Market Basket Analysis with FP-Growth & PCY Algorithms

Antoine Nasra

January 12, 2024

1 Introduction

Market Basket Analysis (MBA) is a data mining technique used to understand the purchase behavior of customers by uncovering associations between different items that customers place in their shopping baskets. The primary goal of MBA is to identify product groupings that frequently co-occur in transactions. This information can be pivotal for retail businesses in terms of shelf placement, promotional strategies, and product bundling. To efficiently conduct Market Basket Analysis, several algorithmic approaches have been developed. Among these, the FP-Growth (Frequent Pattern Growth) algorithm[1] and the PCY (Park-Chen-Yu) algorithm[2] are noteworthy for their efficiency and scalability.

2 Preprocessing

In order to handle the substantial size of the Yelp dataset, we adopted specialized data processing techniques, marking our foray into the realm of "Big Data". Initially, the dataset was segmented into chunks, each containing 100,000 records. We then proceeded to tokenize these segments. This tokenization process was carried out using the **nltk** library. Additionally, we eliminated stopwords, such as "I", "and", "or", etc., to refine our data further. To expedite this process, we employed the MapReduce model along with multiprocessing (utilizing 8 processes), significantly enhancing the efficiency of our workflow. The processed data was then stored in a separate file. This file, stripped of unnecessary columns from the original dataset, forms the core of our analysis. With this clean and pre-processed dataset in hand, we are now well-positioned to begin the actual implementation of our analytical algorithms.

3 FP-Growth Algorithm

The FP-Growth algorithm, a more efficient alternative to the classic Apriori algorithm, generates frequent item sets without the need for candidate generation. This is achieved through the use of a data structure known as an FP-tree,

which retains the itemset association but compresses the dataset, significantly reducing the number of scans required through the database.

Explanation of Algorithm Steps

1. **FPNode Class**: Represents a node in the FP-Tree. It contains the item, its count, its parent, and its children (defaulted to FPNode type).

2. **Construct_FP_Tree Function**: Builds the FP-Tree from transactions. It first counts the frequency of each item, filters out infrequent items based on min_support, and then iterates through each transaction to build the tree.

3. **Mine_Frequent_Itemsets Function**: Identifies frequent itemsets in the FP-Tree. It searches recursively through the tree, extending the suffix and checking if the new itemset meets the min_support threshold.

4. **Generate_Association_Rules Function**: Generates association rules from the frequent itemsets. For each itemset, it examines all possible combinations of antecedents and calculates the confidence of the rule, adding it to the list if it meets the min_confidence threshold.

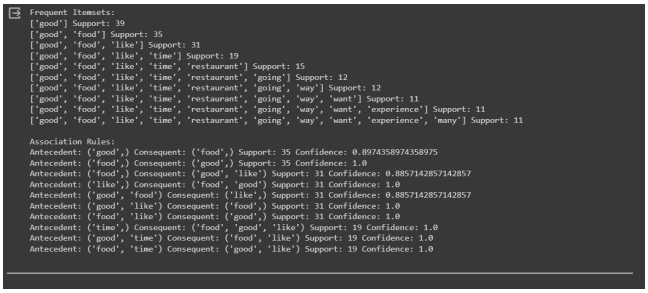

Figure 1: Results

The FP-Growth algorithm demonstrates high efficiency in finding frequent itemsets, especially in large datasets, due to its compact FP-Tree structure and the elimination of candidate generation. Its performance excels over the Apriori algorithm by significantly reducing database scans. However, the algorithm's reliance on recursive tree traversal can lead to high memory usage, particularly with complex or dense datasets. While FP-Growth is efficient for single-machine processing, its recursive nature and tree structure are not ideally suited for distributed computing paradigms like MapReduce, where parallel processing of independent data chunks is preferred. This limitation is evident in scenarios requiring horizontal scalability, as encountered in typical Big Data applications.

4 PCY Algorithm

PCY algorithm enhances the Apriori approach by utilizing additional memory layers for hashing and counting pairs, thereby reducing the number of candidate itemsets. This is particularly useful in handling large datasets, as it efficiently reduces the computational complexity.

