《数据科学与工程算法基础》实践报告

报告题目：使用 PCA 实现图像压缩

姓　　名：张雯怡

学　　号：10195501425

完成日期：2021.11.25

提供中英文摘要

摘要 [中文]：

PCA（Principal Component Analysis）是一种常用的数据分析方法。PCA 通过线性变换将原始数据变换为一组各维度线性无关的表示，可用于提取数据的主要特征分量，常用于高维数据的降维。本实验一共使用了两种 PCA 主成分分析方法，对数据集中 300 张图片其中的 100 张进行了图像压缩。其一是列/行向量上的 PCA，其二是双向二维 PCA（2D-2D PCA）算法，并对其进行了图像压缩前后的信息丢失率计算。

Abstract [英语]

Principal component analysis (PCA) can be used to reduce the dimensions of the matrix (image) and project those new dimensions to reform the image that retains its qualities but is smaller in k-weight. In this experiment, two PCA principal component analysis methods are used to compress 100 of the 300 images in the data set. One is PCA on column / row vector, the other is bidirectional two-dimensional PCA (2D-2D PCA) algorithm, and the information loss rate before and after image compression is calculated.

一、 项目概述（阐明该项目的科学和应用价值，以及相关工作进展并描述项目的主要内容）

一般情况下，在数据挖掘和机器学习中，数据被表示为向量。我们知道，很多机器学习算法的复杂度和数据的维数有着密切关系，甚至与维数呈指数级关联。而在实际机器学习中处理成千上万甚至几十万维的情况也并不罕见，在这种情况下，机器学习的资源消耗是不可接受的，因此我们必须对数据进行降维。降维就是一种对高维度特征数据预处理方法。降维是将高维度的数据保留下最重要的一些特征，去除噪声和不重要的特征，从而实现提升数据处理速度的目的。降维当然意味着信息的丢失，不过鉴于实际数据本身常常存在的相关性，我们可以想办法在降维的同时将信息的损失尽量降低。在实际的生产和应用中，降维在一定的信息损失范围内，可以为我们节省大量的时间和成本。

二、 问题描述（问题定义）

然而，在众多维度的维度的数据中，删除哪一列损失的信息才最小？亦或根本不是单纯删除几列，而是通过某些变换将原始数据变为更少的列但又使得丢失的信息最小？到底如何度量丢失信息的多少？如何根据原始数据决定具体的降维操作步骤？PCA（Principal Component Analysis）就是一种常用的用于解决如上问题的数据分析方法，PCA 是一种具有严格数学基础并且已被广泛采用的降维方法。

在本实验中，一共有 300 张图片的数据集，本实验将使用 PCA 对其中的一百张进行图像压缩，尽量在压缩后还原图片原先的样子，即降低压缩的损失。

三、 方法（问题解决步骤和实现细节）

（一）列/行 PCA 压缩

降维问题的优化目标：**将一组 N 维向量降为 K 维（K 大于 0，小于 N），其目标是选择 K 个单位（模为 1）正交基，使得原始数据变换到这组基上后，各字段两两间协方差为 0，而字段的方差则尽可能大（在正交的约束下，取最大的 K 个方差）**。

以下是 PCA 降维的步骤：

设有 m 条 n 维数据。

1）将原始数据按列组成 n 行 m 列矩阵 X

2）将 X 的每一列（代表一个属性字段）进行零均值化（中心化），即减去这一行的均值

```
    mean = np.array([np.mean(data[:,i]) for i in range(n_features)])
    normal_img = img - mean
```

3）求出协方差矩阵

```
    matrix_ = np.dot(np.transpose(normal_img),normal_img)   # 协方差矩阵
```

4）求出协方差矩阵的特征值及对应的特征向量

```
    eig_val,eig_vec = np.linalg.eig(matrix_)
```

5）将特征向量按对应特征值大小从上到下按行排列成矩阵，取前 k 行组成矩阵 P

```
    eigIndex = np.argsort(eig_val)
    eigVecIndex = eigIndex[:-(k+1):-1]
    feature = eig_vec[:,eigVecIndex]
```

6）即为降维到 k 维后的数据

```
    new_data = np.dot(normal_img,feature)
```

PCA 部分代码如下：

```
def pca(img, k):
    n_samples,n_features = img.shape
    mean = np.array([np.mean(img[:,i]) for i in range(n_features)])
    normal_img = img - mean
```

```
    matrix_ = np.dot(np.transpose(normal_img),normal_img)   # 协方差矩阵
    eig_val,eig_vec = np.linalg.eig(matrix_)
    #print(matrix_.shape)
    #print(eig_val)
    eigIndex = np.argsort(eig_val)
    eigVecIndex = eigIndex[:-(k+1):-1]
    feature = eig_vec[:,eigVecIndex]
    new_data = np.dot(normal_img,feature)
    # 将降维后的数据映射回原空间
    compressed_img = np.dot(new_data,np.transpose(feature))+ mean
    # print(compressed_img)
    newImage = Image.fromarray(compressed_img*255)
    newImage.show()
    return compressed_img
```

