Fuzzy Cluster Analysis of Larger Data Sets

Roland Winkler, Frank Höppner, Frank Klawonn, Rudolf Kruse

March 2, 2009

Abstract

The application of fuzzy cluster analysis to larger data sets can cause runtime and memory overflow problems. While deterministic or hard clustering assigns a data object to a unique cluster, fuzzy clustering distributes the membership of a data object over different clusters. In standard fuzzy clustering, membership degrees will (almost) never become zero, so that all data objects are assigned to – even with very small membership degrees – all clusters. As a consequence, this does not only demand higher computational and memory power, it also leads to the undesired effect that all data objects will always influence all clusters, no matter how far away they are from a cluster. New approaches, modifying the idea of the fuzzifier, have been developed to avoid the problem of nonzero membership degrees for all data and clusters. In this paper, these ideas will be combined with concepts of speeding up fuzzy clustering by a suitable data organization, so that fuzzy clustering can applied more efficiently to larger data sets.

1 Introduction

Scalability refers to different facets of an algorithm. For data analyzing tools, there are two main interests: first, runtime or storage scalability and second, result quality scalability. Scalability in result quality is of interest i.e. for an algorithm that analyses continuous systems that need to be discretized. An other example would be an iterative algorithm to find an approximate solution because the addressed problem is too complex to solve it correctly. Runtime and storage scalability in contrast is an expression for the use of resources (like computation time and local memory) necessary to perform the algorithm. Often it is possible to increase the speed of an algorithm using more storage space and vice versa. If both is limited and crucial to an application, a trade-off between both is not sufficient.

As for prototype-based clustering, the above mentioned scalability in accuracy of the result depends on the definition of the term ‘cluster’. In general, a cluster is defined as a group of data objects in a continuous feature space. The data objects of one group are supposed to be as similar as possible and the data objects of different groups should be as different as possible. If this holds, a group of data objects is called a cluster. The standard Fuzzy c-Means algorithm (FcM) [2] fails to fit into this scheme, if there are groups of data objects in close proximity, that differ in the density of their data objects. This lack of quality is a result of the fact, that the fuzzy c-means algorithm tends to calculate a partition of data objects with equal number of data points. The reason for this is, that each data point influences all prototype. Dense groups of data points attract all prototypes, regardless whether there is a prototype in the centre of this dense cluster or not. Klawonn et al. showed in [15] that this effect occurs due to the calculation
of fuzzy membership values. An alternative fuzzifier function which explicitly allows membership values of 0 or 1 provides an update scheme which localizes the influence of prototypes and provides a better way of expressing the original idea of clusters. This idea is presented in section 4.

The two above mentioned processes to improve the quality of the clustering result and speeding up the calculation process lead to an interesting combination. Since an alternative fuzzifier function includes areas where the membership value of all data objects is 0 for a prototype, neighbourhood information can be used to save computation time. The combination of these two ideas will be presented in this chapter. The first section contains a short introduction to the well known Fuzzy c-Means algorithm, mainly to clarify the notation and to found a good basis for the following sections. Section 4 is dedicated to the changed fuzzifier function and in section 5, the hierarchical data structure that contains the neighbourhood information is presented. In Section 6, this information is used to calculate membership values for sets of data objects. We finish the chapter with some experimental results and close it with the conclusions in section 8.

2 Related Work

This work is related to two major fields of fuzzy clustering. In the first field, the concern is to increase the clustering quality or to adapt FcM to a specific problem because FcM does not generate the desired results. The first approach by Ruspini of Fuzzy c-Means only considered a fuzzifier value of 2 [17]. This approach was extended by Dunn to an adjustable value [8] which influences the softness of the fuzzy approach. Later, several approaches were made to change the behavior of FcM by changing the fuzzifier function i.e. [1, 15].

The second large field this work is related to, is the concern how to apply a FcM algorithm on very large data sets especially if only limited calculation resources are available. In the past, this was a much more important issue than it is today. For this work, we consider that the data set can be loaded fully into the local memory of the computer which provides random access to the data. Our main concern will be to adapt FcM in a way, that reduces the runtime of the algorithm.

Since Fuzzy c-Means is an iterative algorithm, there are in principle three ways to reduce the runtime complexity: a reduction of the data set size via sampling [7, 18, 11, 10] a reduction of the number of iteration steps [12], and a faster calculation for each single step [6, 14]. Höppner presented in [14] a way of reducing the complexity of one iteration step for large sets of data objects by calculating the membership value for sets of data objects that are close together. This is based on a data structure which contains neighbourhood information which we will present in Section 5. A similar approaches are presented by in [16] and [19] for Hard c-Means. A more geometric approach is used in [9] also for Hard c-Means. These three approaches benefit directly from the fact, that only the to a data object closest prototype needs to be considered for the clustering process. In a sense, we do something similar in our approach, the difference is, that we consider a set of closest prototypes.
3  Fuzzy c-Means Algorithm (FcM)

FcM is an algorithm to cluster data sets in a real feature space. The goal is, to find (fuzzy) clusters. In this section, a brief mathematical description of FcM is presented.

3.1 Definition (Fuzzy Set):  
Let $M$ be a set and $\mu : M \rightarrow [0, 1]$ be a continuous function, then $\mu$ is called fuzzy set on $M$ and for $x \in M$ is $\mu(x)$ referred as the membership degree of $x$ to $\mu$.

3.2 Definition (Data):  
A normed vector space $(V, \| \cdot \|)$ is called feature space and a finite, non-empty set $X = \{x_1, \ldots, x_n\} \subset V$ is called data set in $V$ with $n \geq 1$, $n \in \mathbb{N}$ data objects. A subset $W \subset X$ is called a cloud of data objects, if all objects in $W$ are considered to belong together.

It is not really necessary that $X$ is a set, i.e. that $X$ consists of $n$ different data objects. In our notation, we will always use $X$ is an indexed set and we allow objects with different indices to be equal.

3.3 Definition (Fuzzy Cluster):  
Let $V$ be a vector space and $X \subset V$ a data set in $V$ with $n$ data objects. A fuzzy set $\mu : X \rightarrow [0, 1]$ is called fuzzy cluster, if $\mu$ is defined by an algorithm. A finite, not empty set of fuzzy clusters $\Gamma = \{\mu_1, \ldots, \mu_c : X \rightarrow [0, 1]\}$ is called fuzzy partition of $X$, iff for $j = 1, \ldots, n$ holds:

$$1 = \sum_{i=1}^{c} \mu_i(x_j).$$

The difference between a cloud and a cluster is the entity which defines it. A cloud is specified by a human while a cluster is defined by an algorithm. While it is relatively easy for a human to find clouds in data sets with up to 3 dimensions, it is almost impossible for higher dimensions because it is harder to visualize. For a computer, the dimension is less important and it can in principle process data of arbitrary dimension. Clustering algorithms are designed to represent the clouds in a data set as well as possible with clusters. The point here is that the shape of clouds that ought to be found by the algorithm influences the type of algorithm that should be used. FcM belongs to the family of prototype based clustering algorithms. That means clouds have a convex form and can be represented by a vector that is called prototype.

3.4 Definition (Prototype):  
Let $V$ be a vector space and $X \subset V$ be a data set and $\mu : X \rightarrow [0, 1]$ be a fuzzy cluster. A vector $p \in V$ is called prototype of $\mu$, if $p$ represents $\mu$.

For FcM, clouds are considered to be shaped like hyperspheres with the highest density in the middle of a cloud and hence the clusters are defined accordingly: the prototype of a cluster is the center of the hypersphere. In the Fuzzy c-Means algorithm itself, the prototypes are used to calculate the fuzzy sets of data objects. The goal of Fuzzy c-Means is to find a fuzzy partition of a data set $X$ into $c$ fuzzy clusters so that the prototypes represent the data objects as well as possible.

3.5 Definition (Fuzzy c-Means Clustering):
4 ALTERNATIVE FUZZIFIER FUNCTION

Let \((V, \| \cdot \|)\) be a normed vector space, \(X = \{x_1, \ldots, x_n\} \subset V\) a data set, \(\Gamma = \{\mu_1, \ldots, \mu_c\}\) a fuzzy partitioning with the corresponding prototypes \(P = \{p_1, \ldots, p_c\} \subset V\) and the membership matrix \(U \in [0,1]^{c \times n}: u_{ij} = \mu_i(x_j)\) \(i = 1, \ldots, c, j = 1, \ldots, n\). Let \(\omega \in \mathbb{R}, \omega > 1\) be the fuzzifier, \(\Lambda = \{\lambda_1, \ldots, \lambda_n\} \subset \mathbb{R}\) some variables. Finally, let \(d_{ij} = \|p_i - x_j\|, i = 1, \ldots, c, j = 1, \ldots, n\) denote the distance of prototype \(i\) to data object \(j\).

