Decision Tree Problems Solutions:

1)

The functions in this function is exactly similar to the functions I have written in the last homework and it is quite easy to read and understand the code and it is not worth explaining to much, so even though professor has told us to write more than the code, I think the code is just enough in Question1. More I want to explain is totally in the annotations in the python code, so would you mind reading and learning the code, thank you. I would not talk too much about how to design the functions in this Quesiton.

1. For samples generation you should see the file,
   data_generator.py

The main function is,

```
    def data_Generator(self, m, iter = 1):
```

Use the method of weighted average sampling to general random samples and use the *collections.Counter* to get the majority of variables in a domain of *X*.

2. Should see the file,
   decision_tree_classifier.py

In the class *DecisionTreeClassifier*
The function for building a decision tree is,

```
def fit_ID3(self, data_tree, data, originaldata, features,
target_attribute_name = "Y", parent_node_class = None):
```

We use ID3 algorithm to build the tree and return the tree. And in the meanwhile, we store the tree and the tree combined with the data set partitioned in each step in the Instance of the Class *DTC* for further use.
We use information gain to split the dataset.

$$H(Y|X) = \sum_x \mathbb{P}(X = x)\, H(Y|X = x) = \sum_x \mathbb{P}(X = x) \left[-\sum_y \mathbb{P}(Y = y|X = x) \log \mathbb{P}(Y = y|X = x) \right].$$

Every time we choose the feature with the best information gain and then use this feature to partition dataset to both parts.
The structure of each tree is like:

DTC.tree:

```
{'X1': {0: {'X3': {0.0: 1.0,
                   1.0: {'X4': {0.0: {'X2': {0.0: 1.0,
                                             1.0: 0.0}},
                                1.0: 0.0}}}},
        1: {'X3': {0.0: 0.0, 1.0: {'X2': {0.0: 0.0, 1.0: 1.0}}}}}}
```

DTC.tree_with_data:

{'X1': {"data": for root node it is the original data we can denote as D1,
"child": {
{0:
{'X3': {"data": data partitioned from D1 by X1=0 so we get D2,
"child": {
{0.0: 1.0,
1.0: {'X4': {……
……

The function we use to predict the outcome given some X is:

```
def predict(self, data_tree, query, tree):
```

The function we use to calculate the $err_{train}(\hat{f})$ is:

```
def score(self, data):
```

The caller of the Class is.

```
classifier = DecisionTreeClassifier(30, dataset)
tree = classifier.fit_ID3(classifier.tree_with_data, dataset, dataset,
dataset.columns[:-1])

print("The text format tree is: ")
pprint(tree)
pprint(classifier.tree_with_data)

err_train = classifier.score(dataset)
print("err_train = ", err_train)
```

There are no other things need to specific because I think I have written enough annotations for understanding the code.

3. The function is in file *cal_typical_error.py*, the function is:

```
def test_score(self, m, iteration, classifier):
```

The method to call the function is in *demo.py*:

```
typical_err = CalTypicalError()
err_test = typical_err.test_score(30, 5000, classifier)
```

4. The code is in *question1.py*.

For every m, we know that $|H| = 2^{2^k} = 2^{2\,million}$, but my computer is not strong enough for so much data, and from last homework I choose the same 4×10^4 data for test and validate.

And the m is from 10 to 10000, for convenience, I choose the m as following list shows. M = [10,20,30,40,50,60,70,80,90,100,200,300,400,500,600,700,800,900,1000,2000,3000,4000,5000,6000,7000,8000,9000,10000].

In my intuition, I think when m is bigger, the depth of the tree should be larger and because there is more information for prediction, the typical error of the tree should become smaller.
Below is the graph of the result.

Fig 1: Typical error of tree for different value of m.

