HAROKOPIO UNIVERSITY
DEPARTMENT OF INFORMATION & TELEMATICS

Decision Making Systems Work

Group 2:

Kazakos Christos, it22033

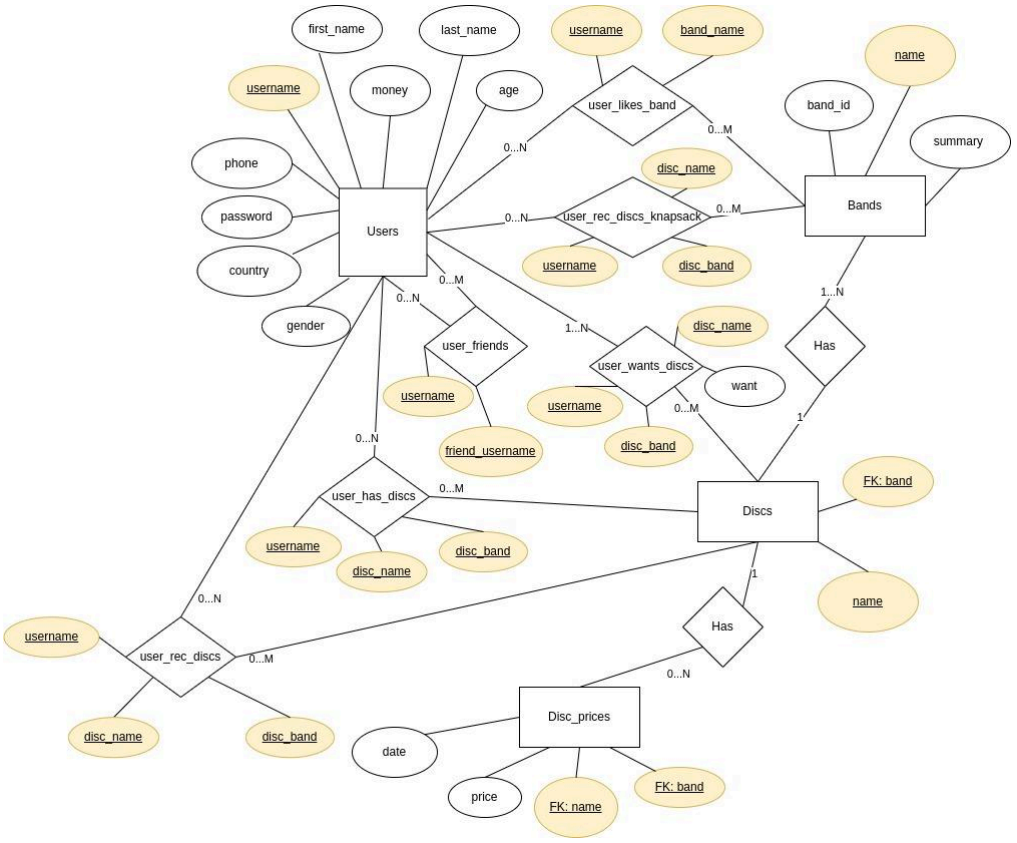
Konstantinos Katsaras, it22045

Manousos Linardakis, it22064

Contents

(a.) Design the data model that will represent the users of the community, the bands they like and the records they own. Implement the data model scheme in a relational (SQL like) database (DB) and create functions that insert random data of your choice for users.

ER Diagram:



Here we have the Queries to create the base tables:

```
CREATE TABLE IF NOT EXISTS Users(
                    usernameVARCHAR(50) PRIMARYKEY,
                    passwordTEXTNOTNULL,
                    first_nameVARCHAR(50)NOTNULL,
                    last_nameVARCHAR(50)NOTNULL,
                    genderCHAR,
                    countryVARCHAR(50),
                    ageINTEGER,
                    phoneVARCHAR(100)NOTNULL
                )

CREATE TABLE IF NOT EXISTSBands (
                    nameVARCHAR(50) PRIMARYKEY,
                    summaryTEXT,
                    band_idINTEGER
                )

CREATE TABLE IF NOT EXISTSDiscs (
                    nameVARCHAR(250)NOTNULL,
                    bandVARCHAR(50)NOTNULL,
                    PRIMARYKEY(name, band),
                    FOREIGNKEY(band)REFERENCESBands (name)
                )

CREATE TABLE IF NOT EXISTSuser_has_discs (
                    usernameVARCHAR(50),
                    disc_nameVARCHAR(250),
                    disc_bandVARCHAR(50),
                    CONSTRAINTpk_user_has_discs PRIMARYKEY(username,
disc_name, disc_band),
                    CONSTRAINTfk_user_has_discs_username FOREIGNKEY(username)
                        REFERENCES users(username),
                    CONSTRAINTfk_user_has_discs_disc FOREIGNKEY(disc_name,
disc_band)
                        REFERENCESdiscs (name, bands)
                )

CREATE TABLE IF NOT EXISTSuser_likes_band (
                    usernameVARCHAR(50),
                    band_nameVARCHAR(50),
                    CONSTRAINTpk_user_likes_band PRIMARYKEY(username,
band_name)
                    CONSTRAINTfk_user_likes_band_username FOREIGNKEY(username)
                        REFERENCES users(username),
                    CONSTRAINTfk_user_likes_band_name FOREIGNKEY(band_name)
                        REFERENCESBands (name)
                )
```

```
CREATE TABLE IF NOT EXISTSUser_Friends (
                        usernameVARCHAR(50)NOTNULL,
                        friend_usernameVARCHAR(50)NOTNULL,
                        PRIMARYKEY(username, friend_username),
                        FOREIGNKEY(username)REFERENCES Users(username),
                        FOREIGNKEY(friend_username)REFERENCES Users(username)
                )

CREATE TABLE IF NOT EXISTSdisc_prices (
                        date DATENOTNULL,
                        valuesFLOAT,
                        nameVARCHAR(250)NOTNULL,
                        bandVARCHAR(50)NOTNULL,
                        FOREIGNKEY(name, band)REFERENCESDiscs (name, band)
                )
CREATE TABLE IF NOT EXISTSuser_rec_discs (
                        usernameVARCHAR(50),
                        disc_nameVARCHAR(250),
                        disc_bandVARCHAR(50),
                        CONSTRAINTpk_user_rec_discs PRIMARYKEY(username,
disc_name, disc_band),
                        CONSTRAINTfk_user_rec_discs_username FOREIGNKEY(username)
                                REFERENCES users(username),
                        CONSTRAINTfk_user_rec_discs_disc FOREIGNKEY(disc_name,
disc_band)
                                REFERENCESdiscs (name, bands)
                )
```

```
CREATE TABLE IF NOT EXISTSuser_wants_discs (
                        usernameVARCHAR(50),
                        disc_nameVARCHAR(250),
                        disc_bandVARCHAR(50),
                        wantINTEGER,
                        CONSTRAINTpk_user_wants_discs PRIMARYKEY(username,
disc_name, disc_band),
                        CONSTRAINTfk_user_wants_discs_username FOREIGNKEY
(username)
                                REFERENCES users(username),
                        CONSTRAINTfk_user_wants_discs_disc FOREIGNKEY(disc_name,
disc_band)
                                REFERENCESdiscs (name, bands)
                )

CREATE TABLE IF NOT EXISTSuser_rec_discs_knapsack (
            usernameVARCHAR(50),
            disc_nameVARCHAR(250),
```

```
        disc_bandVARCHAR(50),
        CONSTRAINTpk_user_rec_discs_knp PRIMARYKEY(username, disc_name,
        disc_band),
        CONSTRAINTfk_user_rec_discs_username_knp FOREIGNKEY(username)
            REFERENCES users(username),
        CONSTRAINTfk_user_rec_discs_disc_knp FOREIGNKEY(disc_name,
        disc_band)
            REFERENCESdiscs (name, bands)
    )
```

(b.) For each user of the community, create functions that retrieve data from opendata in relation to the bands he listens to and the records he is interested in and to fill the corresponding tables in the NW. [Rest calls to[1] [2] and MySQL Lecture 3]

We used the bands (set in the .env file):

```
BAND_NAMES=coldplay scorpionsthe+beatlesqueen acdc u2
```

Also we use both recommended apis (discogs & lastfm). From discogs we get the names of the discs of each band (so that we can use them later in the webscraping of the discogs site), while from lastfm we get additional information such as the band summary and band name. Then at the base, we connect the bands with the disks they have with foreign keys. For users, we then randomly select from the available discs and bands, so as to fill the corresponding tables (user_likes_band, user_has_disc).

(c.) Implement a function that checks for and handles missing[4] , duplicate[5] or outliers[6] values in any of the NW attributes.

To detect outliers, we use the winsorize technique, in which the outliers take the appropriate maximum value of the boundaries. Specifically, we used scipy's winsorize (tutorials). As for the missing values, we take the average of the previous ones and substitute it in place of the NaN. Finally, duplicates are treated with the constraints we have defined in the base (unique, primary keys). In case an identical element already exists in the database, we ignore it (psql has the command INSERT INTO ... ON CONFLICT DO NOTHING, which simply ignores records that are duplicates).

