- Kendrick Kee
- 7366814

Import Statements

In [1]:

```
import numpy as np
import pandas as pd
import csv
import matplotlib.pyplot as plt
import seaborn as sns
eps = np.finfo(float).eps
from numpy import log2 as log
import sys
import copy
import random
from pprint import pprint
```

In [2]:

```
df = pd.read_csv('secondary_data.csv',sep = ';')
df.head()
```

Out[2]:

	class	cap-diameter	cap-shape	cap-surface	cap-color	does-bruise-or-bleed	gill-attachment	gill-spacing	gill-color	stem-height	...	sten roc
0	p	15.26	x	g	o	f	e	NaN	w	16.95	...	
1	p	16.60	x	g	o	f	e	NaN	w	17.99	...	
2	p	14.07	x	g	o	f	e	NaN	w	17.80	...	
3	p	14.17	f	h	e	f	e	NaN	w	15.77	...	
4	p	14.64	x	h	o	f	e	NaN	w	16.53	...	

5 rows × 21 columns

Findings for current dataset.

Every column aside from the class column will consist of empty/null values. Therefore, dropping rows with empty columns is not a viable strategy in preparing this dataset for ML algorithms. I therefore suggest hot encoding all categorical columns and for binary classified columns such as "has-ring", "does-bruise-or-bleed" and "veil-type" (as the 2 possible values are either 'u' or null) into binary bins of 1 and 0

In [3]:

```
#encode the 3 binary datatype columns
encodingmap = {"class":{'p':1,'e':0},
               "has-ring":{'t':1,'f':0}}
df = df.replace(encodingmap)
df.head()
```

Out[3]:

	class	cap-diameter	cap-shape	cap-surface	cap-color	does-bruise-or-bleed	gill-attachment	gill-spacing	gill-color	stem-height	...	stem-roo
0	1	15.26	x	g	o	f	e	NaN	w	16.95	...	
1	1	16.60	x	g	o	f	e	NaN	w	17.99	...	
2	1	14.07	x	g	o	f	e	NaN	w	17.80	...	
3	1	14.17	f	h	e	f	e	NaN	w	15.77	...	
4	1	14.64	x	h	o	f	e	NaN	w	16.53	...	

5 rows × 21 columns

Find columns with large amounts of empty data

In [4]:

```
df.isnull().sum()
```

Out[4]:

```
class                       0
cap-diameter                0
cap-shape                   0
cap-surface             14120
cap-color                   0
does-bruise-or-bleed        0
gill-attachment          9884
gill-spacing            25063
gill-color                  0
stem-height                 0
stem-width                  0
stem-root               51538
stem-surface            38124
stem-color                  0
veil-type               57892
veil-color              53656
has-ring                    0
ring-type                2471
spore-print-color       54715
habitat                     0
season                      0
dtype: int64
```

Here we can observe that only "gill-attachment" and "ring-type" has a reasonable amount of null values to be replaced while the other columns has null values that consist of more than half the entire row count.

Therefore, I will be dropping them as they will not be good predictors of the class type as there is insufficient information provided.

In [5]:

```
#dropping all columns with high null counts.
for col_names in list(df.columns):
    if df[col_names].isnull().sum() > (61069*0.2): #if column has more than 20% miss
        df = df.drop([col_names],axis=1)
df.head()
```

Out[5]:

	class	cap-diameter	cap-shape	cap-color	does-bruise-or-bleed	gill-attachment	gill-color	stem-height	stem-width	stem-color	has-ring	ring-type
0	1	15.26	x	o	f	e	w	16.95	17.09	w	1	g
1	1	16.60	x	o	f	e	w	17.99	18.19	w	1	g
2	1	14.07	x	o	f	e	w	17.80	17.74	w	1	g
3	1	14.17	f	e	f	e	w	15.77	15.98	w	1	p
4	1	14.64	x	o	f	e	w	16.53	17.20	w	1	p

