Apriori All Algorithm

Data mining project

30 November 2012

BALTYN KAROLINA
BYSIEK MATEUSZ

Document metric

Table of contents

1. Description of the problem.

The purpose of this document is to describe important aspects of the data mining project concerning sequential patterns.
Sequence mining is a topic of data mining that is connected with finding relevant patterns between examples of data, where the values are delivered in a sequence. It is important from statistical point of view. An example of such a pattern is that customers typically rent “Matrix”, then “Matrix Reloaded” and finally “Matrix Revolutions”. These rentals do not need to be consecutive. Customers who rent some other videos in between also support this sequential pattern. One of the main applications of sequence mining is increasing profits and decreasing costs by proper management of shelf space allocation and products display.
As a start point, we create a large database of customer transactions, where each transaction consists of customer-id, transaction time and the items bought in the transaction. We are asked to find maximal sequences of items that are most frequently bought in some particular order.
Our main task is to program an application that takes a database as an input and produces results using Apriori All algorithm. The algorithm should always find an answer for any data given in a proper format.

2. Background.

In this section we are going to talk about the general solution, some theory we use and give a pseudocode of the algorithms from our application.
Our solution will be given is this form: first: {item1, item2} next: {item3, item4} while the input will be a set of customer transactions, as described above. The output will be an ordered list of sets of items.
To know if some sequence appears enough number of times in the database, we are going to check the support of this sequenc*e. Suppo*rt of a sequence b in a set of transactions X is defined as the number of clients who have b in their transactions, divided by the total number of clients, or more formally:

$$s_x(b) = \frac{|\{x \in X : b \subseteq X\}|}{|X|}$$

Before the algorithm is run, we should define some minimum support for our database. It is such number x for which all the sequences b we choose have support greater or equal than the minimum. In other words,

$$s_x(b) \geq x$$

If a sequence fulfills the above condition, we can say that it is a *frequent sequence*. We should remember that each subsequence of the frequent sequence must be frequent.

We split the problem of mining sequential patterns into the following phases:

1. **Sort phase.** The database is sorted, with customer-id as the major key and transaction-time (or transaction-order) as the minor key. This step converts the original transaction database into a database of customer sequences.
2. **Frequent itemsets (litemsets) phase.** All frequent sequences of length 1 are found (frequent sets). The set of litemsets is then mapped to a set of contiguous integers. The reason for this mapping is that by treating litemsets as single entities, we can compare two litemsets for equality in constant time, and reduce the time required to check if a sequence is contained in a customer sequence.
3. **Transformation phase.** Each customer transaction is replaced by the sets contained in this transaction. If a transaction contains no frequent set it is not retained in the transaction dataset. If a customer sequence does not contain any frequent sequence then it is dropped from the dataset (however this customer is counted).
4. **Sequence phase.** Frequent sequences are found with the use of Apriori All algorithm.

   Apriori All:

```
{
      $L_1$ = set of all frequent 1-sequences - frequent sets (k-sequence =
      sequence of length k)
            for(k=2;$L_{k-1} \neq \emptyset$; k++)
            {
            $C_k$ = set of candidates for frequent k-sequences for $L_{k-1}$
                  for each customer sequence c
                  {
                        for all candidate sequences d in $C_k$ contained in
c
                        { d.count++} //if the examined sequence d is in
customer's sequence c, then increase the support of d by 1
                  }
                  $L_k$ = { d: d $\in$ $C_k$ , d.count >= s} //s = minimal support
      }
            return (Maximal sequences in $L_k$)
}
```

   Generation of candidates for frequent k-sequences:

   1. Join L_{k-1} with L_{k-1}
   2. Select those sequences which have k-2 consecutive itemsets the same
   3. Delete all such sequences c from C_k that some (k-1)-subsequences of c are not in C_k

5. **Maximal phase.** We find maximal frequent sequences.

3. Description of the own solution.

opis algorytmu

Solution is based on pseudo-code obtained from our lecturer.

In this section we will describe our solution according to the phases described above.

