Using Blind Search and Formal Concepts for Binary Factor Analysis

Aleš Keprt

Dept. of Computer Science, FEI, VSB Technical University Ostrava, Czech Republic
ales.keprt@vsb.cz

Abstract. Binary Factor Analysis (BFA, also known as Boolean Factor Analysis) may help with understanding collections of binary data. Since we can take collections of text documents as binary data too, the BFA can be used to analyse such collections. Unfortunately, exact solving of BFA is not easy. This article shows two BFA methods based on exact computing, boolean algebra and the theory of formal concepts.

Keywords: Binary factor analysis, boolean algebra, formal concepts

1 Binary factor analysis

1.1 Problem definition

To describe the problem of Binary Factor Analysis (BFA) we can paraphrase BMDP's documentation (Bio-Medical Data Processing, see [1]).

BFA is a factor analysis of dichotomous (binary) data. This kind of analysis differs from the classical factor analysis (see [16]) of binary valued data, even though the goal and the model are symbolically similar. In other words, both classical and binary analysis use symbolically the same notation, but their senses are different.

The goal of BFA is to express p variables $X = (x_1, x_2, \ldots, x_p)$ by m factors $(F = f_1, f_2, \ldots, f_m)$, where $m \ll p$ (m is considerably smaller than p). The model can be written as

$$X = F \odot A$$

where $\odot$ is matrix multiplication. For n cases, data matrix X, factor scores F, and factor loadings A can be written as

$$\begin{bmatrix} x_{1,1} \ldots x_{1,p} \\ \vdots \ \ddots \ \vdots \\ x_{n,1} \ldots x_{n,p} \end{bmatrix} = \begin{bmatrix} f_{1,1} \ldots f_{1,m} \\ \vdots \ \ddots \ \vdots \\ f_{n,1} \ldots f_{n,m} \end{bmatrix} \odot \begin{bmatrix} a_{1,1} \ldots a_{1,p} \\ \vdots \ \ddots \ \vdots \\ a_{m,1} \ldots a_{m,p} \end{bmatrix}$$

where elements of all matrices are valued 0 or 1 (i.e. binary).

1.2 Difference to classical factor analysis

Binary factor analysis uses boolean algebra, so matrices of factor scores and loadings are both binary. See the following example: The result is 2 in classical algebra

$$[1\ 1\ 0\ 1] \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 0 = 2$$

but it's 1 when using boolean algebra.

$$[1\ 1\ 0\ 1] \cdot \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 1 \cdot 1 \oplus 1 \cdot 1 \oplus 0 \cdot 0 \oplus 1 \cdot 0 = 1$$

Sign $\oplus$ marks disjunction (logical sum), and sign $\cdot$ mars conjunction (logical conjunction). Note that since we focus to binary values, the logical conjunction is actually identical to the classic product.

In classical factor analysis, the score for each case, for a particular factor, is a linear combination of all variables: variables with large loadings all contribute to the score. In boolean factor analysis, a case has a score of one if it has a positive response for any of the variables dominant in the factor (i.e. those not having zero loadings) and zero otherwise.

1.3 Success and discrepancy

It is obvious, that not every X can be expressed as $F \odot A$. The success of BFA is measured by comparing the observed binary responses (X) with those estimated by multiplying the loadings and the scores ($\hat{X} = F \odot A$). We count both positive and negative discrepancies. Positive discrepancy is when the observed value (in X) is one and the analysis (in $\hat{X}$) estimates it to be zero, and reversely negative discrepancy is when the observed value is zero and the analysis estimates it to be one. Total count of discrepancies d is a suitable measure of difference between observed values $x_{i,j}$ and calculated values $\hat{x}_{i,j}$.

$$d = \sum_{i=1}^{n} \sum_{j=1}^{p} |\hat{x}_{i,j} - x_{i,j}|$$

1.4 Terminology notes

Let's summarize the terminology we use. Data to be analyzed are in matrix X. Its columns x_j represent **variables**, whereas its rows x_i represent **cases**. The

factor analysis comes out from the generic thesis saying that variables, we can observe, are just the effect of the factors, which are the real origin. (You can find more details in [16].) So we focus on factors. We also try to keep number of factors as low as possible, so we can say "reducing variables to factors".

