DECISION_TREES

October 10, 2020

1 Decision Trees

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
from sklearn.model_selection import train_test_split

BINARY_SET = [0,1]
INT_MAX = math.pow(2,31) - 1
```

1.1 Defining Structure of Nodes for Decision Tree

[3]:
```
class Node:
    def __init__(self, feature):
        self.feature = feature # represents
        self.left = None # Represents tree for value=0 for this feature
        self.right = None # Represents tree for value=1 for this feature

    ''' Credits for the below helper functions for displaying a tree
        https://stackoverflow.com/a/54074933/5715047
    '''

    def display(self):
        lines, *_ = self._display_aux()
        for line in lines:
            print(line)

    def _display_aux(self):
        """Returns list of strings, width, height, and horizontal coordinate of␣
↪the root."""
        # No child.
        if self.right is None and self.left is None:
            line = 'Y=%s' % self.feature
            width = len(line)
            height = 1
            middle = width // 2
```

```
            return [line], width, height, middle

        # Only left child.
        if self.right is None:
            lines, n, p, x = self.left._display_aux()
            s = '%s' % self.feature
            u = len(s)
            first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s
            second_line = x * ' ' + '/' + (n - x - 1 + u) * ' '
            shifted_lines = [line + u * ' ' for line in lines]
            return [first_line, second_line] + shifted_lines, n + u, p + 2, n +␣
↪u // 2

        # Only right child.
        if self.left is None:
            lines, n, p, x = self.right._display_aux()
            s = '%s' % self.feature
            u = len(s)
            first_line = s + x * '_' + (n - x) * ' '
            second_line = (u + x) * ' ' + '\\' + (n - x - 1) * ' '
            shifted_lines = [u * ' ' + line for line in lines]
            return [first_line, second_line] + shifted_lines, n + u, p + 2, u //
↪ 2

        # Two children.
        left, n, p, x = self.left._display_aux()
        right, m, q, y = self.right._display_aux()
        s = '%s' % self.feature
        u = len(s)
        first_line = (x + 1) * ' ' + (n - x - 1) * '_' + s + y * '_' + (m - y)␣
↪* ' '
        second_line = x * ' ' + '/' + (n - x - 1 + u + y) * ' ' + '\\' + (m - y␣
↪- 1) * ' '
        if p < q:
            left += [n * ' '] * (q - p)
        elif q < p:
            right += [m * ' '] * (p - q)
        zipped_lines = zip(left, right)
        lines = [first_line, second_line] + [a + u * ' ' + b for a, b in␣
↪zipped_lines]
        return lines, n + m + u, max(p, q) + 2, n + u // 2
```

2 Decision Tree Class

[4]:
```
class DecisionTree():
    dataset = None
    k = m = 0
    H_Y = 0
    tree = None
    prune_depth = INT_MAX
    prune_sample_size = 0

    #### Uncertainty in Y
    def calc_H_Y(self):
        counts = np.bincount(np.array(self.dataset[:,-1], dtype=int),␣
↪minlength=2)

        for label in BINARY_SET:
            P_Y = counts[label]/np.sum(counts)
            self.H_Y += 0 if P_Y == 0 else -1*P_Y*math.log(P_Y,2)

    #### Information Gain
    def calculateInformationGain(self,featureIndex, dataset):
        H_Y_given_X = 0
        totalRows = np.shape(dataset)[0] # Get total rows

        for featureValue in BINARY_SET:
            rows_with_featureValue = dataset[np.where(dataset[:,featureIndex]␣
↪== featureValue)]
            P_X_featureValue = np.shape(rows_with_featureValue)[0]/totalRows  #␣
↪frequency

            if P_X_featureValue == 0: # No rows with this feature value. Hence␣
↪0 entropy
                continue

            H_Y_given_X_featureValue = 0
            for label in BINARY_SET:
                # _, counts = np.unique(dataset[[],:], return_counts=True)
                freq = rows_with_featureValue[np.where(rows_with_featureValue[:
↪,-1] == label)]
                P_Y_given_X = np.shape(freq)[0]/np.
↪shape(rows_with_featureValue)[0] # label frequency
                if P_Y_given_X  != 0:
                    H_Y_given_X_featureValue += -1* P_Y_given_X * math.
↪log(P_Y_given_X, 2)

            H_Y_given_X += P_X_featureValue*H_Y_given_X_featureValue
```

```
        return self.H_Y - H_Y_given_X

    #### Choose Decision Variable with maximum Information Gain
    def findDecisionVariable(self, dataset, level):
        # Pruning conditions
        # Depth and Sample size set to MAX by default
        # So no pruning if no values are set for these variables
        if level >= self.prune_depth or len(dataset) <= self.prune_sample_size:
            maxFreqValue = np.argmax(np.bincount(np.array(dataset[:,-1],␣
↪dtype=int), minlength=2))
            node = Node(maxFreqValue)
            return node

        maxIG = -1
        maxIG_index = -1

        # For all features
        for featureIndex in range(self.k):
            IG = self.calculateInformationGain(featureIndex, dataset)
            if IG > maxIG:
                maxIG = IG
                maxIG_index = featureIndex

        node = Node(maxIG_index)

