回归分析

2020 年 3 月 18 日

回归分析是一种预测性的建模技术，它研究的是因变量和自变量之间的关系。通常用于预测分析，时间序列模型以及发现变量之间的因果关系。通常使用曲线来拟合数据点，目标是使曲线到数据点的距离差异最小。

目录

1 对 Housing 数据集进行探索性分析

数据分析的第一步是进行**探索性数据分析 (Exploratory Data Analysis, EDA)**，理解变量的分布与变量之间的关系。

波士顿房价数据

属性:

```
1. CRIM       per capita crime rate by town 城镇人均犯罪率
2. ZN         proportion of residential land zoned for lots over 25,000 sq.ft.
              住宅用地超过 25000 sq.ft. 的比例
3. INDUS      proportion of non-retail business acres per town 非商业用地百分比
4. CHAS       Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
              查尔斯河虚拟变量
5. NOX        nitric oxides concentration (parts per 10 million) 氮氧化物浓度
6. RM         average number of rooms per dwelling 住宅平均房间数目
7. AGE        proportion of owner-occupied units built prior to 1940
              1940年前建成自用房比例
8. DIS        weighted distances to five Boston employment centres
              到波士顿五个就业服务中心的加权距离
9. RAD        index of accessibility to radial highways 高速公路便利指数
10. TAX       full-value property-tax rate per $10,000 每万元不动产税率
11. PTRATIO   pupil-teacher ratio by town 城镇学生老师比（学生/老师）
12. B         1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
              黑人比例指数
13. LSTAT     lower status of the population 底层人口比例
14. MEDV      Median value of owner-occupied homes in $1000's
              自住房的平均价格，以千美元记（要预测的变量）
```

[1]:
```
# 读取数据
import pandas as pd
df = pd.read_csv('data/housing.csv')
df.head()
```

[1]:
```
      CRIM  ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD  TAX  PTRATIO  \
0  0.00632  18   2.31     0  0.538  6.575  65.2  4.0900    1  296       15
1  0.02731   0   7.07     0  0.469  6.421  78.9  4.9671    2  242       17
2  0.02729   0   7.07     0  0.469  7.185  61.1  4.9671    2  242       17
3  0.03237   0   2.18     0  0.458  6.998  45.8  6.0622    3  222       18
4  0.06905   0   2.18     0  0.458  7.147  54.2  6.0622    3  222       18

        B  LSTAT  MEDV
0  396.90   4.98  24.0
1  396.90   9.14  21.6
2  392.83   4.03  34.7
3  394.63   2.94  33.4
4  396.90   5.33  36.2
```

[2]:
```
df.describe()
```

[2]:
```
             CRIM          ZN       INDUS        CHAS         NOX          RM  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean     3.613524   11.347826   11.136779    0.069170    0.554695    6.284634
std      8.601545   23.310593    6.860353    0.253994    0.115878    0.702617
min      0.006320    0.000000    0.460000    0.000000    0.385000    3.561000
25%      0.082045    0.000000    5.190000    0.000000    0.449000    5.885500
50%      0.256510    0.000000    9.690000    0.000000    0.538000    6.208500
75%      3.677082   12.000000   18.100000    0.000000    0.624000    6.623500
max     88.976200  100.000000   27.740000    1.000000    0.871000    8.780000

              AGE         DIS         RAD         TAX     PTRATIO           B  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000
mean    68.574901    3.795043    9.549407  408.237154   18.083004  356.674032
std     28.148861    2.105710    8.707259  168.537116    2.280574   91.294864
min      2.900000    1.129600    1.000000  187.000000   12.000000    0.320000
25%     45.025000    2.100175    4.000000  279.000000   17.000000  375.377500
```

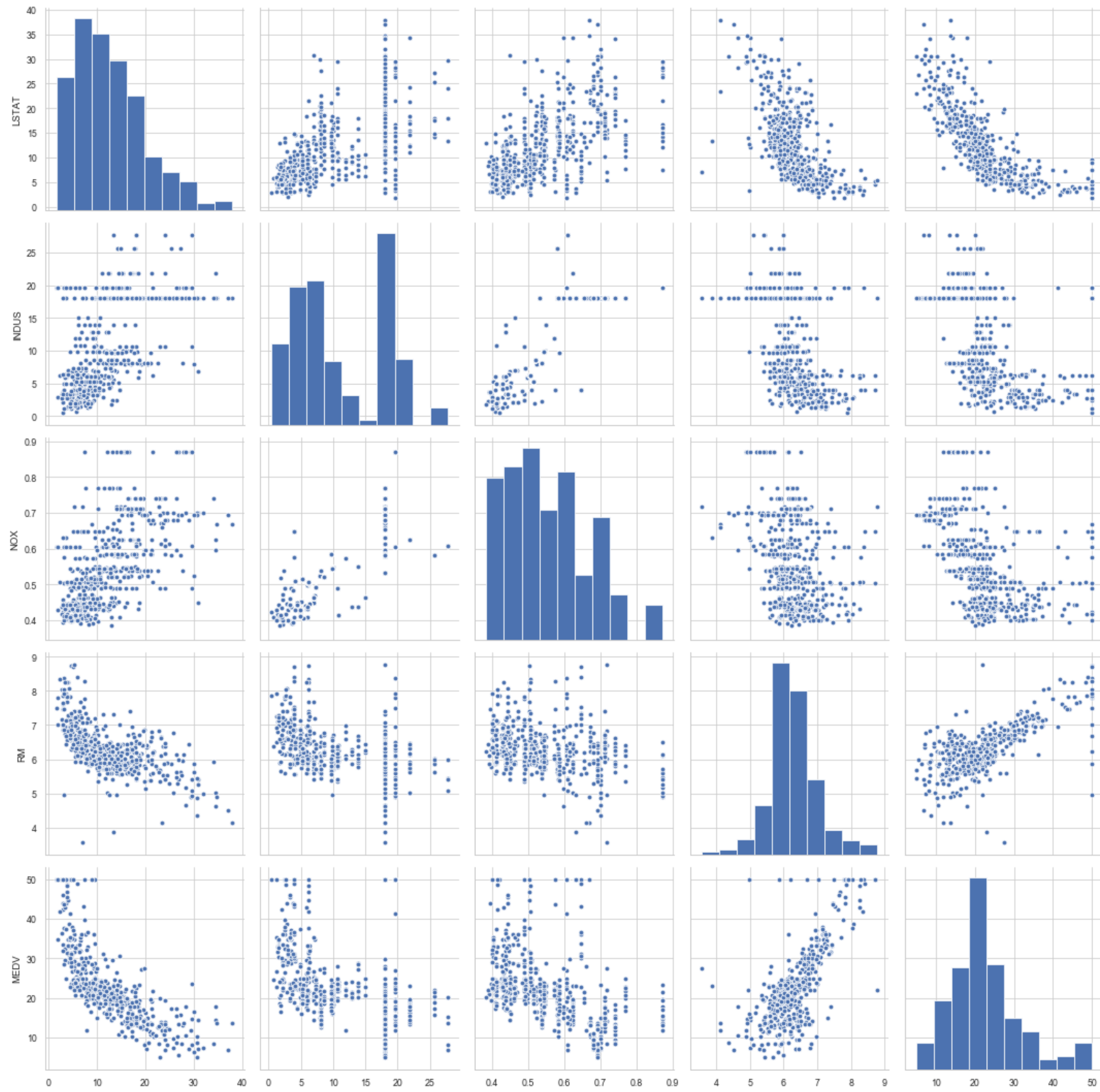
```
50%     77.500000    3.207450    5.000000  330.000000    19.000000  391.440000
75%     94.075000    5.188425   24.000000  666.000000    20.000000  396.225000
max    100.000000   12.126500   24.000000  711.000000    22.000000  396.900000

            LSTAT        MEDV
count  506.000000  506.000000
mean    12.653063   22.532806
std      7.141062    9.197104
min      1.730000    5.000000
25%      6.950000   17.025000
50%     11.360000   21.200000
75%     16.955000   25.000000
max     37.970000   50.000000
```

Seaborn 是基于 matplotlib 的图形可视化 python 包。它提供了一种高度交互式界面，便于用户能够做出各种有吸引力的统计图表。Seaborn 是在 matplotlib 的基础上进行了更高级的 API 封装，从而使得作图更加容易，在大多数情况下使用 seaborn 就能做出很具有吸引力的图，为数据分析提供了很大的便利性。但是应该把 Seaborn 视为 matplotlib 的补充，而不是替代物。

1.1 相关性

[3]:
```
%matplotlib inline

import matplotlib.pyplot as plt
import seaborn as sns
#context 默认'notebook'，还可选“paper”，“talk”，and “poster”，
sns.set(style='whitegrid', context='paper')  # 设定样式，还原可用 sns.
 ↪reset_orig()

# MEDV 是目标变量，为了方便演示，只挑 4 个预测变量
cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']

# scatterplot matrix，对角线上是变量分布的直方图，非对角线上是两个变量的散点图
sns.pairplot(df[cols], height=3)
plt.tight_layout()
# 这是 matplotlib 的方法，# 紧凑显示图片，居中显示
```

```python
# 用下面这行代码可以存储图片到硬盘中
plt.savefig('./figures/scatter1.png', dpi=300)
```



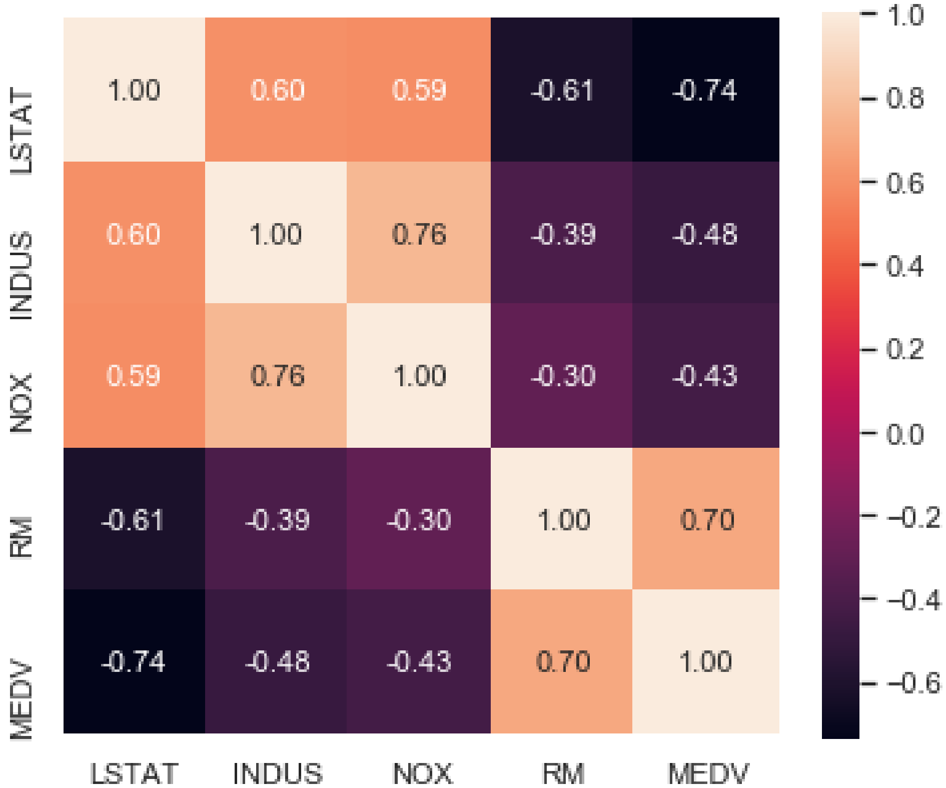
从图中看出 RM 和 MEDV 似乎是有线性关系的, MEDV 类似正态分布.

