Comparative Study on k-NN Algorithms with PUFFINN, HSNW, FAISS-IVF

Team member: Yujia Fan - PUFFINN
Zhiqi Yang – HSNW
Heyang Liu – Faiss-IVF
Nov. 26th, 2019

1. Summary

Our project implement three approximate nearest neighbor algorithms from three categories: PUFFINN(LSH-based), HSNW(graph-based) and FAISS-IVF(quantization-based).
We explain theories behind these different approaches briefly. What kind of data structure are used to hold the nearest neighbor data? What procedures the algorithms take to build such a data structure? What are the core ideas/techniques each of them uses to embody similarities between data points? How the algorithm performs queries to search for the nearest neighbors.
We apply our programs first on dataset GloVe 50 as first round of experiments. We get metrics of query time, index time, index size, recall, to study the behavior of the algorithms. We further tune parameters to analyze their effects on the results.

Choice of parameters	Observed results (effect on metrics)
expected recalls	Actual recall always beats or meets expected recall
hashing functions	'SimHash' is more efficient
hash sources	'Independent' is more optimal also by default
filters	'Default' is most optimal which stops after achieve target recall rate

Table 1: Comparison of Parameters in Puffinn

Choice of parameters	Observed results (effect on metrics)
ef_construction (time/accuracy tradeoff)	ef is between k and data size, greater ef causes longer indexing & query time, but recall rate converges after ef values large enough
M (max # of connections in graph)	Determines memory usage, bigger M leads to higher recall, longer index and query time, index size solely dependent on M

Table 2: Comparison of Parameters in HSNW

Choice of parameters	Observed results (effect on metrics)
M (# of lists will be divided into)	M is the number of lists that original list will be divided into, bigger M will increase the query time and the accuracy of the estimation
n-probe (# of center of clusters will visit)	The default value for n-probe is 1, different values of n-probe will change the run time and accuracy.

Table 3: Comparison of Parameters in Faiss

We also introduce some add-in nice features of the algorithms of empirical appeals. We further study the scalability by applying our program on datasets of different dimensions: GloVe 25/50/100/200. We compare the algorithms with each other for obvious differences in their metrics or metric patterns and proposes explanations which root from their theoretical approach.

2. Objectives of the Experiments

The basic purpose of our project is to learn how to implement fast searching of nearest neighbors by utilizing existing start-of-art ANN algorithms. Based on our experiments, we want to answer the following implementation-level and high-level questions.

Q1. Given the choices of the same dataset and evaluation parameters, how does these three algorithms compare to each other with respect to empirical performance, like index time, index size, query time and actual recall?
Q2. What is the influence of the specific parameters in each algorithm to its quality/Query time?

Q3. What are the potential reasons for their empirical performances according to their theoretical methodologies?

3. Description

3.1 Description of HSNW

Navigable Small World Graph is a graph in which neighbors of any given node are likely to be neighbors of each other and most nodes can be reached from every other node by a small number of hops. HNSW is an approximate K-nearest neighbor search approach based on navigable small world graphs with controllable hierarchy. It incrementally builds a multi-layer structure consisting from hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. This allows producing graphs similar to the NSW structures while additionally having the links separated by their characteristic distance scales. The Hierarchical NSW idea is portrayed by fig1 left and fig1 right shows how a greedy search algorithm works.

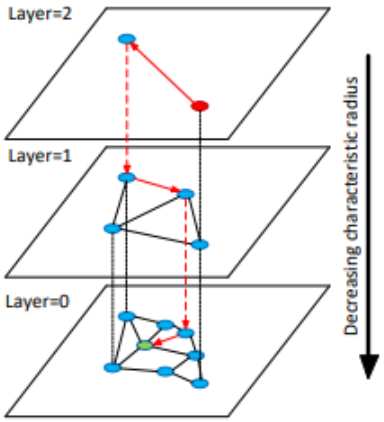


Fig. 1. Illustration of the Hierarchical NSW idea. The search starts from an element from the top layer (shown red). Red arrows show direction of the greedy algorithm from the entry point to the query (shown green).

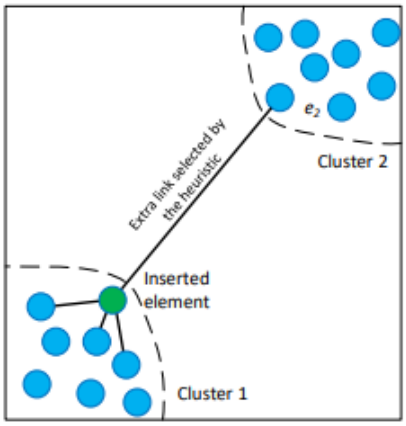


Fig. 2. Illustration of the heuristic used to select the graph neighbors for two isolated clusters. A new element is inserted on the boundary of Cluster 1. All of the closest neighbors of the element belong to the Cluster 1, thus missing the edges of Delaunay graph between the clusters. The heuristic, however, selects element e_2 from Cluster 2, thus, maintaining the global connectivity in case the inserted element is the closest to e_2 compared to any other element from Cluster 1.

Figure 1: Illustration of the Hierarchical NSW Idea and the Heuristic Algorithm

3.2 Description of PUFFINN

PUFFINN, short for Parameterless and Universally Fast FInding of Nearest Neighbors, is a parameterless LSH-based index for solving the k-nearest neighbor problem with probabilistic guarantees. By parameterless it means that the user is only required to specify the maximum amount of memory the index is supposed to use and the result quality that should be achieved. By small adaptions to the query algorithm, the index combines several heuristic ideas and makes heuristics rigorous. PUFFINN excels other state-of-art approaches mainly as follows:

1. It presents a parameterless and universal LSH implementation that solves the exact k-NN problem with probabilistic guarantees.
2. It proves the correctness of an adaptive query mechanism building on top of the LSH forest data structure.
3. It describes an adaptive filtering approach to decrease the number of expensive distance computation.

