A tutorial for **blockcluster** R package
Version 4

Parmeet Singh Bhatia[*], Serge Iovleff [†]

Contents

Abstract

blockcluster is a newly developed **R** package for co-clustering of binary, contingency, continuous and categorical data. The core library is written in **C++** and **blockcluster** API acts as a bridge between **C++** core library and **R** statistical computing environment. The package is based on recently proposed [4], [2], [3] latent block models for simultaneous clustering of rows and columns. This tutorial is based on the package version 4.

[*]Siemens, bhatia.parmeet@gmail.com
[†]INRIA-Lille, serge.iovleff@inria.fr

1 Introduction

Cluster analysis is an important tool in a variety of scientific areas such as pattern recognition, information retrieval, micro-array, data mining, and so forth. Although many clustering procedures such as hierarchical clustering, k-means or self-organizing maps, aim to construct an optimal partition of objects or, sometimes, of variables, there are other methods, called block clustering methods, which consider simultaneously the two sets and organize the data into homogeneous blocks. Let $\mathbf{x}$ denotes a $n \times d$ data matrix defined by $\mathbf{x} = \{(x_{ij}); i \in I \text{ and } j \in J\}$, where I is a set of n objects (rows, observations, cases etc) and J is a set of d variables (columns, attributes etc). The basic idea of these methods consists in making permutations of objects and variables in order to draw a correspondence structure on $I \times J$. For illustration, consider Figure 1

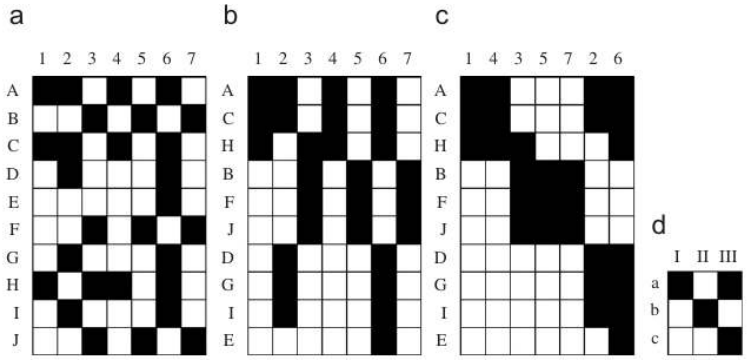


Figure 1: Binary data set (a), data reorganized by a partition on I (b), by partitions on I and J simultaneously (c) and summary matrix (d).

where a binary data set defined on set of $n = 10$ individuals $I = A, B, C, D, E, F, G, H, I, J$ and set of $d = 7$ binary variables $J = 1, 2, 3, 4, 5, 6, 7$ is re-organized into a set of 3×3 clusters by permuting the rows and columns.

Owing to ever increasing importance of Co-clustering in variety of scientific areas, we have recently developed a R package for the same called **blockcluster**. The R package **blockcluster** allows to estimate the parameters of the co-clustering models [[4]] for binary, contingency, continuous and categorical data. This package is unique from the point of view of generative models it implements (latent block models), the used algorithms (BEM, BCEM) and, apart from that, special attention has been given to design the library for handling very huge data sets in reasonable time. The R package is already available on CRAN at `http://cran.r-project.org/web/packages/blockcluster/index.html`.

This aim of this tutorial is to elaborate the usage of R package **blockcluster** and to familiarize its users with its various capabilties. The rest of the article is organized as follows. Section 2 gives various details of the package as well as demonstrate it's usage on similated binary data-set. Section 3 provides two examples with real data-sets.

2 Package details

This package contains two main functions namely **cocluster** and **coclusterStrategy** to perform co-clustering and to set various input parameters respectively. The convenient functions **colclusterBinary**, **coclusterCategorical**, **coclusterContingency** and **coclusterContinuous** are specialized versions of the **cocluster** function. The package also contains two helper functions namely **summary** and **plot** to get the summary of estimated model parameters and to plot the results respectively. We will first go through the details of two main functions.

