**RECOMMENDER**

Rubén Ventura

9305420195

**CONTENT**

[**SUMMARY OF THE PROJECT** 3](#_Toc43002939)

[**DESCRIPTION OF THE CODE** 4](#_Toc43002940)

[read\_file( input\_data\_file, data ): 4](#_Toc43002941)

[write\_file( input\_file\_name, final\_ratings ): 4](#_Toc43002942)

[calculate\_similarity( user\_a, user\_b, common\_items ): 5](#_Toc43002943)

[get\_common\_items( user\_a, user\_b): 6](#_Toc43002944)

[get\_recommendations( target\_user, user\_id, training\_data ): 6](#_Toc43002945)

[main( ): 7](#_Toc43002946)

[**INSTRUCTIONS FOR COMPILING** 8](#_Toc43002947)

[**ADDITIONAL SPECIFICATIONS** 8](#_Toc43002948)

# **SUMMARY OF THE PROJECT**

For this project, we were asked to implement a recommender system, which given some training data and some incomplete test data would be able to predict and “fill the gaps” in the test data. In this specific case, we were given a dataset containing some users whom had rated different movies giving values ranging from 0 to 5, and we were asked to predict which ratings they would be most likely to give to the movies they have not seen yet.

This project was the most challenging one, since I struggled a bit understanding and implementing both the similarity and (especially) the aggregation formulas. I was also surprised that the timestamp values were not used at all.

But the thing that has surprised me the most is the fact that I was able to achieve the best results (RMSE = 0 for all 5 datasets) with an algorithm that is not complete/working as I understand it should work. I have worked on many different versions for the code (I only uploaded one to the repository) which in my opinion were a lot closer to the theorical formula (taking the weighted predicted ratings, using the Pearson’s Correlation Coefficient and the average scores of each user), but the one that has given the best (perfect) results is by only getting the first recommendation we can find for each unrated item for each user, not taking into account any other similar users who have rated that same item (which in my opinion would be the correct way of doing it).

A solution for this problem could be easily implemented by modifying and adding a few lines to the code (around line 90, in the “get\_recommendations( )” function), by keeping track of the ratings and similarities of each user for each common item (i.e.: ratings[item] = {coefficient1: rating1, coefficient2: rating2, …} and then calculating the proper predicted rating using all the ratings. I tried this method, but my results with the RMSE were awfully worse, so I decided to stick with the, in my opinion, worse algorithm.

Back to the program as a whole, I used Python 3 again, it is all written in 1 single file, and the resources I used were the presentation and a lot of different webs with explanations of the Pearson Correlation Coefficient and and Collaborative Filtering.

# **DESCRIPTION OF THE CODE**

## read\_file( input\_data\_file, data ):

# Read test and trainig data and stores in double-keyed (user, item) global dictionary

# Format expected: [user\_id]\t[item\_id]\t[rating]\t[time\_stamp]\n

def read\_file(input\_data\_file, data):

    with open (input\_data\_file, 'r') as f:

        for line in f.read().split('\n'):

            if line != '':

                # 2-key dictionary

                user\_id, item\_id, rest = line.split('\t', 2)

                if int(user\_id) not in data:

                    data[int(user\_id)] = {}

                data[int(user\_id)][int(item\_id)] = int(rest[0])

Function used to read the data from the input files and store it the “data” parameter. Reads line by line and expects at least 3 values per line. The first value will be the user ID and will be used as the primary dictionary key. The next values will be the item ID. For each different item, we will create a key-value pair inside the primary dictionary that uses the user as a key. We read the rest of the items in the line, but we will only be interested in the third one, which will be the rating given by that user to the item we just read in that line, so we store that value and ignore the rest of the line. We repeat the process for the whole file.

## write\_file( input\_file\_name, final\_ratings ):

# Generate output files and write the results

# Output format: [user\_id]\t[item\_id]\t[rating]\n

def write\_file(input\_file\_name, final\_ratings):

    output\_file\_name = input\_file\_name + '\_prediction.txt'

    with open(output\_file\_name, 'w') as f:

        for user in final\_ratings:

            for item in final\_ratings[user].keys():

                f.writelines(str(user) +'\t' + str(item) +'\t' + str(final\_ratings[user][item]) +'\n')

Function used to generate the output file. It needs the name of the input file in order to be named in the format we are asked to (u#.base\_prediction.txt). Reads each value from the final ratings dictionary (which has the format: final\_ratings[user][item]: rating) and writes it in the file following the format “[user] \t [item] \t [rating] \n”.

## calculate\_similarity( user\_a, user\_b, common\_items ):

# Calculates similarity between two users using Pearson's Correlation Coeficient

# I used the formula in the presentation

def calculate\_similarity(user\_a, user\_b, common\_items):

    # Sum of values

    sum\_a = 0

    sum\_b = 0

    # Sum of square values

    square\_sum\_a = 0

    square\_sum\_b = 0

    # Sum of products

    product\_sum = 0

    for item in common\_items:

        sum\_a += user\_a[item]

        sum\_b += user\_b[item]

        square\_sum\_a += user\_a[item] \*\*2

        square\_sum\_b += user\_b[item] \*\*2

        product\_sum += user\_a[item] \* user\_b[item]

    # Denominator

    d = math.sqrt((square\_sum\_a - (sum\_a\*\*2) / len(common\_items)) \* (square\_sum\_b - (sum\_b\*\*2) / len(common\_items)))

    if d == 0:  # Divide by 0, can't happen

        return 0

    else:

        # Numerator

        n = product\_sum - (sum\_a \* sum\_b / len(common\_items))

        return n/d

This function is given two users (dictionaries of item:rating pairs) and a set of common items. With these data, it calculates the Pearson Correlation Coefficient between the two users (I tried to make the function as simple as I could)

## get\_common\_items( user\_a, user\_b):

# Compares item between two users and returns the common ones

def get\_common\_items(user\_a, user\_b):

    return (set(user\_a.keys())).intersection(set(user\_b.keys()))

This little helper function was made for clarity. Given two users (dictionaries) returns a set with all the keys (item ID) shared between them.

## get\_recommendations( target\_user, user\_id, training\_data ):

# Compares the user with the training data and gets a recommended set

def get\_recommendations(target\_user, user\_id, training\_data):

    recommendations = {}

    new\_items = set()

    for training\_user\_id in training\_data:

        if training\_user\_id != user\_id: # dont compare with itself

            common\_items = get\_common\_items(target\_user, training\_data[training\_user\_id])

            # if users have at least 1 item in common, calculate similarity (Pearson's Correlation Coefficient)

            if len(common\_items) > 0:

                PCC = calculate\_similarity(target\_user, training\_data[training\_user\_id], sorted(common\_items))

                if PCC > 0:

                    for item in training\_data[training\_user\_id]:

                        if item not in target\_user or target\_user[item] == 0:

                            recommendations[item] = 0

                            recommendations[item] += training\_data[training\_user\_id][item] \* PCC

    return recommendations

Largest and most important function for the collaborative filtering method. Given one target user, whose ratings we want to predict, we iterate through every other user in our training data and compare their common items. If we find any coincidence, we calculate their correlation. If we get a positive value, we look for all the items the user in the training data rated and were not rated by our target user, and add them to the recommendations dictionary, weighted. We repeat for each item and move to next user in training data. Repeats until all the users have been compared and returns the dictionaries with the predicted values.

## main( ):

# 2 arguments expected: training data name, test data name

def main():

    test\_data = {}

    training\_data = {}

    final\_ratings = {}

    training\_data\_file\_name = sys.argv[1]

    test\_data\_file\_name = sys.argv[2]

    # Read files and store data in global dics

    print("Loading data")

    read\_file(training\_data\_file\_name, training\_data)

    read\_file(test\_data\_file\_name, test\_data)

    # Recommendations using Collaborative Filtering

    print("Calculating")

    for user\_id in test\_data:

        recommendations = get\_recommendations(test\_data[user\_id], user\_id, training\_data)

        if user\_id not in final\_ratings:

            final\_ratings[user\_id] = {}

        for item in recommendations:

            final\_ratings[user\_id][item] = recommendations[item]

        # Merge recommendations with previous ratings

        final\_ratings[user\_id].update(test\_data[user\_id])

    # Generate output file

    print("Writing output file")

    write\_file(training\_data\_file\_name, final\_ratings)

    print("Done")

Main function of the project. Prepares dictionaries for storing the test and training data, and the one that will be filled with the results. Reads the input files and stores the data in the dictionaries, after that, iterates through each user in the test data, gets a recommended set for it and stores it in the final ratings dictionary. After that, merges the predicted ratings with the previous ratings (since we need them to calculate the accuracy with the PA4.exe file) and moves to the next user. When all the users have been given a recommended set, sends the dictionary to the write\_file( ) function to generate the document.

# **INSTRUCTIONS FOR COMPILING**

The whole program is written in Python 3.x, in a single .py file, so no compilation is needed. It expects two arguments as input, in this order: 1) training data name, 2) test data name. So, going to the directory where the program and the input files are, the command for executing would be:

python recommender.py [training\_data\_name] [test\_data\_name]

ex:

python recommender.py u1.base u1.test

# **ADDITIONAL SPECIFICATIONS**

All the files share the same directory, including the program files, the input files and all the newly generated output files. For testing, the output files and the test data files need to be moved to the same directory as the PA4.exe file for the latter to work.