LEARNING RULES AND EXCEPTIONS FOR REGRESSION: THE REX ALGORITHM

by

Zafer Barutçuoğlu

B.A., in Mathematics, Boğaziçi University, 2000

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

Boğaziçi University

2002

LEARNING RULES AND EXCEPTIONS FOR REGRESSION: THE REX ALGORITHM

APPROVED BY:

Assoc. Prof. Dr. Ethem Alpaydın .....
(Thesis Supervisor)

Assoc. Prof. Dr. Lale Akarun .....

Prof. Dr. Günhan Dündar .....

DATE OF APPROVAL: 13.06.2002

ACKNOWLEDGEMENTS

First of all, I am grateful to my wife Itır who patiently endured the countless days and nights of my thesis work and cheered me up when all the numbers seemed wrong, as I worked away a year of our youth. She has put into this thesis more than she is aware of. Not to mention the delicious late night snacks, ice cream and coffee.

I am indebted beyond words to my thesis supervisor Ethem Alpaydın for giving me this opportunity and his precious time. I shall forever try to be worthy of his confidence in me. Under his guidance this research has been a thoroughly enjoyable and inspirational experience.

I would like to thank Lale Akarun and Günhan Dündar for participating in my thesis jury and their careful reviews of the manuscript.

A. Safa Topbaş and the fellow developers at Turk Nokta Net all deserve my gratitude for their hearty support.

ABSTRACT

LEARNING RULES AND EXCEPTIONS FOR REGRESSION: THE REX ALGORITHM

The human mind models many concepts as a general rule and a few specific exceptions. The rule is simple and covers most cases, and the exceptions allow learning obscure examples while still keeping the rule simple and useful. The algorithm *REx* (*R*ules and *Ex*ceptions) applies the same paradigm to machine learning to produce an accurate and interpretable learning model. Previously explored for classification with success, REx is adapted in this thesis to regression problems. Using another simpler algorithm as a base rule, it determines a set of exceptions in the training data, and augments the rule by nondestructively incorporating the exceptions as local experts. Both collaborative and mixture combination schemes are explored, with a possible improvement through finding clusters of exceptions. Also included are detailed examinations of Bagging, AdaBoost variants and Support Vector Machines for regression, because of their relation to REx in emphasizing some examples more than others. Simulations on several datasets provide empirical support for the discussion comparing all algorithms. The results indicate that while the mixture version of REx suffers from certain structural drawbacks that hinder consistent learning, the collaborative version achieves satisfactory performance, especially with simple rules.

ÖZET

REGRESYON İÇİN KURAL VE İSTİSNALARIN ÖĞRENİLMESİ: REX ALGORİTMASI

İnsan zihni birçok kavramı genel bir kural ve birkaç istisna olarak şekillendirir. Kural basit olup çoğu durumda geçerliyken, istisnalar kuralın basitliğini ve kullanışlılığını bozmadan sıradışı örneklerin de öğrenilebilmesini sağlar. *REx* algoritması başarılı ve anlaşılabilir bir öğrenme modeli oluşturmak için aynı prensibi yapay öğrenmeye uygular. Daha önce sınıflandırma için incelenmiş ve başarıya ulaşmış olan REx bu tezde regresyon problemlerine, yani çıktısı sürekli problemlere uyarlanmaktadır. Daha basit başka bir algoritmayı temel kural olarak kullanarak öğrenme verilerinden bir dizi istisna belirlenir, ve istisnaları yerel uzmanlar olarak ekleyerek kural bozulmadan genişletilir. Tezde hem işbirlikçi hem de karışım birleştirme senaryoları incelenmekte, ve yoğun istisna gruplarını bulmaya dayanan bir ek anlatılmaktadır. Aynı zamanda, bazı örnekleri vurgulamak yönünden REx ile ilintili olduklarından, regresyon amaçlı Bagging, AdaBoost çeşitleri ve Destek Vektörü Makinaları da detaylı olarak incelenmektedir. Tüm algoritmaları karşılaştıran tartışmalar çeşitli veri kümeleri üzerinde yapılan benzetimlerden gelen deneysel değerlerle desteklenmektedir. Alınan sonuçlara göre, REx'in karışım türü tutarlı öğrenmeyi zorlaştıran bazı yapısal sorunlar içerirken, işbirlikçi hali özellikle basit kurallar kullanıldığında tatmin edici başarıya sahiptir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

c_i	Combination coefficient for model i
d	Input attribute index
D	Number of input attributes
E	Error function
$F(\cdot)$	Rule model
g_j^t	Softmax output of Gaussian unit j for example t
h_k^t	Output of sigmoidal hidden unit k for example t
i	Model index
j	Gaussian unit index
J	Number of leaf nodes; number of exceptions
k	Hidden unit index
K	Number of hidden units; number of clusters
L	Loss function
N	Number of training examples
p_j^t	Output of Gaussian unit j for example t
r^t	Training label of example t
T_k	Weight of hidden unit k
v_j	Weight of Gaussian unit j
$\mathbf{w}_k$	Input weight vector of hidden unit k
w_{kd}	Weight of input attribute d on hidden unit k
$\mathbf{x}^t$	Input vector of example t
x_d^t	Input attribute d of example t
$\mathcal{X}$	Training set
y^t	Model output for example t
ε	Error threshold
η	Learning rate
μ_j	Mean vector of Gaussian unit j
μ_{jd}	Mean vector attribute d of Gaussian unit j

σ_j	Standard deviation of Gaussian unit j
ln	Logarithm to base e
MLP	Multi-Layer Perception
SVM	Support Vector Machine

1. INTRODUCTION

The task of supervised machine learning consists of approximating an unknown target function for which the outputs are known only for certain inputs. Every learning algorithm assumes a *model* for the candidate functions among which the best approximation to the target function will be sought. The model is defined by a set of model parameters, which are to be determined using a set of known input-output pairs (called the *training* set). This is inherently an ill-posed problem, since the training set by itself does not specify the target function completely. There may be infinitely many possible candidate functions that all comply with the given data but differ elsewhere. Different learning algorithms make different assumptions about the unseen data and the target function, and arrive at different solutions. The final predictive accuracy of a model depends on how well the algorithm's assumptions hold for the given data. Since there is no simple "silver bullet" algorithm, the machine learning practitioner needs to maintain a toolbox of existing algorithms, with an understanding of which to apply to what kind of data.

Learning problems are often categorized into two types, as *classification* and *regression*. Classification is when the output range of the target function is a finite set of values, since the function effectively places the given input into one of a number of classes. Otherwise, if the output is from a (possibly infinite) range of continuous values, the learning process is called regression, or function approximation.

The algorithm *REx* (**R**ules and **Ex**ceptions) was originally proposed in [1], and analyzed for classification tasks in [2, 3, 4, 5], with significant theoretical and empirical results. In this work, we extend the algorithm to regression problems, and compare it to Bagging, AdaBoost variants and Support Vector Machines.

REx works by selecting some of the training examples as exceptions. This approach of isolating "difficult" examples is reminiscent of the well-known AdaBoost and Support Vector Machine methods. REx has a rule algorithm given to it, and it uses the

rule to first determine the exceptions. Then placing Gaussian units centered at each exception, it produces a combined network model of the rule model and the exception units. Finally, the whole combined model is trained together, so that the rule is aware of the exceptions and does not try to fit them. The Gaussian means and variances are also allowed to change, adjusting to the rule in turn. If there are too many exceptions, they can be reduced by clustering. Two different versions of REx are proposed. Collaborative REx uses a linear combination of the rule and the exception outputs, while Mixture REx gives a Gaussian unit also to the rule and uses the softmax function on the outputs to make a single exception or the rule significantly more active than others at a given time.

