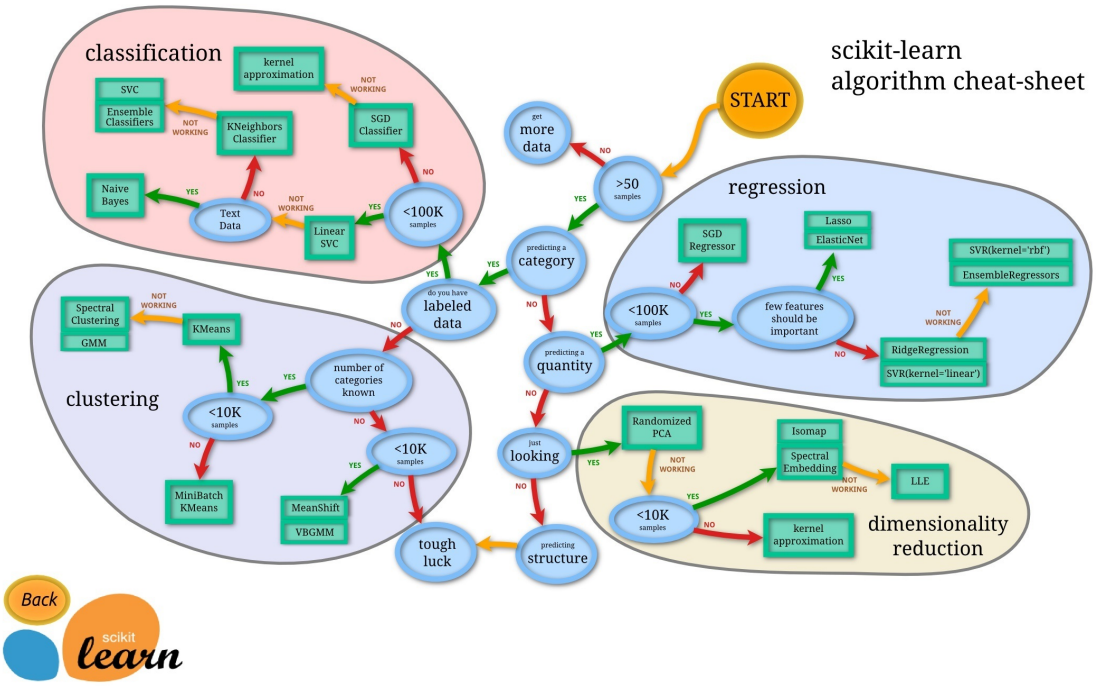
机器学习

scikit-learn

简介

- 网站：https://scikit-learn.org
- 简单有效的数据挖掘和数据分析工具
- 可供所有人访问，并可在各种环境中重复使用
- 基于 NumPy，SciPy 和 matplotlib 构建
- 开源，商业上可用 - BSD 许可证



基础步骤

```
#基础结构.py
#

from sklearn import neighbors, datasets, preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

# 加载数据
iris = datasets.load_iris()

# 划分训练集与测试集
x, y = iris.data, iris.target
x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=33)
```

```
# 数据预处理
scaler = preprocessing.StandardScaler().fit(x_train)
x_train = scaler.transform(x_train)
X_test = scaler.transform(x_test)

# 创建模型
knn = neighbors.KNeighborsClassifier(n_neighbors=5)

# 模型拟合
knn.fit(x_train, y_train)

# 交叉验证
scores=cross_val_score(knn,x_train,y_train,cv=5,scoring='accuracy')
print(scores)#每组的评分结果
print(scores.mean())

# 预测
y_pred = knn.predict(X_test)

# 评估
print(accuracy_score(y_test, y_pred))
```

回归

普通线性回归

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

$$\min_w ||Xw - y||_2^2$$

```
# LinearRegression.py
# 普通线性回归
from sklearn import linear_model
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

x, y = datasets.make_regression(n_samples=100, n_features=1, n_targets=1,
noise=10, random_state=0)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
random_state=0)

reg = linear_model.LinearRegression()
reg.fit(x_train, y_train)
print(reg.coef_, reg.intercept_)

_x = np.array([-2.5, 2.5])
_y = reg.coef_ * _x + reg.intercept_

y_pred = reg.predict(x_test)
```

```
# 平均绝对误差
print(mean_absolute_error(y_test, y_pred))
# 均方误差
print(mean_squared_error(y_test, y_pred))
# R2 评分
print(r2_score(y_test, y_pred))

plt.scatter(x_test, y_test)
plt.plot(_x, _y, linewidth=3, color="orange")
plt.show()
```

对于普通最小二乘的系数估计问题，其依赖于模型各项的相互独立性。当各项是相关的，且设计矩阵的各列近似线性相关，那么，设计矩阵会趋向于奇异矩阵，这种特性导致最小二乘估计对于随机误差非常敏感，可能产生很大的方差。

