# Decision Trees

## 1. write a function to generate a training data set

Refer to function `getData(k, m)`

## 2. write a function to fit a decision tree

Refer to function `decisionTree((X, Y)).fit()`

## 3. k = 4 and m = 30

## Result:

```
D:\Users\endlesstory\Desktop\536\hw1>python DT.py
X =
[[1 1 0 0]
 [1 1 1 0]
 [0 0 0 0]
 [1 0 0 0]
 [0 0 1 1]
 [1 1 1 1]
 [1 1 1 0]
 [1 1 1 1]
 [1 1 1 0]
 [0 0 0 0]
 [1 0 0 0]
 [0 1 1 1]
 [0 0 0 0]
 [0 0 0 0]
 [1 1 1 1]
 [1 1 0 0]
 [0 0 0 0]
 [1 0 0 0]
 [1 0 1 1]
 [1 0 0 1]
```

```
 [1 0 0 0]
 [0 0 0 1]
 [1 1 1 0]
 [1 1 1 0]
 [1 1 0 0]
 [1 1 0 0]
 [1 1 1 1]
 [1 0 1 1]
 [0 0 0 0]
 [1 1 0 0]]
Y =
[0 1 1 0 0 1 1 1 1 1 0 0 1 1 1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 0]
DT.py:111: RuntimeWarning: divide by zero encountered in log2
  info = p * np.log2(p) + q * np.log2(q)
DT.py:111: RuntimeWarning: invalid value encountered in multiply
  info = p * np.log2(p) + q * np.log2(q)
********decision tree*********
error_train = 0.0
X[2] = 1, X[0] = 1, Y = 1
X[2] = 1, X[0] = 0, Y = 0
X[2] = 0, X[0] = 1, Y = 0
X[2] = 0, X[0] = 0, Y = 1
```

## Analysis:

Tree structure:

```
            X[2]
        /           \
    X[0]            X[0]
    /   \           /   \
Y=1     Y=0     Y=0     Y=1
```

It makes sort of sense, but it is still limited by the small-sized training set.

First of all, `X[2]` is strongly related to `X[1]` and `X[3]`. Namely, once `X[2] == 1`, it is much likely at least `X[1] == 1 or X[3] == 1`, and vice versa. Notice that $w_2$, $w_3$, $w_4 \approx 0.33$. Any 2 of these 3 values are `1`

will lead to $\sum_{i=2}^{4} w_i X_i > 0.5$, and vice versa. Therefore, `X[2]` does provide much information.

Second, when we have some idea of $\sum_{i=2}^{4} w_i X_i$, `Y` becomes directly depend on `X[0]`. Therefore, we need to check `X[0]` and then we could tell something about `Y`.

However, notice that it is still possile to get `_010` or `_101` with probability $0.25^2$. In these case, this tree will fail. Yet, these data points are not presented in training set due to the unlikeliness. It is reasonable to make that fault.

## 4. find the average error rate

### Result:

```
D:\Users\endlesstory\Desktop\536\hw1>python DT.py
error = 0.062533
```

### Analysis:

Notice that $0.25^2 = 0.0625$.

## 5. the marginal value of additional training data

Here is the result: (Fig. 1)

