TEKNILLINEN KORKEAKOULU
Teknillisen fysiikan
koulutusohjelma

ERIKOISTYÖ
T-61.5900 Informaatiotekniikan erikoistyö
5. heinäkuuta 2007

Sampling Real Matrices with Given Margins

Markus Ojala
63323T

Contents

1 Introduction

One of the most important problem in data mining is how to test the significance of a result. In this special assignment, we study methods for assessing the results of data mining algorithms which measure the structure of real valued matrices.

Traditional statistical data analysis is suitable for testing the significance of individual findings, but testing, for example, the significance of a clustering is difficult. One solution to this is to compare the data mining results on random samples to the original result. The problem is how we can generate proper random samples. In the case of assessing the structure of a real valued matrix, we would like the randomized matrices to share some statistics with the original matrix. Therefore, we have decided to conserve the sums and variances of rows and columns, which explain the most of the variation in the matrix. We call these the *margins* of the matrix.

The problem has been studied extensively in the cases of 0–1 matrices and contingency tables where only the sums of rows and columns are fixed [3, 6]. In these, the significance testing is based on random samples which are produced by Markov chain approach using swaps or addition masks as transitions. There exist also more traditional permutation tests and methods preserving only row or column margins [7]. Our approach is based on the 0–1 analysis in [6] whose basic ideas we extend into the case of real valued matrices.

The report is organized as follows. In Section 2 we present the usage of random sample sets for significance testing and motivate the importance of margins. In Section 3 we discuss classical data mining algorithms for measuring the structuredness of a matrix. Markov chains and their properties are introduced in Section 4. After that, in Section 5 we discuss the aspects of the problem in more detail and give an error measure for a randomized matrix. The Section 6 contains descriptions of six different methods for sampling random matrices. Finally, in Section 7 we conduct experiments for comparing the methods, and in Section 8 we give some concluding remarks.

2 Using random sample sets

In this section we motivate the usefulness of random sample sets of matrices where row and column sums and variances are kept almost constant. First, methods for assessing statistical significance of data mining results are discussed. After that we give an illustrative example of the effect and importance of margins.

2.1 Testing significance

Let D_0 be our original m-by-n real data matrix. We want to test the significance of data mining results on D_0 using random sample set. Assume that the results of the data mining algorithm $\mathcal{A}$ we are using can be described by one real number, i.e., the algorithm $\mathcal{A}$ is a function $\mathcal{A} : \mathbb{R}^{m \times n} \to \mathbb{R}$. For instance, it can be a measure of structuredness of D_0 obtained by clustering error. Methods for evaluating structuredness are discussed more in detail in the next section.

Let $\mathscr{D} = \{D_1, D_2, \dots, D_k\}$ be a random sample set of m-by-n real matrices with some properties similar to the original data D_0. Ideally in our case, $\mathscr{D}$ will be sampled uniformly and independently from all the matrices having almost the same row and column sums and variances as the original matrix D_0. Let $X_t = \mathcal{A}(D_t)$ be the result of the algorithm $\mathcal{A}$ on the matrix D_t.

The idea of significance testing is now to compare the original result X_0 to the results X_t on the random samples $D_t \in \mathscr{D}$. If X_0 differs substantially from the results on random samples then we can conclude that the data mining result on D_0 is significant and does not just depend on the margins. The converse that the result depends mainly on the margins could be interesting as well, but assessing it would need other methods.

A classical measure for the significance of a result is *p-value*, which is the probability of obtaining a result as extreme as the given one assuming it was produced by chance alone. It can be approximated by the *empirical p-value* which is in our case

$$\frac{\min\left(|\{t \mid X_t \leq X_0\}|, |\{t \mid X_t \geq X_0\}|\right) + 1}{k + 1}. \tag{1}$$

This is the two-sided version of the test. If the empirical p-value is small, e.g., less than 1%, we can conclude that $X_0 = \mathcal{A}(D_0)$ is significant with $p = 0.01$. [5]

The dissimilarity of the result X_0 can be evaluated as well by *Z-score*, which measures the normalized deviation from the average value. Let $\hat{\mu}$ be the empirical mean and $\hat{\sigma}^2$ the empirical variance of $\mathcal{A}(D)$, $D \in \mathscr{D}$. Then the Z-score is

$$Z = \frac{X_0 - \hat{\mu}}{\hat{\sigma}}. \tag{2}$$

If the Z-score differs substantially from zero, the result X_0 is significant.

2.2 Maintaining margins

Randomization is a generally used technique for testing significance. The problem is what kind of randomness we want and how we can generate it. As stated earlier, most of the current randomization methods for real matrices are based on only swapping columns and rows or preserving only, for example, column margins.

0.5	0.5	0.6	0.3	0.5
0.6	0.7	0.8	0.6	0.6
0.1	0.2	0.3	0.0	0.1
0.5	0.4	0.5	0.4	0.5
0.7	0.6	0.5	0.8	0.6
0.9	0.9	1.0	0.7	0.9
0.3	0.1	0.2	0.2	0.0

Data D_1

0.5	0.5	0.2	0.8	0.1
0.6	0.7	0.1	0.7	0.4
0.1	0.2	0.8	0.5	0.3
0.5	0.4	1.0	0.6	0.9
0.7	0.6	0.6	0.8	0.0
0.9	0.9	0.2	0.1	0.8
0.3	0.1	0.4	0.6	1.0

Data D_2

Figure 1: Examples of two real data matrices.

These methods are suitable for significance testing in various real applications, but they do have limitations. We consider an example where maintaining both row and column sums and variances can be useful in interpreting data mining results.

Two real data matrices D_1 and D_2 are presented in Figure 1. They share the first and the second column, and the correlation between these columns is high, 0.92. However, in the matrix D_1 values in each row are close to each other, whereas in the matrix D_2 the third, fourth and fifth columns contain just independent random numbers. If the significance test would consider only the first two columns it couldn't separate the data matrices. It seems plausible that the high correlation between the first and the second column in the data matrix D_1 is only due to the margins and not due to some interesting structure, as it might be the case with the matrix D_2.

To test this observation, we generated random sample sets $\mathscr{D}_1$ and $\mathscr{D}_2$, using method discussed later (*SeekOptimalValue*), with 10 000 independent random samples in each such that the row and column sums and variances of the samples differed at most couple percents from the corresponding original margins of the matrices D_1 and D_2. When we calculated the correlations between the first and second column in the random samples, we found that in the sample set $\mathscr{D}_1$ the minimum correlation is 0.40, maximum is 1.00, average is 0.83 and standard deviation is 0.10, whereas the corresponding values in the sample set $\mathscr{D}_2$ are -0.97, 0.98, -0.03 and 0.39, respectively. In the sample set $\mathscr{D}_1$ there were 1910 samples with higher correlation than in the original data D_1 giving an empirical p-value of 0.1911. In the sample set $\mathscr{D}_2$ there were 16 samples with higher correlation giving an empirical p-value of 0.0017. Thus we may conclude that the similarity between the first and second column in the matrix D_2 is likely not to depend on the margins.

