Data Analytics

Assignment 3 - Association Rule Mining

TEAM - 3

Abhinav Reddy Boddu – 2021101034
Gowlapalli Rohit – 2021101113

Justification for Choosing FP-Growth

- **Efficiency**: The FP-Growth approach is generally more efficient than Apriori, particularly for large datasets. It reduces the need for repeated database scans, making it faster for frequent pattern mining.
- **Reduced Candidate Generation**: Apriori generates a large number of candidate itemsets, many of which may not be frequent. In contrast, FP-Growth significantly reduces candidate generation by compactly storing transactions in a tree structure.
- **Memory Efficiency**: FP-Growth is often more memory-efficient because it compresses the dataset into an FP-Tree, avoiding the need for maintaining a large list of itemsets as Apriori does.
- **Single Dataset Scan**: FP-Growth requires only a single pass over the dataset to build the FP-Tree and count item frequencies, making it ideal for scenarios where multiple scans of the database are costly or infeasible.
- **Reduced Computation**: Apriori generates and evaluates many candidate itemsets, leading to higher computational overhead. FP-Growth reduces the number of itemsets that need to be considered, leading to faster computations.
- **Easier Rule Generation**: FP-Growth simplifies the process of generating association rules, as it maintains direct information about the support of individual itemsets within the tree.

FP-Growth Algorithm Implementation

Dataset Description

The dataset provided includes 100,836 ratings and 3,683 tag applications across 9,742 movies. It captures user ratings using a 5-star rating system from the MovieLens movie recommendation service.

Data Files:

- **ratings.csv**: Contains user ratings. Each entry includes `userId`, `movieId`, `rating`, and `timestamp`.
- **movies.csv**: Contains movie information. Each entry includes `movieId`, `title`, and `genre`.

Assignment Tasks

Data Preprocessing

1. **Forming the Transactional Dataset**: Create a dataset consisting of entries in the form ⟨user id, {movies rated above 2}⟩. Only consider users with more than 10 movie ratings. Drop the `timestamp` field from the `ratings.csv`.

```
import pandas as pd

# Load datasets
ratings_df = pd.read_csv('ratings.csv')
movies_df = pd.read_csv('movies.csv')

# Drop 'timestamp' column and filter out ratings <= 2
ratings_df = ratings_df.drop(labels=['timestamp'], axis=1)
ratings_df = ratings_df[ratings_df['rating'] > 2]

# Keep users with more than 10 ratings
user_counts =
pd.DataFrame(ratings_df['userId'].value_counts().sort_values())
active_users = user_counts[user_counts['userId'] > 10].index
ratings_df = ratings_df[ratings_df['userId'].isin(active_users)]

# Group ratings by 'userId' and list movieIds for each user
transactions = ratings_df.groupby('userId')['movieId'].apply(lambda x:
(list(set(x)), len(x))).reset_index()

# Split 'movieId' into 'movies' and 'count' columns
transactions[['movies', 'count']] =
pd.DataFrame(transactions['movieId'].tolist(), index=transactions.index)
transactions.drop(columns=['movieId'], inplace=True)
```

Association Rule Mining:

1. **Splitting Movies into Training and Testing**: Split each user's movie set into training (80%) and testing (20%) sets.

```
import json

def split_movies(movies, test_size=0.2):
    movies = json.loads(movies)
    split_idx = int(len(movies) * (1 - test_size))
    train_movies = set(movies[:split_idx])
    test_movies = set(movies[split_idx:])
```

```
    return (train_movies, test_movies)

transactions[['trainMovies', 'testMovies']] =
transactions['movies'].apply(lambda movies:
pd.Series(split_movies(movies)))
```

