Final Project Design

IMusic Maker

Team Members: Sanjana Prakash, Lingyi You

Ningke Hu, Tamara Prabhakar, Ruijie Cao

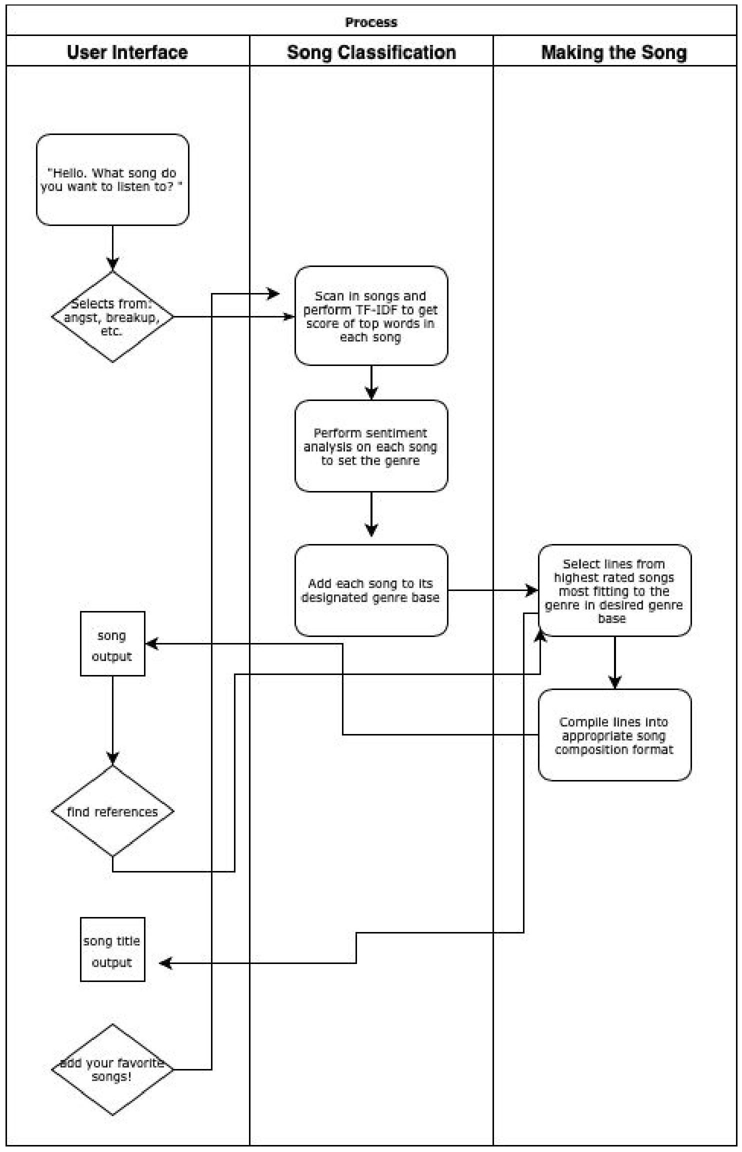
Executive Summary:

Our program is called IMusic Maker, a smart software that generates a brief song based on the user's desired genre input. Essentially, we start out with a database of songs lyrics and then then perform TF-IDF (which we will implement ourselves) to get a set of the most meaningful words in each song. We then take these words and compare them to a trainer set of sentiments as well as subjecting them to the Natural Language Understanding API provided by IBM/Watson to classify songs within categories of love, happy, breakup, yearning, angst, pain, melancholy, and rage. Once the user selects which type of song they want to generate (perhaps based on an emotion they are currently feeling), our program will fetch lines from that genre category in a randomized fashion and display it to the user. This will be the newly created song. The user interface will also display the original songs that these lines are referenced from.

Problem Description:

Our program solves the problem of classification. Based on a collection over one thousand songs, we classify and store each song by genre after analyzing its word sentiment. From each song, we analyze how well each of its sentences fits into its designated genre and rank all of the sentences. From this bank, we determine how to put together an exciting new song from preexisting lines to create the desired song type for the user.

