Emanuele Belli 03720425
Thomas Brunner 03675118
Luca Corbucci 03726968

Exercise Sheet 12

Machine Learning
February 2, 2020

.

Problem 2

- In the case described we are using a liner function $W \in R^{KxD}$ to map data in from R^D into R^K. In particular requiring a zero reconstruction error would require

$$BWX = X <=> BW = Id$$

Where $B \in R^{DxK}$. But as the $rank(W)$ is K and not D, we do not have existence of a left inverse.

- It is possible if $K \geq D$ because we then can define a left inverse.

Problem 3

Compute the extected value:

$$E[X] = E[\sum_k \pi_k N(x|\mu_k, \Sigma_k)] = \sum_k E[\pi_k N(x|\mu_k, \Sigma_k)] = \sum_k \pi_k E[N(x|\mu_k, \Sigma_k)] = \sum_k \pi_k \mu_k$$

Compute the covariance:

$$Cov[X] = E[xx^T] - E[x]E[x]^T = \sum_k \pi_k E[xx^T] - E[x]E[x]^T$$

We know that $E[xx^T] = Cov[x] + E[x]E[x]^T = \sum_k + \mu_k \mu_k^T$
We can conclude that:

$$Cov[X] = \sum_k \pi_k (\Sigma_k + \mu_k \mu_k^T) - E[X]E[X]^T$$

Problem 4

a)

- First we sample the cluster k from $Cat(\pi^x)$
- We draw the sample x from the cluster k: $x \sim \mathcal{N}(\mu_k^x, \Sigma_k^x)$
- We sample the cluster l from $Cat(\pi^y)$
- We draw the sample y from cluster l: $y \sim \mathcal{N}(\mu_l^x, \Sigma_l^x)$
- We can compute $z = y + x = \mathcal{N}(\mu_l^x, \Sigma_l^x) + \mathcal{N}(\mu_k^x, \Sigma_k^x)$

Exercise Sheet 12

b) In *a)* we computed $z = y + x = \mathcal{N}(\mu_l^x, \Sigma_l^x) + \mathcal{N}(\mu_k^x, \Sigma_k^x)$, z is a gaussian because it is the sum of two Gaussian distributions: $z \sim \mathcal{N}(\mu_l^y + \mu_k^x, \Sigma_l^y + \Sigma_k^x)$. The probability of the sample from $\mathcal{N}(\mu_l^y + \mu_k^x, \Sigma_l^y + \Sigma_k^x)$ is $\pi_l^y * \pi_r^x$. To compute the likelihod of z we have a summation of the product of a probability π and a Gaussian so the result is again a Gaussian mixture model.

c)

$$P(z|\pi_z, \mu_z, \Sigma_z) = \sum_{k=1} P(C_k = 1|\pi) P(z|C_k = 1, \mu_k^z, \Sigma_k^z) = \sum_{k=1}^{K} \sum_{l=1}^{K} \pi_l^x \pi_k^y \mathcal{N}(z|\mu_k^x + \mu_l^y, \Sigma_k^x + \Sigma_k^y)$$

Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is

1. Run all the cells of the notebook.
2. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)).
3. Concatenate your solutions for other tasks with the output of Step 2. On linux, you can use `pdfunite` , there are similar tools for other platforms, too. You can only upload a single PDF file to Moodle.

Make sure you are using `nbconvert` version 5.5 or later by running `jupyter nbconvert --version` . Older versions clip lines that exceed page width, which makes your code harder to grade.

Matrix Factorization

In [1]:

```
import time
import scipy.sparse as sp
import numpy as np
from scipy.sparse.linalg import svds
from sklearn.linear_model import Ridge

import matplotlib.pyplot as plt
%matplotlib inline
```

Restaurant recommendation

The goal of this task is to recommend restaurants to users based on the rating data in the Yelp dataset. For this, we try to predict the rating a user will give to a restaurant they have not yet rated based on a latent factor model.

