模式识别与统计学习
实验四：*Principle Component Analysis*实验报告

郭宇航 2016100104014

一　实验原理

1.1　问题引入

在许多领域的研究和应用中，我们常常能够获取到大量的数据，通过大量的数据我们可以进行分析寻找规律。然而在很多的情形下我们发现数据中的大量的变量之间存在相关性，从而增加了问题分析的复杂度，如果减少变量同时不至于带来太多的信息的损失成为了一个很大的研究问题。这一研究问题也被称为数据降维(*Diemnsion Reduction*)，PCA算法就是一种非常重要的降维方法，在处理数据压缩消除冗余以及数据去噪等领域有着广泛的应用。

*Principle Components Analysis*顾名思义，找到数据中最主要的成分，用这些最主要的成分去代替所有的数据。具体来说我们的这个问题就可以转述为：现在假设我们的数据集是n维的，同时有m个这样的样本数据：$x^{(1)}, x^{(2)}, \cdots, x^{(m)}$。我们希望将这$m$个数据从$n$维降低到$n'$维，如果实现了毫无疑问这一过程中会带来一定的信息损失，如何去得到损失最小的结果，这就成为了核心问题，下面从两个不同的角度给出阐释和推导。

1.2　最大投影方差

观察这样一张图片(图1)，图中蓝色的点为原数据点（以二维空间为例），假设原数据只有两个变量，红色的点为在第一个方向上的投影，绿色的点为在第二个方向上的投影。 现在考虑一个简单

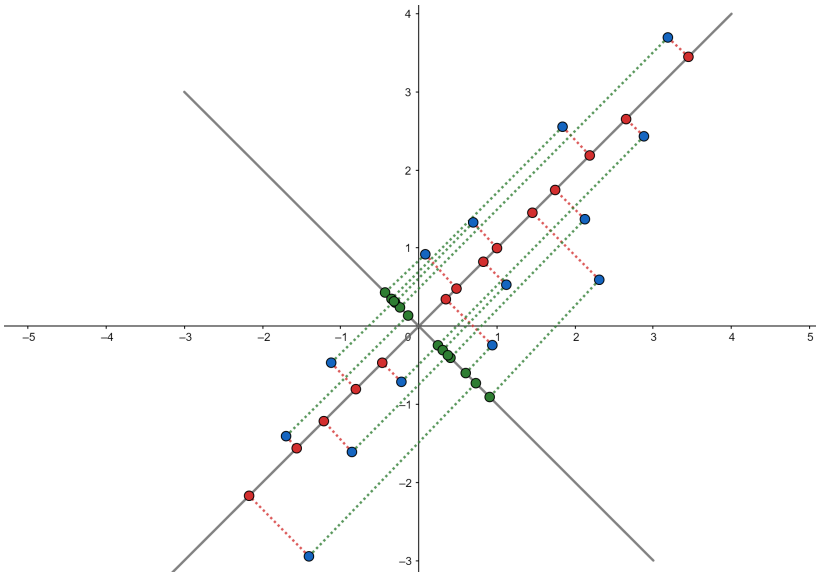

图 1: 二维空间数据投影示意图

的问题，我们的图中共有14个二维数据，现在我们希望将这14个数据进行降维，降为14个一维的数据。如何实现这样的目标呢？一个很容易想到的思路就是进行投影，而进行投影之后，将投影的方向作为一维数轴，所有的数据就表出为一维的数据了，然而这里有个问题是，什么样的方向最适合做这个投影呢？回到之前我们说降维的好坏的评价手段上，我们是希望降维之后的信息损失最小。从这个角度来看上面的图1，是红色的投影方向更好呢还是绿色的投影方向更好呢？其实一眼就能看出来红色的更好，因为在红色的投影方向上，点散布的更加分散，也就是说点与点之间的区分度更大，这样就表明了我们能够保存下来的信息越多，再看绿色的投影方向，我们看到好多的点在进行投影之后变成了同一个点，这显然降维的效果不好，明明二维空间中的两个差距很大的点，降维之后变为了同一点，这显然带来了很大的信息损失，所以绿色的方向是不合适的。

