Python for Data Science

Joe McCarthy (http://interrelativity.com/joe), *Director, Analytics & Data Science*, Atigeo, LLC (http://atigeo.com)

In [1]:
```
from IPython.display import display, Image, HTML
```

Navigation

Notebooks in this primer:

In [2]:
```
# reconstitute relevent elements from the IPython environment active in previous notebook session
from collections import defaultdict
import simple_ml
clean_instances = simple_ml.load_instances('agaricus-lepiota.data', filter_missing_values=True)
attribute_names = simple_ml.load_attribute_names('agaricus-lepiota.attributes')
attribute_names_and_values = simple_ml.load_attribute_names_and_values('agaricus-lepiota.attributes')
```

4. Using Python to Build and Use a Simple Decision Tree Classifier

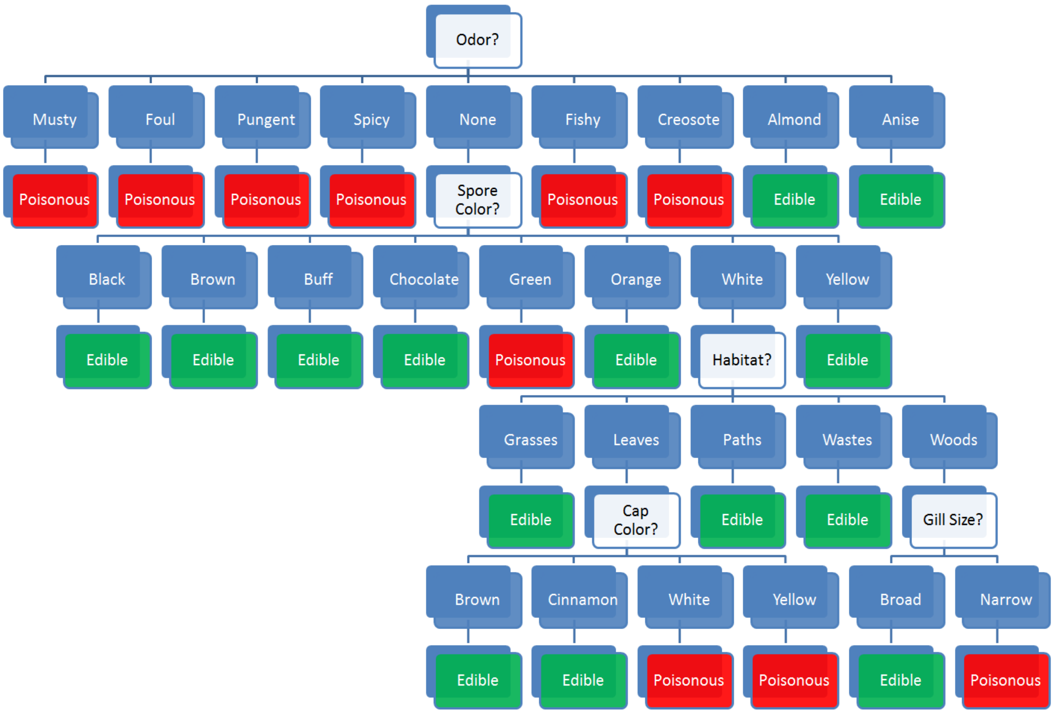
Decision Trees

Wikipedia offers the following description of a decision tree (https://en.wikipedia.org/wiki/Decision_tree) (with italics added to emphasize terms that will be elaborated below):

> A decision tree is a flowchart-like structure in which each *internal node* represents a *test* of an *attribute*, each branch represents an *outcome* of that test and each *leaf node* represents *class label* (a decision taken after testing all attributes in the path from the root to the leaf). Each path from the root to a leaf can also be represented as a classification rule.

The image below depicts a decision tree created from the UCI mushroom dataset that appears on Andy G's blog post about Decision Tree Learning (http://gieseanw.wordpress.com/2012/03/03/decision-tree-learning/), where

- a white box represents an *internal node* (and the label represents the *attribute* being tested)
- a blue box represents an attribute value (an *outcome* of the *test* of that attribute)
- a green box represents a *leaf node* with a *class label* of *edible*
- a red box represents a *leaf node* with a *class label* of *poisonous*



It is important to note that the UCI mushroom dataset consists entirely of categorical variables (https://en.wikipedia.org/wiki/Categorical_variable), i.e., every variable (or *attribute*) has an enumerated set of possible values. Many datasets include numeric variables that can take on int or float values. Tests for such varia typically use comparison operators, e.g., $age < 65$ or $36,250 < adjusted\_gross\_income <= 87,850$. *[Aside: Python supports boolean expressions contain multiple comparison operators, such as the expression comparing adjusted_gross_income in the preceding example.]*

Our simple decision tree will only accommodate categorical variables. We will closely follow a version of the decision tree learning algorithm implementation (http://www.onlamp.com/pub/a/python/2006/02/09/ai_decision_trees.html?page=3) offered by Chris Roach.

Our goal in the following sections is to use Python to

- *create* a simple decision tree based on a set of training instances
- *classify* (predict class labels for) for an instance using a simple decision tree
- *evaluate* the performance of the simple decision tree on classifying a set of test instances

First, we will explore some concepts and algorithms used in building and using decision trees.

Entropy

When building a supervised classification model, the frequency distribution of attribute values is a potentially important factor in determining the relative importa of each attribute at various stages in the model building process.

In data modeling, we can use frequency distributions to compute ***entropy***, a measure of disorder (impurity) in a set.

We compute the entropy of multiplying the proportion of instances with each class label by the log of that proportion, and then taking the negative sum of those terms.

More precisely, for a 2-class (binary) classification task:

$$entropy(S) = -p_1 log_2(p_1) - p_2 log_2(p_2)$$

where p_i is proportion (relative frequency) of class *i* within the set *S*.

From the output above, we know that the proportion of `clean_instances` that are labeled `'e'` (class `edible`) in the UCI dataset is $3488 \div 5644 = 0.618$, a the proportion labeled `'p'` (class `poisonous`) is $2156 \div 5644 = 0.382$.

After importing the Python `math` (http://docs.python.org/2/library/math.html) module, we can use the `math.log(x[, base])` (http://docs.python.org/2/library/math.html#math.log) function in computing the entropy of the `clean_instances` of the UCI mushroom data set as follows:

In [3]:
```
import math
entropy = - (3488 / 5644.0) * math.log(3488 / 5644.0, 2) - (2156 / 5644.0) * math.log(2156 / 5644.0, 2)
print entropy
```

```
0.959441337353
```

Exercise 6: define entropy()

Define a function, `entropy(instances)`, that computes the entropy of `instances`. You may assume the class label is in position 0; we will later see how to specify default parameter values in function definitions.

