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: Kurt Portelli Signature: K.Portelli

Name: Natascha Harth Signature: N.Harth

Name: Ruben Giaquinta Signature: R.Giaquinta

Name: Xu Zhang Signature: X.Zhang

Name: Monica Gandhi Signature: M.Gandhi

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.[4] 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[4] 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.[4]

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 [2] 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 [5] 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 [3] 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[6] 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.[6]

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.[3] 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 [3] with the conclusion that they are not viable since they need to access the raw data to create and maintain their structures.

In [3] 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[3] 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.[3]

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.[3]

Chapter 3

Design & Implementation

3.1 Clustering

The problem of cluster analysis consists in grouping a set of objects, the data set, in clusters (groups), according to similar features. Similarity among objects is mainly related to the concept of Euclidean distance. In this section some clustering algorithms will be presented.

3.1.1 Nearest Neighbour - Average Data

The Algorithm

The Nearest Neighbour algorithm is one of the simplest methods for classification. The algorithm, given a finite set of d-dimensional vectors $X = \{x^t\}_{t=1}^{N}$, each with a class label, and a defined constant k, partitions the space classifying each point $x \in X$ as the majority class between the k nearest neighbors. In order to find the k nearest neighbors, the function calculates for each $x \in X$ the Euclidean distance between x and x', $\forall x' \in X$.

Implementation

The method classify, implemented within the class *Tools* for Online K-Means, can be seen as a frame of the particular case of the Nearest Neighbour algorithm with $k = 1$. The function computes the Euclidean distance between a point and all the centroids calling the *distance* method; the index of the nearest centroid is returned in order to assign the point to its cluster.

```
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;
}
```

3.1.2 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 is 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 first k input stream points are added as centroids; at a later stage, the *classify* function is called in order to search for the nearest centroid and update it accordingly. The *moveCentroid* method is implemented according to the rule:

$$m_i \longleftarrow m_i + \eta(x^t - m_i). \tag{3.6}$$

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.

```
private Tools() {
        r = new Random();
}

public static Tools getInstance() {
        if (instance == null) {
                instance = new Tools();
        }
        return instance;
}

/**
 * Get the average result of the data from that query
 *
 * @param dataSet
 * @param query
 * @param theta
 *            search area next to query
 * @return
 */
```

3.1.4 ART

The Algorithm

Adaptive Resonance Theory (ART) is a competitive learning algorithm used in neural networks. The algorithm follows an incremental approach, initializing just one centroid and adding other centroids as needed. ART does not require the number of clusters to be specified, instead it requires a *vigilance* value in order to create new centroids.

In ART, initially the first input point is chosen as the centroid for the first cluster. When the Euclidean distance between the data point and its nearest centroid is less than the vigilance, then the update is calculated as in Online K-Means. However, if the distance is greater than the vigilance, then a new cluster is created with that point as a centroid.

For the data set $X = \left\{x^t\right\}_{t=1}^{N}$, the following equations are performed for each update:

$$b_i = \parallel m_i - x^t \parallel = \min_{l=1}^{k} \parallel m_l - x^t \tag{3.7}$$

$$\begin{cases} m_{k+1} \leftarrow x^t & if b_i > \rho \\ \triangle m_i = \eta \left(x^t - m_i \right) & otherwise \end{cases} \tag{3.8}$$

m_i is the initial cluster center, ρ is the vigilance value specified by the user and b_i is the minimum distance of the point to its nearest cluster center [1].

Implementation

The ART algorithm has been implemented in Java and includes three classes: *ART*, *Application* and *Tools*. The Application class requires the text file containing the data points, the vigilance value and the alpha learning value as input while the *Tools* class contains a set of multidimensional operations.

The ART class implements the *update* function according to the equations 3.7 and 3.8 for each input data point. The variable *row* is the user input vigilance value and *point* is the next incoming data point. *Tools.distance(point, centroids.get(nearestCentroid))* gives the minimum distance of the point to its nearest cluster center.

```
public Integer update(float[] point) {
        int nearestCentroid = Tools.classify(point, centroids);
        if (nearestCentroid == -1) {
                centroids.add(point);
                nearestCentroid = 0;
        } else {
                //check if the distance from nearest centroid is less than vigilance 'row'
                if (Tools.distance(point, centroids.get(nearestCentroid)) < row) {
                        // add point to the cluster
                        this.centroids.set(nearestCentroid, moveCentroid(point, nearestCentroid));

                } else {
                        //create new centroid
                        centroids.add(point);
                        nearestCentroid = centroids.size() - 1;
                }
        }
        return nearestCentroid;
}
```

