Some Basic Functions for Tree Representations of Bayesian Markov Chain Monte Carlo Clustering

Prof. in Charge: Prof. Anthony Davison[1]
Supervisor: Vahid Partovi Nia[2]
Author : Arpit Chaudhary[3]

Abstract

The paper contains description of the implementation of C code for tree representation of Markov Chain Monte Carlo(MCMC) clustering. The aim of the code is to produce results which helps in visual representation of the most frequent pattern, its agglomerations and divisions and thus helps in constructing a tree also called as *dendrogram*. It illustrates the mergers or divisions which have been made at successive levels. The paper explains codes for various code functions used for finding the most frequent patterns, its agglomerations and divisions from the labels generated by Markov Chain Monte Carlo Simulations. They have been implemented in C and attached to R statistical software. The algorithmic analysis of code and running time analysis has been done with the help of certain test datasets in later section of the report. The following section of the paper contains description of the C files, the explanation of the function implemented in them and its linkage to R so that C function can be called from R for speeding up the execution of the code.

[1]IMA Chair of Statistics, Ecole Polytechnique Federal De Lausanne, Switzerland, anthony.davison@epfl.ch

[2]Project Supervisor, Department of Statistic, EPFL, Switzerland. vahid.partovinia@epfl.ch.

[3]Student, Fourth Year, Department of Mathematics and Computing, IIT Delhi, New Delhi. arpit0642004@gmail.com. Internship Report from May 15, 2007 to July 1, 2007.

Contents

1 Introduction

The model consists of an implementation of an approach to clustering based on spike and slab model for higher dimension and uses the Markov Chain Monte Carlo(MCMC) simulations for producing labels to group the individuals of the dataset. The methodology for generating labels for each individual have not been discussed in the following paper. The aim is to find the most frequent pattern in the labels generated by MCMC simulations, its subsequent agglomerations and divisions so that the data can be represented visually by constructing a tree. The visualization of MCMC result by constructing a tree is generally not easily possible in R. The constructed tree can be useful in analysing the most frequent patterns and its subsequent agglomerations and divisions by cutting the tree at different levels. Applied scientists are used to see a tree corresponds to possible clustering which is not easy to assess in MCMC clustering; it is not implemented in well-known statistical softwares. It is also possible to visualize and obtain pattern which are not most frequent and are neither an agglomeration nor division of the most frequent pattern. This may be of high consequence if its frequency is comparable to the most frequent pattern as our data may also support structures which can not really be written as agglomeration or division of the most frequent pattern.

The following section of this paper discusses the way and techniques used in finally arriving at the visual representation of the labels(patterns) with the help of the tree. We start with the labels generated by MCMC simulations as an input and then explain in detail various methods, functions and their utility with the help of codes written in C and its interface with R. Next, the codes are explained in a sequence in which it is processing the input data. The following sequence of operation in general can also be applied to any model for visualizing and analyzing the patterns by constructing a tree. It has high demand in R nowadays to visualize MCMC clustering outputs in a handy way.

2 A Simple Hierarchical Agglomerative Clustering Example and Pseudocode

Agglomerative clustering is a technique of starting from every individual as a different cluster. It joins the two cluster which has the smallest distance ending up with all individuals in one group. In contrary to agglomerative clustering we may have *divisive clustering* which assumes that initially all the individuals are in one group. It then divides the individuals according to some property until and unless each individual falls into different clusters.

2.1 Pseudocode for hierarchical agglomerative clustering

- **Input :** Set of individuals to be clustered. In our example being considered, there are six individuals { A, B, C, D, E, F}. A matrix of distances between every pair of examples.
- Initially, create one cluster for each individual example
- WHILE (Number of Clusters is greater than 1)
  - Find the nearest pairs of clusters (involves measuring the distance between every pair of clusters)
  - Merge the examples in this pair into a single cluster, i.e.: delete the rows and columns in M associated with these two clusters and add to M a new row and column corresponding to the new cluster.
- Output : A dendrogram

2.2 A Simple Agglomerative Clusterng Example and its Corresponding Dendrogram

Consider an example with six individuals { A, B, C, D, E, F}. Initially, each individual is a different cluster and we have a 6 by 6 distance matrix. We can assign a label {1,2,3,4,5,6} which shows that each individual forms a separate cluster. In the following figures blue colour shows the individuals clustered together recently and red shows the individuals grouped together previously.

- In the first iteration, the pair with the smallest distance between them are put in one cluster. Individuals B and C are grouped together to form a cluster. The labels for this step is {1,2,2,3,4,5}. Five different numbers in the labels represent 5 clusters. This is illustrated in Fig. 1.

