DSC202 Final Report: Hybrid Amazon Product Recommendation System using PostgreSQL, Neo4j, and MongoDB

Hao Wang haw126@ucsd.edu
Tianxiang Wang tiw014@ucsd.edu
Tinahao Zhou tiz032@ucsd.edu

Abstract

Recommendation systems are an essential part of modern e-commerce. In this project, we integrate PostgreSQL, Neo4j, and MongoDB to create a hybrid recommendation system. We demonstrate the advantages of using relational databases for structured queries, graph databases for community-based recommendations, and key-value stores for rapid access. Using Neo4j, we applied the Louvain community algorithm to co-purchase and category data, revealing hidden community structures that reflect user behavior. Our final recommendations are largely based on these communities, leveraging relational insights for relevance. We rank these suggestions by sales rank to ensure quality. Through iterative refinement, our system improves accuracy and efficiency.

1 Introduction

Recommender systems have become a critical component in e-commerce, influencing customer decisions and driving sales. Traditional recommendation models primarily fall into two categories: collaborative filtering (which suggests items based on user interactions)[4] and content-based filtering (which recommends items based on similar product attributes)[2]. However, both approaches have limitations. Collaborative filtering suffers from the cold-start problem, where new items without historical interactions are difficult to recommend[5], while content-based filtering is limited in capturing complex purchase behaviors and relationships between products[1].

To overcome these challenges, we adopt a hybrid approach, leveraging the strengths of multiple database systems: Our system integrates **PostgreSQL** for structured product data retrieval, **Neo4j** for capturing co-purchase relationships through graph-based analysis, and **MongoDB with Redis** for efficiently managing promoted products. This hybrid architecture ensures accurate, scalable, and personalized recommendations by combining metadata-driven filtering, graph-based community detection, and business-driven promotions.

2 DataSet Preprocessing and Exploration With Relational Database

Our dataset comes from the **Amazon Meta Dataset**, containing detailed product information, categories, sales ranks, and co-purchase relationships.

2.1 Relational Database(PostgreSQL)

Two key tables are created using the metadata in our PostgreSQL database:

- **amazon_products**: Stores structured metadata about each product.
- **amazon_reviews**: Stores user reviews and ratings.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

2.2 Key Columns in amazon_products Table

Column Name	Description
asin	Unique Amazon Standard Identification Number for the product.
title	Name of the product.
product_group	The category the product belongs to (Books, DVDs, Electronics, etc.).
sales_rank	Amazon's ranking system indicating product popularity.
similar_products	A list of ASINs frequently co-purchased with the product.
categories	A hierarchical list of categories for the product.

Table 1: Key columns in the amazon_products table

Some columns, such as sales_rank, require further preprocessing and cleaning to ensure accurate retrieval of top-ranked products.

2.3 Key Columns in amazon_reviews Table

Column Name	Description
id	Unique identifier for each review.
asin	The product being reviewed (foreign key to amazon_products).
review_date	Date the review was posted.
customer_id	Unique identifier for the customer.
rating	The star rating given by the customer (1-5).
votes	Number of votes indicating the usefulness of the review.
helpfulness	Number of users who found the review helpful.

Table 2: Key columns in the amazon_reviews table

2.4 Data Preprocessing

To improve query performance and ensure high-quality recommendations, we performed several preprocessing steps:

- **Removing Duplicate Records:** We identified and removed duplicate products based on ASIN, keeping only the entry with the lowest sales_rank.

```
WITH Deduplicated AS (
    SELECT *,
        ROW_NUMBER() OVER (PARTITION BY asin ORDER BY sales_rank ASC) AS row_num
    FROM amazon_products
)
DELETE FROM amazon_products
WHERE asin IN (
    SELECT asin FROM Deduplicated WHERE row_num > 1
);
```

- **Handling Missing Values:** We deleted records with NULL values in critical columns such as ASIN, title, and sales_rank.
- **Creating Indexes:** To improve query efficiency, we created indexes on frequently used fields such as ASIN, sales_rank, customer_id, and categories.

