Project 2

Big Data

Name: Theodoros Mandilaras
A.M.: cs2.190018
MSc DIT/EKPA | Spring 2019-2020.

The project implemented with **Python 3.7**. The given zip file should contain two folders.

- Disk_based_kmeans
- Disk_based_collaborative_filtering

Each folder contains my implementations of the two parts of the project. Also, both of them contains a README.md file with details about execution.

Part A: Disk Based KMeans

Files

The Disk_based_kmeans folder should contain those files:

- __init__.py: This file is empty, and it exists only to make python handle the directory as a package.
- Kmeans.py: It contains the main algorithm, and it is the execution file.
- Cluster.py: Contains the classes that describes the cluster.
- Create_file.py: As we will see, I create a **new data file** for handling the tag related distances. This is the python program that does that job. On README, I have written usage instructions.
- Utils.py: This file contains some useful functions just like Jaccard similarity, a class called **MovieRatings**, and a data structure I use in d3 and in d4.
- README.md: Contains information about the execution.

Basic Algorithm

The basic algorithm is kinda similar to the **BFR** algorithm. The program reads the data file chunk by chunk. In every chunk, it calculates the desired distances for each point with all the existing clusters (using the clusteroir). Then, each point is added to the cluster with the maximum similarity only if the similarity is larger than a threshold. Otherwise, the point is added into a list called remainings. This functionality is executed in the fit() method. According to the desired distance the *fit* method is different.

On the first loop, in the first chunk, the program picks randomly the first K points which will be used as the first clusters. A cluster is described with a class (called either **SimpleCluster** for d1,d2,d3 or **ComplexCluster** for d4). This class contains the information about the current clusteroid, the key of the cluster (its name), a list called membership which contains all the movieIds that belongs in the cluster, and a list called temp which contains the points that are added in the current chunk before they get thrown away.

When a point is added on a cluster, firstly is added in the temp list. When the parsing of the current chunk has completed, the clusters that got new points should recalculate their clusteroid. That happens in the consume() function which is member of the cluster class. The sum of all the distances of each new point with every other new point in the cluster and with the clusteroid is calculated. Similarly, for the clusteroid with every new point int the temp list. The point with the maximum sum (max similarity) is set as the new clusteroid of that cluster. Then, all the movieIds from the new points in the temp list are registered in the membership list, and the temp list gets empty. So in each loop the used points are thrown away.

All the points that didn't achieve to be inserted into a cluster, they are inserted at the remaining list. All those points are described by a class called **RemainEntity**. This class is just like a mini-cluster. Its members are a clusteroid, and a list with members. The point of that class was to create compressed entities that will contain more than one point as a single object in the remaining list. So, I in the end of parsing the chunk, I was doing a hierarchical cluster that I implemented only for those remaining points. This hierarchical cluster was used to compress the RemainEntities and to throw away unnecessary information.

This hierarchical cluster, in every loop it calculates all the distances for each point with every other point (of the remaining list). After that, it merges only the pair of points with the maximum similarity if it is larger than the basic threshold. That loop, is repeated forever until no more merges are happening. However, that function is no more used, cause the algorithm was slowing down the whole process so much, making it not worth keeping. So, the RemainingEntities are not compressed and each one contains one point. The code of the hierarchical cluster still exists in the **utils** file.

When the fit() method is completed, all the chucks of the data file have been parsed. Then, the absorb() method starts. In this function, all the points inside the remaining list are added to their closest cluster which now they are at their final form.

Output

Now, the clustering process has been completed. The cluster details are printed in the screen (key, clusteroid and the amount of the members) by the details() method. The results are either **print** or if the **--export** flag was given as an argument, the results will be stored in the working directory by the export() method in a CSV file named: <distance>_results.csv. This file contains two columns, the movieId and the ClusterKey in which the movie belongs. The movieIds are sorted just like the movies CSV file.

Implementation

The main functionality that was described above, belongs in a class called **KMeans**. In the constructor of this class, basic variables are set. Those variables are the path of the CSV data file, the given number of clusters (k), the desired distance (d<1-4>) and some other optional parameters (*threshold, chunk_size, ratings_path*). All of them are given as arguments in the execution of the program. *More details about the execution and the arguments are in the* ***README.md*** *file.*

According to the desired distance function, a different fit() method is executed.

Distance 1: Genres and Jaccard similarity

For the first distance, I use as a clusteroid the genres of the point which has been selected as a clusteroid in the cluster. The Jaccard similarity is calculated by splitting the genres and converting them into a set. After that I divide the intersection over the union of the sets. It can be found in the utils file.

