Changhe You, Zekai Lin

Homework Report - HW2 Discovery of Frequent Itemsets and Association Rules

ID2222 HT25 Data Mining

Lecturer: Vladimir Vlassov, Sarunas Girdzijauskas and Ahmad Al-Shishtawy

2025

Homework Report - HW2 Discovery of Frequent Itemsets and Association Rules

Changhe You, Zekai Lin

16th November 2025

1 Dataset and How to Run

- **Dataset:** `T10I4D100K.dat`, privided on assignment page.
- **Environment:** PySpark, tested in a Jupyter Notebook with a local Spark session.
- **How to Run:**
  1. Define code 'Block 1' (Helpers), 'Block 2' (A-Priori), 'Block 3' (Rules) and 'Block 4'(Main) in separate Jupyter cells.
  2. Set parameters (`min_support`, `min_confidence`, `input_file`, `output_dir`) at the top of the 'Block 4' (main) cell.
  3. Call `main()` in a final cell to execute the job. Results are printed to the console and saved to `output_dir`.

2 A-Priori Algorithm Implementation

This module finds all frequent itemsets with support $\geq s$, following the A-Priori pass-by-pass logic (L_{k-1} to L_k).

Pass 1 (k=1): $C_1 \rightarrow L_1$

- **Count:** `flatMap` transforms baskets into `(item, 1)` pairs.
- **Reduce:** `reduceByKey` aggregates counts for each item (support of C_1).
- **Filter:** Keeps items with `count >= min_support` to generate L_1.
- The L_1 itemsets are collected to the driver and broadcasted for the next pass.

Pass 2 (k=2): $L_1 \rightarrow L_2$

A specialized `flatMap` path (`generate_k2_pairs`) is used to efficiently count pairs:

- **Generate (in `flatMap`):** For each basket, the function filters it using the broadcasted L_1 set, keeping only frequent items.
- It then generates all 2-item combinations (pairs) only from this smaller, pre-filtered list and yields `(pair, 1)`.
- **Reduce & Filter:** `reduceByKey` aggregates the pair counts, and a `filter` operation keeps those $\geq$ `min_support` to generate L_2.

Pass k (k $\geq$ 3): $L_{k-1} \rightarrow C_k \rightarrow L_k$

For $k = 3$ and higher, the standard "Join-Prune-Count" workflow is used:

- **Join:** On the Driver, `connect_sets(L_{k-1}, k)` joins L_{k-1} with itself (using a k-2 prefix-join) to generate C_k.
- **Prune:** For *each* candidate in C_k, `has_infrequent_subset` checks if all its (k-1)-subsets are present in L_{k-1}. If any subset is missing, the candidate is "pruned" (discarded).
- **Count:** The final, pruned C_k set is broadcasted. A `flatMap` scans the `transactions_rdd`, checking `candidate.issubset(basket)` for each candidate.
- **Filter:** Counts are aggregated (`reduceByKey`) and filtered ($\geq$ `min_support`) to produce L_k. This loop repeats until L_k is empty.

3 Association Rule Generation (Bonus Task)

This module generates high-confidence rules from the frequent itemsets found in Step 1.

Preparation: Broadcasting Support

- All frequent itemset RDDs ($L_1, \ldots, L_k$) are combined into one `all_frequent_rdd` using `sc.union`.
- `collectAsMap()` creates a Python dictionary (hash map) on the Driver: `{frozenset: support_count}`.
- This `support_map` is **broadcast** to all Executors as a read-only lookup table.

Rule Generation: Iterative Pruning

We call `flatMap(generate_confident_rules)` on all frequent itemsets I (where $k \geq 2$). This function executes an A-Priori-style logic for each itemset I:

- **Pass 1 (Consequent size = 1):** The function tests all rules with one item in the consequent (e.g., $ABC \rightarrow D$). It calculates confidence using the broadcasted `support_map`: $\text{conf} = \frac{\text{support\_map}[ABCD]}{\text{support\_map}[ABC]}$.
- If conf $\geq c$, the rule is saved, and its consequent `{D}` is stored in a list H_1 (high-confidence consequents).
- **Pass k (Consequent size = k $\geq$ 2):** The function **Joins** H_{k-1} to generate candidate k-consequents (e.g., `{C,D}`). It then **Prunes** these candidates. Before testing $AB \rightarrow CD$, it checks if its sub-consequents (`{C}` and `{D}`) are present in H_{k-1}. This pruning step ensures that non-confident rules are never computed.

4 Results and Analysis

- **Environment:** MacBook Pro M1, 16GB RAM, Spark local

Experiment 1: s = 1000 (1% Support), c = 0.7

- **Total Runtime:** 6.4 seconds
- **Analysis:** At a high support threshold, the algorithm runs very quickly. The sizes of L_1, L_2, and C_3 are small, so no significant bottlenecks appear.
- **Result Statistics:**

label=– Total Frequent Itemsets found: 385

label=– Total Association Rules found: 3

- **Sample Output:**

```
       --- Spark Application Started ---
Parameters: Min Support (s) = 1000, Min Confidence (c) = 0.7
Input File: T10I4D100K.dat

Total Transactions: 100000
--- Running A-Priori Pass 1 ---
--- Running A-Priori Pass 2 ---

--- Running A-Priori Pass 3 ---
--- Running A-Priori Pass 4 ---
--- Pass 4: No candidates generated after pruning. A-Priori terminating. ---
--- A-Priori Finished ---
--- A-Priori found 385 frequent itemsets ---

--- Printing Top 50 Frequent Itemsets (by support) ---
['368']: 7828
['529']: 7057
['829']: 6810
['766']: 6265
['722']: 5845
['354']: 5835
['684']: 5408
['217']: 5375
['494']: 5102
    ... ...
['12']: 3415
['895']: 3385
['795']: 3361
['510']: 3281
['598']: 3219
['75']: 3151
['487']: 3135
['614']: 3134

--- Starting Association Rule Generation ---
--- Found 3 high-confidence association rules ---
```

```
--- Printing Top 50 Association Rules (by confidence) ---
['704', '825'] -> ['39'] (Support=1035, Confidence=0.9392)
['39', '704'] -> ['825'] (Support=1035, Confidence=0.9350)
['39', '825'] -> ['704'] (Support=1035, Confidence=0.8719)

--- Job Finished ---
```

5 Conclusion

This project successfully implemented the A-Priori algorithm and association rule generator in PySpark.

The implementation showed that the A-Priori logic, including the k=2 optimization and the k-consequent pruning for rules, was effectively mapped to Spark's RDD transformations and broadcast variables.

The analysis also confirmed that the algorithm's performance is critically dependent on the `min_support` threshold. A low `s` value can cause bottlenecks to shift from C_2 (which we optimized for) to later passes (C_3), causing the Pass 3 counting step to become the new bottleneck.