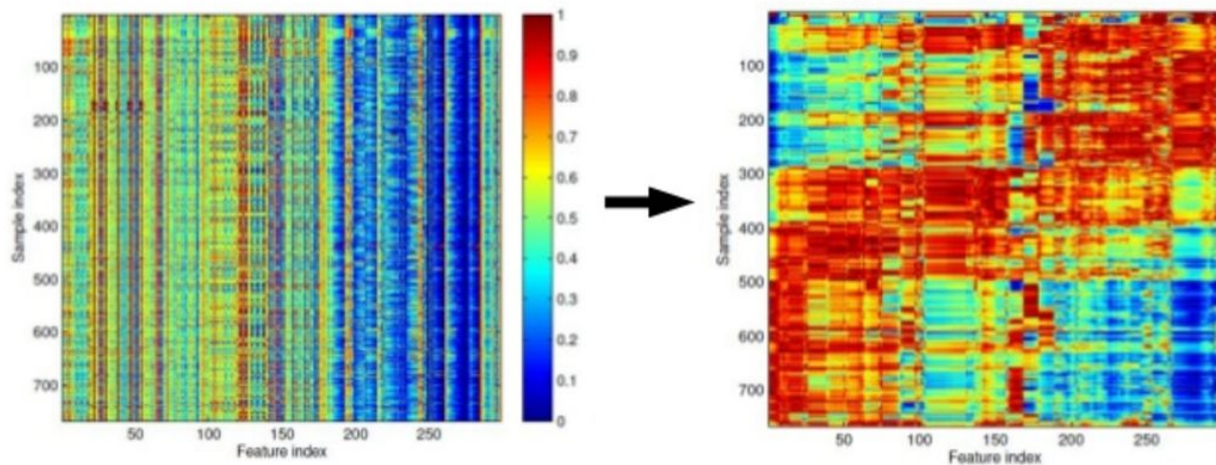


BICLUSTERING ALGORITHM

TETKOSKIE, MUJAWAR, CUMMINGS, CLAYTOR

Biclustering: Before and After



Abstract

In this report, we introduce Biclustering, an unsupervised learning method that clusters rows and columns simultaneously to find inherent groupings of objects in a data set. After providing a hypothetical example and explaining the algorithm, we give tutorials for Biclustering in both R and Python. Finally, we explore a case study using several variations the the algorithm, making use of the “What the kids are up to” dataset collected in Slovakia in 2014.

Algorithm Overview

Biclustering, a term first coined by Boris Mirkin in 1996, is an unsupervised data mining method that is closely related to clustering. Recall, clustering is the process of partitioning a set of data observations into subsets, called clusters, such that objects in a given cluster are similar to each other and dissimilar to other clusters. Biclustering, on the other hand, is a data mining method that simultaneously clusters both rows and columns of a matrix. The process forms groups from inherent properties of the data, and thus allows for a deeper understanding of the objects in the dataset and their relationships to each other. Since its creation, many variations of the original biclustering algorithm have been developed. I will dive into these variations and their applications after first presenting a hypothetical example for understanding.

Hypothetical Example

Suppose you are throwing a party. There will be fifty guests invited to the party, which will be hosted in your three-room apartment. You would like to play music at the party, and your music collection includes thirty albums. Given the wide range of music included in your collection, you are concerned about everyone at the party being satisfied with the selection. So, being the aspiring data scientist that you are, you send out a survey asking each guest if they like each album. The results can thus be organized in a 50×30 binary matrix, M , where $M_{ij} = 1$ if guest i likes album j . Ideally, you would like the guests and albums to be evenly distributed among the three rooms of your apartment. The matrix will then need to be reorganized, with groupings of albums and guests that would correspond to which guests should be in room, and which albums should be played in each room. This scenario can thus be visualized as seen below, where rows are guests, columns are albums,



and the color is blue if the guest likes the album to which the column corresponds. The first image is thus the matrix in its original form, and the second image is the matrix after rows have been reordered to group together guests based on their interests (Eren).

Variations on the Algorithm

Since its creation, there have been various versions of Biclustering developed. These variations include CC clustering, Spectral co-clustering, Spectral biclustering, Plaid Model Biclustering, Quest Motif Biclustering, and more. The variations fit into four main categories, based on the structure of the data and the commonality among the biclusters. These categories are biclusters with constant values, biclusters with constant values on rows, biclusters with constant values on columns, and biclusters with additive or multiplicative coherent values. These variations are visualized below:

a) Bicluster with constant values

2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0

b) Bicluster with constant values on rows

1.0	1.0	1.0	1.0	1.0
2.0	2.0	2.0	2.0	2.0
3.0	3.0	3.0	3.0	3.0
4.0	4.0	4.0	4.0	4.0
5.0	5.0	5.0	5.0	5.0

c) Bicluster with constant values on columns

1.0	2.0	3.0	4.0	5.0
1.0	2.0	3.0	4.0	5.0
1.0	2.0	3.0	4.0	5.0
1.0	2.0	3.0	4.0	5.0
1.0	2.0	3.0	4.0	5.0

Cheng and Church Biclustering

One of the earliest forms of biclustering was that created by Y. Cheng and G.M. Church to solve a problem within gene expression data, in which they cluster both genes and conditions. Their method for biclustering is based on variance, centering on a parameter called the mean square residue, which is a measure of homogeneity. This value is calculated by examining the means within each biclusters rows and columns, and is technically computing as follows: Let A be a matrix, and let the tuple (I, J) represent a bicluster of A , where I is the set of rows in the bicluster, and J is the set of columns. where I is the set of rows in the bicluster, and J is the set of columns. The submatrix of A defined by this bicluster is A_{ij} where $a_{ij} (i \in I, j \in J)$ is an element of the bicluster. The mean residue formula is equal to $a_{ij} - a_{i\cdot} - a_{\cdot j} + a_{\cdot\cdot}$ where the values correspond to the row mean, column mean, and overall mean defined respectively, as follows (below, left):

$$a_{iJ} = \frac{1}{|J|} \sum_j a_{ij}$$

$$a_{Ij} = \frac{1}{|I|} \sum_i a_{ij}$$

$$a_{IJ} = \frac{1}{|I||J|} \sum_{i,j} a_{ij}$$

The goal is thus to place each object in a bicluster such that mean residue for each object is minimized. Cheng and Church thus defined an H score for each biclustering result, a function that is to be minimized. The function goes to zero as the biclusters increase in homogeneity, and is defined as follows:

$$H(I, J) = \frac{1}{|I||J|} \sum_{i \in I, j \in J} (a_{ij} - a_{iJ} - a_{Ij} + a_{IJ})^2$$

As a corollary, note that $H(I, J)$ achieves a minimum when all the rows and columns of the bicluster are shifted versions of each other. Additionally, constant biclusters, biclusters with identical rows or columns, and biclusters with only one row or column clearly have a score of 0. In forming the biclusters, the Cheng and Church algorithm generally finds the largest possible biclusters such that the H score is less than some threshold, δ . Their approach is greedy, starting with the largest possible grouping and removing objects. This can be done without needed to recompute H each time, as the mean squared residue of any row i or any column j in the bicluster is defined as:

$$d(i) = \frac{1}{|J|} \sum_{j \in J} (a_{ij} - a_{iJ} - a_{Ij} + a_{IJ})^2$$

$$d(j) = \frac{1}{|I|} \sum_{i \in I} (a_{ij} - a_{iJ} - a_{Ij} + a_{IJ})^2$$

To delete the objects from the biclusters, Cheng and Church have an accelerated node deletion algorithm that can remove multiple rows at a time, left, with single removal at the right:

Algorithm: multiple node deletion

input: matrix A , row set I , column set J , $\delta \geq 0$, $\alpha > 0$

output: row set I' and column set J' so that $H(I', J') \leq \delta$

while $H(I, J) > \delta$:

 remove from I all rows with $d(i) > \alpha H(I, J)$

 remove from J all columns with $d(j) > \alpha H(I, J)$

if I and J have not changed **then** switch to single node deletion

return I and J

Algorithm: node deletion

input: matrix A , row set I , column set J , $\delta \geq 0$

output: row set I' and column set J' so that $H(I', J') \leq \delta$

while $H(I, J) > \delta$:

 find the row $r = \arg \max_{i \in I} d(i)$ and the column $c = \arg \max_{j \in J} d(j)$

if $d(r) > d(c)$ **then** remove r from I **else** remove c from J

return I and J

Thus, the algorithm stops once H is below a certain threshold. There is also an additional step that attempts to add rows to the biclusters so long as it doesn't negatively affect H (Eren).

