universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Design and Development of a BANG-File Clustering System“

verfasst von / submitted by

Florian Fritz, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2018 / Vienna 2018

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 926
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Wirtschaftsinformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Wien, März 2018

Unterschrift:

_____

Florian Fritz

Acknowledgements

I would like to express my sincere gratitude to my supervisor Univ.-Prof. Dipl.-Ing. Dr. Erich Schikuta for the continuous support of my master thesis with helpful comments, engagement, motivation and most of all, patience.

Contents

1 Introduction **8**
- 1.1 Cluster Analysis . . . 8
- 1.2 Clustering Techniques . . . 11

2 Grid Clustering **14**
- 2.1 Grid File . . . 14
  - 2.1.1 Splitting Operation . . . 16
  - 2.1.2 Merging Operation . . . 16
  - 2.1.3 Searching . . . 17
  - 2.1.4 Inserting . . . 17
  - 2.1.5 Deleting . . . 18
- 2.2 Clustering with Grid File . . . 19
  - 2.2.1 Idea behind GRIDCLUS . . . 19
  - 2.2.2 GRIDCLUS Algorithm . . . 21

3 BANG-File Clustering **23**
- 3.1 BANG File . . . 23
  - 3.1.1 Representation of the Data Space . . . 23
  - 3.1.2 Mapping Function . . . 25
  - 3.1.3 Logical Regions . . . 27
  - 3.1.4 Partitioning and Merging . . . 29
  - 3.1.5 Directory Structure . . . 31
  - 3.1.6 Advantages of BANG-File . . . 32
- 3.2 Clustering with BANG File . . . 33
  - 3.2.1 Density Index . . . 34
  - 3.2.2 Neighborhood . . . 35
  - 3.2.3 Dendogram . . . 36

4 WEKA **39**
- 4.1 Introduction . . . 39
- 4.2 WEKA Workbench . . . 39
- 4.3 User Interfaces . . . 40
- 4.4 Memory Management . . . 42
- 4.5 Package Management System . . . 42
- 4.6 Clustering with WEKA . . . 44

5 Software Documentation **47**
- 5.1 BANG File Implementation . . . 47
  - 5.1.1 BANGFile . . . 47

5.1.2 DirectoryEntry 49
5.1.3 GridRegion 49
5.2 Application Architecture 50
5.3 Developer Guide 52
5.3.1 Creating a new Clustering Method 52
5.3.2 Parametrization 52
5.3.3 Clustering Method Lifecycle 55
5.3.4 Clusterer Factory Design Pattern 56
5.3.5 Abstract Class Clusterer 58
5.3.5.1 Declaration 58
5.3.5.2 Constructor Summary 58
5.3.5.3 Method Summary 58
5.3.5.4 Methods 59
5.4 WEKA Package 61
5.5 User Guide 64
5.5.1 Installation 64
5.5.2 User Interface 65

6 Experiment **69**
6.1 Data Sets 70
6.2 Results 72

7 Conclusion **75**

1 Introduction

1.1 Cluster Analysis

Cluster analysis is essential in the field known as explorative data analysis. It is applied in many scientific and engineering disciplines and helps determining patterns or structures within data based on the data's various attributes. With these patterns, we can form various classes, also referred to as clusters, for the data. Data within a cluster is supposed to have a range of similar attributes while at the same time being as different to other clusters as possible.

Using cluster analysis to find convenient and valid organization of the data can help significantly in understanding the attributes interrelation within the data and if the data is grouped according to preconceived ideas. This may suggest and refine new experiments with the data. Cluster analysis does not require assumptions common to most statistical methods and as such it is referred to as unsupervised learning. In supervised learning we are provided with labeled training patterns which are then used to label new patterns appropriately. On the other hand, in the case of clustering we group unlabeled patterns into meaningful clusters. These labels are solely obtained from the data during the cluster analysis.

Organizing data into groups is fundamental for understanding the data and learning from it. Cluster analysis is the study of algorithms for grouping objects, where an object is described by a set of measurements or the relationships to other objects. These algorithms are designed to find structure within the data, but not to separate future data into categories [12].

The goal of a clustering method is to distribute n data points in m clusters C, with each cluster containing a minimum of one data point and a maximum of n data points.

$$
\begin{aligned}
&C = \{C_1, C_2, \ldots, C_m\} \\
&C_i = \{x_1, x_2, \ldots, x_n\} \qquad\qquad i = 1, \ldots, m
\end{aligned}
$$

Following definitions of a cluster are documented by Everitt [4].

1. "A cluster is a set of entities which are alike, and entities from different clusters are not alike."

2. "A cluster is an aggregation of points in the test space such that the distance between any two points in the cluster is less than the distance between any point in the cluster and any point not in it."

3. "Clusters may be described as connected regions of a multi-dimensional space containing a relatively low density of points."

Everitt's last two definitions assume that the patterns are represented as points in a value space. If we look at a two-dimensional plane, we can recognize clusters with different shapes and sizes. However, the number of clusters we see can depend on the resolution we view the value space with, and cluster membership could also change. Figure 1 demonstrates a data distribution in a two-dimensional value space that, if seen at different scales, can be perceived as having either four or up to twelve clusters. The problem in identifying clusters in the data here is to specify the pattern proximity or distance and how we should measure it.

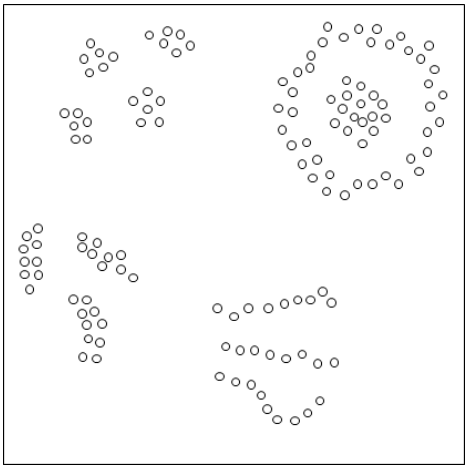

Figure 1: Clusters of data points in a two-dimensional value space

According to Jain and Dubes [12] the typical clustering process consists of the following phases:

1. Pattern representation (optionally including feature extraction and selection)
2. Definition of a pattern proximity measure appropriate to the data domain
3. Clustering (or grouping)
4. Data abstraction if needed
5. Assessment of output if needed

During the pattern representation phase the number of clusters and available patterns are specified as well as number, type and scale of the available attributes within the data set. Following that, one might use an appropriate subset of attributes, chosen via attribute selection or extraction, to use for the clustering to accomplish a better clustering result. Attribute selection is done by identifying a subset of the most effective and significant attributes of the data set. Attribute extraction, on the other hand, requires transformations of one or more attributes of the original input to produce more distinctive attributes.

In the next phase, the pattern proximity is being measured. This is typically done with a distance function over a pair of patterns. A variety of distance measurements methods can be used for clustering. One of the simplest distance measurement method to reflect dissimilarity between a pair of patterns is the Euclidean distance, which is essentially an ordinary straight-line distance between two points.

The clustering step of the process can yield different outputs. Clustering can either be *hard*, where the data is partitioned into groups, or *fuzzy*, where the patterns have a degree of membership to every cluster. Furthermore, hierarchical clustering algorithms will have a nested series of partitions that are split or merged depending on similarity to other partitions. With partitional clustering algorithms we can detect the partition that optimizes the clustering criterion the most.

During the data abstraction phase the extraction of a simple and more compact representation of the data set is attempted. In this context, simplicity may refer to the processing efficiency for automatic analysis or the comprehensiveness and intuitiveness for human reading. However, the extracted data also needs to be a compact description for every cluster.

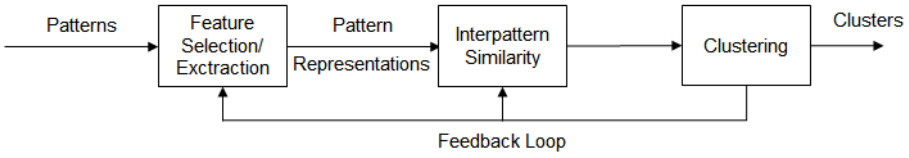

Figure 2: Stages of a clustering process

Regardless of whether the data contains clusters or not, all clustering algorithms will produce clusters. However, clustering algorithms may produce clusters of varying quality. Before the final assessment of the clustering output it is important to assess the data domain, because data that is not expected to contain clusters needs be processed in a different way by the clustering algorithm. Prior to performing the cluster analy-

sis, the input data should be examined to know if a cluster analysis can produce a meaningful output. The assessment of the clustering output can be subjective, as such there are no *gold standards*. However, objective assessment of the output's validity is possible via the existence of a clustering structure that could not have reasonably occurred by chance. Additionally, various statistical methods and testing hypotheses could be used to accomplish objective validation [13].

Validation can be categorized into the studies of external, internal and relative assessments [13]:

- The external assessment compares the output to a previous clustering result.
- The internal assessment tries to determine if the clustering structure is appropriate for the data.
- The relative assessment compares two clustering structures by measuring their relative merit.

1.2 Clustering Techniques

Clustering is a special kind of classification, so before we begin looking at different approaches to clustering data, we first take a look at different types of classification from figure 3. The differentiation over the tree's multiple levels are defined as follows [12]:

1. Exclusive vs. non-exclusive: In exclusive classification, each pattern belongs to exactly one class, or cluster. In non-exclusive classification, also referred to as overlapping classification, a pattern can be assigned to multiple classes.
2. Intrinsic vs. extrinsic: Intrinsic classification is called *unsupervised learning* because it only uses a proximity matrix to perform the classification and has no category labels used for a partition of patterns. Extrinsic classification, on the other hand, uses category labels on the patterns and the proximity matrix. With that, a discriminant surface must be established to separate patterns according to category.
3. Hierarchical vs. partitional: Depending on the structure imposed on the data, we can further subdivide exclusive and intrinsic classifications into hierarchical and partitional classifications. While a partitional classification produces a set of individual and separated

partitions, hierarchical classification produces a nested series of partitions with partitions *sharing* patterns.

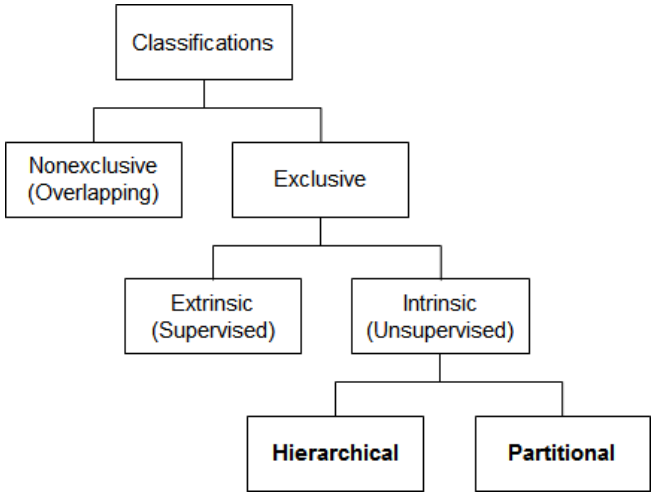

Figure 3: Tree of classification types

We refer to exclusive, intrinsic, partitional classification as clustering and to exclusive, intrinsic, hierarchical classification as hierarchical clustering. When looking at different approaches to clustering data there is a distinction between these two approaches. The algorithms for both clustering and hierarchical clustering have the following primary algorithmic options in common [13]:

- Agglomerative vs. divisive: Divisive, hierarchical classification is a procedure, where we start with one cluster containing all patterns and gradually subdivide it into smaller and more precise clusters. Agglomerative, hierarchical classification, on the contrary, starts with each pattern having its own cluster and then merges these clusters into larger clusters until all are in a single cluster. The same characterization can be applied to (partitional) clustering, where clusters are glued together to partitions (agglomerative) or a single all-inclusive cluster is fragmented into partitions (divisive).

- Exhaustive vs non-exhaustive: Exhaustive classification will cluster every single pattern, while non-exhaustive classification will leave some patterns, those of which with dissimilarity to all detected clusters, without cluster membership [3].

- Incremental vs. non-incremental: Incremental procedures handle one pattern at a time, while non-incremental procedures work with a set of patterns at once. The importance of data mining has forested clustering algorithms that require fewer scans through the pattern set to reduce number of patterns examined during execution. Processing large data sets can severely impact execution time and memory usage and will therefor affect the architecture of the algorithm.

Grid-Clustering, and in extension BANG-Clustering is ultimately an exclusive, intrinsic, partitional classification that in its primary algorithmic options is agglomerative, non-exhaustive and incremental. Note that despite BANG-Clustering using a nested architecture to organize its partitions, these partitions are still all occupying a separate value space, where nested partitions essentially *take over* the value space from their overlying partition.

2 Grid Clustering

Before we go into detail about the BANG file, we will first take a look at the grid file that it is based on as well as the grid-clustering algorithm called *GRIDCLUS*.

2.1 Grid File

The grid file was originally detailed by J. Nievergelt, H. Hinterberger and K.C. Sevcik in *The Grid File: An Adaptable, Symmetric Multikey File Structure* in 1984. While developing the grid file many data structures with multi-key access to records were nothing but an extension of a structure that was originally designed for single-key access. This resulted in drawbacks, in particular when dealing with highly dynamic data. Instead of a clear distinction between keys, i.e. primary and secondary, a bitmap approach was used for the grid file. The problem was treated as a data compression task where the grid file needs to adapt gracefully to the data that is being inserted or deleted and allows single record retrieval with an upper bound of only two disk accesses. It also is a symmetrical structure, were every attribute of the inserted data serves as a primary key and data is stored depending on each attribute, allowing efficient range queries with respect to all attributes [15].

The grid file essentially consists of two major elements, *buckets* and a *grid directory*. Buckets are the storage unit of the grid file and contain a predefined limited amount of patterns. The grid directory serves as the bucket management system and is made up of a grid array and linear scales. Linear scales partition the grid directory into grid blocks with each scale being a 1-dimensional array representing one of the k attributes of the data. Each entry in a scales array represents a (k-1)-dimensional hyper-rectangle that partitions the value space into two partitions orthogonal to the respective scale. As data points are inserted into the grid structure it adapts to the distribution of the data. The grid directory dynamically represents the partitions of the grid structure produced by the k scales. These partitions, or grid blocks, are stored in the grid array, which is a dynamic k-dimensional array containing all grid blocks. Every grid block subsequently refers to exactly one bucket. Buckets can be referenced by multiple adjacent grid blocks as shown in figure 4. The union of multiple grid blocks (or a single block) referencing the same data block is called a grid region, which will always be shaped like a k-dimensional hyperspace (i.e. a rectangular cube) [15].

Because grid regions can only reference one bucket, partitions have to

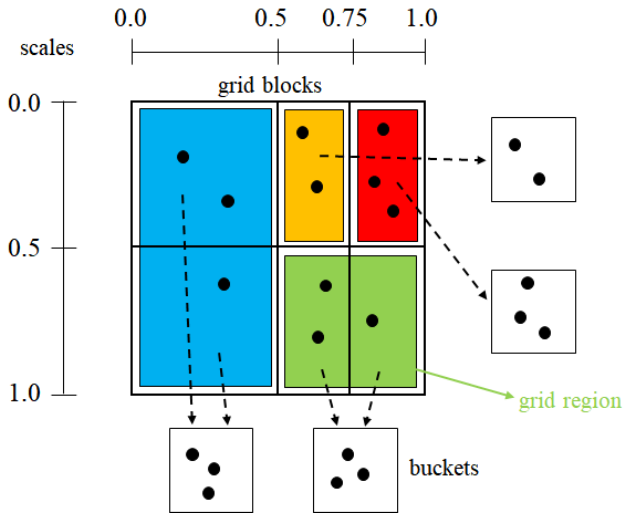

Figure 4: 2-dimensional grid structure

be arranged so that no grid region contains more patterns than can be contained within a bucket. If the addition of a pattern to a grid block within a grid region causes the associated bucket to overflow then the grid region, and its bucket, have to be partitioned again. The newly created partition, being orthogonal to the scale, will split all blocks lying across its plane with a (d-1) dimensional hyper-rectangle as shown in Figure 5, despite only a single bucket overflow occurring. Simultaneously, one bucket must be split into two. These two new buckets will each be referenced by a respective grid region created by the indirectly overflowing grid region. Meanwhile, for all other grid regions effected by this split, their newly partitioned grid blocks will remain in the same region and reference the very same bucket as their originating grid block [18].

