

Film Industry Analysis

Kira Fleischer, Tejas Ramesh, Sanchit Goel

1 Introduction

Our project aims to gain insights about collaborations and trends within the film industry as well as develop a movie recommendation system based on different similarity metrics. Specifically, we attempt to determine frequent actor-director collaborations that result in highest-rated movies and predict which movies a user may enjoy based on movies with a similar cast and crew, as well as movies with a similar plot and content. The primary dataset used for this analysis is the IMDb movie dataset, which we combine with plot synopsis data we scrape from IMDb and Wikipedia. We utilize Neo4J to model person-movie relationships, especially focusing on actor-director relationships. We use PostgreSQL to model our relational data, especially focusing on average movie ratings for different titles and people. Lastly, we utilize FAISS to model the vector data and create embeddings to represent the content of films. We combine all of the data from these different databases into a Jupyter Notebook to ultimately visualize the results of these queries.

1.1 Importance

This project has several important implications. For example, modeling actor-director collaborations that lead to highest-rated films can provide insights into which partnerships in Hollywood are the most successful. This could help casting directors decide which actors they should try and hire for a movie based on a given director in order to potentially maximize movie ratings and therefore profit.

Additionally, movie recommendation systems are extremely important for enhancing user engagement and satisfaction in streaming platforms. In this project, we implemented two types of recommendation approaches:

1. Node Similarity-Based Recommendation:

This method analyzes the similarity between nodes (movies) based on their neighborhood structure and relationships. By considering the shared collaborators, or co-actors, it suggests movies that are structurally similar to a given movie. This approach helps recommend movies that users might enjoy based on their preferences and previously watched content.

2. Content-Based Recommendation:

This approach recommends movies based on the content of the movies themselves, such as plot information. Using feature vectors and similarity measures, it suggests movies that are closely aligned with a user's previously watched movies, allowing for more personalized and relevant suggestions.

By combining these recommendation approaches, the system can provide a more robust and accurate recommendation mechanism that balances structural relationships in the graph with detailed content analysis. This not only improves the quality of recommendations but also ensures users discover content that matches their preferences while exploring similar genres and collaborators.

1.2 Data

Our project utilizes two main sources of data, which will be described here, and three databases, which will be detailed further in the Methods section.

IMDb Data

The IMDb data contains information about movies, cast and crew, average ratings, and more. We used this data for our actor-director collaboration modeling and the movie recommendation system based on cast and crew similarity. This data was stored in seven tables in TSV format from the IMDb website. Since these tables were all very large (multiple gigabytes of data), we had trouble loading this into Neo4j and PostgreSQL due to limited time constraints. Thus, we opted to sample only the first 100,000 rows of each table to import into Neo4j and PostgreSQL. Although this is not the best way to achieve a representative sample, it was sufficient for our analysis because we are not interested in the specific results of our queries, but rather the modeling and application of these queries. However, in the future, we would hopefully like to import all rows from each table into our databases to have a more holistic analysis, or alternatively sample rows in a more representative manner.

Scraped IMDb & Wikipedia Data: For content based recommendation, we scraped IMBD and Wikipedia for movie synopsis of around 1000 movies using beautifulsoup library.

2 Methodology

Our methodology of querying different databases and visualizing results can be depicted in Figure 1 below. Essentially, each different use case involved querying one database, sending the results into a second database, and visualizing the results in a Jupyter notebook. In order to connect these databases in our Jupyter notebook, we utilized the 'neo4j' Python driver to connect

to Neo4j and 'psycopg2' to connect to PostgreSQL. We used the python open source library FAISS to build the vector database. We used pandas to create dataframes to visualize the results of our queries in tabular format, and we utilized Matplotlib to visualize results graphically.