In [6]:

```
#get missing values count again
df.isnull().sum()
```

Out[6]:

```
class                      0
cap-diameter               0
cap-shape                  0
cap-color                  0
does-bruise-or-bleed       0
gill-attachment         9884
gill-color                 0
stem-height                0
stem-width                 0
stem-color                 0
has-ring                   0
ring-type               2471
habitat                    0
season                     0
dtype: int64
```

In [7]:

```
#fill the missing values with the mode of the column
for colname in ['gill-attachment','ring-type']:
    temp = df[colname].mode()[0]
    df[colname].fillna(temp,inplace=True)
df.isnull().sum()#get the null count again
```

Out[7]:

```
class                   0
cap-diameter            0
cap-shape               0
cap-color               0
does-bruise-or-bleed    0
gill-attachment         0
gill-color              0
stem-height             0
stem-width              0
stem-color              0
has-ring                0
ring-type               0
habitat                 0
season                  0
dtype: int64
```

Findings

We can now observe that the dataset no longer has null values

In [8]:

```
#view object types to check if further encoding is needed
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 61069 entries, 0 to 61068
Data columns (total 14 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   class                 61069 non-null  int64  
 1   cap-diameter          61069 non-null  float64
 2   cap-shape             61069 non-null  object 
 3   cap-color             61069 non-null  object 
 4   does-bruise-or-bleed  61069 non-null  object 
 5   gill-attachment       61069 non-null  object 
 6   gill-color            61069 non-null  object 
 7   stem-height           61069 non-null  float64
 8   stem-width            61069 non-null  float64
 9   stem-color            61069 non-null  object 
 10  has-ring              61069 non-null  int64  
 11  ring-type             61069 non-null  object 
 12  habitat               61069 non-null  object 
 13  season                61069 non-null  object 
dtypes: float64(3), int64(2), object(9)
memory usage: 6.5+ MB
```

