Frequent Item set Mining Methods

Jiawei Han und Micheline Kamber. Data Mining – Concepts and Techniques. Chapter 5.2

Julianna Katalin Sipos

Content

Introduction

Frequent sets play an essential role in many Data Mining tasks that try to find interesting patterns from databases, such as association rules, correlations, sequences, episodes, classifiers and clusters. The mining of association rules is one of the most popular problems of all these. The identification of sets of items, products, symptoms and characteristics, which often occur together in the given database, can be seen as one of the most basic tasks in Data Mining.

The original motivation for searching frequent sets came from the need to analyze so called supermarket transaction data, that is, to examine customer behavior in terms of the purchased products (Agrawal et al., 1993). Frequent sets of products describe how often items are purchased together.

Formally let I be the set of items.

A transaction over I is a couple T = (tid, I) where tid is the transaction identifier and I is the set of items from I.

A database D over I is a set of transactions over I such that each transaction has a unique identifier. We omit I whenever it is clear from the context

A transaction T = (tid, I) is said to support a set X, if X C I. The cover of a set X in D consists of the set of transaction identifiers of transactions in D that support X. The support of a set X in D is the number of transactions in the cover of X in D. The frequency of a set X in D is the probability that X occurs in a transaction, or in other words, the support of X divided by the total number of transactions in the database. We omit D whenever it is clear from the context.

A set is called frequent if its support is no less than a given absolute minimal support threshold min_sup with $0 \leq min\_sup_{abs} \leq |D|$. When working with frequencies of sets instead of their support, we use the relative minimal frequency threshold min_sup_{rel}, with $0 \leq min\_sup_{rel} \leq 1$. Obviously $min\_sup_{abs} = [min\_sup_{rel} * |D|]$. In this paper we will mostly use the absolute minimal support threshold and omit the subscript abs.

Let D be a database of transactions over a set of items I, and min_sup the minimal support threshold. The collection of frequent sets in D with respect to min_sup is denoted by F(D, min_sup):={X C I | support(X, D)≥min_sup} or simply F if D and min_sup are clear from the context.

Given a set of items I, a database of transactions D over I, and a minimal support threshold min_sup, find F (D, min_sup).

In practice we are not only interested in the set of sets F, but also in the actual supports of these sets.

For example, consider the database shown in the following table over the set of items I = {beer, chips, pizza, wine}:

Tid	Set of items
100	{beer, chips, wine}
200	{beer, chips}
300	{pizza, wine}
400	{chips, pizza}

Table 1

The following table shows all frequent sets in D with respect to a minimal support threshold equal to 1, their cover in D, plus their support and frequency:

Set	Cover	Support	Frequency
{}	{100, 200, 300, 400}	4	100%
{beer}	{100, 200}	2	50%
{chips}	{100, 200, 400}	3	75%
{pizza}	{300, 400}	2	50%
{wine}	{100, 300}	2	50%
{beer, chips}	{100, 200}	2	50%
{beer, wine}	{100}	1	25%
{chips, pizza}	{400}	1	25%
{chips, wine}	{100}	1	25%
{pizza, wine}	{300}	1	25%
{beer, chips, wine}	{100}	1	25%

Table 2

If we are given the support threshold min_sup, then every frequent set X also represents the trivial rule X=> {} which holds with 100% confidence.

The task of discovering all frequent sets is quite challenging. The search space is exponential in the number of items occurring in the database and the targeted databases tend to be massive, containing millions of transactions. Both these characteristics make it a worthwhile effort to seek the most efficient techniques to solve this task.

The Apriori algorithm

Together with the introduction of the frequent set mining problem, also the first algorithm to solve it was proposed, later denoted as AIS. Shortly after that the algorithm was improved by R. Agrawal and R. Srikant and called Apriori. It is a seminal algorithm, which uses an iterative approach known as a level-wise search, where k-itemsets are used to explore (k+1)-itemsets.

It uses the Apriori property to reduce the search space: All nonempty subsets of a frequent itemset must also be frequent.

- P(I)<min_sup => I is not frequent
- P(I+A)<min_sup => I+A is not frequent either
- Antimonotone property – if a set cannot pass a test, all of its supersets will fail the same test as well

In the next section we will see how the apriori property is used in the Apriori algorithm: Let us look at how Lk-1 is used to find Lk, for k>=2. We can distinct two steps: join and prune.

