Decision Trees To Classify Mushroom Edibility

Manish Maithani, Kamalanathan Sogathur G

Abstract

Decision Trees have a wide scope of application in terms of classification problems and are one of the standard methods in classifying data. It is also the base for ensemble methods like Random Forests that are introduced to avoid the pitfalls of decision trees, i.e., prone to overfitting. We use this classic approach, and in doing so, build a tree from the very node based on the theory and working of decision trees. The implementation is a package on its own and is suitable to not just mushroom data, but any categorical data in general.

Introduction

In this project, we have attempted to solve the problem of mushroom poisoning by classifying the mushrooms as edible or poisonous based solely on the physical aspects and characteristics of the mushrooms as given in the dataset provided by *The Audubon Society Field Guide to North American Mushrooms (1981). G. H. Lincoff (Pres.), New York: Alfred A. Knopf.* The data was made available in the UCI Machine Learning Repository in April 1987. This popular dataset has been used for multiple classification training purposes, and in this project, we have attempted to build a decision tree from the mushroom data from the absolute scratch to get better insights into the working of the decision tree algorithm. A comparison between the accuracies of the results obtained from Naïve Bayes Method and Decision Tree Classification is made, to get a sense of the relative performance of both the classifiers for the Mushroom data.
The dataset has 8124 instances or rows and 23 different features for each data point, including the class label. There are no missing data in the dataset. The data is categorical, i.e., each cell in the matrix can take on a possible finite set of values. The distribution of the classes throughout the data is almost uniform – 51.8% of the instances are labeled *edible* and the remaining 48.2%, *poisonous*. Our aim is to determine these values for an unknown mushroom instance with the 22 physical features.

Background

Decision tree growing is done by creating a decision tree from a data set. Splits are selected, and class labels are assigned to leaves when no further splits are required or possible. The growing starts from a single root node, where a table that contains a training data set is used as input table. The table contains several columns that represent attributes and a single column that represents the class attribute. The expected output is a representation of the decision tree model that is based on the training data set. When you do a split, each of the created descendant nodes corresponds to the applicable subset of the training data set. Further splits of these nodes result in

new nodes that correspond to smaller subsets of data sets, and so on. Nodes that are not split further become leaves.
Decision tree growing is a repetition of the following operations:

Stopping criteria

This operation determines whether a split is done, or whether a node becomes a leaf because it is not split further. The decision is based on the subset of training data sets that correspond to the node.
No split is done if one of the conditions applies:

- All instances in the corresponding subset are of the same class
- The number of instances in the corresponding subset is less than a specified minimum
- The level of the current node is greater than a specified maximum, where the level of the root node is 1, and the level of its descendant nodes is 2, and so on
- The improvement of class impurity according to the best available split is less than a specified minimum

Class label assignment

The most frequent class in the corresponding subset is associated to each tree node. This operation is useful if the tree is subsequently pruned because subsequent pruning turns some nodes into leaves. It also makes the reading of the tree structure easier. The assignment of class labels is done for all nodes.

Split selection

This operation assigns minimum-impurity splits to nodes for which the stopping criteria were not satisfied. The set of candidate splits includes binary equality-based splits for all discrete attributes, and binary inequality-based splits for all continuous attributes. An impurity measure is applied to subsets corresponding to split outcomes to evaluate candidate splits.

Decision Tree Prediction:

Decision tree prediction uses a previously grown and possibly pruned decision tree to generate class predictions for a data set.
To propagate instances from the data set to the corresponding leaves, the splits from the tree nodes are applied. You can use decision tree prediction also to predict class probabilities that are based on the class distribution of training instances in leaves.

Implementation

To implement the Decision Tree algorithm from the scratch and to build a decision tree for a training dataset, the program was split into a bunch of functions. The required functions as in the program are:

`calEntropy(dataset)`

This function takes as its parameters a dataset, and calculates the total entropy in the dataset with respect to the class variables. In our case, the class labels are either edible or poisonous, so the calculation follows

$H = - p_{poisonous} \log_2(p_{poisonous}) - p_{edible}\log_2(p_{edible})$

Sanity checks were implemented when p is zero, but not considering those in the calculation of entropy.

`calcGain(col_name, dataset)`

Given a dataset and a column name or a particular variable in the dataset, the `calcGain()` function computes the information gain that we would get if we split the data on that column.

