Chapter 4

FAR-Miner: Algorithm for Fuzzy Association Rule Mining

Many real-life datasets found nowadays (especially those from big corporations and government organizations) are becoming huger in size (> 1M records), and their fuzzy versions would be even huger. Thus, there is an acute need to innovate a fast fuzzy ARM algorithm as an alternative to Fuzzy Apriori. To this end, we propose a very fast fuzzy algorithm, FAR-Miner [MP09], which performs several times quicker than fuzzy Apriori, especially for huge datasets. In fact, our experimental results detailed in section 4.4 indicate at least 8–19 times speed up for large datasets (1M-10M), and at least 6–10 times speed up for medium-sized datasets (500K-1M), as compared to Fuzzy Apriori. FAR-Miner is loosely based on the ARMOR algorithm [PH03], [PH01], which deals with crisp association rules.

Generation of fuzzy association rules using an appropriate fuzzy ARM algorithm is not a straight-forward process. First, we need to convert the crisp dataset, containing crisp binary and numerical attributes, into a fuzzy dataset, containing crisp binary and fuzzy binary attributes. This involves substantial pre-processing using appropriate techniques, like the one described in chapter 3 and in [MP10b]. Conversion of numerical attributes to fuzzy binary attributes is a part of such pre-processing. Second, any crisp ARM algorithm cannot be used to generate fuzzy association rules just by providing a fuzzy dataset as input. A crisp ARM algorithm and its fuzzy version would be disparate in many ways because crisp data is by nature different from fuzzy data. The major difference is in the counting process; crisp ARM algorithms calculate the frequency of an itemset just by looking for its presence or absence in a transaction of the dataset, but fuzzy ARM algorithms need to take into account the fuzzy membership of an itemset in a particular transaction of the dataset, in addition to its presence or absence. In essence, a crisp ARM algorithm and its fuzzy version are very different in the way they are designed and implemented, and in the way they operate. And, any conversion of a crisp ARM algorithm to its fuzzy version is not trivial and straightforward.

The basic premise behind developing FAR-Miner is to make sure that we use all major resources of a computer, namely processor, main memory, and secondary memory, in the best manner possible so as to generate fuzzy association rules quickly and efficiently. Most other ARM algorithms, crisp or fuzzy ones, fail to utilize all these three resources to the best possible way. But unlike these algorithms, FAR-Miner makes optimum use of each of these resources:

- On the main memory front, FAR-Miner uses all available/allocated RAM, in order to generate fuzzy association rules as soon as possible.
As far as secondary memory is concerned, it makes a full scan of the dataset only once, in the first phase. In addition, it makes a second quick scan in the second phase only if required.

Finally, it makes the best possible use of the processor by minimizing the processing required, and performing only those computations absolutely essential to generate fuzzy association rules.

The goal of FAR-Miner is to work efficiently on large datasets. Such datasets cannot be handled properly by totally in-memory algorithms like FP-Growth, but only by algorithms which are not totally memory dependent, like Apriori and ARMOR. Thus, we have compared FAR-Miner against fuzzy Apriori, which is the only fuzzy ARM algorithm available currently. It is very clear that FAR-Miner makes the best possible use of all available resources as opposed to fuzzy Apriori which does not do the same. This aspect of FAR-Miner helps it a lot to score over fuzzy Apriori in terms of speed and efficiency.

Thus, our contribution in this chapter is the proposal of a new fuzzy ARM algorithm called FAR-Miner which is much faster than fuzzy Apriori, the most popular fuzzy ARM algorithm in use today. This high degree of efficiency and accuracy is achieved through a combination of many novel fuzzy ARM features. To our knowledge there is no scalable and efficient algorithm to mine fuzzy association rules, although there are many such algorithms available for mining crisp association rules. The forte of FAR-Miner is that it can be used to mine fuzzy association rules in datasets of any size, even large ones, with any number and type (numerical or binary) of attributes. This makes it stand out among the rest of the algorithms with respect to the state-of-the-art. We also show how appropriate pre-processing should be done to a crisp dataset in order to trigger the actual fuzzy ARM process.

4.1 Fuzzy Association Rules and their Characteristics

As mentioned in the introduction, by using fuzzy partitions for numerical attributes, we move from an environment involving only crisp association rules to one which has fuzzy association rules. For fuzzy ARM, a number of fuzzy partitions can be defined on the domain of each quantitative attribute. Thus, the original dataset is transformed into an extended one with attribute values in the interval $[0, 1]$. Mining can then be done on the transformed fuzzy dataset to obtain fuzzy association rules.

To obtain such rules, the measures pertaining to crisp association rules have to be generalized in a suitable way for the fuzzy environment. A fuzzy set $A$ in $D$ is a $D \rightarrow [0, 1]$ mapping. Fuzzy-set-theoretical counterparts of complementation, intersection, and union are defined by means of a negator, a t-norm, and a t-conorm \cite{CCK03, CCK05}. An increasing, associative, and commutative $[0, 1]^2 \rightarrow [0, 1]$ mapping is called a t-norm (Eq. 4.1) if it satisfies $T(x, 1) = x, \forall x \in [0, 1]$, and a t-conorm $S$ (Eq. 4.2) if it satisfies $S(x, 0)) = x, \forall x \in [0, 1]$. A negator $N$ (Eq. 4.3) is a decreasing $[0, 1] \rightarrow [0, 1]$ mapping satisfying $N(0) = 1$ and $N(1) = 0$.

