Mining Complex Models from Arbitrarily Large Databases in Constant Time

Geoff Hulten
Dept. of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, U.S.A.
ghulten@cs.washington.edu

Pedro Domingos
Dept. of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, U.S.A.
pedrod@cs.washington.edu

ABSTRACT

In this paper we propose a scaling-up method that is applicable to essentially any induction algorithm based on discrete search. The result of applying the method to an algorithm is that its running time becomes independent of the size of the database, while the decisions made are essentially identical to those that would be made given infinite data. The method works within pre-specified memory limits and, as long as the data is iid, only requires accessing it sequentially. It gives anytime results, and can be used to produce batch, stream, time-changing and active-learning versions of an algorithm. We apply the method to learning Bayesian networks, developing an algorithm that is faster than previous ones by orders of magnitude, while achieving essentially the same predictive performance. We observe these gains on a series of large databases generated from benchmark networks, on the KDD Cup 2000 e-commerce data, and on a Web log containing 100 million requests.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*; I.2.6 [**Artificial Intelligence**]: Learning—*induction*; I.5.1 [**Pattern Recognition**]: Models—*statistical*; I.5.2 [**Pattern Recognition**]: Design Methodology—*classifier design and evaluation*

General Terms

Scalable learning algorithms, subsampling, Hoeffding bounds, discrete search, Bayesian networks

1. INTRODUCTION

Much work in KDD has focused on scaling machine learning and statistical algorithms to large databases. The goal is generally to obtain algorithms whose running time is linear (or near-linear) in the size of the database, and that only access the data sequentially. So far this has been done mainly for one algorithm at a time, in a slow and laborious process. We believe that this state of affairs can be overcome by developing scaling methods that are automatically (or nearly automatically) applicable to broad classes of learning algorithms, and that scale up to databases of arbitrary size by limiting the quantity of data used at each step, while guaranteeing that the decisions made do not differ significantly from those that would be made given infinite data. This paper describes one such method, based on generalizing the ideas initially proposed in our VFDT algorithm for scaling up decision tree induction [3]. The method is applicable to essentially any learning algorithm based on discrete search, where at each search step a number of candidate models or model components are considered, and the best one or ones are selected based on their performance on an iid sample from the domain of interest. Search types it is applicable to include 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, clustering, and probability estimation in discrete spaces, etc., as well as their combinations.

We demonstrate the method's utility by using it to scale up Bayesian network learning [8]. Bayesian networks are a powerful method for representing the joint distribution of a set of variables, but learning their structure and parameters from data is a computationally costly process, and to our knowledge has not previously been successfully attempted on databases of more than tens of thousands of examples. With our method, we have been able to mine millions of examples per minute. We demonstrate the scalibility and predictive performance of our algorithms on a set of benchmark networks and on three large Web data sets.

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 database of iid (independent and identically distributed) examples. We wish to determine which of the two classifiers is more accurate on the database. 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 database, 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* [9]. 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 $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}$. Let $\overline{r}$ be the difference in accuracy between the two classifiers on the first n examples in the database, 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 highest accuracy on those n examples is also the most accurate one on the entire infinite sample. 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. However, in this case we do not care which classifier wins. 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 database.

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 $f(\delta, n)$ that returns the maximum ϵ by which the true mean of r is smaller than the sample mean $\overline{r}$, given a desired confidence $1-\delta$ and sample size n.

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 $f(\delta, n)$ 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 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 $\epsilon_i = f(n_i, \delta^*/[da(b-a)]) < r_i$, where r_i is the difference in accuracy between the ath 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 j at step i. Instead, we can scan Δn examples from the database or data stream at a time, and check whether the goal has been met. Let $f^{-1}(\epsilon, \delta)$ 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 $c = \lceil f^{-1}(\tau, \delta)/\Delta n \rceil$ 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 $\epsilon_i = f(n_i, \delta^*/[cda(b-a)]) < r_i$.

Notice that nothing in the treatment above requires that the models being compared be classifiers, or that accuracy be the evaluation function. 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. The only property required of the evaluation function is that it be decomposable into an average (or sum) over training examples. This leads to the general method for scaling up learning algorithms shown in Table 1.