The objective function \(J(X, U, P)\) is to be minimized under the constraint \(\sum_{i=1}^c u_{ij} = 1, \ j = 1, \ldots, n\) which is expressed with a Lagrange extension \(l(U, \Lambda)\) of \(J\):

\[
L(X, U, P, \Lambda) = \sum_{i=1}^c \sum_{j=1}^n u_{ij}^\omega d_{ij}^2 + \sum_{j=1}^n \lambda_j \left(1 - \sum_{i=1}^c u_{ij}\right) \quad (1)
\]

The result of FcM is the fuzzy partition \(\Gamma\) in a (local) minimum of \(U\) and \(P\).

A (local) minimum in \(U\) and \(P\) is reached iff the partial derivatives in all variables of \(U\) and \(P\) vanish. This leads to an iterative update scheme for the objective function \(J\) which is based on the idea of gradient descent. We assume in this paper, that the Euclidean distance is used for the function \(d\). All algorithms will be applicable as well for a different (global) distance function. Let \(t \in \mathbb{N}\) be a counter for the iteration step, then the variables are updated as follows:

\[
u_{ij}^{t+1} = \frac{(d_{ij}^t)^{\frac{\omega}{\omega-2}}}{\sum_{k=1}^c (d_{kj}^t)^{\frac{\omega}{\omega-2}}} \quad (2)
\]

\[
p_{i}^{t+1} = \frac{\sum_{j=1}^n (u_{ij}^t)^\omega \cdot x_j}{\sum_{j=1}^n (u_{ij}^t)^\omega} \quad (3)
\]

This update scheme is not applicable if one data object is identical to at least one prototype. In the unlikely event that such a situation occurs, the update for \(u_{ij}\) is changed to:

\[
u_{ij}^{t+1} = \begin{cases} \frac{1}{|I_j|}, & i \in I_j \\ 0, & \text{else} \end{cases}
\]

with \(I_j = \{k \in \mathbb{N} : x_j = p_k\}\). For the sake of simplicity, it is assumed that such a situation does not appear.

The iteration process is started by some sort of initialization for the prototypes, which can be done by using a random process or a more sophisticated method. The iteration is stopped when the sequence of membership values converges, i.e. \(\|U^{t+1} - U^t\|_M < \varepsilon\) for some \(\varepsilon > 0\) and \(\|\cdot\|_M\) is the maximum norm for matrices. For further information on the family of fuzzy clustering algorithms we refer the reader to [4, 13].

4 Alternative Fuzzifier Function

The idea of using fuzzy clustering instead of crisp clustering is that it might not be possible to clearly decide whether a data object belongs to just one cluster. For data objects that have no clear nearest prototype, a degree of membership is very useful to express that the clustering algorithm is unsure to which prototype the data object
belongs. Consider the classical example in Figure 1. In a crisp clustering i.e. Hard c-Means, the data object \( y \) would be assigned uniquely to one of the two clusters. Fuzzy c-Means solves that problem by assigning a data object only to a specific degree to a cluster, so that \( y \) would be assigned by around 0.5 to both clusters.

As plausible as fuzzy clustering is for data objects that can not be assigned uniquely, as implausible it is for data objects that are very close to one prototype. Assuming that no data object matches exactly a prototype, all data objects belong to some degree to every cluster. Consider for example data object \( x \) in Figure 1, no matter how close it is to the left prototype, it will always have a strictly larger than 0 membership value to the right prototype. In other words, all data objects influence all prototypes, so that all prototypes are drawn slightly to the centre of gravity of all data objects, which is not very plausible. The effect increases with the number of dimensions of the data set.

Hard c-Means clustering is plausible for data objects that are very near to exactly one prototype and FcM is plausible for data objects that are between at least two prototypes. The question is, why is FcM not plausible for data objects that are very near exactly one prototype?

**Understanding the Fuzzifier**

Klawonn et al. answered the above formulated question in [15] by examining a different objective function which depends on a continuous, strictly increasing function \( h : [0, 1] \rightarrow [0, 1] \) with \( h(0) = 0 \) and \( h(1) = 1 \) instead of simply a fuzzifier value:

\[
J_h(X, U, P) = \sum_{i=1}^{c} \sum_{j=1}^{n} h(u_{ij})d_{ij}^2
\]

Hard clustering can now be expressed by defining \( h \) as identity or FcM by using an exponential function \( h(u) = u^{\omega}, \omega > 1 \). Consider the special case of two clusters like in Figure 1. \( J_h \) must be minimized for every data object and for \( x_j \) in particular, so that the term

\[
J_{\min}(x_j, u_{ij}) = h(u_{1j})d_{1j}^2 + h(u_{2j})d_{2j}^2 = h(u_{1j})d_{1j}^2 + h(1 - u_{1j})d_{2j}^2
\]
must be a minimum. A necessary condition for a minimum is that the derivative of \( \frac{\partial}{\partial u_{ij}} J_{min} \) is zero:

\[
0 = h'(u_{1j})d_{1j}^2 - h'(1 - u_{1j})d_{2j}^2 \\
= h'(u_{1j})d_{1j}^2 - h'(u_{2j})d_{2j}^2 ,
\]

with \( h' \) is the derivation of \( h \). That leads to

\[
\frac{h'(u_{1j})}{h'(u_{2j})} = \frac{d_{2j}^2}{d_{1j}^2} . \tag{4}
\]

This means, the ratio of the (transformed) membership value gradients must correspond to the ratio of the squared distances. In the case of \( h(u) = u^\alpha \), the derivative of \( h \) vanishes if the membership value is zero \( u_{min} = 0 \): \( h'(0) = 0 \). On the other hand, the derivative if \( u_{max} = 1 \) is always larger than zero: \( h'(1) = \omega \cdot 1^{\alpha-1} = \omega > 0 \). Because the ratio of equation (4) still holds, there are no corresponding distances that could cause a crisp membership assignment for the \( FeC \) algorithm. For this reason, every data object influences all prototypes as long as it is not identical to one of the prototypes.

Crisp c-Means has a different problem, in this case \( h \) is the identity: \( h(u) = u \) which leads to a derivative of \( h'(u) = 1, \forall u \). This is not applicable on the update scheme of \( FeC \), since equation (4) only holds, if the distances are equal. So the update function can not be applied in the above specified way. Still the goal is to minimize \( J \) and since only membership values of \( 0 \) and \( 1 \) are allowed, the membership corresponding to the closest prototype is set to \( 1 \) and \( 0 \) otherwise. Obviously, there are no fuzzy membership values, even if the data objects have almost identical distance to all prototypes.

To solve both problems, the fuzzifier function \( h \) must be chosen in a way that \( h'(0) > 0 \) and \( h'(u_1) < h'(u_2) \) for all \( 0 \leq u_1 < u_2 \leq 1 \). The function family of \( h(u) = \alpha u^2 + (1 - \alpha)u, \alpha \in [0, 1] \) satisfies this property. The lower bound of the ratio of the membership value gradients is

\[
\frac{h'(0)}{h'(1)} = \frac{1 - \alpha}{2\alpha + (1 - \alpha)} = \frac{1 - \alpha}{1 + \alpha} .
\]

This means, that the ratio of squared distances must exceed this value or the membership values are set to their limits of \( 0 \) or \( 1 \) respectively. So this fuzzifier function behaves like Hard c-Means if the ratio of squared distances is below \( \frac{1 - \alpha}{1 + \alpha} \) and like \( FeC \) with a changed fuzzifier function otherwise. Because this is a very intuitive property, \( h \) is parametrized by using \( \beta = \frac{1 - \alpha}{1 + \alpha} \), from which follows: \( \alpha = \frac{1 - \beta}{1 + \beta} \):

\[
h(u) = \frac{1 - \beta}{1 + \beta} u^2 + \frac{2\beta}{1 + \beta} u .
\]

**Fuzzy Clustering with Polynomial Fuzzifier Function**

Based on the idea above, the update process for fuzzy clustering must be adapted.

