CS5425 Assignment 1 Task 2 Report by A0212221E – Vincent Andrian Chandra

Disclaimer:

1. Since step 1 from the template already works fine, I did not change anything on that file and only finished step 2 and step 3. For step 4_1 my code is still not working properly and step 4_2 and 5 is based on the output of step 4_1 which is still wrong. On this report I will explain the codes and what I have worked in details.
2. I use the last 3 numbers of my matriculation number (A0212**221**E) to filter the recommendation result in step 5.

Step 1 – Creating a score matrix:

From the data.csv where the file is in the format of a table with 3 columns, in step 1 to build a proper score matrix the mapper emits a key value pair with the format of **userID, itemID:score**. That means the shuffling will do its job to group the values **itemID:score** by userID before the pairs are going to reducer phase. In the reduce function, each value of the same key is appended into a string with the format of **itemID1:score,itemID2:score,…** and a new key value pair is emitted using that format.

Step 2 - Creating a co-occurrence matrix:

The purpose of this step is to build a co-occurrence matrix. In the mapper function, the idea is to do a nested loop with the first for loop going through all the values and the second for loop going through all the values again. Then in the second for loop we emit a new key value pair with the format of (itemID1:itemID2, 1) with itemID1 from the first for loop and itemID2 from the second for loop. This means every time a **itemID1:itemID2** pair is formed by going through the nested loop, they will be the key and the count (which is 1) will be the value.

In the reducer, all grouped values belonging to the same key will be aggregated representing the no. of counts for the key **itemID1:itemID2**, essentially telling us how many users choose itemID1 and itemID2 at the same time. The final result should look like this for each line (assuming itemID1 is 104 and itemID2 is 384) **104:384 32** where 32 is the count.

Step 3_1 – Processing score matrix:

Step 3 has 2 parts. The first part is to process the score matrix, which is the output of step 1, and the second part is to process the co-occurrence matrix, which is the output of step 2.

For the first part, the output from step 1 will be processed so that it will have the format of **userID, itemID:score** as a key value pair. Not only reformatting, but since we have to include non-scored items for each user for matrix multiplication purpose, the items are included here in this step. There is an initialization of hashmap (with the format of itemID and score) which is going to store all items that have been scored by the user (the user is the key in this case). Then we go through a loop of all the scored items for the current user, with each iteration a map of itemID and score is inserted in the hashmap.

Now that the hashmap contains all the scored items by the user, to include non-scored items too a loop going through 0 to 499 is created (499 represents the total items existed since the maximum id of item is 499). For each iteration there is a need to check whether an item with the ID of iteration is already scored by the user. If it has been scored, emit a key value pair with the value being the score saved in the hashmap before. On the contrary if it has never been scored, emit a key value pair with the value of 0.0. The for loop does this for every iteration and this results in the key user having all items along with the scores, which is better to represent a matrix with the each user as the column (with length of total user = 1000) and each item as the row (with length of total item = 499).

Step 3_2 – Processing Co-occurrence matrix:
There is nothing really special about this step except it is only reformatting the output of step 2 which is the co-occurrence matrix. The initial format of the key value pair is (itemID1:itemID2, value) and in this step it will be changed to (itemID1, itemID2:value) to help ease the matrix calculation done in the next step.

Step 4_1 – Matrix Multiplication:
The main goal here is to do matrix multiplication and form a new matrix consisting of itemID1 as row and userID as column, just like regular matrix multiplication. What I am trying to do in the map function is to emit a key value pair such that the key will be grouped accordingly so that the grouped values can be used to do the matrix multiplication later in the reducer function.

Since map function takes 2 inputs (one is from step 3_1 which is a score matrix and the other one is from step 3_2 which is a co-occurrence matrix) there is a flag variable to indicate which output file the map is processing at that particular time. For the value in the k,v pair the mapper is going to emit, we need to append a string ("score:" for step3_1 output and "cooccurence:" for step3_2 output) to the value to note it comes from which file because in order to group the values properly in theory the keys emitted by each flag should have similarities. If the keys have similarities then we do not have the means to know whose value it is unless we add another attribute to the value.

For the multiplication process, all the userID and score of score matrix are saved in a hash table which will be used as one of the multiplication parameter later in co-occurrence matrix (when flag is **step3_2**). The biggest hurdle is deciding on what to use as the key for multiplication result. I am still not sure the correct key to use in this case, as my output for this step is still wrong unless I fix the key emitted from the mapper function (currently it only outputs a 500x500 while it should theoretically output 500x1000). I think to achieve 500x1000, **version 2** (see comment in code) of the key value emitter should be used instead of version 1. However, even if version 2 should have the correct logic it produces error and runs for a very long time since it works like a normal matrix multiplication algorithm with 3 loops inside (**i, j, and k** loops, in **version 2** case it is 500x500x1000) and for every iteration it emits a key value pair.

Reducer function only plays the part of summing the grouped multiplication values resulted by the mapper, so the success of this process depends heavily on how the mapper formats the key and value. The reducer emits a key value pair in the format of **itemID1 userID:result**

Cooccurrence Matrix
(m2)

ItemID1/ItemID2	1	2	3	4	5	…	500
1	I = 1	I = 2	I = 3	I = 4	I = 5	…	I = 500
2							
3							
4							
5							
…							
500							

Score Matrix (m1)

ItemID/userID	1	2	3	4	5	…	1000
1	I = 1						
2	I = 2						
3	I = 3						
4	I = 4						
5	I = 500						
…	…						
500	I = 500						

The yellow-colored rows/columns are the input keys of the reducer function while the red-colored columns/rows are the itemIDs to put inside their respective hashmap table and their length (500) is used to iterate for matrix multiplication. The blue-colored cells are the values (also stored along with the itemID in the hashmap table) used for the matrix multiplication. According to version 2 of my code, the blue row in m2 has the position of **i** and it will increase when the row is moving down as the multiplication goes. The blue column in m1 has the position of **k** which will move to the right as loop goes. The loop is represented by 1000 countered loops in the commented code. Thus, the key will have the format of **i,k**.

Step 4_2 - Reformatting
In this step only formatting of the key value pair is done to the pairs of multiplication result from previous step. The formatting aims to make filtering and sorting easier in the next step. The initial format is **itemID userID:result** and it becomes **itemId userId1:val, userId2:val, userId3:val, ...**

Step 5 – Filtering and Sorting
Since I do not see any use for the second input, which is data.csv, I changed the template so it only takes 1 input in this step and the input is step 4_2's output. In the map function it loops through a long list of values for every key and checks whether the userID in the value is equal to the 3 last numbers of my matriculation number (221). If so, the mapper will emit a key value pair in the format of **221 itemID:val**.

Finally, the sorting is done in the reducer function by going through every grouped item values that links to 221 which is the key. To sort by the recommendation value, every itemID and value is stored in a hashmap (here we do not need to store userID in hashmap since the reducer input only has 1 key which is 221). After saving all the items and values, the sorthashmap function is called and the result is stored in a linked list, thus making it easier to append all the items and values into 1 long string and only needs to emit 1 key value pair.

Errors/bugs:

1. Running the program in single-node environment works without bugs. However, running in docker clusters environment results in step4_1 producing outputs which values are all 0. I suspect it is caused by flag **step3_2** being entered first instead of **step3_1**. In flag **step3_1** all values are recorded into a hash table and in flag **step3_2** multiplication between co-occurrence value and the value in the hash table happens thus emitting the key value pairs,

so if the system enters **step3_2** first then the hash table is null and the value in key-value pair will be 0. That may be why the output is different between single-node and clusters.

2. Step4_1 reducer output is not correct.

Result Snapshot:

Recommendation result filtered by userID of 221 which is the output of step 5. This is run on a single-node environment.