数据预处理和特征工程

一、数据预处理 Preprocessing & Impute

1.1 数据无量纲化

在机器学习算法实践中，我们往往有着将不同规格的数据转换到同一规格，或不同分布的数据转换到某个特定分布的需求，这种需求统称为将数据“无量纲化”。比如梯度和矩阵为核心的算法中，对于逻辑回归，支持向量机，神经网络等，**无量纲化可以加快求解速度**；而在距离类模型，比如K近邻，K-Means聚类中，**无量纲化可以帮我们提升模型精度**，避免某一个取值范围特别大的特征对距离计算造成影响。（一个特例是决策树和树的集成算法们，对决策树我们不需要无量纲化，决策树可以把任意数据都处理得很好。）

数据的无量纲化可以是线性的，也可以是非线性的。线性的无量纲化包括**中心化**（Zero-centered或者Mean-subtraction）处理和**缩放处理**（Scale）。中心化的本质是让所有记录减去一个固定值，即让数据样本数据平移到某个位置。缩放的本质是通过除以一个固定值，将数据固定在某个范围之中，取对数也算是一种缩放处理。

- **数据归一化**

当数据(x)按照最小值中心化后，再按极差（最大值 - 最小值）缩放，数据移动了最小值个单位，并且会被收敛到[0,1]之间，而这个过程，就叫做**数据归一化**(Normalization，又称Min-Max Scaling)。注意，Normalization是归一化，不是正则化，真正的正则化是regularization，不是数据预处理的一种手段。数据归一化公式如下：

$$x^* = \frac{x - min(x)}{max(x) - min(x)}$$

```
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

#学过Numpy的你们，能够判断data的结构嘛?
import numpy as np
X = np.array(data)
X

#使用Numpy来实现归一化
#归一化
X_nor = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_nor

#逆转归一化
X_returned = X_nor * (X.max(axis=0) - X.min(axis=0)) + X.min(axis=0)
X_returned



#如果换成表是什么样子?
import pandas as pd
x = pd.DataFrame(data)
```

```
x

#使用pandas来实现归一化
#归一化
x_nor = (x - x.min()) / (x.max() - x.min())
x_nor

#逆转归一化
x_returned = x_nor * (x.max() - x.min()) + x.min()
x_returned
```

在sklearn当中，我们使用**preprocessing.MinMaxScaler**来实现这个功能。MinMaxScaler有一个重要参数，feature_range，控制我们希望把数据压缩到的范围，默认是[0,1]。

```
from sklearn.preprocessing import MinMaxScaler

data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

#实现归一化
scaler = MinMaxScaler()                 #实例化
scaler = scaler.fit(data)               #fit，在这里本质是生成min(x)和max(x)
result = scaler.transform(data)             #通过接口导出结果
result

result_ = scaler.fit_transform(data)          #训练和导出结果一步达成

scaler.inverse_transform(result)            #将归一化后的结果逆转


#使用MinMaxScaler的参数feature_range实现将数据归一化到[0,1]以外的范围中

data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
scaler = MinMaxScaler(feature_range=[5,10])      #依然实例化
result = scaler.fit_transform(data)          #fit_transform一步导出结果
result

#当X中的特征数量非常多的时候，fit会报错并表示，数据量太大了我计算不了
#此时使用partial_fit作为训练接口
#scaler = scaler.partial_fit(data)
```

- **数据标准化**

当数据(x)按均值(μ)中心化后，再按标准差(σ)缩放，数据就变成满足均值为0，方差为1的分布，而这个过程，就叫做**数据标准化**(Standardization，又称Z-score normalization)，公式如下：

$$x^* = \frac{x - \mu}{\sigma}$$

在sklearn中**preprocessing.StandardScaler**模块可以实现这一功能。

```
from sklearn.preprocessing import StandardScaler
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

scaler = StandardScaler()                #实例化
scaler.fit(data)                      #fit，本质是生成均值和方差

scaler.mean_                         #查看均值的属性mean_
scaler.var_                          #查看方差的属性var_

x_std = scaler.transform(data)              #通过接口导出结果

x_std.mean()                         #导出的结果是一个数组，用mean()查看均值
x_std.std()                          #用std()查看方差

scaler.fit_transform(data)                 #使用fit_transform(data)一步达成结果

scaler.inverse_transform(x_std)            #使用inverse_transform逆转标准化
```

练习

```
1.生成100个随机数
2.对随机数进行升序排序
3.绘制这100个数的直方图
4.对数据进行标准化处理
5.再次绘制标准化处理过后的直方图
```

- **思考：数据标准化能否改变数据的分布?**

对于StandardScaler和MinMaxScaler来说，空值NaN会被当做是缺失值，在fit的时候忽略，在transform的时候保持缺失NaN的状态显示。并且，尽管去量纲化过程不是具体的算法，但在fit接口中，依然只允许导入至少二维数组，一维数组导入会报错。通常来说，我们输入的X会是我们的特征矩阵，现实案例中特征矩阵不太可能是一维所以不会存在这个问题。

- **StandardScaler和MinMaxScaler选哪个?**

看情况。大多数机器学习算法中，会选择StandardScaler来进行特征缩放，因为MinMaxScaler对异常值非常敏感。在PCA，聚类，逻辑回归，支持向量机，神经网络这些算法中，StandardScaler往往是最好的选择。

MinMaxScaler在不涉及距离度量、梯度、协方差计算以及数据需要被压缩到特定区间时使用广泛，比如数字图像处理中量化像素强度时，都会使用MinMaxScaler将数据压缩于[0,1]区间之中。

建议先试试看StandardScaler，效果不好换MinMaxScaler。

除了StandardScaler和MinMaxScaler之外，sklearn中也提供了各种其他缩放处理（中心化只需要一个pandas广播一下减去某个数就好了，因此sklearn不提供任何中心化功能）。比如，在希望压缩数据，却不影响数据的稀疏性时（不影响矩阵中取值为0的个数时），我们会使用MaxAbsScaler；在异常值多，噪声非常大时，我们可能会选用分位数来无量纲化，此时使用RobustScaler。更多详情请参考以下列表。

1.2 缺失值

机器学习和数据挖掘中所使用的数据，永远不可能是完美的。很多特征，对于分析和建模来说意义非凡，但对于实际收集数据的人却不是如此，因此数据挖掘之中，常常会有重要的字段缺失值很多，但又不能舍弃字段的情况。因此，数据预处理中非常重要的一项就是处理缺失值。

```
import pandas as pd
data = pd.read_csv("Narrativedata.csv")
data.head()
```

在这里，我们使用从泰坦尼克号提取出来的数据，这个数据有三个特征，一个数值型，两个字符型，标签也是字符型。（数据说明：Embarked指乘客登船港口，S是英国的Southampton，C是法国的Cherbourg，Q是爱尔兰的Queens town）从这里开始，我们就使用这个数据给大家作为例子，让大家慢慢熟悉sklearn中数据预处理的各种方式。

- **impute.SimpleImputer**

class `sklearn.impute.SimpleImputer`(*missing_values=nan*, *strategy='mean'*, *fill_value=None*, *verbose=0*, *copy=True*)

这个类是专门用来填补缺失值的。它包括四个重要参数：

参数	含义&输入
missing_values	告诉SimpleImputer，数据中的缺失值长什么样，默认空值np.nan
strategy	我们填补缺失值的策略，默认均值。 输入“mean”使用均值填补（仅对数值型特征可用） 输入“median"用中值填补（仅对数值型特征可用） 输入"most_frequent”用众数填补（对数值型和字符型特征都可用） 输入“constant"表示请参考参数“fill_value"中的值（对数值型和字符型特征都可用）
fill_value	当参数startegy为”constant"的时候可用，可输入字符串或数字表示要填充的值，常用0
copy	默认为True，将创建特征矩阵的副本，反之则会将缺失值填补到原本的特征矩阵中去。

```
data.info()

#查看缺失值
data.isnull().sum()

#填补年龄
Age = data.loc[:,"Age"].values.reshape(-1,1)      #sklearn当中特征矩阵必须是二维
Age[:20]

from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer()                    #实例化，默认均值填补
imp_median = SimpleImputer(strategy="median")      #用中位数填补
imp_0 = SimpleImputer(strategy="constant",fill_value=0) #用0填补

#fit_transform一步完成调取结果
imp_mean = imp_mean.fit_transform(Age)
imp_median = imp_median.fit_transform(Age)
imp_0 = imp_0.fit_transform(Age)

imp_mean[:20]
imp_median[:20]
imp_0[:20]

#在这里我们使用中位数填补Age
data.loc[:,"Age"] = imp_median

data.info()

#使用众数填补Embarked
Embarked = data.loc[:,"Embarked"].values.reshape(-1,1)
imp_mode = SimpleImputer(strategy = "most_frequent")
data.loc[:,"Embarked"] = imp_mode.fit_transform(Embarked)
```

```
data.info()
```

BONUS：用Pandas和Numpy进行填补其实更加简单

```
import pandas as pd
data = pd.read_csv("Narrativedata.csv")
data.head()

data.loc[:,"Age"] = data.loc[:,"Age"].fillna(data.loc[:,"Age"].median())
#.fillna 在DataFrame里面直接进行填补

data.dropna(axis=0,inplace=True)
#.dropna(axis=0)删除所有有缺失值的行，.dropna(axis=1)删除所有有缺失值的列
#参数inplace，为True表示在原数据集上进行修改，为False表示生成一个复制对象，不修改原数据，默认False
```

1.3 处理分类型特征：编码与哑变量

在机器学习中，大多数算法，比如k近邻算法，逻辑回归，支持向量机SVM等都只能够处理数值型数据，不能处理文字，在sklearn当中，除了专用来处理文字的算法，其他算法在fit的时候全部要求输入数组或矩阵，也不能够导入文字型数据（其实手写决策树和朴素贝叶斯可以处理文字，但是sklearn中规定必须导入数值型）。

然而在现实中，许多标签和特征在数据收集完毕的时候，都不是以数字来表现的。比如说，学历的取值可以是["小学"，“初中”，“高中”，"大学"]，付费方式可能包含["支付宝"，“现金”，“微信”]等等。在这种情况下，为了让数据适应算法和库，我们必须将数据进行**编码**，即是说，**将文字型数据转换为数值型**。

- **preprocessing.LabelEncoder**：标签专用，能够将分类转换为分类数值

```
from sklearn.preprocessing import LabelEncoder

y = data.iloc[:,-1]                    #要输入的是标签，不是特征矩阵，所以允许一维

le = LabelEncoder()                    #实例化
le = le.fit(y)                         #导入数据
label = le.transform(y)                #transform接口调取结果
data.iloc[:,-1] = label                #让标签等于我们运行出来的结果
data.head()

#查看相关属性
```

```
le.classes_                          #属性.classes_查看标签中究竟有多少类别
label                                #查看获取的结果label

le.fit_transform(y)                  #也可以直接fit_transform一步到位
le.inverse_transform(label)          #使用inverse_transform可以逆转


#如果不需要教学展示的话我会这么写:
from sklearn.preprocessing import LabelEncoder
data.iloc[:,-1] = LabelEncoder().fit_transform(data.iloc[:,-1])
```

- **preprocessing.OrdinalEncoder**：特征专用，能够将分类特征转换为分类数值

```
from sklearn.preprocessing import OrdinalEncoder

data_ = data.copy()
data_.iloc[:,1:-1] = OrdinalEncoder().fit_transform(data_.iloc[:,1:-1])

#接口categories_对应LabelEncoder的接口classes_，一模一样的功能
OrdinalEncoder().fit(data_.iloc[:,1:-1]).categories_

data_.head()
```

- **preprocessing.OneHotEncoder**：独热编码，创建哑变量

我们刚才已经用OrdinalEncoder把分类变量Sex和Embarked都转换成数字对应的类别了。在登船港口Embarked这一列中，我们使用[0,1,2]代表了三个不同的舱门，然而这种转换是正确的吗?

