Improved Music Classification Using Singular Value Decomposition and Feature Scaling

Wyatt Scherer – *University of Washington*

AMATH 582 Final Project, Due 3/19/2020

GitHub Repository: https://github.com/wcscherer/DATA_ANALYSIS-AMATH582

ABSTRACT

Homework 4 used the SVD algorithm and three classifier algorithms to classify music by decomposing the spectrogram of 5-second clips of music. The three cases considered were 1) classify music by artist, 2) classify music by band within a single genre, and 3) classify music by genre. For the original implementation of this music classifier, no classification algorithm was able to achieve a classification score greater than random assignment. This analysis improves the original classification algorithm through rescaling the input data and larger test data splits. Part 1 of this analysis investigates the effects of rescaling the input data and larger test data splits. Part 2 compares the results from the original classification algorithm to the improved algorithm. Part 3 applies the improved classification algorithm to classifying songs by album for a single band.

1. Introduction

The second part of Homework 4 used SVD and pre-built classifier algorithms to classify music of different bands and genres. First the SVD algorithm was used to decompose the spectrogram of 5-second music clips. The Principle Components (PCs) from this decomposition were then fed into classification algorithms for classifying different artists and genres based on the PCs. The three classifier algorithms used to classify the song data were: k-nearest-neighbors (KNN), linear discriminant analysis (LDA), and linear support vector classifier (SVC). All classifier algorithms used are pre-built classifiers from the python scikit.learn library. Three different classification scenarios were considered: 1) classify songs by artist for three artists of different genres, 2) classify songs by artist for three artists within the same genre, and 3) classify songs by genre for songs by various artists. For the results of Homework 4, none of the classifiers were able to successfully classify any bands/artists better than random assignment on test data.

This analysis improves on the original classification algorithm in three ways: wider spectrogram windows, feature scaling, and larger testing sets. An additional classifying algorithm, Gaussian Naïve Bayes is also investigated. The effects of data scaling and test data split size will be explored on the Case 1 analysis for Part 1. Part 2 will use the results from Part 1 to create improved classification models for each of the three original cases. Part 3 applies the improved classification model to classifying songs by album for a single band.

2. Technical Basis for PCA using SVD and Classification

2.1. Singular Value Decomposition

Reducing large amounts of data into a smaller amount of data that faithfully represents the original data set simplifies analyzing complex processes. This is what Singular Value Decomposition (SVD) can do for large, seemingly uncorrelated data sets [1]. Abstractly, SVD decomposes an input matrix transformation $\overleftrightarrow{\boldsymbol{A}}$ into three matrices **U, Σ,** and **V** that satisfy the following relationship [1]:

$$\boldsymbol{A} = \boldsymbol{U\Sigma V}^{*}\text{, rewritten as } \boldsymbol{AV} = \boldsymbol{U\Sigma} \quad (1)$$

where

$$\boldsymbol{A} \in \mathbb{R}^{mxn}, \boldsymbol{U} \in \mathbb{C}^{mxm}\ is\ unitary,$$
$$\boldsymbol{\Sigma} \in \mathbb{R}^{mxn}\ is\ diagonal, \boldsymbol{V} \in \mathbb{C}^{nxn}\ is\ unitary \quad (2)$$

What Equation 1 effectively says is that, for a matrix **A**, there exists unitary transformation matrices **U**, and **V** that when combined with a diagonal scaling matrix **Σ,** recreate the matrix **A.** Writing Equation 1 as a transformation of **A** into a new matrix represented by **U** and **Σ.** This can be represented by Equation 3 [1]:

$$\begin{bmatrix} \\ \boldsymbol{A} \\ \\ \end{bmatrix} [\boldsymbol{v_1} \quad \cdots \quad \boldsymbol{v_n}] = \begin{bmatrix} \\ \boldsymbol{u_1} \quad \cdots \quad \boldsymbol{u_n} \\ \\ \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix}$$
$$\Rightarrow \boldsymbol{Av_j} = \sigma_j \boldsymbol{u_j} \quad 1 \leq j \leq n \quad (3)$$

Using the SVD to determine the principle components of a matrix requires defining **A** as a sum of rank one matrices:

$$A_N = \sum_{j=1}^{N} \sigma_j \boldsymbol{u_j} \boldsymbol{v_j}^{*}\ where\ 0 \leq N \leq rank(A) \quad (4)$$

This interpretation demonstrates that the SVD offers a type of fitting algorithm that generates the

minimum number of principle components necessary to recreate the original matrix **A**. Summing through the jth components of **V**, **U**, and **Σ**, where j represents the jth Principle Component, can approximate the matrix A. How much of the original matrix **A** that this partial sum represents can be quantified by the cumulative energy of the individual principle components. The energy, E, of the j^{th} principle component of N total components and the cumulative sum E^C can be calculated by:

$$E_j = \frac{\sigma_j}{\sum_{j=1}^{N} \sigma_j} \ and \ E_j^C = \frac{\sum_{i=1}^{j} \sigma_i}{\sum_{i=1}^{N} \sigma_i} \quad (5)$$

For normally distributed data, only a few PCs may be needed to recreate +90% of the original matrix **A**. This means that the high dimensional data in **A** can be accurately recreated by j << N independent components. Applying SVD to a covariance matrix, for example, allows for determining which dimensions exhibit the largest changes in the measurement data and allows for easy filtering for the important components [1].

2.2. Overview of Classification Algorithms Used

For this analysis, songs from various artists are filtered and used to classify songs by artist, genre, and album. Four classification algorithms were implemented in this analysis: KNN, LDA, SVC, and GNB. Detailed descriptions of these popular classification algorithms are freely available from many sources, only brief descriptions are given here.

2.2.1. K-Nearest Neighbors (KNN)

KNN classifies an object n based upon the classifications of the objects *nearest* to n. It is mostly used in a supervised classification scheme where a KNN model is initially trained on a known classified set of data. This training allows for the KNN model to categorize the input data by classified input features. The features of a data set are the different dimensions along which an object can be classified, such as weight, height, color, volume, etc. Multiple features mean that the KNN algorithm classifies data along multiple mathematical dimensions: a dataset where each value has 5 separate parameters means that the KNN algorithm operates in 5D mathematical space. Whether or not a new, un-labeled data point belongs to a specific classification depends on the unlabeled data point's distance from other classified objects. The distance is frequently calculated using Euclidian distance between data points. The k in KNN refers to how many nearest neighbors are being considered in the classification. For example, if k = 5, the algorithm only considers the 5 nearest points to the unlabeled point in question. Then, based upon the distance and classification of the known points, the classification of the new point is estimated by weighting the classification of the k nearest neighbors by distance. The unknown point is classified as belonging to the class of nearest neighbors of the highest weight [2].

2.2.2. Linear Discriminant Analysis

LDA is similar to SVD in that it looks for linear combinations of parameters that can be used to represent high-dimensional data. How the two algorithms differ is that SVD seeks to reduce high-dimensional data into lower dimensional representations, whereas LDA seeks to reduce high-dimensional data into lower dimensional representations and classify that data by how it decomposes into its lower dimensional forms. LDA is a supervised learning algorithm; meaning requires classified data in order to classify unknown data. LDA assumes that the conditional probability that an unknown input x belongs to a class y is normally distributed with a known mean and covariance. The choice of class is discriminated by the log of the likelihood ratios being greater than some threshold T. To simplify the calculation, LDA assumes that the covariances of all the individual classes are equal. This allows the classification criterion of being in a given class to be represented as a linear combination of known data. Effectively, LDA projects multidimensional data onto lower dimensional subspaces representing the possible classifications. Whichever lower dimensional subspace most of the unclassified data projects onto becomes the class of the unknown input data [3].

