

在机器学习和数据挖掘的应用中，scikit-learn是一个功能强大的python包。在数据量不是过大的情况下，可以解决大部分问题。学习使用scikit-learn的过程中，我自己也在补充着机器学习和数据挖掘的知识。这里根据自己学习sklearn的经验，我做一个总结的笔记。另外，我也想把这篇笔记一直更新下去。

1 scikit-learn基础介绍

1.1 估计器（Estimator）

估计器，很多时候可以直接理解成分类器，主要包含两个函数：

- fit(): 训练算法，设置内部参数。接收训练集和类别两个参数。
 - predict(): 预测测试集类别，参数为测试集。
- 大多数scikit-learn估计器接收和输出的数据格式均为numpy数组或类似格式。

1.2 转换器（Transformer）

转换器用于数据预处理和数据转换，主要是三个方法：

- fit(): 训练算法，设置内部参数。
- transform(): 数据转换。
- fit_transform(): 合并fit和transform两个方法。

1.3 流水线（Pipeline）

sklearn.pipeline包

流水线的功能：

- 跟踪记录各步骤的操作（以方便地重现实验结果）
- 对各步骤进行一个封装
- 确保代码的复杂程度不至于超出掌控范围

基本使用方法

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。

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())
])
```

1.4 预处理

主要在sklearn.preprocessing包下。

规范化：

- MinMaxScaler :最大最小值规范化

- **Normalizer** :使每条数据各特征值的和为1
- **StandardScaler** :为使各特征的均值为0，方差为1

编码：

- **LabelEncoder** : 把字符串类型的数据转化为整型
- **OneHotEncoder** : 特征用一个二进制数字来表示
- **Binarizer** :为将数值型特征的二值化
- **MultiLabelBinarizer** : 多标签二值化

1.5 特征

1.5.1 特征抽取

包：`sklearn.feature_extraction`

特征抽取是数据挖掘任务最为重要的一个环节，一般而言，它对最终结果的影响要高过数据挖掘算法本身。只有先把现实特征表示出来，才能借助数据挖掘的力量找到问题的答案。特征选择的另一个优点在于：降低真实世界的复杂度，模型比现实更容易操纵。

一般最常用的特征抽取技术都是高度针对具体领域的，对于特定的领域，如图像处理，在过去一段时间已经开发了各种特征抽取的技术，但这些技术在其他领域的应用却非常有限。

- **DictVectorizer**：将dict类型的list数据，转换成numpy array
- **FeatureHasher**：特征哈希，相当于一种降维技巧
- **image**：图像相关的特征抽取
- **text**：文本相关的特征抽取
- **text.CountVectorizer**：将文本转换为每个词出现的个数的向量
- **text.TfidfVectorizer**：将文本转换为tfidf值的向量
- **text.HashingVectorizer**：文本的特征哈希

示例

```
t1 = '今天 今天 天气 不错 我们 愉快 玩耍'
t2 = '今天 锻炼 舒服 天气 一般'
t3 = '天气 糟糕'
```

data.png

CountVectorize只数出现个数

```
[[0 1 2 1 1 1 1 0 0 0]
 [1 0 1 1 0 0 0 0 1 1]
 [0 0 0 1 0 0 0 1 0 0]]
```

count.png

```
[ 2.  2.  6.  5.  2.  0.  3.  4.  1.  3.]
[ 2.  1.  4.  1.  1.  1.  1.  1.  1.  1.]
[ 0.  0.  1.  1.  2.  0.  1.  2.  0.  0.]]
```

hash.png

TfidfVectorizer: 个数+归一化 (不包括idf)

```
0. 0.33333333 0.66666667 0.33333333 0.33333333 0.33333333
0.33333333 0. 0. 0. ]
0.4472136 0. 0.4472136 0.4472136 0. 0. 0.
0. 0.4472136 0.4472136 ]
0. 0. 0. 0.70710678 0. 0. 0.
0.70710678 0. 0. ]]
```

tfidf(without idf).png

1.5.2 特征选择

包: `sklearn.feature_selection`

特征选择的原因如下:

- (1)降低复杂度
- (2)降低噪音
- (3)增加模型可读性

- **VarianceThreshold**: 删除特征值的方差达不到最低标准的特征
- **SelectKBest**: 返回 k 个最佳特征
- **SelectPercentile**: 返回表现最佳的前 $r\%$ 个特征

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 (χ^2)。其他方法还有互信息和信息熵。

- **chi2**: 卡方检验 (χ^2)

1.6 降维

包: `sklearn.decomposition`

- 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。

1.7 组合

包: `sklearn.ensemble`

组合技术即通过聚集多个分类器的预测来提高分类准确率。

常用的组合分类器方法:

- (1)通过处理训练数据集。即通过某种抽样分布,对原始数据进行再抽样,得到多个训练集。常用的方法有装袋 (bagging) 和提升 (boosting)。
- (2)通过处理输入特征。即通过选择输入特征的子集形成每个训练集。适用于有大量冗余特征的数据集。随机森林 (Random forest) 就是一种处理输

入特征的组合方法。

(3)通过处理类标号。适用于多分类的情况，将类标号随机划分成两个不相交的子集，再把问题变为二分类问题，重复构建多次模型，进行分类投票。

- **BaggingClassifier**: Bagging分类器组合
- **BaggingRegressor**: Bagging回归器组合
- **AdaBoostClassifier**: AdaBoost分类器组合
- **AdaBoostRegressor**: AdaBoost回归器组合
- **GradientBoostingClassifier**: GradientBoosting分类器组合
- **GradientBoostingRegressor**: GradientBoosting回归器组合
- **ExtraTreeClassifier**: ExtraTree分类器组合
- **ExtraTreeRegressor**: ExtraTree回归器组合
- **RandomTreeClassifier**: 随机森林分类器组合
- **RandomTreeRegressor**: 随机森林回归器组合

使用举例

