Mining data logs by finding episodes using frequent item sets

Dohn Arms (darms2)

Abstract

Data logs can be mined by using several types of sequential pattern mining algorithms. For the case I have, which is for logs generated by value changes within a controls system, I need to use episode mining, of both the parallel and serial type. The episodes I am to mine also need to allow for repetition of items. The publicly available software did not fit my exact processing needs, so I worked out methods that would let me do episode mining using frequent itemset mining algorithms.

1 Introduction

Data logs have long been mined for information. Logs can be configured in a variety of different ways or have different properties, such as having very different time granularities (subsecond time entries versus date entries), allow different events to occupy the same time stamp (so you are not sure of the order), or have multiple events lumped together with the same time stamp with some extra information (like a maximum and minimum seen for summarized events).

The data logs used for this project are from the EPICS controls system [1], used at the Advanced Photon Source at Argonne National Laboratory. EPICS uses a distributed system of computers, where a device has a controlling computer which presents the device to the distributed system as a record. The computers (or processes on a computer) are referred to as Input/Output Controllers, or IOCs. Records can be of many types (input or output, integer or floating point, enumerations, etc.) The fields of a record can be changed by its system ID, called a process variable (PV), and a change done across the network (instead of internally in the database) is done through what is called channel access.

It is possible to monitor all channel access traffic across the network, by installing special software for each IOC. Any time channel access is used to change the value of a PV, a message is sent to a central log daemon to be stored. All this traffic for a subnet is sent to the same log daemon to get the network’s activity. The time granularity is 1 second, and if PVs are changed too rapidly, this is summed up with a minimum and maximum value set to a time stamp and the PV. There is more information in these logs, but all I was concerned about was two pieces of information: the time of the event and the PV that was being altered (ignoring if multiple changes actually happened).

The logs for several subnets were filtered through a program that reduced the logs to two files. The first was an indexed list of all the PVs that were found in the event sequences. The second was the list of the event sequences of time and PV indexes, with the time value adjusted so that they were all relative to the first event. The filtering also combined all events that had the same time stamp (of different PVs) into an event itemset.
The point of this project is to look for patterns in the logs which might show a user doing something, as opposed to something that is automated. Scientists will use automated procedures as much as possible, but there are times when human intervention comes into play. The most typical pattern is from a data scan, where one or several PVs are changed with a non-periodic time signature, followed by a new PV which in turn causes a new PV to change with a periodic signature.

As regards to mining data logs, this can be done with several types of sequential pattern mining: episodes, periodic patterns, and continuities. Periodic patterns are not appropriate here as we are not interested in periodic PVs. Continuities are not appropriate here as the recurring series of events do not have a strict time signature that continuities need. As the timing of events for the PV changes is not very strict, episodes need to be considered for this project.

2 Episode Mining

Episode mining is a type of sequential pattern mining algorithm, first introduced with the WINEPI algorithm [2]. The data for these types of algorithms is a sequence of time-dependent events, where each event can contain a single item or multiple items (itemset), where an item is what is recorded; in our case, it’s a PV change. A sequence also has associated with it a start and end time. A window is used to create a set of events that are searched for sequential patterns, where the window size is typically fixed. The goal for mining is to find the patterns that meet the criteria that they appear at least a certain number of times, the minimum support; any patterns that meet this are termed frequent. The support of a pattern is typically compared to the total number of windows possible for a sequence; if the window width is \( w \), the start and end times are \( t_s \) and \( t_e \), then the number of windows possible is \( t_e - t_s - w + 1 \).

Event items are represented in this paper as capital letters, such as \( A \), \( B \), and \( C \). An itemset containing all of them is shown as \( (ABC) \). A sequence is a chain of events, and an example is \( <A(BC)D> \), where \( A \) is followed by the itemset of \( B \) and \( C \), which is in turn followed by \( D \).

For episodes in general, the time between events is not important. For parallel episodes, the order of events is also unimportant, only the fact that all events happened in the window. For serial episodes, the order of events is important. A serial sequence pattern of \( <ABC> \), where each letter is an event item, implies that \( t_A \leq t_B \leq t_C \); not only does \( <ABC> \) satisfy the pattern, so does \( <(ABC)> \). An injective episode is one that allows for an item type to occur only once.

