HW3

February 25, 2019

Erin Witmer CSC 440
Homework 3

0.1 Chapter 6

[6.1] Suppose you have the set C of all frequent closed itemsets on a data set D, as well as the support count for each frequen closed itemset. Describe an algorithm to determine whether a given itemset X is frequent or not, and the support of X if it is frequent. An itemset X is closed in set D if there exists no superset Y such than Y has the same support count as X in D. Per the text: "The set of closed frequent itemsets contains complete information regarding the frequent itemsets." A simple algorithm to determine whether X is frequent or not and the support is:

```
sort set C by support count (high to low)

for each itemset (i) in C:

    if X is a subset of C(i):
       return support count for this item

    else:
       next item

return X is not frequent if X is not a subset of any itemset in C
```

We know that the support count of X if it is a subset of the itemset in C is the same as the support count in that itemset based on the definition of the closed itemset. We know it is frequent because C is a set of all frequent closed itemsets. We know that if X is not a subset of any itemset in C, it is not frequent.

[6.3] The Apriori algorithm makes use of prior knowledge of subset support properties.

(a) Prove that all nonempty subsets of a frequent itemset must also be frequent. An itemset (I) is a frequent itemset if the percentage of transactions (T) in a database (D) containing that itemset (I) meets or exceeds a minimum support threshold (min_sup). That is:

$$\frac{countOfTransactionsContaining(I)}{totalTransactionsIn(D)} \geq minSupport$$

if

$$A \subset I$$

then the count of transactions containing A is greater than or equal to the count of transactions containing I, because every transaction containing I, will also contain A. This is true for all non-empty subsets of I. Support of an itemset never exceeds the support of its subsets. This is known as the anti-monotone property of support. The denominator, the count of all transactions, does not change. So if the numerator (count of transactions containing A) is greater than or equal to the count of transactions containing I, and the denominator is constant, then the support level will be greater than or equal to the support level of I. If I is frequent, or higher than the min support level, then A will also be frequent.

(b) Prove that the support of any nonempty subset s of itemset s must be at least as great as the support of s. Again, this can be proven based on the anti-monotone property of support. Support of an itemset never exceeds the support of its subsets. The support count of s is the number of times an itemset (s) appears in the universe of all transactions. For example, if s = {beer, nuts, diapers} then the support count of s is the number of transactions which contain all three of those items. Any subset of s, for example, s' = {beer, nuts} will occur in all transactions in which s occurs. In other words, any transaction which contains {beer, nuts, diapers} will also contain all subsets of that itemset {beer, nuts}. The support count of s' will therefore be at least as high as s, and may be higher if the subset s' occurs in additional transactions.

(c) Given frequent itemset l and subset s of l, prove that the confidence of the rule "s ->(ls)" cannot be more than the confidence of "s->(ls)," where s is a subset of s.

$$confidence(s' -> l - s') = \frac{supportCount(s'(l - s'))}{supportCount(s')}$$

$$confidence(s -> l - s) = \frac{supportCount(s(l - s))}{supportCount(s)}$$

Since we know s' is a subset of s and therefore a subset of l, l-s' represents the complement to the frequent itemset, l. In union with s', it equals the frequent itemset. So the count of s' (l-s') will be the support count of l. Similarly, l-s represents the complement to the frequent itemset, and the count of s (l-s) will be the count of l. So we know the numerators in the equations above are the same. As proved above, the support count of s' as a subset of s will always be greater than or equal to the support count of s. So the denominator in the first equation above will always be greater than or equal to the denominator in the second equation, and the value of the first equation will always be less than or equal to the value of the second equation.

(d) A partitioning variation of Apriori subdivides the transactions of a database D into n nonoverlapping partitions. Prove that any itemset that is frequent in D must be frequent in at least one partition of D. If a given itemset is frequent in D, then it must be locally frequent in at least one nonoverlapping partition of D. The itemset is frequent in D if:

$$\frac{supportCount(I)}{totalTransactionsIn(D)} \geq minSupport$$

If the database is divided up, at least one of the partitions will have to meet the minimum support for that local partition because if no partition did, the global support count needed would

not be reached. For example, if there are 100 transactions, and the min_support is 60%, then there must be 60 instances of the itemset in the database. If this was partitioned into 5 sets of 20 transactions, then the min_support count would be 12 for each partition. If C1...C5 < 12, then the total support count would be < 60, and the itemset would not be frequent. Therefore, at least one of the partitions needs to be locally frequent if the itemset is frequent.

