BITS Pilani, Hyderabad Campus

CS F415: DATA MINING

SECOND SEMESTER 2019-20

Assignment for Graduating Students

BIRCH Algorithm Implementation

Aekansh
2016A7PS0127H

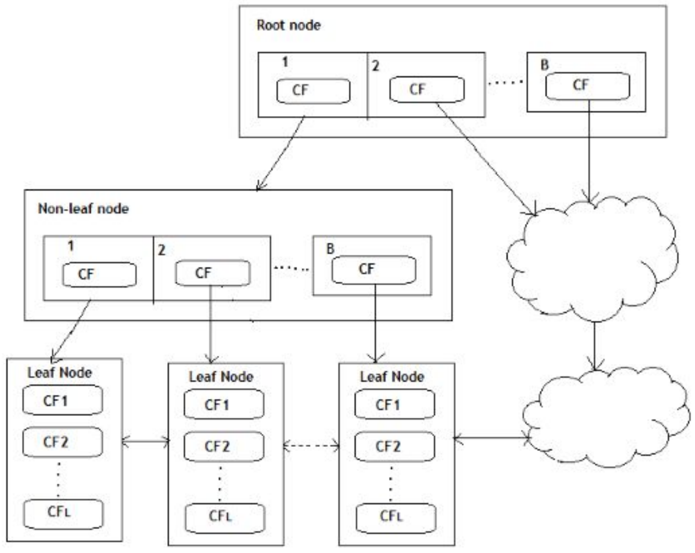
Table of Contents

1. Introduction

BIRCH stands for **Balanced Iterative Reducing and Clustering Using Hierarchies**, which uses hierarchical methods to cluster and reduce data.
The BIRCH algorithm uses a tree structure to create a cluster. It is generally called the Clustering Feature Tree (CF Tree). Each node of this tree is composed of several Clustering features (CF).

CFs of internal nodes have pointers to child nodes, and all leaf nodes are linked by a doubly linked list. In the clustering feature tree, a clustering feature (CF) is defined as follows: Each CF is a triplet, which can be represented by (N, LS, SS).

- Where N represents the number of sample points in the CF, which is easy to understand
- LS represents the vector sum of the feature dimensions of the sample points in the CF
- SS represents the square of the feature dimensions of the sample points in the CF.



An advantage of BIRCH is its ability to incrementally and dynamically cluster incoming, multi-dimensional metric data points in an attempt to produce the best quality clustering for a given set of resources (memory and time constraints). In most cases, BIRCH only requires a single scan of the database.

2. Data preprocessing

The dataset is provided in the form of 25 .arff files. Each file contains coordinates either in the form of tuples (x,y) or (a0,a1).

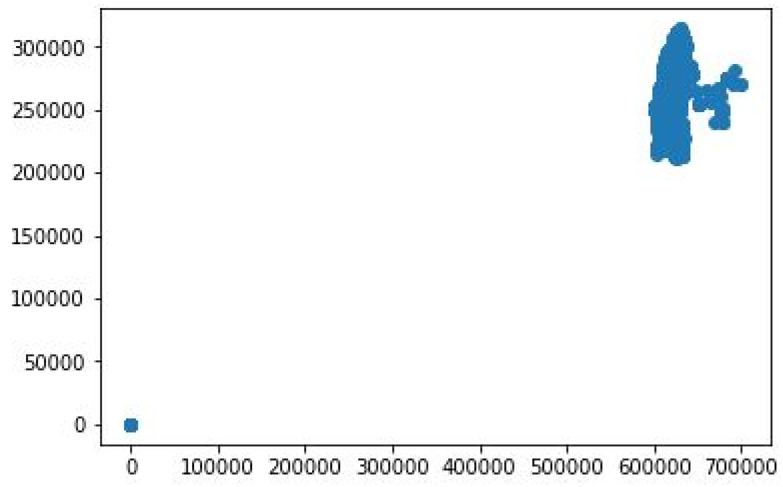
The first step was to convert them into a uniform coordinate system of (x,y).
After loading all the files into a dataframe there were a total of 363656 points.

There were many duplicate points present and as the running time of the algorithm depends on the total number of points duplicate points had to be removed. Alsp BIRCH is sensitive to duplicate points and a drop in number of final clusters was observed. (from 8670 to 8387)
There were a total of 358080 points after this step.

The dataset varied from (-9.98511, -28.8354322536275) to (697835.0,314328.0)

However the data points were not evenly spread. The whole dataset was divided in two big clusters one had points close to (0,0) and other had points close to the upper boundary of (697835.0,314328.0).

