HW1 Report

> Author: Jiayi Liu

This is an individual work. The code can be found at `/code/main.py`.

Implementation Detail

Read File

- Param {str}: The path of .txt file
- Return {dictionary}: A dictionary of infomation including:
  - "record" {list}: a list of training data
  - "attributes" {list}: a list of attributes names
  - "options" {list}: a list of possible options corresponding to each attribute

```
def read_file(file_name):
    try:
        f = open(file_name)
    except FileNotFoundError:
        print('The file does not exist')
        exit()
    else:
        # Init info list
        records = []      # training data records
        attributes = []  # all the names of attributes
        options = []      # corresponding options for each attribute
        # Read all lines from the file
        lines = f.readlines()
        num_lines = len(lines)
        # Extract all the attributes from the first line
        attributes = lines[0][1: -2].split(', ')[:-1]
        # Extract records
        for i in range(2, num_lines):
            records.append(lines[i][4:-2].split(', '))
        f.close()
        # Collect options for each attributes
        for i in range(len(attributes)):
            options.append(list(set([record[i] for record in records])))
        # Return the essential info for the decision trees
        return {"records": records, "attributes": attributes, "options":
options}
```

Class TreeNode

I used N-ary tree as the basic data structure to implement the decision tree. I created a class `TreeNode` as the node for this tree.

```
class TreeNode:
    def __init__(self, value, index, trace, branch_from):
        self.value = value                  # Name of attribute
        self.attr_index = index             # Index of attribute
        self.trace = trace                  # Currently available attributes
        self.branch_from = branch_from      # Where does this node branch from
        self.children = []                  # List of children nodes
        self.branch_to = []                 # List of options corresponding to each
child
```

Class DecisionTree

All the logic of implementing a decision tree model is encapsulated in the class `DecisionTree`.

```
class DecisionTree:
    def __init__(self, info):
        self.records = info["records"]              # List of training data
        self.attributes = info["attributes"]        # List of attributes
        self.options = info["options"]              # List of possible options for
each attribute
        self.dummy_root = TreeNode("dummy_root", -1, [attr for attr in
range(len(self.attributes))], "null")
       # Dummy root of the tree
        self.test_data = []                         # Test data
```

Train Model

Model training is a recursive process of building up a decision tree from top to bottom. Here I inputed a dummy root for the convenience of starting a recursive process.

```
class DecisionTree:
    def train(self):
        self.build_trees(self.dummy_root, [record for record in
range(len(self.records))], 'root')
```

Method `self.build_trees` is the recursive function itself.

- Termination: check three termination conditions first
- Select the spliting attribute: compute Information Gain for each attributes and select the one with the largest IG as the spliting attribute
- Split the records: split the records according to the options
- Branch out: create a new node and attach to the parent node
- Recursion: do this recursively for each branch

```
class DecisionTree:
    def build_trees(self, parent_node, data_pool, branch_name):
        trace = parent_node.trace
        num_attr = len(trace)
        # Check termination condition: run out of attributes
        if num_attr == 0:
            # take majority as label
            leaf_node = self.get_majority(data_pool, branch_name)
            parent_node.children.append(leaf_node)
            return
```

```
        # Check termination condition: pure label
        if len(set([self.records[idx][-1] for idx in data_pool])) == 1:
            # assign unique label
            leaf_node = TreeNode(self.records[data_pool[0]][-1], -1, [],
branch_name)
            parent_node.children.append(leaf_node)
            return

        # Compute IG for each available attribute
        IGs = []              # list of IG values for each attribute
        branches = []         # list of branches for each attribute
        branches_names = []   # list of option names for each branch
        # Compute the entropy before branch out
        entropy_before = self.entropy(data_pool)
        # Compute the information gain for each attribute
        for i in range(num_attr):
            IG, branches_pool, branches_name = self.information_gain(trace[i],
data_pool, entropy_before)
            IGs.append(IG)
            branches.append(branches_pool)
            branches_names.append(branches_name)

        # Check termination condition: all the attributes are identical but with
impure labels
        if len(set(IGs)) == 1:
            # take majority as label
            leaf_node = self.get_majority(data_pool, branch_name)
            parent_node.children.append(leaf_node)
            return

        # Select the attribute with the largest IG
        max_IG = 0
        max_index = 0  # max index of IGs
        for index, IG in enumerate(IGs):
            if IG > max_IG:
                max_IG = IG
                max_index = index

        # Update the trace for new node
        new_trace = trace.copy()
        new_trace.remove(trace[max_index])

        # Attach new node to the parent node
        new_node = TreeNode(self.attributes[trace[max_index]], trace[max_index],
new_trace, branch_name)
        parent_node.children.append(new_node)

        # Recursion for each branch
        for branch_pool, name in zip(branches[max_index],
branches_names[max_index]):
            new_node.branch_to.append(name)
            self.build_trees(new_node, branch_pool, name)
        pass
```

Print the Model

My model will be printed out as follows.

```
*- Occupied(root)
   |- Location(High)
   |  |- Yes(City-Center)
   |  |- No(Talpiot)
   |  |- No(German-Colony)
   |  *- Yes(Mahane-Yehuda)
   |- Location(Moderate)
   |  |- Yes(Ein-Karem)
   |  |- Yes(City-Center)
   |  |- Price(Talpiot)
   |  |  |- Yes(Normal)
   |  |  *- No(Cheap)
   |  |- VIP(German-Colony)
   |  |  |- No(No)
   |  |  *- Yes(Yes)
   |  *- Yes(Mahane-Yehuda)
   *- Location(Low)
      |- Yes(Ein-Karem)
      |- Price(City-Center)
      |  |- Yes(Normal)
      |  *- No(Cheap)
      |- No(Talpiot)
      *- No(Mahane-Yehuda)
```

The model has a hierarchical representation from left to right.

- The strings start with `|-` or `*-` are the value of each node. (`*-` indicates nothing but the last node in the current level for the convenience of knowing when to print next level)
- The strings inside the `()` indicate which option that they branch from (or the value of branches)

> The order of nodes in each level may be different each time we run the code. Because I used `set` type to decide the order of branches. But the tree is consistently correct.

Prediction

- Test data:

Occupied	Price	Music	Location	VIP	Favorite Beer
Moderate	Cheap	Loud	City-Center	No	No

- Prediction: Yes

Code Running result

Run `python main.py`, then we will see the following result:

```
******** Model ********
*- Occupied(root)
   |- Location(High)
```

```
   |  |- Yes(City-Center)
   |  |- No(Talpiot)
   |  |- No(German-Colony)
   |  *- Yes(Mahane-Yehuda)
   |- Location(Moderate)
   |  |- Yes(Ein-Karem)
   |  |- Yes(City-Center)
   |  |- Price(Talpiot)
   |  |  |- Yes(Normal)
   |  |  *- No(Cheap)
   |  |- VIP(German-Colony)
   |  |  |- No(No)
   |  |  *- Yes(Yes)
   |  *- Yes(Mahane-Yehuda)
   *- Location(Low)
      |- Yes(Ein-Karem)
      |- Price(City-Center)
      |  |- Yes(Normal)
      |  *- No(Cheap)
      |- No(Talpiot)
      *- No(Mahane-Yehuda)
******** Prediction Begins ********
The traverse path is:
--Occupied
Moderate
--Location
City-Center
--Yes
********* Prediction Ends *********
Prediction: Yes
```

The decision tree model, prediction process and the predition result were printed out, respectively.