我们来思考三种不同性质的分类数据:

1） 登船港口（S，C，Q）

三种取值S，C，Q是相互独立的，彼此之间完全没有联系，表达的是S≠C≠Q的概念。这是名义变量。

2） 学历（小学，初中，高中）

三种取值不是完全独立的，我们可以明显看出，在性质上可以有高中>初中>小学这样的联系，学历有高低，但是学历取值之间却不是可以计算的，我们不能说小学 + 某个取值 = 初中。这是有序变量。

3） 体重（>45kg，>90kg，>135kg）

各个取值之间有联系，且是可以互相计算的，比如120kg - 45kg = 90kg，分类之间可以通过数学计算互相转换。这是有距变量。

然而在对特征进行编码的时候，这三种分类数据都会被我们转换为[0,1,2]，这三个数字在算法看来，是连续且可以计算的，这三个数字互不相等，有大小，并且有着可以相加相乘的联系。所以算法会把舱门，学历这样的分类特征，都误会成是体重这样的分类特征。这是说，我们把分类转换成数字的时候，忽略了数字中自带的数学性质，所以给算法传达了一些不准确的信息，而这会影响我们的建模。

类别OrdinalEncoder可以用来处理有序变量，但对于名义变量，我们只有使用哑变量的方式来处理，才能够尽量向算法传达最准确的信息：

这样的变化，让算法能够彻底领悟，原来三个取值是没有可计算性质的，是“有你就没有我”的不等概念。在我们的数据中，性别和登船港口，都是这样的名义变量。因此我们需要使用独热编码，将两个特征都转换为哑变量。

```
from sklearn.preprocessing import OneHotEncoder

X = data.iloc[:,1:-1]   #提取出所有的名义变量
result = OneHotEncoder(categories='auto').fit_transform(X).toarray() #进行独热编码
newdata = pd.concat([data,pd.DataFrame(result)],axis=1)   #合并原数据和编码后的结果
newdata.drop(["Sex","Embarked"],axis=1,inplace=True)      #删除原名义变量
newdata.columns =
["Age","Survived","Female","Male","Embarked_C","Embarked_Q","Embarked_S"] #重命名列名
newdata.head()

#=====================以上就是独热编码的完整流程==================#

#=====================下面我们一起来看具体步骤=====================#

#如果你想查看模型各种属性的话，建议fit和transform分开写
#实例化并训练模型
enc = OneHotEncoder(categories='auto').fit(X)

#transform接口调取结果
re = enc.transform(X).toarray()
re

re.shape

#依然可以还原
pd.DataFrame(enc.inverse_transform(re))

#获取模型特证名
enc.get_feature_names()

#合并原数据和独热编码
df = pd.concat([data,pd.DataFrame(re)],axis=1) #axis=1,表示跨行进行合并，也就是将量表左右相连，如果是axis=0，就是将量表上下相连
```

```
df.head()

#删除做过独热编码的特征
df.drop(['Embarked','Sex'],axis=1,inplace=True)
df.head()

#修改列名
df.columns = ['Age','Survived','Female', 'Male', 'Embarked_C', 'Embarked_Q',
'Embarked_S']
df.head()
```

特征可以做哑变量，标签也可以吗？可以，使用类sklearn.preprocessing.LabelBinarizer可以对标签做哑变量，许多算法都可以处理多标签问题（比如说决策树），但是这样的做法在现实中不常见，因此我们在这里就不赘述了。

编码与哑变量	功能	重要参数	重要属性	重要接口
.LabelEncoder	分类标签编码	N/A	.classes_：查看标签中究竟有多少类别	fit, transform, fit_transform, inverse_transform
.OrdinalEncoder	分类特征编码	N/A	.categories_：查看特征中究竟有多少类别	fit, transform, fit_transform, inverse_transform
.OneHotEncoder	独热编码，为名义变量创建哑变量	**categories**: 每个特征都有哪些类别，默认"auto"表示让算法自己判断，或者可以输入列表，每个元素都是一个列表，表示每个特征中的不同类别 **handle_unknown**：当输入了categories，且算法遇见了categories中没有写明的特征或类别时，是否报错。默认"error"表示请报错，也可以选择"ignore"表示请无视。如果选择"ignore"，则未再categories中注明的特征或类别的哑变量会全部显示为0。在逆转(inverse transform)中，未知特征或类别会被返回为None。	.categories_：查看特征中究竟有多少类别，如果是自己输入的类别，那就不需要查看了	fit, transform, fit_transform, inverse_transform, get_feature_names：查看生成的哑变量的每一列都是什么特征的什么取值

BONUS：数据类型以及常用的统计量

数据类型	数据名称	数学含义	描述	举例	可用操作
离散，定性	名义	=，≠	名义变量就是不同的名字，是用来告诉我们，这两个数据是否相同的	邮编，性别，眼睛的颜色，职工号	众数，信息熵 情形分析表或列联表，相关性分析，卡方检验
离散，定性	有序	<，>	有序变量为数据的相对大小提供信息，告诉我们数据的顺序，但数据之间大小的间隔不是具有固定意义的，因此有序变量不能加减	材料的硬度，学历	中位数，分位数，非参数相关分析（等级相关），测量系统分析，符号检验
连续，定量	有距	+，-	有距变量之间的间隔是有固定意义的，可以加减，比如，一单位量纲	日期，以摄氏度或华氏度为量纲的温度	均值，标准差，皮尔逊相关系数，t和F检验
连续，定量	比率	*，/	比变量之间的间隔和比例本身都是有意义的，既可以加减又可以乘除	以开尔文为量纲的温度，货币数量，计数，年龄，质量，长度，电流	几何平均，调和平均，百分数，变化量

1.4 处理连续型特征：二值化与分段

- **preprocessing.Binarizer**

根据阈值将数据二值化（将特征值设置为0或1），用于处理连续型变量。大于阈值的值映射为1，而小于或等于阈值的值映射为0。默认阈值为0时，特征中所有的正值都映射到1。二值化是对文本计数数据的常见操作，分析人员可以决定仅考虑某种现象的存在与否。它还可以用作考虑布尔随机变量的估计器的预处理步骤（例如，使用贝叶斯设置中的伯努利分布建模）。

```
#将年龄二值化

data_2 = data.copy()

from sklearn.preprocessing import Binarizer
X = data_2.iloc[:,0].values.reshape(-1,1)          #类为特征专用，所以不能使用一维数组
transformer = Binarizer(threshold=30).fit_transform(X)

transformer
```

- **preprocessing.KBinsDiscretizer**

这是将连续型变量划分为分类变量的类，能够将连续型变量排序后按顺序分箱后编码。总共包含三个重要参数：

参数	含义&输入
n_bins	每个特征中分箱的个数，默认5，一次会被运用到所有导入的特征
encode	编码的方式，默认“onehot” "onehot"：做哑变量，之后返回一个稀疏矩阵，每一列是一个特征中的一个类别，含有该 类别的样本表示为1，不含的表示为0 “ordinal”：每个特征的每个箱都被编码为一个整数，返回每一列是一个特征，每个特征下含 有不同整数编码的箱的矩阵 "onehot-dense"：做哑变量，之后返回一个密集数组。
strategy	用来定义箱宽的方式，默认"quantile" "uniform"：表示等宽分箱，即每个特征中的每个箱的最大值之间的差为 (特征.max() - 特征.min())/(n_bins) "quantile"：表示等位分箱，即每个特征中的每个箱内的样本数量都相同 "kmeans"：表示按聚类分箱，每个箱中的值到最近的一维k均值聚类的簇心得距离都相同

```
from sklearn.preprocessing import KBinsDiscretizer

X = data.iloc[:,0].values.reshape(-1,1)
est = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
est.fit_transform(X)
```

```
#查看转换后分的箱：变成了一列中的三箱
set(est.fit_transform(X).ravel())

est = KBinsDiscretizer(n_bins=3, encode='onehot', strategy='uniform')
#查看转换后分的箱：变成了哑变量
est.fit_transform(X).toarray()

est = KBinsDiscretizer(n_bins=3, encode='onehot', strategy='quantile')
#查看转换后分的箱：变成了哑变量,且每个类别数量基本相等
est.fit_transform(X).toarray().sum(0)
```

二、特征选择 feature_selection

当数据预处理完成后，我们就要开始进行特征工程了。

特征提取 feature extraction	特征选择 feature selection	特征创造 feature creation
从文字，图像，声音等其他非结构化信息中提取新信息作为特征。比如说，从淘宝宝贝的名称中提取出产品类别，产品颜色，是否时网红产品等等。	从所有的特征中，选择出有意义，对模型有帮助的特征，以避免必须将所有特征都导入模型去训练的情况。	把现有特征进行组合，或互相计算，得出新的特征。如果我们有一列特征是距离，一列特征是速度，我们就可以通过让两列相除，创造新的特征：通过距离所花的时间。其实，大多数的降维算法都是在做特征创造。

在做特征选择之前，有三件非常重要的事：**跟数据提供者开会！跟数据提供者开会！跟数据提供者开会！**

一定要抓住给你提供数据的人，尤其是理解业务和数据含义的人，跟他们聊一段时间。技术能够让模型起飞，前提是你和业务人员一样理解数据。所以特征选择的第一步，其实是根据我们的目标，用业务常识来选择特征。来看完整版泰坦尼克号数据中的这些特征：

	乘客编号	存活	舱位等级	姓名	性别	年龄	同船的兄弟姐妹数量	同船的父辈的数量	票号	乘客的体热指标	船舱编号	乘客登船的港口
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

其中是否存活是我们的标签。很明显，以判断“是否存活”为目的，票号，登船的舱门，乘客编号明显是无关特征，可以直接删除。姓名，舱位等级，船舱编号，也基本可以判断是相关性比较低的特征。性别，年龄，船上的亲人数量，这些应该是相关性比较高的特征。

