Western Engineering

Study and execution of the decision tree algorithm

- Sathyanath Nandakumar

Decision Tree Algorithm

Decision tree learning serves as one of the most accurate supervised machine learning algorithms. It shows the particular resilience to noise and has multiple variants to suit particular tasks.

The aim is to produce an inverted tree with the end leaf nodes having a solely pure set of values (unmixed).

In general we will use an index to measure the uncertainty at a node and an index to measure how much it has reduced between levels.

The final tree can be pruned to prevent overfitting.

Part 1 : Project Code (Function calls)

- Processing the input data

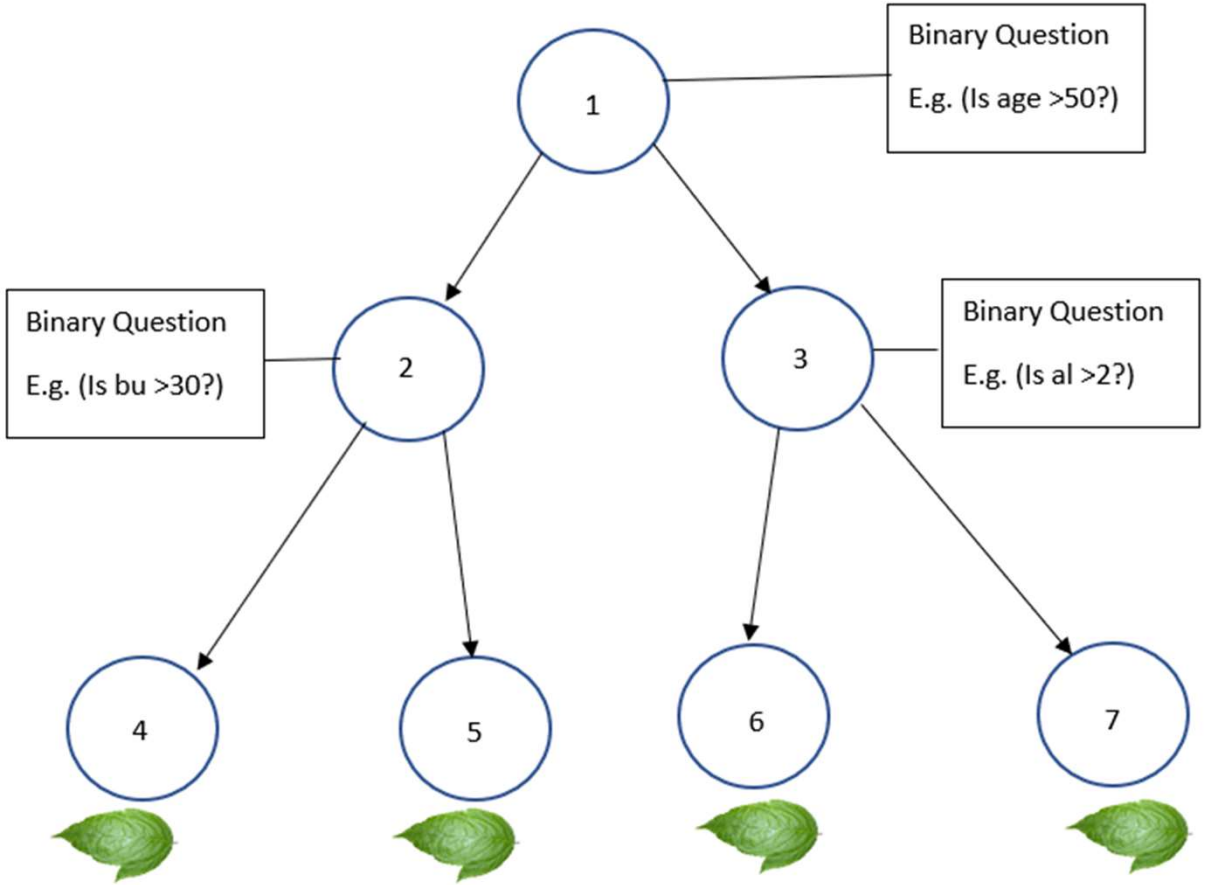
```
def process_split(dt, ts):  # function for processing the values and splitting into training and testing sets
    dtc = list(dt.columns.values)
    label = preprocessing.LabelEncoder()
    for i in dtc:
        dt[str(i)] = label.fit_transform(list(dt[str(i)]))
    xc = dt.values[:, 0:6]
    yc = dt.values[:, 6]
    xt, xts, yt, yts = sklearn.model_selection.train_test_split(xc, yc, test_size=ts)
    return xt, xts, yt, yts
```

- Gini and entropy function call

```
def DT_gini(xg, yg):  # function for criteria as gini
    clf_g = DecisionTreeClassifier(criterion="gini", random_state=100, max_depth=3, min_samples_leaf=5)
    clf_g.fit(xg, yg)
    return clf_g


def DT_entr(xe, ye):  # function for criteria as entropy
    clf_e = DecisionTreeClassifier(criterion="entropy", random_state=100, max_depth=3, min_samples_leaf=5)
    clf_e.fit(xe, ye)
    return clf_e
```

Part 2 : CART Algorithm (Binary Recursive Partitioning) Ground up Build



How to Ask a Question to a Node?