Explanation of PCY Algorithm and Association Rule Generation

PCY Algorithm Implementation

1. **Initialization**: Sets up counting structures for individual items, identifies items meeting minimum support, and initializes a hash table for counting pairs and triplets.
2. **First Pass - Count Individual Items**: Iterates through each transaction, counting each item's occurrence, stored in `count_of_single_items`.
3. **Filtering Items by Minimum Support**: Items meeting or exceeding the minimum support threshold are identified and stored in `items_meeting_min_support`.
4. **Second Pass - Frequent Pairs and Triplets**: Another pass through the transactions generates and counts pairs and triplets of items, using a hash table for efficiency.
5. **Selecting Frequent Itemsets**: Itemsets that meet the minimum support and have adequate counts in the hash table are selected as frequent.

Support Calculation and Association Rules

1. **Calculate Itemset Support**: Determines the support of an itemset as the ratio of transactions containing that itemset.
2. **Generate Association Rules**: Generates rules from frequent itemsets. Iterates through each itemset, generating subsets as potential antecedents, and calculates confidence for each rule.
3. **Writing Results to Files**: Frequent itemsets and association rules are saved to separate text files for analysis.
4. **Displaying Results**: Prints a sample of association rules, showing antecedents, consequents, support, and confidence.

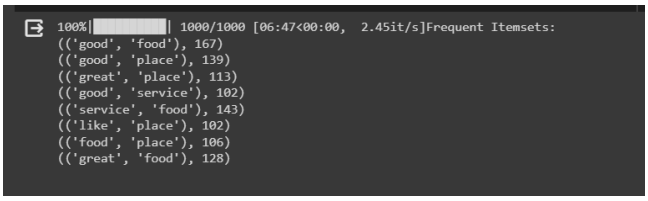

Figure 2: Frequent Itemsets

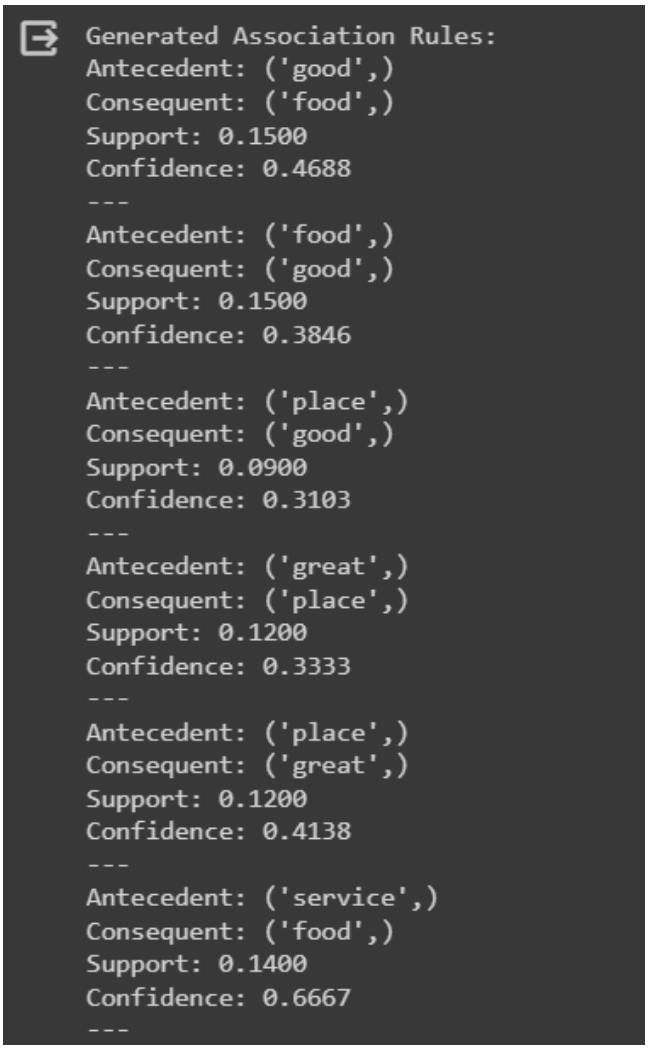

Figure 3: Association Rules

Key Points

- The PCY algorithm efficiently finds frequent itemsets in large datasets, particularly when the dataset is too large for memory.
- It uses hashing to count occurrences of item pairs and triplets effectively.
- The code includes a method to generate and evaluate association rules based on identified frequent itemsets.

