数据挖掘通用框架的搭建方法。

 估计器（Estimator）：用于分类、聚类和回归分析。

 转换器（Transformer）：用于数据预处理和数据转换。

 流水线（Pipeline）：组合数据挖掘流程，便于再次使用。

scikit-learn 估计器

为帮助用户实现大量分类算法， scikit-learn把相关功能封装成所谓的估计器。估计器用

于分类任务，它主要包括以下两个函数。

 fit()：训练算法，设置内部参数。该函数接收训练集及其类别两个参数。

 predict()：参数为测试集。预测测试集类别，并返回一个包含测试集各条数据类别的数组。

用fit方法在训练集上完成模型的创建，用predict方法在测试集上评估效果。

大多数scikit-learn估计器接收和输出的数据格式均为numpy数组。当前pandas和scikit-learn并没有进行整合，但是借助NumPy数组，它们配合地很好。

y\_true = dataset["HomeWin"].values

上面的y\_true数组保存的是类别数据， scikit-learn可直接读取该数组。

转换器

接 调 用fit\_transform()函数，即可完成训练和转换。

X\_transformed = MinMaxScaler().fit\_transform(X)

流水线

流水线的输入为一连串的数据挖掘步骤，其中最后一步必须是估计器，前几步是转换器。输入的数据集经过转换器的处理后，输出的结果作为下一步的输入。最后，用位于流水线最后一步的估计器对数据进行分类。我们流水线分为两大步。

(1) 用MinMaxScaler将特征取值范围规范到0~1。

(2) 指定KNeighborsClassifier分类器。

每一步都用元组（‘名称’，步骤）来表示。现在来创建流水线。

scaling\_pipeline = Pipeline([('scale', MinMaxScaler()),

('predict', KNeighborsClassifier())])

流水线的核心是元素为元组的列表。第一个元组规范特征取值范围，第二个元组实现预测功能。我们把第一步叫作规范特征取值（scale），第二步叫作预测（predict），也可以用其他名字。元组的第二部分是实际的转换器对象或估计器对象。

流水线写好后，运行它很简单。使用先前用到的交叉检验代码看一下实际效果。

scores = cross\_val\_score(scaling\_pipeline, X\_broken, y,

scoring='accuracy')

print("The pipeline scored an average accuracy for is {0:.1f}%".

format(np.mean(transformed\_scores) \* 100))

运行结果跟之前一样（82.3%），表明我们这次用到的步骤跟之前相同。

后续章节会使用更高级的测试方法，而设置流水线就很有必要，因为它能确保代码的复杂程度不至于超出掌控范围

决策树

scikit-learn库实现了分类回归树（Classification and Regression Trees， CART）算法并将其作为生成决策树的默认算法，它支持连续型特征和类别型特征。

scikit-learn库实现的决策树算法给出了退出方法，使用下面这两个选项就可以达到目的。  
θ min\_samples\_split：指定创建一个新节点至少需要的个体数量。  
θ min\_samples\_leaf：指定为了保留节点，每个节点至少应该包含的个体数量。（剪枝用）  
第一个参数控制着决策节点的创建，第二个参数决定着决策节点能否被保留。  
决策树的另一个参数是创建决策的标准，常用的有以下两个。  
θ 基尼不纯度（Gini impurity）：用于衡量决策节点错误预测新个体类别的比例。  
θ 信息增益（Information gain）：用信息论中的熵来表示决策节点提供多少新信息。

随机森林

一棵决策树可以学到很复杂的规则。然而，很可能会导致过拟合问题——学到的规则只适用于训练集。解决方法之一就是调整决策树算法，限制它所学到的规则的数量。例如，把决策树的深度限制在三层，只让它学习从全局角度拆分数据集的最佳规则，不让它学习适用面很窄的特定规则，这些规则会将数据集进一步拆分为更加细致的群组。使用这种折中方案得到的决策树泛化能力强，但整体表现稍弱。

为了弥补上述方法的不足，我们可以创建多棵决策树，用它们分别进行预测，再根据少数服从多数的原则从多个预测结果中选择最终预测结果。这正是随机森林的工作原理。

但上述过程有两个问题。一是创建的多棵决策树在很大程度上是相同的——每次使用相同的输入，将得到相同的输出。我们只有一个训练集，如果尝试创建多棵决策树，它们的输入就可能相同（因此输出也相同）。解决方法是每次随机从数据集中选取一部分数据用作训练集。这个过程叫作装袋（bagging）。

第二个问题是用于前几个决策节点的特征非常突出。即使我们随机选取部分数据用作训练集，创建的决策树相似性仍旧很大。解决方法是，随机选取部分特征作为决策依据。然后，使用随机从数据集中选取的数据和（几乎是）随机选取的特征，创建多棵决策树。这就是随机森林，虽然看上去不是那么直观，但这种算法在很多数据集上效果很好。

