A Review of Online Decision Tree Learning Algorithms

Johns Hopkins University Department of Computer Science

Corbin Rosset

June 17, 2015

Abstract

This paper summarizes the most impactful literature of online decision tree (ODT) algorithms. Except for the introduction and conclusion, and the manner in which the work of other authors is presented, this paper is by no means original. Some algorithms are discussed more fully than others based on their novelty and relevance. It is written for Dr. Raman Arora in the Johns Hopkins University Computer Science Department to set the foundation for researching new algorithms.

1 Introduction

The purpose of online learning algorithms is to learn a concept from a streaming input of examples without assuming any underlying distribution, and hopefully without any dependence on the order, rate, or noisiness of the examples. Unlike batch algorithms such as ID3 and C4.5, which have access to all the examples in memory[1], online algorithms only learn incrementally–viewing each example only once and then discarding it. Online decision tree (ODT) algorithms attempt to learn a decision tree classifier from a stream of *labeled* examples, with the goal of matching the performance (accuracy, precision, recall, etc) of a related batch decision tree learning algorithm with reasonably expeditious runtime, or at least no slower than running a batch algorithm.[2]

The naive ODT learning algorithm is to re-run a canonical batch algorithm, like C4.5, on the collection of examples seen so far whenever a new example is delivered.

[1] or disk algorithms like SLIQ and SPRINT, which store and re-read examples from disk

[2] Runtime has varied definitions for ODT algorithms depending on their behavior. Often the "update" time for handling a single example is quite small, but offset by requiring more examples to reach an optimal decision tree [18], and depending on factors like concept drift, some ODT's make expensive tree manipulations with irregular puntuality. So runtime must account for update, punctual, and global resource expenditures. Curiously, the author has not seen amortized analysis employed often in runtime analyses, although it seems natural to do so.

Bear in mind the problems hindering this naive approach often haunt more sophisticated algorithms: the algorithm becomes too expensive to compute under high streaming rates or if it is forced to consider every example; the algorithm may require more examples than its non-incremental cousin to reach a stable concept; the algorithm may accumulate outdated concepts that decrease performance; the algorithm may not react quickly enough to rapidly changing concepts; the algorithm could be susceptible to noise, missing data, overfitting, or adversarial orderings of input data (resulting in "thrashing" that render a concept unlearnable); and finally, the concept to be learned may depend on an unknown minumum number of attributes. Of course, the practical impetus for streaming algorithms is that the sheer quantity of data generated in the modern era (avoiding the term "big data") often cannot fit in main memory. Thus, one would "prefer an incremental algorithm, on the assumption that it is more efficient to revise an existing hypothesis than it is to generate a new one each time an example is observed" [20]. The algorithms summarized in the (mostly) chronologically ordered sections below address the aforementioned problems; it turns out that concept drift, or the non-stationarity of a distribution over time, is most formidable [19].

It should be noted that the data mining field has taken interest in making ODT algorithms scalable, distributed, and memory efficient, often contributing more to the field than machine learning researchers. The more recent papers introduce fewer groundbreaking algorithms like VFDT, but rather optimize them for large disk-resident datasets, augment them with better pruning or sampling paradigms, or implement an ODT algorithm in the context of a parallel programming model. Lastly, although not discussed here, investigating neural networks promises to be equally fruitful for streaming algorithms.

2 Schlimmer & Fishers ID4

ID4, [18] is the first substantial step away from batch algorithms, but it is not guaranteed to learn all the concepts that it's non-incremental brother, ID3, can learn.[3] To handle incremental updates, all quantities needed to compute the information gain for any attribute node is kept at that node. This information consists of class counts for each value associated with each test attribute as it is encountered in the tree traversal. If any leaf is encountered at which both postive and negative examples are currently classified, then it must be transformed into a test node. To select the most informative attribute from the set of all unused attributes, the stored attribute-value counts are leveraged to calculate information gain in the traditional fashion. There is also a χ^2 test to mitigate overfitting–to prevent replacement of a leaf node by a decision node if there isn't an advantage in doing so. Recursion terminates when all instances at a node are of the same class (a leaf node), a new attribute cannot be reliably chosen (fails χ^2), or when all attributes are used and maximally informative.

