Journal of Machine Learning Research 1 (2005) ?? Submitted 3/05; Published ??/??

Mining Massive Data Streams

Geoff Hulten GHULTEN@CS.WASHINGTON.EDU
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399, USA

Pedro Domingos PEDROD@CS.WASHINGTON.EDU
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, USA

Laurie Spencer LAURIES@INNOVATION-NEXT.COM
Innovation Next
1107 NE 45th St. #427
Seattle, WA 98105, USA

Editor: ??

Abstract

Many organizations today have more than very large databases; they have databases that grow without limit at a rate of several million records per day. Mining these continuous data streams brings unique opportunities, but also new challenges. In this paper we present a method that can semi-automatically enhance a wide class of existing learning algorithms so they can learn from such high-speed data streams in real time. The method works by sampling just enough data from the data stream to make each decision required by the learning process. The method is applicable to essentially any induction algorithm based on discrete search. After its application, the algorithm: learns from data streams in an incremental, any-time fashion; runs in time independent of the amount of data seen, while making decisions that are essentially identical to those that would be made from infinite data; uses a constant amount of RAM no matter how much data it sees; and adjusts its learned models in a fine-grained manner as the data-generating process changes over time. We evaluate our method by using it to produce two systems: the VFDT system for learning decision trees from massive data streams; and CVFDT, an enhanced version of VFDT that keeps the trees it learns up-to-date as the data-generating process changes over time. We evaluate these learners with extensive studies on synthetic data sets, and by mining the continuous stream of Web access data from the whole University of Washington main campus.

Keywords: Data streams, concept drift, data mining, decision trees, scalable learning algorithms, subsampling, Hoeffding bounds

1. Introduction

Machine learning systems are constrained by three main limited resources: time, memory and sample size. In traditional applications, sample size tends to be the dominant limitation: the computational resources for a massive search are available, but carrying out such a search over the small samples available (typically less than 10,000 examples) often leads to overfitting (e.g., Webb (1995), Quinlan and Cameron-Jones (1995), Domingos (1998), Jensen and Cohen (2000)). Thus overfitting avoidance becomes the main concern, and only a fraction of the available computational power is used (Dietterich, 1995). In contrast, in many (if not most) present-day data mining applications, the bottleneck is time and memory, not examples. The latter are typically in over-supply, in the sense that it is impossible with current learning systems to make use of all of them within the available computational resources. As a result, most of the available examples go unused, and underfitting may result: enough data to model very complex phenomena is available, but inappropriately simple models are produced because we are unable to take full advantage of the data. Thus the development of highly efficient algorithms becomes a priority.

Currently, the most efficient algorithms available (e.g., Shafer et al. (1996)) concentrate on making it possible to mine databases that do not fit in main memory by only requiring sequential scans of the disk. But even these algorithms have only been tested on up to a few million examples. In many applications this is less than a day's worth of data. For example, every day WalMart records on the order of 20 million sales transactions, Google handles on the order of 150 million searches, AT&T produces on the order of 275 million call records, large banks process millions of ATM and credit card transactions, scientific data collection (e.g. by earth sensing satellites or astronomical observatories) produces gigabytes of data, and popular Web sites log millions of hits. As the expansion of the Internet continues and ubiquitous computing becomes a reality, we can expect that such data volumes will become the rule rather than the exception. In these domains a few months worth of data can easily add up to billions of records, and the entire history of transactions or observations can be in the hundreds of billions. Current algorithms for mining complex models from data (e.g., decision trees, rule sets) cannot mine even a fraction of this data in useful time. In particular, mining a day's worth of data can easily take more than a day of CPU time, and as a result data accumulates faster than it can be mined, some data remains permanently unexplored, and the quantity of this unexplored data grows without bound as time progresses. Even simply preserving the examples for future use can be a problem when they need to be sent to tertiary storage, are easily lost, corrupted, or become unusable when the relevant contextual information is no longer available. For all of these reasons, when the source of examples is an open-ended data stream, the notion of mining a database of fixed size itself becomes questionable and determining how much data should be gathered before any mining can occur becomes a key challenge.

A further problem is that most statistical and machine learning algorithms make the assumption that training data is a random sample drawn from a stationary distribution. Unfortunately, most of the large databases and data streams available for mining today violate this assumption. They exist over months or years, and the underlying processes generating them change during this time, sometimes radically. For example, a new product or promotion, a hacker's attack, a holiday, changing weather conditions, changing economic conditions, or a poorly calibrated sensor can all lead to violations of this assumption. For classification systems, which attempt to learn a discrete function given examples of its inputs and outputs, this problem takes the form of changes in the target function over time, and is known as concept drift (Widmer and , eds.). Traditional systems learn incorrect models when they erroneously assume that the underlying concept is stationary if in fact it is drifting.

Ideally, we would like to have learning systems that operate continuously and indefinitely, incorporate examples as they arrive, never lose potentially valuable information, that work within strict resource constraints, and keep the models they learn up-to-date as concepts drift over time. In this paper we develop and evaluate a framework that is capable of semi-automatically scaling up learning algorithms from a broad class to work in these situations, and thus to successfully mine massive data streams. The framework works by limiting the quantity of data used at each step of the algorithm, while guaranteeing that the decisions made do not differ significantly from those that would be made given infinite data. It is applicable to essentially any learning algorithm based on discrete search, where, at each step, a number of candidate models or model components are considered, and the best one or ones are selected based on their performance on a sample from the domain of interest. The method is applicable to a range of search types including greedy, hill-climbing, beam, multiple-restart, lookahead, best-first, genetic, etc. It is applicable to common algorithms for decision tree and rule induction, instance-based learning, feature selection, model selection, parameter setting, probabilistic classification and clustering, probability estimation in discrete spaces, etc., as well as to their combinations.

In Section 2 we describe our scaling framework in detail, its properties, and some extensions that allow it to be applied in practical settings. This framework is the first of three main contributions of this paper: it meets a set of requirements that, to our knowledge, are not met by any previous machine learning or data mining framework. Section 3 describes VFDT, a decision tree induction algorithm capable of learning from high-speed data streams. VFDT was created by applying the ideas developed in our framework to a standard decision tree induction algorithm. Sections 4 and 5 describe empirical evaluations of VFDT on synthetic data (in Section 4) and on a Web caching application (in Section 5). These three sections combine to form the second main contribution of this paper: they show how our framework can be used to scale up an existing learning algorithm and demonstrate that the resulting algorithm outperforms existing ones in terms of features, runtime, and quality of the models it learns. Section 6 describes CVFDT, the third main contribution of this paper. CVFDT extends VFDT to time-changing domains by taking advantage of our method's ability to handle concept drift. We explore related work in Section 7, discuss future work in Section 8, and conclude in Section 9.

2. A General Method for Scaling Up Learning Algorithms

Consider the following simple problem. We are given two classifiers A and B, and an infinite data stream of iid (independent and identically distributed) examples. We wish to determine which of the two classifiers is more accurate on the data stream. If we want to be absolutely sure of making the correct decision, we have no choice but to apply the two classifiers to every example in the data stream, taking infinite time. If, however, we are willing to accommodate a probability δ of choosing the wrong classifier, we can generally make a decision in finite time, taking advantage of statistical results that give confidence intervals for the mean of a variable. One such result is known as *Hoeffding bounds* or *additive Chernoff bounds* (Hoeffding, 1963). Consider a real-valued random variable r whose range is R. Suppose we have made n independent observations of this variable, and computed their mean $\overline{r}$. One form of Hoeffding bound states that, with probability $1-\delta$, the true mean of the variable is at least $\overline{r}-\epsilon$, where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{1}$$

Let $\overline{r}$ be the difference in accuracy between the two classifiers on the first n examples in the data stream, and assume without loss of generality that it is positive. Then, if $\overline{r} > \epsilon$, the Hoeffding bound guarantees that with probability $1-\delta$ the classifier with higher accuracy on those n examples is also the more accurate one on the entire infinite data source. In other words, in order to make the correct decision with error probability δ, it is sufficient to observe enough examples to make ϵ smaller than $\overline{r}$.

The only case in which this procedure does not yield a decision in finite time occurs when the two classifiers have exactly the same accuracy. No number of examples will then suffice to find a winner. In this case we do not care which classifier wins because their performance on the metric we are interested in is the same. If we stipulate a minimum difference in accuracy τ below which we are indifferent as to which classifier is chosen, the procedure above is guaranteed to terminate after seeing at most $n = \lceil \frac{1}{2}(R/\tau)^2 \ln(1/\delta) \rceil$ examples. In other words, the time required to choose a classifier is constant, independent of the size of the data stream.

The Hoeffding bound has the very attractive property that it is independent of the probability distribution generating the observations. The price of this generality is that the bound is more conservative than distribution-dependent ones (i.e., it will take more observations to reach the same δ and ϵ). An alternative is to use a normal bound, which assumes $\overline{r}$ is normally distributed. (By the central limit theorem, this will always be approximately true after some number of samples.) In this case, ϵ is the value of $r-\overline{r}$ for which $\Phi((r-\overline{r})/\sigma_r) = 1-\delta$, where $\Phi()$ is the standard normal distribution function.[1] Either way, we can view a bound as a function $f(\delta, n, \mathbf{s})$ that returns the maximum ϵ by which the true mean of r is smaller than the sample mean $\overline{r}$, given a desired confidence $1-\delta$, sample size n, and any sufficient statistics $\mathbf{s}$ needed from the sample (for example, $\mathbf{s} = (\sum x_i, \sum s_i^2)$ when using the normal bound with unknown variance).

Suppose now that, instead of two classifiers, we wish to find the best one among b. Making the correct choice requires that each of the $b-1$ comparisons between the best classifier and all others have the correct outcome (i.e., that the classifier that is best on finite data also be best on infinite data). If the probability of error in each of these decisions is δ, then by the union bound the probability of error in any of them, and thus in the global decision, is at most $(b-1)\delta$. Thus, if we wish to choose the best classifier with error probability at most δ^*, it suffices to use $\delta = \delta^*/(b-1)$ in the bound function $f(\delta, n, \mathbf{s})$ for each comparison. Similarly, if we wish to find the best a classifiers among b with probability of error at most δ^*, $a(b-a)$ comparisons need to have the correct outcome, and to ensure that this is the case with probability of error at most δ, it suffices to require that the probability of error for each individual comparison be at most $\delta^*/[a(b-a)]$.

Suppose now that we wish to find the best classifier by a search process composed of d steps, at each step considering at most b candidates and choosing the best a. For the search

1. If the variance σ_r^2 is estimated from data, the Student t distribution is used instead.

process with finite data to output the same classifier as with infinite data with probability at least $1 - \delta^*$, it suffices to use $\delta = \delta^*/[da(b-a)]$ in each comparison. Thus, in each search step, we need to use enough examples n_i to make $\epsilon_i = f(n_i, \delta^*/[da(b-a)], \mathbf{s}) < r_i$, where r_i is the difference in accuracy between the a^{th} and $(a+1)^{th}$ best classifiers (on n_i examples, at the ith step). As an example, for a hill-climbing search of depth d and breadth b, the required δ would be δ^*/db. This result is independent of the process used to generate candidate classifiers at each search step (as long as this process does not itself access the data), and of the order in which search steps are performed. It is applicable to randomized search processes, if we assume that the outcomes of random decisions are the same in the finite- and infinite-data cases.[2]

In general, we do not know in advance how many examples will be required to make $\epsilon_i < r_i$ for all alternatives at step i. Instead, we can scan Δn examples from the data stream or data, and check whether the goal has been met. Let $f^{-1}(\epsilon, \delta, \mathbf{s})$ be the inverse of the bound function, yielding the number of samples n needed to reach the desired ϵ and δ. (For the Hoeffding bound, $n = \lceil \frac{1}{2}(R/\epsilon)^2 \ln(1/\delta) \rceil$.) Recall that τ is the threshold of indifference below which we do not care which classifier is more accurate. Then, in any given search step, at most $c = \lceil f^{-1}(\tau, \delta, \mathbf{s})/\Delta n \rceil$ goal checks will be made. Since a mistake could be made in any one of them (i.e., an incorrect winner could be chosen), we need to use enough examples n_i at each step i to make $\epsilon_i = f(n_i, \delta^*/[cda(b-a)], \mathbf{s}) < r_i$.

This procedure applies when the evaluation function is decomposable into an average (or sum) over training examples, as is the case for training set accuracy. It also applies for more complex evaluation functions, where each training example's contribution is a function of averages (or sums) of other properties of the training data. For instance, when the evaluation function is entropy the score depends on the example's class and the distribution of the classes of the other relevant examples. Computing an exact value for r_i for such evaluation functions requires an exact value for each contributing average (or sum), which requires access to all the training data, and thus is not possible when learning from an open-ended data stream. Instead, we use $f(.)$ to bound these contributing elements, use these to obtain a high confidence upper and lower bound on the evaluation of each classifier, and collect enough examples in each search step so that the lower bound on the evaluation of the a^{th} classifier is higher than the upper bound of the $a+1^{th}$. Let this difference be r_i^-, and let q be the maximum number of applications of $f(.)$ needed to compute the bound on the evaluation function. We need to use enough examples n_i at each step i to make $\epsilon_i = f(n_i, \delta^*/[cda(b-a)q], \mathbf{s}) < r_i^-$. (See Section 3.1 for an example of how to do this when the evaluation function is information gain.)

Notice that nothing in the treatment above requires that the models being compared be classifiers. The treatment is applicable to any inductive model — be it for classification, regression, probability estimation, clustering, ranking, etc. — and to comparisons between components of models as well as between whole models. This leads to the general method for scaling up learning algorithms shown in Tables 1 and 2.

The key properties of this method are summarized in the following proposition. Let t_{Gen} be the total time spent by L^* generating candidates, and let t_{Sel} be the total time spent

2. This excludes simulated annealing, because in this case the outcome probabilities depend on the observed differences in performance between candidates. Extending our treatment to this case is a matter for future work.

Table 1: Method for scaling up learning algorithms.

```
Given:
    An iid sample $T = \{X_1, X_2, \ldots, X_{|T|}\}$.
    A real-valued evaluation function $E(M, x, T)$ to be maximized, where $M$ is
        a model (or model component), $x$ is an example, $E(M, x, T, \delta)^-$ and
        $E(M, x, T, \delta)^+$ are respectively high-confidence lower and upper bounds
        on $E(M, x, T, \delta)$, and $q$ is the maximum number of applications of f
        needed to compute $E()^+$ and $E()^-$.
    A learning algorithm $L$ that finds a model by performing at most
        $d$ search steps, at each step considering at most $b$ candidates
        and selecting the $a$ with highest $\overline{E}(M, T) = \frac{1}{|T|} \sum_{x \in T} E(M, x, T)$.
    A desired maximum error probability $\delta^*$.
    A threshold of indifference $\tau$.
    A bound function $f(n, \delta, \mathbf{s})$.
    A block size $\Delta n$.
Modify $L$, yielding $L^*$, by at each step replacing the selection of
    the $a$ candidates with highest $\overline{E}(M, T)$ with a call to
    SelectCandidates($\mathcal{M}$), where $\mathcal{M}$ is the set of candidates at that step.
```

Table 2: The *SelectCandidates* subroutine of our method.

```
Procedure SelectCandidates($\mathcal{M}$)
Let $n = 0$.
Repeat
    If $n + \Delta n > |T|$ then let $n' = |T|$.
    Else let $n' = n + \Delta n$.
    For each $M \in \mathcal{M}$
        Let $\Sigma(M)^- = \sum_{i=1}^{n'} E(M, X_i, \{X_1 \ldots X_{n'}\}, \delta^*/[cda(b-a)q])^-$.
        Let $\Sigma(M)^+ = \sum_{i=1}^{n'} E(M, X_i, \{X_1 \ldots X_{n'}\}, \delta^*/[cda(b-a)q])^+$.
        Let $\overline{E}(M, T')^- = \Sigma(M)^-/n'$.
        Let $\overline{E}(M, T')^+ = \Sigma(M)^+/n'$.
    Let $\mathcal{M}' = \{M \in \mathcal{M} : M$ has one of the $a$ highest $\overline{E}(M, T')^-$ in $\mathcal{M}\}$.
    $n = n'$.
Until $[\forall (M_i \in \mathcal{M}', M_j \in \mathcal{M} - \mathcal{M}')$
        $f(n, \delta^*/[cda(b-a)q], \mathbf{s}) < \overline{E}(M_i, T')^- - \overline{E}(M_j, T')^+$
    or $f(n, \delta^*/[cda(b-a)q], \mathbf{s}) < \tau$ or $n = |T|]$.
Return $\mathcal{M}'$.
```

by it in calls to *SelectCandidates*($\mathcal{M}$), including data access. Let L^{∞} be L with $|T| = \infty$, and U be the first terminating condition of the "repeat" loop in *SelectCandidates*($\mathcal{M}$) (see Table 2).

