A Fast Way to Produce Near-Optimal Fixed-Depth Decision Trees

Alireza Farhangfar, Russell Greiner and **Martin Zinkevich**[*]
Dept of Computing Science
University of Alberta
Edmonton, Alberta T6G 2E8 Canada
{farhang, greiner, maz}@cs.ualberta.ca

Abstract

Decision trees play an essential role in many classification tasks. In some circumstances, we only want to consider *fixed-depth* trees. Unfortunately, finding the optimal depth-d decision tree can require time exponential in d. This paper presents a fast way to produce a fixed-depth decision tree that is optimal under the Naïve Bayes (NB) assumption. Here, we prove that the optimal depth-d feature essentially depends only on the posterior probability of the class label given the tests previously performed, but not directly on either the identity nor the outcomes of these tests. We can therefore precompute, in a fast pre-processing step, which features to use at the final layer. This results in a speedup of $O(n/\log n)$, where n is the number of features. We apply this technique to learning fixed-depth decision trees from standard UCI repository datasets, and find this model improves the computational cost significantly. Surprisingly, this approach still yields relatively high classification accuracy, despite the NB assumption.

1 Introduction

Many machine learning tasks involve producing a classification label for an instance. There is sometimes a fixed cost that the classifier can spend per instance, before returning a value; consider for example, per-patient capitation for medical diagnosis. If the tests can be performed sequentially, then the classifier may want to follow a *policy* (Sutton & Barto 1998) — *e.g.*, first perform a blood test, then if its outcome is positive, perform a liver test; otherwise perform an eye exam. This process continues until the funds are exhausted, at which point the classifier stops running tests and returns an outcome; either "healthy" or "sick".

Such policies correspond naturally to decision trees. There are many algorithms for learning a decision tree from a data sample; most systems (Quinlan 1993; Breiman *et al.* 1984) use some heuristics that greedily seek the decision tree that best fits the sample, before running a post-processor to reduce overfitting. This paper focuses on the challenge motivated above: of finding the best "fixed-cost policy", which corresponds to finding the fixed depth decision tree that best matches the data sample. There are several algorithms for this task, which typically use dynamic programming to find the "optimal" depth-d decision tree, *i.e.*, the tree that minimizes the 0/1-loss over the training data. These algorithms are invariably exponential in the depth d, and spend almost all of their time determining which features to test at final level, d (Auer, Holte, & Maass 1995). If one is willing to accept the "Naïve Bayes" assumption (Duda & Hart 1973) — that features are independent of each other given the class — then there is an efficient way to compute the final layer of depth-d features. In particular, under this assumption, we prove that the optimal depth-d feature essentially depends only on the *posterior probability of the class label given the tests previously performed*, but not the outcomes of the individual tests. We can therefore use a fast pre-processing step to create a so-called *opt-feature list*, OFL, that identifies which feature to use as a function of the posterior distribution, then use this list to quickly determine the last level of the tree. This technique results in a speedup of $O(n/\log n)$, where n is the number of features, and effectively means we can compute the optimal depth-d tree in the time typically required to compute the optimal depth-$(d-1)$ tree.

Section 2 surveys the relevant literature. Section 3 summarizing our OPTNBDT algorithm and proves the relevant theories. Section 4 presents empirical results that validates our approach, by applying it to the standard datasets from the UCI repository (Newman *et al.* 1998) and elsewhere. The webpage (Greiner 2007) provides other information about this system, and about our experiments. We find that our approach significantly improves the computational cost of finding a fixed depth tree, surprising at little or no loss in accuracy.

2 Literature Review

As noted above, there are many algorithms for learning decision trees. Many algorithms, including C4.5 (Quinlan 1993) and CART (Breiman *et al.* 1984), begin with a greedy method that incrementally identifies an appropriate feature to test at each point in the tree, based on some heuristic score. Then these algorithms perform a post-processing step to reduce overfitting. In our context of "shallow" decision trees, overfitting is not as big a concern, which explains why many of the algorithms that seek "fixed depth" decision trees simply return the tree that best fits the data (Holte 1993; Dobkin, Gunopoulos, & Kasif 1996; Auer, Holte, & Maass 1995). These algorithms can easily be extended to al-

[*]Now at Yahoo Research, Santa Clara

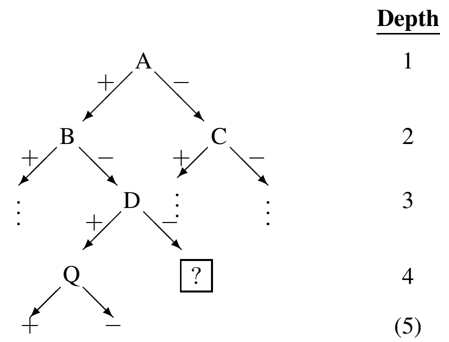

Figure 1: Decision Tree T_1, as it is being built.

low different tests to have different costs (Turney 2000; Greiner, Grove, & Roth 2002). In general, these algorithms require $O(n^d)$ time to find the best depth-d decision tree over n variables. While this is a polynomial-time complexity for fixed d, in practice it is not effective except for small n and tiny d. The results in this paper show that one can achieve a more efficient process by imposing a constraint — here the "Naïve Bayes" assumption (Duda & Hart 1973) — to speedup the process.

