Algorithms for Frequent Itemset Mining

Eric Ingram, Dickson Nakhone, Zahra Shah, Yevhen Melnyk
COSC 254, Professor Matteo Riondato

May 19, 2021

Introduction

Description, Variations, Applications and Challenges

The task of association pattern/rule mining has emerged as an important aspect of knowledge discovery in data mining. The task is to discover a set of attributes or associations between a large number of items in a database using association rules. In the most popular association pattern mining model, the level of association is determined by the frequencies of a set of items and the discovered sets of items are known as frequent itemsets.

It is important to note that while the frequency count model is the one most widely used, the raw frequency of a pattern does not always guarantee that underlying correlation between the items is statistically significant. Thus, some models use other measures to quantify the level of association, such as the Pearson coefficient of correlation, interest-ratios and χ^2 etc.

The frequent itemsets generated can be used to generate association rules of the form of $X \Rightarrow Y$, where X and Y are sets of items. These rules are best understood in the context of supermarket data which includes sets of items bought by customers, also known as transactions. For example, the rule *Eggs, Milk* $\Rightarrow$ *Yoghurt* suggests that if a customer buys eggs and milk together, they are likely to also buy yoghurt. Thus, association pattern mining can have important applications in areas like market sales analysis. Other areas of applications include text mining, Web log analysis and software bug detection.

Unfortunately, the task of discovering all frequent associations can be daunting in very large databases. Most approaches to association rule mining are iterative in nature and scan the database multiple times. Thus, in large databases their performance is poor because the search space is exponential in terms of the database attributes, incurring high I/O costs. Methods that use sampling to get around the problem of exponential search space can be sensitive to data skew which can lead to even poorer performance. In our project, we will be looking at five different algorithms for frequent itemsets mining, which is a crucial first step for extracting associations. Our goal is to efficiently implement them and compare their performance on the grounds of speed and memory consumption.

High Level Descriptions of Algorithms

Hashtree Apriori

This approach to Apriori optimizes candidate lookup in the transaction database. This is achieved with the use of a hash tree to organize the candidate itemsets.

A hash tree is a tree with a fixed degree of the internal nodes where each internal node is associated with a random hash function that maps to the index of different children of that node in the tree; the leaf node in the tree contains a list of lexicographically sorted itemsets such that every itemset belongs to exactly one leaf node of the hash tree. As the tree is traversed at level i, a hash function is applied to the ith item of a candidate itemset to decide which branch of the hash tree to follow. The tree is constructed recursively top-down and a minimum threshold is imposed on the number of candidates in the leaf node to decide where to terminate the extension of the hash tree.

The support counting is performed by discovering all possible k-itemsets that are subsets of a transaction T in a single exploration of the hash tree via recursive traversal. At the root node, all branches are followed such that any of the items in T hash to one of the branches. At a given interior node, if the ith item of the transaction T was previously hashed at a parent node, then all items following it are hashed to determine possible children to follow. By following all these paths, the relevant leaf nodes and hence the relevant itemsets are determined. This process is repeated for every transaction to determine the support of each candidate dataset. Below we discuss some of the design choices that went into our implementations of this algorithm.

Hash Function

In our implementation, we set the degree of internal nodes to the number of different items in the dataset, and the "hash function" used is just the numerical value of the item under consideration. In a sense, what we end up with a classical trie on the alphabet of the dataset. Although this approach results in threes of considerable size (with internal degree of 100 to 1000 for our datasets), it also allows for considerable optimization of candidate generation. Recall that in normal Apriori, new candidates are generated by considering *every pair* of previously frequent itemsets which are then joined if they share all but one item. Our construction, on the other hand, has a property that all of the itemsets sharing all but the last item end up in *the same leaf* in the tree. Therefore, now we are justified in considering joining only the previously frequent candidates from the same leaf. This cuts down a significant portion of computation time, especially for low relative minimum support thresholds, for which the number of intermediate candidates can be massive. This optimization is implemented by keeping the leaf nodes of the hash-tree in a linked list for immediate access.

A potential downside of this design choice is higher computational costs of support calculation compared to what we would get with a narrower tree, since we end up with a lot more recursive calls of the support update function. Here we investigate this trade-off between faster candidate generation and faster support counting by measuring algorithm runtime as a function of out-degree of internal nodes (i.e branch factor).

