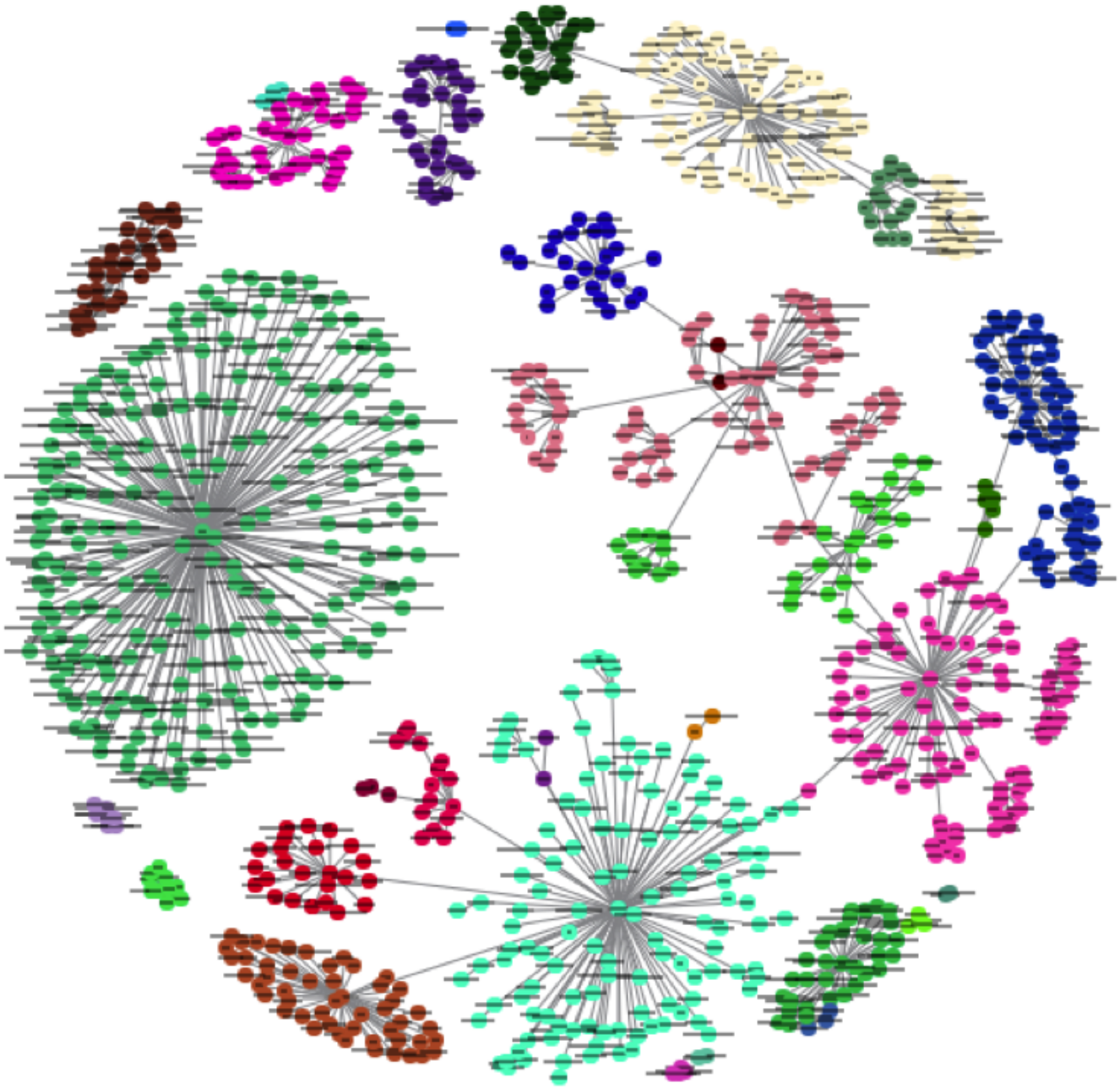
Cory Maklin Follow
Jul 1, 2019 · 6 min read ★ · Listen



BIRCH Clustering Algorithm Example In Python

Existing data clustering methods do not adequately address the problem of processing large datasets with a limited amount of resources (i.e. memory and cpu cycles). In consequence, as the dataset size increases, they scale poorly in terms of running time, and result quality. At a high level, **Balanced Iterative Reducing and Clustering using Hierarchies**, or **BIRCH** for short, deals with large datasets by first generating a more compact summary that retains as much distribution information

as possible, and then clustering the data summary instead of the original dataset. BIRCH actually complements other clustering algorithms by virtue if the fact that different clustering algorithms can be applied to the summary produced by BIRCH. BIRCH can only deal with metric attributes (similar to the kind of features KMEANS can handle). A metric attribute is one whose values can be represented by explicit coordinates in an Euclidean space (no categorical variables).