        partition0, partition1 =  partitionDataset(dataset, maxIG_index)
        if np.shape(partition0)[0] == 1 or len(np.unique(partition0[:,-1]))==1:␣
↪# Leaf node
            node.left = Node(partition0[0][-1])
        else:
            node.left = self.findDecisionVariable(partition0, level+1) # value␣
↪= 0

        if np.shape(partition1)[0] == 1 or len(np.unique(partition1[:,-1]))==1:␣
↪# Leaf node
            node.right = Node(partition1[0][-1])
        else:
            node.right = self.findDecisionVariable(partition1, level+1) # value␣
↪= 1

        return node

    #### Build Decision Tree
    def buildTree(self):
        self.tree = self.findDecisionVariable(self.dataset, 0)
```

```
    #### Calculating the Error
    def error(self,dataset):
        err = 0
        for datapoint in dataset:
            node = self.tree
            trueLabel = datapoint[-1]
            predictedLabel = -1

            while node:
                predictedLabel = node.feature
                if datapoint[node.feature] == 0: # go left
                    node = node.left
                elif datapoint[node.feature] == 1: # go right
                    node = node.right

            if predictedLabel != trueLabel:
                err+=1

        return (err*100)/len(dataset)

    # Pruning Depth and Sample size defaulted to maximum integer values - No␣
↪pruning if these values are not provided
    def __init__(self, dataset, prune_depth=INT_MAX, prune_sample_size=0):
        self.dataset = dataset
        self.k = len(self.dataset[0]) - 1
        self.m = np.shape(self.dataset)[0]

        self.prune_depth = prune_depth
        self.prune_sample_size = prune_sample_size

        self.calc_H_Y()
        self.buildTree()
```

2.0.1 1.1 Generating Dataset

[5]:
```
''' Generating Dataset for Question 1 scheme '''
def generateData_Q1(nfeatures,n):
    datapoints = []
    labels = []

    for datapoint in range(n):
        X = [0]*nfeatures
        # p=1/2
        X[0] = np.random.choice(BINARY_SET,1)[0]
        for feature in range(1, nfeatures):
            prev = X[feature-1]
            X[feature] = np.random.choice([prev, 1-prev],1, p=[3/4, 1/4])[0]
```

```
        Y = X[1] if np.average(X[1:], weights=[math.pow(0.9, featureIndex) for
↪featureIndex in range(1,nfeatures)]) >= 1/2 else (1-X[1])
        # dataset.append((np.array(X, dtype=int),Y))
        labels.append(Y)
        datapoints.append(X)

    return np.column_stack((datapoints, labels))
```

2.0.2 1.2 Helper Function to partition dataset

(Note: The ID3 algorithm, error calculation, etc. is written in the class definition of Decision Tree above)

[6]:
```
''' This function partitions dataset based on values of variable presented at
↪featureIndex '''
def partitionDataset(dataset, featureIndex):
    partition0 = dataset[np.where(dataset[:,featureIndex] == 0)] # Rows with
↪this feature as 0
    partition1 = dataset[np.where(dataset[:,featureIndex] == 1)] # Rows with
↪this feature as 1
    return (partition0, partition1)
```

2.1 1.2 Running on a dataset

[7]:
```
k = 4
m = 30
trainingDataSet = generateData_Q1(nfeatures=k,n=m)
decisionTree = DecisionTree(trainingDataSet)
```

2.1.1 Calculating the Error

[8]:
```
err_train = decisionTree.error(trainingDataSet)
print(f"Training Error err_training(f) is : {err_train}")
```

```
Training Error err_training(f) is : 0.0
```

2.2 1.3 The Decision Tree and why it makes sense

1. The ordering of the variables in our tree shown below makes perfect sense as it shows the depth of the tree to go till max level 4. Which is the number of features.

2.2.1 Drawing the Tree

[9]:
```
print(f"The dataset is as below :\n {trainingDataSet}")
decisionTree.tree.display()
```

```
The dataset is as below :
 [[0 0 0 0 1]
 [0 0 1 1 0]
 [1 1 1 0 1]
 [1 1 1 1 1]
 [1 1 0 0 0]
 [0 0 0 1 1]
 [1 1 1 1 1]
 [0 0 1 1 0]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [1 1 1 1 1]
 [1 1 1 0 1]
 [0 1 1 1 1]
 [1 1 1 1 1]
 [0 0 0 0 1]
 [0 1 1 0 1]
 [0 0 0 1 1]
 [1 1 1 1 1]
 [0 1 1 1 1]
 [0 1 1 0 1]
 [1 1 0 0 0]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [0 0 0 0 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 0 1]
 [0 1 1 1 1]
 [0 0 0 0 1]
 [1 1 1 1 1]]
    ___1___
   /       \
  _2_     _2_
 /   \   /   \
Y=1 Y=0 Y=0 Y=1
```

2.3 1.4 - Finding average error for underlying distribution over multiple large samples

One important thing to see is that error changes when we change the tie breaking condition for features having same IG

For our distribution, since the features that comes first intuitively decides the rest data, winner of the tie-breaking should be the feature that comes first

And hence the condition to update our split variable IG < maxIG instead of IG <= maxIG

[10]:
```
times = 10
totalError = 0
for i in range(times):
    dataSize = 2000
    testingDataset = generateData_Q1(k, dataSize)
    totalError += decisionTree.error(testingDataset)

print(f"Average Error for underlying distribution err(f) is : {totalError/
 ↪times}%")
```

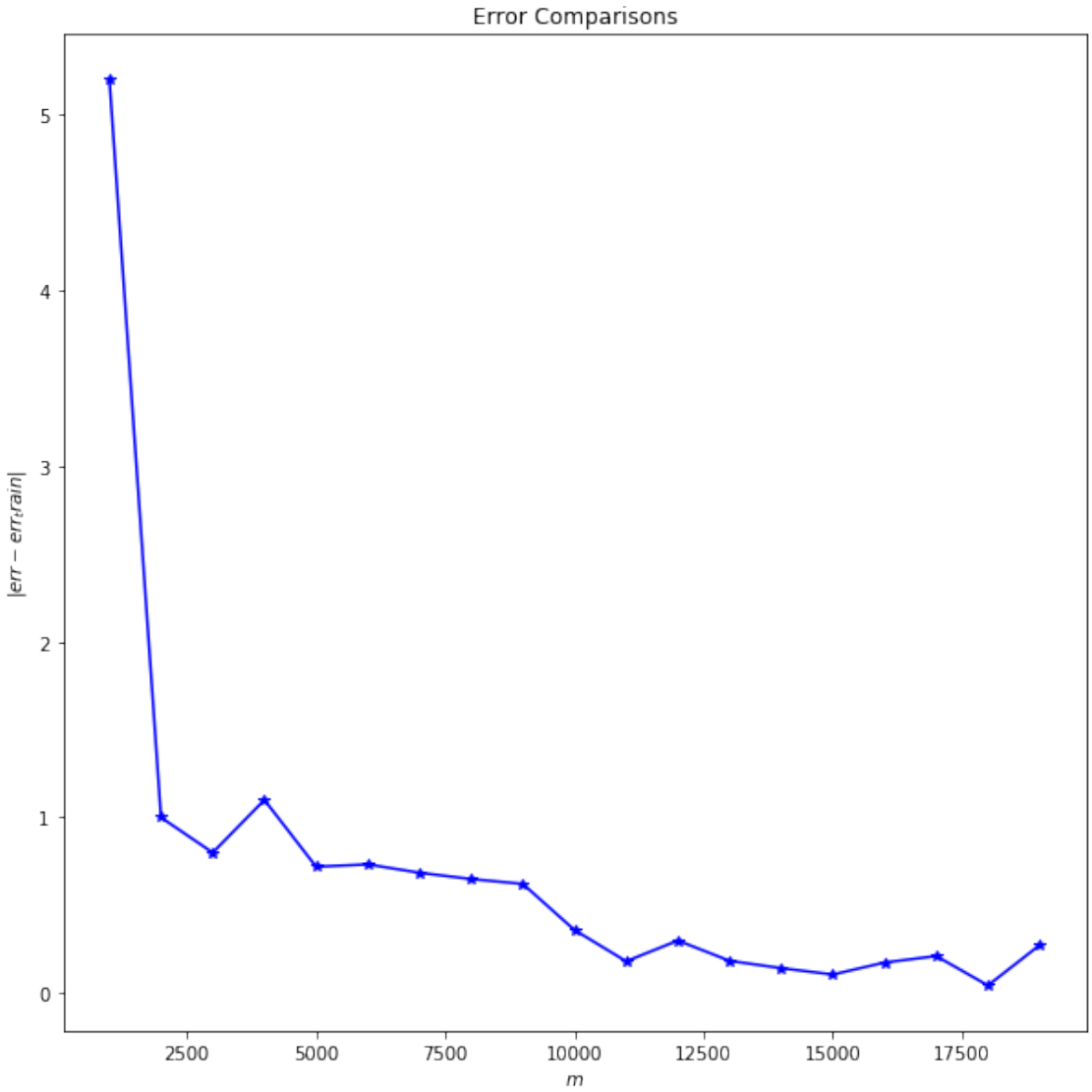
```
Average Error for underlying distribution err(f) is : 6.67%
```

2.4 1.5 - Estimating True and Training Error

The graph shows that as the value of m increases, i.e as training data size increases, we give more knowledge about our underlying distribution to the model and can thus estimate the model better.

Thus, on seeing new data under the same distribution, our model is able to estimate the true values with high probability, thus giving few errors.

[11]:
```
M = []
abs_diff_in_error = []
k = 10

for datasetSize in range(1000, 20000, 1000):
    dataset = generateData_Q1(k, datasetSize)
    trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)

    decisionTree = DecisionTree(trainingDataSet)
    err_train = decisionTree.error(trainingDataSet)
    err = decisionTree.error(testingDataset)

    abs_diff_in_error.append(math.fabs(err_train - err))
    M.append(datasetSize)

# plot graph for this now
fig, (error_compare) = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
plt.plot(np.array(M), np.array(abs_diff_in_error), marker='*', c='b')
error_compare.set_title('Error Comparisons')
error_compare.set_xlabel('$m$')
error_compare.set_ylabel('$|err - err_train|$')


plt.show()
```



1.6 Alternate metric for splitting Data - Gini algorithm

[12]:
```
''' Creating a child class instance overriding required functions '''
class DecisionTree_Gini(DecisionTree):
    def calculateGiniValue(self,dataset):
        Y_0, Y_1 = np.bincount(dataset[:,-1],minlength=2)
        totalsq = (Y_0 + Y_1)**2
        return 1 - ((Y_0**2)/totalsq + (Y_1**2)/totalsq)

    def findDecisionVariable(self,dataset, level=0):
        if len(np.shape(dataset)) == 0: # Dataset is None
            return None
```

```
        if np.shape(dataset)[0] == 1 or len(np.unique(dataset[:,-1]))==1: #␣
↪Only one Node left means leaf node or all same values
            node = Node(dataset[0][-1])
            return node

        minGI = 1
        minGI_index = -1

        # For all features
        partition0_with_min_GIndex = None
        partition1_with_min_GIndex = None
        for featureIndex in range(self.k):
            partition0, partition1 =  partitionDataset(dataset, featureIndex)

            G_0 = self.calculateGiniValue(partition0)
            G_1 = self.calculateGiniValue(partition1)

            GI = np.average([G_0, G_1],weights=[np.shape(partition0)[0], np.
↪shape(partition1)[0]])

            if GI < minGI:
                minGI = GI
                minGI_index = featureIndex
                partition0_with_min_GIndex = partition0
                partition1_with_min_GIndex = partition1

        node = Node(minGI_index)

        node.left = self.findDecisionVariable(partition0_with_min_GIndex) #␣
↪value = 0
        node.right = self.findDecisionVariable(partition1_with_min_GIndex) #␣
↪value = 1

        return node

decisionTreeGini = DecisionTree_Gini(trainingDataSet)
err_train = decisionTreeGini.error(trainingDataSet)
print(f"Error on Training Dataset err(f) is : {err_train}")
```

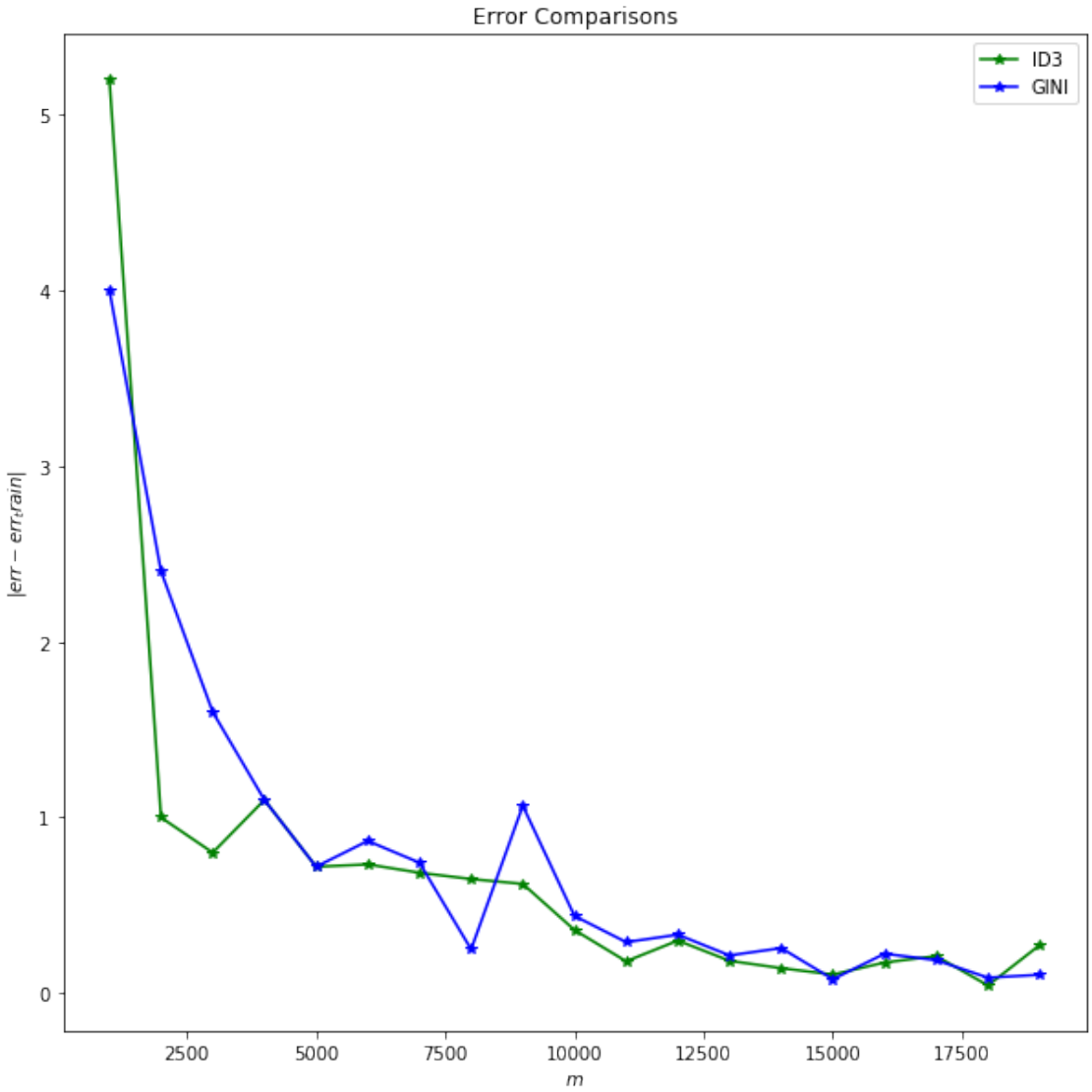
```
C:\Users\harsh\AppData\Roaming\Python\Python37\site-
packages\ipykernel_launcher.py:6: RuntimeWarning: invalid value encountered in
longlong_scalars
```

```
Error on Training Dataset err(f) is : 0.0
```

Estimating True and Training Error for Gini algorithm

[13]:
```
M_gini = []
abs_diff_in_error_gini = []

for datasetSize in range(1000, 20000, 1000):
    dataset = generateData_Q1(k, datasetSize)
    trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)

    decisionTree = DecisionTree_Gini(trainingDataSet)
    err_train = decisionTree.error(trainingDataSet)
    err = decisionTree.error(testingDataset)

    abs_diff_in_error_gini.append(math.fabs(err_train - err))
    M_gini.append(datasetSize)

# plot graph for this now
fig, (error_compare) = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
error_compare.plot(np.array(M), np.array(abs_diff_in_error), marker='*', c='g')
error_compare.plot(np.array(M_gini), np.array(abs_diff_in_error_gini),␣
 ↪marker='*', c='b')
error_compare.legend(["ID3","GINI"])
error_compare.set_title('Error Comparisons')
error_compare.set_xlabel('$m$')
error_compare.set_ylabel('$|err - err_train|$')


plt.show()
```

```
C:\Users\harsh\AppData\Roaming\Python\Python37\site-
packages\ipykernel_launcher.py:6: RuntimeWarning: invalid value encountered in
longlong_scalars
```



3 Pruning Decision Trees

2.1 Generating Dataset Using the same definition of the previously defined Decision Tree class)

[14]:
```
''' Generating Dataset for Question 2 '''
def generateData_Q2(n):
    nfeatures = 21
    datapoints = []
    labels = []

    for datapoint in range(n):
        X = [0]*nfeatures
```

```
        # p=1/2
        X[0] = np.random.choice(BINARY_SET,1)[0]
        for featureIndex in range(1, 15): # Depends on previous values
            prev = X[featureIndex-1]
            X[featureIndex] = np.random.choice([prev, 1-prev],1, p=[3/4, 1/
↪4])[0]

        for featureIndex in range(15, 21): # Independent
            X[featureIndex] = np.random.choice(BINARY_SET,1)[0]

        Y = np.max(X[1:8]) if X[0] == 0 else np.max(X[8:15])

        datapoints.append(X)
        labels.append(Y)

    return np.column_stack((datapoints, labels))
```

[15]:
```
m = 300
trainingDataSet = generateData_Q2(n=m)
decisionTree = DecisionTree(trainingDataSet)

err_train = decisionTree.error(trainingDataSet)
print(f"Training Error err_training(f) is : {err_train}")
```

```
Training Error err_training(f) is : 0.0
```

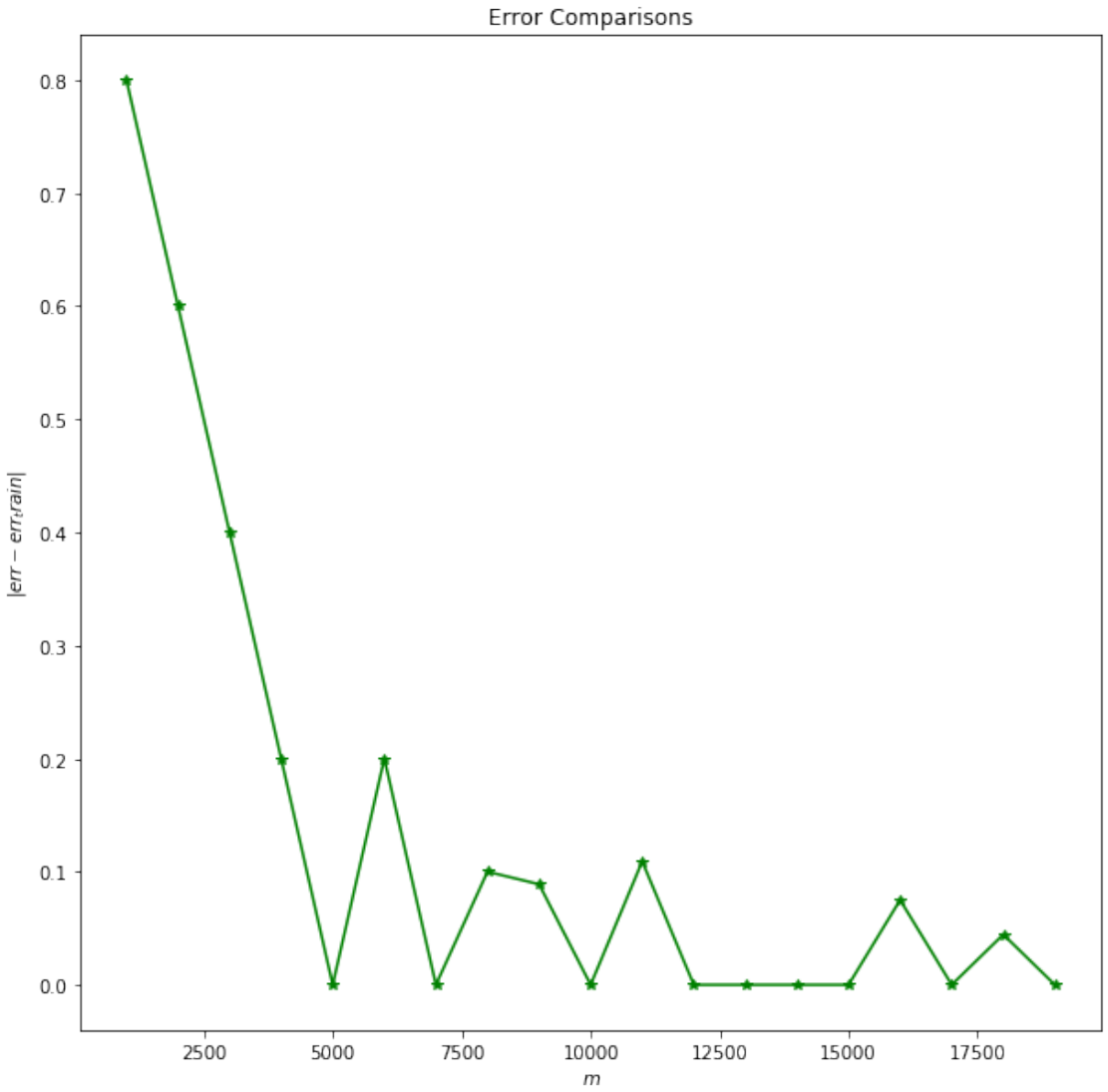
3.1 2.1 - Estimating True and Training Error

The graph shows that as the value of m increases, i.e as training data size increases, we get more knowledge about the underlying distribution and can estimate the model better than one with lesser data

Hence, as our model gets better with increasing values of m, it predicts new data well and decreases the error

[16]:
```
M = []
abs_diff_in_error = []

for datasetSize in range(1000, 20000, 1000):
    dataset = generateData_Q2(datasetSize)
    trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)

    decisionTree = DecisionTree(trainingDataSet)
    err_train = decisionTree.error(trainingDataSet)
    err = decisionTree.error(testingDataset)

    abs_diff_in_error.append(math.fabs(err_train - err))
    M.append(datasetSize)
```

```
# plot graph for this now
fig, (error_compare) = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
error_compare.plot(np.array(M), np.array(abs_diff_in_error), marker='*', c='g')
error_compare.set_title('Error Comparisons')
error_compare.set_xlabel('$m$')
error_compare.set_ylabel('$|err - err_train|$')


plt.show()
```



3.2 2.2 Average number of Irrelevant Variables

[17]:
```
irrelevantFeatures = [15,16,17,18,19,20]

def calculateSpurious(head):
    irrelevantMap = {}
    irrelevantCount = 0

    def DFS(node):
        if not node or (not node.left and not node.right):
            return

        if node.feature in irrelevantFeatures:
            irrelevantMap[node.feature] = 1

        DFS(node.left)
        DFS(node.right)

    DFS(head)
    for value in irrelevantMap.values():
        if value == 1: irrelevantCount+=1

    return irrelevantCount
```

[19]:
```
times = 0
irrelevantCount = 0

for datasetSize in range(2000, 35000, 2000):
    dataset = generateData_Q2(datasetSize)
    trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)
    decisionTree = DecisionTree(trainingDataSet)

    irrelevantCount += calculateSpurious(decisionTree.tree)

    times+=1

print(f"Average number of irrelevant variables (X15 to X20) added to our fit␣
 ↪tree : {irrelevantCount/times}")
```

```
Average number of irrelevant variables (X15 to X20) added to our fit tree :
0.29411764705882354
```

3.2.1 We see that at around 30000 our model gives no irrelvant features in the tree. This makes sense for the below reason.

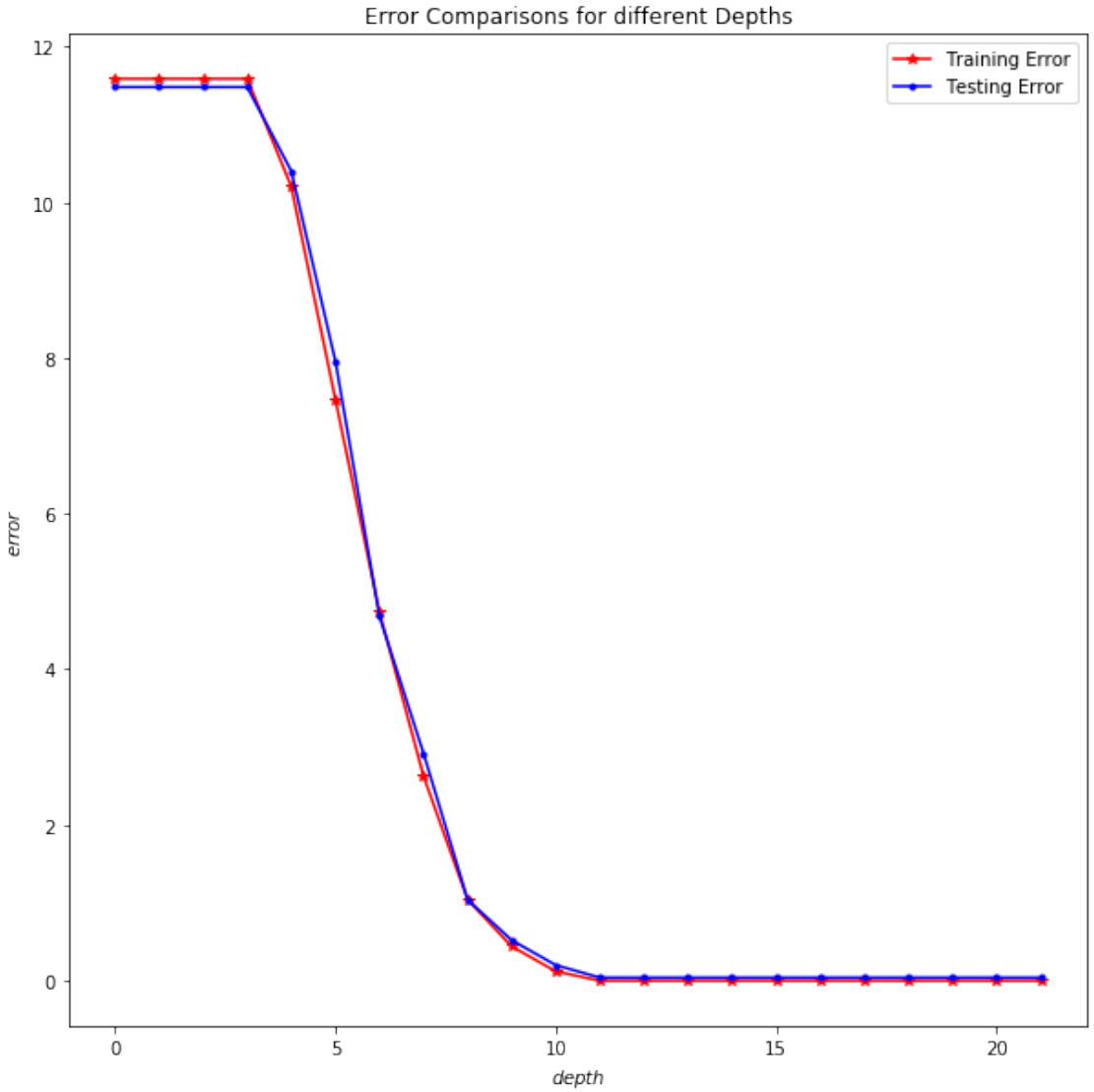
3.2.2 The total number of relevant features is 15 (0 to 14) and thus the total number of data points covered is 2^15 ~ 32000. We know that if we have the entire dataset, we can estimate our model well. Our distribution only depends on 15 features and this ~32000 is our total dataset. Thus, here's a good chance that anything over this will clearly help us identify noise. And there's a good chance that anything below this will not help us identify noise easily.

3.3 2.3 Pruning

[20]:
```
m = 10000
dataset = generateData_Q2(n=m)
trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)
```

3.3.1 2.3.a. Prune with depth

[21]:
```
depths = []
err_training_list = []
err_list = []

for depth in range(22): # Analyze for different depths

    decisionTree = DecisionTree(trainingDataSet,prune_depth=depth)
#     decisionTree.tree.display()
    err_train = decisionTree.error(trainingDataSet)
    err = decisionTree.error(testingDataset)

    depths.append(depth)
    err_training_list.append(err_train)
    err_list.append(err)


# plot graph for this now
fig, (error_compare) = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))

error_compare.plot(np.array(depths), np.array(err_training_list), marker='*',␣
 ↪c='r')
error_compare.plot(np.array(depths), np.array(err_list), marker='.', c='b')
error_compare.legend(["Training Error", "Testing Error"])
error_compare.set_title('Error Comparisons for different Depths')
error_compare.set_xlabel('$depth$')
error_compare.set_ylabel('$error$')

plt.show()
```



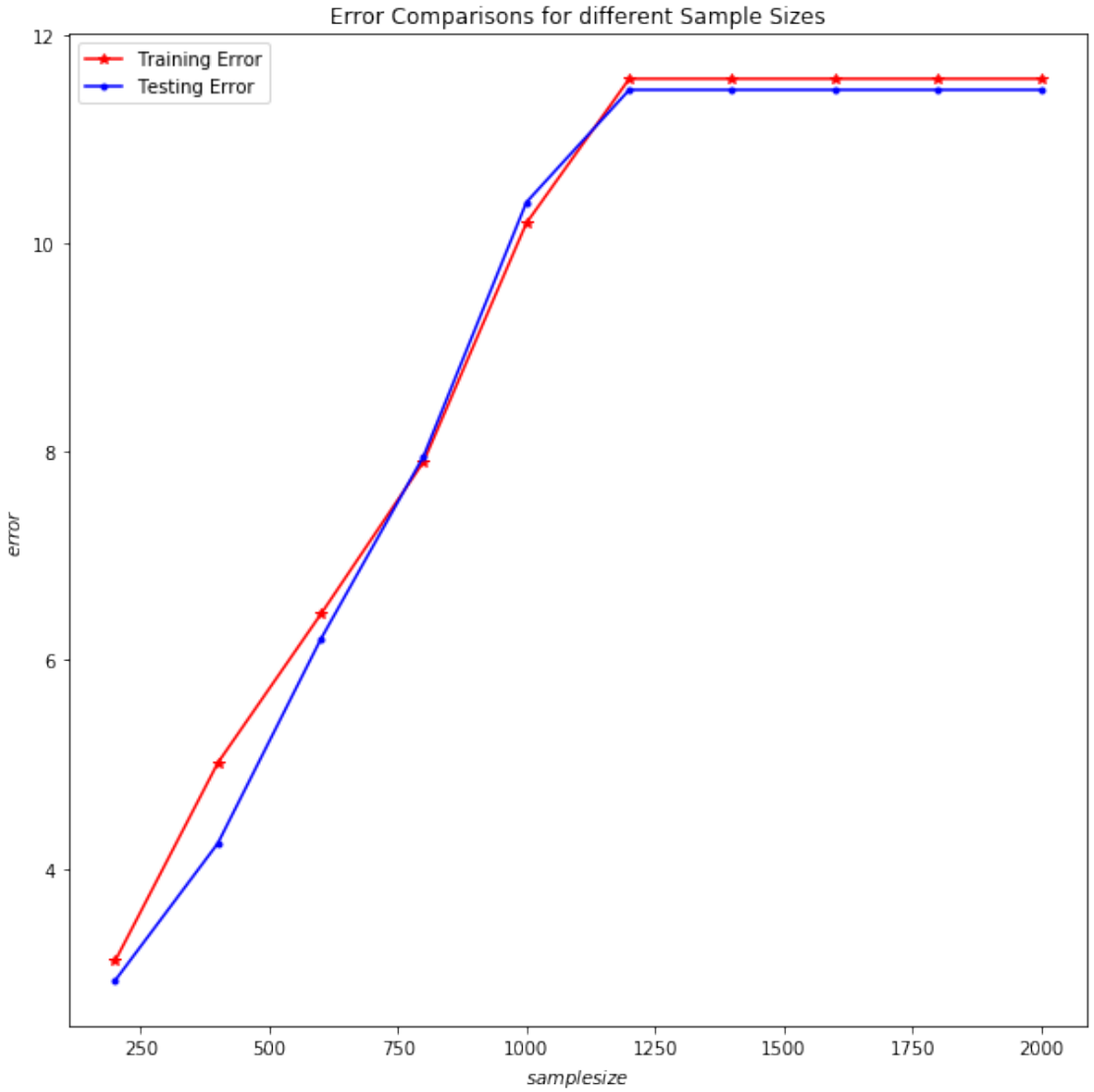
The Graph suggests that threshold depth to be around 12 as the error flatens till that point

However, intuition says that it should be 15, that's the total number of relevant features. But it's probably because of randomness and the amount of data we have trained on. Average value over a few runs and sufficient data may come around 15.

3.3.2 2.3.b Prune with sample size

[22]:
```
sampleSizes = []
err_training_list = []
err_list = []

for sampleSize in range(2000,0,-200): # Analyze for different sample sizes␣
 ↪cutoffs
```

```
    decisionTree = DecisionTree(trainingDataSet,prune_sample_size=sampleSize)
#     decisionTree.tree.display()
    err_train = decisionTree.error(trainingDataSet)
    err = decisionTree.error(testingDataset)

    sampleSizes.append(sampleSize)
    err_training_list.append(err_train)
    err_list.append(err)


# plot graph for this now
fig, (error_compare) = plt.subplots(nrows=1, ncols=1, figsize=(10, 10))
plt.plot(np.array(sampleSizes), np.array(err_training_list), marker='*', c='r')
plt.plot(np.array(sampleSizes), np.array(err_list), marker='.', c='b')
error_compare.legend(["Training Error", "Testing Error"])
error_compare.set_title('Error Comparisons for different Sample Sizes')
error_compare.set_xlabel('$sample size$')
error_compare.set_ylabel('$error$')

plt.show()
```



The Graph suggests that threshold sample size to be around 1400 as the error flatens after that point

3.3.3 2.4 Prune with Depth = 12 => Spurious Variables

3.3.4 Since we cut off the tree at a certain depth, the likelihood of having spurious variables decreases. This is because, these spurious variables have high chance to occur at a higher depth as the other relevant features take priority and occur first in the tree. Thus cutting off the tree at a depth, cuts off certain spurious varibales as well.

[25]:
```
times = 0
irrelevantCount = 0

for _ in range(0, 20):
    dataset = generateData_Q2(10000)
    trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)
    decisionTree = DecisionTree(trainingDataSet, prune_depth=12)

    irrelevantCount += calculateSpurious(decisionTree.tree)

    times+=1

print(f"Average number of irrelevant variables (X15 to X20) added to our fit␣
 ↪tree : {irrelevantCount/times}")
```

```
Average number of irrelevant variables (X15 to X20) added to our fit tree : 0.15
```

3.3.5 2.5 Prune with Sample Size = 1200 => Spurious Variables

3.3.6 Since we cut off the tree when the dataset is smaller than our threshold sample size, the likelihood of having spurious variables decreases. This is because, these spurious variables are not considered till some point as the other relevant features take priority and occur first in the tree.

[26]:
```
times = 0
irrelevantCount = 0

for datasetSize in range(0, 20):
    dataset = generateData_Q2(10000)
    trainingDataSet, testingDataset = train_test_split(dataset, shuffle=True)
    decisionTree = DecisionTree(trainingDataSet, prune_sample_size=1200)

    count = calculateSpurious(decisionTree.tree)
    irrelevantCount+=count

    times+=1

print(f"Average number of irrelevant variables (X15 to X20) added to our fit␣
 ↪tree : {irrelevantCount/times}")
```

```
Average number of irrelevant variables (X15 to X20) added to our fit tree : 0.0
```