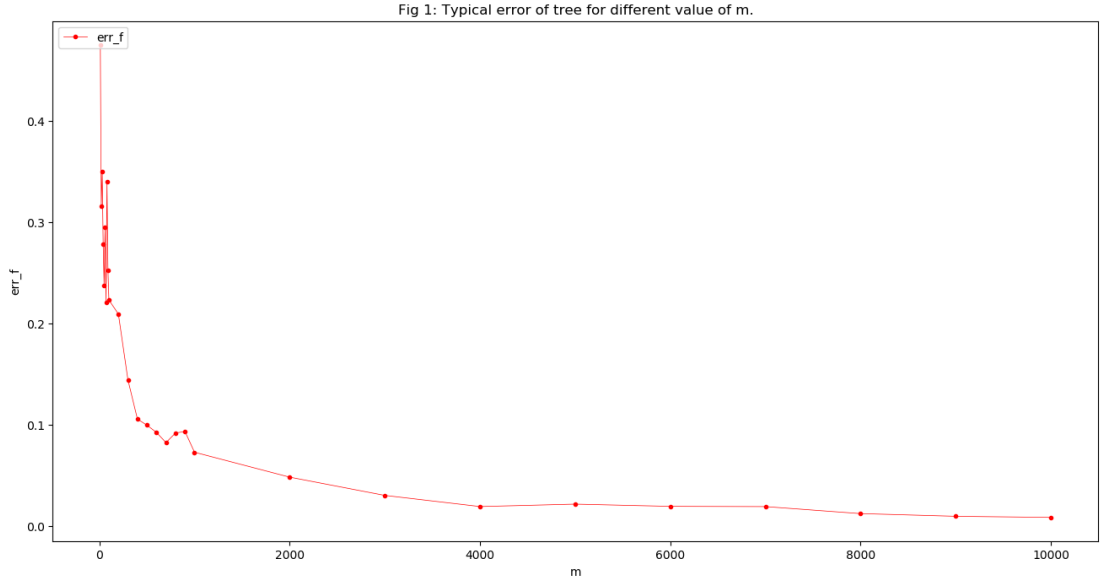
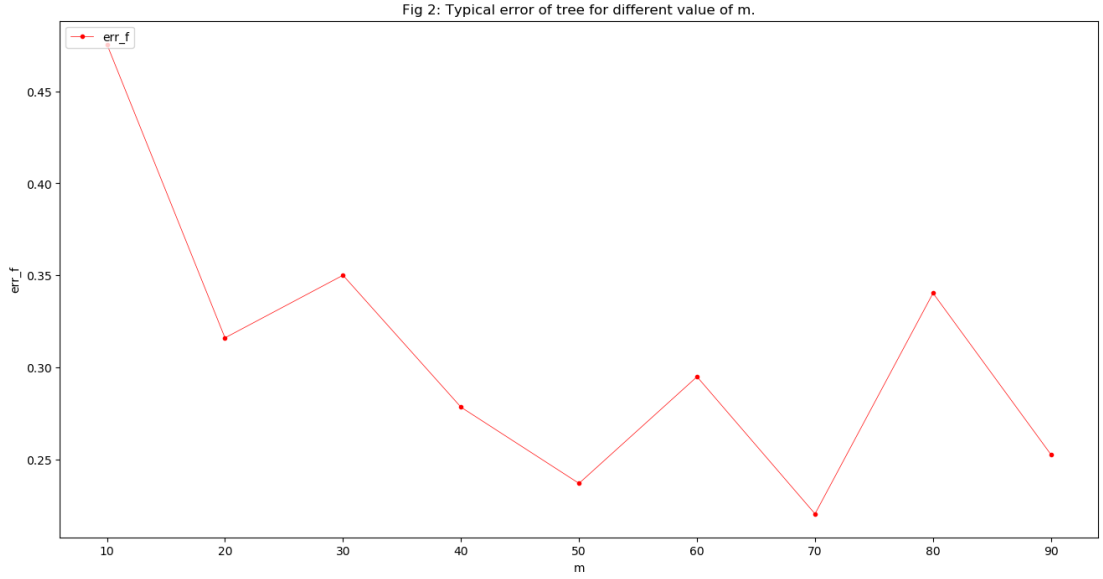


Fig 2: Typical error of tree for different value of m.



We can see that generally the error is becoming smaller when m is getting larger. But there may be some cases that contradict my intuition as we seen in the Fig 2.
But in general, the outcome fits my intuition.

2)

I have supervised the process when ran the question 1, and I found that all the trees I have built in the question 1 have the noise tree node. But when the tree becomes deeper, the proportion of noise node in all tree nodes is becoming smaller.
Considering the limit of my laptop, I will use the following m and number of trees for each m.

m in M = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 5000, 10000, 50000, 100000].