Other Methods and Applications

Most versions of biclustering are based on the initial biclustering algorithms created by Cheng and Church. Another popular method is spectral biclustering, which supposes that normalized microarray data matrices have a checkerboard structure that can be discovered by the use of svd decomposition in eigenvectors, applied to genes (rows) and conditions (columns) (Kluger et al.). There is also Questmotif, a biclustering algorithm made specifically to bicluster questionnaire responses, grouping respondents based on their responses to groups of different questions. Beyond analyzing gene expression data, biclustering can also be used within the realms of market research, financial data, agricultural data, and text mining problems.

Python Implementation Tutorial

Python offers a biclustering implementation as part of the popular sk-learn machine learning package. The biclustering implementation is included within the general clustering package: `sklearn.clustering.bicluster`. The bicluster package is then further broken down into two separate functions: Spectral BiClustering, and Spectral CoClustering. A basic visual of the package structure is shown below in Figure 1.

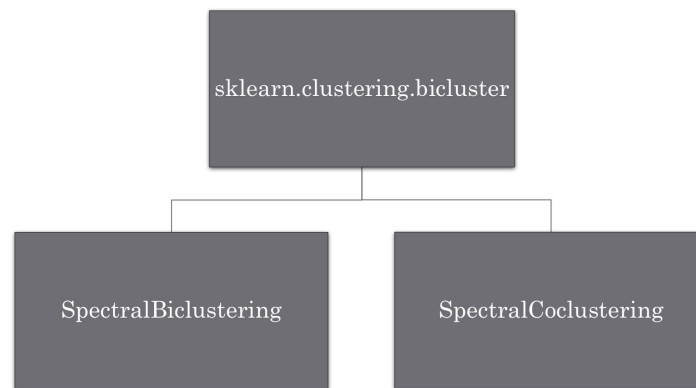


Figure 1: sk-learn bicluster package

Function Application

Each function in the bicluster package serves a specific purpose and it is important to know the differences between them. Before explaining the package implementation, Figure 2 shows a sample output from each biclustering function in Figure 1.

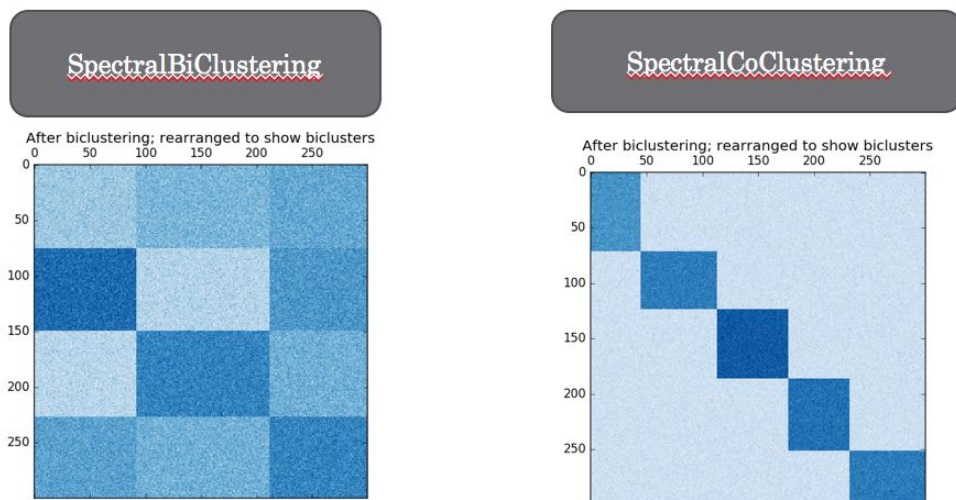


Figure 2: Biclustering Function Sample Output

Figure 2 shows distinctively different outcomes for each function. The SpectralBiClustering (SBC) function creates $(n \times m)$ biclusters to form a checkerboard output. The SpectralCoClustering (SCC) function creates n biclusters arranged along the diagonal. The main difference is that the SBC function can assign a single row or column to several biclusters, while the SCC function assigns each row and column to a single

bicluster. Which function is chosen by the user depends on the specific use case. The hypothetical example use case given in this paper would use the SCC function to assign a specific guest to a single room in which similar music is enjoyed. A potential use case for the SBC using the hypothetical example would be if the party host had many rooms and wanted to give each guest an option of rooms that they would enjoy, instead of a single room. That would produce the checkerboard like output produced by SBC, where each bicluster would represent a different room in the house.

Model Implementation

Model implementation for SCC and SBC functions follows the general format of many other sklearn model implementations. This section aims to provide a detailed walkthrough of each function. Before a function is chosen, you must import the necessary packages.

```
## Bi-cluster packages
from sklearn.cluster.bicluster import SpectralBiclustering
from sklearn.cluster.bicluster import SpectralCoclustering
from sklearn.metrics import consensus_score
## Used to create dataset
from sklearn.datasets import make_biclusters
from sklearn.datasets import samples_generator as sg
import numpy as np
```

Figure 3: Import Packages

SpectralCoClustering (SCC)

The next steps will show how to implement the SCC function. The first step to implement SCC is to initialize the algorithm and set the parameters.

```
model = SpectralCoclustering(n_clusters = 2,
                             svd_method = 'randomized',
                             mini_batch = False,
                             init = 'random',
                             n_init = 10,
                             n_jobs = -1,
                             random_state = 10)
```

Figure 4: Initialize Algorithm

The possible model parameters are displayed in Figure 4 and a description of each value is below in Table 1. Several of these parameters are optimizable and have to deal with model complexity vs run-time. What you choose depends on the size of your dataset and the accuracy tolerance.

Parameter	Description
n_clusters (optimizable)	Number of clusters to solve for
svd_method	'randomized' – Faster for large matrices 'arpack' – Slower but may be more accurate
mini_batch	True/False – To use mini_batch k means as solver -> faster implementation for large datasets
Init (optimizable)	'k-means++' / 'random' / nd_array – Initialization method for k-means
n_init (optimizable)	Number of random k-means initializations
N_jobs	Number of cores to use. -1 = all available cores

Table 1: Model Parameters

After initializing the model, the next step is to fit the model to your data by calling the fit method to the model object using:

```
model.fit(np_data)
```

Once the model has been fit to your dataset, you may want to visualize the output. The sample code shown in Figure 5 uses the _row_labels and _column_labels methods to visualize the data shown in Figure 6. The variable 'np_data' is your original dataset array.

```
fit_data = np_data[np.argsort(model.row_labels_)]
fit_data = fit_data[:, np.argsort(model.column_labels_)]

plt.matshow(fit_data, cmap=plt.cm.Blues)
plt.title("After biclustering; rearranged to show biclusters")
```

Figure 5: Visualize Output

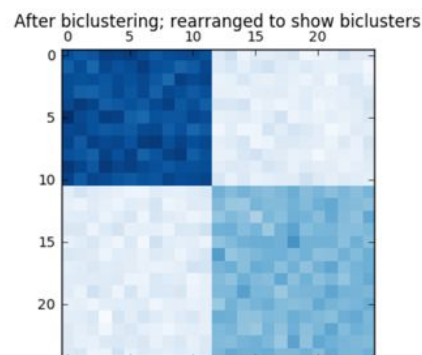


Figure 6: Output Plot

The sk-learn implementation offers other ways to view the output of the SBC and SCC functions. The other output options are shown below in Table 2. You can return the indices, shape, and entire submatrix for each bicluster.

