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 about 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 to 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 or **MovieRatings**, 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 with 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. 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 remaining. This functionality is executed in the `fit()` method.

On the first loop, in the first chunk, the program picks randomly the first K points which they 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 is 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 inserted at the remaining list. All those points are described by a class called **RemainEnity**. 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 throw away useless 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, 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 described above, belongs into 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. 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. By 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.

$$\{\text{movie\_id}: \{\text{user\_id}: \text{rating}\}\}$$

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 used low threshold values because the vectors are very long and 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 make 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 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 is located at the method that print 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 sceps:

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

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 results will be delivered in the .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). Regardless the given method, this class parses the ratings file twice in the constructor.

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 classes. 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, which is very helpful feature as we are going to see later. Lastly, we get the number of the users that has rated movies.

That process takes about 6 seconds.

Pivot Table

In the second parse I create the **pivot_table** in the create_pivot_table() method. This pivot table has as rows the users, as columns the movies and the values are the rating a user did for a specific movie. It is a sparse matrix and it is declared as csr_matrix from scipy.sparse module.

Creation of Table

Firstly the matrix is declared with the already mentioned shape which is got from the lists that created in the previous parse and at first the matrix is empty. Then, it start parsing the ratings CSV file chunk by chunk with chunk size 100.000. In each chunk, it collects the user ids in a list which will be the indexes of the row of each rating. Also, finds for each movie id in the chunk the correct position using the list with the movies ids and stores them in a list. That list will be the indexes of the columns of each rating. The values of this table are the ratings of the current chunk. Using those three lists, I update the matrix efficiently

When that parse ends, the table contains all the ratings in a memory respectful sparse matrix, which at each row contains the ratings for each user (row = user_id - 1 since the users in the ratings CSV are increased one at a time). Then, it subtracts the mean value of the nonzero ratings of each row from the ratings. By this way the ratings are normalized and I solve the problem to treat missing ratings as negative. By this method I use the **Center Cosine.**

Now the pivot table is completed and it is assigned as a member variable called pivot_table.
This process requires about **950** seconds.

Storing/Loading the Table

Because the creation of the table is a process that takes a large amount of time, it is suggested to be created once and store it, so it can be loaded in a later run.

For this job I use the pickle module. The pivot table can be:

- Stored by giving the --store argument
- And it can be loaded, similarly using the --load argument

The path should be given with the -p argument.

Program Usage

The program requires from the user to give two main 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 table, it goes in a while loop in which waits for the user to give a user id. When an id was 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).

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 table I get my users ratings vector, which is the row in the user_id - 1 position.
2. It calculates the cosine_similarity between my user’s rating vector and the pivot table. By this way I get the similarities with every other user.
3. From those results I get the top 20 similar users with the requesting one (ignoring the first one because it is itself with similarity 1).
4. It gets from the pivot table all the movies that those similar user has seen into a set.
5. It subtracts from that set the movies that our user has seen (which have been also gotten from the pivot table).
6. Now we have the movies that are under considerations. For each movie:
    a. Calculates the weighted average rating which is the sum of the similar users ratings multiplied by the similarity of with our users, divided by the sum of the 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 we have created.
9. And returns them as a tuple.

The user based algorithm is running in real time.

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 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 favorite) 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. Then, I move to the next favorite movie and repeat the process. That algorithm loops until It collects 20 ratings, which they get sort and returned as the results.

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 row with the ratings the user has done from the pivot table, and sorts the ratings.
2. A while loop starts the ends only when it has collect 20 different movies.
    a. For every movie my user has seen, I find other similar movies. That is happening by calculating the cosine similarity between the transposed pivot table and the transposed column of that favorite movie.
    b. Then, it takes sorts the results and checks one movie at a time. It 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 I calculate 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
        ii. Sums the similarities multiplied by the ratings the user has make
        iii. Sums the similarities
        iv. And divides those two sums. That's my prediction of the rating

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

    d. The rating is added on a list and repeat the loop.
3. When the list contains 20 movie ratings it breaks the loop.
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 tuple.

This algorithm requires about 30 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 points each prediction has gotten, and returns the top 20 as results.

THE END