The key properties of this method are summarized in the following theorem (A proof can be found in [10]). Let t_{Gen} be the total time spent by L^* generating candidates, and let t_{Sel} be the total time spent 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 1).

THEOREM 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 $1-\delta^*$.*

In other words, the running time of the modified algorithm is independent of the size of the database, as long as the latter is greater than $f^{-1}(\tau, \delta^*/[cda(b-a)])$. A tie between candidates at some search step occurs if τ is reached or the database 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

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

[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 $E(M, x)$, where
        $M$ is a model (or model component) and $x$ is
        an example.
    A learning algorithm $L$ that finds a model by per-
        forming at most $d$ search steps, at each step
        considering at most $b$ candidates and selecting
        the $a$ with highest $\overline{E}(M,T) = \frac{1}{|T|}\sum_{X \in T} E(M,X)$.
    A desired maximum error probability $\delta^*$.
    A threshold of indifference $\tau$.
    A bound function $f(n, \delta)$.
    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.
```

```
Procedure SelectCandidates($\mathcal{M}$)
Let $n = 0$.
For each $M \in \mathcal{M}$, let $\Sigma(M) = 0$.
Repeat
    If $n + \Delta n > |T|$ then let $n' = |T|$.
    Else let $n' = n + \Delta n$.
    For each $M \in \mathcal{M}$
        Let $\Sigma(M) = \Sigma(M) + \sum_{i=n+1}^{n'} E(M, X_i)$.
        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 $\mathcal{M}\}$.
    $n = n'$.
    $c = \lceil f^{-1}(\tau, \delta^*/[cda(b-a)])/\Delta n \rceil$.
Until $[\forall(M_i \in \mathcal{M}', M_j \in \mathcal{M} - \mathcal{M}')$
        $f(n, \delta^*/[cda(b-a)]) < \overline{E}(M_i,T') - \overline{E}(M_j,T')]$
    or $f(n, \delta^*/[cda(b-a)]) < \tau$ or $n = |T|$.
Return $\mathcal{M}'$.
```

of the search steps, the model produced is, with probability $1 - \delta^*$, the same that would be produced with infinite data.

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, $\delta^* = \delta \sum_{i=1}^{d} c_i a_i (b_i - a_i)$, where c_i is the number of goal checks performed in step i, b_i is the number of candidates considered at that step, and a_i is the number selected. Even if δ is computed from c, b, a and the desired δ^* as in Table 1, reporting the actual achieved δ^* is recommended, since it will often be much better than the original target.

Further time can be saved by, at each step, dropping a candidate from consideration as soon as its score is lower than at least a others by at least $f(n, \delta^*/[cda(b-a)])$.

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 in Table 1 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. 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 learning large, complex models 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 due to repeated swapping of memory pages. 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 and ties are broken consistently, these winners are guaranteed to be the same that would be obtained by the single call *SelectCandidates*($\mathcal{M}$). 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.

The running time of an algorithm scaled up with our method is often dominated by the time spent reading data from disk. When a subset of the search steps are independent (i.e., the results of one step are not needed to perform the next one, as when growing the different nodes on the fringe of a decision tree), much time can be saved by using a separate search for each independent model component and (conceptually) performing them in parallel. This means that each data block needs to be read only once for all of the interleaved searches, greatly reducing I/O requirements. When these searches do not all fit simultaneously into memory, we maintain an inactive queue from which steps are transferred to memory as it becomes available (because some other search completes or is inactivated).

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. A common solution to this is to repeatedly apply the learner to a sliding window of examples, which can be very inefficient. Our method can be adapted to efficiently accommodate time-changing data as follows. Maintain $\Sigma(M)$ throughout time for every candidate M considered at every search step. After the first w examples, where w is the window width, subtract the oldest example from $\Sigma(M)$ whenever a new one is added. After every Δn new examples, determine again the best a candidates at every previous search decision point. If one of them is better than an old winner by δ^*/s (s is the maximum number of candidate comparisons expected during the entire run) then there has probably been some concept drift. In these cases, begin an alternate search starting from the new winners. Periodically use a number of new examples as a validation set to compare the performance of the models produced by the new and old searches. Prune the old search when the new models are on average better than the old ones, and 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, prune the lowest-performing ones. This approach to handling time-changing data is a generalization of the one we successfully applied to the VFDT decision-tree induction algorithm [11].