[6.4] Let c be a candidate itemset in Ck generated by the Apriori algorithm. How many length-(k 1) subsets do we need to check in the prune step? Per your previous answer, can you give an improved version of procedure has infrequent subset in Figure 6.4? For every candidate C(k) generated, we need to check k choose k-1 subsets, which is equal to k subsets.

$$\binom{k}{k-1} = k$$

For example, if C(k) = {I1, I2, I3}, we need to check 3 choose 2, or 3 itemsets: {I1, I2}, {I1, I3}, {I2, I3}. One way to improve the procedure `has_infrequent_subset()` is to assume that the items which constructed the candidate set are frequent, so we don't need to check them. So for example, (k-1) frequent itemsets ABC + ABD = ABCD (Candidate of k length). Pruning combinations to check are ABC, ABD, BCD, ACD but it was constructed from ABC, ABD, so those are assumed frequent, you only have to check BCD and ACD or (k-2) sets.

[6.5] Section 6.2.2 describes a method for generating association rules from frequent itemsets. Propose a more efficient method. Explain why it is more efficient than the one proposed there. (Hint: Consider incorporating the properties of Exercises 6.3(b), (c) into your design.) The section proposes the following method for generating association rules from frequent itemsets: - 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) min conf, where min conf is the minimum confidence threshold.

While in general, confidence does not have an anti-monotone property c(ABC D) can be larger or smaller than c(AB D); confidence of rules generated from the same itemset do have an anti-monotone property. So the example provided in the text: frequent itemset X = {I1, I2, I5}, the resulting association rules:

Rule generated	Confidence
{I1, I2} -> I5	confidence = 2/4 = 50%
{I1, I5} -> I2	confidence = 2/2 = 100%
{I2, I5} -> I1	confidence = 2/2 = 100%
I1 -> {I2, I5}	confidence = 2/6 = 33%
I2 -> {I1, I5}	confidence = 2/7 = 29%
I5 -> {I1, I2}	confidence = 2/2 = 100%

The min confidence is 70%. The first rule generated {I1, I2} -> I5 doesn't meet that threshold. Since confidence is anti-monotone with respect to the right side of the rule, we know that any of the rules containing I5 on the right side will not meet the min confidence threshold either. So we can prune the two rules that contain I5 on the right: I1 -> {I2, I5} and I2 -> {I1, I5} as a result of the anti-monotone property without any calculations.

[6.6] A database has five transactions. Let min sup = 60% and min conf = 80%.

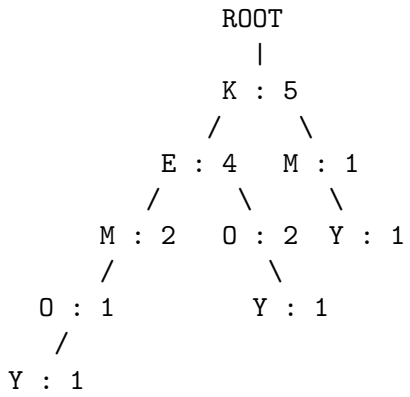
TID	item IDs
T100:	['M','O','N','K','E','Y']
T200:	['D','O','N','K','E','Y']
T300:	['M','A','K','E']
T400:	['M','U','C','K','Y']
T500:	['C','O','O','K','I','E']

(a) Find all frequent itemsets using Apriori and FP-growth, respectively. Compare the efficiency of the two mining processes. Apriori (frequent itemsets shown in bold): - Count C1:
{'M': 3, 'O': 3, 'N': 2, 'K': 5, 'E': 4, 'Y': 3, 'D': 1, 'A': 1, 'U': 1, 'C': 2, 'I': 1}

- Trim infrequent items from C1 to obtain L1:
  L1: {'M': 3, 'O': 3, 'K': 5, 'E': 4, 'Y': 3}
- Generate C2:
  [('E', 'K'), ('E', 'M'), ('E', 'O'), ('E', 'Y'), ('K', 'M'), ('K', 'O'), ('K', 'Y'), ('M', 'O'), ('M', 'Y'), ('O', 'Y')]
- Count C2:
  {('E', 'K'): 4, ('E', 'M'): 2, ('E', 'O'): 3, ('E', 'Y'): 2, ('K', 'M'): 3, ('K', 'O'): 3, ('K', 'Y'): 3, ('M', 'O'): 1, ('M', 'Y'): 2, ('O', 'Y'): 2}
- Trim infrequent items from C2 to obtain L2:
  L2: {('E', 'K'): 4, ('E', 'O'): 3, ('K', 'M'): 3, ('K', 'O'): 3, ('K', 'Y'): 3}
- Generate C3:
  [('E', 'K', 'O'), ('K', 'M', 'O'), ('K', 'M', 'Y'), ('K', 'O', 'Y')]
- Prune C3 based on L2 (infrequent subsets):
  [('E', 'K', 'O')]
- Count C3:
  {('E', 'K', 'O'): 3}
- Trim infrequent to obtain L3:
  L3: {('E', 'K', 'O'): 3}

FPGrowth:

- Count C1:
  {'M': 3, 'O': 3, 'N': 2, 'K': 5, 'E': 4, 'Y': 3, 'D': 1, 'A': 1, 'U': 1, 'C': 2, 'I': 1}
- Trim infrequent items from C1 to obtain L1 and sort:
  L1: {'K': 5, 'E': 4, 'M': 3, 'O': 3, 'Y': 3}
- Sort transactions in L order and build FPTree:



- Construct conditional pattern base from the tree
- From this build a conditional FPTree for each item, eliminating non-frequent paths.
- From this sub tree, generate frequent patterns.

Item	Conditional Pattern Base	Conditional FPtree	Frequent Patterns Generated
Y	{K,E,M,O}: 1, {K,E,O}: 1, {K,M}: 1	{K: 3}	{KY}: 3
O	{K, E, M}: 1, {K, E}: 2	{K: 3, E: 3}	{KO}: 3, {EO}: 3, {KEO}: 3
M	{K, E}: 2, {K}: 1	{K: 3}	{KM}: 3
E	{K}: 4	{K: 4}	{KE}: 3

The FPGrowth process is more efficient. Two major costs associated with the Apriori algorithm are: 1.) the need to generate a huge number of candidate sets if L1 is long and 2.) the need to scan the database repeatedly and check a large set of candidates by pattern matching. The FPGrowth algorithms finds frequent itemsets without candidate generation. It transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively and then concatenating the suffix. It uses the least frequent suffix first, substantially reducing the search cost.

(b) List all the strong association rules (with support s and confidence c) matching the following metarule, where X is a variable representing customers, and item i denotes variables representing items (e.g., "A," "B,"):

$$xtransaction, buys(X, item1) buys(X, item2) -> buys(X, item3) [s, c]$$

The following rules are generated. The two transactions shown in bold match the metarule.

Rule generated	Support	Confidence
{E} -> {K}	support = 4/5 = 80%	confidence = 4/4 = 100%
{K} -> {E}	support = 4/5 = 100%	confidence = 4/5 = 80%
{O} -> {E}	support = 3/5 = 60%	confidence = 3/3 = 100%

Rule generated	Support	Confidence
{M} -> {K}	support = 3/5 = 60%	confidence = 3/3 = 100%
{O} -> {K}	support = 3/5 = 60%	confidence = 3/3 = 100%
{Y} -> {K}	support = 3/5 = 60%	confidence = 3/3 = 100%
{E,O} -> {K}	**support = 3/5 = 60%**	**confidence = 3/3 = 100%**
{K,O} -> {E}	**support = 3/5 = 60%**	**confidence = 3/3 = 100%**
{O} -> {K, E}	support = 3/5 = 60%	confidence = 3/3 = 100%

[6.11] Most frequent pattern mining algorithms consider only distinct items in a transaction. However, multiple occurrences of an item in the same shopping basket, such as four cakes and three jugs of milk, can be important in transactional data analysis. How can one mine frequent itemsets efficiently considering multiple occurrences of items? Propose modifications to the well-known algorithms, such as Apriori and FP-growth, to adapt to such a situation. A simple way to handle this in both the Apriori and FP-growth algorithm is to treat each instance of a duplicate item as a unique item. So rather than compressing a transaction with 4 cakes and 3 jugs of milk to {cake, milk} the transaction would be: {cake_1, cake_2, cake_3, cake_4, milk_1, milk_2, milk_3}. The benefit of this analysis would be you could analyze association rules like, if you buy 2 cartons of eggs, how likely are you to buy 2 cartons of milk? The problem with this is that it dramatically increases the computational complexity of the analysis because the number of unique items increases. Given d itemsets, the total number of itemsets = 2^d, and the total possible association rules = 3^d + 2^(d+1) + 1. So any increase in d will increase the computational complexity of the analysis exponentially. In this situation, counting using hash structure would greatly improve the efficiency of this analysis. It would also be wise to use FPGrowth over Apriori since L1 would likely increase significantly.