(d.) Draw some statistical conclusions about the data features[7] [8]

Questions:

1. average age of users listening to Scorpions
2. countries that listen to music the most
3. where most Scorpions listeners are from
4. what gender are most Scorpions listeners?
5. band with the biggest audience (among user favourites)
6. average drives that users have
7. where are the users listening to Scorpions from
8. where are the users who have the "My Universe" disc from
9. number of users who have the “My Universe” disc.
10. more genders that have the record "My Universe".
11. which band do users prefer per country
12. Top 5 Discs (in quantity)

Execution:

```
print("Q1 ================ ")
print(avg_user_band_age(conn,"scorpions")) print(
"Q2 ================ ")
print(countries_most_music(conn)) print(
"Q3 ================ ")
print(band_most_listeners(conn,"scorpions")) print("Q4
================ ")
print(band_most_gender(conn,"scorpions")) print(
"Q5 ================ ")
print(band_with_most_listeners(conn)) print(
"Q6 ================ ") print
(avg_disc_count(conn)) print("Q7
================ ")
print(band_users_by_country(conn,"scorpions")) print("Q8
================ ")
print(disc_users_country(conn,"My Universe")) print("Q9
================ ")
print(num_users_with_disc(conn,"My Universe")) print(
"Q10 ================ ")
print(disc_most_gender(conn,"My Universe")) print(
"Q11 ================ ")
print(most_listened_bands_by_country(conn)) print(
"Q12 ================ ")
print(top_x_discs_by_quantity(conn))
```

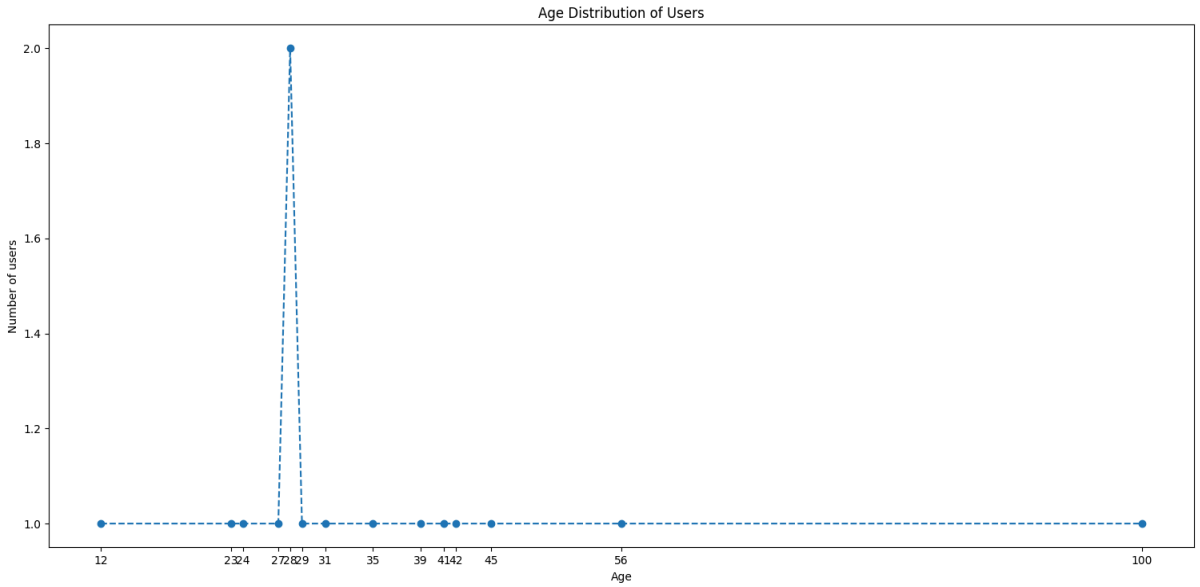
Results:

```
Q1 ===============
17.00
Q2 ===============
{'Italy':1,'United Kingdom':1,'Germany':1,'Canada':2,'Greece': 2,'France':1,'South Africa':
1,'Japan':1,'United States':1, 'Brazil':1,'Australia':1,'Spain':1,'Mexico':3,'Poland':1} Q3
===============

Canada
Q4 ===============
M
Q5 ===============
Queen
Q6 ===============
3
Q7 ===============
{'Canada':1,'Germany':1,'Greece':1,'Italy':1,'Mexico':1, 'United States':1} Q8
===============

['Greece','South Africa'] Q9
===============
2
Q10===============
F
Q11 ===============
{'Australia': ('The Beatles',1),'Brazil': ('Coldplay',1),'Canada': ('Scorpions',3),'Germany': (
'ACDC',2),'Greece': ('Scorpions',3), 'Italy': ('Coldplay',2),'Japan': ('Scorpions',2),'Mexico': (
'Scorpions',1),'South Africa': ('Coldplay',3),'Spain':

('Scorpions',3),'United Kingdom': ('Scorpions',2),'United States': ('Scorpions',1)}

Q12 ===============
[('U218 Singles (deluxe version)','U2',2), ('Moment of Glory',
'Scorpions',2), ('Eye II Eye','Scorpions',2), ('Rock Believer', 'Scorpions',2), ('My
Universe','Coldplay',2)]
```