[Note: the class label in many data files is the *last* rather than the *first* item on each line.]

In [4]:
```
# your function definition here

# delete 'simple_ml.' below to test your function
print simple_ml.entropy(clean_instances)
```

```
0.959441337353
```

Information Gain

Informally, a decision tree is constructed using a recursive algorithm that

- selects the *best* attribute
- splits the set into subsets based on the values of that attribute (each subset is composed of instances from the original set that have the same value for that attribute)
- repeats the process on each of these subsets until a stopping condition is met (e.g., a subset has no instances or has instances which all have the same class label)

Entropy is a metric that can be used in selecting the best attribute for each split: the best attribute is the one resulting in the *largest decrease in entropy* for a set instances. [Note: other metrics can be used for determining the best attribute]

Information gain measures the decrease in entropy that results from splitting a set of instances based on an attribute.

$$IG(S, a) = entropy(S) - [p(s_1) \times entropy(s_1) + p(s_2) \times entropy(s_2) \ldots + p(s_n) \times entropy(s_n)]$$

Where n is the number of distinct values of attribute a, and s_i is the subset of S where all instances have the ith value of a.

In [5]:
```
print 'Information gain for different attributes:\n'
for i in range(1, len(attribute_names)):
    print '{:5.3f}  {:2} {}'.format(simple_ml.information_gain(clean_instances, i), i, attribute_names[i])
```

```
Information gain for different attributes:

0.017   1 cap-shape
0.005   2 cap-surface
0.195   3 cap-color
0.140   4 bruises?
0.860   5 odor
0.004   6 gill-attachment
0.058   7 gill-spacing
0.032   8 gill-size
0.213   9 gill-color
0.275  10 stalk-shape
0.097  11 stalk-root
0.425  12 stalk-surface-above-ring
0.409  13 stalk-surface-below-ring
0.306  14 stalk-color-above-ring
0.279  15 stalk-color-below-ring
0.000  16 veil-type
0.002  17 veil-color
0.012  18 ring-number
0.463  19 ring-type
0.583  20 spore-print-color
0.110  21 population
0.101  22 habitat
```

We can sort the attributes based in decreasing order of information gain.

In [6]:
```
print 'Information gain for different attributes:\n'
sorted_information_gain_indexes = sorted([(simple_ml.information_gain(clean_instances, i), i) for i in range(1, len(att
ute_names))],
                                         reverse=True)
print sorted_information_gain_indexes, '\n'

for gain, i in sorted_information_gain_indexes:
    print '{:5.3f}  {:2} {}'.format(gain, i, attribute_names[i])
```

```
Information gain for different attributes:

[(0.8596704358849709, 5), (0.5828694793608379, 20), (0.46290566555455265, 19), (0.42456477093655975, 12), (0.40865
780788318695, 13), (0.3062989793570199, 14), (0.27891994708759504, 15), (0.2750355212178639, 10), (0.2127971869976
022, 9), (0.19495343617580085, 3), (0.1400386042032834, 4), (0.1097880400299237, 21), (0.10067585994181227, 22), (
0.09733858997769329, 11), (0.05836192763098613, 7), (0.03242975884332899, 8), (0.01740692300090696, 1), (0.0120596
7443646827, 18), (0.004572013423856602, 2), (0.0044397141315495325, 6), (0.0019702590992403124, 17), (0.0, 16)]

0.860   5 odor
0.583  20 spore-print-color
0.463  19 ring-type
0.425  12 stalk-surface-above-ring
0.409  13 stalk-surface-below-ring
0.306  14 stalk-color-above-ring
0.279  15 stalk-color-below-ring
0.275  10 stalk-shape
0.213   9 gill-color
0.195   3 cap-color
0.140   4 bruises?
0.110  21 population
0.101  22 habitat
0.097  11 stalk-root
0.058   7 gill-spacing
0.032   8 gill-size
0.017   1 cap-shape
0.012  18 ring-number
0.005   2 cap-surface
0.004   6 gill-attachment
0.002  17 veil-color
0.000  16 veil-type
```

The following variation does not use a list comprehension:

In [7]:
```
print 'Information gain for different attributes:\n'

information_gain_values = []
for i in range(1, len(attribute_names)):
    information_gain_values.append((simple_ml.information_gain(clean_instances, i), i))

sorted_information_gain_indexes = sorted(information_gain_values,
                                         reverse=True)
print sorted_information_gain_indexes, '\n'

for gain, i in sorted_information_gain_indexes:
    print '{:5.3f}  {:2} {}'.format(gain, i, attribute_names[i])
```

```
Information gain for different attributes:

[(0.8596704358849709, 5), (0.5828694793608379, 20), (0.46290566555455265, 19), (0.42456477093655975, 12), (0.40865
780788318695, 13), (0.3062989793570199, 14), (0.27891994708759504, 15), (0.2750355212178639, 10), (0.2127971869976
022, 9), (0.19495343617580085, 3), (0.1400386042032834, 4), (0.1097880400299237, 21), (0.10067585994181227, 22), (
0.09733858997769329, 11), (0.05836192763098613, 7), (0.03242975884332899, 8), (0.01740692300090696, 1), (0.0120596
7443646827, 18), (0.004572013423856602, 2), (0.0044397141315495325, 6), (0.0019702590992403124, 17), (0.0, 16)]

0.860   5 odor
0.583  20 spore-print-color
0.463  19 ring-type
0.425  12 stalk-surface-above-ring
0.409  13 stalk-surface-below-ring
0.306  14 stalk-color-above-ring
0.279  15 stalk-color-below-ring
0.275  10 stalk-shape
0.213   9 gill-color
0.195   3 cap-color
0.140   4 bruises?
0.110  21 population
0.101  22 habitat
0.097  11 stalk-root
0.058   7 gill-spacing
0.032   8 gill-size
0.017   1 cap-shape
0.012  18 ring-number
0.005   2 cap-surface
0.004   6 gill-attachment
0.002  17 veil-color
0.000  16 veil-type
```

Exercise 7: define information_gain()

Define a function, `information_gain(instances, i)`, that returns the information gain achieved by selecting the `i`th attribute to split `instances`. It shoul
exhibit the same behavior as the `simple_ml` version of the function.

