实验四：线性回归模型

本次实验旨在让同学们了解线性回归模型相关知识，学习scikit-learn机器学习库进行回归的基本使用。

内容

1. 一元线性回归与多元线性回归

- 1.1 回归分析的概念
- 1.2 一元线性回归模型
- 1.3 多元线性回归模型

2. 线性回归求解

- 2.1 最小二乘法
- 2.2 梯度下降法
- 2.3 两种方法对比

3. 线性回归在sklearn中的基本实践

- 3.1 模型调用与训练
- 3.2 模型评估
- 3.3 数据标准化

4. 动手实践

1. 一元线性回归与多元线性回归

1.1 回归分析的概念

通过大量实验或观测数据，用统计方法寻找变量之间的关系和统计规律，回归分析(regression analysis)就是研究这类规律的方法。

1.2 一元线性回归模型

一元线性回归模型也称为简单线性回归模型，这里的'一元'指只有一个自变量X，'线性'表示因变量Y与X之间是一种直线关系。例如，通过工作年限（一个特征变量）对收入进行预测，就属于一元线性回归。其形式可以通过如下公式表达：

$$y=\beta_0+\beta_1 x+\epsilon$$

其中：

- y 是因变量（或被预测的变量）。
- x 是自变量（或预测变量）。
- β_0 是截距。
- β_1 是斜率。
- ϵ 是误差项，表示除了x以外可能影响y的其他因素。

一元线性回归的目标是找到最佳的β_0和β_1值，使得由此得到的预测线尽可能接近实际观测到的数据点。

1.3 多元线性回归模型

多元线性回归比一元线性回归复杂，拥有两个或更多自变量（X）。在多元线性回归模型中，我们试图在多维空间中找到一个超平面，数据点散落在超平面的两侧。多元线性回归模型可以用下面的方程来表示：

$$\begin{aligned}Y&=\beta_0+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_k X_k+\epsilon\\&=W\cdot X+\epsilon\end{aligned}$$

其中：

- Y 是因变量特征矩阵。
- X 是自变量特征矩阵。
- W 是系数矩阵。
- ϵ 是误差项，它代表了除自变量以外可能影响因变量的其他因素。

在多元线性回归中，我们的目标是估计模型中的系数（W值），以便模型能够最好地拟合数据。这些系数可以通过最小化误差项的平方和（即最小二乘法）或导数方法（如梯度下降法、牛顿法、拟牛顿法、共轭梯度法）来估计。

2. 线性回归求解

2.1 最小二乘法

对于一元线性回归模型，为了找到最佳的β_0和β_1值，可以使用最小二乘法，使得所有数据点到回归线的垂直距离的平方和最小，即误差平方和SSE：

$$SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

其中，y_i 是第i个观测值，$\hat{y}_i$ 是模型预测的第i个值，即 $\hat{y}_i = \beta_0 + \beta_1 x_i$。

为什么要平方误差呢？

是因为预测值可能大于也可能小于实际值，从而分别产生负或正的误差。如果没有平方误差值，误差的数值可能会因为正负误差相消而变小，而并非因为模型拟合好。此外，平方误差会加大误差值，所以最小化平方误差可以保证模型更好。

为了最小化SSE，我们对β_0 和 β_1 进行偏导，并令这些导数等于零。这会得到两个方程：

1. 关于 β_0 的偏导数：

$$\frac{\partial SSE}{\partial \beta_0} = -2\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i) = 0$$

2. 关于 β_1 的偏导数：

$$\frac{\partial SSE}{\partial \beta_1} = -2\sum_{i=1}^{n}x_i(y_i - \beta_0 - \beta_1 x_i) = 0$$

通过解这两个方程，我们可以得到β_0和β_1的最优解，其中，$\bar{x}$ 和 $\bar{y}$ 分别是x和y的均值：

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

$$\begin{aligned}\beta_1 &= \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2} \\ &= \frac{\sum_{i=1}^{n} x_i y_i - \bar{y}\sum_{i=1}^{n} x_i - \bar{x}\sum_{i=1}^{n} y_i + n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - 2\bar{x}\sum_{i=1}^{n} x_i + n\bar{x}^2} \\ &= \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2} \quad (\sum_{i=1}^{n} x_i = n\bar{x}, \sum_{i=1}^{n} y_i = n\bar{y})\end{aligned}$$

推广到多元线性回归，误差平方和可以表示为：

$$L(w) = (Y - XW)^T (Y - XW)$$

解这个方程，我们得到 W 的最优估计：

$$\frac{\partial L(w)}{\partial w} = 2X^T(XW - Y) = 0$$

$$W = (X^T X)^{-1} X^T Y$$

2.2 梯度下降法

梯度下降法是一个迭代算法，这种方法的关键在于逐步调整模型参数以最小化成本函数（如均方误差），以下是算法的基本流程：

1. **初始化参数,选择学习率**：初始化参数w、b，设定学习率（α）、迭代次数、最小误差参数。
2. **计算损失函数的梯度**：在当前参数的位置，计算损失函数（如均方误差）相对于每个参数的梯度。梯度是损失函数在当前参数点的斜率，指向损失增加的方向。

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

3. **更新参数**：根据梯度和学习率更新参数。参数应沿着梯度的反方向调整，因为我们的目标是最小化损失函数。

$$w = w - \alpha \frac{\partial}{\partial w} MSE$$

$$b = b - \alpha \frac{\partial}{\partial b} MSE$$

其中，α 是学习率，控制参数在梯度下降过程中更新的步长。

4. **重复迭代**：重复步骤3和4，直到满足停止条件。停止条件可以是“迭代后的误差小于最小误差”（表示接近极小值点），或“迭代达到指定次数”，或者“成本函数的改善低于某个阈值”。

