Shao-Min Yeh

Project Proposal

CS 410: Text Information Systems

Project Code Documentation

There are four different sections of my project: search engine model, similar songs model, sentiment analysis model, and Flask application. With this, each part will have these four or more parts.

1. **Code Function Overview**

This project revolves around enabling users to view different trends with Spotify songs. There are three different models that are implemented within a Flask application.

a. Search Engine: This model has a search query and outputs the 50 most similar songs based on TF-IDF weighting. This utilizes BM25 ranking utility and allows further utility with the option of separating title, artists, and lyrics (due to differing word lengths). Furthermore, a user can run a wide variety of queries to test various parameters, and this allows someone to see which results are most desirable with the various text files. Within the function itself, a user can run the model by changing the main constant values.

b. Similar Songs: This model allows a user to choose a song and finds the 25 most similar songs based on cosine similarity. This utilizes TF-IDF weighting within the song lyrics and musical features found within the dataset. There is utility to allow the user to select which features should be chosen, as TF-IDF and musical features can be enabled/disabled (which are also normalized to allow equivalent significance). Within the function itself, a user can run the model by changing the main song index and feature.

c. Sentiment Analysis: This model allows a user to choose a song and outputs the sentiment analysis, or valence, of a selected song. This utilizes a TF-IDF vectorizing model for the lyrics and various forms of linear regression to train the model. There is also utility with having different regression types, as Ridge/Lasso regression is faster and more accurate while linear regression is the classic model. Within the function itself, a user can run the model by changing the main variables.

d. Flask Application: This application is the enabler to run each model more seamlessly through a web application. Furthermore, this has the exact same utility with changing variables as running the models through the coding environment. This is done by explicitly calling each model separately through different route calls.

2. Software Documentation

Each portion will contain what each method does in the file along with a brief overview of the libraries used to implement each section (and how they interact with each other). Note that methods will be explained sequentially along with a step by step on how they interact.

a. Search Engine: This model utilizes Pandas for its dataframe, Metapy for BM25 ranking function, and Numpy for iterating through lists more efficiently when having the song lists. Lastly, there are config files that are created for separated/combined rankings, which are edited accordingly.

i. Preprocess Tasks: This is the preprocessing tool for the csv and dat files. It first calls Extract Columns to create a well-formatted csv. Then, it calls Make Corpus to create the necessary dat files. Lastly, it returns the file path to the csv.

ii. Load Ranker: This is the BM25 ranking function that is called for ranking. Also, this is the place to change parameters.

iii. Ranking: This is the ranking function that facilitates getting a query and scoring it. It makes an inverted index based on the config file listed, which allows separation from separated/combined rankings. It then loads the query and scores every document using Metapy. After that, we make a Numpy array with the corresponding indices representing the song index in Pandas and change the index value to the corresponding score found.

iv. Separated Ranking: This simply runs the ranking function three times for each config file (title, artist, lyrics). After this, the scores are added with the weighting applied to form the final aggregate list.

v. Results List: With the songs list made, we are able to descendingly sort the scores based on indices to return the highest scores. To do this, we maintain a song_indices array, which sorts the indices using argsort (0 is the highest index score). Furthermore, we maintain a set to delete duplicates. In the end, we iterate through indices until we reach the number of desired songs or until there are no more positively scored songs left.

vi. Print Results: This prints out the top_songs list when called. This is only necessary for the main function to see the list.

vii. Params Test: This is a function used to test a multitude of different queries to compare BM25 parameter values (k1, b, k3). This shouldn't need to be used by users except if they want to optimize the model. Firstly, a user needs to change the ranking parameters in Load Ranker and change the responding txt file in the main block. After that, we simply run the main block while appending to the new file based on the list of queries.

viii. Query Search: This is the main function that is called whenever the model runs, whether that's from the app.py or directly called. Firstly, it gets a formatted song by calling preprocess tasks. Then, it either calls separated ranking or ranking based on parameters. Lastly, it returns the resulting list.

b. Similar Songs: This model utilizes Pandas for its dataframe and Scikit-Learn for modeling (KNN, TF-IDF Vectorizer).

   i. Preprocess Tasks: Same as previous models with Pandas preprocessing.

   ii. Lyric Features: This is the model that finds the closest neighbors based on TF-IDF only. Firstly, the model is the TF-IDF vectorizer and transforms it along the lyrics dataset. Then, we use KNN to fit based on cosine similarity on the features. Lastly, we reshape the query features and return it along the KNN model.

   iii. Music Features: This is the model that finds the closest neighbors based on musical features only. Firstly, the model is the dataset features and transforms it along the lyrics dataset. Then, we use KNN to fit based on cosine similarity on the features. Lastly, we reshape the query features (based on the Pandas dataframe) and return it along the KNN model.

