Assignment 3 Report

Team 51

Muskan Raina (2021101066)

Arghya Roy (2021115008)

Recommendation System Approach

Apriori Algorithm for Frequent Itemsets

The recommendation system uses the Apriori algorithm to identify frequent itemsets (groups of movies often watched together). The algorithm finds sets of items (in this case, movies) that frequently appear together in transactions (user's watched movie lists).

- Step 1: Frequent 1-itemsets
  - The algorithm starts by finding frequent 1-itemsets, which are individual movies that appear in user transactions more than the `min_support` threshold.
  - It counts how often each movie appears, and movies with support (frequency / total transactions) above the threshold are considered frequent.
- Step 2: Generate Candidate Itemsets:
  - The function `generate_candidates` creates larger itemsets (combinations of frequent items found in the previous step).
- Step 3: Prune and Filter:
  - Itemsets that aren't frequent are pruned using the function `prune_candidates`.
- Step 4: Get Frequent Itemsets:
  - The algorithm generates frequent itemsets of increasing size until no more frequent itemsets can be found.

The result is a collection of frequent itemsets, which are sets of movies that users tend to watch together.

Association Rule Generation

Once frequent itemsets are found, the system generates association rules to make movie recommendations. Each rule has the form:

- Antecedent (X) → Consequent (Y), meaning that if a user has watched X, they are likely to watch Y.

The function `generate_rules` creates these rules by:

- For each frequent itemset with more than one movie, it generates rules where:
  - The antecedent (X) is a single movie (as mentioned in the question).
  - The consequent (Y) is the rest of the movies in the itemset.
- For each rule, it calculates:
  - Support
    The fraction of transactions that contain the entire itemset (both antecedent and consequent).
  - Confidence
    The likelihood that a user who watches the antecedent (X) will also watch the consequent (Y), calculated as the ratio of the support of the whole itemset to that of the antecedent.

The system filters rules based on a `min_conf` (minimum confidence threshold) to ensure only strong recommendations are kept.

Ranking and Selecting Top Rules

To make effective recommendations, the rules are sorted based on their **support** and **confidence** metrics:

- **Top 100 by support**: These rules are the ones that apply to the largest number of users (high frequency).
- **Top 100 by confidence**: These rules have the highest probability of the consequent being watched after the antecedent.

The function `top_rules_by_metric` helps select the top rules based on these criteria.

Finally, the system writes the top 100 association rules to two files:

- `51_top100RulesBySup.txt`: Contains the top 100 rules sorted by support.
- `51_top100RulesByConf.txt`: Contains the top 100 rules sorted by confidence.

Note

The system reads the movie titles from a file (`movies.csv`) using the `load_movie_titles` function. This maps each movie's ID to its actual title, allowing for more readable output. The function `movie_ids_to_titles` converts itemsets of movie IDs to sets of movie titles.

