calculate_principle_components

December 17, 2020

[1]:
```python
import pandas as pd
import numpy as np
import scipy.linalg as la
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib import image
import glob
import os
import PIL
if 'data' not in os.listdir():
    os.chdir('../')
```

0.1 Analyses

Part (a) Compute the principal components (PCs) using first 190 individuals' neutral expression image. Plot the singular values of the data matrix and justify your choice of principal components.

Part (b) Reconstruct one of 190 individuals' neutral expression image using different number of PCs. As you vary the number of PCs, plot the mean squared error (MSE) of reconstruction versus the number of principal components to show the accuracy of reconstruction. Comment on your result.

Part (c) Reconstruct one of 190 individuals' smiling expression image using different number of PCs. Again, plot the MSE of reconstruction versus the number of principal components and comment on your result.

Part (d) Reconstruct one of the other 10 individuals' neutral expression image using different number of PCs. Again, plot the MSE of reconstruction versus the number of principal components and comment on your result.

Part (e) Use any other non-human image (e.g., car image, resize and crop to the same size), and try to reconstruct it using all the PCs. Comment on your results.

Part (f) Rotate one of 190 individuals' neutral expression image with different degrees and try to reconstruct it using all PCs. Comment on your results.

0.2 Loading the data

[2]:
```
def jpegs_to_matrix(jpegs):
    """
    converts a list of jpegs of the same size into a matrix.
    """
    data = []
    for fn in jpegs:
        img_mat = image.imread(fn)
        img_mat = img_mat.flatten()
        data.append(img_mat)
    data = np.mat(data)
    return(data)
```

[3]:
```
# Loading all the datasets
neutral_fns = 'data/frontalimages_spatiallynormalized_cropped_equalized_part*/
 ↪*a.jpg'
neutral_fns = glob.glob(neutral_fns)
neutral_data = jpegs_to_matrix(neutral_fns)
neutral_data = neutral_data / 255
neutral_190 = neutral_data[0:190].T
neutral_10 = neutral_data[190:200].T

smiling_fns = 'data/frontalimages_spatiallynormalized_cropped_equalized_part*/
 ↪*b.jpg'
smiling_fns = glob.glob(smiling_fns)
smiling_data = jpegs_to_matrix(smiling_fns)
smiling_data = smiling_data / 255
smiling_data = smiling_data.T
smiling_single = smiling_data[190].T
```

0.3 Calculating Eigenfaces

[4]:
```
# saving the number of columns
M = neutral_190.shape[1]

# calculating the mean value across the row (psi vector)
psi_vec = np.mean(neutral_190, axis=1)

# calculating the standardized values (phi matrix)
phi_matrix = neutral_190.copy()
for j in range(phi_matrix.shape[1]):
    phi_matrix[:,j] = phi_matrix[:,j] - psi_vec

A = phi_matrix

# calculating the covariance matrix C
```

```
C = A * A.T
```

0.4 Solving the subproblem

[5]:
```
# calculating A^t * A and its eigenvectors
L = A.T * A
v_evals, v_evecs = la.eig(L)

# calculating the eigen vectors for the C matrix
u_evecs = np.zeros(A.shape)
for l in range(M):
    Av = A * v_evecs[:, l].reshape((-1, 1))
    norm_Av = np.linalg.norm(Av, ord=2)
    u = (Av / norm_Av)
    u_evecs[:, l] = u.ravel()
```

0.5 Part A: calculate the principle components

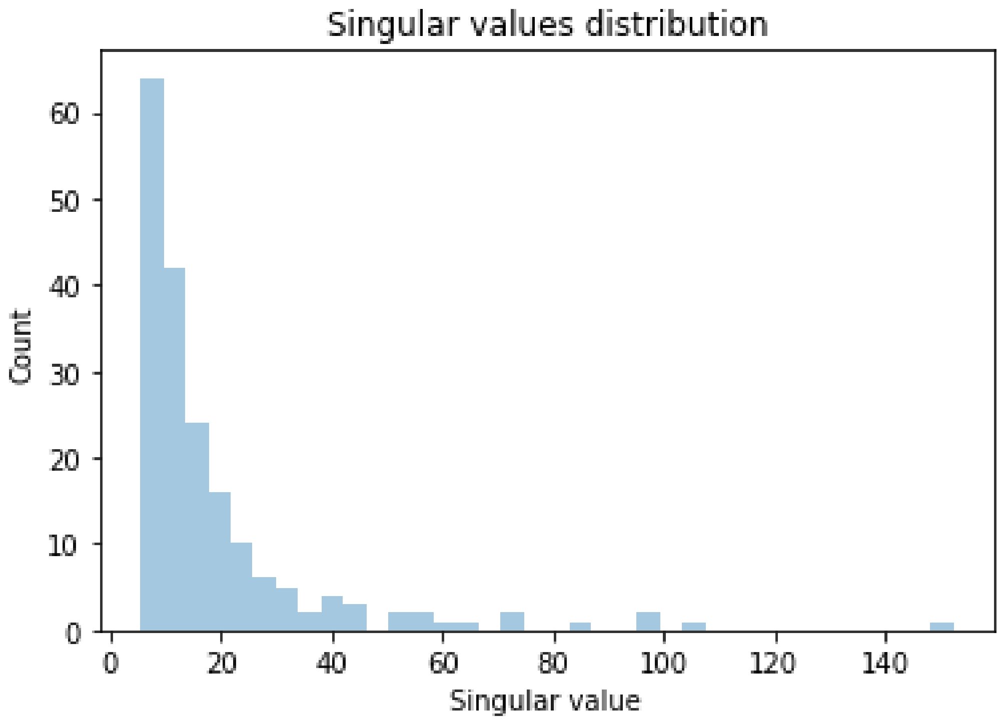
Compute the principal components (PCs) using first 190 individuals' neutral expression image. Plot the singular values of the data matrix and justify your choice of principal components.

