EMORY UNIVERSITY

Homework 2

Student name: *Lance Ding*

Course: *CS470* – Professor: *Dr. Kai Shu*
Due date: *3/2025*

Collaboration Statement

The following resources were utilized in the completion of this project:

- Dr. Kai Shu's lecture slides on canvas
  - Pseudocode and idea of the algorithms
- https://www-users.cse.umn.edu/ kumar001/dmbook/ch5_association_analysis.pdf
  - Pseudocode and text explanations of the algorithms and concepts
- ChatGPT
  - Formatting latex, concept checks.
- FIMI repository
  - Inspired me to look into using tries. While I did not follow their implementation/optimizations, I did end up using a trie.

Apriori.py Documentation

For this assignment, we were tasked with implementing the Apriori algorithm - a popular algorithm used in frequent itemset mining that utilizes the apriori principle. In specific, we were asked to produce a script that would be able to take 3 arguments in the command line - *input*, *min_support*, and *output* - and output a *.txt* file. Each line represents one frequent itemset, and each i_k represents an item, with the support (absolute) of the itemset in parenthesis.

$$i_1\ i_2\ ...\ i_n\ (sup)$$

For the script we created (*apriori.py*), we provide the following command line options:

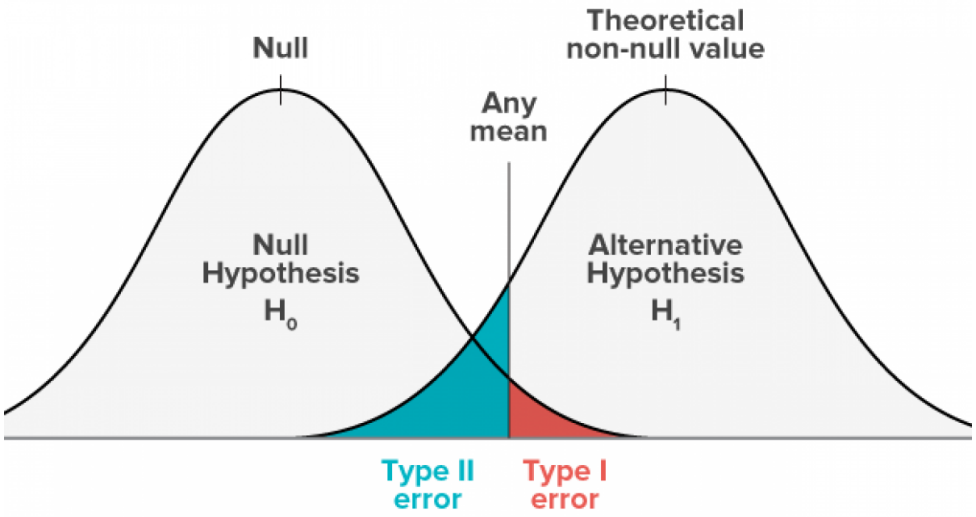
- *--input <filename>* - The name of the input file, or the relative directory of the input file with respect to the *HW2* root. Defaults to *data.csv*
- *--min_support <integer>* - The minimum support for the frequent itemsets. Defaults to 500

- *--output <filename>* - The name of the output file, or the relative directory of the output file with respect to the *HW2* root. Defaults to *output.txt*
- *--colname <string>* - The name of the column to apply the algorithm to. Defaults to *text_keywords*
- *--pickle <boolean>* - Whether or not to export a serialized pickle object of the frequent itemsets as they are returned by the algorithm. Defaults to *False*
- *--algorithm <string>* - The variant of the algorithm to run. The user may choose from *k1km1*, *km1km1*, and *trie*, where *k1km1* corresponds to the naive Apriori algorithm with the $F_1 \times F_{k-1}$ candidate generation scheme, *km1km1* corresponds to the naive Apriori algorithm with the $F_{k-1} \times F_{k-1}$ candidate generation scheme, and *trie* corresponds to an Apriori variant that uses a trie to store frequent itemsets and uses the $F_{k-1} \times F_{k-1}$ candidate generation scheme. Defaults to *km1km1*.

