*

REx: Learning A Rule and Exceptions

Ethem Alpaydin
International Computer Science Institute
1947 Center St Suite 600, Berkeley, CA 94704-1198
ethem@icsi.berkeley.edu

TR-97-040

October 1997

Abstract

We propose a method where the dataset is explained as a "rule" and a set of "exceptions" to the rule. The rule is a parametric model valid over the whole input space and exceptions are nonparametric and local. This approach is applicable both to function approximation and classification. We explain how the rule and exceptions can be learned using cross-validation. We investigate three ways of combining the rule and exceptions: (1) In a multistage approach, if the rule is confident of its output, we use it; otherwise, output is interpolated from a table of stored exceptions. (2) In a multiexpert approach, the exceptions are defined as gaussian units just like in a radial-basis functions network; the rule can be seen as a parametric input-dependent offset to which the gaussian exceptions are added. (3) The rule and exceptions can be written as a mixture model like in Mixtures of Experts and they can be combined in a cooperative or competitive manner. The system can be trained using a gradient based, or in the case of (3) EM, algorithm. The model can be combined with Hidden Markov models for sequence processing. We analyse REx as an arcing method and compare it with bagging and boosting. The proposed approaches are tested on several datasets in terms of generalization accuracy, memory requirement, and training time with significant performance.

*On leave from Department of Computer Engineering, Boğaziçi University, Istanbul TR-80815 Turkey.

1 Introduction

A great percentage of the training cases in many applications can be explained by a simple rule with a small number of exceptions. Holte (1993), based on a survey in the machine learning literature and on his own results on 16 datasets, concludes that very simple classification rules perform well on many datasets. Simple rules have less complexity, are easy to explain and understand, are faster to train and use, and can generalize with smaller training sets. Our previous experience with neural networks applied to classification problems indicate a small difference in accuracy between linear perceptrons and multilayer perceptrons with many hidden units. Though these latter are capable of representing complex nonlinear functions, training time may be large and such structures do not allow incorporating prior knowledge nor extraction of rules from a trained network.

Modular approaches have been proposed where a task is divided into subtasks. Examples are mixture models like Mixture of Experts (ME) and decision trees like CART and ID3 where the input space is divided, in a "hard" or "soft" manner, into a number of regions and simple, constant or linear, surfaces are fit within these regions. These learn much faster than multilayer perceptrons with sigmoidal hidden units and allow knowledge extraction.

In the work we describe here we divide the task into two, though in a manner somehow different from techniques described above. We see learning a task as learning of a "rule" and a set of "exceptions" not covered by the rule. The rule-learner is fast and simple, and covers a large part of the patterns. The exception-learner is consulted rarely and can be slower and costlier but should be able to complement the rule-learner by handling correctly those patterns rejected by the rule-learner. Defining a dataset as a simple general rule and a set of localized exceptions is easy to understand and describe, allowing knowledge extraction. It allows incorporating previous knowledge as a rule. It is also cheaper to implement and faster to train.

We propose to use a parametric linear model for the rule-learner and a nonparametric, memory-based scheme for the exception-learner. The linear model is trained over the whole input space and the exception-learner interpolates from a collection of patterns rejected by the rule-learner. This approach is applicable both to function approximation and classification.

Depending on the dataset, the rule-learner may be a model other than linear, e.g., a decision tree. The important requirement is that it should be simple, easy to train and use. One can also envisage a nested sequence of rules each one covering some of the patterns and leaving the rest as exceptions to the following one, the nesting terminated by the final, memory-based exception-learner.

This work is related to several earlier work. First, in the pattern recognition literature, there is work on *multistage* pattern recognition (Pudil et al., 1992) where inputs rejected by a first stage are handled by a second stage using costlier features or decision making mechanism which is too expensive to use for all inputs. In the neural network literature, the Boosting algorithm (Freund and Schapire, 1996) constructs similarly a sequence of classifiers with each one trained on patterns on which the previous err(s). Multistage approaches differ from *multiexpert* methods in that, in the latter, for a given input all learners respond in parallel after which their responses are combined. Examples to multiexpert methods are voting (Xu et al., 1992), bagging (Breiman, 1996), stacking (Wolpert, 1992).

Second, exceptions can be seen as local models combined with a global rule model and this can be formalized as a Radial-Basis Function (RBF) network or better still, as a mixture model similar to the Mixture of Experts architecture (Jacobs et al., 1991). Third, we utilize the Generalized Linear Model (GLIM) theory (McCullagh and Nelder, 1989) to describe the rule and exceptions and use maximum likelihood estimation to estimate the model parameters. This maximization can be gradient-based or may use the Expectation-Maximization (EM) algorithm.

The paper is organized as follows: In Section 2, we give the probabilistic setting on which the rule and exceptions are built using the GLIM theory. In Section 3, we discuss a technique to learn the exceptions using cross-validation. In Sections 4, 5, and 6, we discuss three ways to combine the rule and exceptions. These are respectively Multistage, Multiexpert with Gaussian exceptions, and

Multiexpert as a mixture model. Section 7 proposes to combine the model with Hidden Markov models to learn time series. We contrast the model to arcing methods (Breiman, 1997) like bagging and boosting in Section 8. Simulation results are given in Section 9. Section 10 concludes.

2 Learning the Rule

2.1 The Model

We discuss supervised learning where input values are elements of $\Re^m$ and there are n outputs. In the case of regression, the outputs are elements of $\Re^n$ and in the case of classification, they are elements of $\{0,1\}^n$. We have a dataset of pairs of inputs and associated outputs $\mathcal{X} = \{\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}\}_{t=1}^N$.

Relating the input to the output is a probabilistic model $P(\boldsymbol{y}|\boldsymbol{x}, \Phi)$. We assume the density P to be a member of the exponential family of densities (McCullagh and Nelder, 1989; Jordan and Jacobs, 1994) with a "location" parameter $\boldsymbol{\mu}$ defined using a link function f of a linear predictor of the input $\boldsymbol{x}$

$$\boldsymbol{\mu} = f(U\boldsymbol{x}) \tag{1}$$

and "dispersion" parameter ϕ. The parameter vector Φ includes the weight matrix U and ϕ. The log likelihood is

$$l(\Phi; \mathcal{X}) = \sum_t \log P(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(t)}, \Phi) \tag{2}$$

Parameters Φ are computed by maximizing this likelihood depending on the particular P model chosen.

2.2 Gaussian Density for Regression

In the case of regression the probabilistic component is taken as an n-dimensional hyperspheric gaussian, i.e., $P(\boldsymbol{y}|\boldsymbol{x}, \Phi) \sim \mathcal{N}(\boldsymbol{\mu}, \sigma^2 I)$. The link function f is identity.

$$P(\boldsymbol{y}|\boldsymbol{x}, \Phi) = \frac{1}{(2\pi)^{n/2}\sigma^n} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu_i)^2\right] \tag{3}$$

$$\mu_i = \boldsymbol{u}_i^T \boldsymbol{x}, i = 1, \ldots, n \tag{4}$$

In this case maximizing Eq (2) reduces to minimizing the sum of squared errors over all classes (indexed i) over all patterns (indexed t)

$$E(\Phi; \mathcal{X}) = \sum_t \sum_i [y_i^{(t)} - \mu_i^{(t)}]^2 \tag{5}$$

2.3 Multinomial Density for Multiway Classification

In n-way classification we use the multinomial density, i.e., $P(\boldsymbol{y}|\boldsymbol{x}, \Phi) \sim \mathrm{Mult}_n(1, \boldsymbol{\mu})$. The link function is softmax. This is the multinomial logit model which is a specific form of GLIM.

$$P(\boldsymbol{y}|\boldsymbol{x}, \Phi) = \prod_{i=1}^{n} \mu_i^{y_i} \tag{6}$$

$$\mu_i = \frac{\exp[\boldsymbol{u}_i^T \boldsymbol{x}]}{\sum_k \exp[\boldsymbol{u}_k^T \boldsymbol{x}]}, i = 1, \ldots, n \tag{7}$$

where $y_i \in \{0, 1\}$ and $\sum_i y_i = 1$. In this case maximizing Eq (2) reduces to minimizing the cross-entropy

$$E(\Phi; \mathcal{X}) = -\sum_t \sum_i y_i^{(t)} \log \mu_i^{(t)} \tag{8}$$

A special case is binary classification with $n = 2$ where multinomial reduces to Bernoulli and softmax reduces to the logistic function.

3 Learning the Exceptions

A model trained as explained in Section 2 applies to a large majority of the input patterns but not to all and in the REx model, we want to determine where such patterns are and we want to treat them as exceptions and learn them separately.

We learn exception positions using cross-validation. Let us say $(\boldsymbol{x}', \boldsymbol{y}')$ is a pattern randomly chosen from $\mathcal{X}$ and we use $\mathcal{X}_0 = \mathcal{X} - \{(\boldsymbol{x}', \boldsymbol{y}')\}$, the set of $N-1$ remaining patterns, to compute Φ_0 that maximize likelihood and train the rule $\boldsymbol{\mu}_0$. If the likelihood of the holded out pattern given the current rule parameters Φ_0 is less than a certain *confidence threshold*, θ, that is if the rule cannot predict the pattern with enough certainty, we take it as not being covered by the rule, that is as an exception. We check if

$$P(\boldsymbol{y}'|\boldsymbol{x}', \Phi_0) < \theta$$

In the case of regression, this is equivalent to checking

$$\sum_i (y_i' - \mu_{0i})^2 > \theta'$$

and in classification for i such that $y_i' = 1$ we check if

$$\mu_{0i} < \theta$$

We repeat this N times for all patterns and find the exceptions and store their $\boldsymbol{x}'$ as position vectors $\boldsymbol{v}_j$ and $\boldsymbol{y}'$ as desired response vectors $\boldsymbol{u}_j, j = 1 \ldots J$ where J is the number of exceptions. Thus exception j defines a constant fit $\boldsymbol{\mu}_j = f(\boldsymbol{u}_j)$ (not a function of the input) when $\boldsymbol{x}$ is close to $\boldsymbol{v}_j$. After finding the exceptions, the rule $\boldsymbol{\mu}_0 = f(U_0\boldsymbol{x})$ is trained and combined with the exceptions in one of the three ways explained in Sections 4, 5, and 6.

The reason why we use cross-validation is that, to find exceptions, we need to use data different from the data used for training the rule as otherwise with a complex rule-learner we may memorize all patterns and have high confidence on all. Leave-one-out can be costly with large samples and one can use k-fold cross-validation.

In our experiments when we find a sample $(\boldsymbol{x}', \boldsymbol{y}')$ for which the rule is not confident, we take it as an exception. When memory space is a premium, at the expense of more computation, one can first check if another exception has already been stored that is close enough in the input space and that generates a close enough output. In such a case, we may assume that this previously stored exception would generate a correct output and that there is no need to store the newcoming exception. That is we take $(\boldsymbol{x}', \boldsymbol{y}')$ as an exception if

$$\begin{aligned} & P(\boldsymbol{y}'|\boldsymbol{x}', \Phi_0) < \theta \text{ AND} \\ & \quad \nexists(\boldsymbol{x}_c, \boldsymbol{y}_c) \in \mathcal{X}_x \text{ s.t. } \boldsymbol{x}_c \approx_x \boldsymbol{x}' \text{ AND } \boldsymbol{y}_c \approx_y \boldsymbol{y}' \end{aligned}$$

where $\mathcal{X}_x$ is the set of exceptions already stored and $\approx_x$ means "close enough" in the $\boldsymbol{x}$ space, formalized by defining a suitable distance measure and threshold.