求解示例：

In [136…

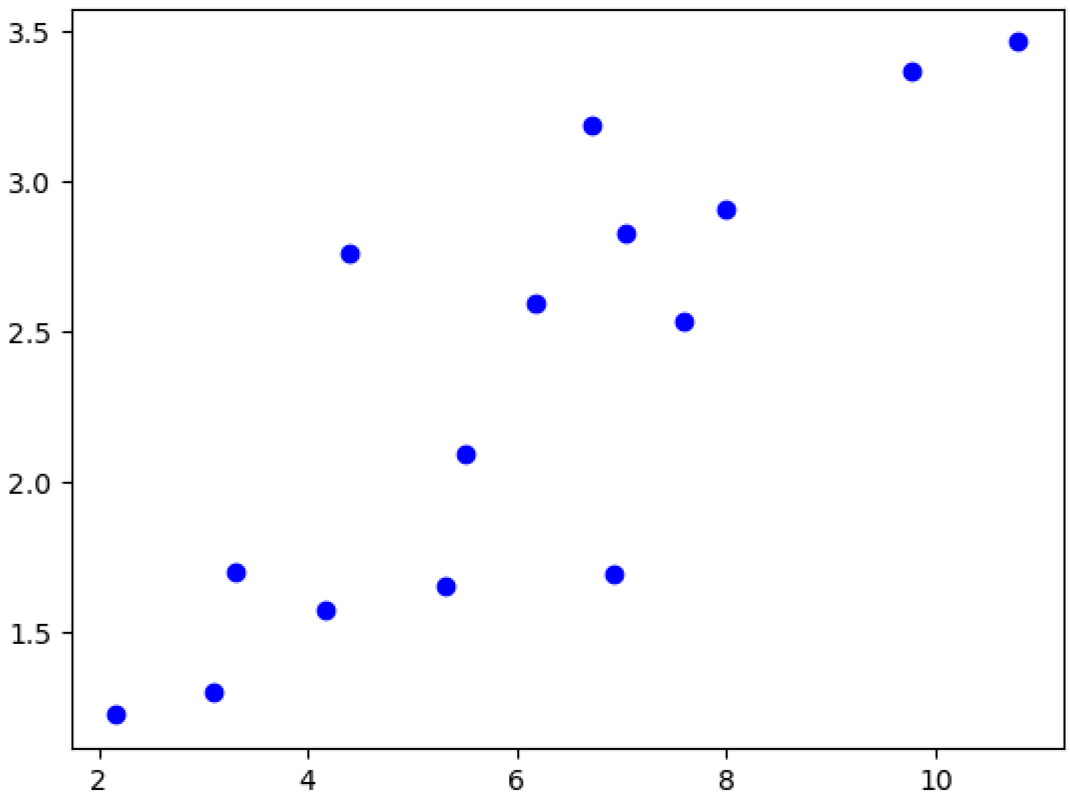
```
import torch
from torch.autograd import Variable
torch.manual_seed(43)
x_train = np.array([[3.3], [4.4], [5.5], [6.71], [6.93], [4.168],
                    [9.779], [6.182], [7.59], [2.167], [7.042],
                    [10.791], [5.313], [7.997], [3.1]], dtype=np.float32)

y_train = np.array([[1.7], [2.76], [2.09], [3.19], [1.694], [1.573],
                    [3.366], [2.596], [2.53], [1.221], [2.827],
                    [3.465], [1.65], [2.904], [1.3]], dtype=np.float32)

# 画出图像
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x_train, y_train, 'bo')
```

Out[136…

```
[<matplotlib.lines.Line2D at 0x284fe7970>]
```



In [137…

```
# 定义参数 w 和 b
w = Variable(torch.randn(1), requires_grad=True) # 随机初始化
b = Variable(torch.zeros(1), requires_grad=True) # 使用 0 进行初始化

# 定义学习率a
a = 0.002

# 构建一元线性回归模型
def linear_model(x):
    return x * w + b

# 输入数据转换成 Tensor  再转换为Variable
x_train = torch.from_numpy(x_train)
y_train = torch.from_numpy(y_train)
x_train = Variable(x_train)
```

```
y_train = Variable(y_train)

#构建线性回归模型
y_ = linear_model(x_train)
```