For further instructions on operating the script, as well as environment setup, refer to *README.md* in the *HW2* root.

Chosen Minimum Support Threshold

While there is no single way of picking a minimum support threshold, so we take inspiration from hypothesis testing. In hypothesis testing, α represents the probability of observing a set of events that is unusual to see given our assumptions of the underlying data distribution. In essence, α is a boundary that we set as a hyperparameter that distinguishes significant and insignificant sets of observations.



It also corresponds to the probability of falsely rejecting the null hypothesis when it is actually true (Type 1 error), so a small value for α is usually set for a higher confidence in the test. A common threshold is $\alpha = 0.01$, meaning that only when our observations fall within a certain 1% area of possible observations can they be considered significant.

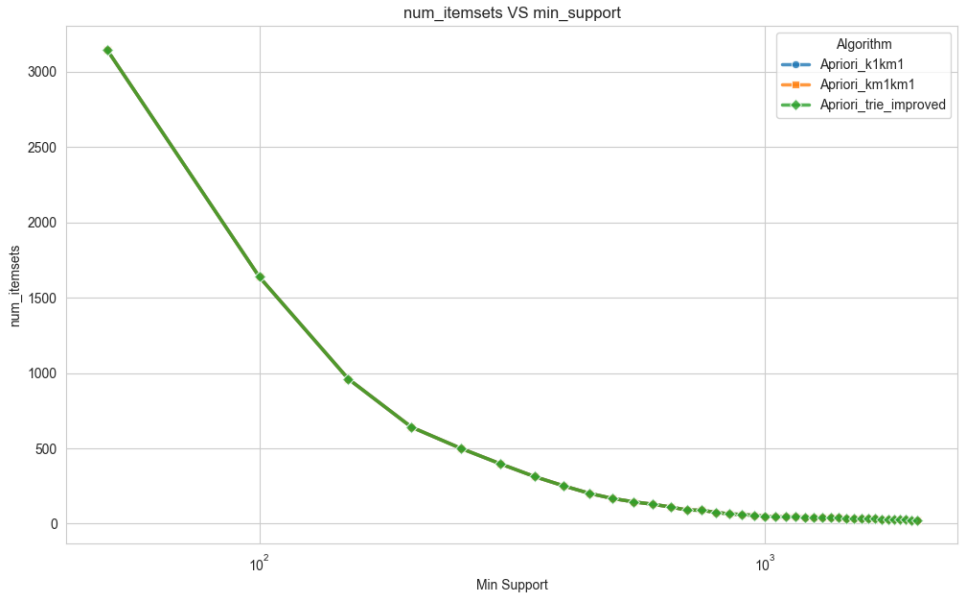
We take inspiration from this concept, and ask that our frequent itemsets also have support that is at least as much as the significance threshold in hypothesis testing. That

is, we set our minimum support threshold such that only itemsets that are supported by at least $\alpha = 0.01 = 1\%$ of the entries will be considered frequent to ensure that only the most statistically significant itemsets are considered.

