Name: Matthew Seman

In [1]:
```
import re
import pandas as pd
import numpy as np
```

In [2]:
```
class Node:
    """ Node class for a decision tree. """
    def __init__(self, names):
        self.names = names

    def classify(x):
        """ Handled by the subclasses. """
        return None

    def dump(self, indent):
        """ Handled by the subclasses. """
        return None


class Leaf(Node):
    def __init__(self, names, value):
        Node.__init__(self, names)
        self.value = value

    def classify(self, x):
        return self.value

    def dump(self, indent):
        print(' %d' % self.value)


class Split(Node):
    def __init__(self, names, var, left, right):
        Node.__init__(self, names)
        self.var = var
        self.left = left
        self.right = right

    def classify(self, x):
        if x[self.var] == 0:
            return self.left.classify(x)
        else:
                return self.right.classify(x)

    def dump(self, indent):
        if indent > 0:
            print('')
        for i in range(0, indent):
            print('| ', end='')
        print('%s = 0 :' % self.names[self.var],end='')
        self.left.dump(indent+1)
        for i in range(0, indent):
            print('| ', end='')
        print('%s = 1 :' % self.names[self.var],end='')
        self.right.dump(indent+1)
```

Helper function computes entropy of Bernoulli distribution with parameter p

In [3]:
```
def entropy(p):
    # >>>> YOUR CODE GOES HERE <<<<

    #Choose parameter p to be probability between 0 and 1

    entropy = 0;

    if p < 0.000000001:
        entropy = 0
    elif p > 0.999999999:
        entropy = 0
    else:
        entropy = -p*np.log2(p) - (1-p)*np.log2(1-p)

    return entropy;
```

Compute information gain for a particular split, given the counts

py_pxi : number of occurences of y=1 with x_i=1 for all i=1 to n

pxi : number of occurrences of x_i=1

py : number of ocurrences of y=1

In [4]:
```
def infogain(py_pxi, pxi, py, total):
    # >>>> YOUR CODE GOES HERE <<<<
    #parameter:
    #py_pxi - number of times target y = 1 and x_i = 1
    #pxi - number of x_i = 1
    #py - number of times target is 1

    total_entropy = entropy(py/total)

    if pxi == 0:
        weighted_entropy = -(1 - pxi/total)*entropy((py-py_pxi)/(total-pxi))
    elif pxi == total:
        weighted_entropy = -(pxi/total)*entropy(py_pxi/pxi)
    else:
        weighted_entropy = -(pxi/total)*entropy(py_pxi/pxi) -(1 - pxi/total)*entropy((py-py_pxi)/(total-pxi))

    infogain = total_entropy + weighted_entropy

    return infogain;
```

OTHER SUGGESTED HELPER FUNCTIONS:

-collect counts for each variable value with each class label

-find the best variable to split on, according to mutual information

-partition data based on a given variable

In [5]:

```
def collect_counts(data, varnames):
    df = pd.DataFrame(data)

    px = []
    py_px = []
    py = df.iloc[:,len(df.columns)-1].sum()
    total = len(data)

    i = 0
    while i < len(df.columns)-1:
        px.append(sum(df.iloc[:,i])) # number of times ith attribute is 1
        py_px.append(sum(np.where((df.iloc[:,i] + df.iloc[:,len(df.columns)-1]) == 2, 1, 0))) # number of times ith attri
        i = i + 1

    return (py_px, px, py, total)
```

In [6]:
```
def find_best_feature(data, varnames, py_px, px, py, total):
    df = pd.DataFrame(data)

    #(py_px, px, py, total) = collect_counts(data, varnames)

    ig = []

    i = 0
    while i < len(df.columns)-1:
        ig.append(infogain(py_px[i], px[i], py, total))
        i = i + 1

    ig_max = max(ig)

    hg_index = ig.index(max(ig))

    return hg_index, ig_max
```

In [7]:
```
def partition_dataset(data, varnames):
    df = pd.DataFrame(data)

    (py_px, px, py, total) = collect_counts(data, varnames)

    hg_index, ig_max = find_best_feature(data, varnames, py_px, px, py, total)

    best_index = df.columns.values[hg_index]

    best_var_name = varnames[best_index]

    grouped = df.groupby(df[best_index])

    if px[hg_index] == total:          # number of 1's in best feature equals total
        data_bf_1 = grouped.get_group(1)
        data_bf_1 = data_bf_1.drop(best_index, axis = 1)
        data_bf_0 = []
    elif px[hg_index] == 0:            # number of 1's in best feature equals zero
        data_bf_0 = grouped.get_group(0)
        data_bf_0 = data_bf_0.drop(best_index, axis = 1)
        data_bf_1 = []
    else:
        data_bf_1 = grouped.get_group(1)
        data_bf_0 = grouped.get_group(0)
        data_bf_1 = data_bf_1.drop(best_index, axis = 1)
        data_bf_0 = data_bf_0.drop(best_index, axis = 1)
    return data_bf_0, data_bf_1, best_index
```

In [8]:

```
# Load data from a file
def read_data(filename):
    f = open(filename, 'r')
    p = re.compile(',')
    data = []
    header = f.readline().strip()
    varnames = p.split(header)
    namehash = {}
    for l in f:
        data.append([int(x) for x in p.split(l.strip())])
    return (data, varnames)
```

Build tree in a top-down manner, selecting splits until we hit a pure leaf or all splits look bad.

In [9]:

```
def build_tree(data, varnames):
    # >>>> YOUR CODE GOES HERE <<<<
    # For now, always return a leaf predicting "1":
    df = pd.DataFrame(data)

    (py_px, px, py, total) = collect_counts(data, varnames)


    #best_feature_index, ig_max = find_best_feature(data, varnames, py_px, px, py, total)

    if py == 0:                  #Pure nodes
        root = Leaf(varnames, 0)
    elif py == total:
        root = Leaf(varnames, 1)
    elif len(df.columns) == 1:      #No more attributes to split
        if py/total > 0.5:
            root = Leaf(varnames, 1)
        else:
            root = Leaf(varnames, 0)
    else:
        #best_feature_index, ig_max = find_best_feature(data, varnames, py_px, px, py, total)
        data_bf_0, data_bf_1, best_index = partition_dataset(data, varnames)
        if len(data_bf_0) == 0:
            left = Leaf(varnames, 0)
        else:
            left = build_tree(data_bf_0, varnames)
        if len(data_bf_1) == 0:
            right = Leaf(varnames, 0)
        else:
            right = build_tree(data_bf_1, varnames)

        root = Split(varnames, best_index, left, right)

    #root.dump(0)
    return root
```

Here we load data. Each example is a list of attribute values, where the last element in the list is the class value.

In [10]:

```
agaricus = ["agaricuslepiotatrain1.csv",
              "agaricuslepiotatest1.csv",
              "agaricuslepiotatest1.csv"]

dataset1 = ["data_sets1/training_set.csv",
            "data_sets1/validation_set.csv",
            "data_sets1/test_set.csv"]


dataset2 = ["data_sets2/training_set.csv",
            "data_sets2/validation_set.csv",
            "data_sets2/test_set.csv"]
# pick the dataset you want to use this time
dataset = agaricus

(train, varnames) = read_data(dataset[0])
(validation, validationvarnames) = read_data(dataset[1])
(test, testvarnames) = read_data(dataset[2])
```

In [11]:

```
pd.DataFrame(train).head()
```

Out[11]:

	0	1	2	3	4	5	6	7	8	9	...	113	114	115	116	117	118	119	120	121	122
0	0	0	1	0	0	0	0	0	0	1	...	0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0	0	0	1	...	0	0	1	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	1	...	0	0	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	0	1	0	...	0	0	0	0	0	0	1	0	0	1

	0	1	2	3	4	5	6	7	8	9	...	113	114	115	116	117	118	119	120	121	122
4	0	0	1	0	0	0	0	0	0	1	...	0	0	1	0	0	0	0	0	0	0

5 rows × 123 columns

In [12]:

```
root = build_tree(train, varnames)
```

Build the decision tree

In [13]:

```
root.dump(0)
```

```
odor-foul = 0 :
| gill-size-broad = 0 :
| | odor-none = 0 :
| | | gill-spacing-close = 0 :
| | | | bruises?-bruises = 0 : 1
| | | | bruises?-bruises = 1 : 0
| | | gill-spacing-close = 1 : 1
| | odor-none = 1 :
| | | stalk-surface-above-ring-silky = 0 :
| | | | bruises?-bruises = 0 : 0
| | | | bruises?-bruises = 1 : 1
| | | stalk-surface-above-ring-silky = 1 : 1
| gill-size-broad = 1 :
| | spore-print-color-green = 0 : 0
| | spore-print-color-green = 1 : 1
odor-foul = 1 : 1
```

Calcuating the accuracy

In [14]:

```
def accuracy(data):
    correct = 0
    # The position of the class label is the last element in the list.
    yi = len(data[0]) - 1
    for x in data:
    # Classification is done recursively by the node class.
    # This should work as-is.
        pred = root.classify(x)
        if pred == x[yi]:
            correct += 1
        acc = float(correct)/len(data)
    return acc;
```

In [15]:

```
print("Train Accuracy: {}".format(accuracy(train)))
```

```
Train Accuracy: 1.0
```

In [16]:

```
print("Test Accuracy: {}".format(accuracy(test)))
```

```
Test Accuracy: 0.9792941176470589
```