This thesis is organized as follows. Section 2 describes AdaBoost and its precursor Bagging. Various AdaBoost alternatives for regression are explored, in addition to several variations proposed on the Bagging theme.

Section 3 is devoted to Support Vector Machines. The topic is motivated beginning with the linear case, and then extended by introducing the use of nonlinear kernels. A variant with automatic insensitivity determination is also described.

Section 4 describes the novel algorithm REx in detail. The Collaborative REx and Mixture REx versions are derived, with the complete set of gradient descent update equations included for both a linear rule and a Multi-Layer Perceptron rule for each version. The section concludes with an improvement for eliminating redundant exceptions by clustering.

Section 5 is about the simulations, where we describe our experiments and present the results on various datasets. The algorithm families are compared with each other and among themselves. Exemplar outputs on a synthetic dataset are plotted to illustrate the actual behaviors of the algorithms.

Section 6 draws conclusions from the results and points in possible directions for future work.

2. BAGGING AND ADABOOST

While some algorithms always produce the same solution for the same training data, others include a random element. For example, a common approach is to start with a randomly initialized model, and iteratively update its parameters until the model gives the correct outputs to the training examples. The randomness introduces a *variance* to the algorithm; multiple runs of the same algorithm on the same training set may converge to different solutions.

Another issue is stability with respect to small changes in training data. Ideally, a training set should perfectly illustrate the behavior of the target function over the whole input space. Hence, two training sets for the same problem should produce identical solutions. However, in practice training data is finite, and often contains *noise* resulting in incorrect attribute values. The existence or nonexistence of even one particular example in a finite training set may be significant enough to produce different model instances at the end of training.

Combining learners is a way to achieve robustness to changes in model assumptions, initial parameters, training set perturbations and noise. Different models can be heterogeneously combined to be able to succeed in union when the inherent structural assumptions of some do not hold. Similarly to reduce the overall sensitivity to different starting parameters and noisy training examples, multiple instances of the same model can be combined. For the latter, two well-known algorithms are *Bagging* [6] and *AdaBoost* [7, 8].

Model aggregation algorithms have been proposed and analyzed for classification in much more detail than regression, possibly due to the wider availability of real-life applications. Adapting classification algorithms to regression, while trivial for many other machine learning methods, raises some issues in this setting.

In this section we describe the Bagging and AdaBoost algorithms and several

variants in the context of regression.

2.1. Model Aggregation

The algorithms discussed below are sometimes called *master* algorithms because they take another algorithm as the *base* algorithm and improve its performance.

Their common approach is to create multiple model instances using the given training data, which are combined to get an aggregate model. The new model combines the outputs of the base models to form its output.

A *master algorithm* f uses a base algorithm g, and its own parameters Φ^f to produce an *aggregate model* H for a training set $\mathcal{X}$. The aggregate model consists of a set of k instances for the base model h (defined by their parameters θ_k^h) whose outputs are combined by an aggregation function F (parameterized by θ^F) for evaluation:

$$f(\mathcal{X}; g, \Phi^f) = \theta^H = (\{\theta_k^h\}_{k=1}^K, \theta^F) \tag{2.1}$$

$$H(\mathbf{x}; \theta^H) = F(\{h(\mathbf{x}; \theta_k^h)\}_{k=1}^K; \theta_F) \tag{2.2}$$

2.2. Bagging

Bootstrapping is a method for creating many different training subsets from a single training set. The subsets, called the *bootstrap samples*, are formed by randomly selecting with replacement a fixed number of examples from the original training set. Random selection with replacement allows examples to be in more than one subset, or even copied multiple times in the same subset. This enables the bootstrap samples to be adequately dissimilar while providing the freedom to keep their size usefully large.

The *Bagging* (Bootstrap Aggregating) algorithm [6] uses bootstrapping on the

- *Training*
  - · For each h_i of K base models
    - · Randomly select with replacement M examples from the training set.
    - · Train base model h_i using the selected examples.
- *Evaluation*
  - · Given input x, for each base model h_i
    - · Evaluate base model output $y_i = h_i(x)$
  - · Output the mean $\overline{y} = \frac{1}{K} \sum_{i=1}^{K} y_i$

Figure 2.1. The Bagging algorithm (for regression)

training set to create many varied but overlapping training sets. The base algorithm is used to create different base model instances using each bootstrap sample.

K samples $\mathcal{X}_i$ are created from the training set $\mathcal{X}$, each containing M examples selected randomly with replacement using uniform probabilities. Every sample $\mathcal{X}_i$ is learned by a different base model instance h_i, and the ensemble output is the average of all base model outputs for a given input:

$$H(\mathbf{x}) = \frac{1}{K} \sum_{i=1}^{K} h_i(\mathbf{x}) \tag{2.3}$$

The algorithm is shown in Figure 2.1.

The best accuracy enhancement by Bagging is when the constructed base model instances are very different from each other. Averaging does not have much effect when the outputs are already close. Hence, the most suitable base models for Bagging are *unstable* models, where small changes in the training set can result in large changes in

- *Training*

  · Remove M of the N training examples to use for validation.

  · For each ratio $r_j \in [0, 1]$ in the given ratio set

    · Construct Bagging model b_j using samples of size $r_j(N - M)$.

    · Evaluate b_j on the M unused validation examples to get error e_j.

  · Choose Bagging model b_j with the smallest validation error e_j.

Figure 2.2. The Best-Ratio Bagging algorithm

model parameters. Multi-layer perceptrons and regression trees are good candidates.

2.2.1. Best-Ratio Bagging

The particular bootstrap sample size being used has an effect on the performance of Bagging. A very large sample size makes the samples too similar to benefit from averaging, while selecting them too small produces diverse but very poor base models which might not be remedied enough by averaging. The optimal ratio of bootstrap sample size to training set size depends on the particular data being used. Large training sets will need smaller ratios, while complicated data will call for a larger ratio than simple data. Instead of finetuning this ratio by hand for each application, we propose a method for automating a coarse adjustment.

Best-Ratio Bagging removes a number of randomly selected examples from the training set as a *validation set* $\mathcal{V}$, and performs multiple Bagging instances on the remaining training set $\mathcal{X}'$ for a range of bootstrap sample size to training set size ratios r_j, $j = 1, \ldots, J$. Each Bagging run uses the same training data $\mathcal{X}'$, only with a different bootstrapping ratio r_j. The resulting Bagging models are compared by their errors on the validation set $\mathcal{V}$, and the Bagging model with the lowest validation error is chosen as the final model.

- *Training*

  · Randomly divide the training set into K equal partitions.

  · For each h_i of K base models

    · Train base model h_i using all examples except those in partition i.

Figure 2.3. The Cross-Validation Aggregating (CVA) algorithm

In the AdaBoost family of algorithms sample size is usually taken equal to the training set size, since the adaptive parameters make variation by subsetting obsolete. In comparison to Bagging this leaves us with one less free algorithm parameter to be manually adjusted. It also creates a problem for comparing AdaBoost algorithms and Bagging as peers, because we will have multiple cases of Bagging to compare with a single AdaBoost instance. Best-Ratio Bagging is useful for illustrating the best case performance of Bagging with respect to sample size in such comparisons.

2.2.2. Weighted Bagging

Bagging takes a simple average of base model outputs. Without radically modifying the training procedure, only the evaluation part of AdaBoost can be adopted to use a weighted median instead. The weights (confidences) can be calculated as in AdaBoost, using average loss with respect to a loss function of choice. See Section 2.3.2.1 for the computation of confidence and weighted median.