iv. <u>Combined Features</u>: This is the model that finds the closest neighbors based on both features. Firstly, the model appends both of the previous models based on hstack, which results in an array with both features normalized. Then, we use KNN to fit based on cosine similarity on the features. Lastly, we reshape the query features to an array and return it along the KNN model.

v. <u>Similar Songs</u>: This is the main function that is called whenever the model runs, whether that's from app.py or directly called. Firstly, it gets a formatted song by calling Preprocess Tasks. Then, we obtain the features according to the selected_feature variable. Lastly, it returns the resulting list.

vi. <u>Results List</u>: With this list of distances, we want to find the 25 closest songs based on the shortest distances. With this, we iterate through indices, which are already ordered correctly, and make a song_pair with the Pandas dataframe and index. Then, we add it to a set and add it to the song list if it doesn't exist. Lastly, we return the list when we have top_k songs.

vii. <u>Print Results:</u> This prints out the top_songs list when called. This is only necessary for the main function to see the list.

c. Sentiment Analysis: This model utilizes Pandas for its dataframe, Metapy for lyric tokenization, and Scikit-Learn for modeling (TF-IDF vectorizer, regression types, analyzers).

   i. Preprocess Tasks: Same as previous models with Pandas preprocessing.

   ii. Tokenize Queries: This is the tokenization function based on lyrics. Firstly, we reuse Metapy analyzers to do various tokenizing (lowercase, stemming, and stopwords). Then with those results, we iterate through the lyrics list and apply the tokenization methods. Lastly, we return the filtered lyrics.

   iii. Train Model: This is the model training based on TF-IDF and regression. Firstly, we find a train-test split, and we use Scikit-Learn with an 80/20 split. Then, we make a TF-IDF vectorizer to fit and transform the splits. Next, we use the model choice and fit the training sets. Lastly, we predict based on the test and return the predictions and MSE.

   iv. Compute Sentiment: This computes the sentiment based on the model found from the trained model. Firstly, we find the respective lyrics based on the chosen index. Then, we transform those lyrics to the TF-IDF model. Finally, we find the score by finding the predictions on the transformation.

   v. Compute Words: This computes the most significant words in the chosen lyrics. Firstly, we find the word names in the TF-IDF model. Then, we zip the index and data, which is the word index and score in a dictionary.

Lastly, we reverse sort based on the scores, such that the first index is the highest/most significant word.

vi. Result Items: This is the main function that is called whenever the model runs, whether that's from app.py or directly called. Firstly, it gets a formatted song by calling preprocess tasks. Then, we tokenize the lyrics if necessary and change the model to their respective name. Lastly, we call the functions to find all the respective items.

vii. Print Items: This prints out the most significant words, MSE, and scores when called. This is only necessary for the main function to see the list.

d. Flask Application: This is the web application that allows the website to work. Each model is imported directly and calls the "main" function. Then, Flask uses the HTML from templates folder, where we utilize the web pages. Note that HTML also uses static CSS for utility. Also, the following only explains how different types of routes are called, as there are many repetitive routes.

   i. User Queries/Home Pages: These simply call their respective HTML pages, where it redirects to the results page based on user choices. These are distinguished by very few lines of code that simply return a page.

   ii. Result Pages: These take the request data from the previous pages and call their respective models. Based on the main variables, we create new local variables within each method and find these when we call the model. Then, we pass these variables to the results list. Note that each method may have various types or if statements based on type necessities.

e. Preprocess Utility Function: This is a utility function that helps with preprocessing the datasets. This is necessary to create a well-formatted csv based on the Kaggle dataset and dat files for Metapy's config. With this, this is run before every single model to create a songs dataframe.

   i. Extract Columns: This extracts the columns from the dataset by removing unnecessary columns and non-English songs. Firstly, it sees if csv and dat files exist, and creates those files if they don't. If they don't, then we drop rows with null values, non-English songs, unnecessary columns, and delete non-ASCII characters. Lastly, we write to the csv file.

   ii. Create Dat File: This creates the dat files for Metapy by formatting them according to their type. Also, most dat files will be called one-by-one in the called function. Firstly, we open the csv file and skip headers. Then, we iterate row by row and write to the chosen dat file according to their format. We use formatted literals to the called dat file, such that lyrics dat only needs lyrics, combined needs all three features, etc. Lastly, we write the lyrics in lowercase and write that row to their respective dat file.

   iii. Make Corpus: This creates the corpus, which is essentially creating all four dat files. With this, we simply call create_dat_file for each path if it doesn't exist.

f. Song Search Utility Function: This is a utility function that aids users in finding a specific song index, which is necessary in finding the respective index for similar songs and sentiment analysis models. Thus, this function processes the songs by deleting duplicates and iterating through a Pandas database.

i. Preprocess Tasks: Same as previous models with Pandas preprocessing.

ii. Nonduplicated Dict: This removes the duplicates in the song data from the Pandas dataframe. Firstly, we keep a song dictionary to see if a title-artist tuple exists. Then, we iterate through the dictionary to find the title-artist tuple. Lastly, we add that index's pair to the dictionary if it doesn't exist (such that the key is the tuple and the value is the song index).

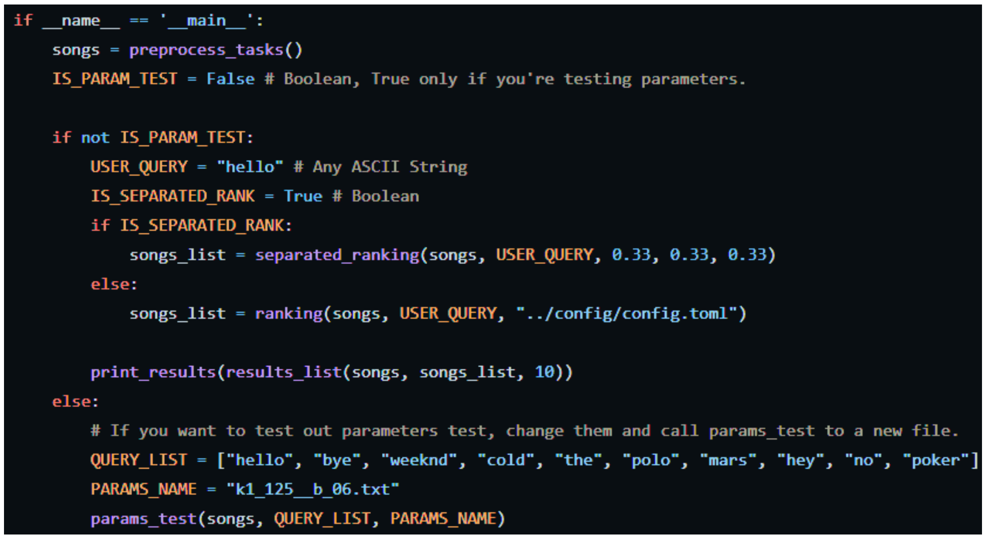
iii. Song Search: This is the function that calls the previous methods to find a song and song index by taking in an index and is_title, which signifies whether we're searching for title or artist. Furthermore, this is the function that the ML models call to find an index. Firstly, it calls preprocess and nonduplicated dict. With this, we iterate through each key in the dict to see if the user_query exists (using python in) the title or artist, depending on the is_title boolean. Lastly, we append the key to the list and return the list at the end.

3. Usage/Installation Documentation

A more structured documentation for installation exists on the Readme. Everything here needs to be done because of the Python version needing to be older for Metapy, so a separate Conda environment is being used along with ensuring correct library versions. Along with this, there is a summarized version of how to use the api calls in the Readme, but the following section is more detailed.

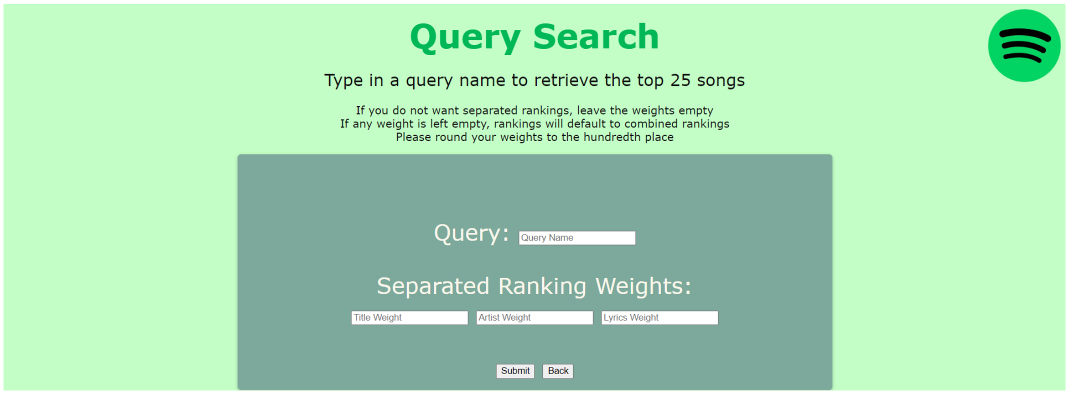
Note, this section will not go over the python calls, as every file can be called by running python <file_name.py> in a command prompt.

a. Search Engine:



```python
if __name__ == '__main__':
    songs = preprocess_tasks()
    IS_PARAM_TEST = False # Boolean, True only if you're testing parameters.

    if not IS_PARAM_TEST:
        USER_QUERY = "hello" # Any ASCII String
        IS_SEPARATED_RANK = True # Boolean
        if IS_SEPARATED_RANK:
            songs_list = separated_ranking(songs, USER_QUERY, 0.33, 0.33, 0.33)
        else:
            songs_list = ranking(songs, USER_QUERY, "../config/config.toml")

        print_results(results_list(songs, songs_list, 10))
    else:
        # If you want to test out parameters test, change them and call params_test to a new file.
        QUERY_LIST = ["hello", "bye", "weeknd", "cold", "the", "polo", "mars", "hey", "no", "poker"]
        PARAMS_NAME = "k1_125__b_06.txt"
        params_test(songs, QUERY_LIST, PARAMS_NAME)
```

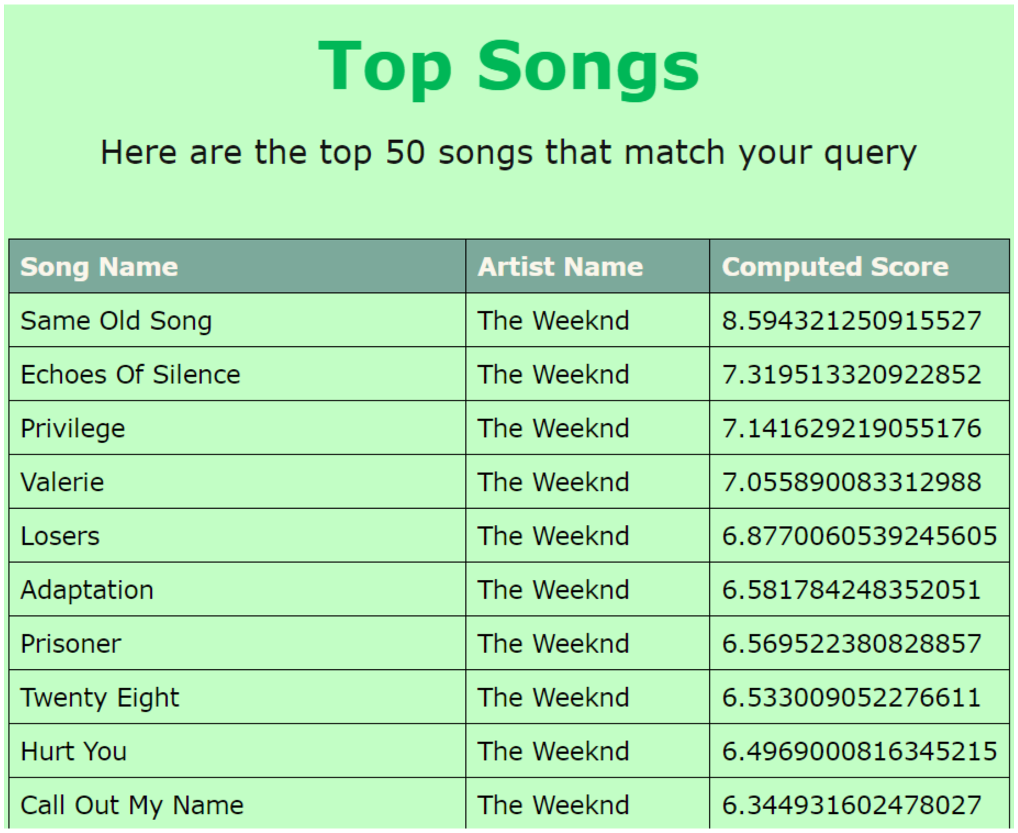
i. The search engine model constants are shown above, and the following changeable variables will be explained: IS_PARAM_TEST is a boolean variable that states whether we want a query list to test variables or the default search engine with singular queries, USER_QUERY is the search query used, IS_SEPARATED_RANK and the weights in separated_ranking allow utility of having rankings with artists/title/lyrics, QUERY_LIST is a list of queries to test across different parameters, and PARAMS_NAME is a text file that the results will be printed to. Note that a user will only have to change the variables within the if/else block based on IS_PARAM_TEST.



Query Search

Type in a query name to retrieve the top 25 songs

If you do not want separated rankings, leave the weights empty
If any weight is left empty, rankings will default to combined rankings
Please round your weights to the hundredth place



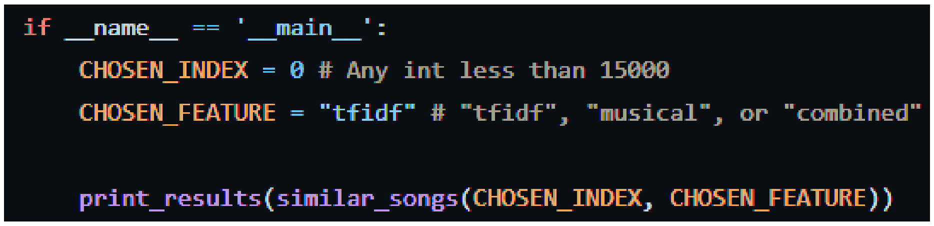
Top Songs

Here are the top 50 songs that match your query

Song Name	Artist Name	Computed Score
Same Old Song	The Weeknd	8.594321250915527
Echoes Of Silence	The Weeknd	7.319513320922852
Privilege	The Weeknd	7.141629219055176
Valerie	The Weeknd	7.055890083312988
Losers	The Weeknd	6.8770060539245605
Adaptation	The Weeknd	6.581784248352051
Prisoner	The Weeknd	6.569522380828857
Twenty Eight	The Weeknd	6.533009052276611
Hurt You	The Weeknd	6.4969000816345215
Call Out My Name	The Weeknd	6.344931602478027

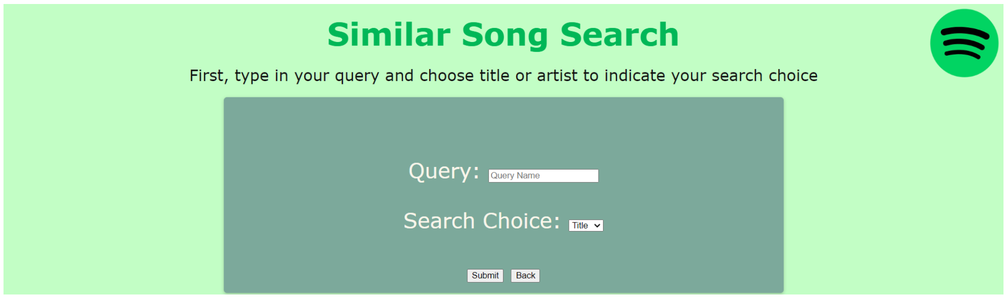
ii. As with the search engine model directly, query and weights can be edited in the boxes to model the search engine feature. In the second picture, we see the list of songs with title, artist, and score. One thing to note is that the params test is not implemented in the website, as only the search engine is needed here.

b. <u>Similar Songs:</u>



```
if __name__ == '__main__':
    CHOSEN_INDEX = 0 # Any int less than 15000
    CHOSEN_FEATURE = "tfidf" # "tfidf", "musical", or "combined"

    print_results(similar_songs(CHOSEN_INDEX, CHOSEN_FEATURE))
```

i. The similar songs model constants are shown above, and the following constants will be explained: CHOSEN_INDEX is the selected song index within the dataset and CHOSEN_FEATURE is the feature selection based on what should be included. The index doesn’t have the most utility, but a user can look at the dataset directly or run song_search.py to find such an index. The feature selection allows users to find songs that are similar sonically, lyrically, or both combined.

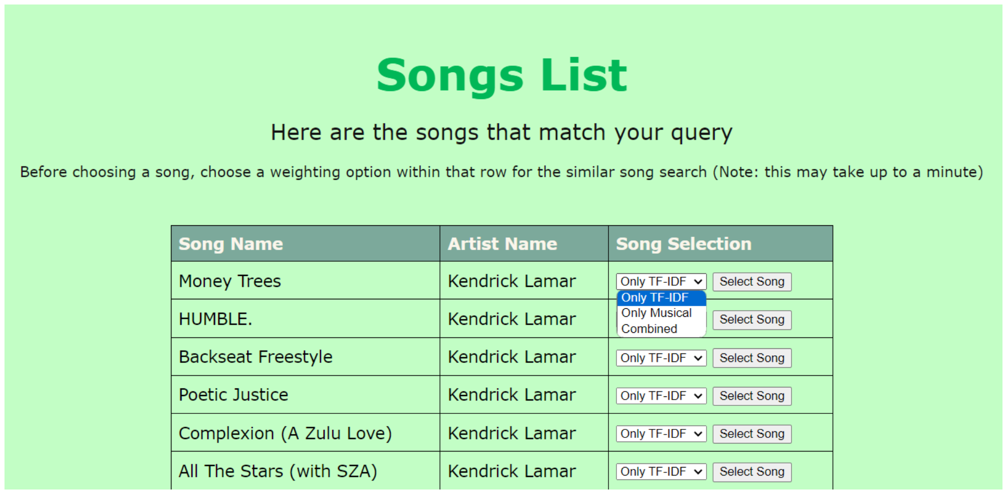


Similar Song Search

First, type in your query and choose title or artist to indicate your search choice

