COMS W4995 Final Project

Yuanchu Dang
yd2466@columbia.edu

Yunfeng Guan
yg2757@columbia.edu

Shaoyu Liu
sl4640@columbia.edu

Chengrui Zhou
cz2664@columbia.edu

April 30, 2021

Contents

1 Introduction

1.1 Abstract

For the final, we chose an implementation-based project, and applied Nearest Neighbor Search and Similarity Search algorithms to a real-world situation. More specifically, we applied the search algorithms to a dataset of documents to create clusters of similar documents and compared the results of different approaches. We picked four methods: vanilla Locality-Sensitive Hashing method (LSH) via the FALCONN package, Neural LSH, Graph-based partitioning and Hierarchical Navigable Small World (NSW). In the following sections, we will discuss in more details the dataset, the algorithms, our implementation of them and comparison of results.

2 Data

We explored B.V n.d., the largest abstract and citation database provided by Elsevier in 2004. In total, it contains over 70 million articles in over 40,000 journals under the four large categories: life sciences, social sciences, physical sciences and health sciences. It covers three types of sources: book series, journals, and trade journals. All journals covered in the Scopus database are reviewed for sufficiently high quality each year according to four types of numerical quality measure for each title; those are h-Index, CiteScore, SJR (SCImago Journal Rank) and SNIP (Source Normalized Impact per Paper). Scopus also offers author profiles which cover affiliations, number of publications and their bibliographic data, references, and details on the number of citations each published document has received. It provides citation data for all 25,000+ active titles such as journals, conference proceedings and books in Scopus and provides an alternative to the impact factor.

Our reason to choose Scopus as our dataset was because it contains a large amount of detailed journal data, which can be quite conveniently converted to large dimentional points.

The scraped corpus is already nicely captured in a PostgreSQL database, which has several advantages: it provides support for JSON and allows for linking with other data stores such as NoSQL which acts as a federated hub for polyglot databases; it offers a rather sophisticated locking mechanism; it is also compatible with various platforms using all major languages and middleware.

2.1 Overview of Plan

The interface of our algorithm was to output N most similar articles on an input article, based on the similarity of their abstracts. In order to implement the algorithms, we must do some data preprocessing.

First, we extracted the abstract data from the academic article database Scopus, and converted them to sentence embeddings using pre-trained models implemented by (Reimers and Gurevych 2019). Next, we ran the various search algorithms discussed in the next section over those embeddings, which included vanilla LSH via FALCONN, Neural LSH, Graph-based partitioning and Hierarchical NSW. Finally, we collected the results and made a comparison.

3 Algorithms

3.1 Feature Extraction

There are many ways to reduce the dimensionality of text. Some of the common methods include keyword search, and sentence similarity based on Jaccard or cosine similarity (with or without tf-idf weighting). Alternatively, researchers can leverage unsupervised Latent Dirichlet Allocation by Blei, Ng, and Jordan 2003 to documents and computing the KL-divergence of their topic similarity, in order to better capture semantic similarity between documents. Since our initial proposal, we decided to simplify the feature engineering part of our project and solely focus on building and experimenting with the core algorithms themselves. Therefore, we chose pre-trained sentence embeddings. Specifically, we used sentence embedding model from Sentence-BERT (SBERT) [1] pre-trained on large scale paraphrase data. In particular, we used paraphrase-distilroberta-base-v1, which was recommended by the authors for semantic textual similarity tasks. This essentially allowed us to represent each document abstract by a 768-dimensional vector, and allowed us to compute cosine similarity between document abstracts.

[1]https://www.sbert.net

3.2 Nearest Neighbor Search

3.2.1 Vanilla LSH

Vanilla LSH simply ties back to what we covered in the lecture. There are two steps to this procedure essentially. First, as a pre-processing step, each data point is mapped to a countable set by a hash function and the whole mapping is stored in a table in the memory. Second, during the query time, we simply look up the entry to which the query is mapped and go through its members until we find an eligible "similar" instance. The full algorithm is shown as follows:

Algorithm 1 LSH Algorithm

```
[1] A $(r, cr, P_1, P_2)$-LSH hash function $h$, a dictionary data structure $V$, and query $q$
$x \in D$ Prepossess h(x) and store it in $V$
$p \in V(h(q))$ compute $||p - q||$ and return $p$ if $||p - q|| \leq cr$
```

3.2.2 Neural LSH

Next we focused on experimenting with the Neural Locality-Sensitive Hashing method for balanced partitioning introduced by Dong et al. 2020. This algorithm incorporated clustering and learning in order to find a high-quality space partition (possibly better than the LSH family in Indyk and Motwani 1998 or RP-Tree in Dasgupta and Sinha 2015)

We briefly outlined the algorithms in Dong et al. 2020 here:

Preprocessing

1. Build a k-NN graph G out of dataset P.
2. Run a balanced graph partitioning algorithm (the authors choose KaHIP) on G into m parts. Let $\pi(p) \in \{1, 2, ..., m\}$ be the part containing p, for each $p \in P$.
3. Train a supervised machine learning model (e.g. small neural networks, logistic regression, etc.) with training set P and labels learned from step 2. Then for each $x \in R^d$ we would have a prediction $M(x) \in \{1, 2, ..., m\}$.

