Machine Learning Research: Four Current Directions

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331

Final Draft

Abstract

Machine Learning research has been making great progress in many directions. This article summarizes four of these directions and discusses some current open problems. The four directions are (a) improving classification accuracy by learning ensembles of classifiers, (b) methods for scaling up supervised learning algorithms, (c) reinforcement learning, and (d) learning complex stochastic models.

1 Introduction

The last five years have seen an explosion in machine learning research. This explosion has many causes. First, separate research communities in symbolic machine learning, computational learning theory, neural networks, statistics, and pattern recognition have discovered one another and begun to work together. Second, machine learning techniques are being applied to new kinds of problems including knowledge discovery in databases, language processing, robot control, and combinatorial optimization as well as in more traditional problems such as speech recognition, face recognition, handwriting recognition, medical data analysis, game playing, and so on.

In this article, I have selected four topics within machine learning where there has been a lot of recent activity. The purpose of the article is to describe the results in these areas to a broader AI audience and to sketch some of the open research problems. The topic areas are (a) ensembles of classifiers, (b) methods for scaling up supervised learning algorithms, (c) reinforcement learning, and (d) learning complex stochastic models.

The reader should be cautioned that this article is not a comprehensive review of each of these topics. Rather, my goal is to provide a representative sample of the research in each of these four areas. In each of the areas, there are many other papers that describe relevant work. I apologize to those authors whose work I was unable to include in the article.

2 Ensembles of Classifiers

The first topic concerns methods for improving accuracy in supervised learning. I begin by introducing some notation. In supervised learning, a learning program is given training examples of the form $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ for some unknown function $y = f(\mathbf{x})$. The $\mathbf{x}_i$ values are typically vectors of the form $\langle x_{i,1}, x_{i,2}, \ldots, x_{i,n}\rangle$ whose components are discrete- or real-valued such as height, weight, color, age, and so on. These are also called the *features* of $\mathbf{x}_i$. I will use the notation x_{ij} to refer to the j-th feature of $\mathbf{x}_i$. In some situations, I will drop the i subscript when it is implied by the context.

The y values are typically drawn from a discrete set of classes $\{1, \ldots, K\}$ in the case of *classification* or from the real line in the case of *regression*. In this article, I will focus primarily on classification. The training examples may be corrupted by some random noise.

Given a set S of training examples, a learning algorithm outputs a *classifier*. The classifier is an hypothesis about the true function f. Given new $\mathbf{x}$ values, it predicts the corresponding y values. I will denote classifiers by $h_1, \ldots, h_L$.

An ensemble of classifiers is a set of classifiers whose individual decisions are combined in some way (typically by weighted or unweighted voting) to classify new examples. One of the most active areas of research in supervised learning has been to study methods for constructing good ensembles of classifiers. The main discovery is that ensembles are often much more accurate than the individual classifiers that make them up.

An ensemble can be more accurate than its component classifiers only if the individual classifiers disagree with one another (Hansen & Salamon, 1990). To see why, imagine that we have an ensemble of three classifiers: $\{h_1, h_2, h_3\}$

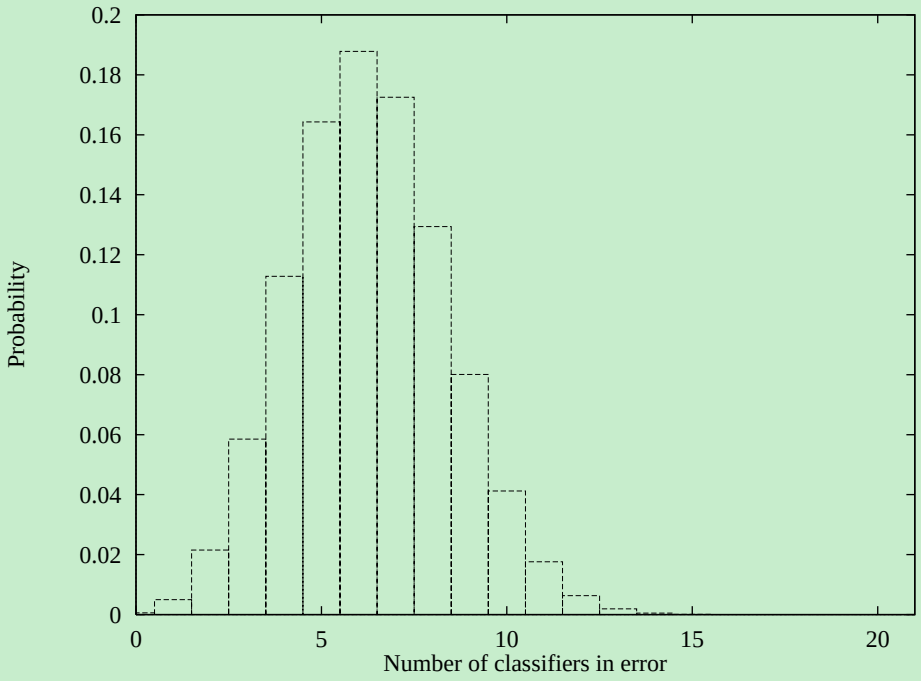

Figure 1: The probability that exactly ℓ (of 21) hypotheses will make an error, assuming each hypothesis has an error rate of 0.3 and makes its errors independently of the other hypotheses.

and consider a new case $\mathbf{x}$. If the three classifiers are identical, then when $h_1(\mathbf{x})$ is wrong, $h_2(\mathbf{x})$ and $h_3(\mathbf{x})$ will also be wrong. However, if the errors made by the classifiers are uncorrelated, then when $h_1(\mathbf{x})$ is wrong, $h_2(\mathbf{x})$ and $h_3(\mathbf{x})$ may be correct, so that a majority vote will correctly classify $\mathbf{x}$. More precisely, if the error rates of L hypotheses h_ℓ are all equal to $p < 1/2$ and if the errors are independent, then the probability that the majority vote will be wrong will be the area under the binomial distribution where more than $L/2$ hypotheses are wrong. Figure 1 shows this for a simulated ensemble of 21 hypotheses, each having an error rate of 0.3. The area under the curve for 11 or more hypotheses being simultaneously wrong is 0.026, which is much less than the error rate of the individual hypotheses.

Of course, if the individual hypotheses make uncorrelated errors at rates exceeding 0.5, then the error rate of the voted ensemble will increase as a result of the voting. Hence, the key to successful ensemble methods is to construct individual classifiers with error rates below 0.5 whose errors are at least somewhat uncorrelated.

2.1 Methods for Constructing Ensembles

Many methods for constructing ensembles have been developed. Some methods are general, and they can be applied to any learning algorithm. Other methods are specific to particular algorithms. We begin by reviewing the general techniques.

2.1.1 Subsampling the Training Examples

The first method manipulates the training examples to generate multiple hypotheses. The learning algorithm is run several times, each time with a different subset of the training examples. This technique works especially well for *unstable* learning algorithms—algorithms whose output classifier undergoes major changes in response to small changes in the training data. Decision-tree, neural network, and rule learning algorithms are all unstable. Linear regression, nearest neighbor, and linear threshold algorithms are generally very stable.

The most straightforward way of manipulating the training set is called *bagging*. On each run, bagging presents the learning algorithm with a training set that consists of a sample of m training examples drawn randomly with replacement from the original training set of m items. Such a training set is called a *bootstrap replicate* of the original training set, and the technique is called *bootstrap aggregation* (from which the term *bagging* is derived; Breiman, 1996a). Each bootstrap replicate contains, on the average, 63.2% of the original training set, with several training examples appearing multiple times.

Table 1: The AdaBoost.M1 algorithm. The formula $[\![E]\!]$ is 1 if E is true and 0 otherwise.

Input: a set S, of m labeled examples: $S = \{(\mathbf{x}_i, y_i), i = 1, 2, \ldots, m\}$,
labels $y_i \in Y = \{1, \ldots, K\}$
Learn (a learning algorithm)
a constant L.

[1] **initialize for all** i: $w_1(i) := 1/m$ — *initialize the weights*
[2] **for** $\ell = 1$ **to** L **do**
[3] **for all** i: $p_\ell(i) := w_\ell(i)/(\sum_i w_\ell(i))$ — *compute normalized weights*
[4] $h_\ell := \text{Learn}(p_\ell)$ — *call Learn with normalized weights.*
[5] $\epsilon_\ell := \sum_i p_\ell(i)[\![h_\ell(\mathbf{x}_i) \neq y_i]\!]$ — *calculate the error of* h_ℓ
[7] **if** $\epsilon_\ell > 1/2$ **then**
[8] $L := \ell - 1$
[9] **goto 13**
[10] $\beta_\ell := \epsilon_\ell/(1 - \epsilon_\ell)$
[11] **for all** i: $w_{\ell+1}(i) := w_\ell(i)\beta_\ell^{1-[\![h_\ell(\mathbf{x}_i) \neq y_i]\!]}$ — *compute new weights*
[12] **end for**

[13] **Output**: $h_f(\mathbf{x}) = \underset{y \in Y}{\text{argmax}} \sum_{\ell=1}^{L} \left(\log \frac{1}{\beta_\ell}\right) [\![h_\ell(\mathbf{x}) = y]\!]$

Another training set sampling method is to construct the training sets by leaving out disjoint subsets of the training data. For example, the training set can be randomly divided into 10 disjoint subsets. Then 10 overlapping training sets can be constructed by dropping out a different one of these 10 subsets. This same procedure is employed to construct training sets for 10-fold cross-validation, so ensembles constructed in this way are sometimes called *cross-validated committees* (Parmanto, Munro, & Doyle, 1996).

The third method for manipulating the training set is illustrated by the AdaBoost algorithm, developed by Freund and Schapire (1995, 1996) and shown in Table 1. Like bagging, AdaBoost manipulates the training examples to generate multiple hypotheses. AdaBoost maintains a probability distribution $p_\ell(\mathbf{x})$ over the training examples. In each iteration ℓ, it draws a training set of size m by sampling with replacement according the probability distribution $p_\ell(\mathbf{x})$. The learning algorithm is then applied to produce a classifier h_ℓ. The error rate ϵ_ℓ of this classifier on the training examples (weighted according to $p_\ell(\mathbf{x})$) is computed and used to adjust the probability distribution on the training examples. (In Table 1, note that the probability distribution is obtained by normalizing a set of weights $w_\ell(i)$ over the training examples.)

The effect of the change in weights is to place more weight on training examples that were misclassified by h_ℓ and less weight on examples that were correctly classified. In subsequent iterations, therefore, AdaBoost constructs progressively more difficult learning problems.

The final classifier, h_f, is constructed by a weighted vote of the individual classifiers. Each classifier is weighted according to its accuracy for the distribution p_ℓ that it was trained on.

In line 4 of the AdaBoost algorithm, the base learning algorithm *Learn* is called with the probability distribution p_ℓ. If the learning algorithm *Learn* can use this probability distribution directly, then this generally gives better results. For example, Quinlan (1996) developed a version of the decision-tree learning program C4.5 that works with a weighted training sample. His experiments showed that it worked extremely well. One can also imagine versions of backpropagation that scaled the computed output error for training example $(\mathbf{x}_i, y_i)$ by the weight $p_\ell(i)$. Errors for "important" training examples would cause larger gradient descent steps than errors for unimportant (low-weight) examples.

On the other hand, if the algorithm cannot use the probability distribution p_ℓ directly, then a training sample can be constructed by drawing a random sample with replacement in proportion to the probabilities p_ℓ. This makes AdaBoost more stochastic, but experiments have shown that this procedure is still very effective.

Figure 2 compares the performance of C4.5 to C4.5 with AdaBoost.M1 (using random sampling). One point is plotted for each of 27 test domains taken from the Irvine repository of machine learning databases (Merz & Murphy, 1996). We can see that most points lie above the line $y = x$, which indicates that the error rate of AdaBoost is less than the error rate of C4.5. Figure 3 compares the performance of bagging (with C4.5) to C4.5 alone. Again, we see that bagging produces sizeable reductions in the error rate of C4.5 for many problems. Finally, Figure 4 compares bagging with boosting (both using C4.5 as the underlying algorithm). The results show that the two techniques are comparable, although boosting appears still to have an advantage over bagging.

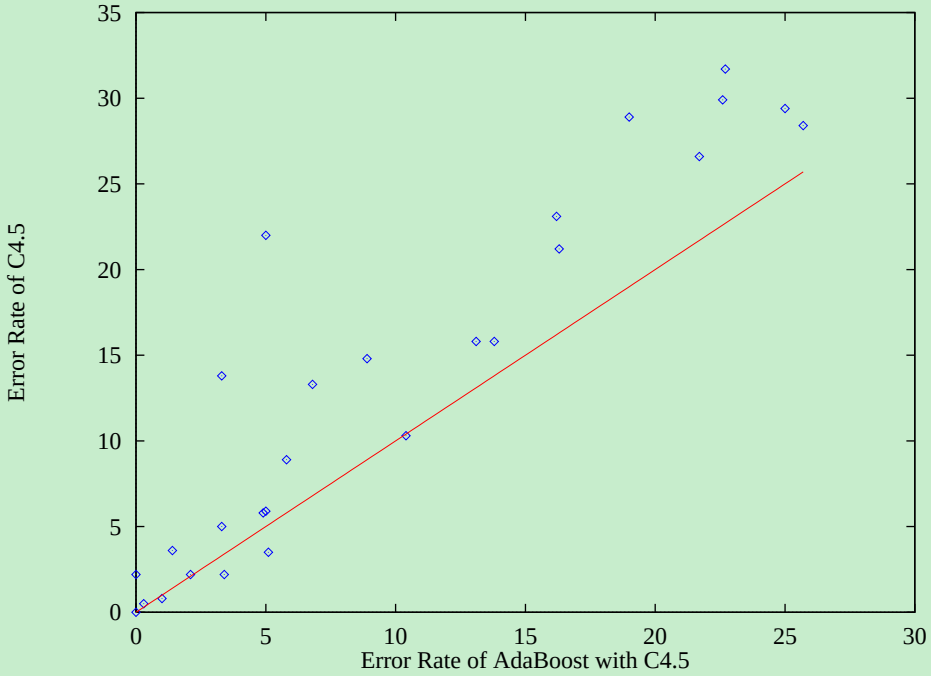

Figure 2: Comparison of AdaBoost.M1 (applied to C4.5) with C4.5 by itself. Each point represents one of 27 test domains. Points lying above the diagonal line exhibit lower error with AdaBoost.M1 than with C4.5 alone. Based on data from Freund & Schapire (1996). Up to 100 hypotheses were constructed.

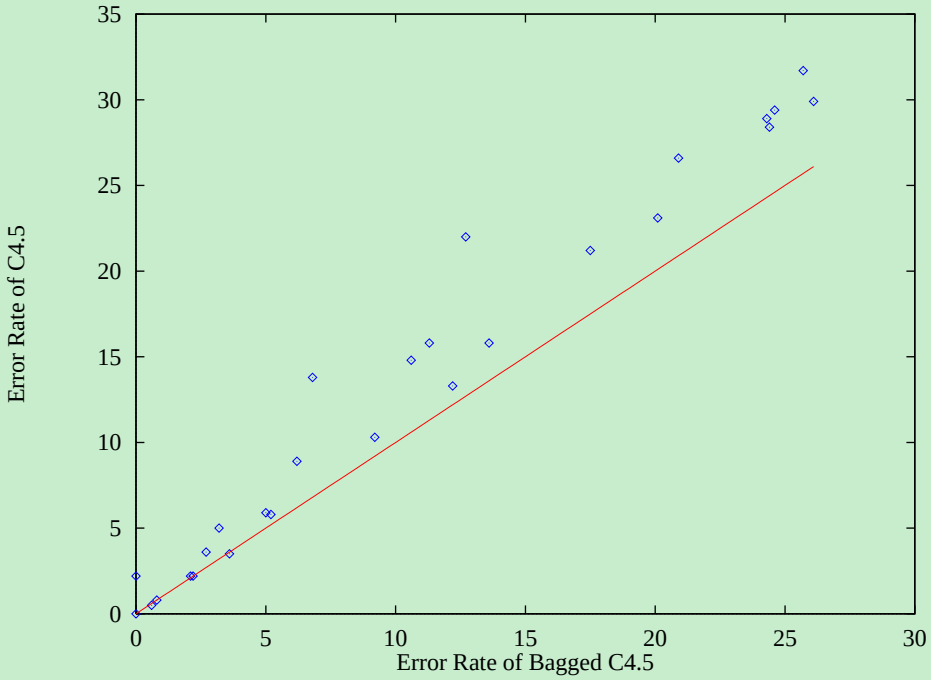

Figure 3: Comparison of bagging (applied to C4.5) with C4.5 by itself. Each point represents one of 27 test domains. Points lying above the diagonal line exhibit lower error with bagging than with C4.5 alone. Based on data from Freund & Schapire (1996). Bagging voted 100 classifiers.

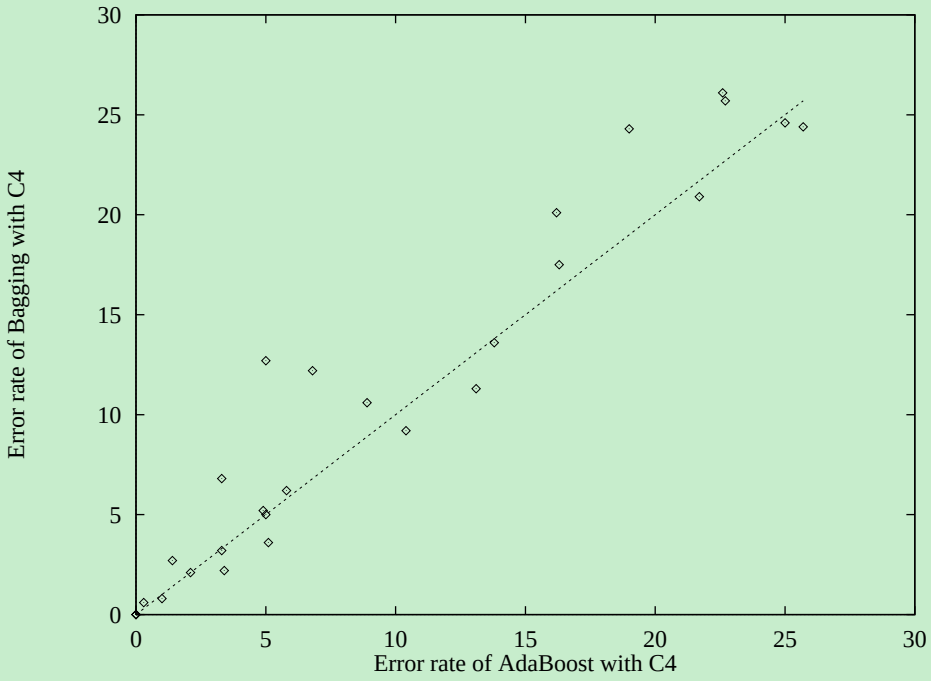

Figure 4: Comparison of bagging (applied to C4.5) with AdaBoost.M1 (applied to C4.5). Each point represents one of 27 test domains. Points lying above the diagonal line exhibit lower error with boosting than with bagging. Based on data from Freund & Schapire (1996).

2.1.2 Manipulating the Input Features

A second general technique for generating multiple classifiers is to manipulate the set of *input features* available to the learning algorithm. For example, in a project to identify volcanoes on Venus, Cherkauer (1996) trained an ensemble of 32 neural networks. The 32 networks were based on 8 different subsets of the 119 available input features and 4 different network sizes. The input feature subsets were selected (by hand) to group together features that were based on different image processing operations (such as principal component analysis and the fast fourier transform). The resulting ensemble classifier was able to match the performance of human experts in identifying volcanoes. Tumer and Ghosh (1996) applied a similar technique to a sonar dataset with 25 input features. However, they found that deleting even a few of the input features hurt the performance of the individual classifiers so much that the voted ensemble did not perform very well. Obviously, this technique only works when the input features are highly redundant.

2.1.3 Manipulating the Output Targets

A third general technique for constructing a good ensemble of classifiers is to manipulate the y values that are given to the learning algorithm. Dietterich & Bakiri (1995) describe a technique called error-correcting output coding. Suppose that the number of classes, K, is large. Then new learning problems can be constructed by randomly partioning the K classes into two subsets A_ℓ and B_ℓ. The input data can then be re-labeled so that any of the original classes in set A_ℓ are given the derived label 0 and the original classes in set B_ℓ are given the derived label 1. This relabeled data is then given to the learning algorithm, which constructs a classifier h_ℓ. By repeating this process L times (generating different subsets A_ℓ and B_ℓ), we obtain a ensemble of L classifiers $h_1, \ldots, h_L$.

Now given a new data point $\mathbf{x}$, how should we classify it? The answer is to have each h_ℓ classify $\mathbf{x}$. If $h_\ell(\mathbf{x}) = 0$, then each class in A_ℓ receives a vote. If $h_\ell(\mathbf{x}) = 1$, then each class in B_ℓ receives a vote. After each of the L classifiers has voted, the class with the highest number of votes is selected as the prediction of the ensemble.

An equivalent way of thinking about this method is that each class j is encoded as an L-bit codeword C_j, where bit ℓ is 1 if and only if $j \in B_\ell$. The ℓ-th learned classifier attempts to predict bit ℓ of these codewords. When the L classifiers are applied to classify a new point $\mathbf{x}$, their predictions are combined into an L-bit string. We then choose the class j whose codeword C_j is closest (in Hamming distance) to the L-bit output string. Methods for designing

Table 2: Results on Five Domains (best error rate in **boldface**)

Task	Test set size	C4.5	200-fold bootstrap C4.5	200-fold random C4.5
Vowel	462	0.5758	0.5152	**0.4870***
Soybean	376	0.1090	**0.0984**	0.1090
Part-of-Speech	3060	0.0827	**0.0765**	0.0788^{a}
NETtalk	7242	0.3000	0.2670***	**0.2500*****
Letter Recognition	4000	0.2010	0.0038***	**0.0000*****

Difference from C4.5 significant at $p < 0.05^{*}$, 0.001^{***}. ^{a}256-fold random.

good error-correcting codes can be applied to choose the codewords C_j (or equivalently, subsets A_ℓ and B_ℓ).

Dietterich and Bakiri report that this technique improves the performance of both the C4.5 and backpropagation algorithms on a variety of difficult classification problems. Recently, Schapire (1997) has shown how AdaBoost can be combined with error-correcting output coding to yield an excellent ensemble classification method that he calls AdaBoost.OC. The performance of the method is superior to the ECOC method (and to bagging), but essentially the same as another (quite complex) algorithm, called AdaBoost.M2. Hence, the main advantage of AdaBoost.OC is implementation simplicity: It can work with any learning algorithm for solving 2-class problems.

