Image Principle Component Analysis and Music Classification Using Singular Value Decomposition

Wyatt Scherer – *University of Washington*

AMATH 582 HW 4, Due 3/06/2020

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

ABSTRACT

Part one of this analysis uses the SVD algorithm to analyze and compare images from the YaleFaces database. The first case uses SVD to analyze cropped images of the subjects' faces. The second case uses SVD to analyze the full images of the subjects' faces. The effect of cropping the images on the resulting principle components of the decomposed images was compared. Part two of this analysis uses the SVD algorithm and three classifier algorithms to classify music by decomposing the spectrogram of 5-second clips of music. The three cases considered are 1) classify music by artist, 2) classify music by band within a single genre, and 3) classify music by genre.

1. Introduction

Singular Value Decomposition (SVD) is a useful analysis tool that assists in reducing high-dimensional data into lower dimensional forms. These lower dimensional representations are often referred to as *principle components (Principle Component Analysis –PCA)* or *proper modes (Proper Mode Decomposition – POD)*. Manipulating lower dimensional data is often easier and more intuitive than dealing with data in its raw form, making PCA a smart first choice for analyzing large data sets [**Error! Reference source not found.**].

The first part of this analysis uses SVD to analyze image data from the Yale Faces database. Two image datasets are used: 1) cropped face images and 2) full images. For both datasets, all the images reformatted so they can be mathematically manipulated. The number of principle components (PCs) needed to recreate the images is compared for both data sets and the differences between the PCs of each data set are compared.

The second part of this analysis uses SVD and pre-built classifier algorithms to classify music of different bands and genres. First the SVD algorithm is used to decompose the spectrogram of 5-second music clips. The PCs from this decomposition are then fed into classification algorithms for classifying different artists and genres based on the PCs. Three common classifier algorithms are used to classify the song data: 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 are 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. The performances of each classification algorithm are compared for varying numbers of PC features considered.

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 $\vec{\boldsymbol{A}}$ into three matrices **U, Σ,** and **V** that satisfy the following relationship [**Error! Reference source not found.**]:

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

where

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

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]:

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

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

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

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 [**Error! Reference source not found.**].

2.2. Overview of Classification Algorithms Used

For Part Two of this analysis, songs from various artists are filtered and used to classify songs by artist and genre. For brevity, only three classification algorithms were implemented in this analysis: KNN, LDA, and SVC. 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 [3].

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 [4].

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 [5].

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. Part One: Yale Faces Principle Component Analysis

Note: for the cropped Yale Faces database, there are 39 folders (1 folder per subject) with 64 photos in each folder: a total of 2496 photos. For the full Yale Faces database, there are 15 folders with 11 photos in each folder: a total of 165 photos.

1) Read in all photos within a subject subfolder as a greyscale photo reshaped into a column vector. Each photo is concatenated into a matrix where each column is a reshaped photo. The custom *import_image* and *import_imagef* functions perform this. *import_image and import_imagef* use different photo import methods since the photo formats were different for each database.

2) Create an array of photo matrices of each database where each matrix is the photo matrix for each subject. The custom function *read_files* calls the custom import image functions, and returns an array where each element in the array is the photo matrix for a single subject. All formats are converted to float for mathematical processing. Two arrays are created: the cropped image database array with 39 matrices and the full image database array with 15 matrices.
3) To process all of the cropped images at once and all of the full images at once, these database arrays are flattened into a single matrix using the custom *flatten_npar* function. This function concatenates all the matrices stored in each database array into a single matrix. Two matrices are created: the cropped image database array with 2496 columns and the full image database array with 165 columns.
4) Two SVD approaches were implemented. The first approach creates a covariance matrix of the full image matrix by subtracting out the average of each column and dividing by the square root of the number of photos minus 1. The principle components of the covariance matrix should show where the largest changes between the photos are in each database. The second approach was to perform SVD on the unmodified image matrix. The principle components of the raw image data should effectively be 'eigenvalues' of all the faces in the matrix. These analyses are performed by the custom *svd_images* function, which is effectively a wrapper around the python numpy svd function. For this analysis, full matrices are not used, which means a reduced SVD algorithm is used to save time.
5) Now that the image data is plotted, the principle components can be plotted and analyzed. Custom functions *energy_s* and *eig_faces* plot the energy of the principle components and cumulative image modes respectively

3.2. Part Two: Music Classification using SVD

Three different classification scenarios are 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. The performances of each classification algorithm are compared for varying numbers of PC features considered. The algorithm for processing the song data and training the classifiers is the same for each case. What differs between each case are the databases of songs used and the classifications of the data.

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 5 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. 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.
6) 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, and SVC 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.
7) 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. This allows the performance of each model to be easily compared.