所以，特征工程的第一步是：理解业务。

当然了，在真正的数据应用领域，比如金融，医疗，电商，我们的数据不可能像泰坦尼克号数据的特征这样少，这样明显，那如果遇见极端情况，我们无法依赖对业务的理解来选择特征，该怎么办呢？我们有四种方法可以用来选择特征：过滤法，嵌入法，包装法，和降维算法。

```
#导入数据，让我们使用digit recognizor数据来一展身手

import pandas as pd
data = pd.read_csv("digit recognizor.csv")

data.shape

X = data.iloc[:,1:]
```

```
y = data.iloc[:,0]

X.shape

"""
这个数据量相对夸张，如果使用支持向量机和神经网络，很可能会直接跑不出来。使用KNN跑一次大概需要半个小时。用这个数据举例，能更够体现特征工程的重要性。
"""
```

2.1 Filter过滤法

过滤方法通常用作预处理步骤，特征选择完全独立于任何机器学习算法。它是根据各种统计检验中的分数以及相关性的各项指标来选择特征。

2.1.1 方差过滤

- **VarianceThreshold**

相关方法	参数说明
.fit(X)	用特征矩阵X训练模型
.transform(X)	将挑选出满足要求的特征矩阵
.fit_transform(X)	用原特征矩阵训练模型，然后保留满足要求的特征并返回
.get_params()	返回模型中的参数值
.get_support(indices=False)	- indices=False,返回长度与X特征数相同的布尔数组，满足要求的为True - indices=True,返回满足要求的特征的索引
.inverse_transform(X)	逆转训练结果

这是通过特征本身的方差来筛选特征的类。比如一个特征本身的方差很小，就表示样本在这个特征上基本没有差异，可能特征中的大多数值都一样，甚至整个特征的取值都相同，那这个特征对于样本区分没有什么作用。**所以无论接下来的特征工程要做什么，都要优先消除方差为0的特征**。VarianceThreshold有重要参数**threshold**，表示方差的阈值，表示舍弃所有方差小于threshold的特征，不填默认为0，即删除所有的记录都相同的特征。

```
from sklearn.feature_selection import VarianceThreshold

VTS = VarianceThreshold()              #实例化，不填参数默认方差为0
X_var0 = VTS.fit_transform(X)            #获取删除不合格特征之后的新特征矩阵

#也可以直接写成 X = VairanceThreshold().fit_transform(X)

X_var0.shape
```

可以看见，我们已经删除了方差为0的特征，但是依然剩下了708多个特征，明显还需要进一步的特征选择。然而，如果我们知道我们需要多少个特征，方差也可以帮助我们将特征选择一步到位。比如说，我们希望留下一半的特征，那可以设定一个让特征总数减半的方差阈值，只要找到特征方差的中位数，再将这个中位数作为参数threshold的值输入就好了：

```
import numpy as np

X_fsvar = VarianceThreshold(np.median(X.var().values)).fit_transform(X)
X_fsvar.shape
```

如果想要查看具体哪些特征被保留下来了呢?

```
#导入相应包
from sklearn.feature_selection import VarianceThreshold

#训练模型
VTS = VarianceThreshold(np.median(X.var().values)) #实例化
VTS = VTS.fit(X)                                   #训练模型
X_fsvar = VTS.transform(X)                         #将x降维，只保留需要的特征

#查看模型相关接口
VTS.get_support(indices=False)    #返回与原特征长度相等的布尔索引，被留下的特征为True
VTS.get_support(indices=True)     #返回被留下的特征的索引

#提取出所有满足要求的特证名
#以下两种表达都可以
X.columns[VTS.get_support(indices=False)]
X.columns[VTS.get_support(indices=True)]

#提取出满足要求的特征矩阵
#以下两种表达都可以
X.iloc[:,VTS.get_support(indices=True)]
X.loc[:,VTS.get_support()]
```

- **方差过滤对模型的影响**

我们进行方差过滤之后，对模型效果会有怎样的影响呢？在这里，我为大家准备了KNN和随机森林分别在方差过滤前和方差过滤后运行的效果和运行时间的对比。KNN是K近邻算法中的分类算法，其原理非常简单，是利用每个样本到其他样本点的距离来判断每个样本点的相似度，然后对样本进行分类。KNN必须遍历每个特征和每个样本，因而特征越多，KNN的计算也就会越缓慢。由于这一段代码对比运行时间过长，所以我为大家贴出了代码和结果。

1. **导入模块并准备数据**

```
#KNN vs 随机森林 在不同方差过滤效果下的对比
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.model_selection import cross_val_score
import numpy as np

X = data.iloc[:,1:]
y = data.iloc[:,0]

X_fsvar = VarianceThreshold(np.median(X.var().values)).fit_transform(X)
```

我们从模块neighbors导入KNeighborsClassfier缩写为KNN，导入随机森林缩写为RFC，然后导入交叉验证模块和numpy。其中未过滤的数据是X和y，使用中位数过滤后的数据是X_fsvar，都是我们之前已经运行过的代码。

2. **KNN方差过滤前**

```
#======【TIME WARNING: 35mins +】======#
cross_val_score(KNN(),X,y,cv=5).mean()

#python中的魔法命令，可以直接使用%%timeit来计算运行这个cell中的代码所需的时间
#为了计算所需的时间，需要将这个cell中的代码运行很多次（通常是7次）后求平均值，因此运行%%timeit的时间会远远超过cell中的代码单独运行的时间

#======【TIME WARNING: 4 hours】======#
%%timeit
cross_val_score(KNN(),X,y,cv=5).mean()
```

```
[55]: cross_val_score(KNN(),X,y,cv=5).mean()

[55]: 0.9658569700264943

[56]: %%timeit
      cross_val_score(KNN(),X,y,cv=5).mean()

      33min 58s ± 43.9 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

3. **KNN方差过滤后**

```
#======【TIME WARNING: 20 mins+】======#
cross_val_score(KNN(),X_fsvar,y,cv=5).mean()

#======【TIME WARNING: 2 hours】======#
%%timeit
cross_val_score(KNN(),X,y,cv=5).mean()
```

```
[57]: cross_val_score(KNN(),X_fsvar,y,cv=5).mean()

[57]: 0.9659997478150573

[58]: %%timeit
      cross_val_score(KNN(),X_fsvar,y,cv=5).mean()

      20min ± 4min 55s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

可以看出，对于KNN，过滤后的效果十分明显：准确率稍有提升，但平均运行时间减少了10分钟，特征选择过后算法的效率上升了1/3。那随机森林又如何呢?

4. **随机森林方差过滤前**

```
cross_val_score(RFC(n_estimators=10,random_state=0),X,y,cv=5).mean()
```

```
[52]: #随机森林 - 方差过滤前
      cross_val_score(RFC(n_estimators=10,random_state=0),X,y,cv=5).mean()

[52]: 0.9380003861799541

[53]: %%timeit
      #查看一下模型运行的时间
      cross_val_score(RFC(n_estimators=10,random_state=0),X,y,cv=5).mean()

      11.5 s ± 305 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

5. **随机森林方差过滤后**

```
cross_val_score(RFC(n_estimators=10,random_state=0),X_fsvar,y,cv=5).mean()
```

[50]:
```
#随机森林 - 方差过滤后
cross_val_score(RFC(n_estimators=10,random_state=0),X_fsvar,y,cv=5).mean()
```

[50]:
```
0.9388098166696807
```

[51]:
```
%%timeit
cross_val_score(RFC(n_estimators=10,random_state=0),X_fsvar,y,cv=5).mean()
```

```
11.1 s ± 72 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

首先可以观察到的是，随机森林的准确率略逊于KNN，但运行时间却连KNN的1%都不到，只需要十几秒钟。其次，方差过滤后，随机森林的准确率也微弱上升，但运行时间却几乎是没什么变化，依然是11秒钟。

为什么随机森林运行如此之快？为什么方差过滤对随机森林没很大的有影响？

这是由于两种算法的原理中涉及到的计算量不同。最近邻算法KNN，单棵决策树，支持向量机SVM，神经网络，回归算法，都需要遍历特征或升维来进行运算，所以他们本身的运算量就很大，需要的时间就很长，因此方差过滤这样的特征选择对他们来说就尤为重要。但对于不需要遍历特征的算法，比如随机森林，它随机选取特征进行分枝，本身运算就非常快速，因此特征选择对它来说效果平平。这其实很容易理解，无论过滤法如何降低特征的数量，随机森林也只会选取固定数量的特征来建模；而K近邻算法就不同了，特征越少，距离计算的维度就越少，模型明显会随着特征的减少变得轻量。因此，过滤法的**主要对象**是：**需要遍历特征或升维的算法们**，而过滤法的**主要目的**是：**在维持算法表现的前提下，帮助算法们降低计算成本**。

思考：过滤法对随机森林无效，却对树模型有效？

从算法原理上来说，传统决策树需要遍历所有特征，计算不纯度后进行分枝，而随机森林却是随机选择特征进行计算和分枝，因此随机森林的运算更快，过滤法对随机森林无用，对决策树却有用

在sklearn中，决策树和随机森林都是随机选择特征进行分枝，但决策树在建模过程中随机抽取的特征数目却远远超过随机森林当中每棵树随机抽取的特征数目（比如说对于这个780维的数据，随机森林每棵树只会抽取10~20个特征，而决策树可能会抽取300~400个特征），因此，过滤法对随机森林无用，却对决策树有用

也因此，在sklearn中，随机森林中的每棵树都比单独的一棵决策树简单得多，高维数据下的随机森林的计算比决策树快很多。

对受影响的算法来说，我们可以将方差过滤的影响总结如下：

	阈值很小 被过滤掉得特征比较少	**阈值比较大 被过滤掉的特征有很多**
模型表现	不会有太大影响	可能变更好，代表被滤掉的特征大部分是噪音 也可能变糟糕，代表被滤掉的特征中很多都是有效特征
运行时间	可能降低模型的运行时间 基于方差很小的特征有多少 当方差很小的特征不多时 对模型没有太大影响	一定能够降低模型的运行时间 算法在遍历特征时的计算越复杂，运行时间下降得越多

在我们的对比当中，我们使用的方差阈值是特征方差的中位数，因此属于阈值比较大，过滤掉的特征比较多的情况。我们可以观察到，无论是KNN还是随机森林，在过滤掉一半特征之后，模型的精确度都上升了。这说明被我们过滤掉的特征在当前随机模式(random_state = 0)下大部分是噪音。那我们就可以保留这个去掉了一半特征的数据，来为之后的特征选择做准备。当然，如果过滤之后模型的效果反而变差了，我们就可以认为，被我们过滤掉的特征中有很多都有有效特征，那我们就放弃过滤，使用其他手段来进行特征选择。

思考：虽然随机森林算得快，但KNN的效果比随机森林更好?
调整一下n_estimators试试看吧O(∩_∩)O，随机森林是个非常强大的模型哦~

- **选取超参数threshold**

我们怎样知道，方差过滤掉的到底时噪音还是有效特征呢？过滤后模型到底会变好还是会变坏呢?