[4]:
```python
import numpy as np
# 计算相关系数
cm = np.corrcoef(df[cols].values.T)
```

[5]:
```python
cm
```

[5]:
```
array([[ 1.        ,  0.60379972,  0.59087892, -0.61380827, -0.73766273],
       [ 0.60379972,  1.        ,  0.76365145, -0.39167585, -0.48372516],
       [ 0.59087892,  0.76365145,  1.        , -0.30218819, -0.42732077],
       [-0.61380827, -0.39167585, -0.30218819,  1.        ,  0.69535995],
       [-0.73766273, -0.48372516, -0.42732077,  0.69535995,  1.        ]])
```

[6]:
```
#sns.reset_orig()
# correlation map
import numpy as np
# 计算相关系数
cm = np.corrcoef(df[cols].values.T)
sns.set(font_scale=1.0)

# 画相关系数矩阵的热点图
plt.figure(figsize=(6,5))
hm = sns.heatmap(cm,
        annot=True,
        square=True,
        fmt='.2f',
        annot_kws={'size': 11},
        yticklabels=cols,
        xticklabels=cols)
#plt.tight_layout()
plt.savefig('figures\\corr_mat.pdf', dpi=300)
```

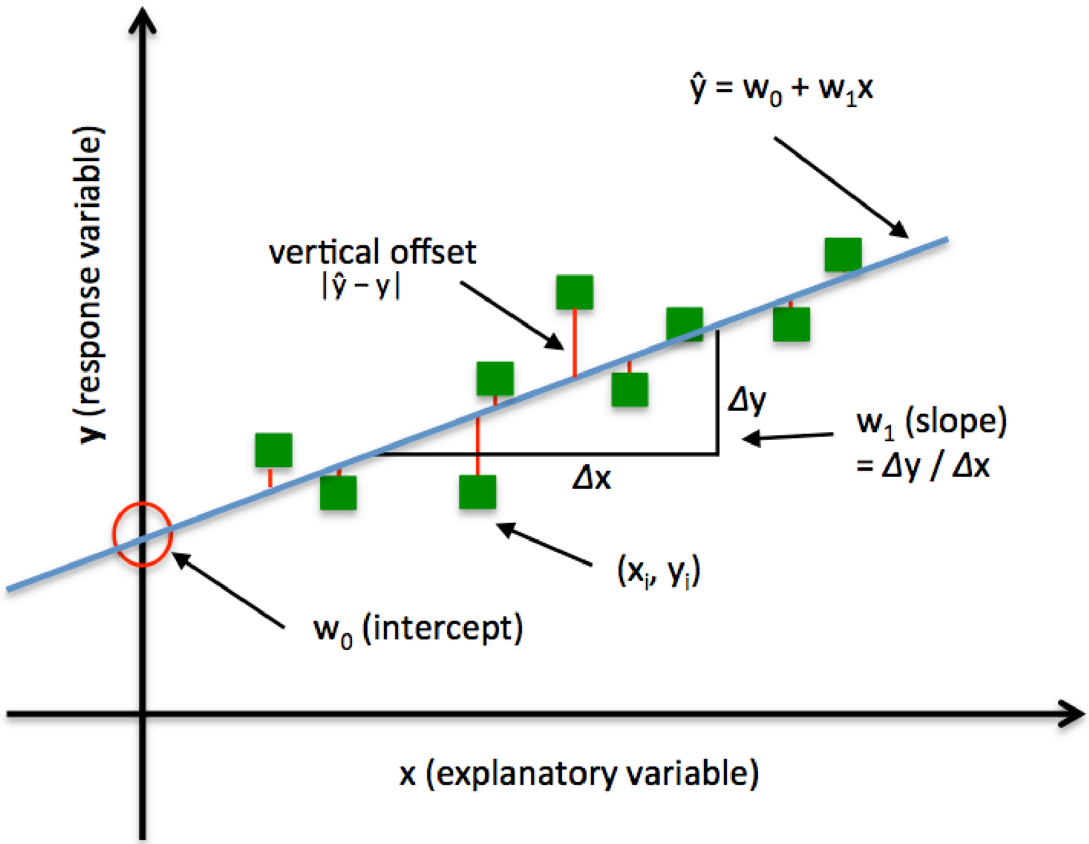


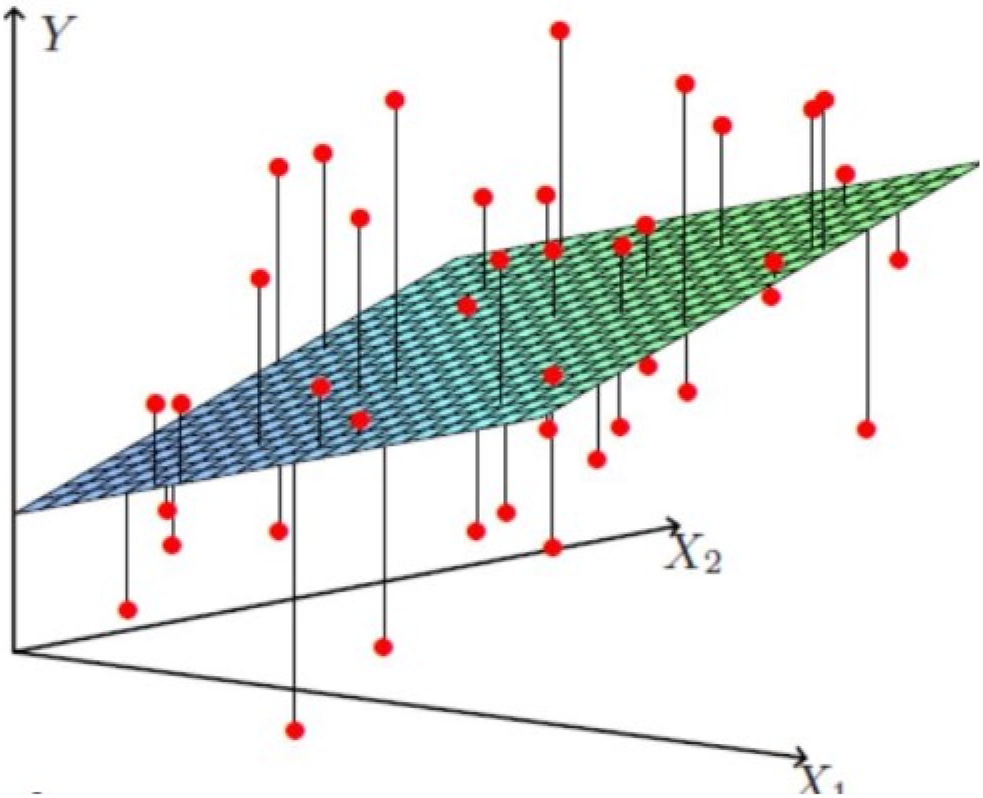
- 我们对与 MEDV 相关性高的变量感兴趣, LSTAT 最高 (-0.74), 其次是 RM (0.7)。
- 但从之前的图看出 MEDV 与 LSTAT 呈非线性关系，而与 RM 更呈线性关系，所以下面选用 RM 来演示简单线性回归。

2 实现简单的线性回归- Ordinary least squares

给定数据集 $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \cdots, (\mathbf{x}^{(n)}, y^{(n)})\}$, 其中 $\mathbf{x}^{(i)} = (x_1^{(i)}, x_1^{(i)}, \cdots, x_m^{(i)})$. 线性回归试图学得一个通过属性的线性组合来进行预测的函数，即

$$y = w_0 + w_1 x_1 + \cdots + w_m x_m$$


$\hat{y} = w_0 + w_1x$
vertical offset
$|\hat{y} - y|$
y (response variable)
Δy
w_1 (slope)
$= \Delta y / \Delta x$
Δx
(x_i, y_i)
w_0 (intercept)
x (explanatory variable)



2.1 利用 gradient descent 求解回归系数

梯度下降法梯度下降法是一个最优化算法，通常也称为最速下降法。最速下降法是求解无约束优化问题最简单和最古老的方法之一，许多有效算法都是以它为基础进行改进和修正而得到的。最速下降法是用负梯度方向为搜索方向的，最速下降法越接近目标值，步长越小，前进越慢。

如果目标函数 $F(x)$ 在点 x 处可微且有定义，那么函数 $F(x)$ 在点 x 沿着梯度相反的方向 $-\nabla F(x)$ 下降最快。其中,∇ 为梯度算子 $\nabla = (\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots, \frac{\partial}{\partial x_n})^T$.

损失函数

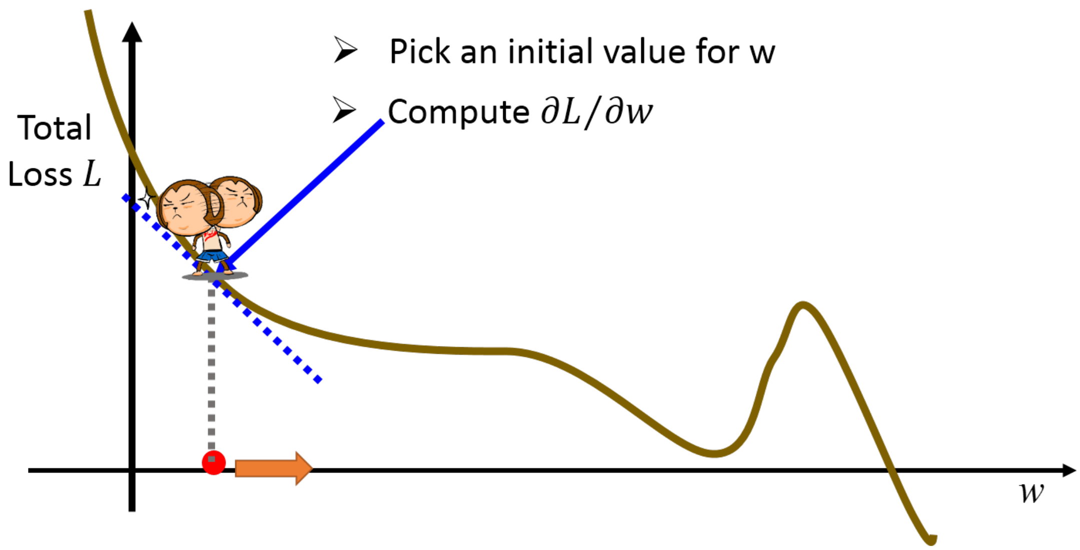
$$J(w) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

