University of Glasgow | School of Computing Science

Large-Scale Learning: Query-driven Machine Learning over Distributed Data

Kurt Portelli
Natascha Harth
Ruben Giaquinta
Xu Zhang
Monica Gandhi

Level M Team Project — 1 December 2015

Abstract

The abstract goes here

Education Use Consent

We hereby give our permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ____________________ Signature: ____________________

Name: ____________________ Signature: ____________________

Name: ____________________ Signature: ____________________

Name: ____________________ Signature: ____________________

Name: ____________________ Signature: ____________________

Name: ____________________ Signature: ____________________

Contents

Chapter 1

Introduction

Chapter 2

Related work

The general approach in learning a large multi-dimensional dataset is to investigate the dataset as a whole and estimate the probability density function. G.Cormode et al. in [**?**] describes the well established techniques used in generating an

Chapter 3

Design & Implementation

3.1 Clustering

3.1.1 Nearest Neighbour - Average Data

3.1.2 Offline K-Means

The Algorithm

Batch K-Means is the oldest and most simple clustering method; it is however very efficient. The algorithm, given a finite data set of d-dimensional vectors $X = \{x^t\}_{t=1}^N$ and k *centroids*, or *codebook vectors*, $m_j, j = 1, ..., k$, partitions the data set into k clusters in order to minimize the so called total *reconstruction error*, defined as follows:

$$E(\{m_i\}_{i=1}^k|X) = \sum_t \sum_i b_i^t \tag{3.1}$$

where

$$b_i^t = \begin{cases} 1 & if \parallel x^t - m_i \parallel = min_j \parallel x^t - m_j \parallel \\ 0 & otherwise. \end{cases} \tag{3.2}$$

Therefore, x^t is represented by m_i with an error proportional to the Euclidean distance $\parallel x^t - m_j \parallel$. The procedure starts initializing m_i randomly; at each iteration b_i^t is calculated for all x^t and m_i are updated according to the following rule:

$$m_i = \frac{\sum_t b_i^t x^t}{\sum_t b_i^t}. \tag{3.3}$$

The algorithm terminates if any of the *codebook vectors* m_i hasn't been changed during the update step. Upon termination the function returns the *codebook verctors*.

Implementation

The Batch K-Means was implemented in Java. The Cluster class has two objects, an *ArrayList* of *points* representing all the points belonging to the cluster, and a *centroid*, the *codebook vector*. The update function searches for the nearest *codebook vector*.

```
for (int i = 0; i < data.size(); i++) {

        double max = 0;
        int maxIndex = -1;
        for (int j = 0; j < Clusters.size(); j++) {
                double powsum = 0;
                for (int k = 0; k < data.get(0).length; k++) {
                        powsum += Math.pow(data.get(i)[k]
                                        - Clusters.get(j).getCentroid()[k], 2);
                }
                double temp = Math.sqrt(powsum);
                if (maxIndex == -1 || temp <= max) {
                        max = temp;
                        maxIndex = j;
                }
        }
        pointsclusters.set(i, maxIndex + 1);
        Clusters.get(maxIndex).getPoints().add(data.get(i));
}
```

At a later stage the method applies the update rule for each of the *codebook vectors*, counting the number of updated *centroids*.

```
for (int k = 0; k < Clusters.size(); k++) {

        double[] c_d = new double[data.get(0).length];
        for (int j = 0; j < c_d.length; j++) {
                c_d[j] = 0;
        }

        int points = Clusters.get(k).getPoints().size();

        for (int i = 0; i < points; i++) {
                for (int w = 0; w < c_d.length; w++) {
                        c_d[w] += Clusters.get(k).getPoints().get(i)[w];
                }
        }

        if (points > 0) {
                for (int w = 0; w < c_d.length; w++) {
                        c_d[w] /= points;
                }
        }

        double[] conditions = new double[c_d.length];

        for (int w = 0; w < c_d.length; w++) {
                conditions[w] = Math.abs(Clusters.get(k).getCentroid()[w]
                                - c_d[w]);
        }

        int condcounter = 0;
        for(int w=0;w<conditions.length;w++){
                if(conditions[w]<0.001)
                        condcounter++;
        }

        if (condcounter == c_d.length) {
                counter++;
        } else {
                for (int l = 0; l < c_d.length; l++) {
                        Clusters.get(k).getCentroid()[l] = c_d[l];
                }
        }
}
```

The function terminates if the value of the variable counting the number of modified centroids is equal to the number of clusters $counter == Clusters.size()$.

3.1.3 Online K-Means

The Algorithm

The Batch K-Means cannot, or at least not efficiently, deal with huge data sets. Storing a vast amount of data in internal memory can be a serious issue. In order to avoid this problem, Online K-Means does not store input data. Therefore, the algorithm initialize k random *codebook vectors* $m_j, j = 1, ..., k$ from the training set X. For all $x^t \in X$, randomly chosen, the update function computes:

$$i \longleftarrow argmin_j \parallel x^t - m_j \parallel \tag{3.4}$$

$$m_i \longleftarrow m_i + \eta(x^t - m_i) \tag{3.5}$$

until m_i converge.

Implementation

The Online K-means was implemented in Java as well. The update method is presented below:

```java
public Integer update(float[] point) {
        if (centroids.size() < k) {
                centroids.add(point);
                return centroids.size() - 1;
        } else {
                Integer nearestCentroid = Tools.classify(point, centroids);
                // Move centroid
                this.centroids.set(nearestCentroid, moveCentroid(point, nearestCentroid));

                return nearestCentroid;
        }
}

public float[] moveCentroid(float[] point, int nearestCentroid) {
        float[] update = Tools.subtract(point, this.centroids.get(nearestCentroid));
        update = Tools.multiply(update, alpha);
        return Tools.add(this.centroids.get(nearestCentroid), update);
}
```

The class Tools defines a set of multi dimensional operations like the Euclidean distance, addition, subtraction and multiplication, and finally a method to find the minimum value.

```java
public static float distance(float[] p1, float[] p2) {

        float dist = 0;
        for (int i = 0; i < p1.length; i++) {
                dist += (p1[i] - p2[i]) * (p1[i] - p2[i]);
        }
        return (float) Math.sqrt(dist);
}

public static int classify(float[] point, List<float[]> centroids) {
        float minDist = Float.MAX_VALUE;
        int ans = 0;
        for (int i = 0; i < centroids.size(); i++) {
                float tempDist = Tools.distance(point, centroids.get(i));
                if (tempDist < minDist) {
                        minDist = tempDist;
                        ans = i;
                }
        }
        return ans;
}
```

3.1.4 ART

3.1.5 Silhouette

3.2 Query Space Clustering

3.2.1 L interest points

3.2.2 Gaussian distribution

3.3 Prediction

3.3.1 Mapping Query clusters to data clusters

3.3.2 Learning algorithm

3.3.3 Prediction algorithm

Chapter 4

Evaluation

Chapter 5

Conclusion

5.1 Contributions