CSC442 Project 4: Learning

By *Jiupeng Zhang*, December 09, 2018

Part I: Decision Tree Learning

- **Program Design**

  To represent a decision tree in our program, we designed a class ***DTNode***, with three attributes: ***type*** (indicates either a leaf or non-leaf node), ***value***, and a list of references to its children nodes called ***branches***, as *Figure 1* shown below:

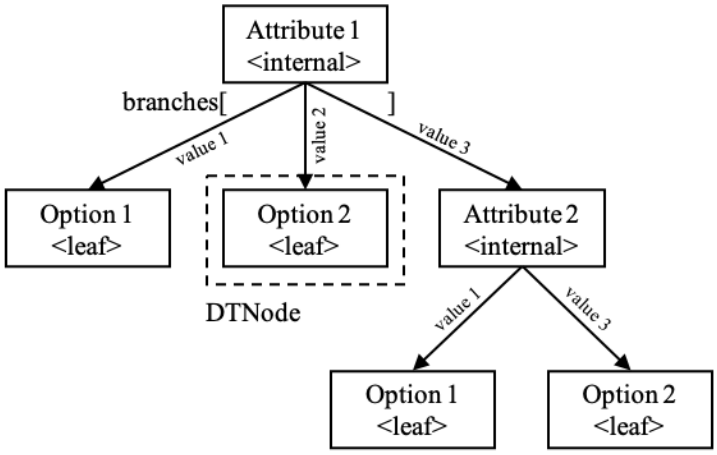

Figure 1: code representation of a decision tree

Before passing the dataset to the program, I transferred them into a specific format to unify loading process for different datasets. One is the file of metadata, the other is data, each represented by a set of node object and a bunch of data. Take the *Restaurant* dataset as an example, the preprocessed metadata nodes and input data are shown below:

(Alternate)			**(Hungry)**			**(Raining)**			Burger	None
	Yes	None		Yes	None		Yes	None		
	No	None		No	None		No	None	**(WaitEstimate)**	
			(Patrons)						0-10	None
(Bar)				None	None	**(Reservation)**			10-30	None
	Yes	None		Some	None		Yes	None	30-60	None
	No	None		Full	None		No	None	>60	None
						(Type)				
(Fri/Sat)			**(Price)**				French	None	**(WillWait)**	
	Yes	None		$	None		Italian	None	Yes	None
	No	None		$$	None		Thai	None	No	None
				$$$	None					

