Prof Zietz
INFO 3510

Discord Music Bot

Introduction:

For my final class project, I started on a significant expansion from my previous work. Reflecting on my earlier project, I realized I had focused too heavily on visual aspects rather than the broader functionality of the code. It was restricted to a single artist and a dataset I handcrafted, limiting its capability. While the code did do something cool it did not have a lot of use outside of this project alone

The moment we started learning Discord bots during our class, I found my project's direction. Considering my friends and I frequently use Discord, I saw an opportunity to explore bot coding further. Initially, it started with me exploring and playing with code personalized to our server. But after a few days of coding with Discord, I set my goal for this project: creating a Discord bot that could fetch a wide variety of music information with commands on my discord server using user inputs. I had to create a scope for this project given the time frame and my work outside of this class so I started small with aspects from my previous project and went from there. Moreover, the bot being part of a server my friends and I own made me spend a lot of my freetime exploring all types of functions that the bot could do. Many of the functions became tailored for this server and did not end up in the final project. The server became my testing ground—a place to conduct trials, gather feedback from friends, debug, and ensure diverse data inputs to cover as many test cases as possible. This project is for sure my favorite coding project that I have done and is the one I spent the most amount of my time on. I plan on continuing working on this project after this class ends as I will continue to tailor it to my personal server. For my next plans for my bot I will try to add a recommendation system using spotify API and fetching much more music information regarding music variables other than sentiment and subjectivity.

Analysis of my code:

- Imports and requirements:

In this analysis, I'll sequentially walk through my code, explaining each function and its purpose step by step. I've put all the debugging information into a separate dedicated section.
At the start of my code, there is a large list of imports and requirements for this bot to work. After coding with python for a few years, I've come to realize that imports and installations are necessary to learn as they give you a lot of tools to work with that makes coding easier. The more libraries and modules you're familiar with, the better you are off coding with python. Using imports and installations gives you so many tools to work with not only streamline coding processes but also speeds up your coding.

```python
import pip
import subprocess
subprocess.run(['pip', 'install', '-r', 'requirements.txt'])
import nest_asyncio
nest_asyncio.apply()
import asyncio
import vl_convert as vlc
import nltk
import pandas as pd
import csv
import re
import toolz
import io
import os
import altair_saver
import lyricsgenius
import requests
import altair as alt
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from nltk.stem import WordNetLemmatizer
from textblob import TextBlob
nltk.download('wordnet')
from lyricsgenius import Genius
import numpy as np
from dotenv import load_dotenv
import discord
from discord.ext import commands
from requests.exceptions import HTTPError, Timeout
```

```
textblob
nltk
discord
lyricsgenius
python-dotenv
altair
discord.py
altair_saver
vl-convert-python
```

I've included a set of requirements in the requirement file for installation using the subprocess import, this means that users do not have to manually install things which makes running the code much faster and easier. The libraries Textblob and NLTK serve purposes like lemmatization, sentiment analysis, and subjectivity analysis. Both discord and lyrics genius libraries are essential for executing Discord bot commands and interacting with the Lyrics Genius API. Installing altair is a safety measure to ensure the imported Altair version is the latest one. Additionally, I integrated discord.py for Discord API usage alongside Asyncio, which is used in the later parts of my code. The installations and imports of Altair_saver and vl-convert-python are from the Altair documentation which allows Altair chart objects to be saved as different files. These were crucial as the initial method taught in class for saving graphs didn't function properly. I've included two options due to potential issues with Altair_Saver, and vl-convert-python serves as a workaround to address these problems. The last import is just for checking connections for lyrics genius to make sure the connection between the server on jupyter and Genius's server.

```
load_dotenv('3510.env')
```

```
True
```

```
Token = os.getenv('DISCORD_TOKEN')
intents = discord.Intents().all()
```

```
try:
    Genius(Token)
except HTTPError as e:
    print(e.errno)    # status code
    print(e.args[0])  # status code
    print(e.args[1])  # error message
except Timeout:
    pass
```

```
lemmatizer = WordNetLemmatizer()

api_key = None

with open('API key.txt', 'r') as api_file:
    api_key = api_file.read().strip()
genius = lyricsgenius.Genius(api_key)

def process_title(title, artist):
    if not title.strip() or not artist.strip():
        return None
    song = genius.search_song(title, artist)

    if song:
        lyrics = song.lyrics

        lemmatized_lyrics = ' '.join([lemmatizer.lemmatize(word) for word in re.sub(r"\n{1,2}", ". ", re.sub(r"\?\n{1,2}", "? ", lyrics)).split()])
        sentiment_polarity = TextBlob(lemmatized_lyrics).sentiment.polarity
        return sentiment_polarity
    else:
        return None
```

