Jakub Niedziela
s0224740

Data Mining 2023 – Assignment 2: Recommendations using non-derivable association rules

Task 1: Issues with the apriori algorithm

Firstly, I have noticed a significant issue with the apriori algorithm that required attention. It seemed to be iterating through the entire dataset, even when the support count dictionary was empty. This meant that all candidates were below the min_support threshold. To address this inefficiency, I implemented a simple solution by adding a break statement. This improvement allowed the algorithm to stop processing when the support of itemsets at the current level, denoted as k, was too low.

Another critical flaw I identified was the incorrect generation of itemsets for the subsequent apriori iteration. The algorithm was considering all items present in transactions, rather than limiting it to frequent subsets. This violated the fundamental principle of monotonicity in the algorithm, which states that "A subset of a frequent itemset must also be frequent, or any superset of an infrequent itemset must also be infrequent!" To rectify this inconsistency, I modified the code to generate itemsets solely from frequent subsets, ensuring the adherence to monotonicity.

During the development process, I sought the opinion of Chat GPT regarding the correctness of the implementation. It raised several concerns, including the efficiency of the itemsets_support function, memory usage, and the utilization of frozensets. I do agree with the first two points raised by Chat GPT, acknowledging the need for further optimization and reduction of memory consumption. However, I believe the use of frozensets may not be entirely unfavorable. In fact, frozensets possess all the necessary properties for this scenario and offer the advantage of immutability. It is worth noting that established open-source Python packages implementing the apriori algorithm also employ frozensets.

Lastly, I conducted an experiment to evaluate the performance of the initial version of the apriori algorithm and the enhancements made. I compared these results with an existing open-source apriori algorithm and presented the results in Table 1. The table showcases the execution times of different algorithms, considering factors such as the size of transaction data (N), the number of unique items (I), and the minimum support threshold. Although the GPT and GPT fixed versions exhibited faster runtimes than the mlxtend apriori for smaller datasets, it is evident that they may not be able to compete with it when dealing with large, real-life datasets. Nevertheless, the improvements I implemented in the initial version of apriori, have demonstrated significant speed enhancements as the dataset size increases.

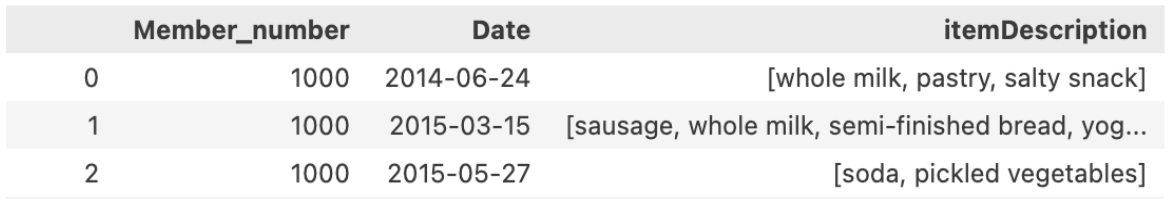
Table 1. Results of comparison of apriori algorithms.

N / Algorithm	Raw GPT version	Fixed GPT version	Mlxtend version
N = 7 | I = 3 | S = 0.1	11.6 us (10^-6 s)	11.8 us (10^-6 s)	930 us (10^-6 s)
N = 20 | I = 7 | S = 0.1	1.12 ms (10^-3 s)	324 us (10^-6 s)	1.51 ms (10^-3 s)
N = 14963 | I = 167 | S = 0.1	Over 30 minutes	108 ms (10^-3 s)	3.27 ms (10^-3 s)
N = 14963 | I = 167 | S = 0.01	---------------------	1.88 s (10^0 s)	122 ms (10^-3s)

Task 2: Improvement via ranking methods.