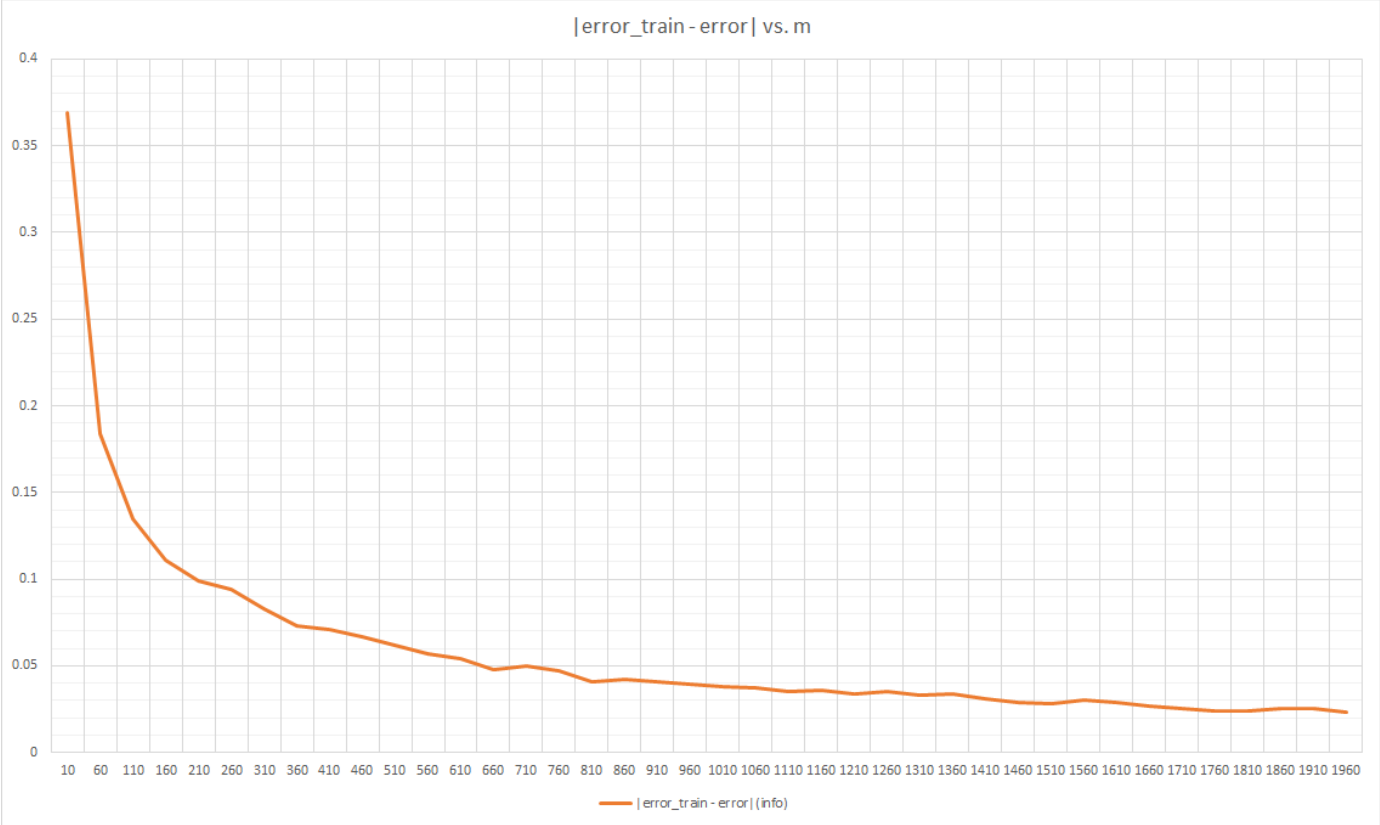


Fig.1 $|err_{train} - err|$ of the decision tree based on information gain.

Notice that when $m \geq 500$, $|err_{train} - err| \leq 0.6$, but decreases much slower.

In a nutshell, as training set gets larger and larger, the preformance of decision trees gets better and better while the progress of the preformance becomes more and more negligable.

In another word, it is much more likely to overfit when the training set is limited-sized.

## 6. an alternative metric

Notice that we want the splitted data are more likely to be only one-sided, either `0` or `1`, not both.

Say $p = P(Y = 1)$. We want $p \approx 1$ or $p \approx 0$.
Notice that $p^2 + (1 - p)^2 \leq 1$. When $p = 0.5$, it gets its minimum $0.5$, and when $p = 0 \, or \, 1$, it gets its maximumu $1$. (It is acutally a variant of Gini index.)