As we noticed the structure of the whole data matrix can have a remarkable effect on the significance of the data mining results. Thus taking the margins into account can in some applications be crucial. Whether to maintain the margins in significance testing or not has to be always considered carefully.

3 Evaluating structuredness

Since we are trying to omit the effect of row and column margins of a matrix, we are interested in data mining algorithms that measure the structuredness of the matrix. In the following, we discuss two classical data mining algorithms for measuring the structuredness: k-means clustering and hierarchical clustering. Only hierarchical clustering will be used in experiments. There exist also various other data mining algorithms and methods where the framework is applicable, e.g., singular values, correlations and so on.

3.1 Clustering

Assume that X is a set of points in $\mathbb{R}^d$. Clustering of X means that X is partitioned into subsets, called *clusters*, such that points in a same cluster are somehow similar. Similarity between the points can be measured, e.g., by Euclidian distance. In the case of a real matrix, either rows or columns can be used as data points. The quality of the clustering gives a measure for the structuredness of the matrix. There exists also so called *biclustering* algorithms where rows and columns are clustered simultaneously [10].

Two different clustering methods are introduced: k-means and hierarchical clustering. K-means iteratively updates the clustering by moving points between the clusters whereas hierarchical clustering combines current clusters to make new ones. For more information about clustering algorithms see [2, 8].

3.1.1 K-means

K-means is a classical clustering algorithm [9]. It tries to find centers $\mu_i \in \mathbb{R}^d$ for clusters C_i such that the error function

$$E = \sum_{i=1}^{k} \sum_{x \in C_i} |x - \mu_i|^2 \tag{3}$$

is minimized, where k is a predefined number of clusters. Usually only a local optimum is found. Quality can be improved by repeating the algorithm several times and keeping the best clustering found. Clustering error (3) can be used as a feature of clustering structure.

The algorithm starts by initializing the centers μ_i randomly. Then the next two steps are repeated until the process has converged:

1. Associate each point $x \in X$ with the cluster C_i whose center μ_i is nearest.
2. Update the centers μ_i to be the new average of the points x associated with the corresponding cluster C_i.

3.1.2 Hierarchical clustering

An agglomerative hierarchical clustering algorithm builds a hierarchy of clusters. The algorithm starts by associating a new cluster with each point $x \in X$. At each step the two clusters which are nearest to each other are sought and they are combined together. Different types of closeness measures can be used, e.g., minimum, mean, median or maximum distance between the clusters. The process forms a binary tree where nodes corresponds to clusters and edges to combination relations. For each node in the tree a height can be given according to the distance of its two subcluster parts. The formed tree is called a *dendrogram*.

A clustering can be obtained by cutting the tree at some height. Then similar measure as in K-means in Equation (3) can be used as a feature of clustering structure. Another measure could be the sum of the heights of the nodes of the dendrogram. By this way, the number of clusters doesn't have to be considered.

4 Markov chains

In this section, we introduce a stochastic concept *Markov Chain* and discuss algorithms for sampling from probability distributions. [1,4]

4.1 Definitions and properties

A Markov chain is a discrete-time stochastic process where the next state depends only on the current state. More formally, a sequence of random variables $X_1, X_2, \ldots$ is called a *Markov chain* if it fulfills the *Markov property*

$$p(X_{n+1} = x \mid X_n = x_n, \ldots, X_1 = x_1) = p(X_{n+1} = x \mid X_n = x_n). \tag{4}$$

Usually we are only interested in *time-homogeneous* Markov chains where the transition probabilities do not depend on time, i.e.,

$$p(X_{n+1} = x \mid X_n = y) = p(X_n = x \mid X_{n-1} = y) \tag{5}$$

for all n. In case of finite state space S we use the *transition probability matrix* P whose element P_{ij} is the probability of moving from state i to state j. A Markov chain is said to be *connected* (or *irreducible*) if every state is reachable in finite number of steps from every other states. A connected chain is called *aperiodic* (or *acyclic*) if for all two states i and j there exist a time n_{ij} such that $P_{ij}^{(n)} > 0$ for all $n \geq n_{ij}$, where $P_{ij}^{(n)}$ is the probability to be in state j after n steps when starting from state i. The *stationary distribution* of a time-homogeneous Markov chain is the distribution where the process converges. The probability vector π of the stationary distribution fulfills

$$\pi^T = \pi^T P. \tag{6}$$

The definitions can easily be generalized also to the case of continuous state space.

An important special case is a *reversible* Markov chain for which all the states $i, j \in S$ fulfill $P_{ij} = P_{ji}$, that is, $P = P^T$. For stationary distributions of reversible Markov chains we have the following theorem:

Theorem 1. *The stationary distribution of a connected, aperiodic, reversible Markov chain is a uniform distribution.*

Proof. Connectedness and aperiodicity of the Markov chain guarantees that the process converges and all states have a positive final probability. As the transition matrix P contains probabilities, $P\mathbf{1} = \mathbf{1}$, where $\mathbf{1}$ is a vector of ones. Since the chain is reversible, $P = P^T$, and we get

$$\mathbf{1}^T P = \mathbf{1}^T, \tag{7}$$

thus the stationary distribution is uniform. □

4.2 Markov chain Monte Carlo

Markov chain Monte Carlo (MCMC) is a general concept for methods for sampling from probability distributions. MCMC is based on constructing a Markov chain with the desired distribution as its stationary distribution. The states of the Markov chain are then used as a sample set from the desired distribution.

One of the most important properties of the MCMC method is the *mixing time* (or *burn-in time*). It describes the number of steps after which the state distribution of the Markov chain has approximately reached the stationary distribution. Only samples obtained after the mixing time of the chain should be accepted as random samples from the stationary distribution.

The mixing time is usually hard to evaluate theoretically. In practice, we can use some distance measure to approximate the mixing time, i.e., when the distance between the starting state and the current state has converged, we can assume that the distribution has converged. Often it is just enough to be sure that the chain is *functionally mixed* which means that the distribution of the values of some relevant function of the samples has converged.

4.3 Metropolis-Hastings

Metropolis-Hastings algorithm is one of the most used MCMC methods. It's a rejection sampling algorithm which uses *proposal density* $Q(y|x)$ that gives the proposal probability of the new state y given the current state x. It's assumed that sampling from the proposal density $Q(\cdot|x)$ is easy, whereas the desired probability $P(x)$ can be calculated up to a constant factor.

At each step a new proposal y is drawn from the distribution $Q(\cdot|x)$. It's accepted as the new state if u sampled randomly from uniform distribution $U(0,1)$ fulfills

$$u < \frac{P(y)Q(x|y)}{P(x)Q(y|x)}, \tag{8}$$

otherwise the chain stays in the current state x. If the chain is connected and aperiodic, the stationary distribution is equal to P.

The proposal distribution Q has a huge impact on the mixing time. It should be as global as possible while allowing a high acceptance rate in Equation (8). The optimal acceptance rate under some reasonable assumptions is around 25% [4]. However, the best case would be if $Q(\cdot|x)$ equalled P for all x. Then the acceptance rate would be 100%. Often symmetric proposal distribution is used, i.e., $Q(y|x) = Q(x|y)$. This is the original version of the method by Metropolis [11].

