Part1

利用PCA算法对绿萝点云数据进行降维处理

一、PCA算法简述

PCA是一种经典的数据分析方法，是一种数据降维的方式。由于早期它在人脸识别的贡献巨大，大家也经常把它称为一种特征提取的方法。PCA(Principal Component Analysis)，即主成分分析法，其目的保留人们所需要的主要信息，而舍弃冗余的信息。

该算法核心就是找到一个P矩阵，使原始数据集X通过，Y=PX映射到Y，得到一个新的数据集。那么如何找到这个P矩阵呢？

（1）对于给定n组数据 ,其中每个数据具有d维,对于本次实验中的绿萝（*Epipremnum aureum*）点云数据集来说，维度为6维。我们先读取需要处理的数据集，并且以矩阵方式存储。

（2）首先我们对每个维度的数据进行中心化处理：

 。

（3）然后计算数据集矩阵X的协方差Cx：

。

（4）对CX进行特征值分解，CX=PTDP；其中D为对角阵，对角线上的元素按递减顺序排列，对应的特征向量在P矩阵中按行排列。特征值的大小决定了相应数据信息的重要程度，特征值越大，表示信息越重要，即主成分。在对角阵D对角线上的元素表示特征值，这个时候可以根据所需要的主成分从P矩阵中挑选相应的行向量。

（5）得到了P矩阵，就可以从X映射到Y了，Y=PX。

二、利用PCA对绿萝点云数据降维

对于数据集中的6个特征，每一行代表点云中一个点所具有的特征，前3个数是空间坐标，后3个数是法向量坐标。实验要求分别保留第1、2主成分和第1、2、4主成分。

1.程序运行平台和配置：

（1）硬件：笔记本电脑，处理器为Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz，显卡为AMD Radeon R5 M200 / HD 8500M Series。

（2）软件：python3.6

注：使用python3.6时利用了pycharm的IDE环境

2.程序代码：

由于要对数组操作，所以导入numpy模块；

由于需要进行可视化操作，所以分别导入matplotlib库和Axes3D。

本实验代码从pycharm里以原格式保存的。具体代码如下：

*'''  
@date:2018.11.21  
@author:wangzhiyuan  
'''*import numpy as np  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
#step1:读取数据  
data=np.loadtxt('PointsNormals绿萝.txt') #读取数据  
data1=data.T #将读取的数据矩阵转置  
n=np.shape(data1)[1] #读取矩阵的列数，即数据的组数  
d=np.shape(data1)[0] #读取矩阵的行数，即数据的维度  
#step2:计算数据向量均值，进行中心化操作  
data\_mean=np.mean(data1,1) #计算矩阵每一行元素均值,1xd大小  
for i in range(n):  
 data1[:,i]=data1[:,i]-data\_mean.T #中心化处理  
#step3:计算协方差矩阵Cx  
Cx=np.cov(data1)  
#step4:对Cx进行特征值分解  
e,v = np.linalg.eig(Cx) #e为特征值，V为相应的特征向量 注意这里列向量为特征向量  
p=v.T  
#step5:降维处理  
#降到二维：  
p\_1=np.delete(p,[2,3,4,5],axis=0) #保留P矩阵中前两行向量  
out\_1=np.dot(p\_1,data1) #得出降维后的矩阵  
#降到三维，且选择第一、二、四主元  
p\_2=np.delete(p,[2,4,5],axis=0) #保留p矩阵中第一、二、四行向量  
out\_2=np.dot(p\_2,data1) # 得出降维后的矩阵  
#step6:可视化  
x1=out\_1[0,:]  
y1=out\_1[1,:] #分别读取二维图的横纵坐标  
fig1 = plt.figure()  
ax1 = fig1.add\_subplot(111)  
ax1.scatter(x1,y1,marker='.',alpha=0.2) #二维图的散点图  
ax1.set\_xlabel('x')  
ax1.set\_ylabel('y')  
x2=out\_2[0,:]  
y2=out\_2[1,:]  
z2=out\_2[2,:] #分别读取三维图的三个坐标  
fig2 = plt.figure()  
ax2 = fig2.add\_subplot(111)  
ax2 = plt.axes(projection='3d')  
ax2.scatter(x2,y2,z2,marker='.',alpha=0.2) #三维图的散点图  
ax2.set\_xlabel('x')  
ax2.set\_ylabel('y')  
ax2.set\_zlabel('z')  
plt.show()

