Knowledge and Data Engineer Report

Nikita Aksjonov, Nikos Kessidis, Bar Melinarskiy, and Konstantinos Zavantias

Introduction and Project Overview

Objective

This project demonstrates the potential of Semantic Web technologies by implementing a personalized movie recommendation system. The system features a web-based interface that allows users to interact with the underlying knowledge graph, providing an intuitive and user-friendly way to discover relevant films.

As illustrated in Fig. 1, users can select a specific movie to find similar movies based on shared characteristics. Additionally, users can refine their search results by specifying preferences such as genre, actors, directors, and country. The backend, developed using Python and the FastAPI library [1], processes these inputs. It leverages the knowledge graph and plot embeddings to generate a curated list of the most relevant movie recommendations based on their similarity to the selected movie and the user-specified criteria. In the results section, we will demonstrate some of the possible queries that can be performed using our application.

Dataset

Dataset Creation from DBpedia

The movie dataset was extracted from the remote end-point of DBpedia using SPARQL queries through a custom Python implementation [2] (see file *get_data_from_dbpedia.py* in GitHub repository. The extraction process was tailored to meet the specific requirements of the application, ensuring high data quality and relevance for the movie recommendation system.

Data Collection Strategy

The fetching process employed three complementary strategies to maximize the coverage of relevant movie information from DBpedia:

1) **Temporal Coverage:** Movies were collected from 1990 to 2024, ensuring the coverage of both classic and modern films.
2) **Popular Movies:** A list of 150 well-known movie titles was used as the starting point for extraction (e.g. Shrek, Titanic, and Inception), ensuring that only widely-viewed films were included.
3) **Production Companies:** The top 100 production companies by movie count were identified and their complete filmographies were included. This method helped to capture commercially significant films.

Extracted Attributes

For each movie, we extracted essential information that would be valuable to generate recommendations.

- **Core Metadata:** Movie URI, title, country, language, release date, runtime, and plot summary.
- **Creative Elements:** Genres, directors, actors, producers, and writers.
- **Additional Context:** Production companies, franchise affiliations, and external references (IMDb and Rotten Tomatoes IDs).

These attributes were specifically chosen to enable content-based filtering, with a particular emphasis on elements that connect different movies together (such as shared genres, actors, or franchise associations).

Technical Implementation

The data extraction was implemented using Python with the SPARQLWrapper library to query DBpedia's SPARQL endpoint [3]. To handle large-scale data collection efficiently, the implementation included:

- **Batch processing** - Queries were performed in chunks to manage rate limits, given the DBpedia endpoint's 30-second timeout.
- **Failsafe mechanism** - Automatic retries for failed queries with a maximum number of attempts to avoid infinite loops.
- **Deduplication** - Removal of overlapping results based on object URIs.
- **Data validation** - Standard cleaning procedures for literals such as strings and dates.

The fetched dataset was saved in CSV format and converted into RDF using the RDFLib library [4], as detailed in the Pre-Processing section. The resulting knowledge graph, shown in Table 1, comprises 32,050 films enriched with available information, resulting in over 90,000 triples. Each movie entry includes attributes such as genres, countries, runtime, budget, actors, and directors. Not every movie had all attributes available; detailed counts for each attribute (predicate) are provided in Table 2.

Statistics

In this section, we present various statistics of the constructed knowledge graph.

Statistic	Value
# of triples	90,515
# of subjects	123,036
# of predicates	32
# of objects	34,618
# of films	32,050

Table 1. Dataset Statistics. Overview of the dataset, including the number of triples, subjects, predicates, objects, and films.

Predicate	# of Subjects	# of Objects
rdf:type	90,515	19
rdfs:subPropertyOf	34	32
rdfs:subClassOf	345	26
rdfs:domain	25	4
rdfs:range	25	17
rdfs:label	123,036	123,043
dbo:abstract	32,050	34,196
dbo:country	29,816	248
dbo:director	25,556	8,806
dbo:genre	841	239
dbo:language	30,338	662
dbo:mainSubject	32,052	34,618
dbo:releaseYear	10,873	118
dbo:runtime	28,039	1,862
dbo:starring	25,644	32,296
dcterms:subject	32,539	22,343
prov:wasDerivedFrom	32,552	32,552
dbo:distributor	20,146	1,747
dbo:producer	16,797	6,690
dbo:productionCompany	19,649	2,580
dbo:cinematographer	13,822	2,081
dbo:composer	15,134	4,193
dbo:writer	19,381	11,132
dbo:boxOffice	6,889	8,574
dbo:budget	5,766	3,418
dbo:imdbID	5,565	5,564
dbo:rottenTomatoesID	182	182
dbo:series	15	15
dbo:plotEmbedding	32,050	32,050

Table 2. Unique Counts per Predicate. Unique counts of subjects and objects for each predicate in the dataset. Prefixes used:

rdf: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
rdfs: `http://www.w3.org/2000/01/rdf-schema#`
dbo: `http://dbpedia.org/ontology/`
dcterms: `http://purl.org/dc/terms/`
prov: `http://www.w3.org/ns/prov#`

Pre-Processing

After extracting the movie data into CSV format, the next crucial step involves converting this tabular data into RDF (Resource Description Framework) using a Python script. Here are the key aspects of this transformation:

- **Country formatting:** The code standardizes country names, resolving variations (e.g., "USA" to "United States") and linking them to their corresponding DBpedia URIs. This ensures that different text variations and typos are handled by the script and movies from the same countries are in the same category.
- **Genre formatting and Organization:** Genre text was pre-processed to remove typos, excessive symbols, and incorrect formatting. Moreover, genres are organized into a hierarchical structure, with super-genres (like Comedy, Drama, and Action) and their respective sub-genres, ensuring a more organized representation of movie genres, for that we used (*rdfs:subClassOf*).

The next step of preprocessing involves creating RDF relationships between movies and their attributes, converting literal values to appropriate data types, generating embeddings for movie plots using the SentenceTransformer model [5], and adding related entities with URIs and labels for entities such as actors, directors, and production companies. Finally, the creation of triples was split according to object type: literals (e.g., strings, integers, dates like release year, budget, language) and objects (e.g., countries, actors, directors). This process constructs an RDF graph using the RDFLib library and serializes it into Turtle format (TTL file), thereby creating a structured knowledge graph for the movie dataset.

To enrich the RDF graph with additional semantic information, we incorporated a comprehensive ontology that defines relationships and classifications among various entities in the dataset. This ontology introduces hierarchical structures, such as subclass and subproperty links, to establish a well-defined taxonomy within the graph. For instance, movies are classified under the dbo:Film class, while genres are associated with movies using the dbo:genre property. Similarly, actors, directors, and production companies are linked to movies via properties like dbo:starring, dbo:director, and dbo:productionCompany.

The ontology also captures numerical and categorical attributes. Using the RDFS.domain and RDFS.range semantics, we describe the relationships of properties such as dbo:runtime (movie duration), dbo:budget (financial expenditure), and dbo:releaseDate (release year), which represent critical metadata about the films. Geographic information is integrated using the dbo:country property to link movies with their associated countries. By defining these relationships, the ontology provides a structured and semantically rich dataset, enabling enhanced querying and analysis.

Additionally, we extended the ontology to include support for embedding-based semantic searches. Movie plot embeddings, generated using the SentenceTransformer model, are linked to movies through a custom property (dbo:plotEmbedding). These embeddings enable advanced similarity queries based on the semantic content of plot summaries.

This structured approach not only standardizes the dataset but also enables complex queries, reasoning, and inferencing. It transforms the dataset into a robust knowledge graph capable of supporting personalized recommendations, advanced analytics, and semantic reasoning.

This graph was then uploaded to a local instance of GraphDB, a semantic graph database [6]. The choice to use GraphDB for storing the knowledge graph was driven by its robust support for SPARQL queries, which allows for efficient querying and manipulation of RDF data. Additionally, GraphDB provides advanced features such as reasoning, inferencing, and full-text search, which enhance the capabilities of the movie recommendation system by enabling more complex and meaningful queries.

Techniques Used

Frameworks

For the user interface (UI), the Dash framework was used, a powerful tool for building simple but interactive web applications in Python [7]. Dash provides a simple and intuitive way to create dynamic pages, which makes it ideal for implementing a movie-recommendation app that allows users to interact with it easily [7].

