层次聚类

基本处理

聚类就是对大量未知标注的数据集，按数据的内在相似性将数据集划分为多个类别，使类别内的数据相似度较大而类别间的数据相似度较小。

在这一实现过程中，可以利用多种方法进行相似度度量，譬如Pearson系数，余弦相似度，而针对某一些点的聚类问题，则可以通过对距离的测量方式进行实现。

而其具体的实现方法，则可以使用聚合聚类与分裂聚类进行处理：

聚合聚类 (bottom-up)

采用自底向上的策略，开始将每个样本各自分到一个类；之后将相距最近的两类合并，建立一个新的类，重复操作直到满足停止条件；得到层次化的类别

分裂聚类 (top-down):

采用自顶向下的策略，开始将所有样本分到一个类；之后将已有类中相距最远的样本分到两个新的类，重复操作直到满足停止条件；得到层次化的类别

聚类实现

在本次实验中，分别针对single linkage，complete linkage与average linkage进行了实现，其基本流程如下：

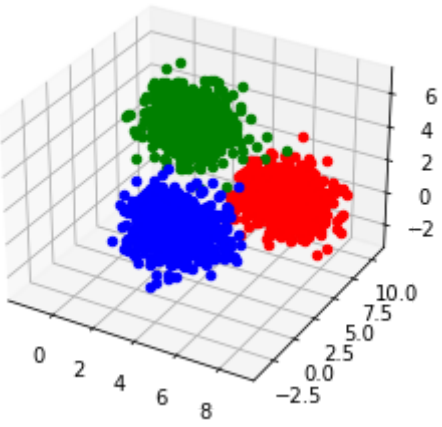
数据生成

```
sample_size=2000
cluster_cat_num=3
def make_dataset(class_num,sample_num=sample_size,dimsions=3):
    matrix=np.zeros((sample_num,dimsions))
    label=[]
    x_mean=np.random.randint(10,size=class_num)
    y_mean=np.random.randint(10,size=class_num)
    z_mean=np.random.randint(10,size=class_num)
    print(x_mean)
    for i in np.arange(sample_num):
        class_type=np.random.randint(class_num)
        #print(class_type)

 gene=np.random.multivariate_normal((x_mean[class_type],y_mean[class_type],z_mea
n[class_type]),cov=np.identity(dimsions),)
        for ide in np.arange(dimsions):
            matrix[i][ide]=gene[ide]
        #atrix[i][dimsions]=class_type
        label.append(class_type)
    return matrix,label
data,label=make_dataset(3)
```

生成了随机3类的，在0~10范围内的三维数据，每个数据集大小为2000，这一过程中生成的数据间协方差为0，因此其中x,y,z三者不具有相关性。

由此可以绘制出三个数据集的基本图像：



距离矩阵计算与维护

为了计算这一过程中的各个集合的间隔，需要生成一个距离矩阵，其初始化为各个点间隔距离，而在后续的合并过程中，需要根据在合并过程中类的合并进行调整，基于此，进行合并项目的筛选，这一过程中，使用欧氏距离作为计算标准：

```
def cal_matrix(matrix):
    distance_mat=np.zeros((len(matrix),len(matrix)))
    for i in np.arange(len(matrix)):
        for j in range(i+1,len(matrix)):
            distance_mat[i][j]=distance_cal(matrix[i],matrix[j],)
            distance_mat[j][i]=distance_cal(matrix[i],matrix[j],)
    for i in np.arange(len(matrix)):
        distance_mat[i][i]=np.inf
    return distance_mat
def distance_cal(a,b,dimsion=3):
    sum_col=0
    for i in np.arange(dimsion):
        sum_col+=(a[i]-b[i])**2
    return sum_col
distance=cal_matrix(data)
```

cluster类

在这一过程中，需要维护一个cluster类，针对其进行不同聚类的合并等操作：

```
class cluster_type:
    def __init__(self,list_num,type_c):
        self.list_num=list_num
        self.leng=len(list_num)
        self.type_cluster=type_c
        self.label_appended=[]
    #def append_item(num_list):
    #    self.list.append_item
    def merge(self,another_cluster):
        for item in another_cluster.list_num:
            self.list_num.append(item)
        self.label_appended.append(another_cluster.type_cluster)
        self.leng=self.leng+another_cluster.leng
    def append(self,num):
        self.list_num.append(num)
```

```
        self.leng+=1
```

