Sequential Pattern Mining

Ali Khudiyev, Nurlan Imanov, Hafiz Gahramanov

April 2021

1 Introduction

Data mining is a process of discovering knowledge from a database. Our goal in data mining tasks is to extract interesting rules from given data-set. Sequential pattern mining is one of the field of data mining, that we are given a database of sequences where each event or element is ordered and our task is to discover all sequential patterns with user-specified minimum support value. There are several algorithms that developed to do this mission in literature such as : PrefixSpan [1] , GSP [2] , SPADE [3] and etc. In our project we used GSP [2] algorithm to find sequential patterns. We built this algorithm from scratch by using C++ language and tested results with SPMF software.

GSP(Generalized Sequential Pattern) [2] is Apriori-based approach which uses level-wise manner to find frequent sequential patterns in the data. It follows below pseudo-code in its procedure :

```
F1 = the set of frequent 1-sequence
k=2,
do while Fk-1 != Null;
    Generate candidate sets Ck (set of candidate k-sequences);
    For all input sequences s in the database D
    do
        Increment count of all a in Ck if s supports a
    End do
    Fk = {a ∈ Ck such that its frequency exceeds the user-specified minimum support value}
    k = k+1;
End do
Result = Set of all frequent sequences is the union of all Fk's
```

Figure 1: Pseudo-code of GSP

Pruning process in GSP algorithm is based on Apriori principle, where we are able to conclude that if the k-item set is not frequent then any super-set of this item set will not be frequent as well in further levels. Since GSP uses Apriori principle in its pruning process as a result it reduces search space. Nevertheless, this algorithm is expensive in terms of time consuming because it scans the given database multiple time while finding number of occurrence of each k-item set and it generates huge set of candidate sequences.

PrefixSpan(Prefix-Projected Sequential Pattern Growth) [1] is depth-first search algorithm and relies on definition of projection. Unlike GSP algorithm there is no candidate sequence generation in PrefixSpan that leads to consume less time. However,it constructs projected databases in its procedure which causes to use enormous memory. In order to avoid such a drawback, a lot of methods are developed such as: bi-level projection, pseudo-projection, partition projection and etc. We can see the pseudo-code of this algorithm from here:

Algorithm 1 (*PrefixSpan*) Prefix-projected sequential pattern mining.

Input: A sequence database S, and the minimum support threshold min_support.

Output: The complete set of sequential patterns

Method: Call $PrefixSpan(\langle\rangle, 0, S)$.

Subroutine $PrefixSpan(\alpha, l, S|_\alpha)$
The parameters are 1) α is a sequential pattern; 2) l is the length of α; and 3) $S|_\alpha$ is the α-projected database if $\alpha \neq \langle\rangle$, otherwise, it is the sequence database S.

Method:
1. Scan $S|_\alpha$ once, find each frequent item, b, such that
   (a) b can be assembled to the last element of α to form a sequential pattern; or
   (b) $\langle b\rangle$ can be appended to α to form a sequential pattern.
2. For each frequent item b, append it to α to form a sequential pattern α', and output α'.
3. For each α', construct α'-projected database $S|_{\alpha'}$, and call $PrefixSpan(\alpha', l+1, S|_{\alpha'})$.

Figure 2: Pseudo-code of PrefixSpan

2 Problem

The problem in sequential pattern mining is finding the all frequent sequential k-item sets that their support value is greater or equal than user-specific threshold value. However, we can get more information after finding these frequent sequential patterns. Since we have the time dimension in our hands from the data, actually we can get the possible minimum and maximum time difference(time interval) between each event of the frequent pattern in order to provide more information. Apart from that, even we can predict the timestamp of the next event from previous event by using machine learning techniques. In this work, we tried to handle all these concepts.

3 Development

To store the given data in an efficient manner we have used several Standard Template Library(STL) containers such as *vector*, *pair*, *map* as well as our custom data structures. As the result, we have created a simple and problem-specific database structure(*DB*) which contains all the sequences, their IDs, events, timestamps and etc. The *DB* also contains user-friendly functions for text-to-database initialization, database-to-text transition, sequence printing on console and mining the sequential patterns. We have been able to develop such a small and very simple API (see table 2) that allows us to work on the project very easily. We have also created several data structures within this API to hold the events, patterns and sequences.