4. Results

4.1. Part 1 – Principle Component Analysis of Yale Faces Database

As described in Section 3.1, there are two Yale Faces databases: Cropped Images and Full Images. For the Cropped Yale Faces database, there are 39 folders (1 folder per subject) with 64 photos in each folder: a total of 2496 photos. For the full Yale Faces database, there are 15 folders with 11 photos in each folder: a total of 165 photos. After importing all of the cropped faces into one matrix, the SVD analysis was performed on the cropped photos. The singular values of the covariance matrix and unmodified matrix for the Cropped Database are below in Figure 1. The energy in for each principle component is in Figure 2. As can be seen from the $\log_{10}$ plots, there are principle components that are significantly higher than zero for both the covariance matrix and the raw image matrix. This suggests a non-Gaussian distribution in the image data when all the images are compared at once. Figure 2 shows the energy of each component and the cumulative energy of each component, along with the plot of the first

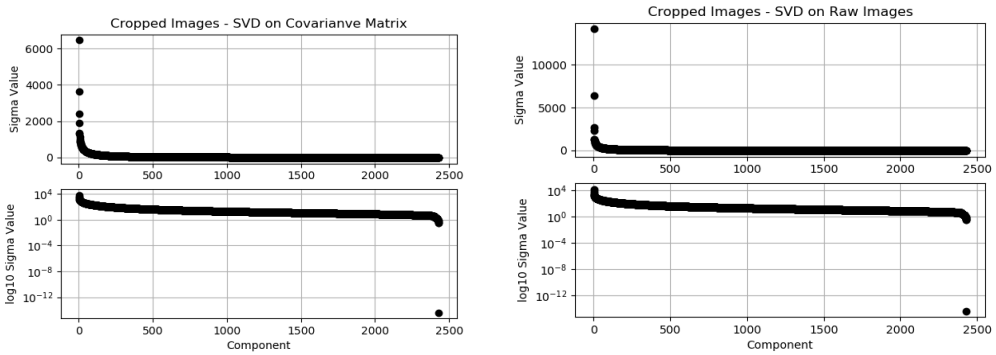

Figure 1: Principle Components for the Cropped Image Covariance and Raw Images

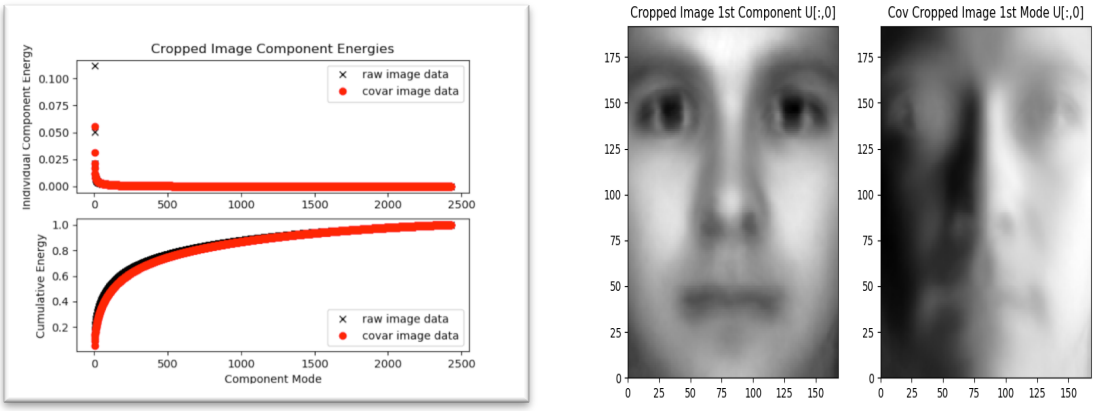

Figure **2**: Cropped Image Component Energies (left) and Reshaped 1st Principle Components U[:,0] (right)

To see the affect of the different modes of the covariance matrix, the cumulative sum of reconstructed Principle Component Images U[:,i] are presented in Figure 3 below.

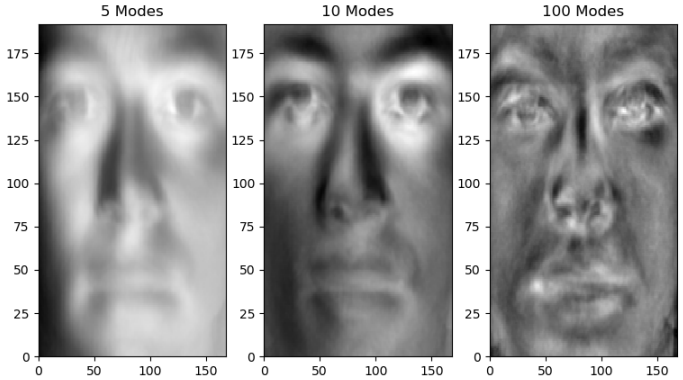

Figure **3**: Reconstruction of the Cropped Covariance Matrix Images U[:1-n]

The same analysis was performed for the Full Yale Faces Database as well. Since these photos are not cropped, the position of the subject's head in each frame changes. When all of the subjects are combined into one matrix, the effect of all of the shifting locations of the subject's creates a noisy matrix. Since there are less photos in this Full Images database, there are less principle modes to explore. The following figures show all the same information as Figures 1-4, except for the Full Images data set. As can be seen in

