Lab activity: analysis of music networks.

1 Part 2: Data preprocessing

1.1 Introduction

In this second part of the lab activity, we will work on the preprocessing of the data downloaded in the first session. In other words, we will prepare the data to make it easier to extract information from it later on using analysis algorithms.

Firstly, we will implement three functions that work with NetworkX graphs, which will allow us to convert directed graphs into undirected graphs and prune graphs according to different criteria. Then, we will create two additional functions that will generate artist similarity graphs based on the audio features of their songs. Finally, we will perform an initial exploration of the original data and the data obtained with these preprocessing functions.

1.2 Data preprocessing

In this second session of the practice, we will use the artist graphs and the song dataset that we obtained in the previous session. Along with the instructions for this second session, a file is attached specifying the functions that need to be implemented for each activity (the file `Lab_AGiCI_202324_P2_skeleton.py`).

1. (**1 point**) Implement the function `retrieve_bidirectional_edges` with the following header:

```python
def retrieve_bidirectional_edges(g: nx.DiGraph, out_filename: str) -> nx.Graph:
    """
    Convert a directed graph into an undirected graph by considering bidirectional edges only.

    :param g: a networkx digraph.
    :param out_filename: name of the file that will be saved.
    :return: a networkx undirected graph.
    """
```

   The function will receive a directed graph as input and return an undirected graph. The resulting undirected graph will have as edges those edges from the original graph that existed in both directions. In other words, the edge $e = (v_i, v_j)$ will exist in the new graph if and only if both edges $e = (v_i, v_j)$ and $e' = (v_j, v_i)$ existed in the input graph. The nodes of the new graph will be defined by the edges contained in the new graph (only nodes that have incident edges in the new graph should appear).

The function retrieve_bidirectional_edges will return the resulting graph and, additionally, it will save it to disk in a file with the specified name as a parameter (in graphml format).

2. (**1 point**) Implement the function prune_low_degree_nodes with the following header:

```python
def prune_low_degree_nodes(g: nx.Graph, min_degree: int, out_filename: str)
    -> nx.Graph:
    """
    Prune a graph by removing nodes with degree < min_degree.

    :param g: a networkx graph.
    :param min_degree: lower bound value for the degree.
    :param out_filename: name of the file that will be saved.
    :return: a pruned networkx graph.
    """
```

The function will receive an undirected input graph and generate a graph from which all nodes with a degree less than min_degree have been removed. The function will have three input parameters: an undirected networkx graph, a variable with the minimum degree value, and the output file name where the generated graph will be saved.

This removal of nodes with a degree less than min_degree should be done in a single pass. In other words, at the end of the traversal of all nodes, there may still be nodes with a degree less than min_degree. In this case, we will keep these nodes. After this process, you should remove the zero-degree nodes from the resulting graph.

3. (**1.5 points**) Implement the function prune_low_weight_edges with the following header:

```python
def prune_low_weight_edges(g: nx.Graph, min_weight: float = None,
   min_percentile: int = None,
                           out_filename: str = None) -> nx.Graph:
    """
    Prune a graph by removing edges with weight < threshold. Threshold can
   be specified as a value or as a percentile.

    :param g: a weighted networkx graph.
    :param min_weight: lower bound value for the weight.
    :param min_percentile: lower bound percentile for the weight.
    :param out_filename: name of the file that will be saved.
    :return: a pruned networkx graph.
    """
```

The function will receive an undirected input graph with weighted edges and generate a graph from which all edges with a weight less than the value specified as a parameter will be removed. This weight can be specified in two different ways: either directly with the threshold value (parameter min_weight) or with the percentile value (parameter min_percentile). The function should raise an exception if neither of the two parameters is specified or if both parameters are specified (i.e., the function call should only specify one of the two parameters).

After the edge removal process, you should remove the zero-degree nodes from the resulting graph.

In addition to returning the resulting graph, the function will also save it to disk in the file specified by the parameter name (in graphml format).