答案是：每个数据集不一样，只能自己去尝试。这里的方差阈值，其实相当于是一个超参数，要选定最优的超参数，我们可以画学习曲线，找模型效果最好的点。但现实中，我们往往不会这样去做，因为这样会耗费大量的时间。我们只会使用阈值为0或者阈值很小的方差过滤，来为我们优先消除一些明显用不到的特征，然后我们会选择更优的特征选择方法继续削减特征数量。

2.1.2 相关性过滤

方差挑选完毕之后，我们就要考虑下一个问题：相关性了。我们希望选出与标签相关且有意义的特征，因为这样的特征能够为我们提供大量信息。如果特征与标签无关，那只会白白浪费我们的计算内存，可能还会给模型带来噪音。在sklearn当中，我们有三种常用的方法来评判特征与标签之间的相关性：卡方，F检验，互信息。

2.1.2.1 卡方过滤

卡方过滤是专门针对离散型标签（即分类问题）的相关性过滤。卡方检验类**feature_selection.chi2**计算每个非负特征和标签之间的卡方统计量，并依照卡方统计量由高到低为特征排名。再结合**feature_selection.SelectKBest**这个可以输入"评分标准"来选出前K个分数最高的特征的类，我们可以借此除去最可能独立于标签，与我们分类目的无关的特征。

另外，如果卡方检验检测到某个特征中所有的值都相同，会提示我们使用方差先进行方差过滤。并且，刚才我们已经验证过，当我们使用方差过滤筛选掉一半的特征后，模型的表现是提升的。因此在这里，我们使用threshold=中位数时完成的方差过滤的数据来做卡方检验（如果方差过滤后模型的表现反而降低了，那我们就不会使用方差过滤后的数据，而是使用原数据）：

```
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

#假设在这里，已知需要300个特征
X_fschi = SelectKBest(chi2, k=300).fit_transform(X_fsvar, y)
X_fschi.shape
```

验证一下模型的效果如何：

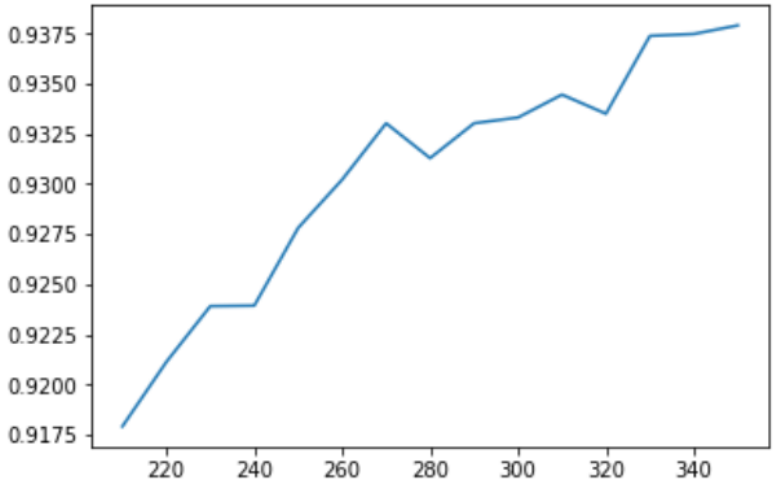
```
cross_val_score(RFC(n_estimators=10,random_state=0),X_fschi,y,cv=5).mean()
```

可以看出，模型的效果降低了，这说明我们在设定k=300的时候删除了与模型相关且有效的特征，我们的K值设置得太小，要么我们需要调整K值，要么我们必须放弃相关性过滤。当然，如果模型的表现提升，则说明我们的相关性过滤是有效的，是过滤掉了模型的噪音的，这时候我们就保留相关性过滤的结果。

选取超参数K

那如何设置一个最佳的K值呢？在现实数据中，数据量很大，模型很复杂的时候，我们也许不能先去跑一遍模型看看效果，而是希望最开始就能够选择一个最优的超参数k。那第一个方法，就是我们之前提过的学习曲线：

```
#======【TIME WARNING: 5 mins】======#

%matplotlib inline
import matplotlib.pyplot as plt

score = []
for i in range(390,200,-10):
    X_fschi = SelectKBest(chi2, k=i).fit_transform(X_fsvar, y)
    once =
cross_val_score(RFC(n_estimators=10,random_state=0),X_fschi,y,cv=5).mean()
    score.append(once)
plt.plot(range(350,200,-10),score)
plt.show()
```



通过这条曲线，我们可以观察到，随着K值的不断增加，模型的表现不断上升，这说明，K越大越好，数据中所有的特征都是与标签相关的。但是运行这条曲线的时间同样也是非常地长，接下来我们就来介绍一种更好的选择k的方法：看p值选择k。

卡方检验的本质是推测两组数据之间是否相互独立，其检验的原假设是"两组数据之间无关"。卡方检验返回卡方值和P值两个统计量，其中卡方值很难界定有效的范围，而p值，我们一般使用0.01或0.05作为显著性水平，即p值判断的边界，具体我们可以这样来看：

P值	<=0.05或0.01	>0.05或0.01
数据差异	差异不是自然形成的	这些差异是很自然的样本误差
相关性	两组数据是相关的	两组数据是相互独立的
原假设	拒绝原假设，接受备择假设	接受原假设

从特征工程的角度，我们希望选取卡方值很大，p值小于0.05的特征，即和标签是相关联的特征。而调用SelectKBest之前，我们可以直接从chi2实例化后的模型中获得各个特征所对应的卡方值和P值。

```
chi2val, pval = chi2(X_fsvar,y)

chi2val
pval
#在卡方检验结果中，卡方值越大，P值越小，原假设H0的拒绝程度越大，也就是特征与标签无关的概率越低

#k取多少？我们想要消除所有p值大于设定值，比如0.05或0.01的特征：
k = chi2val.shape[0] - (pval > 0.05).sum()

#X_fschi = SelectKBest(chi2, k=填写具体的k).fit_transform(X_fsvar, y)
#cross_val_score(RFC(n_estimators=10,random_state=0),X_fschi,y,cv=5).mean()
```

可以观察到，所有特征的p值都是0，这说明对于digit recognizor这个数据集来说，方差验证已经把所有和标签无关的特征都剔除了，或者这个数据集本身就不含与标签无关的特征。在这种情况下，舍弃任何一个特征，都会舍弃对模型有用的信息，而使模型表现下降，因此在我们对计算速度感到满意时，我们不需要使用相关性过滤来过滤我们的数据。如果我们认为运算速度太缓慢，那我们可以酌情删除一些特征，但前提是，我们必须牺牲模型的表现。接下来，我们试试看用其他的相关性过滤方法验证一下我们在这个数据集上的结论。

2.1.2.2 F检验

F检验，又称ANOVA，方差齐性检验，是用来捕捉每个特征与标签之间的线性关系的过滤方法。它即可以做回归也可以做分类，因此包含**feature_selection.f_classif**（F检验分类）和**feature_selection.f_regression**（F检验回归）两个类。其中F检验分类用于标签是离散型变量的数据，而F检验回归用于标签是连续型变量的数据。

和卡方检验一样，这两个类需要和类**SelectKBest**连用，并且我们也可以直接通过输出的统计量来判断我们到底要设置一个什么样的K。需要注意的是，F检验在数据服从正态分布时效果会非常稳定，因此如果使用F检验过滤，我们会先将数据转换成服从正态分布的方式。

F检验的本质是寻找两组数据之间的线性关系，其原假设是"数据不存在显著的线性关系"。它返回F值和p值两个统计量。和卡方过滤一样，**我们希望选取p值小于0.05或0.01的特征，这些特征与标签是显著线性相关的**，而p值大于0.05或0.01的特征则被我们认为是和标签没有显著线性关系的特征，应该被删除。以F检验的分类为例，我们继续在数字数据集上来进行特征选择：

```
from sklearn.feature_selection import f_classif

F, pval_f = f_classif(X_fsvar,y)

F

pval_f

k = F.shape[0] - (pval_f > 0.05).sum()

#X_fsF = SelectKBest(f_classif, k=填写具体的k).fit_transform(X_fsvar, y)
#cross_val_score(RFC(n_estimators=10,random_state=0),X_fsF,y,cv=5).mean()
```

得到的结论和我们用卡方过滤得到的结论一模一样：没有任何特征的p值大于0.05，所有的特征都是和标签相关的，因此我们不需要相关性过滤。

2.1.2.3 互信息法

互信息法是用来捕捉每个特征与标签之间的任意关系（包括线性和非线性关系）的过滤方法。和F检验相似，它既可以做回归也可以做分类，并且包含两个类**feature_selection.mutual_info_classif**（互信息分类）和**feature_selection.mutual_info_regression**（互信息回归）。这两个类的用法和参数都和F检验一模一样，不过互信息法比F检验更加强大，F检验只能够找出线性关系，而互信息法可以找出任意关系。

互信息法不返回p值或F值类似的统计量，它返回“每个特征与目标之间的互信息量的估计”，这个估计量在[0,1]之间取值，为0则表示两个变量独立，为1则表示两个变量完全相关。以互信息分类为例的代码如下：

```
from sklearn.feature_selection import mutual_info_classif as MIC

result = MIC(X_fsvar,y)

k = result.shape[0] - sum(result <= 0)

#X_fsmic = SelectKBest(MIC, k=填写具体的k).fit_transform(X_fsvar, y)
#cross_val_score(RFC(n_estimators=10,random_state=0),X_fsmic,y,cv=5).mean()
```

所有特征的互信息量估计都大于0，因此所有特征都与标签相关。

当然了，无论是F检验还是互信息法，大家也都可以使用学习曲线，只是使用统计量的方法会更加高效。当统计量判断已经没有特征可以删除时，无论用学习曲线如何跑，删除特征都只会降低模型的表现。如果数据量太庞大，模型太复杂，我们还是可以牺牲模型表现来提升模型速度，一切都看大家的具体需求。