2.2.3. Support Vector Classification

SVC is a common algorithm for both clustering unlabeled data and for classifying labeled data. Given a set of high-dimensional data with N features (N dimensional vector), the SVC algorithm determines a hyperplane of dimension N-1 that best separates all the data into different classes or clusters. While many dim(N-1) hyperplanes can be constructed that separate the data, SVC chooses the hyperplane that maximizes the distance between clusters/classes of data. This is often referred to as a maximum-margin classifier. As a classifying algorithm, SVC first creates the desired hyperplane from labeled training data. With the different classes defined by the hyperplane, new unknown data is classified by its distance relative to the nearest section of the hyperplane. The new unknown point is classified by what region it is bounded in by the hyperplane [4].

2.2.4. Gaussian Naïve Bayes Classification

GNB utilizes conditional probability to classify data by input features. These features, **x**, for each class C_x are assumed to be probabilistically independent of each other, regardless if they are in fact independent. This "Naïve" assumption allows for the simplification of calculating the conditional probability that a feature x belongs to the classification C_x. Before the likelihoods are calculated, the data is separated by classification. The probability of belonging to a specific class (the prior) is frequently simply calculated as the inverse of the number of classes. For GNB, the likelihood $p(\mathbf{x}|C_x)$ used in the conditional probability is calculated using the Gaussian normal distribution. The probability that given features belong to a specific class is then calculated as the product of the prior of the class and the likelihood the feature belongs to that class. GNB is considered a simple classification model but it requires smaller training datasets to achieve robust classification [5].

2.2.5. Feature Scaling

Data that spans many orders of magnitude often proves difficult for many classification algorithms to process. Standard data scaling or feature scaling process include renormalizing data by the minimum or maximum value, rescaling by the mean of the features, Z-score normalization and other mathematical processes [6]. This analysis uses a simpler method based upon observations of the distribution of PC magnitude. The principle components of the spectrograms often span from 10^4 to 10^{-10}. Since this data spans approximately 14 orders of magnitude, the base 10 logarithm was chosen as an appropriate way to scale the data. To ensure that the logarithm is applicable, the absolute value of the data is first taken before applying the base 10 logarithm.

$$Data_{scaled} = log_{10}(|Data|) \quad (6)$$

3. Description of Algorithm Implementation

Custom python functions used in this analysis are defined in Appendix B. Standard python package functions are briefly described in Appendix A.

3.1. Music Classification using SVD

For all three Parts of the analysis, the fundamental algorithm is the same. What changes are the filtering window, the use of feature scaling, and different music databases. The performances of each classification algorithm are compared for varying numbers of PC features considered.

1) For each classification type (artist and genre), each song considered has a 5 second sample taken from the song. The songs are randomly sampled from between 10 seconds and 100 seconds into the song to ensure good cross section of each song without trying to sample past the end of a song. The custom function *read_song* uses the python wave package to read in files (as a wav format) and converts the signal to a mono signal and also returns the sampling rate.
2) This 5 second sample is then converted into a spectrogram and reshaped into a column vector. The custom function *spectrogram* uses the python package signal's spectrogram function that builds a spectrogram based on the 5 second clip. This spectrogram is then reshaped into a column vector. Then the average is subtracted out of the column vector for use in creating a covariance matrix.
3) The functions *read_song* and *spectrogram* are called by the custom wrapper function *build_svec* which goes into the file where the music to be sampled is stored. It then takes 5 second samples from each song (number of samples taken is user defined). Each of these samples is then converted into a spectrogram and processed into a column vector. All the spectrogram column vectors are concatenated into one matrix for each classification type (band, genre, etc). The number of columns represents the number of samples times taken from each song and the rows are the reshaped spectrogram values. Each database representing a classification type (genre, artist, band etc) has one matrix that has all the transformed samples as a covariance matrix.
4) SVD is performed by the custom function *svd_songs* which is a wrapper around the numpy SVD method. For this analysis, full matrices are not used, which means a reduced SVD algorithm is used to save time. The function returns U, S, V, and SV* from the decomposition.
5) The principle mode projections from SV* and the principle modes V can be used to train the models. These matrices are reshaped so that each row represents a single clip and each column represents a principle component. This is the dataframe format that python scikit.learn classification models expect.
6) Before the datasets are classified, feature scaling is applied. This analysis takes the absolute value of the SV* data before taking the base ten logarithm. For numbers very close to 0 ($< 10^{-30}$), the logarithm function returns negative infinity. These negative infinity values are replaced by -30, which represents a number that effectively equals zero (10^{-30}).
7) To classify each dataset, an additional column is added that labels each row, e.g. 'Jazz' or 'Hendrix'.

These are the labeled datasets that will be used to train and test different classifiers.

8) The training data is then split into test and train data sets using the python scikit learn model_selection package. This function splits the Xtrain, Xtest, and ytrain, ytest datasets used to train and validate each model. The split data is then fed into each of the classification algorithms, first with the training split. Once the model is trained, it is cross-validated with the test set. The KNN, LDA, SVC, and GNB algorithms all come from the python scikit learn module. This is all performed by the large custom function wrapper *train_models,* which returns the fit score on the test and train data for each classifier.
9) To compare the different classifiers over multiple features and get statistically significant score results, a custom wrapper function *avg_models* was written. This function runs the *train_models* function multiple times (user defined) and returns the min score, max score, average score, and standard deviation of the score for each model for both the training and test datasets.
10) The custom function *plot_results* plots the average score for each classification algorithm for each number of modes trained on. The standard deviations of the scores for each number of modes considered are included as error bars.

4. Results

4.1. Part 1) Testing Effects of Feature Scaling and Training Split Size on Case 1 Data

Part 1 explores the effects of feature scaling and test-train split size. Each of the four classification models was trained 10 times. Note that the default python spectrogram used utilized a 'tukey' window of width 0.25. Also, the default settings for all the KNN, LDA, SVC, and GNB training models were used. The base 10 logarithm feature scaling is applied. The three data train to test ratios considered are 0.10, 0.20, and 0.30. All the models were trained on the principle mode projection data **SV*.**

Since there are three possible classifications, for any classifier to be considered a useful classifier, it must be able to accurately classify songs with a success rate > 33%. The three artists/bands chosen for this classification were 10 *The Police* songs, 10 Tracks from the *Lord of the Rings Sound Track*, and 10 songs by *Otis Redding.* The three categories are 'Police', 'Otis', and 'LoTR'. Each of these 10 songs was sampled 15 times to create the spectrograms of the 5-second clips for a total of 150 clips per artist.

4.1.1. Feature Scaling

The principle components from SVD are the same for both un-scaled and scaled data – see Figure 1. Figure 2 shows the training and testing results for the Case 1 Data without feature scaling. Figure 3 shows the training and testing results for the Case 1 Data with Feature scaling.

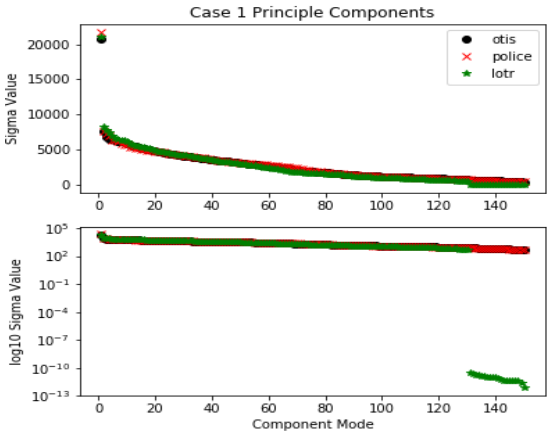

Figure 1: Case 1 Principle Components (Sigma) for 150 Modes

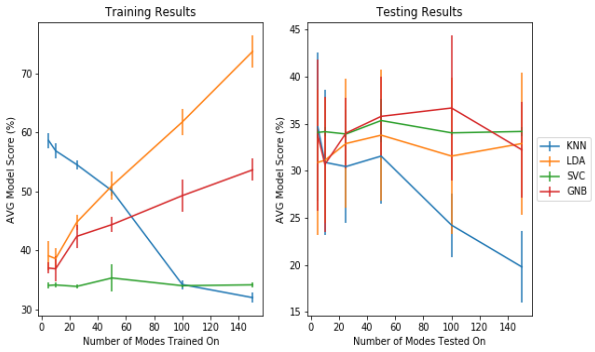

Figure 2: Case 1 Classification Results with no Feature Scaling and Test/Train Split of 10%

Figure 2 shows that none of the four classifiers are able to classify unknown song clips better than random assignment without feature scaling.