最后对压缩前后的图像计算信息丢失率：

```
def error(img, compressed_img):
    sum1 = 0
    sum2 = 0
    D = img - compressed_img
    for i in range(img.shape[0]):
        sum1 += np.dot(img[i],img[i])
        sum2 += np.dot(D[i], D[i])
    error = sum2/sum1
    print(error)
```

另外，只要将中心化求均值时

```
    mean = np.array([np.mean(data[:,i]) for i in range(n_features)])
    normal_img = img - mean
```

改为

```
    mean = np.array([np.mean(data[i,:]) for i in range(n_features)])
    normal_img = img - mean
```

即可进行图像的行上压缩。

（二）双向二维 PCA（2D-2D PCA）

相较于传统 PCA，2D PCA 不用将图片转换为 1D 向量，极大的减少了数据的维度。基本思路和 PCA 相同，也是将数据投影到某组基上，然后使得投影之后数据的协方差矩阵成为对角阵。投影后数据的协方差矩阵为：

$$\begin{aligned}\mathbf{S}_x &= E(\mathbf{Y}-E\mathbf{Y})(\mathbf{Y}-E\mathbf{Y})^T = E[\mathbf{AX}-E(\mathbf{AX})][\mathbf{AX}-E(\mathbf{AX})]^T \\ &= E[(\mathbf{A}-E\mathbf{A})\mathbf{X}][(\mathbf{A}-E\mathbf{A})\mathbf{X}]^T.\end{aligned}$$

这时发现中间的部分恰好为数据矩阵 A 的协方差矩阵，因此我们将其单独定义为 Gt，Gt 即代表数据矩阵 A 的协方差矩阵：

$$\mathbf{G}_t = E[(\mathbf{A}-E\mathbf{A})^T(\mathbf{A}-E\mathbf{A})].$$

Gt 是一个 n*n 的方阵。

对于 M 张训练样本 A（j）（j = 1，2，……，M），可得其协方差矩阵为：

$$\mathbf{G}_t = \frac{1}{M}\sum_{j=1}^{M}(\mathbf{A}_j-\bar{\mathbf{A}})^T(\mathbf{A}_j-\bar{\mathbf{A}}).$$

此时，可将之前投影矩阵 X 的目标函数简化为：

$$J(\mathbf{X}) = \mathbf{X}^T\mathbf{G}_t\mathbf{X},$$

2D PCA，先通过样本集得到样本集的协方差矩阵，只是此时的协方差矩阵不像传统 PCA 先转化为一维矩阵，而是直接将所有样本累加再求平均得到的，然后对得到的协方差矩阵进行特征值分解，将对应的特征向量排列起来即得到了最优的投影矩阵。

而 2D-2D PCA，是指在行方向和列方向都进行一次 2DPCA 运算。

协方差矩阵是不变的，为

$$\mathbf{G} = \frac{1}{M}\sum_{K=1}^{M}(\mathbf{A}_k - \bar{\mathbf{A}})^{\mathrm{T}}(\mathbf{A}_k - \bar{\mathbf{A}}).$$

可以进一步表示为：

$$\mathbf{G} = \frac{1}{M}\sum_{K=1}^{M}\sum_{i=1}^{m}(\mathbf{A}_k^{(i)} - \bar{\mathbf{A}}^{(i)})^{\mathrm{T}}(\mathbf{A}_k^{(i)} - \bar{\mathbf{A}}^{(i)}).$$

观察协方差矩阵发现，其实它和之前最原始的 2DPCA 的协方差矩阵 Gt 的表示形式本质上是一样的，只不过多了行方向上的叠加而已，因此我们成之前的 2DPCA 是工作在行方向上的 PCA 变换。接下来对协方差矩阵 G 进行特征值分解等一系列操作，即可得到相应的映射矩阵，不过这里得到的是行映射矩阵 X（又称左映射矩阵）。

以此类推，接下来就应该介绍工作在列方向上的 2DPCA 了，原理和上面的类似，只不过上面公式总矩阵 A 采用的是按行排列，接下来我们将其改为按列排列, 相应的，此时的训练集协方差矩阵应该调整为如下形式：

$$\mathbf{G} = \frac{1}{M}\sum_{k=1}^{M}\sum_{j=1}^{n}(\mathbf{A}_k^{(j)} - \bar{\mathbf{A}}^{(j)})(\mathbf{A}_k^{(j)} - \bar{\mathbf{A}}^{(j)})^{\mathrm{T}}$$