1. Join
    - finding Lk, a set of candidate k-itemsets is generated by joining Lk-1 with itself
    - The items within a transaction or itemset are sorted in lexicographic order
    - For the (k-1) itemset: li[1]<li[2]<…<li[k-1]
    - The members of Lk-1 are joinable if their first(k-2) items are in common
    - Members l1, l2 of Lk-1 are joined if (l1[1]=l2[1]) and (l1[2]=l2[2]) and … and (l1[k-2]=l2[k-2]) and (l1[k-1]<l2[k-1]) – no duplicates
    - The resulting itemset formed by joining l1 and l2 is l1[1], l1[2],…, l1[k-2], l1[k-1], l2[k-1]

2. Prune
    - Ck is a superset of Lk, Lk contain those candidates from Ck, which are frequent
    - Scanning the database to determine the count of each candidate in Ck – heavy computation
    - To reduce the size of Ck the Apriori property is used: if any (k-1) subset of a candidate k-itemset is not in Lk-1, then the candidate cannot be frequent either,so it can be removed from Ck. – subset testing (hash tree)

Let us take the following example:

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

Table 3

The join and prune steps for this example:

- Scan D for count of each candidate
    - C1: I1 – 6, I2 – 7, I3 -6, I4 – 2, I5 - 2
- Compare candidate support count with minimum support count (min_sup=2)
    - L1: I1 – 6, I2 – 7, I3 -6, I4 – 2, I5 - 2
- Generate C2 candidates from L1 and scan D for count of each candidate
    - C2: {I1,I2} – 4, {I1, I3} – 4, {I1, I4} – 1, …
- Compare candidate support count with minimum support count
    - L2: {I1,I2} – 4, {I1, I3} – 4, {I1, I5} – 2, {I2, I3} – 4, {I2, I4} - 2, {I2, I5} – 2
- Generate C3 candidates from L2 using the join and prune steps:

  - Join: C3=L2xL2={{I1, I2, I3}, {I1, I2, I5}, {I1, I3, I5}, {I2, I3, I4}, {I2, I3, I5}, {I2, I4, I5}}
  - Prune: C3: {I1, I2, I3}, {I1, I2, I5}
- Scan D for count of each candidate
  - C3: {I1, I2, I3} - 2, {I1, I2, I5} – 2
- Compare candidate support count with minimum support count
  - L3: {I1, I2, I3} – 2, {I1, I2, I5} – 2
- Generate C4 candidates from L3
  - C4=L3xL3={I1, I2, I3, I5}
  - This itemset is pruned, because its subset {{I2, I3, I5}} is not frequent => C4=null

The Apriori algorithm:

```
Input:
    § D, database of transactions;
    § min_sup, the minimum support count threshold
Output: L, frequent itemsets in D
Method:
    L1=find_frequent_1-itemsets(D);
    for(k=2; Lk-1!=null;k++){
        Ck=apriori_gen(Lk-1);
        for each transaction t Є D{ // scan D for counts
            Ct = subset(Ck, t); // get the subsets of t that are candidates
            for each candidate c Є Ct
                c.count++;
        }
        Lk={c Є Ck | c.count≥min_sup}
    }
    Return L=UkLk

procedure apriori_gen(Lk-1: frequent(k-1)-itemsets)
    for each itemset l1 Є Lk-1
        for each itemset l2 Є Lk-1
            if(l1[1]=l2[1])^(l1[2]=l2[2])^…^(l1[k-2]=l2[k-2])^(l1[k-1]<l2[k-1]) then{
                c=l1xl2; //join step: generate candidates
                if has_infrequent_subset(c,Lk-1) then
                    delete c; //prune step: remove unfruitful candidate
                else add c to Ck;
            }
     Return Ck;

procedure has_infrequent_subset(c: candidate k-itemset; Lk-1: frequent (k-1)-itemsets);
//use priori knowledge
    for each (k-1)-subset s of c
        if s !Є Lk-1 then
            Return TRUE;
    Return FALSE;
```

Generating association rules from frequent itemsets

Once the frequent itemsets from transactions in a database D have been found, it is straightforward to generate strong association rules from them, where strong association rules satisfy both minimum support and minimum confidence. This can be done using the following equation:

confidence(A=>B)=P(B|A) = support_count(AUB) / support_count(A)

The conditional probability is expressed in terms of itemset support count, where: support_count(AUB) is the number of transactions containing the itemsets AUB and support_count(A) is the number of transactions containing the itemset A. Based on this equation, association rules can be generated as follows:

- For each frequent itemset l, generate all nonempty subsets of l.
- For every nonempty subset s of l, output the rule "s => (l-s)" if support_count(l) / support_count(s)>=min_conf, where min_conf is the minimm confidence threshold.