三、实验结果

由于数据集中的数据过多，所以我对散点进行了透明处理，以便观察。其结果图如下：

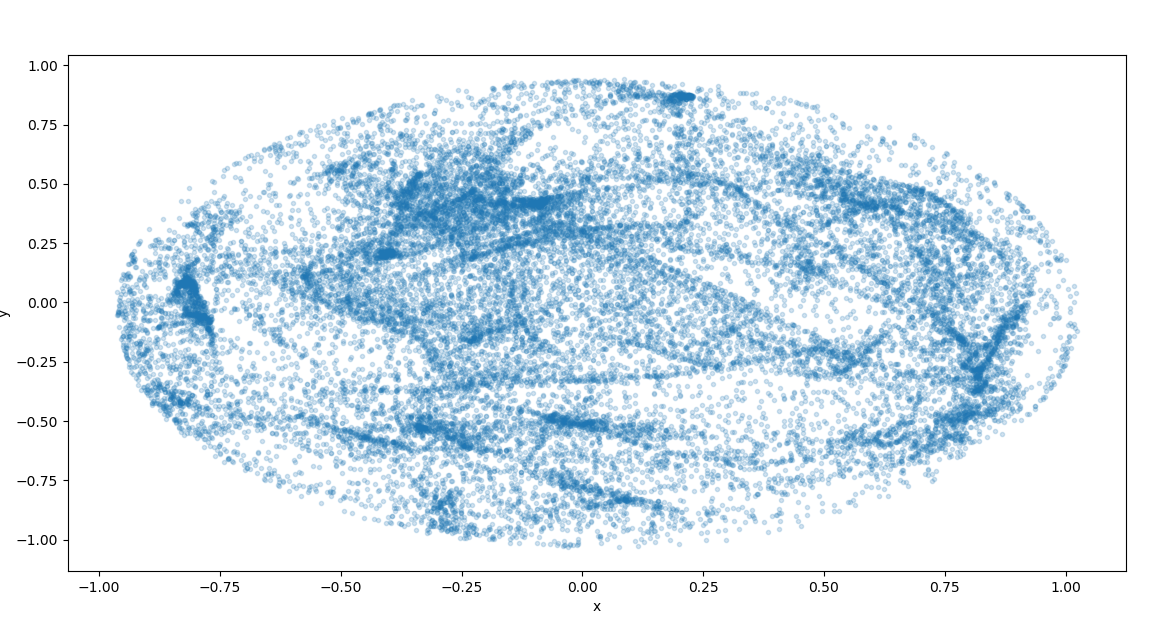


图1.二维点云分布图

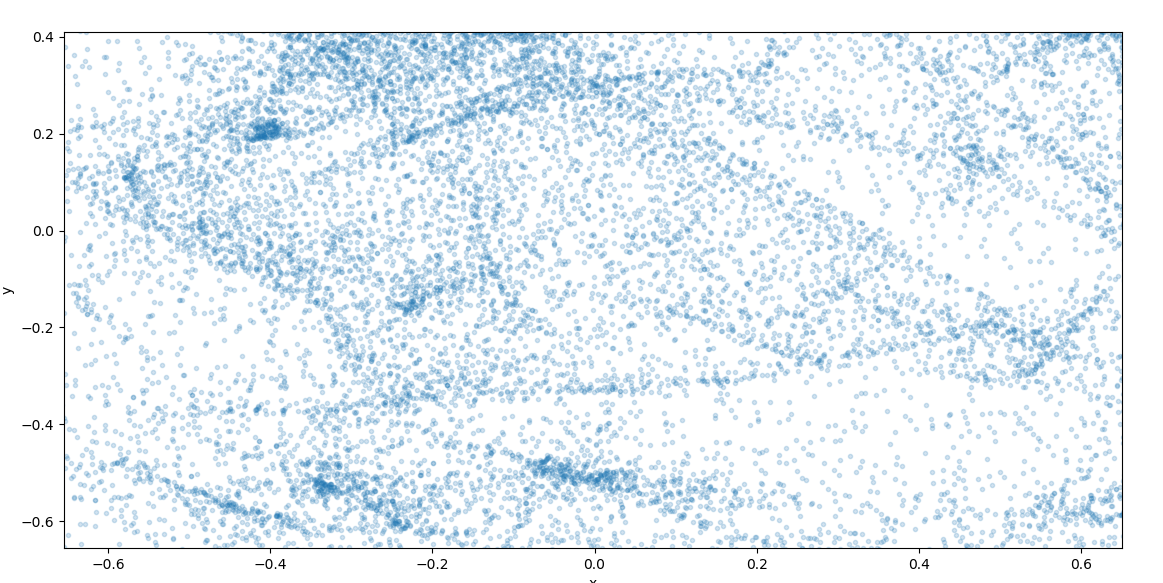


图2.二维点云分布图（放大）

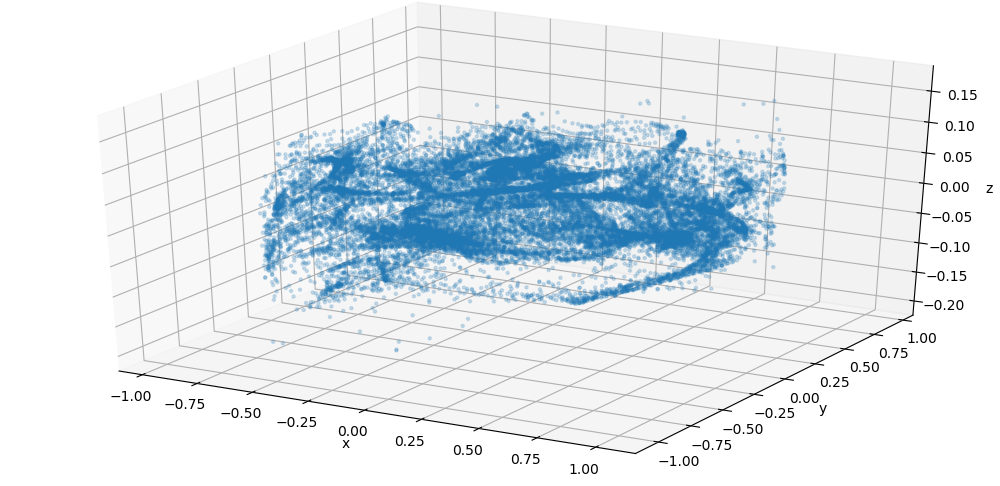


图3.三维点云分布图（原图）

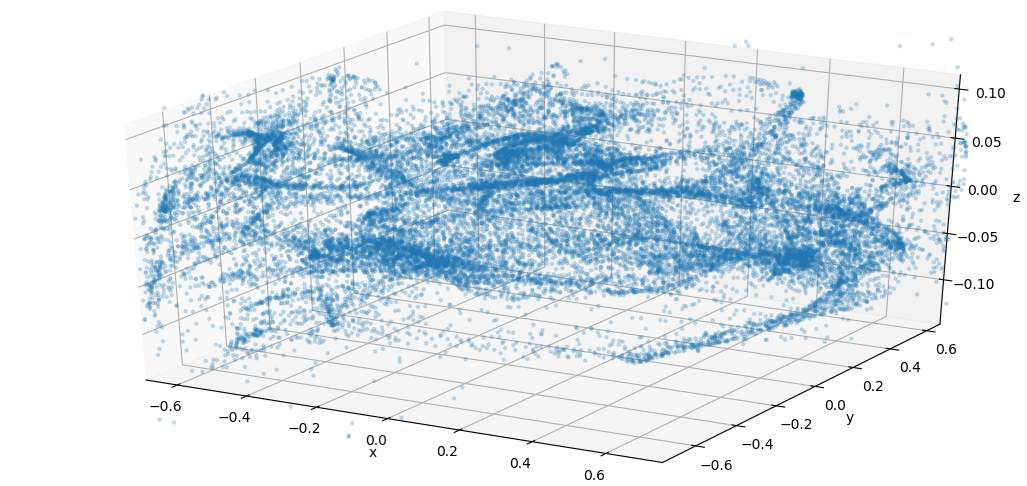


图4.三维点云分布图（放大）