Active learning is a powerful type of subsampling where the learner actively selects the examples that would cause the most progress [1]. Our method has a natural extension to this case when different examples are relevant to different search steps, and some subset of these steps is independent of each other (as in decision tree induction, for example): choose the next Δn examples to be relevant to the step where U is currently farthest from being achieved (i.e., where $f(n, \delta^*/[cda(b-a)]) - \min_{ij}\{\max\{\tau, \overline{E}(M_i, T') - \overline{E}(M_j, T')\}\}$ is highest).

In the remainder of this paper we apply our method to learning Bayesian networks, and evaluate the performance of the resulting algorithms.

3. LEARNING BAYESIAN NETWORKS

We now briefly introduce Bayesian networks and methods for learning them. See Heckerman et al. [8] for a more complete treatment. A Bayesian network encodes the joint probability distribution of a set of d variables, $\{x_1, \ldots, x_d\}$, as a directed acyclic graph and a set of conditional probability tables (CPTs). (In this paper we assume all variables are discrete.) Each node corresponds to a variable, and the CPT associated with it contains the probability of each state of the variable given every possible combination of states of its parents. The set of parents of x_i, denoted $par(x_i)$, is the set of nodes with an arc to x_i in the graph. The structure of the network encodes the assertion that each node is conditionally independent of its non-descendants given its parents. Thus the probability of an arbitrary event $X = (x_1, \ldots, x_d)$ can be computed as $P(X) = \prod_{i=1}^{d} P(x_i|par(x_i))$. In general, encoding the joint distribution of a set of d discrete variables requires space exponential in d; Bayesian networks reduce this to space exponential in $\max_{i\in\{1,\ldots,d\}} |par(x_i)|$.

In this paper we consider learning the structure of Bayesian networks when no values are missing from training data. A number of algorithms for this have been proposed; perhaps the most widely used one is described by Heckerman et al. [8]. It performs a search over the space of network structures, starting from an initial network which may be random, empty, or derived from prior knowledge. At each step, the algorithm generates all variations of the current network that can be obtained by adding, deleting or reversing a single arc, without creating cycles, and selects the best one using the *Bayesian Dirichlet (BD)* score (see Heckerman et al. [8]). The search ends when no variation achieves a higher score, at which point the current network is returned. This algorithm is commonly accelerated by caching the many redundant calculations that arise when the BD score is applied to a collection of similar networks. This is possible because the BD score is *decomposable* into a separate component for each variable. In the remainder of this paper we scale up Heckerman et al.'s algorithm, which we will refer to as HGC throughout.

4. SCALING UP BAYESIAN NETWORKS

At each search step, HGC considers all examples in the training set T when computing the BD score of each candidate structure. Thus its running time grows without limit as the training set size $|T|$ increases. By applying our method, HGC's running time can be made independent of $|T|$, for $|T| > f^{-1}(\tau, \delta)$, with user-determined τ and δ. In order to do this, we must first decompose the BD score into an average of some quantity over the training sample. This is made possible by taking the logarithm of the BD score, and discarding terms that become insignificant when $n \to \infty$, because the goal is to make the same decisions that would be made with infinite data. This yields (see Hulten & Domingos [10] for the detailed derivation):

$$\log BD_\infty(S,T) = \sum_{e=1}^{\infty}\sum_{i=1}^{d} \log \hat{P}(x_{ie}|S, par(x_{ie})) \qquad (1)$$

where x_{ie} is the value of the ith variable in the eth example, and $\hat{P}(x_{ie}|par(x_{ie}))$ is the maximum-likelihood estimate of the probability of x_{ie} given its parents in structure S, equal to n_{ijk}/n_{ij} if in example e the variable is in its kth state and its parents are in their jth state. Effectively, when $n \to \infty$, the log-BD score converges to the log-likelihood of the data given the network structure and maximum-likelihood parameter estimates, and choosing the network with highest BD score becomes equivalent to choosing the maximum-likelihood network. The quantity $\sum_{i=1}^{d} \log \hat{P}(x_{ie}|S, par(x_{ie}))$ is the log-likelihood of an example X_e given the structure S and corresponding parameter estimates. When comparing two candidate structures S_1 and S_2, we compute the mean difference in this quantity between them:

$$\begin{aligned}\Delta L(S_1, S_2) \;=\;& \frac{1}{n}\sum_{e=1}^{n}\sum_{i=1}^{d} \log \hat{P}(x_i|S_1, par(x_i)) \\ &- \frac{1}{n}\sum_{e=1}^{n}\sum_{i=1}^{d} \log \hat{P}(x_i|S_2, par(x_i)) \qquad (2)\end{aligned}$$

Notice that the decomposability of the BD score allows this computation to be accelerated by only considering the components corresponding to the two or four variables with different parents in S_1 and S_2. We can apply either the normal bound or the Hoeffding bound to $\Delta L(S_1, S_2)$. In order to apply the Hoeffding bound, the quantity being averaged must have a finite range. We estimate this range by measuring the minimum non-zero probability at each node P_0^i and use as the range $\sum_{i=1}^{d} log \frac{c}{P_0^i}$, where c is a small integer.[3]

After the structure is learned, the final n_{ijk} and n_{ij} counts must be estimated. In future work we will use the bound from [6] to determine the needed sample size; the current algorithm simply uses a single pass over training data. Together with the parameters of the Dirichlet prior, these counts induce a posterior distribution over the parameters of the network. Prediction of the log-likelihood of new examples is

[3]A value may have zero true probability given some parent state, but this is not a problem, since such a value and parent combination will never occur in the data.

carried out by integrating over this distribution, as in HGC. We call this algorithm VFBN1.

HGC and VFBN1 share a major limitation: they are unable to learn Bayesian networks in domains with more than 100 variables or so. The reason is that the space and time complexity of their search increases quadratically with the number of variables (since, at each search step, each of the d variables considers on the order of one change with respect to each of the other d variables). This complexity can be greatly reduced by noticing that, because of the decomposability of the BD score, many of the alternatives considered in a search step are independent of each other. That is, except for avoiding cycles, the best arc change for a particular variable will still be best after changes are made to other variables. The VFBN2 algorithm exploits this by carrying out a separate search for each variable in the domain, interleaving all the searches. VFBN2 takes advantage of our method's ability to reduce I/O time by reading a data block just once for all of its searches. The generality of our scaling-up method is illustrated by the fact that it is applied in VFBN2 as easily as it was in VFBN1.

Two issues prevent VFBN2 from treating these searches completely independently: arc reversal and the constraint that a Bayesian network contain no cycles. VFBN2 avoids the first problem simply by disallowing arc reversal; our experiments show this is not detrimental to its performance. Cycles are avoided in a greedy manner by, whenever a new arc is added, removing from consideration in all other searches all alternatives that would form a cycle with the new arc.

VFBN2 uses our method's ability to cycle through searches when they do not all simultaneously fit into memory. All searches are initially on a queue of inactive searches. An inactive search uses only a small constant amount of memory to hold its current state. The main loop of VFBN2 transfers searches from the head of the inactive queue to the active set until memory is filled. An active search for variable x_i considers removing the arc between each variable in $par(x_i)$ and x_i, adding an arc to x_i from each variable not already in $par(x_i)$, and making no change to $par(x_i)$, all under the constraint that no change add a cycle to S. Each time a block of data is read, the candidates' scores in each active search are updated and a winner is tested for as in *Select-Candidates*() (see Table 1). If there is a winner (or one is selected by tie-breaking because τ has been reached or T has been exhausted) its change is applied to the network structure. Candidates in other searches that would create cycles if added to the updated network structure are removed from consideration. A search is finished if making no change to the associated variable's parents is better than any of the alternatives. If a search takes a step and is not finished, it is appended to the back of the inactive queue, and will be reactivated when memory is available. VFBN2 terminates when all of its searches finish.