`Gain = Entropy(dataset) - ∑Entropy(subset of dataset containing only one variable in the column col_name)`

If there are $|C|$ categories in the *i*th column (let's call it col) of the dataset, for each $c \in C$, the data is subset as `dataset[, col == c]`. Let's call it $subset_c$. The entropy is calculated for this subset as `calEntropy(subset`$_c$`)` and the result is normalized by the probability of the column variable taking the value c, which is equal to $|c|/|C|$.

Hence, the net calculation is written out as

`calcGain(col_name, dataset) = calEntropy(dataset) -` $\sum_{c\in C} \frac{|c|}{|C|}$ `calEntropy(dataset[,col == c])`

`selectCategory(dataset)`

This function is defined to choose the variable in the dataset, splitting on which, we get the maximum information gain as calculated from `calcGain(col_name, dataset)` function and written the `col_name` for which it is maximum.

`selectCategory(dataset)` = $\text{argmax}_{col \in columns(dataset)}$ `calcGain(col, dataset)`

This ensures that the right category is chosen to split the tree on at any step, but this is also what makes decision trees prone to overfit.

`constructTree(parentNode, parent_cat_val, col_name, dataset)`

The `data.tree` package in R was used to physically construct the trees bottom up using the data that we obtained and arranged from the above calculations. The main page of the `data.tree` package website is cited in the references.

Once the above functions are properly defined in R, it is easy to start constructing the tree by recursively calculating the best column to choose and split on, subsetting the dataset to only have the relevant rows in the new dataset without the new column and recursively continuing this process. The chosen column is made to be the root node in each iteration, its children being the category variables in that column variable.

For each category variable, if the split is pure, it is made to be the leaf in the tree. Otherwise, it is now our new root node for further splitting. This process is continued until all the terminal nodes in the tree are leaf nodes.
The tree is constructed in such a way that each node stores the label of either the next variable name if it is column split, the name of the category in a variable in case of a dataset split after `selectCategory()` or the category name followed by a colon (':') and then 'y=' followed by the class variable that the example belongs to (this happens only in case of pure splits when the entropy is zero). The parser in the program parses through the stored text in each node and decides if the node is a leaf node or not, based on the presence or absence of certain text (like ':y=') to decide if a particular node is a leaf node or not. In case of a leaf node, the class variable is assigned as the text followed by ':y='. Otherwise, the recursion continues on variable names decided by the text stored in the node. This is evident by looking at the tree formed on each iteration.

`classname(query, dataset)`

This is the deciding function in the program that makes the prediction as to what the class variable of the query data must be, based on the tree that was constructed in the previous step. Traversing through the constructed tree, which is a bunch of text put together, requires a lot of parsing of unnecessary characters, and just the right name of the columns and classes. That is done in the function using string parsing and regex. The tree is traversed until a root node is reached. The tree is constructed in such a way that, when trained on the training data in the construction, the terminal leaf nodes are made to hold the class variable name. This helps in prediction as the prediction can be read off of the leaves that our query data ends up landing in.

`test(testdata, dt)`

The test function taken as input the test dataset for which we want to make the predictions, and the decision tree that was constructed in the training in the `constructTree()` method. The test data is made to fit in the tree and the predictions are stored in the output as a separate vector called answers. The values in the answers vector are tested with the true class labels of the test data to calculate the accuracy, which in our case is a perfect 100% when the training data is large enough.

Results

The following figures represent the constructed trees when 50%, 60%, 70%, 80%, 90% and 100% of the data is chosen for training respectively.

1) 50% of the data for training

```
                   levelName
1  odor
2   ¦--p:y=p
3   ¦--a:y=e
4   ¦--l:y=e
5   ¦--n:spore.print.color
```

```
6    ¦    ¦--n:y=e
7    ¦    ¦--w:habitat
8    ¦    ¦    ¦--g:y=e
9    ¦    ¦    ¦--w:y=e
10   ¦    ¦    ¦--p:y=e
11   ¦    ¦    ¦--d:gill.size
12   ¦    ¦    ¦    ¦--n:y=p
13   ¦    ¦    ¦    °--b:y=e
14   ¦    ¦    °--l:cap.color
15   ¦    ¦         ¦--c:y=e
16   ¦    ¦         ¦--n:y=e
17   ¦    ¦         ¦--y:y=p
18   ¦    ¦         °--w:y=p
19   ¦    ¦--k:y=e
20   ¦    ¦--r:y=p
21   ¦    ¦--h:y=e
22   ¦    ¦--b:y=e
23   ¦    ¦--y:y=e
24   ¦    °--o:y=e
25   ¦--f:y=p
26   ¦--c:y=p
27   ¦--y:y=p
28   ¦--s:y=p
29   °--m:y=p
```

2) 60% of the data for training

```
                          levelName
1  odor
2    ¦--a:y=e
3    ¦--p:y=p
4    ¦--f:y=p
5    ¦--s:y=p
6    ¦--c:y=p
7    ¦--y:y=p
8    ¦--n:spore.print.color
9    ¦    ¦--n:y=e
10   ¦    ¦--h:y=e
11   ¦    ¦--k:y=e
12   ¦    ¦--w:habitat
13   ¦    ¦    ¦--w:y=e
14   ¦    ¦    ¦--g:y=e
15   ¦    ¦    ¦--p:y=e
16   ¦    ¦    ¦--d:gill.size
17   ¦    ¦    ¦    ¦--b:y=e
```

```
18  ¦   ¦   ¦   °--n:y=p
19  ¦   ¦   °--l:cap.color
20  ¦   ¦       ¦--c:y=e
21  ¦   ¦       ¦--y:y=p
22  ¦   ¦       ¦--n:y=e
23  ¦   ¦       °--w:y=p
24  ¦   ¦--o:y=e
25  ¦   ¦--b:y=e
26  ¦   ¦--y:y=e
27  ¦   °--r:y=p
28  ¦--l:y=e
29  °--m:y=p
```

3) 70% of the data for training

```
                 levelName
1  odor
2   ¦--f:y=p
3   ¦--y:y=p
4   ¦--n:spore.print.color
5   ¦   ¦--n:y=e
6   ¦   ¦--k:y=e
7   ¦   ¦--w:habitat
8   ¦   ¦   ¦--p:y=e
9   ¦   ¦   ¦--g:y=e
10  ¦   ¦   ¦--w:y=e
11  ¦   ¦   ¦--l:cap.color
12  ¦   ¦   ¦   ¦--c:y=e
13  ¦   ¦   ¦   ¦--n:y=e
14  ¦   ¦   ¦   ¦--y:y=p
15  ¦   ¦   ¦   °--w:y=p
16  ¦   ¦   °--d:gill.size
17  ¦   ¦       ¦--n:y=p
18  ¦   ¦       °--b:y=e
19  ¦   ¦--y:y=e
20  ¦   ¦--b:y=e
21  ¦   ¦--h:y=e
22  ¦   ¦--r:y=p
23  ¦   °--o:y=e
24  ¦--s:y=p
25  ¦--a:y=e
26  ¦--l:y=e
27  ¦--m:y=p
28  ¦--p:y=p
29  °--c:y=p
```

4) 80% of the data for training

```
                      levelName
1  odor
2   ¦--y:y=p
3   ¦--s:y=p
4   ¦--n:spore.print.color
5   ¦   ¦--w:habitat
6   ¦   ¦   ¦--g:y=e
7   ¦   ¦   ¦--w:y=e
8   ¦   ¦   ¦--l:cap.color
9   ¦   ¦   ¦   ¦--y:y=p
10  ¦   ¦   ¦   ¦--n:y=e
11  ¦   ¦   ¦   ¦--w:y=p
12  ¦   ¦   ¦   °--c:y=e
13  ¦   ¦   ¦--d:gill.size
14  ¦   ¦   ¦   ¦--b:y=e
15  ¦   ¦   ¦   °--n:y=p
16  ¦   ¦   °--p:y=e
17  ¦   ¦--k:y=e
18  ¦   ¦--o:y=e
19  ¦   ¦--n:y=e
20  ¦   ¦--r:y=p
21  ¦   ¦--y:y=e
22  ¦   ¦--b:y=e
23  ¦   °--h:y=e
24  ¦--f:y=p
25  ¦--c:y=p
26  ¦--l:y=e
27  ¦--a:y=e
28  ¦--p:y=p
29  °--m:y=p
```

5) 90% of the data for training

```
                      levelName
1  odor
2   ¦--l:y=e
3   ¦--f:y=p
4   ¦--a:y=e
5   ¦--n:spore.print.color
6   ¦   ¦--k:y=e
7   ¦   ¦--n:y=e
8   ¦   ¦--w:habitat
9   ¦   ¦   ¦--g:y=e
10  ¦   ¦   ¦--l:cap.color
```

```
11  ¦   ¦   ¦   ¦--c:y=e
12  ¦   ¦   ¦   ¦--w:y=p
13  ¦   ¦   ¦   ¦--n:y=e
14  ¦   ¦   ¦   °--y:y=p
15  ¦   ¦   ¦--w:y=e
16  ¦   ¦   ¦--d:gill.size
17  ¦   ¦   ¦   ¦--n:y=p
18  ¦   ¦   ¦   °--b:y=e
19  ¦   ¦   °--p:y=e
20  ¦   ¦--r:y=p
21  ¦   ¦--b:y=e
22  ¦   ¦--y:y=e
23  ¦   ¦--o:y=e
24  ¦   °--h:y=e
25  ¦--s:y=p
26  ¦--y:y=p
27  ¦--p:y=p
28  ¦--m:y=p
29  °--c:y=p
```

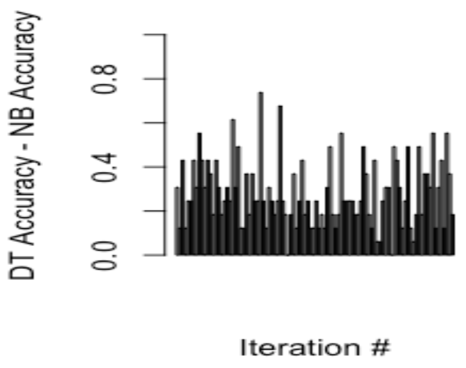
6) 100% of the data for training

```
                 levelName
1  odor
2   ¦--s:y=p
3   ¦--n:spore.print.color
4   ¦   ¦--n:y=e
5   ¦   ¦--k:y=e
6   ¦   ¦--w:habitat
7   ¦   ¦   ¦--w:y=e
8   ¦   ¦   ¦--d:gill.size
9   ¦   ¦   ¦   ¦--n:y=p
10  ¦   ¦   ¦   °--b:y=e
11  ¦   ¦   ¦--l:cap.color
12  ¦   ¦   ¦   ¦--w:y=p
13  ¦   ¦   ¦   ¦--c:y=e
14  ¦   ¦   ¦   ¦--y:y=p
15  ¦   ¦   ¦   °--n:y=e
16  ¦   ¦   ¦--p:y=e
17  ¦   ¦   °--g:y=e
18  ¦   ¦--b:y=e
19  ¦   ¦--y:y=e
20  ¦   ¦--r:y=p
21  ¦   ¦--o:y=e
22  ¦   °--h:y=e
23  ¦--a:y=e
```

```
24  ¦--f:y=p
25  ¦--l:y=e
26  ¦--c:y=p
27  ¦--p:y=p
28  ¦--m:y=p
29  °--y:y=p
```

It is evident that the results are consistent throughout and the accuracies recorded are 100% for each of the cases.

By running the Naïve Bayes classifier on each testing and training data subset out of the main data as for the decision trees, we can see that the Naïve Bayes classifier is also a good fit for this case. The accuracy is not perfect every time but it's very close. This implies that the assumption of independence between variables is a reasonable one to make. The difference in accuracies between the two classifiers never goes more than 1 and the Decision Tree works generally better than the Naïve Bayes in this case.



The graph shows the bar plots drawn for the differences in accuracies between the two classifiers and we can see that the difference ever barely goes beyond 1. All the plots are positive, meaning the Decision Tree almost always works better than Naïve Bayes, even when the independence condition is in place. The mean difference between both the classifiers for 100 runs is found to be 0.2806.

Conclusion

From the results obtained, we can conclude that Decision Trees are a great way of classifying data, but one possible shortcoming is overfitting. There are techniques that are good work arounds for the problem of overfitting, such as pruning the trees after a certain number of branchings. However, in our case, the maximum number of branching was 5, which isn't a bad tree for a dataset with over 8000 rows. The data.tree package was efficiently incorporated to build decision trees with perfect results. As the decision tree construction is generic in nature in the method of build, it can be extended to any dataset with categorical variables and not just mushroom data. The problem of overfitting, however was not looked into in this problem as the number of branchings was optimal for the mushroom dataset.