Nolan Baker
May 15th, 2023
Music Search
CPEG457

Music Search Engine

Motivation

Some people try to find new music, but there are only limited strategies, which include:

- Hearing somewhere
- Curated playlists
- Radio
- Recommendations

While these are great, sometimes users want to find extremely specific songs. Maybe they are thinking of a certain word, or a vibe, or an instrument used in a song. The idea behind my project would be to capitalize on this.

Goal

The goal of this project is to:

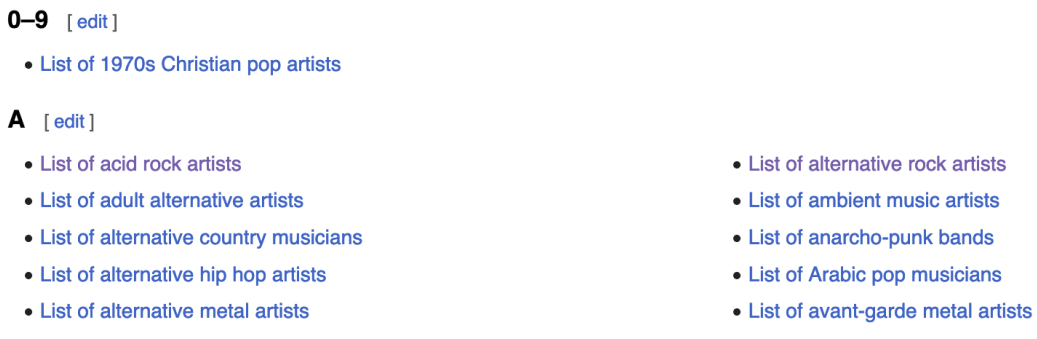
- Curate current top albums (according to genius.com)
- Allow users to search based on lyrics & song descriptions
- Output the link & name of the song

While it would technically be possible, and not difficult, to gather every listed artist & all of their albums, it would take a long time for the code to retrieve all of this data. Because of this, I wanted to use the top 2 albums from the top 20 artists from a-z. In the end, I only gathered around 100 artists worth of data, which is still enough data to search through for the purpose of this project.

Implementation

I started by creating the retrieval for the artists. I began by attempting to retrieve the data from wikipedia, but this was difficult for 3 reasons:

1. They were categorized by genre rather than by letter, which made it difficult to sift through which artists I would want to include:



0–9 [edit]

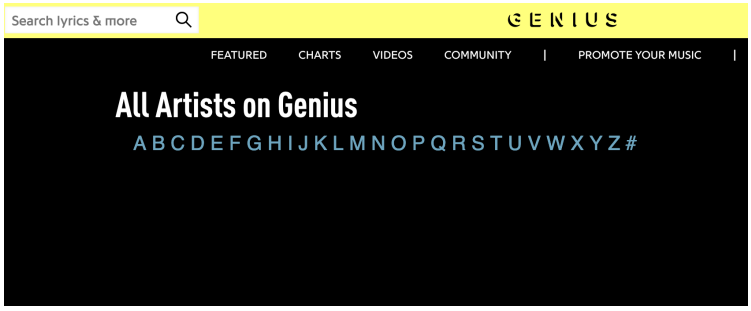
- List of 1970s Christian pop artists

A [edit]

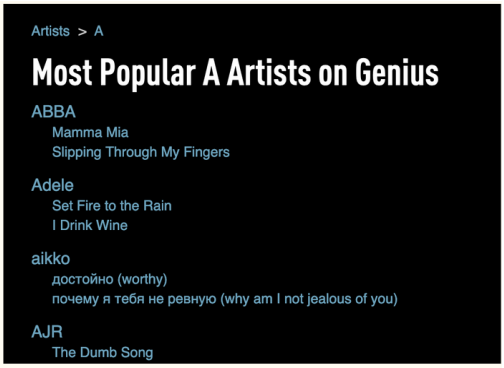
- List of acid rock artists
- List of adult alternative artists
- List of alternative country musicians
- List of alternative hip hop artists
- List of alternative metal artists
- List of alternative rock artists
- List of ambient music artists
- List of anarcho-punk bands
- List of Arabic pop musicians
- List of avant-garde metal artists

2. Due to the nature of wikipedia, each page could (and often do) have entirely different page structures, so it is difficult to create a scheme which would work unanimously.
3. There is no listing based on popularity, so I would have to sift through many unknown artists, many of which have very limited listed on their individual pages, in order to get the popular artists.