Model Methods	Description
Model.get_indices(i)	To return matrix indices of bicluster i
Model.get_shape(i)	Return shape of ith bicluster
Model.get_submatrix(i,data)	Returns matrix of bicluster i

Table 2: Model Parameters

SpectralBiClustering (SBC)

The SBC implementation is very similar to the implementation of SCC function shown above. All of the same steps apply to initialize and fit the model, and to view the output. The only real difference in implementation comes in the parameter values for the model initialization. The model initialization code is shown in Figure 7 and the new parameters are described in Figure 8.

```
model = SpectralBiclustering(n_clusters = 2,
                             method = 'log',
                             n_components = 6,
                             n_best = 3,
                             svd_method = 'randomized',
                             init = 'random',
                             n_init = 10,
                             n_jobs = -1,
                             random_state = 10)
```

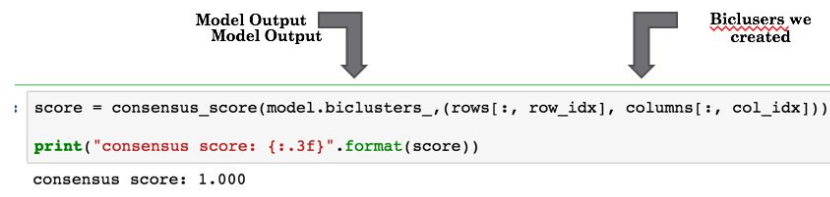
Figure 7: SBC Model Initialization

New Parameters	Description
method	'scale'/'bistochastic','log' 'log' will not work with sparse matrix
n_components (optimizable)	Number of singular vectors to check
N_best (optimizable)	Number of best singular vectors to use for clustering

Figure 8: SBC New Parameter Description

Model Evaluation

Finally, the last step is to evaluate your biclusters. Currently there are no internal methods to evaluate biclusters implemented within sk-learn. This personally is a large drawback in that any evaluation of the clusters internally (without knowing the ground truth) would need to be implemented yourself. The only current implementation for evaluation is to measure them against known ground truth. In many (or most) cases, this is not known. If you do know the ground truth, you can use the `sklearn.metrics.consensus_score` function. A perfect score of 1 means that you perfectly replicated the ground truth. A sample of how to implement consensus score is in Figure 9.



```
score = consensus_score(model.biclusters_,(rows[:, row_idx], columns[:, col_idx]))
print("consensus score: {:.3f}".format(score))
consensus score: 1.000
```

Figure 9: sk-learn consensus score implementation

R Implementation Tutorial:

The R package `biclust()` has several biclustering algorithms along with various functions needed while using these algorithms. For instance the Bimax algorithm searches for submatrices of ones in a logical matrix which means one needs to binarize the data matrix into 1's and 0's before using it in the `biclust()`. For this purpose the package has `binarize()` which converts a real matrix to a binary matrix. This tutorial will be covering the `BCCC()`, various visualization and evaluation functions to give a clear idea of how Cheng and Church algorithm works. Below are some of the important methods and functions provided by the `biclust` package in R .

Methods	Other Functions
<u>BCBimax</u>	<u>drawHeatmap()</u>
<u>BCCC()</u>	<u>heatmapBC()</u>
<u>BCPlaid</u>	<u>parallelCoordinates()</u>
<u>BCQuest</u>	<u>bubbleplot()</u>
<u>BCXmotifs</u>	<u>binarize()</u>
<u>BCSpectral</u>	<u>biclustbarchart()</u>

Figure 11

Input data:

I have hardcoded the data matrix for better visualizations of the results. The `rbinom()` generates required number of random values of given probability from a given sample and the `matrix()` creates a matrix of dimensions 20x20. To visualize the values in the matrix one can use a heat map. It is a graphical representation of data where the individual values contained in a matrix are represented as colors.

```
#CC BICLUSTERING
library(biclust)
#Create data matrix
ccmat <- matrix(rbinom(400, 50, 0.4), 20,
20)
drawHeatmap(ccmat)
```

Figure 12

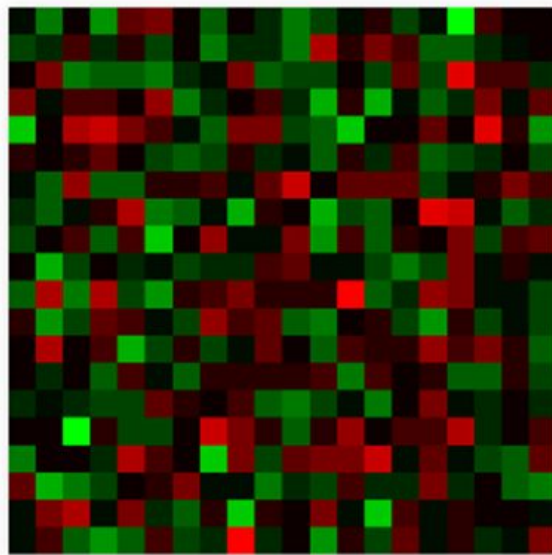


Figure 13

Function Application:

For performing the CC algorithm the method argument in the `biclust()` needs to be passed as `BCCC()` along with three other important parameters; delta, alpha and the number of biclusters to be found. The `biclust()` returns an object of class `Biclust`.

Parameters	Description
x	Data matrix.
method	Here BCCC, to perform CC algorithm
delta	Maximum of accepted score.
alpha	Scaling factor.
number	Number of biclusters to be found.

Figure 14

```
> res <- biclust(ccmat, method=BCCC(), delta=1, alpha=1.5, number=10)
> res
```

An object of class Biclust

call:
biclust(x = ccmat, method = BCCC(), delta = 1, alpha = 1.5, number = 10)

Number of Clusters found: 4

First 4 Cluster sizes:

	BC 1	BC 2	BC 3	BC 4
Number of Rows:	8	4	4	3
Number of Columns:	4	6	4	6

The result contains of 4 biclusters which were obtained with a delta = 1 and an alpha of 1.5. The number of biclusters obtained will change as the values of the 'delta' and 'alpha' change. We will see more of the effects of these values in Parameter tuning. It is important to select the appropriate values of these parameters in order to get homogenous biclusters.

Parameter Tuning:

Cheng and Church use a greedy iterative search to minimize the mean square residue (MSR). The resulting biclusters are as large as possible such that the H-score is below the threshold. This threshold is the delta value. The algorithm searches for a subgroup where the H score < delta and above an alpha fraction of the overall score.

In clustering (k-means) we need to recalculate the centroid after every iteration. In Cheng and church it becomes very computationally expensive to recalculate the H-score for every iteration. Thus deletion of nodes is done by using alpha which prevents the recalculation of the H-score after each addition or deletion of a node. The alpha fraction is nothing but the rate at which we want to delete the nodes selected by the greedy approach of the algorithm.

Delta:

- The quality of the clusters selected is determined by this delta value.

-
- Smaller value of delta gives better results.
 - Very small values of delta result in many small biclusters which may lead to loss of information

Alpha:

- The speed at which the algorithm clusters the data is determined by the alpha value.
- Greater the alpha value faster the deletion

The algorithm searches for a subgroup where the H score $<$ and above α -fraction of the overall score.

Parameter tuning can be done by testing results by varying delta values on x-axis and submatrix size (the number of rows and columns of the selected bicluster) on y-axis. If you plot all these values you'll see that as the delta increases the algorithm starts incorporating more and more of the rows and columns of the matrix eventually ending up with the original data matrix. The key is choosing the x value for delta where the slope changes.

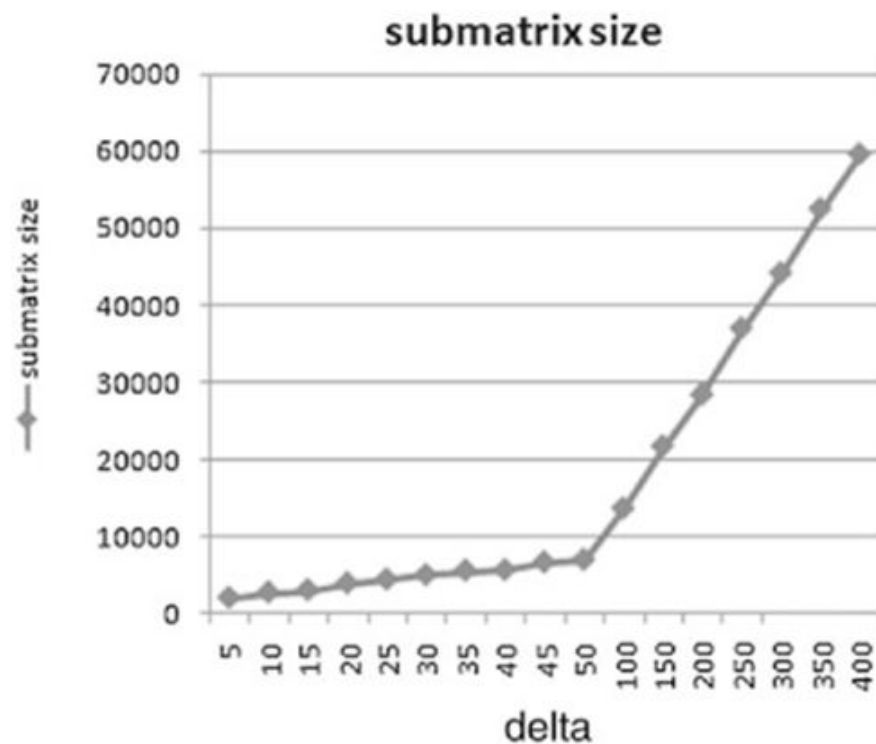


Figure 14 [Proceedings of the Fifth International Conference on Fuzzy and Neuro Computing]

For instance in Figure 14 above the appropriate value of delta would be around 50 for good results. The results of different delta values in `biclust()` are shown below in Figure 15. For `delta=1` we obtained 4 biclusters and as we go on increasing this value at about `delta=10` we obtain the original 20x20 matrix which itself forms 1 bicluster.

Results:

- $\delta = 1$

Number of Clusters found: 4

First 4 Cluster sizes:

	BC 1	BC 2	BC 3	BC 4
Number of Rows:	8	4	4	3
Number of Columns:	4	6	4	6

- $\delta = 3$

Number of Clusters found: 3

First 3 Cluster sizes:

	BC 1	BC 2	BC 3
Number of Rows:	10	6	4
Number of Columns:	8	9	11

- $\delta = 7$

Number of Clusters found: 2

First 2 Cluster sizes:

	BC 1	BC 2
Number of Rows:	17	3
Number of Columns:	15	17

- $\delta = 10$

There was one cluster found with
20 Rows and 20 columns

Figure 15

```
par(mfrow= c(1,2))
drawHeatmap(ccmat,res,1)
drawHeatmap(ccmat,res,2)
```

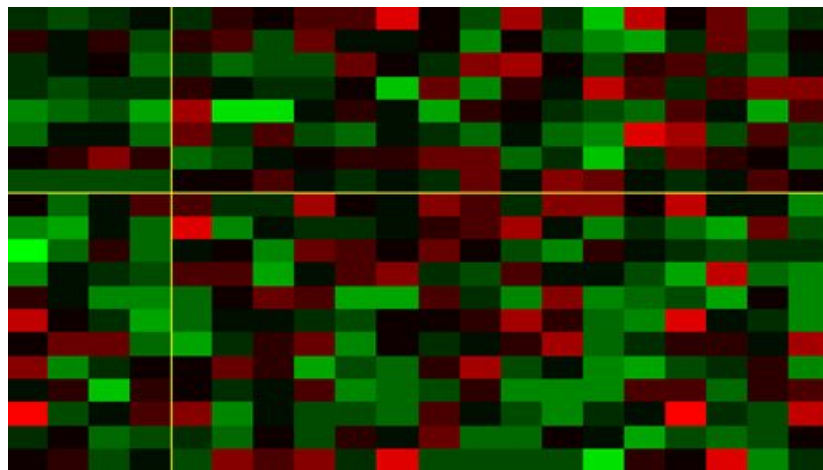


Figure 16

The drawHeatmap() needs to be given each bicluster result to build a heat map. By default the function builds the bicluster heat map and places it at the topmost left corner (Figure 16) and iteratively keeps adding the biclusters in the same pattern. Below are the 4 bicluster heat maps. We can from the color scheme the biclusters are fairly homogenous.

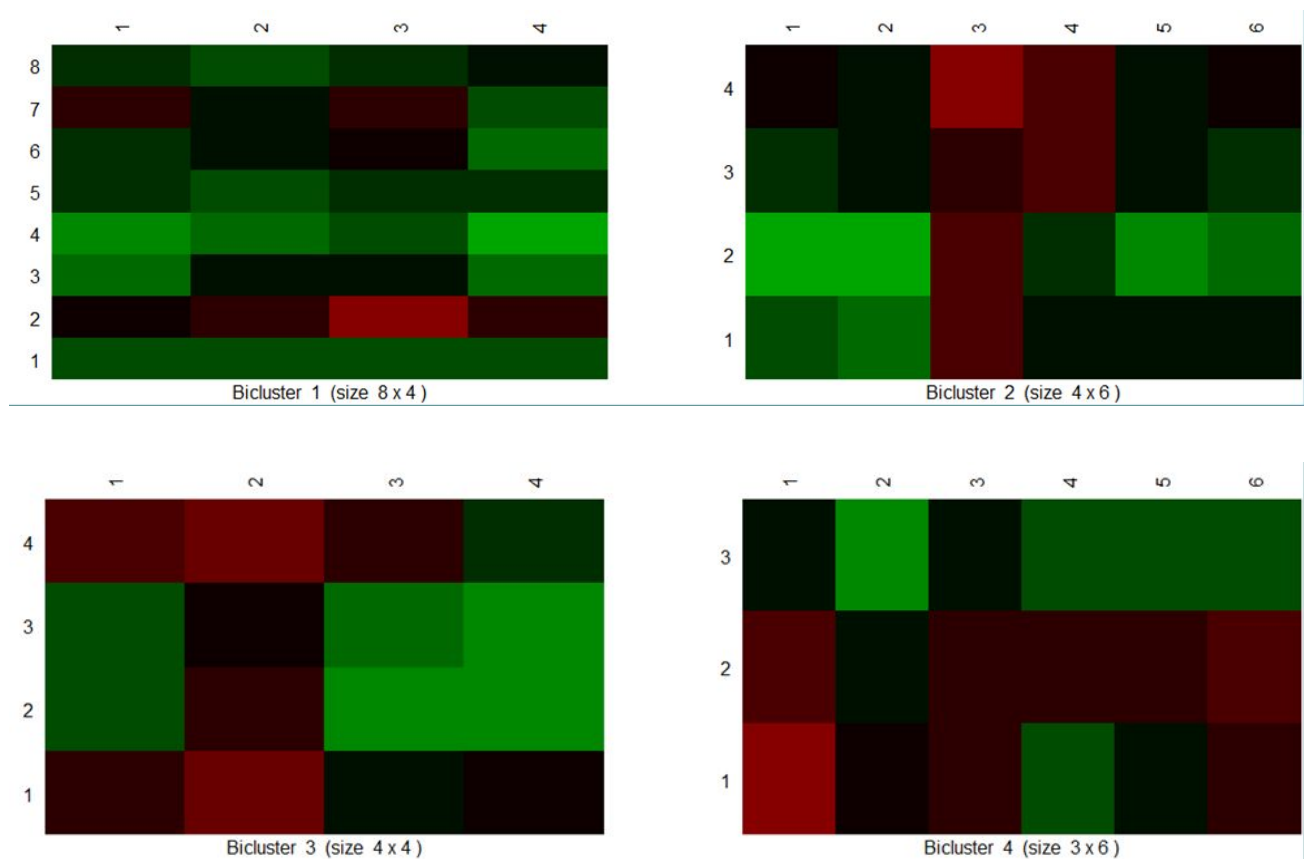


Figure 1

The drawback of using drawHeatmap() is that you have to pass the bicluster number each time for producing the heat maps for all biclusters. The heatmapBC() overcomes this drawback and produces heat maps for all biclusters simultaneously. It also gives us the option of ordering these biclusters. (figure 18)

```
> heatmapBC(x = ccmat, bicResult = res, order=TRUE)
```

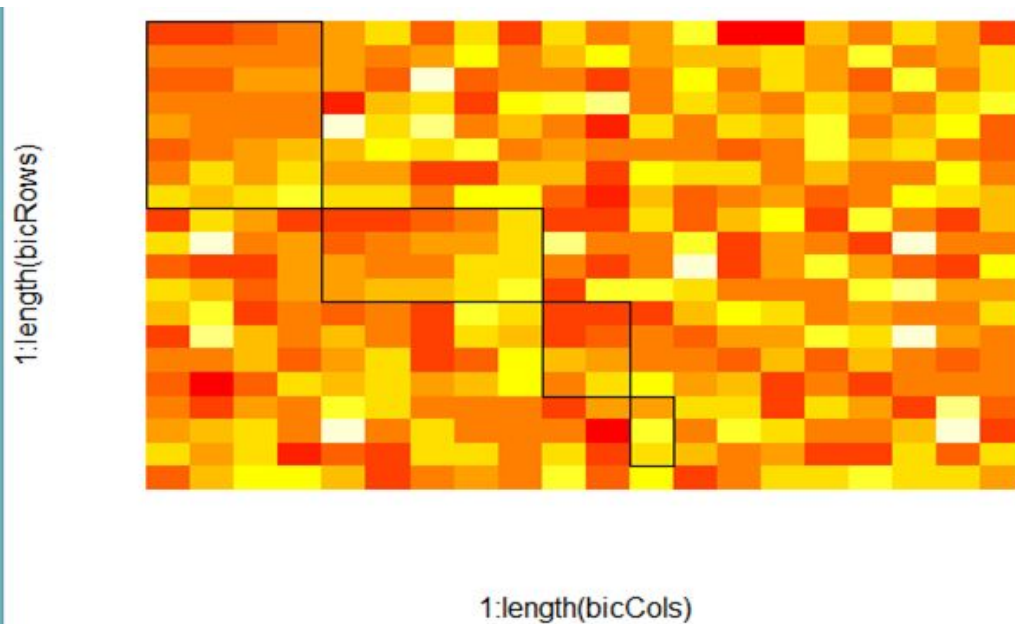


Figure 18

Model Evaluation:

[Parallel coordinates](#) are usually used to visualize high dimensional multivariate data. Visualizing the rows and columns of the biclusters with parallel coordinates plot makes the pattern much easier to see.