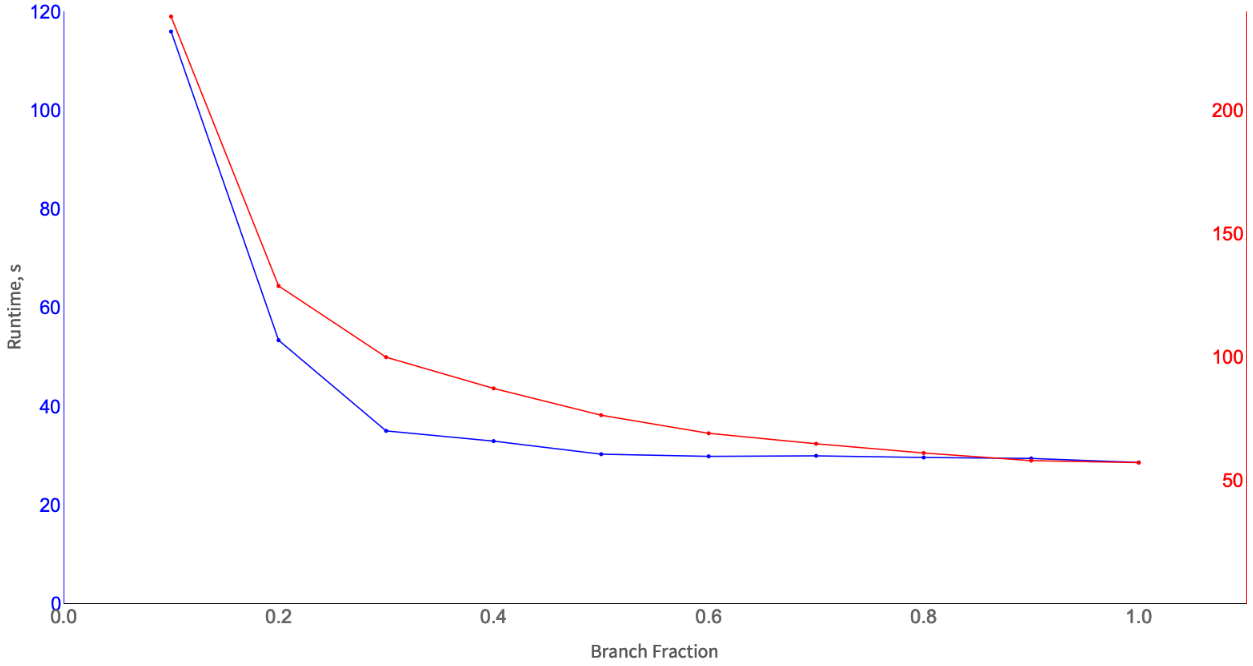

Figure 1: Run time of hash tree apriori on *mushroom* with $minsup = 2000$ (blue) and on *T10I4D100K* with $minsup = 800$ (red). The horizontal axis represents branching factor relative to maximum possible branching factor, which is given by the size of item alphabet.

We observe that the lower run-times correspond to greater branching factors. Therefore, in our subsequent evaluations we will be using max-width trees for hash-tree Apriori.

Generating multiple hash-trees vs extending existing one

Another choice we have to make is whether to generate a new tree for each iteration(i.e for each candidate length), or to extend a single tree by adding new layers of leaf nodes. The latter option seems much more reasonable since we can insert new candidates starting at the leaf node with itemsets that were used to generate the candidate, as opposed to trickling down the candidates from the root of the tree and following all of the hash functions every time. Paradoxically, a naive implementation that doesn't do such optimization and instead generates new trees seems to consistently outperform the single-tree version.

Eclat

Association rule mining algorithms can be broken down into two main categories: horizontal layout mining algorithms and vertical layout mining algorithms. The Apriori algorithm, for example, uses the most common layout, horizontal. In this type of layout, each individual transaction has a transaction identifier and a list of items in the transaction. In contrast, the vertical layout contains the set of items, and their corresponding Transaction IDs.

