Fast Exact Retrieval for Nearest-neighbor Lookup (FERN)

Richard Zhu
Princeton University
ryzhu@princeton.edu

Abstract

Exact nearest neighbor search is a computationally intensive process, and even its simpler sibling — vector retrieval — can be computationally complex. This is exacerbated when retrieving vectors which have high-dimension d relative to the number of vectors, N, in the database. Exact nearest neighbor retrieval has been generally acknowledged to be a $O(Nd)$ problem with no sub-linear solutions. Attention has instead shifted towards Approximate Nearest-Neighbor (ANN) retrieval techniques, many of which have sub-linear or even logarithmic time complexities. However, if our intuition from binary search problems (e.g. $d = 1$ vector retrieval) carries, there ought to be a way to retrieve an organized representation of vectors without brute-forcing our way to a solution. For low dimension (e.g. $d = 2$ or $d = 3$ cases), kd-trees provide a $O(d \log N)$ algorithm for retrieval. Unfortunately the algorithm deteriorates rapidly to a $O(dN)$ solution at high dimensions (e.g. $k = 128$), in practice. We propose a novel algorithm for logarithmic Fast Exact Retrieval for Nearest-neighbor lookup (FERN), inspired by kd-trees. The algorithm achieves $O(d \log N)$ look-up with 100% recall on 10 million $d = 128$ uniformly randomly generated vectors.[1]

1 Introduction

Vector retrieval is pervasive, underlying search engines, transformers, and open-book language models. At heart, one of key attributes of computing systems lie in their ability to retrieve knowledge. Sometimes, when this knowledge is sufficiently broad — and a powerful enough retrieval architecture is built — these computing systems may even be so good at retrieving relevant knowledge that they appear to reason (Bubeck et al., 2023).

Given the power of knowledge retrieval for both commercial and academic pursuits, significant energy has been devoted towards effectively converting various types of data into vectors, from words (Mikolov et al., 2013), images (Radford et al., 2021), and audio (Radford et al., 2022) to documents, sentences, and paragraphs (Dai et al., 2015).

In this work, we differentiate between look-up and search. Look-up involves the retrieval of vectors guaranteed to be contained in the database, while search involves the retrieval of queries not necessarily contained in the database. Retrieving queries without exact matches may involve instead retrieving that vector's nearest neighbors. The definition of nearest can be further disambiguated into Euclidean or cosine similarity measures, among others. Note that under the hood, a Euclidean distance-based nearest neighbor algorithm can be easily adapted to be cosine similarity-based simply by dividing each vector in the database by its magnitude during insertion. During look-up, the query vector is then also divided by magnitude. The resulting nearest neighbors we obtain are also such by cosine similarity.

The scalable and effective look-up of large numbers of high dimensional vectors is thus desired. While the vanilla hashmap algorithm provides $O(1)$ time complexity for scalars, extending to $O(d)$ for vectors in d-dimensional space, this holds only when the cardinality of the hash function range is large relative to the number of elements, N. When the number of elements becomes large relative to the number of bins, b, finding the key within each bin becomes a linear search problem. Since we expect each bin to have $\frac{N}{b}$ collisions, the time complexity for look-up is $O(\frac{Nd}{b})$. While we can also get fast look-up with a heap in low dimensions, FERN can be thought of as an extension of the heap to high dimensions while also presenting a novel approach that could lead to sub-linear vector search.

---

[1]Code available at https://github.com/RichardZhu123/ferns

2 Related work

Prior work has taken 3 major directions to attain fast nearest neighbor search. The approaches involve bucketing, divide and conquer, or graph-based approaches. These techniques are specifically tuned to work well for vectors in high dimensional space, which should be unsurprising since each of these techniques is really just an extension of familiar 1-d concepts: hashmaps, binary search, and breadth first search. I believe there are key learnings that can be taken from both exact and approximate retrieval and search settings, even though we are interested in the exact variant. Some new techniques such as Certified Cosine certificates (Francis-Landau and Van Durme, 2019) offer structure-agnostic tweaks to speed up performance.