Parameters	Description
X	Data matrix
<u>bicResult</u>	Bi-cluster result
<u>plotBoth</u>	If 'TRUE', Parallel Coordinates of rows and column are drawn together.
compare	If 'TRUE', values of the complete data matrix are considered and drawn as shaded lines.
order	Rows and/or Columns are in increasing order.

Parameters	Description
number	Bicluster to be drawn from the result set ' <u>bicResult</u> '
<u>bothlab</u>	Names of the x Axis if <u>PlotBoth</u>
<u>plotcol</u>	If 'TRUE', columns profiles are drawn, so each line represents one of the columns in the <u>bicluster</u>
...	Plot parameters

As a corollary one of the special cases is: constant biclusters, biclusters with identical rows or columns, and biclusters with only one row or column clearly have a score of 0. Evidently shifting of the values for any constant does not change its score.

Another corollary is that if the rows and columns of a bicluster are scaled versions of each other, log-transforming the bicluster makes its H score 0. Therefore, Cheng and Church can also find biclusters with scaling patterns. For example in the Figure 19 you can see that the **vectors differ from each other by only an additive constant**. This is the ideal case where your H-score=0 which means that your biclusters are homogenous.

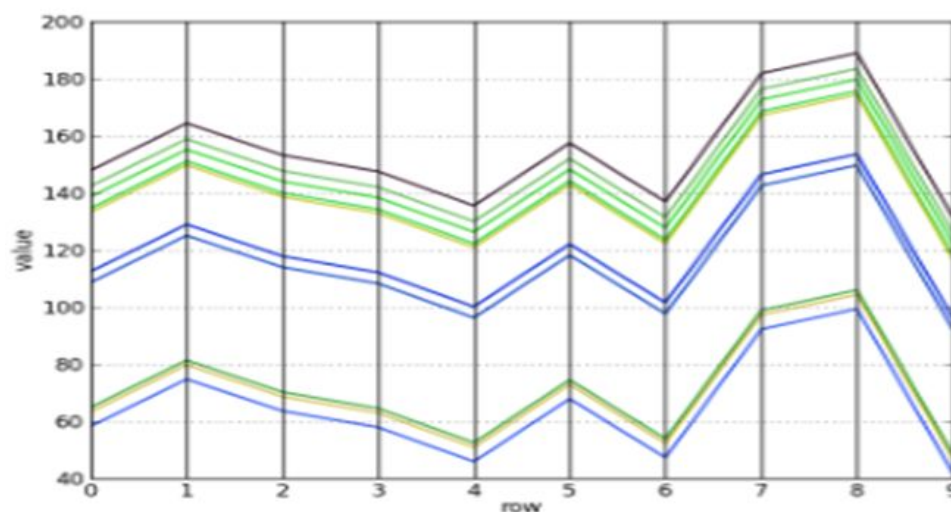


Figure 19[<http://www.kemalaren.com/cheng-and-church.html>]

The parallel coordinates of the biclusters prove to be fairly homogenous as the **vectors are approximately shifted versions of each other**.

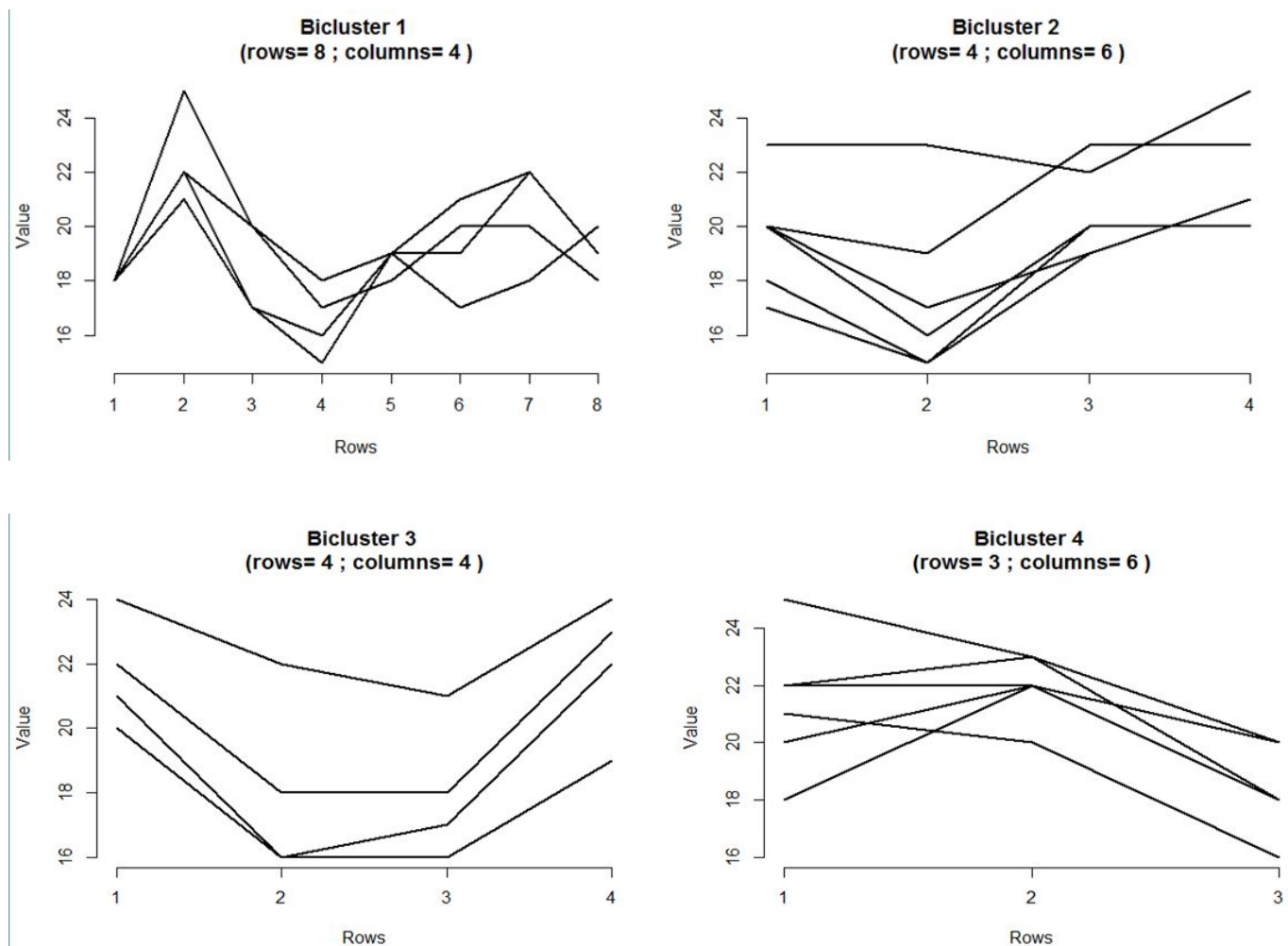


Figure 14

[Note: This R tutorial was prepared with a goal of explaining the correct usage of the various arguments used in the functions given by the biclust package in R, parameter tuning and various visualizations of results. I did not pick a dataset because it would be more like a Case study than a tutorial]

A Biclustering Case Study in R

Introduction

For the purposes of this case study I chose a data set titled [Young People Survey](#) sourced from Kaggle. The data was generated in 2013 when a Statistics class in Slovakia asked their friends to complete a survey about their hobbies, interests, and lifestyles. In all, 1010 respondents between the ages of 15 and 30 addressed 150 questions around subjects such as music interests, health, and spending habits. Through the use of biclustering, we'll attempt to identify subgroups of respondents (rows) who answer similarly to a subgroup of questions (columns). In order to accomplish this we will focus on four major biclustering algorithms. To evaluate the clusters produced, we will consider both the average standard deviation among the columns in the cluster (biclusters with near constant values on columns) as well as a statistic that considers the euclidean distances between all the rows and clusters and columns in a bicluster known as coherence. The complete code used for this Case Study can be found in the Appendix of this report.

