structure of the tree, which makes it even more important.



(a) magnitude of embeddings



(b) sum of embeddings

Figure 4

Generally speaking, the K decides the number of children nodes a node can have. It influences the depth of the tree, which directly relate to the speed and accuracy of the tree structure. Smaller the K, less the children nodes for every nodes, deeper the tree, faster the search speed, and lower the accuracy. Vise versa.

The decision of K value introduced the speed-accuracy trade-off, which is a crucial concept in the model building process.

Here are the K-choosing strategies:

- **Fixed K:** Fixed K is deciding the K value before constructing the tree, which means the K remains same for the entire tree. For example, if we set K to 2, then we get a most common and intuitive binary tree, which is also the fastest and the least accurate one. Also, we can pre-calculate the K value by first limit the depth of the tree to make sure the accuracy won't fall too low.
- **Dynamic K:** Dynamic K is deciding the K independently for every node when constructing the tree, which means each node can have a different K number. It makes the tree more flexible, and can also minimize the loss caused by wrongly clustering. Theoretically, the dynamic K choosing strategy considers both the speed and accuracy, which is better than the Fixed K strategy. To control the shape of the tree, we can further introduce a parameter Alpha, which denote how much the algorithm tend to make the tree deeper. The mathematical expression of Alpha should depends on the specific clustering algorithm.

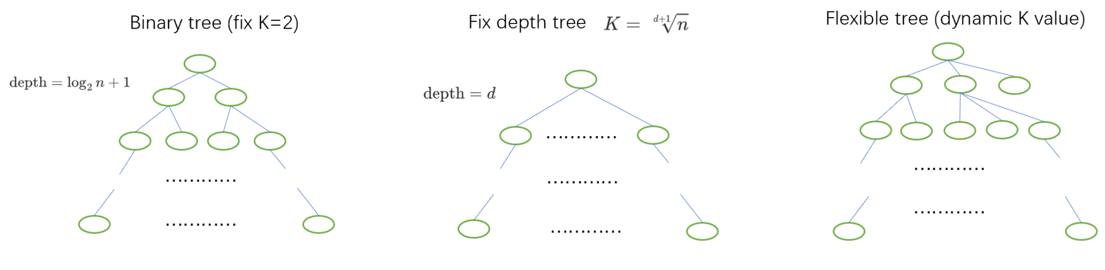

Binary tree (fix K=2)
$\text{depth} = \log_2 n + 1$
Fix depth tree $K = \sqrt[d+1]{n}$
$\text{depth} = d$
Flexible tree (dynamic K value)

Figure 5: Different k-choosing-strategy result in different search trees

Figure 5 shows how the K-choosing strategy influences the tree structure.

3.3 Clustering algorithm

The clustering method controls how we do the clustering, which is the key element for tree construction. Different clustering methods might result in different search accuracy but has no influence on the search's time complexity.

Here are the options for clustering algorithms:

- **K-means:** K-means is a widely used centroid-based clustering algorithm that aims to minimize the sum of squared distances between data points and their assigned cluster centroids. It is well-known and easy to implement.
- **K-mediods:** K-medoids is one alternative solution for K-means. Instead of using centroids, k-medoids use actual data points within the cluster as representatives. This makes k-medoids more robust to outliers. In high-dimensional datasets, outliers are more likely to appear since the high-dimensional space made the dataset more sparse. So in our case, the K-medoids is theoretically more reliable than K-means.
- **Mean shift clustering:** Mean shift is a non-parametric algorithm that doesn't require specifying the number of clusters in advance. It makes mean shift clustering fit the concept of our tree-building model pretty since it does not require the K-choosing strategy and the Alpha hyper-parameters, which makes the model easier to adjust. However, it has a weak tolerance to high dimensionality since it is based on the density of data points, and the density over high dimensional data is not very reliable. Dimensionality reduction needs to be performed first before applying this method.

Despite the advantages of K-medoids and Mean shift clustering, such as their ability to handle outliers and automatically determine the number of clusters, we have decided to use K-means in our demo model due to its versatility and widespread usage.

3.4 Termination strategy

Now we have all the key components for constructing the tree. By applying the hyper-parameters above, we can recursively divide our dataset and let the tree grow. Now comes the last question: termination. We can not keep dividing the dataset all the way down since the goal is for similarity search and the search tree should have the ability to return not only the most similar one but a list of similar documents. Thus, the dividing of data must be terminated at some condition to prevent the cluster on the leaf node to become too small (if we do not terminate, each leaf node only contains 1 document).