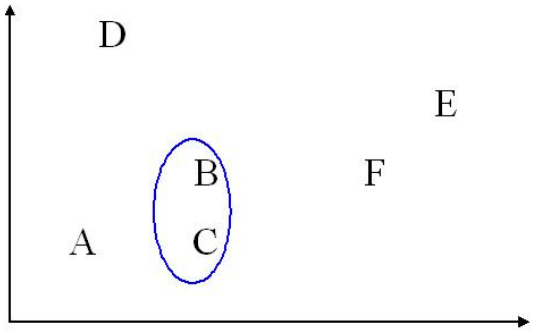

Figure 1: An agglomerative clustering example at 1st level of iteration

- In the second iteration, individuals E and F are grouped together to form another cluster. The labels assigned are {1,2,2,3,4,4} showing the clusters (A), (B,C), (D) and (E,F) respectively. In total there are four clusters. This is illustrated in Fig. 2.

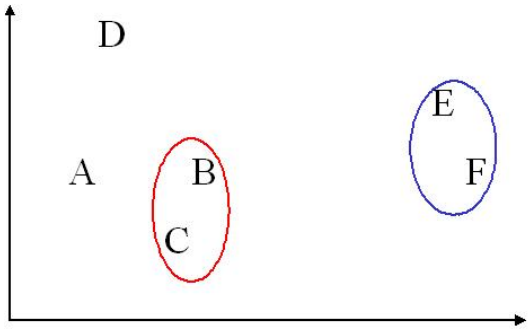

Figure 2: An agglomerative clustering example at 2nd level of iteration

- In the third iteration, individuals A and the group (B,C) are grouped together to form a new cluster along with group (E,F) and D as separate clusters. The labels assigned are {1,1,1,2,3,3} showing the clusters (B,A,C), (E,F) and (D) respectively. This is illustrated in Fig. 3.

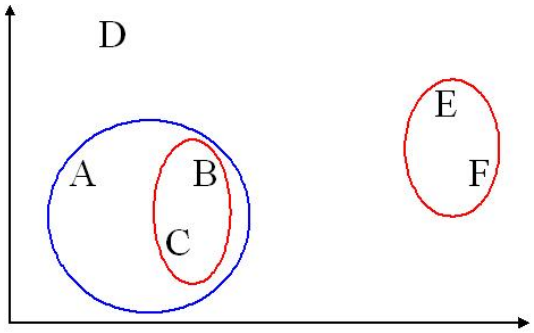

Figure 3: An agglomerative clustering example at 3rd level of iteration

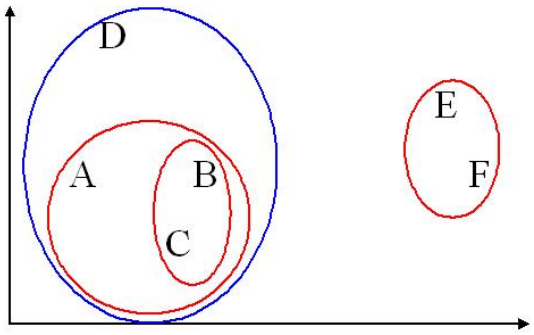

Figure 4: An agglomerative clustering example at 4th level of iteration

- In the fourth iteration, the cluster (A,B,C) is grouped along with (D) in to form a new cluster and cluster (E,F) forms another group. The labels assigned are {1,1,1,1,2,2} showing the individuals A,B,C,D in one group and E,F in another group. In total there are two clusters. This is illustrated in Fig. 4.
- In the last iteration, the only possibility left is to put everything together resulting in single cluster. The next level of agglomeration will lead to {1,1,1,1,1,1} showing all the individuals in the same group. This is illustrated in Fig. 5.

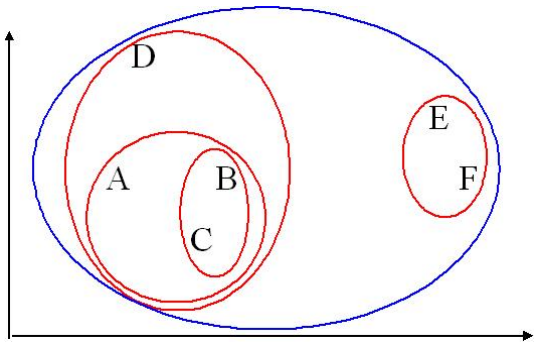

Figure 5: An agglomerative clustering example at 5th level of iteration

The result of the above agglomerative clustering can be represented with the help of a *dendrogram*. A dendrogram is simply a tree representation of the above sequence of iterations. The dendrogram for the above mentioned simple example is as shown in Fig. 6.

