Data Mining Homework 1 Report

Introduction

Finding patterns in large, unstructured data such as a transaction database might at first seem too computationally expensive to perform. Our goal is to find items that occur often together and make inferences about them. In the end, those subsets which appear more than a certain threshold (called its **support**) are considered frequent itemsets. From frequent itemsets we can generate **association rules** by calculating how often the subsets of frequent itemsets occur together.

If we wanted to find frequent patterns in a set of transactions, a naive approach would be to go through each transaction of the database, and count the appearance of all the subsets in each transaction. Such an approach would be prohibitive in terms of memory and time. It also misses a key insight: all subsets of a frequent itemset are themselves frequent. This is called the **a-priori** property, and it is the basis of the two algorithms compared in this report. Using this property, if at any point we encounter a set that is not frequent, we know that all its supersets are also not frequent.

After frequent itemsets are calcualted, association rules are generated which measure the chance of two sets of items occurring together. For instance, the rule A ⇒B means that whenever A occurs, B also occurs with a certain probability p, called the rule's **confidence**.

To generate association rules from frequent itemsets, various algorithms have been proposed. The focus of this report is comparing the **a-priori** algorithm and the **FP-Growth** algorithm.

A-priori Algorithm

The a-priori algorithm uses the a-priori property to build up the frequent itemsets. First, the 1-itemsets are found by scanning the dataset D and getting the support count for each item; only the items with high enough support are kept.

Next, until the frequent itemsets L is empty, we keep generating the new candidate set C_{k+1}. From the set of frequent itemsets L_k, we combine each element l_1 with an element from l_2 if l_2's last element is greater than l_1's last element. Having this requirement preserves the lexicographical order of the itemsets.

We maintain a structure called a **hash tree**, whose leaf nodes contain the itemsets and their counts and the internal nodes contain the hashes of the children nodes. To insert a new item, we check the hash of a certain index of the item, then insert the item to the corresponding bucket in the node.

When the length of a bucket exceeds a certain amount, we have to split the items in the node's buckets into different children corresponding to its hash value. By recursively inserting items and splitting the nodes, eventually by following the hash of each index of the itemset we reach the leaf node that contains its count or where it should be inserted. The hash tree structure allows us to maintain a count and insert items in a more efficient way.

To get the frequent itemsets, we check all the leaf nodes in the hash table that contain a support count greater than the minimum support. The frequent itemsets are the k-itemsets L_k. We repeat this process until there are no more new frequent itemsets, or $L_k = \emptyset$.

FP-Growth Algorithm

The FP-Growth algorithm addresses some of the potential drawbacks of the a-priori algorithm. The first algorithm requires several scans of the database to determine the support count of datasets. Furthermore, for large databases the space required to store the hash tree might be prohibitive as well.

The FP-Growth algorithm uses a compressed form of the database to determine support counts for the itemsets, and avoids the candidate generation approach which might produce a large amount of itemsets without enough support. First we determine the support for the 1-itemsets in the same way as in a-priori: by scanning the database and maintaining the support count of each element in the transactions.

After we determine L_1, we proceed to build the **fp-tree** as follows. First, we loop again through the dataset, sort each transaction in decreasing order according to its support count in L_1 and insert them into the tree. If the item at a given index in the transaction is present in the node's children, we increase its count and recurse on that node, checking increasing the index. In the end, we are left with a tree of the most common items close to the root and less common items closer to the leaves. Since there might be more than one node with the item "1" for example, we keep a table where we maintain a reference to all the nodes for "1", as well as for all the other items.

The header table becomes useful in the next step which involves building the **conditional pattern bases**. For each item, e.g. "1", we build the set of all the paths from these nodes to the roots. We then make smaller fp-tree, called the **conditional fp-tree**, for each element and insert the items in the conditional pattern base to the conditional fp-tree, and maintain the support count. Once we build the conditional pattern tree for an item i, we prune the nodes who do not meet the minimal support count. The remaining items in the tree are the items that comprise the frequent itemsets involving item i.

The overall set of frequent itemsets is thus the concatenation of the frequent itemsets for each item i.

Experiment

To test the differnce between the two algorithms, we test several transitional datasets and compare the execution time and results. There are a couple of datasets we are using: a 9-item file with the textbook example, used to check for correct itemset generation; a 100-item `.data` file produced by

the IBM dataset generator; and the provided `ibm-5000.txt` file. The last set of 25,000 items is found in the `ibm-2021.txt` file.

The results are the average of three runs. The testing results for the a-priori with a support count of 2 and rule confidence of 0.8 algorithm are as follows

Trans. No.	Memory Usage(Mb)	Time(s)
9	12.993	.000185
100	12.902	.0024
4798	17.529	.2013
25,000	39.165	3.4646

For the FP-Growth, the corresponding tests yield the following results

Trans. No.	Memory Usage(Mb)	Time(s)
9	12.998	.0003
100	13.418	.0056
4798	18.388	.0046
25,000	38.997	.4982

Performance Analysis

The datasets can also affect the performance of the two algorithms. For instance, the 100-item dataset used an average transaction length of 3 items, however the number of customers was much higher than the number of transactions. This lead to overall lower confidence in each individual rule and a fewer amount of rules in total, e.g. customer 1000 would have only a few appearances in a smaller dataset. In a realistic setting adjusting the support and confidence requirements in accordance with the dataset yield the best results; however, for this experiment those settings were left constant for all tests. Having a higher minimum support and/or confidence requirement would increase the computation speed since it would result in fewer frequent itemsets, however this could come at the cost of missing potentially useful relationships. Thus, apart from using a more efficient algorithm, other parameters could be adjusted for optimal results.

The motivation for the FP-Growth algorithm involved reducing the number of database scans to also

reduce the computational complexity. This advantage becomes clearer the larger the amount of transactions in the dataset. Even though the memory requirements remained close (possibly a result of the implementation), the time taken to generate frequent patterns and rules in the FP-Growth algorithm is much lower than a-priori for larger datasets. Again, avoiding costly database scans in favor of repeated tree scans can certainly help depending on the structure of the conditional pattern and conditional growth trees.

In terms of memory, both implementations required a similar amount of memory during the experiments. While some of the reason might lie in the implementation details(e.g. built-in structures used for some steps), having a large hashtree or fp-tree would also contribute to the memory requirements. Doing a very quick calculation, for a file containing around $100,000$ unique items (such as `ibm-2021.txt`), 20% of the items being considered frequent, each frequent item appearing on average 14 times [^1], times 64-bits per number and the result is a tree of estimated size

$$100000 \times 0.20 \times 14 \times 64 = 17.920\text{Mb} \tag{1}$$

Although with a low support threshold, it is likely that a larger portion of the dataset is considered "frequent", which would lead to a much larger tree. This example is nevertheless useful in seeing how repeated items in our tree could lead to a bigger tree size.

Support and Confidence

Finding association rules $A \Rightarrow B$ from the set of frequent itemsets F_K requires finding the support of AB. The formula for calculating the confidence in a given rule is

$$confidence(A \Rightarrow B) = \frac{support(A \cup B)}{support(A)} \tag{2}$$

In the algorithms, $A \cup B$ is usually a single frequent itemset f_k, from which we derive the length k subsets that make up the right and left hand sides of the rule. Since every subset of a frequent itemset is itself frequent, we know that both A and B are frequent itemsets. However, the relationship between their individual support counts can lead to interesting results.

As the support of $A \cup B$ and A both go higher, this will lead to a high support count, but the confidence will actually be lower, possibly being rejected. If $A \cup B$ has a higher support count and A has a lower support count (i.e. their support counts do not differ that much), the support for $A \cup B$ will be higher and the confidence as well. If $A \cup B$ has lower support count, and A's support is higher relative to it, the confidence will be lower because of the larger denominator, making this rule a candidate to be dropped. Finally, as the support count of $A \cup B$ and A both tend towards lower values, the support count will be lower but the confidence could turn out to be high.

Conclusion

Determining which items occur together in a set of transactions can be quite a complex task, especially with thousands or even millions of individual transactions. Frequent patterns could help many kinds of businesses know which items should be bundled together in a store, for example, or help in creating enticing discounts. Sifting through large amounts of data to find these groups of items would quickly become prohibitive with a naive approach.

In this assignment, both the a-priori and FP-Growth algorithms were tested in terms of computation time and memory requirements. The a-priori algorithm iteratively builds a candidate item list C_k to determine the k-length itemsets L_k. It uses a hash-tree structure to store the itemsets and their counts as leaf nodes for quickly determining subsets. This approach can generate many candidates that don't result in frequent itemsets.

To address this issue, the fp-algorithm uses a compressed version of the database to determine frequent itemsets. By using an fp-tree, we can determine which items are common prefixes of item i and use those to generate frequent itemsets. This approach results in a more efficient approach since the entire database is stored only once, the computation being mostly done recursively on the fp-tree.

This assignment focused on applying them to a transactional setting; however, many other kinds of scenarios are possible where other methods might be more appropriate. Nevertheless, the usefulness of these algorithms is evident in their wide use and influence in the field. With more general methods, even better and more efficient results could be achieved from a dataset. [^1]: Taken by sorting the count of each item.