Multidimensional Data Structures Report

George Lalas 1093406, 4th year

A few words about the code:

1. Preprocessing, word separation, and lemmatization of each word were done. This process helps us compute Jaccard similarity.

def preprocess_text(text): text = re.sub(r'[^a-zA-Z\s]', '', text) text = text.lower() tokens = text.split() tokens = [WordNetLemmatizer().lemmatize(word) for word in tokens if word not in stopwords.words('english')] return tokens data['combine'] = (data['desc_1'].astype(str) + ' ' + data['desc_2'].astype(str) + ' ' + data['desc_3'].astype(str))

2. Creation of a KD-Tree using specific normalized variables. Initially, we use the scipy.spatial.KDTree library for the implementation and time to calculate the build time. Then, a center point and a radius are defined to set a specific search area in the multidimensional space. Using the query_ball_point method of the KD-Tree, all points within this region are searched.

start = time.time()

kd_tree_data = data[['100g_USD_normalized', 'rating_normalized', 'loc_country_encoded', 'roast_encoded']].values

kd_tree = KDTree(kd_tree_data)

kd_tree_build_time = time.time() - start

print(f"KD-Tree Build Time: {kd_tree_build_time:.4f} seconds")

Εκτέλεση query στο KD-Tree (παράδειγμα: εύρεση σημείων κοντά στο center)

center_point = [0.5, 0.5, 10, 2]

radius = 5.25

kd_tree_indices = kd_tree.query_ball_point(center_point, r=radius)

print(f"KD-Tree found {len(kd_tree_indices)} points within radius {radius} of {center_point}")

3. The 3D Octree structure was created without the use of a library, as this was not feasible. The logic of our implementation is the representation of each point with x, y, z coordinates, organizing them into nodes with defined boundaries, and splitting each node into 8 subspaces (with subdivide) when there are more than 4 points. We used three dimensions of data by adjusting the overall bounds of the structure to cover the entire 3D space. After constructing the octree, a search

is performed within a defined volume (bounding box), and further filtering is applied based on roast_encoded, aiming for more accuracy and better understanding of the examined data.

4. The creation of the 4D R-tree structure used the rtree.index library. Initially, the 4th dimension of space is defined, and each point is represented by four coordinates (x, y, z, w), corresponding to the values of 100g_USD_normalized, rating_normalized, loc_country_encoded, and roast_encoded. A search is executed within specific bounds with rtree_query_bounds, and a filter is applied based on roast_encoded. The R-tree achieves faster and more efficient searches in large databases, reducing the number of points.

```
p = index.Property()

p.dimension = 4

rtree_4d = index.Index(properties=p)

start = time.time()

for i in range(N):

  row = data.iloc[i]

  x = row['100g_USD_normalized']

  y = row['rating_normalized']

  z = row['loc_country_encoded']

  w = row['roast_encoded']

  rtree_4d.insert(i, (x, y, z, w, x, y, z, w))

rtree_build_time = time.time() - start

print(f"R-Tree Build Time: {rtree_build_time:.4f} seconds")


rtree_query_bounds = (0, 0, 0, 0, 1, 1, data['loc_country_encoded'].max(),
data['roast_encoded'].max())

rtree_candidates = list(rtree_4d.intersection(rtree_query_bounds))

filtered_rtree_results = [idx for idx in rtree_candidates if data.iloc[idx]['roast_encoded'] ==
target_roast]

print(f"R-Tree found {len(filtered_rtree_results)} points matching roast={target_roast}")
```

5. The 3D Range Tree structure was created for efficient point searches in a multidimensional space, without the use of external libraries. Each point is represented by coordinates (x, y, z) and organized hierarchically in nodes. The space is divided at the median of each dimension. Searching is based on predefined bounds (bounding box), and filtering is then applied based on

roast_encoded, keeping only the points that correspond to a specific roast type, aiming for greater result accuracy and better understanding.

