Decision Tree Lab

Seong-Eun Cho

October 29, 2019

Introduction

Decision Tree is a tree-based approach of learning. At each node of the tree, the model makes a decision based on the input features to ultimately make a prediction at the leaf nodes. The way Decision Tree learns from data is very intuitive—it looks at one feature at a time and splits based on how well the feature separates different output classes. As implied in its name, it works in a similar way as how we would make a decision in classifying things in the world. For example, suppose we want to classify how good a restaurant's food might be based on the reviews, pricing, and menu options. We might first believe that other people's review reflects most about the restaurant. However, because of inflated reviews, it might be common for an average place to get a rating between 3.5-4.5, so then we might want to look at the pricing of the restaurant. Depending on the pricing, we might then want to look at the number of options there are in the menu since too many choices can lead to low quality food. We are used to following this type of process when making daily decisions.

One of the main advantages of Decision Trees is precisely the fact that it is intuitive. We can look at the resulting tree and quite easily understand how the model is learning and what features is most important in the learning task. Another advantage is that with good implementation and regularization, Decision Trees can learn large amounts of data extremely quickly. Although a single tree might not do as well as other learning methods (fast learning generally doesn't allow for good accuracy), many random trees can also easily be used for *ensemble learning*, and this method of learning is called *Random Forest*. Although Random Forest is made up of simple individual trees, it is one of the most widely used machine learning algorithms today.

1 Code Implementation

In this lab, the Decision Tree is implemented using the ID3 approach. ID3 uses *entropy* values in order to choose a feature to split on. The main idea of this approach is to pick a feature that would give us the most information about the output class. A feature that is deterministic with the output gives us the most information while a feature that is irrelevant to the output gives us little information. Entropy gives us a way we can calculate this information. The formula for Entropy is given by:

$$\text{Entropy}(p) = -p\log_2(p)$$

The basic intuition of entropy is that given a probability value p, $\text{Entropy}(p)$ gives a qualitative value for how random a draw from p is. For example, a Bernoulli with probability of 0.5 has the most entropy since the draws are the most random, and probability 0 gives 0 entropy since the draws are deterministic.

The ID3 algorithm calculates the information gain at each stage by:

$$\text{Gain}(S, F) = \text{Entropy}(S) - \sum_{f \in values(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f)$$

where S is the current set of data, F is specific feature from the set of input features, $f \in values(F)$ represents a value F can take, and S_f is S with f as the F feature. We calculate the Gain of each feature and choose the one with the highest Gain to split the tree on. Observe that in this equation, $\text{Entropy}(S)$ is not reliant on F, so we can actually just find $\sum_{f \in values(F)} \frac{|S_f|}{|S|} \text{Entropy}(S_f)$ then split on the feature with the lowest of this value.

When training, we first use the full training data. Then, we compute the Gain of each feature to find the best feature to split on. We then recursively pass in a partitioned dataset that is split on each value of the feature until we reach a split that has no ambiguity with the output label, at which point it is considered a leaf node. In my implementation, the resulting tree is constructed using Python dictionary.

The keys of the dictionary are n-tuples, where the first n-1 elements are tuples representing the path of the tree with the first entry represents the feature split on and the second entry is what value that feature took in this specific path. The n^{th} element is a scalar representing the current feature we are splitting on. For example, if the key is ((3, 1), (2, 0), 1), we first split on the 3^{th} (counting from 0) feature which took the value of 1, then split the partitioned data at the 2^{th} feature which took the value of 0, and now we are further splitting on the 1^{th} feature.

The value for each key is a list, where the index represents the value of the current feature and the element at that index is the next feature to split on. If the element is also wrapped as a list, it is a leaf node and the value is the output label of the leaf node. After training,the tree can then simply be traversed down at test time until a leaf node is reached which is the prediction.

1.1 Debug and Evaluation

The debug dataset trained on lenses.arff was tested on the all_lenses.arff and scored 0.33 as expected. The predictions that was exported as a .csv file matched with the .csv file provided by the TAs. When using just the lenses.arff and doing a 70-30 split for train and test data, the model performed between 0.6-0.875 on the test set. For the evaluation data, the model was trained on zoo.arff and tested on all_zoo.arff. The score was 0.15 and the resulting predictions were saved in a .csv file that is included in the folder with code.

2 Cars and Voting Dataset

The Cars dataset is a small dataset that has 6 input features which represent the pricing and comfort of a car. The labels of each is the acceptability of the car. The Decision Tree model was trained on the Cars dataset using a 10-fold cross validation. The result of this experiment is summarized in a table below.

Cars dataset 10-fold cross validation result

	1	2	3	4	5	6	7	8	9	10	Average
Train accuracy	100	100	100	100	100	100	100	100	100	100	100
Test accuracy	89.66	91.95	91.95	94.83	95.38	93.02	90.70	94.77	94.19	91.23	92.77

The Voting dataset is a data that shows people's voting pattern and the political party they are associated with. The input feature consists of people's opinion about certain policies and the label is which party they belong to. The model was trained and tested in the same way as above using 10-fold cross validation. The results are summarized in a table below.

Voting dataset 10-fold cross validation result

	1	2	3	4	5	6	7	8	9	10	Average
Train accuracy	100	100	100	100	100	100	100	100	100	100	100
Test accuracy	88.64	86.36	100	95.45	100	93.18	93.18	93.02	92.86	97.62	94.03

3 Summary of Learning for Cars and Voting

In this section, I will describe how the resulting tree shows what the model has learned for both the Cars and Voting data. Because these trees are very deep, I will only summarize the most important features the tree has used and the places where the decision could be made without going too deep.

For Cars, the most important feature about the acceptability of the car is the safety. When the car is considered unsafe (low on the safety feature), it is automatically deemed unacceptable no matther what the other features are. When the safety is medium or high, the next most important feature was the capacity of the car. A 2 person car is unacceptable no matter the other features as well. It seems that overall, cars with higher capacity is more acceptable. The next most important feature is the buying price of the car. In general, the lower the buying price the more acceptable the car is.

For Voting, The most informative feature is the physician-fee-freeze attribute. If the voter responded 'n' for this feature, adoption-of-the-budget-resolution feature is the next most informative feature. All those that responded 'y' or didn't respond were democrats. Otherwise, almost all of them were republicans. If the voter responded 'y' for physician-fee-freeze, synfuels-corporation-cutback is the next most informative feature.

The interesting thing about the Voting dataset is that the most important features Decision Tree used to split on was also the most important features Perceptron classifier put its weights on. This shows that no matter which machine learning algorithm we use, the features will be used in similar ways.

4 Handling Missing Values for Voting

The Voting data was presented with many missing values in the input. There are many ways to approach this problem. For this dataset, I thought that the most adequate approach would be to simply use the missing value as a feature of its own. This is because the missing values don't represent an error in recording, but a conscious decision of the voter to not respond. As stated in the .arff file, it reads "It is important to recognize that '?' in this database does not mean that the value of the attribute is unknown. It means simply, that the value is not 'yea' or 'nay'". This statement is pretty indicative of the fact that it wouldn't be a good idea to use 'y' or 'n' to replace the missing values and supports the choice of using it as a separate feature.

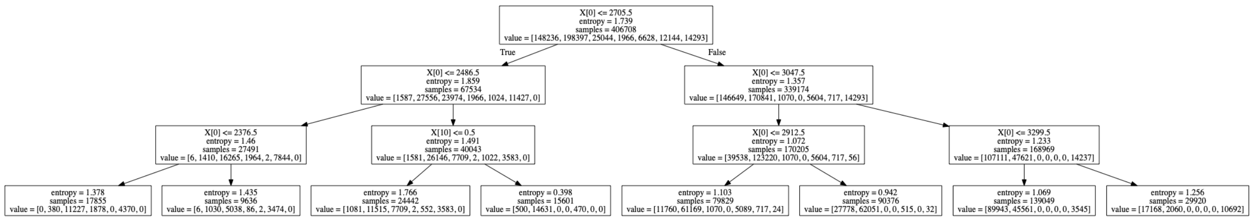
5 Scikit-learn's Implementation

The most noticeable difference when using the Scikit-learn's implementation is speed of training. 10-fold cross validation on the Cars and Voting data took a good 5 seconds using my implementation but was instantaneous with Scikit-learn's. I tried using different hyperparameters like the max_depth, max_leaf_nodes, min_samples_leaf, min_samples_split, and min_impurity_decrease but non of those parameters improved the cross validation accuracy. However, using the Gini criterion seemed to improve the accuracy compared to the Entropy criterion. The average accuracies it got for the two datasets were 98.09% and 98.55% for Cars and Voting data respectively.

The dataset I chose for the final experiment is the UCI Covertype dataset, which uses different features such as elevation, slope, soil type, etc. of a forest to predict the cover type of the forest which is an integer between 1 and 7. The input is composed of 54 features with a mixture of continuous and nominal features. This is a large dataset with 581012 points. I found that even with such large data with many features, not many of the hyperparameters were helpful in improving its generalization. It almost seemed like the Scikit-learn had a built in method to help generalize at test-time (although I'm not sure of this). The max_depth parameter was set to 50 to help it train within a reasonable amount of time. For this data, entropy was a better criterion than Gini. The average test set accuracy I got was 93.94%.

6 Graphviz

The resulting tree was saved using Scikit-learn's export_graphviz function. Because the actual tree is very dense and large, I only used the top 3 levels for visualization.


X[0] <= 2705.5
entropy = 1.739
samples = 406708
value = [148236, 198397, 25044, 1966, 6628, 12144, 14293]
True
False
X[0] <= 2486.5
entropy = 1.859
samples = 67534
value = [1587, 27556, 23974, 1966, 1024, 11427, 0]
X[0] <= 3047.5
entropy = 1.357
samples = 339174
value = [146649, 170841, 1070, 0, 5604, 717, 14293]
X[0] <= 2376.5
entropy = 1.46
samples = 27491
value = [6, 1410, 16265, 1964, 2, 7844, 0]
X[10] <= 0.5
entropy = 1.491
samples = 40043
value = [1581, 26146, 7709, 2, 1022, 3583, 0]
X[0] <= 2912.5
entropy = 1.072
samples = 170205
value = [39538, 123220, 1070, 0, 5604, 717, 56]
X[0] <= 3299.5
entropy = 1.233
samples = 168969
value = [107111, 47621, 0, 0, 0, 0, 14237]
entropy = 1.378
samples = 17855
value = [0, 380, 11227, 1878, 0, 4370, 0]
entropy = 1.435
samples = 9636
value = [6, 1030, 5038, 86, 2, 3474, 0]
entropy = 1.766
samples = 24442
value = [1081, 11515, 7709, 2, 552, 3583, 0]
entropy = 0.398
samples = 15601
value = [500, 14631, 0, 0, 470, 0, 0]
entropy = 1.103
samples = 79829
value = [11760, 61169, 1070, 0, 5089, 717, 24]
entropy = 0.942
samples = 90376
value = [27778, 62051, 0, 0, 515, 0, 32]
entropy = 1.069
samples = 139049
value = [89943, 45561, 0, 0, 0, 0, 3545]
entropy = 1.256
samples = 29920
value = [17168, 2060, 0, 0, 0, 0, 10692]