Ricci and Aha (1997) applied a method that combines error-correcting output coding with feature selection. When learning each classifier, h_ℓ, they apply feature selection techniques to choose the best features for learning that classifier. They obtained improvements in 7 out of 10 tasks with this approach.

2.1.4 Injecting Randomness

The last general purpose method for generating ensembles of classifiers is to inject randomness into the learning algorithm. In the backpropagation algorithm for training neural networks, the initial weights of the network are set randomly. If the algorithm is applied to the same training examples but with different initial weights, the resulting classifier can be quite different (Kolen & Pollack, 1991).

While this is perhaps the most common way of generating ensembles of neural networks, manipulating the training set may be more effective. A study by Parmanto, Munro, and Doyle (1996) compared this technique to bagging and to 10-fold cross-validated committees. They found that cross-validated committees worked best, bagging second best, and multiple random initial weights third best on one synthetic data set and two medical diagnosis data sets.

For the C4.5 decision tree algorithm, it is also easy to inject randomness (Kwok & Carter, 1990). The key decision of C4.5 is to choose a feature to test at each internal node in the decision tree. At each internal node, C4.5 applies a criterion known as the information gain ratio to rank-order the various possible feature tests. It then chooses the top-ranked feature-value test. For discrete-valued features with V values, the decision tree splits the data into V subsets, depending on the value of the chosen feature. For real-valued features, the decision tree splits the data into 2 subsets, depending on whether the value of the chosen feature is above or below a chosen threshold. Dietterich & Kong (1995) implemented a variant of C4.5 that chooses randomly (with equal probability) among the top 20 best tests. Table 2 compares a single run of C4.5 to ensembles of 200 classifiers constructed by bagging C4.5 and by injecting randomness into C4.5. The results show that injecting randomness obtains the best performance in three of the domains. In particular, notice that injected randomness obtains perfect test set performance in the letter recognition task.

Ali & Pazzani (1996) injected randomness into the FOIL algorithm for learning Prolog-style rules. FOIL works somewhat like C4.5 in that it ranks possible conditions to add to a rule using an information-gain criterion. Ali and Pazzani computed all candidate conditions that scored within 80% of the top-ranked candidate, and then applied a weighted random choice algorithm to choose among them. They compared ensembles of 11 classifiers to a single run of FOIL and found statistically significant improvements in 15 out of 29 tasks and statistically significant loss of performance in only one task. They obtained similar results using 11-fold cross-validation to construct the training sets.

Raviv and Intrator (1996) combine bootstrap sampling of the training data with injecting noise into the input features for the learning algorithm. To train each member of an ensemble of neural networks, they draw training examples with replacement from the original training data. The $\mathbf{x}$ values of each training example are perturbed by adding Gaussian noise to the input features. They report large improvements in a synthetic benchmark task and a medical diagnosis task.

A method closely related to these techniques for injecting randomness is the Markov Chain Monte Carlo (MCMC) method, which has been applied to neural networks by MacKay (1992) and Neal (1993) and to decision trees by

hipman, George and McCulloch (1996). The basic idea of the MCMC method (and related methods) is to construc a Markov process that generates an infinite sequence of hypotheses h_ℓ. In a Bayesian setting, the goal is to generate an hypothesis h_ℓ with probability $P(h_\ell|S)$, where S is the training sample. $P(h_\ell|S)$ is computed in the usual way as the (normalized) product of the likelihood $P(S|h_\ell)$ and the prior probability $P(h_\ell)$ of h_ℓ. To apply MCMC, we define a set of operators that convert one h_ℓ into another. For a neural network, such an operator might adjust one of the weights in the network. In a decision tree, the operator might interchange a parent and a child node in the tree or replace one node with another. The MCMC process works by maintaining a current hypothesis h_ℓ. At each step, it selects an operator, applies it (to obtain $h_{\ell+1}$), and then computes the likelihood of the resulting classifier on the training data. It then decides whether to keep $h_{\ell+1}$ or discard it and go back to h_ℓ. Under various technical conditions, it is possible to prove that a process of this kind will eventually converge to a stationary probability distribution in which the h_ℓ's are sampled in proportion to their posterior probabilities. In practice, it can be difficult to tell when this stationary distribution is reached. A standard approach is to run the Markov process for a long period (discarding all generated classifiers) and then collect a set of L classifiers from the Markov process. These classifiers are then combined by weighted vote according to their posterior probabilities.

2.1.5 Algorithm-Specific Methods for Generating Ensembles

In addition to these general-purpose methods for generating diverse ensembles of classifiers, there are several techniques that can be applied to the backpropagation algorithm for training neural networks.

Rosen (1996) trains several neural networks simultaneously and forces the networks to be diverse by adding a correlation penalty to the error function that backpropagation minimizes. Specifically, during training, Rosen keeps track of the correlations in the predictions of each network. He applies backpropagation to minimize an error function that is a sum of the usual squared output prediction error and a term that measures the correlations with the other networks. He reports substantial improvements in three simple synthetic tasks.

Opitz and Shavlik (1996) take a similar approach, but they employ a kind of genetic algorithm to search for a good population of neural network classifiers. In each iteration, they apply genetic operators to the current ensemble to generate new network topologies. These are then trained using an error function that combines the usual squared output prediction error with a multiplicative term that incorporates the diversity of the classifiers. After training the networks, they prune the population to retain the N best networks using a criterion that considers both accuracy and diversity. In a comparison with bagging, they found that their method gave excellent results in four real-world domains.

Abu-Mostafa (1990) and Caruana (1996) describe a technique for training a neural network on *auxiliary tasks* as well as on the main task. The key idea is to add some output units to the network whose role is to predict the values of the auxiliary tasks and to include the prediction error for these auxiliary tasks in the error criterion that backpropagation seeks to minimize. Because these auxiliary output units are connected to the same hidden units as the primary task outputs, the auxiliary outputs can influence the behavior of the network on the primary task. Parmanto, et al. (1994) show that diverse classifiers can be learned by training on the same primary task but with many different auxiliary tasks. One good source of auxiliary tasks is to have the network attempt to predict one of its input features (in addition to the primary output task). They apply this method to medical diagnosis problems.

More recently, Munro and Parmanto (1997) have developed an approach in which the value of the auxiliary output is determined dynamically by competition among the set of networks. Each network has a primary output y and a secondary (auxiliary) output z. During training, each network looks at the training example $\mathbf{x}_i$ and computes its primary and secondary output predictions. The network whose secondary output prediction is highest is said to be the "winner" for this example. It is given a target value for z of 1; the remaining networks are given a target value of 0 for the secondary output. All networks are given the value y_i as the target for the primary output. The effect is to encourage different networks to become experts at predicting the secondary output z in different regions of the input space. Because the primary and secondary outputs share the hidden layer, this causes the errors in the primary outputs to become decorrelated. They show that this method substantially out-performs an ensemble of ordinary networks trained using different initial random weights when trained on a synthetic classification task.

In addition to these methods for training ensembles of neural networks, there are also methods that are specific to decision trees. Buntine (1990) developed an algorithm for learning *option trees*. These are decision trees where an internal node may contain several alternative splits (each producing its own sub-decision tree). To classify an example, each of these sub-decision trees is evaluated, and the resulting classifications are voted. Kohavi & Kunz (1997) describe an option tree algorithm and compare its performance to bagged C4.5 trees. They show that option trees generally match the performance of bagging while producing a much more understandable result.

This completes my review of methods for generating ensembles using a single learning algorithm. Of course, one can always generate an ensemble by combining classifiers constructed by different learning algorithms. Learning

Algorithms based on very different principles will probably produce very diverse classifiers. However, often some of these classifiers perform much worse than others. Furthermore, there is no guarantee of diversity. Hence, when classifiers from different learning algorithms are combined, they should be checked (e.g., by cross-validation) for accuracy and diversity, and some form of weighted combination should be used. This approach has been shown to be effective in some applications (e.g., Zhang, Mesirov, & Waltz, 1992).

2.2 Methods for Combining Classifiers

Given that we have trained an ensemble of classifiers, how should we combine their individual classification decisions? Many methods have been explored. They can be subdivided into unweighted vote, weighted vote, and gating networks.

The simplest approach is to take an unweighted vote as is done in bagging, ECOC, and many other methods. While it may appear that more intelligent voting schemes should do better, the experience in the forecasting literature has been that simple, unweighted voting is very robust (Clemen, 1989). One refinement on simple majority vote is appropriate when each classifier h_ℓ can produce class probability estimates rather than a simple classification decision. A class probability estimate for data point $\mathbf{x}$ is the probability that the true class is k: $P(f(\mathbf{x}) = k|h_\ell)$, for $k = 1, \ldots, K$. We can combine the class probabilities of all of the hypotheses so that the class probability of the ensemble is $P(f(\mathbf{x}) = k) = \frac{1}{L}\sum_{\ell=1}^{L} P(f(\mathbf{x}) = k|h_\ell)$. The predicted class of $\mathbf{x}$ is then the class having the highest class probability.

Many different weighted voting methods have been developed for ensembles. For regression problems, Perrone & Cooper (1993) and Hashem (1993) apply least squares regression to find weights that maximize the accuracy of the ensemble on the training data. They show that the weight applied to h_ℓ should be inversely proportional to the variance of the estimates of h_ℓ. A difficulty with applying linear least squares is that the various hypotheses h_ℓ can be very highly correlated. They describe methods for choosing less correlated subsets of the ensemble and combining them by linear least squares.

For classification problems, weights are usually obtained by measuring the accuracy of each individual classifier h_ℓ on the training data (or a holdout data set) and constructing weights that are proportional to those accuracies. Ali and Pazzani (1996) describe a method that they call likelihood combination in which they apply the Naive Bayes algorithm (see Section 5.2 below) to learn weights for classifiers. In AdaBoost, the weight for classifier h_ℓ is computed from the accuracy of h_ℓ measured on the weighted training distribution that was used to learn h_ℓ. A Bayesian approach to weighted vote is to compute the posterior probability of each h_ℓ. This method requires the definition of a prior distribution $P(h_\ell)$ which is multiplied with the likelihood $P(S|h_\ell)$ to estimate the posterior probability of each h_ℓ. Ali and Pazzani experiment with this method. Earlier work on Bayesian voting of decision trees was performed by Buntine (1990).

The third approach to combining classifiers is to learn a gating network or a gating function that takes as input $\mathbf{x}$ and produces as output the weights w_ℓ to be applied to compute the weighted vote of the classifiers h_ℓ. Jordan and Jacobs (1994) learn gating networks that have the form

$$\begin{aligned} w_\ell &= \frac{e^{z_\ell}}{\sum_u e^{z_u}} \\ z_\ell &= v_\ell^T \mathbf{x} \end{aligned}$$

In other words, z_ℓ is the dot product of a parameter vector v_ℓ with the input feature vector $\mathbf{x}$. The output weight w_ℓ is then the so-called *soft-max* of the individual z_ℓ's. As with any learning algorithm, there is a risk of overfitting the training data by learning the gating function in addition to learning each of the individual classifiers. (Below we will discuss the hierarchical mixture of experts method developed by Jordan and Jacobs that learns the h_ℓ and the gating network simultaneously.)

A fourth approach to combining classifiers, called *stacking,* works as follows. Suppose we have L different learning algorithms $A_1, \ldots, A_L$ and a set S of training examples $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$. As usual, we apply each of these algorithms to our training data to produce hypotheses $h_1, \ldots, h_L$. The goal of stacking is to learn a good combining classifier h^* such that the final classification will be computed by $h * (h_1(\mathbf{x}), \ldots, h_L(\mathbf{x}))$. Wolpert (1992) proposed the following scheme for learning h^* using a form of leave-one-out cross-validation.

Let $h_\ell^{(-i)}$ be a classifier constructed by algorithm A_ℓ applied to all of the training examples in S except example i. In other words, each algorithm is applied to the training data m times, leaving out one training example each time. We can then apply each classifier $h_\ell^{(-i)}$ to example $\mathbf{x}_i$ to obtain the predicted class $\hat{y}_i^\ell$. This gives us a new data set containing "level 2" examples whose features are the classes predicted by each of the L classifiers. Each example has the form $\langle(\hat{y}_i^1, \hat{y}_i^2, \ldots, \hat{y}_i^L), y_i\rangle$. Now we can apply some other learning algorithm to this level 2 data to learn h^*. Breiman (1996b) applied this approach to combining different forms of linear regression with very good results.

2.3 Why Ensembles Work

I have already given the basic intuition for why ensembles can improve performance: uncorrelated errors made by the individual classifiers can be removed by voting. But there is a deeper question lurking here: Why should it be possible to find ensembles of classifiers that make uncorrelated errors? And there is another question as well: Why shouldn't we be able to find a single classifier that performs as well as the ensemble?

There are at least three reasons why good ensembles can be constructed and why it may be difficult or impossible to find a single classifier that performs as well as the ensemble. To understand these reasons, we must consider the nature of machine learning algorithms. Machine learning algorithms work by searching a space of possible hypotheses $\mathcal{H}$ for the most accurate hypothesis (that is, the hypothesis that best approximates the unknown function f). Two important aspects of the hypothesis space $\mathcal{H}$ are its size and whether it contains good approximations to f.

If the hypothesis space is large, then we will need a large amount of training data to constrain the search for good approximations. Each training example rules out (or makes less plausible) all those hypotheses in $\mathcal{H}$ that misclassify it. In a 2-class problem, ideally each training example can eliminate half of the hypotheses in $\mathcal{H}$, so we require $O(\log|\mathcal{H}|)$ examples to select a unique classifier from $\mathcal{H}$.

The first "cause" of the need for ensembles is that the training data may not provide sufficient information for choosing a single best classifier from $\mathcal{H}$. Most of our learning algorithms consider very large hypothesis spaces, so even after eliminating hypotheses that misclassify training examples, there are many hypotheses remaining. All of these hypotheses appear equally accurate with respect to the available training data. We may have reasons for preferring some of these hypotheses over others (e.g., preferring simpler hypotheses or hypotheses with higher prior probability), but nonetheless, there are typically many plausible hypotheses. From this collection of surviving hypothesis in $\mathcal{H}$, we can easily construct an ensemble of classifiers and combine them using the methods described above.

A second "cause" of the need for ensembles is that our learning algorithms may not be able to solve the difficult search problems that we pose. For example, the problem of finding the smallest decision tree that is consistent with a set of training examples is NP-hard (Hyafil & Rivest, 1976). Hence, practical decision tree algorithms employ search heuristics to guide a greedy search for small decision trees. Similarly, finding the weights for the smallest possible neural network consistent with the training examples is also NP-hard (Blum & Rivest, 1988). Neural network algorithms therefore employ local search methods (such as gradient descent) to find locally optimal weights for the network. A consequence of these imperfect search algorithms is that even if the combination of our training examples and our prior knowledge (e.g., preferences for simple hypotheses, Bayesian priors) determines a unique best hypothesis, we may not be able to find it. Instead, we will typically find an hypothesis that is somewhat more complex (or has somewhat lower posterior probability). If we run our search algorithms with a slightly different training sample or injected noise (or any of the other techniques described above), we will find a different (suboptimal) hypothesis. Ensembles can be seen therefore as a way of compensating for imperfect search algorithms.

A third "cause" of the need for ensembles is that our hypothesis space $\mathcal{H}$ may not contain the true function f. Instead, $\mathcal{H}$ may include several equally-good approximations to f. By taking weighted combinations of these approximations, we may be able to represent classifiers that lie outside of $\mathcal{H}$. One way to understand this is to visualize the decision boundaries constructed by learning algorithms. A decision boundary is a surface such that examples that lie on one side of the surface are assigned to a different class than examples that lie on the other side of the surface. The decision boundaries constructed by decision tree learning algorithms are line segments (or more generally, hyperplane segments) parallel to the coordinate axes. If the true boundary between two classes is a diagonal line, then decision tree algorithms must approximate that diagonal by a "staircase" of axis-parallel segments (see Figure 5). Different bootstrap training samples (or different weighted samples created by AdaBoost) will shift the locations of the staircase approximation, and by voting among these different approximations, it is possible to construct better approximations to the diagonal decision boundary.

Interestingly, these improved staircase approximations are equivalent to very complex decision trees. However, those trees are so large that were we to include them in our hypothesis space $\mathcal{H}$, the space would be far too large for the available training data. Hence, we can see that ensembles provide a way of overcoming representational inadequacies in our hypothesis space.

2.4 Open Problems Concerning Ensembles

Ensembles are well-established as a method for obtaining highly accurate classifiers by combining less accurate ones. There are still many questions, however, about the best way to construct ensembles as well as issues about how best to understand the decisions made by ensembles.

Faced with a new learning problem, what is the best approach to constructing and applying an ensemble of classifiers? In principle, there can be no single best ensemble method, just as there can be no single best learning

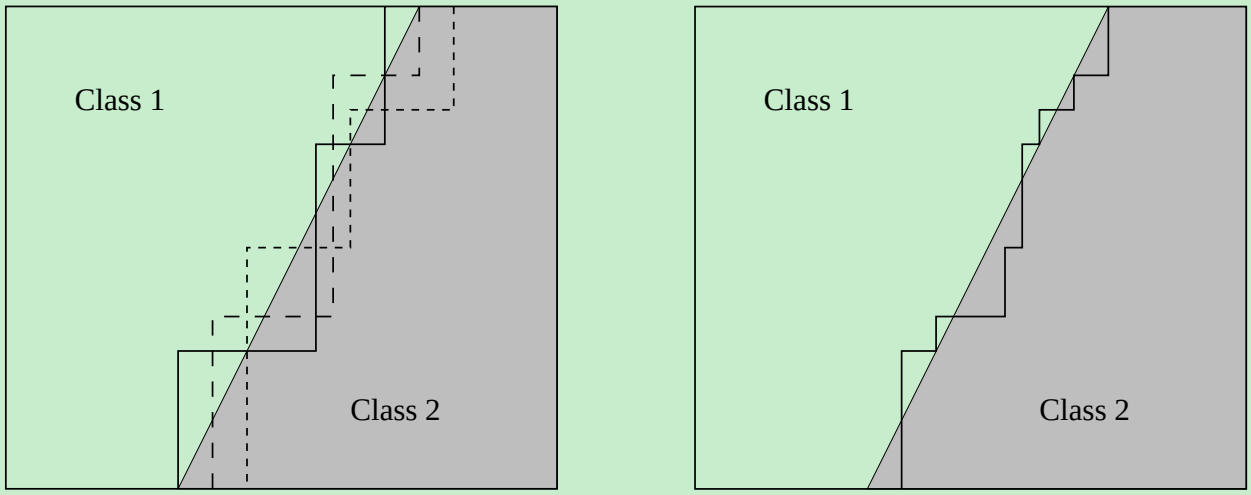

Figure 5: The left figure shows the true diagonal decision boundary and three staircase approximations to it (of the kind that are created by decision tree algorithms). The right figure shows the voted decision boundary, which is a much better approximation to the diagonal boundary.

algorithm. However, some methods may be uniformly better than others. And some methods may be better than others in certain situations.

Experimental studies have shown that AdaBoost is one of the best methods for constructing ensembles of decision trees. Schapire (1997) compares AdaBoost.M2 and AdaBoost.OC to bagging and error-correcting output coding and shows that the AdaBoost methods are generally superior. On the other hand, Quinlan (1996) has shown that in domains with very noisy training data, AdaBoost.M1 can perform very badly—it places high weight on incorrectly-labeled training examples and consequently constructs bad classifiers. Dietterich and Kong (1995) showed that combining bagging with error-correcting output coding improved the performance of both methods, which suggests that combinations of other ensemble methods should be explored as well. Dietterich and Kong also showed that error-correcting output coding does not work well with highly local algorithms (such as nearest neighbor methods).

There have been very few systematic studies of methods for constructing ensembles of neural networks, rule-learning systems, and other types of classifiers. Much work remains in this area.

While ensembles provide very accurate classifiers, there are problems that may limit their practical application. One problem is that ensembles can require large amounts of memory to store and large amounts of computation to apply. For example, earlier I mentioned that an ensemble of 200 decision trees attains perfect performance on a letter-recognition benchmark task. However, those 200 decision trees require 59 megabytes of storage, which makes them impractical for most present-day computers. An important line of research, therefore, is to find ways of converting these ensembles into less redundant representations, perhaps by deleting highly-correlated members of the ensemble or by representational transformations.

A second difficulty with ensemble classifiers is that an ensemble provides little insight into how it makes its decisions. A single decision tree can often be interpreted by human users, but an ensemble of 200 voted decision trees is much more difficult to understand. Can methods be found for obtaining explanations (at least locally) from ensembles? One example of work on this question is Craven's TREPAN algorithm (Craven & Shavlik, 1996).

3 Scaling Up Machine Learning Algorithms

A second major research area has explored techniques for scaling up learning algorithms so that they can apply to problems with millions of training examples, thousands of features, and hundreds of classes. Very large machine learning problems are beginning to arise in database mining applications, where there may be millions of transactions every day, and where it is desirable to have machine learning algorithms that can analyze such large data sets in just a few hours of CPU time. Another area where large learning problems arise is in information retrieval from full-text databases and the world-wide web. In information retrieval, each word in a document can be treated as an input feature, so each training example may be described by thousands of features. Finally, problems in speech

ecognition, object recognition, and character recognition for Chinese and Japanese present problems with hundreds or thousands of classes that must be discriminated.

3.1 Learning with Large Training Sets

Decision tree algorithms have been extended to handle large data sets in three different ways. One approach is based on intelligently sampling subsets of the training data as the tree is grown. To describe how this works, I must review how decision tree algorithms operate.

Decision trees are constructed by starting with the entire training set and an empty tree. A test is chosen for the root of the tree, and the training data are then partitioned into disjoint subsets depending on the outcome of the test. The algorithm is then applied recursively to each of these disjoint subsets. The algorithm terminates when all (or most) of the training examples within a subset of the data belong to the same class. At that point, a leaf node is created and labeled with that class.

The process of choosing a test for the root of the tree (or for the root of each subtree) involves analyzing the training data and choosing the one feature that is the best predictor of the output class. In a large and redundant data set, it may be possible to make this choice based on only a sample of the data. Musick, Catlett, and Russell (1992) presented an algorithm that dynamically chooses the sample based on how difficult the decision is at each node. The algorithm typically behaves by using a small sample near the root of the tree and then progressively enlarging the sample as the tree grows. This can reduce the time required to grow the tree without reducing the accuracy of the tree at all.

A second approach is based on developing clever data structures that avoid the need to store all training data in random access memory. The hardest step for most decision tree algorithms is to find tests for real-valued features. These tests usually take the form $x_j \leq \theta$, for some threshold value θ. The standard approach is to sort the training examples at the current node according to the values of feature x_j and then make a sequential pass over the sorted examples to choose the threshold θ. In standard depth-first algorithms, the data must be sorted by each candidate feature x_j at each node of the tree, which can become very expensive.