  (Full code available in Github *import_preprocesing_relational.sql*)

2.5 Data Analysis

To better understand the dataset and prepare for recommendation, we performed exploratory data analysis:

- **Finding Best-Selling Products:** We initially retrieved the top-selling products but found that many of the first results had `sales_rank = -1`, indicating discontinued items. We refined the query to exclude these invalid rankings.
- **Identifying High Co-Purchase Items:** We analyzed which products had the highest number of co-purchased items to better model user purchasing behavior.

The SQL queries used for these analyses are:

- Find best-selling products (excluding discontinued items – invalid rankings -1)

```
SELECT title, sales_rank
FROM amazon_products
WHERE sales_rank > 0
ORDER BY sales_rank ASC
LIMIT 10;
```

- Find products with high co-purchase relationships

```
SELECT asin, title,
   LENGTH(similar_products) - LENGTH(REPLACE(similar_products, '|', '')) + 1
   AS similar_count
FROM amazon_products
WHERE similar_products IS NOT NULL AND similar_products <> ''
ORDER BY similar_count DESC
LIMIT 10;
```

3 DataSet Creation and Exploration With Graph Database

3.1 Graph Database(Neo4j)

After constructing the relational database, we created our own graph dataset by leveraging two key relationships identified in the dataset:

- **Co-purchasing**: Represented by the CO_PURCHASED_WITH relationship, which connects products frequently bought together.
- **Category**: Represented by the BELONGS_TO relationship, which links products to their respective categories.

The data in Neo4j consists of two main node types:

- **Product Nodes**
- **Category Nodes**

3.2 Preprocessing and Data Cleaning

Before creating and importing datasets into Neo4j, we performed extensive preprocessing to ensure data quality and reduce redundancy. This step was crucial because the initial dataset contained excessive duplicate categories and overly general category labels.

Key steps including:

1. **Modeling Co-Purchase Relationships**: To capture relationships between products frequently bought together, we extracted co-purchase data and exported them to "*co_purchased.csv*". We used the *similar products* column, which contained a pipe-separated list of related product ASINs, and transformed it into pairs of products.

```
SELECT asin AS product_asin,
       UNNEST(string_to_array(similar_products, '|')) AS similar_asin
```

```
FROM amazon_products
WHERE similar_products IS NOT NULL
  AND similar_products <> '';
```