Proposition 1 *If $t_{Sel} > t_{Gen}$ and $|T| > c\Delta n$, the time complexity of L^* is $O(db(a + c\delta n))$. With probability at least $1 - \delta^*$, L^* and L^{∞} choose the same candidates at every step for which U is satisfied. If U is satisfied at all steps, L^* and L^{∞} return the same model with probability at least $1 - \delta^*$.*

c is an upper bound on the number of times *SelectCandidates* will be called for any search step, because after that a tie will be called (and the search step will be ended). Each of these c calls incorporate Δn more examples into the sufficient statistics for each of the b alternatives being considered. The entire search takes d steps. A tie between candidates at some search step occurs if τ is reached or the data stream is exhausted before U is satisfied. With probability $1 - \delta^*$, all decisions made without ties are the same that would be made with infinite data. If there are no ties in any of the search steps, the model produced is, with probability $1 - \delta^*$, the same that would be produced with infinite data (as per the preceding discussion). L^* requires at least bd steps, and this best case occurs when each search decision is easy enough to be made after just Δn samples. More difficult decisions will require more samples, up to a maximum of $c\Delta n$. In other words, the running time of the modified algorithm is independent of the size of the data stream, as long as the latter is greater than $f^{-1}(\tau, \delta^*/[cda(b - a)q], \mathbf{s})$, and instead depends on the complexity of the decisions involved in the learning process.

SelectCandidates($\mathcal{M}$) is an anytime procedure in the sense that, at any point after processing the first Δn examples, it is ready to return the best a candidates according to the data scanned so far (in increments of Δn). If the learning algorithm is such that successive search steps progressively refine the model to be returned, L^* itself is an anytime procedure.

The method is equally applicable to databases (i.e., samples of fixed size that can be scanned multiple times) and data streams (i.e., samples that grow without limit and can only be scanned once). In the database case, we start a new scan whenever we reach the end of the database, and ensure that no search step uses the same example twice (we call a tie just before this would happen). In the data stream case, we simply continue scanning the stream after the winners for a step are chosen, using the new examples to choose the winners in the next step, and so on until the search terminates. When working with a data stream (and the rate of data arriving on the data stream is lower than the rate at which data can be incorporated by the algorithm) it can be advantageous to save old examples in a cache and reuse them for later decisions.

An alternative use of our method is to first choose a maximum error probability δ to be used at each step, and then report the global error probability achieved, $\delta^* = \delta \sum_{i=1}^{d} c_i a_i$ $(b_i - a_i)q_i$, where c_i is the number of goal checks performed in step i, b_i is the number of candidates considered at that step, a_i is the number selected, and q_i is the number of bounds needed for the evaluation function. Even if δ is computed from c, b, a, q and the desired δ^* as in Table 2, reporting the actual achieved δ^* is recommended, since it will often be much better than the original target.

Further time can be saved by dropping candidates from consideration as soon as it becomes clear (with high confidence) that they will not be selected by *SelectCandidates*. In particular, a candidate M_c is dropped when $\forall M_i \in \mathcal{M}' : \overline{E}(M_c, T')^+ + f(n, \delta^*/[cda(b-a)q], \mathbf{s}) < \overline{E}(M_i, T')^-$. Dropping a candidate allows the CPU time, RAM, and data accesses that it was utilizing to be used by other candidates, which allows learning to be carried out more efficiently.

2.1 Pruning

Our method supports several types of pruning: no-pruning, post-pruning, and pre-pruning. The simplest version is no-pruning, where the search steps are taken until the maximum search depth of d is reached. Recall that, since our method only makes refinements that are supported by high-confidence statistical tests, no-pruning need not lead to overfitting.

An alternate approach is to use post-pruning. In this mode some portion of training data is withheld from the training process and used to post-prune the model as needed (using standard techniques). This prune data can be kept in RAM if practical, or stored on disk, and its size can be limited so that it fits within available resources. Note that in an online setting post-pruning can be performed on a copy of the model, so that the original can be further refined while the pruned copy is used for performance tasks.

The final pruning mode supported is pre-pruning. This mode uses a pre-prune parameter τ' and no model is returned by *SelectCandidates* unless it improves over the current model by at least this amount. In particular, let M_0 be the current model. Our method stops early and returns M_0 if $\forall M_i \in \mathcal{M}' : \overline{E}(M_i, T')^- - \overline{E}(M_0, T')^+ < f(n, \delta^*/[cda(b-a)q], \mathbf{s})$ and $f(n, \delta^*/[cda(b-a)q], \mathbf{s}) < \tau'$.

2.2 Working Within Memory Constraints

Notice that for many evaluation functions (including accuracy, entropy, Gini, etc.) *SelectCandidates* does not need direct access to the training data, but only to *sufficient statistics* gathered from the data. In these cases, our method requires only RAM to store candidate models and their associated sufficient statistics. Each example need only be in RAM for the time required to update the appropriate sufficient statistics, and it can then be purged. When learning large, complex models, however, it is often the case that the structure, parameters, and sufficient statistics used by all the candidates at a given step exceed the available memory. This can lead to severe slowdowns as memory pages are repeatedly swapped to and from disk. Our method can be easily adapted to avoid this, as long as $m > a$, where m is the maximum number of candidates that fits within the available RAM. We first form a set $\mathcal{M}_1$ composed of any m elements of $\mathcal{M}$, and run *SelectCandidates*($\mathcal{M}_1$). We then add another $m - a$ candidates to the a selected, yielding $\mathcal{M}_2$, and run *SelectCandidates*($\mathcal{M}_2$). We continue in this way until $\mathcal{M}$ is exhausted, returning the a candidates selected in the last iteration as the overall winners. As long as all calls to *SelectCandidates*() start scanning S at the same example, ties are broken consistently, and data is i.i.d, these winners are the same that would be obtained by the single call *SelectCandidates*($\mathcal{M}$) with high confidence. If k iterations are carried out, this modification increases the running time of *SelectCandidates*($\mathcal{M}$) by a factor of at most k, with $k < b$. Notice that the "new" candidates in each $\mathcal{M}_i$ do not need to be generated until *SelectCandidates*($\mathcal{M}_i$) is to be run, and thus they never need to be swapped to disk.

2.3 Parallel Searches

Many learning algorithms are intrinsically parallel. For example, when performing decision tree induction, the searches for splits at the leaves are completely independent of each other. In such cases, a separate copy of our method is created for each independent component of the model. When new data arrives it is filtered to the searches where it has an effect, added to the samples there, and *SelectCandidates* is called for any search that has accumulated Δn examples since the last call. The procedure used to identify which searches are affected by training examples will vary between learning algorithms. For instance, when learning a decision tree each example affects the search for the model at exactly one leaf, and the partially induced tree can be used to quickly identify which one. Note that the number of parallel searches may change during the course of a learning run (e.g. whenever new leaves are added to a decision tree).

There are two main benefits of this parallel decomposition. The first advantage is that it avoids a great deal of redundant work by allowing decisions that are independent to be made independently. Consider the alternative, learning a decision tree with a single monolithic search. Each step of this monolithic search would involve comparing all the possible splits at all of the leaves of the decision tree, and thus require much more data to achieve the same confidence. Once a search step was made, all of these alternatives would need to be reevaluated on a new sample of data for the next search step.[3] The second advantage is that it allows us to deactivate portions of the learning algorithm in a fine grained manner when CPU or RAM resources become short. In particular, whenever computing resources are exhausted we order the active parallel searches by an estimate of the maximum improvement that can be achieved by continuing them. We then temporarily deactivate those with lowest rankings. For example, in decision tree learning this can be the error rate at the leaf times the fraction of the data stream that filters to it.

2.4 Time-Changing Concepts

The processes that generate massive data sets and open-ended data streams often span months or years, during which the data-generating distribution can change significantly, violating the iid assumption made by most learning algorithms. Our method keeps the model it is learning in sync with such changing concepts by continuously monitoring the quality of old search decisions with respect to a sliding window of data from the data stream, and updating them in a fine-grained way when it detects that the distribution of data is changing. In particular, we maintain sufficient statistics throughout time for every candidate M considered at every search step. After the first w examples, where w is the window width, we subtract the oldest example from these statistics whenever a new one is added. After every Δn new examples, we determine again the best a candidates at every previous search decision point. If one of them is better than an old winner by δ^* then one of two things has happened. Either the original decision was incorrect (which will happen a fraction δ of the

3. The VFBN2 algorithm for learning the structure of Bayesian networks from massive data streams (Hulten and Domingos, 2002) was developed using this insight.

time) or concept drift has occurred. In either case, we begin an alternate search starting from the new winners, while continuing to pursue the original search. Periodically we use a number of new examples as a validation set to compare the performance of the models produced by the new and old searches. We prune an old search (and replace it with the new one) when the new model is on average better than the old one, and we prune the new search if after a maximum number of validations its models have failed to become more accurate on average than the old ones. If more than a maximum number of new searches is in progress, we prune the lowest-performing ones. This approach to handling time-changing data is used to create the CVFDT decision tree induction algorithm, and will be discussed in more detail in Section 6.

2.5 Active Learning

Active learning is a powerful type of subsampling where the learner actively selects the examples that would cause the most progress e.g., Cohn et al. (1994). Our method has a natural extension to this case when examples are relevant to some, but not all, of the parallel searches. When choosing the next Δn examples we never select ones that are only relevant to searches that have been deactivated because of RAM constraints. Further, where possible, our method can choose the next Δn examples so that searches where more progress is possible (according to some heuristic measure) receive more examples.

3. The VFDT System

In this section we apply out method for scaling to a decision tree induction algorithm (Breiman et al., 1984, Quinlan, 1993) and produce VFDT (Very Fast Decision Tree learner), an incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams within strict CPU and RAM constraints. In this section we assume the examples are generated by a stationary stochastic process (i.e., their distribution does not change over time).[4] In Section 6 we extend the VFDT algorithm to work with time-changing concepts.

The classification problem is generally defined as follows. A set of N training examples of the form $(\mathbf{x}, y)$ is given, where y is a discrete class label and $\mathbf{x}$ is a vector of d attributes, each of which may be symbolic or numeric. The goal is to produce from these examples a model $y = f(\mathbf{x})$ that will predict the classes y of future examples $\mathbf{x}$ with high accuracy. For example, $\mathbf{x}$ could be a description of a client's recent purchases, and y the decision to send that customer a catalog or not; or $\mathbf{x}$ could be a record of a cellular-telephone call, and y the decision whether it is fraudulent or not.

Decision trees address this problem as follows. Each internal node in the tree contains a test on an attribute, each branch from a node corresponds to a possible outcome of the test, and each leaf contains a class prediction. The label $y = DT(\mathbf{x})$ for an example $\mathbf{x}$ is obtained by passing the example down from the root to a leaf, testing the appropriate attribute at each node and following the branch corresponding to the attribute's value in the example.

4. If the examples are being read from disk, we assume that they are in random order. If this is not the case, they should be randomized, for example by creating a random index and sorting on it.

A decision tree is learned by recursively replacing leaves by test nodes, starting at the root. The attribute to test at a node is chosen by comparing all the available attributes and choosing the best one according to some heuristic measure. Classic decision tree learners like ID3, C4.5 and CART assume that all training examples can be stored simultaneously in main memory, and are thus severely limited in the number of examples they can learn from. Disk-based decision tree learners like SLIQ (Mehta et al., 1996) and SPRINT (Shafer et al., 1996) assume the examples are stored on disk, and learn by repeatedly reading them in sequentially (effectively once per level in the tree). While this greatly increases the size of usable training sets, it can become prohibitively expensive when learning complex trees (i.e., trees with many levels), and fails when data sets are too large to fit in the available disk space or when data is arriving on an open-ended stream.

In this section we use our method to design a decision tree learner for extremely large (potentially infinite) data sets. We achieve this by noting with Catlett (1991) and others (Musick et al., 1993, Gehrke et al., 1999) that, in order to find the best attribute to test at a given node, it may be sufficient to consider only a small subset of the training examples that are relevant to that node. Given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate attributes there, and so on recursively. This learner requires that each example be read at most once, and it uses only a small constant time to process each example. This makes it possible to directly mine online data sources (i.e., without ever storing the examples), and to build potentially very complex trees with acceptable computational cost.

3.1 High Confidence Upper and Lower Bounds for Information Gain

In order to use our method, we need high-confidence upper and lower bounds on the evaluation function being used. VFDT uses information gain (Quinlan, 1993):

$$G(A,T) = H(T) - \sum_{v \in A} P(T,A,v) H(Sel(T,A,v)) \tag{2}$$

where A is an attribute, T is a set of training examples, $P(T,A,v)$ is the fraction of examples in T that have value v for attribute A, $Sel(T,A,v)$ selects from T all examples with value v for attribute A, and $H(.)$ is the entropy function. The first step to bounding information gain is obtaining a bound for entropy. The entropy of a distribution over classes is

$$H(T) = - \sum_{c \in Classes} P(c|T) \log P(c|T) \tag{3}$$

where $P(c|T)$ is the probability that an example in T has class label c. In our setting, each of these probabilities will be estimated from a sample of data and so we need to bound their values. For this presentation we will use the Hoeffding bound, but we have also used normal bounds in our implementation. One form of the Hoeffding bound (Hoeffding, 1963) says that

$$\hat{P}(c|T) - \sqrt{\frac{\ln(\frac{2}{\delta})}{2n}} \leq P(c|T) \leq \hat{P}(c|T) + \sqrt{\frac{\ln(\frac{2}{\delta})}{2n}} \tag{4}$$

where n is the size of the sample. This gives us a high confidence upper and lower bound for each probability; let these be $P^+_{c_1}, \ldots, P^+_{c_m}$ and $P^-_{c_1}, \ldots, P^-_{c_m}$ respectively. We achieve an upper and lower bound on entropy by maximizing and minimizing the $p \log p$ of each class separately and then summing them. Taking the derivative of $p \log p$ and setting equal to zero we find that the maximum occurs at $1/e$. Also minima occur at 0 and 1. Also notice that the function is concave. Therefore, a high-confidence maximum is obtained by taking for each P_{c_i} the value in the closed interval $[P^+_{c_i}, P^-_{c_i}]$ that is closest to $1/e$. Let $P^{max}(c_i)$ be the value that maximizes this for c_i. A high-confidence minimum is obtained by taking for each P_{c_i} whichever of $\min(P^+_{c_i}, 1)$ and $\max(P^-_{c_i}, 0)$ has a lower value for $p \log p$. Let $P^{min}(c_i)$ be the value that minimizes this for c_i. Let the upper and lower bounds on the entropy be H^+ and H^- respectively

$$\begin{aligned} H(T)^+ &= -\sum_{c \in Classes} P^{max}(c|T) \log P^{max}(c|T) \\ H(T)^- &= -\sum_{c \in Classes} P^{min}(c|T) \log P^{min}(c|T) \end{aligned} \tag{5}$$

We improve on this bound by taking advantage of the constraint that $\sum_{c \in Classes} P(c|T) = 1$. Using this constraint, we know that the entropy function takes its maximum value when each of the P_{c_i}'s are $1/m$, and that this maximum value is $\log m$ (where m is the number of values of the class attribute). If the upper bound we obtain is higher than this value, we return $\log m$ instead.

We can now use these to bound information gain. Notice that the $H(T)$ term in Equation 2 is the same for every attribute. We drop it from further consideration because it will not affect the comparison between attributes. Thus, the final bounds are

$$\begin{aligned} G(A,T)^+ &= \sum_{v \in A} (P(T,A,v) + \sqrt{\ln(1/\delta)/2N}) H(Sel(T,A,v))^+ \\ G(A,T)^- &= \sum_{v \in A} (P(T,A,v) - \sqrt{\ln(1/\delta)/2N}) H(Sel(T,A,v))^- \end{aligned} \tag{6}$$

and q, the number of statistical bounds that must hold for these to hold (recall that this is needed by our method to correct δ), is $2v(c+1)$: for each value both $P(T,A,v)$ and the class probability estimates needed in the entropy calculation must be bounded.

3.2 The VFDT Algorithm

We now present *VFDT*, shown in pseudo-code in Table 3. The counts n_{ijk} are the sufficient statistics needed to compute most heuristic measures, including information gain. The pseudo-code shown is only for discrete attributes; its extension to numeric ones is described in Section 3.5. The sequence of examples S may be infinite, in which case the procedure never terminates, and at any point in time a parallel procedure can use the current tree DT to make class predictions.

