Semantic-Search-Engine

Phase(1) Initial Design & Findings

Team(9)

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

Design:

Layer(1) ……………………

Layer(2) ……………………

Pipeline

1st approach

We initiate the process by applying **LSH** as the initial step. It generates a norm by random hyperplanes and then computes the dot product of each vector in the dataset with these norm planes. This process results in assigning a binary vector to each dataset vector, determining whether it falls below or above the hyperplanes. We then group vectors with the same mapping into a packet. Next, we apply Hamming code to the input query to calculate the nearest distance, directing it to the nearest bucket containing subvectors similar to the input query. Subsequently, we apply K-means to cluster vectors within this bucket. The input query is once again passed to K-means to find the nearest centroid, facilitating a search within this group using **PQ** algorithm or brute force if memory constraints allow.

[NEED GUIDANCE]

Steps:

1. **LSH:** Begin by using LSH to create a norm based on random hyperplanes.
2. **Dot Product with Norm Planes:** Calculate the dot product of each vector in the dataset with the generated norm planes.
3. **Binary Vector Assignment:** Assign a binary vector to each dataset vector, indicating whether it is below or above the hyperplanes.
4. **Grouping:** Group vectors with the same binary mapping into packets.
5. **Hamming Code :** Apply **HC** to the input query to determine the nearest distance.
6. **Bucket Assignment:** Direct the input query to the nearest bucket containing subvectors similar to the query (most approximate similarity).
7. **K-means Clustering**: Apply K-means clustering to cluster vectors within the assigned bucket.
8. **Centroid:** Pass the input query to **K-means** to identify the nearest centroid.
9. **Searching:** Conduct a search within the identified group using **PQ** algorithm or brute force if memory permits.

Example of search space:

The initial LSH process might create **5 buckets**, each containing 20 million / 5 = **4 million records**.

After applying K-means, each bucket is divided into **100 clusters**, resulting in clusters of **40,000 records**.So we can search on them or make it recursive, as we increase plane norms leads to more accuracy.

2nd approach

We thought we were also using hybrid **IVF** with **HNSW**. ivf is responsible for creating an index file of our dataset then partitioning it into clusters or buckets, each containing vectors with similar characteristics (vector representation) . ivf provides a mapping of vectors to these clusters and HNSW searching about query input.. Hnsw works by constructing a hierarchical graph where vectors are connected based on their similarity . After that, navigate through this graph efficiently to find the nearest to the input query.

Handling Input Queries will be as:

When a new query is presented, first, use the **IVF** index to determine the **probable cluster**.Then pass the query to **HNSW** within this cluster for a more focused and optimized search.

We also may use **PQ** to enhance pression after **HNSW** for batched data

Note: We will think about Saving Models used in HNSW Later during implementation to find a turn around for it

[NEED GUIDANCE]

Unique Ideas:

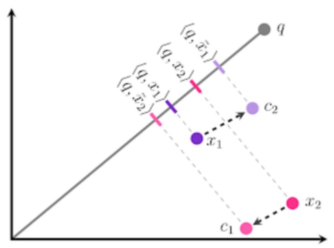
1. Split By Sign Approach: [Failed]

As a first Layer just split data according to the sign of the first 5 features Data will be split to 2^5 subclasses

We have canceled this approach because we are just judging the vector by each first 5 features only and we find out that the most similar approach for this idea in order to split search space is the LSH where splitting is done not based only on the first 5 features but the whole feature vector according to the hyper planes we will choose.

Splitting by sign is like LSH where hyperplanes are x1=0 and x2=0 x3=0 x4=0 x5=0 so taking General LSH we think it will be a better approach

2. SCANN: [To be Tried]



It is clear that x1 is nearer to q than x2

A New Quantization Approach for MIPS

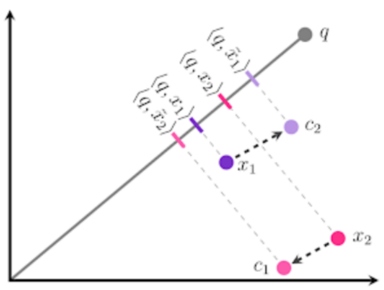
Problem: One of the most common ways to define the query-database embedding similarity is by their inner product; this type of nearest neighbor search is known as maximum inner-product search (MIPS). Because the database size can easily be in the millions or even billions, MIPS is often the computational bottleneck to inference speed. This necessitates the use of approximate MIPS algorithms that exchange some accuracy for a significant speedup over brute-force search.