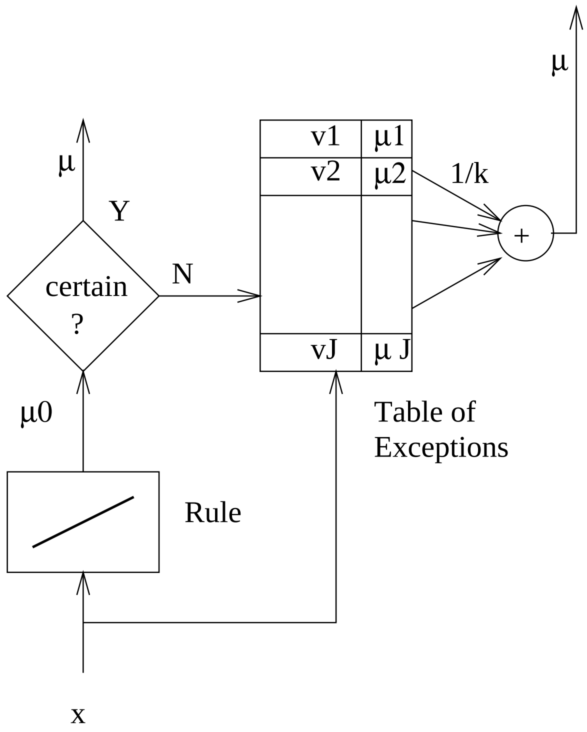

Figure 1: Block diagram of a multistage model with a k-nn based exception-learner.

It may also be the case that the rule is not trained in this manner and may be programmed or in some other way determined a priori by the system designer. After converting its output to probabilities, one can still use this approach to determine the exceptions and combine them with the rule to get an overall system.

4 Multistage Method with k-NN Based Exception-Learner

The first way to combine the rule and exception learners is by a multistage method where either of the two is employed (Alpaydın and Kaynak, 1998). Given a pattern $\boldsymbol{x}$, we first check if the rule-learner is certain. If it is, we use the response provided by the rule. Otherwise we consult the exception-learner. If the rule-learner indeed is certain for a large percentage of the test cases then the costlier exception-learner is consulted rarely.

The measure of "certainty" of the rule is the conditional data likelihood. From the data sample we remove all exceptions and find the patterns that trained the rule and using these estimate $p(\boldsymbol{x}|\text{Rule})$. The rule is not "certain" of its prediction and rejects if

$$p(\boldsymbol{x}|\text{Rule}) < \theta_0$$

This requires an extra learner to approximate this conditional density. In the case of classification we use a shortcut which is used in pattern recognition for rejections. We reject if the highest posterior is less than a given threshold, that is if

$$\max_i P(\omega_i|\boldsymbol{x}) < \theta_0$$

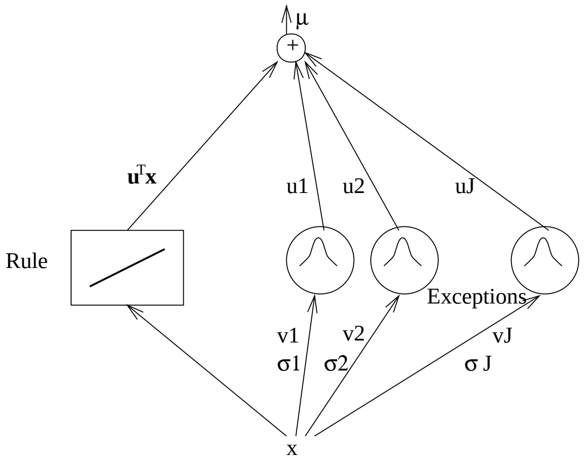

Figure 2: Block diagram of combining the rule with an rbf based exception-learner.

The exception-learner has, in a table, exception positions $\boldsymbol{v}_j$ and their corresponding desired outputs, $\boldsymbol{u}_j$. If the rule rejects, we respond by interpolating from this table using the k-nearest neighbor algorithm. During test, given $\boldsymbol{x}$, if the rule-learner is not certain, we find the k closest $\boldsymbol{v}_j$ to $\boldsymbol{x}$ in the table and average over their $\mu_{ji} = f(u_{ji})$. If the rule-learner is certain, we use its prediction $\mu_{0i} = f(\boldsymbol{u}_i^T \boldsymbol{x})$, for all outputs $i = 1, \ldots, n$ (Fig 1).

$$
\begin{aligned}
\mu_i &= \begin{cases} \mu_{0i} & \text{if rule is certain} \\ \sum_j g_j \mu_{ji} & \text{otherwise} \end{cases} \quad i = 1, \ldots, n \\
g_j &= \begin{cases} 1/k & \text{if } \boldsymbol{v}_j \in \{k \text{ nearest neighbors of } \boldsymbol{x}\} \\ 0 & \text{otherwise} \end{cases} \quad j = 1, \ldots, J
\end{aligned} \tag{9}
$$

The advantage of this scheme is that because the rule covers most of the training cases, only a small percentage is stored as exceptions. Similarly during test, most patterns are handled by the rule, rarely requiring the costlier k-nn. Even when the k-nn is used because we search for the k nearest neighbors in a much smaller set, we gain considerably from computation. For example if 10% of the training set is stored as exceptions and if the rule rejects 20% of the test cases, then the number of distance computations is 2% of what one would have if one used k-nn proper.

5 Multiexpert Method with Gaussian Exceptions

The accuracy of the k-nn based exception-learner can be increased at the expense of more computation. First, instead of having all k exceptions having equal effect and the remaining none, one can use a kernel function to have the weight of an exception decreasing continuously with increasing distance

$$
p_j = \exp\left[-\frac{1}{2\sigma_j^2}||\boldsymbol{x} - \boldsymbol{v}_j||^2\right], j = 1, \ldots, J \tag{10}
$$

where $\boldsymbol{v}_j$ define centers and σ_j^2 define spreads of the gaussian bases. As a heuristic, σ_j is initialized to be half of the distance to the closest other exception. Instead of a multistage, one-of-two, combination, we combine the rule and exceptions to find an overall output (Fig. 2).

$$\mu_i = f\left(\boldsymbol{u}_i^T \boldsymbol{x} + \sum_{j=1}^{J} p_j u_{ji}\right) \tag{11}$$

The exceptions operate like a radial-basis function network where p_j are the Gaussian basis functions and u_{ji} are the weights from the Gaussian units to the output units which are now modifiable. The rule is then a parametric, e.g., linear, input-dependent offset on which the exceptions are superposed. The rule's parameters $\boldsymbol{u}_i$ and the exceptions u_{ji} can be trained together on a given dataset. In regression, we minimize

$$E(\Phi; \mathcal{X}) = \sum_t \sum_i \left[y_i^{(t)} - \mu_i^{(t)}\right]^2 \tag{12}$$

Using gradient-descent, we update not only the rule but also the effects of the exceptions

$$\begin{aligned} \Delta \boldsymbol{u}_{0i} &= \eta(y_i - \mu_i)\boldsymbol{x} \\ \Delta u_{ji} &= \eta(y_i - \mu_i)p_j, j = 1, \ldots, J \end{aligned} \tag{13}$$

where η is the learning rate. This is an online method. Batch updates can be done by summing over all patterns.

This can also be seen as increasing the dimensionality by adding new input dimensions through the Gaussian basis functions. We generate a new input vector of dimensionality $m + J$ such that

$$\boldsymbol{x}_{new} = [x_1, x_2, \ldots, x_m, p_1(\boldsymbol{x}), p_2(\boldsymbol{x}), \ldots, p_J(\boldsymbol{x})]^T$$

As our simulation results suggest, this allows better learning of the task.

At the expense of more computation and possibly slower convergence, exception centers $\boldsymbol{v}_j$ and spreads σ_j can also be finetuned in a supervised manner by gradient-descent. For example again for regression

$$\Delta \boldsymbol{v}_j = \eta \left[\sum_i (y_i - \mu_i) u_{ji}\right] p_j \frac{(\boldsymbol{x} - \boldsymbol{v}_j)}{\sigma_j^2} \tag{14}$$

$\Delta\sigma_j$ can similarly be computed. In classification, we minimize

$$E(\Phi; \mathcal{X}) = -\sum_t \sum_i y_i^{(t)} \log \mu_i^{(t)} \tag{15}$$

where

$$\mu_i = \frac{\exp\left[\boldsymbol{u}_i^T \boldsymbol{x} + \sum_{j=1}^{J} p_j u_{ji}\right]}{\sum_k \exp\left[\boldsymbol{u}_k^T \boldsymbol{x} + \sum_{j=1}^{J} p_j u_{jk}\right]} \tag{16}$$

6 Mixture of Rule and Exceptions

6.1 The Model

Another possibility is to define also a position and spread for the rule which defines where the rule will be active, instead of having it active all the time. In this way, the rule and exceptions will interfere less. The rule-learner, indexed 0, has a position, $\boldsymbol{v}_0$, and spread, σ_0^2, based on which its effect g_0 is computed. $\boldsymbol{v}_0$ is initialized as the mean of all patterns covered by the rule and σ_0 to be

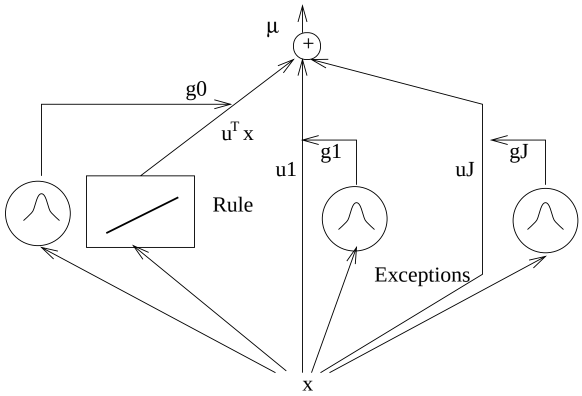

Figure 3: Block diagram of combining the rule and exceptions as a mixture model.

half of the distance between the mean and the most distant pattern. This structure is a Gaussian mixture of rule and exceptions (Fig. 3). The probability of generating $\boldsymbol{y}$ from $\boldsymbol{x}$ using a mixture model is

$$p(\boldsymbol{y}|\boldsymbol{x}) = \sum_{j=0}^{J} P(j|\boldsymbol{x}) P(\boldsymbol{y}|j, \boldsymbol{x}) \tag{17}$$

where $P(j|\boldsymbol{x})$ are the mixing proportions conditioned on the input $\boldsymbol{x}$ and $P(\boldsymbol{y}|j,\boldsymbol{x})$ are the components relating the input to the output

$$\begin{aligned} P(j|\boldsymbol{x}) &= \frac{P(j)P(\boldsymbol{x}|j)}{\sum_k P(k)P(\boldsymbol{x}|k)} \\ g_j &= \frac{\alpha_j \exp\left[-\|\boldsymbol{x}-\boldsymbol{v}_j\|^2/2\sigma_j^2\right]}{\sum_k \alpha_k \exp\left[-\|\boldsymbol{x}-\boldsymbol{v}_k\|^2/2\sigma_k^2\right]}, j = 0, \ldots, J \end{aligned} \tag{18}$$

We assume $p(\boldsymbol{x}|j)$ to be Gaussian with mean $\boldsymbol{v}_j$ and diagonal covariance matrix. We do not use full covariance matrices to have the number of parameters linear in the input dimensionality. We take α_j to be equal and constant. Because of the normalization, as the rule has larger variance, it will be active when input is not very close to one of the exceptions (Fig. 4).

