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).

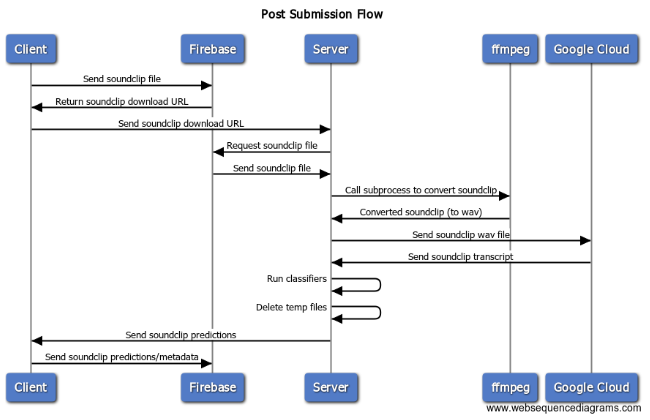
Post Submission Flow
Client
Firebase
Server
ffmpeg
Google Cloud
Send soundclip file
Return soundclip download URL
Send soundclip download URL
Request soundclip file
Send soundclip file
Call subprocess to convert soundclip
Converted soundclip (to wav)
Send soundclip wav file
Send soundclip transcript
Run classifiers
Delete temp files
Send soundclip predictions
Send soundclip predictions/metadata
www.websequencediagrams.com

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.

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.

Visualizing the lyrics dataset see figure 2 we see that a lot of them are common words that the stop words did not remove e.g. wa to was, im to I'm. The interesting words are like, know, love which usually tell some kind of story or emotion indicating a love or sad song.

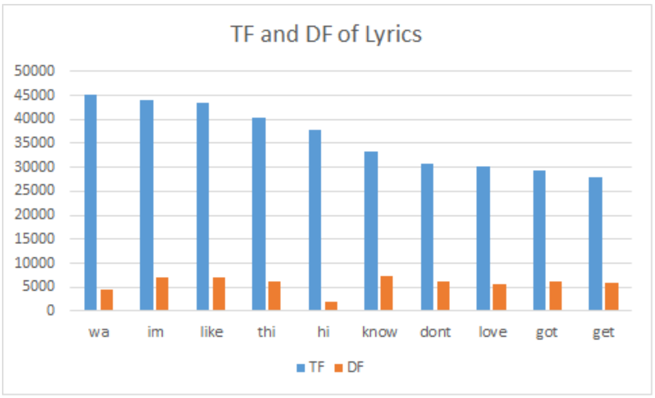

Fig. 2. Barplot of TF and DF of Most Popular Words in Lyrics

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 3, 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.

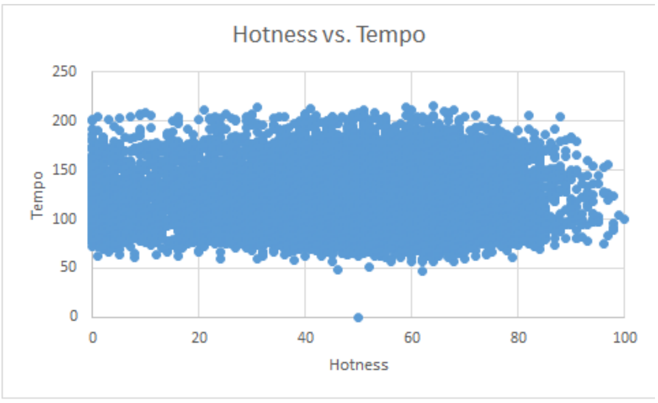

Fig. 3. Scatter plot mapping of tempo vs. hotness in Spotify Dataset

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.

For each model, we wanted to test the combinations of the modalities of the data so Audio + Text, Audio only, and Text only. For the Text (lyrics), we wanted to try both the CountVectorizer and the TfIdfVectorizer as they can both yield different feature representations. Next, we wanted to test the use of unigrams, bigrams, and trigrams because of the often phrasing that comes up in lyrics. Lastly, we tested feature selection using SelectPercentile, which chooses the top $N\%$ features based on the ANOVA (analysis of variance) score. By undergoing this final step, we reduce the computation time, choose more efficient features, as well as reduce chances of overfitting.

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 (audio features, TF-IDF, unigram, 100% features) that the nMSE was only 0.17843.

Testing only the audio features, with no lyrics, actually yielded a better result - 0.17263. This seemed to indicate the the immense nature of the lyrics was making the features more messy.

Further testing of variations of unigram, bigram, and trigram yieleded almost no change in results from the SVM. 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. This most likely indicates that the features are not doing much and the SVM is merely finding a way to minimize loss, such as finding a mean line between the *hotness*.

Support vector regression did not have great results and were stagnant despite feature changes and tuning. The final output is a nMSE of 0.167031.

Final results can be seen in table IV-H.

E. Kernel Ridge Regression

Kernel Ridge Regression is a modification of ridge regression to use the kernel trick, like Support Vector Machines. Ridge regression uses a L2 normalization, a different one from SVMs. The sole purpose of KRR is for regression unlike SVMs, which could also be used for classification.

Testing the base model already resulted in a better nMSE of 0.166820079 than SVMs, which was promising. However, by changing the modality of the feature space to only one of the features Text or Audio, the performance dropped by 12.5%. This leads me to believe that there is some importance of the relationship between the two feature spaces.

After testing TF-IDF unigrams, bigrams, and trigrams with the KRR, the results showed that trigrams were the best. This could be due to the longer phrases and more poeticness of songs - as compared to short texts like tweets.

However, when testing the CountVectorizer with the same unigram, bigram, and trigram set up, we saw signifcantly worse results. The nMSE went up to 4.799 for unigram count vectorizers and even up to 6.26581 for trigrams. It seemed that the CountVectorizer was skewing the data too much with trigrams, in contrast to TF-IDF which has a balancing effect of document frequency and text frequency.

> Kernel ridge regression varied more than SVMs although its final optimal performance was better. It achieved 0.15893764 nMSE.

F. Neural Networks

Neural networks seemed like a good model because of its ability to learn complex relationships between large feature sets. Our lyrics feature set boasts over 17,000 unique words so it definitely qualified. The base model outputted a nMSE of 0.178654, which is comparable to the last two. Once again when testing with no lyrics, the model achieved a better performance than the base. Surprisingly, the model almost had better performance with just the lyrics as well. TF-IDF with trigrams was once again the best tuning and had a nMSE of 0.14098. The nMSE was going down far more than the SVM and KRR, so we can see that the Neural Network was actually learning the feature set better. When tried with the CountVectorizer, a worse result also occurred - but not as much as the effect on the KRR model. Count vector unigrams had a nMSe of 0.5007 but trigrams only 0.311018. we thought that maybe more features might help the countvectorizer, so we tested 4-grams with the model. This actually decreased the performance to 0.40783.

> Finally, with tuning, the Neural Network model achieved a nMSE of only 0.138902 using 10% feature selection on the feature space.

G. XGBoost

Lastly, we wanted to try an ensemble method, as many random regressors could possibly average to a better score than just a single one.

The base model outputted a nMSE of 0.13990, which is already better than most of the other models. With no lyrics, it decreases again to 0.137416, another consistent pattern.

What was interesting is that testing unigrams, bigrams, and trigrams with TF-IDF did not change the result that much - it kept pretty consistent at 0.138. However, the CountVectorizer performed better using only unigrams and a tuning of 20% feature selection. This is highly contradictory to the results of the other models. Table IV-G details the tuning results.

> Further testing, shows that CountVectorizer would be the best for this model. A tuning of 20% feature selection would output a nMSe of 0.135856.

Percentile	**TF-IDF**	**Count**	**Both**
10	0.137081	0.138254	0.138968
20	0.138255	0.135856	0.136852
50	0.14054	0.137905	0.139256
80	0.138667	0.138679	0.139497
100	0.138649	0.138255	0.138432

TABLE I
TUNING RESULTS OF XGBOOST WITH SELECTPERCENTILE AND TF-IDF AND COUNT VECTORIZER

H. Results and Analysis

The best model out of the four we tested was the XGBoost model which does gradient boosting in an ensemble manner. It not only has a better computational speed, but also a better performance. I believe that this boost in performance is due to the ensemble nature of the model. It trains many gradient boost regressors on the data and averages them. Since the predictions of the regressors can be diverse, an average will be closer to the actual truth of the data. This will lead to a smaller nMSE. Thus we state our theory that given badly correslated data for a regressor, the model that produces the least spread from the mean will have the lowest nMSE. Figure 4 shows this in action.

Modification	**SVM**	**NN**	**KRR**	**XGBoost**
No Lyrics	0.17263	0.144932087	0.184904	0.137416
Lyrics Only	0.187654	0.1525354	0.187978	0.140083
TF-IDF Unigrams	0.173765	0.146378	0.168438	0.139903
TF-IDF Bigrams	0.168893	0.141220486	0.164078	0.138649
TF-IDF Trigrams	0.174897	0.140979627	0.162886	0.13843
Count Unigrams	1.54983	0.500767	4.799802	0.138255
Count Bigrams	1.37867	0.400154	1.249304	0.13983
Count Trigrams	5.48932	0.311018	6.26581	0.140412
TF-IDF + Count	1.121776	0.198025143	1.588329	0.138432

TABLE II
RESULTS OF POPULARITY PREDICTION

In figure 4, we can actually see the width of the spreads of predictions for each of the models. KRR has the highest spread and thus the largest nMSE, NN has a lesser spread and thus a less nMSE. Finally we see that XGB follows very closely along the average line, which is probably why it has the lowest. However, this simply shows us that the models are mostly just predicting the mean and not actually learning

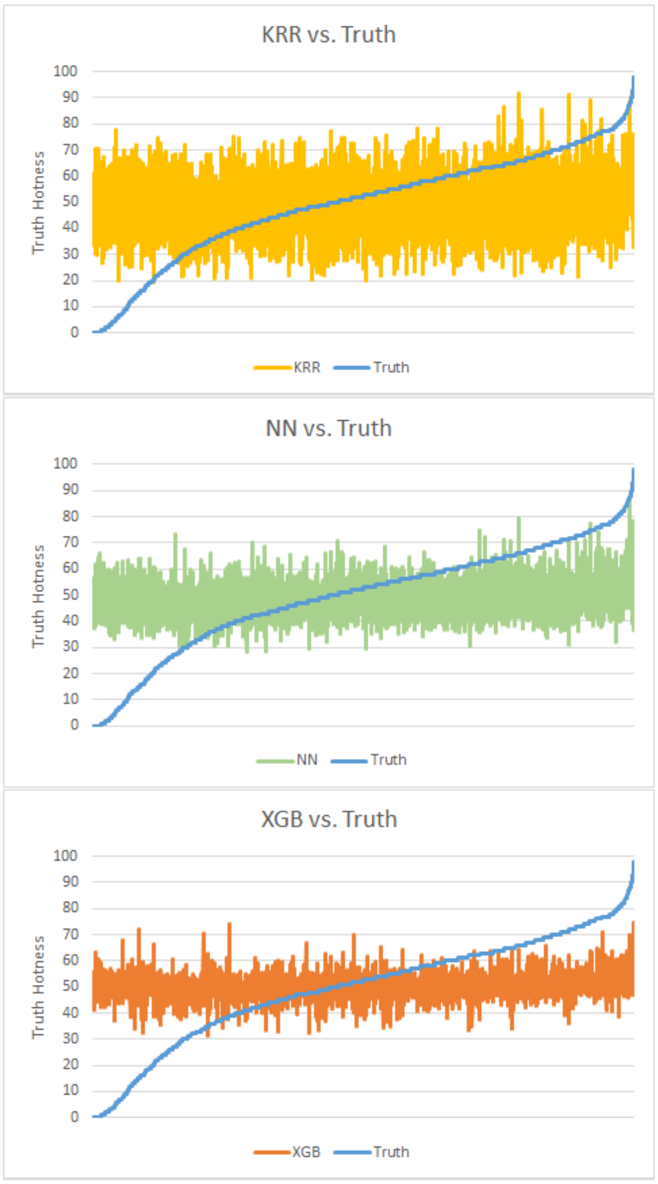

Fig. 4. Results of the individual models vs. the ground truth sorted in increasing order.

anything from the data. For future use, we need to reconsider our data and try to process the actual audio of the songs - although thats even harder to obtain than the lyrics.

I. Platform Integration

To integrate this predictor into our social media platform, we first extract the audio using librosa - a python module. We then obtain the simple mode, key, and tempo using that. Also, using google's speech to text api we get the lyrics of the audio. With those features put together we have the correct input into our predictor.

V. Genre Prediction

For our platform to function as described, we needed a genre classifier to automatically insert songs into their corresponding feeds. This is a much more researched task and we used inspirations from[4]. We would try two different approaches with the audio features: one, a simple neural network using the extracted features as a feature vector; two, a convolutions neural network that would take in the audio features as a spectrometer image.

A. Neural Network

For this model, we wanted to process actual audio data as per the paper[4]. We use the GZTAN data set of 1k songs, of 100 songs per genre. The genres are blues, classical, country, disco, hiphop, jazz, metal, pop, reggae, and rock.

For each of the songs in each genre, we extract the corresponding audio features[5]:

- Mel-frequency cepstral coefficients (MFCC)
- Spectral Centroid,
- Zero Crossing Rate
- Chroma Frequencies
- Spectral Roll-off

The MFCC, put simply, are the features that represent the unique units of sound portions made by sound generation. Mapping these are key to speech detection/recognition.

The Spectral Centroid is where the most mass for a sound is located. This centroid is calculated by taking the weighted sum of the frequencies around it.

The Zero Crossing rate is the rate of sign changes for the audio.

Chroma Frequencies is an representation of the audio where the entire audio spectrum is placed into 12 bins.

Finally, Spectral Rolloff is an approximation of the shape of the audio signal.

With these features, we want to input it into a standard feed forward neural network. We add 3 fully connected layers that finally output a vector of 10 - representing the 10 different genres we have. For our activation functions, we chose to use the industry standard RELU - which takes the max(0,n) for an input n.

Lastly, at the very end we have a softmax output layer, which basically normalizes the final output to add up to 1. We can then simply take the index of the max element in the array for our class label. Finally, we choose to use an adam optimizer - the standard stochastic gradient descent algorithm.

Simply running it with no extra tuning, we obtain a classification report at table V-A. The mapping of labels can be seen in table V-A.

We can see that just as a baseline, the model works well to classify the individual classes. It performs the best with classical and metal for which it has a f1 score of greater than 0.8. However, it performs worse for rock probably due to the messy and diverse nature of the genre.

Since our dataset only contained 1000 samples, and only 100 samples for each class, cross validation was much harder to do. Many of the folds would output F1 scores of 0 or N.A. due to not having any true/false cases for that example. Thus, what we did instead was take the average of multiple runs of the model.

label	genre
0	Blues
1	Classical
2	Country
3	Disco
4	Hiphop
5	Jazz
6	Metal
7	Pop
8	Reggae
9	Rock

TABLE III
LABELS FOR MUSIC GENRE

label	precision	recall	f1-score	support
0	0.64	0.58	0.61	31
1	0.76	0.86	0.81	29
2	0.62	0.56	0.59	32
3	0.7	0.54	0.61	35
4	0.52	0.57	0.54	30
5	0.68	0.58	0.62	26
6	0.88	0.86	0.87	35
7	0.64	0.82	0.72	28
8	0.56	0.64	0.6	22
9	0.52	0.53	0.52	32
micro	0.65	0.65	0.65	300
macro	0.65	0.65	0.65	300
weighted	0.66	0.65	0.65	300

TABLE IV
RESULTS OF THE NEURAL NETWORK FOR GENRE CLASSIFICATION

To tune this model, we wanted to test different hidden layer sizes and number of such - our default are 256, 128, 64. We also tuned the learning rate and added momentum to see if it was stuck at any local maximas. However, the percentage stayed relatively the same.

After retuning, the hidden layer sizes as well as well as the learning rate, we obtain the best weighted average f1 score of of 0.67. Final precision, recall, and f1 scores can be seen in table V-A.

label	precision	recall	f1-score	support
0	0.62	0.58	0.6	31
1	0.81	0.86	0.83	29
2	0.66	0.66	0.66	32
3	0.68	0.66	0.67	35
4	0.59	0.53	0.56	30
5	0.67	0.62	0.64	26
6	0.84	0.89	0.86	35
7	0.62	0.82	0.71	28
8	0.68	0.68	0.68	22
9	0.56	0.47	0.51	32
micro	0.68	0.68	0.68	300
macro	0.67	0.68	0.67	300
weighted	0.67	0.68	0.67	300

TABLE V
FINAL RESULTS FOR THE NEURAL NETWORK MODEL OF GENRE CLASSIFICATION

B. Convolutional Neural Network

The Convolutional Neural Network also seemed very promising for this use case, because audio features can be mapped to a spectral diagram which can be iterated over by the CNN. So instead of extracting the audio features into a list, we mapped it into the spectral image as can be seen in figure 5.

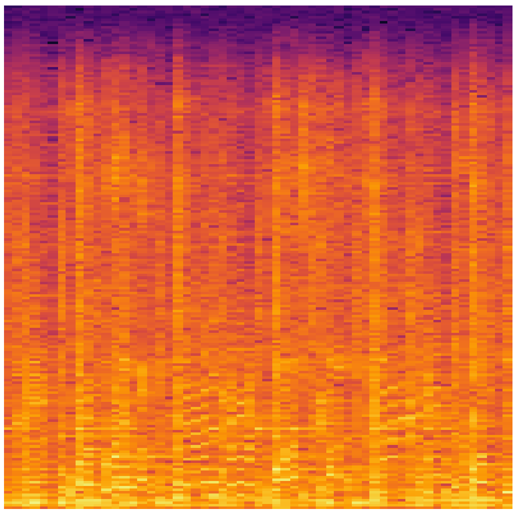

Fig. 5. Spectrogram of Audio Features for an Example Blues Piece

We can start to see the different pixels of the various pitches and changes in the piece.

For our model, we set up a convolutional neural network using Keras with an input size of 256x256x1, the image size, followed by 2D-Convolutional layers and pooling layers.At the very end of the model, we have a fully connected 500 weight layer which computes and outputs into a softmax layer, once again outputting one of the 10 classes.

Unfortunately, this model did not work as well as we had hoped for. I believe the computing system I had was not enough to handle the size of the memory necessary to fully train this model.

Results we obtained are detailed in table V-C.

C. Platform Integration

Our Neural Network model performed better so we chose to place that into our network. Using the same preprocessing library librosa, we extract the same audio features. We then input that into our neural network and get a prediction. Since we also get a confidence vector, we might choose to do a heuristic on the front-end that can override the decision if it is too unsure.

VI. FUTURE WORK

As an additional feature, we wanted to add the user generated data into our training. However, this would require

label	**precision**	**recall**	**f1-score**	support
0	0.32	0.88	0.46	26
1	0	0	0	24
2	0.27	0.49	0.35	35
3	1	0.03	0.06	30
4	0	0	0	28
5	0.23	0.64	0.33	33
6	0.62	0.81	0.7	32
7	0.78	0.49	0.6	37
8	0.33	0.03	0.05	34
9	0.5	0.05	0.09	21
micro	0.36	0.36	0.36	300
macro	0.4	0.34	0.27	300
weighted	0.42	0.36	0.29	300

TABLE VI
RESULTS OF CNN

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.

VII. Conclusion

We learned a lot from doing this project especially about the data necessary to train real machine learning models. Cross validation is still just as important to maintain these results. Our popularity regression model merely predicted the mean because of the extent of our feature space. Our genre classification model was much more successful, and also worked well on new test data. Our platform integrates these two models to provide a seemless experience for audio exploration, and rating to users.

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`

[4] G. Tzanetakis Dept. of Comput. Sci., Princeton Univ., NJ, USA, P. Cook, *Musical genre classification of audio signals*, `https://ieeexplore.ieee.org/document/1021072`

[5] Parul Pandey, *Music Genre Classification*, `https://towardsdatascience.com/music-genre-classification-with-python-c714d032f0d8`