2.2.3. Cross-Validation Aggregating (CVA)

For the purpose of producing multiple similar but perturbed subsets from one training set, *K-fold cross-validation* is an alternative to bootstrapping. Instead of using random sampling to create the subsets, the training set $\mathcal{X}$ is randomly divided into K equally sized parts $\mathcal{V}_i$, and each training subsets are set to be $\mathcal{X}_i = \mathcal{X} - \mathcal{V}_i$. That is, each base model h_i is trained using the examples *not* in $\mathcal{V}_i$. We will call this

algorithm *Cross-Validation Aggregating* (see Figure 2.3). Evaluation is as in Bagging, combining the base outputs by averaging.

As opposed to random selection with replacement in bootstrapping, cross-validation is guaranteed to use all training examples exactly once in exactly $K-1$ subsets. For small K, this leads to more efficient use of data than bootstrapping which might skip some examples and use others multiple times beyond necessity.

However as K increases, the base models are trained on increasingly similar subsets, which should decrease the positive effect of combining. The extreme case is *leave-one-out* cross-validation where for a training set of size N there are N subsets of size $N-1$, each excluding a single different example. For all but the smallest training sets, the subsets will be almost identical, probably learning the same model with each other and the original set, and not much will be gained by aggregating them.

The only random step in CVA is the partitioning of examples, and this relative determinism in comparison to Bagging suggests that for multiple runs on the same data, CVA is more likely to produce similar models. That is, the ensemble model generated by CVA should have less variance over multiple runs. In this sense, CVA should be more stable than Bagging.

Note that there is no parameter in CVA corresponding to the bootstrap sample ratio, since the number of subsets determines the subset size.

2.3. The AdaBoost Approach

The sampling procedure of Bagging assigns an equal probability of selection to each example for each sample. Since individual samples, and hence the models they train, are independent of each other, their collective success is through mere redundancy. The *boosting* approach differs from Bagging by using the base models in active collaboration, working to compensate for the deficiencies of one another. The general idea is to learn a group of models in sequence, where each model concentrates more on

the examples where the previous model had high error. Different ways of realizing this dynamic focus leads to different boosting algorithms.

AdaBoost (*Ada*ptive *Boost*ing) [7, 8] is an efficient and popular implementation of the boosting principle. Like boosting in general, AdaBoost has been specified, applied and analyzed with much deeper interest for classification than for regression.

The original AdaBoost classification algorithm improves on Bagging by attempting to select examples more intelligently. Sampling is now sequential, one sample being selected after another, and each sample is selected using probabilities affected by the previous samples. Selection is still random with replacement, but according to dynamic probabilities assigned to each training example. For the first sample all example probabilities are initialized to be equal, as in Bagging. The first model is trained on this sample, and tested on the whole training set. Examples misclassified by the first model are then updated to have a higher probability of being selected for the subsequent sample. In succession, each model tries to correct the errors of the previous one, and the overall combined model is continuously complemented where most necessary.

With the flexibility in output labels, regression allows greater freedom in constructing algorithms. The basic concept of AdaBoost, relatively constrained in classification, can be generalized in more than one way for regression.

In their work introducing AdaBoost, Freund & Schapire [7] include a version for regression (*AdaBoost.R*) which discretizes each regression prediction into many two-class classifications. However, motivated from a theoretical construction, it has too severe restrictions to be useful in practice, as we shall examine.

More on the lines of classification AdaBoost, Drucker [9] and Zemel & Pitassi [10] give *distribution-based* algorithms where, like Bagging, all models learn the actual training labels. The dynamic parameters are scalar example selection probabilities, adjusted as in classification.

Yet another group of algorithms [11, 12, 13], although from different viewpoints, all aim to minimize *residual* error through the iterations. The first model tries to learn the actual training outputs, but the next model is trained to predict the differences between the first model's outputs and the actual targets. Proceeding this way, at each step the new model learns the error remaining from the previous models. Unlike classical AdaBoost, in these *relabeling* algorithms the selection probabilities are not necessarily dynamic, but training labels are, so the models are not trained to learn the same function.

For our experiments we use the originally proposed AdaBoost.R from the dynamic-loss algorithms, both Drucker's and Zemel & Pitassi's fixed-label algorithms, and Friedman's residual methods LAD_Boost and LS_Boost.

2.3.1. AdaBoost.R

The original AdaBoost adaptation for regression *AdaBoost.R* suggested alongside the classification algorithm is based on decomposing the regression problem into infinitely many classification tasks, where for each output value a classifier decides whether the output is above or below [7]. This theoretical foundation does lead to a feasible implementation, but involves keeping track of updatable and integrable loss functions, differing for each example. Furthermore, the base learner must be able to accommodate such dynamic loss, redefinable per example. The original suggestion is to initialize the loss functions as absolute difference from a center value in which case the function stays piecewise linear through the piecewise multiplicative updates. This *dynamic-loss* approach is also adopted by Ridgeway *et al.* [14] where the dynamic loss functions are discretely approximated and initialized to Laplace distributions. However their experiments on various datasets using naive Bayes base learners yield no significant justification to afford a per-example redefinable loss, seriously constraining the choice of base learners if not time complexity.

The algorithm is given in Figure 2.4 with modified notation. In our implementation we use discrete approximations to the weight distributions. Also we linearly scale

- *Training*

  · For each training example $(\mathbf{x}^t, r^t)$ where $t = 1, \ldots, N$

    · Initialize weight distribution $w^1_{t,y} \leftarrow |y - r^t|$.

  · For each base model h_i where $i = 1, \ldots, K$

    · Normalize weight distributions $w^i_{t,y} \leftarrow \frac{w^i_{t,y}}{Z}$.

      where $Z = \sum_{t=1}^{N} \int_0^1 |y - r^t| dy$.

    · Train base model h_i using the example distributions $w^i_{t,y}$.

    · Evaluate the base model outputs $y^t_i = h_i(\mathbf{x}^t)$.

    · Calculate the average loss $\overline{L}_i = \sum_{t=1}^{N} | \int_{r^t}^{y^t_i} w^i_{t,y} dy |$

    · Calculate $c_i = \ln(\frac{1-\overline{L}_i}{\overline{L}_i})$.

    · Update distributions

$$w^{i+1}_{t,y} = \begin{cases} w^i_{t,y} & \text{if } r^t \le y \le y^t_i \text{ or } y^t_i \le y \le r^t \\ w^i_{t,y} \exp(-c_i) & \text{otherwise} \end{cases}$$

- *Evaluation*

  · Given input $\mathbf{x}$, for each base model h_i

    · Evaluate base model output $y_i = h_i(\mathbf{x})$

  · Output the weighted median

$$H(\mathbf{x}) = \inf\{y_i : \sum_{j:y_j \le y_i} c_j \ge \frac{1}{2} \sum_j c_j\}$$

Figure 2.4. The original AdaBoost.R [7]

our training outputs to $[0, 1]$ as the algorithm requires.

2.3.2. Distribution-Based Algorithms

2.3.2.1. Drucker's AdaBoost. Drucker's AdaBoost algorithm [9] is given in Figure 2.5. The first sample is selected uniform randomly, as in Bagging. The model generated by this sample then evaluates all examples in the training set, and the selection probabilities are modified to favor examples with high error. Thus the next model will be trained where the first model was weak. This process is repeated, each model modifying the probabilities for the next model, until all K models are constructed. The ensemble output is the weighted median of the base model outputs, weighted by the models' training confidences.

The notation in Figure 2.5 is rearranged from the original to resemble the algorithm ZEMEL-PITASSI for comparison, although the algorithm remains unchanged. An element's probability changes with two factors: the loss it incurs alone and the average (weighted) loss over all examples in that iteration.

At each step i the algorithm minimizes the error function

$$J_i = \sum_{t=1}^{N} \frac{1}{\exp(c_i)} \exp\left(c_i L_i^t\right) \tag{2.4}$$

by minimizing per-example losses. Note that another parameter of the function is c_i, a measure of confidence over all examples, also used as the *combination coefficient* during evaluation. While Drucker's AdaBoost chooses

$$c_i = \ln\left(\frac{1 - \overline{L}_i}{\overline{L}_i}\right) \tag{2.5}$$

to minimize error, this appears to be an *ad hoc* adoption of the analytical result from the similar error function in classification.