Let $A(x)$ be the degree to which an attribute $A$ is present in a transaction $x$. Thus, $A$ can be thought of as a fuzzy partition, and the crisp measures discussed in section 2.1 should be generalized accordingly to get fuzzy measures. The cardinality of a fuzzy set in a finite universe $D$ is defined by Eq. 4.4.

$$A \cap_T B(x) = T(A(x), B(x))$$

(4.1)
Replacing the set-theoretical operations by their fuzzy-set-theoretical counterparts, we obtain:

$$\text{support}(A \Rightarrow B) = \frac{\sum_{x \in D}(A \cap_T B)(x)}{|X|}$$

(4.5)

$$\text{confidence}(A \Rightarrow B) = \frac{\sum_{x \in D}(A \cap_T B)(x)}{\sum_{x \in D}(A)(x)}$$

(4.6)

The definitions of fuzzy support and fuzzy confidence shown in equations 4.5 and 4.6 are used in the generation of fuzzy association rules by most fuzzy ARM algorithms, including FAR-Miner. The more generally used t-norms and t-conorms are listed in Table 3.1. And, $T_M$ t-norm is the most popular t-norm used for generating fuzzy association rules. The same t-norm is used in FAR-Miner.

For example, we take the association rule $AB \Rightarrow X$, and the associated itemsets $AB$ and $ABX$. We check record-by-record for $AB$ and $ABX$ and their respective membership values ($\mu$s). For $AB$, the item with the lowest $\mu$ contributes the overall $\mu$ to $AB$ (by virtue of $T_M$ t-norm) for that particular record. If for example, $\mu_A = 0.7$ and $\mu_B = 0.5$, then $\mu_{AB} = 0.5$. Similarly, $\mu_{ABX}$ is also calculated for each record. For example, in the same record if $\mu_X = 0.4$, then as in the above example $\mu_{ABX}$ can be calculated as $0.4$. The overall frequency of $AB$ and $ABX$ is the sum of all $\mu_{AB}$s and $\mu_{ABX}$s calculated for all records in the dataset. If $\text{freq}(AB) = 100$, $\text{freq}(ABX) = 50$ and dataset size = 1000, then after the aforementioned record-by-record calculation, by equations 4.5 and 4.6 support = 0.05 and confidence = 0.5.

### 4.2 FAR-Miner

In this section, we describe FAR-Miner and how it is used for fuzzy rule generation. But, before any rule-mining process can ensue, we need to pre-process the original crisp dataset. The fuzzy pre-processing methodology we use is explained in detail in chapter 3 and in [MP10b]. It enables us to obtain Gaussian-type fuzzy sets for numerical attributes. The bottom-line of this methodology is to use fuzzy clustering as a means of generating fuzzy partitions. Though this methodology is seemingly simple and straightforward, it is effective, accurate, and fast. This pre-processing approach consists of two phases:

- Generation of fuzzy partitions for numerical attributes
- Conversion of a crisp dataset into a fuzzy dataset using a standard way of fuzzy data representation.
Thus, in an iterative manner, each attribute is processed, until all attributes have been exhausted, and we get the final fuzzy version of the dataset. We use this pre-processing technique to convert any crisp dataset \( D \) to its corresponding fuzzy dataset \( E \), with fuzzy records. \( E \) is then used as input to FAR-Miner.

We use the following notations:

- \( E \) = fuzzy dataset generated after pre-processing
- Set of partitions \( P = P_1, P_2, \ldots, P_p \)
- Set of sub-partitions \( SP = SP_1, SP_2, \ldots, SP_q \)
- \( d \) = number of partitions (for any particular itemset \( it \)) that have been processed since the partition in which it was added (including the current partition and the partition in which it was added).
- \( \mu \) = fuzzy membership of any itemset
- \( L \) = global list of itemsets (with their respective \( \mu \)s) which have been enumerated
- \( F \) and \( F' \) = lists (or hash-tables/hash-maps) of \( d \)-frequent \( k \)-itemsets (with their respective tidlists) for the current level of \( k \)

### 4.2.1 FAR-Miner and Fuzzy Association Rule Generation

FAR-Miner operates in two steps to generate fuzzy association rules. In addition, it also facilitates a partition-approach to the processing of any fuzzy dataset generated after appropriate pre-processing. This partition approach enables us to use the main memory in the most effective manner, without any thrashing. The dataset is logically divided into \( p \) disjoint horizontal partitions \( P_1, P_2, \ldots, P_p \). Each partition is as large as can fit in the available main memory. For ease of exposition, we assume that the partitions are equal-sized. However, each partition could be of any arbitrary size as well. Our aim is to make the number of partitions \( p \) as close as possible to 1, i.e. \( p \rightarrow 1 \), and in this section we show how this is possible. Using this scheme, itemsets being counted are enumerated only at the end of each partition, and not after every tuple.