Clustering Feature (CF)

BIRCH attempts to minimize the memory requirements of large datasets by summarizing the information contained in dense regions as Clustering Feature (CF) entries.

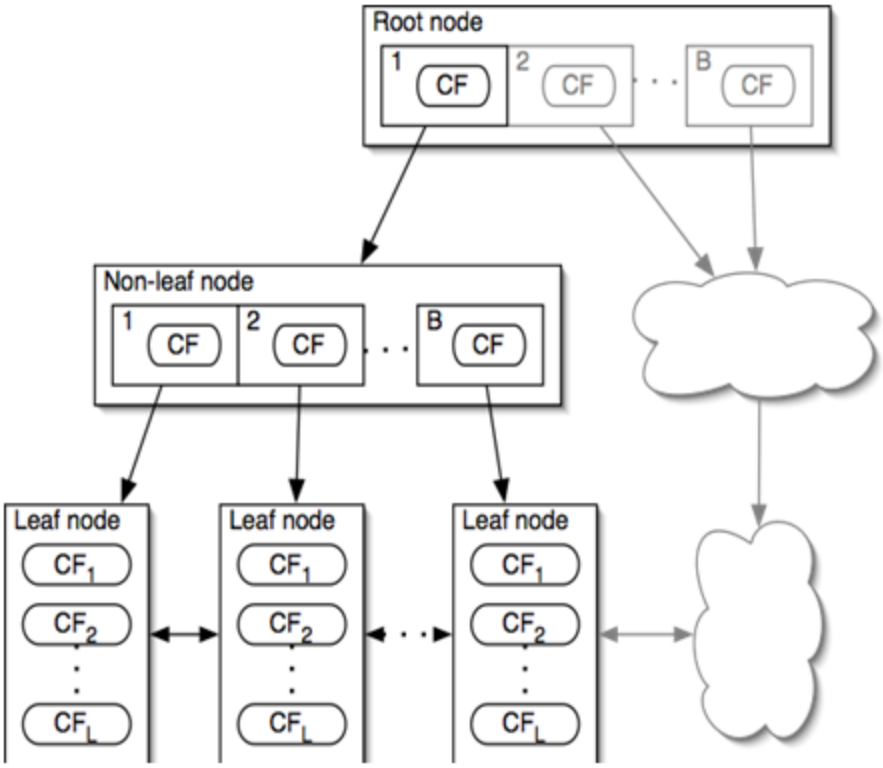
> **CF Definition** : *Given N d-dimensional data points in a cluster: $\{\vec{X_i}\}$ where $i = 1, 2, ..., N$, the* **Clustering Feature (CF)** *entry of the cluster is defined as a triple:* $\mathbf{CF} = (N, \vec{LS}, SS)$, *where N is the number of data points in the cluster, $\vec{LS}$ is the linear sum of the N data points, i.e., $\sum_{i=1}^{N} \vec{X_i}$, and SS is the square sum of the N data points, i.e., $\sum_{i=1}^{N} \vec{X_i}^2$.*

As we're about to see, it's possible to have CFs composed of other CFs. In this case, the subcluster is equal to the sum of the CFs.

> **CF Additivity Theorem** : *Assume that* $\mathbf{CF_1} = (N_1, \vec{LS}_1, SS_1)$, *and* $\mathbf{CF_2} = (N_2, \vec{LS}_2, SS_2)$ *are the CF entries of two disjoint subclusters. Then the CF entry of the subcluster that is formed by merging the two disjoint subclusters is:*
>
> $$\mathbf{CF_1} + \mathbf{CF_2} = (N_1 + N_2, \vec{LS}_1 + \vec{LS}_2, SS_1 + SS_2) \quad (11)$$

CF-tree

The CF-tree is a very compact representation of the dataset because each entry in a leaf node is not a single data point but a subcluster. Each nonleaf node contains at most ***B*** entries. In this context, a single entry contains a pointer to a child node and a CF made up of the sum of the CFs in the child (subclusters of subclusters). On the other hand, a leaf node contains at most ***L*** entries, and each entry is a CF (subclusters of data points). All entries in a leaf node must satisfy a threshold requirement. That is to say, the diameter of each leaf entry has to be less than ***Threshold***. In addition, every leaf node has two pointers, *prev* and *next*, which are used to chain all leaf nodes together for efficient scans.