In [9]:

```
#apply onehot encoding for categorical data which for this case would be columns wit
#print(pd.get_dummies(df["cap-shape"],prefix='cap-shape'))
for key,value in dict(df.dtypes).items():
    if value == 'object':
        dummy = pd.get_dummies(df[key],prefix=key)
        df = df.join(dummy)
        df = df.drop(columns = key)

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 61069 entries, 0 to 61068
Data columns (total 78 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   class                   61069 non-null  int64  
 1   cap-diameter            61069 non-null  float64
 2   stem-height             61069 non-null  float64
 3   stem-width              61069 non-null  float64
 4   has-ring                61069 non-null  int64  
 5   cap-shape_b             61069 non-null  uint8  
 6   cap-shape_c             61069 non-null  uint8  
 7   cap-shape_f             61069 non-null  uint8  
 8   cap-shape_o             61069 non-null  uint8  
 9   cap-shape_p             61069 non-null  uint8  
 10  cap-shape_s             61069 non-null  uint8  
 11  cap-shape_x             61069 non-null  uint8  
 12  cap-color_b             61069 non-null  uint8  
 13  cap-color_e             61069 non-null  uint8  
 14  cap-color_g             61069 non-null  uint8  
 15  cap-color_k             61069 non-null  uint8  
 16  cap-color_l             61069 non-null  uint8  
 17  cap-color_n             61069 non-null  uint8  
 18  cap-color_o             61069 non-null  uint8  
 19  cap-color_p             61069 non-null  uint8  
 20  cap-color_r             61069 non-null  uint8  
 21  cap-color_u             61069 non-null  uint8  
 22  cap-color_w             61069 non-null  uint8  
 23  cap-color_y             61069 non-null  uint8  
 24  does-bruise-or-bleed_f  61069 non-null  uint8  
 25  does-bruise-or-bleed_t  61069 non-null  uint8  
 26  gill-attachment_a       61069 non-null  uint8  
 27  gill-attachment_d       61069 non-null  uint8  
 28  gill-attachment_e       61069 non-null  uint8  
 29  gill-attachment_f       61069 non-null  uint8  
 30  gill-attachment_p       61069 non-null  uint8  
 31  gill-attachment_s       61069 non-null  uint8  
 32  gill-attachment_x       61069 non-null  uint8  
 33  gill-color_b            61069 non-null  uint8  
 34  gill-color_e            61069 non-null  uint8  
 35  gill-color_f            61069 non-null  uint8  
 36  gill-color_g            61069 non-null  uint8  
 37  gill-color_k            61069 non-null  uint8  
 38  gill-color_n            61069 non-null  uint8  
 39  gill-color_o            61069 non-null  uint8  
 40  gill-color_p            61069 non-null  uint8  
 41  gill-color_r            61069 non-null  uint8  
 42  gill-color_u            61069 non-null  uint8  
 43  gill-color_w            61069 non-null  uint8  
```

```
 44  gill-color_y           61069 non-null  uint8
 45  stem-color_b           61069 non-null  uint8
 46  stem-color_e           61069 non-null  uint8
 47  stem-color_f           61069 non-null  uint8
 48  stem-color_g           61069 non-null  uint8
 49  stem-color_k           61069 non-null  uint8
 50  stem-color_l           61069 non-null  uint8
 51  stem-color_n           61069 non-null  uint8
 52  stem-color_o           61069 non-null  uint8
 53  stem-color_p           61069 non-null  uint8
 54  stem-color_r           61069 non-null  uint8
 55  stem-color_u           61069 non-null  uint8
 56  stem-color_w           61069 non-null  uint8
 57  stem-color_y           61069 non-null  uint8
 58  ring-type_e            61069 non-null  uint8
 59  ring-type_f            61069 non-null  uint8
 60  ring-type_g            61069 non-null  uint8
 61  ring-type_l            61069 non-null  uint8
 62  ring-type_m            61069 non-null  uint8
 63  ring-type_p            61069 non-null  uint8
 64  ring-type_r            61069 non-null  uint8
 65  ring-type_z            61069 non-null  uint8
 66  habitat_d              61069 non-null  uint8
 67  habitat_g              61069 non-null  uint8
 68  habitat_h              61069 non-null  uint8
 69  habitat_l              61069 non-null  uint8
 70  habitat_m              61069 non-null  uint8
 71  habitat_p              61069 non-null  uint8
 72  habitat_u              61069 non-null  uint8
 73  habitat_w              61069 non-null  uint8
 74  season_a               61069 non-null  uint8
 75  season_s               61069 non-null  uint8
 76  season_u               61069 non-null  uint8
 77  season_w               61069 non-null  uint8
dtypes: float64(3), int64(2), uint8(73)
memory usage: 6.6 MB
```

As observed, all columns are all now numerically represented.

After applying one hot encoding on the object rows, we now no longer have object data types

Now I will apply correlation to the class to view any significant features to use for model training as to improve model's accuracy and reduce the feature space and computational complexity.

In [10]:

```
pre_processed_df = df
correlation_matrix = df.corr()
for col_name,p_value in dict(correlation_matrix["class"].sort_values()).items():
    if p_value < 0.1 and p_value > -0.1: #selects features with a significant amount
        pre_processed_df = pre_processed_df.drop(columns=col_name)#drop columns of r

c = pre_processed_df.pop('class')
pre_processed_df.insert(18,'class',c)
pre_processed_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 61069 entries, 0 to 61068
Data columns (total 19 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   cap-diameter        61069 non-null  float64
 1   stem-height         61069 non-null  float64
 2   stem-width          61069 non-null  float64
 3   cap-shape_b         61069 non-null  uint8  
 4   cap-shape_o         61069 non-null  uint8  
 5   cap-color_b         61069 non-null  uint8  
 6   cap-color_e         61069 non-null  uint8  
 7   cap-color_n         61069 non-null  uint8  
 8   cap-color_r         61069 non-null  uint8  
 9   gill-attachment_a   61069 non-null  uint8  
 10  gill-attachment_e   61069 non-null  uint8  
 11  gill-attachment_p   61069 non-null  uint8  
 12  gill-color_n        61069 non-null  uint8  
 13  gill-color_w        61069 non-null  uint8  
 14  stem-color_f        61069 non-null  uint8  
 15  stem-color_w        61069 non-null  uint8  
 16  ring-type_z         61069 non-null  uint8  
 17  habitat_g           61069 non-null  uint8  
 18  class               61069 non-null  int64  
dtypes: float64(3), int64(1), uint8(15)
memory usage: 2.7 MB
```

