Movie Recommender System

Tushar Reddy and Shivam Ojha

May 7, 2024

Contents

1 Introduction

This report presents a comprehensive overview of a movie recommender system designed by us, over the past two months. The system leverages IMDb non-commercial datasets and various algorithms to provide accurate movie recommendations. The objective of this system is to assist users in finding movies similar to those they have enjoyed in the past, based on different attributes such as genre and release year.

2 Datasets

The system uses the following IMDb datasets:

- **title.basics**: Contains approximately 10 million movie records.
- **name.basics**: Contains approximately 13 million records of top crew members.

The datasets are available at `https://datasets.imdbws.com/`.

The **title.basics** dataset includes information such as the primary title of the movie, its start year, and genres. The **name.basics** dataset provides details about top crew members, including their primary name and the titles they are known for.

3 Technology Stack

The project employs the following technologies:

- **Python**: Used for Exploratory Data Analysis (EDA) and parsing.
- **Neo4j**: A graph database used to store and query movie data.
- **Cypher**: The query language for Neo4j.
- **Neo4j Bloom**: A tool for graph visualizations.

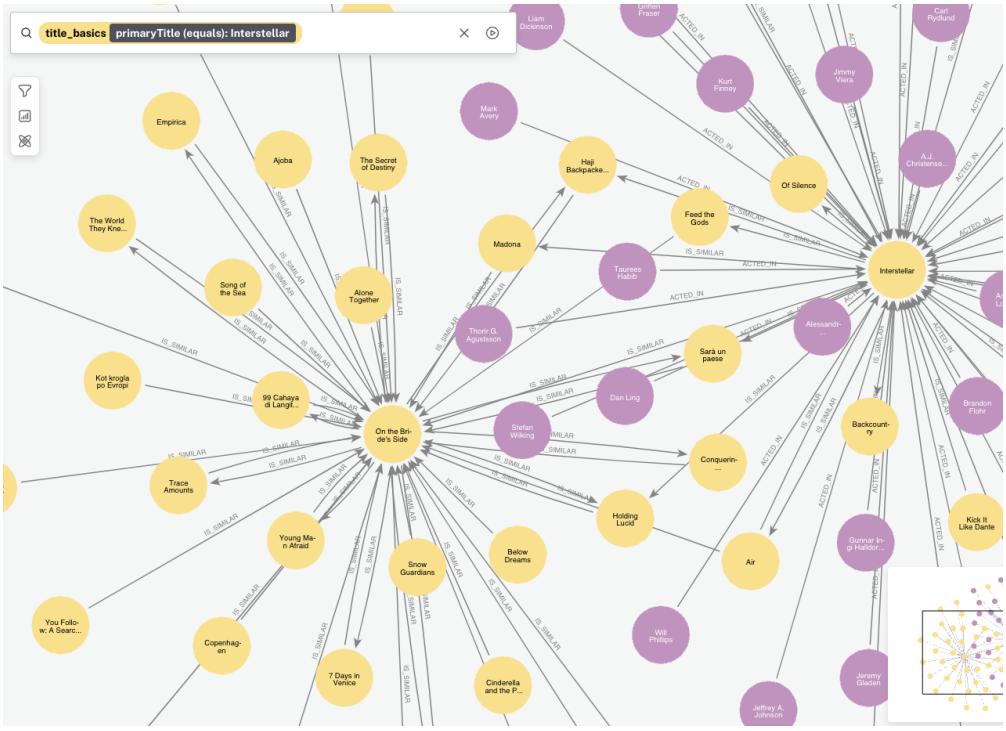


Figure 1: Bloom graph visualizer

3.1 Libraries

- **Pandas**: Python library for data manipulation and analysis.
- **Neo4j APOC**: For batch processing.
- **Neo4j GDS**: For in-memory operations, including similarity algorithms and community detection.

