# 回归模型

## 线性回归

```
输入      输出
0.5       5.0
0.6       5.5
0.8       6.0
1.1       6.8
1.4       7.0
...
y = f(x)
```

预测函数：$y = w_0+w_1x$

x: 输入

y: 输出

$w_0$和$w_1$: 模型参数

**所谓模型训练，就是根据已知的x和y，找到最佳的模型参数$w_0$ 和 $w_1$，尽可能精确地描述出输入和输出的关系。**

$5.0 = w_0 + w_1 \times 0.5$

$5.5 = w_0 + w_1 \times 0.6$

单样本误差：

根据预测函数求出输入为x时的预测值：$y' = w_0 + w_1x$，单样本误差为$1/2(y' - y)^2$。

总样本误差：

把所有单样本误差相加即是总样本误差：$1/2\ \Sigma(y' - y)^2$

损失函数：

$loss = 1/2\ \Sigma(w_0 + w_1x - y)^2$

所以损失函数就是总样本误差关于模型参数的函数，该函数属于三维数学模型，即需要找到一组$w_0$ $w_1$使得loss取极小值。

案例：画图模拟梯度下降的过程

1. 整理训练集数据，自定义梯度下降算法规则，求出$w_0$， $w_1$，绘制回归线。

```
import numpy as np
import matplotlib.pyplot as mp
train_x = np.array([0.5, 0.6, 0.8, 1.1, 1.4])
train_y = np.array([5.0, 5.5, 6.0, 6.8, 7.0])
test_x = np.array([0.45, 0.55, 1.0, 1.3, 1.5])
test_y = np.array([4.8, 5.3, 6.4, 6.9, 7.3])

times = 1000     # 定义梯度下降次数
lrate = 0.01     # 记录每次梯度下降参数变化率
epoches = []     # 记录每次梯度下降的索引
w0, w1, losses = [1], [1], []
```

```
for i in range(1, times + 1):
    epoches.append(i)
    loss = (((w0[-1] + w1[-1] * train_x) - train_y) ** 2).sum() / 2
    losses.append(loss)
    d0 = ((w0[-1] + w1[-1] * train_x) - train_y).sum()
    d1 = (((w0[-1] + w1[-1] * train_x) - train_y) * train_x).sum()
    print('{:4}> w0={:.8f}, w1={:.8f}, loss={:.8f}'.format(epoches[-1],
w0[-1], w1[-1], losses[-1]))
    w0.append(w0[-1] - lrate * d0)
    w1.append(w1[-1] - lrate * d1)

pred_test_y = w0[-1] + w1[-1] * test_x
mp.figure('Linear Regression', facecolor='lightgray')
mp.title('Linear Regression', fontsize=20)
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.scatter(train_x, train_y, marker='s', c='dodgerblue', alpha=0.5, s=80,
label='Training')
mp.scatter(test_x, test_y, marker='D', c='orangered', alpha=0.5, s=60,
label='Testing')
mp.scatter(test_x, pred_test_y, c='orangered', alpha=0.5, s=80,
label='Predicted')
mp.plot(test_x, pred_test_y, '--', c='limegreen', label='Regression',
linewidth=1)
mp.legend()
mp.show()
```

1. 绘制随着每次梯度下降，$w_0$，$w_1$，loss的变化曲线。

```
w0 = w0[:-1]
w1 = w1[:-1]

mp.figure('Training Progress', facecolor='lightgray')
mp.subplot(311)
mp.title('Training Progress', fontsize=20)
mp.ylabel('w0', fontsize=14)
mp.gca().xaxis.set_major_locator(mp.MultipleLocator(100))
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.plot(epoches, w0, c='dodgerblue', label='w0')
mp.legend()
mp.subplot(312)
mp.ylabel('w1', fontsize=14)
mp.gca().xaxis.set_major_locator(mp.MultipleLocator(100))
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.plot(epoches, w1, c='limegreen', label='w1')
mp.legend()

mp.subplot(313)
mp.xlabel('epoch', fontsize=14)
mp.ylabel('loss', fontsize=14)
mp.gca().xaxis.set_major_locator(mp.MultipleLocator(100))
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
```

```
mp.plot(epoches, losses, c='orangered', label='loss')
mp.legend()
```

1. 基于三维曲面绘制梯度下降过程中的每一个点。

```
import mpl_toolkits.mplot3d as axes3d

grid_w0, grid_w1 = np.meshgrid(
    np.linspace(0, 9, 500),
    np.linspace(0, 3.5, 500))

grid_loss = np.zeros_like(grid_w0)
for x, y in zip(train_x, train_y):
    grid_loss += ((grid_w0 + x*grid_w1 - y) ** 2) / 2

mp.figure('Loss Function')
ax = mp.gca(projection='3d')
mp.title('Loss Function', fontsize=20)
ax.set_xlabel('w0', fontsize=14)
ax.set_ylabel('w1', fontsize=14)
ax.set_zlabel('loss', fontsize=14)
ax.plot_surface(grid_w0, grid_w1, grid_loss, rstride=10, cstride=10,
cmap='jet')
ax.plot(w0, w1, losses, 'o-', c='orangered', label='BGD')
mp.legend()
```

1. 以等高线的方式绘制梯度下降的过程。

```
mp.figure('Batch Gradient Descent', facecolor='lightgray')
mp.title('Batch Gradient Descent', fontsize=20)
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.contourf(grid_w0, grid_w1, grid_loss, 10, cmap='jet')
cntr = mp.contour(grid_w0, grid_w1, grid_loss, 10,
                  colors='black', linewidths=0.5)
mp.clabel(cntr, inline_spacing=0.1, fmt='%.2f',
          fontsize=8)
mp.plot(w0, w1, 'o-', c='orangered', label='BGD')
mp.legend()
mp.show()

```

线性回归相关API：

```
import sklearn.linear_model as lm
# 创建模型
model = lm.LinearRegression()
# 训练模型
# 输入为一个二维数组表示的样本矩阵
# 输出为每个样本最终的结果
model.fit(输入, 输出) # 通过梯度下降法计算模型参数
# 预测输出
# 输入array是一个二维数组，每一行是一个样本，每一列是一个特征。
result = model.predict(array)
```

```
浓度        深度        温度            腐蚀速率
0.001     100       -1              0.0002
0.001     100       -1              0.0002
0.001     100       -1              0.0002
0.001     100       -1              0.0002

0.002     200       -2                ?
0.003     300       -4                ?
```

案例：基于线性回归训练single.txt中的训练样本，使用模型预测测试样本。

```
import numpy as np
import sklearn.linear_model as lm
import matplotlib.pyplot as mp
# 采集数据
x, y = np.loadtxt('../data/single.txt', delimiter=',', usecols=(0,1),
unpack=True)
x = x.reshape(-1, 1)
# 创建模型
model = lm.LinearRegression()  # 线性回归
# 训练模型
model.fit(x, y)
# 根据输入预测输出
pred_y = model.predict(x)
mp.figure('Linear Regression', facecolor='lightgray')
mp.title('Linear Regression', fontsize=20)
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.scatter(x, y, c='dodgerblue', alpha=0.75, s=60, label='Sample')
mp.plot(x, pred_y, c='orangered', label='Regression')
mp.legend()
mp.show()
```

## 评估训练结果误差（metrics）

线性回归模型训练完毕后，可以利用测试集评估训练结果误差。sklearn.metrics提供了计算模型误差的几个常用算法：

```
import sklearn.metrics as sm

# 平均绝对值误差：1/m∑|实际输出-预测输出|
sm.mean_absolute_error(y, pred_y)
# 平均平方误差：1/m∑(实际输出-预测输出)^2
sm.mean_squared_error(y, pred_y)
# 中位绝对值误差：MEDIAN(|实际输出-预测输出|)
sm.median_absolute_error(y, pred_y)
# R2得分，(0,1]区间的分值。分数越高，误差越小。
sm.r2_score(y, pred_y)
```

案例：在上一个案例中使用sm评估模型误差。

```
# 平均绝对值误差：1/mΣ|实际输出-预测输出|
print(sm.mean_absolute_error(y, pred_y))
# 平均平方误差：SQRT(1/mΣ(实际输出-预测输 出)^2)
print(sm.mean_squared_error(y, pred_y))
# 中位绝对值误差：MEDIAN(|实际输出-预测输出|)
print(sm.median_absolute_error(y, pred_y))
# R2得分，(0,1]区间的分值。分数越高，误差越小。
print(sm.r2_score(y, pred_y))
```

### 模型的保存和加载

模型训练是一个耗时的过程，一个优秀的机器学习是非常宝贵的。可以模型保存到磁盘中，也可以在需要使用的时候从磁盘中重新加载模型即可。不需要重新训练。

模型保存和加载相关API：

```
import pickle
pickle.dump(内存对象, 磁盘文件) # 保存模型
model = pickle.load(磁盘文件) # 加载模型
```

案例：把训练好的模型保存到磁盘中。

```
# 将训练好的模型对象保存到磁盘文件中
with open('../../data/linear.pkl', 'wb') as f:
    pickle.dump(model, f)

# 从磁盘文件中加载模型对象
with open('../../data/linear.pkl', 'rb') as f:
    model = pickle.load(f)
# 根据输入预测输出
pred_y = model.predict(x)
```