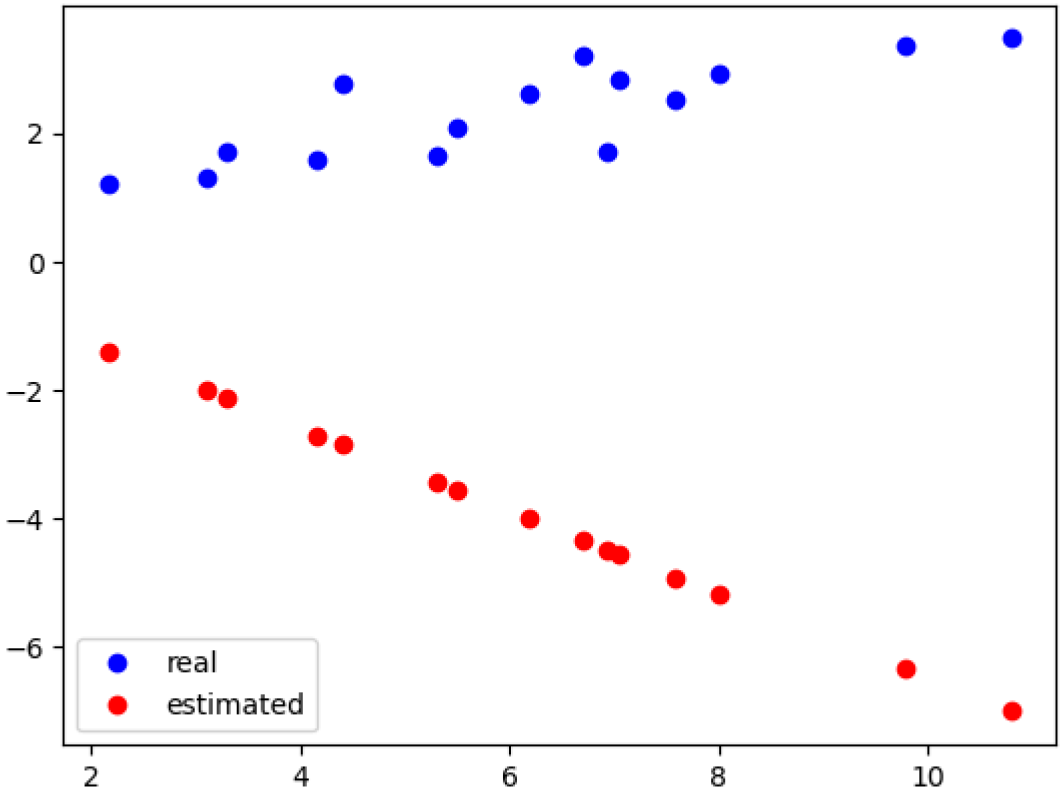
没有使用梯度下降算法之前的数据分布：

In [138…

```
plt.plot(x_train.data.numpy(), y_train.data.numpy(), 'bo', label='real')
plt.plot(x_train.data.numpy(), y_.data.numpy(), 'ro', label='estimated')
plt.legend()
```

Out[138…

```
<matplotlib.legend.Legend at 0x28503ee30>
```



计算此时的损失：

In [139...

```
#误差函数
def get_loss(y_, y):
    return torch.mean((y_ - y_train) ** 2)

loss = get_loss(y_, y_train)
print(loss)
```

```
tensor(43.8924, grad_fn=<MeanBackward0>)
```

In [140...

```
#计算 w 和 b 的梯度
loss.backward()
# 查看 w 和 b 的梯度
print(w.grad)
print(b.grad)
```

```
tensor([-85.9665])
tensor([-12.5134])
```

In [141...

```
# 更新一次参数
w.data = w.data - a * w.grad.data
b.data = b.data - a * b.grad.data
```

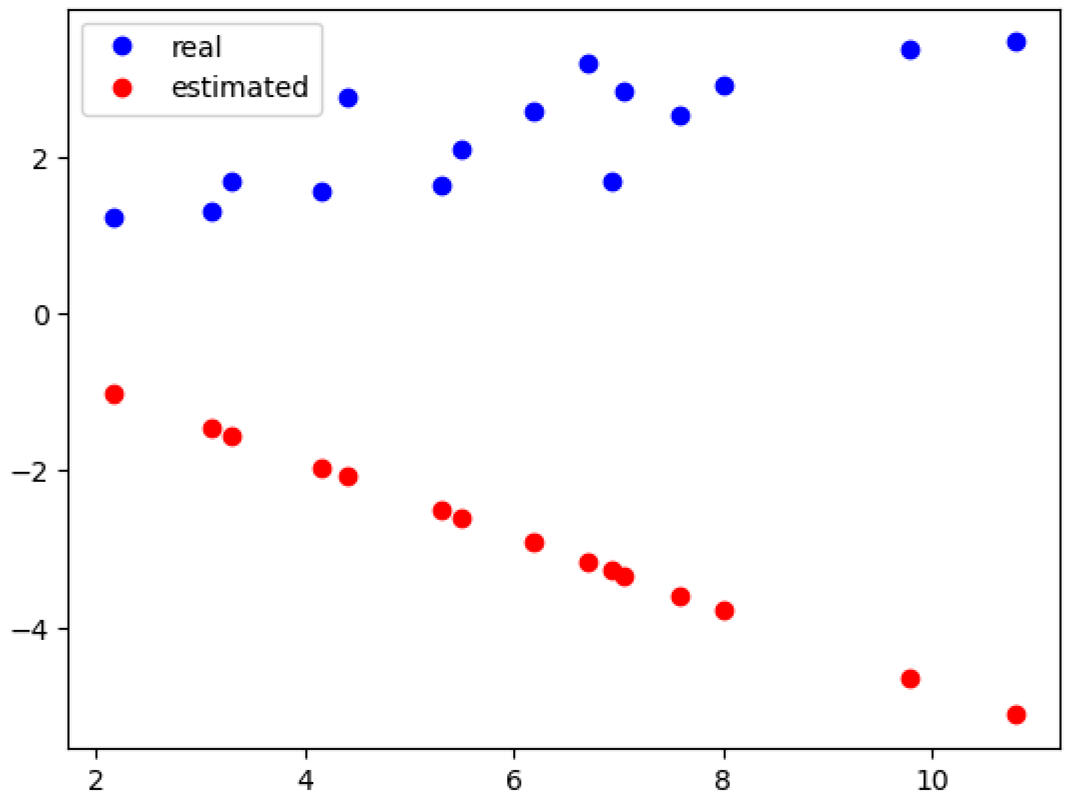
完成一次参数更新参数之后，我们再一次看看模型输出的结果：

In [142...

```
y_ = linear_model(x_train)
plt.plot(x_train.data.numpy(), y_train.data.numpy(), 'bo', label='real')
plt.plot(x_train.data.numpy(), y_.data.numpy(), 'ro', label='estimated')
plt.legend()
```