Data Description and Cleaning

Of the 150 questions (columns) available in the survey, 139 are integer and 11 are categorical. All integer values range from 1 (strongly disagree) to 5 (strongly agree). For example, a respondent may give a response of 5 if they are highly interested in a given subject and 1 if they have a strong disinterest. The remaining columns of responses are categorical. Examples of categorical responses include whether the respondent lives in a house or an apartment or the gender of the respondent. In order to keep similarity calculations consistent across all response types, categorical responses were eliminated from the data set prior to evaluation. Because the numerical responses are already formatted as integers between 1 and 5, no normalization was required. In the event of missing data, a response of 3 was provided as this represents the "middle of the road" response available to survey respondent. Finally, some Biclustering algorithms -- such as

the Xmotifs Biclust algorithm -- require a discrete matrix as input. With this in mind, we created a copy of the matrix using the discretize() function in R to use as input for algorithms of this type.

Experimental Results

As mentioned above, results of four different biclustering algorithms were considered for this analysis. The first considered was the Bimax Biclust algorithm, which performs Bimax Biclustering based on the framework by Prelic et. al.(2006). The Bimax Biclust algorithm (BCBimax) searches for submatrices of ones in a logical matrix. Next, we considered the CC Biclust algorithm (BCCC) which performs CC Biclustering based on the framework by Cheng and Church (2000) and searches for submatrices with a score lower than a specific threshold in a standardized data matrix. Next, we considered the Questmotif Biclust algorithm (BCQuest). This approach was written as a Biclust algorithm for questionnaires based on the framework by Murali and Kasif (2003) and searches subgroups of questionnaires with same or similar answer to some questions. Finally, we consider the Xmotifs Biclust algorithm (BCXmotifs) which Biclustering based on the framework by Murali and Kasif (2003). This approach searches for a submatrix where each row as a similar motif through all columns. The Algorithm also needs a discrete matrix to perform as expected.

For evaluation purposes, we'll consider the average standard deviation of the columns and coherence of the first cluster generated by each of the four algorithms described above. Generally speaking, the lower the average standard deviation and coherence, the more tightly bound a biclust is. Table 3 below displays the attributes of the first biclust generated by each of the four algorithms in question.

Cluster Metric	BCBimax	BCCC	BCQuest	BCXmotifs
Avg Standard Deviation	1.39	1.02	0	0
Number of Columns	135	94	3	33
Number of Rows	1010	661	481	4

Coherence	21.01	8.80	0	0
-----------	-------	------	---	---

Table 3 - Characteristics of first bicluster produced by each algorithm

Looking at the characteristics of these first clusters, it seems that the Questmotif Bicluster algorithm has generated the best cluster in terms of low coherence, low standard deviation, high number of rows, and high number of columns. This holds empirically given that we're evaluating survey data and the Questmotif Bicluster algorithm was written explicitly with this data type in mind. We look further into the clusters generated with this approach to reveal some common attributes among the clusters produced: low coherence, low standard deviation, high number of rows, and a relatively low number of clusters. Table 4 below shows the attributes of the first five clusters generated with this approach.

Cluster Metric	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Avg Standard Deviation	0	0	0	0	0
Number of Columns	3	2	2	3	2
Number of Rows	481	233	80	30	66
Coherence	0	.13	.11	0	.36

Table 4 - the Questmotif Bicluster algorithm

Drilling further still, Table 5 below looks into the columns that were included in each of the first three clusters generated by the Questmotif Bicluster algorithm to see which questions were answered similarly and how.

By way of contrast, note how the standard CC Bicluster algorithm generates clusters that are generally much larger in terms of columns and rows included but which come at the

cost of a higher coherence. Table 6 below displays the characteristics of the firsts five biclusters generated by the CC Bicluster algorithm.

Cluster 1 - 48% of respondents

Metric	I enjoy listening to music	I really enjoy watching movies	Hobbies - Socializing
Standard Deviation	0	0	0
Mean	5	5	5

Cluster 2 - 8% of respondents

Metric	Hobbies - Shopping	I believe all my personality traits are positive
Standard Deviation	0	0
Mean	5	3

Cluster 3 - 14% of respondents

Metric	Movies - Comedies	Hobbies - Gardening
Standard Deviation	0	0
Mean	5	1

Table 5 - Questmotif Clusters

Cluster Metric	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
Avg Standard Deviation	1.02	1.05	1.06	1.05	1.02
Number of Columns	94	38	34	33	55
Number of	661	179	64	36	29

Rows					
Coherence	8.8	6.0	5.3	5.1	4.8

Table 6 - CC Bicluster algorithm

Experimental Analysis and Conclusion

As should be expected with questionnaire data, we found that the Questmotif Bicluster algorithm produced the best biclusters in that they generally have low coherence with a relatively high number of rows. The more standard Bicluster algorithm, by contract, returned larger clusters but at the cost of significantly higher coherence values. An evaluation of the individual responses returned within the clusters generated by the Questmotif Bicluster algorithm holds empirically. Specifically, we see that individuals who strongly agree that they really enjoy listening to music also are likely to enjoy movies and socializing. One interesting outcome is found in the second cluster returned by this algorithm, where individuals who answer that they strongly agree that they enjoy shopping as a hobby tend to be neutral about whether all of their personality traits are positive.

Considerations for future research include performing more fine-tuning of the parameters that are required by each biclustering algorithm. This may result in larger biclusters and/or biclusters with smaller coherence values. We could also make an effort to normalize the categorical responses so that they could also be taken into consideration as part of the analysis.

Citations

Eren, Kemal. "An introduction to biclustering".

<http://www.kemaleren.com/an-introduction-to-biclustering.html>.

Kluger et al., "Spectral Biclustering of Microarray Data: Coclustering Genes and Conditions", Genome Research, 2003, vol. 13, pages 703-716

Appendix

R Tutorial:

```
library(biclust)

#Create data matrix

set.seed(1)

ccmat <- matrix(rbinom(400, 50, 0.4), 20, 20)

drawHeatmap(ccmat)

head(ccmat)

#cc biclustering

res <- biclust(ccmat, method=BCCC(), delta=1, alpha=1.5, number=10)

res

res <- biclust(ccmat, method=BCCC(), delta=3, alpha=1.5, number=10)

res

res <- biclust(ccmat, method=BCCC(), delta=7, alpha=1.5, number=10)

res

res <- biclust(ccmat, method=BCCC(), delta=10, alpha=1.5, number=10)

res

par(mfrow= c(1,1))

drawHeatmap(ccmat,res,1,local=FALSE)

drawHeatmap(ccmat,res,2,local=FALSE)

drawHeatmap(ccmat,res,3)

drawHeatmap(ccmat,res,4)
```

```
heatmapBC(x = ccmat, bicResult = res, order=TRUE)

#bubbleplot

bubbleplot(ccmat, res, showLabels=TRUE)

#biclust member

biclustmember(res, ccmat)

ord<-bicorder(res, cols=TRUE, rev=TRUE)

biclustmember(res, ccmat, which=ord)

#parallel coordinates: Represents condition profiles in a biclust as lines.

parallelCoordinates(x=ccmat, bicResult=res, number=1, plotBoth=FALSE, plotcol=TRUE, compare=FALSE, info=TRUE)

parallelCoordinates(x=ccmat, bicResult=res, number=2, plotBoth=FALSE, plotcol=TRUE, compare=FALSE, info=TRUE)

parallelCoordinates(x=ccmat, bicResult=res, number=3, plotBoth=FALSE, plotcol=TRUE, compare=FALSE, info=TRUE)

parallelCoordinates(x=ccmat, bicResult=res, number=4, plotBoth=FALSE, plotcol=TRUE, compare=FALSE, info=TRUE)
```