There is, of course, a large body of work on building classifiers based on Naïve Bayes systems (Domingos & Pazzani 1997; Lewis 1998), and on analyzing and characterizing their classification performance (Chan & Darwiche 2003). Those results, however, differ significantly from our task, which explicitly involves learning a *decision tree*; we use only the Naïve Bayes *assumption* in modeling the underlying distribution over instances. (But see the empirical comparisons in Section 4.)

Several other systems attempt to learn optimal decision trees. The GTO system (Bennett & Blue 1996) learns a decision tree by simultaneously optimizing a set of "disjunctive linear inequalities" — minimizing a nonlinear error function over a polyhedral region, whose extreme points contain the optimum solution. The DL8 system (Nijssen & Fromont 2007) uses a dynamic program to produce a decision tree that optimizes the "ranking function", which is based on constraints that involve various properties of the decision tree, including its training set error, its expected generalization error, its number of nodes, and its depth. By contrast, our goal is to quickly find a "near-optimal" tree, by seeking a tree that would have optimal classification performance, under the naïve bayes assumption.

Turney (2000) discusses the general challenge of learning the best classifier subject to some explicit cost constraint. Here, our "max depth" requirement corresponds to a constraint on the cost that the *classifier* must pay to see the features at *performance time*.

3 OptNbDt Algorithm

3.1 Foundations

Assume there are n features $\mathcal{F} = \{F^{(1)}, \ldots, F^{(n)}\}$ where each feature $F^{(j)}$ ranges over the $r_{F^{(j)}} \leq r$ values $\mathcal{V}_{F^{(j)}} = \{f_1^{(j)}, \ldots, f_{r_{F^{(j)}}}^{(j)}\}$, and there are two classes $\mathcal{C} = \{+, -\}$.[1] A "decision tree" T is a directed tree structure, where each internal node ν is labeled with a feature $\mathcal{F}(\nu) \in \mathcal{F}$, each leaf $\ell \in \mathcal{L}(T)$ is labeled with one of the classes $c(\ell) \in \mathcal{C}$, and each arc descending from a node labeled F is labeled with a value $f \in \mathcal{V}_F$. We can evaluate such a tree T on a specific instance $\mathbf{f} = \{F^{(j)} = f_i^{(j)}\}_j$, to produce a value $\text{VAL}(T, \mathbf{f}) \in \mathcal{C}$ as follows: If the tree is a leaf $T = \ell$, return its label $c(\ell) \in \mathcal{C}$; that is, $\text{VAL}(T, \mathbf{f}) = c(\ell)$. Otherwise, the root of the tree is labeled with a feature $F \in \mathcal{F}$. We then find the associated value within the $\mathbf{f}$ (say $F = f$), and follow the f-labeled edge from the root to a new subtree; then recur. The value of $\text{VAL}(T, f)$ will be the value of that subtree. Hence, given the instance $\mathbf{f} = \{A = +,\ B = -,\ \ldots\}$ and the tree T_1 in Figure 1, the value of $\text{VAL}(T_1, \mathbf{f})$ will be the value of the subtree rooted in the B-labeled node on this instance (as we followed the $A = +$ arc from the root), which in turn will be the value of the subsubtree rooted in the D-labeled node (corresponding to the subsequent $B = -$ arc), etc. We say a decision tree T is "correct" for a labeled instance $\langle\, \mathbf{f}, c \,\rangle$ if $\text{VAL}(T, \mathbf{f}) = c$. A labeled dataset is a set of labeled instances $S = \{\langle\, \mathbf{f}^{(j)}, c^{(j)} \,\rangle\}_j$.

Our goal is to find the depth-d decision tree with the maximum expected accuracy, given our posterior beliefs, which are constructed given a naïve bayes prior and the dataset S. To define expected accuracy, we must first define the notion of a path $\pi(\nu)$ to any node ν in the tree, which is the sequence of feature-value pairs leading from the root to that node; hence, the path to the $\boxed{?}$ node in Figure 1 is $\pi(\boxed{?}) = \langle\, \langle\, A, + \,\rangle,\ \langle\, B, - \,\rangle,\ \langle\, D, - \,\rangle\, \rangle$. We can use this notion to define the probability[2] of reaching a node, which here is $P(\text{"reaching}\boxed{?}\text{"}) = P(\,\pi(\boxed{?})\,) = P(\,A = +, B = -, D = -\,)$.

