Data Warehousing and Data Mining

SCIENTIA

MANU ET MENTE

UNSW
AUSTRALIA

Project Report

T1, 2020

Team **- The_Dementors**

Kovid Sharma (z5240067)

Nishant Chokkarapu (z5242882)

Table of Contents

Abstract

For this project, we are required to devise an algorithm/technique where we'll implement the product quantization method with L1 distance as the distance function and then will implement the query method using the idea of inverted multi-index. To do achieve this, we are only allowed a max of 3 seconds for the product quantization and 1 second for the query.

Introduction

The test dataset provided for this is toy_example that has 3 files :

1. Data_File with shape (768, 128)
2. Centroids_File with shape (2, 256, 64)
3. Query_File with shape (1, 128).

Part1: PQ for L1 Distance

The function pq() takes FOUR arguments as input:

1. **data** is an array with shape (N,M) and dtype='float32', where N is the number of vectors and M is the dimensionality.
2. **P** is the number of partitions/blocks the vector will be split into. In this we were required to implement a more general case where P can be any integer >= 2. We were allowed to assume that P always divides M.
3. **init_centroids** is an array with shape (P,K,M/P) and dtype='float32', which corresponds to the initial centroids for P blocks. For each block, K M/P-dim vectors are used as the initial centroids. K is fixed to be 256.
4. **max_iter** is the maximum number of iterations of the K-means *clustering algorithm.* ***Note*** *that in this project, the stopping condition of K-means* clustering is that the algorithm has run for max_iter iterations.

The pq() method returns a codebook and codes for the data vectors, where

- **codebooks** is an array with shape (P, K, M/P) and dtype='float32', which corresponds to the PQ codebooks for the inverted multi-index. E.g., there are P codebooks and each one has K M/P-dimensional codewords.

- **codes** is an array with shape (N, P) and dtype=='uint8', which corresponds to the codes for the data vectors. The dtype='uint8' is because K is fixed to be 256 thus the codes should integers between 0 and 255.

Part2: Query using Inverted Multi-index with L1 **Distance**
The query method uses the idea of inverted multi-index with L1 distance. It take the following arguments as input:

1. **queries** is an array with shape (Q, M) and dtype='float32', where Q is the number of query vectors and M is the dimensionality.
2. codebooks is an array with shape (P, K, M/P) and dtype='float32', which corresponds to the codebooks returned by pq() in part 1.
3. **codes** is an array with shape (N, P) and dtype=='uint8', which corresponds to the codes returned by pq() in part 1.
4. **T** is an integer which indicates the minimum number of returned candidates for each query.

The query() method returns an array contains the candidates for each query. Specifically, it returns

- **candidates** is a list with Q elements, where the i-th element is a **set** that contains at least T integers, corresponds to the id of the candidates of the i-th query.

• Methodology

1. We use cdist method with "cityblock" as metric from scipy.spatial.distance to calculate the L1 distance and improve the efficiency of our algorithm, for Part I.
2. Our approach to this project is to extend the algorithm 3.1 mentioned in the article (Artem Babenko and Victor Lempitsky. The Inverted Mulm-Index. TPAMI 2014.) and make it efficiently work for P>2. We evaluate its performance on the data provided. If we are satisfied with the performance of this model, we will submit the results for feedback.

PART I

For **product quantization**, the data is split into P partitions. Then each partition along with their respective codebook with the max-iterations is passed into the K-medians algorithm. The algorithm runs for 20 iterations and for every iteration it updates the code and codebooks. After the 20th iteration, the codes are again updated using the 20th iteration codebooks to get the final codes.

We constructed a method **k_medians** which would give us the codes and the codebooks for the i-th partition of the data.
How it works is, first, the new codes are calculated for using the codebook obtained from the previous iteration or initial centroid if it's the 1st iteration. The new codes are obtained from the distance_cal method. After getting the codes we need to Update the codebook for the next iteration. The updating of the codebook is done in the update_code_book method.