If a test attribute is not the most informative, then it must be replaced by the maximal attribute from a node below it. But what about the subtree below the

[3]ID3' is the bruteforce algorithm: rebuild the tree with ID3 after every new instance

node in question?. Unfortunately, this part of the algorithm (step 3d) is bit poorly conceived for a few reasons. Firstly, it discards the positive/negative counts along with the whole subtree, and requires a "remedial" period to rebuild the tree and the counts. Although Schlimmer and Fisher argue this occurs infrequently, and the important nodes higher in the tree are more likely to be stable, this renders some concepts that ID3 could learn unlearnable by ID4. Thrashing: whenever the relative ordering of the maximal attributes along a path in the tree changes often during training, that is, at least once during a given "remedial" period, it is possible the algorithm will not converge to *any* ordering. If the relative ordering does not stabilize with training, then subtrees will be discarded repeatedly, rendering certain concepts unlearnable by the algorithm. Thrashing usually occurs deeper in the decision tree– three or more levels below the current node and in which there is no stable best test attribute for the node.

Inputs: *A decision tree, One instance.*
Output: *A decision tree.*

1. If this instance is positive, increment the total number of positive instances. Otherwise increment the number of negative instances.
2. If all of the instances are positive or negative then return the decision tree.
3. Else
   (a) Compute the expected information score.
   (b) For each attribute, for each value present in the instance, increment either the number of positive or negative instances.
   (c) Compute the information scores for all attributes.
   (d) If there is no root or the maximal attribute is not the root, then build a new tree.
      i. If the maximal attribute is χ^2 dependent then make it the root of this tree.
      ii. Make a test link from the root for every value of the root attribute.
   (e) Go to step 1 with the subtree found by following the link for the root attribute's value in this instance.

Table 1: Pseudo code for incremental ID4.

The number of examples needed to make ID4's tree stable is the sum of all instances required to stably learn all subtree nodes, while the number of instances to learn a stable ID3 tree is simply the maximum number of instances to stably learn any node in the tree. It should be noted that the computational cost per instance is much lower for ID4 than ID3, but as stated more instances were required.

3 ID5R

Despite being a rather poorly written paper, it does present the first successful attempt of extending traditional batch algorithms into online learners [20] because it restructures rather than discards inaccurate subtrees. It is guaranteed to build the same decision tree as ID3 for any given set of training instances, again, by maintaining counts to compute the information gain for any attribute at a node, making it possible to update the test attribute at a node when it becomes outdated.

Instead of discarding the subtrees below the old test attribute, ID5R restructures the tree so that the desired test attribute is at the root. The restructuring process, called a pull-up, is a tree manipulation that preserves consistency with the observed training instances, and that brings the indicated attribute to the root node of the tree or subtree. The advantage of restructuring the tree is that it lets one recalculate the various positive and negative counts during the tree manipulations, without reexamining the training instances. During pull-up, one recursively checks the accuracy of the the subtrees, restructuring them as necessary so that every test attribute at a node has the highest information gain. Unfortunately the paper offered little more detail than this (see ITI).

The ID5R algorithm builds the same tree as the basic ID3 tree construction algorithm, given the same instances and given the same method for breaking ties among equally good attributes. Note the instances at a node are described only by the attribute-value pairs that have not been determined by tests above the node. This means that the instance descriptions are reduced by one attribute-value pair for each attribute tested above in the tree.

3.1 Performance Metrics

The first metric the authors compute is the total number of attribute comparisons while building the tree. This measures the cost of maintaining the positive and negative instance counts in terms of the number of additions performed. Each addition needed to compute such a count is called an instance-count addition, includes the cost of adding counts during the tree transposition process. The second metric is the cost of attribute selection in terms of the number of E-score (or information gain) computations. This activity is not very costly for the basic ID3 tree construction algorithm, but should be considered for the ID5R and ID4 algorithms, which compute the E-score for every non- test attribute at every decision node in the tree.

Table 3. The ID5R tree-update algorithm.

1. If the tree is empty, then define it as the unexpanded form, setting the class name to the class of the instance, and the set of instances to the singleton set containing the instance.

2. Otherwise, if the tree is in unexpanded form and the instance is from the same class, then add the instance to the set of instances kept in the node.

3. Otherwise,

   (a) If the tree is in unexpanded form, then expand it one level, choosing the test attribute for the root arbitrarily.

   (b) For the test attribute and all nontest attributes at the current node, update the count of positive or negative instances for the value of that attribute in the training instance.

   (c) If the current node contains an attribute test that does not have the lowest E-score, then

      i. Restructure the tree so that an attribute with the lowest E-score is at the root.

      ii. Recursively reestablish a best test attribute in each subtree except the one that will be updated in step 3d.

   (d) Recursively update the decision tree below the current decision node along the branch for the value of the test attribute that occurs in the instance description. Grow the branch if necessary.

Table 4. The ID5R pull-up algorithm.