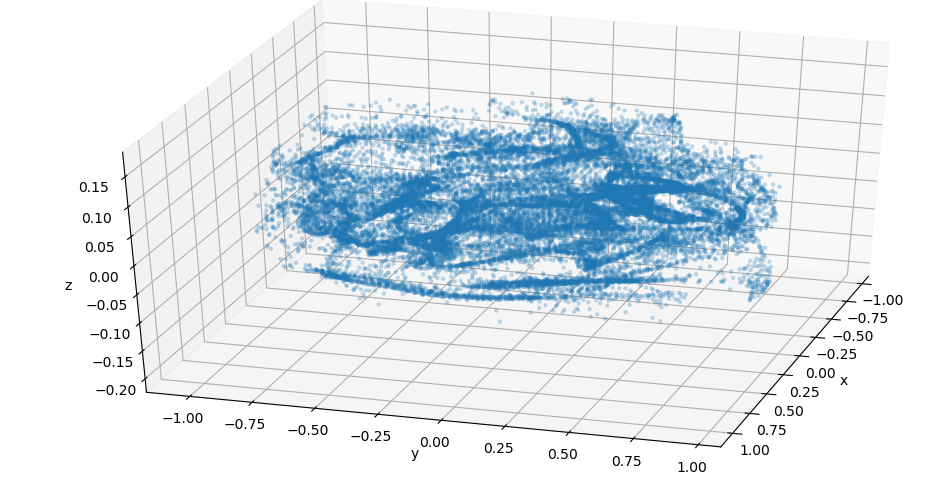


图5.三维点云分布图（旋转90度）

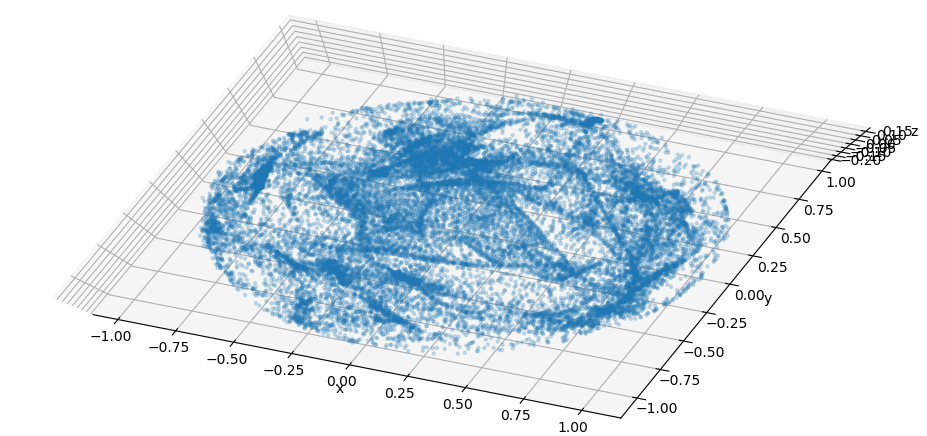


图6.三维点云分布图（俯视图）

四、结果分析和实验中遇到的问题处理

1.结果分析

在进行PCA降维过程中，数据集中的6维数据被依次降成2维和3维。根据二维点分布图，可知二维的数据在平面上表征为一个椭圆。但是椭圆的各部分颜色深度不同，越深的地方表示数据点越密集，有一定的聚类效果；颜色越浅，数据越稀疏。对于三维点分布图，看上去是一个不规则的椭球体，上下层的数据较中部显得稀疏。从俯视图来看，其结果就是二维点分布图。

2.遇到的问题

（1）一些基础的python操作。比如怎样正确使用数组，如何利用python可视化处理。

（2）数据的处理。由于PCA算法的数据矩阵的存储方式不同，所以求P矩阵时要注意是否要转置，这里我在运行代码时都会注意查看各矩阵的行和列，以确保不出错。

（3）数据量过大。绿萝点云数据集还有两万多个数据点，因此画散点图时，必须设置散点透明度，否则将难以分辨，本次实验中设置的不透明度为0.2。在旋转散点图时，也因为数据量过大，速度变慢，而且会卡顿。当然这也跟硬件配置相关。