Because of this, I used genius.com's popular artist pages:



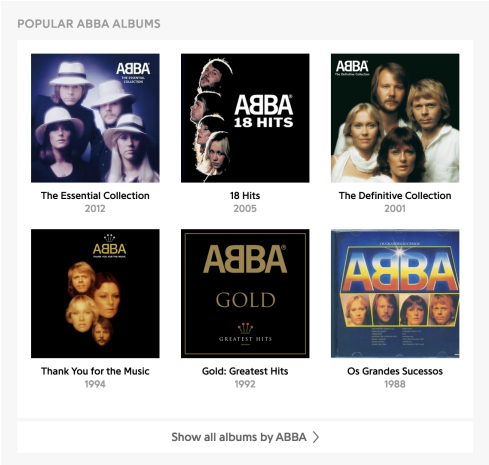
I would then sift through the top 20 artists listed on the page, which are categorized as "most popular":



```
# Extract the links for artists A through Z
a_to_z_links = [link['href'] for link in links if link.get_text() in
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"]

# Retrieve links to popular artists for each letter
popular_artists_links = []
for link in a_to_z_links:
   response = requests.get(link)
   soup = BeautifulSoup(response.content, "lxml")
   popular_artists = soup.select(".artists_index_list-popular_artist")
   popular_artists_links.extend([artist.a['href'] for artist in popular_artists])
```

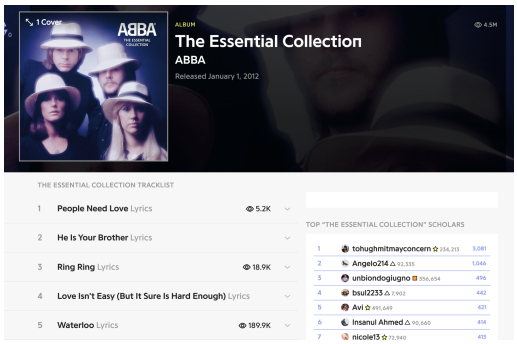
I then could sift through each artist page to get individual albums,




```
def retrieve_albums(artist_link):
   response = requests.get(artist_link)
   soup = BeautifulSoup(response.content, "lxml") #replace with html.parser if lxml
doesnt work
   album_divs = soup.select(".thumbnail_grid-grid_element")
   albums = []

   i = 0
   # Get links to top 2 albums per artist
   while len(albums) < 2 and i < len(album_divs):
       album_link = album_divs[i].find("a")['href']
       albums.append(album_link)
       i += 1
   return albums
```

And could retrieve the songs from each album:




```python
def extract_song_links(album_links): # Get link to each song within the albums
    all_songs = []
    for link in album_links:
        res = requests.get(link)
        soup = BeautifulSoup(res.text, 'lxml')
        song_links = []
        songs = soup.find_all('div', {'class': 'chart_row-content'})
        for song in songs:
            link = song.find('a')['href']
            song_links.append(link)
        all_songs.extend(song_links)
    return all_songs
```

And finally could get the lyrics & description of each song:

[Chorus: Björn, Benny, Agnetha & Frida]
People need hope, people need loving
People need trust from a fellow man
People need love to make a good living
People need faith in a helping hand

[Verse 1: Björn & Benny **and** *Agnetha & Frida*]
Man has always wanted a woman by his side to keep him company
Women always knew that it takes a man to get matrimonial harmony
Everybody knows that a man who's feeling down wants some female sympathy

As the majority of ABBA songs were, “People Need Love” was written and composed by the two male band members, Benny Andersson and Björn Ulvaeus. It was engineered by Michael Tretow who aimed to create a Phil Spector-like wall of sound on the recording. The song begins in the key of B major and modulates up to C♯ major for the final chorus.