PCY Algorithm with MapReduce

Importing Libraries

- `itertools.combinations`: For generating pairs and triplets from transactions.
- `numpy`: Used for efficient array operations, especially for hash tables.
- `multiprocessing.Pool`: Enables parallel processing.
- `tqdm`: Provides progress bars.
- `time`: To measure execution time.
- `pickle`: For loading and saving data.

Function Definitions

1. **map_function**:
    - Processes a chunk of transactions along with frequent items and hash table size.
    - Counts pairs, updates local hash table, and returns pair counts and the hash table.
2. **reduce_function**:
    - Aggregates results from map function.
    - Filters and combines frequent itemsets using global hash table.
    - Returns the frequent itemsets.
3. **run_PCY_algorithm**:
    - Executes the PCY algorithm.
    - Counts single items, splits transactions for parallel processing.
    - Performs map and reduce phases.
    - Returns frequent itemsets.

Main Execution Block

- Loads transactions from a file.
- Sets parameters for the PCY algorithm.
- Measures execution times for different subsets of transactions.
- Runs the algorithm and saves results to files.

Key Features of the Code

- **Parallelization**: Uses multiprocessing for the map phase.
- **Memory Efficiency**: Employs a hash table for counting pairs.
- **Performance Measurement**: Records execution times for scalability analysis.
- **Data Persistence**: Saves metrics and itemsets for further analysis.

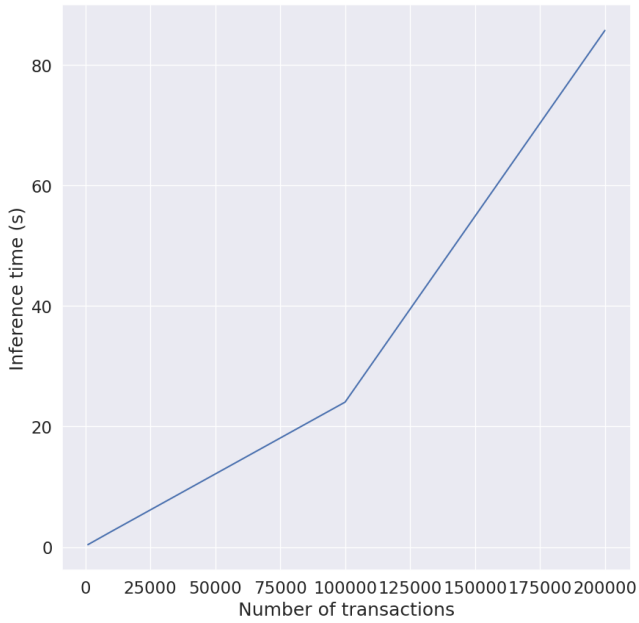

Figure 4: Execution

The PCY algorithm, an advancement over Apriori, stands out for its memory efficiency and scalability, primarily due to its use of a hash table for counting item pairs. This approach not only reduces the number of passes over the dataset but also improves overall performance, especially in large datasets with

many frequent items. However, the algorithm's efficacy is heavily reliant on the quality of the hash function used, which can pose challenges with hash collisions. Additionally, while it is more memory-efficient than Apriori, the PCY algorithm still requires substantial memory for the hash table and can be complex to implement, especially in distributed computing environments where parallelization is not straightforward.

5 Comparison Summary between FP-Growth and PCY Algorithms

Efficiency

FP-Growth is generally more efficient than PCY for datasets with a large number of transactions but fewer distinct items. PCY is more efficient when the dataset has a large number of potential item pairs.

Memory Usage

FP-Growth can require more memory for the FP-Tree, especially in dense datasets. PCY's memory usage is more predictable and is tied to the size of the hash table.

Implementation Complexity

Both algorithms have their complexities, with FP-Growth focusing on tree-based data structures and PCY on efficient hashing and counting.

Scalability and Parallelization

FP-Growth scales well with the number of transactions but less so with item count. PCY can scale efficiently but is dependent on hashing performance. PCY is generally easier to parallelize due to its hash-based approach.

6 Declaration

"I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study."

References

1. Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1-12). ACM.

2. Park, J. S., Chen, M. S., & Yu, P. S. (1995). An effective hash-based algorithm for mining association rules. In *Proceedings of the 21st International Conference on Very Large Data Bases* (pp. 208-219). Morgan Kaufmann Publishers Inc.