1. If the attribute a_{new} to be pulled up is already at the root, then stop.

2. Otherwise,

   (a) Recursively pull the attribute a_{new} to the root of each immediate subtree. Convert any unexpanded tree to expanded form as necessary, choosing attribute a_{new} as the test attribute.

   (b) Transpose the tree, resulting in a new tree with a_{new} at the root, and the old root attribute a_{old} at the root of each immediate subtree.

Table 6. Summary of worst-case complexities.

	Instance-count additions	E-score calculations
ID3	$O(n \cdot d^2)$	$O(b^d)$
ID5	$O(n \cdot b^d)$	$O(n \cdot d^2)$
ID5R	$O(n \cdot d \cdot b^d)$	$O(n \cdot b^d)$
ID3′	$O(n^2 \cdot d^2)$	$O(n \cdot b^d)$

The worst case complexities table shows the number of information gain (E-score) calculations and the cost of maintaining the positive and negative instance counts in terms of the number of additions performed. n is the number of instances, d is the number of attributes, b is the maximum number of values for any given attribute.

4 ID5R version 2: ITI

Curiously, another paper by Utgoff [21] does a much better job at explaining tree transpositions for ID5R, but it was published 8 years after ID5R with almost no change to the algorithm, other than renaming "transpositions" as "mappings" from an existing tree and a new training example to a new tree.

Adding a new example: If the example has the same class as the leaf, the example is simply added to the set of examples saved at the leaf node. If the example has a different class label from the leaf, the algorithm attempts to turn the leaf into a decision node, picking the best test according to the test-selection metric. The examples saved at the node that was just converted from a leaf node to a decision node are then incorporated recursively by sending each one down its proper branch according to the new test.

During the recursive transposition, it may be that the subtrees have been transposed as a by-product of bringing the desired test to the root. At each decision node that requires a different test be installed, the algorithm transposes the tree to install the best test at the node. It could become costly to check every decision node of the subtrees after a transposition. Often, a subtree is not touched during a transposition. To this end, a marker is maintained in each decision node that indicates whether the choice of the installed test is stale. At the root, identify the desired test and install it via recursive transposition; for each subtree, if it is marked stale, then recursively identify its desired test and install it.

While ID5R does exhibit a lower average incremental update cost of updating over building a new decision tree from scratch, it is not necessarily true that the sum of the incremental costs are lower because we care only about the cost of being brought up to date at a particular point in time. The update cost is independent of the number of training examples on which the tree is based, and the resulting tree is independent of the order in which examples are received. The glaring flaw of this algorithm is that for large data sets, saving the examples at the node will not be possible. The rest of the paper explains other luxury features like error correcting modifications, noise tolerance, missing class labels,

Table 1. Procedure add_example_to_tree()

```
add_example_to_tree(node,example)
  if node is NULL
    then convert node to an empty leaf

  if node is a leaf
    then save example at node
         if node should be converted to a decision node
           then construct information at node
                mark node fresh
                for each example j saved at node
                  if test_is_true(node,j)
                    then add_example_to_tree(node->left,j)
                    else add_example_to_tree(node->right,j)
    else update information at node
         mark node stale
         if test_is_true(node,example)
           then add_example_to_tree(node->left,example)
           else add_example_to_tree(node->right,example)
```

5 Very Fast Decision Tree (VFDT)

VFDT[8] learns decision tree incrementally from streaming data using constant time and memory per example. The revolutionary construct used is a Hoeffding tree, which is learned in constant time per example (worst case proportional to the number of attributes) while exhibiting nearly the same structure as a batch learner, given enough examples. Only the tree and necessary statistics are stored in memory, and examples can be processed faster than they can be read from disk. Since the algorithm is designed to handle possibly infinite examples, the authors note it may be sufficient to consider only a small subset of examples that pass through a node, and thus sample-based decision tree constructions were born. The first examples to arrive in the data stream are used to choose the split attribute at the root; subsequent ones are passed through the induced portion of the tree until they reach a leaf, are used to choose a split attribute there, and so on. To determine the number of examples needed for each decision, VFDT uses a statistical result known as Hoeffding bounds or additive Chernoff bounds: after n independent observations of a real valued random variable r with range R, the Hoeffding bound ensure that, with confidence $1-\delta$, the true mean of r is at least $\overline{r}-\epsilon$, where $\overline{r}$ is the sample mean and

$$\epsilon = \sqrt{\frac{R^2 ln1/delta}{2n}}$$

The elegance of this equation is that it is independent of the underlying distribution of the stream, at the cost of being a bit more conservative.

