1. Generating Decision Trees

Answer 1.1

Importing libs and declaring global variables:

```
import numpy as np
import secrets
import pandas as pd
import matplotlib.pyplot as plt
from timeit import default_timer as timer
%matplotlib inline
```

```
EPS = np.finfo(float).eps
```

Random number generator function to calculate probability:

```
# Generate secure random (not pseudo random) value between 0 and 1
def RNG_Prob():
    secGen=secrets.SystemRandom()    # secret generator
    n=secGen.random()
    return n
```

```
# generate m datapoints with k features each
def gen_X(k,m):
    pass

    # init. required 2D array i.e. our data points with features as empty
    X=np.empty((m,k)).astype(int)

    for i in range(m):
        # setting first feature
        if(RNG_Prob()<0.5):
            X[i][0]=1
        else:
            X[i][0]=0

        # setting subsequent features
        for j in range(1,k):
            if(RNG_Prob()<0.75):
                X[i][j]=X[i][j-1]
            else:
                X[i][j]=1-X[i][j-1]
    return X


# generate weights corresponding to the features
def gen_w(k):
    pass

    # init. weight vector as empty 1D array
    w=np.empty((k))

    # calculate the denominator separately
    deno=0
    for i in range(2,k+1):
        deno+=pow(0.9,i)
```

```
    # setting weight values as per the given formula
    for i in range(1,k+1):
        w[i-1]=(pow(0.9,i))/deno
    return w


# generate the output column values from X and w
def gen_Y(k,m,X,w):
    pass

    # init. Y with as empty 1D array; each Y corresponds to each data point
    Y=np.empty((m)).astype(int)

    # setting up Y values as per the given function
    for i in range(m):
        val=0
        for j in range(1,k):
            val+=X[i][j]*w[j]
        if(val>=0.5):
            Y[i]=X[i][0]
        else:
            Y[i]=1-X[i][0]
    return Y
```

```
# generating the complete dataset having Xs and Y
def gen_dataset(k,m):
    pass
    X=gen_X(k,m)
    w=gen_w(k)
    Y=gen_Y(k,m,X,w)

    # row index for X and Y
    rows=[]
    for i in range(m):
        rows.append("DataPoint "+str(i+1))

    # column header for X
    X_header=[]
    for i in range(k):
        X_header.append("X"+str(i+1))

    # creating corresponding DataFrames for X and Y
    X_df=pd.DataFrame(data=X, index=rows, columns=X_header)
    Y_df=pd.DataFrame(data=Y, index=rows, columns=["Y"])

    # merging the DataFrames
    dataset=X_df.merge(Y_df,left_index=True, right_index=True)
    return dataset
```

```
# init and test with given k and m values
k=4
m=30
dataset=gen_dataset(k,m)
dataset
```

	X1	X2	X3	X4	Y
DataPoint 1	1	1	0	1	1
DataPoint 2	0	0	0	0	1
DataPoint 3	0	0	0	0	1
DataPoint 4	0	0	0	0	1
DataPoint 5	1	1	0	0	0
DataPoint 6	0	0	0	1	1
DataPoint 7	0	1	1	1	0
DataPoint 8	1	0	0	0	0

Answer 1.2

```
# to find H(Y)
def get_entropy(data):
    # Taking the last column key (Y column)
    output=data.keys()[-1]
    entropy_Y=0

    # Taking unique values of Y
    output_vals=data[output].unique()

    #calc. entropy
    for val in output_vals:
        p_i=data[output].value_counts()[val]/len(data[output])
        entropy_Y+=-p_i*np.log2(p_i)

    return entropy_Y
```

```
# to find H(Y|X)
def get_entropy_attr(data,feature):
    # Taking the last column key (Y column)
    output=data.keys()[-1]

    # Taking unique values of Y
    output_vals=data[output].unique()

    # Taking unique values of X_i
    feature_vals=data[feature].unique()
    entropy_Y_X=0

    for x_val in feature_vals:
        entropy=0
        for y_val in output_vals:
            # calc. the number of data points that satisfy the feature and
output values.

numer=len(data[feature][data[feature]==x_val][data[output]==y_val])
```

```
            # calc. the total number of data points having feature as 0 or
1
            denom=len(data[feature][data[feature]==x_val])

            p_i=numer/(denom+EPS)
            entropy+=-(p_i)*np.log2(p_i+EPS)
        entropy_Y_X+=-(denom/len(data))*entropy

    return abs(entropy_Y_X)
```

```
# IG(X) = H(Y) - H(Y|X)
def IG_partition(data):
    IG=[]

    # For all X_i
    for key in data.keys()[:-1]:
        IG.append(get_entropy(data)-get_entropy_attr(data,key))

    return data.keys()[:-1][np.argmax(IG)]
```

```
def get_subtree(data,node,value):
    return data[data[node]==value]
```

```
def ID3_build_tree(data,tree=None):
    # Taking the last column key (Y column)
    output=data.keys()[-1]

    # partitioning based on node with max IG
    node=IG_partition(data)

    # Taking unique values of X_i
    feature_vals=data[node].unique()

    if tree is None:
        tree={}
        tree[node]={}

    for x_val in feature_vals:
        #get subtrees
        subtree=get_subtree(data,node,x_val)

        #get unique output values and its respective count
        output_val,output_counts=np.unique(subtree[subtree.keys()[-
1]],return_counts=True)

        #if pure (==1 since we used .unique())
        if (len(output_counts)==1):
            tree[node][x_val]=output_val[0]
        #recursively create subtrees
        else:
            tree[node][x_val]=ID3_build_tree(subtree)

    return tree
```

```
def fit2(row,tree):
    for node in tree.keys():
```

```
        value=row[node]
        tree=tree[node][value]
        prediction=0

        #if not empty
        #if type(tree) is dict:
        if isinstance(tree, dict):
            prediction=fit2(row, tree)
        else:
            prediction=tree
            break

    return prediction
```

```
def fit(data,tree):
    avg_err=0
    for i in range(len(data)):
        prediction=fit2(data.iloc[i], tree)

        #if prediction is equal to Y val.
        if prediction!=data.iloc[i][-1]:
            avg_err+=1

    #calc. avg error
    avg_err/=len(data)

    return avg_err
```