2.1 Bucketing

Locality Sensitive Hashing (Indyk and Motwani, 1998) and k-means (Lloyd, 1982; MacQueen, 1967), including more recent variants like k-means++ (Arthur and Vassilvitskii, 2007), are two techniques that use bucketing to group database vectors with their nearby peers. Since it is hard to deal with queries that fall on the boundaries of adjacent clusters, these algorithms are approximate search algorithms. Both of these algorithms in practice take linear time for sufficiently high dimensions and large numbers of elements.

Divide and conquer kd-trees provide strong algorithms for pruning, with new variants attempting to decrease constant factors in the time complexity. (Zhang et al., 2012) A recent work (Ram and Sinha, 2019) attempts to create partitions based on random rotations of the dataset achieve the same search accuracy guarantees as RPTree (Dasgupta and Sinha, 2013) but with $O(d \log d + \log n)$ time complexity for approximate search.

Graph-based approaches Recent graph-based approaches to search, such as Navigable Small World (NSW) graphs (Malkov et al., 2014) and a later variant involving layers of NSWs, provide increasing granularity. NSWs are graphical representations of databases where each pair of nodes is connected by a small number of hops. Further optimizations have been presented (Fu et al., 2019; Jayaram Subramanya et al., 2019).

3 FERN

Goal We aim to build an algorithm that satisfies two primary goals: we must be able to perform quick look-up on vectors guaranteed to be contained in the database and we must be able to quickly insert vectors. When designing the algorithm, we initially assume our database contains N vectors, each spanning d dimensional space. The i-th vector v_i is defined as follows

$$\mathbf{v_i} = [v_{i,1}, v_{i,2}, \ldots, v_{i,d}]^\top$$
$$\text{where } v_i \sim \mathcal{R}(-1, 1)$$

This process effectively generates vectors lying within a d dimensional "ball"-like shape of radius 1 in each direction. The directions in which the vectors point are also evenly distributed direction-wise. We discover later however, that the algorithm we design with this simplifying assumption maintains logarithmic time lookup for $\mathbf{v_i}$ of any arbitrary direction and length.

In terms of time complexity, we define quick as anything taking logarithmic time — this means that lookup in a database of ten billion vectors should only take seven times longer than lookup in a database of a thousand vectors. That is remarkable, because a naive linear search would take ten million times longer, a nearly intractable time difference at scale. Since a logarithmic lookup time and linear space complexity is state of the art (SoTA), we believe a key contribution of our work is an alternative data structure and algorithm that achieves SoTA while simultaneously providing the capacity to be extended to logarithmic-time nearest neighbor search, given its unique approach to dividing the vectors by hyperplanes defined based on the vectors in the database rather than measuring along a specific direction like the traditional kd-trees process. Each node is an object that stores a vector, pointers to the left and right children, and a pointer to that node's parent node. This results in a binary tree with undirected edges. While we ultimately implement retrieval using a queue structure, this bidirectional edge only adds marginal complexity to the algorithm and underlying data structure while enabling a backtracking-based traversal method. The queue-based method emulates a level-order traversal of candidate nodes while a stack-based backtracking-based traversal method (that fully explores a specific path before backtracking; exploring each sibling node that could not be pruned

without potentially missing the nearest neighbor) emulates a depth-first search.

Methodology We design a novel algorithm that is a variant of kd-trees, but has the capacity to perform better at higher dimensions. Broadly, the structure is a binary tree. Each node that has both left and right children defines a hyperplane using the vectors of its left and right children as support vectors.

The tree is constructed so that all vectors in each child's subtree are on the same side of the hyperplane as that child. This allows us to perform vector look-up in logarithmic time, provided that vectors are added to the database in a sufficiently random manner. It is feasible, however, for an adversarial insertion process to result in a heavily imbalanced tree and consequently worst-case linear look-up time. While we don't observe this as an issue in practice, we can resolve the issue by implementing a slightly more complicated variant of FERN - using a variant of the Red-Black Tree technique (Guibas and Sedgewick, 1978) to guarantee balanced trees, logarithmic depth, and thus logarithmic retrieval time complexity.

There are two key components to our algorithm: one method for insertion (Algorithm 1) and another for lookup (Algorithm 2).