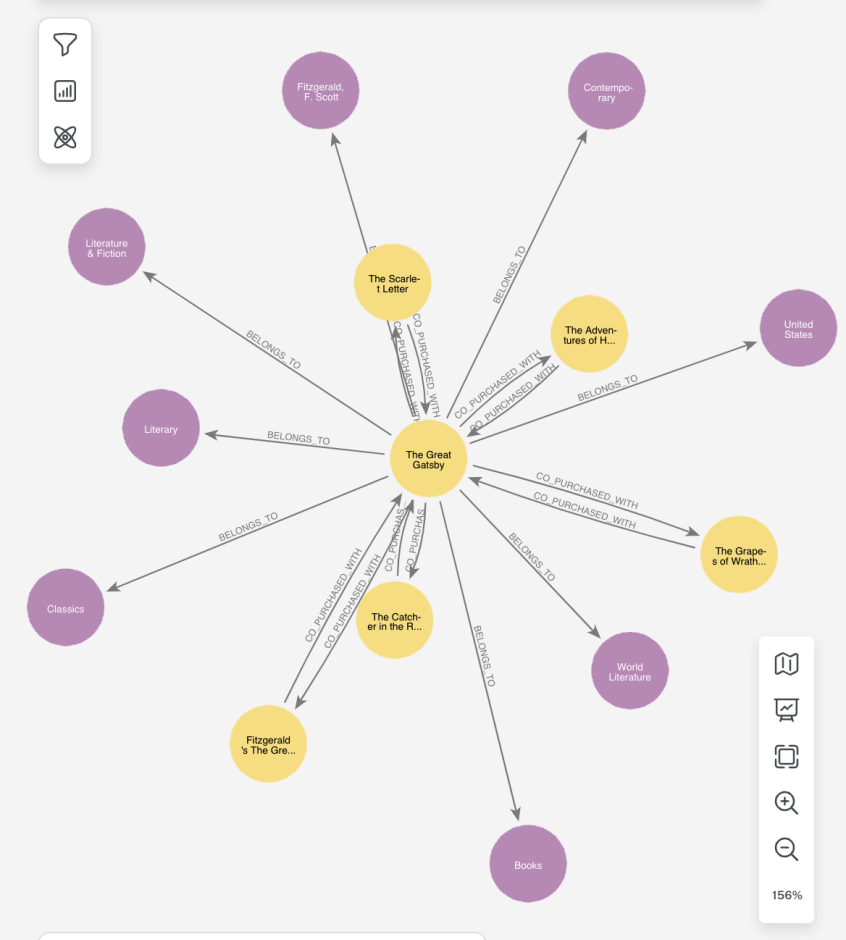

Figure 1: Graph Database Visualization with Nodes and Relationships

2. **Category Trimming and Export**: Categories in the original dataset were often overly general (e.g., 'General', 'Genres') or redundant. We cleaned and trimmed them to focus on meaningful labels, exporting the result to "*trim_category.csv*".
3. **Exporting Product Data for Neo4j**: We prepared a subset of the product data for import into Neo4j by exporting it to a CSV file named "*amazon_products_for_neo.csv*". This file included the following columns:

   asin title product_group sales_rank

   (Full code available in Github *trimming_data_for_neo.sql*)

3.3 Importing Data into Neo4j

After creating and preparing the CSV files, we imported them into Neo4j to construct the graph database. The full code for this process is available in our GitHub repository under "*construct_graph_database_in_neo*"

Since "*trim_category.csv*" is a large file, we used a batch processing technique, importing the data in chunks of 1000 rows. This slower but more stable approach prevents memory overload and ensures the process runs smoothly:

```
LOAD CSV WITH HEADERS FROM 'file:///trim_category.csv' AS row
CALL {
  WITH row
  MATCH (p:Product {asin: row.product_asin})
  MERGE (c:Category {name: row.category_name})
  MERGE (p)-[:BELONGS_TO]->(c)
} IN TRANSACTIONS OF 1000 ROWS;
```

3.4 Graph Analysis

To verify data integrity and explore the graph database, we performed several queries using Neo4j's Graph Data Science (GDS) library. These analyses helped us understand the structure of the product co-purchase network and identify key patterns and influential products.

- **Graph Projection**: We started by creating a graph projection named **productCoPurchaseGraph** to prepare the data for subsequent analyses. This projection includes nodes labeled Product connected by relationships of type *CO_PURCHASED_WITH*, configured as undirected to emphasize mutual co-purchase relationships.
- **Community Detection with Louvain Algorithm**: To uncover groups of products frequently purchased together, we applied the Louvain community detection algorithm to the **productCoPurchaseGraph**. This algorithm identifies communities by optimizing modularity, where products within the same community have denser connections than with those outside it:

  ```
  CALL gds.louvain.write('productCoPurchaseGraph', {
    writeProperty: 'louvainCommunity'
  })
  YIELD communityCount, modularity, modularities;
  ```

- **PageRank Analysis**: To evaluate the importance of individual products within the co-purchase network, we employed the PageRank algorithm. PageRank assigns scores based on the quantity and quality of connections, identifying products that are central or influential in the graph.

4 Key-value Database and In-Memory System

Our recommendation system utilizes MongoDB and Redis alongside other databases to manage promoted products and enhance performance.

4.1 Mongo DB

We use MongoDB to store promoted products in the promoted_collection. These are manually curated items, such as products on sale or new releases, that we want to feature in recommendations.

```
promoted\_data = {
    "asin": product[0],
    "title": product[1],
    "reason": promotion_reason,
    "added_date": "2025-03-18"
}

# Insert into MongoDB
promoted\_collection.insert\_one(promoted\_data)
```

4.2 Redis

To optimize response times, we employ Redis as an in-memory caching system. Redis temporarily stores promoted products, allowing the system to retrieve them quickly without querying MongoDB

repeatedly. This caching layer is essential for maintaining performance, particularly when promoted products are accessed frequently. products are cached it in Redis with a specific key and an expiration time of 7 days.

```
cache\_data = {
    "asin": product[0],
    "title": product[1],
    "reason": promotion_reason,
    "added_date": "2025-03-18"
}

r.set(f"promoted:{product[0]}", json.dumps(cache\_data), ex=604800)  # 604800 seconds = 7 days
```

5 Methodology: Database Connections

To integrate multiple data sources efficiently, we established connections to three different databases and an in-memory caching system: PostgreSQL, Neo4j, MongoDB, and Redis. Each database serves a distinct role in our recommendation system. Figure 1 shows the code for connection.

```
# PostgreSQL (change to your own database/user/password/host/port)
pg_conn = psycopg2.connect(
    database="amazon_db",
    user="postgres",
    password="******",
    host="localhost",
    port="5432"
)
pg_cursor = pg_conn.cursor()

# Test PostgreSQL
pg_cursor.execute("SELECT COUNT(*) FROM amazon_products;")
result = pg_cursor.fetchone()
print("PostgreSQL is connected, total products:", result[0])

# Neo4j
neo4j_driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "********")) # Change to your own password

# Test Neo4j
def test_neo4j_connection():
    query = "MATCH (p:Product) RETURN count(p) AS total_products"
    with neo4j_driver.session() as session:
        result = session.run(query)
        total_products = result.single()["total_products"]
        print("Neo4j is connected, total products:", total_products)

test_neo4j_connection()

# MongoDB
mongo_client = pymongo.MongoClient("mongodb://localhost:27017/")
db = mongo_client["amazon"]
promoted_collection = db["promoted_products"]

# Redis
r = redis.Redis(host='localhost', port=6379, db=0)
```

```
PostgreSQL is connected, total products: 548552
Neo4j is connected, total products: 548552
```

Figure 2: Query integration across PostgreSQL, Neo4j, and MongoDB.

- **PostgreSQL Connection**
  We establish a connection to a **relational database** storing structured product and review data. PostgreSQL allows for efficient filtering and querying of metadata. To verify the connection, we execute:

  ```
  SELECT COUNT(*) FROM amazon_products;
  ```

  This confirms successful access by returning the total number of products.

- **Neo4j Connection**
  We connect to **Neo4j**, a graph database used to model co-purchase relationships between products. To ensure accessibility, we run the following Cypher query:

```
MATCH (p:Product) RETURN count(p) AS total_products;
```

This retrieves the total number of products stored as nodes in the graph.

- **MongoDB Connection**
  We use **MongoDB**, a NoSQL database, to store promoted products. The `promoted_collection` table contains manually selected promotional items that can be retrieved efficiently. This ensures that recommended products include featured items.
- **Redis Connection**
  To improve response times, we utilize **Redis** as an in-memory caching system. Redis stores promoted products temporarily, reducing database load by retrieving frequently requested items instantly.

6 Methodology: How the Recommendation System Works

To generate recommendations, the system follows these steps:

- **Handling User Input:**
  If the input is an **ASIN**, the system directly fetches that product details using the following query first:

```
SELECT asin, title, product_group, sales_rank
FROM amazon_products
WHERE asin = %s;
```

  If the input is a **title or keyword**, PostgreSQL is queried to find 3 products has the keyword in their title first:

```
SELECT asin FROM amazon_products
WHERE title ILIKE %s
ORDER BY sales_rank ASC
LIMIT 3;
```