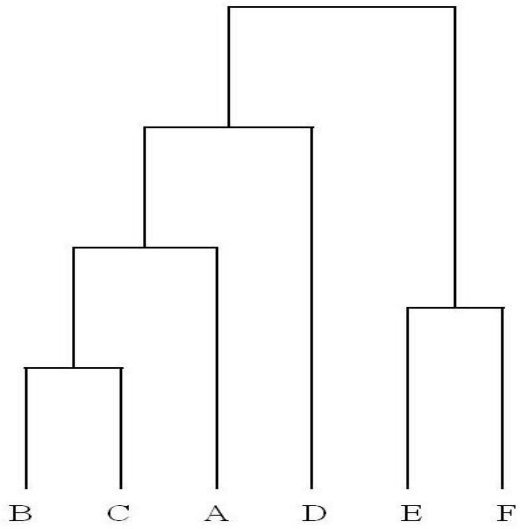

Figure 6: Dendrogram for the above agglomerative clustering example

An actual clustering solution is produced by "pruning" the dendogram, which gives us flexibility to choose the number of clusters. Cutting the dendrogram at just below the root level divides the individuals into 2 clusters {B,A,C,D} and {E,F} represented by red and blue colour respectively in the dendrogram in Fig. 7. The green colour shows the level at which we are cutting the dendrogram.

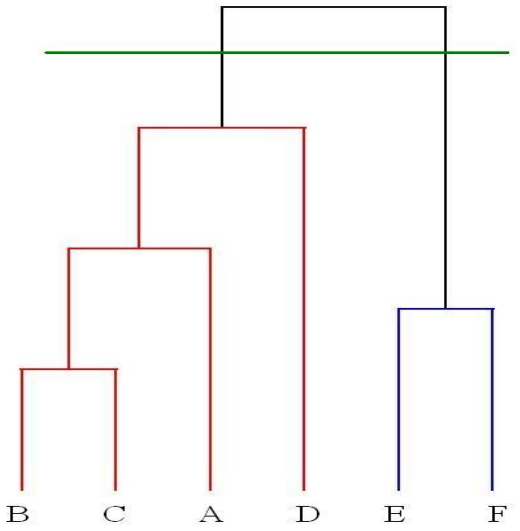

Figure 7: Dendrogram: Cutting just below the root level

Cutting the dendrogram at two levels below the root level divides the individuals into 3 clusters {B,A,C}, {D} and {E,F} represented by red, black and blue colour respectively in Fig 8.

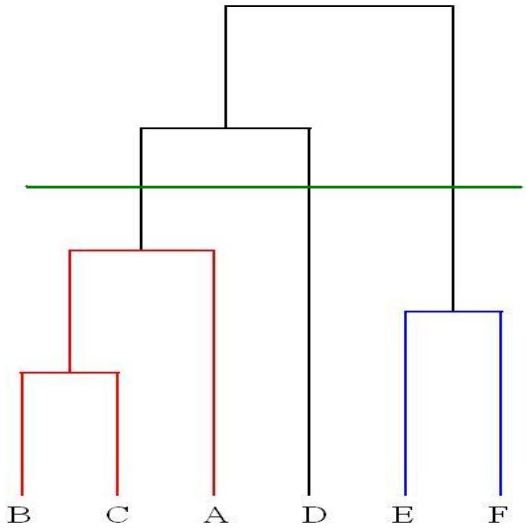

Figure 8: Dendrogram: Cutting two level below the root level

Since constructing dendrograms depends largely on the distance measure we use, changing definition of distance between the two clusters may result in different dendrograms. In contrary to the agglomerative clustering we may construct a dendrogram using divisive clustering technique. In these technique, the decision to merge or split two cluster is greedy i.e. the result of merging or splitting can not be undone at a later stage of constructing the dendrogram. The method also requires to compute the distance between every pair of examples and can be computationally expensive too.

3 Relabeling the Input Data : m_relabel.c

The labels given as input generated by Markov Chain Monte Carlo simulations consists of a fixed number of individuals(which is the same as the Pattern length and is represented as Pat_L in rest of the paper) repeated over a number of times given by the number of iterations(represented by N_Iter). These individuals are grouped together by the labels given as input. Suppose, there are N individuals in the dataset then there may exist $2^N - 1$ different possible labeling of the individual. The aim is to count the frequency of occurrence of each such labels in the dataset such that it represents the same grouping of the individuals. Since, the labels generated may not be unique that is two different labels may correspond to the same grouping of individuals. Consider for example of six individuals(Pattern Length = 6). Suppose two consecutive monte carlo iterations are as follow $\{1, 1, 2, 2, 1, 1\}$ and $\{2, 2, 1, 1, 2, 2\}$. If K is the number of different labels then there exists $K!$ different labels corresponding to the same grouping of the individuals. In our example $K = 2$ so there exist two such labels representing the same grouping of 6 individuals. The two labels are different but they correspond to the same grouping of individuals $1, 2, 5, 6$ in one group and individuals $3, 4$ in another group. It is possible in the second label to change 1 with 2 and vice versa to obtain the same label as in the first one. Since, the input data may consist of several of such labels which will only complicate the procedure of counting the patterns uniquely so we introduce the technique of relabeling the given dataset so that each label occurs only once and it uniquely determines a given classification of individuals with no other labels can produce the same classification. The code for this method is contained in a file called m_relabel.c" which has also been included in first section of the appendix. This C file uniquely relabels the labels generated as MCMC simulations so that no two labels produces the same grouping of individuals as described above. The file has following functions implemented to achieve the desired goal.