The Eclat algorithm uses a vertical layout and a depth-first search to mine all frequent itemsets. The advantages of a vertical layout compared to a horizontal is that one can quickly do frequency counting by doing intersection operations on Transaction IDs as well as automatically pruning irrelevant data. By doing this, candidate generation and counting occurs in a single step and the use of complex data structures is not needed. Irrelevant transactions are pruned as they drop out as a result of an intersection. The first time the algorithm runs through the dataset, all single items are used as well as their Transaction ID sets, and the support is checked against the minimum threshold. Then, the algorithm is called recursively for each level of k (with the itemsets found to be frequent at the current level of k) and checked until there are no candidate itemsets to be generated or no frequent items are found. This approach is advantageous compared to the horizontal approach, as you scan a dataset that is pruned at each value of k instead of scanning the entire dataset each time and use less memory by using a depth-first search.

dEclat

When the cardinality of the Transaction ID set gets very large, a vertical layout begins to suffer as the time to calculate intersections becomes large. Similarly, the size of intermediate Transaction ID sets can also become very large, requiring more memory and therefore affecting the algorithm scalability. As a result, in dense datasets the vertical approach loses its advantage over the horizontal. To help solve this problem, a novel vertical data representation called diffset is introduced. A diffset only keeps track of differences in the Transaction IDs of a candidate pattern from its generating frequent patterns. As a result, diffsets drastically reduce the amount of memory required to store intermediate results. Further, if the initial dataset is stored in diffset format, the total dataset size can decrease. Since diffsets are a small fraction of the original size of a Transaction ID set, intersection operations are much faster.

By using diffsets, the dEclat (DiffEclat) formula is born. This approach makes the already existing Eclat algorithm more scalable, allowing it to generate frequent patterns even in dense datasets for relatively low support values.

FPGrowth

FP growth algorithm utilizes the FP tree data structure to condense the database, while still keeping the important information about the data, including associations and support. In theory, (and also in practice, as we will see in this report) FP-growth outperforms Apriori because it has several ways in which it makes efficiency gains over apriori:
1. a large database is compressed into a highly condensed, much smaller data structure, which avoids costly, repeated database scans.
2.FP-tree-based mining adopts a pattern fragment growth method to avoid the costly generation of a large number of candidate sets.
3. A partitioning-based, divide-and-conquer method is used to decompose the mining task into a set of smaller tasks for mining patterns in conditional databases, which reduces the search space.

The FP Tree Data Structure - Only frequent length-1 items will have nodes in the tree, and the tree nodes are arranged in such a way that more frequently occurring nodes will have better chances of sharing nodes than less frequently occurring ones.

Once we have the FP tree, a conditional pattern base and then a conditional FP tree is constructed for each of the frequent 1-itemsets, and mining is performed recursively on the conditional FP trees.The pattern growth is achieved via concatenation of the suffix pattern(the initial suffix pattern is the frequent 1-itemset) with the new ones generated from the conditional FP-tree.

Evaluation

All evaluations were run on a 2019 MacBook Pro with a 2.4 GHz 8-Core Intel Core i9 CPU and 32 GB 2400MHz DDR4 RAM.

