Eigenfaces & Music Genre Identification

Brandon Goldney[1]

[1]*Department of Applied Mathematics, University of Washington, Email: Goldney@uw.edu*

Abstract

The purpose of this paper is twofold: i) to explore the Yale Faces Database and demonstrate the effect of eigendecomposition on those images and ii) explore the effects of singular value decomposition (SVD) on music. Part 1 of this paper explores the strange but effective methods of facial recognition by converting images into greyscale and then applying PCA on the dataset. Part 2 of the paper attempts to classify pieces of music based off spectrograms of sample music.

1 Section I: Introduction and Overview

Part 1 The accuracy of facial recognition algorithms has significantly increased in recent years. Large companies such as Facebook and Google are leveraging the technology to better serve their customers and strealime certain processes. For example, Facebook has functionality to automaically recognize friends in images, and Google allows users to search for an image (e.g. the Empire State Building) in hopes of correctly identifying it and providing relevant information.

Part 2 Advancements in machine learning have been making impacts in the music industry for several years. For example, currently up for debate is if someone has the right to trademark a song or melody generated by machine learning. Music classification is also important to companies such as Spotify, Apple, and Amazon as they rush to provide customized playlists and music reccommendations through their respective platforms.

2 Section II: Theoretical Background

Part 1 "PCA is defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on." (source: Wikipedia). In order to maximize variance, the first weight vector w_1 satisfies:

$$w_1 = argmax\Sigma_i(t_1)_i^2 = argmax\Sigma(x_i\dot{w})^2$$

The general process for applying PCA is the following:

1. After acquiring data, standardize it by subtracting the mean

2. Calculate the covariance matrix

3. Calcualte the eigenvectors and eigenvalues of the covariance matrix (SVD is the preferred method)

4. Choose components and form a new feature vector

5. Calculate the new data, where $Final\_Data = Row\_Feature\_VectorxRow\_Data\_Adjustment$ and $Row\_Feature\_Vector$ is the matrix with the eigenvectors in the columns transposed so the eigenvectors are in the rows, and $Row\_Data\_Adjustment$ is the mean adjusted data transposed.

Part 2 The Gábor Filter shows the frequency of a signal in the time domain. This is an advancement from the Fourier Transform which shows a signal's amplitude in the frequency domain. In order to localize with respect to both time and frequency Gabor introduced a kernel:

$$g_{t,\omega}(\tau) = e^{i\omega\tau} g(\tau - t)$$

Incorporating Gabor's kernel with the Fourier Transform results in the following:

$$G[f](t,\omega) = \tilde{f}_g(t,\omega) = \int_{-\infty}^{\infty} f(\tau)\bar{g}(\tau - t)e^{i\omega\tau} d\tau = (f, \bar{g}_{t,\omega})$$

The bar denotes the complex conjugate. While this formula appears intimidating at first, it can be broken down into its components to better understand it. Notice, we're integrating over the term τ, which effectively slides the window over the signal. The new term, $\bar{g}(\tau - t)$ was conceptually explained earlier, it localizes the signal with respect to time and frequency. The final term, $e^{-i\omega\tau}$, is the standard Fourier Transform.

The Gábor transform is computed by discretizing the time and frequency domain. Consider the following sample points:

$$v = m\omega_0 \tau = nt_0$$

where *m* and *n* are integers and $w_0, \tau_0 > 0$ are constants. In this case, the discrete version of Gábor's kernel becomes:

$$g_{m,n}(t) = e^{i2\pi m\omega_0 t} g(t - nt_0)$$

Accordingly, the Gábor Transform becomes:

$$\tilde{f}(m,n) = \int_{-\infty}^{\infty} f(t)\bar{g}_{m,n}(t)dt = (f, g_{m,n})$$

3 Section III: Algorithm Implementation and Development

Part 1 Due to the size of the dataset, approximately 2,414 images, we spend some time upfront ensuring the data is in a clean and easy to handle format. There are 40 folders, each with 20+ images. The first step is to utilize the *glob* package in order to create a list of every file name. Once the file names are loaded we can begin processing the data and preparing it for principal components analysis. We start by converting each image into an n-dimensinal array, where each column represents the *i'th* face. Each face has 192x168 pixels, translating into a (32256, 1) array after it is reshaped. After we have the n-dimensional array setup we can normalize the data by subtracting the mean. The next step is to begin PCA on the covariance matrix. Once we have the PCA components, we can then project new data (i.e. attempt to duplicate the faces in the dataset).