Part2

利用迭代式PCA对片状三维数据拟合平面

一、IPCA算法简述

IPCA是一种迭代式PCA算法，可以利用这种算法在点云数据中拟合平面。其核心思路是对于三维片状数据点区域，特征值最小的主成分方向与该片状区域的法向量方向类似。根据点法式平面方程式，只要知道一个点和法向量，就能确定一个平面。

具体实现步骤如下：

（1）从三维数据点集X中挑选一个较为中心的点为初始迭代点。

（2）在数据集X中做PCA主成分分析找出最小特征值对应的P3向量为法向量。

（3）根据点法式平面方程拟合平面F。

（4）删除离群点，对于X中的点经过遍历，确定它们到平面的距离是否超过给定阈值，若超过，则删除该点，更新数据集X。

（5）回到步骤2，进行迭代直到平面稳定。

二、具体实现

1.程序运行平台和配置：

（1）硬件：笔记本电脑，处理器为Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz，显卡为AMD Radeon R5 M200 / HD 8500M Series。

（2）软件：python3.6

2.程序代码：

*'''  
@date:2018.11.22  
@author:wangzhiyuan  
'''*import numpy as np  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
x = np.random.random(100)\*10-5  
y = np.random.random(100)\*10-5 #100个随机点，x,y坐标在（-5,5）之间  
z =2\*x+3\*y +np.random.random(100)\*3-4 #100个随机点，在确定平面上z坐标略微变动  
x1=np.random.random(20)\*10-5  
y1=np.random.random(20)\*10-5  
z1=np.random.random(20)\*40-20 #另外随机生成20个干扰点  
  
temp1=np.vstack((x,y,z)) #合并矩阵  
temp2=np.vstack((x1,y1,z1))  
data1=np.hstack((temp1,temp2))  
#PCA分析求法向量  
def pca(datai):  
 *'''  
 input:datai #输入数据集  
 output:p3 #输出法向量  
 '''* n=np.shape(datai)[1] #读取矩阵的列数，即数据的组数  
 d=np.shape(datai)[0] #读取矩阵的行数，即数据的维度  
 data\_mean=np.mean(datai,1) #计算矩阵每一行元素均值,1xd大小  
 for i in range(n):  
 datai[:,i]=datai[:,i]-data\_mean.T #中心化处理  
 Cx=np.cov(datai)  
 e,v = np.linalg.eig(Cx) #e为特征值，V为相应的特征向量 注意这里列向量为特征向量  
 p=v.T  
 p3=p[2,:] #挑选第三行元素为法向量  
 return p3  
#删除离群点  
def dec(limit,dataset,xs,max):  
 *'''* ***:param*** *max: 迭代次数* ***:param*** *limit: 阈值* ***:param*** *dataset: 数据集* ***:param*** *xs: 中心点* ***:return****: dataset:删除离群点后的数据集  
 np.mat(ex)：离群点的数据  
 np.mat(ea):法向量  
 '''* ps=pca(dataset)  
 ea=[]  
 ex = []  
 ey=[]  
 ea.append(ps)  
 for i in range(max):

c=(np.sqrt(np.square(ps[0])

