

16

Non-negative Matrix Factorization

Lab Objective: *Understand and implement the non-negative matrix factorization generator for recommendation systems with CVXPY.*

Introduction

Collaborative filtering is the process of filtering data for patterns using collaboration techniques. More specifically, it refers to making prediction about a user's interests based on other users' interests. These predictions can be used to recommend items and are why collaborative filtering is one of the common methods of creating a recommendation system.

Recommendation systems look at the similarity between users to predict what item a user is most likely to enjoy. Common recommendation systems include Netflix's "Movies you Might Enjoy" list, Spotify's "Discover Weekly" playlist, and Amazon's "Products You Might Like" suggestions.

Non-negative Matrix Factorization

Non-negative matrix factorization is one algorithm used in collaborative filtering. It can be applied to many other cases, including image processing, text mining, clustering, and community detection. The objective of non-negative matrix factorization is to take a non-negative matrix V and factor it into the product of two non-negative matrices.

For $V \in \mathbb{R}^{m \times n}$, $0 \preceq W$,

$$\begin{aligned} & \text{minimize} && \|V - WH\| \\ & \text{subject to} && 0 \preceq W, 0 \preceq H \\ & \text{where} && W \in \mathbb{R}^{m \times k}, H \in \mathbb{R}^{k \times n} \end{aligned}$$

k is the rank of the decomposition and can either be specified or found using the Root Mean Squared Error (the square root of the MSE), SVD, Non-negative Least Squares, or cross-validation techniques.

For this lab, we will use the Frobenius norm, given by

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2}.$$

It is equivalent to the square root of the sum of the diagonal of $A^H A$

Problem 1. Create the `NMFRecommender` class, which will be used to implement the NMF algorithm. Initialize the class with the following parameters: `random_state` defaulting to 15, `tol` defaulting to $1e-3$, `maxiter` defaulting to 200, and `rank` defaulting to 3.

Add a method called `initialize_matrices` that takes in m and n , the dimensions of V . Set the random seed so that initializing the matrices can be replicated:

```
np.random.seed(self.random_state)
```

Initialize W and H using randomly generated numbers between 0 and 1, where $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$, where $k = \text{rank}$. Store W and H as attributes and return them.

CVXPY

In order to compute the NMF of a matrix we will use the convex optimization package CVXPY. CVXPY is convex optimization package that takes a problem, converts it into standard form, calls a solver, and processes the result. Here is a basic example of how to use CVXPY

```
import cvxpy as cp

# Create two scalar optimization variables.
x = cp.Variable()
y = cp.Variable()

# Create two constraints.
constraints = [x + y == 1,
              x - y >= 1]

# Form objective.
obj = cp.Minimize((x - y)**2)

# Form and solve problem.
prob = cp.Problem(obj, constraints)
prob.solve() # Returns the optimal value.

# print the status of the solution
print(prob.status)
```

The variables `x` and `y` are updated as the problem is solved. Constraints are not required.

Vector valued variables can be created by including the dimension of the variable as an argument like so: `x = cp.Variable(10)`. Similarly matrix valued variables can be created: `x = cp.Variable((5,5))`.

When initialized, variables have values of `None`. A value can be assigned to a variable before a problem is solved. The value can also be extracted after the optimal solution to a problem is found.

```
import numpy as np

random_matrix = np.random.random((5,5))
x = cp.Variable((5,5))
x.value = random_matrix
'''
solve a problem using the variable x
'''
solution_matrix = x.value
```

Problem 2. Finish the NMF class by adding a method `fit` that uses CVXPY to find an optimal W and H . It should accept V as a numpy array.

Constructing the problem so that it is known to be convex is important. Unfortunately, optimizing over a pair of matrices that are being multiplied together breaks the rules of disciplined convex programming. Because of this, you must solve for W and H alternately by solving for an optimal W given an H , then for an optimal H given a W and so on.

Finally add a method called `reconstruct` that reconstructs and returns V by multiplying W and H .

HINT: You can build non-negativity into a CVXPY variable with `x = cp.Variable(n, nonneg=True)`. You can check if the solution is optimal by checking the status of your problem object with `prob.status`.

