PROJECT REPORT

TITLE- Jester Joke Recommendation System

Team –Jay Sharma(jms140730), Aditya Mahajan(axm156630), Krishna Chaitanya Dodda (krd150230),

1. **INTRODUCTION**

   One of the most popular application of machine learning off late has been the development of recommendations for users of an application. Some of most prominent successful examples of this have been Amazon's recommendations to its users as to what they would buy, Facebook's friend recommendation's and the more recent Netflix Movie recommendation systems. All these have demonstrated the successful application of machine learning algorithms to create reliable recommendation systems. This has also motivated us to develop a recommendation system with application of latest technologies for doing the same. In this project, we aim to develop a joke recommendation algorithm based on jester's dataset of ratings for its jokes. The challenge is to successfully demonstrate building the recommendation algorithm on such a large dataset. The second main objective of our project is to provide a meaning comparison of various frameworks for building joke recommendation systems. We plan to do this by using python's Spark API – pyspark making use of the parallel processing power of the spark framework and also the functional language ease of python. We have also demonstrated the use of neo4j graphical API in developing the recommendation system. We have successfully compared both these frameworks in choosing the best framework and model for our recommender system.

2. **DATASET**

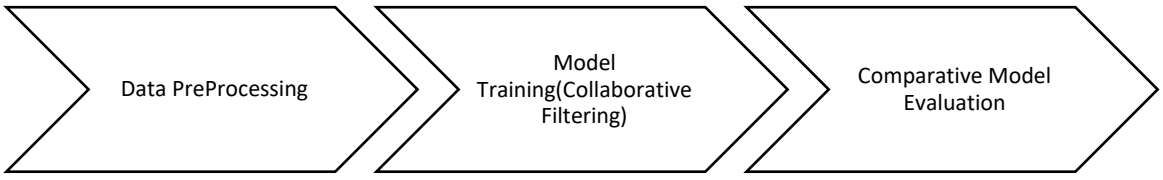
   2.1 Source

   The dataset is provided by provided by UC Berkley and is the anonymous ratings of various jokes provided by the users. (link - http://eigentaste.berkeley.edu/dataset/)

   2.2 Description

   The Jester dataset contains over 4.1 million continuous ratings (-10.00 to +10.00) of 100 jokes from 73,421 users: collected between April 1999 - May 2003. This dataset is provided in three zip files and in .xls format. Each row in the csv file corresponds to each of the users. Hence all the csv files combined we will have 73,421 rows. Each Column is represented by a joke and hence we have 100 columns. Each cell is given a value corresponding to a user and a joke. This value is the rating. Ratings are real values and range from -10 to +10. If, something is not rated, it is assigned a value of 99 (which is a null value). The sub-matrix containing only columns {5, 7, 8, 13, 15, 16, 17, 18, 19, 20} is dense. This means that almost all users have rated these jokes.

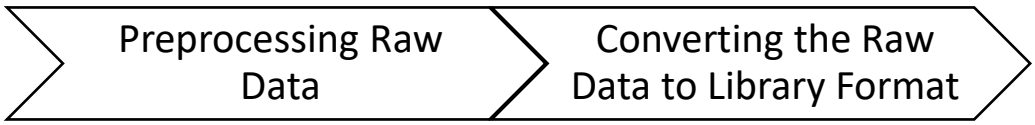
3. **Process Flow**



Data preprocessing is the first stage where the data is converted from its raw form and properly formatted for the libraries. This preprocessing is involves aggregation, modification and filtering operations before being fed to the model. After this we apply the Collaborative filtering Library to the dataset and tune for the best model. We apply this in two frameworks i.e, PySpark and Neo4j and try to draw a comparative analysis on both these models before demonstrating the recommendations.

4. Data Preprocessing

The Data Preprocessing is done in two stages i.e., processing the raw data and then converting this data to the relevant format for the libraries to evaluate.



The preprocessing of the raw data is done in python (DataManipulation.py). The Data provide by the source is in an xls file where each row is represented by a user and a column by the joke. This needed to be converted into a triple format (userid –jokeid –rating) . Also, we have to aggregate three huge xls files provided into a single file. This later part was done using UNIX command bash.

