AMATH 582 Homework 4

Daniel Burnham (https://github.com/burnhamdr)

March 6, 2020

Abstract

The work presented here is motivated by material covered in AMATH 582 Computational Methods For Data Analysis regarding the applications of Singular Value Decomposition (SVD) and classification algorithms. This report will discuss these topics in two sections. The first secion will report on application of SVD to a data set of human faces. SVD is a linear algebra matrix factorization method that represents how the matrix of interest stretches/compresses and rotates a given set of vectors. The SVD is a powerful tool because the constitutive components of its factorization allow for the subsequent projection of a matrix onto low-dimensional representations. In the problem addressed here this is used to determine the rank (dimensionality) of a data set of human faces. The results demonstrate that faces can be sufficiently represented with low rank approximations when the face images are cropped in a standardized manner. In applying SVD to uncropped faces the dimensionality reduction is less effective in creating a low rank approximation of individual faces. The second section will report on the implementation of various classification algorithms for music artist and genre discrimination tasks. Both time domain and frequency spectrum music data will be investigated as data substrates for classification algorithms in the contexts of artist discrimination and genre detection. Linear Discriminant Analysis, Logistic Regression, and K-nearest Neighbors classifiers were utilized for these tasks. The work in presented in the two sections of this report describes fundamental strategies to address modern data science problems.

1 Introduction and Overview

The topics explored here are discussed in two sections. The first section involves the analysis of both cropped and uncropped face image data using SVD. The second section involves music artist and genre classification algorithms.

1.1 SVD Face Data Analysis

Two data sets were analyzed with SVD: cropped face images, and uncropped face images. SVD analysis was performed on both data sets of these data sets to investigate how many modes are necessary for good image reconstruction (i.e. determine the rank of the face space). Differences in the singular value spectrum between the two data sets were subsequently identified and are discussed further in the Computational Results section of this report.

1.2 Music Classification

In keeping with the theme of data feature extraction, characteristics of music data are leveraged to classify genres and artists in this part of the report. Music styles are easily recognizable to the human ear, which raises the question of how are these distinctions drawn? The objective of the work in this section of the report is to to implement computational algorithms that capably classify music pieces based on a 5 second sample. This objective was pursued in the context of three specific tasks:

1. (test 1) Band Classification: Three different bands were considered: Flume, Snoh Aalegra, and Tame Impala. Flume is a future base electronic music producer, Snoh Aalegra is a R&B vocalist, and Tame Impala is a neo-psychadelic rock/pop artist. By taking 5-second clips from a variety of each of these artist's music, statistical testing algorithms were implemented with the objective being to seek methods capable of accurately identifying "new" 5-second clips of music from the three chosen bands.

2. (test 2) Same Genre Band Classification: The above experiment was repeated, but with three bands from within the same genre. The genre chosen was jazz of the early to mid 1900s. The artists chosen were: Louis Armstrong, Miles Davis, Duke Ellington, and John Coltrane. Choosing these artists of the same genre was in an effort to test the abilities of classification algorithms to separate music into artist specific classes.

3. (test 3) Genre Classification: Expanding upon the results of the first two tests, genre classification was then pursued. 5 second music clips were again compiled from genres of jazz, rock, classical, and electronic. The training sets were composed of various bands and artists within each genre.

Each of these test exercises contributed to illustrating the ability of computational methods to distinguish characteristics of musical style. Understanding these methods perhaps can shed light on possible methods used by the brain for similar tasks.

2 Theoretical Background

2.1 Singular Value Decomposition (SVD)

SVD is a linear algebra method for factorization of a matrix into a number of constitutive components. It is rooted in the observation that during matrix vector multiplication of a vector $\mathbf{x}$ by a matrix $\mathbf{A}$ the resulting vector $\mathbf{y}$ has a new magnitude and direction. This transformation of the vector $\mathbf{x}$ by a matrix $\mathbf{A}$ implies that perhaps the action of matrix $\mathbf{A}$ can be replicated through component matrices that perform the same magnitude and direction manipulations. Furthermore, it would be beneficial if these components possessed properties that made them easy to work with, such as orthogonality and diagonality. The SVD factorization of the matrix $\mathbf{A}$ achieves these goals by expanding this observation of vector transformation under multiplication by a matrix to $\mathbb{R}^m$. SVD does this by building from the observation that the image of a unit sphere under any m×n matrix is a hyperellipse. Following from this, one can represent this transformation as $\mathbf{A}\mathbf{v}_j = \sigma_j \mathbf{u}_j$ for $1 \leq j \leq n$. Where $\mathbf{v}_j$ are the vectors of the unit sphere transformed by $\mathbf{A}$, and $\sigma_j \mathbf{u}_j$ are the resulting transformations representing the the semiaxes of the hyperellipse. Rearranging this equation allows for the SVD factorization of $\mathbf{A}$ to be written as follows (Kutz 2013):

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* \tag{1}$$

In this form, $\mathbf{U}$ is unitary and contains the vectors $\mathbf{u}_j$ indicating the directions of the transformed hyperellipse semiaxes, $\mathbf{\Sigma}$ is diagonal and contains the scaling values corresponding to these semiaxes, and $\mathbf{V}^*$ is the Hermitian transpose of $\mathbf{V}$ which contains the orthonormal basis of the vectors that are transformed under $\mathbf{A}$.

It is worth noting that it can be proved that every matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$ has a singular value decomposition and the singular values are uniquely determined (Kutz 2013). Additionally, if the matrix $\mathbf{A}$ is square, the singular vectors $\mathbf{u}_j$ and $\mathbf{v}_j$ are also uniquely determined up to complex signs (Kutz 2013). This is significant because it allows for every $\mathbf{A} \in \mathbb{C}^{m \times n}$ to be factorized by SVD and subsequently represented with lower rank matrix approximations. This will be useful in the context of the face image data set explored here as it represents a way of reducing the dimensionality of the face feature space.

2.2 Frequency Analysis

In the analysis of the music file data both the frequency spectrum of the audio signals are computed prior to implementation of classification methods. The Discrete Fourier Transform (DFT) can be utilized for this purpose when working with sets of data interpreted as discrete points. The foundation of DFT is the observation that a given function can be represented as a sum of sin and cos functions in the set $\{\sin(kx), \cos(kx)\}_{k=0}^{\infty}$. This representation is known as a Fourier Series:

$$f(x) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kx) + b_k \sin(kx). \tag{2}$$

The coefficients a_k and b_k represent how similar the represented function f is to waves with frequencies proportional to k. This transform thus maps the function, or the data, to frequency components. When applying the DFT to arrays of values, like sound wave amplitude data across time, we obtain a discrete set of frequency components. To apply DFT for the computational purposes of this work, the Fast Fourier Transform (FFT) is used. Extracting the frequency content for analysis instead of working with the raw sound wave data affords us the opportunity to filter out frequency components following SVD dimensionality reduction if the computational load of classifying the data set is overly burdensome.

2.3 Classification Methods

2.3.1 Linear Discriminant Analysis

The goal of Linear Discriminant Analysis (LDA) is to find the best projection of the data that maximize class separation (see Figure 1). This is achieved by mathematically constructing the optimal projection basis which separates the data (Kutz 2013). In the case of classification of two distinct classes with LDA, a projection w is constructed as:

$$\mathbf{w} = \arg\max_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} \tag{3}$$

Where $\mathbf{S}_B$ and $\mathbf{S}_W$ are referred to as the scatter matrices for inter-class ($\mathbf{S}_B$) and intra-class ($\mathbf{S}_W$) data are mathematically derived as:

$$\mathbf{S}_B = (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T \quad \mathbf{S}_W = \sum_{j=1}^{2} \sum_{\mathbf{x}} (\mathbf{x} - \mu_j)(\mathbf{x} - \mu_j)^T \tag{4}$$

The μ terms are arrays of the average value for each class. The scatter matrices capture the variance within each of the data classes as well as the variance of the difference in the means between classes. From these scatter matrices Equation 3 can be solved by way of solving the eigenvalue problem

$$\mathbf{S}_B = \lambda \mathbf{S}_W \mathbf{w} \tag{5}$$

The eigen vector of the maximum eigenvalue calculated from Equation 5 is the projection basis for LDA to maximally separate the two classes.

2.3.2 Logistic Regression

Logistic Regression is a widely used method for classification of a dichotomized output variable Y, and works by attempting to model the conditional probability of Y taking on a particular value (i.e. $Pr(Y = 1|X = x)$) as a function of the inputs. A problem in using simple linear regression for this model is that the modelled probability must be between 0 and 1, and a linear function is unbounded. Also it is common to confront systems where the relationship between the conditional probability and the inputs is not linear. The idea behind logistic regression is to instead let the log of the probability be a linear function of of the inputs. By performing a logistic transformation of the log probability the model becomes bounded between 0 and 1.