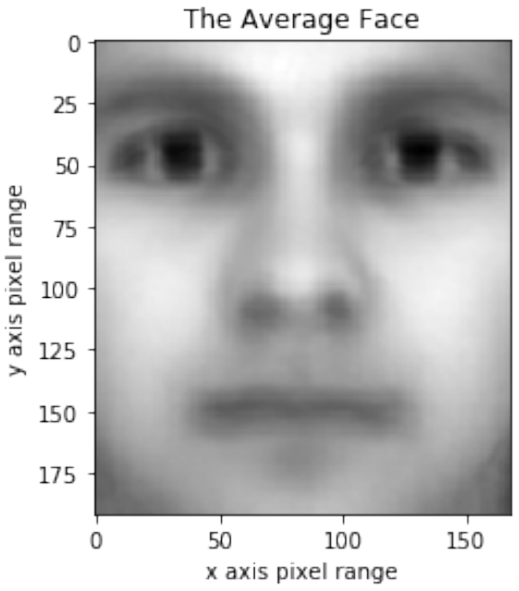
Part 2 After loading the data, the first step is to define the length of the data (i.e usually 5 seconds in this case) and then to discretize it into *n* data points. Next, we rescale the n data points by $(2 * \pi)/L$, where L equals the duration of the song. L is calculated by dividing the number of data points by the sampling frequency. We know in later steps that we will be performing a Fourier Transform, which rearranges X by shifting the zero-frequency component to the center of the array. With that in mind, we shift the *n* data points prior to any further steps. Now that we have the initial variables declared and created the discretized points , we can begin working with the data from the sound file by setting up a for loop, where the window is created and the Fourier Transform is taken on the data.

Additionally steps were also take in an effort to leverage a spectrogram to classify songs from the Beatles, such as taking the SVD of the spectrogram of songs versus themselves. After that approach was not successful, a Random Forest was applied to a series of spectrograms. This created features which could then be used to categorize songs' spectrograms against. This approach had mild success.

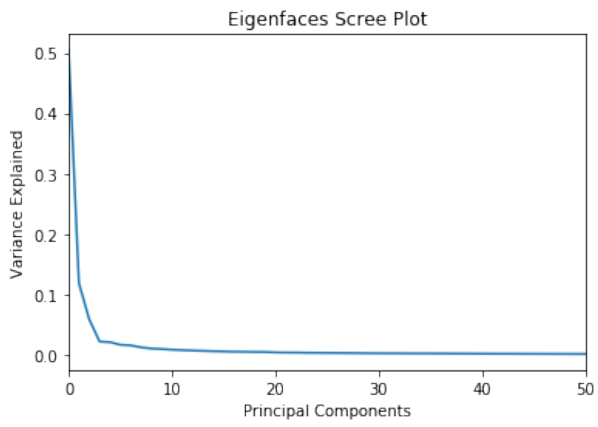
4 Section IV: Computational Results

Part 1

Since we're familiar with PCA at this point, we will pick-up where things can become strange but effective. For example, after subtracting the mean from each image, we can plot the "average" face (shown below).



Additionally, we can use a scree plot to show us how many principal components are needed in order to explain a sufficient amount of variance in order to discern someone's face from a blur of color. The scree plot is shown here:



Also of interest and commonly referenced are the eigenfaces, these are the faces which explain the most variance. Interestingly, it's almost possisble to see the flexibility of PCA. We can see how the nose is mostly in place, but also slightly distorted so it can capture the variance for other people's noses. Similar distortions can be seen in the eyes, mouth, forehead, and complexion.

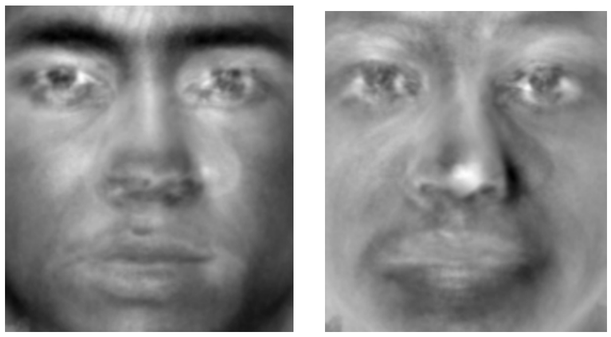
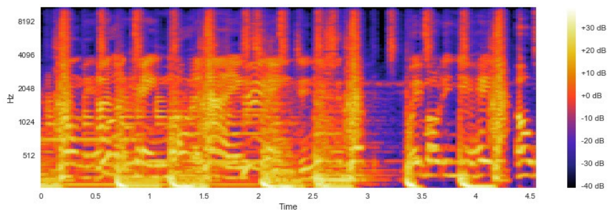


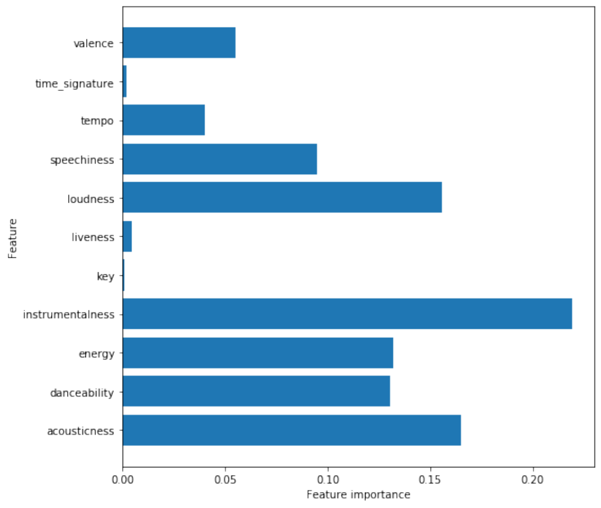
Figure 1. Left: Eigenface 1 Right: Eigenface 2

Part 2 To increase chances of success for this part, we leveraged Ben Frederickson's latent semantic analysis to compare bands. Subsequently the last.fm dataset (more often called the Million Song Dataset) was used to pick the bands and songs. In this section, we'll use The Beatles, The Rolling Stones, and Bob Dylan as the "base bands" to make future comparisons.

The first approach to tackling this problem was to create a spectrogram, then search for certain patterns that are reflective of a specific band and/or genre. However, this approach ran into several problems. For example, the Beatles were notorious for being ahead of their time, thus easily conflating their music with music from other periods. Additionally, music underwent a lot of change from the 60s, 70s, and 80s, increasing the complexity as well. A spectrogram of the Beatles' "Yellow Submarine" is shown below.



Despite reasonable efforts to categorize music by leveraging the spectrogram, no meanigful results were achieved. However, some success was reached using a random forest. For this aspect, the Librosa library from Python was utilized.



5 Section V: Summary and Conclusions

Part 1 PCA has proven to be surprisingly effective at reconstructing faces and providing a reasonably degree of accuracy. It is important to note, while the faces appear blurry to humans, they are less blurry to facial recognition algorithms. The reasoning is because those algorithms often use metrics such as the distance between the nose and mouth, or distance between eyes. As long the facial features are not distorted or spacing is perturbed, the coloring and smoothing can be materially insufficient before it affects an algorithims ability to recognize someone's characteristics.

