In [47]:

```
from __future__ import print_function
```

In [48]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
```

In [126]:

```
cancer = load_breast_cancer()#导入数据
X_train,X_test,y_train,y_test = train_test_split(cancer.data,cancer.target,stratify=ca
header = cancer.feature_names
print(cancer.feature_names)
cancer.target_names
y_train = y_train.reshape((426,1))
y_test = y_test.reshape((143,1))
traindata = np.hstack((X_train,y_train)) #所给程序中标签和特征在同一数组中，在这里把他们弄
testdata = np.hstack((X_test,y_test)) #所给程序中标签和特征在同一数组中，在这里把他们弄到一
```

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

In [127]:

```python
class Question:
    """A Question is used to partition a dataset.

    This class just records a 'column number' (e.g., 0 for Color) and a
    'column value' (e.g., Green). The 'match' method is used to compare
    the feature value in an example to the feature value stored in the
    question. See the demo below.
    """

    def __init__(self, column, value):
        self.column = column
        self.value = value

    def match(self, example):
        # Compare the feature value in an example to the
        # feature value in this question.
        val = example[self.column]
        if is_numeric(val):
            return val >= self.value
        else:
            return val == self.value

    def __repr__(self):
        # This is just a helper method to print
        # the question in a readable format.
        condition = "=="
        if is_numeric(self.value):
            condition = ">="
        return "Is %s %s %s?" % (
            header[self.column], condition, str(self.value))
```

In [128]:

```python
def class_counts(rows):
    """Counts the number of each type of example in a dataset."""
    counts = {}  # a dictionary of label -> count.
    for row in rows:
        # in our dataset format, the label is always the last column
        label = row[-1]
        if label not in counts:
            counts[label] = 0
        counts[label] += 1
    return counts
```

In [129]:

```python
def is_numeric(value):
    """Test if a value is numeric."""
    return isinstance(value, int) or isinstance(value, float)
```

In [130]:

```python
Question(1, 3)
```

Out[130]:

```
Is mean texture >= 3?
```

In [131]:

```
def partition(rows, question):
    """Partitions a dataset.

    For each row in the dataset, check if it matches the question. If
    so, add it to 'true rows', otherwise, add it to 'false rows'.
    """
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return true_rows, false_rows
```

In [132]:

```
def gini(rows):
    """Calculate the Gini Impurity for a list of rows.

    There are a few different ways to do this, I thought this one was
    the most concise. See:
    https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity
    """
    counts = class_counts(rows)
    impurity = 1
    for lbl in counts:
        prob_of_lbl = counts[lbl] / float(len(rows))
        impurity -= prob_of_lbl**2
    return impurity
```

In [133]:

```
lebal = [['malignant'],
                  ['benign']]
```

In [134]:

```
gini(lebal)
```

Out[134]:

```
0.5
```

In [135]:

```
def info_gain(left, right, current_uncertainty):
    """Information Gain.

    The uncertainty of the starting node, minus the weighted impurity of
    two child nodes.
    """
    p = float(len(left)) / (len(left) + len(right))
    return current_uncertainty - p * gini(left) - (1 - p) * gini(right)
```

In [136]:

```
current_uncertainty = gini(traindata)
current_uncertainty
```

Out[136]:

```
0.46786351914302715
```

In [137]:

```
def find_best_split(rows):
    """Find the best question to ask by iterating over every feature / value
    and calculating the information gain."""
    best_gain = 0  # keep track of the best information gain
    best_question = None  # keep train of the feature / value that produced it
    current_uncertainty = gini(rows)
    n_features = len(rows[0]) - 1  # number of columns

    for col in range(n_features):  # for each feature

        values = set([row[col] for row in rows])  # unique values in the column

        for val in values:  # for each value

            question = Question(col, val)

            # try splitting the dataset
            true_rows, false_rows = partition(rows, question)

            # Skip this split if it doesn't divide the
            # dataset.
            if len(true_rows) == 0 or len(false_rows) == 0:
                continue

            # Calculate the information gain from this split
            gain = info_gain(true_rows, false_rows, current_uncertainty)

            # You actually can use '>' instead of '>=' here
            # but I wanted the tree to look a certain way for our
            # toy dataset.
            if gain >= best_gain:
                best_gain, best_question = gain, question

    return best_gain, best_question
```

In [138]:

```
best_gain, best_question = find_best_split(traindata)
best_question
```

Out[138]:

```
Is mean concave points >= 0.04938?
```

In [139]:

```
class Leaf:
    """A Leaf node classifies data.

    This holds a dictionary of class (e.g., "Apple") -> number of times
    it appears in the rows from the training data that reach this leaf.
    """

    def __init__(self, rows):
        self.predictions = class_counts(rows)
```

In [140]:

```
class Decision_Node:
    """A Decision Node asks a question.

    This holds a reference to the question, and to the two child nodes.
    """

    def __init__(self,
                 question,
                 true_branch,
                 false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch
```

In [141]:

```
def build_tree(rows):
    """Builds the tree.