Herein we can find a major weakness of the grid file. With a non-uniform data distribution, the ratio of grid block entries within the directory to actual buckets increases continuously with every split and the grid directory will eventually grow at an exponential rate. Most entries in the grid directory will end up being *dead weight* and reference empty buckets. Furthermore, this problem gets worse the more dimensions the data has [6].

The basis for all operations on the grid file, such as searching, inserting or deleting, lies in the splitting and merging of grid blocks. The reorganization of the grid structure after a split or merge should happen, barely perceivable, without negative or unforeseen consequences.

2.1.1 Splitting Operation

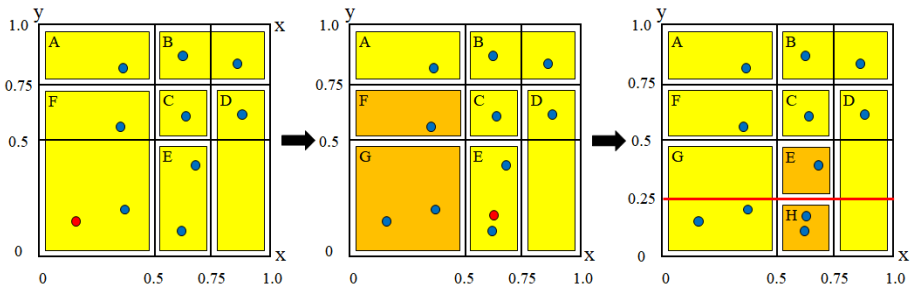

Figure 5: Splitting operations with bucket size of 2

Figure 5 displays the splitting operations occurring whenever a bucket overflows. Generally, a preexisting partition is preferred to a new partition across an entire dimension when splitting a region. The dimension to partition can be chosen either by cyclical order or by finding the least partitioned dimension in the grid file. In the left grid file in figure 5, inserting a pattern into region F would cause its associated bucket to overflow. A suitable partitioning through region F already exists in the grid array so that we do not need to create new partitions. Instead, we separate the partitions within region F into two regions, with which we distribute some records into a new region G and its bucket. Following that, in the grid file in the middle, an insertion of a pattern would again cause an overflow in the bucket of region E. In this case however, no suitable partition through region E exists, so we need to partition all grid blocks across the dimension y. Doing this allows us to form a new region H with the newly created partition. This however also effects the regions G and D, whose grid blocks are needlessly partitioned [15].

2.1.2 Merging Operation

If a regions bucket is relatively empty, it is possible to merge that region with a neighboring region, provided the merge does not cause the bucket to overflow. However, merging entries in the grid directory is often neglected as a subsequent split of the very region has a high probability.

There are two different systems to determine which regions may be suitable for merging with a selected region. The *Buddy*-system exclusively allows a region to only merge with regions it was previously split of from. The *Neighbor*-system, on the other hand, allows merging with all adjacent regions provided the resulting region is convex.

2.1.3 Searching

As previously mentioned a major advantage of the grid file is that it allows single record retrieval with an upper bound of only two disk accesses. To search for a pattern within the grid file, with an upper bound of two disk accesses, we need to do the following [11]:

1. First of all we need to determine the section of every dimension that the looked for pattern belongs to. This is done by looking at the entries of each dimensions scale array and retrieving the section for the patterns value in the respective dimension.

2. With the section of every dimensions value, we can pinpoint the exact grid block that contains the pattern.

3. Reading the entry for the grid block in the grid directory, we find the referenced data bucket.

4. Reading the data bucket itself, and thus performing our second disk access, we look for our desired pattern.

2.1.4 Inserting

In the example presented in figure 6 we attempt to insert a pattern with the value $(0.7, 0.2)$ into a grid file that has a maximum bucket population of two. After determining the grid block it should be inserted into, we notice that the regions bucket of this grid block is already filled with two patterns and therefor at capacity. Before the pattern can be inserted, the grid file has to be partitioned across one of the dimensions. In the presented example the chosen dimension to split across is dimension y, as portrayed by the dotted line. Once the grid region (and the grid block) E has been split into two, both with their very own data bucket, we are able to insert the pattern into the newly created region

While attempting to insert a new pattern, we need to follow a procedure containing various checks [11]:

1. Search for the grid block, that the pattern should be inserted into, and acquire the referenced bucket.

2. With the now loaded bucket, depending on its population, insertion may continue:

   (A) If the bucket has not reached maximum population, we can insert our pattern and end the operation.

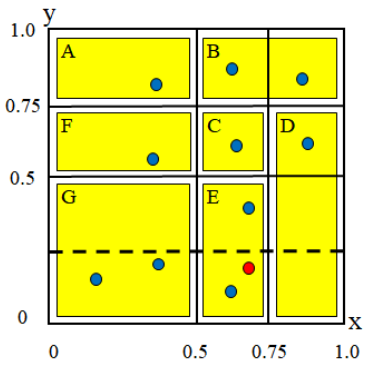

Figure 6: Inserting a pattern into a grid region at capacity

(B) If the bucket has already reached maximum population, then we need to create a new bucket either by using an already existing suitable partition or via a new splitting operation:

(i) In case the bucket is referenced by multiple grid blocks and a suitable partition already exists we can dissolve the grid region into two and have the grid blocks reference different buckets.

(ii) In case the bucket is referenced by only one grid block, we have to partition the grid block by splitting all blocks with a (d-1) dimensional hyper-rectangle on the chosen dimension.

2.1.5 Deleting

To delete an already inserted pattern from the grid file we need to do the following:

1. Look for the grid block and its referenced data bucket containing the pattern we wish to delete.

2. After deleting the pattern from the bucket we may chose a suitable grid block via the *Buddy*-system for a merging operation.

3. In case the patterns of our original grid block and the chosen *Buddy* grid block combined do not exceed the maximum number of patterns in a bucket, we may merge said blocks to a grid region, allowing it to reference a single bucket.

2.2 Clustering with Grid File

Clustering approaches based on the grid file structure are popular for mining clusters in large data sets with high dimensionality, where clusters can be formed around regions with the highest population densities. When dealing with a large number of patterns, conventional clustering algorithms may have to deal with exploding computational complexity, because of the need to compare every pattern to all others while calculating a dissimilarity matrix. Unlike conventional clustering algorithms, which take the data points themselves into consideration, grid-based clustering algorithms organize the value space surrounding the patterns. Because of this, grid-based approaches achieve a significant reduction in computational complexity when dealing with massive data sets.

A couple of clustering approaches exist that are based on the grid file, but in general they follow these basic steps [8]:

1. Creating the grid structure by partitioning the value space into grid regions based on the inserted data.
2. Determining the density of patterns within every grid region.
3. Sorting the grid regions according to their population densities.
4. Identifying high concentration of grid regions (and therefore patterns) as cluster centers.
5. Topologically traversing neighbor grid regions and adding them to cluster centers.

2.2.1 Idea behind GRIDCLUS

We will focus on *GRIDCLUS*, a hierarchical algorithm for clustering very large data sets, proposed by Schikuta in *Grid-Clustering: An Efficient Hierarchical Clustering Method for Very Large Data Sets* in 1996.

As explained in the previous chapters, the grid file structure stores patterns according to their actual values in a d-dimensional value space. Rather than organizing the patterns themselves, the multidimensional grid structure organizes the value space surrounding the patterns and groups them into blocks and regions. These blocks are d-dimensional hyper-rectangle containing a limited amount of patterns upwards to the *block-size bs*. Given d dimensions, a set of n patterns $X = \{x_1, x_2, \ldots x_n\}$ with each being a tuple containing d attributes, and blocks $B_1, B_2, \ldots, B_m$

containing the pattern set X, the following properties are satisfied:

$$\begin{aligned} &\forall x_i, x_i \in B_j \\ &B_j \cap B_k = \emptyset, \text{if } j \neq k && \text{and} \\ &j \neq \emptyset && \text{and} \\ &\bigcup_{j=1}^{m} B_j = X \end{aligned}$$

At initialization, every block is a cluster and every cluster has a pattern cardinality of no more than the block-size bs allows. The GRIDCLUS algorithm then clusters the grid blocks $B_i (i = 1, 2, \ldots, m)$ together with their stored patterns X into a nested sequence of nonempty and disjoint clusterings C. $C_{u1}, C_{u2}, \ldots, C_{uW_u}$ represents the clustering at the u-th iteration and W_u is the number of clusters available in the u-th iteration.

Because every block by definition represents a unique cluster with patterns, at iteration $u = 0$ we initialize the algorithm with W_0 being the number of all blocks m and C containing m clusters with a single block in each of them $C_{0j} = B_j (j = 1, 2, \ldots, m)$ [18].

GRIDCLUS uses the block information of the grid structure and attempts to form clusters with surrounding blocks. To do this, the algorithm first calculates a density index for each block that is defined by the number of patterns in a block p_B and the spatial volume of a block V_B. The spatial volume of a block B is the Cartesian product of the extents e of block B in each dimension.

$$V_B = \prod_{i=1}^{d} e_{B_i} \qquad\qquad i = 1, \ldots, d$$

The density index D_B of block B is the ratio of number of patterns p_B to the spatial volume V_B of block B.

$$D_B = \frac{p_B}{V_B}$$

Sorting the blocks B according to their density index we get an ordered sequence $< B_{p(i)} >$, with $p(i)$ being a permutation of the index i defining the sorted order of the blocks. In case two or more blocks have the same density index, as an example in figure 7 shows, they are referred to as *ties*.

The blocks with the highest density index initially become clustering centers. The remaining blocks are then iteratively clustered in the order of their density index, thereby merging with already existing clusters or

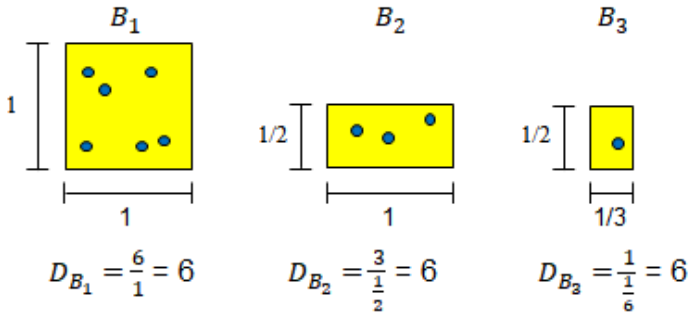

Figure 7: Three *tied* blocks with $D_B = 6$

building new cluster centers. The algorithm only allows blocks to merge with adjacent clusters (*neighbor*). A neighbor search procedure works in a similar way as the traversal of a graph to find the spanning tree, with blocks representing the nodes of a tree. The neighbor search procedure starts from a cluster center and inspects adjacent blocks, if a neighbor is found the search continues recursively from this block. This procedure can be done with a depth-first-search algorithm, where only blocks with a density index equal or smaller to the current block are inspected [8].

2.2.2 GRIDCLUS Algorithm

The grid clustering algorithm consists of five main components:

- Inserting the patterns and building the grid file structure.
- Calculating the density index of every block.
- Sorting the blocks according to their density index.
- Identification of blocks with highest density indexes as cluster centers.
- Recursive traversal of neighbor blocks beginning from cluster centers.

The main procedure of *GRIDCLUS* shown in algorithm 1 iteratively processes all blocks with the recursive procedure *NEIGHBOR-Search*, found in algorithm 2, that assigns them to already existing clusters. The number of the current iteration is u. $W_{[u]}$ stores the number of clusters in iteration u and $C_{[u,v]}$ stores the clustered blocks of run u and cluster v. As previously explained we initialize the algorithm with $W_0 = m$ (number of all blocks) and $C_{0j} = B_j (j = 1, 2, \ldots, m)$ [18].

Algorithm 1 GRIDCLUS algorithm

```
1: procedure GRIDCLUS
2:     Initialize: u := 0, W[] := {}, C[][] := {}
       Create grid structure and calculate block density indexes D_B
       Generate ordered block sequence S :=< B_1', B_2', ..., B_b'
       Mark all blocks 'not active' and 'not clustered'
3:     while "not active" block exists do
4:         u := u + 1
5:         mark first B_1', ..., B_j' with equal density index "active"
6:         for all 'not clustered' block B_k' := B_1', ..., B_j' do
7:             create new cluster set C[u]
8:             W[u] := W[u] + 1
9:             C[u, W[u]] ← B
10:            Mark block B_k' 'clustered'
11:            neighborSearch(B_k', C[u, W[u]])
12:        for all 'not active' block B_l do
13:            W[u] := W[u] + 1
14:            C[u, W[u]] ← B_l
15:        Mark all blocks 'not clustered'
```

Algorithm 2 NEIGHBOR-SEARCH algorithm

```
1: procedure NEIGHBOR-SEARCH(B, C)
2:     for all 'active' and 'not clustered' neighbor B_n of B do
3:         C ← B_n
4:         mark Block B_n clustered
5:         neighborSearch(B_n, C)
```

3 BANG-File Clustering

The BANG file, short for Balanced And Nested Grid, was originally detailed by Michael W. Freeston in *The BANG file: A new kind of grid file* in 1987 [6]. It is a multidimensional structure of the grid file type that is, however, fundamentally different from previous grid file designs in common underlying properties. For instance, it has a tree structured directory with the self-balancing property of a B-tree that, unlike previous grid file designs, continues to expand at the same rate as the inserted data irregardless of the data's distribution. Moreover, its partitioning strategy more accurately reflects the clustering of points in the data space and also adapts gracefully to changes in distribution.

3.1 BANG File

The BANG file structure partitions the data space into regions with successive binary divisions on dimensions, making it an interpolation-based grid file.

3.1.1 Representation of the Data Space

As explained in the chapters about the grid file, when the distribution of data becomes less uniform the directory expansion approaches an exponential rate and most directory entries will end up referencing empty block regions. In interpolation-based grid files however, there is always only one directory entry for each data bucket thanks to its representation of the data space where block regions are represented explicitly. This is achieved by partitioning the data space into a hierarchical set of grid regions, each of which is identified by a unique pair of keys (r, l) consisting of the region number r and the level number l. Regions, by binary partitioning as shown in figure 8, are divided into two equally shaped regions one level further down in the B-tree than the original region.

As such, the root region $(0, 0)$ contains regions $(0, 1)$ and $(1, 1)$ and through subsequent levels the respective regions contained within them. In figure 9 we can see how this allows us to partition the data space at varying levels of granularity appropriate to the data distribution. The dimension to partition has to be determined by a fixed, typically cyclic, sequence.

The BANG file follows the following two axioms [6]:

1. The union of all the sub-spaces into which the data space has been partitioned must span the entire data space.

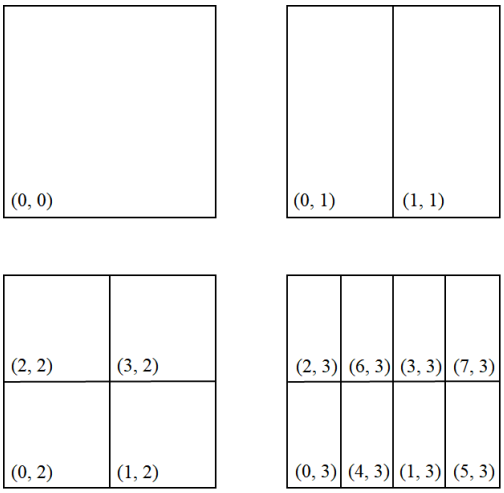

Figure 8: Numbering scheme (*region number*, *level*) for grid regions

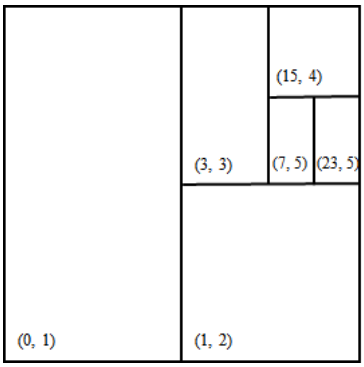

Figure 9: Regions partitioning a non-uniform data distribution

2. If the data space has been partitioned into two sub-spaces which interact, then one of these sub-spaces completely encloses the other.