User Flow:


Process
User Interface
Song Classification
Making the Song
"Hello. What song do you want to listen to? "
Selects from: angst, breakup, etc.
Scan in songs and perform TF-IDF to get score of top words in each song
Perform sentiment analysis on each song to set the genre
Add each song to its designated genre base
Select lines from highest rated songs most fitting to the genre in desired genre base
Compile lines into appropriate song composition format
song output
find references
song title output
add your favorite songs!

Overview of Steps:

1. Download songs/song lyrics from a database of many songs
2. Rank each word in the song according to TF-IDF.
3. Take the words with highest TF-IDF values and map them to the words in the trainer set for love (we will create this trainer list ourselves using words associated with “love”). We first decide if the song can be classified as a “love” song or “not love”. Then we send the songs in both categories through the Watson API and further classify them into 4 emotions. We compare these emotions to whether the songs were in the love/no love category and assign the song its final genre.

4. When the user inputs a song genre, the program will go to the genre storage and pull sentences randomly from the highest ranked song and return to the user as a newly formed song
5. Additional Features: If we have time, we can also add an additional feature where the user can add another song to the database (so they would have to enter a 'title' and 'lyrics' and we can store this in our song database). This would provide a chance to create more varied songs with new input. Another feature might be to have the song with the highest score in a category be played back if the user wishes to hear it (for instance, we could use a Napster/Spotify API to implement this feature).

How to decide the genre of a song:

1. Decide Emotion: For each song, it will be analyzed by IBM watson API to the percentage of 4 emotions(joy, sad, anger, fear). We will rank the percentage of emotions and tag the song by the highest proportion of emotion.
2. Determine Love or not Love by matching the top words from the song (probably around 15 words per song) with a love words trainer set. If the percentage of matched words to the size of topwords is above the threshold, the song is marked with Love. Else, it is marked with No Love.
3. Combine each emotion with love or no love to generate a genre (such as love, melancholy, pain, etc.).

GenreBase

	LOVE	NO LOVE
JOY	Love	Happy
SAD	Breakup	Melancholy
ANGER	Pain(sad)	Rage
FEAR	Yearning	Angst

TF-IDF algorithm:

<u>Purpose and Output:</u>

We use TF-IDF algorithm here to filter out unimportant words, like "or", "to", "for" and analyze the genre of the song based on the most important 15 words in the song. The output of TF-IDF algorithm is a list of **topwords** for each song based on its TF-IDF value.

<u>Implementation:</u>

- STEP1

Calculate term frequency for each word in each song:

Split each song into words and use a hashmap(named **termFreq**) for each song to map each word with its frequency. Count the number of words in each song(named **totalNumOfWords**).
For a word in a Song object s, TF(word) = s.**termFreq.**get(word)/ s.**totalNumOfWords**

- STEP2

Create a hashmap(named **words**) in Classifier to mapping each word to the number of songs with this word in it:
When we create a Song object, we will also put the word in **words**. If word already exists in words, we will add 1 to its value in **words**. If not, we will initialize the value of word to 1.