Query For query point $q \in R^d$, and the number of bins to search b:

1. Run inference on M to compute $M(q)$.

2. Search for a near neighbor of q in the bin $M(q)$. If M furthermore predicts a distribution over bins, search for a near neighbor in the b top-ranked bins according to the ranking induced by the distribution.

Intuitively, this algorithm has employed modern (supervised) machine learning techniques to find data-dependent, balanced, locality-sensitive space partitioning. This can serve as a higher-quality indexing technique for NNS problems.

In this project we implemented both “small neural network” and “logistic regression” in preprocessing step 3, and experiment with different soft label parameters S. Possible extensions of this model can be the open problems highlighted in the *Future directions* section of the paper as well as in **??**.

3.2.3 Graph-based partitioning

Another possible direction we thought was to use the graph-based algorithm proposed by Y. A. Malkov and Yashunin 2020. This is an approximate K-nearest neighbor search based on navigable small world graphs. The high-level idea is to construct a graph (possibly k-NN), and then for each query perform a walk which eventually converges to the nearest neighbor. This method, on the empirical side, is currently the fastest indexing technique for the NNS problems. Therefore, we also compared it with Dong's algorithm in terms of quality and algorithmic efficiency.

3.2.4 ANN-Navigable Small World Graph

The algorithm Y. A. Malkov and Yashunin 2020 is based on and extended from the approximate nearest neighbor search using NSW graph model in Y. Malkov et al. 2014. To find the nearest neighbors of a search point, the NSW model uses a greedy search algorithm starting from random node and traverses the graph while simultaneously increase the node's degree until the characteristic radius of the node links length reaches the scale of the distance to the query. This could lead to the false local minimum problem. A modification is to search starting from a node with the maximum degree. However, in any case, the NSW algorithm has polylogarithmic complexity scalability of a single greedy search at best, and does not perform well on high dimensional data.

3.2.5 Hierarchical NSW

We first provided a briefly overview of the Hierarchical Navigable Small World model, and highlighted the key architecture. Compared to previous Navigable Small World model for approximate K-nearest neighbor search, the hierarchical NSW can make two major extensions which make performance significantly better in high dimensional data and even highly clustered data.

First, the HNSW model separate the links between nodes according to their length scale into different layers and then search in a multilayer graph starting from the top layer. The top layers have longer links. The algorithm greedily traverses through the elements from the upper layer until a local minimum is reached and then switches to search the lower layer from the local minimum element we found in the previous layer. This allows a logarithmic complexity scaling of routing, which is a large improvement.

Another major modification the HNSW makes is the heuristic that takes into account the distances between the candidate elements to create diverse connection during the element insertion step. The heuristic examines the candidates starting from the nearest (with respect to the inserted element) and creates a connection to a candidate only if it is closer to the base (inserted) element compared to any of the already connected candidates. Intuitively, this allows nodes to maintain longer distance connectivity and is useful for facilitating search on highly clustered data or low dimensional data (see Fig 5, 7 in Y. A. Malkov and Yashunin 2020).

4 Implementation

4.1 Preparation

4.1.1 Datasets

As we've mentioned above, we explored B.V n.d., the largest abstract and citation database provided by Elsevier. After grabbing these files, we stored the data in a CSV file called algo_proj.csv. Next, we converted it to embeddings. In each piece of data, we were primarily interested in its abstract part, based on which we constructed metrics of similarity. The original dataset was too large to be trained in short time, so we took part of it and about 50,000 points as a test set and then we evaluate accuracy.

4.1.2 Word Embedding Generation

We ended up using the Sentence Transformer package [2] by Reimers and Gurevych 2019 to get quick pre-trained word embeddings for our abstracts. In particular, we used paraphrase-distilroberta-base-v1, which is recommended by the authors for semantic textual similarity tasks. This essentially allowed us to represent each document abstracted by a 768-dimensional vector, and allowed us to compute cosine similarity between document abstracts.

