Association Rule Mining Project

Data Analytics - I

Assigned by : Prof. Vikram Pudi

Vrund Shah - 2021701025
Aditya Kumar Singh - 2021701010

Abstract

This dataset was used in KDD CUP 2000. It contains clickstream data from an e-commerce.

1. Sequence count is 77512
2. Item Count is 3340
3. Average sequence length is 4.62

”-1” is used as the delimiter and ”-2” denotes end of line/transaction.

Contents

1 FP Growth

Apriori is a known algortihm to generate frequent itemsets for a given data and minimum support. But it has two disadvantages :

1. It generates large number of candidate itemsets. For example , if there are 10^4 1-frequent itemsets , then *Apriori-Gen* will generate more than 10^7 2-frequent itemsets.
2. It needs several passes over the database and check large number of candidates by pattern matching.

FP-Growth algorithm overcomes both these disadvantages by employing a divide-and-conquer strategy to compute frequent itemsets.

FP-Growth algorithm is as follows :

1. The database is compressed to form a FP-Tree which retains the itemset association information.
2. From this FP-Tree, for each distinct item , a Conditional Pattern database is generated which is then used to generate Conditional FP-Tree via recursive calls.
3. Mining each of these Conditional FP-Trees gives the list of frequent itemsets.

Creating conditional pattern-bases for each "pattern fragment" or "itemset" , reduces the size of search space drastically. It contains the set of all prefix paths for that "itemset" in the FP-Tree.

Also, this algorithm requires only two passes over the data-base unlike multiple passes in Apriori. First pass is same as Apriori - to generate 1-frequent itemsets and the second pass to create the FP-Tree.

1.1 Bottom-Up Technique

Once we have all the 1-frequent itemsets after the first pass (say list L, which is sorted in descending order of support count of itemsets) ; we start from the least frequent itemset in L i.e bottom of the list and find all its frequent itemsets using the above mentioned algorithm and then move upwards one itemset at a time. By the time we have mined the FP-Trees for last "length(L) - 1" items, we already have generated all the desired frequent itemsets for the given data-base.

The most-frequent itemset will be directly linked to the root via a single link, so the won't be any prefix paths , hence no Conditional-Pattern base.

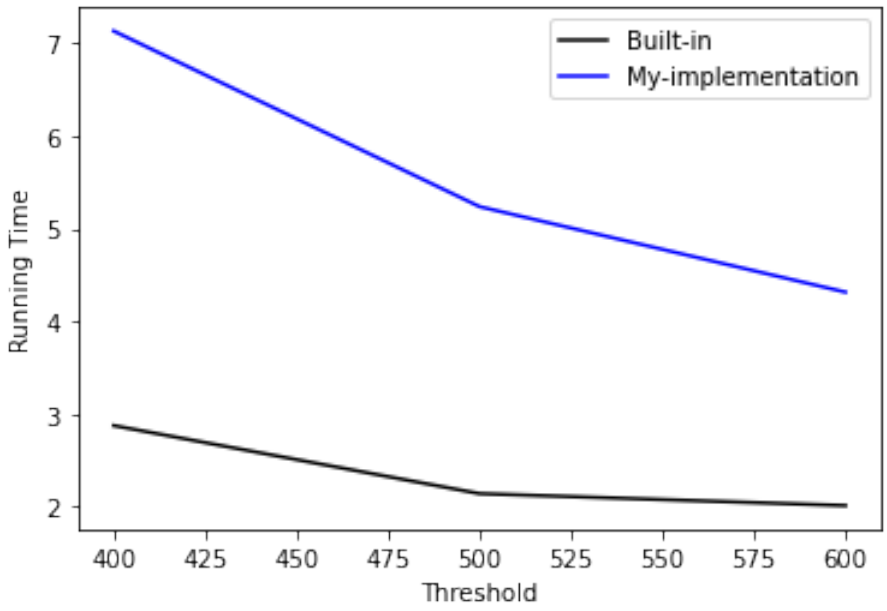
1.2 Merging Strategy

Consider a prefix path from root as follows : $A \longrightarrow B \longrightarrow C \longrightarrow D$. When we determine prefix path for "D" , we get A, B, C , which is added to Conditional Pattern Base of "D". This same path will be traversed when we find prefix path of "C". This consumes both space and time.

Optimization strategy adopted is that once we add A, B, C to conditional base of "D", we push this to Conditional pattern base of "C" and delete this list which frees up space and saves time of going through the same prefix path again for "C".