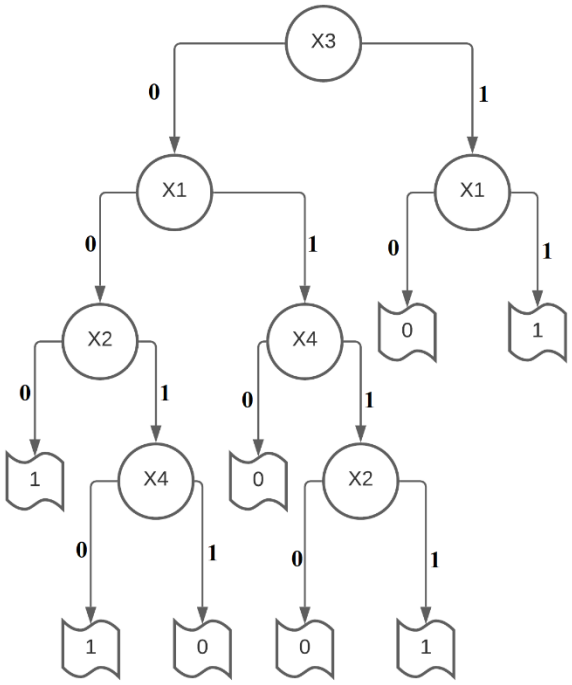
Answer 1.3

```
# init. k and m values to create dataset
k=4
m=30
dataset=gen_dataset(k,m)
```

```
#build tree with ID3 algo.
tree = ID3_build_tree(dataset)
tree
```

```
{'X3': {0: {'X1': {1: {'X4': {1: {'X2': {1: 1, 0: 0}}, 0: 0}},
    0: {'X2': {1: {'X4': {1: 0, 0: 1}}, 0: 1}}}},
  1: {'X1': {1: 1, 0: 0}}}}
```

A picture of the decision tree is given below:



The ordering makes sense, as X_1 appears twice and also appears higher in the decision tree, which suggests X1 has high information content. This is in agreement with how the data was generated since all subsequent X values (X_2 to X_k depend on the previous value of X so indirectly they are all dependent on X_1. And hence the output Y is majorly dependent on X_1.

```
err = fit(dataset, tree)
err
```

```
0.0
```

Answer 1.4

```
def get_typ_err(tree,k,m,MAX_ITER):
    typ_err=0
    for i in range(MAX_ITER):
        data=gen_dataset(k,m)
        typ_err+=fit(data,tree)

    typ_err=typ_err/MAX_ITER
    return typ_err
```

```
typ_err=get_typ_err(tree, k, m, 1000)
typ_err
```

```
0.03230000000000029
```

The new randomly generated datasets basically works as a test data (since the decision tree was built using a previous dataset) and error is about 3.23%.

Answer 1.5

```
# Simulation for diff. values (list) of m
def gen_ds_sim(k,m):
    #error list
    err_ls=[]
    for val in m:
        dataset=gen_dataset(k, val)
        tree=ID3_build_tree(dataset)
        train_err=fit(dataset, tree)
        typ_err=get_typ_err(tree, k,val,50)
        err_ls.append(abs(train_err-typ_err))
    plt.plot(m,err_ls)
    plt.xlabel("m: No. of datapoints")
    plt.ylabel("Absolute diff. between training and typical error")
    plt.title("Plot for difference of error as a function of m")
    plt.show()
    return err_ls
```

```
k=10
m_list=list(range(10,401,20))
err_ID3=gen_ds_sim(k,m_list)
m_ID3=m_list
```

The marginal value is inversely proportional to the number of datapoints (m). This is correct since having more datapoints means we get closer to the complete data which is of size 2^k, where k is the number of features.

Answer 1.6

I used the Gini Impurity (Ref: https://www.youtube.com/watch?v=7VeUPuFGJHk) metric as an alternative to Information Gain.

