Mining Massive Data Sets Midterm Report

1st 522H0036 - Luong Canh Phong
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0036@student.tdtu.edu.com

2nd 522H0092 - Cao Nguyen Thai Thuan
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0092@student.tdtu.edu.com

3rd 522H0075 - Tang Minh Thien An
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0075@student.tdtu.edu.com

4th 522H0167 - Truong Tri Phong
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
522H0167@student.tdtu.edu.com

5th Instructor: Nguyen Thanh An
Faculty of Information Technology
Ton Duc Thang University
Ho Chi Minh City, Vietnam
nguyenthanhan@tdtu.edu.com

***Abstract*—In the age of big data, the ability to mine and extract valuable information from massive datasets can give the user an unparalleled edge against the competition. Therefore, this requirement made by the lecturer is designed to simulate one of the three most fundamental challenges in data mining. Through these series of tasks, we will explore some algorithm implementations and solve different problems as well as explore their trade-offs. Each task is a different algorithm to explore and implement with their corresponding datasets. Through these tasks, we will gain some practical insight and experience in working with these algorithms as well as a better understanding of their pros and cons to be able to cater to each dataset based on their characteristics.**

I. Introduction

This report is divided into three large sections corresponding to the first three tasks provided by the lecturer. We will explore and present our findings while putting the proposed algorithms into practice.

Task 1 proposes utilizing the A-Priori algorithm in a Hadoop MapReduce program to discover groups of customers shopping on the same date as well as interacting with Hadoop Distributes File System (HDFS) to store files. By applying these methods, we will be able to understand how to extract patterns from large datasets locally.

The second task focuses on implementing the Park-Chen-Yu (PCY) algorithm using Object-Oriented Programming (OOP) principles and PySpark DataFrame to identify frequent item pairs and generate association rules from customer purchase data stored in Google Drive. The implementation, while generating association rules, also has to follow object-oriented programming principles inspired by PySpark's Frequent-Pattern Growth (FPGrowth) class.

In the third task, we will implement and compare the MinHashLSH algorithm and an alternative of our choice - in this case, a manual method of calculating Jaccard distance. Both of these approaches should achieve the same goal of searching for similar pairs of dates where the Jaccard distance is above a predetermined threshold. After that, we will visualize their runtime with their threshold ranging from 0 to 1 with 0.1 increments to gauge their performance and outline some characteristics between both approaches. Through these implementations, we demonstrate practical applications of data mining techniques with a given dataset. With these findings, we highlight the trade-offs between various aspects across different algorithms within the given time and constraints.

II. First Task: A-Priori Algorithm for Frequent Customers

A. Overview of A-priori Algorithm

1) What is the A-Priori Algorithm: The A-Priori algorithm is a classic algorithm in data mining used to identify frequent itemsets and derive association rules. In the context of this project, it is used to discover pairs of customers who frequently shop together.

- It is based on the principle that all subsets of a frequent itemset must also be frequent.
- It uses a level-wise, breadth-first search approach to count the frequency of itemsets.
- It is typically implemented in multiple passes: the first pass finds frequent individual items (1-itemsets), and subsequent passes find larger frequent itemsets (e.g., 2-itemsets, 3-itemsets, etc.).

2) How the A-Priori Algorithm Works (in this task):

- **First Pass – Frequent Individual Customers:** Count how many times each individual customer appears in the grouped transaction data. Customers with frequency above a defined threshold are considered frequent.
- **Second Pass – Frequent Customer Pairs:** For each transaction (i.e., group of customers on a given date), generate all possible customer pairs where both customers are frequent. Count the number of times each pair appears.
- **Support Threshold:** A predefined threshold used to filter out itemsets (customers or pairs) that do not appear frequently enough to be considered relevant.
- **Output:** The algorithm outputs customer pairs that occur together at or above the support threshold.