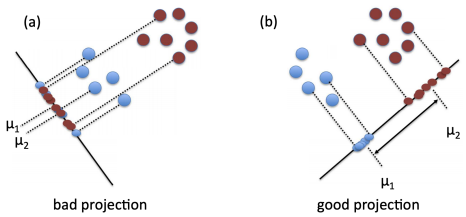

Figure 1: Two-class LDA task where an appropriate projection basis (b) is shown versus a poor performing projection bases (a). (Kutz 2013)

The model is thus given by Equation 6 where x indicates the predictors and β the model parameters (Shalizi 2012).

$$log\left(\frac{p(x))}{1-p(x))}\right) = \beta_0 + x\beta \tag{6}$$

The classification in a two class case will occur for one class when $p \geq 0.5$ and the other class when $p < 0.5$. To predict classes instead of probabilities, maximum likelihood estimation is used to estimate the parameters of a probability distribution by maximizing a likelihood function. In this case the log likelihood function will be maximized (Shalizi 2012).

$$\ell(\beta_0, \beta) = \sum_{n}^{i=1} y_i log(p(x_i)) + (1-y_i) log(1-p(x_i)) \tag{7}$$

In Equation 7 the features are captured by the x_i values and each class is represented by a y_i. At this stage a "penalty" can be added to the classification in an effort to promote sparsity in the model. In the work presented here an L1 penalty is used as defined in Equation 8.

$$\ell^*(\beta_0, \beta) = \ell\beta_0, \beta) - \lambda \sum_{n}^{i=1} \|\beta_i\| \tag{8}$$

Following this penalty, the derivative of the likelihood function is solved numerically to find the maximum likelihood predictions for each class. Overall, logistic regression draws a linear decision boundary between classes and is a unique linear classifier in that the probability of predicting either class depends the proximity to the boundary (Shalizi 2012).

2.3.3 K-nearest Neighbors

The K-nearest Neighbors classification method aims to classify data points by determining the proximity of the data point in the data space to a predefined number of training samples. The proximity is then used to classify the data point by a plurality vote. The number of clusters to create in the training set is set by the implementer. The algorithm is simple in nature compared to others profiled in this work, but has achieved success in many problem contexts especially in situations that have irregular decision boundaries.

2.3.4 Classifier Performance Metrics

In assessing the performance of the classification algorithms confusion matrices and Receiver Operating Characteristic Curves will be used.

A confusion matrix for a classification problems simply indicates the frequency of correct and erroneous class predictions in the test data. The counts indicate how often a class is predicted as itself or if it is frequently misclassified as another class. This is important information in assessing the performance and thus suitability of the classification model.

A Receiver Operating Characteristics (ROC) curve is a plot of False Positive Rate (FPR) versus True Positive Rate (TPR) for the classification of a given data class. These rates can be calculated using a macro-average strategy where the metric will be determined independently for each class before taking the average, or with a micro-average which instead collects all the contributions regardless of class before calculating the average FPR and TPR. In the case where a class imbalance may exist, the micro-average is the preferred method for calculating the ROC curve as it will not all the contributions of one class to disproportionately skew the curve trajectory. The ROC curve is a probability curve that indicates the relationship between the distributions of true positive and true of negative classifications. The area under the curve (AUC) is thus a measure of the degree of separability between a positive and negative classification.

3 Algorithm Implementation and Development

3.1 SVD Face Data Analysis

The main steps of the algorithm are as follows:

1. Input the face image files by iterating through file directory, prepare for analysis
2. Flatten each image into single array, concatenate all images together
3. Compute SVD of flattened image data set, visualize top modes, estimate dimensionality
4. Project onto the left singular vectors, compare reconstructed images to originals

3.2 Music Genre Classification

The main steps of the algorithm are as follows:

1. Input .wav music files, slice into 5 second sections, label for classification
2. Split music samples into training and testing data sets
3. Implement LDA, Logistic Regression, and K-Nearest Neighbors classification methods
4. Investigate classifier performance using confusion matrix and ROC curve

4 Computational Results

4.1 SVD Face Data Analysis

The computational results in this section are a result of performing an SVD analysis of the cropped (Figure 2) and uncropped (Figure 3) Yale face images. The interpretation of the the left singular vectors of the U matrix are the principal reconstruction axes of the image data and the Σ values are the scalings of these axes. The right singular vectors of V apply rotation back into the original axes for the rank reduced reconstructions. The rank of the face space for the cropped Yale faces appears to be rank 4 based on Figure 2a. The rank of the uncropped face space appears to similarly be approximately 4 based on the top modes shown in Figure 3a.

4.2 Music Genre Classification

4.2.1 (test 1) Band Classification

Three different bands were considered: Flume, Snoh Aalegra, and Tame Impala. The LDA classifier presented in the body of this report performed well on the simple band classification task with accuracy score: 0.925824, Precision: 0.926716, Recall: 0.925824. and F1 score: 0.925916. Figure 4 shows the confusion matrix counts with the most common erroneous classifications occurring between the Flume and Snoh Aalegra classes.

4.2.2 (test 2) Same Genre Band Classification

The genre chosen of early to mid 1900s jazz was chosen for this test case with specific artists: Louis Armstrong, Miles Davis, Duke Ellington, and John Coltrane. The LDA classifier again performed well on the jazz artist classification task with accuracy score: 0.958084, Precision: 0.958332, Recall: 0.958084, and F1 score: 0.958125. Figure 5 shows the confusion matrix counts with the most common erroneous classifications occurring between the John Coltrane and Miles Davis classes.

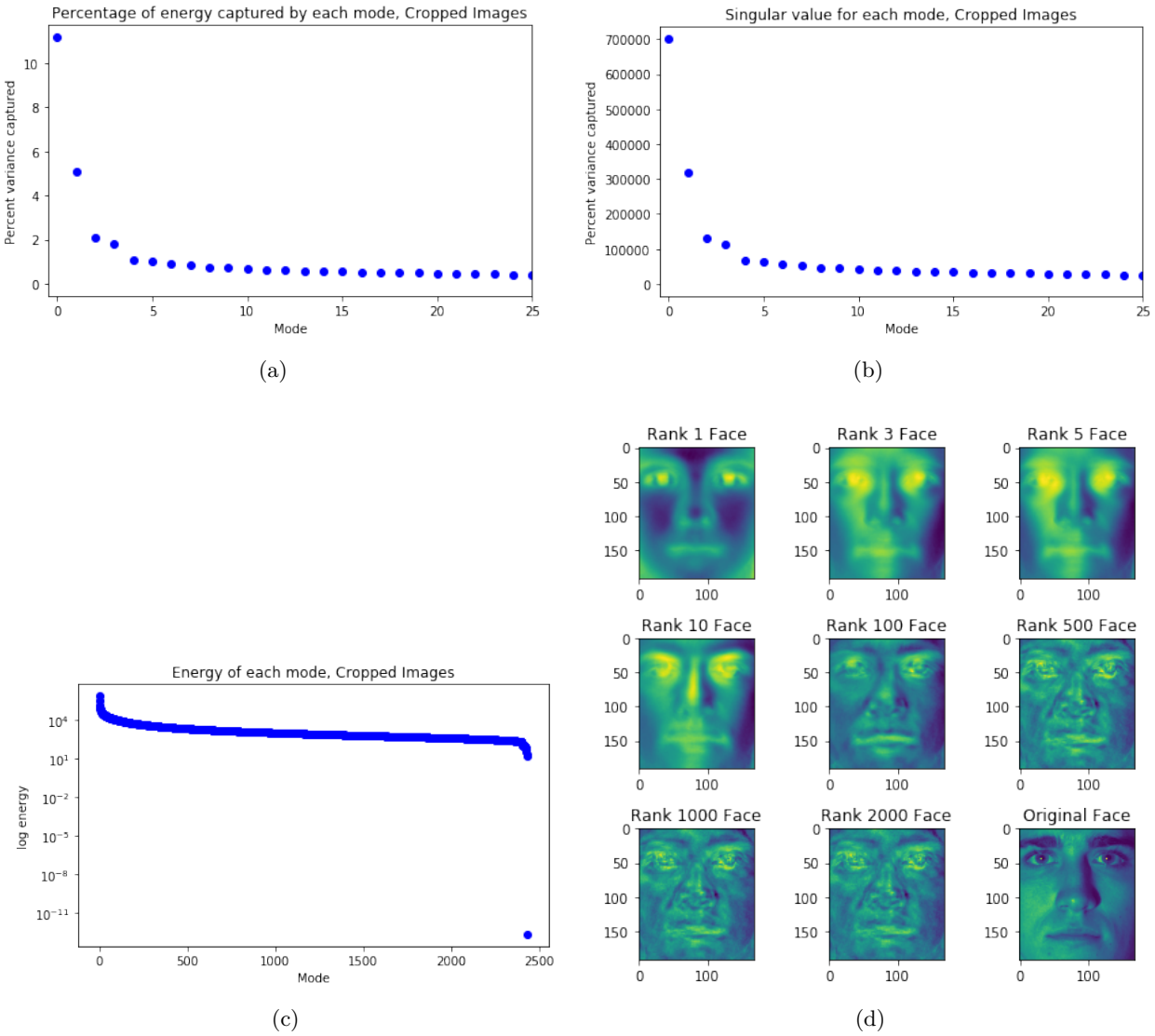

Figure 2: (a) The percentage of energy captured by each SVD mode. (b) Magnitudes of the singular values (c) Log energy of each SVD mode (d) Reconstructed faces with increasing ranks

