Task

- 数据拆分：训练数据集&测试数据集
- 评价分类结果：精准度、混淆矩阵、精准率、召回率、F1 Score、ROC曲线等
- 评价回归结果：MSE、RMSE、MAE、R Squared

数据拆分

判断机器学习算法的性能

机器学习经过训练得到的模型，其意义在于真实环境中的使用； 将全部的原始数据当做训练集直接训练出模型，然后投入到真实环境中，这种做法是不恰当的，存在问题：

- 如果模型效果很差，没有机会通过实际调试就直接应用到实际当中，怎么办？（# 实例：股市预测）
- 在真实环境中，开发者难以拿到真实label（输出结果），则无从得知模型的效果？（# 实例：银行发放信用卡）

方案：训练数据集与测试数据集切分（train test split），将原始数据的80%作为训练数据来训练模型，另外20%作为测试数据，通过测试数据直接判断模型的效果，在模型进入真实环境前改进模型；

鸢尾花train_test

鸢尾花数据集是UCI数据库中常用数据集。

```
import numpy as np
from sklearn import datasets
import matplotlib.pyplot as plt

iris = datasets.load_iris()

X = iris.data
y = iris.target

X.shape
输出：(150, 4)

X.shape
输出：(150,)
```

一般情况下按照0.8:0.2的比例进行拆分，有时数据集是顺序排列的，需要做一个shuffle操作，将数据集打乱；

因数据集的特征和标签是分开的，分别乱序后，原来的关系就不在了， 解决办法：

- 将X和y合并为同一个矩阵，然后对矩阵进行shuffle，之后再分解
- 对y的索引进行乱序，根据索引确定与X的对应关系，最后再通过乱序的索引进行赋值

方法一

```
# 方法1# 使用concatenate函数进行拼接，因为传入的矩阵必须具有相同的形状。因此需要对label进行reshape操作，reshape(-1,1)表示行数自动计算，1列。axis=1表示纵向拼接。
tempConcat = np.concatenate((X, y.reshape(-1,1)), axis=1)

# 拼接好后，直接进行乱序操作
np.random.shuffle(tempConcat)

# 再将shuffle后的数组使用split方法拆分
shuffle_X,shuffle_y = np.split(tempConcat, [4], axis=1)

# 设置划分的比例
test_ratio = 0.2test_size = int(len(X) * test_ratio)

X_train = shuffle_X[test_size:]
y_train = shuffle_y[test_size:]
X_test = shuffle_X[:test_size]
y_test = shuffle_y[:test_size]

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
输出：(120, 4)
(30, 4)
(120, 1)
(30, 1)
```

方法二

```
# 方法2# 将x长度这么多的数，返回一个新的打乱顺序的数组，注意，数组中的元素不是原来的数据，而是混乱的索引
shuffle_index = np.random.permutation(len(X))

# 指定测试数据的比例
test_ratio = 0.2test_size = int(len(X) * test_ratio)

test_index = shuffle_index[:test_size]
train_index = shuffle_index[test_size:]
X_train = X[train_index]
X_test = X[test_index]
y_train = y[train_index]
y_test = y[test_index]
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

输出：(120, 4)
(30, 4)
(120,)
(30,)
```

编写自己的train_test_split

```
import numpy as np
def train_test_split(X, y, test_ratio=0.2, seed=None):
    """将矩阵X和标签y按照test_ration分割成X_train, X_test, y_train, y_test"""

    # the size of X must be equal to the size of y
    assert X.shape[0] == y.shape[0]

    # test_train must be valid
    assert 0.0 <= test_ratio <= 1.0

    if seed:
        # 是否使用随机种子，使随机结果相同，方便debug
        np.random.seed(seed)

    # permutation(n) 可直接生成一个随机排列的数组，含有n个元素
    shuffle_index = np.random.permutation(len(X))

    test_size = int(len(X) * test_ratio)
    test_index = shuffle_index[:test_size]
    train_index = shuffle_index[test_size:]
    X_train = X[train_index]
    X_test = X[test_index]
    y_train = y[train_index]
    y_test = y[test_index]

    return X_train, X_test, y_train, y_test
```

调用

```
from myAlgorithm.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

输出：(120, 4)
(30, 4)
(120,)
(30,)
```

kNN算法使用分割后的数据集

X_train, y_train通过fit传入算法，然后对X_test做预测，得到y_predict。

把y_predict和实际的结果y_test进行一个比较，看有多少个元素一样

```
from myAlgorithm.kNN import kNNClassifier

my_kNNClassifier = kNNClassifier(k=3)
my_kNNClassifier.fit(X_train, y_train)

y_predict = my_kNNClassifier.predict(X_test)
y_predict
y_test
# 两个向量的比较，返回一个布尔型向量，对这个布尔向量（faluse=1，true=0）sum，
sum(y_predict == y_test)
# 29
sum(y_predict == y_test)/len(y_test)
# 0.96666666666667
```

sklearn中的train_test_split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=666)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
输出：(112, 4)
(38, 4)
(112,)
(38,)
```

评价分类结果

分类准确度Accuracy

accuracy_score：函数计算分类准确率，返回被正确分类的样本比例（default）或者是数量（normalize=False）

在多标签分类问题中，该函数返回子集的准确率，

对于一个给定的多标签样本，如果预测得到的标签集合与该样本真正的标签集合严格吻合，则subset accuracy =1.0否则是0.0

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection
import train_test_splitfrom sklearn.neighbors
import KNeighborsClassifier

# 手写数字数据集，封装好的对象，可以理解为一个字段
digits = datasets.load_digits()

# 可以使用keys()方法来看一下数据集的详情
digits.keys()

# 5620张图片，每张图片有64个像素点即特征（8*8整数像素图像），每个特征的取值范围是1～16（sklearn中的不全），对应的分类结果是10个数字
print(digits.DESCR)

# 特征的shape
X = digits.data
X.shape(1797, 64)

# 标签的shape
y = digits.target
y.shape(1797, )

# 标签分类
digits.target_names

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# 去除某一个具体的数据，查看其特征以及标签信息
some_digit = X[666]
some_digit
array([ 0.,  0.,  5., 15., 14.,  3.,  0.,  0.,  0.,  0., 13., 15.,  9.,15.,  2.,
 0.,  0.,  4., 16., 12.,  0., 10.,  6.,  0.,  0.,  8.,16.,  9.,  0.,  8., 10.,
 0.,  0.,  7., 15.,  5.,  0., 12., 11.,0.,  0.,  7., 13.,  0.,  5., 16.,  6.,
 0.,  0.,  0., 16., 12.,15., 13.,  1.,  0.,  0.,  0.,  6., 16., 12.,  2.,  0.,
 0.])
y[666]0
# 也可以这条数据进行可视化
some_digmit_image = some_digit.reshape(8, 8)
plt.imshow(some_digmit_image, cmap = matplotlib.cm.binary)
plt.show()
```

自己实现分类的准确性

```
X_train, X_test, y_train, y_test = train_test_split(X, y)
knn_clf = KNeighborsClassifier(n_neighbors=3)
knn_clf.fit(X_train, y_train)
y_predict = knn_clf.predict(X_test)

# 比对y_predict和y_test结果是否一致
sum(y_predict == y_test) / len(y_test)

# 0.9955555555555555
```

```
import numpy as np
from math import sqrt
def accuracy_score(y_true, y_predict):
    """计算y_true和y_predict之间的准确率"""
    # the size of y_true must be equal to the size of y_predict
    assert y_true.shape[0] != y_predict.shape[0]
    return sum(y_true == y_predict) / len(y_true)
```

