Available online at www.sciencedirect.com
ScienceDirect
Pattern Recognition 40 (2007) 784–795
PATTERN RECOGNITION
THE JOURNAL OF THE PATTERN RECOGNITION SOCIETY
www.elsevier.com/locate/pr

# Gradual model generator for single-pass clustering

Ismo Kärkkäinen*, Pasi Fränti

*Speech and Image Processing Unit, Department of Computer Science, University of Joensuu, FIN-80101, Finland*

Received 20 July 2005; received in revised form 19 May 2006; accepted 22 June 2006

**Abstract**

We present an algorithm for generating a mixture model from a data set by converting the data into a model. The method is applicable when only part of the data fits in the main memory at the same time. The generated model is a Gaussian mixture model but the algorithm can be adapted to other types of models, too. The user cannot specify the size of the generated model. We also introduce a post-processing method, which can reduce the size of the model without using the original data. This will result in a more compact model with fewer components, but with approximately the same representation accuracy as the original model. Our comparisons show that the algorithm produces good results and is quite efficient. The whole process requires only 0.5–10% of the time spent by the expectation-maximization algorithm.


*Keywords:* Clustering; Gaussian mixture model; Single-pass; Large data sets

## 1. Introduction

Clustering algorithms are used to find structure in data. The majority of the existing algorithms store all of the data in memory, or make several passes over the data, which can restrict their usefulness in the case of large data sets that exceed available memory resources. Algorithms designed specifically for very large data sets must be able to read the data as few times as possible, preferably only once. A desired property is to have independence of the order in which the data are processed. Random access to the data may be costly, and therefore, algorithms requiring it can become too slow for practical use.

### 1.1. Algorithms for large data sets

Existing approaches reduce the size of the data set by sampling, by representing several data points in alternative manner or by handling it in small chunks. For example, *BIRCH* [1] generates a tree of *clustering features* or *sub-clusters* using a fixed amount of memory. These can then be clustered independently using any other algorithm. Algorithm by Bradley et al. [2] processes the data into summarized, retained and discarded sets that are clustered separately. The shapes of the clusters depend on what clustering algorithm is used after the data have been processed. A hierarchical clustering method is used in Ref. [1], and *k-means* is used in Ref. [2].

Random sampling has been used to speed up clustering in *CURE* [3] and *CLARANS* [4]. This requires random access to the data being clustered, and relies on the sample being large enough to represent the data well. Density-based methods such as *DBSCAN* [5] identify dense areas as clusters. Data can also be ordered so that the cluster structure will be more visible, as in Ref. [6]. Density-based methods can find arbitrary-shaped clusters. Some of the density-based approaches require an indexing structure to perform $k$ nearest neighbor searches to obtain the density estimate. The problem of finding clusters in subspaces is discussed in Refs. [7,8].

* Corresponding author. Tel.: +358 13 251 7904; fax: +358 13 251 7955.
*E-mail addresses:* ismo.karkkainen@cs.joensuu.fi (I. Kärkkäinen), pasi.franti@cs.joensuu.fi (P. Fränti).

0031-3203/$30.00 
doi:10.1016/j.patcog.2006.06.023

These approaches produce either a partition of the data or they use a centroid model to represent the data.

### 1.2. Algorithms that generate GMM

There are a few algorithms that generate a Gaussian mixture model from a large data set. The *Online EM* algorithm by Sato and Ishii is described in Ref. [9]. It starts with an initial solution and then updates the existing components, deletes them or adds new ones if necessary. The aim is to model the recent data by giving priority to newer events and gradually forgetting the older ones. Components that do not include any newer data points may be eventually deleted, and if a data point appears at a region where there are no components, a new component can be generated using information from existing components. In our case, we want to model the entire data set, and, therefore, we apply the Online EM so that we only modify the initial solution without adding or removing components.

*Scalable EM* by Bradley et al. [10] has been modified from the EM algorithm [11,12] to operate with summarized data along with normal data as in Ref. [2]. The algorithm consists of the EM algorithm, *k*-means and agglomerative clustering algorithm run on the summarized data set. Small, tight subsets of the data are found by *k*-means, which are then summarized. This reduces the number of distinct data points. The authors refer to this as secondary summarization. Joining nearest neighbors further decreases the amount of data as long as the variances of the clusters do not exceed a given limit. The Scalable EM algorithm iterates EM, *k*-means and the agglomerative clustering algorithm until all data have been read and the final model is produced. The authors also perform primary summarization using the GMM produced after each run of the EM algorithm, but this relies on the GMM being a well-fitting model for the data. We omit this step, as there is no guarantee that the summarization of the data using the points close to the component centers would produce good results. Specifically in cases where a component spans several clusters the summarized component will be quite harmful for the final result.

Some other work has also been done to speed up the EM algorithm by performing the model update step several times per pass [13], but several updates do not eliminate the need to perform several passes over the data. The modified and the standard EM algorithm need to store all the data or read them into memory once per iteration, so in practice they are not suitable for data sets larger than the available memory.

In this paper the aim of modeling the data vectors $x_i$, $i = 1, \ldots, N$, using a Gaussian mixture model is to obtain a model of $K$ components consisting of mean vectors $m_j$, covariance matrices $c_j$ and the component weights $w_j$, $j = 1, \ldots, K$. The goodness-of-fit of the model is measured by the average *log-likelihood* of the data [12,14]. Given the probability density of a point $x_i$ with respect to component $j$, $P_{ij}(x_i|w_j, m_j, c_j)$, the average log-likelihood is computed as

$$L = \frac{1}{N} \sum_i \log \left( \sum_j P_{ij}(x_i|w_j, m_j, c_j) \right). \quad (1)$$

### 1.3. Our contribution

In this paper, we propose an algorithm that converts the data into a mixture of multivariate Gaussian distributions by selecting small, compact subsets of the data. Data that lie close to the generated Gaussians are used to update the model. The main idea is that the model is generated from the data, instead of modifying an initial model to fit the data. The results indicate that the proposed method reaches virtually the same clustering performance as the standard EM algorithm in terms of log-likelihood, but uses only 0.5–10% of the time that the EM algorithm uses. Furthermore, our method requires that only a small part of the data needs to be stored at once and that every point needs to be read only once, so it can be used on data sets that are larger than the available memory. Also, as shown in Section 3, we found that the algorithm is robust in regard to the order in which the data points are read. Hence the data can be passed to the algorithm in the order in which it is stored, and no random access to the data is necessary. It is therefore highly useful for single-pass clustering of large data sets.