The combination is a weighted sum where weights are provided by the gating values g_j (Fig. 3)

$$\mu_i = f\left(\sum_{j=0}^{J} g_j u_{ji}\right) \text{ where } u_{0i} = \boldsymbol{u}_i^T \boldsymbol{x} \tag{19}$$

This structure is a special case of the Mixture of Experts architecture (Jacobs et al., 1991; Jordan and Jacobs, 1994; Alpaydın and Jordan, 1996) where one expert (rule) gives a linear fit and the others (exceptions) give constant fits. g_j are the gating values which in the Mixtures of Experts are defined as

$$g_j = \frac{\exp[\boldsymbol{v}_j^T \boldsymbol{x}]}{\sum_k \exp[\boldsymbol{v}_k^T \boldsymbol{x}]} \tag{20}$$

which parametrize discriminant surfaces between experts using softmax. We instead describe the density directly in the input space to be able to initialize $\boldsymbol{v}_j$ by exception positions.

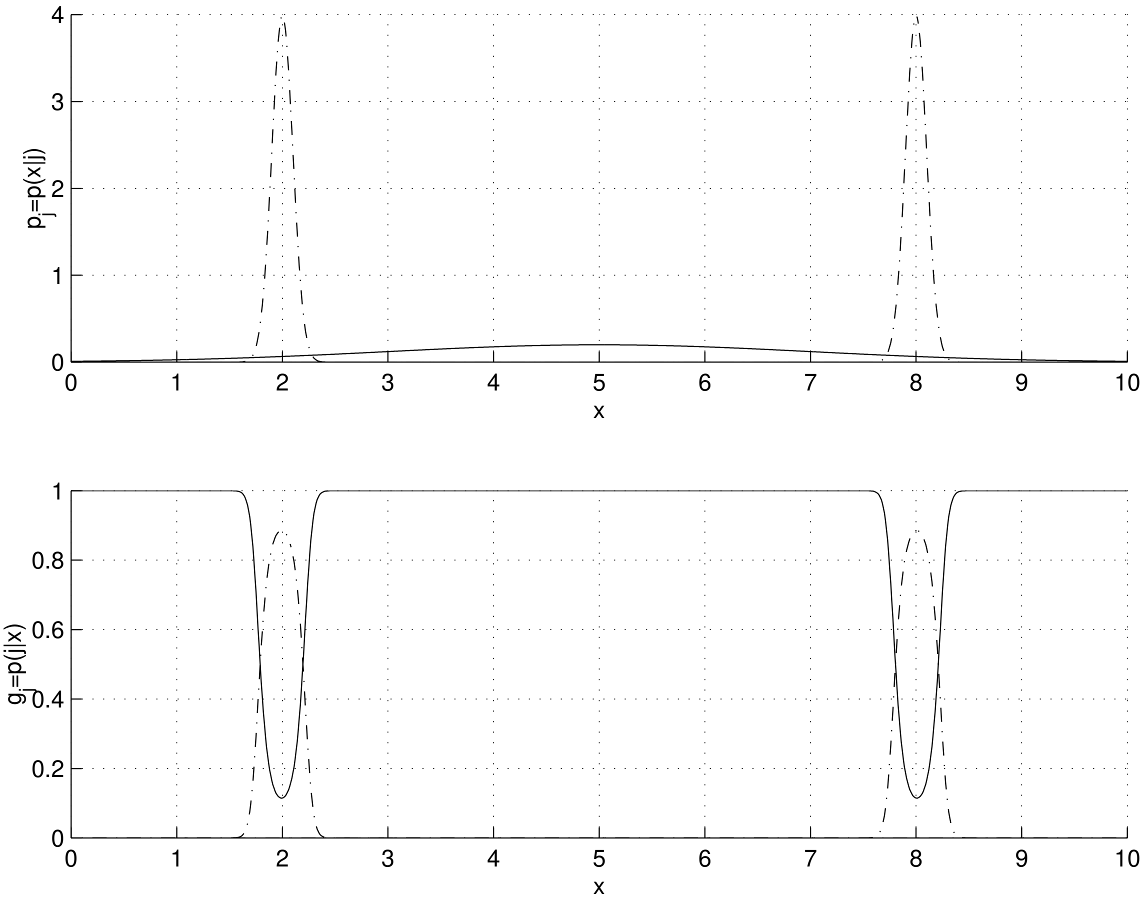

Figure 4: Before and after normalization. Rule $p_0 \sim \mathcal{N}(5, 2)$ and exceptions are $p_1 \sim \mathcal{N}(2, 0.1)$ and $p_2 \sim \mathcal{N}(8, 0.1)$. Priors are $P(0) = 0.8, P(1) = 0.1, P(2) = 0.1$. After normalization because the rule has larger variance, it is effective, i.e., $P(0|x) \approx 1$, where exceptions are not.

6.2 Cooperative Model

In a cooperative setting, we minimize the difference between the required output and the overall output. In regression, we minimize

$$E(\Phi; \mathcal{X}) = \sum_t \sum_i \left[y_i^{(t)} - \mu_i^{(t)} \right]^2 \tag{21}$$

Using gradient-descent, we get

$$\begin{aligned}
\Delta \boldsymbol{u}_{0i} &= \eta(y_i - \mu_i) g_0 \boldsymbol{x} \\
\Delta u_{ji} &= \eta(y_i - \mu_i) g_j \\
\Delta \boldsymbol{v}_j &= \eta \left[\sum_i (y_i - \mu_i) g_j (u_{ji} - \mu_i) \right] \left[\frac{\boldsymbol{x} - \boldsymbol{v}_j}{\sigma_j^2} \right], j = 1, \ldots, J
\end{aligned} \tag{22}$$

In classification, we minimize

$$E(\Phi; \mathcal{X}) = -\sum_t \sum_i y_i^{(t)} \log \mu_i^{(t)} \tag{23}$$

6.3 Competitive Model

The log likelihood of the sample using a mixture model is ($\Phi = \{\boldsymbol{v}_j, \boldsymbol{u}_j\}_{j=0}^J$)

$$l(\Phi; \mathcal{X}) = \sum_t \log \sum_j g_j(\boldsymbol{x}^{(t)} | \boldsymbol{v}_j) P(\boldsymbol{y}^{(t)} | \boldsymbol{x}^{(t)}, \boldsymbol{u}_j) \tag{24}$$

For the case of regression the log likelihood for the Gaussian model is

$$l(\Phi; \mathcal{X}) = \sum_t \log \sum_j g_j \exp \left[-\frac{1}{2} \sum_i (y_i^{(t)} - \mu_{ji}^{(t)})^2 \right] \tag{25}$$

Using gradient-ascent, we get

$$\begin{aligned}
\Delta \boldsymbol{u}_{0i} &= \eta(y_i - \mu_{0i}) h_0 \boldsymbol{x} \\
\Delta u_{ji} &= \eta(y_i - \mu_{ji}) h_j \\
\Delta \boldsymbol{v}_j &= \eta(h_j - g_j) \left[\frac{\boldsymbol{x} - \boldsymbol{v}_j}{\sigma_j^2} \right], j = 1, \ldots, J
\end{aligned} \tag{26}$$

where

$$\begin{aligned}
h_j &= \frac{g_j \exp[-(1/2) \sum_i (y_i - \mu_{ji})^2]}{\sum_k g_k \exp[-(1/2) \sum_i (y_i - \mu_{ki})^2]} \\
P(j | \boldsymbol{y}, \boldsymbol{x}) &= \frac{P(j | \boldsymbol{x}) P(\boldsymbol{y} | j, \boldsymbol{x})}{\sum_k P(k | \boldsymbol{x}) P(\boldsymbol{y} | k, \boldsymbol{x})}
\end{aligned} \tag{27}$$

The difference between Eq. (23) and Eq. (27) is that in the latter we decrease the difference between the required output and the output of the expert (rule or exception) that is responsible from this pattern, as given by h_j. Eq. (14) uses p_j which is $p(\boldsymbol{x}|j)$, Eq. (23) uses g_j, which is $P(j|\boldsymbol{x})$, taking other experts (rule or exceptions) into account and Eq. (27) uses h_j which is $P(j|\boldsymbol{y}, \boldsymbol{x})$ also taking the supervised output into account.

In the case of classification, we minimize

$$\begin{aligned} l(\Phi; \mathcal{X}) &= -\sum_t \log \sum_j g_j \prod_i \mu_{ji}^{y_i} \\ &= -\sum_t \log \sum_j g_j \exp\left[\sum_i y_i^{(t)} \log \mu_{ji}^{(t)}\right] \end{aligned} \tag{28}$$

6.4 EM Algorithm

One can also use the EM algorithm as applied to the mixture of experts architecture (Jordan and Jacobs, 1994; Jordan and Xu, 1995; Xu et al., 1994). In the E-step at iteration l, we compute h_j for each pattern t, as given in Eq. (27) given the current parameter values $\Phi^{(l)} = \{\boldsymbol{v}_j^{(l)}, \boldsymbol{u}_j^{(l)}\}_j$

$$\begin{aligned} h_j^{(l)}(\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)}) &= P(j|\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)}) \\ &= \frac{P(j|\boldsymbol{x}^{(t)}, \boldsymbol{v}^{(l)})P(\boldsymbol{y}^{(t)}|j, \boldsymbol{x}^{(t)}, \boldsymbol{u}^{(l)})}{\sum_k P(k|\boldsymbol{x}^{(t)}, \boldsymbol{v}^{(l)})P(\boldsymbol{y}^{(t)}|k, \boldsymbol{x}^{(t)}, \boldsymbol{u}^{(l)})} \end{aligned} \tag{29}$$

The objective function divides into two new objective functions, one for the rule and exception mappings and one for the positions, called experts and gating in the mixture of experts terminology. Denoting $h_j^{(l)}(\boldsymbol{y}^{(t)}, \boldsymbol{x}^{(t)})$ by $h_j^{l,t}$ to save from ink and eye strain, we get

$$\begin{aligned} \mathcal{Q}^e(\boldsymbol{u}) &= \sum_t \sum_j h_j^{l,t} \log P(\boldsymbol{y}^{(t)}|j, \boldsymbol{x}^{(t)}, \boldsymbol{u}^{(l)}) \\ \mathcal{Q}^g(\boldsymbol{v}) &= \sum_t \sum_j h_j^{l,t} \log P(j|\boldsymbol{x}^{(t)}, \boldsymbol{v}^{(l)}) \end{aligned} \tag{30}$$

In the M-step, we find the new estimate $\Phi^{(l+1)} = \{\boldsymbol{u}_j^{(l+1)}, \boldsymbol{v}_j^{(l+1)}\}_j$ that maximizes this

$$\begin{aligned} \boldsymbol{u}^{(l+1)} &= \arg\max_{\boldsymbol{u}} \mathcal{Q}^e(\boldsymbol{u}) \\ \boldsymbol{v}^{(l+1)} &= \arg\max_{\boldsymbol{v}} \mathcal{Q}^g(\boldsymbol{v}) \end{aligned} \tag{31}$$

Xu et al. (1994) has shown that if the gating network works directly in the input space as here, rather than estimating posteriors directly using softmax as done in the original mixture of experts, that is if

$$g_j = \frac{\alpha_j P(\boldsymbol{x}|j)}{\sum_k \alpha_k P(\boldsymbol{x}|k)} \tag{32}$$

where α_j are the prior expert probabilities, M-step for the gating networks can be solved analytically if $P(\boldsymbol{x}|j)$ is from the exponential family. If $P(\boldsymbol{x}|j) \sim \mathcal{N}(\boldsymbol{v}_j, \Sigma_j)$, then the M-step updates for the gating become

$$\alpha_j^{(l+1)} = \frac{1}{N} \sum_t h_j^{l,t}$$

$$\begin{aligned} \boldsymbol{v}_j^{(l+1)} &= \frac{\sum_t h_j^{l,t}\boldsymbol{x}^{(t)}}{\sum_t h_j^{l,t}} \\ \Sigma_j^{(l+1)} &= \frac{\sum_t h_j^{l,t}(\boldsymbol{x}^{(t)}-\boldsymbol{v}_j^l)(\boldsymbol{x}^{(t)}-\boldsymbol{v}_j^l)^T}{\sum_t h_j^{l,t}} \end{aligned} \tag{33}$$

With linear models for regression, the M-step has an analytical solution. For the exceptions we have

$$\boldsymbol{u}_j^{(l+1)} = \frac{\sum_t h_j^{l,t}\boldsymbol{y}^{(t)}}{\sum_t h_j^{l,t}}, j = 1, \ldots, J \tag{34}$$

For the rule, if we have $P(\boldsymbol{y}|0,\boldsymbol{x}) \sim \mathcal{N}(\boldsymbol{\mu}_0, S_0)$ where $\boldsymbol{\mu}_0 = U_0\boldsymbol{x}$ (Jordan and Xu, 1995)

$$\begin{aligned} U_0^{(l+1)} &= \left[\sum_t h_0^{l,t} X_t (S_0^{(l)})^{-1} X_t^T\right]^{-1} \left[\sum_t h_0^{l,t} X_t (S_0^{(l)})^{-1} \boldsymbol{y}^{(t)}\right] \\ S_0^{(l+1)} &= \frac{\sum_t h_0^{l,t}(\boldsymbol{y}^{(t)} - {U_0^{(l+1)}}^T\boldsymbol{x})(\boldsymbol{y}^{(t)} - {U_0^{(l+1)}}^T\boldsymbol{x})^T}{\sum_t h_0^{l,t}} \end{aligned} \tag{35}$$

where

$$X_t^T = \left\{ \begin{array}{cccc|cccc} (x^t)^T & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & (x^t)^T & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots & \vdots & & & \vdots \\ 0 & \cdots & 0 & (x^t)^T & 0 & \cdots & 0 & 1 \end{array} \right\} \tag{36}$$

In the case of classification, M-step contains an iterative inner loop to maximize

$$\mathcal{Q}^e(\boldsymbol{u}) = \sum_t \sum_j h_j^{l,t} \sum_i y_i \log \mu_{ji} \tag{37}$$

Jordan and Jacobs (1994) propose to use the Iterative Reweighted Least Squares (IRLS). Using first order gradient ascent we get

$$\begin{aligned} \Delta\boldsymbol{u}_{0i} &= \eta\frac{\partial \mathcal{Q}^e}{\partial u_{0i}} = \eta\sum_t h_0^{l,t}(y_i^{(t)} - \mu_{0i}^{(t)})\boldsymbol{x}^{(t)} \\ \Delta u_{ji} &= \eta\sum_t h_j^{l,t}(y_i^{(t)} - \mu_{ji}^{(t)}) \end{aligned} \tag{38}$$

which is the batch version of Eq. (27).

Lam et al. (1997) investigate a hard version of the mixture of experts architecture where the largest h_j is set to one and all others to zero. Applied to our model this would correspond to choosing the rule *or* one of the exceptions for a given input and would be similar to the multistage method investigated in Section 4.

7 Hidden Markov REx

REx model as explained hitherto is restricted to independently identically distributed data. This restriction can be removed and the model generalized to get a REx appropriate for time series. In

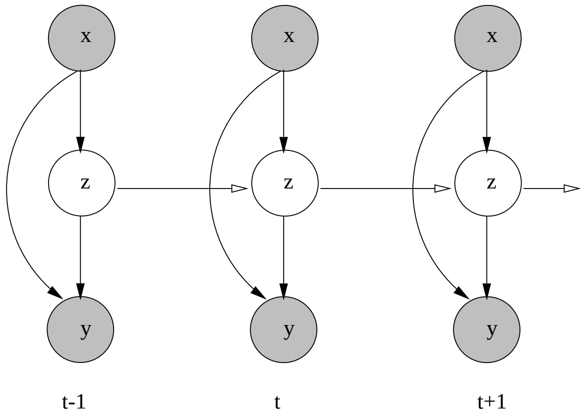

Figure 5: Each block is a REx mixture drawn as a graphical model where the observed variables (input and output) are shaded and the hidden state variable unshaded. The horizontal dimension is time and the arrows in that direction show Markov time dependencies.

the REx mixture model, depending on the input $\boldsymbol{x}$, a decision z is made and depending on input $\boldsymbol{x}$ and state z, output y is generated. The state variable z is a multinomial variable which when $z = 0$ corresponds to using the rule and $z = 1, \ldots, J$ correspond to the exceptions.

In extending REx to have a Markov temporal structure, we assume that the state at any time is dependent on the state in the previous time step (Fig. 5). So as we proceed through time, we may pass from the rule to an exception or vice versa or stay in the rule or jump from one exception to another one. This model is a special case of mixtures of experts through time and can be trained using EM (Bengio and Frasconi, 1996; Jordan et al., 1997). We have not, as yet, practical results on REx on sequence processing.

8 REx as an Arcing Method

Arcing (Breiman, 1997) is adaptively resampling and combining. REx is an arcing method and as such bears much resemblance to other arcing methods like bagging (Breiman, 1996) and boosting (Freund and Schapire, 1997). Bagging generates a number of datasets by bootstrapping from the given dataset, trains several classifiers with these datasets and combines them using majority voting.

There is a stronger relationship between REx and boosting. Boosting is a technique to get an accurate classifier by combining a large number of "weak" classifiers, these latter being classifiers which do a little better than random guessing. AdaBoost (Freund and Schapire, 1996) is one such method which works as follows. Let us say $D_t(i)$ is the probability of drawing sample $\boldsymbol{x}_i$ for training while constructing classifier t. $D_0(i) = 1/N, \forall i$. Classifier t makes a prediction $h_t(\boldsymbol{x}_i)$ and its error is defined as

$$\epsilon_t = \sum_{i : h_t(\boldsymbol{x}_i) \neq \boldsymbol{y}_i} D_t(i) \tag{39}$$

If error probability $\epsilon_t < 1/2$, the sample probabilities are then updated so as to decrease probabilities of samples correctly classified by the current classifier

$$D_{t+1}(i) = \frac{1}{Z} D_t(i) \times \begin{cases} \epsilon_t/(1-\epsilon_t) & \text{if } h_t(\boldsymbol{x}_i) = y_i \\ 1 & \text{otherwise} \end{cases} \tag{40}$$

where Z is a normalization constant that guarantees that $\sum_i D_{t+1}(i) = 1$. In bagging, classifiers do not affect each other's datasets and $D_t(i) = 1/N, \forall t$.

REx model can also be viewed in this framework except that in REx, going from the rule to the exception-learner, probabilities become 0/1. $D_X(i)$, the probability that $\boldsymbol{x}_i$ trains the exception-learner is 0 if the rule-learner is certain (which is a more stringent condition than it is correct) and is 1 otherwise.

$$D_X(i) = \frac{1}{Z} \times \begin{cases} 0 & \text{if } P_R(\boldsymbol{y}_i|\boldsymbol{x}_i) > \theta \\ 1 & \text{otherwise} \end{cases} \tag{41}$$

Instead of one rule and one exception learner, we can of course envisage a sequence of rule-learners with increasing values of confidence thresholds. Although up to now we have discussed a two-learner scheme composed of rule and exception learners, the approach can be generalized to the case where we have several cascaded rules and all previous discussion is valid in the case of multiple rule-learners.

Starting with $t = 1$, we train rule $P_t(\boldsymbol{y}|\boldsymbol{x}, \Phi_t)$ using sample $\mathcal{X}_t$ and determine the exceptions $\mathcal{X}_{t+1}$ by checking against the confidence threshold θ_t

$$\mathcal{X}_{t+1} = \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X}_t | P_t(\boldsymbol{y}|\boldsymbol{x}, \Phi_t) < \theta_t\}$$

The exceptions may be too many and we may have a reason to believe that there is a simpler way of explaining a large percentage of these exceptions by a rule. In this case we build $P_{t+1}(\boldsymbol{y}|\boldsymbol{x}, \Phi_{t+1})$ using $\mathcal{X}_{t+1}$. We then set a new threshold $1 > \theta_{t+1} \geq \theta_t$ and find the next level of exceptions. We continue like this until we get to a point where we have few exceptions and storing them as a table is not more expensive than trying to find a rule that explains them. By storing them as they are, we get as close as we can get to have their $P(\boldsymbol{y}|\boldsymbol{x})$ (and the final confidence threshold) one.

From a boosting perspective, going from rule t to rule $t+1$, probabilities change as follows

$$D_{t+1}(i) = \frac{1}{Z} \times \begin{cases} 0 & \text{if } P_t(\boldsymbol{y}_i|\boldsymbol{x}_i) > \theta_{t+1} \\ 1 & \text{otherwise} \end{cases} \tag{42}$$

with $1 > \theta_{t+1} \geq \theta_t$. One can write a "soft" version of REx where the probabilities are degraded continuously from one towards zero as a function of success by previous classifiers. Then for example we can define error as a sum of errors (one minus the actual probability) weighted by the probabilities of drawing the samples

$$\epsilon_t = \sum_{i:P_t(\boldsymbol{y}_i|\boldsymbol{x}_i)<\theta_t} D_t(i)[1 - P_t(\boldsymbol{y}_i|\boldsymbol{x}_i)] \tag{43}$$

and use Eq. (40) or a variant thereof to compute $D_{t+1}(i)$.

Schapire et al. (1997) write that boosting increases the "margin" which they define as the difference between the prediction for the correct class and the prediction for the highest class different from the correct class. They explain margin intuitively as the "confidence" of the classifier. Note that this is exactly the confidence threshold θ. Let us say $\boldsymbol{x} \in \omega_k$ and that we compute $P(\omega_k|\boldsymbol{x})$. Then

$$\text{margin} = P(\omega_k|\boldsymbol{x}) - \max_{l \neq k} P(\omega_l|\boldsymbol{x})$$

Given that $\max_{l \neq k} P(\omega_l|\boldsymbol{x}) \leq 1 - P(\omega_k|\boldsymbol{x})$, we have margin $\geq 2P(\omega_k|\boldsymbol{x}) - 1$. If $P(\omega_k|\boldsymbol{x}) > \theta$ is also satisfied, we get

$$\text{margin} \geq 2P(\omega_k|\boldsymbol{x}) - 1 > 2\theta - 1 \tag{44}$$

In boosting one requires that the prediction is correct; in REx, we not only require that it is correct but we also require the classifier to be sufficiently confident in its decision. For example in the case of classification, not only that the input should be on the right side of the decision boundary but also within a large distance to it. When we have a sequence of rule learners with increasing confidence thresholds θ_t, what we do is that we effectively require this distance to increase.

Another difference between REx and bagging and boosting is that in REx the way learners are combined is trained. In bagging this is done by averaging and in boosting it is a weighted average where higher weights are given to more accurate learners, i.e., the weight of learner t is $\log((1-\epsilon_t)/\epsilon_t)$.