The second axiom is responsible for the exceptional performance characteristics of the BANG file. It allows nested regions, as shown in figure 9, and enables algorithms to balance the distribution of data points between block regions and reach more compact data structures by redistribution [19]. The balancing algorithm guarantees that there are never any empty block regions, while partitioning and recombination algorithms provide flexibility in regards to changes in the data's distribution [6].

3.1.2 Mapping Function

To map the coordinates of a pattern to the grid region in which it lies (at each defined level of granularity) in the data space we use a set of hash functions. Before that however, we have to transform the set of values from the pattern to the scale $(0 \dots 1)$ of every dimension. With the transformed data values for every dimension we can accurately determine the grid region by using each dimension's scale. With the pattern value for dimension i being p_i and the level of granularity in dimension i being l_i, the scale value representing the coordinate on the dimensions scale d_{i,l_i} is defined as:

$$d_{i,l_i} = [p_i * 2^{l_i}] \qquad\qquad i = 1, \dots, d$$

and the scale value of a lower (closer to root) level of granularity j_i is calculated from d_{i,l_i} as

$$d_{i,j_i} = [\frac{d_{i,l_i}}{2^{(l_i - j_i)}}] \qquad\qquad j_i = 0, \dots, l_i$$

As such, we do not need the original value from the pattern to calculate the scale value of the higher level regions.

While the mapping of the pattern values to the region number can be done in various way, the numbering scheme seen in figure 8 makes mapping relatively simple. The mapping function can be defined by following rules [6]:

- Level $l + 1$ is created on top of level l.
- The number of possible grid regions at level l is 2^l.
- Grid regions (r, l) are divided into two uniquely numbered regions at a higher level $(r, l + 1)$ and $(r + 2^l, l + 1)$.
- Region numbers r and $r + 2^l$ result from extending the binary representation of r by one most significant bit.
- The value domain of a dimensions scale is doubled when the level of granularity of dimension i increases from l_i to $l_i + 1$. This equates to an extension of the binary representation of the scale value by one least significant bit.

Based on these rules it can be concluded that the single grid region in level 0 is assigned region number 0 and that each region number of

level l (for $l > 0$) can be represented by l bits. With that we can define a function $bits(p)$ to determine the minimal number of bits required to represent the value p as [3]:

$$bits(d_{i,l_i}) = l_i, \qquad \forall l_i \geq 0, (i = l, \ldots, n)$$

Moreover, with the sum of all sub-levels comprising the complete value space,

$$l = \sum_{i=1}^{n} l_i, \qquad \forall l_i \geq 0$$

it follows that

$$bits(r) = \sum_{i=1}^{n} bits(d_{i,l_i}) = l, \qquad \forall l_i \geq 0.$$

With this we can obtain the mapping of a set of region coordinates to a unique region number simply by concatenating the binary representations of the scale values in a predefined order, while omitting coordinates of level 0. When a new partition is introduced in a dimension, like in figure 8, we generate two new regions at the next level $l + 1$. Of the two new regions, one has the same region number r as the enclosing region and the other has its region number raised to $r+2^l$. This increase in the region number is represented by extending the binary representation of the region number by one (most significant) bit, while extending the binary representation of the new level's scale value by one (least significant) bit. A fixed cyclic partitioning through the dimensions is advisable, as the value domain of each dimension is between 0 and 1 [18].

Algorithm 3 MAPPING algorithm

```
1: procedure MAPPING(d[i])
2:     Initialize: r = 0, offset = 1, k = 0
3:     while k < l do
4:         i = k mod n + 1
5:         j = k ÷ n
6:         if l_i ≥ j then
7:             r = r + offset * b_{i,j}
8:             offset = offset * 2
9:             k = k + 1
       return r
```

For the corresponding *mapping* algorithm 3, let $b_{i,j}$ be the j^{th} bit of coordinate d_i starting from the least significant bit.

For an example with two dimensions we will use the BANG file structure of figure 9 and map the data record $[0.9; 0.7]$ to a region number:

$l_1 = 3$ //level of granularity on dimension x

$l_2 = 2$ //level of granularity on dimension y

$l = l_1 + l_2 = 5$ //total level of granularity in BANG file

$d_1 = [0.9 * 2^{l_1}] = [0.9 * 8] = 7$ //scale value of dimension x

$d_2 = [0.7 * 2^{l_2}] = [0.7 * 4] = 2$ //scale value of dimension y

scale value $d_1 = 7$	1	1	1		
scale value $d_2 = 2$	1	0			
	bit string length l_i				

By bit-interleaving the scale values of both dimensions, reading from the most significant bit to the least significant bit (left to right) while alternating between the scales in the same sequence the partitioning was done (top to bottom), we get the bit representation 10111. This results in the region number $r = 23$ at level $l = 5$ (length of bit representation). Consider that when the binary representation of scale value d_i does not have the length l_i, that binary representation will be left padded with 0s.

3.1.3 Logical Regions

Looking at figure 10 we have a BANG file structure where the data space S has been partitioned into two grid regions R1 and R2. Grid region R1 encloses the entirety of the data space and grid region R2 is enclosed (or *nested*) within R1. This is possible due to the second axiom of the BANG file described in chapter 3.1.1, which proclaims that if data space has been partitioned into two interacting sub-spaces which interact, then one completely encloses the other.

The grid regions R1 and R2 in figure 10 define two subspaces S1 and S2. Subspace S2 is simply defined by the space that is enclosed by R2. However, subspace S1 is defined by the space that is enclosed by R1 minus the space that is enclosed by S2. Subspaces in the BANG file can be defined by a set of grid regions, with one convex region enclosing all other regions of the set. When determining the subspace of a specific region, the subspace of regions that are enclosed within the region need to be subtracted. As such, to define the subspace S2 we only require R2,

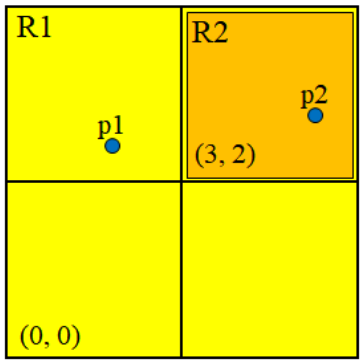

Figure 10: Grid region R2 nested within grid region R2

whereas to define the subspace S1 we require a set of both R1 and R2. This means that subspaces are not necessarily grid regions themselves and also do not have to be hyper-rectangles. They can contain concavities and possess internal, as well as external, boundaries. Furthermore, a subspace my be disjoint and composed of subspaces that do not intersect and do not share a common boundary [18].

As the subspaces are purely logical constructs from sets of grid regions they are also referred to as *logical regions*. The grid directory only stores grid regions, logical regions can only be determined through the stored grid regions. As a grid region may not *own* the entire subspace that it encloses, we are confronted with the fact that despite patterns in data buckets being technically mapped to grid regions, they need to be treated as if they were mapped to a logical region. Therefore, The balancing and distribution of patterns by the partitioning algorithm is done between logical regions.

While mapping a pattern to a specific grid region, we need to consider the danger of non-deterministic mapping results for nested regions. To ensure the mapping function assigns a pattern to the correct correct logical region, the search for grid regions begins at the very highest level and continues to the very bottom level (as in closer to root). This assures that no ambiguity arises in the mapping of patterns to logical regions and their data buckets since the mapping function always finds the grid region at the highest level.

As mentioned earlier the grid directory contains unique region identifiers, consisting of the region number r and the level l. Looking back at the BANG file structure in figure 10, the logical region of R2 is represented in the grid directory by the region set $\{(3, 2)\}$, while the logical region of

R1 is represented by the region set $\{(0,0);(3,2)\}$. When adding pattern p2 to the file the mapping function is applied to the patterns attributes and generates the region identifier $(3,2)$. As described in a previous chapter, with this region identifier we can also generate enclosing grid regions. For grid region $(3,2)$ that would be grid regions $(1,1)$ and $(0,0)$. When searching for the highest level grid region that is recorded in the directory that encloses p2, the search stops at $(3,2)$, which contains a pointer to the data bucket for the logical region R2.

However, when mapping the pattern p1 the region identifier $(2,2)$ will be generated, since that would be the theoretical highest level grid region based on the current level of granularity of ever dimension. This region identifier is not recorded in the grid directory so the search will continue looking for a lower level entry in the grid directory, eventually encountering the entry for grid region $(0,0)$. Therefore, despite the directory not containing any explicit information about logical regions, the correct placement of patterns in logical regions is guaranteed due to the search order within the grid directory [6].

3.1.4 Partitioning and Merging

Whenever the insertion of a pattern causes a data bucket to overflow the partitioning algorithm is invoked. As shown in algorithm 4, the logical region corresponding to the data bucket is first partitioned into two logical regions with a *buddy-split*. The regions do not remain *buddy* regions however, instead the region with less patterns is moved up a level so it encloses the other. Following that, after having partitioned the initial region, the enclosed region (with more patterns) is again partitioned recursively with the redistribution algorithm 5. This is done so long until a region has been split and the created region with a higher pattern population ends up having a lower population than the region that is enclosing the region that was originally split (the comparison is done over three levels: *enclosing region - split region - region created by split*). With the created region having a lower population than the enclosing region, the ideal balance has been reached and the buddy split can be undone. If the ideal balance is attained after the first split, *buddy regions* are created as seen in the examples of figure 11. If not however, then a new logical region is created in the boundary of the partitioned logical region [6].

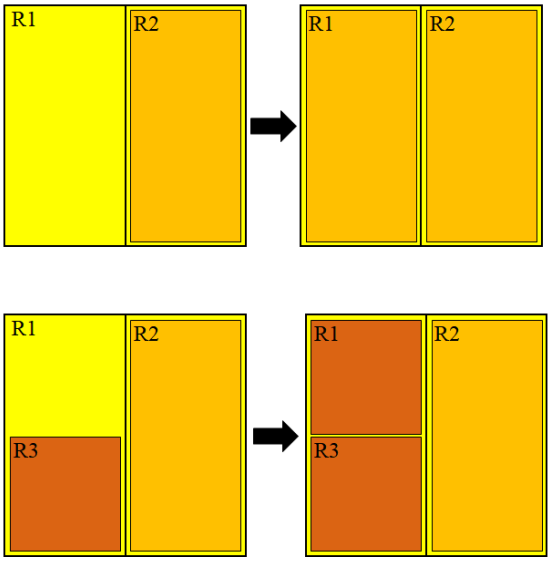

Figure 11: Creation of buddy regions

Algorithm 4 Split-Region algorithm

1: **procedure** SPLITREGION(*region*)
    `Manage the buddy-split of the region and the following redistribution`
2: *buddySplit*(*region*) ▷ produce *sparseRegion* and *denseRegion*
3: *region* := *sparseRegion* ▷ *sparseRegion* becomes enclosing *region*
4: *redistribute*(*denseRegion*, *region*) ▷ redistribute for ideal balance

Algorithm 5 Redistribute algorithm

```
1: procedure REDISTRIBUTE(region, enclosingRegion)
       To ensure a nicely balanced tree we perform
   redistribution after a region split
2:     buddySplit(region)    ▷ produce sparseRegion and denseRegion
3:     if enclosingRegion.population < denseRegion.population then
4:         mergeRegion(enclosingRegion, sparseRegion)        ▷ merge
   sparseRegion into enclosingRegion
5:         denseRegion := checkTree(denseRegion)        ▷ find correct
   position of the region
6:         if enclosingRegion.population < denseRegion.population
   then
7:             redistribute(denseRegion, enclosingRegion)        ▷
   redistribute if best balance not yet reached
8:     else
9:         clearBuddySplit(region)        ▷ Undo the buddy split
```

The merging of logical regions in the BANG file is much simpler than in other grid files as no deadlock checking is required and the recombination does not have to be in the exact reverse order of partitioning. When the population of tuples in a logical region goes below a predetermined minimum the algorithm first attempts to recombine it with a region it encloses beginning with the smallest (at highest level). If no such region exists or the recombination would cause an overflow of a bucket, an attempt to recombine the region with its *buddy region* is made. Finally, if this fails too, an attempt is made to recombine it with its enclosing region, which must always exist (except in the case of the *root* region $(0, 0)$) [3].

3.1.5 Directory Structure

The directory of the BANG file is implemented as a tree structure with the number of levels determining the depth of the tree as seen in figure 12. All levels above the root level are partitions of partitions and if a nodes population overflows, it is again split according to the partitioning algorithm. Grid regions themselves are essentially treated as data points and divide the data space into logical regions. Every node, that is a leaf or only has a single successor, is a grid region that helps forming a logical region in the data space. Additionally, these logical regions can be partitioned by logical regions at a higher level as was previously described in chapter 3.1.3 [6].

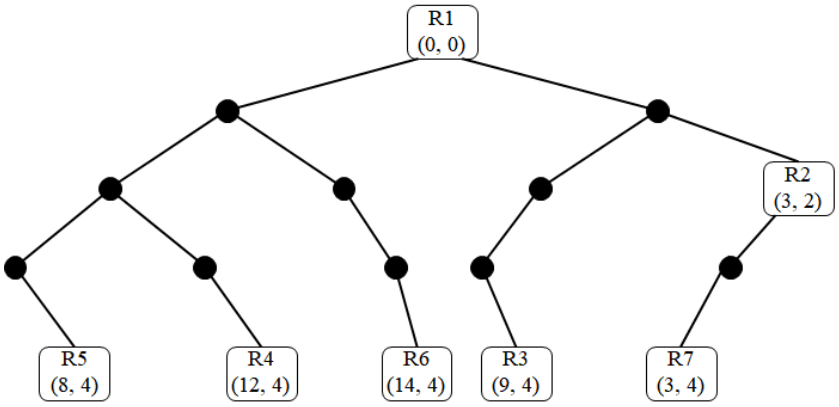

Figure 12: Binary tree storing the directory of a BANG file

When looking for a specific region we only need to look at the region identifiers, which are an ordered set of keys consisting of the region's region number r and the region's level l. We apply the algorithm that was used in chapter 3.1.3 for finding the correct placement of patterns in logical regions by beginning the search at the nodes that are furthest away from the root with a *Reverse Level Order Traversal*. With the partitioning scheme of the directory and the balancing algorithm, which redistributes and merges in the same order, we end up with a very compact and balanced structure similar to a one dimensional B-tree [3].

3.1.6 Advantages of BANG-File

Based on Hinrich's case study Freestone reached following conclusion [5]:

> So the conclusion which we finally draw from the analysis is that:
>
> 1. for a uniform data distribution in two dimensions, the SG and BANG data file will be about the same size, but the BANG directory will be about 1.5 times the size of the SG directory; for three dimensions the directories will be about the same size; for four or more dimensions the BANG directory will be smaller;
> 2. for single or multiple data clusters, the BANG data file can be expected to be between 10% and 15% smaller than the SG data file, and the BANG directory may be

about the same size or perhaps 2 or 3 times smaller than the SG directory, provided that the data distribution of the clusters, and within the clusters, is approximately uniform;

3. the size of the BANG directory will always be proportional to the size of its associated data file, but the SG directory may expand much more rapidly if the data distribution is non-uniform over a substantial proportion of the active data space - notably in the case of correlated data; in such a case the BANG directory could be two orders of magnitude smaller than the equivalent SG directory;
4. the performance of the BANG file relative to the SG file improves as the number of dimensions increases.

3.2 Clustering with BANG File

Many grid file clustering algorithms share similar characteristic problems in specific situations. These problems can be briefly summarized as [21]:

- For specific clusterings the memory requirements of the directory increases over-proportionally compared to the size of the data set, and
- the performance of the grid file clustering decreases with increasing dimensionality (number of pattern attributes) of the data set.

The BANG file, which is derived from the multidimensional grid file data structure, was designed to overcome these drawbacks. It shows not only better behavior for the listed problems, but also adapts more accurately to clusterings in the data's value space [6].