## 2. Gradual model generator algorithm

The proposed algorithm consists of two stages: (1) single-pass model generation, and (2) model simplification as a post-processing step. First, a model is generated from the data by the single-pass algorithm. The model can then be post-processed in order to obtain a simpler representation, if desired. The post-processing can be performed without the original data. The result is a model that represents the same data but with fewer components. The benefit of using two stages is that the first stage's model takes less time and memory to process than the entire data set. The steps are illustrated in Fig. 1.

The model generation algorithm does not need an initial solution. Instead, it builds the model from scratch using data points. Every data point is used to alter the model only once: either to update the existing model or to create new components. All the points that are described well enough by the model are used to immediately update the model. The remaining points are stored for later use. A fixed-size buffer is maintained for this purpose. The points remain in the buffer until they are used to generate a new component, or are used for updating the model once the points are described well enough by the model because of direct updates and the new components added into the model. When the entire data set has been read and processed and the buffer is empty, the algorithm outputs the resulting GMM.

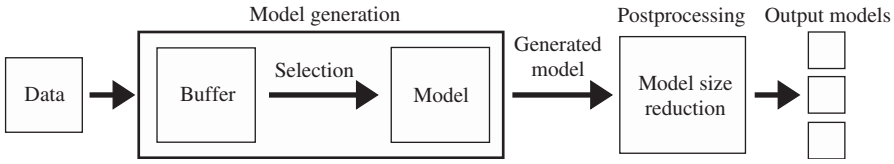


Fig. 1. General outline of the single-pass clustering process.

The first stage of the algorithm tends to generate a model that has an excessive number of components. Although this does not affect the representation accuracy of the model, a more compact model with a smaller number of components could be desired for practical purposes. However, it is not a simple task to restrict the size of the model during the first stage without affecting the quality of the model. We have therefore decided to let the model grow and over-fit the data, and leave the reduction of the model size completely to the second stage of the process. If the model grows too large to be handled in memory, we then simply save the current model, start a new one from scratch, and continue to process the rest of the data. The saved models can later be joined together by adjusting the weights of the components in proportion to the number of points used to generate each model.

The goal of the second stage is to find the actual clusters. For this purpose, an ordinary clustering algorithm can be applied to the components of the model instead of using the original data points. We apply here a hybrid of the EM and *k-means* [15] algorithms to allow us to find ellipsoid-shaped clusters.

The second stage does not require the original data. However, if the optimal number of components must also be solved with regard to the original data, then the original data are most likely needed. Otherwise the user has to rely on the differences between the models as a guideline for selecting the model.

### 2.1. Generating the model

The basic idea of generating the model is to use subsets of the data to create components of the model in regions that have the most points in a small area. Any points that are close to the model are used to update the components of the model directly. Section 2.1.1 describes the process to create new components and how the buffer is used. Section 2.1.2 describes the model update procedure and describes when a point is used to update the model directly.

The pseudocode for the algorithm is given in Fig. 2. The model and the data buffer are initially empty. The buffer first fills with new points. Then the first component is created using a subset of the points from the buffer. Once the model has components, the fit of a new point to the model is checked, and when there is a sufficient fit, the model is updated accordingly and the point is discarded. If the point does not fit, it is stored into the buffer.

```
GMG(data set, buffer size, points to select): GMM
Initialize data buffer and create empty model.
WHILE data points left and data buffer is not empty:
    Read next point.
    IF point fits into model THEN
        Update model
    ELSE
        Insert point into data buffer.
    IF data buffer is full or no data is left, THEN
        Check if any points in buffer fit into model.
        IF some points fit into model THEN
            Update model
        ELSE
            Select points from buffer.
            Create a new component.
            Add it into the model
RETURN model.
```

Fig. 2. Pseudocode of the algorithm.

#### 2.1.1. Creating new components

A new component is created by selecting a point and its $k-1$ neighborhood from the buffer so that the distance from the point to the farthest point in the neighborhood is minimized. The goal is to find a group of points that lie in a small area, and therefore, are likely to belong to the same cluster. However, enough points must be selected so that the covariance matrix can be inverted. The selected $k$ points are used to generate a new component, which is then added to the model, see Fig. 3 for an example. The selected points are labeled with dots ($\cdot$) and the points remaining in the buffer are labeled with plusses (+). The ellipse shows the location of the new component. The selected points are then removed from the buffer.

In practice, we maintain the sum of the selected points along with the sum of their outer products, as proposed in [13,1]. Let $u_{ij}$ be the weight with which the vector $x_i$ is added to the component $j$. The weighted sum of the vectors for component $j$ is

$$M_j = \sum_i u_{ij} x_i. \tag{2}$$

The weighted sum of the outer products for row vectors $x_i$ is

$$C_j = \sum_i u_{ij} x_i^{\mathrm{T}} x_i. \tag{3}$$

If we use only diagonal covariance matrices, it is sufficient to store just the diagonal elements. Furthermore, we need

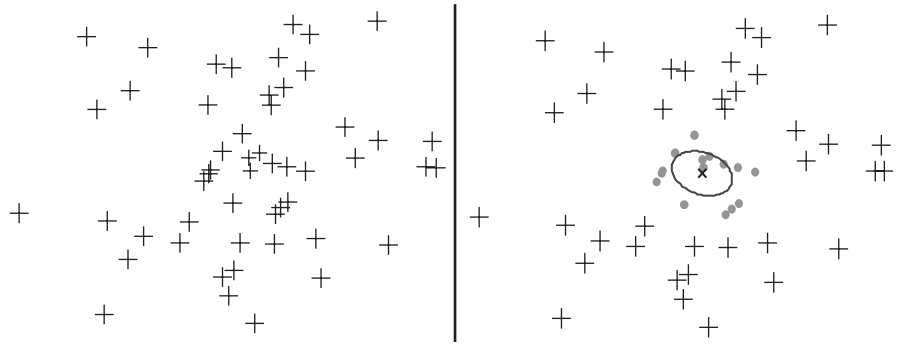

Fig. 3. The data and model before (left) and after (right) adding a new component.

the sum of weights of vectors for each component:

$$U_j = \sum_i u_{ij}. \tag{4}$$

Vectors that are used to create the new component have a weight of 1 for the new component and a weight of 0 for others. The mean vector $m_j$, covariance matrix $c_j$, and weight $w_j$ for the component $j$ are calculated as

$$m_j = M_j / U_j, \tag{5}$$

$$c_j = \frac{C_j}{U_j} - \frac{M_j^{\mathrm{T}} M_j}{U_j^2}, \tag{6}$$

$$w_j = U_j \Big/ \sum_j U_j. \tag{7}$$

At the end when all data points have been read, the remaining points in the buffer are processed. If necessary, more components are created. However, it may turn out that only border points of existing clusters and possible outliers are left and that a component formed from them would cover several clusters. Therefore, before creating a new component, the algorithm tests that no existing component would lie inside the area covered by the potential new component. If this happens, we consider the points as leftover points, and discard them instead of making a new component out of them.