And for each value of m, I general 100 trees. And rather than counting how many irrelevant features have occurred, I choose to count the amount of noise nodes in the decision, because I think if there is a noise node, it will bring about a new noise whether they are caused by the same features or not. So, I think we should treat them as different noise and count them independently. That is we count the times of a noise node's occurrence rather than how many different irrelevant features are there in the tree. So, the average number of irrelevant variables for each tree may be larger than 6 because we count the duplicate features. The same as the following questions 5, 6 and 7.

Then I got the following results.

m	Average number of irrelevant variables' nodes for each tree
10	0.2
20	0.5
30	0.76
40	1.04
50	0.76
60	1.3
70	1.2
80	1.44
90	1.86
100	1.82
200	3.16
300	3.1
400	3.78
500	4.3
600	4.42
700	6.02
800	4.94
900	6.26
1000	5.78
5000	6.2
10000	5.04
50000	4.42
100000	4.22

From the table, we can estimate that the amount of data we need the more is the better. And I think the phenomena occur when m is less than 900 because when m is getting bigger, the more complex the tree is, then the tree will include more variables as its nodes, so the noise will have larger probability to occur, so these conditions should not be included. And in the above form, we can see that when m is greater than 5000, the frequency will decreases with the increase of m, since I don't have enough computational resources to generate sufficient number of trees. So given the result

mentioned above, maybe I shall states that typically the amount of data I need to avoid fitting on noise should be greater than 100000, that is, the more data we have, the less frequent the noise occur.

3)

a) I use a d to control the max depth of the decision tree, if the depth of tree is not exceeding *d*, we will continue growing the tree.

This method is defined in the following function:

```
def fit_ID3_Pruning_Depth
```

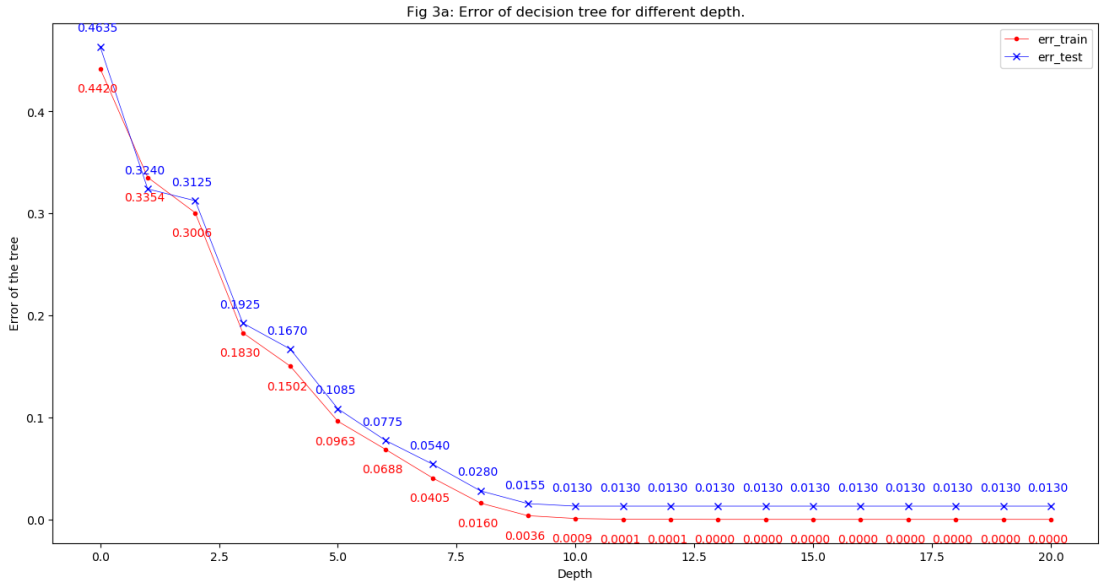
Comparing to the original fit function, I add the following code:

```
# if depth is 0, just return with data.
if depth == 0:
    # If the sub dataset is empty, return the mode target feature value
in the original dataset
    # >>> a = np.array([0,0,1,0,1])
    # >>> b = np.unique(a, return_counts=True)
    # b = (array([0, 1]), array([3, 2], dtype=int64))
    # b[1] is the count, use argmax to get the index of majority
    counter = np.unique(data[target_attribute_name], return_counts =
True)[1]
    if len(counter) == 2:
        if counter[0] == counter[1]:
# number of 0 is equal to number of 1 then we get a tie
# randomly choose one value
return random.choice([0,1])
    return np.unique(data[target_attribute_name])[np.argmax(counter)]
```

The main code is in the *question3a.py*.