随机森林算法的参数

scikit-learn库中的RandomForestClassifier就是对随机森林算法的实现，它提供了

一系列参数。因为它使用了DecisionTreeClassifier的大量实例，所以它俩的很多参数是一致的，比如决策标准（基尼不纯度/信息增益）、 max\_features和min\_samples\_split。

当然，集成过程还引入了一些新参数。

 n\_estimators：用来指定创建决策树的数量。该值越高，所花时间越长，正确率（可能）也越高。

 oob\_score：如果设置为真，测试时将不使用训练模型时用过的数据。

 n\_jobs：采用并行计算方法训练决策树时所用到的内核数量。

scikit-learn库提供了用于并行计算的Joblib库。 n\_jobs指定所用的内核数。默认使用1个内核——如果CPU是多核的，可以多用几个，或者将其设置为1，开动全部马力。

scikit-learn库实现的随机森林算法使用估计器接口，用交叉检验方法调用它即可，代码跟之前大同小异。

from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier(random\_state=14)

scores = cross\_val\_score(clf, X\_teams, y\_true, scoring='accuracy')

print("Accuracy: {0:.1f}%".format(np.mean(scores) \* 100))

可以使用GridSearchCV类搜索最佳参数，代码如下：

parameter\_space = {

"max\_features": [2, 10, 'auto'],

"n\_estimators": [100,],

"criterion": ["gini", "entropy"],

"min\_samples\_leaf": [2, 4, 6],

}

clf = RandomForestClassifier(random\_state=14)

grid = GridSearchCV(clf, parameter\_space)

grid.fit(X\_all, y\_true)

print("Accuracy: {0:.1f}%".format(grid.best\_score\_ \* 100))

这次正确率提升较大，达到了64.2%！

输出用网格搜索找到的最佳模型，查看都使用了哪些参数。代码如下：

print(grid.best\_estimator\_)

# 特征工程

***类别型特征***二值化后就变成了数值型特征

***数值型特征***也可以通过离散化过程转换为类别型特征

特征可以是类别型的也可以是数值型的，看需要进行转换

1. 在开展数据挖掘工作前，有些基础性测试我们要做，比如确保特征值是不同的。如果特征值都相同，就跟没提供什么信息一样，挖掘就失去了意义。scikit-learn中的VarianceThreshold转换器可用来删除特征值的方差达不到最低标准的特征。

from sklearn.feature\_selection import VarianceThreshold

vt = VarianceThreshold()

Xt = vt.fit\_transform(X) # 会直接剔除不达最低方差的特征

输出每一列的方差。

print(vt.variances\_)

1. 选择最佳特征

特征很多的情况下，怎么选出最佳的几个，可有点难度。随着特征数量的增加，寻找子集的任务复杂度呈指数级增长。寻找最佳特征组合的时间复杂度同样是指数级增长的。其中一个变通方法是不要找表现好的子集，而只是去找表现好的单个特征（单变量），依据 是它们各自所能达到的精确度。分类任务通常是这么做的，我们一般只要测量变量和目标类别之间的某种相关性就行。

scikit-learn提供了几个用于选择单变量特征的转换器，其中SelectKBest返回k个最佳特征， SelectPercentile返回表现最佳的前r%个特征。这两个转换器都提供计算特征表现的一系列方法。单个特征和某一类别之间相关性的计算方法有很多。最常用的有卡方检验（χ2）。其他方法还有互信息和信息熵。

使用SelectKBest转换器类，用卡方函数打分，初始化转换器。

from sklearn.feature\_selection import SelectKBest

from sklearn.feature\_selection import chi2

transformer = SelectKBest(score\_func=chi2, k=3)

Xt\_chi2 = transformer.fit\_transform(X, y) # 调用fit\_transform方法，对相同的数据集进行预处理和转换。X为特征，y为类别,最后选出前k个分数最高的特征。

1. 创建特征，PCA

主成分分析算法（Principal Component Analysis， PCA）的目的是找到能用较少信息描述数据集的特征组合。它意在发现彼此之间没有相关性、能够描述数据集的特征，确切说这些特征的**方差**跟整体方差没有多大差距，这样的特征也被称为主成分。这也就意味着，借助这种方法，就能通过更少的特征捕获到数据集的大部分信息。

PCA跟其他转换器用法类似。它只有主成分数量这一个参数。它默认会返回数据集中的所有特征。然而， PCA会对返回结果根据方差大小进行排序，返回的第一个特征方差最大，第二个特征方差稍小，以此类推。因此，前几个特征往往就能够解释数据集的大部分信息。

from sklearn.decomposition import PCA

pca = PCA(n\_components=5)