As long as VFDT processes examples faster than they arrive, which will be the case in all but the most demanding applications, the sole obstacle to learning arbitrarily complex

Table 3: The VFDT algorithm.

```
Inputs:
          $T = \{X_1, X_2, \ldots, X_\infty\}$ a stream of iid samples,
          $\mathbf{X}$          a set of discrete attributes,
          $G(.)^+$ and $G(.)^-$     upper and lower bounds on a split evaluation
                    function $G(.)$ (see Equation 6),
          $\delta$          one minus the desired probability of choosing the correct attribute at
                    any given node,
          $\tau$          a user specified tie threshold,
          $f$          a bound function (we use the Hoeffding bound; see Equation 1).
Output:   $DT$, a decision tree.

Procedure VFDT ($T, \mathbf{X}, G^+, G^-, \delta, \tau, f$)
/* Initialize */
Let $DT$ be a tree with a single leaf $l_1$ (the root).
For each class $y_k$
    For each value $x_{ij}$ of each attribute $X_i \in \mathbf{X}$
        /* The sufficient statistics needed to calculate $G$ */
        Let $n_{ijk}(l_1) = 0$.
/* Process the examples */
For each example $(\mathbf{x}, y)$ in $T$
    Sort $(\mathbf{x}, y)$ into a leaf $l$ using $DT$.
    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 $\overline{G}_l^+(X_i)$ and $\overline{G}_l^-(X_i)$ for each attribute $X_i \in \mathbf{X}_l$
            using the counts $n_{ijk}(l)$.
        Let $X_a$ be the attribute with highest $\overline{G}_l^-$.
        Let $X_b$ be the attribute other than $X_a$ with highest $\overline{G}_l^+$.
        Compute $\epsilon$ using $f(.)$.
        If $\overline{G}_l^-(X_a) - \overline{G}_l^+(X_b) > \epsilon$ or $\epsilon < \tau$, 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 $\mathbf{X_m} = \mathbf{X} - \{X_a\}$.
                For each class $y_k$ and each value $x_{ij}$ of each attribute $X_i \in \mathbf{X_m}$
                    Let $n_{ijk}(l_m) = 0$.
Return $DT$.
```

models will be the finite RAM available. VFDT's memory use is dominated by the memory required to keep counts for all growing leaves. If d is the number of attributes, v is the maximum number of values per attribute, and c is the number of classes, VFDT requires $O(dvc)$ memory to store the necessary counts at each leaf. If l is the number of leaves in the tree, the total memory required is $O(ldvc)$. This is independent of the number of examples seen, if the size of the tree depends only on the "true" concept and is independent of the size of the training set. Notice that, although this is a common assumption in the analysis of decision tree and related algorithms, it often fails in practice. If VFDT ever exceeds the RAM available it uses our method's RAM optimizations (See Section 2.2) and temporarily deactivates the searches at the least promising leaves to make room for new ones. In particular, if p_l is the probability that an arbitrary example will fall into leaf l, and e_l is the observed error rate at that leaf on the training data that reaches it, then $p_l e_l$ is an estimate of the maximum error reduction achievable by refining the leaf. $p_l e_l$ for a new leaf is estimated using the counts at the parent for the corresponding attribute value. The least promising leaves are considered to be the ones with the lowest values of $p_l e_l$. When a leaf is deactivated the memory it was using is freed, except for two numbers required to keep track of $p_l e_l$. A leaf can then be reactivated if it becomes more promising than a currently active leaf. This is accomplished by scanning through all the active and inactive leaves at regular intervals, and replacing the least promising active leaves with the inactive ones that dominate them.

In order to make the best use of all the available memory in the early part of a run (before RAM is filled by sufficient statistics), VFDT caches training examples in RAM. This is similar to what batch learners do, and it allows VFDT to reuse training examples for decisions on several levels of the induced tree. As soon as available memory is exhausted VFDT disables caching for leaves with the lowest $p_l e_l$ values until memory usage is under threshold. VFDT disables all example caches before it begins deactivating leaves.

The full VFDT system takes advantage of all of the other refinements to our method described in Section 2, including: stopping early on attributes that are clear losers; treating the search at each leaf as a parallel search; working on both databases and data streams; supporting pre-pruning, post-pruning and no-pruning; and active learning (tracking time-changing concepts will be discussed in more detail when we introduce CVFDT in Section 6).

3.3 Quality Guarantees

As Equation 1 implies, a key property of the VFDT algorithm is that it is possible to guarantee under realistic assumptions that the trees produced by the core of the algorithm as presented in Table 3 are asymptotically arbitrarily close to the ones produced by a batch learner (i.e., a learner that uses all the examples to choose a test at each node). In other words, the streaming nature of the VFDT algorithm does not significantly affect the quality of the trees it produces. The bound from our general method (see Equation 1) applies directly, but in this section we take advantage of the typical structure of decision trees to develop a stronger bound. We first define the notion of *disagreement* between two decision trees. Let $P(\mathbf{x})$ be the probability that the attribute vector (loosely, example) $\mathbf{x}$ will be observed, and let $I(.)$ be the indicator function, which returns 1 if its argument is true and 0 otherwise.

Definition 1 *The* extensional disagreement Δ_e *between two decision trees* DT_1 *and* DT_2 *is the probability that they will produce different class predictions for an example:*

$$\Delta_e(DT_1, DT_2) = \sum_{\mathbf{x}} P(\mathbf{x}) I[DT_1(\mathbf{x}) \neq DT_2(\mathbf{x})]$$

Consider that two internal nodes are different if they contain different tests, two leaves are different if they contain different class predictions, and an internal node is different from a leaf. Consider also that two paths through trees are different if they differ in length or in at least one node.

Definition 2 *The* intensional disagreement Δ_i *between two decision trees* DT_1 *and* DT_2 *is the probability that the path of an example through* DT_1 *will differ from its path through* DT_2*:*

$$\Delta_i(DT_1, DT_2) = \sum_{\mathbf{x}} P(\mathbf{x}) I[\text{Path}_1(\mathbf{x}) \neq \text{Path}_2(\mathbf{x})]$$

where $\text{Path}_i(\mathbf{x})$ *is the path of example* $\mathbf{x}$ *through tree* DT_i*.*

Two decision trees agree intensionally on an example iff they are indistinguishable for that example: the example is passed down exactly the same sequence of nodes, and receives an identical class prediction. Intensional disagreement is a stronger notion than extensional disagreement, in the sense that $\forall_{DT_1,DT_2}\ \Delta_i(DT_1, DT_2) \geq \Delta_e(DT_1, DT_2)$.

Let p_l be the probability that an example that reaches level l in a decision tree falls into a leaf at that level. To simplify, we will assume that this probability is constant, i.e., $\forall_l\ p_l = p$, where p will be termed the *leaf probability.* This is a realistic assumption, in the sense that it is typically approximately true for the decision trees that are generated in practice. Let $VFDT_\delta$ be the tree produced by the VFDT algorithm with desired probability δ given an infinite sequence of examples S, and DT_* be the asymptotic batch decision tree induced by choosing at each node the attribute with true greatest G (i.e., by using infinite examples at each node). Let $E[\Delta_i(VFDT_\delta, DT_*)]$ be the expected value of $\Delta_i(VFDT_\delta, DT_*)$, taken over all possible infinite training sequences. We can then state the following result.

Theorem 1 *If* $VFDT_\delta$ *is the tree produced by the VFDT algorithm with desired probability* δ *given infinite examples (Table 3),* DT_* *is the asymptotic batch tree, and* p *is the leaf probability, then* $E[\Delta_i(VFDT_\delta, DT_*)] \leq \delta/p$*.*

Proof. For brevity, we will refer to intensional disagreement simply as disagreement. Consider an example $\mathbf{x}$ that falls into a leaf at level l_v in $VFDT_\delta$, and into a leaf at level l_d in DT_*. Let $l = \min\{l_h, l_d\}$. Let $\text{Path}_V(\mathbf{x}) = (N_1^V(\mathbf{x}), N_2^V(\mathbf{x}), \ldots, N_l^V(\mathbf{x}))$ be $\mathbf{x}$'s path through $VFDT_\delta$ up to level l, where $N_i^V(\mathbf{x})$ is the node that $\mathbf{x}$ goes through at level i in $VFDT_\delta$, and similarly for $\text{Path}_D(\mathbf{x})$, $\mathbf{x}$'s path through DT_*. If $l = l_h$ then $N_l^V(\mathbf{x})$ is a leaf with a class prediction, and similarly for $N_l^D(\mathbf{x})$ if $l = l_d$. Let I_i represent the proposition "$\text{Path}_V(\mathbf{x}) = \text{Path}_D(\mathbf{x})$ up to and including level i," with $I_0 = \text{True}$. Notice that $P(l_v \neq l_d)$ is included in $P(N_l^H(\mathbf{x}) \neq N_l^D(\mathbf{x}) | I_{l-1})$, because if the two paths have different lengths then one tree must have a leaf where the other has an internal node. Then, omitting the dependency of the nodes on $\mathbf{x}$ for brevity,

$$\begin{aligned}
&P(\mathrm{Path}_V(\mathbf{x}) \neq \mathrm{Path}_D(\mathbf{x})) \\
&\quad= P(N_1^V \neq N_1^D \vee N_2^V \neq N_2^D \vee \ldots \vee N_l^V \neq N_l^D) \\
&\quad= P(N_1^V \neq N_1^D | I_0) + P(N_2^V \neq N_2^D | I_1) + \ldots + P(N_l^V \neq N_l^D | I_{l-1}) \\
&\quad= \sum_{i=1}^{l} P(N_i^V \neq N_i^D | I_{i-1}) \;\leq\; \sum_{i=1}^{l} \delta \;=\; \delta l
\end{aligned} \tag{7}$$

Let $VFDT_\delta(S)$ be the VFDT tree generated from training sequence S. Then $E[\Delta_i(DT_\delta, DT_*)]$ is the average over all infinite training sequences S of the probability that an example's path through $VFDT_\delta(S)$ will differ from its path through DT_*:

$$\begin{aligned}
&E[\Delta_i(VFDT_\delta, DT_*)] \\
&\quad= \sum_S P(S) \sum_{\mathbf{x}} P(\mathbf{x})\, I[\mathrm{Path}_V(\mathbf{x}) \neq \mathrm{Path}_D(\mathbf{x})] \\
&\quad= \sum_{\mathbf{x}} P(\mathbf{x})\, P(\mathrm{Path}_V(\mathbf{x}) \neq \mathrm{Path}_D(\mathbf{x})) \\
&\quad= \sum_{i=1}^{\infty} \sum_{\mathbf{x} \in L_i} P(\mathbf{x})\, P(\mathrm{Path}_V(\mathbf{x}) \neq \mathrm{Path}_D(\mathbf{x}))
\end{aligned} \tag{8}$$

where L_i is the set of examples that fall into a leaf of DT_* at level i. According to Equation 7, the probability that an example's path through $VFDT_\delta(S)$ will differ from its path through DT_*, given that the latter is of length i, is at most δi (since $i \geq l$). Thus

$$E[\Delta_i(VFDT_\delta, DT_*)] \leq \sum_{i=1}^{\infty} \sum_{\mathbf{x} \in L_i} P(\mathbf{x})(\delta i) = \sum_{i=1}^{\infty} (\delta i) \sum_{\mathbf{x} \in L_i} P(\mathbf{x}) \tag{9}$$

The sum $\sum_{\mathbf{x} \in L_i} P(\mathbf{x})$ is the probability that an example $\mathbf{x}$ will fall into a leaf of DT_* at level i, and is equal to $(1-p)^{i-1}p$, where p is the leaf probability. Therefore

$$\begin{aligned}
&E[\Delta_i(VFDT_\delta, DT_*)] \\
&\quad\leq \sum_{i=1}^{\infty} (\delta i)(1-p)^{i-1} p \;=\; \delta p \sum_{i=1}^{\infty} i(1-p)^{i-1} \\
&\quad= \delta p \left[\sum_{i=1}^{\infty} (1-p)^{i-1} + \sum_{i=2}^{\infty} (1-p)^{i-1} + \cdots + \sum_{i=k}^{\infty} (1-p)^{i-1} + \cdots \right] \\
&\quad= \delta p \left[\frac{1}{p} + \frac{1-p}{p} + \cdots + \frac{(1-p)^{k-1}}{p} + \cdots \right] \\
&\quad= \delta \left[1 + (1-p) + \cdots + (1-p)^{k-1} + \cdots \right] = \delta \sum_{i=0}^{\infty} (1-p)^i \;=\; \frac{\delta}{p}
\end{aligned} \tag{10}$$

This completes the demonstration of Theorem 1.

An immediate corollary of Theorem 1 is that the expected extensional disagreement between $VFDT_\delta$ and DT_* is also asymptotically at most δ/p (although in this case the bound is much looser). Another corollary is that there exists a subtree of the asymptotic batch tree such that the expected disagreement between it and the tree learned by VFDT on finite data is at most δ/p. In other words, if δ/p is small then VFDT's tree learned on finite data is very similar to a subtree of the asymptotic batch tree. A useful application of Theorem 1 is that, instead of δ, users can now specify as input to the VFDT algorithm the maximum expected disagreement they are willing to accept, given enough examples for the tree to settle. The latter is much more meaningful, and can be intuitively specified without understanding the workings of the algorithm or the Hoeffding bound. The algorithm will also need an estimate of p, which can easily be obtained (for example) by running a conventional decision tree learner on a manageable subset of the data.

3.4 Pruning

VFDT supports all three pruning options of our method no-pruning, pre-pruning, and post-pruning. When using the no-pruning option VFDT refines the tree it is learning indefinitely. When using pre-pruning VFDT takes an additional pre-prune parameter, τ', and stops growing any leaf where the difference between the best split candidate all others is less than $f(.)$ and $f(.) < \tau'$ (see Section 2.1). If there is ever a point at which all of the leaves in the tree are pre-pruned, the VFDT procedure terminates and returns the tree. Finally, when using post-pruning, VFDT reserves some portion of the stream T to use as pruning data. VFDT takes two post-pruning parameters: the probability that each example will be reserved for pruning, and the maximum number of pruning examples to maintain. (Note that pruning data can be kept in RAM if resources are available, or stored in disk if needed.) Then, when a parallel procedure needs to use the decision tree for performance tasks, VFDT creates a copy of the tree it is learning, prunes the copy with the pruning data and a standard pruning algorithm (we use reduced error pruning (**?**) in our evaluations), passes the pruned copy to the parallel process, and continues using additional data to further refine the original tree.

3.5 Numeric Attributes

Decision trees support numeric attributes by allowing internal nodes to contain tests of the form "$(A_i < t)$?" – the value of attribute i is less than threshold t. Traditional decision tree induction algorithms learn trees with such tests in the following manner. For each numeric attribute, they construct a set of candidate tests by sorting the values of that attribute in the training set and using a threshold midway between each adjacent pair of values that come from training examples with different class labels. These candidate tests are evaluated along with the discrete ones by modifying $P(T, A, v)$ and $Sel(T, A, v)$ from Equation 2 to count and select examples that pass and fail the threshold test instead of examples with the target value v.

There are several reasons why this standard method is not appropriate when learning from data streams. The most serious of these is that it requires that the entire training set be available ahead of time so that split thresholds can be determined. VFDT uses sampling and considers only a subset of all possible thresholds as candidate split points. (CLOUDS

(Alsabti et al., 1998), and BOAT (Gehrke et al., 1999) also use sampling, although these other systems were designed for fixed-sized data sets and are not appropriate for data streams.) In particular, whenever VFDT starts a new leaf, it collects up to M distinct values for each continuous attribute from the first examples that arrive at it. These are maintained in sorted order as they arrive, and a candidate test threshold is maintained halfway between adjacent values with different classes, as in the traditional method. Once VFDT has M values for an attribute, it stops adding new candidate thresholds and uses additional data only to evaluate the existing ones.

Every leaf uses a different value of M, based on its level in the tree and the amount of RAM available when it is started. For example, M can be very large when choosing the split for the root of the tree, but must be very small once there is a large partially-induced tree, and many leaves are competing for limited memory resources. Notice that even when M is very large (and especially when it is small) VFDT may miss the locally optimal split point. This is not a serious problem in our setting for two reasons. First, if data is an iid sample VFDT should end up with a value near (or an empirical gain close to) the correct one simply by chance. And second, VFDT will be learning very large trees from massive data streams and can correct early mistakes later in the learning process by adding additional splits to the tree. An empirical evaluation of these claims, and of the rest of the VFDT system, is described in the next two sections.

4. Empirical Evaluation of VFDT on Synthetic Data

A system like VFDT is only useful if it is able to learn more accurate trees than a conventional system, given similar computational resources. In particular, it should be able to use the examples that are beyond a conventional system's ability to process to learn better models. In this section we test this empirically by comparing VFDT with C4.5 release 8 (Quinlan, 1993), the incremental Decision tree learner ITI (Utgoff, 1994), and our implementation of SPRINT (Shafer et al., 1996) on a series of synthetic data streams[5]. Using synthetic data streams allows us to freely vary the relevant parameters of the learning process and more fully explore the capabilities of our system. In Section 5 we describe an evaluation on a real-world data set.

