exercise 3

simple proofs, fastmap, and Hopfield nets

solutions due

until **May 30, 2021** at **23:59** via **email**

students handing in this solution set

last name	first name	student ID	enrolled with
Nikolskyy	Oleksander	TODO	TODO
Schier	Marie	TODO	TODO
Doll	Niclas	3075509	Uni Bonn
Safavi	Arash	TODO	TODO
Wani	Mohamad Saalim	TODO	TODO
Bonani	Mayara Everlim	TODO	TODO

general remarks

As you know, your instructor is an avid proponent of open science and education. Therefore, **MATLAB implementations will not be accepted** in this course.

The goal of this exercise is to get used to scientific Python. There are numerous resources on the web related to Python programming. Numpy and Scipy are well documented and Matplotlib, too, comes with numerous tutorials. Play with the code that is provided. Most of the above tasks are trivial to solve, just look around for ideas as to how it can be done.

Remember that you have to achieve at least 50% of the points of the exercises to be eligible to the written exam at the end of the semester. Your grades (and credits) for this course will be decided based on the exam only, but –once again– you have to succeed in the exercises to get there.

Your solutions have to be *satisfactory* to count as a success. Your code and results will be checked and need to be convincing.

If your solutions meets the above requirements and you can demonstrate that they work in practice, it is a *satisfactory* solution.

A *very good* solution (one that is rewarded full points) requires additional efforts especially w.r.t. to readability of your code. If your code is neither commented nor well structured, your solution is not good! The same holds for your discussion of your results: these should be concise and convincing and demonstrate that you understood what the respective task was all about. Striving for very good solutions should always be your goal!

practical advice

The problem specifications you'll find below assume that you use python / numpy / scipy for your implementations. They also assume that you have imported the following

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt
```

task 3.1 [5 points]

whitened data is of unit covariance

In lecture 08, we briefly looked at data pre-processing techniques such as *data whitening*. Recall that, if we are given an $m \times n$ data matrix of real valued vectors

$$\boldsymbol{X} = \begin{bmatrix} | & | & & | \\ \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_n \\ | & | & & | \end{bmatrix}$$

then the process of whitening these data consists of the following three steps

1. compute the transformation
$$\boldsymbol{Y} = \boldsymbol{X}\left(\boldsymbol{I} - \tfrac{1}{n}\,\mathbf{1}\mathbf{1}^\intercal\right)$$
2. determine the spectral decomposition
$$\tfrac{1}{n}\,\boldsymbol{Y}\boldsymbol{Y}^\intercal = \boldsymbol{U}\boldsymbol{\Lambda}\,\boldsymbol{U}^\intercal$$
3. compute the transformation
$$\boldsymbol{Z} = \boldsymbol{U}\,\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^\intercal\boldsymbol{Y}$$

Also recall that we claimed the covariance matrix of the data in $\boldsymbol{Z}$ to be the identity matrix. Your task is now to prove this claim, i.e. prove that

$$\frac{1}{n}\,\boldsymbol{Z}\boldsymbol{Z}^\intercal = \boldsymbol{I}$$

(Remember that $\boldsymbol{U}$ is orthogonal and $\boldsymbol{\Lambda}$ is diagonal . . .)

$$\begin{aligned}
\boldsymbol{Z}\boldsymbol{Z}^\intercal &= (\boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T\boldsymbol{Y})(\boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T\boldsymbol{Y})^T \\
&= \boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T(\boldsymbol{Y}\boldsymbol{Y}^T)\boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T \\
&= \boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T(n \cdot \frac{1}{n}\boldsymbol{Y}\boldsymbol{Y}^T)\boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T \\
&= \boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T(n \cdot \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T)\boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T \\
&= n \cdot \boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T \\
&= n \cdot \boldsymbol{U}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{\Lambda}\boldsymbol{\Lambda}^{-\frac{1}{2}}\boldsymbol{U}^T \\
&= n \cdot \boldsymbol{U}\boldsymbol{U}^T \\
&= n \cdot \boldsymbol{I}
\end{aligned}$$

task 3.2 [5 points]

more on idempotent matrices

In lecture 07, we already proved that, if $\boldsymbol{u} \in \mathbb{R}^m$ is a unit vector, then the outer product matrix $\boldsymbol{P} = \boldsymbol{u}\boldsymbol{u}^\intercal$ is idempotent. Your task is now to establish a more general result.

Prove that, for any $\boldsymbol{u} \neq \boldsymbol{0} \in \mathbb{C}^m$, the matrix

$$\boldsymbol{P} = \frac{\boldsymbol{u}\boldsymbol{u}^\dagger}{\boldsymbol{u}^\dagger\boldsymbol{u}}$$

is idempotent.

$$\boldsymbol{P}^2 = \left(\frac{\boldsymbol{u}\boldsymbol{u}^\dagger}{\boldsymbol{u}^\dagger\boldsymbol{u}}\right)\left(\frac{\boldsymbol{u}\boldsymbol{u}^\dagger}{\boldsymbol{u}^\dagger\boldsymbol{u}}\right) = \frac{\boldsymbol{u}(\boldsymbol{u}^\dagger\boldsymbol{u})\boldsymbol{u}^\dagger}{(\boldsymbol{u}^\dagger\boldsymbol{u})^2} = \frac{\boldsymbol{u}\boldsymbol{u}^\dagger}{\boldsymbol{u}^\dagger\boldsymbol{u}} = \boldsymbol{P}$$

task 3.3 [10 points]

fastmap

In the `Data` folder for this exercise, you will find the file

`faceMatrix.npy`

which contains a data matrix $\boldsymbol{X} \in \mathbb{R}^{361\times 2429}$ whose columns $\boldsymbol{x}_i$ represent tiny face images of size 19×19 pixels. To read this matrix into memory and check its size, you may use