4.2 Algorithm implementation

4.2.1 Vanilla LSH

Provided with the feature extraction algorithms above and measures of similarity in vector space, we first experimented with the vanilla LSH algorithm Andoni et al. 2015 over our dataset using the FALCONN library (Fast Lookups of Cosine and Other Nearest Neighbors) [3]. On the high level, FALCONN is a C++ library that implements vanilla LSH algorithms in a very efficient way. It is designed to minimize run-time overhead and also provides a convenient Python wrapper.

Although the FALCONN package is optimized to deal with cosine similarity over dense and sparse datasets, it is also capable of handling the Euclidean distance among other metrics, thus suitable to be used as a baseline to compare against other more advanced methods to be discussed later. There are two primary degrees of freedom we could toggle: one being the number of hash tables to build L, and the other being the number of points to draw for each query N. As we have discussed in the lectures, setting a larger L could improve query time, but inevitably at the cost of longer preprocessing time to build the additional tables as well as larger space consumption required to store them.

[2] https://www.sbert.net/

[3] https://github.com/FALCONN-LIB/FALCONN

4.2.2 Neural LSH

As is stated in 3.2.2, we implemented this algorithm in two stages. In the preprocessing step, a machine learning model trained upon partitioned dataset essentially partitioned the ambient space. Thus based on the partitioning, the algorithm probed its neighbors which were assigned the same label as the query point.

Because of the limited computation capability, the dataset was partitioned via spectral clustering instead of the KaHIP algorithm used in the original paper. Evidently, this replacement could lead to a (typically) unbalanced partition. However, this difference was not significant if queries came in batches. In other words, while the worst-case time for single queries worsened, the expected/average query time remained on the same level.

As for the learning machine, a 4-level shallow neural network was applied to partition the sample space. During the experiments, this was sufficient for learning with 95% accuracy.

4.2.3 Hierarchical NSW

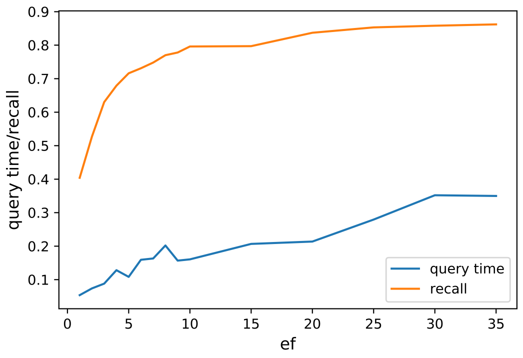
We implemented the Hierarchical NSW method using HNSWLIB library in Python. We implemented the Hierarchical NSW method using HNSWLIB library in Python. We explored several hyper-parameters as discussed in the paper to understand their effect on performance. First, we tuned the parameter M, which controled the number of bi-directional links created for every inserted element during the construction of the NSW graph. Heuristically, larger M worked better on datasets with high intrinsic dimensionality and/or high recall. We used either $M = 48$ or $M = 96$ in our main cases.

We found that finding the nearest neighbor for 1,000 random abstracts from the remaining 49,000 abstracts through linear scan cost 9.68 milisecond per query. We found that increasing ef and $efConstruction$ did improve the recall rate, but they must be quite large for recall rate to be satisfactory. For example, using the default $ef = 10$, $M = 48$ and $efConstruction = 100$ only gave recall rate 39.6% for the nearest neighbor.

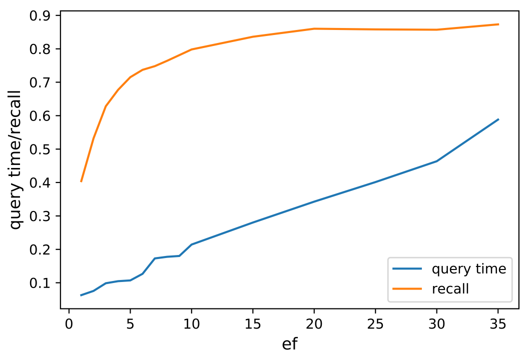
We hence explored two key parameters in the network construction algorithm, ef and $efConstruction$. ef was the size of the dynamic list for the nearest neighbors used during the search. $efConstruction$ controlled for the recall of the greedy search procedure during the second phase of network construction. These two parameters controlled for the quality-speed trade-off of the search.