1. **Sort phase**
   This phase is already done from the beginning. We have a database in xml file, which defines what items where bought by users in each transaction. The database is sorted by customer. Later to simplify we assume, that transactions appear in chronological order. Example of our xml database:
```
<Customers>
  <Customer Id="1">
   <Transaction>
     <Item>30</Item>
     <Item>90</Item>
   </Transaction>
  </Customer>
  <Customer Id="2">
   <Transaction>
     <Item>30</Item>
   </Transaction>
   <Transaction>
     <Item>40</Item>
     <Item>60</Item>
     <Item>70</Item>
   </Transaction>
  </Customer>
</Customers>
```

2. **Frequent 1-itemsets phase**
   Frequent 1-itemsets (litemsets) are found with the use of Apriori algorithm. As an argument our function takes the value of minimum support, to compare with our results and choose the proper litemsets.
```
        foreach (Customer c in customerList.Customers)
              {
                  foreach (Transaction t in c.Transactions)
                  {
                      //generate subsets (candidates for litemsets)
                      List<Litemset> candidateLitemsets =
                      generateCandidates(t.Items);

                      //check if they already exist in litemsets list; if not,
                      add //a litemset to litemsets
```

```
                    foreach (Litemset lset in candidateLitemsets)
                    {
                    ( . . . )
```

We iterate through each client, each transaction and each created litemset. The method generateCandidates() takes the set of items contained in the transaction and transforms it into all possible subsets. The number of such subsets is $2^{t.Items.Count}$ - 1.

```
// checking if lset exists in litemsets
IEnumerable<Litemset> l = litemsets
.Where(litemset => (litemset.Items.Count == lset.Items.Count)
&& litemset.Items.All(item => lset.Items.Exists(lsetItem =>
lsetItem.CompareTo(item) == 0)));
```

We check if in our results (litemsets list) there exists already a litemset lset that we are considering at the moment. If we cannot find it (which means the size of list l is zero), we add this litemset to litemsets list and increase its support by 1. Otherwise, we just take this found litemset and also increase its support. We have to remember that the support can be increased only once per client.

```
// rewrite the litemsets with support >= minimum to a new list
List<Litemset> properLitemsets = new List<Litemset>();
foreach (Litemset litemset in litemsets)
        if (litemset.Support >= minimalSupport)
                   properLitemsets.Add(litemset);

properLitemsets.Sort();
```

In the end we just choose those litemsets with support bigger than the minimal and sort the solution.

3. **Transformation phase**

We determine which of the given litemsets are contained in a customer sequence; to make this test faster we transform each customer sequence in an alternative representation: each transaction is replaced by the set of all litemsets contained in that transaction

1. if a transaction does not contain any litemset, it is not contained in the transformed sequence
2. if a customer sequence does not contain any litemset, this sequence is dropped from the transformed database (however this customer is counted in the total number of customers)
3. a customer sequence is now represented by a list of sets of litemsets

transaction -> a set of litemsets
sequence -> list of sets (transactions)

At first, we define two dictionaries, one encoding, one decoding:

```
 encoding = new Dictionary<Litemset, int>();
decoding = new Dictionary<int, Litemset>();

int i = 1;
foreach (Litemset li in oneLitemsets) {
   encoding.Add(li, i);
   decoding.Add(i, li);
   ++i;
}
```

Then, we encode our customer list (which is, after simplifications, in fact a List of Lists of Lists of Items) as a list of lists of lists of integers:

```
 var encodedList = new List<List<List<int>>>();

foreach (Customer c in customerList.Customers) {
   var encodedCustomer = new List<List<int>>();
   foreach (Transaction t in c.Transactions) {
         var encodedTransaction = new List<int>();
         int id = -1; // temp. variable used as in-out param
         // adding litemsets of length >= 2
         foreach (Litemset li in oneLitemsets) {
               if (li.Items.Count == 1)
                     continue;
               bool someMissing = false;
               foreach (Item litem in li.Items) {
                     if (!t.Items.Contains(litem)) {
                           someMissing = true;
                           break;
                     }
               }
               if (!someMissing) {
                     if (encoding.TryGetValue(li, out id))
                           encodedTransaction.Add(id);
               }
         }
         foreach (Item i in t.Items) {
               // adding litemsets of length == 1
               foreach (Litemset li in oneLitemsets) {
                     if (li.Items.Count > 1)
                           continue;
                     Item item = li.Items[0];
                     if (item.Equals(i)) {
                           if (encoding.TryGetValue(li, out id))
                                 encodedTransaction.Add(id);
                     }
```

```
                }
            }
            if (encodedTransaction.Count > 0)
                encodedCustomer.Add(encodedTransaction);
        }
        if (encodedCustomer.Count > 0)
            encodedList.Add(encodedCustomer);
    }

    return encodedList;
```

