MACHINE LEARNING

Machine Learning Exercises In Python, Part 7

14TH JULY 2016

*This post is part of a series covering the exercises from Andrew Ng's **machine learning** class on Coursera. The original code, exercise text, and data files for this post are available **here**.*

We're now down to the last two posts in this series! In this installment we'll cover two fascinating topics: K-means clustering and principal component analysis (PCA). K-means and PCA are both examples of **unsupervised learning** techniques. Unsupervised learning problems do not have any label or target for us to learn from to make predictions, so unsupervised algorithms instead attempt to learn some interesting structure in the data itself. We'll first implement K-means and see how it can be used it to compress an image. We'll also experiment with PCA to find a low-dimensional representation of images of faces. As always, it helps to follow along using the exercise text for the course (posted **here**).

K-Means Clustering

To start out we're going to implement and apply K-means to a simple 2-dimensional data set to gain some intuition about how it works. K-means is an iterative, unsupervised clustering algorithm that groups similar instances together into clusters. The algorithm starts by guessing the initial centroids for each cluster, and then repeatedly assigns instances to the nearest cluster and re-computes the centroid of that cluster. The first piece that we're going to implement is a function that finds the closest centroid for each instance in the data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
from scipy.io import loadmat
%matplotlib inline

def find_closest_centroids(X, centroids):
    m = X.shape[0]
    k = centroids.shape[0]
    idx = np.zeros(m)

    for i in range(m):
        min_dist = 1000000
        for j in range(k):
            dist = np.sum((X[i,:] - centroids[j,:]) ** 2)
            if dist < min_dist:
                min_dist = dist
                idx[i] = j
    return idx
```

Let's test the function to make sure it's working as expected. We'll use the test case provided in the exercise.

```
data = loadmat('data/ex7data2.mat')
X = data['X']
initial_centroids = initial_centroids = np.array([[3, 3], [6, 2], [8, 5]])

idx = find_closest_centroids(X, initial_centroids)
idx[0:3]
```

```
array([ 0.,  2.,  1.])
```

The output matches the expected values in the text (remember our arrays are zero-indexed instead of one-indexed so the values are one lower than in the exercise). Next we need a function to compute the centroid of a cluster. The centroid is simply the mean of all of the examples currently assigned to the cluster.

```
def compute_centroids(X, idx, k):
    m, n = X.shape
    centroids = np.zeros((k, n))

    for i in range(k):
        indices = np.where(idx == i)
        centroids[i,:] = (np.sum(X[indices,:], axis=1) / len(indices[0])).ravel(

