MACHINE LEARNING
Decision Trees

PROF COWAN CHARLES

RUTGERS UNIVERSITY

Fall 2020

Part - 1

Generate Dataset

For a given value of k, m, (number of features, number of data points), generated a training data set based on the given scheme in part 1.

	1	2	3	4	5	6	7	8	9	10	Y
0	0	0	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	1	0	0	0	0	0
2	0	0	0	0	0	0	0	1	1	1	1
3	1	1	1	0	0	0	1	1	1	1	1
4	1	1	1	1	0	1	1	1	1	1	1
5	0	0	0	0	0	0	1	1	1	1	1
6	1	0	0	0	0	1	1	0	0	1	0
7	0	0	1	1	1	1	1	1	1	1	0
8	1	1	1	1	1	0	0	0	0	0	1
9	1	0	0	0	1	1	0	0	0	0	0
10	1	1	1	0	1	1	0	0	0	0	1
11	1	1	0	1	1	1	1	1	1	1	1
12	0	0	0	0	1	0	1	1	1	1	1
13	1	1	0	1	0	0	0	0	1	1	0
14	0	0	0	1	1	1	1	1	1	1	0
15	1	1	1	1	0	0	1	1	0	1	1
16	0	0	0	0	0	1	0	0	0	0	1
17	1	0	0	0	0	0	1	1	0	0	0
18	0	0	0	0	0	0	0	1	1	1	1
19	1	1	1	1	1	1	1	0	0	0	1
20	1	0	1	1	1	1	1	0	1	1	1
21	1	0	1	0	0	0	0	0	0	0	0
22	1	1	1	0	0	0	0	1	0	0	0
23	0	0	1	1	1	0	0	0	0	0	1
24	0	1	1	1	1	1	0	0	0	0	0
25	1	1	1	0	0	1	1	1	0	0	1
26	1	0	1	1	1	0	1	1	0	1	1
27	0	0	0	0	0	1	1	1	1	1	1
28	0	1	1	1	1	1	1	1	0	0	0
29	1	1	1	1	0	0	0	1	1	1	1

Generating ID3 Tree

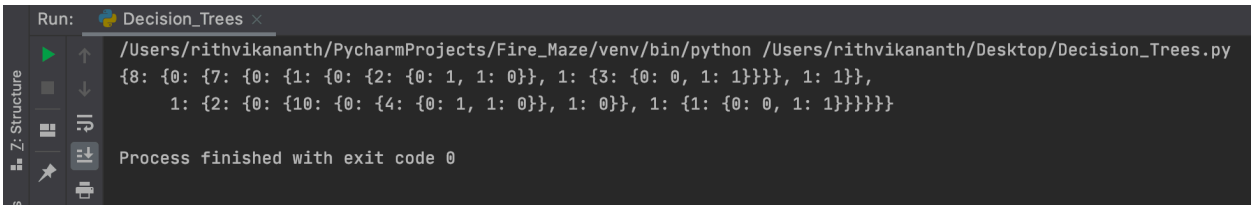
Entropy, conditional entropies calculation functions are written, and decision tree was built using the max information gain of the Xi values.

```
{1: {0: {6: {0: 1, 1: {3: {0: 1, 1: 0}}}},
     1: {4: {0: 0, 1: {2: {0: {7: {0: 0, 1: 1}}, 1: 1}}}}}}
Training error 0.0
```

Training error for the decision tree is 0 which means that the tree fits the dataset.

Finding Training Error and Typical Error for the generated tree

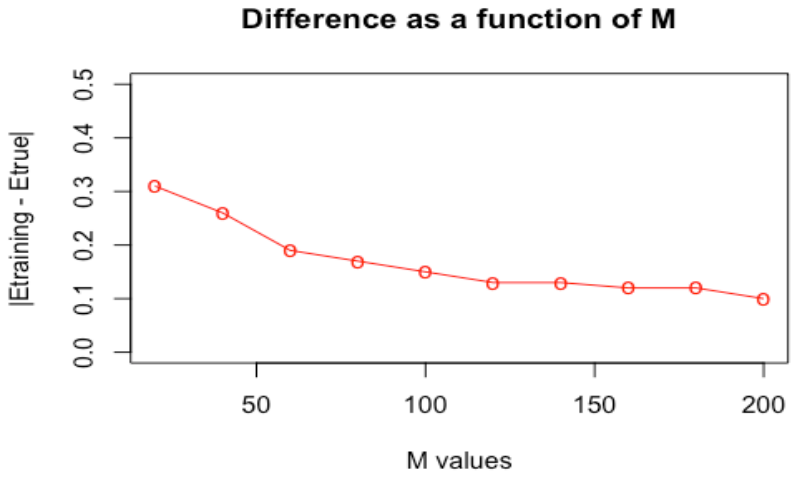
Data for k = 4 and m = 200 was generated and a tree was built.