3.1 Function compare{}

- **Input :** The input to this function is an array of integers, an integer value, an integer value say `T[], m, L`.
- **Output :** The function returns the position of first occurrence of `m` uptill length `L` in an array `T` as defined above.
- **Function Description :** The function finds an element in an array of type integer up to a certain length in an array and returns the position of first occurrence of the element and returns `-1` if the element does not exist in the function.
- **Function Return Type** int
- **Remark** This subfunction is used in function `relabel{}`

3.2 Function relabel{}

- **Input :** The input to this function is an array of integers and an integer value say `A[], L`.
- **Output :** The function stores the address of the 1st element of the uniquely relabeled array(`N_A` which is local to the function `relabel{}`) in a globaly defined pointer called `P1`.
- **Function Description :** The function uniquely relabels an array `A[]` uptill the specified length `L`(which in general is the Pattern length in our case). It is possible to relabel any label in such a way that its first element is always one followed by an increasing sequence of number for any new number encountered and using the same number for the previous numbers that has already been considered(that is replaced by other number). Since the numbers assigned are in increasing order so it is possible to construct a temporary array `T[]` such that at any general step `i` of relabeling, the function checks the ith value in the temporary array and if it exists it simply replaces the value with one more than the position found by using the compare function else it just adds the new element found in the temporary array for future reference and replaces its value by one more than the position at which it was added into a new array. This continues until the entire array is scanned.
- **Example :** Consider an array `A[2,2,2,1,1,1,2,2,2,4,4,3,1,2]` which contains the label of any general MCMC simulations for 14 individuals. The temporary array for this example is `T[2,1,4,3]`. The Relabeled array is produced by replacing all the 2's by one more than the index at which it occurs in the array `T[]`. So, the uniquely relabeled array is `A[1,1,1,2,2,2,1,1,1,3,3,4,2,1]`.
- **Remark :** This subfunction is used in the function M_Relabel.
- **Function Return Type** void

3.3 Function M_Relabel {}

- **Input :** The input to this function is an array of integers `S[]` (input from the MCMC simulation), an integer value `Pat_L`(which gives us the length of each individual pattern), an integer value `N_Iter`(gives the total number of iterations in the MCMC simulations) .
- **Output :** The output of the function is contained in an integer pointer called result passed into the function as input and it contains the uniquely relabeled patterns as an array of integers.
- **Function Description :** The function uniquely relabels an array `S[]` of size Pat_L*N_Iter. At any general step of M_Relabel the function copies the first elements

uptill length `Pat_L` into a temporary array `temp[]` of size `Pat_L` and then call the relabel function with the parameters `temp` and `Pat_L` to uniquely relabel it and stores the result in an output pointer called result and the process continues until every element of the array S has been processed.

- **Function Return Type** void

3.4 Function RM_Relabel {}

- **Input :** The input to this function is a pointer to an array `S[]` (input from the MCMC simulation), a pointer to an integer value `Pat_L`(which gives us the length of each individual pattern), a pointer to an integer value `N_Iter`(gives the total number of iterations in the MCMC simulations) and a pointer to an array of integers called `*result`.
- **Output :** The output of the function is contained in an integer pointer called `*result` passed into the function as input and it contains the uniquely relabeled patterns as an array of integers.
- **Function Description :** This function is only necessary to call the C function from an R interface and everything passed into this function must be of a type pointer. The values to this function is passed by R which is further passed into C function which after operation returns the result in a new pointer called result.
- **Function Return Type** void

3.5 Analysis of the code

The code is linear in nature and has a time complexity of $O(N)$ where N is the size of the array that is it scans each element of the array only once for relabeling. The input datasets were generated using the random function in R and and it was relabeled to analyze the time. The test cases are as mentioned below.

3.5.1 Test case 1 :

- **Number of Individuals :** 500
- **Number of Iterations :** 1000
- **Time to generate the Data in R :** 6 seconds
- **Time to Relabel :** fraction of a second

3.5.2 Test case 2 :

- **Number of Individuals :** 500
- **Number of Iterations :** 10,000
- **Time to generate the Data in R :** 8 minutes
- **Time to Relabel :** 7 seconds

4 Counting The Frequency of Uniquely Relabeled Patterns : count.c

This segment of the code counts the frequency of the occurrence of each uniquely relabeled pattern provided as an input after the processing by the relabel function on the data generated by MCMC simulations. Instead of searching for all unique patterns($2^N - 1$) of the given pattern length say N, the pattern length is processed once to find the frequency of all such uniquely relabeled patterns. The various functions implemented for counting procedure are described below and the code is included in the second part of the appendix(A2).

