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**

We study a novel solution to executing aggregation queries more specifically AVERAGE queries over large scale-data. We investigate cases where the owners restrict data access such that only aggregation operators can be used. It can also be extended to scenarios where access to the data is limited due to cost or slowness. Using distance-based queries with aggregation operators we are able to gain insight on how to best cluster the underlying data. The useful information are the results derived from the aggregation queries which are then clustered based on the distance based queries allowing us to then be able to predict the results of new and unseen queries. We study this approach which is called query-driven machine learning and evaluate its performance.

## 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

With the enormous improvements in performance and price in both data storage devices and network infrastructure it is now very cheap to store data. Since such large amounts of data is now accessible this has created a need and opportunity for machine learning algorithms.[**?**] The challenge nowadays is not to store large amounts of data but to make it as accessible as possible. It is very difficult to query these datasets and return results interactively. According to L.Bottou and Y.Le Cunn[**?**] these technological improvements have outran the exponential evolution of computing power. Thus, we must rely more on learning algorithms to process these large amounts of data with comparatively less computing power. These algorithms are typically split into online and batch. Online algorithms quickly process large datasets by adjusting their parameters as fresh data is inputted. On the other hand batch algorithms keep iterating over the dataset to achieve the optimum solution. It is then argued that online outperform batch algorithms due to the fact they do not iterate over a dataset.[**?**]

In this work we are going to assume we are dealing with datasets which we don't have access to. In a real world scenario this can occur for a variety of reasons. It might be that the dataset is just too large to go through it, or the the third party REST API service that is being used has a cost for each query that is made. Another requirement might be that this third party company does not allow a copy of their data to be held. Thus, batch algorithms won't be able to iterate through the whole dataset or it might be too costly to do so.

We will be investigating the use of online clustering in machine learning with the aim to finally be able to predict the results of queries without running them on the dataset. We will also be using a query driven approach [**?**] which will allow us to only quantize the important areas inside the data space. This approach creates various subspaces of interest which are determined by a focal point in space and radius. The AVERAGE aggregation operator will be studied to gain an insight on how best to cluster the underlying dataset. The goal is to use the results of the queries issued to cluster the underlying data. Online clustering is used because these results represent a stream of infinite data which the clustering can learn over time.

Before going in detail about the training set generation, learning and prediction process, in the following section traditional algorithms and related work are going to be discussed to better compare our achievements.

# 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 [3] describes the well established techniques used in aggregate query processing. They mention histograms, self tuning histograms, sketches, sampling and wavelets. As C.Anagnostopoulos and P.Triantafillou argue in [2] these techniques assume that they have access to the actual data set, thus can store and preserve the statistical model created. For example to be kept up to date, histograms need to scan all the data. On the other hand Self-tuning histograms execute additional queries to adjust the statistical model accordingly.

GENHIST[**?**] is one of the variations of histograms with the same target, to find an approximate density function using a grid. GENHIST achieves this by iteratively split the dataset into regular grids and find the dense areas. In each iteration the density of each bucket with the surrounding buckets is smoothed. The innovation behind this is that in each iteration buckets may overlap thus, revealing new information and a more accurate density function. In each iteration buckets are removed which effect the number of iterations, for example a high value can result in losing important detail. Although the number of iterations is a constant number which depends on the parameters given this still scales directly with the size of the dataset. Each iteration involves doing one pass over the data and since the number of iterations is constant, the running time of the algorithm is constant.[**?**]

As the dataset changes over time the GENHIST algorithm has to be run again to update the probability density function. As the dataset increases in size, traditional histograms such as GENHIST fail to scale well due to the fact that they regularly need to be rebuilt to update the statistical model creating a substantial overhead. It is then noted that the statistical models created by histograms only consider the data distribution without taking into consideration the query pattern of users.[2] Thus, this is not suitable for what we want to achieve, as we are interested in a constructing a model that relies on the query distribution and data distribution. Self-tuning histograms (STH) were proposed to address this by using the cardinality of a query's result to adjust the statistical model. STH still have a fundamental limitation which is the necessity of reading all the dataset because it needs to calculate the probability density function. The use of wavelets, sketches and sampling are also discussed in [2] with the conclusion that they are not viable since they need to access the raw data to create and maintain their structures.