Here are the options for the termination strategies:

- **Size based:** The size-based termination strategy is the most intuitive one. We can simply limit the smallest size of leaf nodes. For example, if we set it to 100, then if a cluster has points less than 100, it will not be divided anymore.
- **Loss based:** The loss-based termination strategy, on the other hand, does not care about the size of the cluster but the loss inside the cluster. If a cluster has a large inside loss, it means it might not be clustered properly, so we will keep dividing it into smaller parts.
- **Hybrid:** The hybrid solution makes use of both size-based and loss-based conditions. In the size-based solution, a nice cluster might be forced to be divided into several parts only because it contains too many points. In the loss-based solution, the tree might grow too large or too small if the threshold value is not set properly. So, considering both these two factors can make better trees.

Because of the termination strategies, the tree can not become a complete tree anymore, so the retrieval speed might fall into a range instead of a fixed value due to the search route being short or long.

4 Result

The performance measured in the benchmark is as follows. 1 Here, most hyper-parameters are set to default. In terms of the index time, when only 10000 vectors are given in the 768-dimension space, the index time of each algorithm has very little absolute difference. However, K-means Tree is the slowest one in indexing, potentially because of recursive k-means operations. The Naive Search does not need index, but some python overhead gives it a measurable delay. The memory usage of each algorithm varies greatly. While the Naive Search unsurprisingly uses the least amount of the memory, because of its very low memory overhead, K-D Tree, however, does not use as much memory as other graph based methods did even with its extra memory overhead in maintaining the tree structure. The memory usage of the Lshashpy3 and Annoy differs greatly even though they are the same algorithm, potentially because of deviant implementation details.

	K-D Tree	Lshashpy3	ANNOY	HNSW	K-means Tree	Naive Search
Index time	0.0274s	0.0269s	0.0116s	0.0320s	0.3901s	1.6689e-06s
Memory usage	400MiB	900MiB	1200MiB	400MiB	3GiB	200MiB
Retrieval time (abs.)	0.0010s	0.01697s	7.512e-05s	4.553e-07s	7.5531e-05s	0.03786s
Retrieval time (rel.)	3456%	223%	50399%	50126%	8314147%	100%

Table 1: Performance of difference algorithms

	K-D Tree	Lshashpy3	ANNOY	HNSW	K-means Tree	Naive Search
Index time	0.0199s	0.0249s	0.0086s	0.0272s	0.344s	1.4305e-06s
Memory usage	400MiB	900MiB	1200MiB	400MiB	2.5GiB	200MiB
Retrieval time (abs.)	0.0004s	0.01593s	5.674e-05s	4.4043-05s	0.021609s	0.03260s
Retrieval time (rel.)	7090%	205%	57458%	74026%	175%	100%

Table 2: Performance of difference algorithms (Normalized)

All the algorithms perform better than the Naive Search in terms of retrieval time. However, difference in speed still can be found among different algorithms. The most pronounced ones are the speed difference between the Lshashpy3 and the ANNOY. They are the same algorithm, but the speed difference is great. Another thing is that the K-means Tree perform seemingly super well in terms of the retrieving. However, because this is not a normalized test, what actually happened is that the k-means model compromise its retrieval accuracy for the speed.

When we tune the hyper-parameters and normalize the average accuracy of the top 20 retrieval to about 60-80% (Naive is always 100% accurate), and redo the benchmark, the performance of the algorithm is as follows. 2 After the normalization, the general trend of the data remain the same. However, the speed of K-means Tree decreases by a significant amount, but it is still statistically faster than the naive search.

When we change the similarity functions of the K-means tree, the retrieval performance does not differ by much.3 However, when using Manhattan distance to index the vectors, the indexing performance dropped significantly. Other than that, the performance difference is not particularly pronounced.

We also tested the influence of termination strategy for the performance of the k-means tree. 4 The termination strategy has some influence on the indexing time potentially because of the amount of the calculation and the resulting tree depth are different for each strategy.

	Euclidean	Angular	Manhattan	Dot
Index time	0.07848s	0.1711s	2.0526s	0.4985s
Retrieval time (abs.)	1.4067e-07s	1.43055e-07s	1.6928e-07s	1.5497e-07s

Table 3: Performance of K-means-tree with different similarity functions

	Size	Loss	Hybrid
Index time	0.0186s	0.0921s	1.0793s
Retrieval time (abs.)	1.3925e-07s	1.304e-07s	1.659e-07s

Table 4: Performance of K-means-tree with termination strategy

5 Discussion

In conclusion, we proposed a faster similarity search over large corpus embeddings by sacrificing some accuracy. The trade-off between speed and accuracy depends on the specific hyper-parameter choices, allowing flexibility in tuning the model based on the requirements of the application.

In the future, we will consider doing a more complete benchmark over a larger dataset, getting rid of the bias in programming languages and the implementation.

Also, we will explore more optimizations in terms of both algorithms and hyperparameters. For example, what if we consider including some supervised data and learned a supervised clustering strategy first, and use the trained divider to build the tree? Or is it possible to let two parent nodes point to one child node, making the structure a graph instead of a tree? Those are all possible methods for improving accuracy.

