In [ ]:

```
from scipy import stats as ss

ss.pearsonr(df[0], df[1])
```

9.3 实战练习

尝试对ridge表单中的三个自变量提取主成分，使用主成分回归办法解决这三个自变量的共线性问题，并将结果和前述岭回归的结果相比较。

尝试对boston数据进行因子分析，各变量的具体含义参见boston.DESCR。

10 聚类分析

10.1 聚类模型概述

10.2 K-均值聚类

class sklearn.cluster.KMeans(

```
    n_clusters = 8
    init = 'k-means++' : 'k-means++'/'random'/ndarray, 初始类中心位置
        'k-means++' : 采用优化后的算法确定类中心
        'random' : 随机选取k个案例作为初始类中心
        ndarray : (n_clusters, n_features)格式提供的初始类中心位置

    n_init = 10, max_iter = 300, tol = 0.0001
    precompute_distances = 'auto' : {'auto', True, False}
        是否预先计算距离，分析速度更快，但需要更多内存
        'auto' : 如果n_samples*n_clusters > 12 million，则不事先计算距离
    verbose = 0, random_state = None, copy_x = True, n_jobs = 1
    algorithm = 'auto' : 'auto', 'full' or 'elkan', 具体使用的算法
        'full' : 经典的EM风格算法
        'elkan' : 使用三角不等式，速度更快，但不支持稀疏数据
        'auto' : 基于数据类型自动选择
```

)

KMeans类的属性：

```
    cluster_centers_ : array, [n_clusters, n_features]
    labels_ :
    inertia_ : float，各样本和其最近的类中心距离之和
```

注意：对于样本量超过1万的情形，建议使用MiniBatchKMeans，算法改进为在线增量，速度会更快

In [ ]:

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters = 3, random_state = 0).fit(iris.data)
kmeans.labels_
```

In [ ]:

```
kmeans.cluster_centers_
```

In [ ]:

```
kmeans.predict([iris.data[0],
                iris.data[100]])
```

10.3 Mean-shift聚类

class sklearn.cluster.MeanShift(

```
    bandwidth = None : 用于确定中心位置的Bandwidth大小
        如果不指定，则使用sklearn.cluster.estimate_bandwidth进行估计
    seeds = None  : array, shape = [n_samples, n_features], 初始化的中心位置
        如果不指定，则使用clustering.get_bin_seeds并结合bandwidth做初始化

    bin_seeding = False : 为True时减少seeds数以加速计算
    min_bin_freq = 1 : int, 放大筛选为种子的网格间距以加速计算

    cluster_all = True : 是否将离群值也强行归并入最近的类中
    n_jobs=1
```

)

sklearn.cluster.MeanShift类的属性：

```
    cluster_centers_ : array, [n_clusters, n_features]
    labels_ : Labels of each point
```

In [ ]:

```
from sklearn.cluster import MeanShift

meanshift = MeanShift().fit(iris.data)
meanshift.labels_
```

In [ ]:

```
meanshift.cluster_centers_
```

In [ ]:

```
meanshift = MeanShift(bandwidth = 0.8).fit(iris.data)
meanshift.labels_
```

In [ ]:

```
meanshift.predict([iris.data[0],
                   iris.data[100]])
```

In [ ]:

```
meanshift = MeanShift(bandwidth = 0.8, cluster_all = False).fit(iris.data)
meanshift.labels_
```

In [ ]:

```
pd.DataFrame(meanshift.labels_)[0].value_counts()
```

10.4 层次聚类

10.4.1 案例聚类

class sklearn.cluster.AgglomerativeClustering(

```
    n_clusters = 2 : 期望获得的类别数
    affinity = 'euclidean' : 使用的距离测量方法
        'euclidean', 'l1', 'l2', 'manhattan', 'cosine', or 'precomputed'
    memory = None, connectivity = None, compute_full_tree = 'auto'
    linkage = 'ward' : 类间距离的计算方法, {'ward', 'complete', 'average'}
        ward : 使各类的方差总和最小化
        average : 使用两个类间所有不同类案例的平均距离
        complete : 使用两个类间最远案例的距离
    pooling_func = <function mean at 0x174b938>
```

)

sklearn.cluster.AgglomerativeClustering类的属性:

```
    labels_ : array [n_samples], 各案例的类标签
    n_leaves_ : int, 聚类树的叶子数 (案例数)
    n_components_ : int, 聚类变量中潜在的成分数
    children_ : array-like, shape (n_nodes-1, 2), 各非终末节点的子节点列表
```

In [ ]:

```
from sklearn.cluster import AgglomerativeClustering

cl = AgglomerativeClustering(3)
cl.fit(iris.data)
```

In [ ]:

```
cl.labels_
```

In [ ]:

```
pd.DataFrame(cl.labels_)[0].value_counts()
```

In [ ]:

```
cl.n_leaves_
```

In [ ]:

```
cl = AgglomerativeClustering(5).fit(iris.data)
pd.DataFrame(cl.labels_)[0].value_counts()
```

In [ ]:

```
cl = AgglomerativeClustering(5, linkage = 'average').fit(iris.data)
pd.DataFrame(cl.labels_)[0].value_counts()
```

10.4.2 特征聚类

class sklearn.cluster.FeatureAgglomeration(

```
    n_clusters = 2, affinity = 'euclidean', memory = None
    connectivity = None, compute_full_tree = 'auto', linkage = 'ward'
    pooling_func = <function mean at 0x174b938>
```

)

10.5 Birch聚类

class sklearn.cluster.Birch(

```
    threshold = 0.5 : float, 每个子类的辐射半径
    branching_factor = 50 : int, 每个节点容许纳入的最大子类数
    n_clusters = 3 : int, 最终的类别数
    compute_labels = True : 是否每次拟合时都计算类别标签
    copy = True
```

)

Birch类的属性：

```
    root_ : _CFNode, CF树的根节点
    dummy_leaf_ : _CFNode, 所有叶节点的起点
    subcluster_centers_ : ndarray, 所有子类的中心位置
    subcluster_labels_ : ndarray, 所有子类的最终标签
    labels_ : ndarray, shape (n_samples,) 所有案例的最终标签
```

In [ ]:

```
from sklearn.cluster import Birch

birch = Birch(n_clusters = 3).fit(iris.data)
birch.labels_
```

In [ ]:

```
birch.subcluster_centers_
```

In [ ]:

```
birch.subcluster_labels_
```

In [ ]:

```
birch.predict([iris.data[0],
               iris.data[100]])
```

In [ ]:

```
dbscan.components_
```

10.6 DBSCAN聚类

class sklearn.cluster.DBSCAN(

```
    eps = 0.5 : float, 两个案例被归为一类的最大距离
    min_samples = 5 : int, 案例被考虑为核心案例的最小数量
    metric = 'euclidean', : string, or callable, 距离的具体算法
    metric_params = None : dict, 距离计算方法所需的其他参数
    algorithm = 'auto' : 具体使用的最近邻算法
        {'auto', 'ball_tree', 'kd_tree', 'brute'}
    leaf_size = 30 : int, BallTree或cKDTree中的最大叶子数量
    p = None : float, Minkowski距离中的指数
    n_jobs = 1
```

)

DBSCAN类的属性:

```
    core_sample_indices_ : array, shape = [n_core_samples]
    components_ : array, shape = [n_core_samples, n_features], 核心样本
    labels_ : array, shape = [n_samples], 各类别标签, 噪声样本为-1
```

注意：DBSCAN无predict方法，只有fit_predict方法

In [ ]:

```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN().fit(iris.data)
dbscan.labels_
```

In [ ]:

```
dbscan.components_
```

In [ ]:

```
dbscan.fit_predict(iris.data)
```

In [ ]:

```
# 增大距离参数
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps = 1).fit(iris.data)
dbscan.labels_
```

10.7 实战练习

将iris数据集的案例顺序彻底随机化，然后重新使用BIRCH方法进行聚类。列出随机化以后聚类结果和真实类别间的交叉表，并且和按照原顺序得到的BIRCH聚类结果和真实类别的交叉表作比较，思考案例顺序随机化处理在BIRCH方法中的重要性。

> 提示：交叉表描述和案例随机化均可以使用Pandas中的功能完成。

将iris数据集的案例顺序彻底随机化，使用K-Means方法进行聚类操作，并比较随机化前后的聚类结果，思考为什么会和BIRCH方法存在这种差异。

进一步梳理一下，在聚类分析的各种方法中，哪些方法是必须要求案例事先随机化的。

11 最近邻分析

11.1 最近邻分析的基本原理

11.2 最近邻分类

11.2.1 KNeighborsClassifier

基于每个点的k个最近邻完成实现分类。

class sklearn.neighbors.KNeighborsClassifier(