4.2.3 (test 3) Genre Classification

Music clips were taken from the genres of jazz, rock, classical, and electronic. The LDA classifier performed adequately on the genre discrimination task with accuracy score: 0.899510, Precision: 0.907349, Recall: 0.899510, and F1 score: 0.899538. Figure 6 shows the confusion matrix counts with the most common erroneous classifications occurring between the jazz and classical music classes.

5 Summary and Conclusions

5.1 SVD Face Data Analysis

Two significant results emerged from the face image SVD processing task. First, face image data can be adequately represented with rank 4 approximations. Second, the cropping of the face images is an important preprocessing/collection step for ensuring successful low rank image reconstructions. The misalignment of faces in each uncropped image contributed to significant differences in the low rank reconstructions (Figure 2d

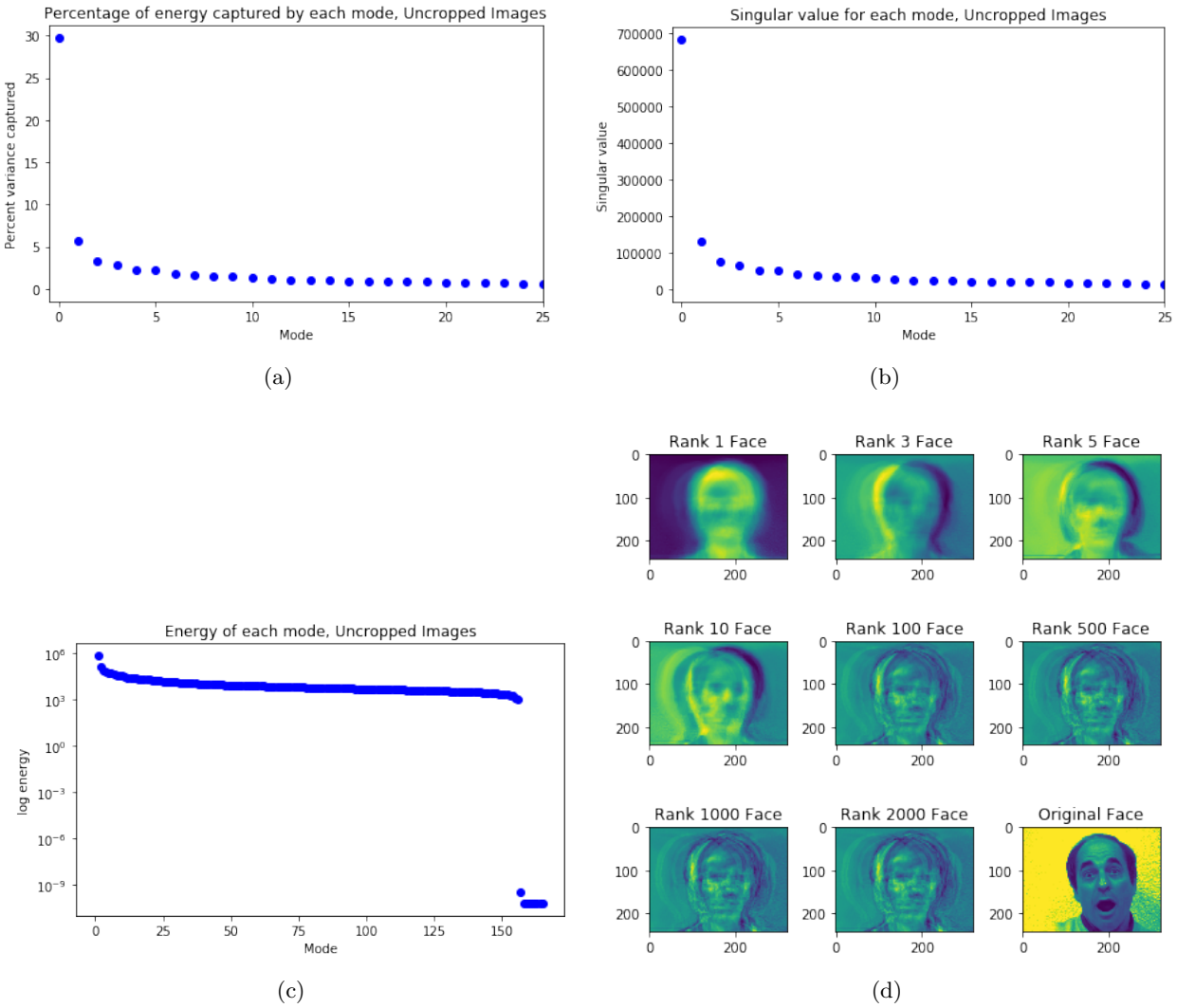

Figure 3: (a) The percentage of energy captured by each SVD mode. (b) Magnitudes of the singular values (c) Log energy of each SVD mode (d) Reconstructed faces with increasing ranks

vs Figure 3d) generated through SVD. This is most assuredly due to increased noise in the data attributable to face orientation/position variability in each image frame.

5.2 Music Classification

Overall a music data sampling and processing pipeline was successfully constructed capable of taking raw sound files, chopping songs into 5 second clips and concatenating this data into a complete data set for classification. Three difference classification algorithms were explored for each test case: LDA, Logistic Regression, and K-Nearest Neighbors classifiers. All were largely successful with the most notable exception being the K-Nearest Neighbors classifier out performing LDA and Logistic Regression on test 3 for genre classification (Figure 12). This is most likely due to the K-Nearest Neighbors classifier being more adaptable to non-linear decision boundaries.

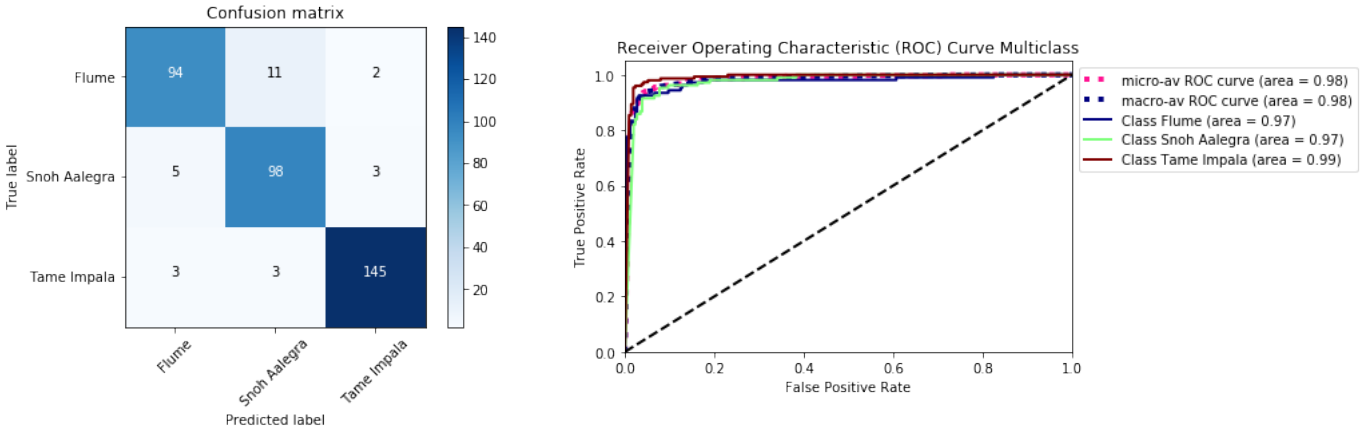

Figure 4: Test 1: (Left) Confusion matrix indicating predicted class counts for LDA classifier. (Right) Receiver Operating Characteristic (ROC) curve for LDA classifier.

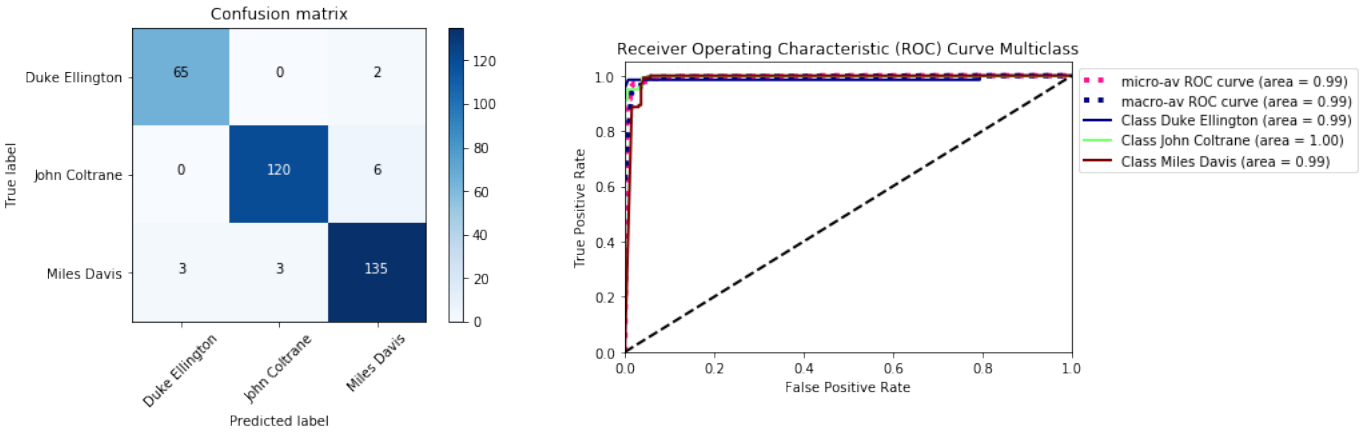

Figure 5: Test 2: (Left) Confusion matrix indicating predicted class counts for LDA classifier. (Right) Receiver Operating Characteristic (ROC) curve for LDA classifier.

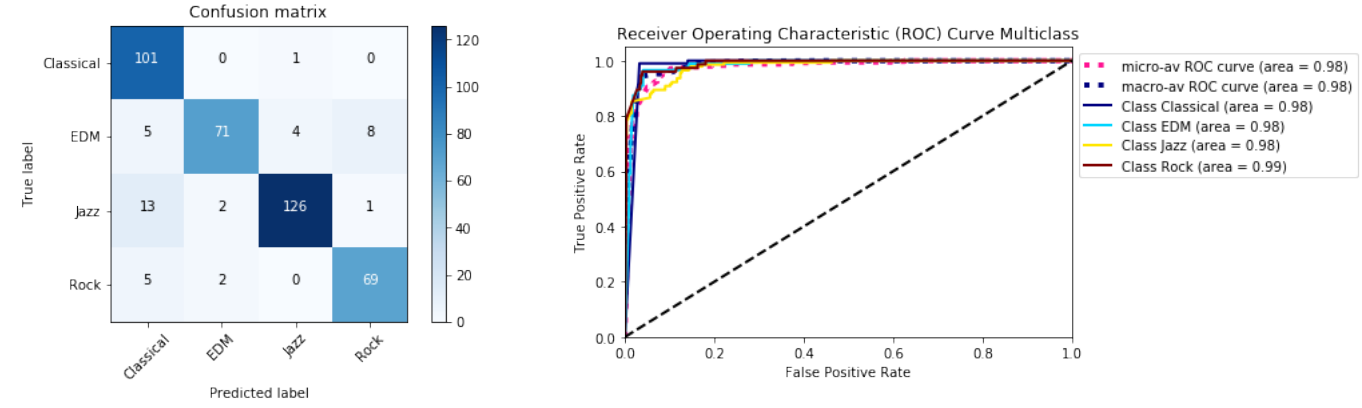

Figure 6: Test 3: (Left) Confusion matrix indicating predicted class counts for LDA classifier. (Right) Receiver Operating Characteristic (ROC) curve for LDA classifier.

References

Kutz, Jose Nathan (2013). *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press.

Shalizi, Cosma Rohilla (2012). "Advanced Data Analysis from an Elementary Point of View". In:

Appendix A Python Functions

- `numpy.linalg.svd(a, full_matrices=True, compute_uv=True, hermitian=False)`
  Singular Value Decomposition.

- `numpy.fft.fft2(a, s=None, axes=(-2, -1), norm=None)`
  Compute the 2-dimensional discrete Fourier Transform. This function computes the n-dimensional discrete Fourier Transform over any axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

- `numpy.amax(a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>)`
  Return the maximum of an array or maximum along an axis.

- `skimage.filters.threshold_otsu(image, nbins=256)`
  Return threshold value based on Otsu's method.

- `sklearn.discriminant_analysis.LinearDiscriminantAnalysis(solver='svd', shrinkage=None, priors=None, n_components=None, store_covariance=False, tol=0.0001)`
  Linear Discriminant Analysis. A classifier with a linear decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule. The model fits a Gaussian density to each class, assuming that all classes share the same covariance matrix. The fitted model can also be used to reduce the dimensionality of the input by projecting it to the most discriminative directions.

- `sklearn.linear_model.LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=`
  `intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None)`
  Logistic Regression (aka logit, MaxEnt) classifier.

- `sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs)`
  Classifier implementing the k-nearest neighbors vote.

- `sklearn.metrics.roc_auc_score(y_true, y_score, average='macro', sample_weight=None, max_fpr=None, multi_class='raise', labels=None)`
  Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores. This implementation can be used with binary, multiclass and multilabel classification, but some restrictions apply (see Parameters).

Appendix B Supplemental Figures

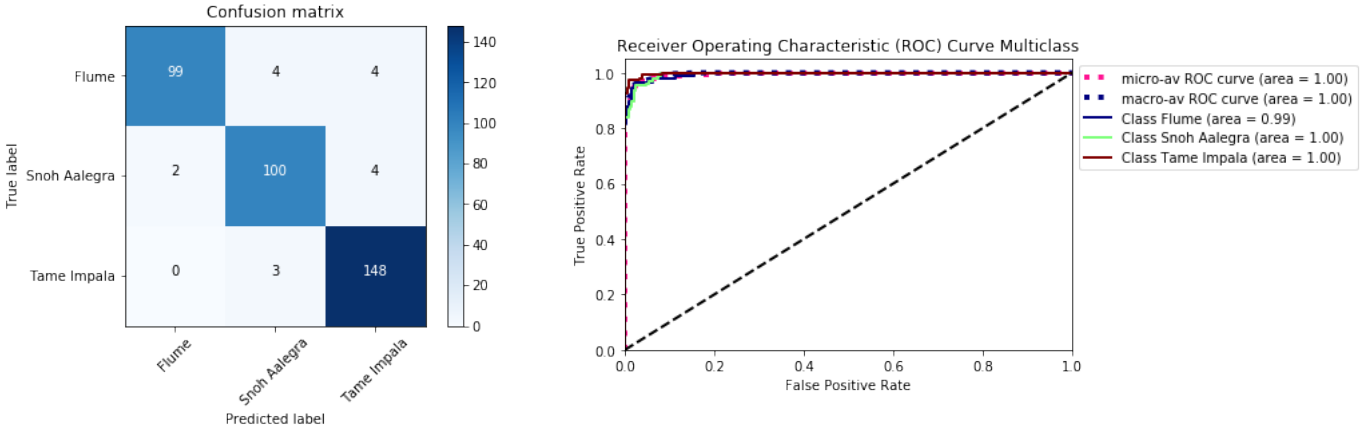

Figure 7: Test 1: (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for Logistic Regression classifier.

