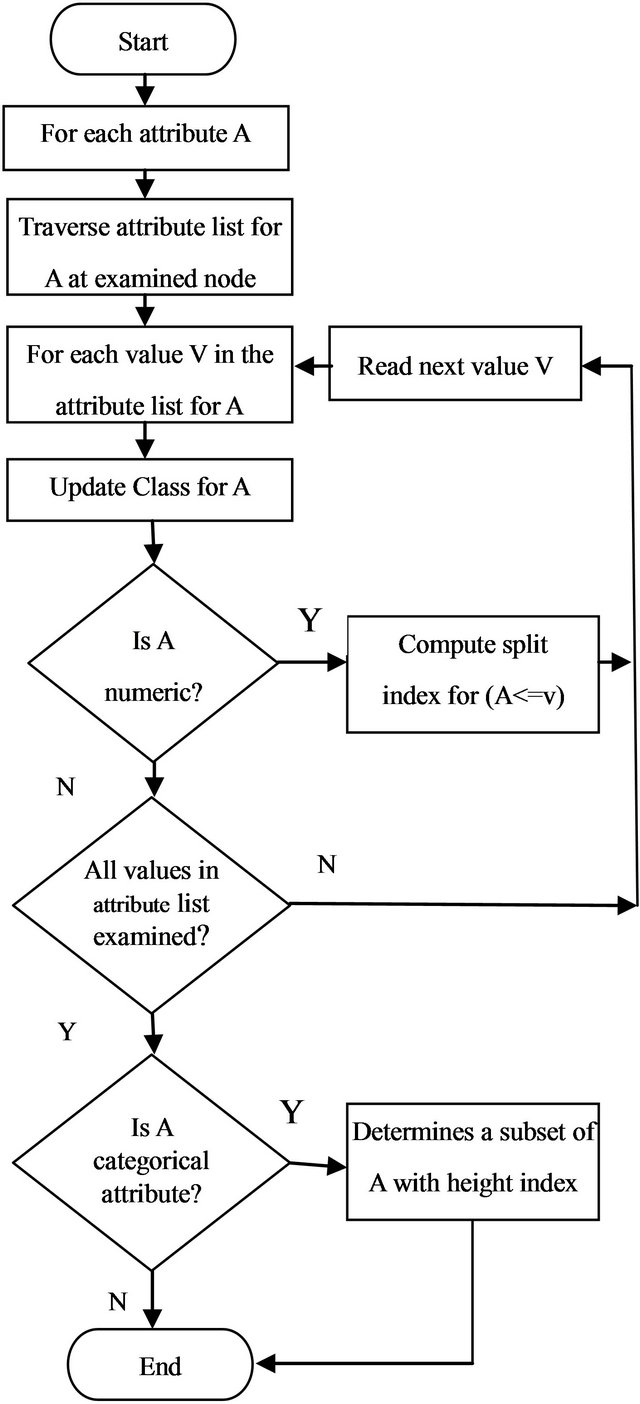
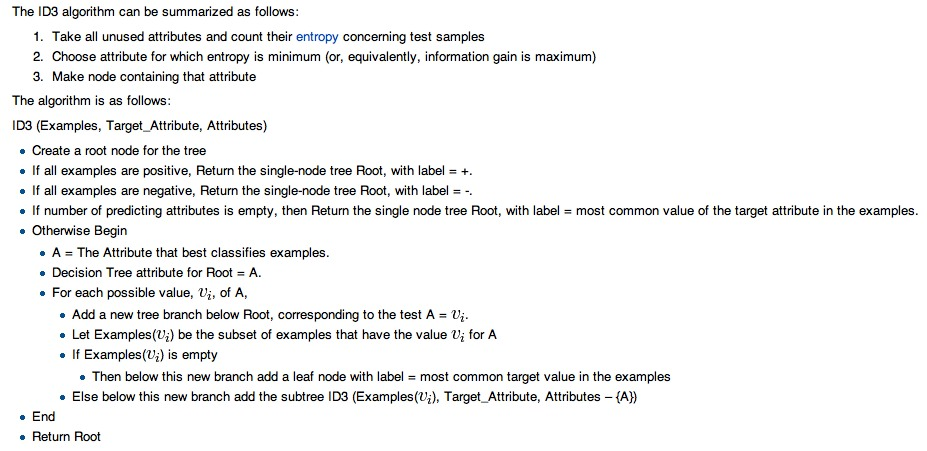
DECISION TREE

**Flowchart**



**Pseudo code**

# Building a Decision Tree in Python

Data structure to represent the tree

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Tree:  def \_\_init\_\_(self, parent=None):  self.parent = parent  self.children = []  self.splitFeature = None  self.splitFeatureValue = None  self.label = None |

In particular, now that features can have more than two possible values, we need to allow for an arbitrarily long list of child nodes. In addition, we add three pieces of data (with default values None): the splitFeature is the feature for which each of its children assumes a separate value; the splitFeatureValue is the feature assumed for its parent’s split; and the label (which is None for all interior nodes) is the final classification label for a leaf.

Data representation

Data is represented as a set list of pairs of the form (point, label), where the point is itself a list of the feature values, and the label is a string.