回归评估指标

Mean absolute error

$$\mathrm{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i| .$$

Mean squared error

$$\mathrm{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2 .$$

Explained variance score

$$explained\_variance(y, \hat{y}) = 1 - \frac{Var\{y - \hat{y}\}}{Var\{y\}}$$

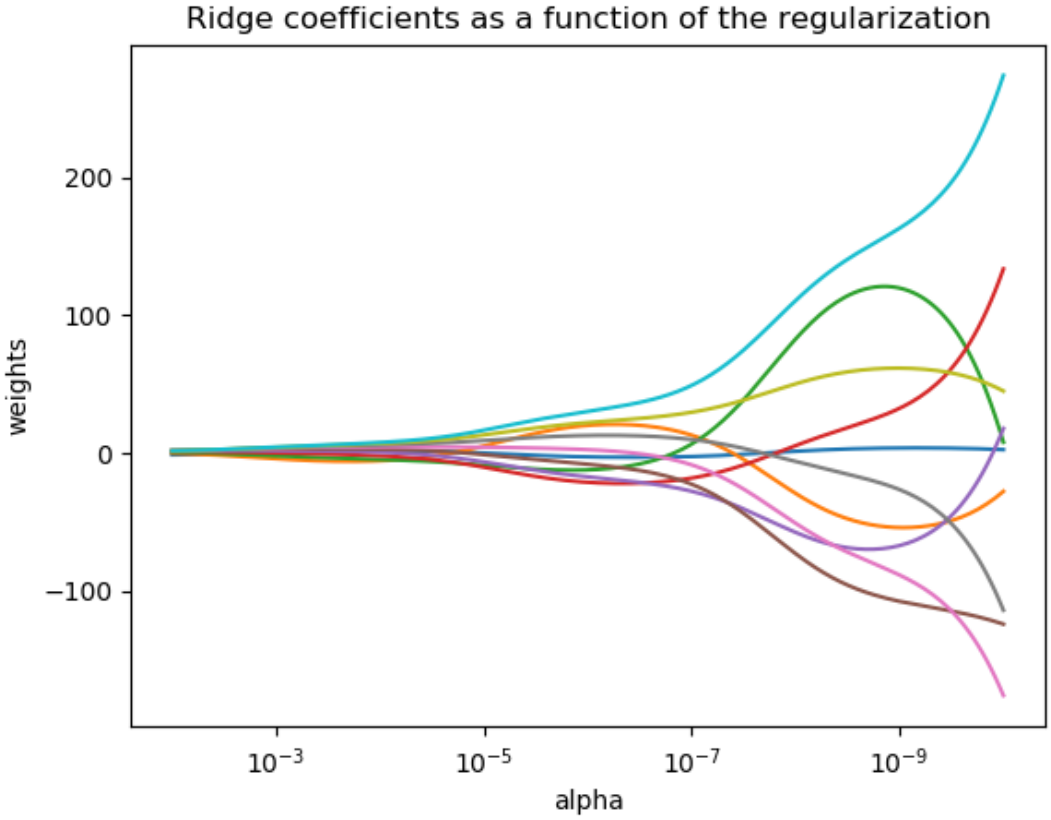
r2_score

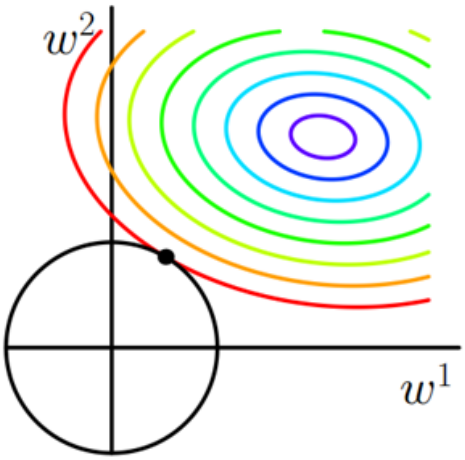
$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}$$

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$$

岭回归

$$\min_{w} ||Xw - y||_2^2 + \alpha ||w||_2^2$$

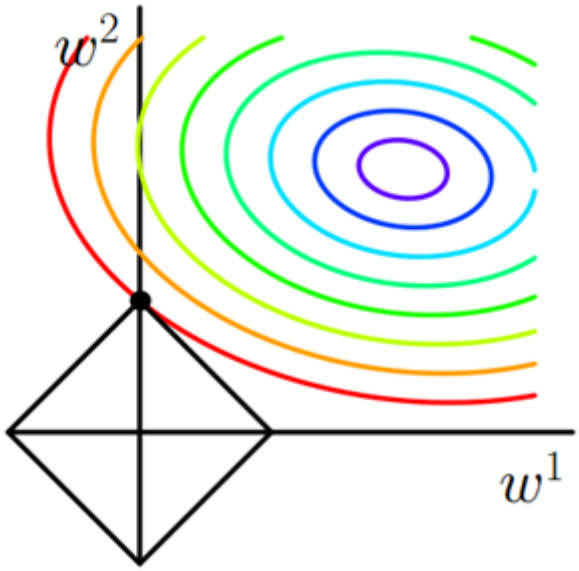





```
reg = linear_model.Ridge(alpha=0.5)
```

LASSO回归

$$\min_w \frac{1}{2n_{\text{samples}}}||Xw-y||_2^2+\alpha||w||_1$$




```
reg = linear_model.Lasso(alpha=0.1)
```

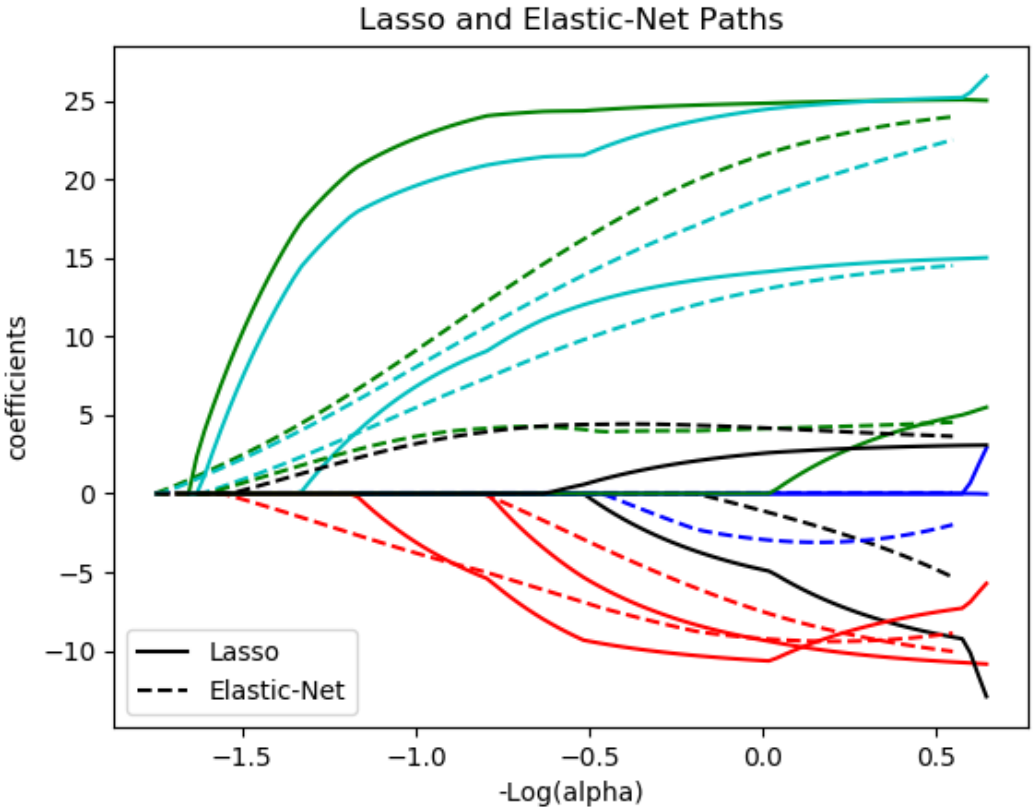
多任务岭回归

```
# MultiTask.py
# 多任务岭回归

from sklearn import linear_model
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score,
explained_variance_score

x, y = datasets.make_regression(n_samples=100, n_features=1, n_targets=2,
noise=10, random_state=0)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
random_state=0)

reg = linear_model.MultiTaskLasso(0.1)

reg.fit(x_train, y_train)
print(reg.coef_, reg.intercept_)

y_pred = reg.predict(x_test)

# 平均绝对误差
print(mean_absolute_error(y_test, y_pred))
# 均方误差
print(mean_squared_error(y_test, y_pred))
# R2 评分
print(r2_score(y_test, y_pred))
# explained_variance
print(explained_variance_score(y_test, y_pred))

_x = np.array([-2.5, 2.5])
```

```
_y = reg.predict(_x[:,None])


plt.scatter(x_test, y_test[:,0],color="green")
plt.plot(_x, _y[:,0], linewidth=3, color="orange")

plt.scatter(x_test, y_test[:,1],color="blue")
plt.plot(_x, _y[:,1], linewidth=3, color="red")
plt.show()
```

