Algebraic Techniques for Analysis of Large Discrete-Valued Datasets

Mehmet Koyutürk [1], Ananth Grama [1], and Naren Ramakrishnan [2]

[1] Dept. of Computer Sciences, Purdue University,
W. Lafayette, IN, 47907, USA.
{koyuturk, ayg}@cs.purdue.edu
http://www.cs.purdue.edu/people/ayg

[2] Dept. of Computer Science, Virginia Tech.,
Blacksburgh, VA, 24061, USA.
naren@cs.vt.edu
http://people.cs.vt.edu/∼ramakris/

Abstract. With the availability of large scale computing platforms and instrumentation for data gathering, increased emphasis is being placed on efficient techniques for analyzing large and extremely high-dimensional datasets. In this paper, we present a novel algebraic technique based on a variant of semi-discrete matrix decomposition (SDD), which is capable of compressing large discrete-valued datasets in an error bounded fashion. We show that this process of compression can be thought of as identifying dominant patterns in underlying data. We derive efficient algorithms for computing dominant patterns, quantify their performance analytically as well as experimentally, and identify applications of these algorithms in problems ranging from clustering to vector quantization. We demonstrate the superior characteristics of our algorithm in terms of (i) scalability to extremely high dimensions; (ii) bounded error; and (iii) hierarchical nature, which enables multiresolution analysis. Detailed experimental results are provided to support these claims.

1 Introduction

The availability of large scale computing platforms and instrumentation for data collection have resulted in extremely large data repositories that must be effectively analyzed. While handling such large discrete-valued datasets, emphasis is often laid on extracting relations between data items, summarizing the data in an error-bounded fashion, clustering of data items, and finding concise representations for clustered data. Several linear algebraic methods have been proposed for analysis of multi-dimensional datasets. These methods interpret the problem of analyzing multi-attribute data as a matrix approximation problem. Latent Semantic Indexing (LSI) uses truncated singular value decomposition (SVD) to extract important associative relationships between terms (features) and documents (data items) [1]. Semi-discrete decomposition (SDD) is a variant of SVD, which restricts singular vector elements to a discrete set, thereby requiring less storage [11]. SDD is used in several applications ranging from LSI [10] and bump-hunting [14] to image compression [17], and has been shown to be effective in summarizing data.

The main objective of this study is to provide an efficient technique for error-bounded approximation of large discrete valued datasets. A non-orthogonal variant of SDD is adapted to discrete-valued matrices for this purpose. The proposed approach relies on successive discrete rank-one approximations to the given matrix. It identifies and extracts attribute sets well approximated by the discrete singular vectors and applies this process recursively until all attribute sets are approximated to within a user-specified tolerance. A rank-one approximation of a given matrix is estimated using an iterative heuristic approach similar to that of Kolda et al. [10]. This approach of error bounded compression can also be viewed as identifying dominant patterns in the underlying data.

Two important aspects of the proposed technique are (i) the initialization schemes; and (ii) the stopping criteria. We discuss the efficiency of different initialization schemes for finding rank-one approximations and stopping criteria for our recursive algorithm. We support all our results with analytical as well as experimental results. We show that the proposed method is superior in identifying dominant patterns while being scalable to extremely high dimensions.

In the next section, we briefly describe the problem and overview related research. Mathematical formulation and implementation of the algorithm are presented in Section 3. Performance of the algorithm in terms of accuracy and run-time scalability is discussed in Section 4. We conclude with a summary of results and ongoing research in Section 5.

2 Background and Related Research

An $m \times n$ rectangular matrix A can be decomposed into $A = U \Sigma V^T$, where U is an $m \times r$ orthogonal matrix, V is an $n \times r$ orthogonal matrix and Σ is an $r \times r$ diagonal matrix with the diagonal entries containing the singular values of A in descending order. Here r denotes the rank of matrix A. The matrix $\tilde{A} = u \sigma_1 v^T$ is a rank-one approximation of A, where u and v denote the first rows of matrices U and V respectively. If we think of a matrix as a multi-attributed dataset with rows corresponding to data items and columns corresponding to features, we can say that each 3-tuple consisting of a singular value σ_k, k^{th} row in U, and k^{th} row in V represents a pattern in A, whose strength is characterized by $|\sigma_k|$. The underlying data represented by matrix A is summarized by truncating the SVD of A to a small number of singular values. This method, used in Latent Semantic Indexing (LSI), finds extensive application in information retrieval [1].