/+np.square(ps[1])+np.square(ps[2])))\*limit  
 D=0-ps[0]\*xs[0]-ps[1]\*xs[1]-ps[2]\*xs[2] #点到平面距离与阈值大小判断的相关计算  
 m=np.shape(dataset)[1]#数据个数  
 for t in range(m):  
 if(abs(np.dot(ps,dataset[:,t])+D)>c):  
 ex.append(dataset[:,t]) #把离群点存储在列表里  
 ey.append(t)  
 dataset = np.delete(dataset, ey, axis=1) # 删除离群点  
 limit=limit-2 #阈值每次迭代后减1  
 ps=pca(dataset)  
 ea.append(ps) #把法向量存储在列表里  
 #print(np.shape(np.mat(ea))[0])  
 return dataset,np.mat(ex),np.mat(ea)  
#可视化操作  
def plot1(ps,xs,dataset,ex):  
 *'''* ***:param*** *ps: 法向量* ***:param*** *xs: 中心点* ***:param*** *dataset: 数据集* ***:param*** *ex: 离群点* ***:return****:  
 '''* x,y=np.mgrid[-5:5:20j,-5:5:20j]  
 z0=xs[2]-(ps[0,0]\*(x-xs[0])+ps[0,1]\*(y-xs[1]))/ps[0,2]  
 z1=xs[2]-(ps[1,0]\*(x-xs[0])+ps[1,1]\*(y-xs[1]))/ps[1,2]  
 z2=xs[2]-(ps[2,0]\*(x-xs[0])+ps[2,1]\*(y-xs[1]))/ps[2,2]  
 #z3=xs[2]-(ps[3,0]\*(x-xs[0])+ps[3,1]\*(y-xs[1]))/ps[3,2] #平面约束  
 fig = plt.figure()  
 ax1 = fig.add\_subplot(111)  
 ax1 = plt.axes(projection='3d')  
 ax1.scatter(dataset[0,0],dataset[1,0],dataset[2,0],color='red')#初始点标记为红色  
 ax1.plot\_surface(x, y, z0, rstride=100, cstride=100, color='gold', alpha='0.5') # alpha为不透明度  
 ax1.plot\_surface(x, y, z1, rstride=100, cstride=100, color='blue', alpha='0.5') # alpha为不透明度  
 ax1.plot\_surface(x, y, z2, rstride=100, cstride=100, color='green', alpha='0.5') # alpha为不透明度  
 #ax1.plot\_surface(x, y, z3, rstride=100, cstride=100, color='purple', alpha='0.5') # alpha为不透明度  
 m=np.shape(ex)[0]  
 for i in range(m):  
 ax1.scatter(ex[i,0],ex[i,1],ex[i,2],color='black') #离群点标记为黑色  
 dataset=np.delete(dataset,0,axis=1) #初始点已经显示了，所以在显示平面上的点时把它删除  
 x1=dataset[0,:]  
 y1=dataset[1,:]  
 z1=dataset[2,:]  
 ax1.scatter(x1, y1, z1,) #正常的点集  
 ax1.set\_xlabel('x')  
 ax1.set\_ylabel('y')  
 ax1.set\_zlabel('z')  
 plt.show()  
#结果  
dataset\_1,ex1,ea1=dec(5,data1,data1[:,0],2)  
plot1(ea1,data1[:,0],dataset\_1,ex1)

三、实验结果

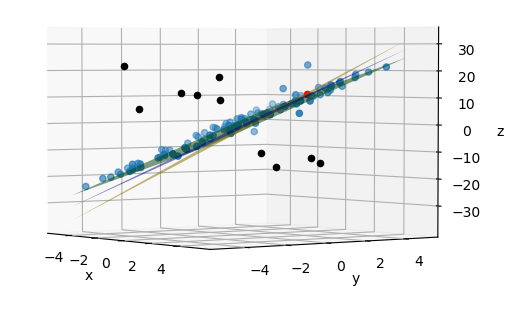


图7.拟合平面结果

注：其中金色平面表示第一次拟合平面，蓝色平面表示第二次拟合平面，绿色表示第三次拟合平面。红色点是初始迭代点，黑色点是离群点，蓝色点是保留点。

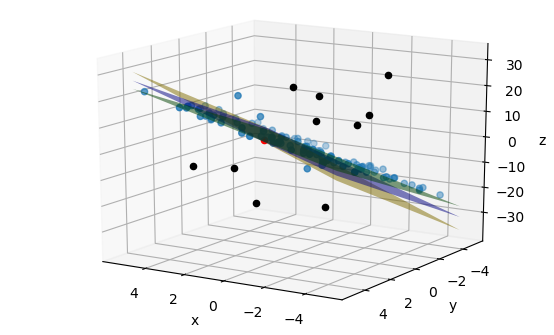


图8.拟合平面结果（另一角度）

四、结果分析和实验中遇到的问题处理

1.结果分析

对结果分析前，先看看数据集。数据集的产生过程如下：

（1）先确定一个平面；

（2）对于平面上的100个点在Z方向上进行随机较小的抖动；

（3）再添加20个随机的散点。

因此，添加的20个散点中必定存在离群点，判断离群点的阈值影响迭代次数。阈值越大，迭代次数越多。阈值越小，迭代次数越少，但是丢失的数据点也会变多。对于阈值的选取，我采取了简单的线性映射方法，每迭代一次，阈值减小2。本次实验初始阈值为5，图7和图8的结果，迭代3次就收敛了。

