Mining Relationships Among Interval-based Events for Classification

Dhaval Patel Wynne Hsu Mong Li Lee
School of Computing
National University of Singapore
Singapore
{dhaval,whsu,leeml}@comp.nus.edu.sg

ABSTRACT

Existing temporal pattern mining assumes that events do not have any duration. However, events in many real world applications have durations, and the relationships among these events are often complex. These relationships are modeled using a hierarchical representation that extends Allen's interval algebra. However, this representation is lossy as the exact relationships among the events cannot be fully recovered. In this paper, we augment the hierarchical representation with additional information to achieve a lossless representation. An efficient algorithm called IEMiner is designed to discover frequent temporal patterns from interval-based events. The algorithm employs two optimization techniques to reduce the search space and remove non-promising candidates. From the discovered temporal patterns, we build an interval-based classifier called IEClassifier to differentiate closely related classes. Experiments on both synthetic and real world datasets indicate the efficiency and scalability of the proposed approach, as well as the improved accuracy of IEClassifier.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications - Data mining

General Terms

Algorithms

Keywords

Interval-based Event Mining, Temporal Relation, Classifier for Interval Data

1. INTRODUCTION

Temporal pattern mining aims to discover useful relations that are hidden among events. Existing temporal mining algorithms [1, 10, 3, 12] have focused on discovering frequent temporal patterns from instantaneous events, that is, events with no duration. This assumption allows the discovered pattern to be simplified to an ordered sequence of events, such as "fever → stomach ache → vomit". However, such sequential patterns are inadequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the events' durations could play an important role.

For example, it has been observed that in many diabetic patients, the presence of hyperglycemia[1] overlaps with the absence of glycosuria[2]. This insight has led to the development of effective diabetic testing kits. In the case of dengue fever, knowing that there will be a decrease of platelet counts on the third day after the onset of fever has led to a better management of the disease. Clearly, there is a need for an efficient mining algorithm that can discover complex relationships among events with duration, also known as interval-based events. Furthermore, these discovered relationships could be used to build a classifier that is able to distinguish closely related classes.

Mining complex interval-based relationships efficiently requires an unique yet lossless representation to capture the temporal relationships among events. Allen's interval algebra [2] has traditionally been used to represent the temporal relationship between two interval-based events. However, capturing the temporal relationships among three or more events remains an issue. Many approaches use a hierarchical representation [6, 14] to encode the temporal relationships among events. This representation is lossy as it does not preserve the underlying temporal structure of the events. Any mining algorithm that is based on a lossy representation will lead to the discovery of many spurious patterns as non-frequent patterns may become frequent.

Existing interval-based mining algorithms are either based on a lossy representation [8] or do not scale well [11, 15]. We overcome these drawbacks by devising a lossless representation. Based on the proposed representation, we design an interval-based event mining algorithm. The algorithm employs two optimization techniques to reduce the search space and remove non-promising candidates. Taking a step further, we examine how the discovered temporal patterns can be utilized in classification to differentiate closely related classes. To the best of our knowledge, this is the first work to build an interval-based classifier.

The contributions of this paper are as follow:

[1]high concentration of glucose in the blood
[2]presence of glucose in the urine

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

1. We augment the hierarchical representation with count information to achieve a lossless representation. We provide a proof that the augmented representation is indeed lossless. This enables us to recover the actual relationships among events in the mining process.

2. We design an Apriori-based algorithm called "IEMiner"(**I**nterval-based **E**vent **Miner**) to discover frequent temporal patterns based on the lossless representation. IEMiner employs two optimization strategies to reduce the search space. The proof of the completeness of IEMiner is detailed.

3. We also build an interval-based temporal pattern classifier called IEClassifier to perform the classification of closely related classes. We apply the classifier to a real world Hepatitis dataset to demonstrate its improved accuracy.

Experimental studies on both synthetic and real world datasets indicate that IEMiner is both efficient and scalable and outperforms state-of-the-art algorithms. The IEClassifier improves the predictive accuracy on the real world Hepatitis dataset over traditional classifiers such as CBA [9], C4.5 [13] and SVM [5].

The rest of the paper is organized as follows. Section 2 gives the related work. Section 3 provides some preliminaries. Section 4 introduces the augmented hierarchical-based representation. Section 5 describes the IEMiner algorithm and the optimization strategies. Section 6 presents the design of IEClassifier. Section 7 gives the experiment results and we conclude in Section 8.

2. RELATED WORK

There has been a stream of research on mining sequential patterns [3, 12, 10, 1]. These works assume that events have zero duration. Recent works have investigated the mining of interval-based events [8, 11, 15]. Kam et. al. [8] design an algorithm that uses the hierarchical representation to discover frequent temporal patterns. However, the hierarchical representation is ambiguous and many spurious patterns are found.

Papapetrou et. al. [11] propose the H-DFS algorithm to mine frequent arrangements of temporal intervals. Both these works transform an event sequence into a vertical representation using id-lists. The id-list of one event is merged with the id-list of other events to generate temporal patterns. This approach does not scale well when the temporal pattern length increases.

Wu et. al. [15] devise an algorithm called TPrefix for mining non-ambiguous temporal pattern from interval-based events. TPrefix first discovers single frequent events from the projected database. Next, it generates all the possible candidates between temporal prefix and discovered frequent events and scans the projected database again for support counting. TPrefixSpan has several inherent limitations: multiple scans of the database is needed and the algorithm does not employ any pruning strategy to reduce the search space.

Discovered sequential patterns can be leveraged to obtain high confidence association rules useful for classification. Liu et. al. [9] propose a classification scheme based on association rules to improve the prediction accuracy. Recently, Cheng et. al. [4] present a detailed study of frequent pattern analysis for classification. In their approach, high discriminating frequent patterns are discovered from the non-sequential data. These patterns are used as additional features for classification. In our approach, we discover discriminative frequent patterns from sequential data and employ them for classification.

3. PRELIMINARIES

An event is denoted by $E = (type, start, end)$, where $E.type$ denotes the type of event, $E.start$ and $E.end$ denote the event's start and end time respectively. We use E_i to denote the i^{th} event. An ordered event list $EL = \{E_1, E_2, \cdots, E_i, \cdots, E_n\}$ is a collection of events, sorted by the event start time, followed by the event end time, in an ascending order. The length of EL, given by $|EL|$, is the number of events in the list. For example, the ordered event list EL_1 in Figure 1 is {A,B,C,D} which has a length of 4.

Each event in an event list has a temporal relation with all the other events in the list. Table 1 shows the 13 temporal relations defined by Allen [2] that can occur between any two interval-based events E_i and $E_j, i \neq j$.

A new composite event E is formed when a temporal relation R is applied to two events E_i and E_j. We denote $E = (E_i\ R\ E_j)$. The start and end times of E are given by $\min\{E_i.start, E_j.start\}$ and $\max\{E_i.end, E_j.end\}$ respectively.

