JMLR: Workshop and Conference Proceedings 36:61–76, 2014 BIGMINE 2014

A Clustering Algorithm Merging MCMC and EM Methods Using SQL Queries

David Sergio Matusevich MATUSEVICH@CS.UH.EDU
and
Carlos Ordonez ORDONEZ@CS.UH.EDU
University of Houston, Houston, TX 77204, USA

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

Clustering is an important problem in Statistics and Machine Learning that is usually solved using Likelihood Maximization Methods, of which the Expectation-Maximization Algorithm (EM) is the most common. In this work we present an SQL implementation of an algorithm merging Markov Chain Monte Carlo methods with the EM algorithm to find qualitatively better solutions for the clustering problem. Even though SQL is not optimized for complex calculations, as it is constrained to work on tables and columns, it is unparalleled in handling all aspects of storage management, security of the information, fault management, etc. Our algorithm makes use of these characteristics to produce portable solutions that are comparable to the results obtained by other algorithms and are more efficient since the calculations are all performed inside the DBMS. To simplify the calculation we use very simple scalar UDFs, of a type that is available in most DBMS. The solution has linear time complexity on the size of the data set and it has a linear speedup with the number of servers in the cluster. This was achieved using sufficient statistics and a simplified model that assigns the data-points to different clusters during the E-step in an incremental manner and the introduction of a Sampling step in order to explore the solution space in a more efficient manner. Preliminary experiments show very good agreement with standard solutions.

Keywords: Clustering, EM, DBMS, SQL, Monte Carlo, Markov chains, Bayesian methods,

1. Introduction

Clustering is a fundamental data mining technique that is frequently used as a building block for the treatment of more complex problems such as Class Decomposition and Bayesian Classifiers. Among clustering algorithms, the Expectation Maximization (EM) algorithm is popular due to its simplicity, its numerical stability and reliability. EM belongs in the class of maximization algorithms, that solve a particular problem by finding a maximum of a quantity that describes it, in this case the log-likelihood.

Maximization algorithms' weakness is that, if the problem to be explored has more than one possible extrema, the solution found might be a "good" solution, rather than the "best" one. This has been extensively studied in the literature. Markov Chain Monte Carlo methods can help us escape local extrema, but at the cost of an increase in orders of magnitude of the number of iterations required for convergence. This negatively impacts

© 2014 D.S. Matusevich & C. Ordonez.

the practicality of the implementation of a MCMC clustering algorithm, in particular in the case of problems with a massive amount of data to process. When dealing with massive data volumes we must also take into account the possibility of using all the RAM memory available, even in modern systems. Therefore it is imperative to factor into the time per iteration, the time needed to retrieve the information from secondary storage, as well as data security and fault tolerance. For these reasons solving the clustering problem using MCMC methods is more challenging, in particular when analyzing big data. This poses a complication since the characteristics of big data often require the use of Markov chain methods.

The SQL query language can thus be an important tool for the programmer. SQL automatically handles all the problems with storage and data security and, since it's implemented inside the DBMS, that is usually the data repository, it simplifies the retrieval of data, in particular when the data in question doesn't fit in main memory. This, however, comes at a cost: SQL is less flexible and slower than high level languages like C++, since it is designed to work with tables inside a relational database.

In this work we present a new implementation of an MCMC EM algorithm for the calculation of clustering in large tables, implemented inside the DBMS, using queries for the calculation of the Expectation step, and an external step for the maximization and sampling of the data. This hybrid approach was taken in order to allow for the use of several separate nodes in a parallel implementation.

This article is organized as follows: Section 2 presents a brief introduction to the EM algorithm and to Markov processes. Section 3 details our efforts to efficiently incorporate MCMC into the EM algorithm. In Section 4 we describe the complete algorithm and show the SQL queries required to calculate the E-Step. Section 5 shows the results obtained and a validation of the method. Finally Sections 6 and 7 explore the related work and our conclusions.

2. Definitions and Preliminaries

2.1. The EM Algorithm

The Expectation Maximization (EM) algorithm is an iterative method to estimate the parameters of a mixture of k normal distributions. The EM algorithm finds the maximum likelihood estimate of the parameters of a distribution for a given data set. In this case, where the probability is a multivariate normal distribution, and each data point is a d-dimensional vector $x = \{x_1, \ldots, x_d\}$, the probability of a given data point belonging to a cluster characterized by parameters C_j (mean) and R_j (covariance) is

$$P(x; C_j, R_j) = \sqrt{(2\pi)^d |R_j|} \times e^{-\frac{1}{2}(x-C_j)^T R_j^{-1}(x-C_j)} \tag{1}$$

It is important to note that R_j, the covariance matrix of cluster j is a $d \times d$ diagonal matrix, so in practice it is convenient to store it in a vector of dimension $d \times 1$. The likelihood is calculated as the probability of the mixture:

$$P(x; C, R, W) = \sum_{j=1}^{k} W_j P(x; C_j, R_j) \tag{2}$$

where W_j is the weight of cluster j. This last equation introduces the $d \times k$ matrices C and R, where the means and covariances of all the clusters are stored, as well as W, the $1 \times k$ matrix of weights (Table 1).

The EM algorithm is characterized by two distinct steps: An Expectation step (E-Step) where the likelihood of a given solution is calculated using the current estimates of the cluster parameters, and a Maximization step (M-Step), where the parameters of the solution are re-estimated, based in the probabilities calculated on the expectation step. The quality of the solution $\Theta = \{C, R, W\}$ is measured, as previously mentioned, by the Log-Likelihood, $L(\Theta)$:

$$L(\Theta) = \frac{1}{n} \sum_{i=1}^{n} \log\left(P\left(x^i; \Theta\right)\right) \tag{3}$$

The importance of this quantity cannot be overestimated, since it is at the core of any Maximum-Likelihood algorithm. The classic EM algorithm alternates E-Steps and M-Steps until the change in L is less than a predetermined limit.

Another important quantity that is commonly associated with mixtures of normal distributions is the *Mahalanobis Distance*. If we want to calculate how far is a point x from a cluster j with mean C_j and covariance R_j, the Mahalanobis distance

$$\delta(x, C_j, R_j) = \delta_{ij} = \sqrt{(x - C_j)^T R_j^{-1} (x - C_j)} \tag{4}$$

gives a much better understanding than the regular Euclidean distance since it gives an idea, not only of how far the data point is from the center of mass of the distribution, but also of the density of the cluster, by scaling the distance by the covariance matrix.

Table 1: Cluster Matrices

Name	Size	Contents
C	$d \times k$	means
R	$d \times d \times k$	variances
W	$1 \times k$	weights

Table 2: Sufficient Statistics

Name	Size
N	$1 \times k$
L	$d \times k$
Q	$d \times k$

2.2. The Markov Process

A Markov process is a stochastic process such that the conditional probability of value x^i at time t^i is uniquely determined by the value x^{i-1} at time t^{i-1} and not by any knowledge of the values at earlier times. In simpler terms it is a rule for randomly generating a new configuration of a system from the present one (Binney et al. (1992)). This rule can be expressed as a set of transition probabilities from state α to state α'. These probabilities must satisfy the sum rule

$$\sum_{\alpha'} P(\alpha \to \alpha') = 1 \tag{5}$$

since it is evident that at each step the system must go somewhere. A *Markov Chain* is a sequence of states generated by a Markov process in which the frequency of occurrence of

a state α is proportional to the *Gibbs Probability* p_α and where the transition probabilities obey the conditions of accessibility and microreversibility (or detailed balance)

In a Monte Carlo algorithm, we generate a Markov chain in which each successive state is generated by its predecessor and accepted (or rejected) according to an acceptance ratio determined by the probabilities of each state. This algorithm randomly attempts to sample the solution space, sometimes accepting the move and sometimes remaining in place. Gibbs Samplers are Markov Chain Monte Carlo algorithms for obtaining sequences of observations of the joint probability distribution of a set of random variables, when direct sampling is difficult. Gibbs Samplers are commonly used as means of Bayesian Inference but they retain many similar characteristics to the maximization step of EM.

Monte Carlo algorithms (in particular the Metropolis Hastings model) were developed to solve this kind of problems Hitchcock (2003). Deterministic schemes, such as the Gauss-Newton Algorithm and its variants (or for that matter, EM) find progressively better solutions in an iterative manner, with the result that, if there are several possible minima, the algorithm can stop at a local one, without finding the global result we are looking for. In contrast Monte Carlo iterations allow the solution to worsen, climbing out of the shallow valleys in order to find the deep ones. In other words, Monte Carlo techniques effectively sample phase space until the best possible solution is found.

There are many advantages to the use of MCMC methods to data analysis, such as the calculation of confidence intervals, as used to calculate indirect effect by Preacher and Selig (2012) and it has been successfully applied to the management of probabilistic data by Jampani et al. (2008).

3. Merging EM with Monte Carlo

In this section we will describe our efforts to incorporate Monte Carlo techniques into the EM algorithm. We will also describe some ways of improving the time performance of the algorithm in order to reduce the number of iterations required for convergence.

3.1. Sufficient Statistics

At the core of the algorithm is the concept of sufficient statistics. Sufficient statistics are multidimensional functions that summarize the properties of clusters of points. One of the interesting properties of these summaries is that they are independent, that is, statistics from one cluster do not depend on the values of the data points from other clusters.

For this algorithm we introduce three statistics per cluster D_j $(j = 1, \ldots, k)$

$$N_j = |D_j| \tag{6}$$

$$L_{i,j} = \sum_{i=1}^{N_j} x_i \text{ where } x \in D_j \tag{7}$$

$$Q_{i,j} = \sum_{i=1}^{N_j} x_i^2 \text{ where } x \in D_j \tag{8}$$

N is a $(1 \times k)$ matrix that stores the size of each cluster D_j, L is a $(d \times k)$ matrix that stores the sum of the values of all data points that belong in each cluster, and Q is a $(d \times k)$ matrix that stores the sum of the squares of the data points. These particular statistics have another important property: Since they are calculated in an additive manner, they are easy to calculate using multiple threads or multiple processors. We can use these sufficient

statistics to reduce the number of reads of the dataset, allowing the periodic estimation of the parameters without resorting to extra I/O steps, thereby significantly reducing the algorithm's running time.