Data structure	**Attributes**	**Description**
Pattern	timestamp events intervals	Holds events that have share the same timestamp
Sequence	SID support patterns	Holds patterns that may contain one or more events
DB	sequences	Loads and transforms the given input file as the array of sequences

Table 1: Custom data structures

DB Function	Description
void init(const string& filepath)	Initializes the database from the given file. **Note that, the input file has to follow the specified format described in Input file.**
void save(const string& filepath)	Stores the database in the specified file. **Note that, the output file follows the specified format described in Output file.**
size_t size()	Returns the number of sequences in the database.
void print(size_t n, ostream& out)	Prints first n sequences from the database to *out*(i.e. *cout* for console output).
vector<Sequence> extract(size_t support, size_t k, MiningMethod algorithm)	Mines sequential patterns from the database with the given minimum *support*, maximum itemset length(k) by using the specified *algorithm*. **Note that, there are only 1 available algorithm for now: GSP.**

Table 2: API (Database)

As the second part of the project required to predict the timestamps of frequent events, we have developed two approaches for this purpose: *global* and *local interval search*. *Global interval search* is very naive way of predicting the time of upcoming events since it requires all of the corresponding time differences to be considered. Once we collect the time differences between two frequent events we can obtain a couple of facts(see table 4):

- Event e_2 happens at least $t_{\min}$ units after e_1
- Event e_2 happens at most $t_{\max}$ units after e_1

By having these facts, we can naively deduce that e_2 usually comes after units after e_1. However, this might be very imprecise prediction depending on the dataset, and we could measure its preciseness by measuring standard deviation on the database.

Local interval search is a little bit different than the first approach; it only considers the time differences between two successive events which are also frequent(see table 4). In this approach, we have also considered the timestamps of e_1's as given inputs as well. So, one should provide a frequent event and its timestamp to obtain timestamp of the upcoming frequent event. For our case, we have mined the frequent itemsets which contain the same event(-1) and this eases our job as well since we do not need to deal with different type of events.

(1,A)	(2,A)	(2,B)	(4,A)	(10,A)
(3,A)	(4,A)	(5,B)	(8,A)	(9,A)
(4,C)	(8,A)	(9,B)	(10,A)	(12,A)

Table 3: Example sequential data

Sequential pattern	Time interval with global search	Time interval with local search	Timestamp (for local search)
(A)(A)	[1,9]	[1,1]	1
(A)(A)	[1,9]	[2,2]	2
(A)(A)	[1,9]	[1,1]	3
(A)(A)	[1,9]	[1,6]	4
(A)(A)	[1,9]	[1,2]	8
(A)(A)	[1,9]	-	9
(A)(A)	[1,9]	[2,2]	10
(A)(A)	[1,9]	-	12

Table 4: Example time intervals

The table above is given for an illustration purpose and it clearly shows a difference between two methods. While we lose a signification precision with the intervals obtained by the global search, it is better to predict with the local search. However, we need a timestamp for use the local search results as they may vary depending on the timestamp. For example, if we were to predict timestamp of the next frequent event(A)

occuring right after (8,A), we could easily predict that it is going to happen after at least 1 and at most 2 time units after the timestamp 8(since all successive (8,A)(t,A) couples only have 1 and 2 time units difference between them; (8,A)(9,A) and (8,A)(10,A)).

3.1 Program arguments

We have developed both CLI(Command Line Interface) and GUI(Graphical User Interface) for the project. To be able to use CLI version, there are several arguments that should be specified by the user on the terminal, however, it can be more easy to use GUI version as it appears more clear to the user to fill blanks within the program. Both versions of the program takes the arguments shown below:

1. Input file
2. Minimum support (%)
3. Maximum sequence length
4. Mining algorithm
5. Output file
6. Verbose(optional, to include [min, max] time intervals)

Note: This arguments have to be given in the above specified order if you are using the command line interface. Otherwise, program may not work properly.

3.2 File formats

3.2.1 Input file

User-specified input file has the format shown below:

```
(t11 e11),(t12 e12),...
(t21 e21),(t22 e22),...
(t31 e31),(t32 e32),...
```

Each pair consists of two numbers - timestamp and event ID. A sequence is a collection of such pairs which occupies only a single line.

Note: We also assume that the input file is sorted respective to the timestamps for each and every sequence.

3.2.2 Output file

Program-generated output can be one of the two forms:

- Sequential Frequent Patterns
- Sequential Frequent Patterns with Time Intervals

Sequential frequent patterns without any time intervals are represented with the format shown below:

```
e11 | e12 | ... | #SUP: 75
e21 e22 | e23 | ... | #SUP: 40
e31 | e32 e33 | e34 | ... | #SUP: 38
```

Note: The program separates the sequential frequent patterns with the "|" character rather than commonly used "-1". The reason behind this is the given input file for this project contained "-1"s as events, and we decided to choose different separator to avoid the confusion.