Previous, we choose the key that maximum information gain:

```
getKey():
    for i in range(k):
        condInfo[i] = sum(cond[i] * infoFun(condProb))
    info = infoFun(prob)
    gain[i] = info - condInfo
    return argmax(gain)
```

Now the idea becomes:

```
getKey():
    for i in range(k):
        gini[i] = sum(cond[i] * giniFun(condProb))
    return argmax(gini)

giniFun(prob):
    return square(prob) + square(1-prob)
```

Here is the result: (Fig. 2)

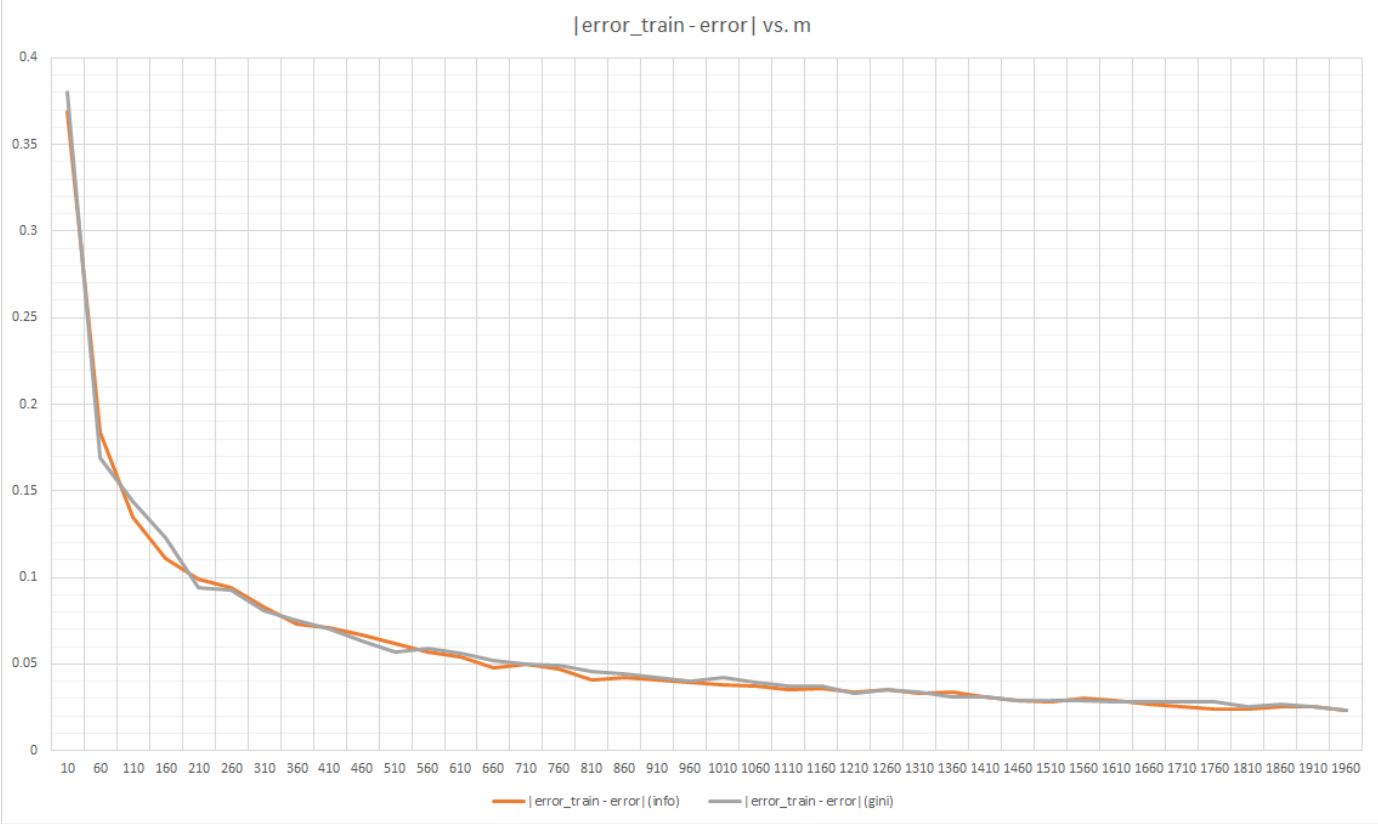


Fig2. $|err_{train} - err|$ of the decision tree based on another metric.

Notice that these 2 curve are similar. Namely, the performance are similar in term of overfitting.

However, `giniFun()` is much easier to compute than `infoFun()`. Therefore, this alternative metric is sort of better than Information Gain.