This method uses a concept of impurity (which is similar to entropy) to create partitions.

```
def get_gini_w(data,feature):
    # Taking the last column key (Y column)
    output=data.keys()[-1]

    # Taking unique values of Y
    output_vals=data[output].unique()

    # Taking unique values of X_i
    feature_vals=data[feature].unique()

    gini_w=0
    for x_val in feature_vals:
        gini = 1
        for y_val in output_vals:
            # calc. the number of data points that satisfy the feature and
output values.

numer=len(data[feature][data[feature]==x_val][data[output]==y_val])

            # calc. the total number of data points having feature as 0 or
1
            denom=len(data[feature][data[feature]==x_val])
            p_i=numer/(denom+EPS)
            gini-=pow(p_i,2)
        gini_w+=(denom/len(data))*gini

    return gini_w
```

```
def gini_partition(data):
    gini=[]
    # For each X_i
    for key in data.keys()[:-1]:
        gini.append(get_gini_w(data,key))

    return data.keys()[:-1][np.argmin(gini)]
```

```
def gini_build_tree(data, tree=None):
    # Taking the last column key (Y column)
    output=data.keys()[-1]

    # partitioning based on node with max gini index
    node=gini_partition(data)

    # Taking unique values of X_i
    feature_vals=data[node].unique()

    #if doesn't exist, create empty dict.
    if tree is None:
        tree={}
        tree[node]={}

    for x_val in feature_vals:
        #get subtrees
        subtree=get_subtree(data,node,x_val)

        #if pure (==1 since we used .unique())
```

```
        output_val,output_counts=np.unique(subtree[subtree.keys()[-
1]],return_counts=True)

        #if pure (==1 since we used .unique())
        if len(output_counts)==1:
            tree[node][x_val]=output_val[0]
        #recursively create subtrees
        else:
            tree[node][x_val]=gini_build_tree(subtree)

    return tree
```

```
tree_gini=gini_build_tree(dataset)
tree_gini
```

```
{'X3': {0: {'X1': {1: {'X4': {1: {'X2': {1: 1, 0: 0}}, 0: 0}},
    0: {'X2': {1: {'X4': {1: 0, 0: 1}}, 0: 1}}}},
  1: {'X1': {1: 1, 0: 0}}}}
```

```
err=fit(dataset,tree_gini)
err
```

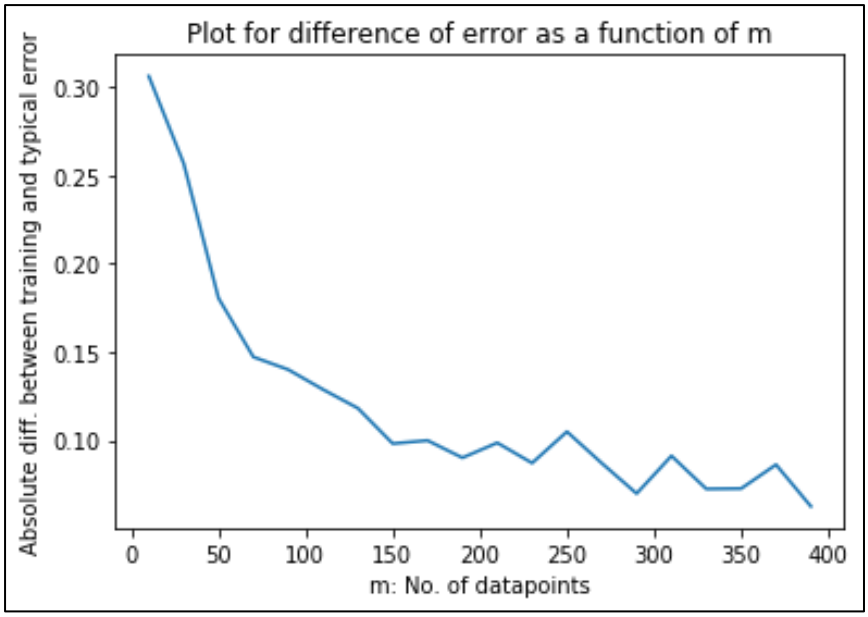
```
0.0
```

```
typ_err=get_typ_err(tree_gini,k,m,1000)
typ_err
```

