Name: Pankaj Bhootra
Matriculation No.: A0144919W
NUSNET ID: E0009011

CS4225 Assignment 1 Report

Task 1

4 MapReduce jobs run in sequence to accomplish the given requirement, namely:

Job 1: Performs word count for input text file 1, using the same approach as in WordCount.java that was provided. Note that the words are converted to lowercase and punctuations are also removed (as discussed with the tutor over email).

Job 2: Performs word count for input text file 2, using the same approach as in WordCount.java that was provided. Note that the words are converted to lowercase and punctuations are also removed (as discussed with the tutor over email).

Job 3: Runs two mappers and one reducer. The first mapper appends "_s1" to the word count to show that this word count is coming from input text file 1. The second mapper appends "_s2" to the word count to show that this word count is coming from input text file 2. The reducer takes upto two values for every key (word). Either it will have two word count values (one for each input text file) or only one (from either of the input text files). In order to filter for only common words, a condition filters for only those keys (words) that have two word count values. Subsequently, we pick the smaller of these two word count values, and output that as value along with the word (key).

Job 4: This job is for sorting the output from previous job in descending order of the word count, so that the most common words appear on top in the output. The mapper takes in input word (key) and the common count (value) and writes the negative of the common count as the key and the word as the value for the reducer. This internally sorts the results before they get to the reducer. Then in the reducer, the input key is this negative count and the input value is the word. We do negative of this key value to get back the positive count. Then for each value (word) that was mapped to this count, we write the count as key and the word as value to the final output.

Task 2

7 MapReduce jobs run in sequence to accomplish the given requirement, namely:

Job 1 (Step1.java): The score matrix is built by grouping the items/scores by users. The mapper receives input in the form "userID, itemID, score". The mapper writes key as "userID", and the value as "itemID:score". The reducer appends all values for the given key (userID) with a comma, giving output of form < "userID", "itemID:score,itemID:score,..." >.

Job 2 (Step2.java): The co-occurrence matrix is built. The mapper takes output of Step 1 as input and writes "item$_A$ID:item$_B$ID" as key and 1 as the value. The reducer sums up the values for each key of the form "item$_A$ID:item$_B$ID", hence giving the co-occurrence matrix as output in the format < "item$_A$ID:item$_B$ID", count > for all item pairs.

Jobs 3 and 4 (Step3.java): Further preprocessing is done to prepare the co-occurrence and score matrices for the multiplication in next step. Step3_1 takes in the score matrix output from Step 1 and converts it into the format < "itemID", "userID:score,userID:score,..." >, i.e., grouping the scores by item for each user. Step3_2 takes in the co-occurrence matrix output from Step 2 and changes it to the form < "item$_A$ID", "item$_B$ID:count,item$_C$ID:count,..." >. This will allow the scores and the co-occurrence count values to be correctly grouped together for multiplication in the next step.

Job 5 (Step4_1.java): This step performs first part of the matrix multiplication, which is multiplying the right numbers as per the rules of matrix multiplication. The mapper receives input from the outputs of both jobs 3 and 4. Hence, it will either receive input in the format < "itemID", "user$_A$ID:score,user$_B$ID:score,..." > or in the format < "item$_A$ID", "item$_B$ID:count,item$_C$ID:count,..." >. Either way, the value gets split by comma and the same key along with each split is output to the reducer.
The reducer receives, for a given itemID (key), a list of values which are in two possible formats: either as < "itemID", "user$_A$ID_user:score" > or as < "item$_B$ID:count" >. This list of values are thus split into two separate lists of user scores and item co-occurrence counts respectively. Then, for each user score, it is multiplied with all the item co-occurrence counts. For each such multiplication, the output for next step is written in the form < "userID,item$_B$ID,itemID", product >.

Job 6 (Step4_2.java): This step performs the second part of matrix multiplication, which is adding the products for each (row, column) index in the resultant matrix. The input that comes in the form < "userID,item$_B$ID,itemID", product > is output to the reducer as < "userID,item$_B$ID", product >. Then, in the reducer, all the matched products for the given key are added together and the same key along with the sum is written to output for the next step. At this stage, we have obtained the calculated recommendation scores for each pair of "userID,itemID".

Job 7 (Step5.java): This step is only for filtering and sorting the results. Here in the mapper, the input is filtered to retain only those that contain a given userID (in this case, the last three digits of my NUSNET ID E0009011, which is 11). The reducer thus only receives a list of item recommendations for a given userID, in the format <"userID", "item$_A$ID:score,item$_B$ID:score,...">. Finally, the SortHashMap.sortHashMap function that was provided is then called, to sort the items in descending order of the calculated recommendation scores. The final output is in the form < "userID", "itemID calculated_recommendation_score" >, sorted in order so as to show the best matching items.