Friend Recommendation System
Collaborative Filtering

Sidharth1 and **Steinunn Sigurjonsdottir**2
1email: sidharth@ksu.edu
2email: steinunns@ksu.edu

1 Overview problem statement

The task we were faced with was to build a collaborative filtering recommendation system that suggests friends to users in a social network like Facebook. The recommendation system will assign each potential friend a score based on mutual friends, where higher score means a better match. The system will then sort the scores and produce friend recommendations in order from best to worst.

2 Background and related work

Before we started coding, we read about recommendation systems in books and articles online to get more familiar with the topic. We got most of our knowledge from Anand D. Rajarman's book, "*Mining of Massive Datasets"* [4]. There is a specific chapter in the book dedicated to recommendation systems. It helped us understand what lies behind building such systems. According to Rajaraman, recommendation systems can be classified into two broad groups; content-based systems, and collaborative filtering systems.

- Content-based systems look into the properties of the items recommended. Netflix is a good example of a content-based system. If a user has watched many sci-fi movies, then the system will recommend a movie in the database classified in the sci-fi genre.
- Collaborative filtering on the other hand recommend items based on similarity measures between users and items. In other words, the system identifies similar users and recommends what similar users like [4].

We got our original data and task outline from The University of Washington's computer science website. Later on, we found a bigger dataset on Stanford's SNAP website [5]. On the University of Washington's website, we saw how to build a collaborative recommendation system by using graphs. The graphs plot the relationships between users in the network and determine friend suggestions by counting mutual nodes and using that as its similarity measure [7]. However, we decided to go with the method introduced in Rajaraman's book on collaborative filtering systems.

According to Rajaraman, concepts like utility matrices and distance measures are needed to build a collaborative filtering system. Utility matrices deal with users and items. They offer known information about the degree to which a user likes an item. Normally, most entries in a utility matrix are unknown, and the ultimate goal of recommendation systems is to predict the values of the unknown entries based on the known entries. In collaborative filtering, the system must be able to measure the similarity of rows and/or columns of the utility matrix. That is where the distance measure comes in [4].

Distance measures are needed to be able to make a recommendation. Users are similar if their vectors are close according to some distance. Where the nearest points are the most similar and the farthest points are the least relevant. We decided to use the Jaccard Similarity Index as our distance measure.

- Jaccard Similarity Index compares the similarity of two sets by calculating the ratio of the size of their intersection to the size of their union [4]. It is a measure of similarity with a range of 0-100%. High percentage indicates high similarity while low percentage indicates a low similarity [6].

When it came to build our model we used Apache Spark's article on Collaborative Filtering as our guideline. The model was trained by using mllib prebuilt libraries. The data's rows consisted of a user, a product and a rating. In our case each row consists of a user, a friend, and Jaccard similarity score as the rating. We used the prebuilt ALS.train() method which is a part of the mllib package [1].

- ALS stands for Alternating Least Squares matrix factorization. The method:

  [trains] a matrix factorization model given 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 [3].

The simple code for building the model is:

1. Declare the values for rank, and number of iterations
   a. Rank is the number of features to use
   b. Iterations is the number of iterations for ALS to run. ALS usually converges to a reasonable solution in 20 iterations or less

2. Construct the model:
   Model = ALS.train(rank, numIter, trainData)

3. Make predictions
   Predictions = model.predictAll(testData)

The recommendations (predictions) are evaluated by measuring the Mean Squared Error (MSE). MSE is a measure of the quality of an estimator, in our model the estimator is the Jaccard similarity score [2].

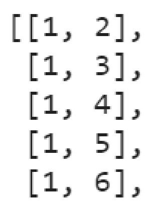
3 Methodology

We wrote our code in PySpark using Jupyter Notebook and Google Collaboratory. We used Tableau for our visualization.

The data we used to build our model with consisted of over 1.7 million connections between nodes (number of rows), and over 64 thousand unique nodes. Each line represented a user-friend relationship which consists of 3 columns. The first two columns contain anonymized user identifiers. The first one is the userID, and the second one is the friendID. Finally, the third column is a timestamp with the time of the user-friend friendship establishment. The average number of friends per user is 25.7 friends.

We also used a bigger dataset to run through our model in order to be able to compare the performance of our mode. The bigger dataset consisted of 30.6 million connections between nodes, and 1.6 million unique nodes. The bigger dataset was of the same format as the smaller one.

First, we loaded the needed packages into Jupyter Notebook/Google Collaboratory. Secondly, we removed the timestamp column as it is irrelevant to the process of recommending friends, and then converted the data into an RDD. Figure 1 shows an example of the first 5 entries of the RDD: [userID, friendIDs]



```
[[1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [1, 6],
```

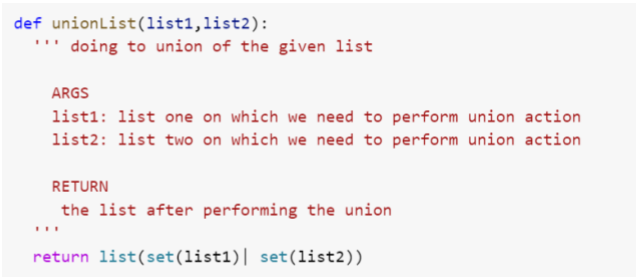
Figure 1

We noticed an error in the RDD. Some IDs only appeared as friendIDs (neighbor nodes) but not as userIDs (main node). The first step in fixing the problem was to make a second RDD that swapped the columns of the original RDD. Now the friendIDs became userIDs and vice versa. The second step was to create matrices for both RDDs by using the groupByKey() function. The first column in the matrix was the userID, the second column was a list of each users' friendIDs. Third, we created a dictionary for the original RDD where each userID is a key/node and all the user's friends are the values/neighbors. Fourth, we created a list out of the reversed RDD. Finally, we filtered all the values in the reversed list that did not appear in the original RDD and added them as UserIDs (main nodes) to the original RDD. Now we had an RDD that included everyone in the dataset.

We created a dictionary that would function as our utility matrix. We created a dictionary from the good RDD. The keys in the dictionary were the userIDs (main nodes), and the values were the users' friends (neighbors).

When the dictionary was complete, we used it to calculate the Jaccard similarity score. As stated above the Jaccard similarity score between two sets, in this case two users, is the ratio between the size of their intersection (mutual friends) to the size of their union (total friends). We created 4 functions, shown in figures 2-5, to calculate and apply the similarity score.

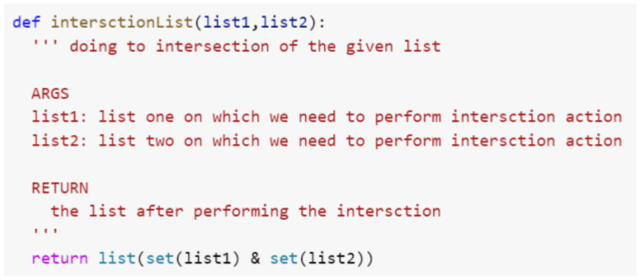
First a function (unionList) to get the union between two users:



```
def unionList(list1,list2):
  ''' doing to union of the given list

    ARGS
    list1: list one on which we need to perform union action
    list2: list two on which we need to perform union action

    RETURN
     the list after performing the union
  '''
  return list(set(list1)| set(list2))
```

Figure 2

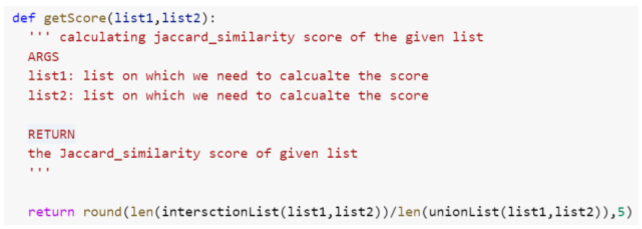
Secondly a function (intersectionList) to calculate the intersection between two users:



```
def intersctionList(list1,list2):
  ''' doing to intersection of the given list

  ARGS
  list1: list one on which we need to perform intersction action
  list2: list two on which we need to perform intersction action

  RETURN
    the list after performing the intersction
  '''
  return list(set(list1) & set(list2))
```

Figure 3

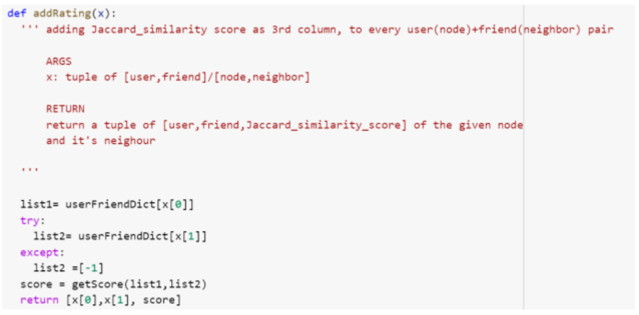
Third, a function (getScore) that calculated the Jaccard similarity score between two users:



```
def getScore(list1,list2):
  ''' calculating jaccard_similarity score of the given list
  ARGS
  list1: list on which we need to calcualte the score
  list2: list on which we need to calcualte the score

  RETURN
  the Jaccard_similarity score of given list
  '''

  return round(len(intersctionList(list1,list2))/len(unionList(list1,list2)),5)
```

Figure 4

Finally, a function (addRating) that adds the similarity score to every user-neighbor pair in the dataset

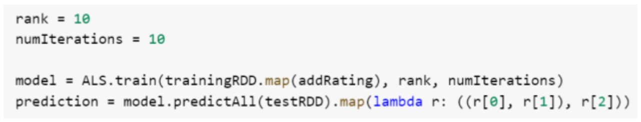


```
def addRating(x):
  ''' adding Jaccard_similarity score as 3rd column, to every user(node)+friend(neighbor) pair

      ARGS
      x: tuple of [user,friend]/[node,neighbor]

      RETURN
      return a tuple of [user,friend,Jaccard_similarity_score] of the given node
      and it's neighour

  '''

  list1= userFriendDict[x[0]]
  try:
    list2= userFriendDict[x[1]]
  except:
    list2 =[-1]
  score = getScore(list1,list2)
  return [x[0],x[1], score]
```

Figure 5

When the functions were complete, we randomly split the data into 80% training data and 20% testing data. Then we created a new RDD using training data that included 3 columns; userID, friendID, and similarity score. Now we were ready to start building our model.

We used prebuilt machine learning libraries to build and train our model see figure 6. Like stated above we used the ALS.train() to train our model. The method trains a matrix factorization model given an RDD of ratings by users for a subset of products. Our ratings were the similarity scores, and the products were users' friends.



```
rank = 10
numIterations = 10

model = ALS.train(trainingRDD.map(addRating), rank, numIterations)
prediction = model.predictAll(testRDD).map(lambda r: ((r[0], r[1]), r[2]))
```

Figure 6

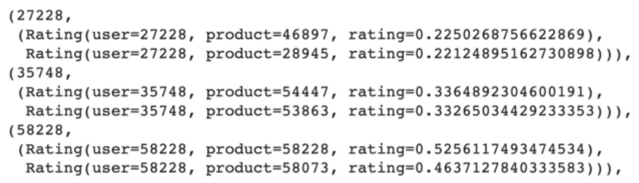
To evaluate the accuracy of our model we used mean squared error. To evaluate the model's performance, we compared the execution times using different number of iterations.

4 Evaluation Criteria

We used couple of measures to evaluate our model's accuracy and performance. We used Mean Squared Error (MSE) to evaluate the accuracy by calculating the mean squared deviation of the Jaccard similarity score. We also recorded the different values of MSE as we changed the number of iterations. To measure the model's performance, we compared the execution times when using different number of iterations and also by using different systems; Google Collaboratory and Jupyter Notebook on our own personal laptop (MacBook Pro 2017). See figures 8-12.

