Principal Component Analysis

Giovanni Garifo - January 6, 2019

The PACS dataset that we want to analyze is composed of images that are 227*227 pixels each, with colors in RGB format: three color components with intensity from 0 to 255. Thus, every image has 227*227*3=154587 features: each image is characterized by three component of red, green and blue for each of its pixels.

Imports and data structures instantiation

First thing to do is to import the necessary packages, and then read the number of total images to analyze so that we can instantiate the **matrix of samples X** and the **matrix of labels Y**

In [1]:
```
from PIL import Image # Python Image Library
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
import glob
from sklearn import preprocessing
from sklearn.decomposition import PCA
import time
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from itertools import product
```

In [2]:
```
# count the total number of samples in the dataset and instantiate matrices
num_images = 0
for filename in glob.glob("data/*/*.jpg"):
    num_images = num_images + 1

# create matrices of zeroes with the given shape for X and Y
X = np.zeros((num_images,154587)) # 154587 = 227x227x3
Y = np.zeros((num_images,1), np.int8)

print("Number of images in the dataset:", num_images, "\n")
```

```
Number of images in the dataset: 1087
```

Data acquisition

Now we need to populate the **sample matrix X** of **num_images*num_features** elements, so that for each row we have the features of a given image.

For doing this, we read each image using PIL and convert it to a numpy matrix. As we said before, each image is composed of 154587 features, that are the RGB components of each of its pixels.

```
[[[ 48  25  19] <- this are the RGB components of the top-left pixel of the first image
  [ 52  32  25]
  [ 56  37  30]
 ...
]]
```

Then we have to flatten the matrix to a 1 dimensional vector, and then tile up all the vectors to obtain the desired sample matrix X, composed of one row for each image, and where the values of the columns are the features of each image.

In [3]:
```
i=0
for filename in glob.glob("data/*/*.jpg"):
    image = Image.open(filename) # open the image

    image_matrix = np.asarray(image) # create matrix from the image
    image_vector = image_matrix.ravel() #flatten the matrix to a vector
    X[i] = image_vector # populate matrix row with given vector

    # assign class to sample
    if "dog" in filename:
        Y[i] = 1
    elif "guitar" in filename:
        Y[i] = 2
    elif "house" in filename:
        Y[i] = 3
    elif "person" in filename:
        Y[i] = 4

    image.close() # close opened image
    i = i+1

print("Vector representation of the images loaded in X, class labels assigned to Y.\n")
```

```
Vector representation of the images loaded in X, class labels assigned to Y.
```

Data standardization

Before starting the Principal Components Analysis, we must standardize the data: in general, this is needed to obtain the best results for all ML algorithm. Standardize means to scale the features such that they have zero as mean and one as variance, like a standard normal distribution. This is obtained by calculating the standardized value for each feature as

```
x_std = (x-u)/s
```

where "x" is a feature value to standarize, while "u" and "s" are the mean and the variance of the input features.

In [4]:
```
X_std = (X-np.mean(X))/np.std(X)
print("Standardization of X done.\n")
```

```
Standardization of X done.
```

Principal components extraction and features reconstruction

Now we can start the actual PCA: the principal components will be calculated, and then used to reconstruct the feature matrix "X" using only those components.

We'll do the PCA for only the first 60, 6, 2 and the last 6 components.

1. PCA for first 60 principal components

In [5]:
```
print("Starting PCA for first 60 components...\n")
pca = PCA(60)
X_60_components = pca.fit_transform(X_std)
print("First 60 principal components of X extracted. Shape of X_60_components: ", X_60_components.shape)
X_60_reconstructed = pca.inverse_transform(X_60_components)
X_60_reconstructed = (X_60_reconstructed*np.std(X))+np.mean(X)
print("X reconstructed using only the first 60 PC. Shape of X_60_reconstructed: ", X_60_reconstructed.shape, "\n")
```

```
Starting PCA for first 60 components...

First 60 principal components of X extracted. Shape of X_60_components:  (1087, 60)
X reconstructed using only the first 60 PC. Shape of X_60_reconstructed:  (1087, 154587)
```