Every time we have selected a feature and go into the sub tree, the *d* used to grow sub tree is subtracted by 1. And when the $d = 0$, we just return the majority value of the remaining dataset.

The result is:



The largest *d* ideally should be 21 because we have 21 features, but in fact we can see that when $d > 13$, the error will not change, so we can say that the largest depth (full tree) in practice may be 13 or 14. From the graph, we can easily choose the $4 < d < 13$ (where error less than 0.1, accuracy larger than 90%), and the smallest gap between two error in this range is $d = 6$ with $|err_{train} - err_{test}| = 0.0087$, so I choose $d = 6$.

b) We have already know $2^{13} < 10000 < 2^{14}$, so len(S) = 13, to increase the sample size in S, we choose every size to be the average of each two numbers in the S. The final sample size we choose is in the set *sample size* = [1, 2, 3, 4, 6, 9, 14, 19, 29, 39, 58, 78, 117, 156, 234, 312, 468, 625, 937, 1250, 1875, 2500, 3750, 5000, 7500, 10000].

This method is defined in the following function:

```
def fit_ID3_Pruning_Size
```

Comparing to the original fit function, I add the following code:

```
# if number of data is less than or equal to sample size,
# just return with majority Y of data.
if len(data) <= sample_size:
    # If the sub dataset is empty, return the mode target feature value
in the original dataset
    # >>> a = np.array([0,0,1,0,1])
    # >>> b = np.unique(a, return_counts=True)
    # b = (array([0, 1]), array([3, 2], dtype=int64))
    # b[1] is the count, use argmax to get the index of majority
    counter = np.unique(data[target_attribute_name], return_counts =
True)[1]
    if len(counter) == 2:
        if counter[0] == counter[1]:
# number of 0 is equal to number of 1 then we get a tie
# randomly choose one value
```

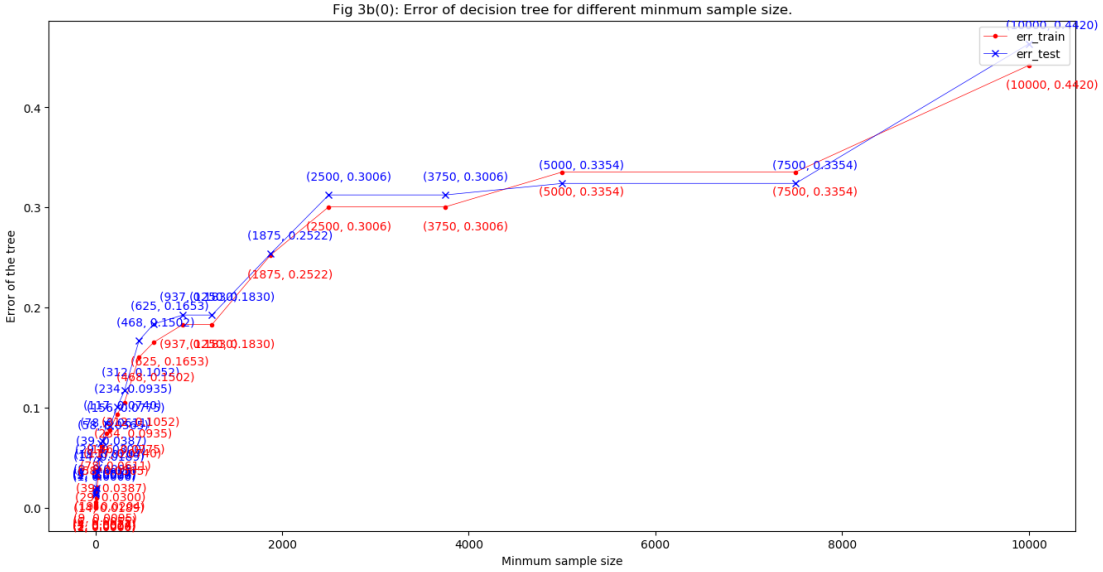
```
return random.choice([0,1])
    return np.unique(data[target_attribute_name])[np.argmax(counter)]
```

The main code is in the *question3b.py*.

Every time we have selected a feature and go into the sub tree, we use the same sample size. And when the remaining number of data is smaller than sample size, we just return the majority value of the remaining dataset.