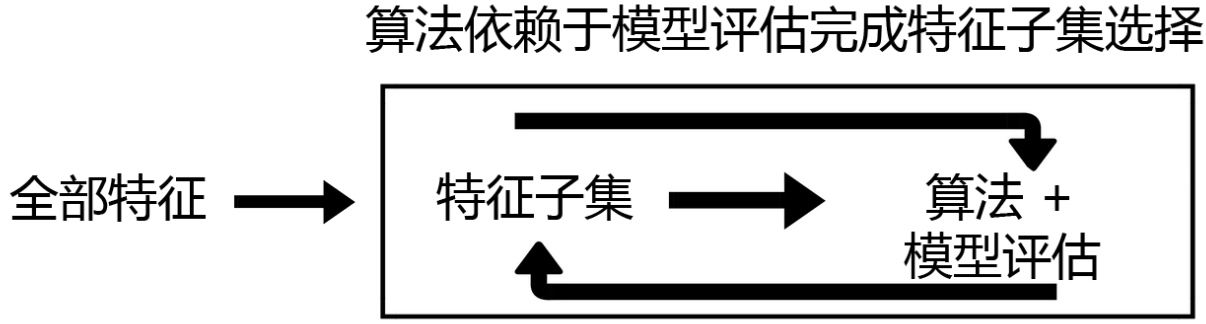
2.1.3 过滤法总结

到这里我们学习了常用的基于过滤法的特征选择，包括方差过滤，基于卡方，F检验和互信息的相关性过滤，讲解了各个过滤的原理和面临的问题，以及怎样调这些过滤类的超参数。通常来说，我会建议，先使用方差过滤，然后使用互信息法来捕捉相关性，不过了解各种各样的过滤方式也是必要的。所有信息被总结在下表，大家自取：

类	说明	超参数的选择
VarianceThreshold	方差过滤，可输入方差阈值，返回方差大于阈值的新特征矩阵	看具体数据究竟是含有更多噪声还是更多有效特征 一般就使用0或1来筛选 也可以画学习曲线或取中位数跑模型来帮助确认
SelectKBest	用来选取K个统计量结果最佳的特征，生成符合统计量要求的新特征矩阵	看配合使用的统计量
chi2	卡方检验，专用于分类算法，捕捉相关性	追求p小于显著性水平的特征
f_classif	F检验分类，只能捕捉线性相关性 要求数据服从正态分布	追求p小于显著性水平的特征
f_regression	F检验回归，只能捕捉线性相关性 要求数据服从正态分布	追求p小于显著性水平的特征
mutual_info_classif	互信息分类，可以捕捉任何相关性 不能用于稀疏矩阵	追求互信息估计大于0的特征
mutual_info_regression	互信息回归，可以捕捉任何相关性 不能用于稀疏矩阵	追求互信息估计大于0的特征

2.2 Embedded嵌入法

嵌入法是一种让算法自己决定使用哪些特征的方法，即特征选择和算法训练同时进行。在使用嵌入法时，我们先使用某些机器学习的算法和模型进行训练，得到各个特征的权值系数，根据权值系数从大到小选择特征。这些权值系数往往代表了特征对于模型的某种贡献或某种重要性，比如决策树和树的集成模型中的feature_importances_属性，可以列出各个特征对树的建立的贡献，我们就可以基于这种贡献的评估，找出对模型建立最有用的特征。因此相比于过滤法，嵌入法的结果会更加精确到模型的效用本身，对于提高模型效力有更好的效果。并且，由于考虑特征对模型的贡献，因此无关的特征（需要相关性过滤的特征）和无区分度的特征（需要方差过滤的特征）都会因为缺乏对模型的贡献而被删除掉，可谓是过滤法的进化版。



然而，嵌入法也不是没有缺点……

过滤法中使用的统计量可以使用统计知识和常识来查找范围（如p值应当低于显著性水平0.05），而嵌入法中使用的权值系数却没有这样的范围可找——我们可以说，权值系数为0的特征对模型丝毫没有作用，但当大量特征都对模型有贡献且贡献不一时，我们就很难去界定一个有效的临界值。这种情况下，模型权值系数就是我们的超参数，我们或许需要学习曲线，或者根据模型本身的某些性质去判断这个超参数的最佳值究竟应该是多少。接下来，我们一起来看，随机森林和决策树模型的嵌入法。

另外，嵌入法引入了算法来挑选特征，因此其计算速度也会和应用的算法有很大的关系。如果采用计算量很大，计算缓慢的算法，嵌入法本身也会非常耗时耗力。并且，在选择完毕之后，我们还是需要自己来评估模型。

- **feature_selection.SelectFromModel**

class `sklearn.feature_selection.SelectFromModel`(*estimator*, *threshold=None*, *prefit=False*, *norm_order=1*, *max_features=None*)

SelectFromModel是一个元变换器，可以与任何在拟合后具有coef_，feature_importances_属性或参数中可选惩罚项的评估器一起使用（比如随机森林和树模型就具有属性feature_importances_，逻辑回归就带有l1和l2惩罚项，线性支持向量机也支持l2惩罚项）。

对于有feature_importances_的模型来说，若重要性低于提供的阈值参数，则认为这些特征不重要并被移除。feature_importances_的取值范围是[0,1]，如果设置阈值很小，比如0.001，就可以删除那些对标签预测完全没贡献的特征。如果设置得很接近1，可能只有一两个特征能够被留下。

选读：使用惩罚项的模型的嵌入法

而对于使用惩罚项的模型来说，正则化惩罚项越大，特征在模型中对应的系数就会越小。当正则化惩罚项大到一定的程度的时候，部分特征系数会变成0，当正则化惩罚项继续增大到一定程度时，所有的特征系数都会趋于0。 但是我们会发现一部分特征系数会更容易先变成0，这部分系数就是可以筛掉的。也就是说，我们选择特征系数较大的特征。

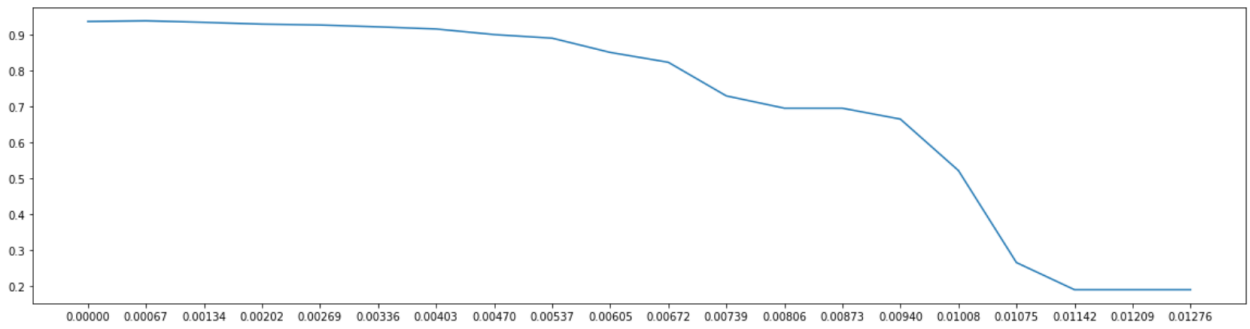
另外，支持向量机和逻辑回归使用参数C来控制返回的特征矩阵的稀疏性，参数C越小，返回的特征越少。Lasso回归，用alpha参数来控制返回的特征矩阵，alpha的值越大，返回的特征越少。

参数	说明
estimator	使用的模型评估器，只要是带feature_importances_或者coef_属性，或带有l1和l2惩罚项的模型都可以使用
threshold	特征重要性的阈值，重要性低于这个阈值的特征都将被删除
prefit	默认False，判断是否将实例化后的模型直接传递给构造函数。如果为True，则必须直接调用fit和transform，不能使用fit_transform，并且SelectFromModel不能与cross_val_score，GridSearchCV和克隆估计器的类似实用程序一起使用。
norm_order	k可输入非零整数，正无穷，负无穷，默认值为1 在评估器的coef_属性高于一维的情况下，用于过滤低于阈值的系数的向量的范数的阶数。
max_features	在阈值设定下，要选择的最大特征数。要禁用阈值并仅根据max_features选择，请设置threshold = -np.inf

我们重点要考虑的是前两个参数。在这里，我们使用随机森林为例，则需要学习曲线来帮助我们寻找最佳特征值。

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier as RFC

RFC_ = RFC(n_estimators =10,random_state=0)

X_embedded = SelectFromModel(RFC_,threshold=0.005).fit_transform(X,y)

#在这里我只想取出来有限的特征。0.005这个阈值对于有780个特征的数据来说，是非常高的阈值，因为平均每个特征只能够分到大约0.001的feature_importances_

X_embedded.shape

#模型的维度明显被降低了
#同样的，我们也可以画学习曲线来找最佳阈值

#======【TIME WARNING：10 mins】======#

import numpy as np
import matplotlib.pyplot as plt

RFC_.fit(X,y).feature_importances_

threshold = np.linspace(0,(RFC_.fit(X,y).feature_importances_).max(),20)

score = []
for i in threshold:
    X_embedded = SelectFromModel(RFC_,threshold=i).fit_transform(X,y)
```

```
    once = cross_val_score(RFC_,X_embedded,y,cv=5).mean()
    score.append(once)
plt.plot(threshold,score)
plt.show()
```



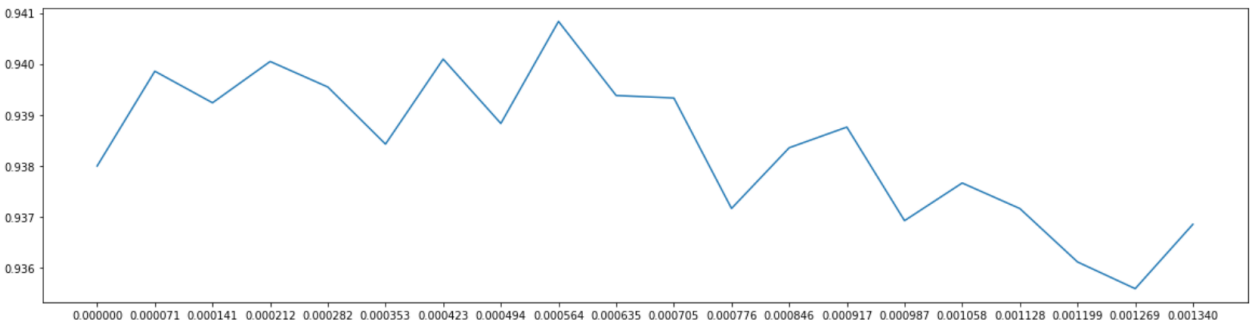
从图像上来看，随着阈值越来越高，模型的效果逐渐变差，被删除的特征越来越多，信息损失也逐渐变大。但是在0.00134之前，模型的效果都可以维持在0.93以上，因此我们可以从中挑选一个数值来验证一下模型的效果。

```
X_embedded = SelectFromModel(RFC_,threshold=0.00067).fit_transform(X,y)
X_embedded.shape

cross_val_score(RFC_,X_embedded,y,cv=5).mean()
```

可以看出，特征个数瞬间缩小到324多，这比我们在方差过滤的时候选择中位数过滤出来的结果392列要小，并且交叉验证分数0.9399高于方差过滤后的结果0.9388，这是由于嵌入法比方差过滤更具体到模型的表现的缘故，换一个算法，使用同样的阈值，效果可能就没有这么好了。

和其他调参一样，我们可以在第一条学习曲线后选定一个范围，使用细化的学习曲线来找到最佳值：