Note: The output of the first subtask (Customer Grouping by Date) will serve as the input for both passes of the second subtask (A-Priori Algorithm):

- First Subtask (Customer Grouping by Date):
  - `args[0]` – Input path to the raw transaction CSV file.
  - `args[1]` – Output path where grouped customer data by date will be written.
- Second Subtask (A-Priori Algorithm):
  - `args[0]` – Input path to the grouped customer data (output from the first subtask).
  - `args[1]` – Output path for the first pass (frequent individual customers).
  - `args[2]` – Output path for the second pass (frequent customer pairs).

This approach allows the workflow to seamlessly transition from the first subtask (grouping by date) to the second subtask (identifying frequent customers and pairs).

B. First subtask

In the first subtask, we are assigned to store the data on HDFS. After which we will implement a Hadoop MapReduce program in Java to discover groups of customers going shopping at the same date.

1) The Mapper Class: CustomerGroupByDateMapper

The Mapper class is responsible for reading the input data and emitting key-value pairs. Key aspects of its implementation include:

- Input Processing: The input is a CSV file where each line contains multiple fields, including `Member_number` (customer ID) and `Date` (transaction date).
- Filtering Headers: The Mapper skips header lines by checking if the first token equals `Member_number`.
- Emitting Key-Value Pairs: The key is the transaction date, and the value is the customer ID. This allows transactions to be grouped by date in the shuffle and sort phase.

Example Output from Mapper:

(01/01/2014, 11111)
(01/01/2014, 22222)
(02/01/2014, 11111)

2) The Reducer Class: CustomerGroupByDateReducer

The Reducer aggregates the values emitted by the Mapper for each unique date. Key implementation features include:

- Collecting Unique Customers: Customer IDs are added to a `HashSet` to remove duplicates.
- Joining Values: The set of unique customer IDs is converted to a comma-separated string.
- Emitting Results: The final output consists of the transaction date as the key, and a list of unique customer IDs as the value.

Example Output from Reducer:

(01/01/2014, 11111,22222)
(02/01/2014, 11111)

C. Second subtask

In the second subtask, we implement the A-Priori algorithm to identify frequent customer pairs. This is achieved using two MapReduce passes.

1) The First Pass: Identifying Frequent Individual Customers

- Mapper Class: *AprioriFirstPassMapper*
  - Function: Reads grouped customer data (output of first subtask), splits the customer list, and emits each customer ID with a value of 1.
  - Input Format: Each line is a tab-separated pair where the key is a date and the value is a comma-separated list of customers.
  - Filtering: Skips malformed lines where the customer list is missing.
  - Example Input:

    01/01/2014 12345,67890

  - Example Output from Mapper:

    (12345, 1)
    (67890, 1)
- Reducer Class: *AprioriReducer*
  - Function: Aggregates the counts for each key (customer).
  - Filtering: Emits only `<key, value>` pairs whose frequency is greater than or equal to the support threshold.
  - Example Output from Reducer (First Pass):

    (12345, 2)
    (67890, 1)

2) The Second Pass: Identifying Frequent Customer Pairs

- Mapper Class: *AprioriSecondPassMapper*
  - Setup: Loads the list of frequent customers from the first pass output using Hadoop's distributed cache.
  - Processing: For each transaction line, splits the list of customer IDs, filters only frequent customers, and generates all valid customer pairs.
  - Emitting: Outputs each pair of frequent customers with a count of 1.
  - Example Output from Mapper:

    (12345,67890, 1)
    (12345,54321, 1)
- Reducer Class: *AprioriReducer*
  - The same class as used in the First Pass.
  - Example Output from Reducer (Second Pass):

    (12345,67890, 3)

III. Second Task: PCY Algorithm for Frequent Items

A. Overview of PCY

Frequent itemsets mining is essential for discovering item associations in transactional data, such as market basket analysis. The PCY algorithm improves efficiency by using hash buckets to reduce the computational cost of finding frequent

item pairs. This project applies the PCY algorithm to mine frequent itemsets and generate association rules based on support and confidence thresholds, using PySpark for scalable data processing.

The PCY algorithm is based on two key passes through the data. In the first pass, frequent individual items are identified and counted. In the second pass, frequent item pairs are counted, and hash buckets are used to prune less frequent pairs. The hash function maps item pairs to buckets, and only pairs that have a sufficient bucket count are considered frequent. This approach significantly reduces the number of pair comparisons and improves algorithm efficiency.