Insertion Algorithm

Let's describe how we'd go about inserting a CF entry (a single data point or subcluster) into a CF-tree.

1. **Identify the appropriate leaf:** Starting from the root, recursively descend the DF-tree by choosing the closest child node according to the chosen distance metric (i.e. euclidean distance).

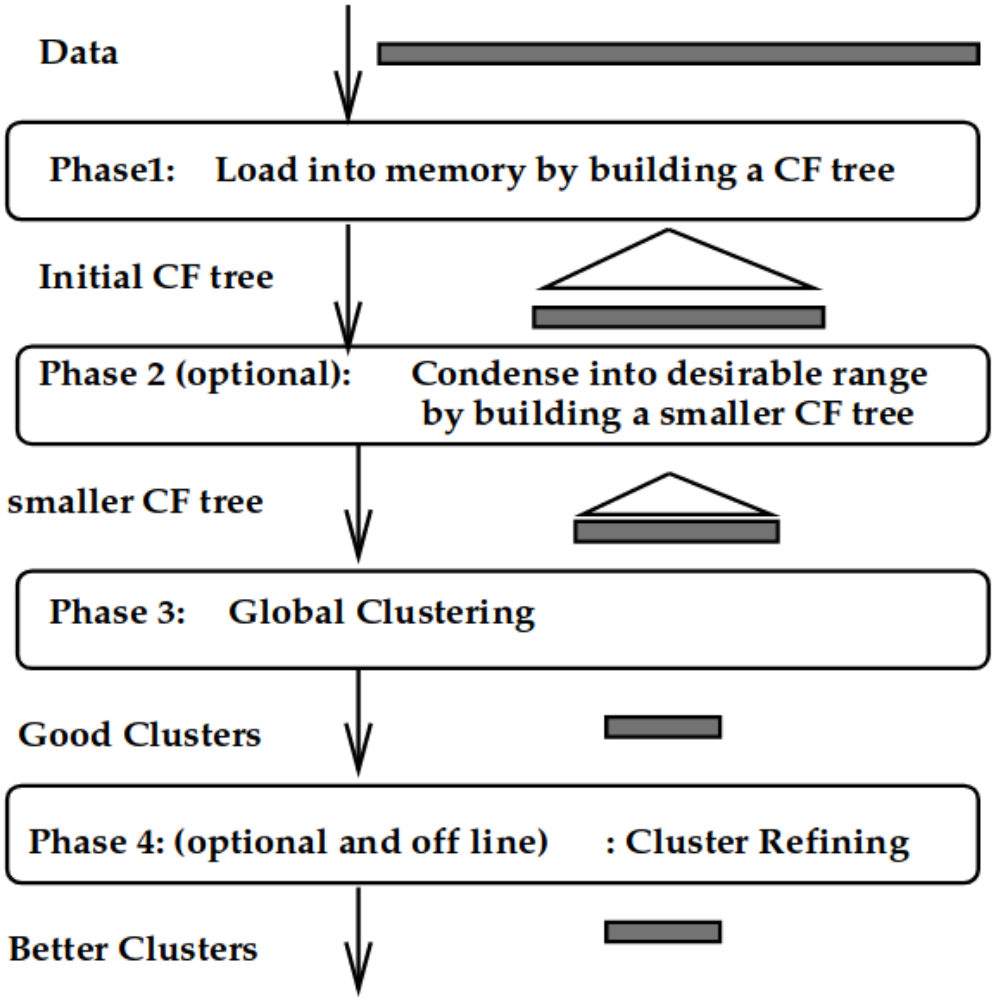
2. **Modify the leaf:** Upon reaching a leaf node, find the closest entry and test whether it can *absorb* the CF entry without violating the threshold condition. If it can, update the CF entry, otherwise, add a new CF entry to the leaf. If there isn't enough space on the leaf for this new entry to fit in, then we must split the leaf node. Node splitting is done by choosing the two entries that are farthest apart as seeds and redistributing the remaining entries based on distance.