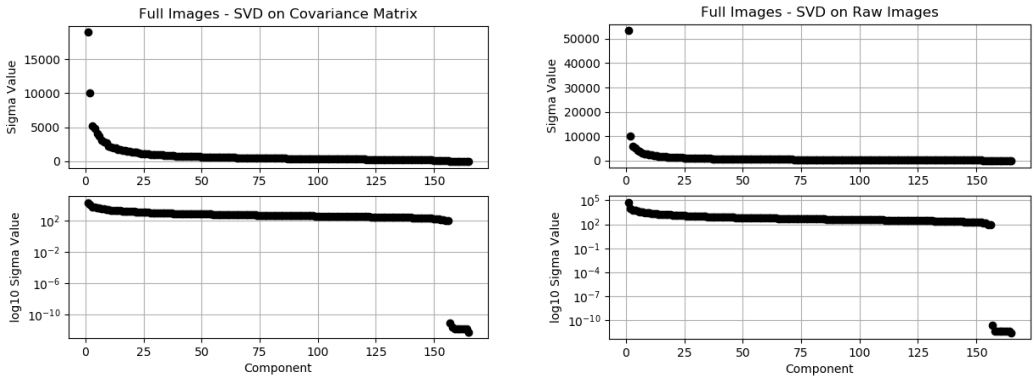

Figure 4: Principle Components for the Full Image Covariance and Raw Images

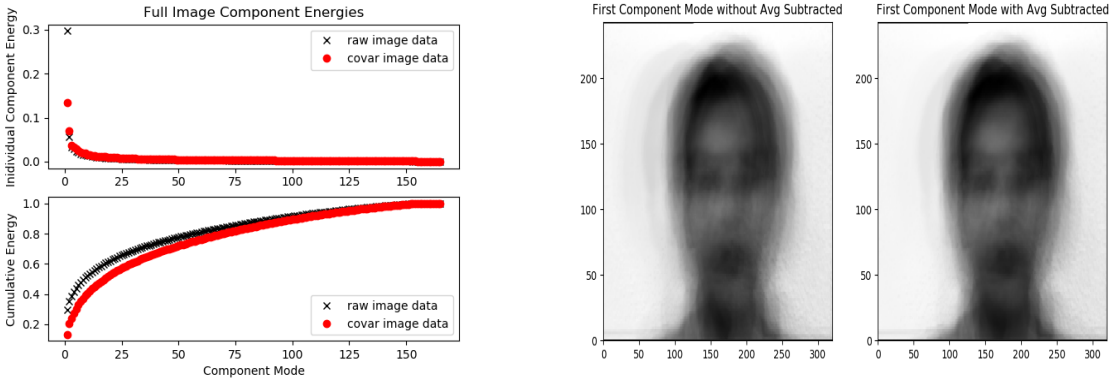

Figure **5**: Full Image Component Energies (left) and Reshaped 1st Principle Components U[:,0] (right)

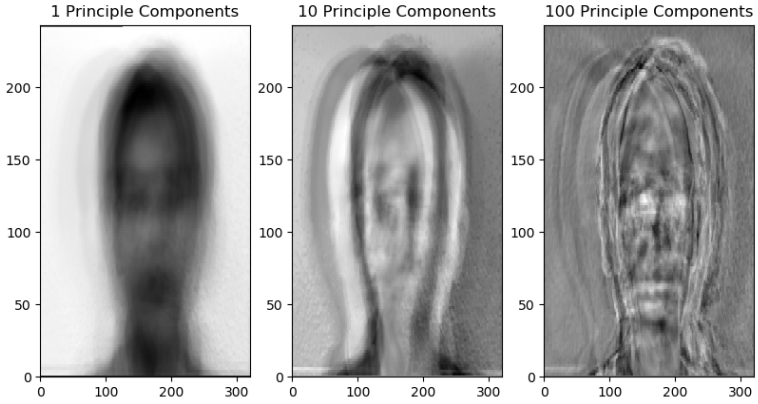

Figure **6**: Reconstruction of the Cropped Covariance Matrix Images U[:1-n]

As expected, with the fewer photos (less data) and more movement in the photos, the average images produced frorm the Full Image dataset are much noisier – Figure 5. Figure 6 shows how much the effect of each subject moving between each photo and between each subject affects the results. What is interesting is that the SVD algorithm effectively crops the ful images to show minimal white space – which makes sense because SVD captures principle components and meaningless background will be a low worth principle component (high component number).

4.2. Part 2) Music Classification Results

For all three analyses, there are three possible classifications. Each of the three classification models were 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, and SVC training models were used. The data train to test ratio is 0.10, meaning 10% of the input data is used for test data and 90% is used for training. 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%.

4.2.1. Case 1) Artist Classification Results

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 50 times to create the spectrograms of the 5-second clips for a total of 500 clips per artist.

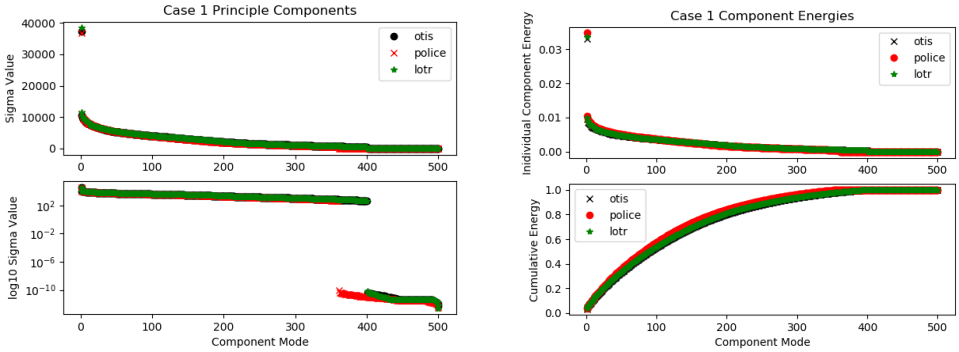

Figure 7: Music Classification Case 1 Principle Components

Trained Classifiers with 5 Features

	KNN Train	KNN Test	LDA Train	LDA Test	SVC Train	SVC Test
Min	0.485926	0.226667	0.357778	0.253333	0.334074	0.26
Max	0.514074	0.353333	0.380741	0.406667	0.341481	0.326667
Avg	0.499481	0.302	0.368667	0.342	0.337407	0.296667
Std	0.00895684	0.0385343	0.00588924	0.0407758	0.0019387	0.0174483