- **k_medians(obs, code_book, k_list, iter=20) takes 4 parameters :**

1. obs:
   The obs are one of the i-th partitions of the data.
2. code_book:
   Codebooks are the centroids of the i-th partition of the data.
3. k_list:
   K_list is a list of integers from 1 to the number of observations in the data
4. iter:
   The maximum number of iterations the K-median Algorithm has to run

- **update_code_book(obs, codes, code_book, k_list) takes 4 parameters :**

1. obs:
   The obs are one of the i-th partitions of the data.
2. code_book:
   Codebooks are the centroids of the i-th partition of the data.
3. codes:
   Codes are the cluster number to which each of the observation belongs to.
4. k_list:
   k_list is a list of integers from 1 to the number of observations in the data

This function returns the updated new codebook for the i-th partition.
Procedure:
First, using the dict_list function each of the observation is assigned to the cluster in a dictionary. For every cluster a list is created to which each of the vector of the cluster is appended from the observations and then converted into an array.
Then the median is calculated using the **np.median** method.
The centroid which does not have any observation close to it will not be present in the cluster dictionary and hence it is calculated and stored in the missing_centroid using the K_list. Then the missing centroid is taken from the code_book and inserted into the new codebook.

- **distance_cal(obs, code_book, query_type) takes 4 parameters :**

1. obs:
   The obs are one of the i-th partitions of the data.
2. code_book:

Codebooks are the centroids of the i-th partition of the data.

3. query_type:
   Query_type is to indicate if the distance calculation is for the K-median which is indicate by 'PQ' or for finding the distance for a query, indicated by "Query".

Procedure:

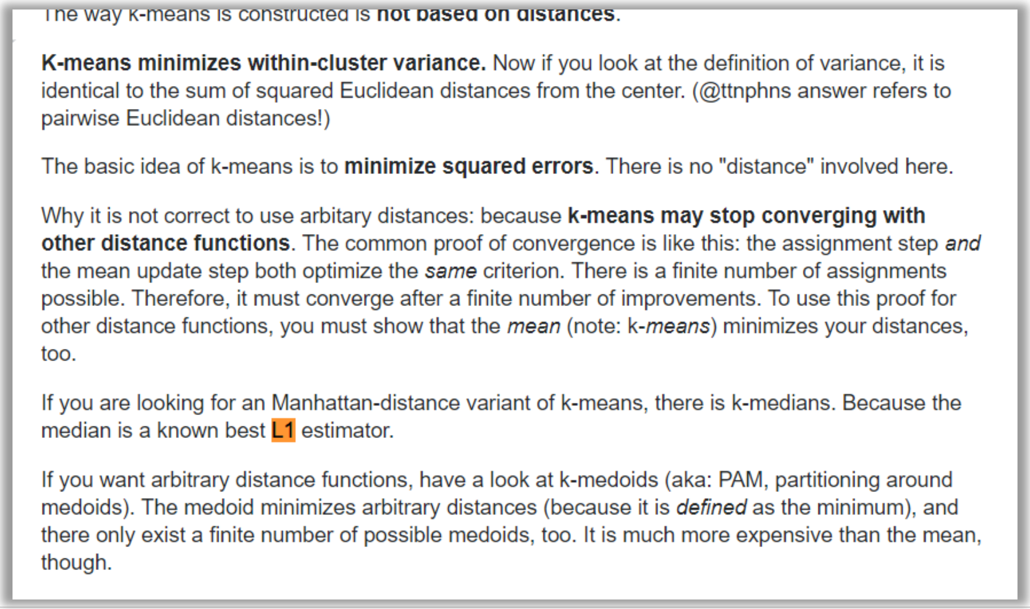
First, the distance array is calculated using the **cdist** function from the **scipy.spatial.distance.**

We used the mertic as "cityblock" to make our PQ algorithm run on L1 distance.

For "PQ",
We have used the **np.argmin** function to find the observation which belongs to which centroid and return the **new codes**.

For "Query",
We have used the **np.agrsort** to sort the distance array to find which centorid is the closest to our query, along with the cost. Then two separate arrays, one with codes and the other with cost of each code is returned

Initially we were using k-means, by taking the mean of the datapoints of each cluster as the new centre (and failing the autotest).
An alternative approach was to take a component wise median, we needed to do a component-wise median for the new centroid values.
And according to the paper, if we choose L1 distance, should we take median for the k-means computing as median is a better estimator for L1, as cited here:
https://stats.stackexchange.com/questions/67742/representative-point-of-a-cluster-with-l1-distance



The way k-means is constructed is **not based on distances**.

K-means minimizes within-cluster variance. Now if you look at the definition of variance, it is identical to the sum of squared Euclidean distances from the center. (@ttnphns answer refers to pairwise Euclidean distances!)

The basic idea of k-means is to **minimize squared errors**. There is no "distance" involved here.

Why it is not correct to use arbitary distances: because **k-means may stop converging with other distance functions**. The common proof of convergence is like this: the assignment step *and* the mean update step both optimize the *same* criterion. There is a finite number of assignments possible. Therefore, it must converge after a finite number of improvements. To use this proof for other distance functions, you must show that the *mean* (note: k-*means*) minimizes your distances, too.

If you are looking for an Manhattan-distance variant of k-means, there is k-medians. Because the median is a known best L1 estimator.

If you want arbitrary distance functions, have a look at k-medoids (aka: PAM, partitioning around medoids). The medoid minimizes arbitrary distances (because it is *defined* as the minimum), and there only exist a finite number of possible medoids, too. It is much more expensive than the mean, though.

PART II

For finding the **candidates**, the query is split into partitions based on the number of codebooks. Then, we create a dictionary where we find which observation belongs to which cluster using the codes, with the help enumerate method. Consider only the i-th partition of the query along with the i-th codebook which is sent to the distance_cal method which calculates the distance from the i-th partition of query and each observation of the codebook and returns the costs and the codes in a sorted manner.
Then, inside the while loop we calculate the candidates for each of the query and append it to the final list.

- **dict_list(codes):**

codes:
Codes are the cluster number to which each of the observation belongs to.
This returns a dictionary with key as the cluster number and values as the list of number. Each number corresponds to the row number in the observation data.

Procedure:
First, we enumerate the codes and then we used the enumerated object to assign each observation to the cluster which it belongs to.
The cluster is a dictionary with key as the cluster number and value is a list with number corresponding to the row number in the observation data.

- **cost_neighbours(queue, ded_up, cost_coor, code_cost, P):**

1. queue:

The queue contains the costs associated with codes in a inverted multi-index

2. ded_up:

It is a set of tuples which contains the coordinates which have been visited.

3. cost_coor:

Cost_coor is a dictionary with cost as key and value as list of lists whose coordinates have the cost from the inverted multi-index.

4. code_cost:

Code_cost consists of cost associated with each query partition and its respective codebook. The cost is calculated and present in each column for every partition.

5. P:

P is the required number of partitions
The function returns the queue, ded_up, cost_coor and coordinates which are updated everytime.

Procedure:
The first value of the queue is taken and used as key. The value associated with the key is popped from the cost_coor and then assigned to coordinates to find its neighbours. If key has only one coordinate then the key is deleted from the cost_coor and the key is also popped from the queue. Else just the coordinates are obtained from the dictionary.
Then, we check if the **coordinates** are **present** in the **ded_up** and if not present, then we find the **number of neighbours** as **P** since only those neighbours have to be added to queue to **avoid violating** the **skyline principle**. Then the cal_cost method is used to calculate the cost of each neighbour and then added to the cost_coor dictionary and finally if a new key is added then the key is added to the queue and then queue is heapified and coordinates are added to the ded_up.

- **<u>cal_cost(new_coordinates, code_cost):</u>**

1. new_coordinates:

These are coordinates of each neighbour whose cost has to be calculated

2. code_cost:

Code_cost consists of cost associated with each query partition and its respective codebook. The cost is calculated and present in each column for every partition.

This returns the calculated cost for the coordinates associated with each code.

Procedure:
A variable cost is created to calculate the cost of each of coordinate from the code_cost with each element of coordinate being the row of code_cost and column being the partition of code_cost. If an out of bound coordinate element is obtained then IndexError might occur and hence to prevent that we used try and catch. In such a case -1 is returned as cost.

- **Conclusion**

We were able to implement both parts within the time stipulated for them.
Part 2 was more challenging given the time limit was one second. Using sets reduced the time to a fraction.

References

1. http://sites.skoltech.ru/app/data/uploads/sites/25/2014/12/TPAMI14.pdf
2. Vector Quantization Part-1 https://www.youtube.com/watch?v=trEDVj9M6Ng
3. Vector Quantization Part-2 https://www.youtube.com/watch?v=eyWMLmC-9R4
4. Product quantization for nearest neighbour search Herve J´egou, Matthijs Douze, Cordelia Schmid - https://lear.inrialpes.fr/pubs/2011/JDS11/jegou_searching_with_quantization.pdf
5. Optimized Product Quantization - http://kaiminghe.com/cvpr13/index.html
6. Product Quantizers - https://mccormickml.com/2017/10/13/product-quantizer-tutorial-part-1/
7. Representative point of a cluster https://stats.stackexchange.com/questions/67742/representative-point-of-a-cluster-with-l1-distance