	Unnamed: 0	userId	count	trainMovies	testMovies
0	0	1	226	{1024, 1, 1025, 3, 2048, 1029, 6, 1030, 1031, ...	{2459, 3489, 1954, 1445, 2470, 423, 4006, 2985...
1	1	2	28	{115713, 122882, 48516, 91529, 80906, 91658, 1...	{46970, 80489, 71535, 74458, 6874, 8798}
2	2	3	18	{70946, 2851, 5764, 4518, 3703, 26409, 2288, 8...	{1371, 5181, 7899, 5919}
3	3	4	167	{1025, 3079, 3083, 21, 1046, 2583, 4121, 538, ...	{904, 908, 910, 912, 914, 919, 920, 1947, 3996...
4	4	5	40	{1, 515, 261, 265, 527, 531, 21, 150, 534, 153...	{608, 232, 364, 110, 367, 247, 253, 349}
5	5	6	294	{2, 3, 4, 5, 6, 7, 8, 515, 10, 11, 516, 13, 52...	{509, 510, 405, 410, 412, 415, 416, 505, 419, ...
6	6	7	111	{1, 34319, 8207, 42002, 3114, 1584, 50, 58, 16...	{4995, 3977, 33162, 6539, 920, 3994, 924, 1954...
7	7	8	43	{2, 11, 141, 527, 21, 150, 282, 539, 32, 34, 2...	{235, 364, 236, 110, 367, 252, 377, 380, 253}
8	8	9	34	{3328, 4993, 5378, 5890, 5893, 1674, 5902, 373...	{5481, 5872, 6001, 371, 627, 1270, 2300}
9	9	10	119	{7169, 33794, 6155, 54286, 72720, 86548, 30749...	{103335, 103339, 78772, 81845, 104374, 81847, ...

2. **Mining Frequent Patterns using FP-Growth:**: Implement the FP-Growth algorithm to mine frequent itemsets from the training dataset.

```
class FPNode:
    def __init__(self, item, count, parent):
        self.item = item
        self.parent = parent
        self.children = {}
        self.link = None
        self.count = count

    def increment(self, count):
        self.count += count

    def pretty_print(self):
        print(f'({self.item}-{self.count})[', end='')
        for child in self.children.values():
            child.pretty_print()
        print('],', end='')


class FPTreeBase:
    def __init__(self, transactions, freq_1_itemsets, min_sup):
        self.transactions = transactions
        self.freq_1_itemsets = freq_1_itemsets
        self.min_sup = min_sup
        self.header_table = defaultdict(list)
        self.root = None

    def _build_tree(self):
        root = FPNode(None, 1, None)
        for transaction in self.transactions:
            self._insert_transaction(root, transaction)
```

```
        self.root = root
        return root

    def _insert_transaction(self, node, transaction):
        if len(transaction) == 0:
            return

        first_item, freq = transaction[0]
        if first_item in node.children:
            node.children[first_item].increment(freq)
        else:
            new_node = FPNode(first_item, freq, node)
            node.children[first_item] = new_node
            self.header_table[first_item].append(new_node)

        remaining_tr = transaction[1:]
        self._insert_transaction(node.children[first_item], remaining_tr)

    def mine_patterns(self, depth=0):
        patterns = {}
        for item in reversed(self.freq_1_itemsets):
            if item not in self.header_table:
                continue
            conditional_patterns = []
            for node in self.header_table[item]:
                path = self._prefix_path(node)
                if path:
                    conditional_patterns.append([[p, node.count] for p in
path])

            leaf_item_freq = defaultdict(int)
            for pattern in conditional_patterns:
                for node, freq in pattern:
                    leaf_item_freq[node] += freq

            leaf_item_freq = {k: v for k, v in leaf_item_freq.items() if v
>= self.min_sup}
            filtered_patterns = [list(filter(lambda p: p[0] in
leaf_item_freq, pattern)) for pattern in conditional_patterns]

            for pattern in filtered_patterns:
                temp = []
                for node, freq in pattern:
                    temp.append([node, freq])

            if len(conditional_patterns) == 1 and conditional_patterns[0]
[0][1] >= self.min_sup:
                patterns[tuple(sorted([node for node, freq in
conditional_patterns[0]]))] = conditional_patterns[0][0][1]
        return patterns
```

	itemset	Support
0	(318,)	307
1	(296,)	289
2	(593,)	265
3	(2571,)	264
4	(356,)	240
...	...	...
68332	(260, 296, 858, 1198, 2762)	50
68333	(260, 589, 858, 1198, 2762)	50
68334	(858, 1196, 1198, 2571, 2762)	50
68335	(260, 858, 1196, 1198, 2571, 2762)	50
68336	(50, 858, 1198, 2762)	50