The insertion algorithm (Algorithm 1) is fairly concise. When inserting a vector into the tree, it is placed at the root if the tree has not been initialized yet. Otherwise, if the current node is missing a left or right child, we insert the vector as a child node. If the node is a leaf node — that is, missing both left and right children — then the left child is always inserted first, before the right child.

During insertion, if a node has both left and right children, then we set the current node instead to the child node that is closest to the vector we are inserting. That is, if we form a hyperplane from the set of points equidistant to both left and right children, then we set the current node to the left child if the vector to be inserted lies on the same side of the hyperplane as the left child, otherwise we set the current node to the right child.

The result of this insertion algorithm is that — for a balanced binary tree — we get a maximum tree depth of $O(\log N)$ where N is the number of elements in the database. Insertion time per vector is thus $O(\log N)$ since we only visit one node per depth level.

When looking up vectors from the data structure,

Algorithm 1 FERN Insertion

1: **Function** Insert(vector)
2: **if** root not initialized **then**
3:     root ← VectorNode(null)
4: **end if**
5: node ← root
6: **while** True **do**
7:     **if** no left child **then**
8:         set left node to vector
9:         **break**
10:     **else if** no right child **then**
11:         set right node to vector
12:         **break**
13:     **else if** vector closer to left child **then**
14:         node ← node.left
15:     **else**
16:         node ← node.right
17:     **end if**
18: **end while**

---

we demonstrate a method (Algorithm 2) that has a per-vector retrieval time proportional to the maximum depth of the tree, since we only look at one node per depth level. However, when extended to search settings where the query is not known to be contained in the database, retrieval time becomes proportional to the number of elements in the tree. We can no longer automatically prune any queries that lie close to the hyperplane boundary since there is a possibility that the nearest neighbor and query may lie on different sides of the hyperplane.

Ostensibly, we would expect a non-negligible proportion of vectors to be sufficiently far from the hyperplane to be pruned. However, we notice that in practice, as the dimensionality of the vectors increase, so too does the proportion of vectors lying close to the boundary. This makes sense intuitively since we are trying to project increasingly higher dimensions of vectors onto a 1-d line (the line normal to the hyperplane and passing through both support vectors). During retrieval, we effectively perform the depth-first or level-order search, as described previously. For a balanced tree with strong boundaries (that is, most queries lie far away from the hyperplane), per-vector time should be logarithmic with respect to the number of elements already present in the database (another word for our proposed data structure) at insertion-time. However, it becomes linear otherwise. We first define `mip` and `mip_vec`, the distance to the nearest neighbor found thus far and the vector representing the near-

44

3

est vector retrieved thus far. We then create a queue and insert the root node.

Algorithm 2 FERN Lookup

Require: query
Ensure: mip_vec
 1: mip, mip_vec $\leftarrow \infty$, None
 2: curr $\leftarrow$ None
 3: queue $\leftarrow$ [self.root]
 4: **while** queue not empty **do**
 5:     curr $\leftarrow$ oldest element in queue
 6:     update mip, mip_vec if curr closer to query

 7:     **if** curr has both left and right children **then**
 8:         **if** query is closer to left child **then**
 9:             queue.append(curr.left)
10:             **if** query close to boundary **then**
11:                 queue.append(curr.right)
12:             **end if**
13:         **else**
14:             queue.append(curr.right)
15:             **if** query close to boundary **then**
16:                 queue.append(curr.left)
17:             **end if**
18:         **end if**
19:     **else if** curr has left child only **then**
20:         queue.append(curr.left)
21:     **else if** curr has right child only **then**
22:         queue.append(curr.right)
23:     **end if**
24: **end while**
25: **return** mip_vec

We then continuously pop a node from the head of the queue until the queue is empty. Each time we pop a node, we check whether its vector is closer to the query vector than the current best candidate for nearest neighbor, mip_vec, which is a Euclidean distance mip away from the query. For lookup, we are looking for an exact match, so we are seeking an mip of 0. If the node has a left child only or a right child only (the latter should never happen, but we have it as a redundancy against exceptions) then we add that node to the queue. Otherwise, if both children exist then we add to the queue the node that shares the same side of the hyperplane as the query. For lookup cases, we consider any query to be sufficiently far from the boundary that only one child node needs to be added to the queue per node. After all, whether a query is close to the boundary is somewhat arbitrary and the exact function definition depends on whether we are performing lookup

or search.