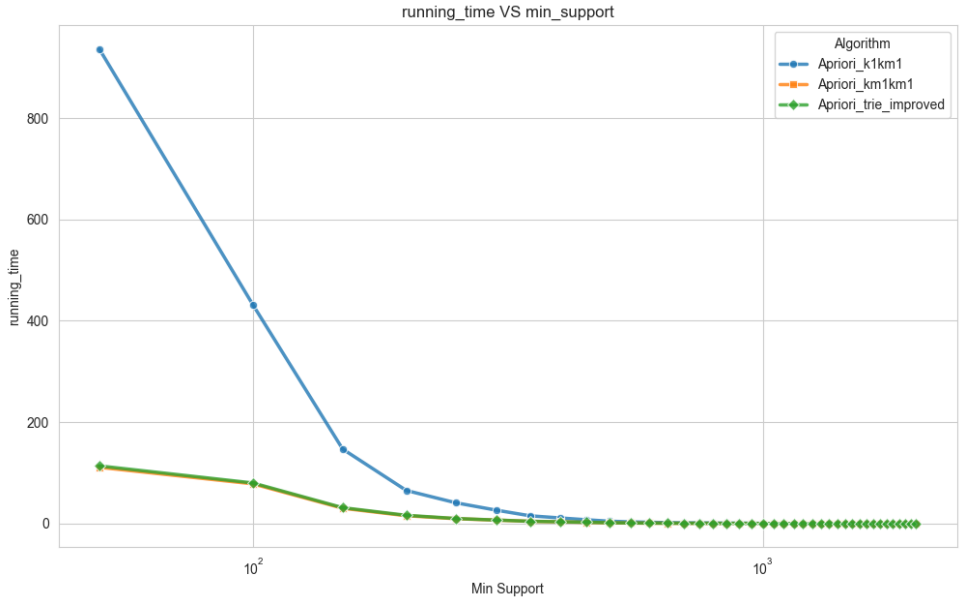
This approach aligns with the idea that frequent patterns should not be mere statistical noise but should represent meaningful structures in the dataset. By setting the minimum support threshold in this way, we aim to balance sensitivity (detecting useful patterns) and specificity (avoiding spurious correlations).

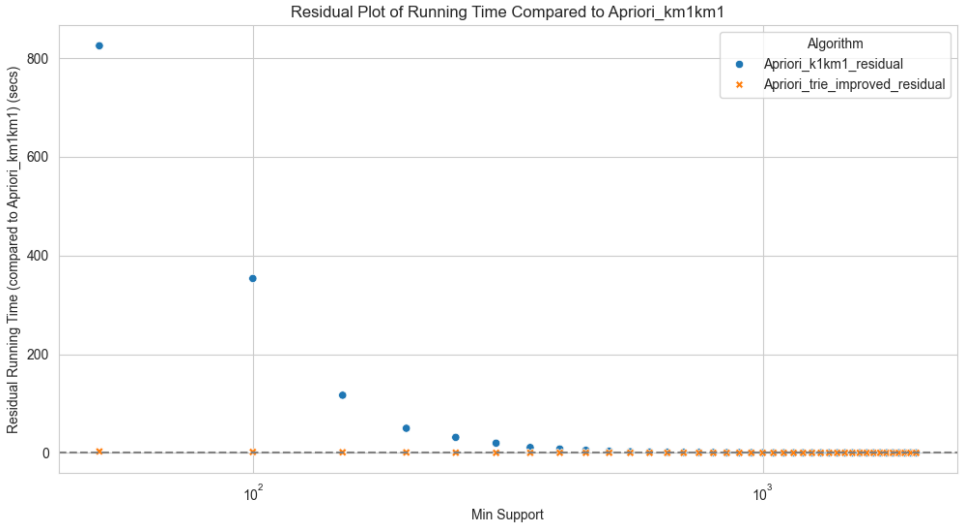
From EDA, we know that there are approximately 26000 entries, and with an α of 0.01, we would require a minimum support of $26000 \cdot \alpha = 26000 \cdot 0.01 \approx 260$. This produces 476 frequent itemsets.

We also considered allowing only $\alpha \cdot n(entries)$ frequent itemsets, but that does not line up with how confidence would work, since we would need to use the number of total possible itemsets to determine the threshold. As it is likely that our data is sparse and does not spread evenly across the space of possible itemsets, we deemed it unfit to proceed with this approach, as the minimum support generated would likely be way too large. Anyhow, if we were to proceed with that approach, we would approximate the required minimum support by sweeping the minimum support and checking how many frequent itemsets are found.



In terms of the tradeoff between efficiency and number of frequent itemsets, we can visualize this utilizing the data collected during the sweep for the minimum support threshold. Running time, in seconds, for all three variants of our Apriori variants are displayed.

