Documentation

Team: Who's That Artist?!

Name	NetID
Hyo Sup Kim (Captain)	hyosup2
Yutong Dai	ymdai2
Chang Hun Park	chp2

1) Code Overview

Our project presents a front-end search bar which takes an artist name from the user and uses the name to extract their most popular songs and perform analyses on their lyrics and musicality. We have written code files dedicated to the search and result UI, as well as separate files analyzing their musical and lyrical significance. Relating back to lecture topics, these insights include utilizing LDA (latent dirichlet allocation) for the topic mining of and theme extraction from their lyrics, identifying nouns to improve the themes and word clouds we generate through natural language content analysis, and summarizing the polarity and subjectivity of their lyrics through sentiment analysis and classification. The code output can thus be used to provide hopefully interesting insights into any desired artist's discography or even serve as a basis to building a comprehensive database detailing similarly intriguing features of their music.

2) Software Documentation Implementation

Our web app is split into three major files: a file that retrieves and computes lyrical significance of an artist's lyrics (*genius.py*), a file that extracts interesting information regarding their songs' musicality (*spotify.py*), and a file that contains the front-end Dash framework that allows a user to input their desired query artist by connecting the outputs of the previous two files. In the end, it presents it as a cohesive output for the user on a results page (*dash_app.py*).

spotify.py

This file provides functionalities for interacting with the Spotify API to retrieve and analyze information related to the user's selected artist, which includes their discography, audio features of their songs and a picture of their Spotify profile, which is usually their face. The integration of data analysis, image retrieval and data visualization allows users to comprehensively explore their chosen artist's musical profile without having to individually compile it themselves.

The relevant functions are as follows:

- *def get_artist_info_csv_smaller:*
    - This function fetches an artist's information, which includes audio features such as acousticness, danceability, energy and tempo of their top tracks. The data is then saved in a CSV file for further analysis.
    - Due to the constraints placed on the Spotify API itself, limits have been arbitrarily placed on the number of songs being fetched as exceeding the rate limit has resulted in the app being temporarily suspended from issuing queries.
    - *Input*:
        - "artist_name" (Name of the artist)
    - *Output*:
        - CSV file with artist information saved locally.
- *def get_artist_face:*
    - This function retrieves and saves the artist's profile image based on the user input. The image is saved in JPG format for the app to display on its UI.
    - Note that the retrieved image retrieves the profile image of the artist on Spotify, which may not be a picture of their face.
    - Functions within this function utilize the Spotipy library for lightweight interactions with the Spotify API.
    - Results are saved in the "Final/Scripts/" directory by default unless otherwise specified.
    - *Input:*
        - "artist_name" (Name of the artist)
    - *Output:*

        - JPEG file containing the artist's profile image saved locally.
- *def create_distribution_plot:*
    - Creates a distribution plot for a specified DataFrame column, using the seaborn library and the matplotlib library. The resulting plot is converted to a base64 encoded image format for the app to access and display.
    - *Input:*
        - "df" (DataFrame generated by "get_info_csv_smaller")
        - "column" (Column name for the distribution plot, default is set to Danceability)
        - "color" (Color of the plot)
        - "title" (Title of the plot)
    - *Output:*
        - Base64 encoded image format for the app to access and display.
- *def create_pairplot:*
    - Generates a pairplot for the DataFrame, allowing for a quick exploratory data analysis overview of relationships between the different variables within the generated DataFrame.
    - *Input*:
        - "df" (DataFrame generated by "get_info_csv_smaller")
    - *Output*:
        - Base64 encoded image format for the app to access and display.

genius.py

This file takes the artist from where it is input in the dash_app file and performs the lyrical portion of the web application analysis. The crux of this file is the lyrics genius API, from which the main function obtains the lyrics for a queried artist that it later uses to determine meaningful information from. The relevant functions are as follows:

➔ *def process_artist_lyrics*:

This is the main function of the file that is called in dash_app with a string input denoting the queried artist. It passes in the query artist to later functions and compiles all their returned results as output.