- Step 1: Count individual items using the support threshold.
- Step 2: Count pairs of frequent items and hash them into buckets.
- Step 3: Prune item pairs that are not frequent based on the bucket counts.
- Step 4: Generate association rules using the confidence threshold.

B. Implementation Details

1) Data Loading and Preprocessing: Data is loaded using PySpark's `read.csv` function. Each transaction is represented as a basket, and the data is grouped by customer and date. The `collect_set` function is used to create a list of items bought together in each transaction.

2) First Pass: Counting Frequent Items: In the first pass, each item's frequency is counted, and only those items that meet the support threshold are considered frequent. The item counts are stored in a dictionary, sorted in descending order.

3) Second Pass: Counting Frequent Pairs: During the second pass, the algorithm generates pairs from frequent items and counts their occurrences. Hashing is applied to map pairs into buckets, and the bucket counts are used to prune pairs that do not meet the minimum support threshold.

4) Association Rule Generation: For each frequent pair, confidence is calculated as the ratio of the pair's count to the individual item count. Association rules are generated if the confidence meets the given threshold. The rules are sorted by confidence.

C. Experimental Results

The PCY algorithm was executed on a transactional dataset of retail transactions, where each transaction (basket) represented a set of items purchased by a customer. The algorithm was applied with the following parameters:

- Support threshold: 2 (minimum count for items to be considered frequent).
- Confidence threshold: 0.5 (minimum confidence for association rules).

1) Frequent Items: The first pass of the algorithm counted the occurrence of each item in the transactions. The following are the top 30 frequent items identified based on the support threshold:

TABLE I
MOST FREQUENT ITEMS WITH THEIR COUNTS

Item	Count
Whole milk	2363
Other vegetables	1827
Rolls/buns	1646
Soda	1453
Yogurt	1285
Root vegetables	1041
Tropical fruit	1014
Bottled water	908
Sausage	903
Citrus fruit	795
Pastry	774
Pip fruit	734

2) Frequent Item Pairs: In the second pass, the algorithm counted pairs of frequent items across all transactions. The top 30 frequent item pairs, sorted by frequency, are as follows:

TABLE II
FREQUENT ITEM PAIRS WITH THEIR COUNTS

Pair	Count
('Whole milk', 'Other vegetables')	222
('Whole milk', 'Rolls/buns')	209
('Whole milk', 'Soda')	174
('Whole milk', 'Yogurt')	167
('Rolls/buns', 'Other vegetables')	158
('Soda', 'Other vegetables')	145
('Whole milk', 'Sausage')	134
('Whole milk', 'Tropical fruit')	123
('Yogurt', 'Other vegetables')	121
('Rolls/buns', 'Soda')	121
('Yogurt', 'Rolls/buns')	117
('Whole milk', 'Root vegetables')	113

3) Association Rules: Once frequent item pairs were identified, association rules were generated based on the confidence threshold of 0.5. The confidence for each rule was computed by dividing the pair count by the count of the antecedent item. The top 30 association rules, sorted by confidence, are presented below:

TABLE III
VALIDATED ASSOCIATION RULES

Rule	Confidence
Preservation products $\rightarrow$ Soups	1.00
Kitchen utensil $\rightarrow$ Pasta	1.00
Kitchen utensil $\rightarrow$ Bottled water	1.00
Kitchen utensil $\rightarrow$ Rolls/buns	1.00
Bags $\rightarrow$ Yogurt	0.50

4) Performance Evaluation: The PCY algorithm effectively identified frequent items, pairs, and association rules with reasonable execution time and memory usage. Given the dataset size, the distributed nature of Spark ensured that the computation was scalable.

- Time Complexity: The use of hashing significantly reduces the complexity of item pair generation, making the PCY algorithm faster than traditional algorithms such as the Apriori algorithm.
- Memory Usage: Memory usage was managed well by leveraging the distributed processing capabilities of Spark.

IV. Third Task: MinHashLSH for Similar Dates

Firstly, we will go through the theoretical basis and its possible implementation in the context of our task, after which, we will see the algorithm in action.

A. Theoretical MinHashLSH

The core of MinHashLSH algorithm is the utilization of two concepts: MinHash signatures and locality-sensitive hashing (LSH) to create and effective algorithm for detecting similar sets. The first concept describes the process of hashing the dataset into a more manageable "signature" for each set. While the latter describes how these "signatures" will be stored to achieve the expected result.

1) Jaccard distance and Shingling: Before performing any calculations to any data, we must convert the raw data into a distinguished vector before using any distance calculation between the two sets to check for their similarity. Shingling performs this task by breaking down text data into smaller units to create "shingles" before hashing them into their representation in the form of a binary vector.

The Jaccard distance describes the similarity between two different sets and is represented by the following formula:

$$d_J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} = 1 - J(A,B) \quad (1)$$

With the result ranging from 0 to 1, we can determine whether the sets in question are related to each other or not for bucket assignment.

2) MinHash Signature: To determine if the pairs are similar or not, we need a way to convert the raw data into usable data for the algorithm to process. In this case, MinHashing is used to perform this task. The binary vector, created through a process of shingling, is converted into a signature vector. Note that if these sets 2 are similar, their signature vector will also have some similarities, and these can be utilized by the algorithm to sort these sets into their suitable buckets.

3) Locality Sensitive Hashing (LSH): The idea of LSH when dealing with the problem of finding similar pairs or sets is to maximize the probability of collision in a bucket due to that fact that the hash-code for these sets would be indifferent (if these sets are the same) and therefore they should be in the same bucket. We can achieve this by breaking the hash-code down even more into subsequences of hash-code that have a higher chance of being similar, giving us a higher chance of finding similar pairs, improving the effectiveness of the algorithm at solving the problem.

B. Optimized Manual Jaccard

In the optimized manual Jaccard method, a broadcast join is used when one DataFrame is much smaller than the other, improving performance. The Jaccard similarity is computed for each pair using a User-Defined Function (UDF).

Optimized Manual Jaccard: Instead of performing a complete cross-join, we use a broadcast join:

- Apply the broadcast join between the smaller DataFrame and the larger one.
- Calculate Jaccard similarity using a UDF.
- Filter pairs based on the given similarity threshold.

With a manual method of calculating the Jaccard distance for each pair, we can utilize this function in our own Brute Force (BF) MinHashLSH algorithm

C. Experimental Results

We evaluated both methods across various similarity thresholds. Table IV presents the execution times and number of pairs processed for each method.

TABLE IV
COMPARISON OF MINHASHLSH AND BRUTE FORCE ACROSS THRESHOLDS

Threshold	LSH Time (s)	BF Time (s)	LSH Pairs	BF Pairs
0.0	33.83	13.21	218216	264628
0.1	36.09	13.30	218158	264520
0.2	35.37	12.55	202905	244672
0.3	21.93	13.08	91729	105487
0.4	12.46	13.04	8462	10113
0.5	11.19	13.19	132	223
0.6	10.79	23.84	2	3
0.7	10.98	21.78	0	0
0.8	10.89	24.46	0	0
0.9	9.25	23.88	0	0
1.0	10.77	21.68	0	0

As you can refer from Table IV and subsequent graph, there is a distinct difference between both algorithms' performance. For the traditional MinHashLSH implemented from the PySpark library, its runtime has a significant decrease as the threshold passes the 0.3 mark, taking up only one third of the time when compared to the runtime of a higher threshold. Meanwhile, our Brute Force function while performing remarkably well at the first few increments of thresholds, it struggles to retain its performance when tasked to do such a task with a higher threshold. All in all, despite having opposite trends, the BF function returns a higher efficiency with lower runtime and higher yield of frequent pairs. This can be attributed to the fact that the BF function uses a different merging method with utilization of broadcast() function to optimize traffic, giving our BF function a better runtime on paper