## 岭回归

普通线性回归模型使用基于梯度下降的最小二乘法，在最小化损失函数的前提下，寻找最优模型参数，于此过程中，包括少数异常样本在内的全部训练数据都会对最终模型参数造成程度相等的影响，异常值对模型所带来影响无法在训练过程中被识别出来。为此，岭回归在模型迭代过程所依据的损失函数中增加了正则项，以限制模型参数对异常样本的匹配程度，进而提高模型面对多数正常样本的拟合精度。

```
import sklearn.linear_model as lm
# 创建模型
model = lm.Ridge(正则强度，fit_intercept=是否训练截距，max_iter=最大迭代次数)
# 训练模型
# 输入为一个二维数组表示的样本矩阵
# 输出为每个样本最终的结果
model.fit(输入，输出)
# 预测输出
# 输入array是一个二维数组，每一行是一个样本，每一列是一个特征。
result = model.predict(array)
```

案例：加载abnormal.txt文件中的数据，基于岭回归算法训练回归模型。

```python
import numpy as np
import sklearn.linear_model as lm
import matplotlib.pyplot as mp
# 采集数据
x, y = np.loadtxt('../data/single.txt', delimiter=',', usecols=(0,1),
unpack=True)
x = x.reshape(-1, 1)
# 创建线性回归模型
model = lm.LinearRegression()
# 训练模型
model.fit(x, y)
# 根据输入预测输出
pred_y1 = model.predict(x)
# 创建岭回归模型
model = lm.Ridge(150, fit_intercept=True, max_iter=10000)
# 训练模型
model.fit(x, y)
# 根据输入预测输出
pred_y2 = model.predict(x)

mp.figure('Linear & Ridge', facecolor='lightgray')
mp.title('Linear & Ridge', fontsize=20)
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.scatter(x, y, c='dodgerblue', alpha=0.75,
           s=60, label='Sample')
sorted_indices = x.T[0].argsort()
mp.plot(x[sorted_indices], pred_y1[sorted_indices],
        c='orangered', label='Linear')
mp.plot(x[sorted_indices], pred_y2[sorted_indices],
        c='limegreen', label='Ridge')
mp.legend()
mp.show()
```

## 多项式回归

```
浓度      深度      温度          腐蚀速率
0.001    100      -1           0.0002
0.001    100      -1           0.0002
0.001    100      -1           0.0002
0.001    100      -1           0.0002

0.002    200      -2             ?
0.003    300      -4             ?
```

若希望回归模型更好的拟合训练样本数据，可以使用多项式回归器。

一元多项式回归

$y=w_0 + w_1 x + w_2 x^2 + w_3 x^3 + ... + w_d x^d$

将高次项看做对一次项特征的扩展得到：

$y=w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + ... + w_d x_d$

那么一元多项式回归即可以看做为多元线性回归，可以使用LinearRegression模型对样本数据进行模型训练。

所以一元多项式回归的实现需要两个步骤：

1. 将一元多项式回归问题转换为多元线性回归问题（只需给出多项式最高次数即可）。
2. 将1步骤得到多项式的结果中 $w_1$ $w_2$ .. 当做样本特征，交给线性回归器训练多元线性模型。

使用sklearn提供的**数据管线**实现两个步骤的顺序执行：

```
import sklearn.pipeline as pl
import sklearn.preprocessing as sp
import sklearn.linear_model as lm

model = pl.make_pipeline(
    sp.PolynomialFeatures(10),  # 多项式特征扩展器
    lm.LinearRegression())      # 线性回归器
```

案例：

```
import numpy as np
import sklearn.pipeline as pl
import sklearn.preprocessing as sp
import sklearn.linear_model as lm
import sklearn.metrics as sm
import matplotlib.pyplot as mp
# 采集数据
x, y = np.loadtxt('../data/single.txt', delimiter=',', usecols=(0,1), unpack=True)
x = x.reshape(-1, 1)
# 创建模型(管线)
model = pl.make_pipeline(
    sp.PolynomialFeatures(10),  # 多项式特征扩展器
    lm.LinearRegression())      # 线性回归器
# 训练模型
model.fit(x, y)
# 根据输入预测输出
pred_y = model.predict(x)
test_x = np.linspace(x.min(), x.max(), 1000).reshape(-1, 1)
pred_test_y = model.predict(test_x)
mp.figure('Polynomial Regression', facecolor='lightgray')
mp.title('Polynomial Regression', fontsize=20)
mp.xlabel('x', fontsize=14)
mp.ylabel('y', fontsize=14)
mp.tick_params(labelsize=10)
mp.grid(linestyle=':')
mp.scatter(x, y, c='dodgerblue', alpha=0.75, s=60, label='Sample')
mp.plot(test_x, pred_test_y, c='orangered', label='Regression')
mp.legend()
mp.show()
```