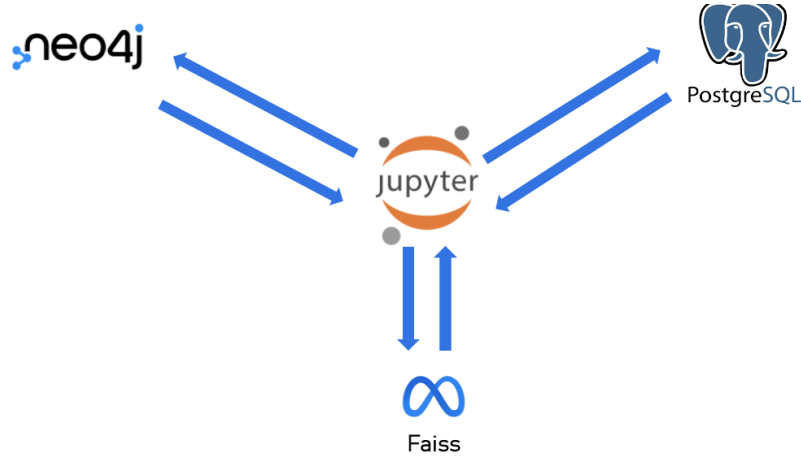


Figure 1: Querying pipeline

2.1 Neo4j

The data was modeled as a knowledge graph with two main node types and multiple relationships:

1. Node Labels:
 - a. Person (55.5k nodes) – Representing actors, directors, producers, and others associated with movies.
 - b. Movie (33.1k nodes) – Representing individual movies.
2. Relationships: Can be one of actor, actress, archive_footage, archive_sound, casting_director, cinematographer, composer, director, editor, producer, production_designer, self, writer; describing the interactions between Person and Movie nodes.

The following is the schema structure of the database: **(:Person)-[:relationship]->(:Movie)**, where 'relationship' can be one of the 13 types mentioned above. The data was loaded using Cypher's LOAD CSV command. All the required cypher queries used to setup the database can be found in the *Neo4j_setup_cypher.txt* file in the repository.



2.2 PostgreSQL

In PostgreSQL, we loaded all seven tables from the IMDb website, as seen in Figure 2 below. The primary keys for these tables were ‘tconst’, which uniquely identifies titles, and ‘nconst’, which uniquely identifies people. We opted not to include foreign keys when creating tables that referenced these primary keys because due to our sampling procedure, this would have excluded many rows of data which we did not want.

title_basics columns 9 tconst text titletype text primarytitle text originaltitle text isadult boolean startyear text endyear text runtime minutes text genres text	title_akas columns 8 titleid text ordering integer title text region text language text types text attributes text isoriginaltitle text	name_basics columns 6 nconst text primaryname text birthyear text deathyear text primaryprofession text knownfortitles text
title_ratings columns 3 tconst text averagerating numeric numvotes integer	title_episode columns 4 tconst text parenttconst text seasonnumber text episodenum text	title_principals columns 6 tconst text nconst text category text job text characters text
		title_crew columns 3 tconst text directors text writers text

Figure 2: PostgreSQL tables

The title_ratings table was perhaps the most important for our actor-director collaboration modeling because this data was not present in Neo4j, and this is what enabled our analysis of determining highest-rated partnerships. This table can be seen in Figure 3 below, with the ‘tconst’ column identifying the movie title, and the ‘averagerating’ column representing its associated rating. For this same analysis, we also utilized the ‘title_principals’ table (as seen in Figure 4 below) which maps movie titles to people, and this was important to determine which actors and directors were associated with each movie.

	tconst	averagerating
1	tt0000001	5.7
2	tt0000002	5.5
3	tt0000003	6.4
4	tt0000004	5.3
5	tt0000005	6.2
6	tt0000006	5
7	tt0000007	5.3
8	tt0000008	5.4
9	tt0000009	5.3
10	tt0000010	6.8

Figure 3: title_ratings table

	tconst	nconst	category
1	tt0000001	nm1588970	self
2	tt0000001	nm0005690	director
3	tt0000001	nm0005690	producer
4	tt0000001	nm0374658	cinematographer
5	tt0000002	nm0721526	director
6	tt0000002	nm1335271	composer
7	tt0000003	nm0721526	director
8	tt0000003	nm0721526	writer
9	tt0000003	nm1770680	producer
10	tt0000003	nm0721526	producer