2.1 Public Software

There is little episode mining software available out in the public. An online search for the WINEPI miner software turned up many people trying to get the code as they had trouble implementing the algorithm. In fact, the only public code I found was TDMiner, which was in the form of Java and C++ code. The problem with that source code was that it only allows for a single item in each itemset for events. The example files shows subsecond timing for events, which my log files do not have.

The WINEPI implementation uses the Apriori method of growing the patterns to be tested from frequent patterns, and the pattern test on the data is done using state machines. The episodes that are searched are also restricted to injective ones. The general method is of a sliding window along the time-
axis of the data. For parallel episodes, the patterns are matched when the total number of component items found in the window is the same as the number of items in the episode. For series episodes, a state machine is spawned for each instance of the pattern any time the first event is seen, and when it matches, the windows are counted until the first item is no longer in the window. As there can be overlapped instances of these matching episodes, care has to be taken to not over-count the overlapped windows.

I had no care for trying to write code that mirrored WINEPI design, but I figured there had to be another way to do the mining, especially without the injective restriction, as I needed to mine episodes that have repetitions of items.

### 2.2 Theoretical Implementation

I will present the considerations I spent a good deal of time thinking about when trying to figure out a different way to implement episode mining. Let us assume we are using windows of size \( w \), and the number of item types is \( I \). To simplify things, we can restrict events to single items (unless noted).

For mining serial episodes, one would love to simply take each window (of length \( w \)) in the sequence and map the series of events into \( w \)-dimensional space, then simply record the resulting episodes that meet the minimum support. The way this would need to be done would need to take into consideration all different permutations of episodes that can result from a series of events. The binary space needed to hold all the permutations would be \( I^w \) in size.

The process would go like this. An \( I \)-wide binary array represents the first item in an episode (all set to False). For all items in the window, their binary bit is set to True, as any item in the window can be seen as the first item of an episode. Anyway bit that is True is then allocated a new \( I \)-length array for the second. Starting with the second item in the window, every item sets to True the bit corresponding to the second the bit corresponding to the second level arrays for each item in front of them; arrays are then made for True selections. For the third round, each item starting with the third uses any serial permutation of two items as its prefix, and so on. By the last round, there will be only one path that gets mapped out, being the episode including all items in the window in order. The true bits are then used to increase values in a parallel tree of integers (which is even bigger in scope).

Ignoring the storage size problem (which is bad), how bad is this iteration process? For a window with \( n \) items in it, there are \( 2^n - 1 \) permutations that need to be mapped out, every time a window is processed. If we were including itemsets with multiple items, the numbers are even worse, as the itemsets need to be represented as all permutations of ordered items: for a simple example, \( \langle AB \rangle \) can be represented by episode patterns \( \langle A \rangle, \langle B \rangle, \langle AB \rangle \) and \( \langle BA \rangle \). The permutation count scales faster than \( 2^n \) for itemsets.

A great deal of the permutation mapping is redundant between neighboring windows. Instead of doing all item permutations for each window, we do all the window permutations for each item. What this means is that we move along the time line, and consider each event separately, but taking into account the past generated permutations with their lifetimes. This is done by generating a pattern tree under each item that has a lifetime associated with it.

Suppose we encounter \( A \) at time 5, where we have a window size of 4. There is no history, so we simply make an \( \langle A \rangle \) tree with lifetime of 4. This tree is then applied to the pattern count arrays, and the lifetime is decreased to 3. The next item is \( B \) at time 7, so we process window 6 first, meaning that \( \langle A \rangle \)
is processed again, and its lifetime decremented to 2. At window 7, we append a new leaf to all existing nodes with B at the end, which means we add <AB> to the <A> tree; there is no lifetime associated with this node, as when A expires, so does this node. A new <B> tree is added with lifetime of 4. All tree patterns are applied to the count arrays, and lifetimes are decremented. This continues until there are no more items, and all patterns have disappeared. This method lets the knowledge of the past be used for the current item, cutting down on permutation solving. A further optimization would be for any time gap of no new items, to calculate the number of windows each tree will contribute (from their lifetimes), and apply that count all at once instead of for each time step.