用classifier将我们的预测值y_predicr预测出来了，再去看和真值的比例。

但是有时候我们对预测值y_predicr是多少不感兴趣，我们只对模型的准确率感兴趣。

```
def score(self, X_test, y_test):
    """根据X_test进行预测，给出预测的真值y_test，计算预测算法的准确度"""
    y_predict = self.predict(X_test)
    return accuracy_score(y_test, y_predict)
```

sklearn 中的准确度

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=666)
knn_clf = KNeighborsClassifier(n_neighbors=3)
knn_clf.fit(X_train, y_train)
y_predict = knn_clf.predict(X_test)

accuracy_score(y_test, y_predict)
输出：0.9888888888888889
# 不看y_predict

knn_clf.score(X_test,y_test)
输出：0.9888888888888889
```

混肴矩阵

对于二分类问题来说，所有的问题被分为0和1两类，混淆矩阵是2*2的矩阵：

	预测值0	预测值1
真实值0	TN	FP
真实值1	FN	TP

- TN：真实值是0，预测值也是0，即我们预测是negative，预测正确了。
- FP：真实值是0，预测值是1，即我们预测是positive，但是预测错误了。
- FN：真实值是1，预测值是0，即我们预测是negative，但预测错误了。
- TP：真实值是1，预测值是1，即我们预测是positive，预测正确了。

现在假设有1万人进行预测，填入混淆矩阵如下：

	预测值0	预测值1
真实值0	9978（FN）	12 (FP)
真实值1	2 (TN)	8 (TP)

混淆矩阵表达的信息比简单的分类准确度更全面，因此可以通过混淆矩阵得到一些有效的指标。

精准率与召回率

- 精准率：$precision = \frac{TP}{TP+FP}$ **即预测值为1，且预测对了的比例**

  在有些数据中，我们通常更关注值为1的特征，准确率为我们关注的那个事件，预测的有多精准。
- 召回率：$recall = \frac{TP}{TP+FN}$ **即所有真实值为1的数据中，预测对了的个数**；也就是我们关注的**那个事件真实发生的情况下**，我们成功预测出的比例是多少

编程实现

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

digits = datasets.load_digits()
X = digits.data
y = digits.target.copy()

# 要构造偏斜数据，将数字9的对应索引的元素设置为1，0～8设置为0
y[digits.target==9]=1
y[digits.target!=9]=0

# 使用逻辑回归做一个分类
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)

log_reg = LogisticRegression()
log_reg.fit(X_train,y_train)
# 得到X_test所对应的预测值
y_log_predict = log_reg.predict(X_test)
```

分类准确度

```
log_reg.score(X_test, y_test)

# 输出：
# 0.9755555555555555
```

定义混淆矩阵的四个指标：TN

```
def TN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    # (y_true == 0)：向量与数值按位比较，得到的是一个布尔向量
    # 向量与向量按位与，结果还是布尔向量
    # np.sum 计算布尔向量中True的个数(True记为1，False记为0)
    return np.sum((y_true == 0) & (y_predict == 0))  # 向量与向量按位与，结果还是向量
TN(y_test, y_log_predict)
# 输出：
# 403
```

定义混淆矩阵的四个指标：FP

```
def FP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    # (y_true == 0)：向量与数值按位比较，得到的是一个布尔向量
    # 向量与向量按位与，结果还是布尔向量
    # np.sum 计算布尔向量中True的个数(True记为1，False记为0)
    return np.sum((y_true == 0) & (y_predict == 1))  # 向量与向量按位与，结果还是向量
FP(y_test, y_log_predict)
# 输出：
# 2
```

定义混淆矩阵的四个指标：FN

```
def FN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    # (y_true == 0)：向量与数值按位比较，得到的是一个布尔向量
    # 向量与向量按位与，结果还是布尔向量
    # np.sum 计算布尔向量中True的个数(True记为1，False记为0)
    return np.sum((y_true == 1) & (y_predict == 0))  # 向量与向量按位与，结果还是向量
FN(y_test, y_log_predict)
# 输出：
# 9
```

定义混淆矩阵的四个指标：TP

```
def TP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    # (y_true == 0)：向量与数值按位比较，得到的是一个布尔向量
    # 向量与向量按位与，结果还是布尔向量
    # np.sum 计算布尔向量中True的个数(True记为1，False记为0)
    return np.sum((y_true == 1) & (y_predict == 1))  # 向量与向量按位与，结果还是向量
TP(y_test, y_log_predict)
# 输出：
# 36
```