3.2. The Fractal Nature of Data

For sufficiently large data sets, a portion of the data-set is representative of the whole. This is particularly true in the case of very large multidimensional datasets, where there are no preferred orderings, and any sufficiently large subset will closely resemble the complete sample.

We can exploit this property of very large datasets to accelerate the convergence of the EM algorithm. Following Ordonez and Omiecinski (2002), we can reduce the number of iterations necessary for convergence by inserting M-Steps every ψ datapoints, instead of waiting until the end of the dataset to maximize. Since the maximization step is performed roughly $\lfloor n/\psi \rfloor$ times per iteration, the time needed for convergence is significantly shortened. The authors arrive experimentally to a value of $\psi = \lfloor \sqrt{n} \rfloor$, however this can be tweaked to fit the particular dataset to be explored.

3.3. The FAMCEM Algorithm

In this section we describe FAMCEM (Fast and Accurate Monte Carlo EM), an algorithm that improves EM by incorporating ideas from Monte Carlos style algorithms, first introduced in Matusevich et al. (2013).

Recalling the definitions of N, L and Q we calculate the parameters of the problem using

$$C_j = \frac{L_j}{N_j} \qquad \text{The } d\text{-dimensional cluster average} \tag{9}$$

$$W_j = \frac{N_j}{\sum_{j'=1}^{k} N_{j'}} \qquad \text{The weight of the cluster} \tag{10}$$

$$R_j = \frac{Q_j}{N_j} - \frac{L_j L_j^T}{N_j^2} + \lambda\Sigma \qquad \text{The } d\text{-dimensional variance} \tag{11}$$

where Σ is the global standard deviation and λ is a small positive constant used to circumvent EM's weakness when variances approach zero. The introduction of the $\lambda\Sigma$ term ensures that the variances will never be zero, even if there is very little change in any particular dimension. It should be chosen small enough that it will not impact negatively in the calculations, but large enough that the covariance matrix is not singular.

FAMCEM introduces two significant changes to the EM algorithm. In the regular EM algorithm, the datapoint is assigned a series of probabilities of belonging to each of the clusters. In FAMCEM we use these probabilities to find a unique cluster to which the data point belongs, this produces a decoupling between the clusters, simplifying (as we will see) the calculation of the Log-Likelihood. However, instead of assigning the data point to the cluster with highest probability, we make a random assignment. For each point we calculate the set of probabilities of belonging to each of the clusters and using the histogram method, we decide to which cluster to assign it.

Algorithm 1 The FAMCEM Clustering Algorithm

Input: $X = \{x_1, x_2, \ldots, x_n\}$ and k
Output: $\Theta = \{C, R, W\}$ and $L(\Theta)$
  // Parameters (defined in the text)
  $\psi \leftarrow \lfloor \sqrt{n} \rfloor$, $\alpha \leftarrow \frac{1}{d \times k}$, $\lambda \leftarrow 0.01$
  for $j = 1$ **to** k **do**
    $C_j \leftarrow \mu \pm \alpha \times r \times \text{diag}|\sigma|$, $R_j \leftarrow \Sigma_j$, $W_j \leftarrow 1/k$
  end for
  $I = 0$
  while $(|L(\Theta)_I - L(\Theta)_{I-1}| > \epsilon$ and $I \leq$ MAXITER) **do**
    // Initialize the sufficient statistic matrices
    for $j = 0$ **to** k **do**
      $N_j = 0$, $L_j \leftarrow \overrightarrow{0}$, $\mathbf{Q_j} = \mathbf{0}$
    end for
    for $i = 1$ **to** n **do**
      // **Expectation Step**: Choose to which cluster the data point x_i belongs
      Find the maximum probability cluster m_0
      Choose m randomly, according to the probabilities p_{ij}.
      $a \leftarrow$ Random number $a \in [0, 1)$
      if $p(m)/p(m_0) > a$ **then**
        Use m
      else
        Use $m \leftarrow m_0$
      end if
      if $(i \mod \psi = 0$ **or** $i = n)$ **then**
        if $(i <$ BURNIN$)$ **then**
          // **Maximizing Step**: During the burnin period, we maximize.
          Update Equations 9 through 11
        else
          // **Sampling Step**: After the burn-in period, we sample instead of maximize
          Update Equations 9 and 10
          // We purposely *de-tune* the centroids, in order to avoid local extrema
          $C_j \leftarrow$ **rnorm**(C_j, R_j)
        end if
      end if
    end for
    Calculate the Log-Likelihood
    $\epsilon \leftarrow (1 - L(\Theta_{I-1})/L(\Theta_I))$
  end while

To illustrate this step, we can imagine a distribution of points in two-dimensional space: Some points are relatively easy to place within a cluster, that is to say, the probabilities are skewed in favor of only one cluster. However, in general this is not the case. In fact any point between two centroids could belong to either one, particularly if the standard deviations

(radii) are large enough. While at some point during the procedure, the probability might favor one cluster to the detriment of others it is possible that this is just an artifact of the current distribution. Allowing a certain opportunity to be included in other clusters, while making the current iteration slightly worse, could lead to an overall improvement of the solution.

