



---

# Suggestions based on Graph Relations and Syntactic Correlations

---

A Report by

**Arafath Cherukuru**

**Praveen Potluri**

**Sai Tummala**

CSE 740/722: Large Scale Machine Learning and Big Data



DECEMBER 13, 2014  
UNIVERSITY AT BUFFALO

## Retrieval of Data:

After the initial analysis on Data from Wikipedia Infoboxes, it was not reliable to use as a perfect source for movie information. So, the data source we used for this project is DBpedia.

DBpedia is a crowd sourced community effort to extract structured information from Wikipedia. We can download this data from the datasets made available by DBpedia. Also can be accessed through a SPARQL endpoint which in turn has all the datasets of DBpedia loaded into it, so that the data can be queried easily from interface.

SPARQL is an RDF query language which is regularly used to retrieve and manipulate data stored in RDF format. We use Apache Jena which provides an API to connect to DBpedia SPARQL interface and query the interface with SPARQL queries.

```
SELECT DISTINCT ?film ?id
WHERE {?film dbo:wikiPageID ?id .
?film rdf:type dbo:Film .
?film dbpedia2:country "United States"@en .
?film dbpedia2:language "English"@en .
OPTIONAL {?film dbo:releaseDate ?releasedate .
    FILTER (?releasedate > "1950-01-01"^^xsd:date). }
?film rdfs:label ?label .
FILTER (LANG(?label ) = 'en') .
FILTER regex(?label , "[A-Za-z](.*)*$", "i" ) . }
ORDER BY ?film
```

The above query is used to query the interface to get the list of movies which belong to country - "United States" and language – "English" which movies were released after 1950. We store this data in a Hashmap and iterate over the movie\_ids. For each movie\_id, we query the SPARQL interface to get various movie properties.

```
SELECT ?film ?property ?data
WHERE {
?film dbo:wikiPageID $id .
?film rdf:type dbo:Film .
?film ?property ?data . }
```

We store various properties from the movie data like "Actors, Directors, Producers, Writers, Categories, etc." We consider all these properties as semantic connections between two movies. So, we try to extract these edges from the data retrieved. If D1,D2,D3 have the same property, then we extract the data in the below fashion so that we can construct a graph based on this data in Neo4j.



## Creating the graph in Neo4j:

As described above, need to persist node and edge information. We use Neo4j, an open-source graph database to store the node and edge information. Neo4j is an embedded, disk based, fully-transactional Java persistence engine that stores data structured in graphs rather than in tables.

We have extracted information of about 20,000 movies. Each movie information is stored in a separate text file with movie id as the name of the file. All the files are parsed and nodes are created in Neo4j. We make use of the embedded database feature provided by Neo4j. This is essentially creating an in memory database server rather than a stand-alone one. Neo4j provides api for creating and accessing the embedded database.

For each node created, we store information that describes the node as properties. Some of the properties that we capture are starring, producer, director, category etc. We are also creating an index based on the movie ids. Because of creation of index, the node creation time increases. But, as we will see later maintaining an index for nodes reduces the time for creating edges between nodes.

Once we have all the nodes created, we will start the creation of edges between nodes. All the edge information is stored in a single file. We iterate through all the lines in the file and create edges accordingly. Two nodes are connected if they have any common property. A nice feature provided by Neo4j is the ability to add properties to an edge. We utilize this feature to capture the common property information between two nodes and store it as a property of the edge.

The properties store in the edges are used to calculate the weight information for each edge. Based on our observations, we have assigned the following weights:

- Starring – 0.25
- Director – 0.175
- Writer – 0.15
- Producer – 0.125
- Music – 0.10
- Editor – 0.10
- Camera – 0.10

The final weight for the edge is calculates using the formula:

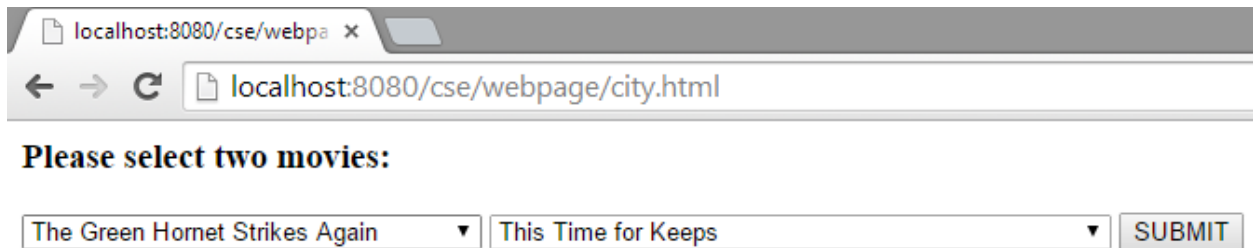
**Weight = (0.25 \* common\_stars) + (0.175 \* common\_directors) + (0.15 \* common\_writers) + (0.125 \* common\_producers) + (0.10 \* common\_musicians) + (0.10 \* common\_editors) + (0.10 \* common\_camera\_men)**

## Suggesting movies based on the graph:

We have a provided a simple UI, where the user can select two movies that he/she has watched. This information is sent back to the Spring service that we implemented. For each movies selected, we fetch all the connected nodes and aggregate the list. If there is movie that is connected to both movie1 and movie2, then the final weight of that movie is the sum of weight with movie1 and weight with movie2. Finally, the aggregated list is provided to the user with their corresponding weights.

## Results:

A screenshot of the UI.



A screenshot of a web browser window. The address bar shows 'localhost:8080/cse/webpage/city.html'. The page content includes the text 'Please select two movies:' followed by two dropdown menus. The first dropdown menu is set to 'The Green Hornet Strikes Again' and the second is set to 'This Time for Keeps'. To the right of the dropdowns is a 'SUBMIT' button.

Few results:

When we select Beowulf and Resident Evil which are scary/thriller genre, we can see that the recommended movie is Scream.



A screenshot of a web browser window, similar to the one above. The address bar shows 'localhost:8080/cse/webpage/city.html'. The page content includes the text 'Please select two movies:' followed by two dropdown menus. The first dropdown menu is set to 'Beowulf' and the second is set to 'Resident Evil'. To the right of the dropdowns is a 'SUBMIT' button.

### Recommended Movies

Movie	Similarity Score
Scream (1996 film)	0.1