    return centroids
```

```
compute_centroids(X, idx, 3)
```

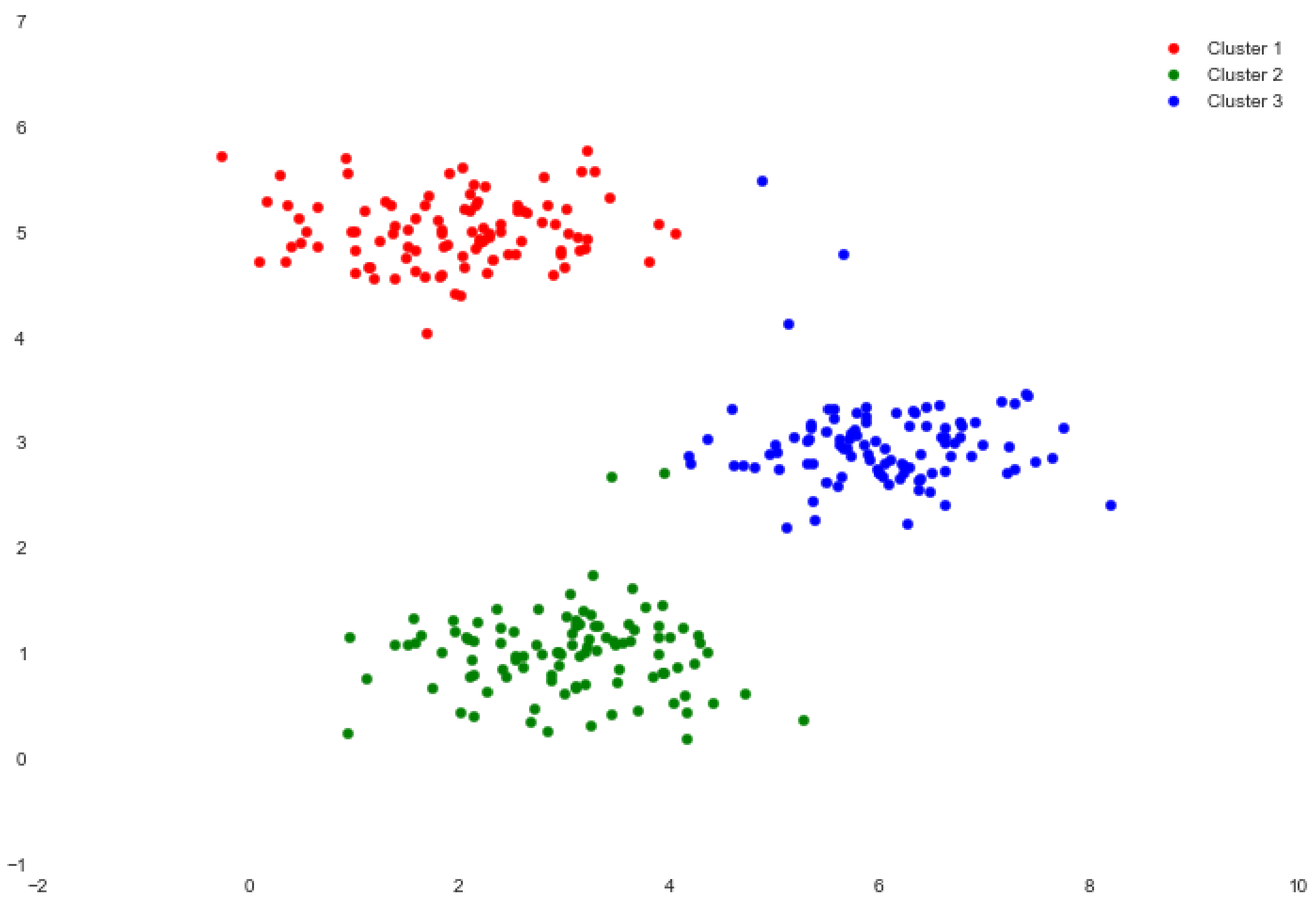
```
array([[ 2.42830111,  3.15792418],
       [ 5.81350331,  2.63365645],
       [ 7.11938687,  3.6166844 ]])
```

This output also matches the expected values from the exercise. So far so good. The next part involves actually running the algorithm for some number of iterations and visualizing the result. This step was implmented for us in the exercise, but since it's not that complicated I'll build it here from scratch. In order to run the algorithm we just need to alternate between assigning examples to the nearest cluster and re-computing the cluster centroids.

```
def run_k_means(X, initial_centroids, max_iters):
    m, n = X.shape
    k = initial_centroids.shape[0]
    idx = np.zeros(m)
    centroids = initial_centroids

    for i in range(max_iters):
        idx = find_closest_centroids(X, centroids)
        centroids = compute_centroids(X, idx, k)

    return idx, centroids

idx, centroids = run_k_means(X, initial_centroids, 10)
```

We can now plot the result using color coding to indicate cluster membership.

```
cluster1 = X[np.where(idx == 0)[0],:]
cluster2 = X[np.where(idx == 1)[0],:]
cluster3 = X[np.where(idx == 2)[0],:]

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(cluster1[:,0], cluster1[:,1], s=30, color='r', label='Cluster 1')
ax.scatter(cluster2[:,0], cluster2[:,1], s=30, color='g', label='Cluster 2')
ax.scatter(cluster3[:,0], cluster3[:,1], s=30, color='b', label='Cluster 3')
ax.legend()
```

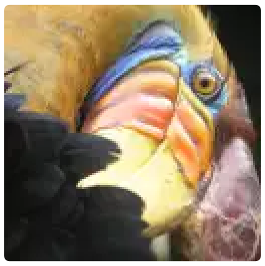


One step we skipped over is a process for initializing the centroids. This can affect the convergence of the algorithm. We're tasked with creating a function that selects random examples and uses them as the initial centroids.

