Mining Order-Preserving Submatrices Under Data Uncertainty: A Possible-World Approach

Ji Cheng*, Da Yan#, Xiaotian Hao*, Wilfred Ng*

*Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
{jchengac, xhao, wilfred}@cse.ust.hk
#Department of Computer Science, The University of Alabama at Birmingham
yanda@uab.edu

***Abstract*—Given a data matrix D, a submatrix S of D is an order-preserving submatrix (OPSM) if there is a permutation of the columns of S, under which the entry values of each row in S are strictly increasing. OPSM mining is widely used in real-life applications such as identifying coexpressed genes, and finding customers with similar preference. However, noise is ubiquitous in real data matrices due to variable experimental conditions and measurement errors, which makes conventional OPSM mining algorithms inapplicable. No previous work has ever combated uncertain value intervals using the *possible world semantics*.**

We establish two different definitions of significant OPSMs based on the *possible world semantics*: (1) expected support based and (2) probabilistic frequentness based. An optimized dynamic programming approach is proposed to compute the probability that a row supports a particular column permutation, and several effective pruning rules are introduced to efficiently prune insignificant OPSMs. These techniques are integrated into our two OPSM mining algorithms, based on prefix-projection and Apriori respectively. Extensive experiments on real microarray data demonstrate that the OPSMs found by our algorithms have a much higher quality than those found by existing approaches.

I. Introduction

Order-preserving submatrix (OPSM) mining is an important data mining problem which given a data matrix, discovers a subset of attributes (columns) over which a subset of tuples (rows) exhibit a similar pattern of rises and falls in the tuples' values. It is useful in many real applications such as bioinformatics and customer segmentation.

For example, in bioinformatics, when analyzing gene expression data from microarray experiments, genes (rows) with simultaneous rises and falls of mRNA expression levels across different time points (columns) may share the same cell-cycle related properties [17]; columns may also represent different experimental conditions as in [7]. In this application, an OPSM represents co-expressed patterns for large sets of genes, shared by a population of patients in a particular stage of a disease, or with the same drug treatment, etc. [3]. In fact, OPSM is well-known as the first bi-clustering method to overcome the drawback of clustering which cannot identify patterns that are common to only a part of the expression data matrix [3].

In recommender systems, we are often presented with a customer-product rating matrix where each row (resp. column) represents a customer (resp. a product), and an OPSM represents a group of users with a similar product preference. OPSM mining has also been successfully applied for analyzing

	t_1	t_2	t_3	t_4	
g_1	49	38	115	82	$t_2 < t_1 < t_4 < t_3$
g_2	67	96	124	48	$t_4 < t_1 < t_2 < t_3$
g_3	65	67	132	95	$t_1 < t_2 < t_4 < t_3$
g_4	81	115	133	62	$t_4 < t_1 < t_2 < t_3$

Figure 1: Gene Expression Matrix without Noise

indoor location tracking data [7], where visitors wearing RFID tags are tracked by RFID readers, and an OPSM represents a group of visitors who likely share a common visiting subroute.

Formally, OPSM mining considers a data matrix $D = (G, T)$ with a set of rows (e.g., genes) G and a set of columns (e.g., microarray tests) T. Each entry $D[g][t]$ of the matrix is a numerical value, e.g., the expression level of gene $g \in G$ under test $t \in T$. Consider the data matrix D shown in Figure 1. For simplicity, let us denote $D[g_i][t_j]$ by D_{ij}; then for row g_1, we have $D_{12} < D_{11} < D_{14} < D_{13}$, i.e., column value order is $t_2 < t_1 < t_4 < t_3$. Note that column value orders are also shown in Figure 1. Given a column permutation $t_1 < t_2 < t_3$, we can see from Figure 1 that rows g_2, g_3 and g_4 supports this permutation while g_1 does not (since $D_{12} < D_{11}$).

Formally, an OPSM of an $n \times m$ matrix D is a pair (G', P), where G' is a subset of G, and $P = (t_{i_1}, t_{i_2}, \ldots, t_{i_\ell})$ is a permutation of a subset of T, such that for any $g_j \in G'$, $D_{ji_1} < D_{ji_2} < \cdots < D_{ji_\ell}$. Here, we say that g supports P, and call P as the *pattern* of the OPSM. In Figure 1, (G', P) is an OPSM for $G' = \{g_2, g_3, g_4\}$ and $P = (t_1 < t_2 < t_3)$.

We are interested in those OPSMs with long patterns supported by sufficient rows, which exhibit statistical significance rather than occurring by chance. Given a row threshold τ_{row} and a column threshold τ_{col}, an OPSM (G', P) with $P = (t_{i_1} < t_{i_2} < \ldots < t_{i_\ell})$ is *significant* if $|G'| \geq \tau_{row}$ and $\ell \geq \tau_{col}$. We call ℓ as the length of pattern P. In other words, a significant OPSM has at least τ_{row} rows and τ_{col} columns.

OPSM Mining on Noisy Matrices. Real data are often noisy. For example, in microarray tests, each value in the matrix is a physical measurement that is subject to measurement errors, variable experimental conditions, and instrumental limitations [7]. Also, a customer usually rates a product using discrete scores (e.g., 1–5 stars), and even if two products both gain 4 stars, a customer may prefer one over another as each score actually represents a range of scores (e.g., $[3.5, 4.5)$).

	t_1	t_2	t_3	t_4
g_1	49, 55, 80	38, 51, 81	115, 101, 79	82, 110, 50
g_2	67, 54, 130	96, 85, 82	124, 92, 94	48, 37, 32
g_3	65, 49, 62	67, 39, 28	132, 119, 83	95, 89, 64
g_4	81, 83, 105	115, 110, 87	133, 108, 105	62, 52, 51

Figure 2: Matrix with Repeated Measurements

	t_1	t_2	t_3	t_4
g_1	[49, 80]	[38, 81]	[79, 115]	[50, 110]
g_2	[54, 130]	[82, 96]	[92, 124]	[32, 48]
g_3	[49, 65]	[28, 67]	[83, 132]	[64, 95]
g_4	[81, 105]	[87, 115]	[105, 133]	[51, 62]

Figure 3: Gene Expression Matrix with Intervals

Traditional OPSM mining algorithms are sensitive to such noise. For example, in Figure 1, if the value of D_{31} is slightly increased from 65 to 69, then g_3 will no longer support pattern $t_1 < t_2 < t_3$. One method to combat noise is **to sample each entry multiple times**, e.g., each microarray test can be repeated to record multiple measurements. Figure 2 illustrates a dataset with three repeated measurements (or replicates).

To handle such an expression data matrix, biologists usually take the average expression levels as the values in the matrix, to strike for higher data quality. OPSMRM [17] takes all the replicates into account, and produced higher-quality OPSMs than those mined from the averaged matrix. OPSMRM is based on the possible world semantics, which assumes that each matrix entry is a random variable taking the value of each replicate with equal probability. For example, in the matrix D shown in Figure 2, D_{11} is assumed to take value 49, 55 or 80 with $33.\dot{3}\%$ probability. The OPSM significance is defined only based on "expected support" (see Section II).

However, the data model of OPSMRM is restrictive. For D_{11} in Figure 2, if test t_1 is conducted on gene g_1 to get another measurement, the result is very likely to be a value between 49 and 80 (e.g. 60), but not any of 49, 55 and 80. To address this issue, [7] proposes the POPSM model which converts the replicates for each entry in the matrix into an interval, and produced higher-quality OPSMs than OPSMRM. Figure 3 shows this **interval-based data model** for the data matrix shown in Figure 2. The interval model is sometimes even the only choice for representing data uncertainty, such as in [7]'s RFID location tracking application where each row g of a matrix represents a loop-free object trajectory, each column t represents an RFID reader (i,e,, a location), and each entry $D[g][t]$ records the time interval when g is detected by reader t. There, location (or subroute) uncertainty is generated when two readers detect the same object at the same time.

However, POPSM is not defined based on the possible world semantics. Since possible world semantics is a robust probability model that has been proven to be effective in handling data uncertainty in various applications, this paper studies how to mine OPSMs with the **interval-based uncertain model** based on the well-established **possible world semantics**, which is shown to generate higher-quality OPSMs than POPSM (c.f. Section VI). Our contributions are summarized as follows:

- This is the first work that studies OPSM mining when matrix entry is modeled with interval and pattern significance is evaluated based on possible world semantics. This model is more robust than both OPSMRM and POPSM, and is shown to generate higher-quality OPSMs.
- Under the possible world semantics, we study two different definitions of OPSM significance: (S1) expected support and (S2) probabilistic frequentness. Note that OPSMRM only considers "(S1)" while POPSM does not even follow the possible world semantics.
- A basic operation in our mining problem is to compute the probability that a row g supports a pattern $P = (t_{i_1} < t_{i_2} < \ldots < t_{i_\ell})$, which is challenging since intervals $D[g][t_i]$ may overlap. We propose a dynamic programming (DP) algorithm to efficiently compute this probability. We further design a smart pay-as-you-go method to reuse DP computation when growing patterns.
- Once the above probability is computed for all rows, we then propose efficient algorithms to check the significance of a pattern P under both "(S1)" and "(S2)". A few efficient pruning rules are checked to prune insignificant pattern P before the more expensive significance check for P. The algorithm for "(S2)" is non-trivial and an efficient Fast Fourier Transform (FFT) algorithm is designed.
- Using the above algorithms to check the significance of each pattern P, two pattern-growth based OPSM mining algorithms are proposed using different techniques, (1) prefix-projection and (2) Apriori. Both algorithms have pros and cons, but they both output higher-quality OPSMs than existing approaches using comparable running time if not better, as verified in our experiments.