弹性网络

$$\min_{w} \frac{1}{2n_{\text{samples}}}||Xw-y||_2^2+\alpha\rho||w||_1+\frac{\alpha(1-\rho)}{2}||w||_2^2$$



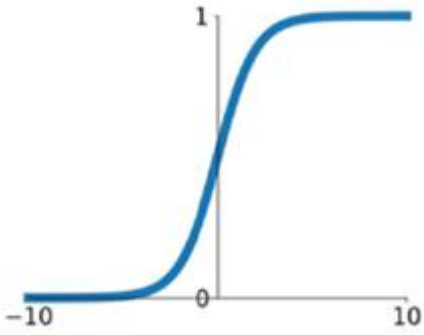

```
reg = linear_model.ElasticNet()
```

逻辑斯蒂回归

$$\min_{w,c} \frac{1-\rho}{2}w^Tw+\rho\|w\|_1+C\sum_{i=1}^{n}\log(\exp(-y_i(X_i^Tw+c))+1)$$

Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$




```
reg = linear_model.LogisticRegression()
```

贝叶斯岭回归

$$p(y|X,w,\alpha) = \mathcal{N}(y|Xw,\alpha)$$

$$p(w|\lambda) = \mathcal{N}(w|0,\lambda^{-1}\mathbf{I}_p)$$

```
reg = linear_model.BayesianRidge()
```

核岭回归

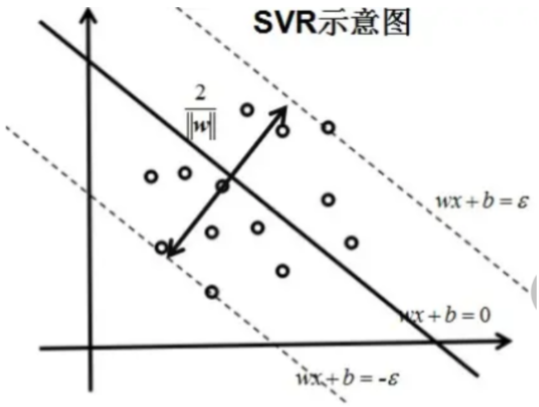
- 核
  - linear: $\langle x, x' \rangle$.
  - polynomial: $(\gamma \langle x, x' \rangle + r)^d$. d is specified by keyword `degree`, r by `coef0`.
  - rbf: $\exp(-\gamma \|x - x'\|^2)$. γ is specified by keyword `gamma`, must be greater than 0.
  - sigmoid ($\tanh(\gamma \langle x, x' \rangle + r)$), where r is specified by `coef0`.

```
import numpy as np
from sklearn.kernel_ridge import KernelRidge
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV

rng = np.random.RandomState(0)

X = 5 * rng.rand(100, 1)
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))

# kr = KernelRidge(kernel='rbf', gamma=0.4)
kr = GridSearchCV(KernelRidge(),
                  param_grid={"kernel": ["rbf", "laplacian", "polynomial",
"sigmoid"],
                              "alpha": [1e0, 0.1, 1e-2, 1e-3],
                              "gamma": np.logspace(-2, 2, 5)})
```

```
kr.fit(X, y)
print(kr.best_score_, kr.best_params_)

X_plot = np.linspace(0, 5, 100)
y_kr = kr.predict(X_plot[:, None])

plt.scatter(X, y)
plt.plot(X_plot, y_kr, color="red")
plt.show()
```

SVR

- 原理



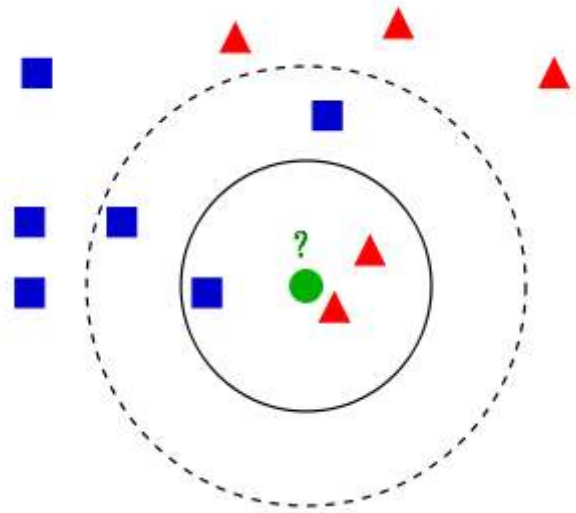
$$\begin{cases} \min \frac{1}{2}\|w\|^2 \\ s.t. \left| y_i - (wx_i + b) \right| \le \varepsilon, \quad \forall i \end{cases}$$

```
#SVR.py
#

import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVR

rng = np.random.RandomState(0)

X = 5 * rng.rand(100, 1)
y = np.sin(X).ravel()

y[::5] += 3 * (0.5 - rng.rand(X.shape[0] // 5))

svr = SVR(kernel='rbf', C=10, gamma=0.1)
svr.fit(X, y)

X_plot = np.linspace(0, 5, 100)
y_svr = svr.predict(X_plot[:, None])
```

```
plt.scatter(X, y)
plt.plot(X_plot, y_svr, color="red")
plt.show()
```

分类

K近邻

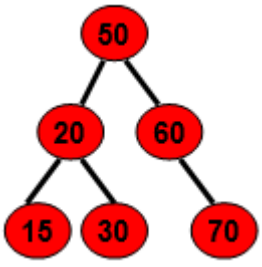


KD-TREE

转：https://blog.csdn.net/ye1215172385/article/details/80214776

kd树（k-dimensional树的简称），是一种分割k维数据空间的数据结构，主要应用于多维空间关键数据的近邻查找(Nearest Neighbor)和近似最近邻查找(Approximate Nearest Neighbor)。

其实KDTree就是二叉查找树（Binary Search Tree，BST）的变种。二叉查找树的性质如下：
1）若它的左子树不为空，则左子树上所有结点的值均小于它的根结点的值；
2）若它的右子树不为空，则右子树上所有结点的值均大于它的根结点的值；
3）它的左、右子树也分别为二叉排序树；