Out[142...

```
<matplotlib.legend.Legend at 0x2850ba4a0>
```



In [143…

```
#进行30次参数更新
for e in range(30):
    y_ = linear_model(x_train)
    loss = get_loss(y_, y_train)

    w.grad.zero_() # 记得归零梯度
    b.grad.zero_() # 记得归零梯度
    loss.backward()

    w.data = w.data - a * w.grad.data # 更新 w
    b.data = b.data - a * b.grad.data # 更新 b
    print('epoch: {}, loss: {}'.format(e, loss.item()))
```

```
epoch: 0, loss: 30.102800369262695
epoch: 1, loss: 20.666921615600586
epoch: 2, loss: 14.210190773010254
epoch: 3, loss: 9.792009353637695
epoch: 4, loss: 6.768754005432129
epoch: 5, loss: 4.70000696182251
epoch: 6, loss: 3.2844033241271973
epoch: 7, loss: 2.3157293796539307
epoch: 8, loss: 1.6528764963150024
epoch: 9, loss: 1.199289083480835
epoch: 10, loss: 0.8888960480690002
epoch: 11, loss: 0.676487147808075
epoch: 12, loss: 0.5311263203620911
epoch: 13, loss: 0.43164440989494324
epoch: 14, loss: 0.3635564148426056
epoch: 15, loss: 0.3169504702091217
epoch: 16, loss: 0.28504398465156555
epoch: 17, loss: 0.26319608092308044
epoch: 18, loss: 0.2482309192419052
epoch: 19, loss: 0.23797544836997986
epoch: 20, loss: 0.23094269633293152
epoch: 21, loss: 0.2261151820421219
epoch: 22, loss: 0.2227967530488968
epoch: 23, loss: 0.22051087021827698
epoch: 24, loss: 0.21893160045146942
epoch: 25, loss: 0.21783573925495148
epoch: 26, loss: 0.21707089245319366
epoch: 27, loss: 0.2165324091911316
epoch: 28, loss: 0.21614889800548553
epoch: 29, loss: 0.2158713936805725
```

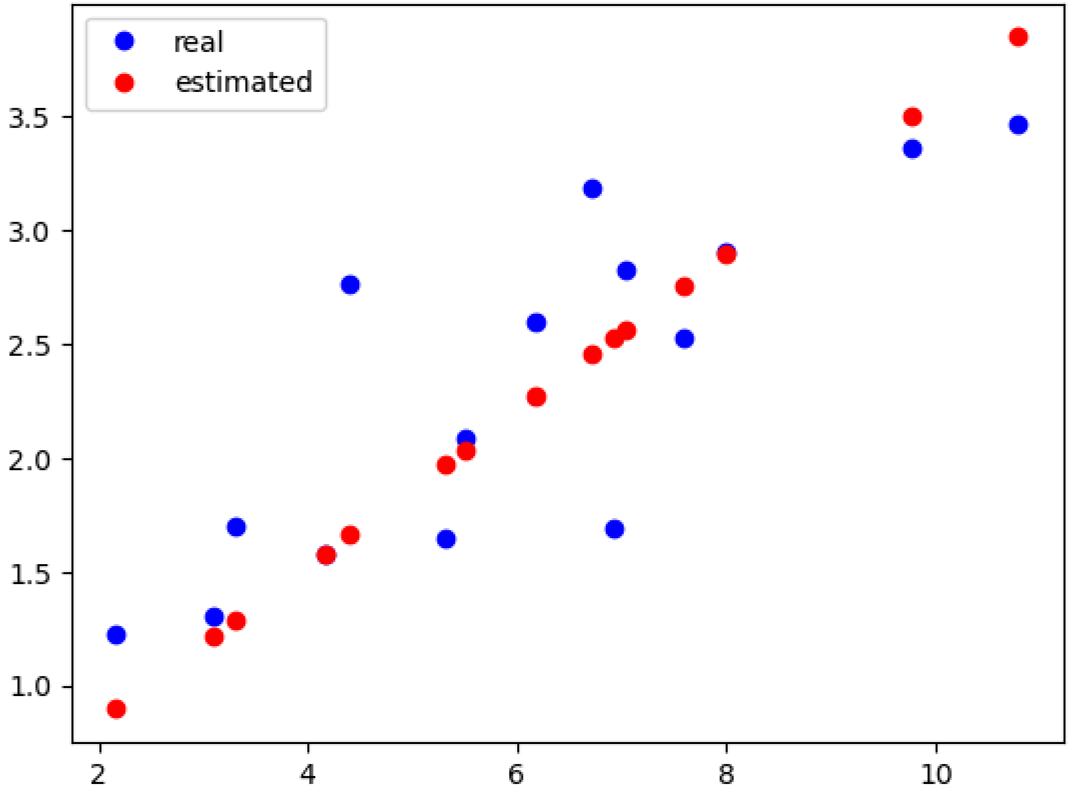
查看迭代30次后模型的预测结果:

In [144…

```
y_ = linear_model(x_train)
plt.plot(x_train.data.numpy(), y_train.data.numpy(), 'bo', label='real')
plt.plot(x_train.data.numpy(), y_.data.numpy(), 'ro', label='estimated')
plt.legend()
```

Out[144…

```
<matplotlib.legend.Legend at 0x28511afe0>
```



经过30次更新，我们发现红色的预测结果已经比较好的拟合了蓝色的真实值。

2.3 两种方法对比

相同点:

- 本质和目标相同： 二者都是经典的学习算法，在给定已知数据的前提下利用求导算出一个模型（函数），使得损失函数值最小，后对给定的新数据进行估算预测。

不同点:

- 损失函数不同： 梯度下降可以选取其它损失函数；而最小二乘法一定是平方损失函数。

- 实现方法不同： 最小二乘法是直接求导找出全局最小，一次运算得出结果；而梯度下降是一种迭代法，需要选择学习率，通过多次迭代得到结果。

- 效果不同： 最小二乘法一定是全局最小，但计算繁琐，且复杂情况下未必有解；梯度下降迭代计算简单，但找到的一般是局部最小，只有在目标函数是凸函数时才是全局最小，到最小点附近收敛速度会变慢，且对初始点的选择极为敏感。