Paper Organization. Section II formally defines significant OPSMs under our interval-based uncertain data model, using expected support and probabilistic frequentness. Section III presents our dynamic programming algorithm to compute row supporting probability for a pattern P, and the pay-as-you-go technique for computation reuse. Given the row supporting probabilities for P, Section IV presents our algorithm for evaluating significance of pattern P and rules for pruning insignificant patterns. Then, Section V introduces our complete mining algorithms that grow patterns and examines their significance. Finally, Section VI empirically compares our algorithms with existing algorithms, Section VII reviews the related work, and Section VIII concludes the paper.

Due to the space limitation, we maintain an online appendix [1] to put proofs of theorems and additional experimental results: http://www.cs.uab.edu/yanda/papers/opsm.pdf.

II. Problem Definition

We assume that different rows of a data matrix are independent, and for each row, its different column entries are independent; we shall justify these assumptions in Sections III and IV. We mine significant OPSMs in two steps: (1) finding the frequent patterns with length at least τ_{col}, and (2) selecting the rows that support each frequent pattern.

Above all, we need to first define pattern frequentness under the interval-based uncertain model. Given a pattern $P = (t_{i_1} < t_{i_2} < \ldots < t_{i_\ell})$, for each matrix row g, let us denote the probability that g supporting P by p_g:

$$p_g = Pr\{\text{row } g \text{ supports pattern } P\}$$

We call p_g the *supporting probability* hereafter, and we will discuss the computation of p_g in Section III. To decide whether pattern P is frequent, we evaluate it using (i) the supporting probabilities of all rows, and (ii) the row threshold τ_{row}.

We define pattern significance using "expected support" and "probabilistic frequentness", two well-established semantics for defining pattern frequentness in mining uncertain data [14]. They both follow the possible world semantics.

Expected Support. Let us consider the expected number of rows that support pattern P. For each row g, we define a random variable X_g as follows:

$$X_g = \begin{cases} 1 & \text{if } g \text{ supports } P \\ 0 & \text{otherwise} \end{cases}$$

Obviously, X_g follows the Bernoulli distribution and its expectation $E(X_g) = p_g$. The number of rows that support pattern P is a random variable $X = \sum_{g \in G} X_g$, and we have

$$E(X) = E(\sum_{g \in G} X_g) = \sum_{g \in G} E(X_g) = \sum_{g \in G} p_g$$

Therefore, the expected support is simply the summation of p_g for all rows $g \in G$, and pattern P is frequent if and only if its expected support is not smaller than τ_{row}.

Probabilistic Frequentness. "Expected support" does not consider the distribution of X but merely its expectation. To be more accurate, "probabilistic frequentness" (or *p-frequentness*) considers the probability mass function (PMF) of X.

Given a matrix D with row set $G = \{g_1, g_2, \ldots, g_n\}$, the support of P is depicted by the PMF of X, denoted as $f_P(c)$ where $c = 0, 1, \cdots, n$:

$$f_P(c) = Pr\{c \text{ rows in } D \text{ support pattern } P\}$$

The PMF $f_P(c)$ can be computed using p_g of all rows $g \in G$ with the realistic assumption that rows are independent of each other, and we shall present more details in Section IV.

Let us denote the cumulative distribution function (CDF) by $F_P(c) = \sum_{i=0}^{c} f_P(i)$. Given a user-specified probability confidence threshold τ_{prob}, pattern P is probabilistically frequent (or *p-frequent*) if and only if

$$Pr\{X \geq \tau_{row}\} \ \geq \ \tau_{prob}, \tag{1}$$

where the L.H.S. can be represented as

$$Pr\{X \geq \tau_{row}\} = \sum_{c=\tau_{row}}^{n} f_P(c) = 1 - F_P(\tau_{row} - 1) \tag{2}$$

If we find that a pattern P is frequent, we only output P if its length is at least τ_{col}.

Row Selection. The second step of OPSM mining is to select the rows that support each frequent pattern. Since the rows are considered as independent to each other, row g is favored over row g' if supporting probability $p_g > p_{g'}$.

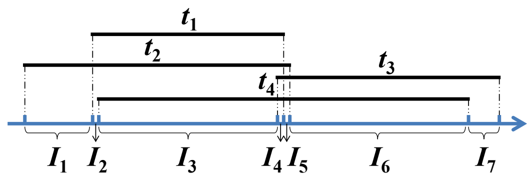

Figure 4: Preprocessing of Row g_1 in Figure 3

There are several possible methods for selecting rows into an OPSM: (a) selecting k rows whose supporting probability are the highest, where k can be set as the expected support $\sum_{g \in G} p_g$, or as the row threshold τ_{row}, or as any user-specified value; (b) selecting all rows whose supporting probability is at least τ_{cut}, where τ_{cut} is a user-specified "inclusion threshold" (different from τ_{prob} in "p-frequentness"). Like [7], we adopt the latter approach for row selection.

III. Supporting Probability Computation

In this section, we introduce how we compute the probability that a row g supports a pattern P, i.e., $p_g(P)$. We abbreviate it as p_g when P is clear from the context.

We compute $p_g(P)$ by dynamic programming. For ease of presentation, let us first assume that each element value in a data matrix follows uniform distribution within an interval. We will extend our dynamic programming method to handle arbitrary value distribution at the end of this section.

A. Preprocessing

Before applying the dynamic programming algorithm, we first need to preprocess each row $g = \langle [\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_m, r_m] \rangle$ to obtain a set of subintervals demarcated by ℓ_i, r_i ($i = 1, \ldots, m$). These subintervals, denoted by $I_1, \ldots, I_s$, are ordered in increasing order of value. Note that $s \leq 2m$. We also denote the interval of I_i as $[\ell(I_i), r(I_i)]$.

Figure 4 shows the subintervals obtained by preprocessing row g_1 in the data matrix in Figure 3: $I_1 = [38, 49]$, $I_2 = [49, 50]$, $I_3 = [49, 79]$, $I_4 = [79, 80]$, $I_5 = [80, 81]$, $I_6 = [81, 110]$ and $I_7 = [110, 115]$.

Obviously, for each row g, given an interval element $D[g][t]$ and a subinterval I_i, we must have: either (1) $I_i \subseteq [\ell_t, r_t]$, or (2) $I_i \cap [\ell_t, r_t] = \emptyset$ or $\{\ell_t\}$ or $\{r_t\}$. For example, in Figure 4, $I_2 \subseteq t_1$, $I_2 \subseteq t_2$, $I_2 \cap t_3 = \emptyset$, and $I_2 \cap t_4 = r(I_2) = \ell_4 = \{50\}$.

Since each $D[g][t]$ is assumed to be uniform, we have:

Property 1. *Let $f_t(x)$ be the PDF of $D[g][t]$ on subinterval I_i, then we have*

$$f_t(x) = \begin{cases} 1/(r_t - \ell_t) & \text{if } I_i \subseteq [\ell_t, r_t], \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Therefore, we can obtain the constant probability density of $D[g][t]$ on any subinterval I_i in $O(1)$ time, by checking whether $I_i \subseteq [\ell_t, r_t]$. Since $f_t(x)$ is constant on I_i, we abbreviate it as f_t from now on. For example, in Figure 4, consider g_1 and subinterval I_2: $f_1 = 1/31$ since $I_2 \subseteq [49, 80]$, while $f_4 = 0$ except at the boundary 50 (which is immaterial for continuous distribution). As we shall see next, Property 1 is critical to the efficiency of computing supporting probability.

Assume data matrix D is $n \times m$, preprocessing each row into intervals $I_1, \ldots, I_s$ requires sorting $s \leq 2m$ values with the cost of $O(s \log s)$ time. Overall, processing the n rows of D takes $O(ns \log s) = O(nm \log m)$.

B. Dynamic Programming Formulation

Given a row g, a pattern $P = (t_{i_1} < t_{i_2} < \ldots < t_{i_\ell})$, and the subintervals $I_1, I_2, \ldots, I_s$ by preprocessing g, we compute $p_g(P)$ using dynamic programming (DP) as follows. We abuse the notation t_i to mean the interval $D[g][t_i]$ since g is given.

The DP algorithm first creates a 2D array A with ℓ rows and s columns. The element $A[j][k]$ denotes the probability of the event $t_{i_1} < \ldots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j}$ located in the interval consisting of the first k subintervals, i.e., $\bigcup_{i=1}^{k} I_i$. Note that the value of $A[\ell][s]$ is exactly $p_g(P)$.

Our algorithm assumes that different column intervals of a row are independent, which is natural since different microarray tests or RFID readers are usually independent. This assumption is used in many places below, such as the proof of Theorem 1, and the last term's product in Eq (6).

Probability Evaluation on One Subinterval. Before describing the recursive formula for computing $A[j][k]$, we first explain how to compute the probability of the event $t_{i_1} < t_{i_2} < \ldots < t_{i_\ell}$ with $t_{i_1}, \ldots, t_{i_\ell}$ located in I_k. We denoted this probability by $P_{I_k}(t_{i_1} < \ldots < t_{i_\ell})$.

Theorem 1. *Given an interval $[\ell, r]$, let $\Delta = r - \ell$. If we have a set of random variables $x_1, x_2, \ldots, x_n$, where each variable x_i has constant probability density p_i on $[\ell, r]$, then $Pr\{(x_1, \ldots, x_n \in [\ell, r]) \wedge (x_1 < x_2 < \ldots < x_n)\} = (\prod_{i=1}^{n} p_i) \cdot \frac{\Delta^n}{n!}$.*

Proof. See Appendix B [1]. □

Theorem 1 implies the following corollary for row g:

Corollary 1. *Let us define $\Delta_k = r(I_k) - \ell(I_k)$, and let f_{t_i} be the probability density of $D[g][t_i]$ on subinterval I_k as computed by Eq (3). Then,*

$$P_{I_k}(t_{i_1} < t_{i_2} < \ldots < t_{i_j}) = \left(\prod_{z=1}^{j} f_{t_{i_z}} \right) \cdot \frac{\Delta_k^j}{j!}. \quad (4)$$

Evaluation of $A[j][k]$. Now, we are ready to present the recursive formula for computing $A[j][k]$. Let us first consider the base case when $k = 1$. In this case,

$$A[j][1] = P_{I_1}(t_{i_1} < t_{i_2} < \ldots < t_{i_j}), \quad (5)$$

which can be computed using Eq (4).

When $k > 1$, the event "$t_{i_1} < t_{i_2} < \ldots < t_{i_j}$ with $t_{i_1}, t_{i_2}, \ldots, t_{i_j} \in \bigcup_{i=1}^{k} I_i$" can be decomposed into the following disjoint events:

- $t_{i_1} < \ldots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j} \in \bigcup_{i=1}^{k-1} I_i$;
- $t_{i_1} < \ldots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j} \in I_k$;
- $t_{i_1} < \ldots < t_{i_z}$ with $t_{i_1}, \ldots, t_{i_z} \in \bigcup_{i=1}^{k-1} I_i$, and $t_{i_{z+1}} < \ldots < t_{i_j}$ with $t_{i_{z+1}}, \ldots, t_{i_j} \in I_k$, where z can take values $1, 2, \ldots, k-1$.

According to the above discussion, we obtain

$$\begin{aligned} A[j][k] &= A[j][k-1] + P_{I_k}(t_{i_1} < \ldots < t_{i_j}) + \\ & \sum_{z=1}^{j-1} A[z][k-1] \cdot P_{I_k}(t_{i_{z+1}} < \ldots < t_{i_j}) \end{aligned} \quad (6)$$

In fact, if we define $A[j][0] = 0$ for any j, then we can even compute $A[j][1]$ using Eq (6).

Note that computing $A[j][k]$ involves:

- The values $A[1][k-1], \ldots, A[j][k-1]$, which should have already been computed;
- Computing $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$ for $z = 1, \ldots, j$.

According to Eq (4), computing $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$ takes $O(j-z+1)$ time. Thus, computing $A[j][k]$ takes $\sum_{z=1}^{j} O(j-z+1) = O(j^2)$ time if we compute each $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$ individually.

We can actually compute $A[j][k]$ in $O(j)$ time as follows. From Eq (4), we can derive the following recursive formula for computing $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$:

$$\begin{aligned} & P_{I_k}(t_{i_z} < \ldots < t_{i_j}) \\ = & \begin{cases} f_{t_{i_j}} \cdot \Delta_k & \text{if } z = j \\ \frac{f_{t_{i_z}} \cdot \Delta_k}{j-z+1} \cdot P_{I_k}(t_{i_{z+1}} < \ldots < t_{i_j}) & \text{if } z < j \end{cases} \end{aligned} \quad (7)$$

Therefore, if we compute $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$ with z from j down to 1, then each $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$ can be computed from $P_{I_k}(t_{i_{z+1}} < \ldots < t_{i_j})$ using Eq (7) in O(1) time. Thus, computing $A[j][k]$ takes $\sum_{z=1}^{j} O(1) = O(j)$ time.

Accordingly, computing $p_g(P)$ requires computing all elements in the $\ell \times s$ array A, which takes $\sum_{j=1}^{\ell} \sum_{k=1}^{s} O(j) = O(\ell^2 s) = O(\ell^2 m)$ time (recall that $s \leq 2m$). We can further optimize the computation: if we find $f_{t_{i_z}} = 0$ when evaluating $P_{I_k}(t_{i_z} < \ldots < t_{i_j})$, then we can terminate early since according to Eq (4), $P_{I_k}(t_{i_{z'}} < \ldots < t_{i_j}) = 0$ for any $z' \leq z$.

C. Pay-as-you-go Approach

From now on, let us call the array for dynamic programming as DP-array. We mine patterns by pattern-growth, i.e., pattern $t_{i_1} < \ldots < t_{i_j} < t_{i_{j+1}}$ is checked after pattern $t_{i_1} < \ldots < t_{i_j}$.

We now consider how to **reuse** the DP-array for $t_{i_1} < \ldots < t_{i_j}$ to compute the DP-array for $t_{i_1} < \ldots < t_{i_j} < t_{i_{j+1}}$.

In the base case, the pattern is a singleton t_i. Assume that the interval of $D[g][t_i]$ is $[\ell_i, r_i]$, then there is only one subinterval $I_1 = [\ell_i, r_i]$ and the DP-array is a 1×1 array with $A[1][1] = Pr_{I_1}(t_i) = 1$.

We now consider how to incrementally compute the DP-array of a pattern grown by one more interval. Referring to the data matrix in Figure 3 again, and let us focus on the computation of $p_{g_1}(t_3 < t_4 < t_1)$.

Figure 5 illustrates the evaluation process, where the array on the left is the DP-array for pattern $t_3 < t_4$ which is already computed, and the array on the right is the DP-array for pattern $t_3 < t_4 < t_1$ that is to be computed. The split points of

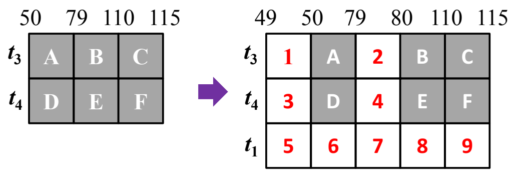

Figure 5: Pay-as-you-go DP-Array Evaluation

subintervals are also marked above the DP-arrays. Note that $A[j][k] = Pr\{(t_{i_1} < \ldots < t_{i_j}) \wedge (t_{i_1}, \ldots, t_{i_j} < r(I_k))\}$. For example, the value of cell E in the left array is the probability of the event $t_3 < t_4$ with $t_3, t_4 < 110$. Obviously, the value of cell E in the right array is exactly the same. In fact, we can copy the values of cells A–F in the left array directly into the corresponding cells in the right array.

On the other hand, for pattern $t_3 < t_4 < t_1$, the introduction of t_1 with interval $[49, 80]$ adds two more split points. For example, $I_2 = [79, 110]$ of the left array is now split into two subintervals for the right array: $I_3 = [79, 80]$ and $I_4 = [80, 110]$. However, the value of cell 4 (the probability of the event $t_3 < t_4$ with $t_3, t_4 < 80$), for example, is not computed in the left array, and therefore it has to be computed using Eq (6). In fact, the values of cells 1–9 have to be computed over the right array using Eq (6).

The complete algorithm is exactly like DP but those already-computed entries in the DP-array are directly copied without recomputation. Algorithm details can be found in Appendix A [1].

We now analyze the cost of this incremental computation of $p_g(t_{i_1} < \ldots < t_{i_j})$. Since at most two new split points ℓ_{i_j} and r_{i_j} are introduced when a pattern is grown with t_{i_j}, for each of the first $(j-1)$ rows in the DP-array, there are at most 2 elements to compute using dynamic programming. Together with the new j-th row, there are totally $2(j-1) + (s-1) = O(j)$ elements (note that the number of split points $s < 2j$) to compute using dynamic programming, which takes $O(j^2)$ time. The remaining $(j-1) \cdot s$ elements are copied from the old DP-array, which takes $O(j \cdot s)$ time. Therefore, the total time complexity is $O(j^2)$, quadratic to the pattern length j.

For a pattern P of length ℓ, time complexity of computing $p_g(P)$ is reduced from $O(\ell^2 m)$ in Section III-B to $O(\ell^2)$ here.

D. Extension to Arbitrary Distributions

We have been assuming that each matrix entry conforms to a uniform distribution defined over its interval, which is proper when there are only several replicates. When there are sufficient number of replicates, we can infer the underlying distribution (e.g., Gaussian distribution) and learn the parameters from the replicates. Then, we can discretize the distribution using an equi-depth histogram. Note that our dynamic programming framework is still applicable here, though each entry now may introduce more than 2 split points.

IV. Pattern Significance Checking

In Section III, we have already described how to compute the supporting probability of row g for pattern P (i.e., $p_g(P)$). This section explains how to evaluate the frequentness of a pattern P given the supporting probabilities all rows.

As discussed in Section II, it is straightforward to check whether a pattern P is frequent in terms of *expected support*: we just check whether $\sum_{g \in G} p_g(P) \geq \tau_{row}$ which takes $O(n)$ time. Therefore, this section focuses on explaining how we determine whether a pattern P is *probabilistically frequent*.

We assume that different rows (e.g., genes, customers/visitors) in the data matrix are independent of each other. This is a reasonable assumption similar to tuple independence in uncertain databases and data point independence in machine learning, and simplifies probability computations.

A. Pattern Support PMF Computation

We define the support of a pattern P in data matrix $D = (G, T)$ as the number of rows in G that supports P. Since each row g supports P only with probability $p_g(P)$, the support of P is a random variable, denoted as X.

We now consider how to compute the PMF of X, using the supporting probabilities $p_g(P)$ of all rows $g \in G$.

Naïve Method. Let $f_i(c)$ be the PMF of the support of P in matrix D_i which consists of the first i rows in D, where $c = 0, 1, \cdots, i$ ($f_i(c) = 0$ for other values of c). Then, we have the following recursive formula:

$$f_{i+1}(c) = p_{g_{i+1}} \cdot f_i(c-1) + (1 - p_{g_{i+1}}) \cdot f_i(c). \tag{8}$$

This is because P's support in D_{i+1} is c, iff (1) P's support in D_i is $(c-1)$ and g_{i+1} supports P, or (2) P's support in D_i is c and g_{i+1} does not support P. Note that Eq (8) holds only for $c = 1, \ldots, i$, and the products of probabilities are due to row independence (the same for later equations and thus omitted).

When $c = 0$ we have

$$f_{i+1}(0) = (1 - p_{g_{i+1}}) \cdot f_i(0), \tag{9}$$

since the support of P in D_{i+1} is 0, if and only if the support of P in D_i is 0 and g_{i+1} does not support P.

When $c = i + 1$ we have

$$f_{i+1}(i+1) = p_{g_{i+1}} \cdot f_i(i), \tag{10}$$

since the support of P in D_{i+1} is $(i+1)$, if and only if the support of P in D_i is i and g_{i+1} supports P.

Furthermore, we have the base case

$$f_0(0) = 1, \tag{11}$$

since there is no row in D_0 and the support of P must be 0.

For a data matrix D with n rows, the PMF of X, denoted by $f_P(c)$, is equal to $f_n(c)$ ($c = 0, \cdots, n$) since $D = D_n$. To compute the PMF $f_P(c)$, we start with $f_0(c)$, and recursively compute $f_{i+1}(c)$ from $f_i(c)$ until $f_n(c)$ is computed. According to Equations (8), (9) and (10), it takes $O(i)$ time to compute f_i and thus $O(n^2)$ time to compute $f_P(c)$.

Divide-and-Conquer Algorithm. The naïve method takes time quadratic to the number of rows, which does not scale well. We now describe another algorithm for computing the support PMF $f_P(c)$, which achieves better time complexity by the divide-and-conquer strategy.

Given a set S of rows, let us define $f^S(c)$ as the PMF of the support of P in row set S, then our ultimate goal is to compute $f^D(c)$. To compute $f^S(c)$, we first divide the rows in S into two sets S_1 and S_2 of equal size. Let us denote $|S|$ by n, then S_1 contains the first $\lfloor \frac{n}{2} \rfloor$ rows, and S_2 contains the remaining $\lceil \frac{n}{2} \rceil$ rows.

Assume that $f^{S_1}(c)$ and $f^{S_2}(c)$ are already computed, then $f^S(c)$ can be obtained by the following formula:

$$f^S(c) = \sum_{i=0}^{c} f^{S_1}(i) \times f^{S_2}(c-i), \tag{12}$$

since the event that "the support of P in S is c" can be decomposed into the disjoint events "the support of P in S_1 is i, and the support of P in S_2 is $(c-i)$" for $i = 0, \cdots, c$.

Note that $|S| = n$, while $|S_1| = \lfloor \frac{n}{2} \rfloor$ and $|S_2| = \lceil \frac{n}{2} \rceil$. In Eq (12), we define $f^{S_1}(c) = 0$ for $c > \lfloor \frac{n}{2} \rfloor$, and define $f^{S_2}(c) = 0$ for $c > \lceil \frac{n}{2} \rceil$.

According to Eq (12), f^S is the convolution of f^{S_1} and f^{S_2}. Therefore, f^S can be computed from f^{S_1} and f^{S_2} in $O(n \log n)$ time using *Fast Fourier Transform* (FFT) [5].

Our divide-and-conquer algorithm for computing f^S is described as follows: S is first divided into two row sets S_1 and S_2 of equal size; then, PMFs f^{S_1} and f^{S_2} are computed by recursion; finally, f^S is computed as the convolution of f^{S_1} and f^{S_2} using FFT. Since each recursion step takes $O(n \log n)$ time (due to FFT), the overall time complexity of computing $f^D(c)$ is $O(n \log^2 n)$.

The base case for recursion is when $S = \{g\}$, in which case we directly return f^S where $f^S(0) = 1 - p_g$ and $f^S(1) = p_g$. In reality, we find that recursion down to $|S| = 1$ does not provide the best performance. The most efficient configuration is to stop recursion when $|S| \leq 500$, and compute f^S directly using the naïve method. We adopted this implementation.

B. Early Frequentness Validation

In the previous subsection, we described how to compute the PMF of the support of pattern P in data matrix D. Once the PMF is computed, we can decide whether pattern P is p-frequent using Equations (1) and (2) in Section II.

However, this two-step approach is time-consuming since it requires to compute the whole PMF vector $f_P(c)$, $c = 0, \cdots, n$. In fact, in order to determine whether pattern P is frequent, it is not always necessary to compute f_P to the end. The theorem below states that pattern P is frequent in D, as long as it is found to be frequent in a subset of D. As a result, in our divide-and-conquer algorithm, in each recursion step (that computes f^S) we will check the frequentness of P over S using Equations (1) and (2), and if it is found to be frequent, we terminate the frequentness checking immediately and conclude that P is frequent.

Theorem 2. *Suppose that pattern P is p-frequent in $S' \subseteq S$, then P is also p-frequent in S.*

Proof. See Appendix C [1]. □

C. Pattern Pruning

The frequentness checking operations described above is still very expensive (i.e., $O(n \log^2 n)$ time). We now present three pruning rules for pruning infrequent patterns. These rules can be checked efficiently in $O(n)$ time, and if any rule determines that a pattern P is infrequent, we do not need to do the expensive frequentness checking for P.

(1) Count-Prune. Let $cnt(P) = |\{g \in G \,|\, p_g(P) > 0\}|$, then pattern P is not p-frequent if $cnt(P) < \tau_{row}$.

(2) Markov-Prune. Pattern P is not p-frequent if $\sum_{g \in G} p_g(P) = E(X) < \tau_{row} \times \tau_{prob}$.

(3) Exponential-Prune. Let $\mu = E(X)$ and $\delta = \frac{\tau_{row} - \mu - 1}{\mu}$. When $\delta > 0$, pattern P is not p-frequent if
(1) $\delta \geq 2e - 1$, and $2^{-\delta\mu} < \tau_{prob}$, or
(2) $0 < \delta < 2e - 1$, and $e^{-\frac{\delta^2 \mu}{4}} < \tau_{prob}$.

The proofs for these rules can be found in Appendix D [1].

V. Mining Algorithms

In this section, we first propose a method for filtering out the rows g such that $p_g(P) = 0$. Then, we describe our two OPSM mining algorithms that integrate the techniques we presented.

A. Row Filtering

Consider the matrix in Figure 3, it is obvious that g_2 can never support pattern $t_3 < t_4$, i.e., $p_{g_2}(t_3 < t_4) = 0$. We now describe how to determine whether $p_g(P) = 0$ efficiently without actually evaluating the supporting probability.

Given a pattern $P = (t_{i_1} < t_{i_2} < \ldots < t_{i_j})$, we define its *valid interval* as the interval $[\ell_P, r_P]$ such that $p_g(P) > 0$ if and only if $t_{i_j} \in [\ell_P, r_P]$. We now consider how to compute the valid interval of a pattern P.

In the base case when $P = t_{i_1}$, its valid interval is exactly the interval $D[g][t_{i_1}] = [\ell_{i_1}, r_{i_1}]$.

For pattern $P = (t_{i_1} < \ldots < t_{i_j})$ $(j > 1)$, we compute its valid interval using that of pattern $P' = (t_{i_1} < \ldots < t_{i_{j-1}})$. Since row g supports P' if and only if $t_{i_{j-1}} \in [\ell_{P'}, r_{P'}]$, row g would support P if and only if $\exists t_{i_{j-1}} \in [\ell_{P'}, r_{P'}]$ such that $t_{i_{j-1}} < t_{i_j}$, or equivalently, $t_{i_j} \in [\max\{\ell_{i_j}, \ell_{P'}\}, r_{i_j}] \triangleq [\ell_P, r_P]$ (where $[\ell_{i_j}, r_{i_j}]$ is the interval $D[g][t_{i_j}]$). Note that $[\ell_P, r_P] = \emptyset$ if $\ell_P > r_P$, and in this case $p_g(P) = 0$ and g can be filtered.

In our mining algorithm when checking pattern P', for each row g, we maintain the following information: (1) its valid interval $[\ell_{P'}, r_{P'}]$, (2) the ordered list of split points, and (3) the DP-array. Since our mining algorithm checks patterns by pattern-growth, P will be checked after P'. To process g for pattern P, we will first compute $[\ell_P, r_P] = [\max\{\ell_{i_j}, \ell_{P'}\}, r_{i_j}]$. If $\ell_P > r_P$, we drop g from further consideration since it does not contribute to the support of P. Otherwise, we will compute the DP-array with that of g for P', using the algorithm of Section III-C, to obtain $p_g(P)$.

B. Algorithms

Pattern Anti-monotonicity. Suppose pattern P' is a sub-pattern of pattern P, then we have the following conclusions.

Algorithm 1 Expected-Support-Based Frequentness Checking

```
1: Filter the rows in $G_{P'}$ with their valid intervals, to obtain $G_P$ along with the updated valid intervals.
2: If $|G_P| < \tau_{row}$, mark $P$ as infrequent and return.
3: $sum \leftarrow 0$, $sum_0 \leftarrow \sum_{g \in G_{P'}} p_g(P')$
4: for each $g \in G_P$ do
5:     Compute the DP-array to obtain $p_g(P)$
6:     $sum \leftarrow sum + p_g(P)$, $sum_0 \leftarrow sum_0 - p_g(P')$
7:     If $sum + sum_0 < \tau_{row}$, mark $P$ as infrequent and return.
8: if $sum \geq \tau_{row}$ then
9:     Mark $P$ as frequent and return.
10: else
11:     Mark $P$ as infrequent and return.
```

Algorithm 2 Probabilistic Frequentness Checking

```
1: Filter the rows in $G_{P'}$ with their valid intervals, to obtain $G_P$ along with the updated valid intervals.
2: If $|G_P| < \tau_{row}$, mark $P$ as infrequent and return.
3: $sum \leftarrow 0$, $sum_0 \leftarrow \sum_{g \in G_{P'}} p_g(P')$
4: for each $g \in G_P$ do
5:     Compute the DP-array to obtain $p_g(P)$
6:     $sum \leftarrow sum + p_g(P)$, $sum_0 \leftarrow sum_0 - p_g(P')$
7:     if $sum + sum_0 < \tau_{row} \times \tau_{prob}$ then
8:         Mark $P$ as infrequent and return.
9: Apply Exponential-Prune and return when $P$ is pruned.
10: if $P$ is validated as frequent by divide-and-conquer then
11:     Mark $P$ as frequent and return.
12: else
13:     Mark $P$ as infrequent and return.
```

(1) For any row g, $g_p(P) < g_p(P')$. This is because in any possible world instantiation of g, P' must be supported if P is supported. (2) When pattern frequentness is defined using expected support, if P' is infrequent, then P must be infrequent. This is because $\sum_{g \in G} g_p(P) < \sum_{g \in G} g_p(P')$. (3) If P' is probabilistically infrequent, then P must be probabilistically infrequent. This is because in any possible world of D, the support of P' is at least the support of P.

