Machine Learning CS 529 Project 1 Report

Esteban Guillen

High-level Description of code

I implemented the ID3 algorithm in Python (Figure 1). All code in contained in a file called ***id3.py***. The code follows the algorithm (https://en.wikipedia.org/wiki/ID3_algorithm) closely and adds the ability to switch between entropy and misclassification error for the impurity measure. Split stopping using the Chi-squared test was also added.

/Users/esteban/unm/MachineLearning/Project1/id3.py — Brackets

```python
#performs the id3 decision tree algorithm (recursive), returns the root node of the tree
#  examples - set (pandas DataFrame) of training examples provided for current iteration of the algorithm
#  target_attribute - the name of the column that provides the classification label
#  attribute_list - set of the names of the attributes for the training examples (excludes target_attribute)
#  alpha - the alpha value ["0.01","0.05","0.5","1.0"] used for chi-square pruning
def id3(examples, target_attribute, attribute_list, alpha):
    root = Node()
    if(all_one_value(examples, target_attribute, positive_value)):
        leaf = Leaf()
        leaf.label = positive_value
        return leaf
    if(all_one_value(examples, target_attribute, negative_value)):
        leaf = Leaf()
        leaf.label = negative_value
        return leaf
    if(len(attribute_list) == 0):
        leaf = Leaf()
        leaf.label = extract_most_common_value(examples,target_attribute)
        return leaf
    #evaluate node selection using entropy or misclassification error
    if evaluation_criteria == 'entropy':
        a = determine_best_attribute_entropy(examples, target_attribute, attribute_list)
    else:
        a = determine_best_attribute_misclassification_error(examples, target_attribute, attribute_list)

    #for pruning get the calculated and threshold chi-square value
    chi_square_calculated = chi_square_calculation(examples,target_attribute,a)
    chi_square_lookup = chi_square_value_for_attribute(a, alpha)

    #if the calculated value is less than the threshold value then prune (decision node -> leaf node)
    if chi_square_calculated < chi_square_lookup:
        leaf = Leaf()
        leaf.label = extract_most_common_value(examples,target_attribute)
        return leaf
    root.decision_attribute = a

    values = possible_range_of_values(a)
    #iterate over all the possible values of an attribute and create branches for each value
    for value in values:
        root.branches[value] = ""
        examples_v = filter_data(examples,a,value)
        size_of_examples_v = size(examples_v)
        #if there are no more examples to train on create a leaf node for this branch
        if size_of_examples_v == 0:
            leaf = Leaf()
            leaf.label = extract_most_common_value(examples,target_attribute)
            root.add_branch(value,leaf)
        else:
            #remove attribute a so we don't reprocess it
            if a in attribute_list:
                attribute_list.remove(a)
            #add the next best decision node or leaf
            root.add_branch(value, id3(examples_v,target_attribute,attribute_list, alpha))
    return root
```

Line 280, Column 51 — 503 Lines | INS | Python | Spaces: 4

Figure 1. ID3 Algorithm

I used pandas DataFrame data structures for holding the training, testing and validation data (Figure 2 shows training and testing data being loaded). The DataFrame made it easy to index into the data and create subsets based on attribute values.

```python
names_of_attributes = names=["label","cap-shape","cap-surface","cap-color","bruises","oder",
                             "gill-attachment","gill-spacing","gill-size","gill-color","stalk-shape",
                             "stalk-root","stalk-surface-above-ring","stalk-surface-below-ring",
                             "stalk-color-above-ring","stalk-color-below-ring","veil-type",
                             "veil-color","ring-number","ring-type","spore-print-color",
                             "population","habitat"]

#read training data into pandas DataFrame
training_data = pd.read_table("data/training.txt", sep=",", names=names_of_attributes)

#read testing data into pandas DataFrame
testing_data = pd.read_table("data/testing.txt", sep=",", names=names_of_attributes)

```

Figure 2. Using pandas DataFrame

I used two types of objects to represent my decision tree (Figure 3). ***Node*** which is a decision node that consists of a string representing the decision attribute and a dictionary to manage the branches and child nodes. ***Leaf*** which simply consists of a string that represents the classification label (e or p).

```python
#represents a decision node in the tree
class Node:
    #identifies which attribute this node tests on
    decision_attribute = ""

    #initialize node
    def __init__(self):
        self.decision_attribute = ""
        self.branches = {}

    #adds a child Node or Leaf
    def add_branch(self,attribute_value,node):
        self.branches[attribute_value] = node

    #returns all branches
    def get_branches(self):
        return self.branches

#represents a leaf node in the tree
class Leaf:
    #label representing the classification value (e or p)
    label = ""
```

Figure 3. Objects used to represent the decision tree

The calculations for entropy and misclassification error can be seen below (Figure 4). The calculations were very straight forward (entropy used the log function from the math library).