The second important change is in the M-Step. After a good estimate for the parameters is reached, we alternate the maximization step with a "sampling step". During this S-Step we update the parameters in Equations 9 and 10, however instead of updating 11, we add an extra step: After we have an estimate for C_j we use a normal distribution to calculate C'_j, using $|R_j|$ as the standard deviation. This allows for further exploration of the solution space, improving our chances of finding the global extrema. It is important to emphasize that the Sampling phase only explores the priors of C and W but not R. R uses a non-informative prior and remains fixed during Sampling, to reduce the complexity of the problem (Liang (2009)). We decide whether to accept this C'_j using the Log-Likelihood as a sort of energy function. For each cluster D_j we:

- Calculate the change in Log-Likelihood: $\Delta L = L(\Theta_i) - L(\Theta_{i-1})$
- If the change is a positive one, we accept it.
- Otherwise we find a random variable $r \in [0, 1]$
- Accept the new configuration if $r \leq e^{-\beta \Delta L}$

where β is a constant that regulates the mixing of the model: large values of β will propitiate more chances of the change being accepted.

4. SQL Query optimization for the FAMCEM Algorithm

In this section we present the main contribution of the paper. We will describe how to implement the FAMCEM algorithm using SQL queries automatically generated by our code, complemented with simple scalar UDFs. The algorithm has two main sections, that are executed in different parts of the system. The first part, the expectation step, is implemented by sending queries to one or more database system (workers). The second part, the maximization or sampling steps, are implemented in the main computer (master). The reason for this separation is that, although they could have been solved as queries, it is convenient to use the M and S steps as a way to join the results from the separate threads that manage the worker servers. Thanks to this approach no data arrays were required and no data was stored in the master computer's main memory.

4.1. The System

The system used in the calculations is composed of a node fulfilling the role of Master, where the algorithm is run and the queries are generated, and a cluster, hosting separate DBMS with replicated tables, in the role of workers. Communications between the master and the workers is accomplished using a ODBC driver.

4.2. Initialization

Given table `X`, we need to compute some initial statistics for the data set, such as the number of rows and the mean and standard deviation of each column (we assume that the table has an ordinal primary key and that the index is clustered). This is accomplished using the following query:

```
SELECT
 sum(1.0),
 AVG(X.X1),STDEV(X.X1),AVG(X.X2),STDEV(X.X2),...,AVG(X.Xd),STDEV(X.Xd)
FROM X
```

These initial statistics are used for to compute the initial parameters of the model. The parameters C_h are chosen randomly in the interval $(\mu_h - 2 \times \sigma_h, \mu_h + 2 \times \sigma_h)$ and we set $R_h = \sigma_h$ for all clusters. These parameters are then stored inside all the DBMS workers in the form of a table `ParamTable(gen, C1_1, R1_1, C2_1,..., Cd_k,Rd_k)`.

The final parameter that needs to be computed is ψ, the size of the disjoint sets our table will be divided. ψ is empirically set to $\psi = \sqrt{n}$, but this number can be tailored to the dataset.

4.3. The Expectation Step

Once the initial setup is completed we can start the optimization process. During the E-Step we first compute the probabilities, given by the formula 1.

```
INSERT INTO P(i,gen,P1,...,Pk)
SELECT
 X.i, g,
 EXP(-0.5*(SQUARE(X.X1-Pa.C1_1)/Pa.R1_1+...+SQUARE(X.Xd-Pa.C1_d)/Pa.R1_d))/
 SQRT(Pa.R1_d*...*Pa.R1_d)+lambda*sigma_1 p1,
 ...
 EXP(-0.5*(SQUARE(X.X1-Pa.Ck_1)/Pa.Rk_1+...+SQUARE(X.Xd-Pa.Ck_d)/Pa.Rk_d))/
 SQRT(Pa.Rk_1*...*Pa.Rk_d)+lambda*sigma_d pk
FROM X,
 (SELECT *
  FROM ParamTable
  WHERE ParamTable.gen = g) Pa
WHERE X.i>g*psi
AND X.i<=(g+1)*psi;
```

In this query, `g` is an integer that denotes in which of the sub-tables we are performing the calculations. `g` goes from 1 to $\lfloor n/\psi \rfloor$. `lambda` is a small constant given by the user, and `sigma` is the standard deviation of the cluster as calculated in the previous iteration. This last term is introduced to insure that no probabilities are zero. `lambda` is chosen in such a way that it is small enough not to interfere with the probabilities, while insuring there are no singular errors due to round up. The probabilities are stored in table `P`.