This snippet handles token retrieval from both Discord and Lyrics Genius while also checking for potential connection issues between the server and Lyrics Genius. For the third cell of code It starts by fetching a song based on provided inputs like the song title and artist name. Following this, the lyrics undergo lemmatization, and newline characters are replaced with periods. Finally, sentiment analysis is conducted and the results are returned. Fortunately, all the functions I've written rely on libraries covered in our class sessions. Using code we wrote in class I was able to accomplish the code's function without any outside help.

```
def process_subj(title, artist):
    if not title.strip() or not artist.strip():
        return None

    song = genius.search_song(title, artist)

    if song:
        lyrics = song.lyrics

        lemmatized_lyrics = ' '.join([lemmatizer.lemmatize(word) for word in re.sub(r"\n{1,2}", ". ", re.sub(r"\?\n{1,2}", "? ", lyrics)).split()])
        subjectivity = TextBlob(lemmatized_lyrics).sentiment.subjectivity
        return subjectivity
    else:
        return None
```

```
def fetch_lyrics(title, artist):
    if not title.strip() or not artist.strip():
        return None

    song = genius.search_song(title, artist)

    if song:
        lyrics = song.lyrics
        return lyrics
```

In the first code cell above, the process of fetching the subjectivity of songs is pretty much the exact same as the sentiment function. The only difference that is between the two is that one

has sentiment.polarity vs sentiment.subjectivity. The second cell above fetches lyrics from an inputted song and artist which took me very little time to do.

```
def lyrics_finder(lyrics):
    request = genius.search_all(lyrics)
    for hit in request['sections'][2]['hits']:
        return hit['result']['title'] + ' by ' + hit['result']['primary_artist']['name']
```

```
def album_gets(album, artist):
    search_query = f'{album} {artist}'
    album_search = genius.search_albums(search_query, per_page=1)
    if 'sections' in album_search and album_search['sections']:
        first_section = album_search['sections'][0]
        hits = first_section.get('hits', [])
        if hits:
            first_hit = hits[0]
            result = first_hit.get('result', {})
            album_id = result.get('id')
            album_cover = result.get('cover_art_url')
    album_tracks = genius.album_tracks(album_id, per_page=50)
    songs = []
    if 'tracks' in album_tracks:
        for track in album_tracks['tracks']:
            song_title = track.get('song', {}).get('title')
            songs.append(song_title)
    if songs == []:
        return [], None
    return songs, album_cover
```

In the first cell above, the function retrieves a song containing lyrics based on user inputs. Unfortunately I was not able to make this function not case sensitive as it requires proper punctuation and spelling for it to work. The formatted and punctuated string required by Genius makes case insensitivity a bit challenging to implement but is a goal I have in the future. Moving to the second cell, the code fetches the album cover art, album ID, and every song from the album, then returns this information. Retrieving data from Lyrics Genius gets a JSON file format, although I am not the biggest fan of using JSON files due to their variability and complexity to work with. Luckily all the information I needed was on one file that wasn't super difficult to work with as all the keys were presented properly.

```
def visualizer(songs, artist):
    sentScores = []
    subjScores = []

    for song in songs:
        sentScores.append(process_title(song, artist))
        subjScores.append(process_subj(song, artist))

    data = pd.DataFrame({
        'song_title': songs,
        'sentiment': sentScores,
        'subjectivity': subjScores})

    avg_sent_score = data['sentiment'].mean()
    avg_subj_score = data['subjectivity'].mean()

    visual = alt.Chart(data).mark_rect().encode(
        x=alt.X("song_title:N"),
        y=alt.Y("sentiment:Q", title="Sentiment Score"),
        color=alt.Color("subjectivity:Q"),
    ).properties(
        title="Album visualization of sentiment and subjectivity of songs"
    )

    return data, visual, avg_sent_score, avg_subj_score
```

In my final function, which serves as the visualization segment of this project, I combined multiple functions, particularly for my !albumGetProp command. Within this function, I use almost all the other functions I have written before at once. Primarily, I gather all the songs from

an album along with their sentiment and subjectivity scores, writing this information into a structured data frame.After, I calculate the average sentiment and subjectivity score for the entire album. Lastly, I utilize Altair to generate a visual representation, using the DataFrame to generate a visualization that visualizes the album's sentiment and subjectivity scores. This visual representation of data can be easier to look at than a data frame full of numbers being sent to discord.

```
@bot.event
async def on_ready():
    print('{0} has connected to Discord.'.format(bot.user.id, bot.user.name))
    for guild in bot.guilds:
        for channel in guild.text_channels:
            if channel.permissions_for(guild.me).send_messages:
                await channel.send("Bot has joined the server!")
                await channel.send('Please type !start to get started')
                break
@bot.command(name="start", help="Instructions for bot")
async def rogan(ctx):
    message = (
        'This discord bot has an array of commands please use freely\n'
        'If servers time out please rerun\n'
        'Inputs are case sensitive so make sure you enter your inputs properly\n'
        'Please do not use multiple commands at the same time\n'
        'Some Commands may take a few minutes to fully send to discord\n'
        'lyricGet command only works with puncuation and proper spelling, have not figured out how to make it not case sensitive\n'
        'The commands for this bot are:\n'
        '`!songSubj`: This gives a song subjectivity score\n'
        '`!songSent`: Gives a song sentiment score\n'
        '`!songlyrics`: Prints song lyrics\n'
        '`!lyricGet`: gets song based of inputed lyrics and artist\n'
        '`!albumGet`: gets songs from an inputed album\n'
        '`!albumGetProp`: gets songs from an inputed album but includes properties and visualization\n'
        '`!albumGetCover`: gets album cover from input\n'
        'When using commands !songsubj, !songsent, !songlyrics, write the song name, then add a comma, and then the artist name\n'
        'When using commands !lyricget ,write the lyrics, remember it is caseSensitve and use proper punctuation\n'
        'When using commands !albumGet, !albumGetProp, !albumGetCover, write the Album name, then add a comma, and then the artist name\n'
        'Example !albumGetProp Graduation, Kanye West\n'
        'Example !songLyrics Eleanor Rigby, Beatles\n'
    )
    await ctx.send(message)
```

One of my first tasks was to ensure that whenever the bot joins a server or starts running, it sends out a signal confirming the successful connection to the discord server. This confirmation code acts as a separate alert that the bot is up and running. Additionally, I created a feature for new users wherein typing the command !start would provide them with a detailed explanation of the bot's functionalities. This more detailed guide is meant to assist users in understanding the bot's capabilities and how to interact with it effectively. While the !help command remains available for quick reference to the functions, !start was crafted specifically to offer a more in depth and instructive introduction to the bot's usage.

```
@bot.command(name="albumGet", help="gets list of songs from an album")
async def songlook(ctx, *, song_info):
    split_info = song_info.split(',', 1)
    if len(split_info) != 2:
        await ctx.send("Please use the format: Album Name, Artist Name")
        return

    album, artist = map(str.strip, split_info)
    songs,album_cover = album_gets(album, artist)
    if not songs:
        await ctx.send("Album not found.")
        return
    await ctx.send('The Album ' + album + ' include these tracks by ' + artist)
    for song in songs:
        if song is not None:
            await ctx.send(f"{song}")
        else:
            await ctx.send(f"Song names for '{album}' by {artist} are unavailable.")

@bot.command(name="albumGetProp", help="gets list of songs from an album with their sentiment, subjectivity scores, album cover, and viusalization ")
async def songlook(ctx, *, song_info):
    split_info = song_info.split(',', 1)
    if len(split_info) != 2:
        await ctx.send("Please use the format: Album Name, Artist Name")
        return
    await ctx.send("Processing may take a few minutes")
    album, artist = map(str.strip, split_info)
    songs,album_cover = album_gets(album, artist)
    if not songs:
        await ctx.send("Album not found.")
        return
    data, visual ,avg_sent_score, avg_subj_score = visualizer(songs, artist)
    if not data.empty:
        formatted_data = data.to_string(index=True)
        await ctx.send(f"Here's the data:\n```{formatted_data}```")
        await ctx.send(f"The average Sentiment Score of this album is: {round(avg_sent_score,2)}\n and the average subjectivity score of this album is: {round(avg_sent_score,2)}\n")
        await ctx.send(f"Album Cover:\n{album_cover}")
    else:
        await ctx.send(f"Album Properties for '{album}' by {artist} are unavailable.")

    if visual:
        png_data = vlc.vegalite_to_png(visual.to_json(), scale=2)
        with open("chart.png", "wb") as f:
            f.write(png_data)
        file_obj = discord.File("chart.png", filename="chart.png")
        await ctx.send("Album Visualization", file=file_obj)
```