Specifically, the objective function (loss) we wanted to optimize is:

$$\mathcal{L} = \min_{P,Q} \sum_{(i,x)\in W} (M_{ix} - \mathbf{q}_i^T \mathbf{p}_x)^2 + \lambda \sum_x \|\mathbf{p}_x\|^2 + \lambda \sum_i \|\mathbf{q}_i\|^2$$

where W is the set of (i, x) pairs for which the rating M_{ix} given by user i to restaurant x is known. Here we have also introduced two regularization terms to help us with overfitting where λ is hyper-parameter that control the strength of the regularization.

Hint 1: Using the closed form solution for regression might lead to singular values. To avoid this issue perform the regression step with an existing package such as scikit-learn. It is advisable to use ridge regression to account for regularization.

Hint 2: If you are using the scikit-learn package remember to set `fit_intercept = False` to only learn the coefficients of the linear regression.

Load and Preprocess the Data (nothing to do here)

In [2]:

```
ratings = np.load("exercise_12_matrix_factorization_ratings.npy")
```

In [3]:

```
# We have triplets of (user, restaurant, rating).
ratings
```

Out[3]:

```
array([[101968,   1880,      1],
       [101968,    284,      5],
       [101968,   1378,      2],
       ...,
       [ 72452,   2100,      4],
       [ 72452,   2050,      5],
       [ 74861,   3979,      5]])
```

Now we transform the data into a matrix of dimension [N, D], where N is the number of users and D is the number of restaurants in the dataset. We store the data as a sparse matrix to avoid out-of-memory issues.

In [4]:

```
n_users = np.max(ratings[:,0] + 1)
n_restaurants = np.max(ratings[:,1] + 1)
M = sp.coo_matrix((ratings[:,2], (ratings[:,0], ratings[:,1]))
, shape=(n_users, n_restaurants)).tocsr()
M
```

Out[4]:

```
<337867x5899 sparse matrix of type '<class 'numpy.
longlong'>'
        with 929606 stored elements in Compressed
Sparse Row format>
```

To avoid the cold start problem (https://en.wikipedia.org/wiki/Cold_start_(computing)), in the preprocessing step, we recursively remove all users and restaurants with 10 or less ratings.

Then, we randomly select 200 data points for the validation and test sets, respectively.

After this, we subtract the mean rating for each users to account for this global effect.

Note: Some entries might become zero in this process -- but these entries are different than the 'unknown' zeros in the matrix. We store the indices for which we the rating data available in a separate variable.

In [5]:

```
def cold_start_preprocessing(matrix, min_entries):
    """
    Recursively removes rows and columns from the input matrix
which have less than min_entries nonzero entries.

    Parameters
    ----------
    matrix      : sp.spmatrix, shape [N, D]
                  The input matrix to be preprocessed.
    min_entries : int
                  Minimum number of nonzero elements per row a
nd column.

    Returns
    -------
    matrix      : sp.spmatrix, shape [N', D']
                  The pre-processed matrix, where N' <= N and
D' <= D

    """
    print("Shape before: {}".format(matrix.shape))

    shape = (-1, -1)
    while matrix.shape != shape:
        shape = matrix.shape
        nnz = matrix>0
        row_ixs = nnz.sum(1).A1 > min_entries
        matrix = matrix[row_ixs]
        nnz = matrix>0
        col_ixs = nnz.sum(0).A1 > min_entries
        matrix = matrix[:,col_ixs]
    print("Shape after: {}".format(matrix.shape))
    nnz = matrix>0
    assert (nnz.sum(0).A1 > min_entries).all()
    assert (nnz.sum(1).A1 > min_entries).all()
    return matrix
```

Task 1: Implement a function that subtracts the mean user rating from the sparse rating matrix

