HCML PROJECT

Product Quantization with KDTREE

Muhammet Tarık Çepi, Dinh Khanh Thi Vo

1. Introduction

Product quantization is a dimensionality reduction technique that can be used to speed up nearest neighbor search. It works by dividing the original feature space into a number of smaller sub-spaces, and then quantizing each sub-space separately. This results in a much smaller representation of the original feature vector, which can be used to quickly search for similar vectors.

Kdtrees are a tree-based data structure that can be used to efficiently search for nearest neighbors in a high-dimensional space. They work by recursively partitioning the data space into smaller and smaller regions, until each region contains only a single point. This allows for efficient lookup of points that are close to a given query point.

Advantages of product quantization

- Speed: Product quantization can significantly speed up nearest neighbor search, especially in high-dimensional spaces.
- Accuracy: Product quantization can achieve high accuracy, even with a relatively small number of bits per dimension.
- Scalability: Product quantization is scalable to large datasets.

Advantages of kdtrees

- Accuracy: Kdtrees can achieve high accuracy, especially for queries that are close to the data points.
- Speed: Kdtrees can be very fast for queries that are close to the data points.
- Scalability: Kdtrees are scalable to large datasets.

Combining product quantization and kdtrees

The combination of product quantization and kdtrees can be used to create a very efficient indexing algorithm for nearest neighbor search. The product quantization step can be used to reduce the dimensionality of the data, which can significantly speed up the kdtree search. The kdtree step can then be used to find the closest neighbors of the query point.

This combination of algorithms has been shown to be very effective for nearest neighbor search in high-dimensional spaces. It can achieve high accuracy and speed, while being scalable to large datasets.

2. Method

2.1 Product Quantization

The embeddings as dataset are in the size of 1024. What Product Quantization does is to divide this embedding into segments. After applying this step to all embeddings in the gallery, we then train our vectors by running k-means clustering on each sub-space. The k-means clustering will generate k centroids (cluster centers) within each sub-space, based on the value of k that we choose. The centroids have the same length as the segment.

Once the training is complete, we can now compress the vectors in the database. For each segment of a vector, we find the nearest centroid from the corresponding sub-space. In other words, we only need to find the nearest centroid out of the 256 centroids in that sub-space.

Once we have found the nearest centroid for each segment, we substitute it with the centroid's ID. The centroid IDs are simply the indices of the centroids within the sub-space.

This process results in a compressed representation of the vectors, which are the short codes named as PQ Codes.