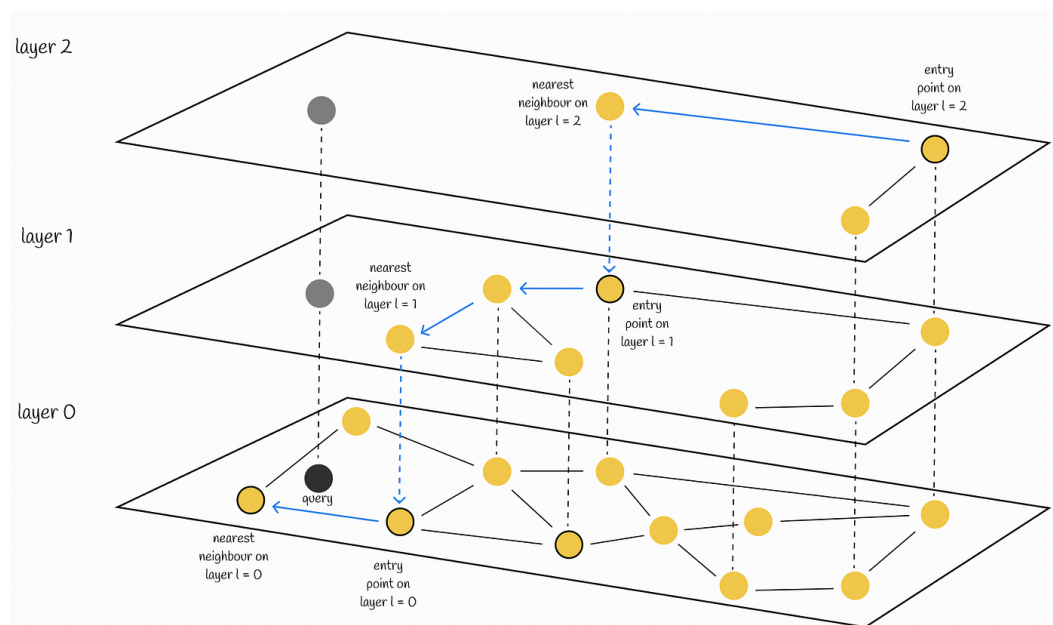
Figure 4: title_principals table

2.3 FAISS

Each plot synopsis was converted into a dense vector embedding using the Sentence-Transformer model (all-MiniLM-L6-v2). This embedding model effectively captures the semantic meaning of movie plots, enabling a similarity search that goes beyond simple keyword matching, facilitating more accurate and meaningful recommendations.

To efficiently handle similarity search among these embeddings, we employed Facebook AI Similarity Search (Faiss), an open-source library optimized for large-scale retrieval in high-dimensional spaces. Faiss stores these embeddings and performs fast approximate nearest-neighbor searches, making it ideal for real-time recommendation scenarios where rapid response times are crucial.

Within Faiss, we specifically utilized the Hierarchical Navigable Small World (HNSW) indexing algorithm. HNSW constructs a hierarchical graph-based structure of the embeddings, organizing them into multiple layers. Each higher layer contains fewer points, strategically chosen to represent clusters within the data. When a query embedding is introduced, HNSW quickly navigates through these hierarchical layers, starting from the sparsest, highest layer, progressively descending through more detailed lower layers. At each step, the algorithm selects the nearest neighbors to refine its search efficiently.



This hierarchical structure drastically reduces the complexity of similarity retrieval from linear or brute-force approaches down to approximately $O(\log N)$, where N is the number of embeddings. Consequently, even as the dataset scales to thousands or millions of items, retrieval remains fast and computationally manageable, allowing users to receive relevant and timely movie recommendations.

