Web-Scale KMean clustering: Paper Review

By: Simon Bassey

Paper

Author:
D. Sculley Google, Inc. Pittsburgh. PA USA

Publisher: 2010

Source: web-scale kmean clustering

Summary of Core Idea

The lloyd's classic batch k-means clustering algorithm has a computation complexity of **O(kns)**

where

k=number of clusters, n = size of dataset, and

s=max. Number of non-zero elements in any one example in n]

And thus, does not scale for large datasets, to meet the near real-time latency requirement of under **1000 milliseconds** for most web applications.

Proposed Experiment

The paper proposed and implemented two approaches to faster clustering of large dataset:

1. **Mini-batch K-Means** clustering algorithm
   For the task of predicting clusters for new new examples, with a lower computation complexity cost of

   ~ O(Kbs) < **O(kns)** ,

   Where **b < n** is a batch size, **10x** in order of magnitude less than **n**.

Proposed Experiment ..

2. Sparse Cluster Centers

That further achieves better server resource utilization with compact storage of the model and reduces network costs, using an efficient projection-gradient method, which projects the centroids after update, to the nearest point within an L1-ball of some given radius **λ**.

Implementation Overview

Mini-batch K-Means

```
def train(self, Xs, bs):
    rand_indices = [random.randrange(len(Xs)) for i in range(self.__k)]
    centroids = Xs[rand_indices]

    clusters = [[] for i in range(self.__k)]
    prev_centroids =None
    iter_count = 0;
    while iter_count < self.__m_iter:
        iter_count +=1
        n_clusters = [[] for i in range(self.__k)]
        batch_Xs = Xs[np.random.choice(Xs.shape[0], bs, replace=True)]
        idx_x_c = np.empty(batch_Xs.shape[0], dtype=int)
        V = np.zeros(centroids.shape[0])

        for i,xi in enumerate(batch_Xs):
            eucl_dst = [np.linalg.norm(xi - centroid) for centroid in centroids]
            min_centr_idx = np.argmin(eucl_dst)
            idx_x_c[i] = min_centr_idx
        prev_centriods = centroids
        print(idx_x_c)
        for j, x in enumerate(batch_Xs):
          V[idx_x_c[j]] +=1
          lr = 1.0/V[idx_x_c[j]]
          centroids[idx_x_c[j]] = (1.0 - lr) * centroids[idx_x_c[j]] + lr*x
          n_clusters[idx_x_c[j]].append(xi.tolist())
        optimised = True
        for c in range(self.__k):
            if np.sum((centroids[c]-prev_centriods[c])/prev_centriods[c] *100) > 0.001:
                optimised = False
        if optimised:
            break
    self.centroids = centroids
    return iter_count,centroids,n_clusters,
```

Train with sparse cluster update

<Not Implemented>

Experiment Results

The resulting mini-batch implementation trained on the RCV1 collection of documents dataset of >800K examples, and tested on >23k test data points achieved:

1. **Great Accuracy -**
   An accuracy slightly lower than the classic lloyd's batch k-means clustering algorithm, but beat the SGD and triangular inequality variants of the k-means clustering algorithm.

2. **Lower computational cost -**
   Very importantly, the implementation had lower computational cost in comparison as seen in the next slide, suitable form most web Applications.

Suggested improvements

As the paper is relatively old, a number of faster implementations have been proposed over the years, even then new implementations looking to use the recommendations for m the paper for fast k-means may need to keep in that:

1. Faster initialization algorithms for choosing the initial centroids for clusters exist today and could be explored.
2. For very large dataset on multi-core machines, the task of choosing the closest centroid for all examples in a batch as well as updates could be parallelized, thus reducing further the complexity to ~ O(ks) instead of O(kbs) as proposed.

3. Although not mentioned in the paper, early stopping could also be explored to stop centroid updates no longer improve whether or not the number of epoch iterations is reached.

Thank you for listening

Questions?