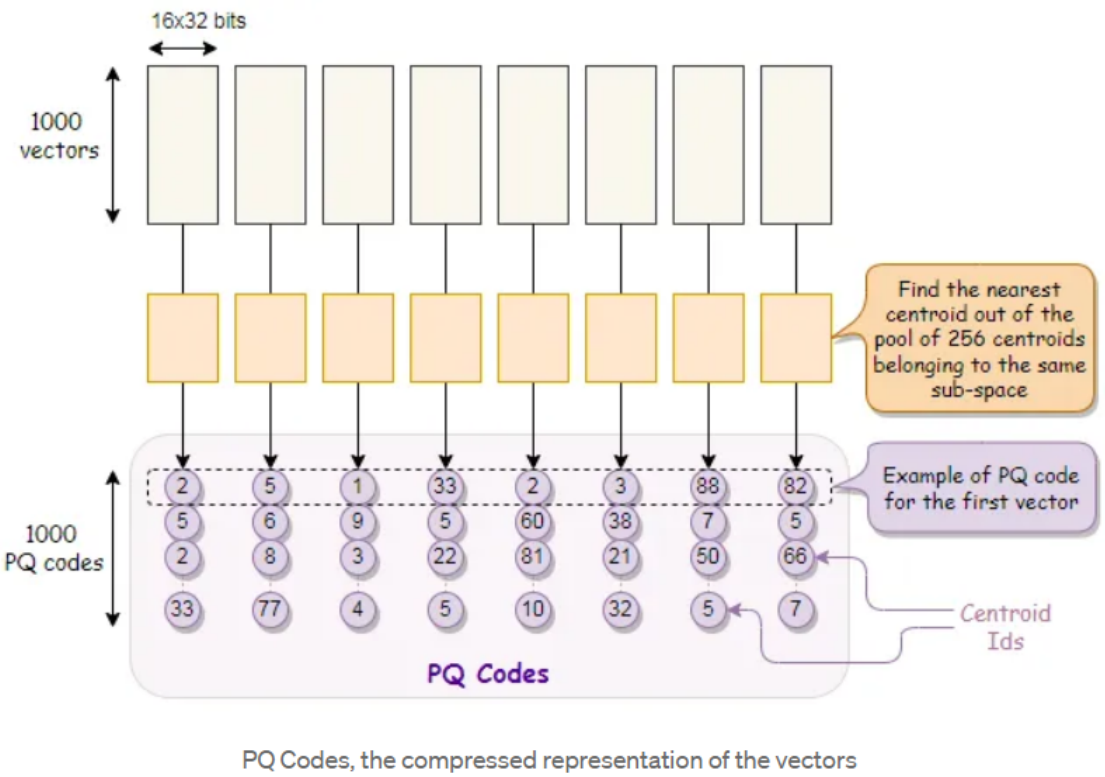

Figure 1(Creation of PQ-Codes)

Now that we have the compressed representations of the vectors in the database, we can start searching.

Suppose we have a query embedding. We can go through the same steps as we did during training, but with one difference. Instead of creating a PQ code for the query embedding, we create a distance table to all the centroids in the clusters.

The distance table is a matrix that stores the distances between the query embedding and all the centroids. This matrix can then be used to find the closest vectors to the query embedding.

