Data randomization for assessing the results of data mining

CS-E4650 Methods of Data Mining

Lecture November 7, 2023

Heikki Mannila

heikki.mannila@aalto.fi

Aalto-yliopisto
Aalto-universitetet
Aalto University

Background

- Data analysis methods are very useful tools
- There are possible problems in using them
- Finding something which is only an artifact of the data or the analysis process, not representative of any real phenomena
- Also, issues related to bias, privacy, etc.

Simple ways of using randomization

- Add some noise to the data
- Does the result change dramatically?

- Remove a part of the data
- How much does the result change?

- If small changes in the data cause large changes in the results, it is usually good to understand why

What does a result mean? Example

- Even if the result seems robust, it does not mean that the result is informative or useful

- Example: Clustering: a clustering algorithm will return a clustering
- Even in the case when there is no cluster structure
- Trivial example: two-dimensional completely regular data
- What kind of cluster structure do we get?
- For example, by using k-means
- Does the result tell us something useful of the data?

- If possible, plot your data!

In this lecture

- We look at some simple ideas for randomizing data to help in assessing the results

- There is some nice theory behind these methods
- We do not discuss the theory much

- Additional reading: Assessing Data Mining Results via Swap Randomization, A. Gionis et al., ACM Transactions on Knowledge Discovery from Data, Vol. 1, No. 3, Article 14 (December 2007), ACMJ346-06.tex

Significance testing

- A statistic of interest f(D) on the data
- A null hypothesis about the data D: a probability model that can be used to tell how likely or unlikely a value f(D) is

- If probability of such a value is small, then it might be that the null hypothesis is not true

- Example: coin flipping

Significance testing of results using randomization

- Describe a randomized process for generating the data D
- Test statistic: a quantity of interest f(D)
- Generate many independent versions D_i of the data
- Test whether the value f(D) of the test statistic on real data is extreme compared to the values $f(D_i)$
- Resulting in an estimate of how likely that is
- Comparison with traditional significance testing?
  - Randomized process replaces the probabilistic model
  - The randomized model can be arbitrarily complex
    - Good and bad

Significance testing has its problems

- Problems related to multiple testing
- If you do sufficiently many analyses, you'll find something even from random data

- Examples:
  - Spurious correlations
  - xkcd Significant https://xkcd.com/882/

- Several methods for handling this
  - Bonferroni correction; false discovery rate (FDR) approach

Significance testing has its problems

- One might do many different analyses without realizing it
- “Garden of forking paths” –
  - Selecting subsets of the data
  - Selecting properties of interest in the data
  - Soon one has actually done many analyses
- These problems remain in the randomization approach

Contents

- Basic idea of randomizing data
  - Example: 0-1 sequence analysis

- Randomizing data in prediction tasks
  - Example: ecological presence-absence data

- Swap randomization for 0-1 matrices
  - Example from in ecological data

- Bootstrapping (very briefly)
- Back to sequences

Basic approach in randomization

- You compute something from the data D: value b = f(D)
- For i=1,…,n
  - D_i = result of randomizing D
  - compute $b_i = f(D_i)$
- Compare the value b obtained from the real data to the set of values $b_1,\ldots, b_n$
- If b is right there among the others, then the value b=f(D) is no big news
- If b is quite extreme among the b_i's, then perhaps the value of b=f(D) is interesting

A simple example

- Monitoring a process
- At each timepoint we see either 0 (all OK) or 1 (something wrong)

- We are interested in finding out whether there are surprisingly long sequences of consecutive 1s in the sequence

- Example: a sequence S, which we know nothing about

Sequence analysis

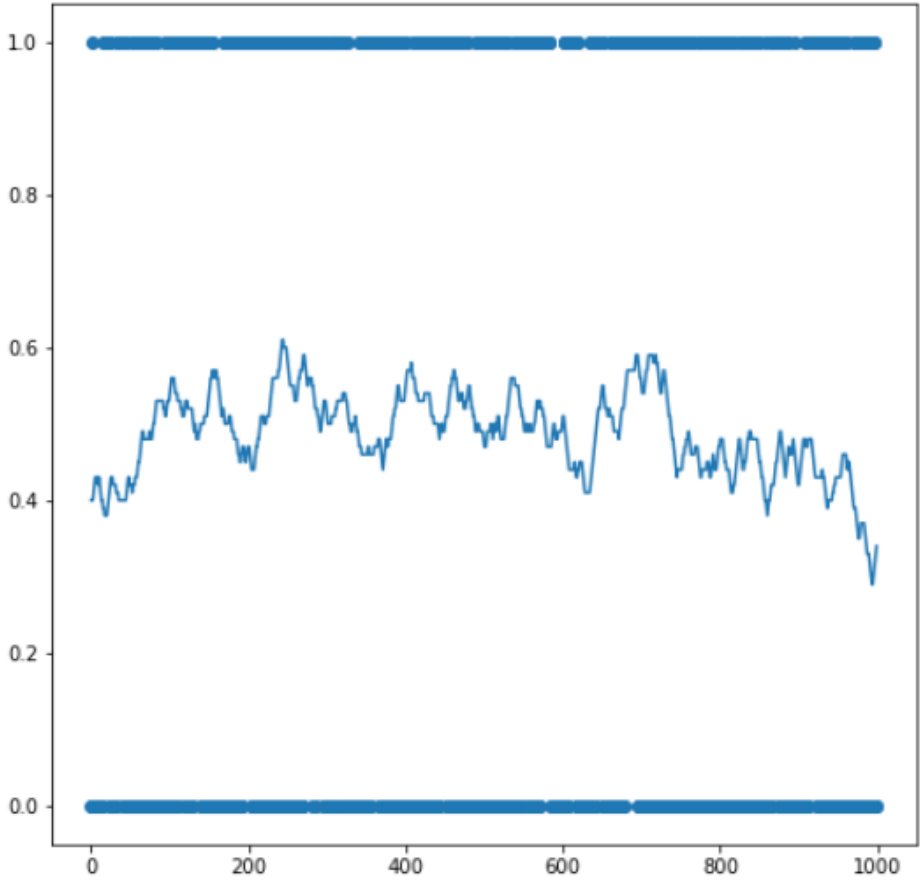
Descriptive statistics

```
In [5]: len(S), (S==0).sum(), (S==1).sum()
```

```
Out[5]: (1000, 513, 487)
```