Puffinn uses LSH tires as its data structure. This data structure is parameterized by integers L, $K \geq 1$ and will consist of a collection of L LSH tries of max depth K. For the query algorithm Puffinn retrieves the points in each trie that collide with the query point in a bottom-up fashion. To reduce the number of LSH evaluations, Puffinn provides the following techniques:

1. **Tensoring**: Assume that K is an even integer and L is an even power of two. Form two collections of $\sqrt{L}$ tuples of $K/2$ LSH functions. Each trie in the LSH forest is now indexed by $j_1, j_2 \in \{1, \ldots, \sqrt{L}\}$. The K LSH functions used in the (j_1, j_2)st trie are taken by interleaving the $K/2$ functions in the j_1st tuple of the first collection and the j_2nd tuple of the second collection. This allows us to construct L LSH tries of max depth K using only $\sqrt{L}K$ independent functions.
2. **Pooling:** Form a pool of m independent LSH functions that will be shared among LSH tries. For each LSH trie in the LSH Forest we independently sample a random subset of K LSH functions from the pool that we use in place of fully random LSH functions. As m increases LSH functions sampled without replacement from the pool will work almost as well as independent samples from the LSH family.
3. **Sketching:** Depending on the LSH scheme used and the distribution of distances between the query point and the data, using 1-bit sketches can replace many of the expensive distance computations performed by the query algorithm with cheaper distance estimations through

The workflow of the Adaptive-KNN(q, k, δ) is described as below, which returns a set of k points, each one being with probability at least $1 - \delta$ among the closest k points to q.

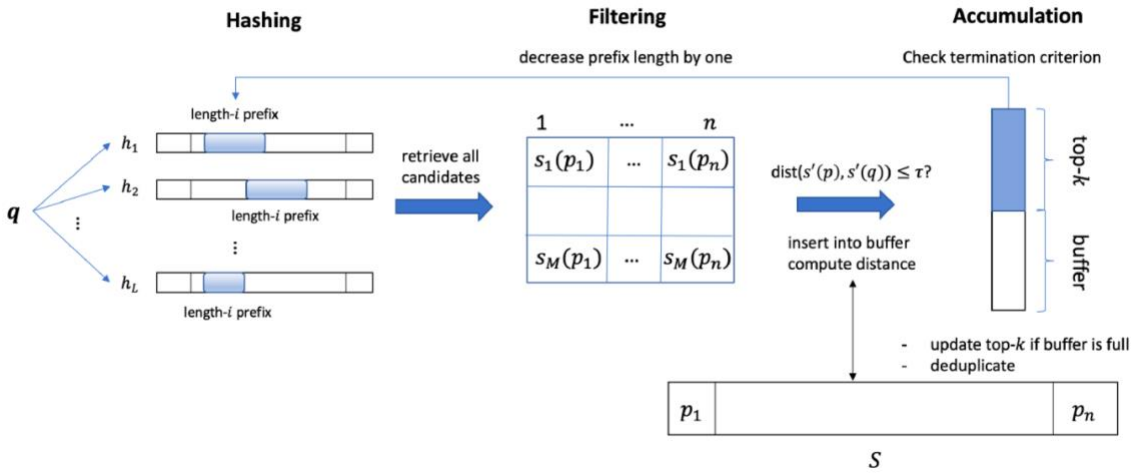


Figure 2: Workflow of Puffinn

3.3 Description of FAISS-IVF

Faiss is a library that is designed by Facebook to perform efficient similarity search. The algorithms take different types of inputs and builds up and train high dimensional convolutional neural networks. Faiss basically combines similarity search and classification, we need the following operations:

- Given a query vector, return the list of database objects that are nearest to this vector in terms of Euclidean distance.

- Given a query vector, return the list of database objects that have the highest dot product with this vector.

Faiss are currently not sufficient for database search operations because query the database directly is not impractical because they're optimized for hash-based searches or 1D interval searches. On the contrary, faiss has the following advantages:

- Faiss provides several similarity search methods that span a wide spectrum of usage trade-offs.
- Faiss is optimized for memory usage and speed.
- Faiss offers a state-of-the-art GPU implementation for the most relevant indexing methods.

To query and data, faiss will first build the model by clustering the dataset into different groups and query the index for searching data.

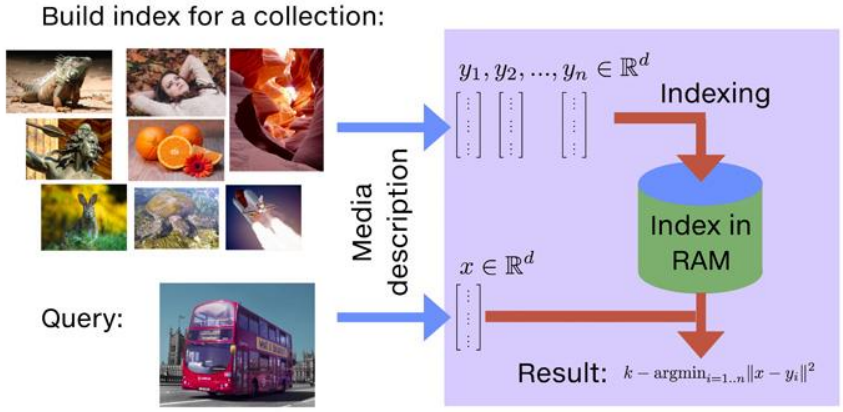


Figure 3: Workflow of Faiss-IVF

4. Experiment Details

4.1 Datasets:

Table 1 gives an overview of the datasets used in the experiments. All data sets are pre-split into train/test and come with ground truth data in the form of the top 100 neighbors and they are all stored in a HDF5 format.