[{'Alternate': 'Yes', 'Bar': 'No', 'Fri/Sat': 'No', 'Hungry': 'Yes', 'Patrons': 'Some', 'Price': '$$$', 'Raining': 'No', 'Reservation': 'Yes', 'Type': 'French', 'WaitEstimate': '0-10', 'WillWait': 'Yes'},
{'Alternate': 'Yes', 'Bar': 'No', 'Fri/Sat': 'No', 'Hungry': 'Yes', 'Patrons': 'Full', 'Price': '$', 'Raining': 'No', 'Reservation': 'No', 'Type': 'Thai', 'WaitEstimate': '30-60', 'WillWait': 'No'},
{'Alternate': 'No', 'Bar': 'Yes', 'Fri/Sat': 'No', 'Hungry': 'No', 'Patrons': 'Some', 'Price': '$', 'Raining': 'No', 'Reservation': 'No', 'Type': 'Burger', 'WaitEstimate': '0-10', 'WillWait': 'Yes'},
{'Alternate': 'Yes', 'Bar': 'No', 'Fri/Sat': 'Yes', 'Hungry': 'Yes', 'Patrons': 'Full', 'Price': '$', 'Raining': 'Yes', 'Reservation': 'No', 'Type': 'Thai', 'WaitEstimate': '10-30', 'WillWait': 'Yes'},
{'Alternate': 'Yes', 'Bar': 'No', 'Fri/Sat': 'Yes', 'Hungry': 'No', 'Patrons': 'Full', 'Price': '$$$', 'Raining': 'No', 'Reservation': 'Yes', 'Type': 'French', 'WaitEstimate': '>60', 'WillWait': 'No'},
{'Alternate': 'No', 'Bar': 'Yes', 'Fri/Sat': 'No', 'Hungry': 'Yes', 'Patrons': 'Some', 'Price': '$$', 'Raining': 'Yes', 'Reservation': 'Yes', 'Type': 'Italian', 'WaitEstimate': '0-10', 'WillWait': 'Yes'},
{'Alternate': 'No', 'Bar': 'Yes', 'Fri/Sat': 'No', 'Hungry': 'No', 'Patrons': 'None', 'Price': '$', 'Raining': 'Yes', 'Reservation': 'No', 'Type': 'Burger', 'WaitEstimate': '0-10', 'WillWait': 'No'},
{'Alternate': 'No', 'Bar': 'No', 'Fri/Sat': 'No', 'Hungry': 'Yes', 'Patrons': 'Some', 'Price': '$$', 'Raining': 'Yes', 'Reservation': 'Yes', 'Type': 'Thai', 'WaitEstimate': '0-10', 'WillWait': 'Yes'},
{'Alternate': 'No', 'Bar': 'Yes', 'Fri/Sat': 'Yes', 'Hungry': 'No', 'Patrons': 'Full', 'Price': '$', 'Raining': 'Yes', 'Reservation': 'No', 'Type': 'Burger', 'WaitEstimate': '>60', 'WillWait': 'No'},
{'Alternate': 'Yes', 'Bar': 'Yes', 'Fri/Sat': 'Yes', 'Hungry': 'Yes', 'Patrons': 'Full', 'Price': '$$$', 'Raining': 'No', 'Reservation': 'Yes', 'Type': 'Italian', 'WaitEstimate': '10-30', 'WillWait': 'No'},
{'Alternate': 'No', 'Bar': 'No', 'Fri/Sat': 'No', 'Hungry': 'No', 'Patrons': 'None', 'Price': '$', 'Raining': 'No', 'Reservation': 'No', 'Type': 'Thai', 'WaitEstimate': '0-10', 'WillWait': 'No'},
{'Alternate': 'Yes', 'Bar': 'Yes', 'Fri/Sat': 'Yes', 'Hungry': 'Yes', 'Patrons': 'Full', 'Price': '$', 'Raining': 'No', 'Reservation': 'No', 'Type': 'Burger', 'WaitEstimate': '30-60', 'WillWait': 'Yes'}]

As you see, we converted the *AIMA_Restaurant-desc.txt* into unconnected nodes. For each node, the ***type*** is an identifier to indicate if a node is a leaf, the ***value*** of a node is the name of its attribute (notice that the leaf nodes are decisions, not an attribute name, so the meaning of the field *value* are different), and the ***branches*** field is a map of children names to their node pointers. At the start, all branches are named (but not loaded) by attribute values, and all its values are set to *None*, that is to say, the mission of the program is to build the decision tree by assembling the attribute nodes.

- **Concepts and Algorithms**

  In my opinion, the main idea of the decision tree learning algorithm is to construct nodes by dividing examples (by attribute value) and then sum up decisions based on a specific sequence. The core implementation is listed below. Please notice that the argument ***query*** is required for this function to exclude the query node (a.k.a. the decision node/label) from the passed in examples and count the corresponded examples.

  However, since the sequence of nodes will affect the depth of the decision tree (e.g. choosing a lower bias attr_node increases the depth but is less helpful in making decisions) and then affects its evaluation performance, strategies are needed to ensure that we access the most important attribute node each time.

  ID3, an entropy-based attribute selection method was introduced to solve the above issue. First, we measure the amount of uncertainty over decisions on given examples ***H(S)***:

$$\mathrm{H}(S) = \sum_{x \in X} -p(x) \log_2 p(x)$$

  The *p(x)* is the proportion of each decision over the total number of examples.

Then, we calculate the ***Information Gain - IG(S, A)*** on that node's attributes to quantify the effect of the split on each attribute (weighted by the proportion of subset entropies) based on the following equation:

$$IG(S, A) = \mathrm{H}(S) - \sum_{t \in T} p(t)\mathrm{H}(t) = \mathrm{H}(S) - \mathrm{H}(S|A).$$

Finally, we choose the attribute with the maximum information gain as our next division node.

Furthermore, another problem will be raised when examples not enough to make decisions. This situation will occur in two cases:

1. All attributes are used but examples are still not fully divided.
2. No examples are provided under current condition.