输出混淆矩阵

```
def confusion_matrix(y_true, y_predict):
    return np.array([
        [TN(y_true, y_predict), FP(y_true, y_predict)],
        [FN(y_true, y_predict), TP(y_true, y_predict)]
    ])
confusion_matrix(y_test, y_log_predict)

# 输出:
# array([[403,   2],
#        [  9,  36]])
```

计算精准率

```
def precision_score(y_true, y_predict):
    tp = TP(y_true, y_predict)
    fp = FP(y_true, y_predict)
    try:
        return tp / (tp + fp)
    except:
        return 0.0
precision_score(y_test, y_log_predict)

# 输出:
# 0.9473684210526315
```

计算召回率

```
def recall_score(y_true, y_predict):
    tp = TP(y_true, y_predict)
    fn = FN(y_true, y_predict)
    try:
        return tp / (tp + fn)
    except:
        return 0.0
recall_score(y_test, y_log_predict)
# 输出:
# 0.8
```

sklearn中的混肴矩阵，精准率和召回率

混淆矩阵

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, y_log_predict)
# 输出:
# array([[403,   2],
#        [  9,  36]])
```

精准率

```
from sklearn.metrics import precision_score

precision_score(y_test, y_log_predict)
# 输出:
# 0.9473684210526315
```

召回率

```
from sklearn.metrics import recall_score

recall_score(y_test, y_log_predict)
# 输出:
# 0.8
```

F1-Score

一般作为多分类问题的最终评测方法，

F1-Score 是精确率与召回率的调和平均数，最大为1， 最小为0

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

F1分数认为召回率和精确率同等重要，

F2分数认为召回率的重要程度是精确率的2倍，

F0.5分数认为召回率的重要程度是精确率的一半。

$$F_\beta = (1 + \beta^2) * \frac{precision * recall}{(\beta^2 * precision) + recall}$$

G分数是另一种统一精确率和的召回率系统性能评估标准，G分数被定义为召回率和精确率的几何平均数。

$$G = \sqrt{precision * recall}$$

ROC曲线

ROC(Receiver Operating Characteristic curve)， 即受试者特征曲线，用来展示分类模型性能的可视化技术

ROC 曲线展示了当改变在模型中识别为正例的阈值时，召回率和精度的关系会如何变化。

如果我们有一个用来识别疾病的模型，我们的模型可能会为每一种疾病输出介于 0 到 1 之间的一个分数，为了将某个病人标记为患有某种疾病（一个正例标签），我们为每种疾病在这个范围内设置一个阈值，通过改变这个阈值，我们可以尝试实现合适的精度和召回率之间的平衡。

ROC 曲线在 Y 轴上画出了真正例率（TPR），在 X 轴上画出了假正例率 (FPR)。

TPR 是召回率，FPR 是反例被报告为正例的概率。这两者都可以通过混淆矩阵计算得到。

评价回归结果

简单线性回归的目标是：

> 已知训练数据样本x、y，找到 a 和 b 的值，使 $\sum_{i=1}^{m}(y^i - ax^i - b)^2$尽可能小

实际上是找到训练数据集中$\sum(y_{train}^i - \hat{y}_{train}^i)^2$ 最小。

衡量标准是看在测试数据集中y的真实值与预测值之间的差距。

均方误差MSE（Mean Squarred Error）

$$\frac{1}{m}\sum_{i=1}^{m}(y_{test}^{i}-\hat{y}_{train}^{i})^2$$

均方根误差RMSE（Root Mean Squarred Error）

$$\sqrt{MSE_{test}}$$

平均绝对误差MAE （Mean Absolute Error）

$$\frac{1}{m}\sum_{i=1}^{m}|y_{test}^{i}-\hat{y}_{test}^{i}|$$

代码实现

1. 数据探索

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets

# 查看数据集描述
boston = datasets.load_boston()
print(boston.DESCR)
```

因测试简单回归算法，选择其中一个特征进行建模，选择RM

```
# 查看数据集的特征列表
boston.feature_names
输出：
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')

# 取出数据中的第六例的所有行（房间数量）
x = boston.data[:,5]
x.shape
输出：
(506,)

# 取出样本标签
y = boston.target
y.shape
输出：
(506,)

plt.scatter(x,y)
plt.show()
```