Part 2 Despite having mixed results for music classification, SVD could prove valueable for music with more consistent tune than the Beatles. An area of further exploration is leveraging a Random Forest, which was adept at identifying features and bucketing songs accordingly.

6 Appendix A: Python functions used and brief implementation explanation

- np.reshape: Gives a new shape to an array without changing its data

- np.zeros: Return a new array of given shape and type, filled with zeros

- sklearn.pca: Linear dimensionality reduction using Singular Value Decomposition of the data to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD
- pd.read_table: Read general delimited file into DataFrame.
- sklearn.RandomForestClassifier: A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting

7 Appendix B: Python code

(see next page)

HW_4

March 17, 2020

0.0.1 Homework 4: Eigenfaces & Music Genre Identification

Github: https://github.com/b-goldney/AMATH_582

Load data

```
[132]: import glob
       path = '/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/
        ↪'

       files = [f for f in glob.glob(path + "**/*.pgm", recursive=True)]

       for f in files:
           print(f)
```

```
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A-050E-40.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+000E+90.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+085E-20.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A-070E+00.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+035E-20.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+020E-10.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+000E+45.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+060E+20.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+110E-20.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A-110E+15.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A+085E+20.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB33/
yaleB33_P00A-025E+00.pgm
```

```
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB15/
yaleB15_P00A+000E+45.pgm
/Users/brandongoldney/Documents/U_Washington/AMATH_582/HW_4/CroppedYale/yaleB15/
yaleB15_P00A+020E-10.pgm
```

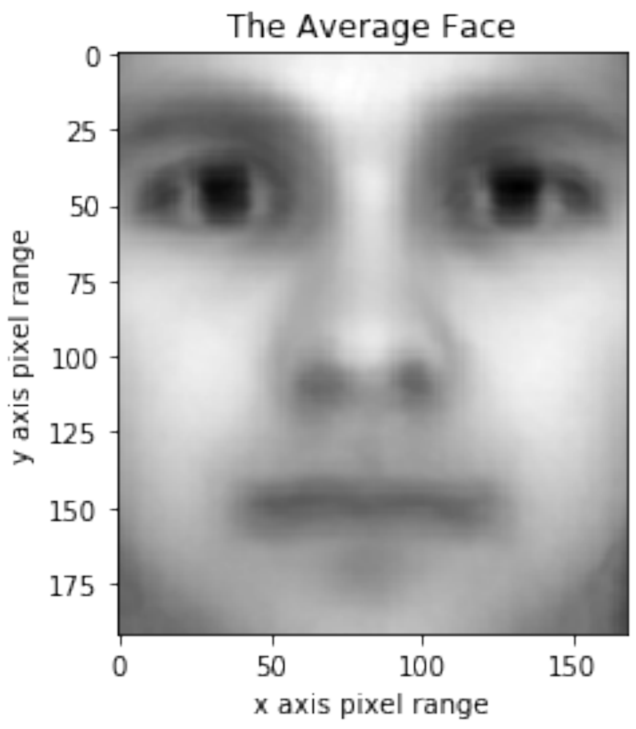
[133]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from skimage.color import rgb2gray
from numpy import asarray
from PIL import Image
import scipy.misc

yale_faces = np.zeros((32256,len(files)))
#np.ndarray(shape=(32256,10), dtype=float, order='F')
for f in range(0,len(files)):
    data = Image.open(files[f])
    yale_faces[:,f] = np.reshape(data,((192*168)))
```

Show the "Average" Face

[160]:
```
mean_image = np.mean(yale_faces, axis=1)
plt.imshow(np.reshape(mean_image,[192,168]), cmap = plt.get_cmap("gray"))
plt.xlabel('x axis pixel range')
plt.ylabel('y axis pixel range')
plt.title('The Average Face')
```

[160]:
```
Text(0.5, 1.0, 'The Average Face')
```



Normalize the Faces