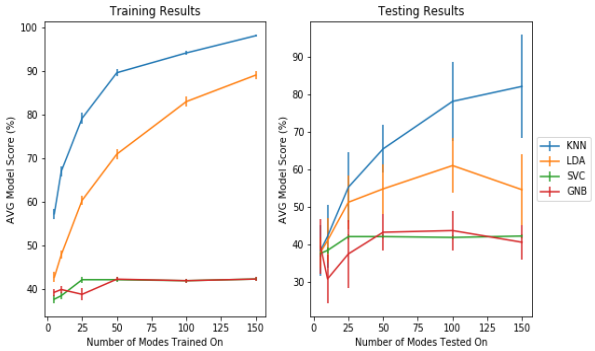

Figure 3: Case 1 Classification Results with Feature Scaling and Test/Train Split of 10%

Figure 3 shows that both KNN and LDA score significantly better with feature scaling, while SVC and GNB do not. KNN approaches 80% correct classification on the cross-validation sets when including all modes, although it has a high variance. Feature scaling is therefore necessary to achieve classification scores above random assignment with KNN and LDA.

4.1.2. Train/Test Split Size

Both Figures 2 and 3 were generated using a Test/Train split of 10%. Figures 2 and 3 have large errors for all tested classifiers as there are a small amount of samples to test. One misclassification significantly affects the resultant score of the tested model when considering averages.

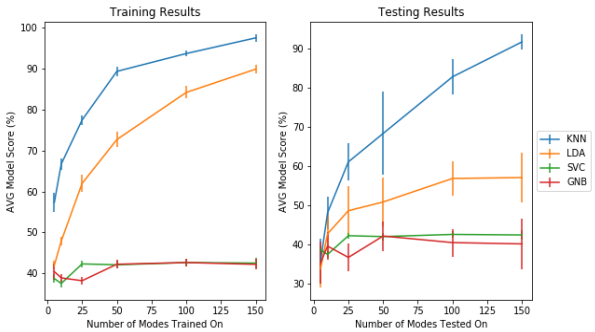

Figure 4: Case 1 Classification Results with Feature Scaling and Test/Train Split of 20%

Figures 4 and 5 were generated using feature scaled data and test/train splits of 20% and 30% respectively. Figures 4 and 5 show simply the effect of improved sampling-the larger the test/train split the better smaller the error. It should be noted that a lower classification error does not mean a better classifier, but larger testing data set. A test/train split of 20% will be used in Part 2 and Part 3 of this analysis.

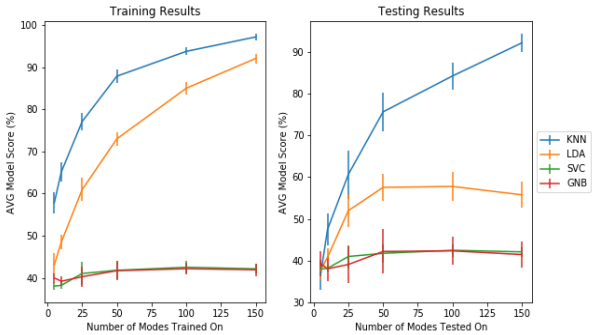

Figure 5: Case 1 Classification Results with Feature Scaling and Test/Train Split of 30%

4.2. Part 2) Comparing Improved Algorithm to the Original Classification Algorithm

Part 2 of this analysis compares the results of the improved classification algorithm for all three of the original cases: 1) classify songs by artist for three artists of different genres, 2) classify songs by artist for three artists within the same genre, and 3) classify songs by genre for songs by various artists. The improved algorithm employs feature scaling and a test/train split of 20%. The original analysis considered 500 samples per each artist/genre and the algorithm was unable to accurately classify song clips. The improved classification algorithm is able to classify different artists with a ~90% accuracy using only 150 samples per artist. Figures 4 and 5 show that with feature scaling fewer modes are needed to accurately classify songs.

4.2.1. Case 1) Different Artist Classification

The three artists/bands chosen for this classification were 10 *The Police* songs, 10 Tracks from the *Lord of the Rings Sound Track*, and 10 songs by *Otis Redding.* The three categories are 'Police', 'Otis', and 'LoTR'. Each of these 10 songs was sampled 15 times to create the spectrograms of the 5-second clips for a total of 150 clips per artist.

As can be seen from Figures 6 and 7, the Principle Components are of the same magnitude as the original analysis and none of the individual component energies are above 10%. Note that the features scaling is implemented after implementing the SVD algorithm, meaning the PCs should be effectively the same between both analyses. Figure 8 shows the improved classification results when using feature scaling.

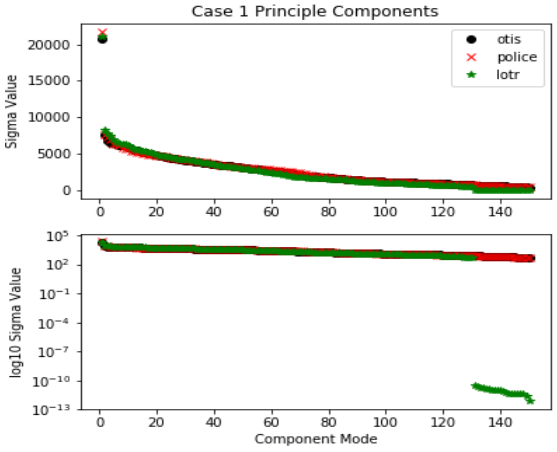

Figure 6: Case 1 Principle Components

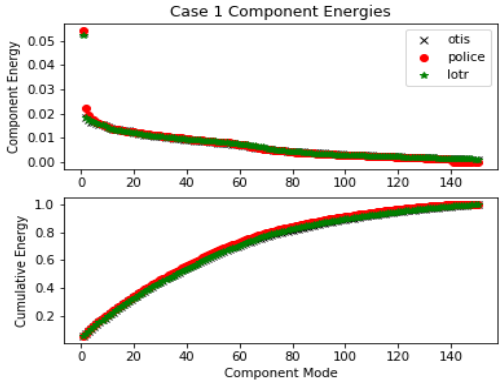

Figure 7: Case 1 Component Energies

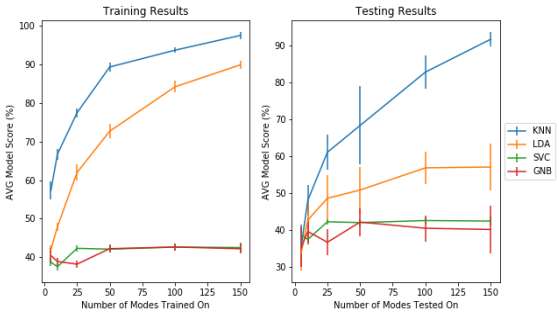

Figure 8: Case 1 Classification Results with Feature Scaling and Test/Train Split of 20%

4.2.2. Case 2) Different Artist within the Same Genre Classification

The three artists chosen for this classic rock classification were 10 *Led Zeppelin* songs, 10 *Jimi Hendrix* songs, and 10 songs by *The Rolling Stones.* The three categories are 'Led Z', 'Jimi', and 'Stones'. Each of these 10 songs was sampled 15 times to create the spectrograms of the 5-second clips for a total of 150 clips per artist. Figures 9 and 10 show the principle components and their energies for Case 2. Figure 11 shows the classification results of the improved algorithm for Case 2.

