song2vec: Determining Song Similarity using Deep Unsupervised Learning

CS229 Final Project Report (category: Music & Audio)

Brad Ross (bross35), Prasanna Ramakrishnan (pras1712)

1 Introduction

Humans are good at figuring out what similar music sounds like. In doing so, we develop a taste for music that we like. Computers, on the other hand, are less able to understand musical similarity. For example, music recommender systems like Spotify's Discover Weekly algorithm measure similarity by computing how often songs appear on playlists together[1] rather than using the music itself.

For this project, our goal is to use an unsupervised deep learning technique called Adversarially Learned Inference [3] to learn a metric embedding on songs and compare that embedding to several baseline metric embeddings that do not rely on deep learning, such as PCA. In other words, we want to learn a function that generates a lower-dimensional representation of an input song without making use of any labels for those songs. These lower-dimensional representations of songs make determining song similarity trivial and thus have applications to a wide variety of music analysis and recommendation problems.

2 Dataset

For the project, we collected a dataset of information about 793,442 tracks (Spotify's label for songs), 315,932 albums, and 191,887 artists from Spotify's Web API. This dataset is comprised of two principal components: song previews and object metadata.

2.1 Song Previews

The first and primary component of the dataset consists of 30 second audio segments (in the form of MP3 files) for 91.9% of the songs in the dataset. To convert these previews into objects that can be manipulated linear-algebraically, we take the 2D discrete Fourier transform of each preview. Doing so converts a preview into a *spectrogram*, a matrix of values corresponding to intensities of logarithmically spaced frequency ranges over discrete slices in time.

Of course, it isn't immediately obvious that these spectrograms are useful for determining the similarity between songs. However, as 2.1 demonstrates, electronic dance songs and orchestral pieces actually display very different frequency signatures, as intuition would suggest. While not all differences between genres are this stark, the comparison validates the use of the spectrogram as a signal-rich representation of audio data for our project.

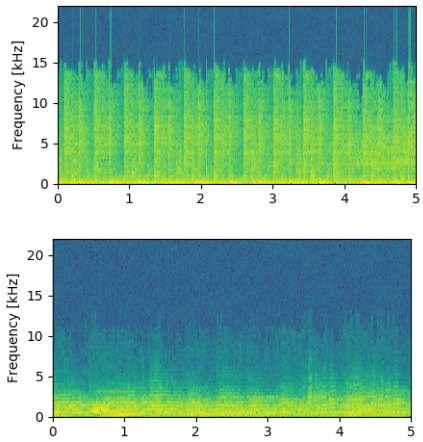

Figure 1: Top: a spectrogram for the first 5 seconds of the preview of an electronic dance song; Bottom: a spectrogram for the first 5 seconds of the preview of an orchestral piece.

2.2 Object Metadata

The second component of the dataset consists of metadata about the tracks, albums, and artists in our dataset. In particular, each track is associated with an album and one or more artists, each album is associated with one or more artists, and each artist can be associated with multiple tracks and albums. Many albums and artists are also labeled with one or more of 1,528 genres ranging from obvious categories like "classical" to esoteric groupings like "kwaito house." Finally, each artist is associated with the 20 or fewer other "related artists" in the dataset who are listened to most frequently by users who listen to the original artist. Some of these structural relationships between objects can be used to validate the metric embeddings we generate, as explained in more detail in the experiments section.

[1] `https://www.youtube.com/watch?v=ai_aImEbBRw`

3 Methods

3.1 Embedding Techniques

Formally, our goal is to determine a metric space (M, d) and an embedding function $f : S \to M$ (were S is the set of all songs) such that $d(f(s_1), f(s_2))$ roughly corresponds to the intuitive similarity between songs s_1 and s_2. In our project, we investigate the performance of two baseline embedding functions, the Raw Embedding function and the Principal Components Embedding, and compare them to the performance of an Adversarially Learned Embedding computed using deep learning techniques.