    Rules of recursion: 1) Believe that it works. 2) Start by checking
    for the base case (no further information gain). 3) Prepare for
    giant stack traces.
    """

    # Try partitioing the dataset on each of the unique attribute,
    # calculate the information gain,
    # and return the question that produces the highest gain.
    gain, question = find_best_split(rows)

    # Base case: no further info gain
    # Since we can ask no further questions,
    # we'll return a leaf.
    if gain == 0:
        return Leaf(rows)

    # If we reach here, we have found a useful feature / value
    # to partition on.
    true_rows, false_rows = partition(rows, question)

    # Recursively build the true branch.
    true_branch = build_tree(true_rows)

    # Recursively build the false branch.
    false_branch = build_tree(false_rows)

    # Return a Question node.
    # This records the best feature / value to ask at this point,
    # as well as the branches to follow
    # dependingo on the answer.
    return Decision_Node(question, true_branch, false_branch)
```

In [142]:

```
def print_tree(node, spacing=""):
    """World's most elegant tree printing function."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        print (spacing + "Predict", node.predictions)
        return

    # Print the question at this node
    print (spacing + str(node.question))

    # Call this function recursively on the true branch
    print (spacing + '--> True:')
    print_tree(node.true_branch, spacing + "  ")

    # Call this function recursively on the false branch
    print (spacing + '--> False:')
    print_tree(node.false_branch, spacing + "  ")
```

In [143]:

```
def print_tree(node, spacing=""):
    """World's most elegant tree printing function."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        print (spacing + "Predict", node.predictions)
        return

    # Print the question at this node
    print (spacing + str(node.question))

    # Call this function recursively on the true branch
    print (spacing + '--> True:')
    print_tree(node.true_branch, spacing + "  ")

    # Call this function recursively on the false branch
    print (spacing + '--> False:')
    print_tree(node.false_branch, spacing + "  ")
```

In [144]:

```
my_tree = build_tree(traindata)
```

In [145]:

```
print_tree(my_tree)
```

```
Is mean concave points >= 0.04938?
--> True:
  Is worst concavity >= 0.2249?
  --> True:
    Is mean texture >= 14.26?
    --> True:
      Is worst perimeter >= 97.65?
      --> True:
        Predict {0.0: 140}
      --> False:
        Is worst texture >= 26.38?
        --> True:
          Predict {0.0: 3}
        --> False:
          Predict {1.0: 3}
    --> False:
      Predict {1.0: 3}
  --> False:
    Is worst fractal dimension >= 0.06599?
    --> True:
      Predict {1.0: 11}
    --> False:
      Predict {0.0: 2}
--> False:
  Is worst radius >= 16.89?
  --> True:
    Is worst texture >= 20.24?
    --> True:
      Is concave points error >= 0.01033?
      --> True:
        Predict {1.0: 2}
      --> False:
        Predict {0.0: 12}
    --> False:
      Predict {1.0: 4}
  --> False:
    Is area error >= 49.11?
    --> True:
      Is worst symmetry >= 0.2179?
      --> True:
        Predict {1.0: 1}
      --> False:
        Predict {0.0: 2}
    --> False:
      Predict {1.0: 243}
```

In [146]:

```
def classify(row, node):
    """See the 'rules of recursion' above."""

    # Base case: we've reached a leaf
    if isinstance(node, Leaf):
        return node.predictions

    # Decide whether to follow the true-branch or the false-branch.
    # Compare the feature / value stored in the node,
    # to the example we're considering.
    if node.question.match(row):
        return classify(row, node.true_branch)
    else:
        return classify(row, node.false_branch)
```

In [147]:

```
classify(traindata[1], my_tree)
```

Out[147]:

```
{1.0: 243}
```

In [148]:

```
def print_leaf(counts):
    """A nicer way to print the predictions at a leaf."""
    total = sum(counts.values()) * 1.0
    probs = {}
    for lbl in counts.keys():
        probs[lbl] = str(int(counts[lbl] / total * 100)) + "%"
    return probs
```

In [149]:

```
print_leaf(classify(traindata[0], my_tree))
```

Out[149]:

```
{0.0: '100%'}
```

In [150]:

```
print_leaf(classify(traindata[0], my_tree))
```

Out[150]:

```
{0.0: '100%'}
```

In [151]:

```
testdata
```

Out[151]:

```
array([[8.219e+00, 2.070e+01, 5.327e+01, ..., 3.322e-01, 1.486e-01,
        1.000e+00],
       [1.225e+01, 1.794e+01, 7.827e+01, ..., 3.113e-01, 8.132e-02,
        1.000e+00],
       [9.295e+00, 1.390e+01, 5.996e+01, ..., 3.681e-01, 8.982e-02,
        1.000e+00],
       ...,
       [1.831e+01, 2.058e+01, 1.208e+02, ..., 3.074e-01, 7.863e-02,
        0.000e+00],
       [1.705e+01, 1.908e+01, 1.134e+02, ..., 3.109e-01, 9.061e-02,
        0.000e+00],
       [1.170e+01, 1.911e+01, 7.433e+01, ..., 3.487e-01, 6.958e-02,
        1.000e+00]])
```

In [152]:

```
for row in testdata:
    print ("Actual: %s. Predicted: %s" %
           (row[-1], print_leaf(classify(row, my_tree))))#此处1和0分别代表是否有癌症
```

```
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 0.0. Predicted: {0.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 0.0. Predicted: {0.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 0.0. Predicted: {0.0: '100%'}
Actual: 0.0. Predicted: {0.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 0.0. Predicted: {0.0: '100%'}
Actual: 0.0. Predicted: {0.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {0.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
Actual: 1.0. Predicted: {1.0: '100%'}
```

用现有模型来对数据集进行训练和测试

In [153]:

```
DTC =DecisionTreeClassifier().fit(X_train,y_train)
accuracy = cross_val_score(DTC, X_test, y_test, scoring='accuracy', cv=5)
print("准确率:",accuracy.mean())
```

```
准确率: 0.9233990147783251
```