In [8]:
```
# your definition of information_gain(instances, i) here

# delete 'simple_ml.' below to test your function
sorted_information_gain_indexes = sorted([(simple_ml.information_gain(clean_instances, i), i) for i in range(1, len(att
ute_names))],
                                         reverse=True)

print 'Information gain for different attributes:\n'
for gain, i in sorted_information_gain_indexes:
    print '{:5.3f}  {:2} {}'.format(gain, i, attribute_names[i])
```

```
Information gain for different attributes:

0.860   5 odor
0.583  20 spore-print-color
0.463  19 ring-type
0.425  12 stalk-surface-above-ring
0.409  13 stalk-surface-below-ring
0.306  14 stalk-color-above-ring
0.279  15 stalk-color-below-ring
0.275  10 stalk-shape
0.213   9 gill-color
0.195   3 cap-color
0.140   4 bruises?
0.110  21 population
0.101  22 habitat
0.097  11 stalk-root
0.058   7 gill-spacing
0.032   8 gill-size
0.017   1 cap-shape
0.012  18 ring-number
0.005   2 cap-surface
0.004   6 gill-attachment
0.002  17 veil-color
0.000  16 veil-type
```

Building a Simple Decision Tree

We will implement a modified version of the ID3 (https://en.wikipedia.org/wiki/ID3_algorithm) algorithm for building a simple decision tree.

```
ID3 (Examples, Target_Attribute, Attributes)
    Create a root node for the tree
    If all examples are positive, Return the single-node tree Root, with label = +.
    If all examples are negative, Return the single-node tree Root, with label = -.
    If number of predicting attributes is empty, then Return the single node tree Root,
    with label = most common value of the target attribute in the examples.
    Otherwise Begin
        A ← The Attribute that best classifies examples.
        Decision Tree attribute for Root = A.
        For each possible value, v_i, of A,
            Add a new tree branch below Root, corresponding to the test A = v_i.
            Let Examples(v_i) be the subset of examples that have the value v_i for A
            If Examples(v_i) is empty
                Then below this new branch add a leaf node with label = most common target value in the examples
            Else below this new branch add the subtree ID3 (Examples(v_i), Target_Attribute, Attributes – {A})
    End
    Return Root
```

In building a decision tree, we will need to split the instances based on the index of the *best* attribute, i.e., the attribute that offers the *highest information gain*. W will use separate utility functions to handle these subtasks. To simplify the functions, we will rely exclusively on attribute indexes rather than attribute names.

Note: the algorithm above is *recursive*, i.e., the there is a recursive call to `ID3` within the definition of `ID3`. Covering recursion is beyond the scope of this primer, there are a number of other resources on using recursion in Python (https://www.google.com/search?q=python+recursion). Familiarity with recursion will be important for understanding both the tree construction and classification functions below.

First, we will define a function to split a set of instances based on any attribute. This function will return a dictionary where the *key* of each dictionary is a distinct value of the specified `attribute_index`, and the *value* of each dictionary is a list representing the subset of `instances` that have that attribute value.

In [9]:
```
def split_instances(instances, attribute_index):
    '''Returns a list of dictionaries, splitting a list of instances according to their values of a specified attribute

    The key of each dictionary is a distinct value of attribute_index,
    and the value of each dictionary is a list representing the subset of instances that have that value for the attrib
'''
    partitions = defaultdict(list)
    for instance in instances:
        partitions[instance[attribute_index]].append(instance)
    return partitions

partitions = split_instances(clean_instances, 5)
print [(partition, len(partitions[partition])) for partition in partitions]
```

```
[('a', 400), ('c', 192), ('f', 1584), ('m', 36), ('l', 400), ('n', 2776), ('p', 256)]
```

Now that we can split instances based on a particular attribute, we would like to be able to choose the *best* attribute with which to split the instances, where *bes* defined as the attribute that provides the greatest information gain if instances were split based on that attribute. We will want to restrict the candidate attributes that we don't bother trying to split on an attribute that was used higher up in the decision tree (or use the target attribute as a candidate).

Exercise 8: define choose_best_attribute_index()

Define a function, `choose_best_attribute_index(instances, candidate_attribute_indexes)`, that returns the index in the list of `candidate_attribute_indexes` that provides the highest information gain if `instances` are split based on that attribute index.

In [10]:
```
# your function here

# delete 'simple_ml.' below to test your function:
print 'Best attribute index:', simple_ml.choose_best_attribute_index(clean_instances, range(1, len(attribute_names)))
```

```
Best attribute index: 5
```

A leaf node in a decision tree represents the most frequently occurring - or majority - class value for that path through the tree. We will need a function that determines the majority value for the class index among a set of instances.

We earlier saw how the [`defaultdict` (http://docs.python.org/2/library/collections.html#collections.defaultdict)](http://docs.python.org/2/library/collections.html#collections.defaultdict) container in the [`collections` (http://docs.python.org/2/library/collections.html)](http://docs.python.org/2/library/collections.html) module can be used to simplify the construction of a dictionary containing the counts of all attribute values for attributes, by automatically setting the count for any attribute value to zero when the attribute value is first added to the dictionary.

The `collections` module has another useful container, a [`Counter` (http://docs.python.org/2/library/collections.html#collections.Counter)](http://docs.python.org/2/library/collections.html#collections.Counter) class, that can furthe simplify the construction of a specialized dictionary of counts. When a `Counter` object is instantiated with a list of items, it returns a dictionary-like container in which the *keys* are the unique items in the list, and the *values* are the counts of each unique item in that list.

This container has an additional method, [`most_common([n])` (http://docs.python.org/2/library/collections.html#collections.Counter.most_common)](http://docs.python.org/2/library/collections.html#collections.Counter.most_common), which retu a list of 2-element tuples representing the values and their associated counts for the most common `n` values; if `n` is omitted, the method returns all tuples.

The following is an example of how we can use a `Counter` to represent the frequency of different class labels, and how we can identify the most frequent value its count.

In [11]:
```
from collections import Counter

class_counts = Counter([instance[0] for instance in clean_instances])
print 'class_counts: {}; most_common(1): {}, most_common(1)[0][0]: {}'.format(
    class_counts, # the Counter object
    class_counts.most_common(1), # returns a list in which the 1st element is a tuple with the most common value and it
ount
    class_counts.most_common(1)[0][0]) # the most common value (1st element in that tuple)
```

```
class_counts: Counter({'e': 3488, 'p': 2156}); most_common(1): [('e', 3488)], most_common(1)[0][0]: e
```

The following variation does not use a list comprehension:

In [12]:
```
class_values = []
for instance in clean_instances:
    class_values.append(instance[0])

class_counts = Counter(class_values)
print 'class_counts: {}; most_common(1): {}, most_common(1)[0][0]: {}'.format(
    class_counts, # the Counter object
    class_counts.most_common(1), # returns a list in which the 1st element is a tuple with the most common value and it
ount
    class_counts.most_common(1)[0][0]) # the most common value (1st element in that tuple)
```

```
class_counts: Counter({'e': 3488, 'p': 2156}); most_common(1): [('e', 3488)], most_common(1)[0][0]: e
```

Before putting all this together to define a decision tree construction function, it may be helpful to cover a few additional aspects of Python the function will utiliz

Python offers a very flexible mechanism for the testing of truth values (http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#testing-for-truth values): in an **if** condition, any null object, zero-valued numerical expression or empty container (string, list, dictionary or tuple) is interpreted as *False* (i.e., *not Tru*

In [13]:
```
for x in [False, None, 0, 0.0, "", [], {}, ()]:
    print '"{}" is'.format(x),
    if x:
        print True
    else:
        print False
```

```
"False" is False
"None" is False
"0" is False
"0.0" is False
"" is False
"[]" is False
"{}" is False
"()" is False
```

Python also offers a conditional expression (ternary operator) (http://docs.python.org/2/reference/expressions.html#conditional-expressions) that allows the functionality of an if/else statement that returns a value to be implemented as an expression. For example, the if/else statement in the code above could be implemented as a conditional expression as follows:

In [14]:
```
for x in [False, None, 0, 0.0, "", [], {}, ()]:
    print '"{}" is {}'.format(x, True if x else False) # using conditional expression as second argument to format()
```

```
"False" is False
"None" is False
"0" is False
"0.0" is False
"" is False
"[]" is False
"{}" is False
"()" is False
```

Python function definitions can specify default parameter values (http://docs.python.org/2/tutorial/controlflow.html#default-argument-values) indicating the value those parameters will have if no argument is explicitly provided when the function is called. Arguments can also be passed using keyword parameters (http://docs.python.org/2/tutorial/controlflow.html#keyword-arguments) indicting which parameter will be assigned a specific argument value (which may or may correspond to the order in which the parameters are defined).

The Python Tutorial page on default parameters (http://docs.python.org/2/tutorial/controlflow.html#default-argument-values) includes the following warning:

> Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

Thus it is generally better to use the Python null object, `None`, rather than an empty `list` (`[]`), `dict` (`{}`) or other mutable data structure when specifying defaul parameter values for any of those data types.

In [15]:
```
def parameter_test(parameter1=None, parameter2=None):
    '''Prints the values of parameter1 and parameter2'''
    print 'parameter1: {}; parameter2: {}'.format(parameter1, parameter2)

parameter_test() # no args are required
parameter_test(1) # if any args are provided, 1st arg gets assigned to parameter1
parameter_test(1, 2) # 2nd arg gets assigned to parameter2
parameter_test(2) # remember: if only 1 arg, 1st arg gets assigned to arg1
parameter_test(parameter2=2) # can use keyword to [only] provide an explicit value for parameter2
parameter_test(parameter2=2, parameter1=1) # can use keywords for either arg, in either order
```

```
parameter1: None; parameter2: None
parameter1: 1; parameter2: None
parameter1: 1; parameter2: 2
parameter1: 2; parameter2: None
parameter1: None; parameter2: 2
parameter1: 1; parameter2: 2
```

Exercise 9: define majority_value()

Define a function, `majority_value(instances, class_index)`, that returns the most frequently occurring value of `class_index` in `instances`. The `class_index` parameter should be optional, and have a default value of `0` (zero).

In [16]:
```
# your definition of majority_value(instances) here

# delete 'simple_ml.' below to test your function:
print 'Majority value of index {}: {}'.format(0, simple_ml.majority_value(clean_instances)) # note: relying on default
ameter here
# although there is only one class_index for the dataset, we'll test it by providing non-default values
print 'Majority value of index {}: {}'.format(1, simple_ml.majority_value(clean_instances, 1)) # using an optional 2nd
ument
print 'Majority value of index {}: {}'.format(2, simple_ml.majority_value(clean_instances, class_index=2)) # using a ke
rd
```

```
Majority value of index 0: e
Majority value of index 1: x
Majority value of index 2: y
```

The recursive `create_decision_tree()` function below uses an optional parameter, `class_index`, which defaults to `0`. This is to accommodate other datas in which the class label is the last element on each line (which would be most easily specified by using a `-1` value). Most data files in the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets.html) have the class labels as either the first element or the last element.

To show how the decision tree is being built, an optional `trace` parameter, when non-zero, will generate some trace information as the tree is constructed. The indentation level is incremented with each recursive call via the use of the conditional expression (ternary operator), `trace + 1 if trace else 0`.