3 Results

3.1 Actor-Director Collaboration Modeling

The goal of this analysis was to determine which actors and directors frequently collaborate together and which collaborations lead to the highest rated films. For this use case, we work entirely in a Jupyter notebook, and we connect to Neo4j and PostgreSQL to send queries between the two databases. We began by writing a Neo4j query to determine actors and directors that have worked on the same movie, and we counted the number of collaborations they have together. We are only interested in partnerships that have worked together at least twice, and we limit the results to only include the top 100 partnerships for ease of computation. This query is depicted in Figure 5 below.

```
# Query Neo4j - find frequent actor-director collaborations
neo4j_query = """
MATCH (a:Person)-[r:actor]->(m:Movie)<-[d:director]-(dir:Person)
WITH a, dir, COUNT(m) AS collaborations
WHERE collaborations >= 2
RETURN a.nconst AS actor_id, a.primaryName AS actor_name,
       dir.nconst AS director_id, dir.primaryName AS director_name,
       collaborations
ORDER BY collaborations DESC
LIMIT 100;
"""
```

Figure 5: Neo4j query

After running this query, we store the results in a pandas dataframe, which can be seen in Figure 6 below. We send this resulting dataframe to PostgreSQL to determine the average film rating for each of these actor-director pairs, ensuring that we filter only actor-director pairs found from the Neo4j query (Figure 7). Then again, we create a pandas dataframe with the result of this query, as seen in Figure 8 below.

	actor_id	actor_name	director_id	director_name	collaborations
0	nm0580271	Jack Mercer	nm0281487	Dave Fleischer	434
1	nm0000305	Mel Blanc	nm0163332	Robert Clampett	227
2	nm0001908	Gilbert M. Broncho Billy Anderson	nm0001908	Gilbert M. Broncho Billy Anderson	189
3	nm0000305	Mel Blanc	nm0000813	Tex Avery	179
4	nm0424530	Arthur V. Johnson	nm0000428	D.W. Griffith	174

Figure 6: Neo4j output

```
pg_query = """
WITH ActorDirectorMovies AS (
    SELECT tp.nconst AS actor_id, tp2.nconst AS director_id, tr.averageRating
    FROM title_principals tp
    JOIN title_principals tp2 ON tp.tconst = tp2.tconst AND tp.nconst <> tp2.nconst
    JOIN title_ratings tr ON tp.tconst = tr.tconst
    WHERE tp.category IN ('actor', 'actress') AND tp2.category = 'director'
    AND (tp.nconst, tp2.nconst) IN %s
)
SELECT actor_id, director_id, AVG(averageRating) AS avg_rating
FROM ActorDirectorMovies
GROUP BY actor_id, director_id
ORDER BY avg_rating DESC;
"""
```

Figure 7: PostgreSQL query

	actor_id	director_id	avg_rating
0	nm0000305	nm0163332	6.4286343612334802
1	nm0000305	nm0000813	6.3581005586592179
2	nm0580271	nm0281487	6.3311059907834101
3	nm0000305	nm0293989	5.9850467289719626
4	nm0366008	nm0000428	5.9788135593220339

Figure 8: PostgreSQL output

Finally, we merge these two resulting dataframes into one, allowing us to determine actor-director partnerships with the highest number of collaborations and their associated rating. We can also visualize these results graphically, as seen in the resulting charts (Figures 9-11).

	actor_id	actor_name	director_id	director_name	collaborations	avg_rating
0	nm0580271	Jack Mercer	nm0281487	Dave Fleischer	434	6.3311059907834101
1	nm0000305	Mel Blanc	nm0163332	Robert Clampett	227	6.4286343612334802
3	nm0000305	Mel Blanc	nm0000813	Tex Avery	179	6.3581005586592179
4	nm0424530	Arthur V. Johnson	nm0000428	D.W. Griffith	174	5.6753164556962025
5	nm0366008	Robert Harron	nm0000428	D.W. Griffith	136	5.9788135593220339

