DAMLAS - Machine Learning At Scale

Assignment - HW4

Data Analytics and Machine Learning at Scale Target, Minneapolis

Name: Mark von Oven
Class: DAMLAS (Section Summer 2016)
Email: Mark.Vonoven@Target.com
Week: 04

In [1]:
```
%%javascript
/***************************************************************************************************
*
Known Mathjax Issue with Chrome - a rounding issue adds a border to the right of mathjax marku
p
https://github.com/mathjax/MathJax/issues/1300
A quick hack to fix this based on stackoverflow discussions:
http://stackoverflow.com/questions/34277967/chrome-rendering-mathjax-equations-with-a-trailing
-vertical-line
**************************************************************************************************
*/

$('.math>span').css("border-left-color","transparent")
```

In [2]:
```
%reload_ext autoreload
%autoreload 2
```

In [3]:
```
import os
import sys #current as of 9/26/2015
import pyspark
from pyspark.sql import SQLContext
# We can give a name to our app (to find it in Spark WebUI) and configure execution mode
# In this case, it is local multicore execution with "local[*]"
app_name = "example-logs"
master = "local[*]"
conf = pyspark.SparkConf().setAppName(app_name).setMaster(master)
sc = pyspark.SparkContext(conf=conf)
sqlContext = SQLContext(sc)
print(sc)
print(sqlContext)
# Import some libraries to work with dates
import dateutil.parser
import dateutil.relativedelta as dateutil_rd
```

```
<pyspark.context.SparkContext object at 0x7f78a452a198>
<pyspark.sql.context.SQLContext object at 0x7f78a468f4a8>
```

Table of Contents

1 Instructions

Back to Table of Contents

- Homework submissions are due by Thursday, 08/18/2016 at 11AM (CT).

- Prepare a single Jupyter notebook (not a requirment), please include questions, and question numbers in the questions and in the responses. Submit your homework notebook via the following form:
  - Submission Link - Google Form (http://goo.gl/forms/er3OFr5eCMWDngB72)

Documents:

- IPython Notebook, published and viewable online.
- PDF export of IPython Notebook.

2 Useful References

Back to Table of Contents

- Lecture Slides on Decision Trees and Ensembles (https://www.dropbox.com/s/lm4vuocqoq6mq7k/Lecture-13-Decision-Trees-PLanet.pdf?dl=0)
- Chapter 17 on decision Trees, https://www.dropbox.com/s/5ca98ah5chqlcmn/Data_Science_from_Scratch%20%281%29.pdf?dl=0 (https://www.dropbox.com/s/5ca98ah5chqlcmn/Data_Science_from_Scratch%20%281%29.pdf?dl=0) [Please do not share this PDF]
- Karau, Holden, Konwinski, Andy, Wendell, Patrick, & Zaharia, Matei. (2015). Learning Spark: Lightning-fast big data analysis. Sebastopol, CA: O'Reilly Publishers.
- Hastie, Trevor, Tibshirani, Robert, & Friedman, Jerome. (2009). The elements of statistical learning: Data mining, inference, and prediction (2nd ed.). Stanford, CA: Springer Science+Business Media. **(Download for free here (http://statweb.stanford.edu/~tibs/ElemStatLearn/printings/ESLII_print10.pdf))**
- Ryza, Sandy, Laserson, Uri, Owen, Sean, & Wills, Josh. (2015). Advanced analytics with Spark: Patterns for learning from data at scale. Sebastopol, CA: O'Reilly Publishers.

3. HW4

Back to Table of Contents

HW4.0 Final Project description

Please prepare your project description using the following format

- 200 words abstract
- data source and description
- pipeline of steps (in a block diagram)
- Metrics for success

PLEASE NOTE: We will probably have project team sizes of 3 people plus/minus 1

(Mark Von Oven, Thomas Reed, Earl Sun)

Situation: In the online world, we have a record of what guests browse. The is no such analog for the physical world. We know what guests buy, but do not know the path they took and what they saw along the way. There is value in understanding the likely path guests take, such as knowing what promotional material could be presented along their way.