- Asking a question to a node

```
class testingcondition:
    def __init__(self, col, colv):
        self.col = col
        self.colv = colv

    def match(self, example):
        val = example[self.col]
        if val > self.colv:
            return True
        else:
            return False

    def __str__(self):
        return 'Is ' + dtc[self.col] + ' > ' + str(self.colv)
# q = testingcondition(0, 50)
# >>> Is age > 50
# example = tdata[0]
# here we access the first list within the list
# : [38  1 23  8  1  1  0]
# r = q.match(example)
# we check the whether the first value, age satisfies the condition
# print(r)   returns a false value
```

- Splitting data on its basis

```
def split(rw, tq):
    t_row, f_row = [], []
    for r in rw:
        if tq.match(r):
            t_row.append(r)
        else:
            f_row.append(r)
    return t_row, f_row

# We create two empty lists to feed the true values
# and the false values as determined by our condiiton

# here the testing condition is age>50
# true_rows, false_rows = split(tdata, testingcondition(0, 50))
# true_rows = [array([52,  2, 39, 14,  0,  1,  0], dtype=int64),
#              array([58,  0, 40, 70,  0,  0,  0], dtype=int64).
# false_rows = [array([38,  1, 23,  8,  1,  1,  0], dtype=int64)
#              array([4, 4, 5, 4, 0, 0, 0], dtype=int64)....
```

How to Find the Best Question to Ask a Node?

- Gini Index

```
def gini(data):
    impurity = 1
    counts = cls_cnt(data)
    for t in counts:
        prob_val = counts[t] / float(len(data))
        impurity = impurity - prob_val**2
    return impurity

# probability of a value is given by
# its count in the column (accessed within the dictionary)
# divided by the len of the data set
# This value is subtracted from the
# impurity which was originally 1

# b = gini(tdata)
# >>> 0.48628100417780123
# (the gini value for the original data set)
```

```
def cls_cnt(items):
    cnt = dict()
    for c in items:
        out = c[-1]
        cnt[out] = cnt.get(out, 0) + 1
    return cnt
# creates a dictionary where the keys are the unique terms
# in the column and the values
# they relate to are the counts of that term
# get(key,value to return if key doesnt exist)
# method returns the value of the item with the specified key

# {0: 1}
# {0: 2}
# {0: 3}
# ...
# {0: 190, 1: 1}
# {0: 190, 1: 2}
# {0: 190, 1: 3}
# ...
# final result :  {0: 190, 1: 136}
```

Information Gain

```
def info_gain(tbran, fbran, uct):

    p = float(len(tbran)) / (len(tbran) + len(fbran))
    ig = uct - p * gini(tbran) - (1 - p) * gini(fbran)
    return ig

# the info gain is given by the gini index of the starting node
# - the weighted gini index of the true branch
# - the weighted gini index of the false branch

# a = gini(tdata) gives the gini index of the starting dataset
# true_rows, false_rows = split(tdata, testingcondition(0, 50))
# we split the starting data set across the condition age>50
# b = info_gain(true_rows, false_rows, a)
# this is a measure of how much uncertainity has been reduced by that split
# print(b)
# >>> 0.0341668057783549
```

Finding the Best Split across All the Values

```
def find_best_split(rows):

    b_gain = 0  # keep track of the best information gain
    b_ques = None  # keep train of the feature / value that produced it
    crt_uncertainty = gini(rows)
    n_feat = len(rows[0]) - 1  # number of columns

    for col in range(n_feat):                # This inner for loop is to obtain the unique values
        a = []                               # in each column output:
        for r in rows:                       # {0, 1, 2, 3,......69, 70, 71}
            a.append(r[col].item())          # {0, 1, 2, 3, 4, 5}
            values = set(a)                  # {0, 1, 2, 3.......98, 99, 100}
                                             # {0, 1, 2, 3.......69, 70, 71}
                                             # {0, 1}
                                             # {0, 1}

       for val in values:  # we iterate along those values and
                           # assign each as the condition to find best gini index and info gain

            tques = testingcondition(col, val)
            trow, frow = split(rows, tques)   # try splitting the dataset

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

            gain = info_gain(trow, frow, crt_uncertainty)

            if gain >= b_gain:
                b_gain, b_ques = gain, tques

    return b_gain, b_ques
```

Main()

```
kdata = pd.read_csv("Cleankidney.txt")
dtc = list(kdata.columns.values)
label = preprocessing.LabelEncoder()
for i in dtc:
    kdata[str(i)] = label.fit_transform(list(kdata[str(i)]))
tdata = kdata.values[:, 0:7]

# we transform the values into integers/ float using label encoder
# across all the column values and then feed them into tdata

best_g, best_q = find_best_split(tdata)
print(best_g)
print(best_q)

# we print out the best gain and the best question to ask the first node
```

```
# FINAL OUTPUT
# Is sc > 8
# 0.285849157175025 1
```