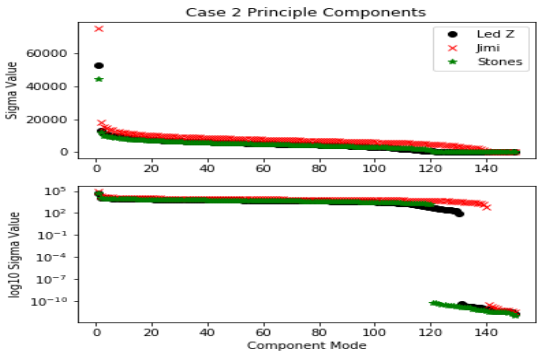

Figure 9: Case 2 Principle Components

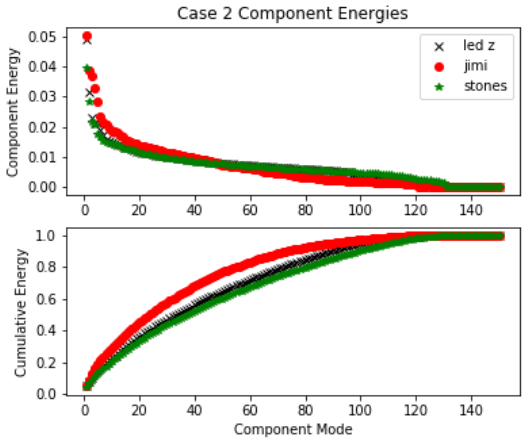

Figure 10: Case 2 Component Energies

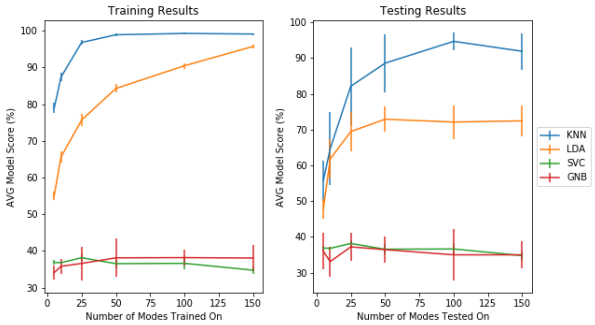

Figure 11: Case 2 Classification Results with Feature Scaling and Test/Train Split of 20%

4.2.3. Case 3) Genre Classification

The three genres chosen for this classification were *Rock*, *Jazz, and Soul.* For Rock, the following bands were used: 3 *Led Zeppelin* songs, 4 *Jimi Hendrix* songs, and 3 songs by *The Rolling Stones.* For Jazz, the following bands were used: 2 *Joshua Redman* songs, *4 Miles Davis* songs, and *4 John Coltrane* songs. For Soul, *5 Otis* Redding songs were used and *5 Marvin Gaye* songs were used. The three categories are 'Rock, 'Jazz', and 'Soul'. Each of these 10 songs was sampled 150 times to create the spectrograms of the 5-second clips for a total of 150 clips per genre. Figures 12 and 13 show the principle components and their energies for Case 3. Figure 14 shows the classification results of the improved algorithm for Case 3.

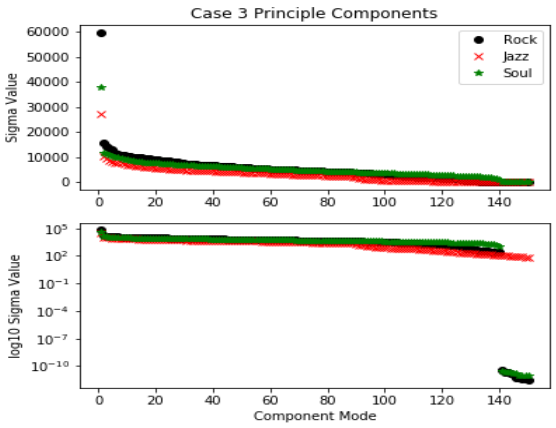

Figure 12: Case 2 Principle Components

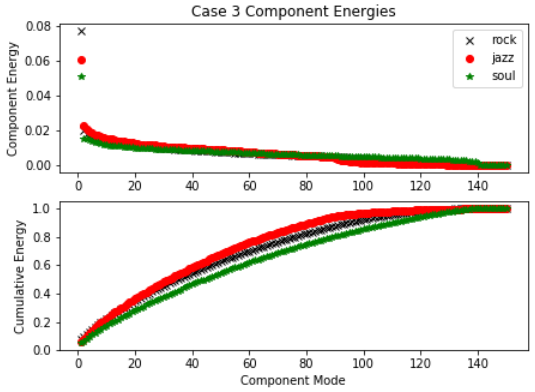

Figure 13: Case 3 Component Energies

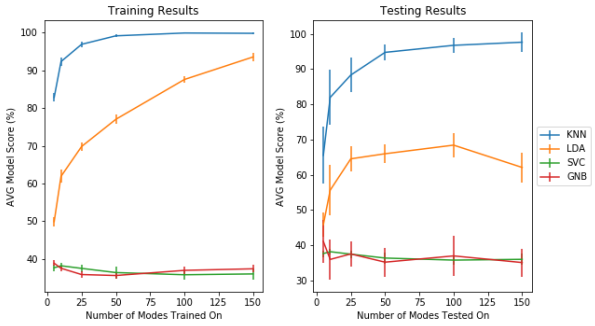

Figure 14: Case 3 Classification Results with Feature Scaling and Test/Train Split of 20%

4.3. Part 3) Classifying Songs by Album for the Same Artist

The band *Rage Against the Machine (RATM)* is often criticized for their songs sounding very similar. To prove once and for all if their songs are too similar, the same classifier used in Parts 1 and 2 will be used to classify RATM's song by album. Their four studio albums are considered in this analysis: *Rage Against the Machine (RATM), Evil Empire (EVIL_E), The Battle for Los Angeles (TBOLA), and Renegades (RENEG).* Ten tracks from each album are considered with 15 samples taken from each track for a total of 150 samples per album. For RATM's songs to be essentially the same song, a sample from any song should be indistinguishable from any other sample; therefore, the best a classifier could do is random assignment (25% correct for four albums). If the improved classifier can correctly classify RATM songs by album above 25%, then RATM songs can no longer be considered the same song. Figures 15 and 16 show the principle components and energies for all four albums. Figure 17 shows the classification results for the four classifiers.