In this distance the fit() method that is executed is: fit_with_new()

Distance 2: Tags and Jaccard similarity

Because the tags are located in a different file, my first approach was to collect all the tags for each movie in the current chunk, for every chunk. However, that approach was very expensive, so I decided to create a **new data-file** that gathers for every movie all the tags. This new file is similar with the movies.csv. It contains three columns, the moviesId, the tags and the genres. The tags are lowered (in order to minimize the different one) and concatenated with the "|" character as a separator into a large string, similar with the genres. In this way I can reuse the same functionality just like in the **Distance 1.**

This **new data file** can be created with the create_file.py program, and it only takes a few seconds to be complete. More details are in the **README.md** file.

The Jaccard similarity is used similarly by splitting the string with the concatenated tags into the tags, and comparing the intersection over the union of those sets with tags. Also, the clusteroid of each cluster is this string with the concatenated tags. However, because of the large variety of different tags, it is very hard to get high similarity. So, it is hardly recommended in the tag related distances, low thresholds to be used.

Also, in this distance the fit() method that is executed is: fit_with_new()

Distance 3: Ratings and Cosine similarity

Just like in the distance 2 my first approach on handling this distance was to parse the ratings CSV file in every chunk and creating the rating vector for each movie in the chunk. However, that was even more expensive than parsing the tags file, because the ratings file is way larger. Moreover, in this case, I could not append in the previous file the ratings vectors for each movie, because the file would get extremely large since each vector will had length of 162541 which is the number of the different users and it would be completely sparse. I tried to shrink that vector using SVD with about 400 components and store it in the **new data file**, but then the time amount for creating this file was getting increased dramatically, and the approach was not well representative. So, I decided to create a data structure located inside a class called **MovieRatings.**

MovieRatings

This data structure is used in order to collect the ratings values wisely and to generate the vectors dynamically with respect on the memory. It is created in the constructor of the class by parsing with a CSV reader all the lines of the ratings file and building the structure line by line. This class is used in the rating related distances (d3 and d4).

This structure is a python dictionary which has as **keys** the movies ids, and as **values** it has another dictionary, in which the **keys** are the user Id and the **value** is his rating for current movie.

{ movie_id: { user_id: rating } }

So, it could look like this:

```
{
        . . . ,
         3: { 230: 1, 235: 3.5, 331: 5},
        4: { 230: 2, 331: 1.5, 441: 1},
        . . .
}
```

Which shows that in the movie with id 3 the users with id 230, 235 and 331 rated with 1, 3.5 and 5. Similarly, for the movie with id 4.

Now in order to get the vectors of all the movies in the chunk I get the vector for each movie individually. Each vector is created with the method get_vector(movie_id). This method creates a numpy array with shape (162541,1) filled with zeros. Then, with the help of the above mentioned structure, it fills in the correct position the rating of each user, and in the end, it converts it as a sparse matrix (scipy.sparse.csr_matrix). I tried different approaches like creating for each chunk a multidimensional vector (instead of creating many vectors, to create one huge vector) which it had in each row the corresponding movie vector. However, the first one was the most efficient, so I stuck with that.

The movie structure is created in about 30 seconds and all the vectors in a chunk needs about 12 seconds get created (with chunk size of 5000).

In the Distance 3 the **KMeans** class fits with the method fit_with_ratings(). The algorithm is similar with the previous distances except that now the clusteroids of the clusters has as content the sparse vector of the point which set as clusteroid, and for measuring the similarity (distance) I use the cosine_similarity from sklearn.metrics.pairwise because it makes a good use of the sparse matrices. It also creates the **MovieRatings** object at the beginning of the fitting. Once again, it is suggested to use low threshold values because the vectors are very long and not so similar.

Distance 4: 0.3*d1 + 0.25*d2 + 0.45*d3

The Main difference in this approach is the cluster class I used to describe the clusters. In the previous distances I use of a class named **SimpleCluster**. It contains the value of the clusteroid (either string with genres or tags or the vector with the ratings), the temp list and the memberships list. In the Distance 4 because one clusteroid is not enough for the complexity of the similarity measurement, I created another similar class called **ComplexCluster.** The only difference is that, instead of using only one member for clusteroid, it has three members to describe the clusteroid (clusteroid_genres, clusteroid_tags, clusteroid_ratings). Each one is used to calculate the different distances between points.

In the Distance 4 the **KMeans** class fits with the method fit_with_all(). It requires the **new data file** that we have created to the distance 2 and also it creates the **MovieRatings** class. In each chunk, for every record, it gets the genres string, the concatenated tags string and the rating vector. It calculates the similarity with every existing cluster by calculating the Jaccard similarity for the genres and tags, and the Cosine similarity for the ratings vectors. The distances are summed with the given weights, and this is how the similarity is measured. One more difference exists in the method that prints the details, which now has to print more information about the clusteroids. Everything else follows the similar logic.

Evaluation

All the tests implemented on a Computer with the below specs:

- CPU: Intel Core i5-6500 @ 3.20GHz × 4
- RAM: DDR4 16GB
- Disk: SSD 256GB

The given arguments where:

- Number of clusters (K): 15
- Threshold: 0.4

- Chunk-Size: 10.000

The durations of the evaluations calculated using the: time.time() from time module.

Creation of the new data file: 6.4 seconds
Creation of the **MovieRatings** object: 33.641 seconds

Distance	Total Durations (seconds)
Distance 1	65.389
Distance 2	99.203
Distance 3	2150.94 (t: 0.2, chunk-size: 5000)
Distance 4	13609.0 (t: 0.2, chunk-size: 5000)

The files with the results will be included in the delivered zip file.

Part B: Disk Based Collaborative Filtering

Files

The Disk_based_collaborative_filtering folder should contain those files:

- collab_filtering.py: This file contains all the functionalities for that part of the project.
- README.md: Contains information about the execution.

Basic Algorithm

The main program contains a class called **CollaborativeFiltering**. This class is responsible for making the predictions. The required arguments that it needs is the locations of the ratings CSV file and which method to use for calculating the predictions (user-based, item-based, mix). All those inputs are given as arguments (more details in the **README.md**). According to the desired method the constructor sets the correct predict() method. Also, the constructor parses the ratings file twice.

The first parse is done by a method called collect_movies_and_users(). The goal of that class is to gather the users ids and the movies ids in two separate lists. By this way, we can know how many movies have been rated, and it works also as a linker between the position of the movie and the movie id since the movieIds are not in sequence. So using this movies list I can get the id of a movie by its position which is a very helpful feature as we are going to see later. Lastly, we get the number of the users that has rated movies (For user we don't want that link feature, because users are sorted, and they are in sequence in the dataset).

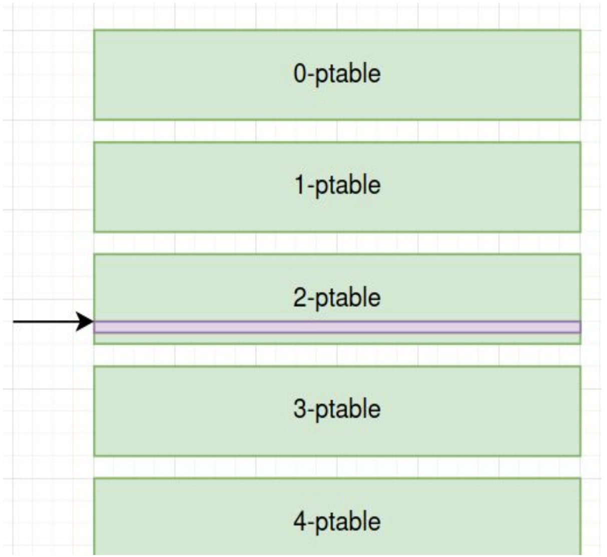
That process takes about **6 seconds**.

Multiple Pivot Tables

In the second parse the constructor creates the **pivot tables** (or utility matrices) in the create_pivot_tables() method. Each pivot table has as rows the users (row = user_id - 1 since the users in the ratings CSV are increased one at a time starting from the 1), as columns the movies and as values the ratings that the user did for a specific movie. It is a **csr**(=Compressed Sparse Row) sparse matrix from the **scipy.sparse** module. Each table contains the ratings from 30.000 different users for all the movies, and they are stored and load as a **.npz** file in the disk using the **save_npz** and **load_npz** methods from the **scipy.sparse** module. Because the users are 162.541, it fits their ratings in 6 tables with the last one not being full.

Creation of the Tables

The tables are created in the create_pivot_tables() which is called from the constructor. Firstly, it creates in the working directory a folder called **pivot-tables**, which is the folder that the program will store the tables. After that the method parses line by line the ratings.csv file and collects in each line the userId, from the movieId gets the index of that movies in the movies list (because in the list they are sorted and there is a sequence), and the rating. Those three values are stored in three different lists. Also, it counts the different users that it founds. When it collects the ratings from 30.000 different users, it stores the current table in the created folder. The table is named as i-ptable.npz where i is an increasing index starting from the 0. By this way I know that in the first table I have all the users with id in (0, 29.999), in the second (30.000, 59.999) and goes on. The tables are created in the create_table() method. This method uses the three lists to create the **csr sparse matrix** efficiently with the collected values. After that, it subtracts the mean values of the nonzero ratings of each row in each row. By this way the ratings are normalized and I solve the problem to treat missing ratings as negative. So, now I use the **Center Cosine** for calculating the similarities. When the parsing of the ratings file completed, then it stores and the last table with the same way.



Creation of the tables needs about **12 minutes**.

Getting Users Ratings Vector

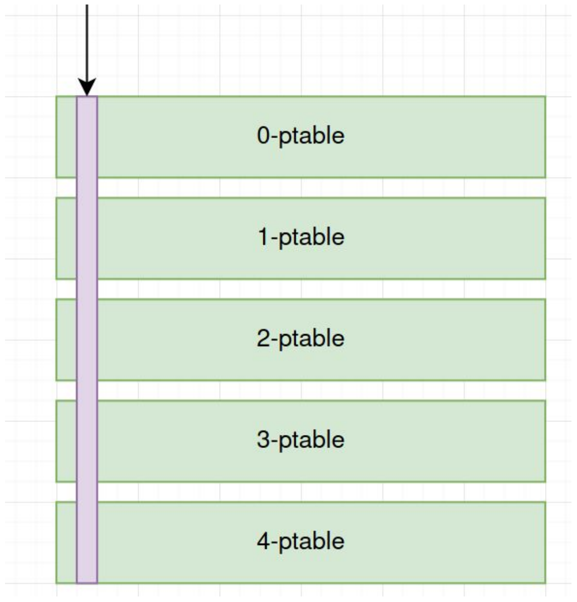
Given a user Id in order to get the vector with his ratings, I have to find in which table this userId is located and in which row.

- The table is given by: **user_table_id** = **int**((**userId** - **1**)/**users_in_each_table**(=**30.000**))
  - For example the user with id = 87.301 is located in the **user_table_id** = **int**(**87.300**/**30.000**)=**int**(**2.9**) = **2**
- The row of the user is given by:
  user_row_id = (**userId-1**) - **user_table_id** * **users_in_each_table**(=**30.000**)
  - In the previous example we get: row = 27.300

So by loading this table we can get the ratings as a sparse vector from the correct row using the getrow() method.

Getting Movies Ratings Vector

Given a movieId is a little trickier to get the vector with the ratings from all the users, because the movieIds are the columns of those tables. So each table contains a part of the ratings of a movie. So, in order to get that vector, first, I get the index of the movieId in the table using the movies list (which I have created in order to get the correct column indexes). Then, by opening all the tables one by one I get the correct column of the movie and I stack them all into one vector using the vstack() method of the sparse matrices. When I parse all the tables, I have created the vector with all the ratings for a movie.



Program Usage

The program requires from the user to give two argument. Firstly the path of the ratings file, and secondly, which method to use for predicting recommendations (more info about the arguments on the **README.md** file). The accepted method values are: ‘user’, ‘item’ and ‘mix’. When the program starts, after creating the pivot tables, it goes in a while loop in which waits for the user to give a user id. When an id is given, then it will calculate and print the top 20 best recommendations that the algorithm made using the requested method. It will print the movie id, the points that it got, and ‘u’ or ‘i’ according to the method that lead to that result (that is used mainly when ‘mix’ method has been selected).

The program terminates when the user gives as input the letter **q** or with the **Ctrl+C** shortcut.

If the requested user has not seen any movies the program returns an empty list. That’s because when we have no ratings for movies, the collaborative filtering is useless. In that case we should recommend items with different approaches (maybe the most popular).

In order to avoid recreating the tables in every run, the program will use the already created tables if the *--load* or *-l* argument is given in the execution command (and if the path is correct).

Methods Algorithms

User-Based

The user based method is selected when the argument `-m user` is given in the execution command. The code of that algorithm is in the `user_based_prediction()` function and gets the desired user id as input. The user based algorithm is explained in the below steps:

1. From the pivot tables it gets the *users ratings vector*, as I already mentioned.
2. After that, It calculates the cosine_similarity between my user's rating vector and all the created pivot tables and the results are concatenated in one vector. By this way I get the similarities of the given user with every other user.
3. From those results it gets the top 20 similar users with the given user (ignoring the first one because it is itself with similarity 1).
4. From the rating vector of the given user, it gets all the movies that the given user has seen (nonzero values) and stores them in a set. Also, for the top20 similar users, it gets the movies that each user has seen with the same way (finding the correct table and extracting it from the row). Those movies are also stored in a set.
5. Then, It subtracts from that set the movies that the target user has seen.
6. Now we have the movies that are *under considerations*. For each movie:
    a. It calculates the weighted average rating which is the sum of the ratings from the similar users multiplied by the similarity of each user with the target user, divided by the sum of those similarities. The equation is shown below

$$r_{xi} = \sum_{y \in N} s_{xy} r_{yi} / \sum_{y \in N} s_{xy}$$

7. Then, it sorts the results.
8. It gets the real movies Ids from the list, which it has created in the constructor.
9. And returns them as a truple (movieId, score, method).

This approach requires about **100 seconds** to complete.

Item-Based

The item based method is selected when the argument `-m item` is given in the execution command. The code of that algorithm is in the `item_based_prediction()` function and gets the desired user id as input. The item based method is used mainly to predict the rating that a user may rate to a specific movie. However, we cannot try to predict all the ratings that a user would do for every movie we have in the dataset and recommend the top 20 because that would need an extreme amount of time to be complete. So I follow a different approach. I find from the movies that my user has seen (starting with his favorites) the most similar (one at a time). If the user has seen it, I find the second most similar and goes on. When I have a similar movie that the user has not seen, I predict its rating for that movie and add it in the list with the results. Then, I move to the next favorite movie and repeat this process. That algorithm loops until It collects 20 ratings, which they get sort and returned as the results. If all the movies that the user has seen parsed, then the program loops them again searching in a deeper level avoiding the movies that the program has already check.

In more details the item based algorithm is explained in the below steps:

1. It finds the movies my target user has seen and sort them from the most favorite to the worst. This is happening by taking the vector with the ratings of the user as I already mentioned, and sorts it by the ratings starting from his best.
2. A while loop starts and it ends only when it has collected 20 different movies.
    a. For every movie my user has seen (starting from his favorites), I find other similar movies. That is done by:
        i. It gets the *vector of the ratings* for the current movie, as I have already mentioned.
        ii. It calculates the cosine similarity between the transposed created vector (ratings of the movie) and with all the other pivot tables, by loading them one at a time.
        iii. It concatenates the results of the cosine similarity into one vector.

    b. Then, it sorts the results, with the most similar at the beginning, and checks one movie at a time. If the user has seen the movie it moves to the next one.
    c. When we have a similar movie that the user has not seen It calculates the predictive rating for that movie of that target user using the predict_rating_for_movie() function.
        i. This function finds similar movies that our user has seen and rated and uses them to predict the rating.
            1. In order to find those movies, it loads and calculates the *movie ratings vectors* for 30.000 movies at a time,
            2. It calculates for those movies the similarities with the target movie and concatenates the results in an array.
            3. It repeats until to parse all the movies and collect all the similarities in the array.

        ii. Then, it sums the similarities of each movie multiplied by the rating of that movie the target user has made,
        iii. Sums those similarities.
        iv. And it divides those two sums. That's my prediction of the rating for that target movie

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

    d. The rating is added on a list and goes to the next movie.

3. When the list contains 20 movie ratings it breaks the loop. If all the movies that user has seen parsed and the method has not collect 20 movies, the repeats the process by searching deeper for movies that the user and the program has not already seen.
4. The results are sorted and the real movies ids are found just like in the user based approach.
5. The predictions are returned similarly as a truple.

This approach requires about **200 seconds** to complete.

Combination (mix)

The mixed method is selected when the argument `-m mix` is given in the execution command. The code of that algorithm is in the `mix_based_prediction()` function and gets the desired user id as input. The mixed algorithm just executes the Item-Based and the User-Based algorithm, collects the results into one list, sorts them according to the score each prediction has gotten, and returns the top 20 as results.

Note

In my first approach, which I have not mentioned, I was creating one pivot table chunk by chunk, with all the ratings inside it. This approach was abandoned because it was not enough "disk based", because at a point I had all the data in the memory. However, It is worthy to mention that, that approach was running almost in Real Time returning results in about **1-2 seconds**.

THE END