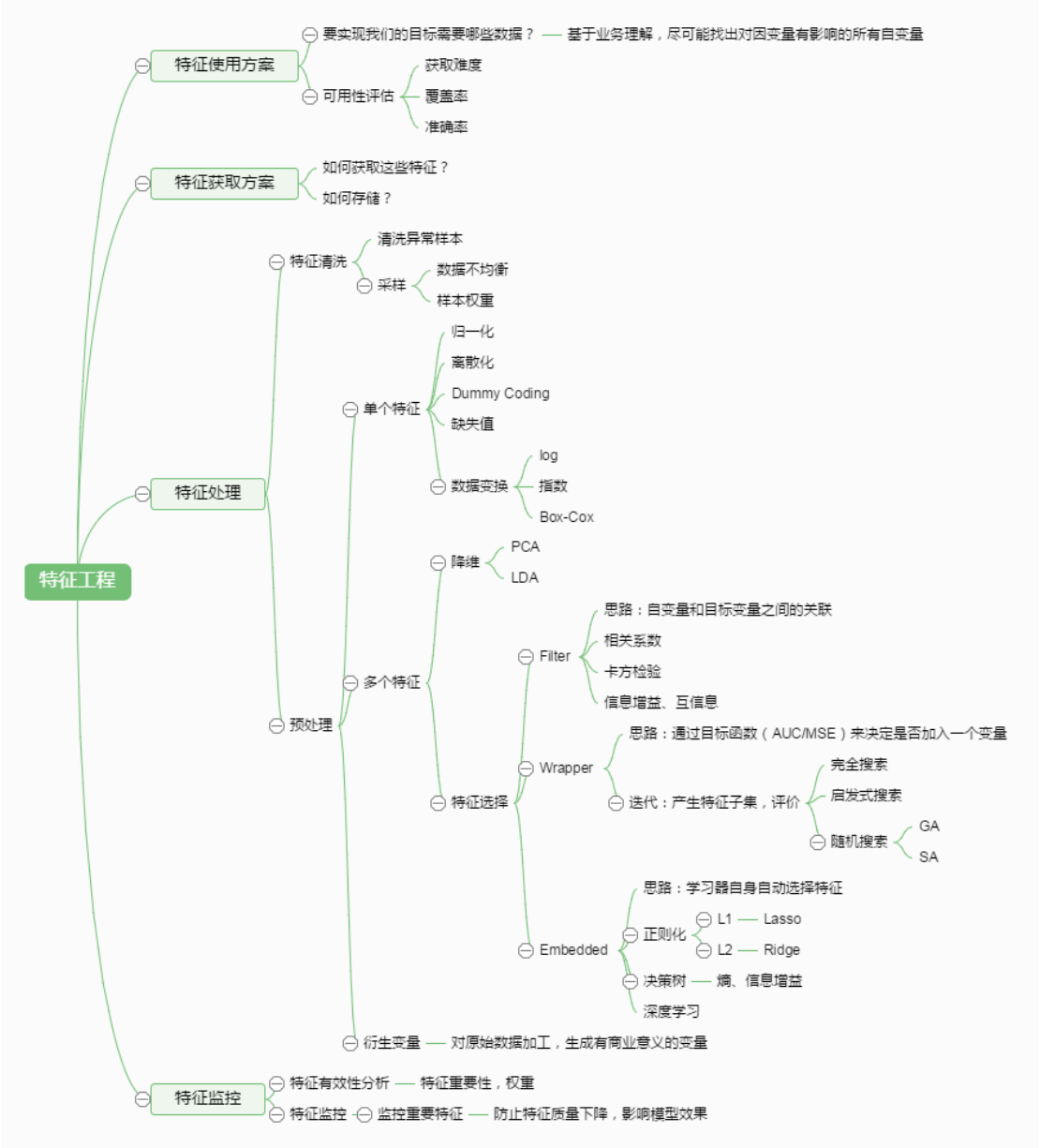
目录

前言

特征是数据中抽取出来的对结果预测有用的信息，可以是文本或者数据。特征工程是使用专业背景知识和技巧处理数据，使得特征能在机器学习算法上发挥更好的作用的过程。过程包含了特征提取、特征构建、特征选择等模块。

特征工程的目的是筛选出更好的特征，获取更好的训练数据。因为好的特征具有更强的灵活性，可以用简单的模型做训练，更可以得到优秀的结果。“工欲善其事，必先利其器”，特征工程可以理解为利其器的过程。互联网公司里大部分复杂的模型都是极少数的数据科学家在做，大多数工程师们做的事情基本是在数据仓库里搬砖，不断地数据清洗，再一个是分析业务不断地找特征。 例如，某广告部门的数据挖掘工程师，2 周内可以完成一次特征迭代，一个月左右可以完成模型的小优化，来提升 auc。



1. 数据采集 / 清洗 / 采样

1.1 数据采集

数据采集应考虑以下几方面问题

（1）数据采集前需要明确采集哪些数据，基于对业务规则的理解，尽可能多的找出对因变量有影响的所有自变量数据。

举例 1：我现在要预测用户对商品的下单情况，或者我要给用户做商品推荐，那我需要采集什么信息呢？

-店家：店铺的评分、店铺类别……

-商品：商品评分、购买人数、颜色、材质、领子形状……

-用户：历史信息（购买商品的最低价最高价）、消费能力、商品停留时间……

（2）数据我们能够采集到吗？线上实时计算的时候获取是否快捷？

（3）数据可用性评估 在获取数据的过程中，首先需要考虑的是这个数据获取的成本； 获取得到的数据，在使用之前，需要考虑一下这个数据是否覆盖了所有情况以及这个数 据的可信度情况。

（4）数据源：

- 用户行为日志数据：记录的用户在系统上所有操作所留下来的日志行为数据 业务数据：商品/物品的信息、用户/会员的信息.....
- 第三方数据：爬虫数据、购买的数据、合作方的数据....