Suppose we have two database embeddings x1 and x2, and must quantize each to one of two centers: c1 or c2.
Our goal is to quantize each xi to x̃i so that :

the inner product <q, x̃i> ≈ <q, xi> as possible

This can be visualized as making the magnitude of the projection of x̃i onto q as similar as possible to the projection of xi onto q.

The traditional approach to quantization:

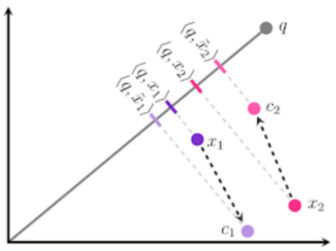


- we would pick the closest center for each xi
- <q, x̃1> > <q, x̃2>, even though <q, x1> < <q, x2>

Incorrect relative ranking of the two points □ x2 is nearer than x1 😟

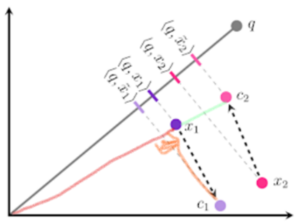
Assign x1 to C1 and x2 to C2:

Correct relative ranking of the two points □ x1 is nearer than x2 😊



Reasoning:

- The direction matters as well as magnitude,
- c1 is farther from x1 than c2 **BUT**
- c1 is offset from x1 in a direction almost entirely **orthogonal to x1**, **while c2's offset is parallel**



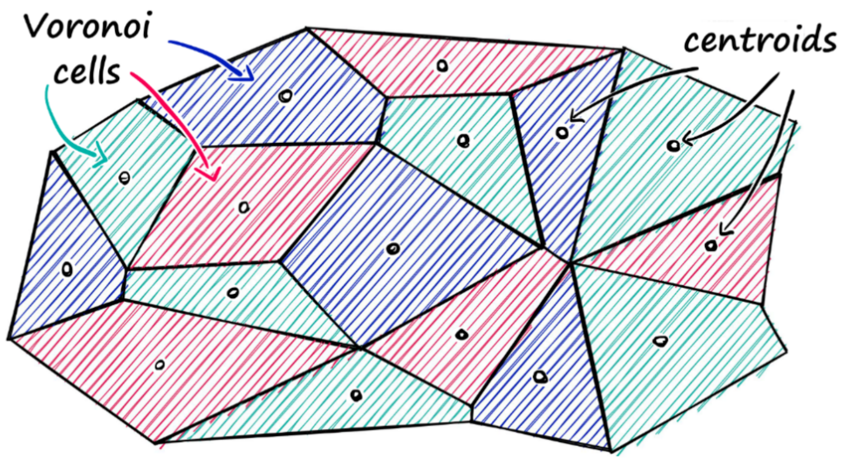
Error in the parallel direction is much more harmful in the MIPS problem because it **disproportionately impacts high inner products,** which is by definition are the ones that MIPS is trying to estimate accurately.

So we will try to use *anisotropic vector quantization*

Similarity Search Algorithms We Have Learned:

FAISS:

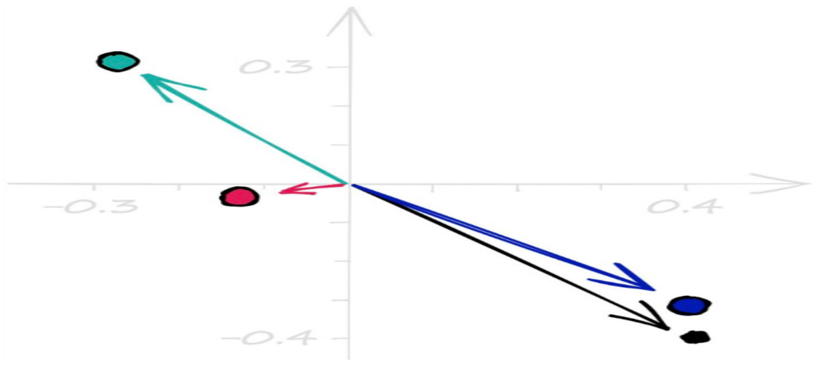
1. Designed for efficient similarity search in **large datasets**. It also provides algorithms and data structures optimized for fast NN search.
2. Choosing type of index based on your nature of dataset.. There are two types **indexflatl2** by using euclidean distance and **indexflat** for advanced approximate search.
3. Initialize index of what you choose index type
4. After that , we will train the index if we had used indexflat ..
5. Passing input query and then matches nearest centroid to query and searching in this voronoi.
6. We can also pass parameters as n-probe to get the most similar neighbors to query in order to be approximate so far and get rid of the bounding problem.