According to https://mathworld.wolfram.com/SingularValue.html the singular values of a matrix A are given by the square roots of the eigenvalues of $A^H A$ which means we can take the square root of the v_evals.

[6]:
```
# calculating the singular values from the eigenvalues
positive_vevals = v_evals[v_evals > 0]
singular_values = np.sqrt(positive_vevals)
sorted_idxs = np.argsort(singular_values)[::-1]
singular_values = singular_values[sorted_idxs]
```

[7]:
```
# plotting the distribution of the singular values
fig, ax = plt.subplots()
ax.set_title('Singular values distribution')
ax.set_xlabel('Singular value')
ax.set_ylabel('Count')
sns.distplot(singular_values, kde=False);
```

```
C:\Users\jreyna\Anaconda3\lib\site-packages\seaborn\distributions.py:2551:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
C:\Users\jreyna\Anaconda3\lib\site-packages\numpy\core\_asarray.py:83:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```



For the PC's I will only use those above the mean + std deviation, this is to ensure I use the PC's that capture the most variance.

[8]:
```
# calculating a minimum singular value
singular_value_sr = pd.Series(singular_values)
singular_value_summary = singular_value_sr.describe()
min_singular_value = singular_value_summary['mean'] +␣
 ↪singular_value_summary['std']
top_singular_values = singular_values[singular_values > min_singular_value]
print("I will use a total of {} pc's according to mean + 1 std.".
 ↪format(top_singular_values.shape[0]))
```

```
I will use a total of 18 pc's according to mean + 1 std.
```

0.6 Part B: reconstruct a neutral face using an increasing amount of PC's

Reconstruct one of 190 individuals' neutral expression image using different number of PCs. As you vary the number of PCs, plot the mean squared error (MSE) of reconstruction versus the number of principal components to show the accuracy of reconstruction. Comment on your result.

[9]:
```
def reconstruct_image(pic, psi_vec, u_evecs, num_pcs=None):
    """
    Return a new picture vector which is a reconstruction of the original.

    Params
```

```
    ------

    pic: vector
    the original image vector

    psi_vec: vector
    the average vector

    u_evecs: matrix
    a matrix of eigenvectors

    num_pcs: the number of pcs to use
    """

    # extract the number of eigen vectors that will be used
    if num_pcs:
        final_evecs = u_evecs[:, 0:num_pcs]
    else:
        num_pcs = u_evecs.shape[1]
        final_evecs = u_evecs

    # calculate the weights
    weights = np.zeros(num_pcs)
    for i in range(num_pcs):
        weights[i] = np.dot(final_evecs[:, i], pic)

    # multiply each of the vectors by their corresponding weights
    reconst = final_evecs * weights.reshape((1, -1))

    # sum up the vectors
    reconst = np.sum(reconst, axis=1)

    # add back the average vector
    reconst = psi_vec + reconst.reshape((-1, 1))

    # multiply by the normalizing factor
    reconst = reconst * 255

    return(reconst)
```

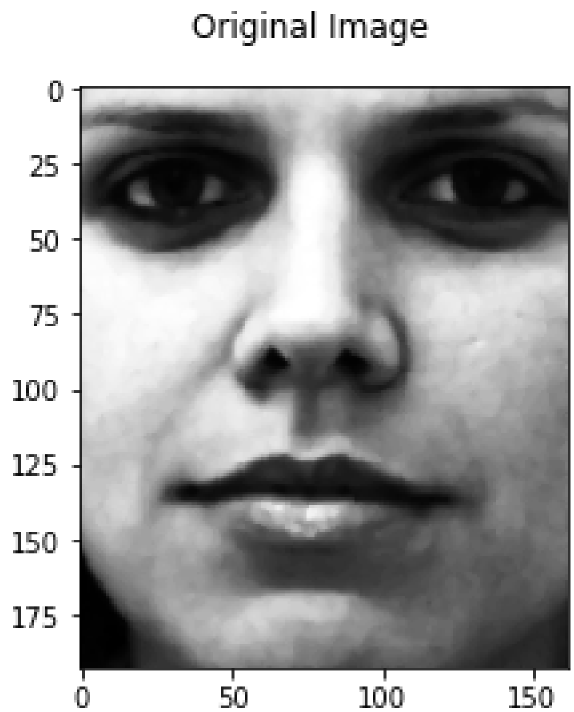
[10]:
```
def mse(x, y):
    """
        Calculated the mse of two picture vectors in column vector form.
    """
    diff = x - y
    diff = diff.reshape(-1).tolist()[0]
    diff = [x**2 for x in diff]
```

