homework2

October 11, 2020

1 CS 536: Decision Trees

Fatima AlSaadeh (fya7)

2 Part 1 Generating Decision Trees

1) For a given value of k, m, (number of features, number of data points), write a function to generate a training data set based on the above scheme.

[3]:
```
%matplotlib inline
import sys
import numpy as np
import matplotlib.pyplot as plt
```

[4]:
```
# k features, m data points
def generate_data(m, k):
    # Initialize input x with m data pints and k features
    x = np.zeros((m, k))
    # initialize output y with m output data points
    y = np.zeros(m)
    # initialize weights vectors
    w = np.zeros(k)
    # Initialize the denominator
    val = np.sum(.9 ** np.arange(1, k))
    for j in range(m):
        for i in range(k):
            if i == 0:
                # X1 = 1 with probability 1/2, X1 = 0 with probability 1/2
                x[j][i] = np.random.choice(a=[0, 1], p=[0.5, 0.5])
            else:
                # For i=2,...,k,Xi =Xi1 with probability 3/4,and Xi =1Xi1 with␣
↪probability 1/4.
                x[j][i] = np.random.choice(a=[x[j][i - 1], 1 - x[j][i - 1]],␣
↪p=[0.75, 0.25])
            # update the weights
            w[i] = ((.9 ** i) / val) * x[j][i]
        if sum(w) >= 0.5:
```

```
            # y = X1 if w2X2+w3X3+...+wkXk  1/2
            y[j] = x[j][0]
        else:
            # y = 1X1 otherwise
            y[j] = 1 - x[j][0]
    return x, y
```

2) Given a data set, write a function to fit a decision tree to that data based on splitting the variables by maximizing the information gain (ID3). Additionally, return the training error of this tree on the data set, errtrain(f) (Hint: this should be easy - why?). It may be useful to have a function that takes a data set and a variable, and returns the data set partitioned based on the values of that variable.

Answer :After we find the decision tree on the data, it is easy ti find error prediction by comparing the y training with y value predicted from the decision tree.

[5]:
```
# TreeNode class to initialize node object the carry all the information we need
class TreeNode:
    def __init__(self, id=None, y=None, zero=None, one=None, d = None):
        #node id (feature index)
        self.id = id
        #node y value
        self.y = y
        #node left zeros branch
        self.zero = zero
        #node right ones branch
        self.one = one
        #node depth in the tree
        self.d = d
```

[6]:
```
# Decision Tree class with the root node
class DecisionTree:
    def __init__(self):
        self.root = TreeNode()
```

[7]:
```
# entropy function for a given probability return the entropy (certainty) of it
def entropy(p_val):
    if p_val == 0:
        return 0
    return - p_val * np.log(p_val)
```

[8]:
```python
# for input x[:,i] and y[i] of length m find the information gain based on:
# find p(y=0|x=0),p(y=1|x=0),p(y=0|x=1),p(y=0|x=1)
# find entropy(y|x)
# find entropy(y)
# find the information gain = entropy(y) - entropy(y|x)
def get_x_to_y_count(x, y, m):
    count_x_to_y = np.zeros(4)
    for i in range(m):
        if y[i] == 0:
            if x[i] == 0:
                count_x_to_y[0] += 1
            else:
                count_x_to_y[1] += 1
        if y[i] == 1:
            if x[i] == 0:
                count_x_to_y[2] += 1
            else:
                count_x_to_y[3] += 1
    return count_x_to_y

def IG(x, y, m):
    p_y = np.count_nonzero(y) / m
    h_y = entropy(p_y) + entropy(1 - p_y)
    p_x_1 = np.count_nonzero(x) / m
    if p_x_1 == 1 or p_x_1 == 0:
        return 0
    count_x_to_y = get_x_to_y_count(x, y, m)
    p_y_x_00 = count_x_to_y[0] / (len(x) - np.count_nonzero(x))
    p_y_x_01 = count_x_to_y[1] / np.count_nonzero(x)
    p_y_x_10 = count_x_to_y[2] / (len(x) - np.count_nonzero(x))
    p_y_x_11 = count_x_to_y[3] / np.count_nonzero(x)
    h_y_x_0 = entropy(p_y_x_10) + entropy(p_y_x_00)
    h_y_x_1 = entropy(p_y_x_11) + entropy(p_y_x_01)
    h_y_x = (1 - p_x_1) * h_y_x_0 + p_x_1 * h_y_x_1
    ig = h_y - h_y_x
    return ig
```

[9]:
```python
#helper function to get the index with the max Information gain value
def get_max_ig_index(x,y,discovered):
    m, k = x.shape
    max_id = None
    max_ig = 0
    for i in range(k):
        if discovered[i] != 1:
            ig = IG(x[:, i], y, m)
            if ig > max_ig:
                max_ig = ig
                max_id = i
    return max_id

#helper function to split the data
def data_split(x,y, max_id):
    x_0 = x[x[:, max_id] == 0]
    y_0 = y[np.where(x[:, max_id] == 0)[0]]
    x_1 = x[x[:, max_id] == 1]
    y_1 = y[np.where(x[:, max_id] == 1)[0]]
    return x_0, y_0, x_1,y_1
# ID3 Implementation, x as input, y output and discovered to keep track of used␣
↪features in the tree
# for each x[i] we estimate the information gain
# pick the x[i] with maximum information gain
# split on it
# recursion on each branch(left and right) of the tree
# stop when remaining data down each branch has same u
def ID3(x, y, root, discovered):
    m, k = x.shape
    temp_discovered = np.copy(discovered)
    max_id = get_max_ig_index(x,y, temp_discovered)
    if max_id is None:
        p_y = np.count_nonzero(y) / m
        root.y = 1 if p_y >= 0.5 else 0
    else:
        temp_discovered[max_id] = 1
        if root is None:
            root = TreeNode(max_id, None, TreeNode(), TreeNode(),0)
        else:
            root.id = max_id
            root.one = TreeNode()
            root.zero = TreeNode()
        x_0, y_0, x_1,y_1 = data_split(x,y,max_id)
        if x_0.shape[0] > 0:
            ID3(x_0, y_0, root.zero, temp_discovered)
        if x_1.shape[0] > 0:
            ID3(x_1, y_1, root.one, temp_discovered)
```