- 适用范围不同： 最小二乘法只适用于线性模型，不适合逻辑回归模型等其他模型；梯度下降法可以适用于各种类型的模型，且当特征数较大时也能适应良好（适合大规模数据）。

3. 线性回归在sklearn中的基本实践

引入Scikit-learn库便可快速搭建线性回归模型。

- 使用最小二乘法的线性回归：

In [ ]:
```
from sklearn.linear_model import LinearRegression
```

- 使用梯度下降法的线性回归：

In [ ]:
```
from sklearn.linear_model import SGDRegressor
```

3.1 模型调用与训练

如下是一个调用sklearn实现多元线性回归对房价进行预测的例子：

In [46]:
```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression #导入线性回归模型
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')
```

```python
# 加载波士顿房价数据集
"""
数据集中有506个样本，包含13个输入特征以及1个学习目标（房价中位数）
1- CRIM      犯罪率; per capita crime rate by town
2- ZN        proportion of residential land zoned for lots over 25,000 sq.ft.
3- INDUS     非零售商业用地占比; proportion of non-retail business acres per town
4- CHAS      是否临Charles河; Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5- NOX       氮氧化物浓度; nitric oxides concentration (parts per 10 million)
6- RM        房屋房间数; average number of rooms per dwelling
7- AGE       房屋年龄; proportion of owner-occupied units built prior to 1940
8- DIS       和就业中心的距离; weighted distances to five Boston employment centres
9- RAD       是否容易上高速路; index of accessibility to radial highways
10- TAX       税率; full-value property-tax rate per $10,000
11- PTRATIO   学生人数比老师人数; pupil-teacher ratio by town
12- B         城镇黑人比例计算的统计值; 1000(Bk - 0.63)^2 where Bk is the proportion of black people by town
13- LSTAT     低收入人群比例; % lower status of the population
14- MEDV      房价中位数; Median value of owner-occupied homes in $1000's
"""
data_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data"
raw_df = pd.read_csv(data_url, sep='\s+',  header=None)
raw_df.columns = ["CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE", "DIS",
                  "RAD", "TAX", "PTRATIO", "B", "LSTAT", "MEDV"]
raw_df.head(5) #显示前5行数据
```

Out[46]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

可以使用 `sklearn.model_selection` 中的 `train_test_split` 进行数据集划分。 `random_state` 表示随机种子,随机数种子控制每次划分训练集和测试集的模式，其取值不变时划分得到的结果一模一样，其值改变时，划分得到的结果不同。若不设置此参数，则函数会自动选择一种随机模式，得到的结果也就不同。 `test_size` 表示分割后测试集占完整数据集的比例，默认为0.25。

In [47]:
```
# 数据集划分
data = raw_df.values[:, :13] #将506个样本13个特征组成的矩阵赋值给data
target = raw_df.values[:, 13] #将506个样本1个预测目标值组成的矩阵赋值给target
X_train, X_test, y_train, y_test = train_test_split(data, target, test_size=0.5, random_state=1)

print(X_train.shape)
print(X_test.shape)
```

```
(253, 13)
(253, 13)
```

In [48]:
```
# 加载线性回归模型
lr = LinearRegression()
# 将训练数据传入开始训练
lr.fit(X_train, y_train)

print(lr.coef_)#系数w
print(lr.intercept_)#截距b
```

```
[-1.03895378e-01  6.56815411e-02 -9.88784599e-03  1.44988900e+00
 -1.72371494e+01  3.31332604e+00  1.08945012e-02 -1.37553794e+00
  3.23677422e-01 -1.20132483e-02 -8.20440741e-01  8.69013924e-03
 -5.28748376e-01]
36.0506458446597
```

3.2 模型评估

回归常用评价指标：

- (1)平均绝对误差(Mean Absolute Error, MAE)
- (2)均方误差(Mean Squared Error, MSE)
- (3)均方根误差(Root Mean Squared Error, RMSE)

In [49]:
```
from sklearn import metrics #引入sklearn模型评价工具库
y_pred = lr.predict(X_test) #预测测试集的房价

MyScore = np.sqrt(metrics.mean_squared_error(y_test, y_pred)) #以RMSE作为评估指标得分MyScore
print('RMSE:',MyScore)
```

```
RMSE: 4.779046666296571
```

- (4)模型自带评估模块: 在Scikit-learn中，回归模型的性能分数可以通过 `score` 方法得到，其计算的是回归平方和与总平方和之比，得分越接近1，拟合程度越好。官方解释

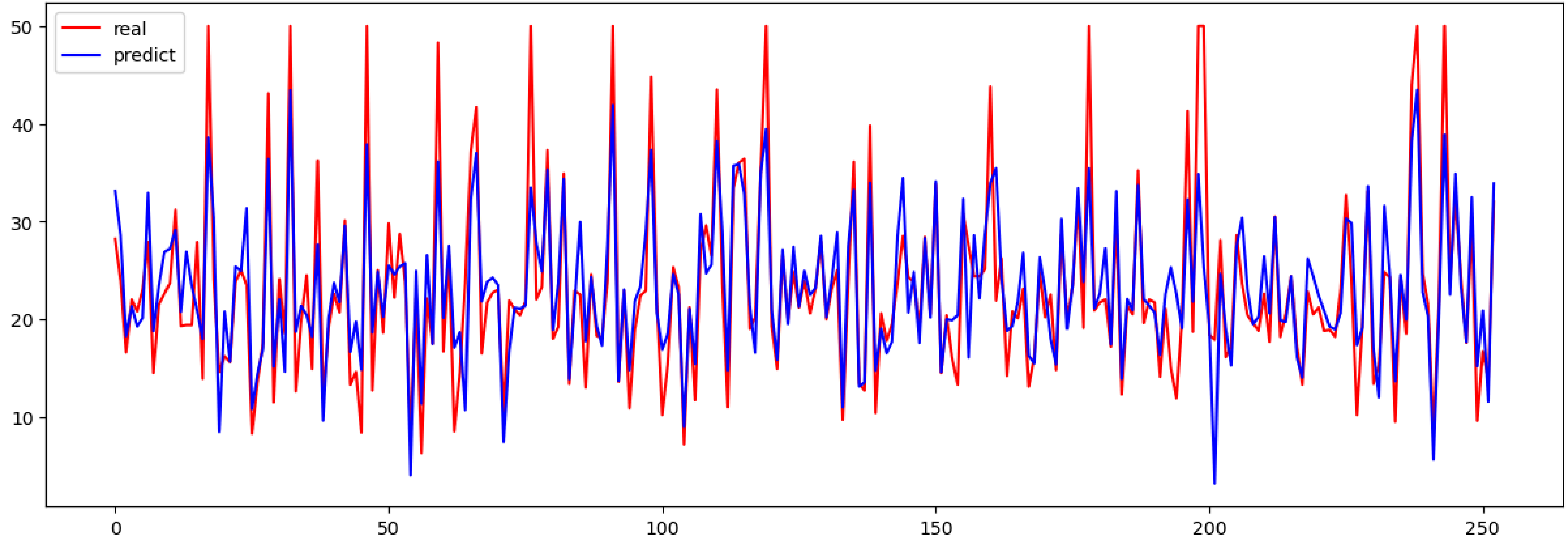
In [37]:
```python
print ('The value of default measurement of LineaRegression is', lr.score(X_test, y_test))
```

```
The value of default measurement of LineaRegression is 0.7397314185094669
```

可以看到均方差根RMSE有4.779，说明拟合度还不够高，我们通过可视化来看一下训练后的预测和真实值之间的差异:

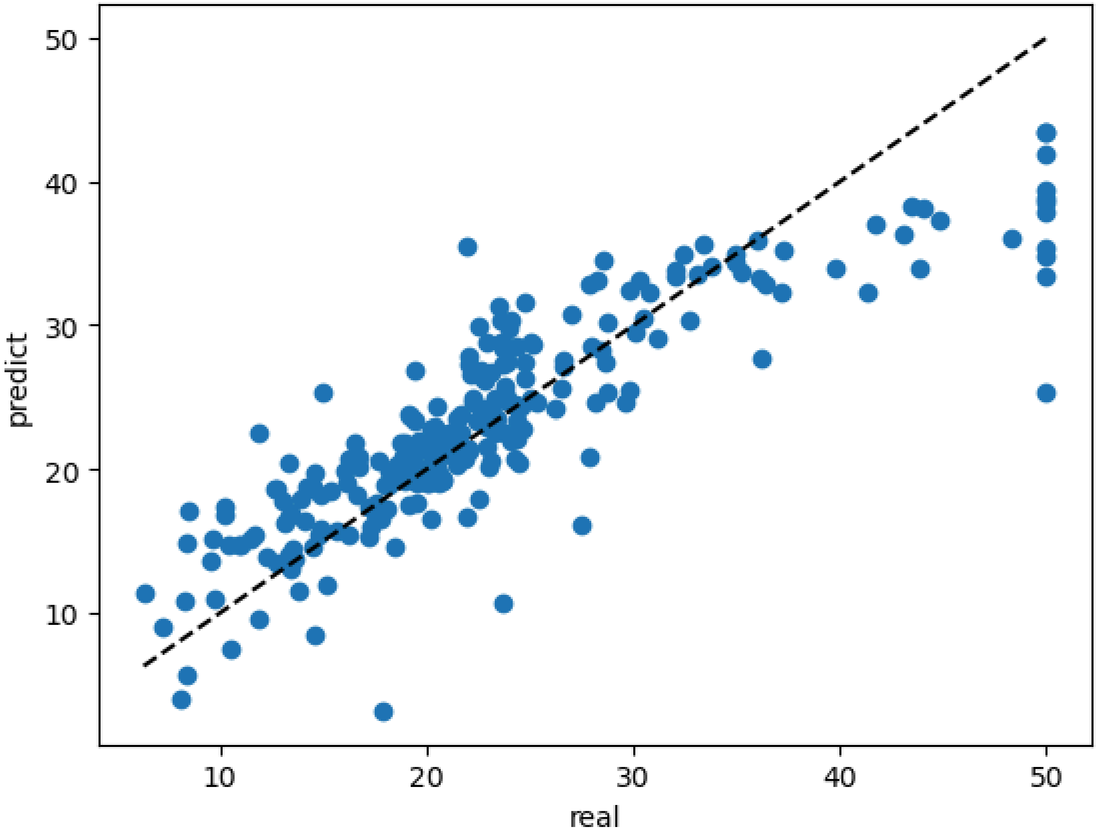
In [38]:
```python
import matplotlib.pyplot as plt
plt.figure(figsize=(15,5))
plt.plot(range(len(y_test)), y_test, 'r', label='real')
plt.plot(range(len(y_test)), y_pred, 'b', label='predict')
plt.legend()
```

Out[38]:
```
<matplotlib.legend.Legend at 0x1451ea860>
```



In [39]:
```python
def plot_LR(y_test, y_pred):
    plt.figure()
    plt.scatter(y_test, y_pred)
    plt.plot([y_test.min(),y_test.max()], [y_test.min(),y_test.max()], 'k--')
    plt.xlabel('real')
    plt.ylabel('predict')

plot_LR(y_test, y_pred)
```



从折线图可以看到许多地方红线会明显超出蓝线，说明模型的拟合度的确不够。从散点图中可以发现，y_test=50的地方有较多的异常值，而线性回归模型的一大缺点就是对异常值很敏感，会极大影响模型的准确性，因此，我们可以根据这一点对模型进行简单优化：

In [50]:

```python
# 去除MEDV=50的异常值
drop_index = raw_df[raw_df['MEDV']==50].index.values
raw_df = raw_df.drop(drop_index)
data = raw_df.values[:, :13]
target = raw_df.values[:, 13]
X_train, X_test, y_train, y_test = train_test_split(data, target, test_size=0.5, random_state=1)
# 对训练集进行训练
lr = LinearRegression()
lr.fit(X_train, y_train)
# 对测试集进行预测
y_pred = lr.predict(X_test)
RMSE = np.sqrt(metrics.mean_squared_error(y_test, y_pred))
print('RMSE:',RMSE)
print ('LineaRegression Score:', lr.score(X_test, y_test))
```

```
RMSE: 3.5286847015366956
LineaRegression Score: 0.7830160006533317
```

可以看到RMSE变成了3.528，模型预测的准确率提高了。

3.3 数据标准化

接下来，让我们试着调用 `SGDRegressor` 使用梯度下降法进行回归：

In [51]:

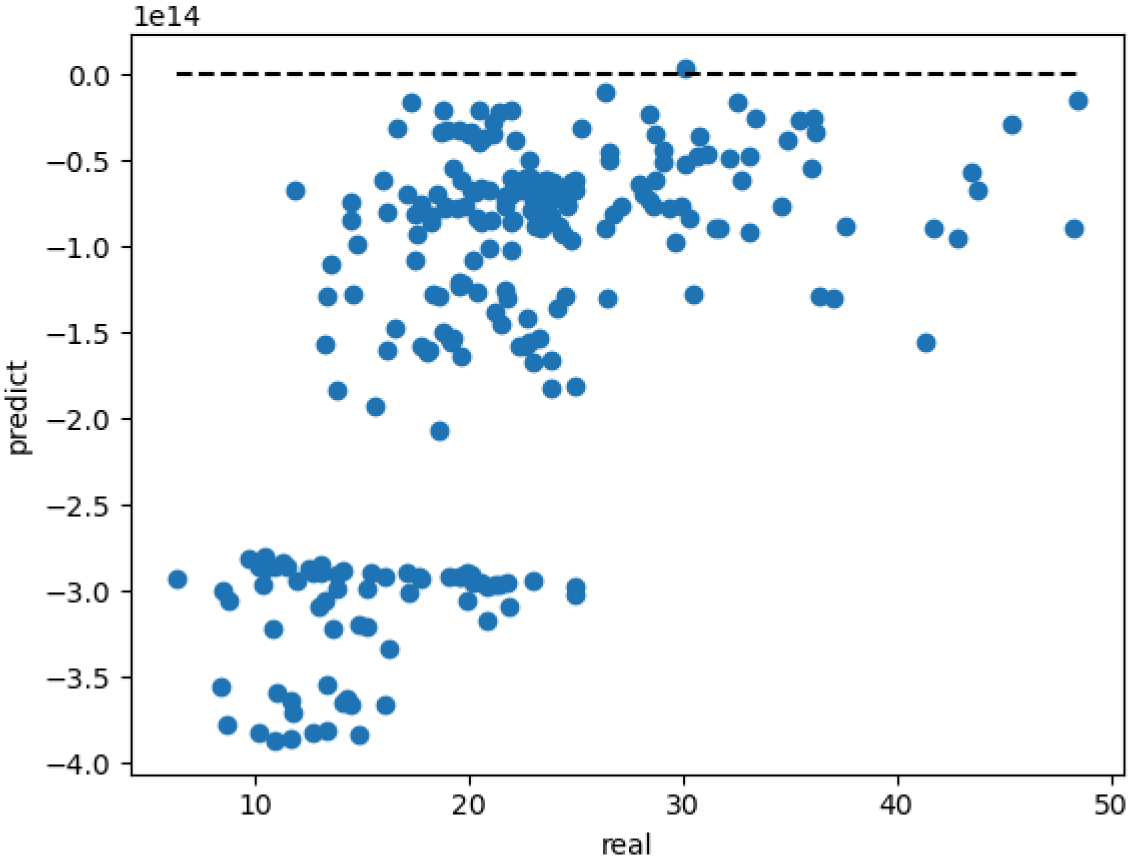
```python
from sklearn.linear_model import SGDRegressor
# 加载模型
lr2 = SGDRegressor(loss='squared_error', alpha=0.0001)
# 将训练数据传入开始训练
lr2.fit(X_train, y_train)

y_predSGD = lr2.predict(X_test)
RMSE2 = np.sqrt(metrics.mean_squared_error(y_test, y_predSGD))
print('RMSE_SGD:',RMSE2)
print('SGDRegressor Score:', lr2.score(X_test, y_test))# 使用SGDRegressor自带的评估模块计算得分
```

```
RMSE_SGD: 183868057462908.78
SGDRegressor Score: -5.8913466302363934e+26
```

结果发现,使用梯度下降法的回归模型的RMSE很高。

In [52]:
```
plot_LR(y_test, y_predSGD)
```



从图中也可看出，模型拟合效果很差。

这是因为我们使用的是原始数据，而RMSE容易受到因变量（目标）和自变量（特征）量纲大小的影响，对于不同量纲大小的数据，模型得到的RMSE可能相差很大。同理，在梯度下降方法中，这会影响损失函数的优化。因此我们需要对数据进行标准化/归一化处理。

可以通过 `sklearn.preprocessing` 中的 `StandardScaler` 对数据进行标准化处理。 `StandardScaler` 是一种常用的数据标准化方法，用于将数据转换为均值为 0，标准差为 1 的标准正态分布。