In [2] the query driven approach is discussed in detail and compared to the techniques mentioned

above. The query driven approach is very useful in the scenario where one does not have access to the data or it is very costly to access the data (maybe due to size, cost, location). The idea behind this approach is that a training set containing a list of queries with their corresponding output is given. After learning this training set the algorithm should be able to predict the output without running the query. Although the training set is extracted from the dataset it is independent from the size of the dataset. Thus the size of the dataset will not impact the performance and the quality of the prediction fully depends on the training set and prediction algorithm used.

C.Anagnostopoulos and P.Triantafillou[2] discuss how this training set can be manipulated to allow the algorithm to predict results from queries as fast and accurate as possible. It is accepted that new queries might not be found in the training set thus a way to identify how close a query is to another is to use euclidean distance. One can go through all the training set, find the closest training query and then return the result of that query. This solution would increase linearly on the size of the training set. But, some queries might be redundant since they are very close to other existing queries while others might be significantly more important since they define another whole separate user interest. This shows the importance to extract information from the query space and be able to find the interest areas. Thus, the solution would be to cluster similar queries into a smaller set of representative queries called $L$.

To arrive at the prediction stage each representative query is assigned a representative result. The representative results are continuously updated while learning and moved around the data space depending on the training set. If the training space is large enough the representative queries and results should converge to represent what is actually inside the raw data. The clear advantage is that the size of $L$ is smaller than the size of the training set which in turn is smaller than the raw data.[2]

Learning can easily be stopped and continued without the need to start from scratch. This approach makes prediction very fast since each new query is associated with a closest representative query and the representative result given. In case the actual result is known the prediction error can be calculated by checking the difference between the actual and predicted result.[2]

# 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*[1].

## 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 [1].

#### Implementation

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

```
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.

```
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

$$q = [x_1, x_2, \theta] = [\vec{x}, \theta] \tag{3.6}$$

$$\bar{x} = [\bar{x_1}, \bar{x_2}] = \frac{1}{n} \sum \vec{x_i} :\parallel \vec{x} - \vec{x_i} \parallel \leq \theta \tag{3.7}$$

$$Trainingsset = [q, \bar{x}] \tag{3.8}$$

$$w[j] = online\, k - means\, centroids\, for\, q \tag{3.9}$$

$$u[j] = online\, k - means\, centroids\, for\, \bar{x} \tag{3.10}$$

$$Predictionset = [w[j], u[j]] \tag{3.11}$$

$$\epsilon_i = \parallel \bar{x} - u[j] \parallel \tag{3.12}$$

$$Mean\, Error = \frac{\sum \epsilon_i}{i} \tag{3.13}$$

### 3.3.2 Learning algorithm

### 3.3.3 Prediction algorithm

# Chapter 4

# Evaluation

# Chapter 5

# Conclusion

We have have studied a novel solution to the problem of predictive analysis over distributed data. This query driven solution is able to abstract query similarity and cluster the underlying data. The query clusters are associated with their related underlying data. The results of new queries are predicted by using the most similar query cluster. We evaluated this solution by using an evaluation data set to confirm that the predicted results are similar to the actual results. The significance of this study lies on the fact that it can predict results with restricted access to the dataset. This is due to the how the online learning mechanism is implemented, the prediction and learning steps are independent to the dataset, thus offering a scale-out and decentralized solution.

## 5.1 Contributions

# Bibliography

[1] Ethem Alpaydn. Introduction to machine learning 2nd edition, 2010.

[2] Christos Anagnostopoulos and Peter Triantafillou. Learning to accurately count with query-driven predictive analytics, 2015.

[3] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(13):1–294, 2011.

[4] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In *In Proceedings of the International Conference on Very Large Databases*, pages 358–369, 1995.