- **Fetching Product Details from PostgreSQL**: After identifying relevant ASINs from user input, the system uses the fetch_product_details function to retrieve detailed product information, including average ratings, from PostgreSQL:

```
def fetch_product_details(pg_cursor, asins):
    if not asins:
        return []
    query = """
    SELECT
        p.asin,
        p.title,
        p.product_group,
        p.sales_rank,
        COALESCE(AVG(r.rating), 0) AS avg_rating
    FROM
        amazon_products p
    LEFT JOIN
        amazon_reviews r ON p.asin = r.asin
    WHERE
        p.asin IN %s
    GROUP BY
        p.asin, p.title, p.product_group, p.sales_rank
    ORDER BY
        p.sales_rank ASC
    """
    pg_cursor.execute(query, (tuple(asins),))
```

Table 3: Example Recommendation for "The Great Gatsby"

ASIN	Title	Product Group	Sales Rank	Avg Rating	Promoted
0684801523	The Great Gatsby	Book	956	4.18	No
6301247485	The Great Gatsby (Video)	Video	2851	3.80	No
0764586017	Fitzgerald's The Great Gatsby (Cliffs Notes)	Book	4163	3.00	No
0738700797	Candlemas: Feast of Flames	Book	168596	4.33	Yes
1559362022	Wake Up and Smell the Coffee	Book	518927	3.75	Yes

```
return pg_cursor.fetchall()
```

- **Fetching Related Products from Neo4j:** Once a product is identified, its co-purchase network is explored using Louvain community detection to retrieve similar items:

```
MATCH (p:Product)-[:CO_PURCHASED_WITH]->(p2)
WHERE p.asin IN $matching_asins
AND p.louvainCommunity = p2.louvainCommunity
RETURN DISTINCT p2.asin AS recommended_asin;
```

  These products are ranked based on popularity and relevance.

- **Fetching Promoted Products from MongoDB:** Additional promoted products are retrieved and stored in Redis for fast access:

```
cached_data = r.keys("promoted:*")
if cached_data:
    promoted = [json.loads(r.get(k).decode()) for k in cached_data]
else:
    promoted = list(promoted_collection.find().limit(3))
```

- **Final Ranking:** The recommendations from PostgreSQL, Neo4j, and MongoDB are merged. Products are ranked first by **average rating** and then by **sales rank**:

```
SELECT p.asin, p.title, p.product_group, p.sales_rank,
    COALESCE(AVG(r.rating), 0) AS avg_rating
FROM amazon_products p
LEFT JOIN amazon_reviews r ON p.asin = r.asin
WHERE p.asin IN %s
GROUP BY p.asin, p.title, p.product_group, p.sales_rank
ORDER BY avg_rating DESC, p.sales_rank ASC;
```

7 Methodology: Examples Recommendation Queries and DEMO

Example 1: Searching for "The Great Gatsby" - Sample Result in Table 3

```
get_recommendations("the great gatsby", False, pg_cursor, neo4j_driver)
```

Example 2: Searching by ASIN for "0684801523" (The Great Gatsby) - Sample Result in Table 4

```
get_recommendations("0684801523", True, pg_cursor, neo4j_driver)
```

The complete recommendation system and demo are available in our GitHub repository under *Recommendation System with Demo.ipynb*.

Table 4: Example Recommendation for ASIN "0684801523"

ASIN	Title	Product Group	Sales Rank	Avg Rating	Promoted
0684801523	The Great Gatsby	Book	956	4.18	No
0316769487	The Catcher in the Rye	Book	60	4.16	No
0142000663	The Grapes of Wrath	Book	201	4.29	No
0738700797	Candlemas: Feast of Flames	Book	168596	4.33	Yes

Simple UI Demostration of the Recommendation System

To further illustrate our recommendation system in action, we present a graphical demonstration of its functionality. This simple UI is available in our Github repository under *Recommendation System with Demo.ipynba*

Product Recommendation System

Enter a search term (title or ASIN) and select the search type, then click 'Get Recommendations'.

Search Term: gatsby

Search by ASIN

Get Recommendati...

Recommendations generated with promoted products!