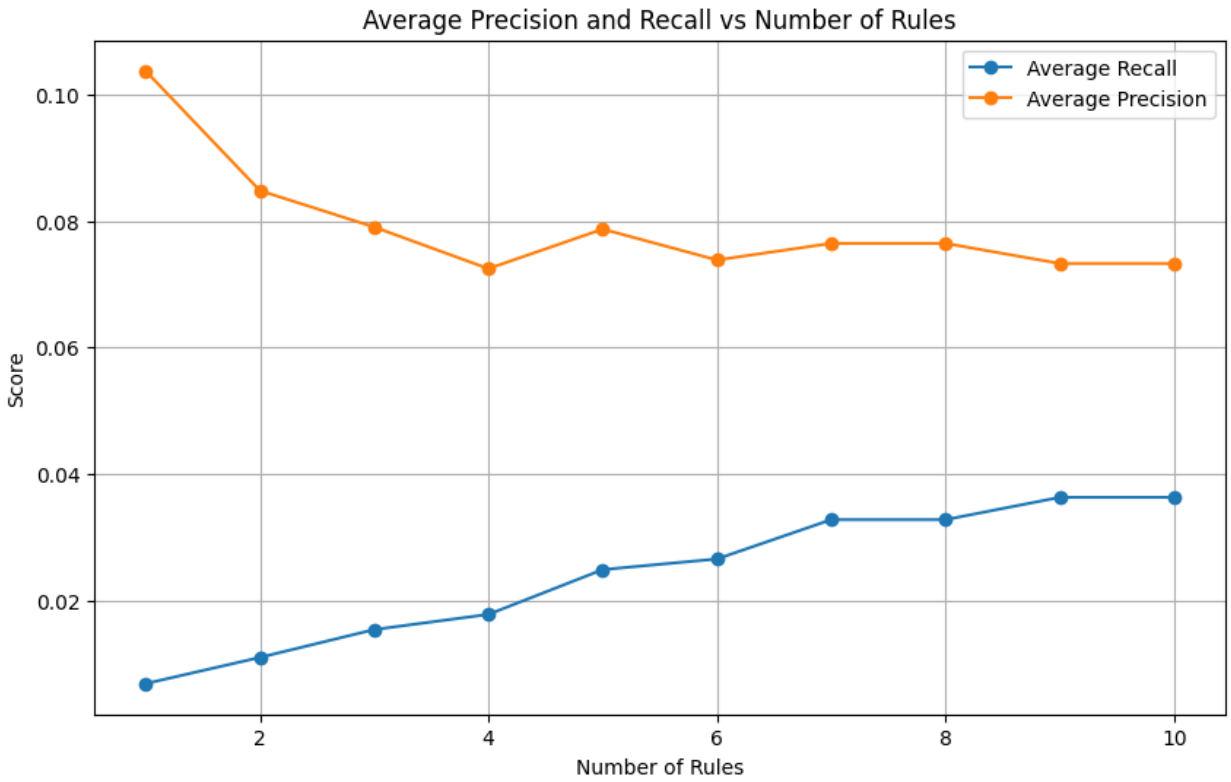
Selection of Algorithm

This recommendation system suggests movies that are frequently watched together and provides a confidence score indicating how likely a user who watched one movie will watch another.

- Apriori Algorithm is well-suited for this problem as it efficiently identifies common patterns (frequent itemsets) in large transactional datasets.
- The algorithm scales well with sparse data, which is typical in recommendation scenarios where users rate only a small fraction of available items.
- Its ability to generate rules based on frequent co-occurrences of items (movies) allows for recommending items based on prior user preferences.

Relevant Plots and Interpretations

Average



Precision

- The precision is highest at the first rule but decreases steadily as the number of rules increases.
- This happens because as you add more rules, the likelihood of generating non-relevant recommendations increases. The system becomes more "inclusive," but many of the new recommendations might not be as relevant, thus lowering precision.