Trained Classifiers with 100 Features

	KNN Train	KNN Test	LDA Train	LDA Test	SVC Train	SVC Test
Min	0.506667	0.306667	0.46	0.3	0.334074	0.273333
Max	0.545185	0.4	0.491111	0.393333	0.40963	0.326667
Avg	0.528222	0.336667	0.470519	0.341333	0.344296	0.302
Std	0.0122231	0.0287904	0.00863718	0.0254384	0.0218758	0.0195619

Trained Classifiers with 250 Features

	KNN Train	KNN Test	LDA Train	LDA Test	SVC Train	SVC Test
Min	0.342963	0.253333	0.561481	0.233333	0.334815	0.226667
Max	0.433333	0.44	0.574815	0.42	0.345185	0.32
Avg	0.392889	0.327333	0.568667	0.336	0.339037	0.282
Std	0.022607	0.0550918	0.00342748	0.0490079	0.00321263	0.0289137

Trained Classifiers with 500 Features

	KNN Train	KNN Test	LDA Train	LDA Test	SVC Train	SVC Test
Min	0.174074	0.173333	0.685185	0.24	0.334815	0.28
Max	0.222963	0.293333	0.71037	0.4	0.339259	0.32
Avg	0.204444	0.237333	0.699185	0.313333	0.337556	0.295333
Std	0.0149989	0.0386667	0.00771545	0.0382971	0.00132715	0.0119443

Figure 8: Classification Results for Case 1

4.2.2.Case 2) Artist Classification within Same Genre Results – Classic Rock

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 50 times to create the spectrograms of the 5-second clips for a total of 500 clips per artist.

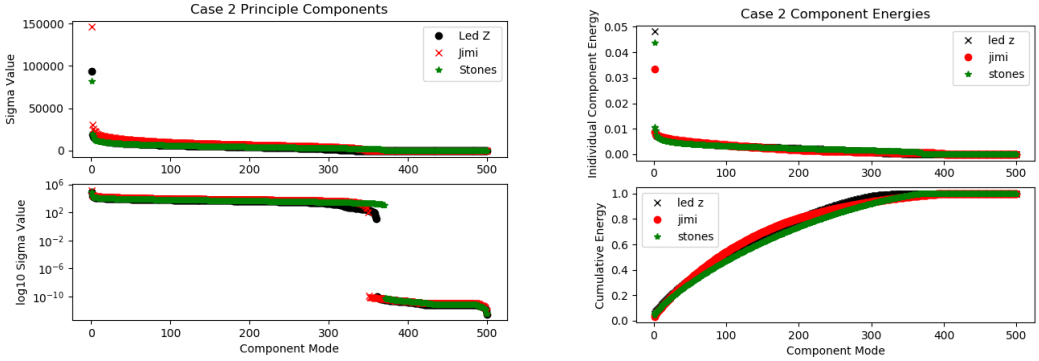

Figure 9: Music Classification Case 2 Principle Components

```
Trained Classifiers with 5 Features

      KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---   ----------   ---------   ----------   ---------   ----------   ---------
Min     0.683704    0.266667     0.277778    0.226667    0.334815    0.246667
Max     0.752593    0.373333     0.403704    0.346667    0.342963    0.32
Avg     0.719185    0.307333     0.327111    0.304667    0.33763     0.294667
Std     0.0167822   0.0307607    0.0359686   0.0357833   0.00236109  0.0212498
/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_analysis.py:442: UserWarning: The priors do not sum to
1. Renormalizing
  UserWarning)

Trained Classifiers with 100 Features

      KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---   ----------   ---------   ----------   ---------   ----------   ---------
Min     0.344444    0.253333     0.418519    0.26        0.335556    0.246667
Max     0.383704    0.42         0.448148    0.38        0.342963    0.313333
Avg     0.363333    0.344667     0.437037    0.309333    0.338296    0.288667
Std     0.0136274   0.0588822    0.00854264  0.0355528   0.00214815  0.0193333

Trained Classifiers with 250 Features

      KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---   ----------   ---------   ----------   ---------   ----------   ---------
Min     0.158519    0.32         0.528148    0.273333    0.335556    0.266667
Max     0.21037     0.406667     0.546667    0.413333    0.340741    0.313333
Avg     0.185926    0.372        0.538963    0.34        0.338667    0.285333
Std     0.0134031   0.0271293    0.0061101   0.0449197   0.00168264  0.0151438

Trained Classifiers with 500 Features

      KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---   ----------   ---------   ----------   ---------   ----------   ---------
Min     0.111111    0.173333     0.668889    0.226667    0.334074    0.266667
Max     0.184444    0.366667     0.692593    0.38        0.340741    0.326667
Avg     0.148222    0.264667     0.677852    0.323333    0.336741    0.302667
Std     0.0232773   0.0702409    0.00751368  0.0460193   0.00207407  0.0186667
```

Figure 10: Classification Results for Case 2

4.3. 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 50 times to create the spectrograms of the 5-second clips for a total of 500 clips per genre.

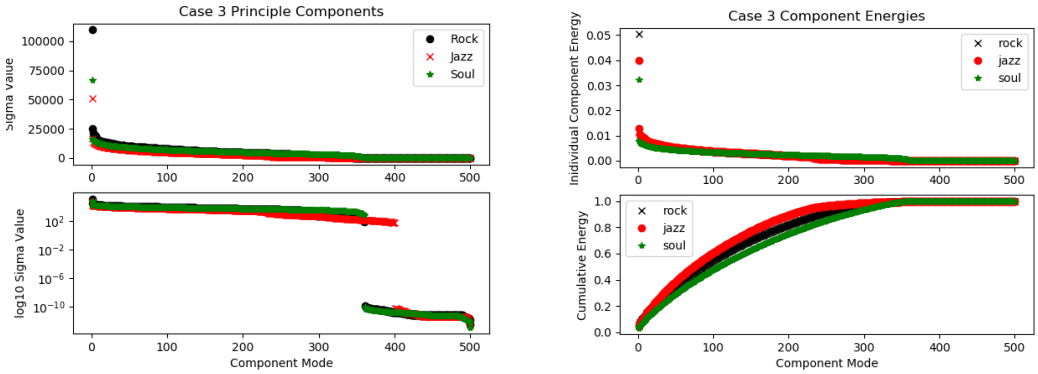

Figure 11: Music Classification Case 3 Principle Components

```
Trained Classifiers with 5 Features

       KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---  -----------  ----------  -----------  ----------  -----------  ----------
Min     0.683704    0.266667     0.277778    0.226667     0.334815    0.246667
Max     0.752593    0.373333     0.403704    0.346667     0.342963    0.32
Avg     0.719185    0.307333     0.327111    0.304667     0.33763     0.294667
Std     0.0167822   0.0307607    0.0359686   0.0357833    0.00236109  0.0212498
/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_analysis.py:442: UserWarning: The priors do not sum to
1. Renormalizing
  UserWarning)

Trained Classifiers with 100 Features

       KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---  -----------  ----------  -----------  ----------  -----------  ----------
Min     0.344444    0.253333    0.418519     0.26        0.335556    0.246667
Max     0.383704    0.42        0.448148     0.38        0.342963    0.313333
Avg     0.363333    0.344667    0.437037     0.309333    0.338296    0.288667
Std     0.0136274   0.0588822   0.00854264   0.0355528   0.00214815  0.0193333

Trained Classifiers with 250 Features

       KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---  -----------  ----------  -----------  ----------  -----------  ----------
Min     0.158519    0.32        0.528148     0.273333    0.335556    0.266667
Max     0.21037     0.406667    0.546667     0.413333    0.340741    0.313333
Avg     0.185926    0.372       0.538963     0.34        0.338667    0.285333
Std     0.0134031   0.0271293   0.0061101    0.0449197   0.00168264  0.0151438

Trained Classifiers with 500 Features

       KNN Train    KNN Test    LDA Train    LDA Test    SVC Train    SVC Test
---  -----------  ----------  -----------  ----------  -----------  ----------
Min     0.111111    0.173333    0.668889     0.226667    0.334074    0.266667
Max     0.184444    0.366667    0.692593     0.38        0.340741    0.326667
Avg     0.148222    0.264667    0.677852     0.323333    0.336741    0.302667
Std     0.0232773   0.0702409   0.00751368   0.0460193   0.00207407  0.0186667
```

Figure 12: Classification Results for Case 3

5. Conclusions

5.1. Part 1) Principle Component Analysis of Yale Faces Database

Figures 2 and 5 show the clear affect that cropping has on the quality and meaning of the principle modes of an image. The average and covariance cropped face results were much clearer and easier to interpret than the full image results. For taking the SVD on the raw image data, U seems to correspond to eigenfaces or modes of an average face, S can be interpreted as the weight with which the particular eigenface should be incorporated, and V is the bases vectors of all the eigenfaces. Taking the SVD on the covariance data seems to highlight features that change the most between subjects, like cheek bones, eye position, and mouth shape. Here U seems to correspond to the visualization of important features, S is the weight of how important that feature is, and V is the bases that the features are expressed on.

5.2. Part 2) Music Classification

For all three cases, none of the classifiers performed statistically better classifying test data than random assignment regardless of the amount of features used – see Figures 8, 10, and 12. Figures 7, 9, and 11 all show that even the primary principal component for every possible classification only represents < 6% of the total energy of the 5-second spectrograms. For all three cases, LDA seemed to perform the best when scoring itself on the training data (see Figure 8 – 500 Features), but even LDA was not statistically significant on test data. This seems to suggest that the spectrograms used in this analysis need to be improved and that the default spectrogram settings are not sufficient for meaningful analysis. All three classifiers performed poorly on the training and test data, signaling that the input data was insufficient for classification. Humans can classify music with relative ease and for Case 1, the artists are very different: rock, movie soundtrack, and soul. A human could easily tell those artists apart whereas the trained KNN, LDA, and SVC classifiers were unable. To improve the classification score for all three cases more song data could be helpful, and better spectrogram data should be used. A suggestion for further improvement would be to try different waveforms within the spectrogram and try filtering over different number of points along the 5-second clip. Part 2 of this analysis demonstrates the difficulties in classifying poorly cleaned data sets, which are common in most real world applications.

6. References

1. "Chs. 15-17." *Data-Driven Modeling Et Scientific Computation: Methods for Complex Systems Et Big Data*, by J. Nathan. Kutz, Oxford Univ. Press, 2013.
2. Belhumeur P., Kriegnan D., 'The Yale Face Database', Yale University, https://www.cs.yale.edu/cvc/projects/yalefaces/yalefaces.html
3. 'K-nearest neighbors algorithm', Wikipedia, https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
4. 'Linear Discriminant Analysis', Wikipedia, https://en.wikipedia.org/wiki/Linear_discriminant_analysis
5. 'Support-vector machine', Wikipedia, https://en.wikipedia.org/wiki/Support-vector_machine

APPENDIX A – GENERIC PYTHON FUNCTIONS USED

- *np.shape* returns the dimensions of a numpy array
- *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

Part 1 Code:

```
"""
Created on Tue Feb 25 18:07:07 2020

@author: wscherer13
"""
import numpy as np
import pywt as wv
import cv2
import os
import glob
from matplotlib import pyplot as plt


def import_image(filepath):
    """
    This function takes a filepath leading to a file with images
    to be imported.  This function uses the opencv function to read in images
    and convert them to greyscale.
    The images in the file are appended to
    a column vector, which is returned by the function
    """

    im_path = os.path.join(str(filepath))
    im_files = glob.glob(im_path)
    im_col = []

    for imc in im_files:
        #imc = plt.imread(img, 0) # changed from cv2.imread
        img = cv2.imread(imc, cv2.COLOR_BGR2GRAY)
        imcg = np.array(img)
        try:
            xy = np.shape(imcg) #added
        except:
            continue
        imgr = imcg.reshape((xy[0]*xy[1],)) #added
        imgr = np.array(imgr)
        im_col.append(imgr.transpose())

    return(im_col)

def import_fimage(filepath):
    """
    This function takes a filepath leading to a file with images
    to be imported. This implementation uses the maplotlib image read, which
    reads generic images better and also converts to greyscale.
    The images in the file are appended to
    a column vector, which is returned by the function. Each column is a
    reshaped image.
    """

    im_path = os.path.join(str(filepath))
    im_files = glob.glob(im_path)
    im_col = []

    for image in im_files:
        imc = plt.imread(image,0) # changed from cv2.imread
        imcg = np.array(imc)
        try:
            xy = np.shape(imcg) #added
        except:
            continue
        imgr = imcg.reshape((xy[0]*xy[1],)) #added
        imgr = np.array(imgr)
        im_col.append(imgr.transpose())

    return(im_col)
```

```
def read_files(filepath,forc):

    """
    This function is a wrapper for importing the images using import_image
    functions from each subfolder.  The image files from each folder are
    concatenated into one large array of all the image files in the directory.
    This image matrix array is returned by the function. It also converts
    the output to float64 for mathematical manipulation.
    """

    dir_all = []
    dir_sub = []

    path = os.path.join(str(filepath))
    for dir_name in os.listdir(path):
        filad = str(dir_name)
        sub_path = str(filepath)+filad+'/*.*'

        if forc == 'cropped':
            dir_sub = import_image(str(sub_path))

        if forc == 'full':
            dir_sub = import_fimage(str(sub_path))


        dir_sub = np.array(dir_sub)

        dir_all.append(dir_sub.transpose())

    return(np.array(dir_all))

def flatten_npar(np_array):
    """
    This function is necessary to combine an array of numpy arrays into
    one large numpy array

    """

    itr = len(np_array)
    start = np_array[0]

    for i in range(1,itr):
        start = np.hstack((start,np_array[i]))

    return(np.array(start))


def plot_svd(s,title):

    xrang = range(1,len(s)+1)
    fig = plt.figure()
    plt.subplot(2,1,1)
    plt.plot(xrang,s,'ko')
    plt.ylabel('Sigma Value')
    plt.title(str(title))
    plt.grid()

    plt.subplot(2,1,2)
    plt.semilogy(xrang,s,'ko')
    plt.xlabel('Component')
    plt.ylabel('log10 Sigma Value')
    plt.grid()
    plt.savefig(str(title)+'.png')

    return(fig)
```

```
def energy_snorm(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])

    fnorm = np.linalg.norm(s_mat)

    norm_2 = s_mat[0]/fnorm
    norm_c = []
    norm_i = []

    for i in range(itr):

        norm_c.append(np.linalg.norm(s_mat[0:i+1])/fnorm)
        norm_i.append(np.linalg.norm(s_mat[i])/fnorm)
    return(norm_2, norm_c, norm_i)


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 sub_mean(nparray):
    """
    This function subtracts the mean from each column in an np array and returns the
    array that has the mean of each column subtracted

    """

    ncol = np.shape(nparray)[1]
    newarr = np.zeros(np.shape(nparray))

    for i in range(ncol):

        newarr[:,i] = nparray[:,i]-np.average(nparray[:,i])

    return(newarr)


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