2. 实验中遇到的问题处理

（1）数据集的生成。由于数据集是每一次运行程序产生的，而不是固定的，因此每次运行程序最后拟合的结果不同。这样也不利于阈值的选取。图9是另外一个数据集的平面拟合结果。

（2）迭代停止条件。这里有两个思路，第一个设置阈值的下限，当阈值减小到一定程度停止；第二个设置迭代次数。本次实验选择了第二种思路，因为在进行可视化操作要对平面进行着色，所以要预先假设一下迭代次数。而且生成的数据集比较简单，迭代次数就不会太多。

（3）阈值的选取。首先阈值肯定不是一个固定值，但是它的变化规律应当根据数据集来定。本次实验采用了每次迭代阈值减小两个单位的方式，来使它自适应化。

各种主观因素对实验都有影响。

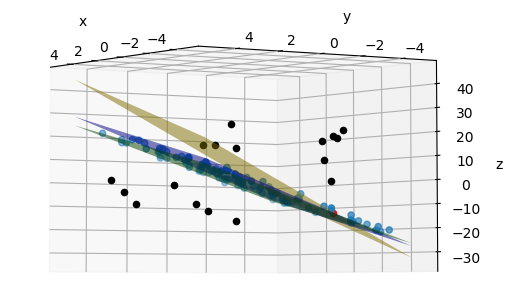


图9.拟合平面结果（其他数据集）

五、思考题

RANSAC算法也能够进行点云数据平面拟合。

1.RANSAC算法

RANSAC(random sample consensus)是一种求解已知模型参数的框架。将这个模型应用于平面拟合的过程如下：

（1）从数据集X中随机选取模型参数估计的最少的数据，选择不在一条直线的3个点，然后可以计算出一个平面。

（2）计算数据集X中其他数据点到（1）中平面的距离d,当d>t1，则判断该点为离群点，d<t1,则判断该点为内点。统计内点个数n，当n>t2时判定该平面为最佳平面m\_best，记录此时每个内点与平面距离误差error\_min。

（3）重复以上（1）（2）过程，计算得到新的平面，比较其统计误差error和error\_min的大小,从而更新m\_best和error\_min。迭代数据集中的每个数据点，最终得到最佳平面。

已知3个点如何确定一个平面呢？如果知道法向量就容易多了，已知p1,p2,p3三个点求平面法向量的过程如下：



其中n就是得到的法向量。然后根据点法式就可以确定平面方程了。

2.算法实现

代码如下：

*'''  
@date:2018.11.23  
@author:wangzhiyuan  
'''*import numpy as np  
import matplotlib.pyplot as plt  
from mpl\_toolkits.mplot3d import Axes3D  
#生成数据集  
x = np.random.random(100)\*10-5  
y = np.random.random(100)\*10-5 #100个随机点，x,y坐标在（-5,5）之间  
z =2\*x+3\*y +np.random.random(100)\*3-4 #100个随机点，在确定平面上z坐标略微变动  
x1=np.random.random(20)\*10-5  
y1=np.random.random(20)\*10-5  
z1=np.random.random(20)\*40-20 #另外随机生成20个干扰点  
  