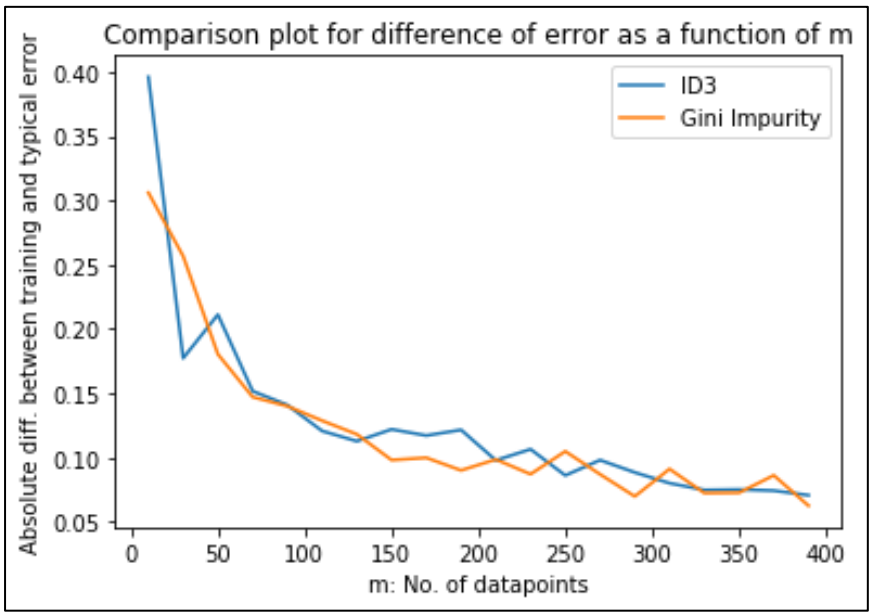
```
0.20076666666666543
```

```
# Simulation for diff. values (list) of m
def gen_gini_ds_sim(k,m):
    #error list
    err_ls=[]
    for val in m:
        dataset=gen_dataset(k, val)
        tree=gini_build_tree(dataset)
        train_err=fit(dataset, tree)
        typ_err=get_typ_err(tree, k, val, 50)
        err_ls.append(abs(train_err-typ_err))
    plt.plot(m,err_ls)
    plt.xlabel("m: No. of datapoints")
    plt.ylabel("Absolute diff. between training and typical error")
    plt.title("Plot for difference of error as a function of m")
    plt.show()
    return err_ls
```

```
k=10
m_list=list(range(10, 401, 20))
err_gini=gen_gini_ds_sim(k,m_list)
m_gini=m_list
```




```
plt.plot(m_ID3, err_ID3, label="ID3")
plt.plot(m_gini, err_gini, label="Gini Impurity")
plt.xlabel("m: No. of datapoints")
plt.ylabel("Absolute diff. between training and typical error")
plt.title("Comparison plot for difference of error as a function of m")
plt.legend()
plt.show()
```



Based on the above plot, it can be concluded that both these algorithms perform similarly.

2. Pruning Decision Trees

Answer 2.1

```
# generate m datapoints with 21 features each
def gen_X2(m):
    pass

    # init. required 2D array i.e. our data points with features as empty
    X=np.empty((m,21)).astype(int)

    for i in range(m):
        # setting first feature
        if(RNG_Prob()<0.5):
            X[i][0]=1
        else:
            X[i][0]=0

        # setting subsequent (X1 to X14) features
        for j in range(1,15):
            if(RNG_Prob()<0.75):
                X[i][j]=X[i][j-1]
            else:
                X[i][j]=1-X[i][j-1]

        # setting subsequent (X15 to X20) features (noise)
        for j in range(15,21):
            if(RNG_Prob()<0.5):
                X[i][j]=1
            else:
                X[i][j]=0
    return X


# generate Y values as per given function
def gen_Y2(m,X):
    pass

    # init. Y with as empty 1D array; each Y corresponds to each data point
    Y=np.empty((m)).astype(int)

    for i in range(m):
        if(X[i][0]==0):
            List=X[i][1:8].tolist()
            Y[i]=max(set(List),key=List.count)
        elif(X[i][0]==1):
            List=X[i][8:15].tolist()
            Y[i]=max(set(List),key=List.count)
    return Y


# generating the complete dataset having Xs and Y
def gen_dataset2(m):
    pass
    X=gen_X2(m)
    Y=gen_Y2(m,X)

    # row index for X and Y
    rows=[]
    for i in range(m):
```

```
        rows.append("DataPoint "+str(i+1))

    # column header for X
    X_header=[]
    for i in range(21):
        X_header.append("X"+str(i))

    # creating corresponding DataFrames for X and Y
    X_df=pd.DataFrame(data=X, index=rows, columns=X_header)
    Y_df=pd.DataFrame(data=Y, index=rows, columns=["Y"])

    # merging the DataFrames
    dataset=X_df.merge(Y_df,left_index=True, right_index=True)
    return dataset
```

```
m=100
dataset=gen_dataset2(m)
dataset
```

	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	...	X12	X13	X14	X15	X16	X17	X18	X19	X20	Y
DataPoint 1	0	0	0	0	0	1	1	1	1	1	...	0	0	0	1	1	0	1	1	0	0
DataPoint 2	1	1	0	1	1	1	1	1	1	0	...	1	1	1	0	0	0	0	1	1	1
DataPoint 3	0	0	0	0	0	1	1	1	0	0	...	0	0	0	1	1	1	0	1	1	0
DataPoint 4	0	0	0	0	0	0	0	1	1	0	...	0	1	0	0	0	0	1	1	1	0
DataPoint 5	1	0	0	1	0	0	1	1	1	1	...	1	1	1	0	1	1	1	0	1	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
DataPoint 96	1	1	1	1	1	1	1	1	1	1	...	1	1	1	0	0	0	1	0	1	1
DataPoint 97	1	1	1	0	0	0	0	0	0	0	...	0	0	0	1	0	1	1	1	1	0
DataPoint 98	0	1	0	0	0	0	1	1	0	0	...	1	1	0	1	1	0	0	1	1	0
DataPoint 99	1	1	0	0	0	1	1	0	1	1	...	1	0	1	0	0	0	1	1	0	1
DataPoint 100	1	0	0	0	0	0	0	0	0	0	...	0	1	1	1	0	0	0	1	1	0