Although their debut album did not bring them to global attention, it follows some standards of ABBA’s style. The ballad is about what people can give each other to make their lives easier and create a better world.

```python
def extract_lyrics(song_links): # Extract lyrics to songs
    all_lyrics = []
    all_links = []
    all_song_names = []
    for link in song_links:
        res = requests.get(link)
        soup = BeautifulSoup(res.text, 'lxml')
        try: # if song name not available, use blank name
```

```
            song_name = soup.select_one('h1[class^="SongHeaderWithPrimis__Title"]
span').text.strip()
        except AttributeError:
            song_name = ''

        # Extract the song description and lyrics
        song_description = soup.select_one('div[class^="SongDescription__Content"]')
        lyrics_container = soup.select_one('div[class^="Lyrics__Container"],
.song_body-lyrics')

        # Combine the song description and lyrics into a single value
        if song_description and lyrics_container:
            t = song_description.get_text(strip=True, separator='\n') + '\n' +
lyrics_container.get_text(strip=True, separator='\n')
        elif song_description:
            t = song_description.get_text(strip=True, separator='\n')
        elif lyrics_container:
            t = lyrics_container.get_text(strip=True, separator='\n')
        else:
            t = ''

        # Remove unecessary text
        t = re.sub(r'\n', ' ', t)
        t = re.sub(r'\[.*?\]', '', t)


        # Add the lyrics to the lists
        if t:
            all_lyrics.append(t)
        else:  # If lyrics not available, add blank lyrics
            all_lyrics.append('')
        all_links.append(link)
        all_song_names.append(song_name)

    return all_song_names, all_lyrics, all_links
```

I decided to use beautiful soup to retrieve the data from each page, as it is well documented online. The main difficulty in retrieving all of this data was finding where the information was held. Genius.com is obviously not a basic html page, so I had to differentiate between multiple listings of the same referenced link to find which one was relevant. Another issue I had with this was that, for some reason, the occasional song

was either not structured the same way or had no lyrics/description, so I had to add a check for this.

After creating these functions, I could then call them for each artist:

```
for artist_link in popular_artists_links:
   album_song_lyrics = []
   album_song_links = []
   print(f"Retrieving albums for {artist_link}:")
   albums = retrieve_albums(artist_link) # Retrieve the artist's album links
   songs = extract_song_links(albums) # Retrieve the album's song links
   album_song_names, artist_lyrics, album_song_links = extract_lyrics(songs) #
Retrieve the song names, song lyrics & song links for each song
   # Add lyrics, links & name to lists
   lyrics.extend(album_song_lyrics)
   song_links.extend(album_song_links)
   song_names.extend(album_song_names)
```

I stored the lyrics, links, and song names in 3 arrays. The index of each would be matched, i.e index 0 would be the first song, etc.

Due to the run-time of the algorithm, I decided to use the pickle library, which allows me to store the lists into files. This is nice as, if the program were to crash, I can have the current lists saved. This also means that the retrieval code does not need to be ran every time a user wants to search.

Next is the search code. I wanted to allow the user to search simply using their terminal & python (along with whatever libraries are needed). There are 2 parts: calculating the score for each word and searching through this score. To get the score, I used a TF-IDF pivoted normalization formula:

$$\sum_{t \in Q,D} \frac{1 + ln(1 + ln(tf))}{(1-s) + s\frac{dl}{avdl}} \cdot qtf \cdot ln\frac{N+1}{df}$$