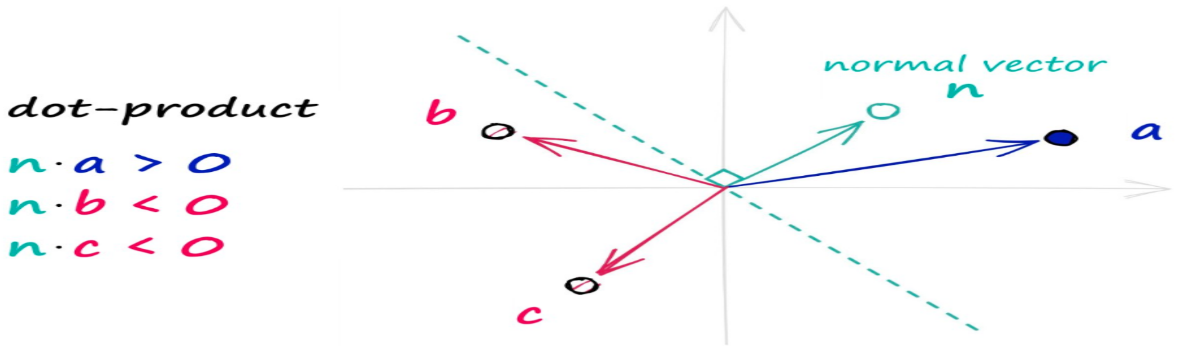
LSH: (using Random Hyperplanes)

Locality-Sensitive Hashing is employed to divide a **large** dataset (by using random hyperplanes) into buckets or hash tables in such a way that **similar** data points are likely to fall into the same or nearby buckets. This process facilitates efficient approximate nearest neighbor search by significantly **reducing the search space.**

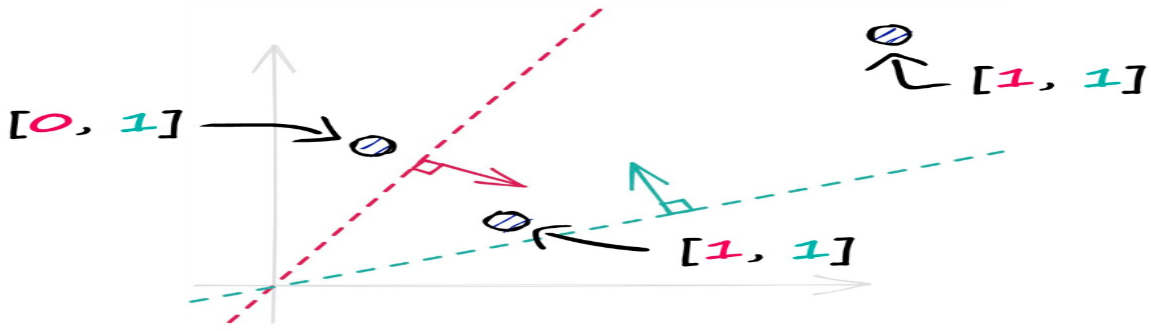
1. Generate nbits random hyperplanes in a space of dimension d. These hyperplanes will be used to divide the search space into 2^nbits buckets



2. For each data point (vector) compute the dot product between it and the normal of all hyperplanes

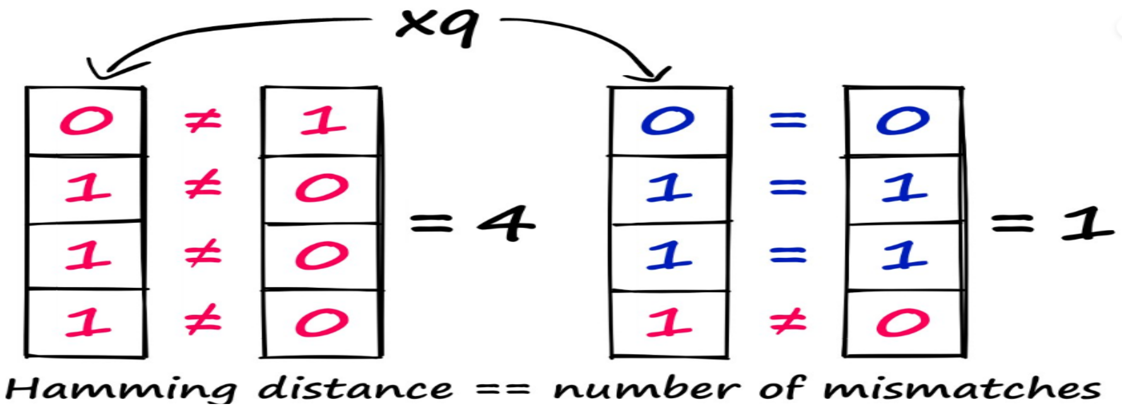


3. Each data point is represented as a binary hash code based on its position relative to these randomly generated hyperplanes.



4. Compute the hash code of the query point using the same random hyperplanes.

5. Compute the Hamming distance between the query hash code and the hash codes of the hash buckets, select the hash bucket with the smallest Hamming distance.



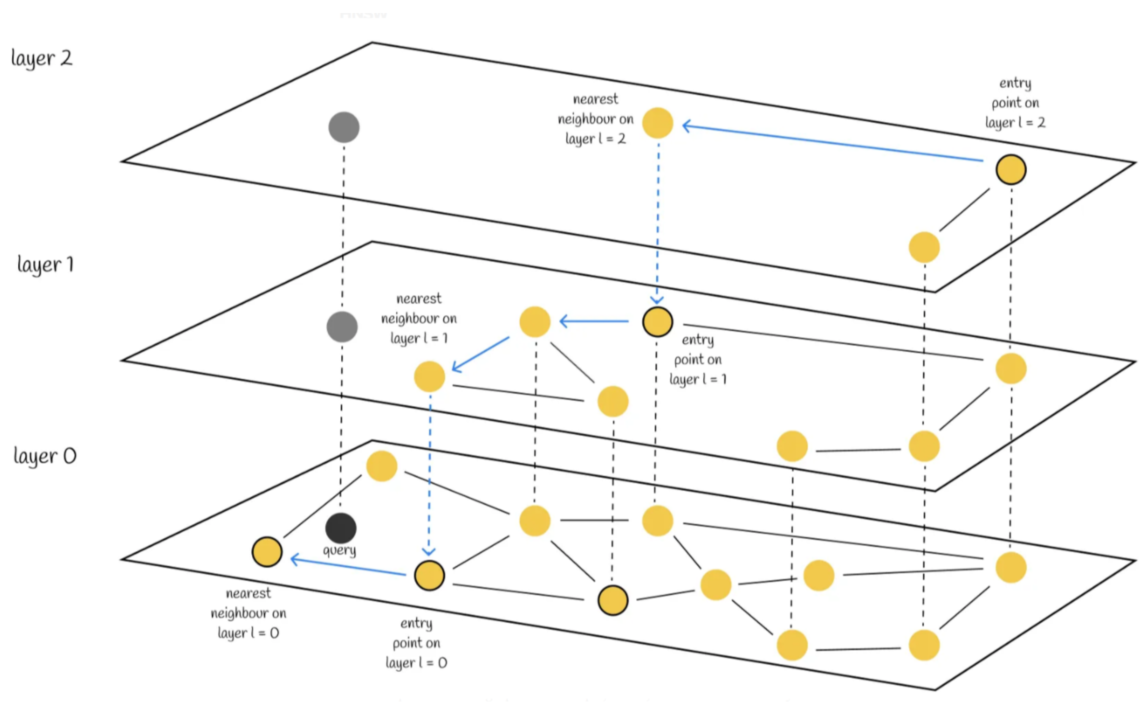
HNSW:

HNSW is an algorithm used for an approximate search of nearest neighbors.

It constructs optimized graph structures.

Its structure represents a multi-layered graph with fewer connections on the top layers and more dense regions on the bottom layers.

The search starts from the highest layer and proceeds to one level below every time the local nearest neighbor is greedily found among the layer nodes. Ultimately, the found nearest neighbor on the lowest layer is the answer to the query.


layer 2
nearest neighbour on layer l = 2
entry point on layer l = 2
layer 1
nearest neighbour on layer l = 1
entry point on layer l = 1
layer 0
query
nearest neighbour on layer l = 0
entry point on layer l = 0

The search quality of HNSW can be improved by using several entry points. Instead of finding only one nearest neighbor on each layer, the **efSearch** (a hyperparameter) closest nearest neighbors to the query vector are found and each of these neighbors is used as the entry point on the next layer.

Graph Construction:

Nodes in HNSW are inserted sequentially one by one. Every node is randomly assigned an integer l indicating the maximum layer at which this node can present in the graph. For example, if l = 1, then the node can only be found on layers 0 and 1. L is chosen randomly for each node with an exponentially decaying probability distribution normalized by the non-zero multiplier mL (mL = 0 results in a single layer in HNSW and non-optimized search complexity). Normally, the majority of l values should be equal to 0, so most of the nodes are present only on the lowest level. The larger values of mL increase the probability of a node appearing on higher layers.

$$l = \text{float}[-\ln(\text{uniform}(0, 1)) \cdot m_L]$$

The number of layers l for every node is chosen randomly with exponentially decaying probability distribution.

To achieve the optimum performance advantage of the controllable hierarchy, the overlap between neighbors on different layers (i.e. percent of element neighbors that also belong to other layers) has to be small

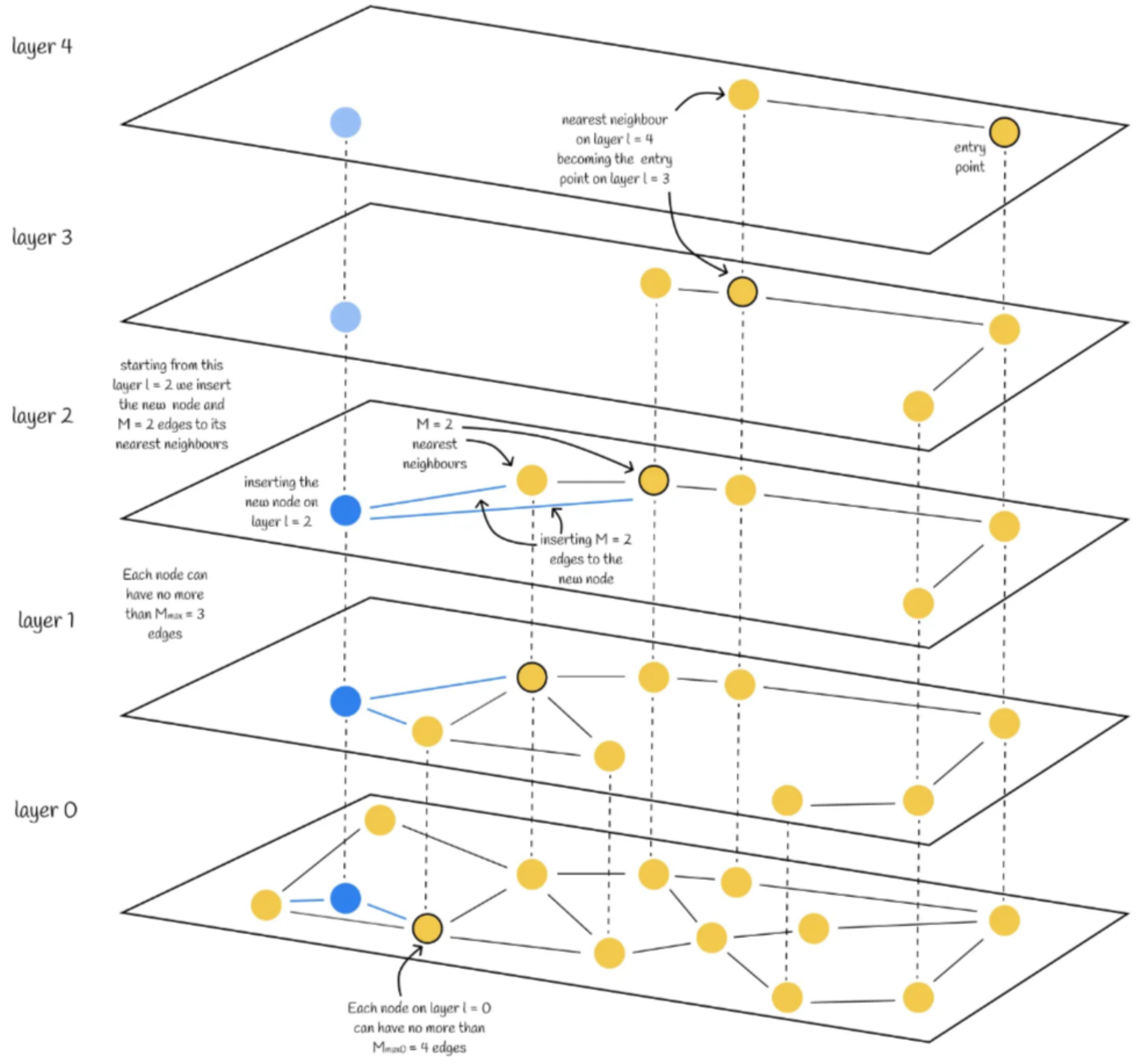
One of the ways to decrease the overlap is to decrease mL. But it is important to keep in mind that reducing mL also leads on average to more traversals during a greedy search on each layer. That is why it is essential to choose such a value of mL that will balance both the overlap and the number of traversals.

Insertion:

After a node is assigned the value l, there are two phases of its insertion:

1. The algorithm starts from the upper layer and greedily finds the nearest node. The found node is then used as an entry point to the next layer and the search process continues. Once the layer l is reached, the insertion proceeds to the second step.
2. Starting from layer l the algorithm inserts the new node at the current layer. Then it acts the same as before at step 1 but instead of finding only one nearest neighbor, it greedily searches for ***efConstruction*** (hyperparameter) nearest neighbors. Then M out of ***efConstruction*** neighbors are chosen and edges from the inserted node to them are

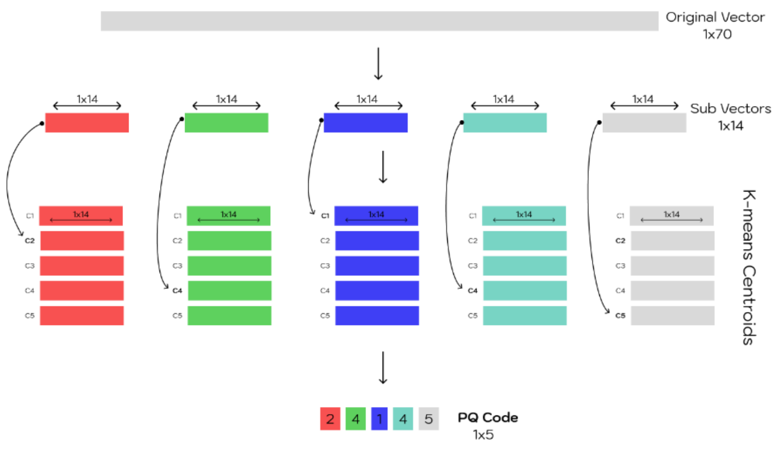
built. After that, the algorithm descends to the next layer and each of found ***efConstruction*** nodes acts as an entry point. The algorithm terminates after the new node and its edges are inserted on the lowest layer 0.


layer 4
nearest neighbour on layer l = 4 becoming the entry point on layer l = 3
entry point
layer 3
starting from this layer l = 2 we insert the new node and M = 2 edges to its nearest neighbours
layer 2
M = 2 nearest neighbours
inserting the new node on layer l = 2
inserting M = 2 edges to the new node
Each node can have no more than M_{max} = 3 edges
layer 1
layer 0
Each node on layer l = 0 can have no more than M_{max0} = 4 edges

Insertion of a node (in blue) in HNSW. The maximum layer for a new node was randomly chosen as l = 2. Therefore, the node will be inserted on layers 2, 1 and 0. On each of these layers, the node will be connected to its M = 2 nearest neighbours.

Product Quantization (PQ):

Main aim of this algorithm is to reduce dimensionality of the vector space so instead of having vector space of 70 dimension we can reduce It to lower dimension while preserving the semantic of all the 70 dimensions :D. It is a Memory-Efficient Algorithm



Note: Numbers used in the Diagram are just for illustration we will discuss K (no of centroids for each sub-vector) and the number of sub-vectors later.

K-means Algorithm is trained on every sub-vector to get k centroids for each sub-vector.

NB: We think we won't need to save a **.joblib** file for each K-mean model (we will have 1 model 2for each sub-vector) instead all we need from that model are its centroids so we will save them only and to get the nearest centroid to a sub-vector of a query just by Euclidean distance or any other metric we can find it suitable.

Finding Similar Data Base Vectors *:*

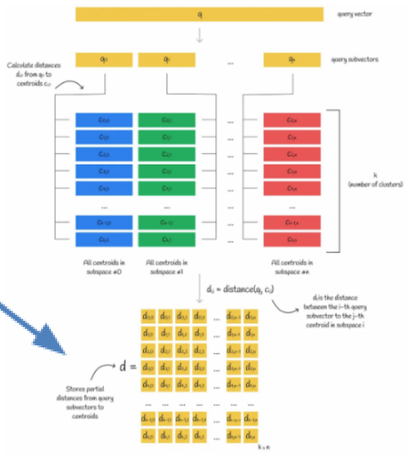
Approach(1): We will use This PQ to calculate the distance to all PQs representation of the query vectors in the DB and chose ones with shortest distance .But we found out that here we will get the distance between the PQ's which are just the IDs of the centroids **NO SEMANTIC MEANING**

Approach(2): We still will make brute force on the Data Base vectors but the problem how to map these database vectors to the PQ and from that we calculate similarity ??!

We Found a tricky way 😉

1. Divide the query vector into subvector just as we did before

2. Instead of Getting the Id of the need centroid for each subvector compute Matrix of the distance between each subvector and all the centroids [Each subvector and all its corresponding centroids] ⎕ d matrix



Obtaining a table d storing partial query subvector-to-centroid distances

That's all for the query vector :D

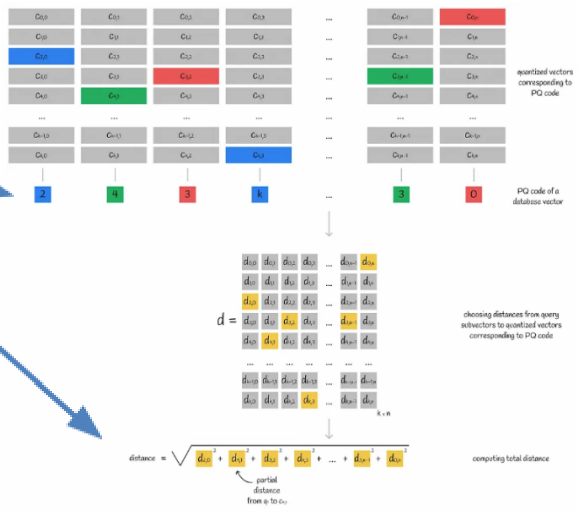
For each Database vector

3. Divide vector into same no of subvectors

4. Get its PQ code (same as we did in query before)

5. Use This PQ and the d matrix(obtained from the query vector) to calculate the Euclidean distance

<u>This distance is simply how this data base vector close to the query vector 😎</u>



Computing distance from a query to database vector by using PQ code and distance table

💡💡 After we have obtained approximate distances for all database rows, we search for vectors with the smallest values. Those vectors will be the nearest neighbors to the query.

📝 Note: This is an approximation because when calculating the distance it assumes that partial distances **d** are very close to actual distances **a** between query and database subvectors.