To ensure our program making reasonable decisions even with insufficient examples, rather than throw an empty data exception, we introduced the ***plurality function***. My implementation on this method was just choosing the most common decision label under partial of given examples.

```
def Decision_Tree_Learning(query, examples, attr_nodes, parent_examples):
    if not examples: return plurality_leaf(query.value, parent_examples)
    elif homo(query.value, examples): return DTNode(examples[0][query.value])
    elif not attr_nodes: return plurality_leaf(query.value, examples)
    else:
        attr_index = np.argmax(list(map(lambda n: importance(query, n, examples), attr_nodes)))
        node = attr_nodes[attr_index]
        del attr_nodes[attr_index]
        for k in node.branches.keys():
            exs = extract_examples(node.value, k, examples)
            subtree = Decision_Tree_Learning(query, exs, attr_nodes, examples)
            node.branches[k] = subtree
        return node
```

Figure 2: pseudo code of the decision tree learning algorithm

- **Implementation**

  This section will just show basic test cases of our implementation, further discussion on dataset preparing, performance and comparison will be covered in chapter three.

  Besides, the ***evaluate*** method for testing/predicting is completed, but for concise, only generated decision trees are printed below:

Testcase #1: *AIMA Restaurant*
$ **python3 decision_tree_generator.py data/AIMA_Restaurant-desc.txt 1**

```
(Patrons)
        None    [No]
        Some    [Yes]
        Full    (Hungry)
                Yes     (Type)
                        French  [Yes?]
                        Italian [No]
                        Thai    (Fri/Sat)
                                Yes     [Yes]
                                No      [No]
                        Burger  [Yes]
                No      [No]
```

Testcase #2: *Iris Classifier*
$ **python3 decision_tree_generator.py data/iris.data.discrete.txt 1**

```
(Petal width)
        S       [Iris-setosa]
        MS      [Iris-versicolor]
        ML      (Petal length)
                S       [Iris-versicolor?]
                MS      [Iris-versicolor]
                ML      (Sepal length)
                        S       [Iris-virginica]
                        MS      (Sepal width)
                                S       [Iris-versicolor?]
                                MS      [Iris-versicolor?]
                                ML      [Iris-versicolor]
                                L       [Iris-versicolor?]
                        ML      [Iris-versicolor?]
                        L       [Iris-versicolor]
                L       [Iris-virginica]
        L       [Iris-virginica]
```

Testcase #3: *Tic-Tac-Toe Game Judger (link)*
$ **python3 decision_tree_generator.py data/tic-tac-toe.data.txt 1**

This dataset encoded a set of possible endgames, where 'x' is assumed to have played first.
p_x_y: the state on board[x][y] (1-index), can pick either 0 (empty), 1 (a 'x') or 2 ('o' in cell)

```
(p_2_2)
        0       (p_1_1)
                0       (p_3_3)
                        0       [X_Wins?]
                        1       [X_Wins]
                        2       [X_Not_Win]
```

```
                1       (p_1_3)
                        0       (p_3_1)
                                0       [X_Wins?]
                                1       [X_Wins]
                                2       [X_Not_Win]
                        1       (p_1_2)
                                0       (p_2_1)
                                        0       [X_Not_Win]
                                        1       (p_2_3)
                                                0       [X_Not_Win]
                                                1       [X_Wins?]
                                                2       [X_Wins]
                                        2       [X_Wins]
                                1       [X_Wins]
                                2       (p_3_2)
                                        0       [X_Wins]
                                        1       [X_Wins?]
                                        2       [X_Wins?]
                        2       [X_Wins?]
                2       [X_Wins?]
        1       [X_Wins?]
        2       [X_Not_Win?]
```

- **Performance Analysis**

  To evaluate the performance of the generated decision tree, we plot a figure to display the accuracy of the model as the ratio of $\frac{\text{training set size}}{\text{testing set size}}$, as shown in Figure 3.