Semi-discrete decomposition (SDD) is a variant of SVD, where the values of the entries in matrices U and V are constrained to be in the set {-1,0,1} [11]. The main advantage of SDD is the small amount of storage required since each vector component requires only 1.5 bits. In our algorithm, since we always deal with 0/1 valued attributes, vector elements can be further constrained to the set {0,1}, requiring only 1 bit of storage. SDD has been applied to LSI and shown to do as well as truncated SVD using less than one-tenth the storage [10]. McConnell and Skillicorn show that SDD is extremely effective in finding outlier clusters in datasets and works well in information retrieval for datasets containing a large number of small clusters [14].

A recent thread of research has explored variations of the basic matrix factorization theme. Hofmann [8] shows the relationship between the SVD and an aspect model involving factor analysis. This allows the modeling of co-occurence of features and data items indirectly through a set of *latent variables*. The solution to the resulting matrix factorization is obtained by expectation maximization (not by traditional numerical analysis). In [12], Lee and Seung impose additive constraints on how the matrix factors combine to model the given matrix; this results in what they call a 'non-negative matrix factorization.' They show its relevance to creating parts-based representations and handling polysemy in information retrieval.

Other work on summarizing discrete-attributed datasets is largely focused on clustering very large categorical datasets. A class of approaches is based on well-known techniques such as *vector-quantization* [4] and *k-means clustering* [13]. The *k-modes* algorithm [9] extends k-means to the discrete domain by defining new dissimilarity measures. Another class of algorithms is based on similarity graphs and hypergraphs. These methods represent the data as a graph or hypergraph to be partitioned and apply partitioning heuristics on this representation. Graph-based approaches represent similarity between pairs of data items using weights assigned to edges and cost functions on this similarity graph [3, 5, 6]. Hypergraph-based approaches observe that discrete-attribute datasets are naturally described by hypergraphs and directly define cost functions on the corresponding hypergraph [7, 16]. Formal connections between clustering and SVD are explored in [2]; this thread of research focuses on first solving a continuous clustering relaxation of a discrete clustering problem (using SVD), and then subsequently relating this solution back via an approximation algorithm. The authors assume that the number of clusters is fixed whereas the dimensionality and the number of data items could change.

Our approach differs from these methods in that it discovers naturally occurring patterns with no constraint on cluster sizes or number of clusters. Thus, it provides a generic interface to the problem which may be used for in diverse applications. Furthermore, the superior execution characteristics of our approach make it particularly suited to extremely high-dimensional attribute sets.

3 Proximus: A Framework for Error-bounded Compression of Discrete-Attribute Datasets

Proximus is a collection of algorithms and data structures that rely on modified SDD to find error-bounded approximations to discrete attributed datasets. The problem of error-bounded approximation can also be thought of as finding dense patterns in sparse matrices. Our approach is based on recursively finding rank-one approximations for a matrix A, i.e. finding two vectors x and y that minimize the number of nonzeros in the matrix $|A - xy^T|$, where x and y have size m and n respectively. The following example illustrates the concept:

Example 1

$$A = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} = xy^T$$

Here, vector y is the *pattern vector*, which is the best approximation for the objective (error) function given. In our case, this vector is [1 1 0]. Vector x is the *presence vector* representing the rows of A that are well approximated by the pattern described by y. Since all rows contain the same pattern in this rank-one matrix, x is a vector of all ones. We clarify the discussion with a slightly non-trivial example.