Figure 9: Combined Neo4j + PostgreSQL output

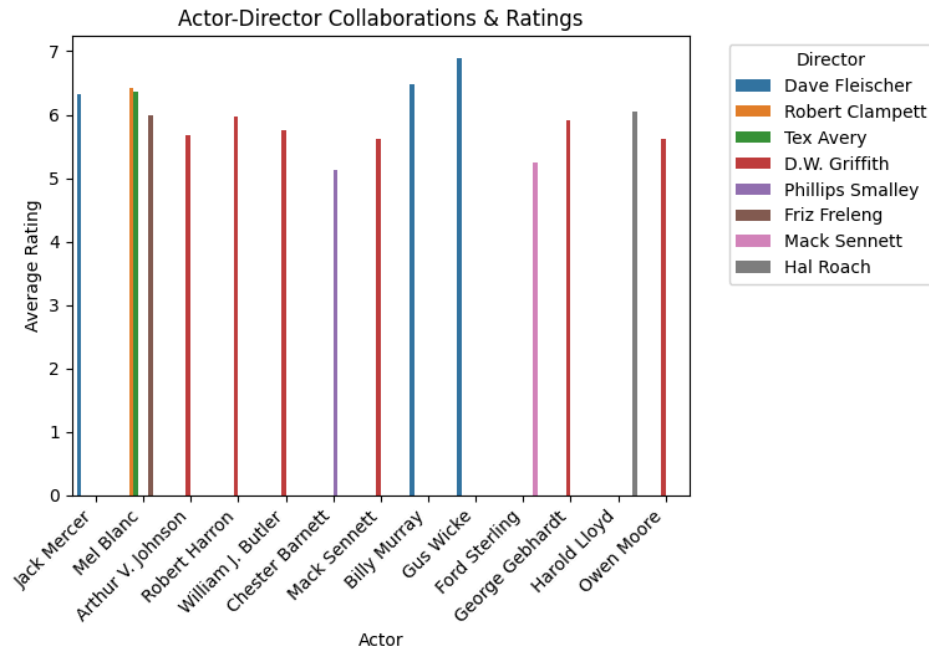


Figure 10: Plot of highest actor-director collaborations and their associated ratings

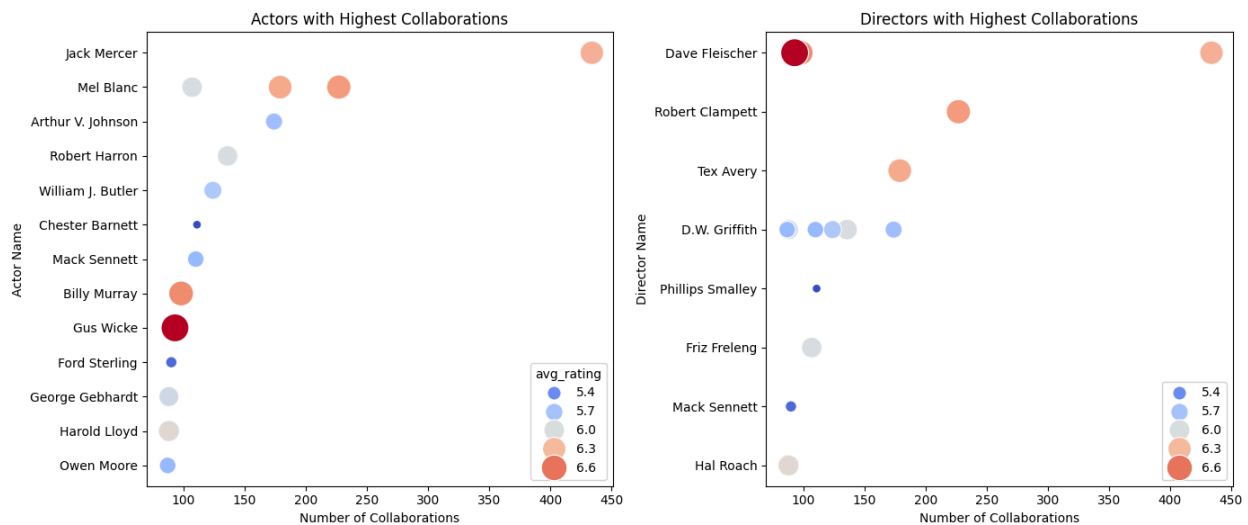


Figure 11: Additional plot of highest actor-director collaborations and their associated ratings

3.2 Movie Recommendations based on node similarity

The goal of this analysis was to build a movie recommendation system based on the similarity of movies using cast and crew information. For this use case, we worked entirely in a Jupyter notebook, connecting to Neo4j and PostgreSQL to send queries and retrieve data from both databases. This is how the data flows between the databases and our application.

