Sheet Hero

CS541 -- Assignment 2

Group 2 | November, 2019

Chaoji Zuo (cz296)
Cong Deng(cd745)
Bowei Feng(bf289)

I. Summary

In this project, we applied 3 methods (FALCONN, Hnswlib, nanopq) from 3 different categories of nearest search algorithm on a huge dataset, and finished a lot of experiments to evaluate the methods as well as figured out which parameter matters in each method.

Through our experiment, we found that the query time and index time always exhibit an inverse relationship with the accuracy in all 3 methods, which means if you want to get a more accurate result, you have to spend more time on the querying.

What's more, for different methods, there always exist some important parameters which would influence the performance a lot.

For FALCONN, it can index very fast with a high accuracy result. In detail, the much more hashing tables and probes, the accuracy would be better. But the increase in the number of hashing functions would decrease accuracy. What's more, the hashing probes have a very small influence on index time.

For Hnswlib, it has the shortest query time with a not bad accuracy. Specifically, with the increment of the ef and ef_construction, the precision could improve apparently but there is another problem that the query time would increase a lot, and the index time is mainly relevant to the value of ef_construction.

For nanopq, it is really slow since it needs to do the clustering before encoding. When the number of sub-space increases, the accuracy would be improved apparently along with the increment of the training time of K-means. However, the size of training data seems to just influence the clustering time and have minimal effect on accuracy and query time. Also, we got a better accuracy when we set Ks to 512, we thought the reason is our data's dimension is really high so we need a longer codewords to do the clustering and encoding.

Another important insight is that data normalization is really important in such Cosine similarity nearest neighbors searching problems. We applied L2 normalization on our data to improve overall accuracy and query time remarkably.

II. Goals

We have 3 goals in this project:

1) Apply 3 nearest search methods on our dataset successfully.
2) Find what parameters would influence the final performance and how they influence.
3) Compare 3 different algorithms to see which one is the best for our dataset.
4) Analyse the experiment result and find interesting insights.

III. Brief description of compared methods

In this assignment, we are asked to use 3 methods from 3 categories, they are quantization-based method, locality sensitive hashing method and proximity-graph based method. Those methods are both focus on nearest searching problem, which is a common and critical problem in the real world. Besides, those methods have different core ideas to address achieve above problem.

Quantization-based method:

> Quantization-based method has a PQ class which is instantiated with the number of sub-vector (M) and the number of codeword for each subspace. Then, we need to train this quantizer by running k-means clustering for each subspace of the training vectors. Given this quantizer, database vectors can be encoded to PQ-codes. For the querying phase, the distance between the query and the database PQ-codes can be computed efficiently.

Locality sensitive hashing method:

> LSH method would use hashing mapping to project the original data points to some new data spaces. The mapped data would be stored in different hashing tables that the nearer points would be stored in the nearer tables. Hence, we can use these tables to find the nearest neighbors.

Proximity-graph based method:

> HNSW is a proximity graph based method which based on navigable small world graphs with controllable hierarchy. Hierarchical NSW incrementally builds a multi-layer structure consisting from hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. The maximum layer in which an element is present is selected randomly with an exponentially decaying probability distribution. This allows producing graphs similar to the previously studied NSW structures while additionally having the links separated by their characteristic distance scales. Starting search from the upper layer together with utilizing the scale separation boosts the performance compared to NSW and allows a logarithmic complexity scaling.

IV. Experiment Detail

Dataset: P53 Mutants Dataset

This dataset contains 31059 instances of 5408-dimensional biological data about mutant p53 proteins. This dataset can be regarded as a high-dimensional datasets.

In our experiment, we split our whole dataset into two parts: training set and test set. Training set would be used to train set up the index or tables. Data points in test set would be the query points. We used Cosine similarity to find the nearest neighbors.

Algorithm/machine:

Proximity-graph based: Hnswilib Project link.

HNSW is an efficient cross-platform similarity search library and a toolkit for evaluation of similarity search methods. The core-library does not have any third-party dependencies.

The goal of the project is to create an effective and comprehensive toolkit for searching in generic and non-metric spaces. Even though the library contains a variety of metric-space access methods, our main focus is on generic and approximate search methods, in particular, on methods for non-metric spaces.

In our experiment, we use hnswlib python package and run it under python(3.7.4) on maxOS Mojave(10.14.6) with Intel Core i5(2.7 GHz)

Locality sensitive hashing: FALCONN Project link.

FALCONN is written in C++ and consists of several modular core classes with a convenient wrapper around them. Many mathematical operations in FALCONN are vectorized through the Eigen and FFHT libraries. The core classes of FALCONN rely on templates in order to avoid runtime overhead.

In our experiment, we use FALCONN's python package and run it under python(3.7.4) and Clang(4.0.1) on maxOS Mojave(10.14.6) with Intel Core i5(2.3 GHz).

To apply FALCONN on our dataset, we normalized and centralized our data to improve the running performance.

Last but not least, FALCONN requires us to use single precision float data to run.

Quantization-based: Nano Product Quantization(nanopq) Project link.

Product quantization (PQ) is one of the most widely used algorithms for memory-efficient approximated nearest neighbor search, especially in the field of computer vision. Nano Product Quantization(nanopq) contains a vanilla implementation of PQ and its improved version, Optimized Product Quantization (OPQ).

The basic idea of PQ is to split an input D-dim vector into M-dim sub-vectors. Each sub-vector is then quantized into an identifier of the nearest codeword.

First of all, a PQ class is instantiated with the number of sub-vector (M) and the number of codeword for each subspace. Next, you need to train this quantizer by running k-means clustering for each subspace of the training vectors. Given this quantizer, database vectors can be encoded to PQ-codes. For the querying phase, the asymmetric distance between the query and the database PQ-codes can be computed efficiently.

In our experiment, we use nanopq's python package and run it under python(3.7.4) and Clang(4.0.1) on maxOS Mojave(10.14.6) with Intel Core i5(2.3 GHz).

V. Evaluation

In this part, we tried to do both horizontal comparison and vertical comparison. Firstly, we tried to find the best parameters inside each algorithms. Then we compared the performance between different algorithms using the best parameters. To avoid the influences from different running machines. We used several machines to run the same algorithm to get the average result. We tried different size of data. We first tried the algorithms on a sample dataset, then apply it on a larger dataset.

Moreover, we split our whole dataset into two parts: training set and test set. Training set would be used to train set up the index or tables. Data points in test set would be the query points.

Besides, we used Cosine similarity to define the distance and use linear scan to calculate the ground truth. Cause our features are scaled in a [0,10000] range, we normalized use L2-norm before processing our data.

We also tried to use the Euclidean distance to search, the accuracy is really high in FALCONN, cause it used Euclidean distance to calculate the distance after hashing. However, it would take an average 10 times longer to do the index and query, i.e. index time: 2 s; per query time: 0.04 s. Considering we need to do a huge amount of test in this project, so we finally decided to select Cosine similarity to save more time in waiting.

To evaluate the scalability, we try different dataset sizes on each methods. We tried 30000 points, 10000 points and 3000 points (include training set and test set). And we compared the query time in different sizes' data.

To evaluate the accuracy, we use the recall rate (TP/(TP+TF)), which can be calculated by right nearest neighbors number divided by required neighbor number in our experiment. In order to get the ground truth, we use linear scan to calculate the cosine similarity and save the result. In each experiment, we would use this result to verify our accuracy.

We used 3 important indicators to show the performance: query time, index time and accuracy.

Evaluation details are as follows.

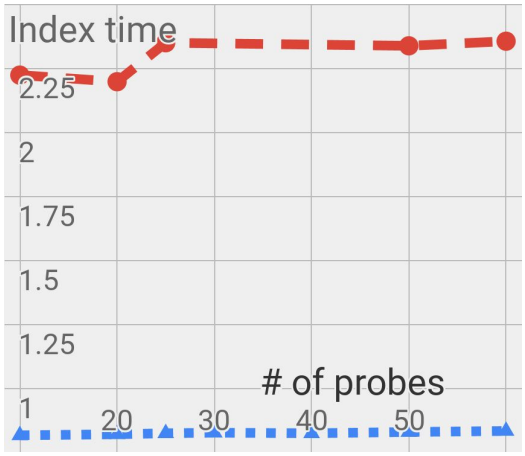
Vertical Comparison:

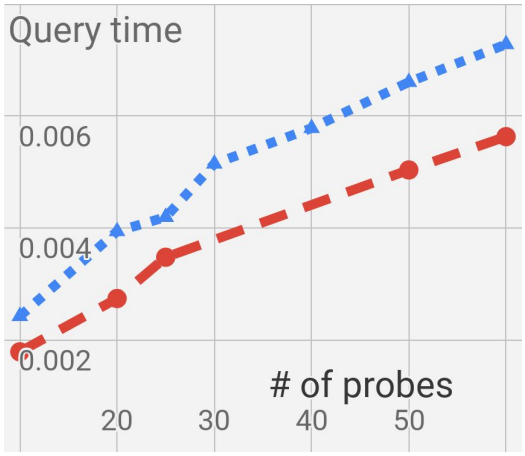
FALCONN:

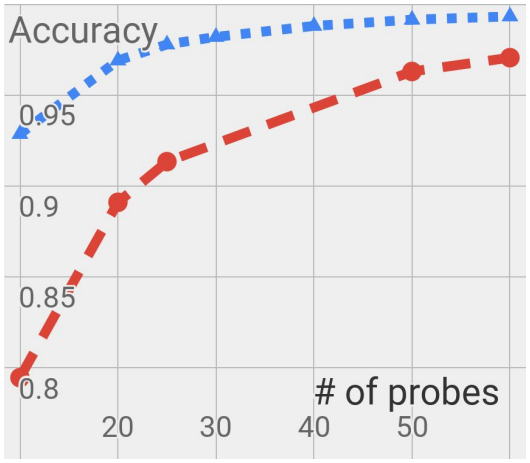
> import factors: the number of hash tables, the number of hash functions, the number of probes(total number of buckets probed across all hash tables), LSH families (FALCONN supports Hyperplane LSH and Cross-polytope LSH).
>
> The default parameters are: 10 hash tables, 10 hash functions, 10 number of probes, Cross-polytope LSH and 5 nearest neighbors each query. To evaluate how those parameters influence the result, we control our variables and change just one parameter in each test. There are also some small parameters like hash table types or how to calculate distance, but we ignore those parameters and leave them as

default value. We use index time, query time and accuracy to evaluate the performance.

Here are the results:



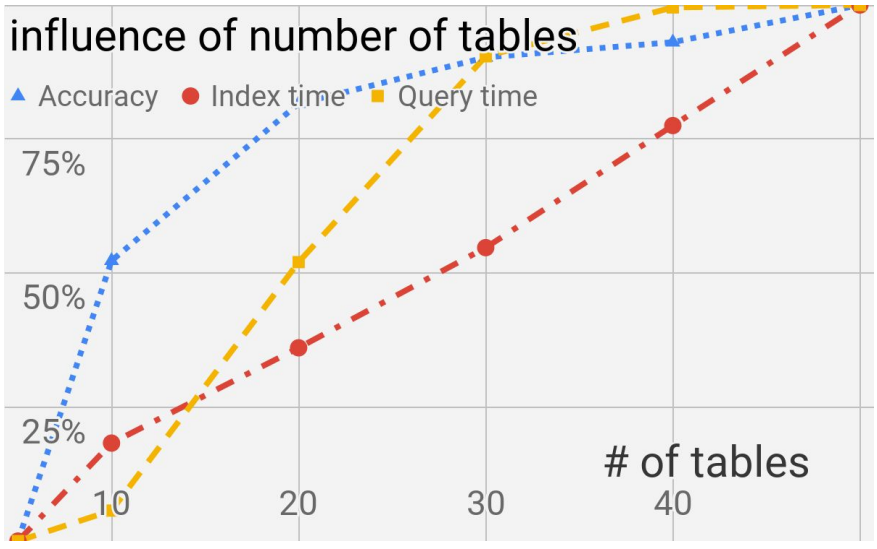




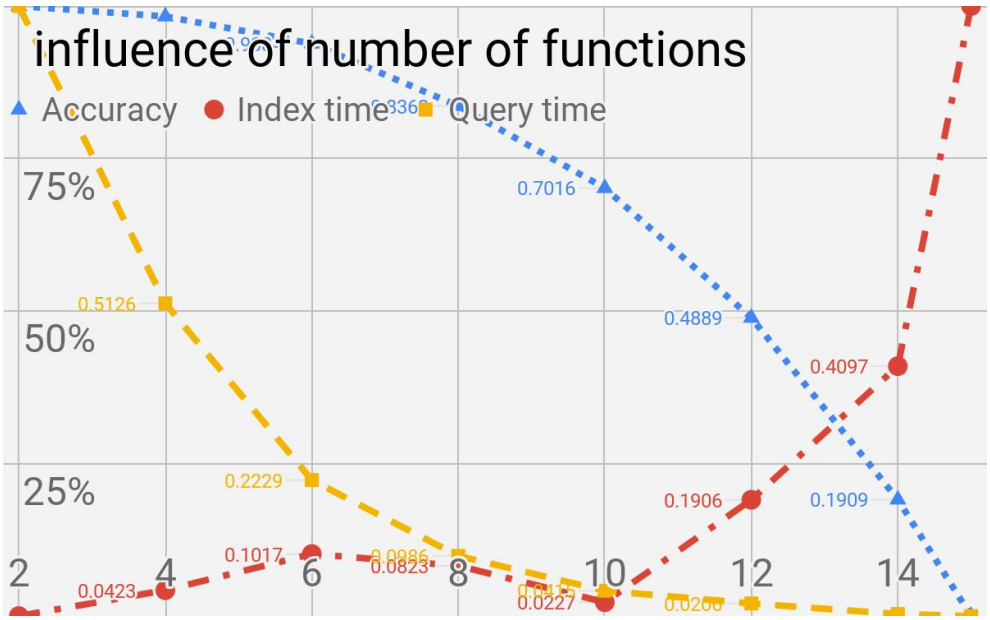
Blue line represents CrossPolytope, red line represent Hyperplane

As you can see, the Hyperplane would has a 2X much more index time but the accuracy not better than Cross-polytope. Hence, CrossPolytope is the better LSH family in our dataset. And the number of probes do influence the performance, when the probes increases, the accuracy and query time would also increase. Besides, the number of probes almost would not influence the index time at all.

Then we evaluate how the number of tables influence the performance. In this case, we care more about the relationship of those indicators rather than the concrete values, so we min-max normalize the numbers and plot the trend lines as below. Since the minimum number of probes is related to the number of tables, we used 50 probes in this experiment. FALCONN doesn't the method to calculate the index size directly, but the number of tables can reflect the index size. The more tables it has, the more index size it would use.

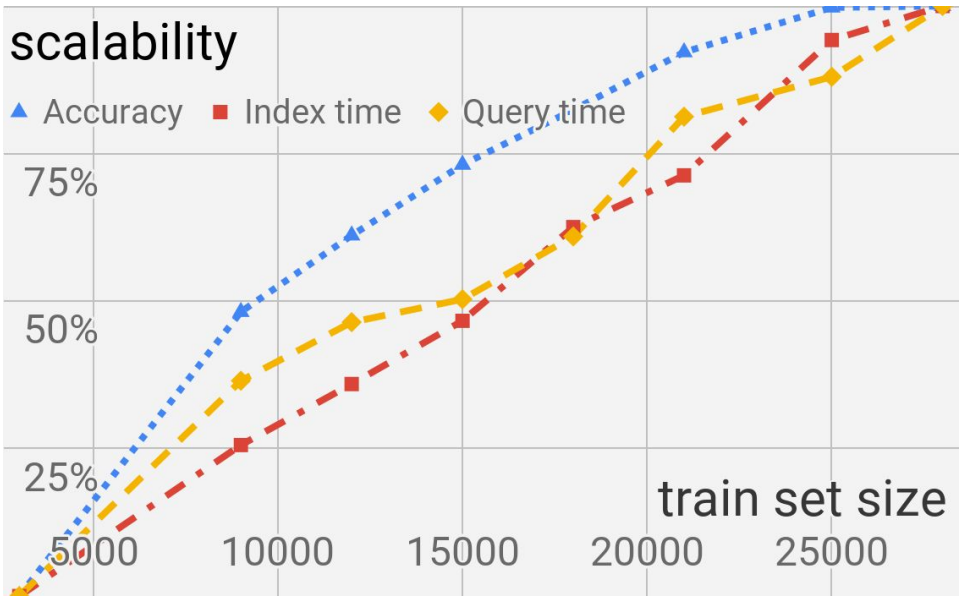


The result shows that when the number of tables increases, the accuracy would increase meanwhile it would cost more time on indexing and querying.



Above figure shows the influence of number of functions. When FALCONN uses less hash functions, the accuracy would be really high but it would cost much more time to query. But the interesting thing is the index time decreases slower than accuracy, so maybe 8 hashing functions is a great trade-off in this method. Another interesting thing is that when the number of functions changes between 2 and 10, the index time would keep in a stable value. But when the number of functions larger than 10, the index time would have a rapid growth.

What's more, we tested the scalability by fixed the test set size as 2800 and change the train set size from 300 to 2800.



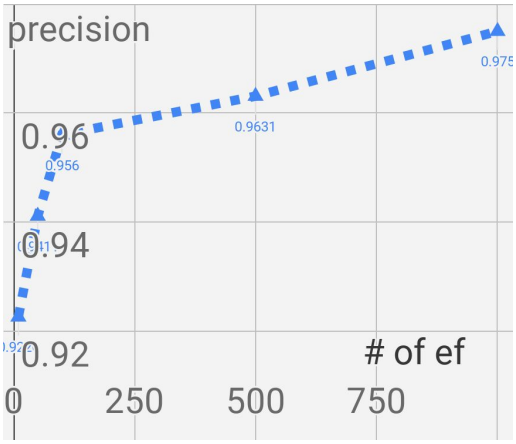
From the above figure, we can see that the scalability really would influence the performance. It's not surprising that the scalability would influence the index time and query time, but it's strange that it would influence the accuracy. Through our analysis, we believed the scalability would influence the accuracy by influencing other parameters, i.e. our default number of tables may works bad on a small dataset. Hence, we can get a conclusion that those parameters work relatively.

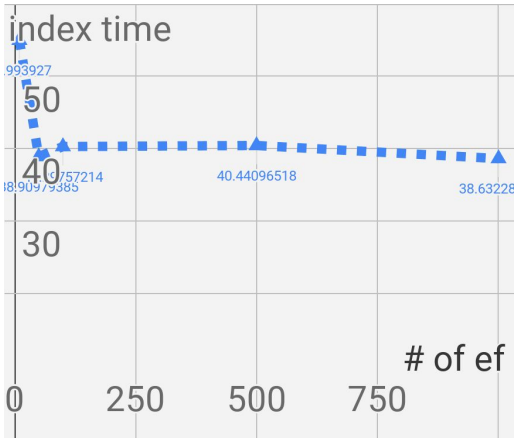
Last but not least, we found that the number of neighbors of each query would not influence the performance significantly, no matter the accuracy or the per query time.

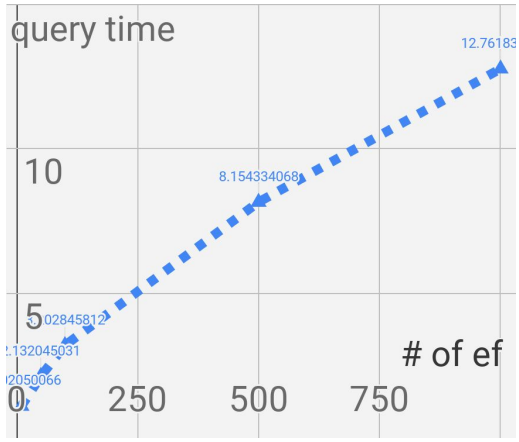
Hnswilib:

import factors:

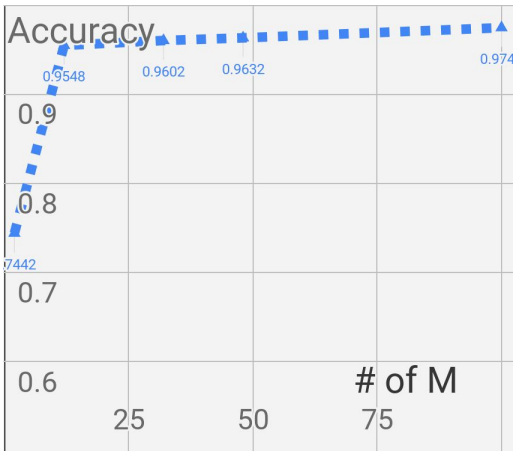
- ef - the size of the dynamic list for the nearest neighbors (used during the search)
- M - the number of bi-directional links created for every new element during construction.
- ef_construction - the parameter has the same meaning as ef, but controls the index_time/index_accuracy.

