Mini Project: SoundBooth

Jinghu Lei & Rudhra Prakash Raveendran

I. Overview

SoundBooth is a social media platform for budding vocalists, where users can record and share short sound clips that others can vote on. Content is sorted into feeds by genre and then ranked by hotness score, both of which are derived from our machine learning algorithms. SoundBooth is currently available world-wide, accessible on mobile and desktop from https://onevone.club.

A. Problem Statement

We want our platform to provide an automated method for artists to spread their creativity. With automatic classification, artists do not have to manually select tags nor maintain a specific genre. Artists will also get automatic feedback on their music resulting in constant growth and better music reaching more people. The goal is that this will also flush out lower quality music and give people what they want to hear on their feeds.

B. Barriers of Entry

The barriers of entry for our application related to taking CS4242 are being able to effectively scrape, join, and clean text lyric data from various sources and vectorize it using bag of words and TF-IDF, as well as using different validation and testing techniques e.g. ensemble methods, SVR, nMSE. Furthermore, the class taught us the important of using cross validation to accurately gauge or results and make consistent predictions - using K-Folds and splitting features of data with an even distribution. Without taking the class, many of these steps can be overlooked or even not known.

II. Program Structure

A. Directory Structure

The majority of the important source code is contained within the *src* and *server* folders. The root folder is mainly dependency information, installation instructions, and hosting configurations. The *public* folder simply contains the HTML template that the front-end is injected into.

The *src* folder contains the code for the React front-end. Almost all the files are JavaScript (exceptions are CSS for styling and images used as static assets). The code is logically separated into concise modules within the *components* subfolder, and application constants are stored inside the *constants* subfolder.

The *server* folder contains the code for the Flask backend and machine learning algorithms, all written in Python. The Flask server is relatively simple, as all its code is contained within *app.py*. The classifier code is inside the *predict* folder.

The *data* folder contains all of the processed data used for training our machine learning algorithms. The raw data is initially stored in the subfolder *raw* within *data* - further details about data collection described below.

B. Server Technologies

The front-end was built using the React JavaScript library and the Bootstrap design library. The compiled static files are then served over a Python Flask application sitting behind a Gunicorn WSGI server with 4 workers. All API keys and application secrets are stored in environment variables so there is no risk of credentials leaking through source code. Currently the platform is hosted on Heroku.

React + Bootstrap was chosen for the front-end as this combination allowed for rapid prototyping of components with clean user interfaces, as well as for the myriad of existing open-source components that could be adapted to our needs. It also enforced a logical directory structure that makes the source code easy to understand.

Although React applications are typically served over a Node.js/Express.js server, we decided on Flask as it's written in Python, allowing for easy integration with our Python-based machine learning libraries - *Keras* and *SkLearn*.

Gunicorn is required for production-ready applications as Python servers such as Flask encounter poor performance under many simultaneous requests. Gunicorn (among other things) balances incoming requests among its workers, which each spawn instances of the Flask application as needed.

Finally, Heroku was chosen as the host as it's very easy to scale as the application grows (however at a certain point it would be more economical to switch over to baremetal servers). Furthermore, Heroku allowed for the easy configuration of a continuous integration pipeline. Whenever a new commit is pushed to the SoundBooth GitHub repository, Heroku automatically pulls down the changes, recompiles the front-end, installs dependencies as needed, and serves the application.

C. APIs

To keep track of users and their data, we used Firebase. Firebase handles all stages of user authentication (abstracting away the need to worry about safe credential storage), offers a realtime NoSQL database (for storing user profiles and post metadata), and provides an object storage service (where we store the soundclips themselves).

SoundBooth also makes use of the Google Cloud Speech to Text service. When a user submits a soundclip, a transcript

is generated using the Speech API, which is used in addition to the audio itself when predicting genre and hotness.

In addition to the Google APIs, SoundBooth utilizes the Spotify API to fetch genres and artists, which are used to populate the ”Favorite Artists” and ”Favorite Genres” input fields on profile pages.