（5）数据储存：

一般情况下，用于后期模型创建的数据都是存在在本地磁盘、关系型数据库或者 一些相关的分布式数据存储平台的。本地磁盘 MySQL Oracle HBase HDFS Hive

1.2 数据清洗

数据清洗也是很重要的一步，机器学习算法大多数时候就是一个加工机器，至于最后的产品如何，取决于原材料的好坏，肮脏数据直接影响到数据分析结果的准确率。数据清洗工作占据数据分析工作整个过程的七成以上的时间，所以说我们要格外的重视数据清洗工作。

为什么要数据清洗？现实世界的数据是“脏的 (Dirty Data)”

数据缺失：记录为空&属性为空

数据重复：完全重复&不完全重复

数据错误：异常值&不一致

数据不可用：数据正确，但不可用

如何预防脏数据出现？

制定数据标准：统一多数据源的属性值编码；尽可能赋予属性名和属性值明确的含义

优化系统设计：关键属性尽可能采用选项方式，而不是手动填写；重要属性出现在醒目的位置，采用必填选项；异常值要给出修改提示。

墨菲定律：凡事只要有可能出错,那就一定会出错。

处理数据缺失

引起缺失值的原因：设备异常；在输入时，有些数据得不到重视而没有被输入。

缺失值要经过推断而补上：忽略该记录；使用默认值；使用属性平均值；使用同类样本平均值；预测最可能的值

处理数据重复

引起重复值的原因：整合多个数据源的数据；在输入时，有些数据重复输入。

重复值经过推断进行合并：删除完全重复的记录；合并不同的表时，增加部分冗余属性（例如时间）。

处理数据错误：不一致

引起不一致的原因：数据录入者习惯不好；数据没有统一的标准。

数据不一致通过匹配进行修改：制定清洗规则表，进行匹配；通过统计描述，找到异常值

处理数据错误：数据噪声

引起数据噪音的原因：数据记录的过程中存在偏差；设备测量数据的过程中存在偏差

数据噪音可以通过：分箱算法、聚类算法、回归算法。

数据集成：将多个数据源中的数据整合到一个一致的存储中，包括模式匹配、数据冗余、数据值冲突。

1.3 数据采样

采集、清洗过数据以后，正负样本是不均衡的，要进行数据采样。采样的方法有随机采样和分层抽样。但是随机采样会有隐患，因为可能某次随机采样得到的数据很不均匀，更多的是根据特征采用分层抽样。

正负样本不平衡处理办法：

正样本 >> 负样本，且量都挺大 => undersampling（下采样/欠采样）：从多数类中随机抽取样本从而减少多数类别样本数据，使数据达到平衡的方式。

正样本 >> 负样本，量不大 =>

1）采集更多的数据

2）oversampling(上采样/过采样，比如图像识别中的镜像和旋转）。采用数据合成的方式生成更多的少数类样本，该方式在小数据集场景下具有比较成功的案例。常见算法是 SMOTE 算法，该算法利用基于 KNN 算法的“插值”来为少数类合成新的样本。

3）修改损失函数/loss function（设置样本权重）：设置损失函数的权重，使得少数类别数据判断错误的损失大于多数类别数据判断错误的损失，即当我们的少数类别数据预测错误的时候，会产生一个比较大的损失值，从而导致模型参数往让少数类别数据预测准确的方向偏。可以通过 scikitlearn 中的 class_weight 参数来设置权重。

不平衡程度相同（即正负样本比例类似）的两个问题，解决的难易程度也可能不同，因为问题难易程度还取决于我们所拥有数据有多大。比如在预测微博互动数的问题中，虽然数据不平衡，但每个档位的数据量都很大——最少的类别也有几万个样本，这样的问题通常比较容易解决；而在癌症诊断的场景中，因为患癌症的人本来就很少，所以数据不但不平衡，样本数还非常少，这样的问题就非常棘手。综上，可以把问题根据难度从小到大排个序：大数据+分布均衡<大数据+分布不均衡<小数据+数据均衡<小数据+数据不均衡。说明:对于小数据集,机器学习的方法是比较棘手的。对于需要解决的问题，拿到数据后，首先统计可用训练数据有多大，然后再观察数据分布情况。经验表明，训练数据中每个类别有 5000 个以上样本，其实也要相对于特征而言，来判断样本数目是不是足够，数据量是足够的，正负样本差一个数量级以内是可以接受的，不太需要考虑数据不平衡问题（完全是经验，没有理论依据，仅供参考）。

解决这一问题的基本思路是让正负样本在训练过程中拥有相同的话语权，比如利用采样与加权等方法。为了方便起见，我们把数据集中样本较多的那一类称为“大众类”，样本较少的那一类称为“小众类”。

采样方法是通过对训练集进行处理使其从不平衡的数据集变成平衡的数据集，在大部分情况下会对最终的结果带来提升。

采样分为上采样（Oversampling）和下采样（Undersampling），上采样是把小众类复制多份，下采样是从大众类中剔除一些样本，或者说只从大众类中选取部分样本。

随机采样最大的优点是简单，但缺点也很明显。上采样后的数据集中会反复出现一些样本，训练出来的模型会有一定的过拟合；而下采样的缺点显而易见，那就是最终的训练集丢失了数据，模型只学到了总体模式的一部分。

上采样会把小众样本复制多份，一个点会在高维空间中反复出现，这会导致一个问题，那就是运气好就能分对很多点，否则分错很多点。为了解决这一问题，可以在每次生成新数据点时加入轻微的随机扰动，经验表明这种做法非常有效。

因为下采样会丢失信息，如何减少信息的损失呢？第一种方法叫做 EasyEnsemble，利用模型融合的方法（Ensemble）：多次下采样（放回采样，这样产生的训练集才相互独立）产生多个不同的训练集，进而训练多个不同的分类器，通过组合多个分类器的结果得到最终的结果。第二种方法叫做 BalanceCascade，利用增量训练的思想（Boosting）：先通过一次下采样产生训练集，训练一个分类器，对于那些分类正确的大众样本不放回，然后对这个更小的大众样本下采样产生训练集，训练第二个分类器，以此类推，最终组合所有分类器的结果得到最终结果。第三种方法是利用 KNN 试图挑选那些最具代表性的大众样本，叫做 NearMiss，这类方法计算量很大，感兴趣的可以参考“Learning from Imbalanced Data”这篇综述的 3.2.1 节。