Dataset	Dimensions	Train size	Test size	Neighbors	Distance	Size
Glove 25	25	1,183,514	10,000	100	Angular	121MB
Glove 50	50	1,183,514	10,000	100	Angular	235MB
Glove 100	100	1,183,514	10,000	100	Angular	463MB
Glove 200	200	1,183,514	10,000	100	Angular	918MB

Table 3: Datasets

4.2 Experiment environment:

PUFFINN, HNSW, IVF are implemented in C++ and come with a wrapper to the Python language. Experiments were run on 2x 32-core Intel Xeon E5-2687W (3.10Ghz) with 256GB of RAM using Ubuntu 18.04 (iLabU1.cs.rutgers.edu). It is compiled using g++ with the compiler flags -std=c++14 -Wall -Wextra -Wno-noexcept-type -march=native -O3 -g -

fopenmp. Index building was multi-threaded, queries where answered sequentially in a single thread. All experiments were conducted in Python 3.6 and Jupyter notebook was used to give graphical evaluation.

5. Evaluation

Quality and Performance Metrics As quality metric we measure the individual recall of each query, i.e., the fraction of points reported by the implementation that are among the true k-NN in the k ground-truth neighbors given by the 100 nearest neighbors in the Glove datasets. As performance metric, we record individual query time, i.e., the total query time for each algorithm to find all k-NNs. Unless stated otherwise, all experiments were carried out with $k = 30$.

Parameter Choices In general for three algorithms, there are three kinds of parameters needed to be decided during the comparison of other parameters. The base criteria for $dimension = 50$, $k = 30$, $expected\ recall = 80\%$. That means we will typically use Glove 50 to retrieve the 30 nearest neighbors with an expected recall as 80%.

5.1 Evaluation in Scalability

We compared the following implementations: PUFFINN, a LSH implementation with recall guarantees; HNSW, an approximate k-NN search approach based on navigable small world graphs; IVF, a k-means clustering based approach using quantization with Glove angular datasets of different dimensions (25/50/100/200) to explore their scalability.

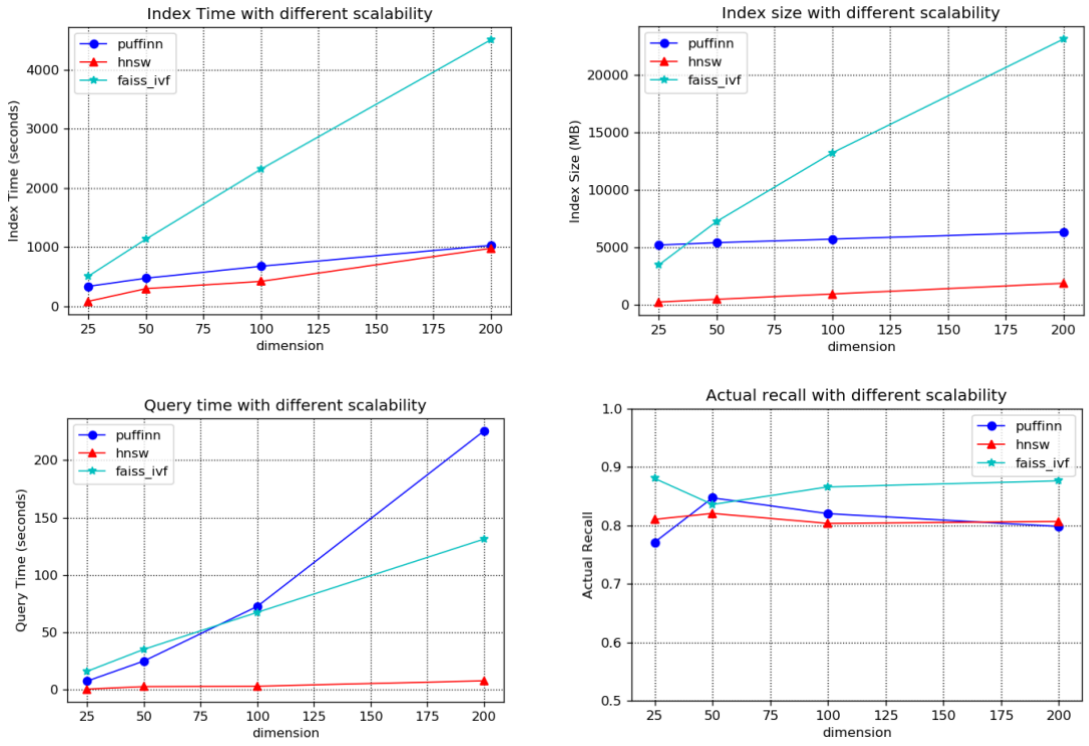


Figure 5: Comparison in Scalability

With the actual recall remaining the same for three algorithms, the indexes (index time, index size, query time) maintain linear growth as the dimension increases. The query time for PUFFINN algorithm increased dramatically comparing with other two methods since PUFFINN

holds a probability warrant so that it took much more time to ensure the actual recall meets the satisfication.

5.2 Evaluation of Puffinn

5.2.1 Comparison with different expected recalls

One of outstanding contributions of Puffinn is that it solves the exact k-NN problem with probabilistic guarantees. This guarantee is given by the user as p to ensure each of the nearest neighbors has at least this probability of being found in the first phase of the algorithm.

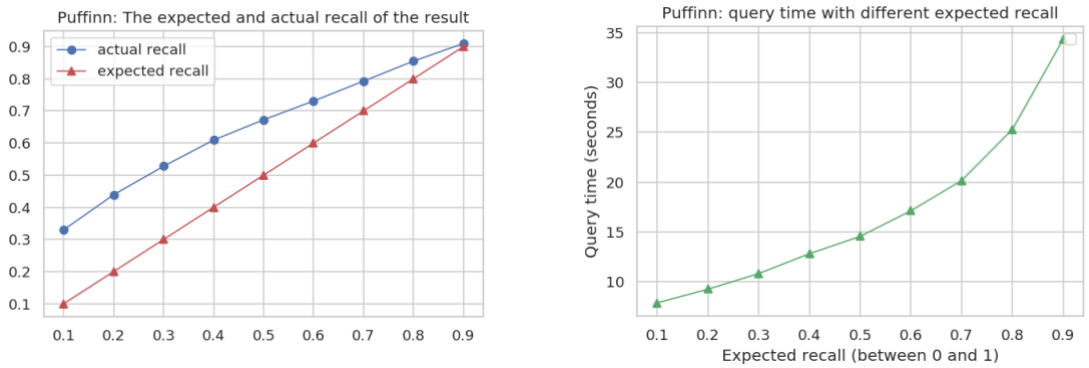


Figure 6: Influence of different expected recalls with space 500MB

In the figure 1 left, the actual recalls of the experiment (the blue line) is always above the expected recalls given by the user (the red line), it proves that Puffinn's guarantee on recall. And with expected probability increase, the total query time increases as checking those results takes more time.

5.2.2 Comparison with different k neighbors

We run PUFFINN with different k neighbors in the set $\{10, 30, 50, 100\}$ as 100 is the maximum ground-truth nearest neighbors given by the dataset.

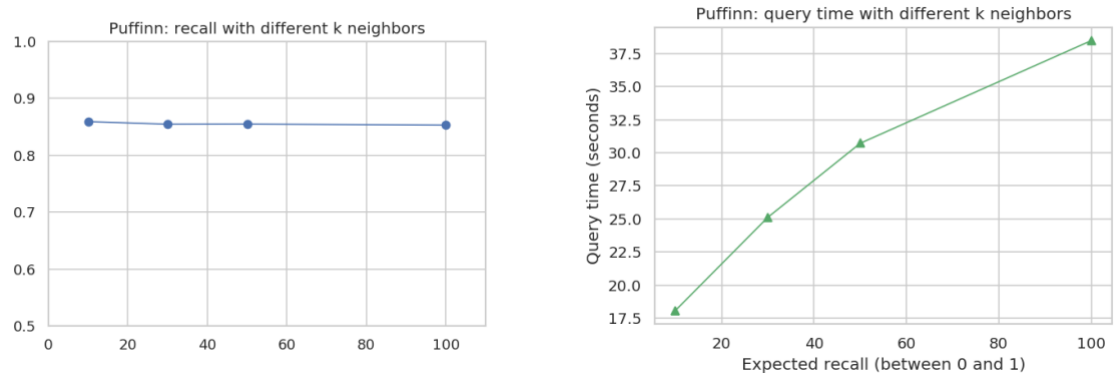


Figure 7: Influence of different expected recalls with space 500MB

The result recalls remain the same as number of required neighbors increasing. The total query time almost linearly increase. And this is understandable with the thoughts if the query size becomes larger, the query time becomes longer.

5.2.3 Comparison with different Hash functions

Glove dataset in calculated in angular distance which measures the cosine value of the angle between two unit vectors. The supported LSH families in Puffinn are CrossPolytopeHash, FHTCrossPolytopeHash and SimHash. SimHash is a one-bit hash function, which creates a random hyperplane at the origin and hashes points depending on which side of the plane the points is located on. CrossPolytopeHash is an implementation of cross-polytope LSH using random rotations. It applies a random rotation to vectors and then maps each vector to the closest axis. And HTCrossPolytopeHash is a more efficient hash function using pseudo-random rotations instead.

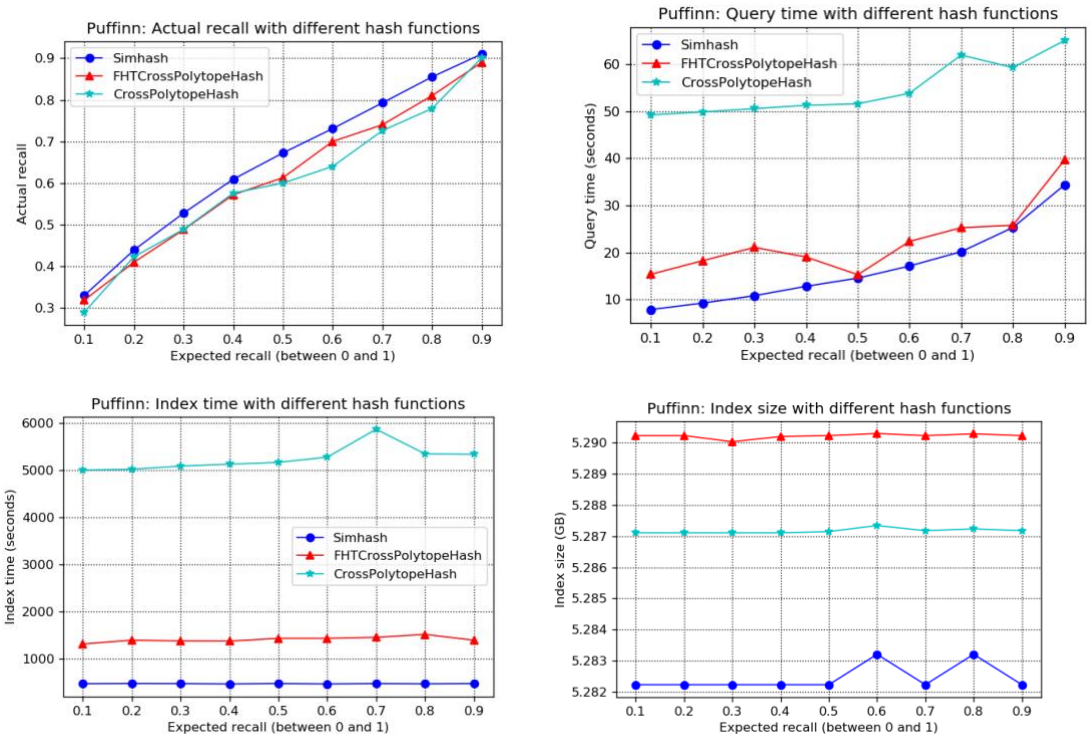
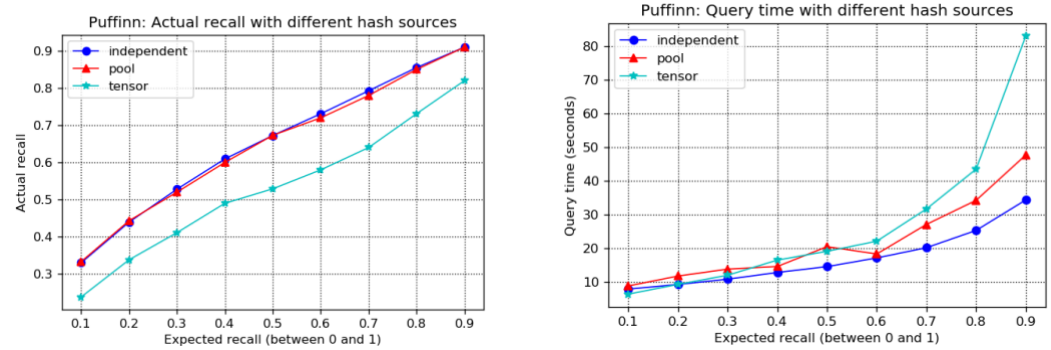


Figure 8: Comparison with different Hash functions with Glove 50, k=30, space=500MB

5.2.4 Comparison with different Hash sources

The supported hash sources in Puffinn are ‘independent’, ‘pool’ and ‘tensor’. They are used to construct the source from which hashes are drawn. ‘Independent’ is the default and optimal hash sources defined in Puffinn.



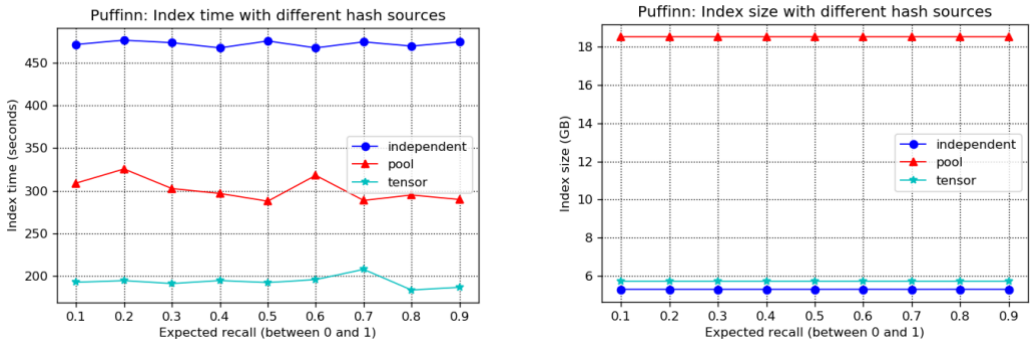


Figure 9: Comparison with different Hash sources with Glove 50, k=30, space=500MB

5.2.5 Comparison with different filters

There are different approaches in the querying process in Puffinn to filter candidates. 'Default' is the most optimized and recommended approach, which stops shortly after the required expected recall has been achieved. 'None' is a simple approach without sketching. It currently looks at every table in the internal structure before checking whether the recall target has been achieved. 'Simple' mirrors 'None', but with filtering. It is only intended to be used to fairly assess the impact of sketching on the result.

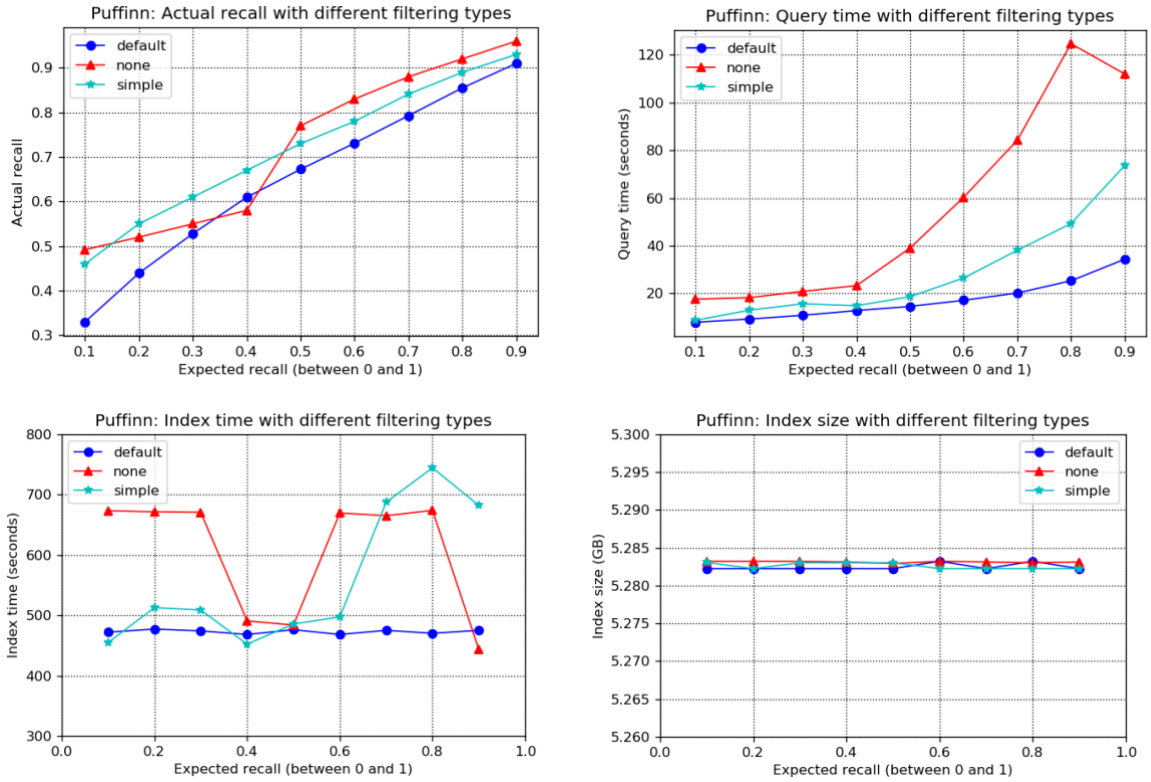


Figure 10: Comparison with different filter types with Glove 50, k=30, space=500MB

5.3 Evaluation of HNSW

The algorithm is evaluated using GloVe dataset which has 50 dimensions. HNSW has two major parameters which affect the balance between quality and cost. We also want to mention that

hnswlib is built with extra qualities such as the ability to run on multi threads, which speed up the whole index process by a constant factor. It also supports incremental index/training process, which means that the index results can be saved and loaded in file format. The two key parameters we tuned are the following:

5.3.1 Comparison of EF_construction

Comparison on ef_construction, which defines a construction time/accuracy trade-off, it is the size of the dynamic list for the nearest neighbors (during construction, another similar parameter ef, which is used during the query), higher ef leads to more accurate but slower construction/search. We do the experiment of combinations of M and ef values, and the effect of ef is shown as below (combined of different Ms):

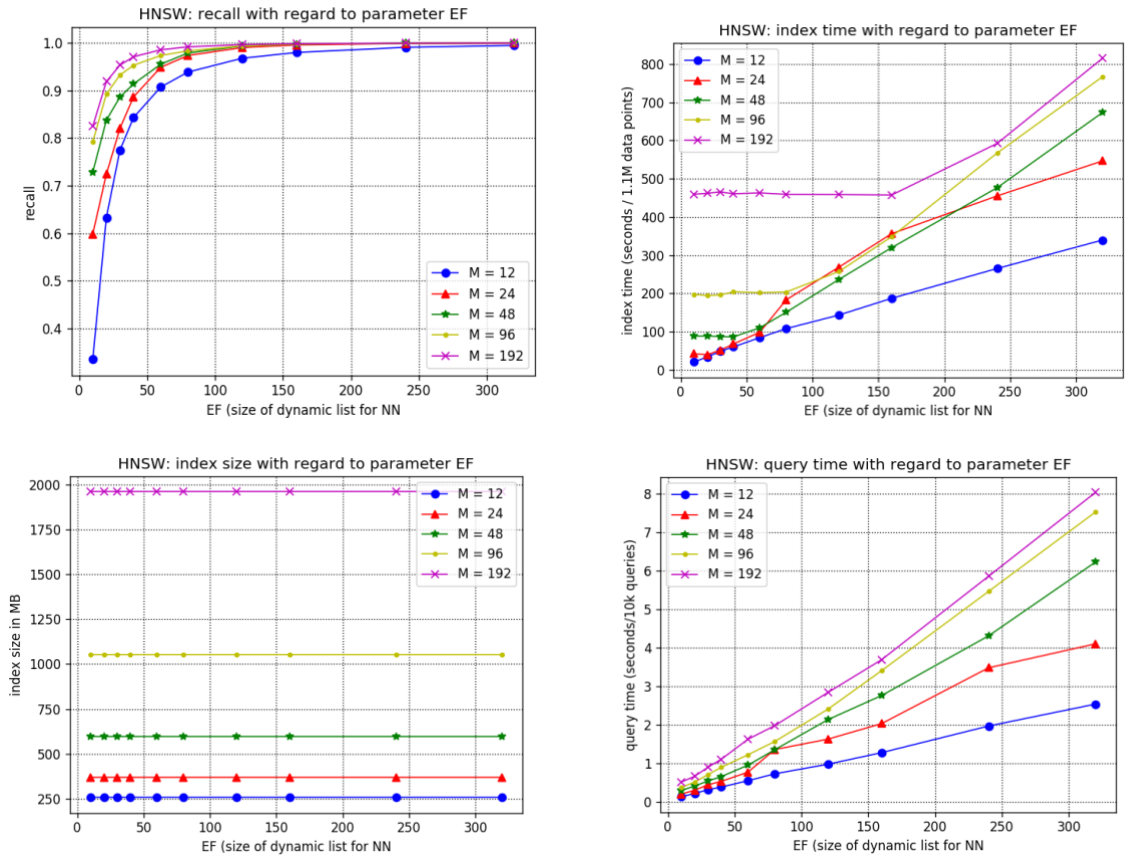


Figure 11: Comparison of EF_construction

5.3.2 Comparison of Maximum Number of outgoing connections

Another one is M, which defines the maximum number of outgoing connections in the graph. Low M works better for datasets with low intrinsic dimensionality and/or low recalls. While higher M are required for optimal performance at high recall as the following evaluation shows (we only show it on EF = 40).

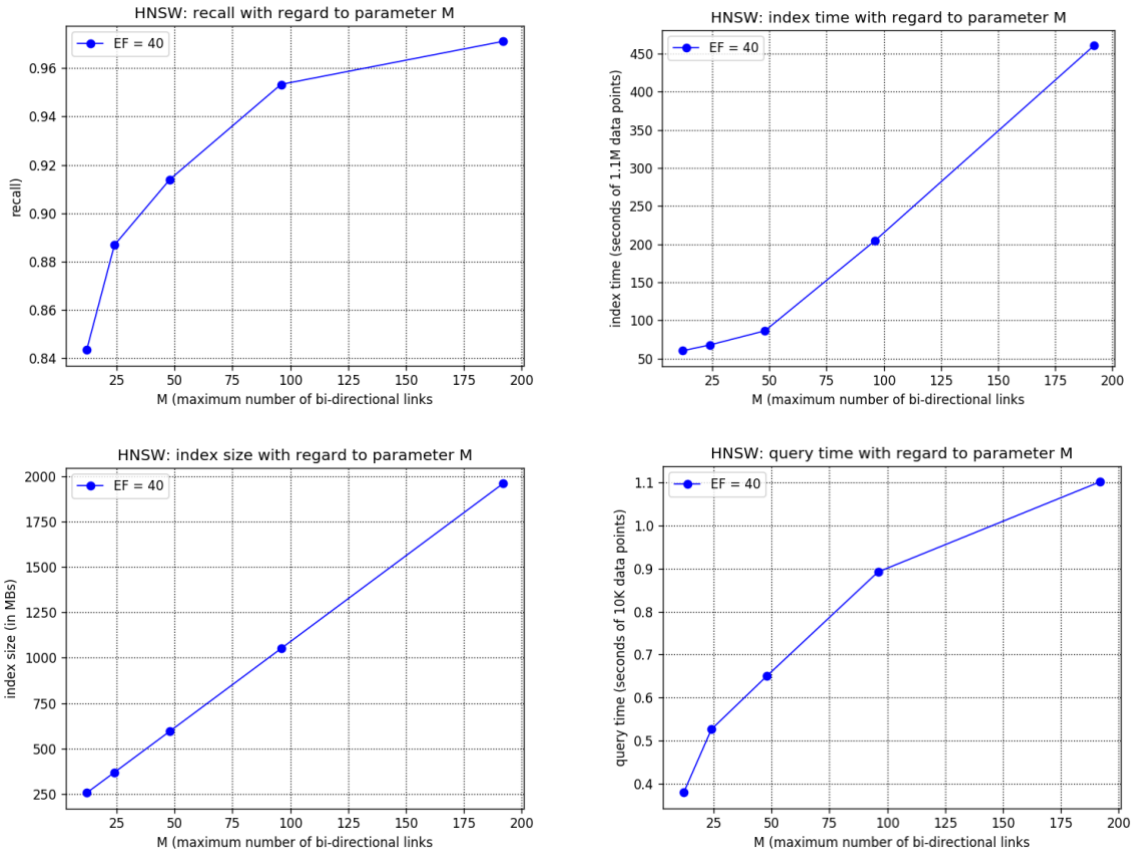


Figure 12: Comparison of the maximum number of outgoing connections

From the above results we can summarize the values of EF and M's effects on our metrics: For recall values, it converges as EF and M increases, which means over a certain threshold, when recall is close to 1 that it has little response to EF and M. For index and query time, it has linear independency on EF and M. The intriguing observation is that for the same dataset, the index size is independent of EF, but has a strict linear dependency on M, which means M determines the algorithms memory consumption.

5.4 Evaluation of Faiss-ivf

To query and data, faiss will first build the model by clustering the dataset into different groups and query the index for searching data.
The algorithm is also evaluated using GloVe dataset for 4 different dimensions, which are 25, 50, 100, 200 dimensions.
We can first see that for all of the different dimensions, if the index time and query time will be different under different number. Since the number of lists we divide them into must be the multiple of the dimensions. Therefore, the common denominator of four of the dimensions are 1, 5, 25. We will set them to be different number of lists and see how it affects the index time, query time, index size for one thread.
To increase the accuracy of the results, we have to increase the number of blocks visits for target, which is indicated by n-probe variable. The default value for n-probe is 1. We can make comparisons between the accuracy to the value of n-probe, we here set n-probe from 1 to 10.