Answering question 3: Yes, the target variable Y depends on X1 as data is generated as such (dependency of X2, X3. X4, … Xk on X1 by weights). The weights are also calculated such that X2 has the next higher preference than X3, X4 … and X3 has higher preference than X4, X5 … so on. **This dependency of Y on the features is not reflected on the decision tree though.**

Error for 1000 sample data was calculated and the **typical error was around 4%.**

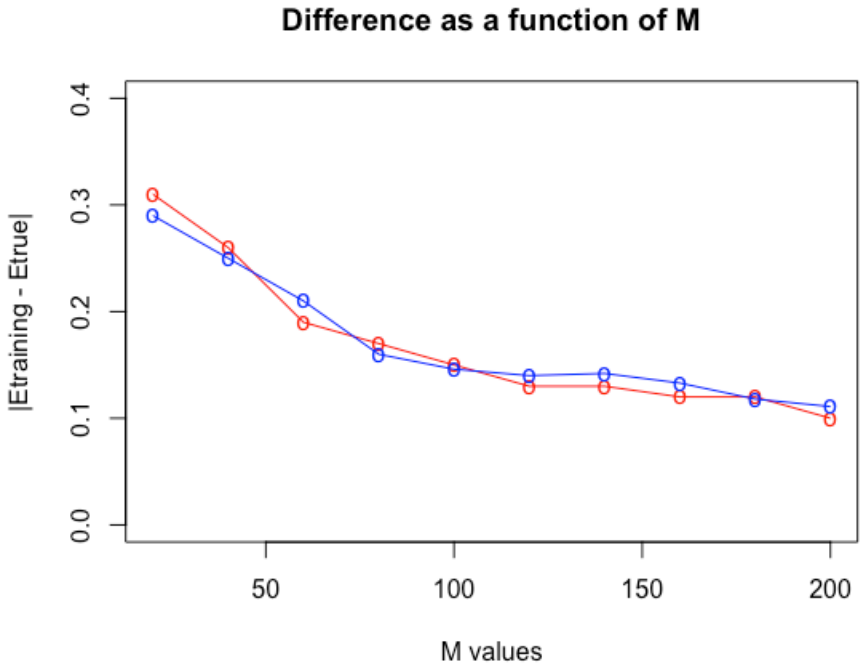
For K = 10, increasing the m values we observe that the absolute difference between the training error and true error decreases.



ID3 – Splitting by Gini Tree

An alternative way of splitting the dataset is by calculating the Gini value for all the columns and split on the column with maximum Gini value.

A comparison between information gain and Gini-index split on | Etraining – Etree | as a function of M:



Blue: Gini-Index, Red: Information Gain

Part – 2

Generating Dataset

For a given value of m (number of data points), generated a training data set based on the given scheme in part 2.

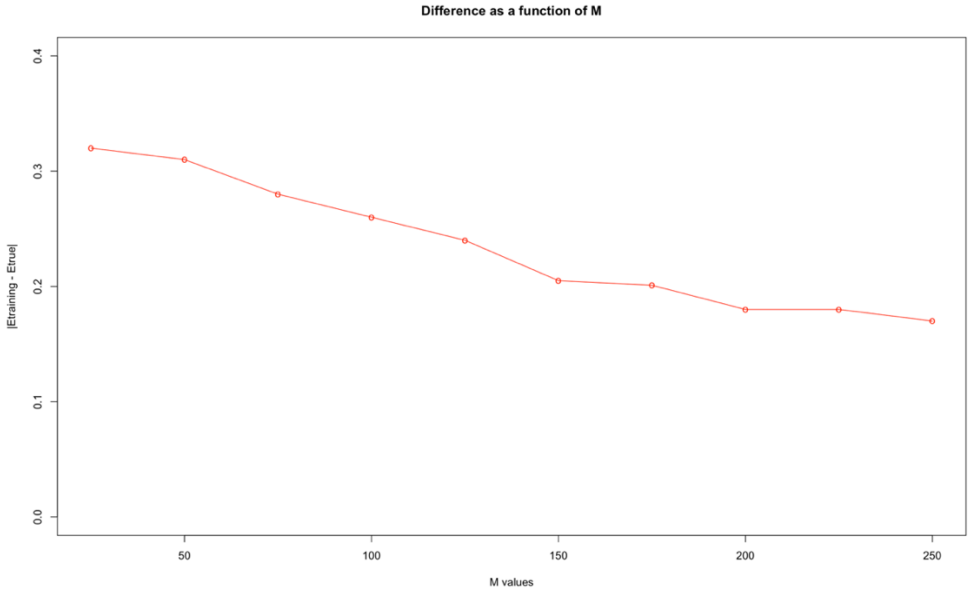
Data was generated according to professors' email later.
(Same idea just shrunk down a little bit)
x0 as a fair coinflip, x1 through x10 each dependent on the previous as in the problem, and x11 through x15 as independent random variables - Y is either the majority of x1 through x5, or the majority of x6 through x10, depending on the value of x0.

(**Note:** generating data as per the given question is also written and commented in the code)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Y
0	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0
1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0
2	0	0	1	0	0	0	0	0	1	1	1	0	0	1	1	0	0
3	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	0	1
4	1	0	0	0	0	0	0	1	1	0	1	1	1	0	1	0	1
5	0	0	1	1	0	1	0	0	0	1	0	1	0	1	1	1	1
6	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	1
7	1	1	1	1	1	0	0	0	0	1	0	0	1	1	0	0	0
8	0	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1
9	0	0	1	1	1	1	1	1	0	0	0	0	1	0	0	1	1
10	0	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	1
11	0	0	1	1	1	1	0	0	0	1	1	0	1	0	1	0	1
12	1	0	0	1	1	0	0	0	0	0	0	0	1	1	0	0	0
13	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1
14	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	0
15	1	1	1	1	1	1	0	1	1	1	1	1	0	1	0	1	1
16	1	1	1	1	1	1	1	1	0	0	0	1	0	1	0	1	0
17	0	1	1	1	0	1	1	0	0	1	1	0	1	0	0	0	1
18	0	1	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1
19	0	0	0	1	1	0	0	1	1	1	0	0	1	0	1	1	1
20	1	1	1	1	0	0	0	0	0	1	1	0	1	0	0	1	0
21	0	1	1	1	0	0	0	0	0	0	1	0	1	1	0	1	1
22	0	0	0	0	0	0	0	0	1	1	1	1	0	1	1	1	0
23	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0
24	0	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	1
25	0	1	0	0	1	1	0	0	0	0	0	0	1	1	0	0	1
26	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	0	1
27	1	1	1	1	1	0	1	0	0	0	0	0	1	0	1	0	0
28	1	1	1	1	1	0	0	0	1	1	1	0	1	1	1	1	1
29	1	1	1	1	0	1	0	0	1	0	0	0	0	1	0	0	0