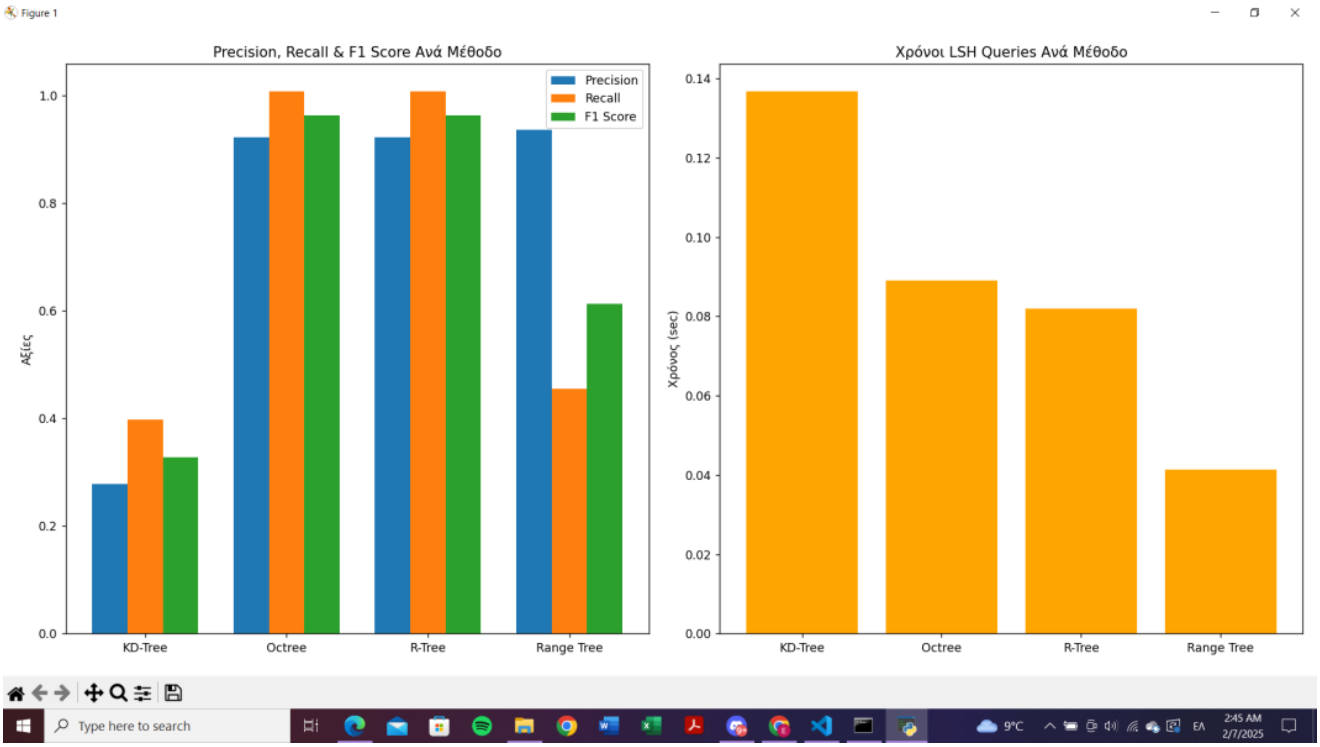
6. LSH creation. There are several techniques for implementing Locality Sensitive Hashing, but we chose the MinHash technique, as the comparison was on texts where similarity is measured using Jaccard Similarity. The data is converted into vector form using TF-IDF, and each text is processed and tokenized. Then, we created a MinHash signature with 128 permutated hashes for the representation of texts and an LSH table with a similarity threshold of 0.5 to store the signatures. This process helps retrieve similar data quickly without comparing all pairs.

7. In part #7: Definition of functions for Precision, Recall, and Jaccard Similarity, we implemented methods for calculating precision and recall. The idea is based on extracting the top (MAX_RESULTS) documents from the list of similar results and comparing them with the ground truth. The is_relevant function checks if a record has the same roast_encoded value as the target. Then, calculate_overall_precision_recall computes the overall precision and recall by comparing the data with the total set of relevant records. Additionally, using Jaccard similarity, we calculate the similarity between two sets based on their shared information. This process assists in objectively comparing the results of different search methods.

8. In the final step, #8: Execution of LSH queries and displaying results with Jaccard Similarity, we used the previously implemented MinHashLSH method for the KD-Tree, Octree, R-Tree, and 3D Range Tree structures. For each text, a comparison is made via LSH and Jaccard similarity, which evaluates the degree of similarity between documents. We recorded the LSH query time to compare the performance of the methods. Finally, overall precision and recall are computed for each structure using the methods in step 7, offering an estimate of search performance for each model.

9. This code section compares the performance of four data structures (KD-Tree, Octree, R-Tree, Range Tree) in terms of Precision, Recall, F1-score, and execution time. It uses Matplotlib diagrams to visualize the results and prints summary metrics for each method. Finally, it selects the optimal solution based on the highest F1-score. Visualization was done using the matplotlib and numpy libraries.

Visualization of Results



The "Precision, Recall & F1 Score" evaluates the accuracy, recall, and F1 Score for the KD-Tree, Octree, R-Tree, and Range Tree methods. The Octree and R-Tree have high values across all metrics, close to 1. The Range Tree has relatively good precision but lower recall. The KD-Tree performs the worst, with low Precision, Recall, and F1 Score.

The "LSH Query Times per Method" chart compares the execution time (in seconds) of the LSH queries for each method. The KD-Tree has the highest execution time. The Octree and R-Tree have similar times, lower than the KD-Tree but higher than the Range Tree. The Range Tree has the shortest execution time, meaning it's the most efficient method in terms of query execution time. However, Range Trees are suited for fast range queries but not for LSH similarity in text, since textual similarity is not expressed as a value range in one dimension. The R-tree is nearly equivalent to the octree but with better time performance.

Conclusion

The Octree is the optimal choice as it offers the best overall performance, combining high accuracy and recall with good execution times.
For the other trees, improvements can be made as follows:

1. KD-Tree can be optimized with substitution or balancing.
2. R-Tree can utilize a Hilbert R-Tree.
3. Range Tree can be combined with hashing for greater accuracy.

Required Libraries for Code Execution:

- pandas
- scipy.spatial.KDTree
- sklearn.feature_extraction.text.TfidfVectorizer
- re
- nltk.corpus.stopwords
- nltk.stem.WordNetLemmatizer
- datasketch.MinHash, MinHashLSH
- time
- rtree.index
- matplotlib.pyplot
- numpy

These libraries were installed via File > Settings > Project: Practice LSH > Python Interpreter and the implementation was done in PyCharm.