简介：

scikit-learn 是一个用于机器学习的 Python 模块，建立在 SciPy 基础之上。

主要特点：

操作简单、高效的数据挖掘和数据分析
无访问限制，在任何情况下可重新使用
建立在 NumPy、SciPy 和 matplotlib 基础上
使用商业开源协议--BSD 许可证

scikit-learn 安装:（ubuntu 版本 14.04.1）

安装依赖：
sudo apt-get install build-essential python-dev python-numpy python-setuptools python-scipy libatlas-dev libatlas3-base python-matplotlib
安装 pip
sudo apt-get install python-pip
安装 scikit-learn
sudo pip install -U scikit-learn

标准库

Classification

1.监督学习

1.1 广义线性模型：

1.1.1 普通最小二乘法：

无偏估计的

通过计算最小二乘的损失函数的最小值来求得参数得出模型

通常用在观测有误差的情况，解决线性回归问题

$\hat{y}(w,x) = w_0 + w_1 x_1 + ... + w_p x_p$

求实际观测值与预测值差的平方最小值

数学公式：$\min_w \|Xw - y\|_2^2$

是由 sklearn.linear_model 模块中的 LinearRegression 类实现回归

LinearRegression 的构造方法：

sklearn.linear_model.LinearRegression(fit_intercept=True　　#默认值为 True，表示计算随机变量，False 表示不计算随机变量

, normalize=False　　#默认值为 False，表示在回归前是否对回归因子 X 进行归一化，True 表示是

, copy_X=True)

LinearRegression 的属性有：coef_和 intercept_。coef_存储 w_1 到 w_p 的值，与 X 的维数一致。intercept_存储 w_0 的值。

LinearRegression 的常用方法有：

decision_function(X)　　#返回 X 的预测值 y

fit(X,y[,n_jobs])　　#拟合模型

get_params([deep])　　#获取 LinearRegression 构造方法的参数信息

predict(X)　　#求预测值　　#同 decision_function

score(X,y[,sample_weight])　　#计算公式为 $1-\left(\frac{(y_{true}-y_{pre})^2}{(y_{true}-y_{truemean})^2}\right)$

set_params(**params)　　#设置 LinearRegression 构造方法的参数值

参考示例：

```
from sklearn import linear_model
X= [[0, 0], [1, 1], [2, 2]]
y = [0, 1, 2]
clf = linear_model.LinearRegression()
clf.fit(X, y)
print clf.coef_
print clf.intercept_
print clf.predict([[3, 3]])
```

```
print clf.decision_function(X)
print clf.score(X, y)
print clf.get_params()
print clf.set_params(fit_intercept = False)
```

普通最小二乘法的复杂性：

假设影响因素 x 为一个 n 行 p 列的矩阵那么其算法复杂度为 $O(np^2)$ 假设 $n \geq p$

缺点：要求每个影响因素相互独立，否则会出现随机误差。

回归用于解决预测值问题

1.1.2 Ridge 回归

有偏估计的，回归系数更符合实际、更可靠，对病态数据的拟合要强于最小二乘

数学公式： $\min_w \|X_w - y\|_2^2 + \alpha \|w\|_2^2$

α >=0， α 越大，w 值越趋于一致

改良的最小二乘法，增加系数的平方和项和调整参数的积

是由 sklearn.linear_model 模块中的 Ridge 类实现

Ridge 回归用于解决两类问题：一是样本少于变量个数，二是变量间存在共线性

Ridge 的构造方法：

```
sklearn.linear_model.Ridge(alpha=1.0                    #公式中α的值，默认为 1.0
                           , fit_intercept=True
                           , normalize=False
                           , copy_X=True
                           , max_iter=None              #共轭梯度求解器的最大迭代次数
                           ,tol=0.001                   #默认值 0.001
                           , solver='auto')             #
```

Ridge 回归复杂性：同最小二乘法

使用：

```
from sklearn import linear_model
X= [[0, 0], [1, 1], [2, 2]]
y = [0, 1, 2]
clf = linear_model.Ridge(alpha = 0.1)
clf.fit(X, y)
```

```
print clf.coef_
print clf.intercept_
print clf.predict([[3, 3]])
print clf.decision_function(X)
print clf.score(X, y)
print clf.get_params()
print clf.set_params(fit_intercept = False)
```

调整参数设置（α）：通过广义交叉验证的方式（RidgeCV）设置调整参数

RidgeCV 构造方法：

```
sklearn.linear_model.RidgeCV(alphas=array([ 0.1, 1., 10. ])
                             , fit_intercept=True
                             , normalize=False
                             , scoring=None          #交叉验证发生器
                             , cv=None
                             , gcv_mode=None
                             , store_cv_values=False)
```

使用示例：

```
from sklearn import linear_model
X= [[0, 0], [1, 1], [2, 2]]
y = [0, 1, 2]
clf = linear_model.RidgeCV(alpha = [0.1, 1.0, 10.0])
clf.fit(X, y)
print clf.coef_
print clf.intercept_
print clf.predict([[3, 3]])
print clf.decision_function(X)
print clf.score(X, y)
print clf.get_params()
print clf.set_params(fit_intercept = False)
```

1.1.3 Lasso

数学公式：$\min_w \frac{1}{2n_{samples}}\|X_w - y\|_2^2 + \alpha\|w\|_1$

估计稀疏系数的线性模型

适用于参数少的情况，因其产生稀疏矩阵，可用与特征提取

实现类是 Lasso，此类用于监督分类

较好的解决回归分析中的多重共线性问题

思想：在回归系数的绝对值之和小于一个常数的约束条件下，使残差平方和最小化

使用：clf = linear_model.Lasso(alpha = 0.1)

设置调整参数（α）：

交叉验证：LassoCV（适用于高维数据集）或 LassoLarsCV（适合于样本数据比观察数据小很多）

基于模式选择的信息标准：LassoLarsIC（BIC/AIC）

1.1.4 Elastic Net

是一个使用 L1 和 L2 训练的线性模型，适合于在参数很少的情况下（如 Lasso）并保持 Ridge 性能的情况，既是多种影响因素依赖与另外一种因素。继承 Ridge 的旋转稳定性。

数学表达式：$\min_w \frac{1}{2n_{samples}}\|X_w - y\|_2^2 + \alpha\rho\|w\|_1 + \frac{\alpha(1-\rho)}{2}\|w\|_2^2$

from sklearn.linear_model import ElasticNet

enet = ElasticNet(alpha=alpha, l1_ratio=0.7)

求取α和ρ的值使用 ElasticNetCV

1.1.5 Multi-task Lasso

用于估计 y 值不是一元的回归问题

用于估计联合多元回归问题的稀疏系数，y 是一个 2 维矩阵（n_samples,n_tasks）。对于所有的回归问题，选的的因素必须相同，也叫 tasks。

使用：clf = linear_model.MultiTaskLasso(alpha=0.1)

1.1.6 Least Angle Regression

是一个计算高维数据的回归算法。实现类是 sklearn.linear_model 中的 Lars 类

优点：

当维数大于点数的时候在上下文中是有效的

复杂度和最小二乘相同，但计算很快

产生一个分片解决方案路径用于交叉验证或相似的调整模型方法

如果两个变量响应相关，则会有相同的增长率

很融合修改变成其他的评估方法

缺点：

噪声敏感

使用：clf = linear_model.Lars(n_nonzero_coefs=1)

1.1.7 LARS Lasso

是一个使用 LARS 算法实现的 lasso 模型。

使用：clf = linear_model.LassoLars(alpha=0.01)

1.1.8 OMP（Orthogonal Matching Pursuit）

通过固定的非零元素数得到最优解

OMP 基于贪心算法实现，每一步的原子项都与残余高度相关。

使用：

omp = OrthogonalMatchingPursuit(n_nonzero_coefs=n_nonzero_coefs)

omp_cv = OrthogonalMatchingPursuitCV()

1.1.9 贝叶斯回归（Bayesian Regression）

可以再估计过程包含正则化参数，参数可以手动设置，用于估计概率回归问题

优点：

适用于手边数据

可用于在估计过程中包含正规化参数

缺点：

耗时

Bayesian Ridge Regression：BayesianRidge 用于估计回归问题的概率模型

用法：clf = linear_model.BayesianRidge()　By default

$\alpha_1 = \alpha_2 = \lambda_1 = \lambda_2 = 1.e^{-6}$

ARD（Automatic Relevance Determination ）：类似于 Bayesian Ridge Regression，但可产生稀疏的 w 值

用法：clf = ARDRegression(compute_score=True)