- *Input:* query_artist *(name of artist queried by user)*
- *Output:*
  - all_songs *(list of processed lyrics of artist's popular songs)*
  - themes *(list of up to 3 themes identified in artist's songs)*
  - img_wordcloud *(image of word cloud of commonly seen words)*
  - img_polarities *(graph of distribution of song polarity values)*
  - img_subjectivities *(distribution graph of song subjectivity values)*
  - subjectivity_rating *(weighted average of song subjectivity ratings)*
  - polarity_verdict *(weighted average of song polarity ratings)*

➔ *def get_lyrics*:

This function retrieves all the lyrics of (up to) the top 20 most popular songs of the input artist, as they are ranked by the Genius API. This is easily the most time-consuming part of the code, so before running this step, the function checks to see if our database list (of artists that have already been searched, whose information is thus already obtained and saved) before calling the Genius API by calling the check_artist function. This database list will be naturally added to as it calls the save_artist function to save any artist information that it does not currently have in the database list by save_artist. From the list of 20 songs obtained, the function further checks for repeats in songs (most often stemming from multiple stage and studio versions of the same lyrics) with check_repeat and saves only one copy of those songs to prevent bias. After all relevant songs have been retrieved, it goes through their lyrics and runs process_lyrics before saving.

- *Input:* query_artist *(name of artist queried by user)*
- *Output:*
  - all_songs *(list of processed lyrics of artist's popular songs)*
  - all_lyrics *(concatenated string containing all processed lyrics)*

➔ *def check_artist:*

This function checks for the existence of an artist in our database, a portion of which we compiled prior to submission to speed up the search and calculation process of our web application. It will first check for the artist name in the database, then check for any potential spelling errors in the artist name, before determining for certain whether the artist already exists in our list or not.

- ◆ *Input:* query_artist *(name of artist queried by user)*
- ◆ *Output:* artist_name *(name of the artist as it exists in our files, else False)*

➔ *def save_artist:*

This function saves the artist name (processed to become a concatenated firstname+lastname combination for consistency) and their titles/lyrics that were initially retrieved through the genius API and processed in process_lyrics.

- ◆ *Input:*
  - artist_name *(name of artist queried by user)*
  - song_titles *(list of processed song titles for artist)*
  - all_songs *(list of processed lyrics for all songs in song titles)*
- ◆ *Output:* None

➔ *def check_repeat:*

This function checks whether a song has already been added to the list of popular songs for an artist, since the Genius API often gives repeats of songs if they have multiple versions (Studio Version, Live, different albums, etc.). It preprocesses the name by removing anything in the title in parentheses or brackets before making the comparison.

- ◆ *Input:*
  - song_list *(list of song titles that have already been added)*
  - new_song_title *(new potential song title to add)*
- ◆ *Output:*
  - boolean *(True if new_song_title is unique, False otherwise)*
  - song_title *(processed song title)*

➔ *def process_lyrics:*

This function processes the raw lyrics by removing less significant words. (1) It removes the title and other miscellaneous information that is prone to be at the start of Genius API-retrieved lyrics. (2) It removes song part detonators like "[Chorus]". (3) It filters out numbers, respells slang (i.e. "goin'" to "going"), checks spelling, and removes common stopwords before appending the final filtered lyrics to the output list.

- ◆ *Input:* lyrics *(list of lyrics of artist's most popular songs)*
- ◆ *Output:* filtered_lyrics *(list of lyrics post-processing as described)*

➔ *def get_theme*:

From the processed lyrics, it extracts the nouns by tokenizing and tagging their part of speech with nltk. Repeat this a few times to make sure any verbs that can double as nouns get filtered out. It then allows gensim to preprocess the words again in preparation for LDA model input. The function then removes the most common non-stopwords that won't meaningfully contribute to the theme from the lyrics (e.g. 'no', 'yes') based on the top100common.txt list before performing another stopword removal. It is then input into an LDA model to extract the top 20 most likely topics. From these twenty output words, the function feeds them into the get_abstract function which ultimately returns the themes. If there are no themes returned, then it will return the top three topics instead.