Comparison of running times of our implementation and that of standard library :

Minimum support was varied as $400, 500, 600$



Our implementation uses two improvisations namely Bottom-Up technique and Merging Strategy. Built-in functions might be using even more optimised implementations , hence the running time difference.

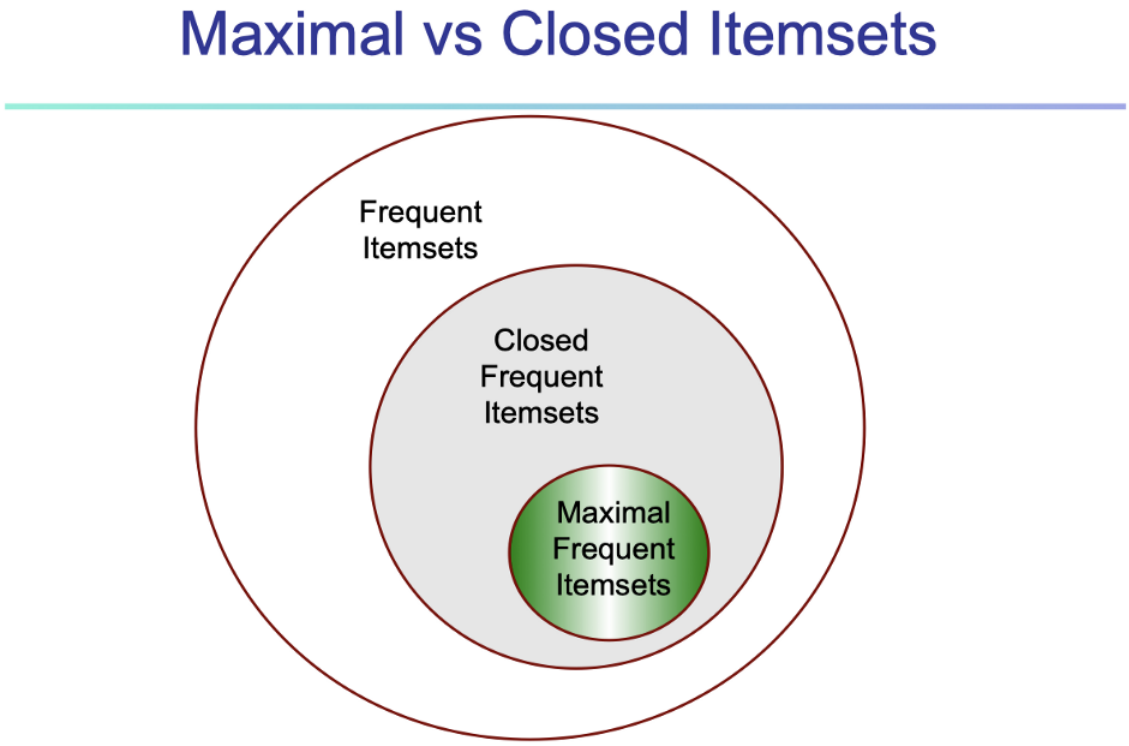
Also, as we increase the minimum support, the length of list L reduces , which inturn reduces the size of the FP-Tree making it easier to mine.

Observation : Depth of FP-Tree is inversely proportional to time required to mine it

For very-high values of minimum support , i.e trees with smaller depth, not much difference was found in running times.

1.3 Closed and Maximal Frequent Itemsets

Definition : An itemset X is closed in a data set D if there exists no proper super-itemset Y such that Y has the same support count as X in D.
An itemset X is a closed frequent itemset in set D if X is both closed and frequent in D.
An itemset X is a maximal frequent itemset (or max-itemset) in a data set D if X is frequent, and there exists no super-itemset Y such that $X \subset Y$ and Y is frequent in D.



Algorithm to mine closed frequent from frequent itemsets :

1. from the list of frequent itemsets, create a dictionary with support count as ”key” and itemsets with that support count as ”value”.
2. scan through the list of frequent itemsets and for each value of support encountered , extract all items from the dictionary created in step-1 with that as key value and scan through the list obtained. If we find Y such that for the given record X , $X \subset Y$, then move to next row in database. If we don't find any such Y, then X is closed frequent.

1.4 How to choose minimum support

Given the details of data-base :

1. Sequence count is 77512
2. Item Count is 3340
3. Average sequence length is 4.62

Formula :
Average count of each item = Seq, count * Avg. seq. / Item cpunt

$\therefore$ Avg. count = 110 This is the case when each item has equal probability of occurrence in the dataset, which is very unlikely to happen in real-world data. We start from this extreme as our minimum support and gradually , increase our minimum support value.