1.1.10 逻辑回归

可以做概率预测，也可用于分类

仅能用于线性问题

通过计算真实值与预测值的概率，然后变换成损失函数，求损失函数最小值来计算模型参数从而得出模型

使用：

```
clf_l1_LR = LogisticRegression(C=C, penalty='l1', tol=0.01)
clf_l2_LR = LogisticRegression(C=C, penalty='l2', tol=0.01)
clf_l1_LR.fit(X, y)
clf_l2_LR.fit(X, y)
```

1.1.11 SGD（Stochastic Gradient Descent ）

用于样本数据非常大的情况

由 SGDClassifier 和 SGDRegressor

使用：

```
clf = linear_model.SGDRegressor()
clf = linear_model.SGDClassifier()
```

对于 SGDClassifier，当 loss=”log”时拟合成一个逻辑回归模型，当 loss=”hinge”时拟合成一个线性支持向量机

1.1.12 Perceptron

是另一个大规模学习的算法

不需要学习率

不是正则的

只有错误的时候更新模型

使用：同 SGDClassifier

1.1.13 Passive Aggressive Algorithms

不要求学习率的时候同 Perceptron，与 Perceptron 不同的是包含正则参数 C

使用：同 SGDClassifier

1.1.14 RANSAC（RANdom SAmple Consensus）

是一个从完整数据集中的一个子集中健壮估计参数的迭代算法，是非确定性算法，此算法把数据分成很多子集，根据子集估计

算法步骤：

（1）从原始数据中选择最小的随机样本，并检查数据时候有效（is_data_valid）

（2）根据（1）的随机子集拟合模型（base_estimator.fit），并检查模型是否有效（is_model_valid）

（3）通过计算估计模型的方法来分类数据

（4）如果临近样本数是最大的，并且当前的评估模型有同样的数则保存模型为最优模型。

使用：model_ransac = linear_model.RANSACRegressor(linear_model.LinearRegression())

1.1.15 多项式回归

机器学习中的常见模式，使用线性模型训练数据的非线性函数

1.2 支持向量机

拟合出来的模型为一个超平面

解决与样本维数无关，适合做文本分类

解决小样本、非线性、高维

是用于分类、回归、孤立点检测的监督学习方法的集合。

优点：

有效的高维空间

维数大于样本数的时候仍然有效

在决策函数中使用训练函数的子集

通用（支持不同的内核函数：线性、多项式、s 型等）

缺点：

不适用于特征数远大于样本数的情况

不直接提供概率估计

接受稠密和稀疏的输入

1.2.1 Classification

由 SVC、NuSVC 或 LinearSVC 实现，可进行多类分类

LinearSVC 只支持线性分类

SVC 和 NuSVC 实现一对一，LinearSVC 实现一对多

使用：

clf = svm.SVC()

lin_clf = svm.LinearSVC()

SVC、NuSVC 和 LinearSVC 均无 support_、support_vectors_和 n_support_属性

1.2.2 回归

支持向量分类的方法也支持向量回归，有 SVR 和 NuSVR，同分类一样，只是 y 向量需要是浮点数

使用：clf = svm.SVR()

1.2.3 Density estimation, novelty detection

由 OneClassSVM 实现，是非监督学习，只需要一个 X 矩阵，没有类标签

1.2.4 Complexity

$O(n_{features} \times n_{samples}^2)$and $O(n_{features} \times n_{samples}^3)$

1.3 随机梯度下降

应用于大量稀疏的机器学习问题，输入数据为稀疏的

常用于文本分类和自然语言处理

是处理大样本数据的非常有效的方法

优点：

高效

易于实施

缺点：

需要一些调整参数

对尺度特征敏感

1.3.1 Classification

使用 SGDClassifier 实现，拟合参数为 X[n_samples,n_features]和 y[N_samples]

可以实现多类分类，此时是通过多个 binary 分类实现实现方式是 one-vs-all（OVA），每一个 binary 分类把其中一个分出来，剩下的作为一个。测试的时候计算每一个分类器的得分并选择得分最高的

1.3.2 Regression

由 SGDRegressor 实现

1.3.3 复杂度

X 为(n,p)，则复杂度为$O(knp)$

1.4 最近邻

无监督的最近邻是其他学习方法的基础，监督的近邻学习分为：分类（数据有离散标记）和回归（数据有连续标记）

最近邻分类：计算待分类样本与训练样本中各个类的距离，求出距离最小的
K 紧邻是求出 k 个最小的，然后计算分别属于某一类的个数，选个数最大的类，若相等则选择跟训练集中的序列有关

近邻分类：解决离散数据
近邻回归：解决连续数据

1.4.1 无监督的最近邻

由 NearestNeighbors 实现，在 sklearn.neighbors 中，有三种算法：ball_tree（BallTree）、kd_tree（KDTree）、brute（brute-force）、auto（默认）

```
from sklearn.neighbors import NearestNeighbors
sklearn.neighbors.NearestNeighbors(n_neighbors=5            #邻居数，默认为 5
                                   , radius=1.0             #参数空间范围,默认值为 1.0
                                   , algorithm='auto'       #用于计算最近邻的算法（ball_tree、kd_tree、brute、auto）
                                   , leaf_size=30           #传递给 BallTree 或 KDTree 叶大小
                                   , metric='minkowski'
                                   , p=2
                                   , metric_params=None
                                   , **kwargs)
```

使用：

```
nbrs = NearestNeighbors(n_neighbors=2, algorithm='ball_tree').fit(X)
distances, indices = nbrs.kneighbors(X)
```

1.4.1.2 KDTree and BallTree

使用：

```
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
kdt = KDTree(X, leaf_size=30, metric='euclidean')
kdt.query(X, k=2, return_distance=False)
```

BallTree 用法同 KDTree

1.4.2 最近邻分类

是基于实例的非泛华学习，有两种不同的最近邻分类：KNeighborsClassifier 和 RadiusNeighborsClassifier（非均匀采样时比较合适）

KNeighborsClassifier：实现 k 近邻，k 是一个用户输入的整数，k 高度依赖与数据

```
sklearn.neighbors.KNeighborsClassifier(n_neighbors=5                    #邻居数，默认为 5
                                       , weights='uniform'              #用于预测的权重方法
                                       , algorithm='auto'               #用于计算最近邻的算法（ball_tree、kd_tree、brute、auto）
                                       , leaf_size=30                   #传递给 BallTree 或 KDTree 叶大小
                                       , p=2                                    #
                                       , metric='minkowski'             #用于树的度量距离
                                       , metric_params=None             #度量参数
                                       , **kwargs)
```

使用：

```
from sklearn.neighbors import KNeighborsClassifier
neigh = KNeighborsClassifier(n_neighbors=3)
neigh.fit(X, y)
```

RadiusNeighborsClassifier：实现基于给定的半径 r 内的邻居数。用于不是均匀采样的数据，不适合高维参数空间的数据

```
sklearn.neighbors.RadiusNeighborsClassifier(radius=1.0
                                            , weights='uniform'             #参数空间范围
                                            , algorithm='auto'              #用于计算最近邻的算法（ball_tree、kd_tree、brute、auto）
                                            , leaf_size=30                  #传递给 BallTree 或 KDTree 叶大小
                                            , p=2
                                            , metric='minkowski'            #用于树的度量距离
                                            , outlier_label=None            #离散群体的标签
                                            , metric_params=None            #度量参数
                                            , **kwargs)
```

使用：

```
from sklearn.neighbors import RadiusNeighborsClassifier
```

```
neigh = RadiusNeighborsClassifier(radius=1.0)
neigh.fit(X, y)
```

1.4.3 近邻回归

近邻回归用于标记数据是连续的情况，有两种不同的最近邻分类：KNeighborsRegressor 和 RadiusNeighborsRegressor

KNeighborsRegressor：实现 k 近邻，k 是一个用户输入的整数

```
sklearn.neighbors.KNeighborsRegressor(n_neighbors=5                    #邻居数，默认为 5
                                      , weights='uniform'              #用于预测的权重方法
                                      , algorithm='auto'               #用于计算最近邻的算法（ball_tree、kd_tree、brute、auto）
                                      , leaf_size=30                   #传递给 BallTree 或 KDTree 叶大小
                                      , p=2                            #
                                      , metric='minkowski'             #用于树的度量距离
                                      , metric_params=None             #度量参数
                                      , **kwargs)
```

使用：

```
from sklearn.neighbors import KNeighborsRegressor
neigh = KNeighborsRegressor(n_neighbors=3)
neigh.fit(X, y)
```