For mapping applications, we can add an additional variable, data (a byte array), to the Node class.

In Algorithms 1 and 2, "closeness" is quantified by Euler distance, and the "boundary" is the midpoint between left and right child nodes.

4 Experimental results

During experiments, we typically utilize $d = 128$ with the same uniform distribution previously assumed. While this may not be characteristic of all data distributions, we note that our architecture is actually agnostic to the distribution of the vectors being inserted. What matters (in terms of potential effects on performance) is the order in which vectors are inserted based on their relative positions.

To run our experiments, we use the Intel Xeon Platinum 8380 CPU (2.30 GHz), the same processor used for running the popular ann-benchmark (Bernhardsson, 2024). For values of N we use 10^4, $5 * 10^4$, 10^5, $5 * 10^5$, and 10^6, $5 * 10^6$, 10^7. The last setting, equivalent to look-up on 10 million vectors, has comparable values of N and d to many of the Euclidean distance based benchmarks in ann-benchmark. We notice a nearly perfect logarithmic time complexity, and at $N = 10^7$ we run approximately 3000 retrievals/second without additional optimizations.

5 Discussion

During our experiments, we noticed that in the search modality, system dynamics can change drastically based on the dimensionality of the vectors. We experiment with different ways of defining the hyperplanes and various algorithms that would balance or repair the tree to try to guarantee logarithmic retrieval for large d and N. Unfortunately, while these algorithms almost universally gave logarithmic time complexity for 100% recall, the performance broke down drastically beyond $d = 2$ or $d = 3$. In particular, we note the importance of having well-defined hyperplane boundaries.

Boundary sharpness We want to be able to maximize pruning since we achieve $O(\log_2 N)$ time complexity when we prune 50% of the nodes in the database each time we measure the distance between a node and the query). Indeed, note that since a k-means based nearest neighbors search allows us to prune up to N/k nodes per comparison, we might wonder why 2-means search doesn't achieve
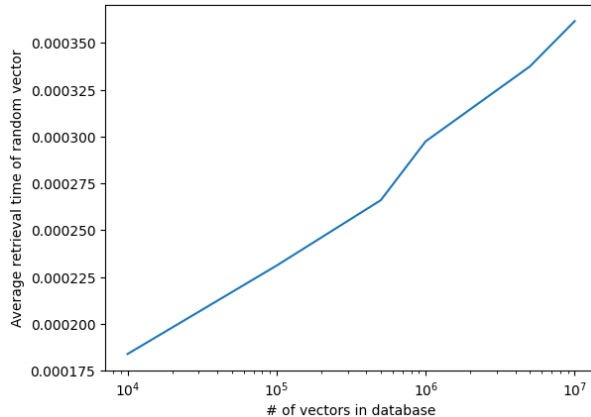
Figure 1: FERN lookup with vectors where $d = 128$ and look-up time is averaged over 1000 vectors randomly sampled from the database
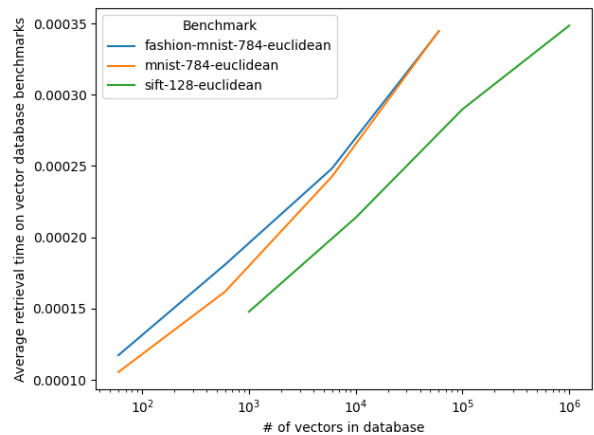

Figure 2: FERN lookup using the train portion (60k-100k vectors) of popular Euclidean-distance-based vector retrieval benchmarks and look-up time is averaged over 1000 vectors randomly sampled from the database. We evaluate 4 decades on each dataset, which is why SIFT-128-Euclidean evaluation starts with vector databases of size 10^3 rather than 600

$O(\log N)$ complexity. It's because k-means has an $O(N/k + k)$ time complexity, the N/k term means that we would need to do a linear number of searches regardless of the number of clusters.

