COMPSCI 383 – Spring 2022

Homework 6 Coding

Due Wednesday, May 4th at 9pm

You are encouraged to discuss the assignment in general with your classmates, and may optionally collaborate with one other student. If you choose to do so, you must indicate with whom you worked. Multiple teams (or non-partnered students) submitting the same code will be considered plagiarism.

Code must be written in a reasonably current version of Python (>3.6), and be executable from a Unix command line. You are free to use Python's standard modules for data structures and utilities, as well as the pandas, scipy, and numpy modules.

The Trees of Mass (70 points)

For this assignment, you will implement a decision tree classification algorithm. Your code will learn a decision tree from a data set which contains facts about Massachusetts municipalities, with a class label indicating the vaccination level of that city or town.

Data and Supplied Code

The file `decision_tree.py` contains code to get you started. The read_data() function parses the data found in `town_vax_data.csv`, and returns a list of dictionaries representing each town in the data set:

```
{ 'town': 'Amherst', 'median_age_2011': 21.3, … 'growth': 'small' }
{ 'town': 'Becket', 'median_age_2011': 49.3, … 'growth': 'medium' }
{ 'town': 'Chicopee', 'median_age_2011': 40.2, … 'growth': 'small' }
```

The class label `growth` captures the change in population for each town between the 2010 and 2020 census, and is one of four values (`negative`, `small`, `medium`, `large`), while the values of all other attributes are numeric and continuous. For some of the examples, a particular attribute value may be missing and denoted by a `None`.

In addition, there are classes to handle the decision and leaf nodes in the classification tree. `DecisionNode` represents a binary split on a single attribute, while `LeafNode` represents the prediction for each path from the root. These classes are fully implemented, and don't need to be modified.

The `DecisionTree` class represents the model itself. You must implement the `learn_tree()` and `classify()` methods. While you are free (and encouraged) to create additional methods or functions, please do not change the API (method signatures or return values) of any of the supplied code.

To construct the tree, you should use “information gain” as the criteria for selecting attributes to split on (see Section 18.3.4 of AIMA, as well as the included excerpt from Provost & Fawcett). Note that since the attributes are continuous, you’ll need to select a threshold to split on. You can do this by trying all possible thresholds (unique attribute values for the data set) or select some finite number of cutoffs (try at least ten). When training, you should ignore examples that have a missing value for the attribute being considered.

A simple algorithm can produce overfitted trees, which perform well on the training data but not the test data. To limit overfitting, your code should implement a threshold on the minimum number of training examples represented by any leaf node, stored in the field `DecisionTree.min_leaf_count`. If splitting on a particular attribute will result in a leaf node that is below the threshold, you should not use it at that part of the tree.

`DecisionTree.classify()` should return a predicted class label, along with a probability estimate for that label. See the docstrings and comments in the code for more details.

When you execute `decision_tree.py` as a script, it will read in the data, divide it into a training and test set, learn a tree, and test it. Additionally, it will output predictions for each town in the test set, along with the accuracy, a confusion matrix, and a totally sweet ASCII-based diagram of the learned tree.

Hints

- You should implement `DecisionTree.learn_tree()` as a recursive function (or have it call a recursive function). At each level of recursion, you must use a list of examples to identify the best (in terms of information gain) attribute and threshold to split on. Using that splitting criteria, you then divide the examples and perform the same procedure on each subset.
- The constructor for `DecisionNode` expects tree nodes (either `DecisionNode` or `LeafNode` objects) as arguments, so they must be constructed first (via recursion). The `miss_lt` parameter is a boolean that determines the child node to be used to classify examples with a missing value, and should specify whichever subtree has more examples in the training data.
- The `DecisionTree.classify()` should utilize the `classify()` methods defined for the tree nodes.
- You should run whatever tests you need to convince yourself that your code is bug-free. One good testing strategy is to make some very small versions of the dataset where you can predict what trees should result from the training process.
- Since this is a multiclass problem, achieving high accuracy can be difficult. You should expect your solutions to achieve an accuracy ~0.4 (and a within-one-level “almost” accuracy of ~0.9).
- For this assignment, you don’t need to write a lot of code — our solution contains less than a hundred lines for the two methods you must implement. That said, this

assignment can be difficult on a conceptual level. Start early, identify areas where you are having trouble, and ask questions!

Testing and Grading

We will run your program on a variety of test cases based on curated test sets. Your grade will be proportional to the number of test cases you pass. You can assume that the input file will be well formed, though the number of rows and columns may differ from the supplied data file.

What to Submit

You should submit a modified version of `decision_tree.py` containing your code, along with `readme.txt` containing:

- Your name(s)
- Any noteworthy resources or people you consulting when doing your project
- Notes or warnings about what you got working, what is partially working, and what is broken