Task: LED lights are being installed in stores, with additional equipment capable of tracking guests, under certain conditions. Using this position information, we can plot the path guests take, what items they came in closer contact with, and which ones they potentially spent more time considering but did not buy. Actions: Our team will develop the analysis based on position and orientation information, and item location within the store. The result of the analysis would be a list of items guests scrutinized closely, the path they took, and possible opportunities to advertise. Data Source: Position data is being recorded by a third party, and apparently also being stored within Target's big data environment (Big Red). The information contains X,Y coordinates within 4 inches when the phone app and camera are active, within 8-10 feet using bluetooth only, a time stamp for each location, and the orientation of the phone. We will also need to secure store layout and item location within the planograms.

Pipeline of Steps:

1. Obtain position data, store layout, item location
2. Algorithm to create paths and speed from X,Y data. Path in terms of vertical and horizontal paths, guest speed, and potentially a sense of direction. a) Break path into line segments b) Within each segment, determine speed (e.g., cruising, perusing, assessing, standing) c) Combine speed with any phone directional information (e.g., phone pointed to the sides)
3. Algorithm to approximate location when accuracy changes from LED to bluetooth modes.
4. Algorithm to estimate which types of items are likely getting more attention.
5. Determine which items received attention but were not purchased.

Metrics for Success:

1. We can list items that guests spent significant time assessing but did not buy.
2. Determine how efficient are guests path. In other words, are trips focused on efficiently filling a cart, or guests take the long route to peruse?

HW4.1 Build a decision to predict whether you can play tennis or not

Back to Table of Contents

Decision Trees

Write a program in Python (or in Spark; this part is optional) to implement the ID3 decision tree algorithm. You should build a tree to predict PlayTennis, based on the other attributes (but, do not use the Day attribute in your tree.). You should read in a space delimited dataset in a file called dataset.txt and output to the screen your decision tree and the training set accuracy in some readable format. For example, here is the tennis dataset. The first line will contain the names of the fields:

```
Day outlook temperature humidity wind playtennis
d1 sunny hot high FALSE no
d2 sunny hot high TRUE no
d3 overcast hot high FALSE yes
d4 rainy mild high FALSE yes
d5 rainy cool normal FALSE yes
d6 rainy cool normal TRUE no
d6 overcast cool normal TRUE yes
d7 sunny mild high FALSE no
d8 sunny cool normal FALSE yes
d9 rainy mild normal FALSE yes
d10 sunny mild normal TRUE yes
d11 overcast mild high TRUE yes
d12 overcast hot normal FALSE yes
d12 rainy mild high TRUE no
```

The last column is the classification attribute, and will always contain contain the values yes or no.

For output, you can choose how to draw the tree so long as it is clear what the tree is. You might find it easier if you turn the decision tree on its side, and use indentation to show levels of the tree as it grows from the left. For example:

```
outlook = sunny
|  humidity = high: no
|  humidity = normal: yes
outlook = overcast: yes
outlook = rainy
|  windy = TRUE: no
|  windy = FALSE: yes
```

You don't need to make your tree output look exactly like above: feel free to print out something similarly readable if you think it is easier to code.

You may find Python dictionaries especially useful here, as they will give you a quick an easy way to help manage counting the number of times you see a particular attribute.

Here are some FAQs that I've gotten in the past regarding this assignment, and some I might get if I don't answer them now.

Should my code work for other datasets besides the tennis dataset? Yes. We will give your program a different dataset to try it out with. You may assume that our dataset is correct and well-formatted, but you should not make assumptions regrading number of rows, number of columns, or values that will appear within. The last column will also be the classification, and will always contain yes or no values.

Is it possible that some value, like "normal," could appear in more than one column? Yes. In addition to the column "humidity", we might have had another column called "skycolor" which could have values "normal," "weird," and "bizarre."

Could "yes" and "no" appear as possible values in columns other than the classification column? Yes. In addition to the classification column "playtennis," we might have had another column called "seasonalweather" which would contain "yes" and "no."