4.1 Function compare{}

- **Input :** The input to this function is an array of integers, an array of integer and an integer value say `int A[], int B[], int Pat` .
- **Output :** The function `returns 1` if the two arrays are equal that is all the element of the two array are equal else it `returns 0`.
- **Function Description :** The function compares all the elements of the given two array and it returns 1 if both arrays are equal and returns the value zero if all the elements of the given two arrays are not equal.
- **Function Return Type** int

4.2 Function count {}

- **Input :** The input to this function is an array of integers `S[]` (output of the relabeling function is the array S), an integer value `Pat_L`(which gives us the length of each individual pattern which is same in our case), an integer value `N_Iter`(gives the total number of iterations in the MCMC simulations), two integer pointer called `Pattern and Frequency` respectively.

- **Output :** The output of the function is contained in an integer pointer called `Pattern` which an array of integers and contains all the patterns consecutively and an another array of integers which contains the frequency of the patterns contained in the array `Frequency`.
- **Function Description :** The function counts the frequency of the uniquely relabeled labels in an array `S` of size $Pat\_L * N\_Iter$. There is a two dimensional array called `D[][Pat_L+1]` whose last columns contains the frequency of various patterns in the corresponding rows. At any general step of count the function marks the next `Pat_L` from array `S` to be searched for and checks for the marked label in the Table `D`. If it exists it simply increases the frequency by one otherwise it adds the current label to the table and sets its frequency as one and it proceeds until it searches for all possible patterns in the `array S[]`.
- **Function Return Type** void

4.3 Function Rcount {}

- **Input :** The input to this function is an integer pointer to an array of integers `S[]` (output of the relabeling function is the array S), an integer pointer to `Pat_L`(which gives us the length of each individual pattern which is same in our case), an integer pointer to `N_Iter`(gives the total number of iterations in the MCMC simulations), two integer pointer called `Pattern and Frequency` respectively.
- **Output :** The output of the function is contained in an integer pointer called `Pattern` and `Frequency` displaying the patterns and corresponding frequencies respectively which is passed into the function as input.
- **Function Description :** This function is only necessary to call the C function from an R interface and everything passed into this function must be of a type pointer. The values to this function is passed by R which is further passed into C function which after operation returns the result in a new pointer called `Pattern and Frequency`.
- **Function Return Type** void

4.4 Analysis of the code

The code scans the input array `S` of size say `N` where $N = Pat\_L * N\_Iter$ which is a huge number. If the number of patterns are relatively small then the complexity of the algorithm will be approximately of the order of cN where c is a small positive number. The code has been run for the following two test cases and the results are as shown below.

4.4.1 Test case 1 :

- **Number of Individuals :** 500
- **Number of Iterations :** 1000
- **Time to generate the Data in R :** 6 seconds
- **Time to Relabel :** fraction of a second
- **Time to Count the Frequency :** fraction of a second

4.4.2 Test case 2 :

- **Number of Individuals :** 500
- **Number of Iterations :** 10,000
- **Time to generate the Data in R :** 8 minutes
- **Time to Relabel :** 7 seconds
- **Time to Count the Frequency :** 10 seconds

5 Finding all the Agglomerations and Divisions of the Most Frequent Pattern : AggDiv.c

Since hierarchical clustering techniques proceeds by either a series of successive mergers or a series of successive divisions it is essential to find all levels of most frequent agglomerations and divisions of the most frequent pattern in our dataset. Agglomerative hierarchical methods start with the individual objects. Thus, initially there are as many clusters as objects. The most similar objects are first grouped, and these initial groups are thus merged according to their similarity. Divisive hierarchical methods work in opposite direction. An initial single group of objects is divided into two subgroups such that objects in one subgroup are far from the objects in the other. These subgroups are then further divided into dissimilar subgroups, the process continues until there are as many subgroups as objects, that is until each object forms a group. The following section of the code finds all such agglomerative and divisive levels for the most frequent pattern which helps in constructing the dendrogram and represents the mergers and divisions which has been made at successive levels. Also, the patterns which are neither agglomeration or division of the most frequent pattern are found out as our data may also support them if they have frequency close to the most frequent pattern. The functions and techniques used for identifying them are contained in the code aggdiv.c(Appendix) and are explained below.