Runtime

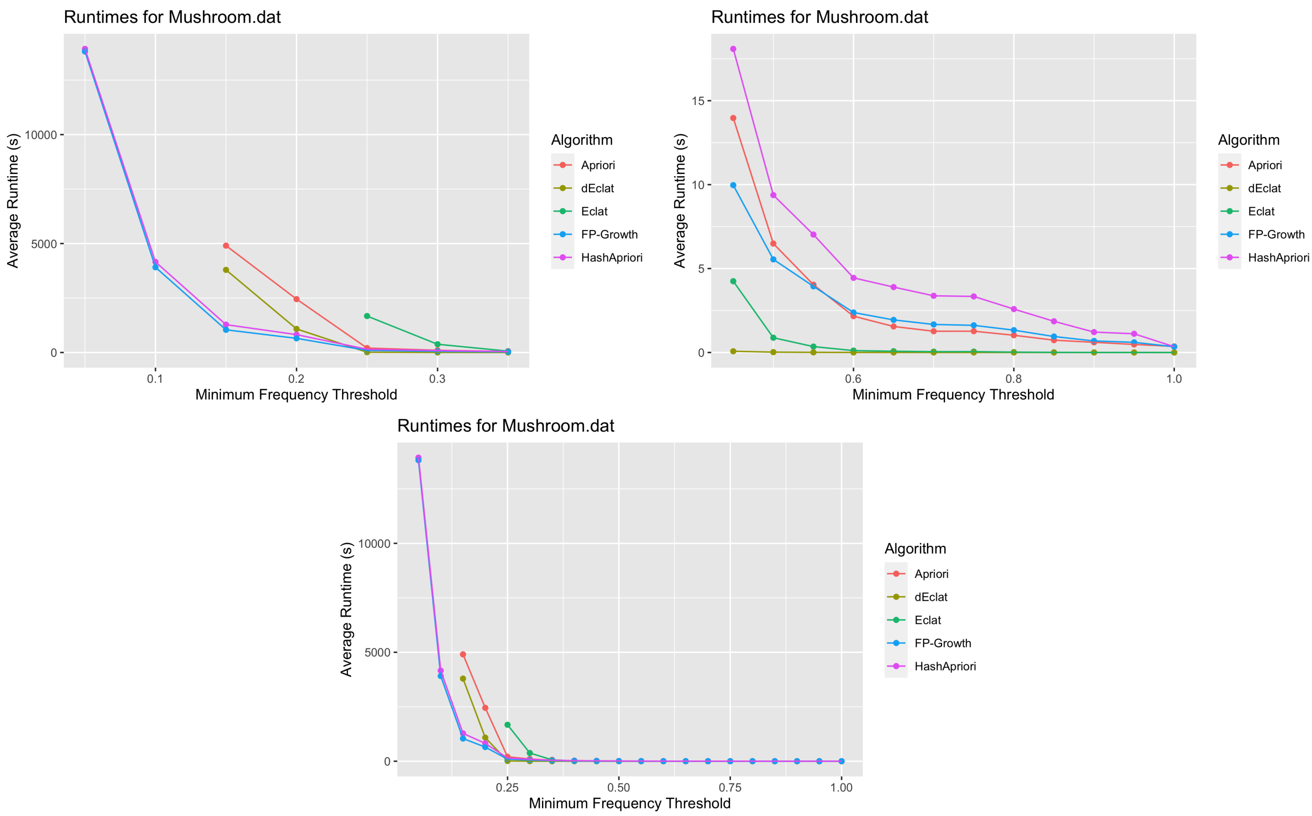

Figure 2: Comparison of runtime as a function of minimum support threshold for $Mushroom$.Three different ranges included to make graphs distinguishable.

In Figure 2, we can see how the runtimes differ for our 5 algorithms at different minimum frequency thresholds. For a minimum frequency threshold below 0.35, Hash-Apriori and FP-Growth outperforms all of the algorithms, as we can see in the top-left subfigure of figure 1. Below a minimum frequency threshold of 0.15,we could not compare the performance of Apriori, dEclat and Eclat (in fact the last minimum frequency threshold that Eclat ran for was even higher, at 0.25) because their runtimes were astronomical and could not be completed on our computer. The only algorithm that completed its run at all thresholds were Hash-Apriori and FP-Growth, but even they had large runtimes. At a threshold of 0.0, Hash-Apriori and FP-Growth both took around 3.9 hours to complete. Given the large runtime, we can extrapolate for our other algorithms and safely assume that they will likely take much more time than Hash-Apriori for thresholds below 0.25.

For thresholds above 0.25, dEclat performs consistently better than all the other algorithms. In fact, its runtime is almost 0 seconds for all thresholds above 0.25 (top-right subfigure). This makes sense intuitively, because dEclat is best suited to dense datasets where calculating the diffsets should not be as hard or time-consuming as calculating tidsets since items are found in most transactions. Surprisingly, after a threshold of around 0.4, Hash-Apriori performs the worst in terms of runtime.

Thus, in term of runtimes, the order of best algorithm performance in the range of 0.0-0.25 minimum frequency threshold is FP-Growth, Hash-Apriori, dEclat, Apriori, Eclat whereas in the range of 0.25-1.0 the order is dEclat, Eclat, FP-Growth, Apriori, Hash-Apriori. So, FP-Growth is overall the best choice as it preforms best for very low minimum frequency thresholds and preforms decently for the second range. This also gives an indication of the possibility that our dEclat is not perfectly optimized since it should, in theory, work well with dense datasets for all values of minimum frequency thresholds. That may be because we are not completely utilizing the concept of prefix classes.

