Lab - 3: Scalable Retrieval with Project Gutenberg

Mihir Gohel (117284926)*
Stony Brook University
CS519, Data Science fundamentals

I. INTRO

All the files to the Lab3 of CS519 are available at Link.

II. PART 1- STEP 1

Book embedding visualization is given in Fig. (1) for max_books = 10000 and for 20 book titles. KNN graph Fig. (2) and threshold graph Fig. (3) for k = 10 and max_books=10000.

Q. Comment on the pros and cons of constructing an NN graph in each of the two ways.
Pros and Cons of NN Graph Construction:

- k-NN graph: Guarantees k neighbors per node; simple to implement. May connect distant points in sparse regions.
- ϵ-NN graph: Connects nodes within actual distance ϵ; preserves local structure. Can produce isolated nodes if ϵ is too small.

Q. How sensitive is the graph structure to the choice of d or e ?

- Sensitivity to hyperparameters: Graph structure is sensitive to k or ϵ; small values may yield disconnected nodes, while large values create denser graphs with possible noisy edges.

Q.Compare how graph connectivity changes with the hyperparameter (clusters vs. isolated nodes).

- Connectivity vs. hyperparameters: Increasing k or ϵ merges clusters and reduces isolated nodes; decreasing them splits clusters and increases disconnected components.

III. PART 1 - STEP 2

Q. What is the computational complexity of finding the exact d neighbors for each query?
Exact d neighbors: Computing exact d nearest neighbors for each query typically requires $O(n \cdot d)$ per query for a naive search over n points; for m queries, it is $O(m \cdot n \cdot d)$.

Q. How could approximate nearest neighbors reduce this complexity?
Approximate NN: Reduces complexity to search only a subset of points, often achieving sublinear time per query.

Q. What problems can occur with approximate NN (e.g., local optima)?
Problems with approximate NN: May return suboptimal neighbors due to local optima, poor coverage, or irregular data distribution.

Q. How could we bypass these problems (e.g., by using more than one random seed)?
Bypassing problems: Use multiple random seeds, ensemble methods, or repeated searches to improve coverage and reduce chance of missing true neighbors.

IV. PART 1 - STEP 3

Q. What is the computational complexity of inserting or deleting nodes exactly in this way? How does this scale as the graph grows?

- **Insertion:** Computing distances to all n existing nodes takes $O(n \cdot d)$, finding nearest neighbors is $O(n \log k)$ for top-k, and connecting edges is $O(k)$. Overall: $O(n \cdot d + n \log k) \approx O(n \cdot d)$.
- **Deletion:** Removing a node is $O(1)$, but repairing neighbors may require recomputing distances among affected nodes. In the worst case, this is $O(k \cdot d)$ per affected neighbor; if many nodes drop below k edges, complexity can approach $O(n \cdot d)$.
- **Scaling:** Both insertion and deletion scale roughly linearly with the number of nodes n, making exact updates expensive for large graphs.

Q. Where do the computational reductions come from in this approximate version? What trade-offs are introduced (e.g. possible missed neighbors, local optima)?
Computational reductions and trade-offs in approximate insertion/deletion:

- **Reductions:** Only a subset of nodes is visited during greedy traversal, avoiding $O(n)$ distance computations. Deletion avoids costly neighbor repairs. This reduces insertion/deletion complexity from $O(n \cdot d)$ to roughly $O(\log n \cdot d)$ or sublinear in practice.
- **Trade-offs:** Approximate methods may miss true nearest neighbors, get trapped in local optima, or