5.1 Function Compare{}

- **Input :** The input to this function is an array of integers, an integer value, an integer value say `T[], m, Pat_L`.
- **Output :** The function returns the position of first occurrence of `m` up till length `Pat_L` in an array `T[]` as defined above.
- **Function Description :** The function finds an element in an array of type integer up to a certain length in an array and returns the position of first occurrence of the element and returns -1 if the element does not exist in the function.
- **Remark :** This subfunction is being used in functions `MostFreqAgg{}` and `MostFreqDiv{}`.
- **Function Return Type** int

5.2 Function count{}

- **Input :** The input to this function is an array of integers and an integer value say `A[], Pat_L`.
- **Output :** The function returns the number of unique numbers in an array `A[]` up till the length `Pat_L`.
- **Function Description :** The function is quite simple and maintains a temp variable which is incremented whenever a new element in an array `A[]` is found else the count in increased to process the array up till the length defined by `Pat_L`. This function is useful in finding the divisions of the most frequent pattern.
- **Remark :** This subfunction is being used in functions `MostFreqAgg{}` and `MostFreqDiv{}`.
- **Function Return Type** int

5.3 Function Max {}

- **Input :** The input to this function is an array of integers `A[]`, an integer value `L`.
- **Output :** The function returns the position of occurrence of the first maximum element in an array up till a length L.
- **Function Description :** The function finds the first position of occurrence of the maximal element in an array by storing the value of the first element in a temporary variable and then checking for the entire array and if any element is larger than the

value in temporary variable its position is passed into the temporary variable else it is just incremented until the length L as specified by user.

- **Remark :** This subfunction is being used in functions `MostFreqAgg{}`, `aggdiv{}` and `MostFreqDiv{}`.
- **Function Return Type** int

5.4 Function MostFreqAgg{}

- **Input :** The input to this function is a two dimensional array of integers `A[][Pat_L+2]` where the number of columns is $Pat\_L + 2$, an integer value `z` (which gives the row containing the most frequent pattern in the two dimensional array `A[][]`), an integer value `Pat_L` which is simply the pattern length and an integer `L`(which gives the number of rows in an array `A[][Pat_L]` and it is known by the count function described in the above section).
- **Output :** The output of the function is contained in an integer pointer called `AA` which holds most frequent agglomeration and its subsequent most frequent agglomeration and so on. Another pointer called `F_AA` stores the frequencies of the corresponding agglomeration respectively. The memory allocation of the described array in the function is done in a function called `aggdiv{}` which is being called by R.
- **Function Description :** The agglomeration functions finds the most frequent agglomeration of the most frequent pattern and then considering it as the most frequent pattern it finds the most frequent agglomeration of the most frequent agglomeration and so on. The code for finding the agglomerations checks for the following two necessary and sufficient conditions every time a recursive call is made and the other data structures used are mentioned below.
  - The input array `A[][Pat_L+2]` has the columns size as $Pat\_L + 2$. The array contains the most frequent pattern along with the other patterns but the most frequent pattern is rearranged as an increasing order that is all 1's clubbed together and so on and all other pattern has been rearranged according to the most frequent pattern to facilitate the search for the subsequent division. The columns `(Pat_L+1)` holds the frequency of the corresponding pattern in the row and the columns `(Pat_L+2)` holds the value 0 or 1 or 2, where 0 signifies neither an agglomeration nor a division of the most frequent pattern, 1 denotes the agglomeration and 2 denotes the division of the given most frequent pattern.
  - The second input which is an array of size `Pat_L` is the most frequent pattern which is passed into the function from the other function called aggdiv described later.

- There is a temporary two dimensional array which stores the point or position where a label is making a change in their numbers which is used later to satisfy one of the necessary condition for agglomeration.
- Initially all the unique patterns passed in the array `A[][]` has 0 as their last columns which represents neither an agglomeration or division. If a pattern is found to be an agglomeration or division its value is changed according to the above defined rule.
- Now it checks whether each label in the table is an agglomeration of the given most frequent pattern. So, it has `L` iteration and for any general iteration if the value of the column $Pat\_L + 2$ is still 0 or `i` is not equal to the row containing the most frequent row itself, where `i` varies from `0 to L` it checks for the following two condition.
- Firstly, the number of count(as described avove) of unique numbers in the most frequent pattern must be more than the count in the pattern being concerned since it is an agglomeration of the above most frequent pattern and must have lesser number of unique numbers.
- Secondly, the place or position where the number changes in the most frequent pattern which is being stored in a table `T[][]`, for the same positions in the pattern being concerned atleast at one of such locations the labels to their left and the labels at that position must be the same.
- If any general pattern into consideration if it satisfies the above condition then its value is set to 1 showing the agglomeration and the following pattern is copied into another temporary array called `AGG[][]` along with its frequency.
- In such a way every agglomerations of most frequent patterns enters the table `AGG[][]` and at the end a recursive call is made to the functions with parameters as `AGG[][]`, the most frequent Agglomeration, `Pat_L` and `agg+1` which is a local variable which states the number of agglomerations found of the most frequent agglomeration.
- If no such agglomeration exists then the function just returns without any value.
- At each step while finding the most frequent agglomeration to be passed into the recursive function in next call it is also copied in an array called `AA[]` and its corresponding frequency is copied into `F_AA` which is later being used as the output in the function `aggdiv{}`.

