CMPE480 Fall 23-24 HW4

Zeynep Buse Aydın

Dataset

The dataset I worked on is for loan prediction. There are 8 attributes:

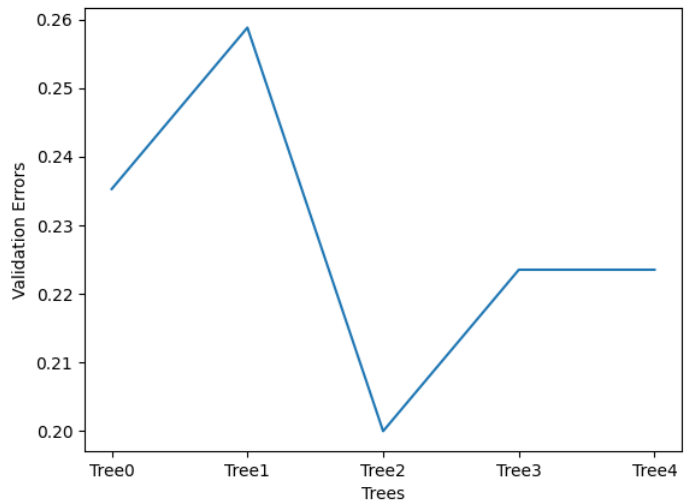
1. **Loan_ID**, a unique ID for each person
2. **Gender**, “Male” or “Female”
3. **Married**, “Yes” or “No”
4. **Education**, “Graduate” or “Not Graduate”
5. **Self_Employed**, “Yes” or “No”
6. **Credit_History**, “0” or “1”
7. **Property_Area**, “Rural”, “Urban” or “Semiurban”
8. **Loan_Status**, “Y” or “N”

Depending on the background features of the given person, it is decided if they will be granted the loan or not. The attributes 2 to 7 are used as the background features of the person, and with them Loan_Status is decided as “Y” indicating they will get the loan, or “N” otherwise.

5-Fold Cross Validation

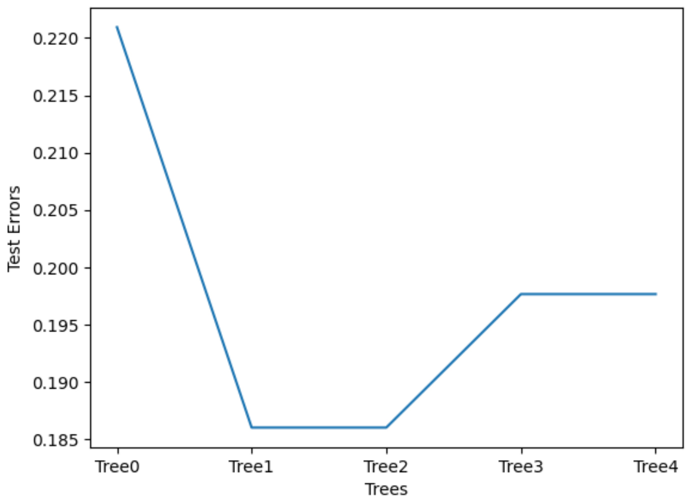
In total, there were 511 entries in the dataset. I first splitted them into 2: 86 for testing, 425 for training and validation. Then, I divided the 425 entries into 2 for 75% for training and 25% for validation. I did this 5 times and at each iteration i I got the i*85 to (i+1)*85 entries as the validation set, the remaining as the training set. At each iteration I created another decision tree and calculated the errors using the validation set. Finally, I calculated the overall error as the average of these 5 errors.