Plotting |Etraining – Etrue| as a function of M

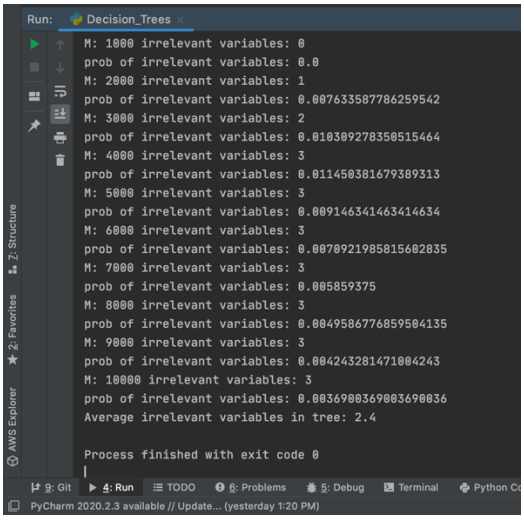
Pruning data was generated and typical error on M data points were plotted. As the m value increases the |Etraining - Etrue| values decrease.

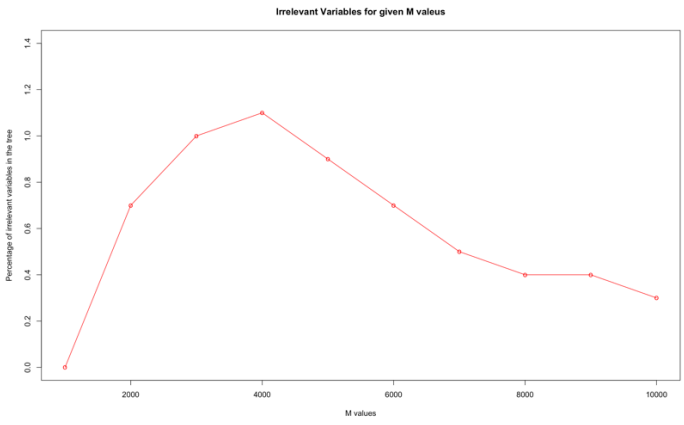


Yes, the above plot agrees with my intuition as there is a marginal decrease in error values with the increase in m values

Irrelevant Variables – M values

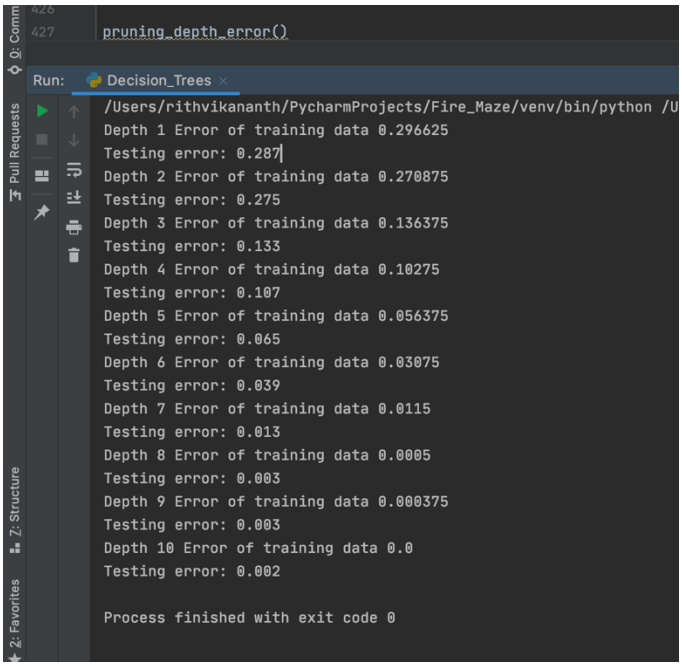
For M values in range 1000, 10000 we see that the probability of finding an irrelevant variable in the tree keeps decreasing.



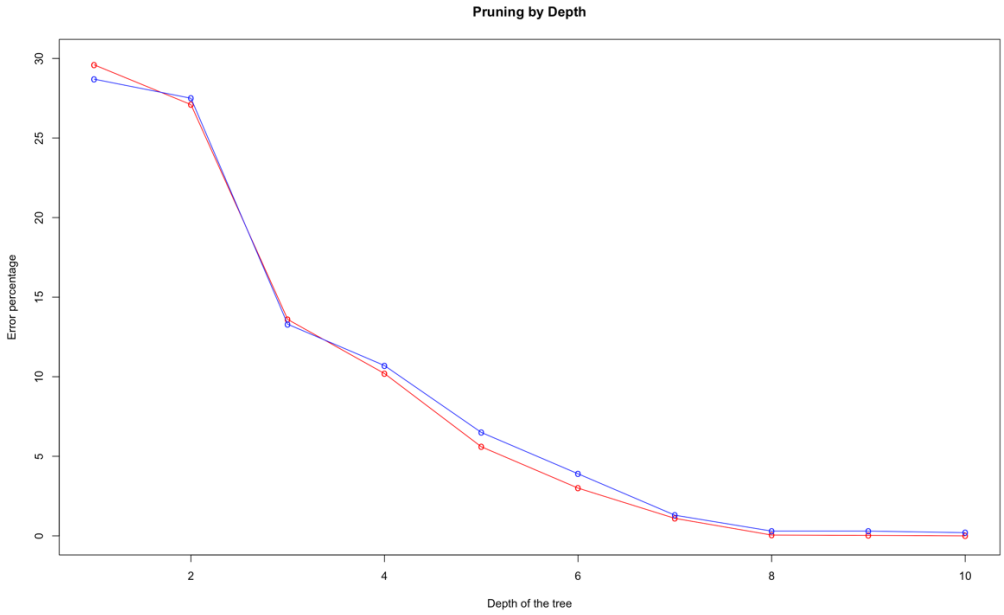


Pruning by Depth – Training and Testing errors

Generating dataset with m = 10000, first 8000 values as training data and remaining 2000 values as testing data.



The tree will be similar to ID3 but after a certain given depth we take the maximum Y value from the data at that point.



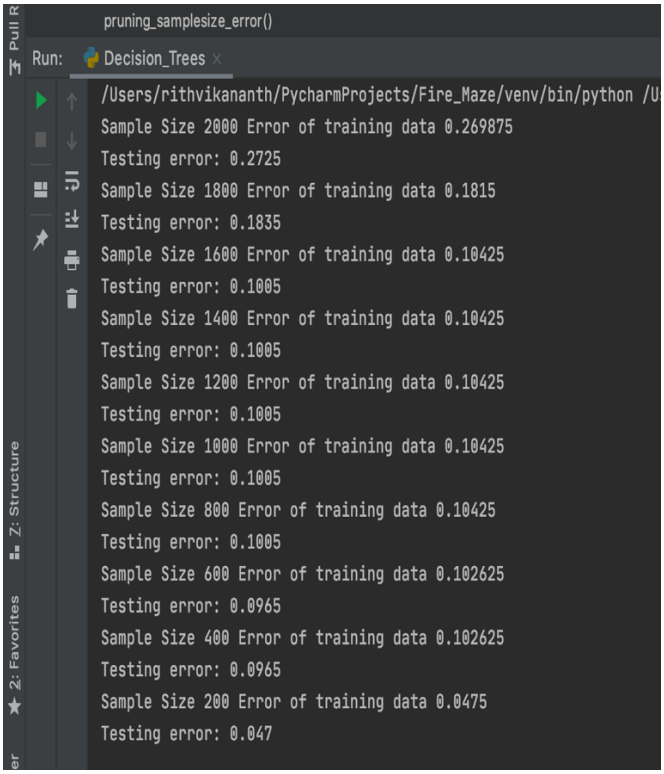
Blue -Testing data, Red – training data.

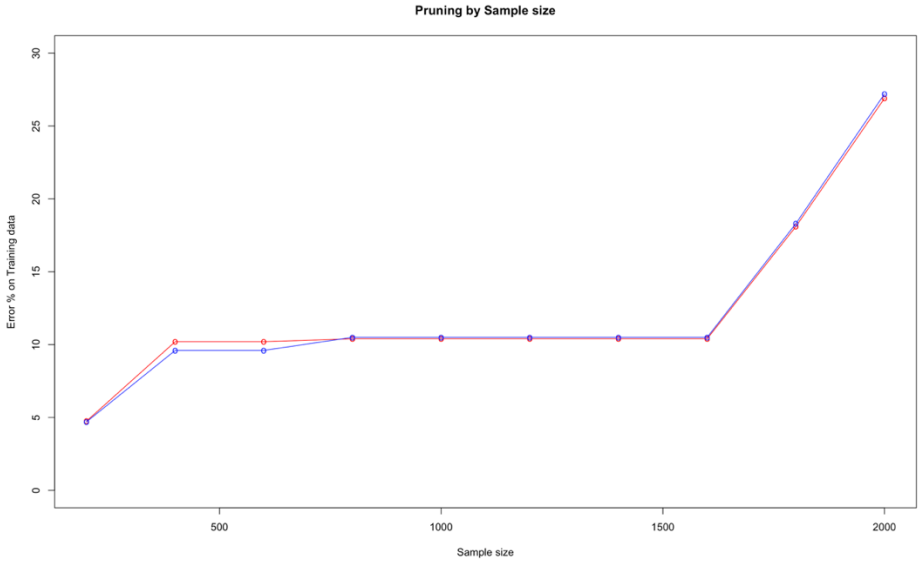
As the depth increases, the error of both training and testing data reduces. At **depth 8** the error on training and testing data is almost 0.

Pruning by Sample Size – Training and Testing errors

The tree will be similar to ID3 but after a sample size shrinks below a given size, we take the maximum Y value from the data at that point.

```
Sample Size 200 Error of training data 0.068625
Testing error: 0.07
Sample Size 190 Error of training data 0.0575
Testing error: 0.0575
Sample Size 180 Error of training data 0.0575
Testing error: 0.0575
Sample Size 170 Error of training data 0.051
Testing error: 0.0545
Sample Size 160 Error of training data 0.051
Testing error: 0.0545
Sample Size 150 Error of training data 0.049875
Testing error: 0.052
Sample Size 140 Error of training data 0.049875
Testing error: 0.052
Sample Size 130 Error of training data 0.049875
Testing error: 0.052
Sample Size 120 Error of training data 0.049875
Testing error: 0.052
Sample Size 110 Error of training data 0.0495
Testing error: 0.0525
Sample Size 100 Error of training data 0.0445
Testing error: 0.0485
Sample Size 90 Error of training data 0.039125
Testing error: 0.046
Sample Size 80 Error of training data 0.0345
Testing error: 0.04
Sample Size 70 Error of training data 0.029625
Testing error: 0.034
Sample Size 60 Error of training data 0.029
```





Blue – Testing data, Red – Training data.

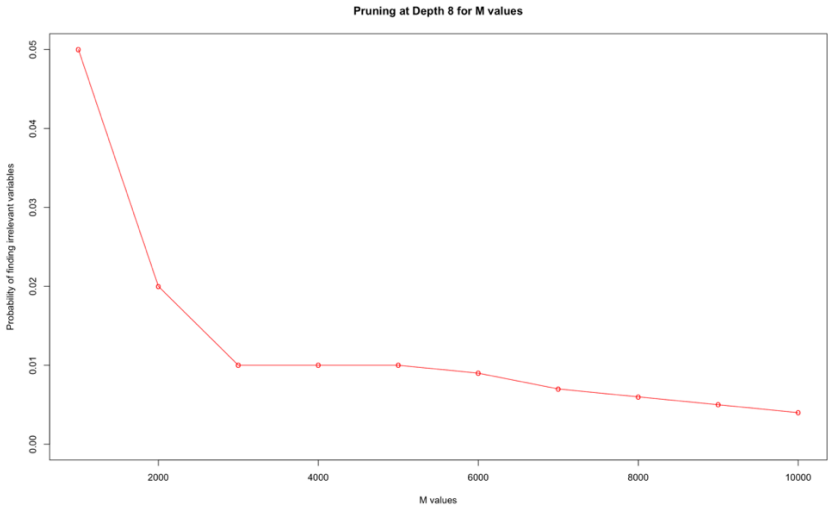
The lower s value we round to get Y, the less error in training data. As we can see in the graph, if the rounding off of the data for Y values increases, the error also increases.

We take **sample size as 100 to be ideal** to avoid irrelevant variables.

Irrelevant Features – Pruning by depth

Now for various m values, probability of finding irrelevant variables at depth 8 is found.

```
Depth 8
M value: 1000
Irrelevant variables 3
prob of irrelevant variables: 0.05263157894736842
M value: 2000
Irrelevant variables 3
prob of irrelevant variables: 0.0234375
M value: 3000
Irrelevant variables 3
prob of irrelevant variables: 0.013888888888888888
M value: 4000
Irrelevant variables 3
prob of irrelevant variables: 0.010638297872340425
M value: 5000
Irrelevant variables 4
prob of irrelevant variables: 0.010666666666666666
M value: 6000
Irrelevant variables 4
prob of irrelevant variables: 0.009070294784580499
M value: 7000
Irrelevant variables 4
prob of irrelevant variables: 0.007366482504604052
M value: 8000
Irrelevant variables 4
prob of irrelevant variables: 0.006309148264984227
M value: 9000
Irrelevant variables 4
prob of irrelevant variables: 0.005449591280653951
M value: 10000
Irrelevant variables 4
prob of irrelevant variables: 0.00477326968973747
Average irrelevant variables in tree: 3.6
```

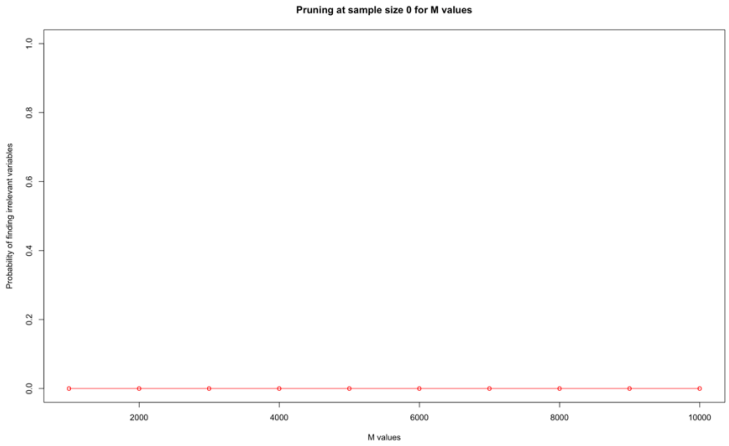


At Depth 8 the probability of finding irrelevant variables is less than 0.05 and for M values greater than 7000 the probability of finding irrelevant variables in the tree is almost 0.

Irrelevant Features – Pruning by Sample Size

As mentioned in 2.3b we take sample size as 100.

```
100
M value: 1000
prob of irrelevant variables: 0.0
M value: 2000
prob of irrelevant variables: 0.0
M value: 3000
prob of irrelevant variables: 0.0
M value: 4000
prob of irrelevant variables: 0.0
M value: 5000
prob of irrelevant variables: 0.0
M value: 6000
prob of irrelevant variables: 0.0
M value: 7000
prob of irrelevant variables: 0.0
M value: 8000
prob of irrelevant variables: 0.0
M value: 9000
prob of irrelevant variables: 0.0
M value: 10000
```



For different m values the probability of finding irrelevant variables in the tree is always 0 if sample size is 100.

