CAIRO UNIVERSITY
FACULTY OF ENGINEERING
CREDIT HOURS SYSTEM
ADVANCED DATABASE SYSTEMS
CMPS401 - FALL 2024

CMPS401 – Project Report

Prepared By:	Ali Mohamed Abdelghani	1210019
	Mohamed Ahmed Ibrahim Sobh	1210288
	Mohanad Tarek	1210313
	Ziad Hesham Fahmy	1210137
Submitted to:	Eng. Farah and Eng. Noaman	

Code Explanation for IVF Class

Introduction

The IVF (Inverted File) algorithm is used for efficient nearest neighbor search in high-dimensional data. It builds an index by clustering the data in multiple levels and allows fast searching by narrowing down to the nearest clusters before finding the most similar vectors. This document explains the steps involved in the IVF algorithm as well as all the failed attempts and the results.

Algorithm Steps

Step 1: Initialize IVF Class

The algorithm begins by initializing the IVF class with specific parameters such as the number of clusters, sub-clusters, and batch size. These parameters are used during clustering to build the hierarchical structure.

Step 2: Build Index

The 'build_index' function builds a hierarchical index of the data using multiple levels of clustering.

1. The NaN and inf data is first removed and the remaining is then normalized to unit vectors to ensure that the distance metric is cosine similarity.

```
# Remove rows with NaN or inf values
new_data = new_data[~np.isnan(new_data).any(axis=1)]
new_data = new_data[~np.isinf(new_data).any(axis=1)]

normalized_data = preprocessing.normalize(new_data)
```

2. The process includes two main clustering steps: the first-level clustering is performed using the KMeans algorithm (MiniBatchKMeans when the dataset is large, less accurate but faster)

```
# Initialize structures
hierarchy = {}
level_centroids = {}

# First level clustering
if len(new_data) >= 1000000:
    print('Using MiniBatchKMeans for first level clustering')
    kmeans_l1 = MiniBatchKMeans(self.cluster_num, random_state=0,
                                batch_size=self.batch_size, max_iter=10, n_init="auto")
    kmeans_l1.fit(normalized_data)
    l1_centroids = kmeans_l1.cluster_centers_
    l1_labels = kmeans_l1.labels_
    print('First level clustering complete')
else:
    l1_centroids, l1_labels = kmeans2(normalized_data, self.cluster_num)

level_centroids[0] = l1_centroids
level_centroids[1] = {}
```

3. The second-level clustering is performed within each cluster. For each first-level cluster, a second-level KMeans (or kmeans2) clustering is applied, resulting in smaller sub-clusters.

```
# Second level clustering
for i in range(self.cluster_num):
    print(f'Clustering for first level cluster {i}')
    cluster_data = normalized_data[l1_labels == i]

    # Remove rows with NaN or inf values in cluster_data
    cluster_data = cluster_data[~np.isnan(cluster_data).any(axis=1)]
    cluster_data = cluster_data[~np.isinf(cluster_data).any(axis=1)]

    # Debugging: Print the shape and check for NaNs or infs
    print(f'Cluster {i} data shape: {cluster_data.shape}')
    if len(cluster_data) == 1:
        # If the cluster contains only one vector, use it as the centroid
        l2_centroids = cluster_data
        l2_labels = np.array([0])
    elif len(cluster_data) > self.batch_size:
        kmeans_l2 = MiniBatchKMeans(self.sub_cluster_num, random_state=0,
                                    batch_size=self.batch_size, max_iter=10, n_init="auto")
        kmeans_l2.fit(cluster_data)
        l2_centroids = kmeans_l2.cluster_centers_
        l2_labels = kmeans_l2.labels_
    else:
        l2_centroids, l2_labels = kmeans2(cluster_data, max(2, self.sub_cluster_num))

    # Store second level centroids
    level_centroids[1][i] = l2_centroids
```

4. The hierarchical structure is saved using 'pickle' for later use in the search step. The hierarchical index is organized by storing indices of data points in files named 'ivf_hierarchy_{i}_{j}.pkl' where 'i' and 'j' refer to the cluster and sub-cluster numbers.

```
# Create leaf nodes with data indices
hierarchy[i] = {}
cluster_indices = np.where(l1_labels == i)[0]
for j in range(len(l2_centroids)):
    sub_cluster_indices = cluster_indices[l2_labels == j]
    hierarchy[i][j] = sub_cluster_indices.tolist()

    # dump the index in the folder
    with open(os.path.join(path, f'ivf_hierarchy_{i}_{j}.pkl'), 'wb') as f:
        pickle.dump(hierarchy[i][j], f)
```

5. The centroids are saved in file named ‘ivf_centroids.pkl’.

```
with open(os.path.join(path, 'ivf_centroids.pkl'), 'wb') as f:
    pickle.dump(level_centroids, f)
```

Step 3: Get One Row of Data

The 'get_one_row' function loads a single row from the database file. The database is stored as a memory-mapped file, which allows efficient loading of individual vectors without reading the entire dataset into memory. The data is loaded using 'np.memmap' with an offset based on the row number, ensuring that only the required row is fetched.