**4.1 Definition** (Fuzzy c-Means Clustering with Polynomial Fuzzifier Function):

Let \( (V, \| \cdot \|) \) be a normed vector space, \( X = \{ x_1, \ldots, x_n \} \subset V \) a data set, \( \Gamma = \{ \mu_1, \ldots, \mu_c \} \) a fuzzy partitioning with the corresponding prototypes \( P = \{ p_1, \ldots, p_c \} \subset V \) and the membership matrix \( U \in [0, 1]^{c \times n} \): \( u_{ij} = \mu_i(x_j) \ i = 1..c, j = 1..n. \ Let
The objective function denotes the distance of prototype \( i \) to data object \( j \) as follows:

\[
L(X,U,P,\Lambda) = \sum_{i=1}^{c} \sum_{j=1}^{n} \lambda_j \left( 1 - \sum_{i=1}^{c} u_{ij} \right) = \sum_{i=1}^{c} \sum_{j=1}^{n} \left( \frac{1 - \beta}{1 + \beta} \sum_{k \in X} u_{kj}^2 + \frac{2\beta}{1 + \beta} u_{ij} \right) d_{ij}^2 + \sum_{j=1}^{n} \lambda_j \left( 1 - \sum_{i=1}^{c} u_{ij} \right)
\]

As before, for all valid solutions, the Lagrange extension \( l(U,\Lambda) \) is equal to zero. As for FcM, \( L \) is w.r.t. \( u_{ij} \) and \( p_i \) are computed to obtain the update formula:

\[
\frac{\delta L}{\delta u_{ij}} = \left( \frac{2(1-\beta)}{1+\beta} u_{ij} + \frac{2\beta}{1+\beta} \right) d_{ij}^2 - \lambda_j = 0
\]

\[
\Rightarrow \quad u_{ij} = \frac{1}{1-\beta} \left( \frac{(1+\beta)\lambda_j}{2d_{ij}} - \beta \right)
\]

The parameter \( \lambda_j \) can be calculated, by using the constraint, \( 1 = \sum_{i=1}^{c} u_{ij} \). Mathematically, the Lagrange extension transfers the optimization problem into a higher dimensional space and restricts it there to a hyperplane of valid solutions. Additionally, if equation (6) is used without taking into account that \( u_{ij} \in [0,1] \), it will produce membership values that are not restricted to \([0,1]\). Because the sum of all membership values is fixed at 1, it is enough to ensure that no membership value is strictly less than zero. In other words, the membership values that are strictly less than zero are set to zero and the remaining membership values are reweighted to gain a sum of membership values of 1. Suppose it is known for which membership values equation (6) gives values greater than zero and that the number of this membership values is \( \hat{c} \):

\[
1 = \sum_{u_{kj} \geq 0} u_{kj} = \sum_{k=1}^{\hat{c}} \frac{1}{1-\beta} \left( \frac{(1+\beta)\lambda_k}{2d_{kj}} - \beta \right)
\]

\[
\Rightarrow \quad \lambda_j = \frac{2(1+\hat{c} - 1)\beta}{(1+\beta) \sum_{u_{kj} \geq 0} d_{kj}^2}
\]

which leads to the final equation for \( u_{ij} \):

\[
u_{ij} = \frac{1}{1-\beta} \left( \frac{1 + (\hat{c} - 1)\beta}{\sum_{k=1}^{\hat{c}} d_{kj}^2} - \beta \right)
\]

At the first glance, this seems to be circular reasoning. And indeed, it is necessary to know for which prototypes the membership value is larger than zero to calculate the membership values. But there is a possibility to solve this problem using the result in Lemma 4.2 and a sequence of tests.
4.2 Lemma (Monotonicity of Membership Values):

Let the $V$, $X$, $P$, $U$, $Λ$, $h$, $β$ and $d_{ij}$ like in definition 4.1, then for each data object $x_j$, $j = 1..n$ and pair of prototypes $p_i$ and $p_k$, $1 \leq i, k \leq c$, it holds:

$$0 \neq u_{ij} \geq u_{kj} \Rightarrow d_{ij} \leq d_{kj} \quad (8)$$
$$d_{ij} \leq d_{kj} \Rightarrow u_{ij} \geq u_{kj} \quad (9)$$

Proof. Because the membership values of the data objects do not depend on each other, it is enough to consider just one data object $x_j$. Also consider for a moment that the membership values are not restricted to the interval $[0, 1]$. Then equation (6) holds for all prototypes. Since $λ_j$ and $β$ are constants, it is easy to see that the membership values are monotonous in the distance values. Because of the restriction to the $[0, 1]$-interval, all prototypes with potential membership values less than 0 are excluded from the calculation process and their membership value is set to 0. That means, the clustering process is done with less prototypes than it would be possible. But because their membership value is set to 0, they have no influence on the value of the objective function $J$. Due to the monotonicity of equation (6), that means that all prototypes with a membership value larger than 0 are closer to $x_j$ than all others. For the first statement, there are two cases:

Case 1: If $u_{kj} = 0$:

Than $p_k$ is excluded from the calculation and since $p_i$ is not excluded, it follows $d_{ij} < d_{kj}$.

Case 2: If $u_{kj} > 0$:

In this case, (6) holds for both values and due to its monotony, it follows $d_{ij} \leq d_{kj}$.

The second statement holds, independently of the excluding process. If $p_i$ is excluded, so is $p_k$ and both membership values are set to 0. If only $p_k$ is excluded, $u_{ij} > u_{kj}$ by construction. And if none of them is excluded, equation (6) holds for both prototypes.

From this lemma, it can be concluded that if equation (6) for a prototype holds, then it holds for all prototypes that are closer to the data object. Even more importantly, it means: if there is one prototype for which (6) does not hold, then for all prototypes further away, it does not hold either.

To break the circular reasoning in equation (7), it is necessary to know the set of prototypes that are involved in the calculation of the membership value. With lemma 4.2, it is already known that the set of prototypes can be split into two subsets, according to their distance to $x_j$, hence the first step is, to sort the prototypes w.r.t. the distance to the considered data object. The second step is to find that prototype for which equation (7) still holds, but no prototype further away can be added to the selection. Let $ϕ$ be a permutation of $(1 \ldots c)$ so that $d_{ϕ(i)j} \leq \ldots \leq d_{ϕ(c)j}$ holds. Due to lemma 4.2 all membership values to the prototypes $p_{ϕ(1)}$, $\ldots$, $p_{ϕ(c-1)}$ with $d_{ϕ(i)j} < d_{ϕ(c)j}$ are
greater than 0. Hence, it is sufficient to test whether \( u_{\phi(i),j} \) is greater than 0:

\[
0 \leq \frac{1}{1 - \beta} \left( \frac{1 + (\hat{c} - 1)\beta}{\sum_{k=1}^{\hat{c}} \phi(k)_j^2} - \beta \right)
\]

\[\iff \quad \sum_{k=1}^{\hat{c} - 1} \frac{d^2_{\phi(j)}}{\phi(k)_j^2} \leq \frac{1}{\beta} + \hat{c} - 1 \]

\[\iff \quad d^2_{\phi(j)} \left( \sum_{k=1}^{\hat{c} - 1} \frac{1}{\phi(k)_j^2} \right) - \hat{c} \leq \frac{1}{\beta} - 2 \]

The test can be done by successively increasing \( \hat{c} \) and performing the test in each iteration. Let \( \hat{c} \) be the highest index for which the test was successful. Note that \( \hat{c} \) has to be calculated for each data object individually, hence it might be helpful to consider \( \hat{c} \) as an indexed variable with the iteration variable \( t \in \mathbb{N} \):

\[
\hat{c}^t_j = \max \left\{ \hat{c} \in \mathbb{N} \left| \hat{c} \leq c, \sum_{k=1}^{\hat{c}} \frac{(d^t_{\phi(j)})^2}{\phi(k)_j^2} \leq \frac{1}{\beta} + \hat{c} - 1 \right. \right\}
\]

for \( j = 1..n \).