This method is nice, but it involves tree management, which is very annoying. A simple way to do this is to realize how they grow. If the empty set is included as a base pattern, any new item doubles the number of patterns stored. This can be done efficiently in an array. In the above example, we can start with [0] as the initial array. When we add A, we append to all existing elements, leading to [0, <A>]. When we add B, we get [0, <A>, <B>, <AB>]. With a new C, we get [0, <A>, <B>, <AB>, <C>, <AC>, <BC>, <ABC>]. Now assume the array positions are indexed starting with zero. To remove item A, remove all the odd items. In fact if the previous optimization is used when there is a time gap of no new items, one can see that for the number of elements dropped D, the patterns where the position modulo \(2^D\) is zero should be added to the counts.

Still, one would need a great deal of memory or a small or predictable set of data to do all this expansion. At this point I switched to implementing parallel episodes in a way that seemed very direct.

3 Parallel Non-Injective Episode Mining

For parallel episode mining, with episodes being injective, it is straightforward to implement this by using frequent item set mining, such as with FPGrowth. As we are looking for the number of windows containing the items, instantiate all the possible windows as itemsets, and give them to FPGrowth. As there is likely to be many duplicate windows, one might be able to specify a weight for the windows being the duplicity.

This might seem like more work than WINEPI, as we are duplicating the data many times into windows. However, if we are using an algorithm like FPGrowth, it compresses the data internally so hopefully this is compensated somewhat.

For episodes that are not injective, it is a bit harder, but it can be done two ways: in a modified FPGrowth, or with pre- and post-processing of the input data. Essentially, for each item that has multiple instances within a window, say A : 3, we need to add to the window the lesser multiples, like A : 2 and A : 1. Suppose you had windows of (A : 1, B : 1), (A : 1, B : 1), and (B : 3, C : 2) and were looking for a support of 3. If the B items were considered differently because they had different counts, the correct pattern of B : 1 would not be found.

In the case of pre-and post-processing, the preprocessing involves computing each window and counting the the number of each item found, and representing these items with their actual counts and their lesser counts. After FPGrowth is run, the post-processing step is for each found pattern to remove any lesser multiples for any item, as they will exist for any pattern item that has a count more than one.

For a modified FPGrowth, the process can be done more efficiently. The input is now window itemsets,
but each item has a count associated with it. The customized program finds all the different value pairs (of type and count) and puts them into a structure array, creating any missing items having a lower count value. The data is then reread to convert the items in the windows to these indexes, and any item with a multiple count will also have the indexes from the items with lower counts added. This is similar to the previous filtering method, except that we now can use the array to check the actual properties.

For FPGrowth, the items with a lower count are at the end of the compressed data, and are searched first. In the case of multiples, we know that there will never be more cases of an item with a higher count than a lower count \((F:2 \text{ support can't be greater than } F:1 \text{ support})\). At worst they are even in count. When the item indexes are sorted according to count, a sort condition is also placed that any indexed items will check to see if the types are the same, and if they are, the one with the lower count is placed earlier. This makes sure that even if \(F:2\) and \(F:1\) have the same count, \(F:2\) will be placed further from the root, and will be searched first.

The reason we want FPGrowth to search the higher count items first is that when one is used in a prefix to generate candidates, it ignores all item index values that refer to an item of the same type. Prefix projection with this higher valued item will create a projected database where all the lower valued items of the same type will have the exact same count, so expanding on these values is wasteful and pointless. Doing this also removes the need to do the type of post-processing needed with the script method, where these lower-valued items are removed.

At the end of this modified FPGrowth, the pattern indexes are reverted back to their original item values. To test all of this, I implemented these methods in FPGrowth code that I wrote for CS412. It worked as I wanted, although it was slow as my implementation was not optimal.

Both version of this parallel episode mining by default would return all matching permutations of items counts, such as \((A:2,B:1)\) would also return the subset \((A:1,B:1)\), (at possibly a different support value). The number of these values can be reduced by implementing methods that reduce the value to closed patterns or maximal patterns.

I am not sure the speed difference between either method of implementation and that of say WINEPI (because I don’t have that code) or TDMiner (as that code works with data that has more restrictions). Of the two methods I presented, the custom FPGrowth was a good deal more work, with many potential pitfalls in implementation, but also had the most potential to be faster. The processing method to me is even more intriguing in that it has a great deal more options that can be easily done, even if done at the expense of speed. For instance, with the filtered input one can drop all instances of items that don’t have a count of at least two, and any result that comes out would have a pattern where each item has a multiplicity of at least two, which cuts down a great deal on the output of the program, but also the processing needed. This is exactly the type of thing my original project goal needs.