	ASIN	Title	Product Group	Sales Rank	Avg Rating	is_promoted
0	0684801523	The Great Gatsby	Book	956	4.1809421841541756	False
1	6301247485	The Great Gatsby	Video	2851	3.8000000000000000	False
2	0764586017	Fitzgerald's The Great Gatsby (Cliffs Notes)	Book	4163	3.0000000000000000	False
3	0316769487	The Catcher in the Rye	Book	60	4.1683593750000000	False
4	0142000663	The Grapes of Wrath : (Centennial Edition)	Book	201	4.2920696324951644	False
5	0553210092	The Scarlet Letter	Book	368	3.5143678160919540	False
6	0553210793	The Adventures of Huckleberry Finn (Bantam Classics)	Book	1138	3.9966555183946488	False
7	B00007KQA4	Of Mice And Men (Special Edition)	DVD	4915	4.5760869565217391	False
8	0764586041	The Adventures of Huckleberry Finn (Cliffs Notes)	Book	9206	4.3333333333333333	False
9	0764585886	The Crucible (Cliffs Notes)	Book	18300	3.5454545454545455	False
10	0738700797	Candlemas: Feast of Flames	Book	168596	4.3333333333333333	True
11	1559362022	Wake Up and Smell the Coffee	Book	518927	3.7500000000000000	True
12	078510870X	Ultimate Marvel Team-Up	Book	612475	3.6250000000000000	True

Figure 3: Example 1: Recommendation Results for the Search Term "Gatsby"

Product Recommendation System

Enter a search term (title or ASIN) and select the search type, then click 'Get Recommendations'.

Search Term: B00007KQA4

Search by ASIN

Get Recommendati...

Recommendations generated with promoted products!

	ASIN	Title	Product Group	Sales Rank	Avg Rating	is_promoted
0	B00007KQA4	Of Mice And Men (Special Edition)	DVD	4915	4.5760869565217391	False
1	0140177396	Of Mice and Men (Penguin Great Books of the 20th Century)	Book	126	4.8000000000000000	False
2	0738700797	Candlemas: Feast of Flames	Book	168596	4.3333333333333333	True
3	1559362022	Wake Up and Smell the Coffee	Book	518927	3.7500000000000000	True
4	078510870X	Ultimate Marvel Team-Up	Book	612475	3.6250000000000000	True

Figure 4: Example 2: Recommendation Results for ASIN Search "B00007KQA4"

Product Recommendation System

Enter a search term (title or ASIN) and select the search type, then click 'Get Recommendations'.

Search Term: Wake

Search by ASIN

Get Recommendati...

Recommendations generated with promoted products!

	ASIN	Title	Product Group	Sales Rank	Avg Rating	is_promoted
0	0380002450	Awakening	Book	499	3.9033333333333333	False
1	B000065U1M	Elmo's World - Wake up with Elmo!	DVD	1534	4.2500000000000000	False
2	B000068MFC	Power Rangers Wild Force - Ancient Awakening	Video	1616	4.6666666666666667	False
3	0060931418	Their Eyes Were Watching God	Book	231	4.2550724637681159	False
4	B00005QFDX	Sesame Street - The Best of Elmo	DVD	330	4.4521739130434783	False
5	0553210092	The Scarlet Letter	Book	368	3.5143678160919540	False
6	B0000648WV	Elmo's World - Babies, Dogs & More	DVD	562	4.3846153846153846	False
7	067973225X	As I Lay Dying (Vintage International)	Book	742	3.9324324324324324	False
8	0684801523	The Great Gatsby	Book	956	4.1809421841541756	False
9	0553210793	The Adventures of Huckleberry Finn (Bantam Classics)	Book	1138	3.9966555183946488	False
10	0738700797	Candlemas: Feast of Flames	Book	168596	4.3333333333333333	True
11	1559362022	Wake Up and Smell the Coffee	Book	518927	3.7500000000000000	True
12	078510870X	Ultimate Marvel Team-Up	Book	612475	3.6250000000000000	True

Figure 5: Example 3: Recommendation Results for the Search Term "Wake"

These figures illustrate how the recommendation system integrates multiple data sources to generate relevant product recommendations. The results are ranked by average rating and sales rank, ensuring high-quality suggestions. Promoted products, fetched from MongoDB and cached in Redis, are highlighted to demonstrate the system's ability to incorporate business-driven promotions.