[10]:
```python
# function that takes a data set and a variable, and returns the data
# set partitioned based on the values of that variable
def partition_data(data, perc):
    perc = int(len(data) * perc)
    data_train = data[:perc]
    data_test = data[perc:]
    return data_train, data_test
```

[11]:
```python
# function that iterate over the tree resulted from our ID3 algorithm and
# try to predict x_train outputs y using it
def ID3_prediction(root, x_train):
    if root.y is not None:
        return root.y
    if root == None or root.id == None:
        return -1
    if x_train[root.id] == 0:
        return ID3_prediction(root.zero, x_train)
    else:
        return ID3_prediction(root.one, x_train)
```

[12]:
```python
# function that takes x, y and decision tree resulted from ID3 algorithm and
# find the error by finiding the number of times the prediction on the training␣
↪wasn't equal the training y value
def err(x, y, root):
    m, k = x.shape
    err = 0.0
    for i in range(m):
        y_pred = ID3_prediction(root, x[i])
        if y[i] != y_pred:
            err += 1
    calc_err_train = err/ m
    return calc_err_train
def err_train(x_train, y_train, root):
    calc_err = err(x_train, y_train, root)
    return calc_err
```

3) For k = 4 and m = 30, generate data and fit a decision tree to it. Does the ordering of the variables in the decision tree make sense, based on the function that defines Y ? Why or why not? Draw the tree.

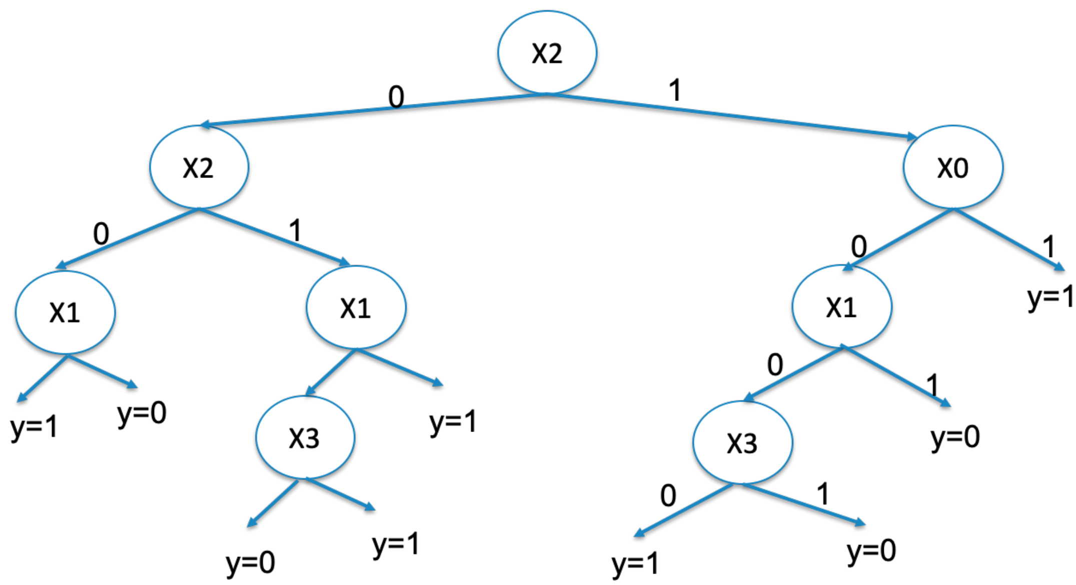
[45]:
```python
#print tree helper function
def print_tree(root, depth):
    if root.one is None or root.zero is None:
        if root.y is not None:
            print(str(depth) + "[Y]=" + str(root.y))
        return
    print(str(depth) +"[X" + str(root.id) + "] == 1:")
```

```
    print_tree(root.one, depth + 1)
    print(str(depth) + "[X" + str(root.id) + "] == 0:")
    print_tree(root.zero, depth + 1)
```

[65]:
```
m = 30
k = 4
discovered_indices = np.zeros(k)
origx, origy = generate_data(m, k)
print(origx)
print(origy)
tree = DecisionTree()
ID3(origx, origy, tree.root, discovered_indices)
print_tree(tree.root, 0)
```

```
[[1. 0. 1. 1.]
 [0. 0. 1. 1.]
 [1. 0. 0. 0.]
 [1. 1. 1. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 1.]
 [0. 1. 1. 1.]
 [0. 0. 0. 0.]
 [0. 1. 1. 1.]
 [1. 0. 0. 0.]
 [1. 1. 0. 0.]
 [1. 1. 0. 0.]
 [0. 1. 1. 0.]
 [1. 0. 0. 1.]
 [0. 1. 1. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 0.]
 [0. 1. 1. 1.]
 [0. 0. 1. 0.]
 [0. 1. 1. 1.]
 [0. 1. 1. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 1. 1.]
 [0. 0. 1. 1.]
 [0. 1. 1. 1.]
 [0. 0. 0. 0.]]
[1. 0. 0. 1. 1. 0. 0. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 0. 1. 0. 0.
 1. 1. 0. 0. 0. 1.]
```

```
0[X2] == 1:
1[X0] == 1:
2[Y]=1
1[X0] == 0:
2[X1] == 1:
3[Y]=0
2[X1] == 0:
3[X3] == 1:
4[Y]=0
3[X3] == 0:
4[Y]=1
0[X2] == 0:
1[X0] == 1:
2[X1] == 1:
3[Y]=1
2[X1] == 0:
3[X3] == 1:
4[Y]=1
3[X3] == 0:
4[Y]=0
1[X0] == 0:
2[X1] == 1:
3[Y]=0
2[X1] == 0:
3[Y]=1
```



Answer : The split data makes sense for me as with probability .75 xi depends on xi-1 value, the ordering and y=1 where the multiplied weights from x1 to xk > 1/2 and that's what appeared most of the time

4)Write a function that takes a decision tree and estimates its typical error on the underlying distribution err(f); i.e., generate a lot of data according to the above scheme, and find the average error rate of this tree over that data.