```
def init_centroids(X, k):
    m, n = X.shape
    centroids = np.zeros((k, n))
    idx = np.random.randint(0, m, k)

    for i in range(k):
        centroids[i,:] = X[idx[i],:]

    return centroids

init_centroids(X, 3)
```

```
array([[ 1.15354031,  4.67866717],
       [ 6.27376271,  2.24256036],
       [ 2.20960296,  4.91469264]])
```

Our next task is to apply K-means to image compression. The intuition here is that we can use clustering to find a small number of colors that are most representative of the image, and map the original 24-bit colors to a lower-dimensional color space using the cluster assignments. Here's the image we're going to compress.



The raw pixel data has been pre-loaded for us so let's pull it in.

```
image_data = loadmat('data/bird_small.mat')
```

```
image_data
```

```
{'A': array([[[219, 180, 103],
         [230, 185, 116],
         [226, 186, 110],
         ...,
         [ 14,  15,  13],
         [ 13,  15,  12],
         [ 12,  14,  12]],
        ...,
        [[ 15,  19,  19],
         [ 20,  20,  18],
         [ 18,  19,  17],
         ...,
         [ 65,  43,  39],
         [ 58,  37,  38],
         [ 52,  39,  34]]], dtype=uint8),
 '__globals__': [],
 '__header__': 'MATLAB 5.0 MAT-file, Platform: GLNXA64, Created on: Tue Jun  5 0
 '__version__': '1.0'}
```

We can quickly look at the shape of the data to validate that it looks like what we'd expect for an image.

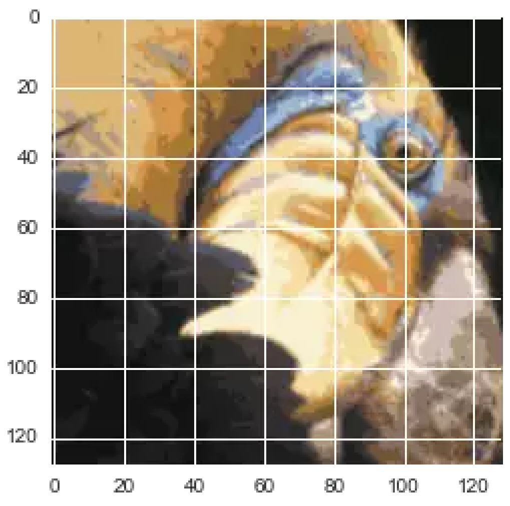
```
A = image_data['A']

A.shape
```

```
(128L, 128L, 3L)
```

Now we need to apply some pre-processing to the data and feed it into the K-means algorithm.

```
# normalize value ranges

A = A / 255.
```

```
# reshape the array
X = np.reshape(A, (A.shape[0] * A.shape[1], A.shape[2]))