Similar to the grid file, the BANG file organizes the value space surrounding the data values, instead of comparing the data values themselves. The data values represent patterns in a k-dimensional value space and are, in a first step, inserted into the BANG file structure. These patterns are stored in the grid directory preserving their topological distribution. In figure 13, a 2-dimensional value space is presented with 10.000 patterns and 70 % of data clustered in three centers. For comparison, in figure 14 we can see the respective BANG file structure organizing the patterns in a set of encasing rectangular blocks. A block is rectangularly shaped and contains up to a definable maximal number of p_{max} patterns. $X = (x_1, x_2, \ldots, x_n)$ is a set of n patterns and x_i is a pattern consisting

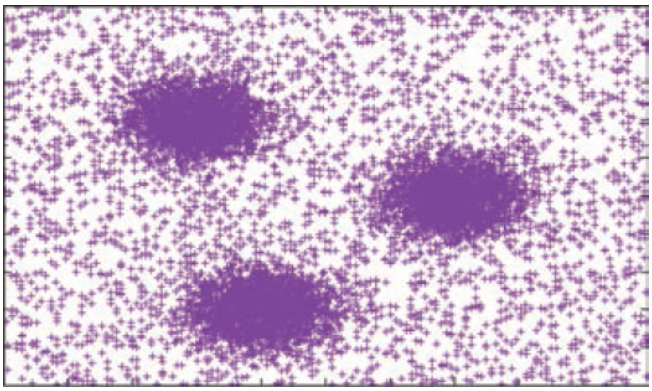

Figure 13: 2-dimensional value space example

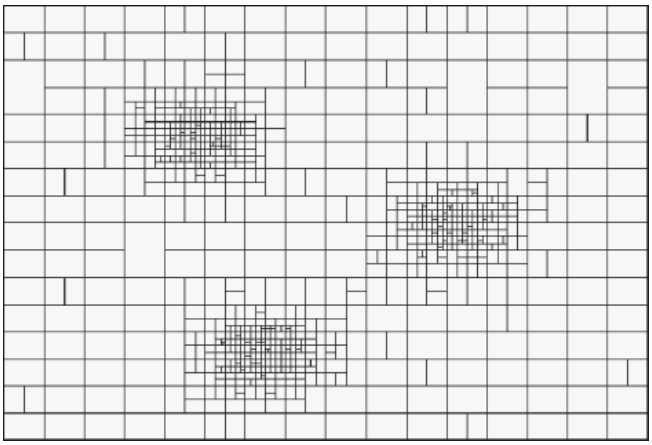

Figure 14: 2-dimensional BANG file structure of figure 13

of a tuple of d describing features $(p_{i1}, p_{i2}, \ldots, p_{id})$, where d is the number of dimensions. Using the block information of the BANG-structure, the algorithm identifies cluster center and clusters the patterns by an iterative neighbor algorithm accordingly to their surrounding blocks [21].

3.2.1 Density Index

Analogously to the GRIDCLUS algorithm from chapter 2.2.1, the BANG-Clustering algorithm calculates a density index for every grid region that is defined by the number of patterns within the region p_R and the spatial volume of the region V_R. More precisely, we use the spatial volume of the logical region. As we have learned in the chapter about logical regions,

because of the BANG file's ability to have nested regions, a region might not *own* the entire space it encloses. To get an accurate spatial volume of a logical region, we have to recursively retrieve all nested regions within the region and subtract their spatial volume.

The spatial volume of region R is the Cartesian product of the extents e of region R in each dimension minus the spatial volume of all nested regions inside the region.

$$V_R = \prod_{i=1}^{d} e_{R_i} - \sum_{j=1}^{n} V_{R_j}$$

The density index D_R of region R is the ratio of number of patterns in the region p_R to the spatial volume of the region V_R.

$$D_B = \frac{p_B}{V_B}$$

Regions with the highest density index show the highest pattern concentration and become our initial clustering centers. The remaining blocks are clustered iteratively in the order of their density index, merging with existing clusters or building new cluster centers in the process. However, only blocks adjacent to a cluster, so called *neighbors*, can be merged [20].

3.2.2 Neighborhood

When talking about neighborhood inside of a BANG file we can differentiate between two types. On the one hand we have *normal neighborhood* when looking at actual regions and on the other hand we have *refined neighborhood* when looking at logical regions. Furthermore, we can define the *degree of neighborhood* via the dimensionality of the area the regions touch on. This area's dimensionality can vary between 0 (representing a point) and $d - 1$ (representing a $d - 1$ dimensional hyperplane). In a 2-dimensional structure such as in figure 15 the *degree of neighborhood* can be 0, which means regions only need to touch at a point in the data space, or 1, which means regions need to touch along an edge. In a 3-dimensional structure they could additionally require a plane touching, making their *degree of neighborhood* 2. If we set the required *degree of neighborhood* to 1 for the grid directory in figure 15, we have *normal neighborhood* between regions R2 and R1, R3, R6 and R7, and a *refined neighborhood* between regions R2 and R1, R6 and R7 [3].

Searching for neighbors is done by comparing the scale values of the grid directory. Comparing the scale values of regions that are on the same level

can be done directly. However, if the regions are on different levels, the region in the lower level has to be transformed to the level of the other region to compare them. When looking at the grid directory in figure 15 we can see that region R2 and region R6 are neighbors, but R2 is on level 2, while R6 is on level 4. For that reason, we need to transform R2 from level 2 (with scale values $x = 1$ and $y = 1$) to level 4. As a region from level 2 is of course larger than from level 4, the resulting scale values will have the ranges $x_{min} = 2$, $x_{max} = 3$ and $y_{min} = 2$, $y_{max} = 3$. R6 on level 4 has the scale values $x = 1$ and $y = 3$ and is therefore touching (the transformed) R2 along an edge (y dimension), making the *degree of neighborhood* of the two regions 1.

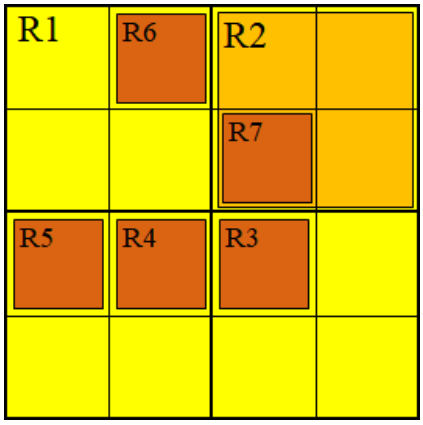

Figure 15: Neighborhood - Grid directory of tree in figure 12

With the region identifiers being an ordered set of keys, the algorithm can search for possible neighbor regions by accessing these keys. This is especially efficient since the administration of the region identifiers is done with the grid structure as a binary tree, as shown in figure 12, where comprising regions are easily found by backtracking the path to the root level [20].

3.2.3 Dendogram

With the density indexes of all regions calculated and the regions sorted by decreasing order, the next step in the clustering algorithm is to build the *dendogram*. The dendogram is an ordered list of regions filled by the dendogram algorithm 6.

Algorithm 6 Dendogram algorithm

1: **procedure** DENDOGRAM
2: **Initialize:** $dendogram \coloneqq []$
Calculate density indexes and generate region list sorted by descending order of density $S \coloneqq < R_{1'}, R_{2'}, \ldots, R_{n'}$
3: $dendogram \coloneqq R_{1'}$ ▷ put highest density region in $dendogram$
4: $remaining \coloneqq R_{2'}, \ldots, R_{n'}$ ▷ rest of regions in $remaining$
5: **while** `'regions in` $remaining$ `list'` **do**
6: $findNeighbors(dendrogram.region, remaining)$ ▷ add neighbors of current $dendogram.region$ and add them to $dendogram$
7: Continue with next region in $dendrogram$

The region with the highest density index is initially added to the dendogram and acts as our first cluster center. Iterating through the remaining regions we look for regions that are a neighbor of a region within the dendogram (initially only our starting region) and add them in descending order to the dendogram like in figure 16. This search for neighbor regions is done recursively for every processed and added region. Following that, detected regions are placed on the right hand side of the last processed region in the dendogram by the following rules [3]:

- Is region R1 a neighbor of region R2 and region R3 a neighbor of R2 and $R1 > R2 > R3$, then these three regions $\{R1, R2, R3\}$ form a cluster when R3 is being processed.
- Is R1 a neighbor of R2 and R3 a neighbor of R2 but $R1 > R2 < R3$, then the clusters R1 and R3 are merged when R2 is being processed to form the cluster $\{R1, R2, R3\}$.

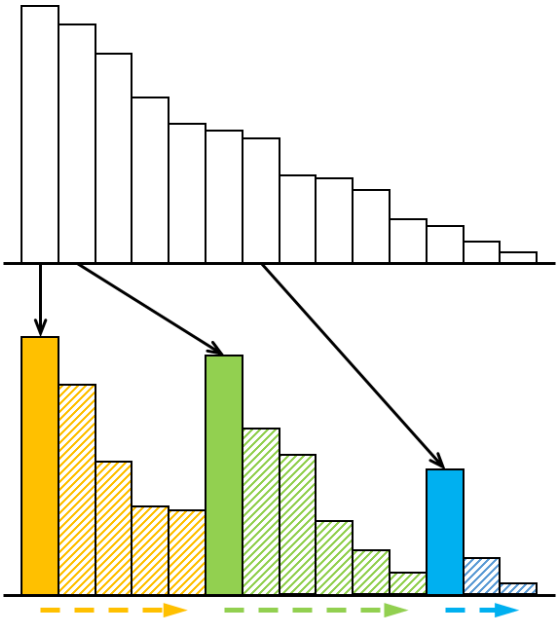

Figure 16: Building a dendogram from a sorted list of regions with three cluster centers

4 WEKA

WEKA is an open source free Java application that enjoys widespread acceptance in both academia and business with an active community. It offers a graphical user interfaces and contains a collection of visualization tools as well as algorithms for data analysis via several data mining tasks like classification, clustering, regression and others. WEKA's different algorithms are located in the package manager, which can be used to extend WEKA with new data analysis algorithms. In addition to offering the BANG-file clustering system in our own Java standalone application, the clustering system is also being released as a WEKA package. This way we can contribute the clustering method as an extension to WEKA and hopefully have the WEKA community take advantage of it.

4.1 Introduction

WEKA, short for *The Waikato Environment for Knowledge Analysis*, was developed in New Zealand at the University of Waikato in hopes of creating a unified workbench allowing researchers easy access to state-of-the-art techniques in machine learning. Allowing comparative studies with different learning algorithms on data sets, WEKA served as a toolbox. Furthermore, it served as a framework for researchers to implement new algorithms while having a comprehensive infrastructure for data manipulation and scheme evaluation already in place. Being open source software, it allowed the creation of many projects that incorporated or extended WEKA which resulted in a widespread acceptance [10]. Being written in Java, it runs on almost any platform and is being distributed under the terms of the GNU General Public License, allowing free use for private or commercial purposes [22].

4.2 WEKA Workbench

The aim of the WEKA project is to provide a collection of learning algorithms and data preprocessing tools to quickly switch between different methods and compare their results when applied on data sets. Applying different learners also allows better assessment of their performance for predictions. Users can preprocess a data set, feed it into a learning scheme and analyze the resulting classifier without the need to write any code. A modular and extensible architecture allows data mining processes to be designed with the provided tools.

The *WEKA Workbench* includes methods for solving common data

mining tasks such as regression, classification, clustering, association rule mining or attribute selection. Most methods provide tunable parameters through a property sheet. Data visualization facilities and data preprocessing tools, called *filters*, allow preliminary exploration of data, which can be combined with statistical evaluation of learning schemes to support models of data mining. These learned models can be used to generate predictions on new instances of the data [22].

4.3 User Interfaces

WEKA offers four graphical user interfaces to chose from in addition to a command line interface to access functionality as seen in figure 17, with the *Workbench* being a unified graphical user interface that combines the other three into a single application. The *Workbench* can be configured by the user to specify what applications and plugins may be loaded and used alongside with their settings [22].

Figure 17: The WEKA GUI Chooser interface

The core and most straight forward graphical user interface however is the *Explorer*, a panel-based interface with different panels for different data mining tasks as seen in figure 18. The individual panels are used to perform the following tasks:

- **Preprocess** - Loads data from files, URLs or databases and allows transformations via preprocessing tools called *filters*.
- **Classify** - WEKA's classification and regression algorithms with textual representation and, if available for the algorithm, graphical representation.
- **Cluster** - Clustering algorithms to run on loaded data with basic statistics to evaluate clustering performance.

- **Associate** - Algorithms for association rule mining with the most well-known algorithms in this area.
- **Select attributes** - Variety of algorithms and evaluation criteria for identifying most important attributes of a data set and exploratory data analysis.
- **Visualize** - Color-coded scatter plot matrix with options of selecting individual plots in the matrix and selecting only portions of the data to visualize [10].

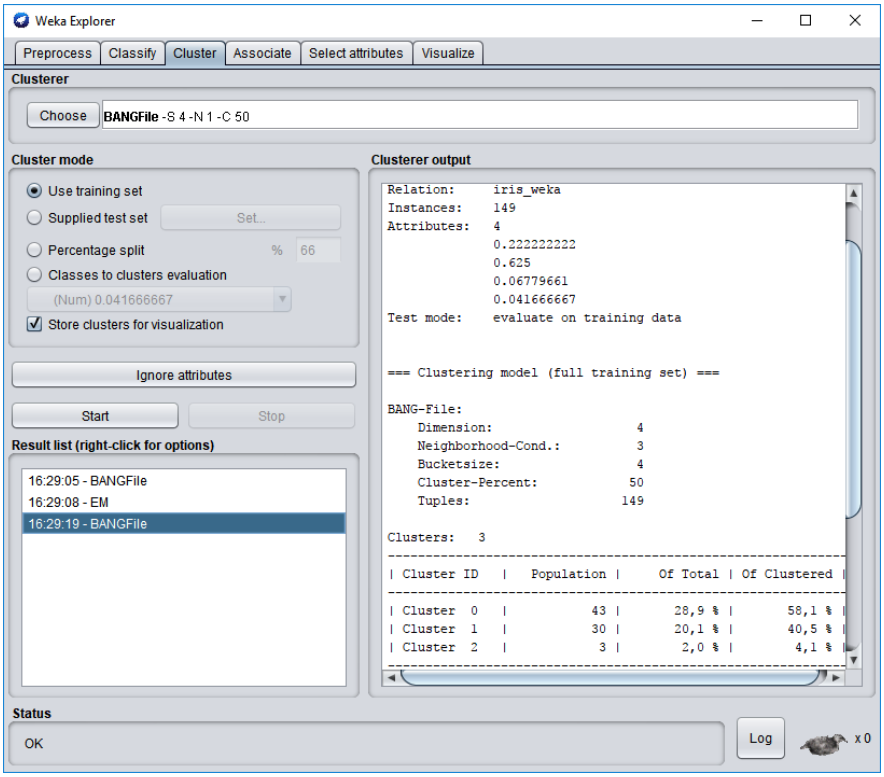

Figure 18: The WEKA Explorer graphical user interface

Another interface is the *Knowledge Flow*, which allows the user to design configurations for processing streamed data. Streaming the data means that unlike the *Explorer*, the *Knowledge Flow* does not have to hold all data in main memory at once. The interface represents the data stream in the form of boxes, representing learning algorithms and data sources that can be joined together in the desired configuration. Components such as

data sources, preprocessing tools, learning algorithms, evaluation methods or visualization modules can be connected to form the data stream.

The *Experimenter* interface is designed to answer basic practical questions when applying classification or regression algorithms. To answer what methods and parameters best work for a given problem the *Experimenter* allows the automation of running classifiers or filters with different parameter settings, while collecting performance statistics and performing significance tests. Furthermore, the *Experimenter* can be set up to distribute the computing load of large-scale statistical experiments across multiple machines via Java remote method invocation [22].

4.4 Memory Management

Most of WEKA's learning algorithms operate on data held in main memory, as such RAM is a limiting factor on the data that can be processed at once. The graphical user interface *Explorer* is designed for batch-based data processing and always loads the complete data set into main memory, which may make handling large data sets impossible.

There are WEKA classifiers and clusterers that can be trained and build in an incremental fashion, one row of data at a time. However, the incremental nature of these algorithms is ignored by the *Explorer* (and also the *Workbench).* To take advantage of incremental handling of data WEKA recommends using their command-line interface instead or even writing the code to build the classifier or clusterer yourself with WEKA's provided API. These methods only require the current data row to be present in main memory and have a runtime that is linear to the number of rows. Clustering methods employing this way of handling data need to implement the *weka.clusterers.UpdateableClusterer* Interface [10].

Additionally, since version 3.7.2, WEKA includes a package that allows use of data streaming learners with Massive On-line Analysis (MOA), a framework to allow massive data stream mining. MOA also includes routines for automatic memory management that can prevent a learned model from exceeding a user-specified maximum memory constraint [9].

4.5 Package Management System

The WEKA software also allows new algorithms and features to be added to the system by the community. Many algorithms and community contributions have been placed into plugin-*packages* that can be selectively installed through the package management system. A WEKA *package* is an archive containing various resources such as source code, compiled

code, documentation and package meta data via property files that is distributed as a ZIP file. The package management system was introduced to make the process of contributing to the WEKA software easier and also ease the maintenance burden on the WEKA development team where only the package metadata has to be tracked.

The package manager displays packages in three lists; installed, available (not yet installed) or all packages. Besides the name, the package list also displays the broad category the algorithm belongs to and the installed or the most recent available version compatible to the used WEKA version. Figure 19 shows information about a package, such as the web location, author, maintainer, license and a brief description. Packages can have dependencies of other packages and every package has at least the minimum version of the WEKA system required as a dependency. If a new *official* package or a new version of an existing one becomes available, the package is appropriately marked in the package list.

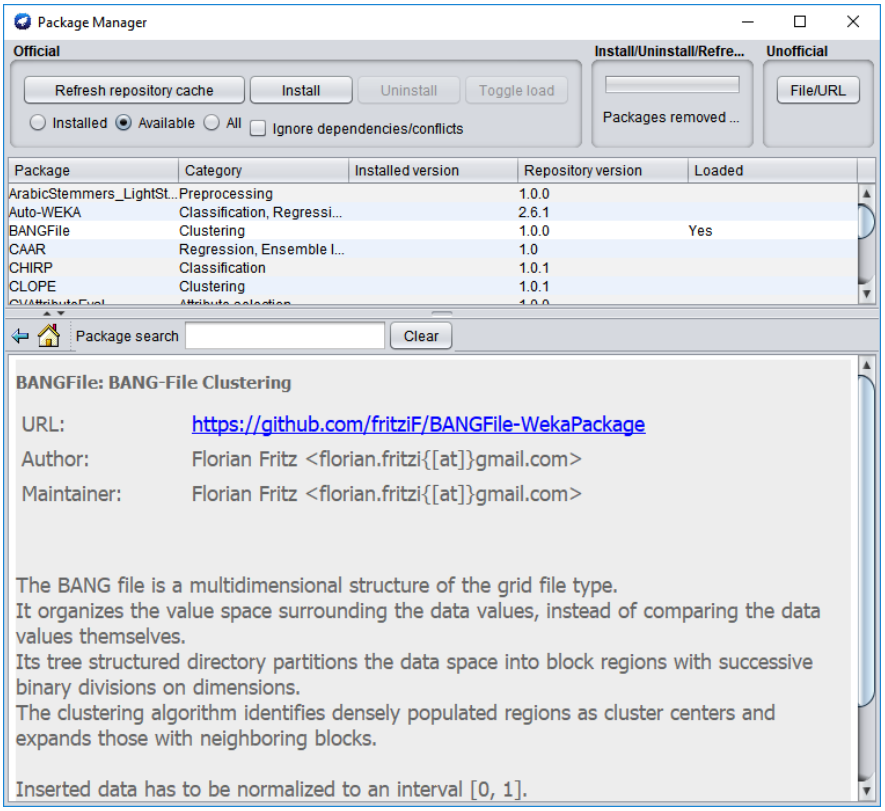

Figure 19: The WEKA Package Manager user interface

The WEKA package manager only lists *official* packages that have been reviewed by the WEKA team and had their metadata added to the official metadata repository. However, packages can also be offered through unofficial capacity. These packages can be installed with the package manager by manually providing the URL or file location to the package and will then be listed in the installed packages list [22].

4.6 Clustering with WEKA

To explain the clustering process using WEKA's tools we will showcase a brief example. For this purpose we will use the core WEKA interface *Weka Explorer* with our our own BANG file clustering method installed as a WEKA *package*.

Data can be loaded from a variety of formats and data sources, but for our example we will be using WEKA's own ARFF file format. As soon as the file has been selected with the **Open file** dialog box the **Preprocess** tab shows information, such as relation, number of instances and number of attributes in the **Current relation** box. We can select an attribute from the **Attributes** list to display additional information about the attribute in the **Selected Attribute** box [22]. This box displays characteristics of the currently selected attribute that may affect preprocessing requirements such as:

- **Name** - Name of the attribute
- **Type** - Type of the attribute, most commonly numeric or nominal
- **Missing** - Number and percentage of instances in data where this attribute is missing

Our desired clustering method requires exclusively numeric data that has been normalized to a $[0, 1]$ interval. In figure 20 we can see that the attributes are numeric, but have not been normalized yet. To do so we apply a WEKA *filter* to transform the data. Clicking the **Choose** button in the **Filter** box we are able to select the `Normalize` filter. Afterwards, by clicking in the textfield next to the **Choose** button we can set the filter's properties in the so called **GenericObjectEditor** dialog. For our example we want to set the option `ignoreClass` to true to normalize all attributes (if set to false the last attribute is considered the class attribute and is not normalized). Once set we can click the **Apply** button and transform the data that has been loaded into memory. In figure 21 we now see that the minimum and maximum values of our attributes are respectively 0 and 1.

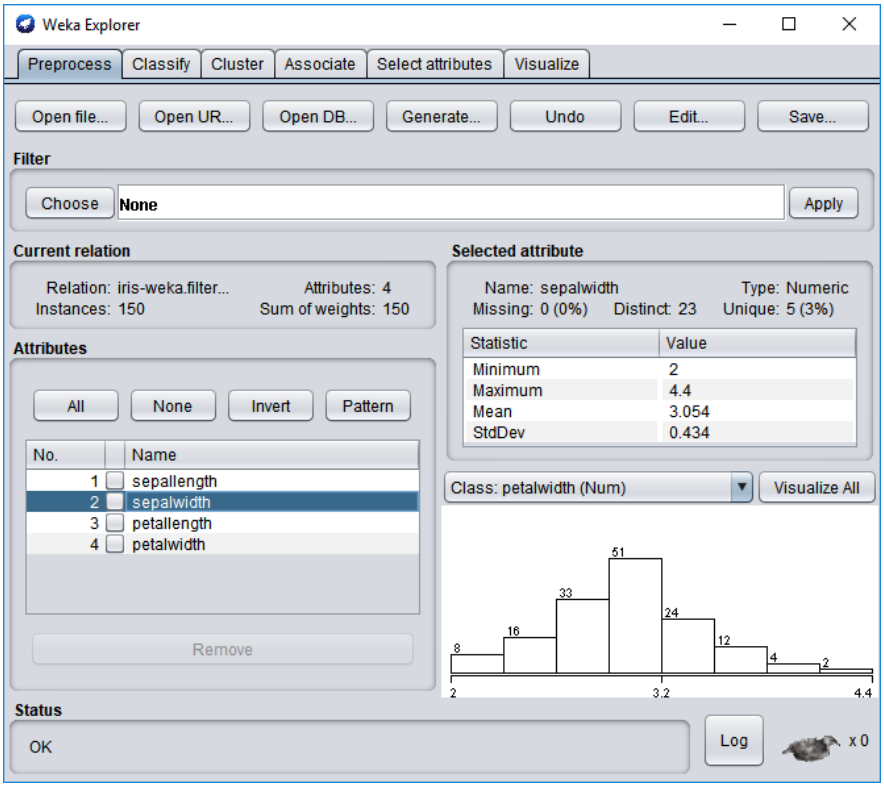

Figure 20: Preprocessing with *Weka Explorer*

In the **Cluster** tab, which can be seen in figure 18, we can select a clustering method similarly to how we selected a filter, by clicking the **Choose** button in the **Clusterer** box. As previously mentioned, we will select our own `BANGFile` clustering method. The **Cluster mode** box provides options that determine what to cluster and how to evaluate the results. Additionally, if certain attributes should be exempt from the clustering we can do so by specifying them with the **Ignore attributes** button. To build the clustering model, we click **Start** and once finished, are presented with information about the data set and the clustering result in the **Clusterer output** textbox. Right clicking the new entry in the **Result list** brings up a menu with the option **Visualize cluster assignments** to visualize the result for further analysis.

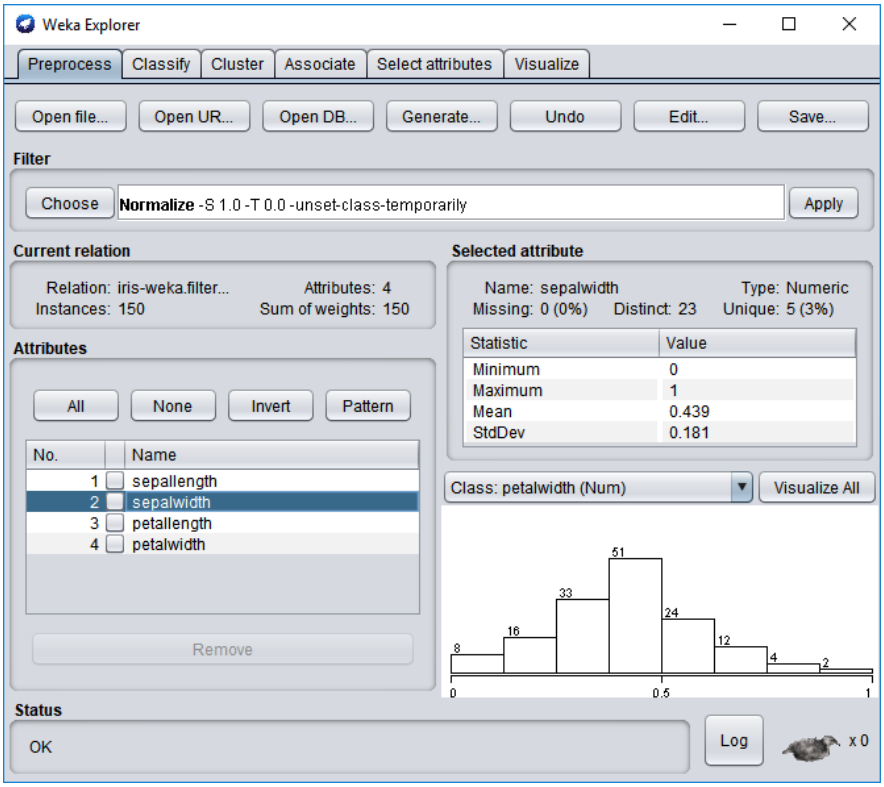

Figure 21: Applying the **Normalize** filter

5 Software Documentation

With this thesis we have also developed the foundation for a flexible software solution that can serve as a framework to analyze multidimensional data sets with various grid based clustering methods, such as the BANG file. The programming language of choice to develop the standalone application was Java. It offers both a command line interface as well as a graphical user interface.

To help analyzing the clustering result, the graphical user interface offers visualizations for the BANG file clustering method. In addition to the dendogram, which shows the density of regions, a two dimensional representation of the final BANG file directory is presented to the user.

A Data Access Object (DAO) layer has been implemented to deal with multidimensional data coming from different data sources. Currently however, only files in the Comma Seperated Value (CSV) format are supported as a data source. In the future the DAO layer can be extended with support for additional types of data sources.

In addition to releasing the BANG file clustering method with our Java standalone application, the clustering system is also being released as a WEKA *package*. Contributing the clustering method as an extension to WEKA will allow the large WEKA community to take advantage of it.

5.1 BANG File Implementation

The BANG file grid structure and its clustering model are managed by three classes inside the `bangfile` package. The UML class diagram in figure 22 shows the `bangfile` package with interactions and references between the classes inside of it. Note that for readability not all operations and attributes are included in the diagram. Additionally, the diagram also includes the abstract class `Clusterer` and the class `ClustererFactory`. The abstract class is used by the application to access operations of the various `Clusterer` implementations. For the `bangfile` package, the class implementing the abstract class is simply called `BANGFile`. The `ClustererFactory` is the very common *Factory* pattern, responsible for creating an object of a class that implemented the `Clusterer` abstract class. This design pattern is further explained in section 5.1.3.

5.1.1 BANGFile

The `BANGFile` class extends the abstract class `Clusterer` and therefor offers all operations required by the application to build a clustering model. Consequently, this class' purposes can be summarized into three parts:

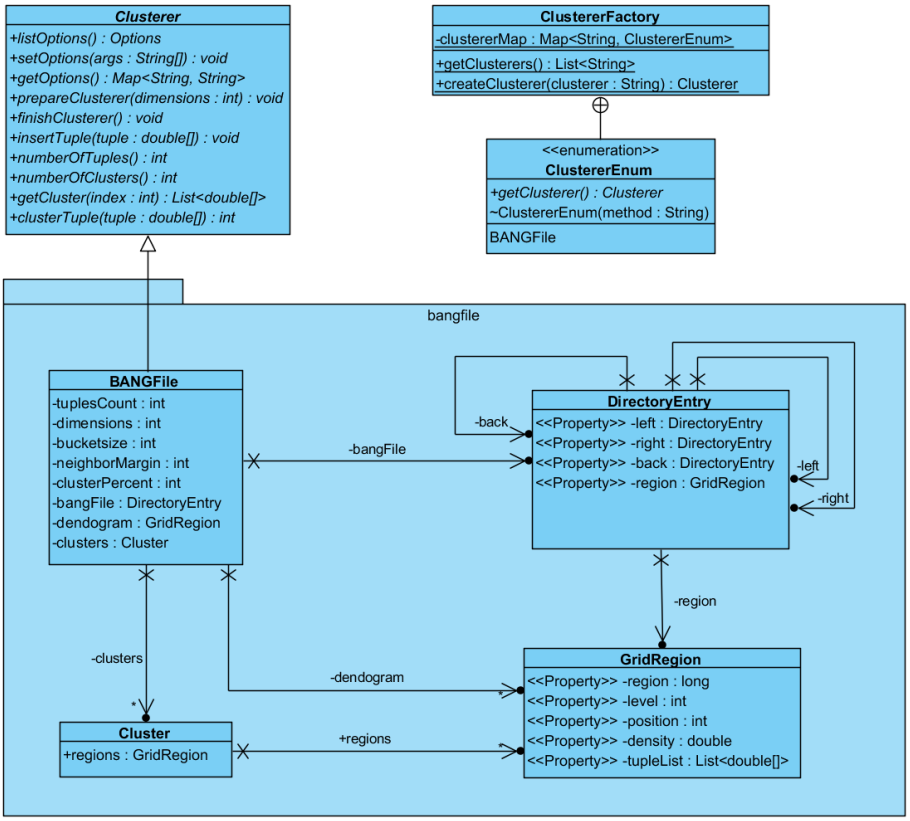

Figure 22: Class Diagram (reduced) of the `bangfile` package

- Handling the parametrization of the BANG file clustering model
- Inserting data into the grid directory and managing the directories structure and balance
- Building clusters from the final BANG file structure

The BANG file clustering model offers three parameters to impact the clustering process and result:

bucketsize: Determines the max population inside a single data bucket. Smaller buckets may yield more accurate results for the cost of performance, depending on the size of the data set.

neighborhood-margin: Determines the margin of touching dimensions required to acquiesce neighborhood between regions. The strictest possible value is 1, meaning a *degree of neighborhood* of $d - 1$ is required.

cluster-percent: Percentage of data and regions to consider when expanding a cluster with regions around initial cluster centers.

Besides handling the parametrization of the BANG file clustering model, this class manages and balances the actual BANG file directory structure. Whenever a pattern is being inserted into the BANG file, the correct grid region within a directory has to be determined. To accomplish this, a single reference to the root `DirectoryEntry` object is stored from which the directory is continuously traversed through. Necessary tasks to keep an effective and nicely balanced directory, such as *split* or *redistribution* operations, are performed if necessary. When all patterns have been inserted, the *dendogram* is filled with regions to ultimately determine clusters with their associated regions.

5.1.2 DirectoryEntry

Objects of the `DirectoryEntry` class represent single entries within the BANG file grid directory. By keeping references to preceding as well as succeeding directory entries it allows the traversal of the entire grid directory in both directions beginning from any entry. In addition to references to other directory entries, the directory entry may also keep a reference to a `GridRegion`, provided the directory entry does not have two directly succeeding entries, turning it into an empty region. The two succeeding entries are referred to as the *left* and the *right* region due to the binary partitioning strategy. Following various operations such as a *split* or a *redistribution* operations, the directory entry ensures that all references are properly updated and that patterns are correctly moved between their `GridRegion` objects. Furthermore, this class calculates both the size and the density of a logical region. To correctly determine the size and density of a logical region, sizes of succeeding regions have to be assessed and subtracted in the calculations.

5.1.3 GridRegion

The `GridRegion` class encapsulates, as the name would suggest, all properties of a single region within the BANG file. It is responsible for storing

information such as region-number, level, population and density, as well as storing the actual patterns that are inserted into the clustering model.

5.2 Application Architecture

To get an overview of the architecture of our standalone clustering application we will first take a look at the project file and hierarchy shown in figure 23.

```
BANGFile
├── src
│   ├── at.ac.univie.clustering
│   │   ├── cli ...  Command Line Interface.
│   │   │   └── CliMain.java
│   │   ├── clusterers ...  Clusterer interface and
│   │   │                   implementations.
│   │   │   ├── bangfile ...  Package of BANG file
│   │   │   │                 implementation.
│   │   │   ├── Clusterer.java
│   │   │   └── ClustererFactory.java
│   │   ├── dataWorkers ...  DAO interface and
│   │   │                    implementations.
│   │   │   ├── CsvWorker.java
│   │   │   ├── DataWorker.java
│   │   │   └── DataWorkerFactory.java
│   │   ├── gui ...  Controllers of JavaFX2
│   │   │           GUI.
│   │   │   ├── dialogs ...  Controllers of dialogs.
│   │   │   ├── Controller.java
│   │   │   └── Main.java
│   └── resources
│       ├── gui ...  Views of JavaFX2 GUI.
│       │   ├── dialogs ...  Views of dialogs.
│       │   └── Main.fxml
├── test ...  JUnit4 Unit Tests.
└── pom.xml ...  Maven Project POM file.
```

Figure 23: Standalone application file hierarchy

For our project we used **Maven** as a project management tool. The `p`

`om.xml` file (POM stands for *Project Object Model*) is the representation of a Maven project, containing all information about the projects build life-cycle, plugins and dependencies to be used during the build process. Most projects have dependencies on other projects to be built and run. Managing dependencies however, can be a complicated task depending on the amount of dependencies. Maven simplifies dependency management with its dependency list in the POM file, which automatically downloads and links dependencies during compilation [17].

With Maven the building process and the deployment of the project can be controlled through a small set of commands. The most important commands for a Maven project are [16]:

clean - remove all files generated by the previous build.
install - install the package into the local Maven repository, for use as a dependency in other projects locally.
test - run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
package - take the compiled code and package it in its distributable format, such as a JAR.

The command line and the graphical user interface are respectively located in the `cli` and `gui` packages of the project. The architectural pattern of *model-view-controller* (MVC) was followed while coding the graphical interface to achieve a more structured and maintainable code. Using **JavaFX2** for our graphical user interface every scene and dialog are made up of a JavaFX FXML *view* file and an associated JavaFX *controller* class. A JavaFX FXML file is an XML file that provides the structure of the user interface separated from the application logic code [1]. The controller class handles the `DataWorker` or `Clusterer` *models* while also accessing and modifying the interface defined in the associated FXML file by tagging objects in the interface with the `@FXML` annotation.

Contained within the `clusterers` package are the `Clusterer` interface, the `ClustererFactory` class and various implementations of clustering methods. Similarly, the `dataWorkers` package contains various implementations of Data Access Objects (DAO) as well as the `DataWorker` interface and the `DataWorkerFactory` class.

The `test` directory contains **JUnit4** software unit tests of implemented clustering methods and DAO to verify that their logic is correct. Running these tests automatically helps identifying software regressions when changes are introduced. Needless to say, higher test coverage results in less manual testing required when developing new features.

5.3 Developer Guide

As previously mentioned, the standalone application was designed with the intention of being extended with additional grid based clustering methods or support for additional data sources in the future. In this section we will explore how a new clustering method can be introduced to the application.

5.3.1 Creating a new Clustering Method

The abstract class `Clusterer` was written to serve as an interface between the application and all future grid based clustering methods. The functions offered by the interface aim to cover the parametrization and preparation of the clustering method, as well as the insertion of patterns and the creation of the clustering model.

For a new clustering method a new package within the `at.ac.univie.clustering.clusterers` package should be created to encapsulate the entire code and all classes of the clustering method. Additionally, it is required for the clustering method to extend the already mentioned abstract class `Clusterer` that is also inside the `at.ac.univie.clustering.clusterers` package as seen in listing 1.

Listing 1: Starting new clustering method

```
1 package at.ac.univie.clustering.clusterers.bangfile;
2
3 import at.ac.univie.clustering.clusterers.Clusterer;
4
5 public class BANGFile extends Clusterer {
6     ...
7 }
```

5.3.2 Parametrization

For parametrization, the **Apache Commons CLI** library was used, which provides an API for parsing command line options passed to programs. While this library is mainly used for command line interfaces, the parameter parsing of the library is used for both the command line interface and the graphical user interface. Consequently, when implementing parameters for a clustering method they are available in both user interfaces without additional code being required.

Three functions need to be implemented to list available options, parse provided options or get currently set options. The `listOptions` function provides meta-information about available options of the clustering method. To be returned is an `Options` object that is essentially a list of `Option` objects. Each `Option` should be made up of a short-option, a long-option, a boolean on whether the option requires an argument or not and a short description as seen in listing 2. Note that in the graphical user interface, only the long-option is displayed in the settings dialog and not the short-option.

Listing 2: Listing available options

```
1  import org.apache.commons.cli.Options;
2  ...
3
4  @Override
5  public Options listOptions(){
6      Options options = new Options();
7      options.addOption("s",       //short option
8          "bucketsize",            //long option
9          true,                    //has argument?
10         "The max ... ");         //description
11     options.addOption("n",
12         "neighborhood",
13         true,
14         "Provided neighborhood ... ");
15     options.addOption("c",
16         "cluster-percent",
17         true,
18         "Percentage of data ... ");
19
20     return options;
21 }
```

The `setOptions` function is responsible for parsing arguments provided to the application and assigning values to corresponding variables of the clustering method. In addition to that, the method is supposed to perform checks on whether the provided argument is a legal value. If an illegal option or argument value is provided, the function throws a `ParseException` with a brief error message as seen in listing 3.

Listing 3: Parse and set provided options

```
1  import org.apache.commons.cli.CommandLine;
2  import org.apache.commons.cli.CommandLineParser;
3  ...
4
5  @Override
6  public void setOptions(String[] args) throws
       ParseException {
7      CommandLineParser parser = new DefaultParser();
8      CommandLine cmdLine = parser.parse(listOptions(),
           args);
9
10     if (cmdLine.hasOption("s")){
11         int s = Integer.parseInt(cmdLine.
               getOptionValue("s", "10"));
12         if (s < 4) {
13             throw new ParseException("Bucketsize may
                   not be lower than 4");
14         }
15         this.bucketsize = s;
16     }
17
18     if (cmdLine.hasOption("n")){
19         int n = Integer.parseInt(cmdLine.
               getOptionValue("n", "1"));
20         if (n < 1){
21             throw new ParseException("Neighborhood-
                   Margin may not be smaller than 1");
22         }
23         this.neighborMargin = n;
24     }
25     ...
26 }
```

Finally, the `getOptions` function simply returns the currently set clustering method options with their assigned value as a `Map` variable as shown in listing 4. It is important for the key of the option in our map to be identical to the option's long-option as defined in listing 2. This allows the application to present the user with the currently selected options in the settings dialog.

Listing 4: Listing available options

```
1 @Override
2 public Map<String, String> getOptions() {
3     Map<String, String> options = new HashMap<String,
          String>();
4     options.put("bucketsize", "" + this.bucketsize);
5     options.put("neighborhood", "" + this.
          neighborMargin);
6     options.put("cluster-percent", "" + this.
          clusterPercent);
7     return options;
8 }
```

5.3.3 Clustering Method Lifecycle

For the implementation of the functions offered by the abstract class it is helpful to understand the *lifecycle* of the clustering method within the application and the order in which the application calls its functions:

1. **ClustererFactory.createClusterer(String clusteringMethod)** - Create object of clustering method (**constructor**)
2. **listOptions()** - Retrieve available options of clustering method
3. **prepareClusterer(int dimensions)** - Provide clustering method with number of dimensions for potential pre-processing
4. **setOptions(String[] args)** - Apply provided options to clustering method
5. **insertTuple(double[] tuple)** - Insert patterns into clustering model
6. **finishClusterer()** - Perform required post-processing of clustering method
7. **toString()** - Present string representation of clustering model to user
8. **numberOfClusters()** - Retrieve number of clusters
9. **getCluster(int index)** - Get all patterns of a single cluster

5.3.4 Clusterer Factory Design Pattern

In order to create the object of the desired clustering method the *Factory Design Pattern* has been implemented, which is categorized as a creational design pattern. A design pattern describes how to solve common recurring design problems during software development to build flexible and reusable object-oriented software [7].

The principle behind the *Factory Design Pattern* is to get an object, in our case that of a clustering method, at runtime based on a parameter we passed. This is being done with the actual implementation of the clustering method effectively staying behind closed doors. At runtime we simply pass the name of the desired clustering method to the `ClustererFactory`, which returns the reference of a newly created instance of the respective class. This way, the application does not need to concern itself with different implementations of clustering methods.

The foundation of the `ClustererFactory` is the `ClustererEnum` enumeration, where every entry has a `getClusterer()` method returning their respective implementation of the `Clusterer` abstract class. A new clustering method needs to be added to this enumeration, as shown with a `NewClusterer` example in listing 5.

Listing 5: Clusterer Enumeration

```
1  private enum ClustererEnum {
2      /**
3       * BANGFile enum entry.
4       */
5      BANGFile("BANGFile"){
6          @Override
7          public Clusterer getClusterer(){
8              return new BANGFile();
9          }
10     },
11     /**
12      * New Clusterer enum entry.
13      */
14     NewClusterer("NewClusterer"){
15         @Override
16         public Clusterer getClusterer(){
17             return new NewClusterer();
18         }
19     };
```

```
20
21      /* clustering method name */
22      private String method;
23
24      /**
25       * Used to return new Clusterer object.
26       *
27       * @return  Clusterer object
28       */
29      public abstract Clusterer getClusterer();
30
31      /**
32       * Provide a method name to entries in the enum.
33       *
34       * @param method    clustering method name
35       */
36      ClustererEnum(String method){
37          this.method = method;
38      }
39  }
```

Additionally, a `Map` variable was used to maintain a list of all clustering methods and their corresponding entry in the enumeration. The name of the clustering method should be used as a key in the map. Adding our **NewClusterer** example to this map, as seen in listing 6, ensures that the application offers it as a selectable clustering method with the chosen key as its name [14].

Listing 6: Map containing enum entries

```
1  /* Map to retrieve objects via method name */
2  private static Map<String, ClustererEnum>
       clustererMap = new HashMap<String, ClustererEnum
       >();
3
4  /**
5   * Class initialization method. New clustering
       methods need to be added to the map.
6   */
7  static {
8       clustererMap.put("BANGFile", ClustererEnum.
            BANGFile);
```

```
9       clustererMap.put("NewClusterer", ClustererEnum.
            NewClusterer);
10  }
```

5.3.5 Abstract Class Clusterer

Abstract class for managing and building a clustering model. Inserting tuples is done in incremental fashion.
The method `prepareClusterer` is called before data is inserted. Afterwards, the method `finishClusterer` will be called.

5.3.5.1 Declaration

public interface Clusterer

5.3.5.2 Constructor Summary

Clusterer() Initialize Clusterer Object.

5.3.5.3 Method Summary

clusterTuple(double[]) Predicts the cluster membership for a provided instance.
finishClusterer() Performs required post-processing of the clustering model after the data has been inserted and produces clusters filled with tuples.
getCluster(int) Return all tuples contained within a specific cluster.
getOptions() Lists options with their currently assigned value.

insertTuple(double[]) Insert tuple into the clustering model.

listOptions() Build Options object used to to list and display available options of clustering method.
numberOfClusters() Return number of clusters available in clustering model.
numberOfTuples() Return number of total tuples inserted into clustering model.
prepareClusterer(int) Generates and resets the clusterer.
setOptions(String[]) Parse provided arguments and set options of clustering method.

5.3.5.4 Methods

- **clusterTuple**

```
public abstract int clusterTuple(double[] tuple)
```

  - **Description**

    Predicts the cluster membership for a provided instance.

  - **Parameters**

    * `tuple` – tuple to be classified

  - **Returns** – index of cluster

- **finishClusterer**

```
public abstract void finishClusterer()
```

  - **Description**

    Performs required post-processing of the clustering model after the data has been inserted and produces clusters filled with tuples.

- **getCluster**

```
public abstract java.util.List getCluster(int
    index) throws java.lang.
    IndexOutOfBoundsException
```

  - **Description**

    Return all tuples contained within a specific cluster.

  - **Parameters**

    * `index` – index of cluster in cluster-list

  - **Returns** – tuples contained within cluster

  - **Throws**

    * `java.lang.IndexOutOfBoundsException` – If index not in cluster-list

- **getOptions**

```
public abstract java.util.Map getOptions()
```

  - **Description**

    Lists options with their currently assigned value.

  - **Returns** – currently set clustering method options

- **insertTuple**

```
public abstract void insertTuple(double[] tuple)
```

  - **Description**

    Insert tuple into the clustering model.

  - **Parameters**

    * `tuple` – tuple of the dataset to be inserted

- **listOptions**

```
public abstract Options listOptions()
```

  - **Description**

    Build Options object used to to list and display available options of clustering method.

  - **Returns** – available clustering method options

- **numberOfClusters**

```
public abstract int numberOfClusters()
```

  - **Description**

    Return number of clusters available in clustering model.

  - **Returns** – number of clusters in clustering model

- **numberOfTuples**

```
public abstract int numberOfTuples()
```

  - **Description**

    Return number of total tuples inserted into clustering model.

- **Returns** – number of tuples in clustering model

- **prepareClusterer**

```
public abstract void prepareClusterer(int
    dimensions) throws java.lang.Exception
```

  - **Description**

    Generates and resets the clusterer.
    Perform setup that may need to happen before inserting data.
    Initialize all variables of the clusterer that were not set with options.

  - **Parameters**

    * `dimensions` – dimensions of dataset

  - **Throws**

    * `java.lang.Exception` – If clusterer setup is not possible

- **setOptions**

```
public abstract void setOptions(java.lang.String
    [] args) throws ParseException
```

  - **Description**

    Parse provided arguments and set options of clustering method.

  - **Parameters**

    * `args` – arguments provided by user

  - **Throws**

    * `ParseException` – If invalid option or illegal value provided

5.4 WEKA Package

To extend WEKA with our BANG file clustering method as a package we first have to take a look at how clustering methods are executed within WEKA. WEKA allows both the implementation of batch trainable and incrementally trainable clustering methods [22]. For our clustering method however, we chose to only implement batch trainable methods. The largest factor for this decision was the ability of the BANG file

WEKA package to automatically normalize the data, which requires the entire data set to be present at once.

Training by batch-insertion of data happens in two stages:

1. **setOptions(String [])** - Set cluster-specific options.
2. **buildClusterer(Instances)** - Provides the entire data set as an Object and builds the clustering model. Subsequent calls of this method need to reset the model and ultimately result in the same model again.

The incremental insertion of data starts with the same two stages as the batch-insertion to initialize the model. When calling the `buildClusterer(Instances)` method, one might use an empty or an initial set of the data for initialization. Following that, incremental clustering happens in two additional stages:

3. **updateClusterer(Instance)** - Updates the model row-by-row.
4. **updateFinished()** - Performs required post-processing of the clustering model after the model has been updated with the new data.