In [4]:
```
%%writefile tennis.txt
Day outlook temperature humidity wind playtennis
d1 sunny hot high FALSE no
d2 sunny hot high TRUE no
d3 overcast hot high FALSE yes
d4 rainy mild high FALSE yes
d5 rainy cool normal FALSE yes
d6 rainy cool normal TRUE no
d6 overcast cool normal TRUE yes
d7 sunny mild high FALSE no
d8 sunny cool normal FALSE yes
d9 rainy mild normal FALSE yes
d10 sunny mild normal TRUE yes
d11 overcast mild high TRUE yes
d12 overcast hot normal FALSE yes
d12 rainy mild high TRUE no
```

```
Writing tennis.txt
```

In [14]:

```
from __future__ import division
from collections import Counter, defaultdict
from functools import partial
import math, random

def entropy(class_probabilities):
    """given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2) for p in class_probabilities if p)

def class_probabilities(labels):
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)

def partition_entropy(subsets):
    """find the entropy from this partition of data into subsets"""
    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )

def group_by(items, key_fn):
    """returns a defaultdict(list), where each input item
    is in the list whose key is key_fn(item)"""
    groups = defaultdict(list)
    for item in items:
        key = key_fn(item)
        groups[key].append(item)
    return groups

def partition_by(inputs, attribute):
    """returns a dict of inputs partitioned by the attribute
    each input is a pair (attribute_dict, label)"""
    return group_by(inputs, lambda x: x[0][attribute])
```

```
def partition_entropy_by(inputs,attribute):
    """computes the entropy corresponding to the given partition"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())

def classify(tree, input):
    """classify the input using the given decision tree"""

    # if this is a leaf node, return its value
    #if tree in [True, False]:
    #    return tree

    if tree == True:
        return 'yes'
    if tree == False:
        return 'no'

    # otherwise find the correct subtree
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)  # None if input is missing attribute

    if subtree_key not in subtree_dict: # if no subtree for key,
        subtree_key = None              # we'll use the None subtree

    subtree = subtree_dict[subtree_key] # choose the appropriate subtree
    return classify(subtree, input)     # and use it to classify the input

def build_tree_id3(inputs, split_candidates=None):

    # if this is our first pass,
    # all keys of the first input are split candidates
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # count Trues and Falses in the inputs
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues
    if num_trues == 0:                  # if only Falses are left
        return False                    # return a "False" leaf

    if num_falses == 0:                 # if only Trues are left
        return True                     # return a "True" leaf

    if not split_candidates:            # if no split candidates left
        return num_trues >= num_falses  # return the majority leaf

    # otherwise, split on the best attribute
    best_attribute = min(split_candidates,
        key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # recursively build the subtrees
    subtrees = { attribute : build_tree_id3(subset, new_candidates)
                 for attribute, subset in partitions.items() }

    subtrees[None] = num_trues > num_falses # default case

    return (best_attribute, subtrees)

def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]

def printFormatTree(tree, indent=0):
    if type(tree) == tuple:
        print ('-----' * indent, tree[0])
        printFormatTree(tree[1], indent)
    if type(tree) == dict:
        indent += 1
        for state in tree.keys():
            if type(tree[state])== tuple:
                print ('-----' * indent, state)
                indent += 1
                printFormatTree(tree[state], indent)
                indent -= 1
            elif type(tree[state])== str:
                print ('-----' * indent, state, ":", tree[state])
        indent -= 1

if __name__ == "__main__":

    trainingset = 'tennis.txt'
    f = open(trainingset)
    rowcnt = 1
    i = 0
    header = []
    t1 = {}
    inputs = []

    for line in f:
        linelist = line.split()
        linelist.pop(0)   #remove the day column
        totalcols = len(linelist)

        if rowcnt == 1:
            for w in linelist:
                header.append(w)
        else:
            for w in linelist:
                if i == (totalcols - 1):
                    i=0
                    break
                else:
                    t1[header[i]]=w
                    i += 1
            if w == 'yes':
                inputs.append((t1, True))
            if w == 'no':
                inputs.append((t1, False))
        rowcnt += 1
        t1 = {}

    for key in header[:totalcols-1]:
        print (key, partition_entropy_by(inputs, key))
    print

#    senior_inputs = [(input, label)
#                     for input, label in inputs if input["level"] == "Senior"]

#    for key in ['lang', 'tweets', 'phd']:
#        print (key, partition_entropy_by(senior_inputs, key))
#    print

    print ("building the tree")
    tree = build_tree_id3(inputs)
    print (tree)
    printFormatTree(tree, indent=0)

    print ("d1", classify(tree,
        { "outlook" : "sunny",
          "temperature" : "hot",
          "humidity" : "high",
          "wind" : "FALSE"} ) )
```

```
outlook 0.6935361388961919
temperature 0.9110633930116764
humidity 0.7884504573082896
wind 0.8921589282623617
building the tree
('outlook', {'sunny': ('humidity', {'high': False, None: False, 'normal': True}), None: Tru
e, 'rainy': ('wind', {None: True, 'TRUE': False, 'FALSE': True}), 'overcast': True})
 outlook
----- sunny
---------- humidity
----- rainy
---------- wind
d1 no
```

In [ ]:
```
('outlook', {'sunny':
             ('humidity', {'high': False,
                           None: False,
                           'normal': True}),
             None: True,
             'rainy':
             ('wind', {None: True,
                       'TRUE': False,
                       'FALSE': True}),
             'overcast': True}
)
```

In [12]:
```
def treeFormat(tree, indent=0):
    if type(tree)==tuple:
        print(tree[0] + ' = ')
        treeFormat(tree[1], )
    if type(tree)==dict:
        print('dict')

treeFormat(tree, indent=0)
```

```
outlook =
```

HW4.1.1 What is the classification accuracy of the tree on the training data?

In [31]:

```
testset = 'tennis.txt'
tst = open(testset)

rowcnt = 1
i = 0
header = []
t1 = {}
inputs = []

for line in tst:
    linelist = line.split()
    totalcols = len(linelist)

    if rowcnt == 1:
        for w in linelist:
            header.append(w)
    else:
        for w in linelist:
            if i == (totalcols - 1):
                i=0
                break
            else:
                t1[header[i]]=w
                i += 1
        inputs.append((t1, w))
#         if w == 'yes':
#             inputs.append((t1, True))
#         if w == 'no':
#             inputs.append((t1, False))
    rowcnt += 1
    t1 = {}

#Run through the testset and find the classification error
correct = 0
possible = len(inputs)
for row in inputs:
    if classify(tree, row[0]) == row[1]:
        correct += 1

if correct > 0:
    score = (correct / possible)*100
else:
    score = 0

print('The classification accuracy was %s' % score + '%')
```

```
The classification accuracy was 100.0%
```

HW4.1.2 Is it possible to produce some set of correct training examples that will get the algorihtm to include the attribute Temperature in the learned tree, even though the true target concept is independent of Temperature? if no, explain. If yes, give such a set.

In [36]:
```
%%writefile tennis_altertemp.txt
Day outlook temperature humidity wind playtennis
d1 sunny hot high FALSE no
d2 sunny hot high TRUE no
d3 sunny hot high FALSE yes
d4 sunny hot high FALSE yes
d5 sunny hot normal FALSE yes
d6 sunny mild normal TRUE no
d6 overcast hot normal TRUE yes
d7 sunny mild high FALSE no
d8 sunny mild normal FALSE yes
d9 rainy mild normal FALSE yes
d10 sunny mild normal TRUE yes
d11 overcast mild high TRUE yes
d12 overcast mild normal FALSE yes
d12 rainy mild high TRUE no
```

```
Overwriting tennis_altertemp.txt
```

In [37]:

```
from __future__ import division
from collections import Counter, defaultdict
from functools import partial
import math, random

def entropy(class_probabilities):
    """given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2) for p in class_probabilities if p)

def class_probabilities(labels):
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)

def partition_entropy(subsets):
    """find the entropy from this partition of data into subsets"""
    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )

def group_by(items, key_fn):
    """returns a defaultdict(list), where each input item
    is in the list whose key is key_fn(item)"""
    groups = defaultdict(list)
    for item in items:
        key = key_fn(item)
        groups[key].append(item)
    return groups

def partition_by(inputs, attribute):
    """returns a dict of inputs partitioned by the attribute
    each input is a pair (attribute_dict, label)"""
    return group_by(inputs, lambda x: x[0][attribute])
```

```
def partition_entropy_by(inputs,attribute):
    """computes the entropy corresponding to the given partition"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())

def classify(tree, input):
    """classify the input using the given decision tree"""

    # if this is a leaf node, return its value
    #if tree in [True, False]:
    #    return tree

    if tree == True:
        return 'yes'
    if tree == False:
        return 'no'

    # otherwise find the correct subtree
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)  # None if input is missing attribute

    if subtree_key not in subtree_dict: # if no subtree for key,
        subtree_key = None              # we'll use the None subtree

    subtree = subtree_dict[subtree_key] # choose the appropriate subtree
    return classify(subtree, input)     # and use it to classify the input

def build_tree_id3(inputs, split_candidates=None):

    # if this is our first pass,
    # all keys of the first input are split candidates
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # count Trues and Falses in the inputs
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues
    if num_trues == 0:                   # if only Falses are left
        return False                     # return a "False" leaf

    if num_falses == 0:                  # if only Trues are left
        return True                      # return a "True" leaf

    if not split_candidates:             # if no split candidates left
        return num_trues >= num_falses  # return the majority leaf

    # otherwise, split on the best attribute
    best_attribute = min(split_candidates,
        key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # recursively build the subtrees
    subtrees = { attribute : build_tree_id3(subset, new_candidates)
                 for attribute, subset in partitions.items() }

    subtrees[None] = num_trues > num_falses # default case

    return (best_attribute, subtrees)

def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]

def printFormatTree(tree, indent=0):
    if type(tree) == tuple:
        print ('-----' * indent, tree[0])
        printFormatTree(tree[1], indent)
    if type(tree) == dict:
        indent += 1
        for state in tree.keys():
            if type(tree[state])== tuple:
                print ('-----' * indent, state)
                indent += 1
                printFormatTree(tree[state], indent)
                indent -= 1
            elif type(tree[state])== str:
                print ('-----' * indent, state, ":", tree[state])
        indent -= 1

if __name__ == "__main__":

    trainingset = 'tennis_altertemp.txt'
    f = open(trainingset)
    rowcnt = 1
    i = 0
    header = []
    t1 = {}
    inputs = []

    for line in f:
        linelist = line.split()
        linelist.pop(0)   #remove the day column
        totalcols = len(linelist)

        if rowcnt == 1:
            for w in linelist:
                header.append(w)
        else:
            for w in linelist:
                if i == (totalcols - 1):
                    i=0
                    break
                else:
                    t1[header[i]]=w
                    i += 1
            if w == 'yes':
                inputs.append((t1, True))
            if w == 'no':
                inputs.append((t1, False))
        rowcnt += 1
        t1 = {}

    for key in header[:totalcols-1]:
        print (key, partition_entropy_by(inputs, key))
    print

#    senior_inputs = [(input, label)
#                     for input, label in inputs if input["level"] == "Senior"]

#    for key in ['lang', 'tweets', 'phd']:
#        print (key, partition_entropy_by(senior_inputs, key))
#    print

    print ("building the tree")
    tree = build_tree_id3(inputs)
    print (tree)
    printFormatTree(tree, indent=0)
```

```
outlook 0.7799774670388573
temperature 0.9389462162661897
humidity 0.7884504573082896
wind 0.8921589282623617
building the tree
('outlook', {'sunny': ('humidity', {'high': ('wind', {None: False, 'TRUE': False, 'FALSE':
('temperature', {'mild': False, None: False, 'hot': True})}), None: True, 'normal': ('wind',
{None: True, 'TRUE': ('temperature', {'mild': True, None: False}), 'FALSE': True})}), None:
True, 'rainy': ('wind', {None: False, 'TRUE': False, 'FALSE': True}), 'overcast': True})
 outlook
----- sunny
---------- humidity
--------------- high
-------------------- wind
------------------------- FALSE
------------------------------ temperature
--------------- normal
-------------------- wind
------------------------- TRUE
------------------------------ temperature
----- rainy
---------- wind
```

HW4.1.3 Now, build a tree using only examples D1–D7. What is the classification accuracy for the training set? what is the accuracy for the test set (examples D8–D14)? explain why you think these are the results.

In [38]:
```
%%writefile D1D7tennis.txt
Day outlook temperature humidity wind playtennis
d1 sunny hot high FALSE no
d2 sunny hot high TRUE no
d3 overcast hot high FALSE yes
d4 rainy mild high FALSE yes
d5 rainy cool normal FALSE yes
d6 rainy cool normal TRUE no
d6 overcast cool normal TRUE yes
d7 sunny mild high FALSE no
```

```
Writing D1D7tennis.txt
```

In [40]:
```
%%writefile D8D14tennis.txt
Day outlook temperature humidity wind playtennis
d8 sunny cool normal FALSE yes
d9 rainy mild normal FALSE yes
d10 sunny mild normal TRUE yes
d11 overcast mild high TRUE yes
d12 overcast hot normal FALSE yes
d12 rainy mild high TRUE no
```

```
Writing D8D14tennis.txt
```

In [39]:

```
from __future__ import division
from collections import Counter, defaultdict
from functools import partial
import math, random

def entropy(class_probabilities):
    """given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2) for p in class_probabilities if p)

def class_probabilities(labels):
    total_count = len(labels)
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)

def partition_entropy(subsets):
    """find the entropy from this partition of data into subsets"""
    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
                for subset in subsets )

def group_by(items, key_fn):
    """returns a defaultdict(list), where each input item
    is in the list whose key is key_fn(item)"""
    groups = defaultdict(list)
    for item in items:
        key = key_fn(item)
        groups[key].append(item)
    return groups

def partition_by(inputs, attribute):
    """returns a dict of inputs partitioned by the attribute
    each input is a pair (attribute_dict, label)"""
    return group_by(inputs, lambda x: x[0][attribute])
```

```
def partition_entropy_by(inputs,attribute):
    """computes the entropy corresponding to the given partition"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())

def classify(tree, input):
    """classify the input using the given decision tree"""

    # if this is a leaf node, return its value
    #if tree in [True, False]:
    #    return tree

    if tree == True:
        return 'yes'
    if tree == False:
        return 'no'

    # otherwise find the correct subtree
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)  # None if input is missing attribute

    if subtree_key not in subtree_dict: # if no subtree for key,
        subtree_key = None              # we'll use the None subtree

    subtree = subtree_dict[subtree_key] # choose the appropriate subtree
    return classify(subtree, input)     # and use it to classify the input

def build_tree_id3(inputs, split_candidates=None):

    # if this is our first pass,
    # all keys of the first input are split candidates
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # count Trues and Falses in the inputs
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues
    if num_trues == 0:                  # if only Falses are left
        return False                    # return a "False" leaf

    if num_falses == 0:                 # if only Trues are left
        return True                     # return a "True" leaf

    if not split_candidates:            # if no split candidates left
        return num_trues >= num_falses  # return the majority leaf

    # otherwise, split on the best attribute
    best_attribute = min(split_candidates,
        key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # recursively build the subtrees
    subtrees = { attribute : build_tree_id3(subset, new_candidates)
                 for attribute, subset in partitions.items() }

    subtrees[None] = num_trues > num_falses # default case

    return (best_attribute, subtrees)

def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]

def printFormatTree(tree, indent=0):
    if type(tree) == tuple:
        print ('-----' * indent, tree[0])
        printFormatTree(tree[1], indent)
    if type(tree) == dict:
        indent += 1
        for state in tree.keys():
            if type(tree[state])== tuple:
                print ('-----' * indent, state)
                indent += 1
                printFormatTree(tree[state], indent)
                indent -= 1
            elif type(tree[state])== str:
                print ('-----' * indent, state, ":", tree[state])
        indent -= 1

if __name__ == "__main__":

    trainingset = 'D1D7tennis.txt'
    f = open(trainingset)
    rowcnt = 1
    i = 0
    header = []
    t1 = {}
    inputs = []

    for line in f:
        linelist = line.split()
        linelist.pop(0)   #remove the day column
        totalcols = len(linelist)

        if rowcnt == 1:
            for w in linelist:
                header.append(w)
        else:
            for w in linelist:
                if i == (totalcols - 1):
                    i=0
                    break
                else:
                    t1[header[i]]=w
                    i += 1
            if w == 'yes':
                inputs.append((t1, True))
            if w == 'no':
                inputs.append((t1, False))
        rowcnt += 1
        t1 = {}

    for key in header[:totalcols-1]:
        print (key, partition_entropy_by(inputs, key))
    print

#    senior_inputs = [(input, label)
#                     for input, label in inputs if input["level"] == "Senior"]

#    for key in ['lang', 'tweets', 'phd']:
#        print (key, partition_entropy_by(senior_inputs, key))
#    print

    print ("building the tree")
    tree = build_tree_id3(inputs)
    print (tree)
    printFormatTree(tree, indent=0)
```

```
outlook 0.3443609377704336
temperature 0.9387218755408671
humidity 0.9512050593046015
wind 0.9512050593046015
building the tree
('outlook', {'sunny': False, None: False, 'rainy': ('wind', {None: True, 'TRUE': False, 'FAL
SE': True}), 'overcast': True})
 outlook
----- rainy
---------- wind
```

In [41]:

```
testset = 'D1D7tennis.txt'
tst = open(testset)

rowcnt = 1
i = 0
header = []
t1 = {}
inputs = []

for line in tst:
    linelist = line.split()
    totalcols = len(linelist)

    if rowcnt == 1:
        for w in linelist:
            header.append(w)
    else:
        for w in linelist:
            if i == (totalcols - 1):
                i=0
                break
            else:
                t1[header[i]]=w
                i += 1
        inputs.append((t1, w))
#        if w == 'yes':
#            inputs.append((t1, True))
#        if w == 'no':
#            inputs.append((t1, False))
    rowcnt += 1
    t1 = {}

#Run through the testset and find the classification error
correct = 0
possible = len(inputs)
for row in inputs:
    if classify(tree, row[0]) == row[1]:
        correct += 1

if correct > 0:
    score = (correct / possible)*100
else:
    score = 0

print('The classification accuracy was %s' % score + '%')
```

```
The classification accuracy was 100.0%
```

In [42]:

```
testset = 'D8D14tennis.txt'
tst = open(testset)

rowcnt = 1
i = 0
header = []
t1 = {}
inputs = []

for line in tst:
    linelist = line.split()
    totalcols = len(linelist)

    if rowcnt == 1:
        for w in linelist:
            header.append(w)
    else:
        for w in linelist:
            if i == (totalcols - 1):
                i=0
                break
            else:
                t1[header[i]]=w
                i += 1
        inputs.append((t1, w))
#        if w == 'yes':
#            inputs.append((t1, True))
#        if w == 'no':
#            inputs.append((t1, False))
    rowcnt += 1
    t1 = {}

#Run through the testset and find the classification error
correct = 0
possible = len(inputs)
for row in inputs:
    if classify(tree, row[0]) == row[1]:
        correct += 1

if correct > 0:
    score = (correct / possible)*100
else:
    score = 0

print('The classification accuracy was %s' % score + '%')
```

```
The classification accuracy was 66.66666666666666%
```

HW4.1.4 In this case, and others, there are only a few labelled examples available for training (that is, no additional data is available for testing or validation). Suggest a concrete pruning strategy, that can be readily embedded in the algorithm, to avoid over fitting. Explain why you think this strategy should work.

Continuously run through the tree - If branches are not leaves, they should be pruned - if subbranches are leaves, see if they can be merged. Test for a reduction in entropy.

HW4.2 Regression Tree (OPTIONAL Homework)

Back to Table of Contents

Implement a decision tree algorithm for regression for two input continous variables and one categorical input variable on a single core computer using Python.

- Use the IRIS dataset to evaluate your code, where the input variables are: Petal.Length Petal.Width Species and the target or output variable is Sepal.Length.
- Use the same dataset to train and test your implementation.
- Stop expanding nodes once you have less than ten (10) examples (along with the usual stopping criteria).
- Report the mean squared error for your implementation and contrast that with the MSE from scikit-learn's implementation on this dataset (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html (http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html))

HW4.3 Predict survival on the Titanic using Python (Logistic regression, SVMs, Random Forests)

Back to Table of Contents

The sinking of the RMS Titanic is one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the Titanic sank after colliding with an iceberg, killing 1502 out of 2224 passengers and crew. This sensational tragedy shocked the international community and led to better safety regulations for ships.

One of the reasons that the shipwreck led to such loss of life was that there were not enough lifeboats for the passengers and crew. Although there was some element of luck involved in surviving the sinking, some groups of people were more likely to survive than others, such as women, children, and the upper-class.

In this challenge, you need to review (and edit the code) in this notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/kmbgrkhh73931lo/Titanic-EDA-LogisticRegression.ipynb) to do analysis of what sorts of people were likely to survive. In particular, please look at how the tools of machine learning are used to predict which passengers survived the tragedy. Please share any usefule graphs/analysis you come up with via the group email.

For more details see:

- https://www.kaggle.com/c/titanic (https://www.kaggle.com/c/titanic)

HW4.4 Heritage Healthcare Prize (Predict # Days in Hospital next year)

Back to Table of Contents

1. Introduction Back to Table of Contents

The Heritage Health Prize (HHP) was a data science challenge sponsored by The Heritage Provider Network. It took place from April 4, 2011 to April 4, 2013. For information on the winning entries, please see here.

Please see the following notebooks for more background and candidate solutions

- Spark Map-Reduce + MMLlib solution (with optional extensions) See Notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/v52cxipe7yftf97/HeritageHealthPrizeUnitTestNotebook_Spark-Map-Reduce.ipynb)
- Spark SQL + MLLib solution (with optional extensions): Notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/s2wxg6g982oho5m/HeritageHealthPrizeUnitTestNotebook_SQL_FINAL.ipynb)

Please look at section 7 in both notebooks complete any one or more the suggested next steps. E.g.,

- Please complete the EDA extensions using inspiration from the Titanic Notebook from above.
- **Complete Section 3.B: EDA-0. Gather information to see what transformations may need to be done on the data.** Answer questions about each raw DataFrame. In general, is the data in good shape? For example, in each of the Target DataFrames (df_target_Y1, df_target_Y2, df_target_Y3), what values does DaysInHospital take on? Are they all integers? What values does ClaimsTruncated take on? Are they all integers? In the Claims DataFrame (df_claims), how many different ProviderIDs are there? How many different PrimaryConditionGroups are there? What are their values? What values can the CharlesonIndex take on? Are they integers? In the Drug Count DataFrame (df_drug_count), what values can DrugCount take on? Are they all integers? Given this information, what transformations are needed?
- **Complete Section 3.D: EDA-1. Create tables and graphs to display information about the transformed DataFrames.** For inspiration, see the Titanic notebook discussed above. Answer questions about each DataFrame. For example, in each of the Target DataFrames (df_target_Y1, df_target_Y2, df_target_Y3), what is the minimum, maximum, mean, and standard deviation of DaysInHospital? In the Claims DataFrame, group by MemberID and Year and count the number of records. What is the minimum, maximum, mean, and standard deviation of the count? Do the same for the Drug Count and Lab Count DataFrames, etc.

- **Please generate ensemble of DT model using 100 trees with 8 nodes and report the Loss** Try additional models. See possibilities here (e.g. Decision Tree Regressor, Gradient-Boosted Trees Regressor, Random Forest Regressor). See an example here. Tune their hyperparameters. Try different feature selections. Try a two-step model!

```
In [ ]:
```