[18]:
```
def true_err(num_data, root, m, k):
    calc_err = 0
    for i in range(num_data):
        x, y = generate_data(m, k)
        calc_err += err(x, y, root)
    return calc_err / num_data
```

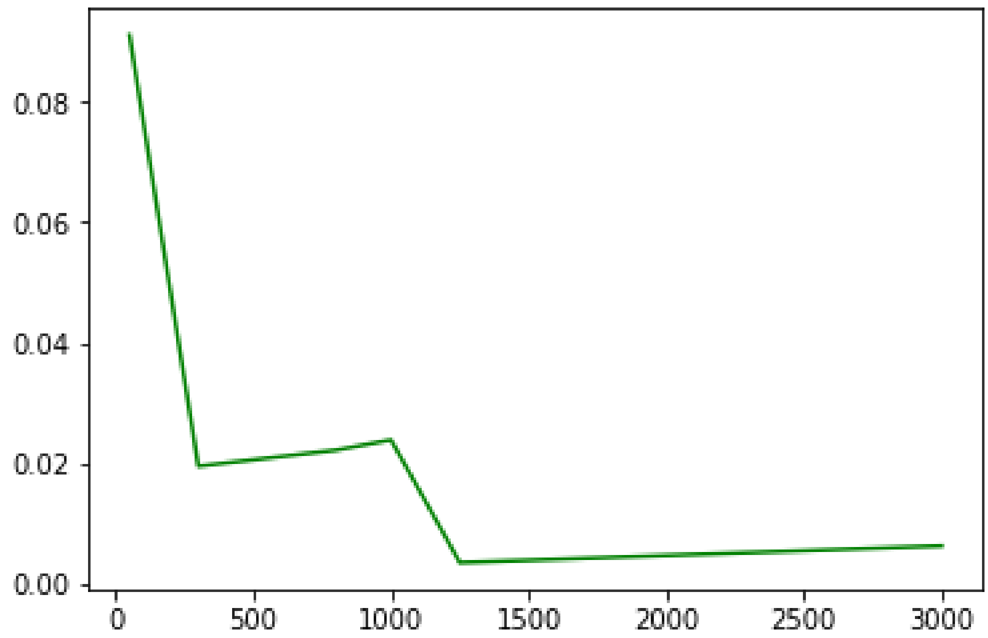
[69]:
```
true_err(1000, tree.root, m, k)
```

[69]: 0.023900000000000164

5) For k = 10, estimate the value of |errtrain(f) err(f)| for a given m by repeatedly generating data sets, fitting trees to those data sets, and estimating the true and training error. Do this for multiple m, and graph this difference as a function of m. What can you say about the marginal value of additional training data?

[24]:
```
#find the absolute error by finding the true error and training error
def err_diff(k, is_plot=True):
    m = [50, 300, 800, 1000, 1250, 3000]
    train_err_calc = np.zeros(len(m))
    true_err_calc = np.zeros(len(m))
    est_err_calc = np.zeros(len(m))
    for i in range(len(m)):
        print(m[i])
        x_train, y_train = generate_data(m[i], k)
        desc_tree = DecisionTree()
        desc_tree.root = TreeNode()
        discovered_indices = np.zeros(k)
        ID3(x_train, y_train, desc_tree.root, discovered_indices)
        train_err_calc[i]= err_train(x_train, y_train, desc_tree.root)
        true_err_calc[i]= true_err(500,desc_tree.root, m[i], k)
        est_err_calc[i] = abs(train_err_calc[i] - true_err_calc[i])
    est_err_all = abs(sum(true_err_calc) - sum(train_err_calc))/len(m)
    if is_plot:
        plt.plot(m, est_err_calc, 'g')
        plt.show()
    return est_err_calc
err_diff(10)
```

```
50
300
800
1000
1250
3000
```




```
[24]: array([0.09104   , 0.01959333, 0.0222475 , 0.023954  , 0.0036336 ,
              0.006398  ])
```

Answer : we can conclude that the absolute error calculated does decrease with the increase of m value, and given the plot as the line go close to zero when m>=1500 we can say that the marginal value of additional training data is 1500

6) Design an alternative metric for splitting the data, not based on information content / information gain. Repeat the computation from (5) above for your metric, and compare the performance of your trees vs the ID3 trees.

```
[25]: #find the variance of x[:,i]
def Var(x):
    count_ones = np.count_nonzero(x)
    count_zeros = len(x) - count_ones
    total_count = len(x)
    mean_zeros = count_zeros/total_count
    mean_ones = count_ones/total_count
    var_zeros = count_zeros * (((0-mean_zeros)**2)/total_count)
    var_ones = count_ones * (((1-mean_ones)**2)/total_count)
    var_x = ((count_zeros/total_count)*var_zeros) +  ((count_ones/
 ↪total_count)*var_ones)
    return var_x
```