In order for our clustering method to be visible in WEKA it must implement the `Clusterer` interface from the package `weka.clusterers`. To make the implementation easier however, WEKA offers the abstract class `AbstractClusterer`, which implements many of the interface's functions.

The `AbstractClusterer` additionally implements other WEKA interfaces, some of which have proven useful or even necessary for the implementation of our BANG file clustering method:

java.io.Serializable - WEKA uses serialization to store the clustering model with all inserted data. This requires all classes associated with the clustering model to be serializable.

weka.core.CapabilitiesHandler - To provide meta-information on what type of data can be handled. For our clustering method we need to specify that only numeric attributes and no missing values are allowed.

weka.core.OptionHandler - Allows listing available options and to provide options to the clustering method.

weka.core.CommandlineRunnable - Allows the method to be run from WEKA's command line interface.

weka.core.TechnicalInformationHandler - Return paper references and publications this clustering method is based on.

WEKA packages are distributed as a ZIP archive. To automatically build the package we used the **Apache Ant** build processes that were already included in a template project offered by WEKA. The anatomy of a WEKA package requires at least a JAR file with compiled java classes `BANGFile.jar` and a property file `Description.props` to exist. Additional resources such as documentation or source code are not mandated, but are encouraged. Our *BANGFile* ZIP contains, besides the required JAR and property files, the generated Javadoc documentation and the entire source code of both the clustering method and associated **JUnit** unit-tests. When the ZIP archive is unpacked it creates the directory structure

```
BANGFile.zip
├── doc ...  Generated Javadoc
│            documentation.
├── src
│   ├── main
│   │   └── java
│   │       ├── at/ac/univie/clustering/clusterers/bangfile
│   │       │   ├── BANGFile.java
│   │       │   ├── DirectoryEntry.java
│   │       │   └── GridRegion.java
│   │       └── weka
│   │           └── clusterers
│   │               └── BANGFile.java
│   └── test
│       └── java
│           ├── at/ac/univie/clustering/clusterers/bangfile
│           │   ├── BANGFileTest.java
│           │   ├── DirectoryEntryTest.java
│           │   └── GridRegionTest.java
│           └── weka
│               └── clusterers
│                   └── BANGFileTest.java
├── BANGFile.jar
└── Description.props
```

Figure 24: WEKA package file hierarchy

shown in figure 24.

When installed with WEKA's package manager, a directory named *BANGFile* will be created in `$WEKA_HOME/packages` holding the contents of the package. The default location for `$WEKA_HOME` is `user.home/wekafiles`, where `user.home` is the user's home directory [22].

5.5 User Guide

5.5.1 Installation

The project for the standalone application, including the BANG file algorithm, is located in the **GitHub** online repository `https://github.com/fritziF/BANGFile`. The prerequisites for building the project are:

- Apache Maven 3.x
- Java Development Kit (JDK) 1.8

To download the project one might choose to **Download ZIP** from the **GitHub**-website or to clone the **Git** repository into a local repository. Most modern integrated development environment (IDE) have a **Git** plugin that enables cloning a repository directly into the IDE. Alternatively, a user may choose to clone the repository via a terminal with the command in listing 7.

Listing 7: Git command to clone repository

```
1 $ git clone https://github.com/fritziF/BANGFile.git
```

The application uses **Maven** as a build automation tool. With the cloned and downloaded **Git** repository, the application can quickly be built by using Maven commands. Listing 8 contains the go-to command for building and installing a package into a local repository. This also automatically downloads all dependencies and performs all unit tests of the package. Note that prior to the building process, all generated files from previous builds are removed.

Listing 8: Maven command to install in local repository

```
1 $ mvn clean install
```

This will compile and package the code into a JAR file **BANGFileClustering-x.x.x.jar** in the **target** directory with all dependencies present. Additionally, the package will be installed into the local maven repository for use as a dependency in other projects.

5.5.2 User Interface

When launching the application the user is greeted with a rather barren interface, as seen in figure 25, having most buttons disabled. Through the menu the user has the menu-item **Clusterer** to select a clustering method and the menu-item **File - Select File...** to open a file dialog.

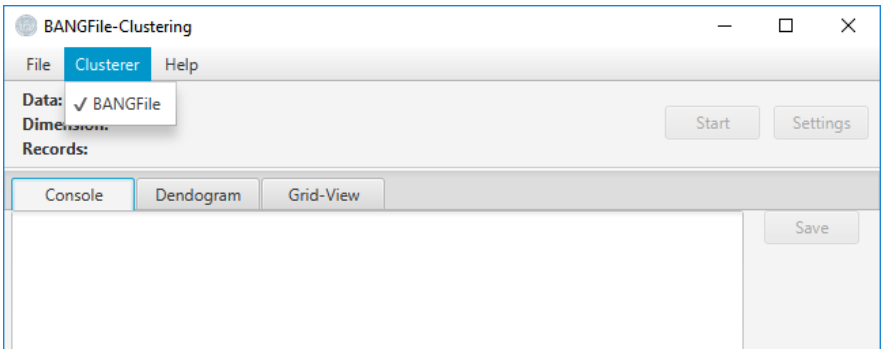

Figure 25: Main scene with menu to select clustering methods

By clicking **File - Select File...**, the dialog of figure 26 opens with a **browse** button and three settings for the format of the CSV file. The **browse** button opens a *file chooser* GUI of the respective operating system for navigating the file system and choosing the desired CSV file containing the data set. As a starting directory the *file chooser* GUI will always point to the users home directory, which depends on the operating system's settings and of course the operating system itself. Once a file has been selected, we can change the settings in the dialog to match the format of the CSV file:

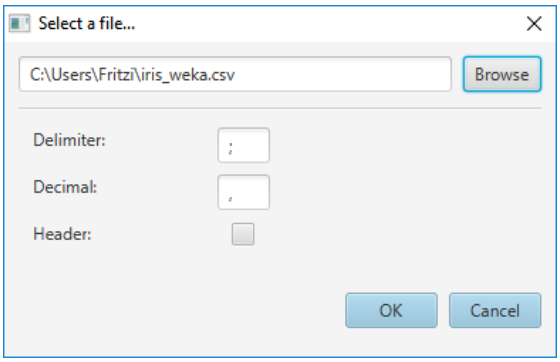

Figure 26: **Select File** dialog

- **Delimiter** - Delimiter symbol used to separate values in a row of the file.
- **Decimal** - The decimal symbol used for the numbers in the data set.
- **Header** - If this *checkbox* has been checked, the first line is skipped.

As soon as the file selection is confirmed with the **OK** button, provided no error occurred while reading the file, the buttons **Start** and **Settings** will become enabled. Additionally, some meta-information is now shown in the interface, such as the filename and the number of dimensions and records of the data set. An example with meta-information is shown in figure 27.

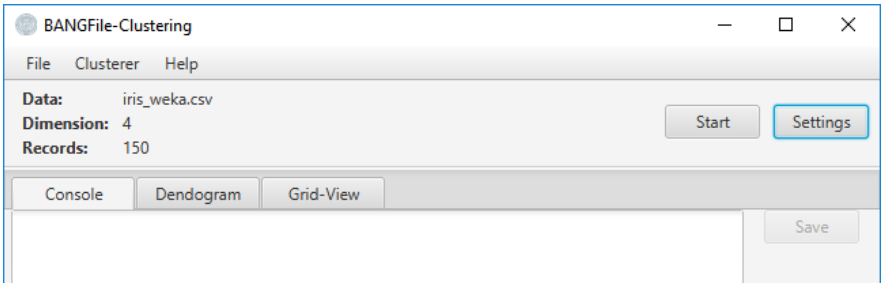

Figure 27: Main scene with file selected

After clicking the **Settings** button, we are presented with the individual options of the selected clustering method and their default values. In the case of the **BANGFile** clustering method, the options shown in figure 28 are available.

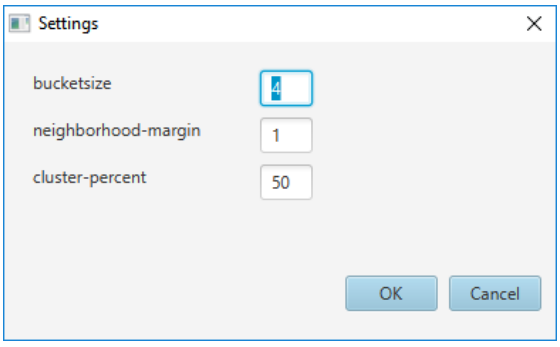

Figure 28: **Settings** dialog of clustering method **BANGFile**

With the data set selected and the parameters of the clustering method set the actual clustering method can now be executed. This is being done by clicking the **Start** button. As soon as the clustering model has finished, the result is presented within a text-area in the **Console** tab as seen in figure 29.

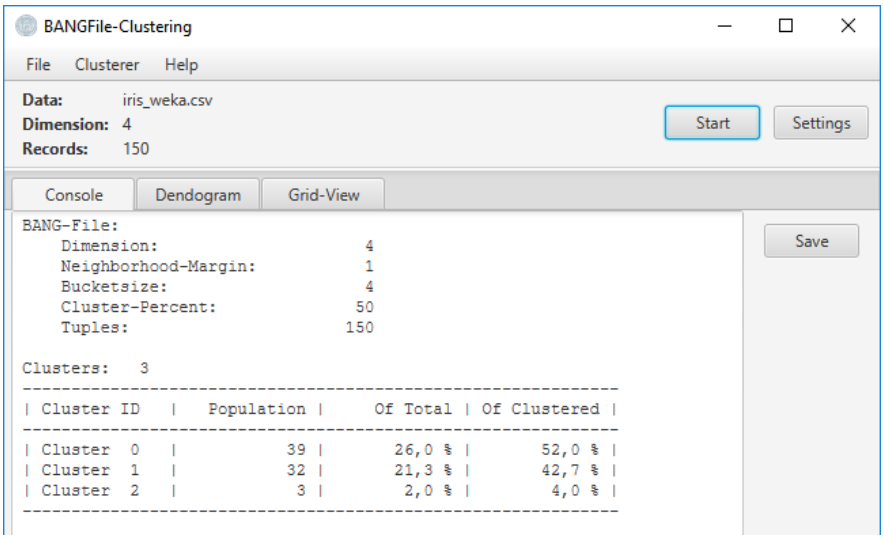

Figure 29: **Console** showing clustering results

The user may save the clustering result by clicking the **Save** button, which opens up a dialog that allows the user to choose a directory-location and a format for the resulting CSV files. These options can be seen in figure 30. Confirming the options with the **OK** button displays the dialog in figure 31 containing information on the files stored to the chosen directory.

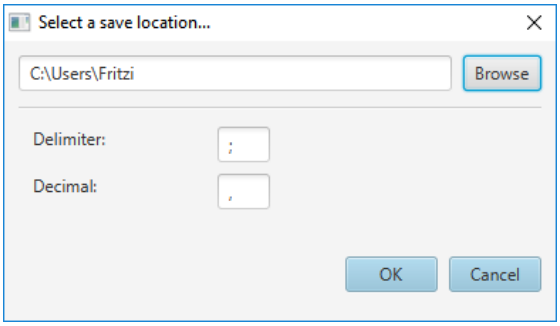

Figure 30: **Save** dialog to store clustering result

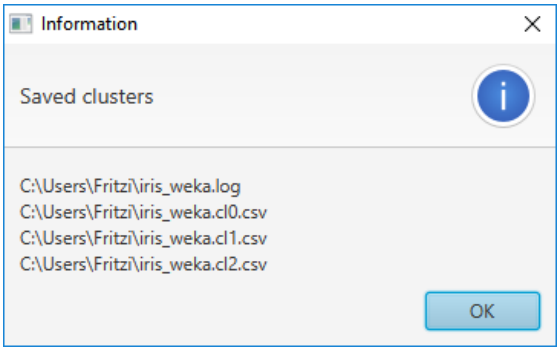

Figure 31: **Information** dialog about saved clustering result

6 Experiment

For this experiment we will use **WEKA 3.8.1** with **Java 8** to compare the BANG file clustering method to some known clustering algorithms. WEKA allows us to easily apply different clustering algorithms on the same data sets. Additionally, if the data set contains a class-attribute that signals the supposed cluster assignment, we can tell WEKA to evaluate the result of the clustering model based on correctly clustered patterns.

We will use the command line interface *Simple CLI* of WEKA for the experiment. The commands for the preprocessing of the data set and execution of the clustering algorithm can be seen in listing 9. This will reduce the additional processing or memory required by the application itself. Unlike the graphical user interface, the command line interface does not store the results of executions in memory, further reducing the impact on successive executions.

WEKA only displays the actual time taken to build the clustering model in a graphical user interface like the *Explorer*, but strangely not the *Simple CLI* interface. Because of this, the source code of WEKA had to be modified to allow the *Simple CLI* interface to print information about the time taken to build the model.

The option `force-batch-training` will be provided to load the entire data set into memory prior to execution, as such the times presented are pure processing times for the completion of the clustering algorithms. As Java by default allows allocation of only a fourth of heap memory from the systems physical memory, we will increase the so called `MaxHeapSize` of Java to 12 GB for processing of larger data sets.

Listing 9: WEKA CLI preprocessing and clustering

```
1 # Convert file from CSV format to WEKA's ARFF format
2 $ java weka.core.converters.CSVLoader input.csv -H -F
    ";" -L "3:Cluster1,Cluster2" > input.arff
3 # Normalize data to a [0, 1] interval with the '
    Normalize' filter
4 $ java weka.filters.unsupervised.attribute.Normalize
    -S 1.0 -T 0.0 -decimal 7 -i input.arff -o
    input_normalized.arff
5 # Running our installed clustering algorithm '
    BANGFile'
6 $ java weka.Run BANGFile -S 150 -t input_normalized.
    arff -c 2 -force-batch-training
7 # Running a pre-installed clustering algorithm
```

```
    included in WEKA
8 $ java weka.clusterers.FarthestFirst -N 3 -S 1 -t
    input_normalized.arff -c 2 -force-batch-training -
    S 150
```

We compared the BANG file clustering method to the well known conventional clustering algorithms *k-means* and *Canopy*. The used **WEKA** packages are called **SimpleKMeans**, **Canopy** and of course our **BANG-File**. For the BANG file we used a dynamic block size of 5% of the data set size. Sizes of the data sets ranged from 150 thousand to 30 million patterns. Presented times are an average of at least three executions.

6.1 Data Sets

For this experiment, we will use synthetic data sets with the supposed cluster assignments already known a priori. Using the cluster assignment of every pattern, we can evaluate the clustering model created by the BANG file clustering algorithm. To obtain data sets with sufficient data and complexity we used the data set generator included in **ELKI**. ELKI is an open source data mining software written in Java with a focus on unsupervised methods in cluster analysis. The generator can be configured with a XML specification file and supports uniform, normal and gamma distributions to generate non-overlapping clusters of specified size, rotation and scaling [2].

Figure 32 shows a two-dimensional data set created using a normal distribution that was generated with the XML specification file seen in listing 10. The data set contains three clusters with each cluster containing 5000 patterns and an additional 500 patterns as *noise* spread across the entire data space. In the corresponding BANG file grid structure in figure 33 we can see the partitioning algorithm adapting to the data distribution.