68337 rows × 2 columns

- **Generating Association Rules**: After mining frequent patterns, generate association rules and sort them by support and confidence.

```
import ast

# Read patterns and transform 'movies' column
patterns = pd.read_csv('frequent-itemsets.csv')
patterns['movies'] = patterns['itemset'].apply(lambda x:
set(ast.literal_eval(x)))

association_df = []

for _, row in patterns.iterrows():
    support, movies = row
    if len(movies) > 1:
        for mv in movies:
            X = mv
            Y = movies.copy()
            Y.discard(X)
```

```
                association_df.append([X, Y, support])

    association_df = pd.DataFrame(association_df, columns=['X', 'Y',
    'Support'])

    # Merge with frequency of itemsets and compute confidence
    freq_1_df = pd.read_csv('1-itemset-frequency.csv')
    freq_1_df['X'] = freq_1_df['mId']
    association_df = pd.merge(association_df, freq_1_df, on='X', how='left')
    association_df['Confidence'] = association_df['Support'] /
    association_df['X Freq']

    # Filter based on minimum confidence
    association_df = association_df[association_df['Confidence'] >= 0.1]
    association_df['Rule'] = association_df['X'].astype(str) + ' -> ' +
    association_df['Y'].apply(lambda y: str(y))

    # Save top 100 rules based on support and confidence
    support_sorted_df = association_df.sort_values(by='Support',
    ascending=False).head(100)
    confidence_sorted_df = association_df.sort_values(by='Confidence',
    ascending=False).head(100)

    support_sorted_df.to_csv('support_sorted_top_100.csv', index=False,
    encoding='utf-8')
    confidence_sorted_df.to_csv('confidence_sorted_top_100.csv', index=False,
    encoding='utf-8')
```

	X	Y	Support	X Freq	Confidence	Rule
0	318	{296}	209	307	0.680782	318 -> {296}
1	296	{318}	209	289	0.723183	296 -> {318}
2	296	{593}	194	289	0.671280	296 -> {593}
3	593	{296}	194	265	0.732075	593 -> {296}
4	318	{593}	189	307	0.615635	318 -> {593}
...	...	...	...	...	...	...
296937	1197	{260, 1210, 2858, 2571, 1196}	50	139	0.359712	1197 -> {260, 1210, 2858, 2571, 1196}
296938	1210	{260, 2858, 2571, 1196, 1197}	50	186	0.268817	1210 -> {260, 2858, 2571, 1196, 1197}
296939	593	{2571, 1213, 1214}	50	265	0.188679	593 -> {2571, 1213, 1214}
296940	2571	{593, 1213, 1214}	50	264	0.189394	2571 -> {593, 1213, 1214}
296941	1197	{592, 1196, 589}	50	139	0.359712	1197 -> {592, 1196, 589}

296942 rows × 6 columns

The initial list includes the top 100 association rules, arranged in order of their support

	X	Y	Support	X Freq	Confidence	Rule
0	296	{318}	209	289	0.723183	296 -> {318}
1	318	{296}	209	307	0.680782	318 -> {296}
2	296	{593}	194	289	0.671280	296 -> {593}
3	593	{296}	194	265	0.732075	593 -> {296}
4	593	{318}	189	265	0.713208	593 -> {318}
...	...	...	...	...	...	...
95	1210	{2571, 260}	131	186	0.704301	1210 -> {2571, 260}
96	2571	{1210, 260}	131	264	0.496212	2571 -> {1210, 260}
97	296	{1196}	131	289	0.453287	296 -> {1196}
98	260	{1210, 2571}	131	239	0.548117	260 -> {1210, 2571}
99	1196	{296}	131	204	0.642157	1196 -> {296}

100 rows × 6 columns

The second list comprises the top 100 rules, prioritizing them according to confidence.

