MAGIC Gamma Telescope Report

a. Preprocessing:

Before implementing any DT classifier, we change class label in order to easily handle with calculation for splitting criteria later.

We will use mask to change the value of class label to numerical value:
- Class ‘h’ is changed to class 0
- Class ‘g’ is changed to class 1

```
maskH = (data[:,-1] == 'h')
data[:,-1][maskH] = 0

maskG = (data[:,-1] == 'g')
data[:,-1][maskG] = 1
```

⇨ With this changes on class label, we can calculate number of samples in class ‘g’ or class 1 easily by using data[:,-1].sum()

⇨ From that number of samples in class ‘h’ or class 0 = len(data) – data[:,-1].sum()

⇨ As a result, we can calculate probability for dataset ‘data’ based on number of samples that we got.

⇨ From that, it is trivial to calculate statistics for splitting criteria :

Gini_Index	Entropy

`def Gini(data):`
`  numOfG = data[:,-1].sum()`
`  pG = numOfG / len(data)`
`  pH = 1 - pG`
`  giniG = 1 - pG * pG`
`  giniH = 1 - pH * pH`
`  return giniG + giniH`
	`def ShannoEnt(data):`
	`  numOfG = data[:, -1].sum()`
	`  pG = numOfG / len(data)`
	`  pH = 1 - pG`
	`  if (pH == 0 or pG == 0):`
	`  return 0`
	`  shannonEnt = -pH * np.log2(pH) - pG * np.log2(pG)`
	`  return shannonEnt`

b. Classes definition:

We will choose binary split and define 2 basic classes in this task:

- class **Node**: represents for internal nodes in a DT classifier
  + Attribute 'split_index' and 'split_value' represent for best split at current node
  + Attribute 'left' and 'right' represent for left and right subtree of current node

  + A subtree of a node can be another internal node or a leaf, which is saved as an numpy array of frequencies for class label.

  + Method getMean() is used to merge all descendants of current node into one node that is current node.
- class **DecisionTree**: represents for Decision Tree Classifier

  + Attribute criteria is used to clearly determine type of splitting criteria that current Decision Tree will be built based on

    - 'IG' means Information_Gain()
    - 'GR' means Gain_Ratio()
    - 'VA' means Variance()
    - 'GI' mean Gini_Index()

  + Attribute root is used to store root of current DT

  + Method induction(data, criteria) is used to train DT based on training set 'data' and splitting criteria 'criteria'.

    For all types of splitting criteria, it has the same process of induction except choosing best split

```
def induction(self, data, criteria):

    numOfG = data[:, -1].sum()
    freqClasses = np.array([len(data) - numOfG, numOfG])

    if (data[:, -1].sum() == data.shape[0]):
        return freqClasses

    if criteria == 'GR':
        bestIndex, bestSplitVal = chooseBestSplit_GR(data)
    else:
        bestIndex, bestSplitVal = chooseBestSplit_VA_GI_IG(data, criteria)

    if (bestIndex is None):
        return freqClasses

    newNode = Node()
    newNode.split_index = bestIndex
    newNode.split_value = bestSplitVal

    subDataLeft, subDataRight = BinarySplit(data, bestIndex, bestSplitVal)

    newNode.left = self.induction(subDataLeft, criteria)
    newNode.right = self.induction(subDataRight, criteria)

    return newNode
```

=> We will implement chooseBestSplit() function based on type of splitting criteria

- For criteria ‘VA’, ‘IG’ and ‘GI’, they have same patterns in choosing best split, so we can implement them in the same function:

```
def chooseBestSplit_VA_GI_IG(data, criteria):

    S = BaseS(data, criteria)

    # print("S = ", S)

    n = data.shape[1]; bestS = inf; bestIndex = 0; bestSplitVal = 0

    for featIndex in range(n-1):
        for splitVal in np.percentile(data[:,featIndex], [10, 20, 30, 40, 50, 60, 70, 80, 90]):

            subData0, subData1 = BinarySplit(data, featIndex, splitVal)

            newS = NewS(subData0, subData1, criteria)

            if (newS < bestS):
                bestS = newS
                bestIndex = featIndex
                bestSplitVal = splitVal

    # print("bestS = ", bestS)

    if (S - bestS) < 0:
        return None, 0

    # check valid split again and return bestIndex, bestSplitValue

    subData0, subData1 = BinarySplit(data, bestIndex, bestSplitVal)

    if (len(subData0) == 0 or len(subData1) == 0):
        return None, 0

    return bestIndex, bestSplitVal
```

- In a split, we will choose a feature_index and a splitting_value to evaluate and get the best split.
  But we do not need to go through every unique value in column feature_index because there are a lot of them, which will make the training so much longer.
  Instead, we can use 9 quantiles defined as np.percentile() in the code.
  As a result, training is very fast and the accuracy is high and good enough.