Shafer, Agrawal, and Mehta (1996) describe their SPRINT method in which the training data is broken up into a separate disk file for each attribute (and sorted by attribute value). For feature j, the disk file contains records of the form $\langle i, x_{i,j}, y_i \rangle$, where i is the index of the training example, $x_{i,j}$ is the value of feature j for training example i, and y_i is the class of example i. To choose a splitting threshold θ for feature j, it is a simple matter to make a serial scan of this disk file and construct class histograms for each potential splitting threshold. Once a value is chosen, the disk file is partitioned (logically) into two files, one containing examples with values less than or equal to θ, and the other containing examples with values greater than θ. As these files are written to disk, a hash table is built (in main memory) in which the index for each training example i is associated with the left or right child nodes of the newly created split. Then each of the disk files for the other attributes $x_l, l \neq j$ is read and split into a file for the left child and a file for the right child. The index of each example is looked up in the hash table to determine the child to which it belongs. SPRINT can be parallelized easily, and it has been applied to data sets containing 2.5 million examples. Mehta, Agrawal, and Rissanen (1996) have developed a closely-related algorithm, SLIQ, that makes more use of main memory but also scales to millions of training examples and is slightly faster than SPRINT. Both SPRINT and SLIQ scale approximately linearly in the number of training examples and the number of features aside from the cost of performing the initial sorting of the data (which need only be done once).

A third approach to large datasets is to take advantage of ensembles of decision trees (Chan & Stolfo, 1995). The training data can be randomly partitioned into N disjoint subsets. A separate decision tree can be grown from each subset in parallel. The trees can then vote to make classification decisions. Although the accuracy of each of the individual decision trees is less than the accuracy of a single tree grown with all of the data, the accuracy of the ensemble is often better than the accuracy of a single tree. Hence, with N parallel processors, we can achieve a speedup of N in the time required to construct the decision trees.

A fourth approach to the problem of choosing splits for real-valued input features is to "discretize" the values of such features. For example, one feature of a training example might be employee income measured in dollars. This could take on tens of thousands of distinct values, and the running time of most decision-tree learning algorithms is linear in the number of distinct values of each feature. This problem can be solved by grouping income into a small number of ranges (e.g., \$0–\$10,000, \$10,000–\$25,000, \$25,000–\$50,000, \$50,000–\$100,000, and greater than \$100,000). If the ranges are chosen well, the resulting decision trees will still be very accurate. Several simple and fast algorithms have been developed for choosing good discretization points (Catlett, 1991; Fayyad & Irani, 1993; Kohavi & Sahami, 1996) prior to running the decision tree algorithms.

A very effective rule-learning algorithm, called RIPPER, has been developed by William Cohen (1995) based on an earlier algorithm, IREP developed by Furnkranz and Widmer (1994). RIPPER constructs rules of the form

$test_1 \wedge test_2 \wedge \ldots \wedge test_l \Rightarrow c_i$, where each test has the form $x_j = \theta$ (for discrete features) or $x_j \leq \theta$ or $x_j > \theta$ (for real-valued features). A rule is said to *cover* a training example if the example satisfies all of the tests on the left-hand-side of the rule. For a training set of size m, the runtime of RIPPER scales as $O(m(\log m)^2)$. This is a major improvement over the rule-learning program C4.5RULES (Quinlan, 1993), which scales as $O(m^3)$.

Table 3 shows pseudo-code for RIPPER. RIPPER works by building an initial set of rules and optimizing the set of rules k times, where k is a parameter (typically set to 2). I describe RIPPER for the case where there are only two classes, 1 and 2. Examples of class 1 will be referred to as the positive examples, and examples of 2 will be referred to as the negative examples. RIPPER can easily be extended to handle larger numbers of classes.

To build a set of rules, RIPPER constructs one rule at a time. Before learning each rule, it divides the training data into a growing set (containing 2/3 of the data) and a pruning set (containing the remaining 1/3). It then iteratively adds tests to the rule until the rule covers no negative examples. Tests are selected via an information gain heuristic developed for Quinlan's FOIL system (Quinlan, 1990). Once a rule is grown, it is immediately pruned by deleting tests in reverse order, $test_l, test_{l-1}, \ldots, test_1$, to find the pruned rule that maximizes the quantity $(p-n)/(p+n)$, where p is the number of positive pruning examples covered by the pruned rule and n is the number of negative pruning examples covered by the pruned rule.

Once the rule has been grown and pruned, RIPPER adds it to the rule set and discards all training examples that are covered by this new rule. It employs a description length criterion to decide when to stop adding rules. The description length of a set of rules is the number of bits needed to represent the rules plus the number of bits needed to identify the training examples that are exceptions to the rules. Minimum description length criteria of this kind have been applied very successfully to rule- and tree-learning algorithms (e.g., Quinlan & Rivest, 1989). RIPPER stops adding rules when the description length of the rule set is more than 64 bits larger than the best description length observed so far. It then considers the rules in reverse order and deletes any rule that will reduce the total description length of the rule set.

To optimize a set of rules, RIPPER considers deleting each rule in turn and re-growing and re-pruning it. Two candidate replacement rules are grown and pruned. The first candidate is grown starting with an empty rule, whereas the second candidate is grown starting with the current rule. The better of the two candidates is selected (via a description length heuristic) and added to the rule set.

Cohen compared RIPPER to C4.5RULES on 37 data sets and found that RIPPER matched or beat C4.5RULES in 22 of the 37 problems. The rule sets that it finds are always smaller than those constructed by C4.5RULES. An implementation of RIPPER is available for research use from Cohen (`http://www.research.att.com/~wcohen/ripperd.html`).

3.2 Learning with Many Features

In many learning problems, there are hundreds or thousands of potential features describing each input object $\mathbf{x}$. Popular learning algorithms such as C4.5 and backpropagation do not scale well when there are many features. Indeed, from a statistical point of view, examples with many irrelevant, but noisy, input features provide very little information. It is easy for learning algorithms to be confused by the noisy features and construct poor classifiers. Hence, in practical applications it is wise to carefully choose which features to provide to the learning algorithm.

Research in machine learning has sought to automate the selection and weighting of features, and many different algorithms have been developed for this purpose. An excellent review has been written by Wettschreck, Aha, and Mohri (1997). A comprehensive review of the statistical literature on feature selection can be found in Miller (1990). I will discuss a few of the most significant methods here.

There are three main approaches that have been pursued. The first approach is to perform some initial analysis of the training data and select a subset of the features to feed to the learning algorithm. A second approach is to try different subsets of the features on the learning algorithm, estimate the performance of the algorithm with those features, and keep the subsets that perform best. The third approach is to integrate the selection and weighting of features directly into the learning algorithm. I will discuss two examples of each approach.

3.2.1 Selecting and Weighting Features by Preprocessing

A simple preprocessing technique is to compute the mutual information (also called the information gain) between each input feature and the class. The mutual information between two random variables is the average reduction in uncertainty about the second variable given a value of the first. For discrete feature j, the mutual information weight w_j can be computed as

$$w_j = \sum_v \sum_c P(y = c, x_j = v) \cdot \log \frac{P(y = c, x_j = v)}{P(y = c)P(x_j = v)},$$

Table 3: The RIPPER algorithm (Cohen, 1995)

```
procedure BuildRuleSet(P,N)
P = positive examples
N = negative examples
RuleSet = {}
DL = DescriptionLength(RuleSet, P, N)
while P ≠ {}
      // Grow and prune a new rule
      split (P, N) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
      Rule := GrowRule(GrowPos, GrowNeg)
      Rule := PruneRule(Rule, PrunePos, PruneNeg)
      add Rule to RuleSet
      if DescriptionLength(RuleSet, P, N) > DL + 64 then
            // Prune the whole rule set and exit
            for each rule R in RuleSet (considered in reverse order)
                  if DescriptionLength(RuleSet − {R}, P, N) < DL then
                        delete R from RuleSet
                        DL := DescriptionLength(RuleSet, P, N)
                        end if
                  end for
            return (RuleSet)
            end if
      DL := DescriptionLength(RuleSet, P, N)
      delete from P and N all examples covered by Rule
      end while
end BuildRuleSet

procedure OptimizeRuleSet(RuleSet, P, N)
for each rule R in RuleSet
      delete R from RuleSet
      UPos := examples in P not covered by RuleSet
      UNeg := examples in N not covered by RuleSet
      split (UPos, UNeg) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
      RepRule := GrowRule(GrowPos, GrowNeg)
      RepRule := PruneRule(RepRule, PrunePos, PruneNeg)
      RevRule := GrowRule(GrowPos, GrowNeg, R)
      RevRule := PruneRule(RevRule, PrunePos, PruneNeg)
      choose better of RepRule and RevRule and add to RuleSet
      end for
end OptimizeRuleSet

procedure Ripper(P, N, k)
RuleSet := BuildRuleSet(P, N)
repeat k times RuleSet := OptimizeRuleSet(RuleSet, P, N)
return (RuleSet)
end Ripper
```

where $P(y = c)$ is the proportion of training examples in class c, and $P(x_j = v)$ is the probability that feature j takes on value v. For real-valued features, the sums become integrals which must be approximated. A good approximation is to apply a discretization algorithm, such as the one advocated by Fayyad and Irani (1993), to covert the real-valued feature into a discrete-valued feature, and then apply the formula above. Wettschereck and Dietterich (1995) have obtained good results with nearest-neighbor algorithms using mutual information weighting.

A problem with mutual information weighting is that it treats each feature independently. For features whose predictive power is only apparent in combination with other features, mutual information will assign a weight of zero. For example, a very difficult class of learning problems involves learning parity functions with random irrelevant features. A parity function over n binary features is equal to 1 if-and-only-if an odd number of the features are equal to 1. Suppose we define a learning problem in which there are four relevant features and 10 irrelevant (random) binary features, and the class is the 4-parity of the four relevant features. The mutual information weights of all features will be approximately zero using the formula above.

An algorithm that overcomes this problem (and is one of the most successful preprocessing algorithms to date) is the RELIEF-F algorithm (Kononenko, 1994), which is an extension of an earlier algorithm called RELIEF (Kira & Rendell, 1992). The basic idea of these algorithms is to draw examples at random, compute their nearest neighbors, and adjust a set of feature weights to give more weight to features that discriminate the example from neighbors of

Table 4: The RELIEF-F algorithm.

procedure RELIEF-F(L, B)
L = the number of random examples to draw
B = the number of near neighbors to compute
for all features j: $w_j := 0.0$
$p_c :=$ the fraction of the training examples belonging to class c
for $l := 1$ **to** L **do**
    randomly select an instance $(\mathbf{x}_t, y_t)$
    let Hit be the set of B examples $(\mathbf{x}_i, y_i)$ nearest to $\mathbf{x}_t$ such that $y_i = y_t$.
    for each class $c \neq y_t$
        let M_c be a set of B examples $(\mathbf{x}_i, y_i)$ nearest to $\mathbf{x}_t$ such that $y_i = c$.
        end for
    for each feature j
        $$w_j := w_j - \frac{1}{LB} \sum_{(\mathbf{x}_i, y_i) \in Hit} \delta(x_{tj}, x_{ij}) + \sum_{c \neq y_t} \frac{p_y}{(1 - p_c) LB} \sum_{(\mathbf{x}_i, y_i) \in M_c} \delta(x_{tj}, x_{ij})$$
        end for j
    end for l
return $w_j \ \forall \, j$.
end RELIEF-F

different classes. Specifically, let $\mathbf{x}$ be a randomly-chosen training example, and let $\mathbf{x}^s$ and $\mathbf{x}^d$ be the two training examples nearest to $\mathbf{x}$ (in Euclidean distance) in the same class and in a different class, respectively. The goal of RELIEF is to set the weight w_j on input feature j to be

$$w_j = P(x_j \neq x_j^d) - P(x_j \neq x_j^s).$$

In other words, the weight w_j should be maximized when $\mathbf{x}^d$ has a high probability of taking on a different value for feature j and $\mathbf{x}^s$ has a low probability of taking on a different value for feature j. RELIEF-F computes a more reliable estimate of this probability difference by computing the B nearest neighbors of $\mathbf{x}$ in each class.

Table 4 describes the RELIEF-F algorithm. In this algorithm, $\delta(u, v)$ for two feature values u and v is defined as follows:

$$\delta(u, v) = \begin{cases} |u - v| & \text{for real-valued features} \\ 0 & \text{if } u = v \\ 1 & \text{if } u \neq v \text{ for discrete features} \end{cases}$$

Kononenko, Šimec, and Robnik-Šikonja (1997) have shown that RELIEF-F is very effective at detecting relevant features, even when those features are highly dependent on other features. For the 4-parity problem mentioned above (with 10 irrelevant random features), RELIEF-F can correctly separate the four relevant features from the 10 irrelevant ones given 400 training examples. Kononenko computes the 10 nearest neighbors in each class (i.e., $B = 10$). In his experiments, he sets the number of sample points L to be equal to the number of training examples, but in large data sets, good results can be obtained from much smaller samples.

Kononenko, et al have also experimented with integrating RELIEF-F into a decision tree learning algorithm called ASSISTANT-R. They show that ASSISTANT-R is able to perform much better than the original ASSISTANT program (which uses mutual information to choose features) in domains with highly dependent features while giving essentially the same performance in domains with independent features.

3.2.2 Selecting and Weighting Features by Testing with the Learning Algorithm

John, Kohavi, and Pfleger (1994) describe a computationally expensive method that they call the *wrapper* method for selecting input features. The idea is to generate sets of features, run the learning algorithm using only those features, and evaluate the resulting classifiers via 10-fold cross-validation (or via a single holdout set). In 10-fold cross-validation, the training data are subdivided randomly into 10 disjoint equal-sized sets. The learning algorithm is applied 10 times, each time on a training set containing all but one of these subsets. The resulting classifier is tested on the one-tenth of the data that was held out. The performance of the 10 classifiers (on their 10 respective hold-out sets) is averaged to provide an estimate of the overall performance of the learning algorithm when trained with the given features.

Kohavi and John explored step-wise selection algorithms that start with a set of features (e.g., the empty set), and considered adding or deleting a single feature. The possible changes to the feature set are evaluated (via 10-fold cross-validation), and the best change is made. Then, a new set of changes is considered. This method is

nly practical for data sets with relatively small numbers of features and very fast learning algorithms, but it gav excellent results on the UC Irvine benchmarks.

Moore and Lee (1994) describe a much more efficient approach to feature selection that combines leave-one-out cross-validation (LOOCV) with the nearest neighbor algorithm. In leave-one-out cross-validation, each training example is temporarily deleted from the training data and the nearest neighbor learning algorithm is applied to predict the class of that example. The total number of classification errors is the leave-one-out cross-validated estimate of the error rate of the learning algorithm. Moore and Lee use the LOOCV error to compare different sets of features with the goal of finding the set of relevant features that minimizes the LOOCV error.

They combine two very clever ideas to achieve this. The first idea is called *racing*. Suppose we are considering two different sets of relevant features, A and B. We repeatedly choose a training example at random, temporarily delete it from the training set, and apply the nearest neighbor rule to classify it using features in set A and the features in set B. We count the number of classification errors corresponding to each set. As we process more and more training examples in this leave-one-out fashion, the error rate for feature set A may become so much larger than the error rate for B that we can conclude with high confidence that B is the better feature set and terminate the race. In their (1994) paper, they apply Bayesian statistics to make this termination decision.

The second idea is based on *schemas*. We can represent each set of relevant features by a bit vector where a 1 in position j means that feature j is relevant, and a 0 means it is irrelevant. A schema is a vector containing 0's, 1's, and $\star$'s. A $\star$ in position j means that this feature should be randomly selected to be relevant 50% of the time. Moore and Lee race pairs of schemas against one another as follows. A training example is randomly selected and temporarily deleted from the training set. The nearest neighbor algorithm is applied to classify it using each of the two schemas being raced. To classify an example using a schema, features indicated by a 0 are ignored, features indicated by a 1 are selected, and features indicated by a $\star$ are selected with probability 0.5. Suppose, for illustration, that we have 5 features. Moore and Lee begin by conducting 5 simultaneous pairwise races:

$1\star\star\star\star$	races against	$0\star\star\star\star$
$\star1\star\star\star$	races against	$\star0\star\star\star$
$\star\star1\star\star$	races against	$\star\star0\star\star$
$\star\star\star1\star$	races against	$\star\star\star0\star$
$\star\star\star\star1$	races against	$\star\star\star\star0$

All of the races are terminated as soon as one of the schemas is found to be better than its opponent. In the next iteration, all single bit refinements of the winning schema are raced against one another. For example, suppose the schema $\star1\star\star\star$ was the winner of the first race. Then the next iteration involves the following four pairwise races:

$11\star\star\star$	races against	$01\star\star\star$
$\star11\star\star$	races against	$\star10\star\star$
$\star1\star1\star$	races against	$\star1\star0\star$
$\star1\star\star1$	races against	$\star1\star\star0$

This continues until all of the $\star$'s are removed from the winning schema. Moore and Lee found that this method never misses important relevant features, even when the features are highly dependent. However, in some rare cases, the races may take a very long time to conclude. The problem appears to be that the algorithm can be very slow to replace a $\star$ with a 0. In contrast, the algorithm is very quick to replace a $\star$ with a 1. Moore and Lee therefore investigated an algorithm, called SCHEMATA+ that terminates each race after 2000 evaluations (in favor of the 0; using the race statistics to choose the feature least likely to be relevant). The median number of training examples that must be evaluated by SCHEMATA+ is only 13% of the number of evaluations required by greedy forward selection and only 11% of the number of evaluations required by greedy backward elimination while achieving the same levels of accuracy. An important direction for future research is to compare the accuracy and speed of SCHEMATA+ and RELIEF-F, to see which works better with various learning algorithms.

3.2.3 Integrating Feature Weighting into the Learning Algorithm

I now discuss two methods that integrate feature selection directly into the learning algorithm. Both of them have been shown to work well experimentally, and the second method, called WINNOW, works extremely well in problems with thousands of potentially relevant input features.

The first algorithm is called the Variable-kernel Similarity Metric or VSM method (Lowe, 1995). VSM is a form of Gaussian radial basis function method. To classify a new data point $\mathbf{x}_t$, it defines a multivariate Gaussian probability distribution φ centered on $\mathbf{x}_t$ with standard deviation σ. Each example $(\mathbf{x}_i, y_i)$ in the training data set

votes" for class y_i with an amount $\varphi(||\mathbf{x}_i - \mathbf{x}_t||; \sigma)$. The class with the highest vote is assigned to be the class $y_t = f(\mathbf{x}_t)$ of the data point $\mathbf{x}_t$.

The key to the effectiveness of VSM is that it learns a weighted distance metric for measuring the distance between the new data point $\mathbf{x}$ and each training point $\mathbf{x}_i$. VSM also adjusts the size σ of the Gaussian distribution depending on the local density of training examples in the neighborhood of $\mathbf{x}_t$.

In detail, VSM is controlled by a set of learned feature weights $w_1, \ldots, w_n$, a kernel radius parameter r, and a number of neighbors R. To classify a new data point $\mathbf{x}_t$, VSM first computes the weighted distances to the R nearest neighbors $(\mathbf{x}_i, y_i)$:

$$d_i = \sqrt{\sum_{j=1}^{n} w_j^2 (x_{tj} - x_{ij})^2}.$$

It then computes a kernel width σ from the average distance to the $R/2$ nearest neighbors:

$$\sigma = \frac{2r}{R} \sum_{i=1}^{R/2} d_i.$$

Finally, it computes the probability that the next point $\mathbf{x}_t$ belongs to each class c (the quantity v_i is the "vote" from training example i):

$$\begin{aligned} v_i &= \exp\left(-\frac{d_i^2}{2\sigma}\right) \\ P[f(\mathbf{x}_t) = c] &= \sum_{i=1}^{R} \frac{v_i [\![y_i = c]\!]}{v_i} \end{aligned}$$

VSM then guesses the class with the highest probability.

How does VSM learn the values of the feature weights and the kernel radius r? By performing gradient descent search to minimize the leave-one-out cross-validated accuracy of the VSM classifier. Lowe (1995) shows how to compute the gradient of the LOOCV error with respect to each of the weights w_j and the parameter r. Starting with initial values for these parameters, VSM computes the R nearest neighbors of each training examples $(\mathbf{x}_i, y_i)$. It then computes the gradient and performs a search in the direction of the gradient to minimize LOOCV error (while keeping this set of nearest neighbors fixed). The search along the direction of the gradient is called a *line search,* and there are several efficient algorithms available (Press, Flannery, Teukolsky, & Verrerling, 1992). Even though the weights are changing during the line search, the set of nearest neighbors (and the gradient) is not recomputed. Once the error is minimized along this direction of the gradient, the R nearest neighbors are recomputed, a new gradient is computed, and a new line search is performed. Lowe applied the conjugate gradient algorithm to select each new search direction, and he reports that generally only 5–30 line searches were required to minimize the LOOCV error. The resulting classifiers gave excellent results on two challenging benchmark tasks.

The last feature weighting algorithm I will discuss is the WINNOW algorithm developed by Littlestone (Littlestone, 1988). WINNOW is a linear threshold algorithm for 2-class problems with binary (i.e., 0/1-valued) input features. It classifies a new example $\mathbf{x}$ into class 2 if

$$\sum_j w_j x_j > \theta$$

and into class 1 otherwise. WINNOW is an online algorithm; it accepts examples one-at-a-time and updates the weights w_j as necessary. Pseudo-code for the algorithm is shown in Table 5.

WINNOW initializes its weights w_j to 1. It then accepts a new example $(\mathbf{x}, y)$ and applies the threshold rule to compute the predicted class y'. If the predicted class is correct $(y' = y)$, WINNOW does nothing. However, if the predicted class is wrong, WINNOW updates its weights as follows. If $y' = 0$ and $y = 1$, then the weights are too low, so for each feature $x_j = 1$, $w_j := w_j \cdot \alpha$, where α is a number greater than 1 called the *promotion parameter.* If $y' = 1$ and $y = 0$, then the weights were too high, so for each feature $x_j = 1$, it decreases the corresponding weight by setting $w_j := w_j \cdot \beta$, where β is a number less than 1 called the *demotion parameter.*

The fact that WINNOW leaves the weights unchanged when the predicted class is correct is somewhat puzzling to many people. It turns out to be critical for its successful behavior, both theoretically and experimentally. One explanation is that, like ADABOOST, this strategy focuses WINNOW's attention on its mistakes. Another explanation is that it helps prevent overfitting.

Littlestone's theoretical analysis of WINNOW introduced the worst-case mistake bound method. The idea is to assume that the true function $y = f(\mathbf{x})$ belongs to some set of classifiers $\mathcal{H}$ and derive a bound on the maximum

Table 5: The WINNOW algorithm

procedure WINNOW(α, β, θ)
$\alpha > 1$ is the promotion parameter
$\beta < 1$ is the demotion parameter
θ is the threshold
initialize $w_j := 1$ for all j
for each training example $(\mathbf{x}, y)$
    $z := \sum_j w_j \cdot x_j$
    $y' := \begin{cases} 0 & \text{if } z < \theta \\ 1 & \text{if } z \geq \theta \end{cases}$
    if $y' = 0$ and $y = 1$ **then**
        $w_j := w_j \cdot \alpha$ for each j such that $x_j = 1$
        end if
    else if $y' = 1$ and $y = 0$ **then**
        $w_j := w_j \cdot \beta$ for each j such that $x_j = 1$
        end if
    end for
end WINNOW

number of mistakes that WINNOW will make when an adversary is allowed to choose f and allowed to choose the order in which the training examples are presented to WINNOW. Littlestone proves the following result:

Theorem 1 *Let f be a disjunction of r out of its n input features. Then* WINNOW *will learn f and make no more than $2 + 3r(1 + \lg n)$ mistakes. (For $\alpha = 2$, $\beta = 1/2$, and $\theta = n$.)*

This theorem shows that the convergence time of WINNOW is linear in the number of *relevant* features r and only logarithmic in the total number of features n. Similar results hold for other values of the α, β, and θ parameters, which permits WINNOW to learn functions where u out of v of the features must be 1 and many other interesting classes of boolean functions.

WINNOW is an example of an *exponential update* algorithm. The weights of the relevant features grow exponentially, while the weights of the irrelevant features shrink exponentially. General results in computational learning theory have developed similar exponential update algorithms for many applications. A common property of these algorithms is that they excel when the number of relevant features is small compared to the total number of features.

WINNOW has been applied to several experimental learning problems. Blum (1997) describes an application to a calendar scheduling task. In this task, a calendar system is given a description of a proposed meeting (including the list of invitees and their properties). It must then predict the start-time, day-of-week, location, and duration of the meeting. There were 34 input features available, some with many possible values. Blum defined boolean features corresponding to all possible values of all pairs of the input features. For example, given two input features event-type and position-of-attendees, he would define a separate boolean feature for each legal combination, such as event-type = meeting and position-of-attendees = grad-student. This gave a total of 59,731 boolean input features. He then applied WINNOW with $\alpha = 3/2$ and $\beta = 1/2$. He modified WINNOW to prune (set to zero) weights that become very small (less than 0.00001).

WINNOW found that 561 of the Boolean features were actually useful for prediction. Its accuracy was better than that of the best previous classifier for this task (which employed a greedy forward selection algorithm to select relevant features for a decision tree learning algorithm).

Golding and Roth (1996) describe an application of WINNOW to context-sensitive spelling correction. This is the task of identifying spelling errors where one legal word is substituted for another, such as *It's not to late*, where *to* is substituted for *too*. The Random House dictionary (Flexner, 1983) lists many sets of commonly-confused words, and Golding and Roth developed a separate WINNOW classifier for each of the listed sets (e.g., {to, too, two}). WINNOW's task is to decide whether each occurrance of these words is correct or incorrect based on its context.

Golding and Roth use two kinds of boolean input features. The first kind are context words. A context word is a feature that is true if a particular word (e.g., "cloudy") appears within 10 words before or after the target word. The second kind are collocation features. These test for a string of 2 words or part-of-speech tags immediately adjacent to the target word. For example, the sequence "$\langle target \rangle$ to VERB" is a collocation feature that checks whether the target word is immediately followed by the word "to" and then a word that can *potentially* be a verb (according to a dictionary lookup). Based on the 1-million word Brown corpus (Kučera & Francis, 1967), Golding and Roth defined more than 10,000 potentially relevant features.

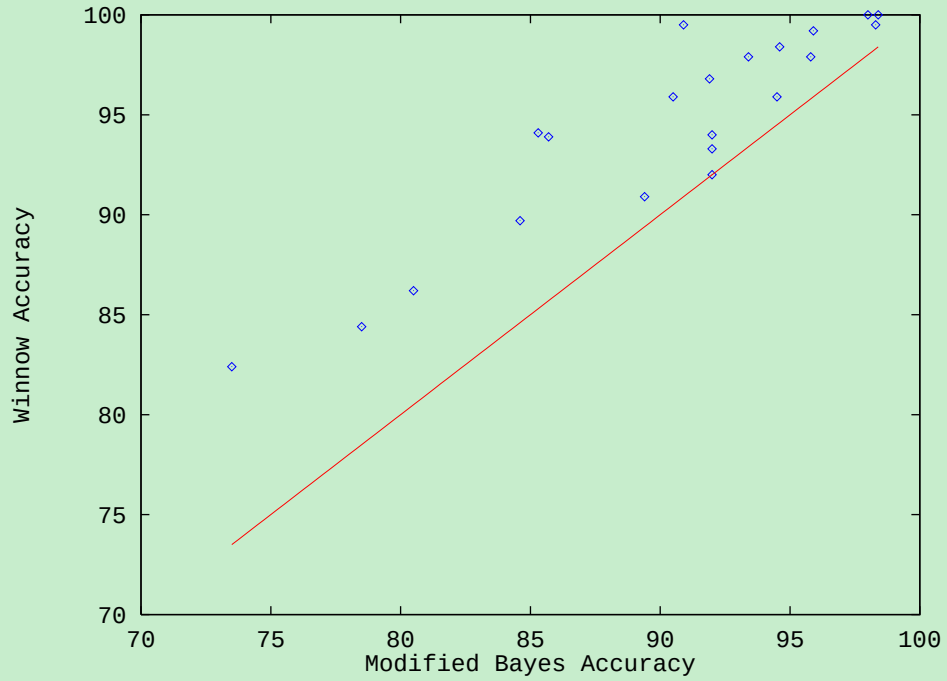

Figure 6: Comparison of the percentage of correct classifications for a modified Bayesian method and WINNOW for 21 sets of frequently-confused words. Trained on 80% of the Brown corpus and tested on the remaining 20%. Points lying above the line $y = x$ correspond to cases where WINNOW is more accurate.

Golding and Roth applied WINNOW (with $\alpha = 3/2$, β varying between 0.5 and 0.9, and $\theta = 1$). They compared its accuracy to the best previous method, which is a modified naive Bayesian algorithm. Figure 6 shows the results for 21 sets of frequently-confused words.

An important advantage of WINNOW, in addition to its speed and ability to operate online, is that it can adapt rapidly to changes in the target function. This was shown to be very important in the calendar scheduling problem, where the scheduling of different types of meetings may shift when semesters change (or during summer break). Blum's experiments showed that WINNOW was able to respond quickly to such changes. By comparison, to enable the decision tree algorithm to respond to changes, it was necessary to decide which old training examples could be deleted. This is difficult to do, because while some kinds of meetings may change with the change in semesters, other meetings may stay the same. A decision to keep, for example, only the most recent 180 training examples, means that examples of rare meetings whose scheduling does not change will be lost, which hurts the performance of the decision tree approach. In contrast, because WINNOW only revises the weights on features when they have the value 1, features describing such rare meetings are likely to retain their weights even as the weights for other features are being modified rapidly.

3.3 Summary: Scaling Up Learning Algorithms

This concludes my review of methods for scaling up learning algorithms to apply to very large problems. With the techniques described here, problems having one million training examples can be solved in reasonable amounts of computer time. However, it is not clear whether the current stock of ideas will permit the solution of problems with billions of training examples. An important open problem is to gather more practical experience with very large problems, so that we can understand their properties and determine where these algorithms fail.

A recurring theme is the use of subsamples of the training data to make critical intermediate decisions (such as the choice of relevant features). Another theme is the development of efficient online algorithms, such as WINNOW. These are "anytime" algorithms that can produce a useful answer regardless of how long they are permitted to run. The longer they run, the better the result that they produce.

An important open topic is the problem of handling thousands of output classes. Section 2 has already described the two methods that are most appropriate in this case: error-correcting output coding and ADABOOST.OC. Both of these methods should scale well with the number of classes. Error-correcting output coding has been tested on

problems with up to 126 classes, but tests on very large problems with thousands of classes have not yet been performed.

4 Reinforcement Learning

The previous two sections have discussed problems in supervised learning from examples. This section addresses problems of sequential decision making and control that come under the heading of *reinforcement learning.*

Work in reinforcement learning dates back to the earliest days of artificial intelligence when Arthur Samuel developed his famous checkers program (Samuel, 1959). More recently, there have been several important advances in the practice and theory of reinforcement learning. Perhaps the most famous work is Gerry Tesauro's (1992) TD-gammon program, which has learned to play backgammon better than any other computer program and almost as well as the best human players. Two other interesting applications are the work of Zhang and Dietterich (1995) on job shop scheduling and Crites and Barto (1995) on real-time scheduling of passenger elevators.

Kaelbling, Littman, and Moore (1996) have published an excellent survey of reinforcement learning and Mahadevan and Kaelbling (1996) report on a recent NSF-sponsored workshop on the subject. Two new books (Barto & Sutton, 1997; Bertsekas & Tsitsiklis, 1996) describe the newly-developed reinforcement learning algorithms and the theory behind them. I will summarize these developments here.

4.1 An Introduction to Dynamic Programming

The most important insight of the past five years is that reinforcement learning is best analyzed as a form of online, approximate dynamic programming (Barto, Bradtke, & Singh, 1995). I will introduce this insight using the following notation. Consider a robot interacting with an external environment. At each time t, the environment is in some state s_t, and the robot has available some set of actions A. The robot executes an action a_t, which causes the environment to move to a new state s_{t+1}. A convenient way of specifying the desired behavior of the robot is to define an *immediate reward function* $R(s_t, a, s_{t+1})$ that specifies a real-valued reward for this transition from s_t to s_{t+1}. For example, we can assign a positive reward to actions that reach a desired goal location and we can assign a negative reward to undesirable actions such as colliding with walls, people, or other robots. The immediate reward in all other states could be defined to be zero.

Our long-term goal for the robot can then be defined as some function of the immediate rewards it receives. A commonly-used criterion is the *cumulative discounted reward*,

$$\sum_{t=0}^{\infty} \gamma^t R(s_t, a, s_{t+1}),$$

where $0 \leq \gamma < 1$ is a *discount factor* that controls the relative importance of short-term and long-term rewards.

A procedure or rule for choosing each action a given state s is called the *policy* of the robot, and it can be formalized as a function $a = \pi(s)$. The goal of reinforcement learning algorithms is to compute the *optimal policy*, denoted π^*, which maximizes the cumulative discounted reward.

Researchers in dynamic programming (e.g., Bellman, 1957) found it convenient to define a real-valued function $f^\pi(s)$ called the *value function* of policy π. The value function $f^\pi(s)$ gives the expected cumulative discounted reward that will be received by starting in state s and executing policy π. It can be defined recursively by the formula,

$$f^\pi(s) = \sum_{s'} P(s'|s, \pi(s)) \cdot [R(s, \pi(s), s') + \gamma f^\pi(s')], \tag{1}$$

where $P(s'|s, \pi(s))$ is the probability that the next state will be s' given that the current state is s and we take action $\pi(s)$.

Given a policy π, a reward runction R, and the transition probability function P, it is possible to compute the value function f^π by solving the system of linear equations containing one equation of the form of Equation (1) for each possible state s. The system of equations can be solved via standard methods, such as Gaussian Elimination or Gauss-Seidel iteration or it can be solved iteratively by converting the equation into an assignment statement:

$$f^\pi(s) := \sum_{s'} P(s'|s, \pi(s)) \cdot [R(s, \pi(s), s') + \gamma f^\pi(s')]. \tag{2}$$

This assignment statement is called a *simple backup*, because it can be viewed as taking the current estimated value(s) $f^\pi(s')$ and "backing them up" to compute a revised estimate for $f^\pi(s)$. An example is shown in Figure 7. In state s,

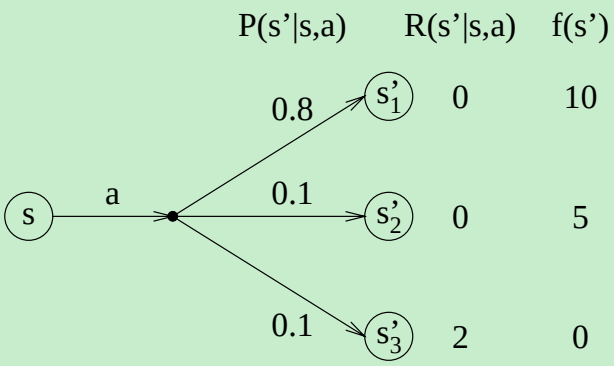

Figure 7: An example of a simple backup. Each arc is labeled with the probability of making that transition. The resulting states are labeled with their associated immediate reward and their value.

Table 6: The Policy Iteration Algorithm

---

procedure POLICYITERATION(P, R)
let π be an arbitrary initial policy
repeat until π is unchanged
    perform simple backups to compute f^π for each state (Equation 2).
    update π for each state (Equation 3).
    end

---

we perform action $a = \pi(s)$. There are three possible resulting states, s'_1, s'_2, and s'_3, with probabilities 0.8, 0.1, and 0.1 (respectively). The immediate rewards for reaching these states are 0, 0, and 2 (respectively) and the estimated values $f(s')$ of the states are 10, 5, and 0 (respectively). Assuming $\gamma = 1$, the new estimated value of $f^{\pi(s)}$ is

$$\begin{aligned} f^\pi(s) &:= (0 + 0.8 \cdot 10) + (0 + 0.1 \cdot 5) + (2 + 0.1 \cdot 0) \\ &:= 10.5. \end{aligned}$$

To compute the value of a policy, the simple backups can be performed in any convenient order—even at random. The only requirement is that γ must be small enough so that the sum of the expected discounted rewards converges and the simple backups must be performed until $f^\pi(s)$ converges for every state s.

Given a value function f^π, it is possible to compute a new policy π' that is guaranteed to be at least as good π. This is accomplished by performing a one-step lookahead search and choosing the action whose backed-up value is the largest:

$$\pi'(s) := \operatorname*{argmax}_a \sum_{s'} P(s'|s,a) \cdot [R(s,a,s') + \gamma f^\pi(s')]. \tag{3}$$

In words, we consider each possible action a, compute its expected backed-up value, and define $\pi'(s)$ to be the action that gives the maximum backed-up value.

By alternately computing the value f^π of policy π and then updating π via Equation (3), we can converge to the optimal policy π^* and its optimal value function f^*. Note that if we plug f^* into Equation (3), the new policy is unchanged—it is still the optimal policy. This defines the algorithm known as *policy iteration*, shown in Table 6. It is easy to show that policy iteration converges in a fixed number of iterations. Unfortunately, each iteration can be expensive, because it requires computing the value of the current policy.

An alternative to policy iteration is to work directly with the value function. Bellman proved that the value function f^* of the optimal policy π^* is the unique fixed point of the *Bellman Equation*:

$$f(s) := \max_a \sum_{s'} P(s'|s,a) \cdot [R(s,a,s') + \gamma f(s')]. \tag{4}$$

In other words, we perform a one-step lookahead just as we did for the policy improvement step of policy iteration, but instead of remembering the best action, we update our estimate of the value of state s. This is called a *Bellman Backup*. It is more expensive than a simple backup, because we must consider each possible action a and then each

Table 7: The Value Iteration Algorithm

procedure VALUEITERATION(P, R)
let f be an arbitrary initial value function
repeat until f is unchanged in all states
    for each state s, perform a Bellman backup (Equation 4)
    end
for each state s, compute the optimal policy (Equation 3)
end VALUEITERATION

possible resulting state s'. By performing enough Bellman backups in every state, we can converge to the optimal value function f^*. This is called the *Value Iteration* algorithm, and it is summarized in Table 7.

Unfortunately, value iteration can be more difficult than policy iteration because (a) each backup is a more expensive Bellman backup rather than a simple backup and (b) the value function may take a very long time to converge. Indeed, it is possible for the optimal policy to have converged long before the value function converges.

A hybrid algorithm that combines aspects of both value iteration and policy iteration is called *Modified Policy Iteration*. This algorithm is essentially the same as Policy Iteration except that only a fixed number of simple backups are performed in each state in each iteration. This means that the estimated value function for the current policy f^π does not completely converge to the correct value function, but under fairly mild conditions, it can be shown that the algorithm will still converge to the optimal policy.

All three of these algorithms—policy iteration, value iteration, and modified policy iteration—require performing backups in every state, so their running time scales with the number of states. In fact, value iteration may run for an infinite amount of time without converging. Policy iteration requires time at least $O(n^3)$ for problems with n states just to compute the value of the policy in each iteration. For small problems, this is not a difficulty, but for problems of interest in artificial intelligence, the state space often has 10^{20} or 10^{40} possible states, which renders these algorithms infeasible. Bellman termed this the *curse of dimensionality*, because the number of states, and hence the running time, increases exponentially with the number of dimensions in the state space.

Another drawback of these algorithms is that they require a complete "model" of the system, by which I mean the transition probabilities $P(s'|s,a)$ and the reward function $R(s,a,s')$. There are many applications where this model is unavailable (e.g., in a robot interacting with an unknown environment) or where the model cannot easily be converted into a transition probability matrix. For example, in the elevator control problem studied by Crites and Barto (1995), a software simulator is available that can take a current state s and a proposed action a and generate the next state s' according to the transition probability distribution. However, that distribution is not explicitly represented anywhere, so it is not available for direct use by a dynamic programming algorithm. The problem of constructing an explicit probability transition matrix and reward function has been called the *curse of modeling*, and in many problems it is just as severe than the curse of dimensionality. Reinforcement learning algorithms provide a way of overcoming these two curses.

Reinforcement learning algorithms have introduced three key innovations: (a) stochastic approximation of backups, (b) value function approximation, and (c) model-free learning. I will discuss these innovations in the context of an algorithm known as $TD(\lambda)$ developed by Sutton (1988).

4.2 Temporal Difference Learning and $TD(\lambda)$

I will begin by describing a simplified version of Sutton's $TD(\lambda)$ algorithm, called $TD(0)$. $TD(0)$ is a method for computing approximate simple backups online. Suppose we are in state s and we follow the current policy by taking action $a = \pi(s)$. If we are interacting with a real external environment (or with a simulator), the environment makes a probabilistic transition to a new state s' and produces the immediate reward $R(s,a,s')$. The $TD(0)$ algorithm observes this new state and reward and updates the value function as follows:

$$f^\pi(s) := (1-\alpha) \cdot f^\pi(s) + \alpha \cdot [R(s,a,s') + \gamma f^\pi(s')],$$

where α is a *learning rate parameter*. Typical values for α are between 0.01 and 0.5. (Technically, the α values must shrink to zero over time in order for $TD(0)$ to converge.)

The basic idea of $TD(0)$ is that if we visit s many times and apply action a many times, then by *sampling over time*, we will get the same effect as if we performed a simple backup. We can do this by sampling from the probability distribution $P(s'|s,a)$ rather than by having direct computational access to P.

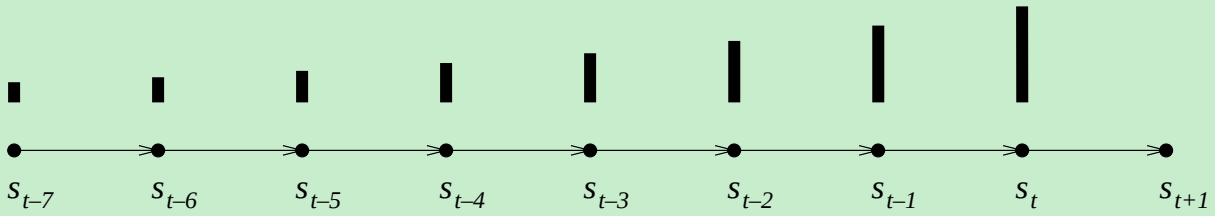

Figure 8: A sequence of states. The eligibility of each state (with $\lambda = 0.8$) is shown as a vertical bar.

This strategy allows $TD(0)$ to compute the value of a policy without having an explicit model. The environment serves as its own model!

When Sutton developed $TD(0)$, he also introduced a second idea. Instead of storing a separate value $f^\pi(s)$ for each state s, suppose we represent the value function as a neural network (or some other differentiable function approximator) of the form $f(s, W)$, where W is a vector of adjustable weights. With this representation, we can't directly assign a value to a state, but we can adjust the weights so that $f(s, W)$ is closer to the desired value. We do this by defining an error function:

$$J(W) = \frac{1}{2}\Big(f^\pi(s, W) - [R(s, a, s') + \gamma f^\pi(s', W)]\Big)^2.$$

This is the squared difference between the current estimated value of state s, $f^\pi(s, W)$, and the backed-up value. This is called the *temporal difference error*. Our goal is to modify W to reduce the temporal difference error $J(W)$. Differentiating $J(W)$ (and treating only the first occurrence of W as adjustable), we obtain the learning rule

$$W := W - \alpha \nabla_W f(s, W)\Big(f^\pi(s, W) - [R(s, a, s') + \gamma f^\pi(s', W)]\Big),$$

where $\nabla_W f(s, W)$ is the gradient of f with respect to the weights W. This takes a step of size α in the direction of the decreasing gradient scaled by the size of the temporal difference error.

By using a smoothly parameterized function approximation $f(s, W)$, $TD(0)$ can circumvent the curse of dimensionality provided that $f(s, W)$ can accurately approximate the true value function $f(s)$ with a small number of parameters W.

Sutton introduced one other wonderful idea in the $TD(\lambda)$ algorithm—the *eligibility trace*. Suppose we have visited a sequence of states s_1, s_2, ..., s_t, s_{t+1}, and we are updating the value of $f(s_t, W_t)$. Sutton suggested that we might want to update the values of the preceding states as well, since the key idea of dynamic programming is to propagate information about expected rewards backward through the state space. Sutton proposed that we should remember the gradient $\nabla_{W_{t-i}} f(s_{t-i}, W_{t-i})$ for each state s_{t-i}, for $i = 1, \ldots, n$, that we have visited. When we update $f(s_t, W_t)$ by taking a step in the direction of $-\nabla_{W_t} f(s_t, W_t)$, we will also take a smaller step in the direction of $-\nabla_{W_{t-1}} f(s_{t-1}, W_{t-1})$ and an even smaller step in the direction of $-\nabla_{W_{t-2}} f(s_{t-2}, W_{t-2})$ and so on. Each stepsize will be decreased by a factor of $\lambda < 1$. This gives us the learning rule

$$W := W - \alpha \left(\sum_{i=0}^{\infty} \lambda^i \nabla_{W_{t-i}} f(s_{t-i}, W_{t-i})\right)\Big(f^\pi(s_t, W) - [R(s_t, a, s_{t+1}) + \gamma f^\pi(s_{t+1}, W)]\Big).$$

The value of λ^i is called the *eligibility* of state s_{t-i}. Figure 8 shows the eligibility of a sequence of states as a bar graph.

The infinite sum $\sum_{i=0}^{\infty} \lambda^i \nabla_W f(s_{t-i}, W_{t-i})$ can be implemented by maintaining a *current gradient vector* G:

$$G_t := \lambda G_{t-1} + \nabla_{W_t} f(s_t, W_t).$$

With this change, we get the full algorithm $TD(\lambda)$ shown in Table 8. Readers familiar with the momentum method for stabilizing backpropagation will note that the eligibility trace mechanism is very similar. However, in the momentum method, previous *weight changes* are remembered, while in the eligibility trace, previous *gradient vectors* are remembered and future temporal differences determine the stepsize along those previous gradients.

There have been many theoretical studies of the behavior of $TD(\lambda)$. The most general results have been obtained by Tsitsiklis and Van Roy (1996). They analyze the case where the function approximator $f(s, W)$ is a linear

Table 8: The $TD(\lambda)$ algorithm for computing the value of a policy π

procedure TD($\pi, \lambda, \gamma, \alpha, s_0$)
initialize $G = \mathbf{0}$
initialize W randomly
s_0 is the starting state
while W has not converged **do**
    take action $a = \pi(s_t)$
    observe resulting state s_{t+1} and reward $R(s_t, a, s_{t+1})$
    $G := \lambda G + \nabla_W f(s_t, W)$
    $W := W - \alpha G \cdot \Big(f(s_t, W) - [R(s_t, a, s_{t+1}) + \gamma f(s_{t+1}, W)]\Big)$
    end while
return W which defines f^π
end TD

combination of fixed (and arbitrary) orthogonal basis functions. To analyze how well $f(s, W)$ can approximate f^π, some notion of the *distance* between two value functions is needed. Tsitsiklis and Van Roy define the following measure:

$$||f - f^\pi||_D = \left(\sum_s [f(s) - f^\pi(s)]^2 D(s)\right)^{1/2},$$

where $D(s)$ is the probability that the policy π visits state s.

Using this measure, let $\hat{f}^\pi$ be the best approximation of f^π that can be represented by a linear combination of the given basis functions. Tsitsiklis and Van Roy prove that $TD(\lambda)$ will converge to a function f such that

$$||f - f^\pi||_D \leq \frac{||\hat{f}^\pi - f^*||_D}{1 - \gamma(1-\lambda)/(1-\lambda\gamma)}.$$

The quantity on the left-hand side is the error between the function f learned by $TD(\lambda)$ and the true value function f^π for policy π. The numerator on the right-hand side is the inherent approximation error resulting from the use of a linear combination of the given, fixed basis functions. The denominator is the error that results from the fact that approximation errors in one state will be propagated backward to earlier states. For large λ, the denominator approaches 1 (no error), but for $\lambda = 0$, the denominator becomes $1 - \gamma$, which could be very small, and hence produce very large errors. For this result to hold, it is essential that the backups performed by $TD(\lambda)$ be performed according to the current policy π. If this condition is not observed, then $TD(\lambda)$ may fail to converge.

The $TD(\lambda)$ algorithm provides a way of computing the value of a fixed policy without direct access to the transition probabilities and reward function. However, this is only of limited utility unless we can perform the policy improvement step from Equation (3) and thereby implement the policy iteration algorithm. Unfortunately, policy improvement requires access to a model that can generate the possible next states and their probabilities.

4.3 Applications of $TD(\lambda)$

There are many domains where such a model is available. For example, in game-playing settings, such as backgammon, it is easy to compute the set of available moves from each state and the probabilities of all successor states. Hence, $TD(\lambda)$ can be combined with policy improvement to learn an optimal policy for backgammon. This is what Tesauro did in his famous TD-gammon system (Tesauro, 1992, 1995).

TD-gammon employs a neural network representation of the value of a state. The state of the backgammon game is described by a vector of 198 features that encode the locations of the pieces on the board and the values shown on the dice. TD-gammon begins with a randomly-initialized neural network. It plays a series of games against itself. At each step, it makes a full one-step lookahead search, applies the neural network to evaluate each of the resulting states, and makes the move corresponding to the highest backed up value. In short, it applies Equation (3) to compute the action to perform next. This is a form of local policy improvement. After making the move, it observes the resulting state and applies the $TD(\lambda)$ rule to update the value function. In effect, TD-gammon is executing a form of modified policy iteration where it alternates between one step of policy evaluation (the $TD(\lambda)$ update) and one step of policy improvement (the computation of the best move to make).

There is no guarantee that this algorithm will converge to the optimal policy. Indeed, it is easy to construct examples where the strategy of always performing the best action based on the current approximation to the value

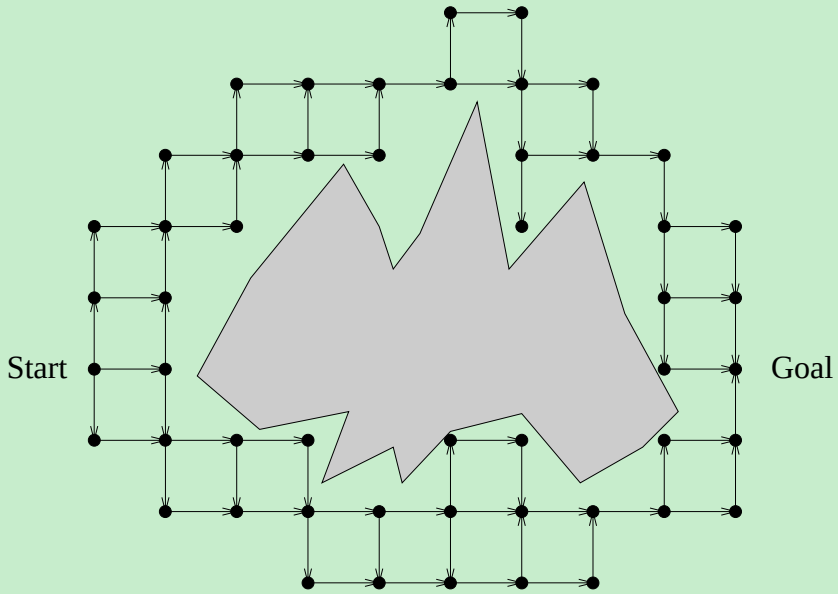

Figure 9: Two networks of roads around a mountain. Without exploration, an initial policy that follows the northern roads will never discover that the southern roads provide a shorter route to the goal.

function, $f(s, W)$, will lead to a local minimum. For example, consider the navigation problem shown in Figure 9. There is a large mountain separating the start state from the goal. A network of roads passes to the north of the mountain, and a similar (and shorter) network passes to the south. Suppose that our initial policy takes us to the north side and we eventually reach the goal. After updating our value function using $TD(\lambda)$, suppose that the north side still appears to be shorter than the south side (because our estimates for the values of the states along the south side are too high). Then, in future trials we will continue to take the north roads, and we will never try the southern route.

This is called the *problem of exploration*. The heart of the problem is that to find the optimal policy, it is necessary to prove that every off-policy action leads to worse expected results than the actions of the optimal policy. In this situation, it is essential to explore the southern path to determine whether it is worse (or better!) than the northern path. The strategy of always taking the action that appears to be optimal based on the current value function is called the *pure exploitation* strategy. The example in Figure 9 shows that the pure exploitation strategy will not always find the optimal policy. Hence, online reinforcement learning algorithms must balance exploitation with exploration.

Fortunately, in backgammon, the random dice rolls inject so much randomness into the game that TD-gammon thoroughly investigates the possible moves in the game. Experimentally, the performance of TD-gammon is outstanding. It plays much better than any other computer program, and it is nearly as good as the world's best players. The results of three versions of the program in three separate matches against human players are shown in Table 9. In some situations, the moves chosen by TD-gammon have been adopted by expert humans.

A similar strategy was applied by Zhang and Dietterich (1995) to the problem of job-shop scheduling. In job-shop scheduling, a set of tasks must be scheduled to avoid resource conflicts. Each task requires certain resources throughout its duration, and each task has prerequisite tasks that must be completed before it can be executed. An optimal schedule is one that completes all of its tasks in the minimum amount of time while satisfying all resource and prerequisite constraints.

Zweben, Daun, and Deale (1994) developed a repair-based search space for this task in which each state is a complete schedule (i.e., all tasks have assigned start times). The starting state is a *critical path* schedule in which every task is scheduled as early as possible subject to its prerequisite constraints and ignoring its resource constraints. The actions in this search space identify the earliest constraint violation and repair it by moving tasks later in time (thus lengthening the schedule). The search terminates when a violation-free schedule is found.

Zhang and Dietterich reformulated this as a reinforcement learning problem where the optimal policy will choose a sequence of repairs that will produce the shortest possible schedule. The immediate reward function gives a small cost to each repair action and a final reward that is inversely proportional to the final length of the schedule. Zhang

Table 9: Summary of the Performance of TD-gammon against some of the world's best players. The results are expressed in net points won (or lost) and in points won per game. Taken from Tesauro (1995).

Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	−13 pts/51 games (−0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey Snellings, Russell, Sylvester	−7 pts/38 games (−0.18 ppg)
TDG 2.1	1,500,000	Robertie	−1 pts/40 games (−0.02 ppg)

and Dietterich applied $TD(\lambda)$ with a feed-forward neural network to represent the value function. The actions in this domain are deterministic, so deliberate exploration is needed. Their system makes a random exploratory move with a given probability, β, which is gradually decreased during learning. After learning, their system finds schedules that are substantially shorter than the best previous method (for the same expenditure of CPU time). This approach of converting combinatorial optimization problems into reinforcement learning problems should be applicable to many other important industrial domains.

These two applications show that when a simulator is available for a task, it is possible to solve the reinforcement learning problem using $TD(\lambda)$ even in very large search spaces. However, in domains involving interaction with hard-to-model real-world environments (e.g., robot navigation, factory automation), some other method is needed. Two approaches have been explored.

One approach is to learn a predictive model of the environment by interacting with it. Each interaction with the environment provides a training example for supervised learning of the form $s' = env(s, a)$, where env is the environment, s and a are the current state and action, and s' is the resulting state. Standard supervised learning algorithms can be applied to learn this model, which can then be combined with $TD(\lambda)$ to learn an optimal policy.

The second approach is a model-free algorithm called Q-learning, developed by Watkins (Watkins, 1989; Watkins & Dayan, 1992), which is the subject of the next section.

4.4 Model-free Reinforcement Learning (Q-learning)

Q-learning is an online approximation of value iteration. The key to Q-learning is to replace the value function $f(s)$ with an *action-value function*, $Q(s, a)$. The quantity $Q(s, a)$ gives the expected cumulative discounted reward of performing action a in state s and then pursuing the current policy thereafter. Hence, the value of a state is the maximum of the Q values for that state:

$$f(s) = \max_a Q(s, a).$$

We can write down the Q version of the Bellman equation as follows:

$$Q(s, a) = \sum_{s'} P(s'|s, a) \left[R(s'|s, a) + \max_{a'} \gamma Q(s', a') \right].$$

The role of the Q function is illustrated in Figure 10, where it is contrasted with the value function f. In the left part of the figure, we see that the Bellman backup updates the value of $f(s)$, by considering the values $f(s')$ of states that result from different possible actions. In the right part of the figure, the analogous backup works by taking the best of the values $Q(s', a')$ and backing them up to compute an updated value for $Q(s, a)$.

The Q function can be learned by an algorithm that exploits the same insight as $TD(0)$—online sampling of the transition probabilities—and an additional idea—online sampling of the available actions. Specifically, suppose we have visited state s, performed action a, and observed the resulting state s' and immediate reward $R(s, a, s')$. We can update the Q function as follows:

$$Q(s, a) := (1 - \alpha)Q(s, a) + \alpha \cdot \Big(R(s, a, s') + \gamma \max_a \; Q(s', a) \Big).$$

Suppose that every time we visit state s, we choose the action a uniformly at random. Then the effect will be to approximate a full Bellman backup (see Equation 4). Each value $Q(s, a)$ will be the expected cumulative discounted reward of executing action a in s, and the maximum of these values will be $f(s)$. The random choice of a ensures that

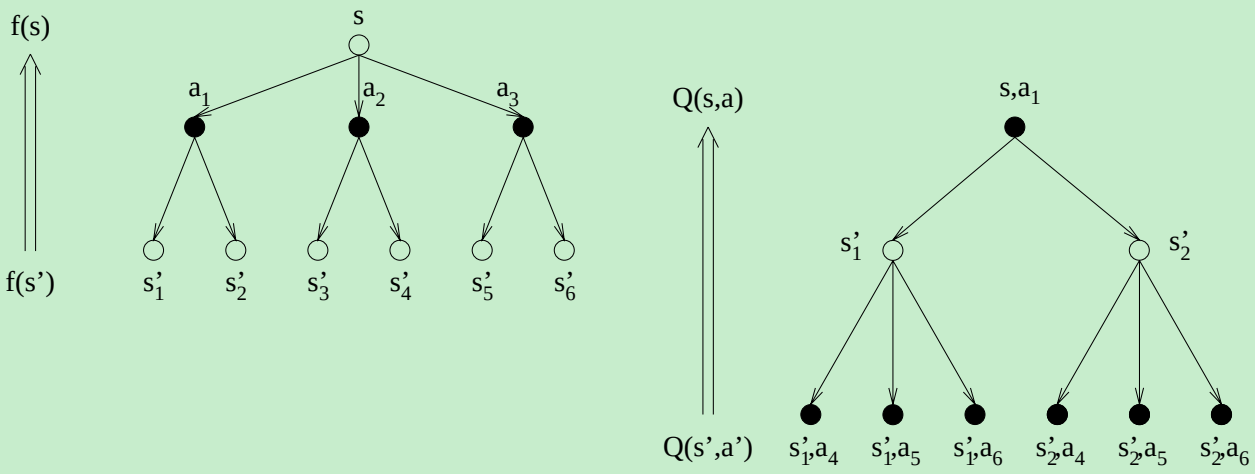

Figure 10: A comparison of the value function f and the Q function. Black nodes represent situations where the agent has chosen an action. White nodes are states where the agent has not yet chosen an action.

we learn Q values for every action available in state s. The online updates ensure that we experience the resulting states s' in proportion to their probabilities $P(s'|s,a)$.

In general, we can choose which actions to perform in any way we like, as long as there is a non-zero probability of performing every action in every state. Hence, a reasonable strategy is to choose the best action (i.e., the one whose estimated value $Q(s,a)$ is largest) most of the time but to choose a random action with some small probability. Watkins proves that as long as every action is performed in every state infinitely many times, the Q function will converge to the optimal function Q^* with probability 1.

Once we have learned Q^*, we must convert it into a policy. While this presented an insurmountable difficulty for $TD(0)$, it is trivial for the Q representation. The optimal policy can be computed as

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a).$$

Crites and Barto (1995) applied Q-learning to a problem of controlling four elevators in a 10-story office building during peak "down traffic." People press elevator call buttons on the various floors of the building to call the elevator to that floor. Once inside the elevator, they may also press destination buttons to request that the elevator stop at various floors. The elevator control decisions are (a) to decide, after stopping at a floor, which direction to go next, and (b) to decide, when approaching a floor, whether to stop at that floor or skip it. Crites and Barto applied rules to make the first decision, so the reinforcement learning problem is to learn whether to stop or skip floors. The goal of the controller is to minimize the square of the time that passengers must wait for the elevator to arrive after pressing the call button.

Crites and Barto used a team of four Q-learners, one for each of the four elevator cars. Each Q-function was represented as a neural network with 47 input features, 20 sigmoidal hidden units, and 2 linear output units (to represent $Q(s,\mathsf{stop})$ and $Q(s,\mathsf{skip})$). The immediate reward was the (negative of the) squared wait time since the previous action. They employed a form of random exploration in which exploratory actions are more likely to be chosen if they have higher estimated Q values.

Figure 11 compares the performance of the learned policy to that of eight heuristic algorithms, including the best non-proprietary algorithms. The left-hand graph shows the squared wait time, while the right-hand graph shows the percentage of passengers that had to wait more than 60 seconds for the elevator. The learned Q policy performs better than all of the other methods.

Another interesting application of Q learning is to the problem of assigning radio channels for cellular telephone traffic. Singh and Bertsekas (1997) showed that Q learning could find a much better policy than some quite sophisticated and complex published methods.

4.5 Open Problems in Reinforcement Learning

Many important problems remain unsolved in reinforcement learning, which reflects the relative youth of the field. I discuss a few of these problems here.

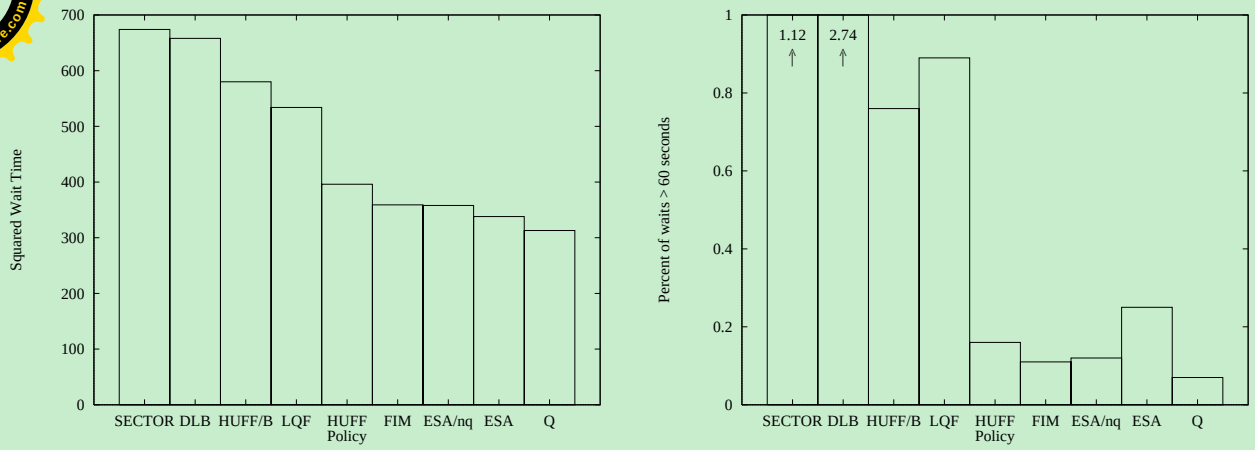

Figure 11: Comparison of learned elevator policy Q with eight published heuristic policies.

First, the use of multilayer sigmoidal neural networks for value function approximation has worked, but there is no reason to believe that such networks are well-suited to reinforcement learning. First, they tend to forget episodes (both good and bad), unless they are retrained on those episodes frequently. Second, the need to make small gradient descent steps makes learning very slow, particularly in the early stages. An important open problem is to clarify what properties an ideal value function approximator would possess and to develop function approximators with those properties. Initial research suggests that value function approximators should be "local averagers" that compute the value of a new state by interpolating among the values of previously visited states (Gordon, 1995).

A second key problem is to develop reinforcement methods for hierarchical problem solving. For very large search spaces, where the distance to the goal and the branching factor are big, no search method can work well. Often such large search spaces have a hierarchical (or approximately hierarchical) structure that can be exploited to reduce the cost of search. There have been several studies of ideas for hierarchical reinforcement learning (e.g., Singh, 1992; Dayan & Hinton, 1993; Kaelbling, 1993).

The third key problem is to develop intelligent exploration methods. Weak exploration methods that rely on random or biased random choice of actions cannot be expected to scale well to large, complex spaces. A property of the successful applications shown above (particularly backgammon and job-shop scheduling) is that even random search will reach a goal state and receive a reward. In domains where success is contingent on a long sequence of successful choices, random search has a very low probability of receiving any reward. More intelligent search methods, such as means-ends analysis, need to be integrated into reinforcement learning systems as they have been integrated into other learning architectures such as SOAR (Laird, Newell, & Rosenbloom, 1987) and PRODIGY (Minton, Carbonell, Knoblock, Kuokka, Etzioni, & Gil, 1989).

A fourth problem is that optimizing cumulative discounted reward is not always appropriate. In problems where the system needs to operate continuously, a better goal is to maximize the average reward per unit time. However, algorithms for this criterion are more complex and not as well-behaved. Several new methods have been put forward recently (Schwartz, 1993; Mahadevan, 1996; Ok & Tadepalli, 1996).

The fifth, and perhaps most difficult, problem is that existing reinforcement learning algorithms assume that the entire state of the environment is visible at each time step. This is not true in many applications, such as robot navigation or factory control, where the available sensors provide only partial information about the environment. A few algorithms for the solution of hidden-state reinforcement learning problems have been developed (Cassandra, Kaelbling, & Littman, 1994; Littman, Cassandra, & Kaelbling, 1995; McCallum, 1995; Parr & Russell, 1995). Exact solution appears to be very difficult. The challenge is to find approximate methods that scale well to large hidden-state applications.

Despite these very substantial open problems, reinforcement learning methods are already being applied to a wide range of industrial problems where traditional dynamic programming methods are infeasible. Researchers in the area are optimistic that reinforcement learning algorithms can solve many problems that have resisted solution by machine learning methods in the past. Indeed, the general problem of choosing actions to optimize expected utility is exactly the problem faced by general intelligent agents. Reinforcement learning provides one approach to attacking these problems.

Learning Stochastic Models

The final topic that I will discuss is the area of learning stochastic models. Traditionally, researchers in machine learning have sought very general-purpose learning algorithms—such as decision tree, rule, neural network, and nearest neighbor algorithms—that could efficiently search a large and flexible space of classifiers for a good fit to training data. While these algorithms are very general, they have a major drawback. In a practical problem where there is extensive prior knowledge, it can be quite difficult to incorporate this prior knowledge into these very general algorithms. A secondary problem is that the classifiers constructed by these general learning algorithms are often difficult to interpret—their internal structure may not have any correspondence to the real-world process that is generating the training data.

Over the past five years or so there has been tremendous interest in a more knowledge-based approach based on stochastic modeling. A *stochastic model* is a model that describes the real-world process by which the observed data are generated. Sometimes the terms *generative stochastic model* and *causal model* are used to emphasize this perspective. The stochastic model is typically represented as a probabilistic network—a graph structure that captures the probabilistic dependencies (and independencies) among a set of random variables. Each node in the graph has an associated probability distribution, and from these individual distributions, the joint distribution of the observed data can be computed. To solve a learning problem, the programmer designs the structure of the graph and chooses the forms of the probability distributions. This yields a stochastic model with many free parameters (i.e., the parameters of the node probability distributions). Given a training sample, learning algorithms can be applied to determine the values of the free parameters and thereby fit the model to the data. Once a stochastic model has been learned, probabilistic inference can be carried out to support tasks such as classification, diagnosis, and prediction.

More details on probabilistic networks are given in two recent textbooks: Jensen (1996) and Castillo, Gutierrez, and Hadi (1997).

5.1 Probabilistic Networks

Figure 12 is an example of a probabilistic network that might be used to diagnose diabetes. There are six variables (with their abbreviations):

- Age: Age of patient (A),
- Preg: Number of pregnancies (N),
- Mass: Body mass (M),
- Insulin: Blood insulin level (after a glucose tolerance test) (I),
- Glucose: Blood glucose level (after a glucose tolerance test) (G), and
- Diabetes: True if the patient has diabetes (D).

In a medical diagnosis setting, the first five variables would be observed and then the computer would estimate the probability that the patient has diabetes (i.e., estimate the probability that the Diabetes variable is true).

This network corresponds to the following decomposition of the joint probability distribution among the six variables:

$$P(A, N, M, I, G, D) = P(A) \cdot P(N) \cdot P(M|A, N) \cdot P(D|M, A, N) \cdot P(I|D) \cdot P(G|I, D).$$

Each node in the network corresponds to a probability distribution of the form $P(\text{Node}|\text{Parents})$, where the Parents of Node are the nodes with arcs pointing to Node. In other words, if we believe the network is a correct representation of the relationships among the variables, then it should be possible to factor the joint probability distribution into the product of these smaller distributions.

The structure of the network—particularly the arcs that are *absent*—can be viewed as specifying conditional independencies. Two variables A and B are conditionally independent given C if

$$P(A, B|C) = P(A|C) \cdot P(B|C).$$

In the graph, Age affects Insulin only through the Diabetes node, so Age and Insulin are conditionally independent given Diabetes. More generally, given the values of its parents, a node is independent of all other nodes in the graph except its descendents. Formally,

$$P(A, B|Parents(A)) = P(B|Parents(A)) \cdot P(A|Parents(A)),$$

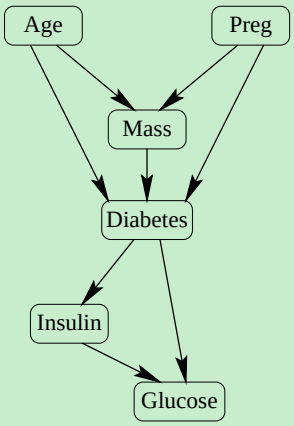

Figure 12: A probabilistic network for diabetes diagnosis

Table 10: Probability tables for the Age, Preg, and Mass nodes from Figure 12. A learning algorithm must fill in the actual probability values based on the observed training data.

Age	$P(A)$
0-25	
26-50	
51-75	
>75	

Preg	$P(N)$
0	
1	
> 1	

		$P(M\|A,N)$		
Age	Preg	0-50	51-100	>100
0-25	0			
0-25	1			
0-25	>1			
26-50	0			
26-50	1			
26-50	>1			
51-75	0			
51-75	1			
51-75	>1			
>75	0			
>75	1			
>75	>1			

unless B is a descendent of A.

In addition to specifying the structure of the network, we need to specify how each probability distribution will be represented. One standard approach is to discretize the variables into a small number of values and represent each probability distribution as a table. For example, suppose we discretized Age into the values {0-25, 26-50, 51-75, >75}, Preg into the values {0, 1, >1}, and Mass into the values {0-50kg, 51-100kg, >100kg}. Then the probability distributions for those three nodes could be represented by the probability tables shown in Table 10. The learning task is to fill in the probability values in these tables. The table for $P(A)$ requires 3 independent parameters (because the four values must sum to 1). The table for $P(N)$ requires 2 parameters, and the table for $P(M|A,N)$ requies 24 parameters (because each row must sum to 1). Similar tables would be required for the other 3 nodes in the network.

Given a set of training examples, this learning problem is very easy to solve. Each probability can be computed directly from the training data. For example the cell $P(N = 1)$ can be computed as the number of patients in the sample that had exactly 1 pregnancies. The parameter $P(M = 51\text{-}100|A = 26\text{-}50, N = 1)$ is the fraction of training examples with Age = 26-50 and Preg = 1 that have a Mass = 51-100kg. Technically, these are the maximum likelihood estimates of each of the probabilities. A difficulty that can arise is that some of the cells in the tables may have very few examples, so the resulting probability estimates are very uncertain. One solution to this is to smooth probabilities for "adjacent" cells in the table. For example, we might require that $P(M = 51\text{-}100|A = 26\text{-}50, N = 1)$ have a value similar to $P(M = 51\text{-}100|A = 26\text{-}50, N = >1)$. Of course we could also take an ensemble approach and generate an ensemble of fitted stochastic models, as described in Section 2 above.

The process of learning a stochastic model consists of three steps: (a) choosing the graphical structure, (b) specifying the form of the probability distribution at each node in the graph, and (c) fitting the parameters of those probability distributions to the training data. In most current applications, steps (a) and (b) are performed by a user and step (c) is performed by a learning algorithm. However, below I briefly discuss methods for automating

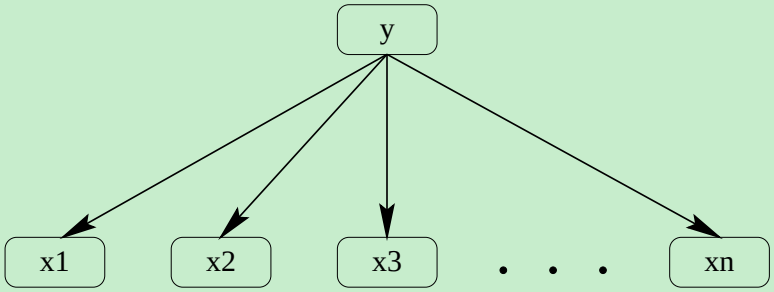

Figure 13: Probabilistic network for the Naive Bayes classifier

step (a)—learning the graphical structure.

Once we have learned the model, how can we apply it to predict whether new patients have diabetes? For a new case, we will observe the values of all of the variables in the model except for the Diabetes node. Our goal is to compute the probability of that node—that is, we seek the distribution $P(D|A, N, M, I, G)$. We could pre-compute this distribution offline before observing any of the data, but the resulting conditional probability table would be immense. A better approach is to wait until the values of the variables have been observed, and then compute the single corresponding row of the class probability table.

This inference problem has been studied intensively, and a very general and elegant algorithm—the junction tree algorithm—has been developed (Jensen, Lauritzen, & Olesen, 1990). In addition, efficient online algorithms have been discovered (e.g., D'Ambrosio, 1993). In the worst case, these algorithms require exponential time, but if the probabilistic network is sparsely connected, the running time is quite reasonable.

5.2 The Naive Bayes Classifier

A very simple approach to stochastic modeling for classification problems is the so-called "naive" Bayes classifier (Duda & Hart, 1973). In this approach, the training examples are assumed to be produced by the probabilistic network shown in Figure 13, where the class variable is y, and the features are $x_1, \ldots, x_n$. According to this model, the environment generates an example by first choosing (stochastically) which class to generate. Then, once the class is chosen, the features describing the example are generated *independently* according to their individual distributions $P(x_j|y)$.

In most applications, the values of the features are discretized so that each feature takes on only a small number of discrete values. The probability distributions are represented as tables as in the diabetes example, and the network can be learned directly from the training data by counting the fraction of examples in each class that take on each feature value.

Because of the simple form of the network, it is easy to derive a classification rule through the application of Bayes' rule. Suppose there are only two classes $\{1, 2\}$. Then our decision rule is to classify a new example into class 1 if $P(y = 1|\mathbf{x}) > P(y = 2|\mathbf{x})$, or equivalently, if $P(y = 1|\mathbf{x})/P(y = 2|\mathbf{x}) > 1$. By Bayes' rule, we can write

$$\begin{aligned} P(y = 1|\mathbf{x}) &= \frac{P(\mathbf{x}|y = 1) \cdot P(y = 1)}{P(\mathbf{x})} \\ P(y = 2|\mathbf{x}) &= \frac{P(\mathbf{x}|y = 2) \cdot P(y = 2)}{P(\mathbf{x})} \end{aligned}$$

Dividing the first equation by the second allows us to cancel the normalizing denominator $P(\mathbf{x})$ and obtain

$$\frac{P(y = 1|\mathbf{x})}{P(y = 2|\mathbf{x})} = \frac{P(\mathbf{x}|y = 1) \cdot P(y = 1)}{P(\mathbf{x}|y = 2) \cdot P(y = 2)}$$

The quantity $P(\mathbf{x}|y = i)$ is just the product of the individual probabilities $P(x_j|y = 1)$, so we have

$$\frac{P(y = 1|\mathbf{x})}{P(y = 2|\mathbf{x})} = \frac{P(x_1|y = 1) \cdot P(x_2|y = 1) \cdots P(x_n|y = 1) \cdot P(y = 1)}{P(x_1|y = 2) \cdot P(x_2|y = 2) \cdots P(x_n|y = 2) \cdot P(y = 2)}.$$

This gives the decision rule that we should classify an example into class 1 if and only if

$$\frac{P(x_1|y = 1)}{P(x_1|y = 2)} \cdot \frac{P(x_2|y = 1)}{P(x_2|y = 2)} \cdots \frac{P(x_n|y = 1)}{P(x_n|y = 2)} \cdot \frac{P(y = 1)}{P(y = 2)} > 1.$$

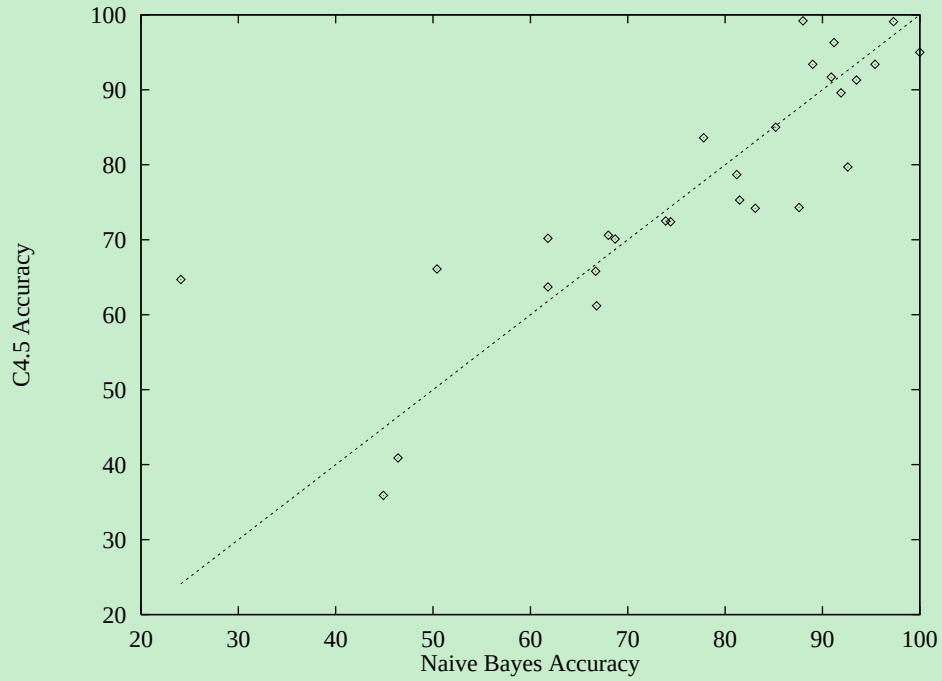

Figure 14: Comparison of C4.5 and the naive Bayesian classifier on 28 data sets.

Despite the fact that the naive Bayes model barely deserves the name "model" in many applications, it performs surprisingly well. Figure 14 compares the performance of C4.5 to the naive Bayes classifier on 28 benchmark tasks (Domingos & Pazzani, 1996). The results show that except for a few domains where naive Bayes performs very badly, it is typically competitive with or superior to C4.5. Domingos and Pazzani showed that the algorithm is very robust to violations of the assumption that the features are generated independently.

5.3 Naive unsupervised learning

An important application of stochastic models is to problems of unsupervised learning. In unsupervised learning, we are given a collection of examples $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, and our goal is to construct some model of how these examples are generated. For example, we might believe that these examples belong to some collection of classes, and we want to determine the properties of those classes. This is sometimes called *clustering* the data into classes, and many algorithms have been developed that apply a measure of the distance between two examples to group together nearby examples.

A stochastic modeling approach for unsupervised learning works essentially the same way as for supervised learning. We begin by defining the structure of the model as a probabilistic network. One commonly-used model is the same "naive" network used by the naive Bayes classifier shown in Figure 13.

Although we can use the same network structure, the problem of learning the parameters of the network is much more difficult, because our data do not contain the values of the class variable y. This is a simple case of the problem of fitting stochastic models that contain *hidden variables*, which are variables whose values are not observed in the training data.

Many different algorithms have been developed for fitting networks containing hidden variables. As with the naive Bayes classification algorithm, the basic goal (at least for this article) is to compute the maximum likelihood estimates of the parameters of the network, which I will refer to as the vector of weights W. In other words, we want to find the value of W that maximizes $P(S|W)$, where S is the observed training sample. This is typically formulated as the equivalent problem of maximizing the log likelihood: $\log P(S|W)$. Under the assumption that each training example in S is generated independently, this is equivalent to maximizing $\sum_i \log P(\mathbf{x}_i|W)$, where $\mathbf{x}_i$ is the i-th training example.

I briefly sketch three algorithms: gradient descent, the expectation-maximization algorithm, and Gibbs sampling.

5.3.1 Gradient Descent for Bayes Networks

Russell, Binder, Koller, and Kanazawa (1995) describe a method for computing the gradient of the log likelihood: $\nabla_W \log P(\mathbf{x}_i|W)$. Let us focus on a particular node in the network representing variable V. Let U be the parents of V, and let $w = P(V = v|U = u)$ be the entry in the conditional probability table for V when $V = v$ and $U = u$. Then Russell et al. show that

$$\frac{\partial \log P(\mathbf{x}_i|W)}{\partial w} = \frac{P(V = v, U = u|W, \mathbf{x}_i)}{w}.$$

The conditional probability in the numerator can be computed by any algorithm for inference in probabilistic networks (including the junction tree algorithm mentioned above).

Using this formula, the gradient of the parameters of a network can be computed with respect to a training sample S. We can then apply standard gradient descent algorithms, such as fixed step size methods or the conjugate gradient algorithm, to search for the maximum likelihood set of weights. One subtlety is that we must ensure that the weights lie between 0 and 1 and sum to 1 appropriately. It suffices to constrain and renormalize them after each step of gradient descent. Russell et al have tested this algorithm on a wide variety of probabilistic network structures.

5.3.2 The Expectation Maximization Algorithm

The second algorithm I will discuss is the Expectation Maximization (EM) algorithm (Dempster, Laird, & Rubin, 1976). EM can be applied to probabilistic networks if the node probability distributions belong to the exponential family of distributions (which includes the binomial, multinomial, exponential, poisson, and normal distributions, and many others). In particular, conditional probability tables have multinomial distributions, so the EM algorithm can be applied to the case I am considering in this article.

The EM algorithm is an iterative algorithm that starts with an initial value for W and incrementally modifies W to increase the likelihood of the observed data. One way to understand EM is to imagine that each training example is augmented to include parameters describing values of the hidden variables. For concreteness, consider the simple case of Figure 13 with the class variable y hidden. Assume y takes on two values 1 and 2. Then we would augment each training example with $P(y = 1|\mathbf{x})$ and $P(y = 2|\mathbf{x})$. (Actually, the $P(y = 2|\mathbf{x})$ is redundant in this case, because $P(y = 2|\mathbf{x}) = 1 - P(y = 1|\mathbf{x})$). More generally, each observed example $\mathbf{x}_i$ will be augmented with the expected values of the *sufficient statistics* for describing the probability distribution of the hidden variables.

The EM algorithm alternates between two steps until W converges:

- **E-step:** Given the current value of W, compute the augmentations $P(y_i = 1|\mathbf{x}_i)$ and $P(y_i = 2|\mathbf{x}_i)$ for each example $\mathbf{x}_i$.
- **M-step:** Given the augmented data set, compute the maximum likelihood estimates for W under the assumption that the probability distribution of the values of the hidden variables is correctly specified in each augmented training example.

In the case of Figure 13, the E-step applies Bayes' theorem:

$$P(y_i = 1|\mathbf{x}_i) = \frac{P(\mathbf{x}_i|y_i = 1) \cdot P(y_i = 1)}{P(\mathbf{x}_i)}$$

All of the quantities on the right-hand side can be computed given the current value for W. The quantity $P(\mathbf{x}_i|y_i = 1)$ is the product of $P(x_{ij}|y_i = 1)$ for each feature j, and $P(y_i = 1)$ is the current estimated probability of generating an example in class 1, $P(y = 1)$.

The M-step for naive Bayes classification must estimate each of the weights from the augmented training examples. To estimate $P(y = 1)$, we sum the augmented value $P(y_i = 1)$ over all i and divide by the sample size m:

$$P(y = 1) = \frac{1}{m} \sum_i P(y_i = 1).$$

To estimate the conditional probability that feature x_j is 1 for examples in class 1, we take each training example i that has $x_{ij} = 1$, sum up the augmented values $P(y_i = 1)$, and divide by the total $P(y = 1)$:

$$P(x_j = 1|y = 1) = \frac{1}{P(y = 1)} \sum_{\{i|x_{ij}=1\}} P(y_i = 1).$$

n effect, we treat each augmented training example as if it were a member of class 1 with probability $P(y_i = 1)$ and as if it were a member of class 2 with probability $1 - P(y_i = 1)$.

A well-known application of the EM algorithm in unsupervised clustering is the Autoclass program (Cheeseman, Self, Kelly, Taylor, Freeman, & Stutz, 1988). In addition to discrete variables (of the kind I have been discussing), Autoclass can handle continuous variables. Instead of computing the maximum likelihood estimates of the parameters, it adopts a prior probability distribution over the parameter values and computes the maximum a posteriori probability (MAP) values. This is easily accomplished by a minor modification of EM. One of the most interesting applications of Autoclass was to the problem of analyzing the infrared spectra of stars. Autoclass discovered a new class of star, and this discovery was subsequently accepted by astronomers (Cheeseman et al., 1988).

5.3.3 Gibbs Sampling

The final algorithm that I will discuss is a Monte Carlo technique called Gibbs sampling (Geman & Geman, 1984). Gibbs sampling is a method for generating random samples from a joint probability distribution $P(A_1, \ldots, A_n)$ when sampling directly from the joint distribution is difficult. Suppose we know the conditional distribution of each variable A_i in terms of all of the others: $P(A_1|A_2, \ldots, A_n)$, $P(A_2|A_1, A_3, \ldots, A_n)$, $\ldots$, $P(A_n|A_1, \ldots, A_{n-1})$. The Gibbs sampler works as follows. We start with a set of arbitrary values, $a_1, \ldots, a_n$, for the random variables. For each value of i, we then sample a new value a_i for random variable A_i according to the distribution $P(A_i|A_1 = a_1, \ldots, A_{i-1} = a_{i-1}, A_{i+1} = a_{i+1}, \ldots, A_n = a_n)$. If we repeat this long enough, then under certain mild conditions the empirical distribution of these generated points will converge to the joint distribution.

How is this useful for learning in probabilistic networks? Suppose we wish to take a full Bayesian approach to learning the unknown parameters in the network. In such cases, we want to compute the posterior probability of the unknown parameters given the data. Let W denote the vector of unknown parameters. In a full Bayesian approach, we treat W as a random variable with a prior probability distribution $P(W)$. Given the training examples $\{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$, we want to compute $P(W|\mathbf{x}_1, \ldots, \mathbf{x}_m)$. This is very difficult, because of the hidden classes $y_1, \ldots, y_m$. However, using the Gibbs sampler, we can generate samples from the distribution

$$P(W, y_1, \ldots, y_m|\mathbf{x}_1, \ldots, \mathbf{x}_m).$$

Then, by simply ignoring the y values, we obtain samples for W: $W_1, \ldots, W_L$.

These W values constitute an ensemble of learned values for the network parameters. To classify a new data point $\mathbf{x}$, we can apply each of the values of W_ℓ to predict the class of $\mathbf{x}$ and have them vote. The voting is with equal weight. If some W values have higher posterior probability than others, then they will appear more often in our sample.

To apply the Gibbs sampler to our unsupervised learning problem, we begin by choosing random initial values for the parameters W of the network and setting ℓ to zero. We then repeat the following loop L times.

1. **Compute new values for $y_1, \ldots, y_n$:** From the probabilistic network in Figure 13, we can compute $P(y_i|W, \mathbf{x}_i)$, because we know (current guesses for) W and (observed values for) $\mathbf{x}_i$. To sample from this distribution, we flip a biassed coin with probability of heads $P(y_i|W, \mathbf{x}_i)$.

2. **Compute new values for W:** Let w_0 be the parameter that represents the probability of generating an example from class 0. Suppose the prior distribution, $P(w_0)$, is the uniform distribution for all values $0 \leq w_0 \leq 1$. Let m_0 be the number of training examples (currently) assigned to class 0. Then the posterior probability $P(w_0|y_1, \ldots, y_m)$ has a special form known as a Beta distribution with parameters $m_0 + 1$ and $m - m_0 + 1$. Algorithms are available for drawing samples from this distribution.

   Similarly, let w_{jv0} be the parameter that represents the probability that the j-th feature will have the value v when drawn from class zero: $P(x_j = v|y_j = 0)$. Again assuming uniform priors for $P(w_{jv0})$, this variable also has a Beta distribution with parameters $c_{jv0} + 1$ and $m_0 - c_{jv0} + 1$, where c_{jv0} is the number of training examples in class 0 having $x_j = v$. As with w_0, we can sample from this distribution.

   We can do the same thing for the parameters concerning class 1: w_{jv1}.

3. **Record the value of the parameter vector.** Let $\ell := \ell + 1$ and set $W_\ell := W$.

To allow the Gibbs sampler to converge to a stationary distribution, we should perform some number of iterations in which we skip Step 3, before recording L values of the parameter vector. This procedure gives us an ensemble of L probabilistic networks which can be applied to classify new data points.

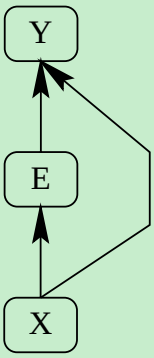

Figure 15: Probabilistic model describing a mixture of experts

There is a close resemblance between Gibbs sampling and the EM algorithm. In both algorithms, the training examples are augmented with information about the hidden variables. In EM, this information describes the probability distribution of the hidden variables, whereas in Gibbs sampling, this information consists of random values drawn according to that probability distribution. In EM, the parameters are recomputed to be their maximum likelihood estimates based on the current values of the hidden variables, whereas in Gibbs sampling, new parameter values are sampled from the posterior distribution given the current values of the hidden variables. Hence, EM can be viewed as a maximum likelihood (or MAP) approximation to Gibbs sampling.

One potential problem that can arise in Gibbs sampling is caused by symmetries in the stochastic model. In the case we are considering, for example, suppose there are two true underlying hidden classes, class A and class B. We want the model to generate examples from class A when $y = 1$ and from class B when $y = 2$. However, there is nothing to force the model to do this. It could just as easily use $y = 1$ to represent examples of class B and $y = 2$ to represent examples of class A. If permitted to run long enough, in fact, the Gibbs sampler should explore both possibilities. If our sample of weight vectors W_ℓ includes weight vectors corresponding to both of these alternatives, then when we combine these weight vectors, we will get bad results. In practice, this problem often does not arise, but in general, steps must be taken to remove symmetries from the model (see Neal, 1993).

Gibbs sampling is a very general method; it can often be applied in situations where the EM algorithm cannot. The generality of Gibbs sampling has made it possible to construct a general-purpose programming environment, called BUGS, for learning stochastic models (Gilks, Thomas, & Spiegelhalter, 1993). In this environment, the user specifies the graph structure of the stochastic model, the form of the probability distribution at each node, and prior distributions for each parameter. The system then develops a Gibbs sampling algorithm for fitting this model to the training data. BUGS can be downloaded from `http://www.mrc-bsu.cam.ac.uk/bugs/`.

5.4 More Sophisticated Stochastic Models

This section presents three examples of more sophisticated stochastic models that have been developed: the Hierarchical Mixture of Experts (HME) model, the Hidden Markov Model (HMM), the Dynamic Probabilistic Network (DPN). Like the naive model from Figure 13, these models can be applied to a wide number of problems without performing the kind of detailed modeling of causal connections that we performed in the diabetes example from Figure 12.

5.4.1 The Hierarchical Mixture of Experts

The Hierarchical Mixture of Experts (HME) model (Jordan & Jacobs, 1994) is intended for supervised learning in situations where one believes the training data are being generated by a mixture of separate "experts." For example, in a speech recognition system, we might face the task of distinguishing the spoken words "Bee", "Tree", "Gate", and "Mate". This naturally decomposes into two hard sub-problems: (a) distinguishing "Bee" from "Tree" and (b) distinguishing "Gate" from "Mate".

Figure 15 shows the probabilistic network for a simple mixture-of-experts model for this case. According to this model, a training example $(\mathbf{x}_i, y_i)$ is generated by first generating the data points $\mathbf{x}_i$, then choosing an "expert" e_i stochastically (depending on the value of $\mathbf{x}_i$), and then choosing the class y_i depending on the values of $\mathbf{x}_i$ and e_i.

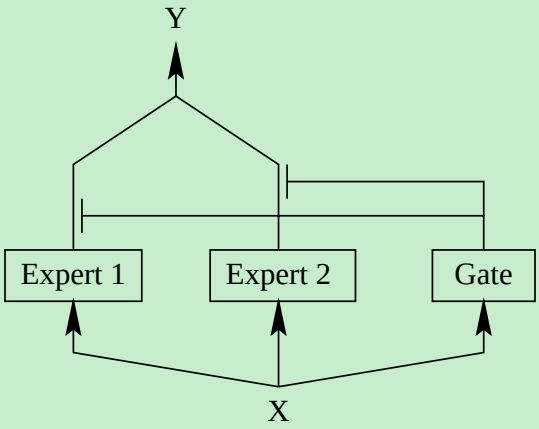

Figure 16: The mixture of experts model viewed as a specialized neural network

There are two things to note about this model. First, the direction of causality is reversed from the naive Bayes network of Figure 13. Second, if we assume that all of the features of each $\mathbf{x}_i$ will always be observed, we do not need to model the probability distribution $P(X)$ on the bottom node in the graph. Third, unless we make some strong assumptions, the probability distribution $P(Y|X, E)$ is going to be extremely complex, because it must specify the probability of the classes as a function of every possible combination of features for X and expert E.

In their development of this general model, Jordan and Jacobs assume that each probability distribution has simple form (see Figure 16, which shows the model as a kind of neural network classifier). Each value of the random variable E specifies a different "expert". The input features x are fed into each of the experts and into a "gating network". The output of each expert is a probability distribution over the possible classes. The output of the gate is a probability distribution over the experts. This overall model has the following analytical form:

$$P(y|\mathbf{x}) = \sum_e g_e(\mathbf{x}) p_e(y|\mathbf{x}),$$

where the index e varies over the different experts. The value $g_e(\mathbf{x})$ is the output of the gating network for expert e. The value $p_e(y|\mathbf{x})$ is the probability distribution over the various classes output by expert e.

Jordan and Jacobs have investigated networks where the individual experts and the gating network have very simple forms. In a 2-class problem, each expert has the form

$$p_e(y|\mathbf{x}) = \sigma(w_e^T \mathbf{x}),$$

where w_e^T is the transpose of a vector of parameters, $\mathbf{x}$ is the vector of input feature values, and σ is the usual logistic sigmoid function $1/(1 + exp(\cdot))$.

The gating network was described earlier in Section 2. The gating values are computed according to

$$\begin{aligned} z_e &= v_e^T \mathbf{x} \\ g_e &= \frac{e^{z_e}}{\sum_u e^{z_u}} \end{aligned}$$

In other words, z_e is the dot product of a weight vector v_e and the input features $\mathbf{x}$. The output g_e is the soft-max of the z_e values. The g_e values are all positive and sum to 1. This is known as the multinomial logit model in statistics.

The problem of learning the parameters for a mixture-of-experts model is similar to the problem of unsupervised learning, except that here, the hidden variable is not the class y_i of each training examples i, but rather the expert e_i that was responsible for generating that training example. If we knew which expert generated each training example, then we could fit the parameters directly from the training data as with did for the naive Bayes algorithm. Jordan and Jacobs have applied both gradient descent and the EM algorithm to solve this learning problem. For the particular choice of sigmoid and soft-max functions for the experts and gates, the EM algorithm has a particularly efficient implementation as a sequence of weighted least squares problems.

The simple one-level hierarchy of experts shown in Figure 15 can be extended to deeper hierarchies. Figure 17 shows a 3-level hierarchy. All of the fitting algorithms apply to this more general case as well.

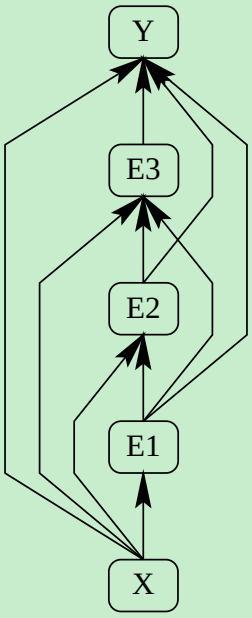

Figure 17: The hierarchical mixture of experts model

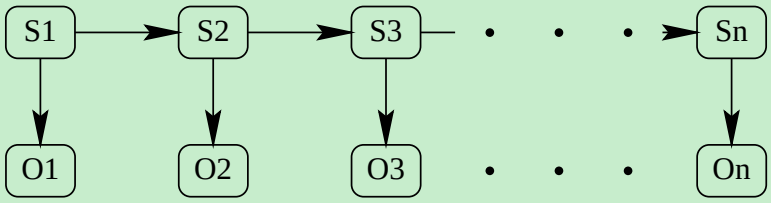

Figure 18: The probabilistic network for a Hidden Markov Model

5.4.2 The Hidden Markov Model

Figure 18 shows the probabilistic network of a hidden Markov model (HMM; Rabiner, 1989). An HMM generates examples that are strings of length n over some alphabet A of letters. Hence, each example $\mathbf{x}$ is a string of letters: $o_1 o_2 \cdots o_n$ (where the o stands for "observable"). To generate a string, the HMM begins by generating an initial state s_1 according to probability distribution $P(s_1)$. From this state, it then generates the first letter of the string according to the distribution $P(o_1|s_1)$. It then generates the next state s_2 according to $P(s_2|s_1)$. Then it generates the second letter according to $P(o_2|s_2)$ and so on.

In some applications, the transition probability distribution is the same for all pairs of adjacent state variables: $P(s_t|s_{t-1})$. Similarly, the output (or emission) probability distribution can be the same for all states: $P(o|s)$. In this variation, the HMM is equivalent to a stochastic finite-state automaton (FSA). At each time step, the FSA makes a probabilistic state transition and then generates an output letter.

Another common variation on HMM's is to include an *absorbing state* or *halting state* as one of the values of the state variable. If the HMM makes a transition into this state, it terminates the string being generated. This permits HMM's to model strings of variable length.

Hidden Markov models have been widely applied in speech recognition, where the alphabet of letters consists of "frames" of the speech signal (Rabiner, 1989). Each word in the language can be modeled as an HMM. Given a new spoken word, a speech recognition system computes the likelihood that each of the word HMM's generated that spoken word. The recognizer then predicts the most likely word. A similar analysis can be applied at the level of whole sentences by concatenating word-level HMM's. Then the goal is to find the sentence most likely to have generated to speech signal.

To learn an HMM, a set of training examples is provided, where each example is a string. The sequence of states

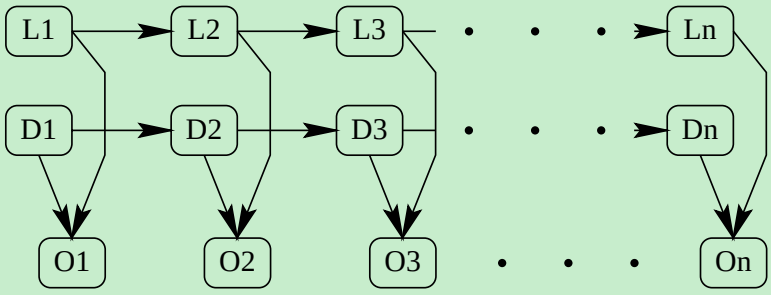

Figure 19: A simple dynamic probabilistic network with two independent state variables, robot location (L_t) and camera direction (S_t).

that generated the string is hidden. Once again, however, the EM algorithm can be applied. In the context of HMM's, it is known as the Baum-Welch or Forward-Backward algorithm. In the E-step, the algorithm augments each training example with statistics describing the hypothesized string of states that generated the example. In the M-step, the parameters of the probability distributions are re-estimated from the augmented training data.

Stolcke and Omohundro (1994) have developed algorithms for learning the structure and parameters of hidden Markov models from training examples. They applied their techniques to several problems in speech recognition and incorporated their algorithm into a speaker-independent speech recognition system.

5.4.3 The Dynamic Probabilistic Network

In the hidden Markov model, the number of values for the state variable at each point in time can become very large. Consequently, the number of parameters in the state transition probability distribution $P(s_t|s_{t-1})$ can become intractably large. One solution is to represent the causal structure within each state by a stochastic model. For example, consider a mobile robot with a steerable television camera. The images observed by the camera will be the output alphabet of the HMM. The hidden state will consist of the location of the robot within a room and the direction the camera is pointing. Suppose there are 100 possible locations and 15 possible camera directions. In an HMM, the hidden state will be a single variable with 1500 possible values.

However, suppose that the robot has separate commands to change its location and to steer its camera. At each time step, it chooses to perform exactly one of these two actions. Then it makes sense to represent the hidden state by two separate state variables: robot location and camera direction. Figure 19 shows the resulting stochastic model, which is variously called a dynamic probabilistic network (DPN, Kanazawa, Koller, & Russell, 1995), a dynamic belief network (DBN, Dean & Kanazawa, 1989), and a factorial HMM (Ghahramani & Jordan, 1996).

Unfortunately, inference and learning with DPN's is computationally challenging. The overall approach of applying the EM algorithm or Gibbs sampling is still sound. However, the E-step of computing the augmented training examples is itself difficult. Ghahramani and Jordan (1996) and Kanazawa, Koller, and Russell (1995) describe algorithms that can perform approximate E-steps. A recent review of this very active research area can be found in Smyth, Heckerman and Jordan (1997).

5.5 Application-Specific Stochastic Models

A major motivation for the stochastic modeling approach to machine learning is to communicate background knowledge to the learning algorithm. While the general models discussed above achieve this goal to some extent, it is possible to go much further in this direction, and many applications of machine learning are pursuing this approach. To give a flavor of the kinds of models being developed, I describe one example out of the many recently published papers.

Revow, Williams, and Hinton (1996) have developed a stochastic model for hardwritten digit recognition, shown in Figure 20. To generate a digit according to this model, we first randomly choose one of the 10 digits. This determines the "home locations" of 8 points that control a uniform B spline (denoted $h_1, \ldots, h_8$ in the figure). The B spline specifies the shape of the digit. The next step randomly perturbs those control points (using a Gaussian distribution) to produce the control points that will be used to generate the handwritten digit (denoted $k_1, \ldots, k_8$ in the figure).

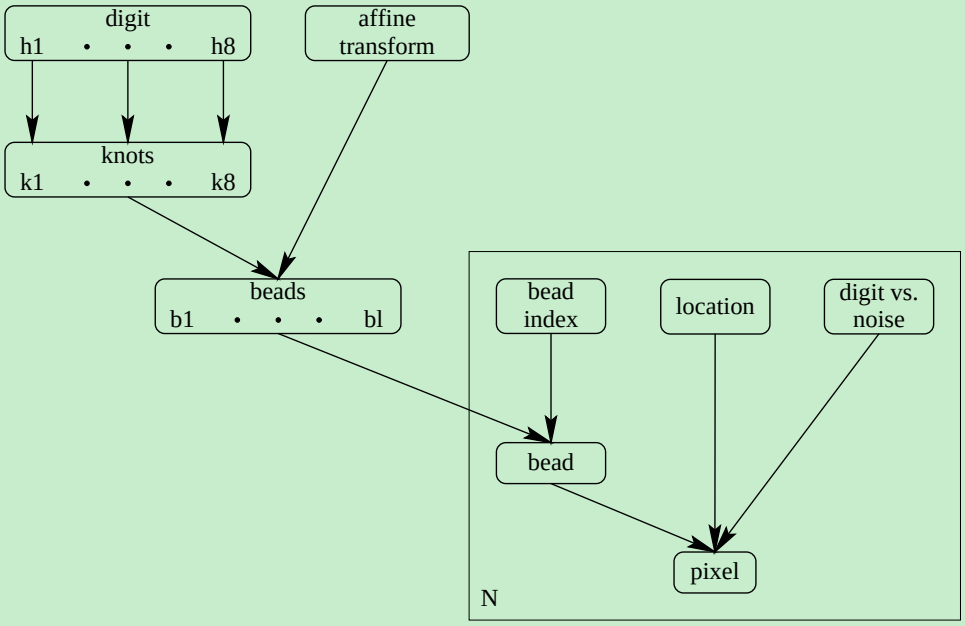

Figure 20: Probabilistic network for generating a digit and choosing N pixels to ink.

Next we generate a 6-degree-of-freedom affine transformation that models translation, rotation, skewing, and so forth. From the randomly chosen affine transformation and the control points, we deterministically lay out a sequence of beads along the spline curve. These beads will act as circular Gaussian ink generators that will ink the pixels in the image. There is a parameter σ that specifies the standard deviation of the Gaussian ink generators. The beads are placed a distance 2σ apart along the spline; hence, the number of beads varies as a function of σ. At this point, we have made all of the choices that are independent of the image pixels.

Now, the model generates N inked pixels (indicated in the diagram by the box with an N in the lower left corner). To generate each pixel, it first generates a boolean variable that indicates whether the pixel will be a noise pixel or a pixel from the digit. If it is a noise pixel, the location is chosen uniformly at random and inked. If it is a digit pixel, one of the l beads is chosen at random (specified by the variable bead index), and a pixel is inked according to a symmetric Gaussian probability distribution.

The parameters of the model include the probability of noise pixels, the standard deviation σ of the Gaussian ink generators, the variance of the Gaussian used to perturb the control points, the parameters controlling the affine transformation, and the home locations of the eight control points.

There are two tasks that we wish to perform with this model: classification and learning. Let us consider classification first. Given an image and 10 learned spline models, our goal is to find the spline model that is most likely to have generated the image. We do this by computing the probability that the spline model for each digit generated the image. This computation is difficult, because we are given only the locations of the inked pixels and the (hypothesized) class of the digit. In principle, we should consider all possible locations for the control points, all possible values for σ, and all possible choices for whether each inked pixel is a noise pixel or a digit pixel. Each such combination could have generated the observed image (with a certain probability). We should integrate over these parameters to compute the probability that a given digit model generated the given image.

In practice, Revow et al. take a maximimum likelihood approach. They compute the values for the control points $(k_1, \ldots, k_8)$, σ, and the noise/digit choices that maximize the likelihood of generating the observed image. The EM algorithm is applied to compute this. Figure 21 shows the fitting of the model for 3 and for 5 to a typical image. Notice that during the fitting, σ is initialized to a very large value, so that the ink generators are likely to capture inked pixels. When EM begins to converge, σ is decreased (according to a given schedule), and new beads are positioned along the spline using the current value of σ. Then, fitting is resumed. This is performed approximately six times per image.

Now let's consider learning. The goal of learning is to learn for each digit the home locations of the control points and the variance for perturbing these control points. In addition, the parameters controlling the affine transformation must be learned, but these are assumed to be the same for all classes. To solve this learning problem, Revow et

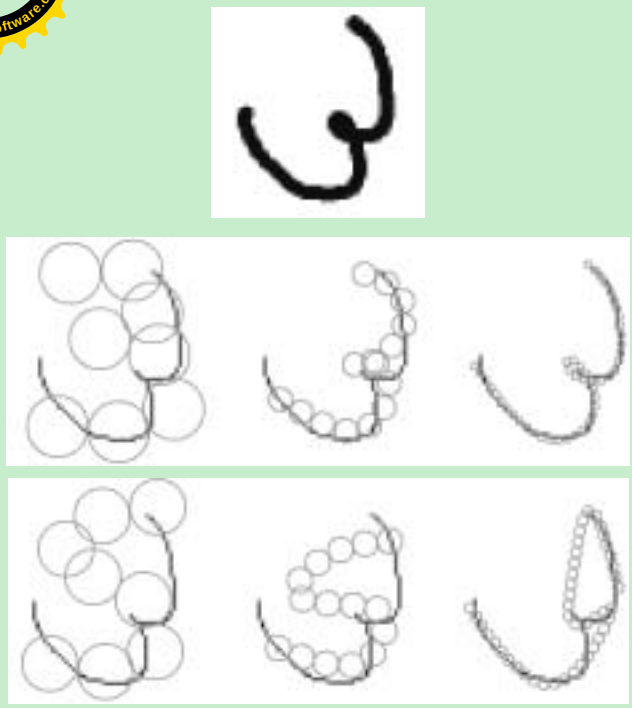

Figure 21: Some stages of fitting models to an image of a 3 (from Revow, et al., 1996, reproduced with permission). The image is displayed in the top row. The next row shows the model for a 3 being fitted. The bottom row attempts to fit the model of a 5. The light circles indicate the value of σ. In the bottom two rows, the image has been artificially thinned so that the circles are visible.

al. again apply EM. In the E-step, the training examples are augmented with the maximimum likelihood locations of the control points and the value of σ, which are computed via a nested EM as described above. In the M-step, the average home location of each control point and the average σ is computed. The model for each digit is trained by fitting only to training examples of that digit. The running time is dominated by the cost of the inner (classification) EM algorithm.

Revow et al report results that are competitive with the best known methods on the task of recognizing digits from US mail zip codes. To achieve this performance, they employ some post-processing steps that analyze how well each digit model fits the image. They also considered learning mixtures of digit models for cases where there are significantly different ways of writing a digit (e.g., the digit 7 with and without a central horizontal bar).

The advantage of the stochastic modeling approach is that learning is very fast both computationally, because EM is very quick, and statistically, because there are only 22 parameters in each digit model (2 coordinates for each control point and 6 affine transformation parameters). Another advantage is that the digit images do not need to be preprocessed (e.g., to remove slants, scale to a standard size, and so forth). This preprocessing can be a significant source of errors as well as requiring extra implementation effort. A related advantage is that the stochastic models can fit a single digit in the context of several other digits—so that precise segmentation is not required prior to classification. The primary disadvantage is that classification is slower, because of the need to perform a search to fit each digit model.

5.6 Learning the Structure of Stochastic Models

All of the methods I have discussed so far learn the parameters of a stochastic model whose structure is given. An important research question is whether this *structure* can be learned from data as well. Recent research has made major progress in developing algorithms for this problem.

One of the first algorithms was developed by Chow and Liu (1968) for learning a network structure in the form of a directed tree. This algorithm first constructs a complete undirected graph where the nodes are the variables and the edges are labeled with the mutual information between the variables. The algorithm then finds the maximum weighted spanning tree of this graph, chooses a root node arbitrarily, and orders the arcs to point away from the root. A nice feature of this algorithm is that it is quite fast—it runs in polynomial time.

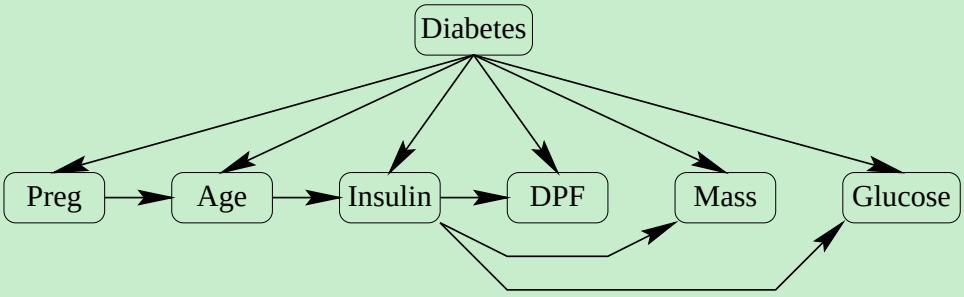

Figure 22: Probabilistic network constructed by the TAN algorithm for a diabetes diagnosis task

Cooper and Herskovits (1992) developed an algorithm, called K2, for learning the structure of a stochastic model where all variables are observed in the training data. They adopt a maximum a posteriori (MAP) approach within a Bayesian framework. The key contribution of their paper was to derive a formula for the posterior probability of network. This formula can be updated incrementally as changes are made to the network.

Using their formula, they implemented a simple greedy search algorithm for finding (approximately) the MAP network structure. Their algorithm requires the user to provide an ordering of the variables, and it will consider adding arcs only from variables earlier in the ordering to variables later in the ordering. This significantly constrains the search and also ensures that the learned network will remain acyclic. K2 begins with a network having no arcs—all variables are independent of one another. K2 evaluates the posterior probability of adding each possible single edge and makes the highest-ranking addition. This greedy algorithm is continued until no single change improves the posterior probability.

Heckerman, Geiger, and Chickering (1995) describe a modification to the K2 method for computing posterior probabilities and a local search algorithm that uses this improvement. Their method requires a prior probability distribution in the form of a prior network and two parameters: (a) an equivalent sample size (which controls how much new data is required to override the prior) and (b) a penalty for each arc that is different from the prior network. Their local search algorithm considers all one-step changes to the network (arc addition, deletion, and reversal), and retains the change that most increases the posterior probability of the network. They obtain results comparable to K2. Interesting, they found that the most important role for the prior network was to provide a good starting point for local search (rather than to bias the objective function for guiding the search).

Friedman and Goldszmidt (1996) developed a network learning algorithm, called Tree Augmented Naive Bayes (TAN), specifically for supervised learning. The TAM algorithm starts with a naive Bayes network of the kind shown in Figure 13 and considers adding arcs to improve the posterior probability of the network. They apply a modification of the Chow and Liu algorithm to learn a tree structure of arcs connecting the x_j variables to one another. Figure 22 shows a network learned by TAN for a diabetes diagnosis problem. Compared to Figure 12, the directions of the arcs are wrong. Nonetheless, the network gives quite accurate classifications. Figure 23 compares the performance of TAN to C4.5 on 22 benchmark problems. The plot shows that TAN outperforms C4.5 on most of the domains.

There are many other important papers on the topic of structure learning for probabilistic networks. Four important references are Verma and Pearl (1990), Spirtes and Meek (1995), Spiegelhalter, Dawid, Lauritzen, and Cowell (1993), Spirtes, Glymour and Scheines (1993) and the references therein.

5.7 Summary: Stochastic Models

This completes my review of methods for learning with stochastic models. There are several good survey articles of this topic (Buntine, 1994, 1996; Heckerman, 1996).

The area of stochastic modeling is very active right now. Journals and conferences that were once devoted exclusively to neural network applications are now presenting many papers on stochastic modeling. The research community is still gathering experience and developing improved algorithms for fitting and reasoning with stochastic models. Many of the stochastic models we would like to work with are intractable. The challenge is to find general-purpose, tractable approximation algorithms for reasoning with these very elegant and expressive stochastic models.

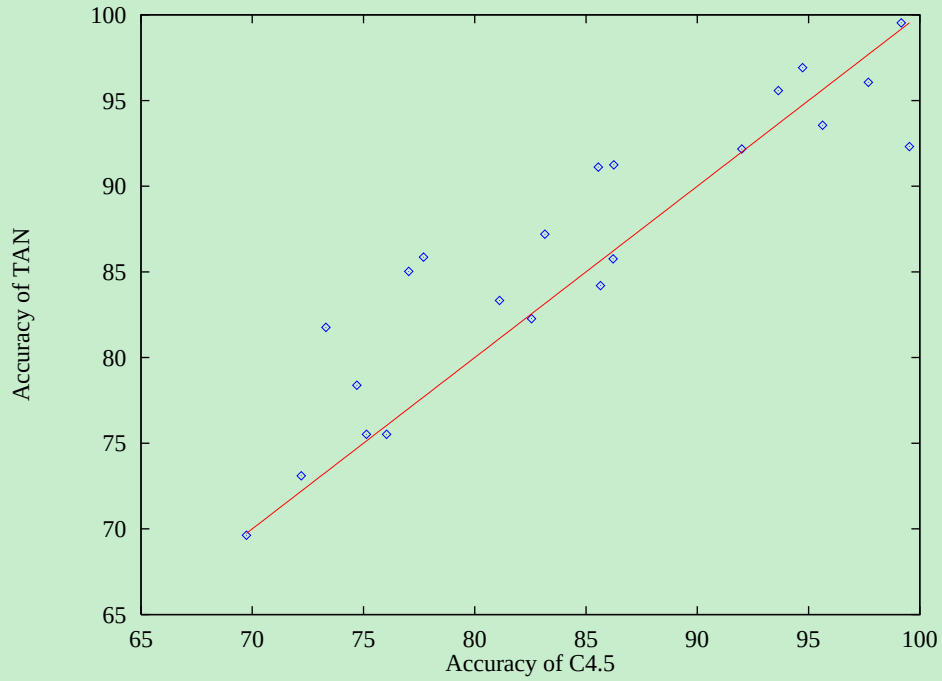

Figure 23: Comparison of TAN and C4.5 on 22 benchmark tasks. Points above the diagonal line correspond to cases where TAN gave more accurate results than C4.5.

6 Concluding Remarks

Any survey must choose particular areas and omit others. Let me briefly mention some other active areas. A central topic in machine learning is the control of overfitting. There have been many developments in this area as researchers explored various penalty functions and resampling techniques (including cross-validation) for preventing overfitting. An understanding of the overfitting process has been obtained through the statistical concepts of bias and variance, and several authors have developed bias/variance decompositions for classification problems.

Another active topic has been the study of algorithms for learning relations expressed as Horn clause programs. This area is also known as Inductive Logic Programming, and many algorithms and theoretical results have been developed in this area.

Finally, many papers have addressed practical problems that arise in applications such as visualization of learned knowledge, methods for extracting understandable rules from neural networks, algorithms for identifying noise and outliers in data, and algorithms for learning easy-to-understand classifiers.

There have been many exciting developments in the past five years, and the relevant literature in machine learning has been growing rapidly. As more areas within artificial intelligence and computer science apply machine learning methods to attack their problems, I expect that the flow of interesting problems and practical solutions will continue. It is a very exciting time to be working in machine learning.

References

Abu-Mostafa, Y. (1990). Learning from hints in neural networks. *Journal of Complexity*, *6*, 192–198.

Ali, K. M., & Pazzani, M. J. (1996). Error reduction through learning multiple descriptions. *Machine Learning*, *24* (3), 173–202.

Barto, A. G., Bradtke, S. J., & Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, *72*, 81–138.

Barto, A. G., & Sutton, R. (1997). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

ertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA.

Blum, A., & Rivest, R. L. (1988). Training a 3-node neural network is NP-Complete (Extended abstract). In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pp. 9–18 San Francisco, CA. Morgan Kaufmann.

Blum, A. (1997). Empirical support for Winnow and Weighted-Majority algorithms: Results on a calendar scheduling domain. *Machine Learning*, *26* (1), 5–24.

Breiman, L. (1996a). Bagging predictors. *Machine Learning*, *24* (2), 123–140.

Breiman, L. (1996b). Stacked regressions. *Machine Learning*, *24*, 49–64.

Buntine, W. L. (1990). *A theory of learning classification rules*. Ph.D. thesis, University of Technology, School of Computing Science, Sydney, Australia.

Buntine, W. (1994). Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, *2*, 159–225.

Buntine, W. (1996). A guide to the literature on learning probabilistic networks from data. *IEEE Transactions on Knowledge and Data Engineering*, *8*, 195–210.

Caruana, R. (1996). Algorithms and applications for multitask learning. In Saitta, L. (Ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 87–95 San Francisco, CA. Morgan Kaufmann.

Cassandra, A. R., Kaelbling, L. P., & Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1023–1028 Cambridge, MA. AAAI Press/MIT Press.

Castillo, E., Gutierrez, J. M., & Hadi, A. (1997). *Expert Systems and Probabilistic Network Models*. Springer Verlag, New York.

Catlett, J. (1991). On changing continuous attributes into ordered discrete attributes. In Kodratoff, Y. (Ed.), *Proceedings of the European Working Session on Learning*, pp. 164–178 Berlin. Springer-Verlag.

Chan, P. K., & Stolfo, S. J. (1995). Learning arbiter and combiner trees from partitioned data for scaling machine learning. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pp. 39–44 Menlo Park, CA. AAAI Press.

Cheeseman, P., Self, M., Kelly, J., Taylor, W., Freeman, D., & Stutz, J. (1988). Bayesian classification. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 607–611 Cambridge, MA. AAAI Press/MIT Press.

Cherkauer, K. J. (1996). Human expert-level performance on a scientific image analysis task by a system using combined artificial neural networks. In Chan, P. (Ed.), *Working Notes of the AAAI Workshop on Integrating Multiple Learned Models*, pp. 15–21. Available from `http://www.cs.fit.edu/~imlm/`.

Chipman, H., George, E., & McCulloch, R. (1996). Bayesian CART. Tech. rep., Department of Statistics, University of Chicago. Available as `http://gsbrem.uchicago.edu/Papers/cart.ps`.

Chow, C., & Liu, C. (1968). Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, *14*, 462–467.

Clemen, R. T. (1989). Combining forcasts: A review and annotated bibliography. *International Journal of Forecasting*, *5*, 559–583.

Cohen, W. W. (1995). Fast effective rule induction. In *The Twelfth International Conference on Machine Learning*, pp. 115–123 San Francisco, CA. Morgan Kaufmann.

Cooper, G. F., & Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, *9*, 309–347.

Craven, M. W., & Shavlik, J. W. (1996). Extracting tree-structured representations from trained networks. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 24–30 Cambridge, MA. MIT Press.

rites, R. H., & Barto, A. G. (1995). Improving elevator performance using reinforcement learning. In *Advances in Neural Information Processing Systems*, Vol. 8 San Francisco, CA. Morgan Kaufmann.

D'Ambrosio, B. (1993). Incremental probabilistic inference. In Heckerman, D., & Mamdani, A. (Eds.), *Ninth Annual Conference on Uncertainty on AI*, pp. 301–308. Morgan Kaufmann.

Dayan, P., & Hinton, G. (1993). Feudal reinforcement learning. In *Advances in Neural Information Processing Systems, 5*, pp. 271–278. Morgan Kaufmann, San Francisco, CA.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*(3), 142–150.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1976). Maximum likelihood from incomplete data via the EM algorithm. *Proceedings of the Royal Statistical Society, B*, *39*, 1–38.

Dietterich, T. G., & Bakiri, G. (1995). Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, *2*, 263–286.

Dietterich, T. G., & Kong, E. B. (1995). Machine learning bias, statistical bias, and statistical variance of decision tree algorithms. Tech. rep., Department of Computer Science, Oregon State University, Corvallis, Oregon. Available from `ftp://ftp.cs.orst.edu/pub/tgd/papers/tr-bias.ps.gz`.

Domingos, P., & Pazzani, M. (1996). Beyond independence: Conditions for the optimality of the simple Bayesian classifier. In Saitta, L. (Ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 105–112 San Francisco, CA. Morgan Kaufmann.

Duda, R. O., & Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. Wiley.

Fayyad, U. M., & Irani, K. B. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1022–1027 San Francisco. Morgan Kaufmann.

Flexner, S. B. (Ed.). (1983). *Random House Unabridged Dictionary, 2nd edition*. Random House, New York.

Freund, Y., & Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. Tech. rep., AT&T Bell Laboratories, Murray Hill, NJ.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In Saitta, L. (Ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 148–156 San Francisco, CA. Morgan Kaufmann.

Friedman, N., & Goldszmidt, M. (1996). Building classifiers using Bayesian networks. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1277–1284 Cambridge, MA. AAAI Press/MIT Press.

Fürnkranz, J., & Widmer, G. (1994). Incremental reduced error pruning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 70–77 San Francisco, CA. Morgan Kaufmann.

Geman, S., & Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *6*, 721–741.

Ghahramani, Z., & Jordan, M. I. (1996). Factorial hidden Markov models. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 472–478 Cambridge, MA. MIT Press.

Gilks, W., Thomas, A., & Spiegelhalter, D. (1993). A language and program for complex Bayesian modelling. *The Statistician*, *43*, 169–178.

Golding, A. R., & Roth, D. (1996). Applying Winnow to context-sensitive spelling correction. In Saitta, L. (Ed.), *Proceedings of the Thirteenth International Conference on Machine Learning*, pp. 182–190 San Francisco, CA. Morgan Kaufmann.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 261–268 San Francisco, CA. Morgan Kaufmann.

Hansen, L., & Salamon, P. (1990). Neural network ensembles. *IEEE Trans. Pattern Analysis and Machine Intell. 12*, 993–1001.

Hashem, S. (1993). *Optimal linear combinations of neural networks.* Ph.D. thesis, Purdue University, School of Industrial Engineering, Lafayette, IN.

Heckerman, D. (1996). A tutorial on learning with Bayesian networks. Tech. rep. MSR-TR-95-06, Microsoft Research, Advanced Technology Division, Redmond, WA.

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

Hyafil, L., & Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-Complete. *Information Processing Letters*, *5*(1), 15–17.

Jensen, F. V., Lauritzen, S. L., & Olesen, K. G. (1990). Bayesian updating in recursive graphical models by local computations. *Computational Statistical Quarterly*, *4*, 269–282.

Jensen, F. (1996). *An Introduction to Bayesian Networks.* Springer, New York.

John, G., Kohavi, R., & Pfleger, K. (1994). Irrelevant features and the subset selection problem. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 121–129 San Francisco, CA. Morgan Kaufmann.

Jordan, M. I., & Jacobs, R. A. (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, *6*(2), 181–214.

Kaelbling, L. P. (1993). Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 167–173 San Francisco, CA. Morgan Kaufmann.

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285.

Kanazawa, K., Koller, D., & Russell, S. (1995). Stochastic simulation algorithms for dynamic probabilistic networks. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 346–351 San Francisco, CA. Morgan Kaufmann.

Kira, K., & Rendell, L. A. (1992). A practical approach to feature selection. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 249–256 San Francisco, CA. Morgan Kauffman.

Kohavi, R., & Kunz, C. (1997). Option decision trees with majority votes. In *Proceedings of the Fourteenth International Conference on Machine Learning* San Francisco, CA. Morgan Kaufmann.

Kohavi, R., & Sahami, M. (1996). Error-based and entropy-based discretizing of continuous features. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* San Francisco. Morgan Kaufmann.

Kolen, J. F., & Pollack, J. B. (1991). Back propagation is sensitive to initial conditions. In *Advances in Neural Information Processing Systems*, Vol. 3, pp. 860–867 San Francisco, CA. Morgan Kaufmann.

Kong, E. B., & Dietterich, T. G. (1995). Error-correcting output coding corrects bias and variance. In Prieditis, A., & Russell, S. (Eds.), *The Twelfth International Conference on Machine Learning*, pp. 313–321 San Francisco, CA. Morgan Kaufmann.

Kononenko, I. (1994). Estimating attributes: Analysis and extensions of relief. In *Proceedings of the 1994 European Conference on Machine Learning*, pp. 171–182 Amsterdam. Springer Verlag.

Kononenko, I., Šimec, E., & Robnik-Šikonja, M. (1997). Overcoming the myopic of inductive learning algorithms with RELIEFF. *Applied Intelligence, In Press.*

Kučera, H., & Francis, W. N. (1967). *Computational analysis of present-day American English.* Brown University Press, Providence, RI.

Kwok, S. W., & Carter, C. (1990). Multiple decision trees. In Schachter, R. D., Levitt, T. S., Kannal, L. N., & Lemmer, J. F. (Eds.), *Uncertainty in Artificial Intelligence 4*, pp. 327–335. Elsevier Science, Amsterdam.

aird, J. E., Newell, A., & Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, *33*(1), 1–64.

Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, *2*, 285–318.

Littman, M. L., Cassandra, A., & Kaelbling, L. P. (1995). Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 362–370 San Francisco, CA. Morgan Kaufmann.

Lowe, D. G. (1995). Similarity metric learning for a variable-kernel classifier. *Neural Computation*, *7*(1), 72–85.

MacKay, D. (1992). A practical Bayesian framework for backpropagation networks. *Neural Computation*, *4*(3), 448–472.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, *22*, 159–195.

Mahadevan, S., & Kaelbling, L. P. (1996). The national science foundation workshop on reinforcement learning. *AI Magazine*, *17*(4), 89–97.

McCallum, R. A. (1995). Instance-based utile distinctions for reinforcement learning with hidden state. In *Proceedings Twelfth International Conference on Machine Learning*, pp. 387–396 San Francisco, CA. Morgan Kaufmann.

Mehta, M., Agrawal, R., & Rissanen, J. (1996). SLIQ: A fast scalable classifier for data mining. In *Lecture Notes in Computer Science*, pp. 18–32 New York. Springer.

Merz, C. J., & Murphy, P. M. (1996). UCI repository of machine learning databases. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Miller, A. J. (1990). *Subset selection in regression*. Chapman and Hall.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O., & Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Moore, A. W., & Lee, M. S. (1994). Efficient algorithms for minimizing cross validation error. In Cohen, W., & Hirsh, H. (Eds.), *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 190–198 San Francisco, CA. Morgan Kaufmann.

Munro, P., & Parmanto, B. (1997). Competition among networks improves committee performance. In *Advances in Neural Information Processing Systems*, Vol. 9, p. In Press.

Musick, R., Catlett, J., & Russell, S. (1992). Decision theoretic subsampling for induction on large databases. In Utgoff, P. E. (Ed.), *Proceedings of the Tenth International Conference on Machine Learning*, pp. 212–219 San Francisco, CA. Morgan Kaufmann.

Neal, R. (1993). Probabilistic inference using Markov chain Monte Carlo methods. Tech. rep. CRG-TR-93-1, Department of Computer Science, University of Toronto, Toronto, CA.

Ok, D., & Tadepalli, P. (1996). Auto-exploratory average reward reinforcement learning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 881–887 Cambridge, MA. AAAI Press/MIT Press.

Opitz, D. W., & Shavlik, J. W. (1996). Generating accurate and diverse members of a neural-network ensemble. In *Advances in Neural Information Processing Systems, 8*, pp. 535–541 Cambridge, MA. MIT Press.

Parmanto, B., Munro, P. W., & et al, H. R. D. (1994). Neural network classifier for hepatoma detection. In *Proceedings of the World Congress on Neural Networks*, Vol. 1 Mahwah, NJ. Lawrence Earlbaum Associates.

Parmanto, B., Munro, P. W., & Doyle, H. R. (1996). Improving committee diagnosis with resampling techniques. In Touretzky, D. S., Mozer, M. C., & Hesselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 882–888 Cambridge, MA. MIT Press.

arr, R., & Russell, S. (1995). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1088–1094 San Francisco, CA. Morgan Kaufmann.

Perrone, M. P., & Cooper, L. N. (1993). When networks disagree: Ensemble methods for hybrid neural networks. In Mammone, R. J. (Ed.), *Neural networks for speech and image processing*. Chapman and Hall.

Press, W. H., Flannery, B. P., Teukolsky, S. A., & Verrerling, W. T. (1992). *Numerical recipes in C : The art of scientific computing, 2nd Edition*. Cambridge University Press, Cambridge, England.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.

Quinlan, J. R. (1993). *C4.5: Programs for Empirical Learning*. Morgan Kaufmann, San Francisco, CA.

Quinlan, J. R. (1996). Bagging, boosting, and C4.5. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 725–730 Cambridge, MA. AAAI Press/MIT Press.

Quinlan, J. R., & Rivest, R. L. (1989). Inferring decision trees using the minimum description length principle. *Information and Computation*, *80*, 227–248.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Raviv, Y., & Intrator, N. (1996). Bootstrapping with noise: An effective regularization technique. *Connection Science*, *8*(3–4), 355–372.

Revow, M., Williams, C. K. I., & Hinton, G. E. (1996). Using generative models for handwritten digit recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *18*(6), 592–606.

Ricci, F., & Aha, D. W. (1997). Extending local learners with error-correcting output codes. Tech. rep., Naval Center for Applied Research in Artificial Intelligence.

Rosen, B. E. (1996). Ensemble learning using decorrelated neural networks. *Connection Science*, *8*(3–4), 373–384.

Russell, S., Binder, J., Koller, D., & Kanazawa, K. (1995). Local learning in probabilistic networks with hidden variables. In *Proc. Fourteenth International Joint Conference on Artificial Intelligence*, pp. 1146–1152 San Francisco, CA. Morgan Kaufmann.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*, 211–229.

Schapire, R. E. (1997). Using output codes to boost multiclass learning problems. Tech. rep., AT&T Research.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In Utgoff, P. (Ed.), *Proceedings of the Tenth International Conference on Machine Learning*, pp. 298–305 San Francisco, CA. Morgan Kaufmann.

Shafer, J., Agrawal, R., & Mehta, M. (1996). SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the Twenty-Second VLDB Conference*, pp. 544–555 San Francisco, CA. Morgan Kaufmann.

Singh, S., & Bertsekas, D. (1997). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Advances in Neural Information Processing Systems*, Vol. 10. MIT Press, Cambridge, MA.

Singh, S. P. (1992). Transfer of learning by composing solutions to elemental sequential tasks. *Machine Learning*, *8*(3), 323–340.

Smyth, P., Heckerman, D., & Jordan, M. I. (1997). Probabilistic independence networks for hidden Markov probability models. *Neural Computation*, *9*(2), 227–270.

Spiegelhalter, D., Dawid, A., Lauritzen, S., & Cowell, R. (1993). Bayesian analysis in expert systems. *Statistical Science*, *8*, 219–282.

Spirtes, P., Glymour, C., & Scheines, R. (1993). *Causation, Prediction, and Search*. Springer-Verlag, New York. Available online at `http://hss.cmu.edu/html/departments/philosophy/TETRAD.BOOK/book.html`.

pirtes, P., & Meek, C. (1995). Learning Bayesian networks with discrete variables from data. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pp. 294–299 San Francisco. Morgan Kaufmann.

Stolcke, A., & Omohundro, S. M. (1994). Best-first model merging for hidden Markov model induction. Tech. rep. TR-94-003, International Computer Science Institute, Berkeley, CA.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257–278.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, *28*(3), 58–68.

Tsitsiklis, J. N., & Van Roy, B. (1996). An analysis of temporal-difference learning with function approximation. Tech. rep., Massachusetts Institute of Technology, Laboratory for Information and Decision Systems, Cambridge, MA.

Tumer, K., & Ghosh, J. (1996). Error correlation and error reduction in ensemble classifiers. *Connection Science*, *8*(3–4), 385–404.

Verma, T., & Pearl, J. (1990). Equivalence and synthesis of causal models. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pp. 220–227 San Francisco, CA. Morgan Kaufmann.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, King's College, Oxford. (To be reprinted by MIT Press.).

Watkins, C. J., & Dayan, P. (1992). Technical note: Q-learning. *Machine Learning*, *8*, 279–292.

Wettschereck, D., Aha, D. W., & Mohri, T. (1997). A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, *10*, 1–37.

Wettschereck, D., & Dietterich, T. G. (1995). An experimental comparison of the nearest-neighbor and nearest-hyperrectangle algorithms. *Machine Learning*, *19*, 5–27.

Wolpert, D. (1992). Stacked generalization. *Neural Networks*, *5*(2), 241–260.

Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *1995 International Joint Conference on Artificial Intelligence*, pp. 1114–1120. AAAI/MIT Press, Cambridge, MA.

Zhang, X., Mesirov, J. P., & Waltz, D. L. (1992). Hybrid system for protein secondary structure prediction. *Journal of Molecular Biology*, *225*.

Zweben, M., Daun, B., & Deale, M. (1994). Scheduling and rescheduling with iterative repair. In Zweben, M., & Fox, M. S. (Eds.), *Intelligent Scheduling*, chap. 8, pp. 241–255. Morgan Kaufmann, San Francisco, CA.