RadiusNeighborsRegressor：实现基于给定的半径 r 内的邻居数。

```
sklearn.neighbors.RadiusNeighborsRegressor(radius=1.0
                                           , weights='uniform'           #参数空间范围
                                           , algorithm='auto'            #用于计算最近邻的算法（ball_tree、kd_tree、brute、auto）
                                           , leaf_size=30                #传递给 BallTree 或 KDTree 叶大小
                                           , p=2
                                           , metric='minkowski'          #用于树的度量距离
                                           , outlier_label=None          #离散群体的标签
                                           , metric_params=None          #度量参数
```

, **kwargs)

使用：

```
from sklearn.neighbors import RadiusNeighborsRegressor
neigh = RadiusNeighborsRegressor(radius=1.0)
neigh.fit(X, y)
```

1.4.4 最近邻算法

1.4.4.1 Brute Force

搜索时通过 sklearn.neighbors 中的 algorithm 参数设置为‘brute’实现，计算是通过 sklearn.metrics.pairwise

此种方法计算效率低，D 维空间的 N 个样本时间复杂度为$O[DN^2]$

1.4.4.2 K-D Tree

为了解决 BruteForce 计算效率低的问题，通过减少计算距离所需的数据实现，思想是：如果 A 距离 B 非常远，B 距离 C 非常近，那么认为 A 与 C 距离非常远而不需要进行计算。相比于 BruteForce，时间复杂度降低为$O[DN\log(N)]$，但高维数据空间效率低

1.4.4.3 BallTree

解决 K-DTree 高维空间效率低的问题

1.4.4.4 最近邻算法选择

N 为样本数，D 为维数，k 表示邻居数

当 N《=30 的时候选择 BruteForce 比较合适

数据集结构化的时候，选择 K-D Tree 或 K-D Tree

当 k 相当于与 N 来说变得非常大时选择 BruteForce

如果查询点数少时 BruteForce

1.4.4.5 leaf_size 的影响

1)影响构建树的时间，leaf_size 越大，构建树时间越短

2)影响查询时间，leaf_size 最合适的值为 30

3)影响内存，随着 leaf_size 的增大，存储内存减小

1.4.5 Nearest Centroid Classifier

构造方法：sklearn.neighbors.NearestCentroid(metric='euclidean'

, shrink_threshold=None)

用法：

```
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 y = np.array([1, 1, 1, 2, 2, 2])
clf = NearestCentroid()
 clf.fit(X, y)
```

1.5 Gaussian Processes

GPML 是一种监督学习方法，主要用于解决回归问题，已经扩展到概率分类，但目前的研究只是一个回归练习的后处理

优点：

预测插值观测

预测是概率的，可以预测经验置信空间，改变预测值

通用

缺点：

非离散

高维空间效率低

分类仅仅是一个后处理

实现类是 GaussianProcess

构造方法：sklearn.gaussian_process.GaussianProcess(regr='constant' #回归函数返回信息

, corr='squared_exponential' #自相关信息

, beta0=None #回归权重向量

, storage_mode='full'

, verbose=False

, theta0=0.1

, thetaL=None

, thetaU=None

, optimizer='fmin_cobyla'

, random_start=1

, normalize=True

, nugget=2.2204460492503131e-15

, random_state=None)

使用：

```
import numpy as np
from sklearn.gaussian_process import GaussianProcess
X = np.array([[1., 3., 5., 6., 7., 8.]]).T
y = (X * np.sin(X)).ravel()
gp = GaussianProcess(theta0=0.1, thetaL=.001, thetaU=1.)
gp.fit(X, y)
```

1.6 Cross decomposition

包括两类算法 PLS 和 CCA，用于计算两个多变数据集的线性相关，需要拟合的多变集 X 和 Y 是 2D 数组

当预测矩阵比实际数据有更多的变量时适合用 PLS

1.7 Naive Bayes

Naive Bayes 方法是监督学习的集合基于假设每对特征之间都是独立的贝叶斯理论

朴素贝叶斯方法是基于贝叶斯理论并假设每个特征都是独立的

应用于文档分类和垃圾邮件过滤

需要训练数据比较少

朴素贝叶斯通过计算属于每个类的概率并取概率最大的类作为预测类

naive Bayes is a decent classifier, but a bad estimator

1.7.1 Gaussian Naive Bayes

实现分类的是高斯贝叶斯算法是实现类 GaussianNB

构造方法：sklearn.naive_bayes.GaussianNB

GaussianNB 类构造方法无参数，属性值有：

```
class_prior_      #每一个类的概率
theta_            #每个类中各个特征的平均
sigma_            #每个类中各个特征的方差
```

示例：

```
import numpy as np
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
Y = np.array([1, 1, 1, 2, 2, 2])
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB()
```

```
clf.fit(X, Y)
```

GaussianNB 类无 score 方法

1.7.2 Multinomial Naive Bayes

用于文本分类

用于处理多项离散数据集的 Naive Bayes 算法的类是 Multinomial NB

构造方法：sklearn.naive_bayes.MultinomialNB(alpha=1.0 #平滑参数
, fit_prior=True #学习类的先验概率
, class_prior=None) #类的先验概率

示例：

```
import numpy as np
X = np.random.randint(5, size=(6, 100))
y = np.array([1, 2, 3, 4, 5, 6])
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB()
clf.fit(X, y)
```

1.7.3 Bernoulli Naive Bayes

处理根据 multivariate Bernoulli 离散的训练和分类数据算法，实现类是 BernoulliNB

构造方法：sklearn.naive_bayes.BernoulliNB(alpha=1.0 #平滑参数
, binarize=0.0 #样本特征阈值二值比
, fit_prior=True #学习类的先验概率
, class_prior=None) #类的先验概率

示例：

```
import numpy as np
X = np.random.randint(2, size=(6, 100))
Y = np.array([1, 2, 3, 4, 4, 5])
from sklearn.naive_bayes import BernoulliNB
clf = BernoulliNB()
clf.fit(X, Y)
```

1.8 Decision Trees

是一个无参数的分类和回归的监督学习方法，目标是创建一个模型用于预测目标变量的值，通过学习从数据特征中推断出来的简单规则。

优点：

易于理解

只需要很少的准备数据

复杂度是数据点数的对数

能够同时处理数值和分类数据

能够处理多输出问题

采用白盒模型

使用统计测试可以验证模型

即使假设有点错误也可以表现很好

缺点：

可以创建复杂树但不能很好的推广

不稳定

是 NP 问题

有很难学习的概念

如果一些类占主导地位创建的树就有偏差

1.8.1 分类

实现类是 DecisionTreeClassifier，能够执行数据集的多类分类

输入参数为两个数组 X[n_samples,n_features]和 y[n_samples],X 为训练数据，y 为训练数据的标记数据

DecisionTreeClassifier 构造方法为：

```
sklearn.tree.DecisionTreeClassifier(criterion='gini'
                                    , splitter='best'
                                    , max_depth=None
                                    , min_samples_split=2
                                    , min_samples_leaf=1
                                    , max_features=None
                                    , random_state=None
                                    , min_density=None
                                    , compute_importances=None
                                    , max_leaf_nodes=None)
```

DecisionTreeClassifier 示例：

```
from sklearn import tree
X = [[0, 0], [1, 1]]
```

```
Y = [0, 1]
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, Y)
```

1.8.2 回归

实现类是 DecisionTreeRegressor，输入为 X，y 同上，y 为浮点数

DecisionTreeRegressor 构造方法为：

```
sklearn.tree.DecisionTreeRegressor(criterion='mse'
                                   , splitter='best'
                                   , max_depth=None
                                   , min_samples_split=2
                                   , min_samples_leaf=1
                                   , max_features=None
                                   , random_state=None
                                   , min_density=None
                                   , compute_importances=None
                                   , max_leaf_nodes=None)
```

DecisionTreeRegressor 示例：

```
from sklearn import tree
X = [[0, 0], [2, 2]]
y = [0.5, 2.5]
clf = tree.DecisionTreeRegressor()
clf = clf.fit(X, y)
clf.predict([[1, 1]])
```

1.8.3 多输出问题

实现类有：DecisionTreeClassifier 和 DecisionTreeRegressor.

DecisionTreeClassifier 构造方法：

```
sklearn.tree.DecisionTreeClassifier(criterion='gini'
                                    , splitter='best'
                                    , max_depth=None
                                    , min_samples_split=2
                                    , min_samples_leaf=1
                                    , max_features=None
                                    , random_state=None
```

, min_density=None

, compute_importances=None

, max_leaf_nodes=None)

DecisionTreeClassifier 示例：

```
from sklearn.datasets import load_iris
from sklearn.cross_validation import cross_val_score
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=0)
iris = load_iris()
cross_val_score(clf, iris.data, iris.target, cv=10)
```

DecisionTreeRegressor 构造方法：

```
sklearn.tree.DecisionTreeRegressor(criterion='mse'
                                   , splitter='best'
                                   , max_depth=None
                                   , min_samples_split=2
                                   , min_samples_leaf=1
                                   , max_features=None
                                   , random_state=None
                                   , min_density=None
                                   , compute_importances=None
                                   , max_leaf_nodes=None)
```

DecisionTreeRegressor 示例：

```
from sklearn.datasets import load_boston
from sklearn.cross_validation import cross_val_score
from sklearn.tree import DecisionTreeRegressor
boston = load_boston()
regressor = DecisionTreeRegressor(random_state=0)
cross_val_score(regressor, boston.data, boston.target, cv=10)
```

1.8.4 复杂度

$$O(n_{features} n_{samples} \log(n_{samples}))$$

1.9 Ensemble methods

目标是结合几种单一的估计方法进行预测值以提高通用和健壮性

两种方式：

1）多种组合方法单独进行估计，平均其得分

2）多种组合方法逐一估计

1.9.1 Bagging meta-estimator

实现类有 BaggingClassifier（用于分类） 和 BaggingRegressor（用于回归）

BaggingClassifier 构造方法：

```
sklearn.ensemble.BaggingClassifier(base_estimator=None
                                   , n_estimators=10
                                   , max_samples=1.0
                                   , max_features=1.0
                                   , bootstrap=True
                                   , bootstrap_features=False
                                   , oob_score=False
                                   , n_jobs=1
                                   , random_state=None, verbose=0)
```

BaggingClassifier 示例：

```
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
bagging = BaggingClassifier(KNeighborsClassifier(),max_samples=0.5, max_features=0.5)
```

BaggingRegressor 构造方法：

```
sklearn.ensemble.BaggingRegressor(base_estimator=None
                                  , n_estimators=10
                                  , max_samples=1.0
                                  , max_features=1.0
                                  , bootstrap=True
                                  , bootstrap_features=False
                                  , oob_score=False
                                  , n_jobs=1
                                  , random_state=None
                                  , verbose=0)
```

1.9.2 Forests of randomized trees

1.9.2.1 Random Forests

实现类是 RandomForestClassifier（分类）和 RandomForestRegressor（回归）

RandomForestClassifier 构造方法：

```
sklearn.ensemble.RandomForestClassifier(n_estimators=10
                                        , criterion='gini'
                                        , max_depth=None
                                        , min_samples_split=2
                                        , min_samples_leaf=1
                                        , max_features='auto'
                                        , max_leaf_nodes=None
                                        , bootstrap=True
                                        , oob_score=False
                                        , n_jobs=1
                                        , random_state=None
```

, verbose=0
, min_density=None
, compute_importances=None)

RandomForestClassifier 示例：

```
from sklearn.ensemble import RandomForestClassifier
X = [[0, 0], [1, 1]]
Y = [0, 1]
clf = RandomForestClassifier(n_estimators=10)
clf = clf.fit(X, Y)
```

RandomForestRegressor 构造方法：

```
sklearn.ensemble.RandomForestRegressor(n_estimators=10
                                       , criterion='mse'
                                       , max_depth=None
                                       , min_samples_split=2
                                       , min_samples_leaf=1
                                       , max_features='auto'
                                       , max_leaf_nodes=None
                                       , bootstrap=True
                                       , oob_score=False
                                       , n_jobs=1
                                       , random_state=None
                                       , verbose=0
                                       , min_density=None
                                       , compute_importances=None)
```

1.9.2.2 Extremely Randomized Trees

实现类是 ExtraTreesClassifier （分类）和 ExtraTreesRegressor（回归）

ExtraTreesClassifier 构造方法：

```
sklearn.ensemble.ExtraTreesClassifier(n_estimators=10
                                      , criterion='gini'
                                      , max_depth=None
                                      , min_samples_split=2
                                      , min_samples_leaf=1
                                      , max_features='auto'
                                      , max_leaf_nodes=None
                                      , bootstrap=False
                                      , oob_score=False
                                      , n_jobs=1
                                      , random_state=None
                                      , verbose=0
                                      , min_density=None
                                      , compute_importances=None)
```

ExtraTreesRegressor 构造方法：

```
sklearn.ensemble.ExtraTreesRegressor(n_estimators=10
```

, criterion='mse'
, max_depth=None
, min_samples_split=2
, min_samples_leaf=1
, max_features='auto'
, max_leaf_nodes=None
, bootstrap=False
, oob_score=False
, n_jobs=1
, random_state=None
, verbose=0
, min_density=None
, compute_importances=None)

1.9.3 AdaBoost

实现类是 AdaBoostClassifier（分类）和 AdaBoostRegressor（回归）

AdaBoostClassifier 构造方法：

```
sklearn.ensemble.AdaBoostClassifier(base_estimator=None
                                    , n_estimators=50
                                    , learning_rate=1.0
                                    , algorithm='SAMME.R'
                                    , random_state=None)
```

AdaBoostClassifier 示例：

```
from sklearn.cross_validation import cross_val_score
from sklearn.datasets import load_iris
from sklearn.ensemble import AdaBoostClassifier
iris = load_iris()
clf = AdaBoostClassifier(n_estimators=100)
scores = cross_val_score(clf, iris.data, iris.target)
scores.mean()
```

AdaBoostRegressor 构造方法：

```
sklearn.ensemble.AdaBoostRegressor(base_estimator=None
                                   , n_estimators=50
                                   , learning_rate=1.0
                                   , loss='linear'
                                   , random_state=None)
```

1.9.4 Gradient Tree Boosting

实现类是 GradientBoostingClassifier（分类）和 GradientBoostingRegressor（回归）

GradientBoostingClassifier 构造方法：

```
sklearn.ensemble.GradientBoostingClassifier(loss='deviance'
                                            , learning_rate=0.1
                                            , n_estimators=100
```

, subsample=1.0
, min_samples_split=2
, min_samples_leaf=1
, max_depth=3, init=None
, random_state=None
, max_features=None
, verbose=0
, max_leaf_nodes=None
, warm_start=False)

GradientBoostingClassifier 示例：

```
from sklearn.datasets import make_hastie_10_2
from sklearn.ensemble import GradientBoostingClassifier
X, y = make_hastie_10_2(random_state=0)
X_train, X_test = X[:2000], X[2000:]
y_train, y_test = y[:2000], y[2000:]
clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,max_depth=1,
random_state=0).fit(X_train, y_train)
clf.score(X_test, y_test)
```

GradientBoostingRegressor 构造方法：

```
GradientBoostingRegressor(loss='ls'
, learning_rate=0.1
, n_estimators=100
, subsample=1.0
, min_samples_split=2
, min_samples_leaf=1
, max_depth=3
, init=None
, random_state=None
, max_features=None
, alpha=0.9
, verbose=0
, max_leaf_nodes=None
, warm_start=False)
```

GradientBoostingRegressor 示例：

```
import numpy as np
from sklearn.metrics import mean_squared_error
from sklearn.datasets import make_friedman1
from sklearn.ensemble import GradientBoostingRegressor
X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
X_train, X_test = X[:200], X[200:]
y_train, y_test = y[:200], y[200:]
est = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,max_depth=1,
random_state=0, loss='ls').fit(X_train, y_train)
mean_squared_error(y_test, est.predict(X_test))
```

1.10 Multiclass and multilabel algorithms

1.10.1 Multilabel classification format

实现类是 MultiLabelBinarizer

使用示例：

```
from sklearn.datasets import make_multilabel_classification
from sklearn.preprocessing import MultiLabelBinarizer
X, Y = make_multilabel_classification(n_samples=5,
random_state=0,return_indicator=False)
Y
MultiLabelBinarizer().fit_transform(Y)
```

1.10.2 One-Vs-The-Rest

实现类是 OneVsRestClassifier.

构造方法：

```
sklearn.multiclass.OneVsRestClassifier(estimator, n_jobs=1)
```

使用示例：

```
from sklearn import datasets
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X, y = iris.data, iris.target
OneVsRestClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
```

1.10.3 One-Vs-One

实现类是 OneVsOneClassifier

构造方法：

sklearn.multiclass.OneVsOneClassifier(estimator, n_jobs=1)

使用示例：

```
from sklearn import datasets
from sklearn.multiclass import OneVsOneClassifier
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X, y = iris.data, iris.target
OneVsOneClassifier(LinearSVC(random_state=0)).fit(X, y).predict(X)
```

1.10.4 Error-Correcting Output-Codes

实现类是 OutputCodeClassifier

构造方法：

sklearn.multiclass.OutputCodeClassifier(estimator, code_size=1.5, random_state=None, n_jobs=1)

使用示例：

```
from sklearn import datasets
from sklearn.multiclass import OutputCodeClassifier
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X, y = iris.data, iris.target
clf = OutputCodeClassifier(LinearSVC(random_state=0),code_size=2, random_state=0)
clf.fit(X, y).predict(X)
```

1.11 Feature selection

特征选择可以提高决策精度，提高高维数据性能

1.11.1 Removing features with low variance

实现类是 VarianceThreshold

VarianceThreshold 构造方法是：

sklearn.feature_selection.VarianceThreshold(threshold=0.0)

VarianceThreshold 示例：

```
from sklearn.feature_selection import VarianceThreshold
X = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1], [0, 1, 0], [0, 1, 1]]
sel = VarianceThreshold(threshold=(.8 * (1 - .8)))
sel.fit_transform(X)
```

1.11.2 Univariate feature selection

实现类有：SelectKBest 、SelectPercentile 和 GenericUnivariateSelect

SelectKBest 构造方法：

sklearn.feature_selection.SelectKBest(score_func=<function f_classif at 0x34fca28>, k=10)

SelectPercentile 构造方法：

sklearn.feature_selection.SelectPercentile(score_func=<function f_classif at 0x34fca28>, percentile=10)

GenericUnivariateSelect 构造方法：

sklearn.feature_selection.GenericUnivariateSelect(score_func=<function f_classif at 0x34fca28>, mode='percentile', param=1e-05)

1.11.3 Recursive feature elimination

实现类是 RFECV

构造方法为：

sklearn.feature_selection.RFECV(estimator, step=1, cv=None, scoring=None, loss_func=None, estimator_params={}, verbose=0)

示例：

```
from sklearn.datasets import make_friedman1
from sklearn.feature_selection import RFECV
from sklearn.svm import SVR
X, y = make_friedman1(n_samples=50, n_features=10, random_state=0)
estimator = SVR(kernel="linear")
selector = RFECV(estimator, step=1, cv=5)
selector = selector.fit(X, y)
selector.support_
selector.ranking_
```

1.11.4 L1-based feature selection

1.11.4.1 Selecting non-zero coefficients

实现类是 linear_model.LogisticRegression 和 svm.LinearSVC

1.11.4.2 Randomized sparse models

实现类是 RandomizedLasso 和 RandomizedLogisticRegression

1.11.5 Tree-based feature selection

实现类是 ExtraTreesClassifier

示例：

```
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.datasets import load_iris
iris = load_iris()
X, y = iris.data, iris.target
```

```
X.shape
clf = ExtraTreesClassifier()
X_new = clf.fit(X, y).transform(X)
clf.feature_importances_
X_new.shape
```

1.12 Semi-Supervised

半监督的学习方法，用于训练数据中有一部分没有被标记，适用于少量有标记数据和大量已标记数据的的数据集

机器学习提高两种标记传播算法：LabelPropagation 和 LabelSpreading

LabelPropagation 构造方法：

```
    sklearn.semi_supervised.LabelPropagation(kernel='rbf'
                                    , gamma=20
                                    , n_neighbors=7
                                    , alpha=1
                                    , max_iter=30
                                    , tol=0.001)
```

LabelPropagation 示例：

```
    from sklearn import datasets
from sklearn.semi_supervised import LabelPropagation
label_prop_model = LabelPropagation()
iris = datasets.load_iris()
random_unlabeled_points = np.where(np.random.random_integers(0, 1,size=len(iris.target)))
labels = np.copy(iris.target)
labels[random_unlabeled_points] = -1
label_prop_model.fit(iris.data, labels)
```

LabelSpreading 构造方法：

```
    sklearn.semi_supervised.LabelSpreading(kernel='rbf'
                                    , gamma=20
                                                , n_neighbors=7
                                    , alpha=0.2
                                    , max_iter=30
                                    , tol=0.001)
```

LabelSpreading 示例：

```
    from sklearn import datasets
from sklearn.semi_supervised import LabelSpreading
label_prop_model = LabelSpreading()
iris = datasets.load_iris()
random_unlabeled_points = np.where(np.random.random_integers(0, 1,size=len(iris.target)))
labels = np.copy(iris.target)
labels[random_unlabeled_points] = -1
label_prop_model.fit(iris.data, labels)
```

1.13 Linear and quadratic discriminant analysis

线性判别分析（LDA）和二次判别分析（QDA）是两个经典的分类，因为他们的解释闭合的便于计算，LAD 适合于判别线性边界，QDA 适合于判别二次边界。

LDA 构造方法：

```
    sklearn.lda.LDA(n_components=None
                    , priors=None)
```

LDA 示例：

```
    import numpy as np
from sklearn.lda import LDA
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 y = np.array([1, 1, 1, 2, 2, 2])
clf = LDA()
clf.fit(X, y)
```

QDA 构造方法：

```
    sklearn.qda.QDA(priors=None
                    , reg_param=0.0)
```

QDA 示例：

```
    from sklearn.qda import QDA
import numpy as np
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
y = np.array([1, 1, 1, 2, 2, 2])
clf = QDA()
clf.fit(X, y)
```

LDA 可用于将维。

1.14 Isotonic regression

拟合非递归数据集，实现类为 IsotonicRegression

构造方法：

```
    sklearn.isotonic.IsotonicRegression(y_min=None          #可选，默认为 None
                                        , y_max=None      #可选，默认为 None
                                        , increasing=True
                                        , out_of_bounds='nan')
```

2.

2.3 Clustering

输入数据可以使各种各样的种矩阵

2.3.2 K-means

要求聚类的数目被指定，能很好的扩展到大数据样本，并且应用到不同领域的大范围领域中。
实现类是： KMeans
KMeans 构造方法：

```
sklearn.cluster.KMeans(n_clusters=8
                       , init='k-means++
                       ', n_init=10
                       , max_iter=300
                       , tol=0.0001
                       ,precompute_distances=True
                       , verbose=0
                       , random_state=None
                       , copy_x=True
                       , n_jobs=1)
```

参考示例：

```
from sklearn.cluster import KMeans
X = [[0],[1],[2]]
y = [0,1,2]
clf = KMeans(n_clusters=2)
clf.fit(X,y)
```

2.3.2.1 Mini Batch K-Means
是 KMeans 的一种变换，目的为了减少计算时间
实现类是 MiniBatchKMeans
MiniBatchKMeans 构造方法：

```
sklearn.cluster.MiniBatchKMeans(n_clusters=8
                                , init='k-means++'
                                , max_iter=100
                                , batch_size=100
                                , verbose=0
                                ,compute_labels=True
                                , random_state=None
                                , tol=0.0
                                , max_no_improvement=10
                                , init_size=None
                                , n_init=3
                                ,reassignment_ratio=0.01)
```

参考示例：

```
from sklearn.cluster import MiniBatchKMeans
X= [[1],[2],[3]]
mbk = MiniBatchKMeans(init='k-means++', n_clusters=3, batch_size=batch_size,
                      n_init=10, max_no_improvement=10, verbose=0)
mbk.fit(X)
```

2.3.3 Affinity Propagation

创建集群通过在成对的样本间传递消息

实现类是 AffinityPropagation

AffinityPropagation 构造方法：

```
sklearn.cluster.AffinityPropagation(damping=0.5
                                    , max_iter=200
                                    , convergence_iter=15
                                    , copy=True
                                    ,preference=None
                                    , affinity='euclidean'
                                    , verbose=False)
```

参考示例：

```
from sklearn.cluster import AffinityPropagation
X= [[1],[2],[3]]
af = AffinityPropagation(preference=-50).fit(X)
```

2.3.4 Mean Shift

实现类是：MeanShift

MeanShift 构造方法：

```
sklearn.cluster.MeanShift(bandwidth=None
                          , seeds=None
                          , bin_seeding=False
                          , min_bin_freq=1
                          ,cluster_all=True)
```

参考示例：

```
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.datasets.samples_generator import make_blobs
centers = [[1, 1], [-1, -1], [1, -1]]
X, _ = make_blobs(n_samples=10000, centers=centers, cluster_std=0.6)
bandwidth = estimate_bandwidth(X, quantile=0.2, n_samples=500)
ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
ms.fit(X)
```

2.3.5 Spectral clustering

实现类：SpectralClustering

SpectralClustering 构造方法：

```
sklearn.cluster.SpectralClustering(n_clusters=8
                                   , eigen_solver=None
                                   , random_state=None
```