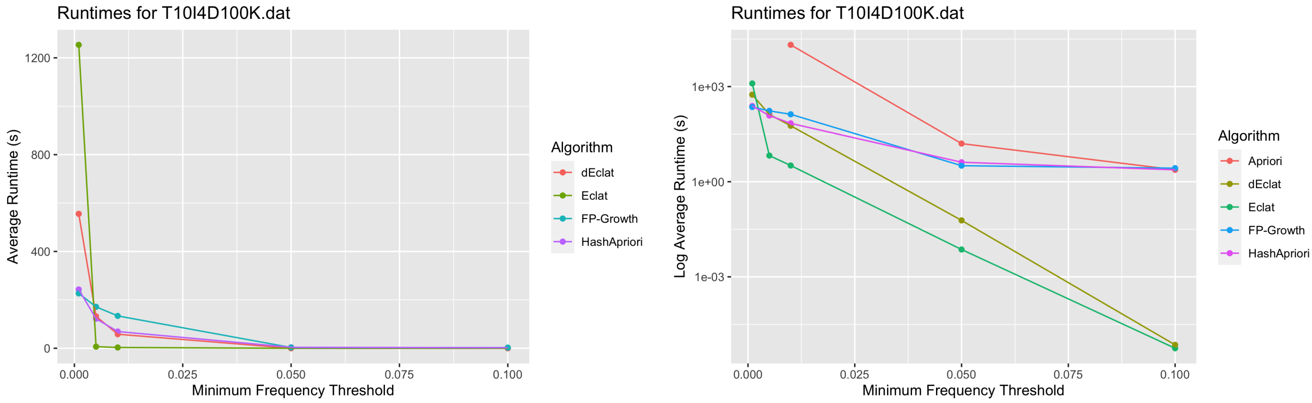

Figure 3: Comparison of runtime as a function of minimum support threshold for $T10I4D100K$. Apriori was excluded from the linear plot since it made the results of other algorithms indistinguishable.

Now we repeat a similar experiment with *T10I4D100K* dataset, which is characterized by having a much bigger item alphabet than that of *mushroom* (114 vs 999). This in turn implies that our algorithms have to deal with significantly more intermediate candidates(since intuitively, the number of candidates of length l under consideration is proportional to $|I|^l$). Therefore, in this experiment it is critical for our algorithms to have a fast way to calculate the supports for the massive number of candidates.

Figure 3 shows the runtime curves for this dataset. We observe that for higher values of minimum frequency thresholds, which is precisely when we have a moderate amount of intermediate candidates, Eclat and dEclat again exhibit run-times that are noticeably smaller that those of Apriori and Hash-Apriori. However, for lower values of frequency threshold, Hash-Apriori and FP-Growth run up to 4 times faster than Eclat and 2 times faster than dEclat. This affirms our previous considerations, since Hash-Apriori and FP-Growth are our only algorithms that leverage a mechanism for efficient support counting that grows much slower with the number of candidates generated. Once again, the choice of the optimal algorithm depends on the task at hand.

Memory Consumption

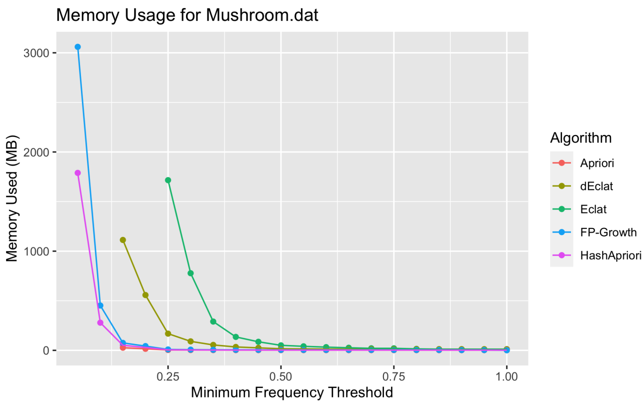

Figure 4: Comparison of memory usage in megabytes as a function of minimum frequency threshold for $Mushroom$, a dense dataset.