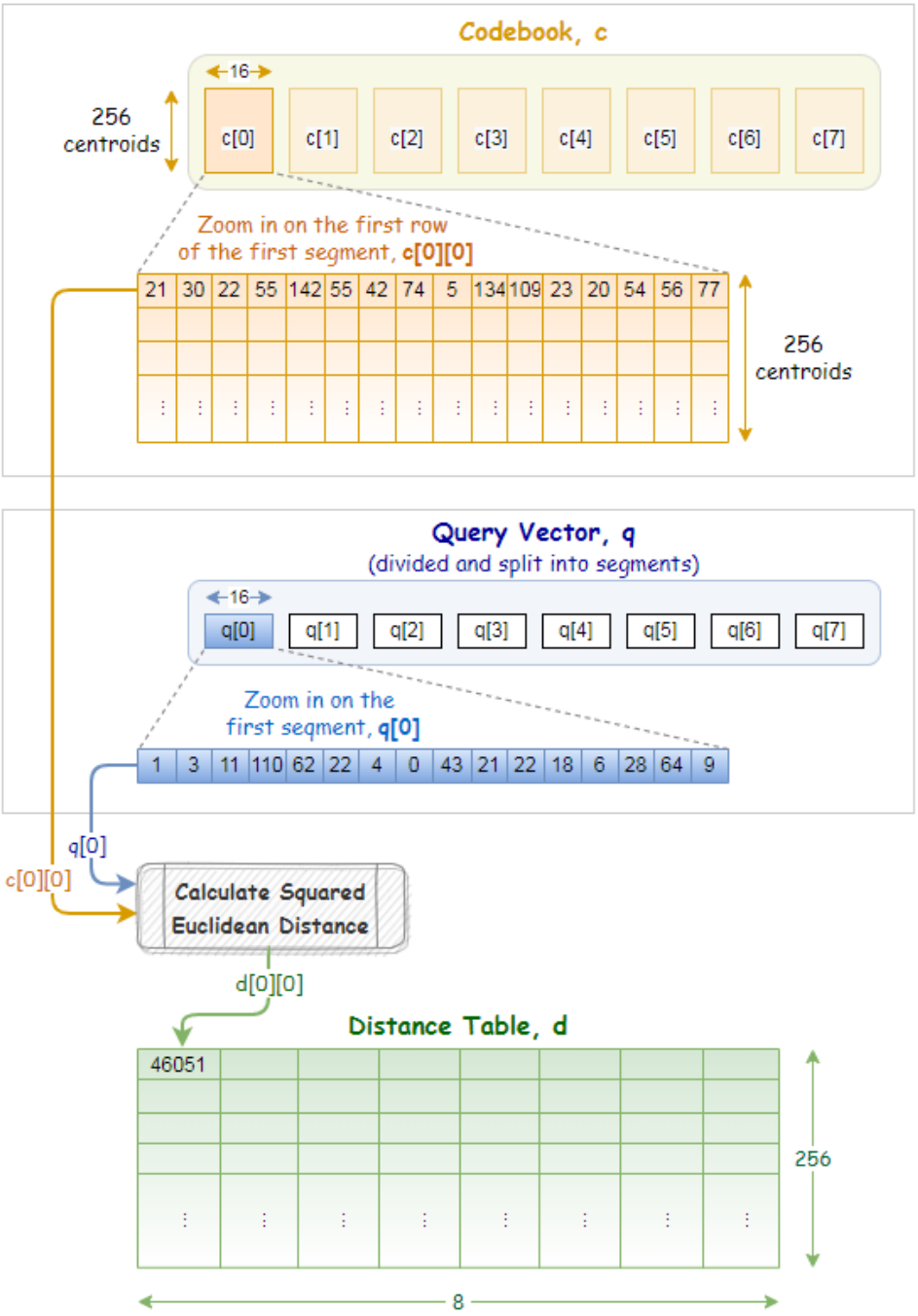

Figure 2(Distance Table Creation Process)

The PQ codes are simply the indices of the centroids that were used to compress the vectors in the database. When we search for a query embedding, we create a distance table to all the centroids. We then take all the PQ codes from the gallery and put them inside our tables. We sum the distances for each PQ code in the gallery, and the PQ code with the smallest distance is the identity we are looking for.

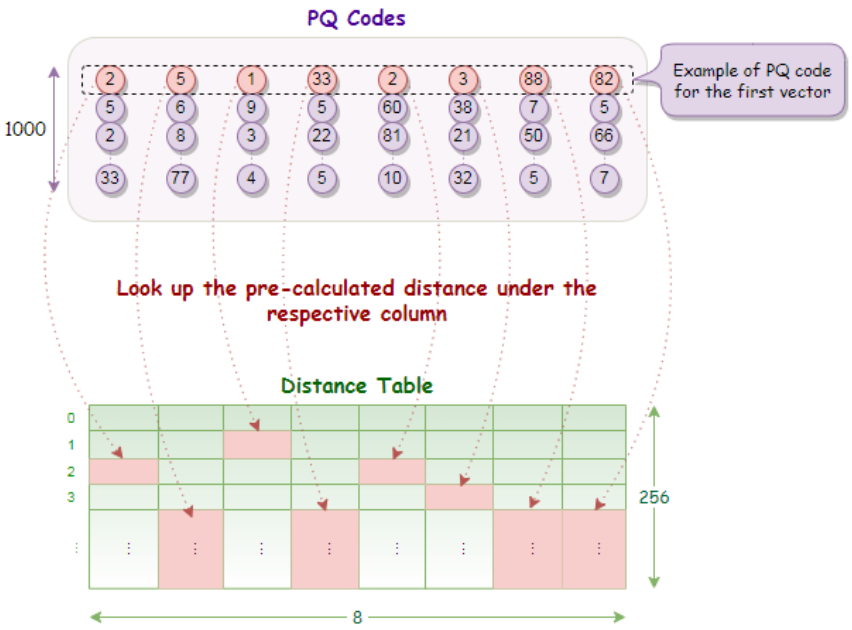

Figure 3 (Searching Process)

2.2 Product Quantization with KDTree

To speed things up we came up with the idea of using the KDTree. We create a tree using the gallery PQ Codes. In the searching part, instead of only creating a distance table, we also create a pq code also for the query embedding. We then give the query pqcode into our kdtree this gives us pq nearest pq codes.

After getting the nearest neighbour PQ Codes, we compute the distance of the returned pq codes and select the one with the least distance as our identity.

3 Evaluations

Evaluations are represented in terms of a plot of hit rate over penetration rate. The computation time is also documented.

Unfortunately, we were not sure of the correct definition of penetration rate. From our understanding, we interpreted the penetration rate in two different ways:

1/ Penetration rate is the percentage of data being used for the identification search. The hit rate would be the percentage of queries correctly answered from all queries.

2/ Penetration rate is the percentage of data being considered candidates for the query. The hit rate would be the percentage of queries correctly answered (with all candidates considered) from all queries.

Since we could not decide for one interpretation, we decided to implement both interpretations into our evaluations. Thus, the range of penetration rate in these two interpretations are different. Furthermore, the computation time is not comparable between the two interpretations, it is rather used to compare different searching techniques in each interpretations

The following “results” section will show evaluations in both interpretations.

4.Results

4.1.1/ Penetration rate as the percentage of data being used for the identification search

Processor: Intel® Core™ i7-8750H Processor

The baseline (Exhaustive Search Algorithm) took about 10 minutes to run and gave a hit rate of 96%.

Random Search worked as expected. X% of the data corresponds to almost X% hit rate. As in the figure below.

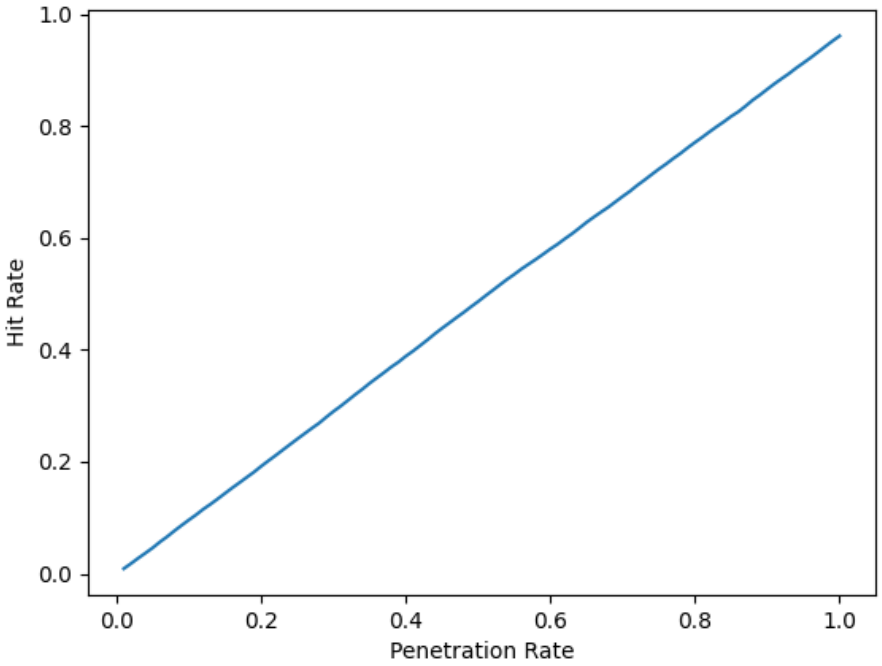

Figure 4 (Random Indxing)

Our approach as also can be seen on Figure 5 gives a slight bad hit rate but penetration rate is better. And also our computation time is much faster.

On a hit rate of 83% and a penetration rate of 43% it’s runtime is about 5 minutes.

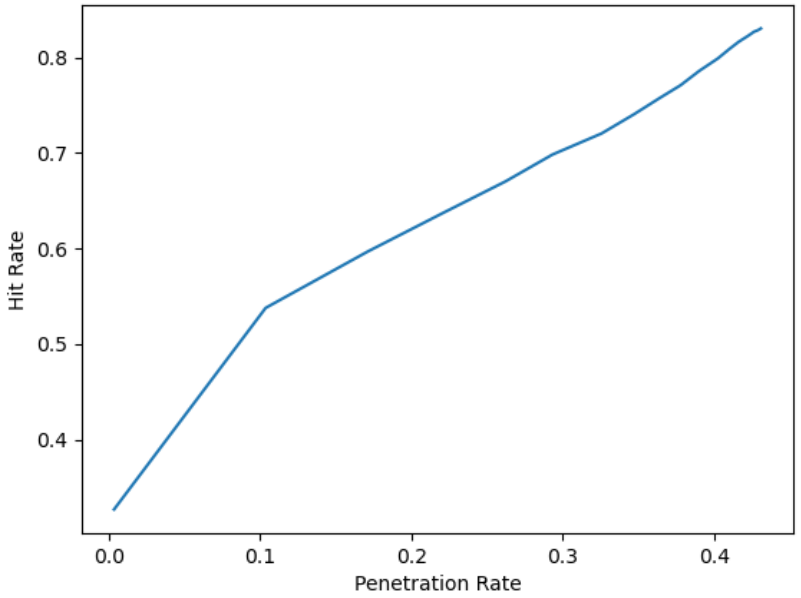

Figure 5 (PQ with KDTREE)

4.1.2 Limitations

Since we run two machine learning algorithms to get the end result, the accuracy drops. We give in on accuracy for computation time but the idea can be developed to get good results. It can be a good starting point to develop an algorithm.

4.2.1/ Penetration rate is the percentage of data being considered candidates for the query

Processor: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz

The baseline (Exhaustive Search Algorithm) took an average of 0.021s computation time for each query:

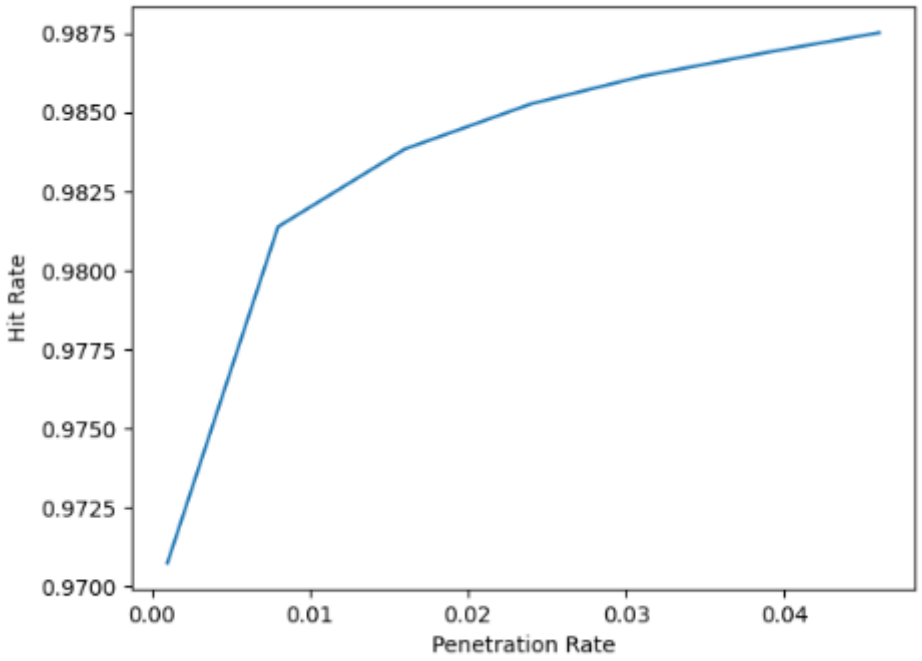

Figure 6 (Exhaustive Search)

The Product Quantization approach (without using KDTree) took an average of 0.015s computation time for each query. We can also see that the hit rate dropped somewhat compared to the exhaustive search:

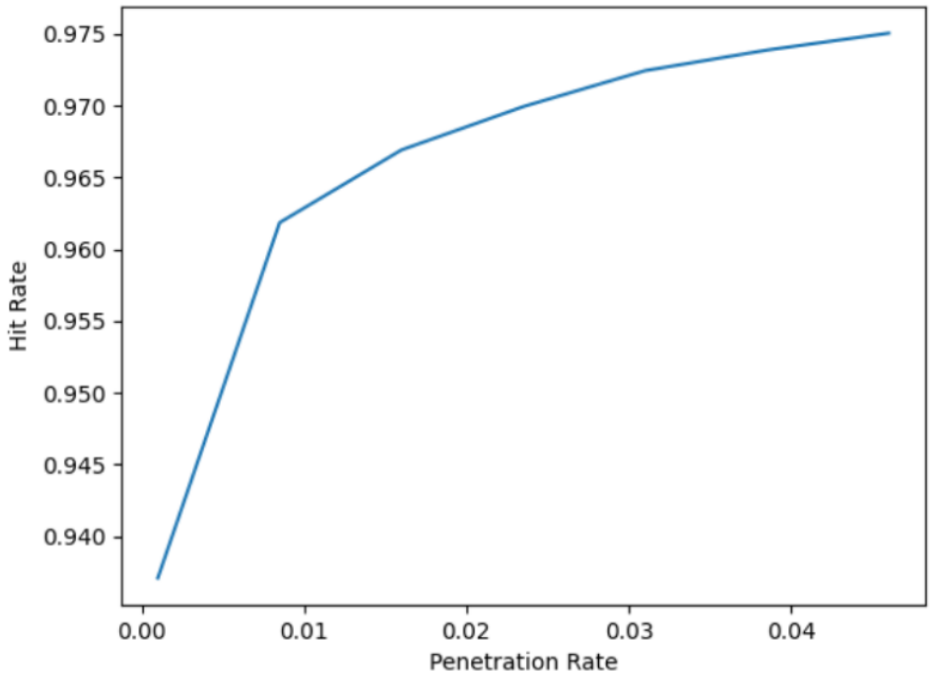

Figure 7 (PQ Exhaustive Search)

The Product Quantization with KDTree approach took an average of 0.0098s computation time for each query. However, even with a wider penetration rate range, here the hit rate dropped significantly:

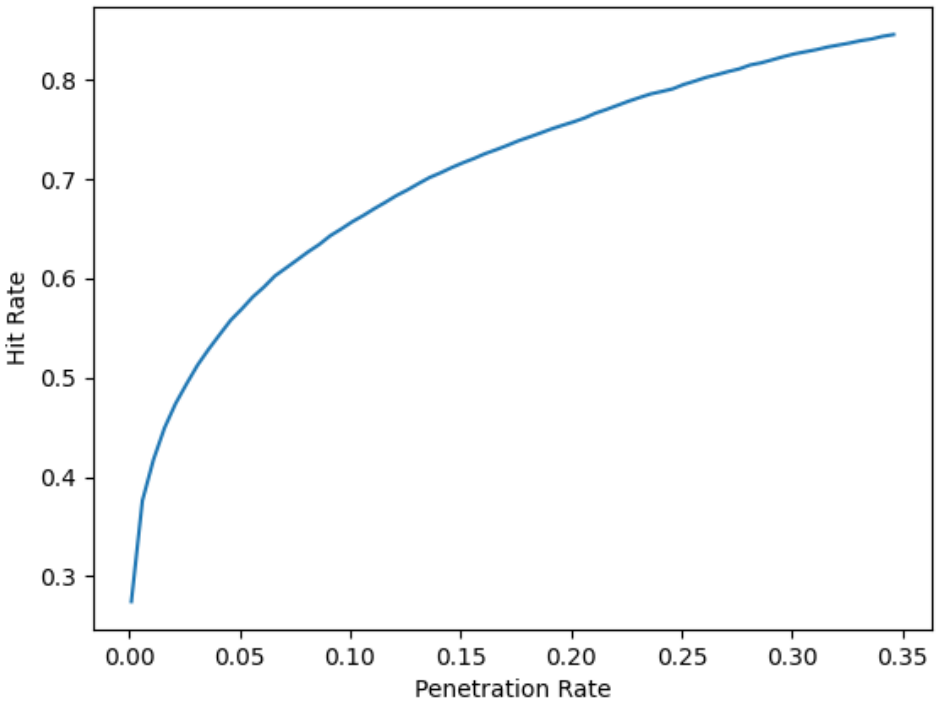

Figure 8 (PQ with KDTREE)

4.2.2 Limitations

We came to the conclusion that there is a trade-off between accuracy and computation time.

5.References

https://towardsdatascience.com/product-quantization-for-similarity-search-2f1f67c5fddd