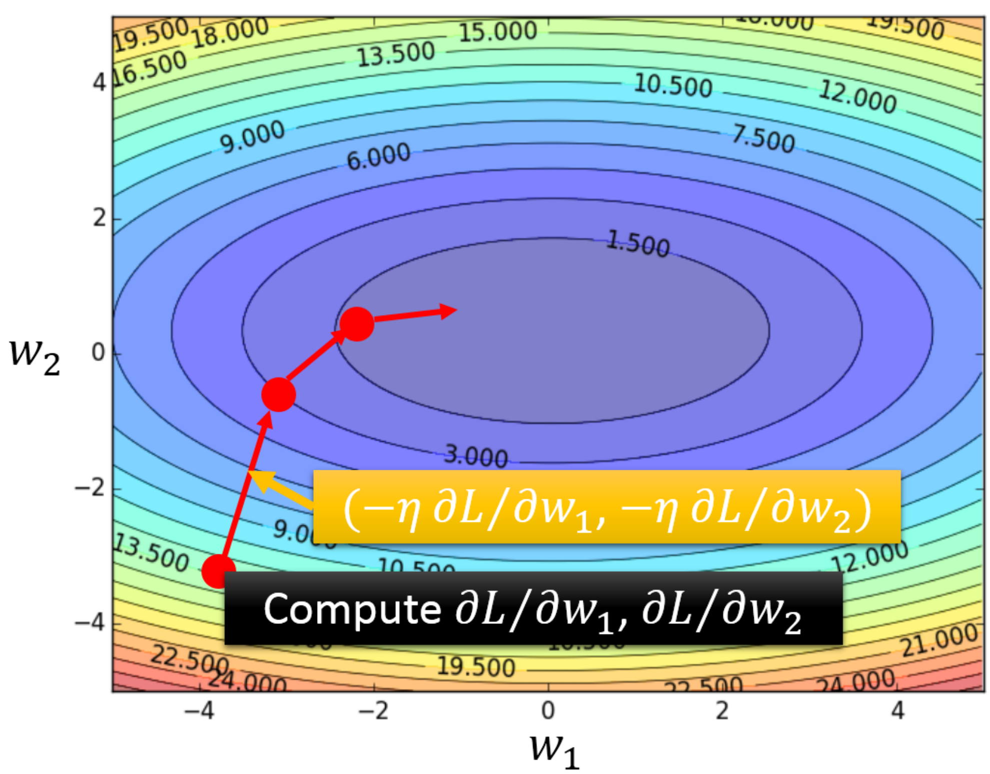
梯度

$$\frac{\partial J}{\partial w_j} = -\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

更新规则

$$w := w - \eta\frac{\partial J}{\partial w}$$






```
[7]: class LinearRegressionGD():


         def __init__(self, eta=0.001, n_iter=20):
             self.eta = eta  # learning rate 学习速率
```

```python
        self.n_iter = n_iter  # 迭代次数

    def fit(self, X, y):  # 训练函数
        # 系数的维度根据 X 的维度进行调整，这个例子中 X 的维度为一
        self.coef_ = np.zeros(shape=(1, X.shape[1]))  # 代表被训练的系数，初始化为 0
        self.intercept_ = np.zeros(1)
        self.cost_ = []   # 用于保存损失的空 list

        for i in range(self.n_iter):
            output = self.net_input(X)  # 计算预测的 Y
            errors = y - output
             # 根据更新规则更新系数，思考一下为什么不是减号？
            self.coef_ += self.eta * np.dot(errors.T, X)
            self.intercept_ += self.eta * errors.sum()  # 更新 bias，相当于 x_0 取常数 1
            cost = (errors**2).sum() / 2.0     # 计算损失
            self.cost_.append(cost)  # 记录损失函数的值
        return self

    def net_input(self, X):   # 矩阵运算，给定系数和 X 计算预测的 Y
        return np.dot(X, self.coef_.T) + self.intercept_

    def predict(self, X):
        return self.net_input(X)
```

[8]:

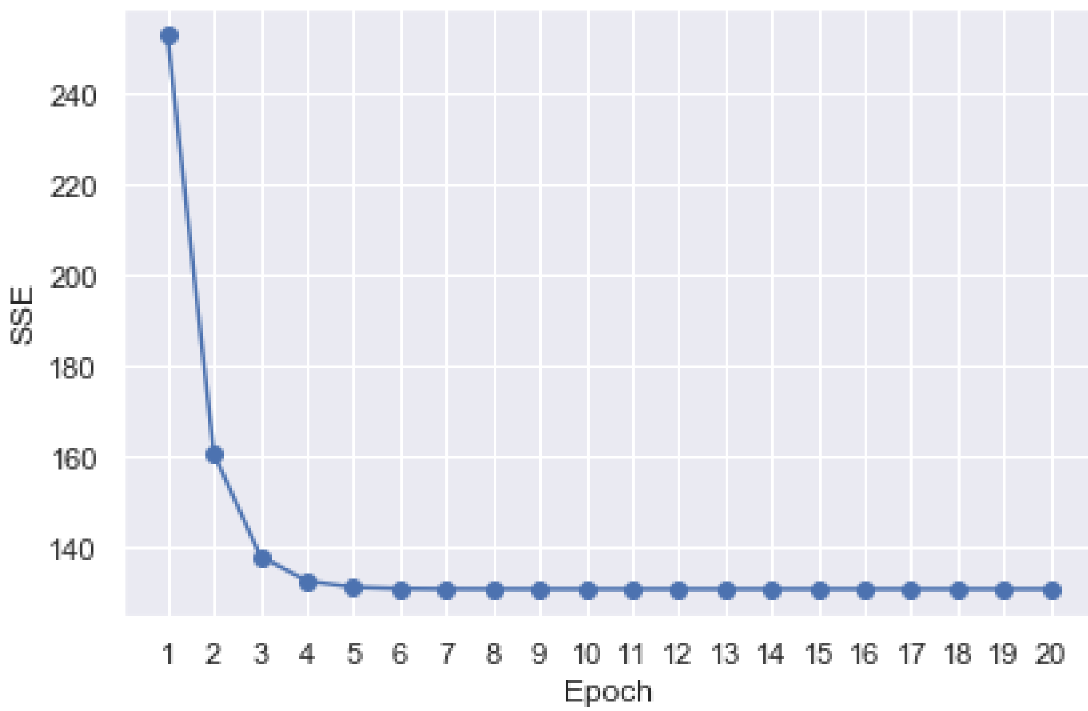
```python
# RM 作为解释变量
X = df[['RM']].values
y = df[['MEDV']].values
```

[9]:

```python
# standardize
from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X_std = sc_x.fit_transform(X)
y_std = sc_y.fit_transform(y)
```

[10]:
```python
lr = LinearRegressionGD()
lr.fit(X_std, y_std);  # 输入数据进行训练
```

[11]:
```python
# cost function
plt.plot(range(1, lr.n_iter+1), lr.cost_,"o-")
plt.ylabel('SSE')
plt.xlabel('Epoch')
plt.tight_layout()
plt.xticks(np.arange(1, lr.n_iter+1, 1))
plt.show()
```



在 epoch 5 之后 cost 基本就不能再减小了.

[12]:
```python
# 定义一个绘图函数用于展示
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='lightblue')
    plt.plot(X, model.predict(X), color='red', linewidth=2)
    return None
```

[13]:
```
# 画出预测
lin_regplot(X_std, y_std, lr)
plt.xlabel('Average number of rooms [RM] (standarized)')
plt.ylabel('Price in $1000\'s [MEDV] (standarized)')
plt.tight_layout()
```

[14]:
```
print('Slope: %.3f' % lr.coef_[0])
print('Intercept: %.3f' % lr.intercept_)
# 直线的斜率及截距
```

```
Slope: 0.695
Intercept: -0.000
```

[15]:
```
# 预测 RM=5 时，房价为多少
num_rooms_std = sc_x.transform([[5.0]])
price_std = lr.predict(num_rooms_std)
print("Price in $1000's: %.3f" % sc_y.inverse_transform(price_std))
```

```
Price in $1000's: 10.840
```

2.2 最常的用方法：利用 scikit-learn 做线性回归

[16]:
```
from sklearn.linear_model import LinearRegression
```

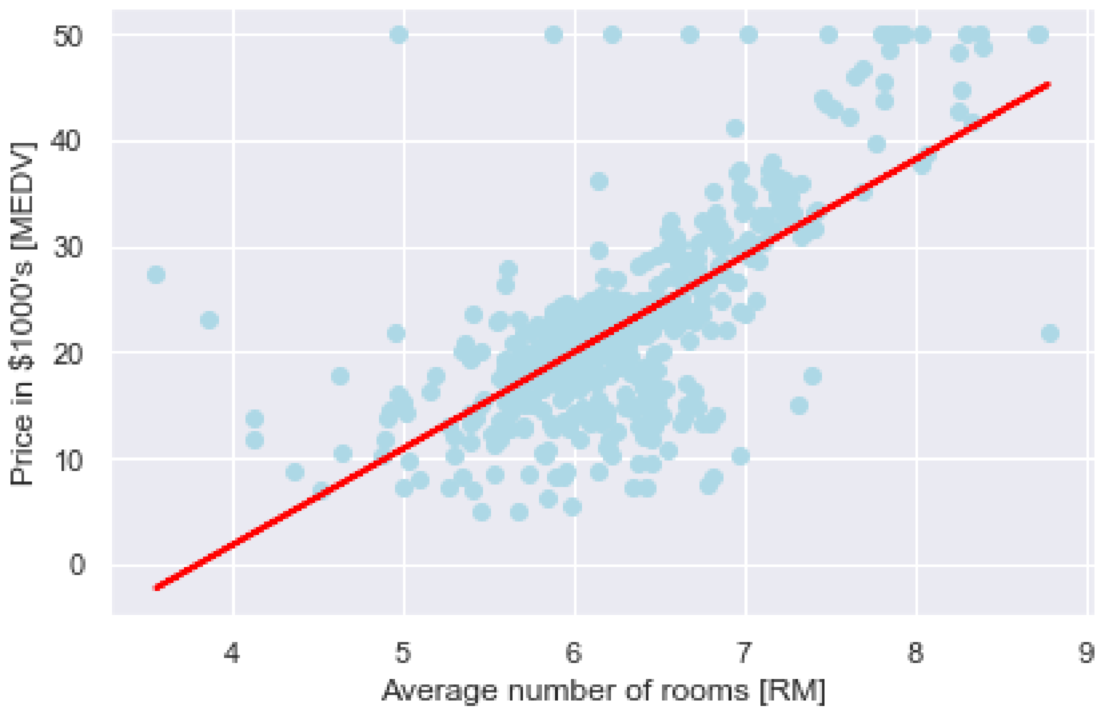
[17]:
```
slr = LinearRegression()
slr.fit(X_std, y_std)
print('Slope: %.3f' % slr.coef_[0])
print('Intercept: %.3f' % slr.intercept_)
```

```
Slope: 0.695
Intercept: -0.000
```