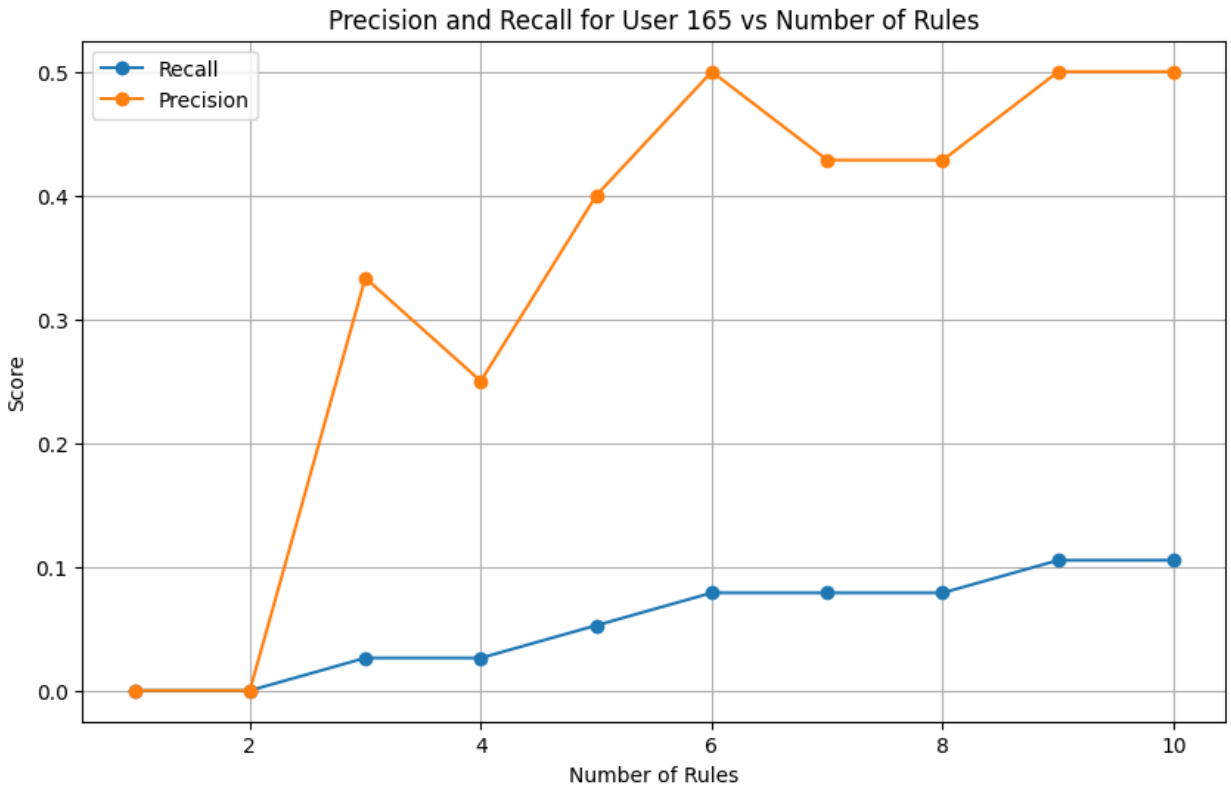
Recall

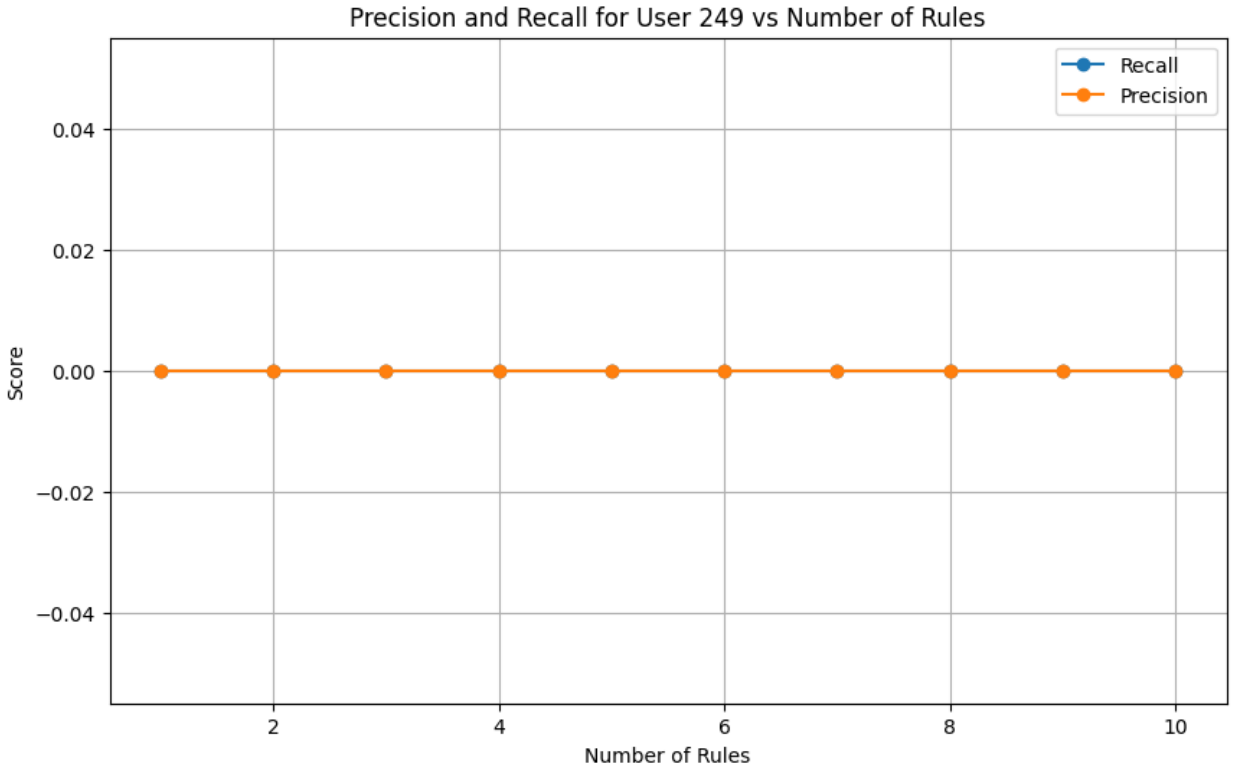
- The recall increases gradually with the number of rules. As more rules are considered, the system recommends more items, increasing the chance of hitting relevant ones.
- The recall score rises because with more rules, the system is capturing more relevant items (even if it also adds non-relevant ones).