100 rows × 22 columns

```
tree = ID3_build_tree(dataset)
tree
```

```
{'X10': {1: {'X4': {0: {'X14': {0: {'X7': {1: {'X8': {1: 0, 0: 1}}, 0: 0}},
      1: {'X0': {1: 1, 0: 0}}}},
    1: {'X11': {1: 1, 0: {'X6': {1: 1, 0: {'X13': {0: 0, 1: 1}}}}}}}},
  0: {'X3': {1: {'X0': {1: {'X11': {1: {'X8': {1: 1, 0: 0}}, 0: 0}},
      0: {'X11': {0: {'X9': {0: 1, 1: {'X2': {0: 1, 1: 0}}}}, 1: 0}}}},
    0: {'X8': {0: 0, 1: {'X5': {0: {'X11': {0: 0, 1: 1}}, 1: 1}}}}}}}
```

```
err = fit(dataset, tree)
err
```

```
0.0
```

```
def get_typ_err2(tree, m, MAX_ITER):
```

```
    typ_err = 0
    for i in range(MAX_ITER):
        data=gen_dataset2(m)
        typ_err+=fit(data,tree)

    typ_err/=MAX_ITER
    return typ_err
```

```
# Simulation for diff. values (list) of m
def gen_ds_sim2():
    #m = list(range(10, 2001, 100))
    err_ls=[]
    for val in m:
        dataset = gen_dataset2(val)
        tree = ID3_build_tree(dataset)
        train_err = fit(dataset, tree)
        typ_err = get_typ_err2(tree,val,50)
        err_ls.append(abs(train_err - typ_err))
    plt.plot(m, err_ls)
    plt.xlabel("m: No. of datapoints")
    plt.ylabel("Absolute diff. between training and typical error")
    plt.title("Plot for difference of error as a function of m")
    plt.show()
    return err_ls
```

```
#m=list(range(10, 10001, 1))
m=list(range(10, 2411, 200))
print(m)
typical_error=gen_ds_sim2()
```

```
[10, 210, 410, 610, 810, 1010, 1210, 1410, 1610, 1810, 2010, 2210, 2410]
```

Due to hardware and time constraints I used the above values of m. The required range is given as a comment and would take a lot of time to execute.

It agrees with our intuition because the as we increase the number of datapoints (m), the error gets lower i.e. the error is inversely proportional to the number of datapoints. Similar was the case in Answer 1.5. Also, like

before having more datapoints will increase the likelihood of the dataset reaching close being complete (2^{21} unique datapoints exist).

Answer 2.2

```
# get list of features (i.e. nodes in the decision tree)
# which were considered relevant by the algorithm
# rvl=relevant variable list
def get_unique_rvl(d, rvl):
    for key, value in d.items():
        if(isinstance(key, str)):
            rvl.append(key)
        if isinstance(value, dict):
            get_unique_rvl(value, rvl)

    return set(rvl)
```

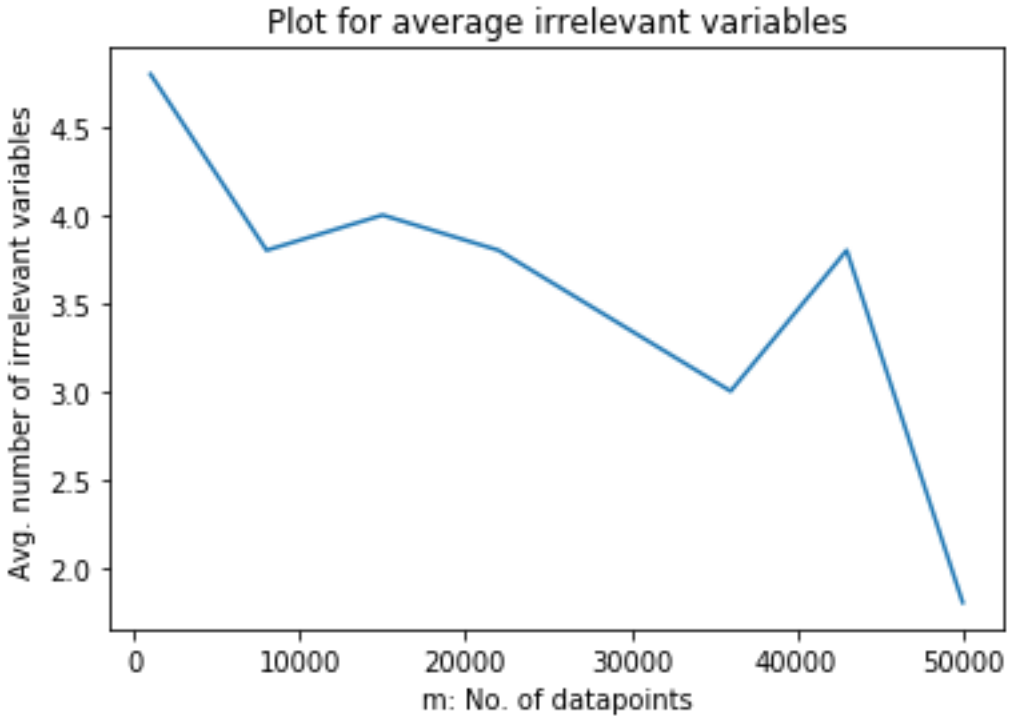
```
# get irrelevant var count by comparing with relevant vars
# ivc=irrelevant variable count
def get_ivc(ivl,rvl):
    count=0
    for item in ivl:
        if item in rvl:
            count+=1
    return count
```

```
# simulation function for getting avg. irrevant variables
# for given size of dataset
def ivc_sim(m,MAX_ITER):
    # ivc_ls=irrelevant variable count list
    ivc_ls = []
    ivl = ['X15', 'X16', 'X17', 'X18', 'X19','X20']

    for m_i in m:
        count = []
        for j in range(MAX_ITER):
            dataset2 = gen_dataset2(m_i)
            tree = ID3_build_tree(dataset2)
            rvl=[]
            get_unique_rvl(tree,rvl)
            count.append(get_ivc(ivl,rvl))

        ivc_ls.append(sum(count)/len(count))


    plt.plot(m, ivc_ls)
    plt.xlabel("m: No. of datapoints")
    plt.ylabel("Avg. number of irrelevant variables")
    plt.title("Plot for average irrelevant variables")
    plt.show()
```

```
# running the simulation
m=list(range(1000,50001,7000))
print("m values (List) =",m)
```

```
MAX_ITER=5
start = timer()
ivc_sim(m,MAX_ITER)
print("Time taken: ", timer()-start)
```

```
m values (List) = [1000, 8000, 15000, 22000, 29000, 36000, 43000, 50000]
```




```
Time taken:  2388.143747099999
```

