In [550]:
```python
import pandas as pd
import numpy as np
from pathlib import Path

from math import log2
from collections import OrderedDict, defaultdict
```

In [551]:
```python
dt_path = Path('ass2data.csv')
dt = pd.read_csv(dt_path)


print(dt.columns)
```

```
Index(['Age', 'Income', 'Student', 'Credit_Rating', 'Buys_Computer'], dtype='object')
```

In [552]:
```python
dt
```

Out[552]:

	Age	Income	Student	Credit_Rating	Buys_Computer
0	<=30	High	No	Fair	No
1	<=30	High	No	Excellent	No
2	31.40	High	No	Fair	Yes
3	>40	Medium	No	Fair	Yes
4	>40	Low	Yes	Fair	Yes
5	>40	Low	Yes	Excellent	No
6	31.40	Low	Yes	Excellent	Yes
7	<=30	Medium	No	Fair	No
8	<=30	Low	Yes	Fair	Yes
9	>40	Medium	Yes	Fair	Yes
10	<=30	Medium	Yes	Excellent	Yes
11	31.40	Medium	No	Excellent	Yes
12	31.40	High	Yes	Fair	Yes
13	>40	Medium	No	Excellent	No

Math

In [747]:
```python
def entropy(P=0, N=0, name='Set', attribute=''):
    if P == 0 or N == 0:
        return 0
    C = P + N
    e = (-P/C) * log2(P/C) - (N/C) * log2(N/C)
    print('Entropy of {}:{}'.format(name, attribute))
    print('-{}/({}+{}) * log2({}/({}+{})) - {}/({}+{}) * log2({}/({}+{})) = {}'.format(P, P, N, P, P, N, N, P, N, N,
P, N, e))
    return e

def avg_entropy(P, N, attr_pne):
    sum = 0
    for i in range(0, len(attr_pne)):
        p, n, e = attr_pne[i][0], attr_pne[i][1], attr_pne[i][2]
        sum += ((p+n) / (P+N)) * e
    return sum

def gini_index(vals):
    s = 0
    for v in vals:
        s += (v ** 2)
    return 1 - s
```

Custom types

In [748]:
```python
class Node:
    '''Represents a node in a Decision Tree.'''
    def __init__(self, attr_name='', branches=None):
        self.attr_name = attr_name
        self.branches = branches
```