Example 2

$$A = \begin{bmatrix} 0\,1\,1\,0\,1 \\ 0\,0\,1\,0\,1 \\ 0\,0\,0\,1\,1 \\ 1\,0\,1\,0\,1 \end{bmatrix} \approx \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0\,0\,1\,0\,1 \end{bmatrix} = \begin{bmatrix} 0\,0\,1\,0\,1 \\ 0\,0\,1\,0\,1 \\ 0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,1 \end{bmatrix}$$

In this example, the matrix A is not a rank-one matrix as before. The pattern vector here is [0 0 1 0 1] and the corresponding presence vector is [1 1 0 1]. This presence vector indicates that the pattern is dominant in the first, second and fourth rows of A. A quick examination of the matrix confirms this. In this way, a rank-one approximation to a matrix can be thought of as decomposing the matrix into a pattern vector and a presence vector which signifies the presence of the pattern.

Using a rank-one approximation for the given matrix, we partition the row set of the matrix into sets A_0 and A_1 with respect to vector x as follows: the i^{th} row of the matrix is put into A_1 if the i^{th} entry of x is 1, it is put into A_0 otherwise. The intuition behind this approach is that the rows corresponding to 1's in the presence vector are the rows of a maximally connected submatrix of A. Therefore, these rows have more similar non-zero structures among each other compared to the rest of the matrix. This partitioning can also be interpreted as creating two new matrices A_0 and A_1. Since the rank-one approximation for A gives no information about A_0, we further find a rank-one approximation and partition this matrix recursively. On the other hand, we use the representation of the rows in A_1 in the pattern vector y to check if this representation is sufficient via some stopping criterion. If so, we decide that matrix A_1 is adequately represented by matrix xy^T and stop; else, we recursively apply the same procedure for A_1 as for A_0.

3.1 Mathematical Formulation

The problem of finding the optimal rank-one approximation for a discrete matrix can be stated as follows.

Definition 1 *Given matrix $A \in \{0,1\}^m \times \{0,1\}^n$, find $x \in \{0,1\}^m$ and $y \in \{0,1\}^n$ that minimize the error:*

$$||A - xy^T||_F^2 = |\{a_{ij} \in |A - xy^T| : a_{ij} = 1\}|. \tag{1}$$

In other words, the error for a rank-one approximation is the number of non-zero entries in the residual matrix. For example, the error for the rank-one approximation of Example 2 is 4. As discussed earlier, this problem can also be thought of as finding maximum connected components in a graph. This problem is known to be NP-complete and there exist no known approximation algorithms or effective heuristics in literature.

Here, we use a linear algebraic method to solve this problem. The idea is directly adopted from the algorithm for finding singular values and vectors in the computation of an SVD. It can be shown that minimizing $||A - xy^T||_F^2$ is equivalent to maximizing the quantity $x^T Ay/||x||_2^2||y||_2^2$ [11]. If we assume that y is fixed, then the problem becomes:

Definition 2 *Find $x \in \{0, 1\}^m$ to maximize $x^T s/||x||_2^2$, where $s = Ay/||y||_2^2$.*

This problem can be solved in $O(m + n)$ time as shown in the following theorem and corollary.

Theorem 1 *If the solution to problem of Defn. 2 has exactly J non-zeros, then the solution is*

$$x_j = \begin{cases} 1, \text{if } \ 1 \leq j \leq J \\ 0, \text{otherwise} \end{cases}$$

where the elements of s, in sorted order, are

$$s_{i_1} \geq s_{i_2} \geq ... \geq s_{i_m}.$$

The proof can be found in [15].

Corollary 1 *The problem defined in Defn. 2 can be solved in $O(m + n)$ time.*

Proof
The entries of s can be sorted via counting sort in $O(n)$ time as the entries of $||y||_2^2 s = Ay$ are bounded from above by n and have integer values. Having them sorted, the solution described in Theorem 1 can be estimated in $O(m)$ time since $J \leq m$, thus the corollary follows.□

The foregoing discussion also applies to the problem of fixing x and solving for y. The underlying iterative heuristic is based on the above theorem, namely we start with an initial guess for y, we solve for x and fix the resulting x to solve for y. We iterate in this way until no significant improvement can be achieved. The proposed recursive algorithm for summarizing a matrix can now be described formally as follows: Using a rank-one approximation, matrix A is split into two submatrices according to the following definition:

Definition 3 *Given a rank-one approximation, $A \approx xy^T$, a split of A with respect to this approximation is defined by two sub-matrices A_1 and A_0 where*

$$A(i) \in \begin{cases} A_1, \text{if } \ x(i) = 1 \\ A_0, \text{otherwise} \end{cases}$$

for $1 \leq i \leq m$. Here, $A(i)$ denotes the i^{th} row of A.