We see from the above plot, that there is trend that the more samples we use, the lesser is the number of irrelevant variables.
But even at 50,000 samples, the average number of irrelevant variables is around 2. This suggests that we need a lot more samples to get the average number of irrelevant variables close to 0.

Answer 2.3

```
new_dataset=gen_dataset2(10000)
train_set=new_dataset[:8000]
test_set=new_dataset[8000:]
tree=ID3_build_tree(train_set)
```

```
train_error=fit(train_set,tree)
train_error
```

```
0.0
```

```
test_error=fit(test_set,tree)
test_error
```

```
0.0175
```

The train and test sets are generated and respective errors are calculated for checking the accuracy of the generated decision tree.

Answer 2.3 a

```
# Calculate the depth of input decision tree 'd'
def get_max_depth(d, depth=[], start=0):
    for key, value in d.items():
        if(isinstance(key, str)):
            # dividing by 2 since edges are not nodes, but is considered in
the dict.
            depth.append((start+2)/2)
        if isinstance(value, dict):
            get_max_depth(value, depth, start=start+1)

    return sorted(set(depth))[-1]+1
```

```
def ID3_build_tree_depth(data, depth_threshold, tree=None):
    # Taking the last column key (Y column)
    output = data.keys()[-1]

    # partitioning based on node with max IG
    node = IG_partition(data)

    # Taking unique values of X_i
    feature_vals = data[node].unique()

    if tree is None:
        tree = {}
        tree[node] = {}

    for val in feature_vals:
        # get subtrees
        subtree = get_subtree(data, node, val)

        # get unique output values and its respective count
        output_val, output_counts = np.unique(subtree[subtree.keys()[-1]],
return_counts=True)

        # if pure (==1 since we used .unique())
        if len(output_counts) == 1:
            print("here1")
            tree[node][val] = output_val[0]

        # recursively create subtrees
        else:
            # create subtree if less than depth_threshold
            if(get_max_depth(tree,[])<depth_threshold):
                tree[node][val] = ID3_build_tree_depth(subtree,
depth_threshold)

            # set leaf node i.e. the decision to max count of Y/output
values i.e. 0 or 1
            elif(get_max_depth(tree,[])==depth_threshold):
                tree[node][val] =
max(set(output_val.tolist()),key=output_val.tolist().count)
                return tree
```

```
    return tree
def pruning_by_depth_sim(depth):
    train_error=[]
    test_error=[]
    for d_i in depth:
        tree=ID3_build_tree_depth(train_set, d_i)

        train_error.append(fit(train_set,tree))
        test_error.append(fit(test_set,tree))

    plt.plot(depth, train_error, label="Training error")
    plt.plot(depth, test_error, label="Testing error")
    plt.xlabel("Depth Threshold")
    plt.ylabel("Error")
    plt.title("Plot for Pruning: Depth vs Error")
    plt.legend()
    plt.show()
```

I couldn't get the result due to some logical inconsistency while creating trees with a given threshold depth. But my approach was the following:

1. Create and append nodes to the decision tree while current depth<depth_thresh (using get_max_depth() function) using ID3 algorithm.
2. If current depth of tree is equal to the depth_threshold, then set leaf node value i.e. the decision to max count of Y/output values i.e. 0 or 1.
3. Finally, run the sim.