We first describe the data streams used for our experiments. They were created by randomly generating decision trees and then using these trees to assign classes to randomly generated examples. We produced the random decision trees as follows. Starting from a tree with a single leaf node (the root) we repeatedly replaced leaf nodes with nodes that tested a randomly selected attribute which had not yet been tested on the path from the root of the tree to that selected leaf. After the first three levels of the tree each selected leaf had a probability of f of being pre-pruned instead of replaced by a split (and thus of remaining a leaf in the final tree). Additionally, any branch that reached a depth of 18 was pruned at that depth. Whenever a leaf was pruned we randomly (with uniform probability) assigned a class label to it. A tree was complete as soon as all of its leaves

5. We also wanted to compare with C5.0, but license issues prevented it from running on the machines we used for our study. However, we did perform some comparisons between C4.5 and C5.0 on our synthetic data streams, and found their accuracy results to be practically indistinguishable (although C5.0 did learn smaller trees).

were pruned. We then generated a stream of 50 million training examples for each tree by sampling uniformly from the instance space, assigning classes to the examples according to the corresponding synthetic tree, and adding class and attribute noise by flipping each with probability n. We also generated 50,000 separate test examples for each concept using the same procedure. We used nine parameter settings to generate trees: every combination of f in $\{12\%, 17\%, 23\%\}$ (these values were selected so that the average concept tree would have about 100,000 nodes) and n in $\{5\%, 10\%, 15\%\}$. We also used six different random seeds for each of these settings to produce a total of 54 data streams. Every domain we experimented with had two classes and 100 binary attributes. We ran our experiments on a cluster of five Pentium III/1GH machines with 1GB of RAM, all running Linux. We reserved nine of the data streams, one from each of the parameter settings above, and used them to tune the algorithms' parameters.

We now briefly describe the learning algorithms used in our experiments:

- **C4.5** (Quinlan, 1986) is a well known and widely used batch decision tree learner for data sets that fit in RAM. Our hypothesis is that VFDT's performance is similar to C4.5's when the entire data set can fit in RAM, and that VFDT is able to take advantage of larger data sets to learn more accurate models. We used the reserved data streams to tune C4.5's '-c' (prune confidence level) parameter, and found 5% to achieve the best results.

- **ITI** (Utgoff, 1994) is an incremental decision tree learner for data sets that fit in RAM. That is, it examines training examples one at a time, adding them to it its tree, and updating any splits that changed based on the information contained in the new example. ITI differs from VFDT in two fundamental ways. First, ITI stores all training examples in the tree it is learning, while VFDT only stores sufficient statistics gathered from the examples (although VFDT is able to store examples when extra RAM is available). Second, ITI maintains the same tree that a batch learner would have learned on the data ITI has seen at any point during its run. This requires it to constantly restructure its tree to fix mistakes (decisions that looked good on the sample it had, but were not correct with respect to the process generating data). VFDT, on the other hand, only makes decisions that are supported by high-confidence statistical tests, and thus does not need to spend time restructuring its trees. Our hypothesis is that VFDT is able to incorporate data much more quickly and thus scale to much larger data sets. We tuned ITI's '-P' parameter, which controls the complexity of the trees ITI learns. This parameter specifies a minimum number of examples of the second most common class that must be present in every leaf. We found 50 to give the best results.

- **SPRINT** (Shafer et al., 1996) is an algorithm designed to learn decision trees from data sets that can fit on disk, but are too large to load into RAM. It does this by performing sequential scans over the data on disk, learning one level of a decision tree for each scan. For example, to learn a tree with 15 levels, SPRINT would read (and write) all training data from disk 15 times. This sounds expensive, but SPRINT is considerably faster than conventional learners (such as C4.5 and ITI) when data does not fit in RAM because it reorders data on disk as needed to maintain locality of access (while the conventional learners rely on OS supported virtual memory, which is not informed about their intentions, and thus is very wasteful with disk accesses). Our hypothesis is that VFDT's ability to learn from a data stream without storing anything to disk allows it to learn from very large data streams much faster than SPRINT, while producing models of similar quality.

We used our own implementation of SPRINT for these experiments. Our implementation shares code with our VFDT implementation (memory management, decision tree ADT, some data access, etc.). Additionally, SPRINT pruned its trees using the same code as VFDT's post-prune option, and both algorithms used the same pruning criteria: reserving 5% of training data for pruning. Once pruning was fixed, our implementation of SPRINT had no remaining tunable parameters.

- **VFDT** was implemented as described in the previous section. We also tried using two different bound functions. For the first we used $G()^+$ and $G()^-$ as described in Equation 6. For the second we used $G()^+ = G()^- = G()$. We found the models built using the second bound contained more nodes (on average by a factor of 4) and achieved better accuracy on testing data. This suggests that the bound from Equation 6 is very conservative. We used the second bound in all further experiments.

  We tried all three forms of pruning supported by our method: no-pruning, pre-pruning (with a pre-prune threshold of 0.005), and post-pruning (reserving 5% of the data up to a maximum of 2 million examples for pruning). We used a Δn (the number examples to accumulate before checking for a winner) of 1,000 for all experiments. We used the reserved data streams to tune VFDT's two remaining parameters: τ and δ. We found the best setting for post-pruning to be $\tau = 0.05, \delta = 1^{-7}$; the best for no-pruning to be $\tau = 0.1, \delta = 1^{-9}$; and the best for pre-pruning to be $\tau = 0.1, \delta = 1^{-5}$.

We compared the algorithms' abilities to learn from very large data streams. The incremental systems, VFDT and ITI, were allowed to learn from the streams directly. We ran ITI on the first 100,000 examples in each stream: it was too slow to learn from more. We ran VFDT on the entire streams and limited its dynamic memory allocations to 600MB (the rest of the memory on our computers was reserved for the data generating process and other system processes). For the batch systems, C4.5 and SPRINT, we sampled several data sets of increasing size from the data streams and ran the learners on each of them in turn. C4.5 was run on data sets up to 800MB in size, which contained 2 million samples. We allowed SPRINT to use up to 1.6GB of data, or 4 million samples.

Figures 1 and 2 show the runtime of the algorithms averaged over all 45 of the data test streams. The runtimes of all the systems seem to increase linearly with additional data. ITI and SPRINT were the slower of the systems by far; they were approximately two orders of magnitude slower than C4.5 and VFDT. ITI was slower because of the cost of incrementally keeping its model up-to-date as every example arrives, and suggests that this type of incremental maintenance is not viable for massive data streams. SPRINT was slower because it needed to do a great deal of additional I/O to scan the entire training set once per level of the tree it learned. VFDT and C4.5 took the same order of time to process the same number of examples: VFDT was approximately 3 times slower (up to the point where C4.5 was not able to run on more data because it ran out of RAM).

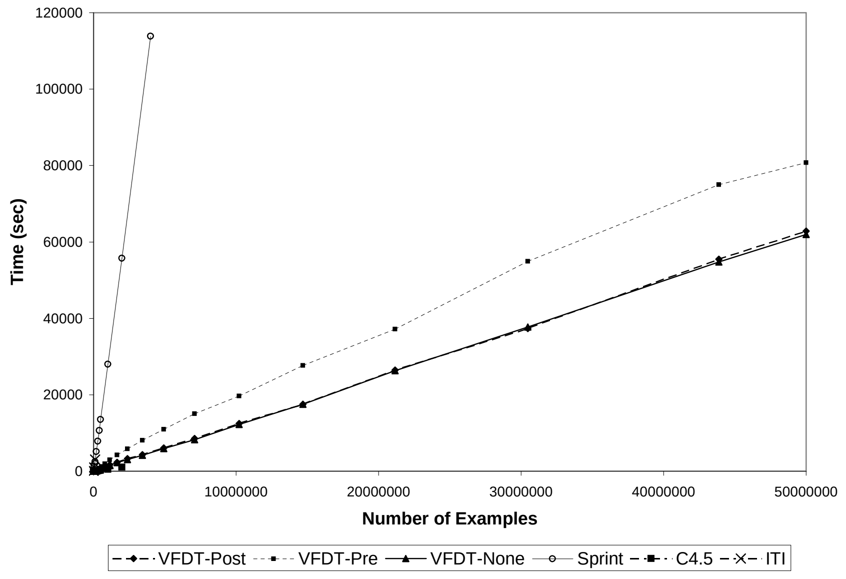

Figure 1: Average runtime of the algorithms.

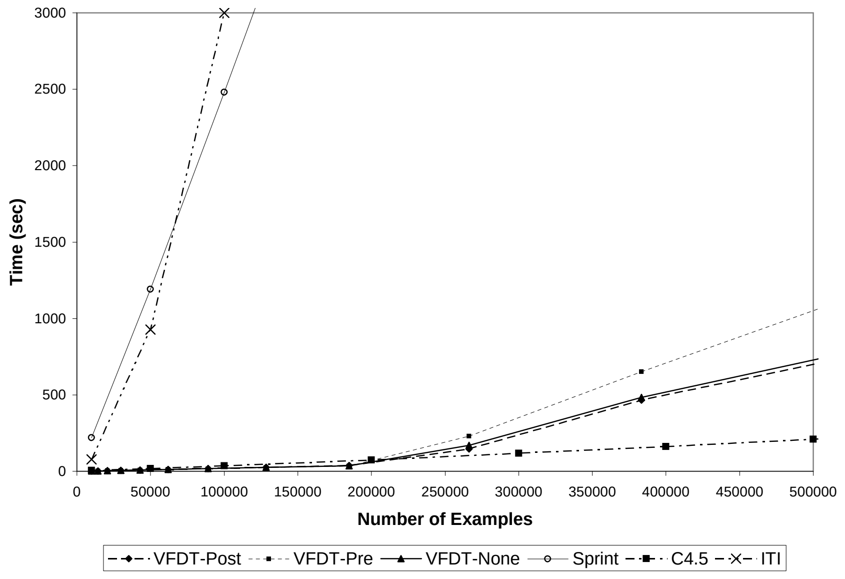

Figure 2: Average runtime of the algorithms (zoom).

VFDT used this additional time to track its memory allocations, swap searches into and out of memory, and manage example caches. Remember, however, that C4.5 needed to be run once for each test point, while VFDT was able to incrementally incorporate examples, and produce all test points with a single pass over each of the data streams. Thus VFDT actually took far less time to run. VFDT-None (with no pruning) was the fastest of the three VFDT settings. VFDT-post was about 2% slower, with the additional time being used to perform post-pruning (at each test point a copy of the tree was made and pruned). VFDT-pre was about 33% slower than the other VFDT settings. That happened because it learned smaller trees, usually had a larger fraction of its searches in RAM at a time, and thus spent more time updating sufficient statistics and checking for winning searches. (The other VFDT settings ignored many more examples that were not relevant to searches they had in memory when those examples arrived.)

Next, we compared the error rates of the models learned by the systems. Figure 3 shows the results. The final error rate achieved by each of the three VFDT systems was better than the final error rate achieved by any of the non-VFDT systems. This demonstrates that VFDT is able to take advantage of examples past those that could be used by the other systems to learn better models. It is also interesting to note that one of the VFDT systems had the best–or nearly the best–average performance after any number of examples: VFDT-Pre with less than 100,000 examples, VFDT-None from there to approximately 5 million examples, and VFDT-Post from there until the end at 50 million examples. We also compared the systems on their ability to quickly learn models with low error rate. Figure 4 shows how the error rates of the models learned by the algorithms improved over time, and thus which is able to most quickly produce high quality results. One of the VFDT settings had the best error rate after nearly any amount of time (C4.5 was slightly better at approximately 1000 seconds). This means that, when faced with an open ended data stream, it is almost always better to let VFDT monitor it than to draw a sample from it and run one of the other algorithms on it.

Figure 5 shows the average number of nodes in the models learned by each of the algorithms. The number of nodes learned scaled approximately linearly with additional training examples for all of the systems. C4.5 learned the largest trees of any of the systems by several orders of magnitude; and VFDT-Post learned the smallest by an order of magnitude. Notice that SPRINT learned many more nodes than VFDT-Post even though they used the same pruning implementation. This occurred because SPRINT's model contained every split that had gain on the training data, while the splits in VFDT-Post's tree were chosen, with high confidence, by our method. VFDT's simultaneous advantage with error rate and concept size is additional evidence that our method for selecting splits is effective.

Figures 6 and 7 show how the algorithms' performance varied with changes to the amount of noise added (n) and to the size of the target concepts (f). Each data point represents the average error rate achieved by the respective algorithm on all the runs with the indicated n and f. VFDT-Post had the best performance in every setting, followed closely by VFDT-None. VFDT-Pre's error rate was slightly higher than SPRINT's on the runs with the highest noise level and the runs with the largest concepts. These results suggest that VFDT's performance scales gracefully as concept size or noise increases.

We evaluated VFDT's ability to learn with more stringent RAM limitations by running VFDT-None on all of the data streams with a memory allocation limit of 150MB. At

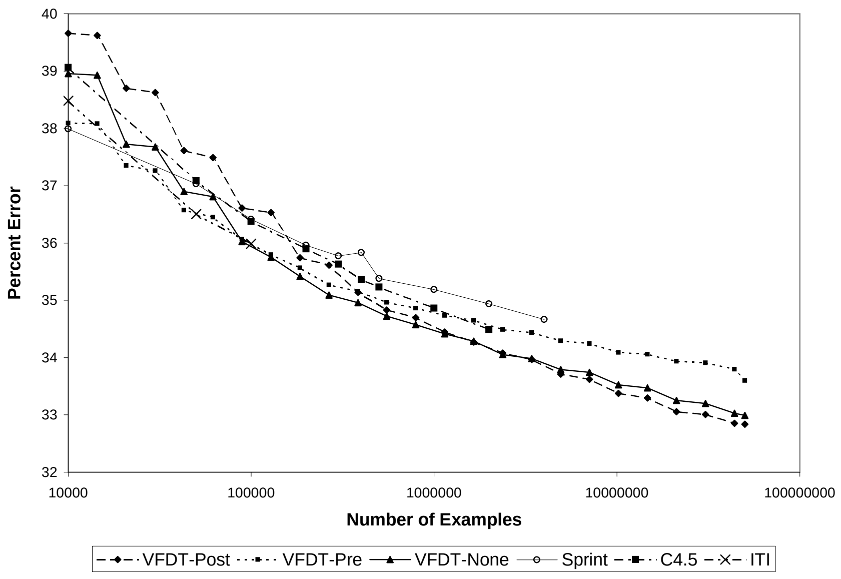

Figure 3: Average test set error rate of the algorithms vs. number of training examples.

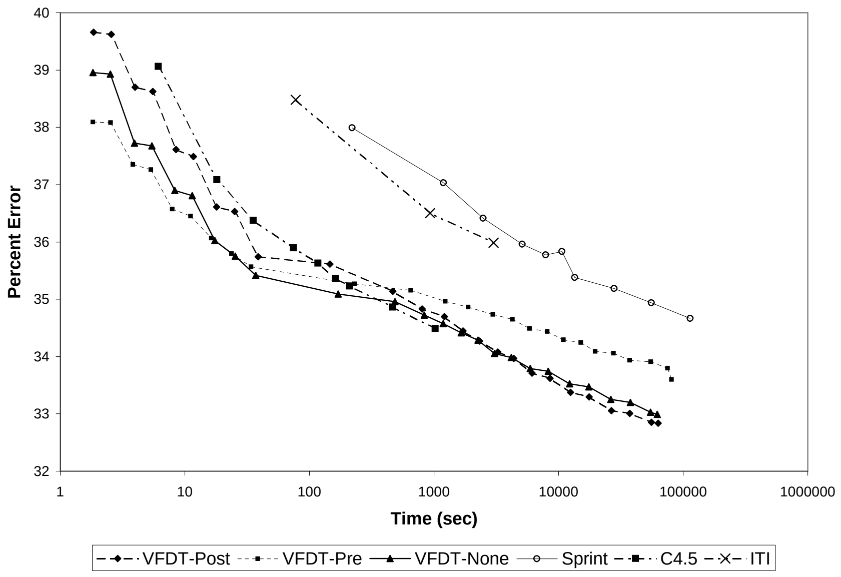

Figure 4: Average test set error rate of the algorithms vs. time.

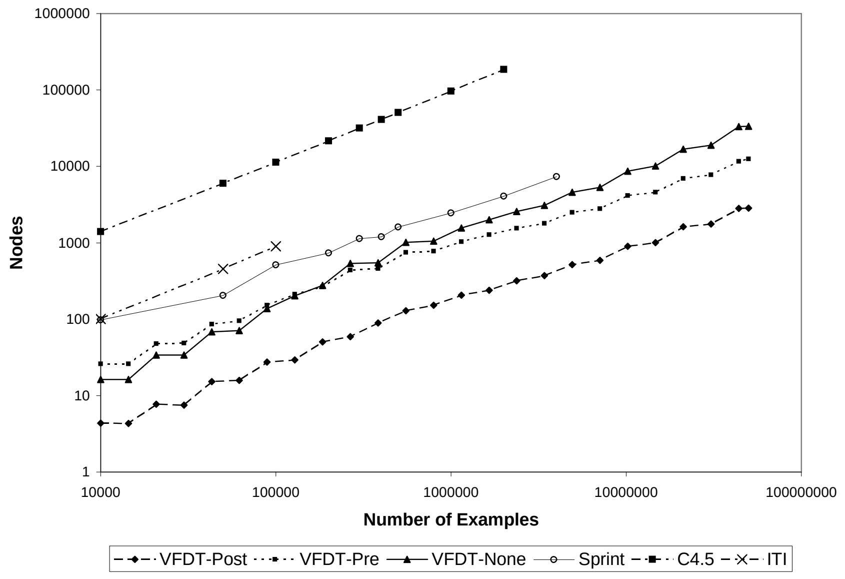

Figure 5: Average number of nodes induced by the algorithms.

the beginning of the run, when VFDT is able to keep all data in 150MB of RAM, the variations produce the same results. After approximately 100,000 training examples, the variation with 150MB of RAM is forced to begin freeing example caches and moving searches out of RAM. At this point its performance lags behind the 600MB version. Somewhat surprisingly, however, the 150MB version is able to catch after processing approximately 5 million examples. We also measured the runtimes of these variations. The 150MB version is approximately twice as fast as the 600MB one. Looking deeper, we found that the 600MB variation spent much more time checking to see if it had seen enough data to make a search step. This makes sense, as the 600MB version had many more searches active at a time than the 150MB version.

