Data Set #1

The file for this algorithm is alg_1.ipynb

Open Files

```
train_file_x_loc = 'data_1\\train\X_train.xlsx'
train_file_y_loc = 'data_1\\train\y_train.xlsx'
test_file_loc = 'data_1\Dataset1_test\X_test.xlsx'

train_data_x = pd.read_excel(train_file_x_loc)
train_data_y = pd.read_excel(train_file_y_loc)
test_data = pd.read_excel(test_file_loc)
```

[2] ✓ 1.6s

Convert to numpy array

```
tr_x = train_data_x.values
tr_y = [i[0] for i in train_data_y.values]
test = test_data.values
num_features = len(tr_x[0])
```

[3] ✓ 0.6s

First we open the files and convert them to numpy arrays. tr_x indicates the value of the features of the training data, tr_y indicates the values of the target feature, test indicates the test data, num_features indicates the number of features (excluding the target feature).

Imputation

```
for cur_f in range(len(tr_x[0])):

    temp = [x for x in tr_x[:,cur_f] if np.isnan(x) == False]
    feature_mean = np.mean(temp)

    for i in range(len(tr_x)):
        if np.isnan(tr_x[i][cur_f]):
            tr_x[i][cur_f] = feature_mean
```

We impute both the training data using the following method; Each empty value is replaced with the average of all the values (excluding any empty values) associated with the feature of the empty value.

Entropy calculating function

```
labels = [3,4,5,6,7,8]

count = np.bincount(tr_y)[3:]

# Take bin count of each label value as input
def get_entropy(count):
    num_data = np.sum(count)
    total = 0

    for i in count:
        prob = i / num_data
        if prob == 0:
            continue
        total += prob * np.log2(prob)

    return -total
```

✓ 0.6s

This is the function to calculate the entropy of a given data set.

The formula is as follows.

$$H = -\sum_{i} p_i(\log_2 p_i)$$

In this data set, the target features have a value of either 3, 4, 5, 6, 7, or 8. Our function takes an array count where the 1^{st} entry corresponds to the amount of entry that has class “3”, the 2^{nd} entry corresponds to class “4”, and so on.

In our implementation, we ignore any class with zero probability (this means in that data set, there are 0 entries that has that particular target feature value).

Class for the decision tree

```python
class Node:
    def __init__(self):
        self.left = None
        self.right = None
        self.data = None    # Feature values, including labels
        self.f_in = None    # The index of the traversal criteria
        self.thresh = None  # The value threshold of the traversal criteria
        self.label = None   # Label with the max amount

    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
        self.f_in = None
        self.thresh = None
        self.label = None
```

[27] ✓ 0.6s

This is a class that is going to help me implement my decision tree. Each class Node will respond to a node in the decision tree and has several variables.

- left and right corresponds to the left and right child node.

- data corresponds to the sub data set that is in the current node, including the “class” target feature.
- f_in and thresh is related to the criteria used for splitting our decision tree and classification. For each node, if f_in <= thresh, we will go to the left child node; otherwise we will go to the right child node.
- label corresponds to the mode of the “class” target feature in the sub data set related to that node.

Create Decision Tree:

```
cur_data = tr_x
temp = []

# Append target feature to cur_data
for i in range(len(cur_data)):
    temp.append( np.append(cur_data[i],int(tr_y[i])) )
cur_data = np.array(temp)

root = Node(cur_data)
cur_node = root

q = queue.Queue() # A queue for processing nodes of the decision tree
q.put(cur_node)
```

From here I begin building my decision tree. First we create our root node of our decision tree. This node will contain all of the data, including the class target feature values.

I will use a queue data structure for our tree building process. The queue will contain a set of nodes to be processed. Each time a node is split into two, it will put their left and right child into the queue (unless certain criteria are met).

```python
while q.empty() == False:

    cur_node = q.get()
    cur_data = cur_node.data

    # If data length is 0, continue
    if len(cur_data) == 0:
        continue

    cur_label = [int(temp) for temp in cur_data[:,num_features]]
    temp_count = np.bincount(cur_label)[3:]

    # Assign the selected label to the current node
    cur_node.label = np.argmax(temp_count) + 3

    # If all data has the same label, stop
    areAllLabelSame = False

    for temp in temp_count:
        if temp == sum(temp_count):
            areAllLabelSame = True
            break
    if areAllLabelSame:
        continue
```

We won’t split our current node if either, there are 0 entries in the data set, or all entries already have the same class value.

We also assign the ‘label’ of our current node which is to provide a classification result. This means that this variable is only relevant for leaf nodes, but I did it for each node anyway since it is not computationally expensive to do.

```python
# Go through each feature, sort the values
# i = current feature index
for i in range(num_features):  # range(num_features)
    cur_feat_val = np.sort(cur_data[:,i])

    # Iterate through each sorted value
    for j in range(len(cur_feat_val) - 1):

        # Get every unique feature values
        if (cur_feat_val[j] !=  cur_feat_val[j+1]):
            cur_thresh = cur_feat_val[j]

            # Split data into two based on cur_thresh, get two labels
            label1 = []
            label2 = []
            for temp_dat in cur_data:
                if temp_dat[i] <= cur_thresh:
                    label1.append(int(temp_dat[num_features])) # Append the label
                else:
                    label2.append(int(temp_dat[num_features]))

            label1 = np.array(label1)
            label2 = np.array(label2)

            count1 = np.bincount(label1)[3:]
            count2 = np.bincount(label2)[3:]

            num_labels_1 = np.sum(count1)
            num_labels_2 = np.sum(count2)

            w_avg_ent = (num_labels_1 / (num_labels_1 + num_labels_2)) * get_entropy(count1) + (num_labels_2 / (num_labels_1 + num_labels_2)) * get_entropy(count2)

            # Maximizing IG == minimizing w_avg_ent

            if w_avg_ent < min_ent:
                min_ent = w_avg_ent
                split_feat_in = i
                split_thresh = cur_thresh
```

We will choose the feature and the threshold that maximizes the information gain. Note that this is equivalent to minimizing the weighted average entropy, since the information gain will be constant on each splitting decision.

- Information Gain (IG)

$$IG(D,A) = H(D) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v)$$

To do this we consider each unique value of each feature as a threshold of that feature. Then we divide the classes into two subsets based on that threshold. We count the number of times each class appear in each subsets, which then we can use to calculate our entropy and therefore our weighted entropy.

```python
# Split data into two
data_left = []
data_right = []

# For each data put in either left / right
for temp_dat in cur_data:
    if temp_dat[split_feat_in] <= split_thresh:
        data_left.append(temp_dat)
    else:
        data_right.append(temp_dat)

data_left = np.array(data_left)
data_right = np.array(data_right)


# Update data structure
cur_node.thresh = split_thresh
cur_node.f_in = split_feat_in

cur_node.left = Node(data_left)
cur_node.right = Node(data_right)

q.put(cur_node.left)
q.put(cur_node.right)
```

After we go through each possible split criteria, we will obtain a split criteria that maximizes the information gain. We divide our data set into two and create a left and right child node based on our two data sets. Both of the child nodes will then be put into the queue to be processed later.

```python
for cur_f in range(len(test[0])):

    temp = [x for x in test[:,cur_f] if np.isnan(x) == False]
    feature_mean = np.mean(temp)

    for i in range(len(test)):
        if np.isnan(test[i][cur_f]):
            test[i][cur_f] = feature_mean
```

✓ 0.1s

Before we test our data, we impute our data sets using the same method we use to impute our training data.

```python
i = 0
pred_label = []

for item in test:
    cur_node = root
    while True:
        if cur_node.left == None or cur_node.right == None:
            pred_label.append(cur_node.label)
            break

        if item[cur_node.f_in] <= cur_node.thresh:
            cur_node = cur_node.left
        else:
            cur_node = cur_node.right

    i += 1

pred_label = np.array(pred_label)

d = pd.DataFrame({'class': pred_label})
df = pd.DataFrame(data=d)
df.to_excel('out_1.xlsx',index=False)
print(d)
```
✓ 0.4s

To get a classification, for each data entry we traverse through the tree. At each node, if the value of a particular feature is less than the threshold, it will go to the left child; otherwise it will go to the right child. A classification will be decided when a leaf node is reached, which then the label variable value - which is the mode of the class of the data set associated with the node - will be selected as the classification.
The output of our test data will be exported to the file “out_1.xlsx”

Data Set #2

The file for this algorithm is alg_2.ipynb

Unfortunately due to time constraint, I wasn't able to finish this one in time. However, this is the method I'd use in my implementation.

I used the demo code as a reference to pre-process my data. For each entry, it will extract words with some constraints (No conjunction, no symbols and numbers, the words are lemmatized, etc). Then I will tokenize our data to obtain a numeric set of each features for each data entry.

Since the size of the training data set is too big, I will use stratified sampling to sample a portion of the data. Afterwards I will use the same method on Data set #1 to build my decision tree.