Finally, equation (7) is slightly modified and extended by an iteration variable \( t \in \mathbb{N} \):

\[
u^{t+1}_{ij} = \frac{1}{1 - \beta} \left( \frac{1 + (\hat{c}^t_j - 1)\beta}{\sum_{k=1}^{\hat{c}^t_j} (d^t_{\phi(j)})^2} - \beta \right) \quad \text{iff} \quad \phi(i) \leq \hat{c}^t_j
\]

\[= 0 \quad \text{otherwise}
\]

The update function for the prototypes does not change much to the one in FcM:

\[
p^{t+1}_i = \frac{\sum_{j=1}^{n} h(u^{t+1}_{ij}) \cdot x_j}{\sum_{j=1}^{n} h(u^{t+1}_{ij})}
\]

Figure 2 illustrates the effect of the polynomial fuzzifier function. As it can be seen, the clusters are detected very well and data objects very close to a prototype are assigned with membership degree of 1. Data objects with no clear nearest prototype are clustered softly. It is worth to point out, that the prototype positions are almost identical to the ones from FcM on the left hand side picture. The polynomial fuzzifier function can be of great use if there are different dense clusters or clusters of different number of data objects. For a more detailed analyses of clustering with polynomial fuzzifier function, see for example [15] and [5].

5 Neighbourhood Representation of Data

The approach of Höppner in [14] reveals that it is not necessary to calculate the membership value for each pair of data object and prototype separately, because the membership values of close data objects might not differ significantly.
Figure 2: The same data set, clustered with FcM (left) and with polynomial fuzzifier function (right). The 'tails' of the prototypes represent the path they took during the clustering process and the large enclosed areas on the right hand side are the convex hulls of all data objects with a membership value of 1. The fuzzifier on the left hand side is $\omega = 2$ and the $\beta$-parameter on the right hand side is $\beta = 0.5$.

Consider the example from the beginning, Figure 1 with normal FcM again. The data objects near the left prototype do not have significant influence on the position of the right prototype. In fact, their exact position is not of much interest for calculating their membership value to the right prototype. FcM is an iterative algorithm, which means, it has a termination test. This test might be $\|U^{t+1} - U^t\|_M < \varepsilon$ for some $\varepsilon > 0$ and $\|\cdot\|_M$ the maximum norm for matrices as described in section 3. The value of the maximum norm $\|U^{t+1} - U^t\|_M = \max_{i=1,\ldots,n,j=1,\ldots,c}(|u_{ij}^{t+1} - u_{ij}^t|)$ is the largest difference of all membership values between two iteration steps. Since the algorithm terminates if the largest difference in membership values is below $\varepsilon$, no difference in membership values smaller than $\varepsilon$ is of interest to the calculation process of FcM. Using this tolerance, it is possible to consider groups of close data objects as if they all are located at the same position.

Again, consider example 1. The circle around $x$ denotes the space in which $x$ can be moved so that the difference in membership values to the right prototype is below $\varepsilon$. This means, all data objects within the circle can be considered to be located at position of $x$ and for none would be the difference in membership values towards the right prototype above $\varepsilon$. This gives a basic idea, how to save computation time here. But to use this property effectively, the data objects have to be stored in a way that neighbourhood information is available. The data structure is organized as a tree and constructed before starting the clustering process.

5.1 Definition (Neighbourhood Tree):

Let $V$ be a normed vector space, $X \subset V$ be a data set and $\delta : \mathbb{R}^+ \to \mathbb{R}^+$ a function with $\delta(r) < r$, $r > 0$. Let $T$ be a tree and $N = (x \in X, C, r \in \mathbb{R}^+)$ a node with $C$ is the set of child nodes. $T$ is called neighbourhood tree, if it holds:

- $\text{succ}(N) \subset B_r(x)$
- for all child nodes $N_1 = (x_1, C_1, r_1) \in C$ holds: $r_1 \leq \delta(r)$
for all pairs \( N_1 = (x_1, C_1, r_1), N_2 = (x_2, C_2, r_2) \in C, N_1 \neq N_2 \) holds:
\[
\|x_1 - x_2\| > \delta(r)
\]
with \( \text{succ}(N) = \{x\} \cup \bigcup_{N' \in C} \text{succ}(N') \) is the set of all successor elements and \( B_r(x) \subset V \) is the hypersphere around \( x \) with radius \( r \). The neighbourhood tree \( T \) of a data set \( X \) is the tree that is associated with the root node \( N_{\text{root}} = (x \in X, C_{\text{root}}, r_{\text{max}}) \) so that \( \text{succ}(N_{\text{root}}) = X, x \) a random element in \( X \) and \( r_{\text{max}} = \max \{\|y - x\| : y \in X \setminus \{x\}\} \).

A node \( N = (x, C, r) \) can be seen as a representative of its successor elements \( \text{succ}(N) \). In this matter, a child node \( N_1 = (x_1, C_1, d_1) \) of \( N \) represents a subset of the data objects \( \text{succ}(N_1) \subset \text{succ}(N) \) which are located in the corresponding hypersphere \( \text{succ}(N_1) \subset B_{r_1}(x_1) \). Each data object \( y \in \text{succ}(N) \setminus \{x\} \) is associated to exactly one of its child nodes. If \( y \) is located in the overlap of the hyperspheres of at least two child nodes \( N_1, N_2 \in C: y \in B_{r_1}(x_1) \cap B_{r_2}(x_2) \), there is no clear regulation to which child node \( y \) belongs. It is possible to apply an ordering over the child nodes to associate \( y \) to the first child node in this ordering, but this is a rather arbitrary regulation. Instead, the neighbourhood tree is extended so that \( y \) is associated with the closest child node which leads to the definition of a strict neighbourhood tree:

5.2 Definition (Strict Neighbourhood Tree):
Let \( V, X, \delta \) and \( T \) be like in definition 5.1 above. \( T \) is called strict, if one more restriction is applied on the child nodes of \( N = (x, C, r) \):

- for all pairs \( N_1 = (x_1, C_1, r_1), N_2 = (x_2, C_2, r_2) \in C \) holds: \( \forall y_1 \in \text{succ}(N_1): \|y_1 - x_1\| \leq \|y_1 - x_2\| \)

A strict neighbourhood tree guarantees that a data object belongs to that child, which is closest to it. In the unlikely event that a data object has exactly the same distance to several child nodes, the data object is associated due to an arbitrary ordering on the child nodes. The following algorithm constructs a strict neighbourhood tree from a set of data objects \( X \) and the contraction function \( \delta \).

5.3 Algorithmus (Construct a Strict Neighbourhood Tree):
INPUT: Data set \( X \), function \( \delta : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \)
OUTPUT: A strict neighbourhood tree \( T = (x \in X, C, r_{\text{max}}) \)

1 \( x \leftarrow \text{randomElement}(X), Y \leftarrow X \setminus \{x\} \)
2 \( N_{\text{root}} \leftarrow (x, \emptyset, \max\{\|y - x\| : y \in Y\}) \)
3 \( \text{INSERT}(N_{\text{root}}, Y) \)
4 \textbf{end.}
5 \( \textbf{function INSERT}(N = (x, C = \emptyset, r), Y) \)
6 \( Z \leftarrow \emptyset, \hat{Y} \leftarrow \emptyset, i \leftarrow 1 \)
7 \( \textbf{for } y \in Y \textbf{ do} \)
8 \( \text{if } \exists x', \emptyset, r' \in C : y \in B_{r'}(x') \textbf{ do} \)
9 \( \hat{Y} \leftarrow \hat{Y} \cup \{y\} \)
10 \textbf{else} \)
11 \( C_i \leftarrow (y, \emptyset, \delta(r)), C \leftarrow C \cup \{C_i\} \)
12 \( Z_i \leftarrow \emptyset, Z \leftarrow Z \cup \{Z_i\} \)
13 \( i \leftarrow i + 1 \)
14 \textbf{end if} \)

Roland Winkler, Frank Höppner, Frank Klawonn, Rudolf Kruse
Accordingly, for example δ.