The result is the pair of matrices. Matrix of **factor scores** F expresses the input data by factors instead of variables. Matrix of **factor loadings** A defines the relation between variables and factors, i.e. each row a_i defines one particular factor.

1.5 An example

As a basic example (see [1]) we consider a serological problem[1], where p tests are performed on the blood of each of n subjects (by adding p reagents). The outcome is described as positive (a value of one is assigned for the test in data matrix), or negative (zero is assigned). In medical terms, the scores can be interpreted as antigens (for each subject), and the loading as antibodies (for each test reagent). See [14] for more on these terms.

1.6 Application to text documents

BFA can be also used to analyse a collection of text documents. In that case the data matrix X is built up of a collection of text documents D represented as p-dimensional binary vectors d_i, $i \in 1, 2, \ldots, n$. Columns of X represent particular words. Particular cell $x_{i,j}$ equals to *one* when document i contains word j, and *zero* otherwise. In other words, data matrix X is built in a very intuitive way.

It should be noted that some kind of smart (i.e. semantic) preprocessing could be made in order to let the analysis make more sense. For example we usually want to take *world* and *worlds* as the same word. Although the binary factor analysis has no problems with finding this kind similarities itself, it is computationally very expensive, so any kind of preprocessing which can decrease the size of input data matrix X is very useful. We can also use WordNet, or thesaurus to combine synonyms. For additional details see [5].

2 The goal of exact binary factor analysis

In classic factor analysis, we don't even try to find 100% perfect solution, because it's simply impossible. Fortunately, there are many techniques that give a good suboptimal solution (see [16]). Unfortunately, these classic factor analysis techniques are not directly applicable to our special binary conditions. While classic techniques are based on the system of correlations and approximations,

[1] Serologic test is a blood test to detect the presence of antibodies against microorganism. See serology entry in [14].

these terms can be hardly used in binary world. Although it is possible to apply classic (i.e. non-boolean non-binary) factor analysis to binary data, if we really focus to BFA with restriction to boolean arithmetic, we must advance another way.

You can find the basic BFA solver in BMDP – Bio-Medical Data Processing software package (see [1]). Unfortunately, BMDP became a commercial product, so the source code of this software package isn't available to the public, and even the BFA solver itself isn't available anymore. Yet worse, there are suspicions saying that BMDP's utility is useless, as it actually just guesses the F and A matrices, and then only explores the similar matrices, so it only finds local minimum of the vector error function.

One interesting suboptimal BFA method comes from Húsek, Frolov et al. (see [15], [7], [6], [2], [8]). It is based on a Hopfield-like neural network, so it finds a suboptimal solution. The main advantage of this method is that it can analyse very large data sets, which can't be simply processed by exact BFA methods.

Although the mentioned neural network based solver is promising, we actually didn't have any one really exact method, which could be used to proof the other (suboptimal) BFA solvers. So we started to work on it.

3 Blind search based solver

The very basic algorithm blindly searches among all possible combinations of F and A. This is obviously 100% exact, but also extremely computational expensive, which makes this kind of solver in its basic implementation simply unusable.

To be more exact, we can express the limits of blind search solver in units of n, p and m. Since we need to express matrix X as the product of matrices $F \odot A$, which are $n \times m$ and $m \times p$ in size, we need to try on all combinations of $m \cdot (n + p)$ bits. And this is very limiting, even when trying to find only 3 factors from 10×10 data set ($m = 3$, $n = 10$, $p = 10$), we end up with computational complexity of $2^{m \cdot (n+p)} = 2^{60}$, which is quite behind the scope of current computers.

4 Revised blind search based solver

In order to make the blind search based solver more usable, we did several changes to it.