The size of a partition directly influences the amount of memory used by the algorithm. Lesser the number of partitions, faster is the computation, as only those itemsets which are potential candidates to become frequent are processed. Having too many partitions would mean that itemsets which are unnecessary and will not become frequent are also processed. On the other hand, having too few partitions may make the memory required by the algorithm exceed the actual amount of memory available/allocated, thus leading to thrashing. Thus, the size of each partition and in effect the number of partitions should be such that nearly all available/allotted main memory is used without any overspill. This would ensure that as few unnecessary itemsets as possible are processed without any thrashing. More the amount of RAM allocated, faster is the processing. Because computers do not have enough RAM to fit tidlists of all itemsets, especially in the case of large datasets, we need to use partitions. Best possible scenario is to use all allocated RAM to achieve the best possible speed on the computer being used, without any thrashing.
The itemsets can be viewed as a directed acyclic graph (DAG) as shown in Fig. 4.1. ARMOR uses a list-like data structure in which the tidlists are maintained. This data structure stores the transaction id (tid) where the itemset occurs for each itemset. Extending the same for the fuzzy context, this data structure would store the fuzzy membership $\mu$ of the itemset in that particular transaction in addition to the tid (Fig. 4.2). In contrast to this, FAR-Miner uses a byte-vector-like data structure (Fig. 4.3). Each cell of the byte-vector stores the $\mu$ of the itemset corresponding to the cell index of the tid to which the $\mu$ pertains. Thus, the $i^{th}$ cell of the byte-vector contains the $\mu$ for the $i^{th}$ tid. If a particular transaction does not contain the itemset under consideration, the cell corresponding to that transaction is assigned a value of 0. When the byte-vector is initialized, each cell by default has 0.

![Figure 4.1 DAG containing power set of \{A, B, C, D\}](image)

![Figure 4.2 List-like data structure for tidlists](image)

Though $\mu$ is a floating-point number, we convert it to a byte by multiplying it by 100, and taking the ceiling of the result to get the final byte representation of the $\mu$. Thus, each $\mu$ can be represented with a precision of two digits after the decimal point. Also, $\mu$ which can have a floating-point value ranging from 0.00 to 1.00, is saved in a range from 0 to 100 in the byte-vector. Whenever we need the actual floating-point value of $\mu$, we just divide the value obtained from the byte-vector by 100. In this way, we achieve a lot of saving in main memory space, and thus speed up the execution of the algorithm by reducing the number of partitions required ($p \rightarrow 1$) and by avoiding thrashing. To save each $\mu$ with a
higher precision, we could use the same technique but with a larger-sized primitive like integer. But, the int-vector would occupy two times or four times (depending on the language and compiler being used) the space occupied by the byte-vector.

Moreover, FAR-Miner uses a generalized compression technique to compress all tidlists created. A tidlist-based approach enables us to enumerate all itemsets in a dataset by scanning it just once. But, the downside is that tidlists can be large, and may occupy a lot of space in the main memory. One remedy is to use multiple partitions, which ARMOR and consequently FAR-Miner, already do. This problem can be alleviated by the proper compression of tidlists. Some ARM algorithms, like VIPER [SHS00], use compression techniques which work well only in a few situations and with few types of data. In fact, most such compression techniques do not work very well in circumstances or with data even slightly different from those that they are designed for. Moreover, some also need a few parameters to be set by the user or the programmer. And, setting these parameters for all types of data might not be easy for the user. For example, VIPER needs the parameters $W_0$ and $W_1$ be set by the programmer. VIPER is the best known ARM algorithm to provide a compression facility. Though compression offers a brilliant solution to save space in main memory, and thus speed up the execution of the algorithm, the actual deflation and inflation processes require substantial processing power and main memory space, and can thus make the execution of algorithm very slow (due to incessant thrashing), if these resources are not readily available. This problem assumes more significance considering the numerous tidlists that would be created during the course of execution of an ARM algorithm.

We have overcome this problem by using an appropriate compression algorithm (zlib compression algorithm) in FAR-Miner. It is a generalized technique which works very well with just about any kind of data, especially tidlists which are in raw byte form (byte-vector). And, this technique uses very less main memory space and processor time for the deflation and inflation processes, especially when the compression method is set for best speed. Using this compression technique, we have achieved considerable speed up, and are able to get any dataset processed with far fewer number of partitions than would have been required in case no compression were used.

### 4.2.1.1 First Phase

In the first phase, FAR-Miner scans each transaction in the current partition of the dataset, and constructs a tidlist for each singleton found (Algorithm 4, lines 1–36). After all singletons in the current
partition have been enumerated, the singletons which are $d$-frequent are expanded further to generate 2-itemsets. An itemset is $d$-frequent if its frequency over $d$ partitions equals or exceeds the support adjusted for $d$ partitions, i.e. it is frequent over $d$ partitions of the dataset (where $d \leq$ number of partitions in the dataset).