```
matX = np.load('faceMatrix.npy').astype('float')
m, n = matX.shape
print (m, n)
```

To have a look at one of the images it contains, say $\boldsymbol{x}_{15}$, you may use this

```
vecX = matX[:,14].reshape(19,19)
plt.imshow(vecX, cmap='gray')
plt.xticks([])
plt.yticks([])
plt.show()
```

Having read matrix $\boldsymbol{X}$ into memory, here is what you are supposed to do:

1. Implement the *fastmap* algorithm which we discussed in lecture 07. Recall that this algorithm involves a function GETDISTANTOBJECTS. How would you implement it in numpy / scipy?

```
# helper function to compute the squared norm of vectors
sq_norm = lambda X: (X ** 2).sum(axis=0)
# function implementing fastmap
def fastmap(
    X:np.ndarray,
    k:int
) -> np.ndarray:
    m, n = X.shape
    # list of the computed approximations
    V = []
    # do the iterations
    for _ in range(k):
        # find the two most distant object indices
        i = np.random.choice(n, size=1)[0]
        j = sq_norm(X - X[:, i:i+1]).argmax()
        i = sq_norm(X - X[:, j:j+1]).argmax()
        j = sq_norm(X - X[:, i:i+1]).argmax()
```

```
        # approximate the major component of the current matrix X
        v = (X[:, j:j+1] - X[:, i:i+1])
        v /= la.norm(v, axis=0)
        # add the major component to the list
        V.append(v)
        # project vectors into subspace
        X = (np.eye(m) - v @ v.T) @ X
    # stack all approximations and return
    return np.concatenate(V, axis=1)
```

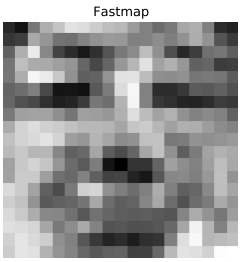
2. Run your *fastmap* implementation to obtain approximations of the top 25 principal components of the data in matrix $\boldsymbol{X}$.

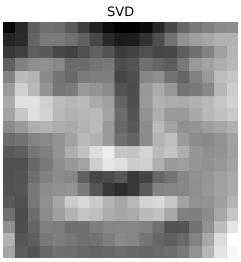
```
# compute approximations of major components using fastmap
V = fastmap(X, k=25)
```

3. Compute the SVD of matrix $\boldsymbol{X}$ and determine the top 25 left singular vectors.