In [11]:

```
pre_processed_df.describe()
```

Out[11]:

	cap-diameter	stem-height	stem-width	cap-shape_b	cap-shape_o	cap-color_b	c
count	61069.000000	61069.000000	61069.000000	61069.000000	61069.000000	61069.000000	61(
mean	6.733854	6.581538	12.149410	0.093239	0.056657	0.020141	
std	5.264845	3.370017	10.035955	0.290769	0.231188	0.140484	
min	0.380000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	3.480000	4.640000	5.210000	0.000000	0.000000	0.000000	
50%	5.860000	5.950000	10.190000	0.000000	0.000000	0.000000	
75%	8.540000	7.740000	16.570000	0.000000	0.000000	0.000000	
max	62.340000	33.920000	103.910000	1.000000	1.000000	1.000000	

As obeserved the feature space is not reduced to 18 columns and is now ready for training

Classification Helper Functions for Decision Tree Algorithm

Classification with Info gain and gini split helper functions

In [14]:

```
# Calculate Entropy of dataset
def findEntropy(df):
    Class = df.keys()[-1]
    entropy = 0
    values = df[Class].unique()

    for value in values:
        fraction = df[Class].value_counts()[value] / len(df[Class])
        entropy += -fraction * np.log2(fraction)
    return entropy
```

In [15]:

```
# Calculate Entropy by attribute
def findEntropyAttribute(df,attribute):
    Class = df.keys()[-1]

    target_variables = df[Class].unique()
    variables = df[attribute].unique()

    entropy2 = 0
    for variable in variables:
        entropy = 0
        for target_variable in target_variables:
            num = len(df[attribute][df[attribute] == variable][df[Class] == target_v
            den = len(df[attribute][df[attribute] == variable])
            fraction = num / (den+eps)
            entropy += -fraction * log(fraction+eps)
        fraction2 = den / len(df)
        entropy2 += -fraction2 * entropy

    return abs(entropy2)
```

In [16]:

```
# Calculate information gain and return the best splitting node (feature)
def infoGain(df):
    IG = []
    for key in df.keys()[:-1]:
        IG.append(findEntropy(df) - findEntropyAttribute(df,key))

    return df.keys()[:-1][np.argmax(IG)]
```

In [17]:

```
def giniImpurity2(valueCounts):
    n = valueCounts.sum()
    p_sum = 0
    for key in valueCounts.keys():
        p_sum = p_sum  +  (valueCounts[key] / n ) * (valueCounts[key] / n )
        gini = 1 - p_sum

    return gini
```

In [18]:

```
# Calculating  gini impurity for the attiributes
def giniSplitAtt2(df, attName):
    attValues = df[attName].value_counts()
    gini_A = 0
    for key in attValues.keys():
        dfKey = df[className][df[attName] == key].value_counts()
        numOfKey = attValues[key]
        n = df.shape[0]
        gini_A = gini_A + (( numOfKey / n) * giniImpurity2(dfKey))

    return gini_A
```

In [19]:

```
def getSubtable(df, node, value):
  return df[df[node] == value].reset_index(drop=True)
```