Answer 2.3 b

```
def get_freq(subtree):
    output_val, output_counts = np.unique(subtree[subtree.keys()[-1]],
return_counts=True)
    if len(output_counts) == 1:
        return output_val[0]
    else:
        if output_counts[1] > output_counts[0]:
            return output_val[1]
        else:
            return output_val[0]
```

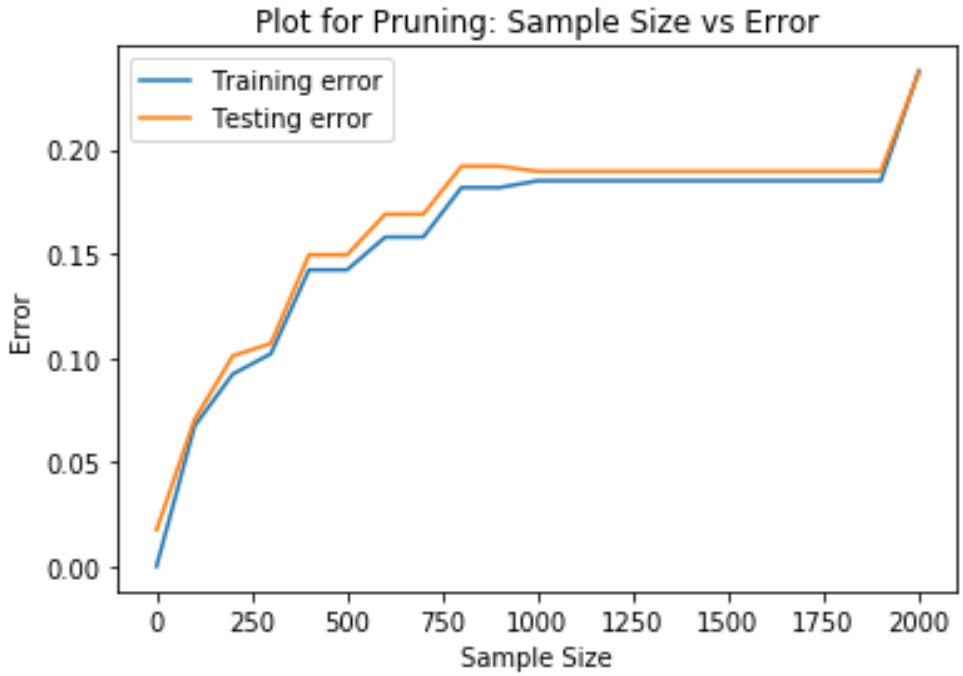
```
def ID3_build_tree_sample(data, size, tree=None):
    output = data.keys()[-1]
    node = IG_partition(data)
    feature_vals = data[node].unique()

    if tree is None:
        tree = {}
        tree[node] = {}

    for val in feature_vals:
        if len(data) <= size:
            tree[node][val] = get_freq(data)
        else:
            subtree = get_subtree(data, node, val)
```

```
            output_val, output_counts = np.unique(subtree[subtree.keys()[-
1]], return_counts=True)
            if len(output_counts) == 1:
                tree[node][val] = output_val[0]
            else:
                tree[node][val] = ID3_build_tree_sample(subtree, size)



    return tree
```

```
def pruning_by_sample_size_sim(size):
    train_error=[]
    test_error=[]
    for size_i in size:
        tree = ID3_build_tree_sample(train_set, size_i)

        train_error.append(fit(train_set,tree))
        test_error.append(fit(test_set,tree))

    plt.plot(size, train_error,label="Training error")
    plt.plot(size, test_error,label="Testing error")
    plt.xlabel("Sample Size")
    plt.ylabel("Error")
    plt.title("Plot for Pruning: Sample Size vs Error")
    plt.legend()
    plt.show()
```

```
size = list(range(1, 2002, 100))
print("size values (List) =",size)
start = timer()
pruning_by_sample_size_sim(size)
print("Time taken: ", timer()-start)
```

```
size values (List) = [1, 101, 201, 301, 401, 501, 601, 701, 801, 901, 1001,
1101, 1201, 1301, 1401, 1501, 1601, 1701, 1801, 1901, 2001]
```




```
Time taken:  253.2424092000001
```

Based on intuition, we know that lesser is the minimum size we do the split on, the more accurate will be the fit. But if this size threshold is 1 or close to it, it may result in overfitting.

From the above plot, the above intuition is proven. A good sample size threshold would be 25-125, which is around 0.3125% to 1.5525% of the training dataset (8,000).

Answer 2.4

Question 2.4 doesn't exist in the given homework.