The helper functions are self-explanatory and I will use them in various examples for better understanding.

2.1 cocluster function

Up to version 3, this is the main function of **blockcluster** package that performs Co-clustering for binary, categorical, contingency and continuous data. The prototype of the function is as follows:

```
cocluster( data, datatype, semisupervised = FALSE
         , rowlabels = numeric(0), collabels = numeric(0)
         , model = character(0), nbcocluster, strategy = coclusterStrategy())
```

The various inputs of **cocluster** functions are as follows:

- **data:** Input data as matrix (or list containing data matrix, numeric vector for row effects and numeric vector column effects in case of contingency data with known row and column effects.)
- **datatype:** This is the type of data which can be "binary", "categorical", "continuous" or "contingency".
- **semisupervised:** Boolean value specifying whether to perform semi-supervised co-clustering or not. Make sure to provide row and/or column labels if specified value is true. The default value is false.
- **rowlabels:** Vector specifying the class of rows. The class number starts from zero. Provide -1 for unknown row class.
- **collabels:** Vector specifying the class of columns. The class number starts from zero. Provide -1 for unknown column class.
- **model:** This is the name of model. The various models that are available in package are given in tables 1, 2, 3 and 4.
- **nbcocluster:** Integer vector specifying the number of row and column clusters respectively.
- **strategy:** This inpu can use to control various input parameters. It can be created using the function **coclusterStrategy** as explained in Section 2.2.

The only mandatory inputs to the function **cocluster** are **data**, **datatype** and **nbcocluster**. The default model for each data-type is the most general model with free row and column proportions and unequal dispersion/variance for each block. Furthermore we have default set of input parameters which works well in most cases which are explained in further details in Section 2.2. The package also comes with OpenMP support (If supported by your Operating system and R). OpenMP will automatically set the number of threads depending on your system.

2.1.1 The coclusterBinary function

The `coclusterBinary` function is a specialization of the `cocluster` function for binary data. The prototype of the function is as follows:

```
coclusterBinary( data, semisupervised = FALSE
    , rowlabels = numeric(0), collabels = numeric(0)
    , model = character(0), nbcocluster, strategy = coclusterStrategy()
    , a=1, b=1)
```

This function has two additional parameters a and b corresponding to the bayesian form of the likelihood function. The default value correspond to the case "no prior". The availables binary models are given in the table 1.

Model	Datatype	Proportions	Dispersion/Variance	Initialization
pik_rhol_epsilonkl	binary	unequal	unequal	CEM
pik_rhol_epsilon	binary	unequal	equal	CEM
pi_rho_epsilonkl	binary	equal	unequal	CEM
pi_rho_epsilon	binary	equal	equal	CEM

Table 1: Binary models available in package **blockcluster**.

2.1.2 The coclusterCategorical function

The `coclusterCategorical` function is a specialization of the `cocluster` function for categorical data. The prototype of the function is as follows:

```
# cocluster for categorical data
coclusterCategorical( data, semisupervised = FALSE
    , rowlabels = numeric(0), collabels = numeric(0)
    , model = character(0), nbcocluster, strategy = coclusterStrategy()
    , a=1, b=1)
```

This function has two additional parameters a and b corresponding to the bayesian form of the likelihood function. The default value correspond to the case "no prior". The availables categorical models are given in the table 2.

Model	Datatype	Proportions	Dispersion/Variance	Initialization
pik_rhol_multi	categorical	unequal	N.A	Random
pi_rho_multi	categorical	equal	N.A	Random

Table 2: Catgorical models available in package **blockcluster**.

2.1.3 The coclusterContinuous function

The `coclusterCategorical` function is a specialization of the `cocluster` function for continuous data. The prototype of the function is as follows:

```
# cocluster for continuous data (Gaussian models)
coclusterContinuous( data, semisupervised = FALSE
    , rowlabels = numeric(0), collabels = numeric(0)
    , model = character(0), nbcocluster, strategy = coclusterStrategy())
```

The availables continuous models are given in the table 3.