2. PCA for first 6 principal components

In [6]:
```
# First 6 principal components
print("Starting PCA for first 6 components...\n")
pca = PCA(6)
X_6_components = pca.fit_transform(X_std)
print("First 6 principal components of X extracted. Shape of X_6_components: ", X_6_components.shape)
X_6_reconstructed = pca.inverse_transform(X_6_components)
X_6_reconstructed = (X_6_reconstructed*np.std(X))+np.mean(X)
print("X reconstructed using only the first 6 PC. Shape of X_6_reconstructed: ", X_6_reconstructed.shape, "\n")
```

```
Starting PCA for first 6 components...

First 6 principal components of X extracted. Shape of X_6_components:  (1087, 6)
X reconstructed using only the first 6 PC. Shape of X_6_reconstructed:  (1087, 154587)
```

3. PCA for first 2 principal components

In [7]:
```
print("Starting PCA for first 2 components...\n")
pca = PCA(2)
X_2_components = pca.fit_transform(X_std)
print("First 2 principal components of X extracted. Shape of X_2_components: ", X_2_components.shape)
X_2_reconstructed = pca.inverse_transform(X_2_components)
X_2_reconstructed = (X_2_reconstructed*np.std(X))+np.mean(X)
print("X reconstructed using only the first 2 PC. Shape of X_2_reconstructed: ", X_2_reconstructed.shape, "\n")
```

```
Starting PCA for first 2 components...

First 2 principal components of X extracted. Shape of X_2_components:  (1087, 2)
X reconstructed using only the first 2 PC. Shape of X_2_reconstructed:  (1087, 154587)
```

4. PCA for last 6 principal components

In [8]:
```
print("Starting PCA for last 6 components...\n")
pca = PCA() # get all principal components
X_all_components = pca.fit_transform(X_std)
X_last_6_components = X_all_components.copy()[:,1081:1087] # take only last six components
pca = PCA(6)
pca.fit(X_std)
print("Last 6 principal components of X extracted. Shape of X_last_6_components: ", X_last_6_components.shape)
X_last_6_reconstructed = pca.inverse_transform(X_last_6_components)
X_last_6_reconstructed = (X_last_6_reconstructed*np.std(X))+np.mean(X)
print("X reconstructed using only the last 6 PC. Shape of X_last_6_reconstructed: ", X_last_6_reconstructed.shape, "\n")
```

```
Starting PCA for last 6 components...

Last 6 principal components of X extracted. Shape of X_last_6_components:  (1087, 6)
X reconstructed using only the last 6 PC. Shape of X_last_6_reconstructed:  (1087, 154587)
```

5. PCA for all components

By taking all the components, we are actually reconstructing the original image.

In [9]:
```
print("Starting PCA for all components...\n")
pca = PCA() # get all principal components
X_all_components = pca.fit_transform(X_std)
print("All principal components of X extracted. Shape of X_all_components: ", X_all_components.shape)
X_all_reconstructed = pca.inverse_transform(X_all_components)
X_all_reconstructed = (X_all_reconstructed*np.std(X))+np.mean(X)
print("X reconstructed using all the PC. Shape of X_all_reconstructed: ", X_all_reconstructed.shape, "\n")
```

```
Starting PCA for all components...

All principal components of X extracted. Shape of X_all_components:  (1087, 1087)
X reconstructed using all the PC. Shape of X_all_reconstructed:  (1087, 154587)
```

Variance expressed by the components

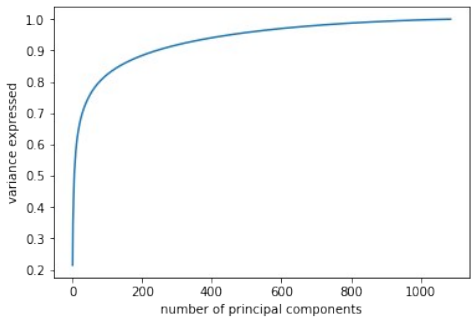
Now that we have computed all the reconstructions, we can analyze the results obtained.