This is clearly visible in the plot of the preprocessed data:



3. Implementation of algorithm

The first step is to create the CF trees. These CF trees remove stress from the memory as we only have to load pointers to the memory.

```
Algorithm 1 BIRCH algorithm
Input: $M$ data points
Output: $N$ subclusters
  repeat
      Calculate the cluster feature $\vec{CF}$ of data points
      Insert record to construct CF tree
  until Insert all records into the CF tree
```

After creating the CF Trees we have to condense into desirable range and reduce the CF Tree. Then we have to form global clusters and refine the clusters.

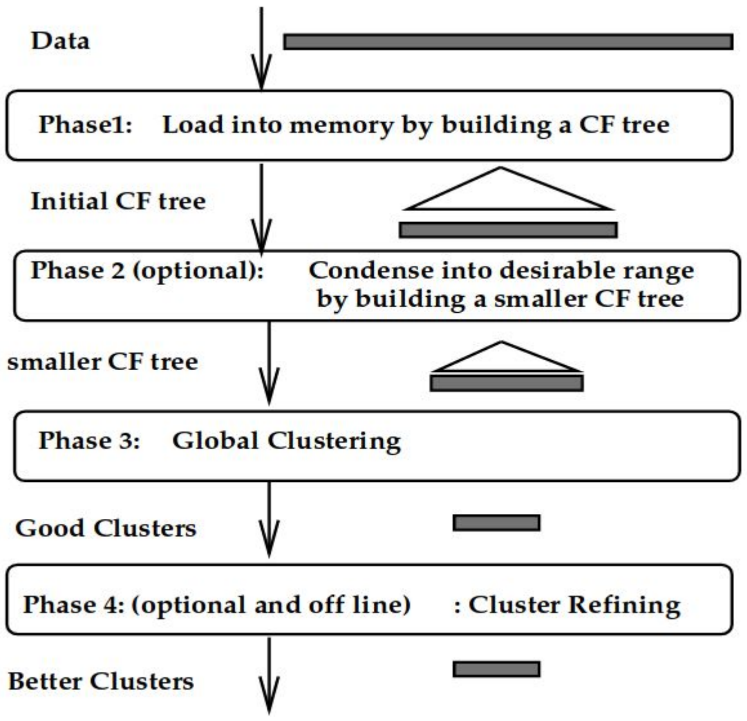
- Centroid: $\vec{C} = \frac{\sum_{i=1}^{N} \vec{X_i}}{N} = \frac{\vec{LS}}{N}$

- Average Linkage Distance between clusters

$$\sqrt{\frac{N_1 \cdot SS_2 + N_2 \cdot SS_1 - 2 \cdot \vec{LS_1} \cdot \vec{LS_2}}{N_1 \cdot N_2}}$$

- Radius: $R = \sqrt{\frac{\sum_{i=1}^{N} (\vec{X_i} - \vec{C})^2}{N}}$

Figure 2. BIRCH Overview


Data
Phase1: Load into memory by building a CF tree
Initial CF tree
Phase 2 (optional): Condense into desirable range by building a smaller CF tree
smaller CF tree
Phase 3: Global Clustering
Good Clusters
Phase 4: (optional and off line) : Cluster Refining
Better Clusters

4. Problems faced

During formation of CF trees I faced many memory errors during matrix multiplication. The points in the dataset have a high precision and the multiplication matrices required a lot of memory which my computer could not handle.

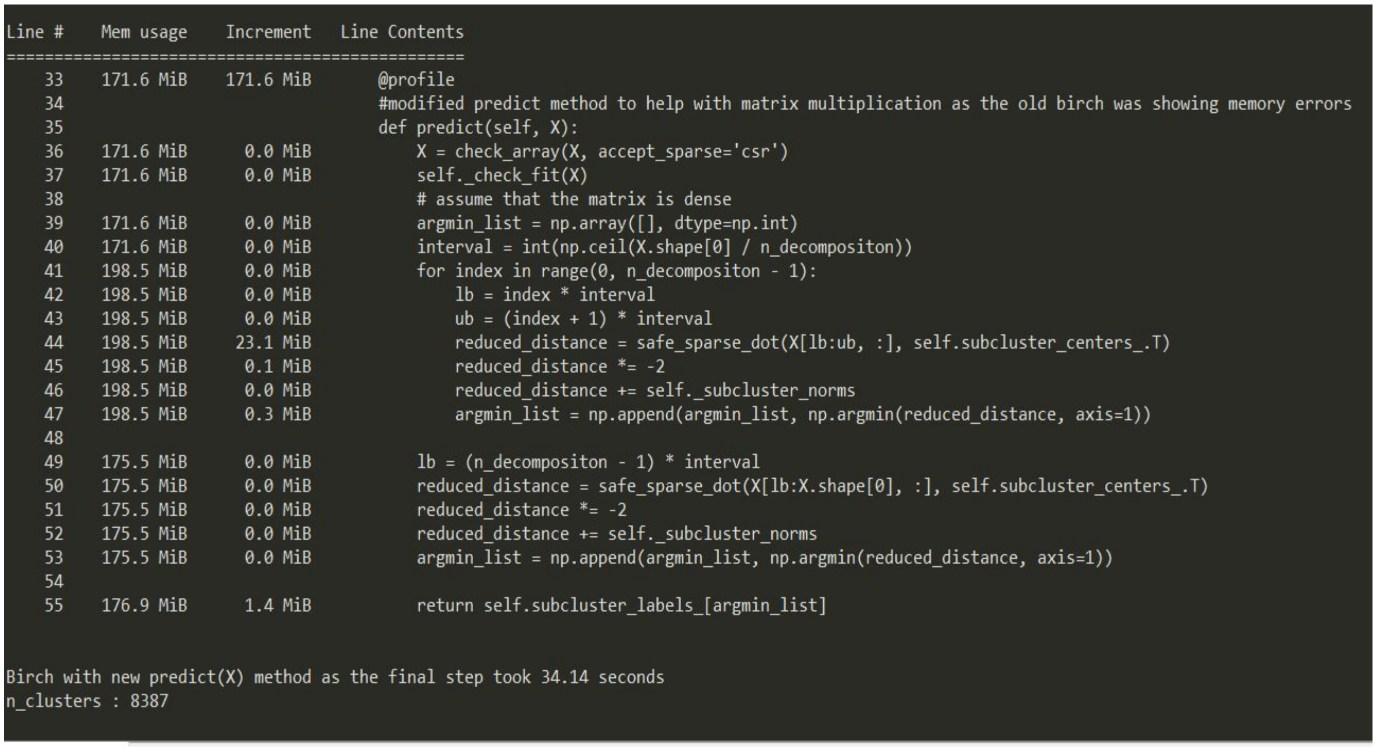
5. Solution

After researching and looking for solutions I found two methods

1) Chunked Birch algorithm : as data might not be evenly distributed it distributes the data points into several ‘chunks’ this greatly reduces the running time and memory use of the algorithm
2) New Birch algorithm : It uses less memory but takes a little longer than conventional birch algorithm. It is similar to the conventional Birch algorithm but it has a modified predict() method to handle large matrix multiplications from causing memory errors.

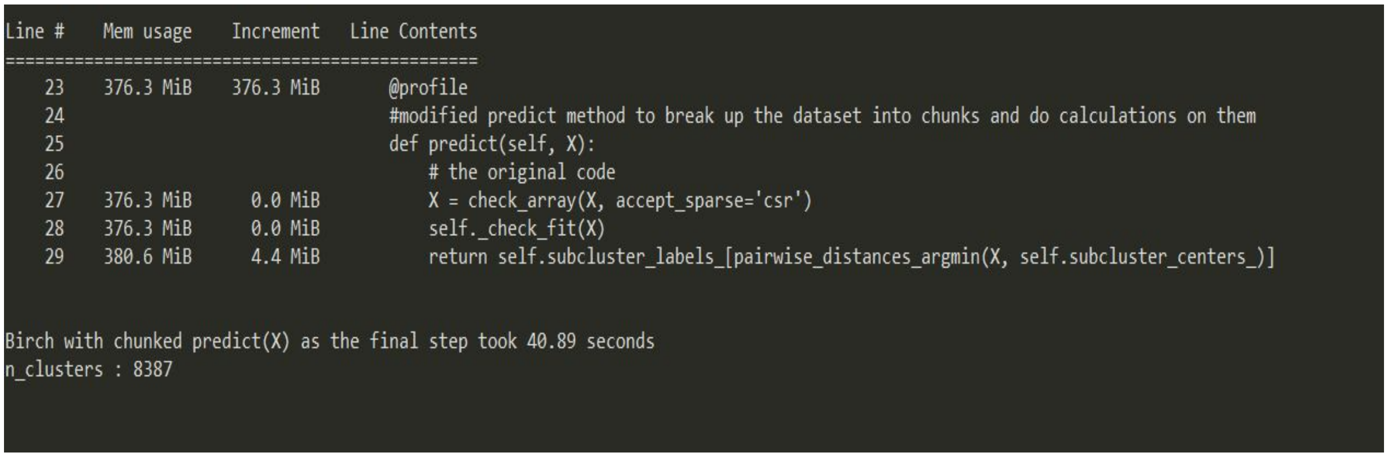
6. Results

- The New Birch Algorithm produced 8387clusters in 59.31 seconds.