刚刚只是举了两个特例，二维空间中寻找一个投影方向有无数个，现在将这个问题进行泛化的处理，我们的问题并没有变化，还是去寻找一个投影的方向，使得数据在新的轴上散布的范围最大，保留的信息越多。

保留之前的假设，我们有m个n维的样本数据：$\{x^{(1)}, x^{(2)}, \cdots, x^{(m)}\}, x^{(i)} \in \mathbb{R}^n$.首先在进行推导之前我们先对所有的数据进行去中心化和标准化：

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \Rightarrow x^{(i)} := x^{(i)} - \mu \\ \sigma_j^2 &= \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)})^2 \Rightarrow x_j^{(i)} := x_j^{(i)}/\sigma_j \end{aligned} \tag{1}$$

对于新的数据，如果希望降到一维，就去寻找一个方向，这个方向上有一个单位基底$\vec{u}_1$，$\vec{u}_1$满足模为1，有了方向可以计算投影，对于单个数据样本与该方向的投影我们可以表示为：$(x^{(i)})^T\vec{u}_1$，对于所有的样本来说可以得到总投影,因此我们去寻找的这个方向基底可以表示为：

$$\vec{u}_1^* = \underset{\vec{u}_1}{\operatorname{argmax}}[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^T\vec{u}_1] \tag{2}$$

将问题一般化，数据从n维降低到n'维，因此需要选择n'个方向，同时对总投影值取平方后可以用矩阵的形式表示为：

$$\boldsymbol{U}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{U} \tag{3}$$

其中$\boldsymbol{X}\boldsymbol{X}^{\boldsymbol{T}}$表示为数据集的协方差矩阵，此协方差矩阵拥有很好的性质，在实数范围考虑，此矩阵是一个半正定的实对称矩阵。对于任意一个样本$x^{(i)}$，在新坐标系下的投影方差可以表示为$\boldsymbol{U}^T x^{(i)}(x^{(i)})^T\boldsymbol{U}$，由此将问题转为优化问题，具体问题如下：

$$\begin{aligned} \boldsymbol{U}^* &= \underset{\boldsymbol{U}}{\operatorname{argmax}}\, \boldsymbol{U}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{U} \\ &\textbf{s.t.} \quad \boldsymbol{U}^T\boldsymbol{U} = I \quad \boldsymbol{X}\boldsymbol{X}^T_{m\times m} \end{aligned} \tag{4}$$

使用拉格朗日乘数法可以将问题写为：

$$H(\boldsymbol{U}) = tr(\boldsymbol{U}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{U} + \lambda(\boldsymbol{U}^{\boldsymbol{T}}\boldsymbol{U} - I)) \tag{5}$$

对$\boldsymbol{U}$进行求导可以得到：

$$\boldsymbol{X}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{U} + \lambda\boldsymbol{U} = 0 \Rightarrow \boldsymbol{X}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{U} = (-\lambda)\boldsymbol{U} \tag{6}$$

观察上式并结合矩阵特征值分解的知识，不难发现$\boldsymbol{U}$为协方差矩阵$\boldsymbol{X}\boldsymbol{X}^{\boldsymbol{T}}$的$n'$个特征向量张成的矩阵，而$-\lambda$则是由这些特征向量对应的特征值张成的对角阵。因此问题转化为求解协方差矩阵的特征值分解的结果。

回到之前的寻找能够保留最多信息的投影方向的问题上，最佳的投影的方向结合特征值分解的几何意义，不难发现最理想的结果就是特征值最大的特征向量的方向。

1.3 最小投影误差

刚刚我们考虑了一种思路，从投影方向方差最大即能够保存最多的信息的角度进行推导，下面从另一个角度考虑，想要评价这个投影方向是不是良好的，还可以这样理解，所有的数据点到这个投影方向所在的直线的距离加起来最小，说明数据点在投影前后的误差是最小的，同样可以认为这个投影方向能够保留住原始数据最多的信息。从这种思路出发，我们进行理论的推导。