In [17]:
```
def create_decision_tree(instances, candidate_attribute_indexes=None, class_index=0, default_class=None, trace=0):
    '''Returns a new decision tree trained on a list of instances.

    The tree is constructed by recursively selecting and splitting instances based on
    the highest information_gain of the candidate_attribute_indexes.

    The class label is found in position class_index.

    The default_class is the majority value for the current node's parent in the tree.
    A positive (int) trace value will generate trace information with increasing levels of indentation.

    Derived from the simplified ID3 algorithm presented in Building Decision Trees in Python by Christopher Roach,
    http://www.onlamp.com/pub/a/python/2006/02/09/ai_decision_trees.html?page=3'''

    # if no candidate_attribute_indexes are provided, assume that we will use all but the target_attribute_index
    if candidate_attribute_indexes is None:
        candidate_attribute_indexes = range(len(instances[0]))
        candidate_attribute_indexes.remove(class_index)

    class_labels_and_counts = Counter([instance[class_index] for instance in instances])

    # If the dataset is empty or the candidate attributes list is empty, return the default value
    if not instances or not candidate_attribute_indexes:
        if trace:
            print '{}Using default class {}'.format('< ' * trace, default_class)
        return default_class

    # If all the instances have the same class label, return that class label
    elif len(class_labels_and_counts) == 1:
        class_label = class_labels_and_counts.most_common(1)[0][0]
        if trace:
            print '{}All {} instances have label {}'.format('< ' * trace, len(instances), class_label)
        return class_label
    else:
        default_class = simple_ml.majority_value(instances, class_index)

        # Choose the next best attribute index to best classify the instances
        best_index = simple_ml.choose_best_attribute_index(instances, candidate_attribute_indexes, class_index)
        if trace:
            print '{}Creating tree node for attribute index {}'.format('> ' * trace, best_index)

        # Create a new decision tree node with the best attribute index and an empty dictionary object (for now)
        tree = {best_index:{}}

        # Create a new decision tree sub-node (branch) for each of the values in the best attribute field
        partitions = simple_ml.split_instances(instances, best_index)

        # Remove that attribute from the set of candidates for further splits
        remaining_candidate_attribute_indexes = [i for i in candidate_attribute_indexes if i != best_index]
        for attribute_value in partitions:
            if trace:
                print '{}Creating subtree for value {} ({}, {}, {}, {})'.format(
                    '> ' * trace,
                    attribute_value,
                    len(partitions[attribute_value]),
                    len(remaining_candidate_attribute_indexes),
                    class_index,
                    default_class)

            # Create a subtree for each value of the the best attribute
            subtree = create_decision_tree(
```

```
                partitions[attribute_value],
                remaining_candidate_attribute_indexes,
                class_index,
                default_class,
                trace + 1 if trace else 0)

            # Add the new subtree to the empty dictionary object in the new tree/node we just created
            tree[best_index][attribute_value] = subtree

    return tree

# split instances into separate training and testing sets
training_instances = clean_instances[:-20]
testing_instances = clean_instances[-20:]
tree = create_decision_tree(training_instances, trace=1) # remove trace=1 to turn off tracing
print tree
```

```
> Creating tree node for attribute index 5
> Creating subtree for value a (400, 21, 0, e)
< < All 400 instances have label e
> Creating subtree for value c (192, 21, 0, e)
< < All 192 instances have label p
> Creating subtree for value f (1584, 21, 0, e)
< < All 1584 instances have label p
> Creating subtree for value m (28, 21, 0, e)
< < All 28 instances have label p
> Creating subtree for value l (400, 21, 0, e)
< < All 400 instances have label e
> Creating subtree for value n (2764, 21, 0, e)
> > Creating tree node for attribute index 20
> > Creating subtree for value k (1296, 20, 0, e)
< < < All 1296 instances have label e
> > Creating subtree for value r (72, 20, 0, e)
< < < All 72 instances have label p
> > Creating subtree for value w (100, 20, 0, e)
> > > Creating tree node for attribute index 21
> > > Creating subtree for value y (24, 19, 0, e)
< < < < All 24 instances have label e
> > > Creating subtree for value c (16, 19, 0, e)
< < < < All 16 instances have label p
> > > Creating subtree for value v (60, 19, 0, e)
< < < < All 60 instances have label e
> > Creating subtree for value n (1296, 20, 0, e)
< < < All 1296 instances have label e
> Creating subtree for value p (256, 21, 0, e)
< < All 256 instances have label p
{5: {'a': 'e', 'c': 'p', 'f': 'p', 'm': 'p', 'l': 'e', 'n': {20: {'k': 'e', 'r': 'p', 'w': {21: {'y': 'e', 'c': 'p
', 'v': 'e'}}, 'n': 'e'}}, 'p': 'p'}}
```

The structure of the tree shown above is rather difficult to discern from the normal printed representation of a dictionary.

The Python `pprint` (http://docs.python.org/2/library/pprint.html) module has a number of useful methods for pretty-printing or formatting objects in a more hum readable way.

The `pprint.pprint(object, stream=None, indent=1, width=80, depth=None)` (http://docs.python.org/2/library/pprint.html#pprint.pprint) method print `object` to a `stream` (a default value of `None` will dictate the use of sys.stdout (http://docs.python.org/2/library/sys.html#sys.stdout), the same destination `print` statement output), using `indent` spaces to differentiate nesting levels, using up to a maximum `width` columns and up to to a maximum nesting level `de` (`None` indicating no maximum).

We will use the a variation on the import statement that imports one or more functions into the current namespace:

```
from pprint import pprint
```

This will to enable us to use `pprint()` rather than having to use dotted notation, i.e., `pprint.pprint()`.

Note that if we wanted to define our own `pprint()` function, we would be best only using

```
import pprint
```

so that we can still access the `pprint()` function in the `pprint` module (since defining `pprint()` in the current namespace would otherwise override the imported definition of the function).

In [18]:
```
from pprint import pprint
pprint(tree)
```

```
{5: {'a': 'e',
     'c': 'p',
     'f': 'p',
     'l': 'e',
     'm': 'p',
     'n': {20: {'k': 'e',
                'n': 'e',
                'r': 'p',
                'w': {21: {'c': 'p', 'v': 'e', 'y': 'e'}}}},
     'p': 'p'}}
```

Classifying Instances with a Simple Decision Tree

Usually, when we construct a decision tree based on a set of *training* instances, we do so with the intent of using that tree to classify a set of one or more *testing* instances.

We will define a function, `classify(tree, instance, default_class=None)`, to use a decision `tree` to classify a single `instance`, where an optional `default_class` can be specified as the return value if the instance represents a set of attribute values that don't have a representation in the decision tree.

We will use a design pattern in which we will use a series of `if` statements, each of which returns a value if the condition is true, rather than a nested series of `i` `elif` and/or `else` clauses, as it helps constrain the levels of indentation in the function.

In [19]:
```
def classify(tree, instance, default_class=None):
    '''Returns a classification label for instance, given a decision tree'''
    if not tree:
        return default_class
    if not isinstance(tree, dict):
        return tree
    attribute_index = tree.keys()[0]
    attribute_values = tree.values()[0]
    instance_attribute_value = instance[attribute_index]
    if instance_attribute_value not in attribute_values:
        return default_class
    return classify(attribute_values[instance_attribute_value], instance, default_class)

for instance in testing_instances:
    predicted_label = classify(tree, instance)
    actual_label = instance[0]
    print 'predicted: {}; actual: {}'.format(predicted_label, actual_label)
```

```
predicted: p; actual: p
predicted: p; actual: p
predicted: p; actual: p
predicted: e; actual: e
predicted: e; actual: e
predicted: p; actual: p
predicted: e; actual: e
predicted: e; actual: e
predicted: e; actual: e
predicted: p; actual: p
predicted: e; actual: e
predicted: e; actual: e
predicted: e; actual: e
predicted: p; actual: p
predicted: e; actual: e
predicted: e; actual: e
predicted: e; actual: e
predicted: e; actual: e
predicted: p; actual: p
predicted: p; actual: p
```

Evaluating the Accuracy of a Simple Decision Tree

It is often helpful to evaluate the performance of a model using a dataset not used in the training of that model. In the simple example shown above, we used all the last 20 instances to train a simple decision tree, then classified those last 20 instances using the tree.

The advantage of this training/testing split is that visual inspection of the classifications (sometimes called *predictions*) is relatively straightforward, revealing that 20 instances were correctly classified.

There are a variety of metrics that can be used to evaluate the performance of a model. [Scikit Learn's Model Evaluation (http://scikit-learn.org/stable/modules/model_evaluation.html)] library provides an overview and implementation of several possible metrics. For now, we'll simply measure the *accuracy* of a model, i.e., the percentage of testing instances that are correctly classified (*true positives* and *true negatives*).