Query: Query Name

Search Choice: Title

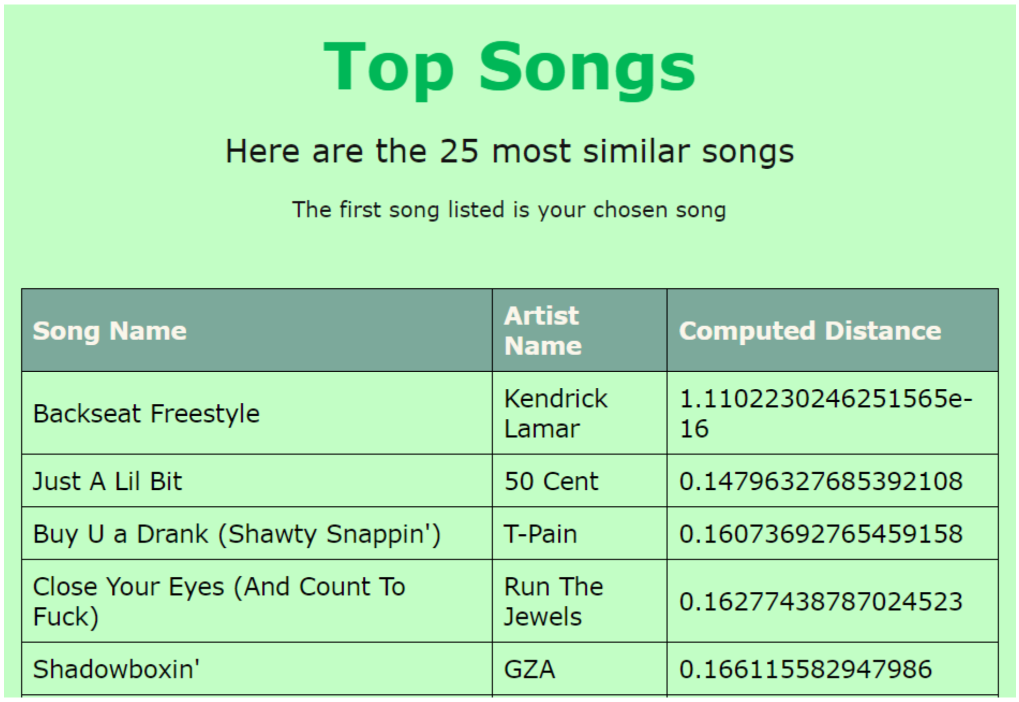


Songs List

Here are the songs that match your query

Before choosing a song, choose a weighting option within that row for the similar song search (Note: this may take up to a minute)

Song Name	Artist Name	Song Selection
Money Trees	Kendrick Lamar	Only TF-IDF / Select Song
HUMBLE.	Kendrick Lamar	Only TF-IDF / Only Musical / Combined / Select Song
Backseat Freestyle	Kendrick Lamar	Only TF-IDF / Select Song
Poetic Justice	Kendrick Lamar	Only TF-IDF / Select Song
Complexion (A Zulu Love)	Kendrick Lamar	Only TF-IDF / Select Song
All The Stars (with SZA)	Kendrick Lamar	Only TF-IDF / Select Song



Top Songs

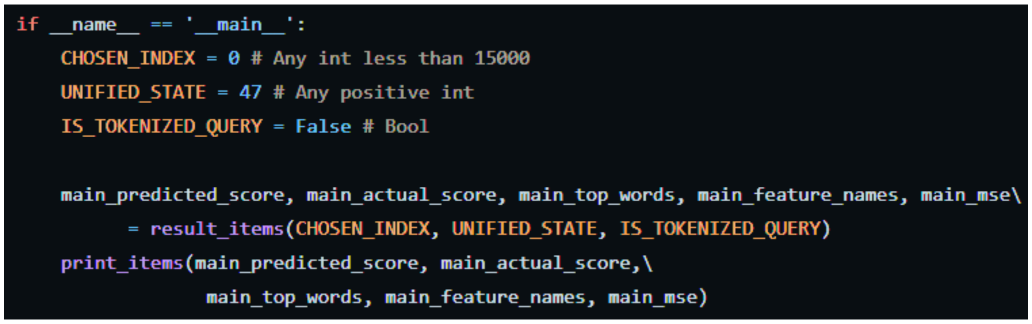
Here are the 25 most similar songs

The first song listed is your chosen song

Song Name	Artist Name	Computed Distance
Backseat Freestyle	Kendrick Lamar	1.1102230246251565e-16
Just A Lil Bit	50 Cent	0.14796327685392108
Buy U a Drank (Shawty Snappin')	T-Pain	0.16073692765459158
Close Your Eyes (And Count To Fuck)	Run The Jewels	0.16277438787024523
Shadowboxin'	GZA	0.166115582947986

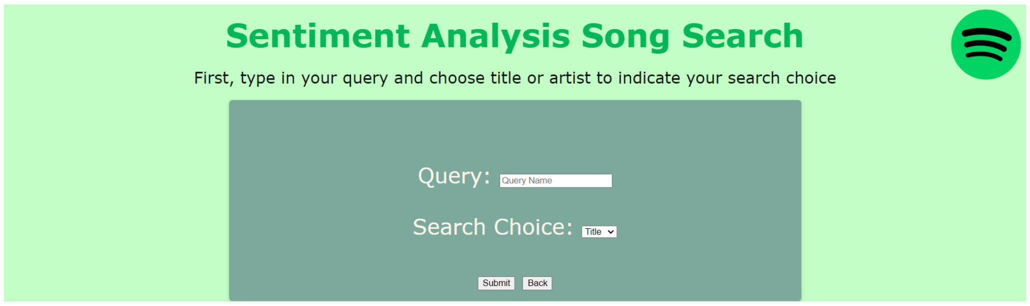
ii. With the similar songs model directly, query and title/artist search allows a user to find a song. With this, the second picture showcases the song search list, in which there are further options with a checkdown menu for the desired similar songs model. Lastly, the third picture showcases the similar songs with the distances listed from the chosen song. One thing to note is that the first song is the chosen song, which explains the super low distance score.