```
# compute the left singular vectors of the data matrix using svd
U, _, _ = la.svd(X)
# get only the first 25 singular vectors and ignore the rest
# note that the singular values are sorted in non-increasing order
# thus the first k singular vectors are the first k major components
U = U[:, :25]
```

4. Compare your *fastmap* results to your SVD results. A good idea for doing this is to visualize / plot them as tiny images; figure out how to create a plot that shows not just a single tiny image but 25 at the same time





task 3.4 [10 points]

Hopfield nets: the k-rooks problem

In lecture 10, we introduced the k-rooks problem and saw how to solve it using a Hopfield network. In particular, we considered the special case where $k = 4$. Building on our discussion, here is what you are supposed to do in this task:

```
# this is pretty much the exact code given in the lecture
# define the signum function
signum = lambda x: np.where(x >= 0, +1, -1)
# run the asynchronous hopfield network updates for some number of steps
def hopfield_run_async(s, W, theta, tmax=1000):
    for u in np.random.randint(0, s.shape[0], tmax):
        s[u] = signum(W[u, :] @ s - theta[u])
    return s
```

1. Use what we discussed in the lecture to implement a Hopfield net (with asynchronous updates) that solves the $k = 4$ rooks problem; run your Hopfield net for, say, 100 iterations

```
# set the dimensions
k = 4
m = k * k
# build the hopfield network weight matrix
# as a block matrix consisting of J and I
I = np.eye(k)
J = np.full((k, k), 1.) - I
W = -0.5 * np.block([
    [J, I, I, I],
    [I, J, I, I],
    [I, I, J, I],
    [I, I, I, J]
])
# create the threshold vector
theta = np.full(m, (k - 2))
# create a random initial state
z = np.random.uniform(0, 1, size=m) >= 0.5
s = 2 * z - 2
# do a few hopfield state updates
s = hopfield_run_async(s, W, theta, tmax=100)
```

2. Now implement a Hopfield net that solves the $k = 8$ rooks problem; to this end, you need to redefine or recompute the network parameters $\boldsymbol{W}_r$, $\boldsymbol{W}_c$, θ_r, and θ_c; the team with the "most elegant" numpy code to set these parameters will receive a honorary mention ;-)

```
# set the dimension to 8 but the code fragment also
# works for arbitrary dimensions
k = 8
m = k * k
# 4-rooks problem
```

```
# build the weight matrix
# this works for arbitrary k
I = np.eye(k)
J = np.full((k, k), 1.) - I
W = -0.5 * np.block([
    [I] * i + [J] + [I] * (k - i - 1)
    for i in range(k)
])
# build the threshold vector
theta = np.full(m, (k - 2))
# create a random initial state
s = np.random.uniform(0, 1, size=m) >= 0.5
s = 2 * s - 1
# do a few hopfield state updates
s = hopfield_run_async(s, W, theta, tmax=100)
```

3. Run your Hopfield net for, say 100 iterations and try to visualize the state it ends up in

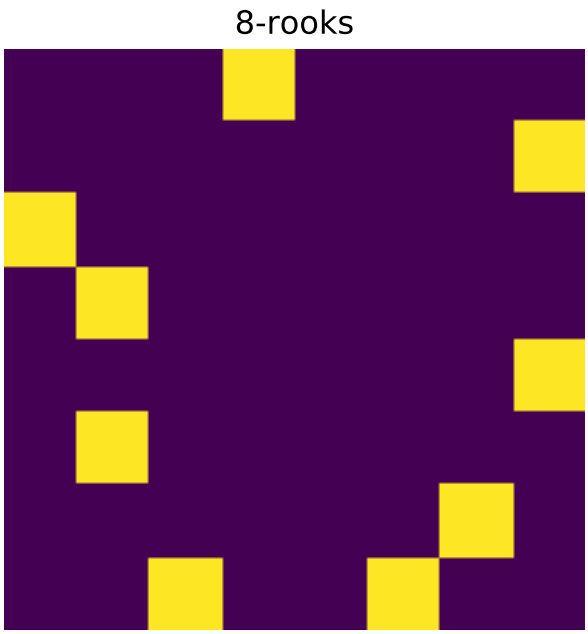


Figure 1: Final state after 100 ansynchronus update steps.

task 3.5 [20 points]

Hopfield nets: finding maximally different images

Once again consider the matrix $\boldsymbol{X}$ of tiny images from task 3.3. However, let us simplify things and only consider the first 100 of these images. Using numpy, we can accomplish this like so