Then, both A_1 and A_0 are matrices to be approximated and this process continues recursively. This splitting-and-approximating process goes on until one of the following conditions holds.

- $h(A_1) < \epsilon$ where $h(A_1)$ denotes the hamming radius of A_1, i.e., the maximum of the hamming distances of the rows of A_1 to the pattern vector y. ϵ is a predetermined threshold.
- $x(i) = 1 \; \forall i$, i.e. all the rows of A are present in A_1.

If one of the above conditions holds, the pattern vector of matrix A_1 is identified as a dominant pattern in the matrix. The resulting approximation for A is represented as $\tilde{A} = UV^T$ where U and V are $m \times k$ and $n \times k$ matrices containing the presence and pattern vectors of identified dominant patterns in their rows respectively and k is the number of identified patterns.

3.2 Initialization of Iterative Process

While finding a rank-one approximation, initialization is crucial for not only the rate of convergence but also the quality of the solutions since a wrong choice can result in poor local optima. In order to have a feasible solution, the initial pattern vector should have a magnitude greater than zero, i.e., at least one of the entries in the initial pattern vector should be equal to one. Possible procedures for finding an initial pattern vector include:

- **All Ones:** Set all entries of the initial pattern vector to one. This scheme is observed to be poor since the solution converges to a rough pattern containing most of the rows and columns in the matrix.
- **Threshold:** Set all entries corresponding to columns that have nonzero entries more than a selected threshold to one. The threshold can be set to the average number of nozeros per column. This scheme can also lead to poor local optima since the most dense columns in the matrix may belong to different independent patterns.
- **Maximum:** Set only the entry corresponding to the column with maximum number of nonzeros to one. This scheme has the risk of selecting a column that is shared by most of the patterns in the matrix since it typically has a large number of nonzeros.
- **Partition:** Take the column which has nonzeros closest to half of the number of rows and select the rows which have a nonzero entry on this column. Then apply the *threshold* scheme taking only the selected rows into account. This approach initializes the pattern vector to the center of a roughly identified cluster of rows. This scheme has the nice property of starting with an estimated pattern in the matrix, thereby increasing the chance of selecting columns that belong to a particular pattern.

All of these schemes require $O(m+n)$ time. Our experiments show that *partition* performs best among these schemes as intuitively expected. We select this scheme to be the default for our implementation and the experiments reported in Section 4 are performed with this initialization. More powerful initialization schemes can improve the performance of the algorithm significantly. However, it is important that the initialization scheme must not require more than $\Theta(m+n)$ operations since it will dominate the runtime of the overall algorithm if it does.

3.3 Implementation Details

As the proposed method targets handling vectors of extremely high dimensions, the implementation and design of data structures are crucial for scalability, both in terms of time and space. In our implementation, we take advantage of the discrete nature of the problem and recursive structure of the proposed algorithm.

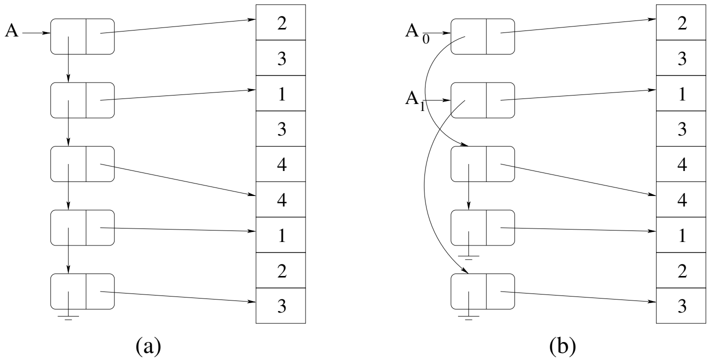


Fig. 1. Illustration of underlying data structure: (a) Original matrix (b) Resulting matrices after split, in row-compressed format.