It is desired to not have only few child nodes per node, the function 22).

done for each child node if at least one additional data object is associated to it (line Z

the number of successors |Z|.

complexity is in $O(1)$.

data objects in 1 and many identical data objects is weighted according to the number of identical data objects.

which would lead to a ridiculous annoying mathematical description. In practice, a subtree that contains

many identical data objects is weighted according to the number of identical data objects.ROLE set would not be considered for clustering. So in future, it is assumed that the data set

produce a list-like tree. But such situation is more of academic nature because the data

balanced. It is possible, that the data set is corrupted in a way, that the algorithm would

that a node’s centre object is located in the middle of its represented data objects. Furthermore, because of the shrinking of the distances in line 21, it is likely that a node has more than one child node, if it represents at least three data objects (The data object contained in the node it self and two more). However, the tree is not constructed to be balanced. It is possible, that the data set is corrupted in a way, that the algorithm would produce a list-like tree. But such situation is more of academic nature because the data set would not be considered for clustering. So in future, it is assumed that the data set is not corrupted in such matter, hence the depth of the tree is logarithmic in the number of data objects. In Figure 3, 4 levels of a strict neighbourhood tree are shown.

If the data set is not corrupted, the runtime complexity of the algorithm is in $O(n \log(n))$. All data objects, associated to a node $N = (x, C, r)$ are contained in the hypersphere $B_r(x)$. In this hypersphere, fit at most $K \in \mathbb{N}$ hyperspheres of radius $\delta(r)$, so that no centre of one hypersphere is contained in any other. Therefore, the computation time of the loops starting in lines 7 and 16 have both a complexity of $O(K \cdot \text{succ}(N))$.

The set of all represented data objects of the nodes in one level of the tree, at most the full set of data objects, hence one level of the tree is constructed in $O(2 \cdot K \cdot n)$. The height of the tree is logarithmic in the number of data objects, so that the construction complexity is in $O(2 \cdot K \cdot n)$. During the construction process, it is also possible to store the number of successors $|\text{succ}(N)|$ of a node $N$. This is important for applying the

\footnote{Otherwise it would not be possible to consider the collectivity of all data objects as an algebraic set which would lead to a ridiculous annoying mathematical description. In practice, a subtree that contains many identical data objects is weighted according to the number of identical data objects.}

The algorithm constructs a strict neighbourhood tree. A random element is selected to generate the root node while the tree is built recursively using the function INSERT. The first step generates a covering for all the represented data objects, lines 7 until 15. The set $\hat{Y} \subset X$ contains the data objects that are not used as centers for the child nodes. The covering is generated by testing for each data object $x \in X$, if it can be associated to an already existing child node and if not, a new child node is generated. Due to simplicity, it is assumed that no data object occurs more than once1. The remaining data objects in $\hat{Y}$ that are not used for child construction are subdivided into the sets $Z_1, \ldots, Z_{|C|}$ according to their closest child node, line 16 until 19. The recursion is done for each child node if at least one additional data object is associated to it (line 22).

The set of all represented data objects of the nodes in one level of the tree, at most the full set of data objects, hence one level of the tree is constructed in $O(2 \cdot K \cdot n)$. The height of the tree is logarithmic in the number of data objects, so that the construction complexity is in $O(2 \cdot K \cdot n)$.
6 Fuzzy Clustering using Neighbourhood Information

The motivation of constructing the neighbourhood tree is that it is not necessary to calculate the exact membership value for each data object, if the difference between an estimation and the real membership value is below the termination threshold $\forall 1 \leq i \leq c, 1 \leq j \leq n : |u_{ij}^t - u_{ij}^{t+1}| < \varepsilon$ of the clustering algorithm. Furthermore, due to the polynomial fuzzifier function, introduced in section 4 there might be vast areas in which the membership values of data objects to one prototype might be 1 or 0. Obviously, the membership values for these data objects need not be calculated for each data object individually.

Figure 3: These four figures represent 4 succeeding levels in a strict neighbourhood tree. The cross symbols represent the centre of gravity of the corresponding data objects.

clustering algorithm later. The storage complexity for the neighbourhood tree and its construction is in $O(n)$ because each node stores one data object.

6 Fuzzy Clustering using Neighbourhood Information
6.1 Membership Value Interval

Suppose it is desired to calculate the membership value of data object \( x_j \) to the prototype \( p_i \). Given a node \( N = (x_j, C, r) \) in the neighbourhood tree guarantees that all data objects \( \text{succ}(N) \) are in a hypersphere \( B_r(x_j) \). So the location of the data objects do not vary more than a distance \( r \) to the data object \( x_j \). Therefore, the distances of the data objects \( y \in \text{succ}(N) \) to a prototype \( p_i \) must be in the interval

\[
[d^-_{ij}, d^+_{ij}] = [\max\{0, \|p_i - x_j\| - r\}, \|p_i - x_j\| + r]
\]

Consider the hypothetical case that the distance of the data object \( x_k \in \text{succ}(N) \) (\( k \neq j \)) to the prototype \( p_i \) is reduced, but the distance to all other prototypes is constant, than due to lemma 4.2 the membership value \( u_{ik} \) is higher than \( u_{ij} \). Again from lemma 4.2 follows that if \( x_k \) is closer to prototype \( p_l \) \((l \neq i)\), but keeps all other distances fixed, the membership value \( u_{kj} \) is lower than \( u_{ij} \). So the minimal hypothetical membership value of a data object \( x_k \in \text{succ}(N) \) towards prototype \( p_i \) would be, if \( x_k \) reduces its distance to all prototypes but \( p_i \) by \( r \) and increases its distance to \( p_i \) by \( r \). Since the norm \( \| \cdot \| \) is symmetric, it does not matter if a data object changes its location or the prototype. According to equation (10), not the actual position of the prototypes is relevant for calculating the membership value, only its distances to the data object is important. So instead of considering the data object move inside the hypersphere \( B_r(x_j) \), consider the prototypes change their distance to \( x_j \). Based on this idea, the following theorem is formulated:

6.1 Theorem (Membership Interval):

Let \( V \) be a normed vector space, \( X = \{x_1, \ldots, x_n\} \subset V \) a data set, \( P = \{p_1, \ldots, p_c\} \subset V \) a set of prototypes, \( x_j \in X \) and \( p_i \in P \) and \( r \in \mathbb{R}, r > 0 \).

\[
\forall x_k \in B_r(x_j) : u_{ik} \in I_i = [I^-_i, I^+_i]
\]

with

\[
I^-_i = \left\{ \begin{array}{ll}
\frac{1}{1-\beta} \left( 1 + \frac{(\hat{c}_i^- - 1) \beta}{1 + \sum_{k=1}^n \frac{(\hat{c}_i^-) \beta}{w_i^- (i) \|x_i\|^2}} \right), & \varphi_i^- (i) \leq \hat{c}_i^- \\
0, & \text{otherwise}
\end{array} \right.
\]

\[
I^+_i = \left\{ \begin{array}{ll}
\frac{1}{1-\beta} \left( 1 + \frac{(\hat{c}_i^+ - 1) \beta}{1 + \sum_{k=1}^n \frac{(\hat{c}_i^+) \beta}{w_i^+ (i) \|x_i\|^2}} \right), & \varphi_i^+ (i) \leq \hat{c}_i^+ \\
0, & \text{otherwise}
\end{array} \right.
\]

with \( d^-_{ij} = \max\{0, \|p_i - x_j\| - r\} \) and \( d^+_{ij} = \|p_i - x_j\| + r \) are the minimal (maximal) hypothetical distances of the data object \( x_j \) in the hypersphere \( B_r(x_j) \) to the prototype \( p_i \). \( \hat{c}_i^- \) and \( \hat{c}_i^+ \) denotes the number of relevant prototypes and \( \varphi_i^- (\varphi_i^+) \) denote a new ordering of prototypes w.r.t. the changed distances.
Before proving the statement, please note that the formula for the interval borders is slightly changed to equation (10). This derives from the fact that the sum in the denominator of the fraction in (10) contains the term $\frac{d_{ij}}{d_i}$ with $i = \varphi(l)$. With changed distances, this would result in $\frac{d_{ij}}{d_{ij}}$ (or $\frac{d_{ij}}{d_{ij}}$) which is not plausible because the distance of the data object $x_j$ to the prototype $p_i$ can not increase and decrease at the same time. Therefore, this term is excluded from the sum. This change makes it possible to separate the distance change of $p_i$ to the other prototypes $p_{\varphi(l)}$, $\varphi(l) \neq i$, $l = 1 \ldots \hat{c}$.

The proof shows that the membership value of $x_j$ towards $p_i$ does not change, if another prototype $p_l$ is added to the set of relevant prototypes exactly at a distance, where equation (10) produces a membership value of zero $u_{lj} = 0$. So in other words, the membership values change continuously in the distances towards the prototypes even if the number of relevant prototypes changes. With this result and the monotony from lemma 4.2, the above theorem is proven.