In [6]:

```
def shift_user_mean(matrix):
    """
    Subtract the mean rating per user from the non-zero elemen
ts in the input matrix.

    Parameters
    ----------
    matrix : sp.spmatrix, shape [N, D]
             Input sparse matrix.
    Returns
    -------
    matrix : sp.spmatrix, shape [N, D]
             The modified input matrix.

    user_means : np.array, shape [N, 1]
                 The mean rating per user that can be used to
recover the absolute ratings from the mean-shifted ones.

    """

    # TODO: Compute the modified matrix and user_means
    sum_ = np.squeeze(np.asarray(matrix.sum(1)))
    nonZero = np.diff(matrix.tocsr().indptr)
    user_means = (sum_/nonZero)

    diag = sp.diags(user_means,0)

    matrix_copy = matrix.copy()
    matrix_copy.data = np.ones_like(matrix_copy.data)


    matrix = matrix - (diag*matrix_copy).todense()
    user_means = user_means.reshape(len(user_means), 1)
    user_means = np.asmatrix(user_means)
    assert np.all(np.isclose(matrix.mean(1), 0))
    return matrix, user_means
```

Split the data into a train, validation and test set (nothing to do here)

In [7]:

```
def split_data(matrix, n_validation, n_test):
    """
```

```
    Extract validation and test entries from the input matrix.

    Parameters
    ----------
    matrix          : sp.spmatrix, shape [N, D]
                      The input data matrix.
    n_validation    : int
                      The number of validation entries to extr
act.
    n_test          : int
                      The number of test entries to extract.

    Returns
    -------
    matrix_split    : sp.spmatrix, shape [N, D]
                      A copy of the input matrix in which the
validation and test entries have been set to zero.

    val_idx         : tuple, shape [2, n_validation]
                      The indices of the validation entries.

    test_idx        : tuple, shape [2, n_test]
                      The indices of the test entries.

    val_values      : np.array, shape [n_validation, ]
                      The values of the input matrix at the va
lidation indices.

    test_values     : np.array, shape [n_test, ]
                      The values of the input matrix at the te
st indices.

    """

    matrix_cp = matrix.copy()
    non_zero_idx = np.argwhere(matrix_cp)
    ixs = np.random.permutation(non_zero_idx)
    val_idx = tuple(ixs[:n_validation].T)
    test_idx = tuple(ixs[n_validation:n_validation + n_test].T
)

    val_values = matrix_cp[val_idx].A1
    test_values = matrix_cp[test_idx].A1

    matrix_cp[val_idx] = matrix_cp[test_idx] = 0
    matrix_cp.eliminate_zeros()

    return matrix_cp, val_idx, test_idx, val_values, test_valu
```

```
    return matrix_cp, val_idx, test_idx, val_values, test_valu
es
```

In [8]:

```
M = cold_start_preprocessing(M, 20)
```

```
Shape before: (337867, 5899)
Shape after: (3529, 2072)
```

In [9]:

```
n_validation = 200
n_test = 200
# Split data
M_train, val_idx, test_idx, val_values, test_values = split_da
ta(M, n_validation, n_test)
```

In [10]:

```
# Remove user means.
nonzero_indices = np.argwhere(M_train)
M_shifted, user_means = shift_user_mean(M_train)
# Apply the same shift to the validation and test data.
val_values_shifted = val_values - user_means[np.array(val_idx)
.T[:,0]].A1
test_values_shifted = test_values - user_means[np.array(test_i
dx).T[:,0]].A1
```

Compute the loss function (nothing to do here)