[18]:
```
lin_regplot(X_std, y_std, slr)
plt.xlabel('Average number of rooms [RM] (standardized)')
plt.ylabel('Price in $1000\'s [MEDV] (standardized)')
plt.tight_layout()
```

[19]:
```
# 如果不标准化，直接用原始数据进行回归
slr.fit(X, y)
```

```
lin_regplot(X, y, slr)
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000\'s [MEDV]')
plt.tight_layout()
```



[20]:
```
slr = LinearRegression()
slr.fit(X, y)
print('Slope: %.3f' % slr.coef_[0])
print('Intercept: %.3f' % slr.intercept_)
```

```
Slope: 9.102
Intercept: -34.671
```

3 特征 scaling（尺度变换）

3.1 为什么进行尺度变换？

- 特征间的单位（尺度）可能不同，比如身高和体重，比如摄氏度和华氏度，比如房屋面积和房间数，一个特征的变化范围可能是 [1000,10000]，另一个特征的变化范围可能是 $[-0.1,0.2]$，在进行距离有关的计算时，单位的不同会导致计算结果的不同，尺度大的特征会起决定性作用，而尺度小的特征其作用可能会被忽略，为了消除特征间单位和尺度差异的影响，以对每维特征同等看待，需要对特征进行归一化。

- 原始特征下，因尺度差异，其损失函数的等高线图可能是椭圆形，梯度方向垂直于等高线，下降会走 Z 字形路线，而不是指向 local minimum。通过对特征进行 zero-mean and unit-variance 变换后，其损失函数的等高线图更接近圆形，梯度下降的方向震荡更小，收敛更快。

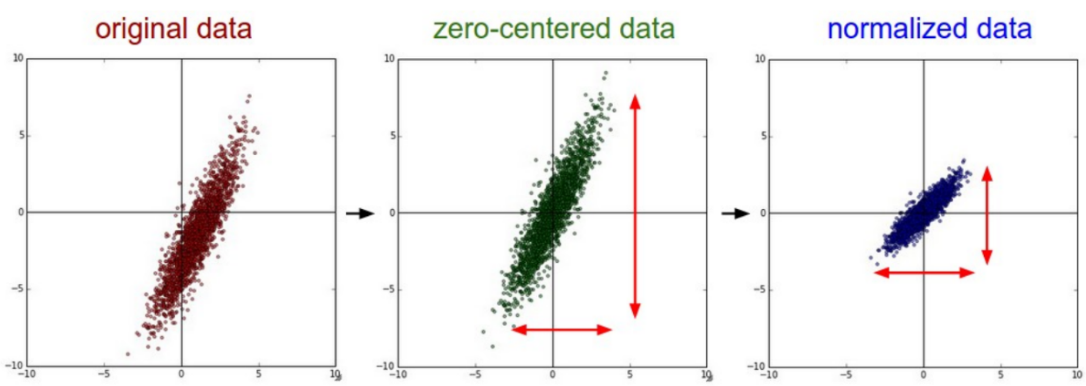
两类常用方法: 归一化（normalization）和标准化（standardization）.

- normalization: rescaling to [0,1], 如 min-max scaling

$$x^{(i)}_{norm} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

- standardization: 更为常见, 因为在一些算法中, weights 初始值都设置为 0, 或者接近 0. standardization 之后会更有利于更新 weights. 并且 standardize 对 outlier 更不敏感，受影响更小

$$x^{(i)}_{std} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$



3.2 什么时候进行特征变换?

- 在聚类过程中，标准化显得尤为重要。这是因为聚类操作依赖于对类间距离和类内聚类之间的衡量。如果一个变量的衡量标准高于其他变量，那么我们使用的任何衡量标准都将受到该

变量的过度影响。

- 在 PCA 降维操作之前。在主成分 PCA 分析之前，对变量进行标准化至关重要。这是因为 PCA 给那些方差较高的变量比那些方差非常小的变量赋予更多的权重。而标准化原始数据会产生相同的方差，因此高权重不会分配给具有较高方差的变量
- KNN 操作，原因类似于 kmeans 聚类。由于 KNN 需要用欧式距离去度量。标准化会让变量之间起着相同的作用。
- 在 SVM 中，使用所有跟距离计算相关的的 kernel 都需要对数据进行标准化。
- 在选择岭回归和 Lasso 时候，标准化是必须的。原因是正则化是有偏估计，会对权重进行惩罚。在量纲不同的情况，正则化会带来更大的偏差。

[21]:
```
# min-max rescaling
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
```

[22]:
```
# standarzation
from sklearn.preprocessing import StandardScaler
stds = StandardScaler()
```

[23]:
```
ex = pd.DataFrame([0, 1, 2 ,3, 4, 5])

# standardize
ex[1] = (ex[0].values - ex[0].values.mean()) / ex[0].values.std()
# normalize
ex[2] = (ex[0].values - ex[0].values.min()) / (ex[0].values.max() - ex[0].
↪values.min())

ex[3] = stds.fit_transform(ex[1].values.reshape(-1,1))

ex[4] = mms.fit_transform(ex[1].values.reshape(-1,1))
ex.columns = ['input', 'standardized', 'normalized', 'standardized_skl',␣
↪'normalized_skl']

ex
```

[23]:

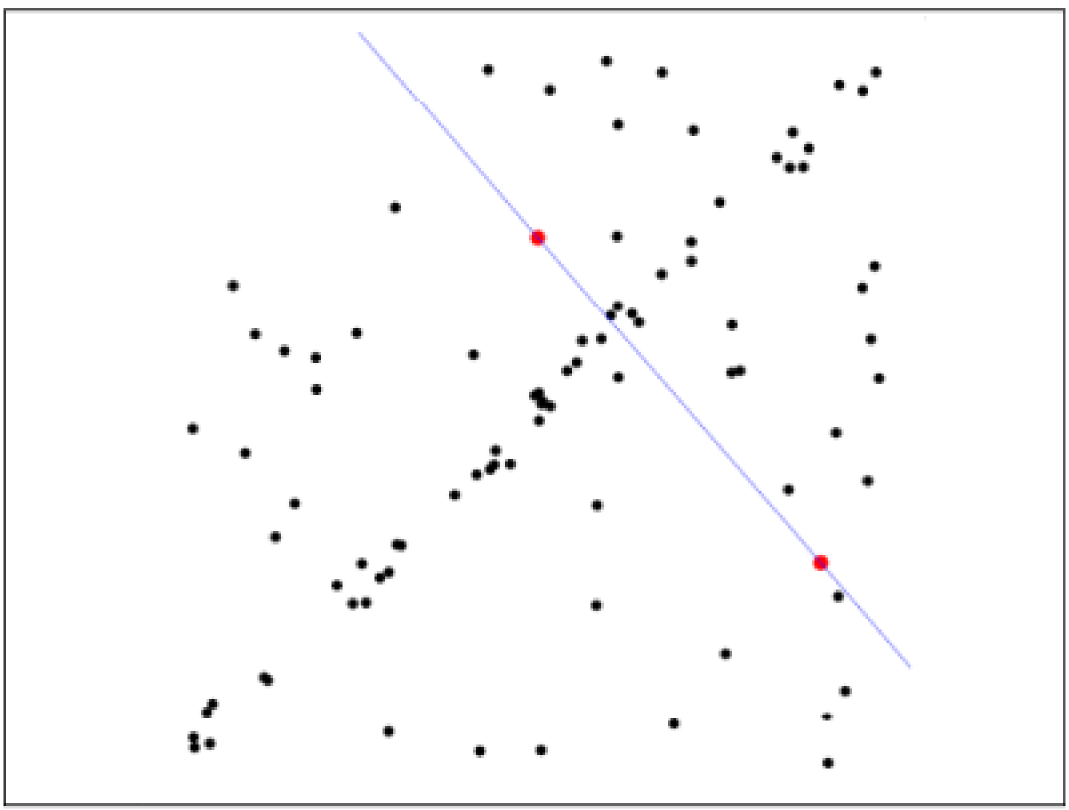
	input	standardized	normalized	standardized_skl	normalized_skl
0	0	-1.46385	0.0	-1.46385	0.0
1	1	-0.87831	0.2	-0.87831	0.2
2	2	-0.29277	0.4	-0.29277	0.4
3	3	0.29277	0.6	0.29277	0.6
4	4	0.87831	0.8	0.87831	0.8
5	5	1.46385	1.0	1.46385	1.0

4 使用 RANSAC 拟合稳健回归

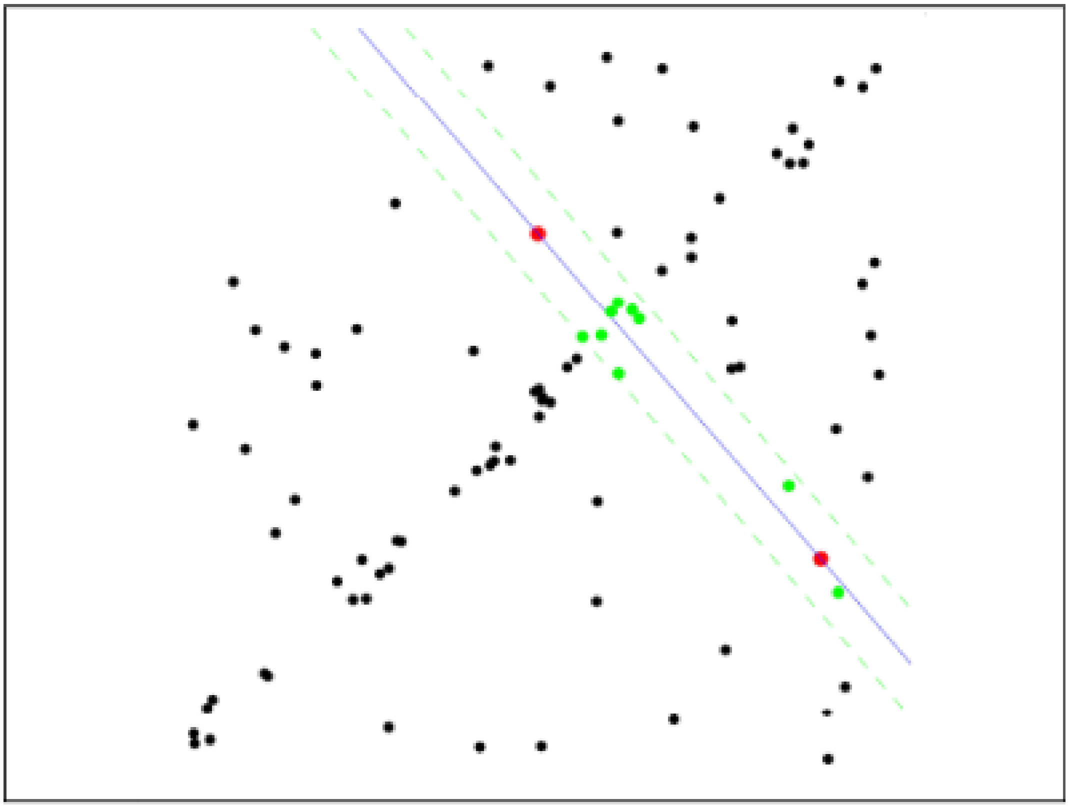
线性回归对离群点比较敏感, 而对是否删除离群点是需要自己进行判断的.