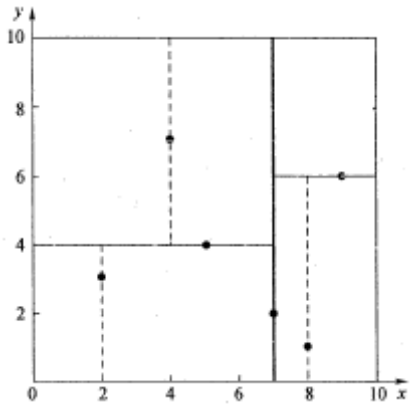
如果我们要处理的对象集合是一个K维空间中的数据集，我们首先需要确定是：怎样将一个K维数据划分到左子树或右子树？在构造1维BST树类似，只不过对于Kd树，在当前节点的比较并不是通过对K维数据进行整体的比较，而是选择某一个维度d，然后比较两个K维数据在该维度 d上的大小关系，即每次选择一个维度d来对K维数据进行划分，相当于用一个垂直于该维度d的超平面将K维数据空间一分为二，平面一边的所有K维数据 在d维度上的值小于平面另一边的所有K维数据对应维度上的值。也就是说，我们每选择一个维度进行如上的划分，就会将K维数据空间划分为两个部分，如果我 们继续分别对这两个子K维空间进行如上的划分，又会得到新的子空间，对新的子空间又继续划分，重复以上过程直到每个子空间都不能再划分为止。以上就是构造 Kd-Tree的过程，上述过程中涉及到两个重要的问题：1）每次

对子空间的划分时，怎样确定在哪个维度上进行划分；2）在某个维度上进行划分时，怎样确保建立的树尽量地平衡，树越平衡代表着分割得越平均，搜索的时间也就是越少。

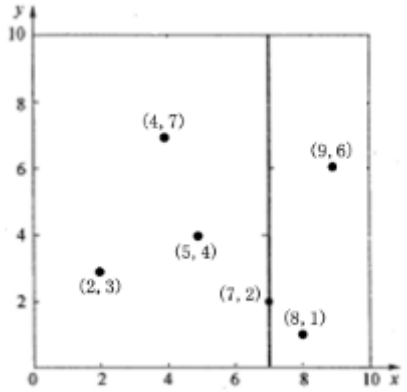
- 在哪个维度上进行划分?
  一种选取轴点的策略是median of the most spread dimension pivoting strategy，统计样本在每个维度上的数据方差，挑选出对应方差最大值的那个维度。数据方差大说明沿该坐标轴方向上数据点分散的比较开。这个方向上，进行数据分割可以获得最好的平衡。
- 怎样确保建立的树尽量地平衡?
  给定一个数组，怎样才能得到两个子数组，这两个数组包含的元素 个数差不多且其中一个子数组中的元素值都小于另一个子数组呢？方法很简单，找到数组中的中值（即中位数，median），然后将数组中所有元素与中值进行 比较，就可以得到上述两个子数组。同样，在维度d上进行划分时，划分点（pivot）就选择该维度d上所有数据的中值，这样得到的两个子集合数据个数就基本相同了。

创建

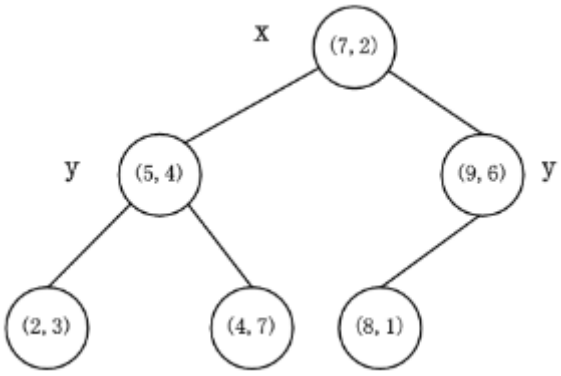
假设有6个二维数据点{（2,3），（5,4），（9,6），（4,7），（8,1），（7,2）}，数据点位于二维空间内（如下图中黑点所示）。kd树算法就是要确定图1中这些分割空间的分割线（多维空间即为分割平面，一般为超平面）。下面就要通过一步步展示kd树是如何确定这些分割线的。



1. 分别计算x，y方向上数据的方差，得知x方向上的方差最大；
2. 根据x轴方向的值2,5,9,4,8,7排序选出中值为7，所以该node中的data = （7,2）。这样，该节点的分割超平面就是通过（7,2）并垂直于x轴的直线x = 7；
3. 确定左子空间和右子空间。分割超平面x = 7将整个空间分为两部分，如下图所示。x < = 7的部分为左子空间，包含3个节点{（2,3），（5,4），（4,7）}；另一部分为右子空间，包含2个节点{（9,6），（8,1）}。



k-d树的构建是一个递归的过程。然后对左子空间和右子空间内的数据重复根节点的过程就可以得到下一级子节点（5,4）和（9,6）（也就是左右子空间的'根'节点），同时将空间和数据集进一步细分。如此反复直到空间中只包含一个数据点，如下图所示：



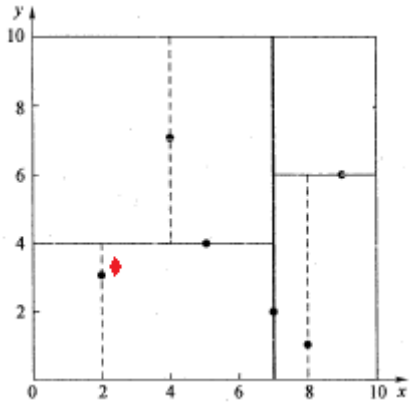
查找

1. 将查询数据Q从根结点开始，按照Q与各个结点的比较结果向下访问Kd-Tree，直至达到叶子结点。 其中Q与结点的比较指的是将Q对应于结点中的k维度上的值与中值m进行比较，若Q(k) < m，则访问左子树，否则访问右子树。达到叶子结点时，计算Q与叶子结点上保存的数据之间的距离，记录下最小距离对应的数据点，记为当前最近邻点nearest和最小距离dis。
2. 进行回溯操作，该操作是为了找到离Q更近的“最近邻点”。即判断未被访问过的分支里是否还有离Q更近的点，它们之间的距离小于dis。 如果Q与其父结点下的未被访问过的分支之间的距离小于dis，则认为该分支中存在离P更近的数据，进入该结点，进行（1）步骤一样的查找过程，如果找到更近的数据点，则更新为当前的最近邻点nearest，并更新dis。如果Q与其父结点下的未被访问过的分支之间的距离大于dis，则说明该分支内不存在与Q更近的点。回溯的判断过程是从下往上进行的，直到回溯到根结点时已经不存在与P更近的分支为止。
3. 注：判断未被访问过的树分支中是否还有离Q更近的点，就是判断"Q与未被访问的树分支的距离|Q(k) - m|"是否小于"Q到当前的最近邻点nearest的距离dis"。从几何空间上来看，就是判断以Q为中心，以dis为半径超球面是否与未被访问的树分支代表的超矩形相交。