Answer 2.5

```
def ivc_sim_depth(m,MAX_ITER,depth):
    ivc = []
    ivl = ['X15', 'X16', 'X17', 'X18', 'X19','X20']
    for m_i in m:
        count = []
        for j in range(MAX_ITER):
            train_data = gen_dataset2(m_i)
            tree = ID3_build_tree_depth(train_data, depth)

            rvl=[]
            get_unique_rvl(tree,rvl)
            count.append(get_ivc(ivl,rvl))

        ivc.append(sum(count)/len(count))

    plt.plot(m, ivc)
    plt.xlabel("m: No. of datapoints")
    plt.ylabel("Avg. number of irrelevant variables")
```

```
    plt.title("Plot for average irrelevant variables")
    plt.show()
```

```
m=list(range(1000,50001,7000))
print("m values (List) =",m)
MAX_ITER=5
# Didnt get result in 2.3 a, so taking 14 (guess)
depth_thresh=14
start = timer()
ivc_sim_size(m,MAX_ITER,size)
print("Time taken: ", timer()-start)
```

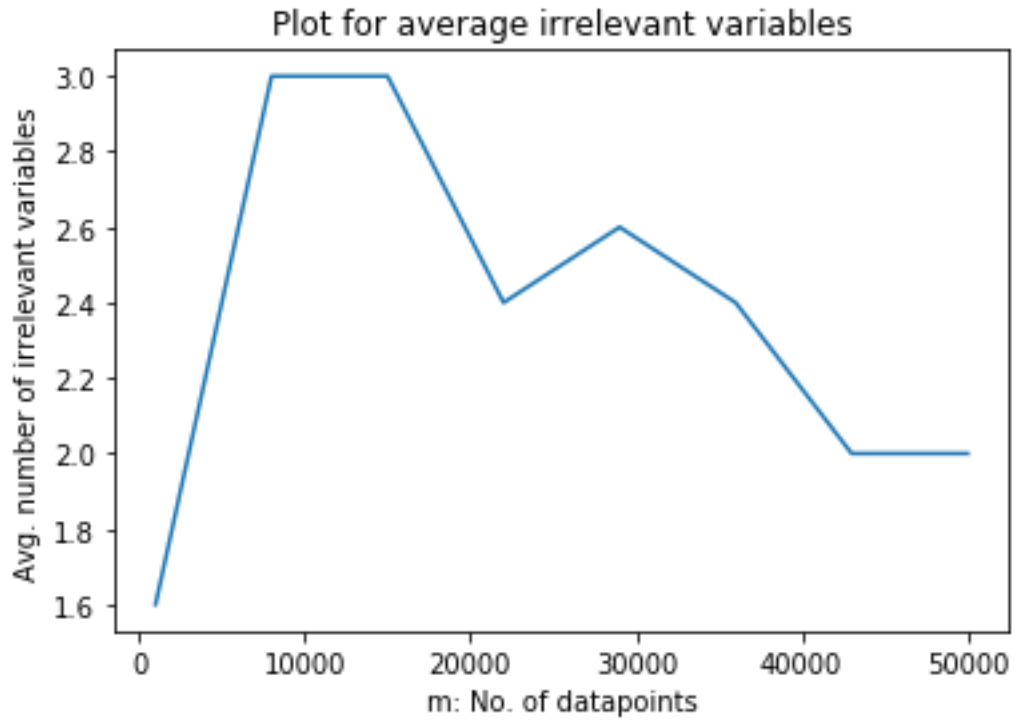
Since I couldn't obtain the best threshold depth from 2.3 a, I couldn't perform the sim. But if 2.3 would have worked, the above code should suffice the get the plot for number of datapoints vs. average number of irrelevant variables (with best depth_thresh).

Answer 2.6

```
def ivc_sim_size(m,MAX_ITER,size):
    ivc = []
    ivl = ['X15', 'X16', 'X17', 'X18', 'X19','X20']
    for m_i in m:
        count = []
        for j in range(MAX_ITER):
            train_data = gen_dataset2(m_i)
            tree = ID3_build_tree_sample(train_data, size)

            rvl=[]
            get_unique_rvl(tree,rvl)
            count.append(get_ivc(ivl,rvl))

        ivc.append(sum(count)/len(count))

    plt.plot(m, ivc)
    plt.xlabel("m: No. of datapoints")
    plt.ylabel("Avg. number of irrelevant variables")
    plt.title("Plot for average irrelevant variables")
    plt.show()
```

```
m=list(range(1000,50001,7000))
print("m values (List) =",m)
MAX_ITER=5
size=25
start = timer()
ivc_sim_size(m,MAX_ITER,size)
print("Time taken: ", timer()-start)
```

```
m values (List) = [1000, 8000, 15000, 22000, 29000, 36000, 43000, 50000]
```




```
Time taken:  2076.4013123000004
```

Took sample size as 25, since it gave good accuracy in 2.3 b.

From the above plot, we notice (similar to 2.2), that the average number of irrelevant variables is initially quite fluctuating, but after 30,000 samples, it seems to converge. If we take more datapoints, we might be able to view convergence.

But due to pruning (here by sample size), convergence is not achieved at 50,000 datapoints, which we achieved in 2.2 (no pruning).

References:

1. Lecture Notes and Videos.
2. https://tohtml.com/python/
3. https://medium.com/@lope.ai/decision-trees-from-scratch-using-id3-python-coding-it-up-6b79e3458de4
4. https://www.youtube.com/watch?v=7VeUPuFGJHk
5. https://www.lucidchart.com/

***