3.1.1 Baseline Embeddings

1. **Raw Embedding**. For a given song s, we compute spectrogram$(s) \in \mathbb{R}^{p \times q}$, and flatten this matrix to get a vector in $\mathbb{R}^{pq}$. We define $f_{\text{raw}}(s)$ to be this vector (i.e., $M = \mathbb{R}^{pq}$, and d is the usual Euclidean metric).

2. **Principal Component Embedding**. Given a number of songs $s_1, ..., s_m$, we first compute the matrix
$$M_{\text{raw}} = \begin{bmatrix} f_{\text{raw}}(s_1) \\ \vdots \\ f_{\text{raw}}(s_m) \end{bmatrix} \in \mathbb{R}^{m \times pq}.$$
We then conduct an r component PCA using scikit-learn [7] to get a matrix $C \in \mathbb{R}^{r \times pq}$ where the ith row of C is the ith principal component of M_{raw}. We then define $f_{\text{PCA}}(s_j)$ to be the jth row of $M_{\text{raw}} \cdot C \in \mathbb{R}^{m \times r}$.

As explained later, our spectrograms are of size $\mathbb{R}^{257 \times 430}$, so our flattened $f_{\text{raw}}(s)$ vectors are of length $110,510$. Since these vectors are so large, it is unfeasible to run any real tests on them. This is why we also use PCA to mitigate the curse of dimensionality, without sacrificing too much of the signal in our data. In our experiments, we use $r = 64$ to match the dimension of the embeddings produced by the ALI embedding described below.

3.1.2 Adversarially Learned Embedding

To compute our Adversarially Learned Embedding, we implement a deep learning model called Adversarially Learned Inference (ALI) [3]. The ALI model extends the Generative Adversarial Network architecture [4] to allow for inference of the latent vector $z \in \mathbb{R}^{64}$ corresponding to a given spectrogram $x \in \mathbb{R}^{257 \times 430}$ in addition to generation of a spectrogram x from a given latent vector z. More formally, we define two join distributions over x and z: the *encoder* distribution $q(x, z) = q(z)q(z|x)$ and the *decoder* distribution $p(x, z) = p(x)p(x|z)$. We also define $p(x)$ to be the true marginal distribution of spectrograms, and we assume the true marginal distribution of z is $\mathrm{N}(0, \mathrm{I})$. The goal of the ALI model is to define an adversarial game such that players are incentivized to minimize the difference between the encoder and decoder distributions. Doing so will ensure that the conditional distributions $q(z|x)$ and $p(x|z)$ match the true conditional distributions of $z|x$ and $x|z$, enabling us to sample from them and conduct inference tasks like computing the most likely latent vector z corresponding to a given spectrogram x and vice versa.

The adversarial game in question consists of three players, each represented by a deep convolutional neural network. The first network, the encoder $G_z(x)$, is tasked with generating samples of latent vectors $\hat{z}$ from $q(z|x)$ that best correspond with a given spectrograms x. The second network, the decoder $G_x(z)$, is tasked with generating samples of spectrograms $\hat{x}$ from $p(x|z)$ that best correspond with a given latent vector z (this network is equivalent to the generator component of the GAN architecture). The final network, the discriminator D, is tasked with classifying a given pair of spectrogram and latent vector (x, z) as having been sampled from either the encoder distribution or the decoder distribution.

At each iteration of the training algorithm, a batch of samples $(x, \hat{z})$ is drawn from the encoder distribution $q(x, z)$ and a batch of samples $(\hat{x}, z)$ is drawn from the decoder distribution $p(x, z)$. The discriminator network tries to minimize the number of pairs from each batch that it classifies incorrectly as having been drawn from the wrong distribution. The encoder and decoder networks then jointly try to maximize the number of pairs the discriminator classifies incorrectly by making pairs sampled from the two distributions they define as similar as possible. This two-step training process is iterated until the weights of the encoder, decoder, and discriminator have converged.