```
    return(u, s, v)

def eig_faces(u_mat, nmode, dim):
    """
    This function takes in the principle component modes of an SVD decomp
    and combines nmodes into one output vector for plotting.  Each column in U
    represents the moddes for a principle component and formats the matrix
    back into the photo dimensions

    """
    n = int(nmode)
    nparray = np.zeros(np.size(u_mat[:,0]))
    for i in range(n):
        nparray = nparray + u_mat[:,i]

    nparray = np.reshape(nparray,dim)
    return(nparray)



#%% Part 1 - Yale Faces: Perform SVD analysis on all cropped faces and
    #full faces

#Import all images into column vectors for cropped and full images
crpd_pth = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/croppedyale/'
im_crpd = read_files(crpd_pth,'cropped')
crpd_dim = (192,168)

crpd_ttl = flatten_npar(im_crpd)
crpd_avg_ttl = sub_mean(crpd_ttl)

full_pth = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/yalefaces/yalefaces/'
im_full = read_files(full_pth,'full')
im_full = np.array(im_full)
full_dim = (243,320)
#
full_ttl = flatten_npar(im_full)
full_avg_ttl = sub_mean(full_ttl)

#%% Part 1 Continued - Performing SVD analysis on cropped image data

uca, sca, vca = svd_images(crpd_avg_ttl)

fig1 = plot_svd(sca,'Cropped Images - SVD on Covarianve Matrix')

eng1a0, eng1ac, eng1ai = energy_s(sca)
a12, ac, ai = energy_snorm(sca)

uc, sc, vc = svd_images(crpd_ttl)

fig1 = plot_svd(sc,'Cropped Images - SVD on Raw Images')

eng10, eng1c, eng1i = energy_s(sc)
n12, nc, ni = energy_snorm(sc)

#%%
# Plotting the energies of the cropped pictures
fig3 = plt.figure(3)
plt.subplot(2,1,1)
plt.plot(range(1,np.shape(sc)[0]+1),eng1i,'kx', label = 'raw image data')
plt.plot(range(1,np.shape(sca)[0]+1),eng1ai,'ro', label = 'covar image data')
plt.ylabel('Inidividual Component Energy')
plt.title('Cropped Image Component Energies')
plt.legend()

plt.subplot(2,1,2)
plt.plot(range(1,np.shape(sc)[0]+1),eng1c,'kx', label = 'raw image data')
```

```
plt.plot(range(1,np.shape(sca)[0]+1),eng1ac,'ro', label = 'covar image data')
plt.ylabel('Cumulative Energy')
plt.xlabel('Component Mode')
plt.savefig('cropped_snorm.png')
plt.legend()

#%%
#plot the first eigenfaces

eface_nm = np.reshape(uc[:,0],crpd_dim)
eface_av = np.reshape(uca[:,0],crpd_dim)

fig5 = plt.figure(5)
plt.subplot(1,2,1)
plt.pcolor(-1*np.flipud(eface_nm), cmap='gray')
plt.title('Cropped Image 1st Component U[:,0]')

plt.subplot(1,2,2)
plt.pcolor(np.flipud(eface_av), cmap='gray')
plt.title('Cov Cropped Image 1st Mode U[:,0]')


fig6 = plt.figure(6)
nfaces = [5, 10, 100]
i = 1
for n in nfaces:
   faces = eig_faces(uc,n,crpd_dim)
   plt.subplot(1,len(nfaces),i)
   plt.pcolor(-1*np.flipud(faces),cmap='gray')
   plt.title(str(n)+' Modes')
   i += 1
plt.savefig('Cropped_MM_Eig.png')

#%% Part 1 Continued - Performing SVD analysis on cropped image data

ufa, sfa, vfa = svd_images(full_avg_ttl)

fig1 = plot_svd(sfa,'Full Images - SVD on Covariance Matrix')

eng1a0, eng1ac, eng1ai = energy_s(sfa)
fa12, fac, fai = energy_snorm(sfa)

uf, sf, vf = svd_images(full_ttl)

fig1 = plot_svd(sf,'Full Images - SVD on Raw Images')

engf0, engfc, engfi = energy_s(sf)
f12, fc, fi = energy_snorm(sf)

#%%

# Plotting the energies of the full pictures
fig7 = plt.figure(3)
plt.subplot(2,1,1)
plt.plot(range(1,np.shape(sf)[0]+1),engfi,'kx', label = 'raw image data')
plt.plot(range(1,np.shape(sfa)[0]+1),eng1ai,'ro', label = 'covar image data')
plt.ylabel('Inidividual Component Energy')
plt.title('Full Image Component Energies')
plt.legend()

plt.subplot(2,1,2)
plt.plot(range(1,np.shape(sf)[0]+1),engfc,'kx', label = 'raw image data')
plt.plot(range(1,np.shape(sfa)[0]+1),eng1ac,'ro', label = 'covar image data')
plt.ylabel('Cumulative Energy')
plt.xlabel('Component Mode')
plt.savefig('cropped_snorm.png')
plt.legend()

