LINGI2364 - Mining Patterns
Project 2 - Implementing Sequence Mining

Group 10
de Voghel Brieuc - 5910 1600
Mersch-Mersch Severine - 3298 1600

Friday 17th April, 2020

1 Introduction

In this report we explain our different implementations of the PrefixSpan sequence mining algorithm. We derived this algorithm to mine sequences in different ways : based on their frequency, their supervised scoring functions, in open and closed forms.

2 Implementations

First we implemented the basic PrefixSpan algorithm to mine frequent sequences according to the sum of their support in the positive and negative class. This allowed us to have a starting structure for the rest of the project, since the other implementations are an improvement on this basic implementation.

We will start by clarifying the data structures used in our implementations.
As a reminder, PrefixSpan adopts a "pattern growth" approach, the sequence mining is thus done in a Best First Search (BFS) manner in a search tree.

2.1 Data structures

Three different data structures are used:

1. *Heap* is a data structure easing the use of a min- or max-priority queue. It stores items that are 2-tuples with the first element being the key, and the second element being the value. It supports all classic methods one could expect.
   Heap is extended for closed sequence mining allowing for more intelligent methods : finding and removing super- or sub-sequences.

2. *Dataset* is a data structure for the transactions in the positive and negative classes. A dataset is instantiated from two class files. It lists every transaction with its symbol's positions allowing for an efficient branching for PrefixSpan.
   It also stores invariants like P, N, the transaction lengths, etc. allowing for better performances.

3. *Node* data structure's instances are the nodes of our BFS PrefixSpan branching tree. These will be contained in the *Heap* representing the BFS tree. Every node represents a sequence in the dataset. Nodes have several fields and methods necessary for PrefixSpan itself :

   - *sequence* is the sequence which the node represents.
   - *pid* is an array of the position id in the transactions (according to PrefixSpan).
   - p and n are respectively the positive and negative supports of the sequence in the dataset.
   - *branch method* returns a new *Node* instance, branched from the actual node according to the given symbol.
     The returned instance will have its pids advanced according to PrefixSpan. It therefore uses the symbol's positions list of the dataset.

- *score method* computes the score of the actual sequence.
  The score can be computed for *supsum* (the sum of the positive and negative supports), *wracc* (the weighted relative accuracy), *abswracc* (the absolute value of weighted relative accuracy) or *infogain* (the information gain based on the entropy impurity); depending on the implementation and on the asked scoring function.
- *heuristic method* decides whether the instance's branching can be pruned from the tree, depending on the given heuristic values (the heuristics themselves are explained later on).

2.2 Frequent Sequence Miner

After reading the input files and creating the dataset, a max-heap representing the BFS tree is created. We iterate and branch on all the elements on this heap in order to grow the sequences. Branched nodes are added to the heap.

As the *supsum* scoring method is anti-monotonic, the pruning heuristic is quite simple. Also, all k first encountered sequences are guaranteed to be the k most frequent ones (thanks to the BFS max-heap and the anti-monotonicity propriety). Those sequences are printed with their corresponding support.

Heuristic

The pruning heuristic of the anti-monotonic *supsum* scoring method is quite straightforward : if (a) the number of branches is larger than the number of sequences we still have to find, or if (b) the branch's score is smaller than the k'th element on the heap, we prune. In both those cases, the branching would result in adding nodes that have no chance of having a better support than those already on the heap.

2.3 Supervised Sequence Miner

This implementation is largely based on the previous one. The only real, but not to be underestimated, difference is the scoring method not being anti-monotonic anymore.

Thus, a second heap is created, this time in ascending order. It will contain the nodes of the k best sequences.

Every time a node is popped from the BFS heap, we check if it can be added to the *k_best_nodes* heap. This means that (a) if this heap already contains the node's score, we can just add the node; (b) if the heap still doesn't contains k different scores we can also add the node; and (c) if the heap already contains k different scores but the node's score is better than the worst score contained in the *k_best_score* heap, we add the new node and remove all the ones having the worst score. In any other case, the new node is not added to this heap.

