MusicRec

A graph-based music recommendation system

Project for the Fundamentals of Artificial Intelligence course
University of Bari Aldo Moro,
Academic Year 2021/2022

Alessia Laforgia, Daniela Grassi
a.laforgia21@studenti.uniba.it, d.grassi6@studenti.uniba.it
742292, 745767

August 24, 2022

Contents

1 Introduction **3**

2 Requirements **3**
- 2.1 Prerequisites . . . 3
- 2.2 Goals . . . 3
- 2.3 Overview of the system . . . 3
- 2.4 Overview on components . . . 4

3 Conceptualization **4**
- 3.1 Entities . . . 5
- 3.2 Relationships . . . 6
- 3.3 Functions . . . 6
  - 3.3.1 Track . . . 6
  - 3.3.2 Artist . . . 8

4 Knowledge Base **8**
- 4.1 Graph Database and Neo4j . . . 9
- 4.2 Dataset and preprocessing . . . 9
- 4.3 Json Dataset to Neo4j . . . 11
- 4.4 Neo4j to Prolog Knowledge Base . . . 13

5 Recommendation **15**
- 5.1 Artist Recommendation . . . 16
- 5.2 Track Recommendation . . . 22

6 User Interface **23**
- 6.1 PySwip module . . . 23
- 6.2 Prolog Queries module . . . 23
- 6.3 Bridge between interface and queries module . . . 23
- 6.4 Interface module . . . 23

7 Overview of the interface **24**

8 Installation **31**
- 8.1 Wordnet Installation . . . 31
- 8.2 Manual installation . . . 31
- 8.3 Automatic Installation . . . 31

9 Conclusions and Future Developments **31**

1 Introduction

MusicRec is a content-based recommender system that, exploiting Prolog inference, is capable of providing advices about songs and musical artists based on insights about mood and genres preferences expressed by the user. A content-based recommendation system is a type of information filtering system that attempts to predict users' preferences and make suggestions based on users' previous likes. It uses item features to recommend other items similar to ones liked by the user, based on its previous actions or explicit feedback. The main purpose of the current work is to build a system that could exploit user sentiment, previous preferences, items features and their similarities to give suggestions about new artists, related albums or tracks to listen to. In particular, the knowledge of the domain that the system uses for reasoning is provided with a knowledge graph, which stores everything about tracks, their features, albums, artists and musical genres. The application can be accessed by the user through a dialogue with a graphical interface: the system will ask the user some questions and, after that, a first bunch of suggestions will be delivered to the user, who is capable of refining it, providing further information and explicit approval.

2 Requirements

In this section, the main prerequisites, goals of the project and application functioning will be illustrated.

2.1 Prerequisites

The system doesn't need any particular prerequisite in terms of computational effort or requirements to be satisfied before to run the application. It just needs Python and SWI Prolog to be installed on the machine in order to perform inference.

2.2 Goals

The main goal is to provide suggestions to the user basing on its explicit preferences, information about the mood and the favourite musical genres. On this basis, the most similar tracks or artists to the user preferences are returned exploiting similarities measures calculated among tracks features, artists, genres, also integrating WordNet.

2.3 Overview of the system

In this section, a first glance at how the system works will be given. The main components used by the system to give suggestions are:

- Mood of the user

- Preferences of genres and tracks
- Tracks features

MusicRec, as output, can provide suggestions of:

- Artists
- Tracks

After having acquired these pieces of information, the system will give the first set of suggestions, for which the user is required to give explicit preference. Once the preferred subset has been identified, tracks or artists will be suggested on the basis of the similarity between the subset and the other tracks or the other artists. The results having the highest similarity score will be the final output of the system. In calculating the similarity there are different measures taken into account, which will be explained in some sections. The system can also show some information about the albums containing the suggested tracks or published by the suggested artist: in the first case, for the suggested tracks the user will be able to display the album which the song comes from and in the second case, if the user is interested in knowing the albums published by a certain artist, it will be able to visualize the list of related albums.

2.4 Overview on components

The system is composed of different parts that guarantee the correctness of the workflow, from the loading of the knowledge base until the production of the final output. The needed components are:

- Knowledge graph containing the expertness of the domain
- Prolog rules and inference modules
- Interface dialogue module
- Connection module between backend and frontend

Every part will be explained in detail in the dedicated section.

3 Conceptualization

In this section the knowledge about the musical domain is formalized, identifying the entities of interest that make the system work properly in order to satisfy the requirements. In particular, the workflow starts from having a knowledge graph storing everything about our musical domain. Through the usage of Cypher queries, the knowledge base in Prolog is extracted from the graph and, using scripts, shaped in the form explained in the next subsections.

3.1 Entities

In this section, entities with their attributes are specified:
artist (ArtistName):

- ArtistName: Name of the artist. (string)

album (AlbumId, AlbumName):

- AlbumId: Unique ID of an album. (string)
- AlbumName: Name of the album. (string)

track (TrackId, TrackName):

- TrackId: Unique ID of a track. (string)
- TrackName: Name of the track. (string)

features(TrackId, Danceability, Energy, Speechiness, Acousticness,Instrumentalness, Liveness, Valence, Speed):

- Danceability: Describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. This feature can have the values *high danceable, medium danceable, low danceable*
- Energy: Represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. This feature can have the values *high energy, medium energy, low energy*
- Speechiness: This detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audiobook, poetry), the closer to 1.0 the attribute value. This feature has been discretized with these values *high speechiness, medium speechiness, low speechiness*
- Acousticness: A confidence measure from 0.0 to 1.0 of whether the track is acoustic. This feature has been discretized with these values *high acoustic, medium acoustic, low acoustic*
- Instrumentalness: Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”. This feature can have the values *high instrumental, medium instrumental, low instrumental*
- Liveness: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. This feature has been discretized with these values *live track, studio track*

- Valence: Describes the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry). This feature has been discretized with these values *high valence, medium valence, low valence*
- Tempo: The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration. This feature has been discretized with these values *fast, moderate, slow.*

genre(GenreName):

- GenreName: Name of the genre (string)

3.2 Relationships

artistgenres(Artistname, GenreName)

- This relation describes the genre of the artist. Every artist can have more than one genre.

album_contains(AlbumId, TrackId)

- This relation indicates which track is contained in an album. Every album contains more than one track

published_by(AlbumId, ArtistName)

- This relation describes which album is composed by which artist. Every artist published more than one album.

like(Genre, SubGenre)

- This relation puts in correlation the genre with its subgenres.

3.3 Functions

3.3.1 Track

In the following section, Prolog rules to obtain the track' suggestions will be described. In particular, to measure the similarity between the tracks Jaccard Similarity is used, and it's calculated on the set of features that describes each track. These rules are:
getFeaturesList(+TrackID, -ListOfFeatures)

Takes in input the track id and returns the list of features related to that track.

similarityByTrackFeatures(+TrackA, +TrackB, -Similarity)

Takes in input two track ids and returns the similarity between these tracks, calculated with Jaccard similarity. Jaccard coefficient measures similarity between finite sample sets, and it is defined as the size of the intersection divided by the size of the union of the sample sets:

$$Jaccard(U, V) = \frac{|U \cap V|}{|U \cup V|}$$

getTracksByFeatures(+N, +Dance, +Energy, + Valence, -Ntracks)

This function contributes to give the first track suggestion on the basis of the mood. Given in input the number of tracks that we want to suggest and a subset of features (Danceability, Energy and Valence) that most describe the mood, returns N track ids that share the same features as those given in input.

getTrackByGenre(+N, +Genre, -Ntracks)

This function contributes to give the first suggestion like the previous one. Since the tracks are not directly linked to the genre, there was the necessity to implement this function. *getTrackByGenre* takes in input the number of tracks that we want to have in output and a list of genres, then retrieves all the tracks related to those indicated genres, going through the artist and the album.

findMostSimilarTracks(+TracksIds, -TotalTracks, -Similarities)

Given a list of track ids that the user likes, this returns the list of all tracks excluding the input ones, and a list of list with all the corresponding Jaccard Similarities calculated between the features of the input tracks and all remaining tracks.

rankTrack(+Similarities, +Tracks, -OrderedTracks)

Given the tracks and related similarities with the tracks that the user liked, this returns the list of tracks, ordered according to decreasing similarity.

suggestTrack(+TracksIds, +N, -NTracks)

Given the track ids that the user likes, this function calls the *findMostSimilarTracks*, ranks the tracks given in output , takes the first N×2 tracks, computes a shuffle and than takes the first N. This passage is performed in order to have more randomization in the suggestion of the tracks.

retrieveAlbumByTrack(+TrackIds, -Album)

Given the tracks the user wants to know more about, this function returns the names of the albums the tracks belong to.