仍然是m个样本数据，假设原始的样本数据x_i在投影之后的变为x_i'，我们的优化目标就变为：

$$\min \sum_{i=1}^{m} ||(x_i' - x_i)||^2 \tag{7}$$

这里明确一个问题，x_i'表示的是在原坐标系下的x_i的投影坐标，现在我们重新定义一个z_i，z_i表示为x_i投影之后在低维空间坐标系中的坐标表示，由此我们可以得到一些关系：$z_i = \boldsymbol{U}^T x_i$，$x_i' = \boldsymbol{U} z_i$.下面对式子7进行代换化简：

$$\begin{aligned}
\sum_{i=1}^{m} ||x_i' - x_i||^2 &= \sum_{i=1}^{m} ||\boldsymbol{U} z_i - x_i||^2 \\
&= \sum_{i=1}^{m} z_i^T \boldsymbol{U}^T \boldsymbol{U} z_i - 2\sum_{i=1}^{m} z_i^T \boldsymbol{U}^T x_i + \sum_{i=1}^{m} x_i^T x_i \\
&= \sum_{i=1}^{m} z_i^T z_i - 2\sum_{i=1}^{m} z_i^T z_i + \sum_{i=1}^{m} x_i^T x_i \\
&= -\sum_{i=1}^{m} z_i^T z_i + \sum_{i=1}^{m} x_i^T x_i \\
&= -tr(\boldsymbol{U^T X X^T U}) + \sum_{i=1}^{m} x_i^T x_i
\end{aligned} \tag{8}$$

由于$\sum_{i=1}^{m} x_i^T x_i$是一个常量，因此目标优化问题变为：

$$\underset{\boldsymbol{U}}{\operatorname{argmax}}(-tr(\boldsymbol{U^T X X^T U})) \quad \textbf{s.t.}\, \boldsymbol{U^T U} = \boldsymbol{I} \tag{9}$$

观察上式9和式6，我们发现无论是从最大投影方差还是最小投影误差的角度来看，最终的优化目标是一致的。因此最终论证了主成分分析算法的合理性。从最小投影误差角度之后的求解过程与最大投影方差一致，这里不再加以赘述。

二 Python代码实现

本次实验中使用到的蝴蝶图片与奇异值分解使用的素材一致，图像如下所示（图2）：

图 2: Butterfly Experiment Figure

2.1 图片读入

上一节给出了本次实验的图片，首先我们对图片的像素点数据进行读入，读入图片像素点矩阵的方式有很多种，这里采用了Pillow库中的Image.open函数进行处理，由于这里我们的图片是一个RGB三通道的图，矩阵的维度是$243 \times 437 \times 3$.实现的代码如下：

```
class Principle_Component_Analysis:
    def load_data(self,filepath):
        input_image = Image.open(filepath)
        width = input_image.size[0]
        height = input_image.size[1]
        print(input_image)
        image_pixel = np.array(input_image)
        #print(image_pixel.shape)
        print('Picture size is %d * %d * %d.' % (image_pixel.shape[0],image_pixel.shape[1],
                                                 image_pixel.shape[2]))
        return image_pixel
```

2.2 数据预处理

数据预处理主要进行数据的去中心化和标准化：

```
    def pre_processing(self,pixel_matrix):
        #decentration
        length,width = pixel_matrix.shape
        new_data = np.zeros([length,width])
        average = np.array(pixel_matrix.mean(axis = 0))
        #print(average)
        #print(pixel_matrix[0] - average)
        for i in range(len(pixel_matrix)):
            new_data[i] = pixel_matrix[i] - average
        #normalize
        #pixel_matrix = pixel_matrix / 255.0
        new_data = new_data / 255.0
        #print(new_data)
        return average,new_data
```