, n_init=10,gamma=1.0
, affinity='rbf'
, n_neighbors=10
, eigen_tol=0.0
, assign_labels='kmeans'
, degree=3
, coef0=1
,kernel_params=None)

2.3.6 Hierarchical clustering

2.3.7 DBSCAN

2.3.8 Clustering performance evaluation

2.5 Decomposing signals in components (matrix factorization problems)

2.5.1 PCA

3.Model selection and evaluation

3.1 Cross-validation: evaluating estimator performance

把数据集自动分成训练集和测试集的方法是 train_test_split
参考示例：

```
import numpy as np
from sklearn import cross_validation
from sklearn import datasets
from sklearn import svm
iris = datasets.load_iris()
print iris.data.shape, iris.target.shape
X_train, X_test, y_train, y_test = cross_validation.train_test_split(iris.data, iris.target, test_size=0.4, random_state=0)
print X_train.shape, y_train.shape
print X_test.shape, y_test.shape
clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
```

```
clf.score(X_test, y_test)
```

3.3.1 Computing cross-validated metrics
调用 cross_val_score 实现
参考示例：

```
import numpy as np
from sklearn import cross_validation
from sklearn import datasets
from sklearn import svm
iris = datasets.load_iris()
print iris.data.shape, iris.target.shape
X_train, X_test, y_train, y_test = cross_validation.train_test_split(iris.data, iris.target, test_size=0.4, random_state=0)
print X_train.shape, y_train.shape
print X_test.shape, y_test.shape
clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
scores = cross_validation.cross_val_score(clf, iris.data, iris.target, cv=5)
clf.score(X_test, y_test)
```

3.1.2 Cross validation iterators
3.1.2.1 K-fold
把所有的样本分成 k 组
实现类是 KFold
KFold 构造方法：

```
sklearn.cross_validation.KFold(n
                               , n_folds=3
                               , indices=None
                               , shuffle=False
                               , random_state=None)
```

参考示例：

```
import numpy as np
from sklearn.cross_validation import KFold
kf = KFold(4, n_folds=2)
for train, test in kf:
    print("%s %s" % (train, test))
```

3.1.2.2 Stratified k-fold

实现类：StratifiedKFold

```
sklearn.cross_validation.StratifiedKFold(y
                                         , n_folds=3
                                         , indices=None
                                         , shuffle=False
                                         , random_state=None)
```

参考示例：

```
import numpy as np
from sklearn.cross_validation import StratifiedKFold
labels = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
skf = StratifiedKFold(labels, 3)
for train, test in kf:
    print("%s %s" % (train, test))
```

3.1.2.3 Leave-One-Out - LOO

实现类 LeaveOneOut

LeaveOneOut 构造方法：

```
sklearn.cross_validation.LeaveOneOut(n
                                     , indices=None)
```

参考示例：

```
from sklearn.cross_validation import LeaveOneOut
loo = LeaveOneOut(4)
for train, test in loo:
    print("%s %s" % (train, test))
```

3.1.2.4 Leave-P-Out - LPO

实现类：LeavePOut

LeavePOut 构造方法：

```
sklearn.cross_validation.LeavePOut(n
                                   , p
                                   , indices=None)
```

参考示例：

```
from sklearn.cross_validation import LeavePOut
lpo = LeavePOut(4, p=2)
for train, test in lpo:
    print("%s %s" % (train, test))
```

3.1.2.5 Leave-One-Label-Out - LOLO

实现类：LeaveOneLabelOut

LeaveOneLabelOut 构造方法：

```
sklearn.cross_validation.LeaveOneLabelOut(labels
                                          , indices=None)
```

参考示例：

```
from sklearn.cross_validation import LeaveOneLabelOut
labels = [1, 1, 2, 2]lolo = LeaveOneLabelOut(labels)

for train, test in lolo:
    print("%s %s" % (train, test))
```

3.1.2.6 Leave-P-Label-Out
实现类：LeavePLabelOut
LeavePLabelOut 构造方法：

```
sklearn.cross_validation.LeavePLabelOut(labels
                                        , p
                                        , indices=None)
```

参考示例：

```
from sklearn.cross_validation import LeavePLabelOut
labels = [1, 1, 2, 2, 3, 3]lplo = LeavePLabelOut(labels, p=2)

for train, test in lplo:
    print("%s %s" % (train, test))
```

3.1.2.7 Random permutations cross-validation a.k.a. Shuffle & Split
实现类：ShuffleSplit
ShuffleSplit 构造方法：

```
sklearn.cross_validation.ShuffleSplit(n
                                      , n_iter=10
                                      , test_size=0.1
                                      , train_size=None
                                      , indices=None
                                      ,random_state=None
                                      , n_iterations=None)
```

参考示例：

```
from sklearn.cross_validation import ShuffleSplit

ss = cross_validation.ShuffleSplit(5, n_iter=3, test_size=0.25,
random_state=0)

for train_index, test_index in ss:
    print("%s %s" % (train_index, test_index))
```

3.2 Grid Search: Searching for estimator parameters

3.2.1 Exhaustive Grid Search
实现类是：GridSearchCV
GridSearchCV 构造方法：

```
sklearn.grid_search.GridSearchCV(estimator
```

```
                                    , param_grid
                                    , scoring=None
                                    , loss_func=None
                                    , score_func=None
                                    ,fit_params=None
                                    , n_jobs=1
                                    , iid=True
                                    , refit=True
                                    , cv=None
                                    , verbose=0
                                    , pre_dispatch='2*n_jobs')
```

使用示例：

```
from sklearn import svm, grid_search, datasets
iris = datasets.load_iris()
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svr = svm.SVC()
clf = grid_search.GridSearchCV(svr, parameters)
clf.fit(iris.data, iris.target)
```

3.2.2 Randomized Parameter Optimization

实现类 RandomizedSearchCV

RandomizedSearchCV 构造方法：

```
sklearn.grid_search.RandomizedSearchCV(estimator
                                       , param_distributions
                                       , n_iter=10
                                       , scoring=None
                                       ,fit_params=None
                                       , n_jobs=1
                                       , iid=True
                                       , refit=True
                                       , cv=None
                                       , verbose=0
                                       , pre_dispatch='2*n_jobs'
                                       , random_state=None)
```

使用示例：

```
from sklearn import svm, grid_search, datasets
from sklearn.ensemble import RandomForestClassifier
from sklearn.grid_search import RandomizedSearchCV


iris = datasets.load_iris()
X, y = iris.data, iris.target
```

```
clf = RandomForestClassifier(n_estimators=20)
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1, 11),
              "min_samples_split": sp_randint(1, 11),
              "min_samples_leaf": sp_randint(1, 11),
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}
n_iter_search = 20random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search)
random_search.fit(X, y)
```

3.3 Pipeline: chaining estimators

链接多个估计变成一个
方便：只需调用 fit 和 predict 即可
联合多个参数：
在 Pipeline 中所有的 estimators（除去最后一个）必须是一个 transformer（既是必须有 transform 方法）

3.3.1 Usage
Pipeline 构造方法：

```
sklearn.pipeline.Pipeline(steps)
```

参考示例：

```
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.decomposition import PCA
estimators = [('reduce_dim', PCA()), ('svm', SVC())]
clf = Pipeline(estimators)
clf.steps[0]
```

可用 make_pipeline 方法替代 Pipeline 类，示例如下：

```
from sklearn.pipeline import make_pipeline
from sklearn.naive_bayes import MultinomialNB
from sklearn.preprocessing import Binarizer
make_pipeline(Binarizer(), MultinomialNB())
```

3.4 FeatureUnion: Combining feature extractors

联合多个 transformer，但对数据进行 fit 的时候每个都是独立的
实现类是 FeatureUnion
FeatureUnion 构造方法：

```
sklearn.pipeline.FeatureUnion(transformer_list      #数据转换对象列表
                              , n_jobs=1            #并行运行作业数量
                              , transformer_weights=None)
```

3.4.1 Usage
参考示例：

```
from sklearn.pipeline import FeatureUnion
from sklearn.decomposition import PCA
from sklearn.decomposition import KernelPCA
estimators = [('linear_pca', PCA()), ('kernel_pca', KernelPCA())]
combined = FeatureUnion(estimators)
```

3.5. Model evaluation: quantifying the quality of predictions

模型评估有三种方法：
1）Estimator score method
2）Scoring parameter
3）Metric functions

3.5.1 The scoring parameter: defining model evaluation rules
3.5.1.1. Common cases: predefined values
设置 3.1 节中的参数 scoring 的值
scoring 的有效值分别为：
1）Classification ： 'accuracy', 'average_precision', 'f1', 'log_loss', 'precision', 'recall', 'roc_auc'
2）Clustering： 'adjusted_rand_score'
3）Regression：'mean_absolute_error', 'mean_squared_error','r2'
若 scoring 的值不为以上值则会出错

3.5.1.2. Defining your scoring strategy from score functions

3.5.2 Function for prediction-error metrics
测量预测误差
测量预测误差的方法在 sklearn.metric 模块中提提供
sklearn.metric 模块中以_score 结尾的返回值越大说明误差越小，以_error 或_loss 结尾的返回值越小说明误差越小

3.5.2.1 Classification metrics

3.5.2.1.1 Accuracy score

计算准确度得分由 accuracy_score 实现

accuracy_score 原型：

```
accuracy_score(y_true                    #实测值
               , y_pred                  #预测值
               , normalize=True          #默认为 True 表示返回值为准确率，如果为 False 则返回正确的样本个数
               , sample_weight=None)     #样本权重
```

accuracy_score 返回值：

1）若 normalize=True 则返回准确率

2）若 normalize=False 则返回正确的样本个数

使用示例：

```
import numpy as np
from sklearn.metrics import accuracy_score
y_pred = [0, 2, 1, 3]
y_true = [0, 1, 2, 3]
print accuracy_score(y_true, y_pred)
print accuracy_score(y_true, y_pred, normalize=False)
```

多标签情况有问题：accuracy_score(np.array([[0.0, 1.0], [1.0, 1.0]]), np.ones((2, 2)))

3.5.2.1.2 Confusion matrix

使用示例：

```
from sklearn.metrics import confusion_matrix
y_pred = [2, 0, 2, 2, 0, 1]
y_true = [0, 0, 2, ,2, 0, 2]
print confusion_matrix(y_true, y_pred)
```

3.5.2.1.3 Classification report

实现方法 classification_report

classification_report 原型：

```
sklearn.metrics.classification_report(y_true                    #实测值
                                      , y_pred                  #预测值
                                      , labels=None             #标签
                                      , target_names=None       # 分类名称
                                      , sample_weight=None)     #样本权重
```

使用示例：

```
from sklearn.metrics import classification_report
y_pred = [0, 1, 2, 2, 0]
```

```
y_true = [0, 0, 2, ,2, 0]
target_names = ['class 0','class 1','class 2']
print classification_report(y_true, y_pred, target_names = target_names)
```

3.5.2.1.4 Hamming loss
计算两个样本集中对应位置不相等的数据的数量所占的比例
实现方法是： hamming_loss
hamming_loss 原型：

```
sklearn.metrics.hamming_loss(y_true
                             , y_pred
                             , classes=None)
```

使用示例：

```
from sklearn.metrics import hamming_loss
y_pred = [1, 2, 3, 4]
y_true = [2, 2, 3, 4]
print hamming_loss(y_true, y_pred)
hamming_loss(np.array([[0.0, 1.0], [1.0, 1.0]]), np.zeros((2, 2)))
```

多标签情况，测试结果与给出结果不一致

3.5.2.1.5 Jaccard similarity coefficient score
由 jaccard_similarity_score 方法实现
jaccard_similarity_score 原型：

```
sklearn.metrics.jaccard_similarity_score(y_true
                                         , y_pred
                                         , normalize=True)
```

使用示例：

```
import numpy as np
from sklearn.metrics import jaccard_similarity_score
y_pred = [0, 2, 1, 3]
y_true = [0, 1, 2, 3]
print jaccard_similarity_score(y_true, y_pred)
print jaccard_similarity_score(y_true, y_pred, normalize=False)
print jaccard_similarity_score(np.array([[0.0, 1.0], [1.0, 1.0]]), np.ones((2, 2)))
```

3.5.2.1.6. Precision, recall and F-measures
实现方法：precision_recall_curve 和 average_precision_score
precision_recall_curve 原型：

```
sklearn.metrics.precision_recall_curve(y_true
```

, probas_pred
, pos_label=None
, sample_weight=None)

precision_recall_curve 使用示例：

```
import numpy as npfrom sklearn.metrics
import precision_recall_curve
y_true = np.array([0, 0, 1, 1])
y_scores = np.array([0.1, 0.4, 0.35, 0.8])
precision, recall, thresholds = precision_recall_curve(
y_true, y_scores)
print precision
print recall
print thresholds
```

precision_recall_curve 只能用于数据为 0 或 1 的情况

average_precision_score 原型：

sklearn.metrics.average_precision_score(y_true
, y_score
, average='macro'
, sample_weight=None)

返回值为 average_precision

average_precision_score 使用示例：

```
import numpy as np
from sklearn.metrics import average_precision_score
y_true = np.array([0, 0, 1, 1])
y_scores = np.array([0.1, 0.4, 0.35, 0.8])
print average_precision_score(y_true, y_scores)
```

average_precision_score 只能用于数据为 0 或 1 的情况

3.5.2.1.7 Hinge loss

实现方法是 hinge_loss

hinge_loss 原型：

sklearn.metrics.hinge_loss(y_true
, pred_decision
, pos_label=None
, neg_label=None)

使用示例：

```
from sklearn import svm
from sklearn.metrics import hinge_loss
```

```
X = [[0], [1]]
y = [-1, 1]
est = svm.LinearSVC(random_state=0)
est.fit(X, y)
pred_decision = est.decision_function([[-2], [3], [0.5]])
print pred_decision
print hinge_loss([-1, 1, 1], pred_decision)
```

3.5.2.1.8. Log loss
实现方法：log_loss
log_loss 原型：

```
sklearn.metrics.log_loss(y_true
                         , y_pred
                         , eps=1e-15
                         , normalize=True)
```

使用示例：

```
from sklearn.metrics import log_loss
y_true = [0, 0, 1, 1]
y_pred = [[.9, .1], [.8, .2], [.3, .7], [.01, .99]]
log_loss(y_true, y_pred)
```

3.6. Model persistence

3.6.1 Persistence example
实现持久化的类是 pickle
参考示例：

```
from sklearn import svm
from sklearn import datasets
clf = svm.SVC()
iris = datasets.load_iris()
X, y = iris.data, iris.target
clf.fit(X, y)
import pickle
s = pickle.dumps(clf)
clf2 = pickle.loads(s)
print clf2.predict(X[0])
```

3.7. Validation curves: plotting scores to evaluate models

高维空间，模型不直观，因此需要选择曲线验证来验证模型的合理性

3.7.1 Validation curve

参数值对学习分数的影响曲线

这一功能由 validation_curve 完成，示例如下

参考示例：

```
import numpy as np
from sklearn.learning_curve import validation_curve
from sklearn.datasets import load_iris
from sklearn.linear_model import Ridge
np.random.seed(0)
iris = load_iris()
X, y = iris.data, iris.target
indices = np.arange(y.shape[0])
np.random.shuffle(indices)
X, y = X[indices], y[indices]
train_scores, valid_scores = validation_curve(Ridge(), X, y, "alpha",
                                              np.logspace(-7, 3, 3))
print train_scores
print valid_scores
```

train_scores 小 valid_scores 大时最好，或者两者都大

3.7.2 Learning curve

通过改变训练样本数目学习分数的影响曲线

训练样本数目对

learning_curve

train_scores 远大于 valid_scores 比较好

参考例子：

```
import numpy as np
from sklearn.learning_curve import learning_curve
from sklearn.datasets import load_iris
from sklearn.svm import SVC
np.random.seed(0)
iris = load_iris()
X, y = iris.data, iris.target
```

```
    indices = np.arange(y.shape[0])
    np.random.shuffle(indices)
    X, y = X[indices], y[indices]
    train_sizes, train_scores, valid_scores = learning_curve(SVC(kernel='linear'), X, y, train_sizes=[50, 80, 110], cv=5)
    print train_sizes
    print train_scores
    print valid_scores
```

4.