2. 特征处理

2.1 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”。譬如梯度和矩阵为核心的算法中，譬如逻辑回归，支持向量机，神经网络，无量纲化可以加快求解速度；而在距离类模型，譬如 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，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。

preprocessing.MinMaxScaler

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。归一化之后的数据服从正态分布，公式如下：

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

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]python 中会有一些函数比

如 MinMaxScaler()将幅度调整到 [0,1] 区间。

由于最大值与最小值可能是动态变化的，同时也非常容易受噪声(异常点、离群点)影响，因此一般适合小数据的场景。此外，该方法还有两点好处：

1）如果某属性/特征的方差很小，如身高：np.array([[1.70],[1.71],[1.72],[1.70],[1.73]])，实际 5 条数据在身高这个特征上是有差异的，但是却很微弱，这样不利于模型的学习，进行 min-max 归一化后为：array([[0.], [0.33333333], [0.66666667], [0.], [1.]])，相当于放大了差异；

2）维持稀疏矩阵中为 0 的条目。

```
from sklearn.preprocessing import MinMaxScaler
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
#不太熟悉 numpy 的小伙伴，能够判断 data 的结构吗？  #如果换成表是什么样子？
import pandas as pd pd.DataFrame(data)
#实现归一化
scaler = MinMaxScaler()
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

#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])
result = scaler.fit_transform(data)
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)
```

BONUS：使用 numpy 来实现归一化

```
import numpy as np
X = np.array([[-1, 2], [-0.5, 6], [0, 10], [1, 18]])
#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
```

preprocessing.StandardScaler

当数据(x)按均值(μ)中心化后，再按标准差(σ)缩放，数据就会服从为均值为 0，方差为 1 的正态分布（即标准正态分布），而这个过程，就叫做数据标准化(Standardization，又称 Z-score normalization)，公式如下：

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

```
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
```

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。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？

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。

无量纲化	功能	中心化	缩放	详解
.StandardScaler	标准化	均值	方差	通过减掉均值并将数据缩放到单位方差来标准化特征，标准化完毕后的特征服从标准正态分布，即方差为1，均值为0
.MinMaxScaler	归一化	最小值	极差	通过最大值最小值将每个特征缩放到给定范围，默认[0,1]
.MaxAbsScaler	缩放	N/A	最大值	通过让每一个特征里的数据，除以该特征中绝对值最大的数值的绝对值，将数据压缩到[-1,1]之间，这种做法并没有中心化数据，因此不会破坏数据的稀疏性。数据的稀疏性是指，数据中包含0的比例，0越多，数据越稀疏。
.RobustScaler	无量纲化	中位数	四分位 数范围	使用可以处理异常值，对异常值不敏感的统计量来缩放数据。 这个缩放器删除中位数并根据百分位数范围（IRQ:Interquartile Range）缩放数据。IQR是第一分位数（25%）和第三分维数（75%）之间的范围。数据集的标准化是通过去除均值，缩放到单位方差来完成，但是异常值通常会对样本的均值和方差造成负面影响，当异常值很多噪音很大时，用中位数和四分位数范围通常会产生更好的效果。
.Normalizer	无量纲化	N/A	sklearn中未明确，依范数原理应当是： l1：样本向量的长度/样本中每个元素绝对值的和 l2：样本向量的长度/样本中每个元素的欧氏距离	将样本独立缩放到到单位范数。每个至少带一个非0值的样本都回被独立缩放，使得整个样本（整个向量）的的长度都为l1范数或l2范数。这个类可以处理密集数组(numpy arrays)或scipy中的稀疏矩阵（scipy.sparse），如果你希望避免复制/转换过程中的负担，请使用CSR格式的矩阵。 将输入的数据缩放到单位范数是文本分类或聚类中的常见操作。例如，两个l2正则化后的TF-IDF向量的点积是向量的余弦相似度，并且是信息检索社区常用的向量空间模型的基本相似性度量。 使用参数norm来确定要正则化的范数方向，可以选择"l1","l2"以及"max"三种选项，默认l2范数。 这个评估器的fit接口什么也不做，但在管道中使用依然是很有用的。
.Power Transformer	非线性 无量纲化	N/A	N/A	应用特征功率变换使数据更接近正态分布。 功率变换是一系列参数单调变换，用于使数据更像高斯。这对于建模与异方差性（非常数方差）或其他需要正态性的情况相关的问题非常有用。要求输入的数据严格为正，power_transform()通过最大似然估计来稳定方差和并确定最小化偏度的最佳参数。 默认情况下，标准化应用于转换后的数据。
.Quantile Transformer	非线性 无量纲化	N/A	N/A	使用百分位数转换特征，通过缩小边缘异常值和非异常值之间的距离来提供特征的非线性变换。可以使用参数output_distribution = "normal"来将数据映射到标准正态分布。
.KernelCenterer	中心化	均值	N/A	将核矩阵中心化。设K(x，z)是由phi(x)^T phi(z)定义的核，其中phi是将x映射到希尔伯特空间的函数。 KernelCenterer在不明确计算phi(x)的情况下让数据中心化为0均值。 它相当于使用sklearn.preprocessing.StandardScaler(with_std = False)来将phi(x)中心化。

2.2 缺失值

sklearn. impute. SimpleImputer

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

参数	含义&输入
missing_values	数据中的缺失值长什么样，默认空值 np.nan
strategy	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()
#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
imp_mean = imp_mean.fit_transform(Age) #fit_transform 一步完成调取结果
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)
```

BONUS：用 Pandas 和 Numpy 进行填补其实更加简单

```
import pandas as pd
data = pd.read_csv(r"C:\work\learnbetter\micro-class\week 3 Preprocessing\Narrativedata.csv",index_col=0)
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
```

2.3 定量特征离散化

sklearn.preproccessing.Binarizer：对定量特征二值化。

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
```