Step 4: Search for Nearest Neighbors

The 'find_nearest' function finds the nearest neighbors for a given query vector.

1. The process is broken down into several stages: first, the query vector is normalized

```
# Convert query to numpy and normalize
query = np.array(query).reshape(1, -1)
query = preprocessing.normalize(query)
```

2. Then, the centroids are loaded from ivf_centroids.pkl file and the similarities between the query and the first-level centroids are computed using cosine similarity.

```
# Load first level centroids
with open(os.path.join(path, 'ivf_centroids.pkl'), 'rb') as f:
    index_centroids = pickle.load(f)

first_level_centroids = index_centroids[0]

# Get similarities to first level centroids
sims_l1 = self.get_similarity(query, first_level_centroids)
sims_l1 = sims_l1.flatten() # Ensure 1D array
```

3. The nearest first-level centroids are identified.

```
# Get indices of nearest first level centroids (highest similarity)
nearest_l1_index = np.argsort(sims_l1)[-no_of_centroids:][::-1]
```

4. For each centroid, the corresponding second-level centroids are used to narrow down the search further. The data points from the second-level clusters are then retrieved and saved with the similarity to the query vector.

```
# Get similarities to second level centroids
sims_l2 = self.get_similarity(query, second_level_centroids)
sims_l2 = sims_l2.flatten() # Ensure 1D array

# Get indices of nearest second level centroids (highest similarity)
nearest_l2_index = np.argsort(sims_l2)[-no_of_centroids:][::-1]

# Collect data indices from nearest second level clusters
for l2_cluster_id in nearest_l2_index:
    l2_cluster_id_int = int(l2_cluster_id)
    with open(os.path.join(path, f'ivf_hierarchy_{l1_cluster_id_int}_{l2_cluster_id_int}.pkl'), 'rb') as f:
        data_indices = pickle.load(f)
    # Retrieve actual data vectors and compute similarities
    for data_index in data_indices:
        data_vector = self.get_one_row(data_index)
        sim = self.get_similarity(query, data_vector.reshape(1, -1))
        candidates.append((data_index, data_vector, float(sim)))
```

5. The search results are sorted by similarity in descending order, and the top 'k' nearest neighbors are returned.

```
# Sort by similarity (descending)
candidates.sort(key=lambda x: x[2], reverse=True)
# Return the top k nearest vectors
return [candidate[0] for candidate in candidates[:no_of_matches]]
```

Step 5: Changing Paths to be Dynamic

All of our paths and data files variables are dynamic, ensuring that there will not be changing in code or defintions when changing file names or changing their paths. This helped in creating index directories.

Perviously Failed Attempts

Single Level of Centroids

Didn't give any good results in term of time and accuracy, did not run on kaggle to check the RAM usage since other parameters were heavily failing on minimum requirements. This had us thinking whether the algorithm is not good enough or it's simply our mistake. And as mentioned by the TA, IVF should result on pretty good scores, not the best however.

One single index file

After fixing the issues, and implementing two level centroids, the accuracy and time taken were pretty decent. However, the RAM usage was not even close to the minimum requirement in 1M line.

Saving by value not ID

We started saving by ID to reduce RAM usage, still did not result on fine score in terms of RAM usage.

Results on Kaggle

Result 1: QUERY_SEED_NUMBER = 19

Database Size	Accuracy	Time in seconds	RAM in MB
1M	-8.0	1.88	0.12
10M	-48.333333333333336	4.48	0.62
15M	-62.666666666666664	5.53	0.25
20M	-693.3333333333334	7.82	0.00

Result 2: QUERY_SEED_NUMBER = 10

Database Size	Accuracy	Time in seconds	RAM in MB
1M	-13.666666666666666	2.06	0.12
10M	-46.333333333333336	4.71	0.12
15M	-65.0	5.54	0.50
20M	-521.0	7.12	0.00

Result 3: QUERY_SEED_NUMBER = 40

Database Size	Accuracy	Time in seconds	RAM in MB
1M	0.0	1.99	0.12
10M	-22.333333333333332	4.57	0.25
15M	-90.0	5.63	0.50
20M	-457.6666666666667	7.36	0.00