D. Conclusion

The optimized manual Jaccard method provides exact results but is computationally expensive, especially for large datasets. In contrast, MinHashLSH is faster and works well for

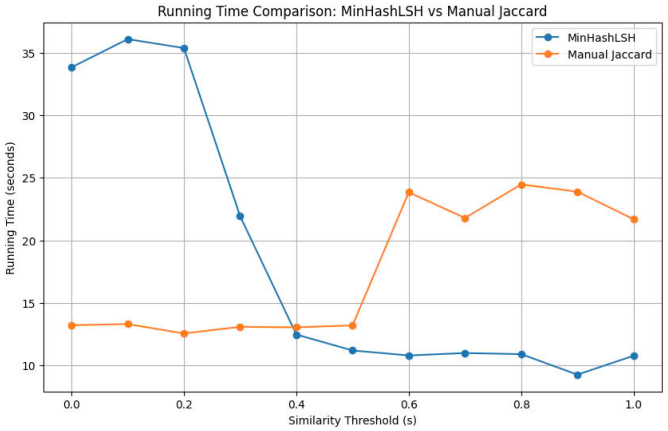

Fig. 1. Comparison of Runtime with Incremental Thresholds

applications where approximate results are acceptable. MinHashLSH is more suitable for performance-critical systems that require fast response times.

V. Contribution

The following table represents the contribution of each member, note that whichever member handles whichever task will also write the report for that task.

TABLE V
MEMBERS CONTRIBUTIONS

ID	Member	Contribution	Progress
522H0036	Luong Canh Phong	Task 1 and Handling Report	100%
522H0092	Cao Nguyen Thai Thuan	Overseer and Report Support	100%
522H0075	Tang Minh Thien An	Task 3	100%
522H0167	Truong Tri Phong	Task 2	100%

VI. Self-evaluation

The following table is our self-evaluation on our tasks:

TABLE VI
SELF-EVALUATION

Task	Task Requirements	Completion Ratio
Task 1	A-Priori Algorithm for Frequent Customers	100%
Task 2	PCY Algorithm for Frequent Items	95%
Task 3	MinHashLSH for Similar Dates	90%
Task 4	Report	100%

VII. Conclusion

We have gone through a variety of techniques and algorithms used in the world of data mining. For the first task, we have to find same-day customers and utilize the A-Priori algorithm to find frequent pairs of customers that shop on the same date and save the output of each pass in a dedicated folder. As we run though the code, the result after sorting is a reasonable ascending list of frequent customer pairs. For the second task, store the given dataset locally and identify baskets, as well as implementing the PCY algorithm to find frequent pairs along with generating metadata with predetermined constraints, the results for this task are two separate lists, one containing all frequent pairs, and the other is a list of association rules based on the user's given support threshold and confidence value. And finally, implement and compare between a traditional and an alternative MinHashLSH function to understand and have a greater insight into how the frequent pairs searching is done. We can see that with a slight modification and a different way of merging, it can result in a notably higher efficiency and better results.

References

[1] Tpoint Tech, "Apriori Algorithm, "
[Online]. Available: https://www.tpointtech.com/apriori-algorithm
[2] Databricks, "MapReduce, " Databricks Glossary, 2025.
[Online]. Available: https://www.databricks.com/glossary/mapreduce
[3] J. S. Park and M. S. Chen, "Using a hash table to eliminate candidates in a frequent itemset mining algorithm, " *IEEE Trans. Knowl. Data Eng.*, vol. 7, no. 3, pp. 464–472, 1995.
[4] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation, " *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, 2000.
[5] PySpark Documentation, "PySpark API Documentation, " 2025.
[Online]. Available: https://spark.apache.org/docs/latest/api/python
[6] PySpark Documentation, "pyspark.ml.feature.MinHashLSH, " Apache Spark, 2025.
[Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.MinHashLSH
[7] Amazon Web Services, "Jaccard similarity, " AWS Neptune Analytics Documentation, 2024.
[Online]. Available: https://docs.aws.amazon.com/neptune-analytics/latest/userguide/jaccard-similarity.html