Once the raw data is processed, it is fed to the code. This raw data is read. It is filtered eliminating the null values and then finally presented in the form of a RDD which is our input to the library. Moreover, since ratings vary from a range of -10 to +10, we perform a scaling operation to bring them within the range -1 to +1 so as to improve accuracy of the algorithm. We then perform a random split of the RDD to create train and test RDDs. A snippet of the resultant pyspark RDD is as follows which is then fed to the libraries. We can see from the snapshot below of the resultant data RDD that we have create a triple containing the userid, the jokeid and the rating which is scaled.

```
[(1, 1, -0.782), (1, 2, 0.8789999999999999), (1, 3, -0.966), (1, 4, -0.8160000000000001), (1, 5, -0.752), (1, 6, -0.85), (1, 7, -0.985), (1, 8, 0.417),
```

In the case of Neo4j, The first step in the code loads the processed raw dataset from the csv file line by line. It then avoids line consisting of Rating score of '99' as it means that user has not rated that Joke.

Then it creates a relationship by creating two nodes consisting of user Id and joke Id linked by their Rating. The whole idea is that an edge is created between a user and a joke having a certain value which is the rating. The code snippet is as follows.

```
// Step 1
USING PERIODIC COMMIT 1000
LOAD CSV WITH HEADERS FROM "file:///ratings.csv" AS line WITH line
LIMIT 10000
WHERE line.Rating <> "99"
MERGE (m:User {userId:line.UserID})
MERGE (j:Joke{jokeId:line.JokeID})
CREATE (m)-[r:RATED {Rating:line.Rating}]->(j)
RETURN m,r,j;
```

5. Model training

There are many models to train a recommendation engine. These can be broadly categorized into content based models and context based models. The former type of models are based on individual users prior history of jokes recommended. So the chances are that that individual might prefer liking similar kind of jokes he has already rated. But these approaches don't take into account there might be items which they have not rated and yet could be used to recommend new products to the user. This is where context based recommender systems like Collaborative filtering have become quite successful. Collaborative filtering prepares a similarity matrix. This similarity matrix contains a rating for every joke id for a particular user. The algorithm does this not just based on the history of the current user but also of the history of the other users. For instance, a user might have similar ratings like another users to a set of jokes, so the user can be recommended based on the later ratings as well. In our project we have trained the model using both neo4j and pySpark.

5.1 pySpark model

The pySpark Mllib library we have used is based on the 'Alternating Least Squares matrix factorization' algorithm. This method is called using the 'ALS.train' method call. It trains a matrix factorization model from an RDD of ratings by users for a subset of products. The ratings matrix is approximated as the

