CART Decision Tree Model

* The representation of the CART model is a binary tree. So each node can have xero, one or two child nodes.
* A node here represents a single attribute or input value (X) and a split point on that variable, assuming that variable is numeric. The leaf nodes or terminal nodes contain the output variable (Y) used to make a prediction.
* Once created, a tree can be navigated with a new row of data following each branch with the splits until a final prediction is made.
* Hence we can look at a binary tree as a process of dividing the input space.
* We use a greedy approach called recursive splitting to create our binary tree. This is a numerical procedure in which all the values are lined up and different split points are judged on the basis of a cost function. The split with the best or minimum cost function is selected.
* The cost function depends on the kind of model we are building:
  + Regression: the cost function is the sum squared errors across all training samples.
  + Classification: the Gini cost function, that refers to the purity of the node wherein purity means how mixed the training data assigned to each node is.
* The Gini Index:
  + A gini score gives an idea of how good the split is. A perfect separation would result in a gini score of 0 whereas a worst case scenario of a 50/50 classes in each group would mean that the gini score is 0.5.
  + We first calculate the proportions or the number of observations of a class in each group. (p) This is then used to calculate the score by weighting it with the size of the groups relative to all samples in the dataset.
* We then need to develop a function to creat splits in the dataset. We do that and then evaluate all splits.
* Splitting a dataset essentially means separating a dataset into two lists of rows given the index of an attribute and a split value for that attribute. Once we have these two groups, we use the best gini index to adjudge the index and split value that suits the data best.
* Creating the root node of the tree is easy. We call our get\_split() function on the entire dataset.
* Building a tree is divided into 3 main parts:
* Terminal nodes
  + We need to decided when to stop growing a tree. We do that using the depth and the number of rows of the training dataset that the node is responsible for.
  + Max tree depth: max number of nodes from the root node. More the nodes, more complex the tree is and more chances of overfitting.
  + Min node records: the number of training rows a node is responsible for. Once at or below it, we stop splitting. Nodes that account for too few training patterns are too specific and may overfit the data.
  + There is one more condition: when we choose a split in which all rows belong to one group. In this case, we will be unable to continue splitting and adding child nodes as we will have no records to split on one side or another.
  + Once we have a terminal node, we select the most common class value in the group and use that to make our prediction.
* Recursive splitting
  + Building a decision tree means calling the get\_split() function over and over again on the groups created for each node.
  + New nodes added to existing nodes are called child nodes.
  + Firstly, the two groups of data split by the node are extracted for use and deleted from the first node. As we work on these groups, the node no longer requires access to these data.
  + Next, we check if either left or right group of rows is empty and if so, we creat a terminal node using our function to\_terminal().
  + We then check for max\_depth reached. If it has been reached already, we create a terminal node.
  + We then process the left child, creating a terminal node if group of rows is too small and processing it further to get\_split() and recursively adding the same function with this node and depth+1 as the argument.
  + Similarly we repeat the above procedure for the right node as well.
* Building a tree
  + This involves creating the root node and calling then calling the split function that then calls itself recursively to build out the whole tree.
* Make a prediction
  + This invloves navigating the tree with the specifically provided row of data.
  + We can use a recursive function where the same prediction routine is used with the left and the right node, depending on how the split affects the provided data.
* Once we write the functions to calculate our requirements for the decision tree algorithm, we write a function to build the tree and predict on a dataset.
* We are using our own function to load in the csv and return the data in list format since that is how we have built our functions.
* We are using k-fold cross-validation to improve the acuracy of our model.
* We write a function to calculate the accuracy metric by looking at the difference between accurate and inaccurate predictions on the test set.
* Finally, we use all the above written functions to return to us the accuracy scores of our decison tree algorithm based on a k-fold cross validation strategy.