5 Results

We were successfully able to build a collaborative filtering system that recommends friends to social network users by using the Jaccard similarity index. Figure 7 shows an example of what our model can do. By using model.recommendProductsForUsers(2), the model will return the 2 best friend recommendations for each user. The two best suggestions for user 27228 are users 46897, and 28945.



```
(27228,
 (Rating(user=27228, product=46897, rating=0.2250268756622869),
  Rating(user=27228, product=28945, rating=0.22124895162730898))),
(35748,
 (Rating(user=35748, product=54447, rating=0.3364892304600191),
  Rating(user=35748, product=53863, rating=0.33265034429233353))),
(58228,
 (Rating(user=58228, product=58228, rating=0.5256117493474534),
  Rating(user=58228, product=58073, rating=0.4637127840333583))),
```

Figure 7

The results from our evaluations measures are shown in the next few figures. First, we recorded the execution times when using different number of iterations and compared them. As expected, the execution time goes up as the number of iterations increases.

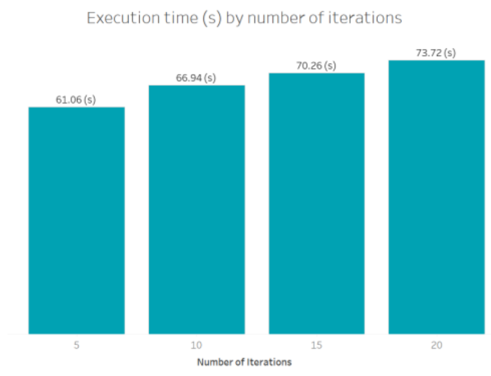
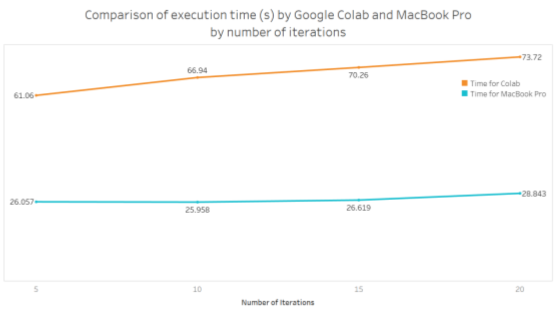


Figure 8

Secondly, we recorded the difference in execution time when running the model in Jupyter Notebook



and Google Collaboratory. Our personal computer was much faster than Google Collaboratory. See figure 9.

Figure 9

Finally, we recorded the difference in execution time when using the bigger and smaller datasets. See figure 10.

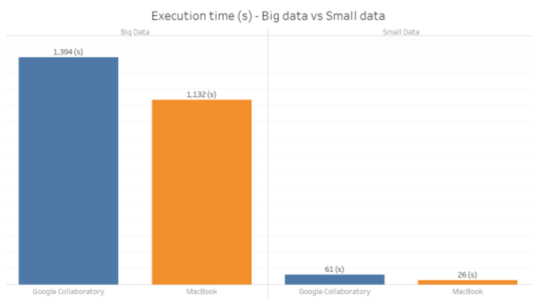


Figure 10

We were interested to see if number of iterations would also affect the MSE like it did with the execution time. We can see in figure 11, the MSE decreases as the number of iterations increase.

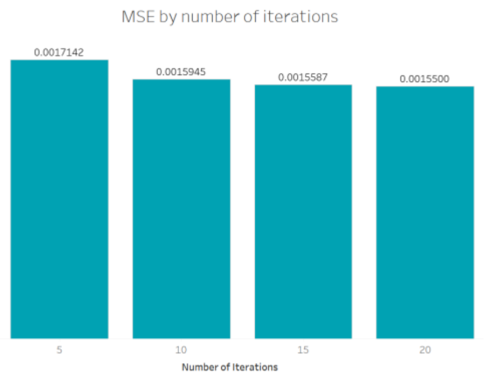


Figure 11

We also compared the MSE in the bigger dataset against the MSE in the smaller dataset. See figure 12. The MSE in the bigger dataset was slightly smaller than in the smaller dataset.