2.3 PCA主函数

PCA算法的主要流程如下(Algorithm 1)： 实现代码如下：

```
    def PCA_main_function(self,pixel_matrix,k):
        corr_matrix = np.dot(pixel_matrix,pixel_matrix.T)
        #print(corr_matrix.shape)
        eigenvalue,eigenvector = eig(corr_matrix)
        #print('eigenvalues: \n{}'.format(eigenvalue))
        #print('eigenvectors: \n{}'.format(eigenvector))
        #find out top-k eigenvalue and eigenvector
        sorted_indices = np.argsort(eigenvalue)
        top_k_eigenvalues = eigenvalue[sorted_indices[:-k-1:-1]]
        #print(top_k_eigenvalues)
        #print(sorted_indices[:-k-1:-1])
        top_k_eigenvector = eigenvector[:,sorted_indices[:-k-1:-1]]
        #print(top_k_eigenvector.shape)
        #pca_processing_data = np.dot(corr_matrix,top_k_eigenvector)
        #print('pca_processing_data',pca_processing_data.shape)
        return top_k_eigenvector,eigenvalue
```

Algorithm 1: Principle Components Analysis Algorithm

输入:

数据样本矩阵：$\boldsymbol{A}_{m\times n}$;

输出:

降维后的样本数据矩阵：$\boldsymbol{A'}$;

初始化数据，对输入的样本矩阵进行去中心化和标准化;

计算样本的协方差矩阵$(\boldsymbol{AA^T})_{m\times m}$;

对协方差矩阵$(\boldsymbol{AA^T})_{m\times m}$进行特征值分解;

取出最大的前n'个特征值以及对应的特征向量$(\vec{u}_1, \vec{u}_2, \cdots, \vec{u}_{n'})$组成特征向量矩阵$\boldsymbol{U}$;

对样本集中的所有样本进行转换：$z_i = \boldsymbol{U}^T x_i$;

输出降维后的样本矩阵$\boldsymbol{A'}$;

2.4 图像绘制以及图片重构

绘制特征值变化曲线以及最后重构图片的代码如下：

```
#plot line chart
def plot_line_chart(self,data_1,data_2,data_3):
    x_lab = list(range(len(data_1)))
    plt.plot(x_lab,data_1,label = 'Red Eigenvalues',color='r')
    plt.plot(x_lab,data_2,label = 'Green Eigenvalues',color='g')
    plt.plot(x_lab,data_3,label = 'Blue Eigenvalues',color='b')
    plt.xlabel('Eigenvalues Index')
    plt.ylabel('Eigenvalues')
    plt.title('Sorted Eigenvalues Line Chart')
    plt.legend()
    plt.show()
def restore_image(self,average,data):
    data = data * 255.0
    for i in range(len(data)):
        data[i] = data[i] + average
    return data
def MatrixToImage(self,data):
    new_im = Image.fromarray(data.astype(np.uint8))
    return new_im
```

三 结果及图形展示

3.1 图形绘制

通过Image.open函数得到的图片读取像素点矩阵的结果如下图3：

```
<PIL.BmpImagePlugin.BmpImageFile image mode=RGB size=437x243 at
 0x2274D7D0940>
Picture size is 243 * 437 * 3.
```

图 3: Image.open Function Load Picture

对协方差矩阵进行特征值分解并根据特征值大小进行排序的变化曲线如下图4： 根据上图4，我们不难发现协方差矩阵的特征值经过排序后只有前30个左右存在较大的值，排序在30之后的特征值