5 Problem definition

In this section, we first formulate variants of the problem, and then we define some error measures for evaluating the quality of random matrices.

5.1 Problem variants

In introduction, we formulated the problem as randomly sampling real matrices with given row and column sums and variances. As we are interested in real matrices we cannot conserve the margins exactly. This raises questions like: How large an error can we tolerate? How should we compare the magnitude of errors in rows with errors in columns or errors in sums with errors in variances? What is "random" in our case? Should we try to conserve something else as well?

Answers to these questions depend on the application. The methods developed and explained later will all have some advantages and drawbacks. The method to be used has to be selected according to what we consider important in the application.

The combined error quantity we are measuring should give equal importance to all error types. This can be obtained by scaling them appropriately. Instead of preserving sums and variances, we can preserve sums and square sums, since variance can be expressed with these. In this way, we get an easier expression for the error. In addition to preserving sums and square sums, we might as well be interested in preserving higher moments, i.e., higher power sums.

In many applications the distribution of the values of the data matrix is important. We would like our random matrix to contain values similar to the original matrix. An easy solution to this is to keep the original values of the matrix, and only to permute them randomly. For some applications this might not be random enough. Another solution is to restrict the values in some specific interval, e.g., in $[0, 1]$, if the values of the original matrix vary in this interval.

By random we mean that the sample matrices should be independently drawn from a fixed distribution which depends on the method. All matrices with the same error should have equal probability for outcome, whereas less erroneous matrices should be more probable than erroneous ones. The methods to be introduced emphasize these three aspects differently since accomplishing all of the objectives is troublesome.

5.2 Measuring the error

Let A be the original m-by-n real matrix whose margins we want to maintain. Let $\hat{A}$ be a randomized m-by-n real matrix whose margins we would like to be close to the corresponding margins of A. Let r_i be the sum of the values in the ith row and c_j the sum of the values in the jth column of the matrix A. Let R_i and C_j be the corresponding sums of squares of the values in row i and column j, that is,

$$r_i = \sum_{j=1}^{n} A_{ij}, \quad c_j = \sum_{i=1}^{m} A_{ij}, \quad R_i = \sum_{j=1}^{n} A_{ij}^2, \quad C_j = \sum_{i=1}^{m} A_{ij}^2. \tag{9}$$

Let $\hat{r}_i$, $\hat{c}_j$, $\hat{R}_i$ and $\hat{C}_j$ be the corresponding values of the randomized matrix $\hat{A}$. Now, let E_{r_i}, E_{c_j}, E_{R_i}, E_{C_j} be the row sum, column sum, row square sum and column square sum errors correspondingly, i.e.,

$$E_{r_i} = |r_i - \hat{r}_i|, \quad E_{c_j} = |c_j - \hat{c}_j|, \quad E_{R_i} = |R_i - \hat{R}_i|, \quad E_{C_j} = |C_j - \hat{C}_j|. \tag{10}$$

Now we are ready to form a general error function which combines all the four types of error and which can be varied by weights. Let w_r and w_s be row and square weights, correspondingly. Let p_1 and p_2 be powers for sums and square sums. The general error function reads

$$E(A, \hat{A}) = w_r \sum_{i=1}^{m} \left(E_{r_i}^{p_1} + w_s E_{R_i}^{p_2} \right) + \sum_{j=1}^{n} \left(E_{c_j}^{p_1} + w_s E_{C_j}^{p_2} \right). \tag{11}$$

$$\begin{array}{c|ccccc|}
 & & c_1 & & c_2 & \\ \hline
 & & \vdots & & \vdots & \\
r_1 & \dots & a & \dots & b & \dots \\
 & & \vdots & & \vdots & \\
r_2 & \dots & b' & \dots & a' & \dots \\
 & & \vdots & & \vdots & \\ \hline
\end{array}
\quad \Longleftrightarrow \quad
\begin{array}{c|ccccc|}
 & & c_1 & & c_2 & \\ \hline
 & & \vdots & & \vdots & \\
r_1 & \dots & b' & \dots & a & \dots \\
 & & \vdots & & \vdots & \\
r_2 & \dots & a' & \dots & b & \dots \\
 & & \vdots & & \vdots & \\ \hline
\end{array}$$

Figure 2: An example of a swap rotation.

This measures the quality of the random matrix $\hat{A}$ compared to the original matrix A. The row and square weights w_r and w_s and powers p_1 and p_2 can be used to affect the importance of each error type. In our experiments, we use parameter values $w_r = m/n$, $w_s = 1$ and $p_1 = p_2 = 2$, which give equal amount of importance to each of the row and column sum and square sum errors.

6 Sampling matrices with given margins

Next, we introduce various algorithms for solving the problem of sampling random matrices with given margins. There are mainly two types of methods: some preserve the original values of the matrix and change the matrix only by swapping the elements, while the others change the values of the matrix elements directly. In the former case, the distribution of values in a randomized matrix is clearly the same as in the original matrix, while this is not the case in the latter algorithms. All the methods form a Markov process since the next state depends only on the current state. A comparison of performance of all methods is presented in the experiments.

6.1 Generation via swap randomization

All methods presented in this subsection are based on a simple swap operation shown in Figure 2. At each step we choose randomly four elements a, b, a' and b' from the current matrix such that they are at the intersection points of two rows r_1, r_2 and two columns c_1, c_2. The new matrix is produced by rotating those four elements as shown in Figure 2, while keeping the other elements of the original matrix untouched. This is repeated until the matrix has mixed.

The difference in the methods is mainly what restrictions these four elements have and what is the starting matrix for the process. If a and a' would be equal as well as b and b' then the swap rotation wouldn't change the row and column sums and variances and neither any power sums. This is the basic idea behind the methods based on swaps.

In addition, there exist three other types of swaps which we could use: rotation counterclockwise and swaps of those four elements between rows or between columns. Generally, these could be used randomly by turns. Nevertheless, we are only using the swap rotation depicted in Figure 2, because the "counterclockwise" rotation can actually be realized with "clockwise" rotation by selecting rows or columns in different order, the rotation preserves one original value in each row and column which is not the case with row or column swaps, and there are more possible states with rotation than with row or column swaps.

6.1.1 Epsilon approach

Next, we present the most basic algorithm based on the idea of swaps. We start from the original matrix A and repeat the swap rotation step. We accept the rotation of four elements a, b, a' and b' if

$$|a - a'| < \varepsilon \quad \text{and} \quad |b - b'| < \varepsilon, \tag{12}$$

where $\varepsilon > 0$ is a parameter of the algorithm. Thus a swap maintains the row and column sums and variances approximately when ε is small. A pseudocode of the method is shown in Algorithm 1.