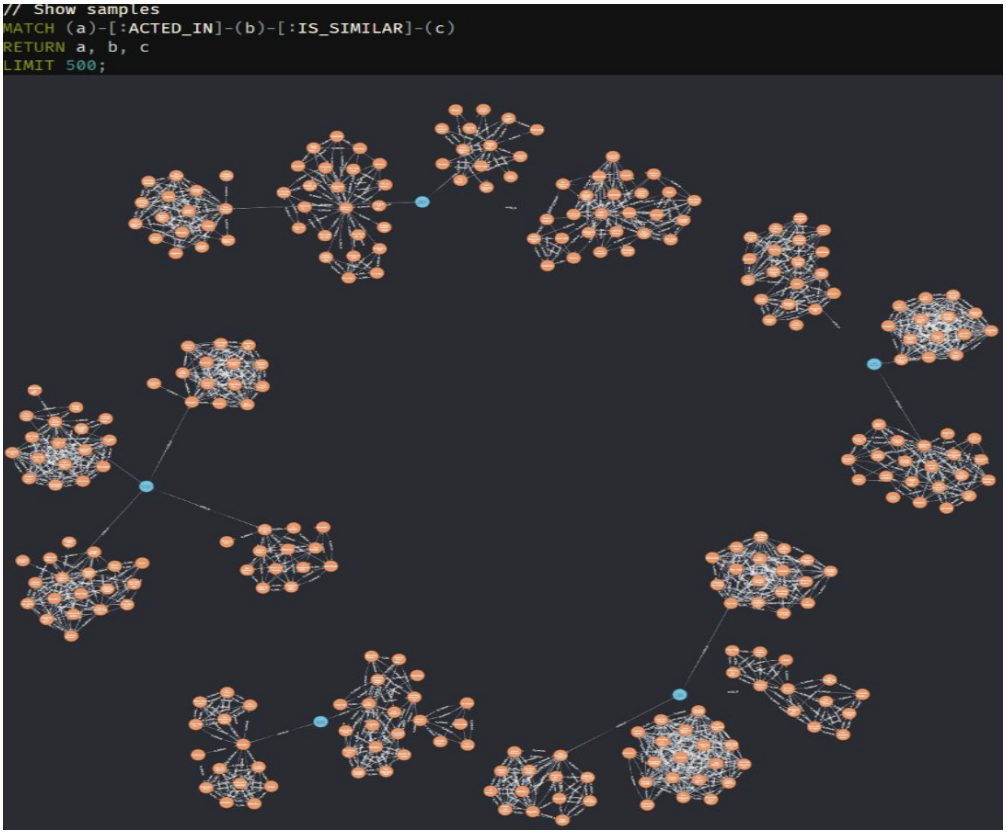
4 Exploratory Data Analysis (EDA)

EDA involves inspecting the datasets to understand their structure and contents. This process helps in identifying missing values, understanding distributions, and finding patterns. Python's Pandas library was primarily used for this purpose.

During the EDA phase, we discovered various interesting patterns in the data. For instance, we identified the distribution of movies across different genres and observed how the number of movies has changed over the years.

5 Graph Database - Neo4j

Neo4j is a graph database management system, which is highly suitable for applications involving complex relationships and networks. In this project, Neo4j is used to model movies and their attributes, such as



genres and release years, as nodes and edges.

5.1 Cypher Query Language

Cypher is the query language for Neo4j. It allows for the retrieval of nodes and relationships, as well as the execution of complex graph algorithms.

5.1.1 Data Modeling and Loading

The first step in setting up our database was to model the data in a way that reflects the relationships between movies, genres, and actors. Movies are represented as nodes, and their attributes (such as genres and release years) are properties of these nodes. Additionally, relationships between actors and movies are established to understand who acted in which movie.

To efficiently load large datasets, we used batch processing. This helps in handling large volumes of data by dividing it into smaller chunks and processing them sequentially. For instance, movies are loaded in batches, with each batch creating nodes for the movies and setting their properties.

