11.数据集转换与模型选择

# 1 实验目的

1.了解主成分分析法降维的实现原理，学会用PCA实现对实验数据的降维；

2.了解流水线原理，学会用sklearn.pipeline构建数据处理、模型训练预测流水线。

# 2 实验要求

本次实验后，要求同学们能：

1.利用主成分分析法减少实验数据的特征数，从而实现减少模型训练时间，或将高维数据在低维可视化；

2.用管道构建数据处理、模型训练流水线，防止交叉验证时训练集数据污染测试集；

3.用管道构建完整的流水线对数据集的数值和标签进行统一处理。

# 3 实验内容

## 3.1 主成分分析

### 3.1.1 原理

（1）降维

许多的机器学习问题涉及到的训练数据有上千甚至上百万的特征，在如此高维的空间中训练模型，不仅会导致训练非常缓慢，还让我们难以找到最好的解决方案。所幸的是，对于绝大多数现实问题，都可以通过各种手段减少特征数量，将复杂高维空间问题转化为简单问题。

比如Scikit-learn自带的手写数字数据集中，每个样本包含了64个特征，但是，手写数字的信息主要就集中在图像的中间位置，所以边缘的像素点几乎可以忽略，如图1，这就给降维提供了可能。

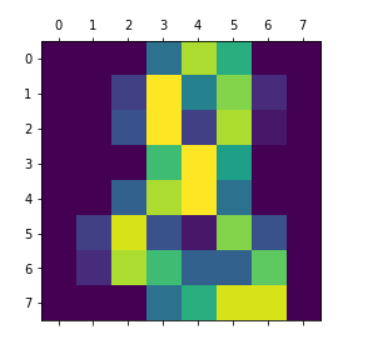


图1 手写数字图像演示

需要注意的是，降维确实能加速训练，但是也确实丢失了一部分信息，会导致性能的下降。

除了加速训练，降维对于数据可视化十分有用。相信通过前几章的学习同学们也深有体会：我们只能清晰直观地观察三维数据的分布情况。比如第4章中对鸢尾花数据集聚类的实验，我们只选择花萼的长度和宽度两个特征维度进行聚类，以便观测聚类可视化结果。

（2）PCA主成分分析

PCA主成分分析法是目前最流行的降维算法，它可以识别出能保留原数据信息最多的超平面，并把数据投影在上面。以二维空间为例，如果要将图1中的散点投影至一维，很显然投影到PC1方向比投影到PC2方向要好得多。

可以从两个角度来说明这个选择。①从保留原数据最大差异性的角度，如果投影到PC1方向，投影后的数据方差更大，保留了更大的差异性，损失了更少的信息；如果投影到PC2方向，投影后的数据方差小，只保留了非常少的差异性，投影后样本点都集中在一起，原本差异很大的样本点都变得难以区分。②从比较原数据各个样本点到投影点之间的均方距离的角度，也是PC1方向上样本点到投影点之间的均方距离最小，所以选择PC1方向更为合理。

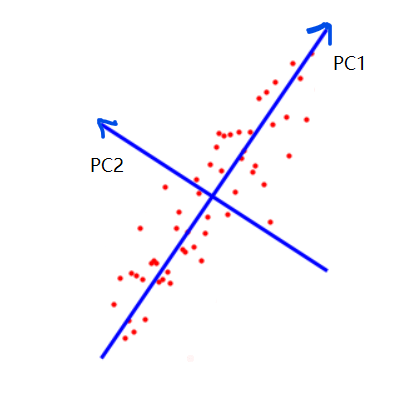


图2 投影方向选择

PCA主成分分析可以识别出把哪个方向作为投影的轴，对保留原数据的差异性贡献最大，如图1中的PC1，同时，与PC1正交的PC2是对剩余差异性贡献最大的轴。由于图1是在二维平面上演示，不存在第三个轴，但是如果是在高维空间，就还会存在与PC1、PC2都正交的对剩余差异性贡献第3大的第三个轴，同理可能还存在第四轴、第五轴……

定义第i个轴的单位向量就称为第i个主成分（PC）。一旦找出了这i个主成分，就可以将原数据投影到这个i个主成分组成的超平面上，实现降维。

用PCA进行降维时，需要选择一个合理的降维维数。除了可视化时确定要将数据降至二维或三维，一般将靠前的主成分的方差解释率依次相加，直到得到一个足够大的比例（如95%）。方差解释率表示每个主成分对于差异性的贡献度，在Scikit-learn中可用通过PCA对象实例的explained\_variance\_ratio\_查看。