c. Sentiment Analysis:



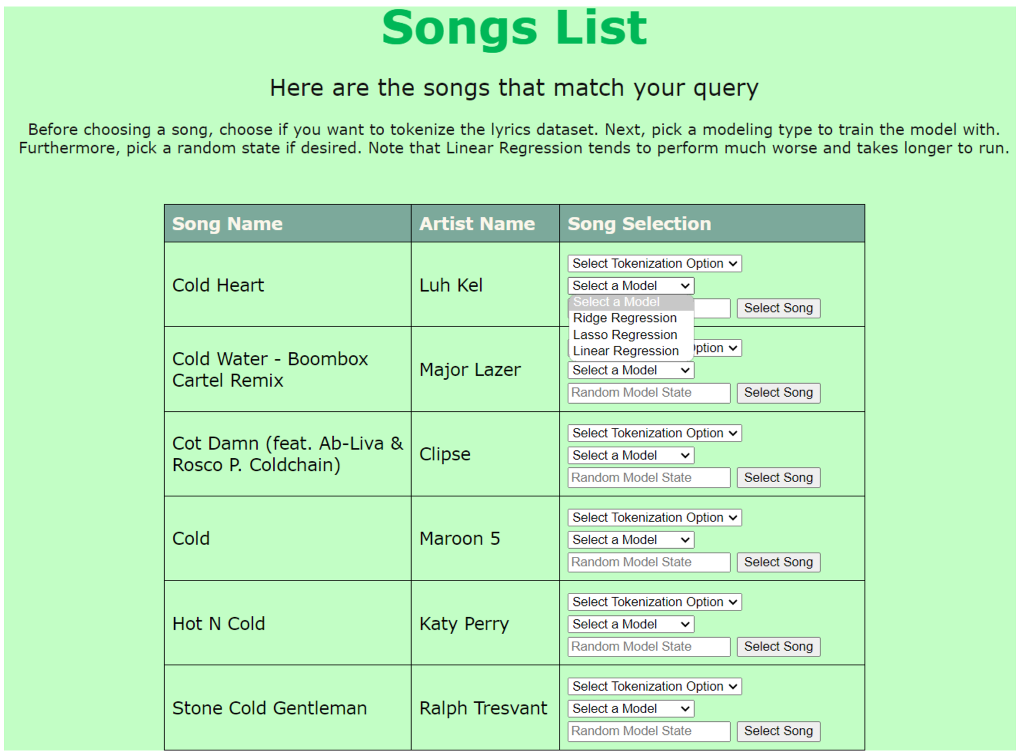
```
if __name__ == '__main__':
    CHOSEN_INDEX = 0 # Any int less than 15000
    UNIFIED_STATE = 47 # Any positive int
    IS_TOKENIZED_QUERY = False # Bool

    main_predicted_score, main_actual_score, main_top_words, main_feature_names, main_mse\
          = result_items(CHOSEN_INDEX, UNIFIED_STATE, IS_TOKENIZED_QUERY)
    print_items(main_predicted_score, main_actual_score,\
                main_top_words, main_feature_names, main_mse)
```

i. The sentiment analysis model constants are shown above, and the following constants will be explained: CHOSEN_INDEX is the selected song index within the dataset, UNIFIED_STATE is the randomized state to allow for the same model/results, IS_TOKENIZED_QUERY allows the user to tokenize lyrics, and MODEL_NAME allows the user to choose which regression type to use. Note that the model name and tokenization option both significantly change sentiment options and accuracy.



Sentiment Analysis Song Search

First, type in your query and choose title or artist to indicate your search choice