8 Lessons Learned from Iterative Improvements

Over multiple iterations, our recommendation system evolved significantly. We experimented with different versions to optimize relevance and performance. The complete iteration history, including early experiments and architecture decisions, is available in our GitHub repository under *Early Ideation and Architecting.ipynb*.

- **Version 1 (Basic SQL Query Matching):** The initial implementation only retrieved exact matches from PostgreSQL. This was fast but limited, as it failed to capture similar products or user behavior.
- **Version 2 (Simple "Customers Also Bought" Model Using Neo4j):** We introduced Neo4j to recommend products based on direct co-purchase links. This improved recommendations but still lacked deeper network-based insights.
- **Version 3 (Graph-Based Community Detection for Better Recommendations):** Instead of only looking at direct co-purchases, we applied Louvain community detection to recommend products within tightly connected clusters. This significantly improved personalization.
- **Version 4 (Hybrid Approach with PostgreSQL, Neo4j, and MongoDB):** The final version combines PostgreSQL metadata, Neo4j graph relationships, and MongoDB for promotions, resulting in a scalable, intelligent recommendation engine that balances efficiency and accuracy.

9 Conclusion

In this project, we developed a robust hybrid Amazon product recommendation system by integrating three distinct database technologies and one in-memory caching system, namely PostgreSQL, Neo4j, MongoDB, and Redis. The integrated approach effectively exploits structured data management, relationship discovery based on graphs, and rapid access to marketing content. Our system demonstrated significant improvements in recommendation accuracy by overcoming the limitations inherent in traditional collaborative and content-based filtering approaches. By extensive data preprocessing that includes deduplication, missing value handling, and indexing, we ensured that we had clean product metadata and user opinions at our command to be efficiently retrieved and mined. Graph walk in Neo4j permitted identification of co-purchase behaviors and product groups via communities, while MongoDB and Redis facilitated efficient returns of carefully curated promotion materials. All these elements combined into a scalable, effective, and adaptive recommendation system.

10 Future Direction

We brainstormed two possible future directions that would require time but would be rewarding.

- **Real-Time Personalization with Reinforcement Learning[6]**: Instead of relying on static data like co-purchases, we could use real-time user actions (clicks, searches, purchases) to make recommendations that adapt instantly. Reinforcement learning (RL) will help us tweak suggestions based on what users actually do, making them more personal and relevant.

  Possible Approaches: Collect live user actions like clicks and purchases from our recommendation system platform. Following a simple reinforcement learning approach to tweak recommendations on the spot. We can assign rewards to actions (+1 for a buy, +0.5 for a click), letting the model learn what users like as they interact. For new users or items, fall back to popular picks until there's enough data.
- **Sentiment Analysis for Review-Based Recommendations[3]**: Instead of relying solely on ratings or purchase history, we could also analyze the sentiment in user reviews to capture qualitative feedback.

  Possible Approaches: We could use the Amazon product review data from snap. Use natural language processing (NLP) tools such as BERT to extract sentiment scores from review texts. Integrate these scores into the recommendation algorithm by adjusting product ratings or weighting them as features in a model. For items with few reviews, default to average sentiment trends.

References

[1] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on Knowledge and Data Engineering*, 17(6), 2005.

[2] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. Content-based recommender systems: State of the art and trends. In *Recommender Systems Handbook*. Springer, 2011.

[3] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM Conference on Recommender Systems*. ACM, 2013.

[4] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001.

[5] Andrew I Schein, Alexandrin Popescul, Lyle H Ungar, and David M Pennock. Methods and metrics for cold-start recommendations. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, 2002.

[6] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. Drn: A deep reinforcement learning framework for news recommendation. In *Proceedings of the 2018 World Wide Web Conference*. ACM, 2018.