- ◆ *Input:* lyrics *(processed lyrics from the output of process_lyrics)*
- ◆ *Output:* themes *(list of up to three words describing theme of the lyrics)*

➔ *def sent_to_words*:

Calls gensim.utils.simple_preprocess on the input lyrics.

- ◆ *Input:* lyrics *(processed lyrics from the output of process_lyrics)*
- ◆ *Output:* yields a sequence of processed lyrics

➔ *def get_abstract*:

This function trains a simple two-category LogisticRegression classifier that differentiates between concrete and abstract nouns from a simple training using the lists found in abstractnouns.txt and concretenouns.txt. It then uses this trained classifier to sort the topics that it gets from the get_theme function. Because themes tend to be abstract, it then returns all the topics identified as abstract as potential themes (in the same order as the input).

- ◆ *Input:* words *(topics extracted from the lyrics in get_theme)*
- ◆ *Output:* themes *(themes as determined by the function)*

➔ *def get_song_sentiments*:

This function uses the TextBlob library to calculate the sentiment polarity and subjectivity for each song in the input list based on its processed lyrics. It then assigns them a category based on the returned value:

Polarity: value [lower bound, higher bound)

-3 [-1, -0.5), -2 [-0.5, -0.25), -1 [-0.25, -0.05), 0 [-0.05, 0.05),
1 [0.05, 0.25), 2 [0.25, 0.5), 3 [0.5, 1]

Subjectivity: value [lower bound, higher bound)

1 [0, 0.1), 2 [0.1, 0.2), 3 [0.2, 0.3), 4 [0.3, 0.4), 5 [0.4, 0.5),
6 [0.5, 0.6), 7 [0.6, 0.7), 8 [0.7, 0.8), 9 [0.8, 0.9), 10 [0.9, 1]

Finally, it passes these lists to the get_song_polarity and get_song_subjectivity functions and compiles their outputs to return.

- *Input:* all_songs
- *Output:*
  - img_polarities *(distribution of polarity values of artist's songs)*
  - img_subjectivities *(graph of subjectivity values of artist's songs)*
  - subjectivity_rating *(weighted average of song subjectivity ratings)*
  - polarity_verdict *(average of individual song polarity ratings)*

➔ *def get_song_polarity*:

Generates a graph with matplotlib based on the categorical values assigned in get_song_sentiments. Generates a verdict based on the overarching polarity of the songs by calculating a weighted average (the number of songs assigned to a value (e.g. -3, -1) multiplied by the number of songs in that category, then summed across all categories to get the final verdict). The output categories of the verdict range from '---' (very negative) to 'o' (neutral) to '+++' (very positive).

- *Input:* polarities
- *Output:*
  - img_polarities *(distribution of polarity values of artist's songs)*
  - polarity_verdict *(average of individual song polarity ratings)*

➔ *def get_song_subjectivity*:

Generates a graph with matplotlib based on the categorical values assigned in get_song_sentiments. Generates a rating based on the mean of the subjectivity category values as the subjectivity rating (value from 1 to 10).

- ◆ *Input:* subjectivities
- ◆ *Output:*
    - img_subjectivities *(graph of subjectivity values of artist's songs)*
    - subjectivity_rating *(weighted average of song subjectivity ratings)*

➔ *def word_cloud*:

This first tokenizes the input pre-processed lyrics and extracts the nouns. The function then uses these nouns to create a word cloud with the WordCloud package from its library that is generated based on the word frequency.

- ◆ *Input:* lyrics *(processed lyrics from the output of process_lyrics)*
- ◆ *Output:* im *(image of word cloud of commonly seen words in lyrics)*

dash_app.py

This file serves as the backbone for a Dash web app interface, where users input their preferred musical artist, and the information provided by the spotify.py and genius.py files is presented in a user-friendly format. The interface consists of two main layouts: layout_search, where the search bar serves as the landing page for inputting the artist's name and layout_result, where the outputs are compiled for easy user viewing.