过于简单的模型，无论对于训练数据还是测试数据都无法给出足够高的预测精度，这种现象叫做欠拟合。

过于复杂的模型，对于训练数据可以得到较高的预测精度，但对于测试数据通常精度较低，这种现象叫做过拟合。

一个性能可以接受的学习模型应该对训练数据和测试数据都有接近的预测精度，而且精度不能太低。

```
训练集R2    测试集R2
0.3         0.4         欠拟合：过于简单，无法反映数据的规则
0.9         0.2         过拟合：过于复杂，太特殊，缺乏一般性
0.7         0.6         可接受：复杂度适中，既反映数据的规则，同时又不失一般性

```

# 决策树

## 基本算法原理

核心思想：相似的输入必会产生相似的输出。例如预测某人薪资：

年龄：1-青年，2-中年，3-老年
学历：1-本科，2-硕士，3-博士
经历：1-出道，2-一般，3-老手，4-骨灰
性别：1-男性，2-女性

| 年龄 | 学历 | 经历 | 性别 | ==> | 薪资 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | ==> | 6000（低） |
| 2 | 1 | 3 | 1 | ==> | 10000（中） |
| 3 | 3 | 4 | 1 | ==> | 50000（高） |
| ... | ... | ... | ... | ==> | ... |
| 1 | 3 | 2 | 2 | ==> | ? |

为了提高搜索效率，使用树形数据结构处理样本数据：

$$\text{年龄}=1\begin{cases}\text{学历}1\\ \text{学历}2\\ \text{学历}3\end{cases}\quad \text{年龄}=2\begin{cases}\text{学历}1\\ \text{学历}2\\ \text{学历}3\end{cases}\quad \text{年龄}=3\begin{cases}\text{学历}1\\ \text{学历}2\\ \text{学历}3\end{cases}$$

首先从训练样本矩阵中选择第一个特征进行子表划分，使每个子表中该特征的值全部相同，然后再在每个子表中选择下一个特征按照同样的规则继续划分更小的子表，不断重复直到所有的特征全部使用完为止，此时便得到叶级子表，其中所有样本的特征值全部相同。对于待预测样本，根据其每一个特征的值，选择对应的子表，逐一匹配，直到找到与之完全匹配的叶级子表，用该子表中样本的输出，通过平均(回归)或者投票(分类)为待预测样本提供输出。

随着子表的划分，信息熵（信息的混乱程度）越来越小，信息越来越纯，数据越来越有序。

决策树回归器模型相关API：

```python
import sklearn.tree as st

# 创建决策树回归器模型   决策树的最大深度为4
model = st.DecisionTreeRegressor(max_depth=4)
# 训练模型
# train_x:  二维数组样本数据
# train_y:  训练集中对应每行样本的结果
model.fit(train_x, train_y)
# 测试模型
pred_test_y = model.predict(test_x)
```

案例：预测波士顿地区房屋价格。

1. 读取数据，打断原始数据集。 划分训练集和测试集。

```
import sklearn.datasets as sd
import sklearn.utils as su
# 加载波士顿地区房价数据集
boston = sd.load_boston()
print(boston.feature_names)
# |CRIM|ZN|INDUS|CHAS|NOX|RM|AGE|DIS|RAD|TAX|PTRATIO|B|LSTAT|
# 犯罪率|住宅用地比例|商业用地比例|是否靠河|空气质量|房间数|年限|距中心区距离|路网密度|房产税|师生比|黑人比例|低地位人口比例|
# 打乱原始数据集的输入和输出
x, y = su.shuffle(boston.data, boston.target, random_state=7)
# 划分训练集和测试集
train_size = int(len(x) * 0.8)
train_x, test_x, train_y, test_y = \
    x[:train_size], x[train_size:], \
    y[:train_size], y[train_size:]
```

1. 创建决策树回归器模型，使用训练集训练模型。使用测试集测试模型。

```
import sklearn.tree as st
import sklearn.metrics as sm

# 创建决策树回归模型
model = st.DecisionTreeRegressor(max_depth=4)
# 训练模型
model.fit(train_x, train_y)
# 测试模型
pred_test_y = model.predict(test_x)
print(sm.r2_score(test_y, pred_test_y))
```