NOTE : refer code for below given information

1. min-sup = 110 : Frequent itemsets are 9180 and Closed Frequent are 9143
2. min-sup = 250 : Frequent itemsets are 1141 and Closed Frequent are 1141 (both equal)

2 Apriori

It was first proposed by **R. Agrawal** and **R. Srikant** in 1994 for mining frequent itemsets for Boolean association rules [AS94b]. The name has come from the fact that the algorithm uses prior knowledge of frequent itemset properties, which we shall exhibit later. This algorithm employs an iterative approach called **level-wise search**, where k-itemsets are used to explore $(k+1)$-itemsets.

- First, the frequent set of 1-itemsets is found by scanning the database to accumulate the count for each item, and their count must satisfy minimum support. The resulting set is denoted by L_1 .
- Next, L_1 is used to find L_2 , the set of frequent 2-itemsets, which again is used to find L_3, and so on, until no more frequent k-itemsets can be found. Construction of each L_k demands one full scan of the database.
- Now, to improve the efficiency for this level-wise generation of frequent itemsets, an important principle called the **Apriori principle** is used to reduce the search space.
  Apriori Principle: *If an itemset is frequent, then all of its subsets must also be frequent.*
- Again, if we go on the same line but a bit reverse then there is something called **anti-monotonicity property**, which states *if a set cannot be frequent, then all of its supersets can't be as well.*

Basically, *Apriori* consists of two main steps:

- **Join step**: To find L_k , a set of candidate k-itemsets is constructed by joining L_{k1} with itself. This set of candidates is denoted C_k .
- **Prune step**: Each candidate in C k is a potential candiadte for L_k after it passes the frequency test (i.e., candidates having count $\geq$ minimum support count are *frequent* by definition, and therefore belong to L_k). However, C_k can be huge, which incurs heavy computation. And to reduce the size of Ck, the **Apriori principle** comes in handy. Applying that we can conclude, if any $(k1)$-subset of a candidate k-itemset is not in L_{k1} , then the candidate cannot be frequent either and hence can be removed from C_k .

Further sin Below is the *algorithm* for *Apriori* (picture taken from *Data Mining. Concepts and Techniques, 3rd Edition (The Morgan Kaufmann Series in Data Management Systems)*

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- min_sup, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck | c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;

procedure apriori_gen(Lk−1:frequent (k − 1)-itemsets)
(1)   for each itemset l1 ∈ Lk−1
(2)       for each itemset l2 ∈ Lk−1
(3)           if (l1[1] = l2[1]) ∧ (l1[2] = l2[2])
                  ∧... ∧ (l1[k − 2] = l2[k − 2]) ∧ (l1[k − 1] < l2[k − 1]) then {
(4)               c = l1 ⋈ l2; // join step: generate candidates
(5)               if has_infrequent_subset(c, Lk−1) then
(6)                   delete c; // prune step: remove unfruitful candidate
(7)               else add c to Ck;
(8)           }
(9)   return Ck;