#%%
```

```
eface_fm = np.reshape(uf[:,0],full_dim)
eface_af = np.reshape(ufa[:,0],full_dim)

fig8 = plt.figure(8)
plt.subplot(1,2,1)
plt.pcolor(-1*np.flipud(eface_fm), cmap='gray')
plt.title('First Component Mode without Avg Subtracted')

plt.subplot(1,2,2)
plt.pcolor(np.flipud(eface_af), cmap='gray')
plt.title('First Component Mode with Avg Subtracted')


fig6 = plt.figure(6)
nfaces = [1, 10, 100]
i = 1
for n in nfaces:
   faces = eig_faces(uf,n,full_dim)
   plt.subplot(1,len(nfaces),i)
   plt.pcolor(-1*np.flipud(faces),cmap='gray')
   plt.title(str(n)+' Principle Components')
   i += 1
plt.savefig('full_faces_svd')
```

Part 2 Code:

```
"""
Created on Sun Mar  1 13:10:30 2020

@author: wscherer13
"""

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 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

    """

    frq, times, spect = signal.spectrogram(song,fs)
#    frq, times, spect = signal.stft(song,fs,nperseg=2048)
    s_vec = np.reshape(np.real(spect),(np.shape(spect)[0]*np.shape(spect)[1],))
    kp = 0

    s_vec = s_vec-np.average(s_vec)

    return(s_vec, kp)

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,5)

            fft_sig, kp = spectrogram(wav,fs)
            song_mat.append(fft_sig)

        i += 1

    song_mat = np.array(song_mat)

    return(song_mat, kp, 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(decision_function_shape = 'ovr')
    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)


    return([knn_train,knn_test,lda_train,lda_test,lsvc_train,lsvc_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  = []

    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])

        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)]

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

    cols = ['KNN Train', 'KNN Test', 'LDA Train', 'LDA Test', 'SVC Train',
        'SVC 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 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()
    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]))
    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]))
    plt.xlabel('Component Mode')
    plt.ylabel('log10 Sigma Value')
    plt.savefig(str(title)+'.png')
```

```
  return(fig)


#%%
# Load in training data sets

# Import police clips
pol_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/police'
police_mat, kp_p, fs_p = build_svec(pol_path,50)

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

pol_sv = pd.DataFrame(pol_sv); pol_v = pd.DataFrame(pol_v)

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


#Import lord of the rings clips
lotr_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/lotr'
lotr_mat, kp_l, fs_l = build_svec(pol_path,50)

lotr_u, lotr_sv, lotr_v, lotr_s = svd_songs(lotr_mat)

lotr_sv = pd.DataFrame(lotr_sv); lotr_v = pd.DataFrame(lotr_v)

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

#Import Otis clips
otis_path = '/Users/wscherer13/Documents/MATLAB/AMATH582/HW4/otis'
otis_mat, kp_o, fs_o = build_svec(pol_path,50)
#
otis_u, otis_sv, otis_v, otis_s = svd_songs(otis_mat)

otis_sv = pd.DataFrame(otis_sv); otis_v = pd.DataFrame(otis_v)
otis_sv['band']='Otis'; otis_v['band']='Otis'

# create labeled data sets



#%%
#Case 1 Build Model
case1_dat = build_train([otis_sv,lotr_sv,pol_sv])

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(2)
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('Inidividual 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.png')
plt.legend()
#%% train models case 1
case1_5 = avg_models(case1_dat,5,0.10,10)

case1_100 = avg_models(case1_dat,100,0.10,10)

case1_250 = avg_models(case1_dat,250,0.10,10)

case1_500 = avg_models(case1_dat,500,0.10,10)

#%%
#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,50)

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

led_sv = pd.DataFrame(led_sv); led_v = pd.DataFrame(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,50)

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

jimi_sv = pd.DataFrame(jimi_sv); jimi_v = pd.DataFrame(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,50)

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

stones_sv = pd.DataFrame(stones_sv); stones_v = pd.DataFrame(stones_v)

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

#%%
case2_dat = build_train([led_sv, jimi_sv, stones_sv])
#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(2)
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('Inidividual 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.png')
plt.legend()
#%%
#Case 2 Train Models

case2_5 = avg_models(case2_dat,5,0.10,10)

case2_100 = avg_models(case2_dat,100,0.10,10)

case2_250 = avg_models(case2_dat,250,0.10,10)

case2_500 = avg_models(case2_dat,500,0.10,10)

#%%
#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,50)

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

rock_sv = pd.DataFrame(rock_sv); rock_v = pd.DataFrame(np.transpose(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,50)

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

jazz_sv = pd.DataFrame(jazz_sv); jazz_v = pd.DataFrame(np.transpose(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,50)

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

soul_sv = pd.DataFrame(soul_sv); soul_v = pd.DataFrame(np.transpose(soul_v))

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

#%%
#Case 3 Build Models

case3_dat = build_train([rock_sv, jazz_sv, soul_sv])

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(2)
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('Inidividual 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.png')
plt.legend()

case3_5 = avg_models(case3_dat,5,0.10,10)

case3_100 = avg_models(case3_dat,100,0.10,10)

case3_250 = avg_models(case3_dat,250,0.25,10)

case3_500 = avg_models(case3_dat,500,0.25,10)
```