```
[124]: yale_norm = np.zeros((32256,len(files)))
       for i in range(0,len(files)):
           yale_norm[:,i] = yale_faces[:,i] - mean_image
```

PCA

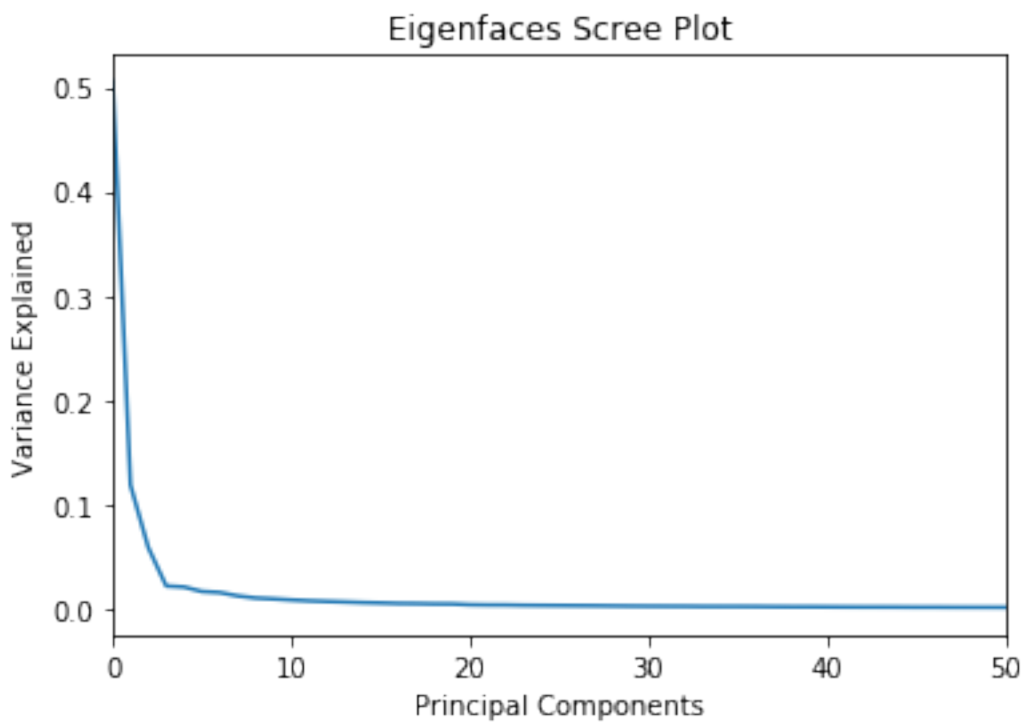
```
[137]: from sklearn.decomposition.pca import PCA

       pca = PCA()
       pca.fit(yale_norm)
```

```
[137]: PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
           svd_solver='auto', tol=0.0, whiten=False)
```

```
[155]: # Let's make a scree plot
       pve = pca.explained_variance_ratio_
       pve.shape
       plt.plot(range(len(pve)), pve)
       plt.title("Eigenfaces Scree Plot")
       plt.ylabel("Variance Explained")
       plt.xlabel("Principal Components")
       plt.xlim(0,50)
```

```
[155]: (0, 50)
```



```
[147]: # eigenfaces
eigenfaces = pca.components_
#plt.imshow(np.reshape(eigenfaces,[192,168]), cmap = plt.
 ↪get_cmap("gray"),dpi=300)
```

```
[161]: img_idx = yale_faces[0]
loadings = pca.components_
n_components = loadings.shape[0]
scores = np.dot(yale_norm[:,:], loadings[:,:].T)

img_proj = []
for n in range(n_components):
    proj = np.dot(scores[img_idx, n], loadings[n,:])
    img_proj.append(proj)
len(img_proj)
```

Music

```
[ ]: # read in triples of user/artist/playcount from the input dataset
data = pandas.read_table("usersha1-artmbid-artname-plays.tsv",
                         usecols=[0, 2, 3],
                         names=['user', 'artist', 'plays'])
```