If the distance between the point and the nearest centroid is less than the vigilance value, the method invokes the *moveCentroid* function which adds the point to the cluster and updates the centroid.

```
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 output is obtained in two different text files, one containing all the centroids of the cluster and the other file containing the cluster ids of each data point.

3.1.5 Silhouette

The Algorithm

Silhouette is an evaluation method used to determine how well a data point lies within its cluster. This method is used to validate the consistency and strength of a cluster. The Silhouette Coefficient

of a data point, s(i) can be determined using the following formula:

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{3.9}$$

which can also be expanded as,

$$s(i) = \begin{cases} 1 - a(i)/b(i) & if a(i) < b(i) \\ 0 & if a(i) = b(i) \\ b(i)/a(i) - 1 & if a(i) > b(i) \end{cases} \tag{3.10}$$

where a(i) is the average distance from the i^{th} point to the other points in the same cluster and b(i) is the minimum average distance from the i^{th} point to points in a different cluster [7].

The values $s(i)$ of a silhouette coefficient usually ranges from -1 to 1. If $s(i)$ is close to 1, it means that the point is in the right cluster. If $s(i)$ is close to zero, then the point lies near the decision boundary of two neighboring clusters. Otherwise, if $s(i)$ has negative values, it means that the point is in the wrong cluster.

The average silhouette values of all the points in the cluster is used to determine the quality of the clustering method used. Hence, the optimal number of clusters for an efficient clustering would be the number of clusters that gives the highest silhouette coefficient.

Implementation

The silhouette algorithm has been implemented in MATLAB using the function *silhouette(X,clust,metric)* where X is the matrix of data points, *clust* is the cluster *ids* of each data point and metric is the inter-point distance function used.

The variable *data* is the path to the text file containing the data points while the variable *kmeans* is the path to the text file containing their corresponding cluster numbers. The metric we used is the Euclidean distance between the points.

```
% the path to the data file
data = csvread('C:\Users\Public\Desktop\AVGDATA_0.1_100000.txt');

% the file path to the clusters
kmeans = csvread('C:\Users\Public\Desktop\pointsclusters_10_0.05.txt');

% silhouette function
s=silhouette(data,kmeans,'Euclidean');

% average silhouette coefficient
mean_silhouette=mean(s);
disp(mean_silhouette)
```

The above program gives the mean Silhouette coefficient of the overall points in the cluster.

3.2 Query Space Clustering

The Online K-Means described at 3.1.3 provides an efficient approach to address huge data sets without accessing raw data. However, it is easily to notice that the input of Online K-Means (i.e., the output of queries) is randomly and uniformly chosen. This is obviously unreasonable. Thus, the challenge here is that how to generate an appropriate input for Online K-Means. Before exploring the specific solution, it is necessary to give a definition for query and explain the generation of a query.

3.2.1 Original Query Generation

A query Q has two parts: the input of the query, which called *query-point*, $\vec{x}$ and the *radius* θ, where $\vec{x}$ is a multidimensional vector from the real dataset and θ is the *radius* with a constant value.

$$Q = [\vec{x}, \theta] = [x_1, x_2, ..., x_n, \theta](n\ dimensions) \tag{3.11}$$

For example, consider dealing with a dataset *S* with two-dimensional points. The query-point is $\vec{x} = [x_1, x_2]$. In the Online K-Means mentioned before, the values of $x_1\ and\ x_2$ are chosen uniformly and randomly from *S*. The next step is to scan whole data and gather a sub dataset, which includes all data points such that the Euclidean distance between a point, for example in two dimensions, $Z = [z_1, z_2]$ and the query-point $[x_1, x_2]$ is less than θ. i.e.,

$$\sqrt{(x_1 - z_1)^2 + (x_2 - z_2)^2} < \theta$$

Subsequently, record the average of all points that satisfy the criterion: less than θ, and notate this as the output of a query.

$$average = \frac{\sum_{i=1}^{n} Z_i}{n}$$

Finally, if there are *M* queries, repeat this process of query generation for *M* times and save all the output as the input for Online K-Means.

Implementation

The original query generation was implemented in Java. The methods are shown below:

```
public static List<Data> generateQuerys(int queryLimit, int noOfAxis) {
        List<Data> data = new ArrayList<Data>();
        Random r = new Random();
        for (int i = 0; i < queryLimit; i++) {
                float[] row = new float[noOfAxis];
                for (int j = 0; j < noOfAxis; j++) {
                        row[j] = r.nextFloat() - 0.5f;
                }
                data.add(new Data(row));
        }
        return data;
}
```

This function is used to generate a set of queries randomly with two parameters: *int queryLimit* (i.e., M queries) and *int noOfAxis* (i.e., the dimensions of points). *r.nextFloat()* will return the next uniformly distributed float value between 0.0 and 1.0 and then minus 0.5 simply because the range of dataset in this case is from -0.5 to 0.5.

```
        for (Data d : dataSet) {
                if (Tools.distance(query, d) < theta) {
                        dataInTheta.add(d);
                }
        }
        return getAverage(dataInTheta);
```

The *for* loop scans whole data spcae *dataSet* and compares each point *Data d* with query point(*query*) to find points that the distance less than θ and finally save the points in a *List dataInTheta*.

```
public static float distance(Data p1, Data p2) {

        float ydist = p1.getRow()[1] - p2.getRow()[1];
        float xdist = p1.getRow()[0] - p2.getRow()[0];
        float distance = (float) Math.sqrt((ydist * ydist) + (xdist * xdist));
        return distance;
}
```

This function *Tools.distance()* can calculate the Euclidean distance for two deminsions between two points *Data p1* and *Data p2*. *xdist* and *ydist* represent the distance for each dimension.

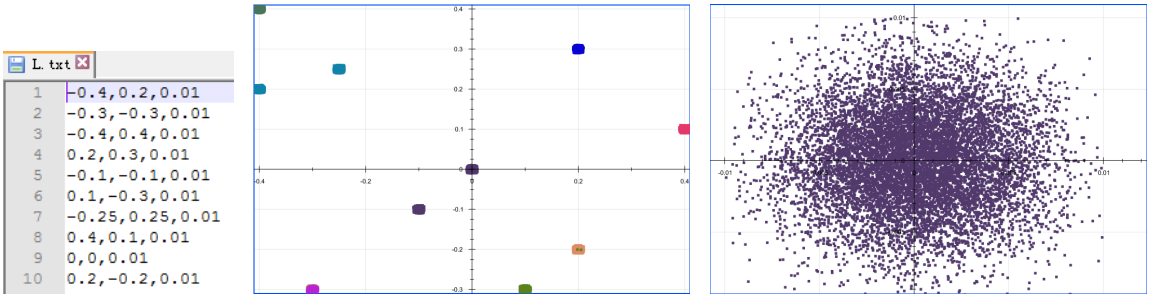
```
public static Data getAverage(List<Data> dataInTheta) {
        if (dataInTheta.isEmpty()) {
                return null;
        }
        float[] avg = new float[dataInTheta.get(0).getRow().length];
        for (Data d : dataInTheta) {
                for (int i = 0; i < dataInTheta.get(0).getRow().length; i++) {
                        avg[i] += d.getRow()[i];
                }
        }
        for (int i = 0; i < avg.length; i++) {
                avg[i] = (float) (Math.round((avg[i] / dataInTheta.size()) * 10000.0) / 10000.0);
        }
        return new Data(avg);
}
```

The idea here is that using a loop to count the sum of the points in *List dataInTheta* for all dimensions respectively and then, compute and store the mean value for each dimension in an float array. Finally, package this array as an average point structure: *Data*.

However, the idea of original query generation has two limitations. Firstly, as mentioned above, users are less likely to issue queries from whole dataset uniformly and randomly. Another problem is that this method needs to scan the entire dataset once for each query. It may cause a heavy load and high cost for computers especially when addressing a large-scale dataset. Query Space Quantization (i.e., Query Space Clustering) is proposed to solve these problems.

3.2.2 Pre-define Data Subspaces by Interest Points

To begin with, it is necessary to pre-define some data subspaces in order to simulate the areas of interest of a user. Assume that there are 10 subspaces in a 2-dimensional dataset. That is, the user normally issues queries from these ten data subspaces. More specifically, fix a data space, say space L, L = $\{l_1, l_2, ..., l_{10}\}$. Each subspace in L is modelled through a Gaussian distribution of 2 dimensions, i.e., a mean value μ_1 for dimension 1 and a mean value μ_2 for dimension 2 are needed, $l_i = (\mu_1, \mu_2)$. Furthermore, for simplicity, assume that the standard deviation σ_1 and σ_2 for both dimensions are the same and fixed, e.g., $\sigma = \sigma_1 = \sigma_2 = 0.01$.



(a) Input of L (b) Visualization of query points in L (c) Zoom in to l_9=(0,0)

Figure (a) is a screenshot for the input file L, which pre-defines ten centers for ten subspaces. The format follows (μ_1, μ_2, σ). Figure (b) and (c) show that the query points generated by Gaussian distribution can form a disc around the input from figure (a). The implementation details will be introduced in the next section.

3.2.3 Select a Subspace and Generate a Query

Initially, assume that there is a need to generate N = *10000* queries from L = 10 subspaces. That is, a subspace in L is firstly chosen from 1 to 10, each one with equal probability, i.e., Random($l_1, ..., l_i, ..., l_{10}$), i = (1...10). Intuitively, 10000/10 = 1000 queries will be generated from each subspace. Next, in order to generate a query Q = $[x_1, x_2, \theta]$ from the *i-th* subspace, it is necessary to set the center of query-points, $l_i = (\mu_1, \mu_2)$, and the θ is fixed, e.g., $\theta = 0.1$. The x_1 is generated by the Gaussian with mean value μ_1 and variance σ^2 ($Reminder : \sigma = \sigma_1 = \sigma_2 = 0.01$). The same holds true for x_2. That is, x_1 = Random.Gaussian(μ_1,σ^2) and x_2 = Random.Gaussian(μ_2,σ^2). Hence, the query Q = $[x_1, x_2, \theta]$ with the center of query-points $l_i = (\mu_1, \mu_2)$ is located within the *i-th* subspace, i.e., in a disc of center (μ_1, μ_2).

Implementation

The generation of query-points, which follow Gaussian distribution, was implemented in Java as well. The function is presented below:

```
private float[] getRandomPointInBox(float[] point, float width) {

        float[] result = new float[point.length];
        for (int i = 0; i < result.length; i++) {
                float g = 0.0f;
                boolean found = false;
                while (!found) {
                        g = (float) ((r.nextGaussian() * (width / 3)) + point[i]);
                        if (g < (point[i] + width) && g > (point[i] - width)) {
                                found = true;
                        }
                }
                result[i] = g;
        }
        return result;
}

private Random r = null;

private Tools() {
        r = new Random();
}
```

There are two parameters. *float[] point* represents the center (μ_1, μ_2). *float width* means the standard deviation σ. *r.nextGaussian()* returns the next Gaussian distributed double value with mean 0.0 and standard deviation 1.0. And then multiply it by (*width/3*) which enables that about 99.7%

of a population will be within the range (-σ, +σ). Finally, moving this point towards the center by adding the mean value μ_1 and μ_2 respectively.

3.2.4 Online Quantization

Finally, the concept here is to address the problem of time-consuming and high cost for executing a query over large-scale dataset by avoiding storing all the queries, scanning all dataset. In the reality where users issuing queries, it is essential to quantize them *online*. That is to incrementally generate queries and then injecting each one to the online K-means algorithm, for quantizing the query vectors. Obviously, if K equals the number of subspaces in L, then after a lot of queries, it can be seen that all the K-means vectors will be the vectors with dimensions ($\mu_1[i]$, ..., $\mu_n[i]$) since, naturally, the K-means algorithm learn the query distribution, which in this case in a 10-modal Gaussian distribution, i.e., 10 Gaussian bells.

Implementation

```
for (int i = 0; i < queryLimit; i++) {
        printQueryCompletion(i, queryLimit);
        // generate query
        Data query = Tools.getInstance().generateQuery(distributions, noOfAxis);
        // update online kmeans and write cluster to file
        int queryClusterId = queriesOnline.update(query.getRow());
        Data dataCentroid = new Data(queriesOnline.getCentroids().get(queryClusterId));
```

int queryLimit in this part of *for* loop is the number that represents M queries. The function *printQueryCompletion()* is just to report the process to console, so ignore it. *Tools.generateQuery()* can produce a query vector as decribed above. *queriesOnline.update()* calls the online K-means and returns the closest centroid id. The centroid can also be acquired by *queriesOnline.getCentroids()*. Thus, it means, for each query, inject it to online K-Means and then repeat M times.

3.3 Prediction

Within the prediction sections 3.1 and 3.2 with their described techniques are join up with each other but only under consideration of two dimensional data. The intent outcome of the prediction is to find the average data point of a query without scan through the behind dataset.

In order to achieve this, a training set is needed to learn the machine algorithm and finally test with another dataset how good this learning was. The goodness of this machine learning algorithm, the exact evaluation, will be explained in chapter 4. In this section the implementation of creating the training and test set, learning the algorithm and predict the outcome from a query input will be described in the following subsections. Each subsection represents a standalone application which can be run through the generated batch file with modifying the depending input variables.

3.3.1 Mapping query and output data

The fundamental for a successful prediction is to create a training set that is mapping the query that was generated through the actual output, the average data point.

A query is defined in equation 3.7 with a point x for each dimension and the radius theta, a constant in our work.

$$q = [x_1, x_2, \theta] = [\vec{x}, \theta] \tag{3.12}$$

For each query an output, containing the average data point of all data points from the real dataset inside the defined radius theta, will be generated. Therefore the output of a query is defined as:

$$\bar{x} = [\bar{x_1}, \bar{x_2}] = \frac{1}{n} \sum \vec{x_i} :\| \vec{x} - \vec{x_i} \| \leq \theta \tag{3.13}$$

Figure 3.1 visualizes this technique by showing a two dimensional dataset, representing a query with a blue point and a dashed line for the radius theta, the actual dataset points with a black point and the average data point with a red one.

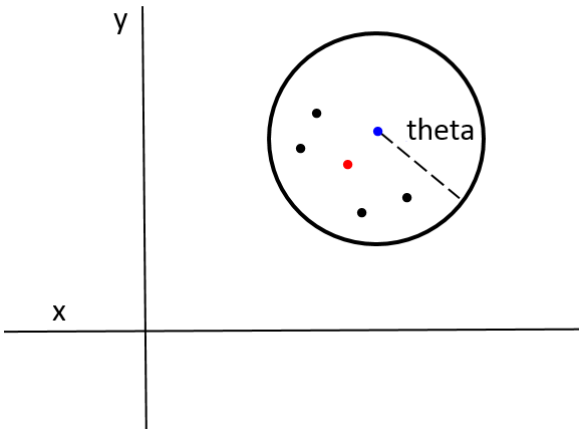


Figure 3.1: Query and average data point creation

The exact approach and implementation will be found in section 3.2.1.. With our definition of query and the related output for each query we can define our training set as:

$$Trainingsset = [q, \bar{x}] \tag{3.14}$$

Implementation

The application for mapping a query and the average data point is implemented in Java; an extract of the java class QueryGE is presented below:

```
for (int i = 0; i < queryLimit; i++) {
        // print status update every 10%
        printCompletion(i, queryLimit);

        // generate query
        float[] query = Tools.getInstance().generateQuery(distributions, noOfAxis);

        // run query to get avg data
        float[] dataCentroid = Tools.getInstance().getAverageDatumFromQuery(dataSet, query, theta);
        if (dataCentroid != null) {
                // write AVGDATA
                avgDataWriter.write(Arrays.toString(dataCentroid).replace("]", "").replace("[", "") + "\n");
                avgDataWriter.flush();
                // write training set = query, avgdata
                Tools.getInstance().writeQueryAndADToWriter(trainingSetWriter, query, dataCentroid);
        }
}
```

Inside the *for* loop a random query will be generated with the method *generateQuery()* inside the ranges of the defined subspace explained in section 3.X; if no subspace is defined the query will will be generated over the whole dataset. After a query is generated the average data point for this query will we located through the method *generateAverageDataFromQuery()*. Both query and average data point are stored in a float array. The final coordinates of the query and the average point will than be written into a semicolon separated file using *BufferedWriter*.

The whole application *TrainingSetAppliction* can be run by giving a dataset, theta, a number of queries and a file with subspaces the user is interested in. After training the machine learning algorithm a test set is needed with the same structure of the training set; therefore this application can be used also for creating the test set.

3.3.2 Learning algorithm

In order to process the learning from the previously generated training set we can use the online k-means algorithm. This concept was explained in section 3.1.3. A short summary of the key facts: the online k-means sets the first k points as cluster centroids and moves the nearest centroid for the next point in its direction with a fixed alpha. The following paragraph will explain how we connected the online k-means approach with our training set and how we created our set for the prediction.

To find the centroid of all queries we give them one by one to the online kmeans algorithm. At the same time another instant of the online k-means gets the related average point. This is implemented in our java class *LearningApplication* in row 64-65, see following code extract:

```
int queryClusterId = queryClustering.update(query);
int dataClusterId = dataClustering.update(avgData);
```

The important thing is that the connection between the cluster of a query and the average point is still established. Wherefore we store the cluster id of our average data on the same position of the query cluster id. This can be seen in this code extract:

```
queryDataClusterMap[queryClusterId] = dataClusterId;
```

Thanks to this, after running the online k-means through the whole training set, we have a list of cluster id of our average data on the position of the query cluster id. As a result we can write the centroid of our queries, defined in equation 3.10, with the related centroid of the average data, defined in equation 3.11, in a file by using *BufferedWriter*. See posterior code detail:

```
// save the queryDataClusterMap - which represents the link between
// query clusters and average data clusters
System.out.println("Starting to write queryDataMap");
try {
        BufferedWriter mapWriter = new BufferedWriter(new FileWriter("queryDataMap_" + k + "_" + alpha + ".txt"));
        for (int i = 0; i < queryDataClusterMap.length; i++) {
                if (queryDataClusterMap[i] != -1) {
                        Tools.getInstance().writeQueryAndADToWriter(mapWriter, queryClustering.getCentroids().get(i),
                                        dataClustering.getCentroids().get(queryDataClusterMap[i]));
                        mapWriter.flush();
                }
        }
```

Therefore our prediction set will be containing the centroid of our queries w[j] and the correspondent centroid of the average data u[j], see the following notations for definition.

$$w[j] = online\, k - means\, centroids\, for\, q \tag{3.15}$$

$$u[j] = online\, k - means\, centroids\, for\, \bar{x} \tag{3.16}$$

$$Prediction\, set = [w[j], u[j]] \tag{3.17}$$

3.3.3 Prediction algorithm

In the last application our previous generated prediction and a new test set will be used to predict for each query inside the test set an average data point without scanning through the dataset.

To predict the average point we try to find for each query the alike query-centroid. This can be done by using the nearest neighbour algorithm. It is calculation the euclidean distance between a point and a list of points and gives you for this point the nearest. For more details and the exact implementation go to section 3.1.1. We need to search for the nearest query centroid for this query in order to find the correspondent average data centroid and declare it as our predicted *xbar*.

This logic is implemented with the java class *PredictionApplication*:

```
// predict the output
Integer queryClusterId = VectorFunctions.classify(query, queryCentroids);
float[] predictedXBar = avgDataCentroids.get(queryClusterId);
```

Within this extract the method *classify()*, which is explained in section 3.1 with more detail, is invoked. This method takes the list of centroids and the new query of the test set as input and returns the nearest centroid of queries for the input query. Afterwards the centroid of the average data is easily found and marked as our predicted *xbar*.

To evaluate the goodness of our predicted *xbar*, we can define an error value for each query. This error value is defined in 3.13 as the euclidean distance between the predicted *xbar*, our centroid, and the actual *xbar*.

$$\epsilon_i = \| \bar{x} - u[j] \| \tag{3.18}$$

```
// calculate the error
float e = VectorFunctions.distance(actualXBar, predictedXBar);
error += e;
count++;
```

For a summary evaluation it is possible to calculate the mean error over the test set by summing each error and divide it through the number of queries in the test set. The mean error is defined in equation 3.14.

$$Mean\,Error = \frac{\sum \epsilon_i}{i} \tag{3.19}$$

