5.Decision_Tree (without SKLearn)

November 9, 2021

Classification using Decision Tree - Without SK-Learn

[20]:
```
# Dataset
# Format: each row is an example
# The last column is the label
# The first two columns are the input features

training_data = [['Green', 3, 'Mango'],
                 ['Yellow', 3, 'Mango'],
                 ['Red', 1, 'Grape'],
                 ['Red', 1, 'Grape'],
                 ['Yellow', 3, 'Lemon']
                ]
header = ["color", "diameter","label"]
```

[21]:
```
# Finding unique values in a column
def unique_vals(rows, col):
    return set([row[col] for row in rows])

# unique_vals(trianing_data, 0)
# unique_vals(trianing_data, 1)
```

[22]:
```
# counting unique values for each type of example in a dataset
def class_counts(rows):
    counts = {}
    for row in rows:
        label = row[-1]
        if label not in counts:
            counts[label] = 0
        counts[label] += 1
    return counts

# class_counts(training_data)
```

[23]:
```
# To check for numeric or float
def is_numeric(value):
    return isinstance(value, int) or isinstance(value,float)
```

[24]:
```
# This records a column number (example, 0 for Color) and column value␣
↪(example, green). The 'match' method is used to
# compare the efature value in an example to the feature value stored in the␣
↪question.
class Question:
    def __init__(self, column, value):
        self.column = column
        self.value = value

    def match(self, example):
        val = example[self.column]
        if is_numeric(val):
            return val >= self.value
        else:
            return val == self.value

    def __repr__(self):
        condition = "=="
        if is_numeric(self.value):
            condition = ">="
        return "Is %s %s %s?" % (header[self.column], condition, str(self.
↪value))
```

[25]:
```
def partition(rows, question):
    # partitions dataset : if matches the question, "true rows", else "false␣
↪rows"
    true_rows, false_rows = [], []
    for row in rows:
        if question.match(row):
            true_rows.append(row)
        else:
            false_rows.append(row)
    return true_rows, false_rows
###
# Demo:
# Lets partition the training data based on whether rows are Red.
# true_rows, false_rows = partition(training_data, Question(0,'Red'))
# This will contain all the 'Red' rows.
# true_rows
# This will contain everything else
# flase_rows
####

def gini(rows):
        """calculate the gini impurity for a list of rows"""
        counts = class_counts(rows)
        impurity = 1
```

```python
        for lbl in counts:
            prob_of_lbl = counts[lbl] / float(len(rows))
            impurity -= prob_of_lbl**2
        return impurity
    ####
    # Demo:
    # Let us see how gini impurity works
    # First we will look at a dataset with no mixing.
    # no_mixing = [['Mango',['Mango']]]
    # gini(no_mixing)
    # this will return 0
    # lots_of_mixing =␣
 ↪[['Mango'],['Orange'],['Grape'],['Grapefruit'],['Blueberry']]
    # gini(lots_of_mixing)
    # This will return 0.8
    ####

def info_gain(left, right, current_uncertainty):
        """information gain. The uncertainity of the starting node, mius the␣
 ↪weighted impurity of two child nodes. """
        p = float(len(left)) / (len(left) + len(right))
        return current_uncertainty - p * gini(left) - (1-p) * gini(right)
    ####
    # /=Demo:
    # calculate the uncertainity of our training data
    # current_uncertianity = gini(training_data)
    #
    # how much information do we gain by partitioning on 'Green'?
    # true_rows, false_rows = partition(training_data, Question(0, 'Green'))
    # info_gain(true_rows, false_rows, current_uncertainity)
    #
    # what about if we partitioned on 'Red' instead?
    # true_rows, false_rows = partition(training_data, Question(0, 'Red'))
    # info_gain(true_rows, false_rows, current_uncertainity)
    ####

def find_best_split(rows):

        """Find the best question to ask by iterating over every feature /␣
 ↪value and calculting the information gain"""

        best_gain = 0                      # keep track of the best information␣
 ↪gain
        best_question = None               # keep track of the feature/value␣
 ↪that produced it
        current_uncertainty = gini(rows)
        n_features = len(rows[0])          # number of columns
```

```
        for col in range(n_features):     # for each feature

            values = set([row[col] for row in rows])   # unique values in the␣
↪column

            for val in values:            # for each value

                question = Question(col, val)

                # split the dataset
                true_rows, false_rows = partition(rows, question)

                # skip this split f it does not divide the dataset
                if len(true_rows) == 0 or len(false_rows) == 0:
                    continue

                # calculate the information gain from this split
                gain = info_gain(true_rows, false_rows, current_uncertainty)

                # You actually can use '>' instead of '>=' here
                # but I wanted the tree to look a certian way for our toy␣
↪dataset
                if gain >= best_gain:
                    best_gain, best_question = gain, question

        return best_gain, best_question
        ####
        # Demo:
        # Find the best question to ask first for out dataset
        # best_gain, best_question = find_best_split(trianing_data)
        # FYI: is color == Red is just as good. See the note in the code above
        # Where I used '>='
        ####
```

[26]:
```
class Leaf:
    """ A leaf node classifies data. This holds a dictionary of clas (ex:␣
↪"Mango") -> number of times it appears
    in the rows from the training data that reach this leaf.
    """
    def __init__(self, rows):
        self.predictions = class_counts(rows)

class Decision_Node:
    """ A decision node asks a question. This holds a reference to the␣
↪question, and to the two child nodes. """
    def __init__(self, question, true_branch, false_branch):
```

```python
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch
```

[27]:
```python
def build_tree(rows):
        # It builds the tree

        # partition the dataset on each of the unique attribute, calculate the␣
 ↪information gain, and return the question that
        # produces the highest gain
        gain, question = find_best_split(rows)

        # Base case: no further info gain
        # Since we can ask no further questions, we will return a leaf
        if gain == 0:
            return Leaf(rows)

        # If we reach here, we have found a userful feature/value to partition␣
 ↪on.
        true_rows, false_rows = partition(rows, question)

        # Recursively build the true branch
        true_branch = build_tree(true_rows)

        # Recursively build the true branch
        false_branch = build_tree(false_rows)

        # Return a question node
        # This records the best feature/value to ask at this point as well as␣
 ↪the branches to follow depending on the answer
        return Decision_Node(question, true_branch, false_branch)

def print_tree(node, spacing=""):
        """Tree model for this dataset"""

        # Base case: we have reached a leaf
        if isinstance(node, Leaf):
            print(spacing + "predict", node.predictions)
            return

        # Print the question at this node
        print(spacing + str(node.question))

        # call this function recursively on the true branch
        print(spacing + '--> True :')
        print_tree(node.true_branch, spacing + "  ")
```

```
        # call this function recursively on the false branch
        print(spacing + '--> False :')
        print_tree(node.false_branch, spacing + "  ")

def classify(row, node):
        # Base case: we have reached a Leaf
        if isinstance(node, Leaf):
            return node.predictions

        # Decide whether to follow the true-branch or the false-branch
        # Compare th feature/value stored in the node to the example we are␣
 ↪considering
        if node.question.match(row):
            return classify(row, node.true_branch)
        else:
            return classify(row, node.false_branch)

def print_leaf(counts):
        """print the predictions at a leaf"""
        total = sum(counts.values()) * 1.0
        probs = {}
        for lbl in counts.keys():
            probs[lbl] = str(int(counts[lbl] / total * 100)) + "%"
        return probs
    ####
    # Demo:
    # printing that a bit nicer
    # print_leaf(classify(training_data[0], my_tree))
    ####
```

[28]:

```
if __name__ == '__main__':
    my_tree = build_tree(training_data)
    print_tree(my_tree)

    # Evaluate
    testing_data = [
        ['Green', 3, 'Apple'],
        ['Yellow', 4, 'Apple'],
        ['Red', 2, 'Grape'],
        ['Red', 3, 'Grape'],
        ['Yellow', 3, 'Lemon'],
    ]

    for row in testing_data:
        print("Actual: %s. Predicted: %s" %(row[-1], print_leaf(classify(row,␣
 ↪my_tree))))
```

```python
# add support for missing (or unseen) attributes
# prune the tree to prevent overfitting
# add support for regression
```

```
Is label == Grape?
--> True :
  predict {'Grape': 2}
--> False :
  Is label == Lemon?
  --> True :
    predict {'Lemon': 1}
  --> False :
    predict {'Mango': 2}
Actual: Apple. Predicted: {'Mango': '100%'}
Actual: Apple. Predicted: {'Mango': '100%'}
Actual: Grape. Predicted: {'Grape': '100%'}
Actual: Grape. Predicted: {'Grape': '100%'}
Actual: Lemon. Predicted: {'Lemon': '100%'}
```