Furthermore, with high probability, the attribute chosen with n examples is the same as that chosen after seeing infinite examples. If $IG^*(x)$ is true information gain of example x, and $IG(x)$ is the sample mean information gain, then $IG(x_1) > IG(x_2)$ with probability $1-\delta$ given n examples have been seen at this node and $\Delta IG = IG(x_1)$ - $IG(x_2) > \epsilon$. Thus with $1-\delta$ probability, x_1 is the correct attribute on which to

split the tree. Note, it is not valid to split a node until enough examples have been such that ϵ is less than ΔIG.

Table 1: The Hoeffding tree algorithm.

Inputs:	S	is a sequence of examples,
	$\mathbf{X}$	is a set of discrete attributes,
	$G(.)$	is a split evaluation function,
	δ	is one minus the desired probability of choosing the correct attribute at any given node.
Output:	HT	is a decision tree.

Procedure HoeffdingTree $(S, \mathbf{X}, G, \delta)$
Let HT be a tree with a single leaf l_1 (the root).
Let $\mathbf{X_1} = \mathbf{X} \cup \{X_\emptyset\}$.
Let $\overline{G}_1(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class in S.
For each class y_k
  For each value x_{ij} of each attribute $X_i \in \mathbf{X}$
    Let $n_{ijk}(l_1) = 0$.
For each example $(\mathbf{x}, y_k)$ in S
  Sort $(\mathbf{x}, y)$ into a leaf l using HT.
  For each x_{ij} in $\mathbf{x}$ such that $X_i \in \mathbf{X}_l$
    Increment $n_{ijk}(l)$.
  Label l with the majority class among the examples seen so far at l.
  If the examples seen so far at l are not all of the same class, then
    Compute $\overline{G}_l(X_i)$ for each attribute $X_i \in \mathbf{X}_l - \{X_\emptyset\}$ using the counts $n_{ijk}(l)$.
    Let X_a be the attribute with highest $\overline{G_l}$.
    Let X_b be the attribute with second-highest $\overline{G}_l$.
    Compute ϵ using Equation 1.
    If $\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ and $X_a \neq X_\emptyset$, then
      Replace l by an internal node that splits on X_a.
      For each branch of the split
        Add a new leaf l_m, and let $\mathbf{X_m} = \mathbf{X} - \{X_a\}$.
        Let $\overline{G}_m(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class at l_m.
        For each class y_k and each value x_{ij} of each attribute $X_i \in \mathbf{X_m} - \{X_\emptyset\}$
          Let $n_{ijk}(l_m) = 0$.
Return HT.

The algorithm contains a mechanism for pruning, attribute-value counts at each node for statistical computations. The authors prove that the quality of the tree asymptotically approaches that exhibited by a batch learner by defining a loss function. Empirical data shows that batch algorithms perform better in their own domain: their accuracy is higher when n is small, but after a few dozen thousand examples, VFDT far outperforms its batch counterparts. They also prove that if d is the maximum number of attributes, v is the maximum number of values for any attribute, and c is the number of classes, then VFDT requires $O(dvc)$ memory per leaf, which is independent of the number of attributes seenIn fact, an exponential decrease in δ

can be obtained by a linear increase in n.

Some other features of the algorithm are: 1) since IG computation is the most expensive step, only recompute it after a certain number of new features have been seen; 2) the least promising subtrees can be deactivated to free up memory (with periodic checks for potential reactivation by scanning all nodes); 3) interestingly, the Hoeffding tree can be "seeded" by a small tree generated by a batch algorithm to shorten the learning curve; and 4) if data is arriving rather slowly, old examples can be re-scanned to improve accuracy.

However, the authors assume that all concepts are stationary. A year later in 2001, however, their algorithm was updated to incorporate concept drift, hence Concept-adapting Very Fast Decision Tree Learner.

6 CVFDT

The primary flaw in VFDT was the assumption of concept stationarity over time. It is more accurate to assume that data was generated by a series of concepts, or by a concept function with time-varying parameters, thus have a sliding window of training examples to build, modify, and keep up to date its Hoeffding tree. The Concept-adapting Very Fast Decision Tree (CVFDT) algorithm has the same speed, accuracy and precision as VFDT, but learns the model by reapplying VFDT to a moving window of examples every time a new example arrives–with $O(1)$ complexity per example, as opposed to $O(w)$ where w is the window size. The window must be sufficiently smaller than the rate of concept drift but large enough to fully learn the current concept. The computational cost of reapplying a learner may be prohibitive if the rate of the example stream or the concept drift is too high, even though each example is still processed in constant time.