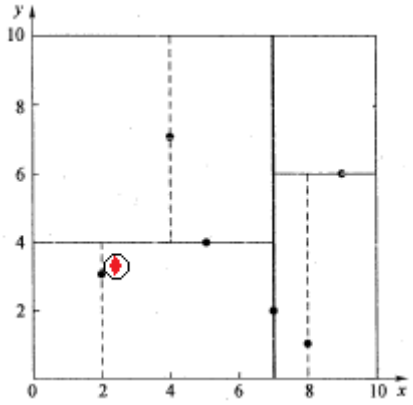
下面举两个例子来演示一下最近邻查找的过程。 假设我们的kd树就是上面通过样本集{（2,3），（5,4），（9,6），（4,7），（8,1），（7,2）}创建的。

例1：查找点Q(2.1,3.1)

如下图所示，红色的点即为要查找的点。通过图4二叉搜索，顺着搜索路径很快就能找到当前的最邻近点（2,3）。



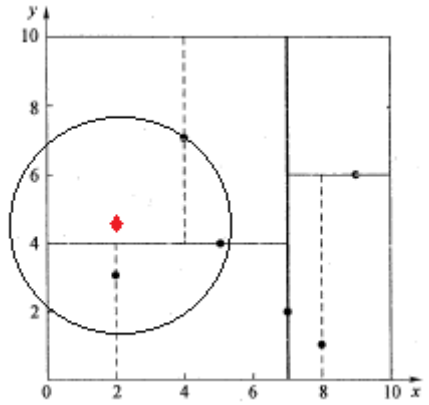
在上述搜索过程中，产生的搜索路径节点有<（7,2），（5,4），（2,3）>。为了找到真正的最近邻，还需要进行'回溯'操作，首先以（2,3）作为当前最近邻点nearest，计算其到查询点Q（2.1,3.1）的距离dis为0.1414，然后回溯到其父节点（5,4），并判断在该父节点的其他子节点空间中是否有距离查询点Q更近的数据点。以（2.1,3.1）为圆心，以0.1414为半径画圆，如图6所示。发现该圆并不和超平面y = 4交割，即这里：|Q(k) - m|=|3.1 - 4|=0.9 > 0.1414，因此不用进入（5,4）节点右子空间中去搜索。



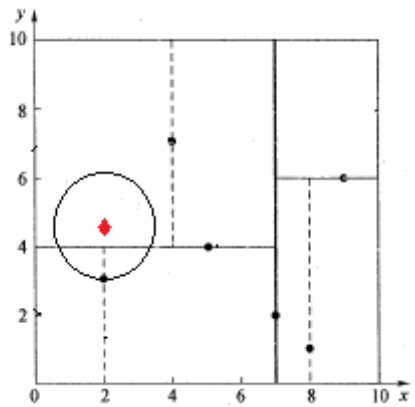
再回溯到（7,2），以（2.1,3.1）为圆心，以0.1414为半径的圆更不会与x = 7超平面交割，因此不用进入（7,2）右子空间进行查找。至此，搜索路径中的节点已经全部回溯完，结束整个搜索，返回最近邻点（2,3），最近距离为0.1414。

例2：查找点Q(2,4.5)

如下图所示，同样经过图4的二叉搜索，可得当前的最邻近点（4,7），产生的搜索路径节点有<（7,2），（5,4），（4,7）>。首先以（4,7）作为当前最近邻点nearest，计算其到查询点Q（2,4.5）的距离dis为3.202，然后回溯到其父节点（5,4），并判断在该父节点的其他子节点空间中是否有距离查询点Q更近的数据点。以（2,4.5）为圆心，以为3.202为半径画圆，如图7所示。发现该圆和超平面y = 4交割，即这里：|Q(k) - m|=|4.5 - 4|=0.5 < 3.202，因此进入（5,4）节点左子空间中去搜索。所以，将(2,3)加入到搜索路径中，现在搜索路径节点有<(7,2), (2, 3)>。同时，注意：点Q(2,4.5)与父节点(5,4)的距离也要考虑，由于这两点间的距离3.04 < 3.202，所以将(5,4)赋给nearest，并且dist=3.04。



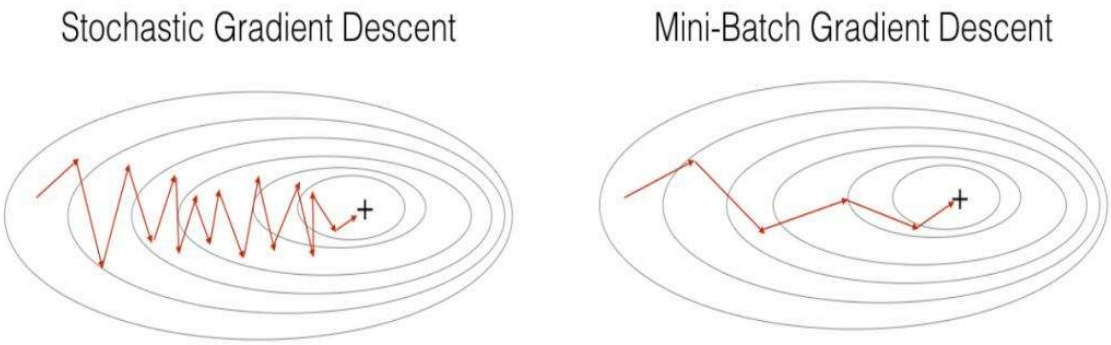
接下来，回溯至（2,3）叶子节点，点Q（2,4.5）和（2,3）的距离为1.5，比距离（5,4）要近，所以最近邻点nearest更新为(2,3)，最近距离dis更新为1.5。回溯至（7,2），如图8所示，以（2,4.5）为圆心1.5为半径作圆，并不和x = 7分割超平面交割，即这里：|Q(k) - m|=|2 - 7|=5 > 1.5。至此，搜索路径回溯完。返回最近邻点（2,3），最近距离1.5。



总结

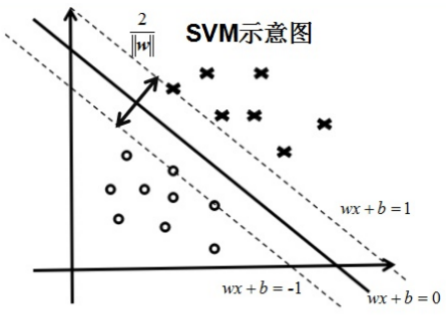
Kd树在维度较小时（比如20、30），算法的查找效率很高，然而当数据维度增大（例如：K≥100），查找效率会随着维度的增加而迅速下降。假设数据集的维数为D，一般来说要求数据的规模N满足N>>2的D次方，才能达到高效的搜索。为了能够让Kd树满足对高维数据的索引，Jeffrey S. Beis和David G. Lowe提出了一种改进算法——Kd-tree with BBF（Best Bin First），该算法能够实现近似K近邻的快速搜索，在保证一定查找精度的前提下使得查找速度较快