Proof. From lemma 4.2, we know, that the membership values change monotonously. In case the number of relevant prototypes $\hat{c}^- = \hat{c}$ does not change, the result follows directly from lemma 4.2. Not clear is, what happens if $\hat{c}^- \neq \hat{c}$. So the proof shows, that the membership value of $x_j$ towards $p_i$ does not change, if another prototype $p_l$ is added to the set of relevant prototypes exactly at a distance, where equation (10) produces a membership value of zero $u_{lj} = 0$. So in other words, the membership values change continuously in the distances towards the prototypes even if the number of relevant prototypes changes. With this result and the monotony from lemma 4.2, the above theorem is proven.

Without loss of generality, suppose the prototypes are sorted according to their distance, so we do not need to consider the permutation. To introduce an additional prototype $p_{\hat{c}+1}$ to equation (7) such that its membership value is zero $u_{(\hat{c}+1)j} = 0$, its distance $d_{(\hat{c}+1)j}$ must follow the following condition:

$$0 = u_{(\hat{c}+1)j} = \frac{1}{1-\beta} \left( \frac{1+((\hat{c}+1)j-1)\beta}{\sum_{k=1}^{\hat{c}+1} d_{k,j}^2} - \beta \right)$$

$$\Leftrightarrow d_{(\hat{c}+1)j}^2 = \frac{1+(\hat{c}-1)\beta}{\beta \sum_{k=1}^{\hat{c}+1} \frac{1}{d_{k,j}^2}}$$

If equation (7) is now evaluated for $u_{ij}$ adding the additional prototype from the previ-
ous calculation, we obtain:

\[
u_{ij} = \frac{1}{1 - \beta} \left( \frac{1 + (\hat{c} + 1 - 1)\beta}{\sum_{k=1}^{\hat{c}} d_{ik}^{2} + \frac{d_{ij}^{2}}{d_{(c+1)j}^{2}}} - \beta \right)
\]

\[
= \frac{1}{1 - \beta} \left( \frac{1 + \hat{c}\beta}{\sum_{k=1}^{\hat{c}} d_{ik}^{2} + \beta \sum_{k=1}^{\hat{c}} d_{kj}^{2} + \frac{d_{ij}^{2}}{d_{(c+1)j}^{2}}} - \beta \right)
\]

\[
= \frac{1}{1 - \beta} \left( \frac{1 + (\hat{c} - 1)\beta}{\sum_{k=1}^{\hat{c}} d_{ik}^{2}} - \beta \right)
\]

This means, the membership value \( u_{ij} \) does not change by introducing the additional prototype. Now it is possible to consider (7) as a continuous function in the distance variables, even if the number of involved prototypes \( \hat{c} \) changes. The only condition up to this point is, that the ordering of the prototypes stays the same.

Therefore it follows with lemma 4.2, equation (7) is a continuous, decreasing function in distances \( d_{lj} \) with \( l \neq i \) and increasing in \( d_{ij} \). If there is a \( x_k \in X \) with \( u_{ik} \notin I_i = [u_{ij}^-, u_{ij}^+] \), there are two cases left to consider:

Case 1: \( u_{ik} < u_{ij}^- \). Since (7) is continuous and monotonous, it means that \( d_{ik} > d_{ij}^+ \) and/or \( \exists l \neq i : d_{lk} < d_{ij}^- \).

Case 2: \( u_{ik} > u_{ij}^+ \). With the same argument, it follows that \( d_{ik} < d_{ij}^- \) and/or \( \exists l \neq i : d_{lk} > d_{ij}^+ \).

In both cases, \( x_k \notin B_d(x_i) \). That means, when ever \( x_k \in B_d(x_j) \), \( u_{ik} \in I_i \).

Consider a node of the neighbourhood tree: \( N = (x_i, C, r) \) let its membership interval with distance \( r \) be \( I_i \subset [u_{ij} - \varepsilon, u_{ij} + \varepsilon] \), then all successors \( \text{succ}(N) \) can be treated exactly like \( x_j \) hence, their membership value towards prototype \( p_i \) does not need to be calculated individually.

Note that the membership value interval is a very pessimistic estimation, since it is assumed that the distances to all prototypes become worst case. In practise, such situation nearly never occurs and in most cases, the real range of membership values is far smaller than the calculated interval. A much better estimation would arise from the following formula:

\[
u_{ij}^- = \inf_{y \in B_d(x_j)} \left( \frac{1}{1 - \beta} \left( \frac{1 + (\hat{c}_y - 1)\beta}{\sum_{k=1}^{\hat{c}_y} \|y - p_k\|^2} - \beta \right) \right)
\]

To give a general case solution for this problem is not trivial and might be even impossible. This question is subject to further research.
6.2 Alternative Fuzzy c-Means

In this subsection, we introduce the Fuzzy-c Means algorithm using a polynomial fuzzifier and the strict neighbourhood tree. There are two versions for this algorithm, one with more memory consumption and the other which uses a different termination rule. At first, the version which uses more memory is presented, because it needs one less user defined parameter and is more consistent with the motivation of using neighbourhood information. The other version of the algorithm is presented in the next subsection.

6.2 Algorithm (Modified FcM):

INPUT: Data set \( X \), function \( \delta : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \), number of prototypes \( c \), termination threshold \( \varepsilon > 0 \), a parameter \( \beta \in [0,1] \)

OUTPUT: A fuzzy partition \( U \)

1. \( N_{\text{root}} \leftarrow \text{neighbourhoodTree}(X, \delta) \)
2. \( P \leftarrow \text{initialize}\{p_1, \ldots, p_c\} \)
3. \( U \leftarrow [0]_{(c,n)}, \ U' \leftarrow [1]_{(c,n)} \)
4. \( \text{while } \exists 1 \leq i \leq c, 1 \leq j \leq n : |u_{ij} - u'_{ij}| > \varepsilon \) do
5. \( U \leftarrow U' \)
6. \( P' \leftarrow \{p'_1 \leftarrow 0, \ldots, p'_c \leftarrow 0\} \)
7. \( T = \{t_1 \leftarrow 0, \ldots, t_c \leftarrow 0\}, Z = \{z_1 \leftarrow -1, \ldots, z_c \leftarrow -1\} \)
8. \( \text{UPDATE}(N_{\text{root}}, R) \)
9. for \( i = 1 \) to \( c \) do
10. \( p_i \leftarrow \frac{u'_{ij}}{t_i} \) end for
11. end while
12. end.

13. function \( \text{UPDATE}(N = (x_j, C, r), Z) \)
14. \( D \leftarrow \{||x_j - p_1||^2, \ldots, ||x_j - p_c||^2\} \)
15. \( \varphi \leftarrow \text{prototypePermutation}(x_j, D) \)
16. \( \hat{c} \leftarrow \text{validPrototypes}(x_j, D, \varphi) \)
17. for \( i = 1 \) to \( c \) do
18. if \( z_i < 0 \) do
19. \( u_{ij}' \leftarrow \text{membershipValue}(x_j, D, i, \hat{c}, \varphi, \beta) \)
20. \( (u^-, u^+) \leftarrow \text{membershipInterval}(x_j, r, D, i, \varphi, \beta) \)
21. if \( u^+ - u_{ij}' < \varepsilon \) and \( u_{ij}' - u^- < \varepsilon \) do
22. \( z_i \leftarrow u_{ij}' \)
23. end if
24. else
25. \( u_{ij}' \leftarrow z_i \)
26. end if
27. \( p'_i \leftarrow p'_i + h(u_{ij}') \cdot x_j \)
28. \( t_i \leftarrow t_i + h(u_{ij}') \)
29. end for
30. for \( N' \in C \) do \( \text{UPDATE}(N', \text{copy}(Z)) \) end for
31. end function \( \text{UPDATE} \)

The first part is quite self explaining. The variables \( P' \) and \( T \) represent global variables
(they are also valid in the function UPDATE) that are used to calculate the new prototype positions. The recursive update function is evoked as long as at least one membership value changes more than \(\varepsilon\). The update function itself is a little more complicated. Basically, it traverses the neighbourhood tree in depth first ordering by recursively evoking itself. The parameters in \(Z\) contains the information whether a set of data objects need further calculation and if not, the membership value is stored in \(Z\) for the corresponding prototype.