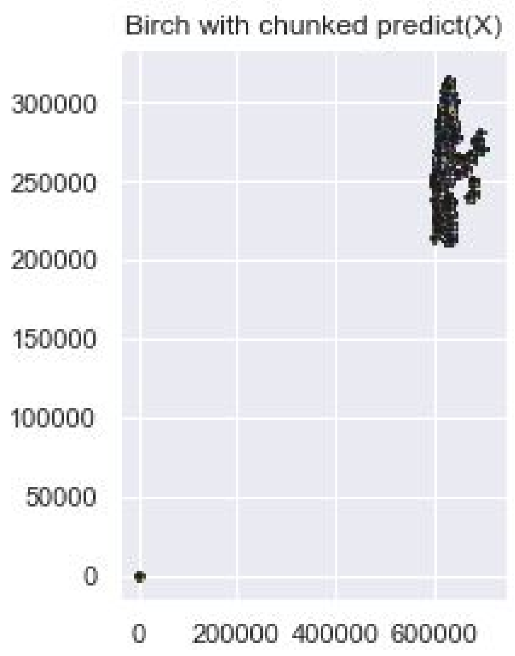


```
Line #    Mem usage    Increment   Line Contents
================================================
    33    171.6 MiB    171.6 MiB       @profile
    34                                 #modified predict method to help with matrix multiplication as the old birch was showing memory errors
    35                                 def predict(self, X):
    36    171.6 MiB      0.0 MiB           X = check_array(X, accept_sparse='csr')
    37    171.6 MiB      0.0 MiB           self._check_fit(X)
    38                                     # assume that the matrix is dense
    39    171.6 MiB      0.0 MiB           argmin_list = np.array([], dtype=np.int)
    40    171.6 MiB      0.0 MiB           interval = int(np.ceil(X.shape[0] / n_decompositon))
    41    198.5 MiB      0.0 MiB           for index in range(0, n_decompositon - 1):
    42    198.5 MiB      0.0 MiB               lb = index * interval
    43    198.5 MiB      0.0 MiB               ub = (index + 1) * interval
    44    198.5 MiB     23.1 MiB               reduced_distance = safe_sparse_dot(X[lb:ub, :], self.subcluster_centers_.T)
    45    198.5 MiB      0.1 MiB               reduced_distance *= -2
    46    198.5 MiB      0.0 MiB               reduced_distance += self._subcluster_norms
    47    198.5 MiB      0.3 MiB               argmin_list = np.append(argmin_list, np.argmin(reduced_distance, axis=1))
    48
    49    175.5 MiB      0.0 MiB           lb = (n_decompositon - 1) * interval
    50    175.5 MiB      0.0 MiB           reduced_distance = safe_sparse_dot(X[lb:X.shape[0], :], self.subcluster_centers_.T)
    51    175.5 MiB      0.0 MiB           reduced_distance *= -2
    52    175.5 MiB      0.0 MiB           reduced_distance += self._subcluster_norms
    53    175.5 MiB      0.0 MiB           argmin_list = np.append(argmin_list, np.argmin(reduced_distance, axis=1))
    54
    55    176.9 MiB      1.4 MiB           return self.subcluster_labels_[argmin_list]


Birch with new predict(X) method as the final step took 34.14 seconds
n_clusters : 8387
```

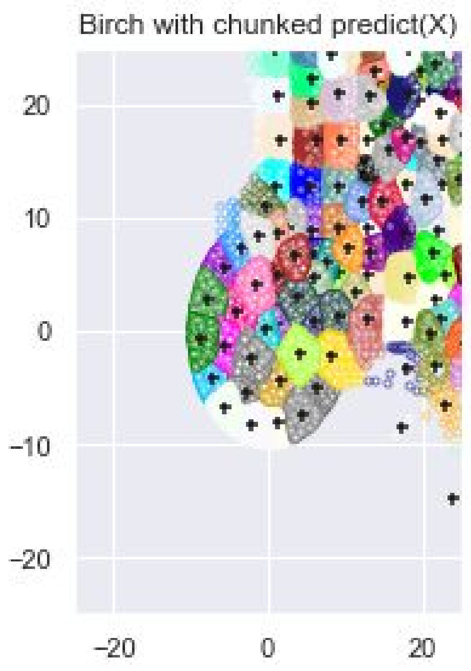
- Chunked Birch Algorithm produced 8387 clusters in 37.16 seconds.



```
Line #    Mem usage    Increment   Line Contents
================================================
    23    376.3 MiB    376.3 MiB       @profile
    24                                 #modified predict method to break up the dataset into chunks and do calculations on them
    25                                 def predict(self, X):
    26                                     # the original code
    27    376.3 MiB      0.0 MiB           X = check_array(X, accept_sparse='csr')
    28    376.3 MiB      0.0 MiB           self._check_fit(X)
    29    380.6 MiB      4.4 MiB           return self.subcluster_labels_[pairwise_distances_argmin(X, self.subcluster_centers_)]


Birch with chunked predict(X) as the final step took 40.89 seconds
n_clusters : 8387
```



(over all dataset)



(zoomed in on points close to 0,0)

Both algorithms produce the same clusters.

Github Repository link:

https://github.com/FaceTheAce/DM_FinalAssign_Aekansh_2016A7PS0127H