Sequential frequent patterns with time intervals are stored in the following format:

```
e11 [4 12] ... #SUP: 75
e21 e22 [1 5] ... #SUP: 40
e31 [1 5] e32 e33 [2 4] e34 [5 5] ... #SUP: 38
```

3.3 User manual

After downloading the program, you can open up the terminal from the project folder and type following commands:

```
// for command line interface
./extract [input] [support] [k] [method] [output] [verbose(opt)]

// for graphical user interface
python miner.py
```

Note: You need to have *python 3.x* with *tkinter* library installed on your computer to be able to run the GUI version.

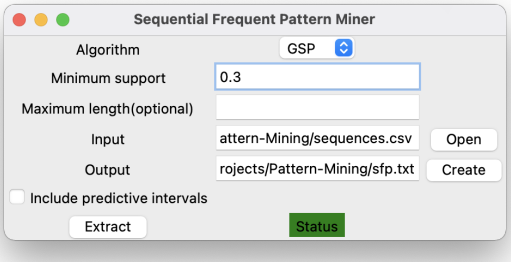


Figure 3: Graphical User Inteface

There is a **STATUS** bar to indicate the different errors or ill-formed arguments given by the user. Status codes and colors are shown in the table below:

Status	Color
Neutral	STATUS
Invalid minimum support	STATUS
Invalid I/O files	STATUS
Success	STATUS
Failure	STATUS

Table 5: Status codes and colors

4 Results

We compared our results with the ones produced by SPMF library and observed that both developed program and SPMF gave the same sequential frequent patterns for different minimum support threshold values such as 30%, 50%, 70%, 80%. However, we also noticed that while the minimum support threshold was set lower, the SPMF library produced results slower than our algorithm; the main reason behind this could be due to the speed of C++ language.

Minimum support	Developed(s)	SPMF(s)
80%	1.03	3.07
70%	0.98	3.065
50%	1.04	20.427
30%	1.1	44.597
10%	199.64	- (1000+)

Table 6: Run-time statistics

As it seems from the table 6, the developed algorithm worked much faster than the SPMF software and when the minimum support was set to 10% we stopped the software algorithm due to exceeding 1000 seconds. However,the results obtained by both the developed program and the SPMF implementation of GSP algorithm can be verified. Here are several examples shown in below figures.

Pattern	#SUP:
0 −1	10,000
0 −1 0 −1	10,000
0 −1 0 −1 0 −1	10,000
0 −1 0 −1 0 −1 0 −1	10,000
0 −1 0 −1 0 −1 0 −1 0 −1	10,000
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	10,000
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,999
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,999
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,998
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,986
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,943
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,834
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,565
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	9,059
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	8,225
0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1 0 −1	7,093

(a) By SPMF software

```
-1 | #SUP: 10000
-1 | -1 | #SUP: 10000
-1 | -1 | -1 | #SUP: 10000
-1 | -1 | -1 | -1 | #SUP: 10000
-1 | -1 | -1 | -1 | -1 | #SUP: 10000
-1 | -1 | -1 | -1 | -1 | -1 | #SUP: 10000
-1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9999
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9999
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9998
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9986
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9943
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9834
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9565
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 9059
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 8225
-1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | #SUP: 7093
```

(b) By the developed program

Figure 4: Result of GSP with minimum support set to 70%

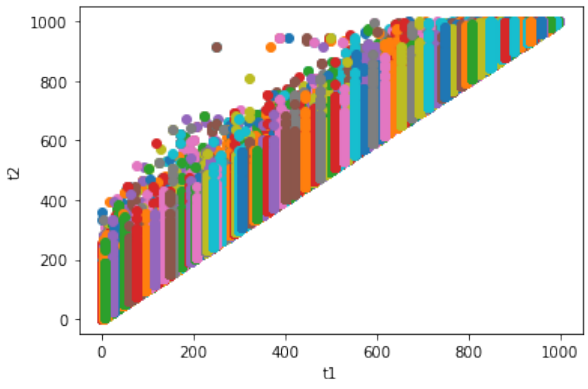
Note that, we have altered the original input file(*sequences.csv*) by adding 1 to all event IDs since it contained −1 event IDs which is considered as a separator in the SPMF documentation. Therefore, all the 0's in figure 4a correspond to -1's in the original database.

