Chapter 4

Frequent itemsets with errors

Datasets obtained by large-scale high-throughput methods for detecting protein-protein interactions typically suffer from a relatively high level of noise. As we have mentioned in Chapter 2, two types of errors are possible: false positive errors (FP) and false negative errors (FN). Purification experiments typically allow a higher degree of confidence when an interaction is observed, but a much lower degree when no interaction is detected (see error estimation (Table 2.1) described in Chapter 2). In other words, most of the errors are false negatives; it is believed that when no interaction is detected, it is quite likely that an interaction actually exists, but the experiment failed to detect it [50].

We believe that false negative errors cause low sensitivity when using frequent modules in Chapter 3 to predict interactions; many proteins should have been part of frequent itemsets, but purifications fail to detect them. Ideally, we would like to have a computational method which would be able to correct many of the errors made by large-scale interaction experiments. In this chapter, we propose an algorithm to detect larger frequent modules that have been missed by large-scale experiments. Our algorithm is an extension of the Apriori algorithm to solve the probabilistic frequent itemsets problem.

The basic idea of the algorithm derives from the hypothesis that protein complexes organize into frequent modules. Several modules join together to form a protein complex [21]. In the last chapter, we have identified frequent modules from protein purifications. In this chapter, we attempt to predict more interactions that join these modules to form a larger complex. We believe that these interactions have been missed by large-scale experiments.

The rest of this chapter is organized as follows: In Section 4.1, we introduce the probabilistic frequent itemsets and the related concept of combinatorial errors. In Section 4.3 we present algorithms to compute probabilistic frequent itemsets and discuss the implementation. In Section 4.5, we discuss our results.

4.1 Problem setting

In the frequent itemset problem, the available set of M transactions can be characterized as an $M \times N$ binary matrix D. Each row of D corresponds to a transaction i and each column of D corresponds to an item j. The entry D_{ij} is one if the ith transaction contains item j and zero otherwise. Under exact frequent itemset mining a transaction supports an itemset if it contains ones for all items in the itemset. An itemset is frequent if its fraction of support exceeds the user-defined minimum support threshold.

While the classical exact frequent itemset definition and the algorithms designed to discover such itemsets have been well studied, the problems created by imperfect data have not. In our case, we have both false negative and false positive errors in the data. In the presence of such noise, the Apriori algorithm finds a large number of small fragments of the true itemset, but too many transactions miss one or more items due to observational error. As a solution, we present an error-tolerant approach to frequent itemset mining based on two different error characteristics: combinatorial error and probabilistic error.

4.2 Combinatorial error

A natural approach of handling errors is to relax the requirement that a supporting transaction must contain ones for all the items in the frequent itemset, for example in Figure 4.1. In this problem setting, a small number of zeros is tolerated to relax the exact matching criterion to yield a more flexible definition of support and consequently of frequent itemsets. Previous works have proposed different ways to quantify the false negative error: fixed integer error [33] and percentage error [49]. We summarize the relaxing definitions according to both works.

Definition 7 *Fault-Tolerant Itemset (FTI): An itemset E is an fault-tolerant itemset having a fixed*

	a	b	c	d	e
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	0	1	1
4	1	1	0	1	1
5	1	1	0	0	0

(a)

	a	b	c	d	e
1	1	1	1	1	0
2	1	1	1	1	0
3	1	1	1	1	0
4	1	1	1	1	0
5	1	1	1	1	0

(b)

	a	b	c	d	e
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	1	1
4	1	1	1	1	1
5	0	0	0	0	0

(c)

	a	b	c	d	e
1	1	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	1
4	1	1	1	1	0
5	0	1	1	1	1

(d)

Figure 4.1: Combinatorial errors. The true frequent itemset is $\{a, b, c, d, e\}$. Five transactions with global density of 80% but different distributions of errors in individual transactions and items.

a	b	c	d
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1

Table 4.1: A levelwise algorithm with $\sigma = 1$, $\epsilon = 0.5$ would discard all proper subsets of $\{a, b, c, d, \}$, although $\{a, b, c, d\}$ is an error-tolerant itemset. However, if we use a fixed integer error, a levelwise algorithm with $\sigma = 1$, $\delta = 2$, $\{a, b, c, d\}$ will be found as frequent.

integer error δ and support σ with respect to a database D having M transactions if in at least σM transactions at least $|E| - \delta$ of the items in E are present [33].

Definition 8 *Error-Tolerant Itemset (ETI): An itemset E is an error-tolerant itemset having error ϵ and support σ with respect to a database D having M transactions if in at least σM transactions at least a fraction $1 - \epsilon$ of the items in E are present [49].*

4.2.1 Algorithm for finding FTIs

Similar to exact frequent itemsets, FTIs have a monotonicity property:

Theorem 1 *If X is not a fault-tolerant δ itemset with $\mathrm{support}(X) > \sigma$, then none of its superset is a fault-tolerant itemset, $|X| > \delta$.*

For example, in Table 4.1, based on the definition of a fault-tolerant itemset, the set $\{a\}$ has an error of 1, $\{a, b\}$ with 2 errors, and $\{a, b, c, d\}$ must have at least 2 errors. If we use a levelwise algorithm with a minimum support of 100% and $\delta = 2$, all proper subsets of $\{a, b, c, d\}$ are fault-tolerant frequent itemsets. With $\delta = 1$, only subsets of length one and two satisfy the definition.