另一种方法就是随机抽样一致算法 RANdom SAmple Consensus (RANSAC)

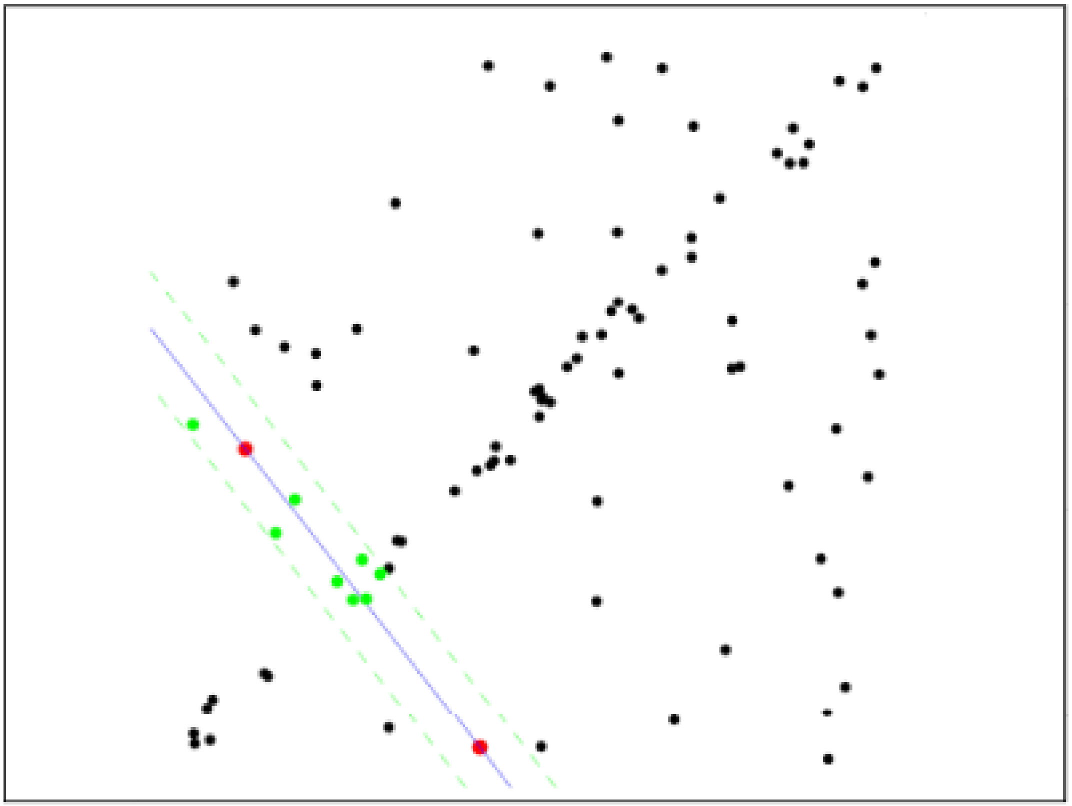
大致算法如下: 第一步：假定模型（如直线方程），并随机抽取 Nums 个（以 2 个为例）样本点，对模型进行拟合：



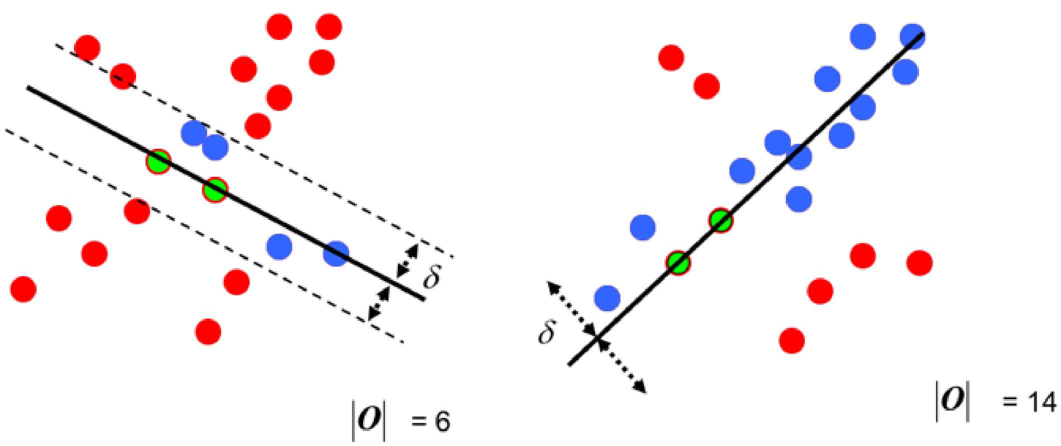
第二步：由于不是严格线性，数据点都有一定波动，假设容差范围为：sigma，找出距离拟合曲线容差范围内的点，并统计点的个数：



第三步：重新随机选取 Nums 个点，重复第一步 ~ 第二步的操作，直到结束迭代：

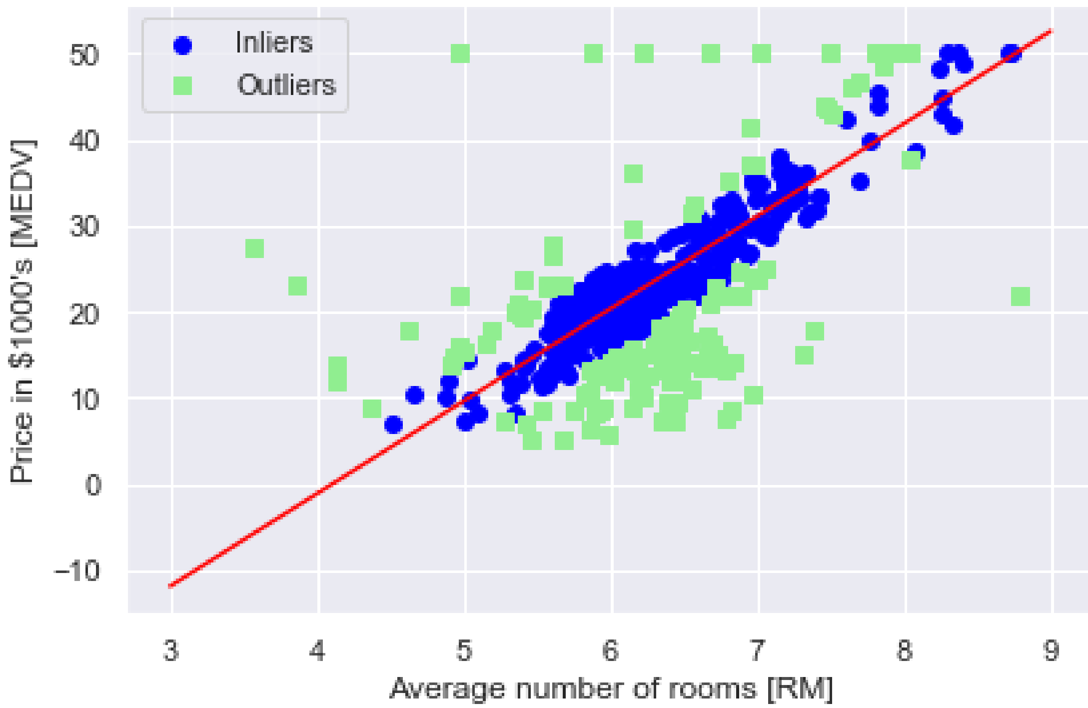


第四步：每一次拟合后，容差范围内都有对应的数据点数，找出数据点个数最多的情况，就是最终的拟合结果：



[24]:
```
# 使用 sklearn 中已有函数
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor(LinearRegression(),
                         # max iteration
```

```
                         max_trials=100,
                         # min number of randomly chosen samples
                         min_samples=50,
                         # absolute vertical distances to measure
                         loss='absolute_loss',
                         # allow sample as inlier within 5 distance units
                         residual_threshold=5.0,
                         random_state=0)
#  all data samples with absolute residuals smaller than the
#  residual_threshold are considered as inliers.

ransac.fit(X, y)

# 分出 inlier 和 outlier
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

line_X = np.arange(3, 10, 1)
line_y_ransac = ransac.predict(line_X[:, np.newaxis])
plt.scatter(X[inlier_mask], y[inlier_mask],
            c='blue', marker='o', label='Inliers')
plt.scatter(X[outlier_mask], y[outlier_mask],
            c='lightgreen', marker='s', label='Outliers')
plt.plot(line_X, line_y_ransac, color='red')
plt.xlabel('Average number of rooms [RM]')
plt.ylabel('Price in $1000\'s [MEDV]')
plt.legend(loc='upper left')

plt.tight_layout()
```



[25]:
```
print('Slope: %.3f' % ransac.estimator_.coef_[0])
print('Intercept: %.3f' % ransac.estimator_.intercept_)
```

```
Slope: 10.735
Intercept: -44.089
```

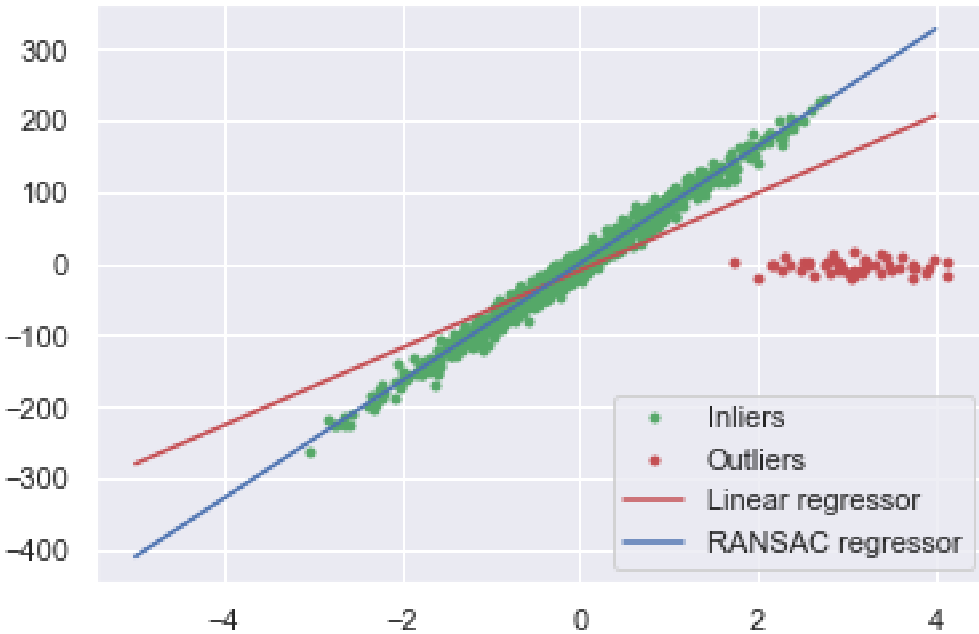
RANSAC 减少了 outlier 的影响, 但对于未知数据的预测能力是否有影响未知.