The next step is to choose a cluster for each row in the sub-table. For this purpose we wrote a scalar user defined function (UDF) and a user defined type (UDT) to pass

it the information. Scalar UDFs are a very common way of simplifying calculations in a DBMS, but we purposely chose to keep it as simple as possible in order to ensure an easy transition to other DBMS, since the syntax can vary from one system to another. The UDF is called `histograms` and takes the probabilities calculated in the previous step and returns an assignment to a random cluster. We must remark that we used SQL Server syntax, and this must be adapted to other systems.

```
INSERT INTO A(i, gen, a)
SELECT
 P.i, g,
 dbo.histograms(dbo.famcemRow::LoadValues(k,P.P1,...,P.Pk))
FROM P
WHERE P.gen = g;
```

Finally we aggregate the results into the sufficient statistics per cluster, N_h, L_h and Q_h. It is important to notice that in a particular cluster h N_h is a scalar quantity, L_h is a d-dimensional vector and Q_h is a $d \times d$ matrix. However in the simplified model we are solving, Q_h is a diagonal matrix, so it can be stored in a d-dimensional array.

```
SELECT
   A.a,
   SUM(1.0) N,
   SUM(X.X1) L1,..., SUM(X.Xd) Ld,
   SUM(X.X1*X.X1) Q1,...,SUM(X.Xd*X.Xd) Qd
FROM X
JOIN A ON X.i = A.i
WHERE A.gen = g
GROUP BY A.a;
```

This is the final query of the E-step and the values of N, L and Q are returned to the Master. Since all the processing required by the E-step has been performed inside the DBMS, using its efficient disk reading routines the calculation has been accelerated considerably. Furthermore, since data never left the DBMS and only the sufficient statistics are transmitted between computers, data security and privacy are preserved.

As mentioned before, we used more than one DBMS to compute the E-step. This was accomplished by spawning m concurrent threads, one per worker and dividing the load evenly between them. Since N, L and Q are additive functions, the results can easily be accumulated at the end of the calculation, using a simple critical section with a spin lock before terminating the threads. It is worthy of note that workers were assigned in a random order, so it was unlikely that the same server would work on the exact same data in successive iterations. This helped prevent the introduction of some bias to the calculation. The data set and the temporary tables are horizontally partitioned using the primary key and exploiting the fact that the tables had a clustered index.

4.4. The Maximization Steps

As mentioned before, in the classic EM the M Step is performed only once per iteration. In FAMCEM we maximize $\lfloor n/\psi \rfloor + 1$ times in a single iteration. This is allowed by the size of

the data set and the fact that part of a very large table, is still a table large enough to be representative of the whole. There are two options for performing the M-Step: Inside the DBMS using SQL queries, or as part of the Master's program. We have chosen the second option to make it double as the accumulation part of the threaded algorithm. Furthermore, compared with the time that it takes for one E-Step to complete (even a fractional one as we do here) the time required to get the sufficient statistics out of the DBMSs and into the Master, is negligible.

In the M-Step we retrieve the results for the computation of sufficient statistics from the databases. Given the relatively small size of these quantities, they are easy to transmit and they can be manipulated in memory, with no disk accesses. They are accumulated in the critical section of the program and added to the quantities previously calculated. Then we update the parameters of the clusters using formulas 9, 11 and 10. We also compute the log-likelihood of the partial model, as a guide to check the evolution of the system (this partial log-likelihood is never used as a parameter). Finally the parameters are sent back to the DBMSs to be used in the next E-Step.

```
INSERT INTO ParamTable
VALUES
 (gen,C1_1,C2_1,...,Cd_k,...,R1_1,R2_1,...,Rd_k)
  g,c1_1, c2_1,...,cd_k,r1_1,...,rd_k ;
```

where `g,c1_1, c2_1,...,cd_k,r1_1,...,rd_k` denote the values to be entered into the table.

4.5. The Sampling Steps

After the algorithm has converged to a local minimum, we replace the M-Steps with S-Steps (Sampling Steps). The structure of the S-Step is analogous to the M-Step: N, L and Q are retrieved from the databases, they are combined and added to the sufficient statistics already in memory and new C_j and W_j are calculated. As previously mentioned R_j is not updated during the S-Step to avoid adding extra complexity to the problem. Now the Monte Carlo detuning explained in 4 can be performed, using the log-likelihood as the maximization parameter.

The de-tuning of each centroid is performed as follows: For each centroid we use a normal distribution function to generate a new centroid, near the original one using a random number generator with a normal distribution. The width of the normal distribution is given by $\sqrt{R_j}$ to insure that while the new parameters will be different from the original, they will not be too far apart.

Once the new centroid parameter is generated we compare the log-likelihood of the model with the original parameters and the likelihood of the new parameter. If there is a positive improvement (positive in the sense of beneficial) we accept it and use it in our `ParamTable`. If the change is detrimental to the log-likelihood, we roll a new random number and use it to decide whether to accept the change or not based on it, in what amounts to a 'coin flip'. It is here where the Monte Carlo randomization happens: We allow the temporary worsening of our parameters in hopes of finding a faster way to the global extrema. This process is summarized in Algorithm 2.