Listing 10: ELKI data set XML specification file

```
1 <dataset random-seed="3">
2   <!-- 3 Clusters with Normal distribution in both
      dimensions -->
3   <cluster name="Cluster1" size="5000">
4     <normal mean="0.3" stddev="0.09" />
5     <normal mean="0.3" stddev="0.09" />
6   </cluster>
7   <cluster name="Cluster2" size="5000">
8     <normal mean="0.85" stddev="0.05" />
```

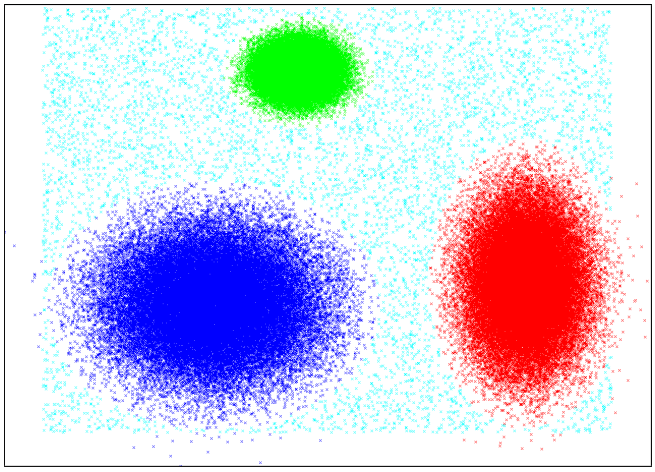

Figure 32: Two-dimensional data set generated with the ELKI's data set generator

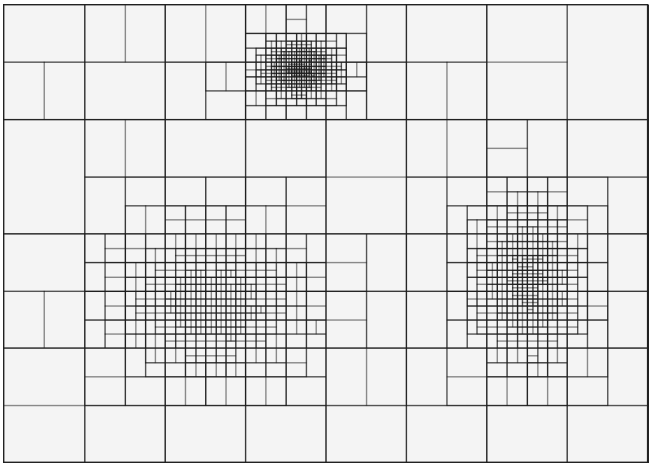

Figure 33: BANG file grid structure of figure 32

```
9       <normal mean="0.35" stddev="0.10" />
10    </cluster>
11    <cluster name="Cluster3" size="5000">
12      <normal mean="0.45" stddev="0.035" />
13      <normal mean="0.85" stddev="0.035" />
14    </cluster>
15    <!-- "Noise" with Uniform distribution across
          entire data space -->
```

```
16    <cluster name="Noise" size="500" density-correction
          ="1.0">
17      <uniform min="0" max="1" />
18      <uniform min="0" max="1" />
19    </cluster>
20  </dataset>
```

The data sets generated for comparing different clustering methods contain either 2, 3 or 10 dimensions, with each data set containing three clusters and additional patterns spread across the entire data space as *noise*.

6.2 Results

With data sets of up to 1.5 million patterns, the BANG file clustering method was marginally slower than the conventional clustering algorithms. However, with the data sets consisting of 15 and 30 million patterns, **BANGFile** managed to outperform the competition. To cluster the data set with 2 dimensions and 30 million patterns, **BANGFile** returned a result in less than halve the time compared to **Canopy** or **SimpleKMeans** (see figure 34). For the data set with 3 dimensions and 30 million patterns, **BANGFile** took only about 55 seconds, while **SimpleKMeans** took 72 and **Canopy** took 88 seconds (see figure 35).

The time difference increased when switching to the data set with 10 dimensions and 30 million patterns, where **BANGFile** managed to severely outperform its peers. **BANGFile** clustered the data in about 124 seconds and was more than three times faster than **Canopy**, which took about 439 seconds (see figure 36). Moreover, **SimpleKMeans** was not able to cluster the massive data set and the **WEKA** application ended up killing the clustering process on multiple attempts.

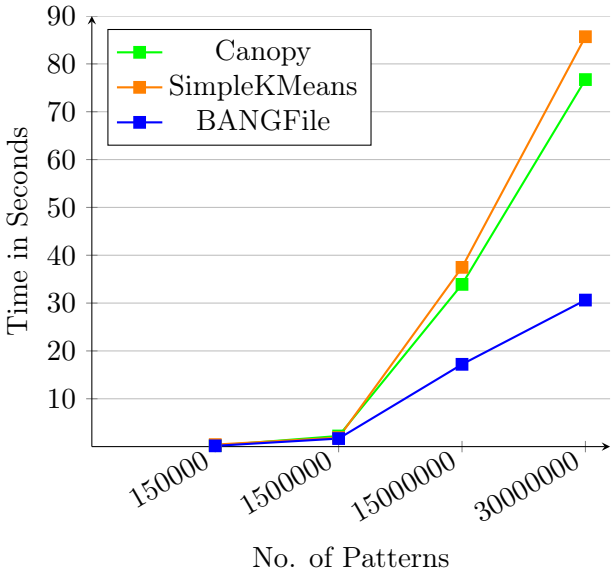

Figure 34: Processing times with 2-dimensional data sets

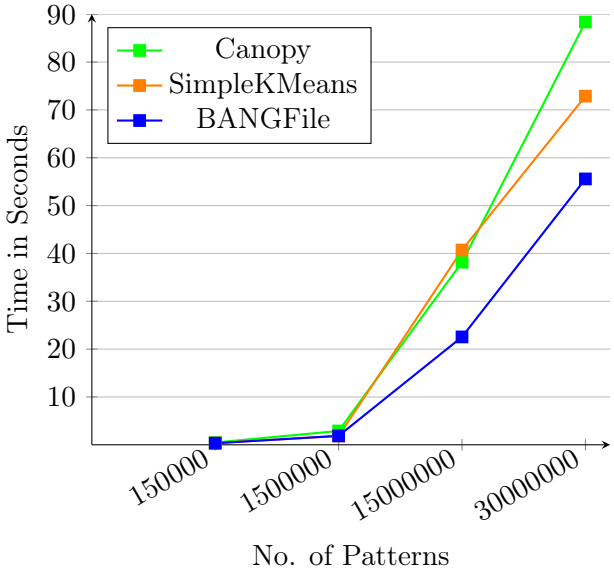

Figure 35: Processing times with 3-dimensional data sets

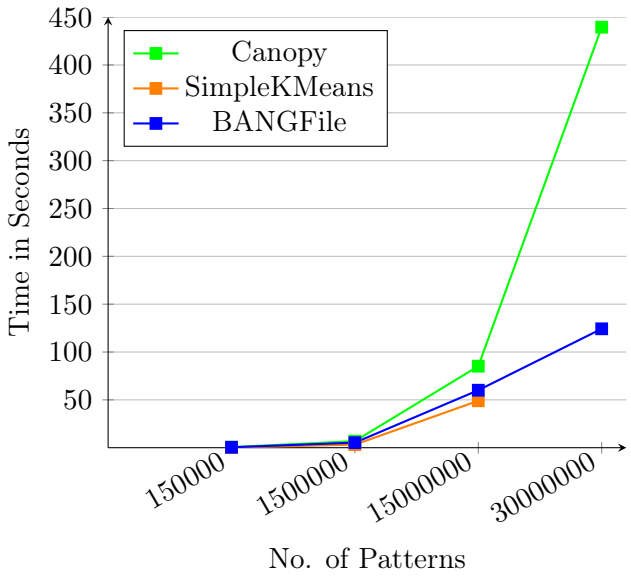

Figure 36: Processing times with 10-dimensional data sets

7 Conclusion

We presented the hierarchical clustering algorithm *BANG file clustering*, which is an extension of the grid file clustering algorithm presented in [18]. The algorithm manages to cluster massive multi-dimensional data sets in a fast and timely manner compared to conventional clustering methods.

To showcase the clustering algorithm a standalone Java application has been developed offering a graphical interface to prepare, configure and execute it on data sets. The application's architecture has also been designed with the intention of being extended with additional grid based clustering methods or support for additional data sources in the future.

Additionally, the clustering algorithm was released as an *official* WEKA package. After a review by the WEKA team the package was added to WEKA's central package repository to have the WEKA community take advantage of it.

References

[1] Gail Chappell and Nancy Hildebrandt. *Oracle JavaFX Documentation Home - Getting Started with JavaFX*. 2013. URL: https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm (visited on 01/27/2018).

[2] *ELKI Data Set Generator*. URL: https://elki-project.github.io/datasets/generator (visited on 02/18/2018).

[3] Martin Erhart. "Entwurf und Implementation eines BANG-File-basierten Clusteranalyseverfahrens". MA thesis. University of Vienna, 1995.

[4] Brian S. Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis*. 4th. Wiley Publishing, 2009. ISBN: 0340761199, 9780340761199.

[5] Michael Freeston. "Data Structures for Knowledge Bases: Multi-Dimensional File Organisations". In: *ECRC, Technical Report TR-KB-13* (1986).

[6] Michael Freeston. "The BANG File: A New Kind of Grid File". In: *SIGMOD Rec.* 16.3 (Dec. 1987), pp. 260–269. ISSN: 0163-5808. DOI: 10.1145/38714.38743. URL: http://doi.acm.org/10.1145/38714.38743.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[8] G. Gan, C. Ma, and J. Wu. *Data Clustering: Theory, Algorithms, and Applications*. Society for Industrial and Applied Mathematics, 2007. DOI: 10.1137/1.9780898718348. eprint: http://epubs.siam.org/doi/pdf/10.1137/1.9780898718348. URL: http://epubs.siam.org/doi/abs/10.1137/1.9780898718348.

[9] Mark Hall. *Pentaho Community Wiki Handling Large Data Sets with Weka*. 2016. URL: https://wiki.pentaho.com/display/DATAMINING/Handling+Large+Data+Sets+with+Weka (visited on 10/12/2017).

[10] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. "The WEKA Data Mining Software: An Update". In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278. URL: http://doi.acm.org/10.1145/1656274.1656278.

[11] Martin Haseneyer. “Grid Files, Eine dynamische Datenstruktur mit mehrdimensionalen Zugriffspfaden”. Friedrich-Schiller-Universität Jena, 2005.

[12] A. K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13-022278-X.

[13] A. K. Jain, M. N. Murty, and P. J. Flynn. “Data Clustering: A Review”. In: *ACM Comput. Surv.* 31.3 (Sept. 1999), pp. 264–323. ISSN: 0360-0300. DOI: 10.1145/331499.331504. URL: http://doi.acm.org/10.1145/331499.331504.

[14] Debadatta Mishra. *DZone Factory Design Pattern - An Effective Approach*. 2012. URL: https://dzone.com/articles/factory-design-pattern (visited on 01/29/2018).

[15] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. “The Grid File: An Adaptable, Symmetric Multikey File Structure”. In: *ACM Trans. Database Syst.* 9.1 (Mar. 1984), pp. 38–71. ISSN: 0362-5915. DOI: 10.1145/348.318586. URL: http://doi.acm.org/10.1145/348.318586.

[16] Brett Porter. *Apache Maven Project Introduction to the Build Lifecycle*. 2015. URL: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html (visited on 01/27/2018).

[17] Eric Redmond and Karl Heinz Marbaise. *Apache Maven Project POM Reference*. 2016. URL: https://maven.apache.org/pom.html (visited on 01/27/2018).

[18] Erich Schikuta. “Grid-clustering: an efficient hierarchical clustering method for very large data sets”. In: *Proceedings of 13th International Conference on Pattern Recognition*. Vol. 2. Aug. 1996, 101–105 vol.2. DOI: 10.1109/ICPR.1996.546732.

[19] Erich Schikuta and Martin Erhart. “BANG-Clustering: A novel grid-clustering algorithm for huge data sets”. In: *Advances in Pattern Recognition: Joint IAPR International Workshops SSPR’98 and SPR’98 Sydney, Australia, August 11–13, 1998 Proceedings*. Ed. by Adnan Amin, Dov Dori, Pavel Pudil, and Herbert Freeman. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 867–874. ISBN: 978-3-540-68526-5. DOI: 10.1007/BFb0033313. URL: https://doi.org/10.1007/BFb0033313.

[20] Erich Schikuta and Martin Erhart. "The BANG-clustering system: Grid-based data analysis". In: *Advances in Intelligent Data Analysis Reasoning about Data: Second International Symposium, IDA-97 London, UK, August 4–6, 1997 Proceedings*. Ed. by Xiaohui Liu, Paul Cohen, and Michael Berthold. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 513–524. ISBN: 978-3-540-69520-2. DOI: 10.1007/BFb0052867. URL: https://doi.org/10.1007/BFb0052867.

[21] Erich Schikuta and Florian Fritz. "An Execution Framework for Grid-clustering Methods". In: *Procedia Comput. Sci.* 80.C (June 2016), pp. 2322–2326. ISSN: 1877-0509. DOI: 10.1016/j.procs.2016.05.430. URL: https://doi.org/10.1016/j.procs.2016.05.430.

[22] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128042915, 9780128042915.

List of Figures

1 Clusters of data points in a two-dimensional value space . 9
2 Stages of a clustering process 10
3 Tree of classification types 12
4 2-dimensional grid structure 15
5 Splitting operations with bucket size of 2 16
6 Inserting a pattern into a grid region at capacity 18
7 Three *tied* blocks with $D_B = 6$ 21
8 Numbering scheme (*region number*, *level*) for grid regions 24
9 Regions partitioning a non-uniform data distribution . . . 24
10 Grid region R2 nested within grid region R2 28
11 Creation of buddy regions 30
12 Binary tree storing the directory of a BANG file 32
13 2-dimensional value space example 34
14 2-dimensional BANG file structure of figure 13 34
15 Neighborhood - Grid directory of tree in figure 12 36
16 Building a dendogram from a sorted list of regions with three cluster centers 38
17 The WEKA GUI Chooser interface 40
18 The WEKA Explorer graphical user interface 41
19 The WEKA Package Manager user interface 43
20 Preprocessing with *Weka Explorer* 45
21 Applying the **Normalize** filter 46
22 Class Diagram (reduced) of the `bangfile` package 48
23 Standalone application file hierarchy 50
24 WEKA package file hierarchy 63
25 Main scene with menu to select clustering methods 65
26 **Select File** dialog 65
27 Main scene with file selected 66
28 **Settings** dialog of clustering method **BANGFile** 66
29 **Console** showing clustering results 67
30 **Save** dialog to store clustering result 68
31 **Information** dialog about saved clustering result 68
32 Two-dimensional data set generated with the ELKI's data set generator 71
33 BANG file grid structure of figure 32 71
34 Processing times with 2-dimensional data sets 73
35 Processing times with 3-dimensional data sets 73
36 Processing times with 10-dimensional data sets 74

Listings

1 Starting new clustering method 52
2 Listing available options 53
3 Parse and set provided options 53
4 Listing available options 54
5 Clusterer Enumeration 56
6 Map containing enum entries 57
7 Git command to clone repository 64
8 Maven command to install in local repository 64
9 WEKA CLI preprocessing and clustering 69
10 ELKI data set XML specification file 70

Abstract

Cluster analysis is essential in the field known as explorative data analysis. The *Balanced And Nested Grid* (BANG) file is a hierarchical clustering system of the grid file type. To efficiently cluster massive data sets the BANG file uses a multidimensional grid structure to organize the value space surrounding pattern values. Its tree structured directory partitions the value space into regions with successive binary divisions on dimensions, which results in self-balancing features of a B-tree. Consequently, unlike previous grid file designs, the directory expands proportionally to the data regardless of the data distribution. The partitioning strategy accurately reflects the clustering of patterns in the value space, with densely populated regions identified as cluster centers, and adapts to changes in the distribution. This thesis concludes with a demonstration of the BANG file clustering system both as a standalone Java application as well as a WEKA package.

Zusammenfassung

Die Clusteranalyse spielt eine zentrale Rolle in der explorativen Datenanalyse. Das *Balanced And Nested Grid* (BANG) File ist ein hierarchisches Clustering-Verfahren des Typs Grid-File. Um riesige Datenmengen effektiv zu clustern bildet das BANG File eine mehrdimensionale Raster-Struktur, welche Daten gruppiert. Der Datenraum wird dabei durch kontinuierliches zweiteilen, orthogonal zu einer der Dimensionen, in Regionen gegliedert. Diese Regionen werden in einem Verzeichnis indexiert, welches die Eigenschaften eines balancierten Baums bietet. Anders als bisherige Grid-File Designs wächst dieses Verzeichnis dadurch proportional zu den Daten, unabhängig von der Verteilung der Daten. Die hierarchische Partitionierungsstrategie spiegelt Cluster im Datenraum wider, wobei Regionen mit den höchsten Dichten an Daten als Zentrum eines Clusters identifiziert werden. Diese Arbeit demonstriert das BANG File Clustering-Verfahren innerhalb einer eigenständigen Java Anwendung und auch als ein WEKA Plugin.