Error Plots



Validation Errors:
Here, each tree that is generated at each iteration during k-fold cross validation is validated and the errors can be seen in the plot.
The errors are as follows:
0.2352941
0.2588235
0.2
0.2235294
0.2235294

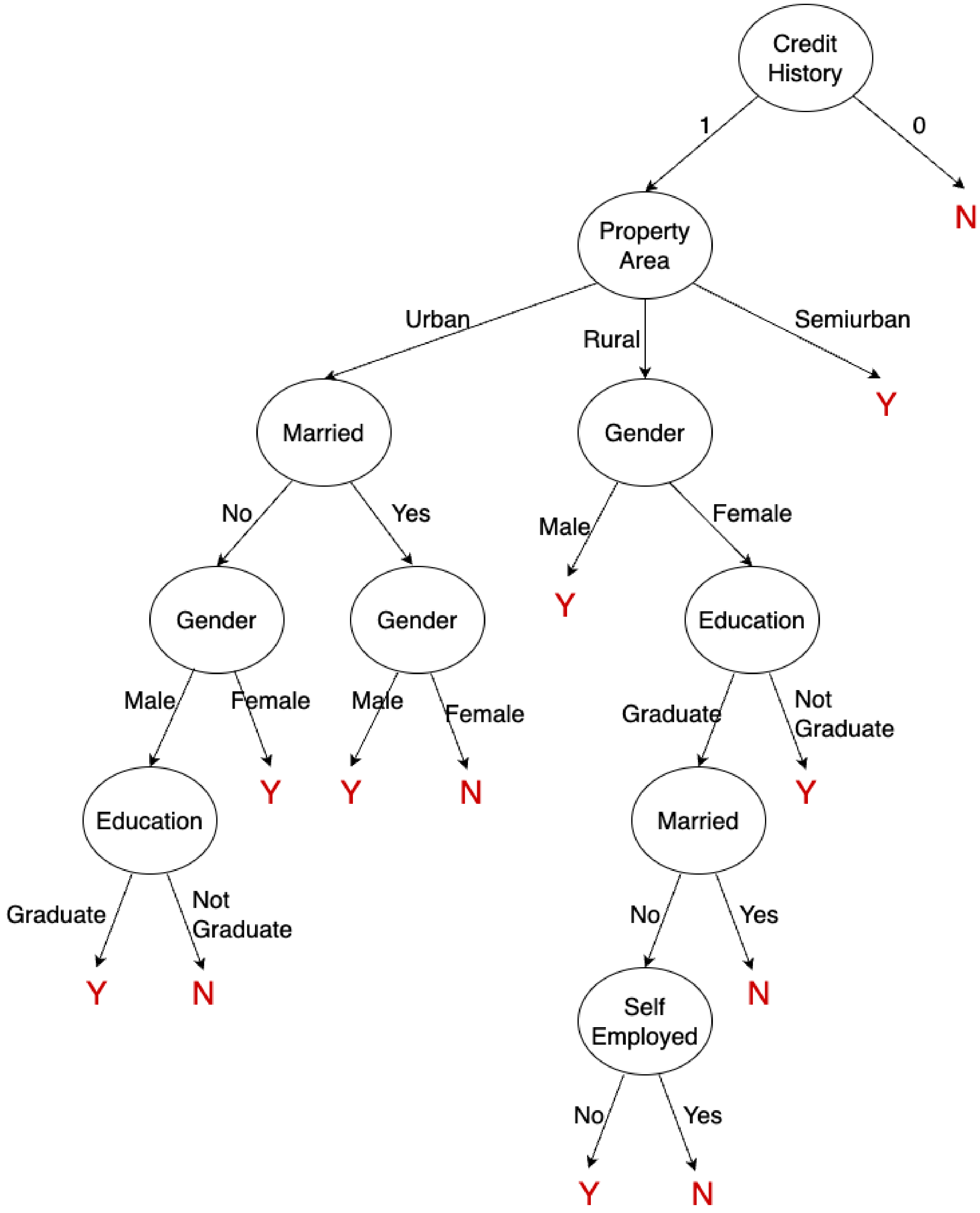
The overall error is the average = 0.2282353



Test Errors:
Each tree is also tested with the test set.
The errors are as follows:
0.22093023255813954
0.18604651162790697
0.18604651162790697
0.19767441860465115
0.19767441860465115

Final Decision Tree

Since the second tree (Tree2) gives the smallest errors, it is the best one.



Source Code

```
import math
import matplotlib.pyplot as plt

# attributes = {Gender: ["Male", "Female"], Education: ["Graduate", "Not
Graduate"], ...}
# attribute_values = for Gender: ["Male", "Female"]
# examples = [{Loan_ID: <id>, Gender: <Gender>, ... }, {...}, {...}]

class Node:
   def __init__(self, goal_value=""):
       self.children = []
       self.attribute = ""
       self.attr_value = ""
       self.goal_value = goal_value  # if leaf node


def entropy(q):     # entropy of a boolean random variable q = pos / (pos+neg)
   if q == 0:
       return -1 * (1-q) * math.log(1-q, 2)
   if q == 1:
       return -1 * q * math.log(q, 2)
   return -1 * (q * math.log(q, 2) + (1-q) * math.log(1-q, 2))


def attribute_counts(attribute, attributes, examples):
   attr_dict = {}
   for attr_val in attributes[attribute]:
       attr_dict[attr_val] = [0, 0]
   for example in examples:
       if example[goal] == positive:
           attr_dict[example.get(attribute)][0] += 1
       else:
           attr_dict[example.get(attribute)][1] += 1
   return attr_dict    # ex: {"Male": [4, 6]} pos, neg


# get the probability of positives in all data
def positive_prob(examples):
   positives = 0
   for example in examples:
       if example[goal] == positive:
           positives += 1
```

```
    return positives/len(examples)


# remainder = sum((pk+nk)/(p+n) * entropy(pk/(pk+nk)))
def remainder(attribute, attributes, examples):
    attr_sum = 0
    attr_dict = attribute_counts(attribute, attributes, examples)
    for attribute in attr_dict.keys():
        positives = attr_dict[attribute][0]
        negatives = attr_dict[attribute][1]
        total = positives + negatives
        if positives + negatives == 0:
            continue
        attr_sum += (total) * entropy(positives / total)
    return 1/len(examples) * attr_sum


def importance(attribute, attributes, examples):
    return entropy(positive_prob(examples)) - remainder(attribute, attributes,
examples)


# return the more occurring goal value
def plurality_value(examples):
    positive_count = 0
    negative_count = 0
    for example in examples:
        if example[goal] == positive:
            positive_count += 1
        else:
            negative_count += 1
    return positive if positive_count >= negative_count else negative


# if all the children nodes result in the same goal value, get rid of them
def eliminate_attributes(node):
    if len(node.children) == 0:
        return
    child0 = node.children[0]
    for child in node.children:
        if child0.goal_value != child.goal_value:
            return
    if child0.goal_value == "":
        return
    node.goal_value = child0.goal_value
```

```
    node.children = []


# if there are unnecessary nodes, get rid of them
def clear_tree(root):
    for i in range(depth+1):
        queue = [root]
        while queue:
            level_size = len(queue)
            while level_size > 0:
                node = queue.pop(0)
                eliminate_attributes(node)
                level_size -= 1
                for child in node.children:
                    queue += [child]


def check_if_all_same(examples):
    first_example = examples[0][goal]
    for example in examples:
        if example[goal] != first_example:
            return False, None
    return True, first_example


# get the attribute which provides the largest information gain
def arg_max_importance(attributes, examples):
    curr_max = float('-inf')
    max_arg = ""
    for a in attributes:
        if a == goal or a == id_ignored:
            continue
        gain = importance(a, attributes, examples)
        if gain > curr_max:
            curr_max = gain
            max_arg = a
    return max_arg


# create new examples list with entries that have <attribute_value>
def create_new_examples(attribute, attribute_value, parent_examples):
    new_examples = []
    for example in parent_examples:
        if example[attribute] == attribute_value:
            new_examples.append(example)
```

```
    return new_examples


# create new attributes list that doesn't have the <attribute>
def create_new_attributes(attribute, attributes):
    new_attributes = {}
    for attr in attributes.keys():
        if attr == attribute:
            continue
        new_attributes[attr] = attributes[attr]
    return new_attributes


def decision_tree_learning(attributes, examples, parent_examples):
    global depth
    depth = 1
    if len(examples) == 0:
        tree = Node(plurality_value(parent_examples))
        return tree

    all_same_class = check_if_all_same(examples)
    if all_same_class[0]:
        tree = Node(all_same_class[1])
        return tree

    if len(attributes) == 2:    # id and goal attributes are ignored
        return Node(plurality_value(examples))

    attribute = arg_max_importance(attributes, examples)
    tree = Node()
    for val in attributes[attribute]:
        new_examples = create_new_examples(attribute, val, examples)
        new_attributes = create_new_attributes(attribute, attributes)
        subtree = decision_tree_learning(new_attributes, new_examples, examples)
        subtree.attribute = attribute
        subtree.attr_value = val
        tree.children.append(subtree)
    depth += 1
    return tree


# {"Attribute1": [attr_val1, attr_val2, ...], ...}
def get_attributes(examples, first_line):
    attributes = {}
    for column in first_line:
```

```
        attributes[column] = []
    for example in examples:
        for key in example.keys():
            if example[key] in attributes[key]:
                continue
            attributes[key].append(example[key])
    return attributes


def get_data_from_csv(g, pos, neg, file_name):
    global goal     # Loan_Status
    global positive # "Y"
    global negative # "N"
    goal = g
    positive = pos
    negative = neg
    data_file = open(file_name, "r")
    columns = data_file.readline().strip().split(",")
    examples = []
    for line in data_file.readlines():
        data = line.strip().split(",")
        data_dict = dict()
        for i in range(len(columns)):
            data_dict[columns[i]] = data[i]
        examples.append(data_dict)
    return examples, columns


def calculate_error(data, root):
    incorrect_count = 0
    for example in data:
        if example[goal] != find_goal_value(example, root):
            incorrect_count += 1
    return incorrect_count / len(data)


def k_fold(k, examples, test_data):
    part_len = int(len(examples)/k)
    e_gen = 0
    test_errors = []
    validation_errors = []
    for i in range(k):
        train_data = examples[ : (i * part_len)] + examples[(i+1) * part_len: ]
        validate_data = examples[i * part_len: (i+1) * part_len]
        root = train(train_data)
```

```python
        clear_tree(root)
        validation_error = calculate_error(validate_data, root)
        e_gen += validation_error
        validation_errors.append(validation_error)
        test_errors.append(calculate_error(test_data, root))
    return e_gen/k, validation_errors, test_errors


def find_goal_value(example, node):
    if node.goal_value != "":
        return node.goal_value
    for child in node.children:
        if example.get(child.attribute) == child.attr_value:
            return find_goal_value(example, child)


def train(train_data):
    attributes = get_attributes(train_data, columns)
    return decision_tree_learning(attributes, train_data, train_data)


def plot_graph(x, y, xname, yname):
    plt.plot(x, y)
    plt.xlabel(xname)
    plt.ylabel(yname)
    plt.show()


if __name__ == "__main__":
    global columns
    global id_ignored
    examples, columns = get_data_from_csv("Loan_Status", "Y", "N", "training.csv")
    id_ignored = columns[0]

    test_data = get_data_from_csv("Loan_Status", "Y", "N", "test.csv")[0]
    k_fold_error, validation_errors, test_errors = k_fold(5, examples, test_data)
    trees = ["Tree0", "Tree1", "Tree2", "Tree3", "Tree4"]
    plot_graph(trees, validation_errors, "Trees", "Validation Errors")
    plot_graph(trees, test_errors, "Trees", "Test Errors")
```