# Semantic-Search-Engine

Phase (2)

## Team (9)

| Name | Sec | BN | Code |
|---|---|---|---|
| Basma Hatem Elhoseny | 1 | 16 | 9202381 |
| Ziad Sherif | 1 | 26 | 9202586 |
| Zeyad Tarek | 1 | 27 | 9202588 |
| Abdel-hameed emad | 1 | 34 | 9202758 |

# Final Approach

## Inverted file Index (IVF)

We've really come a long way in making our VecDB system better, haven't we? It's been quite a journey, tweaking and adjusting things here and there. Let me give you a rundown of what we've been up to:

Firstly ,We moved from **MiniBatchKMeans** to regular KMeans. This change could make a big difference, especially when handling really big datasets. It's a bit of a balancing act, though, since it is usually quicker than **MiniBatchKMeans.**

We calculate initial centroids with just the first chunk of data. It speeds things up, but it might not give us the full picture. Maybe we should consider grabbing a random sample from the whole dataset for a better start.

A key aspect of our system is handling semantic queries. We implemented cosine similarity calculations using **a vectorized NumPy** approach, greatly reducing computation time. Furthermore, we introduced **parallel processing** for different regions, allowing simultaneous handling of dataset parts and faster query responses.

**Note**: After experimenting with various algorithms, we found the best results with the most recent approach implemented in the last two days before submission 🙂. This final method outperformed earlier techniques in handling our datasets effectively.

Clustering Techniques:

1. We thought of using MinibatchKmeans so that we loop over the while data and applying partial fit then we end up with centroids.Another run on the data to get the population of every cluster
2. Trying another metric for clustering that the default euclidean distance used by sk-learn so we use AgglomerativeClustering so we try another metric like the the cosine similarity .But we haven't

seen a lot of improvement so we used original k-means

```
# kmeans = MiniBatchKMeans(n_clusters=K_means_n_clusters, batch_size=k_means_batch_size, max_iter=k_means_max_iter,n_init=k_means_n_init,random_state=42)
kmeans=AgglomerativeClustering(n_clusters=K_means_n_clusters,linkage='average',metric=K_means_metric)
```

Index File Structure:

1. centroids.bin → Bin file with centroids of each cluster
2. cluster1.bin → the elements of cluster no 1
    a. Approach(1) to just store the id of the vector
    b. Approach (2) store vector & id

    firstly we have used the first approach and we found up that the size of the index folder is small but at the retrieval we have to open 2 files for every cluster we think of one to get the population of te cluster and the other is to open the original data.bin file with pointer to the location of the vector between to bt this is little bit slower so we preferred to have longer time indexing such that in the second approach we write more it on the other hand at retrieval we just open one file.

Other insights:

1. From HNSW we took the idea of not looking in the nearest cluster but the nearest k clusters
2. At the brute force stage at end instead sorting all the candidate vectors relative to each other as down in the code provided with the project

```
scores = sorted(scores, reverse=True)[:top_k]
urn [s[1] for s in scores]
```

    We just need the topK sorted so we used heapq with comeolsoty around O(n log k) instead of O(n log n), where her k is the the top_k which is contact so faster retrieval:D

```
    # Process the scores and maintain a heap
    for score, id in zip(similarities, records.keys()):
        if len(top_scores_heap) < top_k:
            heapq.heappush(top_scores_heap, (score, id))
        else:
            heapq.heappushpop(top_scores_heap, (score, id))

    # end_loop = time.time()
    # print("1 Region  Time",end_loop - start_loop)
# Sort and get the top k scores
top_scores_heap.sort(reverse=True)
top_k_ids = [id for _, id in top_scores_heap]
```

3. We thought of using the quantization approach of SCANN proposed by google by the problem is that Kmeans didn't allow us to use such idea.(no option is given to modify the quantzation technique used)

# Trials

## 1.LSH

We try the use of (LSH). However, the issue arose – the data wasn't evenly distributed across buckets. As we attempted to enhance time by increasing the number of buckets, we encountered a decrease in recall. Reducing the number of buckets resulted in increased recall but increasing the time.