4.2 Preprocessing data

4.2.1 标准化
标准化方法：scale
scale 构造方法：

```
sklearn.preprocessing.scale(X                    #要进行标准化的数据
                            , axis=0
                            , with_mean=True    #是否计算均值
                            , with_std=True      #是否计算标准差
                            , copy=True)
```

参考示例：

```
    from sklearn import preprocessing
    import numpy as np
    X = np.array([[ 1., -1.,  2.],
                  [ 2.,  0.,  0.],
                  [ 0.,  1., -1.]])
    X_scaled = preprocessing.scale(X)
    print X_scaled
    print X_scaled.mean(axis=0)
    print X_scaled.std(axis=0)
```

sklearn.preprocessing 模块提供 StandardScaler 用于实现 Transformer API 基于训练数据计算均值和标准差，此类适合于 sklearn.pipeline.Pipeline 早期步骤
StandardScaler 的属性为：mean_和 std_
StandardScaler 构造方法：

```
    sklearn.preprocessing.StandardScaler(copy=True
                                         , with_mean=True          #是否计算平均值
```

, with_std=True)　　#是否计算标准差

参考示例：

```
from sklearn import preprocessing
import numpy as np
X = np.array([[ 1., -1.,  2.],
              [ 2.,  0.,  0.],
              [ 0.,  1., -1.]])
scaler = preprocessing.StandardScaler().fit(X)
print scaler.mean_
print scaler.std_
scaler.transform(X)
scaler.transform([-1., 1., 0.])
```

把数据缩放到一个范围，常常是 0-1 的范围，这时可以直接使用 MinMaxScaler

MinMaxScaler 构造方法：

sklearn.preprocessing.MinMaxScaler(feature_range=(0, 1)　　#要缩放的范围，默认为 0-1
, copy=True)

MinMaxScaler 有两个属性：min_和 scale_

参考示例：

```
from sklearn import preprocessing
import numpy as np
X_train = np.array([[ 1., -1.,  2.],
                    [ 2.,  0.,  0.],
                    [ 0.,  1., -1.]])
min_max_scaler = preprocessing.MinMaxScaler()
X_train_minmax = min_max_scaler.fit_transform(X_train)
print X_train_minmax
print min_max_scaler.fit_transform([-3., -1., 4.])
print min_max_scaler.scale_
print min_max_scaler.min_
```

4.2.2 常规化

计算二次型

常规化方法：normalize

normalize 构造方法：

sklearn.preprocessing.normalize(X
, norm='l2'
, axis=1
, copy=True)

参考示例：

```
from sklearn import preprocessing
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]
X_normalized = preprocessing.normalize(X, norm='l2')
print X_normalized
```

sklearn.preprocessing 模块提供 Normalizer 用于实现 Transformer API
Normalizer 构造方法：

```
sklearn.preprocessing.Normalizer(norm='l2'
                                 , copy=True)
```

此类适合于 sklearn.pipeline.Pipeline 早期步骤
参考示例：

```
from sklearn import preprocessing
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]
normalizer = preprocessing.Normalizer().fit(X)
print normalizer.transform(X)
normalizer.transform([[-1.,  1., 0.]])
```

4.2.3 Binarization
适合下行概率估计
实现类：Binarizer （此类适合于 sklearn.pipeline.Pipeline 早期步骤）
Binarizer 构造方法：

```
sklearn.preprocessing.Binarizer(threshold=0.0        #阈值，特征小于或等于此值用 0 代替，大于用 1 代替，默认值为 0.0
                                , copy=True)
```

参考示例：

```
from sklearn import preprocessing
X = [[ 1., -1.,  2.],
     [ 2.,  0.,  0.],
     [ 0.,  1., -1.]]
binarizer = preprocessing.Binarizer().fit(X)
print binarizer.transform(X)
binarizer = preprocessing.Binarizer(threshold=1.1)
print binarizer.transform(X)
```

4.2.4 Encoding categorical features

实现编码分类特征并可用于机器学习估计的类是 OneHotEncoder

OneHotEncoder 构造方法：

```
sklearn.preprocessing.OneHotEncoder(n_values='auto'                    #每个特征值得数量（‘auto’，‘int’，‘array’），默认为 auto
                                    , categorical_features='all'    #指定被分类的特征（‘all’，‘array’，‘mask’），默认 all
                                    , dtype=<type 'float'>          #输出类型，默认是 np.float
                                    ,sparse=True)                   #若设置将返回稀疏矩阵
```

参考示例：

```
from sklearn import preprocessing
enc = preprocessing.OneHotEncoder()
enc.fit([[0, 0, 3], [1, 1, 0], [0, 2, 1], [1, 0, 2]])
print enc.transform([[0, 1, 3]]).toarray()
```

4.2.5 标签预处理

4.2.5.1 Label binarization

实现类是 LabelBinarizer

LabelBinarizer 构造方法：

```
sklearn.preprocessing.LabelBinarizer(neg_label=0                #负值标签必须编码
                                     , pos_label=1              #正值标签必须编码
                                     , sparse_output=False) #返回稀疏矩阵
```

参考示例：

```
from sklearn import preprocessing
lb = preprocessing.LabelBinarizer()
lb.fit([1, 2, 6, 4, 2])
print lb.classes_
print lb.transform([1, 6])
```

多标签实现类： MultiLabelBinarizer

MultiLabelBinarizer 构造方法：

```
sklearn.preprocessing.MultiLabelBinarizer(classes=None
                                          , sparse_output=False)
```

参考示例：

```
from sklearn import preprocessing
lb = preprocessing.MultiLabelBinarizer()
```

```
print lb.fit_transform([(1, 2), (3,)])
print lb.classes_
```

4.2.5.2 Label encoding

实现类是 LabelEncoder
编码标签的值是 0-n_classes-1,用于写出高效的 Cython 程序
参考示例：

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
le.fit([1, 2, 2, 6])
print le.classes_
print le.transform([1, 1, 2, 6])
print le.inverse_transform([0, 0, 1, 2])
```

4.2.6 Imputation of missing values

实现类 Imputer
Imputer 构造方法：

```
sklearn.preprocessing.Imputer(missing_values='NaN'          #缺失值得占位符
                              , strategy='mean'             #替换缺失值得方法（mean、
                            median、most_frequent）
                              , axis=0                      #填补缺失值的方向（0 表示
                            列、1 表示行）
                              , verbose=0
                              , copy=True)
```

参考示例：

```
from sklearn.preprocessing import Imputer
import numpy as np
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
imp.fit([[1, 2], [np.nan, 3], [7, 6]])
X = [[np.nan, 2], [6, np.nan], [7, 6]]
print(imp.transform(X))
```

4.2.7 Unsupervised data reduction

特征特别多的情况下需要减少特征

4.2.7.1 PCA

```
sklearn.decomposition.PCA(n_components=None
                          , copy=True
                          , whiten=False)
```

参考示例：

```
import numpy as np
from sklearn.decomposition import PCA
X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
pca = PCA(n_components=2)
pca.fit(X)
print(pca.explained_variance_ratio_)
```

4.2.7.2 Random projections
见 4.4
4.2.7.3 Feature agglometration
见 2.3.6

4.4 Random Projection

通过控制精度来减少数据的维数，保持两个样本之间的成对的距离，因此是基于距离的方法

4.4.1 The Johnson-Lindenstrauss lemma
random_projection.johnson_lindenstrauss_min_dim(n_samples #样本数
, eps=0.1) #失真率
参考示例：

```
from sklearn.random_projection import johnson_lindenstrauss_min_dim
print johnson_lindenstrauss_min_dim(n_samples=1e6, eps=0.5)
print johnson_lindenstrauss_min_dim(n_samples=1e6, eps=[0.5, 0.1, 0.01])
print johnson_lindenstrauss_min_dim(n_samples=[1e4, 1e5, 1e6], eps=0.1)
```

4.4.2 Gaussian random projection
实现类是 GaussianRandomProjection，要求处理的数据的 shape 为（d1,d2）
GaussianRandomProjection 构造方法：
sklearn.random_projection.GaussianRandomProjection(n_components='auto' #d2 的大小（int 或‘auto’）
, eps=0.1 #失真率
, random_state=None)
参考示例：

```
import numpy as np
from sklearn import random_projection
X = np.random.rand(100, 10000)
print X.shape
```

```
transformer = random_projection.GaussianRandomProjection().fit(X)
X_new = transformer.transform(X)
print X_new.shape
```

4.4.3 Sparse random projection
实现类是 SparseRandomProjection，要求处理的数据的 shape 为（d1,d2）
SparseRandomProjection 构造方法：

```
sklearn.random_projection.SparseRandomProjection(n_components='auto'          #d2 的大小（int 或‘auto’）
                                                 , density='auto'
                                                 , eps=0.1
                                                 ,dense_output=False
                                                 , random_state=None)
```

参考示例：

```
import numpy as np
from sklearn import random_projection
X = np.random.rand(100, 10000)
print X.shape
transformer = random_projection.SparseRandomProjection().fit(X)
X_new = transformer.transform(X)
print X_new.shape
```

回归和分类：
回归是用于预测
分类用于分类
一般来说回归不用于分类，因回归是连续的模型，而且受噪声影响比较大，但逻辑回归可用于分类
逻辑回归本质上是线性回归，把特征线性求和后再把这些连续的和通过一个 s 函数映射到 0 和 1 上