在图中我们可以看到 50W 美元的档分布着一些点。这些点可能是超出了限定范围（比如在问卷调查中，价格的最高档位是“50万及以上”，那么就全都划到50W上了，因此在本例中，可以将这部分数据去除）

```
np.max(y)
# 这里有一个骚操作，用比较运算符返回一个布尔值的向量，将其作为索引，直接在矩阵里对每个元素进行过滤。
x = x[y < 50.0]
y = y[y < 50.0]
plt.scatter(x,y)
plt.show()
```

2. 简单线性回归预测

```
from myAlgorithm.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, seed=666)
print(x_train.shape)    # (392,)
print(y_train.shape)    #(392,)
print(x_test.shape)     #(98,)
print(y_test.shape)     #(98,)

from myAlgorithm.SimpleLinearRegression import SimpleLinearRegression

reg = SimpleLinearRegression()
reg.fit(x_train,y_train)
print(reg.a_)   # 7.8608543562689555
print(reg.b_)   # -27.459342806705543

plt.scatter(x_train,y_train)
plt.plot(x_train, reg.predict(x_train),color='r')
plt.show()
```

```
y_predict = reg.predict(x_test)
print(y_predict)
```

```
# 均方误差
mse_test = np.sum((y_predict - y_test) ** 2) / len(y_test)
mse_test


from math import sqrt
# 均方根误差
rmse_test = sqrt(mse_test)
rmse_test

# 平均绝对误差
mae_test = np.sum(np.absolute(y_predict - y_test)) / len(y_test)
mae_test
```

sklearn 中的MSE和MAE

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

mean_squared_error(y_test, y_predict)
# 输出：24.156602134387438

mean_absolute_error(y_test, y_predict)
# 输出：3.5430974409463873
```

R Square

$$R^2 = 1 - \frac{SS_{residual}}{SS_{total}} = 1 - \frac{\sum(\hat{y}^i - y^i)^2}{\sum(\bar{y} - y^i)^2}$$

优点：

- 对于分子来说，预测值和真实值之差的平方和，即使用我们的模型预测产生的错误。
- 对于分母来说，是均值和真实值之差的平方和，即认为“预测值=样本均值”这个模型（Baseline Model）所产生的错误。
- 我们使用Baseline模型产生的错误较多，我们使用自己的模型错误较少。**因此用1减去较少的错误除以较多的错误，实际上是衡量了我们的模型拟合住数据的地方，即没有产生错误的相应指标。**

结论：

- $R^2 <= 1$
- R^2 越大越好，越大说明减数的分子小，错误率低；当我们预测模型不犯任何错误时， R^2最大值为1
- 当我们的模型等于基准模型时， R^2 = 0
- 如果$R^2 < 0$， 说明我们学习到的模型还不如基准模型。此时，很有可能我们的数据不存在任何线性关系。

R Square 代码实现

R^2 如果分子分母同时除以m：

$$R^2 = 1 - \frac{\sum(\hat{y}^i - y^i)^2/m}{\sum(\bar{y} - y^i)^2/m} = 1 = \frac{MSE(\hat{y}^i, y)}{Var(y)}$$

```
1 - mean_squared_error(y_test, y_predict) / np.var(y_test)

# 输出：0.61293168039373225
```

```
def score(self, x_test, y_test):
    """根据测试数据x_test、y_test计算简单线性回归准确度（R方）"""
    y_predict = self.predict(x_test)
    return r2_score(y_test, y_predict)
```

```
from myAlgorithm.metrics import r2_score
r2_score(y_test, y_predict)
```

```
from sklearn.metrics import r2_score
r2_score(y_test, y_predict)
```

总结

懒人模式，此次只是跟着参考文档学习了一遍，总体上了解整个模型应用以及模型参数的求解

参考链接

模型之母：线性回归的评价指标

机器学习的敲门砖：kNN算法（中）

评价分类结果（上）