We calculated the rate at which VFDT is able to incorporate training examples to determine how large a data stream it is able to keep up with in real time. VFDT-None was able to learn from an average of 807 examples per second. Extrapolating from this, we estimate it is able to learn from approximately 70 million examples per day. Remember that these experiments were conducted on 1 GH PIII processors–modest hardware by today's standards. Further, if faced with a faster data stream VFDT can be accelerated by increasing Δn, by more aggressively disabling learning at unpromising leaves, etc. With such settings we have observed sustained processing rates of over 2,000 examples per second (several hundred million examples per day).

5. Application of VFDT to Web data

We further evaluated VFDT by using it to mine the stream of Web page requests emanating from the whole University of Washington main campus. The nature of the data is described in detail in Wolman et al. (1999). In our experiments we used a one-week anonymized trace of all the external Web accesses made from the university campus in May 1999. There were

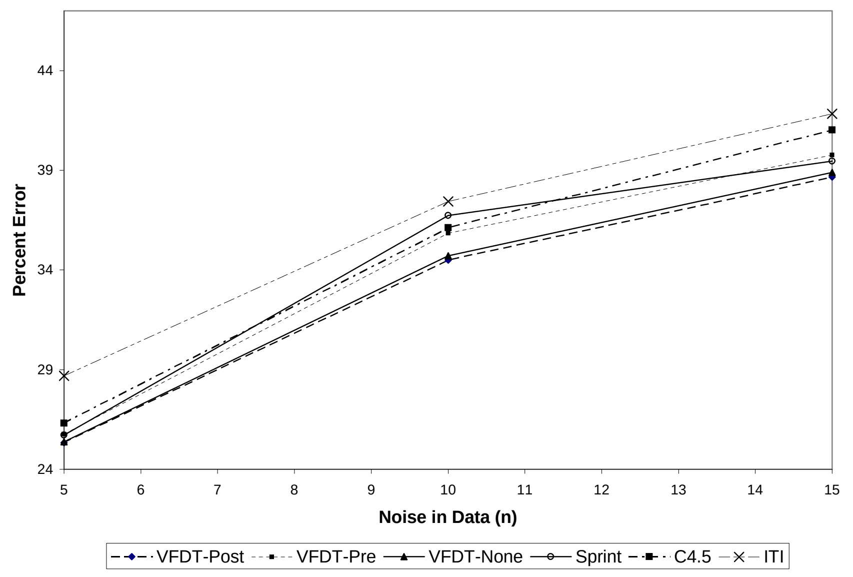

Figure 6: Average performance by noise level.

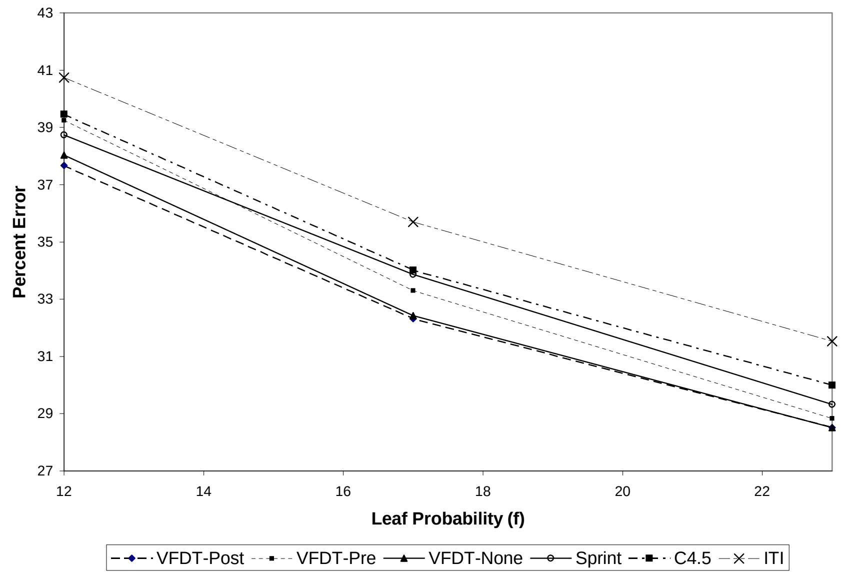

Figure 7: Average performance by concept size.

Table 4: Data sets used in the Web experiments.

Data set	# Train Exs	# Test Exs	% Class 0
30 sec	10,850k	1,944k	46.0
1 min	7,921k	1,419k	45.4
5 min	3,880k	696k	52.5
15 min	2,555k	454k	58.1

Table 5: Results of the Web experiments. 'Match Time' is the time it took VFDT to match C4.5's best accuracy. 'VFDT Time' is the time it took VFDT to do six complete scans of the training set.

Data set	C4.5 Error	VFDT Error	C4.5 Time	Match Time	VFDT Time
30 sec	37.70 %	36.95 %	60k sec	5k sec	247k sec
1 min	37.30 %	36.67 %	71k sec	6k sec	160k sec
5 min	33.59 %	33.23 %	61k sec	15k sec	72k sec

23,000 active clients during this one-week trace period, and the entire university population is estimated at 50,000 people (students, faculty and staff). The trace contains 82.8 million requests, which arrive at a peak rate of 17,400 per minute. The size of the compressed trace file is about 20 GB. Each request in the log is tagged with an anonymized organization ID that associates the request with one of the 170 organizations (colleges, departments, etc.) within the university. One purpose this data can be used for is to improve Web caching. The key to this is predicting as accurately as possible which hosts and pages will be requested in the near future, given recent requests. We applied decision tree learning to this problem in the following manner. We split the campus-wide request log into a series of equal time slices $T_0, T_1, \ldots, T_t, \ldots$. We used time slices of 30 seconds, 1 minute, 5 minutes, and 15 minutes for our experiments. Let N_T be the number of time slices in a day. For each organization $O_1, O_2, \ldots, O_i, \ldots, O_{170}$ and each of the 244k hosts appearing in the logs $H_1, \ldots, H_j, \ldots, H_{244k}$, we maintain a count of how many times the organization accessed the host in the time slice, C_{ijt}. Then for each time slice and host accessed in that time slice (T_t, H_j) we generate an example with attributes: $C_{1,jt}, \ldots, C_{ijt}, \ldots C_{170,jt}$, $t \bmod N_T$, and class 1 if any request is made to H_j in time slice T_{t+1} and 0 if not. This can be carried out in real time using modest resources by keeping statistics on the last and current time slices C_{t-1} and C_t in memory, only keeping counts for hosts that actually appear in a time slice (we never needed more than 30k counts), and outputting the examples for C_{t-1} as soon as C_t is complete. Testing was carried out on the examples from the last day in each of the data sets. See Table 4 for more details of the data sets produced using this method.

We ran C4.5 and VFDT on these data sets, on a 1GH machine with 1GB of RAM running Linux. We tested C4.5 on data sets up to 1.2M examples (roughly 800 MB; the remaining RAM was reserved for C4.5's learned tree and system processes). C4.5's data sets were created by random sampling from the available training data. Notice that the 1.2M examples C4.5 was able to process is less than a day's worth for the '30 sec' and '1 min' data sets and less than 50% of the training data in the remaining cases. We allowed

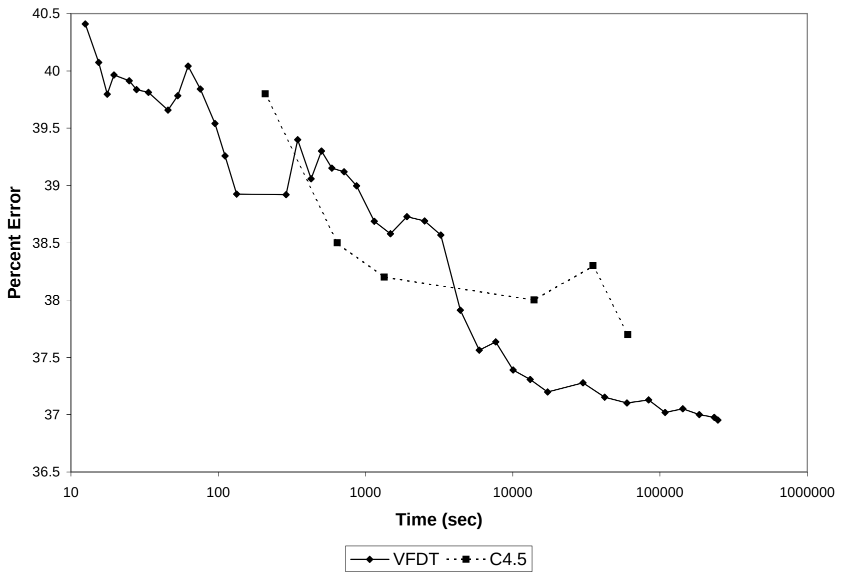

Figure 8: Time vs. Error Rate on the '30 sec' data set.

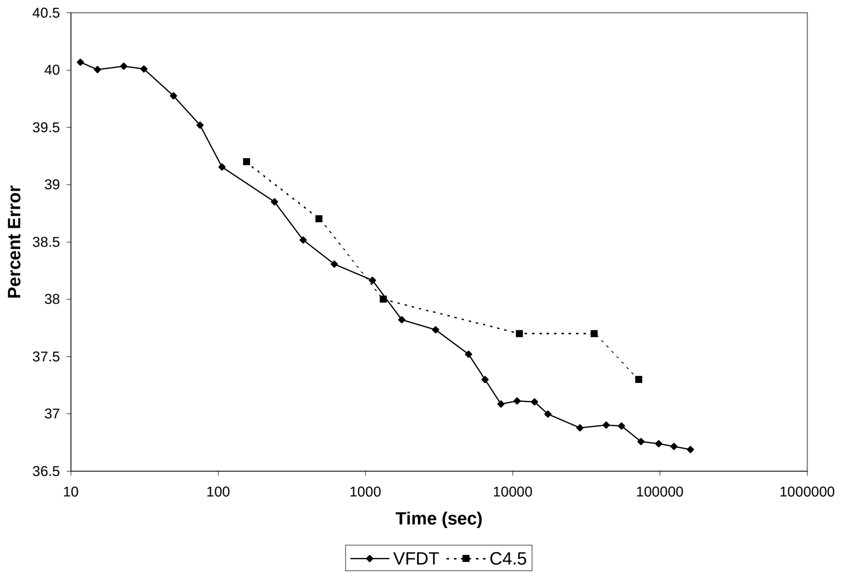

Figure 9: Time vs. Error Rate on the '1 min' data set.

VFDT to do six passes over the training data and thus incorporate each training example in its tree at six different places and we allowed it to use 400MB of RAM. We tuned the algorithms' parameters on the '15 min' data set. C4.5's best prune confidence was 5%, and VFDT's best parameters were: no-pruning, $\delta = 1^{-7}$, $\tau = .1$, and $\Delta_n = 300$. VFDT learned from the numeric attributes as described in Section 3.5 with $M = 1000$ in all cases. (The number of distinct values for an attribute was nearly always less than this limit.)

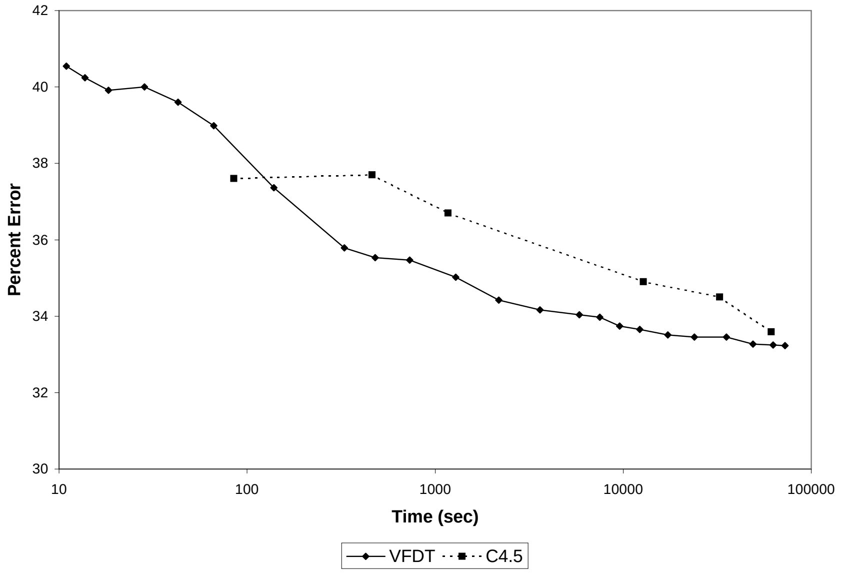

Figure 10: Time vs. Error Rate on the '5 min' data set.

Table 5 summarizes the results of these experiments. VFDT was able to learn a more accurate model than C4.5 on every data set. VFDT's improvement over C4.5 was largest on the '30 sec' data set and was smallest on the '5 min' data set. Notice that C4.5 was able to use about 11% of the '30 sec' data set and about 31% of the '5 min' data set. This supports the notion that VFDT's performance improvement over a traditional system will depend on the amount of additional data that it is able to process past the point where the traditional system runs into a resource limit. (We also expect that VFDT's improvement over a traditional system will depend on the complexity of the underlying concept: simpler concepts can be more successfully modeled with less training data.) VFDT was also substantially faster than C4.5 in the following sense. We noted the accuracy of the best model learned by C4.5 and the time it took it to learn it. We then examined VFDT's results to determine how long it took it to learn a model with that same accuracy. We found that VFDT achieved C4.5's best accuracy an order of magnitude faster on the '30 sec' and '1 min' data sets and 4 times faster on the '5 min' data set. After this point, VFDT was able to use additional data to improve its models. Looking further, we found that VFDT was able to learn from any fixed amount of training data an order of magnitude faster than C4.5 on this domain. Figures 8, 9 and 10 contain a more detailed picture of VFDT's simultaneous speed and accuracy advantage; they show the amount of learning time on a logarithmic scale on the x-axis and the error rate of the model learned in the indicated amount of time on the y-axis. VFDT has a more accurate model than C4.5 after nearly any amount of learning time. (the region where C4.5 has an advantage in Figure 8 is only about 1.6% of the entire run). We attribute this to VFDT's sampling method which allows it to use just the right amount of data to make each decision, while C4.5 must use all data for every decision.

In summary, our experiments in this domain demonstrate the utility of a tool like VFDT in particular and of our method for mining high-speed data streams in general: VFDT achieves the same results as the traditional system, in an order of magnitude less time; VFDT is then able to use additional time and data to learn more accurate models, while traditional system's memory requirements keep them from learning on more data; finally VFDT's incremental nature allows it to smoothly refine its model as more data arrives, while batch systems must be rerun from scratch to incorporate the latest data.

6. Time-Changing Concepts

In this section we use the techniques developed in our general scaling framework to extend VFDT to learn from time-changing data streams. We call this learner CVFDT (Concept-drifting Very Fast Decision Tree learner). CVFDT is an extension to VFDT which maintains VFDT's speed and accuracy advantages but adds the ability to detect and respond to changes in the example-generating process. Like other systems with this capability (Widmer and Kubat, 1996, Ganti et al., 2000), CVFDT works by keeping its model consistent with a sliding window of examples. However, unlike most of these other systems, it does not need to learn a new model from scratch every time a new example arrives. Instead, it monitors the quality of its old decisions on the new data and adjusts those that are no longer correct. In particular, when new data arrives, CVFDT updates the sufficient statistics at its nodes by incrementing the counts corresponding to the new examples and decrementing the counts corresponding to the oldest examples in the window (which need to be forgotten). This will statistically have no effect if the underlying concept is stationary. If the concept is changing, however, some splits that were previous selected will no longer appear best because other attributes will have higher gain on the new data. When this happens, CVFDT begins to grow an alternative subtree with the new best attribute at its root. The old subtree is replaced by the alternate one when the alternate becomes more accurate on new data.