3.3.2 Artist

In the following section, Prolog rules to obtain the artist' suggestions will be described. In particular, to calculate the similarity between the liked artists and those to be suggested, a combined measure of similarity consisting of the summation between Jaccard and WordNet similarity calculated on the genres of each artist is used. For this purpose, a Prolog application [1] that aims to access the lexical database WordNet is exploited.

getWordSense(+Genre, -WordSenses)

Returns the wordsense that maximizes the PATH similarity measure calculated between Genre in input and the noun *music*. Path similarity ranges from 0 to 1 and counts the number of nodes along the shortest path between the senses in the 'is-a' hierarchies of WordNet. Since a longer path length indicate less relatedness, the relatedness value returned is the multiplicative inverse of the path length (distance) between the two concepts. If the two concepts are identical, then the distance between them is one; therefore, their relatedness is also 1, since for real numbers the multiplicative inverse is 1 divided by the number itself.

getSimilarity2Genre(+GenreA, +GenreB, -Similarity)

After having found the wordsense related to both GenreA and GenreB, this function calculates and returns the PATH similarity between GenreA and GenreB

getSimilarityListOfGenres(+GenreA, +GenreB, -AvgOfSimilarities)

Executes the Cartesian product using the function *getSimilarity2Genre* between the two input genre lists. At the end, it computes the average calculated on the output list.

getSuggestedArtists(+Artists, -Genres, -Similarities)

This function returns a list of list of all genres corresponding to all artists, excluded those in input, and the corresponding similarities. The similarities are calculated by summing the Jaccard similarity and the WordNet similarity, then normalizing the summation.

suggestArtist(+Artists, -Genres, -Similarities)

This function retrieves all the artists to suggest with their similarities, computes an ordering, takes the first N $\times$ 2 artists, computes a shuffle and then takes the first N. This passage is performed in order to have more randomization in the suggestion of the artists.

4 Knowledge Base

To construct the Knowledge Base Neo4J has been used. Our use case considers the knowledge already stored on Neo4j. Since we did not find any music graph database, we decided to find a music database in Json format, upload this on Neo4j via Cypher queries and then build the Prolog knowledge base from this starting point.

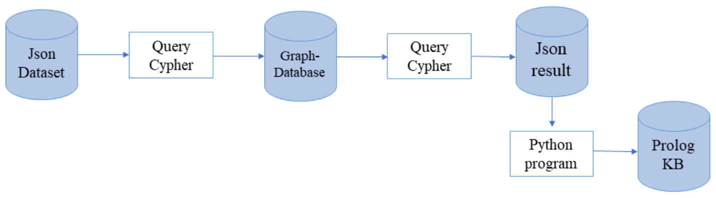


Figure 1: Process of construction the knowledge base

4.1 Graph Database and Neo4j

A graph database is a set of vertices and edges. In particular a graph database can be pictured as an arbitrary set of objects connected by one or more kinds of relationships. The novelty with respect to the other kind of database (like NoSQL database) is that relationships are first class citizen and this allow to build arbitrarily sophisticated models that map closely to our problem domain.
In this project we chose to use Neo4j, that is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend. Moreover, we chose Neo4j because of its well-written documentation and of its query language, Cypher, that is similar to SQL, so it's easier to understand and use.

4.2 Dataset and preprocessing

To populate our Graph Database we selected a dataset from Kaggle [1] and we preprocessed it in order to upload it on Neo4j. The original dataset comprises three main files: **artists.json, albums.json, and tracks.json**. These are collected with the help of Spotify API and are organized as follows:

- artists.json has the following main keys:
  - artistname which is the name of the artist and it is unique
  - artistgenres which is the list of musical genres associated with the artist
  - albumids which is the list of id albums published by the artist
- albums.json has the following main keys:
  - albumid which is a unique id to identify the album
  - trackids which is a list of unique ids of the tracks that are contained in the album

[1] https://www.kaggle.com/datasets/fferri/spotifydata

  - albumname which is the name of the album
  - artistname which is the name of the artist who published the album

- tracks.json has the following main keys:
  - trackid which is a unique id to identify the track
  - trackname which represents the name of the track
  - features which is a list containing the numeric features described in 3.1
  - albumname which is the name of the album that contains the track
  - artistname which is the name of the artist that published the track

Before populating the database, we preprocessed the dataset, in order to perform a first cleaning. In particular, we removed the duplicates among the different files and performed a discretization of the features. For simplicity, we have grouped the features that share the same range in a table 1 . In particular, the rows of the table correspond to the features while the columns to the range whose value we have replaced. For example, if the value of the feature Danceability was less than 0.33, then this values has been replaced with "low_danceable".

	Range	**x <= 0.33**	**0.66 >x >0.33**	**x >= 0.66**
Danceability	[0,1]	low_danceable	medium_danceable	high_danceable
Acousticness	[0,1]	low_acoustic	medium_acoustic	high_acoustic
Energy	[0,1]	low_energy	medium_energy	high_energy
Instrumentalness	[0,1]	low_instrumental	medium_instrumental	high_instrumental
Speechiness	[0,1]	low_speechiness	medium_speechiness	high_speechiness
Valence	[0,1]	low_valence	medium_valence	high_valence

Table 1: Visualization of the discretization

The remaining features have been discretized as follows:

- **Tempo** is the overall estimated tempo of a track in beats per minute (BPM), so following the *basic tempo markings*[2] given by Wikipedia we have replaced the values less than 76 with the word *slow*, the values greater than 168 with the word *fast* and the remaining values with *moderate*
- **Loudness** values have a range between -60 and 0 db. So for values less than -30 we put *studio_track* otherwise *live_track*

[2]https://en.wikipedia.org/wiki/Tempo

4.3 Json Dataset to Neo4j

In Neo4j, nodes describe entities of a domain and can have zero or more labels to define what kind of nodes they are. Instead, a relationship describes how a connection between a source node and a target node are related. Each node can have properties that are key-value pairs that are used for storing data on nodes and relationships.

The Database consists of four nodes with four different labels: **Artist**, **Album**, **Genre** and **Track**.

- The node **Artist** represents the artist and stores one property key that is the name of the artist.
- The node **Album** represents the album and stores two property keys that is the name of the album and the album id.
- The node **Genre** represents the genre and stores one property key that is the name of the genre.
- The node **Track** represents the track and stores two property key that is the name of the track and the track id.

There are three relationship: **ALBUM_OF**, **CONTAINS**, and **GENRES_OF**.

- The relationship **ALBUM_OF** connects artists and their albums.
- The relationship **CONTAINS** connects albums to tracks contained into
- The relationship **GENRES_OF** connects artists to their genres.

To populate the database, graph query language Cypher was necessary. In the following section, the queries and examples of output used to populate the database are reported:

```
CALL apoc.load.json("file:/MusicNet/albums.json") YIELD value
MERGE (a:Album {albumid: value.albumid, albumname:
value.albumname})
WITH a, value
UNWIND value.artistnames AS artistname
MERGE (c:Artist {artistname: artistname})
MERGE (a)-[:ALBUM_OF]->(c)
WITH a, value
UNWIND value.trackids AS tracks
MERGE (p:Track {trackids: tracks})
MERGE (a)-[:CONTAINS]->(p);
```

This query populates the database checking albums.json and track.json, then uploads the nodes Album and Track, and the relationships ALBUM_OF and CONTAINS.