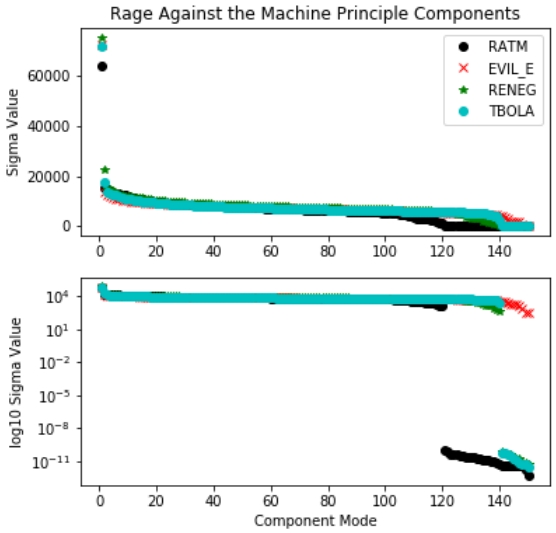

Figure 15: RATM Album Principle Components

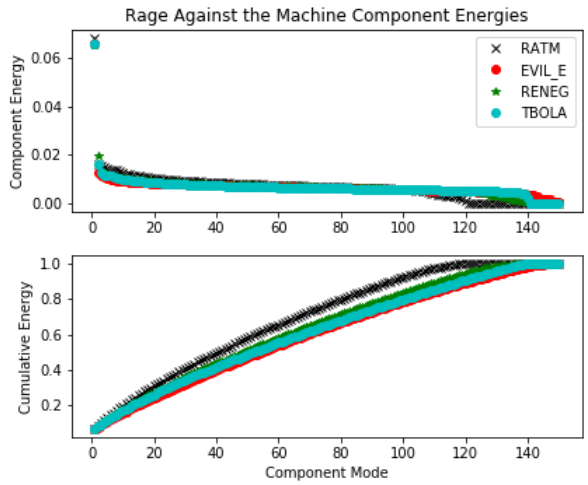

Figure 16: RATM Album Components Energies

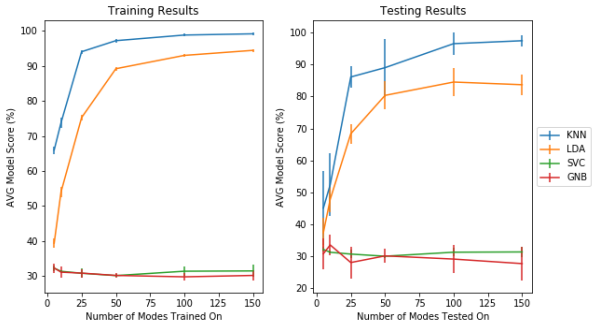

Figure 17: RATM Classification Results with Feature Scaling and Test/Train Split of 20%

5. Conclusions

5.1. Part 1) Testing Effects of Feature Scaling and Training Split Size on Case 1 Data

Feature scaling drastically improved the classification results for KNN and LDA. Surprisingly, it had little effect on SVC or GNB. For the original classifier, model test scores were never above 40%, even with 500 modes. With feature scaling, KNN achieved scores around 90% and LDA around 50% for Case 1 data, both better than random assignment (33%). This demonstrates that for certain classification algorithms like KNN or LDA, feature scaling is necessary to produce meaningful results.

Increasing the test/train split fraction reduces the variance in the model score, but that reduction in variance doesn't improve the model's ability to classify data. It just improves the reporting statistics by increasing the number of data points, which reduces the effect of a single misclassification.

5.2. Part 2) Comparing Improved Algorithm to the Original Classification Algorithm

For all three classification cases, none of the original classifiers with no feature scaling performed statistically better classifying test data than random assignment, regardless of the amount of features used. By including feature scaling using base 10 logarithms, the KNN classifier was able to correctly classify test data more than 90% of the time for all three cases – significantly better than random assignment. This analysis only used 150 samples per artist/band and the original classifier used 500 samples per artist/band. Feature scaling for KNN and LDA was more important than more data for accurate classification. The feature scaling also improved the test scores for LDA classification, but the test results of LDA were never as high as for KNN. Simply adding the feature scaling turned the worthless original classifier algorithm into an extremely successful KNN classifier. This was all done with the default settings for the spectrogram generator function and the classifier functions. By adjusting these default settings, improvements to the testing score for SVC and GNB methods could likely be achieved.

5.3. Part 3) Classifying Songs by Album for the Same Artist

The improved classification algorithm was applied to the four studio albums of *Rage Against the Machine (RATM)*. RATM is commonly accused of writing songs that all sound the same. By successfully classifying RATM songs by album, this claim that all RATM songs are too similar can be dispelled. Figure 16 clearly shows that with feature scaling, KNN and LDA successfully classify RATM songs by album well above random assignment. KNN with feature scaling was able to achieve greater than 90% classification accuracy using only 100 out of 150 modes. Both SVC and GNB did not performing statistically significantly better than random assignment (25%).

References

1. "Chs. 15-17." *Data-Driven Modeling and Scientific Computation: Methods for Complex Systems Et Big Data*, by J. Nathan. Kutz, Oxford Univ. Press, 2013.
2. "K-nearest neighbors algorithm", Wikipedia, https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
3. "Linear Discriminant Analysis", Wikipedia, https://en.wikipedia.org/wiki/Linear_discriminant_analysis
4. "Support-vector machine", Wikipedia, https://en.wikipedia.org/wiki/Support-vector_machine
5. "Naïve Bayes classifier", Wikipedia, https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Gaussian_naive_Bayes
6. "Feature Scaling", Wikipedia, https://en.wikipedia.org/wiki/Feature_scaling

APPENDIX A – GENERIC PYTHON FUNCTIONS USED

- *np.shape* returns the dimensions of a numpy array
- *np.signals.spectrogram* returns the time steps, the frequency range, and the spectrogram data for those times and frequency ranges
- *loadmat* reads in MATLAB .m files
- *np.linalg.svd* implements the SVD algorithm on an input matrix and returns the three matrices defined in Section 2
- *np.vstack* stacks lists as rows into a numpy array
- *np.power* raises the elements of a numpy array to a power

APPENDIX B – PYTHON CODE

Improved Music Classifier Code:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed 3/11/2020

@author: wscherer13

AMATH 582 - Final Project: Building an Improved Music Classifier
"""

import glob
import scipy.io.wavfile as wv
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.svm import SVC as SVC
from sklearn.naive_bayes import GaussianNB
from scipy import signal
from tabulate import tabulate
import pandas as pd
from matplotlib import pyplot as plt


def read_song(wav_file,tstart, incr):
    """
    This function reads in a wav file and creates a 5 second snippet of it

    """

    tend = tstart + incr+1

    fs, signal = wv.read(wav_file)

    mono = np.shape(signal)[1]

    if mono == 2:

        sig_mon = (signal[:,0]+signal[:,1])/2

    else:

        sig_mon = signal[0]

    sec_inc = sig_mon[tstart*fs:tend*fs]

    return(fs, sec_inc)


def spectrogram(song, fs):
    """
    Takes and input signal section and sampling rate and converts it to a
    spectrogram of fft and fft_shift of the input signal. Also produces the
    frequency space of the signal

    """
```

```
#    fq, t, spect = signal.spectrogram(song,fs,nperseg=int(fs*0.2))
    fq, t, spect = signal.spectrogram(song,fs)
    s_vec = np.reshape(np.real(spect),(np.shape(spect)[0]*np.shape(spect)[1],))
    spec_shape = np.shape(spect)

    s_vec = s_vec-np.average(spec_shape)

    return(s_vec, spec_shape)

def build_svec(filepath, nclips):
    """
    builds a matrix of spectrograms where each columns is a spectrogram
    of a 5 second clip of a song. Also returns the sampling rate and wave number
    vector of each spectrogram
    """
    song_mat = []

    song_path = os.path.join(str(filepath))
    song_files = glob.glob(song_path+'/*.*')
    i = 1
    while i <= nclips:
        start = np.random.randint(10,100)
        for song in song_files:

            fs, wav = read_song(song,start,4)

            fft_sig, shape = spectrogram(wav,fs)

            song_mat.append(fft_sig)

        i += 1

    song_mat = np.array(song_mat)

    return(song_mat, shape, fs)


def svd_songs(specmat):
    """
    This function performs the svd on an input spectrogram array
    without using full matrices and returns U, S, V from the decomposition
    """
    n = np.shape(specmat)[1]
    u, s, v = np.linalg.svd(np.transpose(specmat)/np.sqrt(n-1),full_matrices=False)

    sv = np.real(np.matmul(np.diag(s),v))

    return(u, sv, v, s)

def build_train(list_obs):
    """
    Takes in a list of the dataframes to be combined into the dataset

    """

    newpd = pd.concat(list_obs)
    newpd.reset_index(drop=True, inplace=True)

    return(newpd)

def train_models(dframe, nfeat, split):
```

```
    """
    Splits the input labeled dataframe into train and test splits,
    then trains a LDA classifier, SVM classifier, and KNN classifier