随机梯度下降法

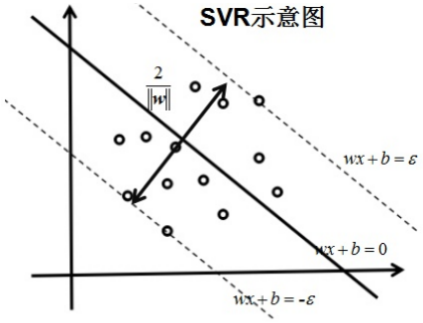



```
clf = linear_model.SGDClassifier()
```

SVM



$$\begin{cases} \min \frac{1}{2}\|w\|^2 \\ s.t.\ y_i(wx_i+b) \geq 1, \quad \forall i \end{cases}$$



$$\begin{cases} \min \frac{1}{2}\|w\|^2 \\ s.t.\ |y_i - (wx_i+b)| \leq \varepsilon, \quad \forall i \end{cases}$$

```
clf = svm.SVC()
```

数据预处理

1. 首先要明确有多少特征，哪些是连续的，哪些是类别的。
2. 检查有没有缺失值，对确实的特征选择恰当方式进行弥补，使数据完整。
3. 对连续的数值型特征进行标准化，使得均值为0，方差为1。

4. 对类别型的特征进行one-hot编码。
5. 将需要转换成类别型数据的连续型数据进行二值化。
6. 为防止过拟合或者其他原因，选择是否要将数据进行正则化。
7. 在对数据进行初探之后发现效果不佳，可以尝试使用多项式方法，寻找非线性的关系。
8. 根据实际问题分析是否需要对特征进行相应的函数转换。

标准化：去均值，方差规模化

Standardization标准化:将特征数据的分布调整成标准正太分布，也叫高斯分布，也就是使得数据的均值维0，方差为1.标准化的原因在于如果有些特征的方差过大，则会主导目标函数从而使参数估计器无法正确地去学习其他特征。标准化的过程为两步：去均值的中心化（均值变为0）；方差的规模化（方差变为1）

```
x_scale = preprocessing.scale(x)
print(x_scale)
print(x_scale.mean(0),x_scale.std(0))
```

```
scaler = preprocessing.StandardScaler().fit(x)
x_scale = scaler.transform(x)
print(x_scale)
print(x_scale.mean(0),x_scale.std(0))
```

MinMaxScaler

在MinMaxScaler中是给定了一个明确的最大值与最小值。它的计算公式如下：

X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X_scaled = X_std / (max - min) + min

MaxAbsScaler

- 原理与上面的很像，只是数据会被规模化到[-1,1]之间。也就是特征中，所有数据都会除以最大值。这个方法对那些已经中心化均值维0或者稀疏的数据有意义。
- 如果对稀疏数据进行去均值的中心化就会破坏稀疏的数据结构。虽然如此，我们也可以找到方法去对稀疏的输入数据进行转换，特别是那些特征之间的数据规模不一样的数据。

```
scaler = preprocessing.MaxAbsScaler()
x_scale = scaler.fit_transform(x)
print(x_scale)
print(x_scale.mean(0), x_scale.std(0))
```

RobustScaler

如果你的数据有许多异常值，那么使用数据的均值与方差去做标准化就不行了。在这里，你可以使用robust_scale 和 RobustScaler这两个方法。它会根据中位数或者四分位数去中心化数据。

```
scaler = preprocessing.RobustScaler()
x_scale = scaler.fit_transform(x)
print(x_scale)
print(x_scale.mean(0), x_scale.std(0))
```

Normalization

正则化是将样本在向量空间模型上的一个转换，经常被使用在分类与聚类中。函数normalize 提供了一个快速有简单的方式在一个单向量上来实现这正则化的功能。正则化有l1,l2等

```
scaler = preprocessing.Normalizer(norm="l2")
x_scale = scaler.fit_transform(x)
print(x_scale)
print(x_scale.mean(0), x_scale.std(0))
```

二值化

特征的二值化是指将数值型的特征数据转换成布尔类型的值

```
scaler = preprocessing.Binarizer(threshold=0)
x_scale = scaler.fit_transform(x)
print(x_scale)
```

OneHotEncoder独热编码

学习sklearn和kagggle时遇到的问题，什么是独热编码？为什么要用独热编码？什么情况下可以用独热编码？以及和其他几种编码方式的区别。　首先了解机器学习中的特征类别：连续型特征和离散型特征。　拿到获取的原始特征，必须对每一特征分别进行归一化，比如，特征A的取值范围是[-1000,1000]，特征B的取值范围是[-1,1].如果使用logistic回归，w1*x1*+*w2*x2，因为x1的取值太大了，所以x2基本起不了作用。所以，必须进行特征的归一化，每个特征都单独进行归一化。

独热码，在英文文献中称做 one-hot code, 直观来说就是有多少个状态就有多少比特，而且只有一个比特为1，其他全为0的一种码制。举例如下：　假如有三种颜色特征：红、黄、蓝。 在利用机器学习的算法时一般需要进行向量化或者数字化。那么你可能想令 红=1，黄=2，蓝=3. 那么这样其实实现了标签编码，即给不同类别以标签。然而这意味着机器可能会学习到“红<黄<蓝”，但这并不是我们的让机器学习的本意，只是想让机器区分它们，并无大小比较之意。所以这时标签编码是不够的，需要进一步转换。因为有三种颜色状态，所以就有3个比特。

即红色：１００，黄色: ０１０，蓝色：００１。如此一来每两个向量之间的距离都是根号2，在向量空间距离都相等，所以这样不会出现偏序性，基本不会影响基于向量空间度量算法的效果。

自然状态码为：000,001,010,011,100,101

独热编码为：000001,000010,000100,001000,010000,100000

```
# one_hot
enc = preprocessing.OneHotEncoder(n_values=4, sparse=False)
ans = enc.fit_transform([[1], [2], [3]])
print(ans)
```

弥补缺失数据

在scikit-learn的模型中都是假设输入的数据是数值型的，并且都是有意义的，如果有缺失数据是通过NAN，或者空值表示的话，就无法识别与计算了。

- missing_values: 空值的类型。默认np.nan
- strategy: 可选：mean, median, most_frequent, constant
- fill_value: 以什么值进行填补，当constant时可用。
- copy：是否创建副本

要弥补缺失值，可以使用均值，中位数，众数等等。Imputer这个类可以实现。请看：

```
imp = preprocessing.Imputer(missing_values='NaN', strategy='mean', axis=0)

#直接补值
y_imp = imp.fit_transform([[np.nan, 2], [6, np.nan], [7, 6]])
print(y_imp)
```

```
#通过学习补值
imp.fit([[1, 2], [np.nan, 3], [7, 6]])
y_imp = imp.transform([[np.nan, 2], [6, np.nan], [7, 6]])
print(y_imp)
```