Algorithm 1 SwapEpsilon

```
Input: Matrix $A$, number of attempts $k$, closeness threshold $\varepsilon > 0$
1: for $i \leftarrow 1, k$ do
2:     Pick $r_1$ and $c_1$ randomly
3:     Pick $r_2$ and $c_2$ randomly with $|A_{r_1c_1} - A_{r_2c_2}| < \varepsilon$
4:     if $r_1 \neq r_2$ and $c_1 \neq c_2$ and $|A_{r_1c_2} - A_{r_2c_1}| < \varepsilon$ then
5:         $A \leftarrow$ Swap($A$, $r_1$, $r_2$, $c_1$, $c_2$ )
6:     end if
7: end for
8: return $A$
```

The auxiliary method *Swap* does the rotation of the four elements at the intersection points of rows r_1, r_2 and columns c_1, c_2 as was shown in Figure 2. The rows and columns r_1, r_2, c_1 and c_2 could all be selected at the same time, after which the condition (12) could be checked. By selecting them in steps, as is done in the Algorithm 1, we can make the acceptance rate higher. The picking of row r_2 and column c_2 in line 3 can be done in constant time if precalculation of the candidates for each element is done beforehand.

The precalculation can be done efficiently by sorting the values of the matrix and calculating the starting and ending indeces of the intervals of neighbours for each element from the sorted array of values. In addition, we need to keep track on where each element is located in the matrix. The precalculation can be done in time $O(L \log(L))$ and the auxiliary structures needs space $O(L)$, where $L = mn$ is the number of elements in the matrix.

The acceptance rate in line 4 can be approximated from below for a random matrix. Let d be the length of the range of the values in the matrix. The worst case is if the values are uniformly distributed in the range. The probability that the distance between the elements $A_{r_1c_2}$ and $A_{r_2c_1}$ is less than ε is then around $\frac{2\varepsilon}{d}$ which is thus a lower bound for the acceptance probability.

The *SwapEpsilon* algorithm has a simple idea how to conserve the margins. Between two successive steps the margins won't change more than $O(\varepsilon)$ and actually the expectation values for the margins are the previous values. But since the process has to be repeated several times, at least comparable to $O(L)$, the error can become huge. The parameter ε has a direct impact on the error, but decreasing it also decreases the acceptance rate and prevents the matrix from mixing well.

6.1.2 Discretized approach

The problem with the epsilon approach of section 6.1.1 is that the error grows as steps are taken. Another simple approach is to discretize the whole matrix into a predefined number N of classes before starting the swapping. This means that all the values of the matrix are replaced with the closest of the N representative values for the classes. After that we can insist a and a' as well as b and b' to be equal in the swap. The pseudocode of this approach is presented in Algorithm 2.

Algorithm 2 SwapDiscretized

```
Input: Matrix $A$, number of attempts $k$, number of classes $N$
1: $A \leftarrow$ Discretize($A$, $N$)
2: for $i \leftarrow 1, k$ do
3:     Pick $r_1$ and $c_1$ randomly
4:     Pick $r_2$ and $c_2$ randomly with $A_{r_1c_1} = A_{r_2c_2}$
5:     if $r_1 \neq r_2$ and $c_1 \neq c_2$ and $A_{r_1c_2} = A_{r_2c_1}$ then
6:         $A \leftarrow$ Swap($A$, $r_1$, $r_2$, $c_1$, $c_2$ )
7:     end if
8: end for
9: return $A$
```

The only error is made in the discretizing process which is done in the auxiliary method *Discretize*. It means that the sums and variances as well as all the other power sums of rows and columns remain the same after the discretizing process. The discretizing itself can be done in various ways. The simplest way is to divide the range $[d_0, d_1]$ of the values in the matrix into smaller, equal sized,

$$\boxed{\begin{matrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{matrix}} \quad \nrightarrow \quad \boxed{\begin{matrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{matrix}}$$

Figure 3: A counterexample of connectedness.

distinct intervals $[d_0, d_0 + d)$, $[d_0 + d, d_0 + 2d)$, ..., $[d_0 + (N-1)d, d_0 + Nd]$, where $d = \frac{d_1 - d_0}{N}$. The discretized values would then be the mid-points of the intervals. Slightly smaller error is obtained when the discretized values are the average of the values in the same interval class. This discretizing function is used in experiments in Section 7.

The lower bound of the acceptance rate of *SwapDiscretized* is similar to the lower bound of the acceptance rate of the algorithm *SwapEpsilon*. Let n_i be the number of elements in class i, thus $\sum_{i=1}^{N} n_i = L$. If $A_{r_1 c_2}$ belongs to class i then the probability of acceptance is $\frac{n_i - 1}{L-1} \approx \frac{n_i}{L}$ for a random matrix. Using Chebyshev's sum inequality or Cauchy-Schwarz inequality we obtain the following approximate lower bound for the acceptance probability:

$$\sum_{i=1}^{N} \left(\frac{n_i}{L}\right)^2 \geq N \left(\frac{\sum_{i=1}^{N} \frac{n_i}{L}}{N}\right)^2 = \frac{1}{N}. \tag{13}$$

The algorithm cannot attain all permutations of values with the same distribution of classes in each row and column than in the original matrix. Consider the counterexample shown in Figure 3 where the two matrices have the same number of ones, twos and threes in each row and column but in neither of them there exist four elements which could be swapped. Thus they cannot be transformed to each other. The given counterexample can be generalized directly to all matrices with either odd number of rows or columns.

Nevertheless, the *SwapDiscretized* algorithm is a simple method for maintaining the margins although it might not randomize the matrix enough. As in the case of epsilon, the selection of parameter N is a compromise between mixing and error.

6.1.3 Uniform sampling

We haven't yet considered the stationary distributions of the previous two methods. In the discretized case, it is easy to require that all possible resulting matrices should have equal probability for outcome since they share the same amount of error. With the algorithm *SwapEpsilon* it is harder to formulate the requirement since the method can produce samples with small and large error, and we do prefer the less erroneous ones.

But indeed, for both algorithms it can be proven that the stationary distributions are uniform, although in the case of *SwapEpsilon* the matrices from the stationary distribution are mainly rubbish since the error increases with time — we have to stop the process before the matrix has totally mixed. In the discussion of algorithm *SwapDiscretized*, we noticed that all the permutations of a matrix with the same margins are generally not connected by swap rotations. Thus the algorithms cannot sample uniformly among all appropriate permutations, but they do sample uniformly among all reachable, appropriate permutations.

Let S be the set of all the possible outcomes of algorithm *SwapDiscretized* with a given starting matrix A. The state space S is then, naturally, connected. The underlying Markov chain is aperiodic since all states are reachable and have a positive probability of staying in itself (e.g., choose $r_2 = r_1$ and $c_2 = c_1$). The chain is also reversible since the probability of selecting rows and columns r_1^i, r_2^i, c_1^i and c_2^i at step i is the same as selecting rows $r_1^{i+1} = r_1^i$, $r_2^{i+1} = r_2^i$ and columns $c_1^{i+1} = c_2^i$, $c_2^{i+1} = c_1^i$ at step $i + 1$ which will undo the swap rotation. Notice that if we had forced the process to move to another state at every step, the reversibility would not hold. Thus, due to Theorem 1 the stationary distribution is uniform. The same applies for method *SwapEpsilon*.

