Assignment 1: PCA

What you will learn

- Working with images using scikit-image
- PCA using scikit-learn
- Practical applications of PCA

Setup

- Download Anaconda Python 3.6 for consistent environment.
- If you use pip environment then make sure your code is compatible with versions of libraries provided within Anaconda's Python 3.6 distribution.

Submission

- Do not change any variable/function names.
- Just add your own code and don't change existing code
- Save this file and rename it to be **studentid_lastname.ipynb** (student id (underscore) last name.ipynb) where your student id is all numbers.
- Export your .ipynb file to PDF (File > Download as > PDF via Latex). **Please don't leave this step for final minutes**.
- Submit both the notebook and PDF files.
- If you happen to use any external library not included in Anaconda (mention in **Submission Notes** section below)

Submission Notes

(Please write any notes here that you think I should know during marking)

[NO MARKS] PCA Warming Up (MUST READ)

I'm adding some code to illustrate examples of PCA using sklearn library.

Let's create some random `5d` data

In [2]:

```
import numpy as np
from sklearn.decomposition import PCA

# 100 points of 5d data
data = np.random.rand(100, 5)
```

Lets convert this `5d` data to `2d` using PCA`

In [3]:

```
# n_components=2 because I want to convert 5d data to 2d (dimensionality reduction)
pca = PCA(n_components=2)
pca.fit(data)
```

Out[3]:

```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

In code above when we call `fit`, it populates two things in `pca`:

1. `mean_`
2. `components_`

In [4]:

```
# mean of the input data (per dimension) used to zeroying the mean
pca.mean_
```

Out[4]:

```
array([0.49238157, 0.51424949, 0.52535091, 0.48753499, 0.50987034])
```

In [5]:

```
# basis vectors
pca.components_
```

Out[5]:

```
array([[-1.82179984e-01, -6.52900796e-01, -5.39639442e-01,
         3.89895344e-01,  3.11932521e-01],
       [ 5.39383708e-01,  1.43077571e-01,  3.11672444e-04,
         7.59500711e-01, -3.34294176e-01]])
```

Now we are ready to transform `5d` from `data` into `2d` using following code

In [6]:

```
data_to_reduce = data[:10]
reduced_data = np.dot(data_to_reduce - pca.mean_, pca.components_.T)

# reduced data from 5d to 2d
reduced_data.shape
```

Out[6]:

```
(10, 2)
```

You can accomplish the same using `transform` function provided in `pca`

In [7]:

```
reduced_data
```

Out[7]:

```
array([[ 0.16564668,  0.14268097],
       [-0.57357638, -0.07735401],
       [-0.54709867, -0.01441193],
       [-0.13606588, -0.08365607],
       [-0.37440652,  0.16751614],
       [ 0.00984816,  0.23033057],
       [-0.08121188, -0.27897597],
       [-0.1346943 ,  0.35806529],
       [ 0.62844905,  0.23750929],
       [-0.15723814, -0.14731035]])
```

Time for inverse transform or changing `2d` data back to `5d`

Compression --> Decompression

In [8]:

```
decompressed_data = np.dot(reduced_data, pca.components_)+pca.mean_
decompressed_data
```

Out[8]:

```
array([[0.53916385, 0.42651309, 0.4360059 , 0.66048616, 0.51384351],
       [0.55515222, 0.87767035, 0.83485125, 0.2051498 , 0.3568122 ],
       [0.58427844, 0.86938863, 0.82058245, 0.26327789, 0.34403029],
```

```
       [0.47204733, 0.59111771, 0.59875136, 0.37094669, 0.4953927 ],
       [0.65094642, 0.78266761, 0.72744765, 0.46878426, 0.3370811 ],
       [0.61482399, 0.54077476, 0.52010825, 0.66631098, 0.43594413],
       [0.35670166, 0.52735759, 0.5690891 , 0.24398841, 0.57779775],
       [0.71005476, 0.65342262, 0.59814887, 0.70696915, 0.34815566],
       [0.50599938, 0.13791686, 0.18628904, 0.91295282, 0.62650606],
       [0.44157041, 0.59583359, 0.6101569 , 0.31434626, 0.51006764]])
```

In [9]:

```
# same can we accomplished using inverse_transform
pca.inverse_transform(pca.transform(data_to_reduce))
```

Out[9]:

```
array([[0.53916385, 0.42651309, 0.4360059 , 0.66048616, 0.51384351],
       [0.55515222, 0.87767035, 0.83485125, 0.2051498 , 0.3568122 ],
       [0.58427844, 0.86938863, 0.82058245, 0.26327789, 0.34403029],
       [0.47204733, 0.59111771, 0.59875136, 0.37094669, 0.4953927 ],
       [0.65094642, 0.78266761, 0.72744765, 0.46878426, 0.3370811 ],
       [0.61482399, 0.54077476, 0.52010825, 0.66631098, 0.43594413],
       [0.35670166, 0.52735759, 0.5690891 , 0.24398841, 0.57779775],
       [0.71005476, 0.65342262, 0.59814887, 0.70696915, 0.34815566],
       [0.50599938, 0.13791686, 0.18628904, 0.91295282, 0.62650606],
       [0.44157041, 0.59583359, 0.6101569 , 0.31434626, 0.51006764]])
```

In [10]:

```
# Lets find compression decompression error (absolute mean error)
np.sum(np.abs(data_to_reduce - decompressed_data))/data_to_reduce.size
```

Out[10]:

```
0.16518751158233452
```

Questions [8 marks]

Answer the questions below as follows:

1) What is 2+2

- 4
- 5
- 6

In [11]:

```
ans1 = 4
ans1
```

Out[11]:

```
4
```

2) (2 marks) For a n-D data, you can ALWAYS reconstruct the data with 0\% error if all `n` PCAs are used.

- True
- False

In [1]:

```
ans2 = True
ans2
```

Out[1]:

```
True
```

3) (2 marks) From the 2nd tutorial, we ran PCA alorithm on faces. We called the extracted PCs--Eigenfaces. What is the value of a dot product between arbitary two eigen faces?

In [4]:

```
ans3 = 0
ans3
```

Out[4]:

```
0
```

4) (4 marks) Using the probablity, find the expected value for the function below:

Note: Lets say for a coin toss, head = 0 and tail = 1. Then the expected value for a coin-toss will be `p(x=tail)*1 + p(x=head)*0 = 1/2*1 = 0.5`.

In [7]:

```
def func():
    arr = [49, 8, 48, 15, 47, 4, 16, 23, 43, 44, 42, 45, 46]
    np.random.shuffle(arr)
    return min(arr[0:6])

ans4 = 8.181
ans4
```

Out[7]:

```
8.181
```

Programming Tasks [92 marks]

Task 1: Building an Image Compression Algorithm

In this section you will build you own compression algorithm for images using PCA.

STEP 1: Read the image

1. Use `imread` function from `skimage.io` to read `leena.jpg`.
2. Show the image using `show_image` function provided to you

In [8]:

```
import matplotlib.pyplot as plt
# make matplotlib to show plots inline
%matplotlib inline

from skimage.io import imread

def show_image(img, title='Image Dimension'):
    if len(img.shape) == 2:
        plt.imshow(img, cmap='Greys_r')
    else:
        plt.imshow(img)
    plt.axis('off')
    plt.title('{0}: {1},{2}'.format(title, img.shape[0], img.shape[1]), fontsize=20)

# !!ADD CODE HERE!!
image = imread('leena.jpg')

show_image(image)
```

STEP 2: Compressing an image using PCA

Image is generally made of 3 (or 4) channels (RGB), we will build a compression algorithm that applies one channel at a time to compress multi-channel image.

In order to compress entire image you will compress all the channels within the image one by one and then serialize compressed channels and any auxiliary data required for decompression (for ex. principle components (components_), means (means_), original image size).

In order to decompress, you will deserialize the data, then uncompress the compressed channel one by one and stack them up to rebuild the uncompressed version of the original image.

Compression Strategy using PCA

- The above image you have read is of size `350 x 780`, i.e. width is `780` and height is `350`
- Patch the image into `10x10` patches yielding `35x78=2730` patches in total.
- Flatten each patch `(10x10)` into `100` dimensional vector.
- Now (for each channel) you will have `2730` number of `100-d` vectors.
- Apply PCA on these vectors.
- Reduce the dimensionality of `100-d` vector to `5-d`.

I've given you two functions `patchify` and `depatchify`.

`patchify` creates `100-d` vectors from all the patches from a given image and `depatchify` combines these `100-d` patches back to the image of the given size.

Please read through code below and figure out how these two functions work.

NOTE: `convert_to_cf` is important function to note (in cell below). It converts channel last format image to channel first format. Images that you read through `imread` function returns array of shape `X x Y x 3` channel is last axis, it is easier if channel were first axis then to extract any channel you can do use first indexer.

In [9]:

```
from sklearn.decomposition import PCA
import numpy as np


def patchify(img, ps=(10, 10)):
    patches = []
    h, w = img.shape
    for i in range(0, h, ps[0]):
        for j in range(0, w, ps[1]):
            patches.append(img[i:i+ps[0], j:j+ps[1]].ravel())
    return np.array(patches)


def depatchify(patches, patch_size=(10, 10), img_size=(350, 780)):
    # normalize
    patches[patches > 255.] = 255.
    patches[patches < 0.] = 0.

    # convert to unint8
    patches = patches.astype('uint8')
    rec_img = np.zeros(img_size, dtype='uint8')
    ph, pw = patch_size

    h, w = img_size
    x = 0
    for i in range(0, h, ph):
        for j in range(0, w, pw):
            rec_img[i:i+ph, j:j+pw] = patches[x].reshape((ph, pw))
            x += 1
```

```
    return rec_img

def convert_to_cf(img_cl):
    # convert image to channel first
    img_cf = np.swapaxes(img_cl.T, 1, 2)
    return img_cf

image=imread('leena.jpg')
img_cf = convert_to_cf(image)

# I'll patchify each channel and depatchify them
# Then I'll stack them together to create original image back
ch1 = depatchify(patchify(img_cf[0])) # first channel
ch2 = depatchify(patchify(img_cf[1])) # second channel
ch3 = depatchify(patchify(img_cf[2])) # third channel

# combine them now
rec_img = np.dstack((ch1, ch2, ch3))
plt.figure(figsize=(12, 12))
show_image(rec_img)
```

STEP 3: Compressing single channel of an image (FILL TWO FUNCTIONS BELOW)

Now you are familiar with how `patchify` and `depatchify` work. Do the following:

1. Write a function `compress` that will take one of the channel of the image as input and outputs compressed channel and auxiliary data required for decompressing. Return type of this function should be dictionary.
2. Write another function `decompress` that will take whatever dictionary data you returned from previous `compress` function and decompresses it into image channel (that was compressed).
3. PCA compression is lossy compression algorithm -- means you will loose the information during decompression.
4. I should be able to call two of your functions like so `decompress(compress(img_ch[0]))` to compress and decompress the given image's channel.

Compress: Pseudo code

- `Patchify` the given image's channel (input).
- Run `PCA` on the patches (you may use sklearn's implementation) to reduce them to `5d` vectors.
- You will need `basis vectors` or `principal components` and `mean` in order to reconstruct the data back to `100d`
- Return dictionary: `{'compressed_patches': 5d vectors, 'aux_data': principal components/basis vectors/means/final size of image}`
- By converting `100d` to `5d`, you reduce size by `20` times.

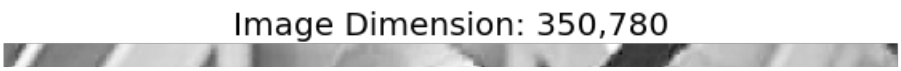
Decompress: Pseudo code

- Input is dictionary as returned by your `compress` function.
- Use `5d` vectors and do inverse PCA (check tutorial 2) and convert them back to `100d` vectors.
- Use `depatchify` function to convert these reconstructed `100d` vectors into an image channel

Tip: Good code is always modular and easy to read.

In [12]:

```
# COMPLETE FOLLOWING FUNCTIONS

def compress(img_ch, n_components=5):
    """
    Inputs
        img_ch: one of the channel of given image
        n_components: number of components returned by PCA
    Returns
        comp_data: Some data structure (may be dict.) that represents compressed form of given
input
        along with auxiliary data required for decompression (components_, mean_ and shape of
input image)
    """
    # patchify img_ch
    data = patchify(img_ch)

    # Fit img to PCA curve
    pca = PCA(n_components=n_components)
    pca.fit(data)

    compressed_data = np.dot(data - pca.mean_, pca.components_.T)
    aux_data = (pca.components_, pca.mean_, img_ch.shape)

    return {'y': compressed_data, 'aux_data': aux_data}


def decompress(comp_data):
    """
    Inputs
        comp_data: data structure that is returned by `compress` function
    Returns
        img_ch: decompressed form of channel compressed and contained inside `comp_data` data
structure
    """
    y = comp_data['y']
    components, means, img_size = comp_data['aux_data']

    # recontruct the patches to 100d using aux_data and inverse PCA
    # depatchify and return img_ch
    decompressed_data = np.dot(y, components)+means

    return depatchify(decompressed_data)

# Red channel
# visualize your compression and decompression
img_ch = img_cf[0]
plt.figure(figsize=(10, 10))
show_image(img_ch)

plt.figure(figsize=(10, 10))
show_image(decompress(compress(img_ch, n_components=5)))
```



STEP 3: Compress and decompress entire image (FILL TWO MORE FUNCTIONS)

Write a `compress_image` function that:

- takes `channel last` (regular image read from `imread`) representation of an image
- convert `channel last` to `channel first` representation using `convert_to_cf` function defined previously
- compresses each of the channel using `compress` function
- outputs a one dictionary that contains all auxiliary data required to reconstruct/decompress entire image back.

Similarly, write `decompress_image` function that:

- decompresses the image compressed by `compress_image` function
- return `channel last` image

NOTE: You would patchify each channel and implement PCA on each channel thus for decompression you need `components_`, `mean_` for each channel and you need shape of input image as well.

In [13]:

```
def compress_image(img):
    """
    Inputs:
        img_cf: `Channel last` image data
    output: A list of channel information.
    """
    #  Returning a list of dict because its a more intutitive data structure than the instruction
    return [compress(ch) for ch in convert_to_cf(img)]

def decompress_image(comp_img):
    """
    Returns:
        img_rec: Decompressed `channel last` image
    """
    ch = [decompress(comp_ch) for comp_ch in comp_img]
    rec_img = np.dstack(ch)
    return rec_img
```

STEP 4: Serialization and de-serialization

You can easily (de)serialize dictionary using `np.save` and `np.load` (see example below)

`compress_and_serialize` should:

- Read image given by `inp_path`
- Use `compress_image` to compress it
- Serialize the compressed data using `np.save` to a file specified by `out_path`

NOTE: All the data required for deserialization must be saved to a one single file only.

`deserialize_and_decompress` should:

- Read the file specified by `inp_path` using `np.load`
- Use `decompress_image` function to decompress it
- Return image (channel last as returned by `decompress image`) function

In [14]:

```
# EXAMPLE OF SAVING AND LOADING DICTIONARY OBJECT TO/FROM FILESYSTEM
d = {'a': [1, 2, 3], 'b': [4, 5, 6, 7, 8]}
np.save('example.npy', d)
ds = np.load('example.npy').item()

ds
```

Out[14]:

```
{'a': [1, 2, 3], 'b': [4, 5, 6, 7, 8]}
```

In [15]:

```
# COMPLETE THESE TWO FUNCTIONS
def compress_and_serialize(inp_path='leena.jpg', out_path='output.bin'):
    comp_img = compress_image(imread(inp_path))
    np.save(out_path, comp_img)

def deserialize_and_decompress(inp_path='output.bin.npy'):
    return decompress_image(np.load(inp_path))
```

STEP 5: What is size of output.bin file?

Did you end of in compressing anything? Conclude your experiments!

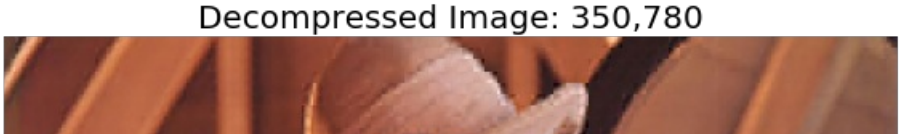
Conclusion

Write your conclusion here!!

The decompressed image is not an 100 percent restoration, its a bit blurred as seemed below.

In [16]:

```
import os, sys

img_name = 'leena.jpg'
img = imread(img_name)
compress_and_serialize()
compressed_img = compress_image(img)

plt.figure(figsize=(10, 10))
show_image(img)
plt.figure(figsize=(10, 10))
show_image(decompress_image(compressed_img), title='Decompressed Image')
```

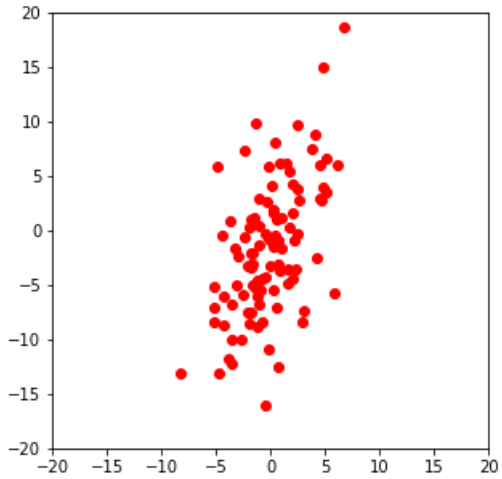


Task 2: Rotation and Translation Invariance in PCA

STEP 1 (done already): Create normally distributed data

In [17]:

```
mean = [0, 0]
cov = [[10, 10], [10, 40]]  # diagonal covariance
x, y = np.random.multivariate_normal(mean, cov, 100).T
X = np.array(list(zip(x, y)))


def plot_data(X):
    plt.figure(figsize=(5, 5))
    plt.plot(X[:, 0], X[:, 1], 'ro')
    plt.xlim([-20, 20])
    plt.ylim([-20, 20])
    plt.gca().set_aspect('equal', adjustable='box')
    plt.show()

plot_data(X)
```



STEP 2: Rotate the data by 45 degrees and create new array X_rot

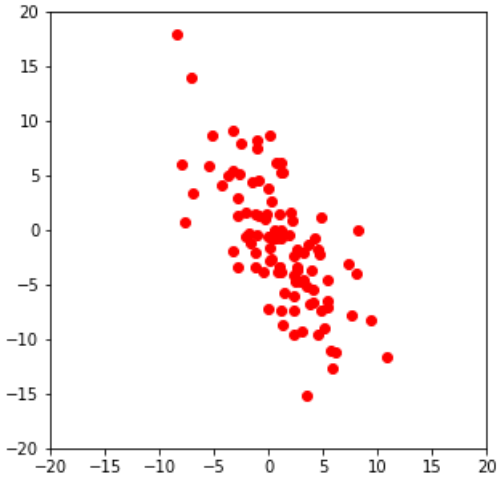
Write code to rotate X by 45

- Use rotation matrix
- Or individually rotate each point in `X` by 45 degrees
- Use `plot_data` function defined in cell above to visualize the rotated data

In [18]:

```
theta = np.radians(45)
# WRITE CODE HERE
# Code to rotate X
'''
[x', y'] = Rotational Matrix * [x, y]
'''
```

```
c, s = np.cos(theta), np.sin(theta)
R = np.array(((c,-s), (s, c)))

X_rot = np.array([np.dot(R, p) for p in X])

plot_data(X_rot)
```



STEP 3 (done but need explanation): Perform pca with n_components=2

Do you see anything interesting?

Explain the code below and write down your observations (2-3 lines only).

Observations

The PCA transoformation of both of the data set are the eact same, even though X_rot is the rotational of X.

In [19]:

```
import pandas as pd

def visualize_components(X, X_rot):
    pca = PCA(n_components=2)

    pca.fit(X)
    df1 = pd.DataFrame(pca.fit_transform(X))
    df2 = pd.DataFrame(pca.fit_transform(X_rot))

    return pd.concat((df1, df2), axis=1)[:20]

visualize_components(X, X_rot)
```

Out[19]:

	0	1	0	1
0	-5.189982	0.602627	-5.189982	0.602627
1	-1.888897	-1.099352	-1.888897	-1.099352
2	-2.081813	2.863682	-2.081813	2.863682
3	-1.872430	-1.042124	-1.872430	-1.042124
4	-5.089427	2.275515	-5.089427	2.275515
5	4.066129	-2.181044	4.066129	-2.181044
6	-10.816238	-0.635995	-10.816238	-0.635995
7	0.049244	-0.950788	0.049244	-0.950788
8	-10.194785	3.930871	-10.194785	3.930871
9	-2.588219	0.389869	-2.588219	0.389869
10	-1.246157	1.203224	-1.246157	1.203224
11	11.632216	-0.676502	11.632216	-0.676502
12	0.312113	0.342299	0.312113	0.342299

	0	1	0	1
13	-3.883421	0.305770	-3.883421	0.305770
14	-12.304085	-1.239035	-12.304085	-1.239035
15	17.311463	0.126306	17.311463	0.126306
16	-5.385357	-2.742851	-5.385357	-2.742851
17	-7.958162	-2.895602	-7.958162	-2.895602
18	-2.163348	-1.408055	-2.163348	-1.408055
19	-2.609474	2.649517	-2.609474	2.649517

STEP 4: Perform PCA again and find angle between principal components

- Perform PCA on X and X_rot with n_components=1
- It will give one basis vector for each of data (X and X_rot)
- Find the angle between these two basis vectors
- Explain your observations?

Observations

The angle between the two basis vectors is 45 degree, which is the rotational angle applied earlier. The rotation of the data caused the principle component to rotate as well, because the principle component for an axis is the best fit vector for a set of data. Therefore, when a set of data rotates, it rotates with the data.

In [20]:

```
from numpy.linalg import norm

def angle_between(a,b):
    # COMPLETE THIS FUNCTION
    # CALCULATE ANGLE IN RAD BETWEEN TWO VECTORS
    '''
    Based on the dot product theory that:
    A dot B = ||A||*||B||*cos(theta)
    => theta = arccos((A dot B)/(||A||*||B||))
    '''
    a_mag = np.linalg.norm(a)
    b_mag = np.linalg.norm(b)
    return np.arccos((np.dot(a, b))/(a_mag*b_mag))

pca = PCA(n_components=1)
pca.fit(X)
c1 = pca.components_

pca.fit(X_rot)
c2 = pca.components_

np.rad2deg(angle_between(c1[0], c2[0]))
```

Out[20]:

```
45.0
```

(NO MARKS) STEP 5: Repeat these experiments for translation as well

There is not marks for this part. You can do this for your own learning.

Now translate every point in `X` by fixed `x` and `y` amount.

```
X = X + [1, 2]
```

like so and repeat all the above experiments in cell below and write down your observations:

In [21]:

```
# PERFORM EXPERIMENTS WITH TRANSLATION HERE
# NO MARKS FOR DOING THIS
```

TASK 3: Recovery of corrupted images using PCA

Check the code below.

It corrupts the `leena.jpg` image that you worked on before.

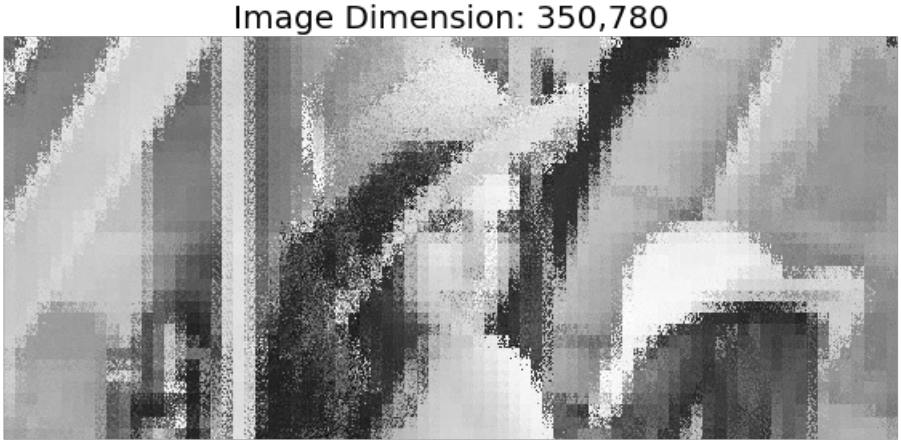
Check very carefully what below code is doing on the image and answer:

Is it same as rotation of data points (like done before), if yes explain (just one liner)?

No, the actual data is the same, they are just placed in different order.

In [22]:

```
# Code for corrupting the leena image
image = convert_to_cf(imread('leena.jpg'))[0]

def corrupt_image(img):
    # lets patchify R channel with patches of 35x78 patches
    patches = patchify(img)

    # Noise 1:
    # lets jumble pixels of patches now
    jumble_idx = list(range(len(patches[0])))
    np.random.shuffle(jumble_idx)

    jumbled_patches = np.array([patch[jumble_idx] for patch in patches])
    rec_jumbled_image = depatchify(jumbled_patches, img_size=img.shape)

    return rec_jumbled_image


cimage = corrupt_image(image)

plt.figure(figsize=(10,10))
show_image(image)

plt.figure(figsize=(10,10))
show_image(cimage)
```



Recovering from the corruption

Use the `compress` function you coded earlier to get compressed form of original leena image and corrupted leena image.

We will try to recover the corrupted leena given the compressed version of original leena.

Notice in code below, I replace `aux_data` of corrupted version with `aux_data` of original version.

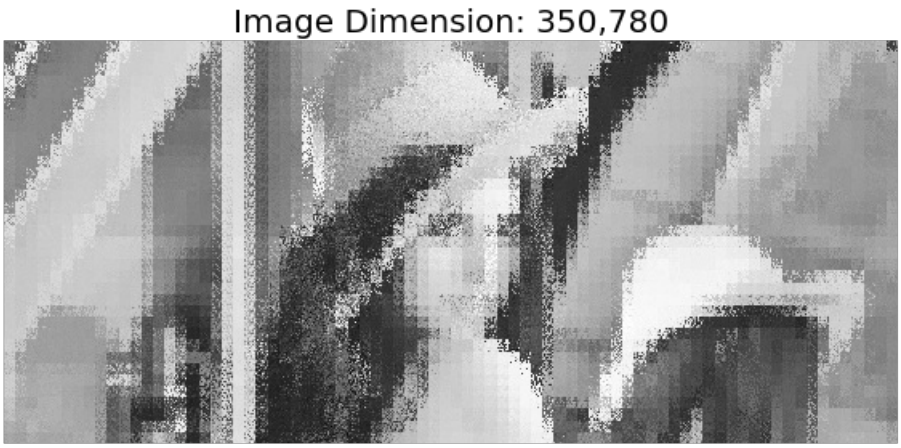
Does the code below code work in recovering the original leena image back?

Explain how below code works and show it actually works?

The underlying data remained the same so the means remained the same, however the principle component was corrupted in depatchify. The corrupted princniple component does not acutally apply to the data, that's why when the original component was applied, it fixed the image.

In [23]:

```
# This assuming your compress function returns dictionary which contains `aux_data`
# if you coded your compress in diferent way then change code below appropriately
# basically u need to replace auxiliary data of corrupted with original while keeping compressed (
transformed) data same
d = compress(image, n_components=50)
d1 = compress(cimage, n_components=50)

plt.figure(figsize=(10, 10))
show_image(decompress(d1))

d1['aux_data'] = d['aux_data']