For the backend, we leveraged FastAPI, a modern web framework known for its high performance and ease of use [1]. FastAPI excels in handling asynchronous requests, making it well-suited for fetching, processing, and storing movie data efficiently. Additionally, by keeping the query logic in the backend, the UI is decoupled from the specifics of the database. This approach allows the UI to remain simple and focused on user experience, without being affected by the complexities of database management.

To further optimize performance, we integrated caching mechanisms using the FastAPI Cache and Redis libraries [8, 9]. These caching solutions temporarily store frequently requested queries in memory, significantly reducing latency and enhancing response times. The processed data is then seamlessly delivered to the Dash framework, ensuring a responsive and interactive user experience.

Embeddings

One of the properties included in the graph database is the plot of the film. By encoding the plot text into embeddings, we can enhance the process of finding similar movies based on their descriptions. To achieve this, we utilized the sentence-transformer model *all-MiniLM-L6-v2* [10]. This model encodes input text into a vector that captures its semantic information. These sentence vectors can then be used for various tasks such as information retrieval, clustering, and sentence similarity [11]. These vectors were then converted into string property and saved as an additional property of the respective movie.

In addition to counting shared properties such as genres and actors, we can use the embedded plot to better calculate similarities between movies using cosine similarity. Cosine similarity measures the cosine of the angle between two vectors in an embedding space, effectively capturing the similarity between two sentences by comparing their vector representations [12]. This metric, commonly used in natural language processing tasks, ranges from -1 (completely dissimilar) to 1 (completely similar), with 0 indicating no similarity. The formula for cosine similarity is:

$$\text{Cosine Similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

where $\mathbf{A}$ and $\mathbf{B}$ are the vector representations of the two sentences, and $\|\mathbf{A}\|$ and $\|\mathbf{B}\|$ represent their magnitudes. This formula effectively estimates how similar two vectors are based on their direction in the vector space [13].

Results

The developed application empowers end-users to execute both simple and complex queries, enabling them to thoroughly explore the curated movies knowledge graph. This functionality allows users to uncover intricate relationships and insights within the dataset, enhancing their understanding and discovery of movie-related information.

Using the supplied filters panel we can answer queries such as:

- **Simple Queries:**
  - Fetch the title of specific entities such as movies, countries, or genres by their names. This is used to fill the filters' options.
  - Retrieve a list of all movies directed by a particular director.
  - Get a full list of available properties of given movies.
- **Complex Queries:**
  - Find movies that share the same genre and were released within a specific time frame.
  - Retrieve movies that feature a particular set of actors and were directed by a given director.
  - Lastly, given a selected movie, we can query for movies that are similar based on plot embeddings and other shared attributes such as genres and directors. The similarity score is calculated as a weighted sum of the shared attribute counts and the cosine similarity score of the plot embeddings.

In Fig. 1, we present an example query for movies similar to *The Lord of the Rings: The Fellowship of the Ring*. Each movie's card in the results section displays the most informative features available, such as genres, directors, and actors. Additionally, the similarity score for each respective movie is shown in the bottom right corner of the card, providing a quantitative measure of similarity based on shared attributes and plot embeddings.

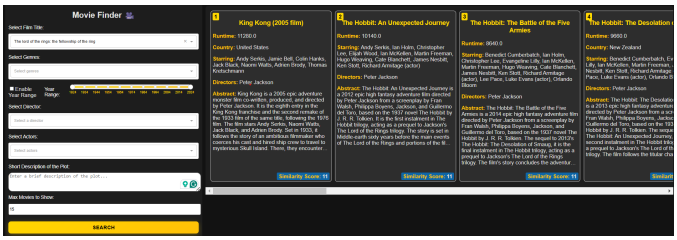

Fig. 1. User Interface. Demonstrating the process of fetching movies similar to *The Lord of the Rings* using the application's interface.

Known Limitations

Unresolved Queries

Given the limited time that was available, the following queries could not be answered:

- **Dynamic Text Input:** The lack of built-in similarity measures like cosine similarity in GraphDB restricted

handling complex queries. Finding similar movies based on user-submitted free text was not feasible as multiple similarity measures had to be implemented.
- **Language Input:** The system only handled English labels, as only English data was fetched during dataset creation. Consequently, inputs in other languages could not be processed.
- **Weight of Selection:** Due to computational cost and complexity, the feature allowing users to prioritize certain properties (e.g., directors over movie titles) in similarity measures was not implemented.
- **Supporting Complex Similarity Queries:** The naive approach to similarity calculations made it difficult to extend the system to support complex queries, such as finding movies similar to a list of movies, similar to Netflix's algorithm that considers the full watch history.

Limitation Overcome

The primary reason many of these queries could not be implemented was the time limit. As their contribution to the result and the difference that could make in user's experience was relative minor compared to the effort required and that is why they were deprioritized.

- **Dynamic Input and Complex Query Handling:** Enhancing dynamic input handling could involve leveraging Natural Language Processing (NLP) models to interpret user input. This approach would allow the system to process incomplete, misspelled, or ambiguous queries effectively. For example, a spell-check and auto-suggestion mechanism could refine the input, while semantic similarity techniques would match the query with the most relevant movies, even if the phrasing differs from the dataset. Additionally, exploring alternative databases that support free-text similarity searches, such as Neo4j, could be a viable option.
- **Multilingual Support:** Supporting multilingual input would require translating all data into multiple languages. While this enhances accessibility, it would significantly increase computational demands, necessitating optimization to manage the added complexity.

RDF & Semantic Web Technologies

RDF data models and Semantic Web Technologies offer significant advantages, such as enhanced data interoperability through standardized formats and improved knowledge representation via formal ontologies. However, they also face certain limitations.

- **Complexity:** RDF's triple-based structure and SPARQL queries require specialized knowledge to create and execute complex queries accurately [14]. The subject-predicate-object format can limit the representation of nuanced relationships, such as temporal dependencies, making it challenging to model certain real-world scenarios [15].
- **Scalability:** Querying and reasoning over large RDF graphs can be computationally expensive, leading to performance issues, especially with complex queries. Researchers are addressing this with specialized storage systems and efficient query engines to mitigate performance bottlenecks [16, 17, 18].
- **Privacy and Security:** Designed for open data sharing, RDF and Semantic Web technologies lack inherent security and privacy features. Managing permissions and enforcing strict privacy controls can be complex, posing challenges for secure data integration [14].

Triple Reduction

Given that this project was developed locally, several steps were taken to manage the size of the knowledge graph. Firstly, we focused our data fetch on relatively recent movies. Additionally, to reduce the number of triples, we set English as the primary language for our application, filtering out non-English entries to avoid redundancy from properties available in multiple languages. Lastly, since movies were fetched using different strategies (as listed previously), we implemented a deduplication process based on the movies' URIs to eliminate duplicate entries.

Lessons Learned

Challenges and Mitigation Strategies

Technological Complications and Setup Difficulties

A primary challenge was ensuring seamless communication between the web interface, backend application, and database. Each service had unique dependencies and configurations, complicating the initial setup. Integrating all services into a Dockerized environment required careful coordination.

For new users, installing and configuring dependencies added complexity. We addressed this by creating a streamlined setup process, merging configuration files, automating dependency installations, and providing clear documentation.

Managing the large RDF dataset files (in TTL format) was another challenge. Due to their size, the dataset had to be managed independently via OneDrive.

Instability of DBpedia Endpoint

Fetching datasets from DBpedia was challenging due to endpoint instability. Large requests often resulted in connection refusals or timeouts. To mitigate this, we implemented a data extraction pipeline that fetched data in smaller batches and included a deduplication step to ensure data integrity.

Construction of Queries

Supporting complex queries from the user interface was difficult. Constructing queries based on user-provided filters required dynamically building queries for each input. This added complexity to the query construction process.

Fetching movies similar to multiple user-selected films was also challenging. Determining similarities between multiple movies required a complex logic that was not straightforward to implement.

Reflections and Improvements

Given the time limitations of the project, we were unable to fully explore and leverage the advanced similarity functions

embedded within the GraphDB database. Due to the database's limitations, we had to encode movie plots into a single embedding as a string, as GraphDB does not natively support list formats for storing embeddings. While this approach allowed us to store and retrieve embeddings of plot data, it resulted in an overwhelmingly large dataset that became increasingly difficult to manage and process efficiently, resulting in scalability issues.

Our lack of practical experience with GraphDB, especially in managing large-scale data and optimizing queries for similarity searches, extended the development time. While GraphDB is powerful for knowledge graph management, it may not be the best fit for this project due to limitations in handling specific data types, such as text embeddings.

An alternative solution is Neo4j, which is also designed for graph-based data but offers more flexibility with list-based structures and handles large datasets more efficiently. Neo4j's built-in support for advanced graph algorithms could enable more sophisticated similarity measures, improving movie recommendation quality [19]. This could enhance system scalability and simplify development, particularly when dealing with complex data relationships [13].

Contributions of Each Group Member

- **Nikita Aksjonov** - Developed the UI and implemented movie plot embeddings.
- **Bar Melinarskiy** - Extracted data from DBpedia, Dockerized the application, created the environment setup script, and developed the REST service.
- **Konstantinos Zavantias** - Also helped develop the UI, converted the dataset from CSV to RDF (TTL) format, and developed the similarity algorithm.
- **Nikos Kessidis** - Performance measurements.

References

[1] Sebastián Ramírez. *FastAPI: The modern, high-performance web framework for building APIs with Python 3.7+*. URL: https://fastapi.tiangolo.com.

[2] DBpedia. *DBpedia - A crowd-sourced community effort to extract structured information from Wikipedia and make this information available on the Web*. Accessed: 2023-10-01. 2023. URL: https://www.dbpedia.org.

[3] SPARQLWrapper Developers. *SPARQLWrapper: A Python wrapper for SPARQL services*. Accessed: 2023-10-01. 2023. URL: https://rdflib.github.io/sparqlwrapper/.

[4] rdflib Developers. *rdflib: A Python library for working with RDF*. Accessed: 2023-10-01. 2023. URL: https://rdflib.dev/.

[5] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. Accessed: 2023-10-01. 2019. URL: https://www.sbert.net/.

[6] Ontotext. *GraphDB: The Fastest RDF Database*. Accessed: 2023-10-01. 2023. URL: https://www.ontotext.com/products/graphdb/.

[7] Plotly Technologies Inc. *Dash: A Python framework for building web applications*. URL: https://dash.plotly.com.

[8] FastAPI Cache Developers. $fastapi_cache$: $Acachinglibraryfor FastAPI$. Accessed: 2023-10-01. 2023. URL: https://github.com/long2ice/fastapi-cache.

[9] Redis Labs. *Redis: An open source, in-memory data structure store, used as a database, cache, and message broker*. Accessed: 2023-10-01. 2023. URL: https://redis.io/.

[10] Nils Reimers and Iryna Gurevych. *all-MiniLM-L6-v2: A Sentence-Transformer model*. Accessed: 2023-10-01. 2020. URL: https://www.sbert.net/docs/pretrained_models.html#sentence-embedding-models.

[11] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: https://arxiv.org/abs/1908.10084.

[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge, England: Cambridge University Press, 2008. URL: https://nlp.stanford.edu/IR-book/.

[13] Aminu Da'u, Aminu Da'u, and Naomie Salim. "Recommendation system based on deep learning methods: a systematic review and new directions". In: *Artificial Intelligence Review* 53.4 (2020), pp. 2709–2748. DOI: 10.1007/S10462-019-09744-1.

[14] Sabrina Kirrane et al. "Privacy, security and policies: A review of problems and solutions with semantic web technologies". In: *Semantic Web* 9.2 (2018), pp. 153–161. DOI: 10.3233/SW-180289. eprint: https://doi.org/10.3233/SW-180289. URL: https://doi.org/10.3233/SW-180289.

[15] M. Dean et al. *OWL Web Ontology Language Reference*. English. Dean04a. World Wide Web Consortium, 2004.

[16] Wria Mohammed and Alaa Jumaa. "A SURVEY ON USING SEMANTIC WEB WITH BIG DATA: CHALLENGES AND ISSUES". In: *Harbin Gongye Daxue Xuebao/Journal of Harbin Institute of Technology* 54 (Mar. 2022), pp. 93–103.

[17] Anh Le-Tuan et al. "Pushing the Scalability of RDF Engines on IoT Edge Devices". In: *Sensors* 20.10 (2020). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/20/10/2788.

[18] Olivier Pelgrin. "Expressive Querying and Scalable Management of Large RDF Archives - Source code and Experiments". Version 1.0. In: (June 2024). DOI: 10.5281/zenodo.11517199. URL: https://doi.org/10.5281/zenodo.11517199.

[19] *Neo4j: The Graph Database Platform*. URL: https://neo4j.com/.