4.1 Preprocessing

We must start with optimizing data matrix. The optimization consist of these steps:

Empty rows or columns All empty rows and empty columns are removed, because they has no effect on the analysis. Similarly, the rows and columns full of one's can be removed too. Although removing rows and columns full of one's can lead to higher discrepancy (see sec. 1.3), it doesn't actually have any negative impact on the analysis.

Moreover, we can ignore both cases (rows) and variables (columns) with too low or too high number of one's, because they are usually not very important for BFA. Doing this kind of optimization can significantly reduce the size of data matrix (directly or indirectly, see below), but we must be very careful, because it can lead to wrong results. Removing too many rows and/or columns may completely degrade the benefit of exact BFA, because it leads to exact computing with inexact data. In regard to this danger, we actually implemented only support for removing columns with too low number of one's.

Duplicate rows and columns With duplicate rows (and columns resp.) are the ones which are the same to each other. Although this situation can hardly appear in classic factor analysis (meaning that two measured cases are 100% identical), it can happen in binary world much easier, and it really does. As for duplicate rows, the main reason of their existence is usually in the preprocessing. If we do some kind of semantic preprocessing, or even forcibly remove some columns with low number of one's, the same (i.e. duplicate) rows occur. We can remove them without negative impact to the analysis, if we remember the repeat-count of each row. We call it *multiplicity*.

Then we can update the discrepancy formulae (see sec. 1.3) to this form:

$$d = \sum_{i=1}^{n} \sum_{j=1}^{p} (m_i^R \cdot m_j^C |\hat{x}_{i,j} - x_{i,j}|)$$

where m_i^R and m_j^C are multiplicity values for row i and column j respectively.

We can also compute the *multiplicity matrix* M:

$$M = \begin{bmatrix} m_{1,1} \dots m_{1,p} \\ \vdots \quad \ddots \quad \vdots \\ m_{n,1} \dots m_{n,p} \end{bmatrix}$$

where $m_{i,j} = m_i^R \cdot m_j^C$. Although this leads to simpler and better readable formulae

$$d = \sum_{i=1}^{n} \sum_{j=1}^{p} (m_{i,j} |\hat{x}_{i,j} - x_{i,j}|)$$

it isn't a good idea, since the implementation is actually inefficient, since it needs a lot of additional memory ($n \cdot p$ numbers compared to $n + p$ ones).

The most important note is, that the merging of duplicate rows and columns lead to a significant reduction in computation time, and still doesn't bring any errors to the computation.

4.2 Bit-coded matrices

Using standard matrices is simple, because it is based on classic two-dimensional arrays and makes the source code well readable. In contrast, we also implemented the whole algorithm using bit-coded matrices and bitwise (truly boolean) operations (like and, or, xor). That resulted in not so nice source code, and also required some tricks, but also saved a lot of computation speed. We actually sped up the code by 20% by using bit-coded matrices and bitwise (boolean) operations.

4.3 The strategy

Although all the optimizations presented above lead to lower computation time, it is still not enough. To save yet more computation time, we need a good strategy.

The main problem is that we need to try too many bits in matrices F and A. Fortunately there exist a way of computing one of these matrices from the other one, thanks to knowing X. Since we are more concerned in A, we check out all bits in that one, and then find the right F. In summary:

1. Build up one particular candidate for matrix A.
2. Find the best F for this particular A.
3. Multiply these matrices and compare the result with X. If the discrepancy is smaller to the so far best one, remember this F, A pair.
4. Back to step 1.

After we go through all possible candidates for A, we're done.

4.4 Computing F from A and X

Symbolically, we can express this problem as follows. We are trying to find F and A, so $X = F \odot A$. Let we know X and A, so we only need to compute F. If we take a parallel from numbers, we can write something like $F = X/A$. Unfortunately, this operation isn't possible with common binary matrices.

