Perceptron-Based Oblique Tree (P-BOT)

Ben Axelrod
G.I.T.
baxelrod@cc.gatech

Stephen Campos
G.I.T.
sjcampos@cc.gatech

John Envarli
G.I.T.
envarli@cc.gatech

Abstract

Decision trees are simple and fast data classifiers with the capability of producing human readable output; however they are constrained to single attribute splits, limiting their expressibility. Neural nets have a much richer hypothesis space, but are slow and prone to overfitting. We propose the Perceptron-Based Oblique Tree (P-BOT), a hybrid extension of the C4.5 decision tree. P-BOT addresses the standard decision tree's limitations by placing simple neural networks in the decision nodes. This approach combines the advantages of decision trees and neural nets. Empirical results back this claim.

1 Introduction

Decision trees are simple and fast data classifiers with the nice property of producing readable output and finding the highest info gain attributes to use; however, they are subject to overfitting, and pruning does not yield high accuracy. One reason for these limitations is the fact that decision trees can only classify data with regards to a single parameter per node. Often, it is the relationship between multiple parameters that can more accurately classify the data. On the other hand, artificial neural networks look at all parameters simultaneously, including extraneous attributes that have no effect on the outcome. As a result, they tend to be slow and also suffer from overfitting.

There have been efforts to combine decision trees with other learners. Murphy et al. have developed the OC1 algorithm in [1]. Their algorithm splits data based on a hyperplane that is a linear combination of the attributes. With a conventional decision tree, these hyperplanes are confined to be axis-parallel. They have used a perturbation algorithm to search through the space of possible hyperplanes and they used several impurity measures as goodness measures for a split. A hyperplane can easily be built by using a neural network that has no hidden layers, using a distance squared error metric. Using a neural network also enables creating multiple hyperplanes for a single decision node.

Utgoff and Brodley have developed an algorithm that they describe as a linear machine decision tree in [2]. Their algorithm is also a multivariate tree that uses linear machines at each node. In terms of functionality, its difference from the OC1 algorithm is that it can handle more than two classes, so a node can have as many children as the number of classes.

Another hybrid learner was developed by Kohavi in [3]. His algorithm, NBTree, uses Naïve-Bayes and decision trees together. A regular decision tree is built and pruned, then Naïve-Bayes is run on the leaf nodes. The NBTree algorithm was shown to have higher accuracy than both Naïve-Bayes and C4.5 for many datasets.

There have been efforts to improve the speed and reduce the complexity of neural networks. Hidden layers raise a problem in speed by increasing the number of weight updates, and create a problem in complexity by introducing what is referred to as the moving target problem. Fahlman and Lebiere (1990) have come up with the Cascade-Correlation Learning Architecture to address the problem in [4]. They add the hidden nodes to the network incrementally, and the input weights are frozen at the time of creation. This makes the node a fixed feature and eliminates the moving target problem. It also helps increase speed by limiting the number of weight updates to the number of non-output nodes times the number of output nodes.

In this paper, we describe a method to implement a decision tree classifier that uses simple neural nets in its internal nodes.

2 Perceptron-Based Oblique Tree (P-BOT)

2.1 Description of the Algorithm

At each node, the tree calculates information gain for continuous and discrete attributes as described in the C4.5 algorithm. Gain values are divided by intrinsic information values to penalize for attributes that cause a high number of splits as described in [5].

In addition to considering attribute-based splits, our algorithm also runs a small neural net in each node in the tree. Like all decision tree splits, this neural network is run only on the portion of the dataset that can reach that node. It is constructed as follows: There is one input node corresponding to every continuous attribute normalized between zero and one. For the discrete attributes, there is an input node for every possible attribute value. By setting the appropriate discrete input node to one and the other input nodes for that attribute to zero, it allows the NN to take discrete attributes as input. The net has no hidden nodes, and one output node for each possible classification. The classification of a data point by the neural net is determined by the output node with the highest output. This produces a system that creates n hyperplanes that separate the dataset into n subsets, where n is the number of classes. The neural network was implemented using Gideon Pertzov's code in [6].

The local neural network is trained for a fixed number of iterations, using distance-squared error and gradient descent to update the weights. The information gain of the split is evaluated in the same manner as any other C4.5 split. Therefore the neural network in our algorithm tries to increase information gain by reducing classification error. Intrinsic information measures are also factored in to punish the splits that create many children. Our algorithm uses reduced-error pruning to prevent overfitting. Ten percent of the training set is designated as the validation set for pruning.

Theoretically, perceptron splits should be able to represent and surpass the continuous C4.5 type splits; however, we found that including attribute splits tended to improve the overall accuracy of the tree. Therefore we still use the single attribute splits. Including single attribute splits provides an additional gain by reducing the number of attributes in children nodes.

Figure 1 shows the structure of P-BOT on an example dataset that has one discrete attribute with three values, two continuous attributes, and three classes. Nodes at higher levels are drawn larger to reflect that they have more data points. The root node uses a neural network split. The left child splits on a discrete attribute. The

subset of data at the right child contains only two classes, so the neural network is created accordingly.

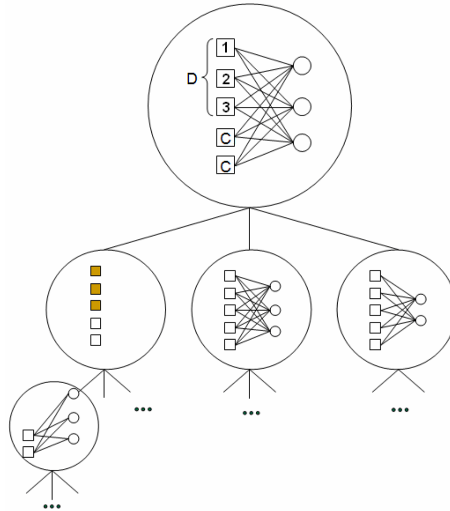

Figure 1: Illustrative example of P-BOT

2.2 Description of the Hypothesis Space

By building hyperplanes recursively, our algorithm separates the data space into convex polytopes, which is defined as, "a finite region of n-dimensional space enclosed by a finite number of hyperplanes" in [7]. Therefore the hypothesis space for our algorithm includes any combination of non-intersecting convex polytopes that cover the entire data space.

Our hypothesis space resembles that of decision trees more closely than neural nets in that the decision tree hypothesis space is effectively a subset of ours. While our algorithm handles all convex polytopes, a typical decision tree is limited to polytopes that are rectangular in form. Multi-layered neural nets, which divide the data along some continuous n-dimensional function, are significantly different. Figure 2 depicts the hypothesis spaces for decision trees and P-BOT for a dataset with two continuous attributes and three classes.

Figure 2: Comparison of hypothesis spaces

2.3 Theoretical Foundation

2.3.1 Difference from a multi-layer neural net

One may think that by forming a tree of simple neural nets, we are still essentially adding multiple classifiers together, resulting in a larger classifier that's no different than a neural network. However, this structure is different from a full neural network because all nodes except the root node use only a subset of the original dataset. In a traditional neural network, every example in the dataset is responsible for the training of the network at any time and any location in the net. For our hybrid tree, only examples reaching a particular node in the tree affect the weights of the net. Additionally, by simplifying the net, we greatly reduce the overall running time of the algorithm. Our results back this claim.

With this in mind, it is easy to see the similarity between our algorithm and other meta-algorithms, such as boosting and cascading. By increasing the importance of points that have been misclassified, boosting focuses on misclassified data. For the neural net, information gain indicates the quality of the classification. If a child has only one classification, it is going to have low entropy and high information gain, and the tree will not go deeper at that node. At nodes that still have high entropy, the neural nets will keep running until the classes are reasonably separated from each other. Once the tree classifies a group of examples successfully, it stops paying attention to them, and it only pays attention to misclassified examples for its weight updates.

Our algorithm differs from boosting and cascading in its approach to classifying datasets. Boosting weights the dataset to increase focus on misclassifications. In contrast, our hybrid tree breaks the dataset into smaller subsets. The smaller subsets facilitate the tree in focusing on the correct decision boundaries. Cascading works similarly here, but it emphasizes complete elimination of false negatives, resulting in what would visually resemble a chain instead of a tree. This makes it optimal in situations where a disproportionably large part of the samples are of a single classification. Our algorithm allows for false classifications at any point, and deals with them by simply extending the tree deeper.

2.3.2 Difference from the OC1 and LMDT Algorithms

The fundamental difference between our algorithm and OC1 arises for datasets with more than two classes. Regardless of the dataset, OC1 builds a single hyperplane at each node, resulting in binary splits. Our algorithm takes the number of classes into account and creates as many children. This makes our tree more efficient for multi-class datasets.

The other major difference is the construction of the hyperplane(s). OC1 uses a perturbation algorithm, a random parallel search based on impurity measures, to approximate the optimal hyperplane. Our algorithm uses gradient descent to minimize classification error to estimate the hyperplane(s). Additionally, our approach still considers axis-parallel, attribute-based splits in addition to hyperplane-based splits. Both OC1 and LMDT do not consider single attribute splits.

Our algorithm also differs from LMDT in the way it builds the separating hyperplanes. Our algorithm uses simple gradient descent search using the derivative of error, as done in neural networks for a set number of epochs. LMDT uses update rules for the weights of the hyperplane set. It focuses on termination conditions and on distance weighting the errors for updates. Instead of focusing on getting the best linear fit, our algorithm focuses on speeding up computation time for the neural network. Our algorithm may or may not build the same hypothesis, but our goal is to build the hypothesis in a shorter time.

2.3.3 Avoiding linearity

Since we are using a linear classifier in the nodes, it is necessary to show that we are not producing a sum of linear systems that is linear itself. We show this by running our algorithm on an XOR dataset, which is not linearly separable. The resulting tree had a neural net that simulated the NAND function at the root. One child was a leaf and the other was a neural net that simulated NOR. The combined affect is an XOR function, which the tree modeled with 100% accuracy.

3 Experiments

3.1 Settings

We tested our algorithm on several datasets from the UCI Machine Learning Repository: mushroom, voting, pen-digits, credit screening, waveform with noise, letter recognition, and income. These datasets have a range of distinguishing characteristics. Some have completely discrete attributes (mushroom, voting), some are all continuous (waveform, pen-digits, letter recognition), and some are a combination of the two (income, credit screening). While most are binary classification problems, pen-digits, waveform, and letter recognition are multi-class. Letter recognition is especially multi-class, with 26 possible classifications.

For the credit screening dataset we removed the data points with unknown values, and for the mushroom dataset we removed the stalk-root attribute because it contained mostly unknown values. This is because our algorithm cannot currently handle unknown values. For baseline algorithms we chose decision trees and neural nets. Both of these algorithms were run in the Weka environment as J48 and Multi-Layer-Perceptron. We used J48, which implements C4.5 with pruning, and trained the neural network with one hidden layer. For both classifiers, we used the default settings to reflect typical usage. For the NN, the hidden layer contained (# attributes + # classes) / 2 nodes, learning rate was 0.3, and 500 epochs were used.

The results are depicted in Table 1.

3.2 Results / Analysis

Table 1: Relative Test Error and Running Time

Dataset	**J48 Error (%) / Time (sec)**	**NN Error (%) / Time (sec)**	**P-BOT Error (%) / Time (sec)**
Mushroom	12.38% / 1	10.85% / 1562	7.09 / 90
Voting	6.25% / 1	5.68% / 296	4.55% / 4
Pen-Digits	7.78% / 2	7.98% / 1085	6.75% / 789
Credit Screening	15.96% / 1	19.15% / 398	15.43% / 8
Income	11.43% / 2	16.40% / 2155	15.79% / 4844
Waveform w/ Noise	25.60% / 2	15.20% / 1663	17.95% / 779
Letter Recognition	28.10% / 2	23.70% / 1579	26.30% / 254

For the pen-digits, mushroom, voting, and credit screening datasets, P-BOT had lower test error than both J48 and NN. Our performance on these datasets can be attributed to our algorithm not overfitting the data. Because our perceptron nodes can represent the decision boundaries better than axis parallel splits, it allows our trees to be shorter. Shorter trees are more general and better by Occam's Razor. For all of the datasets, our trees contained significantly fewer nodes than J48. Additionally, our tree generally did best on datasets containing mostly discrete attributes. Both voting and mushroom datasets are of this type. It is difficult to make any strong claims in terms of the kinds of datasets P-BOT does well on, since these four datasets have a wide range of characteristics.

For the letter recognition and waveform datasets, P-BOT consistently achieved higher testing accuracy than J48, but took longer to run. The opposite was true for the neural net; it had higher error but shorter wall clock time. A similar trend was present with the income dataset; however here P-BOT had a lower error than neural nets, and a higher error than J48. Once again though, P-BOT had an error rate between the two baselines. The mixed performance on these datasets confirms our earlier expectations. By hybridizing two approaches, we were expecting to get an average of their performance in terms of accuracy and speed. The hybrid tree has more expressive power than a decision tree, but at the cost of computation time. On the other hand, a full neural network has more expressive power than our tree. It can draw non-linear boundaries in the feature space as compared with the linear boundaries of P-BOT. P-BOT separates the feature space into many sections to approximate such non-linear boundaries.

We empirically observed test accuracy for different epoch values of our neural nets to see whether overfitting occurs at the node level. Results for the voting dataset are shown in Figure 3 and reveal no overfitting. Analysis of the credit screening dataset revealed similar results. Since these neural nets are capable of linear separations, it is impossible to overfit data that are linearly inseparable. Pruning with a validation set helps avoid overfitting at the tree level. P-BOT consistently achieved lower test error after pruning.