In line 14, the distances from \(x_j\) to all prototypes are stored in \(D\) because these values are used quite often (complexity \(O(c)\)). In the next step, the prototypes are sorted according to their distance (complexity \(O(c \cdot \log(c))\)). In line 16, the number of prototypes with larger than 0 membership function is calculated (complexity: \(O(c)\)). Then, for each prototype, the new membership value is calculated. In case \(r_i < 0\), it means the membership interval was not narrow enough, which means the membership value for \(x_j\) must be calculated. This is done in line 19 with \(O(1)\) complexity because \(\sum_{k=1}^{c} \frac{1}{d_{\phi(k)}}\) can be calculated once before the loop is started. Calculating the membership interval however (line 20), has a complexity of \(O(c)\), since \(\hat{c}_p\) and the above mentioned sum has to be recalculated with the changed distances for each prototype individually. If the resulting membership interval is narrow enough, all subsequent data objects are associated with the same membership value for prototype \(p_i\). This information is stored in \(r_i\) for use in the next recursions.

In line 27, the factors for the new prototype position are calculated. Finally, the recursion is performed in line 30 for all child nodes of \(N\) with a copy of the values of \(R\) because they might be changed differently in the subsequent branches of the neighbourhood tree. So the overall complexity for one iteration step is in

\[
\text{calculation for one node} \cdot \frac{n}{n \text{ nodes}} + \frac{c \cdot n}{\text{termination test}} = O(c^2 \cdot n).
\]

This implementation of the modified FcM is exactly the implementation of the mathematical definition. Unfortunately, it is not feasible for really many data objects, since the membership matrix \(U\) has to be stored. This is necessary because the very essence of the algorithm is the use of \(\varepsilon\) for optimizations. Hence the storage complexity of the algorithm is in \(O(n \cdot c)\) which might be too much in storage critical applications.

For means of termination, it is not necessary to store the membership matrix. It would be equally accurate, to test the convergence of FcM using the distance, prototypes move between two iteration steps. It is easy to show that the convergence in membership values is equivalent to the convergence in prototype position. However, if there is a threshold defined for convergence in prototype positions, it is not trivial to calculate a corresponding threshold for convergence in membership values. In a too harsh estimation, the membership interval that is used to optimize the clustering process would becomes extremely small. Therefore, these two parameter must be chosen separately by the user. The membership value interval can be based on an accuracy parameter \(\varepsilon_m\) while the convergence test is done with the threshold \(\varepsilon_p\).

With using a convergence in prototype positions, it is possible to cease storing the membership matrix during the calculation process. This gives room for even more optimization because now it is not necessary to traverse the entire neighbourhood tree. However, a few technical modifications to the neighbourhood tree are required. It is necessary to store the number of successors \(s[succ(N)]\) of a node \(N\) and the centre of gravity of all succeeding data objects: \(\nu = \frac{1}{s} \sum_{y \in succ(N)} y\). Both can be easily
calculated during the construction process of the neighbourhood tree, so that a node is expanded to

\[ N = (x_j, C, r, s, \nu) \]

### 6.3 Algorithmus (2. Modified FcM):

**INPUT:** Data set \( X \), function \( \delta : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \), number of prototypes \( c \), termination threshold \( \varepsilon_p > 0 \), membership value calculation accuracy \( \varepsilon_m > 0 \), a parameter \( \beta \in [0, 1] \)

**OUTPUT:** The set of Prototypes \( P \)

1. \( N_{\text{root}} \leftarrow \text{extNeighbourhoodTree}(X, \delta) \)
2. \( P \leftarrow \{ p_1 \leftarrow 0, \ldots, p_c \leftarrow 0 \} \)
3. \( P' \leftarrow \text{initialize}(\{ p'_1, \ldots, p'_c \}) \)
4. while \( \exists 1 \leq i \leq c : | p_i - p'_i | > \varepsilon_p \) do
5.  for \( i = 1 \) to \( c \) do \( p_i \leftarrow p'_i \) end for
6.  \( P' \leftarrow \{ p'_1 \leftarrow 0, \ldots, p'_c \leftarrow 0 \} \)
7.  \( T \leftarrow \{ t_1 \leftarrow 0, \ldots, t_c \leftarrow 0 \}, Z \leftarrow \{ 1, \ldots, c \} \)
8.  \text{UPDATE2}(N_{\text{root}}, Z)
9.  for \( i = 1 \) to \( c \) do \( p'_i \leftarrow \frac{p'_i}{t_i} \) end for
10. end while
11. end.
12. function \text{UPDATE2}(N = (x_j, C, r, s, \nu), Z)
13.  \( D \leftarrow \{ \| x_j - p_1 \|^2, \ldots, \| x_j - p_c \|^2 \} \)
14.  \( \varphi \leftarrow \text{prototypePermutation}(D) \)
15.  \( \hat{c} \leftarrow \text{validPrototypes}(x_j, D, \varphi) \)
16.  for \( i \in R \) do
17.      \( (u^-, u^+) \leftarrow \text{membershipInterval}(\nu, r, D, i, \varphi, \beta) \)
18.      if \( u^+ - u^- < \varepsilon_m \) do
19.         \( u \leftarrow \text{membershipValue}(\nu, D, i, \hat{c}, \varphi, \beta) \)
20.         \( Z \leftarrow Z \setminus \{ i \} \)
21.         \( p'_i \leftarrow p'_i + s \cdot h(u) \cdot \nu \)
22.         \( t_i \leftarrow t_i + s \)
23.      else
24.         \( u \leftarrow \text{membershipValue}(x_j, D, i, \hat{c}, \varphi, \beta) \)
25.         \( p'_i \leftarrow p'_i + h(u) \cdot x_j \)
26.         \( t_i \leftarrow t_i + h(u) \)
27.      end if
28.  end for
29.  if \( |Z| > 0 \) do
30.     for \( N' \in C \) do \text{UPDATE2}(N', \text{copy}(Z)) end for
31.  end if
32. end function \text{UPDATE2}

This second algorithm has several advantages over the first version. The storage complexity is in \( O(c + n) \) rather than \( O(c \cdot n) \). Also the calculation can be simplified by using the centre of gravity of a node’s successors. This was not possible in the
first version, because all membership values had to be computed to fill the matrix at all positions, so that the traversal of the neighbourhood tree had to be carried out completely.

In detail, the variables $P'$ and $T$ have the same meaning as in algorithm 6.2, they are used to calculate the new locations of the prototypes. The variable $Z$ holds now the indices of the prototypes for which a membership value needs to be calculated. However, the main difference to the first version of this algorithm is in line 4 where the iteration process is stopped when the difference in prototype position converges. All changes in UPDATE2 are a possible due to this change. The function UPDATE2 differs mainly in its behavior to previously calculated membership values. In its first version, the entire neighbourhood tree is traversed. In this version, only for a subset of all prototypes, the calculation is performed which is defined by $Z$ in line 17. So, if the membership value for the prototype $i$ is not already known, it is calculated in lines 17 until 29.

If the calculated membership value interval is narrow enough, the change in location of prototype $i$ is calculated using the with $s$ weighted membership value $u$ and $i$ is removed from $Z$ to prevent any further calculation in the subtree. For all prototypes the membership interval is not narrow enough, the calculation is done solely for $x_j$.

The recursion in line 31 is done only if there are prototype indices left in $Z$.

The drawback of the optimized algorithm 6.3 is, that the usage of the neighbourhood tree loses its justification provided by the termination threshold. On the other hand, this gives the user the advantage to manage calculation precision and termination threshold independently. Consider the example presented in Figure 4. The same data set is clustered with 4 different algorithms. The upper two examples are the FcM algorithm with the normal fuzzifier function, once in standard form (left) and once using neighbourhood information (right). For the other two examples, the polynomial fuzzifier function were used, again once in normal form (left) and once using neighbourhood information (right). As it can be seen, the clustering result is almost not influenced by using neighbourhood information. The tails of the prototypes show that even the clustering process is almost identical\(^2\) even if the membership interval of $\varepsilon_m = 0.1$ is rather large compared to the calculation accuracy. While this can be an advantage for the calculation process, it is often not helpful to apply too many parameters to an algorithm because it becomes too clumsy and complicated for users who do not know the internal calculation process of the algorithm.

