- K近邻算法

1. Sklearn中的使用

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666, test_size=0.2)
knn_clf = KNeighborsClassifier(n_neighbors=5)
knn_clf.fit(X_train, y_train)
y_predict = knn_clf.predict(X_test)
knn_clf.score(X_test, y_test)
accuracy_score(y_test, y_predict)
```

2. 网格搜索与超参数

```
from sklearn.model_selection import GridSearchCV
param_grid = [
  {
    'weights': ['uniform'],
    'n_neighbors': [i for i in range(1, 11)]
  },
  {
    'weights': ['distance'],
    'n_neighbors': [i for i in range(1, 11)],
    'p': [i for i in range(1, 6)]
  }
]
grid_search = GridSearchCV(knn_clf, param_grid, n_jobs=-1, verbose=2)
grid_search.fit(X_train, y_train)
grid_search.best_estimator_
grid_search.best_score_
grid_search.best_params_
knn_clf = grid_search.best_estimator_
```

3. 数据归一化

最值归一化：把所有数据映射到0-1之间

$x_{scale} = \frac{x - x_{min}}{x_{max} - x_{min}}$ （适用于分布有明显边界的情况）

均值方差归一化：将数据归一到均值为0方差为1的分布中

$x_{scale} = \frac{x - x_{mean}}{s}$ （适用于数据分布没有明显的边界，有可能存在极端数据值）

注意：对测试数据集进行归一化$\frac{(x_{test} - mean_train)}{std_train}$

```
from sklearn.preprocessing import StandardScaler
standardScaler = StandardScaler()
standardScaler.fit(X_train)
standardScaler.mean_
standardScaler.scale_
X_train = standardScaler.transform(X_train)
X_test_standard = standardScaler.transform(X_test)
```

4. 总结

缺点：

计算量大，m个样本，n个特征，预测每一个新的数据，需要$O(m*n)$

高度数据相关

预测结果不具有可解释性

- 线性回归算法

> *简单线性回归*

1. 目标：找到a和b，使得$J(a,b)=\sum_{i=1}^{m}(y^{(i)}-ax^{(i)}-b)^2$尽可能小
2. 推导过程

$$\frac{\partial(a,b)}{\partial b}=0, \frac{\partial(a,b)}{\partial a}=0$$

$$a=\frac{\sum_{i=1}^{m}(x^{(i)}-\overline{x})(y^{(i)}-\overline{y})}{\sum_{i=1}^{m}(x^{(i)}-\overline{x})^2}$$

$$b=\overline{y}-a\overline{x}$$

3. 向量化

$$\sum_{i=1}^{m}w^{(i)}\cdot v^{(i)}$$

$$w=(w^{(1)},w^{(2)},\cdots,w^{(m)})$$

$$v=(v^{(1)},v^{(2)},\cdots,v^{(m)})$$

```
a = (x_train-x_mean).dot(y_train_y_mean) / (x_train-x_mean).dot(x_train-x_mean)
b = y_mean - a * x_mean
```

4. 评价

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

RMSE(均方根误差)$\sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_{test}^{(i)}-\hat{y}_{test}^{(i)})^2}$

MAE(平均绝对误差)$\frac{1}{m}\sum_{i=1}^{m}|y_{test}^{(i)}-\hat{y}_{test}^{(i)}|$

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

$R^2\leq 1$，越大越好。当$R^2<0$时，说明学习到的模型还不如基准模型，可能数据不存在线性关系。

5. Sklearn中的使用

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

> *多元线性回归*

1. 公式

$$y=\theta_0+\theta_1x_1+\theta_2x_2+\cdots+\theta_nx_n$$

$$\hat{y}^{(i)}=\theta_0+\theta_1X_1^{(i)}+\cdots+\theta_nX_n^{(i)}$$

$$\theta=(\theta_0,\theta_1,\cdots,\theta_n)^T$$

$$X^{(i)}=(X_0^{(i)},X_1^{(i)},\cdots,X_n^{(i)}),X_0^{(i)}\equiv 1$$

$$\hat{y}^{(i)}=X^{(i)}\cdot\theta$$

$$X_b = \begin{pmatrix} 1 & X_1^{(1)} & \cdots & X_n^{(1)} \\ 1 & X_1^{(2)} & \cdots & X_n^{(2)} \\ \cdots & & & \\ 1 & X_1^{(m)} & \cdots & X_n^{(m)} \end{pmatrix}$$

$\hat{y} = X_b \cdot \theta$

2. 目标：$\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$尽可能小

即使得$(y - X_b \cdot \theta)^T(y - X_b \cdot \theta)$尽可能小

解得$\theta = (X_b^T X_b)^{-1} X_b^T y$

3. Sklearn中的使用

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
# 线性回归法解决回归问题
from sklearn.linear_model import LinearRegression
boston = datasets.load_boston()
X = boston.data
y = boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
lin_reg.coef_
lin_reg.intercept_
lin_reg.score(X_test, y_test)
# KNN解决回归问题
from sklearn.neighbors import KNeighborsRegressor
knn_reg = KNeighborsRegressor()
lin_reg.fit(X_train, y_train)
lin_reg.score(X_test, y_test)
```

总结

1. 典型的参数学习
2. 只能解决回归问题
3. 不需要数据归一化

- 梯度下降法

1. 线性回归中的梯度下降法

$\theta = (\theta_0, \theta_1, \cdots, \theta_n)$

$-\eta \nabla J$

$\nabla J = (\frac{\partial J}{\partial \theta_0}, \frac{\partial J}{\partial \theta_1}, \cdots, \frac{\partial J}{\partial \theta_n})$

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

$\hat{y}^{(i)} = \theta_0 + \theta_1 X_1^{(i)} + \cdots + \theta_n X_n^{(i)}$

目标：$\sum_{i=1}^{m}(y^{(i)} - \theta_0 - \theta_1 X_1^{(i)} - \cdots - \theta_n X_n^{(i)})^2$尽可能小

$$\nabla J(\theta) = \begin{pmatrix} \sum_{i=1}^{m} 2(y^{(i)} - X_b^{(i)}\theta) \cdot (-1) \\ \sum_{i=1}^{m} 2(y^{(i)} - X_b^{(i)}\theta) \cdot (-X_1^{(i)}) \\ \cdots \\ \sum_{i=1}^{m} 2(y^{(i)} - X_b^{(i)}\theta) \cdot (-X_n^{(i)}) \end{pmatrix}$$

