CS480: Assignment 04 Report
Zachary Ferguson

Design and Implementation:

The algorithms used to implement the decision tree and supporting functions come directly from the lecture notes. Using the pseudo-code to implement the functions in LISP required some thought, however. Reading the pseudo-code it is obvious how one might implement the function in a C-like programming language. This C code could have been reworked into a similar looking LISP version, but some thought is required to utilize LISP's full capabilities. Attempting to utilize mapcar and list building functionalities, I opted for a more in-line solution using lambda expressions. This helped me produce concise easy to follow code.

For Part C, I had difficulty finding a data set that I could easily mapover to create examples. The provided data sets include both categorical and real values, creating a problem for my code. Because, by default, the assignment does not require the handling of floating-point values, this limited the data sets I could use. Two solutions to this are to ignore the floating-point attributes or to convert them to a categorical value in terms of their mean. For simplicity, I choose the former to create usable examples. In addition, I used the UCI Machine Learning Repository to get a data set of purely categorical attributes. I choose to utilize the Balloon Data Set, which came in handy when testing the resulting decision-trees.

Reflection:

This project helped me to better understand the functions behind building a decision tree. The impurity functions are straight forward mathematical summations, but the remainder function was a slight mystery going into the project. My confusions included: what are the inputs to the impurity function, and what is the X_j subset. Implementing the remainder function help me to understand that the impurity function takes the probability of each label in X_j and the X_j subset is the set of all samples with the feature value j.

Implementing the decision tree helped me to better understand the construction and functionality of decision trees. As the pseudo-code indicates, the tree is best built use a recursive method. This an exploit of the recursive nature of decision trees, each subtree is a decision tree. This fact became obvious as I implemented the build-decision-tree function.

Analyzing the Results:

The previously mentioned balloon data set came in handy when analyzing the output decision trees. Provided with each data set was a descriptor for when the label is true (ex. adult-stretch.data Inflated is true if age=adult or act=stretch). This was extremely useful because I could check that my decision trees matched this expected behavior. In essence, I could treat them as unit-tests for my project.

Using the provided function, find-label-for-example, I was also able to test the trees produced to make sure the output labels matched that of the training examples. Similar to Part B but using only one example set, I was able to ensure that for the training set the decision trees always produced the correct label. This test in practice has no realistic purpose, but it is a good test to make sure the decision tree works to some degree.