When it comes to evaluating recommender systems, I obtained a dataset from Kaggle.com, an online platform. The dataset consisted of rows containing unique user IDs (Member_number), purchase dates (Date), names of the items bought (itemDescription), and additional information such as year, month, date, and day of the week. Since each row represented a single item, I needed to group them together. Therefore, I performed grouping based on Member_number and Date, resulting in a groceries dataset comprising 14,963 transactions and 167 unique items. Figure 1 provides a subset of the dataset with N=3.

Figure 1. Groceries dataset.



	Member_number	Date	itemDescription
0	1000	2014-06-24	[whole milk, pastry, salty snack]
1	1000	2015-03-15	[sausage, whole milk, semi-finished bread, yog...
2	1000	2015-05-27	[soda, pickled vegetables]

The next step involved preparing the data by creating train and test datasets. Initially, I opted to randomly split the baskets into 90% train set and 10% test set. For the train set, I generated two dictionaries: test_data and test_labels. These dictionaries were created by selecting a basket that was not present in the train set, removing one random item, and updating the dictionaries accordingly. The test_data dictionary contained baskets with dropped items, while the test_labels dictionary contained baskets with the items that were dropped. This setup allowed us to evaluate whether the recommender system accurately predicted the missing items.

Moving on to the evaluation phase, I utilized three recommender system evaluation methods, each with different levels of apriori's minimum support: 0.0001, 0.001, and 0.01. To identify the best combination of metrics, thresholds, and the number of returned items by the recommender, I employed a grid search approach. However, it's important to note that the values of min_support=0.3 and min_confidence=0.7 proposed by ChatGPT were not applicable in this context, as the apriori algorithm would not generate any frequent itemsets.

To summarize, the flow of the evaluation process was as follows:

1. Encoding the baskets (specific to the mlxtend apriori and association_rules methods).
2. Creating frequent itemsets using the apriori algorithm for each min_support threshold (0.0001, 0.001, 0.01).
3. Conducting a grid search of parameter-threshold pairs.
4. Sorting the results based on each evaluation metric.

The results of the best iterations, sorted by highest precision, are presented in Table 2, while Table 3 displays the results sorted by highest recall, and Table 4 presents the results sorted by the highest f1 score. For the min_support value of 0.01, there is a lack of precise metric/threshold/top_n combinations, as most of them reported the same values. Hence, only the highest value is included.

In conclusion, it can be observed that a lower min_support value yields slightly better results. However, it is challenging to determine if a lower value should be adopted, considering the computational expense in terms of time and memory usage. Moreover, the scores presented in the tables might appear low, but it is important to consider the specific nature of the data and the industry to which it applies. In this case, the dataset consists of small baskets with a limited number of items, possibly from a local shop. In such a context, even recommendations with relatively low scores can be beneficial as they expose customers to other products. However, when applying this recommendation system to an industry where accurate recommendations are crucial, more complex methods would be required.

Table 2. Highest results by recall.

Min_support	Metric	Threshold	Top_n	Score
0.01	-------------------	-------------------	-------------------	0.043
0.001	Confidence	0.05	5	0.181
0.0001	Confidence	0.05	5	0.203

Table 3. Highest results by precision.

Min_support	Metric	Threshold	Top_n	Score
0.01	-------------------	-------------------	-------------------	0.110
0.001	Support	0.01	1	0.098
0.0001	Conviction	1.25	5	0.200

Table 4. Highest results by f1 score.

Min_support	Metric	Threshold	Top_n	Score
0.01	-------------------	-------------------	-------------------	0.031
0.001	Confidence	0.05	5	0.096
0.0001	Confidence	0.05	5	0.096

Task 3. Results of NDI algorithm.

Non-derivable itemsets are a significant concept in the field of data mining and association rule learning. These fields revolve around the analysis of extensive datasets to discover patterns and relationships. In many real-world scenarios, it is impossible to precisely calculate the support of an itemset based on its subsets. These itemsets are referred to as "non-derivable" and are of particular interest. They represent unexpected relationships in the data that cannot be inferred by examining smaller groups of items. For example, the frequent occurrence of bread, milk, and eggs together may not be immediately apparent when considering pairs of these items.

