Assignment 1 – Parallel Distributed Processing

Student: Jesse van Duijne

Student number: 552779

Date of submission: 04-05-2020

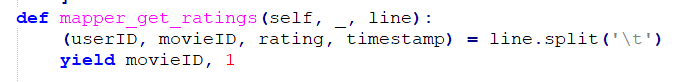
Github link: <https://github.com/Jessevanduijne/PDP_Assignments>

***Note****: this is a private repo. Github-account Michavandermeer is invited as a collaborator to review the code*

# Sum the ratings given for each movie

Functions in order:

1. Mapper: Create key/value pairs of the movieID and 1. One is the amount of ratings given per row that’s being read. This value, the binding, needs to be aggregated in the next step.



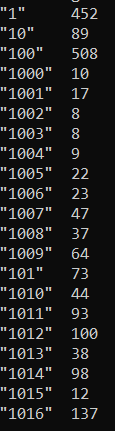
1. Combiner: aggregate (sum up) some of the values (ratings) for the same keys (movieID). The amount of rows will be strongly reduced. **Note:** this is for optimization purposes. If the dataset were to be larger, this would be really useful. In this case I simply implemented to see what it’d do.



1. Reducer: Aggregate (sum up) all of the values (ratings for the same keys (movieID) which are left after the combiner.



Result: 1682 unsorted rows (movies) with an unique movieID and their amount of ratings:



# Extra: Sort the movies by their number of ratings

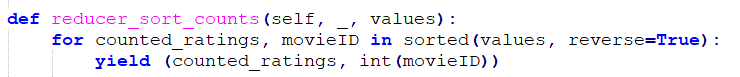
Functions:

1. First, step 1 and 2 from the previous assignment are used, then:
2. Reducer: The counted ratings need to become the new key value, because they need to be sorted. Because we want to sort on the amount of counted ratings, we need this as a new key. A new key/value pair needs to be made: countedRatings/movieID and is returned as the value. The current movieID-key isn’t needed anymore, which is why None is returned. We do now have new pairs like (None, (countedRatings, movieID)). This’ll serve as the input for the sorting in the next step. Sum is used to get the total ratings per movieID.

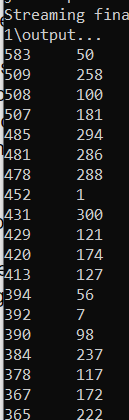


*Because one step can only have one reducer, a new MrStep is initiated..*

1. Reducer: now the ratings can actually be sorted with counted\_ratings as a new key value in the output of this reducer. We’re looping through the sorted rows to yield the highest count first and the lowest one last. Reverse=True is needed in the ‘sorted’ function, otherwise the movieID with the greatest amount of rating-counts would be last. The movieID in the yield is transformed to an integer for readability.



Result: we see that the first value, a movie with the ID of 50 has the largest ratings-count with 583.



# Extra: Sort the genres by total movie ratings

Input: u.data (to get ratings) and u.item (to get genres)

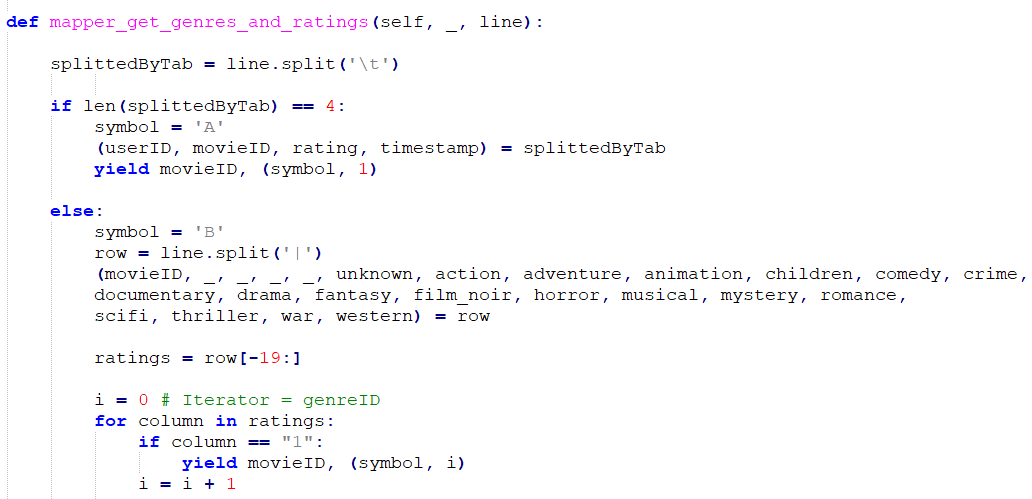
Functions:

1. Mapper: a join is necessary in order to complete this job successfully, because the ratings (u.data) table doesn’t contain movie (u.item) information, like the genre that’s needed. In order to set up the join, we need one list of both tables with a common identifier, which is the movieID.
   1. Input with four columns is identified as a rating (u.data) and yields with a movieID key and a one as a count value. The symbol ‘A’ is passed to let the job recognize this as a rating input later on\*.
   2. Other input, which can only be a movie (u.item) will otherwise be mapped. The input serves every genre as a column, which would lead to a lot of unnecessary mapping. Because of this, the genre-columns are transformed to genreID’s.

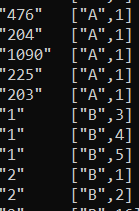
The function loops through all 19 last columns (individual genres) and checks for a boolean with value true (1). If so, the movieID with the genreID is yielded along with symbol ‘B’ for recognition as a genre\*. To secure that the first column is actually the genre with genreID 0, a manual check in file u.genre has been conducted.

It can of course be possible that one movieID has multiple genres. If a movie has two genres, two rows will be created with the same movieID and a different genreID.

\* Why is the symbol really needed? genreID might be 1, which can also be the count of a rating since the key/value is either *movieID, (symbol, count)* *OR movieID, (symbol, genreID).* The symbol creates distinction between the two

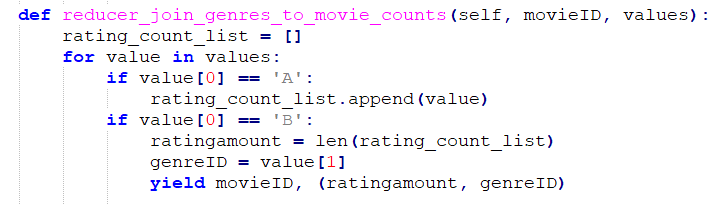


Below a part of the output is shown:



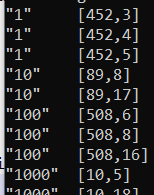
1. Reducer: The reducer shuffles and sorts the mapped input. A line with movieID 1 will for example have 30 values, of which the first 25 are (A, 1) and the last five (B, 1) or (B, 2) is. We can loop through all these values:

* Add ratings (symbol ‘A’) to a new list, so we can get the amount of ratings for a movie
* When genres (symbol ‘B’) is reached, the loop is done looping through values with symbol ‘A’. We can now get the length of that list, which is the amount of ratings per movie, and yield that for every genre that movie has.



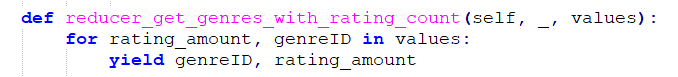
The result is as follows:

<MovieID, ratingCount, genreID>



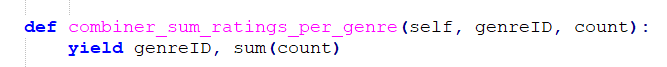
*Because one step can only have one reducer, a new MrStep is initiated..*

1. Reducer: we do need a list with genreID as key, ratingCount as value in order to successfully count the ratings per genre. This is done by looping through the values and yielding genreID/rating\_amount as new key/value pairs. Sorting cannot be done yet, because genreID is not the key yet.

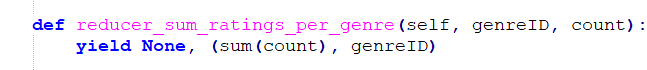


*Because one step can only have one reducer, a new MrStep is initiated..*

1. Combiner: sum up most of the ratings per genreID key. A combiner is optional, but is used as optimization here so the reducer has to do less work.

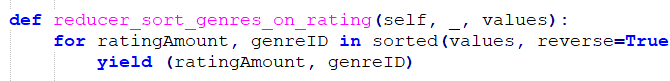


1. Reducer: sum up all of the ratings per genreID key. Also, ratingCount should be the key in the next step because we want the list to be sorted by this column. Because of this, None is returned as the current key and ratingCount/genreID is returned as value.



*Because one step can only have one reducer, a new MrStep is initiated..*

1. Reducer: sorting the ratingAmount/genreID pairs. The sorted values is being looped through, and yielded as final ratingAmount/genreID key/value pair.



The result is as follows:

We can see that genreID 8 contains the most movie-ratings with 39895 ratings. That’s a lot! Especially in comparison to genreID 0, which has only 10. Might be logical, since that genre is called *unknown.*