Part 3 Report – KNN Search Using ALGLIB

Objective

The goal of Part 3 was to reproduce the k-nearest neighbor (KNN) functionality implemented in Parts 1 and 2 using the ALGLIB library. Specifically, we needed to load the same query and passage JSON files as before, wrap them in ALGLIB's array types, build a balanced k-d tree with "kdtreebuildtagged", perform both exact and approximate nearest neighbor searches with "kdtreequeryaknn", and then compare the performance and accuracy of our own KD-tree implementation from Part 2 with the ALGLIB-based implementation. The assignment also required us to investigate how ALGLIB's tunable parameter ε affects the pruning behavior, speed, and accuracy of approximate nearest neighbor (ANN) search.

Implementation

We reused the JSON parsing from Part 2. The query embedding was loaded into an "alglib::real_1d_ array". All passage embeddings were flattened into a row-major "std::vector<double>" and wrapped with "alglib::real_2d_array", while passage IDs were stored in an "alglib::integer_1d_array". With these arrays prepared, we constructed a balanced k-d tree by calling "kdtreebuildtagged(allPoints, tags, N, D, 0, 2, tree);" where "N" = number of passages, "D" = embedding dimension, "0" = no extra Y-values, and "2" = Euclidean norm.

For each query, we invoked "kdtreequeryaknn(tree, query, K, epsilon);" to retrieve the top-K neighbors. Distances and neighbor IDs were then retrieved with "kdtreequeryresultsdistances" and "kdtreequeryresultstags". The output format was kept identical to Part 2: query text followed by each neighbor's id, distance, and text passage.

Timing was measured with "std::chrono::high_resolution_clock" around four phases: total elapsed time (program start to end), input processing/parsing, KD-tree build, and KNN search only. This provided a direct comparison to the metrics collected in Part 2.

Testing Procedure

We ran both implementations on the same dataset with K=10 and ε=0 to measure exact KNN performance. The query file was "../part2/data/queries_emb.json" and the passage file was "../part2/data/passages1.json". We then ran the ALGLIB implementation with varying ε values to measure only the KNN query time (excluding data loading and tree construction) and to compute the accuracy of each ANN run by comparing the returned neighbor IDs to the exact set (ignoring order).

Results

When running exact KNN (ε = 0, K = 10), table 1 shows the timing breakdown for our KD-tree implementation from Part 2 and the ALGLIB-based implementation from Part 3. The ALGLIB run produced the exact same neighbor IDs and distances as our code, demonstrating correctness.

Implementation	Total Elapsed (ms)	Input Processing (ms)	KD-Tree Build (ms)	KNN Search (ms)
Part 2 (Our KD-tree)	6149.17	5858.45	270.96	19.75
Part 3 (ALGLIB)	30015	30015*	1367.97	61.26

Rank	Part 2 ID	Part 2 Dist	Part 3 ID	Part 3 Dist	Same?
1	4	0.648701	4	0.648701	Yes
2	5	0.696549	5	0.696549	Yes
3	12920	0.749874	12920	0.749874	Yes
4	26743	0.758476	26743	0.758476	Yes
5	6	0.769020	6	0.769020	Yes
6	0	0.779192	0	0.779192	Yes
7	3	0.779915	3	0.779915	Yes

8	68740	0.781726	68740	0.781726	Yes
9	40982	0.789043	40982	0.789043	Yes
10	63176	0.798562	63176	0.798562	Yes

Table 1 – Exact KNN Performance (ε=0, K=10)

As the table shows, our own KD-tree runs faster for exact search. ALGLIB's tree build is roughly five times slower due to double precision and additional bookkeeping, and its query time is about three times slower. However, the major gap in "total" time comes from console printing and parsing overhead in our Part 3 build.

ANN Sweep (ALGLIB)

Table 2 shows the effect of ε on KNN query time (tree already built) and accuracy relative to the exact results. Accuracy is defined as the number of overlapping IDs between ANN and exact divided by K, ignoring order.

k	ε (epsilon)	Search Time (ms)	Accuracy (%)
1	0.0	65.67	100
1	5	26.46	100
1	10	6.79	100
5	0.0	65.65	100
5	5	37.01	100
5	10	8.42	80
5	15	4.34	60
10	0.0	75.44	100
10	5	42.71	100
10	10	7.71	60
10	15	5.30	60

Table 2 – ANN Search Performance vs. Accuracy

Discussion:

Our experiments show that the ALGLIB-based KNN search produces exactly the same neighbors as our Part 2 KD-tree implementation when ε = 0, confirming correctness. As we increased ε, we observed the expected speed–accuracy trade-off. For K = 1, even large ε values gave 100 % accuracy with the query time dropping from 65 ms to under 7 ms. For K = 5 and K = 10, query time fell from about 65–75 ms at ε = 0 to roughly 8 ms at ε = 10 and about 5 ms at ε = 15, while accuracy remained 100 % for ε ≤ 5 but dropped to about 80–60 % at higher ε. These results confirm that ALGLIB's ε parameter prunes the k-d tree more aggressively as it increases, yielding dramatic speedups with a gradual loss of accuracy.

Conclusion:

In exact mode (ε = 0), our own KD-tree was faster than ALGLIB's because it is float-based and has less overhead, but both produced identical results. ALGLIB's strength lies in its approximate search capability: by tuning ε, we achieved substantial speedups with minimal accuracy loss for small ε values, and even greater speedups at the cost of accuracy for larger ε. This demonstrates that while a custom KD-tree can be more efficient for exact queries, ALGLIB offers a flexible and powerful way to trade accuracy for speed, which is especially advantageous for larger datasets or higher K values.