Data Structures The discrete vectors are stored as binary arrays, i.e., each group of consecutive W (word size) entries are stored in a word. This allows us to reduce the memory requirement significantly as well as to take advantage of direct binary operations used in matrix-vector multiplications. The matrices are stored in a row-compressed format which fits well to the matrix-splitting procedure based on rows. Figure 1 illustrates the row-compressed representation of a sample matrix A and the result of splitting this matrix into A_0 and A_1, where

$$A = \begin{bmatrix} 0\,1\,1\,0 \\ 1\,0\,1\,1 \\ 0\,0\,0\,1 \\ 1\,1\,0\,0 \\ 0\,0\,1\,0 \end{bmatrix} \qquad A_0 = \begin{bmatrix} A(1) \\ A(3) \\ A(4) \end{bmatrix} = \begin{bmatrix} 0\,1\,1\,0 \\ 0\,0\,0\,1 \\ 1\,1\,0\,0 \end{bmatrix} \qquad A_1 = \begin{bmatrix} A(2) \\ A(5) \end{bmatrix} = \begin{bmatrix} 1\,0\,1\,1 \\ 0\,0\,1\,0 \end{bmatrix}$$

In this format, the column id's of the nonzero entries are stored in an array such that the non-zero entries of a row are stored in consequent locations. The list of rows of the matrix is a linked list in which each row has an additional pointer to the start of its non-zero entries in the non-zero list. Since the columns of the original matrix are never partitioned in our recursive implementation, the matrices appearing in the course of the algorithm can be easily created and maintained. While splitting a matrix, it is only necessary to split the linked list containing the rows of the matrix as seen in Figure 1(b). This is particularly important as splitting large sparse structures can be a very significant (and often dominant) overhead as we learnt from our earlier implementations.

Matrix Computations The heuristic used to estimate rank-one approximations necessitates the computation of Ax and $A^T y$ alternatingly. Although a row-compressed format is suitable for the computation of Ax, it is more difficult to perform the computation of $A^T y$ since each column of A is multiplied with y in this operation. However, it is not necessary compute A^T to perform this operation. In our implementation, we compute $s = A^T y$ with the following algorithm based on a row-compressed format:

initialize $s(i) = 0$ **for** $1 \leq i \leq n$
for $i \leftarrow 1$ **to** m **do**
  if $y(i) = 1$ **then**
    for $\forall j \in nonzeros(A(i))$ **do**
      $s(j) \leftarrow s(j) + 1$

This algorithm simply multiplies each row of A with the corresponding entry of y and adds the resulting vector to s and requires $O(nz(A))$ time in the worst-case.

This combination of restructured matrix transpose-vector product and suitable core data structures makes our implementation extremely fast and scalable.

Stopping Criteria As discussed in Section 3.1, one of the stopping criteria for the recursive algorithm is if each row in the matrix is well approximated by the pattern (i.e., the column singular vector is a vector of all ones). However, this can result in an undesirable local optimum. Therefore, in this case we check if $h(A) < \epsilon$, i.e., the hamming radius around the pattern vector for all the row vectors is within a prescribed bound. If not, we further partition the matrix into two based on hamming distance according to the following rule.

$$A(i) \in \begin{cases} A_1, \text{ if } \ h(A(i), y) < r \\ A_0, \text{ otherwise} \end{cases}$$

for $1 \leq i \leq m$. Here $h(A(i), y)$ denotes the hamming distance of row i to the pattern vector y and r is the prescribed radius of the virtual cluster defined by A_1. The selection of r is critical for obtaining the best cluster in A_1. In our implementation, we use an adaptive algorithm that uses a sliding window in n-dimensional space and detects the center of the most sparse window as the boundary of the virtual cluster.

4 Experimental Results

In order to illustrate the effectiveness and computational efficiency of the proposed method, we conducted several experiments on synthetic data specifically generated to test the methods on problems where other techniques typically fail (such as overlapping patterns, low signal-to-noise ratios). In this section we present the results of execution on a number of test samples to show the approximation capability of Proximus, analyze the structure of the patterns discovered by Proximus and demonstrate the scalability of Proximus in terms of various problem parameters.

4.1 Data Generation

Test matrices are generated by constructing a number of patterns, each consisting of several distributions (mixture models). Uniform test matrices consist of uniform patterns characterized by a set of columns that may have overlapping patterns. For example, the test matrix of Figure 2(a) contains four patterns with column sets of cardinality 16 each. A pattern overlaps with at most two other patterns at four columns, and the intersection sets are disjoint. This simple example proves to be extremely challenging for conventional SVD based techniques as well as k-means clustering algorithms. The SVD-based techniques tend to identify aggregates of overlapping groups as dominant singular vectors. K-means clustering is particularly susceptible for such examples to specific initializations.

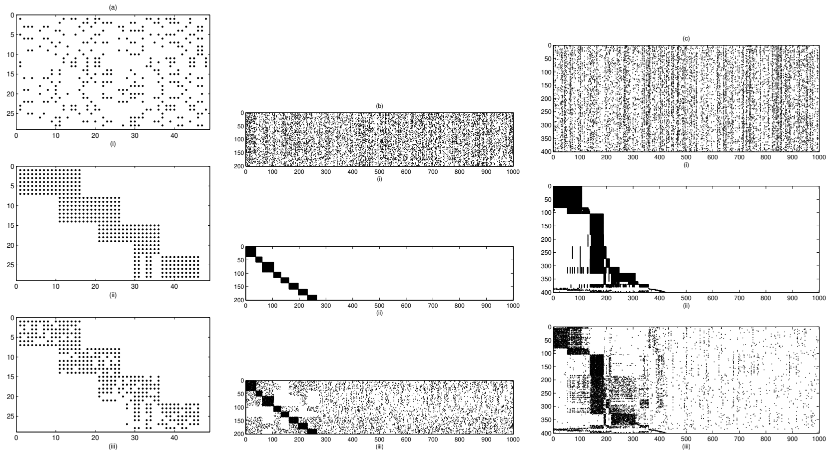
Gaussian matrices are generated as follows: a distribution maps the columns of the matrix to a Gaussian probability function of distribution $\mathcal{N}(\mu, \sigma)$, where $1 \leq \mu \leq n$ and σ are determined randomly for each distribution. Probability function $p(i)$ determines the probability of having a non-zero on the i^{th} entry of the pattern vector. A pattern is generated by superposing a number of distributions and scaling the probability function so that $\sum_{i=1}^{n} p(i) = E_{nz}$ where E_{nz} is the average number of non-zeros in a row, which is determined by $E_{nz} = \delta n$. δ is a pre-defined density parameter. The rows and columns of generated matrices are randomly ordered to hide the patterns in the matrix (please see first row of plots in Figure 2 for sample inputs).

4.2 Results

Effectiveness of Analysis As the error metric defined in the previous section depends on the nature of the pattern matrix, it does not provide useful information for evaluating the performance of the proposed method. Thus, we qualitatively examine the results obtained on sample test matrices. Figure 2(a) shows the performance of Proximus on a small uniform test matrix. The first matrix in the figure is the original generated and reordered matrix with 4 uniform patterns. The second matrix is the reordered approximation matrix which is estimated as XY^T where X and Y are 28x5 and 48x5 presence and pattern matrices containing the information of 5 patterns detected by Proximus. The 5^{th} pattern is characterized by the intersection of a pair of patterns in the original matrix. The matrix is reordered in order to demonstrate the presence (and extent) of detected patterns in input data.

The performance of Proximus on a simple Gaussian matrix is shown on Figure 2(b). In this example, the original matrix contains 10 patterns containing one distribution each and Proximus was able to detect all these patterns as seen in the figure.

Figure 2(c) shows a harder instance of the problem. In this case, the 10 patterns in the matrix contain at most 2 of 7 Gaussian distributions. The patterns and the distributions they contain are listed in Table 4.3(a). The matrix is of dimension 400×1000, and each group of 40 rows contain a pattern. As seen in the figure, Proximus was able to detect most of the patterns existing in the matrix. Actually, 13 significant patterns were identified by Proximus, most dominant 8 of which are displayed in Table 4.3(b). Each row of Table 4.3(b) shows a pattern detected by Proximus. The first column contains the number of rows conforming to that pattern. In the next 10 columns, these rows are classified into original patterns in the matrix. Similarly, the last 7 columns show the classification of these rows due to distributions in the original data. For example, 31 rows contain the same detected pattern shown in the second row of the table, 29 of which contain pattern P7 and other two contain pattern P8 originally. Thus, distributions D4 and D7, both having 29 rows containing them, dominated this pattern. Similarly, we