下面通过对手写数字数据集进行PCA降维的实验，简单介绍Scikit-learn中PCA算法的使用方法。

### 3.1.2 实验

（1）导入依赖、加载数据

本实验用手写数字数据集进行PCA降维演示，手写数字数据集包含1797个手写数字样本，每个样本包含64维特征，分别代表每个像素点上的灰度值。

先导入依赖并加载手写数字数据，对数据进行归一化预处理。

from sklearn.datasets import load\_digits

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

from sklearn.decomposition import PCA

# 加载数据

digits\_data = load\_digits()

data = digits\_data.data

target = digits\_data.target

# 归一化

MMS = MinMaxScaler()

MMS.fit(data)

data = MMS.transform(data)

# 划分数据集

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(

data,

target,

test\_size=0.2,

random\_state=1

)

（2）进行主成分分析

使用sklearn.decomposition模块的PCA类可以创建主成分分析算法实例，其构造器主要参数及含义如下：

**n\_components：**设置用PCA算法降维保留的主成分个数，即，设置降维维数。默认为None，所有主成分都被保留。可以赋值为正整数i，表示要降至i维。也可以赋值为一个[0,1]的浮点数a，自动计算降维维数，且保留的主成分的方差解释率之和大于a。

**copy：**默认为True，表示在原数据的副本上进行主成分分析，算法不改变原数据。如果设置为False，则直接在原数据上进行主成分分析，算法完成后原数据改变。

可以用parameter\_（参数名加下划线）的形式访问PCA算法模型的成员变量，查看以下信息：

**components\_：查看保留的的主成分（降维空间各个轴的单位向量）。**

**n\_components\_：**查看保留的主成分个数，即，降维维数。

**explained\_variance\_：**查看各个主成分上的解释方差量。

**explained\_variance\_ratio\_：**查看各个主成分上的方差解释率。

**n\_features\_：**查看原训练数据的特征数量。

**n\_samples\_：**查看原训练数据样本数量。

与数据预处理的步骤类似，为了不把测试集的信息带入训练集，我们只在训练集上进行主成分分析。找出可以将数据降维至能保留95%的方差所需要的最低维度，查看主成分分析的结果，实验代码如下：

# 计算最佳维度（方差解释率达到95%视为一个好的降维）

pca = PCA(n\_components=0.95)

pca.fit(X\_train)

# 查看出成分分析结果

print("Original Train Features Number：", pca.n\_features\_)

print("95% Best Dimensions:", pca.n\_components\_)

print("Components:\n", pca.components\_)

print("Explained Variance Ratio:\n", pca.explained\_variance\_ratio\_)

（3）验证降维效果

在训练集上进行主成分分析，根据分析结果，将训练集、测试集降维。

创建SVC实例，使用降维后的训练集进行训练，查看模型在降维后的测试集上的评分。同时，用没降维的原始数据作为对照组，用SVC在原始的64维数据上进行训练，查看在原测试集上评分，比较差异。

降维前：

model = SVC()

model.fit(X\_train, Y\_train)

print("Train Score Before decomposition: ", model.score(X\_train, Y\_train))

print("Test Score Before Decomposition: ", model.score(X\_test, Y\_test))

降维后：

X\_train\_pca = pca.transform(X\_train)

X\_test\_pca = pca.transform(X\_test)

# 降维后测试

model = SVC()

model.fit(X\_train\_pca, Y\_train)

print("Train Score After decomposition: ", model.score(X\_train\_pca, Y\_train))

print("Test Score After decomposition: ", model.score(X\_test\_pca, Y\_test))