Frequentness Checking. Since our mining algorithms check patterns by pattern-growth, when checking pattern $P = (t_{i_1} < \ldots < t_{i_j})$, we make use of the information of the rows for computation when checking $P' = (t_{i_1} < \ldots < t_{i_{j-1}})$.

Given a pattern P, **let $\mathbf{G_P}$ be the set of rows $\mathbf{g} \in \mathbf{G}$ with $\mathbf{p_g(P)} > \mathbf{0}$**, where each row is associated with its valid interval, split point list and DP-array.

The expected-support-based frequentness checking of pattern P is described by Algorithm 1. In Line 2, we prune P using the fact that $p_g(P) \leq 1$ and thus $\sum_{g \in G_P} p_g(P) \leq |G_P|$. Furthermore, we maintain two variables sum and sum_0. Let C be the set of rows already processed, then $sum = \sum_{g \in C} p_g(P)$ and $sum_0 = \sum_{g \in (G_P - C)} g_p(P')$. Line 7 prunes P using the fact that $\sum_{g \in G_P} p_g(P) = sum + \sum_{g \in (G_P - C)} p_g(P) \leq sum + sum_0$.

Algorithm 2 checks the p-frequentness of pattern P, where the pruning rules described in Section IV-C is used: Line 2 applies Count-Prune, Lines 7–8 applies Markov-Prune, and Line 9 applies Exponential-Prune.

Mining by Prefix-Projection. Our first mining algorithm is based on the idea of prefix-projection used by sequential pattern mining [19], where the current pattern $P = (t_{i_1} < \ldots < t_{i_j})$ is processed using the projected row set $G_{P'}$ of $P' = (t_{i_1} < \ldots < t_{i_{j-1}})$.

The recursive algorithm, denoted as $DFS(P, G_{P'})$[1], finds all the frequent patterns with prefix P from $G_{P'}$. Specifically, $DFS(P, G_{P'})$ first checks whether P is frequent using $G_{P'}$ (Algorithm 1 or 2). If so, it recursively calls $DFS(t_{i_1} < \ldots < t_{i_j} < t, G_P)$ for all $t \in T - \{t_{i_1}, \ldots, t_{i_j}\}$. The mining algorithm starts by calling $DFS(\emptyset, G)$.

For each recursion that processes P, G_P is maintained so that the subsequent recursions for $t_{i_1} < \ldots < t_{i_j} < t$ may use it. The maintenance of G_P incurs a space cost of $O(n \cdot j^2)$ since a DP-array of size $j \times s = O(j^2)$ is maintained for each row in G_P, and the space can only be released after its corresponding recursion is done.

Since our algorithm works in a depth-first manner, at most m projected row sets are maintained at any time, with the total space cost $\sum_{j=1}^{m} O(n \cdot j^2) = O(n \cdot m^3)$.

For a pattern P of length ℓ, computing $p_g(P)$ for all rows g using the pay-as-you-go algorithm of Section III-C takes $O(n\ell^2)$ time; then frequency checking takes $O(n)$ (resp. $O(n \log^2 n)$) for expected support (resp. p-frequentness with FFT computation). If C patterns are checked, then the time cost is bounded by $C \cdot O(n\ell^2 + n) = O(Cnm^2)$ (resp. $C \cdot O(n\ell^2 + n \log^2 n) = O(Cn(m^2 + \log^2 n))$. Additionally, row preprocessing of Section III-A takes $O(nm \log m)$ time.

We remark that the above analysis is very loose, and m can be replaced by ℓ_{max}, the length of the longest pattern checked.

The benefit of this algorithm is that, for any pattern P checked, its DP-arrays (of the rows in G_P) is computed exactly once; however, it can be expensive when the column set T is large, since the recursions have to be called for all $t \in T - \{t_{i_1}, \ldots, t_{i_j}\}$ and column candidate pruning is lacking.

Mining by Apriori. The Apriori algorithm works as follows: starting with the set of all length-1 patterns, we construct length-j frequent patterns from the set of all length-$(j-1)$ frequent patterns, until there is no frequent pattern left.

We organize the set of length-j frequent patterns using a prefix-tree T_j, like the FP-tree proposed in [8]. When computing length-j frequent patterns $P = (t_{i_1} < \ldots < t_{i_j})$, we need $G_{P'}$ for $P' = (t_{i_1} < \ldots < t_{i_{j-1}})$.

However, the space cost is prohibitive if we maintain $G_{P'}$ for each frequent length-$(j-1)$ pattern P' due to the breadth-first search order. Therefore, we choose to recompute $G_{P'}$ over the prefix-tree T_{j-1} in a depth-first manner (like the prefix-projection method without pattern pruning), and when recursing to the last tree-node $t_{i_{j-1}}$, we check whether $P = (t_{i_1} < \ldots < t_{i_{j-1}} < t_{i_j})$ is frequent for all possible

[1]We name the algorithm as DFS is due to its depth-first recursion nature.

column candidates $t_{i_j} \in C$ (we will discuss how to compute the candidate set C later), and if P is checked to be frequent, it is inserted to the new prefix-tree T_j.

Unlike the prefix-projection method, we need to recompute G_P when checking all prefix-trees T_z $(z \geq j)$. However, the computation in the same tree is still shared. For example, $G_{t_1<t_2}$ is used for computing both $G_{t_1<t_2<t3}$ and $G_{t_1<t_2<t_4}$.

We now consider how to determine t_{i_j}'s candidate set C.

Theorem 3. *For any column t that is not a child of tree-node $t_{i_{j-2}}$ in T_{j-1}, pattern $P = (t_{i_1} < \ldots < t_{i_{j-1}} < t)$ is infrequent.*

Proof. See Appendix E [1]. □

Thus, we choose C to include all child nodes of node $t_{i_{j-2}}$ in T_{j-1}. Compared with the prefix-projection method, C is much smaller than $(T - \{t_{i_1}, \ldots, t_{i_{j-1}}\})$ and thus the recursion has a much smaller fan-out.

The pattern anti-monotonicity also allows for the following pattern pruning:

Theorem 4. *For any length-j pattern P, if not all of the length-$(j-1)$ sub-pattern of P exist in T_{j-1}, then P is infrequent.*

Theorem 4 is efficient to check, and is thus examined before the more expensive frequentness checking of Algorithm 1 or 2 which requires computing the DP-array for all rows in G_P.

In the Apriori algorithm, computing $p_g(P)$ of patterns of length ℓ requires re-computing the DP-array for frequent patterns of length $< \ell$; this brings an additional factor of ℓ_{max} to our prefix-projection algorithm's time complexity since each P is re-processed for at most ℓ_{max} times. In return, the algorithm enjoys a small recursion fan-out, which however is not reflected in time complexity analysis.

VI. Experiments

This section evaluates the performance of our proposed algorithms on real datasets, and compares it with existing approaches POPSM (using uniform distribution) and OPSMRM.

We denote our prefix-projection (resp. Aprioi) based mining algorithm by "DFS" (resp. "Apri"), and for pattern frequentness, we denote expected support (resp. p-frequentness) by "ES" (resp. "PF"). Thus, we have four algorithm variants: *DFS-ES*, *Apri-ES*, *DFS-PF* and *Apri-PF*. In addition, since the FFT algorithm for checking p-frequentness of a pattern is expensive, we apply the approximation technique introduced in Section 6 of [20] to check the p-frequentness, which improves the cost from $O(n \log^2 n)$ to $O(n)$; we denote algorithms using this approximation technique by *DFS-PFA* and *Apri-PFA*.

Since whether the mining algorithm is "DFS" or "Apri" does not impact the output, when we evaluate result quality, we use *ES* for both *DPS-ES* and *Apri-ES*, use *PF* for both *DPS-PF* and *Apri-PF*, and use *PFA* for both *DPS-PFA* and *Apri-PFA*. All experiments were conducted on a PC with Intel i7-6700K quad core CPU at 4GHz, 16GB DDR4 memory and 120GB SSD. All our codes are released at the following GitHub link: https://github.com/RobinJCheng/OPSM.

A. Experimental Setup

Datasets: Two real public microarray gene expression datasets are used in our experiments: (1) the GAL dataset[2] about yeast galactose utilization [16] which is also used by [7]; and (2) the GDS2003 dataset[3] used by [17], which is a microarray dataset of the baker's yeast Saccharomyces cerevisiae from the Gene Expression Omnibus (GEO) database [2]. The GAL dataset contains 205 gene probes (rows) and 20 experimental conditions (columns) with 4 replicates for each entry in the matrix, while GDS2003 contains 5617 gene probes (rows) and 10 samples (columns) with 3 replicates for each entry.

Besides the gene expression datasets, we also use a benchmark movie rating dataset[4] with 100,000 ratings from 943 users on 1682 movies [9] to evaluate the algorithms.

Preprocessing: For GAL and GDS2003, we set the lower and upper bounds of every matrix entry's interval as $[min, max]$, where min (resp. max) is the minimum (resp. maximum) replicated value of the entry from repeated experiments.

The movie rating dataset is an incomplete matrix, and we use TensorFlow to conduct factorization based matrix completion which provides a complete 943×1682 user rating matrix M, where M_{ij} estimates User i's rating towards Movie j. To consider only the popular movies, we select the top-100 movies that get the most user ratings, which generates a 943×100 submatrix M_s of M. Since each new rating r is now a low-rank approximation, we introduce uncertainty to each rating r. A rating in the original data take its value from $\{1, 2, 3, 4, 5\}$; after matrix completion, a rating is a real value like $r = 3.4$, in which case we assign an interval $[3, 4]$ to the matrix entry, and for OPSMRM, we assign replicates $\{3, 4\}$.

Evaluation Metrics: The following metrics are studied:

1) Result Quality: for GAL and GDS2003, we use *biological significance* of the mined OPSMs to demonstrate the result quality of our proposed method. We adopt a widely used metric, *p*-value [11], [7], [17], which measures the association between OPSMs mined and the known gene functional categories. Specifically, a smaller *p*-value indicates a stronger association between an OPSM and gene categories, i.e., biologically more significant. Following [7], we also consider four *p*-value ranges, $[0, 10^{-40})$, $[10^{-40}, 10^{-30})$, $[10^{-30}, 10^{-20})$ and $[10^{-20}, \infty)$, as *significance levels*. We compared the result quality of our algorithms with the existing algorithms in terms of the number of OPSMs mined at each significance level.

For the movie rating dataset, we define a *Kendall tau score* (KTS) motivated by the concept of Kendall tau distance: for a mined OPSM with movie order $t_1 < t_2 < \ldots < t_k$, we compute a KTS for each user g_i in the OPSM, which equals the fraction of all possible C_k^2 movie pairs (t_i, t_j) where the rating order is consistent with that in our 943×100 submatrix M_s; the KTS of the OPSM is then computed as the average KTS over all its users.

[2]http://genomebiology.com/content/supplementary/gb-2003-4-5-r34-s8.txt
[3]https://www.ncbi.nlm.nih.gov/sites/GDSbrowser?acc=GDS2003
[4]https://grouplens.org/datasets/movielens/100k/

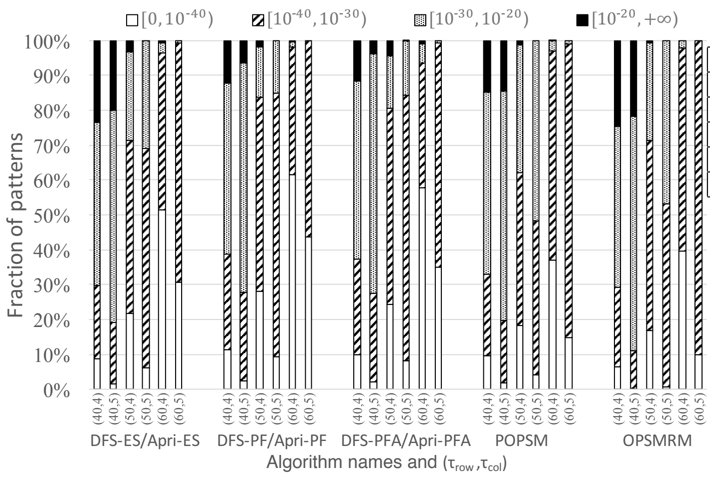

Figure 6: OPSM Result Distribution w.r.t. Size Thresholds

Table I: Number of OPSMs w.r.t. Size Thresholds

	(40, 4)	(40, 5)	(50, 4)	(50, 5)	(60, 4)	(60, 5)
DFS/Apri-ES	6296	3246	2537	834	1033	144
DFS/Apri-PF	4867	2253	1967	562	804	94
DFS/Apri-PFA	4179	1842	1751	452	658	50
POPSM	5687	3177	3003	1290	1475	355
OPSMRM	2989	887	1141	169	406	10

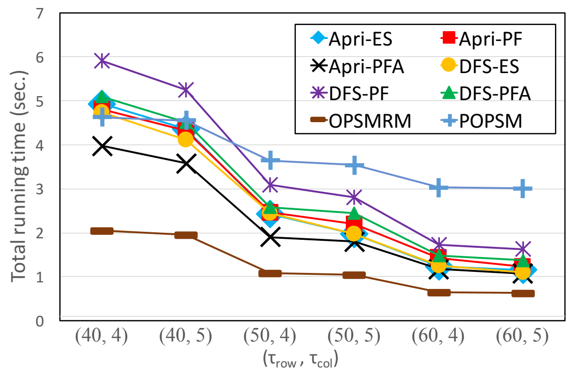

Figure 7: Total Runtime w.r.t. Size Thresholds (τ_{row}, τ_{col})

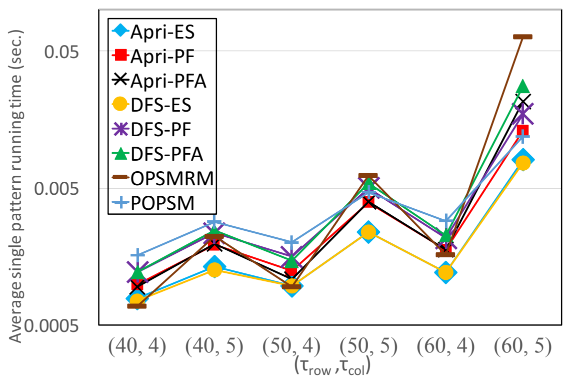

Figure 8: Average Single-Pattern Runtime w.r.t. (τ_{row}, τ_{col})

2) Time Efficiency: we evaluate the time efficiency of the algorithms by considering the following two metrics:

- *TR-Time*: total running time for mining the OPSMs;
- *SP-Time*: the average time for mining a single OPSM.

Here, *SP-Time* equals *TR-Time* divided by the number of patterns mined.

3) Space Efficiency: we report the peak memory consumption of each program run in our experiments.

All experiments were repeated for 10 times and the reported metrics are averaged over the 10 runs (although results observed from different runs are actually quite stable/similar).

B. Experiments on GAL Dataset

Effect of Size Thresholds τ_{row} and τ_{col}. We follow [7]'s parameter settings of τ_{row} and τ_{col} for evaluating of GAL, by fixing $\tau_{cut} = 0.5$, $\tau_{prob} = 0.95$. This is because with $\tau_{prob} = 0.95$ the number of OPSMs mined by different algorithms are similar to each other, which makes it fair to compare the percentage of patterns in each significance level. Also, since the results are similar for various τ_{cut} values, we only show results when $\tau_{cut} = 0.5$ to save space.

Result Quality. We vary τ_{row} among $\{40, 50, 60\}$ and vary τ_{col} among $\{4, 5\}$. Figure 6 presents the fraction of mined OPSMs that fall in each *significance level*. For example, the first bar "(40, 4), DFS-ES/Apri-ES" represents the distribution of the patterns mined by *DFS-ES* or *Apri-ES* with $\tau_{row} = 40, \tau_{col} = 4$. We see that our algorithms find larger fractions of high-quality OPSMs, as they have much taller white bars (representing highest significance level with pvalue $\in [0, 10^{-40})$) than POPSM and OPSMRM. For example, when $\tau_{row} = 60, \tau_{col} = 4$, our proposed *ES* and *PF* has 51.4% and 61.7% of patterns falls into the highest significance level, while *POPSM* and *OPSMRM* has only 36.9% and 39.7%, respectively. Among our algorithms, *PF* performs the best, followed by *PFA* and then by *ES*, which verifies that considering PMF gives more better results than considering only expectation, even with approximation adopted.

Table I presents the number of OPSMs mined. If we consider the fraction of OPSMs mined that fall into the highest significance level, $[0, 10^{-40})$, on average (across the different groups of parameters tested) *PF* has 30% more OPSMs than *ES*, 84% more OPSMs than *POPSM*, and 112% more OPSMs than *OPSMRM*. Both of our proposed methods can discover more patterns than *OPSMRM*, though in some cases, *POPSM* discovers slightly more patterns.

Time Efficiency. Figures 7 and 8 show the total running time *TR-Time* and average single-pattern running time *SP-Time* of our proposed methods and the existing algorithms OPSMRM and POPSM for different (τ_{row}, τ_{col}). From Figure 7 we can see that our fast Apriori based mining algorithms *Apri-ES* and *Apri-PF* have a shorter total running time compared with *POPSM*, except when $(\tau_{row} = 40, \tau_{col} = 4)$ where *Apri-ES* is 0.3 second slower than *POPSM* (but discovers 611 more OPSMs). *OPSMRM* has the shortest running time among all the methods, but it discovers the fewest number of OPSMs. In fact, according to Figure 8, *OPSMRM* has the worst *SP-Time*. *Apri-ES* and *Apri-PF* perform better than POPSM in terms of *SP-Time*, as they discovered more OPSMs. *PFA* algorithms are always faster than *PF*, and while *Apri-PFA* is faster than *Apri-ES*, *DFS-PFA* is slightly slower than *DFS-ES*.