4 Serial Non-Injective Episode Mining

Serial episode mining for non-injective episodes can be done using unmodified FPGrowth, while it is not as simple as with parallel episode mining. The basic structure of the algorithm is Apriori in nature, and is similar in structure to GSP [3] and SPADE [4]. While they are intended for mining unrelated sequences,
they can be used on all the possible windows of the sequence. One could simply use PrefixSpan [5] directly on these windows, but as the number of windows now is very large, this might become very slow.

In GSP and SPADE, episode lengths are grown one item at a time by combining frequent $n$-length patterns to create $(n+1)$-length pattern candidates (removing any that have infrequent subpatterns). This proposed algorithm does the same thing, but by a different process and exerting a new condition on candidate generation. For a given round $n+1$, it uses the found candidate episode patterns (from the last round) to create new sequence windows that replace the ordered items with a group of indices corresponding to the found patterns. These transformed windows will only contain $n$-length patterns, and they are not known to be frequent yet. Instead of determining the $n$-item frequent patterns by counting, we run the windows through the FPGrowth, which does this, but also more. We get returned frequent itemsets of the $n$-length patterns, and only combinations of $n$-length patterns represented by these frequent itemsets can be frequent. At this point, all frequent itemsets of count 2 (either filtered out of the full set of frequent itemsets or from the maximal itemsets) are attempted to be used to create candidate $(n+1)$-length patterns episode pattern, which are then given to the next round; candidates also need to be attempted to be combined with their own selves, no allow for episode pattern repetition. This process will constrain the search area to only the possible combinations represented within the frequent episode itemsets, hopefully speeding up the search.

The non-injective episodes are taken care of naturally during this process, as all possible combination of frequent patterns are combined, which will generate patterns including the same item multiple times.

The main difference between GSP and SPADE is how the data is stored in memory to allow for searching for the candidate sequences (with SPADE’s version being faster). During this process, additional restrictions can be placed on the data, like enforcing a minimum time between events or only looking for patterns that contain at most $N$ item types.

Due to time constraints, and the need for more programming to implement this algorithm, I only tested it with toy examples and by running it by hand to make sure it worked. Ideally I would have tested the speed of this method versus simply running PrefixSpan on the windows and compared the results.

5 Log Mining for User Events

Unfortunately, by the time I figured out how to get non-injective episode mining truly working, I ran out of time. I present below how I propose to do the mining on the data logs.

I would do parallel data mining on the log files, using a window of 100 seconds. I would restrict the pattern items to having at least a count of three. From these patterns, I would look for their occurrences and see if they fall in localized areas of the sequences (meaning they are not background items) with no consistent time spacing. If so, for each occurrence I would find the last instance of the pattern items, and then pull the next 100 seconds as a subsequence. I would then use serial pattern matching on these new subsequences to look for any frequent serial episodes. I would check these frequent episodes against the whole data sequence to make sure they are localized, and the timing between events is generally consistent. If so, this would imply the sort of pattern I am looking for.

Ideally this would show a user tweaking one or several PVs multiple times before running a scan, which has a definite order of events. Unfortunately, I don’t
have results to show this at this time.

6 Conclusion

While I was unable to finish my goal of being able to find instances of user interaction in my EPICS log files, I was able to figure out a way to do both non-injective parallel and non-injective serial episode mining using tools that are available to me.

The proposed parallel episode mining algorithm is fairly straightforward. By using the processing version, one leverages the immense amount of optimization that has gone into the FPGrowth program available on the Internet.

The proposed serial episode mining algorithm is more complicated, and essentially becomes a type of sequential pattern mining algorithm (I didn’t realize this until very late), where it uses FPGrowth internally to do the optimization. I unfortunately have not had time to test how well this works in reality versus other algorithms such as GSP, SPADE, and PrefixSpan.

For the future, I plan to write the code for the serial episode mining algorithm, so that I can test how well it works. I will then follow through on trying to use these tools for looking for user interaction in the EPICS PV streams.

References