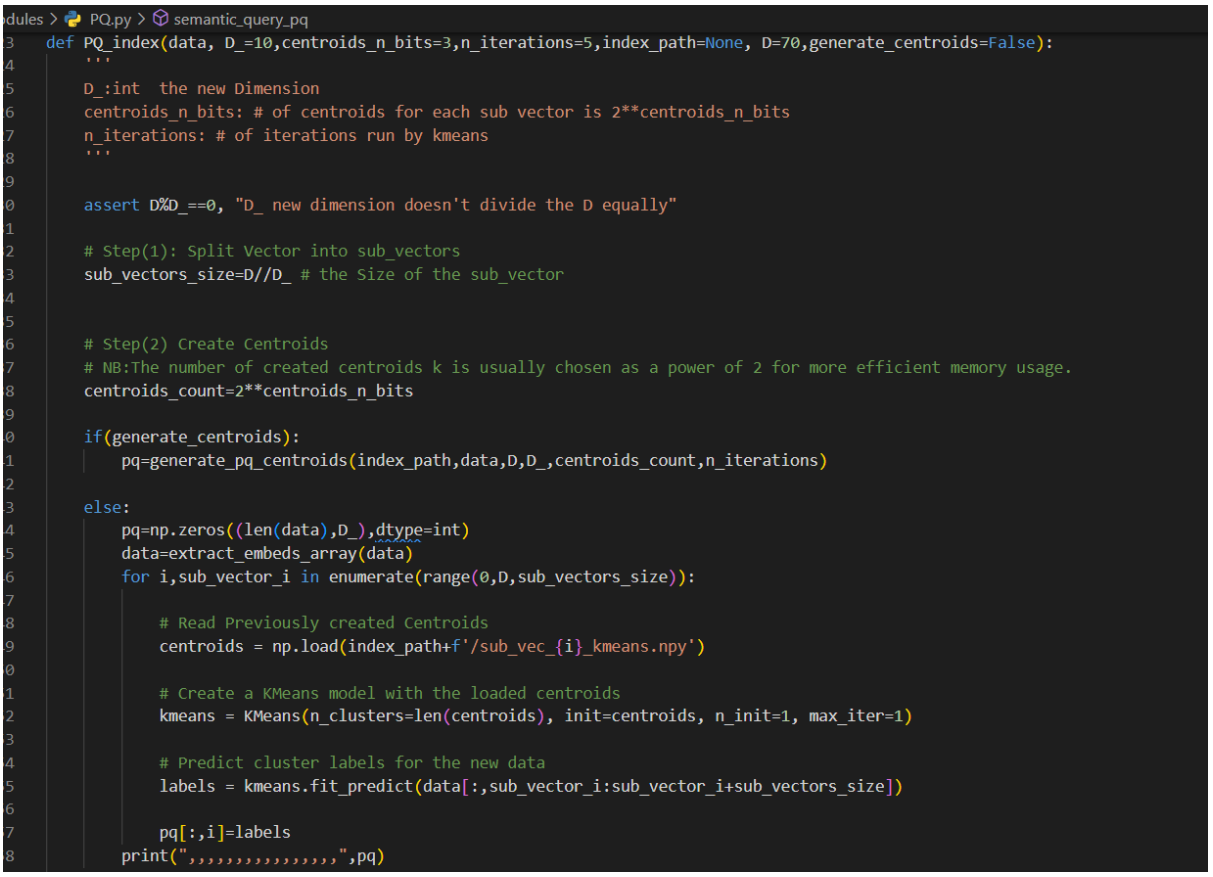
In order to increase the recall and solve the edge problem we try to select the top-k nearest buckets based on Hamming distance. This approach aimed to enhance recall by capturing more relevant data. Unfortunately, a trade-off emerged as the increase in recall was accompanied by an increasing search time.

here also we tried more than one level but due to inefficient distribution the recall decrease

# 2. PQ

In PQ it reduce the features too so it decrease the recall

We have tried to use PQ to reduce the dimensionality form 70 to 10 but we got very bad recall that at small databases at which we try initially we even got intersection between ours and the true id to be an empty set so we think this way don't we promising



```
dules > PQ.py > semantic_query_pq
def PQ_index(data, D_=10,centroids_n_bits=3,n_iterations=5,index_path=None, D=70,generate_centroids=False):
    '''
    D_:int  the new Dimension
    centroids_n_bits: # of centroids for each sub vector is 2**centroids_n_bits
    n_iterations: # of iterations run by kmeans
    '''

    assert D%D_==0, "D_ new dimension doesn't divide the D equally"

    # Step(1): Split Vector into sub_vectors
    sub_vectors_size=D//D_ # the Size of the sub_vector


    # Step(2) Create Centroids
    # NB:The number of created centroids k is usually chosen as a power of 2 for more efficient memory usage.
    centroids_count=2**centroids_n_bits

    if(generate_centroids):
        pq=generate_pq_centroids(index_path,data,D,D_,centroids_count,n_iterations)

    else:
        pq=np.zeros((len(data),D_),dtype=int)
        data=extract_embeds_array(data)
        for i,sub_vector_i in enumerate(range(0,D,sub_vectors_size)):

            # Read Previously created Centroids
            centroids = np.load(index_path+f'/sub_vec_{i}_kmeans.npy')

            # Create a KMeans model with the loaded centroids
            kmeans = KMeans(n_clusters=len(centroids), init=centroids, n_init=1, max_iter=1)

            # Predict cluster labels for the new data
            labels = kmeans.fit_predict(data[:,sub_vector_i:sub_vector_i+sub_vectors_size])

            pq[:,i]=labels
        print(",,,,,,,,,,,,,,,",pq)
```

## 3. IVF_PQ

We tried blending both ivf and PQ after clustering the data using K Means [Way discussed Below]. The we get the residual between the elements of each cluster so that all vectors are around the origin.In that step we no more need to store the vector itself we just need its residual [of course every vector with respect to its centroid].The reason behind steering teh residuals not the original vectors if we use vectors,then each subspace would contain  more various subvectors (because subspaces would store subvectors from different non-intersecting Voronoi partitions which could have been very far from each other in space).

Then next step is to apply PQ on the residuals [1x70] but here is another trick we apply PQ is not executed for each partition separately  at would be inefficient because the number of partitions usually tends to be high which will could result in a lot of memory needed to store all the codebooks.So apply PQ with all subregions together as we did before.

We haven’t continue din that step because we found ourselves making clustering in 2 parts one in the ivf and another time in PQ which ended up that we took many time in building the index which burdens us from trying a lot so we thought that we can use the first step of that approach is to just using IVF and increase no of clusters as shown above in our final approach