The result is:

The total result is:



In order to see the detail, I split it to four sub graphs:

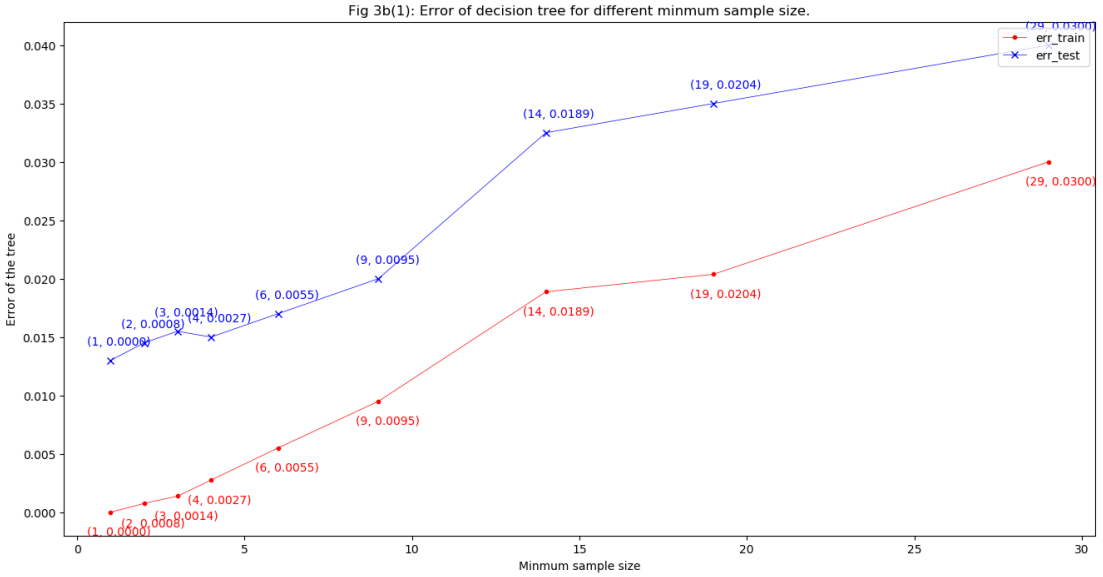


Fig 3b(2): Error of decision tree for different minmum sample size.

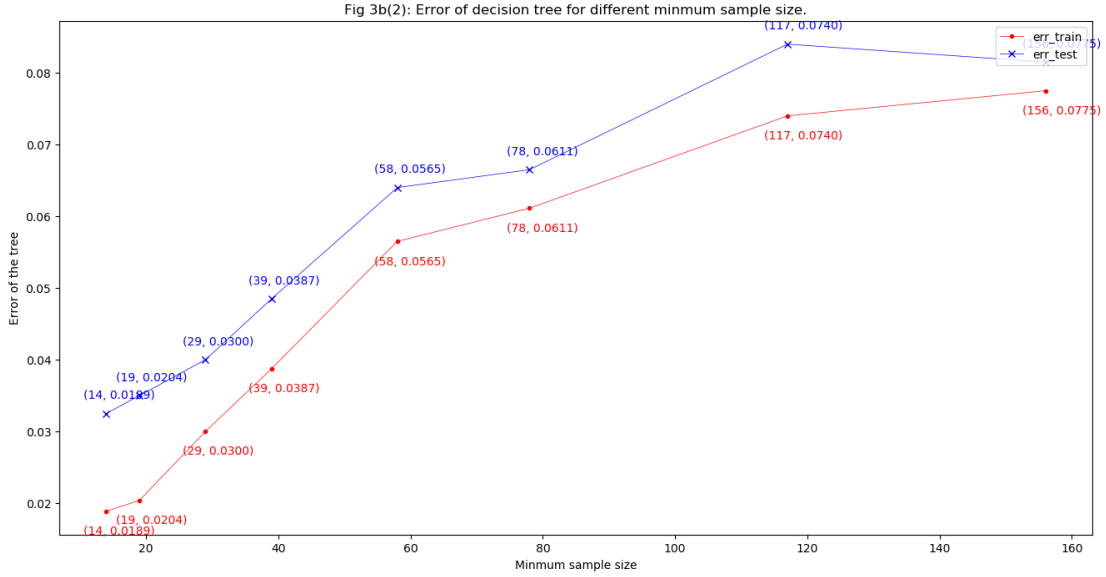


Fig 3b(4): Error of decision tree for different minmum sample size.