Let's try an example based on the transactional data shown on Table 3. Suppose the data contain the frequent itemset l = {I1, I2, I5}. The nonempty subsets of l are: {I1, I2}, {I1, I5}, {I2, I5}, {I1}, {I2} and {I5}. The resulting association rules are as shown below, each listed with its confidence:

I1 and I2=>I5	Conf=2/4=50%
I1 and I5=>I2	Conf=2/2=100%
I2 and I5=> I1	Conf=2/2=100%
I1=>I2 and I5	Conf=2/6=33%
I2=>I1 and I5	Conf=2/7=29%
I5=>I1 and I2	Conf=2/2=100%

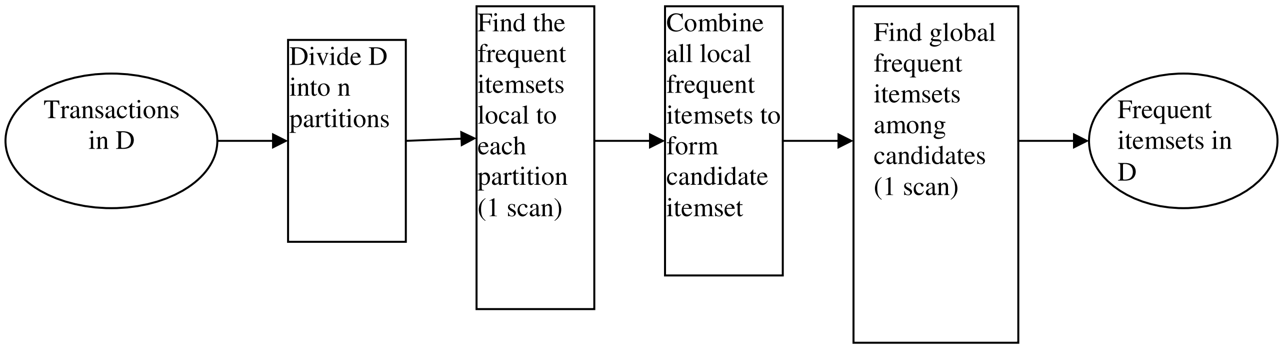
If the minimum confidence threshold is 70%, then only the second, third and last rules above are output, because these are the only ones generated that are strong.

Improving the efficiency of Apriori

Many variations of the Apriori algorithm have been proposed that focus on improving the efficiency of the ariginal algorithm. Several of these variations are summarized as follows:

1. Hash-based technique can be used to reduce the size of the candidate k-itemsets, Ck, for k>1. For example when scanning each transaction in the database to generate the frequent 1-itemsetes, L1, from the candidate 1-itemsets in C1, we can generate all of the 2-itemsets for each transaction, hash them into a different buckets of a hash table structure and increase the corresponding bucket counts:
    a. H(x,y)=((order of x)X10+(order of y)) mod 7
    b. A 2-itemset whose corresponding bucket count in the hash table is below the threshold cannot be frequent and thus should be removed from the candidate set.
2. Transaction reduction – a transaction that does not contain any frequent k-itemsets cannot contain any frequent k+1 itemsets. Therefore, such a transaction can be marked or removed from further consideration because subsequent scans of the database for j-itemsets, where j>k, will not require it.
3. Partitioning (partitioning the data to find candidate itemsets): A partitioning technique can be used that requires just two database scans to mine the frequent itemsets. It consists of two phases. In phase I, the algorithm subdivides the transactions of D into n non-overlapping partitions. If the minimum support threshold for transactions in D is min_sup, then the minimum support count for a partition is min_sup X the number of transactions in that partition. For each partition, all frequent itemsets within the partition are found. These are referred to as local frequent itemsets. A local frequent itemset may or may not be frequent with respect to the entire database, D. Any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of the partitions. Therefore all local frequent itemsets are candidate itemsets with respect to D. The collection of frequent itemsets from all partitions forms the global candidate itemsets with respect to D. In phase II, a second

scan of D is conducted in which the actual support of each candidate is assessed in order to determine the global frequent itemsets



4. Sampling (mining on a subset of a given data): The basic idea of the sampling approach is to pick a random sample S of the given data D, and then search for frequent itemsets in S instead of D. In this way, we trade off some degree of accuracy against efficiency. The sample size of S is such that the search for frequent itemsets in S can be done in main memory, and so only one scan of the transactions in S in required overall. Because we are searching for frequent itemsets in S rather than in D, it is possible that we will miss some of the global frequent itemsets. To lessen this possibility, we use a lower support threshold than minimum support to find the frequent itemsets local to S.
5. Dynamic itemset counting (adding candidate itemsets at different points during a scan): A dynamic itemset counting technique was proposed in which the database is partitioned into blocks marked by start points. In this variation new candidate itemsets can be added at any start point, which determines new candidate itemsets only immediately before each complete database scan. The resulting algorithm requires fewer database scan than Apriori.

Mining frequent itemsets without candidate generation

In many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However it can suffer from two nontrivial costs:

It may need to generate a huge number of candidate sets. For example if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.

It may need to repeatedly scan the database and check a large set of candidates by pattern matching.

The solution is the frequent-pattern growth, or simply FP-growth, which mines the complete set of frequent itemsets without candidate generation. This method adopts a divide-and-conquer strategy as follows: first it compresses the database representing frequent items into frequent-pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional database, each associated with one frequent item or pattern fragment, and mines each such database separately.

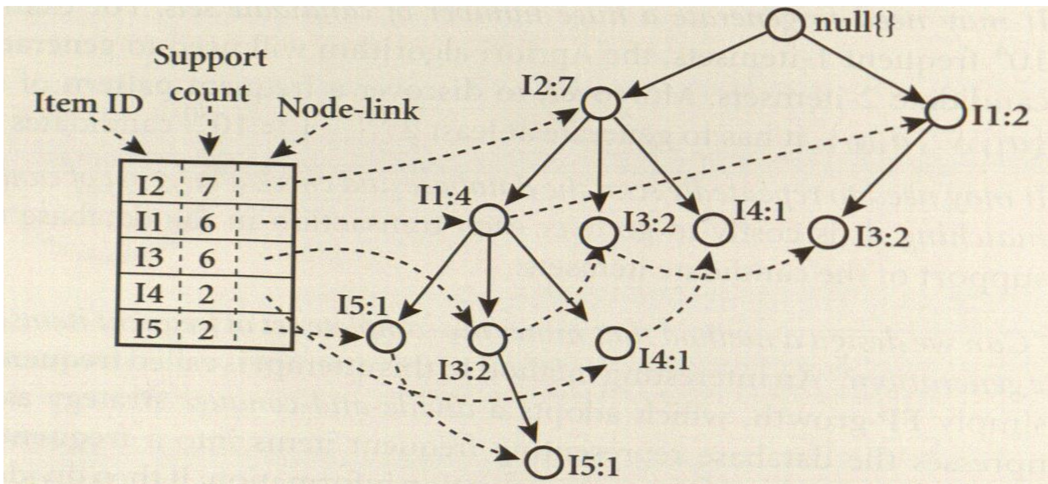
Let us create the FP-tree for the example from Table 3:

- First we scan the database and determine the set of frequent items (1-itemsets) and their support counts(frequencies): L={{I2:7},{I1:6},{I3:6},{I4:2},{I5:2}}
- Then we create the root of the FP-tree and label it with "null"
- We take each transaction, sort the items according to descending support count, and create a branch for it. For example the scan of the first transaction "T100:I1, I2, I5", which contain tree items: I2, I1 and I5 in sorted descending, leads to the construction of the first branch of the tree: (I2:1), (I1:1), (I5:1).
- The second transaction T200 contains the items I2 and I4. This would result a branch where I2 is linked to the root and I4 is linked to I2. However this branch would share a common prefix, i2, with the existing path for T100. Therefore we instead increment the count of the 12 node by 1 and create a new node (I4:1), which is linked as a child of (I2:2).

In general when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1 and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. In this way the problem of mining frequent pattern in database is transformed to that of mining the FP-tree.

The FP-tree is mined as follows: Start from each frequent length-1 pattern, as an initial suffix pattern, construct its conditional pattern base, a sub-database, which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern, then construct its conditional FP-tree and perform mining recursively on such a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.



- Let us consider I5, which is the last item in L. I5 occurs in two branches of the FP-tree:
    - (I2, I1, I5:1)
    - (I2, I1, I3, I5:1)
- I5 is a suffix, so its corresponding two prefix paths are
    - (I2, I1:1)
    - (I2, I1, I3:1)
- Its conditional FP-tree contains only a single path: (I2:2, I1:2); I3 is removed because its support count of 1 is less than the minimum support count
- The single path generates all the combinations of frequent patterns:
    - {I2,I5:2}
    - {I1,I5:2}

    - {I2, I1, I5:2}
- For I4 exist 2 prefix path, which form the conditional pattern base:
    - {{I2, I1:1},{I2:1}}
- This generates a single-node conditional FP-tree:
    - (I2:2)
- The frequent pattern: {I2, I1:2}

The following table shows the frequent pattern generated for each node:

Item	Conditional Pattern Base	Conditional FP-tree	Frequent Pattern Generated
I5	{{I2, I1:1}, {I2, I1, I3:1}}	(I2:2, I1:2)	{I2, I5:2}, {I1, I5:2}, {I2, I1, I5:2}
I4	{{I2, I1:2}, {I2:1}}	(I2:2)	{I2, I4:2}
I3	{{I2, I1:2}, {I2:2}, {I1:2}}	(I2:4, I1:2), (I1:2), (I2:4)	{I2, I3:4}, {I1, I3:4}, {I2, I1, I3:2}, {I2, I1:4}
I1	{{I2:4}}	(I2:4)	{I2, I1:4}

The FP-growth method transforms the problem of finding long frequent patterns to searching for shorter ones recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs.

The FP-growth algorithm: mine frequent itemsets using an FP-tree by pattern fragment growth.
Input:
- D, a transaction database
- min_sup, the minimum support count threshold

Output: the complete set of frequent patterns.
Method:
(1) the FP-tree is constructed
(2) The FP-tree is mined by calling FP-growth(FP_tree, null):

```
procedure FP_growth(Tree, α)
    if Tree contains a single path P then
        for each combination (denoted as β) of the nodes in the path P
            generate pattern βUα with support_count = minimum support count of nodes in β;
    else for each a_i in the header of Tree{
        generate pattern β=aiUα with support_count = a_i.support_count
        construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
        if Tree_β != 0 then
            call FP_growth(Tree_β, β); }
```

Mining frequent itemsets using vertical data format

The Apriori and the FP-growth methods mine frequent patterns from a set of transactions in TID-itemset format, where TIS is a transaction id and itemset is the set of items bought in transaction TID. This data format is known as horizontal data format. Alternatively data can also be presented in item-TID_set format, where item is an item name and TID_set is the set of transaction identifiers containing the item. This format is known as vertical data format. In the following table you can see the vertical data format of the example, shown in table 3:

Itemset	TID_set
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

The 2-itemsets in vertical data format:

Itemset	TID_set
{I1, I2}	{T100, T400, T800, T900}
{I1, I3}	{T300, T700, T800, T900}
{I1, I4}	{T400}
{I1, I5}	{T100, T800}
{I2, I3}	{T300, T600, T800, T900}
{I2, I4}	{T200, T400}
{I2, I5}	{T100, T800}
{I3, I5}	{T800}

The 3itemsets in vertical data format:

Itemset	TID_set

{I1, I2, I3}	{T800, T900}
{I1, I2, I5}	{T100, T800}

First we transform the horizontally formatted data to the vertical format by scanning the data set once. The support count of an itemset is simply the length of the TID_set of the itemset. Starting with k=1 the frequent k-itemsets can be used to construct the candidate (k+1)-itemsets. This process repeats with k incremented by 1 each time until no frequent itemsets or no candidate itemsets can be found.

Advantages of this algorithm:

- Better than Apriori in the generation of candidate (k+1)-itemset from frequent k-itemsets
- There is no need to scan the database to find the support (k+1) itemsets (for k>=1). This is because the TID_set of each k-itemset carries the complete information required for counting such support.

The disadvantage of this algorithm consist in the TID_set being to long, taking substantial memory space as well as computation time for intersecting the long sets.

Conclusions

The major problem with frequent set mining methods presented previews is the explosion of the number of results, it is difficult to find the most interesting frequent item sets. We are facing the following disadvantages: many transactions, huge database, many data and not enough information. Large sets of frequent item sets describe essentially the same set of transactions. This problem was approached in the paper 'Item Sets That Compress', It uses the MDL principle to reduce the number of the item sets: the best set of frequent item sets is that set that compresses the database set. The following four heuristic algorithms give a dramatic reduction in the number of frequent item sets:

1. Naive compression
2. Naive compression & Pruning
3. Naive compression + Sanitize
4. Naive compression & Pruning + Sanitize

The experiments were made on both the closed and all frequent item sets for min_sup = 724 on the mushroom database, which was taken from the FIMI web site.
The results on the closed frequent item sets were impressive; they ended up with less than 2% of the closed frequent item sets. The results for all frequent items sets were even more impressive with 0,035%. This small subset gives a much better compression than the one constructed from closed frequent item sets.

A special case of the frequent item set mining problem is the frequent string mining, where the main goal is to find all substrings of a collection of string databases which satisfy database specific minimum and maximum frequency constraints. An algorithm was presented in the paper “Optimal string mining under frequency constraint” from J. Fischer, which solves the frequent string mining problem in linear time under the assumption that the number of databases is treated as a constant. The space consumption of this algorithm is proportional to the total size of all databases. Adrian Kügel and Enno Ohlebusch improved this algorithm in such way that its space consumption is proportional to the size of the largest database and it takes linear time regardless of the number of databases. This algorithm is more flexible, because one of several databases can be replaced without having to recalculate everything; the intermediate data can be stored on file and be reused.

The problem of frequent item set mining was extended to sequential pattern mining. By the issue of finding frequent sequences we are facing with the problem of having a big number of frequent sequences and many redundancies. The article “Mining conjunctive sequential patterns” presents an algorithm for non-derivable conjunctive sequential patterns and shows its use in mining association rules for sequences. The experiments show the efficiency of this algorithm.

The next group of articles handled the data mining problem having as basis the decision tree, which is a decision support tool that uses a tree-like graph or model of decisions and their possible consequences. In data mining a decision tree is a predictive model; that is a mapping from observations about an item to conclusions about its target value. In this tree structure leaves represent classifications and branches represent conjunctions of features that lead to those classifications.

The paper “Distributed decision tree induction in peer-to-peer systems” offers a scalable and robust distributed algorithm for decision tree induction in large P2P environments. The algorithm is called PeDiT algorithm and it uses a misclassification gain as a splitting criteria and a stopping rule the depth of the tree. The optimal depth of the tree is three, a higher depth decreases the efficiency of the algorithm.

The SVM provides a new approach to the problem of pattern recognition together with regression estimation and linear operator inversion with clear connections to the underlying statistical learning theory. Advantage of the algorithm is that the SVM training always finds the global minimum and their simple geometric interpretation provides fertile ground for further investigation. The main challenge is the choice of its kernel. Different methods were worked out for training support vector machines. One of it is the sequential minimal optimization (SMO), where the main idea is to break the quadratic programming (QP) problem into a series of smallest possible QP problems, which are solved analytically. Its advantages are: the algorithm is easy to implement, the amount of memory required for the algorithm is linear in the training set size, which allows SMO to handle very large training sets, lower execution time.

The next group of papers handles the problem of community mining. The traditional bipartite model of ontologies was extended with the social dimension, leading to a tripartite model of actors, concepts and instances. We have seen a fast algorithm for finding overlapping communities in networks: CONGA and a modification of it called CONGO, which is more efficient and faster than the first one. The next algorithm presented was the context-specific cluster tree (CCT) for community exploration on large bipartite graphs. The resulting CCT can provide a compressed representation of the graph and facilitate visualization. The experiments showed that both space and computational efficiency are achieved in several large real graphs.

Data mining is about analyzing data; for information about extracting information out of data. It is a very actual and interesting issue having more and more data stored in database. The most important usage: customer segmentation in marketing, shopping cart analyzes, management of customer relationship, campaign management, Web usage mining, text mining, player tracking and so on.

References

Jiawei Han, Micheline Kamber. Data Mining – Concepts and Techniques. Morgan Kaufmann, 2 edition, 2006.

Agrawal R, Imielinski T and Swami A. Mining association rules between sets of items in large databases. In Buneman P. and Jajodia S., editors, Processing of the 1993 ACM SIGMOD International Conference on Management of Data

Adrian Kügel and Enno Ohlebusch. A space efficient solution to the frequent string mining problem for many databases. Data mining knowledge discovery, 2008.

Manila, H. Local and global methods in Data Mining: Basic techniques and open problems. In Widmayer, P., Ruiz, F., Morales, R., Hennessy, M., Eidenbenz, S., and Conejo, R., editors, Proceedings of the 29th International Colloquium on Automata, Languages and Programming.

Han, J., Pei, J., Yin, Y., and Mao, R Mining frequent pattern without candidate generation. A frequent-tree approach. Data Mining and Knowledge discovery, 2004