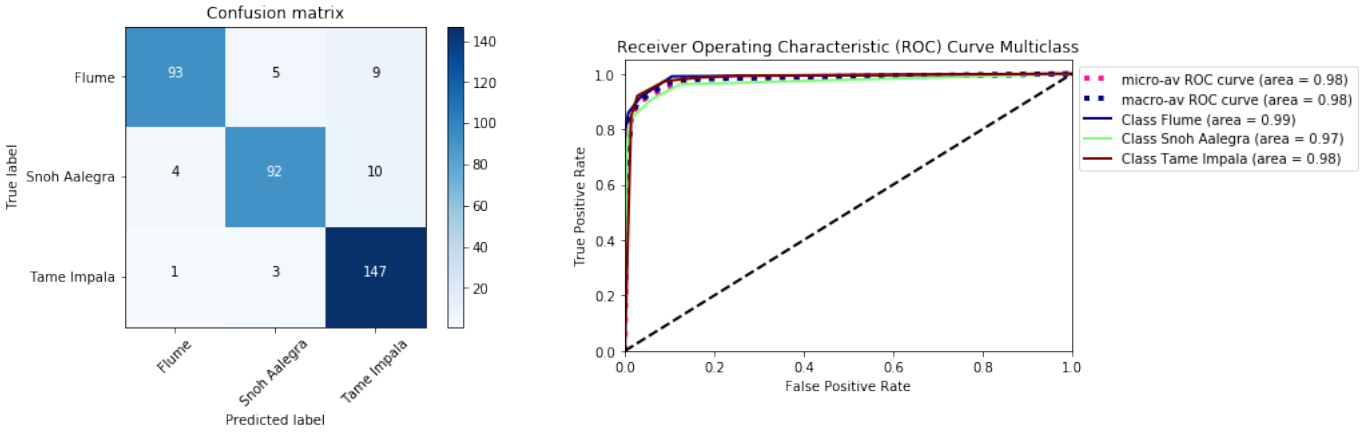

Figure 8: Test 1: (Left) Confusion matrix indicating predicted class counts for K-Nearest Neighbors classifier. (Right) Receiver Operating Characteristic (ROC) curve for K-Nearest Neighbors classifier.

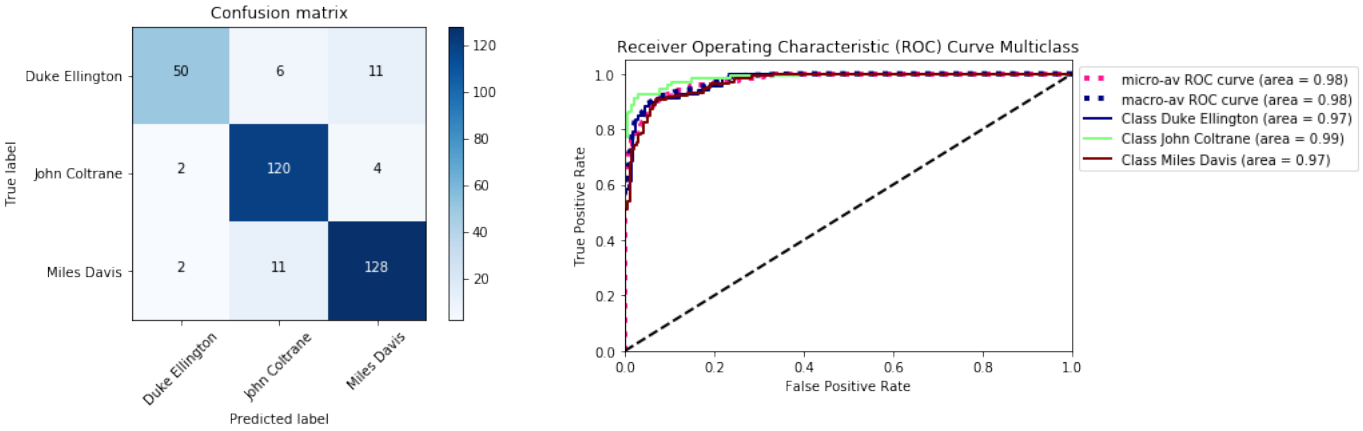

Figure 9: Test 2: (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for Logistic Regression classifier.

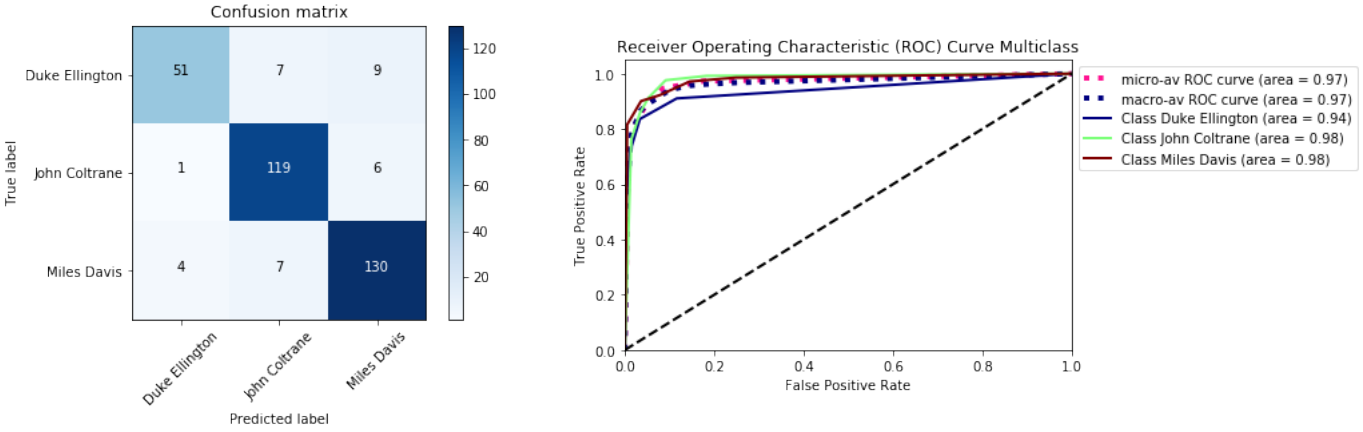

Figure 10: Test 2: (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for K-Nearest Neighbors classifier.

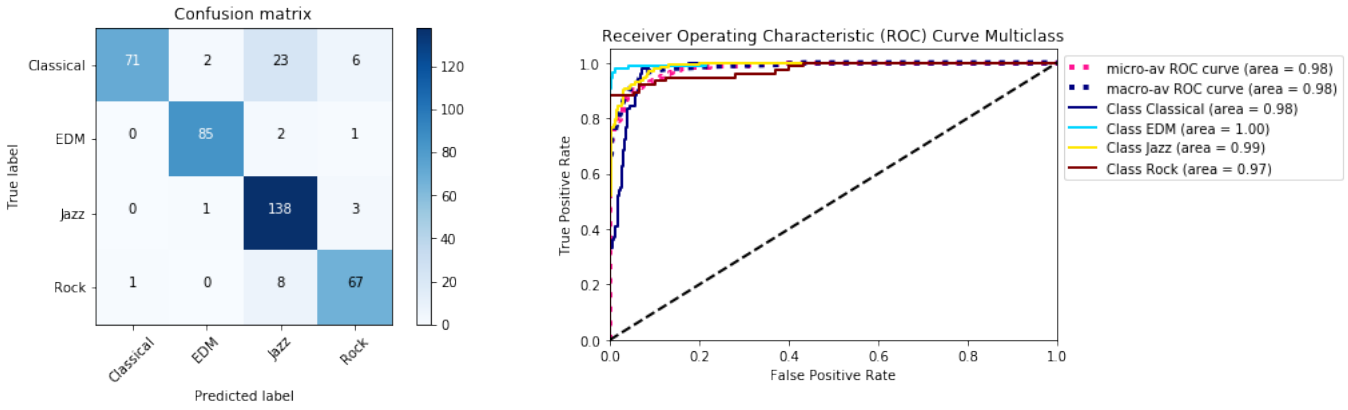

Figure 11: Test 3: (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for Logistic Regression classifier.

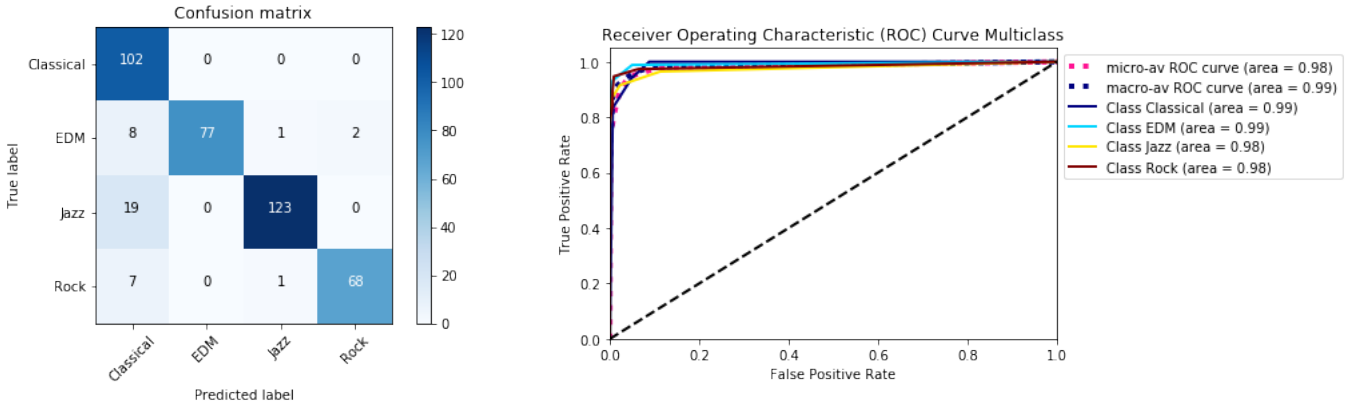

Figure 12: Test 3: (Left) Confusion matrix indicating predicted class counts for logistic regression classifier. (Right) Receiver Operating Characteristic (ROC) curve for K-Nearest Neighbors classifier.