product of two lower-rank matrices of a given rank (number of features). To solve for these features, ALS is run iteratively with a configurable level of parallelism. The parameters are rank , iterations , lamda , blocks and seed. 'Blocks' is specified to parallelize the computation hence we set it to the default values so that this is auto configured. It doesn't affect the accuracy or the algorithm computation logic. Similarly, iterations specify how many times the ALS is run over the entire dataset before developing the similarity matrix. The default is 5. Our data set has a nearly 5 million rows and hence we chose to set this value at 10 as a tradeoff between speed of processing and computation accuracy. The remaining parameters that needed to be tuned are the rank and the regularization parameter. Rank specifies the number of features to be computed while regularization parameter helps in reaching a global minimum.

```
Model with rank=2 and lambda=0.001 has rmse=0.470667654392; abs_error=0.37279583505; variance=0.0528300425069; squared_error=0.221528040891
Model with rank=4 and lambda=0.001 has rmse=0.468901682112; abs_error=0.367749681396; variance=0.0606251653617; squared_error=0.219868787488
Model with rank=6 and lambda=0.001 has rmse=0.471069575725; abs_error=0.368546529371; variance=0.0641749308739; squared_error=0.221906545174
Model with rank=8 and lambda=0.001 has rmse=0.474202397882; abs_error=0.370617217466; variance=0.0656321627449; squared_error=0.224867914157
Model with rank=12 and lambda=0.001 has rmse=0.481664409515; abs_error=0.375655363497; variance=0.0722746172142; squared_error=0.232000603393
Model with rank=18 and lambda=0.001 has rmse=0.498404998017; abs_error=0.3868968246; variance=0.0822959448426; squared_error=0.248407542048
Model with rank=2 and lambda=0.005 has rmse=0.468483621792; abs_error=0.370277927139; variance=0.0526347628793; squared_error=0.219476903887
Model with rank=4 and lambda=0.005 has rmse=0.467405976777; abs_error=0.366890770686; variance=0.058407330982; squared_error=0.218468347127
Model with rank=6 and lambda=0.005 has rmse=0.467643480048; abs_error=0.365900669137; variance=0.0621667091299; squared_error=0.218690424432
Model with rank=8 and lambda=0.005 has rmse=0.470611804891; abs_error=0.367298669009; variance=0.0649133082969; squared_error=0.221475470902
Model with rank=12 and lambda=0.005 has rmse=0.476241760085; abs_error=0.371952258123; variance=0.0679343092291; squared_error=0.226806214049
Model with rank=18 and lambda=0.005 has rmse=0.482156685329; abs_error=0.375407541179; variance=0.0742203538972; squared_error=0.232475069207
Model with rank=2 and lambda=0.05 has rmse=0.471230048122; abs_error=0.380507638481; variance=0.0300322032391; squared_error=0.222057758253
Model with rank=4 and lambda=0.05 has rmse=0.469249903828; abs_error=0.378344657056; variance=0.0305668532574; squared_error=0.220195472242
Model with rank=6 and lambda=0.05 has rmse=0.468613362023; abs_error=0.377693572465; variance=0.0305279272689; squared_error=0.219598483066
Model with rank=8 and lambda=0.05 has rmse=0.468011826713; abs_error=0.377098431968; variance=0.0304963824126; squared_error=0.219035069944
Model with rank=12 and lambda=0.05 has rmse=0.467699857996; abs_error=0.376763650651; variance=0.0305320793411; squared_error=0.218743157169
Model with rank=18 and lambda=0.05 has rmse=0.467546540411; abs_error=0.37664950975; variance=0.030525628901; squared_error=0.21859976745
Model with rank=2 and lambda=0.5 has rmse=0.534387271253; abs_error=0.45257614857; variance=0.00551467012178; squared_error=0.285569755677
Model with rank=4 and lambda=0.5 has rmse=0.534387271253; abs_error=0.45257614857; variance=0.00551467012178; squared_error=0.285569755677
Model with rank=6 and lambda=0.5 has rmse=0.534387271253; abs_error=0.45257614857; variance=0.00551467012178; squared_error=0.285569755677
Model with rank=8 and lambda=0.5 has rmse=0.534387271253; abs_error=0.45257614857; variance=0.00551467012178; squared_error=0.285569755677
Model with rank=12 and lambda=0.5 has rmse=0.534387271253; abs_error=0.45257614857; variance=0.00551467012178; squared_error=0.285569755677
Model with rank=18 and lambda=0.5 has rmse=0.534387271253; abs_error=0.45257614857; variance=0.00551467012178; squared_error=0.285569755677
0.467405976777
The best model was trained with rank 4 and lambda=0.005
This model has metrics as follows
root mean square error=0.467405976777
mean absolute error=0.366890770686
```

From the snap shot above, we can see the model being tuned for various values of regularization parameter (lambda) and rank . We have calculated various metrics like rmse, absolute error, variance and squared error. We have chosen the best model based on root mean square error (rmse). After tuning the model and testing on the test set, we have achieved a best model with rank=4 and lamba=0.005.