    Returns the success rate of each classifier on the test and train data
    """

    X_train, X_test, y_train, y_test = train_test_split(dframe.iloc[:,0:nfeat],
                                    dframe['band'], test_size = split)

    #Scale input data for classifiers
    scaler = MinMaxScaler()
    X_train = scaler.fit_transform(X_train)
    X_test  = scaler.fit_transform(X_test)

    #Train K Nearest Neighbors Classifier
    knn = KNeighborsClassifier()
    knn.fit(X_train,y_train)
    knn_train = knn.score(X_train,y_train)
    knn_test  = knn.score(X_test,y_test)

    #Train LInear Discriminant Analysis Classifier
    lda_t = LDA()
    lda_t.fit(X_train,y_train)
    lda_train = lda_t.score(X_train,y_train)
    lda_test = lda_t.score(X_test,y_test)


    #Train LInear Support Vector Classifier
    lsvc_t = SVC()
    lsvc_t.fit(X_train,y_train)
    lsvc_train = lsvc_t.score(X_train,y_train)
    lsvc_test  = lsvc_t.score(X_test,y_test)

    #Train Naive Bayes Classifier
    nbay_t = GaussianNB()
    nbay_t.fit(X_train,y_train)
    nbay_train = nbay_t.score(X_train,y_train)
    nbay_test  = nbay_t.score(X_test,y_test)

    return([knn_train,knn_test,lda_train,lda_test,lsvc_train,lsvc_test,nbay_train,nbay_test])


def avg_models(dframe, nfeat, split, nruns):
    """
    This fuction reads in a dataframe, passes it through the training function
    nrun times and reports the min, max, and average score on the test and
    train classification for each classifier algorithm

    """
    knn_train = []
    knn_test  = []
    lda_train = []
    lda_test  = []
    svc_train = []
    svc_test  = []
    nbay_train = []
    nbay_test  = []

    i = 1
    while i <= int(nruns):
```

```
        vals = train_models(dframe, nfeat, split)
        knn_train.append(vals[0])
        knn_test.append(vals[1])
        lda_train.append(vals[2])
        lda_test.append(vals[3])
        svc_train.append(vals[4])
        svc_test.append(vals[5])
        nbay_train.append(vals[6])
        nbay_test.append(vals[7])

        i += 1

    knn_train_dat = [min(knn_train), max(knn_train),
                np.average(knn_train),np.std(knn_train)]
    knn_test_dat = [min(knn_test), max(knn_test),
                np.average(knn_test),np.std(knn_test)]

    lda_train_dat = [min(lda_train), max(lda_train),
                np.average(lda_train),np.std(lda_train)]
    lda_test_dat = [min(lda_test), max(lda_test),
                np.average(lda_test), np.std(lda_test)]

    svc_train_dat = [min(svc_train), max(svc_train),
                np.average(svc_train), np.std(svc_train)]
    svc_test_dat = [min(svc_test), max(svc_test),
                np.average(svc_test), np.std(svc_test)]

    nbay_train_dat = [min(nbay_train), max(nbay_train),
                np.average(nbay_train), np.std(nbay_train)]
    nbay_test_dat = [min(nbay_test), max(nbay_test),
                np.average(nbay_test), np.std(nbay_test)]

    data = [knn_train_dat,knn_test_dat,lda_train_dat,lda_test_dat,svc_train_dat
          ,svc_test_dat,nbay_train_dat,nbay_test_dat]

    cols = ['KNN Train', 'KNN Test', 'LDA Train', 'LDA Test', 'SVC Train',
          'SVC Test', 'GNB Train', 'GNB Test']
    idx = ['Min', 'Max', 'Avg', 'Std']

    df = pd.DataFrame(np.transpose(data),columns = cols, index = idx)

    print('\nTrained Classifiers with',nfeat,'Features \n')
    print(tabulate(df, headers='keys', showindex = 'always', tablefmt='simple'))

    return(df)

def plot_results(case_dat, modes, split, nruns,title):
    """
    This function returns the plot of the different trained models

    """

    cases = []
    for mode in modes:

        case = avg_models(case_dat,mode,split,nruns)
        cases.append(case)


    i = 0
```

```
knn_train = []
knn_test  = []
lda_train = []
lda_test  = []
svc_train = []
svc_test  = []
nbay_train = []
nbay_test = []
for mode in modes:
   knn_train.append(100*cases[i]['KNN Train']['Avg'])
   knn_test.append( 100*cases[i]['KNN Test']['Avg'])
   lda_train.append(100*cases[i]['LDA Train']['Avg'])
   lda_test.append(100*cases[i]['LDA Test']['Avg'])
   svc_train.append(100*cases[i]['SVC Train']['Avg'])
   svc_test.append(100*cases[i]['SVC Train']['Avg'])
   nbay_train.append(100*cases[i]['GNB Train']['Avg'])
   nbay_test.append(100*cases[i]['GNB Test']['Avg'])
   i += 1

knn_err = []
knn_ert  = []
lda_err = []
lda_ert  = []
svc_err = []
svc_ert  = []
nbay_err = []
nbay_ert = []
j = 0
for mode in modes:
   knn_err.append(100*cases[j]['KNN Train']['Std'])
   knn_ert.append( 100*cases[j]['KNN Test']['Std'])
   lda_err.append(100*cases[j]['LDA Train']['Std'])
   lda_ert.append(100*cases[j]['LDA Test']['Std'])
   svc_err.append(100*cases[j]['SVC Train']['Std'])
   svc_ert.append(100*cases[j]['SVC Train']['Std'])
   nbay_err.append(100*cases[j]['GNB Train']['Std'])
   nbay_ert.append(100*cases[j]['GNB Test']['Std'])
   j += 1


fig1 = plt.figure(figsize=(9,5))
plt.subplot(1,2,1)
plt.errorbar(modes,knn_train, yerr=knn_err, label = 'KNN')
plt.errorbar(modes,lda_train, yerr=lda_err, label = 'LDA')
plt.errorbar(modes,svc_train, yerr=svc_err, label = 'SVC')
plt.errorbar(modes,nbay_train,yerr=nbay_err,label = 'GNB')
plt.xlabel('Number of Modes Trained On')
plt.ylabel('AVG Model Score (%)')
plt.title('Training Results')


plt.subplot(1,2,2)
plt.errorbar(modes,knn_test, yerr=knn_ert, label = 'KNN')
plt.errorbar(modes,lda_test, yerr=lda_ert, label = 'LDA')
plt.errorbar(modes,svc_test, yerr=svc_ert, label = 'SVC')
plt.errorbar(modes,nbay_test,yerr=nbay_ert,label = 'GNB')
plt.xlabel('Number of Modes Tested On')
plt.ylabel('AVG Model Score (%)')
plt.legend(loc='center left', bbox_to_anchor=(1,0.5))
plt.title('Testing Results')
```