(a) (b) (c)

Fig. 2. Performance of Proximus on a (a) 28×48 uniform matrix with 4 patterns intersecting pairwise, (b) 200×1000 Gaussian matrix with 10 patterns each consist ing of one distribution; (c) 400×1000 Gaussian matrix with 10 patterns each consisting of at most 2 distributions.

can conclude that the third row is dominated by pattern P1 (distribution D2), and the fourth and seventh rows are characterized by distribution D5. The interaction between the most dominant distribution D5 and the distributions D1, D3, D4 and D6 shows itself in the first row. Although this row is dominated by D5, several other distributions are classified in this pattern since these distributions share some pattern with D5 in the original matrix. These results clearly demonstrate the power of Proximus in identifying patterns even for very complicated datasets.

4.3 Runtime Scalability

Theoretically, each iteration of the algorithm for finding a rank-one approximation requires $O(nz(A))$ time since a constant number of matrix-vector multiplications dominates the runtime of an iteration. As the matrices created during the recursive process are more sparse than the original matrix, the total time required to estimate the rank-one approximation for all matrices at a level in the recursion tree is asymptotically less than the time required to estimate the rank-one approximation for the initial matrix. Thus, the total runtime of the algorithm is expected to be $O(nz(A))$ with a constant depending on the number of dominant patterns in the matrix which determines the height of the recursion tree. The results displayed in Figure 3 illustrate the scalability of the algorithm in terms of number of columns, rows and non-zeros in the matrix. These experiments are performed by:

1. varying the number of columns, where number of rows and the average number of non-zeros per column are set to constant values 1000 and 50 respectively.
2. varying the number of rows, where number of columns and the average number of non-zeros per row are set to constant values 1000 and 50 respectively.
3. varying the number non-zeros, where the average non-zero density in rows is set to constant value 50 and the number of rows and columns are kept equal.

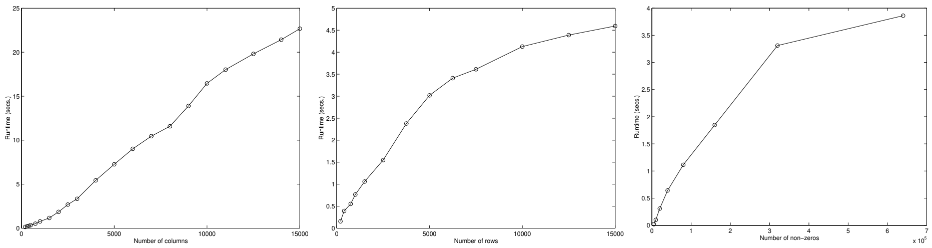
Pattern	Distributions
P1	D2
P2	D3, D6
P3	D1, D5
P4	D2, D7
P5	D5, D6
P6	D3, D5
P7	D4, D7
P8	D5
P9	D5
P10	D4, D5

(a)

Number of rows	Patterns										Distributions						
	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	D1	D2	D3	D4	D5	D6	D7
105			14		26	20	4	12	7	22	14		20	26	101	26	4
31							29	2						29	2		29
24	24											24					
24								8	16						24		
23	13			10								23					10
23	2	21										2	21			21	
22								15	7						22		
20			13	2		1					13	2		5	19		2

(b)

Table 1. (a) Description of patterns in the original data, (b) Classification of patterns detected by Proximus by original patterns and distributions.



(a) (b) (c)

Fig. 3. Runtime of Proximus (secs.) with respect to (a) number of columns (b) number of rows (c) number of non-zeros in the matrix.

All experiments are repeated with different randomly generated matrices 50 times for all values of the varying parameter. The reported values are the average run-times over these 50 experiments. In cases 1. and 2. above, the number of non-zeros grows linearly with the number of rows and columns, therefore, we expect to see an asymptotically linear runtime. As seen in Figure 3, the runtime of Proximus is asymptotically linear in number of columns. The runtime shows an asymptotically sublinear behavior with growing number of rows. This is because each row appears in at most one matrix at a level of the recursion tree. The behavior of runtime is similar when increasing the number of non-zeros.