4.1 对比 RANSAC 回归和 OLS 回归

[26]:
```
# 造一组数据进行模拟
from sklearn import datasets

n_samples = 1000
n_outliers = 50

# Generate a random regression problem.
# coef: coefficient of the underlying linear model
X, y, coef = datasets.make_regression(n_samples=n_samples,
```

```
                                          n_features=1,
                                          n_informative=1, noise=10,
                                          coef=True, random_state=0)

# 添加 outlier 数据
np.random.seed(0)
X[:n_outliers] = 3 + 0.5 * np.random.normal(size=(n_outliers, 1))
y[:n_outliers] = -3 + 10 * np.random.normal(size=n_outliers)

# 使用所有数据进行拟合
model = LinearRegression()
model.fit(X, y)

# 利用 RANSAC 算法进行拟合
model_ransac = RANSACRegressor(LinearRegression())
model_ransac.fit(X, y)
inlier_mask = model_ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)

# 预测数据
line_X = np.arange(-5, 5)
line_y = model.predict(line_X[:, np.newaxis])
line_y_ransac = model_ransac.predict(line_X[:, np.newaxis])

# 对比预估的系数
print("Estimated coefficients:")
print("underlying linear model: ",coef)
print("RLS: ", model.coef_)
print("ransac: ", model_ransac.estimator_.coef_)

plt.plot(X[inlier_mask], y[inlier_mask], '.g', label='Inliers')
plt.plot(X[outlier_mask], y[outlier_mask], '.r', label='Outliers')
plt.plot(line_X, line_y, '-r', label='Linear regressor')
plt.plot(line_X, line_y_ransac, '-b', label='RANSAC regressor')
plt.legend(loc='lower right');
```

```
Estimated coefficients:
underlying linear model:  82.1903908407869
RLS:  [54.17236387]
ransac:  [82.08533159]
```



5 评估线性回归模型的性能

训练过程中在模型没有见过的数据上进行性能评测也是非常重要的，因为这样才能得到没有偏差的评估。

[27]:
```python
from sklearn.model_selection import train_test_split

X = df.iloc[:, :-1].values
y = df['MEDV'].values
```

[28]:
```python
from sklearn.model_selection import train_test_split

X = df.iloc[:, :-1].values
y = df['MEDV'].values
```

```python
# 切分训练集与测试集
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0)
# 70% 用于 train, 30% 用于 test
```

[29]:
```python
slr = LinearRegression()

slr.fit(X_train, y_train)
y_train_pred = slr.predict(X_train)
y_test_pred = slr.predict(X_test)
```

5.1 残差图

[30]:
```python
# residual plot, 经常被用来检查回归模型
plt.scatter(y_train_pred,  y_train_pred - y_train,
            c='blue', marker='o', label='Training data')
plt.scatter(y_test_pred,  y_test_pred - y_test,
            c='lightgreen', marker='s', label='Test data')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
plt.xlim([-10, 50])
plt.tight_layout()
# plt.savefig('./figures/slr_residuals.png', dpi=300)
```

如果预测都是正确的, 那么 residual 就是 0. 这是理想情况, 实际中, 对一个好的回归模型，期望误差是随机分布的，同时残差也随机分布于中心线附近。

如果我们从残差图中找出规律，就意味着模型遗漏了某些能够影响残差的解释信息。此外，还可以通过残差图来发现异常值，这些异常值看上去距离中心线有较大的偏差。

5.2 评估指标

- SSE(和方差、误差平方和)：The sum of squares due to error

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

- MSE(均方差、方差)：Mean squared error 就是 SSE 的平均值

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

- RMSE(均方根、标准差)：Root mean squared error

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2}{n}}$$

- SSR：sum of square of the regression()，预测数据与原始数据均值之差的平方和

$$SSR = \sum_{i=1}^{n}(\hat{y}^{(i)} - \bar{y}^{(i)})^2$$

- SST：total sum of square，即原始数据和均值之差的平方和

$$SST = \sum_{i=1}^{n}(y^{(i)} - \bar{y}^{(i)})^2$$

- R-square(确定系数)：Coefficient of determination，代表着有多少百分比的数据被模型解释. 越高代表模型拟合越好

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{MSE}{Var(y)}$$

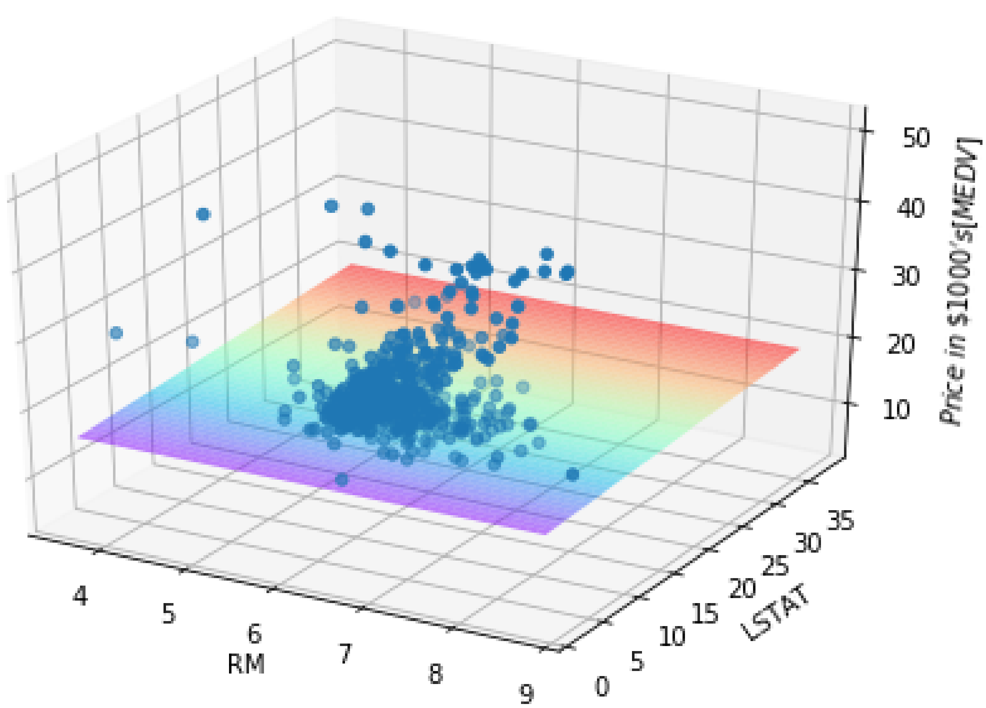
[31]:
```
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```

```
MSE train: 20.007, test: 27.431
R^2 train: 0.764, test: 0.671
```

6 多元线性回归

[32]:
```
sns.reset_orig()
from mpl_toolkits.mplot3d.axes3d import Axes3D
# 选择两个变量进行回归
columns = ['RM','LSTAT']
X_rm = df['RM'].values
X_lstat = df['LSTAT'].values
X = df[columns].values
z = df['MEDV'].values

regr = LinearRegression()
regr = regr.fit(X, z)
```

```python
x = np.linspace(X_rm.min(),X_rm.max(),50).reshape(-1,1)
y = np.linspace(X_lstat.min(),X_lstat.max(),50).reshape(-1,1)
X_mesh = np.concatenate((x,y),axis=1)
X,Y = np.meshgrid(x,y)
Z = regr.predict(X_mesh).reshape(-1,1)

fig = plt.figure()
axes3d = Axes3D(fig)

axes3d.scatter(X_rm,X_lstat,z, depthshade=True)
axes3d.plot_surface(X,Y,Z, alpha=0.5, cmap=plt.cm.rainbow)
axes3d.set_xlabel('RM')
axes3d.set_ylabel('LSTAT')
axes3d.set_zlabel('$Price \; in \; \$1000\'s [MEDV]$')
plt.show()
```



7 多项式回归 Polynomial regression

线性的拟合不好？改变特征？换换模型？

[33]:
```python
import numpy as np
```

[34]:
```python
X = np.array([258.0, 270.0, 294.0,
              320.0, 342.0, 368.0,
              396.0, 446.0, 480.0, 586.0])[:, np.newaxis]

y = np.array([236.4, 234.4, 252.8,
              298.6, 314.2, 342.2,
              360.8, 368.0, 391.2,
              390.8])
```

[35]:
```python
from sklearn.preprocessing import PolynomialFeatures

lr = LinearRegression()
pr = LinearRegression()
quadratic = PolynomialFeatures(degree=2)
X_quad = quadratic.fit_transform(X)
```

[36]:
```python
print(X.shape)
print(X_quad.shape)
```

```
(10, 1)
(10, 3)
```

[37]:
```python
X
```

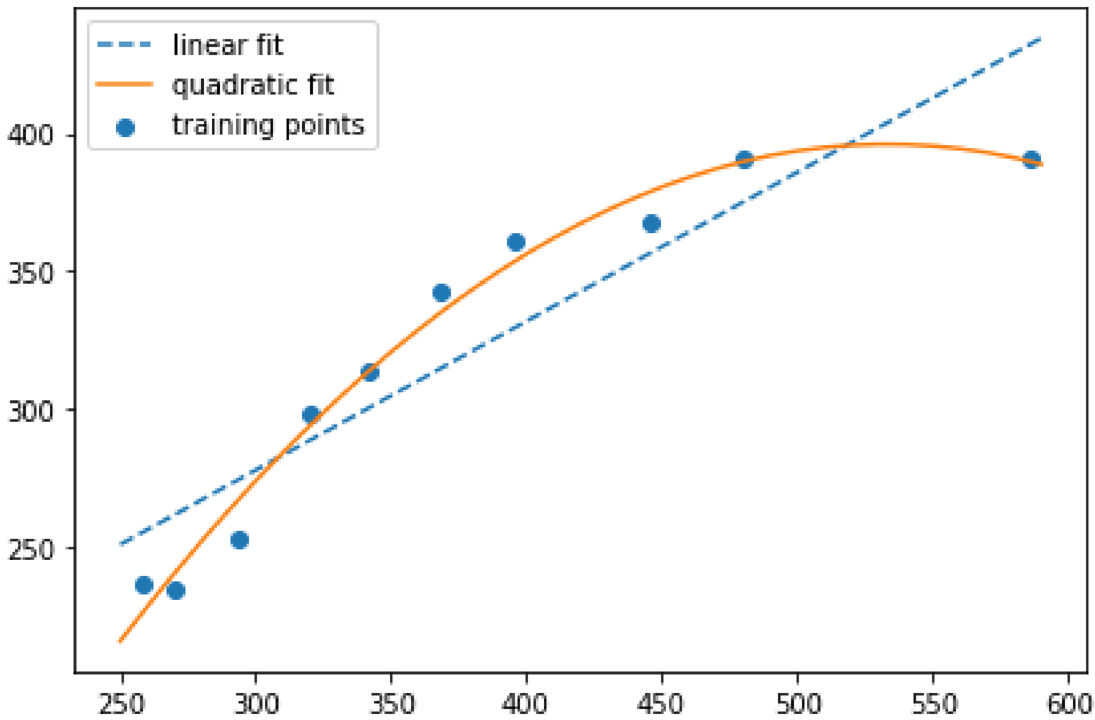
[37]:
```
array([[258.],
       [270.],
       [294.],
       [320.],
       [342.],
       [368.],
       [396.],
       [446.],
       [480.],
```