So 1000 elements, about equally 0s and 1s

Sequence analysis – plot the data



Sequence analysis – longest run of consecutive ones

In [14]:
```
z = longestrun(S)
print('The original sequence S had a run of ', z, 'consecutive 1s')
```

```
The original sequence S had a run of  17 consecutive 1s
```

In [15]:
```
pS = np.random.permutation(S)
pz = longestrun(pS)
print('A permuted version of S had a run of ', pz, 'consecutive 1s')
```

```
A permuted version of S had a run of  9 consecutive 1s
```

Is 17 consecutive one a lot or not?

Sequence analysis

Permute 1000 times and collect data

```
iter = 1000
lenres = np.zeros(iter)
for i in range(iter):
    lenres[i]=longestrun(np.random.permutation(S))

L = longestrun(S)
print(L, lenres.mean(),lenres.std(), lenres.max())
# How many times was the result on randomized data smaller than on the true data
print((lenres>=L).sum())
```

```
17 8.929 1.8520148487525685 21.0
5
```

A simple example, cont.

- Quantity of interest F: length of the longest substring of 1s
- For the original data S we have F(S)=17

- We randomized the sequence S by permuting it
- The permuted sequence
  - Has the same number of 1s and 0s as the original sequence S
  - But the order has been lost

- The longest sequence of 1s in the 1000 permuted sequences was of length 21
- For 5 permutations the length of the longest sequence was at least 17

Basic approach in randomization (see slide 11)

- If b=f(D) is right there among the others, then the value is no big news

- If b is quite extreme among the $b_i=f(D_i)$, then perhaps b=f(D) is interesting
- Compute

  z = | { i | 1 <= I <= n & f(D) > f(Di)} | / n
- Or

  z = | { i | 1 <= I <= n & f(D) < f(Di)} | / n

In the example: 5/1000 = 0.005

Wait a minute …

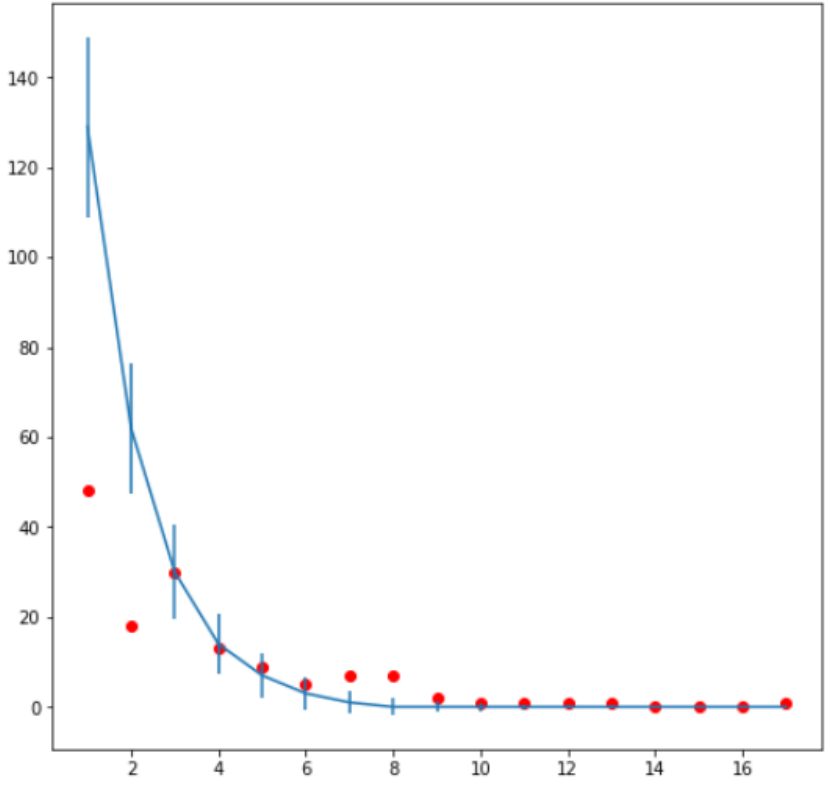
- Couldn't we have estimated the probability of finding a substring with k 1s in a random permutation fairly easily?
- Yes, in this example that would have been possible

- For more complex examples, e.g.,
  - Distribution of all-1 substrings of size k for varying k
  - Longest subsequence with density at least a given bound
- It is harder to determine the probability analytically
- Quite easy using the randomization approach

Sequence analysis – distribution of the lengths of runs of 1s

Why does the distribution of the lengths of the 1s look like it does?