Algorithm 2 The De-Tuning Algorithm

Input: $C = \{C_1, C_2, \ldots, C_n\}$ and $R = \{R_1, R_2, \ldots, R_n\}$
Output: $C' = \{\tilde{C_1}, \tilde{C_2}, \ldots, \tilde{C_n}\}$
  for $j = 1$ **to** k **do**
    for $h = 1$ **to** d **do**
      Generate $\tilde{C_{hj}}$ using a normal distribution of width $\sqrt{R_ij}$
    end for
    Calculate $\Delta_L = L(\tilde{\Theta}) - L(\Theta)$
    if $\Delta_L > 0$ **then**
      Use $\tilde{\Theta}$
    else
      Generate a random number r
      if $\exp(\beta\Delta_L) > r$ **then**
        Use $\tilde{\Theta}$
      else
        Use Θ
      end if
    end if
  end for

The β parameter can be set by the user to fine-tune the speed of convergence of the algorithm. It could be convenient for some data-sets to allow for less probable centroids during the initial iterations to sample the solution space efficiently, and to reduce the randomness when we are confident we are near the global extrema.

4.6. Time Complexity

In the worst case scenario the time complexity is $O(ndk)$ per iteration. The number of iterations for the EM algorithm is affected b the distribution of data and the shape of the clusters. However, per force Monte Carlo style algorithms need a much larger number of iterations, than more conventional EM, since it much more difficult to achieve stability and convergence.

Two important aspects must be emphasized: Most of the time is spent during the E-step, and this time is dominated by disk access. The accumulation step, since is only $O(\sqrt{n}kd)$ and performed in memory is negligible in comparison.

5. Experimental Validation

The software used in this section can be found in Ordonez (2014) along with instructions for compiling and running it.

We evaluated our proposed method by using a real data-set from the UCI Machine Learning Repository (Bache and Lichman (2013)). The data-set, represented measurements of electric power consumption in one household with a one-minute sampling rate over a period of almost 4 years. The data-set has 9 dimensions that include date of the measurement, time and 4 electrical measurements and 3 dimensions that represented characteristics of

Table 3: DBMS and Hardware characteristics.

	Master	Worker 1	Worker 2	Worker 3
DBMS	SQL Server 2012	SQL Server 2008R2	SQL Server 2005	SQL Server 2008
Operating Sys.	MS Windows 7 Pro	MS Windows Server 2003, Enterprise Edition	MS Windows Server 2003, Enterprise Edition	MS Windows Server 2003, Enterprise Edition
Processor	Intel Q9550, 2.83 GHz	Intel Dual Core CPU 3.0 GHz	Intel Dual Core CPU 3.0 GHz	Intel Quad Core CPU 2.153 GHz
RAM	4 GB	4 GB	4 GB	4 GB
Hard Disk	320 GB; 7200RPM	1 TB; 7200 RPM	1 TB; 7200 RPM	650 Gb; 7200 RPM

Table 4: Log-Likelihood Improvement $k = 10$.

n	FREM	FAMCEM
$10K$	-41.09	-22.22
$100K$	-44.93	-42.32
$1M$	-39.11	-36.99

Table 5: Time to convergence $k = 10$.

n	Time [s]	Iterations
$10K$	32	31
$100K$	280	32
$1M$	2833	60

the households. For our experiments we used the 5 non-categorical dimensions: $d = 5$. It was expanded to occupy 10 million rows of data and occupied approximately 400 Mb in the DBMS by performing a Cartesian product of the data set with itself. The hardware configuration is described in Table 3. We used a personal computer as the master, since no great stress was placed on it as part of the system. The workers were three rack servers with similar profiles to insure all threads finish at roughly the same time. We only used three servers since this was a proof of concept.

In order to give a measure of the improvement of our algorithm we compared the value of the log-likelihood function to the one obtained using FREM Ordonez and Omiecinski (2002) (Table 4). We think the comparison is a fair one, since both algorithms work with massive data inside a DBMS. The main difference is that while FREM is able to show convergence in a few iterations, we require an order of magnitude more passes of the data set to be confident of convergence. In the three cases we compared, FAMCEM reached a state of higher likelihood, suggesting that the solutions are of higher quality. We must remark that the log likelihood function is not the best way of comparing solutions, but it is an indicator of the goodness of the fit. Figure 1 we show the behavior of the Log-Likelihood function at the end of each iteration for three different size of problem. We show an improved behavior in all three cases.

Table 5 shows the time required for the algorithm to finish its calculation. We used a very simple criterion for stopping the iterative process: If no improvement was found after 20 iterations, the algorithm terminated. We plan to introduce a more sophisticated mechanism in the future. As we know, the time for shorter datasets is dominated by the sequential part of the process (transmission delays) so when going from $n = 10k$ to $n = 100k$, the algorithm required less than twice the time. However when dealing with a larger dataset, $n = 1M$ the time increases by an order of magnitude. We believe there is margin for improvement in this area.

We tested our algorithm with one, two and three workers, and for three different table sizes: 100 thousand rows, 1 million rows and 10 million rows. We calculated the time required per iteration for different numbers of clusters considered, from three to twenty five. These results are condensed in tables 6, 7 and 8. We cleared the buffers of the DBMSs and did not operate in main memory. Future experiments will be performed using a geometric progression of servers $(1, 2, 4, 8, \cdots)$. In this set of experiments we concentrated solely in the time improvements over quality of solutions.

Table 6: Time per iteration in seconds $n = 100000$.

k	1 Worker	2 Workers	3 Workers
3	10	9	8
5	11	7	7
10	12	10	9
15	14	10	10
20	16	11	13
25	17	15	14

Table 6 shows that for small datasets there is very little change in the time required per iteration. The time per iteration is dominated by the transmission times, therefore the addition of more workers will only affect the results negatively.

Table 7: Time per iteration in seconds $n = 1000000$.

k	1 Worker	2 Workers	3 Workers	Acceleration
3	73	49	37	1.92
5	74	50	38	1.94
10	84	55	45	1.86
15	93	59	50	1.86
20	101	65	52	1.94
25	111	71	57	1.94

Table 7 shows a more interesting example. We show the time behaviour for 1 million rows of data. We can clearly see that, even though the addition of more servers is reducing the time per iteration, the decrease is not linear. Accumulation and transmission times are sequential, and with larger number of clusters and more dimensions, the Amdahl's law will prevent us from improving the results. We also present a short comparison for the times of

Table 8: Time per iteration in seconds $k = 10$.

Workers	$n = 10K$	$n = 100K$	$n = 1M$	$n = 10M$
1	1	12	84	656
2	4	10	50	459
3	5	9	38	309

operation for three dataset sizes. We can see that the results are consistent for the larger datasets, the introduction of two more servers reduces the time per operation is reduced by about one half.

We compared our results to a similar algorithm implemented in R, presented in Liang (2009). The largest dataset we could test had only 10000 rows. In this dataset, the R package took 61 seconds per iteration, for 3 and 5 clusters and 62 seconds when considering 10 clusters. We can see that R was 60 times slower than FAMCEM, even when using only one worker server.

Table 9: Time per iteration in seconds for R and FAMCEM.

k	n	R	FAMCEM
3	10000	61	1.0
5	10000	61	1.1
10	10000	62	1.3

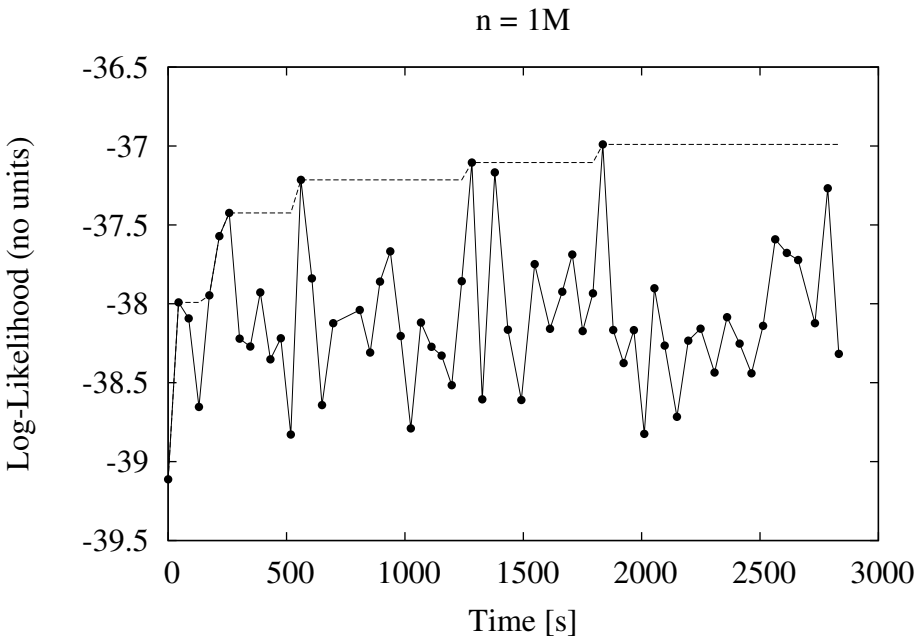

Figure 1: Log-Likelihood vs time.

Finally we calculated the time to convergence of an EM algorithm in R with a similar problem with FAMCEM. The dataset had $n = 10k$ rows and we considered $k = 10$ clusters. as can be seen in Table 5 for that size of problem FAMCEM converged in 32 seconds, while R took 637 seconds, almost 20 times more, even though our algorithm is a non-deterministic one.

6. Related Work

Considerable work has been invested in improving the EM algorithm and into accelerating its convergence. Bradley et al. (1998) show a method for a scalable framework that needs at most, one scan of the database. Ueda et al. (2000) present a survey of several approaches from the Machine Learning community, with mathematical justifications for them. MC techniques use random walks to explore the solution space. In general it is found that, even though MC algorithms take a long time to converge, they reach very good solutions by avoiding being trapped in local extrema. While in recent years the problem of accelerating the EM algorithm has been studied, (Kumar et al. (2009) and Thiesson et al. (2001) among others) not much has been done to speed-up the convergence of Gibbs Samplers for mixtures of Gaussians. Matusevich et al. (2013) shows an implementation of FAMCEM in C++.

There has not been much focus in recent years on building machine learning algorithms in SQL, mainly due to its degree of difficulty when compared to C++. Most efforts, like Lin and Kolcz (2012), concentrate on crafting user defined functions that, while general in scope, are specific to the DBMS they use. The EM algorithm was coded using SQL queries in Ordonez and Cereghini (2000), proving that it is possible to program a clustering algorithm entirely with queries. The main difference between this approach and ours is that it is an iterative one while ours is incremental. Pitchaimalai et al. (2008) present an in-depth study of distance calculations with SQL. More recently Sun et al. (2013) proposed an extension to SQL called `CLUSTER BY` to calculate clustering of data and implemented it in Posgress.

7. Conclusions

We present an efficient parallel implementation of an MCMC Clustering algorithm in remote servers. The algorithm uses a C++ program to generate SQL queries that are evaluated in one or more servers concurrently, allowing for a considerable speed-up of the time spent per iteration. We use SQL queries instead of UDF's because of their generality and portability. In our comparisons with a similar algorithm written in R, we see a hundred-fold improvement in small data sets. We project an even better performance differential with larger datasets, where the efficiency of reads from physical media gives the DBMS a definite edge in speed. Our algorithm is general enough that with very few changes it can be implemented in different database systems, and is independent of the operating system of the worker machines. The algorithm's time complexity is linear in the size of the data set and we see an almost linear speedup when increasing the number of workers. Preliminary experiments prove the worth of the approach.

Future avenues of research include the evaluation of the correctness of the algorithm using synthetic data and more rigorous measures of model quality, the implementation in an array stored relational database using native operators, the comparison with other clustering algorithms and the processing of massive data sets in a large parallel cluster. Finally we would like to explore the influence of the different parameters in the correctness of the results, and implement a simulated annealing scheme to improve convergence times.

References

K. Bache and M. Lichman. UCI machine learning repository, 2013. URL http://archive.ics.uci.edu/ml.

James J Binney, NJ Dowrick, AJ Fisher, and M Newman. *The theory of critical phenomena: an introduction to the renormalization group*. Oxford University Press, Inc., 1992.

P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. ACM KDD Conference*, pages 9–15, 1998.

David B Hitchcock. A history of the Metropolis–Hastings algorithm. *The American Statistician*, 57 (4):254–257, 2003.

Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. MCDB: a monte carlo approach to managing uncertain data. In *Proc of ACM SIGMOD*, pages 687–700. ACM, 2008.

NSLP Kumar, Sanjiv Satoor, and Ian Buck. Fast parallel expectation maximization for gaussian mixture models on GPUs using CUDA. In *HPCC'09.*, pages 103–109. IEEE, 2009.

Liwen Liang. On simulation methods for two component normal mixture models under Bayesian approach. *Uppsala Universitet, Project Report*, 2009.

Jimmy Lin and Alek Kolcz. Large-scale machine learning at twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 793–804. ACM, 2012.

D. S. Matusevich, C. Ordonez, and V. Baladandayuthapani. A fast convergence clustering algorithm merging MCMC and EM methods. In *Proc. of ACM CIKM Conference*, pages 1525–1528. ACM, 2013.

C. Ordonez. DBMS group software repository, 2014. URL http://www2.cs.uh.edu/~ordonez/co_research_software.html.

C. Ordonez and P. Cereghini. SQLEM: Fast clustering in SQL using the EM algorithm. In *Proc. ACM SIGMOD Conference*, pages 559–570, 2000.

C. Ordonez and E. Omiecinski. FREM: Fast and robust EM clustering for large data sets. In *ACM CIKM Conference*, pages 590–599, 2002.

S. Pitchaimalai, C. Ordonez, and C. Garcia-Alvarado. Efficient distance computation using SQL queries and UDFs. In *IEEE HPDM*, pages 533–542, 2008.

Kristopher J Preacher and James P Selig. Advantages of monte carlo confidence intervals for indirect effects. *Communication Methods and Measures*, 6(2):77–98, 2012.

Peng Sun, Yan Huang, and Chengyang Zhang. Cluster-by: An efficient clustering operator in emergency management database systems. In *Web-Age Information Management*, pages 152–164. Springer, 2013.

Bo Thiesson, Christopher Meek, and David Heckerman. Accelerating EM for large databases. *Machine Learning*, 45(3):279–299, 2001.

N. Ueda, R. Nakano, Z. Ghahramani, and G. Hinton. SMEM algorithm for mixture models. *Neural Computation*, 12(9):2109–2128, 2000.