### 3.1.3 实验结果

（1）主成分分析结果

可以看到，通过主成分分析，发现只需要保留前29个主成分即可保留95%的方差量，也就是说降维至29维就可以保留训练集很大的差异性。

可以查看到找出的出成分方向向量、每个主成分的方差量如下图。

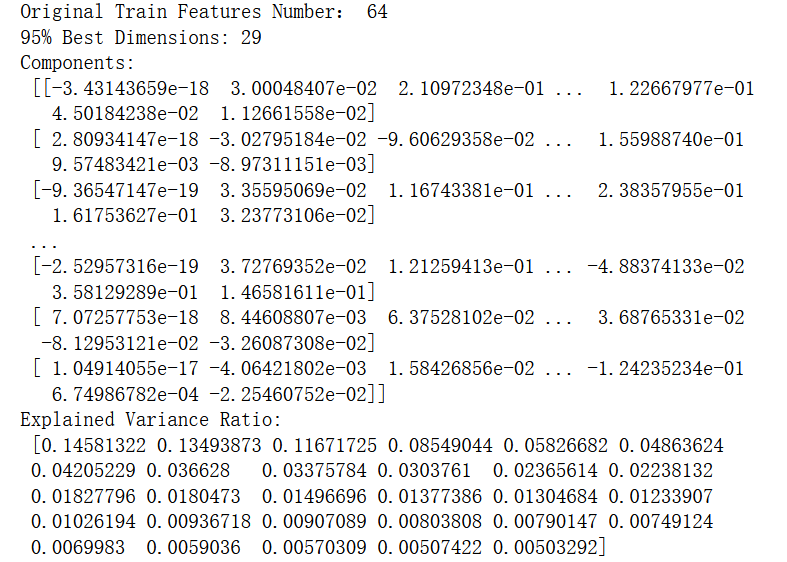


图3 PCA主成分分析结果

（2）验证降维效果

降维前（64个特征）支持向量机在训练集、测试集上的评分为：



图4 原始数据（64维）上评分

降维后（29个特征）支持向量机在训练集、测试集上的评分为：



图5 降维后的数据（29维）上评分

可以看到，利用主成分分析法（PCA）对手写数字数据进行降维，将参与训练的数据的特征从64个减少至29个，大幅度削减了训练时间。而支持向量机算法模型在降维前后的数据上得到了相近的准确度。

## 3.2 管道Pipeline

### 3.2.1 原理

在机器学习任务中，很多步骤需要按一定的顺序来执行，比如先对数据进行归一化、标准化，再进行主成分分析，再用它拟合数据集。而Pipeline管道就是将数个估算器（包括转换器、回归器、分类器等等）封装在一起，然后按顺序执行，以流水线的形式对数据进行统一操作，简化了程序，也避免了出错。

由于Scikit-learn依赖于“鸭子类型”的编译，而不是继承，只需要创建一个包含fit()、transform()、fit\_transform()三个方法的类，就可以把它看作是一个估算器加入到流水线中去。

流水线的成员方法与流水线上最后一个估计器的成员方法相同。当调用流水线的fit()方法时，会按照流水线上各个估算器的顺序依次调用他们的fit\_transform()方法，将上一个估算器的输出作为下一个估算器的参数，直到传递到最后一个估算器，只会调用它的fit()方法。

需要注意的是，Pipeline除了最后一个estimator，其他步骤都只能是转换器，而最后一个步骤可以是转换器，也可以是分类器或估计器等任意类型的estimator。

用sklearn.pipeline中的Pipeline可以创建流水线，用sklearn.pipeline中的FeatureUnion可以连接多个流水线。下面用两个实验介绍流水线的使用方法。

### 3.2.2 实验1：用流水线实现在交叉验证中标准化数据

（1）导入依赖并加载数据

在上一章，由于难以在交叉验证中对每一折的数据分别进行标准化（只在训练集上fit，在全部数据上transform），最后KNN算法在红酒数据集上交叉验证得到的性能评估很低。使用管道创建数据处理、模型训练流水线就可以轻松解决这个问题。

下面导入依赖，并加载红酒数据集。

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.datasets import load\_wine

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import cross\_val\_score

# 加载数据

wine\_data = load\_wine()

data = wine\_data.data

target = wine\_data.target

（2）创建流水线

使用sklearn.pipeline模块的Pipeline类可以将数个估计器封装成一个流水线实例，流水线实例的成员方法与最后一个估计器一致。其构造器重要参数及含义如下：

**steps：用列表指定流水线中的估计器，列表的每一个项为一个（“估算器名称”，估计器）的元组。**

**memory：设置是否在各个步骤之间启用缓存，因为每个步骤的效率可能会有较大差别，启用缓存可以加快流水线执行速度。默认为None，不执行缓存。可以通过传入字符串类型的缓存目录路径启用缓存。**

**verbose：设置是否在流水线各个估算器运行的同时打印出各自的运行时间。**

本实验中创建一个标准化转换器和一个k值为7的KNN分类器组成的流水线。创建一个不具备标准化功能的普通KNN实例作为对照，分别对KNN流水线和普通KNN分类器进行交叉验证，观察结果。