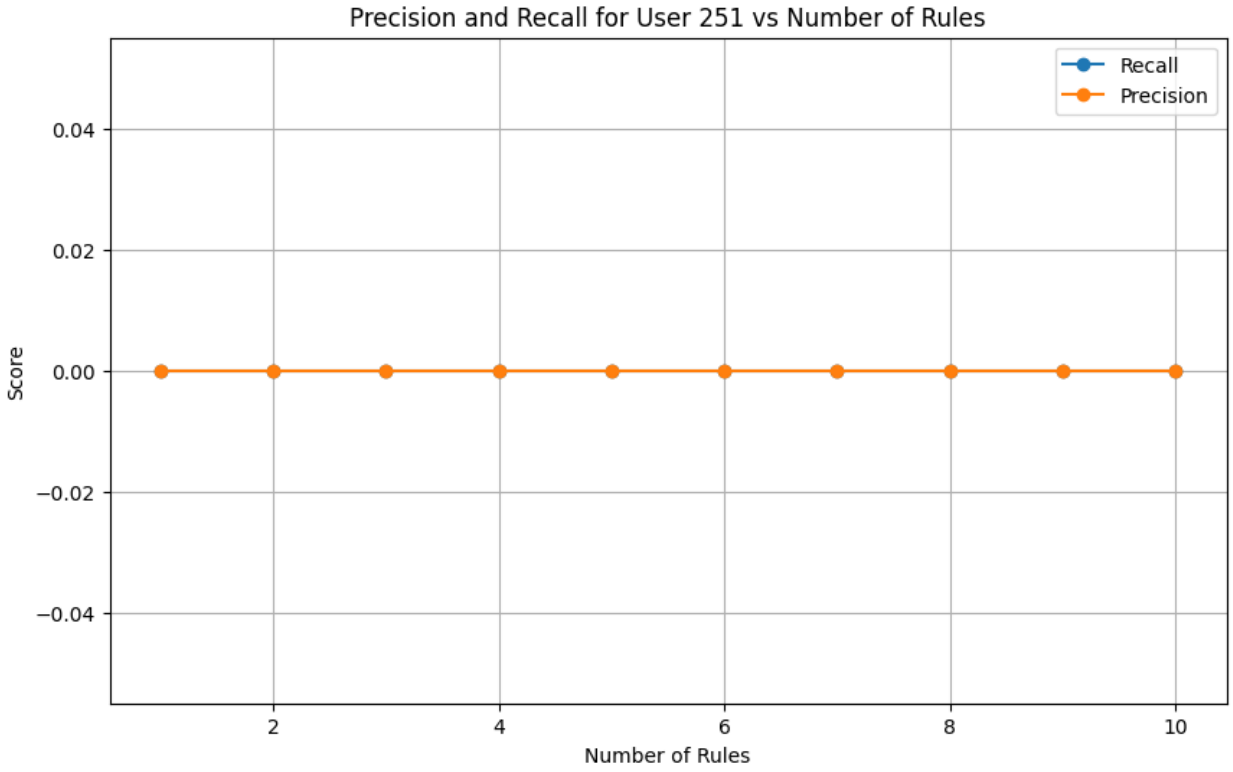
The trade-off between precision and recall is a common behavior in recommendation systems: adding more rules increases coverage (recall) but