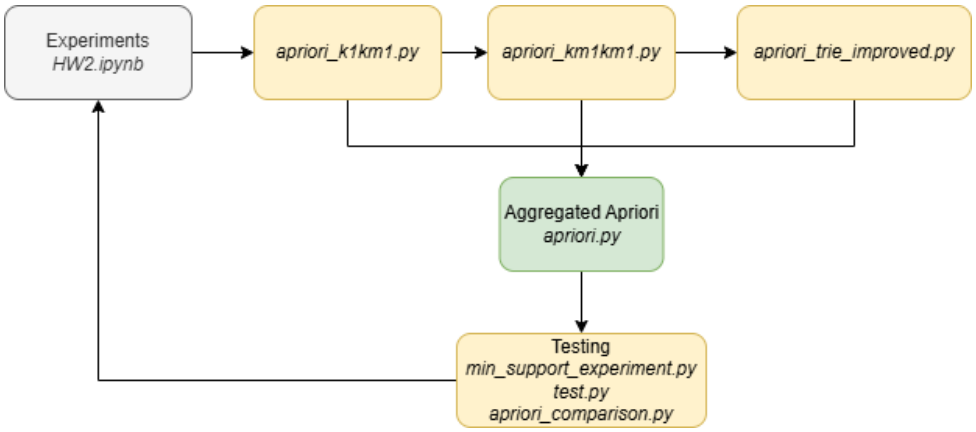




The sweep starts at a minimum support of just 50, and steps up to 2000, with each step being 50. We can see a drastic decrease in the amount of time required to run the algorithm as we sweep the minimum support threshold. We can also see that the $F_1 \times F_{k-1}$ implementation is significantly slower than the other two. However we can see that for our data, running time is not a real concern because even at a minimum support of 50, which is just $\approx 0.2\%$ of our data, the running time for our faster algorithms does not exceed 3 minutes.

Algorithm Implementation and Optimizations

We implemented the Apriori algorithm according to Dr. Kai Shu's lecture materials and the online source we referenced at the front. We started with the $F_1 \times F_{k-1}$ candidate generation scheme, before moving on to the $F_{k-1} \times F_{k-1}$ scheme, and finally adding a trie. The development process can be roughly described with the following flowchart:



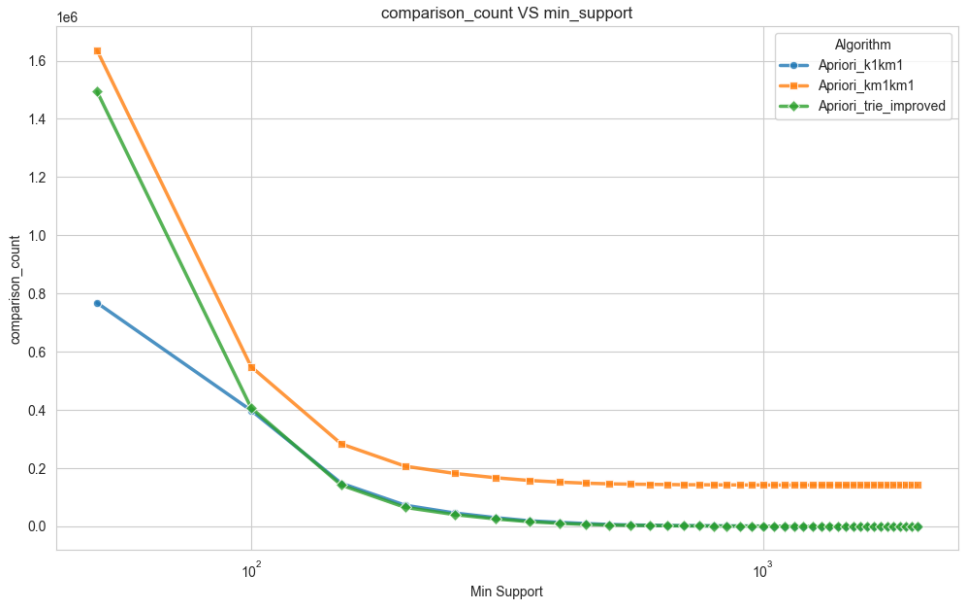
We utilize several optimizations that are not explicitly mentioned in the lecture slides. We initially planned on stopping at the $F_1 \times F_{k-1}$ implementation, but after realizing how slow it was due to the amount of extra candidates it would generate, we decided to implement the $F_{k-1} \times F_{k-1}$ candidate generation scheme.