To obtain the time intervals we have implemented two different approaches. The first approach is to take the global time intervals between the couple of events after which minimum and maximum Δt is calculated. This approach leads us to obtain the following result shown below:

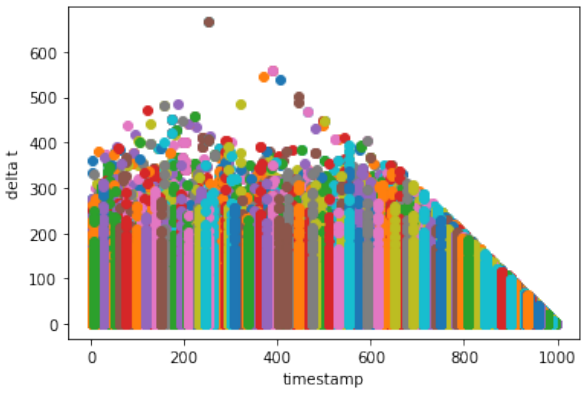
```
-1 #SUP: 10000
-1 [1,980] -1 #SUP: 10000
-1 [1,980] -1 [1,980] -1 #SUP: 10000
-1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 10000
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 10000
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 10000
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9999
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9999
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9998
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9986
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9943
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9834
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9565
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 9059
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 8225
-1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 [1,980] -1 #SUP: 7093
```

Figure 5: Result of GSP with time intervals (by the developed program)

The second approach considers the time intervals locally which means we have searched for only successive pair of events and then calculated minimum and maximum Δt. Since we have only worked with successive couples of events, the obtained result shows us the Δt values according to the given timestamps(see 6).



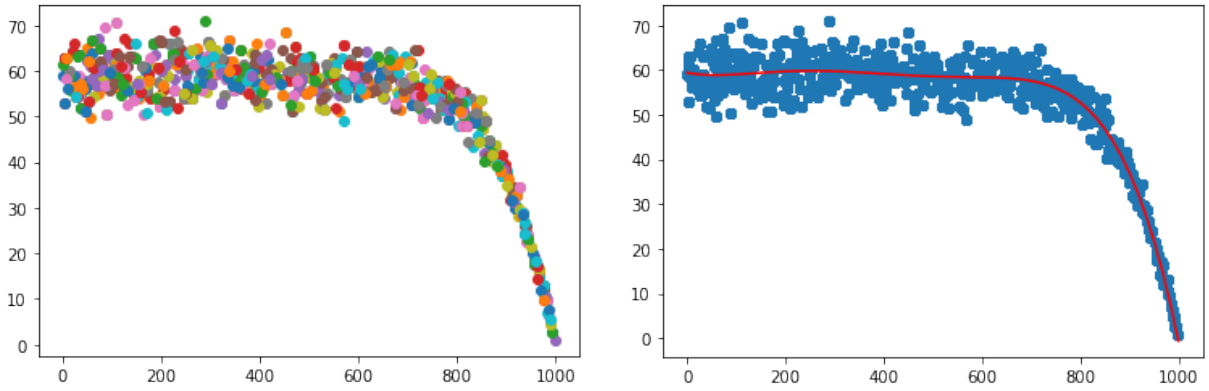
(a) (t_1, t_2) pairs



(b) $(t_1, \Delta t = t_2 - t_1)$ pairs

Figure 6: Correlation between successive frequent events (minimum support = 70%)

If we take the mean values of t_2 corresponding to each timestamp t_1, then we may get a idea about when the next frequent event is going to occur. To do so, the underlying approach not only uses the frequent event ID's but also their timestamps to predict timestamp of the next frequent event.



(a) $(t_1, (\bar{\Delta}t))$ pairs

(b) Prediction with polynomial of degree 5

Figure 7: Predicted Δt's between the time interval of [0, 1000)

We can observe from the above figure 7a that it gets even more clear to predict the Δt of a frequent event when we are dealing with larger timestamps. After fitting the polynomial of degree 5 to the raw $(t_1, \bar{\Delta}t)$ pairs, the obtained result is shown in figure 7b.

References

[1] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pintoa, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Mining Sequential Patterns by Pattern-Growth:The PrefixSpan Approach. *IEEE Transactions on knowledge and Data Engineering*, 16:215–226, 2004.

[2] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. *Springer, Heidelberg(2006)*, 1057:1–17, 1996.

[3] MOHAMMED J. ZAKI. SPADE: An Efficient Algorithm for MiningFrequent Sequences. *Machine Learning 42*, pages 31–60, 2001.