5 Conclusions and Ongoing Work

In this paper, we have presented a powerful new technique for analysis of large high-dimensional discrete valued attribute sets. Using a range of algebraic techniques and data structures, this technique achieves excellent performance and scalability. The proposed analysis tool can be used in applications such as dominant and deviant pattern detection, collaborative filtering, clustering, bounded error compression, and classification. Efforts are currently under way to demonstrate its performance on real applications in information retrieval and in bioinformatics on gene expression data.

Acknowledgements

This work is supported in part by National Science Foundation grants EIA-9806741, ACI-9875899, ACI-9872101, EIA-9984317, and EIA-0103660. Computing equipment used for this work was supported by National Science Foundation and by the Intel Corp. The authors would like to thank Profs. Vipin Kumar at the University of Minnesota and Christoph Hoffmann at Purdue University for many useful suggestions.

References

1. M.W. Berry, S.T. Dumais, and G.W. O'Brien. Using Linear Algebra for Intelligent Information Retrieval. *SIAM Review*, Vol. 37(4):pages 573–595, 1995.
2. P. Drienas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering in Large Graphs and Matrices. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–299, 1999.
3. D. Gibson, J. Kleingberg, and P. Raghavan. Clustering Categorical Data: An Approach Based on Dynamical Systems. *VLDB Journal*, Vol. 8(3–4):pages 222–236, 2000.
4. R.M. Gray. Vector Quantization. *IEEE ASSP Magazine*, Vol. 1(2):pages 4–29, 1984.
5. S. Guha, R. Rastogi, and K. Shim. ROCK: A Robust Clustering Algorithm for Categorical Attributes. *Information Systems*, Vol. 25(5):pages 345–366, 2000.
6. G. Gupta and J. Ghosh. Value Balanced Agglomerative Connectivity Clustering. In *Proceedings of the SPIE conference on Data Mining and Knowledge Discovery III*, April 2001.
7. E.H. Han, G. Karypis, V. Kumar, and B. Mobasher. Hypergraph-Based Clustering in High-Dimensional Datasets: A Summary of Results. *Bulletin of the IEEE Technical Committee on Data Engineering*, Vol. 21(1):pages 15–22, March 1998.
8. T. Hofmann. Probabilistic Latent Semantic Indexing. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–57, 1999.
9. Z. Huang. A Fast Clustering Algorithm to Cluster Very Large Categorical Data Sets in Data Mining. In *Proceedings of the ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1997.
10. T.G. Kolda and D.P. O'Leary. A Semidiscrete Matrix Decomposition for Latent Semantic Indexing in Information Retrieval. *ACM Transactions on Information Systems*, Vol. 16(4):pages 322–346, October 1998.
11. T.G. Kolda and D.P. O'Leary. Computation and Uses of the Semidiscrete Matrix Decomposition. *ACM Transactions on Mathematical Software*, Vol. 26(3):pages 416–437, September 2000.
12. D.D. Lee and H.S. Seung. Learning the Parts of Objects by Non-Negative Matrix Factorization. *Nature*, Vol. 401:pages 788–791, 1999.
13. J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium*, volume 1, pages 281–297, 1967.
14. S. McConnell and D.B. Skillicorn. Outlier Detection using Semi-Discrete Decomposition. Technical Report 2001-452, Dept. of Computing and Information Science, Queen's University, 2001.
15. D.P. O'Leary and S. Peleg. Digital Image Compression by Outer Product Expansion. *IEEE Transactions on Communications*, Vol. 31(3):pages 441–444, 1983.
16. M. Ozdal and C. Aykanat. Clustering Based on Data Patterns using Hypergraph Models. *Data Mining and Knowledge Discovery*, 2001. Submitted for publication.
17. S. Zyto, A. Grama, and W. Szpankowski. Semi-Discrete Matrix Transforms (SDD) for Image and Video Compression. Purdue University, 2002. Working manuscript.