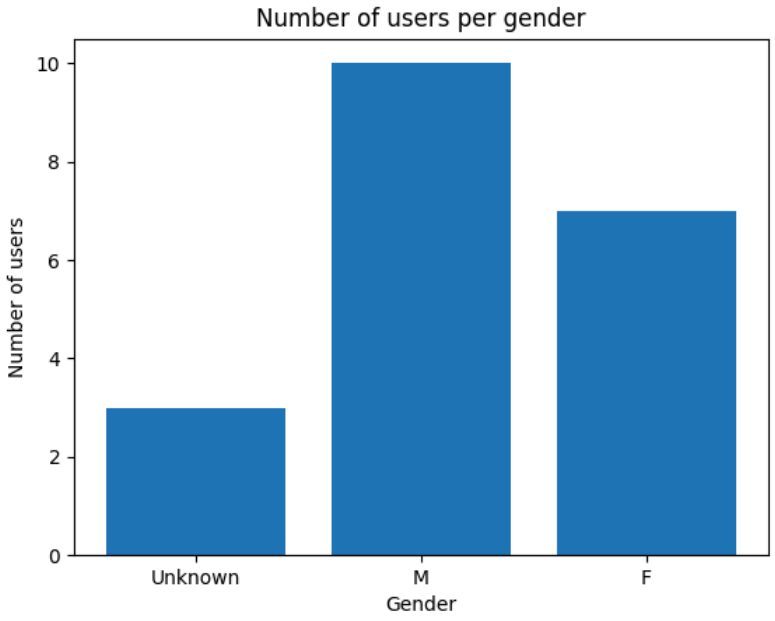
The above results are the results of the queries listed above above . Q1 is the answer to question 1, Q2 to question 2 and so on.

(e.) Graphically represent some attributes of your data and explain the conclusions you draw [data visualization lecture].

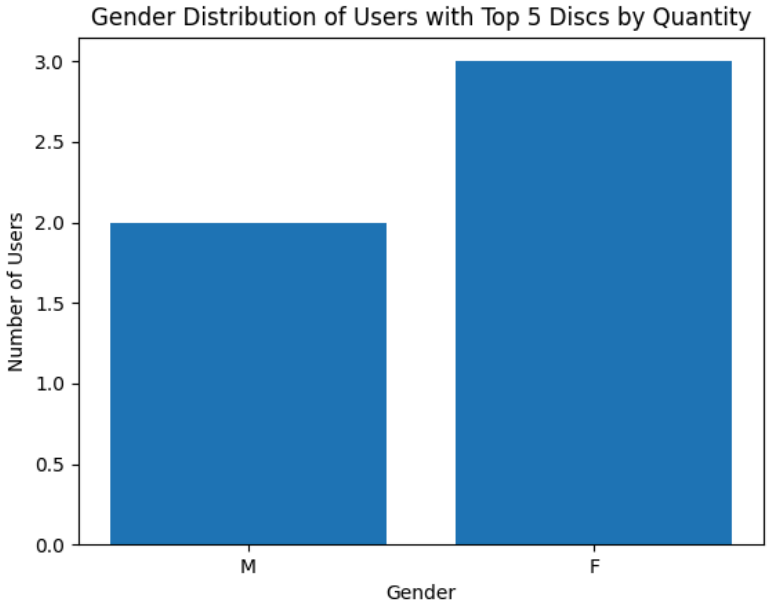
Running stats.py produced the following graphs:



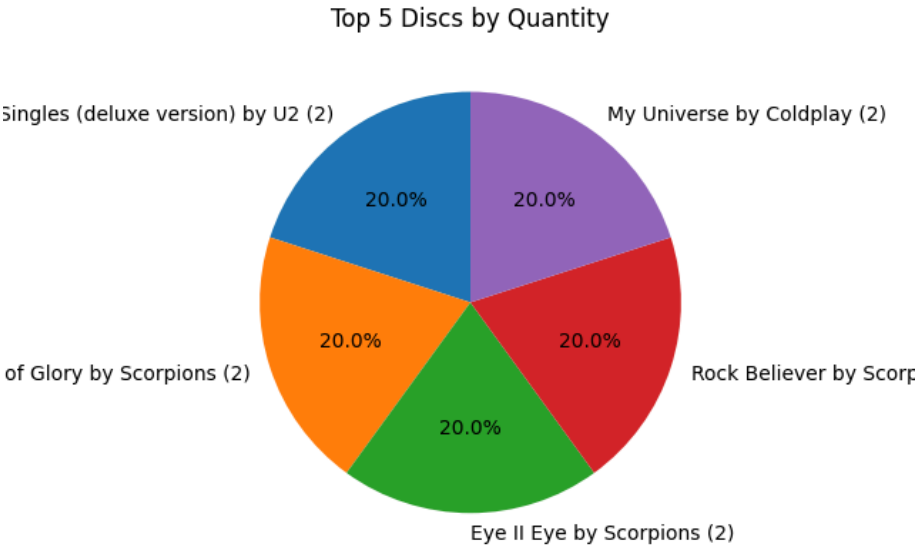
In the diagram above we see how users are distributed based on their age.



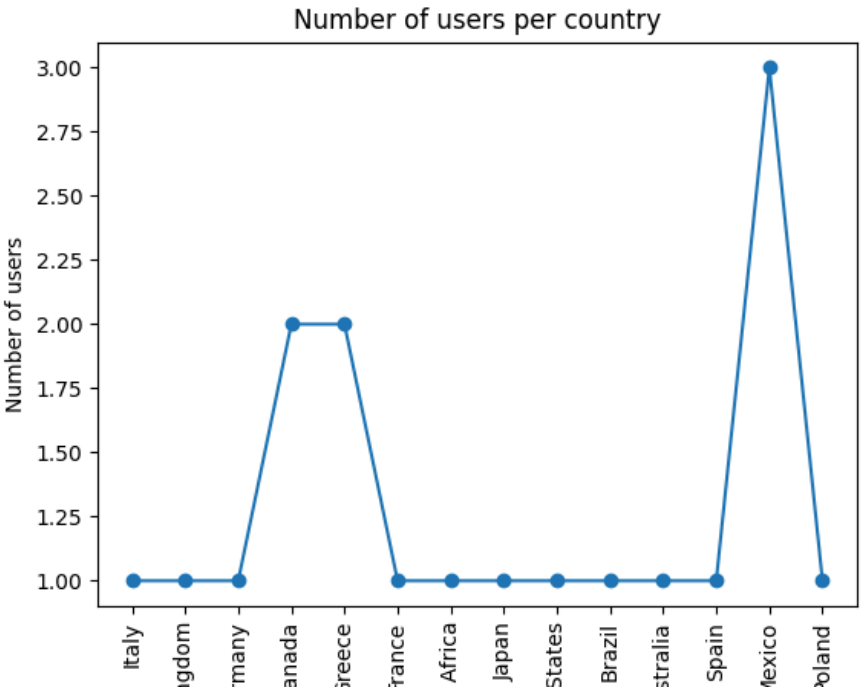
In the diagram above we see the distribution of users based on their gender. Specifically, there are 10 men, 7 women, while 3 more people have not declared their gender.



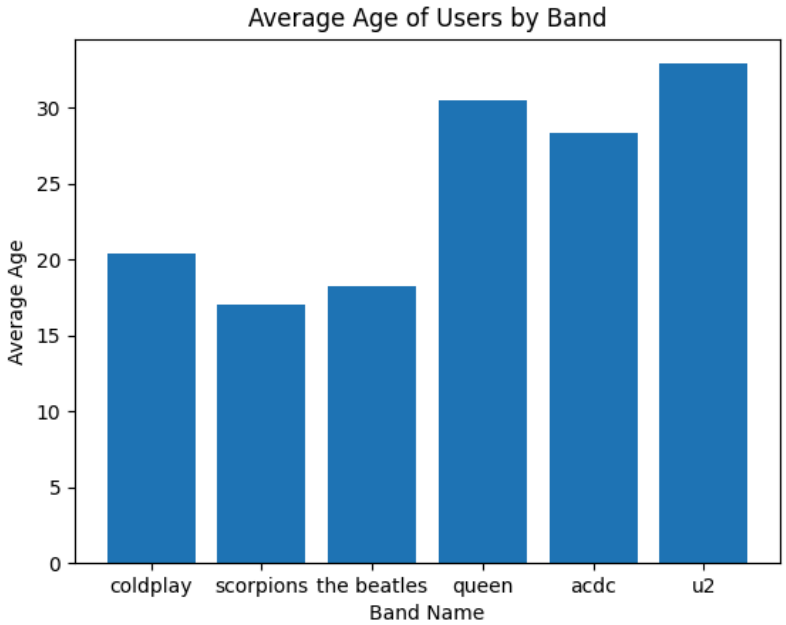
In the diagram above we see how the top 5 discs are distributed to users based on their gender. That is, women own 3 of them while men have 2.



In the chart above we see the top 5 discs that users own. Specifically, 20% of users own each of the disks shown in the "pie".

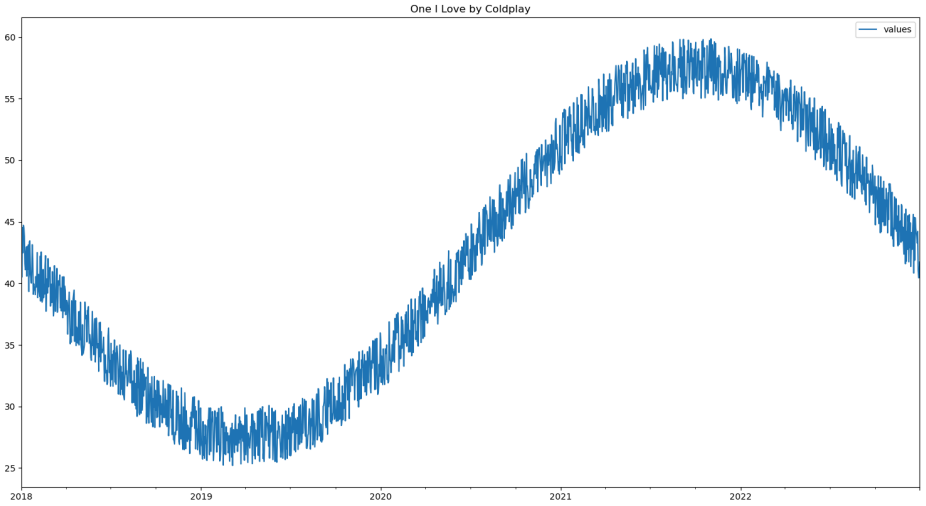


In the diagram above we see the countries of origin of the users. Specifically there is one user from Italy, UK, Germany, France, Japan, USA, Brazil, Australia, Spain, Poland and from Africa. At the same time, 2 users are from Greece, 2 from Canada and 3 from Mexico.



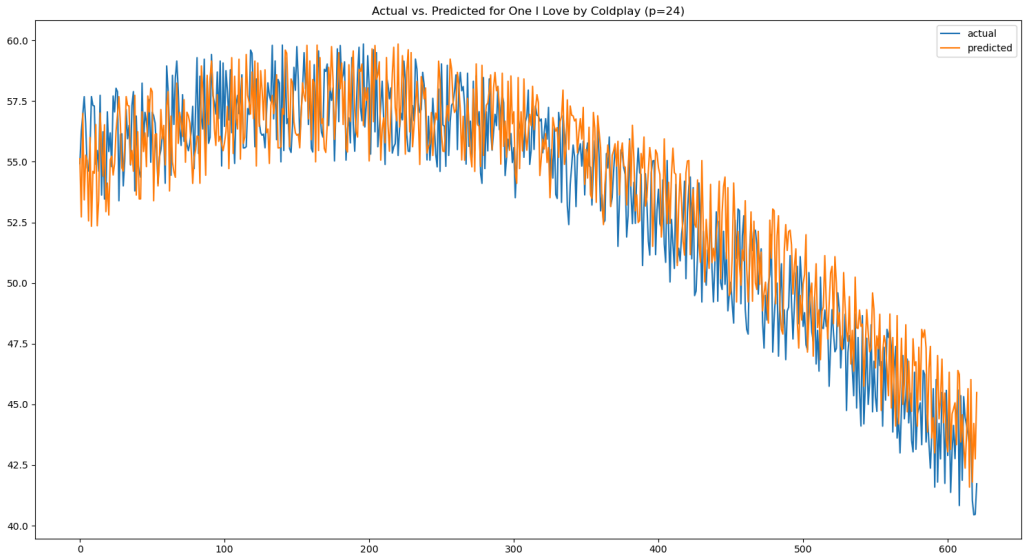
In the chart above we see the average age of users who listen to each band. So the average age of Coldplay fans is 20, Scorpions 17 and so on.

(f.) Let be the synthetic time series dataset[9] that has the lowest price a record is sold for each day, represent the time series and decompose it into trend, seasonality and residuals [lecture and lab time series].



In this diagram we see the variation of prices in previous years based on the givencsv file or web scraping from discogs, depending on the user's choice.

(g.) Split the time series data into 66% training and 34% testing, train an ARIMA model in the training split to predict the lowest price a record will sell for the next day, and evaluate its performance in the testing split [lecture and lab time series].



Graphic representation of difference between actual value and predicted. We notice that the predictions are almost the same as the reality.
Split testing was performed using the RMSE metric and resulted in RMSE = 2.233.

(h.) Using some graph mining metric on the community graph to suggest to a user a disk that he has not selected [graph mining lecture].

For the recommendation we used networkx. Specifically, after we built the barabasi model, we also built a new graph that has the usernames as nodes and the records of these users as data features. It is noteworthy that the edges of the new graph showing the friendships between users were built based on the edges of the barabasi model. So after we made this new graph, we took for each node (or user) its neighbors and counted the frequencies of each disc (that its neighbors - friends have) without counting the discs that the user already has.

(i.) Let be an array of users[10] who have some money and have expressed a measured desire to buy a series of records. The desire is expressed with a score from 1 to 5 for each disc and each disc has a monetary cost[11] . Use a genetic algorithm to select the discs to recommend users to buy in order to maximize the overall desirability of the discs, given the amount of money available to each user. Compare the results with a random selection of disks. [genetic algorithms lecture].

To implement the above query, we implemented our own genetic algorithm, which we will describe here:

In the file genetic.py we first have the function generate genome, which creates a binary string of 0 and 1. 1 means existence of the disk, while 0 the opposite. Then, generate population generates a population of genomes (we also pass the population number as a parameter).

Then we have the fitness function which checks for each genome if it has exceeded the user's value limit. If it has exceeded it, we return 0, otherwise we return the sum of the "desires" of all disks (that the genome has), as this is a good evaluation of each genome.

The selection pair then chooses a random combination for the generation's parents. It is worth noting that we have specified weights to make it more likely that the “strong” parents will be selected. We calculate these weights through the fitness function and are essentially the evaluation of each genome of the generation. If this list has only 0, the choices function with determination of the weights does not work, so we choose two random parents from the population.

Then, the crossover function "marries" the two parents and specifically produces 2 children, which have a (random part) of one parent and the other. If we split the first parent into a and b and the second into c d (randomly), then the two children will be of the form: ad the first and cb the second.

We also have the mutation function, which changes a bit of the genome (by default) to its complementary one with a probability of 0.5 (by default).

Additionally, the run evolution function is the one that produces the best genome by running for a predetermined number of generations. It is important to note that in the "evolution loop" we keep the two strongest parents in the next generation (elitism). Then we test for the rest**number**of the pairs and produce the strongest children using the functions mentioned above (extracting parents from the generation – which are most likely to be the best, then crossover the parents to produce 2 children and finally mutation of each child). We store the children of this process in the next generation list, which is the population of the next generation. This repetition takes place as many times as we have defined as the number of generations.

As soon as we reach the last generation, a reverse sort of the list of the population (of the generation) is done based on the fitness function, with the result that the strongest genome of all generations is in the first place of the list.

Finally with the appropriate queries in the base we get the available money of each user (to define the price limit), the user's desire to buy the disc (according to the "want" attribute of the "user_wants_discs" table) to define the " weight" of each disc (the bigger the better). We also get the most**recent**price of the disc (so that we know when we have exceeded the price limit) and the name and band of the disc. We have initialized these values before when creating the base (also, we have made sure that the user does not have the disks he wants). Having then taken into account that each bit of the genome indicates the absence (0) or the presence of the disk (1), we search for each user the appropriate combination of disks so that for the money he has he has the maximum "desire" of the discs it wants (the results of which discs we finally recommend from this knapsack problem are in the "user_rec_discs_knapsack" table).

It is also important to note that if the user has less money than the cheapest disk (than the ones he wants) then we do not need to run the genetic algorithm (since he cannot buy any disk) and so we do not recommend any disk. We do the opposite if the user has more or equal money than the sum of the prices of all the desired disks (then we return all the disks he wants). In this way we "save" actions that would lead us to the same result.

Here is a comparison of the result with a random selection of the user's desired disks (we simply created an "unevolved" genome and assigned 0 where the disk is not present and 1 where it is):

```
❯ python3 genetic.py
Genetic KnapSack Result:
User btonnesen6 with money 321.0 achieved want level 31 with cost 290.5559474213929.
=========
Random Result:
User btonnesen6 with money 321.0 achieved want level 41 with cost 456.58791737647454.
=========
```

We notice that the random genome far exceeded the user's price limit (it is invalid).

Let's run the comparison again:

```
❯ python3 genetic.py
Genetic KnapSack Result:
User btonnesen6 with money 321.0 achieved want level 31 with cost 290.5559474213929.
=========
Random Result:
User btonnesen6 with money 321.0 achieved want level 30 with cost 332.0639399101633.
=========
```

This time the rank of the options is worse than what we found with the genetic algorithm and the price is again higher than what the user has.

The conclusion is that the genetic algorithm produces much better results than the random algorithm used.

Flask Api

For the work we also prepared an api in flask, in which the user can retrieve base data from web scraping, questions with data features and other useful information. In some endpoints, basic authorization with the user's details (username & password) is required to access the data (eg authorization is required for the /discs endpoint and it returns the discs that the logged in user has). It is noteworthy that the passwords are encrypted at the base and the encryption is done based on the SECRET_KEY defined in the .env file. Here are usage instructions with the available api endpoints with what they return and if they need authorization. flask runs onlocalhost:5000 and you can try it in postman by putting the appropriate authorizations (if required).

Api Endpoint	Returns	Authorization
/discs	Returns the logs of the logged in user.	Yes
/bands	Returns the bands that the logged in user listens to.	Yes
/recommend	He returns them recommended discs to the logged in user.	Yes
/friends	You return them all friends of the logged in user (and their details, without the password).	Yes
/friends/{username}	You return the requested friend of the logged in user (and their details, without the password).	Yes
/discs/friends	Returns all of them disks of the logged in user's friends.	Yes

/discs/friends/{username}	Returns all of them disks of the request friend of the logged in user.	Yes
/bands/friends	Returns the bands that all friends of the logged in user are listening to.	Yes
/bands/friends/{username}	Returns the bands listened to by the requested friend of the logged in user.	Yes
/price/{disc_name}/history	Returns all values available from disk web scraping.	Yes
info/discs/{disc_name}	Returns the most its recent value requested disk, along with other information such as for the record band.	No
info/bands/{band_name}	Returns information about the requested band.	No
/stats/average/age/{band_name}	Returns the average age of a band's listeners (Q1).	Yes
/stats/average/discs	Returns the average number of disks that users have (Q6).	Yes
/stats/countries/mostmusic	Returns the countries and how many people listen to music from them (Q2).	Yes
/stats/bandcountry/{band_name}	Returns the origin of the most users listening to the requested band (Q3).	Yes
/stats/mostgender/{band_name}	Returns the gender most people have listeners of the requested band (Q4).	Yes
/stats/mostband	Returns the band with the longest listenership (Q5).	Yes
/stats/bands/heritage/{band_name}	Returns the origin of users listening to a band (and how many there are	Yes

	– Q7).	
/stats/discs/heritage/{disc_name}	Returns the origin of the users who have the requested disk (Q8).	Yes
/stats/discs/usercount/{disc_name}	Returns the number of users who have a disk (Q9).	Yes
/stats/discs/mostgender/{disc_name}	Returns the gender that listens to him the most requested disc (Q10).	Yes
/stats/bands/mostbands	Returns them bands that users listen to by country (along with the number of such users – Q11).	Yes
/stats/topdiscs/{x}	Returns a list of the top x disks (in quantity – Q12).	Yes

Important notes:

- If there are spaces in the name, replace them with '-'. For example, Coldplay's album “Ode To Deodorant” will be written “ode-to-deodorant”.

Web Scraping

For web scraping we used a combination of Selenium Frameworks and BeautifulSoup (which we explain in this[presentation]).

Github Repository:

https://github.com/manouslinard/music-recommender

Run Code:

To run the code (after the configuration of the .env file has been done correctly) you can run the following command (in the repository folder):

```
python3reset_db.py
```

This command fills the database with all the answers to the previous queries. To view the graphs, run the following command:

```
python3stats.py
```

By default this file "draws" the time series. To see other graphs or answers to data features questions, you can uncomment the desired questions in the main of stats.py.