procedure has_infrequent_subset(c: candidate k-itemset;
          Lk−1: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)       if s ∉ Lk−1 then
(3)           return TRUE;
(4)   return FALSE;
```

Figure 1: Aprior Algorithm

2.1 Improving Apriori

We have used two methods (*as asked to do*), namely, **Partition Algorithm** and **Hash Based Technique**.

- **Partition Algo**: Here we apply the **basic apriori** as discused above to generate all level of frequent itemsets, but for each **partition**. Now taking *union* of all those frequent itemsets we construct a set of all possible candidates which is to be verified if they are frequent or not by doing a second scan of database. And also the min_sup gets changed for each partition according to the number of transactions in each partition.
  The basic underlying idea is as follows:
  A **local frequent itemset** may or may not be frequent with respect to the entire database, **D**. However, 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** and hence collected to form the global candidate

itemsets with respect to **D**. In second scan of **D** is conducted in which the actual support of each candidate is assessed to determine the global frequent itemsets.

- **Hash-based technique**: Here we have implemented a hash function as follows:

$$\text{hash\_key} = \text{Index(itemSet)} \mod 7$$

Here we would have 7 buckets whose keys are basically counts of itemsets. Those buckets with keys < min_sup are discarded that substantially reduce the number of candidate k-itemsets examined. For this we have taken help of *Hash Tree*, where we have two benefits:

1. Due to it's tree based architecture, traversal is super-fast ($\sim \mathcal{O}(\log_2(n))$).
2. Since we are hashing our itemsets into bucket, we're interacting with our itemsets through it's count which we call *bucket-key* that becomes beneficial when min_sup comes into play (whole bucket gets removed when < min_sup).

2.1.1 Analysis

- **Partition-Based**:

1. **Assumptions**: Here we have taken partitions of roughly (or we can say almost) same size. And onto that we're unaware of the fact that which partition contains how many count of k-itemsets from each level of itemsets.
2. The *main idea* behind implementing this algorithm is to know *how much time* and *how efficient* it is when run parallelly on multi-core system.
3. Hence, we run it sequentially on single-core (our system) to know on which partition the total time taken is less and accordingly we can feed those number of partitions (or say those partitions) into multi-core systems.
4. Moreover, what we can examine further is to put some part of data combined to one core, and the part which is taking more time (e.g., the part which contains higher level itemsets) to another. This work distribution would be efficient. For example, say, we have split our data into 8 non-overlapping parts. Now form those, which are taking less (or least) time are fed into one core and the other into another.
5. Again, we can distribute the work of a running core to those who have completed their work to make it more efficient.
6. Here is the plot for different *no. of partition* v/s *time taken*

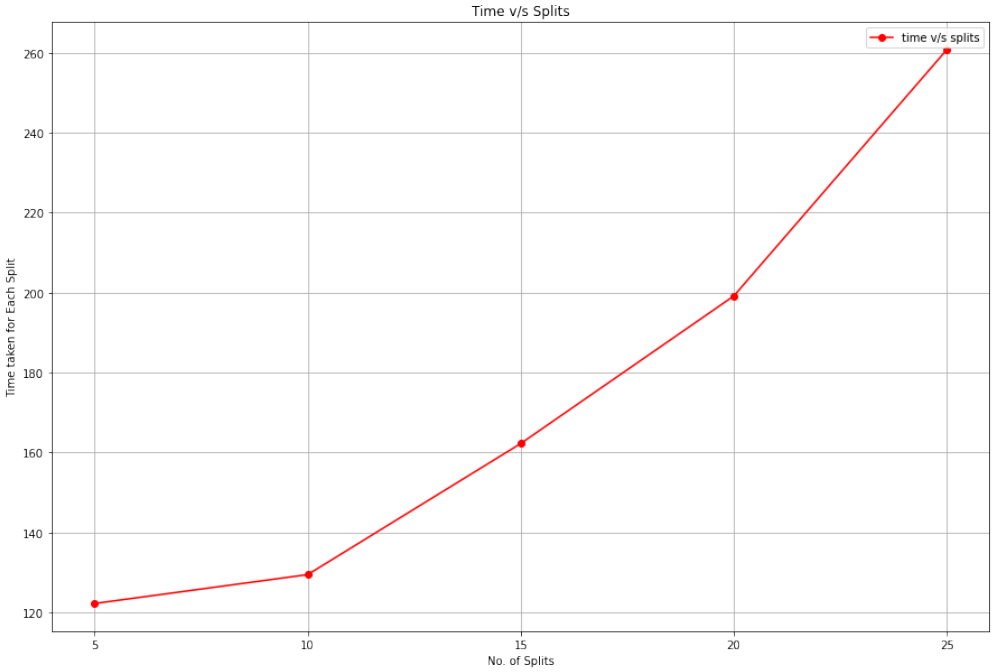


Figure 2: Time v/s Split for Partition-Based

7. Further, we've compared which fold (or part) is taking how much time and from this we can effectively parallellise our task with memory distribution.

   For no. of partitions = 5, partition P3 takes the maximum time and thus, serves as bottleneck for introducing parallelism. For distributing work among 4 cores , in this example ; one possible distribution can be P1 , P2 , P3 and P4 and P5.

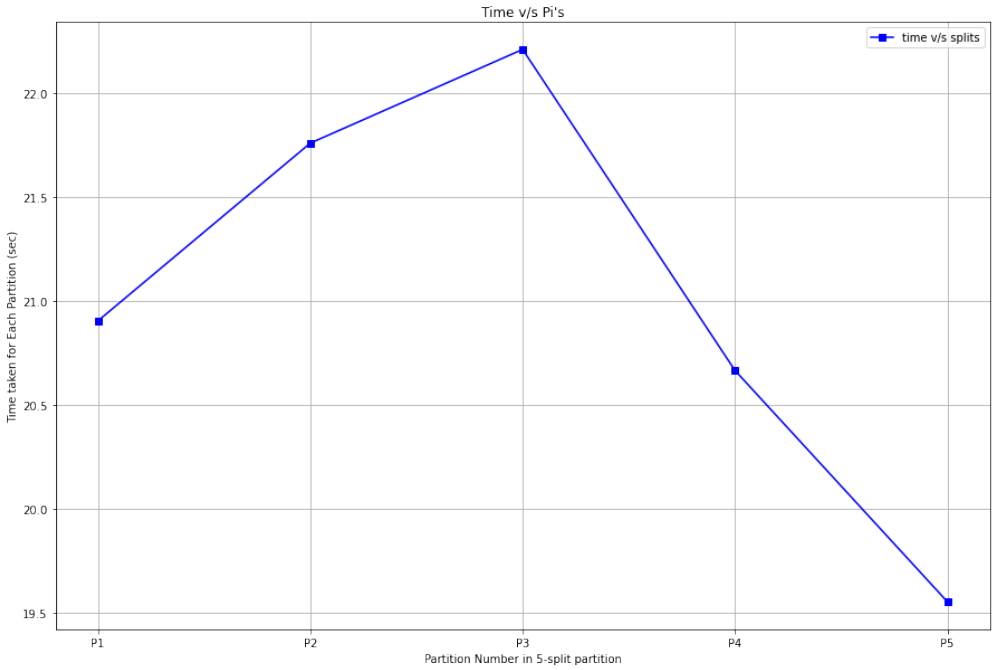


Figure 3: Fold-Wise Time taken for 5-split Data

- **Comparison**:

    1. From above graph, we got our optimal no. of split = 5. Using this optimal split for *Partition-Based* and varying the *min_sup* for other algorithms we got the following plot.

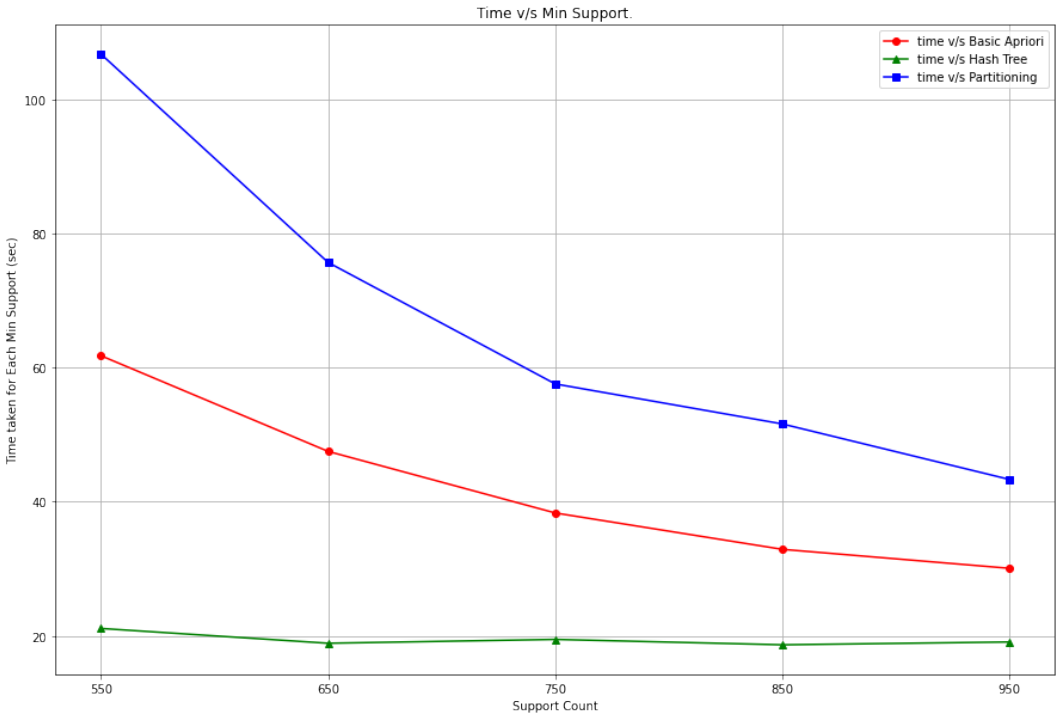


Figure 4: Comparison of Different Algorithm: Apriori; Hash-based; Partition

- **Conclusion**: So what we got at the end basically aligns with the presumptions we have for our models. And as predicted ***Hash-Tree*** is the winner among these 3.

3 References

1. Understand and Build FP-Growth Algorithm in Python
2. FP Growth: Frequent Pattern Generation in Data Mining with Python Implementation
3. Implementing Apriori algorithm in Python
4. Apriori: Association Rule Mining In-depth Explanation and Python Implementation
5. How to Find Closed and Maximal Frequent Itemsets from FP-Growth
6. Beginner's Guide To Understanding Apriori Algorithm With Implementation In Python
7. Data Mining : Concepts and Techniques