Appendix C Python Code

```
#!/usr/bin/env python
# coding: utf-8

# In[93]:


#Import statements
from scipy.io import loadmat
from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2
from copy import deepcopy
import math
from matplotlib import gridspec
from PIL import Image
import pandas as pd
import random
import itertools
from scipy import interp
import matplotlib.cm as cm
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import *
from sklearn.preprocessing import label_binarize
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.multiclass import OneVsRestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis


# In[94]:


'''Adapted from stack overflow response from Jeru Luke,
https://stackoverflow.com/questions/51285593/converting-an-image-to-grayscale-using-numpy

Takes in color image and converts to gray scale'''
def grayConversion(image):
    grayValue = 0.07 * image[:,:,2] + 0.72 * image[:,:,1] + 0.21 * image[:,:,0]
    gray_img = grayValue.astype(np.uint8)
    return gray_img

def spectroPlot(Sgt_spec, tslide, ks, xlim, ylim, title):
    '''spectroPlot generates a spectrogram plot of Gabor filtered two dimensional signal data

    INPUT:
    Sgt_spec: Sgt_spec: numpy.ndarray of fourier transformed and shifted data in each time bin
    tslide: numpy.ndarray of the time bins used in analysis
    ks: numpy.ndarray of shifted wave numbers
    xlim: tuple indicating x axis minimum and maximum (min, max)
    ylim: tuple indicating y axis minimum and maximum (min, max)'''

    f, ax = plt.subplots()
    ax.pcolormesh(tslide, ks, np.transpose(Sgt_spec), cmap = 'hot')
    ax.set_ylabel('Frequency [Hz]')
    ax.set_xlabel('Time [sec]')
    ax.set_title(title)
    ax.set(xlim = xlim, ylim = ylim)
'''Function that plots an ROC curve per class of a multiclass classifier case
by using the One Vs Rest technique.  This function was adapted from code
at: http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html
demonstrating the multiclass implementation of ROC curves for classifier
evaluation.

Inputs:
    X_train: dataframe object containing the data to train on.
    y_train: dataframe object containing the label binarized class outputs of
            the training set.
    X_test: dataframe object containing the data for testing the model
    y_test: dataframe object containing the label binarized class outputs of
            the test set.
    model: classifier object delineating the model to fit
    classes: list containing the classes of the data to be used in plotting

Outputs:
    ROC curve plot containing curves for each class, as well as the micro and
    macro average ROC curves.
'''
def multiclassROC(X_train, y_train, X_test, y_test, model, classes):
    classifier = OneVsRestClassifier(model)
    y_score = classifier.fit(X_train, y_train).predict_proba(X_test)
    y_train = label_binarize(y_train, classes=np.unique(y_train))
    y_test = label_binarize(y_test, classes=np.unique(y_test))
    n_classes = y_train.shape[1]
    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_score.ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])

    # Compute macro-average ROC curve and ROC area

    # First aggregate all false positive rates
    all_fpr = np.unique(np.concatenate([fpr[i] for i in range(n_classes)]))

    # Then interpolate all ROC curves at this points
    mean_tpr = np.zeros_like(all_fpr)
```

```
    for i in range(n_classes):
        mean_tpr += interp(all_fpr, fpr[i], tpr[i])

    # Finally average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = all_fpr
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])

    # Plot all ROC curves
    plt.figure()
    lw=2
    plt.plot(fpr["micro"], tpr["micro"],
             label='micro-av ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["micro"]),
             color='deeppink', linestyle=':', linewidth=4)

    plt.plot(fpr["macro"], tpr["macro"],
             label='macro-av ROC curve (area = {0:0.2f})'
                   ''.format(roc_auc["macro"]),
             color='navy', linestyle=':', linewidth=4)

    start = 0.0
    stop = 1.0
    number_of_lines= n_classes
    cm_subsection = np.linspace(start, stop, number_of_lines)
    colors = [ cm.jet(x) for x in cm_subsection ]

    for i, color in zip(range(n_classes), colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=lw,
                 label='Class {0} (area = {1:0.2f})'
                 ''.format(classes[i], roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--', lw=lw)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve Multiclass')
    plt.legend(bbox_to_anchor=(1.00, 1.00))
    plt.show()

"""
This function prints and plots the confusion matrix.
Normalization can be applied by setting `normalize=True`.
Adapted from code at:
    http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
"""
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    plt.figure()
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    bottom, top = plt.ylim()
    plt.ylim(bottom + 0.5, top - 0.5)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()


# In[3]:
```

```
#cropped image flattening and data matrix concatenation
parent_dir = "/Users/danielburnham/Documents/AMATH Masters/AMATH582/HW4/CroppedYale/"
cropped_faces = []
for subdir, dirs, files in os.walk(parent_dir):
    for d in dirs:
        abs_path = os.path.join(parent_dir, d)
        for subdir, dirs, files in os.walk(parent_dir + d + "/"):
            for file in files:
                im_path = os.path.join(abs_path, file)
                face = cv2.imread(im_path, -1)
                cropped_faces.append(face)
flat_faces = []
for i in range(0, len(cropped_faces)):
    flat_face = cropped_faces[i].flatten()
    flat_faces.append(flat_face)
cropped_data = np.transpose(np.array(flat_faces))


# In[4]:


#Apply SVD to cropped faces data
pix, fac = cropped_data.shape#pix = pixels, fac = faces
#Subtract off the mean for each row of data
average_face = np.mean(cropped_data, axis=1)#compute average column of the cropped data matrix (i.e. the average face)
'''
X = np.empty((pix, fac))

for i in range(0, fac):
    col_av = np.average(cropped_data[:,i])
    col_centered = cropped_data[:,i] - col_av
    X[:,i] = col_centered

'''
X = cropped_data
#Compute SVD
U, Sigma, VT = np.linalg.svd(X, full_matrices=False)


#Plot percentage of variance captured by differnt modes
#Principle component oscillations
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot((Sigma / sum(Sigma)) * 100, 'bo')
ax1.set_title('Percentage of energy captured by each mode, Cropped Images')
ax1.set_ylabel('Percent variance captured')
ax1.set_xlabel('Mode')
ax1.set_xlim(( -0.5, 25))

f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(Sigma, 'bo')
ax1.set_title('Singular value for each mode, Cropped Images')
ax1.set_ylabel('Percent variance captured')
ax1.set_xlabel('Mode')
ax1.set_xlim(( -0.5, 25))

f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.semilogy(np.arange(1, fac + 1), Sigma, 'bo')
ax1.set_title('Energy of each mode, Cropped Images')
ax1.set_ylabel('log energy')
ax1.set_xlabel('Mode')
#ax1.set_xlim(( -0.5, 25))
#ax1.set_ylim((10**-3, 10**1))


# In[5]:


#Project onto the left singular vectors, compare reconstructed images to originals
Sigma_mat = np.diag(Sigma)
ranks = [1, 3, 5, 10, 100, 500, 1000, 2000]
image = 5

cols = 3
rows = int(math.ceil((len(ranks) + 1) / cols))

gs = gridspec.GridSpec(rows, cols)
fig = plt.figure(figsize=(2*rows, 2*cols))
for i in range(0, len(ranks) + 1):
    ax = fig.add_subplot(gs[i])
    #if loop has gone through all desired ranks, lastly plot the original image
```

```
    if i == len(ranks):
        X_r_reshape = X[:, image].reshape(cropped_faces[image].shape)#set the X_r_reshape to be original image for plotting
        ax.set_title('Original Face')
    else:
        rank_reducer = np.ones(fac)
        rank_reducer[ranks[i]:] = 0
        sm = np.dot(Sigma_mat, np.diag(rank_reducer))
        X_r = np.dot(U, np.dot(sm, np.transpose(VT)))
        X_r_reshape = (X_r[:, image]).reshape(cropped_faces[image].shape)
        ax.set_title('Rank ' + str(ranks[i]) + ' Face')

    ax.imshow(X_r_reshape)
fig.tight_layout()


# In[15]:


#uncropped image flattening and data matrix concatenation
parent_dir = "/Users/danielburnham/Documents/AMATH Masters/AMATH582/HW4/yalefaces_uncropped/"
uncropped_faces = []
for subdir, dirs, files in os.walk(parent_dir):
    for d in dirs:
        abs_path = os.path.join(parent_dir, d)
        for subdir, dirs, files in os.walk(parent_dir + d + "/"):
            for file in files:
                im_path = os.path.join(abs_path, file)