# 普通KNN实例作为对照

knn\_clf = KNeighborsClassifier(n\_neighbors=7)

# 流水线

knn\_pipeline = Pipeline([

("std\_scaler", StandardScaler()),

("knn\_clf", KNeighborsClassifier(n\_neighbors=7))

])

# 对未标准化数据对普通KNN进行交叉验证

scores = cross\_val\_score(knn\_clf, data, target, scoring="accuracy", cv=4)

print("KNN Scores: ", scores)

print("KNN Mean: ", scores.mean())

# 对带标准化功能的KNN流水线进行交叉验证

scores = cross\_val\_score(knn\_pipeline, data, target, scoring="accuracy", cv=4)

print("KNN\_pipeline Scores: ", scores)

print("KNN\_pipeline Mean: ", scores.mean())

（3）更改流水线中估算器参数

以estimatorName\_\_parameterName的形式（注意是两个下划线“\_\_”而不是单个的“\_”）可以访问流水线内部任意估算器的参数。例如在上面创建的流水线中，可以用knn\_clf\_\_ n\_neighbors访问KNN分类器的k值。这在网格搜索中十分重要，可以利用这一点，对流水线内各个估算器的各个参数进行网格搜索。

用named\_steps还可以以字典的形式查看到流水线中的各个估算器。

下面用knn\_clf\_\_n\_neighbors更改流水线中KNN分类器的k值，并用named\_steps检验参数更改是否成功。

# 更改流水线中估算器参数

print("以字典形式查看流水线中的估算器：\n", knn\_pipeline.named\_steps)

# 查看更改前的参数

print("\n更改前", knn\_pipeline.named\_steps["knn\_clf"])

# 查看更改后的参数

print("更改后", knn\_pipeline.named\_steps["knn\_clf"])

（4）实验结果

①如图3，利用管道创建含标准化转换器的KNN流水线后，避免了训练数据对测试数据的污染。在流水线中经过标准化，KNN算法准确率大幅度提升。

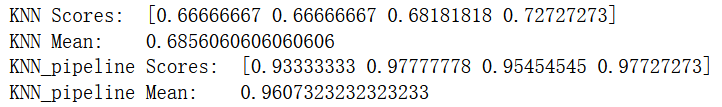


图6 用流水线进行数据标准化

②如图4，成功使用knn\_clf\_\_n\_neighbors改变了流水线中KNN分类器的参数。

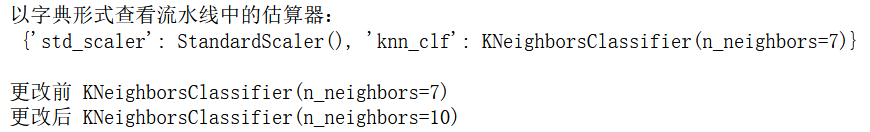


图7 更改流水线中估算器的参数

### 3.2.3 实验2：用流水线并行处理数值和分类属性

在有些任务中，数据的不同维度需要进行不同的预处理。比如在手写数字集中，对数据需要进行归一化预处理和降维等，对标签需要进行二值化处理。可以创建两条流水线，在用sklearn.pipeline中的FeatureUnion将两条流水线合并，实验步骤如下。

（1）导入依赖并加载数据

from sklearn.pipeline import Pipeline

from sklearn.pipeline import FeatureUnion

from sklearn.preprocessing import MinMaxScaler

from sklearn.preprocessing import LabelBinarizer

from sklearn.decomposition import PCA

from sklearn.datasets import load\_digits

# 加载数字图像数据集

digits = load\_digits()

（2）自定义选择转换器

自定义一个带有fit()、transform()、fit\_transform()方法的（数据-标签）选择转换器Data\_Value\_Selector（由于是鸭子类型，只要一个对象能实现fit()、transform()、fit\_transform()方法，就可以把它当做是一个转换器）。

构造时，转入一个“data”代表要选择的是数据，调用fit\_transform(digits)后返回digits数据集中“data”的内容；传入“target”代表要选择的是标签，调用fit\_transform(digits)后返回digits数据集中“target”的内容。

可以用sklearn.base下的BaseEstimator类和TransformerMixin类方便地创建自定义估算器。在自定义估算器类中添加TransformerMixin类可以得到fit()、transform()、fit\_transform()方法；添加BaseEstimator类可得到get\_params()、set\_params()方法。