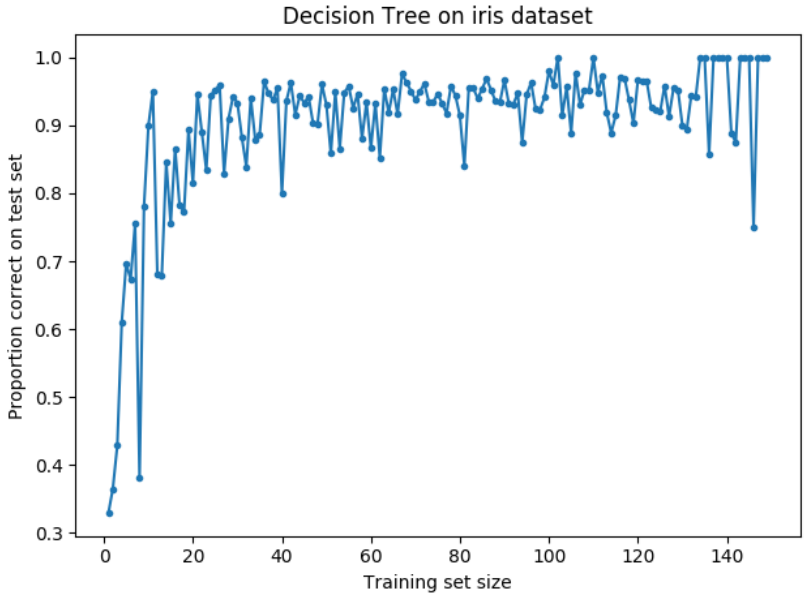

Figure 3. The proportion correct on test set as training set size grows

We have 150 rows of data in the Iris dataset, so we trained 150 models with different partial of training data fed (step=1), and the remained data are for test use. As we see, the correctness is generally increasing when more training data are fed. At the start, the model performed badly because it was arbitrarily making decisions (only a few parts of the training data were learned), then, the correctness rapidly increased in a relatively small period. After that, an oscilloscope occurred at the right side of the diagram, this is because the testing set is too small, and it is not objective enough to represent the accuracy of the whole model.

Part II: Neural Networks

- **Network Design**

  We chose two datasets (*Iris, Tic-Tac-Toe Endgame*) for both the decision tree and the neural network classifiers. In fact, since the datasets are different, our structure on networks are slightly tuned.

  i. Layer sizes

  As the Iris classifier, we use a simple fully connected three-layer structure. In fact, as we change the size of each layer, the arrangement of layer-size [4, 8, 3] performed the best during our tests.

  For the Tic-Tac-Toe classifier, we designed a double hidden layer [9, 27, 9, 2] structure, we expect the model learned the value of position (if 'x', 'o', '_' is at each place) for the first hidden layer and conclude the game status for each position in the second hidden layer, and the output layer indicates whether 'x' wins or not.

  ii. Loss function

  The loss function evaluates the bias between the output of the model as the expected result, thus, the quality of the loss function will significantly affect the performance of a classifier, in our experiment, two loss functions were considered in comparison: Mean Square Error (MSE) for the Iris classifier, and the Binary Cross Entropy (BCE) for the Tic-Tac-Toe classifier.

  iii. Learning rate (α)

  We tried to train the model multiple times under different learning rates and verified that, as the learning rate rises, the total loss dropped down faster, but that does not mean the more the better. If a learning rate was set too large, the loss will drop quickly at the start, however, the closer we reach the local minimum, the harder it will converge at the local minima because of the fixed high learning rate. See in *Figure 4*, the green line was a training curve with a learning rate of 0.9, as the number of training increases, oscilloscope occurs (a.k.a. That is to say, if the weights of our model are currently close to a local minimum, the loss curve will diverge because the high learning rate prevents the model from approaching the optima.