The layout_search encompasses the landing page where users can input an artist's name. The name of the artist must be in English and the artist's songs must have lyrics; artists with non-English names or no lyrics in their songs will not be available. Once the artist's name is entered, the layout_result displays the artist's image, a word cloud of their songs, the distribution of song polarity, subjectivity rating, and the distribution of song subjectivity. These outputs are organized into 4 tabs, ensuring a seamless user experience without the need for excessive scrolling.

The two main functions, 'process' and 'display_page,' play crucial roles in the functionality of the web app. The 'process' function invokes the primary functions from the spotify.py and genius.py files. It retrieves the queried artist's name from the search bar input and passes it as an argument to the main functions of both files. Once these functions complete their execution and process the outputs, the Dash application redirects the user to the "/result" page.

The 'display_page' function updates the output results, including the latest images and themes generated from the relevant python files. To avoid errors, there is always a default file available for retrieval, typically the result of the user's last search. This default file reloads whenever the "/result" page is accessed.

Folders

- Artists/
    - *Titles/:* contains txt files of titles of the songs used for the searched artist
    - *Lyrics/:* contains txt files of lyrics of the songs used for the searched artist; these lyrics are already processed by the code prior to saving
    - *artists.txt*: list of artists that the database has titles and lyrics for

- Scripts/
    - *Figures/:* Figures like the word cloud image, polarity and subjectivity distribution graphs, and theme lists are saved here for access by the Dash application.
    - Lists/
        - *abstractnouns.txt*: abstract nouns used for Logistic Regression classifier training in the get_abstract function in genius.py
        - *concretenouns.txt*: concrete nouns used for Logistic Regression classifier training in the get_abstract function in genius.py
        - *top100common.txt*: list of common words used in the English language used to eliminate non-meaningful words while determining themes for an artist’s lyrics in get_theme

3) Software Usage Installation

Code was developed and executed on Python3.8 and above. Use pip/pip3 as appropriate for your system/terminal.

a) To begin, retrieve the code from our GitHub repository by using the "pull" command in your version control system.

The following steps assume that you are executing any terminal commands from inside the folder Final/Scripts/. If executing from another location, please adjust the paths to any named files as necessary.

b) After obtaining the code, proceed to install the required dependencies by executing the install requirements command (courtesy of pipreqs) in your terminal. If you find that you are missing any other basic packages, please install them as necessary.
   i) Command: *pip3 install -r requirements.txt*
   ii) Additionally, to install the relevant spaCy package, please run: *python3 -m spacy download en_core_web_md*

c) Next, ensure that you have created the "credentials.py" file as shown in the video, which contains the API keys for the Spotify API and the Genius API. The keys are available in the README.md file within the Scripts folder. Rename the variables as shown in the video. Place the file in the current folder (Final/Scripts/).

d) Finally, run the Dash app by running the "dash_app.py" file through your text editor or by running the command "python3 dash_app.py" or the equivalent command applicable to your environment in the terminal.

4) Team Member Contributions

- Hyo Sup Kim (Team Captain)
    - Scheduled weekly meetings and created weekly meeting notes from W5 to W16, which can be found in the project proposal.
    - Proposed project topic, established communication methods, rules and guidelines for effective teamwork.
    - Primary contributor of “spotify.py”, responsible for its components on the Dash app.
- Marcia Dai
    - Majority contributor of “genius.py”, responsible for most of its implementation on the Dash app.
    - Made significant contributions towards the establishment of the Dash app’s main structure.
- Chang Hun Park
    - Minority contributor of “genius.py”.
    - Contributor of “spotify.py”, specifically the “get_artist_face” function’s implementation in the Dash app.
    - Made early contributions to the framework of the Dash app’s main structure.