The test involves computing the mean vector $m$ of the selected points. For each component mean $m_j$, we compute the difference vector $y_j$ between $m_j$ and $m$. We then compute the scalar projections of $y_j$ to all vectors from $m$ to the selected points and divide them by the length of $y_j$. If all positive scalar projections are less than one, the selected points are considered to be surrounding the mean vector $m_j$ and they are discarded as leftover points.

The buffer size ($b$) and the number of points ($k$) used to generate a new component are the parameters of the algorithm. The buffer size is mainly a question of processing time: we should use as large buffer size as processing time permits. The parameter $k$ should be small enough so that the new components would consist of points from single cluster. However, if $k$ is set too small, we could create too many components, which will result in a longer processing time (see Section 3.4.2). A larger buffer size means $k$ can also be higher and that the points can still be selected from the same cluster. According to our experiments, $k = 10$–$15$ points is a good choice for two-dimensional data. In the case of higher-dimensional real data sets, our rule of thumb is that $k$ should be set to approximately two times that of the dimensionality of the data.

### 2.1.2. Model update

Primarily we try to include new points to the existing model. For every input point, we first test how well it fits into the model. If the probability density is higher than a given threshold determined by the GMG algorithm, the model is updated the same way as in the EM algorithm. We compute the update weight of the point with regard to every component by normalizing the probability densities. Then we update the weight, mean and covariance matrix of each component in proportion to the update weight. The probability density for vector $i$ with regard to the component $j$ in $d$-dimensional vector space is

$$P_{ij}(x_i | w_j, m_j, c_j) = \frac{w_j}{\sqrt{(2\pi)^d |c_j|}} \mathrm{e}^{-1/2(x_i - m_j) c_j^{-1} (x_i - m_j)^{\mathrm{T}}}. \tag{8}$$

The acceptance threshold for the model updates is revised every time a new component is created. This is done by using the points that were selected for the new component. We take the minimum probability density of the selected points with respect to the entire model. The new limit is lower of the old limit and the minimum. Any point that has a higher probability density than the limit is accepted for direct model update. Those points are described better by the model than the point that had the limit probability density. If a point is accepted for model update, the update weights

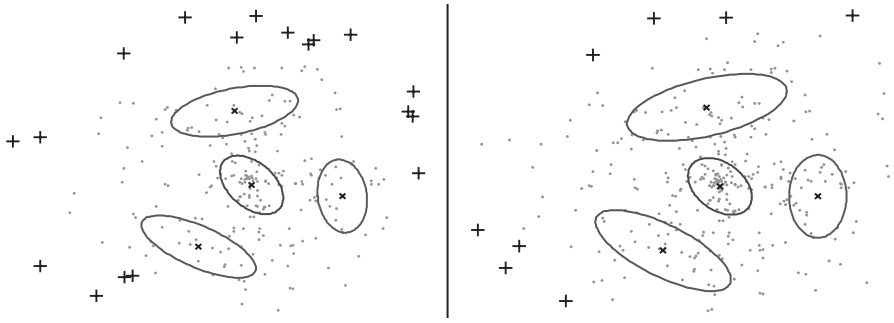

Fig. 4. Model before (left) and after (right) updates.

$u_{ij}$ of the point are

$$u_{ij} = \frac{P_{ij}(x_i|w_j, m_j, c_j)}{\sum_s P_{is}(x_i|w_s, m_s, c_s)}. \quad (9)$$

The model updates, however, are not made immediately. Instead, we store the points and perform several updates at once, as in Ref. [13]. In our case, the model is updated after 100 new points have been accepted for direct update. In practice, the sums (2)–(4) are computed first and the mean vector, covariance matrix and component weight can be obtained when needed.

Once the data buffer fills up, we try to update the model with the contents of the buffer. This is done because points may be accepted for direct model update after the limit of acceptance has changed, or after new components have been added. This limits the growth of the number of components in the model since we can postpone creating a new component. In practice, adding points directly into the model could be handled just by testing the points in the buffer, but that would be significantly slower since the same points that were not accepted would be tested for direct update repeatedly.

Fig. 4 demonstrates a series of model updates. The region is the same as in Fig. 3 but at a later stage in the process. Experimentally, we have found that 84% of the points are processed immediately through direct model update and are never stored in the buffer. Of the remaining points, 13% are stored in the buffer and used later for direct model update. Only 3% of the points are used for creating new components. The points that are stored in the buffer are tested for direct update 34 times, on average.

### 2.2. Postprocessing the model

The model obtained using the GMG can have an unnecessarily high number of components, see, for example, Fig. 5. It also shows that despite a higher number of components, the major modes in the contour plots of the probability density distributions are in the same places in both plots. In order to determine if the data are clustered, we can use adapted clustering algorithms to find the clusters in the generated model. We use a hybrid of EM and $k$-means to find ellipsoidal clusters. The hybrid algorithm does not require the original data as input.

The basic idea of the adapted $k$-means (AKM) algorithm is to treat each component of the input GMM as a data point. The result is a GMM. Each component of the input model is used to update only one component of the resulting GMM, in the same way that one data point in $k$-means is used to update only one centroid. An EM algorithm for clustering features of *BIRCH* algorithm [1] is presented in Ref. [18].

The initialization of the output GMM is done by picking the desired number of components randomly and by fixing the weight sum to one. We use Bhattacharyya-distance [16,17] between the components of input and output models to determine the closest component in the output model

$$D((m_j, c_j), (m_k, c_k)) = \frac{1}{8}(m_j - m_k)^{\mathrm{T}}\left(\frac{c_j + c_k}{2}\right)^{-1}(m_j - m_k) + \frac{1}{2}\ln\frac{|\frac{1}{2}(c_j + c_k)|}{\sqrt{|c_j| + |c_k|}}. \quad (10)$$

Updating the components is done using the sums of Eqs. (2)–(4). They can be computed from the results of Eqs. (5)–(7) by inverting the procedure used to compute the values used in the GMM. Using $T_j$ for the number of points in one group and $T$ for the total number of points we get

$$T_j = w_j T, \quad (11)$$

$$M_j = m_j T_j, \quad (12)$$

$$C_j = T_j\left(c_j + \frac{M_j^{\mathrm{T}} M_j}{T_j^2}\right). \quad (13)$$

After the sums have been computed, we can use formulas (5)–(7), with $T$'s substituted for $U$'s, to obtain the final model. However, the total number of points, $T$, can be set arbitrarily, e.g. to 1. In that case the component means and weights are already computed, so we need to compute the covariance matrix only.