聚类操作

```
def up_date_distance(dis_map,cluster_list):
    distance_map=dis_map.copy()
    for iter_item in cluster_list: #需要被删除的项
        mat=[]
        #print(cluster_list[0].list)
        for item in iter_item.list_num:
            mat.append(distance_map[item])
        arg_list=np.min(mat,axis=0)
        #print(arg_list)
        for item in iter_item.list_num:
            distance_map[item]=arg_list
    return distance_map
def
cluster_data(class_to,distance_matrix,label_array_before,init_class,type_c='sing
le'):
    if type_c=='single' or type_c=='average':
        set_num=100000
    elif type_c=='complete':
        set_num=-100000
    else:
        return None
    distance=copy.deepcopy(distance_matrix)
    label_array=copy.deepcopy(label_array_before)
    cluster_list=[]
    class_num=class_to
    record=0
    while(1):
        row,col=indmin_matrix(distance)
        #distance[row][col]=np.inf
        #distance[col][row]=np.inf
       # print(row,col)
        if label_array[row]==np.inf and label_array[col]==np.inf:
            #print("case 1")
            cluster=cluster_type([row,col],type_c=record)
            cluster_list.append(cluster)
            label_array[row],label_array[col]=record,record
        elif label_array[row]!=np.inf and label_array[col]!=np.inf:
            #print("case 2")
            col1=get_index_of_cluster(row,cluster_list)
            col2=get_index_of_cluster(col,cluster_list)
            cluster_list[col1].merge(cluster_list[col2])
            del cluster_list[col2]
            label_array[row]=label_array[col]
        elif label_array[row]==np.inf and label_array[col]!=np.inf:#row需要并入
            #print("case 3")
            col2=get_index_of_cluster(col,cluster_list)
            #print(col2)
            cluster_list[col2].append(row)
            label_array[row]=label_array[col]
        elif label_array[col]==np.inf and label_array[row]!=np.inf:#col并入
            col2=get_index_of_cluster(row,cluster_list)
            cluster_list[col2].append(col)
```

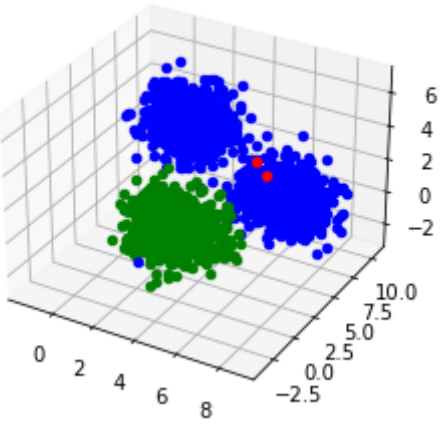
```
            label_array[col]=label_array[row]
        record+=1
        #print(cluster_list)
        for iter_item in cluster_list: #需要被删除的项
            mat=[]
        #print(cluster_list[0].list)
            for item in iter_item.list_num:
                mat.append(distance[item])
            for i in np.arange(len(iter_item.list_num)):
                for j in  range(i,len(iter_item.list_num)):
                    distance[iter_item.list_num[i]]
[iter_item.list_num[j]]=set_num
                    distance[iter_item.list_num[j]]
[iter_item.list_num[i]]=set_num
            if type_c=='single' :
                arg_list=np.min(mat,axis=0)
            elif type_c=='complete':
                arg_list=np.max(mat,axis=0)
            else :
                arg_list=np.mean(mat,axis=0)
            #arg_list=np.min(mat,axis=0)
            #print(arg_list)
            for item in iter_item.list_num:
                distance[item]=arg_list
        #distance=up_date_distance(distance,cluster_list)
        init_class=init_class-1
        #print(init_class)
        if(init_class<20 and len(cluster_list)==class_to-1 ):
            break
    return cluster_list
def single_HC(data_show,list_output,method):   #画图
    labels_after=np.zeros(sample_size)
    type_clu_of_sample=0
    for item in list_output:
        for i in item.list_num:
            labels_after[i]=type_clu_of_sample
        type_clu_of_sample+=1
    draw_distribution(data_show,labels_after,method=method)
single_HC(data,list_output_1,"single")
single_HC(data,list_output_2,"average")
```

这一步骤的基本流程有两部分：

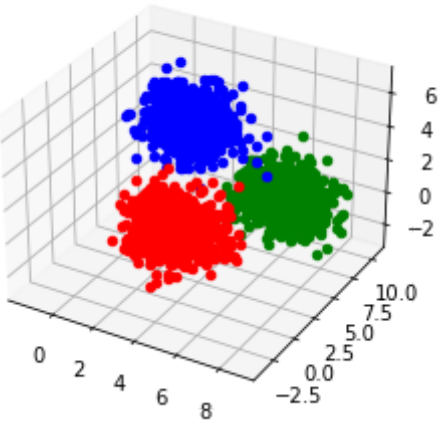
- 筛选出符合相应条件的聚类进行合并
- 对合并后的聚类间隔进行更新，这一过程通过对距离矩阵的操作进行实现，即`up_date_distance`函数：