Model	Datatype	Proportions	Dispersion/Variance	Initialization
pik_rhol_sigma2kl	continuous	unequal	unequal	CEM
pik_rhol_sigma	continuous	unequal	equal	CEM
pi_rho_sigma2kl	continuous	equal	unequal	CEM
pi_rho_sigma2	continuous	equal	equal	CEM

Table 3: Conatinuous models available in package **blockcluster**.

2.1.4 The coclusterContingency function

The `coclusterContingency` function is a specialization of the `cocluster` function for contingency data. The prototype of the function is as follows:

```
# cocluster for contingency data (Poisson models)
coclusterContingency( data, semisupervised = FALSE
   , rowlabels = numeric(0), collabels = numeric(0)
   , model = character(0), nbcocluster, strategy = coclusterStrategy())
```

The availables contingency models are given in the table 4.

Model	Datatype	Proportions	Dispersion/Variance	Initialization
pik_rhol_unknown	contingency	unequal	N.A	CEM
pi_rho_unknown	contingency	equal	N.A	CEM
pik_rhol_known	contingency	unequal	N.A	Random
pi_rho_known	contingency	equal	N.A	Random

Table 4: Contingency models available in package **blockcluster**.

2.2 coclusterStrategy function

In the package **blockcluster**, we have a function called **coclusterStrategy** which can be used to set the values of various input parameters. The prototype of the function is as follows:

```
coclusterStrategy( algo = "BEM", initmethod = character()
                 , stopcriteria = "Parameter", semisupervised = FALSE
                 , nbiterationsxem = 50,  nbiterationsXEM = 500
                 , nbinititerations = 10, initepsilon = 0.01
                 , nbiterations\_int = 5, epsilon\_int = 0.01
                 , epsilonxem = 1e-04, epsilonXEM = 1e-10, nbtry = 2
                 , nbxem = 5, hyperparam = c(1,1))
```

In the following example, we call the function **coclusterStrategy** without any arguments and then we called the overloaded function **summary** to see default values of various input parameters.

```
> defaultstrategy <- coclusterStrategy()
> summary(defaultstrategy)

**********************************************************************
Algorithm:  BEM
Initialization method(There is no default value):
Stopping Criteria:  Parameter
```

```
Various Iterations
******************
Number of global iterations while running initialization:  10
Number of iterations for internal E-step:  5
Number of EM iterations used during xem:  50
Number of EM iterations used during XEM:  500
Number of xem iterations:  5
Number of tries:  2

Various epsilons
****************
Tolerance value used while initialization:  0.01
Tolerance value for internal E-step:  0.01
Tolerance value used during xem:  1e-04
Tolerance value used during XEM:  1e-10
Hyper-parameters:  1 1
*************************************************************
```

One thing which is worth noting in the summary output (above) is that there is no default value for initialization method. It will be set automatically depending on the type of input model. To set these input parameters, we have to pass appropriate arguments to function **coclusterStrategy** as shown in example below where we set **nbtry**, **nbxem** and **algo** parameters.

```
> newstrategy <- coclusterStrategy(nbtry=5, nbxem=10, algo='BCEM')
```

The **newstrategy** object can then be passed to function **cocluster** to perform Co-clustering using the newly set input parameters. The various input arguments for the function **coclusterStrategy** are as follows:

- **algo:** The valid values for this parameter are "BEM" (Default), "BCEM" and "BSEM" which are respectively Block EM, Block Classification EM and Block Stochastic EM algorithms.
- **stopcriteria:** It specifies the stopping criteria. It can be based on either relative change in parameters value (preferred) or relative change in log-likelihood. Valid criterion values are "Parameter" and "Likelihood". Default criteria is "Parameter".
- **initmethod:** Method to initialize model parameters. The valid values are "cemInitStep", "fuzzyCemInitStep" and "randomInit". For now only one kind of initialization exist for every model currently available in the package. Hence default value for initialization is set according to the model.
- **nbinititerations:** Number of Global iterations used in initialization step. Default value is 10.
- **initepsilon:** Tolerance value used inside initialization. Default value is 1e-2.
- **nbiterations_int:** Number of iterations for internal E step. Default value is 5.
- **epsilon_int:** Tolerance value for relative change in Parameter/likelihood for internal E-step. Default value is 1e-2.

- **nbtry:** Number of tries (XEM steps). Default value is 2.
- **nbxem:** Number of xem steps. Default value is 5.
- **nbiterationsxem:** Number of EM iterations used during xem step. Default value is 50.
- **nbiterationsXEM:** Number of EM iterations used during XEM step. Deafault value is 500.
- **epsilonxem:** Tolerance value used during xem step. Default value is 1e-4.
- **epsilonXEM:** Tolerance value used during XEM step. Default value is 1e-10.

To understand many of the above input parameters, we need to have some basic idea about the algorithms and the way they are run inside package **blockcluster**, which is why there is a separate dedicated section 2.2.1 for the same.

2.2.1 Understanding various input parameters

You might be wondering why there are so many types of iterations and tolerances inside the package. Well, to get some basic understanding about various input parameters, it is important to know a bit about the algorithms. We will not go through full fledged theory of these algorithms here but will provide enough details to make you understand the meaning of all the input parameters. From now on everything will be explained using BEM but it is applicable in same way to BCEM as well as to BSEM algorithm. The BEM algorithm can be defined as follows in laymen language.

1. Run EM algorithm on rows.
2. Run EM algorithm on columns.
3. Iterate between above two steps until convergence.

The following strategy is employed to run various algorithms.

1. Run the BEM Algorithm for **'nbxem'** number of times (with high tolerance and low number of iterations) and keep the best model parameters (based on likelihood) among these runs. We can this step as **'xem'** step.
2. Starting with the best model parameters, run the algorithm again but this time with a low value of epsilon (low tolerance) and a high number of iterations. We can this step as **'XEM'** step.
3. Repeat above two steps for **'nbtry'** number of times and keep the best model estimation.

With this background, the various input parameters are explained as follows.

- **nbxem, nbtry:** As explained above these numbers represents the number of time we run **'xem'** step and **'xem'+'XEM'** step respectively. The tuning of the values of **'nbxem'** and **'nbtry'** need to be done intuitively, and could have a substantial effect on final results. A good way to set these values is to run co-clustering few number of times and check if final log-likelihood is stable. If not, one may need to increase these values. In practice, it is better to increment **'nbxem'** as it could lead to better (stable) results without compromising too much the running time.

- **nbiterationsxem, nbiterationsXEM:** These are number of iterations for BEM algorithm i.e the number of times we run EM on rows and EM on columns. As the name suggests, they are respectively for **'xem'** and **'XEM'** steps.
- **nbiterations_int:** This is the number of iterations for EM algorithm on rows/columns.
- **epsilonxem, epsilonXEM:** These are tolerance values for BEM algorithm during **'xem'** and **'XEM'** step respectively.
- **epsilon_int:** This is the tolerance value for EM algorithm on rows/columns.
- **initepsilon, nbinititerations:** These are the tolerance value and number of iterations respectively used during initialization of model parameters.
- **bayesianform:** Boolean parameter to indicate whether to run algorithms in bayesian settings or not. Algorithms under bayesian settings are currently only implemented for binary and categorical models. Default value is false.
- **hyperparam:** Hyper-parameters ("a" and "b") in case of Bayesian settings. If the assigned value of "a" and "b" is 1, then the algorithms are no different than their original form.

2.3 Model Parameters

When **summary** function is called on the output **cocluster** fuction, it gives the estimated values of various model parameters. The parameters that are common among all the models are row and column mixing proportions. The model parameter for various data-types are as follows.