In this scenario, we were tasked with using a C++ implementation of the Non-Derivable-Itemsets (NDI) algorithm. To ensure a more comprehensive and robust comparison of results, I chose to employ the same train and test data as in Task 2. The analysis was conducted in a similar manner to Task 2, and the outcomes are presented in tables 5 to 7. It is evident from the results that the recall values are slightly worse, while the precision values exhibit similar behavior. The same trend is observed for the f1 score. However, it is important to note that the dataset used in this analysis is relatively small and may not accurately represent real-life situations. One potential improvement could involve obtaining a larger and more representative sample, followed by a parameter search to compare the results. The decision to use a smaller dataset was primarily driven by the lengthy runtime associated with the parameter search loop.

In conclusion, although the results of the NDI algorithm show slightly worse performance, it does not necessarily imply that the algorithm itself is inferior. The NDI algorithm offers the advantage of speed when dealing with lower support thresholds, which can be highly beneficial when handling extremely large datasets.

Table 5. Highest results by recall.

Min_support	Metric	Threshold	Top_n	Score
0.01	Support	0.0005	5	0.033
0.001	Conviction	0.500 / 1.000	5	0.170
~ 0.0001	Support	0.001	5	0.142

Table 6. Highest results by precision.

Min_support	Metric	Threshold	Top_n	Score
0.01	Confidence	0.1	1	0.103
0.001	Confidence	1	1	0.093
~ 0.0001	Support	0.005	5	0.050

Table 7. Highest results by f1 score.

Min_support	Metric	Threshold	Top_n	Score
0.01	Support	0.0005	5	0.067
0.001	Conviction	0.500 / 1.000	5	0.089
~ 0.0001	Support	0.001	5	0.074

Appendix

The project structure is as follows:

```
.
├── assignment_02_task1
│   ├── apriori_comparison.ipynb  → time comparison of algorithms
│   ├── apriori_fixed.py  → fixed version of GPT apriori algorithm
│   ├── apriori_raw.py  → raw version of GPT algo (no changes)
│   ├── apriori_split.ipynb  → jupyter version of apriori for small changes
│   ├── baskets.pkl  → dataset for comparison
│   └── helpers.py  → helper functions
├── assignment_02_task2
│   ├── Groceries.csv → file with all transactions
│   ├── baskets.pkl  →  contains all baskets
│   ├── baskets_by_user.pkl  → baskets purchased grouped by user
│   ├── encoded_baskets.dat  → data encoded as number for task 3 (ndi 2)
│   ├── groceries.dat  → file with all transactions grouped
│   ├── items_by_user.pkl → all items purchased by users
│   ├── preprocess.ipynb  → data preprocessing file
│   ├── recomm.ipynb  → experiments on algorithm metrics and thresholds
│   ├── test_labels.pkl  → data for testing and param search (also used in task3)
│   ├── test_set.pkl  → data for testing and param search (also used in task3)
│   └── train_set.pkl  → data for training (also used in task3)
└── assignment_02_task3
    ├── Groceries.csv
    ├── baskets.pkl
    ├── decoded_out_135_10.pkl  → decoded output of ndi 2 script (support ~ 0.01)
    ├── decoded_out_14_10.pkl  → decoded output of ndi 2 script (support ~ 0.001)
    ├── decoded_out_2_10.pkl  → decoded output of ndi 2 script (support ~ 0.0001)
    ├── encoded_baskets.dat
    ├── encoder_decoder.ipynb  → code to process basket data
    ├── ndi 2  → folder with c++ ndi scripts
    ├── out_135_10.txt  → encoded output of ndi 2 scripts (support ~ 0.01)
    ├── out_14_10.txt  → encoded output of ndi 2 scripts (support ~ 0.001)
    ├── out_2_10.txt  → encoded output of ndi 2 scripts (support ~ 0.0001)
    └── recommender.ipynb
```