The threshold used to determine $d$-frequency is the minimum support pertaining (proportional) to the number of partitions over which an itemset has been processed. For example, if five partitions are used and the minimum support is 0.2 (dataset size 100; absolute_minimum_support = 20), then in the third partition of the first phase, an itemset is $d$-frequent if its frequency over the first three partitions that have been processed is $\geq 12(\frac{3}{5} \times 20)$.

All singletons, irrespective of their frequency, are put in $L$ with their respective counts. If the singleton is already present in $L$, then its count is updated to the cumulative count till the current partition. But the tidlists for singletons which are in the negative border, i.e. not $d$-frequent, are removed. Only the tidlists for singletons which are $d$-frequent are carried forward to generate the tidlists for 2-itemsets. An itemset is said to be in the negative border if its frequency is less than the minimum support. We cannot derive a frequent child itemset from such an itemset.

Unlike ARMOR or any other partition-based ARM algorithm, FAR-Miner uses the concept of sub-partitions. Each partition is further divided into $q$ equal-sized sub-partitions. The sub-partitions have been made equal-sized just for the sake of simplicity in implementation of the algorithm, and could have been of any arbitrary sizes. When FAR-Miner processes any partition $i$, it does not scan the whole partition in one shot. On the contrary, it does so in a sub-partition-by-sub-partition manner. For each partition $i$, it scans each sub-partition $j$, and in the process, creates the tidlists for all singletons found in sub-partition $j$. At the end of the sub-partition scan, the tidlist for any singleton $s$ is appended to the cumulative tidlist of $s$ created till the $(j - 1)^{th}$ sub-partition. Of course, as the cumulative tidlist till the $(j - 1)^{th}$ sub-partition exists in compressed form in the main memory, it is uncompressed before this appending operation can take place. After the appending, the new cumulative tidlist till the $j^{th}$ sub-partition is compressed. It is needless to say that the old tidlists till the $(j - 1)^{th}$ sub-partition (compressed and uncompressed forms) and the tidlist for just the $j^{th}$ partition for each singleton are discarded. This process of alternating creation and compression of tidlists for each sub-partition speeds up the dataset scan and also the tidlist generation for singletons. Without this sub-partition technique, the algorithm would start thrashing on the onset itself, i.e. during singleton enumeration, without even starting the process for the enumeration of larger itemsets (with $k \geq 2$).

To generate larger itemsets, we use breadth-first search (BFS) technique in a fashion similar to one used in Apriori (Algorithm 4, lines 37–50). We just use BFS for simplicity sake and to illustrate that ARMOR (originally implemented using DFS) or any ARMOR-like algorithm can be implemented using BFS also. Sometimes computation (creating an itemset) with DFS may take slightly longer than that taken by BFS, as a few unnecessary infrequent itemsets may also be generated, which would not be the case with BFS because of the use of Apriori property. For example, lets us take three itemsets, $AB$, $AC$, and $ABC$, with $ABC$ being generated as a result of the combination of $AB$ and $AC$. With DFS we would necessarily generate $ABC$, even if $BC$ were infrequent. But, with BFS we would not generate $ABC$ at all if $BC$ were infrequent.
At the $k$th level, each $(k-1)$-itemset is combined with another $(k-1)$-itemset to generate a $k$-itemset, if the two $(k-1)$-itemsets differ by just one singleton (Algorithm 5, lines 1–23). The tidlist for each $k$-itemset $r$ is generated by intersecting the tidlists of its parent $(k-1)$-itemsets $p$ and $q$. Tidlist creation is done as soon as an itemset has been created. The tidlists of the parents are uncompressed before we can proceed further. To minimize the number of times uncompressing is done, we fix one parent itemset $p$ and try to combine it with all other parent itemsets to obtain child itemsets. After $p$ has been combined with all other parent itemsets, its tidlist can be discarded. By doing so, we save main memory space and reduce the number of inflations.

We use the $T_M$ fuzzy t-norm for the intersection process. During the intersection of parent tidlists, we look at each cell with index $l$ of the two byte-vectors (tidlists are in byte-vector form). We obtain two fuzzy membership values $\mu_1$ and $\mu_2$ from the $l^{th}$ cells of the parent byte-vectors. The resultant fuzzy membership $\mu_3$ for the $l^{th}$ cell of the child byte-vector is the minimum of $\mu_1$ and $\mu_2$. By taking the minimum, we are in effect applying the $T_M$ fuzzy t-norm. Any other t-norm could also be used. The intersection is done for each cell of the parent byte-vectors (tidlists) to obtain the child byte-vector (tidlist).

After the creation of the tidlist for each $k$-itemset, we check to see if it is $d$-frequent or not. If yes, then the tidlist is compressed for further generation of tidlists of child itemsets, i.e. $(k+1)$-itemsets. If not, the tidlist is discarded. But irrespective of its $d$-frequency, the itemset and its frequency are put in $L$. If the itemset is already present in $L$, then its count is updated to the cumulative count till the current partition. But, supersets of itemsets in the negative border (not $d$-frequent) are removed from the global list $L$. The definition of $d$-frequent (for singletons), as detailed above, can be extended to larger itemsets also. Finally, the uncompressed versions of the tidlists of the parent itemset $q$ and the current child itemset are discarded.