Let D be a set of ordered event lists. A temporal pattern TP is of the form (E, sup) where E is a composite event and sup is the number of event lists from D that supports E. The length of TP is given by the number of events in E. A temporal pattern of length n is also known as a n-pattern, denoted by TP_n. A k-subpattern of TP_n is a composite event consisting of k events in E, $k < n$. A k-prefix of TP, $k < |E|$, is a composite event with the first k events in E.

The support of a pattern TP is the number of event lists in D in which TP occurs. If sup is no less than a user specified support threshold $minsup$, we say TP is a frequent pattern in D. We define the problem of mining interval-based temporal patterns as finding the frequent set of temporal patterns in a database D. Note that a temporal pattern satisfies the downward closure property. Hence, if TP is frequent, then all its sub-patterns are also frequent.

4. AUGMENTED HIERARCHICAL REPRESENTATION

A composite event is traditionally modeled using a hierarchical representation. While the hierarchical representation provides an attractive and compact mechanism to express the temporal relations among events, it is not unique. For example, the composite event involving the four events of EL_1 in Figure 1 has many possible representations, such as
$((A\ Overlap\ B)\ Overlap\ C)\ Contain\ D)$ or
$(A\ Overlap\ C)\ Contain\ (B\ Before\ D)$ etc.

A canonical representation of the composite event can be obtained if we generate the relations between events according to their start times. Thus,
$((A\ Overlap\ B)\ Overlap\ C)\ Contain\ D)$ is the canonical representation for EL_1. Similarly canonical representation of EL_2 is $((A\ Before\ F)\ Before\ G)$.

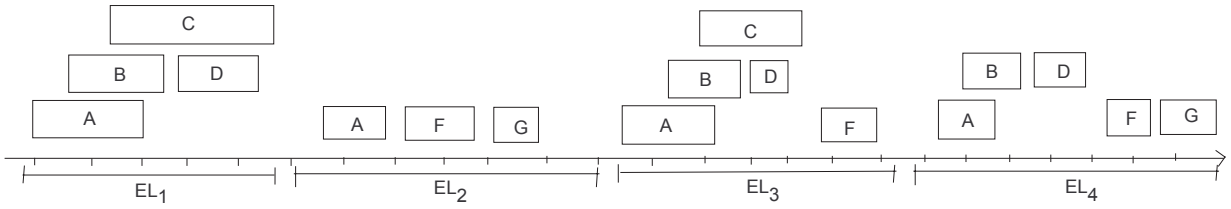

Figure 1: Example database of event lists

Table 1: Temporal relation between event E_i and E_j

Relation	Interval Algebra	Dual Relation
E_i Before E_j	$(E_i.end < E_j.start)$	After
E_i Meet E_j	$(E_i.end = E_j.start)$	Met-by
E_i Overlap E_j	$(E_i.end > E_j.start) \wedge (E_i.end < E_j.end)$ $\wedge\ (E_i.start < E_j.start)$	Overlapped-by
E_i Start E_j	$(E_i.start = E_j.start) \wedge (E_i.end < E_j.end)$	Started-by
E_i Finished-by E_j	$(E_i.end = E_j.end) \wedge (E_i.start < E_j.start)$	Finish
E_i Contain E_j	$(E_i.start < E_j.start) \wedge (E_i.end > E_j.end)$	During
E_i Equal E_j	$(E_i.start = E_j.start) \wedge (E_i.end = E_j.end)$	Equal

Conversely, given a canonical representation (A Overlap B) Overlap C, we cannot infer whether C is overlapped-by only B or C is overlapped-by both A and B in the corresponding ordered event list. Figure 2 show the different interpretations of temporal patterns. Such multiple interpretations will result in an incorrect inference of the exact relationship among events. To overcome this problem, we augment the hierarchical representation with additional information.

We observe that the first two cases in Figure 2 can be differentiated by using an *overlap count* to track the number of events in E that actually overlaps with C. For example, the *overlap count* for Figure 2(a) and 2(b) is 1 and 2 respectively. Figure 2(c) can be differentiated by using an additional *meet count* to indicate the number of events in E that meets C.

An exhaustive enumeration shows that we need 5 variables, namely, *contain count* c, *finish_by count* f, *meet count* m, *overlap count* o, and *start count* s to differentiate all the possible cases. Figure 3 shows a partial listing of the various cases. We augment the representation for a composite event E to include the count variable as follows:

$$E = (E_1\ R_1[c,f,m,o,s]\ E_2)\ R_2[c,f,m,o,s]\ E_3) \cdots R_{n-1}[c,f,m,o,s]\ E_n)$$

Thus, the temporal patterns in Figure 2 are represented as

(A Overlap[0,0,0,1,0] B) Overlap[0,0,0,1,0] C

(A Overlap[0,0,0,1,0] B) Overlap[0,0,0,2,0] C

(A Overlap[0,0,0,1,0] B) Overlap[0,0,1,1,0] C

In order to prove that the augmented representation is lossless, we use the concept of linear ordering of an event list. Given an ordered event list, a linear ordering is obtained by the chronological order of the start and end points of the events in the list. For example, the linear ordering of EL_1 in Figure 1 is :

$\{\{A+\}\{B+\}\{C+\}\{A-\}\{B-\}\{D+\}\{D-\}\{C-\}\}$

where + indicates an event's start point and − indicates an event's end point. A representation is lossless if we can recover the complete linear ordering of the start and end times of all the events which correspond to the underlying ordered event list.

THEOREM 1. *The augmented hierarchical representation is lossless.*

PROOF. We prove the theorem by induction.

Base case: A composite event consisting of two events is lossless. This inferred directly from the Interval algebra between two events given in Table 1.

Induction Step: Suppose a composite event E^k consisting of k events is lossless. Let $E^{k+1} = E^k\ R[c,f,m,o,s]\ E$ be a composite event consisting of k+1 events where E is a new event.

Since E^k is lossless, we can recover the linear ordering of the k events. With the new event E, we observe that the start time of E is constrained as follows:

1. $E.start = E'.start$ for all E' where E' is an event in E^k and E' Start E. The number of events that satisfy this condition is given by the *start count* s.

2. $E.start = E'.end$ for all E' where E' is an event in E^k and E' Meet E. The number of events that satisfy this condition is given by the *meet count* m.

3. $E.start < E'.end$ for all E' where E' is an event in E^k and E' Overlap E or E' *Finished_By* E or E' Contain E or E' Start E. The number of events that satisfy this condition is given by the *overlap count* o, *finished_by count* f, *contain count* c and *start count* s.