D. Machine Learning

Our project had two use cases for machine learning, the popularity prediction and the genre classification. For both of the cases, the steps we needed to take were as follows:

- Data Collection and Preprocessing
- Training, Testing, and Validation
- Predict

The raw downloaded data would be stored in *data/raw* and the postprocessed data in the *data* folder under *genre_info* and *song_info*. The scripts to collect and preprocess the data are inside the data/genre and data/lyrics folders. The training and testing scripts are inside the *predict* folder under *train_**, where star would be the model. Finally the predict scripts, which are used as apis from our social media platform are also in the *predict* folder and called *predict_genre.py* and *predict_popularity*.

III. Social Media Platform

A new visitor to the application is only able to access the landing page; content feeds and user profiles require an authenticated user. After signing up/in, they'll be redirected to the home page, where content is displayed in feeds sorted by genre. The order of the content in feeds is determined by $h * max(1, log(S))$, where h is the predicted hotness score and S is the sum of +/- votes made by users (using $log(S)$ ensures that rankings aren't too volatile for new posts).

On the profile page, users can edit their bios, favorite artists, and favorite genres (information that is accessible to other users), as well as update their username or passwords on a separate settings tab.

The majority of the business logic for SoundBooth is contained on the Post page, where users submit content. There, users press the ”Start Recording” button which starts a timer and begins recording audio through their microphone. Once complete, users can listen to their sample and re-record as necessary. When ready, users can submit their soundclip after providing a title and brief description.

When a user presses submit, the recorded audio is first sent to Firebase storage and saved under a generated V4 UUID. Once it's saved, the download URL for the audio clip is retrieved and sent to our backend, which downloads the clip into a temporary file. Since the clip was recorded through the browser, it is encoded as a webm video, and so it's converted into a temporary wav audio file through a subprocess call to ffmpeg. The wav file is then sent to Google Cloud's Speech to Text API, which returns a set of possible transcripts. The transcripts and audio file are then passed to our classifiers, which predict genre and hotness rating. After deleting the temporary audio files, the predictions and transcripts are then returned to the client application, which saves them in the Firebase realtime database (along with other metadata like title, description, author username/uid, timestamp, etc).

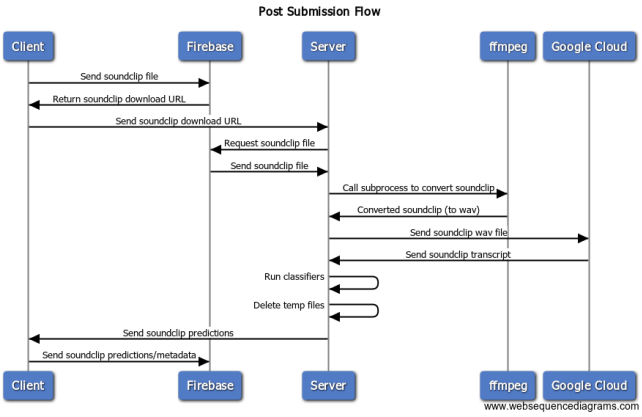

Fig. 1. Network Sequence Diagram for Submissions

Now, the soundclip post is available for all users to view. It will appear in the appropriate genre feed depending on its predicted genre. From there, users can listen to and upvote/downvote the soundclip. Since the number of listens and vote score for posts are stored in the Firebase realtime database, when any user listens to or votes on a post, all other users will immediately see the changes to the listen count or vote score. To ensure that the values for listen count and vote score are updated correctly, we use the Firebase transaction function, which ensures that updates are executed atomically.

To prevent malicious actors from burning through our Heroku dyno-hours or rate-limits for our various API keys, all onevone.club/api/ routes require a ?key query parameter (which is a randomly generated 25 character string). If a key is not provided or is invalided, requests will fail with error code 401.

IV. Popularity Prediction

Popularity prediction is a commonly commercial goal for predicting success of artists and their song. It can be either formulated as a classification problem or regression problem for a specific score. We chose to do the latter because we need individual scores to actually rank songs as per our use case. We wanted to have two different modals of features to experiment with: audio and text. Audio would be features such as tempo or key. Text feature would be the lyrics of the song. These features all had to be able to be generated with our personal tools since we wanted to predict entirely new samples of music as well for our platform.