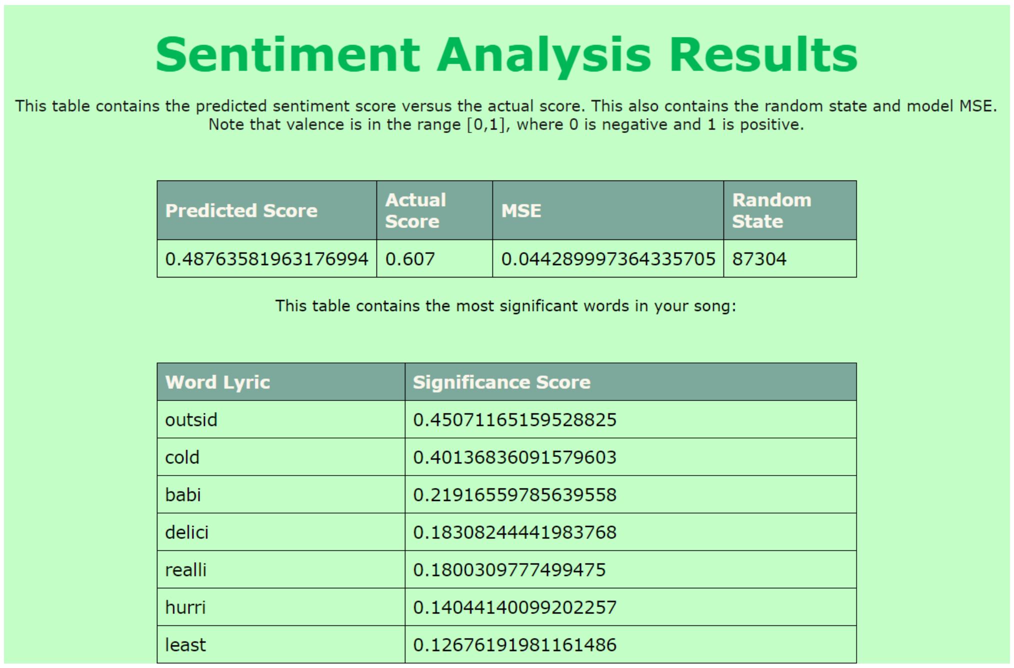


Songs List

Here are the songs that match your query

Before choosing a song, choose if you want to tokenize the lyrics dataset. Next, pick a modeling type to train the model with. Furthermore, pick a random state if desired. Note that Linear Regression tends to perform much worse and takes longer to run.

Song Name	Artist Name	Song Selection
Cold Heart	Luh Kel	Select Tokenization Option; Select a Model (Select a Model, Ridge Regression, Lasso Regression, Linear Regression); Select Song
Cold Water - Boombox Cartel Remix	Major Lazer	ption; Select a Model; Random Model State; Select Song
Cot Damn (feat. Ab-Liva & Rosco P. Coldchain)	Clipse	Select Tokenization Option; Select a Model; Random Model State; Select Song
Cold	Maroon 5	Select Tokenization Option; Select a Model; Random Model State; Select Song
Hot N Cold	Katy Perry	Select Tokenization Option; Select a Model; Random Model State; Select Song
Stone Cold Gentleman	Ralph Tresvant	Select Tokenization Option; Select a Model; Random Model State; Select Song



Sentiment Analysis Results

This table contains the predicted sentiment score versus the actual score. This also contains the random state and model MSE. Note that valence is in the range [0,1], where 0 is negative and 1 is positive.

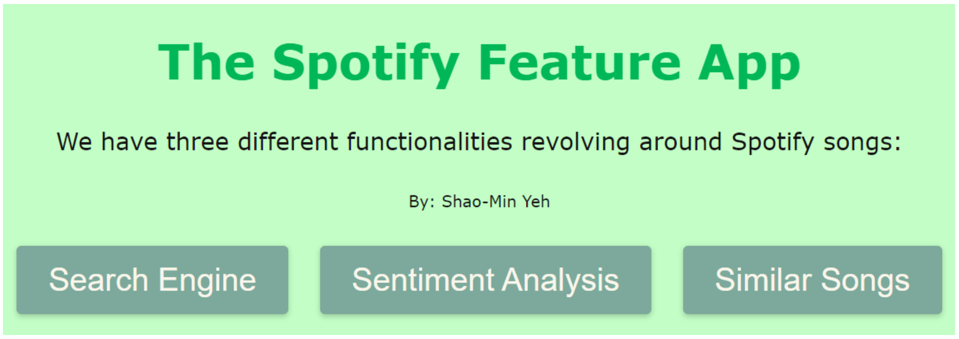
Predicted Score	Actual Score	MSE	Random State
0.48763581963176994	0.607	0.044289997364335705	87304

This table contains the most significant words in your song:

Word Lyric	Significance Score
outsid	0.45071165159528825
cold	0.40136836091579603
babi	0.21916559785639558
delici	0.18308244441983768
realli	0.1800309777499475
hurri	0.14044140099202257
least	0.12676191981161486

ii. With the sentiment analysis model directly, query and title/artist search allows a user to find a song. This is the exact same as the similar song model; however, there are different options in the song list (tokenization option, model type, and random state). Lastly, the third picture showcases the sentiment analysis along with the most significant words listed from the chosen song. There are many different stats in the last picture, and this allows for information regarding choices and insights. Note that the word list has “incorrect” spellings, but this is due to query stemming in Metapy.

d. Flask Application:



The Spotify Feature App

We have three different functionalities revolving around Spotify songs:

By: Shao-Min Yeh

i. The Flask application can be opened by running app.py, and the following home page appears. There are three different buttons, all of which signify the different models.

4. Team Contribution

This is an individual project.