Notice that we do not need to require that all the states attainable with one swap from the current state would have equal probability — the only thing we require is that the probabilities of moving back and forth between two states are the same. Nevertheless, the probabilities can have a huge impact on the mixing time of the process. Thus, selecting some appropriate weighting for the states could speed up the process dramatically.

6.1.4 Metropolis approach

Both of the previous approaches had a primitive error handling method. Next, we develop a more error tolerant method based on Metropolis algorithm. The idea is to generate samples A from probability distribution

$$P(A) = C \exp(-wE(A_0, A)), \tag{14}$$

where A_0 is the original matrix, $w > 0$ is an error scaling constant, C is a normalizing constant and $E(A_0, A)$ is the error of sample A as defined in Equation (11). The proposal distribution Q is a uniform distribution among all the matrices attainable with one swap rotation from the current matrix. A direct implementation of the Metropolis approach is presented in Algorithm 3.

The method starts from the original matrix. However, there is no need for that. Let *SwapMetropolisRandom* be the method obtained by replacing the line 1 by line

Algorithm 3 SwapMetropolis

```
Input: Matrix $A_0$, number of attempts $k$, error scaling constant $w > 0$
 1: $A \leftarrow A_0$
 2: for $i \leftarrow 1, k$ do
 3:     Pick $r_1 \neq r_2$ and $c_1 \neq c_2$ randomly
 4:     $\hat{A} \leftarrow$ Swap($A$, $r_1$, $r_2$, $c_1$, $c_2$)
 5:     $u \leftarrow$ Uniform(0,1)
 6:     if $u < \exp(-w(E(A_0, \hat{A}) - E(A_0, A)))$ then
 7:         $A \leftarrow \hat{A}$
 8:     end if
 9: end for
10: return $A$
```

```
1: $A \leftarrow$ RandomPermutation($A_0$),
```

where the auxiliary method *RandomPermutation* gives a matrix with the elements of A_0 permuted randomly. In both of these methods we allow error but we can control its magnitude. Additionally, the parameter w can be used for changing the accuracy of the method versus the randomness and convergence rate. Note that all swaps that would decrease the error are accepted.

The error difference in line 4 can be calculated in constant time if we keep track on row and column sums and square sums, respectively. The new matrix will differ only in rows r_1, r_2 and columns c_1, c_2 thus the difference of the errors will contain only few terms.

The *SwapMetropolisRandom* method is highly related to a simulated annealing approach where the aim would be to find a matrix with a minimal error. Later in Section 6.2.2, we present a method based on local search for finding extremely accurate random matrices but where we are not able to evaluate the distribution of resulting random matrices. Conversely, the good features of the Metropolis approaches are the clear theoretical properties of the stationary distributions.

There is still one notable property of the final error. In Equation (14) we have only put a restriction for one resulting matrix. But since there exist a lot more matrices with notable error than matrices with almost zero error, the final distribution of the error of resulting matrices is not like (14) — the error will be concentrated more away from the zero. More precisely, the distribution of the final error is

$$p(E(A_0, A) = x) = C \exp(-x) q(x), \tag{15}$$

where $q(x)$ is the distribution of error x of matrices with the original values permuted randomly. Deeper theoretical study of the distribution of final error and error of random permutation is left to further work.

$$\begin{array}{r|ccccc|}
 & & c_1 & & c_2 & \\ \hline
 & & \vdots & & \vdots & \\
r_1 & \dots & +\alpha & \dots & -\alpha & \dots \\
 & & \vdots & & \vdots & \\
r_2 & \dots & -\alpha & \dots & +\alpha & \dots \\
 & & \vdots & & \vdots & \\ \hline
\end{array}$$

Figure 4: The addition in *GeneralMetropolis*

6.2 Generation via transformation of values

All the methods presented earlier were based on a simple swap operation which kept the values unchanged, but permuted them appropriately. Next, we inspect two methods which change the values directly without any swapping. Now, feasible methods are much harder to develop, since there exists more possible output matrices than with the previous swap based algorithms.

6.2.1 General Metropolis approach

The Metropolis algorithm can be used also in other ways than was presented in section 6.1.4. We use the same probability function $P(A)$ for samples A as in Equation (14) but now the samples A can be any real matrices. A new matrix is formed from the current one by selecting rows r_1, r_2 and columns c_1, c_2 randomly and adding the mask presented in Figure 4 to the four intersection elements where α is some random value. The proposal distribution is again symmetric; thus we are really using just Metropolis. The method is presented in Algorithm 4.

Algorithm 4 GeneralMetropolis

```
Input: Matrix $A_0$, number of attempts $k$, error scale $w > 0$, random scale $s > 0$
 1: $A \leftarrow A_0$
 2: for $i \leftarrow 1, k$ do
 3:     Pick $r_1 \neq r_2$ and $c_1 \neq c_2$ randomly
 4:     $\alpha \leftarrow$ Uniform($-s$,$s$)
 5:     $\hat{A} \leftarrow$ AddQuartet($A$, $\alpha$, $r_1$, $r_2$, $c_1$, $c_2$)
 6:     $u \leftarrow$ Uniform(0,1)
 7:     if $u < \exp(-w(E(A_0, \hat{A}) - E(A_0, A)))$ then
 8:         $A \leftarrow \hat{A}$
 9:     end if
10: end for
11: return $A$
```

The auxiliary method *AddQuartet* does the addition of the mask presented in

Figure 4. The parameter random scale s is used to affect the range of random values α which are selected from uniform distribution in $[-s, s]$. Other distributions, e.g., normal distribution, could be used as well for drawing α.

The method has the good property that it doesn't change the sums of rows and columns at all due to the form of the addition mask, but it will change the variances. Thus, in the beginning of the process, it cannot randomize the matrix without making error to the variances of rows and columns. As the error scale w is kept the same during the process, we have to accept quite large error in order to allow enough transformations in the beginning, but then the final outcomes will also contain remarkable amount of error. Another way would be to use two error scales w_1 and w_2, $w_1 \ll w_2$, the first one in the beginning and the second one in the end of the process. In this way, we would allow the matrix first to get randomized and then to correct the error in variances. More detailed study of this approach is left to further work.

6.2.2 Local search approach

The next method is an extremely harsh way for minimizing the error. It is based on a local search where we minimize the error function by changing one value of the matrix at a time. To keep the values reasonable we assume that the values can vary in a range $[a, b]$. At each step we select an element from the current matrix A in random row r and column c. The new matrix is formed by selecting the matrix $\hat{A}$ from the neighbourhood

$$N(A, r, c, [a, b]) = \{\hat{A} \in \mathbb{R}^{m \times n} \mid \hat{A}_{rc} \in [a, b] \text{ and } \hat{A}_{ij} = A_{ij} \text{ otherwise}\}, \quad (16)$$

which minimizes the error function $E(A_0, \hat{A})$. The simple method is presented in Algorithm 5.