Unfortunately, unlike the Apriori algorithm for finding exact frequent itemsets, a levelwise algorithm to generate fault-tolerant frequent itemsets is not feasible due to the large number of candidates. For example, given 1000 items in the database, to find a fault-tolerant itemset of length 3, with $\delta = 1$, we will need to generate $\binom{1000}{3}$, or all possible combinations of subsets of length k. We followed the idea of Pei et al. [33] who introduced a weaker definition of fault-tolerant frequent itemset by adding a constraint on minimum frequency of items:

Definition 9 . *(constraint fault-tolerant frequent itemset) Given a fixed integer maximum error $\delta > 0$, let Y be an itemset such that $|Y| > \delta$. A transaction T is said to contain Y with tolerance δ iff there exists $Y' \subseteq Y$ such that $Y' \subseteq T$ and $|Y'| \geq (|Y| - \delta)$. The number of transactions in a database containing Y with fault tolerance δ is called the FT-support of Y, denoted by $\tilde{\mathrm{sup}}(Y)$.*

Let $\tilde{B}(X)$ be the set of transactions containing X with tolerance δ. Given (1) a **frequent-item minimum support** *σ_{item} and (2) an* **FT-support threshold** *σ_{FT}, an itemset X is called a* **fault-tolerant frequent itemset** *or* **FT-itemset**, *iff*

1. *$\tilde{\mathrm{sup}}(X) \geq \sigma_{FT}$ and*
2. *for each item $x \in X$, $\mathrm{support}_{\tilde{B}(X)}(x) \geq \sigma_{item}$, where $\mathrm{support}_{\tilde{B}(X)}(x)$ is the fraction of transactions in $\tilde{B}(X)$ containing item x.*

The above definition has two support thresholds. The frequent-item minimum support σ_{item} is used to filter out infrequent items. We will see that this contraint is a key to reduce the number of candidates generated. The second threshold (FT-support threshold) σ_{FT} is used to capture frequent itemsets which allow at most δ missing items per transaction.

An item x is called a globally frequent item iff $\mathrm{support}(x) \geq \sigma_{item}$, i.e., item x appears in at least $\sigma_{item}M$ transactions of the whole database. It follows that an FT-itemset contains only globally frequent items. To avoid triviality, an FT-itemset must have at least ($\delta + 1$) items. We now introduce the following lemma for ($\delta + 1$) FT-itemset [33].

Lemma 1 *Let X be an itemset containing ($\delta + 1$) globally frequent items. X is an FT-itemset iff* $\tilde{\sup}(X) \geq \sigma_{FT}$.

In short, we can say that a ($\delta + 1$)-itemset consisting of globally frequent itemsets is an FT-itemset if it passes the FT-support threshold. In practice, there could be a large number of ($\delta + k$) FT-itemsets when k is small, like $k = 1$ or 2. Note that short FT-itemsets may not be interesting because too many false negatives lead to itemsets without enough actual support. In general, we also apply a minimum length threshold on the size of FT-itemsets as well. We can state our constraint problem as follows: given a transaction database D, a maximum tolerance δ, a frequent-item support threshold σ_{item}, an FT-support threshold σ_{FT} and a length threshold, the problem of fault-tolerant frequent itemsets mining is to find the complete set of FT-itemsets longer than the minimum length threshold. We can now propose a candidate generation-and-test algorithm to compute fault-tolerant frequent itemsets with a constraint on item frequency as shown in Algorithm 5. The initialization of Algorithm 5 is based on Lemma 1. The rest of the algorithm is based on the anti-monotonicity property of FT-itemset (Theorem 1). So, we can show that the algorithm is correct and complete.

4.2.2 Algorithm for finding ETIs

Given the definition of error-tolerant itemset (ETI), the frequent itemset problem can be stated as follow; given a database D of M transactions (rows) and N items (columns), an error tolerance level ϵ, and a minimum support σ in $[0, 1]$, determine all error-tolerant itemsets with respect to D. An error threshold $\epsilon = 0$ reduces Definition 8 to the exact frequent itemset definition. For $\epsilon > 0$, the problem is to determine itemsets exceeding the minimum support threshold, requiring that ($1 - \epsilon$) of the m items in the ETI be present. For example, in Figure 4.1(d), the itemset $\{a, b, c, d, e\}$ is an error-tolerant itemset with minimum support $\sigma = 0.2$ (20%) and $\epsilon = 0.2$.

The level-wise Apriori algorithm does not apply to the ETI problem because unlike the ordinary support of frequent itemsets, the error-tolerant property is not monotonic. Consider an example database in Table 4.1. With a support $\sigma = 1$, $\{a\}$ tolerates error rate of 1.0, $\{a, b\}$ has error of 1.0, $\{a, b, c\}$ tolerates 0.67 error rate and $\{a, b, c, d\}$ tolerates 0.5 error rate. Thus, a levelwise algorithm

Algorithm 4: FT-Apriori

```
Input  : A transaction database $D$, a maximum fixed integer tolerance $\delta$, frequent-item
         support threshold $\sigma_{item}$, FT-support threshold $\sigma_{FT}$ and minimum length thresh-
         old $min_l$.
Output : All FT-itemsets of length $\geq min_l$.
$F_1 \leftarrow \{x : \text{support}(x) \geq \sigma_{item}\}$
Let $C_{\delta+1}$ be the set of all ($\delta + 1$)-subsets of $F_1$. Let $i = \delta + 1$.
repeat
    // prune
    $F_i \leftarrow \{Y : Y \in C_i, \tilde{\sup}(Y) \geq \sigma_{FT}\}$
    if $F_i$ not empty then
        // generate $C_{i+1}$ from $F_i$
        for $U \in F_i$ do
            for $V \in F_1$ do
                $W = U \cup V$;
                if $|W| == i + 1$ then
                    $C_{i+1} = C_{i+1} \cup W$;
    $i = i + 1$;
until $F_{i-1}$ or $C_i$ is empty;
```

Algorithm 5: Constraint-FT-Apriori

```
Input  : A transaction database $D$, a maximum fixed integer tolerance $\delta$, frequent-item
         support threshold $\sigma_{item}$, FT-support threshold $\sigma_{FT}$ and minimum length thresh-
         old $min_l$.
Output : All FT-itemsets of length $\geq min_l$.
$F_1 \leftarrow \{x : \text{support}(x) \geq \sigma_{item}\}$
Run the Apriori algorithm to generate $C_\delta = \{I : |I| = \delta, \text{support}(I) \geq \sigma_{FT}\}$. Let $i = \delta$.
repeat
    // prune
    $F_i \leftarrow \{Y : Y \in C_i, \tilde{\sup}(Y) \geq \sigma_{FT}\}$
    if $F_i$ not empty then
        // generate $C_{i+1}$ from $F_i$
        for $U \in F_i$ do
            for $V \in F_1$ do
                $W = U \cup V$;
                if $\tilde{\sup}(V) \geq \sigma_{item}$ and $|W| == i + 1$ then
                    $C_{i+1} = C_{i+1} \cup W$;
    $i = i + 1$;
until $F_{i-1}$ or $C_i$ is empty;
```

1	2	3	4	5	6	7	8	9	10
1	1	1	1	1					1
1	1	1	1					1	
1	1	1	1	1					
1	1	1	1	1					
1	1	1	1	1					
1	1	1	1	1					
1	1	1	1	1					
1	1	1	1		1	1		1	
1	1	1	1	1	1		1	1	
					1	1	1	1	
					1	1	1	1	
					1	1	1	1	
					1	1		1	
				1					
									1
		1							
			1						
							1		
	1						1		
	1								1

Table 4.2: Strong and weak error-tolerant itemsets (ETIs). With $\sigma = 25\%$ and $\epsilon = 20\%$, $\{1, 2, 3, 4, 5\}$ defines a strong ETI and $\{6, 7, 8, 9\}$ defines a weak ETI.

such as the Apriori algorithm with $\sigma = 1$, $\epsilon = 0.5$ would discard all proper subsets of $\{a, b, c, d\}$, although $\{a, b, c, d\}$ is an error-tolerant itemset.

In the following, we present the algorithm proposed by [49] to find almost all error-tolerant itemsets. In the binary representation of the database D, an error-tolerant itemset is represented as a set of dimensions (columns),called *defining dimensions*, where one appears with high frequency over a set of rows. This algorithm uses two other ETI definitions, a strong ETI and a weak ETI.

Definition 10 *Strong ETI. A strong ETI consists of a set of items, called defining dimensions $\mathbb{D} \subseteq \mathbb{P}$, such that there exists a set of transactions R consisting of at least σM transactions and, for each $r \in R$, the fraction of items in $\mathbb{D}$ which are present in r is at least $1 - \epsilon$.*

Definition 11 *Weak ETI. A weak ETI consists of a set of items, called defining dimensions $\mathbb{D} \subseteq \mathbb{P}$, such that there exists a set of transactions R consisting of at least σM transactions and,*

$$\frac{\sum\limits_{x \in R} \sum\limits_{d \in \mathbb{D}} d(x)}{|R||\mathbb{D}|} \geq (1 - \epsilon),$$

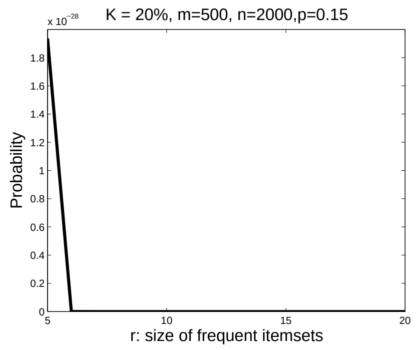


Figure 4.2: Probability of finding an ETI by chance.

where $d(x)$ is one if item d occurs in x and zero otherwise.

For example, in Table 4.2 with $\sigma = 25\%$ and $\epsilon = 20\%$, both $\{1, 2, 3, 4\}$ and $\{1, 2, 3, 4, 5\}$ define a strong ETI and $\{6, 7, 8, 9\}$ defines a weak ETI.

The definition of ETIs fits well with the concept of statistical significance. By the definition of ETI, as the error threshold ϵ increases, the expected number of items in ETIs will also increase. Assume that an entry in an $M \times N$ binary database is chosen at random with probability p. Then the probability of a frequent error-tolerant itemset with r items, support σ and error ϵ to occur is $\binom{M}{\sigma M}\binom{N}{r}p^{(1-\epsilon)\sigma Mr}(1-p)^{\epsilon\sigma Mr}$. This probability decreases exponentially as the size of the itemset grows. See Figure 4.2. For example, for $p = 0.15, \epsilon = 0.2, \sigma = 0.2, N = 2,000, M = 500$ and $r = 5$, we obtains that the probability of finding an ETI with five items by chance is approximately 10^{-30}.

Although the strong and weak ETI property are still not monotonic in the usual sense, the weak ETI has been proven to exhibit another type of monotonicity in the following lemma [49].

Lemma 2 *If E is a weak ETI with $|E| = m$, there exists a weak ETI $E' \subseteq E$ with $|E'| = m - 1$.*

Yang et al. [49] proposed the following exhaustive algorithm to find maximal ETIs by first finding weak ETIs and filtering them in the same manner as the Apriori algorithm [1].

1. Find all items d_i where its occurrence is at least $\sigma M(1-\epsilon)$. Each dimension forms a singleton set $\{d_i\}$ which defines a weak ETI. Set $k = 1$.

2. For every candidate that contains k dimensions, grow it by adding a dimension to it so the new candidate defines a weak ETI. A new candidate has $k + 1$ dimensions.

3. Increment i and repeat step 2 until no more candidates are left.

4. Choose only candidates that satisfy the strong ETI definition.

The exhaustive algorithm is exponential in the number of items. Yang et al. [49] proposed a greedy method to improve the running time and still find almost all ETIs.

4.3 Probabilistic error

Another model of input error is probabilistic. Instead of quantifying errors in terms of the absolute or relative error threshold (as in FTI and ETI, respectively), the experimental error of protein purifications can be quantified as false negative and false positive rate for any interaction between bait and prey. For example, given the binary matrix representation of purifications in Figure 4.3(a), false negative rate (0.2) and false positive rate (0.3), we can assign probability to each observation as in Figure 4.3(b). Formally, given the false negative rate ν and false positive rate ϕ, the input $\hat{D}$ with error can be constructed from the database D as follows:

$$\hat{D}_{ij} = \begin{cases} 1 - \phi & : \quad B(i) \neq j \text{ and } D_{ij} = 1 \\ \nu & : \quad D_{ij} = 0, \end{cases}$$

where $B(i)$ is the bait of the ith purification. Given the input with probabilistic error, we can write the expectation of an itemset I as

$$\mathrm{E}[I] = \sum_{i=1}^{M} \prod_{j \in I} \hat{D}_{ij}.$$

In general, the probabilistic noise present in the input data undermines the correctness of the levelwise algorithm such as the Apriori algorithm which aims to recover itemsets that appear without error in a sufficient fraction of transactions. In fact, it has been shown that, when noise is present, a levelwise algorithm discovers multiple small fragments of the true itemset, but miss the true itemset. The problem is worse for longer itemsets as they are more vulnerable to noise [30]. As we will discuss next, with random noise in the input data, any levelwise algorithm such as the Apriori algorithm will never be able to closely approximate a complete set of probabilistic frequent itemsets.

	a	b	c	d	e
a	1	1	1	0	1
b	1	1	1	0	1
c	1	0	1	1	1
d	1	0	1	1	0
e	1	1	0	1	1

(a) Observed transactions

	a	b	c	d	e
a	1.0	0.7	0.7	0.2	0.7
b	0.7	1.0	0.7	0.2	0.7
c	0.7	0.2	1.0	0.7	0.7
d	0.7	0.2	0.7	1.0	0.7
e	0.7	0.7	0.7	0.7	1.0

(b) Probabilistic interpretation

Figure 4.3: Representation of purifications as database with individual probabilistic errors. The errors are quantified as hypothetical false negative rate (0.2) and false positive rate (0.3). The first column indicates bait proteins.

4.3.1 Fragmentation of patterns by noise

With the simple statistical model, we can observe a significant effect of noise on frequent itemsets mining. Consider a simple model for the observed binary data matrix $\mathbf{Y}$,

$$\mathbf{Y} = \mathbf{X} \oplus \mathbf{Z}, \tag{4.1}$$

where $\mathbf{Y}$, $\mathbf{X}$ and $\mathbf{Z}$ are $m \times n$ binary matrices and $\oplus$ is the entry-wise exclusive-or operation. The matrix $\mathbf{X}$ contains the unobserved "true" data values of interest, in the absence of noise, and $\mathbf{Z}$ is a binary noise matrix whose entries $z_{i,j}$ are independent Bernoulli random variables with $\mathbf{P}[z_{i,j} = 1] = p = 1 - \mathbf{P}[z_{i,j} = 0]$ for some $p \in (0, \frac{1}{2})$. In this case we can write $\mathbf{Z} \sim \text{Bernoulli}(p)$.

Suppose that $m = n$, and let $K(\mathbf{Y})$ be the largest k such that the matrix $\mathbf{Y}$ contains a $k \times k$ submatrix of 1s, or equivalently, the largest k such that $\mathbf{Y}$ contains k transactions having k common items. The following proposition is proposed in [43]. It extends the earlier result on the clique number of random graphs by Bollobás [5] to binary random matrices.

Proposition 1 *With probability* 1, $K(\mathbf{Y}) \leq 2\log_a n - 2\log_a \log_a n$ *when* n *is sufficiently large, regardless of the structure of* $\mathbf{X}$. *Here* $a = \frac{1}{1-p}$.

Proposition 1 states that, even for small noise levels $p > 0$, large blocks of 1s or other structures in the true matrix $\mathbf{X}$ leave behind only fragments of logarithmic size in $\mathbf{Y}$. Thus, no exact frequent itemset mining algorithm will be able to recover such underlying structure directly from $\mathbf{Y}$.

To demonstrate this effect, we perform the experiment of adding noise to a square matrix of

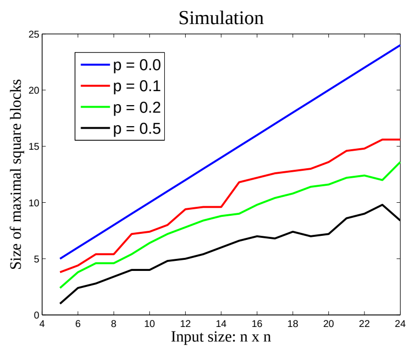


Figure 4.4: When noise is present with probability p, the observed size of largest k sets having k common items, $k \times k$ sub-matrix, increases far more slowly than the original input size (Proposition 1).

1s [30]. Each entry of the initial matrix was changed to 0 with some probability p independently. This matrix represents an input database of n sets and n items with only false negative error. We then applied the Apriori algorithm to the corrupted matrix and applied the experiment to matrices of different sizes. Figure 4.4 plots the size of the largest recovered square sub-matrix of 1s (largest frequent itemsets) against the size of the original matrix, for different values of p (corresponding to different levels of data corruption). In the presence of noise, only a fraction of the initial block of 1s was recovered, and this fraction diminished rapidly with an increase in the size of the original matrix.

The result is directly related to our problem of finding subcomplex modules from purification data and it agrees with our hypothesis that only a fragment of modules was discovered by the exact frequent itemset mining. The failure of the Apriori algorithm to detect simple patterns in the presence of random errors causes it to miss true association when such errors are present. Clearly the exhaustive algorithm (Algorithm 6) is exponential in the number of items and consequently this solution is not feasible. In the following section, we propose a greedy algorithm with pruning strategy to recover probabilistic frequent itemsets.

4.3.2 Algorithm for probabilistic frequent itemsets

Because the exhaustive algorithm that combines candidates of different length is exponential in the number of items, it is not a feasible solution. We employ a greedy strategy to improve the running time by keeping at each length only a fixed number of candidate itemsets, that have the

Algorithm 6: Exhaustive algorithm

```
Input  : A database $D$, false negative rate $\nu$, false positive rate $\phi$, maximum itemset size
         $k_{\max}$
Output : Probabilistic frequent itemsets of length $1 \ldots k_{\max}$
$C[1] \leftarrow \{\{i\} : i \in \mathbb{P}, \mathrm{Freq}(i) \geq 2\}$
$k = 2$
while $k \leq k_{\max}$ do
    for $l = 1$ to $\frac{k}{2}$ do
        JoinCandidate($C[k-l]$, $C[l]$, $L$);
    $C[k] \leftarrow L$;
    $k$++;
```

highest expectation values. In this strategy, the time consuming part of the algorithm is candidate generation and computation of expectation. The trivial implementation of the greedy strategy is given in Algorithm 8. The candidate generation is done by the procedure JoinCandidate. In this procedure, a new candidate of length k is generated from a union of two itemsets of length $k-l$ and l and it is accepted only if those two sets are disjoint. Candidates of length k generated thus far are kept in a set data structure to prevent insertion of duplicate itemsets. Let $C[k]$ be a set of candidates of length k and $|C[k]|$ its cardinality. For each length k, the JoinCandidate procedure (Algorithm 8) takes time $|C[k-l]| \times |C[l]|$ with the worst case memory requirement of N_C^2 to store candidates, where N_C is a user-defined maximum number of candidates per length. To eliminate itemsets with low expectation values, we compute the expectation for all candidates and keep only the N_C most frequent itemsets with highest expectation. Candidates are extended from length k to $k+1$ until the specified maximum length is reached.

Because the number of candidates kept are usually much fewer than the number of candidates generated, the memory requirement can be reduced by using a priority queue data structure to maintain more frequent candidates. Instead of computing and storing all candidates before computing expectation, the expectation value can be computed during the candidate generation as in the procedure JoinCandidatePQ (Algorithm 10) and only N_C frequently expected itemsets are stored in the priority queue, greatly reducing the amount of memory required.

Procedure `JoinCandidate(`$C[k-1]$, $C[l]$, L`)`

```
Input   : $C[k-l]$: frequent itemsets of length $k-l$, $C[l]$: frequent itemsets of length $l$,
          $L$:current candidate itemsets of length $k$
foreach $U \in C[k-l]$ do
    foreach $V \in C[l]$ do
        $W = U \cup V$;
        if $|W| == |U| + |V|$ then
            $L \leftarrow L \cup W$;
```

Algorithm 8: Greedy algorithm

```
Input   : A database $D$, false negative rate $\nu$, false positive rate $\phi$, maximum itemset size
          $k_{\max}$, number of candidates $N_C$
Output  : $N_C k_{\max}$ frequent itemsets of length $1 \dots k_{\max}$
$C[1] \leftarrow \{\{i\} : i \in \mathbb{P}, \mathrm{Freq}(i) \geq 2\}$
$k = 2$
while $k \leq k_{\max}$ do
    for $l = 1$ to $\frac{k}{2}$ do
        JoinCandidate($C[k-l]$, $C[l]$, $L$);
    foreach $I \in L$ do
        $\mathrm{E}[I] = \sum_{i=1}^{M} \prod_{j \in I} \hat{D}_{ij}$;
    $X \leftarrow \arg\operatorname{sort}_{I \in L}(\mathrm{E}[I])$;
    $C[k] \leftarrow X[1 \dots N_C]$;
    $k$++;
```

Procedure `JoinCandidatePQ(`$C[k-l]$, $C[l]$, $\mathrm{PQ}[k]$`)`

```
Input   : $C[k-l]$: frequent itemsets of length $k-l$, $C[l]$: frequent itemsets of length $l$,
          $\mathrm{PQ}[k]$:priority queue storing candidate itemsets of length $k$
foreach $U \in C[k-l]$ do
    foreach $V \in C[l]$ do
        $W = U \cup V$;
        if $|W| == |U| + |V|$ then
            if $\mathrm{PQ}[k].size() < N_C$ then
                $\mathrm{PQ}[k].union(W)$;
            else
                $Z = \mathrm{PQ}[k].top()$;
                if $\mathrm{E}[W] > \mathrm{E}[Z]$ then
                    $\mathrm{PQ}[k].pop()$;
                    $\mathrm{PQ}[k].union(W)$;
```

Algorithm 10: Greedy algorithm with priority queue

```
Input  : A database $D$, false negative rate $\nu$, false positive rate $\phi$, maximum itemset size
         $k_{\max}$, number of candidates $N_C$
Output : $N_C k_{\max}$ frequent itemsets of length $1 \ldots k_{\max}$
$C[1] \leftarrow \{\{i\} : i \in \mathbb{P}, \mathrm{Freq}(i) \geq 2\}$
$k = 2$
while $k \leq k_{\max}$ do
    for $l = 1$ to $\frac{k}{2}$ do
        JoinCandidatePQ($C[k-l]$, $C[l]$, $\mathrm{PQ}[k]$);
    $k$++;
```

4.4 Merging maximal frequent itemsets

When the experimental error is assumed to only consist of false negative proteins, another approach to recover frequent itemsets with errors is merging of maximal frequent itemsets. A disadvantage of this approach is that a rule for merging is based on heuristics, without a quantitative model of false negative error. In summary, this approach involves the following:

- Compute maximal exact frequent itemsets at low minimum support,
- Merge overlapping maximal frequent itemsets if the size of overlapping items is greater than a given threshold. The support of the new set is the sum of support of its subsets,
- Delete all subsets that are not maximal frequent itemsets,
- Repeat the merging steps until all sets are maximal.

The computation of maximal frequent itemsets can be done very quickly and there are many efficient algorithms to compute such sets. We use an algorithm called MAFIA which provides a freely available implementation (`http://himalaya-tools.sourceforge.net/Mafia/`). For protein purifications, the number of maximal frequent itemsets are between 500 and $20,000$ sets depending on the minimum support level. Therefore, the most time consuming part of this algorithm is the merging step which is quadratic in the number of maximal frequent itemsets. Efficient data structure such as suffix trees to store maximal frequent itemsets can improve the running time, but we choose the naive implementation.

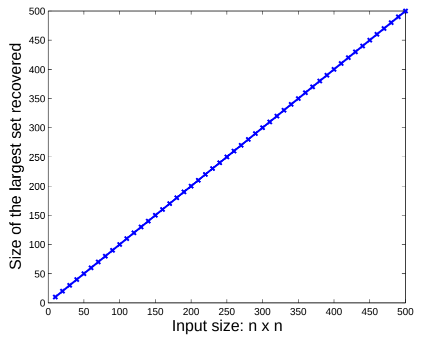
4.5 Result and discussion

We have performed two experiments to test the effectiveness of our algorithm to handle errors: on simulated data and on protein purification data. First, we wanted to see if the algorithm for probabilistic frequent itemsets can improve the fragmentation of patterns on simulated square matrix of 1s by comparing the size of the largest recovered square sub-matrix discovered by our greedy algorithm to the result from the Apriori algorithm. For the second experiment on purification data, we tried to improve the sensitivity of prediction due to recovery of longer frequent itemsets, when compared to the result obtained from the exact frequent itemsets.

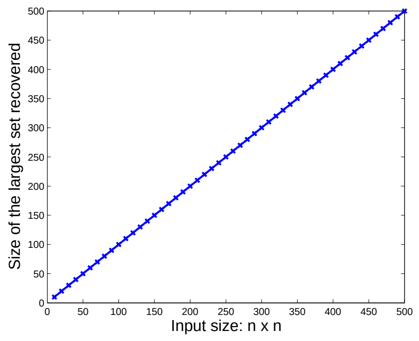
To test the recovery of corrupted input, we performed the same experiment of adding noise to a square matrix of 1s. We then applied our greedy algorithm to the corrupted matrix. We performed the experiment for various matrix sizes. Figure 4.5 plots the size of the largest recovered sub-matrix of 1s (largest frequent itemsets) against the size of the original matrix, for different values of p. The important factor here is a number of candidates kept per length. In this example, even with a small number of candidates, we can still recover the original input size. We tested two values of maximum candidates per length: 5 and 100 for input sizes from $[10, 500]$. The result shows that our greedy algorithm is able to recover the original input matrix even when keeping only very few candidates.

Our algorithm still suffers from the fact that it relies on the user specified maximum itemset size and maximum number of candidates per length. These two quantities are important because they help limit the search space for frequent itemsets. For longer frequent itemsets, our algorithm is still $\mathrm{O}(N_C^3)$. Long frequent itemsets pose a problem for both our algorithm and the Apriori algorithm. Due to noise, the Apriori algorithm only recovers their fragments, while our algorithm needs to run much longer to recover them.