To implement this, I first attempted to use pandas to create a data frame. While this worked, it was extremely slow. I researched online and realized that numpy could be faster, so I switched to using a numpy 2d array:

```
def rank(docs): # Function for calculating tf-idf of all songs
    words_dict = {}
    n_docs = len(docs)

    # Dictionary with words in doc
    for i, doc in enumerate(docs):
        words = doc.split(' ')
        for word in words:
            if word not in words_dict:
                words_dict[word] = len(words_dict)

    n_words_set = len(words_dict)  # Number of unique words

    # empty 2d array of size n_docs x n_words_set
    tf = np.zeros((n_docs, n_words_set))

    s = 0.2
    tot_dl = 0

    # Calculate average document length
    for i in range(n_docs):
        tot_dl += len(words)
    avdl = tot_dl / n_docs

    # Calculate term frequency
    for i, doc in enumerate(docs):
        words = doc.split(' ')
        dl = len(words)
        tot_dl += dl
        for word in words:
            word_index = words_dict[word]
            if tf[i][word_index] == 0:
                tf[i][word_index] = 1
            tf[i][word_index] = ((1 + np.log10(1 + np.log10(tf[i][word_index]))) / ((1
- s) + s * (dl / avdl))) if dl > 0 else 0

    #calculate inverse document frequency
    idf = {}
    for word, index in words_dict.items():
        df = np.count_nonzero(tf[:, index])  # Number of documents containing the word

        idf[word] = np.log10((n_docs + 1) / df)
```

```
    tf_idf = {}

    #combine term frequency and inverse document frequency
    for word, index in words_dict.items():
        tf_idf[word] = tf[:, index] * idf[word]
    return tf_idf
```

I also used a dictionary to store the words and the tf_idf, as it increased efficiency. I first stored each word in a dictionary, storing the number of unique words. I then calculated TF, which used the document length, average document length, as well as the term frequency, as well as a constant 0.2 I then calculated IDF, which simply used log(ndocs + 1 / df) Finally, I multiply the two values together for each word. I decided not to use query term frequency, as it would be more difficult to implement and not provide much for this use case.

Finally, all I had left was to allow the user to create a query:

```
def search_docs(tf_idf, query): # Function for allowing user to search through the
tf-idf
    # Split query into separate words & lowercase all searches
    query_words = query.lower().split()
    relevance_scores = np.zeros(len(docs))

    for word in query_words:
        if word in tf_idf:
            relevance_scores += tf_idf[word]

    if np.sum(relevance_scores) == 0:
        return {"Sorry, no matching songs were found with your query!": ""}

    indexes = np.argsort(relevance_scores)[::-1][:5]
    # Return the top 5 most relevant documents
    return {song_names[i]: song_links[i] for i in indexes}
```

I first lowercase all queries, as I wanted the searches to be case insensitive. I then create a score for each word that is searched compared to the tf_idf, and simply retrieve the index for the top 5, and finally return the top 5 name and links in order from most relevant to least.

<u>Evaluation</u>

Overall, the project works quite well. There are 2 primary issues in its current implementation:

1. The retrieval for lyrics takes quite a long time, at around 40 seconds per artists, which means, in an ideal situation, 40s * 20 albums per letter * 26 letters = 20800s / 60 ≈ 347 mins or over 4 hours. And, because of this,
2. Limited amount of songs. Ideally, I would want every song available, but obviously I do not have the same storage that genius.com has. Even despite this, there are many popular artists that aren't listed. (in letters more contentious such as k, compared to unpopular letters like q).

After the lists of lyrics are saved, the search is quite fast. I am happy about this as when I was using pandas it would take quite a long time, up to an hour.

Conclusion

Overall, the music search program I have created is, while very simply, definitely useful. If users want to easily search through Genius's list of music, without considering song names, this program could be more useful than their built-in search.

I am satisfied with my ranking, as it considers both the lyrics and descriptions. This means sometimes songs I wouldn't expect to be retrieved top their searches. I have found new songs I've never heard of by testing my program.

I am also satisfied with my retrieval, as, despite its lengthy retrieval time, it is able to effectively gather all of the relevant information.