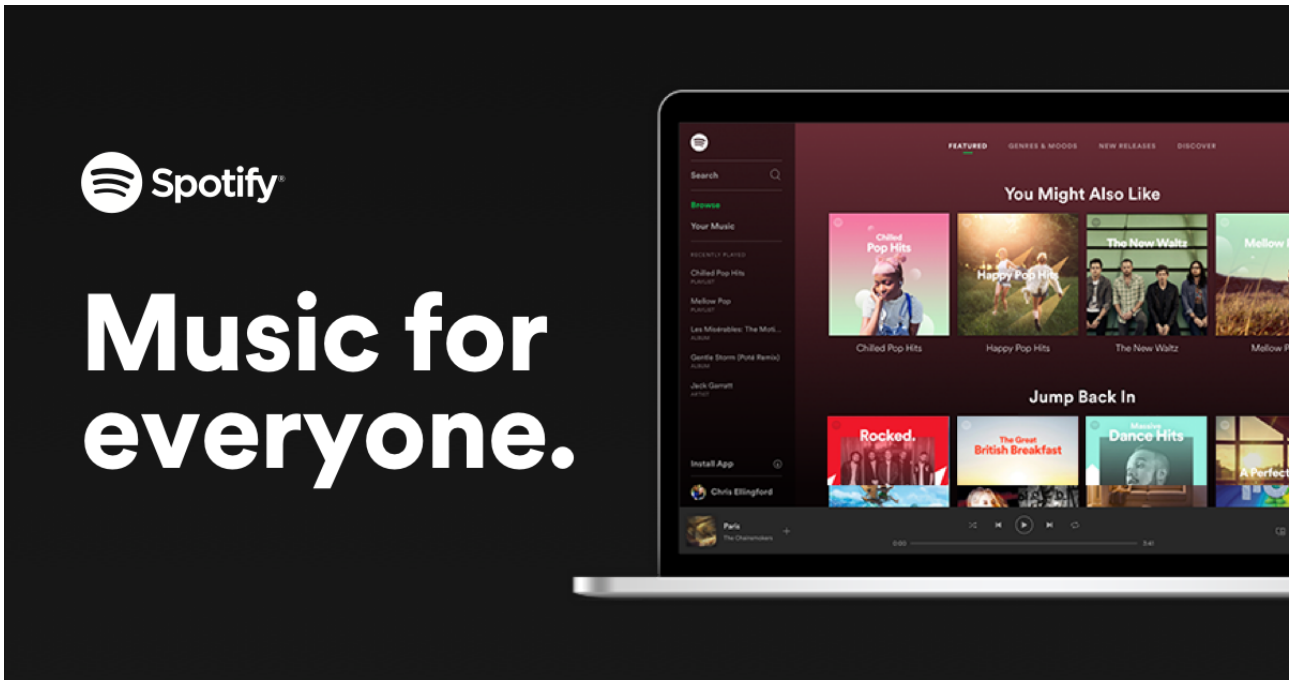
SPOTIFY SOCIAL NETWORK ANALYSIS

Introduction

Social Network Analysis (SNA) offers different methods, tools and techniques to study relationships, interactions and structures related to networks. It is based on the graph theory, so these concepts are used to calculate some particular metrics, such as the density, the centrality and so on.
The main objective of this discipline is to understand a community by mapping the relationships that connect them as a network, and then trying to draw out the key indivudals, groups within the network and/or the associations between the individuals.

Spotify is born as an audio streaming platform, which provides to its users the possibility to listen to any song of different artists.
Being such a dominant force in the music consumption market, Spotify is a unique source of data for the analysis of music; in fact, it has one of the largest digital collections of music in the world, and it employs a variety of algorithms to tag, quantify and classify its data to use in its recommendation system. During time, it has become a real social network, thanks to some additional functionalities, like sharing playlists, following other users, following artists.



By applying SNA to Spotify, it is possible to reach and satisfy different objectives:

- understanding how different artists are linked and which are the relationships between them;
- visualizing these relationships;
- studying all the main characteristics of the obtained network;
- identifying the most important artists in the network;
- observing what kind of music a particular user listen to;
- applying some clustering technique;
- performing statistical analysis;
- discovering new trends or patterns.

This work is focused on the SNA of Spotify, considering the artists stored on the platform. In the first section, there will be a description of the necessary settings and configurations used to retrieve data, including also the use of Spotify's APIs. The second section explains the creation of the graph and dataset, which will contain the data to be analyzed. The third section regards the social network analysis, in which all the related measures are calculated.

Configuration

Libraries

Spotify has its own developers API and a corresponding python library that can be used for these purposes.

In order to get the artists and their info, there is the need to perform some calls to the Spotify API. This can be done by using the *spotipy* Python package, that must be installed.

```
#install spotipy package
!pip install spotipy
```

```
Collecting spotipy
  Downloading https://files.pythonhosted.org/packages/7a/cd/e7d9a35216ea5bfb923
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: requests>=2.20.0 in /usr/local/lib/python3.6/dis
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/d
```

```
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.6/di
Installing collected packages: spotipy
Successfully installed spotipy-2.16.1
```

Different other libraries are needed in order to calculate the SNA metrics and to visualize the structure of the nodes and their links.