# randomly initialize the centroids
initial_centroids = init_centroids(X, 16)

# run the algorithm
idx, centroids = run_k_means(X, initial_centroids, 10)

# get the closest centroids one last time
idx = find_closest_centroids(X, centroids)

# map each pixel to the centroid value
X_recovered = centroids[idx.astype(int),:]

# reshape to the original dimensions
X_recovered = np.reshape(X_recovered, (A.shape[0], A.shape[1], A.shape[2]))

plt.imshow(X_recovered)
```

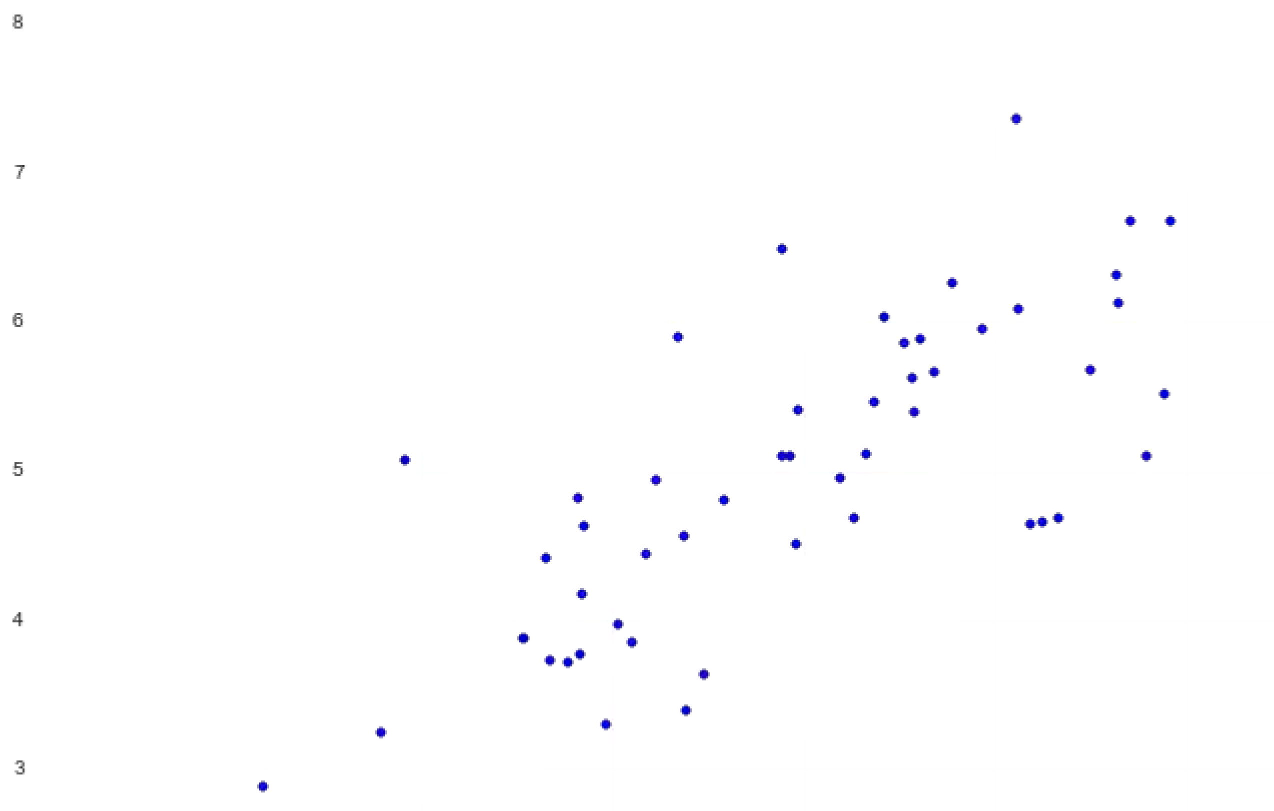


Cool! You can see that we created some artifacts in the compression but the main features of the image are still there despite mapping the original image to only 16 colors. That's it for K-means. We'll now move on to principal component analysis.

Principal Component Analysis

PCA is a linear transformation that finds the "principal components", or directions of greatest variance, in a data set. It can be used for dimension reduction among other things. In this exercise we're first tasked with implementing PCA and applying it to a simple 2-dimensional data set to see how it works. Let's start off by loading and visualizing the data set.

```
data = loadmat('data/ex7data1.mat')
X = data['X']

fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(X[:, 0], X[:, 1])
```



The algorithm for PCA is fairly simple. After ensuring that the data is normalized, the output is simply the singular value decomposition of the covariance matrix of the original data. Since numpy already has built-in functions to calculate the covariance and SVD of a matrix, we'll use those rather than build from scratch.

```
def pca(X):
    # normalize the features
    X = (X - X.mean()) / X.std()

    # compute the covariance matrix
    X = np.matrix(X)
    cov = (X.T * X) / X.shape[0]

    # perform SVD
    U, S, V = np.linalg.svd(cov)

    return U, S, V

U, S, V = pca(X)
U, S, V
```

```
(matrix([[-0.79241747, -0.60997914],
         [-0.60997914,  0.79241747]]),
 array([ 1.43584536,  0.56415464]),
 matrix([[-0.79241747, -0.60997914],
         [-0.60997914,  0.79241747]]))
```

Now that we have the principal components (matrix U), we can use these to project the original data into a lower-dimensional space. For this task we'll implement a function that computes the projection and selects only the top K components, effectively reducing the number of dimensions.