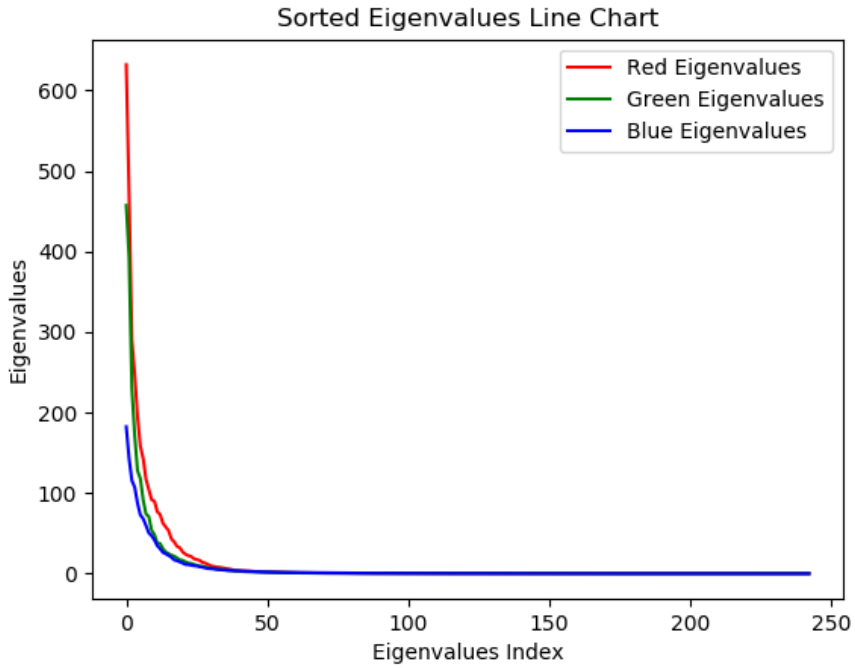

图 4: Sorted Eigenvalues Line Chart

都十分接近0，对图片的构成的贡献成分非常小。换句话说，对于图片像素点构成的协方差矩阵，特征值的下降速度非常快。

下面给出PCA的主函数代码：

```
if __name__ == '__main__':
    pca_test = Principle_Component_Analysis
    file_path = 'butterfly.bmp'
    pixel_matrix = pca_test.load_data(pca_test,file_path)
    length,width,height = pixel_matrix.shape
    #print(length,width,height)
    red_pixel = np.array(pixel_matrix[:,:,0])
    green_pixel = np.array(pixel_matrix[:,:,1])
    blue_pixel = np.array(pixel_matrix[:,:,2])
    #print(red_pixel)
    #print(red_pixel.shape)
    #data pre-processing handle RGB
    average_1,red_pixel_preprocessing = pca_test.pre_processing(pca_test,red_pixel)
    average_2,green_pixel_preprocessing = pca_test.pre_processing(pca_test,green_pixel)
    average_3,blue_pixel_preprocessing = pca_test.pre_processing(pca_test,blue_pixel)
    #print('average',average_1)
    #giving the dimensions number after reduction
    #print('red_pixel',red_pixel_preprocessing)
    dimension_reduction_k = 30
    result_data_red,eigenvalues_red = pca_test.PCA_main_function(pca_test,
                                                red_pixel_preprocessing,
                                                dimension_reduction_k)
    result_data_green,eigenvalues_green = pca_test.PCA_main_function(pca_test,
                                                green_pixel_preprocessing,
                                                dimension_reduction_k)
    result_data_blue,eigenvalues_blue = pca_test.PCA_main_function(pca_test,
                                                blue_pixel_preprocessing,
                                                dimension_reduction_k)
```

```
result_rgb = np.array([result_data_red.T,result_data_green.T,result_data_blue.T])
print('rgb shape',result_rgb)
#print('Red shape:',result_data_red.shape)    #243*50        yasuo: 50 * 437  = 50 * 243
                                                 times 243 *  437
result_data_reduction_red = np.dot(result_data_red.T,red_pixel_preprocessing)
result_data_reduction_green = np.dot(result_data_green.T,green_pixel_preprocessing)
result_data_reduction_blue = np.dot(result_data_blue.T,blue_pixel_preprocessing)
print('test_values',result_data_reduction_red)
#result_data_reduction_rgb = np.array([result_data_reduction_red,
                                                 result_data_reduction_green,
                                                 result_data_reduction_blue])
final_r = np.dot(result_data_red,result_data_reduction_red)
final_g = np.dot(result_data_green,result_data_reduction_green)
final_b = np.dot(result_data_blue,result_data_reduction_blue)
final_r_1 = pca_test.restore_image(pca_test,average_1,final_r)
final_g_1 = pca_test.restore_image(pca_test,average_2,final_g)
final_b_1 = pca_test.restore_image(pca_test,average_3,final_b)
final_rgb = np.array([final_r_1.T,final_g_1.T,final_b_1.T])
print(final_rgb.T)
#print(result_rgb.T.shape)
#reduction_matrix =
pca_test.plot_line_chart(pca_test,eigenvalues_red,eigenvalues_green,eigenvalues_blue)
# pca_test.plot_line_chart(pca_test,eigenvalues_green)
# pca_test.plot_line_chart(pca_test,eigenvalues_blue)
old_im = pca_test.MatrixToImage(pca_test,pixel_matrix)
old_im.show()
#old_im.save('old.png')
new_im = pca_test.MatrixToImage(pca_test,final_rgb.T)
new_im.show()
new_im.save('new_k30.png')
#green_pixel = pixel_matrix[:,:,1]
# x_lab = list(range(length))
# plt.plot(x_lab,eigenvalues)
# plt.show()
```