2.3.1 Binary Models

The parameters $\boldsymbol{\alpha}$ of the underlying distribution of a binary data set is given by the matrix $\mathbf{p} = (p_{k\ell})$ where $p_{k\ell} \in]0,1[\ \forall \ k = 1,\ldots,g$ and $\ell = 1,\ldots,m$ and the probability distribution $f_{k\ell}(x_{ij};\mathbf{p}) = f(x_{ij};p_{k\ell})$ is the Bernoulli distribution

$$f(x_{ij};p_{k\ell}) = (p_{k\ell})^{x_{ij}}(1-p_{k\ell})^{1-x_{ij}}.$$

we re-parameterize the model density as follows:

$$f_{k\ell}(x_{ij};\boldsymbol{\alpha}) = (\varepsilon_{kj})^{|x_{ij}-a_{k\ell}|}(1-\varepsilon_{kj})^{1-|x_{ij}-a_{k\ell}|}$$

where

$$\begin{cases} a_{k\ell} = 0, \ \varepsilon_{k\ell} = p_{k\ell} & \text{if } p_{k\ell} < 0.5 \\ a_{k\ell} = 1, \ \varepsilon_{k\ell} = 1 - p_{k\ell} & \text{if } p_{k\ell} > 0.5. \end{cases}$$

Hence the parameters $p_{k\ell}$ of the Bernoulli mixture model are replaced by the following parameters:

- The binary value $a_{k\ell}$, which acts as the center of the block k,ℓ and which gives, for each block, the most frequent binary value,
- The value $\varepsilon_{k\ell}$ belonging to the set $]0,1/2[$ that characterizes the dispersion of the block k,ℓ and which is, for each block, represents the probability of having a different value than the center.

2.3.2 Categorical Models

The idea behind categorical models is simple extension of binary models for more than 2 modalities. Hence instead of Bernoulli distribution, we used Multinomial (categorical) distribution. Hence the model parameters for each block k, l are $\boldsymbol{\alpha}_{k\ell} = (\boldsymbol{\alpha}^h_{k\ell})_{h=1,..r}$ and $\sum_h \boldsymbol{\alpha}^h_{k\ell} = 1$ where r is the number of modalities.

2.3.3 Continuous Models

In this case, the continuous data is modeled using unidimensional normal distribution. Hence the density for each block is given by:

$$f_{k\ell}(x_{ij}; \boldsymbol{\alpha}) = \frac{1}{\sqrt{2\pi\sigma^2_{k\ell}}} \exp -\{\frac{1}{2\sigma^2_{k\ell}}(x_{ij} - \mu_{k\ell})^2\}$$

The parameters of the model are $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_{11}, \ldots, \boldsymbol{\alpha}_{gm})$ where $\boldsymbol{\alpha}_{k\ell} = (\mu_{k\ell}, \sigma^2_{k\ell})$ i.e the mean and variance of block k, l.

2.3.4 Contingency Models

In this case, it is assumed that for each block k, ℓ, the values x_{ij} are distributed according to Poisson distribution $\mathcal{P}(\mu_i \nu_j \gamma_{k\ell})$ where the Poisson parameter is split into μ_i and ν_j the effects of the row i and the column j respectively and $\gamma_{k\ell}$ the effect of the block $k\ell$. Then, we have

$$f_{k\ell}(x_{ij}; \boldsymbol{\alpha}) = \frac{e^{-\mu_i \nu_j \gamma_{k\ell}} (\mu_i \nu_j \gamma_{k\ell})^{x_{ij}}}{x_{ij}!}$$

where $\boldsymbol{\alpha} = (\boldsymbol{\mu}, \boldsymbol{\nu}, \boldsymbol{\gamma})$ with $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_n)$, $\boldsymbol{\nu} = (\boldsymbol{\nu}_1, \ldots, \nu_d)$ and $\boldsymbol{\gamma} = (\gamma_{11}, \ldots, \gamma_{gm})$. The row and column effects are either provided by the user for models **pik_rhol_known** and **pi_rho_known** or estimated by the package itself for models **pik_rhol_unknown** and **pi_rho_unknown**.