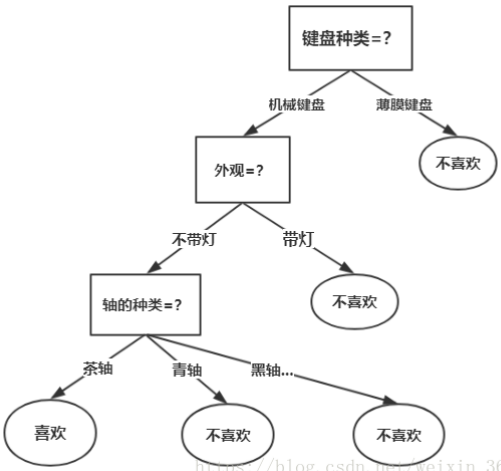
决策树与回归树

决策树

决策树模型是运用于分类以及回归的一种树结构。决策树由节点和有向边组成，一般一棵决策树包含一个根节点、若干内部节点和若干叶节点。决策树的决策过程需要从决策树的根节点开始，待测数据与决策树中的特征节点进行比较，并按照比较结果选择选择下一比较分支，直到叶子节点作为最终的决策结果。

- 内部节点：对应于一个属性测试
- 叶节点：对应于决策结果
- 根节点包含样本全集；
- 每个节点包括的样本集合根据属性测试的结果被划分到子节点中；
- 根节点到每个叶节点的路径对应对应了一个判定测试路径；

决策树的结构还是比较好理解的，如果不明白，可以看一下图中的例子，这是一个简单判断这个键盘我喜不喜欢的决策树模型



决策树学习算法：

- 特征选择
- 决策树生成
- 决策树剪枝

特征选择也即选择最优划分属性，从当前数据的特征中选择一个特征作为当前节点的划分标准。我们希望在不断划分的过程中，决策树的分支节点所包含的样本尽可能属于同一类，即节点的“纯度”越来越高。而选择最优划分特征的标准不同，也导致了决策树算法的不同。

为了找到最优的划分特征，我们需要先了解一些信息论的知识：

- 信息熵(information entropy)
- 信息增益(information gain)
- 信息增益率(information gain ratio)
- 基尼指数(Gini index)

熵：

在信息论和概率统计中，熵(entropy)是表示随机变量不确定性的度量，设X是一个取有限值的离散随机变量，其概率分布为

$$P(X=x_i)=p_i,\qquad i=1,2,\ldots,n$$

则随机变量XX的熵定义为

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

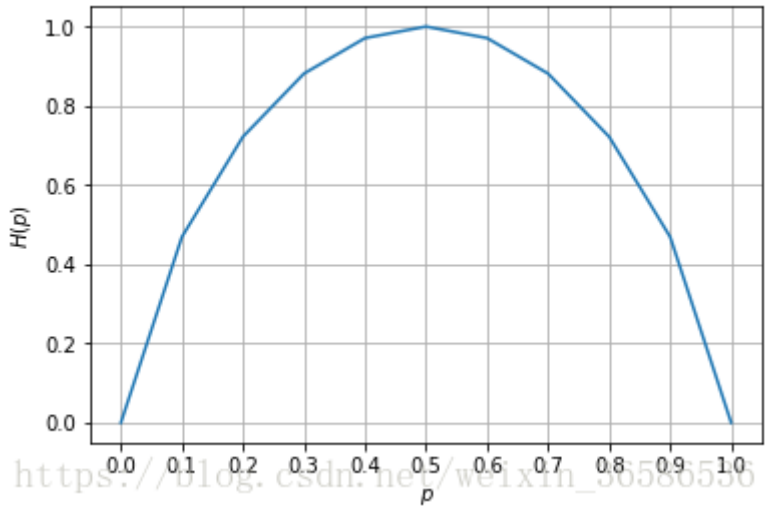
上述公式中的对数通常以2为底
熵越大，随机变量的不确定性越大。为了能够更好地理解熵的意义，我们下面举一个例子来说明。
当随机变量只取两个值，如1，0时，即XX的分布为

$$P(X=1)=p,\qquad P(X=0)=1-p,\qquad 0\le p\le 1$$

则熵为

$$H(p)=-plog_2 p-(1-p)log_2(1-p)$$

H(p)的函数图像如下：



条件熵：

设有随机变量(X,Y)。条件熵H(Y|X)表示在已知随机变量X的条件下随机变量YY的不确定性。随机变量X给定的条件下随机变量Y的条件熵H(Y|X)定义为X给定条件下Y的条件概率分布的熵对XX的数学期望

$$H(Y|X)=\sum_{i=1}^{n}p_iH(Y|X=x_i)$$

$$p_i=P(X=x_i),\quad i=1,2,\dots,n$$

信息增益

信息增益表示得知特征X的信息而使得类Y的信息的不确定性减少程度。接下来给出定义，特征A对训练数据集D的信息增益g(D,A),为集合D的熵H(D)与特征A给定条件下D的条件熵H(D|A)之差，即

$$g(D,A)=H(D)-H(D|A)$$

信息增益率：

特征A对训练数据集D的信息增益gR(D,A)定义为其信息增益g(D,A)与训练数据集D关于特征AA的值的熵HA(D)之比，即

$$g_R(D,A)=\frac{g(D,A)}{H_A(D)}$$

基尼指数：

基尼指数Gini(D)表示集合D不确定性，基尼指数Gini(D，A=a)表示集合DD经A=a分割后的不确定性(类似于熵)，基尼指数越小，样本的不确定性越小。

$$Gini(p)=\sum_{k=1}^{K}p_k(1-p_k)=1-\sum_{k=1}^{K}p_k^2$$

决策树生成

生成算法	划分标准
ID3	信息增益
C4.5	信息增益率
CART	基尼指数

ID3:

ID3算法的核心是在决策树各个节点上根据信息增益来选择进行划分的特征，然后递归地构建决策树。
具体方法：

1. 从根节点开始，对节点计算所有可能的特征的信息增益，选择信息增益值最大的特征作为节点的划分特征；
2. 由该特征的不同取值建立子节点；
3. 再对子节点递归地调用以上方法，构建决策树；
4. 到所有特征的信息增益都很小或者没有特征可以选择为止，得到最终的决策树

ID3的局限：

- 没有剪枝
- 采用信息增益作为选择最优划分特征的标准，然而信息增益会偏向那些取值较多的特征(这也是C4.5采用信息增益率的原因)