A. Data Collection

Finding the right dataset was an extremely difficult task. Our first look was at the Million Songs Dataset[1] and the MusixMatch lyrics dataset however this proved too large (>280gb) for our use case. We instead found a Spotify dataset[2] of 19,000 songs with audio features and a hotness

score describing the popularity of the feature. However this was missing the text modality - lyrics - that we wanted alongside. We discovered that there is no easy way to gather lyrics data for songs due to copyright limitations. So, we chose to scrape lyrics from the Genius[3] site using the BeautifulSoup python library. As a back up we would scrape lyrics from AZlyrics. Out of the 19,000 songs we were able to scrape 13,081 song lyrics. Then we started cleaning and dropping columns of features that were not reproducible by us. For example *acousticness* and *speechiness* for which the Spotify algorithm was private. We were left with just *audio_mode* (major or minor), *key*, and *tempo* for the Spotify features and *lyrics* that we scraped. Finally, to clean the dataset, we eliminated duplicate songs and got rid of rows with any non-existant features.

Visualizing the lyrics dataset we see that ... and ... show up most often in popular songs. [[bar graph]]

For the audio features, the only one that makes sense to look at correlation for is the *tempo*. Calculating the correlation coefficient and sample covariance of *tempo* and *hotness*, we obtain -0.029748013 and -17.57160589 respectively. Both indicating a negative relationship. If we look at the scatter plot of hotness vs. tempo in figure 2, we see a more general lack of a relationship as all tempos correspond with all hotnesses. This might indicate that this feature will not be useful, which we will use as information in feature selection for the models.

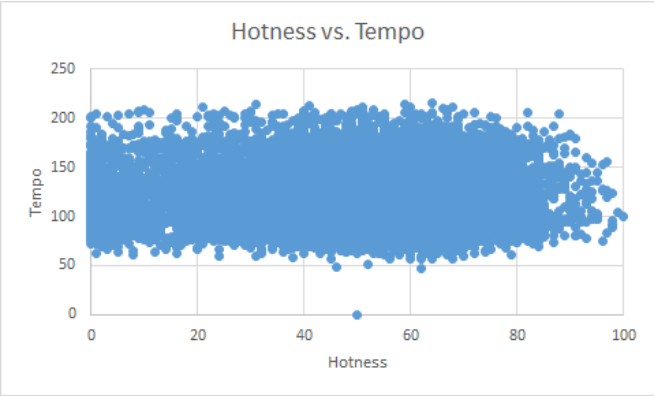

Fig. 2. Scatter plot mapping of tempo vs. hotness in Spotify Dataset

B. Text Preparation

For the features, we used a text-processer (from Lab 1) that turned the lyrics into a space separated string of stemmed words. Since, the lyrics included many languages, but mostly English and Spanish, we added stop words for both. The processor then stripped the string of punctuation and stemmed each word into its final format.

C. Training and Testing

For training and testing, we ran identical tests for each of the different models regarding features. Tuning was done separately. We used normalized Mean Square Error (nMSE) to assess the performance of the models and modifications. Validation is done through both a mean of 10 random data tests as well as 10-Fold cross validation.

D. Support Vector Regression

Support vector regression is a simple model to immediately test and see a base performance. We saw from the base model a nMSE of [!!!!]. Even after tuning it was clear that the SVR could not interpret the features to a good degree and we obtained almost no improvement on any feature changes. Results can be seen in table...

E. Results

1) list

Model	Average nMSE
Linear Regression	2.071262
Ridge	2.071911
Lasso LARS	2.123549
Bayesian Ridge	1.979979

Fig. 3. table

V. Future Work

As an additional feature, we wanted to add the user generated data into our training. However, this would require a lot of sample data and we were unsure if we could do this accurately. So we created a script to save the data to allow this in the future. We would have the classifiers to score soundclips re-run daily, as user-generated content is added to our training dataset. A daily Heroku cron job would make a POST request to https://onevone.club/api/repredict, which retrains the classifier with all available user-generated soundclips (with their vote score as a label for hotness), and then repredicts the hotness of every submitted soundclip. As more clips are posted to the platform and more users vote on them, the classifier will get better at predicting the hotness of new soundclips.

References

[1] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The Million Song Dataset. In Proceedings of the 12th International Society for Music Information Retrieval Conference (ISMIR 2011), 2011.

[2] Edwin Ramirez, *19,000 Spotify Songs*, https://www.kaggle.com/edalrami/19000-spotify-songs

[3] LyricsGenius, https://github.com/johnwmillr/LyricsGenius