```
matX = matX[:,:100]
```

Given this truncated version of matrix $\boldsymbol{X}$, standardize the data it contains and store the result in a matrix $\boldsymbol{Y}$.

Next, given this new matrix $\boldsymbol{Y}$, compute a squared Euclidean distance matrix $\boldsymbol{D}^{n\times n}$ where, in our case $n = 100$ and

$$D_{ij} = \left\|\boldsymbol{y}_i - \boldsymbol{y}_j\right\|^2.$$

If you do not know how to do this efficiently, you might want to consult

> C. Bauckhage, **"NumPy / SciPy Recipes for Data Science: Squared Euclidean Distance Matrices"**, technical report, 2014

Now, recall that, in exercise 01, we were concerned with the problem of finding k data points that are as far apart as possible. Back then, we formalized this problem as follows

$$\begin{aligned} S^* = \underset{S\subset X}{\operatorname{argmax}} & \sum_{\boldsymbol{y}_i\in S}\sum_{\boldsymbol{y}_j\in S}\left\|\boldsymbol{y}_i - \boldsymbol{y}_j\right\|^2 \\ \text{s.t.} \quad & |S| = k \end{aligned}$$

Working with the above distance matrix $\boldsymbol{D}$ and introducing a binary indicator vector $\boldsymbol{z} \in \{0,1\}^n$, this problem can also be cast as

$$\begin{aligned} \boldsymbol{z}^* = \underset{\boldsymbol{z}\in\{0,1\}^n}{\operatorname{argmax}} & \ \boldsymbol{z}^\mathsf{T}\boldsymbol{D}\boldsymbol{z} \\ \text{s.t.} \quad & \mathbf{1}^\mathsf{T}\boldsymbol{z} = k \end{aligned} \tag{$\star$}$$

This looks like a problem a Hopfield network could handle! So, go ahead and implement a Hopfield network that solves $(\star)$.

Note: $(\star)$ is written in terms of binary vectors $\boldsymbol{z}$. However, for the Hopfield networks we are interested in, we require problem formulations in terms of bipolar vectors $\boldsymbol{s}$. That is, you'll need to rewrite the problem in $(\star)$ as a QUBO that *minimizes* over bipolar vectors (and make sure that the resulting weight matrix is hollow) before you implement the Hopfield net. (This is possible . . . all you need to do is to be very careful and diligent with your algebraic manipulations).

Run your Hopfield net for several choices of k, say $k \in \{9, 16, 25\}$.

Note: Your Hopfield net will end up in a state $\boldsymbol{s} \in \{-1, +1\}^n$. *If all goes well*, k entries of $\boldsymbol{s}$ will have a value of $+1$ and the remaining $n - k$ entries will have a value of -1. The $+1$ entries of $\boldsymbol{s}$ will index k maximally diverse columns of $\boldsymbol{Y}$. However, $\boldsymbol{Y}$ is a standardized version of $\boldsymbol{X}$. Hence, to visualize the outcome of your Hopfield network computations, use the resulting state vector $\boldsymbol{s}$ as an index into the columns of $\boldsymbol{X}$ and visualize the images you obtain this way.

Lets first derive the weights of the hopfield network for this specific optimization problem.

Start by rewriting the initial quadratic program as follows

$$\begin{aligned} \Rightarrow \boldsymbol{z}^* = \underset{\boldsymbol{z}\in\{0,1\}^n}{\operatorname{argmin}} \; & -\boldsymbol{z}^\intercal \boldsymbol{D}\boldsymbol{z} \\ \text{s.t.} \quad & (\mathbf{1}^\intercal \boldsymbol{z} - k)^2 = 0 \end{aligned}$$

Then convert from binary valued vectors to bipolar vectors using the isomorphic transformation $s = \frac{1}{2}(z + 1)$

$$\begin{aligned} \Rightarrow \boldsymbol{s}^* = \underset{\boldsymbol{s}\in\{-1,+1\}^n}{\operatorname{argmin}} \; & -\frac{1}{4}(\boldsymbol{s}+\mathbf{1})^\intercal \boldsymbol{D}(\boldsymbol{s}+\mathbf{1}) \\ \text{s.t.} \quad & (\frac{1}{2}\mathbf{1}^\intercal(\boldsymbol{s}+\mathbf{1}) - k)^2 = 0 \end{aligned}$$

