**Vrije Universiteit Amsterdam**

**Computational Thinking**

**Project Assignment: Spotify (v2)**

**Group number: 28**

**Members with student numbers:**

Mateusz Kielan 2774206

Jan Burakowski 2764440

Szymon Czternasty 2774641

IJsbrand de Meijere 2679000

**Date: 14.12.2022**

Inhoudsopgave

[Context Task 3](#_Toc121924828)

[Design process 3](#_Toc121924829)

[Algorithms 4](#_Toc121924830)

[Main.py 4](#_Toc121924831)

[Task1.py 4](#_Toc121924832)

[Task2.py 4](#_Toc121924833)

[Task3.py 5](#_Toc121924834)

[Pseudocode 5](#_Toc121924835)

[Flowchart 5](#_Toc121924836)

[Divisions of tasks 5](#_Toc121924837)

[Appendix 6](#_Toc121924838)

[Appendix 1.1 6](#_Toc121924839)

[Appendix 1.2 6](#_Toc121924840)

[Appendix 1.3 7](#_Toc121924841)

[Appendix Flowcharts 8](#_Toc121924842)

[Problem 1 8](#_Toc121924843)

[Problem 2 9](#_Toc121924844)

[Problem 3 9](#_Toc121924845)

[Bibliography 10](#_Toc121924846)

### Context Task

Recommendation systems were invented shortly after the invention of the world wide web (WWW). Belkin and Croft [1], used a recommendation system for the retrieval of data. Goldberg invented the Tapestry algorithm [2], where users would be recommended certain content based on evaluation of historic data.

Since then the systems have been studied in academia and applied in almost every industry. Recommendation systems are now one of the most widely used algorithms in the (virtual)world [3]. The most famous being the Google, YouTube and Spotify algorithms.

Spotify uses two algorithms to find the best recommendations for its users. The first being a contend based filtering. This algorithm focusses on the classification of the music the user listens to. The second algorithm is the Singular Value Decomposition (SVD) algorithm, based on collaborative filtering. This algorithm focusses on the user’s preferences, and change of preferences over time. This recommendation is predicting the next popular song/genre/artist for the user based on constantly changing selections.

Our task is to create a recommendation system that enables the users of Spotify to discover new music and enjoy their listening experience even more. The system should be able to recommend songs to the user based on; individual selection, the mood of the user. The program should also be able to select a next song for the user, that based on their listening habits is ideal for the user.

### Design process

The design of the process was done in several steps. Firstly we subdivided the problems. This to clearly see what the necessary steps were to solve the problem. Discussing different solution strategies to see what we thought of the best to solve the problem. After finding the optimal strategy, a flowchart of the process was created. The flowchart gives a clear visual illustration of the solution the group came up with and clearly visualises the way to tackle the problem. Following the flowchart is writing the pseudocode. This is done to see what the structure and logic of the program are going to look like. After seeing how the program should be designed, the group could clearly see how the algorithm must be programmed. The last step of the solution is the coding of the program.

Finding the optimal way to tackle a problem was one of the encountered difficulties. However, by discussing within the group about the possible options to solve the problem this was tackled quickly.

### Algorithms

In this chapter we will be describing the used algorithms and their functions. The explanation of the algorithms is followed by the flowchart and then the algorithms pseudocode.

##### Main.py

Main.py is the main file of the project. In this file we create the class “User” which represents every user in the program. Every user has 2 attributes: user\_songs and number. User\_songs attribute represents all songs that user has listened to (we assign those randomly) and the number attribute represents the user number from 1-100. This attribute is used in the final output of the task. Furthermore the main takes care of the file handling. This means that the file is opened, filtered and all the values are converted to the desired type in the main. Task1, Task2, Task3 are all stored in separate files. In the main we call all the functions and the file with final results is created.

The output of the main.py pipeline is a custom CSV file output to the listeners specific preference. The output of our recommendation system can be found by running the code.

##### Task1.py

The first program aims to suggest a playlist based on the songs that the user has listened to. The framework looks as follows:

* Take the dictionary of playlists and a list of songs that the user has listened to as input.
* Take each playlist and perform a linear search for each of the songs that the user has listened to and record how many songs match the requirements:

1. Take each playlist from the dictionary.
2. Take each song from the list of user songs and perform a linear search
3. If it matches a song in a playlist, add 1 to the "in user songs" counter.
4. If it does not match the song, add 1 to the "not in user songs" counter.

* If both counters fulfil the requirements, a playlist (a playlist should contain three songs that the user has listened to and three that the user did not listen to) is suggested to the user.

##### Task2.py

In the second week the algorithm should return 5 songs according to the users favourite genre. For the second task the algorithm is as follows: First, define two functions: ‘find\_genre’ and ‘find\_favourite’.

Function find\_genre takes two parameters. List of songs that the user has listened to and a list of all songs.  It loops through every song that the user has listened to and creates a dictionary where keys are the genres of the songs and values are the number of occurrences of those genres. Please note that we calculate the occurrences for every genre in the dataset. After the dictionary was created the find\_favourite function is called. Find \_favourite converts the dictionary to a list of tuples and sorts it using bubble sort. Therefore the last element of the list will be the most favorite genre of the user. After that loop through all the songs and create a list of songs with the user's favorite genre. Then select 5 random songs from that list as an output. If there arent enough songs in the list of songs with a favorite genre then fill in the missing songs by selecting random songs from the dataset.

##### Task3.py

0: happy

1: party  
2: calming

3: lounge

|  |  |
| --- | --- |
| Happy | valence > average valence of all songs liveness > average liveness of all songs speechiess > average speechiess of all songs |
| Party | BPM > average BPM of all songs energy > average energy of all songs danceability > average danceability of all songs popularity > average popularity of all songs |
| Calming | danceability < average danceability of all songs BPM < average BPM of all songs length > average length of all songs |
| Lounge | speechiess < average speechiess of all songs length > average length of all songs loudness < average loudness of all songs |

In the third week, the algorithm should check the user’s mood based on the songs listened to previously. First, the global variables are defined - the mean of all values of particular features in the song dictionary. Type\_det function is created, the function takes all songs (list of all songs) and user\_songs (list of songs the user has already listened to) and returns the list of four elements, each representing a different type.

The variables with names of the features are created and stored the values of features. The song is being classified based on the following criteria:

If the song meets the criteria of the type, the value at the corresponding index in the types variable is changed to 1. Finally, the types variable is returned.

Then the algorithm creates the pref\_dict variable which stores the type in the numerical form as the key and the frequency as the value. Next, the pref\_dict is transformed into a sorted list of tuples to get the most popular type.

Support variables first\_list, seecond\_list, and suggestions are created. If the pref\_dict contains more than 1 element (there is more than 1 type listened to by the user) the first\_list and second\_list are populated with the most popular type songs and second most popular type songs respectively. If there are at least 3 songs in the "first\_list" and 2 songs in the "second\_list", random 3 and 2 songs respectively are added to the "suggestions". If there are fewer songs than it is required it adds more from the other list. In the situation when both lists are too small to recommend 5 songs, random songs chosen from all songs are added. When it is only 1 type of music listened to by the user in the earlier weeks, 5 songs from the most popular type are added to the suggestions, when there are not 5 elements in first\_list, random songs are added. Finally, the suggestions are transformed into a list of titles of the recommended songs and the list is returned from the type\_det function.

##### Pseudocode

The pseudocode of the program can be found in the appendix 1.1, 1.2 and 1.3. The pseudocodes describe how the program works and how the operations of the algorithm follow one another.

##### Flowchart

The flowcharts of the programs were created in Draw.io. The code to finding the charts within the Draw.io infrastructure are found attached in the Zip file attached to this report.

### Divisions of tasks

The divisions of the roles were separated in to four parts. The strategy of the team was thought of by all of us together. The meetings were efficient effective and pleasant. Working together on how to tackle the subproblems and thinking of a best suited strategy. Problem one was worked on and solved by Jan Burakowski. The second problem was solved by Mateusz Kielan. The third problem was worked on and solved by Szymon Czternasty. Assistance with the code, the presentation and report were done by IJsbrand de Meijere. The joint effort of the team made this a successful and meaningful learning experience for all of us.

### Appendix

This is the appendix of the report. Refferences to the appendices can be found in the text.

###### Appendix 1.1

**Task 1.**

Function program\_1( all playlists, user songs) returns a list of 5 songs from recommended playlist:

chosen\_playlist = -1

loop for playlist in all playlists:

counter\_is = 0

counter\_not = 0

loop for song in user songs:

if song is in playlist:

counter\_is + 1

else:

counter\_not + 1

if counter\_is and counter\_not are both greater or equal to 3:

chosen\_playlist = playlist

end the loop

If chosen playlist is still equal to -1:

return a string with information that playlist was not found

else:

recommended = list()

loop for all of the song in chosen\_playlist:

add song to the recommended list

return random 5 songs from from the recommended list

###### Appendix 1.2

**Task 2.**

Function find\_genre(user songs, all songs) returns function:

gener\_dict = dict() ← dictionary of genre name as key and number of occurrences as value

Loop for song in user\_songs:

If song,genre in genre\_dict:

counter + 1

Else:

Create a new key:value pair = song.genre : counter=1

Endloop

return find\_favourite(genre\_dict, all\_songs)

Function find\_favourite(gener\_dict, all\_songs) returns a list of 5 songs:

var genre\_list ← converting the genre\_dict to list of tuples

var favourite\_genre\_list = [] ← list of songs of the users favourite genre

genre\_list = bubblesort(genre\_list) #sorting the list using bubblesort

var favourite\_genre = gener\_list[-1][0] ← favourite\_ genre stores favourite user genre

loop for song in all\_songs:

if song.genre == favourite\_genre:

favourite\_genre\_list add song.title

endloop

return 5 random songs from favourite\_genre\_list

###### Appendix 1.3

**Task 3.**

function week\_3(all songs, user songs):

function type\_det(song):

types = [0, 0, 0, 0]

if song[valence] > avg\_valence and song[liveness] > avg\_liveness and speechiness > avg[speechiness]:

types[0] = 1

if song[BPM] > avg\_BPM and song[energy] > avg\_energy and danceability > avg[danceability]:

types[1] = 1

if song[danceability] > avg\_danceability and song[BPM] > avg\_BPM and length > avg[length]:

types[2] = 1

if song[speechiness] > avg\_speechiness and song[length] > avg\_length and loudness > avg[loudness]:

types[3] = 1

return types

pref\_dict = {}

loop for song in user songs:

if type\_det(song)[0] == 1:

if 0 in pref\_dict:

pref\_dict[0] = pref\_dick[0] + 1

else:

pref\_dict[0] = 1

if type\_det(song)[1] == 1:

if 1 in pref\_dict:

pref\_dict[1] = pref\_dick[1] + 1

else:

pref\_dict[1] = 1

if type\_det(song)[2] == 1:

if 2 in pref\_dict:

pref\_dict[2] = pref\_dick[2] + 1

else:

pref\_dict[2] = 1

if type\_det(song)[3] == 1:

if 3 in pref\_dict:

pref\_dict[3] = pref\_dick[3] + 1

else:

pref\_dict[3] = 1

end loop

pref \_dict = list(sort(pref\_dict))

first\_list = []

second\_list = []

suggestions = []

loop for song in all songs:

if type\_det(song)[pref\_dict[-1][0]] == 1:

first\_list.append(song)

if type\_det(song)[pref\_dict[-2][0]] == 1:

second\_list.append(song)

end loop

if len(pref\_dict) > 1:

suggestion.append(three random elements(first\_list))

suggestion.append( two random elements(second\_list))

else:

suggestion.append(five random elements(first\_list)

return suggestion

###### Appendix Flowcharts

###### Problem 1

###### Problem 2

###### Problem 3

### Bibliography

Nicholas J. Belkin and W. Bruce Croft. 1992. Information filtering and information retrieval: two sides of the same coin? Communications of The ACM 35, 12 (1992), 29–38.

David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. 1992. Using collaborative filtering to weave an information tapestry. Communications of The ACM 35, 12 (1992), 61–70

Zhenhua Dong , Zhe Wang, Jun Xu , Ruiming Tang , Jirong Wen. 2022. A Brief History of Recommender Systems, 1-9. Gaoling School of Artificial Intelligence, Renmin University of China, China.

Yehuda Koren. 2010. Collaborative filtering with temporal dynamics. Communications of The ACM 53, 4 (2010), 89–97.