KBinsDiscretizer：将连续型变量划分为分类变量类别，能够将连续型变量排序后按顺序分箱后编码。

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。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)
strategy	"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()
```

2.4 定性特征哑编码

在机器学习中，大多数算法，譬如逻辑回归，支持向量机SVM，k近邻算法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（其实手写决策树和普斯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
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
data.iloc[:,-1] = label #让标签等于我们运行出来的结果
data.head()
```

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
#接口 categories_对应 LabelEncoder 的接口 classes_，一模一样的功能
data_ = data.copy()
data_.head()
OrdinalEncoder().fit(data_.iloc[:,1:-1]).categories_
data_.iloc[:,1:-1] = OrdinalEncoder().fit_transform(data_.iloc[:,1:-1])
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。在舱门 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） 舱门（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，这三个数字相互不等，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，性别和舱门，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。

```
data.head()
from sklearn.preprocessing import OneHotEncoder
X = data.iloc[:,1:-1]
enc = OneHotEncoder(categories='auto').fit(X)
result = enc.transform(X).toarray()
result
#依然可以直接一步到位，但为了给大家展示模型属性，所以还是写成了三步
OneHotEncoder(categories='auto').fit_transform(X).toarray()
#依然可以还原
pd.DataFrame(enc.inverse_transform(result))
enc.get_feature_names()
result
result.shape
#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
newdata = pd.concat([data,pd.DataFrame(result)],axis=1)
newdata.head()
newdata.drop(["Sex","Embarked"],axis=1,inplace=True)
newdata.columns =
["Age","Survived","Female","Male","Embarked_C","Embarked_Q","Embarked_S"]
newdata.head()
```

特征可以做哑变量，标签可以做哑变量吗？可以，类 sklearn.preprocessing.LabelBinarizer 可以对做哑变量，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。

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

2.5 文本特征向量化

机器学习的模型算法均要求输入的数据必须是数值型的，所以对于文本类型的特征属性，需要进行文本数据转换，也就是需要将文本数据转换为数值型数据。常用方式如下：词袋模型(BOW/TF)、TF-IDF(Term frequency-inverse document frequency)模型、HashTF 模型、Word2Vec 模型(主要用于单词的相似性考量)。

2.5.1 词袋模型

词汇表：单词构成的集合，集合自然每个元素都只有一个，也即词汇表中的每个单词都只有一个。

词袋模型：假定对于一个文本，忽略其词序和语法，句法，将其仅仅看做是一个词集合，或者说是词的一个组合，文本中每个词的出现都是独立的，统计其出现的次数。

词袋是在词汇表的基础上增加了频率的维度，词汇表只关注有和没有，词袋还要关注有几个。

例如，有语料库（corpus）如下：

John likes to watch movies. Mary likes movies too.

John also likes to watch football games.

把上述语料中的词汇整理出来并进行排序，假设我们的词汇表排序结果如下：

{"John": 1, "likes": 2, "to": 3, "watch": 4, "movies": 5, "also":6, "football": 7, "games": 8, "Mary": 9, "too": 10}

得出如下**词向量**表示：

John: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

likes: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

……

文档向量的表示方法是直接把各词的词向量表示加和，那么原来的两句话的向量表示如下：

[1, 2, 1, 1, 2, 0, 0, 0, 1, 1]

[1, 1, 1, 1, 0, 1, 1, 1, 0, 0]

文档向量中，列的值表示词在文档中出现的次数。

构建词袋模型的步骤：

（1）文本分词：把每个文档中的文本进行分词

（2）构建词汇表：把文本分词得到的单词构建为一个词汇表，包含文本语料库中的所有单词，并对单词进行编号，假设词汇表有 n 个单词，单词编号从 0 开始，到 n-1 结束，可以把单词编号看作是单词的索引，通过单词编号可以唯一定位到该单词。

（3）词向量表示：每个单词都表示为一个 n 列的向量，在单词编号（词汇索引）位置上的列值为 1，其他列的值为 0

（4）统计频次：统计每个文档中每个单词出现的频次。

【例子】有如下包含三个文档（doc）的语料库，每个文档是一行文本：

Doc 1: I love dogs.

Doc 2: I hate dogs and knitting.

Doc 3: Knitting is my hobby and passion.

根据语料库，对文本进行分词，创建词汇表。根据词汇表计算每个文档中的单词出现的次数，这个矩阵叫做文档-词矩阵（DTM，Document-Term Matrix）。

	I	love	dogs	hate	and	knitting	is	my	hobby	passion
Doc 1	1	1	1							
Doc 2	1		1	1	1	1				
Doc 3					1	1	1	2	1	1

这个矩阵使用的是单个词，也可以使用两个或多个词的组合，叫做 bi-gram 模型或 tri-gram 模型，统称 n-gram 模型。

使用 sklearn 构建词袋模型：

```
CountVectorizer(input='content', encoding='utf-8', decode_error='strict', strip_accents=None,
                lowercase=True,      preprocessor=None, tokenizer=None, stop_words=None,
                token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer='word',
                max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False,
                dtype=<class 'numpy.int64'>)
```

常用参数注释：

- input：默认值是 content，表示输入的是顺序的字符文本
- decode_error：默认为 strict，遇到不能解码的字符将报 UnicodeDecodeError 错误，设为 ignore 将会忽略解码错误
- lowercase：默认值是 True，在分词（Tokenize）之前把文本中的所有字符转换为小写。
- preprocessor：预处理器，在分词之前对文本进行预处理，默认值是 None
- tokenizer：分词器，把文本字符串拆分成各个单词（token），默认值是 None
- analyzer：用于预处理和分词，可设置为 string 类型，如'word'，'char'，'char_wb'，默认值是 word
- stop_words：停用词表，如果值是 english，使用内置的英语停用词列表；如果是一个列表，那么使用该列表作为停用词，设为 None 且 max_df∈[0.7，1.0)将自动根据当前的语料库建立停用词表
- ngram_range：tuple（min_n，max_n），表示 ngram 模型的范围
- max_df：可以设置为范围在[0.0 1.0]的浮点数，也可以设置为没有范围限制的整数，默认为 1.0。这个参数的作用是作为一个阈值，当构造语料库的词汇表时，如果某个词的 document frequence 大于 max_df，这个词不会被当作关键词。如果这个参数是 float，则表示词出现的次数与语料库文档数的百分比，如果是 int，则表示词出现的次数。如果参数中已经给定了 vocabulary，则这个参数无效

- min_df：类似于 max_df，不同之处在于如果某个词的 document frequence 小于 min_df，则这个词不会被当作关键词
- max_features：对所有关键词的 term frequency 进行降序排序，只取前 max_features 个作为关键词集
- vocabulary：默认为 None，自动从输入文档中构建关键词集，也可以是一个字典或可迭代对象。
- binary：默认为 False，一个关键词在一篇文档中可能出现 n 次；如果 binary=True，非零的 n 将全部置为 1，这对需要布尔值输入的离散概率模型的有用的
- **dtype**：用于设置 fit_transform() 或 transform()函数返回的矩阵元素的数据类型

模型的属性和方法：

- vocabulary_：词汇表，字典类型
- get_feature_names()：所有文本的词汇，列表型
- stop_words_：停用词列表

模型的主要方法：

- fit(raw_document)：拟合模型，对文本分词，并构建词汇表等
- transform(raw_documents)：把文档转换为文档-词矩阵
- fit_transform(raw_documents)：拟合文档，并返回该文档的文档-词矩阵

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...         'This is the first document.',
...         'This document is the second document.',
...         'And this is the third one.',
...         'Is this the first document?',
... ]
>>> vectorizer = CountVectorizer()
>>> dt = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> print(dt.toarray())
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

2.5.2 TF-IDF 模型

TF-IDF 是一种统计方法，用以评估一个词对一个文件集或一个语料库中的一份文件的重要程度。当一个词在特定的文档中经常出现，而在其他文档中出现的频次很低，那么给予该词较高的权重；当一次词在多个文档中出现的频次都很高，那么给予该词较低的权重。如果一次单词在特定的文档中出现的频次很高，而在其他文档中出现的频次很低，那么这个单词很可能是该文档独有的词，能够很好地描述该文档。

TF（ Term Frequency）是词频，表示每个单词在文档中的数量（频数），TF 依赖于 BoW 模型的输出。

IDF（Inverse Document Frequency）是逆文档频率，代表一个单词的普遍程度，当一个词越普遍（即

有大量文档包含这个词）时，其 IDF 值越低；反之，则 IDF 值越高。IDF 是包含该单词的文档数量和文档总数的对数缩放比例。

TF-IDF=TF * IDF

TF：Term Frequency——某个词在该文件中出现的频率

$$TF(t) = \frac{\text{词}t\text{在当前文档出现的频次}}{\text{文档中全部词出现的频次}}$$

IDF：逆序文档频度（inverse document frequency）

$$IDF(t) = ln(\frac{\text{总文档数量}}{\text{包含}t\text{的文档数量}})$$

也就是 IDF 是一个以（总文档数量/包含 t 的文档数量）为自变量的 ln 函数，lb 函数是单调增函数，且”总文档数量/包含 t 的文档数量>1”，故 ln 函数是一个大于 0 且单调增的函数，也就是包含 t 文档的数量越少，IDF 值越大。

如果词条 t 在当前文档出现次数多，且包含词条 t 的文档越少，也就是在语料库中出现的次数越少，则 IDF 值越大，说明词条越重要，具有很好的类别区分能力。

在文档中具有高 tf-idf 的单词，大多数情况下，只发生在给定的文档中，并且在其他文档中不存在，所以这些词是该文档的特征词汇。

	I	love	dogs	hate	and	knitting	is	my	hobby	passion
Doc 1	0.18	**0.48**	0.18							
Doc 2	0.18		0.18	**0.48**	0.18	0.18				
Doc 3					0.18	0.18	**0.48**	**0.95**	**0.48**	**0.48**

使用 TfidfVectorizer()函数构建 TF-IDF 模型

```
TfidfVectorizer(input='content', encoding='utf-8', decode_error='strict', strip_accents=None,
                lowercase=True,    preprocessor=None, tokenizer=None, stop_words=None,
                token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer='word',
                max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False,
                dtype=dtype=<class 'numpy.float64'>,
                norm='l2', use_idf=True, smooth_idf=True, sublinear_tf=False)
```

大部分参数和 CountVectorizer 相同，TfidfVectorizer 独有的参数注释：

- norm=’l2’：每个输出行具备单位规范，当引用'l2'范式时，所有向量元素的平方和为 1；当应用 l2 范数时，两个向量之间的余弦相似度是它们的点积。 *'l1'：向量元素的绝对值之和为 1。
- use_idf=True：启用 IDF 来重新加权
- smooth_idf=True：平滑 idf 权重，向文档-词频矩阵的所有位置加 1，就像存在一个额外的文档，只包含词汇表中的每个术语一次，目的是为了防止零分裂。
- sublinear_tf=False：应用次线性 tf 缩放，默认值是 False。

举个例子，使用 TfidfVectorizer()函数构建以下四个句子的 TF-IDF 模型：

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = [
...     'This is the first document.',
...     'This document is the second document.',
...     'And this is the third one.',
...     'Is this the first document?',
... ]
>>> vectorizer = TfidfVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> print(vectorizer.get_feature_names())
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> print(X.toarray())
[[0.         0.46979139 0.58028582 0.38408524 0.         0.         0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762 0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.         0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.         0.38408524 0.         0.38408524]]
```

从原始文档列表（语料）中获取特征列表

```
>>> print(vectorizer.get_feature_names())
['adr', 'authorized', 'contact', 'device', 'distributor', 'hub', 'information', 'installer',
'interested', 'microsoft', 'partner', 'program', 'receive', 'reseller', 'sign', 'surface', 'updates', 'website']
```

词汇表中的词汇和其对应的索引

```
>>> items=vectorizer.vocabulary_.items()
>>> print(items)
dict_items([('information', 6), ('surface', 15), ('hub', 5), ('microsoft', 9), ('authorized', 1),
('device', 3), ('reseller', 13), ('adr', 0), ('interested', 8), ('contact', 2), ('distributor', 4),
('sign', 14), ('receive', 12), ('updates', 16), ('program', 11), ('partner', 10), ('website', 17), ('installer', 7)])
```

把 dict_items 结构转换为 Python 的字典结构，key 是索引，value 是词汇

```
>>> feature_dict = {v: k for k, v in vectorizer.vocabulary_.items()}
```

模型的 fit_transform()或 tranform()函数返回的是词-文档矩阵，在词-文档矩阵中列代表的是特征，行代表的原始文档的数量，列代表该文档包含的特征（即词汇），列值是特征的 TD-IDF 值，范围从 0-1。

```
>>> print(X.toarray())
[[0.         0.46979139 0.58028582 0.38408524 0.         0.         0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762 0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.         0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.         0.38408524 0.         0.38408524]]
```

2.5.3 Word2vec 模型

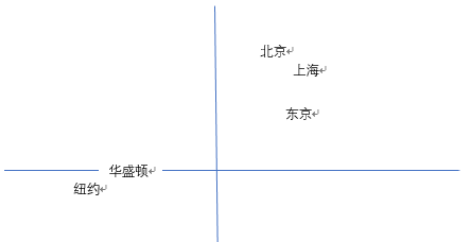
（词向量 word embeddings）作用就是将自然语言中的字词转为计算机可以理解的稠密向量（Dense

Vector)。在 word2vec 出现之前，自然语言处理经常把字词转为离散的单独的符号，也就是 One-Hot Encoder。

杭州 [0,0,0,0,0,0,0,1,0,..... , 0,0,0,0,0,0,0]
上海 [0,0,0,0,0,1,0,0,0,0,..... , 0,0,0,0,0,0,0]
宁波 [0,0,0,1,0,0,0,0,0,..... , 0,0,0,0,0,0,0]
北京 [0,0,0,0,0,0,0,0,0,..... , 1,0,0,0,0,0,0]

在语料库中，杭州、上海、宁波、北京各对应一个向量，向量中只有一个值为 1，其余都为 0。但是使用 One-Hot Encoder 有以下问题。一方面，城市编码是随机的，向量之间相互独立，看不出城市之间可能存在的关联关系。其次，向量维度的大小取决于语料库中字词的多少。如果将世界所有城市名称对应的向量合为一个矩阵的话，那这个矩阵过于稀疏，并且会造成维度灾难。

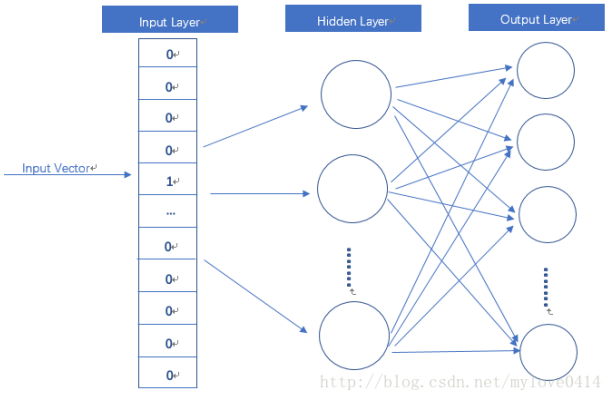
word2vec 可以将独热码转化为低维度的连续值，也就是稠密向量，并且其中意思相近的词将被映射到向量空间中相近的位置。



华盛顿和纽约聚集在一起，北京上海聚集在一起，且北京到上海的距离与华盛顿到纽约的距离相近。也就是说模型学习到了城市的地理位置，也学习到了城市地位的关系。

word2vec 模型解释：

该模型实际上是一个具有隐含层的神经网络，输入是 one-hot 编码的词汇向量表，隐含层没有激活函数，也就是线性单元，输出层维度和输入层维度相同，使用 Softmax 回归，我们要获取的稠密向量其实就是隐藏层的输出单元。



假如我们利用一个 2x6 的 one-hot 向量作为输入，中间层节点数为 3，这就相当于分别提取出了权值矩阵中的第一行和第二行，这就和所谓的字向量的查表的意义是一样的，这就是所谓的 Embedding 层，该层就是以 one-hot 为输入，中间层节点为字向量维数的全连接层，而这个全连接层的参数，就是一个“字向量表”

严格来讲，神经网络都是有监督的，而 Word2Vec 之类的模型，准确来说应该是“自监督”的，它事实上训练了一个语言模型，通过语言模型来获取词向量。所谓语言模型，就是通过前 n 个字预测下一个字的概率，就是一个多分类器而已，我们输入 one hot，然后连接一个全连接层，然后再连接若干个层，最后接一个 softmax 分类器，就可以得到语言模型了，然后将大批量文本输入训练就行了，最后得到第一个全连接层的参数，就是字、词向量表，当然，Word2Vec 还做了大量的简化，但是那都是在语言模型本身做的简化，它的第一层还是全连接层，全连接层的参数就是字、词向量表。

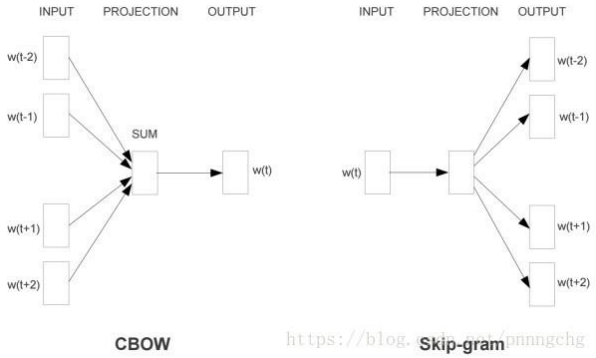
当然，由于标签数据一般不会很多，因此这样容易过拟合，因此一般先用大规模语料无监督训练字、

词向量，降低过拟合风险。注意，降低过拟合风险的原因是可以使用无标签语料预训练词向量出来（无标签语料可以很大，语料足够大就不会有过拟合风险），跟词向量无关，词向量就是一层待训练参数，有什么本事降低过拟合风险？

word2vec 的两种预测模型：

1、CBOW：主要用来从原始语句推测目标词汇

2、skip-gram：从目标词汇推测原始语境



Word2Vec 的思想和自编码器很相似。具体来说，我们用机器学习的方法训练神经网络，但是最终我们关心的不是使用这个神经网络计算输出结果，我们想要得到的是神经网络隐藏层的参数，这个权重矩阵就是我们需要的词向量 Word vector。换句话说，建模并不是我们的根本目的。

2.5.4 HashTF-IDF

不管是前面的词袋法还是 TF-IDF，都避免不了计算文档中单词的词频，当文档数量比较少、单词数量比较少的时候，我们的计算量不会太大，但是当这个数量上升到一定程度的时候，程序的计算效率就会降低下去，这个时候可以通过 HashTF 的形式来解决该问题。HashTF 的计算规则是：在计算过程中，不计算词频，而是计算单词进行 hash 后的 hash 值对应的样本的数量(有的模型中可能存在正则化操作)；

HashTF 的特点：运行速度快，但是无法获取高频词，有可能存在单词碰撞问题(hash 值一样)

3. 特征选择

特征选择，就是从多个特征中，挑选出一些对结果预测最有用的特征。因为原始的特征中可能会有冗余和噪声。特征选择和降维有什么区别呢？前者只踢掉原本特征里和结果预测关系不大的，后者做特征的计算组合构成新特征。

特征提取 (feature extraction)	特征创造 (feature creation)	特征选择 (feature selection)
从文字，图像，声音等其他非结构化数据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，是否是网红产品等等。	把现有特征进行组合，或互相计算，得到新的特征。比如说，我们有一列特征是速度，一列特征是距离，我们就可以通过让两列相处，创造新的特征：通过距离所花的时间。	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。

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。

在真正的数据应用领域，比如金融，医疗，电商，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。

3.1 过滤型

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。

3.1.1 方差过滤

1、VarianceThreshold

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
selector = VarianceThreshold()     #实例化，不填参数默认方差为 0
X_var0 = selector.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.var().values
np.median(X.var().values)
X_fsvar.shape
```

当特征是二分类时，特征的取值就是伯努利随机变量，这些变量的方差可以计算为：

Var[X]=p(1-p)，其中 X 是特征矩阵，p 是二分类特征中的一类在这个特征中所占的概率。

```
#若特征是伯努利随机变量，假设 p=0.8，即二分类特征中某种分类占到 80%以上的时候删除特征
X_bvar = VarianceThreshold(.8 * (1 - .8)).fit_transform(X) X_bvar.shape
```

2、方差过滤对模型的影响

我们这样做了以后，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()
cross_val_score(RFC(n_estimators=10,random_state=0),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；而最近邻算法就不同了，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（决策树参数random_state），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，随机森林是个非常强大的模型

3、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。

3.1.2 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。

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，模型的表现时提升的。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

#假设在这里我一直我需要 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

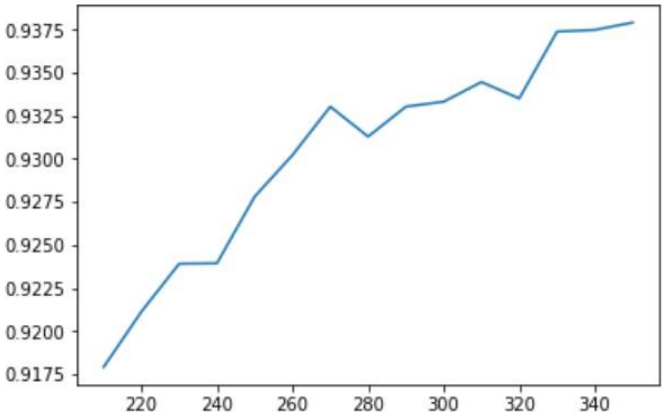
```
cross_val_score(RFC(n_estimators=10,random_state=0),X_fschi,y,cv=5).mean()
```

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。

2、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的差异，其检验的原假设是”两组数据是相互独立的”。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。

```
chivalue, pvalues_chi = chi2(X_fsvar,y)
chivalue
pvalues_chi
#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 = chivalue.shape[0] - (pvalues_chi > 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。

3、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，这些特征与标签时显著线性相关的，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, pvalues_f = f_classif(X_fsvar,y)
F
pvalues_f
k = F.shape[0] - (pvalues_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 0.01，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。

4、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。当然了，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。

3.1.3 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，先使用方差过滤，然后使用互信息法来捕捉相关性。

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

3.2 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。在我们之后的学习当中，每次讲解新的算法，我都会为大家提到这个算法中的特征工程是如何处理，包括具体到每个算法的嵌入法如何使用。在这堂课中，我们会为大家讲解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。

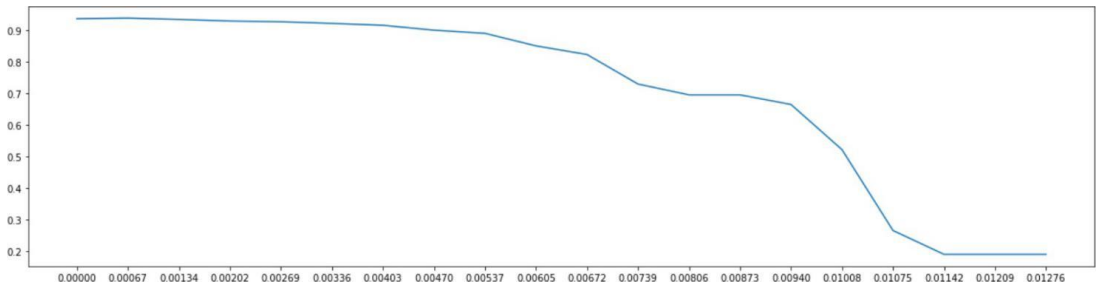
选读：使用惩罚项的模型的嵌入法

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()
```



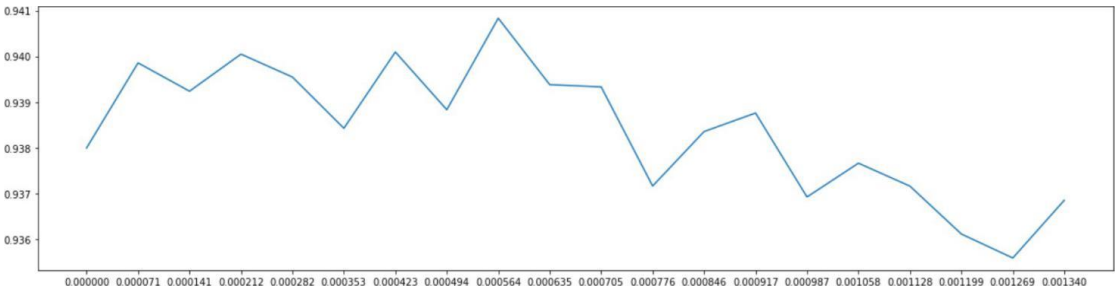
从图像上来看，随着阈值越来越高，模型的效果逐渐变差，被删除的特征越来越多，信息损失也逐渐变大。但是在 0.00134 之前，模型的效果都可以维持在 0.93 以上，因此我们可以从中挑选一个数值来验证一下模型的效果。

```
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：

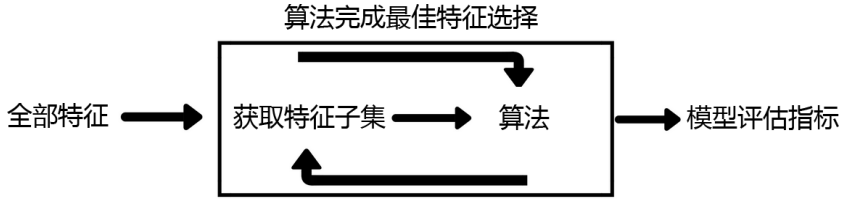
```
X_embedded = SelectFromModel(RFC_,threshold=0.000564).fit_transform(X,y)
X_embedded.shape

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。

3.3 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

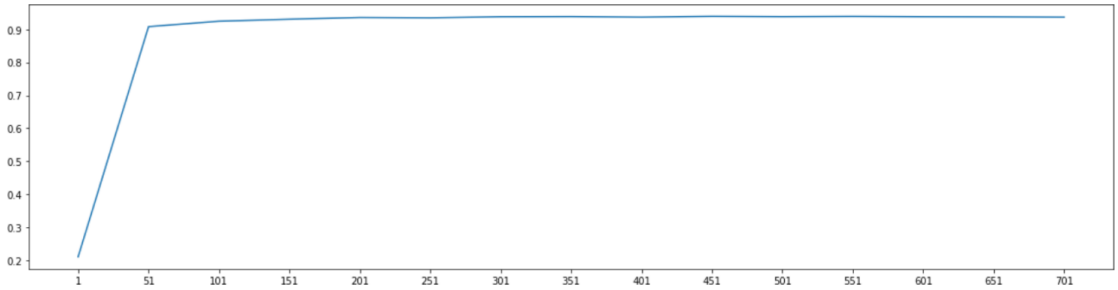
class sklearn.feature_selection.RFE (*estimator, n_features_to_select=None, step=1, verbose=0*)

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，以及.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。

3.4 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。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。

4 降维

在实际的机器学习项目中，特征选择/降维是必须进行的，因为在数据中存在以下几个方面的问题：

- 数据的多重共线性：特征属性之间存在着相互关联关系。多重共线性会导致解的空间不稳定，从而导致模型的泛化能力弱；
- 高纬空间样本具有稀疏性，导致模型比较难找到数据特征；
- 过多的变量会妨碍模型查找规律；
- 仅仅考虑单个变量对于目标属性的影响可能忽略变量之间的潜在关系。
- 数据集特征矩阵过大，导致计算量比较大，训练时间长。

通过特征选择/降维的目的是：

- 减少特征属性的个数
- 确保特征属性之间是相互独立的

以 k 近邻学习（kNN）为例简述下，不降维的情况下可能带来的问题。

给定测试样本，基于某种距离度量找出训练集中与其最接近的 k 个训练样本，然后基于这 k 个邻居的信息来进行预测。分类任务可使用投票法（或加权投票）；回归任务可使用平均法（或加权平均）。特点：没有显示的训练过程，是“懒惰学习”，k 值和距离度量方式对预测结果影响较大。

使用 kNN 方法需保证样本“密采样”，即保证近邻的存在性。

在高维空间中，样本属性较多，要保证“密采样”，需要的样本数目会很多。如若测试样本 x 在δ邻域范围内存在邻近点，当δ=0.01，样本 x 的维度为 d 时，则要满足“密采样”需要的样本数目为 100^d。

“维数灾难”问题：在高维空间中出现样本稀疏、距离计算困难等问题。

缓解维数灾难的两个主流方法：特征选择和降维。

常见的降维方法包括:低维嵌入、PCA、核化线性降维、流形学习、度量学习。一般情况会先使用线性的降维方法再使用非线性的降维方法，通过结果去判断哪种方法比较合适。

4.1 主成分分析法(PCA)

使用 sklearn.decomposition 库的 PCA 类选择特征的代码如下：

```
from sklearn.decomposition import PCA
# n_components is the dimensionality after dimension reduction.
PCA(n_components=2).fit_transform(iris.data)
```

4.2 线性判别分析(LDA)

使用 sklearn.lda 库的 LDA 类选择特征的代码如下：

```
from sklearn.lda import LDA
# n_components is the dimensionality after dimension reduction.
LDA(n_components=2).fit_transform(iris.data, iris.target)
```