In [11]:

```
def loss(values, ixs, Q, P, reg_lambda):
    """
    Compute the loss of the latent factor model (at indices ix
s).
    Parameters
    ----------
    values : np.array, shape [n_ixs,]
        The array with the ground-truth values.
    ixs : tuple, shape [2, n_ixs]
        The indices at which we want to evaluate the loss (usu
ally the nonzero indices of the unshifted data matrix).
    Q : np.array, shape [N, k]
        The matrix Q of a latent factor model.
    P : np.array, shape [k, D]
        The matrix P of a latent factor model.
    reg_lambda : float
        The regularization strength

    Returns
    -------
    loss : float
           The loss of the latent factor model.

    """
    mean_sse_loss = np.sum((values - Q.dot(P)[ixs])**2)
    regularization_loss =  reg_lambda * (np.sum(np.linalg.norm
(P, axis=0)**2) + np.sum(np.linalg.norm(Q, axis=1) ** 2))

    return mean_sse_loss + regularization_loss
```

Alternating optimization

In the first step, we will approach the problem via alternating optimization, as learned in the lecture. That is, during each iteration you first update Q while having P fixed and then vice versa.

Task 2: Implement a function that initializes the latent factors Q and P

In [12]:

```
def initialize_Q_P(matrix, k, init='random'):
    """
    Initialize the matrices Q and P for a latent factor model.

    Parameters
    ----------
    matrix : sp.spmatrix, shape [N, D]
             The matrix to be factorized.
    k      : int
             The number of latent dimensions.
    init   : str in ['svd', 'random'], default: 'random'
             The initialization strategy. 'svd' means that we
use SVD to initialize P and Q, 'random' means we initialize
             the entries in P and Q randomly in the interval [
0, 1).

    Returns
    -------
    Q : np.array, shape [N, k]
        The initialized matrix Q of a latent factor model.

    P : np.array, shape [k, D]
        The initialized matrix P of a latent factor model.
    """
    np.random.seed(0)

    # TODO: Compute Q and P
    N = matrix.shape[0]
    D = matrix.shape[1]
    if(init == 'random'):
        Q = np.random.rand(N,k)
        P = np.random.rand(k,D)
    else:
        u, s, vt = svds(matrix)
        P = u @ s
        Q = vt

    assert Q.shape == (matrix.shape[0], k)
    assert P.shape == (k, matrix.shape[1])
    return Q, P
```