Besides the *spotipy* package, there are also:

- *NetworkX*, useful for the creation, manipulation and study of the structure, dynamics and functions of networks;
- *Numpy*, a powerful package for scientific computing;
- *Pandas*, imported to have a better view of the results in tabular form;
- *Matplotlib*, a comprehensive library for creating static and interactive visualizations.
- *Random*, that is used for the selection of random elements.

```
#libraries
import json
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
import networkx as nx
from networkx.readwrite import json_graph
import networkx.algorithms.community as nxComm
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from random import choice
```

Functions

Two functions have been created, that exploits the Spotify's API.

The first function *get_artist* allows to retrieve all the related information, given an artist, and to put them into a dictionary. The selected attributes that will be used in this work are:

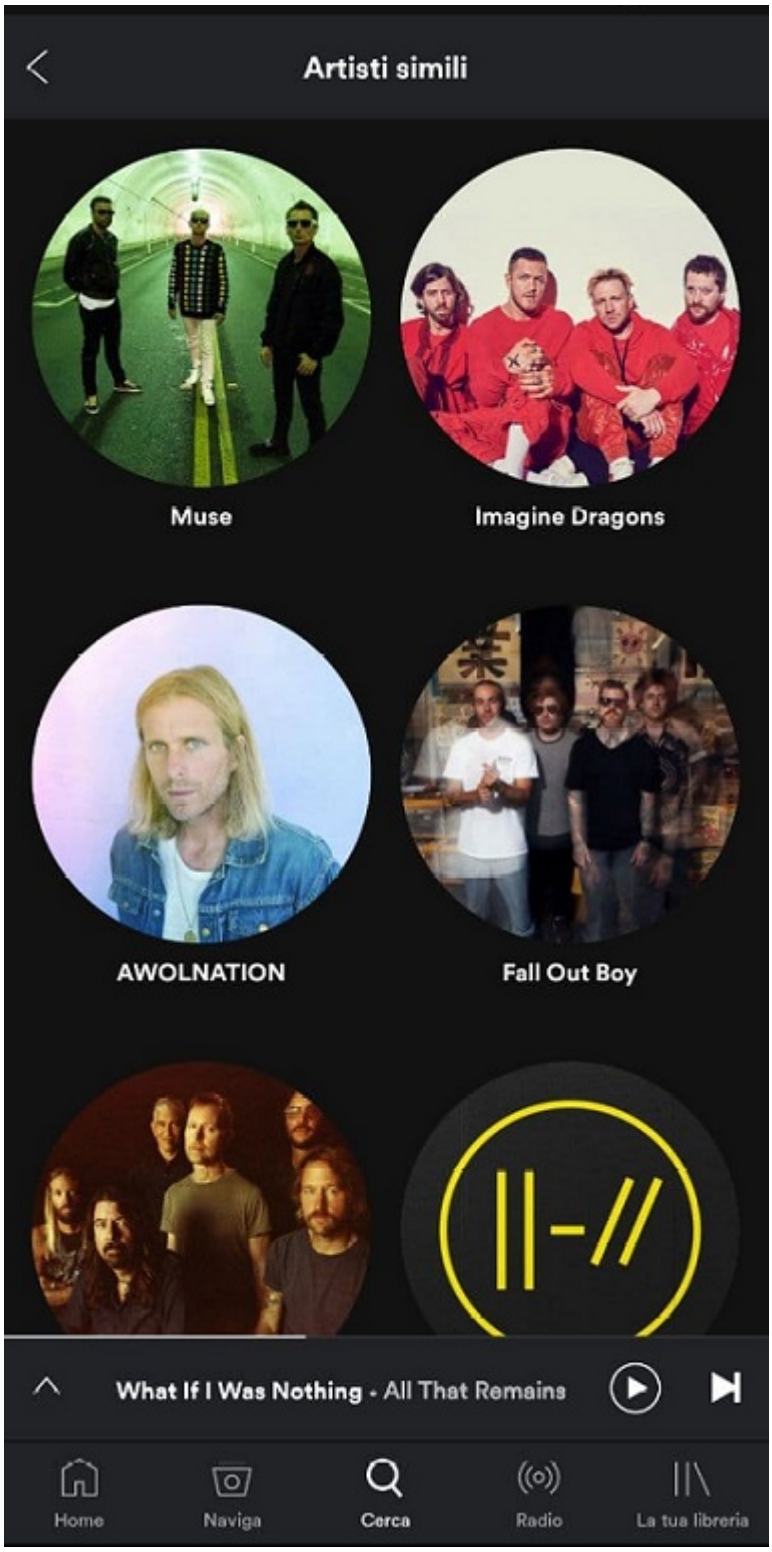
- name of the artist;
- the id of the artist, which can be obtained from the link of the related Spotify's page;
- the popularity, the degree that measures how much an artist is known in the community;
- the number of followers;

- the type of the artist;
- the profile image of the artist.

```
#function to get an artist with attributes
def get_artist(artist):
    info = dict()
    info['id'] = artist['name']
    info['uri'] = artist['uri']
    info['popularity'] = artist['popularity']
    info['followers'] = artist['followers']['total']
    info['type'] = artist['type']
    info['image'] = artist['images'][0]["url"]

    return info
```

The second one, which is called *related_artists*, is able to get all the related artists from a given one, by using the function *artist_related_artists* provided by the spotipy package and, again, they are stored into a dictionary. The results corresponds to the section "Ai fan piace anche" in Spotify ("Related artists" in the english version).