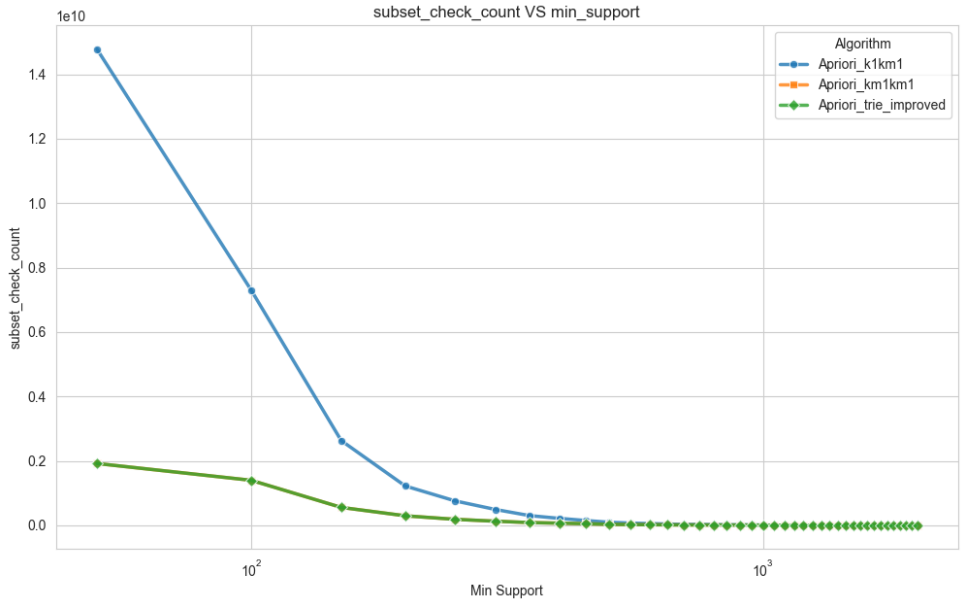
Furthermore, we utilize python's *frozenset* class to store itemsets. Unlike the more commonly used *set* object, the *frozenset* is immutable. This allows for more efficient memory usage, and potentially faster set operations. Additionally, using sets allows us to very quicly perform mathematical set operations when compared to other data structures like lists.

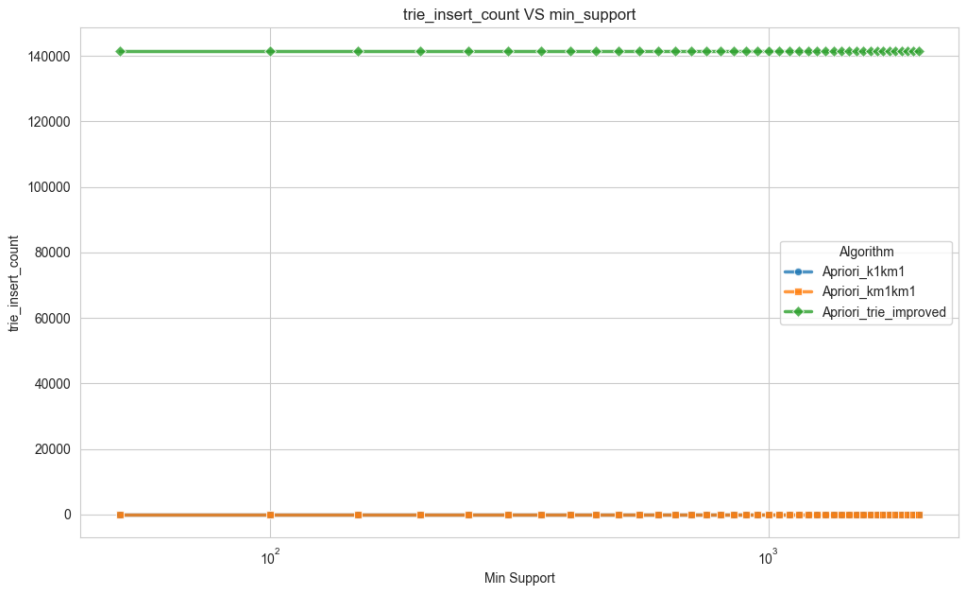
Additionally, we implement a trie-based apriori algorithm. This was inspired by the FIMI repository, which outlined a trie implementation of the apriori algorithm that was supposedly a major improvement. We found the technical details of that implementation to be complicated, so we took inspiration by creating a trie-based apriori algorithm. This algorithm uses a trie to store and count the frequent itemsets. Theoretically, this saves space since hash-based sets typically require a much larger amount of space. While we still utilize sets in the trie implementation, we utilize it per node to store its children. If the number of overlaps between itemsets is large, then the amount of space this implementation can save would be quite large. We also improve on this approach by sorting the itemsets before processing them in the trie, which improves the chance that itemsets with overlapping elements necessarily fall under the same branch. However