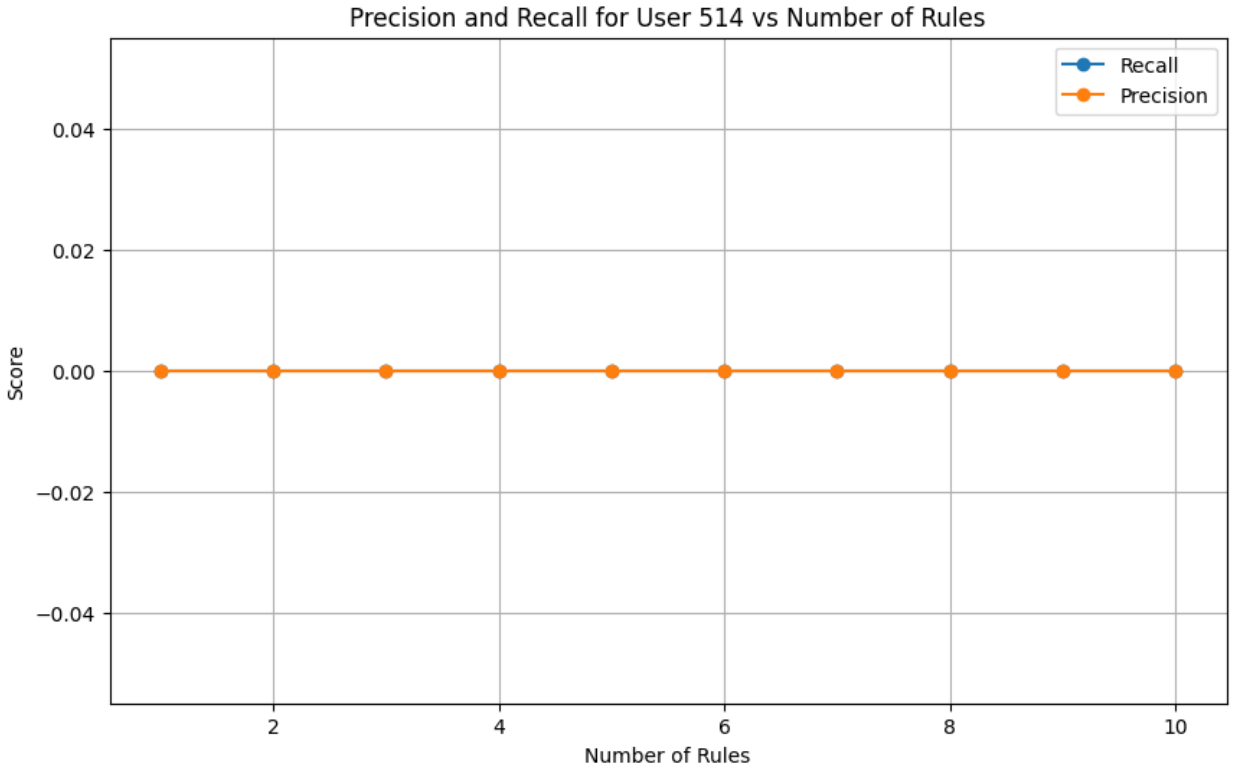
decreases the relevance of the recommendations (precision).

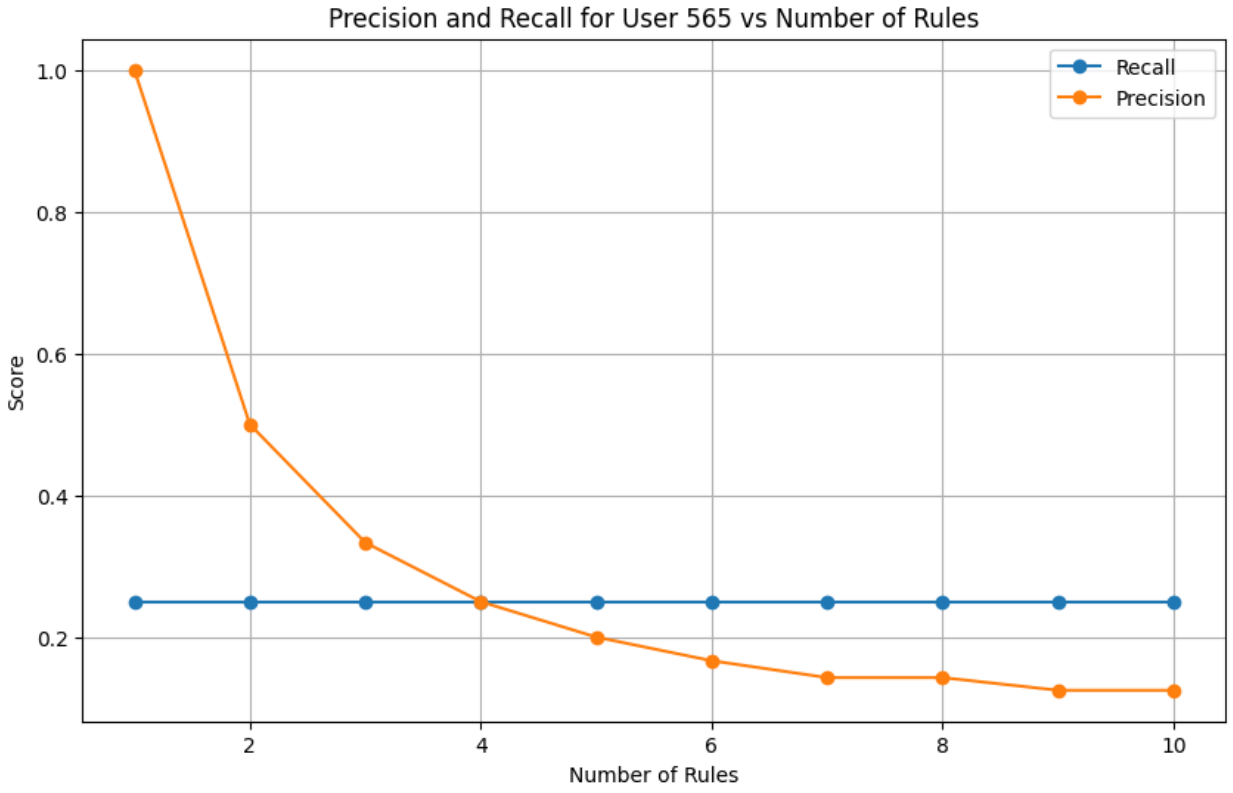
User-wise




Precision and Recall for User 249 vs Number of Rules
Recall
Precision
0.04
0.02
0.00
−0.02
−0.04
Score
2
4
6
8
10
Number of Rules


Precision and Recall for User 251 vs Number of Rules
Recall
Precision
0.04
0.02
0.00
−0.02
−0.04
Score
2
4
6
8
10
Number of Rules


Precision and Recall for User 514 vs Number of Rules
Recall
Precision
0.04
0.02
0.00
−0.02
−0.04
Score
2
4
6
8
10
Number of Rules


Precision and Recall for User 565 vs Number of Rules
Recall
Precision
1.0
0.8
0.6
0.4
0.2
Score
2
4
6
8
10
Number of Rules

Observations:

- The stark differences between the graphs indicate significant variability in how the system performs for different users. This suggests that user characteristics or behaviors have a substantial impact on system performance.
- As the number of rules increases, we don't see a consistent linear improvement in precision or recall across all users. This implies that simply adding more rules does not guarantee better performance for all users.
- The varied patterns across users highlight the potential need for personalized approaches in rule application or system optimization, rather than a one-size-fits-all strategy.

Overall, these graphs reveal that the system's performance is highly context-dependent and user-specific, suggesting that effective optimization and deployment may require sophisticated, adaptive, and possibly personalized approaches.