To increase the bot's functionality, I made multiple commands with slight variations in their outputs. Although these commands are similar the outputs of each of them contain different levels of information. Not everyone might want the album cover, the album properties, and a visualization accompanying the album's song list. Additionally, "albumGetProp" might take a few minutes to execute, while "albumGet" is almost instantaneous. I put effort into incorporating extensive error handling to assist users in case of incorrect formats or if the album isn't found due to spelling errors or unavailability in the Genius database (although the latter is quite unlikely). Rounding the sentiment and subjectivity scores was implemented to improve the readability and presentation of the output sent out to users. Dealing with Altair for generating visualizations had some challenges, especially when sending out the visualizations. Fortunately, Altair's documentation provided helpful knowledge, specifically in saving charts as PNGs. It even recommended using the vlc saver instead of Altair saver due to potential webdriver issues, which I encountered during development.

```python
@bot.command(name="albumGetCover", help="gets Album Cover for inputed album")
async def songlook(ctx, *, song_info):
    split_info = song_info.split(',', 1)
    if len(split_info) != 2:
        await ctx.send("Please use the format: Album Name, Artist Name")
        return
    album, artist = map(str.strip, split_info)
    songs,album_cover = album_gets(album, artist)
    if not songs:
        await ctx.send("Album not found.")
        return
    if songs:
        await ctx.send(f"Album Cover:\n{album_cover}")
    else:
        await ctx.send(f"Album cover for '{album}' by {artist} are unavailable.")

@bot.command(name="songSent", help="fetch song print sentiment")
async def songlook(ctx, *, song_info):

    split_info = song_info.split(',', 1)
    if len(split_info) != 2:
        await ctx.send("Please use the format: Album Name, Artist Name")
        return

    song_title, artist_name = map(str.strip, split_info)

    sent = process_title(song_title, artist_name)

    if sent is not None:
        await ctx.send(f"Sentiment polarity of '{song_title}' by {artist_name}: {round(sent,2)}")
    else:
        await ctx.send("Song not found or lyrics unavailable.")

@bot.command(name="songSubj", help="fetch song print subjectivity")
async def songsubj(ctx, *, song_info):
    split_info = song_info.split(',', 1)
    if len(split_info) != 2:
        await ctx.send("Please use the format: Song Title, Artist Name")
        return

    song_title, artist_name = map(str.strip, split_info)

    subj = process_subj(song_title, artist_name)

    if subj is not None:
        await ctx.send(f"Subjectivity polarity of '{song_title}' by {artist_name}: {round(subj,2)}")
    else:
        await ctx.send("Song not found or lyrics unavailable.")
```

```python
@bot.command(name="songLyrics", help="fetch song print lyrics")
async def songlyrics(ctx, *, song_info):
    split_info = song_info.split(',', 1)
    if len(split_info) != 2:
        await ctx.send("Please use the format: Song Title, Artist Name")
        return

    song_title, artist_name = map(str.strip, split_info)

    lyrics = fetch_lyrics(song_title, artist_name)

    if lyrics:
        chunks = lyrics.split('\n\n')

        for chunk in chunks:
            await ctx.send(f"{song_title}, {artist_name}:\n{chunk}")
    else:
        await ctx.send("Song not found or lyrics unavailable.")

@bot.command(name="lyricGet", help="Searches for a song containing inputed lyrics")
async def getsong(ctx, *, song_info):
    get = lyrics_finder(song_info)
    print(f"Returned value from lyrics_finder(): {get}")

    if get:
        await ctx.send(f"Song name is '{get}'")
    else:
        await ctx.send("Song not found or lyrics unavailable.")

bot.run(Token)
```

The remaining code sections largely follow a similar structure in their functionality, with minor differences such as sending out lyrics. When I began working on this project over the Thanksgiving break, one of the initial functionalities I considered was fetching lyrics based on user input. However, Discord has limitations on the maximum character count for outputs. To fix this issue, I had to make a workaround wherein Discord sends out the lyrics in smaller chunks individually, a method we later covered in class as well. This approach allowed for the successful sending of lyrics without encountering Discord's limitations on character count for messages.

Debugging process and ChatGPT:

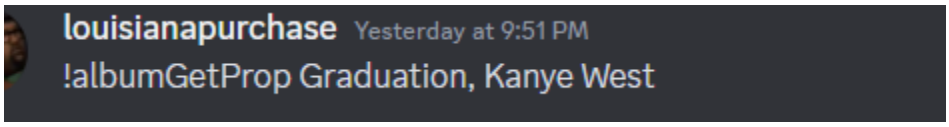
Although this project has been my favorite so far to work on it did not come without its frustrations and time spent debugging. This project like every other I worked on and will work on are reminders that most of coding is spent debugging instead of actually writing. I already talked about a few of the errors I got and my workarounds but I want to go more in depth. For most of my defined functions they were fairly simple and things we have already done in class however sending the visualization which is a major part of this project over discord was brutal. Unfortunately there wasn't a way to send charts that had interactive components but that did mean I had more time to figure out these weird web driver issues. There weren't many fixes on the internet that could tell me why it specifically was happening and even trying to debug with chatGPT led me nowhere. ChatGPT gave me recommendations like importing and installing things like selenium and imgkit however neither of those two fixes did anything. For some reason saving the Altair graph object as a png was a huge hassle and saving it like we did in class for matplotlib did not work at all. One of my failed attempts had me saving the chart as an html and then having the code screenshot the html and then send it to discord which again did not work. Finally I decided to dive more into the Altair library which is probably what I should

have done in the first place to find a solution. Altair had two ways to save their charts as a file that could be sent to discord. I tried their first solution of using Altair Saver which I thought would work because it is the one with their name in it and the one they showed first. However since it wasn't saving locally I got weird webDriver errors that I tried to fix by installing more imports but it still did not work. The Altair website then said that the second solution saves the file locally which I did not entirely want for a discord bot but it did work. Using the install Vl-Convert along with its import the chart object was able to be successfully saved and sent as a png file. Although I know we are not supposed to use ChatGPT in code generation I feel that my usage was conservative enough for it to be considered as a helpful debugging tool rather than code generation.

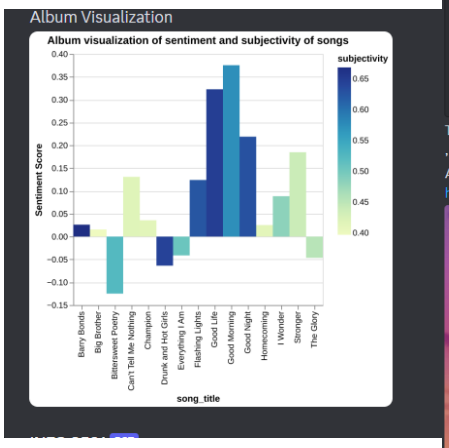
Example Usage:

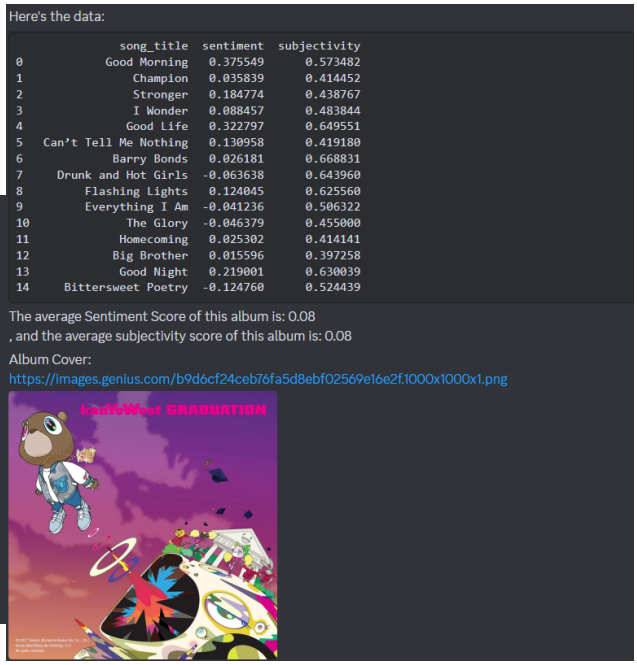
I want to show some example usage of my discord bot to show its capabilities a bit and how it works in real time just in case the user is confused.

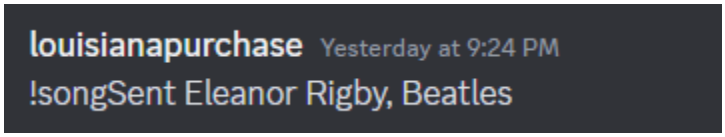
- Example 1



-
- When this input gets sent in discord, Calling !albumGetProp followed by the album name rather than a comma than the artist name the function should be able to properly use the imputed data to fetch and send information.
- The comma is necessary as that is how the code differentiates the album name with the artist, this works the same way with my other commands but it is the song title name and the artist.
- Because this code usually takes a few minutes to process due to my lack of knowledge in parallel processing the bot sends out a code letting the user know it may take a few minutes to send.
- These two outputs will be sent out into discord after the bot has finished with its searches.





- Example 2
- 

- If this input is written into the main text channel with the proper formatting the output will look like this
-