- **Remark :** This subfunction is being used in the main function `aggdiv{}`.
- **Function Return Type** void

5.5 Function MostFreqDiv{}

- **Input :** The input to this function is a two dimensional array of integers `A[][Pat_L+2]` where the number of columns is $Pat\_L + 2$, an integer value `z` (which gives the row containing the most frequent pattern in the two dimensional array `A[][]`), an integer value `Pat_L` which is simply the pattern length and an integer `L`(which gives the number of rows in an array `A[][Pat_L]` and it is known by the count function described in the above section).

- **Output :** The output of the function is contained in an integer pointer called `DD[][]` which holds most frequent Division and its subsequent most frequent division and so on. Another pointer called `F_DD[]` which stores the frequencies of the corresponding divisions respectively. The memory allocation of the described array in the function is done in a function called `aggdiv{}` which is being called by R and acts as a main function in this case.

- **Function Description :** The division function finds the most frequent division of the most frequent pattern and then considering it as the most frequent pattern it finds the most frequent division of the most frequent division and so on. The code for finding the division checks for the following two necessary and sufficient conditions every time a recursive call is made and the other data structures used for the purpose are mentioned below.

  - The input array `A[][Pat_L+2]` has the columns size as $Pat\_L + 2$. The array contains the most frequent pattern along with the other patterns but the most frequent pattern is rearranged as an increasing order that is all 1's clubbed together and so on and all other pattern has been rearrranged according to the most frequent pattern to facilitate the search for the subsequent division. The columns $Pat\_L + 1$ holds the frequency of the corresponding pattern in the row and the columns $Pat\_L + 2$ holds the value 0 or 1 or 2, where 0 signifies neither an agglomeration nor a division of the most frequent pattern, 1 denotes the agglomeration and 2 denotes the division of the given most frequent pattern.
  - The second input which is an array of size `Pat_L` is the most frequent pattern which is passed into the function from the other function called `aggdiv{}` described later.
  - There is a temporary two dimensional array which stores the point or position where a label is making a change in their numbers which is used later to satisfy one of the necessary condition for division.
  - Initially all the unique patterns passed in the array `A[][]` has 0 as their last columns which represents neither an agglomeration or division. If a pattern is found to be an agglomeration or division its value is changed according to the above defined rule.

- Now it checks whether each label in the table is a division of the given most frequent pattern. So, it has `L` iteration and for any general iteration if the value of the column $Pat\_L + 2$ is still 0 or `i` is not equal to the row containing the most frequent row itself, where `i` varies from `0 to L` it checks for the following two condition.
- Firstly, the number of count(as described above) of unique numbers in the most frequent pattern must be less than the count in the pattern being concerned since it is a division of the above most frequent pattern and must have more number of unique numbers in its label.
- Secondly, the place or position where the number changes in the most frequent pattern which is being stored in a table `T[]`, for the same positions in the pattern being concerned at all such locations the labels to their left and the labels at that position must be different beside it may be different internally too, that is in between those positions marked in table `T[]`.
- Since, the next label which is being considered for the most frequent pattern may not be in an increasing sequence so we relabel all the divisions according to the most frequent found division and makes a recursive call to find the next most frequent pattern.
- If any general pattern into consideration if it satisfies the above condition then its value is set to 2 showing the division and the following pattern is copied into another temporary array called `DIV[][]` along with its frequency.
- In such a way every divisions of most frequent patterns enters the table `DIV[][]` and at the end a recursive call is made to the functions with parameters as `DIV[][], i(row containing most frequent Division), Pat_L and div+1` which is a local variable which states the number of divisions found of the most frequent division.
- If no such division exists then the function just returns without any value.
- At each step while finding the most frequent division to be passed into the recursive function in next call it is also copied in an array called `DD[]` and its corresponding frequency is copied into `F_DD` which is later being used as the output in the function aggdiv.

- **Remark :** This subfunction is being used in the main function `aggdiv{}`.
- **Function Return Type** void

5.6 Function Rearrange{}

- **Input :** The input to this function is a two dimensional array of integers `D[][Pat_L+2]` where the number of columns is $Pat\_L + 2$, an integer value `Pat_L` which is simply the pattern length, an integer `L`(which gives the number of rows in

an array `A[][Pat_L]` and it is known by the count function described in the above section) and an integer called `z` which gives the row containing the most frequent pattern according to which all other patterns has to be rearranged.