```
AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
algorithm="SAMME",
n_estimators=200)
```

解释

装袋 (bagging): 根据均匀概率分布从数据集中重复抽样 (有放回), 每个自助样本集和原数据集一样大, 每个自助样本集含有原数据集大约63%的数据。训练k个分类器, 测试样本被指派到得票最高的类。

提升 (boosting): 通过给样本设置不同的权值, 每轮迭代调整权值。不同的提升算法之间的差别, 一般是 (1) 如何更新样本的权值, (2) 如何组合每个分类器的预测。其中Adaboost中, 样本权值是增加那些被错误分类的样本的权值, 分类器C_i的重要性依赖于它的错误率。

Boosting主要关注降低偏差, 因此Boosting能基于泛化性能相当弱的学习器构建出很强的集成; Bagging主要关注降低方差, 因此它在不剪枝的决策树、神经网络等学习器上效用更为明显。偏差指的是算法的期望预测与真实预测之间的偏差程度, 反应了模型本身的拟合能力; 方差度量了同等大小的训练集的变动导致学习性能的变化, 刻画了数据扰动所导致的影响。

1.8 模型评估 (度量)

包: **sklearn.metrics**

sklearn.metrics包含评分方法、性能度量、成对度量和距离计算。

分类结果度量

参数大多是y_true和y_pred。

- **accuracy_score**: 分类准确度
- **confusion_matrix**: 分类混淆矩阵
- **classification_report**: 分类报告
- **precision_recall_fscore_support**: 计算精确度、召回率、f、支持率
- **jaccard_similarity_score**: 计算jaccard相似度
- **hamming_loss**: 计算汉明损失
- **zero_one_loss**: 0-1损失

- **hinge_loss**: 计算hinge损失
- **log_loss**: 计算log损失

其中，F1是以**每个类别**为基础进行定义的，包括两个概念：准确率（precision）和召回率（recall）。准确率是指预测结果属于某一类的个体，实际属于该类的比例。召回率是被正确预测为某类的个体，与数据集中该类个体总数的比例。F1是准确率和召回率的调和平均数。

回归结果度量

- **explained_varicance_score**: 可解释方差的回归评分函数
- **mean_absolute_error**: 平均绝对误差
- **mean_squared_error**: 平均平方误差

多标签的度量

- **coverage_error**: 涵盖误差
- **label_ranking_average_precision_score**: 计算基于排名的平均误差Label ranking average precision (LRAP)

聚类的度量

- **adjusted_mutual_info_score**: 调整的互信息评分
- **silhouette_score**: 所有样本的轮廓系数的平均值
- **silhouette_sample**: 所有样本的轮廓系数

1.9 交叉验证

包: **sklearn.cross_validation**

- **KFold**: K-Fold交叉验证迭代器。接收元素个数、fold数、是否清洗
- **LeaveOneOut**: LeaveOneOut交叉验证迭代器
- **LeavePOut**: LeavePOut交叉验证迭代器
- **LeaveOneLabelOut**: LeaveOneLabelOut交叉验证迭代器
- **LeavePLabelOut**: LeavePLabelOut交叉验证迭代器

LeaveOneOut(n) 相当于 KFold(n, n_folds=n) 相当于LeavePOut(n, p=1)。

LeaveP和LeaveOne差别在于leave的个数，也就是测试集的尺寸。LeavePLabel和LeaveOneLabel差别在于leave的Label的种类的个数。

LeavePLabel这种设计是针对可能存在第三方的Label，比如我们的数据是一些季度的数据。那么很自然的一个想法就是把1,2,3个季度的数据当做训练集，第4个季度的数据当做测试集。这个时候只要输入每个样本对应的季度Label，就可以实现这样的功能。

以下是实验代码，尽量自己多实验去理解。

```
#coding=utf-8
import numpy as np
import sklearnfrom sklearn
import cross_validation
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8],[9, 10]])
y = np.array([1, 2, 1, 2, 3])
def show_cross_val(method):
    if method == "lolo":
        labels = np.array(["summer", "winter", "summer", "winter", "spring"])
        cv = cross_validation.LeaveOneLabelOut(labels)
    elif method == 'lplo':
        labels = np.array(["summer", "winter", "summer", "winter", "spring"])
        cv = cross_validation.LeavePLabelOut(labels,p=2)
    elif method == 'loo':
```