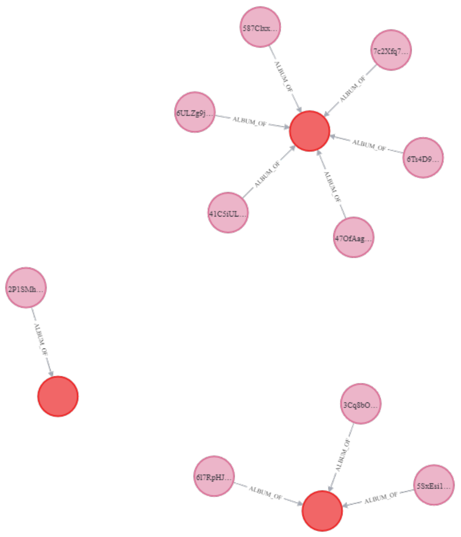


Figure 2: Node Artist (red) connected with node Album (Pink) through the relationship ALBUM_OF

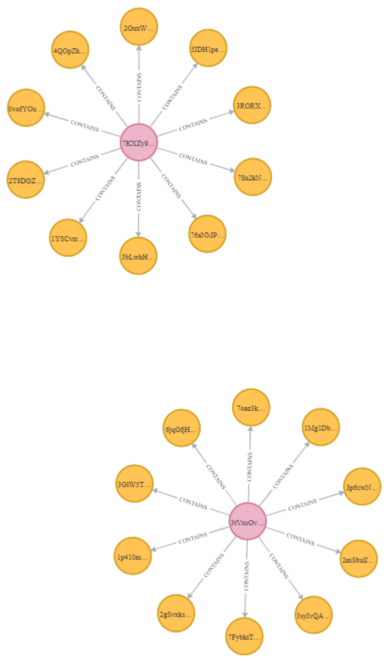


Figure 3: Node Album (pink) connected with node Track (yellow) through the relationship CONTAINS

```
CALL apoc.load.json("file:/MusicNet/artists.json") YIELD value
MERGE (a:Artist {artistname: value.artistname})
WITH a, value
UNWIND value.artistgenres AS genres
```

```
MERGE (c:Genres {artistgenres: genres})
MERGE (a)-[:GENRES_OF]->(c);
```

This query populates the database checking artists.json, then uploads the node Artist and Genre, and the relationship GENRES_OF.

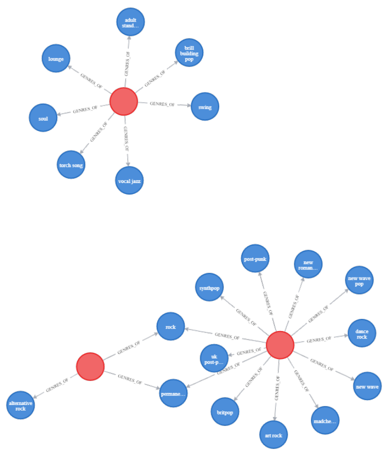


Figure 4: Node Artist (red) connected with node Genre (blue) through the relationship GENRE_OF

Another query has been performed in order to add the features that described each track.

```
CALL apoc.load.json("file:/MusicNet/tracks.json") YIELD value
UNWIND value.trackid as t
UNWIND value.trackname as name
MATCH (n:Track {trackids: t})
set n.trackname = name
set n.features = value.features
```

4.4 Neo4j to Prolog Knowledge Base

To build the knowledge base, all nodes and relationships were downloaded via query Cypher in a Json format. The Json files are seven: four for the nodes and three for the relationships. These files were processed with a Python script in order to construct the Prolog knowledge base. In the following section, there are descriptions of all knowledge base Prolog files.

artist.pl

Contains information about artists. The artist name is unique.

```
artist("queen").
```

album.pl

Contains information about albums, which are the id and the name.

```
album("6i6folBtxKV28WX3msQ4FE", " bohemian rhapsody (the original
↪  soundtrack)").
```

track.pl

Contains information about tracks, which are the id and the name.

```
track("1F9NVicWfNQA5ki8WmEtk8", "crazy little thing called love").
```

features.pl

Contains the information about the features that describe the tracks. Each rule is composed by the id track, its danceability, its energy, its speechiness, its acousticness, its instrumentalness, its loudness, its valence and its tempo.

```
features("1F9NVicWfNQA5ki8WmEtk8", "medium_danceable", "high_energy",
↪  "low_speechiness", "high_acoustic", "low_instrumental",
↪  "studio_track", "high_valence", "moderate").
```

genre.pl

Contains information about genres related to the artists

```
genres("art rock").
```

likes.pl

Describes the relation between genres and their sub-genres. This file was not extracted from the knowledge base trivially but it has been built differently, with a study on the various genres stored in the knowledge base. In fact, the total number of genres is more than 900 and for this reason is necessary to establish a relation between general genres and their subgenres. Thanks to external sources, more insights about genres have been obtained and the genres classified. Through a Python script the Prolog knowledge base have been obtained, establishing the relations.

```
like("hip hop", "german hip hop").
like("hip hop", "atl hip hop").
```

published_by.pl

Describes the relation between the albums and the artist who released them.

```
published_by("6i6folBtxKV28WX3msQ4FE", "queen").
```

album_contains.pl

Describes the relation between the album and the songs it contains.

```
album_contains("3tVxsOv2aTeaF3zsuWsd3e","2gSvxks9MP9mtVhOo51q9n").
```

artist_genres.pl

Describes the relation between the artist and its genres.

```
artistgenres("queen", "glam rock").
```

5 Recommendation

As mentioned in the section 1, the system has two levels of recommendation. In the first level of recommendation the user is asked to provide information about its mood and the genres it likes in order to skim the songs and understand what are the tastes of the user. The information concerning the mood are coded in danceability, energy and valence, and these together with the genres allow the system to show the first tracks that the user should select to have the second recommendation. All the rules that allow the recommendation are written in Prolog.
getTracksByFeatures and *getTrackByGenre* are the two rules that retrieve the first tracks that allow the second recommendation.

```
getTracksByFeatures(N, Dance, Energy, Valence, NTracks) :-
    findall(TrackId, (features(TrackId, Dance, Energy, _, _, _,
    ↪  _,Valence, _)), Tracks),
    length(Tracks, N1),
    (
    N1 > N
    ->
    random_permutation(Tracks, TracksPer),
    take(TracksPer, N, NTracks)
    ;
    random_permutation(Tracks, NTracks)
    ).
```

This rule finds all the tracks whose features match the input features, performs a random perturbation on these tracks and takes the first N tracks. Moreover, if we suppose that the tracks matching the input features are not N and that N is greater than all tracks found, then the rule returns all the available tracks with that feature pattern.

```
getTrackByGenre(N, Genre, NTrack) :-
    getSimilarGenre(Genre, ListGenre),
    flatten(ListGenre, FlattenGenre),
    getArtistByGenres(FlattenGenre, ListArtist),
    flatten(ListArtist, FlattenA),
    list_to_set(FlattenA, Artist),
    getAlbumByArtist(Artist, ListAlbum),
    flatten(ListAlbum, FlattenAlbum),
    getTrackByAlbum(FlattenAlbum, ListTrack),
    flatten(ListTrack, FlattenTrack),
    random_permutation(FlattenTrack, Track),
    take(Track, N, NTrack).
```

This rule retrieves all tracks that are related to the list of Genres in input. Since the genres and tracks are not directly related this rule first finds all the sub-genres related to the input genre, then finds all the artists who match the found genres, then all the albums of these artists and finally the tracks through the albums found.

To give the first suggestion, the combination of the two rules is used. Specifically, the user receives five suggestions with the first rule and five with the second, and this is done to increase the variety of the suggested songs. Then it is up to the user to select which of the suggested tracks he knows or likes so that the second recommendation can be made.

In the second level of recommendations, the system provides two types of recommendations: Artist and Tracks.

5.1 Artist Recommendation

The Artist Recommendation is based on a combination of two kinds of similarity: WordNet and Jaccard both computed on the artist's genres. All the rules that compute the artist recommendation can be found in the module **artistsuggest.pl**

WordNet Similarity

WordNet is a lexical English language database and, specifically, resembles a thesaurus in what it groups words together based on their meanings. WordNet stores words of four syntactic categories: nouns, verbs, adjectives, and adverbs. These words are grouped into sets of synonyms called synsets. Each word of a synset have the same meaning in a determined context and they represent a concept (or word sense). To use the WordNet similarity, the recommendation system loads and incorporates a library of Prolog programs implemented by Juliàn-Iranzo and Sàenz-Pérez, able to retrieve informations from WordNet. [1]. In particular in this system, the similarity is calculated on the words concerning the artist's genres, so for example between the word *rock* and the word *pop*. The similarity function of the integrated modules [1] requires a specific syntax wich is

Word[:SS_type[:Sense_num]]

where:

- SS_type is a one character code indicating the synset type: (n NOUN, v VERB, a ADJECTIVE, s ADJECTIVE SATELLITE, r ADVERB)

- Sense_num is the sense number of the word that identify the word's meaning among the others in the synset.

In the similarity computation SS_type is always set to n while for the Sense_num a specific rule has been implemented.

```
getWordSense(Genre, WordSenses) :-
    findall(Rank, (wn_path(music:n:4, Genre:n:Ind, Rank)), SimList),
    findall(Ind, (max_list(Sim, _, Ind)), WordSenses).
```

This rule finds the Sense_num of the genre such that it maximizes the similarity with the wordsense music. Specifically, it calculates the similarity between the noun *music* and all possible Sens_num of the genre given in input. Then, the rule returns the Sens_num associated with the greater similarity. The Sens_num of the noun *music* is set to 4 because indicates the meaning of music as "(music) the sounds produced by singers or musical instruments (or reproductions of such sounds)" among all the other meanings of the music word. Once the Sens_num of the genres has been obtained, it is possible to calculate the similarity between two genres.

```
getSimilarity2Genre(GenreA, GenreB, Sim) :-
    getWordSense(GenreA, [WordSenseA|_]),
    getWordSense(GenreB, [WordSenseB|_]),
    wn_path(GenreA:n:WordSenseA, GenreB:n:WordSenseB, Sim), !.
```

This rule calculates the similarity between two genres after retrieving the Sens_num. The similarity is calculated with *wn_path* that implements the PATH similarity measure. The path similarity calculates the shortest path distance between the two Synsets and their common hypernym:

$$sim_{PATH}(c1, c2) = 1/len(c1, c2)$$

where

$$len(c1, c2) = (depth(c1) - LCS\_depth) + (depth(c2) - LCS\_depth) + 1$$

and LCS is the least common subsumer off two concepts, which is the most specific concept they share as an ancestor. The similarity calculated on two genres was not enough because each artist is associated with several genres, so to deal with this problem the following rules have been implemented.

```
% Takes two list of genres of two different artists and returns the
↪  similarity
getSimilarityGenres(GenreA, GenreB, AvgOfSimilarities) :-
     findall(Sim,(member(X,GenreA),member(Y,GenreB),
     getSimilarity2Genre(X, Y, Sim)), ListOfSimilarities),
     avg(ListOfSimilarities, AvgOfSimilarities).
```

```
% Calculate the similarity between the input Artist and all the others
↪   artists
calculateArtistWSimilarity(GenresA, [GenreB], [WSimilarity]) :- !,
    getSimilarityGenres(GenresA, GenreB, WSimilarity).
calculateArtistWSimilarity(GenresA, [GenreB|GenreT],
↪   [WSimilarity|SimilarityT]) :-
    getSimilarityGenres(GenresA, GenreB, WSimilarity),
    calculateArtistWSimilarity(GenresA,  GenreT, SimilarityT).
```

The first rule computes the PATH similarity of each pair of genres belonging to the two lists given in input. For example, if GenreA = [“Pop”, “Rock”] and GenreB = [“Pop”, “Folk”, “Metal”], then the rule computes [SimPath(“Pop”, “Pop”), SimPath(“Pop”, “Folk”), SimPath(“Pop”, “Metal”), SimPath(“Rock”, “Pop”), SimPath(“Rock”, “Folk”), SimPath(“Rock”, “Metal”)], and computing the similarity (SimPath) the output is: [1,0.11,0.09,0.09,0.11,0.2]. At the end, to have one single measure of similarity the average is computed.

Jaccard Similarity

The second measure used to perform the similarity is Jaccard similarity, calculated on the set of genres of two artists. This choice is due to the fact that some sub-genres are not listed in the knowledge base given by WordNet and these case the similarity calculated with the process described above return always zero.

Final Similarity Measure

The final similarity measure between two artists is computed with the sum between WordNet similarity and Jaccard similarity. Computing the addition, the artists related to genres that are in the WordNet’s knowledge base will always be favored because the computation is on two non-zero quantities. For this reason a *minmax normalization* is computed on the set of similarities with the following formula:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

The final similarity measure is computed between each input artist and all the artists in the Knowledge Base. In the Figure 6 is summarized the process of calculating the similarity between an artist and all the others:

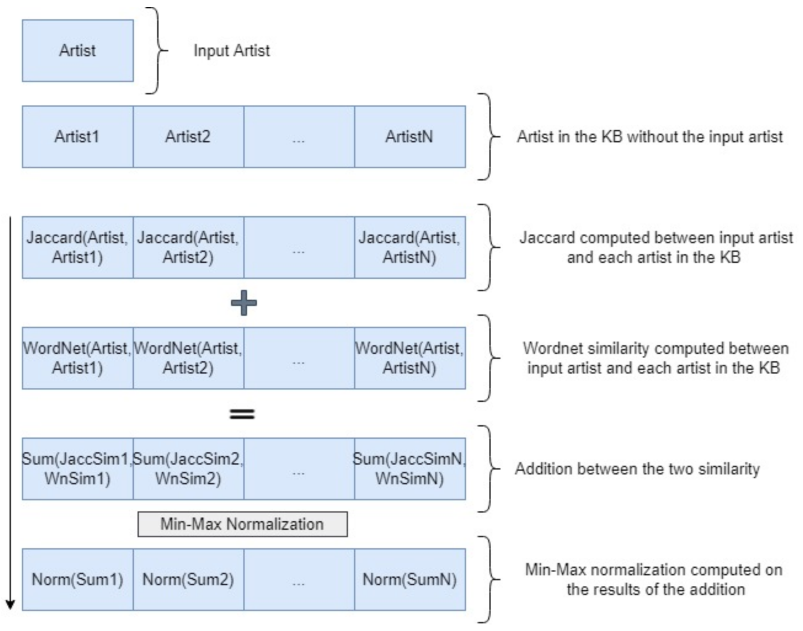


Figure 5: Similarity Computation Flow

Suppose there are N+1 artists in the Knowledge Base and one artist in input, the system computes the Jaccard similarities and then the WordNet similarities between the input artist's genres and the remaining N artists' genres, obtaining two lists of N similarities. Then the two lists are summed and the min-max normalization is computed. In the following are reported the Prolog rules that implement the process describe above.

```
% Computation of the similarity with Jaccard
calculateArtistJSimilarity(GenresA, [GenreB], [JSimilarity]) :- !,
    jaccard(GenresA, GenreB, JSimilarity).

calculateArtistJSimilarity(GenresA, [GenreB|GenreT],
↪   [JSimilarity|SimilarityT]) :-
    jaccard(GenresA, GenreB, JSimilarity),
    calculateArtistJSimilarity(GenresA,  GenreT, SimilarityT).

% Computation of the similarity with WordNet
 calculateArtistWSimilarity(GenresA, [GenreB], [WSimilarity]) :- !,
    getSimilarityGenres(GenresA, GenreB, WSimilarity).

calculateArtistWSimilarity(GenresA, [GenreB|GenreT],
↪   [WSimilarity|SimilarityT]) :-
    getSimilarityGenres(GenresA, GenreB, WSimilarity),
    calculateArtistWSimilarity(GenresA,  GenreT, SimilarityT).
```

```
%Computation of the final similarity measure
getSuggestedArtistAggregate([Artist], GenreResult, [NormSimilarities])
↪  :-
    !,
    %Retrieve all the Genre of the Artist
    findall(GenreA, (artistgenres(Artist, GenreA)), ListGenresA),
    %Jaccard Similarity
    calculateArtistJSimilarity(ListGenresA, GenreResult, JSimilarity,
    %Wordnet similarity
    calculateArtistWSimilarity(ListGenresA, GenreResult, WSimilarity),
    % Addition of the two
    sum(JSimilarity, WSimilarity, Similarities),
    % Compute minmax normalization to return the range between [0,1]
    minmax_normalization(Similarities, NormSimilarities).

getSuggestedArtistAggregate([Artist|TArtist], GenreResult,
↪  [NormSimilarities|TSimilarities]) :-
    %Retrieve all the Genre of the Artist
    findall(GenreA, (artistgenres(Artist, GenreA)), ListGenresA),
    %Jaccard Similarity
    calculateArtistJSimilarity(ListGenresA, GenreResult, JSimilarity),
    %Wordnet similarity
    calculateArtistWSimilarity(ListGenresA, GenreResult, WSimilarity),
    %Addition of the two
    sum(JSimilarity, WSimilarity, Similarities),
    %Compute minmax normalization to return the range between [0,1]
    minmax_normalization(Similarities, NormSimilarities),
    getSuggestedArtistAggregate(TArtist, GenreResult, TSimilarities).
```

Considering the previous example, if the input artists are K, the rule computes K different list containing the similarity for each artist with all the artists in the Knowledge Base. To take into account all the artists in input a data aggregation is needed. The aggregation formula used is the addition between all the lists, i.e. the i-th elements of each list are added together obtaining the final list of similarities.

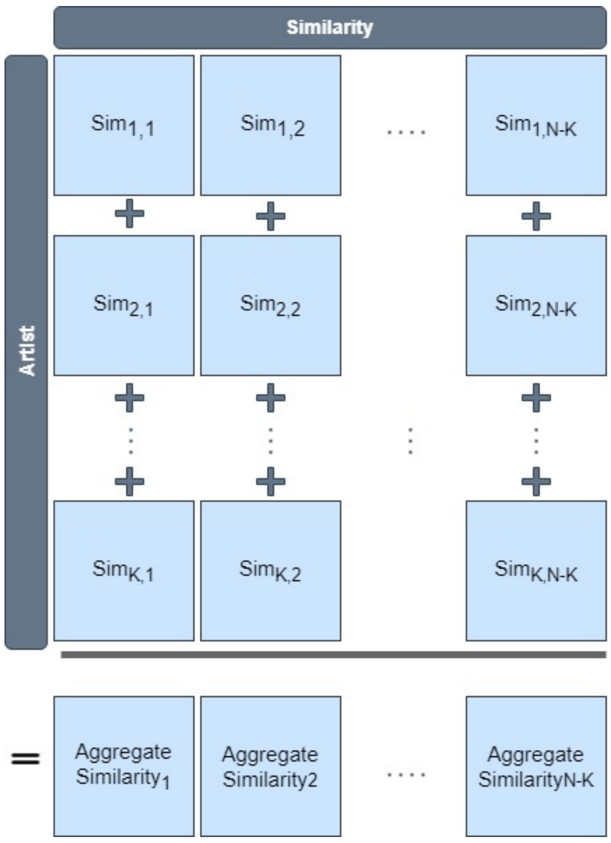


Figure 6: Data Aggregation to obtain the final computation of the similarities

The aggregate list is sorted, 20 artists are taken, a shuffle on the list is computed, then the first 10 artists are taken as output.

```
suggestArtist(Tracks, N, NArtists) :-
    retrieveArtistsByID(Tracks, Artist),
    %Find all artists excluding those in input
    getAllArtistsAndGenreExceptSome(Artist, Artists, GenreResult),
    %Get artists with similarities
    getSuggestedArtistAggregate(Artist,GenreResult, Similarities),
    %Vertical sum of the lists of similarities
    sum_list(Similarities, SumSimilarities),
    %Sorting
    rankArtist(SumSimilarities, Artists, OrderedArtist),
    N1 is N*2,
    take(OrderedArtist, N1, N1Artists),
    random_permutation(N1Artists, ArtistPer),
    take(ArtistPer, N, NArtists).
```

5.2 Track Recommendation

The suggestion of the tracks is based on the feature related to each track, specifically the similarity used is that of Jaccard, described in 5. When the user provide an input regarding what tracks he likes, the system computes the Jaccard similarity on the sets of features between the liked tracks and all the tracks in the Knowledge Base. Then, the Tracks are sorted according to their decreasing similarities, from the ordered list 20 tracks are taken, a shuffle is computed, and finally 10 tracks are returned.The rules regarding the suggestion of the tracks can be found in **tracksuggest.pl** module

```
findMostSimilarTrackAggregate([TrackId],Tracks, [Similarity]) :-
    !,
    trackSimilarity(TrackId, Tracks, Similarity).

findMostSimilarTrackAggregate([TrackId|TracksIds],
↪   Tracks,[Similarity|TSimilarity]) :-
    trackSimilarity(TrackId, Tracks, Similarity),
    findMostSimilarTrackAggregate(TracksIds, Tracks, TSimilarity).

% restituisce tutte le tracce con la loro similarità alla traccia data
↪   in input
trackSimilarity(TrackIdA, [TrackIdB], [Sim]) :-
    similarityByTrackFeatures(TrackIdA,TrackIdB,Sim), !.

trackSimilarity(TrackIdA, [TrackIdB|T], [Sim|SimT]) :-
    similarityByTrackFeatures(TrackIdA,TrackIdB,Sim),
    trackSimilarity(TrackIdA, T, SimT).

suggestTrack(TrackIds, N, NTracks) :-
    getAllTracksExceptSome(TrackIds, TracksTotal),
    findMostSimilarTrackAggregate(TrackIds,TracksTotal, Similarities),
    sum_list(Similarities, SumSimilarities),
    rankTrack(SumSimilarities, TracksTotal, OrderedTracks),
    N1 is N*2,
    take(OrderedTracks, N1, N1Tracks), % Take the first n*2 most similar
    ↪   tracks
    random_permutation(N1Tracks, TracksPer), % compute a shuffle on the
    ↪   n*2 most similar tracks
    take(TracksPer, N, NTracks).
```

6 User Interface

This system is provided with a custom graphical interface developed with a Python framework called Streamlit. It is an open-source framework used to develop interactive user interfaces, often as support to expert modules. Also, to allow exchange of information between Prolog and Python, a library called PySwip is used. It's a Python - SWI-Prolog bridge enabling to query SWI-Prolog directly in Python programs. In fact, the whole application is deployed as a web-app, with Streamlit guaranteeing the communication between the user and the system, PySwip allowing the dialogue between Python and Prolog, and Prolog as backend.

6.1 PySwip module

The PySwip module is called *kbAccess.py* in the App folder, together with the interface files. This file specifies defines a class called PrologInterfaces, that involves the Prolog path and the Prolog extension of the files to consider as knowledge base and function modules. Also, it comprehends some built-in functions to load the rules and run the queries.

6.2 Prolog Queries module

The module containing Prolog queries is called *queries.py.* It defines different functions (one for each query) to perform on the knowledge base. For each of these functions, an object of the type PrologInterface is instantiated, the needed knowledge bases are loaded and the queries actually performed.

6.3 Bridge between interface and queries module

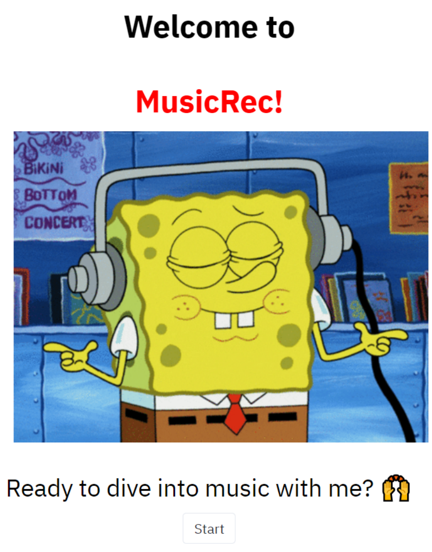
Utilities.py is a module that manipulates data coming from the user interface, processes it and send it to the queries module for interacting with the backend and getting an output. It's written in Python and it helps in managing the input, through the functions defined in it. Among these, in addition to manipulating the input, there is *discretization* that translates the moods options selected by the user in track features useful to research suggestions.

6.4 Interface module

The whole interface functioning is implemented in *interface.py*, as the Streamlit workflow needs. The user dialogue consists of a set of question and answers between the user and the system, where the first is asked information about the mood, the liked musical genres and a preliminar feedback. To correctly work, the interface needs an additional module called *SessionState.py* that allows to store variables in a permanent way during the same user session, since in Streamlit every piece of form is independent. The workflow is organized in the following way:

- The user is welcomed by the system by showing the main page.
- The system asks some questions about user mood. The questions are: *'Are you happy?'*, *'Are you energic?'* and *'Are you in the mood for dancing?'*.
- The system asks about favourite musical genres of the user. The main genres identified are:
  hip hop, pop, jazz, rock, country, rap, soul, folk, classical, metal, funky, indie, house, punk, electronic, reggae, latin, songwriter, children, soundtrack, relax
- A first bunch of suggestions is generated and the user is asked to express explicit preference.
- The user is asked if it would like suggestions about Artists or Tracks.
- After the selection, the refined bunch of suggestions is performed, according to all the information the user has given to the system and the user choice.
- For both choices (Artists or Tracks), the user can respectively get additional information, in the first case about the album published by the chosen artist, and in the second case about the album the chosen track belongs to.

7 Overview of the interface



Welcome to

MusicRec!

Ready to dive into music with me? 🙌

Figure 7: Main page

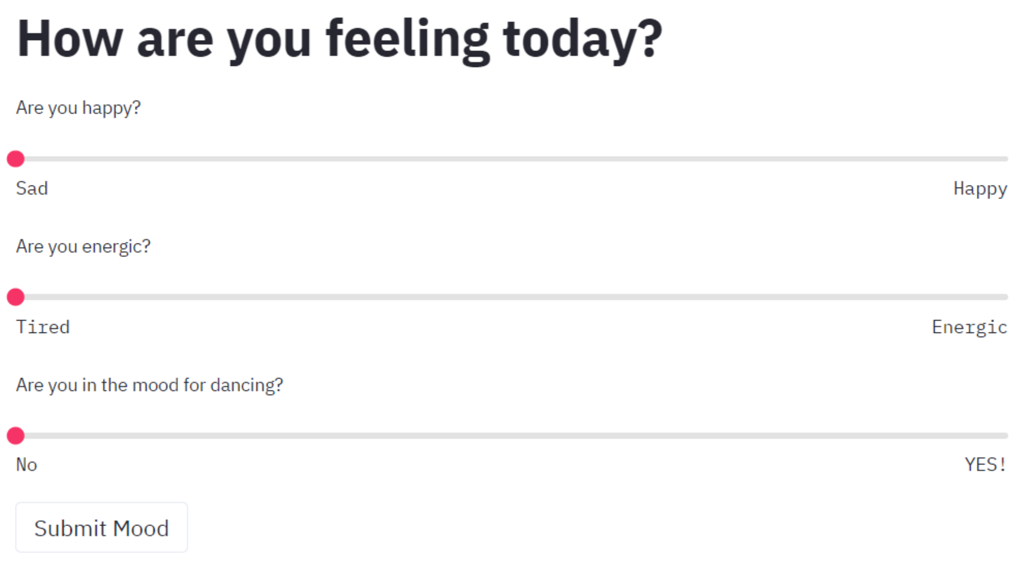


Figure 8: Mood form

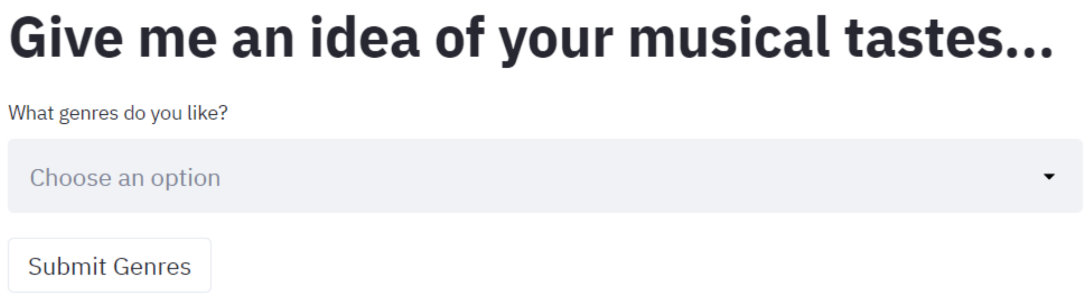


Figure 9: Genre form

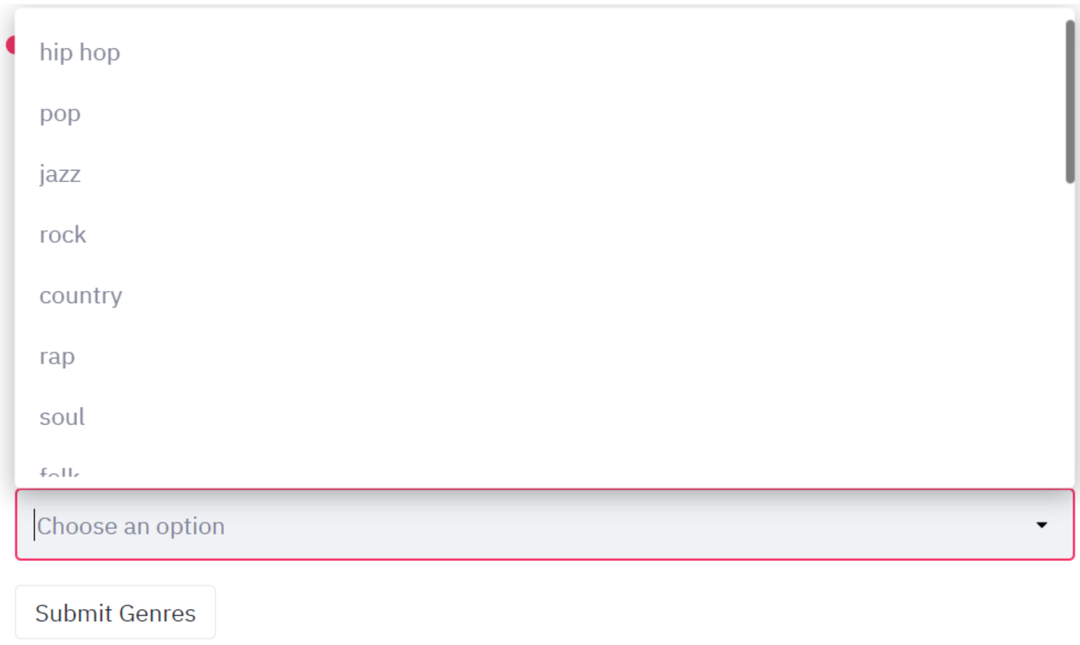
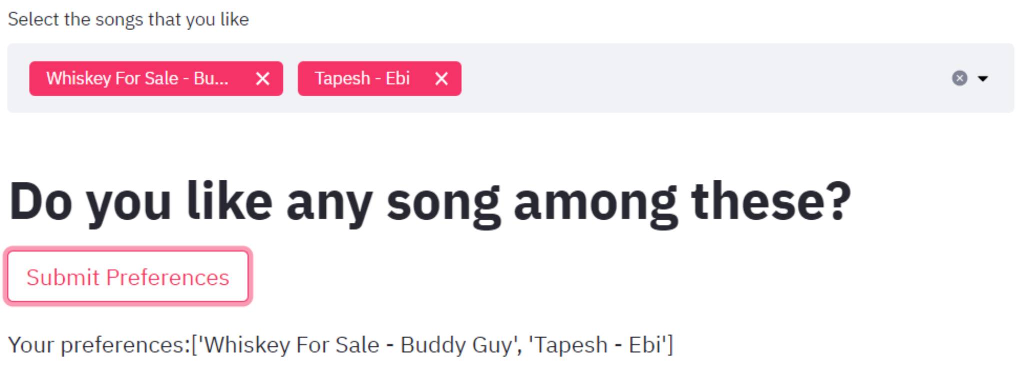
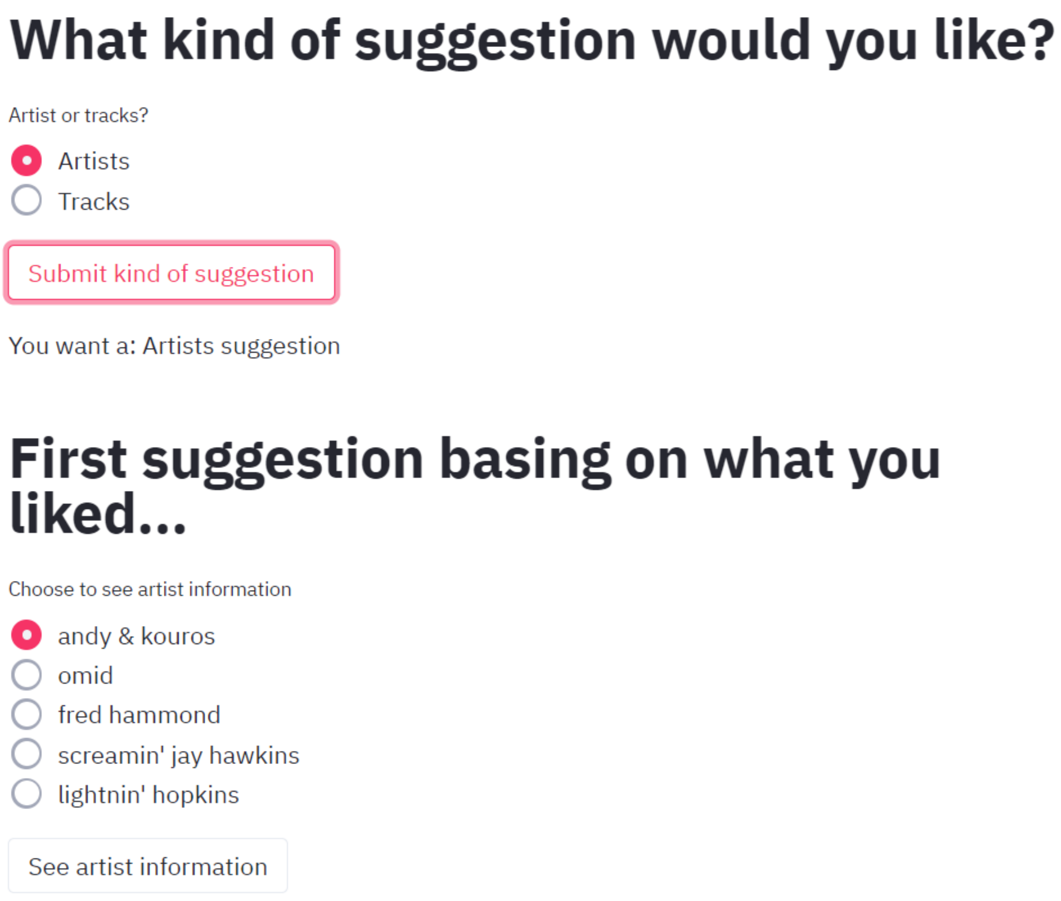


Figure 10: Genre insertion


Select the songs that you like
Whiskey For Sale - Bu...
Tapesh - Ebi
Do you like any song among these?
Submit Preferences

Your preferences:['Whiskey For Sale - Buddy Guy', 'Tapesh - Ebi']

Figure 11: Insertion first suggestions


What kind of suggestion would you like?

Artist or tracks?

[✓ Artists]
Tracks

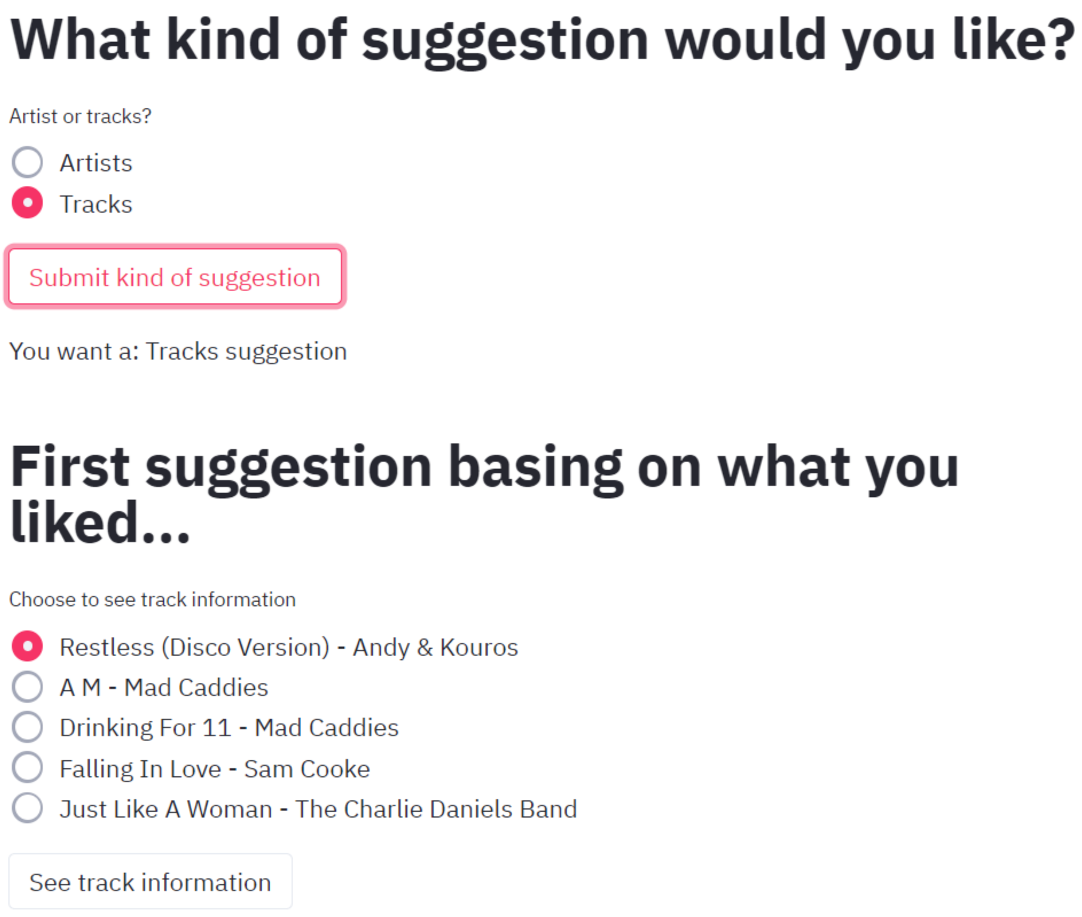
You want a: Artists suggestion

First suggestion basing on what you liked...

Choose to see artist information

[✓ andy & kouros]
omid
fred hammond
screamin' jay hawkins
lightnin' hopkins

Figure 12: Artist suggestion



What kind of suggestion would you like?

Artist or tracks?

Artists [✓ Tracks]

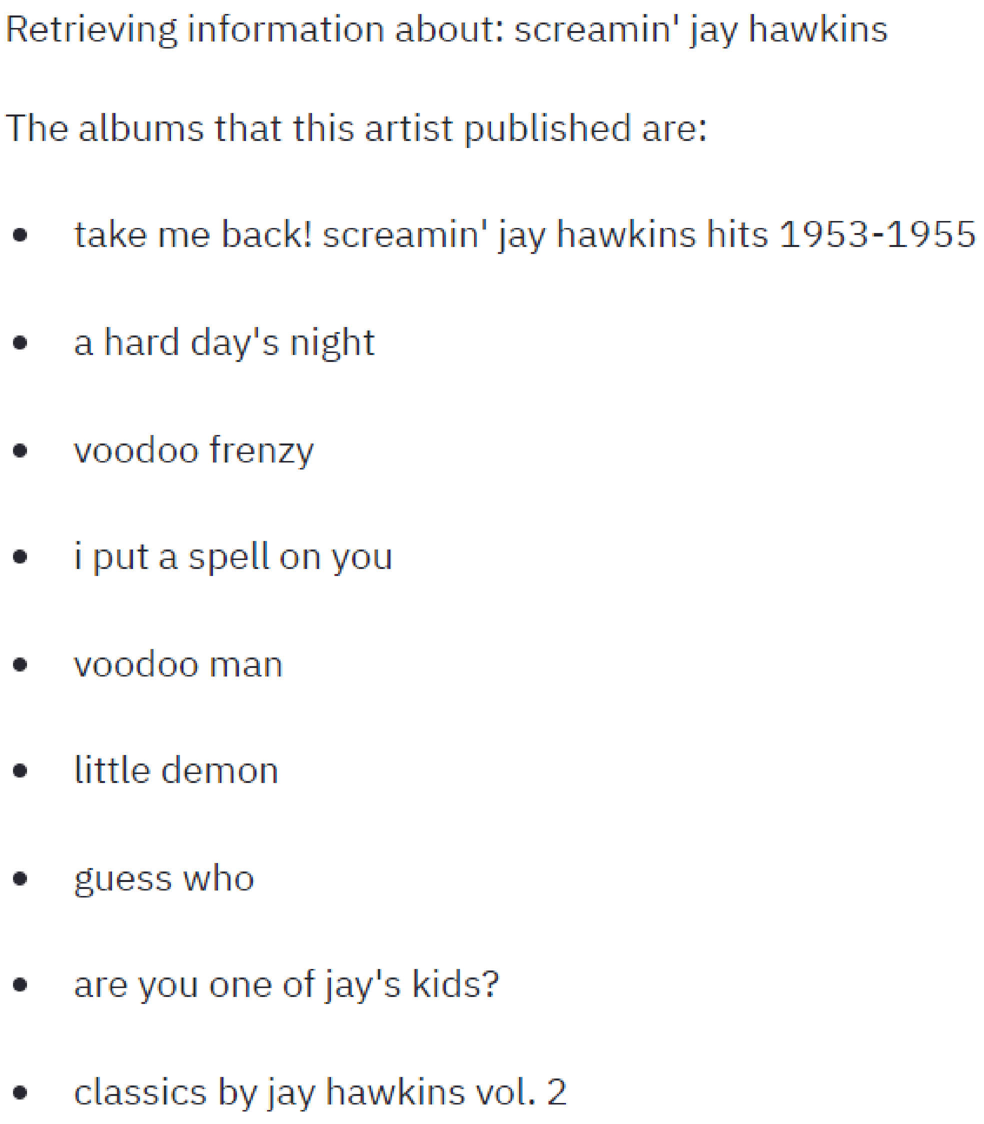
You want a: Tracks suggestion

First suggestion basing on what you liked...

Choose to see track information

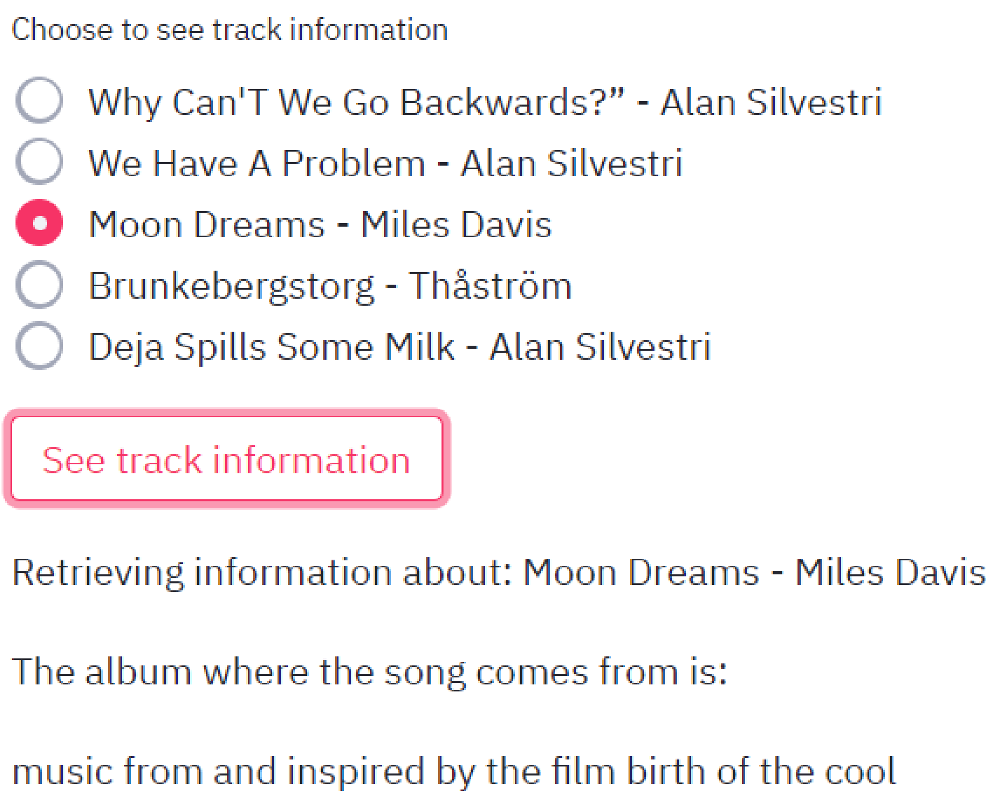
[✓ Restless (Disco Version) - Andy & Kouros]
A M - Mad Caddies
Drinking For 11 - Mad Caddies
Falling In Love - Sam Cooke
Just Like A Woman - The Charlie Daniels Band

Figure 13: Track suggestion



```
Retrieving information about: screamin' jay hawkins

The albums that this artist published are:

•  take me back! screamin' jay hawkins hits 1953-1955

•  a hard day's night

•  voodoo frenzy

•  i put a spell on you

•  voodoo man

•  little demon

•  guess who

•  are you one of jay's kids?

•  classics by jay hawkins vol. 2
```

Figure 14: Artist Information



Choose to see track information

- ◯ Why Can'T We Go Backwards?" - Alan Silvestri
- ◯ We Have A Problem - Alan Silvestri
- [✓ Moon Dreams - Miles Davis]
- ◯ Brunkebergstorg - Thåström
- ◯ Deja Spills Some Milk - Alan Silvestri

Retrieving information about: Moon Dreams - Miles Davis

The album where the song comes from is:

music from and inspired by the film birth of the cool

Figure 15: Track information

8 Installation

Being a web-app, this application doesn't require any particular installation process, but it needs some libraries to be already installed on the host machine.
To develop the project it has been required to correctly set the environment variables related to WordNet.

8.1 Wordnet Installation

This procedure was executed only once, before to develop the project, in order to integrate WordNet. The instruction are for Windows cmd. At first, set the environment variable WNDB to the directory of the repository, in the following way:
set WNDB=MusicRec/prolog/prolog
Then, set the PATH environment variable in the following way:
set PATH=MusicRec/prolog/wn

8.2 Manual installation

After having set Wordnet, it's necessary to clone Git repository locally, download Streamlit and PySwip, and then type in the terminal
streamlit run App/main.py.
Then a new window on the browser will be opened and the interface will be run.

8.3 Automatic Installation

To help in the installation of all the required components of the project, a batch file was included in the directory of the project. It contains the directives to automatically download the required components of the project and run the user interface. It's necessary to follow the easy passages illustrated below:

- Download the zip archive and extract it.
- Open the folder and click on install.bat: this batch file will installed the required dependencies to correctly run the project, including Streamlit and PySwip.
- When finishing the procedure, an automatic browser tab will open with the user interface and the system will be ready to be used.

9 Conclusions and Future Developments

The system that has been developed has the aim to exploit user's mood and preferences to give Artists or Tracks suggestions, using an underlying Prolog expert system capable of using similarities measures to give advices as accurate as possible. The knowledge

base comprehends, as already explained, nine files having ".pl" extension. Then, the other four files contain the functions used to manipulate data coming from the input and provide suggestions.
One of the main improvements that could be performed is enlarging the knowledge base, including more recent songs and further information, like the year of releasing of albums and tracks, other features describing each song or artist and a refinement about the genre of each track.
Unfortunately, accomplishing these goals request to perform a wide research on musical genres and integration from other sources that in terms of time was impossible to bring on. For what concerns the quality of suggestions, obiouvsly new similarity measures can be studied and then combined to reach even more refined results. In addition, there could even be a management of the dislikes in order to give a negative weight to the tracks that are not liked by the user.
Another point regards the tracks features provided by Spotify API. Some of the features reported in the API are not so relevant for the purpose of the system, and this has caused a rejection of many features because they did not bring any informativeness. A refinement could be building brand new features to describe the songs in order to consider different shades in their suggestion, calculating ,for example, the similarity according to different points of view. This can be a way to add variety and be more accurate in the advices.
For what about explanability, it could be useful to have a module that explains to the user why a certain suggestion was returned, but it would request the usage of an inference engine different from Prolog's one.

References

[1] Pascual Julian-Iranzo and Fernando Saenz-Perez. "WordNet and Prolog: why not?" In: *11th Conference of the European Society for Fuzzy Logic and Technology (EUSFLAT 2019).* Atlantis Press. 2019, pp. 827–834.