Results

Besides the plots already shown, we can also measure the amount of computation required for each variant of our algorithms. We can look at the number of comparisons, subset checks, and trie insertions:







We note that the trie implementation's trie insertion count is equivalent to the difference between the $F_{k-1} \times F_{k-1}$ implementation and the trie implementation. In practice, we see that the $F_1 \times F_{k-1}$ implementation is by far the slowest, while the $F_{k-1} \times F_{k-1}$ implementation is fractionally faster than the trie implementation. This seems to indicate that the time it takes for python to do comparisons is longer than the time it takes python to perform a trie insert, which requires object instantiation and a set operation. This difference may also be affected by the way we are counting operations.

With a minimum support of 260, 476 frequent itemsets were generated. The results can be found at *260output.txt*. Additionally, the output for a minimum support of 500 can be found at *std_output.txt*.

In terms of patterns discovered, we can see that the most frequent itemsets are, unsurprisingly, about either the flu, or the flu shot. Many frequent itemsets describe sickness, getting the flu shot, and time-marked versions of other frequent itemsets. Some itemsets point out the arm getting the flu shot, and the physical feelings that the people felt. Since this is a dataset about tweets relating to flu shots, these findings are to be expected.

There is a general negative sentiment to the flu shots, which is expected since they are associated with illness and negative physiological reactions. Past that, however, there aren't many clear insights that can be derived from the frequent itemsets, besides the fact that the people tweeting have recently received, or are talking about, their flu shots and how they have been affected.

Lessons Learned

During the implementation of the Apriori algorithm, several challenges arose that provided valuable learning experiences. One of the most significant difficulties was interpreting the results and drawing meaningful insights from the frequent itemsets. While the algorithm efficiently identified frequent patterns, understanding their significance within the dataset required additional analysis and contextual knowledge.

Another challenge was following the logic of the FIMI trie-optimized Apriori algorithm. While inspired by the FIMI repository, the complexity of its optimizations made it difficult to fully grasp and implement. Instead, a simpler trie-based approach was developed, but there is still room to refine and further optimize this method in the future.

Determining an appropriate minimum support threshold also proved to be a non-trivial task. While a hypothesis-testing-inspired approach was used, alternative methods could have been explored to ensure a balance between computational efficiency and meaningful results.

For future work, several improvements can be made. Spending more time analyzing the dataset before implementation could help in setting a better minimum support threshold and refining the algorithm's parameters. Additionally, a deeper understanding of existing implementations, such as those from the FIMI repository, could lead to more efficient optimizations. Exploring different optimizations, including better data structures or pruning techniques, could further enhance performance. Finally, switching to a faster programming language like C++ could significantly improve execution speed for large datasets.