Using NMF for Recommendations

Consider the following marketing problem where we have a list of five grocery store customers and their purchases. We want to create personalized food recommendations for their next visit. We start by creating a matrix representing each person and the number of items they purchased in different grocery categories. So from the matrix, we can see that John bought two fruits and one sweet.

$$V = \begin{pmatrix} 0 & 1 & 0 & 1 & 2 & 2 \\ 2 & 3 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 2 & 3 & 4 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{matrix} \textit{Vegetables} \\ \textit{Fruits} \\ \textit{Sweets} \\ \textit{Bread} \\ \textit{Coffee} \end{matrix}$$

After performing NMF on V , we'll get the following W and H .

$$W = \begin{pmatrix} & \textit{Component1} & \textit{Component2} & \textit{Component3} \\ \textit{Vegetables} & 2.1 & 0.03 & 0. \\ \textit{Fruits} & 1.17 & 0.19 & 1.76 \\ \textit{Sweets} & 0.43 & 0.03 & 0.89 \\ \textit{Bread} & 0.26 & 2.05 & 0.02 \\ \textit{Coffee} & 0.45 & 0. & 0. \end{pmatrix}$$

$$H = \begin{pmatrix} \textit{John} & \textit{Alice} & \textit{Mary} & \textit{Greg} & \textit{Peter} & \textit{Jennifer} \\ \textit{Component1} & 0.00 & 0.45 & 0.00 & 0.43 & 1.0 & 0.9 \\ \textit{Component2} & 0.00 & 0.91 & 1.45 & 1.9 & 0.35 & 0.37 \\ \textit{Component3} & 1.14 & 1.22 & 0.55 & 0.0 & 0.47 & 0.53 \end{pmatrix}$$

W represents how much each grocery feature contributes to each component; a higher weight means it's more important to that component. For example, component 1 is heavily determined by vegetables followed by fruit, then coffee, sweets and finally bread. Component 2 is represented almost entirely by bread, while component 3 is based on fruits and sweets, with a small amount of bread. H is similar, except instead of showing how much each grocery category affects the component, it shows a much each person belongs to the component, again with a higher weight indicating that the person belongs more in that component. We can see the John belongs in component 3, while Jennifer mostly belongs in component 1.

To get our recommendations, we reconstruct V by multiplying W and H .

$$WH = \begin{pmatrix} \textit{John} & \textit{Alice} & \textit{Mary} & \textit{Greg} & \textit{Peter} & \textit{Jennifer} \\ \textit{Vegetables} & 0.0000 & 0.9723 & 0.0435 & 0.96 & 2.1105 & 1.9011 \\ \textit{Fruits} & 2.0064 & 2.8466 & 1.2435 & 0.8641 & 2.0637 & 2.0561 \\ \textit{Sweets} & 1.0146 & 1.3066 & 0.533 & 0.2419 & 0.8588 & 0.8698 \\ \textit{Bread} & 0.0228 & 2.0069 & 2.9835 & 4.0068 & 0.9869 & 1.0031 \\ \textit{Coffee} & 0.0000 & 0.2025 & 0.0000 & 0.1935 & 0.45 & 0.405 \end{pmatrix}$$

Most of the zeros from the original V have been filled in. This is the **collaborative filtering** portion of the algorithm. By sorting each column by weight, we can predict which items are more attractive to the customers. For instance, Mary has the highest weight for bread at 2.9835, followed by fruit at 1.2435 and then sweets at .533. So we would recommend bread to Mary.

Another way to interpret WH is to look at a feature and determine who is most likely to buy that item. So if we were having a sale on sweets but only had funds to let three people know, using the reconstructed matrix, we would want to target Alice, John, and Jennifer in that order. This gives us more information that V alone, which says that everyone except Greg bought one sweet.

Problem 3. Use the `NMFRecommender` class to run NMF on V , defined above, with 2 components. Return W , H , and the number of people who have higher weights in component 2 than in component 1.

Sklearn NMF

We also can compute the NMF using SkLearn. SkLearn uses a similar process as our class above. Here rank is represented by the parameter `n_components`.

```
>>> from sklearn.decomposition import NMF

>>> model = NMF(n_components=2, init='random', random_state=0)
>>> W = model.fit_transform(V)
>>> H = model.components_
```

Endmember Detection using NMF

NMF can be used for analysis and identification of images. If each image corresponds to a $j \times k$ array of pixel intensities, then the data matrix is constructed by flattening the image into a vector of length jk and using these vectors as the columns of V , so V has dimensions $jk \times n$, where n is the number of images to analyze. Typically the materials in an image are referred to as the endmembers, which can be thought of as basis images which can be combined to reconstruct any image in the dataset. In the NMF decomposition $H[k, j]$ represents the abundance of the k th endmember or basis face in the j th image. $W[:, k]$ represents the spectral signature of the k th endmember. Then $V[:, j] \cong WH[:, j]$.

Example

```
from sklearn.datasets import load_sample_images
import numpy as np

# function to convert colored image to gray scale image
def gray_convert(rgb):
    r, g, b = rgb[:, :, 0], rgb[:, :, 1], rgb[:, :, 2]
    gray = 0.2989 * r + 0.5870 * g + 0.1140 * b
    return gray

# load in sample images
dataset = load_sample_images()
# grab the first image
image = dataset.images[0]
# convert image to gray scale
image = gray_convert(image)

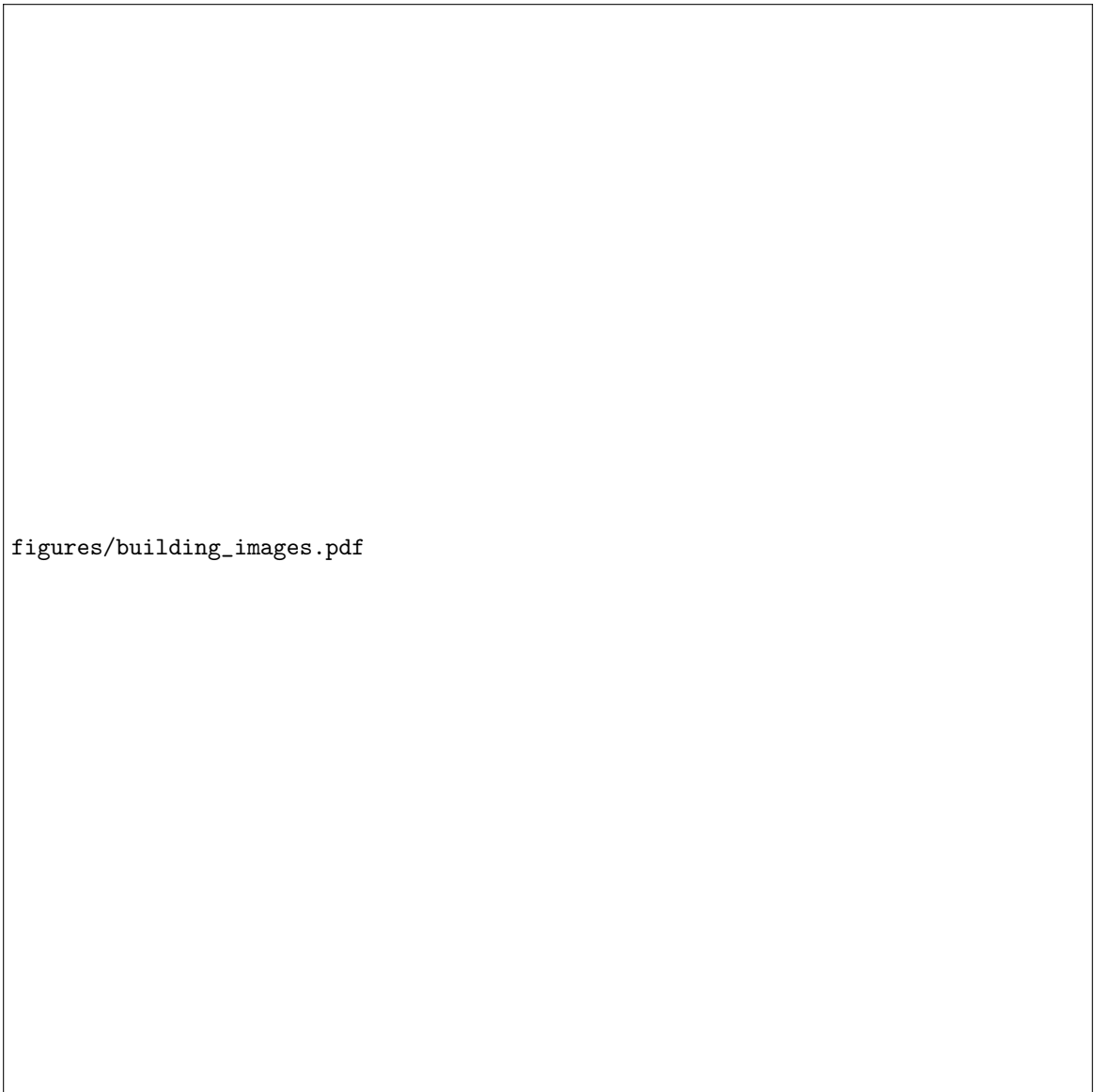
# get augmentations for additional images
flipLR = np.fliplr(image)
flipUP = np.flipud(image)

# create matrix V
images = [np.ravel(image), np.ravel(flipLR), np.ravel(flipUP)]
images = np.transpose(images)
# decompose using NMF
```

```
model = NMF(n_components = 5,max_iter = 1000)
W = model.fit_transform(images)
H = model.components_

# plot basis images
plt.subplots_adjust(wspace = .02,hspace = .05)
plt.figure(figsize=(20, 8))
for i in range(W.shape[1]):
    plt.subplot(1,5,i+1)
    plt.xticks([], [])
    plt.yticks([], [])
    plt.imshow(reshape_image(W[:,i],427,640),cmap = 'gray')
```