$$\begin{aligned} (\boldsymbol{s}+\mathbf{1})^\intercal \boldsymbol{D}(\boldsymbol{s}+\mathbf{1}) &= \boldsymbol{s}^\intercal \boldsymbol{D}\boldsymbol{s} + 2\cdot\mathbf{1}^\intercal \boldsymbol{D}\boldsymbol{s} + \mathbf{1}^\intercal \boldsymbol{D}\mathbf{1} \\ \frac{1}{2}\mathbf{1}^\intercal(\boldsymbol{s}+\mathbf{1}) - k &= \frac{1}{2}(\mathbf{1}^\intercal \boldsymbol{s} + \mathbf{1}^\intercal\mathbf{1}) - k \\ &= \frac{1}{2}(\mathbf{1}^\intercal \boldsymbol{s} + n - 2k) \\ &= \frac{1}{2}(\mathbf{1}^\intercal \boldsymbol{s} + c) \quad \text{for } c = n - 2k \end{aligned}$$

$$\Rightarrow \boldsymbol{s}^* = \underset{\boldsymbol{s}\in\{-1,+1\}^n}{\operatorname{argmin}} \; -\frac{1}{4}\boldsymbol{s}^\intercal \boldsymbol{D}\boldsymbol{s} - \frac{1}{2}\cdot \mathbf{1}^\intercal \boldsymbol{D}\boldsymbol{s}$$
$$\text{s.t.} \quad \frac{1}{4}(\mathbf{1}^\intercal \boldsymbol{s} + c)^2 = 0$$

Now using Lagrange-Multipliers this can be rewritten as an unconstraint optimization problem

$$\begin{aligned}
\Rightarrow L(s,\lambda) &= -\frac{1}{4}\boldsymbol{s}^\intercal \boldsymbol{D}\boldsymbol{s} - \frac{1}{2}\cdot \mathbf{1}^\intercal \boldsymbol{D}\boldsymbol{s} + \frac{1}{4}\lambda(\mathbf{1}^\intercal \boldsymbol{s} + c)^2 \\
&= -\frac{1}{4}\boldsymbol{s}^\intercal \boldsymbol{D}\boldsymbol{s} - \frac{1}{2}\cdot \mathbf{1}^\intercal \boldsymbol{D}\boldsymbol{s} + \frac{1}{4}\lambda(\boldsymbol{s}^\intercal \mathbf{1}\mathbf{1}^\intercal \boldsymbol{s} + 2c\mathbf{1}^\intercal \boldsymbol{s} + c^2) \\
&= \frac{1}{4}\boldsymbol{s}^\intercal(\lambda \mathbf{1}\mathbf{1}^\intercal - \boldsymbol{D})\boldsymbol{s} + \frac{1}{2}\cdot(\lambda c\mathbf{1}^\intercal - \mathbf{1}^\intercal \boldsymbol{D})\boldsymbol{s} + \frac{1}{4}\lambda c^2 \\
&= \frac{1}{4}\boldsymbol{s}^\intercal(\lambda \mathbf{1}\mathbf{1}^\intercal - \boldsymbol{D})\boldsymbol{s} + \frac{1}{2}\cdot(\lambda c\mathbf{1} - \boldsymbol{D}^\intercal \mathbf{1})^\intercal \boldsymbol{s} + \frac{1}{4}\lambda c^2
\end{aligned}$$

Lastly the coefficient matrix needs to be hollow, which can be achived as follows

$$\begin{aligned}
\Rightarrow L(s,\lambda) &= \frac{1}{4}\boldsymbol{s}^\intercal(\lambda \mathbf{1}\mathbf{1}^\intercal - \boldsymbol{D} - \lambda \boldsymbol{I})\boldsymbol{s} + \boldsymbol{s}^\intercal \lambda \boldsymbol{I}\boldsymbol{s} + \frac{1}{2}\cdot(\lambda c\mathbf{1} - \boldsymbol{D}^\intercal \mathbf{1})^\intercal \boldsymbol{s} + \frac{1}{4}\lambda c^2 \\
&= \frac{1}{4}\boldsymbol{s}^\intercal(\lambda \mathbf{1}\mathbf{1}^\intercal - \boldsymbol{D} - \lambda \boldsymbol{I})\boldsymbol{s} + \frac{1}{2}\cdot(\lambda c\mathbf{1} - \boldsymbol{D}^\intercal \mathbf{1})^\intercal \boldsymbol{s} + \lambda n + \frac{1}{4}\lambda c^2
\end{aligned}$$

Note that the distance matrix $\boldsymbol{D}$ is already hollow by definition.

This final form of the optimization problem can be directly optimized using a hopfield network with

$$\boldsymbol{W} = -\frac{1}{2}(\lambda \mathbf{1}\mathbf{1}^\intercal - D - \lambda \boldsymbol{I})$$
$$\boldsymbol{\theta} = \frac{1}{2}(\lambda c\mathbf{1} - \boldsymbol{D}^\intercal \mathbf{1})$$

$$\Rightarrow s^* = \underset{\boldsymbol{s}\in\{-1,+1\}^n}{\operatorname{argmin}} \; -\frac{1}{2}\boldsymbol{s}^\intercal \boldsymbol{W}\boldsymbol{s} + \boldsymbol{\theta}^\intercal \boldsymbol{s}$$

```
# load and preprocess data
X = np.load('Data/faceMatrix.npy').astype('float')
X = X[:, :100]
Y = X - X.mean(axis=-1, keepdims=True)
# also convert the data matrix to the actual image
# layout for easier visualization later on
imgs = X.reshape(19, 19, -1).transpose(2, 0, 1)

# compute the distance matrix using method 5 from
# "NumPy/SciPy Recipes for Data Science:Squared Euclidean Distance Matrices"
def squared_EDM(X:np.ndarray):
    m, n = X.shape
    # compute gram matrix
    G = X.T @ X
    # compute matrix H
    H = np.tile(np.diag(G), (n, 1))
    return H + H.T - 2*G
D = np.sqrt(squared_EDM(Y))

def maximally_different(
    D:np.ndarray,
    k:int,
    lam:float =7000     # we handpicked lambda by try and error :)
):
    # get the number of objects to choose from
    n = D.shape[0]
    # build hopfield network parameters that were derived above
    W = -0.5 * (np.full((n, n), lam) - lam * np.eye(n) - D)
    theta = 0.5 * (np.full(n, lam * (n - 2*k)) - D.sum(axis=-1))
    # create an initial state
    s = np.ones(n)

    # apply hopfield optimization
    s = hopfield_run_async(s, W, theta, tmax=100_000)
    # create mask indicating maximally different objects
    return (s == 1.0)

# helper function for easily visualization
# of the found candidates
def show_maximally_different(
    imgs:np.ndarray,
    D:np.ndarray,
    k:int
):
    mask = maximally_different(D, k=k)
    imgs = imgs[mask, ...]
    print(mask.sum())
    assert mask.sum() >= k
    # visualize
    n, m = ceil(np.sqrt(k)), int(np.sqrt(k))
    fig, axs = plt.subplots(n, m)
    for i, ax in enumerate(axs.flatten()):
        ax.axis('off')
        ax.imshow(imgs[i, ...], cmap='gray')
    # return the figure
    return fig

# create the plots for k = 9, 16, 25
fig = show_maximally_different(imgs, D, k=9)
fig.savefig("Figures/imgs_k9.pdf")
fig = show_maximally_different(imgs, D, k=16)
```

```
fig.savefig("Figures/imgs_k16.pdf")
fig = show_maximally_different(imgs, D, k=25)
fig.savefig("Figures/imgs_k25.pdf")
```