Algorithms

In [769]:
```python
''' Recursive implementation of Decision Tree building algorithm using Gini Index '''

def gini_rec(dt, root, current, nodes, current_branch):
    labels = 'Buys_Computer'

    if dt.empty:
        print('Skipping empty partition')
        print('------------')

    if root:
        print('------------')
        print('Decision Tree:')
        print_tree(None, root, 1)
        print('*' * 50)
        print('\n')

    # Attribute columns without class labels
    dt_attr = dt.drop([labels], axis=1)
    # Table to store Gini Indeces for each Attribute
    attr_cols = list(dt_attr)
    gini_idxs = pandas.DataFrame(data=[[-1] * len(attr_cols)], columns=attr_cols)

    # For each Attribute, calculate Gini Index
    for key, value in dt_attr.iteritems():
        # Attribute with class labels
        attr = value.to_frame().join(dt[labels].to_frame())
        val_counts = attr.groupby(attr.columns[0]).count()
        total_vals = val_counts.sum()[0]
        attr_gini_index = 0

        # Probability of values of Attribute
        p_vals = val_counts.apply(lambda x: x / total_vals)

        # Probability of 'Buys_Computer' == 'Yes' with values of Attributes.
        p_P_vals = attr[attr['Buys_Computer'] == 'Yes'].groupby(attr.columns[0]).count().divide(val_counts).fillna(0)
        # Probability of 'Buys_Computer' == 'No' with values of Attributes.
        p_N_vals = attr[attr['Buys_Computer'] == 'No'].groupby(attr.columns[0]).count().divide(val_counts).fillna(0)
        # Join 2 tables above
        p_P_N_vals = pd.concat([p_P_vals, p_N_vals], axis=1)
        # Calculate Gini Index value-wise
        p_P_N_vals['gini_idx'] = p_P_N_vals.apply(lambda row: gini_index(row), axis=1)

        # Calculate Attribute Gini Index
        for i, row in pd.concat([p_vals, p_P_N_vals], axis=1).iterrows():
            attr_gini_index += (row[0]*row[3])

        gini_idxs[attr.columns[0]] = attr_gini_index

    if not root:
        print('Entire Set:')
        print(dt)
    else:
        print('Partition:')
        print(dt)

    if gini_idxs.empty:
        print('------------')
        print('No Gini Indeces for this partition')
    else:
        print('------------')
        print('Gini Indexes:')
        print(gini_idxs)

    # Get min Gini Index to find Node
    min_gini = gini_idxs.sum().sort_values(ascending=True).to_dict(OrderedDict)
    if min_gini:
        min_value, min_name = list(min_gini.values())[0], list(min_gini.keys())[0]
        for n in nodes:
            if n.attr_name == min_name:
                if root is not None: # Grow tree by attaching newly selected Node to current.
                    for i, b in enumerate(current.branches):
                        if b[0] == current_branch:
                            if current.branches[i][1] is None:
                                current.branches[i][1] = n
                # Add leaves, if value is of pure class
                for i, b in enumerate(n.branches):
                    dt_attr_val = dt[dt[n.attr_name] == b[0]]
                    if len(dt_attr_val['Buys_Computer'].unique()) == 1:
                        n.branches[i][1] = dt_attr_val['Buys_Computer'].unique()[0]
                # Run algorithm for every branch that is not a leaf
                for i, b in enumerate(n.branches):
                    dt_attr_val = dt[dt[n.attr_name] == b[0]]
                    if not dt_attr_val.drop(n.attr_name, axis=1).empty:
                        gini_rec(dt_attr_val.drop(n.attr_name, axis=1), \
                                 root if root is not None else n, n, nodes, b[0])


''' Recursive implementation of the ID3 algorithm. '''

def id3_rec(dt, root, current, nodes, current_branch):
    labels = 'Buys_Computer'
```

```python
    # Attribute columns
    dt_attr = dt.drop([labels], axis=1)

    # Count positive and negative labels
    num_classes = dt[labels].value_counts()
    P = num_classes['Yes'] if 'Yes' in num_classes else 0
    N = num_classes['No'] if 'No' in num_classes else 0

    if root:
        print('------------')
        print('Decision Tree:')
        print_tree(None, root, 1)
        print('*' * 50)
        print('\n')

    if P == 0 or N == 0:
        print('Skipping partition with 0 entropy:')
        print(dt)
        print('*' * 50)
        print('\n')
        return

    if dt.empty:
        print('Skipping empty partition')
        print('------------')
        return
    elif root:
        print('Partition:')
        print(dt)
    # Entropy of the entire set
    ES = entropy(P, N)
    print('------------')

    # Table for Information Gain for each Attribute, starting with Entropy of the entire set
    attr_columns = list(dt_attr)
    gains = pandas.DataFrame(data=[[ES] * len(attr_columns)], columns=attr_columns)
    avg_infos = []

    # For each Attribute, calculate Entropy for each value in Attribute
    for key, value in dt_attr.iteritems():
        # Attribute with labels
        attr = value.to_frame().join(dt[labels].to_frame())
        entr = pd.DataFrame([], columns=[attr.columns[0], 'p', 'n', 'Entropy'])
        for i, val in enumerate(value.unique()):
            attr_val = pd.DataFrame(attr.loc[attr.iloc[:,0] == val])
            # Number of values with positive label
            p_counts = attr_val[labels].value_counts()['Yes'] if 'Yes' in attr_val[labels].value_counts() else 0
            # Number of values with negative label
            n_counts = attr_val[labels].value_counts()['No'] if 'No' in attr_val[labels].value_counts() else 0
            # Calculate value-wise Entropy
            entr.loc[i] = [val, p_counts, n_counts, entropy(p_counts, n_counts, val, attr.columns[0])]
        # P, N and Entropy for value of Attribute
        pne = [[row['p'], row['n'], row['Entropy']] for i, row in entr.iterrows()]

        # Calculate Average Info Entropy for Attribute
        avg_info_entropy = avg_entropy(P, N, pne)

        print('------------')
        print('Average Information Entropy for {}: {}'.format(attr.columns[0], avg_info_entropy))
        print('------------')
        avg_infos.append([attr.columns[0], avg_info_entropy])

    # Calculate table of Information Gain for each Attribute
    # (Entropy) - (Average Info Entropy) = Information Gain
    for avg in avg_infos:
        attr_name, info = avg[0], avg[1]
        gains[attr_name] = gains[attr_name].apply(lambda x: x - info)

    if gains.empty:
        return
    if gains.max().max() == 0:
        return

    print('------------')
    print('Attributes Information Gain:')
    print(gains)

    # Get max Information Gain Attribute
    max_attr = gains.sum().sort_values(ascending=False).to_dict(OrderedDict)
    if max_attr:
        max_gain, max_name = list(max_attr.values())[0], list(max_attr.keys())[0]
        for n in nodes:
            if n.attr_name == max_name:
                if root is not None: # Grow tree by attaching newly selected Node to current.
                    for i, b in enumerate(current.branches):
                        if b[0] == current_branch:
                            if current.branches[i][1] is None:
                                current.branches[i][1] = n
                # Add leaves, if necessary
                for i, b in enumerate(n.branches):
                    dt_attr_val = dt[dt[n.attr_name] == b[0]]
                    if len(dt_attr_val['Buys_Computer'].unique()) == 1:
                        n.branches[i][1] = dt_attr_val['Buys_Computer'].unique()[0]
```

```python
                # Run algorithm for every branch that is not a leaf
                for i, b in enumerate(n.branches):
                    dt_attr_val = dt[dt[n.attr_name] == b[0]]
                    if not dt_attr_val.drop(n.attr_name, axis=1).empty:
                        id3_rec(dt_attr_val.drop(n.attr_name, axis=1),\
                                root if root is not None else n, n, nodes, b[0])

# Call to choose algorithm and create tree for data
def tree(dt, alg):
    feats = ['Age', 'Income', 'Student', 'Credit_Rating'] # Feature columns
    nodes = [] # Nodes of Decision Tree

    # Initialize unused Nodes
    for f in feats:
        n = Node(f, [[val, None] for val in dt[feats][f].unique()])
        nodes.append(n)

    if alg == 'id3':
        id3_rec(dt, None, None, nodes, None)

    if alg == 'gini':
        gini_rec(dt, None, None, nodes, None)


# Print Decision Tree (roughly)
def print_tree(last_branch, tree, space):
    spaces = ' ' * space
    if last_branch is None:
        print(tree.attr_name)
    else:
        print('{}{}->{}'.format(spaces, last_branch, tree.attr_name))
    for b in tree.branches:
        if type(b[1]) is str:
            spaces = ' ' * space * 4
            print('{}{} -> {}'.format(spaces, b[0], b[1]))
        elif b[1] is None:
            print('{}{} -> ?'.format(spaces, b[0], b[1]))
        else:
            print_tree(b[0], b[1], space * 2)
```