ADBI: Final Capstone Project

Team:

Members (First, Last)	username	email
Vishnu Ramachandran	vcramach	vcramach@ncsu.edu
Mohd. Sharique Khan	mKhan8	mKhan8@ncsu.edu
Natansh Negi	nnegi2	nnegi2@ncsu.edu

Datasets:

For the dataset of our project we used approximately 50000 movies from IMDB. This list of movies was used for us to extract information from Wikipedia. From Wikipedia, the summary of approximately 9000 movies was extracted using a scraper(Scrapy) and the different attributes for each were stored as json objects. This allowed us to easily segregate movie names and their respective summaries for us to process the data further. So the data we now have is partially structured and we could work on the summaries alone to extract entities and their relations

Technique/algorithm:

1. Preprocessing of the summaries for each movie was the first step of the project. Spacy was used to apply part of speech (POS) tagging so as to both tokenize and tag the different words based on their parts of speech. This allowed us to set certain orderings among the parts of speech by trial and error to create triples of entity-relationship-entity tuples for further usage.
2. Another method we used for extracting relationship tuples was to use the Stanford OpenIE library to find relationships between the different words and it let us observe certain orderings amongst the different parts of speech. This allowed to create “regexes” of combinations of parts of speech to match them with their occurences in the actual summaries of the movie dataset we used.

3. Once we extracted all the named entities and relations from our dataset we created a node for each entity and stored them in a Dictionary. For each entity we create a node, if that node is already present in the Dictionary we use that node instead of creating a new one.
4. We used Neo4j graph database for storing our knowledge graph. For integrating Neo4j with python we used Py2neo library. We also integrated Neo4j with elasticsearch so whenever a new node is created in Neo4j the corresponding details are also pushed to the elasticsearch server.
5. Once we had the properly structured data in the form of entities and relationships we created our knowledge graph in Neo4j.
6. In addition to this we created an application in Python flask which serves as an interface for the end user to interact with the knowledge graph. The application was deployed on our local server.
7. We exposed 3 APIs which can be which could be used to query the database. The APIs would use name of a ‘PERSON’,’MOVIES’ name and ‘DATES’ as an input. We used ‘Random Walk’ API provided by Neo4j to randomly get the data from knowledge graph based on user input and display the results on the flask application.
8. We then compared our results with Google’s knowledge graph API.

Summary

Through this project we were able to create a version of the Google Knowledge Graph on a smaller scale and recreate certain APIs that are provided to us. Relation extractions on the summaries of the movies allowed for creation of tuples of attribute relation triples so as to generate a knowledge graph to allow querying of the dataset. The RandomWalk algorithm provided by the neo4j library provided a medium to extract connections between different movies based on common nodes existing between them and this acted as a form of recommender system wherein related movies or attributes are returned by the API we provided. The product of this project is a system of APIs which allows querying of an extended movie database based on person/movie/date tags which returns entity relationships and extended attribute relations deduced from random walks through the graph generated.
There is certainly room for further improvement with better relation extraction techniques and other algorithms to extract relationships between different nodes. These changes could lead to enhancements with respect to relevance of different relations generated by the queries.