If we bit-code matrices X and A on row-by-row basis, so $X = [x_1, \ldots, x_n]^T$ and $A = [a_1, \ldots a_p]^T$, then

$$x_i = \sum_{k=1}^{m} f_{i,k} \cdot a_j$$

From this formulae we can compute F on row-by-row basis, which significantly speeds up whole algorithm. The basic idea still relies on checking out all bit combinations for each row of F, which is $2^m \cdot m$ in total, but we can possibly find a better algorithms in future. In our implementation we compute discrepancy together with finding F, so we can abort the search whenever the

partial discrepancy is higher than the so far best solution. This way we get some speedup which could be made yet higher by pre-sorting rows of A by the discrepancies caused by particular rows, etc. Exploration of these areas isn't very important, because the possible speedup is quite scanty.

Note that in this place we can also focus to positive or negative discrepancy exclusively. It can be done using boolean algebra without any significant speed penalties.

5 Parallel implementation

The bind-search algorithm (including the optimized version presented above) can exploit the power of parallel computers (see [9]). We used PVM interface (Parallel Virtual Machine, see [4]) which is based on sending messages. The BFA blind search algorithm is very suitable for this kind of parallelism, because we just need to find a smart way of splitting the space of possible solutions to be checked out to a set of sub-spaces, and distribute them among available processor in our parallel virtual machine.

We tested this method using 2 to 11 PC desktop computers on a LAN (local area network). We managed to gain the absolute efficiency around 92%[2], which is very high compared to usual parallel programs. (The number 92% says that it takes 92% of time to run 11 consecutive runs on the same data, compared to a single run of the parallel version on the network of 11 computers).

6 Concept lattices

Another method of solving BFA problem is based on concept lattices. This section gives minimum necessary introduction to concept lattices, and especially *concepts*, which are the key part of the algorithm.

Definition 1 (Formal context, objects, attributes).
Triple (X, Y, R)*, where* X *and* Y *are sets, and* R *is a binary relation* $R \subseteq X \times Y$*, is called* **formal context***. Elements of* X *are called* **objects***, and elements of* Y *are called* **attributes***. We say "object* A *has attribute* B*", just when* $A \subseteq X$*,* $B \subseteq Y$ *and* $(A, B) \in R$*.* □

Definition 2 (Derivation operators).
For subsets $A \subseteq X$ *and* $B \subseteq Y$*, we define*

$$A^{\uparrow} = \{b \in B \mid \forall a \in A : (a, b) \in R\}$$
$$B^{\downarrow} = \{a \in A \mid \forall b \in B : (a, b) \in R\}$$

□

[2] It was measured in Linux, while Windows 2000 performed a bit worse and its performance surprisingly fluctuated.

In other words, $A^{\uparrow}$ is the set of attributes common to all objects of A, and similarly $B^{\downarrow}$ is the set of all objects, which have all attributes of B.

Note: We just defined two operators $^{\uparrow}$ and $^{\downarrow}$:

$$^{\uparrow} : P(X) \to P(Y)$$
$$^{\downarrow} : P(Y) \to P(X)$$

where $P(X)$ and $P(Y)$ are sets of all subsets of X and Y respectively.

Definition 3 (Formal concept).
Let (X, Y, R) be a formal context. Then pair (A, B), where $A \subseteq X$, $B \subseteq Y$, $A^{\uparrow} = B$ and $B^{\downarrow} = A$, is called **formal concept** *of (X, Y, R).*

Set A is called **extent** *of (A, B), and set B is called* **intent** *of (A, B).* □

Definition 4 (Concept ordering).
Let (A_1, B_1) and (A_2, B_2) be formal concepts. Then (A_1, B_1) is called subconcept of (A_2, B_2), just when $A_1 \subseteq A_2$ (which is equivalent to $B_1 \supseteq B_1$). We write $(A_1, B_1) \leq (A_2, B_2)$. Reversely we say, that (A_2, B_2) is superconcept of (A_1, B_1). □

In this article we just need to know the basics of concepts and their meaning. For more detailed, descriptive, and well understandable introduction to Formal Concept Analysis and Concept Lattices, see [3], [11] or [13].