```
def project_data(X, U, k):
    U_reduced = U[:,:k]
    return np.dot(X, U_reduced)

Z = project_data(X, U, 1)
Z
```

```
matrix([[-4.74689738],
        [-7.15889408],
        [-4.79563345],
        [-4.45754509],
        [-4.80263579],
        ...,
        [-6.44590096],
        [-2.69118076],
        [-4.61386195],
        [-5.88236227],
        [-7.76732508]])
```

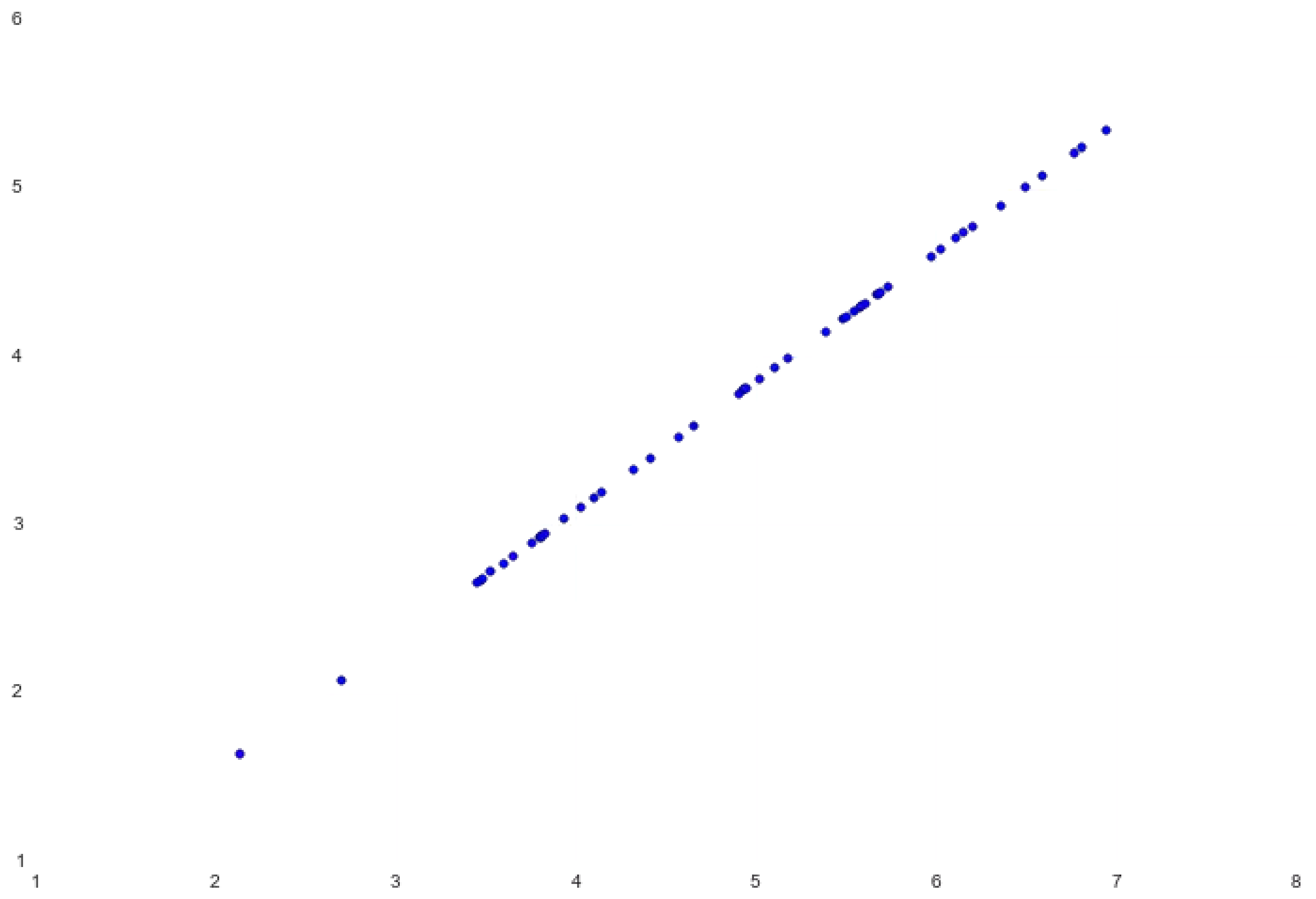
We can also attempt to recover the original data by reversing the steps we took to project it.