基于此，可以绘制出不同聚类下的结果图：

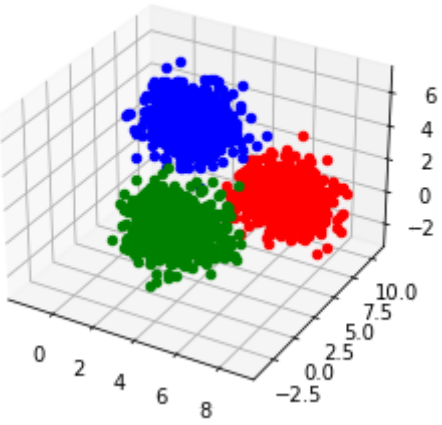
single



average



complete



针对结果判定，在每个分类结果中，以其中含有最多的类别作为该簇的标签，通过与原有标签进行比较，即可对不同参数下分类效果做出比较。

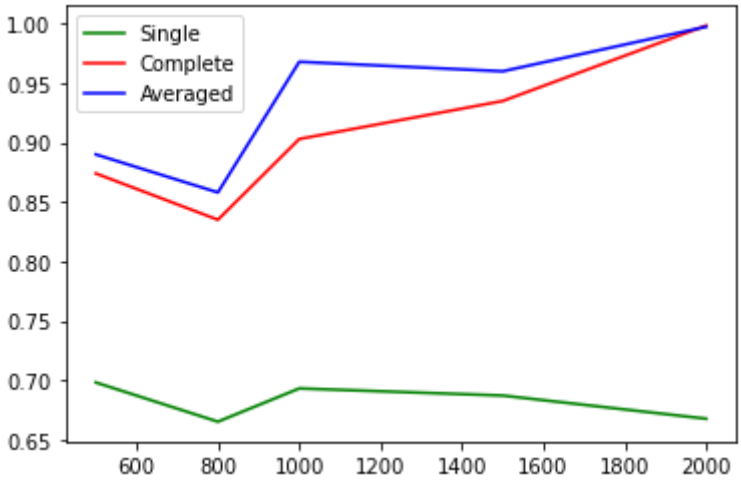
```
def get_acc(list_cat,cat_num=3):
    acc=0
    for item in list_cat:
        cat_stat=np.zeros(cat_num)
        for num in  sorted(item.list_num):
            cat_stat[label[num]]+=1
        print(cat_stat)
        acc+=np.max(cat_stat)
    print(acc/sample_size)
    return acc/sample_size
```

对比

在对不同数据集大小情况下，不同方法进行比较时，可以得到以下实验结果：

数据集大小	Single-linkage准确率	Complete-linkage准确率	Average-linkage准确率
2000	0.6675	0.9985	0.9975
1500	0.687	0.935	0.96
1000	0.693	0.903	0.968
800	0.665	0.835	0.858
500	0.698	0.874	0.89

可做出以下图：



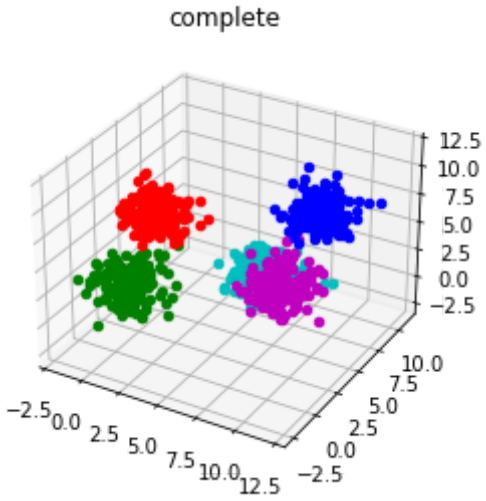
由此可以看出，Complete-linkage方法与Average-linkage方法明显好于Single-linkage方法，**实际上，在运行过程中，Single方法极易出现示例中的情况，即两类被合并而留下一类仅有个别点，这也是造成其准确率聚集在2/3的原因。**