/Users/esteban/unm/MachineLearning/Project1/id3.py — Brackets

```python
#calculates the entropy on a subset (examples_v) of data
def calculate_entropy(examples_v,target_attribute):
    entropy = 0.0
    total_size = size(examples_v)
    #if there is no data just return a zero value (does not contribute to calculation)
    if total_size == 0:
        return 0.0
    positive_filtered_examples_v = filter_data(examples_v, target_attribute, positive_value)
    negative_filtered_examples_v = filter_data(examples_v, target_attribute, negative_value)

    num_positive = size(positive_filtered_examples_v)
    calc_positive = 0.0
    if num_positive != 0:
        calc_positive = -(num_positive/total_size)*math.log((num_positive/total_size),2)


    num_negative = size(negative_filtered_examples_v)
    calc_negative = 0.0
    if num_negative != 0:
        calc_negative = - (num_negative/total_size)*math.log((num_negative/total_size),2)

    #returns -p(+)log p(+) - p(-)log p(-)
    return calc_positive + calc_negative

#calculates the misclassification error on a subset (examples_v) of data
def calculate_misclassification_error(examples_v, target_attribute):
    error = 0.0
    total_size = size(examples_v)
    #if there is no data just return a zero value (does not contribute to calculation)
    if total_size == 0:
        return 0.0

    positive_filtered_examples_v = filter_data(examples_v,target_attribute,positive_value)
    negative_filtered_examples_v = filter_data(examples_v,target_attribute,negative_value)

    num_positive = size(positive_filtered_examples_v)
    p_positive = (num_positive/total_size)

    num_negative = size(negative_filtered_examples_v)
    p_negative = (num_negative/total_size)

    error = 1.0 - max([p_positive,p_negative])
    #returns 1 - max(probability of positive,probability of negative)
    return error
```

Line 217, Column 54 — 503 Lines | INS | Python | Spaces: 4

Figure 4. Entropy and misclassification calculations

Gain was used to determine the “best attribute” for each recursive call of the ID3 algorithm. The implementations for entropy and misclassification can be seen below (Figure 5).

/Users/esteban/unm/MachineLearning/Project1/id3.py — Brackets

```python
#returns the attribute with the best informatin gain using entropy as the impurity measure
def determine_best_attribute_entropy(examples, target_attribute, attributes):
    best_attribute = ""
    max_information_gain = 0.0

    #find the entropy of S (examples)
    entropy_s = calculate_entropy(examples,target_attribute)
    size_of_s = size(examples)


    #loop through attributes and calculate the gain
    for attribute in attributes:
        values = possible_range_of_values(attribute)
        information_gain = entropy_s
        for value in values:
            examples_v = filter_data(examples,attribute,value)
            entropy_v = calculate_entropy(examples_v,target_attribute)
            size_of_examples_v = size(examples_v)
            weight = (size_of_examples_v/size_of_s)
            information_gain = information_gain - weight*entropy_v
        #if we have a new candidate for the best attribute store the gain and attribute name
        if information_gain > max_information_gain:
            max_information_gain = information_gain
            best_attribute = attribute

    return best_attribute

#returns the attribute with the best gain using misclassification error as the impurity measure
def determine_best_attribute_misclassification_error(examples, target_attribute, attributes):
    best_attribute = ""
    max_gain = 0.0
    #find the misclassification error of S (examples)
    error_s = calculate_misclassification_error(examples,target_attribute)
    size_of_s = size(examples)

    #loop through attributes and calculate the gain
    for attribute in attributes:
        values = possible_range_of_values(attribute)
        gain = error_s
        for value in values:
            examples_v = filter_data(examples,attribute,value)
            error_v = calculate_misclassification_error(examples_v,target_attribute)
            size_of_examples_v = size(examples_v)
            weight = (size_of_examples_v/size_of_s)
            gain = gain - weight*error_v
        #if we have a new candidate for the best attribute store the gain and attribute name
        if gain > max_gain:
            max_gain = gain
            best_attribute = attribute

    return best_attribute
```

Line 217, Column 54 — 503 Lines | INS | Python | Spaces: 4

Figure 5. Gain calculations for entropy and misclassification

The Chi-square test was used for split stopping and the code for calculating the Chi-square value is shown below (Figure 6). The calculation was broken up into 2 functions, the first calculates the Chi-square for a specific value of an attribute and the second sums all those calculations (over all possible values of an attribute).

/Users/esteban/unm/MachineLearning/Project1/id3.py — Brackets

```python
#sub calculation for chi-square, calculates the chi-square for an attribute value
def chi_square_calculation_attribute_v(examples_v, target_attribute,attribute_v):
    total_count = size(examples_v)
    #if the size of examples_v is zero then there is nothing to calculate and return 0
    if total_count == 0:
        return 0.0

    positive_filtered_examples = filter_data(examples_v, target_attribute, positive_value)
    negative_filtered_examples = filter_data(examples_v, target_attribute, negative_value)

    observed_positive = size(positive_filtered_examples)
    expected_positive = (total_count/2.0)

    observed_negative = size(negative_filtered_examples)
    expected_negative = (total_count/2.0)

    pos = ((observed_positive - expected_positive)**2)/expected_positive
    neg = ((observed_negative - expected_negative)**2)/expected_negative
    return pos + neg

#returns the chi-square value for an attribute
def chi_square_calculation(examples, target_attribute,attribute):
    chi_square_sum = 0.0
    values = possible_range_of_values(attribute)
    for value in values:
        examples_v = filter_data(examples, attribute,value)
        chi_square_v = chi_square_calculation_attribute_v(examples_v,target_attribute,value)
        chi_square_sum = chi_square_sum + chi_square_v
    return chi_square_sum
```

Line 217, Column 54 — 503 Lines | INS | Python | Spaces: 4

Figure 6. Chi-square calculation code

I implemented a number of helper functions to make the code more readable and maintainable. Check the full source code (***id3.py***) for more details.

Accuracy Results

All of my decision trees (entropy and misclassification error for 99, 95, 50, and 0 CL) produced 100% accuracy results (Figure 7). None of my trees got pruned using the Chi-squared test, all of my calculated values were less then the threshold values. The trees that were produced were small (4 levels for entropy and 5 levels for misclassification error, with most decision nodes having only one branch pointing to another decision node). The training data provided must have produced a near optimal tree, and no amount of pruning would improve performance (can’t get better than 100%). Most of the training data was classified at the root node and 8 of the 9 branches pointed to Leaf nodes.

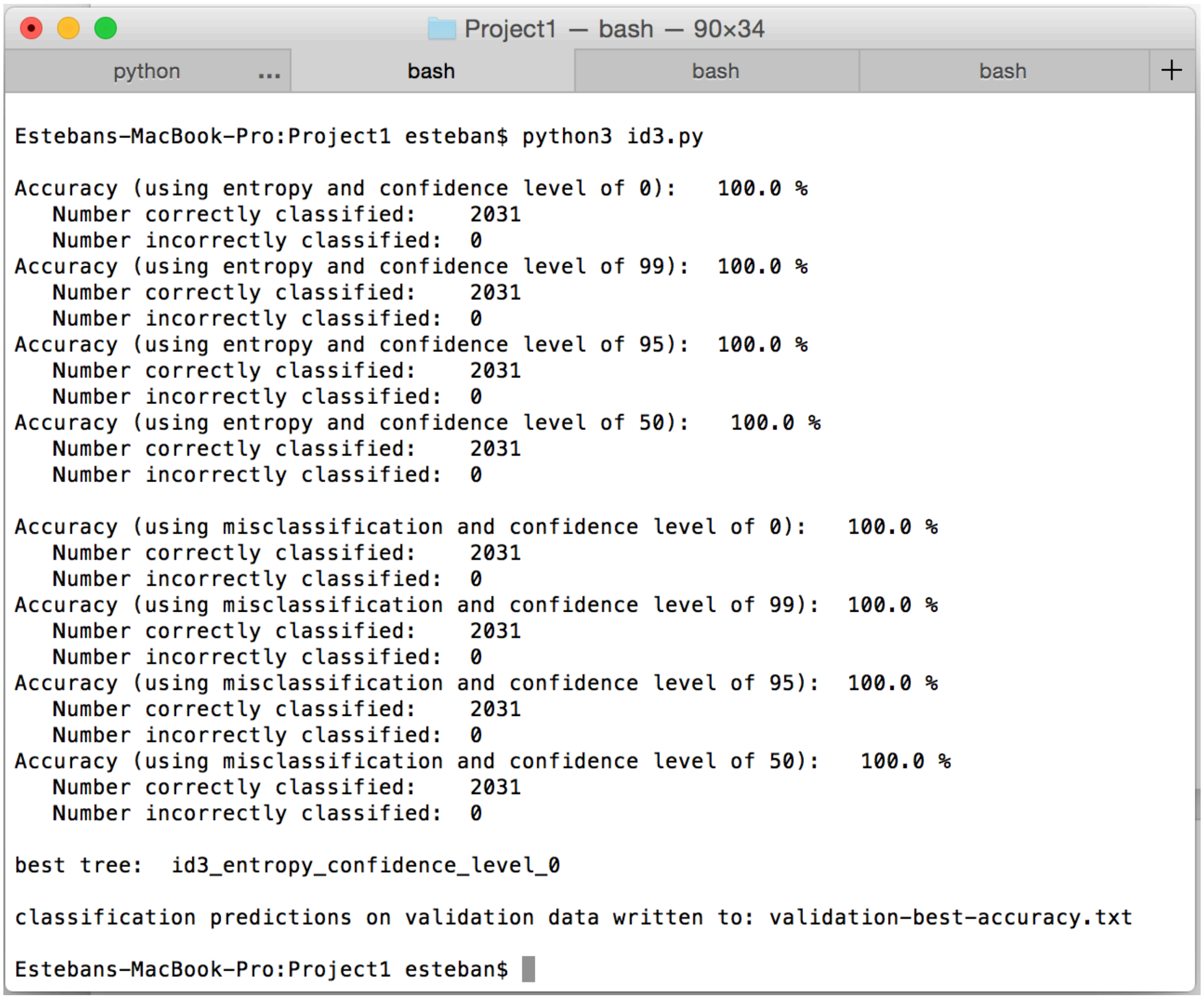

Figure 7. Classification results on the training.txt data

I used the entropy with CL 0% decision tree to classify the validation data. The 100% classification results can be seen below (Figure 8)

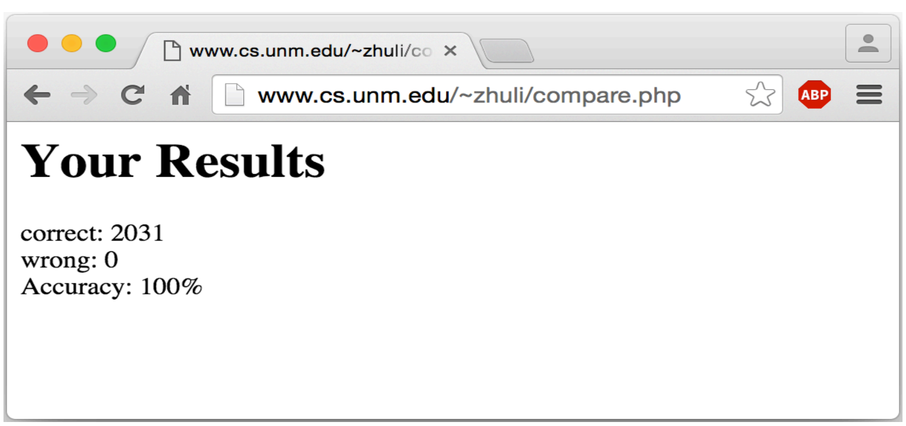

Figure 8. Accuracy for the validation.txt data

Folklore Rules

For proving or disproving the folklore rules I look at the data and/or my decision tree to come to a conclusion. I concluded that a folklore rule was true if the data or my decision tree backed-up the claim with a <u>high probability</u> (most of the data supported the claim). The included HTML export of my IPython Notebook shows how I queried the data.

<u>Rule 1</u>: Poisonous mushrooms are brightly colored:
False: I found there to be a fairly even split between edible and poisonous mushrooms for brightly colored (cap-color) mushrooms.

<u>Rule 2</u>: Poisonous mushrooms taste/smell bad:
True: The data provided strongly supports this (for smell bad). Almost all mushrooms with a bad odor were poisonous. My decision tree also supports this rule.

<u>Rule 3</u>: Poisonous mushrooms have a pointed or umbrella shaped cap:
False: The data provided strongly disproves this theory. Almost an even split between edible and poisonous mushroom having a pointed or umbrella shaped cap.

<u>Rule 4</u>: Edible mushroom have flat rounded shaped cap:
False: The data provided strongly disproves this theory. Almost an even split between edible and poisonous mushroom having a flat shaped cap.

<u>Rule 5</u>: Poisonous mushrooms have warts or scales on the cap:
False: The data provided strongly disproves this theory. Almost an even split between edible and poisonous mushroom having warts or scales on the cap.

<u>Rule 6</u>: Poisonous mushrooms have a bulbous cup or sac around the base:
False: The data provided strongly disproves this theory. Almost an even split between edible and poisonous mushroom having a bulbous cup or sac around the base.

<u>Rule 7</u>: Poisonous mushroom have a ring around the stem:
False: The data provided strongly disproves this theory. Almost an even split between edible and poisonous mushroom having a ring around the stem.

<u>Rule 8</u>: Poisonous mushrooms have gills that are thin and white:
False: The data provided strongly disproves this theory. Almost an even split between edible and poisonous mushroom having gills that are thin and white.

Proposed Rules

I could only find one rule not mentioned above to be true. The training data strongly (100%) supports "green spore print color mushroom are poisonous".
<u>Rule 9</u>: Poisonous mushrooms have green spore print color: