Numerical Methods, Homework 1

Niko Grupen

March 2019

1 Nonnegative Matrix Factorization

Nonnegative matrix factorization (NMF) is a method of linear dimensionality reduction (LDR) that decomposes a matrix into a set of sparse feature vectors. As formalized in [1], given an $(m \times n)$ input matrix X with nonnegative values, NMF factors X such that:

$$X \approx WH \tag{1}$$

where W and H are lower-rank matrices of dimension $(m \times k)$ and $(k \times n)$, respectively. The corresponding optimization problem is outlined in [2]:

$$\min_{W \in \mathbb{R}^{(m \times k)}, H \in \mathbb{R}^{(k \times n)}} \|X - WH\|_F^2, \quad s.t. \quad W \geq 0, H \geq 0 \tag{2}$$

NMF was solved using multiplicative update rules with a squared L2-distance cost function [1]:

$$W \leftarrow W \frac{(XH^T)}{(WHH^T)}, \quad H \leftarrow H \frac{(W^T X)}{(W^T WH)} \tag{3}$$

$$\|A - B\|^2 = \Sigma_{ij}(A_{ij} - B_{ij})^2 \tag{4}$$

Factorizing a matrix in this way is particularly useful if our matrix X represents a high-dimensional data set. For example, if the columns of X are populated with some n-dimensional data vectors, solving (2) yields a set of expressive basis vectors W and weights H, such that a linear combination of the two forms a rough reconstruction of the input data. Thus, with a value k smaller than m and n, NMF results in a compressed approximation of the input data (WH).

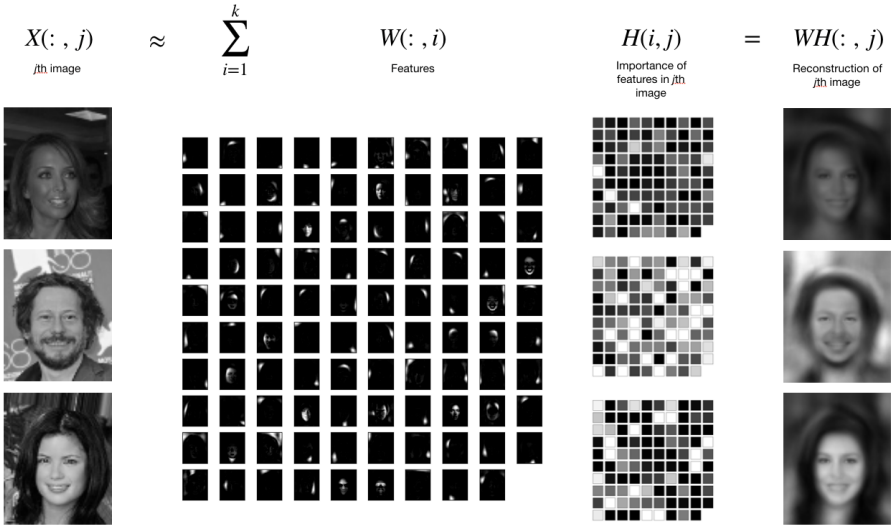

Figure 1: Results of NMF factorization on CelebA data set with k = 99. From left to right: Original input images $X(:, j)$, learned basis images $\sum_{i=1}^{k} W(:, i)$, weights $H(i, j)$, and approximate reconstructions $WH(:, j)$.

2 Qualitative Analysis

Taking motivation from [2], NMF is used to extract meaningful features from a set of input images. Here, $X \in \mathbb{R}^{(m \times n)}$ is constructed so that each column is a vectorized grayscale image. NMF yields the two factors W and H, where each column of W is a basis image representing some feature of the input space and H is the set of nonnegative weights used for reconstruction.

2.1 CelebA Dataset

NMF is run on a subset of images from the CelebA dataset [3]. CelebA consists of approx. 200k images of celebrities and is used for numerous computer vision tasks, including facial recognition/detection, landmark detection/localization, and as a test-bed for generative models.

The CelebA images are more complex than those used in [2] in that each image is not cropped to contain only facial features. The images in the CelebA contain a variety of face/hair styles – including those of both male and female subjects – across a wide range of poses, orientations, and lighting conditions. In an effort to test the

expressive power of NMF for complex images, no pre-processing was applied to the images (besides resizing).

(a) High quality reconstructions. (b) Low quality reconstructions.

Figure 2: Example reconstructions for various levels of the factorization rank ($k = 49, k = 99, k = 500$).

2.2 Results

A selection of 4,000 images from the CelebA data set were arranged in an input matrix X such that each column $X(:,\ j)$ was a flattened image vector. Each image was scaled by a factor of 0.4 to be size (87×71), so X has dimensions (6177×4000).

Fig.1 shows the decomposition produced from 1000 iterations of NMF via multiplicative updates. Of particular note is the set of basis images learned during factorization. From examination, it appears the basis vectors encode important image features such as the shape/pose of the face, the makeup of facial features (i.e. eyes, chin, forehead), and the texture/color of the background, amongst other things. The full set of basis images is presented again in the appendix.

Choice of the factorization rank, k in our case, presents a tradeoff between the amount of compression in the factorized WH and the expressivity of the latent representation. Fig.2 provides examples of the expressivity tradeoff and Fig.3 shows the effect on cost, as by (4), of the factorization rank. As expected, increasing the factorization rank leads to lower costs and improves the quality of the approximate reconstruction.

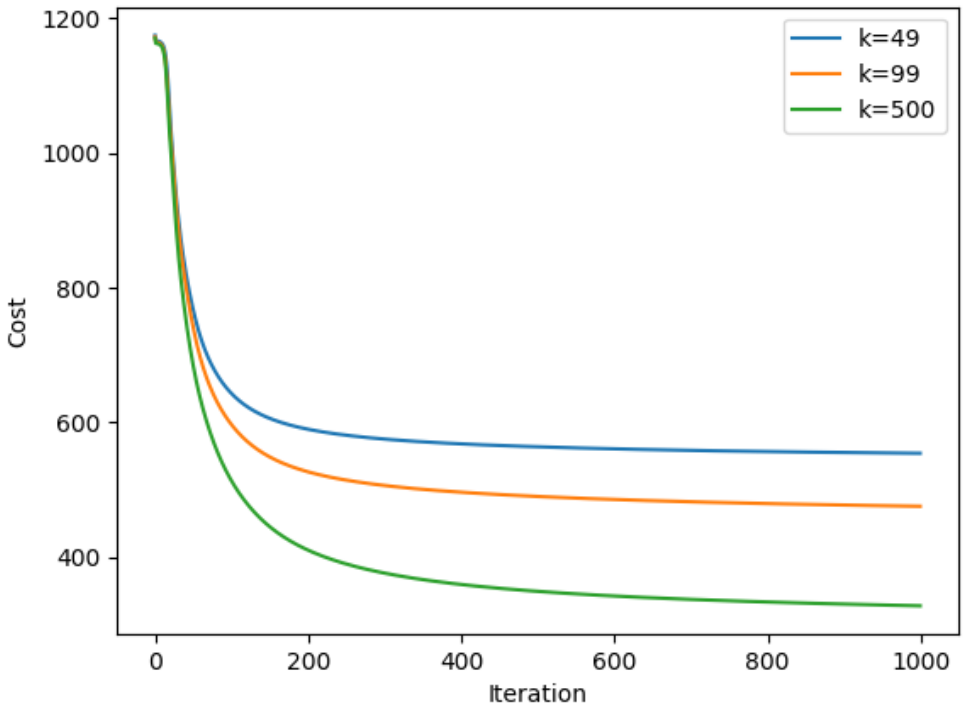

Figure 3: Comparison of costs for various factorization ranks.

3 Conclusion

NMF is a straightforward method of LDR that works surprisingly well on complex images. Though certainly less expressive than non-linear techniques and more modern approaches to latent representation (i.e. AutoEncoders, GANs, etc.), NMF offers a lower overhead and easy-to-digest form of matrix decomposition.

References

[1] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Proceedings of the 13th International Conference on Neural Information Processing Systems*, NIPS'00, (Cambridge, MA, USA), pp. 535–541, MIT Press, 2000.

[2] N. Gillis, "The Why and How of Nonnegative Matrix Factorization," *arXiv e-prints*, Jan 2014.

[3] Z. Liu, P. Luo, X. Wang, and X. Tang, "Deep learning face attributes in the wild," in *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

A Appendix

A.1 Implementation

```
import pickle
import random
from matplotlib import pyplot as plt
from PIL import Image
import numpy as np
from scipy import misc
import os
import cv2

IMG_H = int(218 * 0.4)
IMG_W = int(178 * 0.4)
IMG_CHANNELS = 1
NUM_IMGS = 4000 # also run with 10k
MAX_ITRS = 1000

class NMF_Solver(object):

    def __init__(self, x, k=4):
        # x should have shape (row, col) = (IMG_H*IMG_W*IMG_CHANNELS, len(files))
        self.x = x
```

```
        # init W (m x k) and H (k x n)
        m, n = x.shape[0], x.shape[1]
        self.w = np.random.rand(m, k)
        self.h = np.random.rand(k, n)
        self.costs = []

        print('x shape = {}'.format(self.x.shape))
        print('w shape = {}'.format(self.w.shape))
        print('h shape = {}'.format(self.h.shape))

    def solve(self, itrs):
        for i in range(itrs):
            # update W and H sequentially
            self.w = self.update_w()
            self.h = self.update_h()

            # calculate cost (EUCLIDEAN DISTANCE between imgs and HW)
            cost = np.linalg.norm(self.x - np.linalg.multi_dot([self.w, self.h]))
            self.costs.append(cost)
            print('Iteration {}: cost = {}'.format(i, cost))

            # start saving checkpoints halfway
            if i > 49 and i % 50 == 0:
                self.save_checkpoint(i)

        self.save_checkpoint('final', final=True)

    def update_w(self):
        # W .* (X * H')
        top = self.w * np.linalg.multi_dot([self.x, np.transpose(self.h)])

        # W * (H * H')
        bot = np.linalg.multi_dot([self.w, np.linalg.multi_dot([self.h, np.transpose(self.h)])])

        return top / bot

    def update_h(self):
        # H .* (W' * X)
        top = self.h * np.linalg.multi_dot([np.transpose(self.w), self.x])

        # (W * W') * H
        bot = np.linalg.multi_dot([np.linalg.multi_dot([np.transpose(self.w), self.w]), self.h])

        return top / bot

    def save_checkpoint(self, itr, final=False):
        # save W, H matrices and costs up to this point
        print('saving checkpoint!')
        np.save('checkpoints/gray_49_4k/w_mat_{}'.format(itr), self.w, allow_pickle=True)
        np.save('checkpoints/gray_49_4k/h_mat_{}'.format(itr), self.h, allow_pickle=True)
        np.save('checkpoints/gray_49_4k/costs_{}'.format(itr), np.asarray(self.costs), allow_pickle=True)

        if final:
            # save X matrix at very end
            np.save('checkpoints/gray_49_4k/x_mat_{}'.format(itr), self.x, allow_pickle=True)
```

```
def load_imgs(fp, num):
    # load images into numpy arrays
    files = os.listdir(fp)[:num]

    imgs = []
    for f in sorted(files):
        print('working on file {}/{}'.format(len(imgs), len(files)))
        img = misc.imread(os.path.join(fp, f))
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # shrink by half
        img = cv2.resize(img, (0,0), None, 0.4, 0.4)

        flat = img.reshape(img.shape[0] * img.shape[1]) / 255
        imgs.append(flat)

    # stack into matrix where each col is an img vector
    mat = np.stack(imgs, axis=1)

    return mat


def show_img(img):
    # CelebA images are (218, 178), but we are decreasing size to 0.4 of that for performance
    img = img.reshape([IMG_H, IMG_W])

    plt.imshow(img, cmap='gray')
    plt.show()


def main():
    # load CelebA
    fp = '/Users/nikogrupen/Documents/developer/playground/data/img_align_celeba'
    imgs = load_imgs(fp, NUM_IMGS)

    # sample random image to display
    rand = random.randint(0, NUM_IMGS)
    show_img(imgs[:, rand])

    # init NMF
    nmf = NMF_Solver(imgs, k=49)

    # solve
    nmf.solve(MAX_ITRS)


if __name__  == '__main__':
    main()
```

A.2 Basis Images

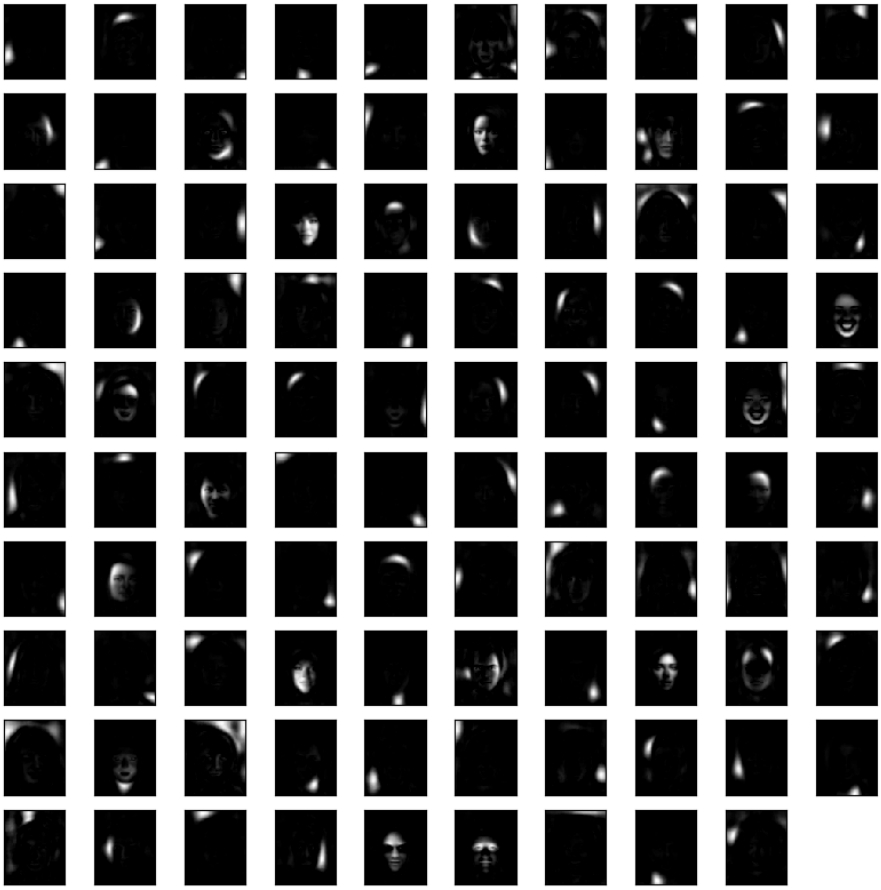

Figure 4: Full set of basis images ($k = 99$).