1. User Input and Initial Query

The recommendation process begins when the user requests similar movies to a specific movie, such as "The Arrival of a Train". This request is sent from the application to the Neo4j database, where the system retrieves the corresponding movie node and initiates a similarity search.

```
#1. Query neo4j for the top_n movies
query = f"""
MATCH (m:Movie {{primaryTitle: $movie_name}})
WITH id(m) AS movieId
CALL gds.nodeSimilarity.stream('{graph_name}', {{
    similarityCutoff: 0.1,
    topK: {top_n}
}})
YIELD node1, node2, similarity
WHERE node1 = movieId OR node2 = movieId
WITH CASE WHEN node1 = movieId THEN node2 ELSE node1 END AS similarMovieId, similarity
MATCH (rec:Movie) WHERE id(rec) = similarMovieId
RETURN rec.tconst as tconst, similarity AS similarity_score
ORDER BY similarity DESC
LIMIT {top_n}
"""
```

Figure 12: Neo4j query to perform node similarity search

2. Node Similarity Calculation in Neo4j

Neo4j uses the GDS (Graph Data Science) library to compute node similarity, leveraging the Jaccard similarity metric to compare the neighboring nodes (cast and crew) of the target movie with other movies in the graph. A Cypher query is executed to identify the top 5 most similar movie nodes based on their shared collaborators (Figure 12). The query returns a list of these similar movies, providing the tconst (unique movie identifier) and the corresponding similarity score (Figure 13).

	tconst	similarity_score
0	tt0000070	0.300000
1	tt0000016	0.300000
2	tt0000089	0.272727
3	tt0000029	0.272727
4	tt0000041	0.200000

Figure 13: Neo4j's output to the application, wrapped as a pandas dataframe

3. Data Retrieval from PostgreSQL

Once the list of tconst values and similarity scores is generated, this data is sent to the PostgreSQL database. PostgreSQL performs a left join operation (Figure 14) to retrieve the relevant attributes of these movies. For each movie, the attributes include title, rating,

genres, and the similarity_score obtained from Neo4j. This step enriches the recommendations by adding descriptive metadata about the movies.

```
#2. Query Postgres for the corresponding details of movies
query = f"""
SELECT distinct d.primarytitle, d.genres, r.averagerating, t.similarity_score
FROM (SELECT unnest(array{tconst_values}) AS tconst, unnest(array{similarity_scores}) AS similarity_score) t
LEFT JOIN title_ratings r ON t.tconst = r.tconst
LEFT JOIN title_basics d ON t.tconst = d.tconst
"""
```

Figure 14: PostgreSQL query to lookup relevant information of the top 5 movies

4. Data Wrapping Using Pandas

The returned data from PostgreSQL is then wrapped in a Pandas DataFrame to format the results in a structured and user-friendly manner (Figure 15). The DataFrame contains columns for title, rating, genres, and similarity_score, making it easy to visualize and manipulate the data before displaying it to the user.

	title	genres	average_rating	similarity_score
0	Baby's Meal	Documentary,Short	5.9	0.2727272727272727
1	Bataille de neige	Comedy,Documentary,Short	6.7	0.2
2	Boat Leaving the Port	Documentary,Short	5.9	0.3
3	Demolition of a Wall	Documentary,Short	6.4	0.3
4	Leaving Jerusalem by Railway	Documentary,Short	6.2	0.2727272727272727

Figure 15: Final output displayed to the user

5. Displaying Results to the User

Finally, the DataFrame is presented to the user, showcasing the top 5 most similar movies along with their relevant attributes. This hybrid approach—using Neo4j for similarity computation and PostgreSQL for retrieving movie attributes—ensures that the recommendations are both fast and accurate, enhancing the overall user experience.

3.3 Movie Recommendations based on movie content

This method uses high-dimensional semantic embeddings derived from movie plot synopses, enabling the system to deeply understand thematic and narrative similarities between movies. By converting each movie’s plot into an embedding vector, we can rapidly compare and retrieve movies with closely matching storylines or themes.

A key advantage of this approach is its ability to overcome the cold-start problem—unlike collaborative filtering methods, which rely heavily on user interaction data, our method can immediately recommend relevant movies even for newly added or less popular titles. This makes it especially valuable for recommending niche films or recent releases without extensive user ratings or viewing history.