In order to avoid empty groups, we find for each output component the closest input component and map the input

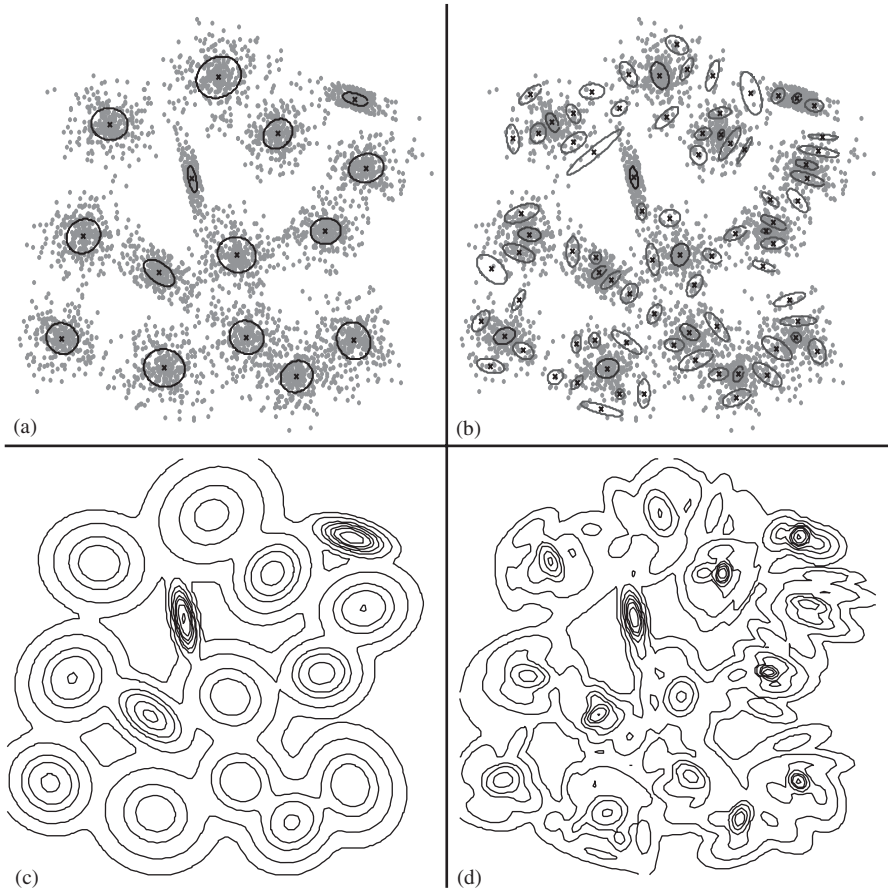


Fig. 5. (a) The GMM generated by EM, (b) by the GMG, and (c and d) the corresponding contours of the probability density distributions.

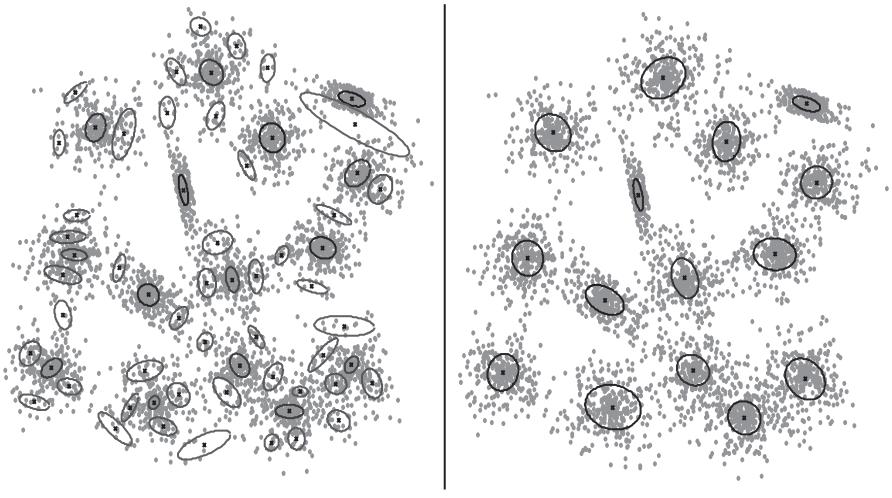

Fig. 6. Model generated by GMG from the original data (left) and the reduced model (right).

component to the corresponding group regardless whether the component would be closer to another output component. Otherwise the algorithm proceeds like $k$-means. Since the input model from GMG is considerably smaller than the original data, this procedure is much faster than using the original data. Fig. 6 shows an example of applying the AKM algorithm. The data points are shown for illustration purposes only. Looking closely, one can see only minor differences between final model of Fig. 6 (right side) and the EM model (Fig. 5a).

## 3. Test results

We tested the combination of GMG and AKM, and compared it against Online EM [9] and Scalable EM [10]. The result obtained by the EM algorithm [11,12] was used as a reference point. We computed log-likelihoods using Eqs. (1) and (8) for the models produced by each method. Likelihood values vary from 0 to $+\infty$, and their logarithm (log-likelihood) can take values from $-\infty$ to $+\infty$. The higher the value, the better the model.

In general, the EM algorithm will give better results than the other methods, but one must consider that if all of the data cannot fit into memory at one time, using the EM algorithm is not straightforward. In that case, one must read the data every time an EM iteration is performed, which will probably be quite slow. We can see from the results in Section 3.4.2 that the standard EM algorithm will take time, so adding the overhead of reading the data from mass storage may in practice result in an unacceptably high processing time.

All tested algorithms, with the exception of EM, depend on the processing order of the data points. We therefore used four different orderings to study the dependency: the original order in which the data sets are available, an increasing order sorted according to the first variable, and two random orderings. Ideally, results with different orderings should not vary much. Due to the fact that Online EM forgets past data, it was only tested with the random orderings. This means that we were more likely to get a model that represents the whole data fairly accurately since new points are expected to be evenly distributed along different regions where there are data. Thus, no part becomes completely forgotten during the process. With sorted data this obviously is not the case, as Sato and Ishii demonstrated [9].

### 3.1. Data sets

We used both synthetic and real data sets, see Table 1. The *BIRCH* data sets include 100 000 points organized by predefined structures [1]. The sets *BIRCH*1 and *BIRCH*2 consist of 100 clusters, whereas *BIRCH*3 was visually determined to be best represented by a GMM with 78 components. The *D*-sets have progressively higher-dimensional data in nine well-separated clusters. Each cluster in a set has the same number of points, and the number of points in each cluster increases linearly as dimensionality increases.

Table 1
Attributes of the synthetic data sets

| Name | Dimensionality | Points | Clusters |
|---|---|---|---|
| *BIRCH*1 | 2 | 100 000 | 100, 10 times 10 grid |
| *BIRCH*2 | 2 | 100 000 | 100, along sine curve |
| *BIRCH*3 | 2 | 100 000 | 78, approximately |
| *D*2–*D*15 | 2–15 | 1350–10 125 | 9 |

Table 2
Attributes of the real data sets

| Name | Dimensionality | Points |
|---|---|---|
| *CM* | 9 | 68 040 |
| *CT* | 16 | 68 040 |
| *El Niño* | 7 | 93 935 |

The first two real data sets are the Corel Image features previously used in Ref. [19]. We selected the co-occurrence texture (*CT*) and color moments (*CM*) data sets. The third real data set is the *El Niño* data set previously used in the American Statistical Association Statistical Graphics and Computing Sections 1999 Data Exposition. Indices and points with missing attributes have been removed from the *El Niño* data set. All three data sets were obtained from Ref. [20]. The number of clusters and whether the data are clustered at all are unknown. The data sets are summarized in Table 2.

### 3.2. Parameters used with different algorithms

The only parameters given to the EM algorithm are the number of components and a halting threshold for the relative improvement of log-likelihood, which was set to $2^{-16}$.

The Online EM requires the number of components, and also a factor that affects how fast the old data are forgotten. This factor is multiplied with the weight of the point used in the update of the component. We used a factor of 0.01 so that the components would forget old data very slowly.

The Scalable EM requires the number of components, the number of points and components that are stored, a variance limit for the summarization stage, and the amount of how much the limit is increased if no summarization appears to be possible. The number of points and components stored correspond to the amount of memory available for storage. We treated this number as the sum of the number of points and components that are stored. This favors the algorithm since the components take up more space than individual points especially when using full covariance matrices as in these tests. We used 1000 for the synthetic data sets and used 2000 and 4000 for the real data sets. The initial variance limit was estimated directly from the data. Such estimate corresponds to a good guess. If no summarization appeared to be possible within the limit, the limit was multiplied by 1.5 and the summarization was retried. The number of centroids in the secondary summarization phase was set to $n/2D$, where $n$ is the size of the retained set and $D$ is the dimensionality of the data. This meant that the secondary summarization was independent of the number of components, which varied greatly in our tests.

The GMG was run with buffer sizes $b$ of 250, 500, 1000, and 2000 for the synthetic data sets and also 4000 for the real data sets. The number of points selected, $k$, from the buffer to create a component was set to $(D+1)$ multiplied

Table 3
Average log-likelihoods for BIRCH data sets

| Data set | EM | Online EM | Scalable EM | GMG | |
|---|---|---|---|---|---|
| | | | | No rejection test | With rejection test |
| *BIRCH*1 | −7.221 | −7.312 | −7.351 | −7.338 | −7.333 |
| *BIRCH*2 | −7.437 | −8.409 | −7.651 | −7.778 | −7.756 |
| *BIRCH*3 | −7.392 | −8.376 | −8.114 | −8.008 | −7.996 |

Table 4
GMG output model sizes and AKM log-likelihoods for *BIRCH*1

| | | Sizes of generated models | | | | Log-likelihoods after applying AKM | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Buffer size | | | | Buffer size | | | |
| | | 250 | 500 | 1000 | 2000 | 250 | 500 | 1000 | 2000 |
| Points selected for new components | 6 | 271 | 554 | 941 | 1674 | −7.335 | −7.324 | −7.375 | −7.401 |
| | 9 | 135 | 183 | 352 | 732 | −7.311 | −7.342 | −7.342 | −7.363 |
| | 12 | 80 | 116 | 252 | 415 | −7.320 | −7.309 | −7.376 | −7.332 |
| | 15 | 78 | 106 | 157 | 258 | −7.327 | −7.293 | −7.309 | −7.352 |

by 2, 3, 4 and 5 for the synthetic data sets. For the real data sets 20, 30, 40 and 50 points were selected. Furthermore, we tested accepting all generated components and using the rejection test described in Section 2.1.1. The algorithm was run once for each parameter combination for each ordering of each data set.

### 3.3. Results for synthetic data sets

Since the number of clusters was known for synthetic data sets, we compared the GMM with the same number of components as the number of clusters in the data sets as listed in Table 1.

For the proposed method, we show the log-likelihoods obtained by first running GMG and then generating a smaller model using AKM. Due to the high number of tested parameter combinations we show the mean log-likelihood over all parameter combinations. This indicates average performance while keeping the figures readable. For the proposed method we treat separately the cases where the rejection test for new components was used, and where all new components were accepted in order to see what effect the rejection test had.

Table 3 shows the average log-likelihoods for the *BIRCH* data sets, with data in the first random order. Scalable EM produced results of about the same quality as the proposed method, whereas the Online EM produced clearly the worst results with *BIRCH*2 set. When the rejection test was used it appeared to produce slightly better results, on average, although the differences are quite small. The lower result for the *BIRCH*3 data set by Scalable EM might be explained by the difference in the size of the clusters in the data. Very compact but heavy clusters are merged together with their neighbors, and thus, the resulting components lie between clusters.

Table 4 shows the sizes of the models generated by GMG and the mean log-likelihoods obtained using AKM. The variation in log-likelihood is small despite the large differences in the sizes of the models produced by GMG. The results show clearly the effect that the buffer size ($b$) and the number of selected points ($k$) have on the size of the generated model. With small buffer size and high number of selected points there are fewer components than known clusters.

Fig. 7 shows the log-likelihoods for *D*-sets. Since the log-likelihoods varied from 3 to 40 as dimensionality increased, the log-likelihoods relative to that of the best result from EM-algorithm are shown. The data order is the second random order. Scalable EM performed about as well as the proposed method while the results of Online EM became worse as dimensionality increased. These data sets, especially the high-dimensional ones have quite compact clusters that are well separated.