Table 6 contains a pseudo-code outline of the CVFDT algorithm. CVFDT does some initializations, and then processes examples from the stream S indefinitely. As each example $(\mathbf{x}, y)$ arrives, it is added to the window[6], an old example is forgotten if needed, and $(\mathbf{x}, y)$ is incorporated into the current model. CVFDT periodically scans DT and all alternate trees looking for internal nodes whose sufficient statistics indicate that some new attribute would make a better split than the attribute currently used to split at the node. An alternate subtree is started at each such node.

Table 7 contains pseudo-code for the tree-growing portion of the CVFDT system. It is similar to the VFDT algorithm, but CVFDT monitors the validity of its old decisions by maintaining sufficient statistics at every node in DT (instead of only at the leaves like VFDT). Forgetting an old example is slightly complicated by the fact that DT may have grown or changed since the example was initially incorporated. To avoid forgetting an example from a node that has never seen it, nodes are assigned a unique, monotonically increasing ID as they are created. When an example is added to W, the maximum ID of the leaves it reaches in DT 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 in DT whose ID is less than or equal to the stored ID.

6. The window is stored in RAM if resources are available: otherwise it is kept on disk.

Table 6: The CVFDT algorithm.

```
Inputs:  $S$          is a sequence of examples,
         $\mathbf{X}$          is a set of discrete attributes,
         $G(.)^{+}$ and $G(.)^{-}$, upper and lower bounds on a split evaluation function $G(.)$
         $\delta$          is one minus the desired probability of choosing the correct
                    attribute at any given node,
         $\tau$          is a user specified tie threshold,
         $f$, is a bound function (we use the Hoeffding bound, see Equation 1).
         $w$          is the size of the window,
         $n_{drift}$ is the number of examples between checks for drift.
Output:  $DT$        is a decision tree.

Procedure CVFDT($S, \mathbf{X}, G^{+}, G^{-}, \delta, \tau, f, w, n_{drift}$)
/* Initialize */
Let $DT$ 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 $W =$ be the window of examples, initially empty.
    For each value $x_{ij}$ of each attribute $X_i \in \mathbf{X}$
        /* The sufficient statistics needed to calculate $G$ */
        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 $DT$ 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 end of $W$.
    If $|W| > w$
        Let $((\mathbf{x}_w, y_w), ID_w)$ be the last element of $W$
        ForgetExample($DT, n, (\mathbf{x}_w, y_w), ID_w$)
        Let $W = W - ((\mathbf{x}_w, y_w), ID_w)$
    CVFDTGrow($DT, n, G^{+}, G^{-}, (\mathbf{x}, y), \delta, \tau$)
    If there have been $n_{drift}$ examples since the last checking of alternate trees
        CheckForDrift($DT, n, \delta, \tau$)
        DiscardPoorAlternatives($DT$)
Return $DT$.
```

Table 7: The CVFDTGrow procedure.

```
Inputs: $DT$      Partially induced tree to check for growth,
        $\mathbf{X}$       is a set of discrete attributes,
        n         the set of sufficient statistics being maintained for the run,
        $G(.)^+$ and $G(.)^-$, upper and lower bounds on a split evaluation function $G(.)$
        $(\mathbf{x}, y)$  the training example to add to the tree,
        $\delta$         is one minus the desired probability of choosing the correct
                  attribute at any given node,
        $\tau$         is a user specified tie threshold,
        $f$, a bound function (we use the Hoeffding bound, see Equation 1).

Procedure CVFDTGrow($DT, \mathbf{X}, n, G, (\mathbf{x}, y), \delta, \tau, f$)
Sort $(\mathbf{x}, y)$ into a leaf $l$ using $DT$.
Let $P$ be the set of nodes traversed in the sort.
For each node $l_p$ 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 $l$.
If the examples seen so far at $l$ are not all of the same class, then
    Compute $\overline{G}_l^+(X_i)$ and $\overline{G}_l^-$ for each attribute $X_i \in \mathbf{X}_l$
        using the counts $n_{ijk}(l)$.
    Let $X_a$ be the attribute with highest $\overline{G}_l^-$.
    Let $X_b$ be the attribute other than $X_a$ with highest $\overline{G}_l^+$.
    Compute $\epsilon$ using $f(.)$.
        If $\overline{G}_l^-(X_a) - \overline{G}_l^+(X_b) > \epsilon$ or $\epsilon < \tau$, 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 $ALT(l_m) = \{\}$.
            For each class $y_k$ and each value $x_{ij}$ of each attribute $X_i \in \mathbf{X_m}$
                Let $n_{ijk}(l_m) = 0$.
```

CVFDT determines that a chosen split attribute would no longer be selected when $\Delta\overline{G}_l \leq \epsilon$ and $\epsilon > \tau$. When it finds such a node, CVFDT knows that it either initially made a mistake splitting on X_a (which should happen less than δ% of the time), or that something about the example generating process has changed. In either case, CVFDT will need to take action to correct DT. Let T_c be the subtree rooted by the node where the change was detected. CVFDT grows, in parallel, an alternate subtree to T_c as described in pseudo-code in Table 8. The criteria used to determine when to start alternate trees is very similar to the one used to choose initial splits, except the tie threshold is tighter ($\tau/2$ instead of τ) to avoid excessive alternate tree creation. Alternate trees are grown the same way DT is, via recursive calls to the CVFDT procedures. Periodically, each node with a non-empty set of alternate subtrees enters a testing mode to determine if it should be replaced by one of its alternate subtrees. Once in this mode, the testing node l_{test} collects the next m training examples that arrive at it and, instead of using them to learn additional structure, it uses them to compare the accuracy of the subtree it roots with the accuracies of all of its alternate subtrees. If the most accurate alternate subtree is more accurate than l_t, l_t is replaced by the alternate. During the test phase CVFDT also prunes any alternate subtree that is not making progress (i.e., whose accuracy is not increasing over time). In particular, for each alternate subtree of l_t, l_a^i, CVFDT remembers the smallest accuracy difference ever achieved between the two, $\Delta_{min}(l_t, l_a^i)$. CVFDT prunes any alternate whose current validation phase accuracy difference is not at least $\Delta_{min}(l_t, l_a^i) + 1\%$. Note that this parameter can be adjusted (even dynamically) so that CVFDT is more aggressive about pruning unpromising alternate subtrees when RAM is short. In addition, CVFDT supports a parameter which limits the total number of alternate trees being grown at any one time; once this number is reached no new alternates are started until some are pruned.

Instead of using alternate trees one might consider this simpler method for adjusting to change: prune any node where change is detected to a leaf and start growing from there. However, this policy performs poorly as it forces a single leaf to do the job previously done by a whole subtree: even if a subtree is outdated, it will often still be better than the best single leaf. Pruning to a leaf will be particularly bad when the change is at or near the root of DT, as it will result in drastic short-term reductions in accuracy, not acceptable when a parallel process is using DT to make critical decisions. Our alternate tree method avoids such drastic changes and allows CVFDT to adjust to drift in a smooth, fine-grained manner.

The rate of drift can change during a run, and thus CVFDT has the ability to dynamically change w to adapt. For example, it may make sense to shrink w when many of the nodes in DT become questionable at once, or in response to a rapid change in data rate, as these events could indicate a sudden concept change. Similarly, it may make sense to increase w when there are few questionable nodes, because this may indicate that the concept is stable. CVFDT is able to dynamically adjust the size of its window in response to user-supplied events. Events are specified in the form of hook functions which monitor S and DT and set the window size when appropriate. CVFDT changes the window size simply by updating w and immediately forgetting any examples that no longer fit in W.

We now discuss a few of the properties of the CVFDT system and briefly compare it with VFDT-Window, a learner that reapplies VFDT to W for every new example. CVFDT requires memory proportional to $O(ndvc)$ where n is the number of nodes in CVFDT's

Table 8: The CheckForDrift procedure.

Inputs:	DT	Partially induced tree to check for growth,
	$\mathbf{X}$	is a set of discrete attributes,
	n	the set of sufficient statistics being maintained for the run,
	$G(.)^+$ and $G(.)^-$, upper and lower bounds on a split evaluation function $G(.)$	
	δ	is one minus the desired probability of choosing the correct attribute at any given node,
	τ	is a user specified tie threshold,
	f, a bound function (we use the Hoeffding bound, see Equation 1).	

```
Procedure CheckForDrift($DT, \mathbf{X}, n, G^+, G^-, \delta, \tau, f$)
For each node $l$ in $DT$ that is not a leaf
    For each tree $T_{alt}$ in $ALT(l)$
        CheckForDrift($T_{alt}, n$)
    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 $\epsilon$ using $f(.)$.
        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 $\overline{G}$ of 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 $n_{ijk}(l_m) = 0$.
```

main tree and all alternate trees, d is the number of attributes, v is the maximum number of values per attribute, and c is the number of classes. The window of examples can be in RAM, or can be stored on disk. Therefore, like VFDT, CVFDT's memory requirements are dominated by the sufficient statistics and are independent of the total number of examples seen. At any point during a run, CVFDT will have available a model which reflects the current concept generating W. It is able to keep this model up-to-date in time proportional to $O(l_c dvc)$ per example, where l_c is the length of the longest path an example will have to take through DT and all alternate trees. VFDT-Window requires $O(l_v dvcw)$ time to keep its model up-to-date for every new example, where l_v is the maximum depth of DT. VFDT is a factor of wl_v/l_c worse than CVFDT; empirically, we observed l_c to be smaller than l_v in all of our experiments (because CVFDT has a smaller tree that more accurately represents the concept). Despite this large time difference, CVFDT's drift mechanisms allow it to produce a model of similar accuracy to VFDT. The structure of the models induced by the two may, however, be significantly different for the following reason. VFDT-Window uses the information from each training example at one place in the tree it induces: the leaf where the example falls when it arrives. This means that VFDT-Window uses the first examples from W to make a decision at its root, the next to make a decision at the first level of the tree, and so on. After an initial building phase, CVFDT will have a fully induced tree available. Every new example is passed through this induced tree, and the information it contains is used to update statistics at every node it passes through. This difference can be an advantage for CVFDT, as it allows the induction of larger trees with better probability estimates at the leaves. It can also be a disadvantage and VFDT-Window may temporarily be more accurate when there is a large concept shift part-way through W. This is because VFDT-Window's leaf probabilities will be set by examples near the end of W while CVFDT's will reflect all of W. Also notice that, even when the structure of the induced tree does not change, CVFDT and VFDT-Window can outperform VFDT simply because their leaf probabilities (and therefore class predictions) are updated faster, without the "dead weight" of all the examples that fell into leaves before the current window.

6.1 Empirical Study

We conducted a series of experiments comparing CVFDT to VFDT and VFDT-Window. Our goals were to evaluate CVFDT's ability to scale up, to evaluate CVFDT's ability to deal with varying levels of drift, and to identify and characterize the situations where CVFDT outperforms the other systems.

Our experiments used synthetic data that was generated from a changing concept based on a rotating hyperplane. A hyperplane in d-dimensional space is the set of points $\mathbf{x}$ that satisfy

$$\sum_{i=1}^{d} w_i x_i = w_0 \tag{11}$$

where x_i is the ith coordinate of $\mathbf{x}$. Examples for which $\sum_{i=1}^{d} w_i x_i \geq w_0$ are labeled positive, and examples for which $\sum_{i=1}^{d} w_i x_i < w_0$ are labeled negative. Hyperplanes are useful for simulating time-changing concepts because we can change the orientation and position of the hyperplane in a smooth manner by changing the relative size of the weights.

In particular, sorting the weights by their magnitudes provides a good indication of which dimensions contain the most information; in the limit, when all but one of the weights are zero, the dimension associated with the non-zero weight is the only one that contains any information about the concept. This allows us to control the relative information content of the attributes-and thus change the optimal order of tests in a decision tree representing the hyperplane-simply by changing the relative sizes of the weights. We sought a concept class that maintained the advantages of hyperplanes, but where the weights could be randomly modified without potentially causing the decision frontier to move outside the range of the data. To meet these goals we used a series of alternating class bands separated by parallel hyperplanes. We started with a reference hyperplane whose weights were initialized to 0.2 except for w_0 which was $0.25d$. To label an example, we substituted its coordinates into the left hand side of Equation 11 to obtain a sum s. If $|s| \leq w_0/10$ the example was labeled positive, otherwise if $|s| \leq w_0/5$ the example was labeled negative, and so on. Examples were generated uniformly in a d-dimensional unit hypercube. They were then labeled using the concept, and their continuous attributes were uniformly discretized into five bins. Noise was added by randomly switching the class labels of $p\%$ of the examples. Unless otherwise stated, each experiment used the following settings: five million training examples; $\delta = 0.0001$; $n_{drift} = 20,000$; $\Delta n = 300$; $\tau = 0.05$; $w = 100,000$; CVFDT's window on disk; no memory limits; no pre-pruning; a test set of 50,000 examples; and $p = 5\%$. CVFDT put nodes into alternate tree test mode after 9,000 examples and used test samples of 1,000 examples for the comparison. All runs were done on a 1GH Pentium III machine with 512 MB of RAM, running Linux.

The first series of experiments compares the ability of CVFDT and VFDT to deal with large concept-drifting datasets. Concept drift was added to the data sets in the following manner. Every 50,000 examples, w_1 was modified by adding $0.01d\sigma$ to it, and the test set was relabeled with the updated concept (with $p\%$ noise as before). σ was initially set to 1 and was multiplied by -1 at 5% of the drift points and also just before w_1 fell below 0 or rose above $0.25d$. Figure 11 compares the accuracy of the algorithms as a function of d, the dimensionality of the space. The reported values are obtained by testing the accuracy of the learned models every 10,000 examples throughout the run and averaging these results. Drift level, reported on the minor axis, is the average percentage of the test set that changes label at each point the concept changes. CVFDT is substantially more accurate than VFDT, by approximately 10% on average, and CVFDT's performance improves slightly with increasing d (from approximately 13.5% when $d = 10$ to approximately 10% when $d = 150$). Figure 12 compares the average size of the models induced during the run shown in Figure 11 (the reported values are generated by averaging after every 10,000 examples, as before). CVFDT's trees are substantially smaller than VFDT's, and the advantage is consistent across all the values of d we tried. This simultaneous accuracy and size advantage derives from the fact that CVFDT's tree is built on the 100,000 most relevant examples, while VFDT's is built on millions of outdated examples.

We next carried out a more detailed evaluation of CVFDT's concept drift mechanism. Figure 13 shows a detailed view of one of the runs from Figures 11 and 12, the one for $d = 50$. The minor axis shows the portion of the test set that is labeled negative at each test point (computed before noise is added to the test set) and is included to illustrate the concept drift present in the data set. CVFDT is able to quickly respond to drift, while VFDT's error

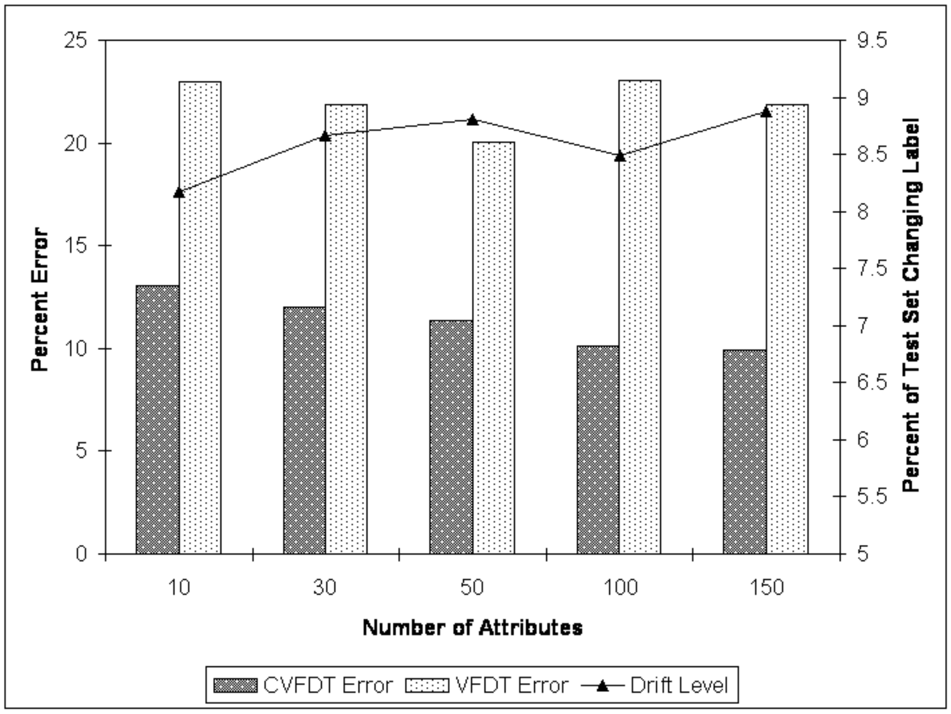

Figure 11: Error rates as a function of the number of attributes.