Then, as for the frequent miner, the branches of the current node are added to the BFS heap if they satisfy the heuristic.

It is only at the end, when the BFS heap is empty, that the k best sequences are printed.

Heuristic

$$Wracc(x) = (\frac{P}{P+N} * \frac{N}{P+N}) * (\frac{p(x)}{P} - \frac{n(x)}{N}) \tag{1}$$

The weighted relative accuracy (1) is not anti-monotonic.

We thus use the following pruning heuristic :

The *min_p* and *max_n* values denote respectively the smallest positive support and the largest negative support a sequence (and thus its branches, since these proprieties are anti-monotonic) can have in order to have its Wracc higher than the worst of the k best scores.
Our heuristic thus prunes the sequences having neither a positive support higher than *min_p* nor a negative support smaller than *max_n*.

- *min_p* is worth $\frac{(P+N)^2}{N} * wracc_{min}$
- *max_n* is worth $\frac{-(P+N)^2}{P} * wracc_{min}$

with P and N being the dataset's total positive and negative support, and $wracc_{min}$ being the worst of the k best scores.

2.4 Supervised Closed Sequence Miner

Again, this implementation is based on the previous one, with this time only a feature added for when a sequence has to be pushed on the *k_best_nodes* heap. For this, the extension of *Heap* (described above) had to be implemented.

Before adding a node to the *k_best_nodes* heap, we verify if the heap does not already contain a closed super-sequence of this node. If not, the node is pushed on the heap and all its sub-sequences that he closes, that were previously added, are removed, keeping only the closed sequences.

Heuristic

Since the scoring method does not change, the heuristic does not need to change either.

2.5 Alternative scoring functions

In order to modify the scoring function, we did not have to modify the previous implementation a lot : only the heuristics and the computation of the score had to be adapted.

2.5.1 Absolute value of the Wracc score

The score is simply the absolute value of the Wracc score (1).

For the heuristic, on the other hand, the *max_n* value has to be transformed into *min_n*, its opposite. The sequence's negative support has thus now to be larger than this value.

2.5.2 Information Gain

The score is computed as seen in the course, with the impurity being the entropy :

$$\begin{aligned} InfoGain(x) = impurity(\frac{P}{P+N}) - \frac{p(x)+n(x)}{P+N} * impurity(\frac{p(x)}{p(x)+n(x)}) \\ - \frac{P+N-p(x)-n(x)}{P+N} * impurity(\frac{P-p(x)}{P+N-p(x)-n(x)}) \\ impurity(x) = -x * log_2(x) - (1-x) * log_2(1-x) \qquad (2) \end{aligned}$$

In this case, we did not need any heuristic.

3 Analysis of the sequences found using the different scoring functions

We compared the closed patterns obtained with our supervised closed sequence miner with the 3 scoring functions wracc, abswracc and infogain (Table 1). The tests were done on the "Test" set as infogain always leads to a *MemoryError* on other, too large datasets.

In Table 1 we identified the recurring sequences as follow : **bold sequences** are the ones present in wracc that can also be found in abswracc and infogain, and *cursive sequences* are the ones present in abswracc that can also be found in infogain.

Note that some of our conclusions may depend on the tested set and are thus not generalisable to other datasets.

Comparing the output of the 3 scoring functions

As we can see, the best scores of wracc are contained in abswracc. This is normal since abswracc takes both the best and the worse scores of wracc, being the absolute value of wracc. If the positive and negative class are balanced, we can expect having the first half of wracc contained in abswracc, for the same k.
Here, we can see that for this tested set, the positive class is better represented since nearly all the sequences present in wracc are present in abswracc.

Infogain is a different way of computing, thus we expect some differences in the outputted sequences. However, we can see that there are a lot of sequences (all in this case) from abswracc that are present in infogain, although they have different final scores.

Infogain also has less diversity in his scores. Sequences with completely different supports may end up with the same score (take [B, A] 3 3 0.12809 and [A, C, C, B, A, B] 1 0 0.12809 for example - this may be due to the rounding to 5 decimal places). This leads to more outputted sequences for the same k as wracc or abswracc.

wracc, k = 7	abswracc, k = 7	infogain, k = 7
[A, A, C, A] 3 0 0.18367	*[C, B, A] 1 3 0.18367*	**[A, A, C, A] 3 0 0.52164**
[A, C, A] 4 1 0.16327	*[C, B, B] 1 3 0.18367*	*[C, B, A] 1 3 0.52164*
[A, B, A] 3 1 0.10204	**[A, A, C, A] 3 0 0.18367**	*[C, B, B] 1 3 0.52164*
[A, C, C, A] 3 1 0.10204	**[A, C, A] 4 1 0.16327**	**[A, C, A] 4 1 0.46957**
[A, A] 4 2 0.08163	*[C, B, B, A] 0 2 0.16327*	*[C, B, B, A] 0 2 0.46957*
[A, C, C, A, C, A] 1 0 0.06122	*[B, B] 2 3 0.12245*	*[B, B] 2 3 0.29169*
[A, B, B, A, C, A] 1 0 0.06122	*[C, B, A, B] 1 2 0.10204*	*[A, C, B, C, A, B] 0 1 0.19812*
[A, A, B, C, C, A] 1 0 0.06122	*[B, B, A] 1 2 0.10204*	*[C, B, B, A, B, B] 0 1 0.19812*
[A, C, C, B, A, B] 1 0 0.06122	**[A, B, A] 3 1 0.10204**	**[A, A] 4 2 0.19812**
[A, A, B] 2 1 0.04082	**[A, C, C, A] 3 1 0.10204**	*[C, C, B, B, A, A] 0 1 0.19812*
[A, B, C, A] 2 1 0.04082	*[C, C, B] 1 2 0.10204*	*[B, A] 3 3 0.12809*
[C, C, A] 3 2 0.02041	*[A, C, B, C, A, B] 0 1 0.08163*	*[C, B, A, B] 1 2 0.12809*
[A, B] 3 2 0.00598	*[C, B, B, A, B, B] 0 1 0.08163*	*[C, C, B] 1 2 0.12809*
	[C, C, B, B, A, A] 0 1 0.08163	**[A, C, C, A, C, A] 1 0 0.12809**
	[A, A] 4 2 0.08163	**[A, A, B, C, C, A] 1 0 0.12809**
	[B, A] 3 3 0.06122	*[B, B, A] 1 2 0.12809*
	[A, C, C, B, A, B] 1 0 0.06122	**[A, B, A] 3 1 0.12809**
	[A, C, C, A, C, A] 1 0 0.06122	**[A, B, B, A, C, A] 1 0 0.12809**
	[A, B, B, A, C, A] 1 0 0.06122	**[A, C, C, A] 3 1 0.12809**
	[A, A, B, C, C, A] 1 0 0.06122	**[A, C, C, B, A, B] 1 0 0.12809**
	[A, A, B] 2 1 0.04082	**[A, A, B] 2 1 0.02024**
	[A, B, B] 2 2 0.04082	**[A, B, C, A] 2 1 0.02024**
	[A, B, C, A] 2 1 0.04082	*[A, B, B] 2 2 0.02024*
		[B, B, A, A] 1 1 0.00598
		[C, C, B, A] 1 1 0.00598
		[C, C, A] 3 2 0.00598
		[A, B] 3 2 0.00598
		[C, C, B, B] 1 1 0.00598
		[C, C, A, A] 1 1 0.00598
		[A, C, B, A, B] 1 1 0.00598
		[A, C, C, A, B] 1 1 0.00598

Table 1: 7 best closed sequences for each scoring function.

4 Conclusion

After understanding frequent sequence mining algorithms and implementing them, we compared the different scoring functions. Some efficiency issues were solved using position tables for PrefixSpan, saving up to 50% of computing time.

We observed that infogain and abswracc scoring output almost the same best sequences, while wracc has less common scores for the same k.