5.2 Neo4j model

In the neo4j graphical model, we created our own algorithm of constructing a similarity matrix between users to know the users who share similar interests or have given similar rating to the common jokes between them. Using this matrix, we recommend the respective jokes that are applicable for a user as shown in the code snippet below.

```
// Step 2
MATCH (user1:User{userId : '1'})-[r1:RATED]-> (j:Joke)<-[r2:RATED]-(user2:User)
WITH user1.userId AS user1Id, user2.userId AS user2Id, user1, user2,count(j) AS commonJokes,

// Step 3
collect((toInt(r1.Rating)-toInt(r2.Rating))^2) AS ratings,collect(j.jokeId) AS JokeIDs
WITH commonJokes, JokeIDs, user1, user2, ratings
MERGE (user1)-[s:Similarity]->(user2) SET s.similarity = 1-(SQRT(reduce(sum=0.0, k in extract(r in ratings | r/commonJokes) | sum+k))/20);
```

The series of steps (2 and 3) are used to calculate similarity distance for a given user The similarity between two users is calculated in terms of the Euclidean distance. Here the ratings for a user are

thought to be in terms of Cartesian co-ordinates and they exist as vectors in multi-dimensional space. The distance between two such points is calculated using the Pythagorean formula as shown below:

$$\mathrm{d}(\mathbf{p}, \mathbf{q}) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}.$$

Step 2, the snapshot above, performs retrieving all the users who have rated the same jokes as our current user and the total number of such jokes. Step 3 performs calculation of the similarity relationship that is mentioned above. Here we use the Pythagoras theorem to calculate the distance between two user's ratings.

```
// Step 4
MATCH (user1:User)-[r:RATED]->(j:Joke),(user1)<-[s:Similarity]-(user2:User {userId:'1'})
WHERE NOT ((user2)-[:RATED]->(j))
WITH j, r.Rating AS rating,s.similarity as similarity
WHERE similarity > 0.6

// Step 5
WITH j, COLLECT(rating) AS ratingCollection, COLLECT(similarity) AS similarityCollection
```

Step 4, in the snapshot above, finds all the users who have similarity greater than 0.6 with the current user. Next, Step 5 finds all the jokes that our current user has not rated but other similar users have rated it, also collects the ratings and similarity for that jokes. Finally Step 6 as shown below computes average of the similarities and the ratings by using data of the similar users who have rated it to complete the construction of our similarity matrix which can be used for recommendations.

```
// Step 6
WITH REDUCE(num = 0, s IN similarityCollection | toInt(num)+toInt(s))*1.0 / LENGTH(similarityCollection) AS avgSimilarity,
j,similarityCollection, REDUCE(num = 0, s IN ratingCollection | toInt(num)+toInt(s))*1.0 / LENGTH(similarityCollection) AS avgRating, ratingCollection
```

This completes the model building process using Neo4j.

The main challenge we encountered while working with Neo4j was to upload a very large dataset into neo4j database. Neo4j Client took forever to load the entire dataset on our machine configuration. It does require some latest hardware like SSD's and high capacity of RAM in order to load the entire dataset and generate nodes for the users and jokes and creates relationship between them. We also tried some performance tuning that is mentioned below but it marginally improved the performance of the entire process. For demo purpose, we are limiting our dataset to 50,000 records and we provide joke recommendations to a particular user from it.

5.2.1 Performance Tuning:

We have tuned certain parameters in order to maximize the performance. They are listed as follows:

1) Page Cache Sizing:

- Page Cache sizing provides caching of the Neo4j data. This avoids costly access to the disk. Thus it makes sure that the performance is not degraded.
- This can be implemented by setting the "dbms.pagecache.memory" property in neo4j.conf file.

2) Increase heap size:

- The heap size is also increased from its default value to sustain multiple operations simultaneously.
- This does provide a significant performance boost for Neo4j.

- The can be implemented by setting dbms.memory.heap.initial_size and dbms.memory.heap.max_size in neo4j.conf.

6. Analysis

6.1 Comparative model Evaluation

Any model evaluation would require certain metrics for it to be evaluated. In this case, the rating prediction of an unknown class variable is a real number. Hence, what we have here is a regression problem. In our case, we have evaluated it based on three parameters i.e, Root Mean Square error (rmse), absolute error , variance and squared error. Rmse is the root value of the squared error and hence they can be considered identical. Hence, rmse and variance are the two primary evaluators.