行映射的协方差矩阵和列映射的协方差矩阵区别就在于转置相乘的顺序上。

得到行、列方向上的投影矩阵 X、Z 之后，其联合映射的形式如下：

$$\mathbf{C} = \mathbf{Z}^{\mathrm{T}}\mathbf{A}\mathbf{X}.$$

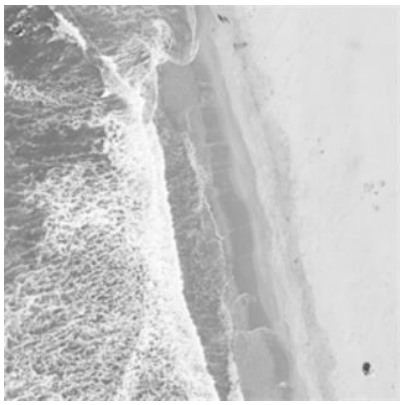
对于原矩阵的重构结果如下：

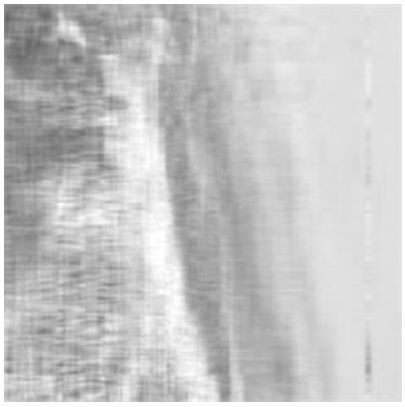
$$\hat{\mathbf{A}} = \mathbf{Z}\mathbf{C}\mathbf{X}^{\mathrm{T}}.$$

四、 实验结果（验证提出方法的有效性和高效性）

（一）列/行 PCA 压缩

首先将图片转化为灰度图，然后使用 PCA 进行列压缩，这里先取 k = 10，即保留 10 个主成分，压缩前后如下：





最后对压缩前后的图像计算信息丢失率：

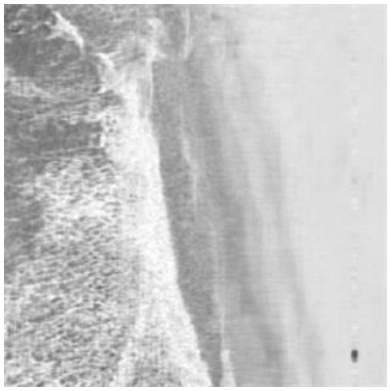
```
def error(img, compressed_img):
    sum1 = 0
    sum2 = 0
    D = img - compressed_img
    for i in range(img.shape[0]):
        sum1 += np.dot(img[i],img[i])
        sum2 += np.dot(D[i], D[i])
    error = sum2/sum1
    print(error)
```

```
if __name__ == '__main__':
    path = './Images/beach/beach00.tif'
    data = loadImage(path)
    k = 10
    recdata = pca(data,k)
    error(data,recdata)
```

```
0.0039850495047639165
```

可以看到当 k = 10 时，99.6%以上的图像信息在压缩后被保留。

再取 k = 30，压缩后图像如下：

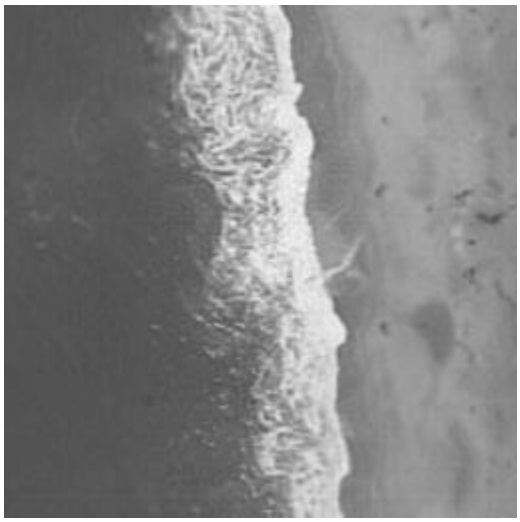


```
if __name__ == '__main__':
    path = './Images/beach/beach00.tif'
    data = loadImage(path)
    k = 30
    recdata = pca(data,k)
    error(data,recdata)
```

```
0.0015078578511876738
```

当 k = 30 时，可以看到原图像信息几乎被保留下来，信息丢失率仅为 0.15%。

（二）双向二维 PCA（2D-2D PCA）



五、 结论（对使用的方法可能存在的不足进行分析，以及未来可能的研究方向进行讨论）

由于图像本身为一个二维矩阵（以灰度图为例），传统 PCA 在构造协方差矩阵时需要先将图像转换成行向量的形式，然后将多个样本图片对应的多个行向量组成一个大规模的协方差矩阵，然后求解此协方差矩阵的特征值及特征向量来构建映射矩阵。这里就存在两个问题：（1）每次进行 PCA 变换前都需要将二维矩阵（图片）转换为一维矩阵（行向量），也就避免不了对图片进行遍历，在图片数量很多、尺寸较大的情况下，这一步是很耗时的。（2）得到的协方差矩阵的尺寸通常情况下也很大。