We can have a measure of how much variance we can express with respect to the number of components used for the reconstruction. As it's possible to see from the plot, the sweet spot that allows to have an high value of variance expressed (around 80%) by using the smallest number of principal components required is around 60 principal components.

If we would take around 500 principal components, we will express 95% of the variance of the input features. This is a very high dimensionality reduction with respect to the original 154587 features of an image!

In [10]:
```
# the latest pca instance is computed by taking all the principal components
plt.plot(np.cumsum(pca.explained_variance_ratio_))
plt.xlabel('number of principal components')
plt.ylabel('variance expressed')
```

Out[10]:
```
Text(0,0.5,'variance expressed')
```



Comparison of principal components

Now we are going to plot the values of the first and the second, the third and the fourth, the tenth and eleventh and the last two principal components. A different color is used for each class in the dataset.

For the first two components, we can clearly see that the points are clustering together: in fact, the first two components are the ones that express the highest variance, this means that they retain much more information compared to other components, thus is normal to see that the first two components have very different values for one class in comparison to another.

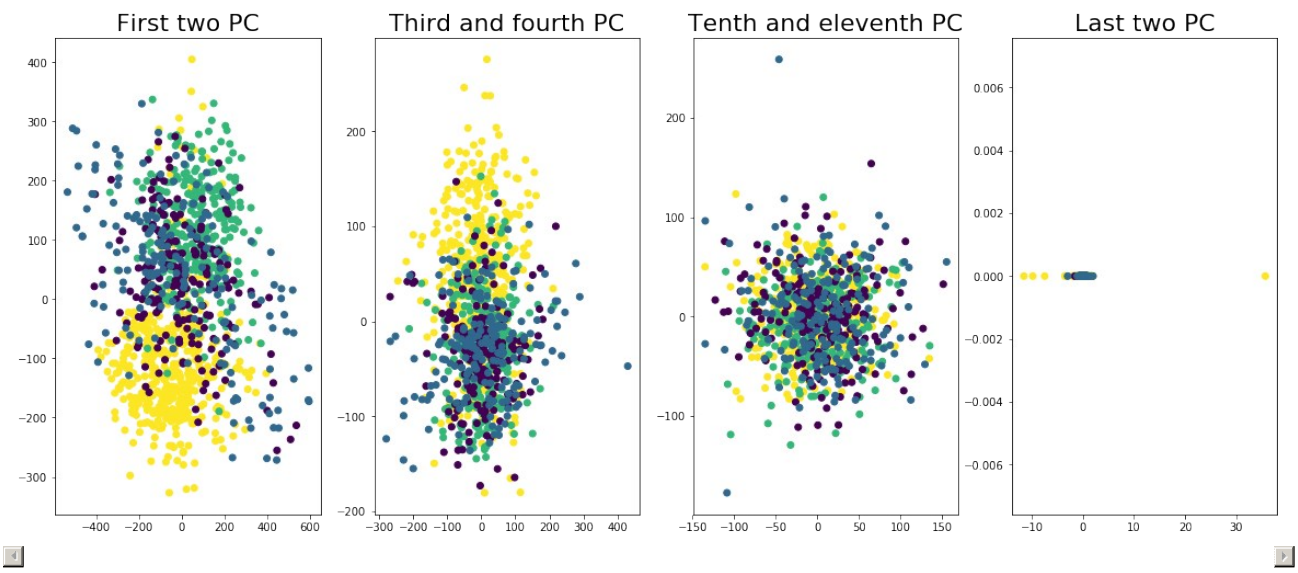
This behaviour is less clear for the third and fourth component, but still can bee seen some form of clustering: they still retain information about the particoular features of a class.