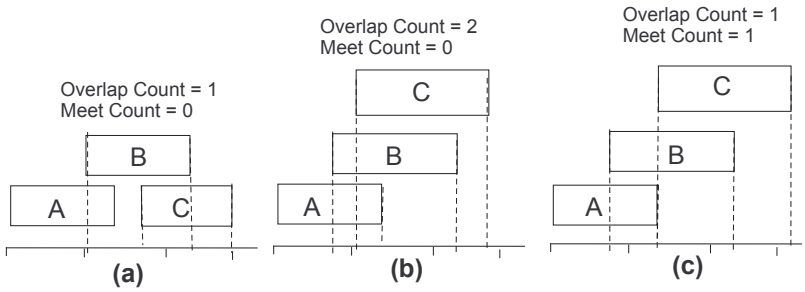

Figure 2: Interpretations of pattern (A Overlap B) Overlap C

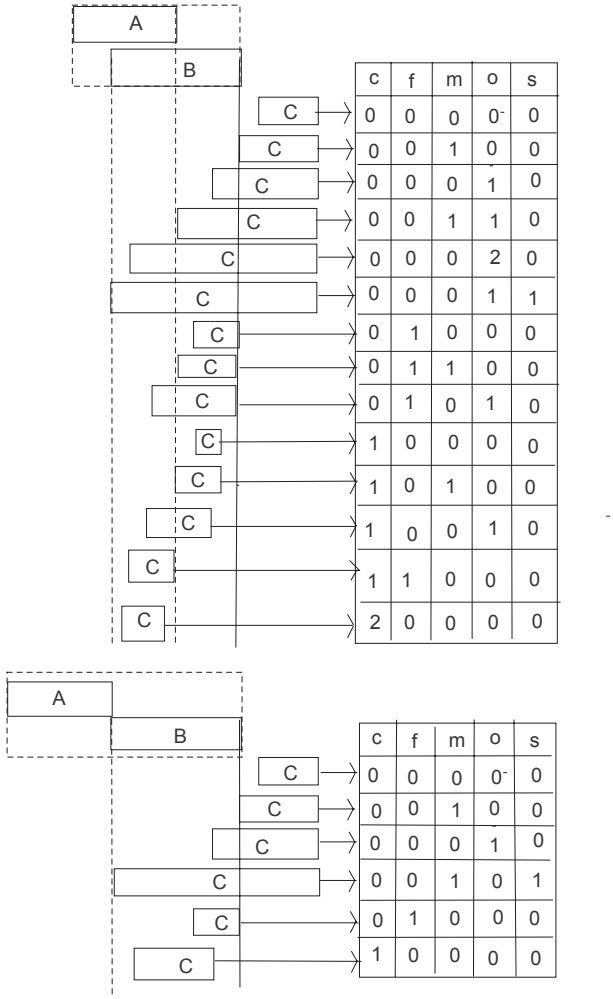

Figure 3: Partial enumeration of the possible cases involving 3 events

Based on the above, we know that there are exactly $c+f+o+s$ events whose end times come after the start time of E.

Similarly, the end time of event E is constrained as follows:

1. $E.end < E'.end$ for all E' where E' is an event in E^k and E' Contain E.
2. $E.end = E'.end$ for all E' where E' is an event in E^k and E' $Finished\_By$ E.

In other words, there are exactly c events whose end times come after the end time of E. With the linear ordering of E^k, we can determine the position of the start and end points of the new event E to obtain the linear ordering of E^{k+1}. Hence, we have shown that the augmented representation is lossless. □

5. INTERVAL-BASED EVENT MINING

In this section, we present the proposed algorithm IEMiner to discover frequent temporal patterns from interval-based events. IEMiner follows a level-by-level generation of interval-based temporal patterns. Unlike existing Apriori-based algorithms that generates a level-k pattern from two level-(k-1) frequent patterns, IEMiner employs a careful design of the candidate generation process to significantly reduce number of candidates generated. It also has an efficient support counting procedure that counts the support of all candidate patterns in a single scan of the event lists at each level.

5.1 Candidate Generation

A quick observation indicates that generating level-k candidates from two $k-1$ frequent temporal patterns will lead to a large number of invalid patterns being generated. Based on this observation, we introduce the concept of a dominant event in a pattern P. An event is said to be a dominant event in the pattern P if it occurs in P and has the latest end time among all the events in P. During the candidate generation process, a (k-1)-pattern TP_{k-1} is joined to a frequent 2-pattern TP_2 if the dominant event in TP_{k-1} is also the first event in TP_2.

We illustrate candidate generation process with the help of frequent patterns given in Table 2. We obtain these frequent 3-patterns from the set of event lists given in Figure 1. Our support value is 2. In Table 2, the dominant event of the frequent 3-pattern is underlined and the first event in the frequent 2-pattern is bold. To generate the candidate 4-patterns, we join the 3-patterns with the 2-patterns if they share the dominant event. For example, joining the third pattern in the first column of Table 2 with the first pattern in the second column will generate the candidate pattern ((A Overlap[0,0,0,1,0] B) Overlap[0,0,0,2,0] C) Contain[1,0,0,0,0] D.

One key point to note is that maintaining the set of frequent 2-patterns up-to-date is crucial to reducing the number of candidates generated.

THEOREM 2. *A $(k+1)$-pattern is a candidate pattern if it is generated from a frequent k-pattern and a 2-pattern where the 2-pattern occurs in at least $k-1$ frequent k-patterns.*

PROOF. Let TP_{k+1} be a $(k+1)$-pattern and a 2-pattern consisting of events E_i and E_j where E_i and E_j are the i^{th} and j^{th} events in TP_{k+1} respectively. By the downward closure property, TP_{k+1} will have $k+1$ frequent k-patterns. Among them, we have at least k frequent k-patterns that contain E_i. Similarly, we have at least k frequent k-patterns that contain E_j. This implies that both E_i and E_j must occur in at least $k-1$ frequent k-patterns. □

Based on the above theorem, during the $(k+1)^{th}$ iteration, when we generate the list of 2-patterns from the set of frequent k-patterns, we also maintain a count for each entry in the list to indicate the number of frequent patterns that contains the entry. Entries with count less than $k-1$ are removed from the 2-pattern list as they cannot contribute to a candidate $(k+1)$-pattern.

To illustrate working of above theorem, consider a case where 2-pattern is present in less than $k-1$ frequent k-patten and utilized for candidate generation processs. 2-pattern "F Before[0,0,0,0,0] G" is only present in second pattern from frequent 3-pattern column in Table 2. If we utilize this 2-pattern to expand the frequent 3-pattern "(A Before[0,0,0,0,0] D) Before[0,0,0,0,0] F", we generate candidate pattern "((A Before[0,0,0,0,0] D) Before[0,0,0,0,0] F) Before[0,0,0,0,0] G". According to downward closure property all of it's subpattern must be frequent. But, it's one subpattern "(D Before[0,0,0,0,0] F) Before[0,0,0,0,0] G" is not frequent. Hence even if we utilize such 2-pattern which are present in less than k-1, frequent k-pattern, we are not generating any valid candidate temporal patterns. Next we prove that our candidate generation is complete.