We then generate all possible $k$-itemsets by exhausting the combination process for all $(k-1)$-itemsets. A similar process, of alternating itemset generation and tidlist generation, is ensued for higher levels of $k$, until no more itemsets can be generated, i.e. at any level $k$, we have only one $d$-frequent itemset. In a similar manner, we process each partition, until all partitions have been exhausted. This marks the end of the first phase of FAR-Miner, after which $L$ contains all $d$-frequent itemsets and itemsets in the negative border, with their respective $d$-frequencies.

### 4.2.1.2 Second Phase

Then we move on to the second phase of the algorithm. ARMOR executes the second phase in more or less the same manner in which it executes the first phase. But, the second phases of ARMOR and FAR-Miner are totally different. The basic motive behind these modifications to the second phase is to reduce the time taken by the second phase as much as possible, to make this time negligible in comparison to that taken by the first phase, and to obtain better efficiency.

First, we output all frequent itemsets (with their respective counts) in $L$ that were added in any partition $i$ (in the first phase), before actually processing $i$ in the second phase. The support for this frequency comparison is based on the whole dataset. Also, any itemset added in partition $i$ in the first phase, would have been enumerated over the whole dataset, and thus can be deleted from $L$ (Algorithm 6).
Then, we create the tidlists for all singletons in partition \( i \), in a manner similar to the one followed in the first phase (using sub-partitions and compression of tidlists after every sub-partition). After that, for each itemset \( r \) in \( L \), we create its tidlist by intersecting the (uncompressed) tidlists of all singletons involved in \( r \). The corresponding count of \( r \), obtained from the tidlist, is updated in \( L \) (Algorithm 6 lines 7–14). The uncompressed versions of the tidlists of the singletons are discarded after the intersection. Thus, we alternate between outputting and deleting itemsets from \( L \), and creating tidlists for itemsets, until no more itemsets are left in \( L \). Then the algorithm terminates (Algorithm 6 lines 15–20).

### 4.2.2 Illustration of FAR-Miner and Fuzzy Association Rule Generation

Let us take an example of two itemsets \( p \) ("ABC") and \( q \) ("BCD"). Their child itemset would be \( r \) ("ABCD"). List-style (as in ARMOR) and byte-vector-style (as in FAR-Miner) representations of their tidlists are illustrated in Figs. 4.2 and 4.3 respectively. Assuming that there are five partitions, \( i.e. \ p = 5 \), and that \( r \) became \( d \)-frequent and was added to \( L \) in the third partition in the first phase. Tidlists for \( r \) are generated in the third through fifth partitions, assuming that no parent of \( r \) goes into the negative border. Thus, at the end of the first phase, the frequency of \( r \) is available for the third through fifth partitions.

In the second phase, we enumerate the frequency of \( r \) in partitions 1 and 2. We generate tidlists of all singletons possible for each of these two partitions. Then we generate tidlists for \( r \) once in each of the two partitions 1 and 2, by taking an intersection of the tidlists of the singletons involved in \( r \), \( i.e. \ "A", "B", "C", \) and \( "D" \). Thus, at the start of processing of the third partition (in the second phase), \( r \) would have been enumerated over the whole dataset, and can be output, if frequent over the whole dataset \( E \). It would also be deleted from \( L \). All other itemsets are processed in a similar manner. The algorithm terminates, when there are no more itemsets left in \( L \) at start of a partition (in the second phase).

### 4.2.3 Space and Time Complexities of FAR-Miner

The following is the notation used for the analysis of theoretical complexities of space and time:

- \(|E|\) = size of the fuzzy dataset generated after pre-processing
- \(t_{avg\_size}\) = average tidlist size
- \(c\) = inverse compression ratio \((1 \leq c < \infty)\)
- \(|p|\) = number of partitions
- \(|i_{d\_freq}|\) = number of all \( d \)-frequent itemsets in a particular partition \( p \) in the first phase
- \(|i_{counted}|\) = number of all itemsets that need to be counted (\( d \)-frequent itemsets and their negative border) in the first phase.
- \(|i_{neg\_border}|\) = number of all itemsets in the negative border in the first phase.
- \(|s|\) = number of all \( d \)-frequent singletons in a particular partition \( p \) in the second phase
• \(|i_{d-freq,1}| = number of \(d\)-frequent itemsets at end of first phase

The space complexity (Eq. 4.7) of the first phase of FAR-Miner is a function of \(|i_{d-freq}| and \(t_{avg-size}\), and is inversely proportional to \(c\) and \(p\). Thus, less the number of partitions, less the space complexity. Similarly, more the compression ratio, less the space complexity. The time complexity (Eq. 4.8) of the first phase is directly proportional to the number of itemsets being counted over the whole fuzzy dataset. This includes all \(d\)-frequent itemsets and their negative border. As mentioned in Section [4.2.1.1] we use BFS for candidate itemset generation. If DFS were used instead, then the time complexity would also have to take into account all infrequent itemsets that are generated as a result of DFS not using the Apriori property.

The space complexity (Eq. 4.9) of the second phase is directly proportional to the number of \(d\)-frequent singletons in each partition, whatever be the number the actual \(d\)-frequent itemsets. Though this does not reduce the space complexity theoretically, but practically it reduces the overall space (memory) required. The time complexity (Eq. 4.10) of the second phase is much lower than that of the first phase, and is proportional to just the number of \(d\)-frequent itemsets (enumerated over \(E\)) that remain at the end of the first phase.

\[
O \left( \frac{|i_{d-freq}| \times t_{avg-size}}{c \times p} + |i_{neg-border}| \right) \quad (4.7)
\]

\[
O(|i_{counted}| \times |E|) \quad (4.8)
\]

\[
O \left( \frac{s \times t_{avg-size}}{c \times p} \right) \quad (4.9)
\]

\[
O(|i_{d-freq,1}| \times |E|) \quad (4.10)
\]

### 4.3 Performance Study Model

In section [4.2] we have described the FAR-Miner algorithm and in this section, we assess the performance of FAR-Miner with respect to fuzzy Apriori, which is the most popular and widely used online fuzzy mining algorithm. Our experiments cover two disparate datasets and various mining workloads, both typical and extreme ones, with various values of support. Specifically, the datasets we have considered have sizes significantly larger than the available main memory. The performance metric in all the experiments is the total execution time taken by the mining operation. We have performed each of the experiments on two different computers with different configurations, especially in terms of RAM sizes:

- Computer \(C_1\) – AMD Sempron 2600+ (1.6 GHz), 512 MB DDR RAM, and PATA 7200 RPM HDD.
- Computer \(C_2\) – Intel Core 2 Duo 2.8 GHz, 1 GB DDR2 RAM, SATA 7200 RPM HDD.
Because both FAR-Miner and fuzzy Apriori (like most other ARM algorithms) are monolithic in nature, their implementations are sequential and not parallel in nature. Thus, though Computer C_2 has two cores, only one of them is used at a time by these algorithms. These implementations have been written in Java. The two different real-life datasets, one medium sized and one large, used for testing are:

- **FAM95**: Of the 23 attributes in the dataset, we have used the first 18, of which six are quantitative and the rest are binary. For each of the six quantitative attributes, we have generated fuzzy partitions using FCM, and then generated the fuzzy version of the dataset (using a threshold for membership function \( \mu \) as 0.1) as detailed in Chapter 3. The original dataset contains around 63K transactions, and the fuzzy version, on which the actual fuzzy mining was performed, has more than 425K transactions.

- **USCensus1990raw**: We have used 12 attributes present in the dataset, of which eight are quantitative and the rest are binary. The fuzzy version of the dataset was generated in the same manner as that in FAM95. The crisp dataset has around 2.5M transactions and the fuzzy dataset has around 10M transactions.

### 4.4 Experimental Results

In this section we present the results of the performance study detailed in section 4.3.

#### 4.4.1 Experiment 1: On Dataset USCensus1990raw

Fig. 4.4 illustrates the results obtained on \( C_1 \) on the USCensus1990raw dataset, using various values of support ranging from 0.075 to 0.4. It can be clearly observed that FAR-Miner performs 8–19 times faster than fuzzy Apriori, depending on the support used. Please note that the execution times for fuzzy Apriori for support values 0.075 and 0.1 have not been calculated as the time exceeded 50K seconds. Fig. 4.5 illustrates the results obtained by running the same experiment on \( C_2 \). On this computer, we observe that FAR-Miner has speeds nearly 8-13 times that of fuzzy Apriori. More importantly, for any dataset there is a particular support value for which decent number of itemsets is generated and for supports less than this value, we get a flood of itemsets which are of no practical use. From our experiments, we have observed that FAR-Miner performs most efficiently and speedily at this support value, which occurs in the range of 0.15–0.2 for this dataset.

We had claimed in section 4.2 that we would endeavor to reduce the number of partitions as much as possible (\( p \to 1 \)). We have actually executed our algorithm with just one partition, i.e. \( p = 1 \), for support values 0.2–0.4 on \( C_1 \), and 0.1–0.4 on \( C_2 \). This means the second phase is not required for these support values, and the corresponding second phase execution times are 0. Less number of partitions means faster processing and less consumption of resources like main memory and processor. If we try to forcibly increase the number of partitions for a dataset (with a specific support), which can be processed with lesser number of partitions, we end up increasing the overall time required for processing. More
number of partitions results in a greater chance of more itemsets becoming \(d\)-frequent (though finally only overall frequent itemsets are output), and more time is spent generating and analyzing far greater number of itemsets.

For other support values, \(p\) is as close to 1 as possible, keeping in mind that the main memory is utilized in the best manner possible, without any thrashing. Moreover, we also corroborate our previous claim that the time taken by the second phase (where \(p \geq 2\)) would be negligible as compared to that taken by the first phase. From these experiments, we observe that the time taken by the second phase ranges from 0.02–0.1 times (depending on the support value and number of partitions used) that taken by the first phase. Furthermore, with our compression technique, we have got compressed tidlists which are 2%–20% the sizes of uncompressed tidlists.

Figures 4.6 and 4.7 do a comparison of the time taken for execution as the number of partitions is varied. In Fig. 4.6, we see that as the number of partitions is increased, the time taken for itemset generation also increases progressively. Fig. 4.7 illustrates the same comparison for minimum support
= 0.075, but with a small difference that the number partitions used starts from two. When just one partition was used, thrashing occurred, but with two partitions all available/allocated memory was used to the fullest, providing the fastest execution possible. As is the case in Fig. 4.6 an increase in the number of partitions resulted in an increase in execution time.

![Figure 4.6](image)

**Figure 4.6** Effect on execution time as number of partitions is varied

![Figure 4.7](image)

**Figure 4.7** Effect on execution time as number of partitions is varied (for minimum support = 0.075)

### 4.4.2 Experiment 2: On Dataset FAM95

Fig. 4.8 illustrates the results obtained on $C_1$ and $C_2$ on the FAM95 dataset, using various values of support ranging from 0.15 to 0.4. On both the computers, we have achieved 6–10 times speed ups for FAR-Miner as compared to fuzzy Apriori, even for this medium-sized dataset. Significantly, on both the computers and for all values of support, we have used just one partition, i.e. $p = 1$, which means all the fuzzy association rules are generated in the first phase itself, and the second phase is not required at all.
Even for this dataset, FAR-Miner performs best for the support range we mentioned in Section 4.4.1 for which decent number of rules are generated.

![Figure 4.8 Experiment 2 on \(C_1\) and \(C_2\)](image)

**Figure 4.8** Experiment 2 on \(C_1\) and \(C_2\)

4.4.3 Effect of Compression on Tidlist Generation

Fig. 4.9 does a comparison of the average time (in milliseconds) taken for each deflation, inflation, and tidlist generation for an itemset given two parent itemsets (for FAM 95 dataset). The average time taken for inflation is nearly 0.65–0.75 times that taken for deflation. And, the average time taken for tidlist generation is much greater than that required for either inflation (5.5–10 times) or deflation (4.2–5.9 times). A similar kind of comparison is done for the USCensus1990raw dataset in Fig. 4.10. As is the case in the previous comparison done in Fig. 4.9, the average time taken for inflation is lower (0.67–0.75 times) than that taken for deflation. The average time taken for tidlist generation is comparable to that taken for either inflation (0.9–4.6 times) or deflation (0.65–3.15 times).

On either dataset there is some overhead that deflation and inflation add. This overhead varies according to the type of dataset and the minimum support threshold used. But, this overhead is negligible as compared to the speed-up achieved due to in-memory compression of tidlists. A compression of 2% to 20% means that if we had not used any compression at all, then we would have required nearly 4 to 50 times the number of partitions we have currently used. Thus, this drastic increase in the number of partitions indicates the extremely slow speed of itemset generation we would have got if no compression were used.

4.4.4 Comparison of FAR-Miner and ARMOR

Fig. 4.11 does a comparison of the ratio of execution times of second and first phases (execution times of second phase/first phase) of ARMOR and FAR-Miner as minimum support is varied. This ratio is 0.1–0.42 for ARMOR, but just 0.02–0.1 for FAR-Miner. Interestingly, the ratio increases as minimum support threshold is reduced in the case of ARMOR, but for FAR-Miner the ratio reduces as minimum
support is reduced. This clearly indicates not only that the second phase of FAR-Miner is much faster and better than that of ARMOR, but also that the second phase of FAR-Miner takes comparatively less time (especially at minimum support threshold range when the most favorable number of itemsets is generated, around 0.2 in this case) as minimum support is reduced.

4.4.5 Corroboration of Space and Time Complexities through Experimental Results

The theoretical complexities presented in section 4.2.3 are confirmed by the experimental results presented in section 4.4. From an execution time perspective, the first phase takes much longer than the second phase (Fig. 4.11), which corroborates the time complexities (time complexity of the first phase being a magnitude higher than that of the second phase) calculated in section 4.2.3. As far as the space complexity is concerned both phases have the same magnitudes, but the space complexity of the
second phase in only dependent on the number of singletons in each partition where as that of the first phase is dependent on the number of all possible frequent itemsets at each level $k$ in each partition. This theoretical result is again corroborated by the experimental results as the second phase requires memory only to store the tidlists of singletons in each partition, which is considerably lower than that required by the first phase to store the tidlists of all frequent $k$-itemsets at each level $k$ in each partition. According to common knowledge of ARM, the number of itemsets tends to be more at level $k = 2$ to 5 as compared to other levels of $k$. Thus, the memory requirements of the first phase are also highest at level $k = 2$ to 5, when the number of itemsets is very high (much higher than at $k = 1$ - singletons or at $k \geq 6$).

### 4.4.6 Comparison of Number of Rules Generated for Different Supports

Fig. 4.12 illustrates the number of fuzzy association rules generated for the USCensus1990raw datasets as support is varied. We see that for very low supports, there is a deluge of fuzzy association rules generated. For very high supports, very few rules are generated, thereby providing no insights at all. When support is around 0.15–0.2 a decent (not too many or too less) number of rules are generated, which can help better understand the dataset. Fig. 4.13 illustrates the same information as Fig. 4.12 but for the FAM95 dataset. The observations of the FAM95 dataset are nearly similar to that made about the USCensus1990raw dataset. Thus, the change in number of fuzzy association rules generated as support is varied follows the same pattern (curve) for most datasets.