HGC and VFBN1 require $O(d^2v^k)$ memory, where k is the maximum number of parents and v is the maximum number of values of any variable in the domain. This is used to store the CPTs that differ between each alternative structure and S. VFBN2 improves on this by using an inactive queue to temporarily deactivate searches when RAM is short. This gives VFBN2 the ability to make progress with $O(dv^k)$ memory, turning a quadratic dependence on d into a linear one. VFBN2 is thus able to learn on domains with many more variables than HGC or VFBN1. Further, HGC and VFBN1 perform redundant work by repeatedly evaluating $O(d^2)$ alternatives, selecting the best one, and discarding the rest. This is wasteful because some of the discarded alternatives are independent of the one chosen, and will be selected as winners at a later step. VFBN2 avoids much of this redundant work and learns structure up to a factor of d faster than VFBN1 and HGC.

We experimented with several policies for breaking ties, and obtained the best results by selecting the alternative with highest observed BD score. First, it limits the number of times an arc can be added or removed between each pair of variables to two. Second, to make sure that at least one search will fit in the active set, it does not consider any alternative that requires more than M/d MB of memory, where M is the system's total available memory in MB. Third, it can limit the number of parameters used in any variable's CPT to be less than a user-supplied threshold.

5. EMPIRICAL EVALUATION

We compared VFBN1, VFBN2, and our implementation of HGC on data generated from several benchmark networks, and on three massive Web data sets. All experiments were carried out on a cluster of 1 GHz Pentium III machines, with a memory limit M of 200MB for the benchmark networks and 300MB for the Web domains. In order to stay within these memory limits, VFBN1 and HGC discarded any search alternative that required more than M/d^2 MB of RAM. The block size Δn was set to 10,000. VFBN1 and VFBN2 used the normal bound with variance estimated from training data. To control complexity we limited the size of each CPT to 10,000 parameters (one parameter for every 500 training samples). VFBN1 and VFBN2 used $\delta = 10^{-9}$ and $\tau = 0.05\%$. All algorithms started their search from empty networks (we also experimented with prior networks, but space precludes reporting on them).

Benchmark Networks For the benchmark study, we generated data sets of five million examples each from the networks contained in the repository at http://www.cs.huji.ac.-il/labs/compbio/Repository/. The networks used and number of variables they contained were: (Insurance, 27), (Water, 32), (Alarm, 37), (Hailfinder, 56), (Munin1, 189), (Pigs, 441), (Link, 724), and (Munin4, 1041). Predictive performance was measured by the log-likelihood on 100,000 independently generated test examples. HGC's parameters were set as described in Hulten & Domingos [10]. We limited the algorithms to spend at most 5 days of CPU time (7200 minutes) learning structure, after which the best structure found up to that point was returned.

Table 2 contains the results of our experiments. All three systems were able to run on the small networks (Insurance, Water, Alarm, Hailfinder). Because of RAM limitations only VFBN2 could run on the remaining benchmark networks. The systems achieve approximately the same likelihoods for the small networks. Further, their likelihoods were very close to those of the true networks, indicating that our scaling method can achieve high quality models. VFBN1 and VFBN2 both completed all four runs within the allotted 5 days, while HGC did so only twice. VFBN2 was an order of magnitude faster than VFBN1, which was an order of magnitude faster than HGC. VFBN2 spent less than five minutes on each run. This was less than the time it spent doing a single scan of data to estimate parameters, by a factor of five to ten. VFBN2's global confidence bounds δ^* were better than VFBN1's by at least an order of magnitude in each experiment. This was caused by VFBN1's redundant search forcing it to remake many decisions, thus requiring that many more statistical bounds hold. We also ran HGC on a random sample of 10,000 training examples. This variation always had worse likelihood than both VFBN1 and VFBN2. It also always spent more time learning structure than did VFBN2.

For the large networks, we found VFBN2 to improve significantly on the initial networks, and to learn networks with likelihoods similar to those of the true networks. Not surprisingly, we found that many more changes were required to learn large networks than to learn small ones (on these, VFBN2 made between 566 and 7133 changes to the prior networks). Since HGC and VFBN1 require one search step for each change, this suggests that even with sufficient RAM they would learn much more slowly compared to VFBN2 than they did on the small networks. We watched the active set and inactive queue during the runs on the large data sets. We found the proportion of searches that were active was high near the beginning of each run. As time progressed, however, the size of the CPTs being learned tended to increase, leaving less room for searches to be active. In fact, during the majority of the large network runs, only a small fraction of the remaining searches were active at any one time. Recall that VFBN1 and HGC can only run when all remaining searches fit in the active set. For these networks the 200 MB allocation falls far short. For example, on a typical run on a large network we found that at the worst point only 1.31% of the remaining searches were active. Assuming they all take about the same RAM, HGC and VFBN1 would have required nearly 15 GB to run.