	X	Y	Support	X Freq	Confidence	Rule
0	33493	{2571}	61	63	0.968254	33493 -> {2571}
1	33493	{260}	61	63	0.968254	33493 -> {260}
2	33493	{1196}	61	63	0.968254	33493 -> {1196}
3	6934	{2571}	55	57	0.964912	6934 -> {2571}
4	33493	{2571, 1196}	60	63	0.952381	33493 -> {2571, 1196}
...	...	...	...	...	...	...
95	1258	{296}	79	94	0.840426	1258 -> {296}
96	4973	{356}	68	81	0.839506	4973 -> {356}
97	2683	{260}	68	81	0.839506	2683 -> {260}
98	1291	{1196}	114	136	0.838235	1291 -> {1196}
99	5378	{1210}	62	74	0.837838	5378 -> {1210}

100 rows × 6 columns

The rules that appear in both the lists arranged based on their confidence score

	X	Y	Support	X Freq	Confidence	Rule
0	1196	(260,)	182	204	0.892157	1196 -> {260}
1	1210	(260,)	165	186	0.887097	1210 -> {260}

Evaluation Metrics

1. **Precision**:
   Precision measures the accuracy of the recommendations. It answers the question:
   "Of all the items we recommended, how many are relevant?"

Precision is calculated as:

$$ \text{Precision} = \frac{\text{Number of relevant items recommended}}{\text{Total number of items recommended}} $$

Precision decreases when there are more false positives, meaning more items that are recommended but are not actually relevant. In other words, if the recommendations include more irrelevant items, precision decreases.

2. **Recall**:
   Recall measures the coverage of relevant items. It answers the question:
   "Of all the relevant items, how many were successfully recommended?"

Recall is calculated as:

$$ \text{Recall} = \frac{\text{Number of relevant items recommended}}{\text{Total number of relevant items}} $$

Recall increases as we receive more relevant recommendations. As the number of recommended items (K) increases, the system captures a larger portion of the relevant items.

3. **Recommendation and Evaluation**:

For each user in the test set, we selected association rules X→Y, where X corresponds to a movie from the training set. The set Y represents the recommended movies. To evaluate the recommendation system, we computed the average precision and recall metrics. We varied the number of rules considered from 1 to 10.

Inferences from Precision and Recall vs Num Rules

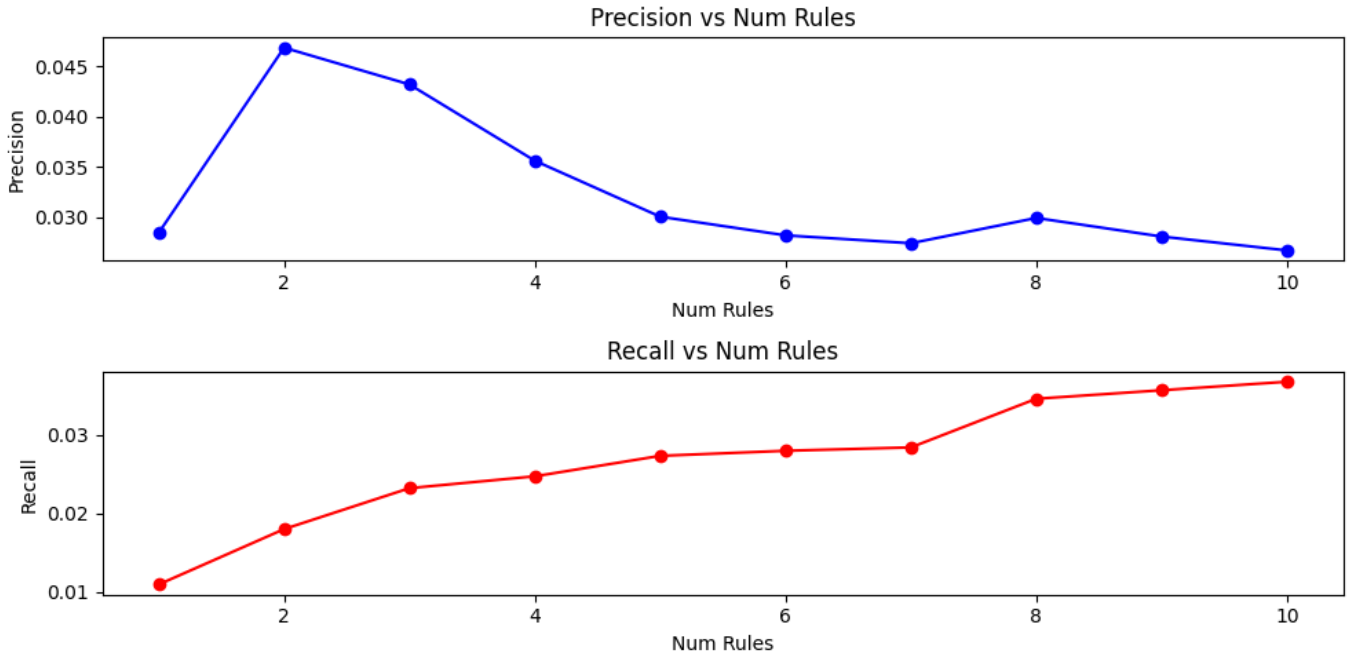
1. **Precision vs Num Rules**:

   - Initially, as the number of rules increases from 1 to 2, there is a sharp rise in precision, peaking at around 0.045.
   - After reaching the peak at 2 rules, precision steadily declines as the number of rules increases further, indicating diminishing accuracy in recommendations.
   - This trend suggests that a small number of rules is sufficient for maintaining high precision, but adding more rules leads to a higher number of false positives, reducing precision.

2. **Recall vs Num Rules**:
    - Recall shows a gradual and steady increase as the number of rules increases, moving from 0.01 at 1 rule to approximately 0.03 at 10 rules.
    - This indicates that as more rules are applied, the system is able to capture more relevant items, improving the recall metric.
    - However, the increase in recall tends to flatten after reaching 8 rules, suggesting that beyond a certain point, adding more rules does not significantly improve the coverage of relevant items.

Overall Observation:

- **Trade-off between Precision and Recall**: The graph highlights a typical trade-off between precision and recall. Increasing the number of rules improves recall but negatively impacts precision, implying that more recommendations capture a larger portion of relevant items but also include more irrelevant items.



4. **Sample User Analysis**: To gain a deeper understanding of the system's performance, we selected a sample of users from the test set and examined their precision and recall values. The results were plotted as precision and recall graphs for the sample users.

Inferences from Precision and Recall by User

Precision by User:

1. **Variation in Precision**: The precision scores vary significantly across users. Some users exhibit relatively high precision, such as:

- User 69 with 0.14
- User 239 with 0.10
- User 330 with 0.17
- User 588 with 0.22

On the other hand, a large number of users have a precision of 0, indicating that no relevant items were recommended for them.

1. **High Precision Users**: The highest precision observed is 0.22 (User 59), suggesting that for this user, a good proportion of recommended items were relevant. Other users like 69, 239, 330, and 588 also performed well in terms of precision.

2. **Low Precision Users**: Many users, particularly those between Users 8 to 45, 283 to 312, and 444 to 473, have precision scores of 0, implying that irrelevant recommendations were predominantly made for these users.

Recall by User:

1. **Variation in Recall**: Similar to precision, recall scores also show wide variability among users. Some users, particularly:

- User 48 with 0.14
- User 189 with 0.25
- User 399 with 0.29
- User 588 with 0.22