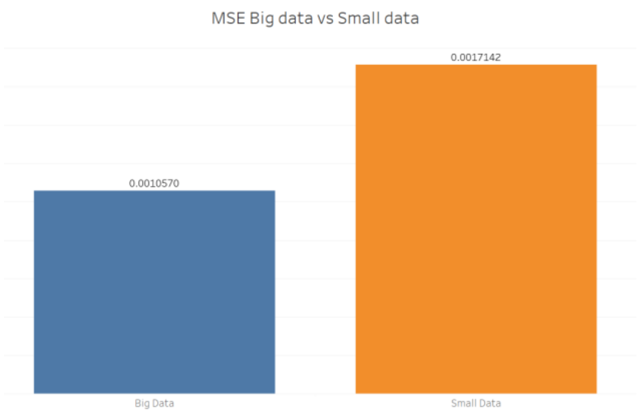


Figure 12

We think our MSE is acceptable. You want the MSE to be low and close to 0, but you don't want it to be exactly 0 because then your predictions are 100% correct which is highly unlikely.

At the beginning of this process we wrote a project proposal. A part of that proposal was to set milestones to help us organize our work to be able to finish the project on time. There were 8 weeks until the due date at the time we wrote set the milestones. Our first milestone was to spend the first two weeks reading and learning what there is to know about collaborative filtering systems and then come up with a plan on how we want to construct our model. Our second milestone was to build the program during weeks 3-7. Our last milestone was to spend the last week on the final report and presentation, as well as cleaning up the code if we needed to. We were able to stick to our milestones and complete all our goals on time.

We learned a lot in this process. Before we started, we did not know much about recommendation systems, but now we know the main concepts that lie behind them. Content-based systems look into the products' properties while collaborative-filtering systems identify similar users and recommend what similar users like. What we learned regarding big data was that you have to handle it with care. You need to know your data and figure out ways to get the most out of it. If you don't handle your data with care your system might crash like we experienced couple of times in this process.

The most technically challenging part of this project was the data handling. It took us great deal of time to figure out how to get the data into the right form to be able to use it to build our model.

6 Summary & Future Work

The problem we were faced with was to build a collaborative recommendation system that is able to suggest friends to users of a social network. We were successfully able to build a model that takes users' mutual friends to account and ranks possible friends by the similarity score. We used the Jaccard similarity index as our similarity measure. We used the ALS prebuilt package from the mllb library to build our model. We evaluated our model by calculating the mean squared error and recording the execution times while using different number of iterations and running them on Google Collaboratory and our personal computer. We also compared the performance of our model when using a bigger dataset. The results showed the more iterations, the lower MSE and higher execution time. The bigger data hat a lower MSE score than the smaller data but the execution time was much longer than of the smaller dataset.

We were able to uphold all our preset milestones and complete the project on time and we learned a lot in this process, both about recommendation systems and working with big data.

Currently, we do not plan on continuing with this project, however if we were, we would like to find a way to implement it on an actual social network on the internet. Right now, our model only lives in a notebook, but it would be interesting to see it in live action. Even though we do not plan on continuing with the project we are positive that we will use recommendation systems in the future.

References

[1] Collaborative Filtering - RDD-based API. (n.d.). Retrieved November 15, 2019, from https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html.

[2] freeCodeCamp.org. (2018, October 8). Machine learning: an introduction to mean squared error and regression lines. Retrieved December 4, 2019, from https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/.

[3] pyspark.mllib package¶. (n.d.). Retrieved November 15, 2019, from https://spark.apache.org/docs/latest/api/python/pyspark.mllib.html#pyspark.mllib.recommendation.ALS.

[4] Rajaraman, A., Ullman, J. D., & Leskovec, J. (2014). *Mining of massive datasets*.

[5] Stanford SNAP. (n.d.). Pokec social network. Retrieved December 1, 2019, from https://snap.stanford.edu/data/soc-Pokec.html.

[6] Stephanie. (2019, August 26). Jaccard Index / Similarity Coefficient. Retrieved November 5, 2019, from https://www.statisticshowto.datasciencecentral.com/jaccard-index/.

[7] Washington University. (n.d.). Social networking and recommendation systems. Retrieved October 30, 2019, from https://courses.cs.washington.edu/courses/cse140/13wi/homework/hw4/homework4.html#Recommendation_systems.