For the tenth and eleventh, the clustering isn't visible: the principal components of the images at this level assumes very similar values.

The information preserved by the last two components is useless: the points are practically collapsing together in one point, having all the same value around zero, because this are the components that collect the lowest variance about the features of the images.

In [11]:
```
plt.figure(figsize=(20,8))

# first two components
plt.subplot(1, 4, 1)
plt.scatter(X_all_components[:,0],X_all_components[:,1], c=Y.ravel())
plt.title('First two PC', fontsize = 22)

# third and fourth components
plt.subplot(1, 4, 2)
plt.scatter(X_all_components[:,3],X_all_components[:,4], c=Y.ravel())
plt.title('Third and fourth PC', fontsize = 22)

# tenth and eleventh
plt.subplot(1, 4, 3)
plt.scatter(X_all_components[:,10],X_all_components[:,11], c=Y.ravel())
plt.title('Tenth and eleventh PC', fontsize = 22)

# last two components
plt.subplot(1, 4, 4)
plt.scatter(X_all_components[:,1085],X_all_components[:,1086], c=Y.ravel())
plt.title('Last two PC', fontsize = 22)
```

Out[11]:
```
Text(0.5,1,'Last two PC')
```



Re-projection with principal components

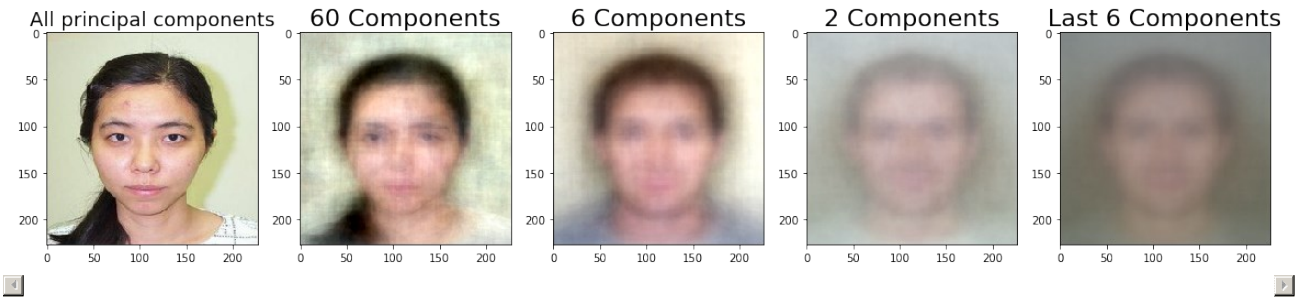
What we said above can be seen more clearly by plotting one image reconstructued using the different sets of principal componenents.

By using the first 60 components, that as we seen express the 80% of the variance of the data, we are able to reconstruct an image that is very similar to the original one: in fact, all the relevant information about the images are expressed by the first 60 components, so with this components we are able to reconstruct a good approximation of the original image.

Even with only the first 6 or first 2 components we're able to reconstruct something that recall a face: the information that can be used to reconstruct the actual face of the selected image is not present, but still the reconstruction can be used to recognize the class of the selected image.

In [12]:
```
# Plot one image for each reconstruction for comparison:
selected_img = 1
plt.figure(figsize=(20,4))

# Original Image (X without standardization)
original_img = np.reshape(X_all_reconstructed[selected_img], (227,227,3)).astype(int)
plt.subplot(1, 5, 1)
plt.imshow(original_img, interpolation='nearest')
plt.title('All principal components', fontsize = 18)

# Reconstructed image using only 60 components
img_60_components = np.reshape(X_60_reconstructed[selected_img], (227,227,3)).astype(int)
plt.subplot(1, 5, 2)
plt.imshow(img_60_components, interpolation='nearest')
plt.title('60 Components', fontsize = 22)

# Reconstructed image using only 6 components
img_6_components = np.reshape(X_6_reconstructed[selected_img], (227,227,3)).astype(int)
plt.subplot(1, 5, 3)
plt.imshow(img_6_components, interpolation='nearest')
plt.title('6 Components', fontsize = 22)

# Reconstructed image using only 2 components
img_2_components = np.reshape(X_2_reconstructed[selected_img], (227,227,3)).astype(int)
plt.subplot(1, 5, 4)
plt.imshow(img_2_components, interpolation='nearest')
plt.title('2 Components', fontsize = 22)

# Reconstructed image using only last 6 components
img_last_6_components = np.reshape(X_last_6_reconstructed[selected_img], (227,227,3)).astype(int)
plt.subplot(1, 5, 5)
plt.imshow(img_last_6_components, interpolation='nearest')
plt.title('Last 6 Components', fontsize = 22)
```

Out[12]:
```
Text(0.5,1,'Last 6 Components')
```



Classification with Naive Bayes Classifier

Now we'll use the dataset to train a Naive Bayes Classifier: the dataset will be splitted in two parts, a train set used for the training of the classifier, and a test set used for testing it's performance.

We'll train and test four classifiers. One will be trained and tested over the original dataset, the only transformation applied will be the standardization. The second will be trained and tested over the dataset after the application of a dimensionality reduction to only the first two principal components of the images. The third one will be trained and tested by using only the third and fourth principal components of the images, and the last one will use only the first 60 principal components as input features.

In [13]:
```
datasets = (X_std, # all features
            X_2_components, # just first two principal components
            X_all_components[:,2:4], # just third and second principal components
            X_60_components) # just first 60 principal components

models = [] # save in this list all the trained models
titles = ("All features", "First two PC", "3rd & 4th PC", "First 60 PC ")

for dataset, title in zip(datasets, titles):

    # separate data in training and test set
    X_train, X_test, Y_train, Y_test = train_test_split(dataset, Y, test_size=0.20)

    # create, fit and test model
    start_time = time.time()
    model = GaussianNB()
    model.fit(X_train, Y_train.ravel()) # fitting on training set
    Y_predicted = model.predict(X_test) # make predictions on test set
    end_time = time.time()

    models.append(model) # save model to plot later

    # compute some statistics on how the classifier performed
    i = 0
    num_mislabeled_points = 0
    Y_predicted.reshape(Y_predicted.shape[0],1)

    while(i != Y_test.shape[0]):
        if Y_predicted[i] != Y_test[i]:
            num_mislabeled_points += 1
        i+=1

    perc_mislabeled_points = (100*num_mislabeled_points)/X_test.shape[0]

    print("%s -> Total=%d, mislabeled=%d (%.1f%%), execution time: %.6f" % (title, X_test.shape[0], num_mislabeled_points, per
c_mislabeled_points, end_time-start_time))
```

```
All features -> Total=218, mislabeled=51 (23.4%), execution time: 2.594890
First two PC -> Total=218, mislabeled=87 (39.9%), execution time: 0.000858
3rd & 4th PC -> Total=218, mislabeled=129 (59.2%), execution time: 0.000716
First 60 PC  -> Total=218, mislabeled=54 (24.8%), execution time: 0.001238
```

Classification results

As we can see from the above results, the best accuracy is obtained by the model trained over the dataset with all the features, this is not astonishing, since every image is represented by 154587 features.

What is important to notice, is how well the the classification perform by using the dataset with the PCA dimensionality reduction applied: using only the first two principal components, our accuracy is only 30% worse in average compared to the model trained by using all the features, moreover, the execution time is several order of magnitude lower compared to the dataset without dimensionality reduction, since we are dealing with only two features for each image.

If we use the first 60 principal components as input features, we see that we loose only 1-2% of accuracy with respect to the model trained on all the input features. This is because, as we previously stated, this are the components who express around 80% of the variance of the input samples. This model required an execution time for training and testing three orders of magnitude lower in comparison with the first model, that used all the features. This is why is always a good choice to perform some dimensionality reduction of the input features before dealing with a classification task.

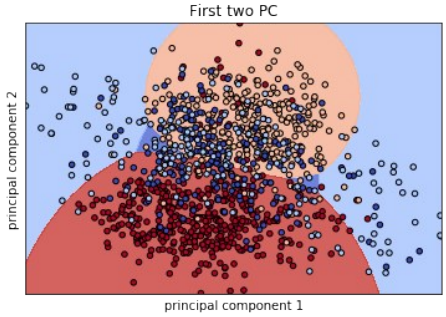
Plotting the decision boundaries

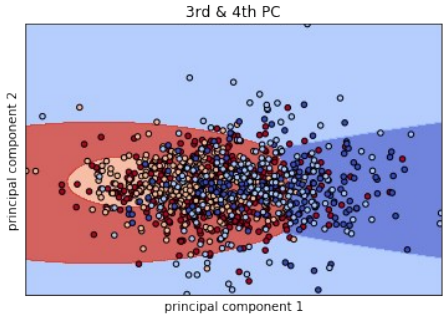
Given that after the dimensionality reduction performed by the PCA for the second and the third model there are only two features for each sample, we can plot in two dimensions the results of the classification with the decision boundaries of the classifier.

As we seen, the mislabeled points for this two models are from 40 to 50 percent: showing the decision boundaries, we can clearly see that this is normal due to the fact that the clusters are overlapped: too less variance is express by only two principal components, even if they're the most important, so the classifier is unable to separate the classes with a good accuracy.

In [14]:

```
def make_meshgrid(x, y, h=.5):
    '''
    Description
    -----------
        Create a mesh of points to plot in

    Parameters
    ----------
        x: data to base x-axis meshgrid on
        y: data to base y-axis meshgrid on
        h: stepsize for meshgrid, optional

    Returns
    -------
        xx, yy : ndarray
    '''

    # calc range max and min values
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1

    # Return coordinate matrices from coordinate vectors.
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    return xx, yy


def plot_contours(axes, model, xx, yy, **params):
    '''
    Description
    -----------
        Plot the decision boundaries for a classifier.

    Parameters
    ----------
        ax: matplotlib axes object
        clf: a classifier
        xx: meshgrid ndarray
        yy: meshgrid ndarray
        params: dictionary of params to pass to contourf, optional
    '''
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = axes.contourf(xx, yy, Z, **params)
    return out


def plotModel(X, Y, model, title):
    '''
    Description
    -----------
        create a figure and plot the result of the prediction with the decision boundaries

    Parameters
    ----------
    fignum : int
        number of the figure
    X : vector
        sample vector
    Y : vector
        labels vectorS
    model : SVC
        the trained svm
    '''
    # create a figure and a set of subplots, return a figure and an array of Axes objects
    fig, axes = plt.subplots(nrows = 1, ncols = 1)
    plt.subplots_adjust(wspace=0.6, hspace=0.6)

    # obtain a coordinate matrices to use as grid
    X0, X1 = X[:, 0], X[:, 1]
    xx, yy = make_meshgrid(X0, X1)

    # assemble plot
    plot_contours(axes, model, xx, yy, cmap=plt.cm.coolwarm, alpha=0.8)
    axes.scatter(X0, X1, c=Y.ravel(), cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    axes.set_xlim(xx.min(), xx.max())
    axes.set_ylim(yy.min(), yy.max())
    axes.set_xlabel('principal component 1')
    axes.set_ylabel('principal component 2')
    axes.set_xticks(())
    axes.set_yticks(())
    axes.set_title(title)

# plot the two models with decision boundaries
plotModel(datasets[1], Y, models[1], titles[1])
plotModel(datasets[2], Y, models[2], titles[2])
```




3rd & 4th PC
principal component 2
principal component 1