Task 3: Implement the alternating optimization approach

In [13]:

```
def latent_factor_alternating_optimization(M, non_zero_idx, k,
val_idx, val_values,
                                           reg_lambda, max_ste
ps=100, init='random',
                                           log_every=1, patien
ce=5, eval_every=1):
    """
    Perform matrix factorization using alternating optimizatio
n. Training is done via patience,
    i.e. we stop training after we observe no improvement on t
he validation loss for a certain
    amount of training steps. We then return the best values f
or Q and P oberved during training.

    Parameters
    ----------
    M                : sp.spmatrix, shape [N, D]
                       The input matrix to be factorized.

    non_zero_idx     : np.array, shape [nnz, 2]
                       The indices of the non-zero entries of
the un-shifted matrix to be factorized.
                       nnz refers to the number of non-zero e
ntries. Note that this may be different
                       from the number of non-zero entries in
the input matrix M, e.g. in the case
                       that all ratings by a user have the sa
me value.

    k                : int
                       The latent factor dimension.

    val_idx          : tuple, shape [2, n_validation]
                       Tuple of the validation set indices.
                       n_validation refers to the size of the
validation set.

    val_values       : np.array, shape [n_validation, ]
                       The values in the validation set.

    reg_lambda       : float
                       The regularization strength.

    max_steps        : int, optional, default: 100
                       Maximum number of training steps. Note
that we will stop early if we observe
```

```
                        no improvement on the validation error
for a specified number of steps
                        (see "patience" for details).

    init                : str in ['random', 'svd'], default 'ran
dom'
                        The initialization strategy for P and
Q. See function initialize_Q_P for details.

    log_every           : int, optional, default: 1
                        Log the training status every X iterat
ions.

    patience            : int, optional, default: 5
                        Stop training after we observe no impr
ovement of the validation loss for X evaluation
                        iterations (see eval_every for details
). After we stop training, we restore the best
                        observed values for Q and P (based on
the validation loss) and return them.

    eval_every          : int, optional, default: 1
                        Evaluate the training and validation l
oss every X steps. If we observe no improvement
                        of the validation error, we decrease o
ur patience by 1, else we reset it to *patience*.

    Returns
    -------
    best_Q              : np.array, shape [N, k]
                        Best value for Q (based on validation
loss) observed during training

    best_P              : np.array, shape [k, D]
                        Best value for P (based on validation
loss) observed during training

    validation_losses : list of floats
                        Validation loss for every evaluation i
teration, can be used for plotting the validation
                        loss over time.

    train_losses        : list of floats
                        Training loss for every evaluation ite
ration, can be used for plotting the training
                        loss over time.

    converged_after     : int
                        it - patience*eval_every, where it is
```

```
the iteration in which patience hits 0,
                        or -1 if we hit max_steps before conve
rging.

    """

    # TODO: Compute best_Q, best_P, validation_losses, train_l
osses and converged_after

    Q, P = initialize_Q_P(M, k, init)
    best_Q = 0
    best_P = 0
    validation_losses = []
    train_losses = []
    converged_after = 0
    fit_intercept = False
    original_patience = patience
    prev_validation_error = 0

    rr_P = Ridge(alpha=reg_lambda)
    rr_Q = Ridge(alpha=reg_lambda)

    train_idx = M.nonzero()
    _,_, train_values = sp.find(M)


    stop = False
    count_patience = 0
    noImprovement = 0

    while(converged_after < max_steps and not stop):
        rr_P = rr_P.fit(Q, M)
        P = rr_P.coef_.T

        rr_Q = rr_Q.fit(P.T, M.T)
        Q = rr_Q.coef_

        if(converged_after % eval_every == 0):
            validation_error = loss(val_values, val_idx, Q, P,
reg_lambda)
            train_error = loss(train_values, train_idx, Q, P,
reg_lambda)
            validation_losses.append(validation_error)
            train_losses.append(train_error)

            if (len(validation_losses) > 1):
                if(validation_error >= validation_losses[-2]):
                    patience -= 1
```

```
                    noImprovement += 1

                else:
                    patience = original_patience
                    best_P = P
                    best_Q = Q
                    noImprovement = 0

        if(noImprovement == patience):
            stop = True

        converged_after+=1

        if(converged_after % log_every == 0 and len(validation
_losses) > 1):
            print(f"Iteration {converged_after}, Train Error:
{train_losses[-1]}, Validation Error: {validation_losses[-1]}"
)



    return best_Q, best_P, validation_losses, train_losses, co
nverged_after
```

Train the latent factor (nothing to do here)

In [14]:

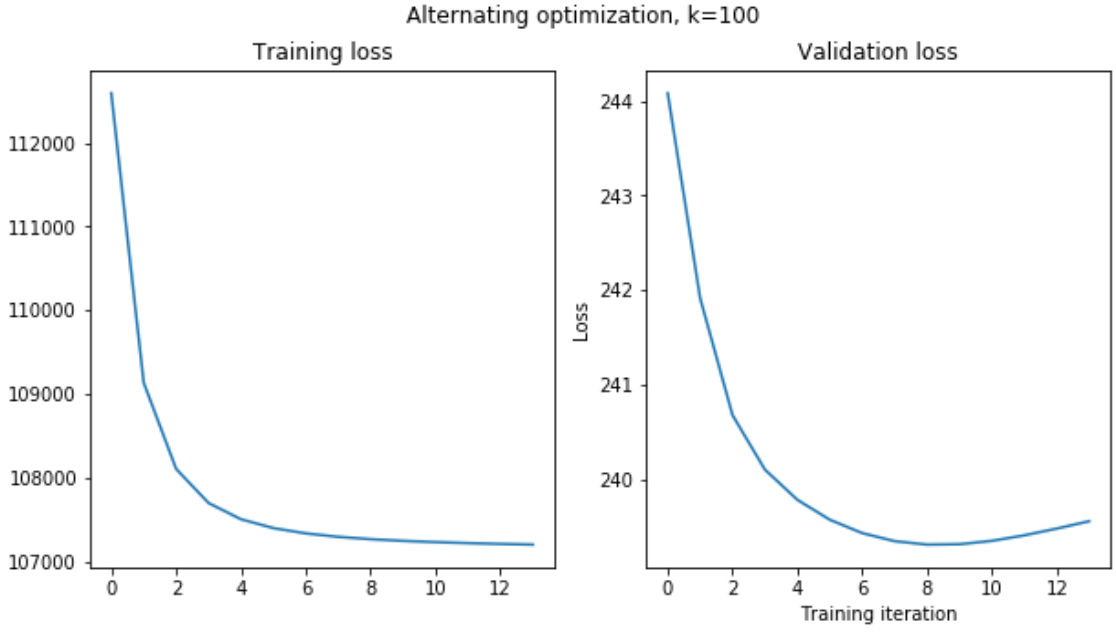
```
Q, P, val_loss, train_loss, converged = latent_factor_alternat
ing_optimization(M_shifted, nonzero_indices,

k=100, val_idx=val_idx,

val_values=val_values_shifted,

reg_lambda=1e-4, init='random',

max_steps=100, patience=10)
```

```
Iteration 2, Train Error: 112593.71599916997, Vali
dation Error: 244.08298773093534
Iteration 3, Train Error: 109138.30631146854, Vali
dation Error: 241.9162392621014
Iteration 4, Train Error: 108109.30699564128, Vali
dation Error: 240.6777749118207
Iteration 5, Train Error: 107701.7854206022, Valid
ation Error: 240.1017507813578
Iteration 6, Train Error: 107508.04165209441, Vali
dation Error: 239.78316060737086
Iteration 7, Train Error: 107402.53211614337, Vali
dation Error: 239.5725894762025
Iteration 8, Train Error: 107339.0629723255, Valid
ation Error: 239.43108141595647
Iteration 9, Train Error: 107298.0624986813, Valid
ation Error: 239.34599955778714
Iteration 10, Train Error: 107270.08091241548, Val
idation Error: 239.3094900829268
Iteration 11, Train Error: 107250.03527464424, Val
idation Error: 239.31364799949782
Iteration 12, Train Error: 107234.98064089604, Val
idation Error: 239.34954756447107
Iteration 13, Train Error: 107223.14173628515, Val
idation Error: 239.40775733822056
Iteration 14, Train Error: 107213.4362298555, Vali
dation Error: 239.47939412971326
Iteration 15, Train Error: 107205.20878231204, Val
idation Error: 239.55702669032163
```

Plot the validation and training losses over for each iteration (nothing to do here)

In [15]:

```
fig, ax = plt.subplots(1, 2, figsize=[10, 5])
fig.suptitle("Alternating optimization, k=100")

ax[0].plot(train_loss[1::])
ax[0].set_title('Training loss')
plt.xlabel("Training iteration")
plt.ylabel("Loss")


ax[1].plot(val_loss[1::])
ax[1].set_title('Validation loss')
plt.xlabel("Training iteration")
plt.ylabel("Loss")

plt.show()
```



In [ ]:

Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is

1. Run all the cells of the notebook.
2. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)).
3. Concatenate your solutions for other tasks with the output of Step 2. On linux, you can use `pdfunite` , there are similar tools for other platforms, too. You can only upload a single PDF file to Moodle.

Make sure you are using `nbconvert` version 5.5 or later by running `jupyter nbconvert --version` . Older versions clip lines that exceed page width, which makes your code harder to grade.