                gif = cv2.VideoCapture(im_path)
                ret,frame = gif.read() # ret=True if it finds a frame else False. Since your gif contains only one frame, the next read() will give
                image = Image.fromarray(frame)
                face = grayConversion(np.array(image))
                uncropped_faces.append(face)
flat_faces = []
for i in range(0, len(uncropped_faces)):
    flat_face = uncropped_faces[i].flatten()
    flat_faces.append(flat_face)
uncropped_data = np.transpose(np.array(flat_faces))


# In[16]:


#Apply SVD to cropped faces data
pix, fac = uncropped_data.shape#pix = pixels, fac = faces
#Subtract off the mean for each row of data
average_face = np.mean(uncropped_data, axis=1)#compute average column of the cropped data matrix (i.e. the average face)

Xuc = uncropped_data


#Compute SVD
U, Sigma, VT = np.linalg.svd(Xuc, full_matrices=False)


#Plot percentage of variance captured by differnt modes
#Principle component oscillations
f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot((Sigma / sum(Sigma)) * 100, 'bo')
ax1.set_title('Percentage of energy captured by each mode, Uncropped Images')
ax1.set_ylabel('Percent variance captured')
ax1.set_xlabel('Mode')
ax1.set_xlim(( -0.5, 25))

f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.plot(Sigma, 'bo')
ax1.set_title('Singular value for each mode, Uncropped Images')
ax1.set_ylabel('Singular value')
ax1.set_xlabel('Mode')
ax1.set_xlim(( -0.5, 25))

f, ax1 = plt.subplots(1, 1, figsize=(7, 4))
ax1.semilogy(np.arange(1, fac + 1), Sigma, 'bo')
ax1.set_title('Energy of each mode, Uncropped Images')
ax1.set_ylabel('log energy')
ax1.set_xlabel('Mode')
#ax1.set_xlim(( -0.5, 25))
#ax1.set_ylim((10**-3, 10**1))


# In[17]:
```

```
#Project onto the left singular vectors, compare reconstructed images to originals
Sigma_mat = np.diag(Sigma)
ranks = [1, 3, 5, 10, 100, 500, 1000, 2000]
image = 5

cols = 3
rows = int(math.ceil((len(ranks) + 1) / cols))

gs = gridspec.GridSpec(rows, cols)
fig = plt.figure(figsize=(2*rows, 2*cols))
for i in range(0, len(ranks) + 1):
    ax = fig.add_subplot(gs[i])
    #if loop has gone through all desired ranks, lastly plot the original image
    if i == len(ranks):
        X_r_reshape = (Xuc[:, image]/np.sqrt(fac - 1)).reshape(uncropped_faces[image].shape)#set the X_r_reshape to be original image for plotting
        ax.set_title('Original Face')
    else:
        rank_reducer = np.ones(fac)
        rank_reducer[ranks[i]:] = 0
        sm = np.dot(Sigma_mat, np.diag(rank_reducer))
        X_r = np.dot(U, np.dot(sm, np.transpose(VT)))
        X_r_reshape = (X_r[:, image]).reshape(uncropped_faces[image].shape)
        ax.set_title('Rank ' + str(ranks[i]) + ' Face')

    ax.imshow(X_r_reshape)
fig.tight_layout()


# In[148]:


#PART II
from pydub import AudioSegment
'''
# convert mp3 files to wav files
parent_dir = "/Users/danielburnham/Documents/AMATH Masters/AMATH582/HW4/Part1_songs/"
for subdir, dirs, files in os.walk(parent_dir):
    for file in files:
        if file.endswith(".mp3"):
            song_path = os.path.join(parent_dir, file)#get os path of song
            sound = AudioSegment.from_mp3(song_path)#use pydub to read in data from mp3
            filename, file_extension = os.path.splitext(song_path)#split the file name from extension for output
            out_path = os.path.join(parent_dir, filename + '.wav')#create new path for convereted file
            sound.export(out_path, format="wav")#export mp3 converted to wav at the output path
'''


# In[95]:


#read in wave files, get song names
parent_dir = "/Users/danielburnham/Documents/AMATH Masters/AMATH582/HW4/Part1_songs/"
songs = []
sample_rates = []
names = []
for subdir, dirs, files in os.walk(parent_dir):
    for file in files:
        if file.endswith(".wav"):
            song_path = os.path.join(parent_dir, file)
            sr, data = wavfile.read(song_path)
            song_ = data.T[0] / 32768 # grab one stereo channel, this is 16-bit track, now normalized on [-1,1)
            song = np.trim_zeros(song_)#trims trailing and leading zeros from start and end of song
            songs.append(song)
            sample_rates.append(sr)
            names.append(file)

#clean up file names to use as artist labels
labels = []#hold artist labels for songs
for name in names:
    artist = name.split(' - ')[0]
    if artist.startswith('Time-H'):
        labels.append('Snoh Aalegra')
    elif artist.startswith('Discl'):
        labels.append('Flume')
    elif artist.startswith('Yes'):
        labels.append('Tame Impala')
    elif artist.startswith('Situat'):
        labels.append('Snoh Aalegra')
    elif artist.startswith('Love'):
        labels.append('Snoh Aalegra')
```

```
    elif artist.startswith('Fool'):
        labels.append('Snoh Aalegra')
    elif artist.startswith('Somet'):
        labels.append('Snoh Aalegra')
    elif artist.startswith('Flume'):
        labels.append('Flume')
    else:
        labels.append(artist)


# In[96]:


#split songs into 5 second chunks and create data matrix
sample_length = 5 #desired length of song samples in seconds
data_list = []
new_labels = []
for i in range(0, len(songs)):
    print("Progress {:2.1%}".format(i / len(songs)), end="\r")#print progress of paint can position acquisition
    num_entries = sample_rates[i]*sample_length
    sample = []
    song = songs[i]
    for j in range(0, len(song)):
        if (j % (num_entries) == 0) and (not(j == 0)):
            artist = labels[i]#get artist name
            new_labels.append(artist)#add artist name to label list for new data list of 5 second samples
            data_list.append(sample)#create data entry of 5 second sample
            sample = []
        sample.append(song[j])

data = np.vstack(data_list)#concatenate list of song sample arrays into one numpy array, each row is a sample


# In[97]:


#compute frequency spectrum for each song sample
r, c = data.shape#get shape of data array
Sgt_spec = np.empty((r, c))#array for storing frequency spectrum data for song samples
for i in range(0, r):
    Sgt = np.fft.fft2(data[i,:].reshape((1, len(data[i,:])))) #calculate fourier transform of song sample
    Sgt_spec[i, :] = abs(np.fft.fftshift(Sgt)) #save shifted fourier transform for spectrogram plotting


# In[98]:


get_ipython().run_line_magic('matplotlib', 'inline')
#Implement classification methods
rs = random.seed(15)#set random seed
classes, y = np.unique(new_labels, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(Sgt_spec, y, test_size=.33, random_state=rs)

""" CLASSIFICATION MODELS """
#Linear Discriminant Analysis (LDA)
clf = LinearDiscriminantAnalysis(tol=0.01)
#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_lda = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_lda)#print series of evaluation metrics

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
#####################
```

```
#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
#####################

# k Nearest Neighbors classifier
print ('\n\nK nearest neighbors classifier\n')
k = 5 # number of neighbors
distance_metric = 'euclidean'
#set up model
knn = KNeighborsClassifier(n_neighbors=k, metric=distance_metric)
#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, knn, classes)
knn.fit(X_train, y_train)#train the model
preds_knn = knn.predict(X_test)#test the model
aScore = accuracy_score(y_test, preds_knn)#accuracy score
CM = confusion_matrix(y_test, preds_knn)# Confusion Matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_knn, average='weighted')#precision score
R = recall_score(y_test, preds_knn, average='weighted')#recall score
F1 = f1_score(y_test, preds_knn, average='weighted')#F1 score
a = {'Results': [aScore, P, R, F1]}#series of evaluation results
aFrame_k = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_k)
###################


# In[99]:


#read in wave files, get song names
parent_dir = "/Users/danielburnham/Documents/AMATH Masters/AMATH582/HW4/Part2_songs/"
songs = []
sample_rates = []
names = []
for subdir, dirs, files in os.walk(parent_dir):
    for file in files:
        if file.endswith(".wav"):
            song_path = os.path.join(parent_dir, file)
            sr, data = wavfile.read(song_path)
            if data.ndim == 2:
                song_ = data.T[0] / 32768 # grab one stereo channel, this is 16-bit track, now normalized on [-1,1)
            else:
                song_ = data.T / 32768
            song = np.trim_zeros(song_)#trims trailing and leading zeros from start and end of song
            songs.append(song)
            sample_rates.append(sr)
