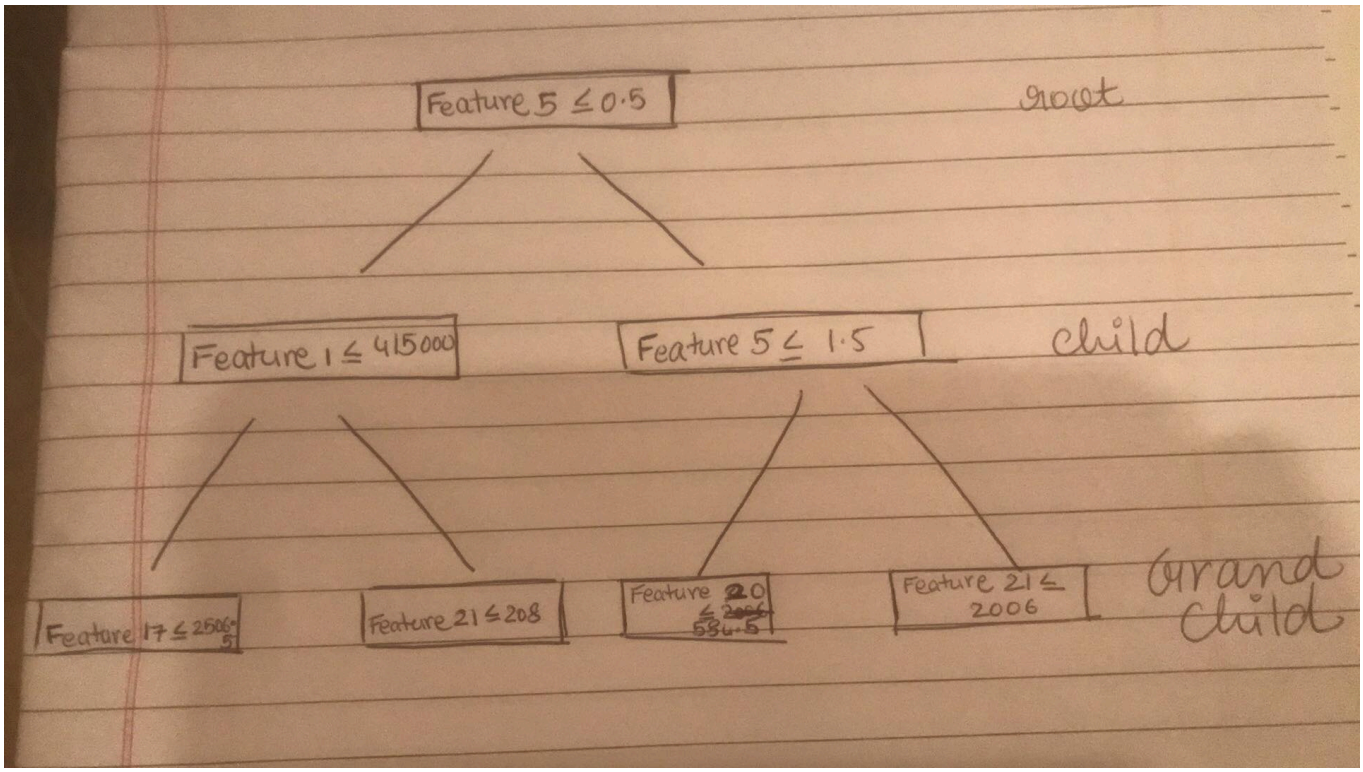
Programming Assignment 2

Hetsvi Navnitlal
CSE 151

Q1. The below picture are the decision rules selected.



Root
Feature 5 <= 0.5
The training points are 2000

Left Child
Feature 1 <= 415000
The training points are 1319

Right Child
Feature 5 <= 1.5
The training points are 681

Left Left Grandchild
Feature 17 <= 2505.6
The training points are 1284

Left Right Grandchild
Feature 21 <= 208
The training points are 35

Right Left Grandchild
Feature 20 <= 584.5
The training points are 292

Right Right Grandchild
Feature 21 <= 2006
The training points are 389

Q2. The training error is 0.0
The test error is 0.498

Q3. Validation error before pruning: 0.461

Pruning
Decision rule is feature 1 <= 415000
Validation error is 0.442
Test error is 0.485

Pruning
Decision rule is feature 4 <= 2.5
Validation error is 0.442
Test error is 0.485

Q4. Feature 5 which is payment delay September

```
import numpy as np
from collections import Counter
import math
import random

text = np.loadtxt('./pa2train.txt')
validate = np.loadtxt('./pa2validation.txt')
other = np.loadtxt('./pa2test.txt')

def total_entropy(values):
   counts = Counter([x for x in values])
   total_length = len(values)
   probability = []
   for i in counts.values():
```

```
        probability.append(i/total_length)
    return [sum([-x*math.log(x) for x in probability]), counts]

def information_gain(values, rule, rule_Value):
    split = []
    sp =[]
    for i in values:
        if i[rule] > rule_Value:
            split.append(i)
        else:
            sp.append(i)
    split_labels = []
    for j in split:
        split_labels.append(j[len(j)-1])

    label = []
    for a in sp:
        label.append(a[len(a)-1])

    proability_split = total_entropy(split_labels)[0]
    probability_label = total_entropy(label)[0]

    probaility_s = len(split_labels)/ len(values)
    probality = len(label)/ len(values)

    conditional = (probaility_s*proability_split) + (probality*probability_label)
    value_ttoal = []
    for b in values:
        value_ttoal.append(b[len(b)-1])

    return [total_entropy(value_ttoal)[0] - conditional, total_entropy(value_ttoal)[1]]

def candidate_rules(values):
    temp = []
    for y in range(22):
        temp_values = []
        for x in values:
            temp_values.append(x[y])
        sort_value = sorted(np.unique(temp_values))
        for i in range(len(sort_value)-1):
            temp.append([(sort_value[i]+sort_value[i+1])/2, y])
    return temp

def id3(values):
```

```
root = Tree()
root.data = values
root.left = Tree()
root.right = Tree()

value_label = []
ig = []
for x in values:
    value_label.append(x[len(x)-1])
counts = Counter([x for x in value_label])
if (len(counts) == 1):
    print(value_label[0])
    root.data = value_label[0]
    root.left = None
    root.right = None
    return root

elif(len(value_label) <= 0):
    return
else:
    cr = candidate_rules(values)
    for j in cr:
        i = information_gain(values, j[1], j[0])
        ig.append([i[0], [j[1], j[0]], i[1]])

max_value = -1
for c in ig:
    if c[0] > max_value:
        max_value = c[0]


max_ig = []
for c in ig:
    if c[0] == max_value:
        max_ig.append(c)

random_ig = random.choice(max_ig)
decison_feature = random_ig[1][0]
decison_value = random_ig[1][1]
keys = []
value_counts = []
for i in random_ig[2].items():
    keys.append(i[0])
    value_counts.append(i[1])
```

```
    count_feature = [keys[0], value_counts[0]]
    count_other = [keys[1], value_counts[1]]


    split_left = []
    split_right = []

    for d in values:
        if d[decison_feature] <= decison_value:
            split_left.append(d)
        else:
            split_right.append(d)

    root.feature = decison_feature
    root.threshold = decison_value
    root.pure = False
    root.count = count_feature
    root.other = count_other
    decison = decison_feature+1
    print("decision rule", decison, "<=", decison_value)
    root.left = id3(split_left)
    root.right = id3(split_right)
    return root

class Tree:
    def __init__(self):
        self.left = None
        self.right = None
        self.data = None
        self.threshold = None
        self.count = None
        self.pure = False
        self.other = None
        self.feature = None

def e(train, other):
    c = 0
    for i in range(len(other)):
        if other[i][-1] != train[i]:
            c = c+1
    return (c/len(other))

def make_predict(tree, values):
    a = []
```

```
    for i in values:
        def make_redict(tree, values):
            if (tree.pure == True):
                if(tree.count[1] > tree.other[1]):
                    a.append(tree.count[0])
                    return
                else:
                    a.append(tree.other[0])
                    return
            else:
                if( tree.left == None and tree.right == None):
                    a.append(tree.data)
                    return
                if values[tree.feature] <= tree.threshold:
                    make_redict(tree.left, values)
                else:
                    make_redict(tree.right, values)
        make_redict(tree, i)
    return a

a = id3(text)
b = make_predict(a, text)
c = e(b, other)
d = e(b, text)
print(c)
print(d)

def tr(tree):
    queue = []
    queue.append(tree)
    counter = 0
    while queue and counter <= 6:
        want = queue.pop(0)
        print(want.feature, want.threshold, want.count, want.other)
        queue.append(want.left)
        queue.append(want.right)
        counter = counter +1
tr(a)

def prune(tree):
    queue = []
    queue.append(tree.left)
    queue.append(tree.right)
    before = e(b, validate)
```

```
    print(before, "b")
    counter = 0
    while queue and counter < 2:
        wanted = queue.pop(0)
        print(wanted.feature, wanted.threshold)
        wanted.pure = True
        counter =counter +1
        after = e(make_predict(a, text), validate)
        t_after = e(make_predict(a, text), other)
        print(after, "a")
        print(t_after)
        if after <= before:
            wanted.left = None
            wanted.right = None
            before = after
        if after > before:
            wanted.pure = False
            counter = counter -1
        if wanted.left != None:
            queue.append(wanted.left)

        if wanted.right != None:
            queue.append(wanted.right)


    return tree

pruned = prune(a)
print(pruned)
```