The accuracy of the model above, given the set of 20 testing instances, is 100% (20/20).

The function below calculates the classification accuracy of a `tree` over a set of `testing_instances` (with an optional `class_index` parameter indicating th position of the class label in each instance).

In [20]:
```
def classification_accuracy(tree, testing_instances, class_index=0, default_class=None):
    '''Returns the accuracy of classifying testing_instances with tree, where the class label is in position class_inde
'
    num_correct = 0
    for i in xrange(len(testing_instances)):
        prediction = classify(tree, testing_instances[i], default_class)
        actual_value = testing_instances[i][class_index]
        if prediction == actual_value:
            num_correct += 1
    return float(num_correct) / len(testing_instances)

print classification_accuracy(tree, testing_instances)
```

```
1.0
```

The `zip([iterable, ...])` (http://docs.python.org/2.7/library/functions.html#zip) function combines 2 or more sequences or iterables; the function returns a of tuples, where the *i*th tuple contains the *i*th element from each of the argument sequences or iterables.

In [21]:
```
zip([0, 1, 2], ['a', 'b', 'c'])
```

Out[21]:
```
[(0, 'a'), (1, 'b'), (2, 'c')]
```

We can use list comprehensions (http://docs.python.org/2/tutorial/datastructures.html#list-comprehensions), the `Counter` class and the `zip()` function to mod `classification_accuracy()` so that it returns a packed tuple with

- the number of correctly classified instances
- the number of incorrectly classified instances
- the percentage of instances correctly classified

In [22]:
```
def classification_accuracy(tree, instances, class_index=0, default_class=None):
    '''Returns the accuracy of classifying testing_instances with tree, where the class label is in position class_inde
'
    predicted_labels = [classify(tree, instance, default_class) for instance in instances]
    actual_labels = [x[class_index] for x in instances]
    counts = Counter([x == y for x, y in zip(predicted_labels, actual_labels)])
    return counts[True], counts[False], float(counts[True]) / len(instances)

print classification_accuracy(tree, testing_instances)
```

```
(20, 0, 1.0)
```

We sometimes want to partition the instances into subsets of equal sizes to measure performance. One metric this partitioning allows us to compute is a learnin curve (https://en.wikipedia.org/wiki/Learning_curve), i.e., assess how well the model performs based on the size of its training set. Another use of these partition (aka *folds*) would be to conduct an *n-fold cross validation* (https://en.wikipedia.org/wiki/Cross-validation_(statistics)) evaluation.

The following function, `partition_instances(instances, num_partitions)`, partitions a set of `instances` into `num_partitions` relatively equally si: subsets.

We'll use this as yet another opportunity to demonstrate the power of using list comprehensions, this time, to condense the use of nested `for` loops.

In [23]:
```
def partition_instances(instances, num_partitions):
    '''Returns a list of relatively equally sized disjoint sublists (partitions) of the list of instances'''
    return [[instances[j] for j in xrange(i, len(instances), num_partitions)] for i in xrange(num_partitions)]
```

Before testing this function on the 5644 `clean_instances` from the UCI mushroom dataset, let's create a small number of simplified instances to verify that th function has the desired behavior.

In [24]:
```
instance_length = 3
num_instances = 5

simplified_instances = [[j for j in xrange(i, instance_length + i)] for i in xrange(num_instances)]

print 'Instances:', simplified_instances
partitions = partition_instances(simplified_instances, 2)
print 'Partitions:', partitions
```

```
Instances: [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]
Partitions: [[[0, 1, 2], [2, 3, 4], [4, 5, 6]], [[1, 2, 3], [3, 4, 5]]]
```

The following variations do not use list comprehensions.

In [25]:
```
def partition_instances(instances, num_partitions):
    '''Returns a list of relatively equally sized disjoint sublists (partitions) of the list of instances'''
    partitions = []
    for i in xrange(num_partitions):
        partition = []
        # iterate over instances starting at position i in increments of num_paritions
        for j in xrange(i, len(instances), num_partitions):
            partition.append(instances[j])
        partitions.append(partition)
    return partitions

simplified_instances = []
for i in xrange(num_instances):
    new_instance = []
    for j in xrange(i, instance_length + i):
        new_instance.append(j)
    simplified_instances.append(new_instance)

print 'Instances:', simplified_instances
partitions = partition_instances(simplified_instances, 2)
print 'Partitions:', partitions
```

```
Instances: [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]
Partitions: [[[0, 1, 2], [2, 3, 4], [4, 5, 6]], [[1, 2, 3], [3, 4, 5]]]
```

The `enumerate(sequence, start=0)` (http://docs.python.org/2.7/library/functions.html#enumerate) function creates an iterator that successively returns the index and value of each element in a `sequence`, beginning at the `start` index.

In [26]:
```
for i, x in enumerate(['a', 'b', 'c']):
    print i, x
```

```
0 a
1 b
2 c
```

We can use `enumerate()` to facilitate slightly more rigorous testing of our `partition_instances` function on our `simplified_instances`.

In [27]:
```
for i in xrange(5):
    print '\n# partitions:', i
    for j, partition in enumerate(partition_instances(simplified_instances, i)):
        print 'partition {}: {}'.format(j, partition)
```

```
# partitions: 0

# partitions: 1
partition 0: [[0, 1, 2], [1, 2, 3], [2, 3, 4], [3, 4, 5], [4, 5, 6]]

# partitions: 2
partition 0: [[0, 1, 2], [2, 3, 4], [4, 5, 6]]
partition 1: [[1, 2, 3], [3, 4, 5]]

# partitions: 3
partition 0: [[0, 1, 2], [3, 4, 5]]
partition 1: [[1, 2, 3], [4, 5, 6]]
partition 2: [[2, 3, 4]]

# partitions: 4
partition 0: [[0, 1, 2], [4, 5, 6]]
partition 1: [[1, 2, 3]]
partition 2: [[2, 3, 4]]
partition 3: [[3, 4, 5]]
```

Returning our attention to the UCI mushroom dataset, the following will partition our `clean_instances` into 10 relatively equally sized disjoint subsets. We will a list comprehension to print out the length of each partition

In [28]:
```
partitions = partition_instances(clean_instances, 10)
print [len(partition) for partition in partitions]
```

```
[565, 565, 565, 565, 564, 564, 564, 564, 564, 564]
```

The following variation does not use a list comprehension.