```
    plt.tight_layout()
    plt.savefig(title+'.png')

    return(fig1, cases)

def energy_s(s_mat):
    """
    Returns the energy of each component of the s matrix from SVD decomposition

    s is a list of singular values

    """

    itr = int(np.shape(s_mat)[0])
    sums = sum(s_mat)
    norm_1 = s_mat[0]/sums
    norm_c = []
    norm_i = []

    for i in range(itr):

        norm_c.append(sum(s_mat[0:i+1])/sums)
        norm_i.append(s_mat[i]/sums)
    return(norm_1, norm_c, norm_i)

def plot_svd(sar,title,labels):
    """
    Plots the principle components of a matrix
    """


    xrang = range(1,len(sar[0])+1)
    fig = plt.figure(figsize=(6,6))
    plt.subplot(2,1,1)
    plt.plot(xrang,sar[0],'ko',label = str(labels[0]))
    plt.plot(xrang,sar[1],'rx',label = str(labels[1]))
    plt.plot(xrang,sar[2],'g*',label = str(labels[2]))
    try:
        plt.plot(xrang,sar[3],'co',label = str(labels[3]))
        plt.ylabel('Sigma Value')
        plt.legend()
        plt.title(str(title))
    except:
        plt.ylabel('Sigma Value')
        plt.legend()
        plt.title(str(title))

    plt.subplot(2,1,2)
    plt.semilogy(xrang,sar[0],'ko',label = str(labels[0]))
    plt.semilogy(xrang,sar[1],'rx',label = str(labels[1]))
    plt.semilogy(xrang,sar[2],'g*',label = str(labels[2]))
    try:
        plt.semilogy(xrang,sar[3],'co',label = str(labels[3]))
        plt.xlabel('Component Mode')
        plt.ylabel('log10 Sigma Value')
    except:
        plt.xlabel('Component Mode')
        plt.ylabel('log10 Sigma Value')

    plt.savefig(str(title)+'.png')
```

```
    return(fig)

def rescale_data(svec_list):
    """
    This function rescales the SV* and V matrices by taking the absolute
    value of each matric and then the log10 of each matrix - elementwise.
    If a value is -inf, it is replaced by -30
    """
    svec_rsc = []

    for svec in svec_list:
        imt = np.log10(np.abs(svec))
        svec_rsc.append(imt.replace(-np.inf,-30))

    return(svec_rsc)


#%%
# Load in training data sets

# Import police clips
pol_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/police'
police_mat, sp_p, fs_p = build_svec(pol_path,15)

pol_u, pol_sv, pol_v, pol_s = svd_songs(police_mat)

pol_sv = pd.DataFrame(pol_sv)
pol_v = pd.DataFrame(pol_v)
#Feature Scaling
#pol_sv = pd.DataFrame(np.log10(np.abs(pol_sv)))
#pol_v = pd.DataFrame(np.log10(np.abs(pol_v)))

pol_sv['band']='Police'; pol_v['band']='Police'


#Import lord of the rings clips
lotr_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/lotr'
lotr_mat, sp_l, fs_l = build_svec(pol_path,15)

lotr_u, lotr_sv, lotr_v, lotr_s = svd_songs(lotr_mat)
lotr_sv = pd.DataFrame(lotr_sv)
lotr_v = pd.DataFrame(lotr_v)

#Feature Scaling
#lotr_sv = pd.DataFrame(np.log10(np.abs(lotr_sv)))
#lotr_v = pd.DataFrame(np.log10(np.abs(lotr_v)))

lotr_sv['band']='LoTR'; lotr_v['band']='LoTR'
#

#Import Otis clips
otis_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/otis'
otis_mat, sp_o, fs_o = build_svec(pol_path,15)
#
otis_u, otis_sv, otis_v, otis_s = svd_songs(otis_mat)
otis_sv = pd.DataFrame(otis_sv)
otis_v = pd.DataFrame(otis_v)

#Feature Scaling
#otis_sv = pd.DataFrame(np.log10(np.abs(otis_sv)))
#otis_v = pd.DataFrame(np.log10(np.abs(otis_v)))
```

```
otis_sv['band']='Otis'; otis_v['band']='Otis'

#create labeled data sets

#%%
#Case 1 Build Model
case1_dat = build_train([otis_sv.replace(-np.inf,-30),
                    lotr_sv.replace(-np.inf,-30),
                    pol_sv.replace(-np.inf,-30)])

otis1, otisc, otisi = energy_s(otis_s)
ploice1, policec, policei = energy_s(pol_s)
lotr1, lotrc, lotri = energy_s(lotr_s)

svecs1 = [otis_s,pol_s,lotr_s]
labels1 = ['otis','police','lotr']
title1 = 'Case 1 Principle Components'
# Plotting the energies of principle components for each artist
fig1 = plot_svd(svecs1,title1,labels1)


fig2 = plt.figure(figsize=(6,5))
plt.subplot(2,1,1)
plt.plot(range(1,np.shape(otis_s)[0]+1),otisi,'kx', label = 'otis')
plt.plot(range(1,np.shape(pol_s)[0]+1),policei,'ro', label = 'police')
plt.plot(range(1,np.shape(lotr_s)[0]+1),lotri,'g*', label = 'lotr')
plt.ylabel('Component Energy')
plt.title('Case 1 Component Energies')
plt.legend()

plt.subplot(2,1,2)
plt.plot(range(1,np.shape(otis_s)[0]+1),otisc,'kx', label = 'otis')
plt.plot(range(1,np.shape(pol_s)[0]+1),policec,'ro', label = 'police')
plt.plot(range(1,np.shape(lotr_s)[0]+1),lotrc,'g*', label = 'lotr')
plt.ylabel('Cumulative Energy')
plt.xlabel('Component Mode')
plt.savefig('case1_snorm_noscale.png')

#%% train models case 1

fig1, case1 = plot_results(case1_dat, [5,10,25,50, 100,150],0.20,10,'Artists_score_noscale_30')

#%%
#Case 2 Build Band Classifier in Same Genre

# Led Zeppelin Data
led_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/led_zep'
led_mat, kp_l, fs_l = build_svec(led_path,15)

led_u, led_sv, led_v, led_s = svd_songs(led_mat)

led_sv = pd.DataFrame(np.log10(np.abs(led_sv)))
led_v = pd.DataFrame(np.log10(np.abs(led_v)))

led_sv['band']='Zeppelin'; led_v['band']='Zeppelin'


# Jimi Hendrix Data
jimi_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/jimi'
jimi_mat, kp_j, fs_j = build_svec(jimi_path,15)
```

```
jimi_u, jimi_sv, jimi_v, jimi_s = svd_songs(jimi_mat)

jimi_sv = pd.DataFrame(np.log10(np.abs(jimi_sv)))
jimi_v = pd.DataFrame(np.log10(np.abs(jimi_v)))

jimi_sv['band']='Hendrix'; jimi_v['band']='Hendrix'

# Stones Hendrix Data
stones_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/stones'
stones_mat, kp_j, fs_j = build_svec(stones_path,15)

stones_u, stones_sv, stones_v, stones_s = svd_songs(stones_mat)

stones_sv = pd.DataFrame(np.log10(np.abs(stones_sv)))
stones_v = pd.DataFrame(np.log10(np.abs(stones_v)))

stones_sv['band']='Stones'; stones_v['band']='Stones'
#

#%%
case2_dat = build_train([led_sv.replace(-np.inf,-30),
                    jimi_sv.replace(-np.inf,-30)
                    ,stones_sv.replace(-np.inf,-30)])
#Case 2 Build Models
led1, ledc, ledi = energy_s(led_s)
jimi1, jimic, jimii = energy_s(pol_s)
stones1, stonesc, stonesi = energy_s(stones_s)

svecs2 = [led_s,jimi_s,stones_s]
labels2 = ['Led Z','Jimi','Stones']
title2 = 'Case 2 Principle Components'
# Plotting the energies of principle components for each artist
fig1 = plot_svd(svecs2,title2,labels2)


fig2 = plt.figure(figsize=(6,5))
plt.subplot(2,1,1)
plt.plot(range(1,np.shape(led_s)[0]+1),ledi,'kx', label = 'led z')
plt.plot(range(1,np.shape(jimi_s)[0]+1),jimii,'ro', label = 'jimi')
plt.plot(range(1,np.shape(stones_s)[0]+1),stonesi,'g*', label = 'stones')
plt.ylabel('Component Energy')
plt.title('Case 2 Component Energies')
plt.legend()

plt.subplot(2,1,2)
plt.plot(range(1,np.shape(led_s)[0]+1),ledc,'kx', label = 'led z')
plt.plot(range(1,np.shape(jimi_s)[0]+1),jimic,'ro', label = 'jimi')
plt.plot(range(1,np.shape(stones_s)[0]+1),stonesc,'g*', label = 'stones')
plt.ylabel('Cumulative Energy')
plt.xlabel('Component Mode')
plt.savefig('case2_snorm_150.png')