*Electronic address: mgohel@cs.stonybrook.edu

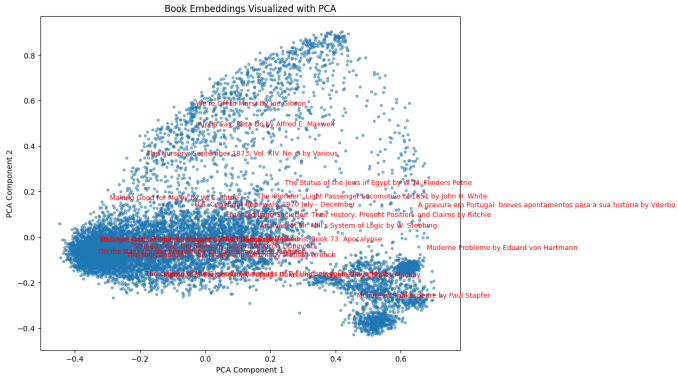

FIG. 1: Visualization with PCA

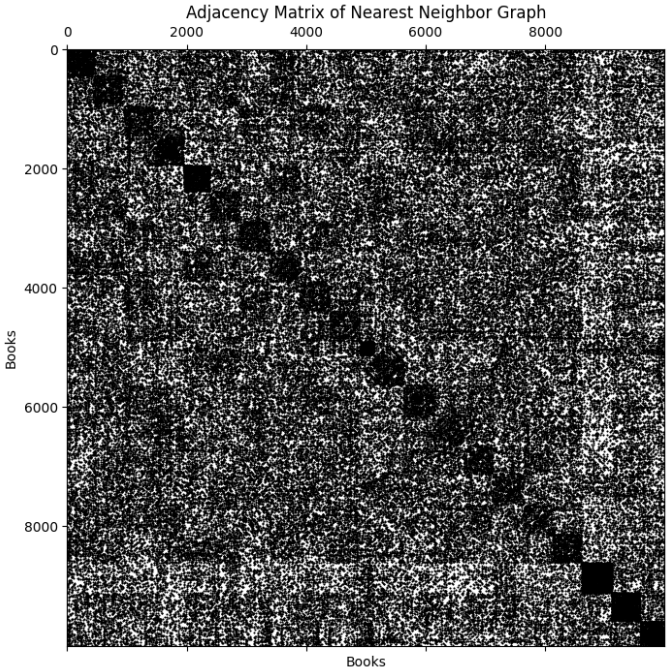

FIG. 2: KNN graph

create uneven connectivity. Graph quality depends on traversal strategy and randomness; coverage may be incomplete.

Q. Why is online add/delete functionality important in real retrieval systems?
Importance of online add/delete functionality:

- Real retrieval systems often receive new data or remove outdated items continuously.
- Online updates allow the graph to reflect the current dataset without rebuilding from scratch.
- Enables fast, up-to-date queries in dynamic environments (e.g., recommendation systems, search engines, social networks).

Q. Compare the costs of exact vs approximate methods. In what scenarios would exact updates be necessary, and when is approximation acceptable?
Cost comparison of exact vs. approximate methods:

- **Exact updates:** $O(n \cdot d)$ per insertion/deletion; costly for large datasets but guarantees correctness.
- **Approximate updates:** Sublinear or $O(\log n \cdot d)$ per update; much faster but may miss some neighbors or produce suboptimal connections.
- **Use cases:**
  - Exact: Critical systems requiring full accuracy (e.g., scientific datasets, legal or financial records).

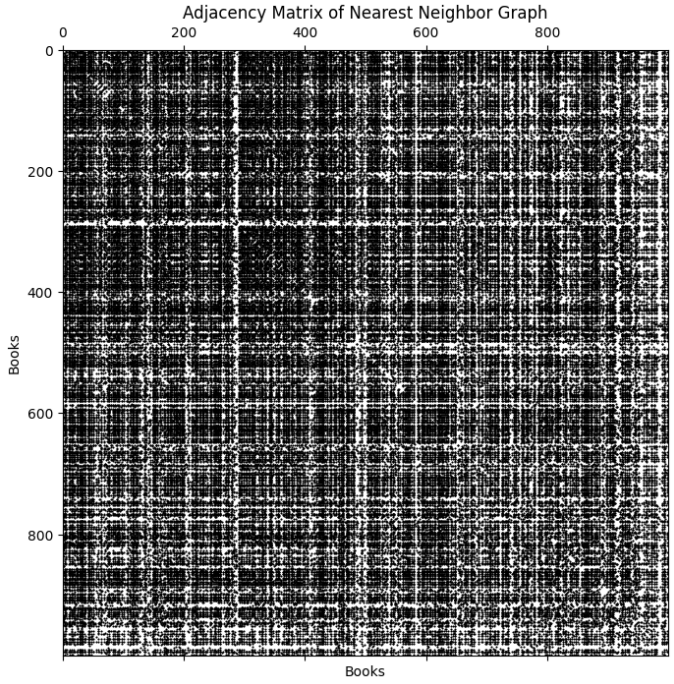

FIG. 3: Threshold graph

– Approximate: Large-scale, real-time systems where speed is more important than absolute precision (e.g., recommender systems, multimedia retrieval).

V. PART 2 - STEP 1

Q. Which layers is the densist?
The lowest layer ($\ell = 0$) is the densest, since it contains all nodes and the most detailed local connections.