Source Code:

```
import random
import numpy as np
from pprint import pprint
import pandas as pd

def generate_td(k, m):
    final = {}
    dataset = []
    header = []
    for i in range(1, k+1):
        header.append(i)
    header.append("Y")
    dataset.append(header)
    w_denominator = 0
    #finding the denominator of the weights
    for d in range(2, k + 1):
        w_denominator += pow(0.9, d)
    #generating the X,Y values as per the mentioned formulas
    for m in range(1, m+1):
        data = []
        x1 = int(random.randint(0, 1))
        #print(x1)
        data.append(x1)
        previous_value = x1
        features_weight = 0
        for i in range(2, k+1):
            a = int(np.random.binomial(1, 0.75, 1))
            #print(a)
            if a == 1:
                d1 = previous_value
            else:
                d1 = 1 - previous_value
            data.append(d1)
            previous_value = d1
        #print(data)
            features_weight += (pow(0.9, i)/w_denominator)*d1
        if features_weight >= 0.5:
            Y = x1
        else:
            Y = 1 - x1
        data.append(Y)
        dataset.append(data)
    columns = dataset[0]
    final_data = pd.DataFrame(dataset[1:], columns=columns)
    return final_data



def entropy(data):
    target = data.columns[-1]
    y_values = data.iloc[:, -1:]
    prob1 = len(y_values[y_values[target] == 1])/len(y_values)
```

```
    prob0 = len(y_values[y_values[target] == 0])/len(y_values)
    if prob0 == 0 or prob1 == 0:
        return 0
    entropy = -prob1*np.log2(prob1) - prob0*np.log2(prob0)
    return entropy

def conditional_entropy(data, c):
    # c is the given column
    entropy_conditional = 0
    y_data = data.iloc[:, -1:]
    target = data.columns[-1]
    Y_values = [0, 1]           # y_data[target].unique()
    X_values = [0, 1]              # data[c].unique()
    for i in X_values:
        e_con_for_x = 0
        len_xcon = len(data[c][data[c] == i])
        #length of the dataset where Xi is either 0 or 1,
        #conditional entropy becomes 0 if there are no values for the x
        if len_xcon == 0:
            continue
        for j in Y_values:
            len_ycon = len(data[c][(data[c] == i) & (data[target] == j)])
            #print(len_ycon)
            if len_ycon == 0:
                continue
            p = len_ycon/(len_xcon)
            e_con_for_x += -p*np.log2(p)
        entropy_conditional += (len_xcon/len(data))*e_con_for_x
    return entropy_conditional

def decision_tree(data, features):
    info_gains = []
    tree = 0
    count0 = 0
    count1 = 0
    target = data.columns[-1]
    y_values = data[target].values.tolist()
    # Termination conditions
    if len(data) == 0:
        return 0
    for i in y_values:
        if i == 0:
            count0 += 1
        else:
            count1 += 1
    if len(features) == 0:
        if count1 >= count0:
            return 1
        else:
            return 0
    if len(np.unique(y_values)) <= 1:
        return y_values[0]
    # finding the maximum information gain column from the data
    for col in data.columns[:-1]:
        info_gains.append(entropy(data) - conditional_entropy(data, col))
    # print(info_gains)
    max_gain = data.columns[info_gains.index(max(info_gains))]
```

```
    # print(max_gain)
    if tree == 0:
        tree = {}
        tree[max_gain] = {}
    features = data.columns[:-1]
    features = [i for i in features if i != max_gain]
    # split on the max info gain node where its value is 0 and 1
    subtree = decision_tree(data[data[max_gain] == 0], features)
    tree[max_gain][0] = subtree
    subtree = decision_tree(data[data[max_gain] == 1], features)
    tree[max_gain][1] = subtree
    return tree

# training_data = generate_td(10, 30)
# pprint(training_data)
# features = training_data.columns[:-1]
# tree = decision_tree(training_data, features)
# pprint(tree)


def tree_value(instance_data, tree):
    for node in tree.keys():
        value = instance_data[node]
        tree = tree[node][value]
        t_value = 0
        if type(tree) is dict:
            t_value = tree_value(instance_data, tree)
        else:
            t_value = tree
            break
    return t_value

def error(data, tree):
    error = 0
    td_error_calculation = data.iloc[:, :-1]
    for i in range(len(data)):
        decisiontree_ans = tree_value(td_error_calculation.iloc[i], tree)
        actual_value = data.iloc[i]['Y']
        if decisiontree_ans != actual_value:
            error += 1
    return error / len(data)

def typical_error(k, m, tree):
    te = 0
    for i in range(0,200):
        generate_data = generate_td(k,m)
        te += error(generate_data,tree)
    te = te/200
    return te

def error_estimate():
    es_error = 0
    m = 20
    for mvalue in range(0, 10):
        print("M value", m)
        for i in range(0,10):
            data = generate_td(10, m)       # k = 10
```

```
            features = data.columns[:-1]
            tree = decision_tree_gini(data, features)            #tree =
decision_tree(data, features)
            es_error += error(data, tree) - typical_error(10, m, tree)
        es_error = es_error/10
        print("Difference of training and true", es_error)
        m += 20

# print("Training error",error(training_data, tree))
# print("typical error:", typical_error(4, 30, tree))
# error_estimate()

def gini_index(data, target):
    gini_value = 1 - (len(data[data[target] == 1]) / len(data)) ** 2 - (
            len(data[data[target] == 0]) / len(data)) ** 2
    return gini_value


def decision_tree_gini(data, features):
    tree = 0
    count0 = 0
    count1 = 0
    target = data.columns[-1]
    y_values = data[target].values.tolist()
    # Termination conditions
    if len(data) == 0:
        return 0
    for i in y_values:
        if i == 0:
            count0 += 1
        else:
            count1 += 1
    if len(features) == 0:
        if count1 >= count0:
            return 1
        else:
            return 0
    if len(np.unique(y_values)) <= 1:
        return y_values[0]
    # finding the maximum gini value of the columns
    gini_values = []

    for col in data.columns[:-1]:
        gini_values.append(gini_index(data, col))

    gini = data.columns[gini_values.index(max(gini_values))]
    #print(max_gain)
    if tree == 0:
        tree = {}
        tree[gini] = {}
    features = data.columns[:-1]
    features = [i for i in features if i != gini]
    # split on the max info gain node where its value is 0 and 1
    subtree = decision_tree(data[data[gini] == 0], features)
    tree[gini][0] = subtree
    subtree = decision_tree(data[data[gini] == 1], features)
    tree[gini][1] = subtree
```

```
    return tree


# Part-2 of the question

def pruning_data (m):
    # shrinking it and generating Y values as per professors email
    k = 15          # k = 20
    dataset = []
    header = []
    for i in range(0, k+1):
        header.append(i)
    header.append("Y")
    dataset.append(header)
    #generating the X,Y values as per the mentioned formulas
    for m in range(1, m+1):
        data = []
        x0 = int(random.randint(0, 1))
        data.append(x0)
        previous_value = x0
        for i in range(1, k+1):
            k = i
            if k < 11:                                      # k<15
                a = int(np.random.binomial(1, 0.75, 1))
                if a == 1:
                    d1 = previous_value
                else:
                    d1 = 1 - previous_value
            elif k >= 11:                                   # k>=15
                d1 = int(np.random.binomial(1, 0.5, 1))
            data.append(d1)
            previous_value = d1
        count0 = 0
        count1 = 0
        if x0 == 0:
            for i in range(1, 5):           #range(1, 8)
                if data[i] == 0:
                    count0 += 1
                else:
                    count1 += 1
            if count0 > count1:
                y = 0
            else:
                y = 1
        else:
            for i in range(6, 11):           #range(8,15)
                if data[i] == 0:
                    count0 += 1