\(^2\)The difference is not visible on the pictures, but the paths slightly different due to the approximation of using neighbourhood information.
7 Experimental Results

The four algorithms standard FcM, FcM using Neighbourhood Information (NFeC), FcM with Polynomial Fuzzifier Function (PFcM), and FcM with Polynomial Fuzzifier Function and using Neighbourhood Information (NPFeC), compared in this section, differ in their property of scaling. Runtime tests for these algorithms regarding the number of data objects and the number of prototypes are presented. The values of the parameter $\beta$, had no, or only very little influence on the runtime of PFcM and NPFeC. In all cases, the tests are performed using 2, 5 and 10 dimensional artificial data. It was not focused on the algorithm convergence properties because the convergence of the algorithms using neighbourhood information do not differ distinctly from those without. The average euclidean distance (in one experiment) in prototype positions is almost always below 0.001. Still, it is useful to compare FcM with and without polynomial fuzzifier functions.

Two different artificially created test environments are used, a hypercube with an edge...
EXPERIMENTAL RESULTS

Figure 5: Two examples for the test data sets: uniform distributed data (left) and gaussian distributed data objects in randomly positioned clusters

length of 1, filled with uniformly distributed data objects (Figure 5) and randomly placed cluster centres with normally distributed data objects (Figure 5 right). In the second case, there are always as many clusters in the data set as there are prototypes, the number of data objects in the data set is not effected by the number of prototypes. In each test, the algorithms perform 100 iterations, which is usually enough for convergence.

Since NFcM is already well discussed in [14], we do not discuss runtime differences due to the fuzzifier, a constant value of \( \omega = 2 \) is used in all cases. This value was chosen, because the polynomial fuzzifier function of PNcM and NPFcM is a linear combination of crisp clustering and fuzzy clustering with a fuzzifier of \( \omega = 2 \). Our tests have shown that even a relatively large maximal membership value interval of \( \varepsilon_m = 0.1 \) is usable for NFcM and NPFcM. The parameter \( \beta \) was set to 0.5. The data set and initialization of the prototypes were identical for all clustering algorithms.

In Figures 6 and 7, we present our test results. Always on the left hand side, we present the results of the test environment with uniformly distributed data objects while on the right side the environment with gaussian clusters is shown. The discussion regarding the algorithms properties is done below.

**Standard FcM**

Not surprisingly, standard FcM has linear complexity in the number of data objects, number of prototypes and number of dimensions in all examples. The shape of data sets does not effect the runtime of the algorithm.

**FcM using Neighbourhood Information**

The behaviour of NFcM is quite interesting and shows the potential of the optimization process and some problems in FcM with high dimensional (> 5 dimensions) data. The potential is well visible in the 2-dimensional data sets. If the data reaches a certain density, the runtime of this algorithm does increase because only groups of data objects
Figure 6: Runtime experiments for 2, 5 and 10 dimensions with variations in the number of data objects (x-axis, in 1000 data objects). The remaining parameter are: 5 prototypes, $\omega = 2$, $\beta = 0.5$, maximal membership interval length 0.1, 100 iterations, no termination due to converging prototype positions.

Figure 7: Runtime experiments for 2, 5 and 10 dimensions with variations in the number of prototypes (x-axis). The remaining parameter are: 50000 data objects, $\omega = 2$, $\beta = 0.5$, maximal membership interval length 0.1, 100 iterations, no termination due to converging prototype positions.

are used for calculation. This effect is well visible in Figures 6, line 'NFcM 2D'. Even though, the algorithm is linear in the number of prototypes, a higher number of prototypes leads to smaller sets of data objects during the calculation (Figure 7).

In higher dimensions, the density of the data object decreases considerably in Figure 6 left, lines NFcM 5D and 10D, if the data objects are uniformly distributed. The result is a huge increase of the runtime due to the missing optimization. In case of several gaussian distributed clusters (Figure 6 right), the density of the data objects is still high enough for the neighbourhood optimization to work. As it is said before, higher dimensional data leads to a reduced density and more prototypes leads to smaller sets of data objects during the calculation process. In combination, the both effects cause a huge increase in runtime which is even worse than standard Fuzzy c-Means because
the membership interval still has to be calculated for every data object (Figure 7 left), line NFcM 5D and 10D.

Figure 8: NFcM clustering result from a 10 dimensional data set, projected on 2 dimensions.

Very surprisingly is the runtime graph in Figure 7 right for NFcM 10D. The source for this effect is that FcM does not work for higher dimensions. In Figure 8, a 2D projection of a 10 dimensional data set is shown. All prototypes go to the centre of gravity of the complete data set, hence the membership values of all data objects are almost identical for all prototypes. This leads to huge sets of data objects, that can be treated identical, hence the optimization is very effective. Unfortunately, the clustering result is completely useless.

It is also worth mentioning again that the difference of the clustering result compared to standard FcM is almost identical. In data sets with maximal 5 dimensions or in data sets where the majority of the data objects is located in high density areas, this algorithm is almost independent to the number of data objects. Hence it is highly scalable in data size if the prototypes are well separated from each other.

**FcM with Polynomial Fuzzifier Function**

The runtime of this algorithm is no surprise, since it works basically the same way as standard FcM. The higher runtime is a result of the increased calculation complexity of the membership value. The prototypes have to be sorted and the a subset has to be calculated for each data object. Therefore, the algorithm has a runtime complexity of $O(c \log(c))$ for each data object. This makes it even less scalable in the number of prototypes than FcM.

Also for this algorithm, the clustering result in higher dimensions is questionable at best. But in contrast to Fuzzy c-Means, it is not completely useless. Figure 9 shows a 2-dimensional projection of the same 10 dimensional data set as in Figure 8, but this time clustered with PFcM.
FCM with Polynomial Fuzzifier Function and using Neighbourhood Information

Finally, this algorithm is a combination of the last two, it uses neighbourhood information as NFeM and has a fuzzifier function like PFcM. This combination has several advantages and some disadvantages. Due to the polynomial fuzzifier function, data objects in the surrounding area of a prototype have a membership value of 1. Hence the membership value to all other prototypes is 0 which leads to very large sets of data objects with a membership interval of 0 length. This can speed up the calculation process considerably in comparison to the other algorithms. The drawback, however, is that the prototypes have to be sorted for the membership value interval of each prototype. Because only one prototype is changed every time, this sorting process has a complexity of $O(c)$. But this must be done for each prototype, hence the calculation complexity for each data object is in $O(c^2)$.

The algorithm has the same problems with low density data sets as NFeM and which are enforced by the much higher runtime complexity. In data sets with well separated high density areas, the algorithm is faster, compared to PFcM. This contrast is well visible in Figure 7. In the low density case of uniform distributed data, the algorithm is much slower than all others. Only in the 2 dimensional case, the density is high enough so that the neighbourhood information can reduce the runtime complexity considerably. Like NFeM, NPFcM is highly scalable in data object size if the density areas are well separated and found by prototypes. That this is not always the case is shown in Figure 7 right. For each combination of parameter, only one runtime test is performed. Depending on the initialization, the algorithm separates the data well or less well which has a high impact on the runtime. If there are no high density areas, the algorithm has a very bad runtime performance, as can be seen in 7 left and 7 left.

The polynomial fuzzifier function prevents that the clustering result in higher dimensions is completely useless which makes it possible that the algorithm can use neighbourhood information which makes it faster than PFcM.
8 Conclusions

We presented and compared 4 prototype based algorithms and tested them on two families of artificial data sets. We showed that using sets of data object that are located close to each other can make FcM almost independent on the number of data objects and we have shown that a polynomial fuzzifier function can be used to make a FcM-based algorithm more useful for high dimensional data sets. The combination of both approaches can be fast, but it has a very bad performance if the data set is corrupted in the sense that it has no dense areas. But in such a case, clustering would not reveal any knowledge about a data set. Conclusively, it can be said that none of the presented algorithms is perfect for all data sets, so a rough knowledge about the data set is required to choose the best algorithm for a given problem.

9 Acknowledgments

This work was done at the DLR in Braunschweig in cooperation with the Technical University Braunschweig and the Otto-von-Guericke University of Magdeburg, Germany. We want to thank especially the kind reviewer who helped to increase the quality of this work with a lot of constructive comments.

References


*List of Todos