最后挑选了特征值前1,5,10,20,30,50个作为重构投影矩阵并给出重构的图片(图5)。

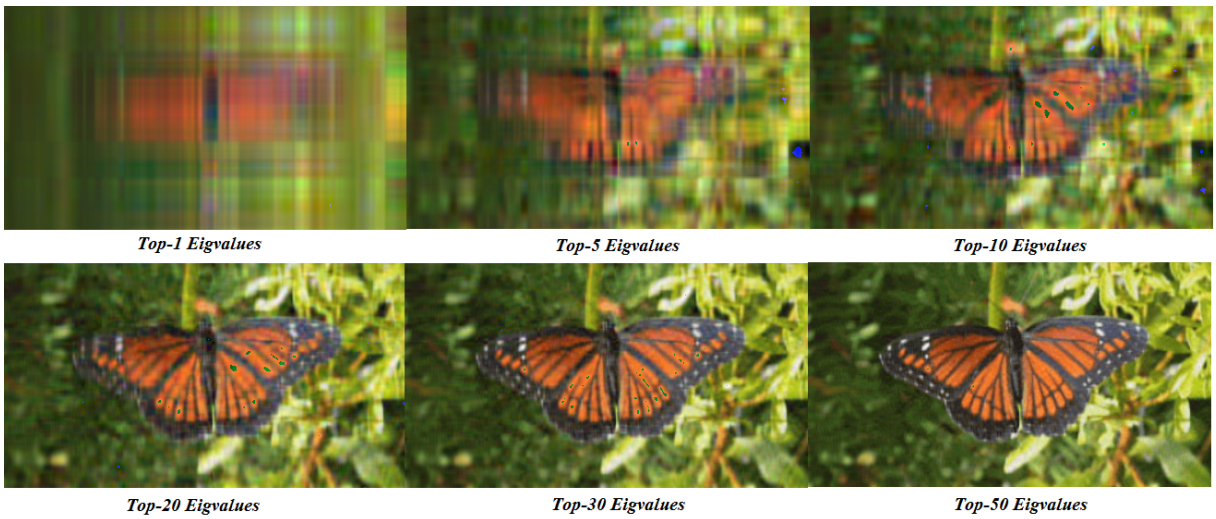

图 5: Sorted Eigenvalues Line Chart

四 总结体会

关于PCA的实验过程，其实整体来说与SVD相似，都是利用特征值下降的速度非常快的性质，通过进行特征值分解将大部分非常接近于0的特征值（对数据的影响非常小的特征值）进行剔除，从而实现数据的压缩。做PCA实验收获最大的一点在于在实际进行PCA操作时的一点技巧，一般求解需要先求协方差矩阵$\boldsymbol{AA^T}$，但是对于高维的数据来说，这样求矩阵的维度会很高，但是联系到SVD分解就会发现可以用SVD分解先求$\boldsymbol{A^TA}$求解右奇异矩阵和奇异值对角阵，对角阵的平方和PCA中的特征值对角阵是对应的，同时可以通过右奇异矩阵求解左奇异矩阵（对应PCA中的特征向量张成的矩阵），这样可以大大地减少计算量，其中有一个主要的原因就在于对大多数的矩阵操作来说计算复杂度十分高，导致运算过慢。这算是一个小小的trick可以用到以后的实验中。