In [1]:

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
from sklearn.datasets import load_sample_image

%matplotlib inline

def compare_images(img, img_compressed, k):
    """Show the compressed and uncompressed image side by side
.
    """

    fig, axes = plt.subplots(1, 2, figsize=(16, 12))
    axes[0].set_axis_off()
    if isinstance(k, str):
        axes[0].set_title(k)
    else:
        axes[0].set_title(f"Compressed to {k} colors")
    axes[0].imshow(img_compressed)
    axes[1].set_axis_off()
    axes[1].set_title("Original")
    axes[1].imshow(img)
```

K-Means

In this first section you will implement the image compression algorithm from Bishop, chapter 9.1.1. Take an RGB image $X \in \mathbb{R}^{h \times w \times 3}$ and interpret it as a data matrix $X \in \mathbb{R}^{N \times 3}$. Now apply k-means clustering to find k colors that describe the image well and replace each pixel with its associated cluster.

In [2]:

```
# Alternatively try china.jpg
X = load_sample_image("flower.jpg")


# or load your own image
# X = np.array(Image.open("/home/user/path/to/some.jpg"))
```

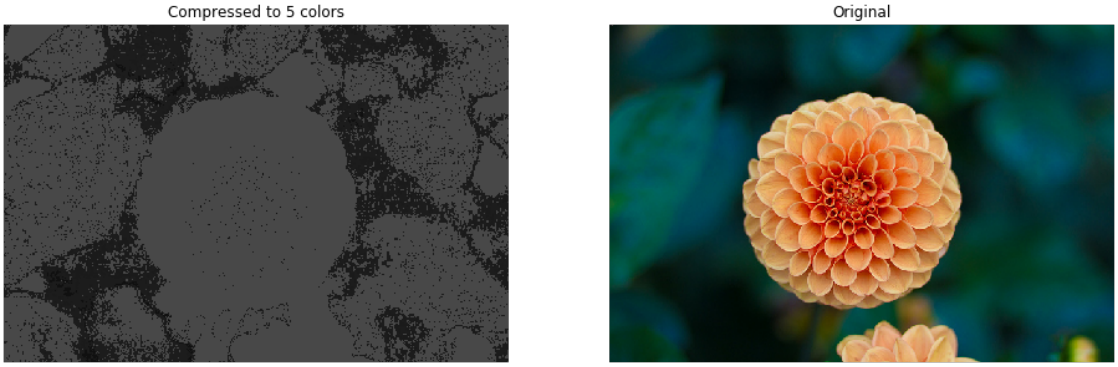
In [3]:

```
def computeDistance(point, mu):
    d = [np.linalg.norm(point-centroid) for centroid in mu]
    return d

def kmeans(X, k):
    """Compute a k-means clustering for the data X.

    Parameters
    ----------
    X : np.array of size N x D
        where N is the number of samples and D is the data dim
ensionality
    k : int
        Number of clusters

    Returns
    -------
    mu : np.array of size k x D
        Cluster centers
    z : np.array of size N
        Cluster indicators, i.e. a number in 0..k - 1, for eac
h data point in X
    """

    # TODO: Compute mu and z

    # Initialize centroids
```

```
    mu = X[:k]
    z = []
    distances = []
    clusters = [[] for _ in range(k)]
    convergence = False
    prevZ = []

    while (not convergence):
        # Compute distance of each point to the centroids and
assign each point to a cluster
        prevZ = z
        for point in X:
            d = computeDistance(point, mu)
            cluster = np.argmin(d)
            z.append(cluster)
            clusters[cluster].append(point)

        # Compute the new centroids
        for i in range(k):
            mu[i] = np.mean(clusters[i])

        if(prevZ == z):
            convergence = True

    return mu, z
```

In [4]:

```
# Cluster the color values
k = 5
mu, z = kmeans(X.reshape((-1, 3)), k)

# Replace each pixel with its cluster color
X_compressed = mu[z].reshape(X.shape).astype(np.uint8)

# Show the images side by side
compare_images(X, X_compressed, k)
```



Gaussian Mixture Models & EM

Now you will repeat the same exercise with GMMs.

In [5]:

```
def gmm_log_probability(X, pi, mu, sigma):
    """Compute the joint log-probabilities for each data point
and component.