7 BFA using formal concepts

If we want to speed up the simple blind-search algorithm, we can try to find some factor *candidates*, instead of checking out all possible bit-combinations. The technique which can help us significantly is Formal Concept Analysis (FCA, see [11]). FCA is based on concept lattices, but we actually work with formal concepts only, so the theory we need is quite simple.

7.1 The strategy

We can still use some good parts of the blind-search program (matrix optimizations, optimized bitwise operations using boolean algebra, etc.), but instead of checking out all possible bit combinations, we work with concepts as the factor candidates. In addition, we can adopt some strategy optimizations (as discussed above) to concepts, so the final algorithm is quite fast; its strength actually relies on the concept-building algorithm we use.

So the BFA algorithm is then as follows:

1. Compute all concepts of X. (We use a standalone program to do this.)
2. Import the list of concepts, and *optimize* it, so it correspond to our optimized data matrix X. (This is simple. We just merge objects and attributes the same way, as we merged duplicate rows and columns of X respectively.)
3. Remove all concepts with too many one's. (The number of one's per factor is one of our starting constraints.)
4. Use the remaining concepts as the factor candidates, and find the best m-element subset (according to discrepancy formulae).

This way we can find the BFA solution quite fast, compared to the blind search algorithm. Although the algorithm described here looks quite simple[3], there is a couple of things, we must be aware of.

7.2 More details

The most important FCA consequence is that 100% correct BFA solution can always be found among all subsets of concepts. This is very important, because it is the main guarantee of the correctness of the concept based BFA solver.

Other important feature of FCA based concepts is that they never directly generate any negative discrepancy. It is a direct consequence of FCA qualities, and affects the semantic sense of the result. As we discussed above (and see also [1]), negative discrepancy is a case when $F \odot A$ gives 1 when it should be 0. From semantic point of view, this (the negative discrepancy) is commonly unwanted phenomenon. In consequence, the fact that there's no negative discrepancy in the concepts, may have negative impact on the result, but the reality is usually right opposite. (Compare this to the quick sort phenomenon.)

The absence of negative discrepancies coming from concepts applies to A matrix only. It doesn't apply to F matrix, we still can use any suitable values for it. In consequence, we always start with concepts not generating negative discrepancy, which are semantically better, and end up with best suitable factor scores F, which give the lowest discrepancy. So it seems to be quite good feature.

7.3 Implementation issues

It's clear that the data matrix X is usually quite large, and makes the finding of the formal concepts the main issue. Currently we use the standalone CL (concept lattice) builder. It is optimized for finding concept lattices, but that's not right what we need. In the future, we should consider adopting some kind of CL building algorithm directly into BFA solver. This will save a lot of time

[3] *Everything's simple, when you know it.*

when working with large data sets, because we don't need to know the concept hierarchy.

We don't even need to know all the formal concepts, because the starting constraints limit the maximum number of one's in a factor, which is directly applicable to CL building.

8 Comparing the results and the computation times

The two algorithms presented in this article were tested on the test data suite taken from the neural algorithm mentioned above (see [15], [7], [6], [2], [8]). We focused to test data sets `p2` and `p3`, which are both 100×100 values in size, and differs in the ones' density. All three algorithms gave the same results, so they all appear to be correct (from this point of view).

Table 1. Computation times

data set	factors	one's	time (m:s)	discrepancy	notes
p3.txt	5	2–4	61:36	0	375 combinations
p3.txt	5	3	0:12	0	120 combinations
p3.txt	5	1–10	0:00	0	8/10 concepts
p2.txt	2	6	11:44	743	54264 combinations
p2.txt	5	1–10	0:07	0	80/111 concepts
p2.txt	5	6–8	0:00	0	30/111 concepts

The results are shown in table 8. Data set `p3` is rather simple, its factor loadings (particular rows of A) all have 3 one's. The first row in the table shows that it takes over 61 minutes to find these factors, when we search all combinations with 2, 3 or 4 one's per factor. If we knew that there are just 3 one's per factor, we can specify it as a constraint, and we get the result in just 12 seconds (see table 1, row 2). Indeed we usually don't know it in real situations.

Third row shows that when using formal concepts, we can find all factors in just 0 seconds, even when we search all possible combinations with 1 to 10 one's per factor. You can see the concept lattice in picture 1, with factors expressively circled.

Data set `p2` is much more complex, because it is created from factors containing 6 one's each. In this case the blind-search algorithm was able to find just 2 factors. It took almost 12 minutes, and discrepancy was 743. In addition, the two found factors are wrong, which is not a surprise according to the fact that there are actually 5 factors, and they can't be searched individually. It was not possible to find more factors using blind-search algorithm. Estimated times for computing 3 to 5 factors with the same constraints (limiting number of one's per factor to 6) are shown in table 8. It shows that it would take up to 3.5×10^9 years to find all factors. Unfortunately, we can't afford to wait so long...

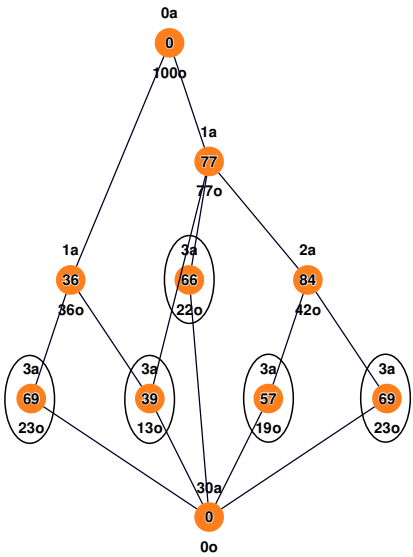

Fig. 1. Concept lattice of `p3` data set.

Table 2. Estimated computation times

data set	factors	one's	estimated time
p2.txt	3	6	440 days
p2.txt	4	6	65700 years
p2.txt	5	6	3.5×10^9 years

As you can see at the bottom of table 1, we can find all 5 factors of `p2` easily in just 7 seconds, searching among candidates containing 1 to 10 one's. The time can be reduced to 0 seconds once again, if we reduce searching to the range of 6 to 8 one's per factor. You can see the concept lattice in picture 1, with factors marked as well. As you can see, the factors are non-overlapping, i.e. they are not connected to each other. Note that this is not a generic nature. Generally, factors can arbitrarily overlap.

9 Conclusion

This article presented two possible algorithms for exact solving of Binary Factor Analysis. The work on them originally started as a simple blind search algorithm in order to check out the results of P8M of BMDP (see [1]), and the promising neural network solver (see [15], [7], [6], [2], [8]). As the work progressed, the theory of Concept Lattices and Concept Analysis was partially adopted into it, and it was with an inexpectably good results. For sure, the future work will more

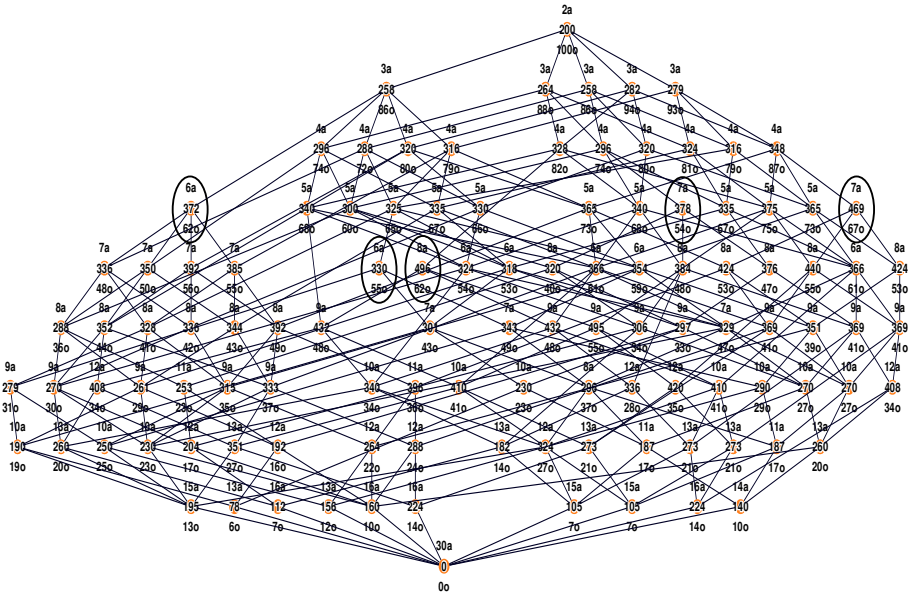

Fig. 2. Concept lattice of p2 data set.

focus on the possibilities of exploiting formal concepts and concept lattices for BFA.

References

1. *BMDP (Bio-Medical Data Processing).* A statistical software package. SPSS. http://www.spss.com/
2. A.A.Frolov, A.M.Sirota, D.Húsek, I.P.Muraviev, P.A.Polyakov: *Binary factorization in Hopfield-like neural networks: Single-step approximation and computer simulations.* 2003.
3. Bernhard Ganter, Rudolf Wille: *Formal Concept Analysis: Mathematical Foundations.* Springer–Verlag, Berlin–Heidelberg–New York, 1999.
4. Al Geist et al.: *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Massachusetts, USA, 1994.
5. Andreas Hotho, Gerd Stumme: *Conceptual Clustering of Text Clusters.* In Proceedings of FGML Workshop, pp. 37–45. Special Interest Group of German Informatics Society, 2002.
6. D.Húsek, A.A.Frolov, I.Muraviev, H.Řezanková, V.Snášel, P.Polyakov: *Binary Factorization by Neural Autoassociator.* AIA Artifical Intelligence and Applications - IASTED International Conference, Benalmádena, Málaga, Spain, 2003.
7. D.Húsek, A.A.Frolov, H.Řezanková, V.Snášel: *Application of Hopfield-like Neural Networks to Nonlinear Factorization.* Proceedings in Computational Statistics Compstat 2002, Humboldt-Universitt, Berlin, Germany, 2002.

8. D.Húsek, A.A.Frolov, H.Řezanková, V.Snášel, A.Keprt: *O jednom neuronovém přístupu k redukci dimenze.* In proceedings of Znalosti 2004, Brno, CZ, 2004. ISBN 80-248-0456-5.
9. Aleš Keprt: *Paralelní řešení nelineární booleovské faktorizace.* VŠB Technical University, Ostrava (unpublished paper), 2003.
10. Aleš Keprt: *Binary Factor Analysis and Image Compression Using Neural Networks.* In proceedings of WOFEX 2003, Ostrava, 2003. ISBN 80-248-0106-X.
11. Christian Lindig: *Introduction to Concept Analysis.* Hardvard University, Cambridge, Massachusetts, USA.
12. Christian Lindig: *Fast Concept Analysis.* Harvard University, Cambridge, Massachusetts, USA.
`http://www.st.cs.uni-sb.de/~lindig/papers/fast-ca/iccs-lindig.pdf`
13. Christoph Schwarzweller: *Introduction to Concept Lattices.* Journal Of Formalized Mathematics, volume 10, 1998. Inst. of Computer Science, University of Bialystok.
14. *Medical encyclopedia Medline Plus.* A service of the U.S. National Library of Medicine and the National Institutes of Health.
`http://www.nlm.nih.gov/medlineplus/`
15. A.M.Sirota, A.A.Frolov, D.Húsek: *Nonlinear Factorization in Sparsely Encoded Hopfield-like Neural Networks.* ESANN European Symposium on Artifical Neural Networks, Bruges, Belgium, 1999.
16. Karl Ueberla: *Faktorenanalyse* (2^{nd} edition). Springer–Verlag, Berlin–Heidelberg–New York, 1971. ISBN 3-540-04368-3, 0-387-04368-3.
(slovenský překlad: Alfa, Bratislava, 1974)