```
#======【TIME WARNING: 10 mins】======#
score2 = []
for i in np.linspace(0,0.00134,20):
    X_embedded = SelectFromModel(RFC_,threshold=i).fit_transform(X,y)
    once = cross_val_score(RFC_,X_embedded,y,cv=5).mean()
    score2.append(once)
plt.figure(figsize=[20,5])
plt.plot(np.linspace(0,0.00134,20),score2)
plt.xticks(np.linspace(0,0.00134,20))
plt.show()
```



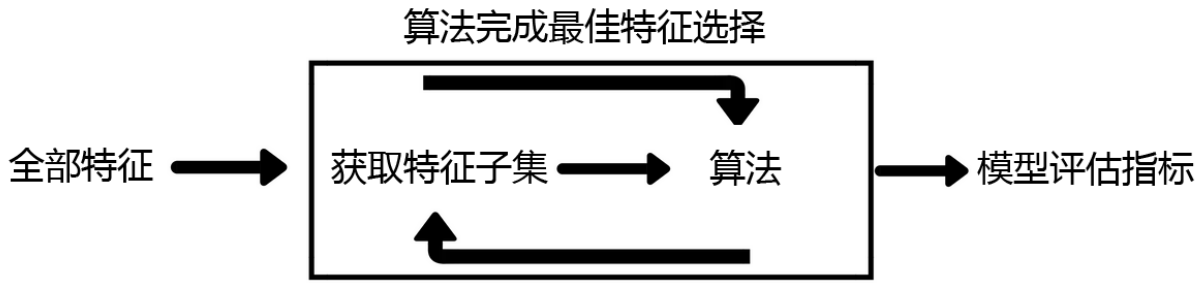
查看结果，果然0.00067并不是最高点，真正的最高点0.000564已经将模型效果提升到了94%以上。我们使用0.000564来跑一跑我们的SelectFromModel：

```
X_embedded = SelectFromModel(RFC_,threshold=0.000564).fit_transform(X,y)
X_embedded.shape

cross_val_score(RFC_,X_embedded,y,cv=5).mean()

#=====【TIME WARNING：2 min】=====#
#我们可能已经找到了现有模型下的最佳结果，如果我们调整一下随机森林的参数呢?
cross_val_score(RFC(n_estimators=100,random_state=0),X_embedded,y,cv=5).mean()
```

得出的特征数目依然小于方差筛选，并且模型的表现也比没有筛选之前更高，已经完全可以和计算一次半小时的KNN相匹敌（KNN的准确率是96.58%），接下来再对随机森林进行调参，准确率应该还可以再升高不少。可见，在嵌入法下，我们很容易就能够实现特征选择的目标：减少计算量，提升模型表现。因此，比起要思考很多统计量的过滤法来说，嵌入法可能是更有效的一种方法。然而，在算法本身很复杂的时候，过滤法的计算远远比嵌入法要快，所以大型数据中，我们还是会优先考虑过滤法。

2.3 Wrapper包装法

包装法也是一个特征选择和算法训练同时进行的方法，与嵌入法十分相似，它也是依赖于算法自身的选择，比如coef_属性或feature_importances_属性来完成特征选择。但不同的是，我们往往使用一个目标函数作为黑盒来帮助我们选取特征，而不是自己输入某个评估指标或统计量的阈值。包装法在初始特征集上训练评估器，并且通过coef_属性或通过feature_importances_属性获得每个特征的重要性。然后，从当前的一组特征中修剪最不重要的特征。在修剪的集合上递归地重复该过程，直到最终到达所需数量的要选择的特征。区别于过滤法和嵌入法的一次训练解决所有问题，包装法要使用特征子集进行多次训练，因此它所需要的计算成本是最高的。



注意，在这个图中的“算法”，指的不是我们最终用来导入数据的分类或回归算法（即不是随机森林），而是专业的数据挖掘算法，即我们的目标函数。这些数据挖掘算法的核心功能就是选取最佳特征子集。

最典型的目标函数是递归特征消除法（Recursive feature elimination, 简写为RFE）。它是一种贪婪的优化算法，旨在找到性能最佳的特征子集。 它反复创建模型，并在每次迭代时保留最佳特征或剔除最差特征，下一次迭代时，它会使用上一次建模中没有被选中的特征来构建下一个模型，直到所有特征都耗尽为止。 然后，它根据自己保留或剔除特征的顺序来对特征进行排名，最终选出一个最佳子集。包装法的效果是所有特征选择方法中最利于提升模型表现的，它可以使用很少的特征达到很优秀的效果。除此之外，在特征数目相同时，包装法和嵌入法的效果能够匹敌，不过它比嵌入法算得更见缓慢，所以也

不适用于太大型的数据。相比之下，包装法是最能保证模型效果的特征选择方法。

- **feature_selection.RFE**

class `sklearn.feature_selection.RFE`(*estimator*, *n_features_to_select=None*, *step=1*, *verbose=0*)

参数**estimator**是需要填写的实例化后的评估器，**n_features_to_select**是想要选择的特征个数，**step**表示每次迭代中希望移除的特征个数。除此之外，RFE类有两个很重要的属性:

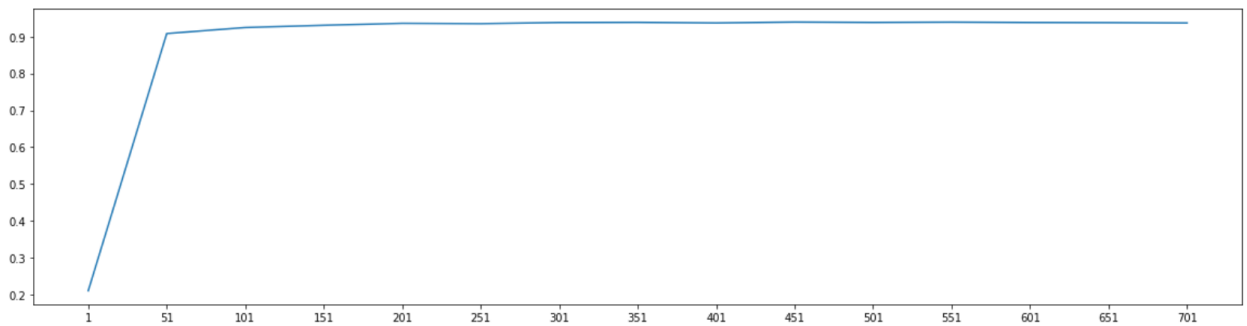
- **.support_**：返回所有的特征的是否最后被选中的布尔矩阵
- **.ranking_**：返回特征的按数次迭代中综合重要性的排名

类feature_selection.RFECV会在交叉验证循环中执行RFE以找到最佳数量的特征，增加参数cv，其他用法都和RFE一模一样。

```
from sklearn.feature_selection import RFE
RFC_ = RFC(n_estimators =10,random_state=0)
selector = RFE(RFC_, n_features_to_select=340, step=50).fit(X, y)

selector.support_.sum()

selector.ranking_

X_wrapper = selector.transform(X)