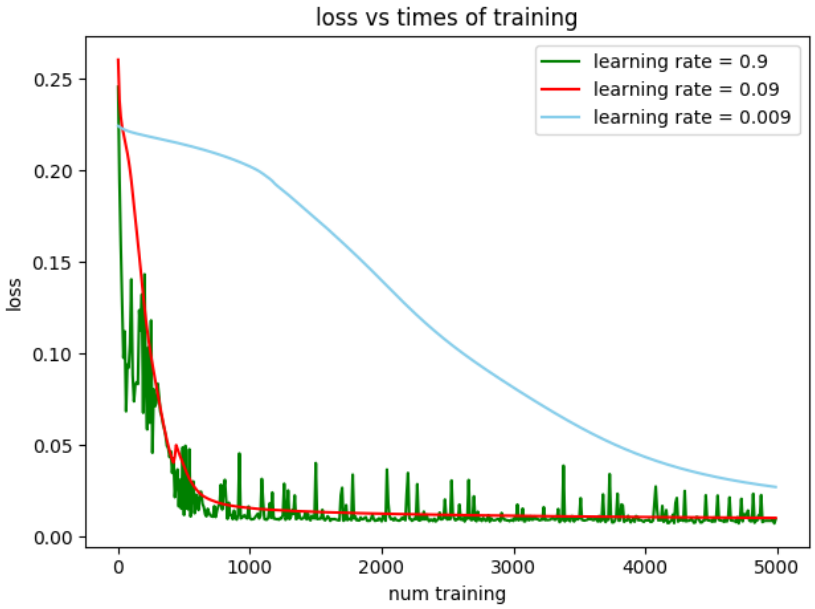

Figure 4. The descent of loss under different learning rates

iv. Activation Functions

To avoid the gradient vanishing issue caused by the sigmoid function, and for a faster speed in training, we use the ***ReLU*** activation for hidden layers in our model, besides, as the model is a classifier, we use the ***SoftMax*** activation for a "soft" classification in our output layer.

v. Initialization of weights and bias terms

To ensure us having a better startup at the beginning of each training, we need to normalize the weight matrix to some number around zero to break the symmetry between neurons in each layer.

vi. Batch size for stochastic/batch/minibatch

I tried three different configurations on batch size for our network: 5, 15, 30, and the speed of gradient descent varies when batch size changed, as shown in *Figure 5* below:

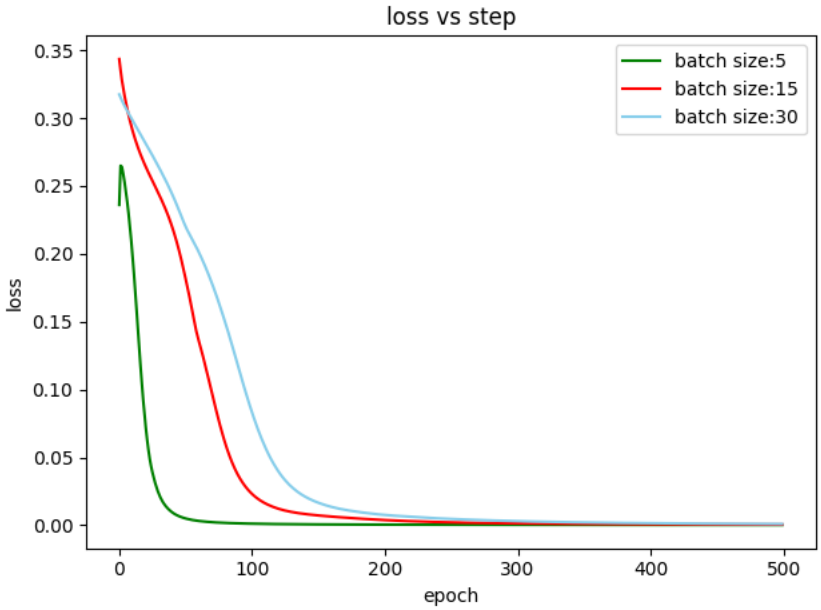

Figure 5. The descent of loss as iteration count increases

We notice that the speed of descending on the loss performed faster when we have a smaller batch size (compare the green line with the red or blue line in the graph). The above figure is on the model for the Iris dataset, use all of the given data for training, we know that each epoch will have 150 iterations. After about 20 iterations, we find a small peek on the green line (with a batch size of 5), which indicates that we are unable to ensure the monotonic after each iteration because we updated our model based on the gradient calculated by partial training data (size < epoch length). However, it will finally converge for the reason that the whole training data will be fed after an epoch, so it is normal to have some hops on loss during iterations.

Part III: Decision Tree vs. Neuron Network

We applied the ***Kfold*** (*k=2,5,10*) algorithm to compare the implemented two models based on the Iris dataset (*n=150*) and calculated the accuracy of them:

K	**Decision Tree**	**Neuron Network**
2	0.9040725715573585	0.5422222222222226
2	0.9067615945514496	0.5511111111111114
5	0.9176504976504976	0.9466666666666667
5	0.9188409738409739	0.9777777777777779
10	0.8830952380952383	0.968888888888889
10	0.9196031746031746	0.9555555555555557
Avg.	0.908337342	0.823703704

Part IV: Extra Works

I personally tried to implement a neural network from the scratch but faced a numerical problem sometimes when calculating the derivative of activation and loss functions. Although it is now buggy, to declare my work, I attached the code “mlp_classifier.py” for your review.