            names.append(file)

#clean up file names to use as artist labels
labels = []#hold artist labels for songs
for name in names:
    artist = name.split(' - ')[0]
    labels.append(artist)

#split songs into 5 second chunks and create data matrix
sample_length = 5 #desired length of song samples in seconds
data_list = []
new_labels = []
for i in range(0, len(songs)):
    print("Progress {:2.1%}".format(i / len(songs)), end="\r")#print progress of paint can position acquisition
    num_entries = sample_rates[0]*sample_length#some songs have 44100 Hz sampling rates.. use 48000 Hz so array lengths are constant between songs
    sample = []
    song = songs[i]
    for j in range(0, len(song)):
```

```
        if (j % (num_entries) == 0) and (not(j == 0)):
            artist = labels[i]#get artist name
            new_labels.append(artist)#add artist name to label list for new data list of 5 second samples
            data_list.append(sample)#create data entry of 5 second sample
            sample = []
        sample.append(song[j])

data_jazz = np.vstack(data_list)#concatenate list of song sample arrays into one numpy array, each row is a sample


# In[100]:


#compute frequency spectrum for each song sample
r, c = data_jazz.shape#get shape of data array
Sgt_spec = np.empty((r, c))#array for storing frequency spectrum data for song samples
for i in range(0, r):
    Sgt = np.fft.fft2(data_jazz[i,:].reshape((1, len(data_jazz[i,:])))) #calculate fourier transform of song sample
    Sgt_spec[i, :] = abs(np.fft.fftshift(Sgt)) #save shifted fourier transform for spectrogram plotting


# In[101]:


get_ipython().run_line_magic('matplotlib', 'inline')
#Implement classification methods
rs = random.seed(15)#set random seed
classes, y = np.unique(new_labels, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(Sgt_spec, y, test_size=.33, random_state=rs)

""" CLASSIFICATION MODELS """
#Linear Discriminant Analysis (LDA)
clf = LinearDiscriminantAnalysis(tol=0.01)
#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_lda = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_lda)#print series of evaluation metrics

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
######################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
######################

# k Nearest Neighbors classifier
print ('\n\nK nearest neighbors classifier\n')
```

```
k = 5 # number of neighbors
distance_metric = 'euclidean'
#set up model
knn = KNeighborsClassifier(n_neighbors=k, metric=distance_metric)
#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, knn, classes)
knn.fit(X_train, y_train)#train the model
preds_knn = knn.predict(X_test)#test the model
aScore = accuracy_score(y_test, preds_knn)#accuracy score
CM = confusion_matrix(y_test, preds_knn)# Confusion Matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_knn, average='weighted')#precision score
R = recall_score(y_test, preds_knn, average='weighted')#recall score
F1 = f1_score(y_test, preds_knn, average='weighted')#F1 score
a = {'Results': [aScore, P, R, F1]}#series of evaluation results
aFrame_k = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_k)
###################


# In[102]:


#read in wave files, get song names
parent_dir = "/Users/danielburnham/Documents/AMATH Masters/AMATH582/HW4/Part3_songs/"
songs = []
sample_rates = []
names = []
for subdir, dirs, files in os.walk(parent_dir):
    for file in files:
        if file.endswith(".wav"):
            song_path = os.path.join(parent_dir, file)
            sr, data = wavfile.read(song_path)
            if data.ndim == 2:
                song_ = data.T[0] / 32768 # grab one stereo channel, this is 16-bit track, now normalized on [-1,1)
            else:
                song_ = data.T / 32768
            song = np.trim_zeros(song_)#trims trailing and leading zeros from start and end of song
            songs.append(song)
            sample_rates.append(sr)
            names.append(file)

#clean up file names to use as artist labels
labels = []#hold artist labels for songs
for name in names:
    artist = name.split(' - ')[0]
    labels.append(artist)

#split songs into 5 second chunks and create data matrix
sample_length = 5 #desired length of song samples in seconds
data_list = []
new_labels = []
for i in range(0, len(songs)):
    print("Progress {:2.1%}".format(i / len(songs)), end="\r")#print progress of paint can position acquisition
    num_entries = sample_rates[0]*sample_length#some songs have 44100 Hz sampling rates.. use 48000 Hz so array lengths are constant between songs
    sample = []
    song = songs[i]
    for j in range(0, len(song)):
        if (j % (num_entries) == 0) and (not(j == 0)):
            artist = labels[i]#get artist name
            new_labels.append(artist)#add artist name to label list for new data list of 5 second samples
            data_list.append(sample)#create data entry of 5 second sample
            sample = []
        sample.append(song[j])

data_genre = np.vstack(data_list)#concatenate list of song sample arrays into one numpy array, each row is a sample


# In[103]:


#compute frequency spectrum for each song sample
r, c = data_genre.shape#get shape of data array
Sgt_spec = np.empty((r, c))#array for storing frequency spectrum data for song samples
for i in range(0, r):
    Sgt = np.fft.fft2(data_genre[i,:].reshape((1, len(data_genre[i,:])))) #calculate fourier transform of song sample
    Sgt_spec[i, :] = abs(np.fft.fftshift(Sgt)) #save shifted fourier transform for spectrogram plotting


# In[104]:
```

```
get_ipython().run_line_magic('matplotlib', 'inline')
#Implement classification methods
rs = random.seed(15)#set random seed
classes, y = np.unique(new_labels, return_inverse=True)#get unique class names and translate to integer values
#split into train and test dataframes for predictive modeling
X_train, X_test, y_train, y_test = train_test_split(Sgt_spec, y, test_size=.33, random_state=rs)

""" CLASSIFICATION MODELS """
#Linear Discriminant Analysis (LDA)
clf = LinearDiscriminantAnalysis(tol=0.01)
#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_lda = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_lda)#print series of evaluation metrics

# Logistic regression classifier
print ('\n\n\nLogistic regression classifier\n')
C_parameter = 50. / len(X_train) # parameter for regularization of the model
class_parameter = 'ovr' # parameter for dealing with multiple classes
penalty_parameter = 'l1' # parameter for the optimizer (solver) in the function
solver_parameter = 'saga' # optimization system used
tolerance_parameter = 0.1 # termination parameter
######################

#setting up the model
clf = LogisticRegression(C=C_parameter, multi_class=class_parameter,
                         penalty=penalty_parameter, solver=solver_parameter,
                         tol=tolerance_parameter)

#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, clf, classes)
clf.fit(X_train, y_train)#training the model
preds_logr = clf.predict(X_test)#apply the model
aScore = accuracy_score(y_test, preds_logr)#get accuracy score
# Confusion Matrix
CM = confusion_matrix(y_test, preds_logr)#create confusion matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_logr, average='weighted')#get precision
R = recall_score(y_test, preds_logr, average='weighted')#get recall
F1 = f1_score(y_test, preds_logr, average='weighted')#get F1 score
a = {'Results': [aScore, P, R, F1]}# create series of evaluation metrics
aFrame_l = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_l)#print series of evaluation metrics
######################

# k Nearest Neighbors classifier
print ('\n\nK nearest neighbors classifier\n')
k = 5 # number of neighbors
distance_metric = 'euclidean'
#set up model
knn = KNeighborsClassifier(n_neighbors=k, metric=distance_metric)
#function returns multiclass ROC plot with AUC scores per class
multiclassROC(X_train, y_train, X_test, y_test, knn, classes)
knn.fit(X_train, y_train)#train the model
preds_knn = knn.predict(X_test)#test the model
aScore = accuracy_score(y_test, preds_knn)#accuracy score
CM = confusion_matrix(y_test, preds_knn)# Confusion Matrix
plot_confusion_matrix(CM, classes)#plot of the multiclass confusion matrix
P = precision_score(y_test, preds_knn, average='weighted')#precision score
R = recall_score(y_test, preds_knn, average='weighted')#recall score
F1 = f1_score(y_test, preds_knn, average='weighted')#F1 score
a = {'Results': [aScore, P, R, F1]}#series of evaluation results
aFrame_k = pd.DataFrame(a, index = ['Accuracy', 'Precision', 'Recall', 'F1'])
print(aFrame_k)
###################
```