THEOREM 3. *The set of candidates generated by IEMiner is complete.*

PROOF. Initially, IEMiner generates all the 2-patterns. In the subsequent iterations, IEMiner generates a $(k+1)$-pattern from a frequent k-pattern and a 2-pattern. Now, assume that IEMiner generates the complete set of k-patterns. We prove that IEMiner will generate the complete set of $(k+1)$-patterns.

Suppose IEMiner does not generate the complete set of $(k+1)$-patterns, in other words, there exists a frequent TP_{k+1} that has not been generated by IEMiner. Without loss of generality, suppose TP_{k+1} can be generated from a frequent k-pattern TP_k and a 2-pattern TP_2. TP_k would have been generated by IEMiner since it is a k-pattern. The only way in which TP_{k+1} is not generated is if TP_2 is missing from the set of 2-patterns.

Since TP_{k+1} is frequent, this implies that there are $k+1$ frequent k-patterns of TP. Among these $k+1$ frequent patterns, TP_2 (generated from the list of $k+1$ k-patterns) must be present in at least $k-1$ patterns. By theorem 2, TP_2 will be generated and used, indicating that TP_{k+1} will be generated by IEMiner. This completes the proof. □

Algorithm GetNextCandidateSet (see Figure 4) gives the details. Line 1 initializes the set of candidates to an empty set. We obtain the frequent 2 patterns from the set of frequent k patterns in Line 2. Lines 3-10 generates a $(k+1)$-pattern from a frequent k-pattern and a 2-pattern as described above.

5.2 Support Counting

After the candidate patterns have been generated, we need to count the number of occurrences of each candidate to

Table 2: Candidate 4-patterns generation for level 4

frequent 3-pattern	frequent 2-pattern
(A Overlap[0,0,0,1,0] B) Before[0,0,0,1,0] D	**C** Contain[1,0,0,0,0] D
(A Before[0,0,0,0,0] F) Before[0,0,0,0,0] G	**A** Before[0,0,0,0,0] D
(A Overlap[0,0,0,1,0] B) Overlap[0,0,0,2,0] C	**B** Before[0,0,0,0,0] D
(A Overlap[0,0,0,1,0] C) Contain[1,0,0,0,0] D	**A** Before[0,0,0,0,0] F
(A Before[0,0,0,0,0] D) Before[0,0,0,0,0] F	**A** Overlap[0,0,0,1,0] C
(A Overlap[0,0,0,1,0] B) Before[0,0,0,0,0] F	**B** Overlap[0,0,0,1,0] C
(B Overlap[0,0,0,1,0] C) Contain[1,0,0,0,0] D	**D** Before[0,0,0,0,0] F
(B Before[0,0,0,0,0] D) Before[0,0,0,0,0] F	**A** Overlap[0,0,0,1,0] B
	B Before[0,0,0,0,0] F

Table 3: Support counting for event list EL_1 in Figure 1

nextEvent	Generated pattern	*passive_TP*	*active_TP*
A	-	-	A
B	(A Overlap[0,0,0,1,0] B)	-	A, B (A Overlap[0,0,0,1,0] B)
C	**(A Overlap[0,0,0,1,0] B) Overlap[0,0,0,2,0] C** *A Overlap[0,0,0,1,0] C* *B Overlap[0,0,0,1,0] C*		A, B, C
D	*C Contain[1,0,0,0,0] D* *A Before[0,0,0,0,0] D* *B Before[0,0,0,0,0] D*	A, B	C, D

determine whether they are frequent or not. Traditionally, support counting is done by scanning the event list for each candidate pattern. However, checking the occurrence of a k-pattern in a given event list with m events takes O(km) time. Repeating this process for n k-patterns takes O(kmn) time. In other words, an event in the event list will end up being scanned multiple times.

Instead, we utilize a single-pass support counting procedure where each event in the event list is scanned only once to determine the occurrence of all k-patterns. Algorithm CountSupport (see Figure 5) gives the details. The inputs are an event list EL, a set of candidate event list C, and the level number L. The idea is to keep track of the active events as we scan the event list. An event is considered active at time point t if the start time of the event is less than t while the end time of the event is greater than t. Otherwise, the event is passive. Active events are maintained in *active_TP* list and passive events are maintained in *passive_TP* list. As a new event E arrives, we update the *active_TP* and the *passive_TP* to reflect the completion of previously active events (Lines 6-7). Next, new composite events are formed between events from the *active_TP* and the new event E (Lines 9-10). If the composite event is a candidate pattern, its support count is incremented (Lines 11-12). If it is the prefix of any candidate pattern, we store it in the *active_TP* (Lines 13-15). Similarly, new composite events are formed between events from the *passive_TP* and the new event E (Lines 18-26). Event E is then inserted into *active_TP* (Line 27). With this, we only need to scan the event list once and count the support of all candidate patterns.

To illustrate the support counting process, let us consider the two candidate patterns (A Overlap[0,0,0,1,0] B) Overlap[0,0,0,2,0] C and (A Before[0,0,0,0,0] C) Overlap [0,0,0,1,0] D and the event list EL_1 in Figure 1. Table 3 shows the patterns generated as we process an active event. The patterns in italic are discarded since they are not the prefix of any candidate patterns. The pattern (A Overlap[0,0,0,1,0] B) Overlap[0,0,0,2,0] C is the only valid candidate pattern and we increase its support count.

Algorithm GetNextCandidateSet

Input: Frequent k pattern set f_k
Output: Candidate $k+1$ pattern set

```
1: CandidateSet ← φ
2: obtain frequent 2 patterns f_2 from f_k
3: for all (p ∈ f_k) do
4:    for all (q ∈ f_2) do
5:       if (first event of q = dominant event of p) then
6:          tmp_can ← join p and q
7:          CandidateSet ← CandidateSet ⋃ tmp_can
8:       end if
9:    end for
10: end for
11: return CandidateSet
```

Figure 4: Algorithm to generate candidate set

5.3 Algorithm IEMiner

We now present Algorithm IEMiner (see Figure 6). We first scan the database to obtain all the single frequent events (Line 1). These events are put in a $frequentSet$ (Line 2). We call the function $GetNextCandidateSet$ to obtain an ini-