2.4 Example using simulated Binary dataset

The various parameters used to simulate this binary data-set are given in Table 5. The class mean and dispersion are respectively represented by $\mathbf{a}$ and $\boldsymbol{\epsilon}$ whereas $\boldsymbol{\pi}$ and $\boldsymbol{\rho}$ represents row and column proportions respectively. The data consist of 1000 rows (samples) and 100 columns (variables) with two clusters on rows and three clusters on columns. The following R commands shows how to load the library, process the data and visualize/summarize results using **blockcluster**.

$\mathbf{a}$, $\boldsymbol{\epsilon}$	0, 0.1	0, 0.3	1, 0.1
	1, 0.3	1, 0.2	0, 0.1

$\boldsymbol{\pi}$	.6	.4

$\boldsymbol{\rho}$	.3	.3	.4

Table 5: Parameters for simulation of binary data.

```
> data("binarydata")
> out<-coclusterBinary(binarydata, nbcocluster=c(2,3))

Co-Clustering successfully terminated!

> summary(out)
```

```
*****************************************************************
Model Family : Bernoulli Latent block model
Model Name : pik_rhol_epsilonkl
Co-Clustering Type : Unsupervised
ICL value:  -45557.07

Model Parameters..

Class Mean:
      [,1]  [,2]  [,3]
[1,]  TRUE FALSE FALSE
[2,] FALSE FALSE  TRUE

Class Dispersion:
          [,1]       [,2]      [,3]
[1,] 0.1011803 0.09798014 0.3022391
[2,] 0.1006314 0.30176927 0.2003679

Row proportions:  0.618 0.382
Column proportions:  0.34 0.29 0.37
Pseudo-likelihood:  -0.4552043
*****************************************************************
```

Note that you also get the Integrated complete likelihood (ICL) value in case binary and categorical models. This value can be used for model selection. The following **R** command is used to plot the original and co-clustered data (Figure 2(a)) with default value of **asp** which is 0 (FALSE). When **asp** is FALSE, R graphics will optimize the output figure for the display, hence the original aspect ratio may not be conserved. To conserve the original aspect ratio, set the value of **asp** as 1 or TRUE.

```
> plot(out, asp = 0)
```

To Plot various block distributions (Figure 2(b)), the following **R** command is used with **type** argument of overloaded **plot** function set to 'distribution' (**type** is 'cocluster' by default which plots the original and Co-clustered data as shown in (Figure 2(a))).

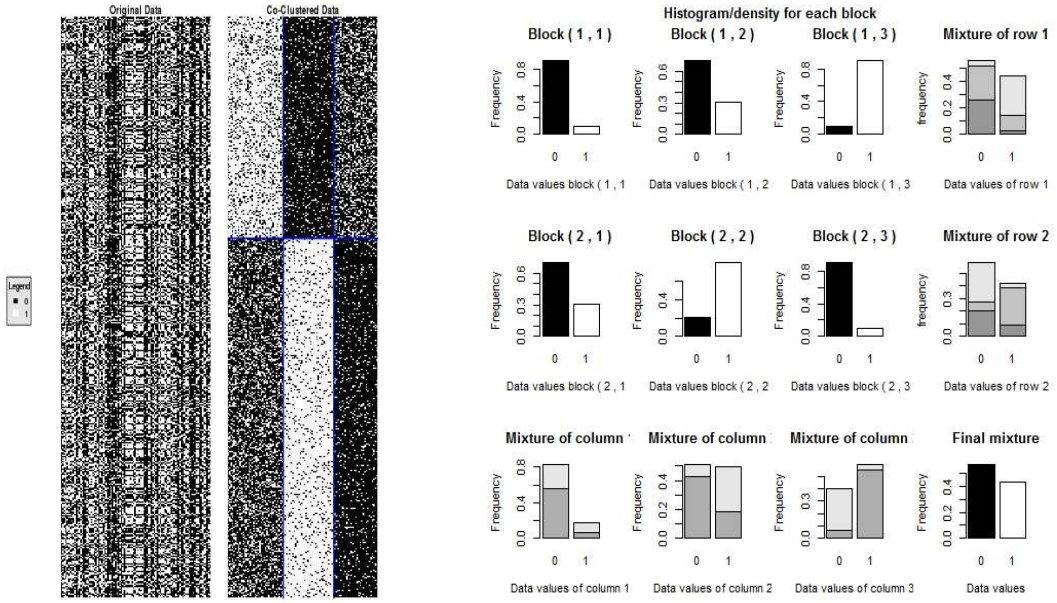
```
> plot(out, type = 'distribution')
```