In general, the accuracy of any tree T is

$$\text{Acc}(\,T\,) \quad = \sum_{\ell \in \mathcal{L}(T)} P(\,\pi(\ell)\,) \times \text{Acc}(\,\pi(\ell)\,)$$

over the set of leaf nodes $\mathcal{L}(T)$. Note that this accuracy has been factored into a sum of the accuracies associated with each leaf: it is the probability $P(\,\pi(\ell)\,)$ of reaching each leaf node $\ell \in \mathcal{L}(T)$ times the (conditional) accuracy associated with that node $\text{Acc}_{\pi(\ell)}(\,\ell\,) = P(\,C = c(\ell) \,|\, \pi(\ell)\,)$. This factoring tells us immediately that we can decide on the appropriate class label for each leaf node, $c^*(\ell|\pi(\ell)\,) = \text{argmax}_c\, P(\,c \,|\, \pi(\ell)\,)$, with an associated accuracy of

$$\text{Acc}^*(\,\pi(\ell)\,) = \max_c P(\,c \,|\, \pi(\ell)\,) \tag{1}$$

based only on the single path; *i.e.*, it is independent of the rest of the tree.

[1]We use CAPITAL letters for variables and lowercase for values, and **bold** for sets of variables.

[2]All probability values, as well as Accuracy scores, are based on a posterior generated from a Naïve Bayes Prior and the labeled data S. Note also that we will use various forms of $\text{Acc.}(\,\cdot\,)$, for trees, paths and features appended to paths; here, the meaning should be clear from context.

We are searching for the "best" depth-d tree, $\text{argmax}_{T\in \text{DT}(d)} \text{Acc}(T, S)$, where $\text{DT}(d)$ is the set of decision trees of depth at most d — *i.e.*, each path from root to any leaf involves at most d variables. An earlier system (Kapoor & Greiner 2005) precomputed the accuracy associated with each possible sequence of d tests (requiring $O(\binom{n}{d} r^d)$ time), and then constructed the best tree given these values, which required $O((nr)^d)$ time. Here we present a different technique that requires less time by precomputing a data structure that quickly provides the optimal feature to use for each position in the bottom row.

To understand our approach, consider determining the feature $F(\nu)$ to use at the "final internal node" along a path $\pi(\nu)$ — *e.g.*, determine which feature to test at $\boxed{?}$ in Figure 1. As an obvious extension of the above argument, this decision will depend only on the path $\pi(\nu)$ to this node. Then for the $\boxed{?}$ node, it will depend on $\pi(\boxed{?}) = \langle \langle A, + \rangle, \langle B, - \rangle, \langle D, - \rangle \rangle$. Given our NB assumption, for any feature $F \in \mathcal{F}$ (except the features in π — *i.e.*, except for A, B, and D), the component of accuracy associated with the path that begins with π then performs F (and then descends to the leaf nodes immediately under this F), is

$$\begin{aligned}
&\text{Acc}(\,\pi \circ F\,)\\
&\quad= \sum_{f\in\mathcal{V}_F} P(\,\pi,\; F = f\,) \times \text{Acc}^*(\,\pi \circ\; F{=}f\,)\\
&\quad= \sum_{f\in\mathcal{V}_F} P(\,\pi,\; F{=}f\,) \times \max_c P(\,C{=}c\,|\,\pi,\; F{=}f\,)\\
&\quad= \sum_{f\in\mathcal{V}_F} \max_c P(\,C = c,\; \pi,\; F = f\,)\\
&\quad= \sum_{f\in\mathcal{V}_F} \max_c P(\,c\,)\, P(\,\pi\,|\,c\,) P(\,f\,|\,c\,) \qquad (2)\\
&\quad= P(\,\pi\,) \sum_{f\in\mathcal{V}_F} \max_c P(\,C{=}c\,|\,\pi\,) P(\,F{=}f\,|\,C{=}c\,)
\end{aligned}$$

where Equation 2 is based on the Naïve Bayes assumption. Note that the feature F that optimizes $\text{Acc}(\,\pi \circ F\,)$ will also optimize

$$\begin{aligned}
\text{Acc}_\pi(\,F\,) &= \frac{1}{P(\,\pi\,)} \text{Acc}(\,\pi \circ F\,)\\
&= \sum_{f\in\mathcal{V}_F} \max_c P(\,C{=}c\,|\,\pi\,) P(\,F{=}f\,|\,C{=}c\,).
\end{aligned}$$

While our analysis will work for classes that range over any (finite) number of values, this paper will focus on the binary case. Letting $x_{\pi,+} = P(\,C = +\,|\,\pi\,)$ and abbreviating "$F{=}f$" as "f", we have

$$\begin{aligned}
\text{Acc}_\pi(\,F\,) &= \sum_{f\in\mathcal{V}_F} \max \left\{ \begin{array}{c} x_{\pi,+}\, P(\,f\,|\,+\,) \\ (1 - x_{\pi,+})\, P(\,f\,|\,-\,) \end{array} \right\}\\
= \sum_{f\in\mathcal{V}_F} &\left\{ \begin{array}{ll} x_{\pi,+} P(\,f\,|\,+\,) & \text{if } x_{\pi,+} \le \frac{P(\,f\,|\,-\,)}{P(\,f\,|\,+\,)+P(\,f\,|\,-\,)} \\ (1 - x_{\pi,+}) P(\,f\,|\,-\,) & \text{otherwise.} \end{array} \right.
\end{aligned} \qquad (3)$$