Model	Root Mean Squared Error	Variance	Mean Absolute error	Squared Error
Alternating Least Squares	0.467405976777	0.058407330982	0.366890770686	0.218468347127

We can see from the snapshot that our model was achieved for a low variance and relatively low rmse .Ideally, rmse of less than 0.2 would have been good. But, given the large dataset of nearly 4 million users for 100 jokes, this rmse value is acceptable. Also, the model is low on variance and deviation with a value of less than 0.1. The mean absolute error (MAE) is a quantity used to measure how close forecasts or predictions are to the eventual outcomes. It measures how far a predicted value is away from the mean value. Compared to the similar Mean Absolute Error, RMSE amplifies and severely punishes large errors. Hence, outliers are more severely punished in the case of RMSE than in the case of mean absolute error. Therefore, rmse is often taken as a true evaluator as compared to mean absolute error and that has been the case in our model as well. The neo4j recommender recommends using the following code snippet below (Step 7).

```
// Step 7
ORDER BY LENGTH(ratingCollection) DESC, avgRating DESC, avgSimilarity DESC
WHERE avgRating >2
RETURN j AS Joke, avgRating AS Recommendation
```

In this step, it orders in descending manner using the result by length of ratings, average ratings and average similarities and returns the recommended joke and its ratings if average ratings > 2. We thought 2 was a reasonable threshold to choose from since our ratings range from -10 to +10 and these jokes are rated fairly above average.

6.1.1 Advantages of using Neo4j:

- Neo4j uses Cypher queries for graph traversal and manipulation that is easier to learn, understand and write. It also provides faster traversing over the graph.
- Provides different graph representation like Visual graph representation, row representation and text representation.
- There are many different ways we can perform operations on graph using the Neo4j library. It can be used in different programming languages like Java and Python; It can also be used in command shells like Gremlin shell and neo4j-shell. It can also be used on their own Neo4j Client.
- Visual representation of the graph provides better understanding of the relationships between the nodes.

6.1.2 Disadvantages of using Neo4j:

- Loading large dataset requires having a machine with better hardware.
- It does not support Sharding.

6.2 Results

The output for recommender system using mllib library is displayed below:

The screenshot below shows array of every joke Id with 5 user Ids each. This output displays recommendation of a joke to the top 5 users who have not heard the joke and will most probably like the joke.