9 Simulation Results

9.1 Datasets

We test variants of REx on several datasets (Table 1). Two of them are our own datasets and are accessible through the web[1]. `digit` is on optical handwritten digit recognition. It was created using the set of programs made available by NIST (Garris et al., 1994). The 32 by 32 normalized bitmaps were low-pass filtered and undersampled to get 8 by 8 matrices where each element is an integer in the range 0 to 16. 44 people filled in forms which were randomly divided into two clusters of 30 and 14 forms. From the first 30, three sets were generated: A training set, a validation set on which to tune hyperparameters and a writer-dependent set. The other 14 forms containing examples from distinct writers make up the writer-independent (WI) test set which we use as the real test set.

`pen` is a dataset on pen-based handwritten digit recognition. Similar to `digit`, people have written down samples on a touch-sensitive tablet which were then resampled and normalized to a temporal sequence of eight pairs of (x, y) coordinates. We again divided the samples into four parts of training, validation, writer-dependent test, and writer-independent test sets.

`vowel` and `thyroid` are standard datasets from the UCI repository with their training and test sets separated.

On all datasets, we use 2-fold cross-validation to determine the exceptions. Results we report are averages over ten independent runs starting from random initial weights. Unless otherwise stated, we use linear models as rule-learner in REx and to bag and boost with.

Table 1: Characteristics of the datasets used.

Name	No of Inputs	No of Classes	Training Examples	Test Examples
`digit`	64	10	1,934	1,797
`pen`	16	10	3,748	3,498
`vowel`	11	6	528	462
`thyroid`	21	3	3,772	3,428

9.2 Results with REx-Multistage

We first check the behaviour as a function of the confidence threshold θ. On all datasets, as expected, the number of exceptions during training increase when θ is increased. On `digit`, even for θ as large as 0.99, the exceptions are only 7% showing that the linear model that we use as the rule does find a good underlying rule explaining the majority of the cases with sufficient confidence (Fig 6-a). Note that because we use data other than data with which we trained the rule, we can use such high θ values.

We also see that this small extra set of stored patterns increases the accuracy on the test set (Fig 6-c). During testing the slow and cumbersome exception-learner is rarely consulted (Fig 6-b) and even when it is, response is faster because the table is much smaller (Fig 6-d). For example on

[1] At `ftp://www.cmpe.boun.edu.tr/people/ethem/{digit,pen}`

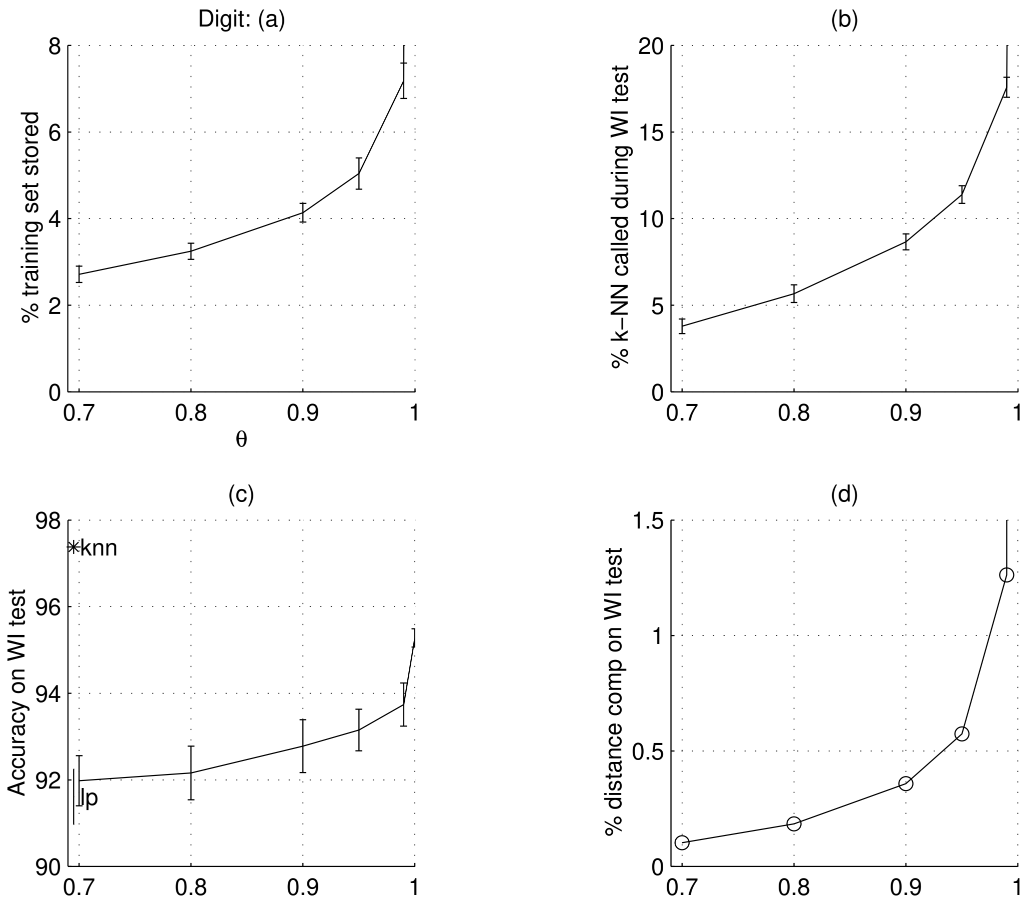

Figure 6: On the **digit** dataset, results by combining a linear rule and k-NN exception-learner for different values of θ with $\theta \in \{0.7, 0.8, 0.9, 0.95, 0.99, 1.00\}$. (a) % training patterns stored, (b) % exception-learner called during test (c) % accuracy on the writer-independent test set with k-NN as exception-learner (accuracy with linear model and k-NN alone are also given for comparison) and (d) % of distance computations made (d=a*c). When $\theta = 1$, all patterns are stored as exceptions and k-NN is always consulted (100% in (a), (b) and (d); this corresponds to taking a simple vote over a linear model and k-nn proper.

Table 2: Comparison of multistage 'lp+knn' type REx with linear model and k-nn proper on the datasets used. Accuracy values are averages and standard deviations of ten independent runs. We also give average number of exceptions stored and percentage of times exception-learner is consulted. θ is chosen that best trades off accuracy with the cost of storing and using exceptions. The differences between the accuracies of 'lp' and 'lp+knn' are significant, i.e., larger than two stdevs.

	Accuracies				
Dataset	lp	lp+knn	k-nn	% stored	% used
`digit`, $\theta = 0.99$	91.61 0.64	93.74, 0.50	97.38	139/3,774	7.6
`pen`, $\theta = 0.8$	91.12, 0.17	94.57, 0.55	97.28	379/3,748	11.6
`vowel`, $\theta = 0.7$	40.22, 1.61	47.53, 3.39	59.31	451/528	37.0
`thyroid`, $\theta = 0.8$	95.78, 0.05	96.66, 0.14	94.40	211/3,772	4.8

`digit`, k-nn proper requires 1,934 distance computations for each test character and we have 1,797 WI test characters. With cascading when $\theta = 0.99$, the exception table stores 7% and only 18% of the test set uses the exception-learner k-NN thus we need $0.18 * 0.07 = 1.3\%$ distance computations of the former (Fig. 6-d).

On `pen` (Fig. 7), similarly, with increasing θ, we store more exceptions and consult them more frequently and get higher accuracy. Percentages for the exception-learner is higher implying that a linear rule-learner is not as ideal for `pen` as it is for `digit`. Still with $\theta = 0.99$, though we store 40% of the exceptions, we get as much average accuracy as we get with $\theta = 1.0$ when we store all patterns as exceptions.

An interesting case is that of `thyroid` where the linear rule-learner is more accurate than the k-nearest neighbor classifier, probably due to noise (Fig. 8-c). But here as well with $\theta = 0.8$, we store 6% of the data and get higher accuracy than both.

`vowel` is a dataset where a linear rule is not appropriate; there is not enough data to infer the existence of a general linear rule and thus a large percentage of patterns is stored as exceptions (Fig. 9). Here it is better to use 10-fold cross-validation than 2-fold. With 2-fold, we have only half of the data to train and this does not lead to a confident model which stores a large percentage (Fig. 9-a); many almost duplicates are also probably stored when the two halves are swapped (Section 3 mentions a strategy to avoid unnecessary exceptions). When the rule is trained on the whole dataset (Fig. 9-b), it is more confident for small θ.

On `vowel`, we see the effect of the complexity of the rule-learner on accuracy where we also use multilayer perceptrons with 25 and 50 hidden units as rule-learners (Fig. 10); we have not tested this with the other larger training sets due to time limitations. A more accurate rule-learner leads to a more accurate final REx model, though this does not decrease the percentage of stored exceptions on this small dataset. A multilayer perceptron may not be ideal as a rule-learner as it does not allow knowledge extraction; a decision tree may be preferred in such a case. An alternative is to look for a sequence of nested *simpler*, but possibly more confident rules. When rules check for the value of one feature, this is a decision tree.

We compare REx-multistage, 'lp+knn', with the linear model and k-nn on all datasets (Table 2). There is a statistically significant increase in accuracy. The price we pay is that we store a small percentage of exceptions required during a small percentage of times. In `vowel`, because of the small dataset, the rule does not converge to a solution with high confidence and therefore rejects frequently.

9.3 Results with REx-Multiexpert

We compare different ways of combining the rule and exception-learners on the datasets in terms of accuracy on the test set and the number of bits required to store paramaters. Except `digit`, we assume we require 8 bits to store original training patterns used in k-nn and 'lp+knn', 12 bits to

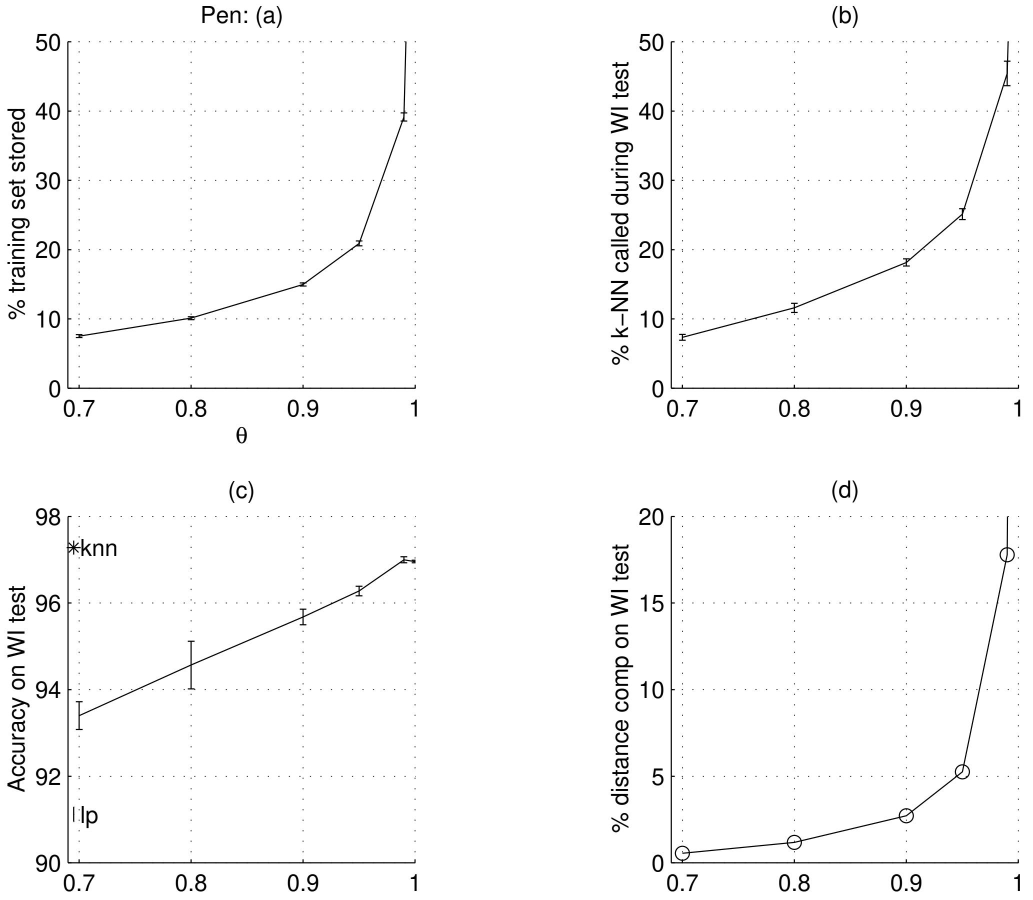

Figure 7: On the **pen** dataset, results by combining a linear rule and k-NN exception-learner for different values of θ with $\theta \in \{0.7, 0.8, 0.9, 0.95, 0.99, 1.00\}$. (a) % training patterns stored, (b) % exception-learner called during test (c) % accuracy on the writer-independent test set with k-NN as exception-learner (accuracy with linear model and k-NN alone are also given for comparison) and (d) % of distance computations made (d=a*c).