目标：$J = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2$尽可能小

$J(\theta) = MSE(y, \hat{y})$

$$\nabla J(\theta) = \frac{2}{m}\begin{pmatrix} \sum_{i=1}^{m}(X_b^{(i)}\theta - y^{(i)}) \\ \sum_{i=1}^{m}(X_b^{(i)}\theta - y^{(i)})\cdot X_1^{(i)} \\ \cdots \\ \sum_{i=1}^{m}(X_b^{(i)}\theta - y^{(i)})\cdot X_n^{(i)} \end{pmatrix}$$

2. 向量化

$$\frac{2}{m}\cdot(x_b^{(1)}\theta - y^{(1)}, x_b^{(2)}\theta - y^{(2)}, \cdots, x_b^{(m)}\theta - y^{(m)})\cdot\begin{pmatrix} 1 & X_1^{(1)} & \cdots & X_n^{(1)} \\ 1 & X_1^{(2)} & \cdots & X_n^{(2)} \\ \cdots \\ 1 & X_1^{(m)} & \cdots & X_n^{(m)} \end{pmatrix} = \frac{2}{m}\cdot(X_b\theta - y)^T\cdot X_b$$

$$\nabla J(\theta) = \frac{2}{m}\cdot X_b^T(X_b\theta - y)$$

3. 实现线性回归中的梯度下降法(something wrong)

```python
import numpy as np
x = 2 * np.random.random(size=100).reshape(-1, 1)
y = x * 3. + 4. + np.random.normal(size=100)
X_b = np.hstack([np.ones((len(x), 1)), x])
initial_theta = np.zeros(X_b.shape[1])
eta = 0.01
def J(theta, X_b, y):
    try:
        return np.sum((y - X_b.dot(theta))**2) / len(X_b)
    except:
        return float('inf')
def dJ(theta, X_b, y):
    return X_b.T.dot(X_b.dot(theta) - y) * 2. / len(X_b)
def gradient_descent(X_b, y, initial_theta, eta, n_iters = 1e4, epsilon=1e-8):
    theta = initial_theta
    print(theta)
    i_iter = 0
    while i_iter < n_iters:
        gradient = dJ(theta, X_b, y)
        last_theta = theta
        theta = theta - eta * gradient
        if abs(J(theta, X_b, y) - J(last_theta, X_b, y)) < epsilon:
            break
        i_iter += 1
    return theta
```

4. 随机梯度下降法

$$2\begin{pmatrix} (X_b^{(i)}\theta - y^{(i)}) \\ (X_b^{(i)}\theta - y^{(i)})\cdot X_1^{(i)} \\ \cdots \\ (X_b^{(i)}\theta - y^{(i)})\cdot X_n^{(i)} \end{pmatrix} = 2\cdot(X_b^{(i)})^T\cdot(X_b^{(i)}\theta - y^{(i)})$$

$\eta = \frac{a}{i\_iters + b}$ 经验取值$a = 5, b = 50$

```python
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=5)//默认值为5
sgd_reg.fit(X_train_standard,y_train)
sgd_reg.score(X_test_standard,y_test)
```

5. 梯度的调试

$$\theta_0^+ = (\theta_0 + \varepsilon, \theta_1, \cdots, \theta_n)$$

$$\theta_0^- = (\theta_0 - \varepsilon, \theta_1, \cdots, \theta_n)$$

$\frac{\partial J}{\partial \theta_0} = \frac{J(\theta_0^+)-J(\theta_0^-)}{2\varepsilon}$

```python
def dJ_debug(theta,x_b,y,epsilon=0.01):
    res = np.empty(len(theta))
    for i in range(len(theta)):
        theta_1 = theta.copy()
        theta_1[i] += epsilon
        theta_2 = theta.copy
        theta_2[i] -= epsilon
        res[i] = (J(theta_1,x_b,y)-J(theta_2,x_b,y))/(2*epsilon)
    return res
```

6. 总结

数据需要归一化

随机搜索

梯度上升法：最大化效用函数$+\eta\frac{dJ}{d\theta}$

- 主成分分析法

1. 目标

$Var(X_{project}) = \frac{1}{m}\sum_{i=1}^{m}||X_{project}^{(i)} - \overline{X}_{project}||^2$最大

demean处理，$\frac{1}{m}\sum_{i=1}^{m}||X_{project}^{(i)}||^2$

$w = (w_1, w_2)$

$X^{(i)} = (X_1^{(i)}, X_2^{(i)})$

$\frac{1}{m}\sum_{i=1}^{m}(X^{(i)} \cdot w)^2$

拓展到n维：$\frac{1}{m}\sum_{i=1}^{m}(X_1^{(i)}w_1 + X_2^2 w_2 + \cdots + X_n^{(i)}w_n)^2$

2. 梯度上升法

$$\nabla f = \frac{2}{m}\begin{pmatrix} \sum_{i=1}^{m}(X^{(i)}w)X_1^{(i)} \\ \sum_{i=1}^{m}(X^{(i)}w)X_2^{(i)} \\ \cdots \\ \sum_{i=1}^{m}(X^{(i)}w)X_n^{(i)} \end{pmatrix} = \frac{2}{m}X^T(Xw)$$

$$\frac{2}{m}\cdot(X^{(1)}w, X^{(2)}w, \cdots, X^{(m)}w)\cdot\begin{pmatrix} X_1^{(1)} & \cdots & X_n^{(1)} \\ X_1^{(2)} & \cdots & X_n^{(2)} \\ \cdots \\ X_1^{(m)} & \cdots & X_n^{(m)} \end{pmatrix} = \frac{2}{m}\cdot(Xw)^T\cdot X$$

3. 实现主成分分析法

```python
import numpy as np
class PCA:
    def __init__(self, n_components):
        """初始化PCA"""
        assert n_components >= 1, "n_components must be valid"
        self.n_components = n_components
        self.components_ = None
    def fit(self, X, eta=0.01, n_iters=1e4):
        """获得数据集X的前n个主成分"""
        assert self.n_components <= X.shape[1], \
            "n_components must not be greater than the feature number of X"
        def demean(X):
            return X - np.mean(X, axis=0)
        def f(w, X):
            return np.sum((X.dot(w) ** 2)) / len(X)
        def df(w, X):
            return X.T.dot(X.dot(w)) * 2. / len(X)
        def direction(w):
            return w / np.linalg.norm(w)
        def first_component(X, initial_w, eta=0.01, n_iters=1e4, epsilon=1e-8):
            w = direction(initial_w)
            cur_iter = 0
            while cur_iter < n_iters:
                gradient = df(w, X)
                last_w = w
                w = w + eta * gradient
                w = direction(w) #将向量化为单位向量
                if (abs(f(w, X) - f(last_w, X)) < epsilon):
                    break
                cur_iter += 1
            return w
        X_pca = demean(X)
        self.components_ = np.empty(shape=(self.n_components, X.shape[1]))
        for i in range(self.n_components):
            initial_w = np.random.random(X_pca.shape[1]) #不能从0向量开始
            w = first_component(X_pca, initial_w, eta, n_iters)
            self.components_[i,:] = w
            X_pca = X_pca - X_pca.dot(w).reshape(-1, 1) * w
        return self
    def transform(self, X):
        """将给定的X，映射到各个主成分分量中"""
        assert X.shape[1] == self.components_.shape[1]
        return X.dot(self.components_.T)
    def inverse_transform(self, X):
        """将给定的X，反向映射回原来的特征空间"""
        assert X.shape[1] == self.components_.shape[0]
        return X.dot(self.components_)
    def __repr__(self):
        return "PCA(n_components=%d)" % self.n_components
```

注意：不能使用StandardScaler标准化数据

4. 高维数据向低维数据映射

$$X = \begin{pmatrix} X_1^{(1)} & \cdots & X_n^{(1)} \\ X_1^{(2)} & \cdots & X_n^{(2)} \\ \cdots & & \\ X_1^{(m)} & \cdots & X_n^{(m)} \end{pmatrix}$$

$$W_k = \begin{pmatrix} W_1^{(1)} & \cdots & W_n^{(1)} \\ W_1^{(2)} & \cdots & W_n^{(2)} \\ \cdots & & \\ W_1^{(k)} & \cdots & W_n^{(k)} \end{pmatrix}$$ 前k个主成分

$X \cdot W_k^T = X_k$ 高维向低维映射

$X_k \cdot W_k = X_m$ 低维恢复为高维，X_m和X并不等同

5. Sklearn中的使用

```
from sklearn.decomposition import PCA
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
digits = datasets.load_digits()
X = digits.data
```

```
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y)
pca = PCA(0.95) #解释95%的方差,还可以传入n_components
pca.fit(X_train)
X_train_reduction = pca.transform(X_train)
X_test_reduction = pca.transform(X_test)
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_reduction, y_train)
print(knn_clf.score(X_test_reduction, y_test))
print(pca.explained_variance_ratio_)
print(pca.n_components_)
```

降到2维后的可视化

```
pca = PCA(n_components=2)
pca.fix(X)
X_reduction = pca.transform(X)
for i in range(10):
  plt.scatter(X_reduction[y==i,0], X_reduction[y==i,1], alpha=0.8)
plt.show()
```

6. PCA降噪应用-特征脸

```
import numpy as np
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
faces = fetch_lfw_people()
random_indexes = np.random.permutation(len(faces.data))
X = faces.data[random_indexes]
pca = PCA(svd_solver="randomized")
pca.fit(X)
```

- 多项式回归与模型泛化

1. 引例

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
x = np.random.uniform(-3, 3, size=100)
X = x.reshape(-1, 1)
y = 0.5 * x**2 + x + 2 + np.random.normal(0, 1, size=100)
plt.scatter(x, y)
X2 = np.hstack([X, X**2])
lin_reg = LinearRegression()
lin_reg.fit(X2, y)
y_predict = lin_reg.predict(X2)
plt.plot(np.sort(x),y_predict[np.argsort(x)],color='r')
plt.show()
print(lin_reg.coef_)
print(lin_reg.intercept_)
```

2. PolynomialFeatures

PolynomialFeatures(degree=3)

$$x_1, x_2 \begin{cases} 1, x_1, x_2 \\ x_1^2, x_2^2, x_1 x_2 \\ x_1^3, x_2^3, x_1^2 x_2, x_1 x_2^2 \end{cases}$$

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
x = np.random.uniform(-3, 3, size=100)
X = x.reshape(-1, 1)
```

```
y = 0.5 * x**2 + x + 2 + np.random.normal(0, 1, size=100)
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2)
poly.fit(X)
X2 = poly.transform(X)
print(X2.shape)
lin_reg = LinearRegression()
lin_reg.fit(X2, y)
y_predict = lin_reg.predict(X2)
plt.scatter(x, y)
plt.plot(np.sort(x), y_predict[np.argsort(x)], color='r')
plt.show()
#PolynomialFeatures
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
X = np.arange(1, 11).reshape(-1, 2)
poly = PolynomialFeatures(degree=2)
poly.fit(X)
X2 = poly.transform(X)
print(X2)
```

3. pipeline

```
import numpy as np
from sklearn.linear_model import LinearRegression
x = np.random.uniform(-3, 3, size=100)
X = x.reshape(-1, 1)
y = 0.5 * x**2 + x + 2 + np.random.normal(0, 1, size=100)
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
poly_reg = Pipeline([
    ("poly", PolynomialFeatures(degree=2)),
    ("std_scaler", StandardScaler()),
    ("lin_reg", LinearRegression())
])
poly_reg.fit(X, y)
y_predict = poly_reg.predict(X)
```

4. 误差

```
from sklearn.metrics import mean_squared_error
print(mean_squared_error(y, y_predict))
```

5. 过拟合与欠拟合

欠拟合：算法所训练的模型不能完整表述数据关系

过拟合：算法训练的模型过多地表达了数据间的噪音关系

6. 学习曲线

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
x = np.random.uniform(-3, 3, size=100)
X = x.reshape(-1, 1)
y = 0.5 * x**2 + x + 2 + np.random.normal(0, 1, size=100)
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
poly_reg = Pipeline([
    ("poly", PolynomialFeatures(degree=2)),
    ("std_scaler", StandardScaler()),
    ("lin_reg", LinearRegression())
])
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)
train_score = []
test_score = []
for i in range(1, 76):
    lin_reg = poly_reg
    lin_reg.fit(X_train[:i],y_train[:i])
    y_train_predict = lin_reg.predict(X_train[:i])
    train_score.append(mean_squared_error(y_train[:i], y_train_predict))
    y_test_predict = lin_reg.predict(X_test)
    test_score.append(mean_squared_error(y_test, y_test_predict))

plt.plot([i for i in range(1, 76)], np.sqrt(train_score), label="train")
plt.plot([i for i in range(1, 76)], np.sqrt(test_score), label="test")
plt.legend()
plt.show()
```

7. 验证数据集与交叉验证

> *数据划分*

训练数据

验证数据

测试数据：不参与模型创建，作为衡量最终模型性能的数据集

> *交叉验证*

训练数据化为A、B、C，BC训练A验证，AC训练B验证，AB训练C验证，k个模型均值为结果调参

```
import numpy as np
from sklearn import datasets
digits = datasets.load_digits()
X = digits.data
y = digits.target
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
knn_clf = KNeighborsClassifier()
print(cross_val_score(knn_clf, X_train, y_train, cv=5)) #默认分成3份
```

k-folds交叉验证：把训练数据集分成k份，每次训练k个模型，相当于整体性能慢了k倍

留一法LOO-CV：将训练数据集分成m份(m个样本)，Leave-One-Out Cross Validation，完全不受随机的影响，最接近模型真正的性能指标，但计算量巨大

8. 偏差与方差

> *偏差与方差*

偏差（Bias）：对问题本身假设不正确，欠拟合，如对非线性数据使用线性回归

方差（Variance）：数据的一点点扰动都会较大地影响模型，通常原因在于使用的模型太复杂，过拟合，如高阶多项式回归

> *相关算法*

天生高方差算法：KNN，非参数学习通常都是高方差算法，因为不对数据进行任何假设

天生高偏差算法：线性回归，参数学习通常都是高偏差的算法，因为对数据具有极强的假设

> *调整偏差和方差*

KNN中对k的调整：k越小，模型越复杂，偏差越小；k越大，模型越简单，方差越小。

线性回归中使用多项式回归，degree越小，模型越简单，偏差越大；degree越大，模型越复杂，方差越大。

> *解决高方差*

- 降低模型复杂度
- 减少数据维度，降噪
- 增加样本数
- 使用验证集
- 模型正则化

9. 模型正则化

> *岭回归：使$J(\theta) = MSE(y, \hat{y}; \theta) + \alpha \frac{1}{2} \sum_{i=1}^{n} \theta_i^2$尽可能小*

```
from sklearn.linear_model import Ridge
def RidgeRegression(degree, alpha):
    return Pipeline([
        ("poly", PolynomialFeatures(degree=degree)),
        ("std_scaler", StandardScaler()),
        ("ridge_reg", Ridge(alpha=alpha))
    ])
ridgel_reg = RidgeRegression(20, 0.001)
```

> *LASSO回归：使$J(\theta) = MSE(y, \hat{y}; \theta) + \alpha \sum_{i=1}^{n} |\theta_i|$尽可能小*

```
from sklearn.linear_model import Lasso
def RidgeRegression(degree, alpha):
    return Pipeline([
        ("poly", PolynomialFeatures(degree=degree)),
        ("std_scaler", StandardScaler()),
        ("ridge_reg", Lasso(alpha=alpha))
    ])
ridgel_reg = RidgeRegression(20, 0.01)
```

LASSO趋向于使一部分theta值变为0，可作为特征选择用

> *弹性网：使$J(\theta) = MSE(y, \hat{y}; \theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^{n} \theta_i^2$尽可能小*

- 逻辑回归与分类评价

1. Sigmoid函数

$\hat{p} = \sigma(\theta^T \cdot x_b) = \frac{1}{1+e^{-\theta^T \cdot x_b}}$

$\hat{y} = \begin{cases} 1 & \hat{y} \geq 0.5 \\ 0 & \hat{y} < 0.5 \end{cases}$

2. 损失函数

$cost = \begin{cases} -log(\hat{p}) & y = 1 \\ -log(1-\hat{p}) & y = 0 \end{cases}$

$cost = -ylog(\hat{p}) - (1-y)log(1-\hat{p})$

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}log(\hat{p}^{(i)}) + (1-y^{(i)})log(1-\hat{p}^{(i)})$$

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}log(\sigma(X_b^{(i)}\theta)) + (1-y^{(i)})log(1-\sigma(X_b^{(i)}\theta))$$

3. 损失函数的梯度

$$\frac{J(\theta)}{\theta_j} = \frac{1}{m}\sum_{i=1}^{m}(\sigma(X_b^{(i)}\theta) - y^{(i)})X_j^{(i)}$$

$$\nabla J(\theta) = \frac{1}{m}\begin{pmatrix} \sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)}) \\ \sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})\cdot X_1^{(i)} \\ \dots \\ \sum_{i=1}^{m}(\hat{y}^{(i)} - y^{(i)})\cdot X_n^{(i)} \end{pmatrix} = \frac{1}{m}X_b^T\cdot(\sigma(X_b\theta) - y)$$

4. 决策边界

$\theta^T\cdot x_b = 0$

5. 多项式特征

正则化：$C\cdot J(\theta) + L_1/L_2$

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(666)
X = np.random.normal(0,1,size=(200,2))
y = np.array(X[:,0]**2 + X[:,1] < 1.5,dtype='int')
for _ in range(20):
    y[np.random.randint(200)] = 1
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
def PolynomialLogisticRegression(degree, C, penalty='l2'):
    return Pipeline([
        ('poly', PolynomialFeatures(degree=degree)),
        ('std_scaler', StandardScaler()),
        ('log_reg', LogisticRegression(C=C, penalty=penalty))
    ])
poly_log_reg = PolynomialLogisticRegression(degree=20, C=0.1,penalty='l1')
poly_log_reg.fit(X_train, y_train)
poly_log_reg.score(X_test, y_test)
```

6. OvR & OvO

OvR：n个类别进行n次分类

OvO：n个类别进行C(n,2)次分类

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(multi_class="multinomial",solver="newton-cg")#默认OvR, multinomial等价于
OvO
log_reg.fit(X_train, y_train)
log_reg.score(X_test, y_test)
```

```
from sklearn.multiclass import OneVsRestClassifier
from sklearn.multiclass import OneVsOneClassifier
ovr = OneVsRestClassifier(log_reg)
ovr.fit(X_train, y_train)
ovr.score(X_test, y_test)
```

7. 评价分类结果

> *混淆矩阵*

真实\预测	0	1
0	预测negtive正确TN	预测positive错误FP
1	预测negtive错误FN	预测positive正确TP

> *精准率与召回率*

精准率：$precision = \frac{TP}{TP+FP}$

召回率：$recall = \frac{TP}{TP+FN}$

```
import numpy as np
from sklearn import datasets
digits = datasets.load_digits()
X = digits.data
y = digits.target.copy()
y[digits.target==9] = 1
y[digits.target!=9] = 0
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=666)
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
log_reg.score(X_test, y_test)
y_log_predict = log_reg.predict(X_test)
def TN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 0) & (y_predict == 0))
def FP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 0) & (y_predict == 1))
def FN(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 1) & (y_predict == 0))
def TP(y_true, y_predict):
    assert len(y_true) == len(y_predict)
    return np.sum((y_true == 1) & (y_predict == 1))

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)]
    ])

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

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
```

sklearn中的实现

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
confusion_matrix(y_test, y_log_predict)
precision_score(y_test, y_log_predict)
```

股票预测：注重精准率

病人诊断：注重召回率

> *F1 Score*

$F1=\frac{2\cdot precision\cdot recall}{precision+recall}$(调和平均值)

调和平均值，有一个指标比较小，整体值也偏小

```
from sklearn.metrics import f1_score
print(f1_score(y_test, y_log_predict))
```

调整分类阈值

决策边界：$\theta^T\cdot x_b=threshold$ 默认为0

```
decision_scores = log_reg.decision_function(X_test)#(默认decision_scores=0)
y_log_predict2 = np.array(decision_scores >= 5, dtype='int')
```

提高阈值，往往会提高精准率，降低召回率。

> *精准率、召回率曲线*

```
precisions = []
recalls = []
thresholds = np.arange(np.min(decision_scores), np.max(decision_scores), 0.1)
for threshold in thresholds:
    y_predict = np.array(decision_scores >= threshold, dtype='int')
    precisions.append(precision_score(y_test, y_predict))
    recalls.append(recall_score(y_test, y_predict))

plt.plot(thresholds, precisions)
plt.plot(thresholds, recalls)
plt.plot(precisions, recalls) #查看精准率、召回率制约曲线
plt.show()
```

sklearn中的封装

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_test, decision_scores)
plt.plot(thresholds, precisions[:-1]) #thresholds比precision少一个数据
plt.plot(thresholds, recalls[:-1])
plt.show()
```

> *ROC曲线*

$TPR = \frac{TP}{TP+FN}$ 即召回率

$FPR = \frac{FP}{TN+FP}$

```
from sklearn.metrics import roc_curve
fprs, tprs, thresholds = roc_curve(y_test, decision_scores)
plt.plot(fprs, tprs)
plt.show()
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test, decision_scores) //面积越大越好
```

ROC曲线对有偏数据不敏感

> *多分类问题的评价*

多分类精准率

```
from sklearn.metrics import precision_score
precision_score(y_test, y_predict, average="micro") #average默认为binary
```

多分类混淆矩阵

```
#天然支持多分类混淆矩阵
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_predict)
```

可视化混淆矩阵

```
from sklearn.metrics import confusion_matrix
cfm = confusion_matrix(y_test, y_predict)
row_sums = np.sum(cfm, axis=1)
err_matrix = cfm / row_sums
np.fill_diagonal(err_matrix, 0)
plt.matshow(err_matrix, cmap=plt.cm.gray)
plt.show()
```

- 支撑向量机

1. hard margin

直线方程$w^Tx+b=0$

$$\begin{cases} \frac{w^Tx^{(i)}+b}{||w||d} \geq 1 \quad \forall y^{(i)}=1 \\ \frac{w^Tx^{(i)}+b}{||w||d} \leq -1 \quad \forall y^{(i)}=-1 \end{cases}$$

$||w|| = \sqrt{w_1^2+w_2^2+\cdots+w_n^2}$

$y^{(i)}(w^Tx^{(i)}+b) \geq 1$

有条件的最优化问题：$max: \frac{|w^Tx+b|}{||w||} \to min: \frac{1}{2}||w||^2 \ (\text{st.} \ y^{(i)}(w^Tx^{(i)}+b) \geq 1)$

2. soft margin与正则化

$y^{(i)}(w^Tx^{(i)}+b) \geq 1-\zeta_i \quad \zeta_i \geq 0$

$min: \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\zeta_i$ L1正则 (L2正则：$\frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\zeta_i^2$)

C越大，容错空间越小

3. Sklearn中的使用

```python
from sklearn import datasets
X, y = datasets.make_moons(noise=0.15, random_state=666)
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.svm import LinearSVC
from sklearn.pipeline import Pipeline
def PolynomialSVC(degree, C=1.0):
    return Pipeline([
        ("poly", PolynomialFeatures(degree=degree)),
        ("std_scaler", StandardScaler()),
        ("linerSVC", LinearSVC(C=C))
    ])
poly_svc = PolynomialSVC(degree=3)
poly_svc.fit(X, y)
```

使用核函数

```python
from sklearn.svm import SVC
def PolynomialKernelSVC(degree, C=1.0):
    return Pipeline([
        ("std_scaler", StandardScaler()),
        ("kernelSVC", SVC(kernel="poly", degree=degree, C=C))
    ])
```

4. 核函数

> *多项式核函数：*

$K(x,y) = (x\cdot y+1)^2 = (\sum_{i=1}^{n} x_iy_i+1)^2 = \sum_{i=1}^{n}(x_i^2)(y_i^2) + \sum_{i=2}^{n}\sum_{j=1}^{i-1}(\sqrt{2}x_ix_j)(\sqrt{2}y_iy_j) + \sum_{i=1}^{n}(\sqrt{2}x_i)(\sqrt{2}y_i)+1 = x'y'$

$x' = (x_n^2,\cdots,x_1^2,\sqrt{2}x_nx_{n-1},\cdots,\sqrt{2}x_n,\cdots,\sqrt{2}x_1,1)$

拓展：$K(x,y)=(x\cdot y+c)^d$，可以指定degree与coef

> *线性核函数：*

$K(x,y) = x\cdot y$

> *高斯核函数：*

$K(x,y)$表示x和y的点乘

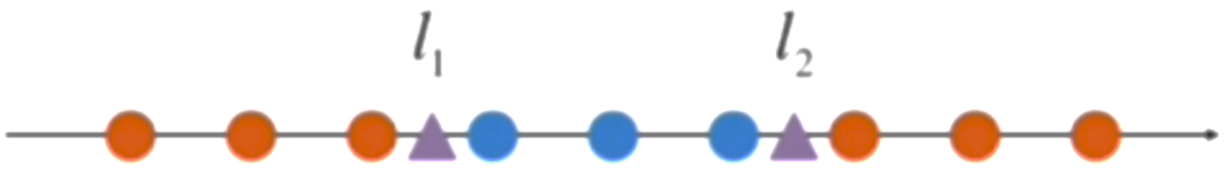
$K(x,y) = e^{-\gamma||x-y||^2}$ 将每一个样本点映射到一个无穷维的特征空间

高斯函数$g(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$

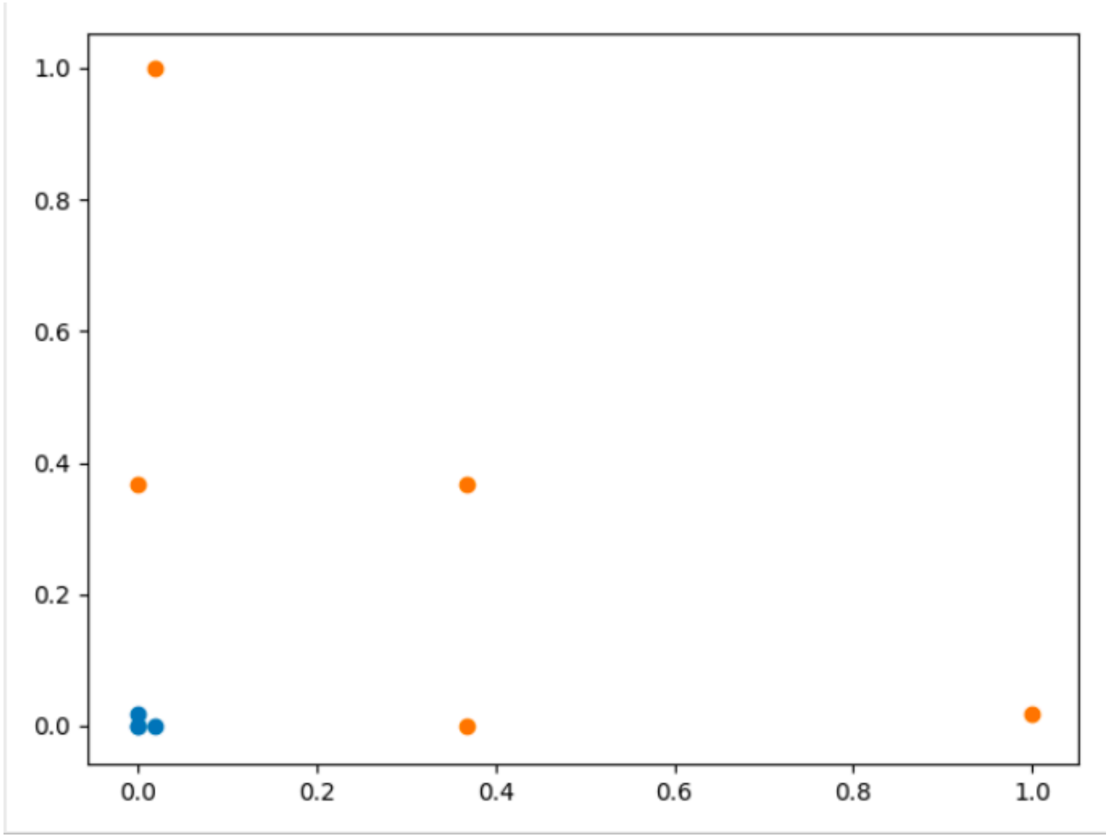
多项式特征：依靠升维使原本线性不可分的点线性可分

以两个地标点$l1, l2$为例，对其可视化理解

$x \rightarrow (e^{-\gamma||x-l_1||^2}, e^{\gamma||x-l_2||^2})$



```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(-4, 5, 1)
y = np.array((x >= -2) & (x <= 2), dtype='int')
def gaussian(x, l):
    gamma = 1.0
    return np.exp(-gamma * (x-l)**2)
l1, l2 = -1, 1
X_new = np.empty((len(x), 2))
for i, data in enumerate(x):
    X_new[i, 0] = gaussian(data, l1)
    X_new[i, 1] = gaussian(data, l2)
plt.scatter(X_new[y==0,0], X_new[y==0,1])
plt.scatter(X_new[y==1,0], X_new[y==1,1])
plt.show()
```



m*n的数据映射成了m*m的数据

```
from sklearn.svm import SVC
def RBFKernelSVC(gamma=1.0):
    return Pipeline([
        ("std_scaler", StandardScaler()),
        ("svc", SVC(kernel="rbf", gamma=gamma))
    ])
```

gamma较大：过拟合

gamma较小：欠拟合

5. 解决回归问题

```
from sklearn import datasets
from sklearn.svm import LinearSVR
from sklearn.svm import SVR
from sklean.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
boston = datasets.load_boston()
X = boston.data
y = boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y)
def StandardLinerSVR(epsilon=0.1):
    return Pipeline([
        ("std_scaler", StandardScaler()),
        ("linearSVR", LinearSVR(epsilon=epsilon))
    ])
svr = StandardLinerSVR()
svr.fit(X_test, y_test)
svr.score(X_test, y_test)
```

- 决策树

1. 信息熵

$H = -\sum_{i=1}^{k} p_i log(p_i)$

熵越大，信息不确定性越大

2. 基尼系数

$G = 1 - \sum_{i=1}^{k} p_i^2$

基尼系数越大，信息不确定性越大

3. 代码示例

```
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
from collections import Counter
from math import log
iris = datasets.load_iris()
X = iris.data[:,2:]
y = iris.target
#使用Sklearn
dt_clf = DecisionTreeClassifier(max_depth=2, criterion="entropy") #默认为基尼系数，基尼系数稍快
dt_clf.fit(X, y)
#部分实现
def split(X, y, d, value):
    index_a = (X[:,d] <= value)
    index_b = (X[:,d] > value)
    return X[index_a], X[index_b], y[index_a], y[index_b]
def entropy(y):
    counter = Counter(y)
    res = 0.0
    for num in counter.values():
        p = num / len(y)
        res += -p * log(p)
    return res
def gini(y):
    counter = Counter(y)
    res = 1.0
    for num in counter.values():
        p = num / len(y)
        res -= -p**2
    return res
def try_split(X, y):
    best_entropy = float('inf')
    best_d, best_v = -1, -1
    for d in range(X.shape[1]):
        sorted_index = np.argsort(X[:,d])
        for i in range(1, len(X)):
            if X[sorted_index[i-1], d] != X[sorted_index[i], d]:
                v = (X[sorted_index[i-1], d] + X[sorted_index[i], d]) / 2
                X_l, X_r, y_l, y_r = split(X, y, d, v)
                e = entropy(y_l) + entropy(y_r)
                #e = gini(y_l) + gini(y_r)
                if e < best_entropy:
                    best_entropy, best_d, best_v = e, d, v
    return best_entropy, best_d, best_v
```

4. 常用超参数

max_depth：2 (越高越容易过拟合)

min_sample_split： 10 (越高越不容易过拟合)

min_samples_leaf：6 (越高越不容易过拟合)

max_leaf_nodes：4 (越多越容易过拟合)

5. 绘制决策边界

```
def plot_decision_boundary(model,axis):
    x0, x1 = np.meshgrid(
      np.linspace(axis[0], axis[1], int((axis[1] - axis[0]) * 100)).reshape(-1, 1),
      np.linspace(axis[2], axis[3], int((axis[3] - axis[2]) * 100)).reshape(-1, 1)
    )
    x_new = np.c_[x0.ravel(), x1.ravel()]
    y_predict = model.predict(x_new)
    zz = y_predict.reshape(x0.shape)
    from matplotlib.colors import ListedColormap
    custom_cmap = ListedColormap(['# EF9A9A','#FFF59D','#90CAF9'])
    plt.contourf(x0, x1, zz, linewidth = 5, cmap = custom_cmap)
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn import datasets
X, y = datasets.make_moons(noise=0.25)
dt_clf = DecisionTreeClassifier(max_depth=2)

def plot_decision_boundary(model,axis):
    x0, x1 = np.meshgrid(
      np.linspace(axis[0], axis[1], int((axis[1] - axis[0]) * 100)).reshape(-1, 1),
      np.linspace(axis[2], axis[3], int((axis[3] - axis[2]) * 100)).reshape(-1, 1)
    )
    x_new = np.c_[x0.ravel(), x1.ravel()]
    y_predict = model.predict(x_new)
    zz = y_predict.reshape(x0.shape)
    from matplotlib.colors import ListedColormap
    custom_cmap = ListedColormap(['#EF9A9A','#FFF59D','#90CAF9'])
    plt.contourf(x0, x1, zz, linewidth = 5, cmap = custom_cmap)

dt_clf.fit(X, y)
plot_decision_boundary(dt_clf, axis=[-1.5,2.5,-1.0,1.5])
plt.scatter(X[y==0,0],X[y==0,1])
plt.scatter(X[y==1,0],X[y==1,1])
plt.show()
```

6. 解决回归问题

```
from sklearn.tree import DecisionTreeRegressor
from sklearn import datasets
from sklearn.model_selection import train_test_split
boston = datasets.load_boston()
X = boston.data
y = boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y)
dt_reg = DecisionTreeRegressor()
dt_reg.fit(X_train, y_train)
print(dt_reg.score(X_train, y_train))
print(dt_reg.score(X_test, y_test))
```

7. 局限性

决策边界是横平竖直的

对个别数据敏感

- 集成学习和随机森林

1. hard voting

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import VotingClassifier
X, y = datasets.make_moons(n_samples=500, noise=0.3)
X_train, X_test, y_train, y_test = train_test_split(X, y)
voting_clf = VotingClassifier(estimators=[
    ('log', LogisticRegression()),
    ('svm', SVC()),
    ('dt_clf', DecisionTreeClassifier())
], voting='hard')
voting_clf.fit(X_train, y_train)
print(voting_clf.score(X_test, y_test))
```

2. soft voting

思想：根据分类概率加权

要求：模型能够估计概率

```
voting_clf = VotingClassifier(estimators=[
    ('log', LogisticRegression()),
    ('svm', SVC(probability=True)),
    ('dt_clf', DecisionTreeClassifier())
], voting='soft')
```

3. Bagging和pasting

思想：创建若干子模型，每个模型只看样本数据一部分，子模型不需要有很高准确度，但要有差异，

取样方式：放回取样(Bagging)、不放回取样(Pasting)

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(DecisionTreeClassifier(),n_estimators=500,
                            max_samples=100,bootstrap=True)
#Sklearn中统一为BaggingClassifier，如果bootstrap为True，则为放回取样(Bagging)，否则为(Pasting)。
```

4. OOB(Out of Bag)

思想：放回取样可能导致一部分样本没有取到，可以不使用测试数据集，而是使用未被取到的样本测试，

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(DecisionTreeClassifier(),n_estimators=500,
                         max_samples=100,bootstrap=True,oob_score=True,n_jobs=-1)
bag_clf.oob_score_
```

5. 更多Bagging的探讨

思想：针对特征随机采样(max_samples设置和n_estimators一致)

```
random_subspaces_clf = BaggingClassifier(DecisionTreeClassifier(),n_estimators=500,
                         max_samples=500,bootstrap=True,
                         oob_score=True,n_jobs=-1,
                         max_features=5,bootstrap_features=True)
```

思想：既针对样本，又针对特征随机采样

```
random_patches_clf = BaggingClassifier(DecisionTreeClassifier(),n_estimators=500,
                         max_samples=100,bootstrap=True,
                         oob_score=True,n_jobs=-1,
                         max_features=5,bootstrap_features=True)
```

6. 随机森林

决策树在节点划分上，在随机的特征子集上寻找最优划分特征

```
from sklearn.ensemble import RandomForestClassifier
rf_clf = RandomForestClassifier(n_estimators=500, oob_score=True)
```

决策树在节点划分上，使用随机的特征和阈值

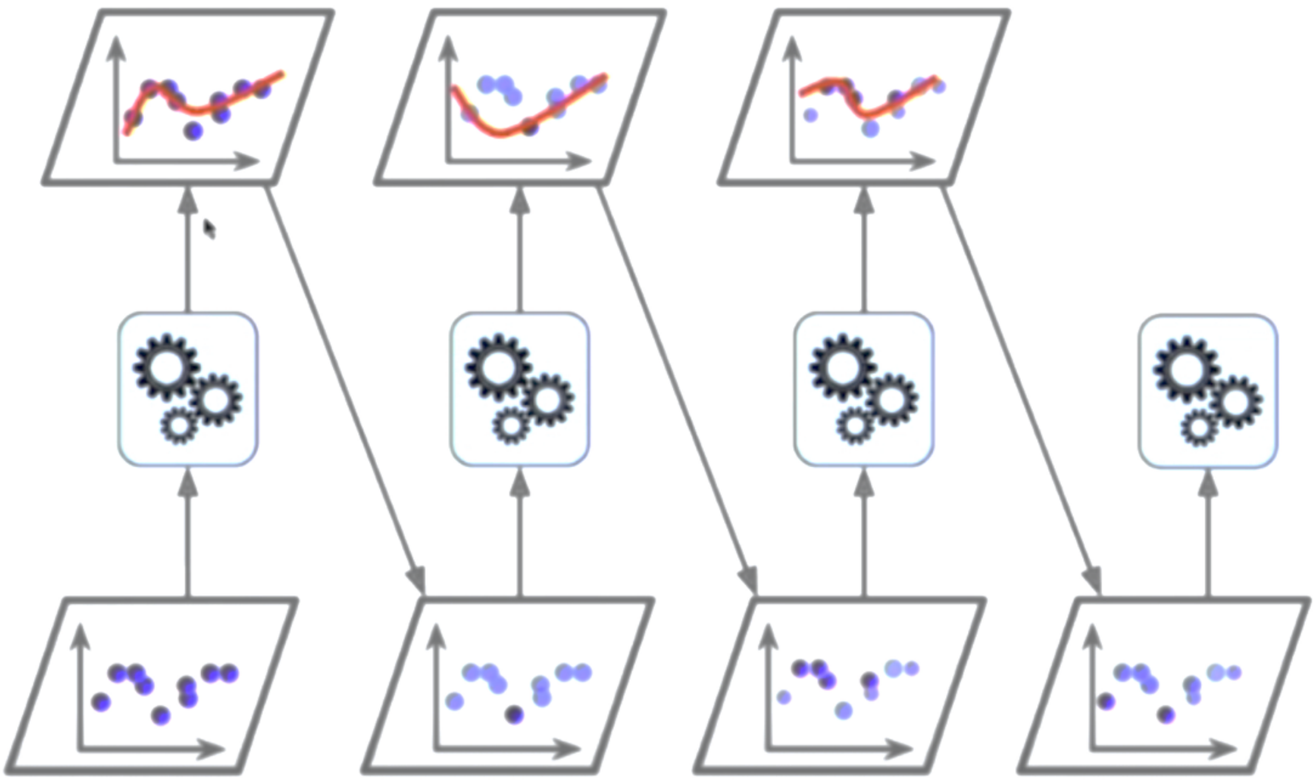
```
from sklearn.ensemble import ExtraTreesClassifier
```

集成学习也可以解决回归问题

7. Boosting

> *Ada Boostring*

图示



```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
X, y = datasets.make_moons(n_samples=500, noise=0.3)
X_train, X_test, y_train, y_test = train_test_split(X, y)
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2),n_estimators=500)
ada_clf.fit(X_train, y_train)
```

> *gradient boosting*

训练一个模型m1，产生错误e1

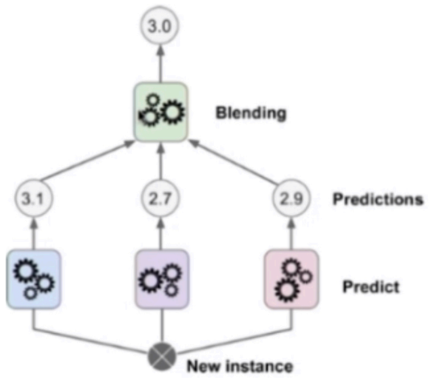
针对e1训练第二个模型m2，产生错误e2

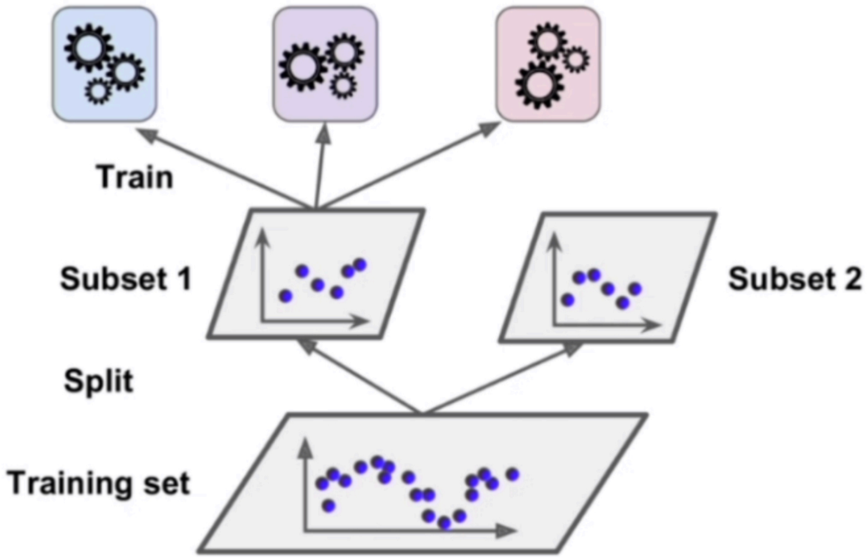
针对e2训练第三个模型m3，产生错误e3

最终预测结果为m1+m2+m3

```
from sklearn.ensemble import GradientBoostingClassifier
gb_clf = GradientBoostingClassifier(max_depth=2, n_estimators=30) #默认基于决策树
```

8. Stacking


3.0
Blending
3.1
2.7
2.9
Predictions
Predict
New instance


Train
Subset 1
Subset 2
Split
Training set