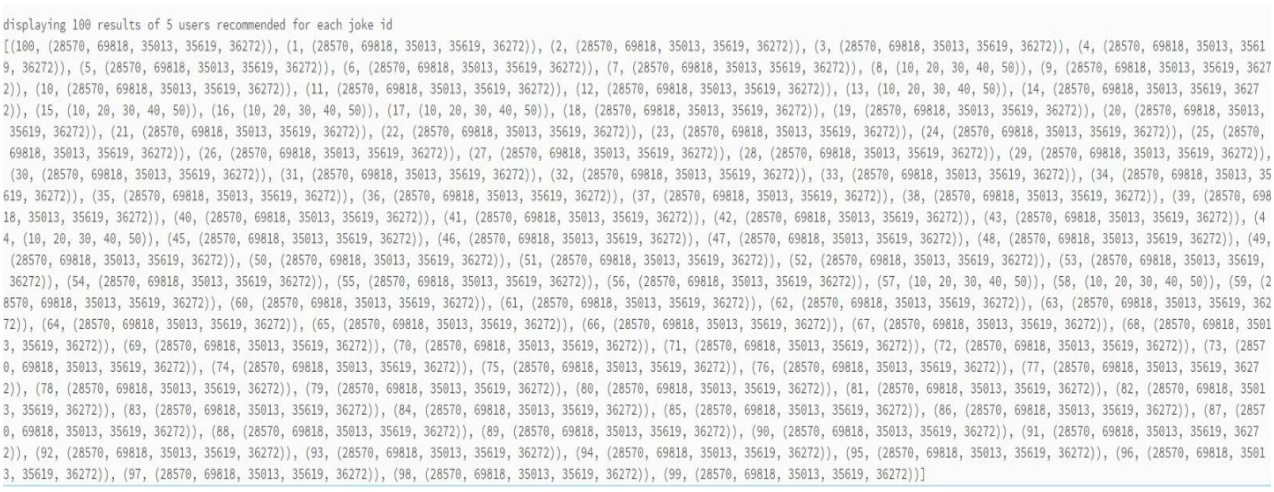


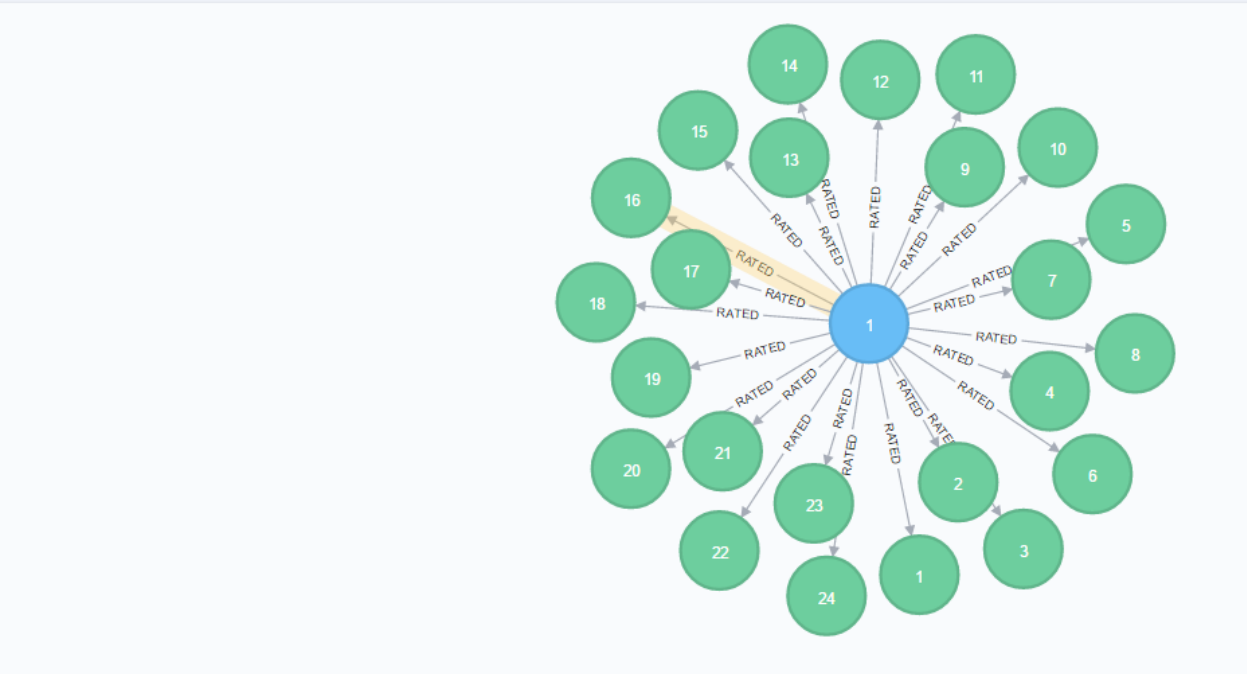
Fig: List of 100 Jokes each containing recommendation to top 5 users

The screenshot below consists of array of every user Id with 5 joke Ids each. This output displays recommendation of top 5 jokes to every user that they have not heard of and will most probably like.

Fig: List of 100 Users each containing recommendation to top 5 jokes

The output for recommender system using neo4j library is as follows:

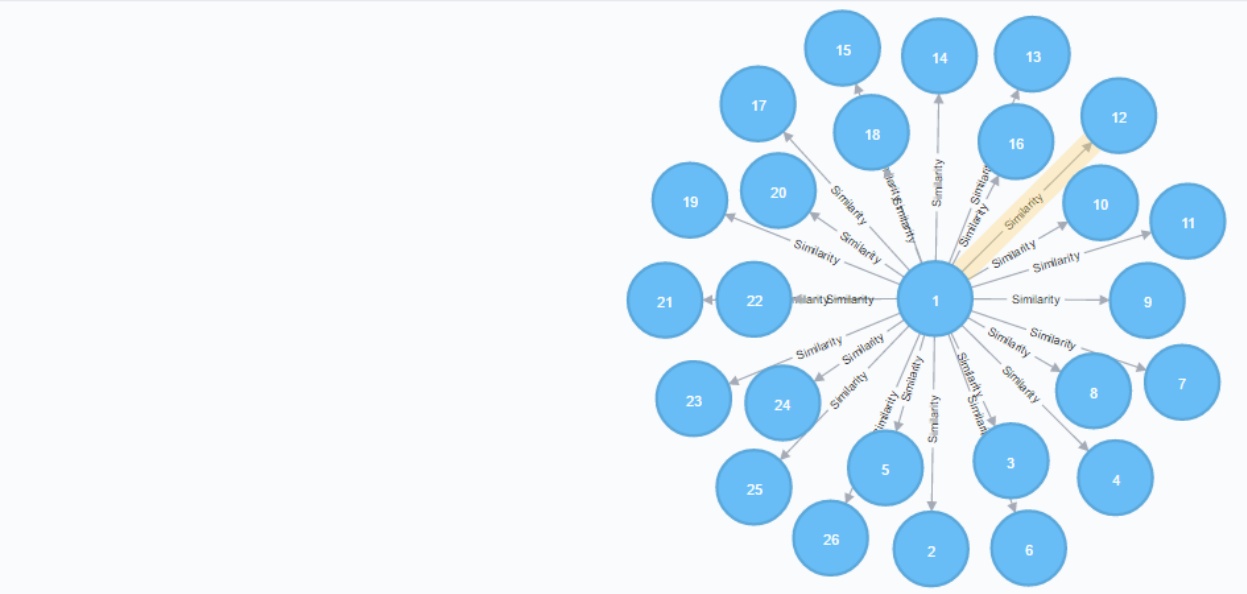
The screenshot below displays a graph containing user and joke relationship on ratings for a user with Id 1.



RATED **<id>:** 15 **Rating:** -7.52

Fig: Graph for user with Id 1 consisting of ratings relationship with some jokes.

The next screenshot contains relationship of a user with Id 1 to other similar users using the calculated similarity index.



Similarity **<id>:** 7403 **similarity:** 0.6655985581306141

Fig: Similarity relationships for the current user having ID 1 with other similar users

The last screenshot shows the recommendation that is provided to the user with id 1 some jokes that they will most probably like.

Joke	Recommendation
{jokeId: 93}	2.28
{jokeId: 89}	3.272727272727273

Fig: Recommended Jokes with their IDs and their average ratings.

7. **Conclusion**

   The ultimate aim of the project after building our model is to generate Joke recommendations for a user and we have successfully achieved this by implementing our recommender system using two different libraries i.e. mllib and neo4j and gaining knowledge and understanding in this arena. We are very thankful to Professor Anurag Nagar for providing us an opportunity and also the freedom to test our knowledge and skills on such an interesting project.

8. **Future Work**

   In the future we plan to develop an online recommendation service that provides a visual interface to the users. We plan to provide a feedback system that will display some jokes to the user and they will rate it based on their likeability. Once we have got some ratings for a particular user we will use our existing recommender system to recommend some jokes that they will most probably like. This will provide an interactive system for users to interact with our recommender system.

9. **Contribution of Team Members**

   We are a group of three – Jay , Krishna and Aditya . The project provided us a great opportunity to test our skills at a both technical and not technical level. At a non-technical level, it helped us in efficient time management and perform as a team and not as a group of isolated individuals. We formed whatsapp groups and scheduled meetups at the Library as well as the UTD open lab. At a technical level , we congregated our efforts and technical skills in a properly coordinated fashion. We brainstormed which libraries and coding language and also the appropriate way to approach the project. Jay's knowledge in Neo4j, Krishna's experience in working in Big data Platforms in Enterprises and Aditya's knowledge in python helped in efficient planning and completion of project. We sat together and discussed the algorithm, Aditya helped with preprocessing of data for the pyspark , Krishna tuned the model and analyzed the results and Jay sat through the Neo4j code. Overall we had an equal contribution in the entire project and also the report.

10. **References**

1. Eigentaste: A Constant Time Collaborative Filtering Algorithm. Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Information Retrieval, 4(2), 133-151. July 2001
2. http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.recommendation.ALS
3. Neo4j Performance Tuning - https://neo4j.com/developer/guide-performance-tuning/
4. Beer Recommender System - http://mikelam.azurewebsites.net/beer-recommendations-with-user-based-collaborative-filtering/
5. Course Lecture slides