In this way, it is possible to retrieve the artists that are linked, in some way, to another one; this implies that these artists can produce the same genre of music, can have some band's member in common or they are involved in a collaboration.



This function is also responsible for changing the format of the retrieved results. Since, for each node the APIs return many attributes that, in this case study, are not needed, the format of the data is changed into another one, which results to be more appropriate. This is done through a series of for loops that goes deeper in the results structure and changes it.

```
#function to get related artists from a given one
def related_artists(sp, artist_id, artists, artists_list, aristas):
    artists_list.setdefault(str(1), [])
    artist_info = sp.artist_related_artists(artist_id)

    artists = sp.artist(artist_id)
    artists = artists["name"]
    aristas.setdefault(artists, [])

    for artist in artist_info['artists']:
        for level in artists_list.keys():
            for art in artists_list[level]:
                if artist['name'] == art['id']:
                        index_to_delete = artists_list[level].index(art)
                        del artists_list[level][index_to_delete]

            artist_id = artist['uri'].split(':')[2]
            artists_list[str(1)].append(get_artist(artist))
            if artist["name"] not in aristas[artists]:
                aristas[artists].append(artist["name"])
```

API Calls

In order to use Spotify API, there is the need to have an account on the platform and obtain an access token. This can be done in the *developer* section on the Spotify platform and by requesting the client id and the client secret.

The first step is to store those data.

```
#data for authentication
client_id = "398fd82988c04c35b6d7829f25a32cec"
client_secret = "d70f252182914ddcbda3ee72d3eebf52"
```

Then, by using the Spotipy package and passing the authentication data, it is possible to execute the call to the API.

```
#call to API
client_credentials_manager = SpotifyClientCredentials(client_id=client_id, client_sec
sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)
```

Dataset

Initialization

In the following blocks of code, there are some initializations, like the name of the dataset which will contain all the retrieved artists, the array of artists' ids choosed "by hand" and the initialization of the graph.

The dataset will be stored on the *Google Drive* platform, in a specific directory.

```
#name of the dataset
dataset_name = '/My Drive/Social Computing/related_artists.json'
```

id_array contains some artists' ids that will be used to retrieve the artists; this means that not all the nodes in the network are obtained in a random manner, but some nodes are choosen.
All these artists have been choosen according to the music I listen to and that I like; from this perspective, the network that will be obtained can suggest which kind of music a particular user love, if some bands and singers that user have listened to are selected.

The fact of choosing artists "by hand" is a necessity: this was performed with the only objective to obtain more nodes in the network.

```
#array that contains the id of artists
id_array = ['0RqtSIYZmd4fiBKVFqyIqD', '77SW9BnxLY8rJ0RciFqkHh', '7jy3rLJdDQY21OgRLCZ9
            '7FBcuc1gsnv6Y1nwFtNRCb', '5aYyPjAsLj7UzANzdupwnS', '0qT79UgT5tY4yudH9Vfs
            '2ye2Wgw4gimLv2eAKyk1NB', '1DFr97A9HnbV3SKTJFu62M', '5M52tdBnJaKSvOpJGz8m
            '6XyY86QOPPrYVGvF9ch6wz', '6deZN1bslXzeGvOLaLMOIF', '5S6hjAxgxjsLylsTtMIi
            '0k17h0D3J5VfsdmQ1iZtE9', '36QJpDe2go2KgaRleHCDTp', '22WZ7M8sxp5THdruNY3g
```

In the end, it is possible to initialize the graph and some variables that will be used subsequently.

From the following lines of code and considering the context of this work, it is possible to observe that the graph is not directed and it is not a multigraph.

```
#initialization of the graph
related = 0
```

```
related = 0
artists = None
artists_list = dict()

graph = dict()
graph["directed"] = False
graph["multigraph"] = False
graph['graph'] = {}
graph["nodes"] = list()
graph["links"] = list()
```

Creation of the graph

Through the use of the previous function *get_artist*, this code allows also to get all the data related to a particular artist; so, the info of the starting node, which is represented by *Thirty Seconds to Mars*, are obtained.

```
#get info about the starting node
aristas = dict()
related_artists(sp, id_array[0], artists, artists_list, aristas)

main_artist_info = sp.artist(id_array[0])
artists_list[str(related)] = []
artists_list[str(related)].append(get_artist(main_artist_info))
```

In order to retrieve the future nodes and edges of the network, a new function has been created. *create_graph* function allows to obtain all the related artists from a given one, it uses 2 important dictionaries to create the graph: *artists_list[level]* which contains all the artists corresponding to the nodes in the graph; *aristas* which contains the sources and the targets for each artist corresponding to the edges of the graph.

Nodes and edges are put together through a *for loop* that iterates for each of the choosen artists.

```
def create_graph(sp, id, artists):
    aristas = dict()
    artists_list = dict()
    related_artists(sp, id, artists, artists_list, aristas)

    for level in artists_list.keys():
        for node in artists_list[level]:
            graph["nodes"].append(node)
```

```
    for artist in aristas.keys():
        for friend_artist in aristas[artist]:
            edge = dict()
            edge["source"] = artist
            edge["target"] = friend_artist
            graph["links"].append(edge)

    rel_art = sp.artist_related_artists(id)

    return rel_art
```

At this point, it is possible to call the previous function so that the final graph can be built.

For each artist listed in *id_array*, the function *create_graph* is called three times: this means that there are 3 levels of the depth. Besides this, the node from which the research of artists should go on, is choosen randomly.

In this way, the graph is built in two ways: by using some artists already choosen and stored in *id_array* and by searching through the Spotify network.

```
#adding to the graph nodes and edges
for id in id_array:
    related_art = create_graph(sp, id, artists)
    rm_node = choice(related_art['artists'])

    related_art = []
    related_art = create_graph(sp, rm_node["id"], artists)
    rm_node = choice(related_art['artists'])

    related_art = []
    related_art = create_graph(sp, rm_node["id"], artists)
```

Creation of the dataset

By using the graph previously built, the final dataset is created; it is stored on Google Drive and its format is json.

```
from google.colab import drive
drive.mount('/content/drive')

base_path = '/content/drive'

with open(base_path + dataset_name, 'w') as outfile:
```

```
  json.dump(graph, outfile)
```

```
    Mounted at /content/drive
```

```
#open dataset
with open(base_path + dataset_name) as f:
    js_graph = json.load(f)
```

Social Network Analysis

The *NetworkX* library allows to compute all the measurements related to the SNA. The first thing to do is creating a NetworkX graph; since the graph is undirected, the function *Graph* is called.

```
#create graph with networkx
g = nx.Graph(json_graph.node_link_graph(js_graph))
```

Overview of the dataset

In order to observe all the graph's characteristics, the *pandas* library is used for the visualization of the results.

The following table represents the list of nodes in the graph, with the related attributes. For each of the obtained nodes, there are: the id of the artist (this corresponds to the name of a singer or band), the uri that contains the real id of the node, the degree of popularity, the number of followers, the type and the link to the image.

```
df = pd.DataFrame(js_graph["nodes"])
df
```

	id	uri	popularity	followers
0	Muse	spotify:artist:12Chz98pHFMPJEknJQMWvI	80	6083585
1	Imagine Dragons	spotify:artist:53XhwfbYqKCa1cC15pYq2q	90	30850970
2	AWOLNATION	spotify:artist:4njdEjTnLfcGImKZu1iSrz	70	1833557
3	Fall Out Boy	spotify:artist:4UXqAaa6dQYAk18Lv7PEgX	84	8327707
4	Foo Fighters	spotify:artist:7jy3rLJdDQY21OgRLCZ9sD	82	8116470

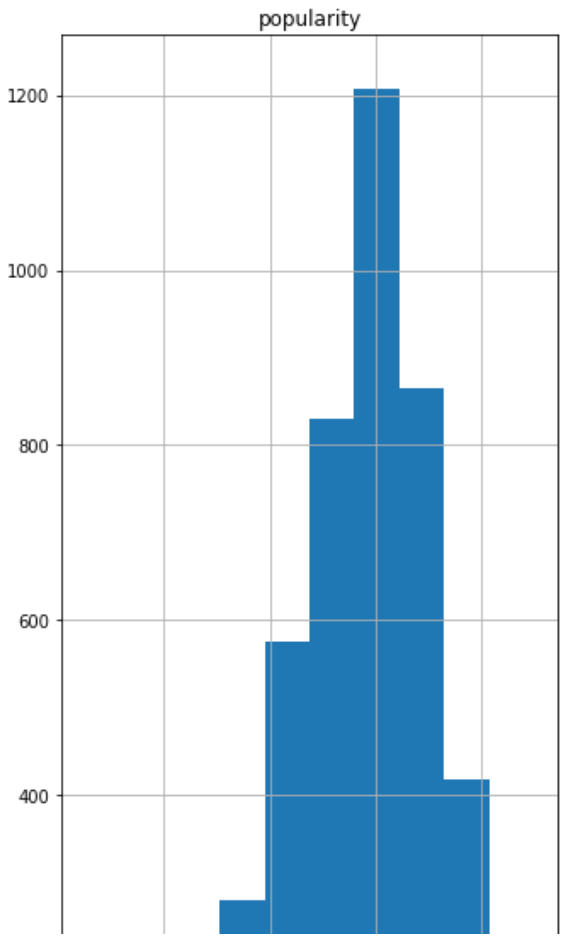
In this table, the nodes of the graph represents bands and singers that have been retrieved with the previous functions; the edges are the link between nodes: the relationship expressed by these links indicates that the actors involved in the relationship produce the same type of music.
An important thing to observe is that the number of nodes in the json file is much bigger with respect to the number of nodes in the graph: this happens because in the json file many nodes are duplicated.