```
        [586.]])
```

[38]:
```
X_quad
```

[38]:
```
array([[1.00000e+00, 2.58000e+02, 6.65640e+04],
       [1.00000e+00, 2.70000e+02, 7.29000e+04],
       [1.00000e+00, 2.94000e+02, 8.64360e+04],
       [1.00000e+00, 3.20000e+02, 1.02400e+05],
       [1.00000e+00, 3.42000e+02, 1.16964e+05],
       [1.00000e+00, 3.68000e+02, 1.35424e+05],
       [1.00000e+00, 3.96000e+02, 1.56816e+05],
       [1.00000e+00, 4.46000e+02, 1.98916e+05],
       [1.00000e+00, 4.80000e+02, 2.30400e+05],
       [1.00000e+00, 5.86000e+02, 3.43396e+05]])
```

[39]:
```
# fit linear features
lr.fit(X, y)
X_fit = np.arange(250,600,10)[:, np.newaxis]
y_lin_fit = lr.predict(X_fit)

# fit quadratic features
pr.fit(X_quad, y)
y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))

# plot results
plt.scatter(X, y, label='training points')
plt.plot(X_fit, y_lin_fit, label='linear fit', linestyle='--')
plt.plot(X_fit, y_quad_fit, label='quadratic fit')
plt.legend(loc='upper left')

plt.tight_layout()
```



图上可以发现 quadratic fit 比 linear 拟合效果更好

[40]:
```
y_lin_pred = lr.predict(X)
y_quad_pred = pr.predict(X_quad)
```

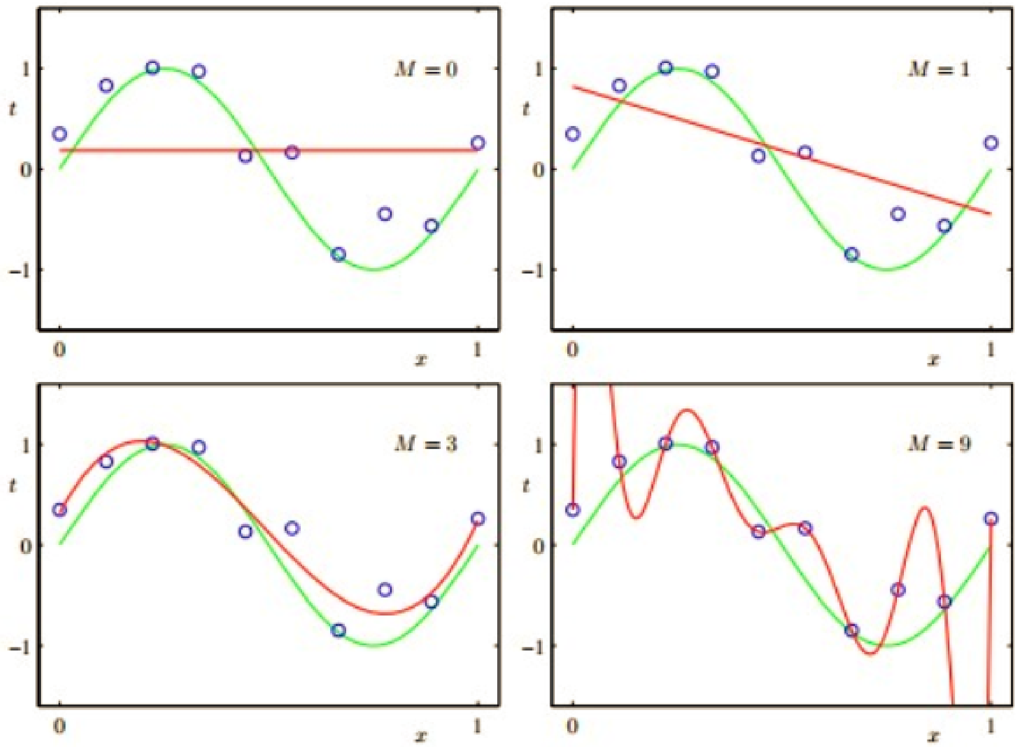
[41]:
```
print('Training MSE linear: %.3f, quadratic: %.3f' % (
        mean_squared_error(y, y_lin_pred),
        mean_squared_error(y, y_quad_pred)))
print('Training  R^2 linear: %.3f, quadratic: %.3f' % (
        r2_score(y, y_lin_pred),
        r2_score(y, y_quad_pred)))
```

```
Training MSE linear: 569.780, quadratic: 61.330
Training  R^2 linear: 0.832, quadratic: 0.982
```

MSE 下降到 61, R^2 上升到 98%, 说明在这个数据集上 quadratic fit 效果更好.

8 正则化

8.1 过拟合



引起过拟合的原因

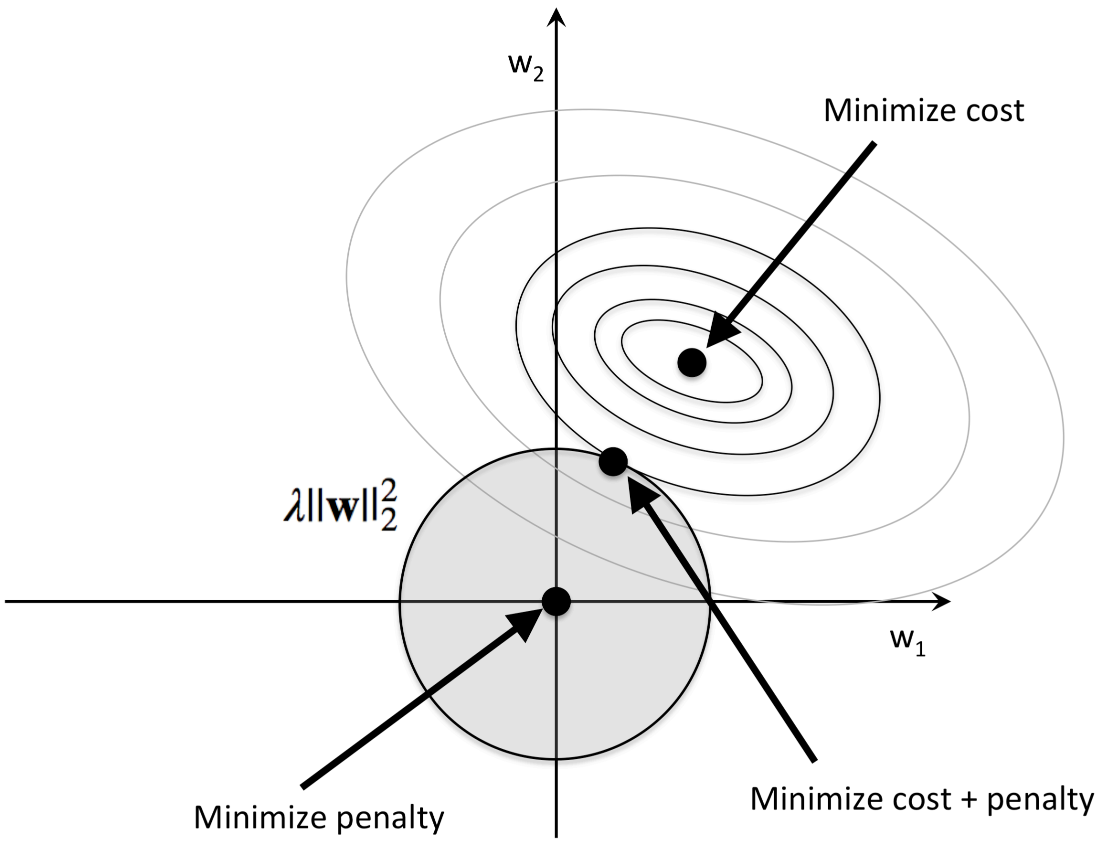
- 训练数据过少，训练数据的分布不能表示整体样本的分布。
- 特征过多。特征太多其实也属于模型复杂。
- 模型过于复杂。高阶多项式。

8.2 岭回归 (Ridge 回归)

岭回归是基于 L_2 惩罚项的模型，是在损失函数中加入权重的平方和。损失函数

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda\|w\|_2$$

其中，$\|w\|_2^2 = \sum_{j=1}^{m} w_j^2$。




```
[42]: # Ridge regression 岭回归
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=10)
ridge.fit(X_quad, y)
print('Weights: ',ridge.coef_)
print('Intercept: %.3f' % ridge.intercept_)
```

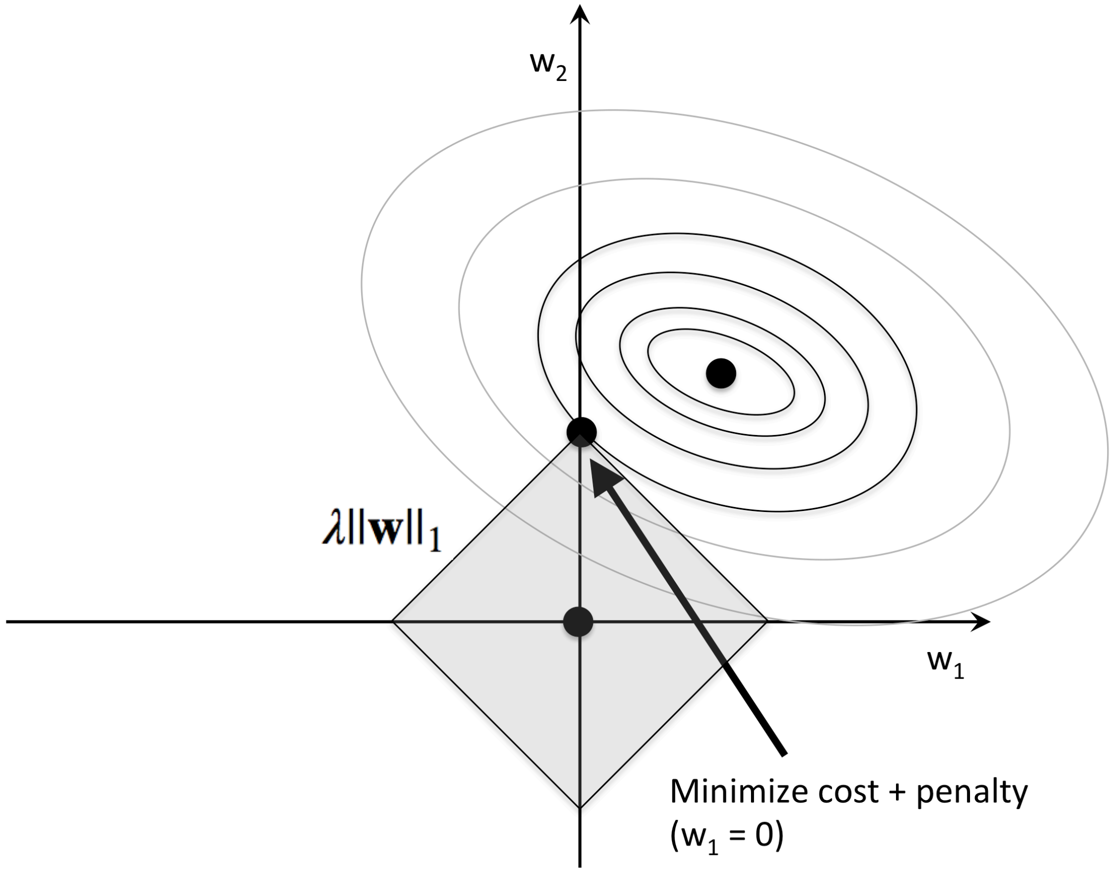
```
Weights:  [ 0.00000000e+00  2.38249608e+00 -2.23060370e-03]
Intercept: -240.009
```

8.3 LASSO 回归

对于基于稀疏数据训练的模型，还有另外一种解决方案，即 LASSO。基于正则化项的强度，某些权重可以为零（使得对应的权重 x_i 失去作用），这也使得 LASSO 成为一种监督特征选择技术。LASSO 是在损失函数上加上权重的 L_1 范数。损失函数

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda\|w\|_1$$

其中 $\|w\|_1 = \sum_{j=1}^{m} |w_j|$。



[43]:
```
#  Lasso 回归
from sklearn.linear_model import Lasso
# 调节 alpha 可以实现对拟合的程度
lasso = Lasso(alpha=0.01)
lasso.fit(X_quad, y)
print('Weights: ', lasso.coef_)
print('Intercept: %.3f' % lasso.intercept_)
```

