**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

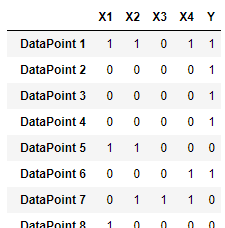
# init and test with given k and m values

k=4

m=30

dataset=gen\_dataset(k,m)

dataset



**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)

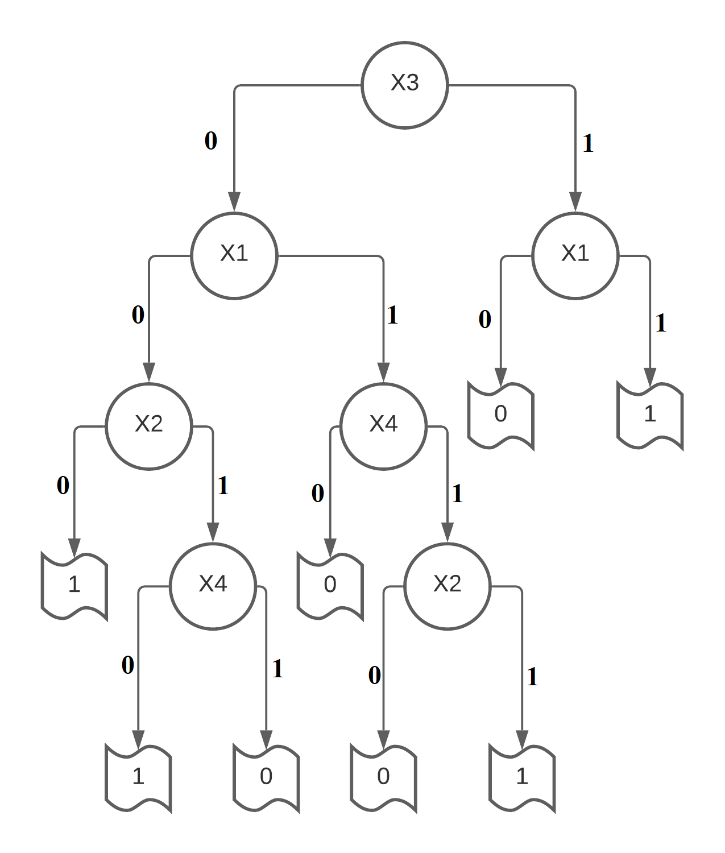
tree

{'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 X1 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 (X2 to Xk depend on the previous value of X so indirectly they are all dependent on X1. And hence the output Y is majorly dependent on X1.

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()

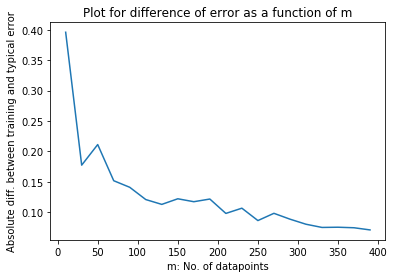
**return** err\_ls

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 2k, 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()

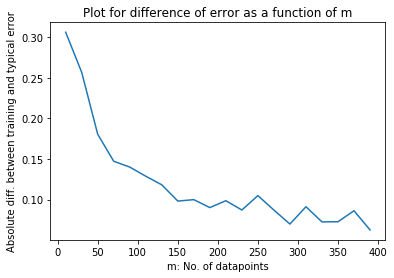
**return** err\_ls

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")

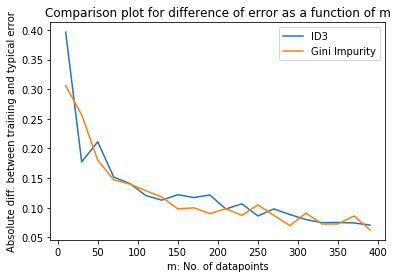
plt.xlabel("m: No. of datapoints")

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

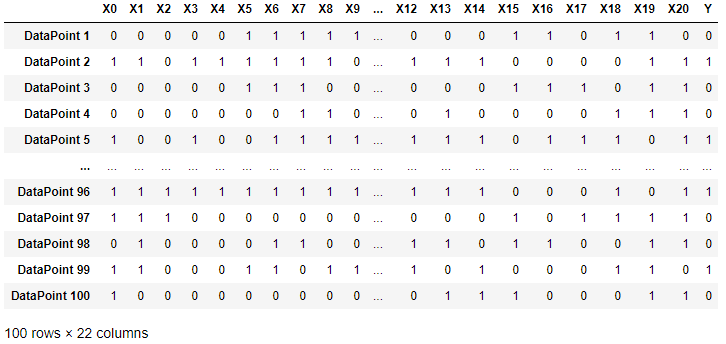
dataset=X\_df.merge(Y\_df,left\_index=True, right\_index=True)

**return** dataset

m=100

dataset=gen\_dataset2(m)

dataset



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

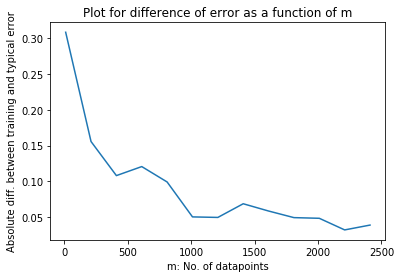
#m=list(range(10, 10001, 1))

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 (221 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)

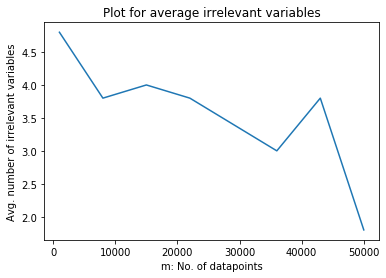
MAX\_ITER=5

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

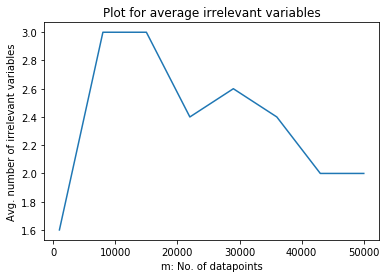
size=25

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/>

\*\*\*