In [29]:
```
for partition in partitions:
    print len(partition),  # note the comma at the end
print
```

```
565 565 565 565 564 564 564 564 564 564
```

The following shows the different trees that are constructed based on partition 0 (first 10th) of `clean_instances`, partitions 0 and 1 (first 2/10ths) of `clean_instances` and all `clean_instances`.

In [30]:
```
tree0 = create_decision_tree(partitions[0])
print 'Tree trained with {} instances:'.format(len(partitions[0]))
pprint(tree0)

tree1 = create_decision_tree(partitions[0] + partitions[1])
print '\nTree trained with {} instances:'.format(len(partitions[0] + partitions[1]))
pprint(tree1)

tree = create_decision_tree(clean_instances)
print '\nTree trained with {} instances:'.format(len(clean_instances))
pprint(tree)
```

```
Tree trained with 565 instances:
{5: {'a': 'e',
     'c': 'p',
     'f': 'p',
     'l': 'e',
     'm': 'p',
     'n': {20: {'k': 'e', 'n': 'e', 'r': 'p', 'w': 'e'}},
     'p': 'p'}}

Tree trained with 1130 instances:
{5: {'a': 'e',
     'c': 'p',
     'f': 'p',
     'l': 'e',
     'm': 'p',
     'n': {20: {'k': 'e',
                'n': 'e',
                'r': 'p',
                'w': {21: {'c': 'p', 'v': 'e', 'y': 'e'}}}},
     'p': 'p'}}

Tree trained with 5644 instances:
{5: {'a': 'e',
     'c': 'p',
     'f': 'p',
     'l': 'e',
     'm': 'p',
     'n': {20: {'k': 'e',
                'n': 'e',
                'r': 'p',
                'w': {21: {'c': 'p', 'v': 'e', 'y': 'e'}}}},
     'p': 'p'}}
```

The only difference between the first two trees - *tree0* and *tree1* - is that in the first tree, instances with no `odor` (attribute index `5` is `'n'`) and a `spore-print-color` of white (attribute `20 = 'w'`) are classified as `edible` (`'e'`). With additional training data in the 2nd partition, an additional distinction is made such that instances with no `odor`, a white `spore-print-color` and a clustered `population` (attribute `21 = 'c'`) are classified as `poisonous` (`'p'`), while all other instances with no `odor` and a white `spore-print-color` (and any other value for the `population` attribute) are classified as `edible` (`'e'`).

Note that there is no difference between `tree1` and `tree` (the tree trained with all instances). This early convergence on an optimal model is uncommon on mos datasets (outside the UCI repository).

Now that we can partition our instances into subsets, we can use these subsets to construct different-sized training sets in the process of computing a learning curve.

We will start off with an initial training set consisting only of the first partition, and then progressively extend that training set by adding a new partition during eac iteration of computing the learning curve.

The [`list.extend(L)` (http://docs.python.org/2/tutorial/datastructures.html#more-on-lists)](http://docs.python.org/2/tutorial/datastructures.html#more-on-lists) method enables us to extend `list` by appending all the items in another list, `L`, to the end of `list`.

In [31]:
```
x = [1, 2, 3]
x.extend([4, 5])
print x
```

```
[1, 2, 3, 4, 5]
```

We can now define the function, `compute_learning_curve(instances, num_partitions=10)`, that will take a list of `instances`, partition it into `num_partitions` relatively equally sized disjoint partitions, and then iteratively evaluate the accuracy of models trained with an incrementally increasing combination of instances in the first `num_partitions - 1` partitions then tested with instances in the last partition. That is, a model trained with the first partit will be constructed (and tested), then a model trained with the first 2 partitions will be constructed (and tested), and so on.

The function will return a list of `num_partitions - 1` tuples representing the size of the training set and the accuracy of a tree trained with that set (and testec the `num_partitions - 1` set). This will provide some indication of the relative impact of the size of the training set on model performance.

In [32]:
```
def compute_learning_curve(instances, num_partitions=10):
    '''Returns a list of training sizes and scores for incrementally increasing partitions.

    The list contains 2-element tuples, each representing a training size and score.
    The i-th training size is the number of instances in partitions 0 through num_partitions - 2.
    The i-th score is the accuracy of a tree trained with instances
    from partitions 0 through num_partitions - 2
    and tested on instances from num_partitions - 1 (the last partition).'''

    partitions = partition_instances(instances, num_partitions)
    testing_instances = partitions[-1][:]
    training_instances = []
    accuracy_list = []
    for i in xrange(0, num_partitions - 1):
        # for each iteration, the training set is composed of partitions 0 through i - 1
        training_instances.extend(partitions[i][:])
        tree = create_decision_tree(training_instances)
        partition_accuracy = classification_accuracy(tree, testing_instances)
        accuracy_list.append((len(training_instances), partition_accuracy))
    return accuracy_list

accuracy_list = compute_learning_curve(clean_instances)
print accuracy_list
```

```
[(565, (562, 2, 0.9964539007092199)), (1130, (564, 0, 1.0)), (1695, (564, 0, 1.0)), (2260, (564, 0, 1.0)), (2824,
(564, 0, 1.0)), (3388, (564, 0, 1.0)), (3952, (564, 0, 1.0)), (4516, (564, 0, 1.0)), (5080, (564, 0, 1.0))]
```

Due to the quick convergence on an optimal decision tree for classifying the UCI mushroom dataset, we can use a larger number of smaller partitions to see a lit more variation in acccuracy performance.

In [33]:
```
accuracy_list = compute_learning_curve(clean_instances, 100)
print accuracy_list[:10]
```

```
[(57, (55, 1, 0.9821428571428571)), (114, (56, 0, 1.0)), (171, (55, 1, 0.9821428571428571)), (228, (56, 0, 1.0)),
(285, (56, 0, 1.0)), (342, (56, 0, 1.0)), (399, (56, 0, 1.0)), (456, (56, 0, 1.0)), (513, (56, 0, 1.0)), (570, (56
, 0, 1.0))]
```

Object-Oriented Programming: Defining a Python Class to Encapsulate a Simple Decision Tree

The simple decision tree defined above uses a Python dictionary for its representation. One can imagine using other data structures, and/or extending the decisi tree to support confidence estimates, numeric features and other capabilities that are often included in more fully functional implementations. To support future extensibility, and hide the details of the representation from the user, it would be helpful to have a user-defined class for simple decision trees.

Python is an object-oriented programming (https://en.wikipedia.org/wiki/Object-oriented_programming) language, offering simple syntax and semantics for defir classes and instantiating objects of those classes. *[It is assumed that the reader is already familiar with the concepts of object-oriented programming]*

A Python class (http://docs.python.org/2/tutorial/classes.html) starts with the keyword **`class`** followed by a class name (identifier), a colon ('**`:`**'), and then any number of statements, which typically take the form of assignment statements for class or instance variables and/or function definitions for class methods. All statements are indented to reflect their inclusion in the class definition.