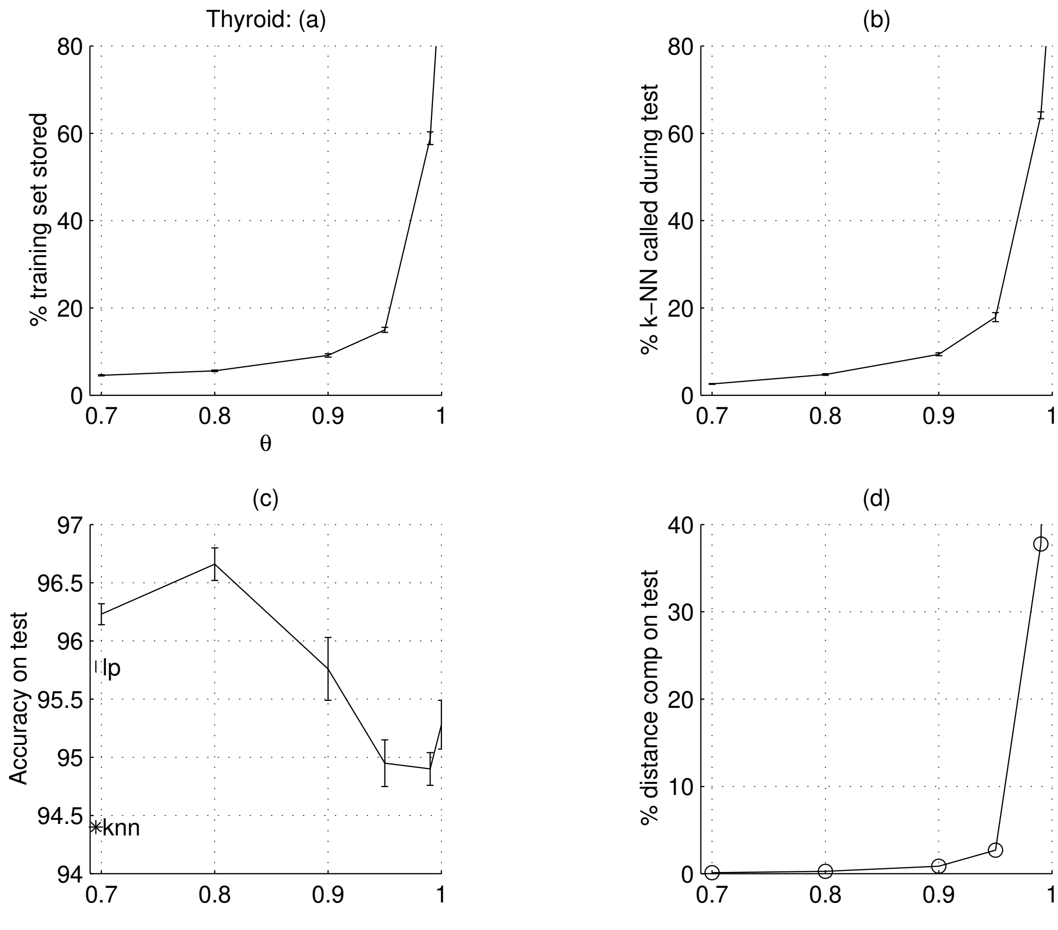

Figure 8: On the **thyroid** dataset, results by combining a linear rule and k-NN exception-learner for different values of θ with $\theta \in \{0.7, 0.8, 0.9, 0.95, 0.99, 1.00\}$. There probably is a lot of noise in this data. Linear model is more accurate than k-nn but still, adding some extra patterns improves accuracy.

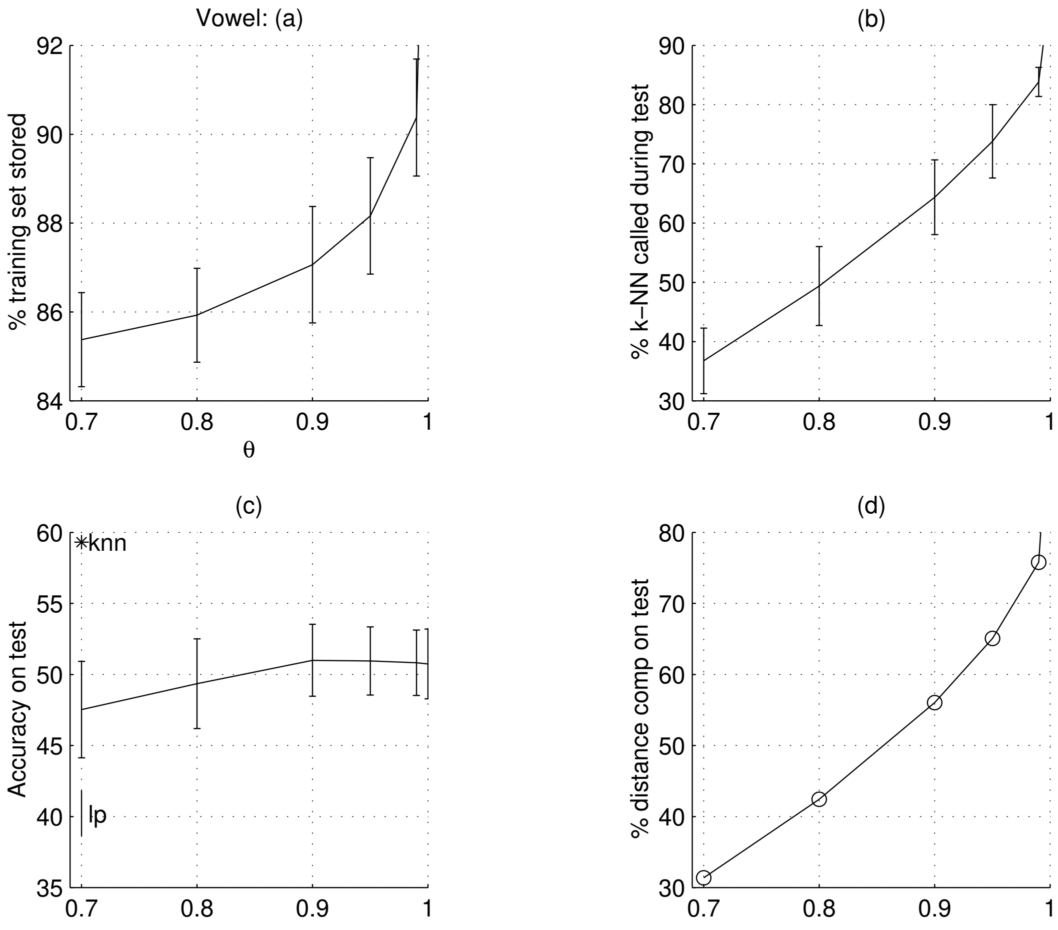

Figure 9: On the `vowel` dataset, results by combining a linear rule and k-NN exception-learner for different values of θ with $\theta \in \{0.7, 0.8, 0.9, 0.95, 0.99, 1.00\}$. There is not enough data to support the hypothesis that the rule is linear. Accuracy is improved but there is not much gain from memory and computation.

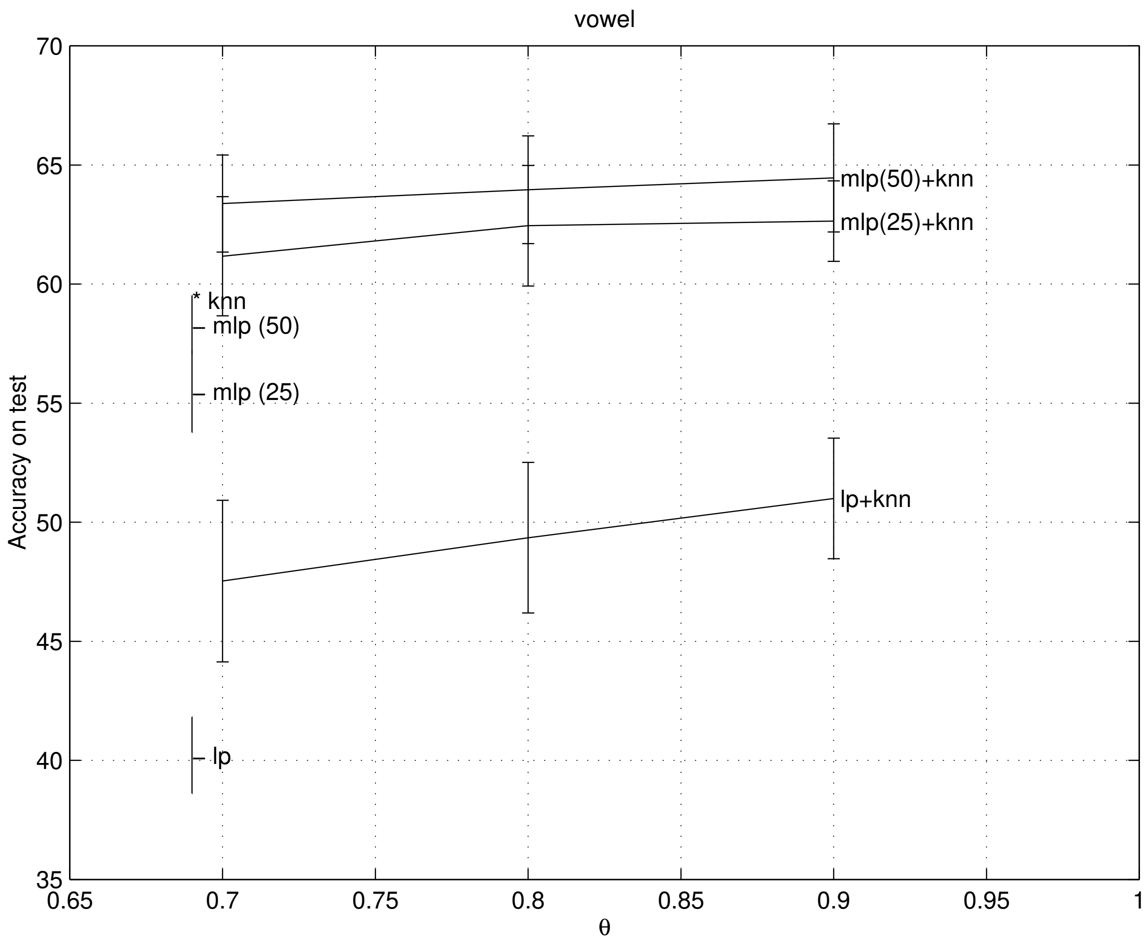

Figure 10: On the `vowel` dataset, we test the effect of the complexity of the rule-learner on accuracy. We combine a linear rule and multilayer perceptrons with 25 and 50 hidden units with the k-NN exception-learner for different values of θ with $\theta \in \{0.7, 0.8, 0.9\}$. A more accurate rule-learner leads to a more accurate final REx.

store modifiable exception positions used in REx-rbf and REx-mix and 24 bits to require a weight of the linear model. In `digit`, these are 4, 18, and 16 bits respectively. The absolute values are not important but the relative magnitudes are. Exception positions are less precisely stored than weights of the linear model because they are finetuned only within a small region. In fact, we expect multiexpert REx methods to be accurate even if the exception positions are not finetuned.

In Fig. 11, on `pen`, we compare the three variants of REx with multilayer perceptrons of different number of hidden units in terms of accuracy on the writer independent test set and the number of bits required to store parameters. We notice that the accuracy of REx multistage 'lp+knn' depends on the number of stored exceptions but with multiexpert method with gaussian exceptions ('lp+rbf') or the cooperative mixture variant ('mix'), because we modify the effects of exceptions (and also their positions and spreads), with a small number of exceptions, we can get quite large accuracy. When $\theta = 0.7$ with 238 exceptions on the average (6%), we get larger accuracy than the multilayer perceptron and store less bits. Accuracy of k-nn, storing the whole dataset, is less than 0.5% higher but requires 20 times more bits. We bag/boost on 5, 10, and 20 linear models with less accuracy, similar memory use and more training time. Combining more, one may get higher accuracy but this may be beyond the memory and computational capacity at hand.

Training takes around 20 epochs on all REx variants, lp and mlp. Training k-nn takes one epoch. The competitive mixture trains faster but is not as accurate as the cooperative mixture. Because we use 2-fold cross-validation, training a REx model takes three times as much. Bagging and boosting on five models take five times as much.

Comparison on `digit` is given in Fig. 12. We see that the mixture variant uses as much space as boosting and is as accurate though requiring less training epochs. In this case, a multilayer perceptron with 30 hidden units has accuracy of 95% and one may use it as the rule-learner or bag/boost using it, if the principal aim is to get high accuracy.

On `thyroid`, we see that even the basic multistage REx, 'lp+knn', is more accurate, uses less memory and trains faster than bagging and boosting.

On `vowel` (Fig. 14), because of the small training set, we cannot infer the existence of an underlying linear rule with high confidence and thus reject many samples. We store a large percentage of exceptions with a linear model and do not gain much in terms of memory when compared with k-nn; we do gain from computation when the rule model is certain. Still, we see that all REx variants are more accurate than bagging/boosting using comparable amount of memory; REx is also faster to train in that REx with 2-fold cross-validation takes 3*25 epochs where boosting with 5 models takes 5*25 epochs to train.

The accuracy of bagging and boosting can be increased by combining a larger number of models. Breiman (1997) bags and boosts over 50 decision trees and Freund and Schapire (1996) on 100 models. When memory, computation and time necessary for training and combining such large numbers of models is not available, REx stands out as a successful alternative.

10 Conclusions

We advocate the idea of explaining a dataset as a simple general rule and a set of exceptions not covered by the rule with sufficiently high confidence. We investigate the multistage and multiexpert methods for combining the rule and exceptions and detail strategies for training the model from labelled training data. We also show that the model can be extended to learn time series.

Our experiments show that the model has high accuracy and low memory and computation need. One other advantage it has over neural network models like the multilayer perceptrons is that it allows incorporating prior knowledge. One can for example think of a scenario where the rule is predefined by the user, or is not trained but is programmed a priori. Then REx allows determination of the exceptions and combining the rule and exceptions. Also in cases where knowledge extraction in converting labelled data to high level rules is important, explaining the dataset as a simple e.g.

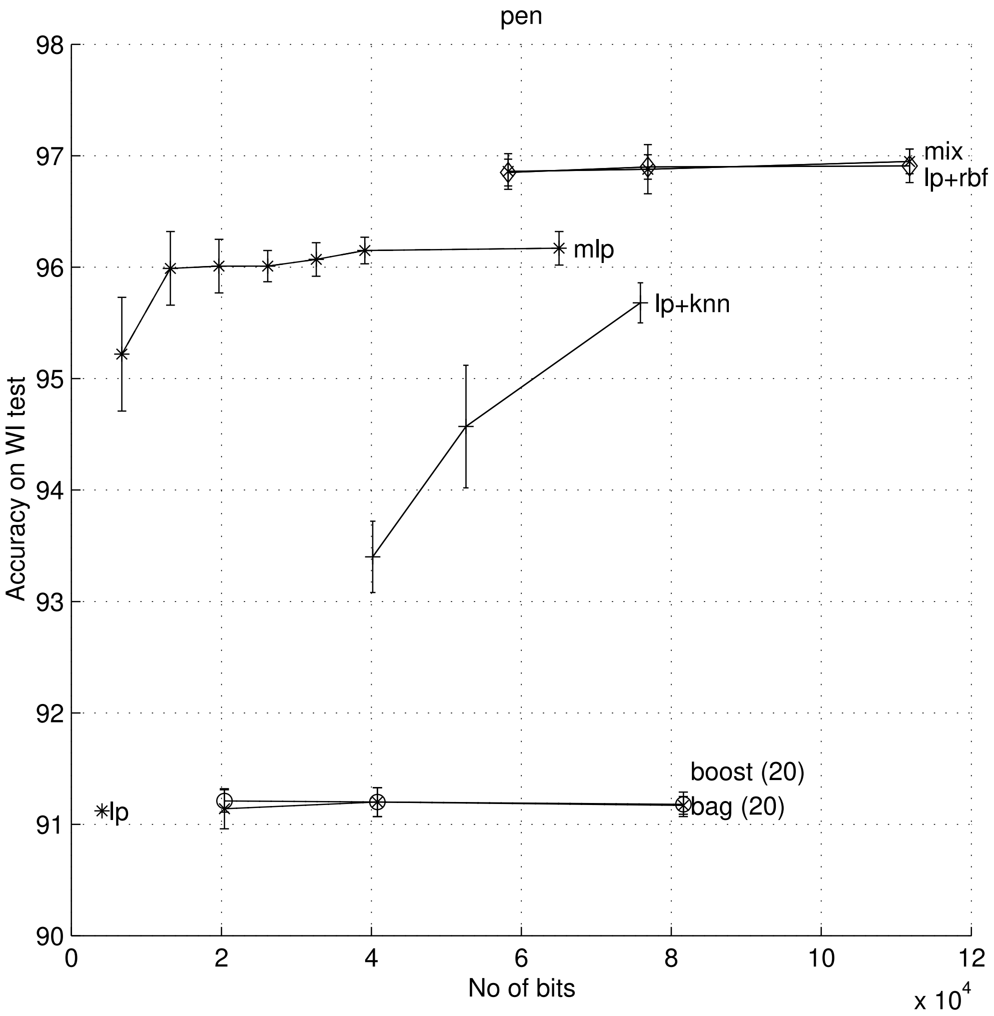

Figure 11: On the pen dataset, comparison of REx variants as a function of accuracy vs memory required with $\theta \in \{0.7, 0.8, 0.9\}$. 'mlp' is a multilayer perceptron with the number of hidden units ranging from 10 to 60 with increments of 10 and with 100 hidden units. One standard deviation error bars are also given. k-nn requires $8 * 10^5$ bits and is not shown; its accuracy is 97.28. We also report results with bagging and boosting on 5, 10, and 20 linear models.

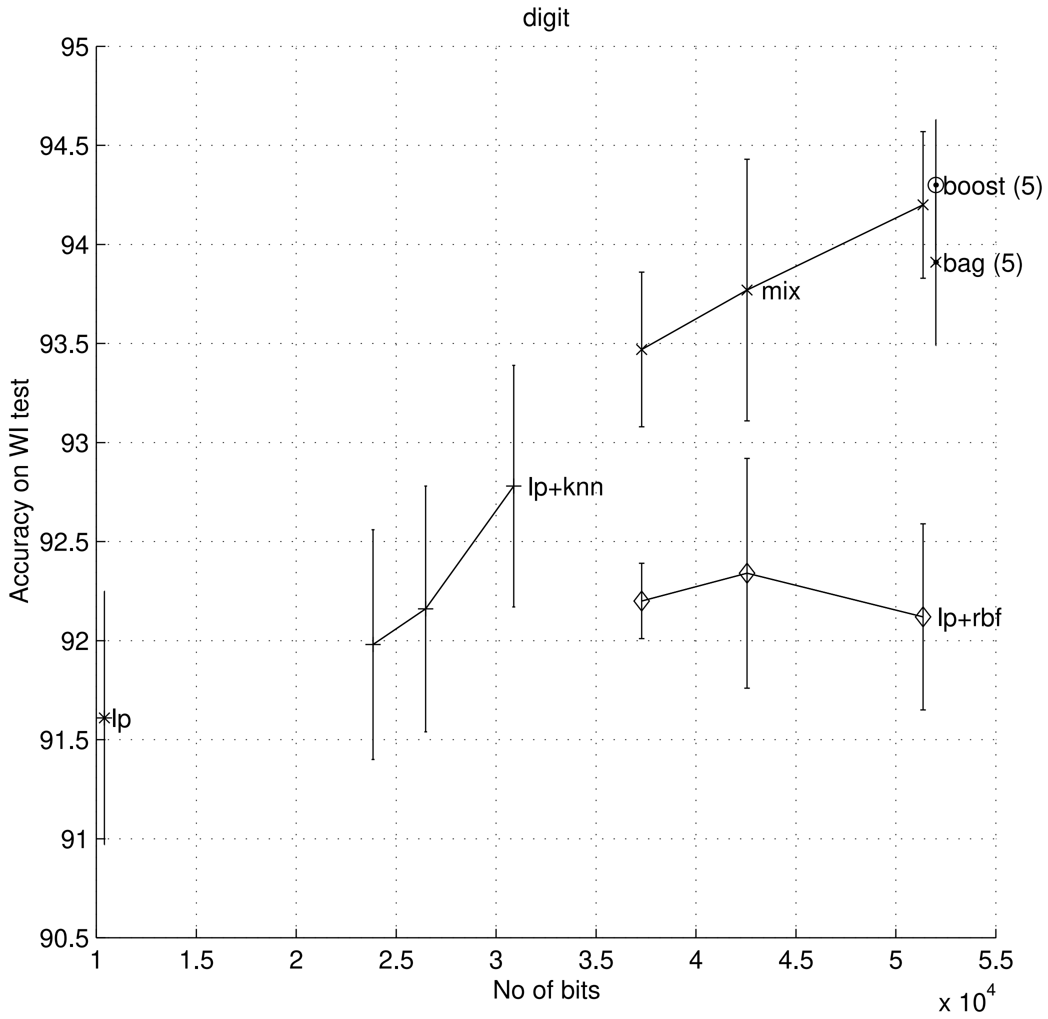

Figure 12: On the `digit` dataset, comparison of REx variants as a function of accuracy vs memory required with $\theta \in \{0.7, 0.8, 0.9\}$. We also report results with bagging and boosting on five linear models. k-nn requires $1 * 10^6$ bits and is not shown; its accuracy is 97.38. Training the basic linear model takes 11 epochs. The mixture variant is as accurate as boosting, uses as much memory but is faster to train. Bagging/boosting ten models, accuracy increases to 94.5% but this uses twice as much memory; the same effect can be gotton with a larger θ.

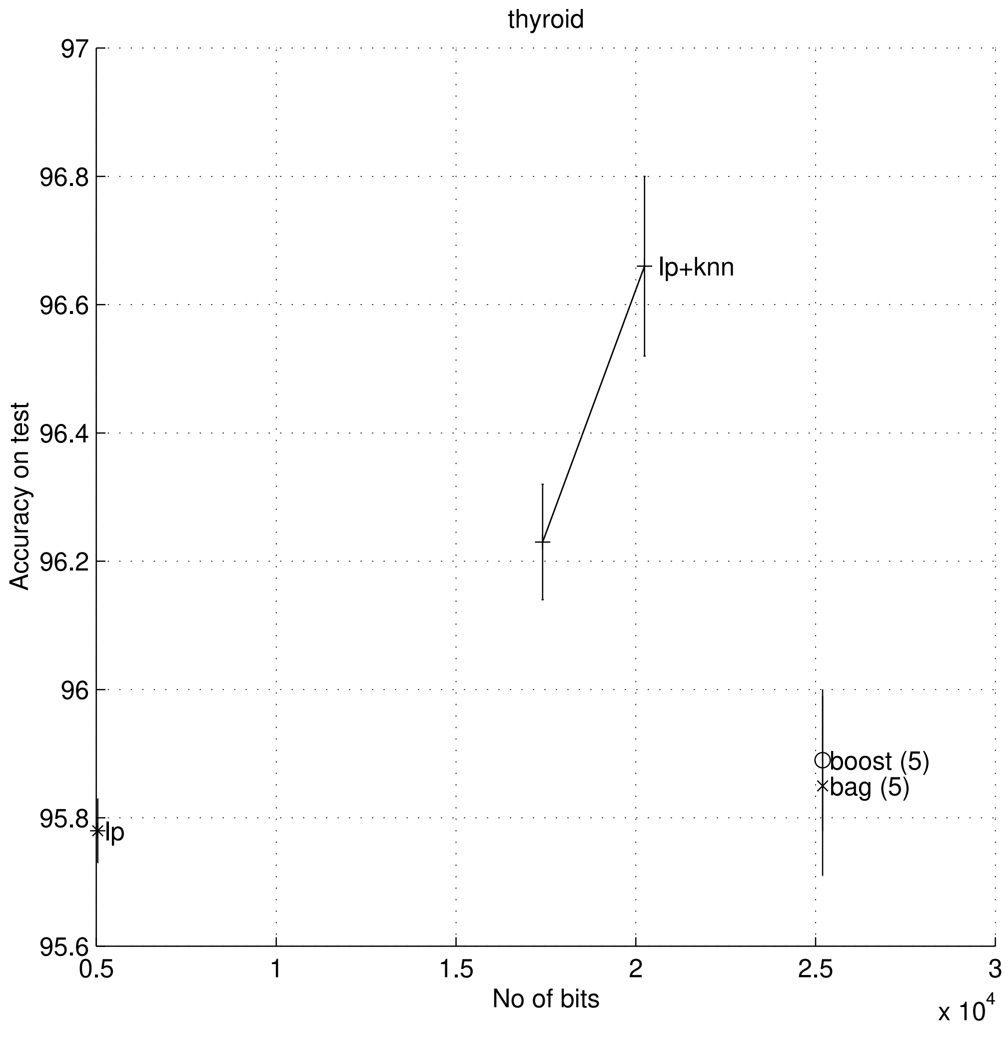

Figure 13: On the **thyroid** dataset, comparison of ‘lp+knn’ with $\theta \in \{0.7, 0.8\}$ and bagging and boosting on five linear models. k-nn requires $4 * 10^5$ bits and is not shown; its accuracy is 94.40. ‘lp+knn’ is more accurate, uses less memory and is faster to train than bagging and boosting. Bagging/boosting ten models does not increase accuracy significantly. Other REx variants do not increase accuracy significantly and are costlier.

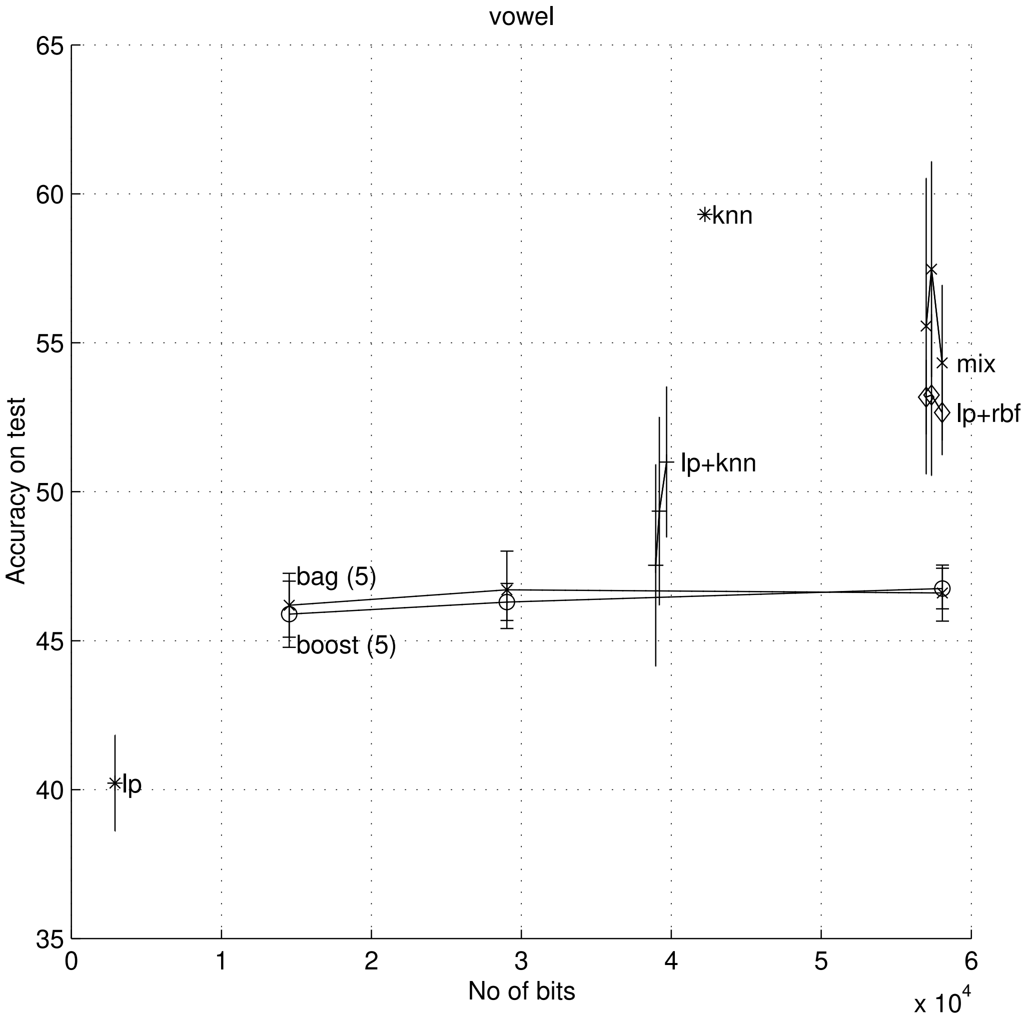

Figure 14: On the **vowel** dataset, comparison of REx variants with $\theta \in \{0.7, 0.8, 0.9\}$ and bagging and boosting on 5, 10 and 20 linear models. The linear model is not certain and we do not gain from memory; the dataset is too small to infer the existence of a decision rule (note the large stdevs). We see that all REx variants are more accurate than bagging/boosting using comparable amount of memory. REx is also faster to train.

linear rule and a set of exceptions is useful. Where a linear rule may not be sufficient, one can have a more complex rule like a decision tree, or construct rules in a cascaded manner one after the other, each explaining exceptions of the previous one.

Other arcing methods like bagging and boosting, for high accuracy, require training and combining large numbers of learners. When memory and computational resources are limited -as they usually are-, REx is an arcing method that can be used with high accuracy and low need of memory and computation.

Acknowledgements

The author is a Fulbright Scholar. Earlier part of this work was supported by Grant EEEAG-143 from Turkish Scientific and Technical Research Council (Tübitak). The form processing routines were made available by NIST. `digit` database was created by Cenk Kaynak and `pen` by Fevzi Alimoğlu.

References

[1] Alpaydın, E., & Jordan, M. I. (1996). Local Linear Perceptrons for Classification. *IEEE Transactions on Neural Networks, 7*, 788-792.

[2] Alpaydın, E., & Kaynak, C. (1998). Cascaded Classifiers. *Kybernetika*, to appear.

[3] Bengio, Y., & Frasconi, P. (1996). Input-Output HMM's for Sequence Processing. *IEEE Transactions on Neural Networks, 7*, 1231–1249.

[4] Breiman, L. (1996). Bagging Predictors. *Machine Learning, 26*, 123–140.

[5] Breiman, L. (1997) Arcing Classifiers. TR-460, Statistics Department, University of California at Berkeley.

[6] Freund, Y., & Schapire, R. E. (1996). Experiments with a New Boosting Algorithm. *Proceedings of the Thirteenth International Conference on Machine Learning*, 148–156.

[7] Garris, M. D., Blue, J. L., Candela, G. T., Dimmick, D. L., Geist, J., Grother, P. J., Janet, S. A., & Wilson, C. L. (1994). NIST Form-Based Handprint Recognition System. NISTIR 5469.

[8] Holte, R. C. (1993). Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. *Machine Learning, 11*, 63–91.

[9] Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive Mixtures of Local Experts. *Neural Computation, 3*, 79–87.

[10] Jordan, M. I., & Jacobs, R. A. (1994). Hierarchical Mixtures of Experts and the EM Algorithm. *Neural Computation, 6*, 181–214.

[11] Jordan, M. I., & Xu, L. (1995). Convergence Results for the EM Approach to Mixtures of Experts Architecture. *Neural Networks, 8*, 1409–1431.

[12] Jordan, M. I., Ghahramani, Z., & Saul, L. K. (1997). Hidden Markov Decision Trees. In M. C. Mozer, M. I. Jordan, T. Petsche (Eds.), *Advances in Neural Information Processing Systems 9*, Cambridge, MA: MIT Press.

[13] Lam, W., Yung, F., & Xu, L. (1997). An Experimental Comparative Study on Several Soft and Hard-cut EM Algorithms for Mixture of Experts. *International Conference on Neural Networks*, Houston Texas, June.

[14] McCullagh, P., & Nelder, J. A. (1989). *Generalized Linear Models*, 2nd Edition, Chapman & Hall.

[15] Pudil, P., Novovicova, J., Blaha, S., & Kittler, J. (1992). Multistage Pattern Recognition with Reject Option. *11th IAPR International Conference on Pattern Recognition B, II*, 92–95.

[16] Schapire, R. E., Freund, Y., Bartlett, P., & Lee, W. S. (1997). Boosting the Margin: A new explanation for the effectiveness of voting methods.

[17] Wolpert, D. H. (1992). Stacked Generalization. *Neural Networks, 5*, 241–259.

[18] Xu, L., Krzyżak, A., & Suen, C. Y. (1992). Methods of Combining Multiple Classifiers and Their Applications to Handwriting Recognition. *IEEE Transactions on Systems, Man, and Cybernetics, 22*, 418–435.

[19] Xu, L., Jordan, M. I., & Hinton, G. E. (1994). A Modified Gating Network for the Mixture of Experts Architecture. *World Congress on Neural Networks II*, San Diego CA, June, 405–410.