Introduction

The Apriori Algorithm is one of the first algorithms learned when mining frequent patterns from transaction data. It relies on the knowledge of the transaction database size, which can be easily found after reading the data into your programming language of choice, and also requires the setting of a *minimum relative support threshold*, which indicates what fraction of the transactions need to include an itemset for it to become frequent.

The Apriori Algorithm

The implementation of the algorithm is fairly straightforward, only requiring some understanding of set operations and some understanding of data structures in your language of choice. Python is the language in which the implementation is the most simple because of the built-in set data structure, though pattern discovery time will be considerably slower in a dynamically typed and interpretive language such as Python.

Generate the Frequent 1-Itemsets
Beginning with the dataset, the frequent 1-itemsets are the first thing that need to be generated. In Python, this can simply be done by traversing through each list, counting the number of times each item appears, then saving the items (and their counts) whose counts are higher than the minimum absolute support threshold.

Generate the Frequent K-Itemsets
Once the frequent 1-itemsets have been generated, the set of candidate 2-itemsets are combinations of two items from the list of frequent 1-itemsets. The candidate 2-itemsets can be run against the transaction database and the frequent 2-itemsets can be discovered. We now begin a repeating process, there the frequent *(k-1)*-itemsets are used to generate the candidate frequent *k*-itemsets by brute-force calculating the set of unions between all combinations of the frequent itemsets that are of length *k*. The candidate k-itemsets are then pruned, by ensuring that all *(k-1)*-subsets of the candidate are present in the previous step's frequent itemsets (making use of the theorem that if a *k*-itemset is frequent, then so are all of its *(k-1)*-subsets). Finally, the pruned candidate *k*-itemsets are then run against the transaction database, and the ones with counts that exceed the minimum absolute support threshold are deemed frequent.

When to Break the Loop
The algorithm continues to loop until all frequent itemsets with length up to the length of the frequent 1-itemsets are discovered. However, more commonly there will be situations where the largest frequent *k*-itemset is of substantially smaller size than the size of the frequent

1-itemsets. In this case, we know that once there are no frequent itemsets of a size *k*, we can break the loop, as there must be at least two itemsets of size *k* to generate a frequent itemset of size *k+1*.

Lessons Learned From Implementation

When performing set logic operations, work with integers as frequently as possible.
My initial implementation of the algorithm took ages to run and generated some incorrect results. I eventually traced the poor performance back to my use of set operations on string based sets. Because Python's strings are iterable structures (which is not true in some other languages, such as Java) each string was also being treated as its own set, which made computations incredibly long and highly susceptible to errors. My first full completion of the algorithm took 4 hours on a transaction database of ~77,000 transactions. Needless to say, this would not scale well to a massive transaction database at all.

To resolve this issue, I decided to map each category present in the transaction database to an integer ID number. Because set operations are much faster on sets that contain integers, this ended up speeding my program up by three orders of magnitude and generated more accurate results. This method would scale much better to larger transaction databases (though it still wouldn't be great) and can be even further optimized by implementing in a compiled language (i.e Java, C++, Rust) or by doing any of the numerous search optimizations seen in the lecture videos.

The key takeaway here is that if there is ever a situation where working with sets and set operations in a Python program is necessary, any strings present in the set should be mapped to integer IDs, and converted back into their respective strings once all of the set operations have been completed.