5.1.2 Batch Processing with APOC

To load titles, we employed the APOC library for efficient batch processing. This allowed us to process large volumes of data in manageable chunks. By doing so, we ensured that the database could handle the insertion of millions of records without performance degradation.

```
CALL apoc.periodic.iterate("
    MATCH (t:title_basics_7thMay)
    MATCH (n:name_basics_7thMay)
    WHERE t.tconst in n.knownForTitles
    RETURN t, n
    ", "
    MERGE (n)-[:ACTED_IN]->(t)
    ", {batchSize: 10000, parallel: true}
)
```

Figure 2: A section where we used APOC

To load the movie titles, the data is read from a CSV file and nodes are created for each movie. The genres are split into individual elements, allowing for the creation of genre-specific nodes and relationships. This approach helps in efficiently managing the data and ensuring that each movie is correctly categorized.

5.1.3 Optimizations with Constraints and Indexes

To optimize the performance of our database, we created constraints to ensure unique identifiers for each movie and actor. Additionally, we indexed the primary titles of movies to speed up search queries. These optimizations are crucial for improving query response times and ensuring data integrity.

Creating constraints ensures that each movie node has a unique identifier, preventing duplicate entries. Similarly, indexing the primary titles allows for faster retrieval of movie records when queried by their titles. These optimizations significantly enhance the performance and scalability of the recommender system.

6 Similarity Algorithms

To recommend movies, it is crucial to measure the similarity between movies. We have implemented several similarity algorithms to achieve this.

6.1 Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors projected in a multi-dimensional space. For our purposes, these vectors represent different movies. This measure is particularly useful in

determining how similar two movies are based on their genre attributes.

$$\text{Cosine Similarity}(x, y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2} \sqrt{\sum_i y_i^2}} \tag{1}$$

Where:

- x and y are vectors representing two movies.
- x_i and y_i are the respective components of the vectors.

6.1.1 Genre Similarity

We use cosine similarity to measure the similarity between movies based on their genres. Movies are represented in a multi-dimensional space where each dimension corresponds to a genre (e.g., Crime, Drama, Adventure, Sci-Fi).

	tconst	primaryTitle	startYear	genres	Horror	Adventure	Animation	Western	Crime	Documentary	Adult	Family	Game-Show	Action	Drama	Mystery	War	Romance	News	Thriller	Biography	Talk-Show	Comedy
8829504	tt5637536	Avatar 5	2031.0	Action,Adventure,Drama	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
6234449	tt26217472	Rajju	2029.0	Adventure,Fantasy,Sci-Fi	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7423643	tt3095356	Avatar 4	2029.0	Action,Adventure,Fantasy	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
6234434	tt26217400	Aevum	2029.0	Adventure,Fantasy,Sci-Fi	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2246366	tt12310834	Kidnapping Love	2028.0	Comedy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 3: Cosine Similarity for Genres

Table 1: Genre Representation for Movies

Movie	Crime	Drama	Adventure	Sci-Fi
The Godfather	1	1	0	0
Interstellar	0	1	1	1
A Clockwork Orange	1	0	0	1

In the table above, each movie is represented as a vector in the genre space. The cosine similarity between these vectors helps us determine how similar the movies are in terms of their genres. For instance, "The Godfather" and "A Clockwork Orange" both belong to the Crime genre, resulting in a higher similarity score compared to movies from entirely different genres.

6.2 Time Similarity

We also consider the release year of movies. The similarity decreases exponentially with the absolute difference in release years. This approach helps in capturing the relevance of movies released around the same time period.

$$y = e^{-\frac{|x|}{10}} \tag{2}$$

Where:

- y is the similarity score.
- x is the absolute difference in years.

The exponential decay function illustrated above shows that as the year difference between two movies increases, their similarity score decreases. This reflects the idea that movies released closer in time are more likely to be similar in style, technology, and cultural relevance.

7 K-Nearest Neighbors (K-NN)

The K-NN algorithm sorts the neighbors of a liked movie based on similarity scores and selects the top-k most similar movies. This algorithm is particularly effective in collaborative filtering and recommendation systems.