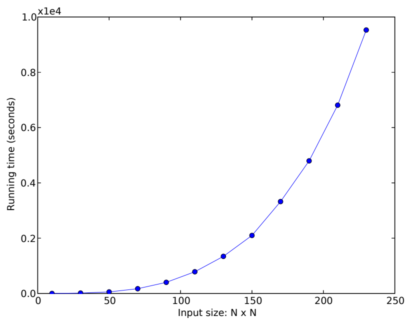
When evaluating the prediction accuracy on protein purifications data, we unfortunately do not significantly improve the prediction made by the exact frequent itemset mining. We restrict the evaluation to the bait proteins of the Gav02 and Gav06 sets. Figure 4.6 compares the analysis of the Gav02 data set by fault-tolerant itemsets to exact frequent itemsets. Figure 4.7 compares probabilistic frequent itemsets to exact frequent itemsets. Figure 4.8 compares the maximal merging approach to exact frequent itemsets. All three advanced algorithms have similar accuracy as exact frequent itemsets, but run much longer, because the number of candidates is much larger for both



(a) No. candidates per length = 5



(b) No. candidates per length = 100



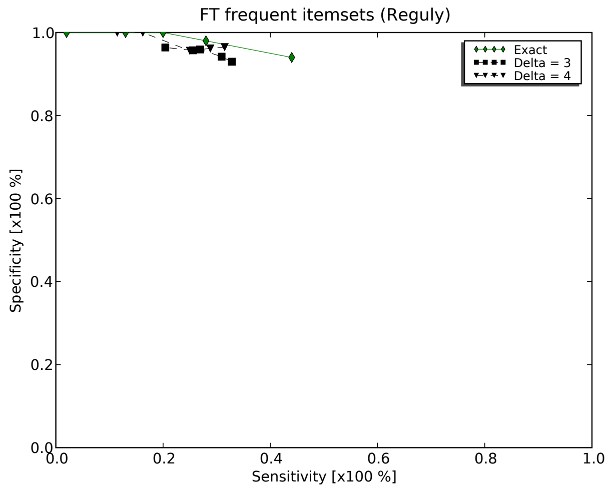
(c) Running time (max. candidates per length = 100)

Figure 4.5: Recovery of the largest frequent itemsets on a square input matrix with noise parameter $p = 0.3$. The input size is taken from $[10 : 20 : 500]$. Our algorithm is able to recover the original input set even with much fewer candidates.

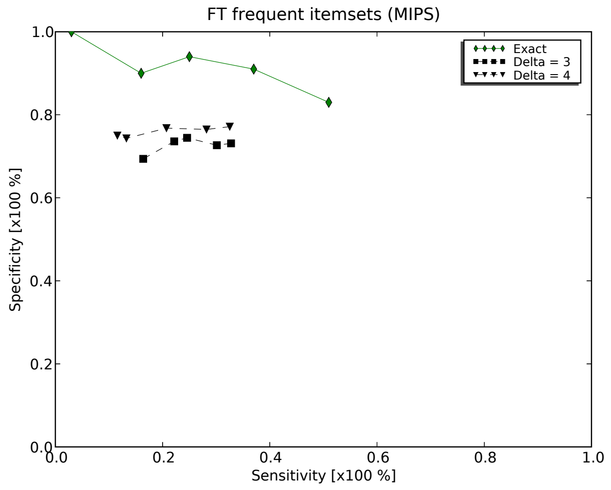
fault/error-tolerant itemsets and probabilistic itemsets. The number of candidates examined in fault-tolerant itemsets at high error and low support is exponential. It is not even feasible to examine them. For the probabilistic frequent itemsets, the increase in running time over the Apriori algorithm is cubic.

Of all techniques discussed in this chapter, merging maximal exact frequent itemsets performs best on real world data. It provides good results and is more efficient than the other techniques. It is able to handle thousands of transactions and proteins and still yield results comparable to other clustering solutions. See Figure 4.9.

Efficient approximation of frequent itemsets with any kind of error remains a challenging problem mainly because the number of candidates needed to be examined can grow exponentially as the number of error increases. To predict overlapping protein complexes, the results in this chapter show that mining exact frequent itemsets is a reasonable approach, even though the sensitivity is not as good as of partitioning clustering. We will show in Chapter 5 that, given contemporary data sets, a partitioning model can better predict protein complexes than an overlapping model. Furthermore, with frequent itemset techniques, accounting for error in purifications does not significantly improve accuracy, all the while requiring longer running time and more memory. Therefore, we conclude that among the methods we tested, finding exact frequent itemsets is the best and most efficient method to predict overlapping complexes.

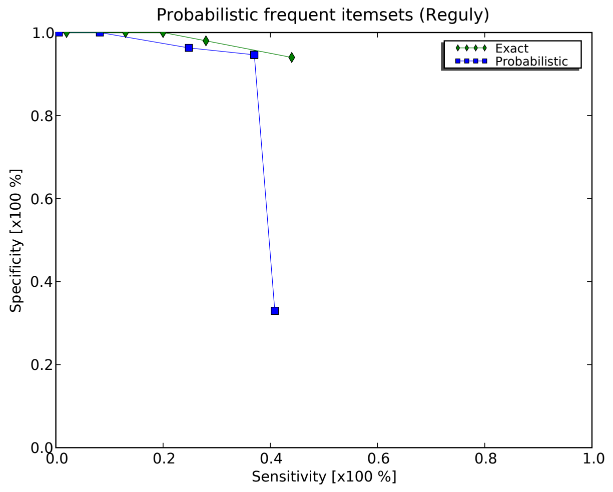


(a) Reguly benchmark

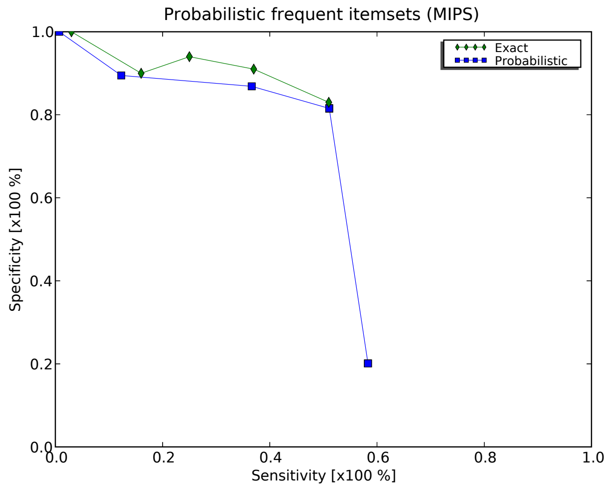


(b) MIPS benchmark

Figure 4.6: Fault-tolerant itemsets on the Gav02 data set: (a) on Reguly benchmark, compared with exact solution and (b) on MIPS benchmark, compared with exact solution. The evaluation is restricted to the bait proteins.

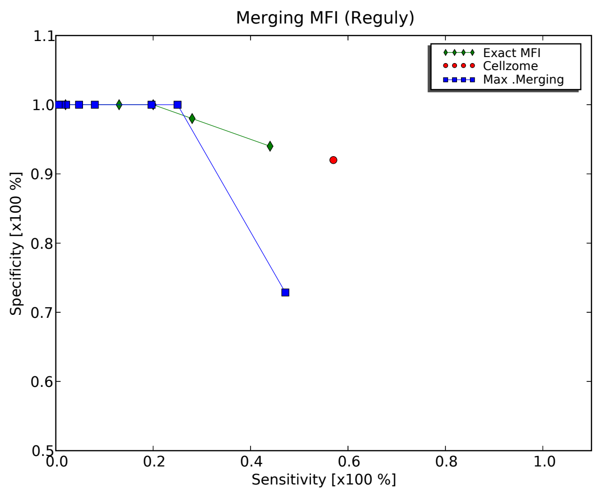


(a) Reguly benchmark

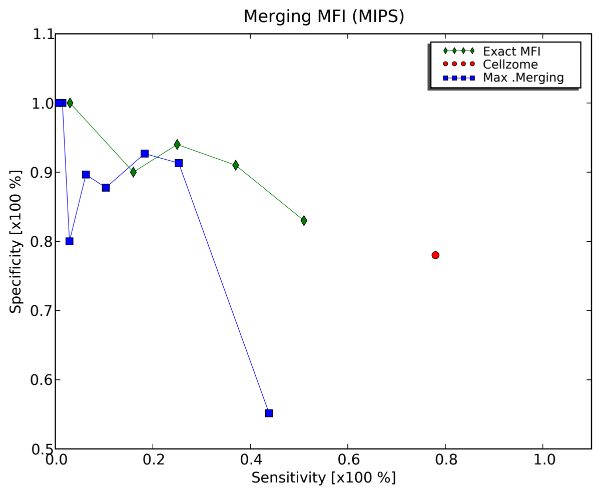


(b) MIPS benchmark

Figure 4.7: Probabilistic frequent itemsets on the Gav02 data set: (a) on Reguly benchmark, compared with exact solution and (b) on MIPS benchmark, compared with exact solution. The evaluation is restricted to the bait proteins.



(a) Reguly benchmark



(b) MIPS benchmark

Figure 4.8: Merging of maximal frequent itemsets on the Gav02 data set: (a) on Reguly benchmark, compared with exact solution and (b) on MIPS benchmark, compared with exact solution. The evaluation is restricted to the bait proteins.

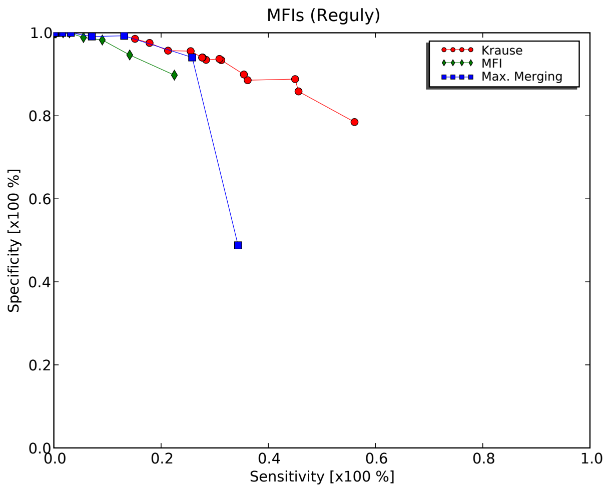


Figure 4.9: Merging of maximal frequent itemsets on the Gav06 data set, on Reguly benchmark, compared with clustering solutions of Krause. The evaluation is restricted to the bait proteins.