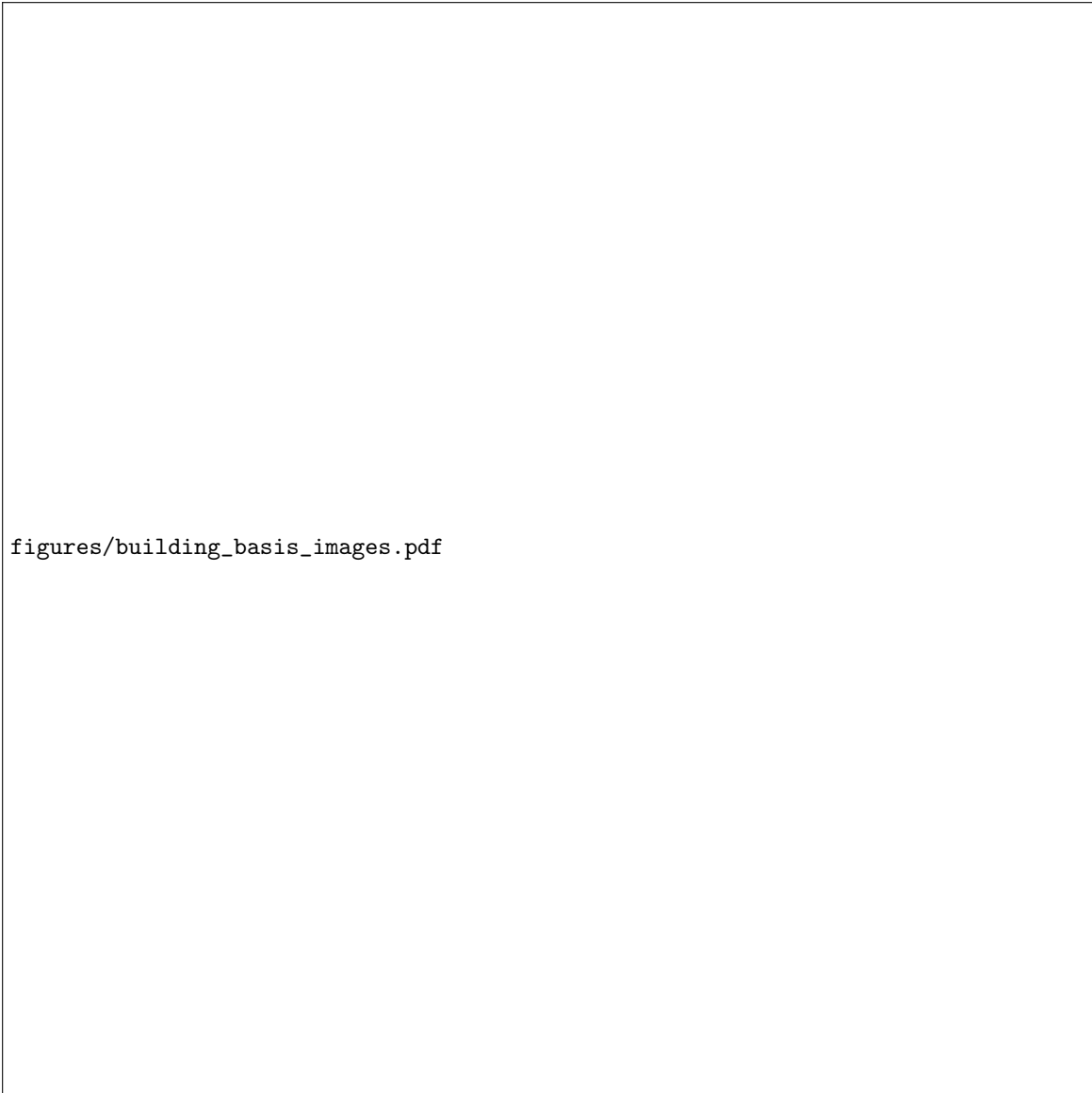
The following three flattened images now make up the columns of V .



figures/building_images.pdf

Figure 16.1: Original Images

Using a rank 5 reconstruction we see that the features used to reconstruct each image deal with the orientation of the building as well as the positive and negative space in each building.



figures/building_basis_images.pdf

Figure 16.2: Basis images for a rank 5 deconstruction. Each basis image comes from a column of W .

For the next two problems we will be using a dataset of facial images that we can load in with the code below.

```
def get_faces(path="./faces94"):  
    """Traverse the specified directory to obtain one image per subdirectory.  
    Flatten and convert each image to grayscale.  
  
    Parameters:  
        path (str): The directory containing the dataset of images.  
  
    Returns:  
        ((mn,k) ndarray) An array containing one column vector per
```



```

        subdirectory. k is the number of people, and each original
        image is mxn.
    """

    # Traverse the directory and get one image per subdirectory.
    faces = []
    for (dirpath, dirnames, filenames) in os.walk(path):
        for fname in filenames:
            if fname[-3:]=="jpg":          # Only get jpg images.
                # Load the image, convert it to grayscale,
                # and flatten it into a vector.
                faces.append(np.ravel(imread(dirpath+"/"+fname, as_gray=True)))
                break
    # Put all the face vectors column-wise into a matrix.
    return np.transpose(faces)

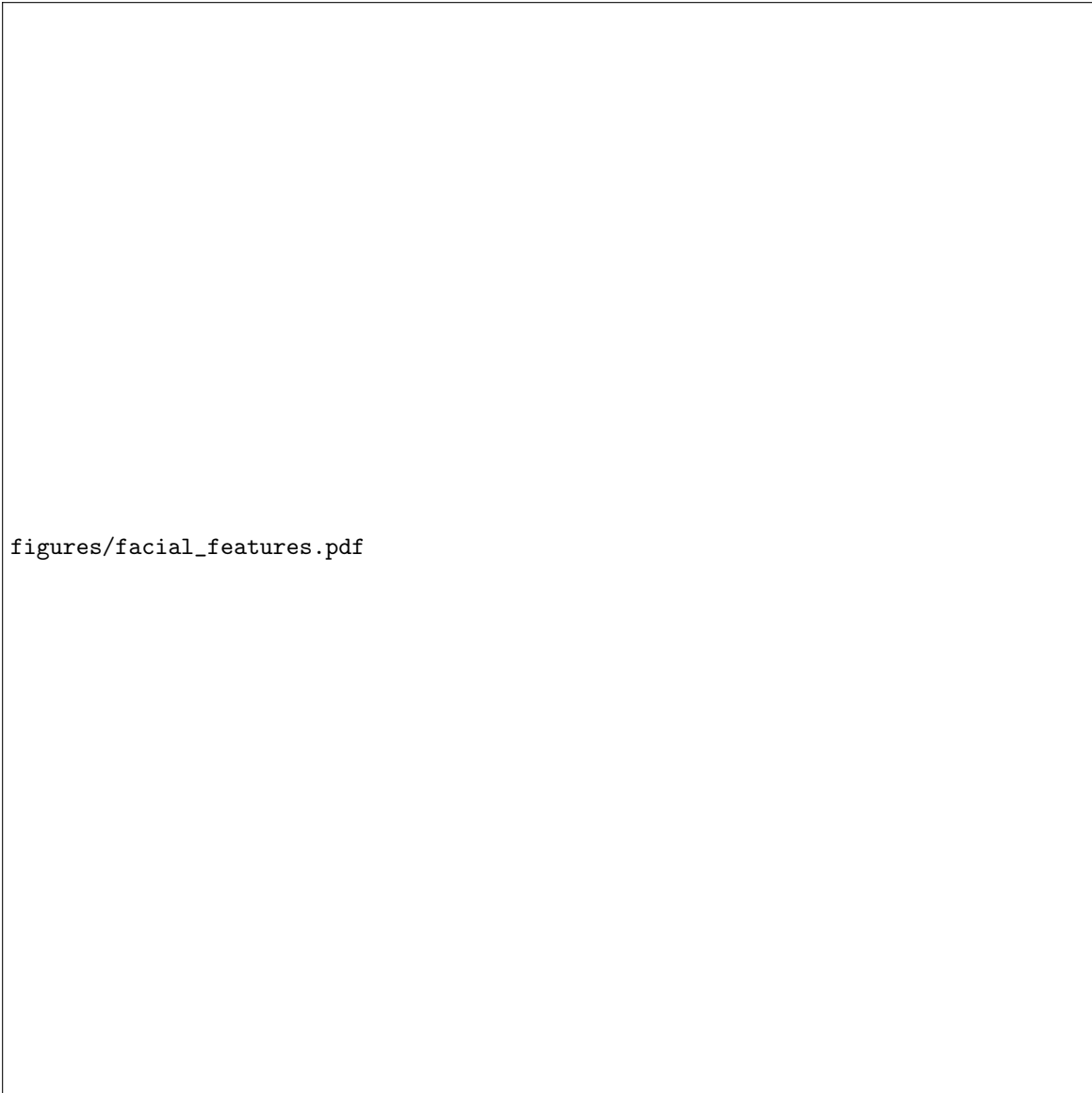
def show(image, m=200, n=180, plt_show=False):
    """Plot the flattened grayscale 'image' of width 'w' and height 'h'.

    Parameters:
        image ((mn,) ndarray): A flattened image.
        m (int): The original number of rows in the image.
        n (int): The original number of columns in the image.
        plt_show (bool): if True, call plt.show() at the end
    """
    #scale image
    image = image / 255
    #reshape image
    image = np.reshape(image, (m,n))
    #show image
    plt.imshow(image, cmap = "gray")

    if plt_show:
        plt.show()

```

Similar to the example, we have basis faces that are used to reconstruct the images in the original dataset. A sample of basis faces for a rank 75 reconstruction of the faces dataset seem to correspond to the following endmembers; forehead, glasses, hair.



figures/facial_features.pdf

Figure 16.3: A sample of basis faces. Each basis face comes from a column of W .

Problem 4. Load in the facial dataset. SkLearn has the option to add regularization terms with coefficients (`alpha` and `l1_ratio`) to the objective function $\|V - WH\|$. Reconstruct the third face in the dataset using SkLearn's NMF. Perform a grid search over the following: `n_components = [75]`, `alpha = [0, .2, .5]`, and `l1_ratio = [0, 10-5, 10]`. Note this will take a while to run (approximately 15 minutes). Plot all reconstructions of the third face and put the parameters in the title, use subplots.

Problem 5. Run NMF on the facial dataset again, using the best parameters from the problem above. Next, for the second and twelfth faces in the dataset, find the 10 basis faces with the largest coefficients. Plot these basis faces along with the original image using subplots. In a markdown block write a sentence or two about differences you notice in the features of the basis faces (look closely).