相较于传统 PCA，2D PCA 基本思路和 PCA 相同，也是将数据投影到某组基上，然后使得投影之后数据的协方差矩阵成为对角阵，不用将图片转换为 1D 向量，而是直接将所有样本累加再求平均得到的，然后对得到的协方差矩阵进行特征值分解，将对应的特征向量排列起来即得到了最优的投影矩阵，极大的减少了数据的维度。

2D-2D PCA 与原始的 2DPCA 先比最大的优势在于其映射结果 X 的维数很小，为 k*k，进一步完成了对 2DPCA 的降维操作。

源代码：

（一）PCA

```
from PIL import Image
import numpy as np

def getImage(path):
    image = Image.open(path)
    image = img.convert("L")
    # PIL 中图像的 size 是（宽，高）
    width = image.size[0]   # width 取 size 的第一个值
    height = image.size[1]   # height 取第二个
    img = image.getdata()
    # 为了避免溢出，这里对数据进行一个缩放，缩小 100 倍
    img = np.array(img).reshape(height,width)/255
    # 查看原图的话，需要还原数据
    new_img = Image.fromarray(img*255)
    new_img.show()
    return img

def pca(img, k):
    n_samples,n_features = img.shape
    mean = np.array([np.mean(img[:,i]) for i in range(n_features)])
    normal_img = img - mean
    matrix_ = np.dot(np.transpose(normal_img),normal_img)   # 协方差矩阵
    eig_val,eig_vec = np.linalg.eig(matrix_)
    #print(matrix_.shape)
    #print(eig_val)
    eigIndex = np.argsort(eig_val)
    eigVecIndex = eigIndex[:-(k+1):-1]
    feature = eig_vec[:,eigVecIndex]
    new_data = np.dot(normal_img,feature)
    # 将降维后的数据映射回原空间
    compressed_img = np.dot(new_data,np.transpose(feature))+ mean
    # print(compressed_img)
    newImage = Image.fromarray(compressed_img*255)
    newImage.show()
    return compressed_img

def error(img, compressed_img):
    sum1 = 0
    sum2 = 0
    D = img - compressed_img
```

```
    for i in range(img.shape[0]):
        sum1 += np.dot(img[i],img[i])
        sum2 += np.dot(D[i], D[i])
    error = sum2/sum1
    print(error)

if __name__ == '__main__':
    path = './Images/beach/beach00.tif'
    img = loadImage(path)
    k = 30
    compressed_img = pca(img,k)
    error(img,compressed_img)
```

（二）2D-2D PCA

```
# a implementation of 2D^2 PCA algorithm

import numpy as np
from PIL import Image

def PCA2D_2D(samples, row_top, col_top):
    '''samples are 2d matrices'''
    size = samples[0].shape
    # m*n matrix
    mean = np.zeros(size)

    for s in samples:
        mean = mean + s

    # get the mean of all samples
    mean /= float(len(samples))

    # n*n matrix
    cov_row = np.zeros((size[1],size[1]))
    for s in samples:
        diff = s - mean;
        cov_row = cov_row + np.dot(diff.T, diff)
    cov_row /= float(len(samples))
    row_eval, row_evec = np.linalg.eig(cov_row)
    # select the top t evals
    sorted_index = np.argsort(row_eval)
    # using slice operation to reverse
    X = row_evec[:,sorted_index[:-row_top-1 : -1]]
```

```
    # m*m matrix
    cov_col = np.zeros((size[0], size[0]))
    for s in samples:
        diff = s - mean;
        cov_col += np.dot(diff,diff.T)
    cov_col /= float(len(samples))
    col_eval, col_evec = np.linalg.eig(cov_col)
    sorted_index = np.argsort(col_eval)
    Z = col_evec[:,sorted_index[:-col_top-1 : -1]]

    return X, Z


samples = []
for i in range(0,99):
    im = Image.open('./Images/beach/beach0'+str(i)+'.tif')
    im = im.convert("L")
    im_data  = np.empty((im.size[1], im.size[0]))
    for j in range(im.size[1]):
        for k in range(im.size[0]):
            R = im.getpixel((k, j))
            im_data[j,k] = R/255.0
    samples.append(im_data)

X, Z = PCA2D_2D(samples, 90, 90)

res = np.dot(Z.T, np.dot(samples[0], X))
res = np.dot(Z, np.dot(res, X.T))

row_im = Image.new('L', (res.shape[1], res.shape[0]))
y=res.reshape(1, res.shape[0]*res.shape[1])

row_im.putdata([int(t*255) for t in y[0].tolist()])
row_im.save('X.png')
```