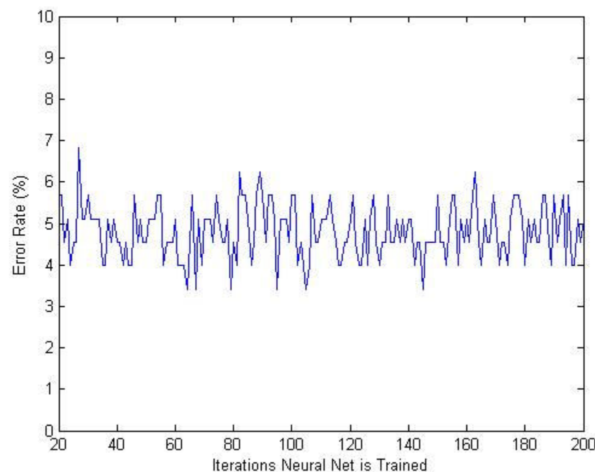

Figure 3: Test error as a function of neural net epochs

P-BOT was considerably slower than J48, even when perceptron based splits were disabled. This is most likely due to a suboptimal implementation of C4.5 used by our hybrid tree, specifically when processing continuous attributes. A more optimized splitting algorithm may lower our running time significantly. For most datasets, enabling the perceptron-based splits did not add much to the running time but reduced error significantly. In fact, on pen-digits and waveform (both with all

continuous attributes and non-binary classifications) adding the perceptron actually sped up the algorithm. This result occurred because the tree can have splits with more than two children. On datasets with many discrete attributes, the perceptron predictably slowed down the run time.

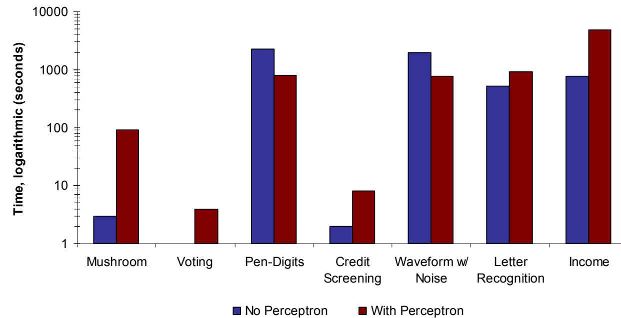

Figure 4: P-BOT with and without perceptron splits

We explored using Naïve-Bayes classifiers at leaf nodes to increase accuracy. In cases with all discrete parameters, adding Naïve-Bayes did not have a significant positive or negative effect. Preliminary tests on continuous datasets indicate that adding Naïve-Bayes improves accuracy slightly when the perceptrons are not trained much; however, accuracy drops as the perceptron is trained beyond about 50 epochs. This may be because as the perceptron nodes fit the data more closely, any benefit obtained by Naïve-Bayes is eliminated.

4 Conclusions and Future Work

Our P-BOT algorithm is a simple way to combine the advantages of decision trees and neural networks. When compared with typical decision trees, it is consistently more accurate. With neural nets, it usually achieves an accuracy that is either higher, or at least comparable, while running significantly faster. Furthermore, our algorithm appears to be robust against overfitting, since varying the number of perceptron training epochs within decision nodes had little impact on accuracy.

In the future, we seek to explore what exact parameters result in higher accuracy and faster runtime. For example, since P-BOT's accuracy does not seem to change much with the number of perceptron training epochs, how few epochs can be used without jeopardizing accuracy? Our results were typically obtained at 100 epochs, but running fewer would reduce the overall runtime. In this situation, would adding in Naïve-Bayes classifier for leaf nodes be a good policy? If the trends observed in our preliminary trials hold, this many be the case; however more research is necessary to in order to reach a conclusive answer. Finally, we would like to empirically compare our algorithm with other hybrid algorithms, such as OC1 and LMDT. Implementations of these algorithms were not readily available so such a comparison was not possible at this time.

References

[1] S. Murthy, S. Kasif, S. Salzberg, and R. Beigel. (1993) *OC1: Randomized induction of oblique decision trees.* In Proceedings of the Eleventh National Conference on Artificial Intelligence, pages 322--327, Boston, MA, 1993. MIT Press.

[2] Utgoff, P. E., & Brodley, C. E. (1991). *Linear machine decision trees*. Tech. rep. 10, University of Massachusetts at Amherst.

[3] Kohavi, R. (1996). *Scaling up the accuracy of naive-Bayes classifiers: a decision-tree hybrid.* Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. (1996) 202--207. The AAAI Press.

[4] S. E. Fahlman and C. Lebiere. (1990) *The cascade-correlation learning architecture.* Technical Report CMU-CS-90-100, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1990.

[5] Mitchell, T. M. (1997) *Machine Learning*. McGraw-Hill.

[6] Gideon Pertzov. http://gpdev.net/NeuroDriver_bpnet.html.

[7] http://mathworld.wolfram.com/Polytope.html