Recitation 2
Decision Trees

10-301/10-601: Introduction to Machine Learning
01/27/2023

1 Programming: Tree Structures and Algorithms

Topics Covered:
- Depth of nodes and trees
- Recursive traversal of trees
  - Depth First Search
    - Pre-order Traversal
    - In-order Traversal
    - Post-order Traversal
  - Breadth First Search (Self Study)
- Debugging in Python

Questions:

1. Depth of a tree definition

   $\text{Depth} = \text{# of levels until you reach the last leaf node.}$

   $\Rightarrow \text{Breadth of the longest path from the root node to a leaf node.}$

2. Depth of a node definition

   $\text{Distance from root} = \# \text{ of edges.}$
3. What is the depth of node A? What is the depth of node X in the tree?

4. What is the depth of node B?

5. What is the depth of node C? What are the steps to connect node A and node X in the tree?

---

**Python Code:**

```python
# This class represents an individual node.
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

    def insert(self, key):
        if self.val:
            if key < self.val:
                if self.left:
                    self.left.insert(key)
                else:
                    self.left = Node(key)
            else:
                if self.right:
                    self.right.insert(key)
                else:
                    self.right = Node(key)
        else:
            self.val = key

    def inOrder(self):
        if self:
            if self.left:
                self.left.inOrder()
            print(self.val)
            if self.right:
                self.right.inOrder()

def build_tree(root):
    root = Node(4)
    root.left = Node(2)
    root.right = Node(5)
    root.left.left = Node(1)
    root.left.right = Node(3)
    return root
```

---

**Explanation:**

The code above defines a class `Node` which represents a node in a binary tree. The `insert` method inserts a new key into the tree, and the `inOrder` method prints the tree in an in-order traversal. The `build_tree` function constructs a tree with the given structure.
If _name_ == '__main__':
    root = build_a_tree()
    print('traversal of the binary tree is: ')
    traversal1(root)
    print()
    print('traversal of the binary tree is: ')
    traversal2(root)
    print()
    print('traversal of the binary tree is: ')
    traversal3(root)

Now, identify which traversal function is pre-order, in-order, post-order DFS:
- traversal1() is
- traversal2() is
- traversal3() is

Code Output

traversal1 of the binary tree is:

traversal2 of the binary tree is:

traversal3 of the binary tree is:

2 ML Concepts: Mutual Information

Information Theory Definitions:
- \( H(Y) = - \sum_{y \in \text{universe}_Y} P(Y = y) \log_2 P(Y = y) \)
- \( H(Y \mid X) = - \sum_{x \in \text{universe}_X} P(Y = y) \sum_{x \in \text{universe}_X} P(Y = y \mid X = x) \log_2 P(Y = y \mid X = x) \)
- \( H(Y \mid X) = - \sum_{x \in \text{universe}_X} P(X = x) \log_2 P(X = x) \)
- \( I(X; Y) = H(Y) - H(Y \mid X) \)

Exercises:
1. Calculate the entropy of tossing a fair coin.
   \( P(H) = 0.5 \)
   \( -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = 1 \)

2. Calculate the entropy of tossing a coin that lands only on tails. Note: \( 0 \cdot \log_2 0 = 0 \).
   \( P(H) = 0 \quad P(T) = 1 \)
   \( -0 \cdot \log_2 0 - |\log_2 1| = 0 \)

3. Calculate the entropy of a fair dice roll.
   \( \sum_{k=1}^{6} \frac{1}{6} \log_2 \frac{1}{6} = 1 \)

4. When is the mutual information \( I(X; Y) = 0 \)?
   \( H(Y) - H(Y \mid X) = 0 \)
   \( H(Y) = H(Y \mid X) \)
   \( I(X; Y) = 0 \) if \( X \perp \! \! \! \perp Y \)
3 ML Concepts: Construction of Decision Trees

In this section, we will go over how to construct a decision tree based on a high level.

The following concepts will help guide us:

1. What exactly are the nodes we are creating?
2. What are the inputs and outputs at testing stage?
3. What do we need to do at training stage?
4. What happens at test stage?
5. What happens if max depth is reached?

1) The tasks:
- Given a set of train data, test data, max-depth, we want:
  - train data to learn decision tree classifier
  - use the trained classifier to predict labels for train and test data
  - calculate error for both train and test data

2) Train inputs
- train data
- max-depth of tree

Train outputs
- fully trained decision tree classifier
- predictions for every entry/row in test dataset

3) Class node:
- def __init__(self, attr):
- self.
- self.

4) Consider "stopping criterion"
- max-depth reached
- node is pure (data is all one label, entropy = 0)
- simple majority vote

- calculate entropy and mutual information for only non-used attributes, and select best attribute to split on
- split the data according to the best attribute

5) depth of tree = min (4 attributes, max-depth)
- majority vote

6) 1 depth of tree = min (4 attributes, max-depth)
- stop growing tree after all attributes have been used
4 Programming: Debugging with Trees

pelb and common commands
- import pdb; pdb.set_trace() (breakpoint() also allowed as per PEP 553)
- p variable (print value of variable)
- n (next)
- s (step into subroutine)
- ENTER (repeat previous command)
- q (quit)
- l (list where you are)
- b (breakpoint)
- c (continue)
- r (continue until the end of the subroutine)
- tcode (run Python code)

Real Practice
These are some (simplified) examples based on actual bugs previous students had. Link to the code: https://colab.research.google.com/drive/1M3p2Az7F0ud44dRj1/FH4Qybf8t4CCr?
usp-sharing

Buggy Code

```python
# Reverse the rows of a 2D array
def reverse_rows(original):
    rows = len(original)
    cols = len(original[0])
    new = [[0] * cols for _ in range(rows)]

    for i in range(rows):
        for j in range(cols):
            new[i][j] = original[j][i]

    return new

if __name__ == '__main__':
    a = [[1, 2], [3, 4], [5, 6]]
    print(reverse_rows(a))
```

---

Buggy Code

```python
import numpy as np

# biggest_col takes a binary 2D array and returns the index of the
# column with the most non-zero values. In case of a tie, return
# the smallest index.
def biggest_col(mat):
    sum_col = np.sum(mat, axis=0)
    max_count = -1
    max_index = -1

    # iterate over the columns of the matrix
    for col in range(len(sum_col)):
        # counts the number of nonzero values
        count = np.count_nonzero(mat[:, col])
        # change max if needed
        if count >= max_count:
            max_count = count
            max_index = col

    return max_index

# Helper function that returns the number of nonzero elements in
# a row in column col.
def get_count(mat, col):
    sum_row = np.sum(mat)
    count = 0
    for row in range(sum_row):
        count += (mat[row][col] == 0)
    return count

if __name__ == '__main__':
    # Expected answer: column index 2
    mat = [[1, 0, 0, 1],
           [0, 1, 1, 0],
           [1, 0, 0, 0],
           [0, 1, 1, 0],
           [0, 1, 1, 0]]
    assert biggest_col(mat) == 2
```