What's more, we haven't explored much in the possibility of utilizing dimensional reduction. The t-SNE gives a nice separable result in 2d space, but the cluster's shape is not consistent. In this case, the distance-based K-means is not functioning, but the SVM can perform well since it divides space using gaps.

In summary, we have found a feasible direction in solving the similarity search problem, but there are still numerous intriguing avenues to explore, and we will keep contributing to the advancement of our approach for faster and more accurate similarity search over large corpus embeddings.

References

[1] J. Devlin, M.-W. Chang, K. Lee **and** K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," **in***Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 2019, **pages** 4171–4186. DOI: 10.18653/v1/N19-1423. **url**: https://www.aclweb.org/anthology/N19-1423/.

[2] H. Jégou, M. Douze, J. Johnson, L. Hosseini **and** C. Deng, "Faiss: Similarity search and clustering of dense vectors library," *Astrophysics Source Code Library*, ascl–2210, 2022.

[3] J. Guo, Y. Fan, Q. Ai **and** W. B. Croft, "A deep relevance matching model for ad-hoc retrieval," **in***Proceedings of the 25th ACM international on conference on information and knowledge management* 2016, **pages** 55–64.

[4] G. Salton **and** C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information processing & management*, **jourvol** 24, **number** 5, **pages** 513–523, 1988.

[5] S. Robertson, H. Zaragoza **andothers**, "The probabilistic relevance framework: BM25 and beyond," *Foundations and Trends® in Information Retrieval*, **jourvol** 3, **number** 4, **pages** 333–389, 2009.

[6] L. Van der Maaten **and** G. Hinton, "Visualizing data using t-SNE," *Journal of machine learning research*, **jourvol** 9, **number** Nov, **pages** 2579–2605, 2008.

[7] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen **and** W.-t. Yih, *Dense Passage Retrieval for Open-Domain Question Answering*, 2020. eprint: arXiv:2004.04906.

[8] F. Hamborg, N. Meuschke, C. Breitinger **and** B. Gipp, "news-please: A Generic News Crawler and Extractor," **in***Proceedings of the 15th International Symposium of Information Science* Berlin, **march** 2017, **pages** 218–223. DOI: 10.5281/zenodo.4120316.

[9] J. Nievergelt, H. Hinterberger **and** K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM Trans. Database Syst.*, **jourvol** 9, **number** 1, **pages** 38–71, **march** 1984. DOI: 10.1145/348.318586. **url**: https://doi.org/10.1145/348.318586.

[10] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, **jourvol** 18, **pages** 509–517, 9 1975. DOI: 10.1145/361002.361007. **url**: https://doi.org/10.1145/361002.361007.

[11] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," **in***Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data* **jourser** SIGMOD '81, Ann Arbor, Michigan: Association for Computing Machinery, 1981, **pages** 10–18. DOI: 10.1145/582318.582321. **url**: https://doi.org/10.1145/582318.582321.

[12] P. Ciaccia, M. Patella **and** P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," **in***Proceedings of the 23rd International Conference on Very Large Data Bases* **jourser** VLDB '97, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, **pages** 426–435.

[13] S. M. Omohundro, "Five Balltree Construction Algorithms," International Computer Science Institute, techreport TR-89-063, **december** 1989.

[14] R. Weber, H.-J. Schek **and** S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," **in***Proceedings of the 24rd International Conference on Very Large Data Bases* **jourser** VLDB '98, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, **pages** 194–205.

[15] P. Indyk **and** R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," **in***Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing* **jourser** STOC '98, Dallas, Texas, USA: Association for Computing Machinery, 1998, **pages** 604–613. DOI: 10.1145/276698.276876. **url**: https://doi.org/10.1145/276698.276876.

[16] H. Wang, J. Cao, L. Shu **and** D. Rafiei, "Locality Sensitive Hashing Revisited: Filling the Gap between Theory and Algorithm Analysis," **in***Proceedings of the 22nd ACM International Conference on Information amp; Knowledge Management* **jourser** CIKM '13, San Francisco, California, USA: Association for Computing Machinery, 2013, **pages** 1969–1978. DOI: 10.1145/2505515.2505765. **url**: https://doi.org/10.1145/2505515.2505765.

[17] Y. A. Malkov **and** D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *CoRR*, **jourvol** abs/1603.09320, 2016. arXiv: 1603.09320. **url**: http://arxiv.org/abs/1603.09320.

[18] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees," *Commun. ACM*, **jourvol** 33, **number** 6, **pages** 668–676, **june** 1990. DOI: 10.1145/78973.78977. **url**: https://doi.org/10.1145/78973.78977.

[19] H. Jégou, M. Douze **and** C. Schmid, "Product Quantization for Nearest Neighbor Search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **jourvol** 33, **number** 1, **pages** 117–128, 2011. DOI: 10.1109/TPAMI.2010.57.