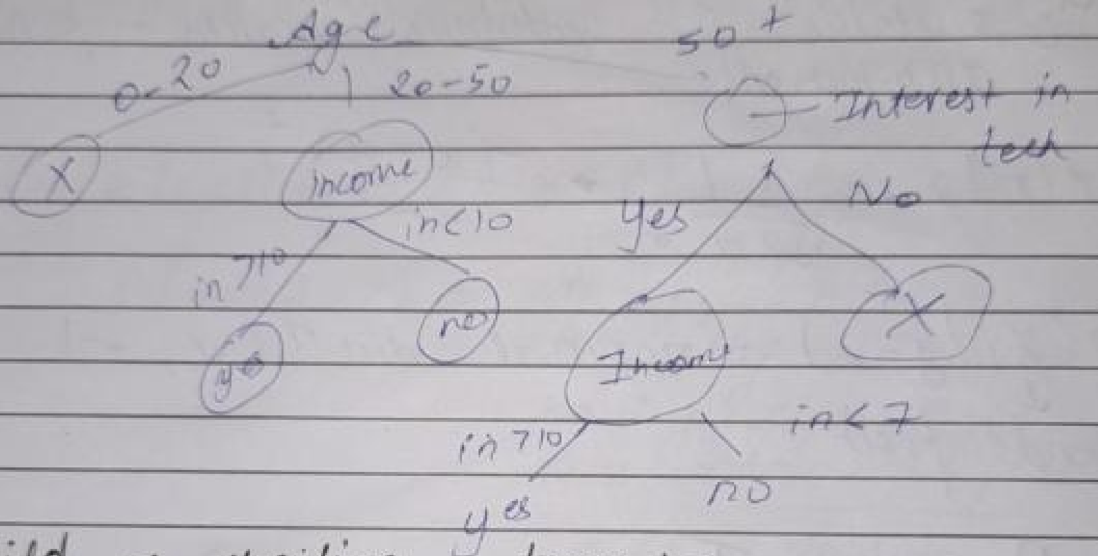
Decision Trees

- simple tree like structure, mode makes a decision at every node, its useful for simple tasks.
- easily explainable, easy to show how the decision process works
- easy to interpret and present
- well defined logic, mimic human thought
- random forests, ensembles of decision trees are more powerful classifiers
- feature value are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.

example :-

Problem :- To predict whether someone will buy a self driving car or not

online	sex	Income	car	tech	age
	M	<5 lac	yes	yes	10-20
	F	5-10 lac	no	no	20-50
		7-10 lac			50+



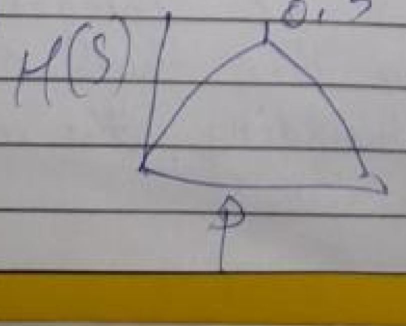
Build a decision tree :-

outlook	temp	humidity	windy	play
sunny	hot	high	false	yes
overcast	mild	normal	true	no
rainy	cold	low	false	yes

entropy - measures the randomness of the system.

randomness $H(S) = -\sum P_c \log P_c$

prob of class C



Date :

3R, 2G, 1W

$$H(S) = -\left(\frac{3}{6}\log\frac{3}{2} + \frac{2}{6}\log\frac{2}{6} + \frac{1}{6}\log\frac{1}{6}\right)$$

calc entropy . classes [Yes, no]
count no. of class members

$$= -\left[\frac{9}{14}\log\frac{9}{14} + \frac{5}{14}\log\frac{5}{14}\right]$$

$= 0.41 + 0.53$

$H(S) = 0.94$

? choose this

our goal is to choose head node
to choose that we will see
which features gives us less entropy
information gain

$$IG(S, A) = H(S) - \sum \frac{|S_v|}{|S|} H(S_v)$$

new entropy

set divided by attribute A

old entropy

ratio of no of ex of new set / all the sets

maximize information gain, reduce entropy of system

IG = windy

f	t
8	6
n	no
n	no
y	~~y~~
y	y
y	y
y	y
y	
y	

IG

$$H(S) = -\sum \frac{S_v}{S} H(S_v)$$

$$= -\sum \frac{8}{14}\left(-\frac{6}{8}\log\frac{6}{8} - \frac{2}{8}\log\frac{2}{8}\right)$$

f wind

$$+ \frac{6}{14}\left[-\frac{3}{6}\log\frac{3}{6} - \frac{3}{6}\log\frac{3}{6}\right]$$

$$= H(S) - \frac{8}{14}(0.81) + \frac{6}{14}$$

$= 0.94 - 0.892$
$= 0.0048$ ← very less infogain

outlook	temp	humidity	windy	play
sunny	hot	high	false	N
"	"	"	T	N
overcast	"	"	F	y
rainy	mild	"	F	y
"	cool	normal	F	y
"	"	"	T	N
overcast	"	"	T	y
sunny	mild	high	F	N
"	cool	normal	F	y
rainy	mild	"	F	y
sunny	"	"	T	y
overcast	"	high	T	y
"	hot	normal	F	y
rainy	mild	high	T	N

$$IG(S, outlook = -\left[\frac{5}{14}\left(\log\frac{5}{14}\right) + \frac{4}{14}\log\frac{4}{14} + \frac{5}{14}\log\left(\frac{5}{14}\right)\right] = HS$$

$$IG(S, outlook = H(S) - \sum \frac{|S_v|}{S}\left(H(S_v)\right)$$

outlook

sunny	overcast	rainy
N	y	y
N	y	N
N	y	y
y	y	y
y		N

$$= -\left[\left(\frac{5}{14}\left(-\frac{3}{5}\log\left(\frac{3}{5}\right) - \frac{2}{5}\log\frac{2}{5}\right) + \frac{4}{14}\left(-\log 1\right) + \frac{5}{14}\left(-\frac{2}{5}\log\frac{2}{5} - \frac{3}{5}\log\frac{3}{5}\right)\right)\right]$$