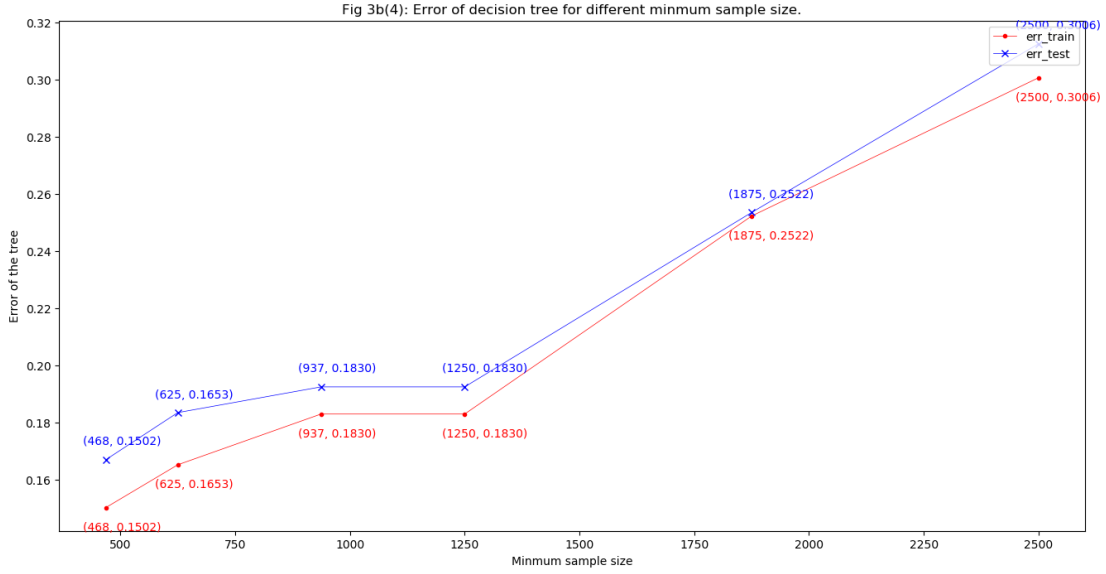
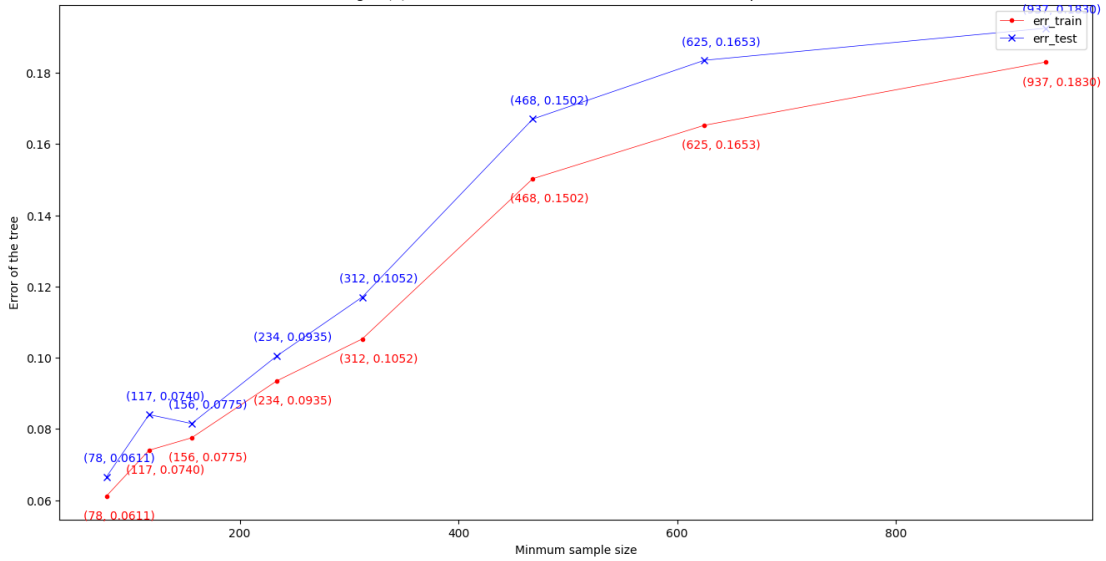


Fig 3b(3): Error of decision tree for different minmum sample size.



