C4.5 Trees & AdaBoost

Brennan Kuo
Thomas Redding

May 31, 2017

Introduction

The primary goal of this project was to understand how and why C4.5 Trees and AdaBoost work. To facilitate this, we considered a dataset of 300 people's tea preferences [3] [5], and tried to predict whether each person prefered Black, Earl Grey, or Green tea based on their other variables in the dataset. This goal makes sense, because both C4.5 Trees and AdaBoost are used in (supervised) classification.

The "input" dimensions were mostly binary and included variables like

- Do they drink tea with breakfast?
- Do they drink tea at work?
- Do they drink tea with sugar?
- etc.

To these ends our paper has three parts. First, we will discuss C4.5 trees, an algorithm used to make decision trees. Next, we will turn towards AdaBoost and why it "is considered to be one of the best out-of-the-box classifiers today" [8]. Finally, we will apply these tools to the aforementioned tea dataset.

C4.5 Algorithm

The basic premise of the C4.5 algorithm is that we we want to build a tree where at each node, we split the data among the category that most accurately partitions the data. Specifically, at the root level, we find the category that best partitions the complete dataset. We then partition the dataset according to the category that does it best, and then on these two smaller datasets, find the category that best partitions these, and repeat. This keeps going, either until we reach our desired depth, or our depth reaches the number of categories that we have to split on [4].

The most important and challenging part of the C4.5 algorithm is figuring out which category is the best for splitting the data at any given node. This is done by calculating the information gain for every category, and then choosing the category that has the max information gain as our category to split on.

In order to calculate the information gain, however, we need to first be able to calculate the entropy for both the undivided dataset and the dataset as it would be if we were to use the given category to split the data. Entropy can be thought of as the complement of the likelihood that we will be able to guess where something belongs given a piece of data and prior knowledge of other data. If we can't figure out where a piece of data is supposed to be assigned, then it is going to have a very high entropy. Therefore, at every stage, our goal is to minimize the entropy from what it was in the previous level of the tree.

The equation for calculating entropy at each level before the data is split is:

$$-\sum_{i=1}^{n} P(x_i) \cdot \log(P(x_i))$$

To calculate the entropy after the data split for each category, we calculate the entropies of each of the new nodes that we potentially create, and then weight them by the proportion of the data that each new node takes. Therefore, we get the following:

$$\sum_{j=1}^{m} prop(j) \cdot -\sum_{i=1}^{n} P(x_i) \cdot \log(P(x_i))$$

We will then take the first result and subtract it from the second to get the information gain, which is demonstrated in the following equation:

$$-\sum_{i=1}^{n} P(x_i) \cdot \log(P(x_i)) - \sum_{j=1}^{m} prop(j) \cdot -\sum_{i=1}^{n} P(x_i) \cdot \log(P(x_i))$$

The reason we use this method of calculating entropy and minimizing it (which is also the maximization of the information gain) is because by decreasing entropy, we are increasing our likelihood of being able to classify a new piece of data. This method allows us to create trees that classify the most data properly, which is the point of the C4.5 algorithm.

Finally, we added three more features:

1. We stopped recursively extending the tree when a depth limit was reached, because AdaBoost requires *weak* learners.
2. We stopped recursively extending the tree when a certain accuracy threshold was obtained.
3. We pruned branches after the fact if the p-value of the split actually improving accuracy isn't met [2].

Of particular note in our implementation: We have actually implemented the C3 algorithm instead of the C4.5 algorithm, because in terms of our dataset, they are identical. The C4.5 algorithm has the added benefits of being able to classify gradients and non-numerical data, which we did not have in our dataset. Therefore, we did not implement the necessary checks and conversions for our algorithm to work on this data.

AdaBoost

AdaBoost is based on two main ideas [6]:

1. Experts whose mistakes are relatively independent can become more accurate by voting - a truism sometimes called "wisdom of the crowds".
2. Experts who focus on correctly classifying the things other experts wrongly classified are relatively independent of those other experts.

In AdaBoost, these experts are *weak learners*. Weak learners are models constructed by some other machine learning algorithm that achieve better than random classification accuracy.

Although, originally conceived to perform binary classification, AdaBoost has been recently adapted for application to multi-class problems [9]:

Let n denote the number of data points, x_i denote the i^{th} point, y_i denote its class, and K denote the number of classes.

1. Initialize a weight for each data point: $w_i = 1$
2. Repeat Until Convergence:
    a. Create a weak learner C_m. Let $C_m(x_i)$ denote its

```
        classification.
    b. Compute the error, err, by adding the weight of all
       misclassified points. Then divide by ∑_{i=1}^{n} w_i.
    c. Let α_m = ln((1−err)/err) + ln(K − 1)
    d. Multiply each weight by e^{α_m} if the point is
       misclassified.
```

Then, to classify a point, we do

```
Let x be the point to classify.

float[] votes = new float[K]
for learner in weak_learners:
    votes[learner.classify(x)] += α_m

The class with the most votes is what we output.
```

This raises an important question: why does this work?

To be more precise, it's unclear

1. Why we compute α_m the way we do.
2. Why we reweigh points by multiplying by e^{α_m}.

These questions are answered in great detail by other authors [8] [9], and we provide our own explanation based of these in the appendix. However, the basic explanation for binary AdaBoost is that we can think of it as greedily decreasing an error function.

In English, it's current net-vote for each point can be represented by a number (the weighted sum of the weak learner's votes). If that number is positive, it intends to classify the data point as +1, and if it's negative it will guess -1.

Ideally, we'd like AdaBoost to assign large, positive numbers to all +1 points and large, negative numbers to all -1 points. This means we don't have a target weighted sum - the bigger the better (in the right direction). On the other hand, we should clearly care more about moving a +1 point's sum from -0.3 to +0.3 than from moving a similar point's sum from 99.7 to 100.3, because the former actually changes the classification.

Moreover, because we don't know what future weak learners we will be adding, we should be somewhat cautious, which justifies also want an error function that emphasizes moving from +0.7 to 1.3 over moving from 99.7 to 100.3.

Given these criteria, a natural choice of error function is exponential error. That is, if v_i represents the current voting-sum for point x_i, then the error should be e^{-v_i} if the point is a +1 point and e^{v_i} if the error is a -1 point.

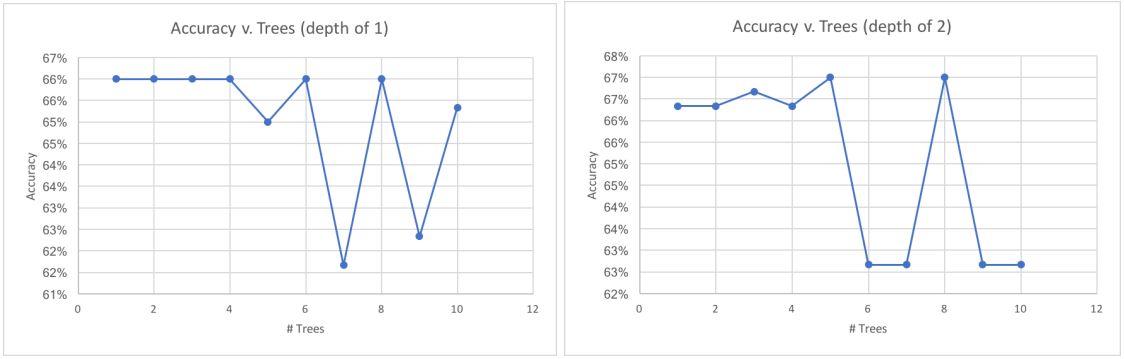
Some reflection by the reader should demonstrate that this meets the criteria we laid out above.

http://www.cs.man.ac.uk/~stapenr5/boosting.pdf

Statistical Results

It turns out that predicting someone's tea preferences is hard! We see essentially no gain above baseline, which was always guessing Earl Grey, which 64.3% of the people prefer. This is unfortunate, because it means that our analysis of parameter tweaking is fairly limited. However, here is the overview.

First, we set the depth of the trees to 1 without any stopping for sufficient accuracy or pruning. We can graph the error against the number of trees:



[note, we choose to not start the y-axis at zero, because that strictly reduces the information conveyed by the chart]

As you can see, we see absolutely no gain after the first tree, and even the first tree see's only a 1.7% improvement - an increase of 5 correct classifications. Given that we used 34 variables, this is almost certainly just noise. There could, however, be interaction terms, so we also tried increasing the depth to 2. This time, we saw a 2.33% gain, which, given yet the more degrees of freedom, is probably not significant either.

A natural question is to ask what tweaking the other parameters did. The answer is more-or-less, nothing in terms of accuracy. They just make the tree's smaller (either during construction or pruning). In short, if we had more time, it'd probably be a good idea to try a different dataset.

Appendix

This Appendix may prove useful to understanding the theoretical justifications of AdaBoost more rigorously. However, this less useful for understanding (1) how to implement AdaBoost and (2) why AdaBoost works on an intuitive level. However, we found the writing of this section enlightening, so we decided to keep it rather than send it into the void of deleted bytes.

Recall our questions:

3. Why we compute α_m the way we do.
4. Why we reweigh points by multiplying by e^{α_m}.

To answer the questions questions, we're going to simplify our review by considering analysis of the binary classification case [8]. However, the algorithm and analysis are very similar in the multi-class case and are extended by others [9].

Because we are considering the binary case, y_i becomes a binary variable (-1 or +1), and we make the slight tweak that each weak learner, C_j, outputs +1 or -1, depending on

whether it votes for 1 or 0, respectively. The only other tweak to our algorithm is that we also multiply weights by $e^{-\alpha_m}$ if we classify it correctly.

Imagine we have computed the first $m-1$ classifiers and are trying to determine what α_m should be for our new classifier, C_m. Recall that the first $m-1$ classifiers work by computing the sum

$$v_i = \alpha_1 C_1(x_i) + \cdots + \alpha_{m-1} C_{m-1}(x_i)$$

Unlike in the multi-class algorithm we implemented, in binary classification each classifier, C_i, makes a binary classification represented by +1 or -1.

After each iteration, we multiplying each point by e^{α_m} if it is misclassified and $e^{-\alpha_m}$ otherwise. This implies that $w_i = e^{v_i y_i}$, where v_i is the sum of all the α's whose classifiers classified x_i incorrectly. This, tells us that $v_i \geq 0$ iff the first $m-1$ classifiers are misclassifying the point in aggregate.

With just one more bit of information, we can answer both our questions. That bit of information stems from our choice of error function. In AdaBoost, the error function is $e^{v_i y_i}$.

The choice of this function is somewhat arbitrary, but, due to the increasing derivative, it more strongly encourages the algorithm to focus on correcting -1 points with positive v_i and +1 points with negative v_i. Moreover, the more "off" these points are, the greater their derivative, and therefore the more we push AdaBoost to care about them.

So, now, we can prove the formula for α_m by trying to minimize our error.

$$E = \sum_{i=1}^{n} e^{y_i(v_i + \alpha_m C_m(x_i))}$$

$$E = \sum_{i=1}^{n} e^{y_i v_i} e^{y_i \alpha_m C_m(x_i)}$$

Note, by our observation before, though, that $w_i = e^{v_i}$. The reason we chose this method for reweighing [question 2] data points is because of this next bit:

$$E = \sum_{i=1}^{n} w_i e^{y_i \alpha_m C_m(x_i)}$$

Now, we can find the optimal α:

$$0 = \frac{dE}{d\alpha_m} = \sum_{i=1}^{n} y_i C_m(x_i) w_i e^{y_i \alpha_m C_m(x_i)}$$

Note, that $y_i C_m(x_i) = 1$ if C_m guessed correctly and -1, otherwise. So, if we reorder the points so that the first c are correctly classified by C_m and the remainder aren't we can break up the sum:

$$0 = \sum_{i=1}^{c} w_i e^{\alpha_m} - \sum_{i=c+1}^{n} w_i e^{-\alpha_m}$$

Finally, some math yields

$$\alpha_m = \frac{1}{2} \ln\left(\frac{\sum_{i=c+1}^{n} w_i}{\sum_{i=1}^{c} w_i}\right)$$

which answers our first question.

Works Cited

1. Agrawal, G. L., & Gupta, H. (2013). Optimization of C4.5 Decision Tree Algorithm for Data Mining Application. International Journal of Emerging Technology and Advanced Engineering, 3(3), 341-345. Retrieved May 30, 2017, from http://www.ijetae.com/files/Volume3Issue3/IJETAE_0313_57.pdf
2. Alvarez, S. (n.d.). Decision Tree Pruning based on Confidence Intervals (as in C4.5). Retrieved May 31, 2017, from http://www.cs.bc.edu/~alvarez/ML/statPruning.html
3. FactoMineR. (n.d.). Multiple Correspondence Analysis. Retrieved May 30, 2017, from http://factominer.free.fr/classical-methods/multiple-correspondence-analysis.html
4. Miertschin, S. (n.d.). Decision Trees. Lecture, Houston. Retrieved May 30, 2017, from http://www.uh.edu/~smiertsc/4397cis/C4.5_Decision_Tree_Algorithm.pdf
5. R Documentation. (n.d.). Tea (data). Retrieved May 30, 2017, from http://artax.karlin.mff.cuni.cz/r-help/library/FactoMineR/html/tea.html
6. Rojas, R. (2009). AdaBoost and the super bowl of classifiers a tutorial introduction to adaptive boosting. *Freie University, Berlin, Tech. Rep*.
7. Sharma, S., Agrawal, I., & Sharma, S. (2013). Classification Through Machine Learning Technique: C4.5 Algorithm based on Various Entropies. International Journal of Computer Applications, 82(16), 20-27. Retrieved May 30, 2017, from http://research.ijcaonline.org/volume82/number16/pxc3892444.pdf
8. SRJOGLEKAR246. (2016, March 06). A (small) introduction to Boosting. Retrieved May 27, 2017, from https://codesachin.wordpress.com/2016/03/06/a-small-introduction-to-boosting/
9. Zhu, J., Zou, H., Rosset, S., & Hastie, T. (2009). Multi-class adaboost. *Statistics and its Interface*, *2*(3), 349-360.