7.1 Implementation of K-NN

In our implementation, we use a combination of genre and time similarity to determine the overall similarity between movies. The final similarity score is computed as a weighted average of the genre and time similarities.

$$S = \alpha \cdot G + (1 - \alpha) \cdot T \tag{3}$$

Where:

- S is the final similarity score.
- G is the genre similarity.
- T is the time similarity.
- α is a random variable between 0.5 and 1.

The K-NN algorithm then sorts the movies based on this final similarity score and selects the top-k movies as the most similar recommendations.

```
// Create similarity relationships 'IS_SIMILAR' and write back to the database
CALL gds.knn.write('title_basics_graph',
    {nodeLabels : ['title_basics'],
    nodeProperties: {genre_embeddings: 'COSINE', startYear: 'DEFAULT'},
    topK: 12,
    writeRelationshipType: 'IS_SIMILAR', writeProperty: 'similarity'});
```

Figure 4: K-Nearest Neighbors

By leveraging the K-NN algorithm, we can provide users with a list of movies that are most similar to the ones they have enjoyed in the past. This method ensures that the recommendations are relevant and tailored to the user's preferences.

8 Community Detection with Louvain Algorithm

The Louvain algorithm is a hierarchical clustering algorithm used for community detection. It recursively merges communities into a single node and executes the modularity clustering on the condensed graphs. This algorithm helps in identifying clusters or communities of movies that are highly related to each other based on their attributes.

8.1 Implementation of Louvain Algorithm

To implement the Louvain algorithm, we first projected the graph into memory using Neo4j's GDS library. This projection included nodes representing movies and their attributes, as well as the relationships between them.

The Louvain algorithm then optimizes modularity, which measures the density of links inside communities as compared to links between communities. Higher modularity indicates better-defined communities.

The results of the community detection revealed 187 communities with increasing modularities. This indicates that the algorithm effectively grouped movies into cohesive communities, which can then be used to provide more targeted recommendations.

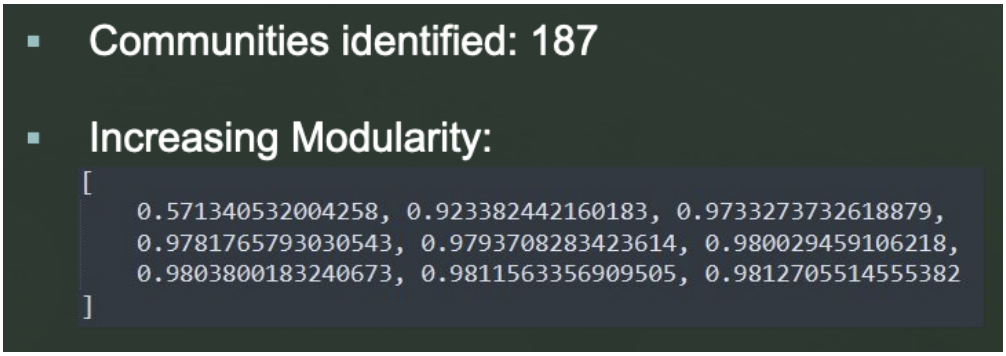


Figure 5: Community Detection Results

9 Implementation Details

9.1 Data Loading and Processing

The process of loading and processing the data involved several steps. Initially, we parsed the CSV files containing movie and actor information. Using batch processing, we loaded these datasets into Neo4j, ensuring that we efficiently handled the large volumes of data.

```
// Load titles in batch
:auto
LOAD CSV WITH HEADERS FROM 'file:///title_basics_7thMay.csv' AS row
CALL {
WITH row
CREATE(:title_basics_7thMay {tconst: row.tconst, primaryTitle: row.primaryTitle, startYear: toInteger(row.startYear), genres: split(
toFloat(row.Thriller), realityTV: toFloat(row.`Reality-TV`), gameShow: toFloat(row.`Game-Show`), biography: toFloat(row.Biography),
toFloat(row.Comedy), action: toFloat(row.Action), drama: toFloat(row.Drama), sciFi: toFloat(row.`Sci-Fi`), adventure: toFloat(row.Ad
} IN TRANSACTIONS OF 10000 ROWS;


// Create constraints to improve performance
CREATE CONSTRAINT title_basics_7thMay_tconst FOR (t:title_basics_7thMay) REQUIRE t.tconst IS UNIQUE;


// Create primaryTitle index
CREATE TEXT INDEX title_basics_index_7thMay FOR (n:title_basics_7thMay) ON (n.primaryTitle);

// Load names in batch
:auto
LOAD CSV WITH HEADERS FROM 'file:///name_basics_7thMay.csv' AS row
CALL {
WITH row
CREATE(:name_basics_7thMay {nconst: row.nconst, primaryName: row.primaryName, knownForTitles: split(row.knownForTitles, ',')})
} IN TRANSACTIONS OF 100000 ROWS;
```