Algorithm CountSupport

```
Input: Level L,
       Event List EL,
       CandidateSet C
 1: while (( nextEvent ← getNextevent(EL)) ≠ NULL ) do
 2:   if (nextEvent is a frequent event) then
 3:     currentTime = nextEvent.startTime
 4:     for all (tp ∈ active_TP) do
 5:       if (tp.endTime < currentTime) then
 6:         passive_TP ← passive_TP ∪ tp
 7:         active_TP ← active_TP − tp
 8:       else
 9:         relation ← getRelation(tp, nextEvent)
10:         newTP = prepareNewTP(tp,nextEvent,relation)
11:         if (newTP.size = L && newTP ∈ C) then
12:           Update count for newTP
13:         else if (newTP is a prefix of a pattern in C)
            then
14:           active_TP ← active_TP ∪ newTP
15:         end if
16:       end if
17:     end for
18:     for all (tp ∈ passive_TP) do
19:       Set relation ← Before
20:       newTP = prepareNewTP(tp,nextEvent,relation)
21:       if (newTP.size = L && newTP ∈ C) then
22:         Update count for newTP
23:       else if (newTP is a prefix of a pattern in C)
          then
24:         active_TP ← active_TP ∪ newTP
25:       end if
26:     end for
27:     active_TP ← active_TP ∪ nextEvent
28:   end if
29: end while
```

Figure 5: Algorithm to count support of candidate patterns

Algorithm IEMiner

```
 1: scan database and obtain all single frequent events
 2: frequentSet ← all single frequent events
 3: candidateSet ← GetNextCandidateSet(frequentSet)
 4: Level ← 2
 5: repeat
 6:   for all (EL ∈ EventListSet) do
 7:     countSupport(Level,EL,candidateSet)
 8:   end for
 9:   frequentSet ← Obtain frequent patterns
10:   candidateSet ← GetNextCandidateSet(frequentSet)
11:   Level ← Level + 1
12: until (candidateSet = ∅)
```

Figure 6: Algorithm IEMIner

tial *candidateSet* at level 2 (Lines 3-4). Our objective is to identify frequent temporal patterns from the *candidateSet*. The *CountSupport* procedure is called for each EL in *EventListSet* to determine the support count for each temporal patterns in *candidateSet* (Lines 6-8). Once all the EL in *EventListSet* are finished, we obtain the frequent patterns (Line 9). The *GetNextCandidateSet* function returns the candidate temporal patterns for the next level $(k+1)$ (Lines 10-11). Algorithm IEMiner terminates when *candidateSet* is empty (Line 12).

5.4 Optimization Strategies

Besides the novel candidate generation and support counting procedures, we further introduce two optimization strategies to achieve greater efficiency for IEMiner.

The first strategy involves building a list of *blacklisted* windows. A window W is *blacklisted* if it has less than k frequent events in the window as such window does not have enough events to generate a k-pattern and hence it cannot affect the support counts of the k-pattern candidates. This implies we can safely omit W from scanning during the support counting procedure from the k^{th} iteration onwards.

The second optimization strategy aims to further reduce the number of candidate patterns generated by utilizing the following theorem.

THEOREM 4 (PREFIX COUNT). *Suppose a k-pattern TP_k is generated from a frequent $(k-1)$-pattern TP_{k-1} and a 2-pattern. Let n_w denote the number of windows in which the $(k-2)$-prefix of TP_{k-1} occurs at least twice. TP_k cannot be frequent if $n_w < minsupport$.*

PROOF. We prove the theorem using proof by contradiction. Assume TP_k is frequent and $n_w < minsupport$. Let TP_{k-1} and TP'_{k-1} be two frequent $(k-1)$-patterns. We form TP''_{k-1} by taking the $(k-2)$-prefix of TP_{k-1} and merge it with the last event in TP'_{k-1}. Clearly, TP''_{k-1} is a sub-pattern of TP_k and it must be frequent. This is because if TP_k is frequent, then all sub-patterns of TP_k (in this case, TP''_{k-1}) are frequent. In other words, TP_{k-1} and TP''_{k-1} are generated in at least *minsupport* number of windows. Since TP_{k-1} and TP''_{k-1} have the same $(k-2)$-prefix, we say that the $(k-2)$ prefix of TP_{k-1} occurs in at least *minsupport* number of windows where the prefix of TP''_{k-1} is also observed. Thus $n_w >= minsupport$. This is a contradiction. Hence we have proved the theorem. □

With this theorem, we maintain the n_w values for each pattern TP_k as we scan the windows. The n_w value is incremented if there is another temporal pattern having the same $(k-1)$ prefix being generated in same window and the window has not been blacklisted. Apriori-based candidate generation process use all frequent temporal patterns discovered at the current round to generate candidates pattern for the next round. However, using the above theorem, we only need to expand those frequent patterns whose $n_w >= minsupport$.

6. IECLASSIFIER

To the best of our knowledge, this is the first work on utilizing interval-based temporal patterns for classification. Existing works such as [9, 4, 7] utilized frequent itemset patterns for classification. However, the direct adaptation of existing approaches for the interval-based patterns is not

Algorithm Best_Conf

Input: Event sequence I
1: best_class ← Default class label
2: conf ← 0
3: sup ← 0
4: **for all** (temporal pattern $TP \in PatternMatch_I$) **do**
5: **if** (conf(TP) > conf) **then**
6: best_class = class label of TP
7: conf = conf(TP)
8: sup = sup(TP)
9: **else if** (conf(TP) == conf and sup(TP) > sup) **then**
10: best_class = class label of TP
11: conf = conf(TP)
12: sup = sup(TP)
13: **end if**
14: **end for**
15: Assign best_class as a class label to I

Figure 7: Algorithm to assign class label based on best confidence

Algorithm Majority_Class

Input: Event sequence I
1: **for all** (class C_i, $1 \leq i \leq c$) **do**
2: $count[C_i] \leftarrow 0$
3: **end for**
4: **for all** ($TP \in PatternMatch_I$) **do**
5: clabel ← class label of TP
6: $count[clabel]$++
7: **end for**
8: $max \leftarrow 1$
9: **for all** class C_i, $1 \leq i \leq c$ **do**
10: **if** ($count[C_i] > count[max]$) **then**
11: $max \leftarrow i$
12: **end if**
13: **end for**
14: Assign class C_{max} to I

Figure 8: Algorithm to assign class label based on majority vote

straightforward and scalable due to the large number of frequent temporal patterns generated. On the other hand, transforming the temporal patterns by treating each temporal relation between any two events as an independent attribute will result in a very high dimensional space and may suffer from the curse of dimensionality. To address these problems, we propose a classifier called IE-Classifier.

The building of IEClassifier has two aspects. The first aspect deals with the selection of a subset of patterns that is able to discriminate one class from another with high degree of accuracy. The second aspect deals with the assignment of an unknown input event sequence to a class given the selected subset of patterns.

