Decision Tree Classifier

Team: Vivek Tiwari, Caleb Andrade

Final project CSE 537

December 11th 2015, Stony Brook University

1 Code implementation

The Forest.py is divided into four sections

- Helper functions
- Node class object
- Decision Tree function (ID3 algorithm)
- Forest class object

Following is a brief description of each of this sections

1.1 Helper functions

- **ReadFile**. This method reads the 'crx.data.txt' file and creates a list of instances, each instance is itself a list of attribute values. Categorical values are parsed to string, numerical data to float.
- **ConvertInstance**. In order to query an instance in the decision tree, we need to convert those values for numerical attributes to boolean values. The criteria is based on comparing the numerical value to a given threshold t, if this value is less than t then this value is replaced by $True$, otherwise is replaced by $False$. This thresholds are calculated by the **setThreshold** method.
- **Copy**. Creates a copy of a list of instances.
- **signCounter**. This method counts the numbers of + and − signs in a data subset, considering only those instances that have a specific value for some attribute. This will be useful when calculating empirical distributions for purposes of computing entropy.
- **priorDist**. Computes the empirical distribution of + and − signs in a data set.
- **candidThreshold**. In Tom Mitchell's ML book is described a method to convert numerical values to boolean. One of the steps is to sort the instances in a data set according to the numerical attribute under consideration. Then, we look at those values where there is a change of classification (switch from + to − or viceversa). The average in between those numerical values where the switch happens is saved in a list of *threshold candidates*.
- **setThreshold**. After applying the previous method, this method takes each candidate as a threshold and assigns boolean values to the values of the numerical attribute under consideration, then, the information gain is calculated by splitting on this attribute. This is done for every candidate, the candidate that maximizes the information gain is then selected as the threshold. For more details see Tom Mitchell's ML.
- **entropy**. Compute the entropy over a set of probabilities (empirical probabilites), given by $-\sum p_i \log_2 p_i$.
- **condEntropy**. Computes the conditional entropy of a data set classification for a specific attribute value. Calls **entropy** on the probability given by counting the number of + and − signs with **signCounter** and dividing by the number of instances in such data set.

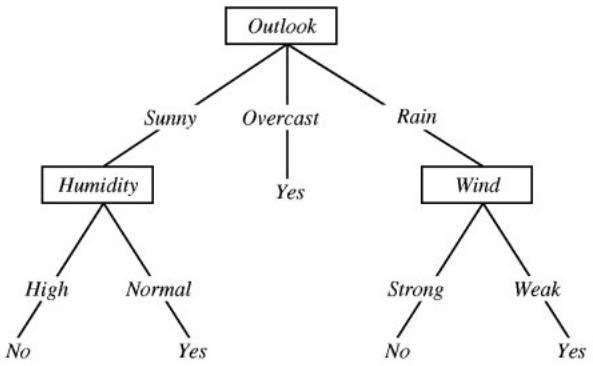
- **attValues**. This method retrieves the values that an attribute actually takes in a given data subset. This method is used to calculate the expected information gain.
- **expGain**. This method computes the expected information gain conditioning on a specific attribute: $IG(S) = H(S) - \sum P(Attribute = value_i) H(S \mid Attribute = value_i)$. This value is considered to choose the best attribute to split data when constructing the ID3 algorithm.
- **missingValue**. Following Mitchell's approach when data is missing, this method looks at those values within the data subset, for the same attribute of the missing value, and selects from them according to their empirical distribution, that is, with probability $p = \frac{value_i \ occurrences}{size \ of \ subset}$.
- **completeValues**. This method sweeps data set once in search for missing values and assigns them a value according to **missingValue**.
- **numToBool**. This method applies the numerical-to-boolean conversion to all the values of a specified numerical attribute in a data set.
- **testSign**. Verifies if all instances have the same classification in a data subset (100% of purity, 0 entropy).
- **getInstances**. Given an attribute and a specific value for it, this method returns all instances in a data subset with the same attribute's value.
- **mostCommon**. While building the decision tree, it might be possible that an attribute doesn't take all its possible values in a given data subset, so a *leaf* node is created for those *non-realized* values taking the most common classification in the given subset.

1.2 Node class

This class was created to support the construction of a decision tree. It stores the basic information of a node such as

- **Node type**. A node can be *internal* when it represents an attribute to split data. A node is a *leaf* when a data subset is 100% classified (either all signs are + or −). Queries end always in a leaf and the sign value is returned.
- **Sign**. This is the *leaf's* classification.
- **Value**. This is in fact the *branch value*, that is, parent nodes have children nodes, where the edge represents one of the values that the internal node's attribute takes. This value is what we call the branch value.
- **Attribute.** When a node is internal, it represents an attribute on which data is splitted, according to the values that it takes in the respective data subset.

Other standard methods are implemented to handle the basic operations during the decision tree creation, that are self explanatory by their names: **setAttribute, setSign, setValue, setNodeType, setChildren, addChildren, getAttribute, getSign, getValue, getNodeType, getChildren, toString**. A classical example is given below



Also, a query function **query** is a recursive method to search in the tree for a given instance's classification. It searches down the tree according to the attributes' values of the queried instance until it reaches a leaf and returns the respective classification. **testAccuracy** computes the percentage of *hits* that a decision tree has on some validation data (not used during training), using the **query** method.

1.3 Decision tree (ID3)

In this section the **decisionTree** function builds recursively a tree by using the class node and the methods described in section 1.2. We follow Tom Mitchell's algorithm pseudocode description on ID3

ID3(*Examples*, *Target_attribute*, *Attributes*)
Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = −
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for A
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *Target_attribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, *Target_attribute*, *Attributes* − {A}))
- End
- Return *Root*

1.4 Forest class

A forest is a collection of decision trees built on the same data set, but in a convenient way. In our implementation we followed the next guidelines to construct this forest object

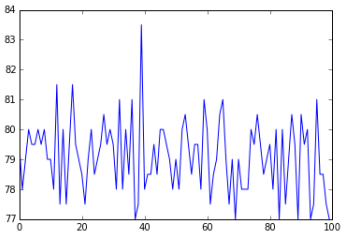
- We first splitted our original data set into 2/3 and 1/3, the first partition for training purposes, the second for validation. During initialization we take care of converting numerical to boolean and also fill in any missing value.
- To create each tree we sample uniformly 2/3 of the first 2/3 of instances (4/9 of the total data), each time building a different decision tree due to a different sampling, and validating its accuracy on the remaining 1/3 of the initial 2/3 (2/9 of the total data). This is done to address the problem of overfitting, as have been suggested by authors such as Tin Kam Ho.
- To validate the forest's accuracy we use the original 1/3 partition that was not considered for training purposes, by calling the object's method **forestAccuracy**.
- To perform a query, **forestQuery**, we query an instance on every tree of the forest, and to determine its classification we consider the *majority vote*, that is, consider the classification given by the majority of the trees.

Important to mention is that we considered two approaches to select the *best attribute* to split data during the construction of a decision tree. The usual approach is to compute the information gain produced by splitting on all the attributes, and then picking the one that maximizes gain. The problem with this approach is that it might favor the phenomena of *overfitting*. A way to address this is to select any attribute randomly, and spare the computation of information gain. In Forest.py this can be easily adjusted by uncommenting line 594 and commenting line 593. The default setting is to use the best information gain approach. Now, to see if there was any substantial difference in our credit worthiness data set, we performed a couple of experiments to compare performance on both cases, next section shows this results.

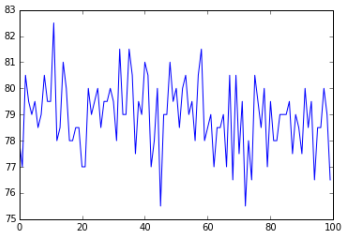
2 Experiments

In order to compare the impact of selecting an attribute to split data randomly or according to the highest information gain, to diminish the effect of overfitting (already addressed in our approach by using a forest instead of a single decision tree), we ran 100 iterations for each case. Also, we considered different number of forests: 5, 10 and 15 trees per forest, to see the behavior depending on the size of the forest. Following are the results with graphics of the accuracies observed while running the experiments.

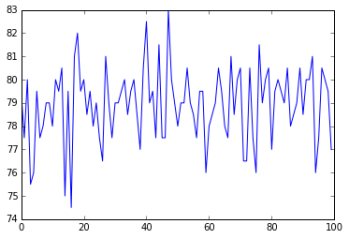
2.1 Highest gain attribute approach



Forest size : 15
Average accuracy: 79.1 %
Best accuracy: 83.5 %
Worst accuracy: 77.0 %

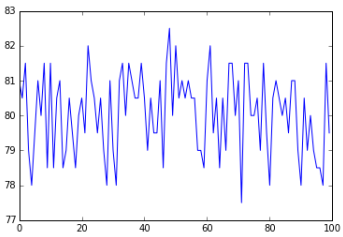


Forest size : 10
Average accuracy: 78.9 %
Best accuracy: 82.6 %
Worst accuracy: 75.4 %

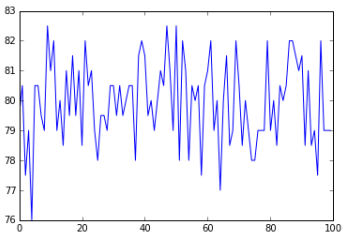


Forest size : 5
Average accuracy: 78.9 %
Best accuracy: 83 %
Worst accuracy: 74.4 %

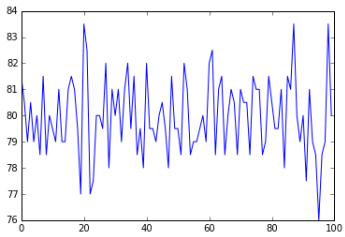
2.2 Random attribute selection approach



Forest size : 15
Average accuracy: 80.1 %
Best accuracy: 82.5 %
Worst accuracy: 77.6 %



Forest size : 10
Average accuracy: 80.0 %
Best accuracy: 82.5 %
Worst accuracy: 76.0 %



Forest size : 5
Average accuracy: 79.9 %
Best accuracy: 83.6%
Worst accuracy: 76.1%

Overall we can note that the more trees we use in our forest, the better the accuracy, in both the random attribute and highest gain attribute approaches, although the differences doesn't seem to be truly substantial. Also, we can note that the performance of the decision tree built using the random attribute approach is better than using the high gain approach, but again, the differences doesn't seem to be substantial, but the advantage here is that we avoid the additional computation of information gain for every attribute, which could be more running time consuming for larga data sets.

3 Conclusion

ID3 algorithm seems to be a very fast and concise way to classify data, based on previously seen data to train the decision tree. Query times are in the order of $\log(n)$, which is very fast, once the data structure has been built. The approach we followed to avoid overfitting seems to be working fine, obtaining on average 80% of accuracy. Overfitting was addressed in two fronts: by constructing a forest of decision trees instead of a single decision tree, and by considering the random attribute approach to split data during tree construction. It is not very clear from our observations, if it represents a substantial improvement to build larger and larger forests, it could be a waste of computing time, while the improvement could be little, if any. Also, although randomly selecting an attribute to split data yields slightly better accuracy, there is no strong suggestion that the improvements are substantial over selecting an attribute depending on the highest gain paradigm. Nonetheless, in terms of number of operations, the random attribute approach is less costly than the highest gain of information approach.