In Figure 3, we can see the memory consumed by a run of each of our 5 algorithms across different minimum frequency thresholds. At a threshold of .15, Apriori surprisingly used the least amount of memory at 27 MB. At the same threshold, Hash Apriori used nearly twice as much memory at 51 MB. These numbers pale in comparison to dEclat, however, as it used 1.1 GB of memory at the same threshold (an over 200% increase from Hash Apriori). We couldn't compare Eclat at the same threshold, however, as Eclat's runtime surpassed what we could measure at a minimum frequency threshold of .15. But, we can extrapolate from a threshold of .25 as both the Eclat and dEclat curves seem to be following a similar exponential pattern. At that threshold, Eclat used 1.7GB of memory compared to 168 MB for dEclat. With such a drastic difference at that threshold, we can only expect that an even larger difference exists at a threshold of .15 and that Eclat takes many gigabytes to run. As a result, we can conclude that in terms of memory efficiency for $Mushroom$, the ideal order is Apriori, Hash Apriori, FP-Growth, dEclat, and then Eclat.

This conclusion supports the premise behind dEclat. dEclat was created to address Eclat's weakness of large transaction ID lists being created in dense datasets causing memory usage to skyrocket. We can see the drastic decrease of 1.5 GB of memory usage between Eclat and dEclat at a threshold of .25 as evidence that diffsets do work to decrease memory usage in dense datasets. However, Eclat, dEclat, and Apriori suffered from large runtimes that did not allow us to gather memory analytics for extremely low minimum frequency thresholds. We were able to gather analytics for Hash Apriori and FP-Growth down to a threshold of .05, however, showing that their runtime was the lowest at a threshold of .05 and as a result likely had the lowest memory usage as well. As a result, we conclude off of Figure 3 that Hash Apriori is the most memory efficient (as well as time efficient) algorithm for $Mushroom$.

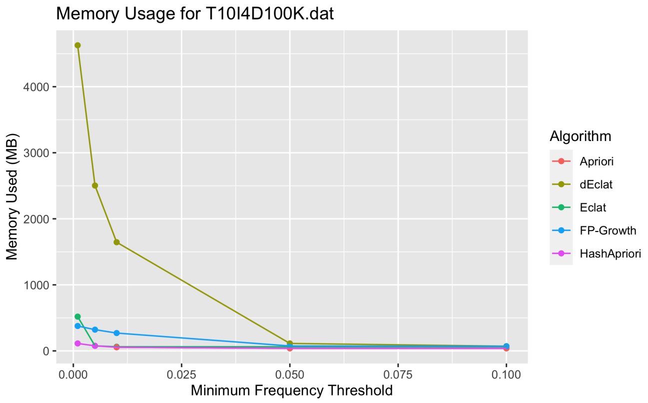

Figure 5: Comparison of memory usage in megabytes as a function of minimum frequency threshold for $T10I4D100K$, a sparse dataset.

For this dataset, Hash Apriori and Apriori seem to outperform the other three algorithms in memory usage at all levels of the minimum frequency threshold recorded. FP-Growth seems to follow a similar pattern to Hash Apriori, but at a higher amount of memory (about a 300 MB difference) At very low levels of the frequency threshold, dEclat performs the worst, with a memory usage of more about 5.4GB. Eclat performs significantly better at this level, with a memory usage of about 500MB. For both dEclat and Eclat, the memory usage decreases exponentially as the minimum frequency threshold increases. If this dataset is a sparse dataset, then our results support the theoretical claims of the algorithms. While the use of diffsets allows dEclat to use less memory when a dataset is dense, it is a drawback when the dataset is sparse. In fact, Zaki and Gouda state that diffsets could occupy several magnitudes more space than tidlists when the dataset is sparse. This explains why dEclat uses the most memory on this dataset. As the minimum frequency threshold increases, we expect memory usage to decrease for both dEclat and Eclat because of fewer frequent itemsets. Our results support this. Our results show that for this dataset, at very low levels of the minimum frequency threshold, the most memory-efficient algorithms are in this order: Apriori, HashApriori, FP-Growth, Eclat, dEclat. However, as the minimum frequency threshold increases, all the algorithms seem to have similar memory requirements, although HashApriori and Apriori still retain a small edge.