3. **Modify the path to the leaf:** Recall how every nonleaf node is itself a CF composed of the CFs of all its children. Therefore, after inserting a CF entry into a leaf, we update the CF information for each nonleaf entry on the path to the leaf. In the event of a split, we must insert a new nonleaf entry into the parent node and have it point to the newly formed leaf. If according to ***B***, the parent doesn't have enough room, then we must split the parent as well, and so on up to the root.

Clustering Algorithm

Now that we've covered some of the concepts underlying BIRCH, let's walk through how the algorithm works.

Figure 2. BIRCH Overview

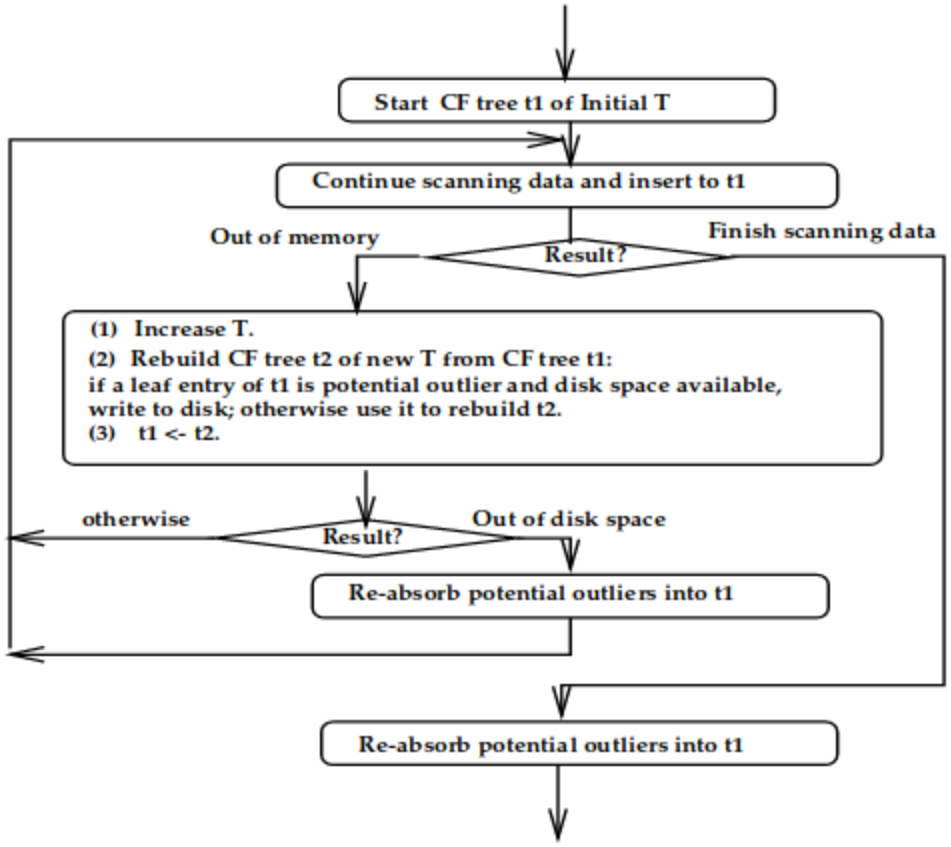


Phase 1

The algorithm starts with an initial threshold value, scans the data, and inserts points into the tree. If it runs out of memory before it finishes scanning the data, it increases the threshold value, and rebuilds a new, smaller CF-tree, by re-inserting the leaf entries of the old CF-tree into the new CF-tree. After all the old leaf entries have been re-inserted, the scanning of the data and insertion into the new CF-tree is resumed from the point at which it was interrupted.

A good choice of threshold value can greatly reduce the number of rebuilds. However, if the initial threshold is too high, we will obtain a less detailed CF-tree than is feasible with the available memory.

Figure 3. Flow Chart of Phase 1



http://www.cs.uvm.edu/~xwu/kdd/BIRCH.pdf

Optionally, we can allocate a fixed amount of disk space for handling outliers. Outliers are leaf entries of low density that are judged to be unimportant with respect to the overall clustering pattern. When we rebuild the CF-tree by reinserting the old leaf entries, the size of the new CF-tree is reduced in two ways. First, we increase the threshold value, thereby allowing each leaf entry to absorb more points. Second, we treat some leaf entries as potential outliers and write them out to disk. An old leaf entry is considered a potential outlier if it has far fewer data points than average. An increase in the threshold value or a change in the distribution in response to the new data could well mean that the potential outlier no longer qualifies as an outlier. In consequence, the potential outliers are scanned to check if they can be re-absorbed in the tree without causing the tree to grow in size.