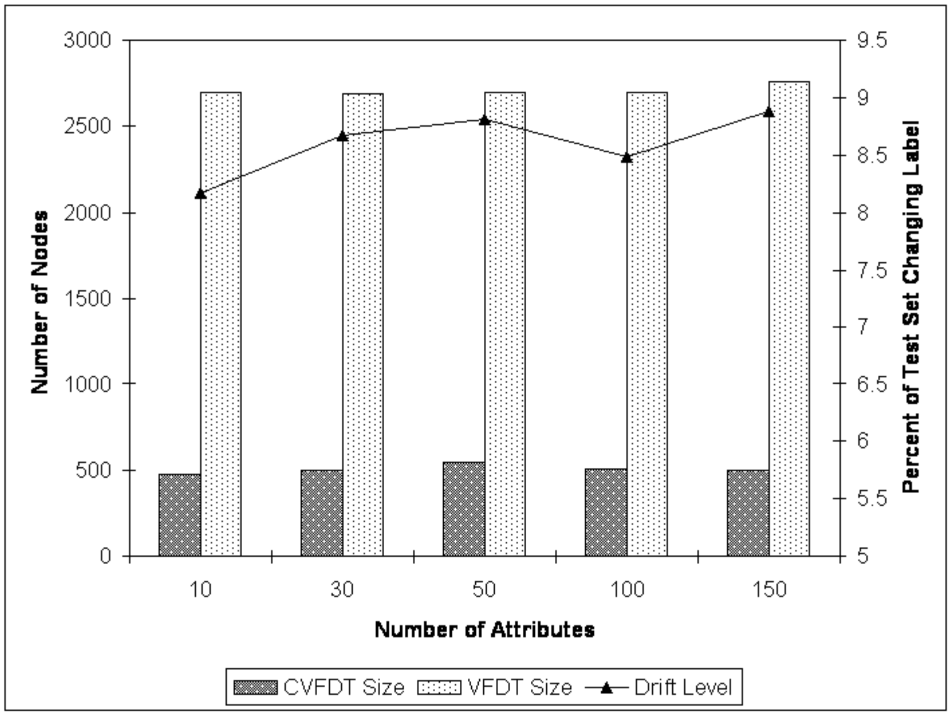

Figure 12: Tree sizes as a function of the number of attributes.

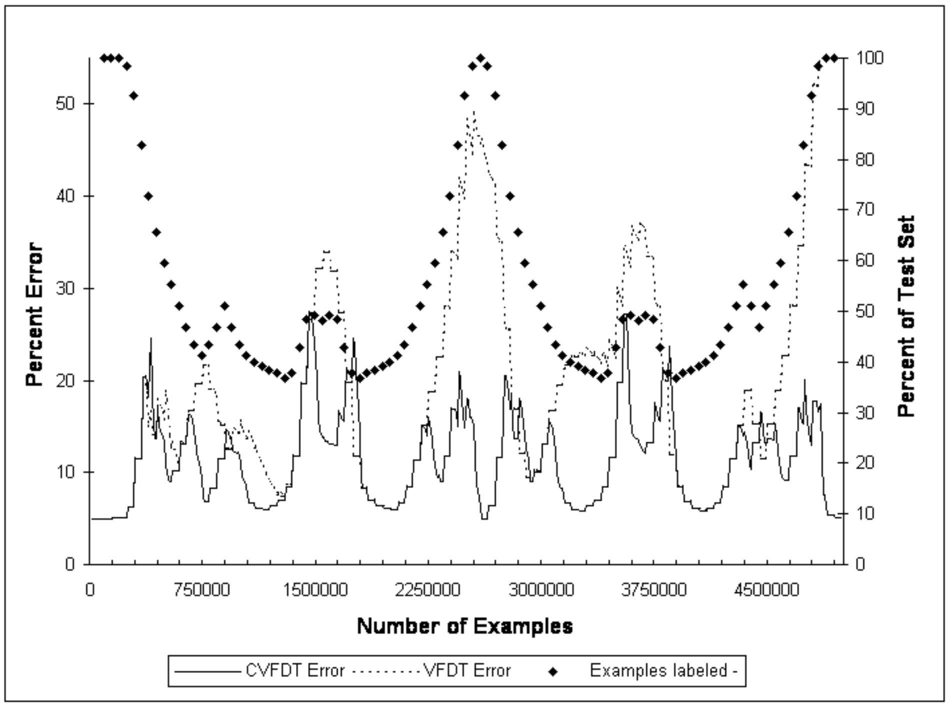

Figure 13: Error rates of learners as a function of the number of examples seen.

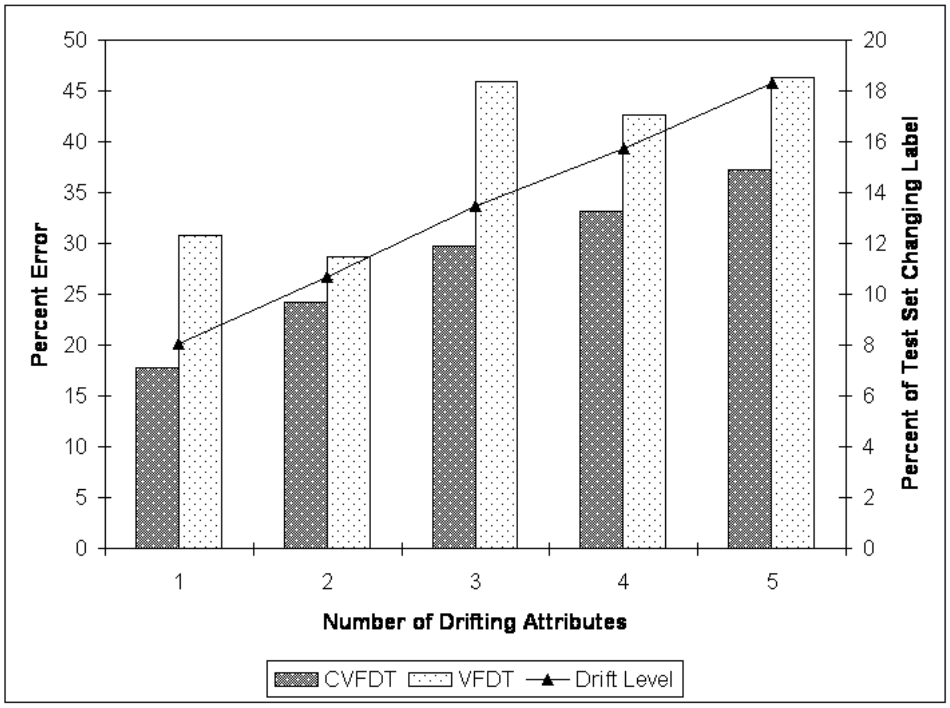

Figure 14: Error rates as a function of the amount of concept drift.

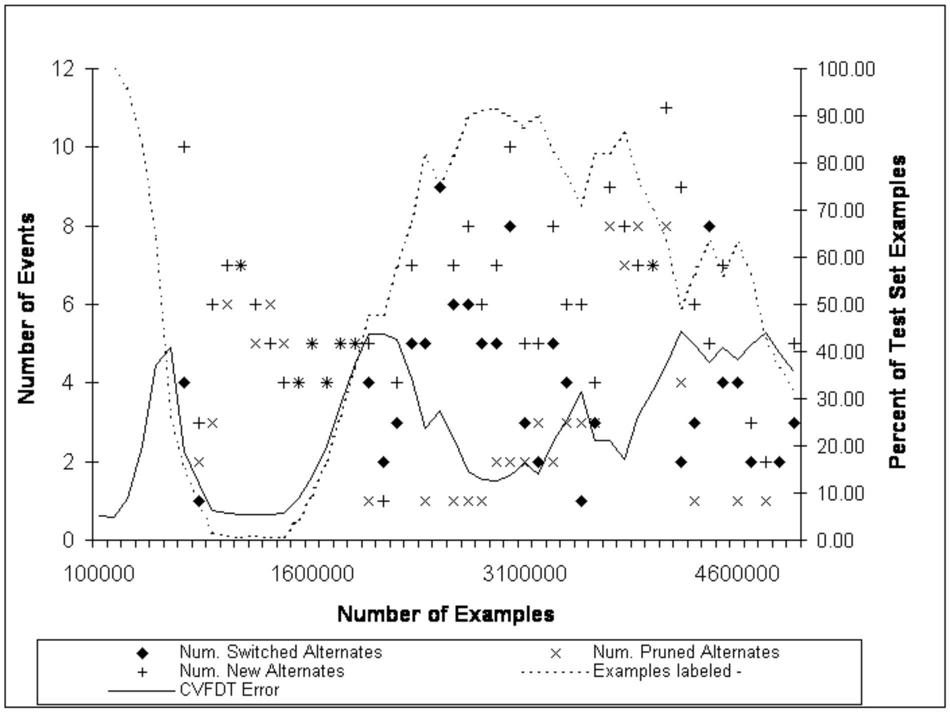

Figure 15: CVFDT's drift characteristics.

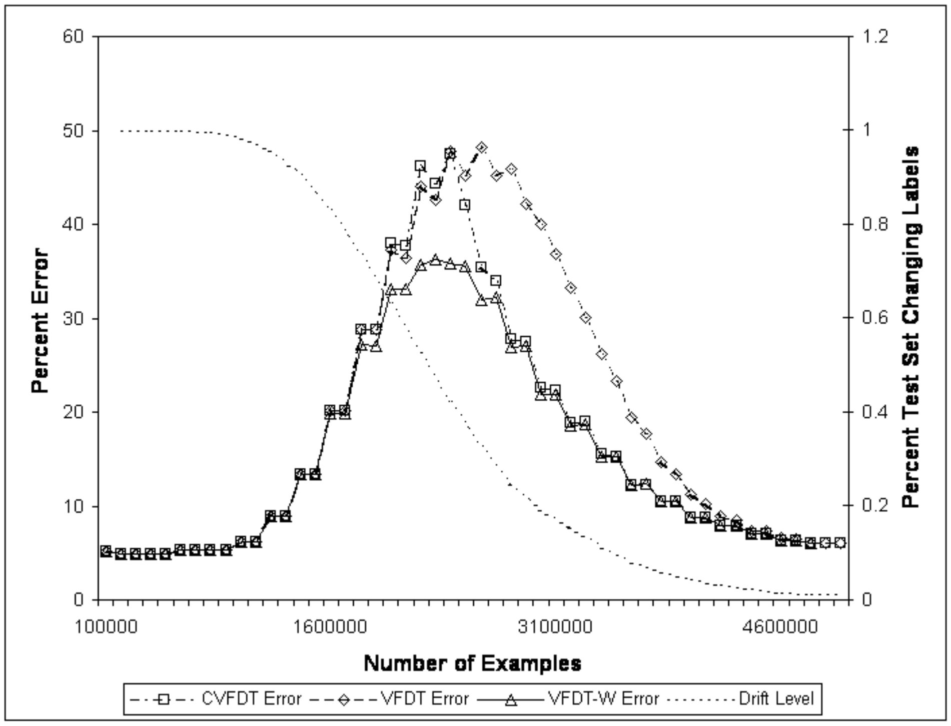

Figure 16: Error rates over time of CVFDT, VFDT, and VFDT-Window.

rate often rises drastically before reacting to the change. Further, VFDT's error rate seems to peak at higher and higher values as the run goes on, while CVFDT's error peaks seem to have approximately constant height. We believe this happens because VFDT has more trouble responding to drift when it has induced a larger tree and must replicate corrections across more outdated structure. CVFDT does not face this problem because it replaces subtrees when they become outdated. We gathered some detailed statistics about this run. CVFDT took 4.3 times longer than VFDT (5.7 times longer if including time to do the disk I/O needed to keep the window on disk). VFDT's average memory allocation over the course of the run was 23 MB while CVFDT's was 16.5 MB. The average number of nodes in VFDT's tree was 2696 and the average number in CVFDT's tree was 677, of which 132 were in alternate trees and the remainder were in the main tree.

Next we examined how CVFDT responds to changing levels of concept drift on five data sets with $d = 50$. Drift was added using a parameter d'. Every 75,000 examples, d' of the concept hyperplane's weights were selected at random and updated as before, $w_i = w_i + 0.01d\sigma_i$ (although σ_i now has a 25% chance of flipping signs, chosen to prevent too many weights from drifting in the same pattern). Figure 14 shows the comparison on these data sets. CVFDT substantially outperformed VFDT at every level of drift. Notice that VFDT's error rate approaches 50% for $d' > 2$, and that VFDT's variance from data point to data point is large. CVFDT's error rate seems to grow smoothly with increasing levels of concept change, suggesting that its drift adaptations are robust and effective.

To gain some insight into the way CVFDT starts new alternate subtrees, prunes existing ones, and replaces portions of DT with alternates. For this purpose, we instrumented a run of CVFDT on the $D = 2$ data set from Figure 14 to output a token in response to each of these events. We aggregated the events in chunks of 100,000 training examples, and generated data points for all non-zero values. Figure 15 shows the results of this experiment. (For example, the + near the top left of the figure indicates that 10 new alternate trees were started in preceding 100,000 examples.) During the run 264 alternate trees were started; 109 alternate subtrees were swapped into the main tree; and 128 alternate trees were pruned. Notice that most of the activity seems to occur when the examples in the test set are changing labels quickly.

To see how well CVFDT would compare to a system using traditional drift-tracking methods. We thus compared CVFDT, VFDT, and VFDT-Window. We approximated VFDT-Window by running VFDT on W for every 100,000 examples instead of for every example. The data set for the experiment had $d = 50$ and used the same drift settings used to generate Figure 14 with $D = 1$. Figure 16 shows the results. CVFDT's error rate was the same as VFDT-Window's, except for a brief period during the middle of the run when class labels were changing most rapidly. CVFDT's average error rate for the run was 16.3%, VFDT's was 19.4%, and VFDT-Window's was 15.3%. The difference in runtimes was very large. VFDT took about 10 minutes, CVFDT took about 46 minutes, and we estimate that VFDT-Window would have taken 548 days to do its complete run if applied to every new example. Put another way, VFDT-Window provides a 4% accuracy gain compared to VFDT, at a cost of increasing the running time by a factor of 17,000. CVFDT provides 75% of VFDT-Window's accuracy gain, and introduces a time penalty of less than 0.1% of VFDT-Window's.

CVFDT's alternate trees and additional sufficient statistics do not use too much RAM. For example, none of CVFDT's $d = 50$ runs ever grew to more than 70MB. We never observed CVFDT to use more RAM than VFDT; in fact it often used as little as half the RAM of VFDT. The systems' RAM requirements are dominated by the sufficient statistics which are kept at the leaves in VFDT, and at every node in CVFDT. We observed that VFDT often had twice as many leaves as there were nodes in CVFDT's tree and all alternate trees combined. This is what we expected: VFDT considers many more examples and is forced to grow larger trees to make up for the fact that its early decisions become incorrect due to concept drift. CVFDT's alternate tree pruning mechanism seems to be effective at trading memory for smooth transitions between concepts. Further, there is room for more aggressive pruning if CVFDT exhausts available RAM. Exploring this tradeoff is an area for future work.

7. Related Work

In this paper we explored the use of our method to learning decision trees from high-speed time-changing data streams. In previous work we used it (or an earlier version of it) to learning the structure of Bayesian networks (Hulten and Domingos, 2002), to k-mean clustering (Domingos and Hulten, 2001), to EM for mixtures of Gaussians (Domingos and Hulten, 2002), and to learning from very large relational databases (Hulten et al., 2003).

7.1 Scaling Up Learning Algorithms

There has been a great deal of work on scaling up learning algorithms to very large data sets. Perhaps the most relevant falls into the general category of sequential analysis (Wald, 1947). Examples of relevant machine learning frameworks that use these ideas include PALO (Greiner, 1996), Hoeffding races (Maron and Moore, 1994), the sequential induction model (Gratch, 1996), AdaSelect (Domingo et al., 2002), and the sequential sampling algorithm (Scheffer and Wrobel, 2001). The core idea in these is that training samples should be accumulated (roughly) one at a time until some quality criteria is met, and no longer. For example, PALO uses sampling to accelerate any hill-climbing search when the utility of each but must be estimated by sampling (because it can not be calculated directly). From an initial search state, PALO samples the quality of all adjacent states until one appears better than the rest with high confidence, moves to this winner state, and repeats. PALO assures that the final state of such a search will be a local optimum with a δ quality guarantee similar to ours, and achieves a global quality bound by using a stricter δ for each successive decisions (while we estimate the total number of decisions needed for the entire learning run and use the same δ for each). Our work extends the ideas in PALO (and these other sequential analysis systems) in the following three fundamental ways. First, we address a series of complications that occur when trying to apply these types of frameworks to real machine learning algorithms, including: working within limited RAM; using parallel searches; working with search types other than hill-climbing; early stopping on poor candidates; etc. Second, our framework explicitly addresses time-changing concepts, while none of the others do. Third, we have evaluated our work by applying it to more learning algorithms, and by conducting empirical studies on synthetic and real-world data sets orders of magnitude larger than any of these.