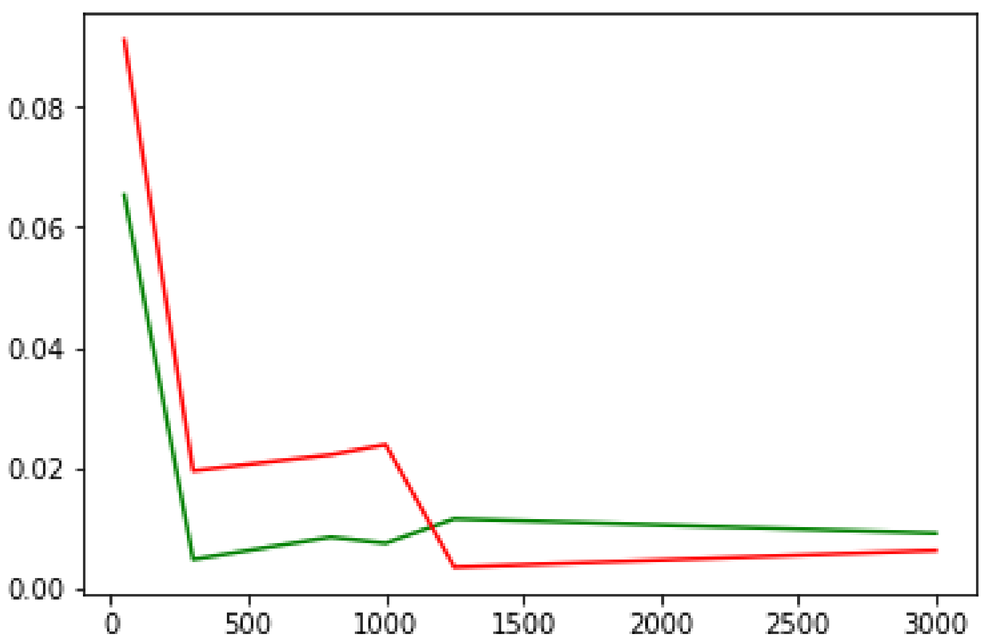
```
[26]: #helper function to get the index with the min variance value
def get_min_var_index(x,y,discovered):
```

```
    m, k = x.shape
    min_var_id = None
    min_var = sys.maxsize
    for i in range(k):
        if discovered[i] != 1:
            var = Var(x[:, i])
            if var < min_var:
                min_var = var
                min_var_id = i
    return min_var_id

#helper function to split the data
def data_split(x,y, min_id):
    x_0 = x[x[:, min_id] == 0]
    y_0 = y[np.where(x[:, min_id] == 0)[0]]
    x_1 = x[x[:, min_id] == 1]
    y_1 = y[np.where(x[:, min_id] == 1)[0]]
    return x_0, y_0, x_1,y_1

# a new verion of ID3 where we use the variance of x as a splitting metric we
#find the variance for each x and split on the minimum variance
def Variance_ID3(x, y, root, discovered):
    m, k = x.shape
    temp_discovered = np.copy(discovered)
    min_var_id = get_min_var_index(x,y,temp_discovered)
    if max_id is None:
        p_y = np.count_nonzero(y) / m
        root.y = 1 if p_y >= 0.5 else 0
    else:
        temp_discovered[min_var_id] = 1
        if root is None:
            root = TreeNode(min_var_id, None, TreeNode(), TreeNode())
        else:
            root.id = min_var_id
            root.one = TreeNode()
            root.zero = TreeNode()
        x_0, y_0, x_1,y_1 = data_split(x,y, min_var_id)
        if x_0.shape[0] > 0:
            ID3(x_0, y_0, root.zero, temp_discovered)
        if x_1.shape[0] > 1:
            ID3(x_1, y_1, root.one, temp_discovered)
```

[27]:
```
#find the absolute error by finding the true error and training error
def err_var_diff(k):
    m = [50, 300, 800, 1000, 1250, 3000]
    train_err_calc = np.zeros(len(m))
    true_err_calc = np.zeros(len(m))
```

```
    est_err_calc = np.zeros(len(m))
    for i in range(len(m)):
        print(m[i])
        x_train, y_train = generate_data(m[i], k)
        desc_tree = DecisionTree()
        desc_tree.root = TreeNode()
        discovered_indices = np.zeros(k)
        Variance_ID3(x_train, y_train, desc_tree.root, discovered_indices)
        train_err_calc[i]= err_train(x_train, y_train, desc_tree.root)
        true_err_calc[i]= true_err(500,desc_tree.root, m[i], k)
        est_err_calc[i] = abs(train_err_calc[i] - true_err_calc[i])
    est_err_all = abs(sum(true_err_calc) - sum(train_err_calc))/len(m)
    #from previous question calculation to save some running time
    est_err_all_orig = [0.09104, 0.01959333, 0.0222475 , 0.023954  , 0.0036336␣
 ↪,0.006398  ]
    plt.plot(m, est_err_calc, 'g')
    plt.plot(m, est_err_all_orig, 'r')
    plt.show()
    return est_err_calc
err_var_diff(10)
```

50
300
800
1000
1250
3000



[27]: array([0.06532   , 0.00492667, 0.008585 , 0.007584 , 0.011632 ,
       0.00923867])

Answer: The new metric "in the green" which is the variance of x, that splits the tree on the x with minimum variance used appears to be working well with absolute error around the same of the original algorithm that uses the information gain

3 Part 2 Pruning Decision Trees

1) Write a function to generate m samples of (X,Y), and another to fit a tree to that data using ID3. Write a third function to, given a decision tree f, estimate the error rate of that decision tree on the underlying data, err(f). Do this repeatedly for a range of m values, and plot the 'typical' error of a tree trained on m data points as a function of m. Does this agree with your intuition?

[18]:
```
#generate data on the new format in the pruning section
def generate_data_pruning(m, k=21):
    x = np.zeros((m, k))
    y = np.zeros(m)
    w = np.zeros(k)
    val = np.sum(.9 ** np.arange(1, k))
    for j in range(m):
        x[j][0] = np.random.choice(a=[0, 1], p=[0.5, 0.5])
        for i in range(1, k):
            if i <= 14:
                x[j][i] = np.random.choice(a=[x[j][i - 1], 1 - x[j][i - 1]],␣
↪p=[0.75, 0.25])
            else:
                x[j][i] = np.random.choice(a=[0, 1], p=[0.5, 0.5])
        if x[j][0] == 0:
            y[j] = max(set(x[j][1:7]), key=list(x[j][1:7]).count)
        else:
            y[j] = max(set(x[j][8:14]), key = list(x[j][8:14]).count)
    return x, y
```

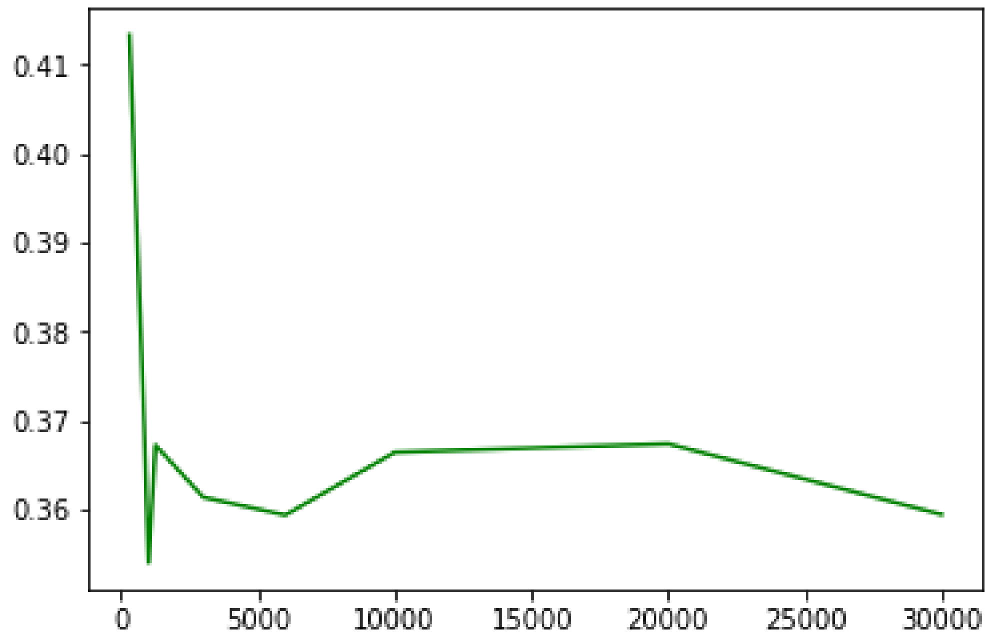
[15]:
```
# fir the tree using the new data generator
def fit_tree_pruning(m=10):
    origx, origy = generate_data_pruning(m)
    x_train, x_test = partition_data(origx, .8)
    y_train, y_test = partition_data(origy, .8)
    m, k = x_train.shape
    desc_tree = DecisionTree()
    root = TreeNode()
    discovered_indices = np.zeros(k)
    ID3(x_train, y_train, desc_tree.root, discovered_indices)
```

```
    return desc_tree.root
```

[16]:
```
def err_pruning(x, y, root):
    m, k = x.shape
    err = 0.0
    for i in range(m):
        y_pred = ID3_prediction(root, x[i])
        if y[i] != y_pred:
            err += 1
    calc_err_train = err/ m
    return calc_err_train
def true_err_pruning(num_data, root, m, k):
    calc_err = 0
    for i in range(num_data):
        x, y = generate_data_pruning(m, k)
        calc_err += err_pruning(x, y, root)
    return calc_err / num_data
```

[40]:
```
#find the typical error
def typical_err_pruning(k=21):
    m = [300, 800, 1000, 1250, 3000, 6000, 10000,20000,30000]
    true_err_calc = np.zeros(len(m))
    for i in range(len(m)):
        print(m[i])
        x_train, y_train = generate_data_pruning(m[i], k)
        desc_tree = DecisionTree()
        desc_tree.root = TreeNode()
        discovered_indices = np.zeros(k)
        ID3(x_train, y_train, desc_tree.root, discovered_indices)
        true_err_calc[i]= true_err_pruning(1,desc_tree.root, m[i], k)
    est_err_all = sum(true_err_calc)/len(m)
    plt.plot(m, true_err_calc, 'g')
    plt.show()
    return true_err_calc
typical_err_pruning()
```

```
300
800
1000
1250
3000
6000
10000
20000
30000
```




```
[40]: array([0.41333333, 0.3725    , 0.354     , 0.3672    , 0.36133333,
             0.35933333, 0.3664    , 0.36735   , 0.3594    ])
```

Answer: The error starts high but after training with larger datapoints the error starts to be lower (the decision tree get trained on data that covers more combinations), the error plot shown is with m = 30000 and looks like the error won't get any lower at some point. which is what I would expect.

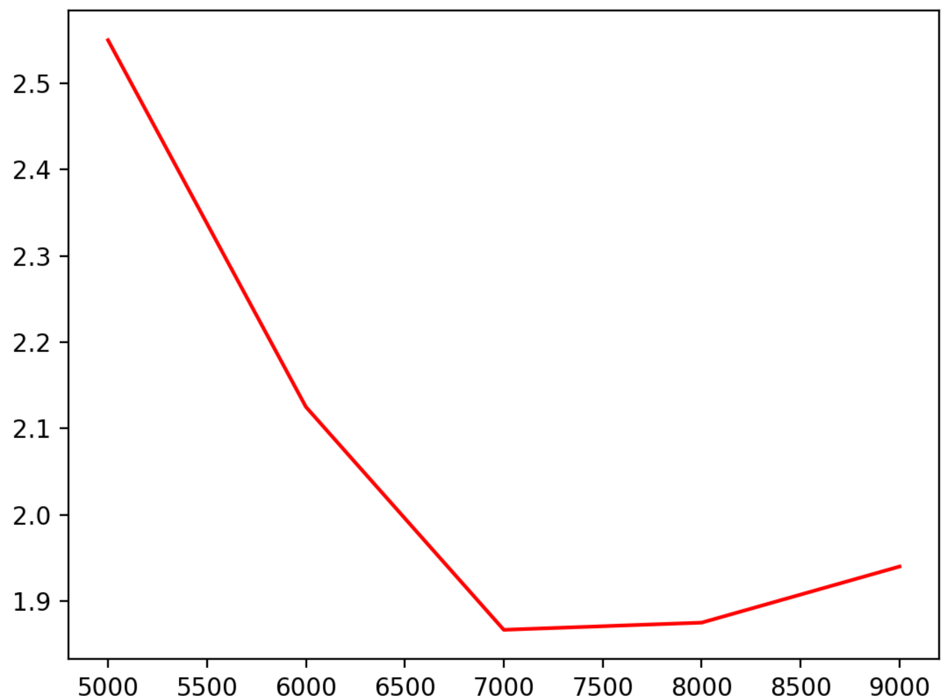
2) Note that X15 through X20 are completely irrelevant to predicting the value of Y . For a range of m values, repeatedly generate data sets of that size and fit trees to that data, and estimate the average number of irrelevant variables that are included in the fit tree. How much data would you need, typically, to avoid fitting on this noise?

```
[17]: #helper function to get the index with the max Information gain value
def get_max_ig_index(x,y,discovered):
    m, k = x.shape
    max_id = None
    max_ig = 0
    for i in range(k):
        if discovered[i] != 1:
            ig = IG(x[:, i], y, m)
            if ig > max_ig:
                max_ig = ig
                max_id = i
    return max_id
```

```python
#helper function to split the data
def data_split(x,y, max_id):
    x_0 = x[x[:, max_id] == 0]
    y_0 = y[np.where(x[:, max_id] == 0)[0]]
    x_1 = x[x[:, max_id] == 1]
    y_1 = y[np.where(x[:, max_id] == 1)[0]]
    return x_0, y_0, x_1,y_1
# new version of ID3 where we keep track of irrelevant variables
irrelevant_track = np.zeros(6)
def ID3_version2(x, y, root, discovered):
    m, k = x.shape
    temp_discovered = np.copy(discovered)
    max_id = get_max_ig_index(x,y,temp_discovered)
    if max_id is None:
        p_y = np.count_nonzero(y) / m
        root.y = 1 if p_y >= 0.5 else 0
    else:
        if max_id>=15 and max_id<=20:
            irrelevant_track[20-max_id]+=1
        temp_discovered[max_id] = 1
        if root is None:
            root = TreeNode(max_id, None, TreeNode(), TreeNode())
        else:
            root.id = max_id
            root.one = TreeNode()
            root.zero = TreeNode()
        x_0, y_0, x_1,y_1 = data_split(x,y, max_id)
        if x_0.shape[0] > 0:
            ID3_version2(x_0, y_0, root.zero,temp_discovered)
        if x_1.shape[0] > 1:
            ID3_version2(x_1, y_1, root.one,temp_discovered )
```

[19]:
```python
#find the average of irrelevant x participated in the calculation using the new␣
↪version of ID3
def find_irrelevant_avg():
    total_avg = []
    num_irrelevants = []
    m = []
    m_arr=[300,500,800,1000,1500,3000]
    for i in range(5000,10000,1000):
        print(i)
        for j in range(20):
            origx, origy = generate_data_pruning(i)
            desc_tree = DecisionTree()
            root = TreeNode()
            desc_tree.root = root
            discovered_indices = np.zeros(21)
```

```
            irrelevant_track = np.zeros(6)
            ID3_version2(origx, origy, desc_tree.root, discovered_indices)
            num_irrelevants.append(np.sum(irrelevant_track))
        m.append(i)
        total_avg.append(sum(num_irrelevants)/len(num_irrelevants))
    plt.plot(m, total_avg, 'r')
    plt.show()
    avg_irrelevants = sum(total_avg)/len(total_avg)
    print(avg_irrelevants)
find_irrelevant_avg()
```



Answer : As we see the noise become less with the data points increase based on the plot we have the number of data points should be more than 7000 in order to avoid fitting on the noise, it seems the after 8000 it increases if we test it on larger data points it gets to decrease again after 30000 data points

3) Generate a data set of size m = 10000, and set aside 8000 points for training, and 2000 points for testing. The remaining questions should all be applied to this data set.

[77]:
```
m = 10000
origx, origy = generate_data_pruning(m)
x_train, x_test = partition_data(origx, .8)
y_train, y_test = partition_data(origy, .8)
```

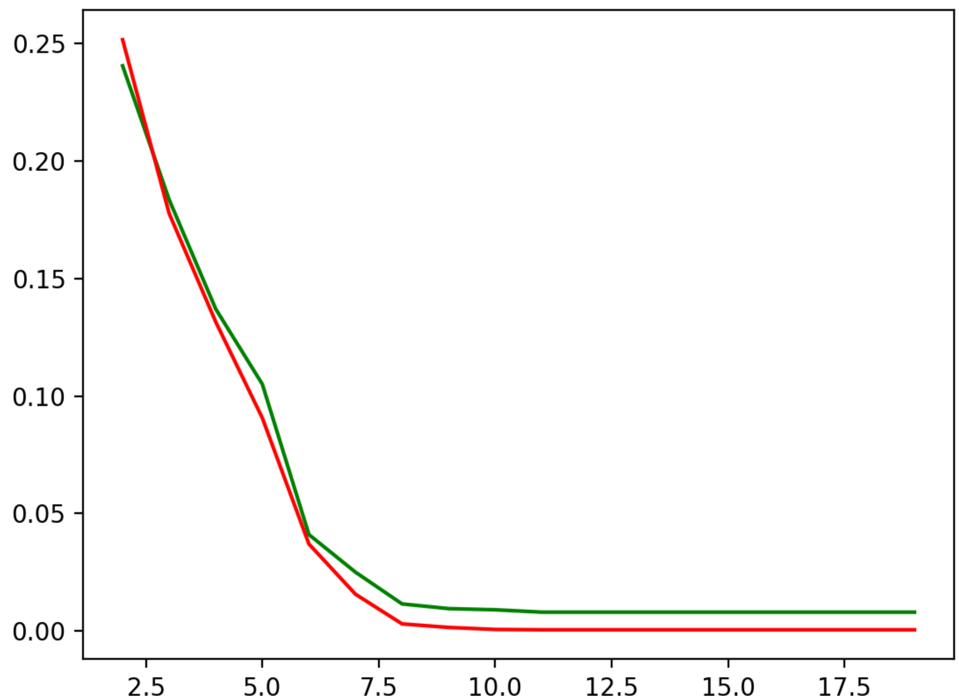
```
[[0. 0. 0. ... 0. 0. 1.]
 [0. 0. 0. ... 1. 1. 1.]
 [0. 0. 0. ... 1. 0. 1.]
 ...
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 0. 1.]
 [1. 1. 1. ... 0. 0. 0.]]
```

a) Pruning by Depth: Consider growing a tree as a process - running ID3 for instance until all splits up to depth d have been performed. Depth d = 0 should correspond to no decisions - a prediction for Y is made just on the raw frequencies of Y in the data. Plot, as a function of d, the error on the training set and the error on the test set for a tree grown to depth d. What does your data suggest as a good threshold depth?

[20]:
```
#new version of ID3 to prune by size and depth
irrelevant_track = np.zeros(6)
def ID3_pruning_version3(x, y, root, discovered, by_depth = False, d=None,␣
 ↪by_size=False, size=None):
    m, k = x.shape
    #if reach the desired depth or size break ties assign y value with .5␣
 ↪probability and return
    if (by_depth and d == 0) or (by_size and m <= size):
        root.d = d
        root.y = np.random.choice(a=[0, 1], p=[0.5, 0.5])
        return
    temp_discovered = np.copy(discovered)
    max_id = get_max_ig_index(x,y,discovered)
    if max_id is None:
        p_y = np.count_nonzero(y) / m
        root.y = 1 if p_y >= 0.5 else 0
        root.d = d
    else:
        if max_id>=15 and max_id<=20:
            irrelevant_track[20-max_id]+=1
        temp_discovered[max_id] = 1
        if root is None:
            root = TreeNode(max_id, None, TreeNode(), TreeNode())
        else:
            root.id = max_id
            root.one = TreeNode()
            root.zero = TreeNode()
        x_0, y_0, x_1,y_1 = data_split(x,y, max_id)
        if x_0.shape[0] > 0:
```

```
            if by_depth:
                d -=1
            ID3_pruning_version3(x_0, y_0, root.zero,temp_discovered, by_depth,␣
 ↪d, by_size, size)
        if x_1.shape[0] > 1:
            if by_depth:
                d -=1
            ID3_pruning_version3(x_1, y_1, root.one,temp_discovered, by_depth,␣
 ↪d, by_size, size )
```

[84]:
```
#prune the tree by depth
def prune_by_depth():
    err_train = []
    err_test = []
    d = []
    for i in range(2,20):
        desc_tree = DecisionTree()
        discovered_indices = np.zeros(21)
        root = TreeNode()
        desc_tree.root = root
        ID3_pruning_version3(x_train, y_train, desc_tree.root,␣
 ↪discovered_indices, True, i)
        err_train.append(err_pruning(x_train, y_train, desc_tree.root))
        err_test.append(err_pruning(x_test, y_test, desc_tree.root))
        d.append(i)
    plt.plot(d, err_test, 'g')
    plt.plot(d, err_train, 'r')
    plt.show()
prune_by_depth()
```

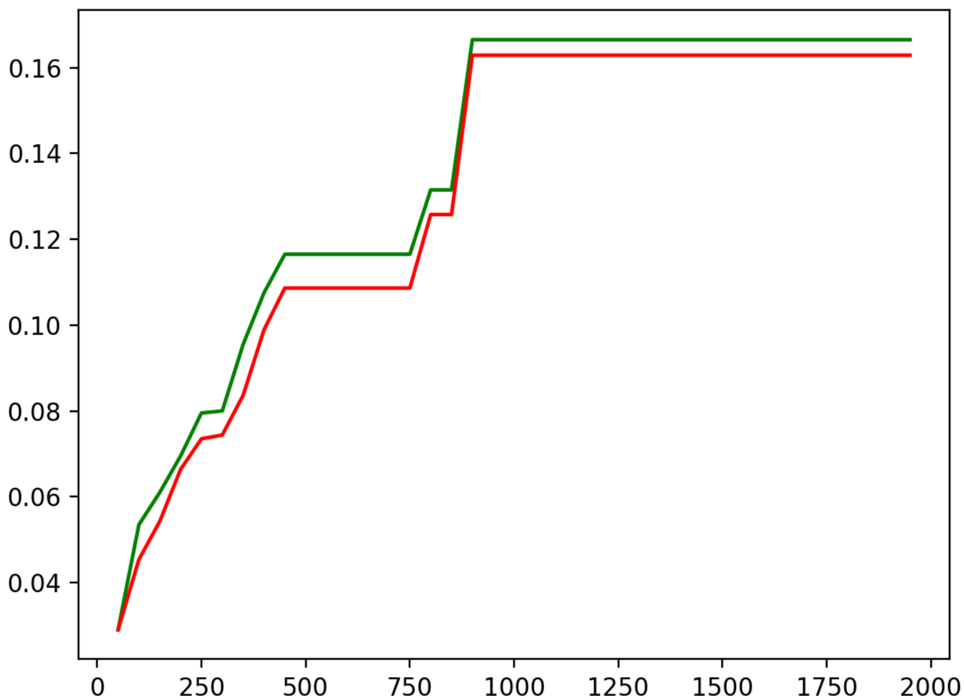


Answer: The results suggest 8 as a good depth for pruning after that the error stay almost the same.

b) Pruning by Sample Size: The less data a split is performed on, the less 'accurate' we expect the result of that split to be. Let s be a threshold such that if the data available at a node in your decision tree is less than or equal to s, you do not split and instead decide Y by simple majority vote (ties broken by coin flip). Plot, as a function of s, the error on the training set and the error on the testing set for a tree split down to sample size s. What does your data suggest as a good sample size threshold?

[21]:
```
#prune the tree by size
def prune_by_size():
    err_train = []
    err_test = []
    d = []
    for i in range(50,2000,50):
        desc_tree = DecisionTree()
        discovered_indices = np.zeros(21)
        root = TreeNode()
        desc_tree.root = root
```

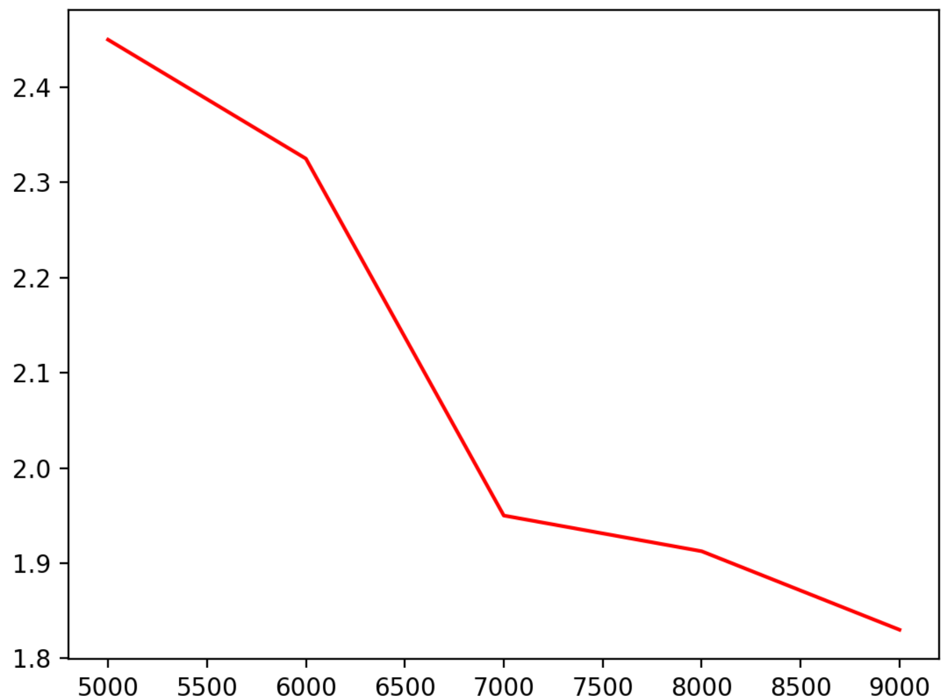
```
        ID3_pruning_version3(x_train, y_train, desc_tree.root,␣
↪discovered_indices,False,0,True,i)
        err_train.append(err_pruning(x_train, y_train, desc_tree.root))
        err_test.append(err_pruning(x_test, y_test, desc_tree.root))
        d.append(i)
    plt.plot(d, err_test, 'g')
    plt.plot(d, err_train, 'r')
    plt.show()
```



Answer: the plots are showing that the error increase with the sample size threshold increasing, it suggest threshold s=50 as the best threshold

5)

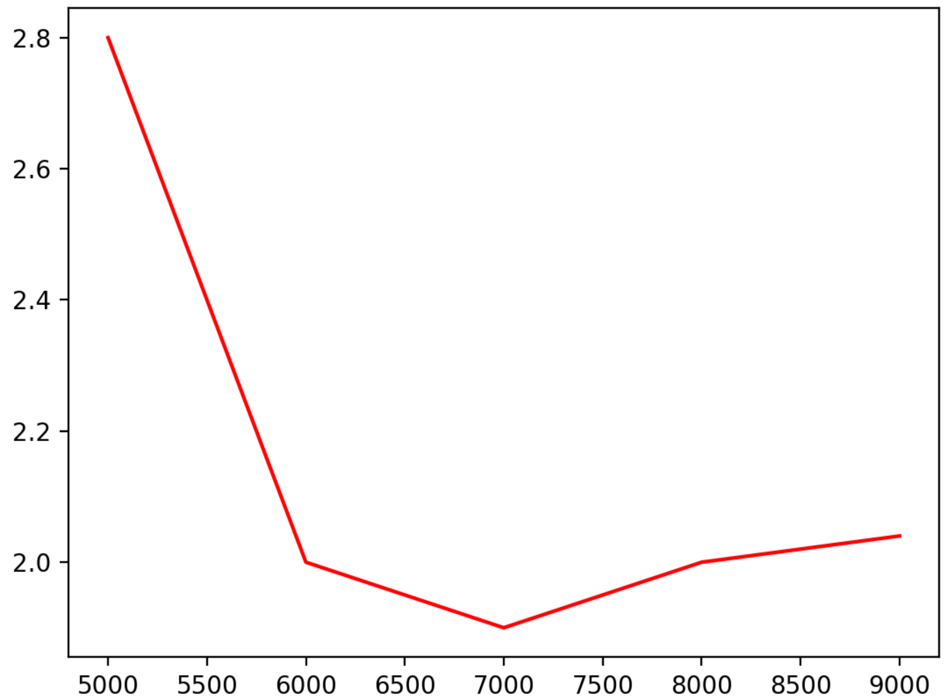
[89]:
```
#find the average irrelevant variables when we prune by the depth we calculated
def irrelevant_avg_by_depth():
    total_avg = []
    num_irrelevants = []
    m = []
    m_arr=[3000,5000,10000,20000,30000]
    for i in m_arr:
        print(i)
        for j in range(20):
            origx, origy = generate_data_pruning(i)
            desc_tree = DecisionTree()
            root = TreeNode()
            desc_tree.root = root
            discovered_indices = np.zeros(21)
            irrelevant_ track = np.zeros(6)
            ID3_pruning_version3(origx, origy, desc_tree.root,␣
 ↪discovered_indices, True, 8)
            num_irrelevants.append(np.sum(irrelevant_ track))
        m.append(i)
        total_avg.append(sum(num_irrelevants)/len(num_irrelevants))
    plt.plot(m, total_avg, 'r')
    plt.show()
    avg_irrelevants = sum(total_avg)/len(total_avg)
    print(avg_irrelevants)
irrelevant_avg_by_depth()
```



Answer: Here I'm pruning the tree on the depth I found in the previous questions depth = 8, and as the plot shows that the irrelevant values "The noise" decrease because as the tree go deeper it starts considering fitting on the noise, if we prune the depth earlier the chance to fit on noise become less
6)

[ ]:
```
#find the average irrelevant variables when we prune by the size we calculated
def irrelevant_avg_by_size():
    total_avg = []
    num_irrelevants = []
    m = []
    for i in range(5000,10000,1000):
        print(i)
        for j in range(20):
            origx, origy = generate_data(i)
            desc_tree = DecisionTree()
            root = TreeNode()
            desc_tree.root = root
            discovered_indices = np.zeros(21)
            global irrelevant_track
            irrelevant_track = np.zeros(6)
```

```
            ID3_pruning_version3(origx, origy, desc_tree.root,␣
↪discovered_indices, False, 0,True, 50)
            num_irrelevants.append(np.sum(irrelevant_track))
        m.append(i)
        total_avg.append(sum(num_irrelevants)/len(num_irrelevants))
    plt.plot(m, total_avg, 'r')
    plt.show()
    avg_irrelevants = sum(total_avg)/len(total_avg)
    print(avg_irrelevants)
```



Answer: Here I'm pruning the tree on the size threshold I found in the previous questions s = 50, and as the plot shows that considering the irrelevant values "The noise" decreases a little bit on some points "when m =6000 for example" but I think this still needs further search for a better s threshold.

Note: I was having the algorithms in a file and the questions functions in another file so for question 2,5, and 6 in pruning I was using the irrelevant_track variable as a global variable in the algorithm file algo.irrelevant_track that's way I was able to keep track of the use of the irrelevant variables using this global var