R Case Study Code:

```
#install.packages("biclust")

library(biclust)

# initiate data frame to hold results

df.cols <- c("s.d.", "numcols", "numrows", "coherence")

df <- data.frame(df.cols)

#df.d <- data.frame(df.cols)

# load data

survey <- read.csv("C:\\Users\\johnr\\OneDrive\\Ross\\CSC529\\final\\responses.csv", header=T)

# replace NA values

survey[is.na(survey)] <- 3

# remove categorical variables

survey$Smoking <- NULL

survey$Alcohol <- NULL

survey$Punctuality <- NULL
```

```

survey$Lying <- NULL
survey$Internet.usage <- NULL
survey$Gender <- NULL
survey$Left...right.handed <- NULL
survey$Education <- NULL
survey$Only.child <- NULL
survey$Village...town <- NULL
survey$House...block.of.flats <- NULL
survey$Age <- NULL
survey$Weight <- NULL
survey$Number.of.friends <- NULL
survey$Number.of.siblings <- NULL

# transform to matrix
survey <- as.matrix(survey)
survey.d <- discretize(survey)

# BCBimax The Bimax Biclust algorithm
BCBimax <- biclust(survey,method=BCBimax(), minr=2, minc=2)
#BCBimax.d <- biclust(survey.d,method=BCBimax(), minr=2, minc=2)

# collect attributes of first biclust
BCBimax.number1 <- biclusternumber(BCBimax,1)
BCBimax.c1 <- survey[BCBimax.number1$Biclust1$Rows,BCBimax.number1$Biclust1$Cols]
BCBimax.c1.sd <- mean(apply(BCBimax.c1, 2, sd))
BCBimax.c1.colnames <- colnames(BCBimax.c1)
BCBimax.c1.cols <- length(BCBimax.number1$Biclust1$Cols)
BCBimax.c1.rows <- length(BCBimax.number1$Biclust1$Rows)
BCBimax.c1.coherence <- constantVariance(survey, BCBimax, 1, dimension="both")
new.rec <- c(BCBimax.c1.sd,BCBimax.c1.cols,BCBimax.c1.rows,BCBimax.c1.coherence)
df <- cbind(df,BCBimax=new.rec)

#plotclust(BCBimax,survey,biclust=TRUE,legende=FALSE,noC=1,wyld=3,Titel="BCBimax")

# discretized data
#BCBimax.d.number1 <- biclusternumber(BCBimax.d,1)

```

```
#BCBimax.d.c1 <- survey.d[BCBimax.d.number1$Biclust1$Rows,BCBimax.d.number1$Biclust1$Cols]
#BCBimax.d.c1.sd <- mean(apply(BCBimax.d.c1, 2, sd))
#BCBimax.d.c1.colnames <- colnames(BCBimax.d.c1)
#BCBimax.d.c1.cols <- length(BCBimax.d.number1$Biclust1$Cols)
#BCBimax.d.c1.rows <- length(BCBimax.d.number1$Biclust1$Rows)
#BCBimax.d.c1.coherence <- constantVariance(survey, BCBimax.d, 1, dimension="both")
#new.rec <- c(BCBimax.d.c1.sd,BCBimax.d.c1.cols,BCBimax.d.c1.rows,BCBimax.d.c1.coherence)
#df.d <- cbind(df.d,BCBimax.d=new.rec)
```

```
# BCCC The CC Biclust algorithm
BCCC <- biclust(survey, method=BCCC(), delta = 1.0, alpha=1.5, number=100)
#BCCC.d <- biclust(survey.d, method=BCCC(), delta = 1.0, alpha=1.5, number=100)
# collect attributes of first biclust
BCCC.number1 <- biclusternumber(BCCC,1)
BCCC.c1 <- survey[BCCC.number1$Biclust1$Rows,BCCC.number1$Biclust1$Cols]
BCCC.c1.sd <- mean(apply(BCCC.c1, 2, sd))
BCCC.c1.colnames <- colnames(BCCC.c1)
BCCC.c1.cols <- length(BCCC.number1$Biclust1$Cols)
BCCC.c1.rows <- length(BCCC.number1$Biclust1$Rows)
BCCC.c1.coherence <- constantVariance(survey, BCCC, 1, dimension="both")
new.rec <- c(BCCC.c1.sd,BCCC.c1.cols,BCCC.c1.rows,BCCC.c1.coherence)
df <- cbind(df,BCCC=new.rec)
#plotclust(BCCC,survey,biclust=TRUE,legende=FALSE,noC=1,wyl=3,Titel="BCCC")
# discretized data
#BCCC.d.number1 <- biclusternumber(BCCC.d,1)
#BCCC.d.c1 <- survey.d[BCCC.d.number1$Biclust1$Rows,BCCC.d.number1$Biclust1$Cols]
#BCCC.d.c1.sd <- mean(apply(BCCC.d.c1, 2, sd))
#BCCC.d.c1.colnames <- colnames(BCCC.d.c1)
#BCCC.d.c1.cols <- length(BCCC.d.number1$Biclust1$Cols)
#BCCC.d.c1.rows <- length(BCCC.d.number1$Biclust1$Rows)
#BCCC.d.c1.coherence <- constantVariance(survey, BCCC.d, 1, dimension="both")
#new.rec <- c(BCCC.d.c1.sd,BCCC.d.c1.cols,BCCC.d.c1.rows,BCCC.d.c1.coherence)
```

```
#df.d <- cbind(df.d,BCCC.d=new.rec)

BCCC.c1.mean <- apply(BCCC.c1, 2, mean)
BCCC.c1.sd <- apply(BCCC.c1, 2, sd)

BCCC.number2 <- biclusternumber(BCCC,2)
BCCC.c2 <- survey[BCCC.number2$Biclusternumber2$Rows,BCCC.number2$Biclusternumber2$Cols]
BCCC.c2.mean <- apply(BCCC.c2, 2, mean)
BCCC.c2.sd <- mean(apply(BCCC.c2, 2, sd))
BCCC.c2.rows <- length(BCCC.number2$Biclusternumber2$Rows)
BCCC.c2.cols <- length(BCCC.number2$Biclusternumber2$Cols)
BCCC.c2.coherence <- constantVariance(survey, BCCC, 2, dimension="both")

BCCC.number3 <- biclusternumber(BCCC,3)
BCCC.c3 <- survey[BCCC.number3$Biclusternumber3$Rows,BCCC.number3$Biclusternumber3$Cols]
BCCC.c3.mean <- apply(BCCC.c3, 2, mean)
BCCC.c3.sd <- mean(apply(BCCC.c3, 2, sd))
BCCC.c3.rows <- length(BCCC.number3$Biclusternumber3$Rows)
BCCC.c3.cols <- length(BCCC.number3$Biclusternumber3$Cols)
BCCC.c3.coherence <- constantVariance(survey, BCCC, 3, dimension="both")

BCCC.number4 <- biclusternumber(BCCC,4)
BCCC.c4 <- survey[BCCC.number4$Biclusternumber4$Rows,BCCC.number4$Biclusternumber4$Cols]
BCCC.c4.mean <- apply(BCCC.c4, 2, mean)
BCCC.c4.sd <- mean(apply(BCCC.c4, 2, sd))
BCCC.c4.rows <- length(BCCC.number4$Biclusternumber4$Rows)
BCCC.c4.cols <- length(BCCC.number4$Biclusternumber4$Cols)
BCCC.c4.coherence <- constantVariance(survey, BCCC, 4, dimension="both")

BCCC.number5 <- biclusternumber(BCCC,5)
BCCC.c5 <- survey[BCCC.number5$Biclusternumber5$Rows,BCCC.number5$Biclusternumber5$Cols]
BCCC.c5.mean <- apply(BCCC.c5, 2, mean)
BCCC.c5.sd <- mean(apply(BCCC.c5, 2, sd))
```

```

BCCC.c5.rows <- length(BCCC.number5$Bicluster5$Rows)
BCCC.c5.cols <- length(BCCC.number5$Bicluster5$Cols)
BCCC.c5.coherence <- constantVariance(survey, BCCC, 5, dimension="both")

# BCQuest The Questmotif Bicluster algorithm
BCQuest <- biclust(survey, method=BCQuest(), ns=10, nd=10, sd=5, alpha=0.05, number=100)
#BCQuest.d <- biclust(survey.d, method=BCQuest(), ns=10, nd=10, sd=5, alpha=0.05, number=100)

# collect attributes of first bicluster
BCQuest.number1 <- biclusternumber(BCQuest,1)
BCQuest.c1 <- survey[BCQuest.number1$Bicluster1$Rows,BCQuest.number1$Bicluster1$Cols]
BCQuest.c1.mean <- apply(BCQuest.c1, 2, mean)
BCQuest.c1.sd <- mean(apply(BCQuest.c1, 2, sd))
BCQuest.c1.colnames <- colnames(BCQuest.c1)
BCQuest.c1.cols <- length(BCQuest.number1$Bicluster1$Cols)
BCQuest.c1.rows <- length(BCQuest.number1$Bicluster1$Rows)
BCQuest.c1.coherence <- constantVariance(survey, BCQuest, 1, dimension="both")
new.rec <- c(BCQuest.c1.sd,BCQuest.c1.cols,BCQuest.c1.rows,BCQuest.c1.coherence)
df <- cbind(df,BCQuest=new.rec)

BCQuest.c1.mean <- apply(BCQuest.c1, 2, mean)
BCQuest.c1.sd <- apply(BCQuest.c1, 2, sd)

BCQuest.number2 <- biclusternumber(BCQuest,2)
BCQuest.c2 <- survey[BCQuest.number2$Bicluster2$Rows,BCQuest.number2$Bicluster2$Cols]
BCQuest.c2.mean <- apply(BCQuest.c2, 2, mean)
BCQuest.c2.sd <- mean(apply(BCQuest.c2, 2, sd))
BCQuest.c2.rows <- length(BCQuest.number2$Bicluster2$Rows)
BCQuest.c2.cols <- length(BCQuest.number2$Bicluster2$Cols)
BCQuest.c2.coherence <- constantVariance(survey, BCQuest, 2, dimension="both")

BCQuest.number3 <- biclusternumber(BCQuest,3)
BCQuest.c3 <- survey[BCQuest.number3$Bicluster3$Rows,BCQuest.number3$Bicluster3$Cols]
BCQuest.c3.mean <- apply(BCQuest.c3, 2, mean)