Figure 6: Queries for data loading and processing

We also created constraints and indexes to optimize query performance. Constraints ensured that each movie and actor had a unique identifier, while indexes on movie titles improved the speed of search queries.

9.2 Creating Relationships

In addition to loading the data, we established relationships between different entities. For example, we created *ACTED_IN* relationships to link actors with the movies they acted in. This step is crucial for understanding the connections between different entities and for enabling more complex queries.

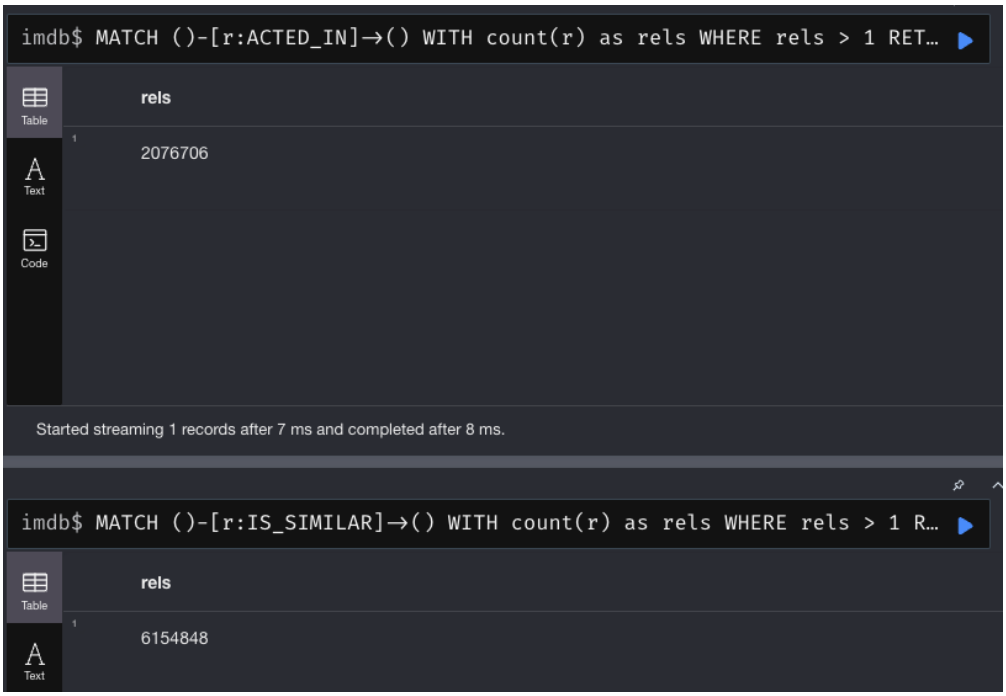


Figure 7: ACTED IN & IS SIMILAR count

To handle the relationships efficiently, we used the APOC library for batch processing. This allowed us to process relationships in parallel and in large batches, which significantly improved the performance.

9.3 Graph Projection and Similarity Computation

After loading the data and establishing relationships, we projected the graph into memory using Neo4j's Graph Data Science (GDS) library. This projection included nodes representing movies and their attributes, as well as the relationships between them.