```

# map each artist and user to a unique numeric value
data['user'] = data['user'].astype("category")
data['artist'] = data['artist'].astype("category")

# create a sparse matrix of all the artist/user/play triples
plays = coo_matrix((data['plays'].astype(float),
                   (data['artist'].cat.codes,
                    data['user'].cat.codes)))
```

```
[ ]: class TopRelated(object):
         def __init__(self, artist_factors):
             # fully normalize artist_factors, so can compare with only the dot␣
      ↪product
             norms = numpy.linalg.norm(artist_factors, axis=-1)
             self.factors = artist_factors / norms[:, numpy.newaxis]

         def get_related(self, artistid, N=10):
             scores = self.factors.dot(self.factors[artistid])
             best = numpy.argpartition(scores, -N)[-N:]
             return sorted(zip(best, scores[best]), key=lambda x: -x[1])
```

```
[ ]: def alternating_least_squares(Cui, factors, regularization, iterations=20):
         users, items = Cui.shape

         X = np.random.rand(users, factors) * 0.01
         Y = np.random.rand(items, factors) * 0.01

         Ciu = Cui.T.tocsr()
         for iteration in range(iterations):
             least_squares(Cui, X, Y, regularization)
             least_squares(Ciu, Y, X, regularization)

         return X, Y

     def least_squares(Cui, X, Y, regularization):
         users, factors = X.shape
         YtY = Y.T.dot(Y)

         for u in range(users):
             # accumulate YtCuY + regularization * I in A
             A = YtY + regularization * np.eye(factors)

             # accumulate YtCuPu in b
             b = np.zeros(factors)

             for i, confidence in nonzeros(Cui, u):
```

```
            factor = Y[i]
            A += (confidence - 1) * np.outer(factor, factor)
            b += confidence * factor

        # Xu = (YtCuY + regularization * I)^-1 (YtCuPu)
        X[u] = np.linalg.solve(A, b)
```

[ ]:
```
frequencies = np.arange(5,105,5)
# Sampling Frequency
samplingFrequency= 400

# Create ndarrays
s1 = np.empty([0])

s2 = np.empty([0])

# Start Value of the sample
start = 1

# Stop Value of the sample

stop= samplingFrequency+1

for frequency in frequencies:
    sub1 = np.arange(start, stop, 1)
    # Signal - Sine wave with varying frequency + Noise
    sub2 = np.sin(2*np.pi*sub1*frequency*1/samplingFrequency)+np.random.
 ↪randn(len(sub1))
    s1      = np.append(s1, sub1)
    s2      = np.append(s2, sub2)
    start   = stop+1
    stop    = start+samplingFrequency

# Plot the signal
plot.subplot(211)
plot.plot(s1,s2)
plot.xlabel('Sample')
plot.ylabel('Amplitude')




# Plot the spectrogram

plot.subplot(212)
powerSpectrum, freqenciesFound, time, imageAxis = plot.specgram(s2,␣
 ↪Fs=samplingFrequency)
```

```
plot.xlabel('Time')
plot.ylabel('Frequency')


plot.show()
```

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(scaled_df, y, test_size=0.
 ↪2, random_state=3)
forest = RandomForestClassifier(n_estimators=100, max_depth= 5)
forest.fit(X_train, y_train)
```

[ ]:
```
def plot_feature_importances(model):
    n_features = X_train.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), X_train.columns.values)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
```

[ ]:
```
pred = forest.predict(X_test)
print(confusion_matrix(y_test, pred))
print(classification_report(y_test, pred))
```

[ ]:
```
# From BriansRebsnik
def plot_feature_importances(model):
    n_features = X_train.shape[1]
    plt.figure(figsize=(8,8))
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), X_train.columns.values)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")
```

[ ]:
```
pred = forest.predict(X_test)
print(confusion_matrix(y_test, pred))
print(classification_report(y_test, pred))
```