```

```

BCQuest.c3.sd <- mean(apply(BCQuest.c3, 2, sd))
BCQuest.c3.rows <- length(BCQuest.number3$Bicluster3$Rows)
BCQuest.c3.cols <- length(BCQuest.number3$Bicluster3$Cols)
BCQuest.c3.coherence <- constantVariance(survey, BCQuest, 3, dimension="both")

BCQuest.number4 <- biclusternumber(BCQuest,4)
BCQuest.c4 <- survey[BCQuest.number4$Bicluster4$Rows,BCQuest.number4$Bicluster4$Cols]
BCQuest.c4.mean <- apply(BCQuest.c4, 2, mean)
BCQuest.c4.sd <- mean(apply(BCQuest.c4, 2, sd))
BCQuest.c4.rows <- length(BCQuest.number4$Bicluster4$Rows)
BCQuest.c4.cols <- length(BCQuest.number4$Bicluster4$Cols)
BCQuest.c4.coherence <- constantVariance(survey, BCQuest, 4, dimension="both")

BCQuest.number5 <- biclusternumber(BCQuest,5)
BCQuest.c5 <- survey[BCQuest.number5$Bicluster5$Rows,BCQuest.number5$Bicluster5$Cols]
BCQuest.c5.mean <- apply(BCQuest.c5, 2, mean)
BCQuest.c5.sd <- mean(apply(BCQuest.c5, 2, sd))
BCQuest.c5.rows <- length(BCQuest.number5$Bicluster5$Rows)
BCQuest.c5.cols <- length(BCQuest.number5$Bicluster5$Cols)
BCQuest.c5.coherence <- constantVariance(survey, BCQuest, 5, dimension="both")

#plotclust(BCQuest,survey,bicluster=TRUE,legende=FALSE,noC=1,wyld=3,Titel="BCQuest")
# discretized data
#BCQuest.d.number1 <- biclusternumber(BCQuest.d,1)
#BCQuest.d.c1 <- survey.d[BCQuest.d.number1$Bicluster1$Rows,BCQuest.d.number1$Bicluster1$Cols]
#BCQuest.d.c1.sd <- mean(apply(BCQuest.d.c1, 2, sd))
#BCQuest.d.c1.colnames <- colnames(BCQuest.d.c1)
#BCQuest.d.c1.cols <- length(BCQuest.d.number1$Bicluster1$Cols)
#BCQuest.d.c1.rows <- length(BCQuest.d.number1$Bicluster1$Rows)
#BCQuest.d.c1.coherence <- constantVariance(survey, BCQuest.d, 1, dimension="both")
#new.rec <- c(BCQuest.d.c1.sd,BCQuest.d.c1.cols,BCQuest.d.c1.rows,BCQuest.d.c1.coherence)
#df.d <- cbind(df.d,BCQuest.d=new.rec)

```

```

# BCSpectral The Spectral Biclust algorithm

BCSpectral <- biclust(survey, method=BCSpectral(), normalization="log", numberOfEigenvalues=3,minr=2, minc=2,
withinVar=1)

#BCSpectral.d <- biclust(survey.d, method=BCSpectral(), normalization="log", numberOfEigenvalues=3,minr=2, minc=2,
withinVar=1)

# collect attributes of first bicluster

BCSpectral.number1 <- biclusternumber(BCSpectral,1)

BCSpectral.c1 <- survey[BCSpectral.number1$Bicluster1$Rows,BCSpectral.number1$Bicluster1$Cols]

BCSpectral.c1.sd <- mean(apply(BCSpectral.c1, 2, sd))

BCSpectral.c1.colnames <- colnames(BCSpectral.c1)

BCSpectral.c1.cols <- length(BCSpectral.number1$Bicluster1$Cols)

BCSpectral.c1.rows <- length(BCSpectral.number1$Bicluster1$Rows)

BCSpectral.c1.coherence <- constantVariance(survey, BCSpectral, 1, dimension="both")

new.rec <- c(BCSpectral.c1.sd,BCSpectral.c1.cols,BCSpectral.c1.rows,BCSpectral.c1.coherence)

df <- cbind(df,BCSpectral=new.rec)

#plotclust(BCSpectral,survey,bicluster=TRUE,legende=FALSE,noC=1,wyld=3,Titel="BCSpectral")

# discretized data

#BCSpectral.d.number1 <- biclusternumber(BCSpectral.d,1)

#BCSpectral.d.c1 <- survey.d[BCSpectral.d.number1$Bicluster1$Rows,BCSpectral.d.number1$Bicluster1$Cols]

#BCSpectral.d.c1.sd <- mean(apply(BCSpectral.d.c1, 2, sd))

#BCSpectral.d.c1.colnames <- colnames(BCSpectral.d.c1)

#BCSpectral.d.c1.cols <- length(BCSpectral.d.number1$Bicluster1$Cols)

#BCSpectral.d.c1.rows <- length(BCSpectral.d.number1$Bicluster1$Rows)

#BCSpectral.d.c1.coherence <- constantVariance(survey, BCSpectral.d, 1, dimension="both")

#new.rec <- c(BCSpectral.d.c1.sd,BCSpectral.d.c1.cols,BCSpectral.d.c1.rows,BCSpectral.d.c1.coherence)

#df.d <- cbind(df.d,BCSpectral.d=new.rec)


# BCXmotifs The Xmotifs Biclust algorithm

BCXmotifs <- biclust(survey, method=BCXmotifs(), ns=10, nd=10, sd=5, alpha=0.05, number=100)

#BCXmotifs.d <- biclust(survey.d, method=BCXmotifs(), ns=10, nd=10, sd=5, alpha=0.05, number=100)

# collect attributes of first bicluster

BCXmotifs.number1 <- biclusternumber(BCXmotifs,1)

```

```
BCXmotifs.c1 <- survey[BCXmotifs.number1$Bicluster1$Rows,BCXmotifs.number1$Bicluster1$Cols]
BCXmotifs.c1.sd <- mean(apply(BCXmotifs.c1, 2, sd))
BCXmotifs.c1.colnames <- colnames(BCXmotifs.c1)
BCXmotifs.c1.cols <- length(BCXmotifs.number1$Bicluster1$Cols)
BCXmotifs.c1.rows <- length(BCXmotifs.number1$Bicluster1$Rows)
BCXmotifs.c1.coherence <- constantVariance(survey, BCXmotifs, 1, dimension="both")
new.rec <- c(BCXmotifs.c1.sd,BCXmotifs.c1.cols,BCXmotifs.c1.rows,BCXmotifs.c1.coherence)
df <- cbind(df,BCXmotifs=new.rec)

#plotclust(BCXmotifs,survey,bicluster=TRUE,legende=FALSE,noC=1,wylid=3,Titel="BCXmotifs")

# discretized data
BCXmotifs.d <- biclust(survey.d, method=BCXmotifs(), ns=10, nd=10, sd=5, alpha=0.05, number=100)
BCXmotifs.d.number1 <- biclusternumber(BCXmotifs.d,1)
BCXmotifs.d.c1 <- survey.d[BCXmotifs.d.number1$Bicluster1$Rows,BCXmotifs.d.number1$Bicluster1$Cols]
BCXmotifs.d.c1.sd <- mean(apply(BCXmotifs.d.c1, 2, sd))
BCXmotifs.d.c1.colnames <- colnames(BCXmotifs.d.c1)
BCXmotifs.d.c1.cols <- length(BCXmotifs.d.number1$Bicluster1$Cols)
BCXmotifs.d.c1.rows <- length(BCXmotifs.d.number1$Bicluster1$Rows)
BCXmotifs.d.c1.coherence <- constantVariance(survey, BCXmotifs.d, 1, dimension="both")
new.rec <- c(BCXmotifs.d.c1.sd,BCXmotifs.d.c1.cols,BCXmotifs.d.c1.rows,BCXmotifs.d.c1.coherence)
df.d <- cbind(df.d,BCXmotifs.d=new.rec)
```