We observe empirically that the proportion of nodes in the database that are visited increases sharply when there are more nodes that are closer to the hyperplane than the support vectors that define the plane. That is to say, during the insertion process, there will be vectors that lie between a support vector and the hyperplane. Now when we're retrieving, the query may lie on one side of the hyperplane but its nearest neighbor may be one of these "in-between" nodes on the other side of the hyperplane. This means we now must be much more prudent when pruning which decreases the proportion of vectors that are pruned and thus increases the time complexity.

However in the look-up modality, we achieve logarithmic time complexity on both a vector database of dimension $d = 128$ and size 10^7 that are randomly and uniformly generated (Figure 1) and three Euclidean benchmarks (Table 1) from ann-benchmark (Figure 2), using the larger train splits ($N = 60,000$ to $N = 1,000,000$). We observe logarithmic time complexity over a diverse dimensions and vector distributions.

Dataset	Dim.	Train Size	Test Size
Fashion-MNIST	784	60,000	10,000
MNIST	784	60,000	10,000
SIFT	128	1,000,000	10,000

Table 1: Properties of ANN Benchmark datasets used for evaluation

6 Conclusion

We are able to achieve our goal of creating a novel vector database structure that achieves state of the art look-up time complexity that is logarithmic in the number of vectors. The algorithm presented here, FERN, maintains 100% recall while performing lookup on vectors in high-dimensional space (e.g. $d = 128$ to $d = 784$) with N varying from 10^2 to 10^7, and presents a potential path towards a data structure and algorithm that will allow for the first sub-linear exact nearest neighbor retrieval process.

We believe that the exact process for attempting to perform binary search on a vector database requires carefully defined hyperplanes, which presents an area for further work. We find the "fixing" step of the Red-Black tree algorithm to be particularly inspirational as a direction of future work. Evaluation on additional datasets and further investigations of recall-retrieval-time trade-offs could help in the pursuit of sub-linear search.

We further also believe that an alternative for hyperplanes is to use a graph based approach, similar to the approach taken in many recent works (Malkov et al., 2014; Fu et al., 2019; Jayaram Subramanya et al., 2019), since this could allow us to more easily divide the database in a well-defined and easy to update way. Overall we are excited by the potential and hope to further develop this algorithm in pursuit of sub-linear exact search.

References

David Arthur and Sergei Vassilvitskii. 2007. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 1027–1035, USA. Society for Industrial and Applied Mathematics.

Erik Bernhardsson. 2024. erikbern/ann-benchmarks. Original-date: 2015-05-28T13:21:43Z.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4.

Andrew M. Dai, Christopher Olah, and Quoc V. Le. 2015. Document Embedding with Paragraph Vectors. ArXiv:1507.07998 [cs].

S. Dasgupta and Kaushik Sinha. 2013. Randomized partition trees for exact nearest neighbor search. *ArXiv*.

Matthew Francis-Landau and Benjamin Van Durme. 2019. Exact and/or Fast Nearest Neighbors. ArXiv:1910.02478 [cs].

Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment*, 12(5):461–474.

Leo J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. ISSN: 0272-5428.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing - STOC '98*, pages 604–613, Dallas, Texas, United States. ACM Press.

Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137. Conference Name: IEEE Transactions on Information Theory.

J. MacQueen. 1967. Some methods for classification and analysis of multivariate observations.

Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. ArXiv:1301.3781 [cs].

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. ArXiv:2103.00020 [cs].

Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022. Robust Speech Recognition via Large-Scale Weak Supervision. ArXiv:2212.04356 [cs, eess].

Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1378–1388, Anchorage AK USA. ACM.

Yuezhuo Zhang, Li Zha, Jia Liu, and Zhiwei Xu. 2012. A Large-Scale Online Search System of High-Dimensional Vectors Based on Key-Value Store. In *2012 Eighth International Conference on Semantics, Knowledge and Grids*, pages 233–236.