For the per-example loss function three candidates were suggested: linear loss $L =$

- *Training*

  · For each training example $(\mathbf{x}^t, r^t)$ where $t = 1, \ldots, N$

    · Initialize probability $p^t \leftarrow \frac{1}{N}$.

  · For each base model h_i where $i = 1, \ldots, K$

    · Randomly select N training examples with replacement,

      where the selection probability of example t is p^t.

    · Train base model h_i using the selected examples.

    · For each example in the training set,

      · Evaluate the base model output $y_i^t = h_i(\mathbf{x}^t)$.

      · Calculate the loss $L_i^t = L(|y_i^t - r^t|) \in [0, 1]$

        e.g. a linear loss $L_i^t = |y_i^t - r^t| / \sup_j |y_j^t - r^t|$

    · Calculate the average loss $\overline{L}_i = \sum_{t=1}^{N} L_i^t p^t$

    · Set $c_i = \ln(\frac{1-\overline{L}_i}{\overline{L}_i})$.

    · Calculate $J_i^t = \frac{1}{\exp(c_i)} \exp(c_i L_i^t)$

    · Update probabilities $p^t \leftarrow p^t J_i^t$.

    · Normalize probabilities $p^t \leftarrow p^t / \sum_{j=1}^{N} p^j$.

- *Evaluation*

  · Given input $\mathbf{x}$, for each base model h_i

    · Evaluate base model output $y_i = h_i(\mathbf{x})$

  · Output the weighted median of y_i by c_i

$$H(\mathbf{x}) = \inf\{y_i : \sum_{j:y_j \le y_i} c_j \ge \frac{1}{2} \sum_j c_j\}$$

Figure 2.5. Drucker's AdaBoost algorithm [9]

- *Training*

  · For each training example $(\mathbf{x}^t, r^t)$ where $t = 1, \ldots, N$

    · Initialize probability $p^t \leftarrow \frac{1}{N}$.

  · For each base model h_i where $i = 1, \ldots, K$

    · Randomly select N training examples with replacement,
    where the selection probability of example t is p^t.

    · Train base model h_i using the selected examples.

    · For each example in the training set,

      · Evaluate the base model output $y_i^t = h_i(\mathbf{x}^t)$.

    · Set $0 < c_i \leq 1$ to minimize $\sum_t J_i^t$ (using line search)
    where $J_i^t = \frac{1}{\sqrt{c_i}} \exp\left[c_i |y_i^t - r^t|^2\right]$

    · Update probabilities $p^t \leftarrow p^t J_i^t$.

    · Normalize probabilities $p^t \leftarrow p^t / \sum_{j=1}^{N} p^j$.

- *Evaluation*

  · Given input x, output the weighted mean

    · $H(\mathbf{x}) = \sum_i c_i h_i(\mathbf{x}) / \sum_i c_i$

Figure 2.6. Zemel & Pitassi's algorithm

$|y-r|/D$, square loss $L_S = |y-r|^2/D^2$ and exponential loss $L_{exp} = 1-\exp[-|y-r|/D]$, where $D = \sup_t |y^t - r^t|$. We used linear loss (absolute difference) as DRUCKER.AD and square loss as DRUCKER.S for the experiments.

The evaluated ensemble output is the weighted median of model outputs, weighted by the combination coefficients c_i. The weighted median can be computed by first sorting the outputs in order of magnitude, and then summing their weights until the sum exceeds half the weight total. If the weights were integers, this would be analogous to duplicating the outputs by their weights and taking the regular median.

2.3.2.2. The Zemel-Pitassi Algorithm (using square loss). Zemel & Pitassi provide an algorithm similar to Drucker's, but with alternative mathematical particulars. The algorithm is given in Figure 2.6. It is illustrative to examine the algorithm in comparison to Drucker's.

Here the error function is

$$J_i = \sum_{t=1}^{N} \frac{1}{\sqrt{c_i}} \exp\left[c_i |y_i^t - r^t|^2\right] \tag{2.6}$$

where the c_i is the combination coefficient.

The loss function is fixed as squared error, and not scaled to $[0, 1]$ by D.

The multiplier is $\frac{1}{\sqrt{c_i}}$ here, replacing Drucker's $\frac{1}{\exp(c_i)}$. Nevertheless with $0 < c_i \leq 1$ they behave similarly (except near the boundaries), so this alone should not cause a significant difference in performance.

Notably Zemel & Pitassi acknowledge that given this error function, c_i cannot be analytically determined, and the algorithm resorts to simple line search to optimize it.

Finally, to combine base model outputs this algorithm uses weighted mean as opposed to Drucker's weighted median.

To compare algorithms on equal terms, we implemented this algorithm as Zemel-Pitassi.S and Zemel-Pitassi.AD, using the original square loss and linear loss respectively. To get the best approximator of minimum absolute error, we replaced weighted mean by weighted median in Zemel-Pitassi.AD. See Section 2.3.2.1 for weighted median computation.

2.3.3. Relabeling Algorithms

Unlike previous methods, in relabeling algorithms the base models are not trained to predict the actual training set labels. The per-example training errors of the current combined model are called *residues*, and each model learns artificial labels formed using the residues. After training each model i the residues are updated by subtracting the prediction y_i^t of the new model weighted by its coefficient c_i.

Due to the subtraction of model errors from the residues at each step, the combination rule is additive, using a weighted sum. This incremental addition of models is motivated as gradient descent in function space in [12].

The principle of AdaBoost, emphasizing difficult (high-error) examples, can be applied in two ways in relabeling algorithms. Example probabilities may be manipulated as before, or training labels may be transformed. Both methods are obtainable from the loss function being used.

2.3.3.1. The LS_Boost Algorithm. The Least-Squares regression boosting algorithm is an instantiation of Friedman's gradient-based boosting strategy [12] using square loss $L = (y - r)^2/2$ where r is the actual training label and y is the current cumulative output $y_i = c_0 + \sum_{j=1}^{i} c_j h_j + c_i h_i = y_{i-1} + c_i h_i$. The new training labels $\hat{r}$ should be set to the direction that minimizes the loss, which is the negative gradient with respect to y evaluated at y_{i-1}. So $\hat{r} = [-\partial L/\partial y]_{y=y_{i-1}} = r - y_{i-1}$ which is the current residual error. Substituting into the loss, we get the training error

$$E = \sum_{t=1}^{N} [c_i h_i^t - \hat{r}^t] \tag{2.7}$$

where $\hat{r}^t$ are the current residual labels. Setting $\partial E/\partial c_i = 0$ to find the combination coefficients c_i yields the algorithm LS_Boost, shown in Figure 2.7.

Note that the bias term set in the initial step is redundant for base models that

- *Training*

  · Set $c_0 = \frac{1}{N} \sum_{t=1}^{N} r^t$.

  · For $t = 1, \ldots, N$, initialize residues $\hat{r}^t \leftarrow r^t$.

  · For each base model h_i where $i = 1, \ldots, K$

    · Train base model h_i on $\{\mathbf{x}^t, \hat{r}^t\}_{t=1}^{N}$.

    · For $t = 1, \ldots, N$ where $h_i(\mathbf{x}^t) \neq 0$,

      · Evaluate outputs $y_i^t = h_i(\mathbf{x}^t)$.

    · Set $c_i = \sum_t \hat{r}^t y_i^t / \sum_t (y_i^t)^2$.

    · Update residues $\hat{r}^t \leftarrow \hat{r}^t - c_i y_i^t$.