In [20]:

```
def giniIndex2(df, attributeNames):
    giniAttribute = {}
    minValue = sys.maxsize
    for key in attributeNames:
        giniAttribute[key] = giniSplitAtt2(df, key)
        if giniAttribute[key] < minValue:
            minValue = giniAttribute[key]
            selectedAttribute = key
    minValue = min(giniAttribute.values())
    return selectedAttribute
```

Tree Induction Function

In [21]:

```
def buildTree(df,model,tree=None):
    Class = df.keys()[-1]
    if model == 'infoGain':
        #print("building infoGain")
        node = infoGain(df)
    else:
        #print("building gini Index")
        node = giniIndex2(df, attName)
    attValueBT = np.unique(df[node])
    #Create an empty dictionary to create tree
    if tree is None:
        tree = {}
        tree[node] = {}
    #We make loop to construct a tree by calling this function recursively.
    #In this we check if the subset is pure and stops if it is pure.

    for value in attValueBT:
        #print('value (buildTree): ', value)
        subtable = getSubtable(df,node,value)

        clValue,counts = np.unique(subtable[className],return_counts=True)

        if len(counts) == 1:
            tree[node][value] = clValue[0]
        else:
            tree[node][value] = buildTree(subtable, model)
    return tree
```

Creating Decision Tree

Helper function to get training and testing split

In [12]:

```
def train_test_split(df, test_size):

    if isinstance(test_size, float):
        test_size = round(test_size * len(df))

    indices = df.index.tolist()
    test_indices = random.sample(population=indices, k=test_size)

    test_df = df.loc[test_indices]
    train_df = df.drop(test_indices)

    return train_df, test_df
```

In [ ]:

```
train_df, test_df = train_test_split(pre_processed_df, test_size=0.4)#get 60% traini
```

Splitting the data 60/40

In [ ]:

```
test_df, valid_df = train_test_split(test_df, test_size=0.5)#get 20% 20% test and va
```

Spliting 40% of the data allocated to testing into half for a 20/20 test and validation split

In [91]:

```
attName = list(train_df.columns)[:-1]
className = 'class'
```

Building the Information Gain and Gini Index Tree

In [93]:

```
info_gain_tree = buildTree(train_df,"infoGain")
```

In [95]:

```
gini_index_tree = buildTree(train_df,"giniIndex")
```

Accuracy of Infomation Gain Split Criteria Decision Tree Pre-Pruning

Helper Function for Decision Tree Classification and Accuracy Report

In [26]:

```
def accuracy_of_the_tree(instance, tree, default=None):
    attribute = list(tree.keys())[0]
    if instance[attribute] in tree[attribute].keys():
        result = tree[attribute][instance[attribute]]
        if isinstance(result, dict):
            return accuracy_of_the_tree(instance, result)
        else:
            return result
    else:
        return default
```

In [27]:

```
def _unique(seq, return_counts=False, id=None):

    found = set()
    if id is None:
        for x in seq:
            found.add(x)

    else:
        for x in seq:
            x = id(x)
            if x not in found:
                found.add(x)
    found = list(found)
    counts = [seq.count(0),seq.count(1)]
    if return_counts:
        return found,counts
    else:
        return found

def _sum(data):
    sum = 0
    for i in data:
        sum = sum + i
    return sum
```

Accuracy of Information Gain Split Criteria Decision Tree Pre Pruning

Training set

In [126]:

```
temptrain1 = pd.DataFrame()
temptrain1['predicted'] = train_df.apply(accuracy_of_the_tree, axis=1, args=(info_ga
print( 'Accuracy with info gain ' +  (str( sum(train_df['class']==temptrain1['predic
```

```
Accuracy with info gain 63.55448814169919
```

Testing set

In [127]:

```
temptest1 = pd.DataFrame()
temptest1['predicted'] = test_df.apply(accuracy_of_the_tree, axis=1, args=(info_gair
print( 'Accuracy with info gain ' +  (str( sum(test_df['class']==temptest1['predicte
```

```
Accuracy with info gain 63.88570492877027
```

Findings

We can observe that the accuracy of the training set and testing set are hovering around 63% accuracy, therefore we can conclude the no overfitting has occured and the model has achieved a decent level of generalisation.

Accuracy of Gini Index Split Criteria Decision Tree Pre Pruning

Training set

In [128]:

```
temptrain1 = pd.DataFrame()
temptrain1['predicted2'] = train_df.apply(accuracy_of_the_tree, axis=1, args=(gini_i
print( 'Accuracy with gini index ' +  (str( sum(train_df['class']==temptrain1['predi
```

```
Accuracy with gini index 63.63090527005267
```

Testing set

In [129]:

```
temptest1 = pd.DataFrame()
temptest1['predicted2'] = test_df.apply(accuracy_of_the_tree, axis=1, args=(gini_inc
print( 'Accuracy with gini index ' +  (str( sum(test_df['class']==temptest1['predict
```

```
Accuracy with gini index 63.91026690682823
```

Findings

We can observe that the accuracy of both the training and testing data set are hover around 63% as well. Therefore, the tree using the gini index split criteria does not have an overfitting issues and has obtained generalisation

Final conclusion of both models

Both split criteria of Information Gain and Gini Index Yielded similar accuracies of 63% with no signs of overfitting and displayed the ability to generalise for unseen data. We will therefore attempt to tune and improve its performance but Post Pruning the decision tree in hopes of reducing computational complexity and model accuracy.

Post Pruning Helper Funtion

In [199]:

```
def preorder (temptree, number):
    if isinstance(temptree, dict):
        attribute = list(temptree.keys())[0]
        if temptree[attribute]['number'] == number:
            if(temptree[attribute][0]!=0 and temptree[attribute][0]!=1):
                temp_tree = temptree[attribute][0]
                if isinstance(temp_tree, dict):
                    temp_attribute = list(temp_tree.keys())[0]
                    temptree[attribute][0] = temp_tree[temp_attribute]['best_class']
            elif(temptree[attribute][1]!=0 and temptree[attribute][1]!=1):
                temp_tree = temptree[attribute][1]
                if isinstance(temp_tree, dict):
                    temp_attribute = list(temp_tree.keys())[0]
                    temptree[attribute][1] = temp_tree[temp_attribute]['best_class']
        else:
            left = temptree[attribute][0]
            right = temptree[attribute][1]
            preorder(left, number)
            preorder(right,number )
    return temptree
```

In [254]:

```
def count_number_of_non_leaf_nodes(tree):
    if isinstance(tree, dict):
        attribute = list(tree.keys())[0]
        left = tree[attribute][0]
        print(left)
        right = tree[attribute][1]
        print(right)
        return (1 + count_number_of_non_leaf_nodes(left) +  count_number_of_non_leaf
    else:
        return 0;
```

In [201]:

```
def post_prune(L, K, tree):
    best_tree = tree
    for i in range(1, L+1) :
        temp_tree = copy.deepcopy(best_tree)
        M = randint(1, K);
        for j in range(1, M+1):
            n = count_number_of_non_leaf_nodes(temp_tree)
            if n> 0:
                P = randint(1,n)
            else:
                P = 0
            preorder(temp_tree, P)
        test_data['accuracyBeforePruning'] = test_data.apply(accuracy_of_the_tree, a
        accuracyBeforePruning = str( sum(test_data['Class']==test_data['accuracyBefo
        test_data['accuracy_after_pruning'] = test_data.apply(accuracy_of_the_tree,
        accuracy_after_pruning = str( sum(test_data['Class']==test_data['accuracy_af
        if accuracy_after_pruning >= accuracyBeforePruning:
            best_tree = temp_tree
    return best_tree
```

Accuracy Report for post pruning

I am unable to proceed as the functions created were generating and error due to the multi way split nature of my tree induction function