temp1=np.vstack((x,y,z)) #合并矩阵  
temp2=np.vstack((x1,y1,z1))  
data=np.hstack((temp1,temp2))  
#根据3个点数据求平面方程  
def plane(data1,data2,data3):  
 *'''* ***:param*** *data1: 第1个数据点* ***:param*** *data2: 第2个数据点* ***:param*** *data3: 第3个数据点* ***:return****: c: 平面法向量  
 '''* a = data1 - data2  
 b = data1 - data3  
 c = np.array([0, 0, 0])  
 c[0] = a[1] \* b[2] - a[2] \* b[1]  
 c[1] = a[2] \* b[0] - a[0] \* b[2]  
 c[2] = a[0] \* b[1] - a[1] \* b[0]#根据3个点确定法向量  
 return c  
#判断内外点，并计算点到平面的误差  
def dec(limit1,limit2,dataset,xs,ps):  
 *'''* ***:param*** *limit1: 判断距离的阈值t1* ***:param*** *limit2: 判断内点个数是否达标的阈值t2* ***:param*** *dataset: 数据集* ***:param*** *xs: 一个数据点* ***:param*** *ps: 法向量* ***:return****: eb: 用来拟合平面的一个点  
 ea: 法向量  
 '''* ea=[]  
 ex=[]  
 ey=[]  
 eb=[]  
 es=[]  
 c=(np.sqrt(np.square(ps[0])+np.square(ps[1])+np.square(ps[2])))  
 D=0-ps[0]\*xs[0]-ps[1]\*xs[1]-ps[2]\*xs[2] #点到平面距离与阈值大小判断的相关计算  
 m=np.shape(dataset)[1]#数据个数  
 for t in range(m):  
 d=abs(np.dot(ps,dataset[:,t])+D)  
 if d<(limit1\*c):  
 ey.append(dataset[:,t])#存储内点  
 ex.append(d\*c) #把每个点到平面的距离存储在列表里  
 if len(ey)>=limit2:  
 es.append(sum(ex)) #每个点到平面的距离之和，即误差  
 eb.append(xs)  
 ea.append(ps) #把法向量存储在列表里  
 return eb,ea,es  
m=np.shape(data)[1]  
sum\_list=[]  
p\_list=[]  
x\_list=[]  
#反复迭代，遍历所有的点，求出所有符合条件的平面  
for i in range(0,m,3):  
 p=plane(data[:,i],data[:,i+1],data[:,i+2])  
 point,p1,s1=dec(2,105,data,data[:,i],p)  
 sum\_list.append(s1)  
 p\_list.append(p1)  
 x\_list.append(point)  
#删除列表中的空列表  
while [] in sum\_list:  
 sum\_list.remove([])  
while [] in p\_list:  
 p\_list.remove([])  
while [] in x\_list:  
 x\_list.remove([])  
minNum = min(sum\_list)  
n=sum\_list.index(minNum)#找到误差最小的平面  
print(minNum,n)  
#可视化操作  
x1,y1=np.mgrid[-5:5:20j,-5:5:20j]  
xs=np.mat(x\_list[n])  
p=np.mat(p\_list[n])  
z1=xs[0,2]-(p[0,0]\*(x1-xs[0,0])+p[0,1]\*(y1-xs[0,1]))/p[0,2]  
fig = plt.figure()  
ax1 = fig.add\_subplot(111)  
ax1 = plt.axes(projection='3d')  
ax1.plot\_surface(x1, y1, z1, rstride=100, cstride=100, color='blue', alpha='0.5') # alpha为不透明度  
ax1.set\_xlabel('x')  
ax1.set\_ylabel('y')  
ax1.set\_zlabel('z')  
ax1.scatter(data[0,:], data[1,:], data[2,:])  
plt.show()

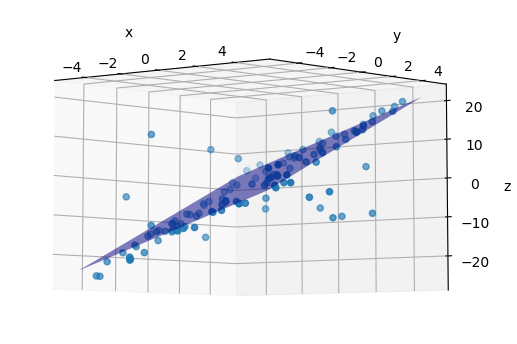


图10.RANSAC算法拟合平面结果

3.结果分析

该算法的两个参数t1和t2的选择比较重要。由于拟合过程中的数据集就是自己有规律生成的，所以凭经验确定阈值t1和t2。本来有100个点在一个平面上随机波动（波动范围小），增加的20个干扰点随机运动。考虑到一些干扰点会跟平面接近的情况，我把判断是否为一个好平面的内点阈值设置为105。同样地，根据数据的特性，把判断内点的阈值设置为2。实验证明该算法能够在有干扰的点云数据中很好地拟合出平面。