COMP9318-2020T1

Data Warehousing and Data Mining

Project Report

Student:

Chirag Panikkasseril Unni - z5241855

Bharath Naga Chandra Surampudi - z5251342

Introduction

The project aims to introduce concepts of handling search of huge dataset queries in an efficient manner. We take speed, efficiency, scalability, and memory usage into consideration when implementing such algorithms. Here we will be applying a modified version of k-means clustering algorithm as part of the product quantization and use a query search using inverted index concept to retrieve the nearest neighbors.

Part1:

PQ for L1 Distance

In this question, we will implement the product quantization method with L1 distance as the distance function. The pq() function takes the input datapoints, initial centroid points and the max iteration for the k-means*(modified k-means) algorithm. The pq function first dimensionally reduce the datapoint by P value and utilize k-means* over each of the partition. This is added to a NumPy array of codebook and codes of P partition and returned as the output for Part1 question.

We compute the clustering algorithm as usual and for the distance computation we make the following changes. In order to get the L1 norm or 'cityblock 'computation of the distance instead of usual 'Euclidian' or L2 norm we apply the following:

```
# Measure the distance to every centroid location in codebook
distances = cdist(data, codebook_curr, 'cityblock')
# Assign all training data to closest centroid in codes
codes = np.argmin(distances, axis=1)
```

the cdist function from SciPy is applied for the distance calculation over the codebooks or the centroid locations and then it is fed to NumPy argmin to assign datapoints to the closest clusters and then update our codes.

As we made the distance computation in L1 norm we need to make few modifications to get the convergence of our k-means* algorithm. This is obtained by computing the new centroid locations in codebook by updating the centroid values over median location of each datapoints instead of mean value.

```
for c in range(K):
      points = data[codes == c]
      if points.size != 0:
            codebook_curr[c] = np.median(points, axis=0)
```

Here we can see that we ignore those empty clusters when computing the median and retain the existing value if empty. The codebook thus obtained is checked for convergence by calculating the difference between previous codebook for early termination.

```
error = np.linalg.norm(codebook_curr - codebook_prev) / np.sqrt(N)
```

Once the loop ends, we update the codes one last time to get the updated codes against the last calculated codebook. The codebook and codes are returned as our K-means* function output.

Part2:

Query using Inverted Multi-index with L1 Distance

The query function is used to obtain T nearest candidates for the given query against the P partitioned codebooks and codes. The first step is to partition the query into P parts and compute the q(0) vs U and q(1) vs V and so on and so forth where U,V are the corresponding part codebooks. Which gives you the sorted distance from query to each centroid with respect to the corresponding partition.

```
# split queries into P parts
qparts = np.stack(np.hsplit(queries, P), axis=1)

# Compute query vs codebook distance table
pqTable = pqTable_func(query)
```

We apply a min heap computation algorithm using *heapq* module over list h to incrementally find the nearest distant neighbor centroid location. We store their corresponding (i,j...) dimensional coordinates which will be mapped with the codes for the corresponding *label* to find the given T candidate set for each query. Although we do this we make sure not to search for already searched neighbors by tracking it in a *trav* dict and remove item from it when completely processed.

```
# dict to keep track of traversed
trav = {}

trav[tuple(neigh)] = True

trav.pop(tuple(q_i)) #delete when processed save memory
```

Following optimization has been done on the algorithm 3.1 to handle following scenarios.

1)In order to get the coordinates for dimensions where P>2 we utilize a list of length P which can be translated to i,j..., so on and so forth for our nearest distance computation in min heap.

```
# initialize  an array to uses as Index lookup table for P size
q_i = np.asarray([0 for _ in range(P)],dtype='uint8')
```

And in our min heap object we use an Identity matrix Id of P x P size to do traversal to the nearest neighbors and push nearest distant elements to the heap. This iteration will be done P times.

```
# PxP Identity Matrix
Id = np.identity(P,dtype=bool)


# Heapifying and traversing in P dimension
for p in range(P):
# Using identity matrix to search for nearest distant neighbor
      neigh=np.add(q_i, Id[p])
      if q_i[p] < K - 1 and not trav.get(tuple(neigh)):
            heappush(h, (getvalue(tuple(neigh), pqTable)))
            trav[tuple(neigh)] = True
```

2)In order to efficiently handle the codes from part1 we can utilize Python dict() *indexdict* and create an inverted index for the data vectors and can be reused for each query and use set() *Cset* as it will not add the duplicate candidates from the codes. Therefore, can be used to fill the candidate set. The final output will have the list of all candidate set for every query passed in *CandidateList* .

```
# Create inverted Index
indexdict = createIndex(codes)


# run query search for each query vector
for q in range(nQ):
      # Run query search and store each of the candidate set
      Cset = querysearch(qparts[q])
      CandidateList.append(Cset)
```

Conclusion

The project concludes that we can use the product quantization method and inverted index search query to store and retrieve massive amount of data with efficient and fast computation with optimal accuracy. We can easily scale the algorithm up and still achieve the results in less space time cost.

References

1. H. Jégou, M. Douze and C. Schmid, "Product Quantization for Nearest Neighbor Search," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 33, no. 1, pp. 117-128, Jan. 2011.

2. A. Babenko and V. Lempitsky, "The inverted multi-index," 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, 2012, pp. 3069-3076.