    Parameters
    ----------
    X : np.array of size N x D
        where N is the number of samples and D is the data dim
ensionality
    pi : np.array of size k
        Prior weight of each component
    mu : np.array of size k x D
        Mean vectors of the k Gaussian component distributions
    sigma : np.array of size k x D x D
        Covariance matrices of the k Gaussian component distri
butions

    Returns
    -------
    P : np.array of shape N x k
        P[i, j] is the joint log-probability of data point i u
nder component j
    """

    # TODO: Compute P
    K = len(pi)
    P = np.log(np.sum([pi[i]*np.random.multivariate_normal(mu[
i],cov[i]).pdf(X[0]) for i in range(K)]))
    print(P)
    log_likelihoods = []
    log_likelihoods.append(np.log(np.sum([k*multivariate_norma
l(mu[i],sigma[j]).pdf(X) for k,i,j in zip(pi,range(len(mu)),ra
nge(len(sigma)))])))
    print(log_likelihoods)
    return P


def em(X, k, tol=0.001):
    """Fit a Gaussian mixture model with k components to X.
```

```
    Parameters

    ----------
    X : np.array of size N x D
        where N is the number of samples and D is the data dim
ensionality
    k : int
        Number of clusters
    tol : float
        Converge when the increase in the mean of the expected
joint log-likelihood
        is lower than this

    The algorithm should stop when the relative improvement in
the optimization
    objective is less than rtol.

    Returns
    -------
    pi : np.array of size k
        Prior weight of each component
    mu : np.array of size k x D
        Mean vectors of the k Gaussian component distributions
    sigma : np.array of size k x D x D
        Covariance matrices of the k Gaussian component distri
butions
    """


    return pi, mu, sigma
```

In [6]:

```
# Fit the GMM
k = 5
pi, mu, sigma = em(X.reshape((-1, 3)), k)

# Determine the most likely cluster of each pixel
log_p = gmm_log_probability(X.reshape((-1, 3)), pi, mu, sigma)
z = log_p.argmax(axis=1)

# Replace each pixel with its cluster mean
X_compressed = mu[z].reshape(X.shape).astype(np.uint8)

# Show the images side by side
compare_images(X, X_compressed, k)
```

```
---------------------------------------------------
-------------------------
NameError
Traceback (most recent call last)
<ipython-input-6-5c2c54e53fa0> in <module>
      1 # Fit the GMM
      2 k = 5
----> 3 pi, mu, sigma = em(X.reshape((-1, 3)), k)
      4
      5 # Determine the most likely cluster of eac
h pixel

<ipython-input-5-1761a0cee8fe> in em(X, k, tol)
     56
     57
---> 58     return pi, mu, sigma

NameError: name 'pi' is not defined
```

Sampling Unseen Datapoints

You have trained a generative model which allows you to sample from the learned distribution. In this section, you sample new images.

In [ ]:

```
def gmm_sample(N, pi, mu, sigma):
    """Sample N data points from a Gaussian mixture model.

    Parameters
    ----------
    N : int
        Number of data points to sample
    pi : np.array of size k
        Prior weight of each component
    mu : np.array of size k x D
        Mean vectors of the k Gaussian component distributions
    sigma : np.array of size k x D x D
        Covariance matrices of the k Gaussian component distri
butions

    Returns
    -------
    X : np.array of shape N x D
    """

    # TODO: Sample X

    return X
```

In [ ]:

```
# Sample pixels and reshape them into the size of the original
image
X_sampled = gmm_sample(np.prod(X.shape[:-1]), pi, mu, sigma).r
eshape(X.shape).astype(np.uint8)

# Compare the original and the sampled image
compare_images(X, X_sampled, "Sampled")
```

Explain what you see in the generated images. (1-3 sentences)