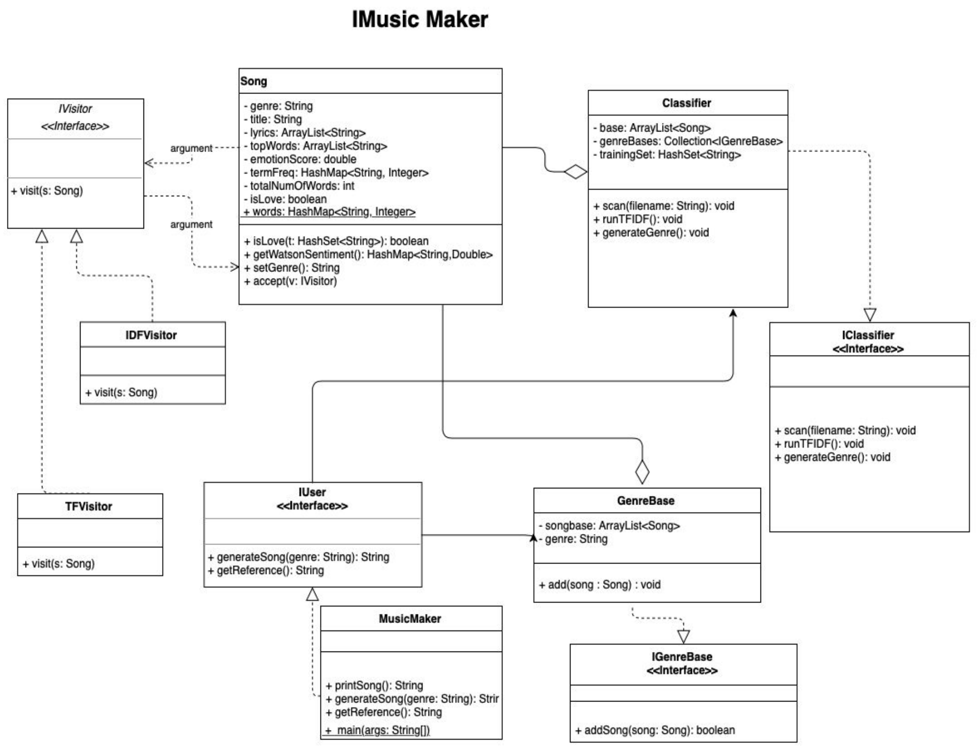
- STEP3

Calculate Inverse Document Frequency for each word:
After we create **Song** objects for all the song, for each word, we can get the number of songs with this word by **words.**get(word), and we can also get the total number of songs in **Classifier** by **base**.size() , **base** is an ArrayList in **Classifier** to store all the **Song**.
For classifier c, IDF(word) = log_e (c.**base.**size()/c.**words.**get(word))

- STEP4

Generate a list of **topWords** for each song by its TF-IDF value:
For each word in a song, TF-IDF(word) = TF(word)*IDF(word)
We get a ranking of all words in a song based on its TF-IDF value and choose top 15 words into **topWords**.

Class Diagram:



Class Diagram Description:

1. Song (Object): has *title*, *genre*, *lyrics* which stores in ArrayList line by line, *topWords* which contains top 15 important words in lyrics, *emotionScore*, *termFreq* mapping the words in the lyrics to their frequency, *totalNumOfWords*, isLove which indicate if this song is a love song, *Words* which is a static field to mapping every word in all songs to the number of song with this word in it.
2. GenreBase(Object): Will contain a database of the songs according to the genre they fall into. It implements IGenreBase. Ex: song base Love, song base Rage...
3. Classifier: has *base* to store all songs in the whole dataset, *genrebase* to store all GenreBase in a list and a love words training set. It reads in the song by *scan(),* analyzes it according to TF-IDF by *runTFIDF()*, determines if the song is a love song, assigns a count to emotion, combines the emotion with love/no love to see which ultimate category/genre that word falls into by *generateGenre()*. This class implements the IUser interface.

   ```
    /* Read file to get the song lyrics.
     Create a Song object for each song by it' title and the lyrics.
     Analyze the song by TF-IDF to get the top important words
    Train each song with love set and IBM watson API to decide its genre
     Add each Song into corresponding SongBase according to it's genre. */
   scan(filename: String)
   runTFIDF()
   generateGenre()
   ```
4. MusicMaker : writes and displays the newly created song in one genre by randomly picking several sentence from the corresponding GenreBase.

   ```
   /* return the new song lyrics*/
   String printSong();
   ```

Design Patterns: Visitor Pattern

When implementing TF-IDF algorithm on each song, we need to do two distinct operations on Song object: TF calculation and IDF calculation. To avoid "polluting" the Song class, we seperate TF and IDF calculation and move those operations to another class with the help of Visitor Pattern.

The Visitor Pattern consists two parts: Visit() and Accept()

- Visit(): TFVisitor overrides visit() method in IVisitor to implements a hashmap(named **termFreq**) and records **totalNumOfWords** for each Song object; IDFVisitor overrides visit() method in IVisitor to update a static hashmap(named **words**) in Song object.
- Accept(): Song class provide a accept() method to accept visitors.

The table below illustrates how each component in visitor pattern works in our project.

Client	**Classifier**
Visitor	<<Inteface>> **IVisitor**
Concrete Visitor A	**TFVisitor**
Concrete Visitor B	**IDFVisitor**
Visitable	**Song**