Web Applications In order to evaluate VFBN2's performance on large real-world problems, we ran it on two large Web traces. The first was the data set used in the KDD Cup 2000 competition [12]. The second was a trace of all requests made to the Web site of the University of Washington's Department of Computer Science and Engineering between January 2000 and January 2002.

The KDD Cup data consists of 777,000 Web page requests collected from an e-commerce site. Each request is annotated with a requester session ID and a large collection of attributes describing the product in the requested page. We focused on one of the fields in the log, "Assortment Level 4", which contained 65 categorizations of pages into product types (including "none"). For each session we produced a training example with one Boolean attribute per category – "True" if the session visited a page with that category. There were 235,000 sessions in the log, of which we held out 35,000 for testing.

The UW-CSE-80 and UW-CSE-400 data sets were created from a log of every request made to our department's Web site between late January 2000 and late January 2002. The log contained on the order of one hundred million requests. We extracted the two training sets from this log in a manner very similar the KDD Cup data set. For the first we identified the 80 most commonly visited level two directories on the site (e.g. */homes/faculty/* and */education/undergrads/*). For the second we identified the 400 most commonly visited Web objects (excluding most images, style sheets, and scripts). In both cases we broke the log into approximate sessions, with each session containing all the requests made by a single host until an idle period of 10 minutes; there were 8.3 million sessions. We held out the last week of data for testing.

Table 2: Empirical results. Samples is the total number of examples read from disk while learning structure, in millions. Times in bold exceeded our five day limit and the corresponding runs were stopped before converging.

Network	Algorithm	Log-Likel	Samples	Minutes
Insurance	True	−13.048	–	–
	HGC	−13.048	320.00	2446.08
	VFBN1	−13.069	16.69	39.72
	VFBN2	−13.070	0.52	1.02
Water	True	−12.781	–	–
	HGC	−12.783	375.00	5897.53
	VFBN1	−12.805	35.80	360.45
	VFBN2	−12.795	0.88	1.85
Alarm	True	−10.448	–	–
	HGC	−10.455	279.93	**7200.15**
	VFBN1	−10.439	15.07	87.87
	VFBN2	−10.447	0.81	2.92
Hailfinder	True	−49.115	–	–
	HGC	−54.403	123.12	**7200.62**
	VFBN1	−48.885	23.80	194.97
	VFBN2	−48.889	0.17	3.22
Munin1	True	−37.849	–	–
	VFBN2	−38.417	0.95	47.38
Pigs	True	−330.277	–	–
	VFBN2	−319.559	1.24	636.98
Link	True	−210.153	–	–
	VFBN2	−232.449	12.29	2451.95
Munin4	True	−171.874	–	–
	VFBN2	−173.757	4.61	3003.28
KDD-Cup	Empty	−2.446	–	0.00
	HGC	−2.282	69.59	6345.13
	VFBN1	−2.272	12.05	335.37
	VFBN2	−2.301	0.42	15.40
UW-80	Empty	−1.611	–	0.00
	HGC	−1.346	75.53	**7201.05**
	VFBN1	−1.273	14.93	457.18
	VFBN2	−1.269	1.98	12.08
UW-400	Empty	−7.002	–	0.00
	VFBN2	−4.556	3.63	905.30

The UW-CSE-400 domain was too large for VFBN1 or HGC. HGC ran for nearly five days on the other two data sets, while VFBN2 took less than 20 minutes for each. The systems achieved similar likelihoods when they could run, and always improved on their starting network. Examining the networks produced, we found many potentially interesting patterns. For example, in the e-commerce domain we found products that were much more likely to be visited when a combination of related products was visited than when only one of those products was visited.

6. RELATED WORK

Our method falls in the general category of sequential analysis [17], which determines at runtime the number of examples needed to satisfy a given quality criterion. Other recent examples of this approach include Maron and Moore's racing algorithm for model selection [13], Greiner's PALO algorithm for probabilistic hill-climbing [7], Scheffer and Wrobel's sequential sampling algorithm [16], and Domingo et al.'s AdaSelect algorithm [2]. Our method goes beyond these in applying to any type of discrete search, providing new formal results, working within pre-specified memory limits, supporting interleaving of search steps, learning from time-changing data, etc. A related approach is progressive sampling [14, 15], where successively larger samples are tried, a learning curve is fit to the results, and this curve is used to decide when to stop. This may lead to stopping earlier than with our method, but stopping can also occur prematurely, due to the difficulty in reliably extrapolating learning curves.