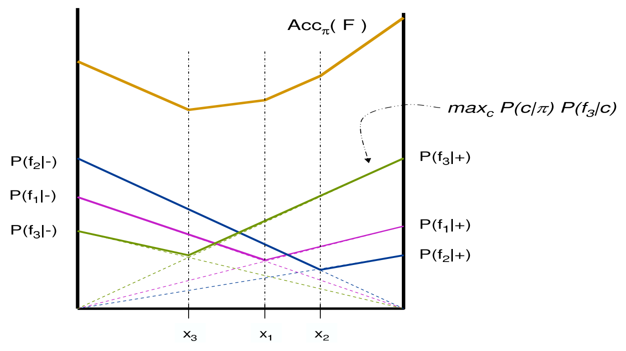

Figure 2: Computing $\text{Acc}_\pi(\,F\,)$ (Equation 3).

```
OPTNBDT( $d$: int; $P(\,\cdot\,)$: distribution over $\{C\} \cup \mathcal{F}$)
  Compute opt-feature list
      OFL = $\{F_i^*(x) \mid x \in [0,1],\; i = 1..d\}$
  Build optimal depth-$d-1$ tree
      using DynamicProgramming
    At each length-$d-1$ path $\pi$,
        with associated probability $x_{\pi,+} = P(\,C = +\,|\,\pi\,)$
      Let $\vec{\mathcal{F}} = \langle\, F_1^*(x_{+,\pi}), \ldots, F_{d-1}^*(x_{+,\pi})\,\rangle$.
      Let $i^* = \min_i\{F_i^*(x_{+,\pi}) \notin \pi\}$ be the first in $\vec{\mathcal{F}}$ not in $\pi$
      Use feature $F_{i^*}^*(x)$ at level $d$, after $\pi$
```

Figure 3: **OPTNBDT** algorithm.

For fixed values of $P(\,F = f\,|\,C = c)$, *this* $\text{Acc}_\pi(\,F\,)$ *value does not depend on the features in* π *nor their values, but only on* $x_{\pi,+}$; we can therefore express $\text{Acc}_\pi(\,F\,)$ as $\text{Acc}(\,F,\, x_{\pi,+})$ to make this dependency explicit. For any value of $x_{\pi,+} \in [0,1]$, each summand in Equation 3, corresponding to a single $F = f$, is the maximum of two lines. Over the set of r_F values, this function is therefore a sequence of at most $1 + r_F$ linear segments; see Figure 2.

Before beginning to build the actual decision tree, our OPTNBDT algorithm will first compute these $\text{Acc}(\,F^{(j)},\, x)$ functions for each $F^{(j)}$. It then uses these functions to compute, for each $x \in [0,1]$, the optimal feature:

$$F_1^*(x) \;=\; \underset{F}{\text{argmax}}\{\text{Acc}(\,F,\, x)\}. \qquad (4)$$

It also computes the *2nd-best* feature $F_2^*(x)$ for each x value, as well as $F_i^*(x)$ for $i = 3, 4, \ldots, d$. We refer to this $\{F_i^*(x) \mid x \in [0,1],\; i = 1..d\}$ dataset as the OFL ("opt-feature list").

The OPTNBDT algorithm then uses a dynamic program to build the tree, but only to depth $d - 1$. Whenever it reaches a depth $d - 1$ node, at the end of the path $\pi = \langle\, \langle\, F_{\pi(1)}, f_{\pi(1)}\,\rangle, \langle\, F_{\pi(2)}, f_{\pi(2)}\,\rangle, \ldots, \langle\, F_{\pi(d-1)}, f_{\pi(d-1)}\,\rangle\,\rangle$, with associated conditional probability $x_{\pi,+}$, it then indexes this $x_{\pi,+}$ into the OFL, which returns an ordered list of d features, $\vec{\mathcal{F}}(x_{+,\pi}) = \langle\, F_1^*(x_{+,\pi}), \ldots, F_d^*(x_{+,\pi})\,\rangle$. OPTNBDT then returns the first F_i that is *not* in the $\vec{\mathcal{F}}(x_{+,\pi})$ list. This algorithm is shown in Figure 3.