The largest sample size ideally should be 1 because we have 21 features and easily to know we use 13 of them. For error less than 0.1, from the graph, we can easily choose the smallest gap between two error in this situation is sample size $= 156$ with $|err_{train} - err_{test}| = 0.004$, so I choose sample size $= 156$ as the threshold.

c) From $\mathrm{P}(T \geq T_0) \approx \int_{T_0}^{\infty} \frac{1}{\sqrt{2\pi}\sqrt{t}} e^{-t/2}\, dt$ we can get a form of T_0 and significance.

T_0	$\mathrm{P}(T \geq T_0)$
0.455	0.50
0.708	0.40
1.323	0.25
2.072	0.15
2.706	0.10
3.841	0.05
5.024	0.025
6.635	0.010
7.879	0.005
10.828	0.001

So I choose the T_0 from this form.

This method is defined in the following function:

```
def fit_ID3_Pruning_Sig
```

The method for calculating T for chi-squared test is:

```
def cal_Norm_Deviation(self, data, split_attribute_name, target_name =
"Y"):
```

The explain is in the annotations and I think them are sufficient enough to understand this function.

Comparing to the original fit function, I add the following code:

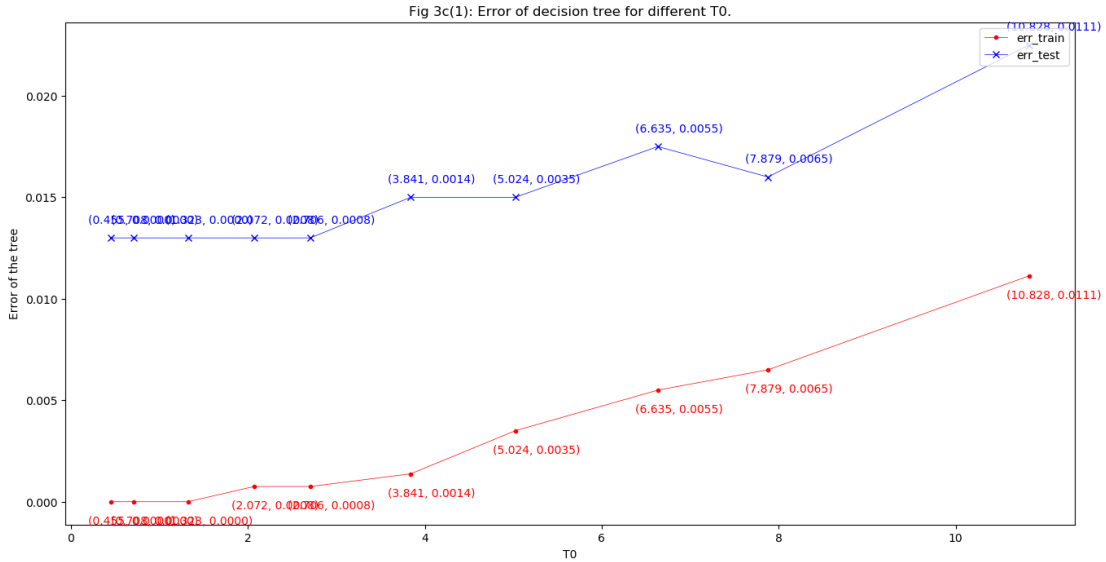
```
# Select the feature which best splits the dataset - max information
gain
# Return the information gain values for the features in the dataset
item_values = [(-1*self.info_Gain(data, feature,
target_attribute_name)) for feature in features]
# get the indices of the features sorted with info_gain from large to
small
best_feature_index = np.argsort(item_values)

best_feature = None
for best_f in best_feature_index:
    if self.cal_Norm_Deviation(data, features[best_f]) >= T_0:
        best_feature = features[best_f]
        # Remove the best feature from the feature space
        features = [i for i in features if i != best_feature]
        break
```