```python
    mean_val = np.mean(diff)
    return(mean_val)
```

Below is the original imaged I will be reconstructing

[11]:
```python
fig, ax = plt.subplots()
fig.suptitle('Original Image')
ax.imshow(neutral_190[:, 0].reshape(193, 162), cmap ='gray')
```

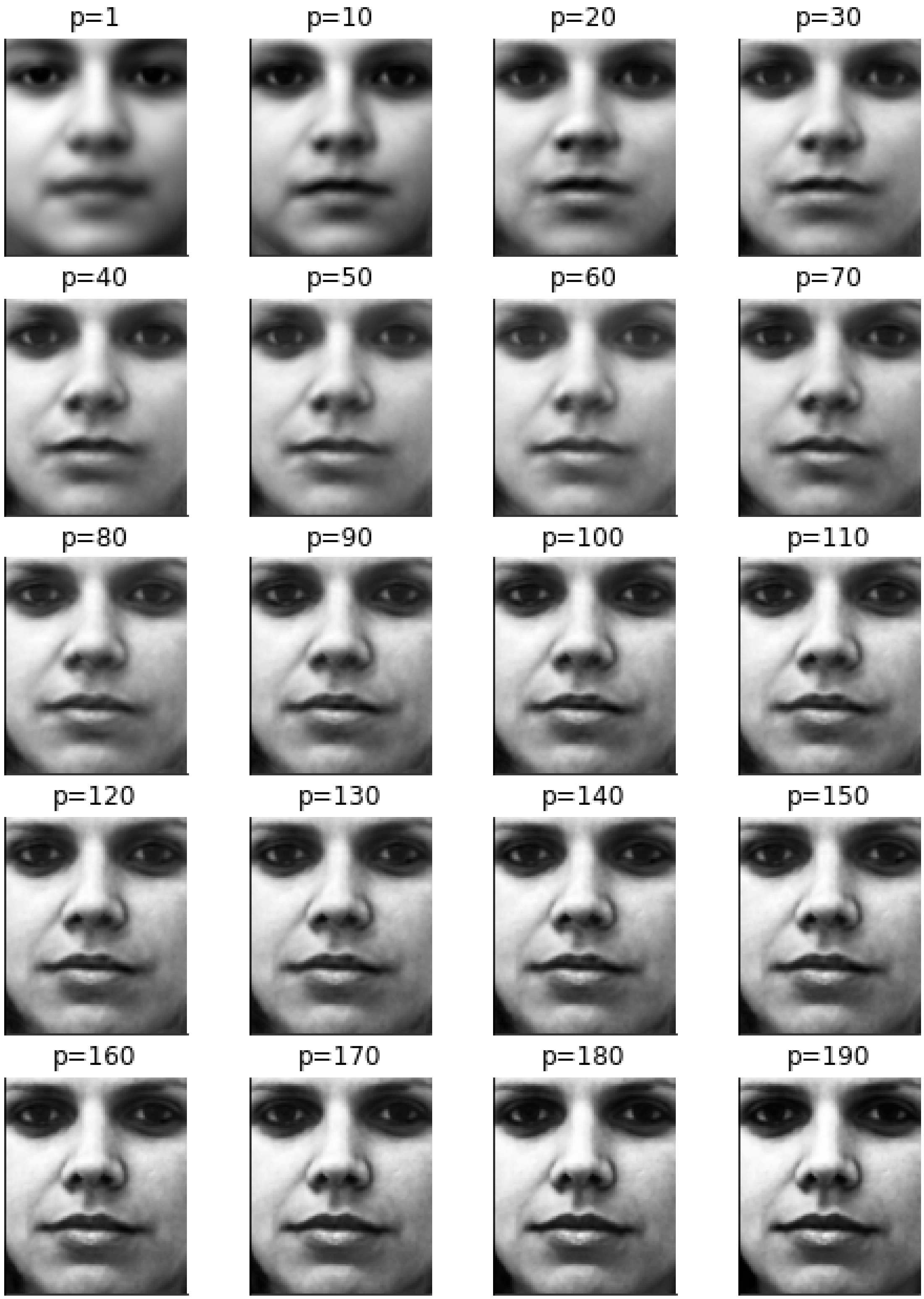
[11]: <matplotlib.image.AxesImage at 0x1888f2a1820>



And in this next section I am reconstructing this face with different numbers of eigenvectors from 1 to 190 and in multiples of 10 (i.e. 1, 10, 20, 30,..., 190.

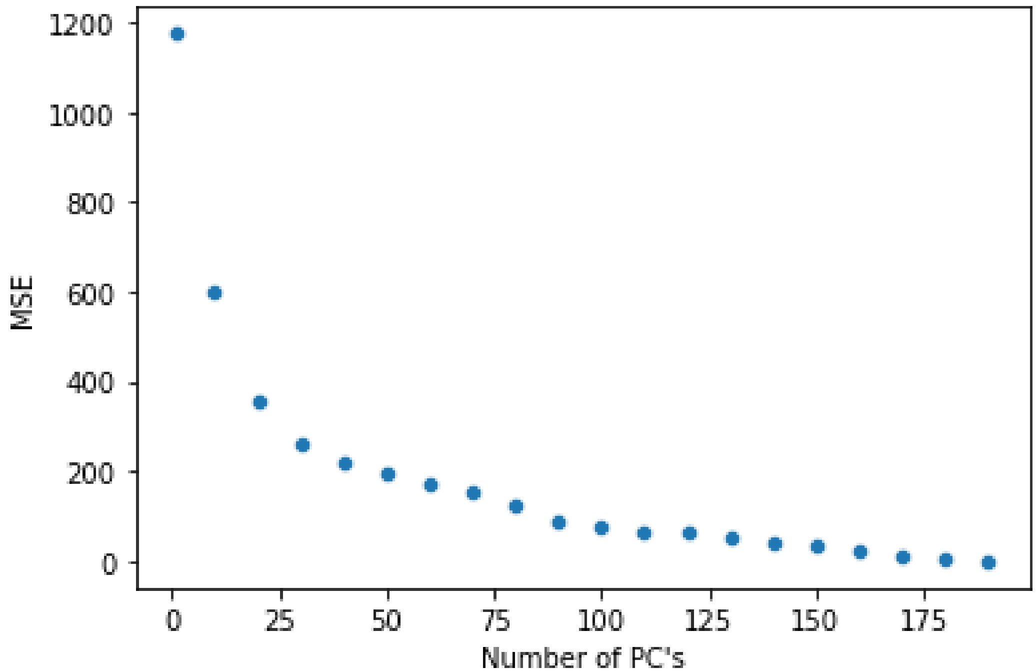
[12]:
```python
fig, axes = plt.subplots(figsize=(8, 11), nrows=5, ncols=4)
axes = axes.flatten()
neutral_pic = A[:, 0]
orig_pic_vec = (neutral_pic + psi_vec) * 255

test_pcs = [1] + list(range(10, 191, 10))

mse_data = []
for i, num_pcs in enumerate(test_pcs):
```

```
    # reconstruct the picture
    reconstruction = reconstruct_image(neutral_pic, psi_vec, u_evecs,␣
↪num_pcs=num_pcs)

    # calculate the MSE
    mse_val = mse(orig_pic_vec, reconstruction)
    mse_data.append([num_pcs, mse_val])

    # plot a face on the current ax
    ax = axes[i]
    ax.imshow(reconstruction.reshape(193, 162), cmap ='gray')
    sns.despine(ax=ax, left=False, bottom=False)
    ax.tick_params(
        axis='both',
        which='both',
        bottom=False,
        left=False,
        labelbottom=False,
        labelleft=False)
    ax.set_title('p={}'.format(num_pcs))
```


p=1
p=10
p=20
p=30
p=40
p=50
p=60
p=70
p=80
p=90
p=100
p=110
p=120
p=130
p=140
p=150
p=160
p=170
p=180
p=190

[13]:
```
mse_data = pd.DataFrame(mse_data, columns=["Number of PC's", 'MSE'])
sns.scatterplot(data=mse_data, x="Number of PC's", y='MSE')
```

[13]: <AxesSubplot:xlabel="Number of PC's", ylabel='MSE'>



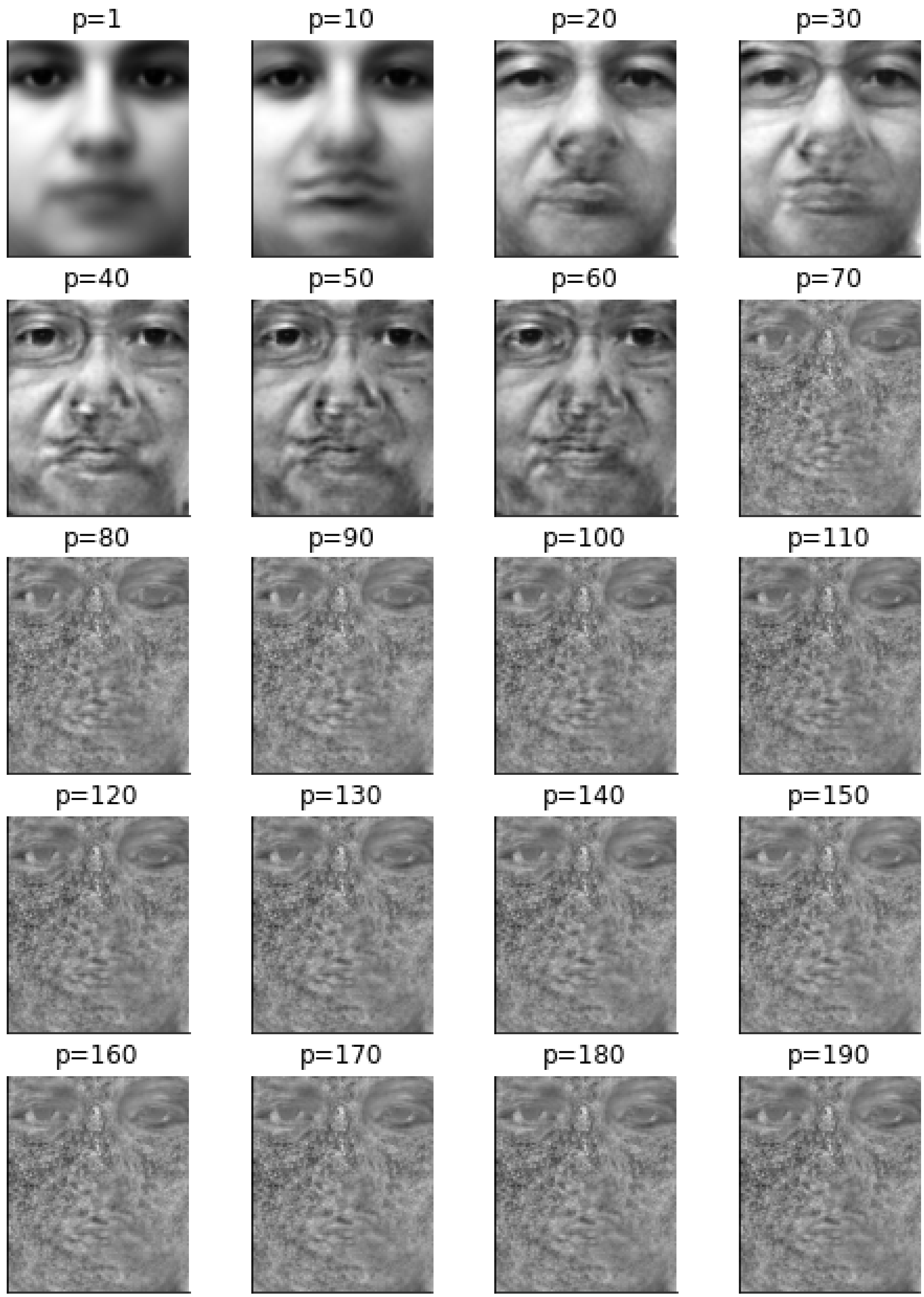
As you increase the number of principal components the face becomes more and more accurate and the MSE error goes down.

0.7 Part C: reconstruct a smiling face using an increasing amount of PC's

Reconstruct one of 190 individuals' smiling expression image using different number of PCs. Again, plot the MSE of reconstruction versus the number of principal components and comment on your result.

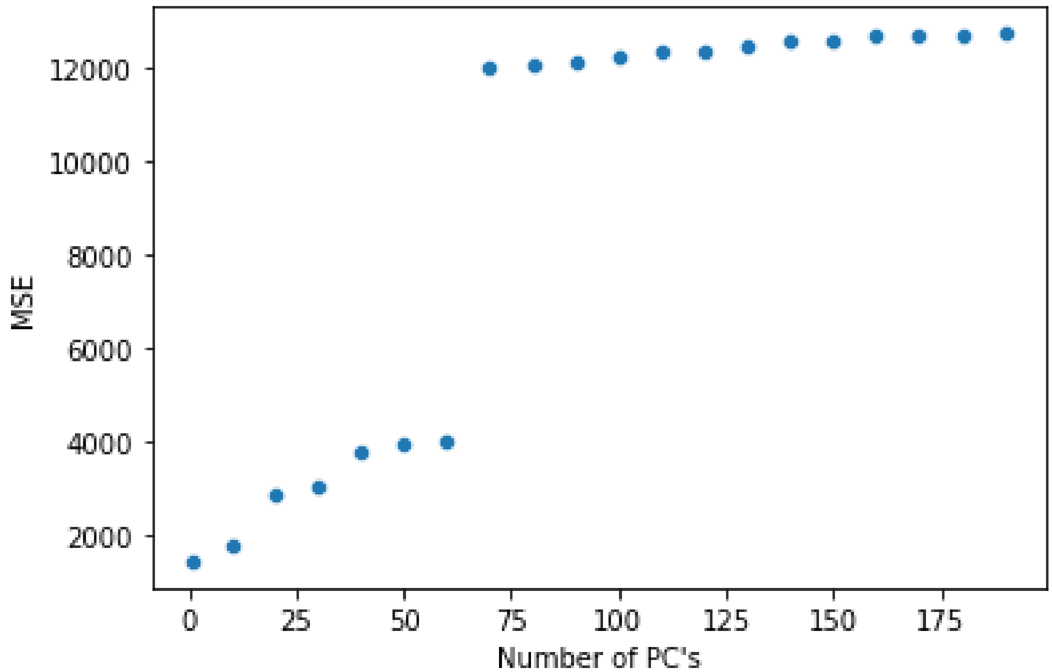
[14]:
```
fig, axes = plt.subplots(figsize=(8, 11), nrows=5, ncols=4)
axes = axes.flatten()
smiling_pic = smiling_data[:, 0]
smiling_pic = smiling_pic / 255
smiling_pic = smiling_pic - psi_vec

test_pcs = [1] + list(range(10, 191, 10))

mse_data = []
for i, num_pcs in enumerate(test_pcs):
```

```
    # reconstruct the picture
    reconstruction = reconstruct_image(smiling_pic, psi_vec, u_evecs,␣
↪num_pcs=num_pcs)

    # calculate the MSE
    mse_val = mse(orig_pic_vec, reconstruction)
    mse_data.append([num_pcs, mse_val])

    # plot a face on the current ax
    ax = axes[i]
    ax.imshow(reconstruction.reshape(193, 162), cmap ='gray')
    sns.despine(ax=ax, left=False, bottom=False)
    ax.tick_params(
        axis='both',
        which='both',
        bottom=False,
        left=False,
        labelbottom=False,
        labelleft=False)
    ax.set_title('p={}'.format(num_pcs))
```


p=1
p=10
p=20
p=30
p=40
p=50
p=60
p=70
p=80
p=90
p=100
p=110
p=120
p=130
p=140
p=150
p=160
p=170
p=180
p=190

[15]:
```
mse_data = pd.DataFrame(mse_data, columns=["Number of PC's", 'MSE'])
sns.scatterplot(data=mse_data, x="Number of PC's", y='MSE')
```

[15]: <AxesSubplot:xlabel="Number of PC's", ylabel='MSE'>



As you increase the number of PC's the image reconstruction actually looks works and the MSE keeps increaseing and grows drastically around 75 PC's.

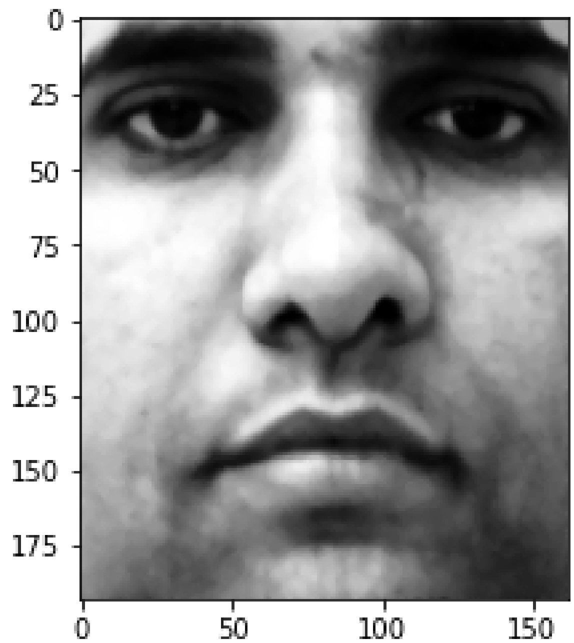
0.8 Part D: reconstruct a neutral face (from the 10 unused samples) using an increasing amount of PC's

Reconstruct one of the other 10 individuals' neutral expression image using different number of PCs. Again, plot the MSE of reconstruction versus the number of principal components and comment on your result.

The neutral picture I am using is the one below:

[16]:
```
neutral_pic = neutral_10[:, 0]
fig, ax = plt.subplots()
ax.imshow(neutral_pic.reshape(193, 162), cmap ='gray')
```

[16]: <matplotlib.image.AxesImage at 0x18a71e87d60>

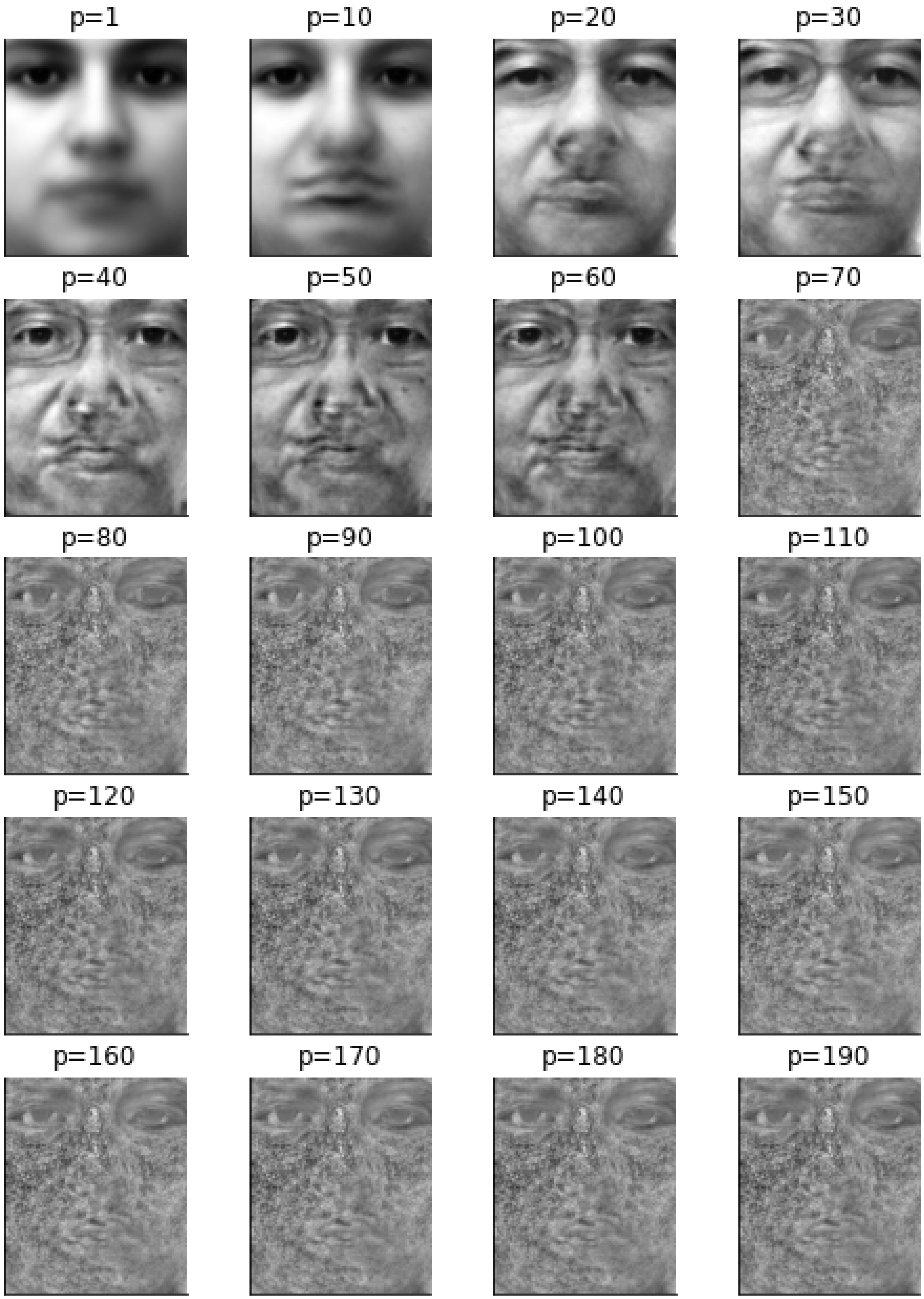


I then process this picture and reconstruct it using an increasing number of eigenvectors as I did before.

[17]:
```
neutral_processed = neutral_pic / 255
neutral_processed = neutral_processed - psi_vec
```

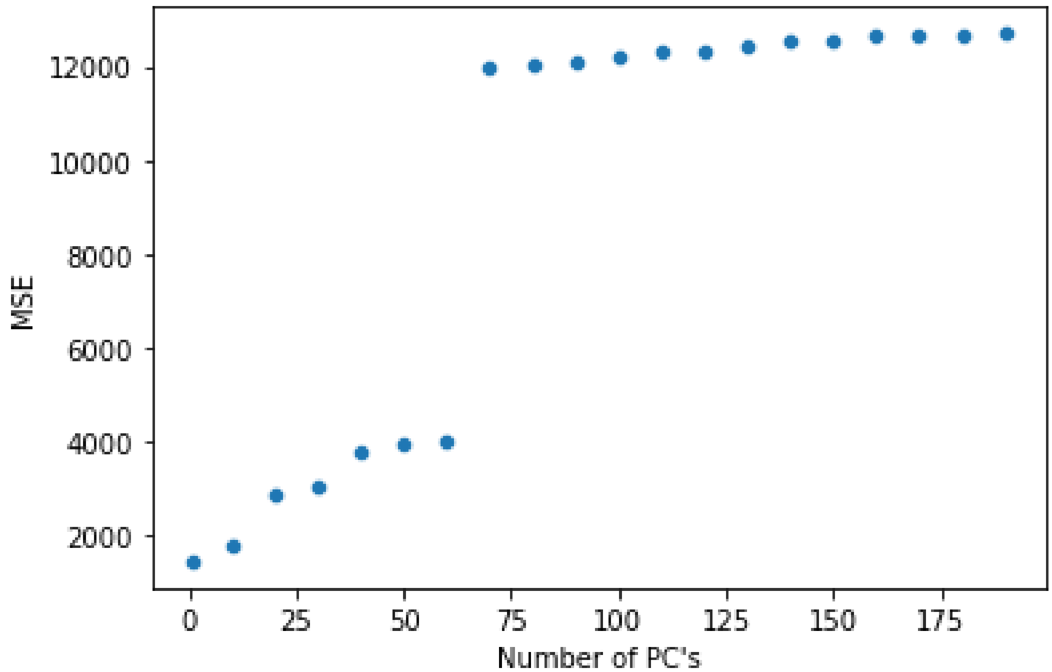
[18]:
```
fig, axes = plt.subplots(figsize=(8, 11), nrows=5, ncols=4)
axes = axes.flatten()

mse_data = []
test_pcs = [1] + list(range(10, 191, 10))
for i, num_pcs in enumerate(test_pcs):

    # reconstruct the picture
    reconstruction = reconstruct_image(neutral_processed, psi_vec, u_evecs,␣
 ↪num_pcs=num_pcs)

    # calculate the MSE
    mse_val = mse(orig_pic_vec, reconstruction)
    mse_data.append([num_pcs, mse_val])

    # plot a face on the current ax
    ax = axes[i]
    ax.imshow(reconstruction.reshape(193, 162), cmap ='gray')
    sns.despine(ax=ax, left=False, bottom=False)
    ax.tick_params(
```

```
        axis='both',
        which='both',
        bottom=False,
        left=False,
        labelbottom=False,
        labelleft=False)
    ax.set_title('p={}'.format(num_pcs))
```


p=1
p=10
p=20
p=30
p=40
p=50
p=60
p=70
p=80
p=90
p=100
p=110
p=120
p=130
p=140
p=150
p=160
p=170
p=180
p=190

```
[19]: mse_data = pd.DataFrame(mse_data, columns=["Number of PC's", 'MSE'])
      sns.scatterplot(data=mse_data, x="Number of PC's", y='MSE')
```

```
[19]: <AxesSubplot:xlabel="Number of PC's", ylabel='MSE'>
```



As you increase the number of PC's the image reconstruction actually looks worse and the MSE keeps increasing and increases drastically around 75 PC's. Seems like this idea mainly works for faces which are part of the training set and not completely new faces which would explain why the authors of the original paper thought of this method as a facial recognition tool.

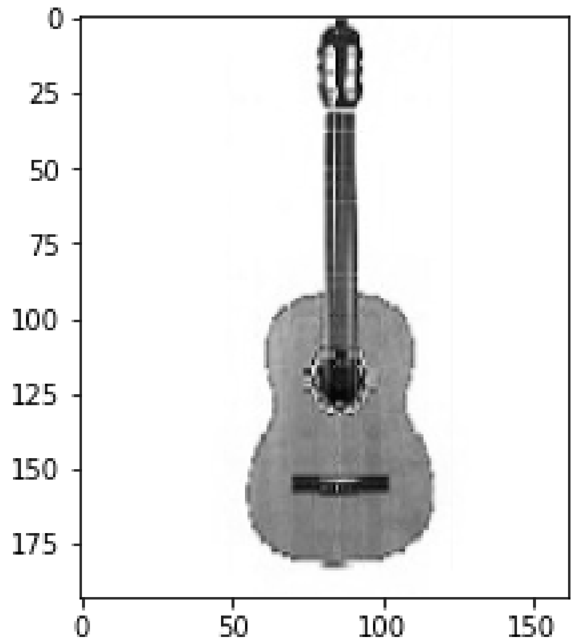
0.9 Part E: use any other non-human image

For the non-human image I am using a guitar.

```
[20]: guitar = PIL.Image.open('figures/guitar_bw_193_162.jpg')
      guitar = guitar.convert('L')
      guitar = np.array(guitar)
      guitar = guitar.flatten()
      orig_guitar = guitar.copy()
      guitar = guitar / 255
      guitar = guitar - psi_vec.reshape(-1)
      guitar = np.array(guitar).reshape((-1,))
```

```
[21]: fig, ax = plt.subplots()
      ax.imshow(orig_guitar.reshape(193, 162), cmap ='gray')
```

[21]: <matplotlib.image.AxesImage at 0x1888f34b400>



[22]:
```
reconstruction = reconstruct_image(guitar, psi_vec, u_evecs, num_pcs=10)

fig, ax = plt.subplots()

ax.imshow(reconstruction.reshape(193, 162), cmap ='gray')

sns.despine(ax=ax, left=False, bottom=False)

ax.tick_params(
    axis='both',
    which='both',
    bottom=False,
    left=False,
    labelbottom=False,
    labelleft=False)
```

After reconstruction we still have a face structure so the projection onto the face space still produces a face image but it looks slightly distorted.

0.10 Part F: rotate an individuals neutral face and reconstruct with all PC's

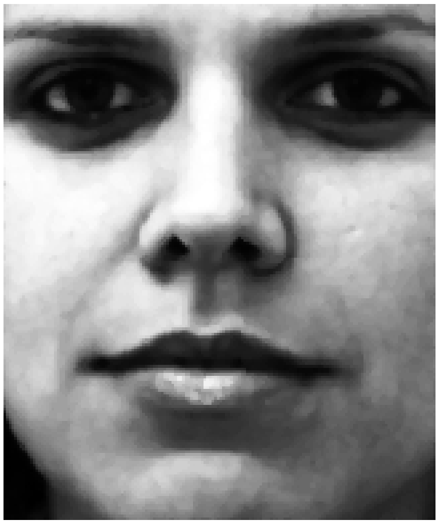
Rotate one of 190 individuals' neutral expression image with different degrees and try to reconstruct it using all PCs. Comment on your results.

I will rotate the face from part B which is how again below:

[23]:
```
# Create an Image object from an Image
neutral_pic = PIL.Image.open(neutral_fns[0])
```

[24]:
```
neutral_pic
```

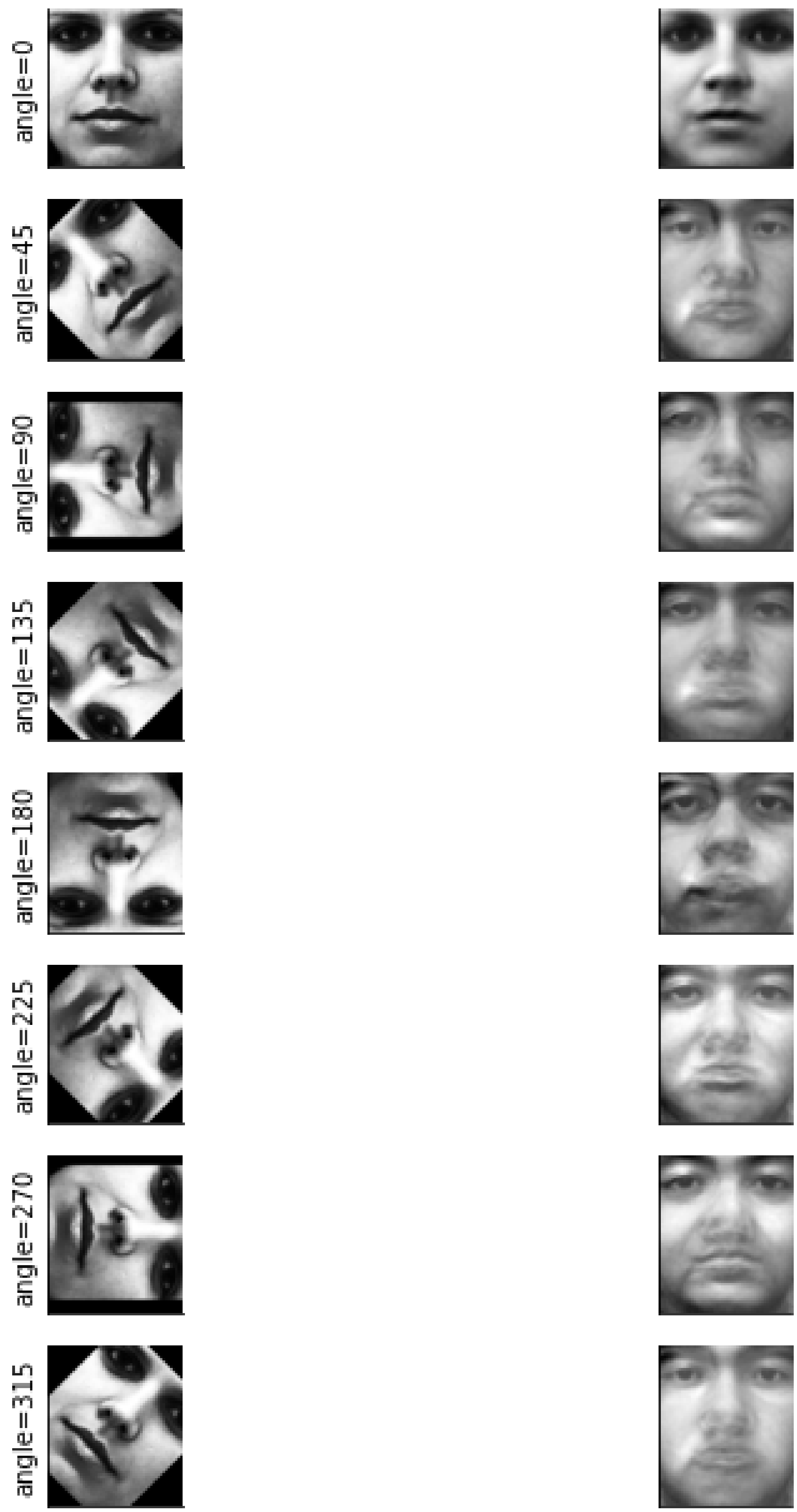
[24]:



I performed several 45 degree rotations and reconstructed each of the images.

[25]:
```python
angles = [0, 45, 90, 135, 180, 225, 270, 315]
fig, axes = plt.subplots(figsize=(8, 11), nrows=len(angles), ncols=2)

for i, angle in enumerate(angles):

    # Rotate it by 45 degrees
    normal = neutral_pic.rotate(angle)
    normal = np.array(normal).flatten()

    # drawing the rotated original image
    ax = axes[i, 0]
    ax.imshow(normal.reshape(193, 162), cmap ='gray')
    sns.despine(ax=ax, left=False, bottom=False)
    ax.tick_params(
        axis='both',
        which='both',
        bottom=False,
        left=False,
        labelbottom=False,
        labelleft=False)
    ax.set_ylabel('angle={}'.format(angle))

    # reconstruction using the rotated images
    rotated  = normal / 255
    rotated = rotated - psi_vec.reshape(-1)
    rotated = np.array(rotated).reshape(-1)
    reconstruction = reconstruct_image(rotated, psi_vec, u_evecs,␣
↪top_singular_values.shape[0])
```

```
# draw the reconstruction matrix
ax = axes[i, 1]
ax.imshow(reconstruction.reshape(193, 162), cmap ='gray')
sns.despine(ax=ax, left=False, bottom=False)
ax.tick_params(
    axis='both',
    which='both',
    bottom=False,
    left=False,
    labelbottom=False,
    labelleft=False)
```


angle=0
angle=45
angle=90
angle=135
angle=180
angle=225
angle=270
angle=315

No matter how we rotate we always construct an image that is rightside up and most image are severally distorted. The only image that is vaguely human is the picture rotated 180 degrees.