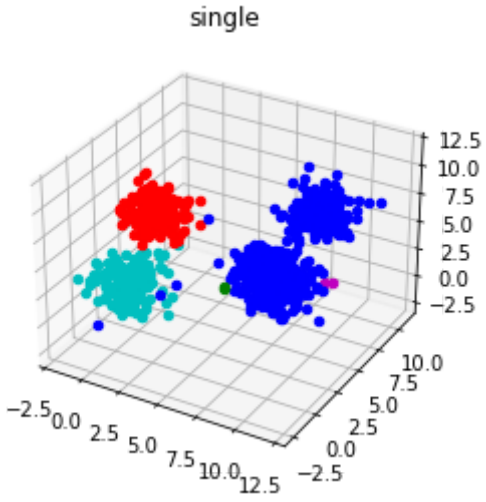
而在实际运行中，随着生成数据的变化，各个曲线表现的稳定性不足(以上数据采用3次求取平均的方法得到)，这也是导致其在不同大小下变化较大的原因，事实证明，层次聚类法的效果依赖于数据生成，鲁棒性不强。

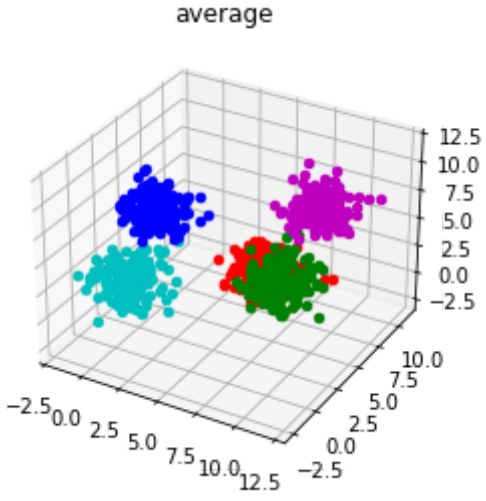
实际上，可以猜测，在真正的聚类过程中，在较后期的步骤中，由于随机性，往往比较大的聚类间的最小距离(single-linkage方法)会较小，而平均(average-linkage方法)距离与最大距离(complete-linkage方法)将较大，这使得在single-linkage方法下，较大簇类将更有可能进行相互合并，而在其他方法中，更倾向于较小的聚类融合进入较大聚类，这也造成了在single方法中，往往较容易出现类别分布极端不均匀而使得准确率较低的情况。

簇数目对比：

在本次实验中，分别对【3，5，8，10】的簇数目，以1000为大小的数据集进行计算，其结果如下：

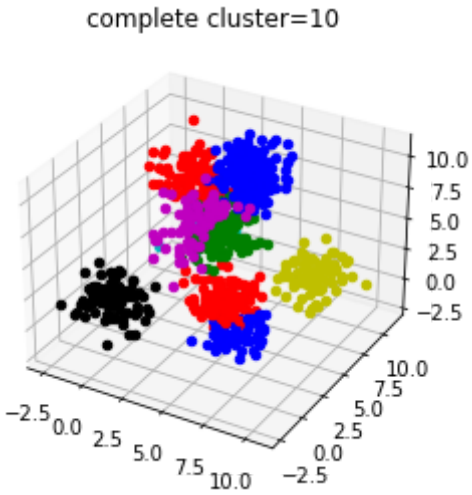


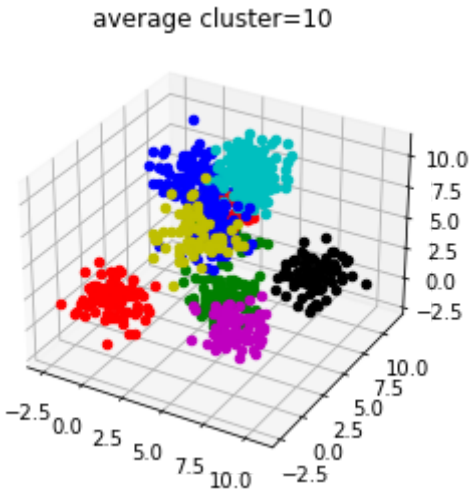




簇数目	Single-linkage准确率	Complete-linkage准确率	Average-linkage准确率
3	0.6675	0.9985	0.9975
5	0.61125	0.89125	0.8925
8	0.4675	0.83875	0.84375
10	0.41375	0.7975	0.79375

可以发现，随着聚类簇的数目增大，其准确率·将会下降，这一点也是符合常理的，在簇类数目较大时，聚类中样本更少而难以提供足够的信息以供归类：





基于此，可以发现，在多种情况（不同聚类簇，不同数据集大小）下，**Complete方法与Average方法效果近乎相当，均明显好于Single方法，说明选取最短距离作为聚类分组依据是存在一定问题的。**

总结

在本次实验中，实现了基本要求中对single-linkage，complete-linkage以及中级要求中average-linkage 层次聚类算法，与此同时，完成了提高与拓展要求，对三种算法在不同样本大小以及不同聚类簇数目的对比，通过可视化的方法展示出分类的过程，完成了从底部搭建出一个聚类划分树，通过不断合并，减小最终分类簇数目，完成了该实验。