Algorithm 5 SeekOptimalValue

```
Input: Matrix $A_0$, number of attempts $k$, range $[a, b]$
1: $A \leftarrow$ Uniform($a$,$b$)$^{m \times n}$
2: for $i \leftarrow 1, k$ do
3:     Pick $r$ and $c$ randomly
4:     $A \leftarrow \arg\min\{E(A_0, \hat{A}) \mid \hat{A} \in N(A, r, c, [a, b])\}$
5: end for
6: return $A$
```

If we use the default parameter values for the error function then it is a fourth degree polynomial and the minimizer in line 4 can be calculated accurately by differentiating. The minimizer is either a or b or one of the real zeros of the third degree derivative polynomial.

Compared to the other methods developed *SeekOptimalValue* always accepts the new state, and the error decreases all the time during the process. The method is likely to find an extremely accurate matrix, but it is hard to say what the stationary distribution will be. However, the resulting matrix is likely to contain a couple of a:s and b:s which can be an unwanted side effect.

7 Empirical results

We perform various experiments to compare the properties of the six introduced methods: *SwapEpsilon*, *SwapDiscretized*, *SwapMetropolis*, *SwapMetropolisRandom*, *GeneralMetropolis* and *SeekOptimalValue*. We evaluate their convergence speed and quality as well as their error rate.

In the experiments we use mainly a publicly available gene expression data[1] studied in [12]. The data consist of gene expression measurements of 1375 genes (rows) in 60 human cancer cell lines (columns), thus the matrix contains $L = 82\,500$ elements. About 2% of the values in the matrix are missing. They were replaced by the average values of the corresponding rows. Finally, the matrix was linearly scaled to the range $[0, 1]$. A significance test for the hierarchical clustering of the cancer cell lines is performed to assess the results in [12].

7.1 Convergence and performance

As all of our methods contain various parameters it is hard to compare their properties objectively. Thus we have just selected some reasonable values for the parameters so that the results are comparable with each other. For *SwapDiscretized* we selected $N = 30$ classes. To make the acceptance rate of *SwapEpsilon* to be similar as with *SwapDiscretized* we selected $\varepsilon = \frac{1}{2N}$. For methods *SwapMetropolis* and *SwapMetropolisRandom* we used value $w = 1$ for the error scaling constant where as with *GeneralMetropolis* we used values $w = 10$ and $s = 0.1$. In method *SeekOptimalValue* we let the values vary in the natural range $[0, 1]$.

We use Frobenius distance as a measure of convergence. If A is the original matrix and $\hat{A}$ is the randomized matrix then the Frobenius distance is

$$\|A - \hat{A}\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{n} (A_{ij} - \hat{A}_{ij})^2. \tag{17}$$

Notice that it doesn't measure the error in the margins but it measures the dissimilarity between the two matrices.

[1]Dataset downloaded June 1, 2007, from http://discover.nci.nih.gov/datasetsNature2000.jsp

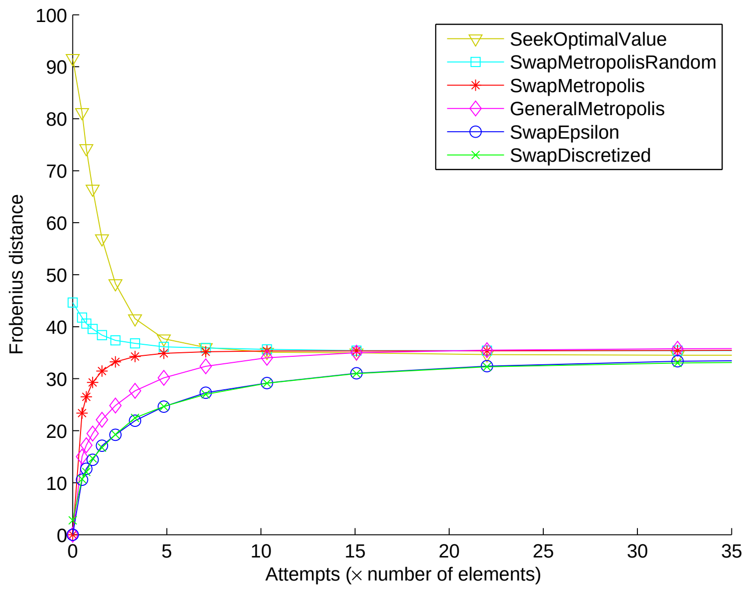


Figure 5: Convergence: Frobenius distance between the original and randomized matrix as a function of attempts.

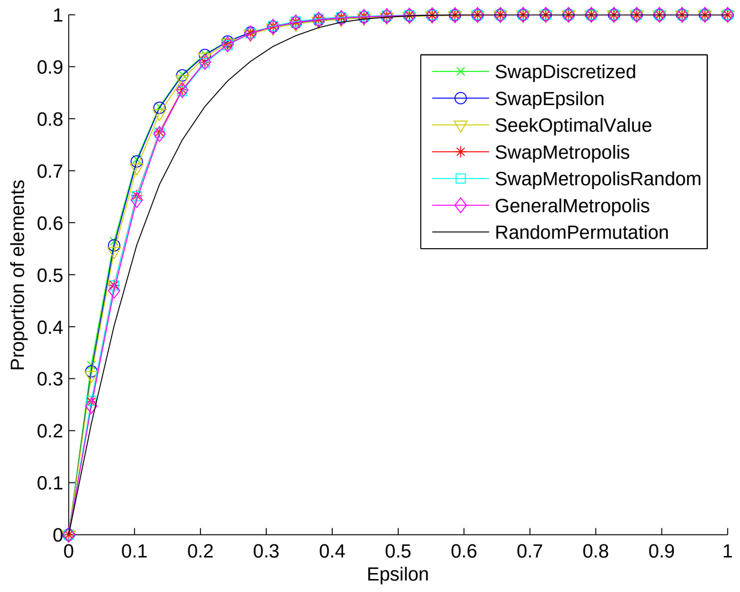


Figure 6: Proportion of elements in a randomized matrix whose values differ less than epsilon from the values in the corresponding locations of the original matrix. RandomPermutation is a matrix with the original values permuted randomly.

Method	Time (s)	Acc. rate
SwapEpsilon	3.8	0.11
SwapDiscretized	2.8	0.11
SwapMetropolis	8.9	0.41
SwapMetropolisRandom	9.3	0.41
GeneralMetropolis	6.2	0.57
SeekOptimalValue	7.6	1.00

Table 1: Running times and acceptance rates with $30L$ attempts.

In Figure 5 we present the Frobenius distance as a function of attempts. For each method we generated 16 samples with different number of attempts varying in the range $[0, 100L]$. Only one sample was used for each number of attempts since the variation between different runs were minimal. Notice that all samples were independently produced by starting from the original matrix and keeping only the last matrix of the Markov chain. In the figure, x-axis is bounded to $35L$ since all methods had converged before that in $O(L)$ steps. The constant factor depends on the method and the parameters used. Especially with *SwapEpsilon* and *SwapDiscretized* the parameters have a direct impact on the convergence speed. We observe that all methods converge to around the same Frobenius distance, i.e., they randomize the matrices somehow equally well. This is actually partly due to the parameter selection. Due to the different kind of approaches the Frobenius distance decreases with *SeekOptimalValue* and *SwapMetropolisRandom* since they start from a random matrix. As a consequence the resulting matrix is in some sense closer to the original matrix than a totally random matrix would be.

In Figure 6 we present the number of the elements of the randomized matrix which are close to the elements of the original matrix in the same locations. The randomized matrices are produced with $30L$ attempts. If the result matrix is randomized well, the curve of the corresponding method in Figure 6 should be close to the curve of random permutation. Thus, it seems that all methods have been able to forget the original matrix, excluding its margins. The high concentration of the values in the gene expression data matrix explains the shape of the RandomPermutation curve.

In Table 1 we give the running times and acceptance rates of the different methods with $30L$ attempts. The tests were done using modest C++ implementations integrated with Matlab on a 2.8GHz Pentium 4 machine with 512MB of memory. We notice that all the methods performed relatively fast. Since the acceptance rates of *SwapEpsilon* and *SwapDiscretized* are small they are slightly faster than the other methods. However, in the case of the other methods notably less than $30L$ attempts would likely be sufficient.

7.2 Error rate

In Figure 7 the progress of the total error is presented. The error is calculated by Equation (11) with default parameter values. We notice that the error with *SwapDiscretized* stays the same as it should, whereas with *SwapEpsilon* it continues increasing. Both methods *SwapMetropolis* and *SwapMetropolisRandom* converge to the same error rate so we can say that the starting state really doesn't seem to matter. The method *SeekOptimalValue* obtains an extremely small error and the error would have still decreased with the time. To get some concrete idea of the amount of error we consider a sample produced by *SwapMetropolisRandom* with $30L$ steps. In this sample the maximum errors of row sums, column sums, row square sums and column square sums are 1.76, 0.51, 0.74 and 0.42 correspondingly whereas the average row sum, columns sum, row square sum and column square sum are 34.68, 794.64, 20.77 and 475.91 correspondingly. Thus the errors are actually relatively small and they could be still decreased by tuning the parameters.

In Figure 8 we present the error distribution of the randomized matrices produced by *SwapMetropolisRandom* when the original matrix was a small 5-by-5 random real matrix. There is also presented the error distribution of random permutations. The theoretical curve is calculated by Equation (15). The matrix was chosen to be small so that the histograms would overlap and the theory could be tested. Notice that the error of randomized matrix is concentrated away from zero and the theory curve fits the error of randomized matrix well.

7.3 Clustering

The gene expression data we are using was studied in detail in [12]. The 60 human cancer cell lines (columns of the data matrix) contained a clear clustering structure, which was a remarkable result. In the following we study the significance of this result by the introduced methods.

In Figure 9, dendrograms of average-linkage hierarchical clustering of the original data matrix and a randomized data matrix are presented. The randomized sample was produced by *SwapMetropolisRandom* with $30L$ attempts. The original data contains a clear clustering whereas the randomized data contains no structure. Similar results were obtained with all the other methods as well.

To test the significance more carefully we generated 10 000 samples with each of the methods using $30L$ attempts, parameters being the same as explained in section 7.1. To measure the structure of a sample we summed the Euclidean distances of the hierarchical clustering together to obtain a single number expressing the clustering error. In Table 2 statistics of clustering errors of the 10 000 samples generated are presented for each method. The corresponding clustering error of

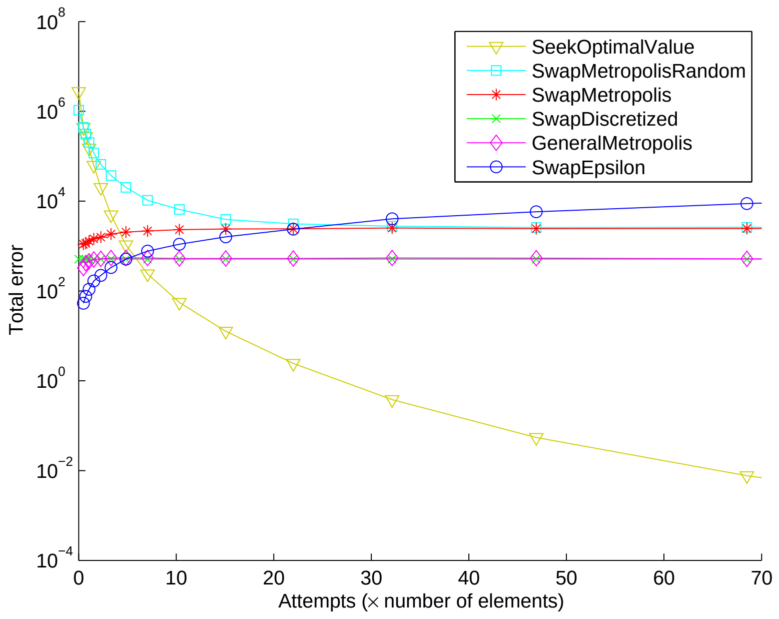


Figure 7: Total error as a function of attempts.

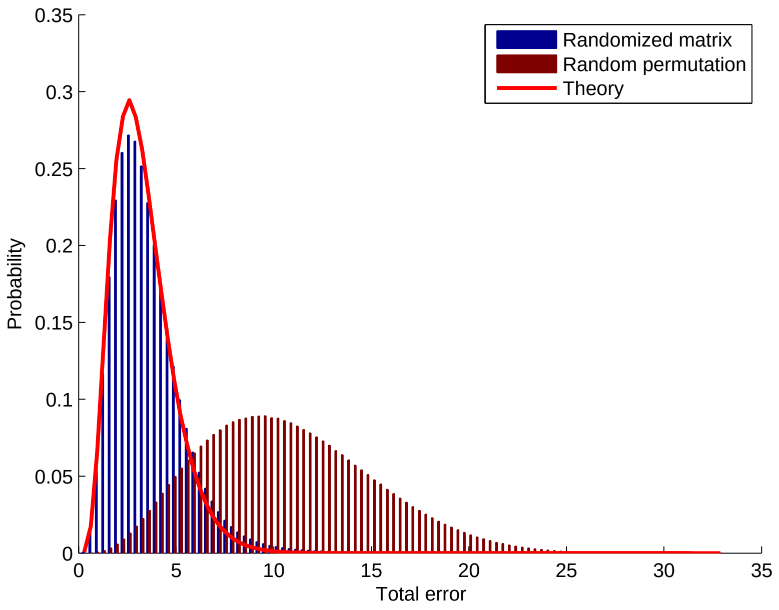
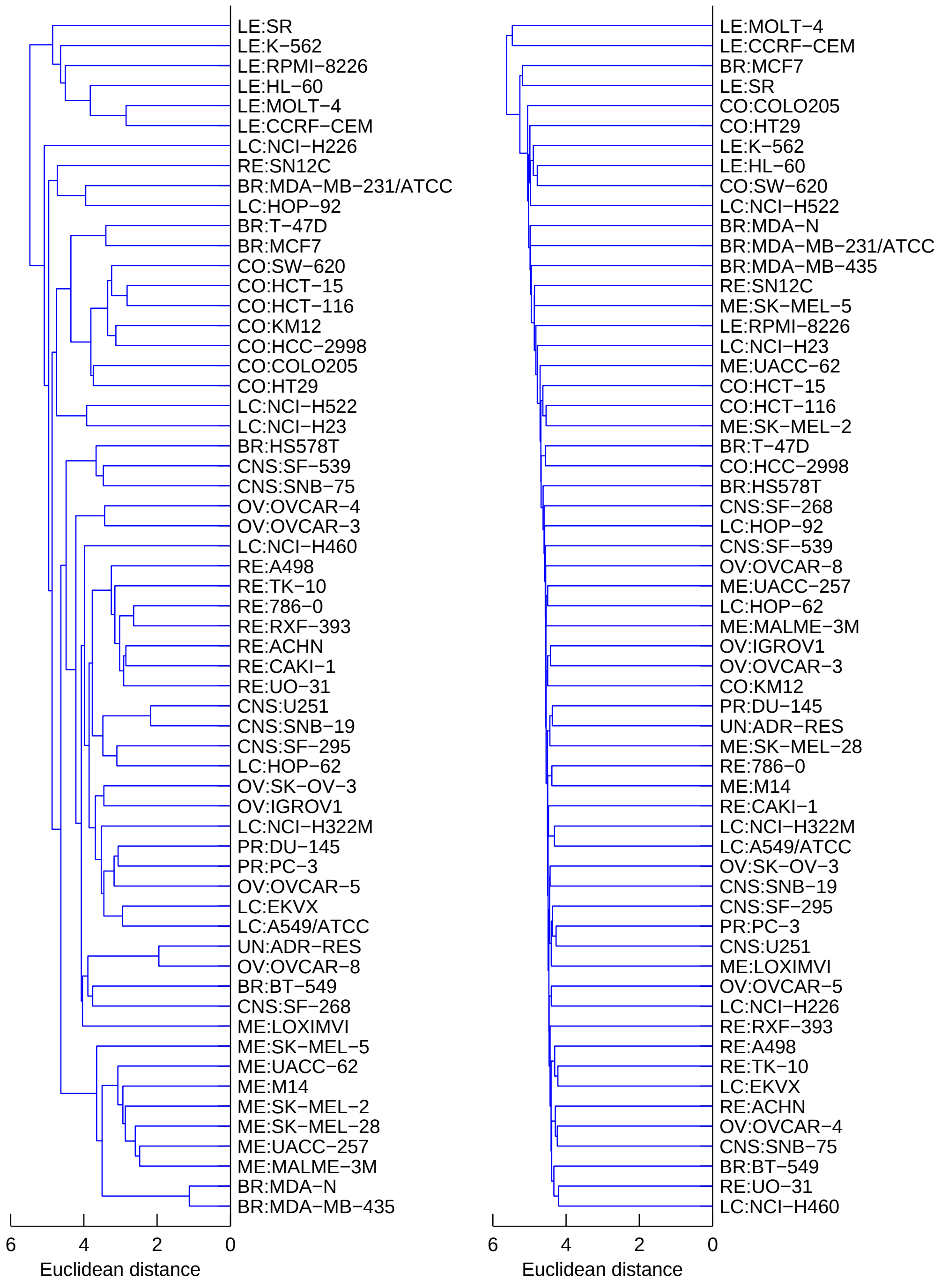


Figure 8: Distributions of total error of matrices randomized by *SwapMetropolis-Random* and matrices with random permutation. Theory depicts the distribution of total error of randomized matrices by total error of random permutations.



(a) Original data (b) Randomized data

Figure 9: Dendrograms of hierarchical clustering of the 60 human cancer cell lines of the original data and the same data randomized by *SwapMetropolisRandom*.

Method	Min	Mean (Std)	p-value	Z-score
SwapEpsilon	257.39	259.11 (0.41)	0.0001	-116.88
SwapDiscretized	254.41	255.94 (0.40)	0.0001	-110.75
SwapMetropolis	271.09	272.40 (0.31)	0.0001	-193.49
SwapMetropolisRandom	271.37	272.67 (0.32)	0.0001	-192.71
GeneralMetropolis	281.39	282.92 (0.42)	0.0001	-170.09
SeekOptimalValue	257.22	258.85 (0.41)	0.0001	-115.56

Table 2: Clustering errors of randomized matrices

the original gene expression data was 221.54. The p-values and Z-scores were calculated using Equations (1) and (2). We note that none of the generated samples were more structured than the original data and the difference is huge as is seen from the Z-scores and minimum values. Thus the original gene expression data contains a significant clustering structure which is not just due to the margins. Actually the row and column sums and square sums of the original data are close to each other thus the result is believable.

8 Conclusions and future work

We have introduced six different methods for sampling random, real valued matrices with given margins, compared their properties and performance, and discussed how they can be used in assessing results of data mining algorithms. The six methods, *SwapEpsilon*, *SwapDiscretized*, *SwapMetropolis*, *SwapMetropolisRandom*, *GeneralMetropolis*, and *SeekOptimalValue*, have different advantages and disadvantages. The main differences are in the error rate, uniformity, value distribution, and convergence speed. The method to use has to be chosen according to the application.

We conducted the experiments using only one real world data matrix, where the results were clear, but more study on suitable applications is needed. The approaches we used for randomizing real matrices were promising. However, the methods should be analyzed in more detail and new ones should be developed. The effect of the error measure should be considered as well.

9 Acknowledgments

I thank Niko Vuokko for co-operation in developing the methods, Heikki Mannila for giving the original problem and useful ideas, and Kai Puolamäki for instructive support during the writing process.

References

[1] Christophe Andrieu, Nando de Freitas, Arnaud Doucet, and Michael I. Jordan. An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1):5–43, 2003.

[2] Pavel Berkhin. Survey Of Clustering Data Mining Techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[3] Yuguo Chen, Persi Diaconis, Susan P. Holmes, and Jun S. Liu. Sequential Monte Carlo Methods for Statistical Analysis of Tables. *Journal of the American Statistical Association*, 100:109–120, 2005.

[4] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*. Chapman & Hall, New York, 2nd edition, 2003.

[5] George Casella, Roger L. Berger. *Statistical Inference*. Duxbury Press, 2nd edition, 2001.

[6] Aristides Gionis, Heikki Mannila, Taneli Mielikäinen, and Panayiotis Tsaparas. Assessing Data Mining Results via Swap Randomization. In *Proceedings of the 12th ACM Conference on Knowledge Discovery and Data Mining (KDD)*, 2006.

[7] Phillip Good. *Permutation Tests: A Practical Guide to Resampling Methods for Testing Hypotheses*. Springer, 2nd edition, 2000.

[8] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[9] S. Lloyd. Least Squares Quantization in PCM. Technical report, Bell Laboratories, 1957.

[10] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE Transactions on Computational Biology and Bioinformatics*, 1(1):24–45, 2004.

[11] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21(1):1087–91, 1953.

[12] Uwe Scherf *et al.* A Gene Expression Database for the Molecular Pharmacology of Cancer. *Nature Genetics*, 24(3):236–244, 2000.