"Probably close enough" learning (John and Langley, 1996) can be viewed as a predecessor of our work. They consider a finite-size database (as opposed to an infinite one) as the reference for loss, and their approach is based on fitting a learning curve, offering no guarantees that the criterion will be satisfied. Learning curve methods (Meek et al., 2002, Provost et al., 1999) produce models on successively larger sub-samples of the training set until model quality appears to asymptote. Progressive sampling approaches attempt to iteratively determine the best number of examples to use, often via extrapolation of learning curves. However, they are hindered by the fact that real learning curves do not fit power laws and other simple forms well enough for reliable extrapolation (Provost et al., 1999). Our method provides guarantees that the necessary number of examples to satisfy a loss criterion has been reached.

Our method bears an interesting relationship to recent work in computational learning theory that uses algorithm-specific and run-time information to obtain better loss bounds (e.g., Freund (1998), Shawe-Taylor et al. (1996)). A first key difference is that we attempt to bound a learner's loss relative to the same learner running on infinite data, instead of relative to the best possible model (or the best possible from a given class). A second key difference is that we make more extensive use of run-time information in our bounds. These two changes make realistic bounds for widely-used learners (e.g., decision tree induction, K-means clustering) possible for the first time.

Another category of related work attempts to scale up algorithms by improving the way they access data on disk. For example, the SPRINT (Mehta et al., 1996) and SLIQ (Shafer et al., 1996) decision tree induction algorithms fall into this category. Rainforest (Gehrke et al., 2000), is a framework which uses these ideas, and others, and generalizes them to help construct decision tree induction algorithms for large data sets. Agrawal et al. (1993) presents another general discussion of how to develop mining algorithms with efficient data access. Their work casts several common mining algorithm (including decision tree induction and rule set learning) as search and uses a parallel decomposition (similar to ours) to learn independent parts of the models with a single scan over data. These systems have two major shortcomings which prevent them from mining high-speed data streams: they do not model time-changing concepts, and they require access to all training data before they do any mining – our framework overcomes both of these shortcomings.

Another important method for improving data access is AD-trees (Komarek and Moore, 2000). AD-trees are a summary structure which contain all the counts that could possibly be needed by a learning algorithm from a data set. AD-trees will often be much smaller than the original data set, and can thus greatly reduce the number disk accesses needed by a learning algorithm. Exploring ways of combining the advantages of AD-trees with our method is an interesting area for future work.

Incremental learning methods (also known as online, successive or sequential methods) fulfill many of the same desiderata as our framework. An incremental algorithm is one that can incorporate new training data into an existing model at any point. However, as we saw in our evaluation of ITI (Utgoff, 1994) in Section 4 many of the available algorithms of this type have significant shortcomings when learning from very large data sets. Some are reasonably efficient, but do not guarantee that the model learned will be similar to the one obtained by learning on the same data in batch mode. Some are highly sensitive to example ordering, potentially never recovering from an unfavorable set of early examples. Others produce the same model as the batch version, but at a high cost in efficiency.

One of the most relevant scalable learners is BOAT (Gehrke et al., 1999), an incremental decision tree induction algorithm that uses sampling to learn several levels of a decision tree in a single pass over training data. BOAT learns a decision tree as follows. It uses conventional methods to build a number of trees on samples of the database that fit in RAM. Any structure that all of these sample trees agree on is added to BOAT's main tree. And then BOAT is recursively called for each leaf of this tree. Periodically BOAT does a complete scan of the training database to verify that all of the structure in its final tree is the same that would have been added by a conventional algorithm learning from the entire data set. Any structure that is not validated is removed and regrown. BOAT works incrementally as follows. When new data is added to the database BOAT immediately performs a validation scan, removes portions of the tree that were changed by the new data, and relearns them. When the new data is drawn from the same distribution as the old one (that is, when the underlying concept is stable), BOAT can perform this maintenance extremely quickly. When drift is present, BOAT must discard and regrow portions of its induced tree, which can be very expensive when the drift is large or affects nodes near the root of the tree. CVFDT avoids the problem by using alternate trees and by relaxing the requirement that it learn exactly the tree that a batch system would. A comparison between BOAT and CVFDT is an area for future work.

7.2 Learning from Drifting Concepts

The STAGGER system (Schlimmer and Granger, 1986) was one of the first to explicitly address the problem of concept drift. Salganicoff (1993) studied drift in the context of nearest-neighbor learning. The FLORA system (Widmer and Kubat, 1996) used a window of examples, but also stored old concept descriptions and reactivated them if they seemed to be appropriate again. All of these systems were only applied to small databases (by today's standards). Kelly et al. (1999) addressed the issue of drifting parameters in probability distributions. Theoretical work on concept drift includes Long (1999) and Bartlett et al. (2000).

Several pieces of research on concept drift and context-sensitive learning are collected in a special issue of the journal *Machine Learning* (Widmer and , eds.). Other relevant research appeared in the ICML-96 Workshop on Learning in Context-Sensitive Domains (Kubat and Widmer, 1996) and the AAAI-98 Workshop on AI Approaches to Time-Series Problems (Danyluk et al., 1998). Turney (1998) maintains an online bibliography on context-sensitive learning.

Aspects of the concept drift problem are also addressed in the areas of activity monitoring (Fawcett and Provost, 1999), active data mining (Agrawal and Psaila, 1995) and deviation detection (Chakrabarti et al., 1998). The main goal here is to explicitly detect changes, rather than simply maintain an up-to-date concept, but techniques for the latter can obviously help in the former.

Perhaps the system most closely related to our approach is the combination of the DEMON framework (Ganti et al., 2000) and the incremental BOAT decision tree induction algorithm. DEMON was designed to help adapt any incremental learning algorithms to work effectively with time-changing data streams. DEMON assumes data arrives periodically, perhaps daily, in large blocks. Roughly, DEMON builds a model on every interesting suffix of the data blocks that are in the current window. For example, if the window contains n data blocks $B_1, ..., B_n$, DEMON will have a model for blocks $B_1, ..., B_n$; a model for blocks $B_2, ..., B_n$; a model for $B_3, ..., B_n$; etc. When a new block of data (block B_{n+1}) arrives, DEMON incrementally refines the model for $B_2, ..., B_n$ to quickly produce the n block model for the new window. It then uses offline processing time to add B_{n+1} to all of the other models it is maintaining. Wang et al. (2003) develop a similar system for modeling drifting concepts by learning a model on each chunk of data as it arrives, and also learning a weight for each of its previously learned modes based on how they perform on the new data. Then, to classify a new example, they use a weighted average of all previous models. The simplicity of these systems is an advantage: they can easily be applied to existing learning algorithm. The disadvantages of these systems are the granularity at which they track concept-drift (in periodic blocks compared to every example for CVFDT) and the overhead required to store (and update) all of the required models.

8. Future Work

Building a scaled-up system using the techniques developed in this paper requires some significant programming. A more desirable solution would allow a user to describe the learning procedure in a high-level programming language, and let a tool automatically generate a scaled up learning program from this description. To our knowledge no such system exists for generically scaling up learning algorithms. There are, however, several such systems for learning the parameters of probabilistic models. These include BUGS (Thomas et al., 1992), The Bayes Net Toolbox (Murphy, 2001), and AutoBayes (Fischer and Schumann, 2002). In order to develop similar solutions for our domain we require: an expressive language for specifying model structure; a structural search control language capable of expressing parallelism, simple communication among parallel searches, and easily changing search strategies (i.e. hill climbing, look ahead, etc.); and a catalog of model evaluation functions with associated high confidence bounds.

Continuous search spaces are another area for future work. We have begun to explore these with our work on VFKM (Domingos and Hulten, 2001) and VFEM (Domingos and Hulten, 2002) clustering algorithms, but more work and closer integration with structural search is needed. The most critical items here are to develop tighter bounds for continuous spaces, to experiment with heuristics based on our existing bounds, and to come up with conceptually simple ways to simultaneously perform structural learning and parameter learning from data stream (i.e. how should time and data be divided between the two tasks, can the combined system maintain the desirable properties of our structural search framework, etc.).

Previously, our approach has been to take existing learning algorithms and scale them up using the techniques developed in our method. It seems likely, however, that the algorithms, models, and heuristics that work well for small data sets will not be the best match to the requirements found with very large data sets or data streams. Thus, another interesting area for future work is to use the scaling techniques developed in this paper (as well as those developed by many other researchers) to design new learning algorithms for new situations.

We made some small steps in this direction with the VFBN2 system (Hulten and Domingos, 2002) which proposes a new search method for learning the structure of Bayesian networks and with the VFREL system (Hulten et al., 2003) for learning from massive relational databases, but we believe there is a great deal more work to be done.

9. Conclusion

In this paper we developed a general method for semi-automatically scaling up a wide class of learning algorithms to learn from massive time-changing data streams. The core of our method uses sampling in a fine-grained manner to decide exactly how much data to use for each decision that needs to be made during the course of the learning run. Our method also contains many extensions that allow it to meet the requirements present in the real world such as working within limited RAM, adapting to time-changing data, and producing high-quality results very quickly. We evaluated our method by using it to implement the VFDT system for learning decision trees from high-speed data streams and extending it to develop the CVFDT system for keeping trees up-to-date with time-changing data streams. We evaluated these systems with extensive studies on synthetic data sets, and with an application to a massive real-world data stream. We found that the systems we implemented using our method were able to outperform traditional systems by running faster and by taking advantage of additional data to learn higher quality models. Finally, we have released the tools we developed so that others may apply them in their own settings (see Hulten and Domingos (2004)).

Acknowledgments

This research was partly supported by a gift from the Ford Motor Company, and by NSF CAREER and IBM Faculty awards to the second author.

References

R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. In Nick Cercone and Mas Tsuchiya, editors, *Special Issue on Learning and Discovery in Knowledge-Based Databases*, number 5(6), pages 914–925. Institute of Electrical and Electronics Engineers, Washington, U.S.A., 1993.

R. Agrawal and G. Psaila. Active data mining. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pages 3–8, Montréal, Canada, 1995. AAAI Press.

Khaled Alsabti, Sanjay Ranka, and Vineet Singh. CLOUDS: A decision tree classifier for large datasets. In *Knowledge Discovery and Data Mining*, pages 2–8, 1998.

P. L. Bartlett, S. Ben-David, and S. R. Kulkarni. Learning changing concepts by exploiting the structure of change. *Machine Learning*, 41:153–174, 2000.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont, CA, 1984.

J. Catlett. *Megainduction: Machine Learning on Very Large Databases*. PhD thesis, Basser Department of Computer Science, University of Sydney, Sydney, Australia, 1991.

S. Chakrabarti, S. Sarawagi, and B. Dom. Mining surprising patterns using temporal description length. In *Proceedings of the Twenty-Fourth International Conference on Very Large Data Bases*, pages 606–617, New York, NY, 1998. Morgan Kaufmann.

D. Cohn, L. Atlas, and R. Ladner. Improving generalization with active learning. *Machine Learning*, 15:201–221, 1994.

A. Danyluk, T. Fawcett, and F. Provost, editors. *Proceedings of the AAAI-98 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis*. AAAI Press, Madison, WI, 1998.

T. G. Dietterich. Overfitting and undercomputing in machine learning. *Computing Surveys*, 27:326–327, 1995.

C. Domingo, R. Gavalda, and O. Watanabe. Adaptive sampling methods for scaling up knowledge discovery algorithms. *Data Mining and Knowledge Discovery*, 6:131–152, 2002.

P. Domingos. A process-oriented heuristic for model selection. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 127–135, Madison, WI, 1998. Morgan Kaufmann.

P. Domingos and G. Hulten. A general method for scaling up machine learning algorithms and its application to clustering. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 106–113, Williamstown, MA, 2001. Morgan Kaufmann.

P. Domingos and G. Hulten. Learning from infinite data in finite time. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, pages 673–680. MIT Press, Cambridge, MA, 2002.

T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 53–62, San Diego, CA, 1999. ACM Press.

B. Fischer and J. Schumann. Generating data analysis programs from statistical models. *Journal of Functional Programming*, 12, 2002.

Y. Freund. Self bounding learning algorithms. In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, Madison, WI, 1998. Morgan Kaufmann.

V. Ganti, J. Gehrke, and R. Ramakrishnan. DEMON: Mining and monitoring evolving data. In *Proceedings of the Sixteenth International Conference on Data Engineering*, pages 439–448, San Diego, CA, 2000.

J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-L. Loh. BOAT: optimistic decision tree construction. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 169–180, Philadelphia, PA, 1999. ACM Press.

Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. Rainforest - a framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2/3):127–162, 2000.

J. Gratch. Sequential inductive learning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 779–786, Portland, OR, 1996. AAAI Press.

R. Greiner. PALO: A probabilistic hill-climbing algorithm. *Artificial Intelligence*, 84:177–208, 1996.

W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

G. Hulten and P. Domingos. Mining complex models from arbitrarily large databases in constant time. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 525–531, Edmonton, Canada, 2002. ACM Press.

G. Hulten and P. Domingos. VFML: A toolkit for mining large data streams. http://www.cs.washington.edu/dm/vfml/. 2004.

G. Hulten, P. Domingos, and Y. Abe. Mining massive relational databases. In *IJCAI 2003 Workshop on Learning Statistical Models from Relational Data*, Acapulco, Mexico, 2003.

D. Jensen and P. R. Cohen. Multiple comparisons in induction algorithms. *Machine Learning*, 38:309–338, 2000.

G. John and P. Langley. Static versus dynamic sampling for data mining. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 367–370, Portland, OR, 1996. AAAI Press.

M. G. Kelly, D. J. Hand, and N. M. Adams. The impact of changing populations on classifier performance. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 367–371, San Diego, CA, 1999. ACM Press.

P. Komarek and A. Moore. A dynamic adaptation of AD-trees for efficient machine learning on large data sets. In *Proc. 17th International Conf. on Machine Learning*, pages 495–502. Morgan Kaufmann, San Francisco, CA, 2000.

M. Kubat and G. Widmer, editors. *Proceedings of the ICML-96 Workshop on Learning in Context-Sensitive Domains.* Bari, Italy, 1996.

P. M. Long. The complexity of learning according to two models of a drifting environment. *Machine Learning*, 37:337–354, 1999.

O. Maron and A. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, San Mateo, CA, 1994.

C. Meek, B. Thiesson, and D. Heckerman. The learning-curve method applied to model-based clustering. *Journal of Machine Learning Research*, 2:397–418, 2002.

M. Mehta, A. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Extending Database Technology*, pages 18–32, Avignon, France, 1996. Springer.

K. Murphy. The Bayes net toolbox for Matlab. *Computing Science and Statistics*, 33, 2001.

R. Musick, J. Catlett, and S. Russell. Decision theoretic subsampling for induction on large databases. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 212–219, Amherst, MA, 1993. Morgan Kaufmann.

F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 23–32, San Diego, CA, 1999. ACM Press.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

J. R. Quinlan and R. M. Cameron-Jones. Oversearching and layered search in empirical learning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1019–1024, Montréal, Canada, 1995. Morgan Kaufmann.

M. Salganicoff. Density-adaptive learning and forgetting. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 276–283, Amherst, MA, 1993. Morgan Kaufmann.

T. Scheffer and S. Wrobel. Incremental maximization of non-instance-averaging utility functions with applications to knowledge discovery problems. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 481–488, Williamstown, MA, 2001. Morgan Kaufmann.

J. C. Schlimmer and R. H. Granger, Jr. Beyond incremental processing: Tracking concept drift. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 502–507, Philadelphia, PA, 1986. Morgan Kaufmann.

J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the Twenty-Second International Conference on Very Large Databases*, pages 544–555, Bombay, India, 1996. Morgan Kaufmann.

J. Shawe-Taylor, P. L. Bartlett, R. C. Williamson, and M. Anthony. Structural risk minimization over data-dependent hierarchies. Technical Report NC-TR-96-053, Department of Computer Science, Royal Holloway, University of London, Egham, UK, 1996.

A. Thomas, D. J. Spiegelhalter, and W. R. Gilks. BUGS: A program to perform bayesian inference using gibbs sampling. *Bayesian statistics*, 4:837–842, 1992.

P. Turney. Context-sensitive learning bibliography. Online bibliography, Institute for Information Technology of the National Research Council of Canada, Ottawa, Canada, 1998. http://ai.iit.nrc.ca/bibliographies/context-sensitive.html.

P. E. Utgoff. An improved algorithm for incremental induction of decision trees. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 318–325, New Brunswick, NJ, 1994. Morgan Kaufmann.

A. Wald. *Sequential Analysis*. Wiley, New York, NY, 1947.

H. Wang, W. Fan, P. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Washington D.C., 2003.

G. I. Webb. OPUS: An efficient admissible algorithm for unordered search. *Journal of Artificial Intelligence Research*, 3:431–465, 1995.

G. Widmer and M. Kubat (eds.). Special issue on context sensitivity and concept drift. *Machine Learning*, 32(2), 1998.

G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23:69–101, 1996.

A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of Web-object sharing and caching. In *Proceedings of the Second USENIX Conference on Internet Technologies and Systems*, pages 25–36, Boulder, CO, 1999.