```

cv = cross_validation.LeaveOneOut(n=len(y))
elif method == 'lpo':
    cv = cross_validation.LeavePOut(n=len(y),p=3)
for train_index, test_index in cv:
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    print "X_train: ",X_train
    print "y_train: ", y_train
    print "X_test: ",X_test
    print "y_test: ",y_test
if __name__ == '__main__':
    show_cross_val("lpo")

```

常用方法

- **train_test_split**: 分离训练集和测试集（不是K-Fold）
- **cross_val_score**: 交叉验证评分，可以指认cv为上面的类的实例
- **cross_val_predict**: 交叉验证的预测。

1.10 网格搜索

包: `sklearn.grid_search`

网格搜索最佳参数

- **GridSearchCV**: 搜索指定参数网格中的最佳参数
- **ParameterGrid**: 参数网格
- **ParameterSampler**: 用给定分布生成参数的生成器
- **RandomizedSearchCV**: 超参的随机搜索
通过`best_estimator_.get_params()`方法，获取最佳参数。

1.11 多分类、多标签分类

包: `sklearn.multiclass`

- **OneVsRestClassifier**: 1-rest多分类（多标签）策略
- **OneVsOneClassifier**: 1-1多分类策略
- **OutputCodeClassifier**: 1个类用一个二进制码表示

示例代码

```

#coding=utf-8
from sklearn import metrics
from sklearn import cross_validation
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import MultiLabelBinarizer
import numpy as np
from numpy import random
X=np.arange(15).reshape(5,3)
y=np.arange(5)
Y_1 = np.arange(5)
random.shuffle(Y_1)
Y_2 = np.arange(5)
random.shuffle(Y_2)
Y = np.c_[Y_1,Y_2]
def multiclassSVM():
    X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y, test_size=0.2,random_state=0)

    model = OneVsRestClassifier(SVC())
    model.fit(X_train, y_train)
    predicted = model.predict(X_test)
    print predicted

```

```

print predicted
def multilabelSVM():
    Y_enc = MultiLabelBinarizer().fit_transform(Y)
    X_train, X_test, Y_train, Y_test = cross_validation.train_test_split(X, Y_enc, test_size=0.2, random_state=0)
    model = OneVsRestClassifier(SVC())
    model.fit(X_train, Y_train)
    predicted = model.predict(X_test)
    print predicted
if __name__ == '__main__':
    multiclassSVM()
    # multilabelSVM()

```

上面的代码测试了svm在OneVsRestClassifier的包装下，分别处理多分类和多标签的情况。特别注意，在多标签的情况下，输入必须是二值化的。所以需要MultiLabelBinarizer()先处理。

2 具体模型

2.1 朴素贝叶斯 (Naive Bayes)

包：sklearn.cross_validation

$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i | y),$$

朴素贝叶斯.png

朴素贝叶斯的特点是分类速度快，分类效果不一定是最好的。

- GasussianNB：高斯分布的朴素贝叶斯
- MultinomialNB：多项式分布的朴素贝叶斯
- BernoulliNB：伯努利分布的朴素贝叶斯

所谓使用什么分布的朴素贝叶斯，就是假设P(x_i|y)是符合哪一种分布，比如可以假设其服从高斯分布，然后用最大似然法估计高斯分布的参数。

$$P(x_i | y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

高斯分布.png

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

多项式分布.png

$$P(x_i | y) = P(i | y)x_i + (1 - P(i | y))(1 - x_i)$$

伯努利分布.png

3 scikit-learn扩展

3.0 概览

具体的扩展，通常要继承`sklearn.base`包下的类。

- **BaseEstimator**：估计器的基类
- **ClassifierMixin**：分类器的混合类
- **ClusterMixin**：聚类器的混合类
- **RegressorMixin**：回归器的混合类
- **TransformerMixin**：转换器的混合类

关于什么是Mixin（混合类），具体可以看这个[知乎链接](#)。简单地理解，就是带有实现方法的接口，可以将其看做是组合模式的一种实现。举个例子，比如说常用的TfidfTransformer，继承了BaseEstimator，TransformerMixin，因此它的基本功能就是单一职责的估计器和转换器的组合。

3.1 创建自己的转换器

在特征抽取的时候，经常会发现自己的一些数据预处理的方法，sklearn里可能没有实现，但若直接在数据上改，又容易将代码弄得混乱，难以重现实验。这个时候最好自己创建一个转换器，在后面将这个转换器放到pipeline里，统一管理。

例如《Python数据挖掘入门与实践》书中的例子，我们想接收一个numpy数组，根据其均值将其离散化，任何高于均值的特征值替换为1，小于或等于均值的替换为0。

代码实现：

```
from sklearn.base import TransformerMixin
from sklearn.utils import as_float_array

class MeanDiscrete(TransformerMixin):

    #计算出数据集的均值，用内部变量保存该值。
    def fit(self, X, y=None):
        X = as_float_array(X)
        self.mean = np.mean(X, axis=0)
        #返回self，确保在转换器中能够进行链式调用（例如调用transformer.fit(X).transform(X)）
        return self

    def transform(self, X):
        X = as_float_array(X)
        assert X.shape[1] == self.mean.shape[0]
        return X > self.mean
```