- The rest of work is just defining functions for calculation of splitting criteria:

```
def BaseS(data, criteria):

    if criteria == 'VA':
        S = np.var(data[:,-1])
    elif criteria == 'GI':
        S = Gini(data)
    else:
        S = ShannoEnt(data)

    return S
```

```
def NewS(subData0, subData1, criteria):

    if (len(subData0) == 0 or len(subData1) == 0):
        return inf

    total = len(subData0) + len(subData1)
    p0 = len(subData0) / total
    p1 = len(subData1) / total

    if criteria == 'VA':
        newS = np.var(subData0[:, -1]) * p0 + np.var(subData1[:, -1]) * p1

    elif criteria == 'GI':
        newS = Gini(subData0) * p0 + Gini(subData1) * p1
    else:
        newS = ShannoEnt(subData0) * p0 + ShannoEnt(subData1) * p1

    return newS
```

- For criteria ‘GR’, it is quite different because

$$\text{SplitInfo}(P) = -\sum_{i=1}^{v} \frac{|D_i|}{|D|} \log\left(\frac{|D_i|}{|D|}\right)$$

$$\text{GainRatio} = \text{InfoGain}(P)/\text{SplitInfo}(P)$$

Therefore, we need to calculate SplitInfo(P) for each partition P and get best split by getting max(GainRatio) in all partitions

```
baseS = ShannoEnt(data)

total = len(subData0) + len(subData1)
p0 = len(subData0) / len(data)
p1 = len(subData1) / len(data)

newS = ShannoEnt(subData0) * p0 + ShannoEnt(subData1) * p1

splitInfo = - p0 * np.log2(p0) - p1 * np.log2(p1)

GR = (baseS - newS) / splitInfo

if (GR > bestGR):
    bestGR = GR
    bestIndex = featIndex
    bestSplitVal = splitVal
```

c. Prediction and evaluation:

When we split in training, we use binarySplit(data, split_index, split_value) that:

- Data of left child is data with the mask (data[:, split_index] < split_value)
- Data of right child is data with the mask (data[:, split_index] >= split_value)

```
def BinarySplit(data, featIndex, splitVal):
    subData0 = data[data[:, featIndex] < splitVal]
    subData1 = data[data[:, featIndex] >= splitVal]

    return subData0, subData1
```

Similarly, to predict a sample with a DT, we go from the DT’s root and based on split_index and split_value of the node to go the appropriate child branch.

- If sample[split_index] < split_value, then we go left
- If sample[split_index] >= split_value, then we go right

```
def classify_sample(node, test_sample):
    if isLeaf(node):
        return np.argmax(node)
    C = 0
    if (test_sample[node.split_index] < node.split_value):
        C = classify_sample(node.left, test_sample)
    else:
        C = classify_sample(node.right, test_sample)

    return C
```

⇨ We can classify a dataset by predicting each sample in the dataset (classify_data(node,data))

d. Ensembled DT (M*):

- class Ensembled_DT is based on 3 other DTs that are M_IG, M_GR, M_VA

- predicting of Ensembled_DT is based on voting function.
  We have 3 predicts from M_IG, M_GR, M_VA and 2 classes to predict
  Therefore, at least there is one class that have 2 predicts, which are also 2 votes.
  As a result, we choose to predict with class having major votes.

e. 10-fold cross-validation:

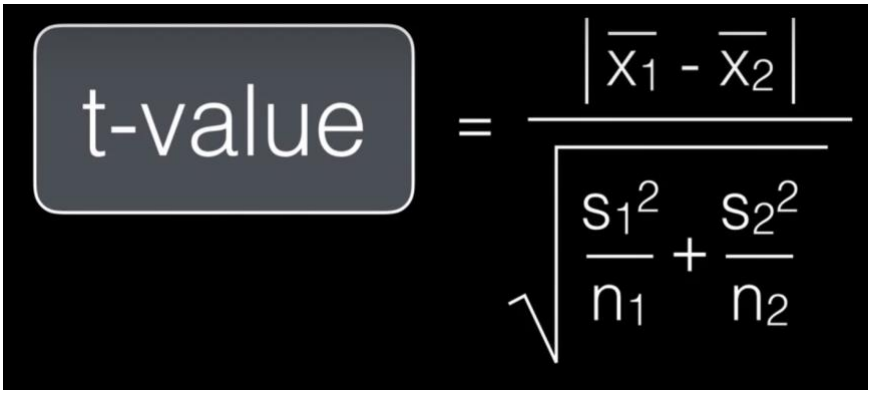
- First of all, we will divide dataset randomly into 10 equal folds

- We will have 10 iterations to evaluate M* and M_GI

- At iteration i, we use mask to get training_set, testing_set.
  After that, we build M* as well as M_GI based on current training_set and get the predicts for testing_set.

  Based on predictions, we can get error rate from each type of DT and assumes:
  + x[i] = error_rate of M*
  + y[i] = error_rate of M_GI

- And based on formula:



We can calculate t-value as follows:

```
x1 = x.mean()
x2 = y.mean()
var1 = x.var()# s1 = x.std()
var2 = y.var()# s2 = y.std()

t_val = np.abs(x1 - x2) / np.sqrt((var1 + var2) / n_folds)
```

- We choose significant level = 0.05 and we have 10 – 1 = 9 degrees of freedom, so by looking on the table, we know the critical_value = 2.26

- Finally, with calculated t_val < critical_value , we can conclude that “There is no statistically significant difference between two classifiers”

f. Post-pruning:

- Firstly, we randomly generate three subsets for training, pruning and evaluation by using numpy.random.permutation for indices and then take each subset respectively with the shuffle indices.

```
l = len(data)

indices = np.random.permutation(data.shape[0])

train_idx, prune_idx, test_idx = indices[:int(l/3)], indices[int(l/3):int(2*l/3)], indices[int(2*l/3):]

train_data, prune_data, test_data = data[train_idx,:], data[prune_idx,:], data[ test_idx, :]
```

- Then we train the M_GI with train_data to build the DT of Gini_Index criteria. After that, we need to get predicts from unpruned M_GI on test_data to calculate accuracy and TP, FP rates before pruning.

```
BEFORE PRUNING :

    Confusion matrix :
     [3508.  565.]
     [ 645. 1622.]
    TP rate :  0.8612816106064326
    FP rate :  0.28451698279664756
    Accuracy :  0.8091482649842271
    Precision :  0.8446905851191909
--------------------------------------------------------------------------------
```

- Now, we start pruning. We will DFS from root and consider pruning action based on type of nodes.

  + If current node is leaf, we do not need to prune anything.
  + If after many splits on the way to current node and the prune_data is empty, then we prune the current node

  + Now we divide prune_data into 2 subsets based on split_index and split_value of current node
  + If any of child nodes is an internal node, we DFS to that node with corresponding subset to keep pruning.

+ If both child nodes are leaves, we need to consider 2 cases: Merge or NotMerge 2 child nodes to current node ?

Therefore, we have to calculate the accuracy in both cases to see which one is better and choose the case with higher accuracy

```
if (isLeaf(node.left) and isLeaf(node.right)):

    predictNoMerge = classify_data(node, prune_data)
    noMergeAcc = calAccuracy(predictNoMerge, prune_data)

    mergeNode = node.getMean()
    predictMerge = classify_data(mergeNode, prune_data)
    mergeAcc = calAccuracy(predictMerge, prune_data)

    if (mergeAcc > noMergeAcc):
        return mergeNode
```

- Finally, we get predicts from pruned M_GI on test_data to calculate accuracy and TP, FP rates after pruning.

```
AFTER PRUNING :

    Confusion matrix :
     [3519.  554.]
     [ 645. 1622.]
    TP rate :  0.8639823226123251
    FP rate :  0.28451698279664756
    Accuracy :  0.8108832807570978
    Precision :  0.8451008645533141
--------------------------------------------------------------------------------
```

⇨ As we can see, the TP rate, accuracy and precision is slightly improved after pruning