Now given a data set the first thing we need to do is compute its entropy. For that we can first convert it to a distribution (in the sense defined above, a list of probabilities which sum to 1):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def dataToDistribution(data):  ''' Turn a dataset which has n possible classification labels into a  probability distribution with n entries. '''  allLabels = [label for (point, label) in data]  numEntries = len(allLabels)  possibleLabels = set(allLabels)    dist = []  for aLabel in possibleLabels:  dist.append(float(allLabels.count(aLabel)) / numEntries)    return dist |

And we can compute the entropy of such a distribution:

|  |  |
| --- | --- |
| 1  2  3 | def entropy(dist):  ''' Compute the Shannon entropy of the given probability distribution. '''  return -sum([p \* math.log(p, 2) for p in dist]) |

In order to compute the gain of a data set by splitting on a particular value, we need to be able to split the data set. To do this, we identify features with their index in the list of feature values of a given data point, enumerate all possible values of that feature, and generate the needed subsets one at a time. In particular, we use a Python generator object:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def splitData(data, featureIndex):  ''' Iterate over the subsets of data corresponding to each value  of the feature at the index featureIndex. '''    # get possible values of the given feature  attrValues = [point[featureIndex] for (point, label) in data]    for aValue in set(attrValues):  dataSubset = [(point, label) for (point, label) in data  if point[featureIndex] == aValue]    yield dataSubset |

Entropy Gain

So to compute the gain, we simply need to iterate over the set of all splits, and compute the entropy of each split. In code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def gain(data, featureIndex):  ''' Compute the expected gain from splitting the data along all possible  values of feature. '''    entropyGain = entropy(dataToDistribution(data))    for dataSubset in splitData(data, featureIndex):  entropyGain -= entropy(dataToDistribution(dataSubset))    return entropyGain |

The best split (represented as the best feature to split on) is given by such a line of code as:

bestFeature = max(range(n), key=lambda index: gain(data, index))

Base case will be when we run out of data to split; that is, when our input data all have the same classification label. To check for this we implement a function called “homogeneous”

|  |  |
| --- | --- |
| 1  2  3 | def homogeneous(data):  ''' Return True if the data have the same label, and False otherwise. '''  return len(set([label for (point, label) in data])) <= 1 |

|  |  |
| --- | --- |
|  |  |

Here we see the base cases, and the selection of the best feature to split on. As a side remark, we observe this is not the most efficient implementation. We admittedly call the gain function and splitData functions more often than necessary, but we feel what is lost in runtime speed is gained in code legibility.

Once we bypass the three base cases, and we have determined the right split, we just do it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def buildDecisionTree(data, root, remainingFeatures):  ''' Build a decision tree from the given data, appending the children  to the given root node (which may be the root of a subtree). '''  # add child nodes and process recursively  for dataSubset in splitData(data, bestFeature):  aChild = Tree(parent=root)  aChild.splitFeatureValue = dataSubset[0][0][bestFeature]  root.children.append(aChild)  buildDecisionTree(dataSubset, aChild, remainingFeatures - set([bestFeature]))  return root |

Here we iterate over the subsets of data after the split, and create a child node for each. We then assign the child its corresponding feature value in the splitFeatureValue variable, and append the child to the root’s list of children.

Now the first call to this function requires some initial parameter setup, so we define a convenience function that only requires a single argument: the data.

|  |  |
| --- | --- |
| 1  2 | def decisionTree(data):  return buildDecisionTree(data, Tree(), set(range(len(data[0][0])))) |

Classifying New Data

The last piece of the puzzle is to classify a new piece of data once we’ve constructed the decision tree. This is a considerably simpler recursive process. If the current node is a leaf, output its label. Otherwise, recursively search the subtree (the child of the current node) whose splitFeatureValue matches the new data’s choice of the feature being split. In code,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def classify(tree, point):  ''' Classify a data point by traversing the given decision tree. '''    if tree.children == []:  return tree.label  else:  matchingChildren = [child for child in tree.children  if child.splitFeatureValue == point[tree.splitFeature]]    return classify(matchingChildren[0], point) |

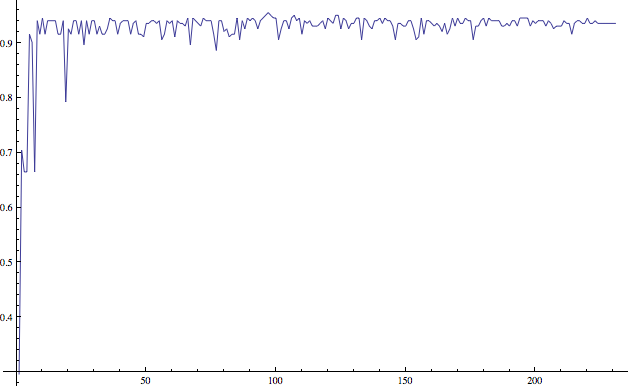
|  |  |
| --- | --- |
|  |  |

Using the Data Set

Our goal is to learn party membership based on the voting records. This data set is rife with missing values, roughly half of the members abstained from voting on some of these measures. So we constructed a decision tree from the clean portion of the data, and use that to classify the remainder of the data.

|  |  |
| --- | --- |
| 1  2  5  6  7  8  9 | with open('house-votes-1984.txt', 'r') as inputFile:  lines = inputFile.readlines()  data = [line.strip().split(',') for line in lines]  data = [(x[1:], x[0]) for x in data]  cleanData = [x for x in data if '?' not in x[0]]  noisyData = [x for x in data if '?' in x[0]]  tree = decisionTree(cleanData) |

Output



The size of the subset used to construct the tree versus its accuracy in classifying the remainder of the data. Note that the subsets were chosen uniformly at random without replacement. The x-axis is the number of points used to construct the tree, and the y-axis is the proportion of labels correctly classified.

Inspecting the trees generated in this process, it appears that the most prominent feature to split on is the adoption of a new budget resolution. Very few Democrats voted in favor of this, so for many of the random subsets of the data, a split on this feature left one side homogeneously Republican.