Using the GDS library, we computed the cosine similarity between movies based on their genre embeddings. We also included the release year as a factor in the similarity computation. The similarity scores were then used to create *IS_SIMILAR* relationships between movies.

9.4 Community Detection with Louvain Algorithm

The final step in our implementation was community detection using the Louvain algorithm. By clustering the graph based on the similarity relationships, we identified communities of movies that share similar attributes. This clustering helps in providing more targeted and relevant recommendations to users.

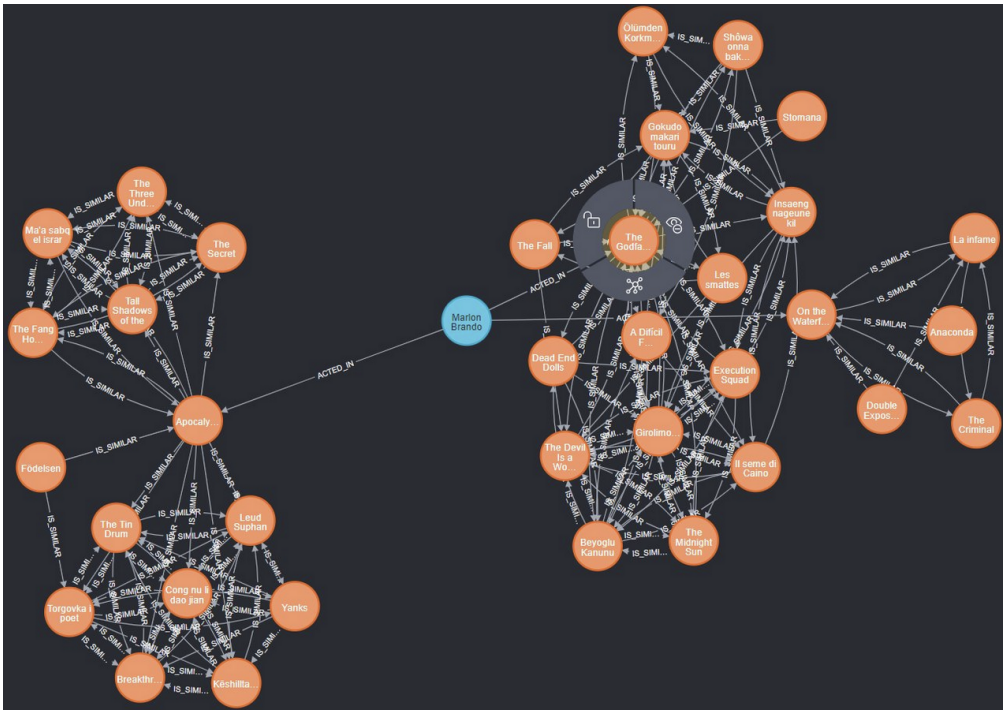


Figure 8: Zoomed in result of a community

The Louvain algorithm optimizes modularity, which is a measure of the density of links within communities compared to links between communities. Higher modularity indicates better-defined communities.

10 Challenges

Throughout the development of our movie recommender system, we encountered several challenges:

10.1 Performance Bottlenecks

Due to the sheer volume of data, performance bottlenecks were a significant concern. We mitigated this issue by narrowing down the IMDb titles to just 500K movies and using 3 million actors out of the initial 13M figures.

10.2 Platform Challenges

When considering multiple similarity metrics, the GDS KNN library averages the similarities, which prevents the use of weighted similarities. However, this made community detection easier with modular community sizes.

11 Future Work

Our future work will focus on several enhancements to the recommender system:

- Use actor nodes to provide better recommendations.
- Explore libraries that support weighted similarities.
- Integrate with a front-end application and collect user feedback on movie interests.
- Add user nodes in the graph to support collaborative filtering.

12 Conclusion

I hope this provided a detailed overview of how our movie recommender system works. We utilized IMDb datasets, various similarity algorithms, and the Louvain algorithm for community detection to deliver accurate recommendations. Future work will focus on enhancing the system with user feedback and additional data sources.