4. **Sequence phase**

We use the set of litemsets to find the desired sequences

In our implementation, list of k-sequences is partitioned by k. i.e. i-th element of resulting List contains all i-sequences:

```
    var kSequences = new List<List<List<int>>>();
```

at first, we add our initial data gained from previous steps, i.e. encoded 1-sequences:

```
    kSequences.Add(new List<List<int>>()); // placeholder for 0-sequences (empty)
    kSequences.Add(new List<List<int>>()); // 1-seq, already done, just copy:
    foreach (Litemset li in oneLitemsets) {
        var lst = new List<int>();
        lst.Add(encoding[li]);
        kSequences[1].Add(lst);
    }
```

then, we proceed with Apriori All. Contents of supporting method `GenerateCandidates()` is given after the body of main part of the algorith

```
    for(int k = 2; kSequences.Count >= k && kSequences[k - 1].Count > 0; ++k) {
        // list of kSequences, initially empty
        kSequences.Add(new List<List<int>>());
        var prev = kSequences[k - 1];
        // generate candidates
        var candidates = new Dictionary<List<int>, int>();
        GenerateCandidates(prev, candidates);
        // calculate support of each candidate by analyzing the whole encoded input
        Dictionary<List<int>, int>.KeyCollection keysOrig = candidates.Keys;
        List<List<int>> keys = new List<List<int>>(keysOrig);
        for (int i = 0; i < keys.Count; ++i) {
            List<int> candidate = keys[i];
            // check every customer for compatibility with the candidate
```

```
                foreach (List<List<int>> encodedCustomer in encodedList) {
                    bool allNeededItemsArePresent = true;
                    foreach (int candidateItem in candidate) {
                        // check all transactions, item must exist in any of them
                        bool foundCandidateItem = false;
                        foreach (List<int> encodedTransaction
                                in encodedCustomer) {
                            //if (encodedTransaction.Count < k)
                            //    continue;
                            foundCandidateItem =
encodedTransaction.Contains(candidateItem);
                            if (foundCandidateItem)
                                break;
                        }

                        // if item does not exist in any of the transactions,
this customer
                        // is not compatible with 'candidate' sequence
                        if (!foundCandidateItem) {
                            allNeededItemsArePresent = false;
                            break;
                        }
                    }
                    if (allNeededItemsArePresent) {
                        candidates[candidate] += 1;
                    }
                }
                // confront results with min. support
                if (candidates[candidate] >= minSupport)
                    kSequences[k].Add(candidate);
            }
        }

        return kSequences;
```

Below, code of `GenerateCandidates(List<List<int>> prev, Dictionary<List<int>,int> candidates)`
where:

1. prev are previous k-sequences, i.e. (k-1)-sequences
2. candidates: output, candidates for k-sequences

```
     for (int i1 = 0; i1 < prev.Count; ++i1) {
        List<int> l1 = prev[i1];
        for (int i2 = i1 + 1; i2 < prev.Count; ++i2) {
            List<int> l2 = prev[i2];
            int differentValue = -1;
            int diff = 0;
            foreach (int elem1 in l1) {
                  if (!l2.Contains(elem1)) {
                        ++diff;
                        if (diff > 1)
                              break;
                        differentValue = elem1;
                  }
            }
            // we cannot form candidates from lists that differ in more than one
element
            if (diff != 1)
                  continue;
            List<int> candidate = new List<int>(l2);
            candidate.Add(differentValue);
            candidate.Sort();
            // we don't want to add any duplicates
            bool foundEqual = false;
            foreach (List<int> cand in candidates.Keys)
                  if (cand.Count == candidate.Count) {
                        bool isEqual = true;
                        for (int icc = 0; icc < candidate.Count; ++icc)
                              if (cand[icc] != candidate[icc]) {
                                    isEqual = false;
                                    break;
                              }
                        if (isEqual) {
                              foundEqual = true;
                              break;
                        }
                  }
            if (foundEqual)
                  continue;
            candidates.Add(candidate, 0);
      }
 }
```

5. **Maximal phase**

We find the maximal sequences among the set of large sequences. In some algorithms this phase is combined with the sequence phase to reduce the time wasted to count nonmaximal sequences. We decided to not combine this phase, because in our opinion in our case it would jeopardize the integrity of list k-sequences. We start from the largest sequences, and then we proceed to remove smaller ones. Method `PurgeAllSubSeqsOf()` is a recursive method which removes all smaller subsequences when any obsolete subsequence is found. By using such optimization we decrease complexity of the whole operation, because we can be 100% sure that if we find that sequence A is not maximal, surely all its subsequences are also not maximal.

Other optimization, which we use but which is not visible at first glance, is a method which checks if a sequence is a subsequence of other one: `IsSubSequence(sequence, longerSequence)`. It has linear time because earlier we make sure that all encoded sequences are sorted. And comparing sorted sequences for inclusion can be easily done in worst case linear time. Best case running time of such check can be constant/unit, if first elements of sequences are related in certain way.

```
 // additional "-1" because all largest k-sequences are for sure maximal
for (int k = kSequences.Count - 1 - 1; k >= 0; --k) {
   List<List<int>> sequencesOfLengthK = kSequences[k];
   for (int n = sequencesOfLengthK.Count - 1; n >= 0; --n) {
         List<int> sequence = sequencesOfLengthK[n];
         for (int i = k + 1; i < kSequences.Count; ++i) {
               foreach (List<int> longerSequence in kSequences[i])
                     if (IsSubSequence(sequence, longerSequence)) { // if
sequence is a sub-seqence of s
                           // purge sequence and all its subsequences
                           PurgeAllSubSeqsOf(kSequences, k, n);
                           sequencesOfLengthK.RemoveAt(n);
                     }
         }
   }
}
```

It is important to notice that after the usual steps of the algorithm, the output is not ready. We have to decode the list using decoding dictionary prepared at the beginning of transformation phase. At this point we also simplify list of k-sequences, for convenience of the programmer.

```
 var decodedList = new List<Customer>();

var compactSequences = CompactSequencesList(kSequences);

foreach (List<int> sequence in compactSequences) {
   Customer c = new Customer();
   foreach (int encodedLitemset in sequence) {
         Transaction t = new Transaction(decoding[encodedLitemset].Items);
         c.Transactions.Add(t);
   }
   decodedList.Add(c);
}
```

Then, we have perform another step a.k.a. **Maximal Phase’** to remove non-maximal sequences from decoded output. This has to be done due to relationships between litemsets, which were not visible when the litemsets were stored in encoded format. For example, when we encode like this: (milk)->1 (corn-flakes)->2 (milk corn-flakes)->3, the sequence (1 2 3) may be considered maximal. But, after decoding, hidden relationships become visible. Because of that fact we can complete the maximal phase only after decoding.

```
 foreach (Customer customer in decodedList) {
    for (int tn = customer.Transactions.Count - 1; tn >= 0; --tn) {
          // tn : transaction no.
          Transaction t = customer.Transactions[tn];
          foreach(Transaction comparedTransaction in customer.Transactions){
                if (Object.ReferenceEquals(t, comparedTransaction))
                      continue;
                if (t.Items.Count >= comparedTransaction.Items.Count)
                      continue;
                if(IsSubSequence<Item>(t.Items, comparedTransaction.Items)){
                      customer.Transactions.RemoveAt(tn);
                      break;
                }
          }
    }
}
return decodedList;
```

The final return statement marks the end of the execution, the results stored in that variable at this point are the output of the whole algorithm.

4. Algorithmic analysis.

Time complexity of Apriori All algorithm should be calculated in steps (according to phases). Later the greatest time complexity of all phases is going to be chosen as our final result. Because the code for algorithms has already been pasted above, we will only show our results of calculations.

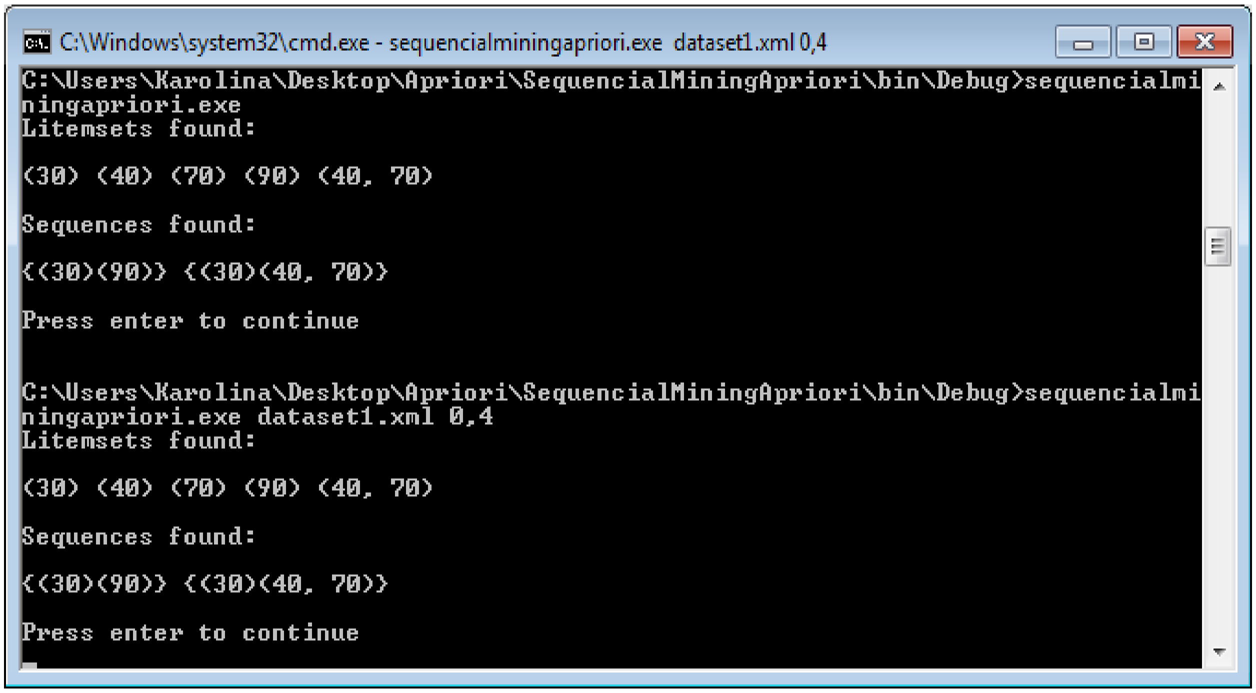
1. **Sort phase**
   The database (our input) was already sorted from the beginning, so there is nothing to calculate.
2. **Litemsets phase**
   This is the Apriori algorithm part. Apriori algorithm in our implementation has time complexity $O(d \cdot n \cdot 2^n - 1)$, where n is the number of distinct items that appear in the database and d is the total number of transactions. The worst case is calculated for the case when each transaction contains all possible items. 2^n-1 is the number of all subsets (without repetition) minus empty set. This is a brute force approach, where every subset is checked for being frequent.
3. **Transformation phase**
   $O(n \cdot d)$ where n is the number of distinct items that appear in the database, and d is total number of transactions. During transformation phase, program scans and converts every item in every transaction to its encoded form, i.e. integer.

4. **Sequence phase**
   For every k, generation of candidate sequences for k-sequences takes $O(x^5)$, where x is number of (k-1)-sequences. Maximum number of k-sequences for arbitrary k may be O(O(m choose k ⋅k!) (where m is number of distinct litemsets) due to the fact that in any k-sequence, any litemset may be present or not. Also, maximum value of k is equal to number of distinct litemsets.
   The process of finding sequences from prepared candidates of one length takes $O(d \cdot m^2)$ because the whole list must be scanned to gather information about support of a given candidate.
   In conclusion, total complexity of this phase is $O(m! \cdot d \cdot m^3)$.
5. **Maximal phase**
   Determining whether a given sequence is maximal is rather simple, because it can be done in $O(m \cdot d)$. Checking all sequences for maximality will take $O(m \cdot d^3)$.

5. User's Manual.

In order to test custom sequences, users can place their own databases in the root folder of the program. A database has to be an xml file and follow some pattern, which was already shown above.
In order to start the program with own input data, one can specify command-line arguments. The first argument is the name of the xml file (with extension), the second is the minimum support that a sequence must have. Minimum support is a double number greater than 0 and lower than 1.
It is also possible to run without arguments. Then the output is the solution of dataset1 database with minimum support 0.4

Example:



C:\Windows\system32\cmd.exe - sequencialminingapriori.exe dataset1.xml 0,4

```
C:\Users\Karolina\Desktop\Apriori\SequencialMiningApriori\bin\Debug>sequencialmi
ningapriori.exe
Litemsets found:

(30) (40) (70) (90) (40, 70)

Sequences found:

<(30)(90)> <(30)(40, 70)>

Press enter to continue


C:\Users\Karolina\Desktop\Apriori\SequencialMiningApriori\bin\Debug>sequencialmi
ningapriori.exe dataset1.xml 0,4
Litemsets found:

(30) (40) (70) (90) (40, 70)

Sequences found:

<(30)(90)> <(30)(40, 70)>

Press enter to continue
```

6. Technical documentation.

Documentation is generated from comments in the code. It is placed in `/doc` folder of the project.

7. Tests.

Apart from our program, we also created test projects, which check if the results obtained from the program are correct. Below we will present our two test databases and the correct outputs, which passed the tests.

Database #1

Client's ID	Items per transaction (in chronological order)
1	30 90
2	10, 20

	30 40, 60, 70
3	30, 50, 70
4	30 40, 70 90
5	90

Support = 0.25
Output: {(30) (90)}, {(30) (40, 70)}

Database #2

Client's ID	Items per transaction (in chronological order)
1	30 30, 40, 50 80 90
2	10, 20 30 40, 60, 70
3	10, 20 30, 50, 70
4	30, 40 50, 60 70, 80, 90
5	80 90
6	10, 50 80

Support: 0.4
Output:{(10)}, {(30) (40)}, {(30) (50)}, {(30) (70)}, {(50) (80)}, {(80) (90)}

8. Conclusions.

Apriori All algorithm is a good example of a way of finding sequential patterns. It has a very broad usage, especially in marketing.The solution given by Apriori All is a sequence of items/products that were chosen/bought one after another.
While working on the project, we came to a conclusion that the algorithm is really complex and it checks all the possibilities of maximal frequent sequences. It involves 5 main phases, including also implementation of basic Apriori algorithm first. We expected the complexity to be polynomial, but finally we noticed it was not enough. The algorithm stops for every input and produces correct output for every proper input. Several tests were done to prove that it produces good results.

9. Sources

General information about AprioriAll algorithm:
http://www-users.cs.umn.edu/~desikan/research/dataminingoverview.html#aprioriall

http://webdocs.cs.ualberta.ca/~zaiane/courses/cmput695-04/slides/Sequential-Yunping.pdf

http://www.docstoc.com/docs/123859368/Algorithm-AprioriAll

http://www.docstoc.com/docs/93423573/Web-Log-Mining-by-an-Improved-AprioriAll-Algorithm

About Apriori algorithm:
http://www-users.cs.umn.edu/~kumar/dmbook/ch6.pdf page 339