We plotted the algorithm recall rate for the following choice of parameters $efConstruction = [100, 300, 500]$ and $M = [48, 96]$, $k = 1$ while experimenting with different ef. It was suggested to set M in the range of 48 to 64 for optimal performance at high recall, and larger $efConstruction$ for better accuracy. We found that roughly for $ef = 35$, the recall rate stabilized around 87% and further increase of $efConstruction$ parameters did not lead to search efficiency gain but led to higher construction time and memory consumption. $efConstructure = 300$ had a significant better recall rate compared to $efConstruction = 100$, but further increase of it to 500 did not lead to any efficiency gain in search. To visualize the effect, we plotted graphs for recall error and query time compared with the change of ef below. As we can see from the graph, as we increased the size of the dynamic list for nearest neighbors ef, we improved the quality of our search while also increased query time. The increase of query time tended to be at faster rate than the increase of recall. We did not observe a substantial increase of construction time as we increased the parameter ef.

(a) $efc = 300; M = 48$



(b) $efc = 500; M = 48$



4.2.4 Comparisons

We compare the query time for baseline LSH algorithm from FALCONN with that of the HNSW algorithm at different recall rates. LSH achieved a recall rate of 0.9 with 3 millisecond per query on average, while the best HNSW algorithm (with $M = 48$ and $ef = 80, efConstruction = 300$) only takes 0.83 millisecond per query, which seems to be a significant improvement.

We plot recall error versus query time for HNSW and LSH below, separately for cases when $efConstruction = 300$ and $efConstruction = 500$. We ignore the case for $efConstruction = 100$ because its recall accuracy is too low. As we can see in Figure (2), when the recall rate ranges between 0.4 and 0.9, HNSW is significantly faster than LSH, but such difference becomes smaller in Figure (3), suggesting the importance to optimize meta-parameters ef and $efConstruction$ of the HNSW model. However, the construction time for the LSH algorithm is much faster than HNSW in almost all cases (roughly 8 times faster at 0.9 recall rate). At the right end of HNSW plot, there is some noise as a result of using different ef parameter. The choice of M does not affect our performance much at the given range.

Figure 2: Comparison between LSH and HNSW model with $efc = 300$

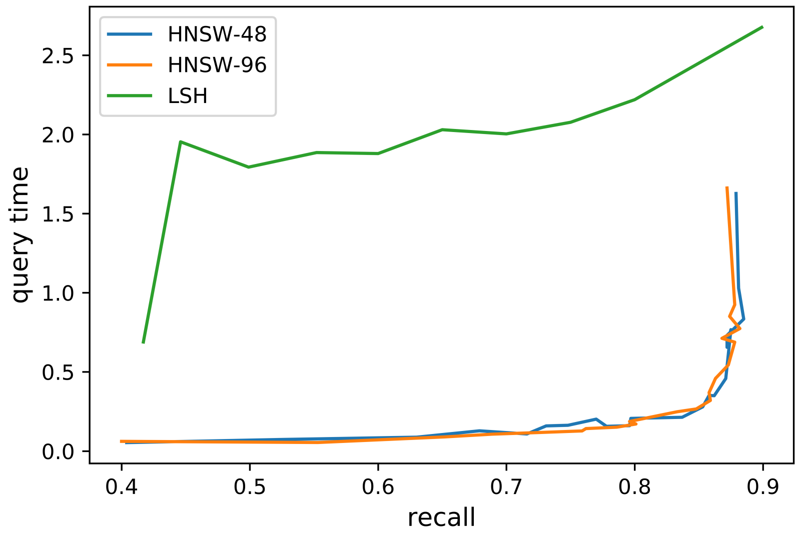
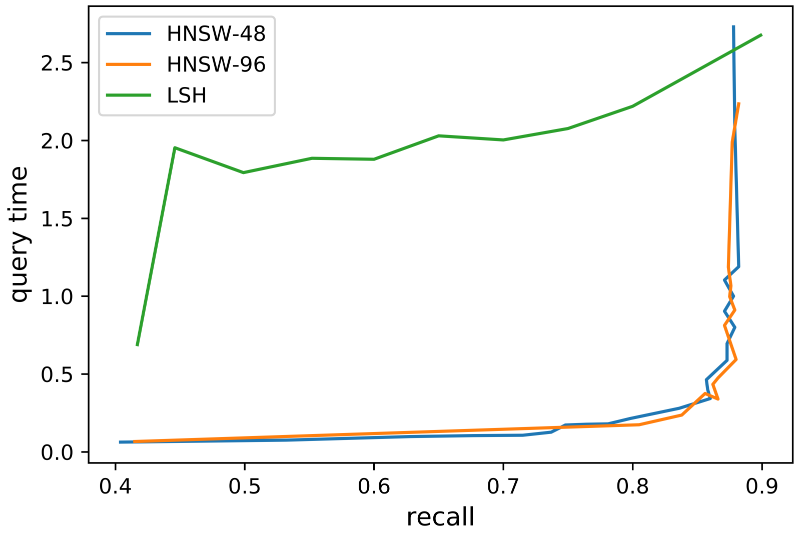


Figure 3: Comparison between LSH and HNSW model with $efc = 500$

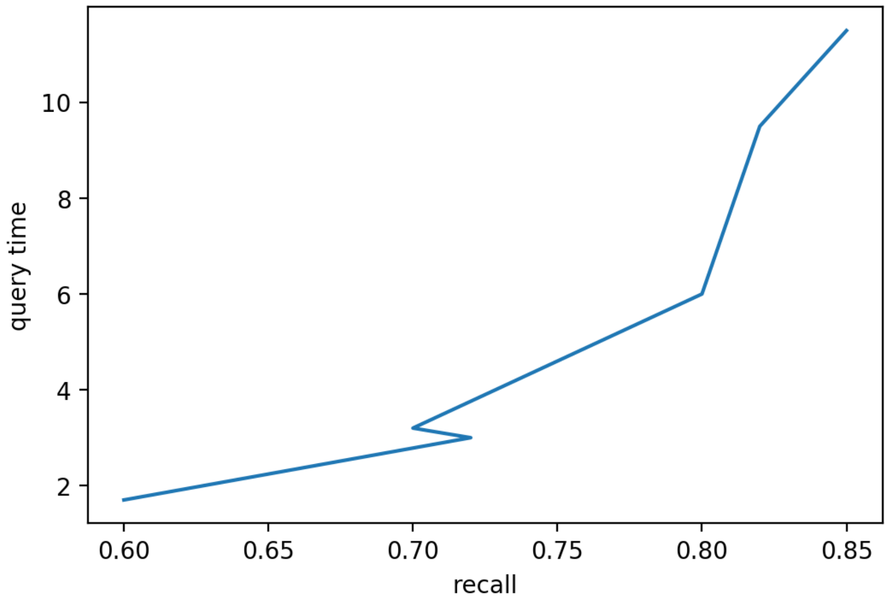


As for the Neural LSH algorithm, the preprocessing time is found unaffordable if the entire dataset is used for experiment. Therefore, we only test this algorithm on a portion of samples and obtain the following results (not comparable with the other two approaches):

On a dataset of size $N = 5000$, prerocessing stage shatters the space based on the labels of data points with accuracy 95%. And the following graph shows the query time and recall rate with bin number $k = 4, 8, 12, 20$.

Similar to the other two approaches, we can find with large bin number, the increase of query time brings less recall rate improvement. As the query time is determined by the size of each bin, this phenomenon is in line with our expectation.

Figure 4: Neural LSH model



5 Conclusion

There are some limitations of our current discussion, which can be pursued in further work. First, we are not able to run and compare the algorithms on larger dataset because of computational complexity. Second, the dimension of our current embeddings is fixed at 768. Exploring the effect of different embeddings dimensions on these different methods can be an interesting direction. Lastly, more fine-tuning of the model meta-parameters for the HNSW model could potentially be helpful in improving its performance, especially if we would like to apply our methods to the actual dataset.

6 Github Link

Finally, we provide a link to our GitHub repository containing our source code for data pre-processing and algorithm implementation: `https://github.com/Yuanchu/adv_algo.git`. A brief introduction to its structure and usage can be found in the README markdown.

References

[And+15] Alexandr Andoni et al. “Practical and Optimal LSH for Angular Distance”. In: *Advances in Neural Information Processing Systems*. Vol. 28. 2015.

[BNJ03] David Blei, Andrew Ng, and Michael Jordan. “Latent Dirichlet Allocation”. In: *Journal of Machine Learning Research* 3 (May 2003), pp. 993–1022. DOI: 10.1162/jmlr.2003.3.4-5.993.

[BV] Elsevier B.V. *Scopus*. https://www.scopus.com. Data by Elsevier B.V.

[Don+20] Yihe Dong et al. “Learning Space Partitions for Nearest Neighbor Search”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rkenmREFDr.

[DS15] Sanjoy Dasgupta and Kaushik Sinha. “Randomized Partition Trees for Nearest Neighbor Search”. In: *Algorithmica* 72.1 (2015), pp. 237–263.

[IM98] Piotr Indyk and Rajeev Motwani. “Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality”. In: STOC ’98. 1998, pp. 604–613.

[Mal+14] Yury Malkov et al. “Approximate nearest neighbor algorithm based on navigable small world graphs”. In: *Information Systems* 45 (2014), pp. 61–68. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2013.10.006.

[MY20] Y. A. Malkov and D. A. Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2020), pp. 824–836. DOI: 10.1109/TPAMI.2018.2889473.

[RG19] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. arXiv: 1908.10084 [cs.CL].