4)

B: how the code works: There are two relevant files: id3.py (the driver) and node.py (provides the two classes that form the tree structure). The driver reads in the Tennis and Art datasets. For both sets, it builds a decision tree iteratively, one node at a time, recording the errors on the training and testing sets as well as the monotonically increasing number of non-leaf nodes. For the Art dataset, it does 20 different iterations of the ID3, where it builds the tree and then prunes it iteratively (each iteration generates a new validation set randomly).

As for node.py, there is the basic Tree structure with which the user interfaces. It possesses all the metadata (e.g. perform cross-validation or not) and further sets up the problem by identifying the attributes and all their possible values. It maintains pointers to all of its leaves to make its display method easier (draws all possible paths to leaves) as well as candidates for possible pruning (i.e. those nodes whose children are all leaves). It also stores a FIFO queue for node exploration/expansion. Expansion ends when all leaves are pure or there are no more attributes to split on. Lastly, it has a pointer to the root node through which the evaluate function is directed in order to determine the performance of the tree on a particular dataset.

As you could guess, the other structure is the Node class. A node contains a lot of information. A node is represented as the child with a particular value belonging to the attribute on which its parent was split on. (It maintains a pointer to its parent.) And for non-leaf nodes, it is also represented by the attribute on which it itself is split. The node is responsible for expanding (if possible) following the information gain heuristic for choosing attributes. If it is a leaf node, then evaluation will return the plurality of instances that fall under it. If not, evaluation will then be passed on to the appropriately-valued child. It maintains the path of how to get from the root to the node. Every node has a pointer back to the singular Tree object that allows each to update globally relevant information (e.g. list of leaf nodes and pruning candidates).

C: talk about how there weren’t enough instances to make a meaningful chart; but still can see patterns we’ve discussed in class; \*\*\*interesting to look at paths --- might want to share\*\*\*

D: error plot: can see pattern of testing set error increasing as iterations increase and overfitting occurs

* Feel free to incorporate paths here too (maybe share one?)

E: explanation of pruning code: the user (i.e. id3.py) invokes tree.prune() (which returns 0 if node was removed and -1 if no node should be removed). This then iterated through all potential pruning candidates. This list is dynamically updated by the nodes to ensure that a node is a candidate iff every one of its children is a leaf node; this is done so that pruning starts from the bottom and works upwards. It temporarily prunes each candidate, evaluating the performance of the validation set on this temporary tree. It “unprunes” the candidate and moves to the next one. It then chooses the node which performs the best, removing any from the list that don’t improve performance. If no node improves performance, then it returns -1 signaling the user to stop pruning. Note: each successful pruning reduces the number of non-leaf nodes by 1.

F: out of the 20 validation iterations, few actually yielded any pruning (i.e. most only had candidates that didn’t improve performance)…compare errors between two or more runs; \*\*\*\*look at run18 for example of two rounds of successful pruning; be sure to answer question posed in Part 3. (4)