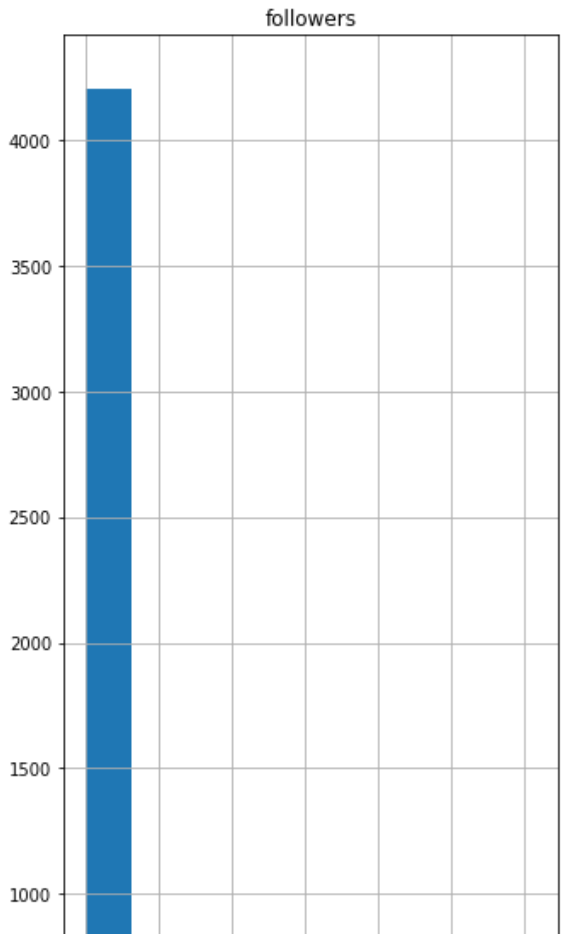
Then some plots describing the dataset.

In the built dataset, there are only two columns that represent numerical values; for this reason, only two plots are obtained, showing the popularity and the number of followers across the network.

```
fig, axes = plt.subplots(figsize = (12,12))
df.hist(ax = axes)
plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: UserWarning: To
```





The *Pandas* library allows to observe some characteristics of the retrieved data in numerical form, such as the mean, standard deviation, the minimum and maximum element. Again, this is done for the attributes popularity and number of followers.

```
df.describe()
```

The following code performs a test to understand if the graph is connected or not.

```
print('The graph {} connected.'.format('is' if nx.is_connected(g) else 'is not'))
```

```
The graph is not connected.
```

This result describes the fact that in the network there are different groups of nodes not connected with each other.

Nodes and Edges

List of nodes.

```
print(g.nodes)
```

```
['Muse', 'Imagine Dragons', 'AWOLNATION', 'Fall Out Boy', 'Foo Fighters', 'Twen
```

List of edges.

```
artists_df = nx.to_pandas_edgelist(g)
artists_df
```

	source	target
0	Muse	Thirty Seconds To Mars

```
print("Number of nodes: ", g.number_of_nodes())
print("Number of edges: ", g.number_of_edges())
```

```
Number of nodes:  1390
Number of edges:  3366
```

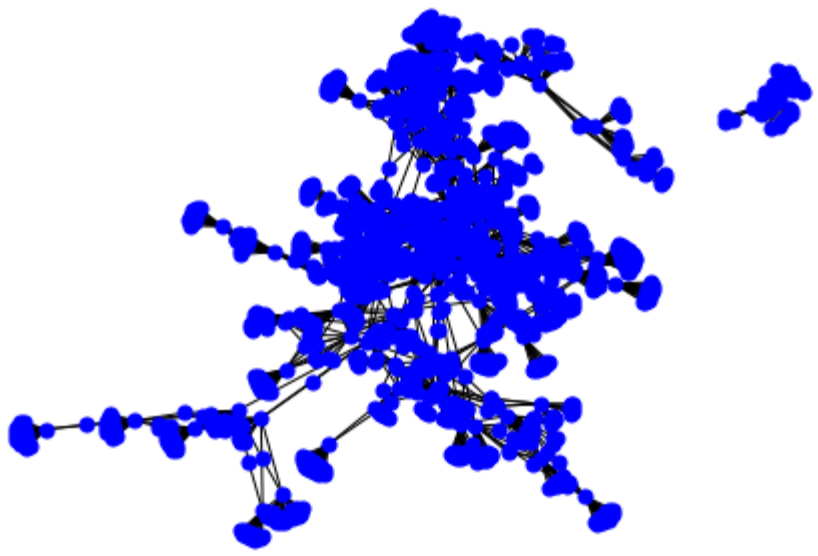
These values represent the effective number of nodes and edges in the graph.