Q. Which layer contains the entry point?
The topmost layer (highest L) contains the entry point, which serves as the starting node for searches and insertions, guiding traversal down through the hierarchy.

Q. What does the method `sample_level` return?
It returns the maximum level L assigned to a node, sampled from an exponential distribution. This determines how many layers the node will appear in — higher L values mean the node acts as a hub connecting multiple layers.

Q. hat is m_L in a graph?
m_L denotes the maximum number of connections (neighbors) a node can have at level L. Typically, m_L decreases with higher levels — fewer edges at top layers (for efficiency) and more edges at lower layers (for accuracy).

VI. PART 2 - STEP 2

Q. How many nodes are visited?
Flat graph visited 50 nodes.
HNSW graph visited 9 nodes.

Q. How does the path length differ?
Flat KNN Top 5: [20, 27, 8, 5, 39]
HNSW Top 5: [5]

Q. How are levels assigned to each node under this exponential scheme? Is a node assigned to just one layer, or multiple layers? Explain the exact assignment scheme. What are the advantages and disadvantages of this method compared to deterministic or uniform assignments?

- **Assignment scheme:** Each node's maximum level L is sampled from an exponential distribution, typically as

$$L = \lfloor -\ln(U) \cdot \lambda \rfloor,$$

where $U \sim \text{Uniform}(0, 1)$ and λ controls the decay rate. The node is then inserted into all layers from level 0 up to L. Thus, most nodes appear only in lower layers, while a few high-level nodes act as global hubs.

- **Advantages:**
  - Naturally forms a hierarchical small-world structure with logarithmic search complexity.
  - Ensures a balance between local density (bottom layers) and global connectivity (top layers).
  - No manual tuning of node distribution across layers is needed.

- **Disadvantages:**
  - Randomness may lead to uneven hub distribution or instability across runs.
  - Some important nodes might not appear in higher layers, slightly affecting search efficiency.

Q. Where do the main costs arise (construction, traversal, updates)? At which stages are these bottlenecks reduced or avoided? Why is this hierarchical design favored over a single flat graph?

1. **Main cost sources:**
   - **Construction:** Computing and maintaining nearest neighbors for each layer (especially the dense lower levels).
   - **Traversal:** Greedy search through multiple layers; distance computations dominate.
   - **Updates:** Inserting or deleting nodes requires adjusting connections across several layers.
2. **Bottleneck reduction:**
   - Higher (sparse) layers act as shortcuts, reducing search space and traversal time.
   - Only a small subset of nodes is explored during insertion and search.
   - Approximate neighbor selection avoids exhaustive distance computation.
3. **Why hierarchical design is favored:**
   - Provides a balance between global reach (top layers) and local precision (bottom layers).
   - Achieves logarithmic search complexity versus linear for flat k-NN graphs.
   - Enables scalable and efficient approximate nearest neighbor search with minimal accuracy loss.

VII. PART 3

Q. Computation: How does query speed change between the flat index and FAISS HNSW as the dataset size grows?

- **Flat Index:** Performs an exhaustive search over all n points, requiring $O(n \cdot d)$ distance computations per query. As the dataset grows, query time increases **linearly** with n.
- **FAISS HNSW:** Uses a hierarchical small-world structure for approximate search. Query time grows roughly logarithmically with n — only a small subset of nodes is visited. Thus, HNSW scales much better and remains fast even for large datasets.

Q. Memory: How much extra memory is used to store the graph structure in FAISS HNSW compared to the flat index?

- **Flat Index:** Stores only the raw vectors — memory usage is $O(n \cdot d)$.
- **FAISS HNSW:** In addition to the vectors, it stores graph connectivity (neighbor lists) for each node. Memory usage is approximately

$$O(n \cdot (d + M)),$$

where M is the average number of neighbors per node (typically 16–64). Hence, HNSW requires several times more memory than the flat index, trading space for faster query speed.

Q. Quality: How close are the retrieved neighbors from FAISS HNSW to the ground-truth neighbors found by exact search? (Measurectly measure the speed difference?

- **Quality:** Retrieved neighbors from FAISS HNSW are usually very close to ground-truth results. Accuracy is measured using Recall@k With proper tuning (`efSearch`, M), HNSW typically achieves $90-99\%$ recall.
- **Speed:** HNSW offers orders of magnitude faster queries than exact search, reducing complexity from $O(n \cdot d)$ to roughly $O(\log n \cdot d)$ per query.