Phase 2

Given that certain clustering algorithms perform best when the number of objects is within a certain range, we can group crowded subclusters into larger ones resulting in an overall smaller CF-tree.

Phase 3

Almost any clustering algorithm can be adapted to categorize Clustering Features instead of data points. For instance, we could use KMEANS to categorize our data, all the while deriving the benefits from BIRCH (i.e. minimize I/O operations).

Phase 4

Up until now, although the tree may have been rebuilt multiple times, the original data has only been scanned once. Phase 4 involves additional passes over the data to correct inaccuracies caused by the fact that the clustering algorithm is applied to a coarse summary of the data. Phase 4 also provides us with the option of discarding outliers.

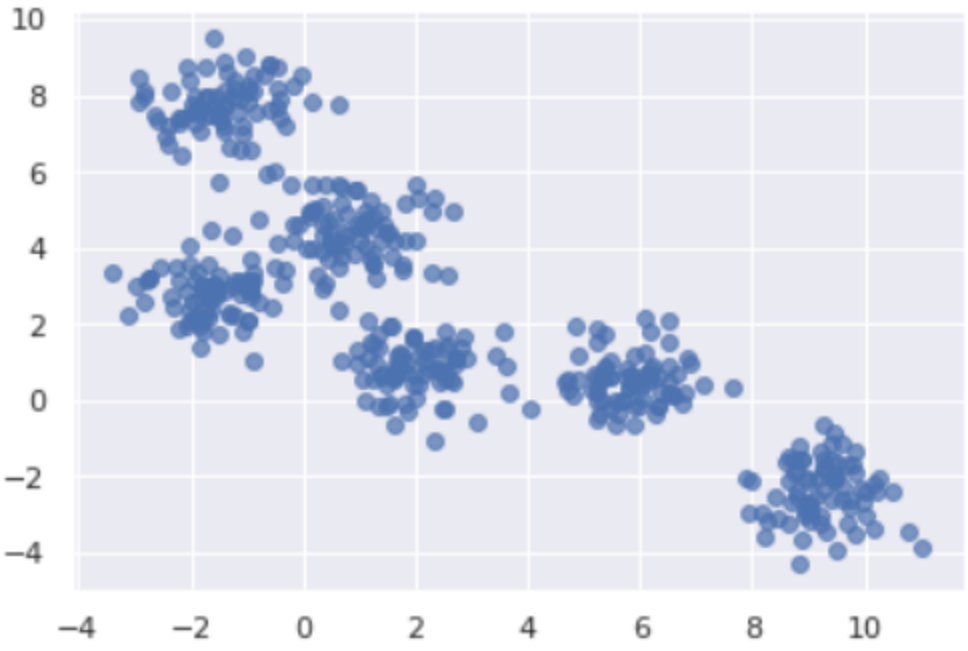
Code

Next, we'll implement BIRCH in Python.

```
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
sns.set()
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import Birch
```

We use `scikit-learn` to generate data with nicely defined clusters.

```
X, clusters = make_blobs(n_samples=450, centers=6, cluster_std=0.70,
random_state=0)
plt.scatter(X[:,0], X[:,1], alpha=0.7, edgecolors='b')
```



Next, we initialize and train our model, using the following parameters:

- **threshold:** The radius of the subcluster obtained by merging a new sample and the closest subcluster should be lesser than the threshold.
- **branching_factor:** Maximum number of CF subclusters in each node
- **n_clusters:** Number of clusters after the final clustering step, which treats the subclusters from the leaves as new samples. If set to `None`, the final clustering step is not performed and the subclusters are returned as they are.

```
brc = Birch(branching_factor=50, n_clusters=None, threshold=1.5)

brc.fit(X)
```

We use the `predict` method to obtain a list of points and their respective cluster.

```
labels = brc.predict(X)
```

Finally, we plot the data points using a different color for each cluster.

```
plt.scatter(X[:,0], X[:,1], c=labels, cmap='rainbow', alpha=0.7, edgecolors='b')
```

Final Thoughts

BIRCH provides a clustering method for very large datasets. It makes a large clustering problem plausible by concentrating on densely occupied regions, and creating a compact summary. BIRCH can work with any given amount of memory, and the I/O complexity is a little more than one scan of data. Other clustering algorithms can be applied to the subclusters produced by BIRCH.

Sources

1. http://www.cs.uvm.edu/~xwu/kdd/BIRCH.pdf