To make this more concrete, imagine that in Figure 1 the list associated with $\pi(\boxed{?}) = \langle\, \langle\, A, +\,\rangle, \langle\, B, -\,\rangle, \langle\, D, -\,\rangle\,\rangle$

and $x_{\pi,+} = 0.29$ was $\vec{\mathcal{F}}(0.29) = \langle B, A, E, D \rangle$. While we would like to use the first value $F_1^*(0.29) = B$ as it appears to be the most accurate, we cannot use it as this feature has already been appeared in this π path and therefore Equation 3 does not apply. OPTNBDT would then consider the second feature, which here is $F_2^*(0.29) = A$. Unfortunately, as that appears in π as well, OPTNBDT have to consider the third feature, $F_3^*(0.29) = E$. Since that does not appear, OPTNBDT labels the $\boxed{?}$ node with this "E" feature.

3.2 Implementation

This section provides the details of our implementation, which requires recursively computing $x = x_{\pi,+}$, and computing and using the opt-feature list, OFL.

Computing $x_{\pi,+}$: It is easy to obtain $x_{\pi,+}$ as we grow the path π in the decision tree. Here, we maintain two quantities, $y_\pi^+ = P(\pi, C = +)$ and $y_\pi^- = P(\pi, C = -)$, then for any path π, then set $x_{\pi,+} = \frac{y_\pi^+}{y_\pi^+ + y_\pi^-}$. For $\pi_0 = \{\}$, $y_{\pi_0}^c = P(C = c)$ for $c \in \{+, -\}$. Now consider adding one more feature-value pair $\langle F, f \rangle$ to π_t, to form $\pi_{t+1} = \pi_t \circ \langle F, f \rangle$. Then thanks to our NB assumption, $y_{\pi_{t+1}}^c = y_{\pi_t}^c \times P(F = f \mid C = c)$.

Computing and using the OFL: As noted above, OFL corresponds to a set of *piecewise linear functions* (see Figure 2), each of which is the union of a finite number of linear functions. Formally, a piecewise linear function $f : [0, 1] \to \Re$ (with k pieces) can be described by a sequence of real number triples $\langle (a_1, m_1, b_1), \ldots, (a_k, m_k, b_k) \rangle$ where the a_is are endpoints such that $0 = a_0 < a_1 < \ldots < a_k = 1$, and for all $i \in \{1, \ldots, k\}$ we have $f(x) = m_i x + b_i$ for all $x \in [a_{i-1}, a_i]$.

A linear function is a piecewise linear function with one piece. The sum of two piecewise linear functions with k_1 and k_2 pieces is a piecewise linear function with no more than $k_1 + k_2 - 1$ pieces. We can compute this sum in $O(k_1 + k_2)$ time, as each component of the sum $f = f_1 + f_2$ is just the sum of the relevant m and b from both f_1 and f_2. Similarly, the maximum of two piecewise linear functions with k_1 and k_2 pieces is a piecewise linear function with no more than $(k_1 + k_2)$ pieces. This computation is slightly more involved, but can be done in a similar way.

For each feature F and each value $x_\pi \in [0, 1]$, we need to compute the sum over all $|\mathcal{V}_F| \leq r$ values of $F = f$:

$$\mathrm{Acc}_\pi(F) = \sum_{f \in \mathcal{V}_F} \mathrm{Acc}_\pi(F = f)$$

$$\mathrm{Acc}_\pi(F = f) = \begin{cases} x_{\pi,+} P(f_i^{(j)} \mid +) & \text{if } x_{\pi,+} \leq \frac{P(f_i^{(j)} \mid -)}{P(f_i^{(j)} \mid +) + P(f_i^{(j)} \mid -)} \\ (1 - x_{\pi,+}) P(f_i^{(j)} \mid -) & \text{otherwise.} \end{cases}$$

We compute this total by adding the associated $|\mathcal{V}_F| \leq r$ piecewise linear functions, $\{\mathrm{Acc}_\pi(F = f)\}_{f \in \mathcal{V}_F}$. We can do this recursively: Letting $r = |\mathcal{V}_F|$ and $\ell_i = \mathrm{Acc}_\pi(F = f_i)$ be the i^{th} function, we first define $\ell_{12} = \ell_1 + \ell_2$ as the sum of ℓ_1 and ℓ_2, $\ell_{34} = \ell_3 + \ell_4$, and so forth until $\ell_{r-1,r} = \ell_{r-1} + \ell_r$; we next let $\ell_{14} = \ell_{12} + \ell_{34}$, $\ell_{58} = \ell_{56} + \ell_{78}$, etc.; continuing until computing $\ell_{1,r}$ which is the sum over all r functions. By recursion, one can prove that this resulting piecewise linear function has no more than $r + 1$ pieces, and that the time complexity is $O(r \log r)$, due to $\log r$ levels of recursion, each of which involves a total of $O(2^i \times r/2^i) = O(r)$ time.

Note that Equation 4 is the maximum of all of the piecewise linear functions $\{\mathrm{Acc}_\pi(F)\}_{F \in \mathcal{F}}$ that were constructed in the previous step. We again use a divide and conquer technique to reduce the problem of maximizing over n piecewise linear functions to sequence of $\log n$ problems, each of maximizing over two piecewise linear functions. An analysis similar to mergesort shows that the overall computational cost of the process is $O(nr \log nr)$.

When $F_1^*(\cdot)$ (Equation 4) involves $k = O(nr)$ linear pieces at arbitrary points, we can compute $F_1^*(x)$ in $O(\log k) = O(\ln nr)$ time, using a binary search to find the appropriate segment and a constant amount of time to compute the value given this segment. As we are computing this maximum value, we can also specify which feature represents this segment.

We can compute $F_i^*(x)$ for $i = 2, 3, \ldots, d$ in a similar way. Consequently, when we compute the maximum of two functions, we also store the d highest functions at every point. Note the amount of memory storage required here increases linearly in d.

Hence, each lookup function can be constructed in $O(nr \log(nr))$ time, requires $O(nr)$ memory to store, and most importantly, can be queried (to find the optimal value for a specific x) in $O(\log(nr))$ time. Note that a naïve implementation that merely tests all the features in the last level of the tree to construct the leaves (called "NBDT" below) will require a time linear in nr, whereas our technique has complexity logarithmic in nr.

Collectively, these results show

Theorem: *Given any labeled dataset S over n r-ary variables and binary class labels, the* OPTNBDT *algorithm (Figure 3) will compute the depth-d decision tree that has the highest accuracy under the Naïve Bayes assumption. Moreover, it requires $O(n|S| + (nr)^{d-1} d \log(nr))$ time*[3] *and $O(d(nr)^d)$ space.*

4 Empirical Results

We performed an extensive set of experiments to compare the performance of OPTNBDT to various learners, both state-of-the-art decision tree learners, and Naïve Bayes systems. This section first describes the datasets and experimental setup, then the experimental results and finally our analyses.

Experimental setup: The experiments are performed using nine datasets from UCI Machine Learning Repository (Newman *et al.* 1998) as well as our own Breast Cancer and Prostate Cancer "single nucleotide polymorphism" datasets

[3]The additional $n|S|$ term is the time required to compute the basic Naïve Bayes statistics over the dataset S.

Table 1: Datasets used in the experiments

Dataset	Num. features	Num. instances	Num. feature values	Abbre-viation
Tic-tac-toe	10	985	27	Tic
Flare	10	1066	31	Flr
Hepatite	13	80	26	Hep
Crx	13	653	30	Crx
Vote	16	435	48	Vot
Lymph-ography	18	145	50	Lym
Chess	36	3196	73	Chs
Connect-4	42	5000	126	Con
Prostate-cancer	47	81	126	Prs
Promoters	58	106	228	Prm
Breast-cancer	98	332	392	Snp

obtained from our colleagues at the Cross Cancer Institute. All the selected datasets have binary class labels. Table 1 characterizes these datasets.

The experiments are performed using 5-fold cross validation, performed 20 times; we report the average error rate and standard deviation across these trials. We compare the classification performance and running time of OptNbDt with different depths — here 2, 3 and 4 — to the following fixed-depth decision trees:

- *3-ID3* (Quinlan 1986) is a decision tree learning algorithm that uses an entropy based measure as its splitting criterion, but stops at depth 3.
- *1*-NbDt (Holte 1993) is a decision stump that splits the data with only one feature — *i.e.*, it is a depth-one decision tree.

Experimental results: Figure 4 shows the average classification error rate and standard deviation of each algorithm.[4] We see that no algorithm is consistently superior over all the datasets. While deeper decision trees sometimes improve classification performance (classification accuracy increases with the depth of the tree for the `Hep`, `Chs`, `Con`, and `Prm` datasets), deeper trees can cause overfitting, which may explain why shallower trees can be better: Decision stumps, which only split the data based on one feature, outperform the other trees for the `flr`, `crx`, and `vot` datasets.

Figure 4 shows that the classification error of 3-OptNbDt is often lower than the errors from 3-ID3, which shows that the optimal Naïve Bayes-based decision trees can perform better than heuristic, entropy-based trees. In fact, a paired student's t-test over these eleven datasets show that both 3-OptNbDt and 2-OptNbDt are statistically better (more accurate) than 3-ID3 (at $p < 0.05$). As expected,

[4]We omit the 4-OptNbDt result for snp, as that took too long to run; see Figure 6.

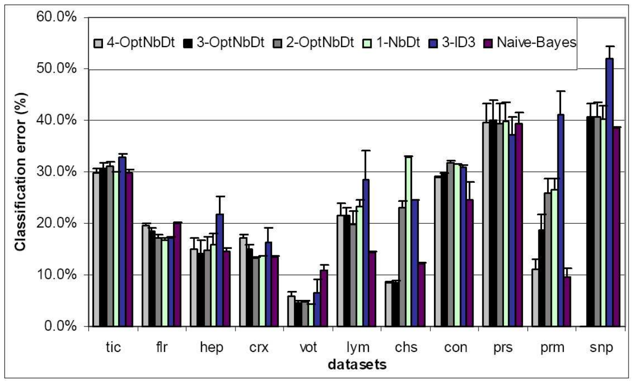

Figure 4: Average classification error (with 1 standard deviation bars) for 1-NbDt, 2-OptNbDt, 3-OptNbDt, 4-OptNbDt, 3-ID3 and Naïve Bayes classifiers.

we found that (on average) the error improves for deeper trees: 4-OptNbDt (0.197) $<$ 3-OptNbDt (0.220) $<$ 2-OptNbDt (0.238) $<$ 1-NbDt (0.250), although none of the methods are significantly better than its neighbors. (Recall that all of these are better than 3-ID3, whose average is 0.281.)

Figure 4 also includes the results of the Naïve Bayes classifier. While it does works fairly well in many of these datasets, note that it is not a contender: recall that our goal is to produce an effective shallow decision tree, which involves only a (small) subset of the features, probed sequentially. However, these Naïve Bayes systems will use all of the feature values, which must all be available.

Running Time: Now consider the running time of these algorithms, as well as 3-NbDt, which implements the dynamic programming algorithm that produces the fixed depth decision tree that is optimal over the training data, given the Naïve Bayes assumption — *i.e.*, this d-NbDt algorithm explicitly constructs and tests all possible leaves up to depth d, whereas d-OptNbDt uses a pre-computed opt-feature list (ofl) to efficiently construct the final (depth d) level of the decision tree. Of course, these two algorithms give exactly the same answers.

The previous section proved that OptNbDt is $O(\log(n)/n)$ times faster than NbDt, where n is the number of features. To explore this claim empirically, we extracted eighteen artificial datasets from the Promoters `Prm`) dataset, using $3i$ features, for $i = 1 : 18$. Figure 5 plots $\log(f(n) \times n/\log n)$ versus $\log(g(n))$, where $f(n)$ (resp., $g(n)$) is the run-time that 3-NbDt (resp., 3-OptNbDt) requires for n features. This confirms that OptNbDt is significantly more efficient than NbDt, especially when there are many features in the dataset.

Figure 6 shows the running time of the various algorithms: Each line corresponds to one of the algorithms, formed by joining a set of (x, y) points, whose x value is the total number of feature values $\sum_i r_i = O(nr)$ of a particular dataset, and y is the (log of the) run time of the algorithm on that

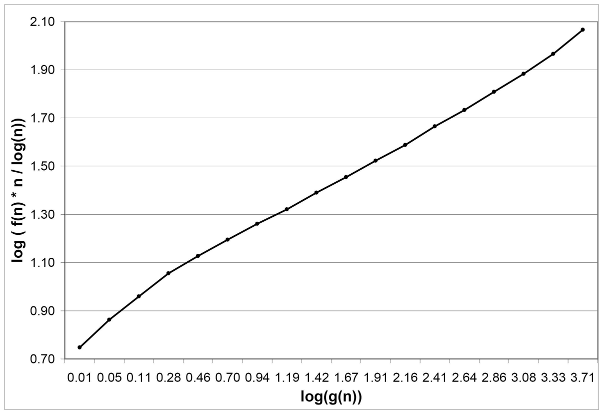

Figure 5: Comparing 3-OPTNBDT vs 3-NBDT: $g(n)$ and $f(n)$ = time that 3-NBDT and 3-OPTNBDT require for n features, respectively.

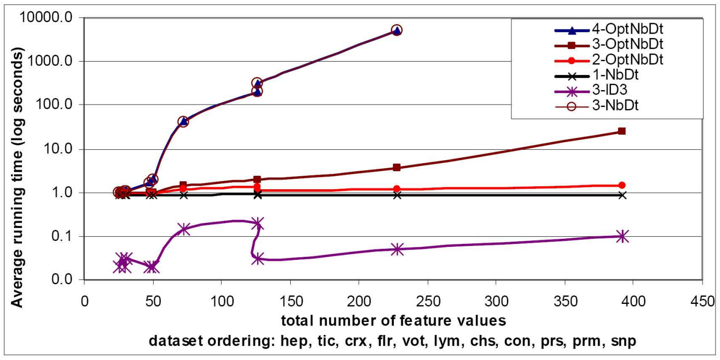

Figure 6: Run-time (log of seconds) of 4-OPTNBDT, 3-OPTNBDT, 2-OPTNBDT, 1-NBDT, 3-ID3, and 3-NBDT algorithms

dataset. As expected, the 3-ID3 points are fairly independent of this number of features. The other lines, however, appear fairly linear in this log plot, at least for larger values of $\sum_i r_i$; this is consistent with the Theorem. Finally, to understand why the 4-OPTNBDT timing numbers are virtually identical to the 3-NBDT numbers, recall that 4-OPTNBDT basically runs 3-NBDT to produce a depth-3 decision, then uses the OFL to compute the 4th level. These numbers show that the time required to compute OFL, and to use it to produce the final depth, is extremely small. (The actual timing numbers appear in (Greiner 2007).)