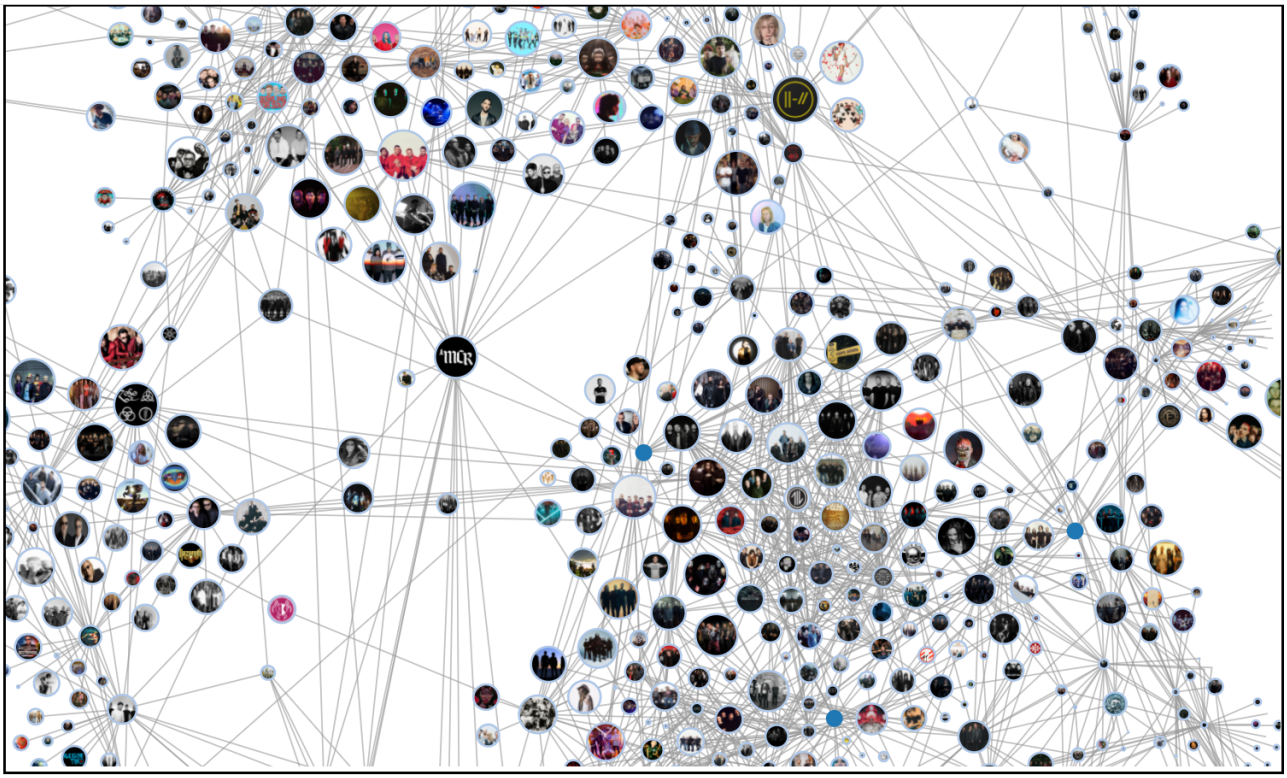
3361	YUNGBLUD	Bohnes

Graph

The graph can be visualized in different ways.

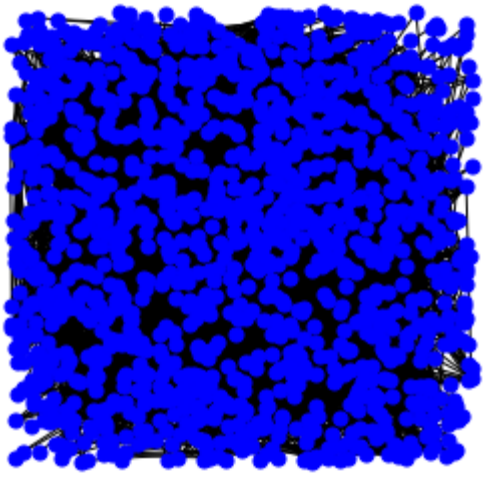
```
#graph
options = {
    'node_color': 'b',
    'with_labels': False,
    'node_size': 50
}

nx.draw(g, **options)
```





Random

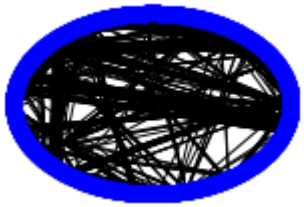
```
plt.figure(figsize=(10, 10))
plt.subplot(221)
nx.draw_random(g, **options)
```



Circular

```
plt.subplot(222)
```

```
nx.draw_circular(g, **options)
```

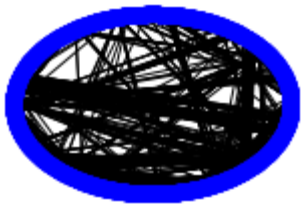


Spectral

```
plt.subplot(223)
nx.draw_spectral(g, **options)
```

Shell

```
plt.subplot(224)
nx.draw_shell(g, **options)
```



Density and LCC

Density is the ratio between the actual and possible edges in the network.

```
#density
density = nx.density(g)
print("Density of Graph: ", density)
```

```
Density of Graph:  0.0034868001926752334
```

The obtained results show that this network is not so much dense; this means that there is a low number of edges between the nodes in this graph and that the information doesn't trasmit very efficiently across the network because it has to go from member to member, rather than diffusing from one member rapidly to all the others. This aspect is useful if the purpose is to recommend some artists.

Since the graph is not connected, it is possible to identify the Largest Connected Component (LCC), which is the biggest component in the graph.
Different components indicate different genres of music; the biggest one represents rock music.

```
#Graph's Largest Connected Component
lcc = max(nx.connected_components(g), key=len)
print("Largest Connected Component: ", lcc)
print("Number of nodes in the lcc: ", len(lcc))
```

```
Largest Connected Component:  {'UFO', 'Rapsoul', 'Morphine', 'Attila', 'Alkalin
Number of nodes in the lcc:  1348
```

Density of lcc.

```
#Density of lcc
LCC = g.subgraph(lcc)
print("Density of the lcc: ", nx.density(LCC))
```

```
Density of the lcc:  0.003600704059355993
```

Average Path Length

The *Average Path Length* is the average of the shortest path length, averaged over all pairs of nodes.

If a network contains disconnected components, the APL diverges to infinity; one way to avoid this problem is to calculate this parameter only from nodes in a connected component.
In order to calculate the APL, it is necessary to consider a component in the graph and, then, measure the apl of that component.

```
#Average Path Length
connected_components = list(nx.connected_components(g))
connected_component_1 = g.subgraph(connected_components[0])
print("Component:", list(connected_component_1.nodes))
apl = nx.average_shortest_path_length(connected_component_1)
print("\nAverage Path Length: ", apl)
```

```
Component: ['Muse', 'Imagine Dragons', 'AWOLNATION', 'Fall Out Boy', 'Foo Fight

Average Path Length:  6.7508453779031985
```

Degree Centrality

Degree Centrality allows to identify the most important nodes in the network. In this case, the degree represents the number of edges that connect the node to other nodes.

```
#Centrality
print("\nDegree Centrality")
deg_cent = nx.degree_centrality(g).items()
degcent_df = pd.DataFrame(columns=['Node', 'Centrality'])
for node, cent in deg_cent:
  degcent_df = degcent_df.append({
      'Node': node,
      'Centrality': cent,
  }, ignore_index=True)

degcent_df
```

```
Degree Centrality
```

	Node	Centrality
0	Muse	0.001440
1	Imagine Dragons	0.014399
...	...	...

Through the previous table, it is possible to identify the very connected individuals, the most popular artists. In this case, all the artists who have a high degree centrality are very popular and they have more followers than others.

Closeness Centrality

Regarding the closeness centrality, the importance of a node depends on its closeness to other nodes. So, the more central a node is, the closer it is to all other nodes.
Closeness centrality measures each individual's position in the network via a different perspective from the other network metrics, capturing the average distance between each vertex and every other vertex in the network.

```
#Closeness Centrality
print("\nCloseness centrality")

dc = nx.degree_centrality(g)
dc_sorted = {k: v for k, v in sorted(
    dc.items(),
    key=lambda item: item[1],
    reverse=True
)}

clos_cent = dc_sorted.items()
closcent_df = pd.DataFrame(columns=['Node', 'Closeness'])
for node, close in clos_cent:
  closcent_df = closcent_df.append({
      'Node': node,
      'Closeness': close,
  }, ignore_index=True)

closcent_df
```

```
Closeness centrality
```

	Node	Closeness
0	blessthefall	0.023038
1	A Skylit Drive	0.022318
2	Seether	0.021598
3	The Word Alive	0.020878
4	Escape the Fate	0.020878
...	...	...
1385	Jack Stauber's Micropop	0.000720
1386	mxmtoon	0.000720
1387	The Happy Fits	0.000720
1388	The Brobecks	0.000720

In this way, the more important nodes in the network are identified, but this is done from a different perspective with respect to the first metric.

Betweenness Centrality

The betweenness centrality calculates the number of times that a node behaves as a bridge in the shortest path between two other nodes.

```
print("\nBetweenness centrality")

betcent_df = pd.DataFrame(columns=['Node', 'Betweenness'])
bet_cent = nx.betweenness_centrality(g).items()

for node, bet in bet_cent:
  betcent_df = betcent_df.append({
      'Node': node,
      'Betweenness': bet,
  }, ignore_index=True)

betcent_df
```

```
Betweenness centrality
```

	Node	Betweenness
0	Muse	0.000000
1	Imagine Dragons	0.086527
2	AWOLNATION	0.000000
3	Fall Out Boy	0.016214
4	Foo Fighters	0.204774
...	...	...
1385	The Pretty Reckless	0.029893
1386	Metallica	0.211796
1387	Nirvana	0.004891
1388	Linkin Park	0.021637

A high betweenness indicates that a particular node has more authority over the other nodes in the network.

Gatekeepers

By using the betweenness centrality values, the identification of the *gatekeepers* can be performed.

Gatekeepers are those nodes with a high betweenness centrality and they hold a critical position between other nodes that are not directly linked and thus, in a network, they are nodes that provide a connection, and serve as a bridge.

```
#gatekeepers
gatekeepers = sorted(nx.betweenness_centrality(g).items(), key = lambda x : x[1],  re

print("Network gatekeepers:")

for gatekeeper in gatekeepers:
  print(gatekeeper)
```

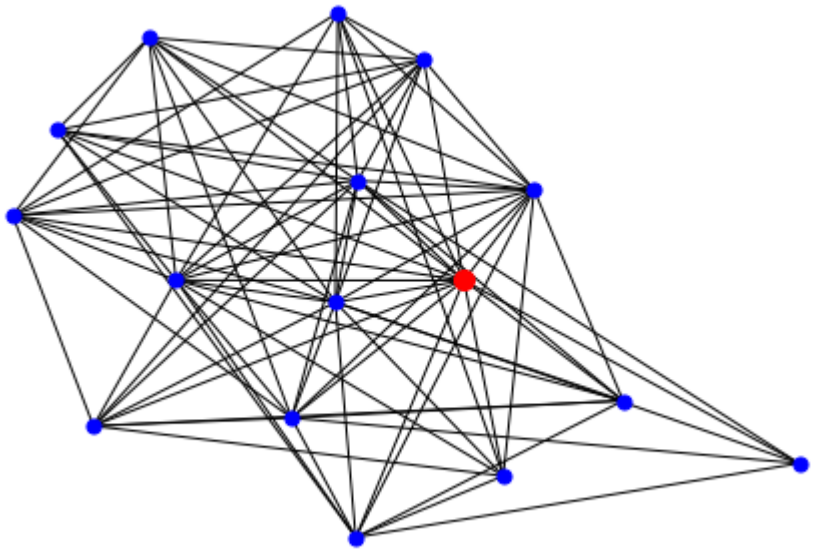
```
Network gatekeepers:
('Thirty Seconds To Mars', 0.3744676888270558)
('Metallica', 0.21179639121732452)
('Foo Fighters', 0.20477430961534873)
```

Ego Network

An *ego network* is defined as a classical network with nodes and edges, but it is based on a focal node (ego) and the nodes to whom ego is directly connected to (alters).

In order to create a ego network, an ego node must be choosen and this is done through the *random* library, that allows to select a random node in the graph.

```
from random import choice

rm_node = choice(list(g.nodes))

ego_network = nx.ego_graph(g, rm_node)
pos = nx.spring_layout(ego_network)
nx.draw(ego_network, pos, **options)
nx.draw_networkx_nodes(ego_network, pos, nodelist=[rm_node], node_size=100, node_colo
plt.show()
```



Cliques of the Ego Network

A *clique* is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent.

```
from networkx.algorithms import approximation, clique
```

```
# Find the number of maximal cliques in the graph
n_of_cliques = clique.graph_number_of_cliques(ego_network)
print("Number of cliques in the ego network:", n_of_cliques, end='\n\n')

# Print all the cliques
all_cliques = clique.find_cliques(ego_network)
print("All the cliques in the ego network:")
cli = pd.DataFrame(dict(CLIQUES = all_cliques))
print(cli)

# Find the maximum clique in the network
max_clique = approximation.clique.max_clique(ego_network)
print('\nThe bigger clique:', max_clique)

# Clique number of the graph (size of the largest clique in the network)
graph_clique_number = clique.graph_clique_number(ego_network)
print('Graph clique number:', graph_clique_number)
```

```
Number of cliques in the ego network: 15

All the cliques in the ego network:
                                               CLIQUES
0   [All Time Low, Forever The Sickest Kids, We Th...
1   [All Time Low, Forever The Sickest Kids, We Th...
2   [All Time Low, Forever The Sickest Kids, We Th...
3   [All Time Low, Forever The Sickest Kids, We Th...
4   [All Time Low, Forever The Sickest Kids, We Th...
5   [All Time Low, Forever The Sickest Kids, We Th...
6   [All Time Low, Forever The Sickest Kids, We Th...
7   [All Time Low, Forever The Sickest Kids, We Th...
8   [All Time Low, Forever The Sickest Kids, We Th...
9   [All Time Low, Forever The Sickest Kids, We Th...
10  [All Time Low, Forever The Sickest Kids, We Th...
11  [All Time Low, Forever The Sickest Kids, We Th...
12  [All Time Low, Forever The Sickest Kids, We Th...
13  [All Time Low, Forever The Sickest Kids, The A...
14  [All Time Low, Forever The Sickest Kids, The A...

The bigger clique: {'You Me At Six', 'We The Kings', 'Cute Is What We Aim For',
Graph clique number: 8
```

The found cliques represents those artists that are connected to the others in the ego network, creating a more compact group.

Limitations and Conclusion

It is very important to consider that there are different limitations to this study.

The first one and, maybe the most important, comes from the use of the Spotify platform. Each artist can have a maximum number of related nodes, which is equal to 20: this can be seen as a problem, because this characteristic of Spotify doesn't allow to find many nodes in the network.
For this reason, some artists are choosen "by hand", inserting the related ids in an array; then, for each artist it was performed an exploration in depth in the network, taking into account 3 levels of depth and selecting one random node a level. Even with this method, the reached number of nodes is equal to 1267, which is not too much high.

Another limitation is due to the fact that the analysis is based on data which have the potential to be incomplete or untimely; so, the results may be most usefully considered in combination with other sources of information and operational experience.

Despite all these situations, the decribed approach does not limit itself to identifying the structure of the network, groups of artists and the most important nodes.