Mini-Batch K-Means Report

Operation

Mini-Batch vs Batch K-Means

The standard K-Means algorithm requires all the data to be clustered to be held in memory at one time.Each iteration involves calculating distance metrics between each row and each centroid with each iteration involving k x n distance calculations where k is the number of centroids and n is the number of observations in the dataset. For very large datasets, storing all the data in main memory may be impossible and even if it is possible, it will be computationally expensive. For this reason, Mini-Batch K-Means (MBK) is used as an alternative. Mini-batch K-Means only requires a relatively small percentage of the data to be held in main memory at any given time. This allows the algorithm to scale very well as datasets increase in size. For user-facing web clustering applications such as News Aggregation, MBK provides fast, high quality clusters [1]. In the case of malware detection in android devices, MKB has outperformed K-means in terms of accuracy and time taken. [2]

How it Works

The MBK algorithm proposed by D. Sculley, first selects k random centroids (as set by the user) from the dataset [1]. It then selects a random sample of rows of size b. The square of the Euclidean distance between each row in the sample and each centroid is calculated and the nearest centroid is found for each row. Hence, for each centroid, there is a set of rows that form its cluster. These clusters are then iterated through, to find the Learning Rate and the Gradient Step. These values update the centroids for each mini-batch so as to reduce the value of the Cost Function (See gradient step).

Gradient Step

The metric to assess the quality of a k-means clustering implementation is the cost function. The cost function is the sum of the squared differences between each row and its centroid for the whole dataset. It is one number that represents how well the centroids form clusters. The function (F) for a particular sample row (X_m) with an assigned centroid, $C_{i,}$ is written as follows.

$$F(C_{i,} X_m) = (X_m - C_i)^2$$ [1]

If we find the partial derivative with respect to the centroid for a particular iteration of the sum, we find the gradient for that particular row. The derivative is as follows.

$$\delta F / \delta C_i = 2(C_i - X_m)$$

In words, this derivative is the amount that the cost function, for this iteration, will increase for a unit increase in the centroid, assuming that the sample point remains constant (which it does). For simplicity the constant 2 will be ignored. By moving in the opposite direction, i.e. by changing the sign, we move towards a minimum value of the cost function. To implement this, we find the gradient, multiply it by a constant (see learning rate) and change the sign. For each row in the mini-batch, the centroid is updated.

$Ci \leftarrow Ci - \alpha(C_i - X_m)$ where α = learning rate

Learning Rate

To further optimise the process, a weight is assigned to the gradient of the cost function at a given row in the mini-batch. This weight is proportional to the learning rate. The learning rate is the inverse of the number of rows belonging to the centroid of that row. The larger the number of rows in the cluster, the lower the learning rate.This means that the more times the centroid has been adjusted, the smaller the increments. This, presumably, helps against “over-shooting” the minimum.

$\alpha = 1/N$ where N = number of rows belonging to the cluster for the given mini-batch

References

[1] D. Sculley. 2010. Web-scale k-means clustering. In Proceedings of the 19th international conference on World wide web (WWW '10). Association for Computing Machinery, New York, NY, USA, 1177–1178.

[2] A. Feizollah, N. B. Anuar, R. Salleh and F. Amalina, "Comparative study of k-means and mini batch k-means clustering algorithms in android malware detection using network traffic analysis," *2014 International Symposium on Biometrics and Security Technologies (ISBAST)*, Kuala Lumpur, Malaysia, 2014, pp. 193-197, doi: 10.1109/ISBAST.2014.7013120.

Source Code

```
import numpy as np

X = np.genfromtxt("clusteringData.csv", delimiter=',') # loading data

#####################################################
# Initialising parameters and arrays to hold values
#####################################################

b = 1000 # setting batch size

k=6 # setting number of clusters

data_clusters = np.zeros(shape=(1,len(X))) # intialising array to hold all clusters

t = 20 # setting num iterations

num_restarts = 20

distortions = np.zeros(shape=(1,num_restarts))

################################
# functions
################################

def create_centroids(data, k): # Picks k random rows from data
        indexes = np.random.choice(len(data), size=k, replace = False) # picks k distinct
random indexes
        centroids = data[indexes] # assigns centroids based on indexes
        return centroids

# creates a batch of size b
def create_batch(X, M_indexes): # takes a set of b random indexes and dataset
        M = X[M_indexes] # selects mini-batch of random rows from the data of size b
        return M

# Finds distance between a single query point and a data set with same number of columns
def calculate_distance(X,point):
        diffs=np.subtract(X,point)  # finds the difference by element between each
                        # row of data and a single query point
        squared_diffs=np.power(diffs,2) # Finds squared differences by row
        sum_squares=np.sum(squared_diffs, axis=1) # Sums up to find squared euclidian
distance
        return sum_squares

# Assigns centroids based on the minimum distance
# between each point in a dataset or mini-batch
def assign_centroids(X, C, k):
        distance_array = np.zeros(shape=(k,len(X))) # creating array to hold distances
        for i in range(k): # calculates distances through k centroids
        distance_array[i] = calculate_distance(X, C[i])

        cluster_indices = np.argmin(distance_array,axis=0)
        # selects the minimum distance for each point
        # to create an array of cluster indexes
        return cluster_indices


# Calculates gradient step for each iteration
def gradient_step(m_batch_clusters, M, C, N):
        for i in range(b): # loops through each point in the mini-batch
        cluster_index = m_batch_clusters[i] # saves sample to variable cluster_index
        centre = C[cluster_index] # finds centroid nearest the sample
        N[m_batch_clusters[i],] += 1 # updates the number of points belonging to the cluster
        N_sample = N[m_batch_clusters[i],]
        lr = 1/N_sample # calculates inverse for learning rate
        C[cluster_index] = (1 - lr)*centre + lr*M[i] # updates cluster according to gradient step
        return C

# finds sum of squared distance between each point and its centroid
# then add them up to assess the quality of the clusters
def distortion_cost(X,data_clusters,C):
        distortion=0 # intitialsing distortion value
        k=C.shape[0] # assigning a value to k to match the number of rows in the centroids
array
        for i in range(k): # loops through rows in centroid array
        diffs=np.subtract(X[data_clusters==i],C[i]) # find differences between each centroid
                                                    # and the data points in its cluster
        squareds=np.power(diffs,2) # squaring the difference
        sum_squared=np.sum(squareds) # summing up the total squared differences for
each centroid
        distortion+=sum_squared # iteratively adding up the squared differences for all
centroids
        return distortion

##########################################################################
#############
# Loops through the whole algorithm for each set number of different initial centroids
##########################################################################
#############

for j in range(num_restarts):
        N = np.zeros(shape=(k,1)) # Initialise array to hold the number of rows in a cluster for
a given restart
```

```
        C = create_centroids(X, k) # create k random centroids from data

        for i in range(t): # implementing the algorithm over t iterations
        M_indexes = np.random.choice(len(X), size=b, replace=False) # finding M distinct
choices
        M = create_batch(X, M_indexes) # create batch
        m_batch_clusters = assign_centroids(M, C, k) # assigning centroids to points in the
mini-batch
        C = gradient_step(m_batch_clusters, M, C, N) # updates centroids with gradient step

        # finding clusters for the whole dataset based on final
        # set of centroids for the restart
        data_clusters = assign_centroids(X, C, k)

        # Find value of cost function for the dataset for each restart
        distortions[0,j] = distortion_cost(X, data_clusters, C)


distortions
np.min(distortions[0])
```