cross_val_score(RFC_,X_wrapper,y,cv=5).mean()
```

我们也可以对包装法画学习曲线：

```
#======【TIME WARNING: 15 mins】======#

score = []
for i in range(1,751,50):
    X_wrapper = RFE(RFC_,n_features_to_select=i, step=50).fit_transform(X,y)
    once = cross_val_score(RFC_,X_wrapper,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(1,751,50),score)
plt.xticks(range(1,751,50))
plt.show()
```



明显能够看出，在包装法下面，应用50个特征时，模型的表现就已经达到了90%以上，比嵌入法和过滤法都高效很多。我们可以放大图像，寻找模型变得非常稳定的点来画进一步的学习曲线（就像我们在嵌入法中做的那样）。如果我们此时追求的是最大化降低模型的运行时间，我们甚至可以直接选择50作为特征的数目，这是一个在缩减了94%的特征的基础上，还能保证模型表现在90%以上的特征组合，不可谓不高效。

同时，我们提到过，在特征数目相同时，包装法能够在效果上匹敌嵌入法。试试看如果我们也使用340作为特征数目，运行一下，可以感受一下包装法和嵌入法哪一个的速度更加快。由于包装法效果和嵌入法相差不多，在更小的范围内使用学习曲线，我们也可以将包装法的效果调得很好，大家可以去试试看。

2.4 特征选择总结

至此，我们讲完了降维之外的所有特征选择的方法。这些方法的代码都不难，但是每种方法的原理都不同，并且都涉及到不同调整方法的超参数。经验来说，过滤法更快速，但更粗糙。包装法和嵌入法更精确，比较适合具体到算法去调整，但计算量比较大，运行时间长。当数据量很大的时候，优先使用方差过滤和互信息法调整，再上其他特征选择方法。使用逻辑回归时，优先使用嵌入法。使用支持向量机时，优先使用包装法。迷茫的时候，从过滤法走起，看具体数据具体分析。

其实特征选择只是特征工程中的第一步。真正的高手，往往使用特征创造或特征提取来寻找高级特征。在Kaggle之类的算法竞赛中，很多高分团队都是在高级特征上做文章，而这是比调参和特征选择更难的，提升算法表现的高深方法。下面我们一起来看一下特征创造。

三、降维算法

大多数的降维算法都属于特征创造范畴，这里我们一起来学习sklearn中的一些降维算法。

特征提取 **feature extraction**	**特征选择** **feature selection**	**特征创造** **feature creation**
从文字，图像，声音等其他非结构化信息中提取新信息作为特征。比如说，从淘宝宝贝的名称中提取出产品类别，产品颜色，是否时网红产品等等。	从所有的特征中，选择出有意义，对模型有帮助的特征，以避免必须将所有特征都导入模型去训练的情况。	把现有特征进行组合，或互相计算，得出新的特征。如果我们有一列特征是距离，一列特征是速度，我们就可以通过让两列相除，创造新的特征：通过距离所花的时间。其实，大多数的降维算法都是在做特征创造。

3.1 sklearn中的降维算法

sklearn中降维算法都被包括在模块decomposition中，这个模块本质是一个矩阵分解模块。在过去的十年中，如果要讨论算法进步的先锋，矩阵分解可以说是独树一帜。矩阵分解可以用在降维，深度学习，聚类分析，数据预处理，低纬度特征学习，推荐系统，大数据分析等领域。在2006年，Netflix曾经举办了一个奖金为100万美元的推荐系统算法比赛，最后的获奖者就使用了矩阵分解中的明星：奇异值分解SVD。

类	说明
主成分分析	
decomposition.PCA	主成分分析（PCA）
decomposition.IncrementalPCA	增量主成分分析（IPCA）
decomposition.KernelPCA	核主成分分析（KPCA）
decomposition.MiniBatchSparsePCA	小批量稀疏主成分分析
decomposition.SparsePCA	稀疏主成分分析（SparsePCA）
decomposition.TruncatedSVD	截断的SVD（aka LSA）
因子分析	
decomposition.FactorAnalysis	因子分析（FA）
独立成分分析	
decomposition.FastICA	独立成分分析的快速算法
字典学习	
decomposition.DictionaryLearning	字典学习
decomposition.MiniBatchDictionaryLearning	小批量字典学习
decomposition.dict_learning	字典学习用于矩阵分解
decomposition.dict_learning_online	在线字典学习用于矩阵分解
高级矩阵分解	
decomposition.LatentDirichletAllocation	具有在线变分贝叶斯算法的隐含狄利克雷分布
decomposition.NMF	非负矩阵分解（NMF）
其他矩阵分解	
decomposition.SparseCoder	稀疏编码

SVD和主成分分析PCA都属于矩阵分解算法中的入门算法，都是通过分解特征矩阵来进行降维，它们也是我们j接下来要讲解的重点。

在降维过程中，我们会减少特征的数量，这意味着删除数据，数据量变少则表示模型可以获取的信息会变少，模型的表现可能会因此受影响。同时，在高维数据中，必然有一些特征是不带有有效的信息的（比如噪音），或者有一些特征带有的信息和其他一些特征是重复的（比如一些特征可能会线性相关）。我们希望能够找出一种办法来帮助我们衡量特征上所带的信息量，让我们在降维的过程中，能够**既减少特征的数量，又保留大部分有效信息**——将那些带有重复信息的特征合并，并删除那些带无效信息的特征等等——逐渐创造出能够代表原特征矩阵大部分信息的，特征更少的，新特征矩阵。

在上面的特征选择中，我们讲了重要的特征选择方法：方差过滤。如果一个特征的方差很小，则意味着这个特征上很可能有大量取值都相同（比如90%都是1，只有10%是0，甚至100%是1），那这一个特征的取值对样本而言就没有区分度，这种特征就不带有有效信息。从方差的这种应用就可以推断出，如果一个特征的方差很大，则说明这个特征上带有大量的信息。因此，在降维中，**PCA使用的信息量衡量指标，就是样本方差，又称可解释性方差，方差越大，特征所带的信息量越多。**

$$Var = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{x})^2$$

Var代表一个特征的方差，n代表样本量，xi代表一个特征中的每个样本取值，xhat代表这一列样本的均值。

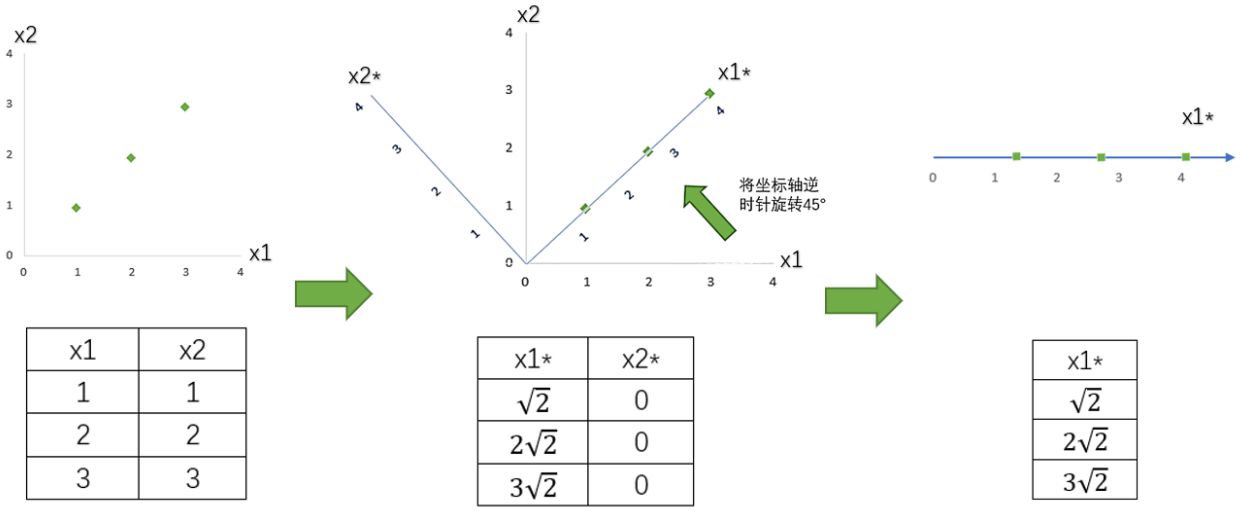
面试高危问题

方差计算公式中为什么除数是n-1?
这是为了得到样本方差的无偏估计，更多大家可以自己去探索~

3.2 降维究竟是怎样实现的?

class `sklearn.decomposition.PCA`(*n_components=None, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', random_state=None*)

PCA作为矩阵分解算法的核心算法，其实没有太多参数，但不幸的是每个参数的意义和运用都很难，因为几乎每个参数都涉及到高深的数学原理。为了参数的运用和意义变得明朗，我们来看一组简单的二维数据的降维。



x1	x2
1	1
2	2
3	3

x1*	x2*
$\sqrt{2}$	0
$2\sqrt{2}$	0
$3\sqrt{2}$	0

x1*
$\sqrt{2}$
$2\sqrt{2}$
$3\sqrt{2}$

我们现在有一组简单的数据，有特征x1和x2，三个样本数据的坐标点分别为(1,1)，(2,2)，(3,3)。我们可以让x1和x2分别作为两个特征向量，很轻松地用一个二维平面来描述这组数据。这组数据现在每个特征的均值都为2，方差则等于：

$$x1\_var = x2\_var = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{2} = 1$$

每个特征的数据一模一样，因此方差也都为1，数据的方差总和是2。

现在我们的目标是：只用一个特征向量来描述这组数据，即将二维数据降为一维数据，并且尽可能地保留信息量，即让数据的总方差尽量靠近2。于是，我们将原本的直角坐标系逆时针旋转45°，形成了新的特征向量x1*和x2*组成的新平面，在这个新平面中，三个样本数据的坐标点可以表示为$(\sqrt{2}, 0)$，$(2\sqrt{2}, 0)$，$(3\sqrt{2}, 0)$。可以注意到，x2*上的数值此时都变成了0，因此x2*明显不带有任何有效信息了（此时x2*的方差为0了）。此时，x1*特征上的数据均值是$2\sqrt{2}$，而方差则可表示成：

$$x1^*\_var = \frac{(\sqrt{2}-2\sqrt{2})^2 + (2\sqrt{2}-2\sqrt{2})^2 + (3\sqrt{2}-2\sqrt{2})^2}{2} = 2$$

x2*上的数据均值为0，方差也为0。

此时，我们根据信息含量的排序，取信息含量最大的一个特征，因为我们想要的是一维数据。所以我们可以将x2*删除，同时也删除图中的x2*特征向量，剩下的x1*就代表了曾经需要两个特征来代表的三个样本点。通过旋转原有特征向量组成的坐标轴来找到新特征向量和新坐标平面，我们将三个样本点的信息压缩到了一条直线上，实现了二维变一维，并且尽量保留原始数据的信息。一个成功的降维，就实现了。

不难注意到，在这个降维过程中，有几个重要的步骤：

步骤	**2维特征矩阵**	**n维特征矩阵**
1	输入原数据，结构为 (3,2) 找出原本的2个特征对应的直角坐标系，本质是找出这2个特征构成的2维平面	输入原数据，结构为 (m,n) 找出原本的n个特征向量构成的n维空间V
2	决定降维后的特征数量：1	决定降维后的特征数量：k
3	旋转，找出一个新坐标系 本质是找出2个新的特征向量，以及它们构成的新2维平面 新特征向量让数据能够被压缩到少数特征上，并且总信息量不损失太多	通过某种变化，找出n个新的特征向量，以及它们构成的新n维空间V
4	找出数据点在新坐标系上，2个新坐标轴上的坐标	找出原始数据在新特征空间V中的n个新特征向量上对应的值，即“将数据映射到新空间中”
5	选取第1个方差最大的特征向量，删掉没有被选中的特征，成功将2维平面降为1维	选取前k个信息量最大的特征，删掉没有被选中的特征，成功将n维空间V降为k维

在步骤3当中，我们用来**找出n个新特征向量，让数据能够被压缩到少数特征上并且总信息量不损失太多的技术就是矩阵分解**。PCA和SVD是两种不同的降维算法，但他们都遵从上面的过程来实现降维，只是两种算法中矩阵分解的方法不同，信息量的衡量指标不同罢了。PCA使用方差作为信息量的衡量指标，并且特征值分解来找出空间V。降维时，它会通过一系列数学的神秘操作（比如说，产生协方差矩阵$\frac{1}{n}XX^T$）将特征矩阵X分解为以下三个矩阵，其中Q和Q^{-1}是辅助的矩阵，Σ是一个对角矩阵（即除了对角线上有值，其他位置都是0的矩阵），其对角线上的元素就是方差。降维完成之后，PCA找到的每个新特征向量就叫做“主成分”，而被丢弃的特征向量被认为信息量很少，这些信息很可能就是噪音。

$$X \rightarrow \text{数学神秘的宇宙} \rightarrow Q\Sigma Q^{-1}$$

而SVD使用奇异值分解来找出空间V，其中Σ也是一个对角矩阵，不过它对角线上的元素是奇异值，这也是SVD中用来衡量特征上的信息量的指标。U和V^{T}分别是左奇异矩阵和右奇异矩阵，也都是辅助矩阵。

$$X \rightarrow \text{另一个数学神秘的宇宙} \rightarrow U\Sigma V^T$$

在数学原理中，无论是PCA和SVD都需要遍历所有的特征和样本来计算信息量指标。并且在矩阵分解的过程之中，会产生比原来的特征矩阵更大的矩阵，比如原数据的结构是(m,n)，在矩阵分解中为了找出最佳新特征空间V，可能需要产生(n,n)，(m,m)大小的矩阵，还需要产生协方差矩阵去计算更多的信息。而现在无论是Python还是R，或者其他的任何语言，在大型矩阵运算上都不是特别擅长，无论代码如何简化，我们不可避免地要等待计算机去完成这个非常庞大的数学计算过程。因此，降维算法的计算量很大，运行比较缓慢，但无论如何，它们的功能无可替代，它们依然是机器学习领域的宠儿。

思考：PCA和特征选择技术都是特征工程的一部分，它们有什么不同?

特征工程中有三种方式：特征提取，特征选择和特征创造。仔细观察上面的降维例子和上周我们讲解过的特征选择，你发现有什么不同了吗?

特征选择是从已存在的特征中选取携带信息最多的，选完之后的特征依然具有可解释性，我们依然知道这个特征在原数据的哪个位置，代表着原数据上的什么含义。

而PCA，是将已存在的特征进行压缩，降维完毕后的特征不是原本的特征矩阵中的任何一个特征，而是通过某些方式组合起来的新特征。通常来说，**在新的特征矩阵生成之前，我们无法知晓PCA都建立了怎样的新特征向量，新特征矩阵生成之后也不具有可读性**，我们无法判断新特征矩阵的特征是从原数据中的什么特征组合而来，新特征虽然带有原始数据的信息，却已经不是原数据上代表着的含义了。以PCA为代表的降维算法因此是特征创造（feature creation，或feature construction）的一种。

可以想见，PCA一般不适用于探索特征和标签之间的关系的模型（如线性回归），因为无法解释的新特征和标签之间的关系不具有意义。在线性回归模型中，我们使用特征选择。

3.3 重要参数n_components

n_components是我们降维后需要的维度，即降维后需要保留的特征数量，降维流程中第二步里需要确认的k值，一般输入[0, min(X.shape)]范围中的整数。一说到K，大家可能都会想到，类似于KNN中的K和随机森林中的n_estimators，这是一个需要我们人为去确认的超参数，并且我们设定的数字会影响到模型的表现。如果留下的特征太多，就达不到降维的效果，如果留下的特征太少，那新特征向量可能无法容纳原始数据集中的大部分信息，因此，n_components既不能太大也不能太小。那怎么办呢?

可以先从我们的降维目标说起：如果我们希望可视化一组数据来观察数据分布，我们往往将数据降到三维以下，很多时候是二维，即n_components的取值为2。

3.3.1 鸢尾花数据集的可视化