- **Output :** The output of the function is the array `D[][]` itself passed as an input to the function but it now contains the most frequent pattern in an increasing order and every other pattern has been rearranged accordingly.
- **Function Description :** The function simply scans the most frequent pattern and if the prvious element is not smaller than the next element then it swaps the entire column with the 1st largest number in the table until it reaches the end of the array. In the end, all the pattern are now rearranged as per the increasing sequence of the most frequent pattern.
- **Function Return Type** void

5.7 Function aggdiv{}

- **Input :** The input to this function are as follows :
  - An array of integers `A[]` which is passed from the output of the count code and is contained in Pattern,
  - An integer array `B[]` which is the frequency of the corresponding output `Pattern array` from count code and is contained in `Frequency`,
  - An integer value `Pat_L` which is simply the pattern length,
  - An integer `L`(which is the length of the `array B[]`),
  - A pointer to the `PatInBet` which gives the patterns that are neither an agglomeration or division of the given most frequent pattern,
  - A pointer to the `MostFreqPat` which gives the most frequent pattern,
  - A pointer to `F_MostFreqPat` which gives the frequency of the most frequent pattern,
  - A pointer to `Agglomeration` which gives all the most frequent agglomeration of the most frequent pattern,
  - A pointer to `F_Agglomeration` which gives the frequency of the corresponding agglomeration,
  - A pointer to `Division` which gives all the most frequent division of the most frequent pattern,
  - A pointer to `F_Division` which gives the frequency of the corresponding division.

- **Output :** The output of the function is contained in the last seven parameters passed as an input to the functions and it contains the data as described above in the input function.
- **Function Description :** The function generates the desired output by calling all the above function in a desired sequence as described below.
  - The input array which a single dimensional array is converted in to a 2 dimensional array `D[][Pat_L+2]` where the length of the column is $Pat\_L + 2$. The columns $Pat\_L + 1$ holds the frequency of the corresponding pattern in the row which is passed in the column from the `input array B[]` and the columns $Pat\_L + 2$ holds the value 0 initially where 0 signifies that the pattern has not yet been considered before calling agglomeration and division and it represents neither an agglomeration nor a division if the value still remains zero after a functional call to agglomeration and division both.
  - Then row containing the most frequent pattern is found out by making a functional call to `max{}` function and its value is stored in a temporary variable called `z`.
  - The array `D[][]` along with the variable `z` is passed into the function rearrange so that it is as per the increasing sequence of the most frequent pattern and all other patterns are rearranged as per the most frequent pattern.
  - Then a functional call to agglomeration is done to find all possible agglomeration and its subsequent agglomeration.
  - Then the array `D[][]` is also passed along with other parameters to Division function for finding the possible most frequent divisions.
  - Then the patterns which are neither agglomeration nor divisions of the most frequent pattern have the value 0 as their columns in $Pat\_L + 2$ and it is copied into output `PatInBet`.
  - Other output are generated as well and passed into output variables accordingly.
- **Function Return Type** void

5.8 Function Raggdiv {}

- **Input :** The input to this function is a pointer to every input of the function `aggdiv{}`.
- **Output :** The output of the function is as described in the output of the function `aggdiv{}`.

- **Function Description :** This function is only necessary to call the C function from an R interface and everything passed into this function must be of a type pointer. The values to this function is passed by R which is further passed into C function which after operation returns the result in their respective pointers.
- **Function Return Type** void

5.9 Analysis of the code

The procedure for finding the agglomeration, division and neither an agglomeration and division has been performed for the following input datasets which were randomly generated by R software and time analysis was done to see the speed of the algorithm. The test cases are as mentioned below.

5.9.1 Test case 1 :

- **Number of Individuals :** 500
- **Number of Iterations :** 1000
- **Time to generate the Data in R :** 6 seconds
- **Time to Relabel :** fraction of a second
- **Time to Count the Frequency :** fraction of a second
- **Time to find the Agglomeration, Divisions, Patterns which are neither an agglomeration and Division :** fraction of a second

5.9.2 Test case 2 :

- **Number of Individuals :** 500
- **Number of Iterations :** 10,000
- **Time to generate the Data in R :** 8 minutes
- **Time to Relabel :** 7 seconds
- **Time to Count the Frequency :** 10 seconds
- **Time to find the Agglomeration, Divisions, Patterns which are neither an agglomeration and Division :** 40 seconds

6 Summary

It was possible to construct a tree for visualization based on Bayesian Markov Chain Monte Carlo iterations for clustering. The input to the problem was a set of labels produced as an output of the MCMC simulations. These labels represents various possible grouping of the individuals. These labels were uniquely relabeled to eliminate the representation of same grouping of the individuals by different labels. Then, the frequency of occurrence of each label was calculated to find the most frequent pattern in the dataset. Next, we found the most frequent agglomeration(and division) and its all subsequent most frequent agglomerations(and divisions). Using the most frequent pattern and all its most frequent divisions and agglomerations it is possible to visually represent them by constructing a tree called as dendrogram. This dendrogram can be pruned at different levels to generate different numbers of initial grouping of the individuals. Also, it was possible to find other structures which are neither an agglomeration nor a division of the most frequent pattern but which might be supported by the data as their frequency are close to the most frequent pattern.

7 Future Work

A document which will provide general insight for the beginners to use the interface between C and R with ease would be documented. That report will contains simple examples implemented in C and executed from R.