                else:
                    count1 += 1
            if count0 > count1:
                y = 0
            else:
                y = 1
        data.append(y)
        dataset.append(data)
    columns = dataset[0]
```

```
    final_data = pd.DataFrame(dataset[1:], columns=columns)
    return final_data

# training_data = pruning_data(30)
# pprint(training_data)
# features = training_data.columns[:-1]
# tree = decision_tree(training_data, features)
# pprint(tree)
# test_data = pruning_data(30)
# print(error(test_data, tree))
# print("typical error:", typical_error(4, 30, tree))
# error_estimate()

def typical_error_pd(m, tree):
    te = 0
    for i in range(0,200):
        generate_data = pruning_data(m)
        te += error(generate_data, tree)
    te = te/200
    return te


def error_estimate_pd():
    es_error = 0
    m = 200
    for mvalue in range(0, 10):
        print("M value", m)
        for i in range(0,10):
            data = pruning_data(m)
            features = data.columns[:-1]
            tree = decision_tree(data, features)
            es_error += error(data, tree) - typical_error_pd(m, tree)
        es_error = es_error/10
        print("Difference of training and true", es_error)
        m += 200

error_estimate_pd()


def tree_variables(data, tree, a = []):
    for node in tree.keys():
        a.append(node)
        value0 = tree[node][0]
        #print("val0", value0)
        if type(value0) is dict:
            tree_variables(data, value0)
        value1 = tree[node][1]
        #print("val1", value1)
        if type(value1) is dict:
            tree_variables(data, value1)
        else:
            break
    return a

def irrilavent_variables():
    m = 1000
    iv_count = 0
```

```
    for mvalue in range(0, 10):
        data = pruning_data(m)
        # print(data)
        features = data.columns[:-1]
        tree = decision_tree(data, features)
        tree_variable = tree_variables(data, tree)
        # print(tree_variable)
        b = []
        for variables in tree_variable:
            if variables > 10:
                b.append(variables)
        iv_count += len(b)
        # print("M:", m, "irrelevant variables:", len(b))
        # print("prob of irrelevant variables:", (len(b)) /
(len(tree_variable)))
        m += 1000
    return print("Average irrelevant variables in tree:", iv_count/10)

# irrilavent_variables()
# pprint(testing_dataset)
# pprint(len(testing_dataset.columns))

def decision_tree_depth(data, features, depth, d):
    info_gains = []
    tree = 0
    count0 = 0
    count1 = 0
    target = data.columns[-1]
    y_values = data[target].values.tolist()
    # Termination conditions
    if len(data) == 0:
        return 0
    for i in y_values:
        if i == 0:
            count0 += 1
        else:
            count1 += 1
    if len(features) == 0:
        if count1 >= count0:
            return 1
        else:
            return 0
    if len(np.unique(y_values)) <= 1:
        return y_values[0]

    # finding the maximum information gain column from the data
    for col in data.columns[:-1]:
        info_gains.append(entropy(data) - conditional_entropy(data, col))
    # print(info_gains)
    max_gain = data.columns[info_gains.index(max(info_gains))]
    # print(max_gain)
    if tree == 0:
        tree = {}
        tree[max_gain] = {}
    features = data.columns[:-1]
    features = [i for i in features if i != max_gain]
    # split on the max info gain node where its value is 0 and 1
```

```
    if d < depth:
        d += 1
        subtree = decision_tree_depth(data[data[max_gain] == 0], features,
depth, d)
        tree[max_gain][0] = subtree
        subtree = decision_tree_depth(data[data[max_gain] == 1], features,
depth, d)
        tree[max_gain][1] = subtree
        # print(tree)
    else:
        if count1 >= count0:
            tree[max_gain][0] = 1
            tree[max_gain][1] = 1
        else:
            tree[max_gain][0] = 0
            tree[max_gain][1] = 0
    return tree


def pruning_depth_error():
    data = pruning_data(10000)
    training_data = data.iloc[:8000, :]
    testing_data = data.iloc[8000:, :]
    depth = 1
    features = training_data.columns[:-1]
    for i in range(0,10):
        tree = decision_tree_depth(training_data, features, depth, 0)
        print("Depth",depth, "Error of training data", error(training_data,
tree))
        print("Testing error:", error(testing_data, tree))
        depth += 1

# pruning_depth_error()


def decision_tree_sample_size(data, features, sample_size):
    info_gains = []
    tree = 0
    count0 = 0
    count1 = 0
    target = data.columns[-1]
    y_values = data[target].values.tolist()
    # Termination conditions
    if len(data) == 0:
        return 0
    for i in y_values:
        if i == 0:
            count0 += 1
        else:
            count1 += 1
    if len(features) == 0:
        if count1 >= count0:
            return 1
        else:
            return 0
    if len(np.unique(y_values)) <= 1:
        return y_values[0]
```

```
    # finding the maximum information gain column from the data
    for col in data.columns[:-1]:
        info_gains.append(entropy(data) - conditional_entropy(data, col))
    # print(info_gains)
    max_gain = data.columns[info_gains.index(max(info_gains))]
    # print(max_gain)
    if tree == 0:
        tree = {}
        tree[max_gain] = {}
    features = data.columns[:-1]
    features = [i for i in features if i != max_gain]
    size = len(data)
    # split on the max info gain node where its value is 0 and 1
    if size >= sample_size:
        subtree = decision_tree_sample_size(data[data[max_gain] == 0],
features, sample_size)
        tree[max_gain][0] = subtree
        subtree = decision_tree_sample_size(data[data[max_gain] == 1],
features, sample_size)
        tree[max_gain][1] = subtree
        # print(tree)
    else:
        if count1 >= count0:
            tree[max_gain][0] = 1
            tree[max_gain][1] = 1
        else:
            tree[max_gain][0] = 0
            tree[max_gain][1] = 0
    return tree

# training_dataset = pruning_data(8000)
# features = training_dataset.columns[:-1]
# tree = decision_tree_sample_size(training_dataset, features, 2000)
# pprint(tree)

def pruning_samplesize_error():
    data = pruning_data(10000)
    training_data = data.iloc[:8000, :]
    testing_data = data.iloc[8000:, :]
    sample_size = 200
    features = training_data.columns[:-1]
    for i in range(0,15):
        tree = decision_tree_sample_size(training_data, features,
sample_size)
        print("Sample Size",sample_size, "Error of training data",
error(training_data, tree))
        print("Testing error:", error(testing_data, tree))
        sample_size -= 10

# pruning_samplesize_error()


def irrilavent_variables_depth(depth):
    m = 1000
    iv_count = 0
    print("Depth", depth)
```

```
    for mvalue in range(0, 10):
        data = pruning_data(m)
        features = data.columns[:-1]
        tree = decision_tree_depth(data, features, depth, 0)
        tree_variable = tree_variables(data, tree)
        b = []
        for variables in tree_variable:
            if variables > 10:
                b.append(variables)
        iv_count += len(b)
        print("M value:", m)
        print("Irrelevant variables", len(b))
        print("prob of irrelevant variables:", len(b)/ len(tree_variable))
        m += 1000

    return print("Average irrelevant variables in tree:", iv_count/10)

# irrilavent_variables_depth(8)   #depth = 8 from 2.3a

def irrilavent_variables_samplesize(sample_size):
    m = 1000
    print(sample_size)
    for i in range(0, 10):
        data = pruning_data(m)
        features = data.columns[:-1]
        tree = decision_tree_sample_size(data, features, sample_size)
        tree_variable = tree_variables(data, tree)
        # print(tree_variable)
        b = []
        for variables in tree_variable:
            if variables > 10:
                b.append(variables)
        print("M value:", m)
        print("prob of irrelevant variables:", len(b) / len(tree_variable))
        m += 1000


# sample_size = 8000
# irrilavent_variables_samplesize(100)
```