The update step for a single example is similar to that of VFDT: "increment the counts corresponding to the new example, but decrement the counts corresponding to the oldest example in the window (which now needs to be forgotten"[4]). This will statistically have no effect if the underlying concept is stationary. If the concept is changing, however, some splits that previously passed the Hoeffding test will no longer do so, because an alternative attribute now has higher gain (or the two are too close to tell). In this case CVFDT begins to sprout a hiddent, alternative subtree' to replace the obsolete one [7]. CVFDT periodically scans the internal nodes looking for ones where the chosen split attribute would no longer be selected (note that the relevant statistics are stored at every node as well as a finite list of possible alternate split attributes). In this "testing" phase, the next m examples are used to compare the still-hidden alternate attributes against the current one; and separately, those

[4] "Forgetting an old example is slightly complicated by the fact that the tree may have grown or changed since the example was initially incorporated. Therefore, nodes are assigned a unique, monotonically increasing ID as they are created. When an example is added to window W, the maximum ID of the leaves it reaches in a branch and all alternate trees is recorded with it. An example's effects are forgotten by decrementing the counts in the sufficient statistics of every node the example reaches whose ID is less than that stored"

subtrees whose accuracy has not been improving are pruned. When it finds the current split attribute to be less accurate than an alternate, "CVFDT knows that it either initially made a mistake splitting on that attribute (which should hap- pen less than δ % of the time), or that something about the process generating examples has changed". And thus the hidden subtree replaces the old one with no "remedial" period needed to thicken it. Choosing a new split attribute is similar to choosing an initial split attribute for a leaf, but with tighter criteria to avoid excessive tree growth.

Table 2: The CVFDT algorithm.

Inputs:	S	is a sequence of examples,
	$\mathbf{X}$	is a set of symbolic attributes,
	$G(.)$	is a split evaluation function,
	δ	is one minus the desired probability of choosing the correct attribute at any given node,
	τ	is a user-supplied tie threshold,
	w	is the size of the window,
	n_{min}	is the # examples between checks for growth
	f	is the # examples between checks for drift.
Output:	HT	is a decision tree.

Procedure CVFDT$(S, \mathbf{X}, G, \delta, \tau, w, n_{min})$
/* Initialize */
Let HT be a tree with a single leaf l_1 (the root).
Let $ALT(l_1)$ be an initially empty set of alternate trees for l_1.
Let $\overline{G}_1(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class in S.
Let $\mathbf{X_1} = \mathbf{X} \cup \{X_\emptyset\}$.
Let W be the window of examples, initially empty.
For each class y_k
  For each value x_{ij} of each attribute $X_i \in \mathbf{X}$
    Let $n_{ijk}(l_1) = 0$.
/* Process the examples */
For each example $(\mathbf{x}, y)$ in S
  Sort $(\mathbf{x}, y)$ into a set of leaves L using HT and all trees in ALT of any node $(\mathbf{x}, y)$ passes through.
  Let ID be the maximum id of the leaves in L.
  Add $((\mathbf{x}, y), ID)$ to the beginning of W.
  If $|W| > w$
    Let $((\mathbf{x}_w, y_w), ID_w)$ be the last element of W
    ForgetExamples$(HT, n, (\mathbf{x}_w, y_w), ID_w)$
    Let $W = W$ with $((\mathbf{x}_w, y_w), ID_w)$ removed
  CVFDTGrow$(HT, n, G, (\mathbf{x}, y), \delta, n_{min}, \tau)$
  If there have been f examples since the last checking of alternate trees
    CheckSplitValidity(HT, n, δ)
Return HT.

Table 3: The CVFDTGrow procedure.

Procedure CVFDTGrow$(HT, n, G, (\mathbf{x}, y), \delta, n_{min}, \tau)$
Sort $(\mathbf{x}, y)$ into a leaf l using HT.
Let P be the set of nodes traversed in the sort.
For each node l_{pi} in P
  For each x_{ij} in $\mathbf{x}$ such that $X_i \in \mathbf{X}_{l_p}$
    Increment $n_{ijy}(l_p)$.
  For each tree T_a in $ALT(l_p)$
    CVFDTGrow$(T_a, n, G, (\mathbf{x}, y), \delta, n_{min}, \tau)$
Label l with the majority class among the examples seen so far at l.
Let n_l be the number of examples seen at l.
If the examples seen so far at l are not all of the same class and n_l mod n_{min} is 0, then
  Compute $\overline{G}_l(X_i)$ for each attribute $X_i \in \mathbf{X}_l - \{X_\emptyset\}$ using the counts $n_{ijk}(l)$.
  Let X_a be the attribute with highest $\overline{G}_l$.
  Let X_b be the attribute with second-highest $\overline{G}_l$.
  Compute ϵ using Equation 1 and δ.
  Let $\Delta\overline{G}_l = \overline{G}_l(X_a) - \overline{G}_l(X_b)$
  If $((\Delta\overline{G}_l > \epsilon)$ or $(\Delta\overline{G}_l <= \epsilon < \tau))$ and $X_a \neq X_\emptyset$, then
    Replace l by an internal node that splits on X_a.
    For each branch of the split
      Add a new leaf l_m, and let $\mathbf{X_m} = \mathbf{X} - \{X_a\}$.
      Let $ALT(l_m) = \{\}$.
      Let $\overline{G}_m(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class at l_m.
      For each class y_k and each value x_{ij} of each attribute $X_i \in \mathbf{X_m} - \{X_\emptyset\}$
        Let $n_{ijk}(l_m) = 0$.

Table 4: The ForgetExample procedure.

Procedure ForgetExample$(HT, n, (\mathbf{x}_w, y_w), ID_w)$
  Sort $(\mathbf{x_w}, y_w)$ through HT while it traverses leaves with id $\leq ID_w$,
  Let P be the set of nodes traversed in the sort.
  For each node l in P
    For each x_{ij} in $\mathbf{x}$ such that $X_i \in \mathbf{X}_l$
      Decrement $n_{ijk}(l)$.
    For each tree T_{alt} in $ALT(l)$
      ForgetExample$(T_{alt}, n, (\mathbf{x}_w, y_w), ID_w)$

Table 5: The CheckSplitValidity procedure.

Procedure CheckSplitValidity(HT, n, δ)
For each node l in HT that is not a leaf
  For each tree T_{alt} in $ALT(l)$
    CheckSplitValidity(T_{alt}, n)
  Let X_a be the split attribute at l.
  Let X_n be the attribute with the highest $\overline{G}_l$
      other than X_a.
  Let X_b be the attribute with the highest $\overline{G}_l$
      other than X_n.
  Let $\Delta\overline{G}_l = \overline{G}_l(X_n) - \overline{G}_l(X_b)$
  If $\Delta\overline{G}_l \geq 0$ and no tree in $ALT(l)$ already splits on
        X_n at its root
    Compute ϵ using Equation 1 and δ.
    If $(\Delta\overline{G}_l > \epsilon)$ or $(\epsilon < \tau$ and $\Delta\overline{G}_l \geq \tau/2)$, then
      Let l_{new} be an internal node that splits on X_n.
      Let $ALT(l) = ALT(l) + \{l_{new}\}$
      For each branch of the split
        Add a new leaf l_m to l_{new}
        Let $\mathbf{X_m} = \mathbf{X} - \{X_n\}$.
        Let $ALT(l_m) = \{\}$.
        Let $\overline{G}_m(X_\emptyset)$ be the $\overline{G}$ obtained by predicting
          the most frequent class at l_m.
        For each class y_k and each value x_{ij} of each
            attribute $X_i \in \mathbf{X_m} - \{X_\emptyset\}$
          Let $n_{ijk}(l_m) = 0$.

The window size parameter, w, does not fit all concepts, so it needs to be dynamic. "For example, it may make sense to shrink w when many of the nodes in HT become questionable at once, or in response to a rapid change in data rate, as these events could indicate a sudden concept change. Similarly, some applications may benefit from an increase in w when there are few questionable nodes because this may indicate that the concept is stable – a good time to learn a more detailed model. CVFDT is able to dynamically adjust the size of its window in response to user-supplied events, which are caught by hook functions.

Again, many of the advantages of VFDT apply to CVFDT, such as a memory requirement independent of the number of examples seen. Some criticisms of CVFDT include: the discarding subtrees that are out of date rather than persisting them for some time in case they need to be "reactivated" as the FLORA algorithm does. Some of these subtrees might be useful in the future because there might be repeating patterns in the data. Also, the dynamic window size may not be flexible enough to match the accuracy obtained if the concept-drift window sizes were known ahead of time, and some argue there was not enough testing to this end. CVFDT is unable to handle data with an uncertain attribute, something UCVFDT addresses [12]