One should check the correctness!



Sequence analysis – longest subsequence with density at least 0.65? A slow implementation

In [28]:
```
iter = 100
ssres = np.zeros(iter)
for i in range(iter):
    ssres[i], maxi, maxj = ldss(np.random.permutation(S),0.65)
```

In [29]:
```
truess, maxi, maxj = ldss(S,0.65)
print(truess, ssres.mean(),ssres.std(), ssres.max())
# How many times was the result on randomized data smaller than on the true data
print((ssres>=truess).sum())
```

```
69 56.35 17.73943347460679 143.0
24
```

24/100 = 0.24

Garden of forking paths?

What does the example tell us?

- There are longer all-1 substrings in the data than in a random permuted sequence [with the same number of 1s]
- The distribution of all-1 substring of varying lengths differs from that in a randomly permuted sequence
- The longest subsequence with density at least 0.65 is about the same length as in a randomly permuted sequence
- Garden of forking paths?

Where did the sequence come from?

- A 2-state Markov chain
- In state 0, output 0 with probability 0.7 and stay in state 0
- In state 0, output 1 with probability 0.3 and go to state 1
- In state 1, output 1 with probability 0.7 and stay in state 1
- In state 1, output 0 with probability 0.3 and go to state 0

```
def genseq(n,a,b):
    res = np.zeros(n)
    state = 0
    for i in range(n):
        r = np.random.uniform()
        if state==0 and r < a:
            res[i]=0
        elif state==0 and r > a:
            res[i]=1
            state=1
        elif state==1 and r < b:
            res[i]=1
        elif state==1 and r > b:
            res[i]=0
            state=0
    return res
```

```
N = 1000
S = genseq(N,0.7,0.7)
```

Randomization in a prediction task

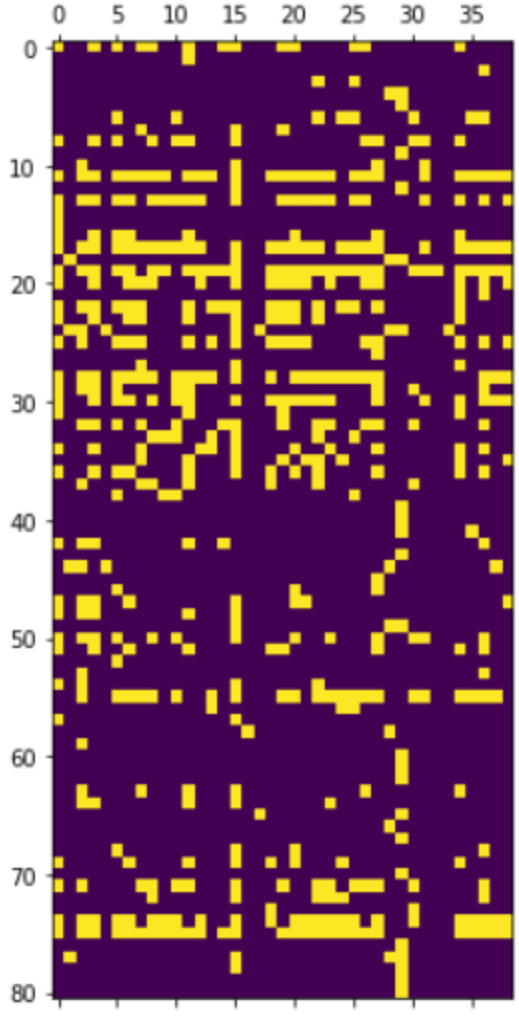
- In the previous example we computed some useful statistics from the data
- Conside now a prediction task: given data matrix X, predict target value y
- We want to find out whether the rows of X give us some information on the values of y
- How to use randomization here?
- Permute the values of y
- Cut all connection between X and y
- **Maintain the distribution of the values of y**

Prediction task - data

	Aoteapsyche sp1 FW_013_05	Aphrophila noevaezelandiae	Archicaulioides diversus	Austraclima jollyae	Austrosimulium australense	Costachorema xanthoptera	Cricotopus I	Cricotopus II	Deleatidium sp1 FW_013_05	Eukiefiriella	...
Achnanthes lanceoloata	1	0	0	0	1	0	1	0	1	1	...
Achnanthes linearis	1	1	0	0	1	0	1	0	1	1	...
Achnanthes minutissima	0	0	0	0	0	0	0	0	0	0	...
Achnanthes saxonica	0	0	0	0	0	0	0	0	0	0	...
Achnanthes sp1 FW_013_05	0	0	0	0	0	0	0	0	0	0	...
...	...	...	...	...	...	...	...	...	...	...	...

- Reference:Thompson, R., Townsend, C. (2003) IMPACTS ON STREAM FOOD WEBS OF NATIVE AND EXOTIC FOREST: AN INTERCONTINENTAL COMPARISON. Ecology, 84(1), pp. 145–161
- Source:https://esajournals.onlinelibrary.wiley.com/doi/abs/10.1890/0012-9658(2003)084%5B0145:IOSFWO%5D2.0.CO;2,
- Species:98
- Predators (Columns) Preys (Rows)
- https://www.web-of-life.es/map.php

Prediction task – plot the data



Basic check

- Suppose we are interested in predicting the value of colum 11
- The data seems to be fairly sparse (lots of 0s)
- How accurate is predicting “0” for every row?

```
In [50]: (1 - otagonpdata[:,colofinterest].sum()/otagonpdata.shape[0]).round(2)
```

- We get 68% accuracy without looking at the data in any detail

Prediction task: logistic regression

```
colofinterest = 11
y = otagonpdata[:,colofinterest] # column colofinterest
X = np.delete(otagonpdata.copy(),colofinterest,1) # delete column colofinterest from a copy
#print(y.shape, X.shape)
```

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0).fit(X, y)
clf.score(X, y)
```

```
0.9259259259259259
```

```
# So we could predict column 11 by using the other columns with 93 % accuracy
```

```
permy = np.random.permutation(y)
permclf = LogisticRegression(random_state=0).fit(X, permy)
# print(y, permclf.predict(X), permy-permclf.predict(X))
permclf.score(X, permy)
```

```
0.8024691358024691
```

```
#... and we could predict a random column with the same number of 1s with 85 % accuracy
```

```
1-colsums[11]/81
```

```
0.6790123456790124
```

```
# And just by saying "each entry is 0" would give us 68 % accuracy
```

In the demo example

- The accuracy on the randomized data was almost as high as for the original data
- The reason: lots of predictors, few 1s in the column to be predicted

- Conclusion: we should not be two enthusiastic about the prediction results

- (What happens for the other columns in the data?)

An aside: a paper from 2017/2020

[3446776 (acm.org)]

DOI:10.1145/3446776

Understanding Deep Learning (Still) Requires Rethinking Generalization

By Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals

Abstract

Despite their massive size, successful deep artificial neural networks can exhibit a remarkably small gap between training and test performance. Conventional wisdom attributes small generalization error either to properties of the model family or to the regularization techniques used during training.

Through extensive systematic experiments, we show how these traditional approaches fail to explain why large neural networks generalize well in practice. Specifically, our experiments establish that state-of-the-art convolutional networks for image classification trained with stochastic gradient methods easily fit a random labeling of the training data. This phenomenon is qualitatively unaffected by explicit regularization and occurs even if we replace the true images by completely unstructured random noise. We corroborate these experimental findings with a theoretical construction showing that simple depth two neural networks already have perfect finite sample expressivity as soon as the number of parameters exceeds the number of data points as it usually does in practice.

We interpret our experimental findings by comparison with traditional models.

We supplement this republication with a new section at the end summarizing recent progresses in the field since the original version of this paper.

More complex randomizations

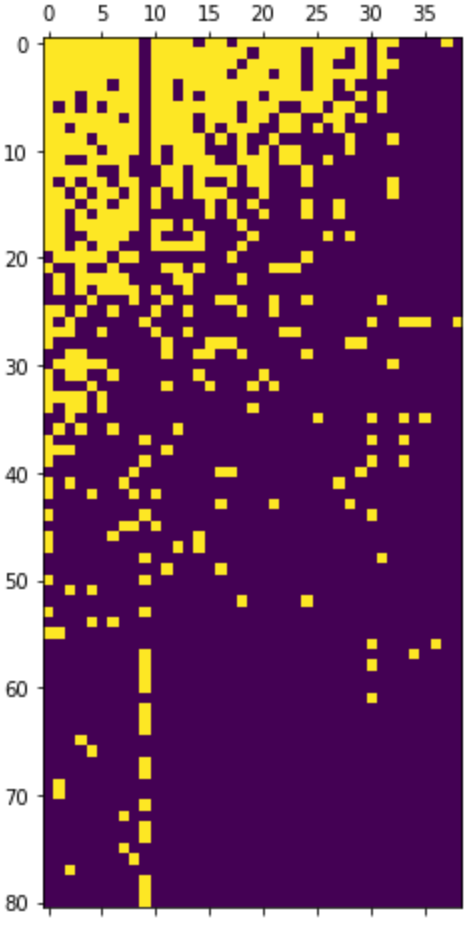
- In the previous examples we randomized *a sequence or a column* of the data
- What remained constant in the randomizations?
  - Total number of 1s
  - Counts of different values in the target column y

- Sometimes it is useful to randomize *the whole data matrix*
- For 0-1 data it is natural to maintain the row and column sums
- Number of 1s in each row and in each column
- Why would we be interested in this? Skewed distributions

Skewed distributions

- Previous data
- Rows and columns reordered by row sum and column sum

- Very skewed distributions of row and column sums



Example

X and Y are correlated in the data.

In the second dataset, their correlation can be attributed to the highly skewed row sums in the data set.

X	Y
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	0
1	0
0	1
0	1
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0

dataset $\mathcal{D}_1$

X	Y
1	1
1	1
1	1
1	1
1	1
1	1
1	1
1	0
1	0
0	1
0	1
0	0
0	0
0	0
0	0
0	0
0	0
0	0
0	0

dataset $\mathcal{D}_2$

Example (cont.)

- X and Y are correlated
- If we consider only columns for X and Y the correlation is clear
- If we look at the larger matrix, we notice that we can perhaps explain the correlation between X and Y by the number of 1s in the rows
- X and Y are correlated because they both tend to occur in the rows with many 1s
- This might be useful to notice if the data is, e.g., ecological data
- Two species X and Y are correlated because they occur in environments where there are many species

Swap randomization

- Randomization idea:
  - Given 0-1 matrix E
  - Produce randomized versions E_i of E
  - Having the same row and column sums as E

- How can we do this?

Swap randomization

- Given a matrix, change the data by applying the operation below:

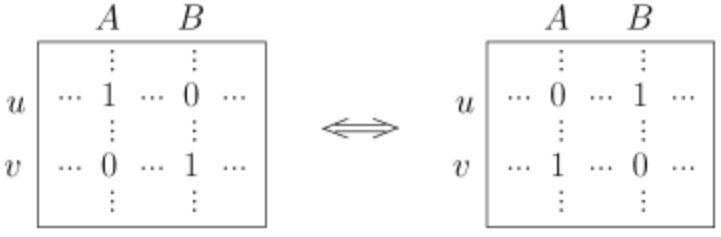


Fig. 1. A swap in a 0–1 matrix.

- This maintains the number of 0s and 1s in rows and columns

Swap randomization

- Randomization: do enough swaps to get a random matrix with the same row and column sums as the original data

- As in the earlier examples, compute the quantity of interest both the original data and for the randomized data

Toy example: matrices D1 and D2

```
iter = 500
origXYcorr = np.corrcoef(D1,rowvar=False)[0,1]

XYcorrs= np.zeros(iter)

for i in range(iter):
    D1s, succ = swaprandomization(D1,10000)
    XYcorrs[i] = np.corrcoef(D1s,rowvar=False)[0,1]

print(origXYcorr, XYcorrs.mean(), XY
```

```
0.55 -0.04175 0.25939484863813317
```

```
iter = 500
origXYcorr2 = np.corrcoef(D2,rowvar=False)[0,1]

XYcorrs2= np.zeros(iter)

for i in range(iter):
    D2s, succ = swaprandomization(D2,10000)
    XYcorrs2[i] = np.corrcoef(D2s,rowvar=False)[0,1]

print(origXYcorr2, XYcorrs2.mean(), XYcorrs2.std())
```

```
0.55 0.5198499999999999 0.14651698706975919
```

Thus

- [Computing correlations for such small (0-1) datasets is not good practice, but we use it to illustrate the point. Instead, one could compute, e.g., the dot product of columns X and Y.]

- X and Y are highly correlated.
- Randomize by looking at all matrices with the same row and column counts as D1
- The correlation is for data obtained from D1 by randomization is low
- For data obtained from D2 by randomization the correlation is hight

- For D2, the correlation of variables X and Y is explained by the row and column counts of D2

Swap randomization

- Theorem [Ryser 1957]: Assume D and E are n x m matrices with the same row and column sums. Then there is a sequence of swap operations that transforms D to E.
  - HJ Ryser: “Combinatorial properties of matrices of zeros and ones” - Canadian Journal of Mathematics, 1957.
- Not an easy proof, but not terribly difficult.

- Graph formulation:
  - Vertices: matrices with the same row and column sums
  - Edges: between D and E, if doing one swap to D gives E
- Ryser’s theorem: this graph is connected

Swap randomization

- Theorem, informal: if you do sufficiently many random swap operations starting from D, then the resulting matrix is a random sample from the set of all matrices with the same row and column sums as D.

- Thus: we can produce random matrices with the same row and column sums as D
- And do randomization testing using those.

Huh? “Sufficiently many”?

- The number of random swaps needed to achieve “full mixing” is a deep question
- A random walk on the graph where the nodes are the matrices with the given row and column sums

- Experimentally it seems that about 5*L swaps is enough, if the matrix D has L 1s.

How to do swap randomization

- Easy to code the simplest possible algorithm
  - Generate randomly row numbers u and v
  - Generate randomly column numbers A and B
  - If the submatrix has the desired form, do the swap
  - Repeat until sufficiently many successful swaps
  - Slow
  - Really slow for large data

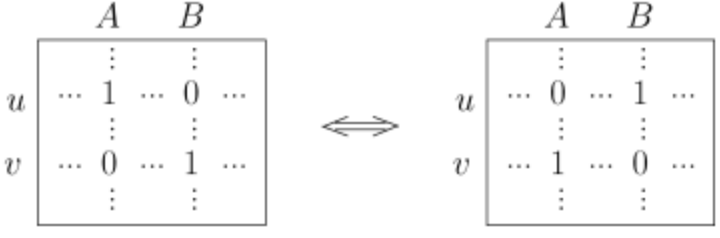


Fig. 1. A swap in a 0–1 matrix.

How to do swap randomization faster

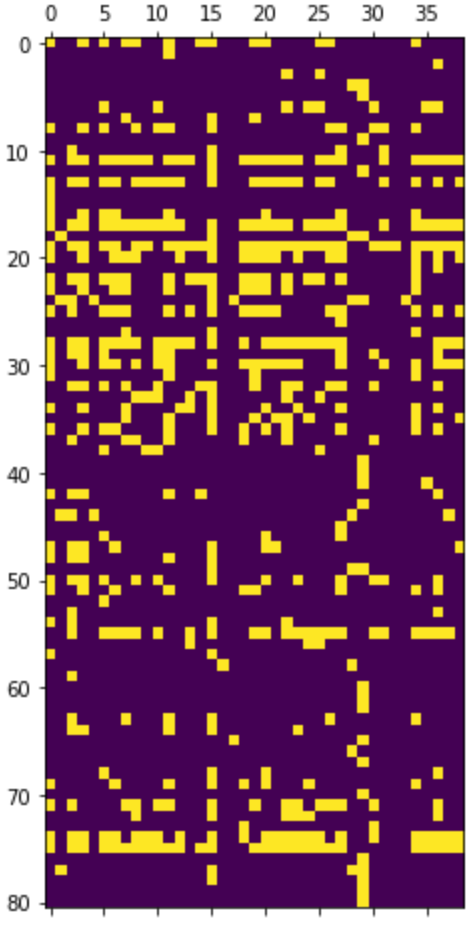
- Many algorithms exist
- Not simple to guarantee that the method gives a random sample of the matrices with the given row and column sums

- See, e.g., A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals | Nature Communications from 2014

Clustering

- Clustering methods have already been covered in the course
- Randomization for clustering

- Are there clusters in the data?
- The naïve randomization: probability of 1 is the same in the randomized data as in the original data



Example: clustering

- Clustering methods have already been covered in the course
- Randomization for clustering

- Example

Trivial randomization for clustering

```
ones = (otagonpdata==1).sum()
print(ones)
```

```
577
```

```
# Frequency of 1s in the data
(rc, cc) = otagonpdata.shape
freq =  ones/(rc*cc)
print('Frequency of 1s in the data', ones, rc, cc, freq)
```

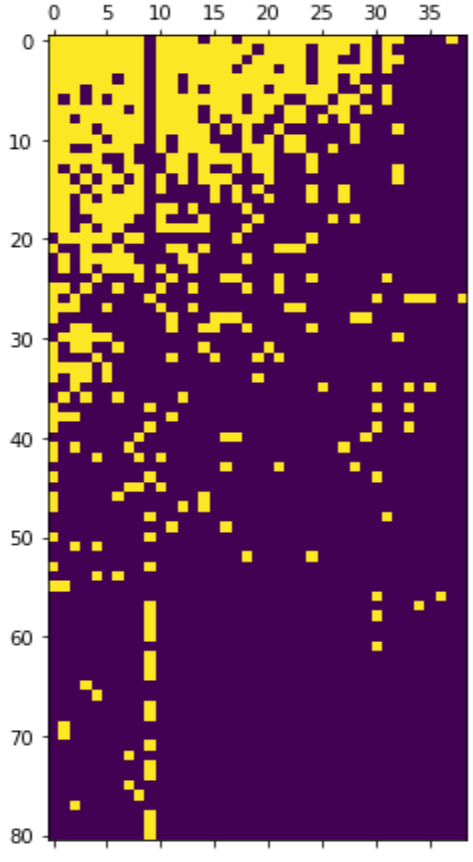
```
Frequency of 1s in the data 577 81 39 0.18265273820829375
```

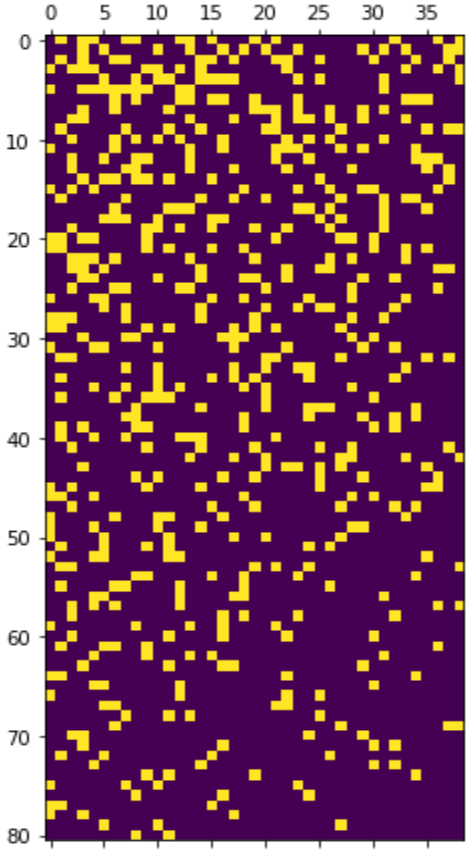
```
randm = (np.random.random_sample(size=(rc,cc))<freq).astype(int)

print(randm.shape, (randm==0).sum(), (randm==1).sum(), (randm==1).sum()/(rc*cc))
randm
```

```
(81, 39) 2611 548 0.17347261791706237
```

Trivial randomization for clustering: original and randomized reordered – row and column sums can change





Trivial randomization for clustering

```
kmeans = KMeans(n_clusters=3).fit(otagonpdata)
scoredata = -kmeansr.score(otagonpdata)
scoredata
```

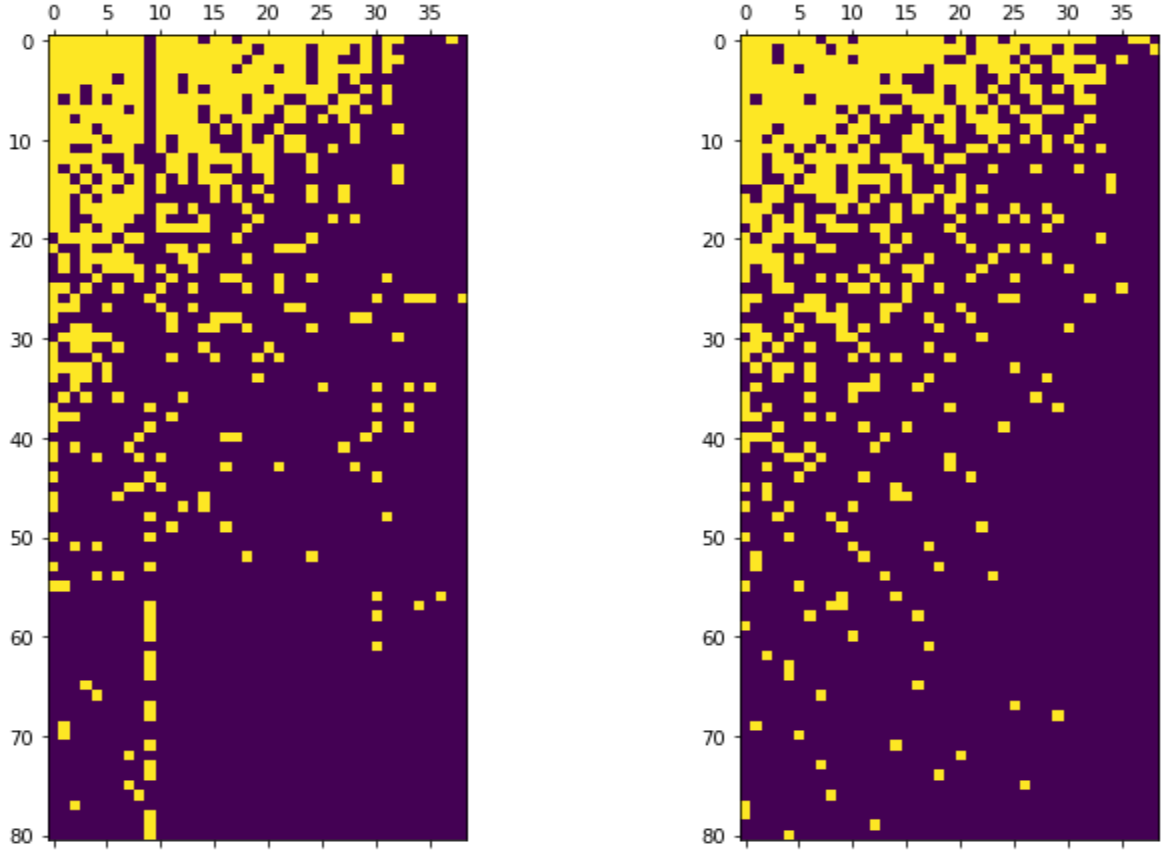
```
293.6834138724523
```

```
iter = 500
clusres = np.zeros(iter)
for i in range(iter):
    randm = (np.random.random_sample(size=(rc,cc))<freq).astype(int)
    kmeansr = KMeans(n_clusters=3).fit(randm)
    clusres[i]=-kmeansr.score(randm)
```

```
print('Original data ', scoredata, 'randomized mean', clusres.mean(), 'std', clusres.std()
print('Number of cases with better score', (clusres<scoredata).sum())
```

```
Original data  293.6834138724523 randomized mean 425.72046638298644 std 12.824839253082015
Number of cases with better score 0
```

Swap randomization for clustering: row and column sums stays the same



Swap randomization for clustering

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4).fit(otagonpdata)
scoredata = kmeans.score(otagonpdata)
-scoredata
```

```
233.39814814814812
```

```
iter = 100
sclusres = np.zeros(iter)
swaprdata, junk = swaprandomization(otagonpdata,500000)

for i in range(iter):
    print(i)
    swaprdata, junk = swaprandomization(swaprdata,500000)
    kmeansr = KMeans(n_clusters=4).fit(swaprdata)
    sclusres[i]=-kmeansr.score(swaprdata)
```

```
scoredata, sclusres.mean(), sclusres.std(), (sclusres<scoredata).sum()
```

```
(-233.39814814814812, 276.1954270272572, 3.2930663706204175, 0)
```

Hence

- The clustering on the real data has a better score than those for swap randomized data
- The difference is a lot smaller than for the trivial randomization

- Thus the cluster structure might not be completely due to the skewd row and column distributions in the data

Larger experiments (Gionis et al 2007)

Table VII. Results on Clustering

Dataset	k	E	mean	std	Z	p
S1	10	1777.3	3669.9	11.1	170.43	0.01
	20	1660.7	3303.2	11.3	145.33	0.01
S2	10	4075.4	4084.4	11.6	0.77	0.22
	20	3686.2	3691.3	12.1	0.42	0.36
COURSES	10	17541.6	24405.1	30.2	227.09	0.01
	20	16062.0	23588.4	31.9	235.92	0.01
PALEO	10	1040.7	1401.7	4.8	74.74	0.01
	20	800.1	1193.9	5.9	67.09	0.01
RETAIL	10	23920.9	24086.0	135.2	1.22	0.10
	20	22276.3	22481.1	235.3	0.87	0.18

k: number of clusters used in k-means; E: clustering error in the original dataset; mean: mean clustering error in the sampled datasets; std: standard deviation of the clustering error in the sampled datasets; Z: distance of E from mean, measured in standard deviations; p: empirical p-value.

- Dataset sizes: S1, S2: 1000 x 20, Courses 2405 x 5021,
  Paleo 124 x 139, Retail 88162 x 16470

Boostrapping

- Also called resampling; a slightly different idea
- A statistic of interest f, again
- Want to understand the distribution of f(D)
- Assume D = $\{x_1, x_2, \ldots, x_n\}$

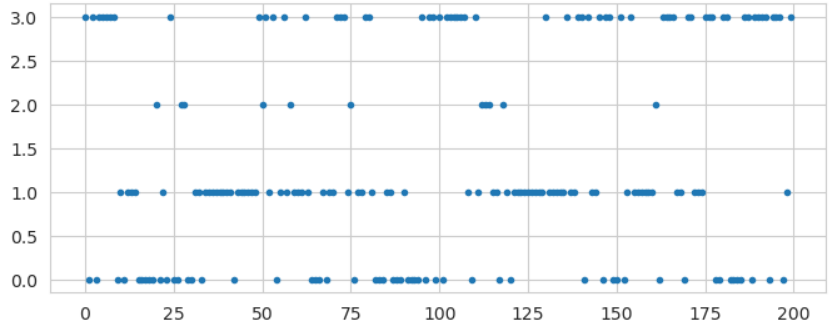
- Do many times:
  - Sample n entries from D with resampling, result D_j
  - Same entry can (and will) be many times in D_j
  - Compute $f(D_j)$
- Compute the average and std of $f(D_j)$'s

Back to sequences: another simple example

- Sentence embeddings
- A document with 740756 characters
- Look at fragments of size 500, no overlap; 1402 fragments
- Do sentence embedding (large language model style)
- For each fragment we get a 384-dimensional vector
- Is there some continuity in the vectors?
- Approach: cluster the vectors; see if we stay in the same cluster for longer time than for a randomized clustering

Approach A: cluster

- Cluster



- Count how many times we go from one cluster to another
- Sum of the diagonal: how many times stayed in the same

```
def changes(labels):

    k = np.max(labels)+1
    res = np.zeros((k,k))
    for i in range(len(labels)-1):
        res[labels[i],labels[i+1]] += 1
    return res
```

```
from sklearn.cluster import KMeans

for n_clusters in range(4,5):
    kmeans = KMeans(n_clusters=n_clusters, n_init='auto').fit( # , random_state=0).fit(
        embeddings
    )
    print(changes(kmeans.labels_))
    print(np.diag(changes(kmeans.labels_)).sum())
```

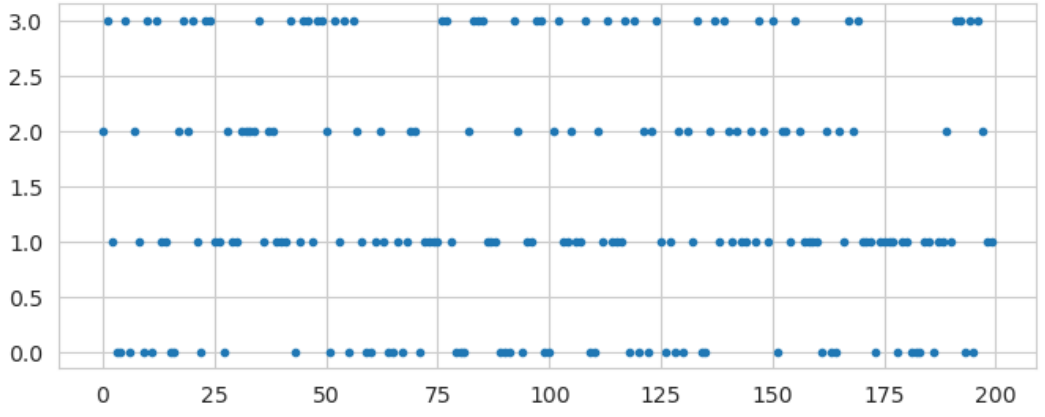
```
[[163.  87.  49.  85.]
 [ 84. 236.  67.  80.]
 [ 44.  62.  75.  44.]
 [ 94.  82.  34. 115.]]
589.0
```

Approach A: cluster

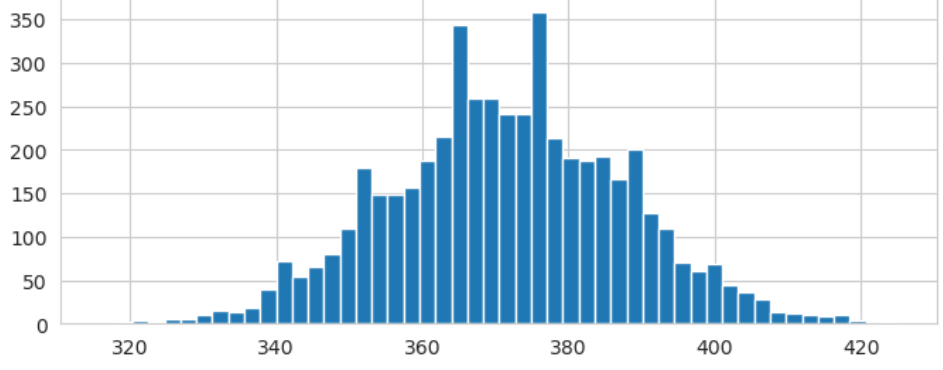
- Randomize the cluster labels

```
huu = np.random.permutation(kmeans.labels_)
plt.plot(huu[:200],'.')
```

```
[<matplotlib.lines.Line2D at 0x2ac360c0e710>]
```



- For each randomization count the sum of the diagonal
- For the true data the sum was 589



Wrapping up

- Randomization: a tool for assessing data mining results
- Widely applicable, but not a solution to every problem
- Problem 1: how to decide what to randomize?
  - Determines our null model – what are we comparing against
- Problem 2: how do we know that the randomization algorithm really works correctly
  - Not trivial
- Randomization approach does not remove the problems of multiple testing or garden of forking paths
- xkcd: Correlation