```
#导入模块和包
from sklearn.datasets import load_iris   #导入sklearn中内置的鸢尾花数据集
from sklearn.decomposition import PCA    #导入PCA降维算法
import matplotlib.pyplot as plt          #绘图包

#提取数据集
iris = load_iris()
y = iris.target
X = iris.data

#调用PCA建模
pca = PCA(n_components=2)          #实例化
pca = pca.fit(X)                   #拟合模型
X_dr = pca.transform(X)            #获取新矩阵

#绘制可视化图形
#要将三种鸢尾花的数据分布显示在二维平面坐标系中，对应的两个坐标（两个特征向量）应该是三种鸢尾花降维后的x1和x2，怎样才能取出三种鸢尾花下不同的x1和x2呢?

X_dr[y == 0, 0] #这里是布尔索引，看出来了么?

#要展示三中分类的分布，需要对三种鸢尾花分别绘图
#可以写成三行代码，也可以写成for循环

"""
plt.figure()
plt.scatter(X_dr[y==0, 0], X_dr[y==0, 1], c="red", label=iris.target_names[0])
plt.scatter(X_dr[y==1, 0], X_dr[y==1, 1], c="black",
label=iris.target_names[1])
plt.scatter(X_dr[y==2, 0], X_dr[y==2, 1], c="orange",
label=iris.target_names[2])
plt.legend()
plt.title('PCA of IRIS dataset')
plt.show()
"""
```

```
colors = ['red', 'black', 'orange']
iris.target_names

plt.figure()
for i in [0, 1, 2]:
    plt.scatter(X_dr[y == i, 0]
                ,X_dr[y == i, 1]
                ,alpha=.7
                ,c=colors[i]
                ,label=iris.target_names[i]
               )
plt.legend()
plt.title('PCA of IRIS dataset')
plt.show()
```

鸢尾花的分布被展现在我们眼前了，明显这是一个分簇的分布，并且每个簇之间的分布相对比较明显，也许versicolor和virginia这两种花之间会有一些分类错误，但setosa肯定不会被分错。这样的数据很容易分类，可以遇见，KNN，随机森林，神经网络，朴素贝叶斯，Adaboost这些分类器在鸢尾花数据集上，未调整的时候都可以有95%上下的准确率。

- **探索降维后的数据**

```
#属性explained_variance_，查看降维后每个新特征向量上所带的信息量大小（可解释性方差的大小）
pca.explained_variance_

#属性explained_variance_ratio，查看降维后每个新特征向量所占的信息量占原始数据总信息量的百分比
#又叫做可解释方差贡献率
pca.explained_variance_ratio_
#大部分信息都被有效地集中在了第一个特征上

pca.explained_variance_ratio_.sum()
```

3.3.2 选择最好的n_components

- **累积可解释方差贡献率曲线**

当参数n_components中不填写任何值，则默认返回min(X.shape)个特征，一般来说，样本量都会大于特征数目，所以什么都不填就相当于转换了新特征空间，但没有减少特征的个数。一般来说，不会使用这种输入方式。但我们却可以使用这种输入方式来画出累计可解释方差贡献率曲线，以此选择最好的n_components的整数取值。

累积可解释方差贡献率曲线是一条以降维后保留的特征个数为横坐标，降维后新特征矩阵捕捉到的可解释方差贡献率为纵坐标的曲线，能够帮助我们决定n_components最好的取值。

```
import numpy as np
pca_line = PCA().fit(X)
plt.plot([1,2,3,4],np.cumsum(pca_line.explained_variance_ratio_))
plt.xticks([1,2,3,4]) #这是为了限制坐标轴显示为整数
plt.xlabel("number of components after dimension reduction")
plt.ylabel("cumulative explained variance ratio")
plt.show()
```

- **最大似然估计自选超参数**

除了输入整数，n_components还有哪些选择呢？之前我们提到过，矩阵分解的理论发展在业界独树一帜，勤奋智慧的数学大神Minka, T.P.在麻省理工学院媒体实验室做研究时找出了让PCA用最大似然估计(maximum likelihood estimation)自选超参数的方法，输入“mle”作为n_components的参数输入，就可以调用这种方法。

```
pca_mle = PCA(n_components="mle")
pca_mle = pca_mle.fit(X)
X_mle = pca_mle.transform(X)

X_mle
#可以发现，mle为我们自动选择了3个特征

pca_mle.explained_variance_ratio_.sum()
#得到了比设定2个特征时更高的信息含量，对于鸢尾花这个很小的数据集来说，3个特征对应这么高的信息含量，并不需要去纠结于只保留2个特征，毕竟三个特征也可以可视化
```

- **按信息量占比选超参数**

输入[0,1]之间的浮点数，并且让参数svd_solver =='full'，表示希望降维后的总解释性方差占比大于n_components指定的百分比，即是说，希望保留百分之多少的信息量。比如说，如果我们希望保留97%的信息量，就可以输入n_components = 0.97，PCA会自动选出能够让保留的信息量超过97%的特征数量。

```
pca_f = PCA(n_components=0.97,svd_solver="full")
pca_f = pca_f.fit(X)
X_f = pca_f.transform(X)

pca_f.explained_variance_ratio_
```

3.4 案例：PCA对手写数字数据集的降维

还记得前面我们在讲解特征选择的时候，使用的手写数字的数据集吗？数据集结构为(42000, 784)，用KNN跑一次半小时，得到准确率在96.6%上下，用随机森林跑一次12秒，准确率在93.8%，虽然KNN效果好，但由于数据量太大，KNN计算太缓慢，所以我们不得不选用随机森林。我们使用了各种技术对手写数据集进行特征选择，最后使用嵌入法SelectFromModel选出了324个特征，将随机森林的效果也调到了96%以上。但是，因为数据量依然巨大，还是有300多个特征。现在，我们就来试着用PCA处理一下这个数据，看看效果如何。

1. **导入需要的模块和库**

```
from sklearn.decomposition import PCA
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

2. **导入数据，探索数据**

```
data = pd.read_csv("digit recognizor.csv")

X = data.iloc[:,1:]
y = data.iloc[:,0]

X.shape
```

3. **画累计方差贡献率曲线，找最佳降维后维度的范围**

```
pca_line = PCA().fit(X)
plt.figure(figsize=[20,5])
plt.plot(np.cumsum(pca_line.explained_variance_ratio_))
plt.xlabel("number of components after dimension reduction")
plt.ylabel("cumulative explained variance ratio")
plt.show()
```

4. **降维后维度的学习曲线，继续缩小最佳维度的范围**

```
#======【TIME WARNING：2mins 30s】======#

score = []
for i in range(1,101,10):
    X_dr = PCA(i).fit_transform(X)
    once = cross_val_score(RFC(n_estimators=10,random_state=0)
                           ,X_dr,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(1,101,10),score)
plt.show()
```

5. **细化学习曲线，找出降维后的最佳维度**

```
#======【TIME WARNING：2mins 30s】======#

score = []
for i in range(10,25):
    X_dr = PCA(i).fit_transform(X)
    once =
cross_val_score(RFC(n_estimators=10,random_state=0),X_dr,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(10,25),score)
plt.show()
```

6. **导入找出的最佳维度进行降维，查看模型效果**

```
X_dr = PCA(23).fit_transform(X)

#======【TIME WARNING:1mins 30s】======#
cross_val_score(RFC(n_estimators=100,random_state=0),X_dr,y,cv=5).mean()
```

模型效果还好，跑出了94.56%的水平，但还是没有我们使用嵌入法特征选择过后的96%高，有没有什么办法能够提高模型的表现呢?

7. **突发奇想，特征数量已经不足原来的3%，换模型怎么样?**

在之前的建模过程中，因为计算量太大，所以我们一直使用随机森林，但事实上，我们知道KNN的效果比随机森林更好，KNN在未调参的状况下已经达到96%的准确率，而随机森林在未调参前只能达到93%，这是模型本身的限制带来的，这个数据使用KNN效果就是会更好。现在我们的特征数量已经降到不足原来的3%，可以使用KNN了吗?

```
from sklearn.neighbors import KNeighborsClassifier as KNN
cross_val_score(KNN(),X_dr,y,cv=5).mean()
```

8. **KNN的k值学习曲线**

```
#======【TIME WARNING:6mins 30s 】======#

score = []
for i in range(10):
    X_dr = PCA(23).fit_transform(X)
    once = cross_val_score(KNN(i+1),X_dr,y,cv=5).mean()
    score.append(once)
plt.figure(figsize=[20,5])
plt.plot(range(10),score)
plt.show()
```

9. **定下超参数后，模型效果如何，模型运行时间如何?**

```
cross_val_score(KNN(4),X_dr,y,cv=5).mean()


#=======【TIME WARNING: 3mins】======#
%%timeit
cross_val_score(KNN(4),X_dr,y,cv=5).mean()
```

可以发现，原本785列的特征被我们缩减到23列之后，用KNN跑出了目前位置这个数据集上最好的结果。再进行更细致的调整，我们也许可以将KNN的效果调整到98%以上。PCA为我们提供了无限的可能，终于不用再因为数据量太庞大而被迫选择更加复杂的模型了！

特征工程非常深奥，虽然我们日常可能用到不多，但其实它非常美妙。若大家感兴趣，也可以自己去网上搜一搜，多读多看多试多想，技术逐渐会成为你的囊中之物。