7 Online Adaptive Decision Tree, OADT

OADT a classification algorithm that maintains the structure of a tree and employs the gradient descent learning algorithm like neural networks for supervised learning in the online mode [8]. Inputs to a node include an activation from its parent as well as the example (the root only receives the example). Unlike conventional feedforward neural networks, the intermediate nodes are not "hidden" in that they can see the raw input example as it traverses down the tree. Activations functions are sigmoidal, and they are mutliplied when propogated: node i delivers to its children the product its own activation value and that of its parent, thus the activation at a leaf is the product of the activations produced by the intermediate nodes along the path to it. If an leaf node's class 'is 'negative", then the activation is multiplied by -1, and unchanged if it's "positive". As for the tree structure, OADT is a complete binary tree of depth l where l is specified beforehand. It has a set of intermediate nodes D, whose size is $2l$ 1 and a set of leaf nodes L where $L = 2l$, such that the total number of nodes in the tree is $2l + 1$. Each intermediate node i in the tree stores a decision hyperplane as a normalized n-dimensional weight vector for n-dimensional input examples, and it stores the sigmoidal activation function.

The decision region is always explicitly formed by the collective activation of a set of leaf nodes. The updating procedure is rather complicated for discussion here, but involves a modified gradient descent algorithm. There is a surprising resilience to overfitting, as the complexity of the decision region saturates after a certain depth of the tree (specified at runtime). OADT is more robust to concept drift and infers the same decision tree as the nonincremental ID3 algorithm

8 Numerical Interval Pruning (NIP) for VFDT

The authors [9] present an improvement to the VFDT algorithm that reduces runtime by 39%, and reduces by 37% the number of samples needed to build the same model by more aggressive sampling. These improvements were not applied to CVFDT, however, they could and should be easily extended.

They replace the Hoeffding bounds for sampling with the multivariate delta method, which is proven to not increase the number of samples needed. Essentially the authors notice that the difference of the expected gains of a function of two attributes x and y is a normal random variable. Then they simply find the appropriate bounds using a normal test (why they didn't use a chi-squared is an open question). With no more observed examples than Hoeffding bounds require, they can say the expected gain of attribute x is greater than that of y with some probability δ. In practice, the multivariate delta method drastically improves runtime without sacrificing performance. See the cited paper for details.

9 Uncertainty-handling and Concept-adapting Very Fast Decision Tree, UCVFDT

A more reknowned paper [16] introduces the topic of uncertainty applied to decision trees as a whole. And although poorly written, the paper [12] introducing the UCVFDT is worth mentioning because it proposes a solution to this problem specifically for the X-VFDT lineage of algorithms. Uncertainty is defined as fuzzy, imprecise, or unknown attribute values in examples; values with ranges like 5-10 for a quantity that should be an exact number such as 7.85 are an example. They simply define and defend a metric known as Uncertain Information Gain incorporated into the algorithm in place of other split tests.

10 Ensemble Classifiers

Ensemble classifiers incorporate many ODT sub-algorithms with different weights to collectively classify streaming data. Instead of continuously revising a single model, they train and pool a multitdude of classifiers from sequential data chunks in the stream before a decision is made. Combining classifiers can be done via averaging, or more commonly, weighted averaging [5]. Ensemble classifiers can outperform single classifiers in mid-to-low concept-drift environments.

Google's PLANET [15] (Parallel Learner for Assembling Numerous Ensemble Trees) implements each of these sub-algorithms in a map-reduce model to handle truly massive datasets. While a good read, the paper focuses narrowly on utilizing the the properties of its distributed computation model.

11 Conclusions

For development and testing of any new online decision tree learning algorithms, ID5R should be the benchmark because of its simplicity and satisfactory performance. By Occam's razor, any algorithm more complicated than ID5R that does not have significantly better performance or runtime should not be used. Overall, the author belives that the CVFDT algorithm [8] is most worthy of implementing because of it's innovative mathematical approach (Hoeffding bounds), modular subroutines that are easily interchanged and updated, and novel structural features like node ID's and hidden alternate subtrees.[6] Moreover, the research community has heavily revised and optimized VFDT subroutines, like replacing Hoeffding bounds with the multivariate delta method [9] and adding uncertainty handling to information gain calculation [16] [12]. There were some other moderately useful modifications to VFDT not discussed here, such as that proposed by Gama [5].

[5] where the weight is inversely proportional to the expected error

[6] CVFDT seemed to capstone the field of serial online algorithms, however, there was some criticism by Cal and Wozniak in 2013 [3]

Then the advent of parallel computing introduced a flurry of parallelized online decision tree learning algorithms, leading to ScalParC [11], SPDT [2] [3] and Google's PLANET [15], each of which exploit in one way or another a parallel architecture like map-reduce. These algorithms are not discussed here because they lean more towards systems, architecture, and data mining, but they are certainly worth investigating because of the attractive performance distributing computation. In fact the future of online algorithms probably rests, at least in practice, in parallel computing. For example, SPDT [2] is empirically shown to be as accurate as a standard decision tree classifier, while being scalable for processing of streaming data on multiple processors. SPIES is more of a mining algorithm for parallel disk-resident datasets, but it combines RainForest [6] paradigm with Hoeffding sampling. Most of the modern literature on decision trees exploits a specific parallel computation paradigm or platform, but retain key elements of the aforementioned algorithms, such as storing attribute-value counts. If one actually wants an algorithm to be marketable, it should probably be distributed or at least scalable in some way.