4. (**1 point**) Implement the function `compute_mean_audio_features` with the following header:

```
def compute_mean_audio_features(tracks_df: pd.DataFrame) -> pd.DataFrame:
    """
    Compute the mean audio features for tracks of the same artist.

    :param tracks_df: tracks dataframe (with audio features per each track).
    :return: artist dataframe (with mean audio features per each artist).
    """
```

   The function will receive a `dataframe` of songs (the result of the `get_track_data` function implemented in session 1) and will return another `dataframe` with the average audio characteristics of each artist. In other words, the resulting `dataframe` will have one row for each artist, which will contain both the identification data of that artist (at least, the identifier and name) and the average values of each audio characteristic (*danceability, energy, loudness, ...*) of all the songs by that artist.

5. (**1.5 points**) Implement the function `create_similarity_graph` with the following header:

```
def create_similarity_graph(artist_audio_features_df: pd.DataFrame,
    similarity: str, out_filename: str = None) -> nx.Graph:
    """
    Create a similarity graph from a dataframe with mean audio features per
     artist.

    :param artist_audio_features_df: dataframe with mean audio features per
     artist.
    :param similarity: the name of the similarity metric to use (e.g. "
    cosine" or "euclidean").
    :param out_filename: name of the file that will be saved.
    :return: a networkx graph with the similarity between artists as edge
    weights.
    """
```

   The function will receive a `dataframe` with the average audio characteristics for different artists and a similarity measure, and will construct a complete graph (undirected and with weighted edges) that stores the similarity between each pair of artists as the weight of the connecting edge.

   The function will return the resulting graph and also save it to the specified file (in `graphml` format).

6. (**1 point**) Using the previous functions, obtain:

   (a) Two undirected graphs (g'_B and g'_D) of artists obtained by applying the programmed function in exercise 1, `retrieve_bidirectional_edges`, to the graphs obtained by the crawler in session 1, g_B and g_D.

   (b) One undirected graph with weights (g^w) obtained from the similarity between the artists. To obtain it, it will be necessary to calculate the vector of average audio features (`compute_mean_audio_features`) for each artist appearing in both graphs (g_B and g_D) and create a similarity graph with these features (`create_similarity_graph`).

1.3 Questions to be addressed in the report

1. (**1.5 points**) Justify whether the directed graphs obtained from the initial exploration of the crawler (g_B and g_D) can have more than one weakly connected component and strongly connected component, and explain why. Indicate the relationship with the selection of a single seed.

2. (**0.5 points**) Can the number of connected components in the undirected graphs (g'_B and g'_D) be higher than the number of weakly connected components of its respective directed graph (g_B and g_D)? Provide a minimal example to showcase your answer.

3. (**1 point**) Generate a preliminary report from the undirected graph with weights (g^w).

   (a) Which are the two most (respectively, least) similar artists? What graph attribute allows you to answer this question?

   (b) Which is the artist most (and least) similar to all the other artists in the network? What graph attribute allows you to answer this question?

1.4 Evaluation and expected results for this part of the practice

This second session of the practice will account for 25% of the total grade for the practice.

Remember that it is important that **the Python code includes sufficient comments** to understand its functionality and **respect the headers** of the provided functions.

The files to be obtained in this first part of the practice for the final submission are as follows:

- `Lab_AGX_202324_P2_skeleton.py`: a plain python file where each of the functions defined in this document has been implemented.
- `Lab_AGX_202324_report.pdf`: the final report (a pdf file) must have a section called **Part 2: data preprocessing** where all the questions asked in this document will be answered in detail. The answers to the questions should be numbered and in order.
- The data files obtained in exercise 6 (three `graphml` files, containing the graphs g'_B, g'_D, and g^w).
  - `gBp.graphml`: a graphml file with the undirected artist graph g'_B.
  - `gDp.graphml`: a graphml file with the undirected artist graph g'_D.
  - `gw.graphml`: a graphml file with the undirected artist graph g^w.