Frequent interval-based temporal patterns are generated from a set of training data that has been partitioned according to their class labels C_i, $1 \leq i \leq c$, where c is the number of class labels. A frequent pattern which occurs in only one class is more discriminating than one that occurs in all the classes. To identifying such discriminating patterns, we compute the information gain of each pattern TP using the following equation:

$$\begin{aligned} \text{G(TP)}= \quad & -\sum_{i=1}^{c} p(C_i) log\ p(C_i) + \\ & p(TP) \sum_{i=1}^{c} p(C_i|TP) log\ p(C_i|TP) + \\ & p(\overline{TP}) \sum_{i=1}^{c} p(C_i|\overline{TP}) log\ p(C_i|\overline{TP}) \end{aligned}$$

In the above formula, $p(TP)$ is probability of pattern TP to occur in datasets. Also $p(\overline{TP})$=1-$p(TP)$. We calculate information gain for all frequent patterns using above formula. Those temporal patterns whose information gain values are below a predefined $info\_gain$ threshold are removed. The remaining temporal patterns are the discriminating patterns. We assign to each discriminating pattern the class label with the highest conditional probability $p(C|TP)$. $p(C|TP)$ is also known as the confidence of TP (conf(TP)). At the end of the process, each discriminating pattern TP is assigned a class label (clabel(TP)) with the support count sup(TP).

For an unknown input event list I, we match I against all the discriminating patterns. Let $PatternMatch_I$ be the set of discriminating patterns that are contained in I. Intuitively, there are two ways to assign a class label to I. The first way is to assign I to the class label with the highest confidence pattern in $PatternMatch_I$. The second way is to assign I to the majority class labels of the patterns in $PatternMatch_I$. Algorithms Best_Conf (see Figure 7) and Majority_Class (see Figure 8) show the details.

7. EMPIRICAL STUDIES

In this section, we present the results of experiments conducted to evaluate IEMiner and IEClassifier.

We first compare the performance of IEMiner with state-of-the-art algorithms GenPrefixSpan [3], TPrefixspan [15] and H-DFS [11] to evaluate its efficiency and scalability. We use GenPrefixSpan as the baseline for IEMiner since it only finds the Before relationship while IEMiner is able to generate all the temporal relationships among the events. Then we examine the effectiveness of the two optimization strategies proposed in Section 5. We also apply IEMiner on two real world datasets, namely the American Sign Language (ASL) dataset[3] and the Hepatitis dataset[4]. Finally, we verify the accuracy of IEClassifier on the Hepatitis data set.

All the algorithms are implemented in C#. The experiments are performed on a 1.6 GHz centrino duo with 1.5GHz RAM running window operating system. We modify the IBM data quest generator[5] by including an additional parameter "$EvtDen$"(i.e., number of events active at a time) to generate the synthetic data sets. The control parameters used in the data generator are:

1. number of windows (i.e., D)
2. number of event types (i.e., T)
3. average number of events, active at a time (i.e., $EvtDen$)
4. average length of patterns (i.e., L)

[3]http://www.bu.edu/asllrp/
[4]http://ecmlpkdd.isti.cnr.it/
[5]http://www.almaden.ibm.com/software/quest/Resources/index.shtml

5. probability of similar event appear in same window (i.e., P)

We keep $T = 500$ for all the experiments. The notation "$Data\_D\_T\_L\_P\_EvtDen$" represents dataset generated using D, T, L, P and $EvtDen$ control parameters.

7.1 Experiments on Synthetic Datasets

First, we analyze the effect of varying minimum support on runtime. Figure 9 shows the results when minimum support varies from 2% to 12%. We observe that as support value decreases, the time required by all the algorithms increases. However, the runtime for H-DFS and TPrefixSpan increase drastically compared to IEMiner. We also note that IEMiner has a comparable runtime as GenPrefixSpan even though GenPrefixSpan only finds the Before relationship while IEMiner generates all types of interval-based relationships.

Next, we examine the effect of varying sizes of D on runtime. We select 4% as a support value and vary D from 100K windows to 400K windows. Average number of events in each window is 15, hence average number of events vary from 1500K to 6000K. Figure 10 shows the experimental results. The runtime of IEMiner increases linearly as value of D increases while the runtime of TPrefixSpan increases exponentially.

We also investigate the effect of varying L on run time. We keep D and the support value constant. Figure 11 shows the results. As the value of L increases, the runtime of IEMiner increases but at a slower rate compared to H-DFS and TPrefixSpan. This demonstrates that IEMiner is effective in reducing the number of candidates generated, thereby allowing a much longer pattern to be discovered.

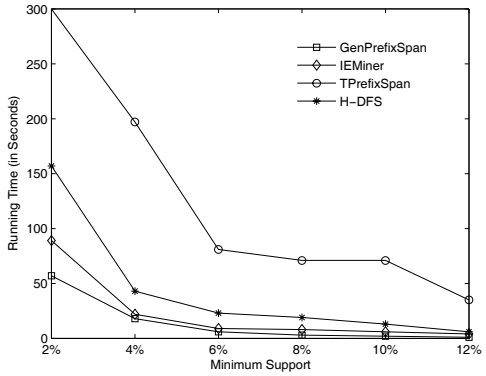

Figure 9: Effect of Varying Minimum Support

In the next set of experiments, we investigate the effect of varying $EvtDen$ on run time. Figure 12 shows the runtime of IEMiner for varying values of $EvtDen$. We observe that as $EvtDen$ increases, the number of temporal relations among the events also increases. Hence, the support count for each pattern is reduced. As a result, fewer number of frequent patterns are generated compared to GenPrefixSpan. Note that $EvtDen = 1$ means that there is only one active event at each time.

Finally, we analyze the effectiveness of the two optimization strategies. Two variations of IEMiner are implemented.

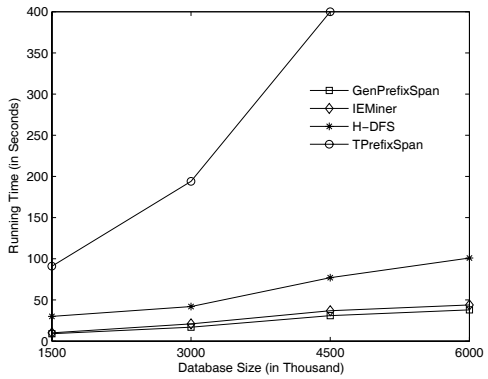

Figure 10: Effect of Varying Database Size (Data_?k_500_15_0.3_2)

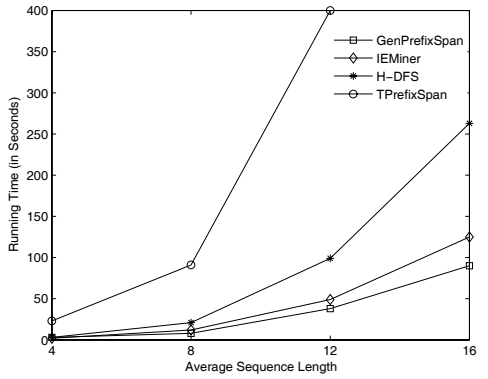

Figure 11: Effect of Varying Pattern Length (Data_200k_500_?_0.3_2)

IEMiner-1 uses only the window blacklisting strategy, while IEMiner-2 uses only the prefix count strategy (Theorem 4). Figure 13 shows the results. We see that the window blacklisting strategy (IEMiner-1) is able to improve the performance of IEMiner more as compared to the prefix count strategy (IEMiner-2).