References

[1] Basak, Jayanta. "Online Adaptive Decision Trees." Neural Computation 16.9 (2004): 1959-981. Web.

[2] Ben-Haim, Yael and Tom, Elead-Tov (2010) "A Streaming Parallel Decision Tree Algorithm" Journal of Machine Learning Research, 11, 849-872: http://www.jmlr.org/papers/volume11/ben-haim10a/ben-haim10a.pdf

[3] Cal, Piotr, and Micha Woniak. "Parallel Hoeffding Decision Tree for Streaming Data." Distributed Computing and Artificial Intelligence Advances in Intelligent Systems and Computing (2013): 27-35

[4] Cohen, L., Avrahami, G., Last, M., Kandel, A. (2008) Info-fuzzy algorithms for mining dynamic data streams. Applied soft computing. 8 1283-1294

[5] Gama, Joo, Ricardo Rocha, and Pedro Medas. "Accurate Decision Trees for Mining High-speed Data Streams." Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '03 (2003)

[6] Gehrke, Johannes, Raghu Ramakrishnan, and Venkatesh Ganti. "RainForestA Framework for Fast Decision Tree Construction of Large Datasets." Data Mining and Knowledge Discover 4 (2000): 127-62. Print.

[7] Hulten, Geoff, Laurie Spencer, and Pedro Domingos. "Mining Time-changing Data Streams." Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '01 (2001)

[8] Hulten, Geoff and Pedro Domingos. "Mining High-Speed Data Streams." Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '00 (2000)

[9] Jin, Rouming, and Gagan Agrawal. "Efficient Decision Tree Construction on Streaming Data." Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '03 (2003)

[10] Jin, Ruoming, and Gagan Agrawal. "Communication and Memory Efficient Parallel Decision Tree Construction." Proceedings of the 2003 SIAM International Conference on Data Mining (2003)

[11] Joshi, M.v., G. Karypis, and V. Kumar. "ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets." Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998): n. pag. Web.

[12] Liang, Chunquan, Yang Zhang, and Qun Song. "Decision Tree for Dynamic and Uncertain Data Streams." Asian Conference on Machine Learning (2010): 209-24. Print.

[13] Mitchell, Tom M. Machine Learning. New York: McGraw-Hill, 1997.

[14] Mohri, Mehryar, Afshin Rostamizadeh, and Ameet Talwalkar. Foundations of Machine Learning. Cambridge, MA: MIT Press, 2012.

[15] Panda, Biswanath, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. "Planet." Proceedings of the VLDB Endowment Proc. VLDB Endow. 2.2 (2009): 1426-437.

[16] B. Qin, Y. Xia, and F. Li. DTU: A Decision Tree for Uncertain Data. In Advances in Knowledge Discovery and Data Mining.(PAKDD09), pages 415, 2009a.

[17] Russell, Stuart J, and Peter Norvig. Artificial Intelligence : a Modern Approach. 3rd ed. Upper Saddle River, N.J.: Prentice Hall, 2010.

[18] Schlimmer, Jeffrey, and Douglas Fisher. "A Case Study of Incremental Concept Induction." Proceedings of the Fifth National Conference on Artificial Inteligence (1986): 495-501. Print.

[19] Tsymbal, Alexey. "The Problem of Concept Drift: Definitions and Related Work." Department of Computer Science, Trinity College Dublin, Ireland (2004): n. pag. Web.

[20] Utgoff, P. E. (1989). Incremental induction of decision trees. Machine Learning, 4, 161-186

[21] Utgoff, P. E., Berkman, N. C., and Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. Machine Learning, 29, 5-44

[22] Wang, Haixun, Wei Fan, and Philip Yu. "Mining Concept-Drifting Data Streams Using Ensemble Classifiers." SIGKDD (2003): n. pag. Web.

[23] Yang, Hang, Simon Fong, and Yain-Whar Si. "Multi-objective Optimization for Incremental Decision Tree Learning." Data Warehousing and Knowledge Discovery Lecture Notes in Computer Science (2012): 217-28. Web.