1. Generate embeddings

Movie plot synopses are transformed into semantic embeddings using the Sentence - Transformer model (MiniLM-L6-v2).

```
final_df = movie_df.merge(titles_df, left_on="imdb_id", right_on="tconst", how="left")
movie_df = movie_df[['imdb_id', 'title', 'plot_synopsis']].dropna()

# Step 2: Generate SBERT Embeddings
print("Generating embeddings...")
model = SentenceTransformer('all-MiniLM-L6-v2')
```

2. Build FAISS index

The generated embeddings are indexed into a FAISS vector database. Specifically, using the Hierarchical Navigable Small World (HNSW) algorithm, which allows efficient and scalable approximate nearest-neighbor retrieval in high-dimensional embedding spaces.

```
# Step 3: Build FAISS HNSW Index
embedding_dim = plot_embeddings.shape[1]
index = faiss.IndexHNSWFlat(embedding_dim, 32)
index.hnsw.efConstruction = 200

# Add embeddings to index
index.add(plot_embeddings)

# Save FAISS index
faiss.write_index(index, "movie_hnsw.index")
np.save("movie_ids.npy", movie_df['imdb_id'].values)
```

3. Recommendation System pipeline

The recommendation system pipeline accepts a user query—a movie title for which similar movies are desired. It retrieves the corresponding movie synopsis from our consolidated dataset and generates an embedding using the Sentence-Transformer model. This query embedding is then matched against the indexed movie embeddings in the FAISS vector database. Finally, the top-k movies with the most similar plot embeddings are returned as recommendations.

```

# Movie Recommendation Function
def recommend_movies(query_title, top_k=10):
    """Finds the top-k similar movies based on plot synopsis."""

    query_row = notna_merged_df[notna_merged_df['title'].str.lower() == query_title.lower()]
    if query_row.empty:
        return f"Movie '{query_title}' not found in database."

    query_embedding = model.encode(query_row.iloc[0]['plot_synopsis']).reshape(1, -1).astype('float32')

    distances, indices = index.search(query_embedding, top_k + 1)

    recommendations = []
    for i, idx in enumerate(indices[0][1:]):
        movie_title = movie_df.iloc[idx]['title']
        recommendations.append((movie_title, distances[0][i+1]))

    return recommendations


```

Output:

```

query_movie = "Hotel Transylvania"
recommendations = recommend_movies(query_movie, top_k=10)
print("Recommended Movies:", recommendations)

```

 Recommended Movies: [('Captain Horatio Hornblower R.N.', 0.74598217), ('The Dunwich Horror', 0.85303336), ('And the Children Shall Lead', 0.89539623), ('

4 Limitations

One major limitation of this project was the size of the dataset we were working with. As previously mentioned, the IMDb data was too large to load completely into our databases, so we opted for sampling the first 100,000 rows of each table, rather than attempting to load everything. Since the titles in these tables were roughly in chronological order, this meant that we ended up sampling approximately the earliest 100,000 films in the dataset. As a result, our analysis inevitably was skewed towards earlier movies, and we lacked sufficient data from more recent years. In the future, given more time and resources, we hope to load all data from these tables, leading to a more robust and comprehensive analysis.

5 Conclusion and Future Work

From this film industry analysis, we were able to successfully deliver two different movie recommendation approaches, graph structure based and content based. Additionally, we were successfully able to model actor-director collaborations to determine which partnerships most frequently led to highest film ratings, and which members collaborated most frequently with others. This leads to valuable insights about which actors and directors are the most desirable to work with and have the highest probability of creating a successful film.

For future work, we are interested in potentially determining which actors are the most popular based on their average rating and number of films starred in. Ideally we could combine this with

geographic data about actor birthplaces to determine if there is any correlation between location and probability of future success. Additionally, we are interested in modeling movie genre trends over time. Perhaps we could combine this data with historical news/popular headlines data to determine if current events lead to differing movie genre popularities over time. We can also come up with a sophisticated recommendation algorithm which takes into account the content of movies, common cast & crew, genre information, time of release, etc. to deliver recommendations to users using a holistic approach.