Efficient Mining of Frequent Sequence Generators

Chuancong Gao, Jianyong Wang, Yukai He, Lizhu Zhou
Tsinghua University, Beijing, 100084, P.R.China
{\{gaocc07, heyk05\}@mails.tsinghua.edu.cn, \{jianyong, dcszlz\}@tsinghua.edu.cn

What is generator?
All the non-empty frequent sequences contained in the same set of input sequences form an equivalence class. The maximal sequences in an equivalence class are called closed sequences, while the minimal ones are called sequence generators.

Pruning Strategy

THEOREM 1. Given two sequences Sp1 and Sp2, if Sp1 ⊏ Sp2 and SDBSp1 = SDBSp2, then any extension to Sp2 cannot be a generator.

THEOREM 2. Given subsequence Sp = e1e2...en and an item e', if SDBSp = SDBSp(i)(i = 1, 2, ..., n), then we have SDBSp e' = SDBSp(i)e'.

LEMMA 1. (Forward Pruning). Given subsequence Sp = e1e2...en and an item e', if sup(Sp) = sup(Sp*) holds and for any frequent item u of Sp* we always have SDBSp u = SDBSp* u, then Sp* can be safely pruned.

LEMMA 2. (Backward Pruning). Given subsequence Sp = e1e2...en, if there exists an index i(i = 1, 2, ..., n - 1) and a corresponding index j(j = i + 1, i + 2, ..., n) such that SDBSp(i) = SDB(Sp(i))j, then Sp can be safely pruned.

Why is generator useful?
• The number of sequence generators is much smaller than all frequent sequences.
• Sequence generators are the shortest ones in an equivalence class, thus have a shorter or equal length to closed sequences and all frequent sequences.
• The average size of sequence generators tends to be smaller than that of closed sequences and all frequent sequences.

Problem Formulation

THEOREM 1. Given two sequences Sp1 and Sp2, if Sp1 ⊏ Sp2 and SDBSp1 = SDBSp2, then any extension to Sp2 cannot be a generator.

THEOREM 2. Given subsequence Sp = e1e2...en and an item e', if SDBSp = SDBSp(i)(i = 1, 2, ..., n), then we have SDBSp e' = SDBSp(i)e'.

LEMMA 1. (Forward Pruning). Given subsequence Sp = e1e2...en and an item e', let Sp* = Sp + e'. If sup(Sp) = sup(Sp*) holds and for any frequent item u of Sp* we always have SDBSp u = SDBSp* u, then Sp* can be safely pruned.

LEMMA 2. (Backward Pruning). Given subsequence Sp = e1e2...en, if there exists an index i(i = 1, 2, ..., n - 1) and a corresponding index j(j = i + 1, i + 2, ..., n) such that SDBSp(i) = SDB(Sp(i))j, then Sp can be safely pruned.

Generator Checking Scheme

THEOREM 3. A sequence S is a generator if and only if 3i such that 1 ≤ i ≤ n and sup(S) = sup(S(i)) holds.

We devise a generator checking scheme as shown in Theorem 3 in order to assert whether each mined frequent subsequence is a generator, and it can be done efficiently during pruning process by checking whether there exists such an index i(i = 1, 2, ..., n) that |SDBS| = |SDBS(i)|, as sup(S) = |SDBS| holds.

ACKNOWLEDGEMENTS

This work was partly supported by 973 Program under Grant No. 2006CB303103, and Program for New Century Excellent Talents in University under Grant No. NCET-07-0491, State Education Ministry of China.
Algorithm

Algorithm 1: FEAT(SDB, minSup) : FGS

Input : initial database SDB, minimum support min_sup
Output: set of frequent generators FGS
begin
1. FGS ← ∅;
2. minSup(0, SDB, min_sup, FGS);
3. return FGS;
end

Algorithm 2: minSup(Sp, SDB, min_sup, FGS)

Input : Prefix sequence Sp, Sp’s projected database SDB, minimum support min_sup, and result set FGS
begin
1. foreach i in localFrequentItems(SDB, min_sup) do
2. S′i ← <Sp∪i>;
3. SDB′i ← projectedDatabase(SDB, S′i);
4. canPrune ← false;
5. isGenerator ← true;
6. if |SDB′i| = |SDB| then
7. canPrune ← ForwardPrune(Sp, SDB, S′i, SDB);
8. isGenerator ← false;
9. if not canPrune then
10. BackwardPrune(Sp, SDB, canPrune, isGenerator);
11. if isGenerator then
12. FGS ← FGS ∪ {S′i};
13. if not canPrune then
14. FEAT(Sp, SDB, min_sup, FGS);
end

Algorithm 3: forwardPrune(Sp, SDB, SDB, min_sup) : canPrune

Input : previous prefix sequence Sp, previous projected sequence database SDB, prefix sequence Sp, projected sequence database SDB, minimum support min_sup
Output: whether Sp can be pruned, whether Sp is a generator
begin
1. foreach i in localFrequentItems(SDB, min_sup) do
2. if projectedDatabase(SDB, Sp ∪ i) = projectedDatabase(SDB, Sp ∪ {S′i}) then
3. return false;
4. return true;
end

Algorithm 4: backwardPrune(Sp, SDB, canPrune, isGenerator)

Input : prefix sequence Sp, projected sequence database SDB, whether Sp can be pruned, whether Sp is a generator
begin
1. for i in |Sp| to 1 do
2. for j ← i + 1 to |Sp| do
3. Sp ← (Sp)−{S′i};
4. SDB′i ← projectedDatabase(SDB, S′i);
5. Sp ← (Sp)∪{S′i};
6. if |SDB′i| = |SDB| then
7. isGenerator ← false;
8. if SDB′i = SDB then
9. canPrune ← true;
10. return;
end

Runtime Evaluation

Tests were finished on the Gazelle data set.

Tests were finished on the Program Trace data set.

Applications

We used generators and all frequent sequence as features to build SVM and Naïve Bayesian classifiers respectively to try to classify Amazon production reviews as (+1) Like or (-1) Dislike. The results for Office07Review dataset show that both generator-based and all frequent sequence-based models achieve almost the same accuracy. With a minimum support of 2% and a minimum confidence of 75%, both generator-based and all frequent sequence-based Naïve Bayesian classifiers can achieve the same best accuracy of 80.6%. As generator-based approach is more efficient, it has an edge over all frequent sequence-based approach in terms of efficiency.

Used Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th># seqs.</th>
<th># items</th>
<th>Avg. len.</th>
<th>Max. len.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gazelle</td>
<td>29369</td>
<td>1423</td>
<td>3</td>
<td>651</td>
</tr>
<tr>
<td>Program Trace</td>
<td>10</td>
<td>105</td>
<td>488</td>
<td>989</td>
</tr>
<tr>
<td>Office07Review</td>
<td>320</td>
<td>240</td>
<td>80</td>
<td>94</td>
</tr>
</tbody>
</table>

Scalability