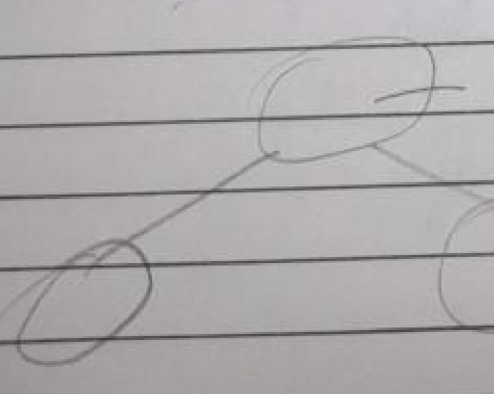
IG(S, out look) = 0.247

```
def entropy (col):
    counts = np.unique(col, return_counts = True)
    N = float( col. shape[0])
    ent = 0.0
    for ix in counts[1]:
        p = ix / N
        ent += -(1.0 * p * np. log2(p))
    return ent

def divide_data(x_data, fkey, fval):
    #create two empty df
    x_right = pd.dataframe([], columns = x_data.
    x_left = pd. dataframe([], columns=x.columns)

    # copy data to those empty df acc to
          condition
    for ix in range(x_data. shape[0]):
        val = x_data [fkey]. loc[ix]
        if val > fval:
            x_right = x_right.append(x_data.loc[ix])
        else:
            x_left= x_left. append(x_data. loc[ix])

    return x_left, x_right
```



fkey = feature

fval = threshold value

Date :

def information_gain (x_data, fkey, fval):
    left, right = divide_data (x_data, fkey, fval)

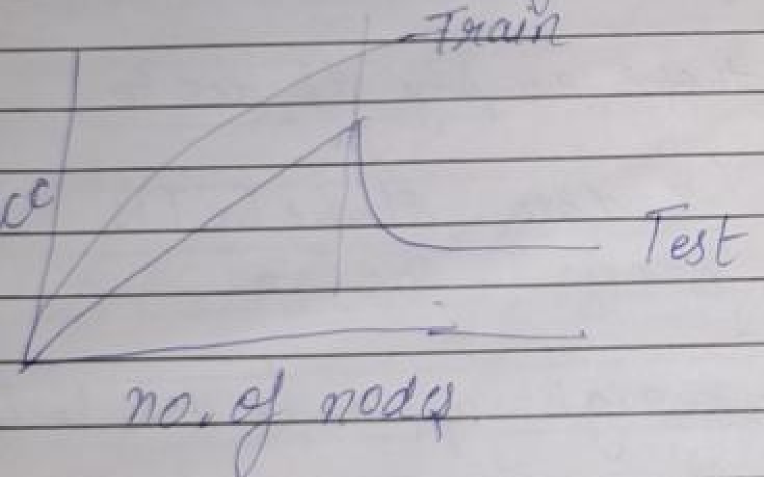
    # % of total samples are on left & right
    l = float (left.shape[0]) / x_data.shape[0]
    r = float (right.shape[0]) / x_data.shape[0]

    # all ex comes to one side
    if left.shape[0] == 0 or right.shape[0] == 0
        return -10000000 #min info gain

    i_gain = entropy (x_data.survived) -
             l * entropy (left.survived)
             + r * entropy (right.survived)

    return i_gain

class DecisionTree:-
    # constructor



Page No.
Date :

```
class Decision_Tree :
    # Constructor
    def _init_ (self, depth = 0, max_depth = 5):
            self. left = None           self. fval = None
            self. right = None          self. max_depth = max_depth
            self. fkey = None           self. depth = depth
                                        self. target = None
    def train ( self, X_train)
        features = ["pclass", "sex", "age", "sibsp", "parch",
                                                      "fare"]
        info_gains = [ ]

    for ix in features:
        i_gain = information_gain (X_train, ix,
                          X_train[ix]. mean ()
        info_gains. append ( i_gain)
        self. fkey = features [np. argmax( info_gain)
        self. fval = X_train [self. fkey]. mean()
        print ("Head node is", self. key)
        # split data
        data_left, data_right = divide_data (X_train,
                          self. fkey, self. fval)
        data_left = data_left. reset_index(drop = True)
X_train. Survived data_right = data_right. reset_index (drop = True)
. mean > 0.5
    # truly a one side node
    if data_left. shape [0] = = 0 or data_right. shape[0]
                                                      = = 0
        self. target = "Survive"
    else
        self. target = "Dead"
```

Date :

```
# Stop early when depth >= max depth
if (self.depth >= self.max_depth)
    if (X_train.Survived.mean() >= 0.5:
        self.target = "Survive"
    else
        self.target = "Dead"
    return

# Recursive Case
self.left = DecisionTree(depth=self.depth+1, max_depth=max_depth)
self.left.train(data_left)
self.right = DecisionTree(depth=self.depth+1,
             max_depth = self.max_depth)
self.right.train(data_right)
# you can set the target at every node
if X_train.Survived.mean() >= 0.5:
    self.target = "Survive"
else
    self.target = "Dead"
return

def predict(self, test):
    if test[self.fkey] > self.fval:
        # go to right
        if self.right is None:
            return self.target
        return self.right.predict(test)
    else:
        if self.left is None:
            return self.target
        return self.left.predict(test)
```