### 3.4. Results for real data sets

Since the number of clusters was not known for the real data sets, we show log-likelihood as a function of the number of components. For GMG, the graphs show the log-likelihoods of models obtained after the model size has been reduced using AKM. With the other algorithms, we generated all the models from scratch. Since the rejection test in GMG appeared to produce slightly better results, on average, or at least it did not produce worse results, it was used in all tests with real data sets.

#### 3.4.1. Log-likelihoods

We compared the log-likelihoods computed using the GMMs obtained from various algorithms with different model sizes. The aim was to see what effect the ordering of the data set has on the results. Tables 5–7 show the average

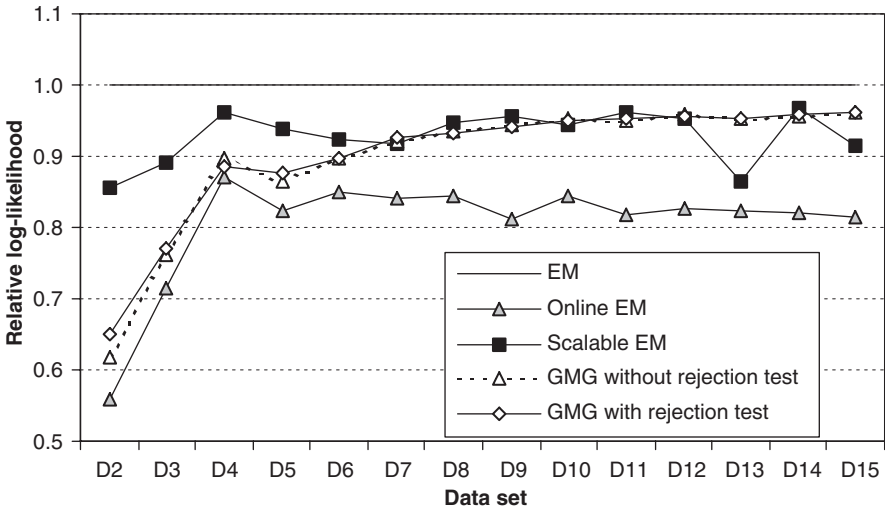


Fig. 7. Average log-likelihoods for *D*-sets relative to result of EM-algorithm.

Table 5
Log-likelihoods for data set *CT* (model size 10)

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | 16.33 | 16.33 | 16.33 | 16.33 |
| Scalable EM | 12.80 | 11.70 | 15.87 | 15.59 |
| Online EM | N/A | N/A | 14.87 | 14.59 |
| GMG(4000,20)/AKM | 14.65 | 14.65 | 14.59 | 14.70 |

Table 6
Log-likelihoods for data set *CT* (model size 30)

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | 17.78 | 17.78 | 17.78 | 17.78 |
| Scalable EM | 14.05 | 11.75 | 16.29 | 15.97 |
| Online EM | N/A | N/A | 16.28 | 15.72 |
| GMG(4000,20)/AKM | 16.24 | 16.17 | 16.23 | 16.26 |

Table 7
Log-likelihoods for data set *CT* (model size 70)

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | 18.65 | 18.65 | 18.65 | 18.65 |
| Scalable EM | 14.02 | 11.82 | 15.91 | 15.75 |
| Online EM | N/A | N/A | 15.85 | 15.52 |
| GMG(4000,20)/AKM | 16.99 | 16.97 | 16.82 | 16.99 |

log-likelihoods for data set *CT* with model sizes of 10, 30 and 70, respectively. Each value is an average over 20 runs. We can see from Table 5 that for small models and random order, Scalable EM outperformed the proposed method and Online EM. However, if the data were not in random order, or the model size was larger, our method outperformed Scalable EM, as can be seen from Tables 6 and 7. None of the methods reached the log-likelihood of the EM algorithm.

Fig. 8 shows a plot of the average log-likelihood as a function of the model size for the *CT* and *El Niño* data sets when the data were in original order. Results for the same data sets in sorted order are shown in Fig. 9. In both cases, Scalable EM performed noticeably worse than the proposed method. However, in Fig. 10, which shows the results for random order, Scalable EM performance improved significantly. It shows quite clearly that Scalable EM actually needed to have the data in random order. The difference between the results for different orderings was much smaller for the proposed method than for Scalable EM. Each curve represents an average log-likelihood of 20 runs over the range from 2 to 100 components. For GMG the buffer size $b$ was 4000 and the number of points selected $k$ was 20, unless otherwise noted.

Tables 8–10 show the average log-likelihoods for the *El Niño* data set and model sizes 10, 30 and 70, respectively. With a small model size, Scalable EM performed better than Online EM or the proposed method provided that the data were in random order. Once the model size grew the results became worse: this happened with the *CT* data set also.

### 3.4.2. Running times

Here, we show how much processing time was spent in obtaining the results. The total time for *CM*, *CT* and *El Niño* data sets consists of the sum of average running times for each number of components. The times for Online EM and Scalable EM show how much time, on average, it took to generate one solution for model sizes of 2–100 components. For GMG/AKM the time includes the time spent by GMG in constructing the model plus the time spent by AKM for model simplification.

Tables 11–13 show the processing times for *CM*, *CT* and *El Niño* data sets. The times are shown for each ordering of

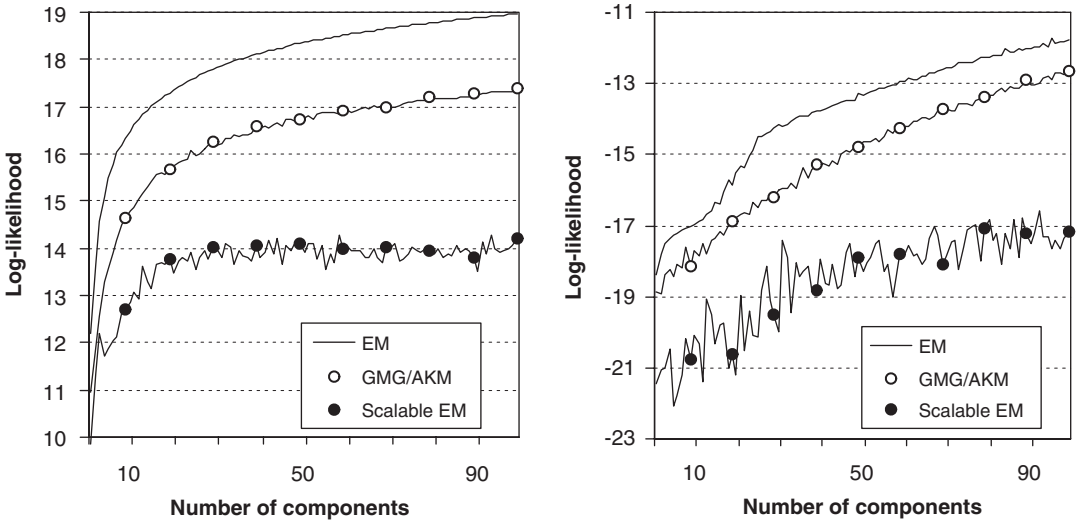


Fig. 8. Average log-likelihoods for *CT* (left) and *El Niño* (right) data sets, data in original order.

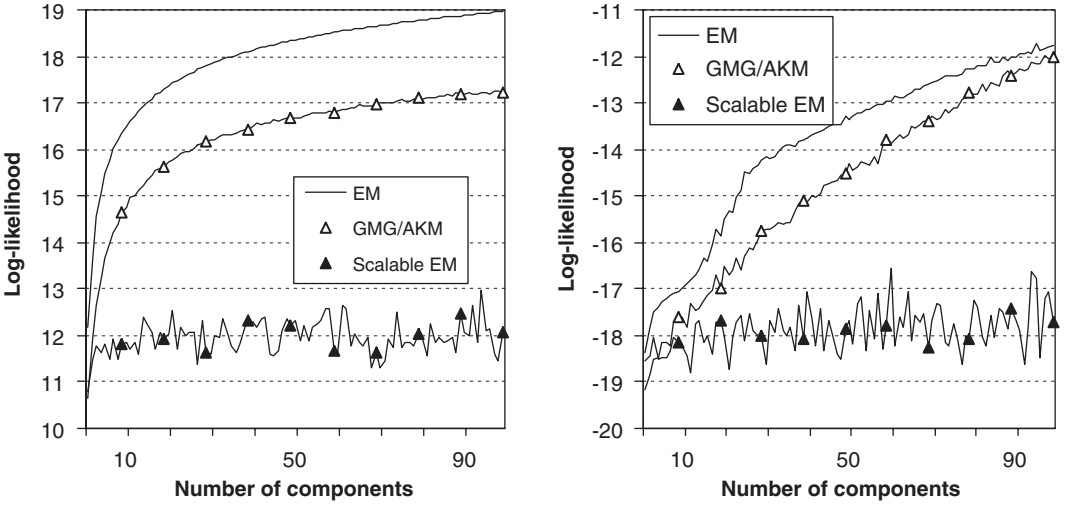


Fig. 9. Average log-likelihoods for *CT* (left) and *El Niño* (right) data sets, data in sorted order.

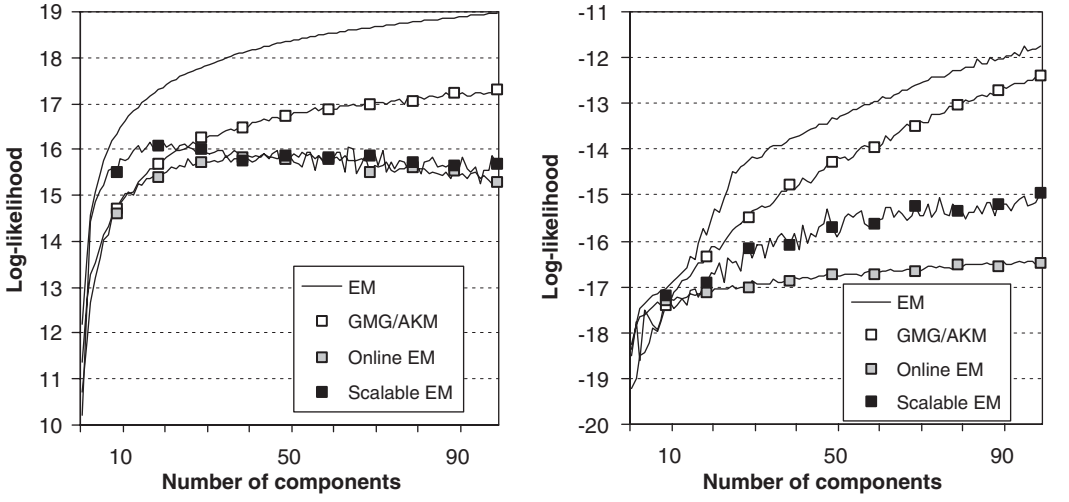


Fig. 10. Average log-likelihoods for *CT* (left) and *El Niño* (right) data sets, data in random order.

Table 8
Log-likelihoods for data set *El Niño* (model size 10)

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | −17.04 | −17.04 | −17.04 | −17.04 |
| Scalable EM | −20.55 | −18.36 | −17.46 | −17.26 |
| Online EM | N/A | N/A | −17.34 | −17.29 |
| GMG(4000,20)/AKM | −18.16 | −17.61 | −17.43 | −17.41 |

Table 9
Log-likelihoods for data set *El Niño* (model size 30)

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | −14.24 | −14.24 | −14.24 | −14.24 |
| Scalable EM | −18.64 | −17.94 | −16.23 | −16.21 |
| Online EM | N/A | N/A | −16.92 | −17.00 |
| GMG(4000,20)/AKM | −16.18 | −15.75 | −15.49 | −15.49 |

Table 10
Log-likelihoods for data set *El Niño* (model size 70)

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | −12.59 | −12.59 | −12.59 | −12.59 |
| Scalable EM | −17.59 | −17.99 | −15.42 | −15.23 |
| Online EM | N/A | N/A | −16.49 | −16.65 |
| GMG(4000,20)/AKM | −13.72 | −13.40 | −13.64 | −13.49 |

Table 11
Total processing times for data set *CM* in seconds

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | 40 203 | 40 203 | 40 203 | 40 203 |
| GMG(4000,20)/AKM | 279 | 421 | 205 | 251 |
| GMG(4000,30)/AKM | 89 | 84 | 70 | 84 |
| Scalable EM | 14 445 | 93 906 | 7483 | 7322 |
| Online EM | N/A | N/A | 1060 | 1136 |

Table 12
Total processing times for data set *CT* in seconds

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | 81 523 | 81 523 | 81 523 | 81 523 |
| GMG(4000,20)/AKM | 7323 | 19 651 | 5613 | 5200 |
| GMG(4000,30)/AKM | 96 | 135 | 96 | 81 |
| Scalable EM | 15 043 | 50 977 | 13 137 | 13 201 |
| Online EM | N/A | N/A | 2257 | 2125 |

Table 13
Total processing times for data set *El Niño* in seconds

| Algorithm | Data order | | | |
|---|---|---|---|---|
| | Original | Sorted | Random 1 | Random 2 |
| EM | 37 800 | 37 800 | 37 800 | 37 800 |
| GMG(4000,20)/AKM | 364 | 287 | 210 | 193 |
| GMG(4000,30)/AKM | 238 | 259 | 127 | 103 |
| Scalable EM | 90 828 | 82 154 | 28 476 | 23 476 |
| Online EM | N/A | N/A | 1146 | 1120 |

the data set. For the EM algorithm, the same time is reported for each order since the algorithm is independent of the order of the data. For the proposed method, the buffer size and points selected to generate new component are shown in parentheses. For Scalable EM, the buffer size is 4000 points. Times for the *CM* data set show that the proposed method used less time than Online EM, and significantly less time than Scalable EM. For data set *CT*, the processing time was still less than that of Scalable EM, but when only 20 points were selected from a large buffer the total time taken by the proposed method exceeds that of Online EM. The reason may have been that we selected 20 points from 16-dimensional data to create new components, which might have produced lots of small components, as with a buffer size of 2000 and 6 points selected as shown in Table 4. When 30 points were selected the processing time dropped noticeably. Hence the number of points selected apparently should be close to twice the dimensionality of the data, at minimum. For the *El Niño* data set the times are roughly similar to those of the *CM* data set except that picking 20 points did not result in an excessive running time. For Online EM, times did not vary much for single data set, as expected.

In practice it may be necessary to repeat the EM algorithm several times to ensure that it has not converged on a bad local optimum, which would multiply the running time by the number of repetitions. AKM takes 11% of the reported GMG/AKM totals, on average. Hence, given a model generated by GMG it is possible to generate several candidates of desired size in a short period using AKM without increasing the running time significantly.

## 4. Summary

Clustering large data sets presents challenges in terms of computational resources, namely the limitations of main memory and processing time. In this paper, we present an algorithm that converts the input data into a Gaussian mixture model in a two-stage process. The basic method in the first stage is to find small, compact subsets of the data and turn these into multivariate Gaussians. Our experiments showed that, on average, 3% of the data were used to generate new Gaussians and the rest of the data were used to update the parameters of the model. The user does not specify the number

of components in the resulting model. The model can then be post-processed as the second stage to generate a model with the desired number of components without using the original data.

We compared our algorithm with Online EM and Scalable EM algorithms using both synthetic and real-life data sets. Each data set was ordered in four different ways to find out if there are any implicit requirements regarding the order of the data. We found out that the order of the data has only a small effect on the performance of the proposed algorithm. Moreover, the proposed algorithm reads the data only once and stores only small amount of data, which allows us to handle very large data sets fast without excessive memory requirements.

## References


[1] T. Zhang, R. Ramakrishnan, M. Livny, BIRCH: a new data clustering algorithm and its applications, Data Mining and Knowledge Discovery 1 (2) (1997) 141–182.

[2] P. Bradley, U. Fayyad, C. Reina, Scaling clustering algorithms to large databases, in: Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining, 1998, pp. 9–15.

[3] S. Guha, R. Rastogi, K. Shim, CURE: an efficient clustering algorithm for large databases, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, 1998, pp. 73–84.

[4] R. Ng, J. Han, CLARANS: a method for clustering objects for spatial data mining, IEEE Trans. Knowledge Data Eng. 14 (5) (2002) 1003–1016.

[5] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, 1996, pp. 226–231.

[6] M. Ankerst, M. Breunig, H.-P. Kriegel, J. Sander, OPTICS: ordering points to identify the clustering structure, in: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data, 1999, pp. 49–60.

[7] R. Agrawal, J. Gehrke, D. Gunopulos, P. Raghavan, Automatic subspace clustering of high dimensional data for data mining applications, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, 1998, pp. 94–105.

[8] C. Procopiuc, M. Jones, P. Agarwal, T. Murali, A Monte Carlo algorithm for fast projective clustering, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, 2002, pp. 418–427.

[9] M. Sato, S. Ishii, On-line EM algorithm for the normalized Gaussian network, Neural Comput. 12 (2) (2000) 407–432.

[10] P. Bradley, U. Fayyad, C. Reina, Clustering very large databases using EM mixture models, in: Proceedings of the 15th International Conference on Pattern Recognition, vol. 2, 2000, pp. 76–80.

[11] A. Dempster, N. Laird, D. Rubin, Maximum likelihood from incomplete data via the EM algorithm, J. R. Statist. Soc. B 39 (1977) 1–38.

[12] G. McLachlan, T. Krishnan, The EM Algorithm and Extensions, Wiley, New York, 1997.

[13] C. Ordonez, E. Omiecinski, FREM: fast and robust EM clustering for large data sets, in: Proceedings of the 11th International Conference on Information and Knowledge Management, 2002, pp. 590–599.

[14] C.M. Bishop, Neural Networks for Pattern Recognition, Oxford University Press, Clarendon, 1996.

[15] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.

[16] A. Bhattacharyya, On a measure of divergence between two statistical populations defined by their distributions, Bull. Calcutta Math. Soc. 35 (1943) 99–110.

[17] K. Fukunaga, Introduction to Statistical Pattern Recognition, Academic Press, Boston, 1990.

[18] H. Jin, K.-S. Leung, M.-L. Wong, Z.-B. Xu, Scalable model-based cluster analysis using clustering features, Pattern Recognition 38 (5) (2005) 637–649.

[19] M. Ortega, Y. Rui, K. Chakrabati, K. Porkaew, S. Mehrotra, T.S. Huang, Supporting ranked Boolean similarity queries in MARS, IEEE Trans. Knowledge Data Eng. 10 (6) (1998) 905–925.

[20] S. Hettich, S. Bay, The UCI KDD Archive [http://kdd.ics.uci.edu], University of California, Department of Information and Computer Science, Irvine, CA, 1999.


**About the Author**—PASI FRÄNTI received his M.Sc. and Ph.D. degrees in Computer Science in 1991 and 1994, respectively, from the University of Turku, Finland. From 1996 to 1999 he was a postdoctoral researcher of the Academy of Finland. Since 2000, he has been a Professor in the Department of Computer Science, University of Joensuu, Finland. His primary research interests are in image compression, clustering and speech technology.

**About the Author**—ISMO KÄRKKÄINEN received his M.Sc. and Ph.D. degrees in Computer Science in 1999 and 2006, respectively, from the University of Joensuu, Finland. Currently he is a postdoctoral researcher in the same department. His primary research interests are in clustering and speech technology.