R-code for plotting:

```
plot(mvalue, difference_IG, type="o", col="red", pch="o", lty=1, ylim=c(0,0.4),
     ylab="|Etraining - Etrue|", xlab = "M values") +
  title("Difference as a function of M") +
  points(mvalue, difference_GI, type="o", col="blue", pch="o", lty=1) +
  legend(1, 95, legend=c("Information gain", "Gini Index"),
         col=c("red", "blue"), lty=1:2, cex=0.8)

plot(m, training_error, type="o", col="red", pch="o", lty=1, ylim=c(0,30),
     ylab="Error % on Training data", xlab = "Sample size") +
  title("Pruning by Sample size") +
  points(m, testing_error, type="o", col="blue", pch="o", lty=1)

plot(m, prob, type="o", col="red", pch="o", lty=1, ylim=c(0,1),
     ylab="Probability of finding irrelevant variables", xlab = "M values") +
  title("Pruning at sample size 0 for M values")

plot(m, error, type="o", col="red", pch="o", lty=1, ylim=c(0,30),
     ylab="Error % on Training data", xlab = "Sample Size (s)") +
  title("Pruning by sample size")

plot(depth, error, type="o", col="red", pch="o", lty=1, ylim=c(0,30),
     ylab="Error percentage", xlab = "Depth of the tree") +
  title("Pruning by Depth") + points(depth, error2, type="o", col="blue", pch="o", lty=1)
```

```
plot(mvalue, difference_IG, type="o", col="red", pch="o", lty=1, ylim=c(0,0.4),
     ylab="|Etraining - Etrue|", xlab = "M values") +
  title("Difference as a function of M")
```