7.2 Experiments on Real World Datasets

In this section, we apply the four mining algorithms (IEMiner, TPrefixSpan, H-DFS and GenPrefixSpan) on two real world datasets, namely, the American Sign Language (ASL) dataset and the Hepatitis dataset.

We use the ASL dataset to investigate the relationship between grammatical structure and gesture field. This dataset has 730 utterances. Each utterance contains recurrent ASL gestural and grammatical field. We obtain the frequent temporal patterns at various support values. The set of mined patterns is verified against the ground truth [11]. The results are shown in Figure 14.

The Hepatitis dataset contains a total of 771 patient records over a period of 10 years. In this dataset, a patient either has Hepatitis B or Hepatitis C. There are about 230 tests that a patient may undergo, out of which 25 tests are conducted

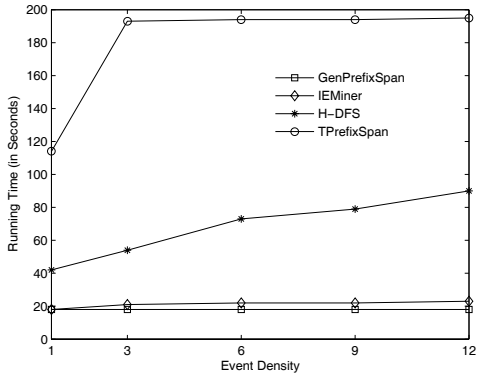

Figure 12: Effect of Varying Event Density (minimum support = 4%)

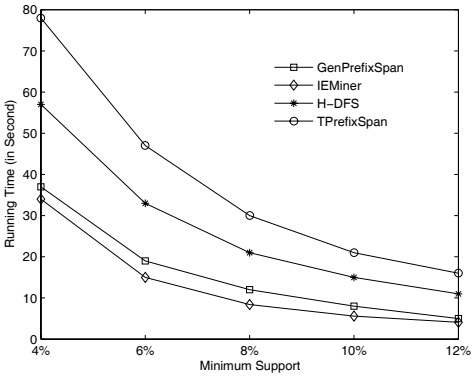

Figure 14: Experiments on ASL dataset

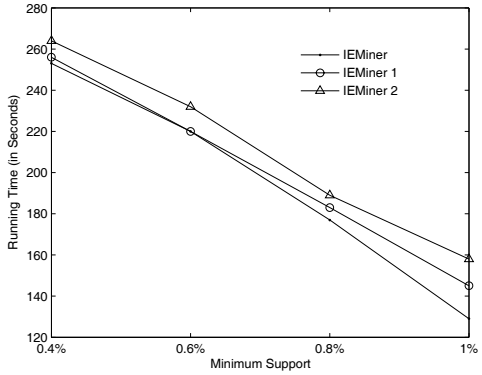

Figure 13: Effect of Optimization Techniques (Data_200k_500_20_0.3_2)

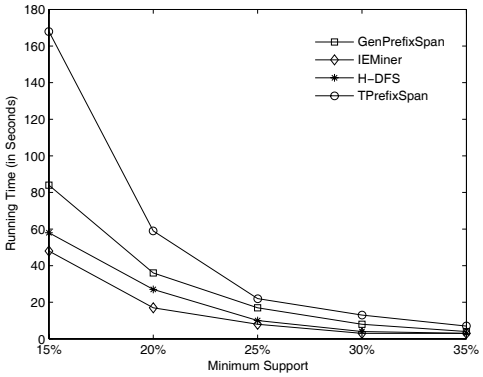

Figure 15: Experiments on Hepatitis dataset

regularly at each visit to the hospital. We transform the test results over time into interval based events as follows:

1. If the results of a test $Test$ during an interval $[start, end]$ consistently falls within the normal range of values for the test $Test$, that is, N(ormal), we map it to the event ($Test$-N, $start, end$).
2. If the results of a test $Test$ during an interval $[start, end]$ consistently falls below the normal range of values for the test $Test$, that is, L(ow), we map it to the event ($Test$-L, $start, end$).
3. If the results of a test $Test$ during an interval $[start, end]$ consistently falls above the normal range of values for the test $Test$, that is, H(igh), we map it to the event ($Test$-H, $start, end$).
4. If the results of a test $Test$ during an interval $[start, end]$ oscillates between Low and Normal, we map it to the event ($Test$-NL, $start, end$).
5. If the results of a test $Test$ during an interval $[start, end]$ oscillates between Normal and High, we map it to the event ($Test$-NH, $start, end$).
6. If the results of a test $Test$ during an interval $[start, end]$ oscillates between Low and High, we map it to the event ($Test$-LH, $start, end$).

After mapping the test results into interval based events, we create an event list for each patient. We obtain a total of 498 event lists that correspond to patients who undergo the 25 tests regularly.

Figure 15 shows the results of applying the mining algorithms on the transformed Hepatitis dataset (Hep-T). We observe that IEMiner perform best compared to all algorithms. Here, average length of underlying event list is around 200 events. GenPrefixspan did not perform well because it consider events without duration and as a result many patterns are generated compared to other three algorithm.

7.3 Accuracy of IEClassifier

Finally, we investigate whether the discovered temporal patterns will improve the accuracy of classification. We compare the accuracy of IEClassifier with standard classifiers such as C4.5, CBA and SVM which do not use the temporal information.

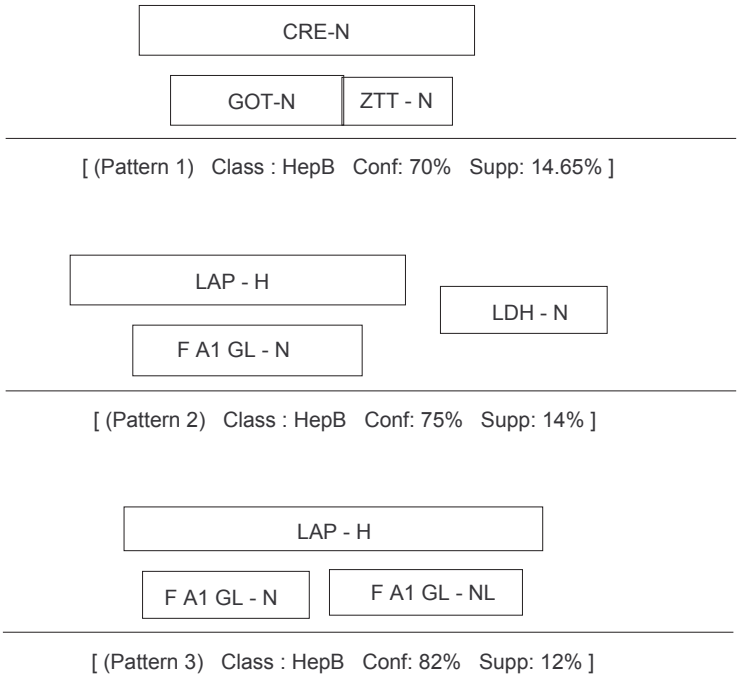

Figure 16: Sample of temporal patterns for Hepatitis B disease

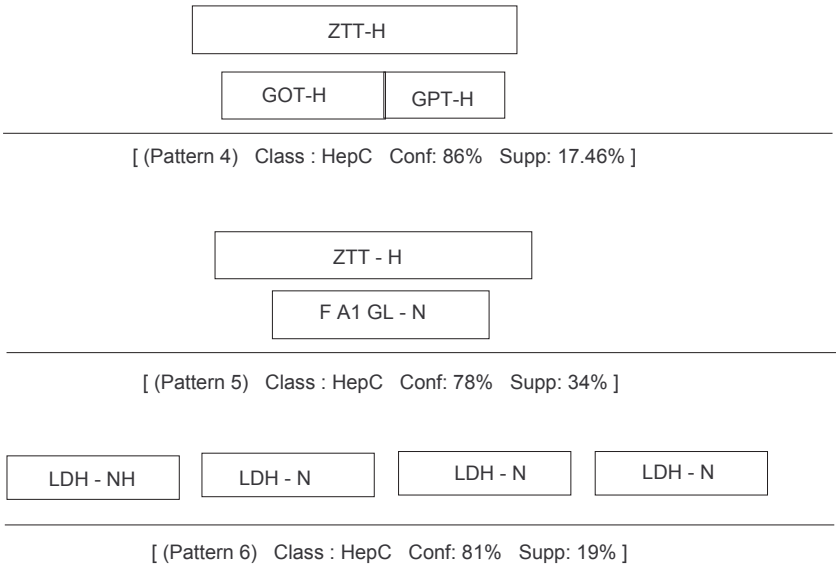

Figure 17: Sample of temporal patterns for Hepatitis C disease

We apply C4.5[6], CBA[7] and SVM[8] classification tools on the original Hepatitis dataset where each test per visit is considered as an attribute. In total, we have around 10,000 attributes.

Next, we build the IEClassifier from the interval-based Hep-T dataset obtained in the previous section. We label an event list in the Hep-T dataset as HepB or HepC to indicate that the patient corresponding to the event list has Hepatitis B or Hepatitis C. in total, we have 203 event lists labeled as HepB and 295 event lists labeled as HepC. The info-gain threshold is set at 0.02 with minsup of 10%.

10-fold cross validation testing strategy is adopted. Table 4 shows the results.We observe that the classifiers that make use of temporal relationships can indeed improve the prediction accuracy. Overall, the Majority_Class voting strategy achieves the best accuracy.

Table 4: Testing accuracy

Classifier	Testing Accuracy
C4.5	78.13%
CBA	76.49%
SVM	78.72%
IEMiner (Majority_Class)	**82.13**%
IEMiner (Best_Conf)	78.91%

Figure 16 and Figure 17 shows a sample of the temporal patterns that is able to discriminate between the HepB and HepC classes. The first 5 patterns reveal the temporal relations between different tests in the Hepatitis B and Hepatitis C patients. For example, pattern 3 describes the behavior of F-A1.GL with respect to LAP test. We discover that during the period in which LAP's value is in the normal range, the F-A1.GL test starts in the normal range and then begins to oscillate between the low and normal range. This pattern is observed in Hepatitis B patients with 82% confidence. It is present in 25.54% hepatitis B patient data and 3% of hepatitis C patient data. Pattern 6 reveals how a particular test, the LDH test, evolves in the Hepatitis C patients. Initially, the LDH's value ranges between normal and high, and as time passes, it's value becomes normal. This pattern is present in 26% of hepatitis C patient as opposed to 8% in hepatitis B patients.

8. CONCLUSION

In this paper, we have examined the problem of mining relationships among interval-based events. We augmented existing hierarchical representation with additional count information to make the representation lossless. Based on this new representation, we have developed an Apriori-based IEMiner algorithm to mine frequent temporal patterns from interval-based events. We designed an efficient support counting procedure. The performance of IEMiner is further improved by employing a window blacklisting strategy and a prefix counting strategy. Experiments on synthetic data sets and real world datasets demonstrate the efficiency and scalability of our proposed approach. Beyond this, we have designed the first interval-based classifier, IEClassifier to improve the predictive accuracy of closely related classes. Experiment results on the Hepatitis dataset show that IEClassifier outperforms traditional classifiers such as C4.5, CBA, and SVM.

[6]http://www.cs.waikato.ac.nz/ml/weka/
[7]http://www.comp.nus.edu.sg/ dm2/
[8]http://svmlight.joachims.org/ and also weka

9. REPEATABILITY ASSESSMENT RESULT

All the results in this paper were verified by the SIGMOD repeatability committee.

10. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. *IEEE ICDE*, 1995.

[2] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM, 26(11)*, 1983.

[3] C. Antunes and A. L. Oliveira. Generalization of pattern-growth methods for sequential pattern mining with gap constraints. *Machine Learning and Data Mining in Pattern Recognition*, 2003.

[4] H. Cheng, X. Yan, J. Han, and C.-W. Hsu. Discriminative frequent pattern analysis for effective classification. *IEEE ICDE*, 2007.

[5] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[6] A. Hakeem, Y. Sheikh, and M. Shah. A hierarchical event representation for the analysis of videos. *AAAI*, 2004.

[7] T.B. Ho, T.D. Nguyen, S. Kawasaki, S.Q. Le, D.D. Nguyen, H. Yokoi, and K. Takabayashi. Mining hepatitis data with temporal abstraction. *SIGKDD*, 2003.

[8] P.S. Kam and A.W.C. Fu. Discovering temporal patterns for interval-based events. *DaWaK*, 2000.

[9] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. *SIGKDD*, 1998.

[10] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. *SIGKDD*, 1995.

[11] P. Papapetrou, G. Kollios, S. Sclaroff, and D. Gunopulos. Discovering frequent arrangements of temporal intervals. *IEEE ICDM*, 2005.

[12] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: mining sequential patterns e efficiently by prefix-projected pattern growth. *IEEE ICDE*, 2001.

[13] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., 1993.

[14] T. Zhao R. Nevatia and S. Hongeng. Hierarchical language-based representation of events in video streams. *IEEE Workshop on Event Mining*, 2003.

[15] S. Wu and Y. Chen. Mining nonambiguous temporal patterns for interval-based events. *IEEE TKDE, 19(6)*, 2007.