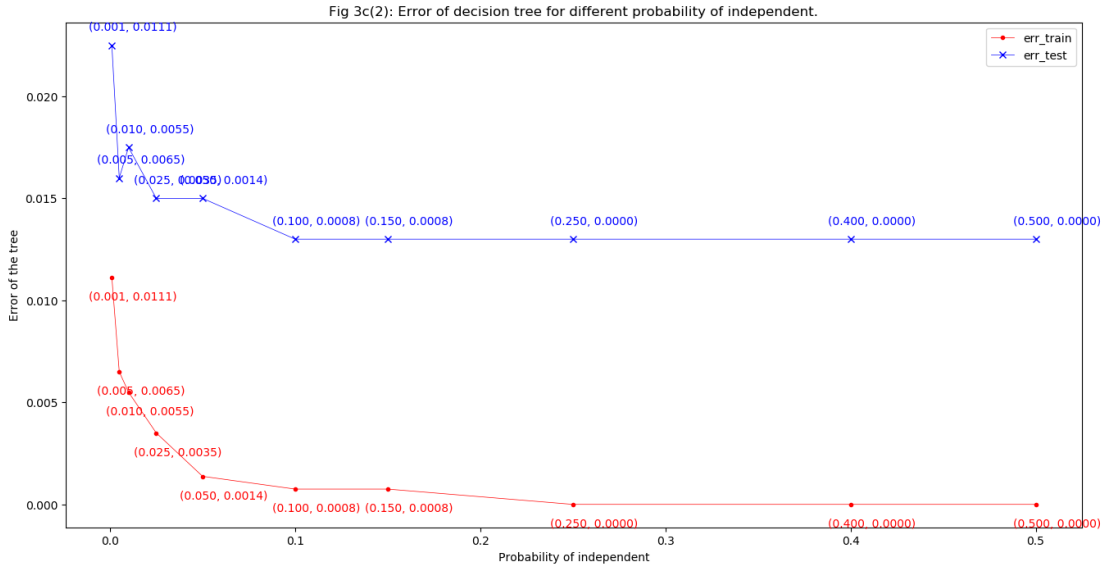
```
    else:
        # Remove the feature with less significant from the feature
space
        features = [i for i in features if i != features[best_f]]
if best_feature == None:
    # if all significant features have been selected, no more feature to
choose
    # return the majority of Y
    counter = np.unique(data[target_attribute_name], return_counts =
True)[1]
    if len(counter) == 2:
        if counter[0] == counter[1]:
# number of 0 is equal to number of 1 then we get a tie
# randomly choose one value
return random.choice([0,1])
    return np.unique(data[target_attribute_name])[np.argmax(counter)]
```

I use T to choose the best feature, if in one step the feature with largest information gain has a T less than T_0, I would abandon that feature and remove it from the features we will use in the future. When the stack of features is empty, we just return the majority as the above approach we have stated.

The main code is in the *question3c.py*.

The result is:





I ignore the point with error larger than 0.1 (accuracy less than 90%), and the smallest gap between two error in this range is $T_0 = 7.879$, $P(T \geq T_0) = 0.005$, so the probability of not independent is larger than 99.5%, with $|err_{train} - err_{test}| = 0.0095$, so I choose $T_0 = 7.879$.

4)

5)

6)

7)

{'origin tree': 5.75, 'pruning tree': 3.63} 100 t=7.879

For the following three questions, we choose m = 10000 and general 200 iterations and calculate the average number of noise node's occurrence as I have done in question 2.

So, we get that the average number of noise nodes in the original decision tree is 5.75

For question 5, the frequency in the pruning with depth tree with d = 6 is 0.02 for each tree.

For question 6, the frequency in the pruning with depth tree with sample size = 156 is 0.135 for each tree.

For question 7, the frequency in the pruning with depth tree with $T_0 = 7.879$ is 3.345 for each tree and the frequency in the pruning with depth tree with $T_0 = 3.841$ is 5.31 for each tree

Below are the results.

Question	Parameter	Average number of irrelevant variables' nodes for each tree	
		Original	Pruning
5: Depth	$d = 6$	5.75	0.02
6: Sample Size	$sample\ size = 156$		0.135
7: Significant ($\chi^2\ test$)	$T_0 = 7.879$		3.345
	Other: $T_0 = 3.841$		5.31

From the results we can know that $T_0 = 7.879$ is indeed better than 3.841 for solving this problem and can better know the independent relation between X an Y. But beyond my expectation, the tree pruned by significant is worse than the tree pruned by depth or sample size. In my view, I think the significant can better reflect the independent relations, but actually it is not. Maybe $T_0 = 7.879$ with a probability of independent 99.5 % is not good enough in this problem and we should try a probability larger than 99.9 % next time to find a better T_0 in order to get the same result as the other two approaches.

But in generally, these three approaches efficiently prune the tree and eliminate the irrelevant features to some extent.