#%%
#Case 2 Train Models

fig2, case2 = plot_results(case2_dat, [5,10,25,50,100,150],0.20,10,'CRock_score_150')

#%%
#Case 3 Import Data
#Rock Data
rock_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/rock'
```

```
rock_mat, kp_r, fs_r = build_svec(rock_path,15)

rock_u, rock_sv, rock_v, rock_s = svd_songs(rock_mat)

rock_sv = pd.DataFrame(np.log10(np.abs(rock_sv)))
rock_v = pd.DataFrame(np.transpose(np.log10(np.abs(rock_v))))

rock_sv['band']='Rock'; rock_v['band']='Rock'


# Jazz Data
jazz_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/jazz'
jazz_mat, kp_j, fs_j = build_svec(jazz_path,15)

jazz_u, jazz_sv, jazz_v, jazz_s = svd_songs(jazz_mat)

jazz_sv = pd.DataFrame(np.log10(np.abs(jazz_sv)))
jazz_v = pd.DataFrame(np.transpose(np.log10(np.abs(jazz_v))))

jazz_sv['band']='Jazz'; jazz_v['band']='Jazz'

# Sound Track Data
soul_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/soul'
soul_mat, kp_t, fs_t = build_svec(soul_path,15)

soul_u, soul_sv, soul_v, soul_s = svd_songs(soul_mat)

soul_sv = pd.DataFrame(np.log10(np.abs(soul_sv)))
soul_v = pd.DataFrame(np.transpose(np.log10(np.abs(soul_v))))

soul_sv['band']='Soul'; soul_v['band']='Soul'
##

#%%
#Case 3 Build Models

case3_dat = build_train([rock_sv.replace(-np.inf,-30),
                 jazz_sv.replace(-np.inf,-30),
                 soul_sv.replace(-np.inf,-30)])

rock1, rockc, rocki = energy_s(rock_s)
jazz1, jazzc, jazzi = energy_s(jazz_s)
soul1, soulc, souli = energy_s(soul_s)

svecs3 = [rock_s,jazz_s,soul_s]
labels3 = ['Rock','Jazz','Soul']
title3 = 'Case 3 Principle Components'
# Plotting the energies of principle components for each artist
fig1 = plot_svd(svecs3,title3,labels3)


fig2 = plt.figure(figsize=(6,5))
plt.subplot(2,1,1)
plt.plot(range(1,np.shape(rock_s)[0]+1),rocki,'kx', label = 'rock')
plt.plot(range(1,np.shape(jazz_s)[0]+1),jazzi,'ro', label = 'jazz')
plt.plot(range(1,np.shape(soul_s)[0]+1),souli,'g*', label = 'soul')
plt.ylabel('Component Energy')
plt.title('Case 3 Component Energies')
plt.legend()

plt.subplot(2,1,2)
```

```
plt.plot(range(1,np.shape(rock_s)[0]+1),rockc,'kx', label = 'rock')
plt.plot(range(1,np.shape(jazz_s)[0]+1),jazzc,'ro', label = 'jazz')
plt.plot(range(1,np.shape(soul_s)[0]+1),soulc,'g*', label = 'soul')
plt.ylabel('Cumulative Energy')
plt.xlabel('Component Mode')
plt.savefig('case3_snorm_150.png')



fig3, case3 = plot_results(case3_dat, [5,10,25,50,100,150],0.30,10,'Genre_score_150')
#%%
#Case 4 Rage Against The Machine - 4 Albums

# Rage Against of the Machine
ratm_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/ratm'
ratm_mat, kp_r, fs_r = build_svec(ratm_path,15)

ratm_u, ratm_sv, ratm_v, ratm_s = svd_songs(ratm_mat)

ratm_sv = pd.DataFrame(np.log10(np.abs(ratm_sv)))
ratm_v = pd.DataFrame(np.transpose(np.log10(np.abs(ratm_v))))

ratm_sv['band']='RATM'; ratm_v['band']='RATM'

# Evil Empire Album
evil_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/evile'
evil_mat, kp_e, fs_e = build_svec(evil_path,15)

evil_u, evil_sv, evil_v, evil_s = svd_songs(evil_mat)

evil_sv = pd.DataFrame(np.log10(np.abs(evil_sv)))
evil_v = pd.DataFrame(np.transpose(np.log10(np.abs(evil_v))))

evil_sv['band']='EVILE'; evil_v['band']='EVILE'

# Renegades Album
reneg_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/reneg'
reneg_mat, kp_re, fs_re = build_svec(reneg_path,15)

reneg_u, reneg_sv, reneg_v, reneg_s = svd_songs(reneg_mat)

reneg_sv = pd.DataFrame(np.log10(np.abs(reneg_sv)))
reneg_v = pd.DataFrame(np.transpose(np.log10(np.abs(reneg_v))))

reneg_sv['band']='RENEG'; reneg_v['band']='RENEG'

# The Battle of Los Angeles
tbola_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/Final/tbola'
tbola_mat, kp_tb, fs_tb = build_svec(tbola_path,15)

tbola_u, tbola_sv, tbola_v, tbola_s = svd_songs(tbola_mat)

tbola_sv = pd.DataFrame(np.log10(np.abs(tbola_sv)))
tbola_v = pd.DataFrame(np.transpose(np.log10(np.abs(tbola_v))))

tbola_sv['band']='TBOLA'; tbola_v['band']='TBOLA'
#%% RATM Build Models
RATM_dat = build_train([ratm_sv.replace(-np.inf,-30),
                        evil_sv.replace(-np.inf,-30),
                        reneg_sv.replace(-np.inf,-30),
                        tbola_sv.replace(-np.inf,-30)])
```

```
ratm1, ratmc, ratmi = energy_s(ratm_s)
evil1, evilc, evili = energy_s(evil_s)
reneg1, renegc, renegi = energy_s(reneg_s)
tbola1, tbolac, tbolai = energy_s(tbola_s)

svecs4 = [ratm_s,evil_s,reneg_s, tbola_s]
labels4 = ['RATM','EVIL_E','RENEG','TBOLA']
title4 = 'Rage Against the Machine Principle Components'
# Plotting the energies of principle components for each artist
fig4 = plot_svd(svecs4,title4,labels4)


fig5 = plt.figure(figsize=(6,5))
plt.subplot(2,1,1)
plt.plot(range(1,np.shape(ratm_s)[0]+1),ratmi,'kx', label = 'RATM')
plt.plot(range(1,np.shape(evil_s)[0]+1),evili,'ro', label = 'EVIL_E')
plt.plot(range(1,np.shape(reneg_s)[0]+1),renegi,'g*', label = 'RENEG')
plt.plot(range(1,np.shape(tbola_s)[0]+1),tbolai,'co', label = 'TBOLA')
plt.ylabel('Component Energy')
plt.title('Rage Against the Machine Component Energies')
plt.legend()

plt.subplot(2,1,2)
plt.plot(range(1,np.shape(ratm_s)[0]+1),ratmc,'kx', label = 'RATM')
plt.plot(range(1,np.shape(evil_s)[0]+1),evilc,'ro', label = 'EVIL_E')
plt.plot(range(1,np.shape(reneg_s)[0]+1),renegc,'g*', label = 'RENEG')
plt.plot(range(1,np.shape(tbola_s)[0]+1),tbolac,'co', label = 'TBOLA')
plt.ylabel('Cumulative Energy')
plt.xlabel('Component Mode')
plt.tight_layout()
plt.savefig('RATM_snorm_150.png')


#plt.figure(figsize=(8,12))
fig4, case4 = plot_results(RATM_dat, [5,10,25,50,100,150],0.30,10,'RATM_score_150')

#%% Classify songs within Evil Empire
```