Friedman et al.'s Sparse Candidate algorithm [5] alternates between heuristically selecting a small group of potential relatives for each variable and doing a search step limited to considering changing arcs between a variable and its potential relatives. This procedure avoids the quadratic dependency on the number of variables in a domain. Friedman et al. evaluated it on data sets containing 10,000 samples and networks with up to 800 variables.

This paper describes a general method for scaling up learners based on discrete search. We have also developed a related method for scaling up learners based on search in continuous spaces [4].

7. CONCLUSION AND FUTURE WORK

Scaling up learning algorithms to the massive data sets that are increasingly common is a fundamental challenge for KDD research. This paper proposes that the time used by a learning algorithm should depend only on the complexity of the model being learned, not on the size of the available training data. We present a framework for semi-automatically scaling any learning algorithm that performs a discrete search over model space to be able to learn from an arbitrarily large database in constant time. Our framework further allows transforming the algorithm to work incrementally, to give results anytime, to fit within memory constraints, to support interleaved search, to adjust to time-changing data, and to support active learning. We use our method to develop a new algorithm for learning large Bayesian networks from arbitrary amounts of data. Experiments show that this algorithm is orders of magnitude faster than previous ones, while learning models of essentially the same quality.

Directions for future work on our scaling framework include combining it with the one we have developed for search in continuous spaces, improving the bounds by taking candidate dependencies into account, constructing a programming library to facilitate our framework's application, and scaling up additional algorithms using it. Future work on VFBN includes extending it to handle missing data values, developing better mechanisms for controlling complexity when data is abundant, and scaling VFBN further by learning local structure at each node (e.g., in the form of a decision tree).

Acknowledgments We thank Blue Martini and Corin Anderson for providing the Web data sets and the donors and maintainers of the Bayesian network repository. This research was partly supported by NSF CAREER and IBM Faculty awards to the second author, and by a gift from the Ford Motor Co.

8. REFERENCES

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

[2] 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.

[3] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. 6th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pp. 71–80, Boston, MA, 2000.

[4] P. Domingos and G. Hulten. Learning from infinite data in finite time. In *Advances in Neural Information Processing Systems 14*. MIT Press, Cambridge, MA, 2002.

[5] N. Friedman, I. Nachman, and D. Peér. Learning Bayesian network structure from massive datasets: The "sparse candidate" algorithm. In *Proc. 15th Conf. on Uncertainty in Artificial Intelligence*, pp. 206–215, Stockholm, Sweden, 1999.

[6] N. Friedman and Z. Yakhini. On the sample complexity of learning Bayesian networks. In *Proc. 12th Conf. on Uncertainty in Artificial Intelligence*, pp. 274–282, Portland, OR, 1996.

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

[8] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

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

[10] G. Hulten and P. Domingos. A general method for scaling up learning algorithms and its application to Bayesian networks. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 2002.

[11] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. 7th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pp. 97–106, San Francisco, CA, 2001.

[12] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

[13] O. Maron and A. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in Neural Information Processing Systems 6*. Morgan Kaufmann, San Mateo, CA, 1994.

[14] 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.

[15] F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *Proc. 5th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pp. 23–32, San Diego, CA, 1999.

[16] T. Scheffer and S. Wrobel. Incremental maximization of non-instance-averaging utility functions with applications to knowledge discovery problems. In *Proc. 18th International Conf. on Machine Learning*, pp. 481–488, Williamstown, MA, 2001.

[17] A. Wald. *Sequential analysis*. Wiley, New York, 1947.