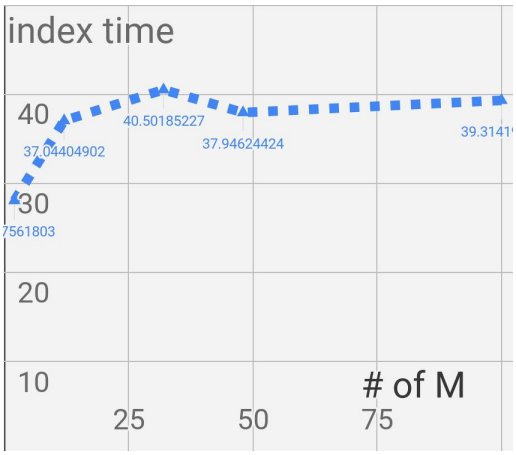


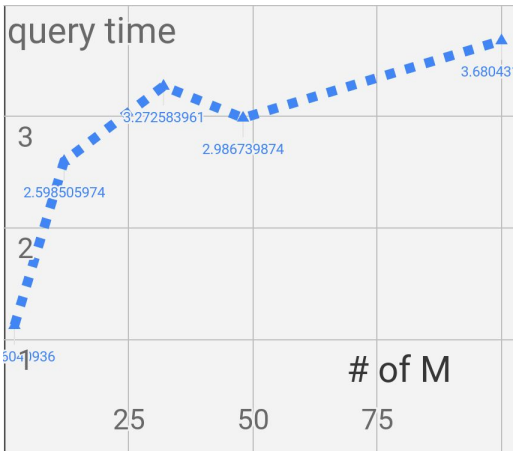




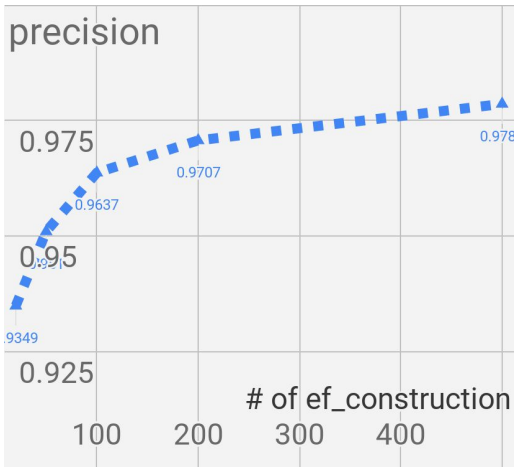
As 3 figures shown above, with the increment of the attributes ef, the precision increased, but it seems that the precision has a limit, the index time seems irrelevant with the value of ef, and the query time increases apparently with the increment of the ef.

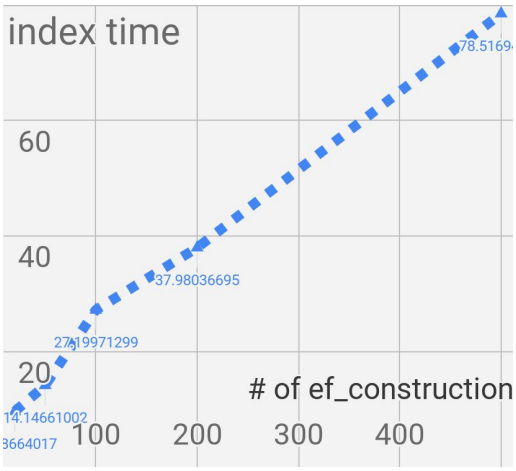


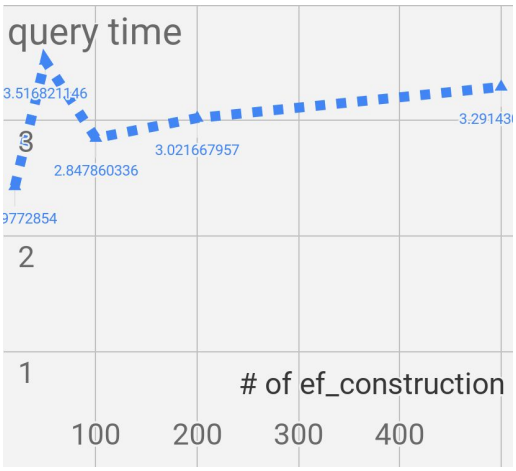




As 3 figures shown above, reasonable range for M is 2-100, we set M from 2, 12, 32, 48 and 10. With the increment of the M, the precision increases but it has a limit, the index time also increases apparently but it seems having a limit, the query time also increases and trends to solid.



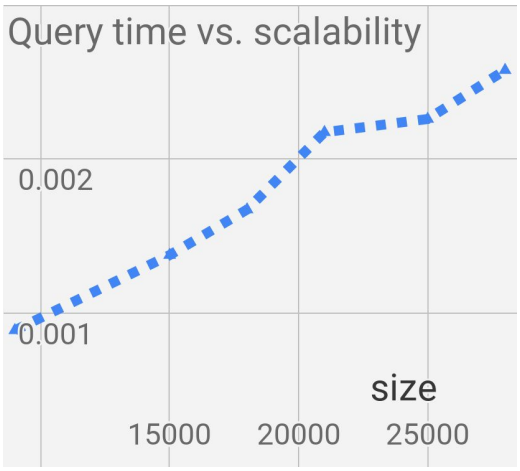


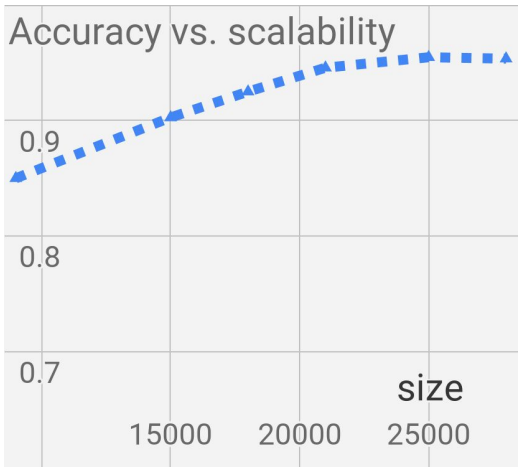


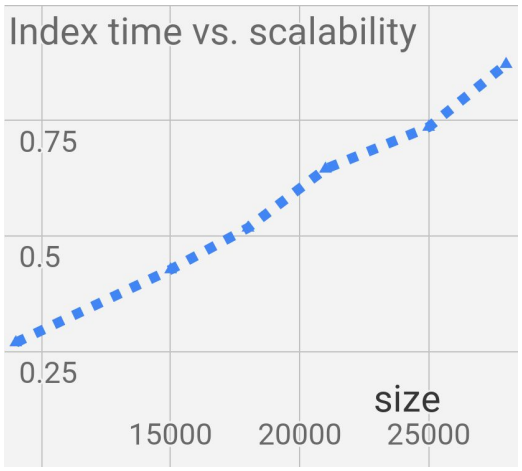
As 3 figures shown above, with the increment of the ef_construction, the precision increases apparently, but the index time also increases which means the algorithm needs more time to construction, but improves the accuracy, and the query time seems irrelevant with the value of M.

And from the experiment, when we set the ef and ef_construction both to 1000, the precision could approach 0.99, which seems no error, but the index and query time will increase apparently.

Moreover, the scalability really would influence the performance. When we set the dataset to 3000 and the query to 300, even though we use some low level attributes which not performs well on large dataset, the accuracy is also very high.

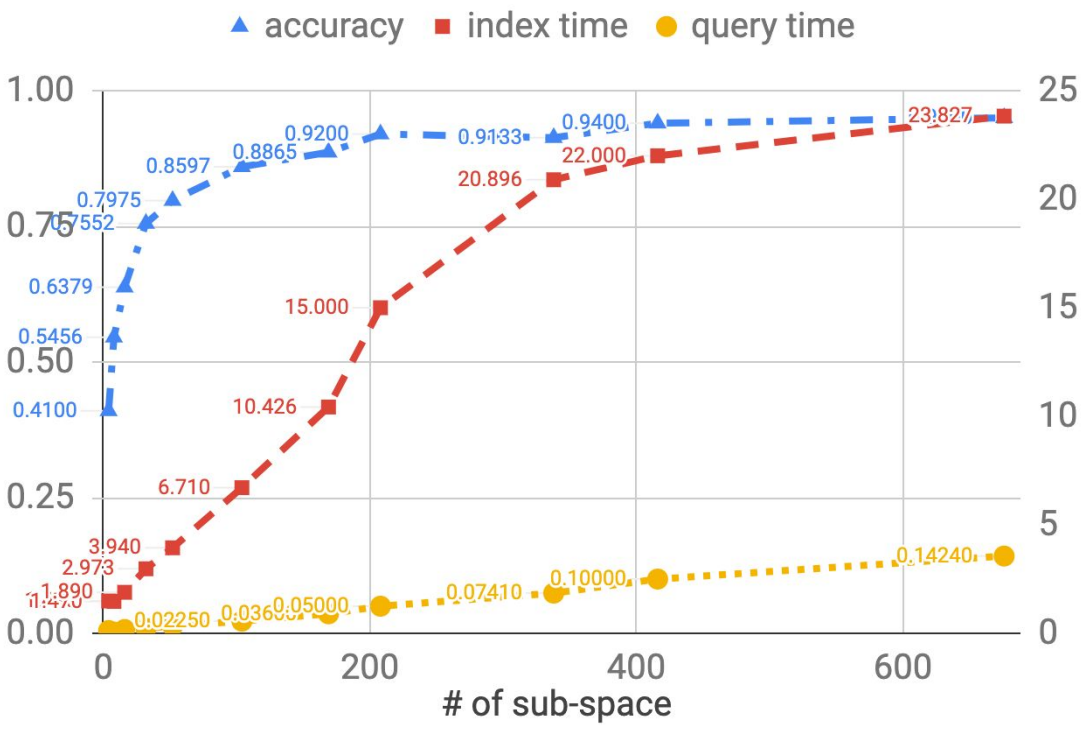






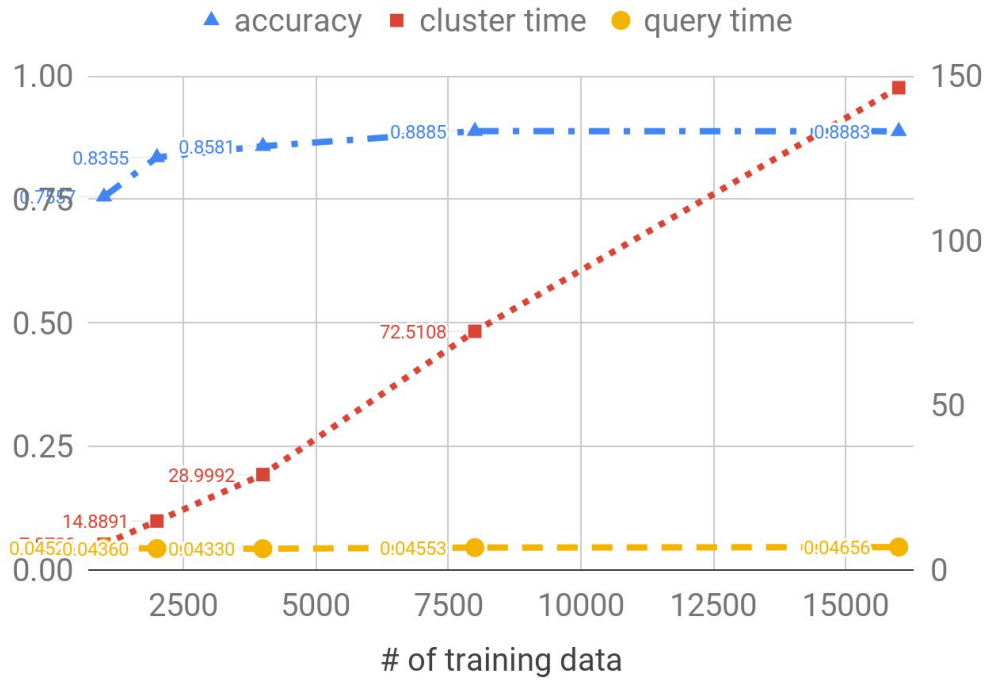
Nanopq:

A PQ method is with the number of sub-vector (M) and the number of codeword for each subspace. Next, we need to train this quantizer by running k-means clustering for each subspace of the training vectors. Number of sub-space(M) is a parameter to control the trade off of accuracy and memory-cost. If we set larger M , we can achieve better quantization (i.e., less reconstruction error) with more memory usage. So we test the impact of the number of sub-vectors on precision, index time and query time.



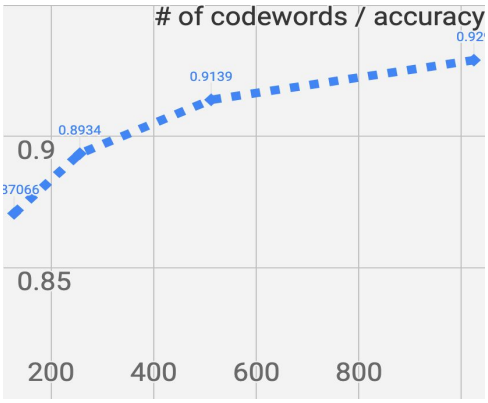
As we can see from the figure, when we apply larger number of sub-space, the index time and query time will grow linearly. This is because we need more calculation to determine to assign the node to which sub-space and match the query vector to the closest sub-space. The variations in the average time to query are acceptable. It also proves that sub-space(M) is a parameter to control the trade off of accuracy and memory-cost.

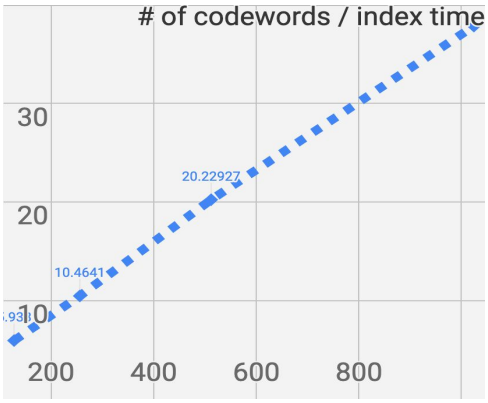
Next, we set the number of sub-space to 208 to test the algorithm using different sizes of training data. The result is shown as below:

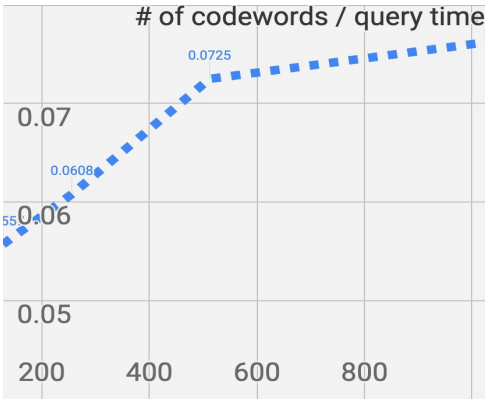


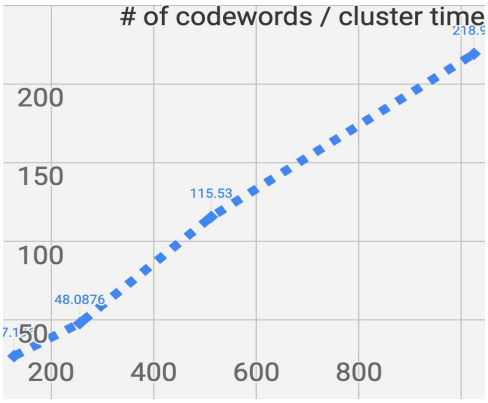
It shows that the size of the training data only has a small effect on the accuracy and query time but will make the training time longer which is understandable. However, it would increase the clustering time in a linear relationship which is a reasonable result.

What's more number of codewords for each subspace is also important. This parameter would influence the index time and accuracy a lot. Intuitively, it would increase the index time and cluster time linearly. It would increase the cluster time by slower each iteration. And we found that it's a good tradeoff to set the number of codewords nearby 400.



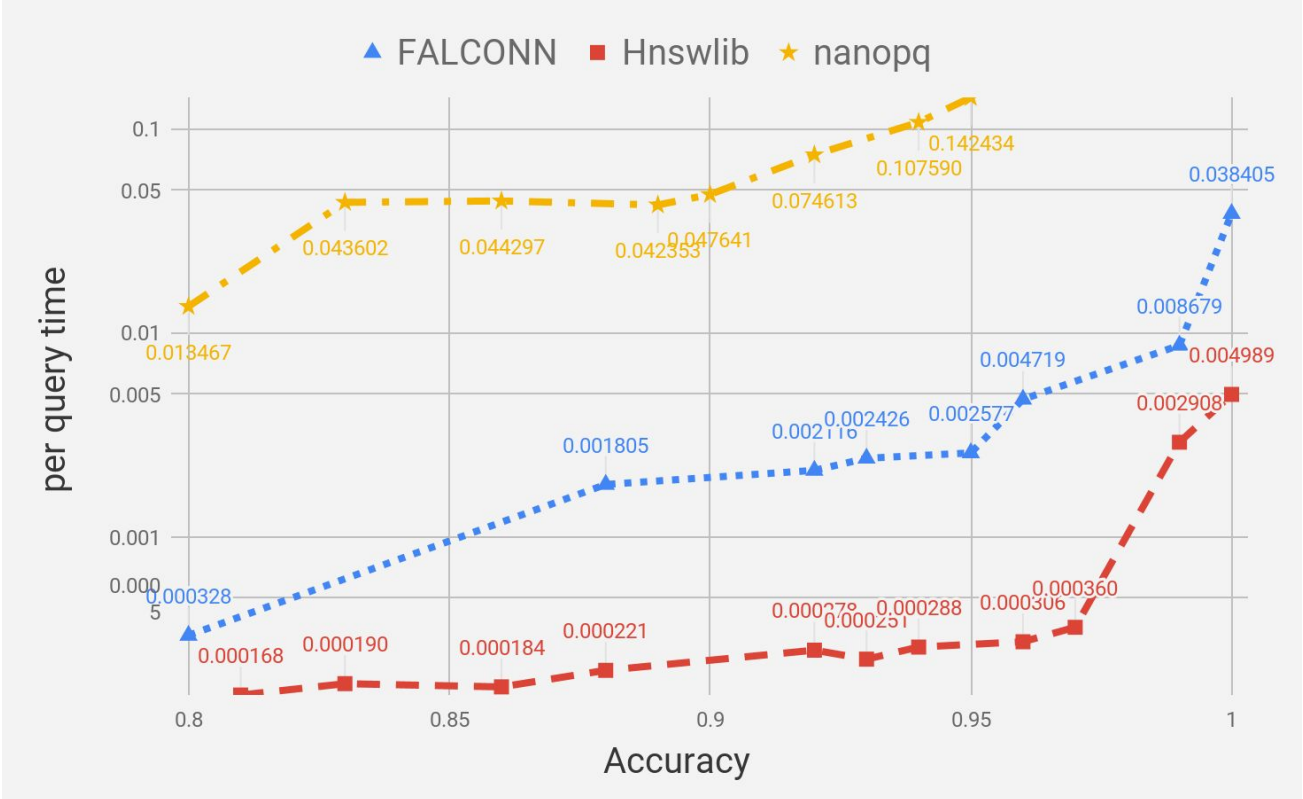






Horizontal Comparison:

To compare the performance of different algorithms, we learned from Erikbern's work and plot the accuracy vs. query time. We run 3000 test set and 28000 training set on all 3 methods multiple times using different parameters to get different accuracy and query time.



As you can see, our three methods have some obvious differences. Hnswlib can query faster than the other two methods. However, when it comes to high accuracy case, it would take a long time in indexing. Meanwhile, FALCONN would have a much better performance on index time.

Accuracy	FALCONN index time	Hnswlib index time
0.98	0.80	79
0.99	0.97	147

If you want accuracy larger than 90%, FALCONN need at least 0.8 second to index, Hnswilib need at least 40 second to index, nanopq need at least 20 second to index.

As for nanopq, it has the worst performance and it could not get very high accuracy in our test. We thought maybe because this method is written in pure Python while the other two are implemented by C++ and support python interfaces. nanopq's index time is shorter than Hnswlib's, but nanopq need a much longer clustering time since it needs to do the cluster before encoding.

To sum up, these three methods are very different, if you want to query faster, you should choose Hnswlib; if you want to get a high accuracy and fast indexing method, you should choose FALCONN; if you want to use a pure Python method, you can choose nanopq.

VI. Supplementary

Before this experiment, we thought it is not that hard to run 3 algorithms and show the results. But we sadly found it is really hard to set up the experiment and decide the flow to test different parameters. Especially when there are a lot of important parameters would influence the final performance.

The main problems we had faced are as follows:

1) How to split the data and evaluate the scalability? How many for train and how many for test?
2) How to evaluate the parameters in each method? How to set the default value? What if the parameters are not independent and they would influence each other?
3) How to compare different methods together, since the parameters are different and we need to choose the parameters for each method. What's more, each of our members try one method on his own machine, and the performance of each machine is different.

And here are the corresponding solutions we camp up with:

1) We decide to scale the train set and test set together. We kept the size of test set as 10 percent of the size of the train set, i.e. 3000 train set would have 300 test set.

2) We control the variables to evaluate different parameters. Each time we just modify one parameters and see how it influences the query time. index time and accuracy. And for the default value, we just arbitrary choose some numbers because we are focusing on the relationship and influence rather than finding the best parameters.
3) To address the horizontal comparison problem, we try all 3 methods on one machine to avoid the hardware influence. Besides, we run a lot of tests by modifying important factors of each method. Then we reviewed all the results to find the faster one in different ranges of accuracy. In this way, we can compare the accuracy vs. query time using the best combination from different methods. Our approach is inspired by Erikbern's work.

We learn a lot from this experiment project, applying the algorithms we have learned from classes give us have a clearer view of how those algorithms run and what are the differences.

VII. Appendix

Our detailed experiment results can be found in this google sheet:
https://docs.google.com/spreadsheets/d/1XEIp2EnEMg1pz-OVq5iLB2tujp1Nb5vVS9Sr9RBAAmE/edit?usp=sharing

VIII. Reference

https://github.com/erikbern/ann-benchmarks

https://github.com/nmslib/hnswlib

https://github.com/matsui528/nanopq

https://github.com/falconn-lib/falconn

http://archive.ics.uci.edu/ml/datasets/p53+Mutants