```
def recover_data(Z, U, k):
    U_reduced = U[:,:k]
    return np.dot(Z, U_reduced.T)

X_recovered = recover_data(Z, U, 1)
X_recovered
```

```
matrix([[ 3.76152442,  2.89550838],
        [ 5.67283275,  4.36677606],
        [ 3.80014373,  2.92523637],
        [ 3.53223661,  2.71900952],
        [ 3.80569251,  2.92950765],
        ...,
        [ 5.10784454,  3.93186513],
        [ 2.13253865,  1.64156413],
        [ 3.65610482,  2.81435955],
        [ 4.66128664,  3.58811828],
        [ 6.1549641 ,  4.73790627]])
```

If we then attempt to visualize the recovered data, the intuition behind how the algorithm works becomes really obvious.

```
fig, ax = plt.subplots(figsize=(12,8))
ax.scatter(X_recovered[:, 0], X_recovered[:, 1])
```



Notice how the points all seem to be compressed down to an invisible line. That invisible line is essentially the first principal component. The second principal component, which we cut when we reduced the data to one dimension, can be thought of as the variation orthogonal to that line. Since we lost that information, our reconstruction can only place the points relative to the first principal component.

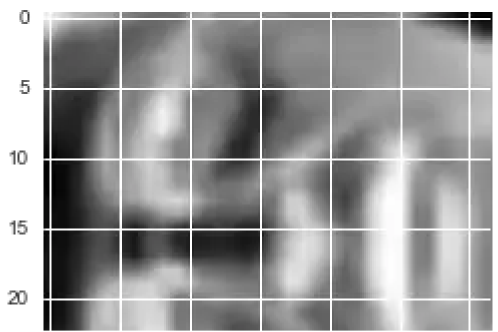
Our last task in this exercise is to apply PCA to images of faces. By using the same dimension reduction techniques we can capture the "essence" of the images using much less data than the original images.

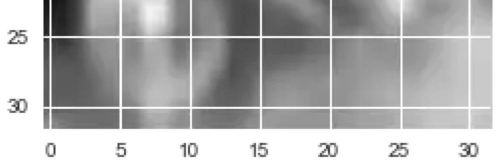
```
faces = loadmat('data/ex7faces.mat')
X = faces['X']
X.shape
```

```
(5000L, 1024L)
```

The exercise code includes a function that will render the first 100 faces in the data set in a grid. Rather than try to re-produce that here, you can look in the exercise text for an example of what they look like. We can at least render one image fairly easily though.

```
face = np.reshape(X[3,:], (32, 32))
plt.imshow(face)
```



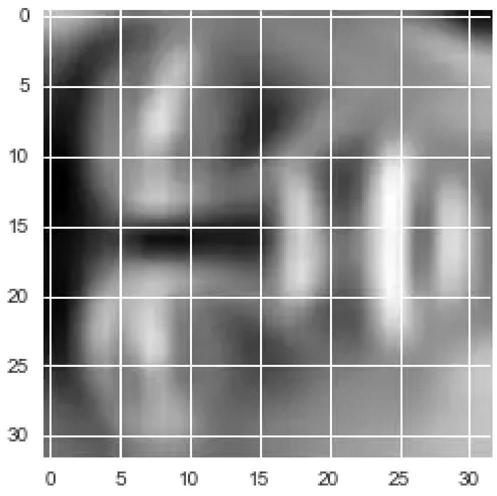


Yikes, that looks awful! These are only 32 x 32 grayscale images though (it's also rendering sideways, but we can ignore that for now). Our next step is to run PCA on the faces data set and take the top 100 principal components.

```
U, S, V = pca(X)
Z = project_data(X, U, 100)
```

Now we can attempt to recover the original structure and render it again.

```
X_recovered = recover_data(Z, U, 100)
face = np.reshape(X_recovered[3,:], (32, 32))
plt.imshow(face)
```



Notice that we lost some detail, though not as much as you might expect for a 10x reduction in the number of dimensions.

That concludes exercise 7! In the final exercise we'll implement algorithms for anomaly detection and build a recommendation system using collaborative filtering.

Follow me on twitter to get new post updates.

Follow @jdwittenauer

MACHINE LEARNING DATA SCIENCE DATA VISUALIZATION

AUTHOR

John Wittenauer

Data scientist, engineer, author, investor, entrepreneur

COMMENTS

0 Comments Curious Insight 1 Login

Recommend Share Sort by Best

Start the discussion…

Be the first to comment.