5 Conclusion

Extensions: (1) While this paper only discusses how to use the *opt-feature list* to fill in the features at the *last level* of the tree, our techniques are not specific to this final row. These ideas can be extended to pre-compute an optimal tree list with the final depth-q subtree at the end of a path given the label posterior, rather than just the final internal node; *i.e.*, so far, we have dealt only with $q = 1$. While this will significantly increase the cost of the precomputational stage, it should provide a signifiant computational gain when growing the actual tree. (2) A second extension is to develop further tricks that allow us to efficiently build and compare a *set* of related decision trees — perhaps trees that are based on slightly different distributions, obtained from slightly different training samples. This might be relevant in the context of some ensemble methods (Dietterich 2000), such as boosting or bagging, or in the context of budgeted learning of bounded classifiers (Kapoor & Greiner 2005). (3) Our approach uses a simple cost model, where every feature has unit cost. An obvious extension would allow different features to have different costs, where the goal is to produce a decision tree whose "cost depth" is bounded.[5]

[5]The "cost depth" of a leaf of a decision tree is the sum of the costs of the tests of the nodes connecting the root to this leaf; and the "cost depth" of a tree is the maximal cost depth over the leaves. Note "cost depth" equals "depth" if all features have unit cost.

Contributions: There are many situations where we need to produce a *fixed-depth decision tree* — *i.e.*, the bounded policy associated with per-patient capitation. This paper has presented OPTNBDT, an algorithm that efficiently produces the optimal such fixed-depth decision tree, given the Naïve Bayes assumption — *i.e.*, assuming that the features are independent given the class. We proved that this assumption implies that the optimal feature at the last level of the tree essentially depends only on $x_{\pi,+} \in [0, 1]$, the posterior probability of the class label given the tests previously performed. We then describe a way to efficiently pre-compute which feature is best, as a function of this probability value, and then, when building the tree itself, to use this information to quickly assign the final feature on each path. We prove theoretically that this results in a speedup of $O(n/\log n)$ compared to the naïve method of testing all the features in the last level, then provide empirical evidence, over a benchmark of eleven datasets, supporting this claim. We also found that OPTNBDT is not just efficient, but (surprisingly, given its NB assumption) is often more accurate than entropy based decision tree learner like ID3.

6 Acknowledgements

All authors gratefully acknowledge the support from the Alberta Ingenuity Centre for Machine Learning (AICML), NSERC, and iCORE. We also appreciate the many insightful comments from the anonymous referees.

References

Auer, P.; Holte, R. C.; and Maass, W. 1995. Theory and applications of agnostic PAC-learning with small decision trees. In *ICML*.

Bennett, K., and Blue, J. A. 1996. Optimal decision trees. *Rensselaer Polytechnic Institute Math Report* (214).

Breiman, L.; Friedman, J.; Olshen, J.; and Stone, C. 1984. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks.

Chan, H., and Darwiche, A. 2003. Reasoning about bayesian network classifiers. In *UAI*, 107–116.

Dietterich, T. G. 2000. Ensemble methods in machine learning. In *First International Workshop on Multiple Classifier Systems*.

Dobkin, D.; Gunopoulos, D.; and Kasif, S. 1996. Computing optimal shallow decision trees. In *International Symposium on Artificial Intelligence and Mathematics*.

Domingos, P., and Pazzani, M. 1997. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning* 29:103–130.

Duda, R. O., and Hart, P. E. 1973. *Pattern Classification and Scence Analysis*. Wiley.

Greiner, R.; Grove, A.; and Roth, D. 2002. Learning cost-sensitive active classifiers. *Artificial Intelligence* 139:137–174.

Greiner. 2007. `http://www.cs.ualberta.ca/~greiner/RESEARCH/DT-NB`.

Holte, R. C. 1993. Very simple classification rules perform well on most commonly used datasets. *Machine Learning* 11:63–91.

Kapoor, A., and Greiner, R. 2005. Learning and classifying under hard budgets. In *Proceedings of the Sixteenth European Conference on Machine Learning (ECML-05)*, 170–181. Springer.

Lewis, D. D. 1998. Naïve Bayes at forty: The independence assumption in information retrieval. In *ECML*.

Newman, D. J.; Hettich, S.; Blake, C. L.; and Merz, C. J. 1998. *UCI Repository of Machine Learning Databases*. University of California, Irvine, Dept. of Information and Computer Sciences: `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Nijssen, S., and Fromont, E. 2007. Mining optimal decision trees from itemset lattices. In *The 13th International Conference on Knowledge Discovery and Data Mining (KDD2007)*.

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

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

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning*. MIT Press.

Turney, P. D. 2000. Types of cost in inductive concept learning. In *Workshop on Cost-Sensitive Learning (ICML-2000)*.