自定义转换器类定义如下：

from sklearn.base import BaseEstimator, TransformerMixin

# 自定义转换器。

class Data\_Value\_Selector(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, Data\_or\_Value):

self.Data\_or\_Value = Data\_or\_Value

def fit(self, X, y=None):

return self

def transform(self, X):

return X[self.Data\_or\_Value]

（3）自定义标签二值化转换器

由于版本问题，用Scikit-learn自带的标签二值化转换器LabelBinarizer构建流水线会报错。需要编写一个自定义的标签二值化转换器MyLabelBinarizer，它的fit()、transform()、fit\_transform()方法需要能接受三个参数（直接加一个新参数y=0，调用原来的LabelBinarizer就行了）。代码如下：

from sklearn.base import TransformerMixin #gives fit\_transform method for free

class MyLabelBinarizer(TransformerMixin):

def \_\_init\_\_(self, \*args, \*\*kwargs):

self.encoder = LabelBinarizer(\*args, \*\*kwargs)

def fit(self, x, y=0):

self.encoder.fit(x)

return self

def transform(self, x, y=0):

return self.encoder.transform(x)

（4）构建子流水线

流水线1：对数据进行归一化。

# 对数据进行归一化

data\_pipeline = Pipeline([

("selector", Data\_Value\_Selector("data")),

("mms\_scaler", MinMaxScaler()),

("reduce\_dim", PCA(n\_components=0.95)),

])

流水线2：对标签进行二值化编码。

# 对标签进行编码

target\_pipeline = Pipeline([

("selector", Data\_Value\_Selector("target")),

("LB", MyLabelBinarizer()),

])

（5）合并成一整条流水线，并处理数据。

用sklearn.pipeline模块的FeatureUnion类可以将两条流水线合并，其构造器的重要参数及含义如下：

**transformer\_list：要连接的转换器列表，可以是转换器也可以是流水线。列表的元素是（“转换器名称”，转换器实例）形式的元组。**

**n\_jobs：指定并行运行的CPU内核数，默认为None，会顺序执行被连接的转换器。如果设置为-1会用全部CPU内核，被连接的转换器并行执行。**

**verbose：设置是否打印每个转换器的运行时间**

下面将上面创建的两条流水线合并在一起，用完整的流水线取处理手写数字数据集，并展示出处理后的数据。

# 将对数据和标签的操作合在一起

full\_pipeline = FeatureUnion(transformer\_list=[

("data\_pipeline", data\_pipeline),

("target\_pipeline", target\_pipeline),

])

# 处理书据

digits\_preprocessed = full\_pipeline.fit\_transform(digits)

# 打印出处理后的数据

print("data")

data = digits\_preprocessed[:, :30]

print(data)

print("\ntarget")

target = digits\_preprocessed[:, 30:]

print(target)

（6）实验结果

用处理数值和分类属性的完整流水线对手写数字数据集进行预处理，最终将数据集转变为如下形式：

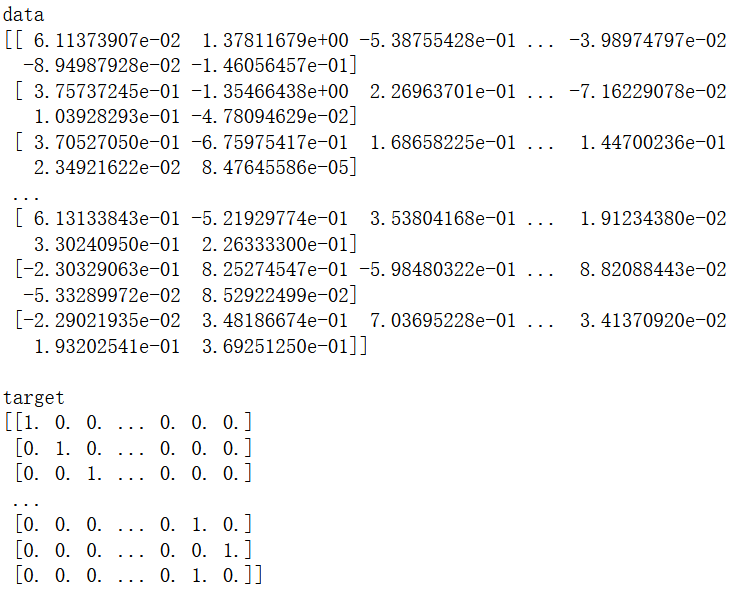


图8 用流水线来同时处理数据和数据标签