3 Examples with real datasets

This section demonstrates the applicability of package on real data. Two examples are used: one for Image segmentation and other for document (co-)clustering.

3.1 Image segmentation

Automatic image segmentation is an important technique and have numerous application especially in fields of Medical imaging. Here I present an interesting application of co-clustering (as pre-processing step) for segmenting object(s) in image. I assume that the object pixels follows Gaussian distribution. Hence I run the **blockcluster** package with Gaussian Family model **pik_rhol_sigma2kl** on image shown in Figure 3. It can be clearly seen that the image got nicely segmented into snake and insect in two different blocks.



(a) (b)

Figure 2: Original and co-clustered binary data (a), and distributions for each block along with various mixture densities (b).

3.2 Document clustering

Document clustering is yet another data mining technique where co-clustering seems to be very useful. Here we run our package on one of the datasets being used in [1] which is publicly available at `ftp://ftp.cs.cornell.edu/pub/smart`. We mix Medline (1033 medical abstracts) and Cranfield (1398 aeronautical abstracts) making a total of 2431 documents. Furthermore, we used all the words (excluding stop words) as features making a total of 9275 unique words. The data matrix consist of words on the rows and documents on the columns with each entry giving the term frequency, that is the number of occurrences of corresponding word in corresponding document. I assume that the term frequency follows Poisson distribution. Hence we can apply the model **pik_rhol_unknown** available in our package for contingency (Poisson Family) datasets with unknown row and column effects. Table 6 shows the confusion matrix and compare our results with classical bipartite spectral graph partitioning algorithm of [[1]] where we have obtained 100 percent correct classification. Figure 4 depicts the 2×2 checkerboard pattern in the data matrix, hence confirming the more frequent occurrence of particular set of words in one document and vice-versa. Please note that the data matrix images are extremely sparse (data points almost invisible) and have been processed using simple image processing tools for visualization purpose only.

	Medline	Cranfield
Medline	1026	0
Cranfield	7	1400

(a)

	Medline	Cranfield
Medline	1033	0
Cranfield	0	1398

(b)

Table 6: Confusion Matrix: Results reported in [1] (a), and Results using **blockcluster** (b). The difference in number of Cranfield documents is because we made use of the already available data extracted from the documents and there are two less documents data in the same.