举例：

编号	色泽	根蒂	敲声	纹理	脐部	触感	好瓜
1	青绿	蜷缩	浊响	清晰	凹陷	硬滑	是
2	乌黑	蜷缩	沉闷	清晰	凹陷	硬滑	是
3	乌黑	蜷缩	浊响	清晰	凹陷	硬滑	是
4	青绿	蜷缩	沉闷	清晰	凹陷	硬滑	是
5	浅白	蜷缩	浊响	清晰	凹陷	硬滑	是
6	青绿	稍蜷	浊响	清晰	稍凹	软粘	是
7	乌黑	稍蜷	浊响	稍糊	稍凹	软粘	是
8	乌黑	稍蜷	浊响	清晰	稍凹	硬滑	是
9	乌黑	稍蜷	沉闷	稍糊	稍凹	硬滑	否
10	青绿	硬挺	清脆	清晰	平坦	软粘	否
11	浅白	硬挺	清脆	模糊	平坦	硬滑	否
12	浅白	蜷缩	浊响	模糊	平坦	软粘	否
13	青绿	稍蜷	浊响	稍糊	凹陷	硬滑	否
14	浅白	稍蜷	沉闷	稍糊	凹陷	硬滑	否
15	乌黑	稍蜷	浊响	清晰	稍凹	软粘	否
16	浅白	蜷缩	浊响	模糊	平坦	硬滑	否
17	青绿	蜷缩	沉闷	稍糊	稍凹	硬滑	否

1、选择最优划分属性

ID3采用信息增益作为选择最优的分裂属性的方法，选择熵作为衡量节点纯度的标准，根据以上公式对上述的例子的根节点进行分裂，分别计算每一个属性的信息增益，选择信息增益最大的属性进行分裂。

（1）、计算根节点的信息熵

本例中正例(好瓜)占 8/17，反例占 9/17 ，根结点的信息熵为：

$$Ent(D) = -\sum_{k=1}^{2} p_k log_2 p_k = -(\frac{8}{17} log_2 \frac{8}{17} + \frac{9}{17} log_2 \frac{9}{17})$$

计算各个特征的信息增益，选取最大的

计算当前属性集合{色泽，根蒂，敲声，纹理，脐部，触感}中每个属性的信息增益

色泽有3个可能的取值：{青绿，乌黑，浅白}

D1(色泽=青绿) = {1, 4, 6, 10, 13, 17}，正例 3/6，反例 3/6

D2(色泽=乌黑) = {2, 3, 7, 8, 9, 15}，正例 4/6，反例 2/6

D3(色泽=浅白) = {5, 11, 12, 14, 16}，正例 1/5，反例 4/5

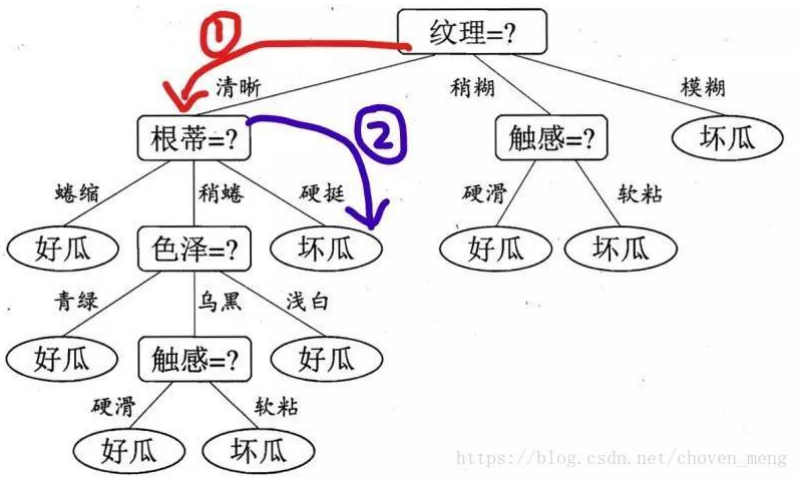
计算出用“色泽”划分之后所获得的3个分支结点的信息熵为：

$$Ent(D^2) = -(\frac{4}{6}log_2\frac{4}{6} + \frac{2}{6}log_2\frac{2}{6}) = 1.918;$$

由此可得“色泽”的信息增益为：

$$Gain(D, d) = Ent(D) - \sum_{v=1}^{3}\frac{|D^v|}{D}Ent(D^v)$$

类似的，我们可以计算出其他属性的信息增益，选取信息增益最大的进行划分，依次类推，最终得到决策树：



C4.5

C4.5与ID3相似，但对ID3进行了改进，在这里不再详细描述C4.5的实现，就讲一下有哪些基于ID3的改进：

- 用信息增益率来选择划分特征，克服了用信息增益选择的不足
- 在构造树的过程中进行剪枝
- 可对连续值与缺失值进行处理

CART

CART(classification and regressiontree),分类回归树算法，既可用于分类也可用于回归，在这一部分我们先主要将其分类树的生成。区别于ID3和C4.5,CART假设决策树是二叉树，内部节点特征的取值为“是”和“否”，左分支为取值为“是”的分支，右分支为取值为”否“的分支。这样的决策树等价于递归地二分每个特征，将输入空间(即特征空间)划分为有限个单元。CART的分类树用基尼指数来选择最优特征的最优划分点，具体过程如

1. 从根节点开始，对节点计算现有特征的基尼指数，对每一个特征，例如AA，再对其每个可能的取值如aa,根据样本点对A=aA=a的结果的”是“与”否“划分为两个部分，利用

   $$Gini(D, A = a) = \frac{|D_1|}{|D|}Gini(D_1) + \frac{|D_2|}{|D|}Gini(D_2)$$

   进行计算
2. 在所有可能的特征AA以及该特征所有的可能取值aa中，选择基尼指数最小的特征及其对应的取值作为最优特征和最优切分点。然后根据最优特征和最优切分点，将本节点的数据集二分，生成两个子节点
3. 对两个字节点递归地调用上述步骤，直至节点中的样本个数小于阈值，或者样本集的基尼指数小于阈值，或者没有更多特征后停止；
4. 生成CART分类树；

```
# 决策树.py
#
import numpy as np
from sklearn import tree, datasets, preprocessing
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import graphviz

np.random.RandomState(0)

# 加载数据
iris = datasets.load_iris()

# 划分训练集与测试集
x, y = iris.data, iris.target
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)

# 数据预处理
scaler = preprocessing.StandardScaler().fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)

# 创建模型
clf = tree.DecisionTreeClassifier()

# 模型拟合
clf.fit(x_train, y_train)

# 预测
y_pred = clf.predict(x_test)

# 评估
print(accuracy_score(y_test, y_pred))

dot_data = tree.export_graphviz(clf, out_file=None,
                                feature_names=iris.feature_names,
                                class_names=iris.target_names,
                                filled=True, rounded=True,
                                special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iris")
```

回归树

决策树是一种基本的分类与回归方法，本文叙述的是回归部分。回归决策树主要指CART(classification and regression tree)算法，内部结点特征的取值为“是”和“否”， 为二叉树结构。

所谓回归，就是根据特征向量来决定对应的输出值。回归树就是将特征空间划分成若干单元，每一个划分单元有一个特定的输出。因为每个结点都是“是”和“否”的判断，所以划分的边界是平行于坐标轴的。对于测试数据，我们只要按照特征将其归到某个单元，便得到对应的输出值。