plt.figure(figsize=(10, 10))
show_image(decompress(d1))
```



Task 4: Decrypting manuscript of the lost civilization

You belong to one of the advanced civilization, while exploring the universe you land on the planet Earth. However, there is no inhabitants on the planet anymore. While exploring Earth, you come across some damaged hard drive that contains various images. You suspect that these images form a manuscript of how "humans" use to write different digits in maths. You recovered all the data from the hard drive safely. You opened up your jupyter notebook (python being universal language and popular among alien species), you started plotting the images you just recovered.

In [25]:

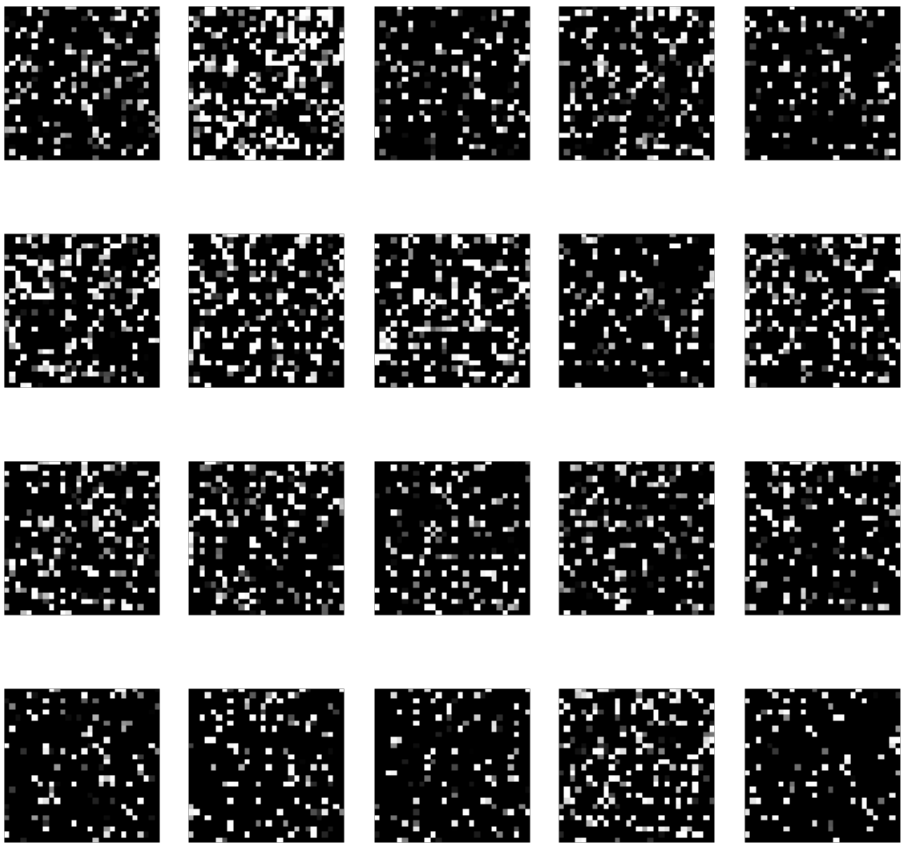
```
rimages = np.load('recovered_images.npy')
rimages.shape
```

Out[25]:

```
(20, 28, 28)
```

In [26]:

```
def plot_manuscript(images):
    for i, img in enumerate(images):
        plt.subplot(4, 5, i+1)
        plt.imshow(img, cmap='Greys_r')
        plt.axis('off')

plt.figure(figsize=(10, 10))
plot_manuscript(rimages)
```



You being a smart alient, figured out that these images are encrypted and not in their original form. You also figured out that encryption is just fixed jumbling of the pixels of the images. Since you're unaware of the manuscript, you cannot decrypt the images just by themselves even if they follow the fixed jumbling pattern. However, fortunately you found the `principal components` and `mean` of the the original manuscript somewhere in the same harddisk. Now, your task is to recover all the 20 images and plot them nicely in cell below.

Use `plot_manuscript` function from above to plot the recovered manuscript.

In [27]:

```
original_components = np.load('original_pca.npy')
original_mean = np.load('original_mean.npy')

# Convert images to 1D from 2D
rimages_1d = [img.ravel() for img in rimages]

# Fit images with as many components as the original componets and compress it
pca = PCA(n_components = 20)
pca.fit(rimages_1d)
reduced_data = pca.transform(rimages_1d)

# Restore data using original components and means similar to Task 3
recovered_imgs_1d = np.dot(reduced_data, original_components) + original_mean

# Convert images back to 2D
recovered_images_2d = [np.reshape(img, (-1, 28)) for img in recovered_imgs_1d]

plt.figure(figsize=(10, 10))
plot_manuscript(recovered_images_2d)
```