```
Weights:  [ 0.00000000e+00  2.39885313e+00 -2.25010921e-03]
Intercept: -243.214
```

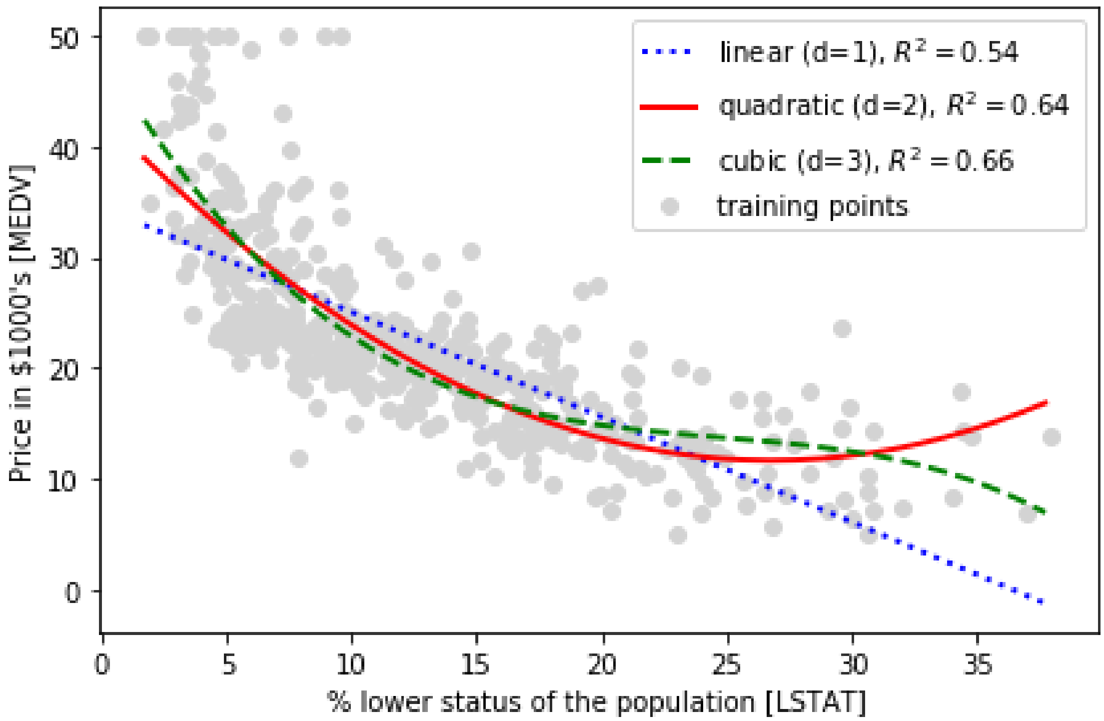
9 为 Housing 数据集进行非线性建模

我们将房价与 LSTAT 的 quadratic 及 cubic polynomials 进行拟合, 并与线性模型进行对比。

[44]:
```
X = df[['LSTAT']].values
y = df['MEDV'].values

regr = LinearRegression()

# create quadratic features
quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)
X_quad = quadratic.fit_transform(X)
X_cubic = cubic.fit_transform(X)

# fit features
X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]

regr = regr.fit(X, y)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X))

regr = regr.fit(X_quad, y)
y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
quadratic_r2 = r2_score(y, regr.predict(X_quad))

regr = regr.fit(X_cubic, y)
y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
cubic_r2 = r2_score(y, regr.predict(X_cubic))



# plot results
plt.scatter(X, y, label='training points', color='lightgray')

plt.plot(X_fit, y_lin_fit,
         label='linear (d=1), $R^2=%.2f$' % linear_r2,
         color='blue',
         lw=2,
         linestyle=':')
```

```
plt.plot(X_fit, y_quad_fit,
         label='quadratic (d=2), $R^2=%.2f$' % quadratic_r2,
         color='red',
         lw=2,
         linestyle='-')

plt.plot(X_fit, y_cubic_fit,
         label='cubic (d=3), $R^2=%.2f$' % cubic_r2,
         color='green',
         lw=2,
         linestyle='--')

plt.xlabel('% lower status of the population [LSTAT]')
plt.ylabel('Price in $1000\'s [MEDV]')
plt.legend(loc='upper right')

plt.tight_layout()
# plt.savefig('./figures/polyhouse_example.png', dpi=300)
print("linear_r2: ",linear_r2)
print("quadratic_r2: ",quadratic_r2)
print("cubic_r2: ", cubic_r2)
```

```
linear_r2:  0.5441462975864799
quadratic_r2:  0.6407168971636611
cubic_r2:  0.6578476405895719
```



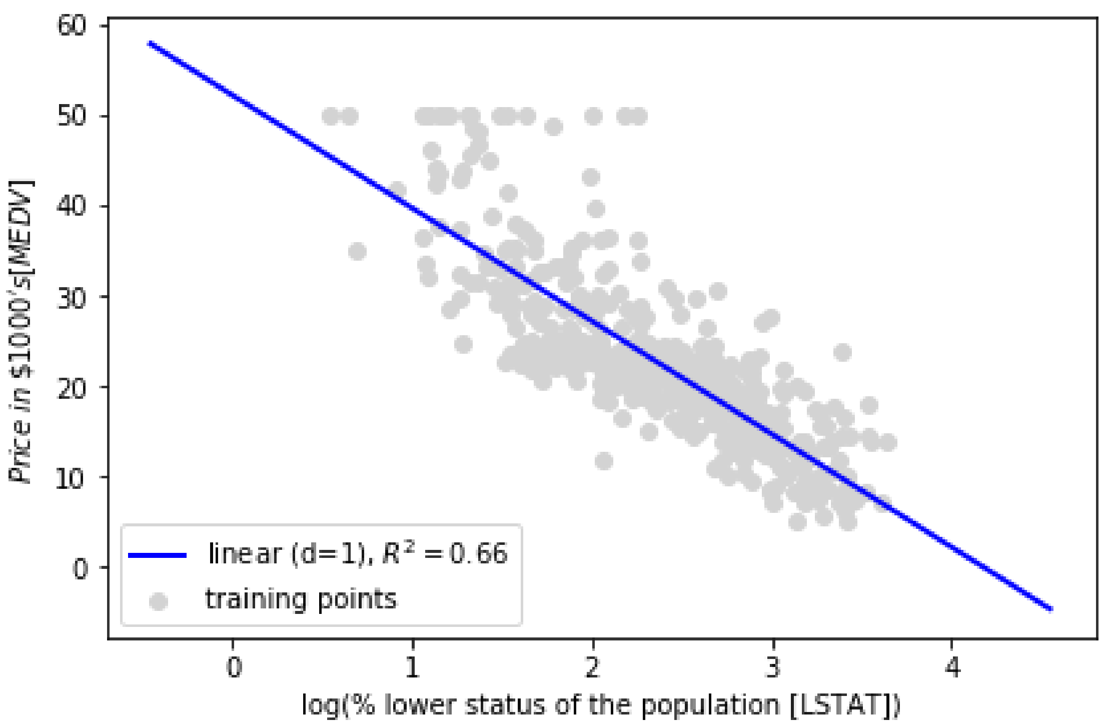
10 变换特征

[45]:

```python
X = df[['LSTAT']].values
y = df['MEDV'].values

# transform features
X_log = np.log(X)

# fit features
X_fit = np.arange(X_log.min()-1, X_log.max()+1, 1)[:, np.newaxis]

regr = regr.fit(X_log, y)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X_log))

# plot results
plt.scatter(X_log, y, label='training points', color='lightgray')
```

```
plt.plot(X_fit, y_lin_fit,
         label='linear (d=1), $R^2=%.2f$' % linear_r2,
         color='blue',
         lw=2)

plt.xlabel('log(% lower status of the population [LSTAT])')
plt.ylabel('$Price \; in \; \$1000\'s [MEDV]$')
plt.legend(loc='lower left')

plt.tight_layout()
#plt.savefig('./figures/transform_example.png', dpi=300)
#plt.show()
print("linear_r2: ",linear_r2)
```

```
linear_r2:  0.6649462248792692
```



经过 log 变换后，线性拟合效果已经不错, 比单纯 polynomial fit 更好

11 练习：用房价数据的其它自变量一起做一个多元模型看看 R2 有没有改善

[46]:
```
a = df[['LSTAT']].values
a_log = np.log(a)
df['xxx'] = a_log
# 构造特征

cola = ['CRIM','LSTAT','xxx']
X = df[cola].values
y = df['MEDV'].values
# 构造 X, y

regr = LinearRegression()
regr = regr.fit(X, y)
# 模型拟合

linear_r2 = r2_score(y, regr.predict(X))
# 算分
print("linear_r2: ",linear_r2)
```

```
linear_r2:  0.6848169354285594
```

[47]:
```
columns = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
        'DIS', 'RAD', 'TAX','PTRATIO', 'B', 'LSTAT']
X = df[columns].values
y = df['MEDV'].values
# 构造 X, y

regr = LinearRegression()
regr = regr.fit(X, y)
# 模型拟合

linear_r2 = r2_score(y, regr.predict(X))
# 算分
print("linear_r2: ",linear_r2)
```

```
linear_r2:  0.7395067658012934
```