A look into the Random Forest algorithm

Guillermo Serrahima

April, 2019

1 Introduction

This document will give an overview of the Random Forest algorithm, which is a bagging technique applied to the Decision Tree classification algorithm, and go through an implementation of the algorithm as an example of the theory explained.

2 Available data and target value

As a classification algorithm, it makes sense to work on a tangible set of data that we want to classify, both to support the theory with clear examples and for the following implementation.

In this case, we're going to work with the HYG Database (version 3), a list of over 100 thousand nearby stars, with different physical data (like magnitude, temperature, luminosity, ...), and their spectral classification (O, B, A, F, G, K, or M, from biggest and hottest to smallest and coolest). Specifically, the spectral classification is the target value that we are aiming to predict.

3 Decision Trees

First, a reminder of what a Decision Tree consists on. It's a predictive model that goes from observations about an item (in the form of evaluation of its parameters to decide to which subset of the data belongs) in order to reach a conclusion (a prediction) about a target value of the item. There are 2 types, depending on the type of outcome we expect:

- **Classification tree:** when the outcome is a discrete type of value.
- **Regression tree:** when the outcome is a continuous type of value.

In our case, the tree we will be using will be a classification tree (our target value is one among only 7 possible values); however, an umbrella term for both of these (since they share similarities) is CART (*Classification and Regression Tree*), so we may refer to trees in this document with that name as well.

4 Decision Tree Learning

For the CART evaluation of a data item, at each node the item is compared against a condition, and moved to a child node or another depending on if they meet a specific condition or not; eventually, the item should go through a series of evaluations reaching a leaf where there is only one possible target value for it, or where the decision tree cannot further tell apart from the possible target values, which usually a very small subset of the target value's dominion.

In our case, since star data is imprecise, some stars could belong to more than one class, and this is reflected in the dataset. In our implementation, a star is considered to belong to each of those possible classes, and thus, newly evaluated stars with similar characteristics could belong to either.

Thus, for the training of a Decision Tree, at each node we aim to evaluate the dataset into 2 subsets, depending on if each entry meets a condition or not, trying to "get the best split", so as to be able to reach subsets with the least unique target values in the minimum amount of splits.

There are different ways to define what is the best split. Two popular methods are: the Gini impurity, which tells how mixed or unmixed a set is (the higher the impurity, the more variance among values of a set exists) - and thus the information gain is defined as the reduction of the impurity between a node and its children; and information entropy (where higher entropy means higher variance in a set), and information gain is defined as the reduction in the entropy score between a node and its children.

The formulas for each are the following:

$$Gini = 1 - \sum_{i=1}^{n} p^2(c_i)$$

$$Entropy = \sum_{i=1}^{n} -p(c_i)log_2(p(c_i))$$

where $p(c_i)$ is the probability/percentage of class c_i in a node.

Figure 1: Gini impurity and entropy score formulas

We will be calculating the Gini impurity, but either work equally fine.

During the training, for each node we will give the belonging subset of the dataset (the root will receive the full dataset, the left branch will always be the branch with the subset of entries that meet the condition, and the right branch will be the subset of entries that do not). For the received set, a set of unique values for each of the possible columns is formed, and for each possible unique value a Question (an object testing the condition) is made. For a column of a numeric value, the Question evaluates if a given entry is strictly bigger than the Question's value. For a non-numeric value, the test is simply if the entry has the same value as the one in the Question.

Then, across all possible Questions, the information gain from the split is calculated, and the Question which gives the biggest information gain is chosen, the dataset split, and each of the 2 children receive the subset that either meets or doesn't meet the condition.

Since the search is extensive, for sets bigger than 1000 entries, the Question and split evaluation is parallelized (since the information gain of one Question is independent from the rest). For the linear evaluations, though, only the current best question and split is kept, and they're only updated when a better one is found; there's no need to keep track of all possible splits at the same time.

5 Decision Tree Testing

To train and test a Decision Tree, the following steps are done:

1. We parse the whole HYG Database, creating a list of entries only with the values that we will be using: star's radial velocity, absolute magnitude, color index, and luminosity; and, obviously, the spectral class (which is kept aside, as the star's "label", since it's the target value). Those with null spectral class, or (as stated in the database description) have a dubious distance value (which could mean all of that star's data is erroneous) are discarded.

2. We shuffle the resulting dataset, and then split on a cutoff value to indicate how much will be used in training, given as a value between 0 (no entries) and 1 (use the whole database).

3. The training subset is given to the Tree when creating an instance of the class, which in turn passes it onto it's root Node (and this one, recursively, to its children).

4. The testing subset is then iterated, and for each entry value, we ask the tree to evaluate it; we then measure how many of these evaluations were correct to calculate the accuracy.

Some notes:

- Among the star's data, only the magnitude, color and luminosity are directly dependent of the stellar class (and vice versa); the radial velocity is experimentally given as a "distraction value"; a good tree should discern properly the class despite this.

- The shuffling was added later, so the first tests always split the database in the order given.

- For stars that have more than one label (could belong to more than one class), each one is counted separately when counting the unique values in the Gini impurity computing. Obviously, these can't be split with Questions, so some leafs have possible outcomes and $1/n$ prediction confidence for each.

- For leaves with more than one possible outcome, a "success" is calculated as the average of successes across hypothetical random choosing - in other words, it's counted as a partial success of value n/m, where n is the amount of occurrences of the correct class in the leaf's possible predictions, and m is the total of possible prediction values.

6 Decision Tree: Results and Problems

Across different tests with different cutoffs (from cutoffs of 2% to 90%), the accuracy is always around 66%. This highlights the biggest issue of classification trees: they're not that good.

It's well known that CARTs are subpar predictive models; small changes in the dataset can signify huge changes in the tree; work well for sets of data that fit very neatly in clearly differentiated niches but have a hard time dealing with fuzzy borders; are prone to overfitting; ... They're only well known because they're easy to conceptualize and easy to use. Which brings us to the main subject: Random Forests.

7 Random Forests

The Random Forest algorithm is an attempt at boosting the Decision Tree algorithm. Specifically, the technique used is called "bagging", which consists of creating new datasets from the original one by random selection with replacement. In other words, a Random Forest will have n trees, each trained with a different dataset of the same size of the original one, built by randomly selecting entries, allowing duplicates.

The prediction, then, follows a "voting" system: a new entry to be tested is evaluated by each tree, and the prediction value that is obtained the most times is the final predicted value. The idea behind this is that each individual tree, if correct, will all agree on the proper class; but if incorrect, each tree may fail at different decision branches, giving different incorrect values, and thus thinning out the relevancy of errors. The difference in datasets is to allow the creation of similar but different trees, which may give more importance to certain values (the repeated entries) than others.

In the implementation, to avoid keeping track of multiple dataset lists, which would be a huge hit to the memory space, a new version of the Tree class is defined, that receives the original dataset and then a list of indices, which correspond to the bootstrapped values to be used in training. The number of Trees to be used in a Forest is also passed as an argument to the Forest.

This allows for a new metric as well: the Out-of-Bag error. For each bootstrapped dataset, the amount of expected unique entries is the following, for n samples across a dataset of x entries:

$$x \cdot (1 - (1 - \frac{1}{x})^n) \tag{1}$$

This tends to $1 - \frac{1}{e}$, or approximately 0.63, or 2/3 of the original dataset. The remaining 1/3 is said to be Out-of-Bag (specifically for the tree that doesn't see it in training). So, for each entry that is Out-of-Bag on any tree of the Forest, is evaluated **only** by those trees that didn't see it in testing, using the vote of the majority of those trees, and from the results, the percentage of error is called the Out-of-Bag error.

8 Random Forest Training and Testing

The training is the same as for an individual Decision Tree, but repeated for each bootstrapped version of the dataset.

For the sake of completing the tests in reasonable time, small cutoff values have been used (which, in a dataset as big as the HYG Data, means that a "small" value of 20% is already over 26 thousand stars, so it's still a reasonably sized training set). However, for tests of 20% to 40% of the dataset, already an increase in accuracy is shown from around 66% to 75%; even for Forests as small as 10 or even 4 trees; and despite the OOB error observed for such small forests being as high as 40%.

9 Future Testing and Further Work

The code is available online to be downloaded and used on this repository, to check that the values given in this document are in line with the code execution itself. The "*_tester.py" files are the ones to be executed ("forest_tester" already includes a single tree test though, for comparison, so "tree_tester" is unneeded).

Modifications can be made to the data used, as well as the configuration. The absolute magnitude of a star is calculated from the apparent magnitude and the distance from the Earth; both these values can also be used in conjunction to the ones mentioned above for finer grained testing. likewise, the temperature and the color index are directly dependent; one could be used instead of another, or together. And there are a plethora of irrelevant data that can be thrown in to test the sturdiness of the Forest. Doing tests on Forests with bigger cutoff values should be interesting as well.

In addition, these can be adapted to any other dataset easily, as long as the parser gives a list of strings that are the names of the value fields, and a list of objects that have both a "label" attribute and a "data" attribute, storing a list of labels (target values) and the list of values that correspond to each field in the fields list, respectively.

10 Conclusion

I was aware of the subpar effectiveness of the decision tree as a predictive model, but being able to test it out with a practical example has been incredibly useful to gauge it; and likewise, the Random Forest algorithm is an incredibly effective correction for the simplicity that the concept has.

Aside from that, doing both algorithm implementations along the research has also been incredibly helpful to understand the different metrics used (Gini impurity, Out-of-Bag error, ...) with such clear examples. As a last thought, this has been a satisfying project.