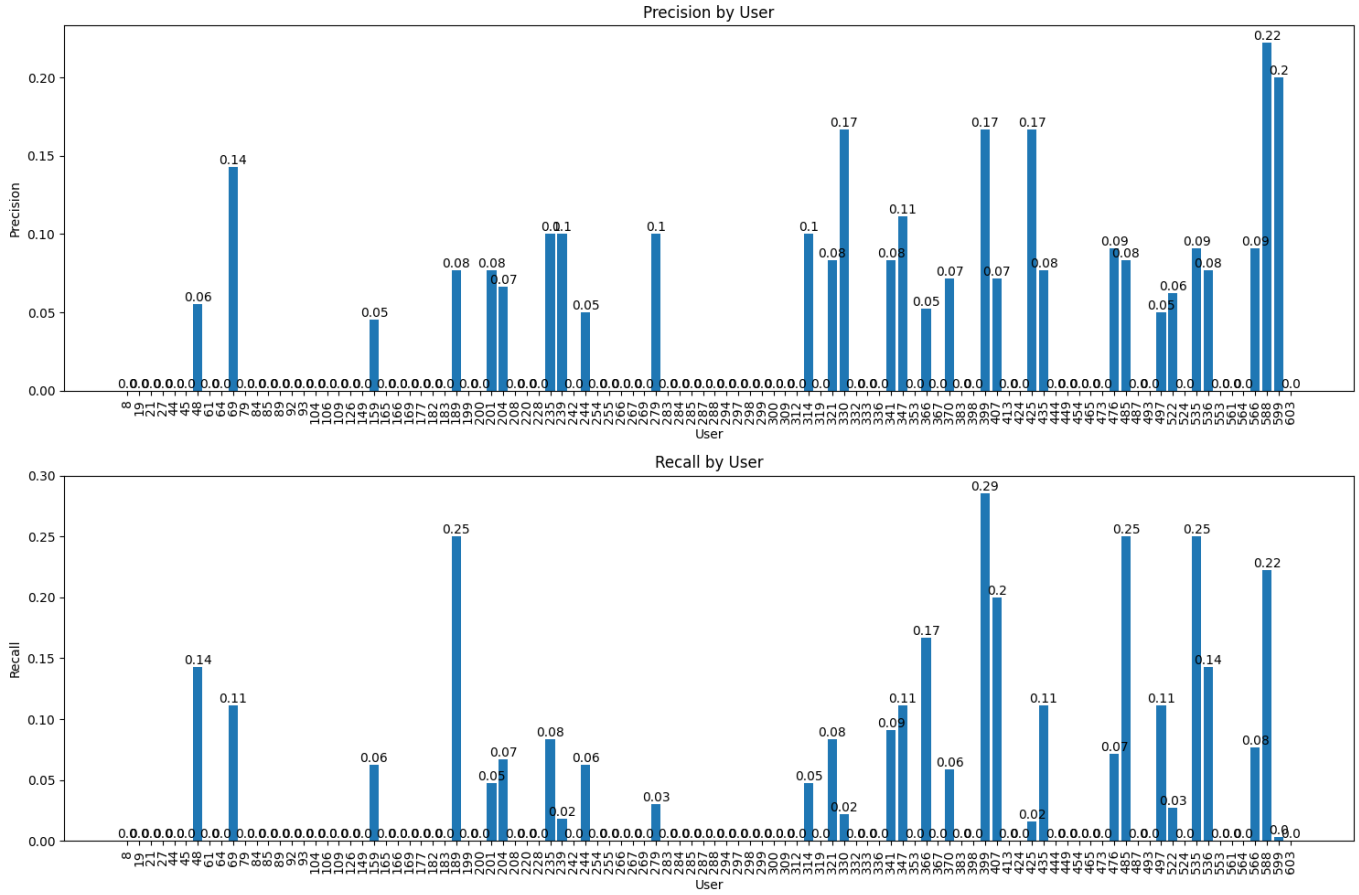
exhibit higher recall, indicating that a larger portion of the relevant items was successfully recommended to them.

1. **High Recall Users**: The highest recall value observed is 0.29 (User 399), meaning that nearly 30% of the relevant items for this user were recommended. Users 189 and 588 also exhibit relatively high recall values.

2. **Low Recall Users**: Similar to precision, many users have recall values of 0, implying that no relevant items were recommended for these users. This is especially prominent in Users 79 to 149, 283 to 312, and 444 to 473.

Overall Observations:

- **Disparity in Recommendation Performance**: There is a significant disparity in how well recommendations perform across users. Some users see a high level of precision and recall, while many others do not receive relevant recommendations.
- **Correlation Between Precision and Recall**: Users who have higher precision generally also exhibit higher recall, suggesting that when the system recommends relevant items for a user, it tends to recommend more of them.
- **Improvement Areas**: A significant number of users have both precision and recall values of 0, highlighting the need to improve the recommendation system for these users to make the predictions more relevant and effective.


Precision by User
Precision
User
Recall by User
Recall
User