- *Evaluation*

  · Given input x, output the weighted sum

    · $H(\mathbf{x}) = \sum_i c_i h_i(\mathbf{x})$

Figure 2.7. The LS_Boost algorithm

already have or are able to simulate a bias term. It should only be necessary if they are local functions expensive at approximating global bias.

Duffy & Helmbold give an algorithm SQUARELEV.R (Square Leveraging for Regression) [11] which only minimizes the variance of the residuals, normalizing them to zero mean at the very end of training. However this trivially amounts to using the same square loss function leading to the same coefficients, only adding the bias term at the end. The two algorithms are identical in effect.

SQUARELEV.C, a variant of SQUARELEV.R, is more interesting in that it takes the alternative approach of modifying example probabilities to achieve boosting. The residues are calculated as before, but the base learner is fed not the residues $\hat{r}$, but their signs $sign(\hat{r}) \in \{-1, +1\}$. However now the distribution weight of each example is made proportional to $|\hat{r}|$, so each example is still "emphasized" in proportion to its residual error. At the cost of resampling or handling probabilities in training, SQUARELEV.C allows using a binary classifier as the base model.

2.3.3.2. The LAD_BOOST Algorithm. The Least-Absolute-Deviation regression boosting algorithm is derived from the same gradient-based framework as LS_BOOST, this time with linear loss (absolute deviation). In the original it was further specialized to a certain type of regression trees, but here we use the general form. The algorithm is shown in Figure 2.8.

The gradient of LAD_BOOST's linear loss translates to the sign of the residue for base model targets. This means that all base models are trained on $\{+1, -1\}$ labels, which also allows using classifiers for the base algorithm as a bonus.

The coefficient c_i should minimize the total absolute distance to the residues

$$c_i \quad = \quad \arg\min_c \sum_{t=1}^{N} \left| \hat{r}^t - c y_i^t \right|$$

- *Training*

  · Set $c_0 = \frac{1}{N}\sum_{t=1}^{N} r^t$.

  · For $t = 1, \ldots, N$, initialize residues $\hat{r}^t \leftarrow r^t - c_0$.

  · For each base model h_i where $i = 1, \ldots, K$

    · Train base model h_i on $\{\mathbf{x}^t, sign(\hat{r}^t)\}_{t=1}^{N}$.

    · For $t = 1, \ldots, N$ where $h_i(\mathbf{x}^t) \neq 0$,

      · Evaluate outputs $y_i^t = h_i(\mathbf{x}^t)$.

      · Calculate weighted inverse outputs $z_i^t = \hat{r}^t / y_i^t$.

    · Set c_i to the weighted median of z_i^t by $|y_i^t|$:

      $c_i = \inf\{z_i^t : \sum_{j: z_i^j \leq z_i^t} |y_i^j| \geq \frac{1}{2}\sum_j |y_i^j|\}$

    · Update residues $\hat{r}^t \leftarrow \hat{r}^t - c_i y_i^t$.

- *Evaluation*

  · Given input x, output the weighted sum

    · $H(\mathbf{x}) = c_0 + \sum_i c_i h_i(\mathbf{x})$

Figure 2.8. The LAD_Boost algorithm

$$
\begin{aligned}
&= \arg\min_{c} \sum_{t=1}^{N} \left|y_i^t\right| \cdot \left|\frac{\hat{r}^t}{y_i^t} - c\right| \\
&= \arg\min_{c} \sum_{t=1}^{N} \left|y_i^t\right| \cdot \left|z_i^t - c\right| \\
&= \inf\{z_i^t : \sum_{j:z_i^j \le z_i^t} |y_i^j| \ge \frac{1}{2} \sum_{j} |y_i^j|\}
\end{aligned}
$$

which is the weighted median.

Once the optimal weight is found, the residues are updated to exclude the error corrected by the new model, and iteration may proceed with the next model.

3. SUPPORT VECTOR MACHINES

In this section we describe the family of machine learning methods called *Support Vector Machines* with a focus on regression problems. After illustrating the fundamental idea in the linear case, the *kernel* concept will be introduced for generalization to nonlinear problems.

3.1. Overview

The *Support Vector Machine* (SVM) is a machine learning method that has attracted considerable research and industry attention since its relatively recent development. Supported by both a sound background in statistical learning theory and an application-oriented research focus, it has become one of the most successful methods with respect to generalization performance.

The algorithm is characterized by (and named after) selecting and storing a subset of the training examples as the important ones (called the *support vectors*), such that knowing only these critical training examples is enough to label any previously unseen input. For example, in a linearly separable two-class classification problem, a number of examples that are closest to the assumed discrimination boundary are selected as the support vectors. All other training examples are then unnecessary, since the support vectors alone sufficiently constrain the boundary from both sides. If training took place again with all but the support vector examples different and such that no new example came closer to the boundary, then the same support vectors would be chosen, and the same boundary would be defined for evaluation, although much of the training set was different. This invariance to changes in "irrelevant" training data is the merit of Support Vector Machines, making them resistant to overfitting noisy data.

Compared to non-parametric learners like k-Nearest Neighbors which store all training examples for evaluation, the much abused term "semi-parametric" is applicable to the SVM algorithm, though in a very different way than, e.g., neural networks.

The fundamental Support Vector Machine formulation is readily applicable to two-class and multi-class classification, regression and density estimation. In accord with the emphasis of our research, we will introduce and examine the concepts in the context of regression problems in particular.

3.2. The Linear Problem

Given training data $\mathcal{X} = \{\mathbf{x}^t, r^t\}_{t=1}^N$ and a threshold parameter ε, the most basic form of SVM seeks a linear solution that has at most ε absolute deviation from the training label for each example. The linear function can be expressed as

$$f(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b \tag{3.1}$$

where $\mathbf{w} \in \Re^d$ and $b \in \Re$ are the parameters. $vw^{\mathrm{T}}\mathbf{x}$ denotes the inner product of the input and the weight vector.

If there are multiple functions satisfying the error bound, the simplest should be preferred, with Occam's Razor in mind. Simplicity in this setting is the *flatness* of the function, controlled by the magnitude of the vector $\mathbf{w}$.

If such a function exists, it can be found by solving the convex optimization problem

$$\begin{aligned} &\text{minimize} \quad \tfrac{1}{2}\|\mathbf{w}\|^2 \\ &\text{subject to} \quad \begin{cases} r^t - \mathbf{w}^{\mathrm{T}}\mathbf{x}^t - b \leq \varepsilon \\ \mathbf{w}^{\mathrm{T}}\mathbf{x}^t + b - r^t \leq \varepsilon \end{cases} \end{aligned} \tag{3.2}$$

However this problem is not *feasible* if a perfect solution does not exist for the given training set and error bound. In that case some errors beyond ε must be allowed by the problem if a solution is to be found at all. This is achieved by adding

slack variables to relax the possibly infeasible constraints. The optimization problem becomes

$$\begin{aligned} &\text{minimize} \quad \tfrac{1}{2}\|\mathbf{w}\|^2 + C\sum_{t=1}^{N}(\xi_t + \xi_t^*) \\ &\text{subject to} \quad \begin{cases} r^t - \mathbf{w}^{\mathrm{T}}\mathbf{x}^t - b \leq \varepsilon + \xi_t \\ \mathbf{w}^{\mathrm{T}}\mathbf{x}^t + b - r^t \leq \varepsilon + \xi_t^* \\ \xi_t, \xi_t^* \geq 0 \end{cases} \end{aligned} \tag{3.3}$$

This formulation effectively employs an ε-insensitive error function. Deviations are completely tolerated up to ε, and penalized linearly thereafter:

$$E(y, r) = \begin{cases} 0 & \text{if } |y - r| \leq \varepsilon \\ |y - r| - \varepsilon & \text{otherwise} \end{cases} \tag{3.4}$$

The constant $C > 0$ in the objective function determines the balance between the simplicity of the solution and the tolerance to errors above ε. A lower value of C will emphasize minimizing the weights more than the slack variables, yielding a flatter but more error-prone function.

3.3. A Solution Strategy

The optimization problem at hand can be solved more conveniently in its dual formulation:

$$\begin{aligned} &\text{maximize} \quad \begin{cases} -\tfrac{1}{2}\sum_{t=1}^{N}\sum_{s=1}^{N}(\alpha_t - \alpha_t^*)(\alpha_s - \alpha_s^*)\mathbf{x}^{t\mathrm{T}}\mathbf{x}^s \\ -\varepsilon\sum_{t=1}^{N}(\alpha_t + \alpha_t^*) + \sum_{t=1}^{N} r^t(\alpha_t - \alpha_t^*) \end{cases} \\ &\text{subject to} \quad \begin{cases} \sum_{t=1}^{N}(\alpha_t - \alpha_t^*) = 0 \\ \alpha_t, \alpha_t^* \in [0, C] \end{cases} \end{aligned} \tag{3.5}$$

The constant C is now the upper bound for the new slack variables α_t and α_t^*, and is called the *capacity* parameter.

The dual formulation also yields

$$\mathbf{w} = \sum_{t=1}^{N} (\alpha_t - \alpha_t^*)\mathbf{x}^t \tag{3.6}$$

and hence

$$f(x) = \sum_{t=1}^{N} (\alpha_t - \alpha_t^*)\mathbf{x}^{\mathrm{T}}\mathbf{x}^t + b \tag{3.7}$$

This expression shows that the parameter vector $\mathbf{w}$ can be written as a linear combination of the training examples $\mathbf{x}^t$. This fact, that a weighted sum of the training examples completely describes the solution function, is the essence of Support Vector Machines. The above expression is therefore called the *Support Vector expansion.*

By the Karush-Kuhn-Tucker (KKT) conditions [15, 16], the product of the dual variables and the constraints should be zero at the optimal solution:

$$\begin{aligned} \alpha_t(\varepsilon + \xi_t - r^t + \mathbf{w}^{\mathrm{T}}\mathbf{x}^t + b) &= 0 \\ \alpha_t^*(\varepsilon + \xi_t^* + r^t - \mathbf{w}^{\mathrm{T}}\mathbf{x}^t - b) &= 0 \end{aligned} \tag{3.8}$$

and

$$\begin{aligned} (\alpha_t - C)\xi_t &= 0 \\ (\alpha_t^* - C)\xi_t^* &= 0 \end{aligned} \tag{3.9}$$

It can be seen that $\alpha_t\alpha_t^* = 0$ so only one slack variable may be nonzero for a given example. Equation (3.9) implies that the examples outside the ε-insensitive boundary around f are those with $\alpha_t^{(*)} = C$. Furthermore, if $\mathbf{x}^t$ is inside the boundary, then the second factors in Equation (3.8) will be nonzero, making both α_t and α_t^* zero. The

latter illustrates a very important property of SVMs: the Support Vector expansion is *sparse*, i.e., only the examples with error ε and above, called the *Support Vectors*, are necessary to express the solution. The rest, which are inside the boundary, may be discarded since their coefficients will be zero. This sparsity property allows SVMs to store only the necessary examples after training, reducing memory requirements and time complexity.

From Equation (3.8) we also get an expression for computing b given an example $\mathbf{x}^t$:

$$b = \begin{cases} r^t - \mathbf{w}^{\mathrm{T}}\mathbf{x}^t - \varepsilon & \text{for } \alpha_t \in (0, C) \\ r^t - \mathbf{w}^{\mathrm{T}}\mathbf{x}^t + \varepsilon & \text{for } \alpha_t^* \in (0, C) \end{cases} \tag{3.10}$$

The final b may be found by averaging over all examples having $\alpha_t \in (0, C)$ or $\alpha_t^* \in (0, C)$

3.4. Nonlinear Kernels

Although the SVM derivation described so far is exclusively tailored for the linear problem, it can also be used for the nonlinear case, for example through preprocessing. By introducing new input dimensions as nonlinear transformations of the original input, the nonlinear problem can be projected into a feature space where it has a linear solution. Then the linear SVM algorithm can be used as is. However it is not always possible to find the necessary feature mapping to a linear space, and as more nonlinear features are added, this approach quickly becomes computationally infeasible.

Fortunately, looking at the SVM equations reveals that input vectors only appear in dot products, so instead of explicitly finding a nonlinear mapping $\Phi(\mathbf{x}^t)$ into a feature space $\mathcal{F}$, it is enough to find a *kernel* function $k(\mathbf{x}^t, \mathbf{x}^s)$ that directly computes $\Phi(\mathbf{x}^t)^{\mathrm{T}}\Phi(\mathbf{x}^s)$. It is thus possible to implicitly work in a higher dimensional feature space without explicitly computing the features.

Using the kernel trick, the SVM algorithm in Equation (3.5) becomes

$$\begin{array}{ll} \text{maximize} & \begin{cases} -\frac{1}{2}\sum_{t=1}^{N}\sum_{s=1}^{N}(\alpha_t-\alpha_t^*)(\alpha_s-\alpha_s^*)k(\mathbf{x}^t,\mathbf{x}^s) \\ \\ -\varepsilon\sum_{t=1}^{N}(\alpha_t+\alpha_t^*)+\sum_{t=1}^{N}r^t(\alpha_t-\alpha_t^*) \end{cases} \\ \text{subject to} & \begin{cases} \sum_{t=1}^{N}(\alpha_t-\alpha_t^*) = 0 \\ \alpha_t,\alpha_t^* \in [0,C] \end{cases} \end{array} \tag{3.11}$$

The Support Vector expansion in Equation (3.7) can now be written as

$$f(x)=\sum_{t=1}^{N}(\alpha_t-\alpha_t^*)k(\mathbf{x}^t,\mathbf{x})+b \tag{3.12}$$

The final general SVM regression algorithm is given in Figure 3.1.

Note that while $\mathbf{w}=\sum_{t=1}^{N}(\alpha_t-\alpha_t^*)\Phi(\mathbf{x}^t)$ is not explicitly computed anymore, it still exists in the original formulation, and its magnitude is minimized as before. Hence the nonlinear SVM ensures "flatness" in the feature space, as opposed to the input space.

The conditions for defining kernel functions are given in [17]. Many useful properties and classes of kernels are described in [18, 19].

Some commonly used SVM kernels are

Polynomial:

$$k(\mathbf{x}^t,\mathbf{x}^s)=({\mathbf{x}^t}^{\mathrm{T}}\mathbf{x}^s+\theta)^p \quad \text{where } p\in\aleph,\ \theta\geq 0 \tag{3.13}$$

- *Training*

  Given the training set $\mathcal{X} = \{\mathbf{x}^t, r^t\}_{t=1}^N$, $\varepsilon > 0$ and $C > 0$,

  · Compute α_t, α_t^* by solving the optimization problem

$$\text{maximize} \quad \begin{cases} -\frac{1}{2}\sum_{t=1}^N \sum_{s=1}^N (\alpha_t - \alpha_t^*)(\alpha_s - \alpha_s^*)k(\mathbf{x}^t, \mathbf{x}^s) \\ \\ -\varepsilon \sum_{t=1}^N (\alpha_t + \alpha_t^*) + \sum_{t=1}^N r^t(\alpha_t - \alpha_t^*) \end{cases}$$

$$\text{subject to} \quad \begin{cases} \sum_{t=1}^N (\alpha_t - \alpha_t^*) & = & 0 \\ \alpha_t, \alpha_t^* & \in & [0, C] \end{cases}$$

  · Compute b by averaging over the examples that have $\alpha_t^{(*)} \in (0, C)$:

$$b_t = \begin{cases} r^t - \sum_{s=1}^N (\alpha_s - \alpha_s^*)k(\mathbf{x}^s, \mathbf{x}) - \varepsilon & \text{for } \alpha_t \in (0, C) \\ \\ r^t - \sum_{s=1}^N (\alpha_s - \alpha_s^*)k(\mathbf{x}^s, \mathbf{x}) + \varepsilon & \text{for } \alpha_t^* \in (0, C) \end{cases}$$

  · Store $\mathcal{S} = \{(\mathbf{x}^t, r^t, \alpha_t, \alpha_t^*) : \ \alpha_t > 0 \text{ or } \alpha_t^* > 0\}$

- *Evaluation*

  · Given input $\mathbf{x}$, output $y(\mathbf{x}) = \sum_{\mathbf{x}^t \in \mathcal{S}} (\alpha_t - \alpha_t^*)k(\mathbf{x}^t, \mathbf{x}) + b$

Figure 3.1. The Support Vector Regression algorithm

Neural:

$$k(\mathbf{x}^t, \mathbf{x}^s) = \tanh(\beta {\mathbf{x}^t}^{\mathrm{T}} \mathbf{x}^s + \theta) \quad \text{where } \beta,\ \theta \geq 0 \tag{3.14}$$

Radial:

$$k(\mathbf{x}^t, \mathbf{x}^s) = \exp(-\gamma \|\mathbf{x}^t - \mathbf{x}^s\|^2) \quad \text{where } \gamma > 0 \tag{3.15}$$

3.5. Tuning Insensitivity

The particular choice of ε directly affects the selection of support vectors, changing their number and hence model complexity. Too small values may result in overly complex models with poor generalization while too large values will ignore too many examples and yield very crude models. The optimal value depends on the noise in the data at hand.

Taking advantage of the dependency between ε and the number of support vectors, a method has been developed to automatically adjust ε by controlling the number of support vectors [20]. With the addition of a new parameter ν, ε is made a minimized variable in the original optimization problem. Making the dual transformation as usual, the ν-SVM algorithm is obtained:

$$\text{maximize} \quad \begin{cases} -\frac{1}{2} \sum_{t=1}^{N} \sum_{s=1}^{N} (\alpha_t - \alpha_t^*)(\alpha_s - \alpha_s^*) k(\mathbf{x}^t, \mathbf{x}^s) \\ + \sum_{t=1}^{N} r^t (\alpha_t - \alpha_t^*) \end{cases}$$

$$\text{subject to} \quad \begin{cases} \sum_{t=1}^{N} (\alpha_t - \alpha_t^*) & = & 0 \\ \sum_{t=1}^{N} (\alpha_t + \alpha_t^*) & \leq & C\nu N \\ \alpha_t, \alpha_t^* & \in & [0, C] \end{cases} \tag{3.16}$$

To adjust ε, the new parameter ν asymptotically specifies the ratio of support vectors in the training set. νN is an upper bound for the number of support vectors outside the ε-boundary ($\alpha^{(*)} = C$) and a lower bound for the total number of support vectors. See [20] for a proof.

3.6. Remarks

Although the bulk of the SVM algorithm is left to a generic optimization algorithm once the problem is formulated, developing a dedicated optimization algorithm for solving SVMs provides many opportunities for improving efficiency. There are various SVM implementations available which considerably reduce complexity by taking advantage of the nature of the problem. For example the explicit computation of b, omitted here for ν-SVM, is in general not necessary, since it is obtained as a by-product of the optimization algorithm.

As a final note, it should be mentioned that SVM models are hardly interpretable, sharing the black-box nature of neural networks, in contrast to e.g. decision trees. For applications where the learned model will be used to deduce human-understandable rules from data, Support Vector Machines provide little use.

4. REX

It is typical of human thinking to manage knowledge in rules, and exceptions to the rules where necessary. In learning the past tense forms of English verbs for example, the rule "append *–ed*" covers most cases, and we call such verbs "regular". The irregular ones are simply memorized, since any attempt to extend the rule to cover irregular verbs would make the rule neither as simple nor as useful.

Exceptions usually also allow generalization in their vicinity. For example a person who knows that the past tense of "bend" is "bent", when faced with the verb "spend" for the first time, will consider both the rule ("spended") and the close exception ("spent"), where indeed the exception generalizes correctly. Of course for many other similar verbs like "suspend" the rule holds, but we see nevertheless that exceptions may be useful beyond predicting just the memorized example.

The algorithm *REx* (**R**ules and **Ex**ceptions) uses this familiar paradigm to tackle machine learning problems. Most of the data is assumed to be produced by a relatively simple rule, the other examples being exceptions to the rule. The idea was first presented in [1], and extensively analyzed and applied for classification in [2, 3, 4, 5], with significant theoretical and empirical results. In this work, we extend the algorithm to regression problems, and compare it to Bagging, AdaBoost variants and Support Vector Machines.

In this section we describe REx for regression.

4.1. Partitioning

The main assumption of REx is that there is a rule that explains most of the data. The algorithm is given another algorithm that learns the rule, and this rule algorithm determines which training examples are the difficult ones, using a *threshold* parameter and cross-validation.

`FindExceptions`(training set $\mathcal{X} = \{(\mathbf{x}^t, r^t)\}_{t=1}^N$, rule algorithm F, threshold $\varepsilon > 0$)
· Randomly divide $\mathcal{X}$ into two equal-sized parts $\mathcal{X}_1$, $\mathcal{X}_2$
· Run rule algorithm F on $\mathcal{X}_1$ to get rule model f_1
· Set $S_1 = \{(\mathbf{x}, r) \in \mathcal{X}_2 : |f_1(\mathbf{x}) - r| > \varepsilon\}$
· Run rule algorithm F on $\mathcal{X}_2$ to get rule model f_2
· Set $S_2 = \{(\mathbf{x}, r) \in \mathcal{X}_1 : |f_2(\mathbf{x}) - r| > \varepsilon\}$
· `return` $S_1 \cup S_2$

Figure 4.1. REx: Determining exceptions

The simplest case is two-fold cross-validation, where the training data is divided into two, the rule is trained on the first part, and tested on the second part. The test examples with error above the threshold value are marked as exceptions. Then the rule is trained from scratch on the second part, and tested on the first part, marking the necessary examples in the first part as exceptions. Combining the two sets of exceptions completes the partitioning of the training set. We employed this two-fold case in our implementation. See Figure 4.1.

4.2. Combining

Once the exceptions have been set apart, they should be put to good use in conjunction with the rule. In classification, the confidence of the rule classifier may dictate whether the input will be checked with the exceptions or not, but simply storing the exception is not enough for regression, since there is no simple way to get a confidence value from a regressor. Training yet another regressor to learn the confidence of the first is not guaranteed to be an easier problem than trying to learn all data directly.

One idea is to have a radius parameter such that if an example is within that radius of an exception, it will be handled by the exception and not the rule. A better way is to make this exception membership continuous, which can be achieved by placing Gaussians centered at each of the exceptions.

If there are J exceptions, there will be J Gaussians with parameters (μ_j, Σ_j). To keep the number of parameters low, we can refrain from using general covariance matrices and Mahalanobis distance, and work with Euclidean distance assuming equal variances in all dimensions. Then parameters can be written as (μ_j, σ_j) where the μ_j are the inputs $\mathbf{x}$ of the exceptions. The j'th Gaussian output for test input $\mathbf{x}^t$ is:

$$p_j^t = \exp\left[-\frac{\|\mathbf{x}^t - \mu_j\|^2}{2\sigma_j^2}\right] \tag{4.1}$$

σ_j can be set to the quarter of the distance to the nearest other exception, so that their Gaussians will marginally overlap if the variances are the same.