Xd = pca.fit\_transform(X)

返回的结果Xd矩阵只有五个特征，但是不容小觑，我们看一下每个特征的方差。

np.set\_printoptions(precision=3, suppress=True)

pca.explained\_variance\_ratio\_

支持向量机多元分类

SVC实现多元分类，使用decision\_function\_shape

**>>>** X = [[0], [1], [2], [3]]

**>>>** Y = [0, 1, 2, 3]

**>>>** clf = svm.SVC(decision\_function\_shape='ovo')

**>>>** clf.fit(X, Y)

## [sklearn.linear\_model](http://sklearn.apachecn.org/cn/0.19.0/modules/classes.html#module-sklearn.linear_model): 广义线性模型

1. sklearn.linear\_model.**LinearRegression**(fit\_intercept=True, normalize=False, copy\_X=True, n\_jobs=1) # 线性回归，最小二乘法，属性有coef\_ ：x的系数，intercept\_ :截距
2. sklearn.linear\_model.**Ridge**(alpha=1.0, fit\_intercept=True, normalize=False, copy\_X=True, max\_iter=None, tol=0.001, solver='auto', random\_state=None) #岭回归，关键参数alpha \alpha \geq 0 是控制系数收缩量的复杂性参数： \alpha 的值越大，收缩量越大，这样系数对共线性的鲁棒性也更强。

故：一般写estimator = Ridge(alpha=0.5)

属性有：coef\_、intercept、n\_iter\_

效果比最小二乘法好

linear\_model.**RidgeCV**(alphas=[0.1, 1.0, 10.0]) #添加了交叉检验，只是对alphas的

1. sklearn.linear\_model.**BayesianRidge**(n\_iter=300, tol=0.001, alpha\_1=1e-06, alpha\_2=1e-06, lambda\_1=1e-06, lambda\_2=1e-06, compute\_score=False, fit\_intercept=True, normalize=False, copy\_X=True, verbose=False) # 贝叶斯岭回归，alpha与lambda为超参，一般用默认，也可以调。对于解决病态问题，效果比最小二乘法好。
2. sklearn.linear\_model.**LogisticRegression**(penalty='l2', dual=False, tol=0.0001, C=1.0, fit\_intercept=True, intercept\_scaling=1, class\_weight=None, random\_state=None, solver='liblinear', max\_iter=100, multi\_class='ovr', verbose=0, warm\_start=False, n\_jobs=1) # 逻辑回归，做分类的

参数solver有以下选项：“liblinear”， “newton-cg”， “lbfgs”， “sag” 和 “saga” 等这些优化算法 区别看文档

“liblinear”： 坐标下降算法

“sag”平均随机梯度下降算法

saga” 求解器 [[7]](http://sklearn.apachecn.org/cn/0.19.0/modules/linear_model.html#id41) 是 “sag” 的一类变体，它支持非平滑（non-smooth）的 L1 正则选项 penalty="l1" 。因此对于稀疏多项式 logistic 回归 ，往往选用该求解器。“saga” 一般都是最佳的选择，但出于一些历史遗留原因默认的是 “liblinear” 。

# [sklearn.svm](http://sklearn.apachecn.org/cn/0.19.0/modules/classes.html#module-sklearn.svm).SVC

sklearn.svm.**SVC**(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache\_size=200, class\_weight=None, verbose=False, max\_iter=-1, decision\_function\_shape='ovr', random\_state=None)

kernel: ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable.

support\_vectors\_

做多元分类时，使用默认的decision\_function\_shape='ovr',或者’ovo’

# [sklearn.naive\_bayes](http://sklearn.apachecn.org/cn/0.19.0/modules/classes.html#module-sklearn.naive_bayes).GaussianNB

sklearn.naive\_bayes.**GaussianNB**(priors=None) # 高斯朴素贝叶斯，因为假定服从高斯分布。此外还有多项式分布和伯努利分布的贝叶斯算法。

**priors** : array-like, shape (n\_classes,) 各类的先验概率

**class\_prior\_、class\_count\_、theta\_、sigma\_**

# [sklearn.ensemble](http://sklearn.apachecn.org/cn/0.19.0/modules/classes.html#module-sklearn.ensemble).RandomForestClassifier

sklearn.ensemble.**RandomForestClassifier**(n\_estimators=10, criterion='gini', max\_depth=None, min\_samples\_split=2, min\_samples\_leaf=1, min\_weight\_fraction\_leaf=0.0, max\_features='auto', max\_leaf\_nodes=None, min\_impurity\_decrease=0.0, min\_impurity\_split=None, bootstrap=True, oob\_score=False, n\_jobs=1, random\_state=None, verbose=0, warm\_start=False, class\_weight=None)

# [sklearn.ensemble](http://sklearn.apachecn.org/cn/0.19.0/modules/classes.html#module-sklearn.ensemble).AdaBoostClassifier

sklearn.ensemble.**AdaBoostClassifier**(base\_estimator=None, n\_estimators=50, learning\_rate=1.0, algorithm='SAMME.R', random\_state=None)

# [sklearn.ensemble](http://sklearn.apachecn.org/cn/0.19.0/modules/classes.html#module-sklearn.ensemble).GradientBoostingClassifier

sklearn.ensemble.**RandomForestClassifier**（n\_estimators = 10，criterion =' gini '，max\_depth = None，min\_samples\_split = 2，min\_samples\_leaf = 1，min\_weight\_fraction\_leaf = 0.0，max\_features ='auto'，max\_leaf\_nodes = None，min\_impurity\_decrease = 0.0，min\_impurity\_split = None，bootstrap = True，oob\_score = False，n\_jobs = 1，random\_state = None，verbose = 0，warm\_start = False，class\_weight = None ）# 梯度提升决策树

**Note**超过两类的分类问题需要在每一次迭代时推导 n\_classes 个回归树。因此，所有的需要推导的树数量等于n\_classes \* n\_estimators 。对于拥有大量类别的数据集我们强烈推荐使用 **[RandomForestClassifier](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.ensemble.RandomForestClassifier.html" \l "sklearn.ensemble.RandomForestClassifier" \o "sklearn.ensemble.RandomForestClassifier)** 来代替[**GradientBoostingClassifier**](http://sklearn.apachecn.org/cn/0.19.0/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html#sklearn.ensemble.GradientBoostingClassifier) 。

基于sklearn聚类算法的问题

**sklearn.cluster.KMeans(n\_clusters=8, init='k-means++', n\_init=10, max\_iter=300, tol=0.0001, precompute\_distances='auto', verbose=0, random\_state=None, copy\_x=True, n\_jobs=1, algorithm='auto')**

1. 收敛：KMean中参数init=’k-means++’，自动地能够更快地找到收敛路径
2. 收敛：n\_init=10，遍历10次数据，每次的中心种子不同，最后返回最好的收敛路径
3. 预先计算距离：**precompute\_distances** : {‘auto’, True, False},预先计算距离，速度快但很耗内存，一般选择默认的auto。auto以数据量为判断是否预先计算距离，大于12 million，大概100MB,则不预先计算
4. 计算的算法选择：**algorithm** : “auto”, “full” or “elkan”, default=”auto”。密集数据选择elkan,稀疏数据选择full
5. KMeans的属性

|  |  |
| --- | --- |
| **Attributes:** | **cluster\_centers\_** : array, [n\_clusters, n\_features]  Coordinates of cluster centers  **labels\_ :** :  Labels of each point  **inertia\_** : float |

# 网格搜索

* (构建网格，增) GridSearchCV(estimator,param\_grid,scoring,cv)#仅列出常用的，其他的用默认值
* (获取信息，查)

cv\_results\_:具体的结果

**best\_estimator\_，best\_score\_，best\_params\_，scorer\_（仅列出常用的）**

# 管道Pipeline

* (构建管道，**增**)pipe = make\_pipeline(A(),B()...) # make)pipeline自动给评估器命名
* (这是获得评估器具体信息，**查**）管道中的评估器作为一个列表保存在steps属性内，并作为dict保存在named\_steps中去别在于pipe.steps[0]返回第一个评估器，pipe.named\_steps[‘A’]返回名为A的评估器。
* （修改指定评估器的参数，**改**）通过 <estimator>\_\_<parameter> 【注意，有两条下划线】语义来访问，如pipe.set\_params(clf\_\_C=10) 这对网格搜索非常关键
* 结合网格搜索

**>>> from** **sklearn.model\_selection** **import** GridSearchCV

**>>>** param\_grid = dict(reduce\_dim\_\_n\_components=[2, 5, 10],

**...**  clf\_\_C=[0.1, 10, 100])注意网格参数的命名，这是评估器的修改参数！！！

**>>>** grid\_search = GridSearchCV(pipe, param\_grid=param\_grid)

* 使用缓存转化器，避免重复计算，（适配转化器很耗费计算资源的）设置memory参数即可

**>>>** estimators = [('reduce\_dim', PCA()), ('clf', SVC())]

**>>> cachedir = mkdtemp()**

**>>>** pipe = Pipeline(estimators, **memory**=cachedir)