The members - methods, class variables and instance variables - of a class are accessed by prepending `self.` to each reference. Class methods always includ `self` as the first parameter.

All class members in Python are *public* (accessible outside the class). There is no mechanism for *private* class members, but identifiers with leading double underscores (*__member_identifier*) are 'mangled' (translated into *_class_name__member_identifier*), and thus not directly accessible outside their class, and can used to approximate private members by Python programmers.

There is also no mechanism for *protected* identifiers - accessible only within a defining class and its subclasses - in the Python language, and so Python programmers have adopted the convention of using a single underscore (*_identifier*) at the start of any identifier that is intended to be protected (i.e., not to be accessed outside the class or its subclasses).

Some Python programmers only use the single underscore prefixes and avoid double underscore prefixes due to unintended consequences that can arise when names are mangled. The following warning about single and double underscore prefixes is issued in Code Like a Pythonista (http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#naming):

> try to avoid the __private form. I never use it. Trust me. If you use it, you WILL regret it later

We will follow this advice and avoid using the double underscore prefix in user-defined member variables and methods.

Python has a number of pre-defined special method names (http://docs.python.org/2/reference/datamodel.html#special-method-names), all of which are denote by leading and trailing double underscores. For example, the `object.__init__(self[, ...])` (http://docs.python.org/2/reference/datamodel.html#object.__init__) method is used to specify instructions that should be executed whenever a new object of a class is instantiated.

The code below defines a class, `SimpleDecisionTree`, with a single pseudo-protected member variable `_tree` and a pseudo-protected tree construction method `_create()`, two public methods - `classify()` and `pprint()` - and an initialization method that takes an optional list of training `instances` and a `target_attribute_index`.

The `_create()` method is identical to the `create_decision_tree()` function above, with the inclusion of the `self` parameter (as it is now a class method). `classify()` method is a similarly modified version of the `classify()` and `classification_accuracy()` functions above, with references to `tree` conver to `self._tree`. The `pprint()` method prints the tree in a human-readable format.

Note that other machine learning libraries may use different terminology for the methods we've defined here. For example, in the `sklearn.tree.DecisionTreeClassifier` (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html) class (and in most `sklearn` classifier classes), the method for constructing a classifier is named `fit()` (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier.fit) - since it "fits" the data to a model - and the method for classifying instances is named `predict()` (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier.predict) - since it is predicting the class label for instance.

Most comments and the use of the trace parameter have been removed to make the code more compact, but are included in the version found in `SimpleDecisionTree.py`.

In [34]:
```
class SimpleDecisionTree:

    _tree = {} # this instance variable becomes accessible to class methods via self._tree

    def __init__(self, instances=None, target_attribute_index=0): # note the use of self as the first parameter
        if instances:
            self._tree = self._create(instances, range(1, len(instances[0])), target_attribute_index)

    def _create(self, instances, candidate_attribute_indexes, target_attribute_index=0, default_class=None):
        class_labels_and_counts = Counter([instance[target_attribute_index] for instance in instances])
        if not instances or not candidate_attribute_indexes:
            return default_class
        elif len(class_labels_and_counts) == 1:
            class_label = class_labels_and_counts.most_common(1)[0][0]
            return class_label
        else:
            default_class = simple_ml.majority_value(instances, target_attribute_index)
            best_index = simple_ml.choose_best_attribute_index(instances, candidate_attribute_indexes, target_attribute
dex)
            tree = {best_index:{}}
            partitions = simple_ml.split_instances(instances, best_index)
            remaining_candidate_attribute_indexes = [i for i in candidate_attribute_indexes if i != best_index]
            for attribute_value in partitions:
                subtree = self._create(
                    partitions[attribute_value],
                    remaining_candidate_attribute_indexes,
                    target_attribute_index,
                    default_class)
                tree[best_index][attribute_value] = subtree
            return tree

    # calls the internal "protected" method to classify the instance given the _tree
    def classify(self, instance, default_class=None):
        return self._classify(self._tree, instance, default_class)

    # a method intended to be "protected" that can implement the recursive algorithm to classify an instance given a tr
    def _classify(self, tree, instance, default_class=None):
        if not tree:
            return default_class
        if not isinstance(tree, dict):
            return tree
        attribute_index = tree.keys()[0]
        attribute_values = tree.values()[0]
        instance_attribute_value = instance[attribute_index]
        if instance_attribute_value not in attribute_values:
            return default_class
        return self._classify(attribute_values[instance_attribute_value], instance, default_class)

    def classification_accuracy(self, instances, default_class=None):
        predicted_labels = [self.classify(instance, default_class) for instance in instances]
        actual_labels = [x[0] for x in instances]
        counts = Counter([x == y for x, y in zip(predicted_labels, actual_labels)])
        return counts[True], counts[False], float(counts[True]) / len(instances)

    def pprint(self):
        pprint(self._tree)
```

The following statements instantiate a `SimpleDecisionTree`, using all but the last 20 `clean_instances`, prints out the tree using its `pprint()` method, anc then uses the `classify()` method to print the classification of the last 20 `clean_instances`.

In [35]:
```
simple_decision_tree = SimpleDecisionTree(training_instances)
simple_decision_tree.pprint()
print
for instance in testing_instances:
    predicted_label = simple_decision_tree.classify(instance)
    actual_label = instance[0]
    print 'Model: {}; truth: {}'.format(predicted_label, actual_label)
print
print 'Classification accuracy:', simple_decision_tree.classification_accuracy(testing_instances)
```

```
{5: {'a': 'e',
     'c': 'p',
     'f': 'p',
     'l': 'e',
     'm': 'p',
     'n': {20: {'k': 'e',
                'n': 'e',
                'r': 'p',
                'w': {21: {'c': 'p', 'v': 'e', 'y': 'e'}}}},
     'p': 'p'}}

Model: p; truth: p
Model: p; truth: p
Model: p; truth: p
Model: e; truth: e
Model: e; truth: e
Model: p; truth: p
Model: e; truth: e
Model: e; truth: e
Model: e; truth: e
Model: p; truth: p
Model: e; truth: e
Model: e; truth: e
Model: e; truth: e
Model: p; truth: p
Model: e; truth: e
Model: e; truth: e
Model: e; truth: e
Model: e; truth: e
Model: p; truth: p
Model: p; truth: p

Classification accuracy: (20, 0, 1.0)
```

Navigation

Notebooks in this primer: