Project 4 Write-Up

To begin with, my choice of machine learning algorithm to implement was decision tree learning. The reason I choose the said algorithm was that I figured results I would attain with discrete values would be much clearer than ones I would get with continuous values. As usual, my main source of reference was pseudo-code in the textbook (AIMA 18.3).

Since my program has a fair number of functions, I think it would be helpful to the reviewer to know how my though process went and, by that extension, to know how pieces or, in this particular instance, functions all come together. To that end, I have made a table, shown below, delineating the order of major function calls. I hope it gives the reviewer a bird-eye-view of the whole program.

Order	Function Name	Brief Description
1	decision_tree_learning	decision tree learning algorithm
2	split_on_which_attribute	Decides which attribute to split on
3	information_gain	Calculate entropy of each attribute in a subset
4	entropy	Calculate entropy (formula for entropy)
5	calculate_info_gain_for_each_attribute	Calculate information gain of each attribute (parent – child)
6	partition	This is where a node splits examples into subsets
7	exclude_previous_attribute	Disregard the attribute that was used previously to split examples
8	plurality_value	Base case for 1
9	are_examples_pure	Base case for 1

That said, I have made my program to print steps the program is taking in order to produce the result. For example, it shows entropy value of each subset and which attribute it decided to use to split examples in a parent node. Therefore, output of the program might be longer than the reviewer of the program might have expected, but I have chosen to implement such feature, believing that it would give much clearer insight.

To dive into specific parts of the program, I first began the project by starting with the simplest function first – pluarality_value(). It was a great first starting point since it is used for two base cases of decision_tree_learning algorithm. The function is used to select the most common output value among a set of examples, breaking ties randomly. As shown below, it first finds unique targets in the examples and stores the number of occurrences of those targets in a dictionary. Lastly, it finds the most commonly occurring target using ‘max()’ function.

```
def plurality_value(examples):
    '''select the most common output value
    among a set of examples, breaking ties randomly'''
    unique_output_values = get_unique_classes(examples)

    output_value_dictionary = {unique_value: 0 for unique_value in unique_output_values}

    for each_unique_output_values in unique_output_values:
        count = 0
        for row in examples:
            if row[-1] == each_unique_output_values:
                count += 1
        output_value_dictionary[each_unique_output_values] = count

    most_common_value = max(output_value_dictionary, key=(lambda key: output_value_dictionary[key]))
    return most_common_value
```

Next, I made a few trivial functions that are fairly self-explanatory. I believe reading function docstrings suffices in understanding purpose of these functions. As a sidenote, are_examples_pure() is used for base case of our decision_tree_algorithm().

```
def get_unique_classes(examples):
    '''getting unique values from a list of example(s)'''
    unique_value_list = []
    for row in examples:
        unique_value_list.append(row[-1])
    return set(unique_value_list)


def get_subsets_of_examples(attribute, examples):
    '''Examples that are divided into subsets based on values'''
    values = []
    for row in examples:
        values.append(row[attribute])
    return set(values)


def are_examples_pure(examples):
    '''base case to see if a set or a subset is pure'''
    count_of_class = []
    for each_example in examples:
        count_of_class.append(each_example[-1])
    if len(set(count_of_class)) == 1:
        return True
    else:
        return False
```

Entropy() function finds the probability of examples with certain targets occurring within a subset. It is also used to find entropy of a parent node. Just before the function returns a value, it uses entropy formula in order to produce an output.

```
def entropy(examples):
    '''get entropy'''
    unique_classes = get_unique_classes(examples)
    total = len(examples)
    probability_table = []
    for each_class in unique_classes:
        count_for_probability = 0
        for each_example in examples:  # each row
            if each_example[-1] == each_class:
                count_for_probability += 1
        probability = (count_for_probability/total)
        probability_table.append(probability)
    current_entropy = sum(-p * math.log2(p) for p in probability_table)

    return current_entropy
```

Information_gain() function serves two purposes in my program in that it shows numbers that are used to decide how the program should split a node. Also, it tallies up “simulated” entropy in order to attain how much information would remain after a split.

```
def information_gain(attribute, examples):
    '''calculate information gain at a node'''
    # importance function in the textbook
    # calculate entrophy from each subset
    # and subtract them from parent's entrophy

    list_of_unique_values = get_subsets_of_examples(attribute, examples)
    # print(list_of_unique_values)

    entrophy_list = []
    value_entropy_pair = namedtuple('Value_Entrophy_Pair', 'value entrophy probability')

    print("**** Attribute: {} ****\n".format(attribute))
    for each_value in list_of_unique_values:
        subset = [example for example in examples if example[attribute] == each_value]
        print("Attribute Value: " + str(each_value))
        print("Entrophy value: " + str(entropy(subset)) + "\n")
        probability = float(len(subset)/len(examples))
        entrophy_list.append(value_entropy_pair(each_value, entropy(subset), probability))

    remainder = 0
    for each_entropy_value in entrophy_list:
        remainder += each_entropy_value.entrophy * each_entropy_value.probability

    remainder = math.ceil(remainder*10000)/10000
    print("Remainder for attribute {} is: {}\n".format(attribute, remainder))

    attribute_remainder_pair = namedtuple('attribute_remainder_pair', 'attribute remainder')

    return attribute_remainder_pair(attribute, remainder)
```

Next, we subtract remainder from entropy of a parent node in order to find the maximum information gain possible. Therefore, I think one could say that “importance” function in the text was split into two in this case.

```
def calculate_info_gain_for_each_attribute(remainders, parent_examples):
    information_gain_list = []
    information_gain = namedtuple('Information_Gain', 'attribute gain')
    for each_remainder in remainders:
        # print(entropy(parent_examples))
        gain = math.ceil((entropy(parent_examples) - each_remainder.remainder) * 10000) / 10000
        information_gain_list.append(information_gain(each_remainder.attribute, gain))
    return information_gain_list
```

Lastly, using information we would have from functions above, we find the best attribute that we can use to split a node in order to gain more knowledge as to how we should classify a given data.

```
def split_on_which_attribute(examples, attributes):
    '''Find the best attribute to split on'''
    information_gain_table = []
    for attribute in attributes:
        if attribute is not None:
            information_gain_table.append(information_gain(attribute, examples))
        else:
            pass

    table_of_remainders = calculate_info_gain_for_each_attribute(information_gain_table, examples)
    max_gain = max([info.gain for info in table_of_remainders])

    split_on = []
    for each_remainder in table_of_remainders:
        if each_remainder.gain == max_gain:
            split_on.append(each_remainder.attribute)

    print("Spliting on {} gives us the maximum info gain of {}".format(split_on[0], max_gain))
    return split_on[0]
```

Attached below are snippets of screenshots of results that one would get from running the program using “WillWait-data.txt”

```
$ python decision_tree.py

Welcome to Decision Tree Algorithm Test

Please enter a file name: WillWait-data.txt


**** Attribute: 0 ****

Attribute Value: No
Entrophy value: 1.0

Attribute Value: Yes
Entrophy value: 1.0

Remainder for attribute 0 is: 1.0

**** Attribute: 1 ****

Attribute Value: No
Entrophy value: 1.0

Attribute Value: Yes
Entrophy value: 1.0

Remainder for attribute 1 is: 1.0

**** Attribute: 2 ****

Attribute Value: No
Entrophy value: 0.9852281360342515

Attribute Value: Yes
Entrophy value: 0.9709505944546686

Remainder for attribute 2 is: 0.9793
```

```
**** Attribute: 8 ****

Attribute Value: Burger
Entrophy value: 1.0

Attribute Value: French
Entrophy value: 1.0

Attribute Value: Thai
Entrophy value: 1.0

Attribute Value: Italian
Entrophy value: 1.0

Remainder for attribute 8 is: 1.0

**** Attribute: 9 ****

Attribute Value: 30-60
Entrophy value: 1.0

Attribute Value: 0-10
Entrophy value: 0.9182958340544896

Attribute Value: 10-30
Entrophy value: 1.0

Attribute Value: >60
Entrophy value: 0.0

Remainder for attribute 9 is: 0.7925

Spliting on 4 gives us the maximum info gain of 0.5408
```

Leaf nodes of a tree

```
LeafNode: [['Yes', 'No', 'Yes', 'No', 'Full', '$$$', 'No', 'Yes', 'French', '>60', 'No'], ['N
LeafNode: [['Yes', 'Yes', 'Yes', 'Yes', 'Full', '$', 'No', 'No', 'Burger', '30-60', 'Yes']]
LeafNode: [['Yes', 'Yes', 'Yes', 'Yes', 'Full', '$$$', 'No', 'Yes', 'Italian', '10-30', 'No']
LeafNode: [['Yes', 'No', 'No', 'Yes', 'Full', '$', 'No', 'No', 'Thai', '30-60', 'No']]
LeafNode: [['Yes', 'No', 'Yes', 'Yes', 'Full', '$', 'Yes', 'No', 'Thai', '10-30', 'Yes']]
LeafNode: [['Yes', 'No', 'No', 'Yes', 'Some', '$$$', 'No', 'Yes', 'French', '0-10', 'Yes'], [
'No', 'Yes', 'No', 'Yes', 'Some', '$$', 'Yes', 'Yes', 'Italian', '0-10', 'Yes'], ['No', 'No',
LeafNode: [['No', 'Yes', 'No', 'No', 'None', '$', 'Yes', 'No', 'Burger', '0-10', 'No'], ['No'
```

```
The following attributes were used to split examples (in order):
[4, 3, 8, 2]
```

- Iris.data.discrete.txt

```
LeafNode: [['ML', 'ML', 'L', 'L', 'Iris-virginica'], ['MS', 'MS', 'ML', 'L', 'Iris-vir
inica'], ['L', 'MS', 'L', 'L', 'Iris-virginica'], ['L', 'ML', 'L', 'L', 'Iris-virginic
ica'], ['ML', 'MS', 'L', 'L', 'Iris-virginica'], ['MS', 'S', 'ML', 'L', 'Iris-virginic
ica'], ['L', 'L', 'L', 'L', 'Iris-virginica'], ['L', 'MS', 'L', 'L', 'Iris-virginica']
], ['L', 'MS', 'L', 'L', 'Iris-virginica'], ['ML', 'ML', 'L', 'L', 'Iris-virginica'],
'L', 'L', 'L', 'L', 'Iris-virginica'], ['ML', 'MS', 'L', 'L', 'Iris-virginica'], ['L',
 'MS', 'ML', 'L', 'Iris-virginica'], ['ML', 'MS', 'L', 'L', 'Iris-virginica'], ['ML',
, 'ML', 'L', 'L', 'Iris-virginica'], ['ML', 'ML', 'L', 'L', 'Iris-virginica'], ['ML',
 'MS', 'ML', 'L', 'Iris-virginica'], ['ML', 'ML', 'ML', 'L', 'Iris-virginica']]
LeafNode: [['S', 'ML', 'S', 'S', 'Iris-setosa'], ['S', 'MS', 'S', 'S', 'Iris-setosa'],
ML', 'S', 'S', 'Iris-setosa'], ['MS', 'L', 'S', 'S', 'Iris-setosa'], ['S', 'ML', 'S',
is-setosa'], ['S', 'MS', 'S', 'S', 'Iris-setosa'], ['MS', 'ML', 'S', 'S', 'Iris-setosa
, 'MS', 'S', 'S', 'Iris-setosa'], ['MS', 'L', 'S', 'S', 'Iris-setosa'], ['MS', 'L', 'S
'Iris-setosa'], ['MS', 'L', 'S', 'S', 'Iris-setosa'], ['S', 'L', 'S', 'S', 'Iris-setos
S', 'ML', 'S', 'S', 'Iris-setosa'], ['S', 'ML', 'S', 'S', 'Iris-setosa'], ['S', 'ML',
, 'Iris-setosa'], ['MS', 'ML', 'S', 'S', 'Iris-setosa'], ['MS', 'ML', 'S', 'S', 'Iris-
, ['MS', 'ML', 'S', 'S', 'Iris-setosa'], ['MS', 'L', 'S', 'S', 'Iris-setosa'], ['MS',
, 'S', 'Iris-setosa'], ['MS', 'ML', 'S', 'S', 'Iris-setosa'], ['S', 'MS', 'S', 'S', 'I
sa'], ['S', 'ML', 'S', 'S', 'Iris-setosa'], ['S', 'S', 'S', 'S', 'Iris-setosa'], ['S',
', 'S', 'Iris-setosa'], ['S', 'MS', 'S', 'S', 'Iris-setosa'], ['S', 'L', 'S', 'S', 'Ir
sa'], ['S', 'ML', 'S', 'S', 'Iris-setosa']]
```

```
LeafNode: [['L', 'ML', 'ML', 'ML', 'Iris-versicolor']]
LeafNode: [['S', 'S', 'ML', 'ML', 'Iris-virginica']]
LeafNode: [['ML', 'S', 'ML', 'ML', 'Iris-versicolor'], ['ML', 'S', 'ML', 'ML', 'Iris-versicolor'], ['
LeafNode: [['ML', 'ML', 'ML', 'ML', 'Iris-versicolor'], ['ML', 'ML', 'ML', 'ML', 'Iris-versicolor']]
LeafNode: [['MS', 'MS', 'MS', 'ML', 'Iris-versicolor'], ['MS', 'MS', 'MS', 'ML', 'Iris-versicolor']]
LeafNode: [['S', 'S', 'MS', 'MS', 'Iris-versicolor'], ['S', 'S', 'MS', 'MS', 'Iris-versicolor'], ['MS
s-versicolor'], ['MS', 'S', 'MS', 'MS', 'Iris-versicolor'], ['ML', 'MS', 'ML', 'MS', 'Iris-versicolor
S', 'Iris-versicolor'], ['MS', 'S', 'MS', 'MS', 'Iris-versicolor'], ['MS', 'MS', 'MS', 'MS', 'Iris-ve
 'ML', 'MS', 'Iris-versicolor'], ['S', 'S', 'MS', 'MS', 'Iris-versicolor'], ['MS', 'MS', 'ML', 'MS',
```

```
The following attributes were used to split examples (in order):
[3, 2, 0, 1]
```

Finally, attributes are represented using their indexes. Therefore, it would be helpful to add +1 to each attribute to match real attributes.

Special Note: I am lost as to how I could go about evaluating the decision tree.