Memory Efficiency. Figure 9 shows the peak memory

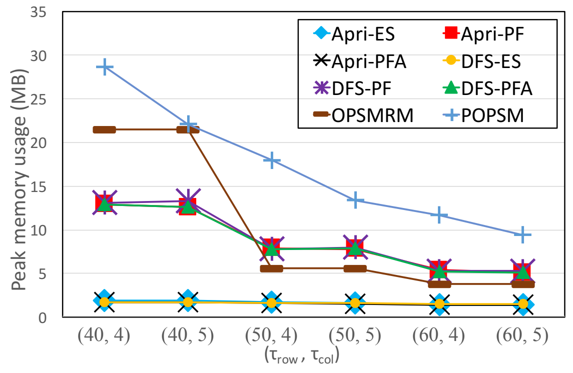

Figure 9: Peak Memory w.r.t. Size Thresholds (τ_{row}, τ_{col})

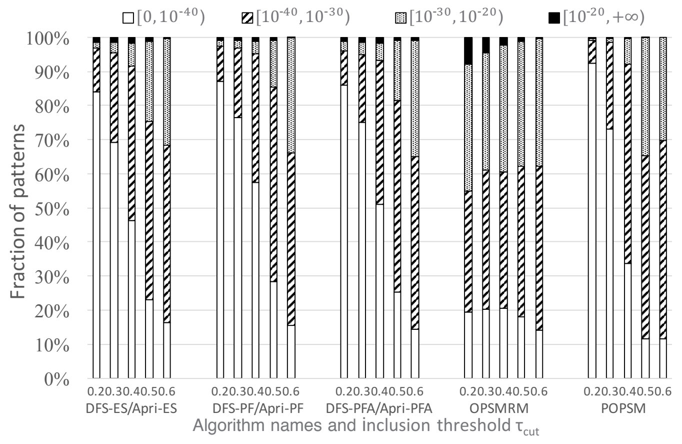

Figure 10: OPSM Result Distribution w.r.t. τ_{cut}

Table II: Number of OPSMs w.r.t. Inclusion Thresholds τ_{cut}

	0.2	0.3	0.4	0.5	0.6
DFS/Apri-ES	2537	2537	2531	2402	1531
DFS/Apri-PF	1967	1967	1965	1930	1521
DFS/Apri-PFA	1751	1751	1696	1613	1183
POPSM	24953	10990	5731	3002	1516
OPSMRM	1141	1141	1138	751	496

usage of various algorithms. We can see that *Apri-ES*, *DFS-ES* and *Apri-FPA* have a much lower memory consumption compared with the other algorithms. For example, when $\tau_{row} = 40$ and $\tau_{col} = 4$, *ES* consumes 11.9 times less memory than *OPSMRM*, and 15.9 times less memory than *POPSM*. The less memory usage by *ES* does not compromise the number of mined OPSMs: as Table I shows, *ES* discovers 2.1 times more OPSMs than *OPSMRM* and 1.1 times more OPSMs than *POPSM*. Our *PF* algorithms use more memory than *ES* due to the need of processing PMF vectors (c.f. Section IV-B), but they generate higher-quality OPSMs (in terms of p-value) than *ES* (c.f. Figure 6). Compared with *OPSMRM* and *POPSM*, *PF* algorithms have a much lower memory consumption than *OPSMRM* and *POPSM* especially when $(\tau_{row}, \tau_{col}) = (40, 4)$ and $(40, 5)$ where the number of OPSMs mined are large. In fact, as Figure 9 shows, POPSM consistently uses the most memory under different parameter settings.

Effect of Inclusion Threshold τ_{cut}. We fix $\tau_{row} = 50, \tau_{col} = 4$ since the above experiments show that various methods generate comparable number of OPSMs with these parameters. We also fix $\tau_{prob} = 0.8$ for *DFS-PF* and *Apri-PF*, but vary τ_{cut} among $\{0.2, 0.3, 0.4, 0.5, 0.6\}$. Figure 10 shows the fraction of OPSMs in each significance level for every algorithm with different inclusion threshold τ_{cut}. Table II presents the number of OPSMs mined with different inclusion threshold τ_{cut}. We can see that our algorithms consistently outperform *POPSM* and *OPSMRM* at various values of τ_{cut} (c.f. the white bars).

Our algorithms have both a higher OPSM quality and a larger number of OPSMs found compared with *OPSMRM*. *POPSM* sometimes finds more OPSMs but our algorithms have a better OPSM quality. For example, when $\tau_{cut} = 0.4$, 46% and 58% of the OPSMs found by our *ES* and *PF* algorithms fall into the highest significance level, while *POPSM* and *OPSMRM* have only 33% and 32%, respectively.

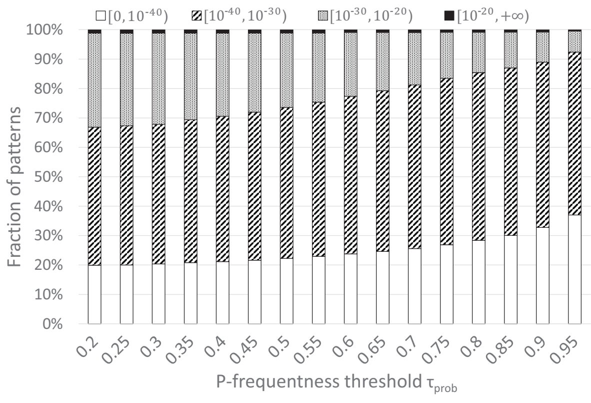

Figure 11: OPSM Result Distribution w.r.t. τ_{prob}

Table III: Number of OPSMs w.r.t. Confidence Threshold τ_{prob}

	0.2	0.3	0.4	0.5	0.6	0.7	0.8
DFS/Apri-PF	3621	3212	2910	2664	2422	2209	1967

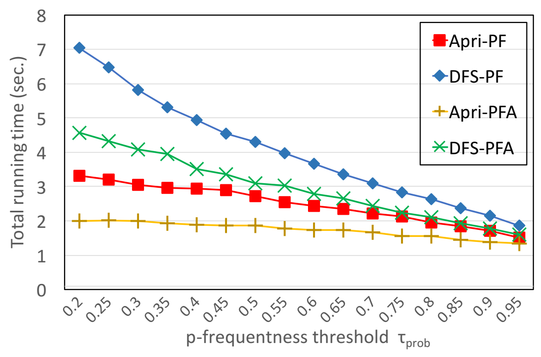

Figure 12: Total Running Time w.r.t τ_{prob}

Table IV: Total Running Time w.r.t. Size Thresholds

	(200, 4)	(200, 5)	(300, 4)	(300, 5)	(400, 4)	(400, 5)
Apri-ES	1604	1286	262	225	61	59
Apri-PF	2871	2261	473	440	123	125
Apri-PFA	1420	1138	221	189	52	50
DFS-ES	2358	1938	412	370	100	96
DFS-PF	6345	5640	1155	1104	300	297
DFS-PFA	4998	4107	937	841	252	241
POPSM	OOM	OOM	14643	14209	5133	5200
OPSMRM	235	235	216	216	197	197

Effect of Probability Threshold τ_{prob}. We study the effect of τ_{prob} used in *PF*, by fixing $(\tau_{row} = 50, \tau_{col} = 4)$ (which has the largest fraction of OPSMs fall in the top-2 levels in Figure 6), and $\tau_{cut} = 0.5$; and varying τ_{prob} from 0.2 to 0.95 with a step length of 0.05. Figure 11 and Table III shows that our model is very robust, not sensitive to parameter τ_{prob}, and has consistent performance in terms of result quality.

Figure 12 shows the *TR-Time* of *PF* and *PFA* algorithms w.r.t. diifferent τ_{prob}. We can see that our Apriori-based mining algorithm *Apri-PF(A)* has shorter running time than the naïve depth-first-based method *DFS-PF(A)*, especially when τ_{prob} is small (and hence more OPSMs are found). *Apri-PF(A)* also has a stable running time with different τ_{prob}. Figure 12 also shows that the approximation algorithms *PFA* gain an obvious speedup compared with *PF* algorithms.

Wrapup. To summarize, this set of experiments show that our *ES* and *PF* models are robust and output higher-quality OPSMs than *OPSMRM* and *POPSM*, and consistently find more OPSMs than *OPSMRM*. Our Apriori-based algorithms also consistently outperform *POPSM* in terms of *SP-Time*. *ES* consumes the least amount of memory, and *PF* consumes more memory than *ES* but less memory than *OPSMRM* and *POPSM*.

C. Experiments on the GDS Dataset

The results on the GDS dataset is similar to those on the GAL dataset described above, and due to the space limitation, we put them in Appendix F [1].

D. Experiments on the Movie Rating Dataset

Time Efficiency. Table IV shows the *TR-Time* of various algorithms with different (τ_{row}, τ_{col}) (w.l.o.g., we fix $\tau_{cut} = 0.3$, $\tau_{prob} = 0.6$), where *OOM* means out of memory. Note that when $\tau_{row} = 200$, *POPSM* always runs out of memory after 10 hours. *OPSMRM* has the shortest *TR-Time* but it failed to produce any OPSMs that satisfy the τ_{col} threshold. We see that our methods are tens of times faster than *POPSM*, and ES methods are faster than *PF* as they do not process PMF vectors. Apriori-based algorithms are also faster than their *DFS*-based counterparts due to effectiveness pattern pruning. Finally, approximation algorithms *PFA* provides reasonable speedup to *PF*.

Effectiveness. Without loss of generality, we consider $\tau_{cut} = 0.3$, $\tau_{prob} = 0.6$, $\tau_{row} = 300$ and vary $\tau_{col} = 5$, and visualize the top-10 OPSMs mined by *DFS/Apri-ES*, *DFS/Apri-PF* and *POPSM* with the highest KTS. We remark that here our *DFS-ES* consumes 2582× less memory than *POPSM* and 3294× less memory than *OPSMRM*.

Table V(a) lists the top-10 OPSM patterns with the highest KTS produced by *ES*, Table V(b) by *PF*, Table V(c) by *PFA*, and Table V(d) by *POPSM*. To be space efficient, we use abbreviations for movie names, and their meanings are listed in Table VI. We can see that the OPSMs mined by our methods generally have a higher KTS.

E. Scalability Experiments

To examine how well our algorithms scale to the number of rows and columns of a data matrix D, we duplicate the rows

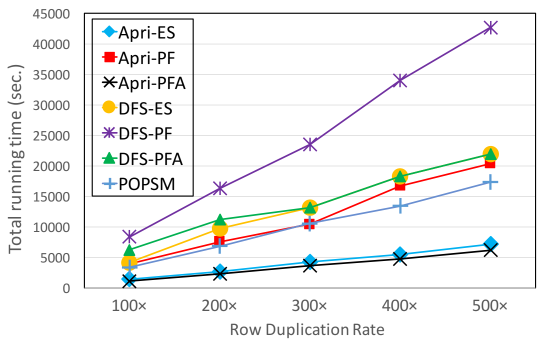

Figure 13: Scalability to the Number of Rows

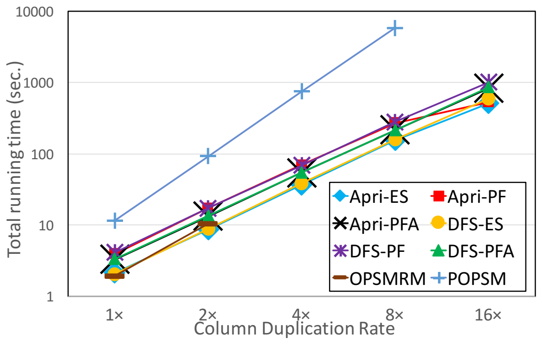

Figure 14: Scalability to the Number of Columns

and columns of our 943 × 100 movie rating submatrix M_s to generate larger datasets for running scalability experiments.

To test row scalability, we duplicate the rows of M_s for 100, 200, 300, 400 and 500 times, and run the various algorithms on them. Without loss of generality, we set $\tau_{row} = 0.6 * n$ (where n is the row number) and fix $\tau_{col} = 5$ and $\tau_{prob} = 0.6$. Unfortunately, OPSMRM runs out of memory even on the smallest data with M_s's rows duplicated for 100 times, and thus we cannot report its result. The results for the other algorithms are shown in Figure 13, where we observe that POPSM is much slower than our algorithms, and that *Apri-ES* and *Apri-PFA* are much faster than the other algorithms. Overall, the time of all algorithms scale linearly with row number n, which matches our derived time complexity $O(Cn(m^2 + \log^2 n))$ in Section V-B (i.e., linear to n and quadratic to m).

To test column scalability, we duplicate the columns of M_s for 2, 4, 8 and 16 times, and run the various algorithms on them. Here, we set $\tau_{row} = 700, \tau_{col} = 5$ and $\tau_{prob} = 0.6$. The results are shown in Figure 14, where we observe that the running time of various algorithms increase quickly with column number m, which aligns with our analysis. Although OPSMRM has a similar running time to our algorithms, the memory consumption rockets up with more columns. Also, OPSM-RM runs out of memory when the columns are duplicated for 4 times and thus we only plot 2 points for it in Figure 14. Finally, POPSM is not only slower than all our algorithms, its increase rate is also sharper.

VII. Related Work

Many problems have been studied in the context of data uncertainty, such as top-k queries [12], [10], [15], frequent itemset mining [13], [14], frequent sequence mining [19], [21],

Table V: Movie Pattern Visualization

(a) Top-10 OPSMs by *DFS/Apri-ES*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	RK	AN	DW	MPHG	99.37%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
ET	TK	AN	TT	MPHG	99.27%
TI	MI	AFO	IO	MPHG	99.26%
TI	ID	US	IO	MPHG	99.24%
TI	ID	SS	IO	MPHG	99.20%

(b) Top-10 OPSMs by *DFS/Apri-PF*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	RK	AN	DW	MPHG	99.37%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
TI	MI	AFO	IO	MPHG	99.24%
TI	ID	US	IO	MPHG	99.24%
TI	ID	SS	IO	MPHG	99.19%
TI	ID	G	IO	MPHG	99.18%

(c) Top-10 OPSMs by *DFS/Apri-PFA*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
TI	ID	Jaws	CA	MPHG	99.26%
TI	R	RK	AN	MPHG	99.24%
TI	R	RK	AN	PB	99.18%
TI	R	Jaws	CA	MPHG	99.17%
TI	ET	SL	CA	MPHG	99.17%

(d) Top-10 OPSMs by *POPSM*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.32%
TI	RK	AN	CA	MPHG	99.24%
R	RK	AN	PB	MPHG	99.19%
R	RK	AN	IO	MPHG	99.19%
TI	RK	MB	IO	MPHG	99.19%
ET	RK	AN	S	MPHG	99.19%
TI	ID	SL	IO	MPHG	99.19%
TI	ID	MB	IO	MPHG	99.19%
TI	RK	AN	PB	MPHG	99.16%
TI	ID	SL	CA	MPHG	99.15%

Table VI: Movie Names and Their Abbreviations

Titanic (1997)	TI	Ransom (1996)	R
Independence Day (1996)	ID	Rock, The (1996)	RK
Air Force One (1997)	AFO	Apocalypse Now (1979)	AN
In & Out (1997)	IO	Chasing Amy (1997)	CA
Princess Bride, The (1987)	PB	Schindier's List (1993)	SL
Sting, The (1987)	S	Men in Black (1997)	MB
Dances with Wolves (1990)	DW	Terminator, The (1984)	TT
Mission: Impossible (1996)	MI	Jerry Maguire (1996)	JM
Usual Suspects, The (1995)	US	Sense and Sensibility (1995)	SS
Game, The (1997)	G	E.T. the Extra-Terrestrial (1982)	ET
Monty Python and the Holy Grail (1974)			MPHG

and recently, OPSM mining [17], [7]. Most of these works adopt the possible world semantics to address the respective problems and demonstrated their effectiveness.

The existing works that are closely related to ours are [3], [17] and [7]. The OPSM mining problem is first introduced in [3] to analyze gene expression data. Later, [4] develops a more efficient mining algorithm based on a new data structure, the head-tail trees. However, these works cannot cope with noisy gene expression data that are common. OP-clustering [11] generalizes the OPSM model by grouping attributes into equivalent classes. Other models relax the order requirement instead, such as AOPC [18] and ROPSM [6].

OPSMRM [17] combats noise in microarray analysis by letting each test be repeated to obtain several measurements, forming possible worlds with discrete probability distribution. Only expected support is used to evaluate pattern significance, and for each significant pattern, all rows whose supporting probability is at least the inclusion threshold τ_{cut} are selected to compose a submatrix. However, as Section VI shows, *PF* generates higher-quality OPSMs than *ES* due to considering the whole PMF, and interval model delivers better results.

POPSM [7] attempts to model the underlying distributions that generate the observed measurements, and thus, the value in each matrix entry is given by an interval with its associated continuous probability distribution. However, instead of defining pattern frequentness according to possible world semantics, POPSM adopts a simple requirement that every row in an output OPSM has its supporting probability no less than a user-defined threshold τ_{cut}, which is unclear how to set properly due to the lack of semantics from probability theory. Section VI verifies that our use of possible world semantics allows the generation of higher-quality OPSMs than POPSM.

VIII. Conclusion

This work studied probabilistically-frequent OPSM mining over matrix with continuous value uncertainty, following the well-established possible world semantics. To our knowledge, this is the first OPSM mining work that combines both possible world semantics and interval-based data model. We proposed many techniques to efficiently determine pattern significance and to prune unpromising patterns. Experiments show that our algorithms find OPSMs with much higher quality than existing approaches, and the time cost is also competitive.

References

[1] **Online Appendix:** http://www.cs.uab.edu/yanda/papers/opsm.pdf.

[2] T. Barrett, D. B. Troup, S. E. Wilhite, P. Ledoux, D. Rudnev, C. Evangelista, I. F. Kim, A. Soboleva, M. Tomashevsky, K. A. Marshall, et al. Ncbi geo: archive for high-throughput functional genomic data. *Nucleic acids research*, 37(suppl 1):D885–D890, 2009.

[3] A. Ben-Dor, B. Chor, R. M. Karp, and Z. Yakhini. Discovering local structure in gene expression data: The order-preserving submatrix problem. *Journal of Computational Biology*, 10(3/4):373–384, 2003.

[4] L. Cheung, D. W. Cheung, B. Kao, K. Y. Yip, and M. K. Ng. On mining micro-array data by order-preserving submatrix. *IJBRA*, 3(1):42–64, 2007.

[5] T. H. Cormen and etcl. *Introduction to algorithms*. MIT press, 2009.

[6] Q. Fang, W. Ng, and J. Feng. Discovering significant relaxed order-preserving submatrices. In *SIGKDD*, pages 433–442, 2010.

[7] Q. Fang, W. Ng, J. Feng, and Y. Li. Mining order-preserving submatrices from probabilistic matrices. *TODS*, 39(1):6:1–6:43, 2014.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[9] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *TiiS*, 5(4):19:1–19:19, 2016.

[10] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1):502–513, 2009.

[11] J. Liu and W. Wang. Op-cluster: Clustering by tendency in high dimensional space. In *ICDM*, pages 187–194, 2003.

[12] M. A. Soliman, I. F. Ilyas, and K. C. Chang. Top-k query processing in uncertain databases. In *ICDE*, pages 896–905, 2007.

[13] L. Sun, R. Cheng, D. W. Cheung, and J. Cheng. Mining uncertain data with probabilistic guarantees. In *SIGKDD*, pages 273–282, 2010.

[14] Y. Tong, L. Chen, Y. Cheng, and P. S. Yu. Mining frequent itemsets over uncertain databases. *PVLDB*, 5(11):1650–1661, 2012.

[15] D. Yan and W. Ng. Robust ranking of uncertain data. In *DASFAA*, pages 254–268, 2011.

[16] K. Y. Yeung, M. Medvedovic, and R. E. Bumgarner. Clustering gene-expression data with repeated measurements. *Genome biology*, 4(5):R34, 2003.

[17] K. Y. Yip, B. Kao, X. Zhu, C. K. Chui, S. D. Lee, and D. W. Cheung. Mining order-preserving submatrices from data with repeated measurements. *TKDE*, 25(7):1587–1600, 2013.

[18] M. Zhang, W. Wang, and J. Liu. Mining approximate order preserving clusters in the presence of noise. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 160–168, 2008.

[19] Z. Zhao, D. Yan, and W. Ng. Mining probabilistically frequent sequential patterns in uncertain databases. In *EDBT*, pages 74–85, 2012.

[20] Z. Zhao, D. Yan, and W. Ng. Mining probabilistically frequent sequential patterns in large uncertain databases. *IEEE Trans. Knowl. Data Eng.*, 26(5):1171–1184, 2014.

[21] Z. Zhao, D. Yan, and W. Ng. Mining probabilistically frequent sequential patterns in large uncertain databases. *IEEE Trans. Knowl. Data Eng.*, 26(5):1171–1184, 2014.