### 4.5 Summary of FAR-Miner

We have presented a novel fuzzy ARM algorithm called FAR-Miner as an alternative to fuzzy Apriori, which is the most widely used algorithm for fuzzy ARM. Through our experiments on two different sized real-life datasets, we have shown that FAR-Miner is 8–19 times faster for large fuzzy datasets, and 6–10 times faster for medium-sized datasets, as compared to fuzzy Apriori. The main reasons for the considerable speed up achieved by FAR-Miner are properties like two-phased tidlist-style processing.
Figure 4.12 Variation in number of fuzzy association rules for changes in support (USCensus1990raw dataset)

Figure 4.13 Variation in number of fuzzy association rules for changes in support (FAM95 dataset)

using partitions, tidlists in the form of byte-vectors, effective compression of tidlists, faster generation of new tidlists from the intersection of parent tidlists, and a tauter and quicker second phase of processing.
for each partition $P_i \in P$ where $i = 1, 2, \ldots, p$ do

$F = \emptyset$

for each sub-partition $SP_j \in SP$ where $j = 1, 2, \ldots, q$ do

$F' = \emptyset$

for each transaction $t_f \in SP_j$ do

$F' = \emptyset$

for each singleton $s \in t_f$ do

$\mu = \text{fuzzy membership of } s \text{ in } t_f$

if $s \in F'$ then

$td' = \text{tidlist (byte-vector) of } s \text{ obtained from } F'$

else

$td' = \text{new tidlist created for } s$

end if

$td'[f] = \mu$

update $td'$ for $s$ in $F'$

if $s \in L$ then

$\mu' = \text{cumulative fuzzy membership of } s \text{ in } L$

$\mu' += \mu$

update $\mu'$ for $s$ in $L$

else

put $s$ along with $\mu$ in $L$

end if

end for

for each $s \in F'$ do

$td = \text{tidlist of } s \text{ obtained from } F$

$td' = \text{tidlist of } s \text{ obtained from } F'$

$td = td \cup td'$

compress($td$)

update $td$ for $s$ in $F$

end for

end for

for each $s \in F'$ do

end for

end for

for each $s \in L$ do

if $s$ is not $d$-frequent then

remove $s$ and corresponding $td$ from $F$ {/*prune s*/}

end if

end for

while $|F| \geq 2$ do

$F' = F$

$F = \emptyset$

for $x = 1$ to $|F'| - 1$ do

itemset $it_1 = F[x]$

$td_1 = \text{tidlist of } it_1 \text{ obtained from } F$

uncompress($td_1$)

for $y = (x + 1)$ to $|F'|$ do

itemset $it_2 = F[y]$

intersect($it_1, it_2, td_1, F, F'$)

end for

end for

end while

end for

Algorithm 4: Pseudo-code for Phase 1
Algorithm 5: Pseudo-code for function intersect()

```plaintext
1: intersect(it1, it2, td1, F, F')
2: µ = 0
3: it3 = it1 ∩ it2
4: td2 = tidlist of it2 obtained from F'
5: uncompress(td2)
6: td3 = new tidlist for it3
7: for z = 1 to partition size p do
8:   td3[z] = T_M(td1[z], td2[z]) = min(td1[z], td2[z]) /*apply t-norm T_M*/
9:   µ += td3[z]
10: end for
11: if it3 is d-frequent then
12:   compress(td3)
13:   put td3 for it3 in F
14: else
15:   remove all supersets of it3 and corresponding µs from L
16: end if
17: if it3 ∈ L then
18:   µ' = cumulative fuzzy membership of it3 in L
19:   µ' += µ
20: update µ' for it3 in L
21: else
22:   put it3 along with µ in L
23: end if
```

Algorithm 6: Pseudo-code for Phase 2

```plaintext
1: for each partition P_i ∈ P where i = 1, 2, ..., p do
2:   for each itemset it ∈ L added in P_i in the first phase do
3:     if it is frequent over the whole dataset E then
4:       output(it)
5:     end if
6:   remove it from L /*generate singleton tidlists*/
7:   same as lines 2 to 29 of the first phase (Algorithm 4)
8:   for each itemset it ∈ L do
9:     identify singletons s_1, s_2, ..., s_m ∀ it = s_1 ∩ s_2 ∩ ... ∩ s_m
10:    td = uncompress(td_1) ∩ uncompress(td_2) ∩ ... ∩ uncompress(td_m) ∀ td_1, td_2, ..., td_m are tidlists of s_1, s_2, ..., s_m obtained from F /*tidlist intersection is same as in phase 1*/
11:    µ = calculate using td /*same as in phase 1*/
12:    µ' = cumulative fuzzy membership of it in L
13:    µ' += µ
14:    update µ' for it in L
15:   end for
16: if L = ∅ then
17:   exit
18: end if
19: end for
20: end for
```

42