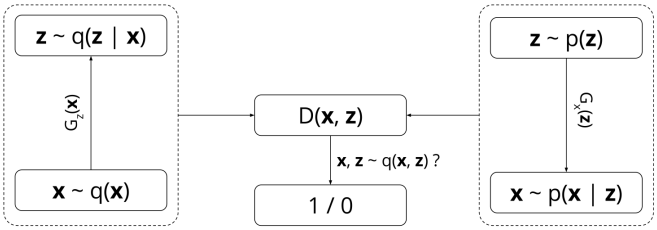

Figure 2: A diagram of the ALI architecture. The left box represents the encoder, the right box represents the decoder, and the center box represents the discriminator.

Of course, it is impossible to sample from $q(x, z)$ and

$p(x, z)$ because the distributions $q(z|x)$ and $p(x|z)$ are unknown. Instead, we resort to having the encoder and decoder networks learn deterministic approximations of samples from the conditional distributions, $G_z(x) \approx z \sim q(z|x)$ and $G_x(z) \approx x \sim p(x|z)$. Doing so is justified by a technique called the "reparameterization trick" [3]. We can then sample pairs (x, z) from the encoder and decoder joint distributions by first sampling $z \sim q(z)$ or $x \sim p(x)$ and then computing the corresponding $x = G_x(z)$ or $z = G_z(x)$.

In the original ALI paper, the game is formalized by the following objective:

$$\min_{G_z, G_x} \max_{D} \mathbb{E}_{x \sim p(x)}[\log(D(x, G_z(x)))] + \mathbb{E}_{z \sim q(z)}[\log(1 - D(G_x(z), z))]$$

Once the training algorithm has converged to some fixed point, we ignore the decoder and discriminator networks and use the encoder network to generate embeddings by defining $f_{\text{ALI}}(s) = G_z(s)$.

3.2 Evaluation Techniques

As stated before, a good embedding is one that defines a function $f : S \to M$ such that $d(f(s_1), f(s_2))$ represents the similarity between songs s_1 and s_2. Intuitively, this means that if we have a number of points in M, and we apply a clustering algorithm that minimizes distances within clusters (such as k-means or Mixture of Gaussians), the clusters should correspond to groups of similar songs. Since *genres* represent groups of similar songs, it's reasonable to say that the better an embedding is, the better clustering algorithms should perform at the task of unsupervised genre classification.

We use this intuition to develop a method for evaluating the representation power of our embeddings. For a set of songs $S = \{s_1, ..., s_m\}$ with genre labels $G = \{g_1, ..., g_m\}$ drawn from k possible genres, we cluster the unlabeled embeddings $f(S) = \{f(s_1), ..., f(s_m)\}$ into k clusters using both k-means and Mixture of Gaussians, giving us predicted labels $P = \{p_1, p_2, ..., p_m\}$. We then score how similar P and G are. The better the performance on this task, the more effective f is.

There are a number of metrics to gauge the similarity between the ground truth clusters G and the predicted clusters P [1]. We use two measures: V-measure and AMI (Adjusted Mutual Information). V-measure is the geometric mean of the homogeneity (the proportion of points within the clusters of P that are in the same cluster in G) and the completeness (the proportion within the cluster of G that are in the same cluster in P). AMI gives a measure of the agreement between G and P independent of permutations, adjusted for random chance.

Both of these metrics give a score between 0 and 1, where random clusterings are given scores close to 0, and perfect clusterings are given scores of 1. Full mathematical formulations can be found at [1].

4 Results

4.1 Training the ALI Model

To train the ALI model, we generated a training set by selecting 15 different genres from our dataset and sampling 800 songs from our dataset belonging to each of those genres. For each song, we downloaded its 30-second audio preview, selected 10 random 5-second segments $s_1, \ldots, s_{10}$ from the preview, and computed $\text{spectrogram}(s_1), \ldots, \text{spectrogram}(s_{10}) \in \mathbb{R}^{257 \times 430}$. All together, this process yielded a training set of $120,000$ random 5-second spectrograms from 12,000 unique songs and 15 different genres. We implemented the architecture using the deep learning framework Keras [2] and the adversarial learning library Keras Adversarial [9].

Unfortunately, when we first trained our ALI model on this training set, the model quickly converged to a fixed point in which the discriminator simply classified all spectrogram/latent-vector pairs as having been sampled from either only the encoder or only the decoder. This outcome was not surprising, since getting GANs to converge to optimal fixed points is an open research problem [8]. Given that the discriminator very quickly converged to a the classification pattern described, we hypothesized that there were two problems. First, the discriminator was training much faster than the encoder and decoder. Second, because the discriminator used a sigmoid activation function for its output layer, the networks were suffering from gradient saturation. To rectify these issues, we implemented several fixes.

First, in accordance with the LSGAN architecture [6], we changed the activation function of the output layer of our discriminator to a linear activation function and switched our loss function to a least-squares loss function to avoid gradient saturation. Intuitively, squared-error loss penalizes extremely wrong predictions much more harshly than cross-entropy loss, allowing more gradient to flow to the encoder and decoder so they can correct their parameters more quickly. In addition, we employed a poplar trick for training adversarial networks described in the [3] paper: we swapped the labels used when updating the encoder and decoder and switched the objective of the encoder and decoder to a minimization objective. After these two changes, the objectives of the discriminator and the encoder and decoder became the following:

Discriminator Objective:

$$\min_D \mathbb{E}_{x\sim p(x)}[(D(x, G_z(x)))^2] + \mathbb{E}_{z\sim q(z)}[(D(G_x(z), z) - 1)^2]$$

Encoder and Decoder Objective:

$$\min_{G_z, G_x} \mathbb{E}_{x\sim p(x)}[(D(x, G_z(x)) - 1)^2] + \mathbb{E}_{z\sim q(z)}[(D(G_x(z), z))^2]$$

Next, we implemented four changes to slow down the rate at which the discriminator could learn relative to the encoder and the decoder. First, we trained the encoder and decoder on four batches for every one batch we used to train the discriminator, rather than training both the discriminator and the encoder and decoder on one batch each. Next, we regularized the kernels of the convolutional layers in the discriminator. We also implemented label smoothing as described in [**?**]; we changed all the positive labels given to the discriminator to randomly sampled values between 0.7 and 0.9. Finally, we added random noise to the spectrograms given to the discriminator so the pairs inputted to the discriminator became $(x + \epsilon, z)$, where $\epsilon \sim N(0, 10 \cdot \mathrm{I})$.

All of these changes together allowed the ALI model to converge to a non-trivial fixed point, as shown in 4.1.

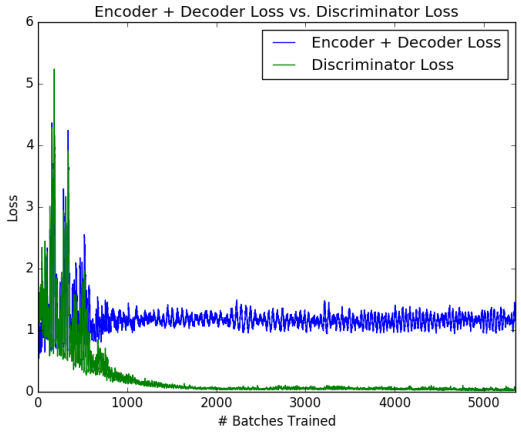

Figure 3: A plot of the discriminator loss and the encoder and decoder loss over the course of the training process. Both losses converge after roughly 2000 training batches.

4.2 Genre Clustering Performance

We ran the genre classification test on two new sets of 5-second spectrograms. The first consisted of 50 spectrograms from all 15 different genres in the training dataset. The second consisted of 50 spectrograms from just four genres: *classical*, *edm* (electronic dance music), *rock*, and *indie rock*. We conducted the second test to see if the embedding models could distinguish between both very different genres (classical and edm) and very similar but different genres (rock and indie rock).

We found that across all datasets, clustering algorithms, and performance metrics, the ALI embedding performed significantly better than the PCA embedding.

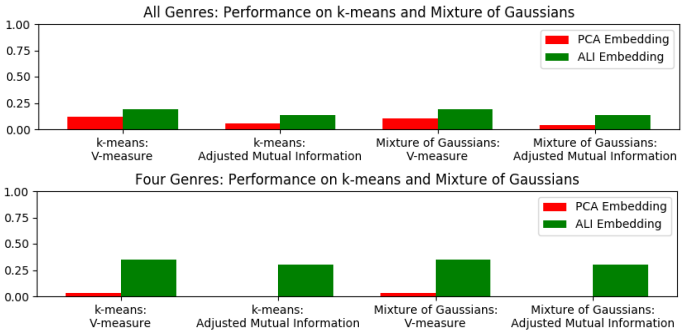

Figure 4: Genre clustering performance metrics for the two tests.

In particular, note that the second test shows that on the four genre dataset, the PCA embedding does only slightly better than a random clustering (meaning that it has no clear notion of difference between indie rock, rock, classical, and edm), but the ALI embedding does significantly better than a random clustering.

To visualize the ALI embedding, we used the t-SNE dimension reduction technique [5], which preserves topological structure and clustering while projecting the data into a two-dimensional space. In doing so, we found some fascinating results:

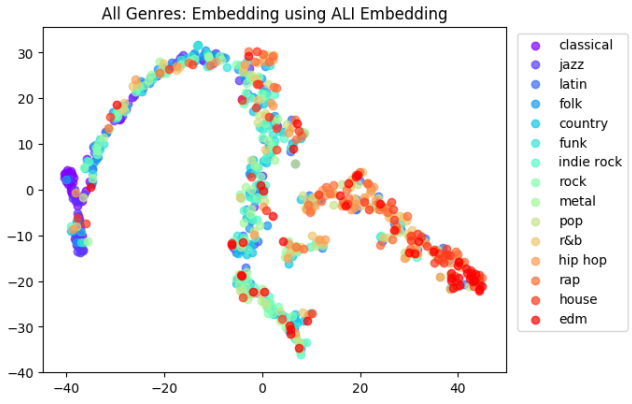

Figure 5: A t-SNE plot of embedded song spectrograms from 15 different genres. The ALI embedding is able to disambiguate more instrumental genres from more electronic genres.

After looking at this first plot, we concluded that the ALI embedding seemed to understand how electronic or

instrumental a given song is. In particular, when we ranked our 15 genres on a gradient from most instrumental (violet) to most electronic (red), we found that gradient to be visually apparent from the embedding.

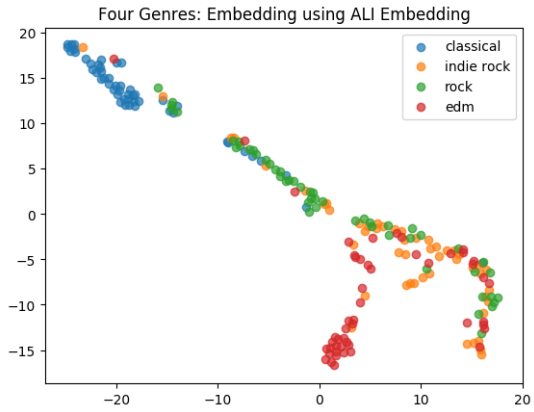

Figure 6: A t-SNE plot of embedded song spectrograms from four different genres. The ALI embedding is able to separate very different genres like classical and edm while locating songs from similar genres like rock and indie rock closer to each other in space.

The second plot provides a visualization of the second test. It's clear that the classical and edm songs are mostly separated from the rest of the points, but the indie rock and rock songs are more closely related. Even so, there is some separation between the indie rock and rock songs, since the former resides mostly in the bottom right of the lot, and the latter resides mostly in the center.

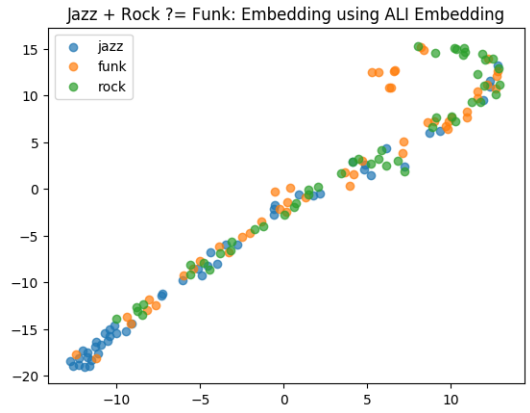

Figure 7: A t-SNE plot of embedded song spectrograms from jazz, rock, and funk genres. The ALI embedding tends to place funk songs in between jazz songs and rock songs, reflecting the fact that the funk style is a fusion of jazz and rock music.

Before constructing the third plot, we postulated that the embedding might learn that funk music is a combination of jazz and rock music, since funk has heavy jazz influences with rock band instrumentation. The plot to some extent confirms this hypothesis, since jazz is concentrated towards the bottom left, rock is concentrated towards the upper right, and funk is scattered across the spectrum.

As a final test of our ALI embedding's ability to disambiguate song styles, we tried to determine whether the ALI embedding would be able to discriminate between Coldplay's older indie rock songs and Coldplay's newer pop-influenced songs. To do so, we scraped previews of the songs from all of Coldplay's albums. We then plotted their ALI embeddings (using PCA to visualize them so as to preserve distances) and colored them according to a gradient from newest album to oldest album. We found that the model was able to differentiate between Coldplay's older, more rock-like songs, and their newer, more pop-like songs:

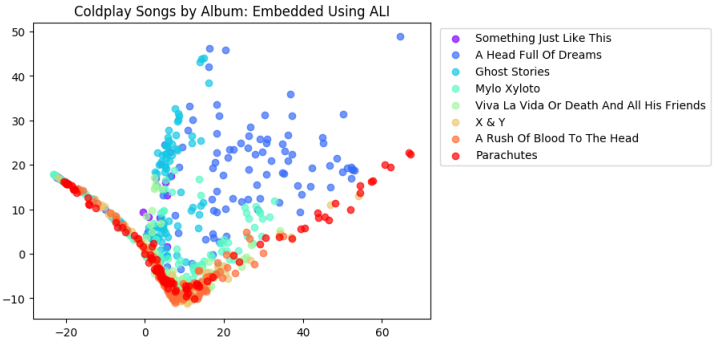

Figure 8: A PCA plot of embedded song spectrograms from all of Coldplay's albums. The ALI embedding is able to illustrate the difference between Coldplay's older rock-like songs and their newer, more pop-like songs.

5 Conclusions and Next Steps

This project demonstrates that deep unsupervised embeddings were able to capture salient differences between songs much better than traditional embedding methods like PCA. Although our experiments were able to demonstrate that the ALI embedding can distinguish between high-level music styles and genres, it is still unclear exactly which attributes of songs the ALI model uses to determine its embeddings. With more time, we would conduct experiments to test whether the ALI embeddings separate songs by other salient attributes such as tempo and key. It would also be interesting to see whether or not ALI embeddings do a good job representing artist similarity, which would be possible by taking advantage of Spotify's related artist data.

Contributions

Here is how we split up the work for the project:

- **Pras and Brad**: Formulated baseline embedding and embedding evaluation strategies, implemented and began training ALI model.
- **Brad**: Built Spotify crawler to download the data, set up data infrastructure on Google Cloud, built data cleaning and spectrogram computation pipelines, debugged ALI training.
- **Pras**: Implemented the baseline embeddings (raw and PCA), set up embedding evaluation frameworks using both k-means and Mixture of Gaussians, wrote plotting code.

References

[1] Clustering with scikit-learn. `http://scikit-learn.org/stable/modules/clustering.html`, 2017. [Online; accessed 14-Dec-2017].

[2] François Chollet et al. Keras, 2015.

[3] Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Olivier Mastropietro, Alex Lamb, Martin Arjovsky, and Aaron Courville. Adversarially learned inference, 2016.

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[5] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[6] Xudong Mao, Qing Li, Haoran Xie, Raymond YK Lau, Zhen Wang, and Stephen Paul Smolley. Least squares generative adversarial networks. *arXiv preprint ArXiv:1611.04076*, 2016.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[9] Ben Striner. Keras adversarial, 2016.