In [53]:

```
from sklearn.preprocessing import StandardScaler # 导入数据标准化模块
# 分别初始化对特征和目标值的标准化器
ss_X = StandardScaler()
ss_y = StandardScaler()

# 分别对训练和测试数据的特征以及目标值进行标准化处理
X_train_s = ss_X.fit_transform(X_train)
X_test_s = ss_X.transform(X_test)

y_train_s = ss_y.fit_transform(y_train.reshape(-1, 1))
y_test_s = ss_y.transform(y_test.reshape(-1, 1))
```

使用 `fit_transform` 方法计算均值和标准差，并将数据标准化为标准正态分布。 使用 `transform` 方法对数据进行标准化处理。注意，在对'X_test'与'y_test'处理时不使用 `fit_transform` 方法，而是直接使用 `transform` 方法，是因为测试数据是新数据，不能再重新进行均值和方差的计算，而是直接使用在训练数据上得到的均值和方差进行标准化

In [54]:

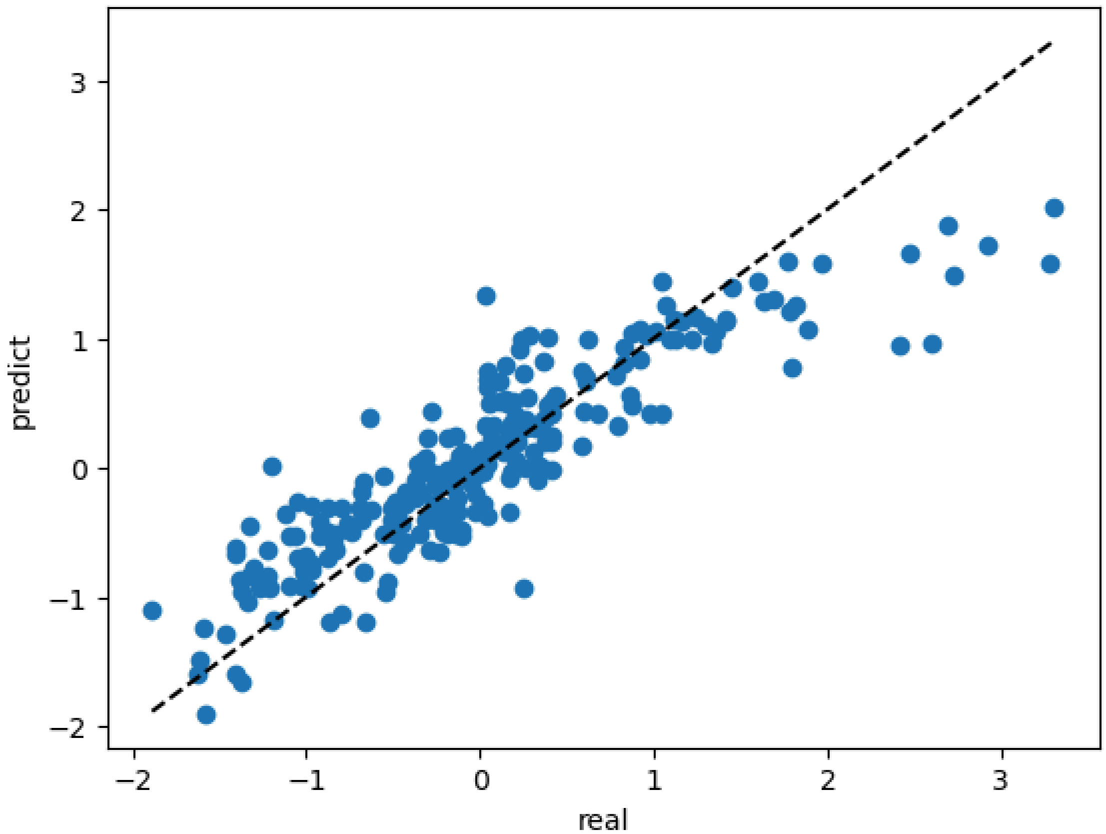
```
# 使用标准化后的数据训练SGDRegressor模型
lr2 = SGDRegressor(loss="squared_error", alpha=0.0001)
lr2.fit(X_train_s, y_train_s)

y_pred_s = lr2.predict(X_test_s)
RMSE2 = np.sqrt(metrics.mean_squared_error(y_test_s, y_pred_s))
print('RMSE_SGD:',RMSE2)
print('SGDRegressor Score:', lr2.score(X_test_s, y_test_s))
```

```
RMSE_SGD: 0.4364353800342414
SGDRegressor Score: 0.7806351141473384
```

In [55]:

```
plot_LR(y_test_s, y_pred_s)
```



如此，使用梯度下降法的回归模型的也能正常求解了。

对于线性模型，标准化后不同维度的特征在数值上更具比较性，使梯度下降过程更加平缓，更易正确的收敛到最优解，提高准确率。

4. 动手实践

- 对2.2中梯度下降方法，调整学习率α 的大小，观察回归效果，讨论学习率对算法的影响。
- 对3.2中的MyScore变量进行修改，分别实现平均绝对误差MAE、均方误差MSE，有想法的同学可以自己设计一种评估指标。
- sklearn中有许多自带的经典数据集，请选择一个线性回归问题的数据集，将样本数据按3:1随机划分成训练集和测试集，分别使用最小二乘法和梯度下降法进行线性回归求解，求出线性回归系数，并**可视化最后的回归结果。**

In [ ]:

```
# sklearn调用数据集
from sklearn.datasets import fetch_california_housing #California房价预测数据集

housing_california = fetch_california_housing() #创建数据集对象
feature = housing_california.feature_names #每个特征的名称
X = housing_california.data #特征数据
Y = housing_california.target #标签数据

from sklearn.datasets import load_diabetes #糖尿病预测数据集
diabetes = load_diabetes()
feature = diabetes.feature_names
X = diabetes.data
Y = diabetes.target

# 新老版本的sklearn可能调用方式不同，报错时具体实现可上sklearn官网查找
```