```
float meanError = error / (float) count;
```

Further evaluation can be done by reading the result file that will be produced through the prediction. This file contains, for each query, the predicted average data point and, the actual data point and the error value. At the end of this file the mean error is displayed.

Chapter 4

Evaluation

During the evaluation we used public data. For the algorithm k-means (batch and online), ART and query space we used a dataset about gas sensor data from the machine learning repository of UIC. The Gas data is containing the first 5 columns and is limited to the first 500.000 rows because of memory issues during the silhouette evaluation. For the prediction we used this dataset but only with 2 columns and another dataset from this repository 'tamilnadu electricity board hourly reading'.

4.1 Silhouette

Silhouette is a algorithm that validate the consistency and strength of a cluster. This algorithm was described in Section 3.1.5 in detail. A mean Silhouette value between 0 to 1 means a good cluster. If the value is 1 it means a perfect cluster. For values between 0 and -1 the points are mostly not in the nearest or best cluster.

4.1.1 K-means

Batch and Online K-Means were tested as well with different values of k $(50, 150, 400)$ and $theta$ $(0.05, 0.2)$ for the queries. For K-Means functions the number of radius queries is 100.000.

If we compare online and offline k-means over real data (see table below) we can see that online k-means is producing a better silhouette value than the offline k-means in case of 150 cluster. Therefore it can assume that the online k-means is as good as the offline k-means, sometimes even better.

	K=50	K=150	K=400
Batch K-means		0.3584	out of memory
Online K-Means		0.4713	

The silhouette value over queries for $theta$ 0.05 and 0.2 using online and batch k-means is displayed in the following tables. For $theta$ 0.05 no silhouette values could be created. This causes of to less datapoints that were given as result from the queries.

Theta=0.2 and M=10k	K=50	K=150	K=400
Batch K-Means	0.2569	0.-0.0815	-0.1345
Online K-Means	0.339	0.2894	NaN

Theta=0.2 and M=100k	K=50	K=150	K=400
Batch K-Means	0.4163	0.3025	0.2492
Online K-Means	0.3778	0.3337	0.2579

Theta=0.2 and M=200k	K=50	K=150	K=400
Batch K-Means	0.1899	0.0767	-0.0252
Online K-Means	0.3719	0.3402	0.2808

From the above tables it is possible to assume that the online k-means is getting better with more queries. The values of the silhouette for the average data are even with a small number of queries nearly the value over real data from the batch k-means. The batch k-means is the opposite, it is getting worse with more queries. This is because .....

4.1.2 ART

For a better understanding of the impact of vigilance and theta values in ART, we tested the algorithm with different values for both variables, evaluating the average clustering results with silhouette. As we already mentioned before, silhouette algorithm computes the average result for all the clusters evaluations. It is possible to notice that the performance is proportional to theta; however, for greater vigilance values the difference in performance as theta varies becomes more negligible. The number of radius queries generated and executed over the original data space in order to test ART is 200.000. In figure 4.1 the 3D plot with the relation between vigilance, theta and the mean silhouette value is shown. The follow table shows the exact values of silhouette for each theta (row) and vigilance (column).

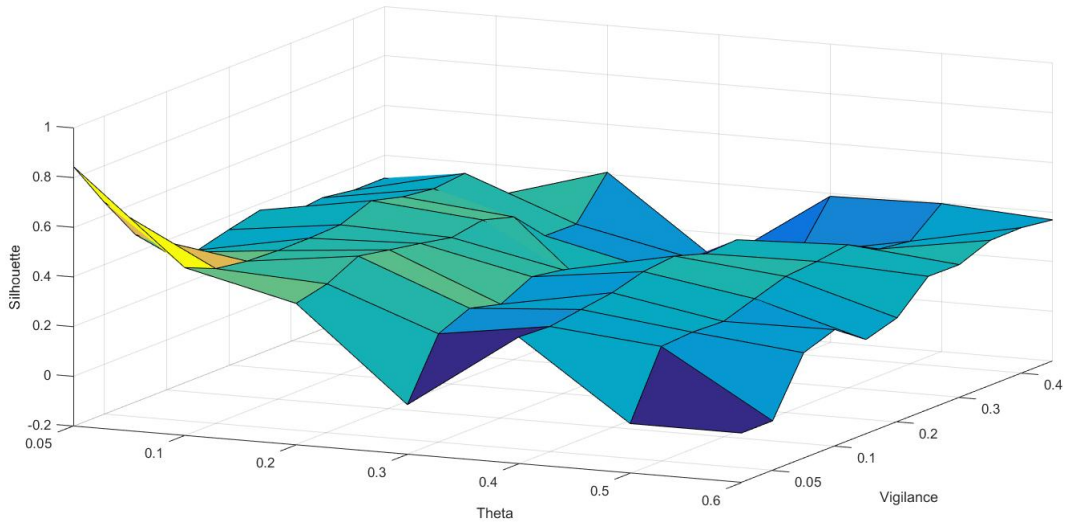
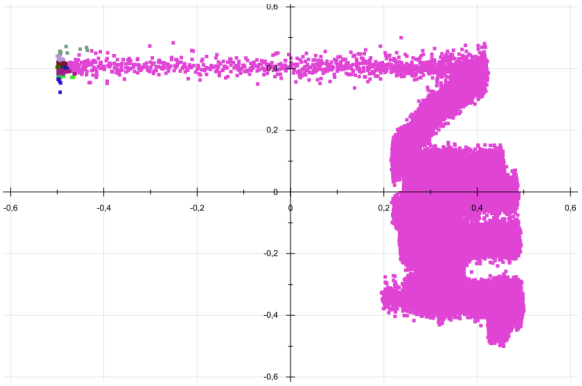


Figure 4.1: The impact of Theta and vigilance in ART

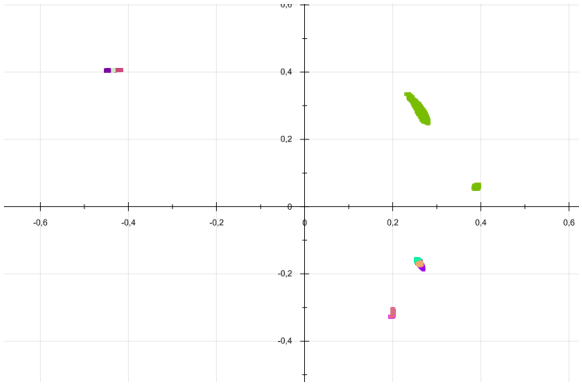
	0.05	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.05	0.8448	0.6508	0.4719	0.3514	0.3125	0.3458	0.3752	0.3117	0.3271	0.3083	0.3083
0.1	0.4757	0.4226	0.4038	0.3974	0.3981	0.4032	0.4511	0.4177	0.4013	0.4145	0.2752
0.2	0.3697	0.3993	0.4621	0.4381	0.4152	0.3899	0.4117	0.3749	0.2501	0.2481	0.4076
0.3	NaN	0.2365	0.2902	0.2573	0.3191	0.2981	0.1307	-0.0128	0.1476	0.1539	0.0862
0.4	0.3105	0.3038	0.3234	0.3417	0.3777	0.3914	0.3519	0.3591	0.2602	0.2086	0.387
0.5	NaN	0.2617	0.27	0.2644	0.3068	0.3476	0.3588	0.377	0.311	0.2933	0.3957
0.6	NaN	NaN	0.2251	0.2693	0.1804	0.2188	0.3368	0.3349	0.3841	0.3862	0.3684

4.2 Query Space

For evaluating the query space we do not cluster the query output instead we clustered the actual query. To evaluate this we used visualization.

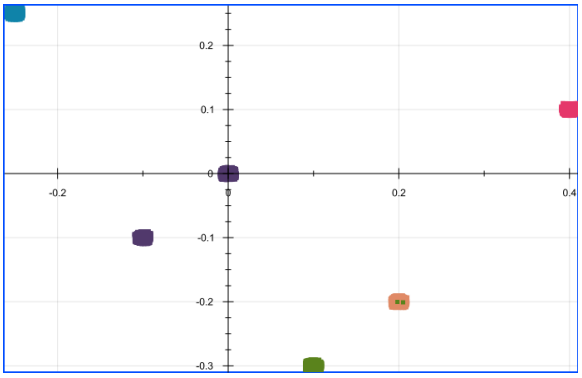


(a) Real data

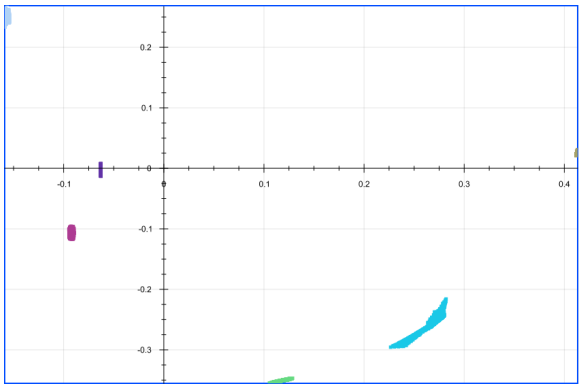


(b) Real data subspace output

The idea here for evaluation is to check the similarity of the queries. The methods is to plot the K means over the space to compare their positions with the center points.



(c) Query points distribution



(d) Query clusters

	No.1	No.2	No.3
Mean value of query points	(-0.25,0.25)	(-0.1,-0.1)	(0,0)
Centroid of query cluster	(-0.1522,0.2484)	(-0.0912,-0.1108)	(-0.0628,-0.0023)
Euclidean Distance	0.097813	0.013931	0.062842
	No.4	No.5	No.6
Mean value of query points	(0.1,-0.3)	(0.2,-0.2)	(0.4,0.1)
Centroid of query cluster	(0.117,-0.3465)	(0.2653,-0.2567)	(0.4115, 0.0255)
Euclidean Distance	0.04951	0.086481	0.075382

From the images and table above, each subspace has a corresponding cluster. Although the shape changes a lot, it is still likely to conclude that there is no significantly difference between the positions of queries since the distances are similar.

Further work

For each subspace, it is chosen from L with equal probability. It is a rare situation in the reality. Some probability theories and predictive methods could be involved in, such as Bayesian inference. On the other hand, for each subspace, it is modeled by Gaussian distribution. Although it is the most common distribution, some other models, such as linear regression, could also be considered. Combining with some inference theories, this can be more accurate.

4.3 Error

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 set cardinality in distance nearest neighbours, 2015.

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

[4] Lon Bottou and Yann LeCun. Large scale online learning. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schlkopf, editors, *NIPS*, pages 217–224. MIT Press, 2003.

[5] 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.

[6] Dimitrios Gunopulos, George Kollios, J. Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, April 2005.

[7] Peter Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20(1):53–65, November 1987.