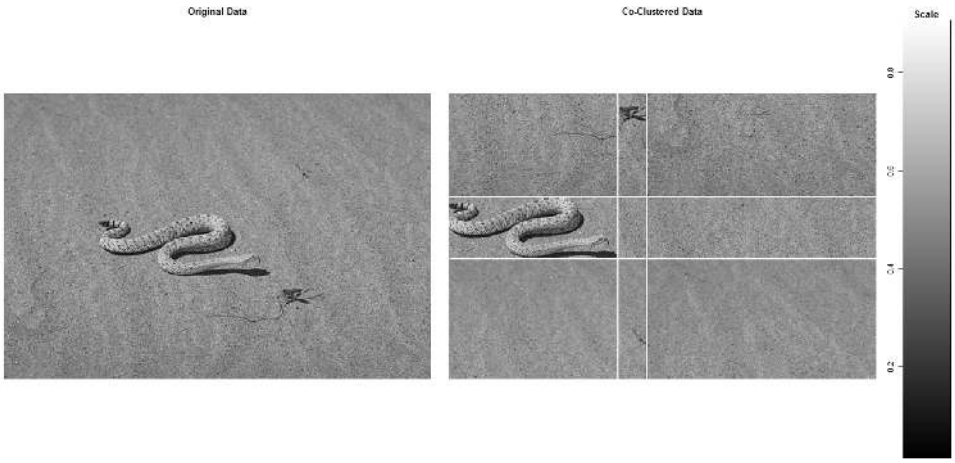


Figure 3: Original and co-clustered (segmented) image.

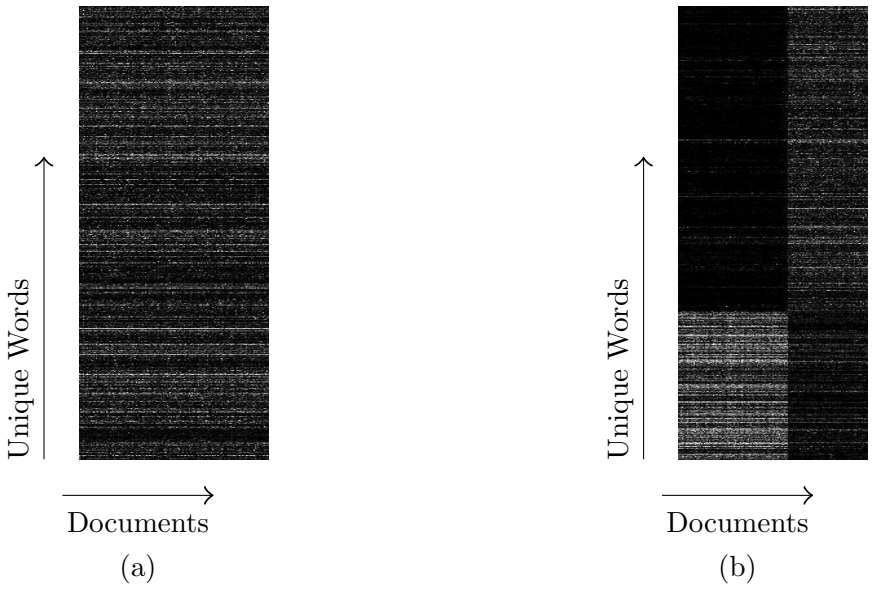


Figure 4: Original data matrix with words on rows and documents on columns (a), and checkerboard pattern in words by documents matrix obtained after performing co-clustering (b).

4 Remarks

This tutorial gives a brief introduction about the **blockcluster** R package. It demonstrates the use of package using Binary data-set but the package can be used in similar fashion for other types of data namely **Contingency**, **Continuous** and **Categorical**. Please note that this tutorial is based on version 3. If you have any questions,suggestions or remarks, do not hesitate to put it on public forum at `https://gforge.inria.fr/forum/forum.php?forum_id=11190&group_id=3679`.

References

[1] Inderjit S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge*

discovery and data mining, KDD '01, pages 269–274, New York, NY, USA, 2001. ACM.

[2] G. Govaert and M. Nadif. Block clustering with bernoulli mixture models: Comparison of different approaches. *Computational Statistics & Data Analysis*, 52(6):3233–3245, 2008.

[3] G. Govaert and M. Nadif. Latent block model for contingency table. *Communications in Statistics - Theory and Methods*, 39(3):416–425, 2010.

[4] Gérard Govaert and Mohamed Nadif. Clustering with block mixture models. *Pattern Recognition*, 36(2):463 – 473, 2003.