5.4.1 Comparison of M

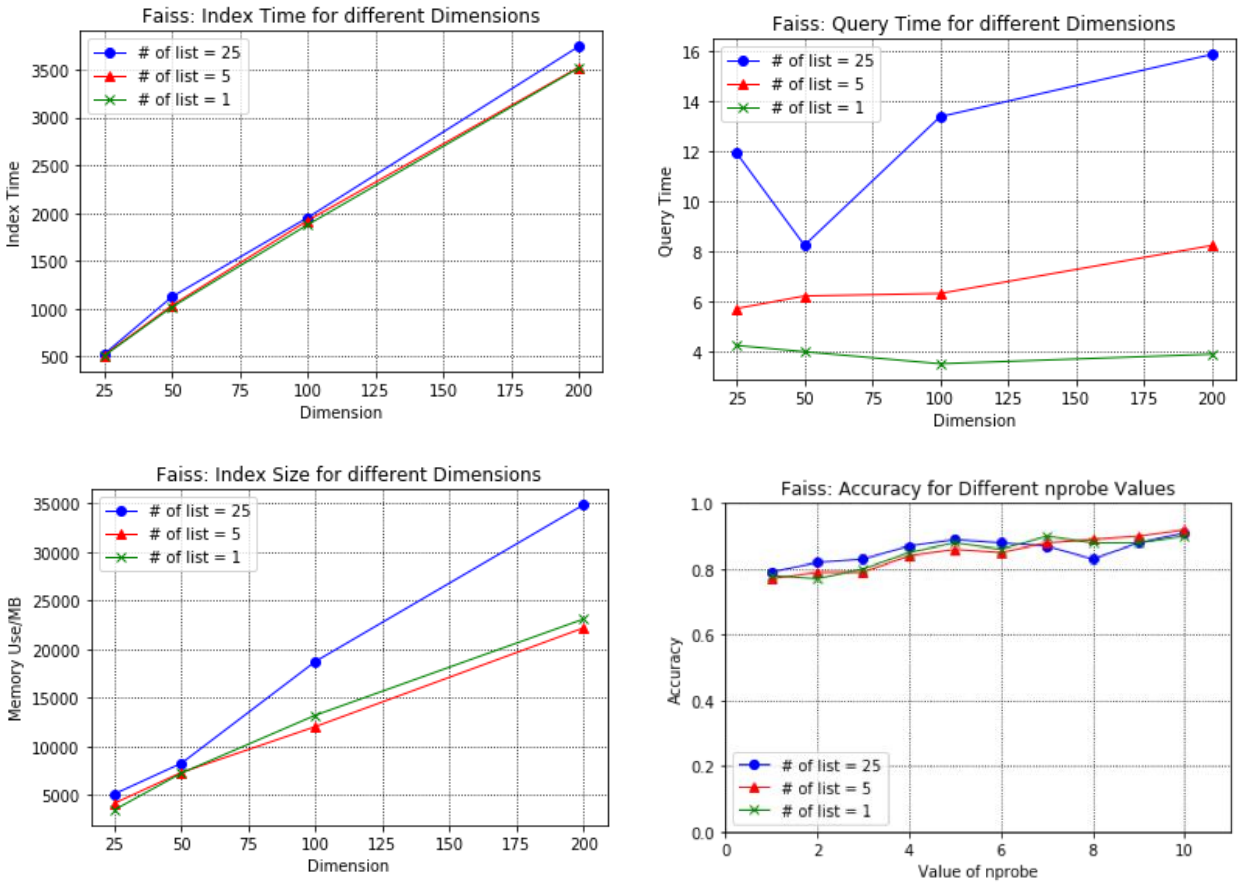


Figure 13: Comparison of M in Faiss-IVF

From the results shown above, we can see that as the increase of the number of lists divided, the query time and the accuracy of the estimation will increase. The query time and accuracy increases can be explained since the more groups we divide, the more time the algorithms will run to put them into different clusters. Since the more cluster we have there is a higher chance that we can put them into the correct groups. On the contrary, the index time and memory uses is not very different, since index time only changed one step, which is divide the lists into more groups. The sudden change in number of lists in 25 can be a outlier, since other constants are pretty stable. Similarly, the index size will not vary very much, even if there is a big jump from number of lists of 5 to 25 at the dimension of 100. In this case, it is possible to claim that the divide groups will increase memory use by very little, there increase in memory use can be used to store the memory address differently as a tag for future to fetch the data.

5.4.2 Comparison of n-probe

In faiss algorithms, there is one variable that we can choose to control the accuracy and make time management, ta tis n-probe, a variable that determines how many groups to visits, therefore, it can be as large as the number of lists. The default value for n-probe is 1, different values of n-probe will change the run time and accuracy.

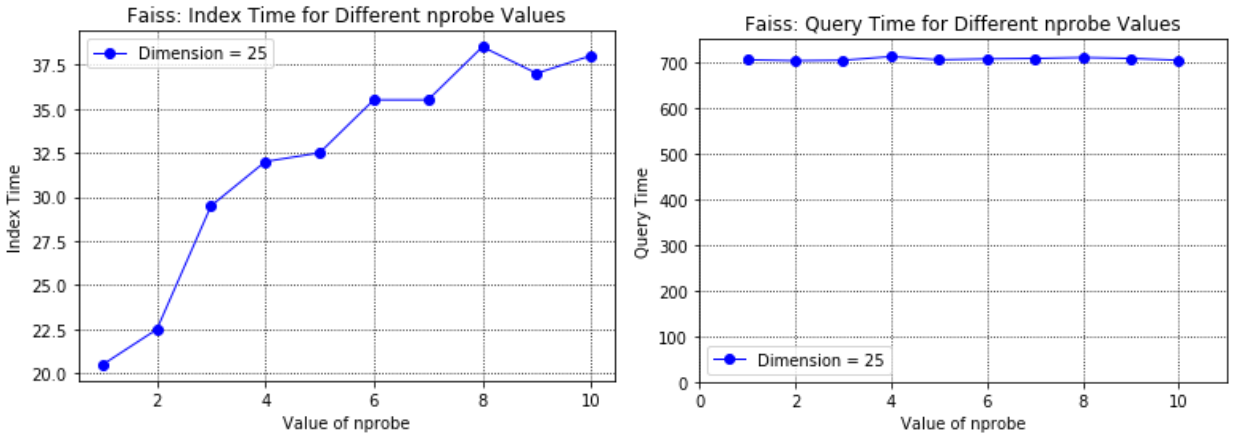


Figure 14: Comparison of n-probe in Faiss-IVF

Based on this issue, we run the result solely on n-probe. From the above result, we can see that the result is of the index time will vary dramatically and increase as the n-probe increase. However, the query time almost remains the same, it is consistent as the algorithms shows, the n-probe will only work at the query process, since it is mainly used to search. The indexing process will not be affected.

6. References

[1] PUFFINN: Parameterless and Universal Fast FInding of Nearest Neighbors, M. Aumüller, T. Christiani, R. Pagh, and M. Vesterli. ESA 2019.

[2] Faiss-ivf: https://github.com/facebookresearch/faiss

[3] HNSW: Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs, Yu. A. Malkov, D. A. Yashunin. arXiv:1603.09320

[4] M. Aumüller, E. Bernhardsson, A. Faithfull: ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. Information Systems 2019. DOI: 10.1016/j.is.2019.02.006