4.3. Collaborative REx

When there is a Gaussian for each exception, the combined output can be a linear combination of the rule output and all exception outputs. If we regard the whole combined model as a single model with parameters including the linear weights, all Gaussian parameters and the rule parameters, provided that the rule is a differentiable function, we can train the whole model as one. This allows not only learning the linear weights and the rule, but also finetuning the exception centers and variances to suit the rule better. This algorithm is called *Collaborative REx*, (denoted as C-REx for short), since the rule and the exceptions work together to form the combined output.

4.3.1. Linear Rule

In the case of a linear rule, since linear transformations of a linear function are still linear, the two stages of linearity can be combined into one, connecting input units to the output directly. See Figure 4.2 for a network diagram.

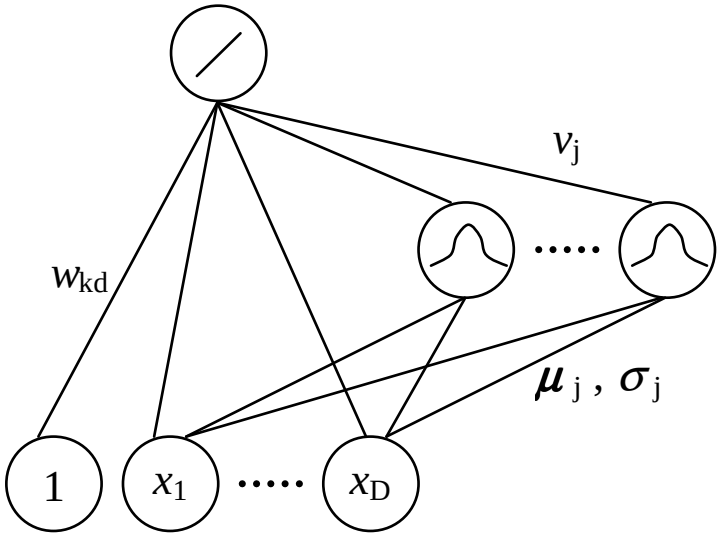


Figure 4.2. Network diagram of C-REx using linear rule

The combined output for input $\mathbf{x}^t$ is then

$$y^t = \sum_{d=0}^{D} w_d x_d^t + \sum_{j=1}^{J} v_j p_j^t \tag{4.2}$$

where x^t includes a bias term $x_0 = 1$.

Gradient descent using a sum-of-squares error yields the training equations:

$$\Delta w_d = \eta(r^t - y^t)x_d^t \tag{4.3}$$

$$\Delta v_j = \eta(r^t - y^t)p_j^t \tag{4.4}$$

$$\Delta \mu_{jd} = \eta(r^t - y^t)v_j p_j^t \left(\frac{x_d^t - \mu_{jd}}{\sigma_j^2}\right) \tag{4.5}$$

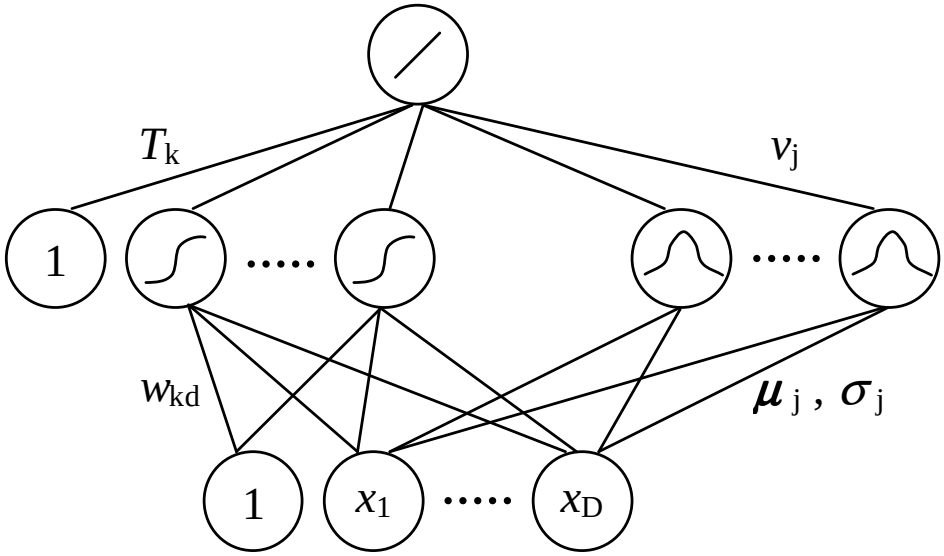


Figure 4.3. Network diagram of C-REx using MLP rule

$$\Delta\sigma_j^2 = \eta(r^t - y^t)v_j p_j^t \frac{\|\mathbf{x}^t - \mu_j\|}{2\sigma_j^4} \tag{4.6}$$

where η is a learning rate constant and may actually be different for each equation.

4.3.2. MLP Rule

Using a multi-layer perceptron for the rule, the second layer is again linear, so the output unit may be shared by the Gaussians as before. The diagram is in Figure 4.3.

The hidden unit outputs are the sigmoid of their net input:

$$\text{sigmoid}(\theta) = \frac{1}{1 + e^{-\theta}} \tag{4.7}$$

$$h_k^t = \text{sigmoid}\left(\sum_{d=0}^{D} w_{kd} x_d^t\right) \tag{4.8}$$

The combined output is:

$$y^t = \sum_{k=0}^{K} T_k h_k^t + \sum_{j=1}^{J} v_j p_j^t \tag{4.9}$$

And the gradient descent update equations are:

$$\Delta T_k = \eta(r^t - y^t)h_k^t \tag{4.10}$$

$$\Delta w_{kd} = \eta(r^t - y^t)T_k h_k^t(1 - h_k^t)x_d^t \tag{4.11}$$

and Equations (4.4), (4.5), (4.6) as before.

4.4. Mixture REx

Instead of taking the linear combination of the Gaussians and the rule, an alternative is to view the model like a *Mixture of Experts* architecture [21], where the exceptions and the rule compete for the output, the one with the maximum gating value being selected. We call this version *Mixture REx*, abbreviated as M-REx. The gating values of the exceptions are based on their Gaussian outputs, and if selected they will return a linear parameter. The rule will have its own Gaussian gating unit p_0 with (μ_0, σ_0^2). For differentiability, the `softmax` function will be used, which is simply the gating values normalized by their sum:

$$g_j^t = \frac{p_j^t}{\sum_{k=0}^{K} p_k^t} \tag{4.12}$$

And the combined network output is:

$$y^t = g_0^t F(\mathbf{x}^t) + \sum_{j=1}^{J} g_j^t v_j \tag{4.13}$$

where $F(\mathbf{x}^t)$ is the rule output.

After tedious differentiation, gradient descent yields:

$$\Delta v_j = \eta(r^t - y^t)g_j^t \tag{4.14}$$

$$\Delta \mu_{jd} = \eta(r^t - y^t)(v_j - y^t)g_j^t \left(\frac{x_d^t - \mu_{jd}}{\sigma_j^2}\right) \tag{4.15}$$

$$\Delta \sigma_j^2 = \eta(r^t - y^t)(v_j - y^t)g_j^t \frac{\|\mathbf{x}^t - \mu_j\|}{2\sigma_j^4} \tag{4.16}$$

where $v_0 \equiv F(\mathbf{x}^t)$ to update the rule mean and variance.

The update equations for rule parameters are, for the linear rule (Figure 4.4):

$$\Delta w_d = \eta g_j^t (r^t - y^t)x_d^t \tag{4.17}$$

and for the MLP rule (Figure 4.5):

$$\Delta T_k = \eta g_j^t (r^t - y^t)h_k^t \tag{4.18}$$

$$\Delta w_{kd} = \eta g_j^t (r^t - y^t)T_k h_k^t (1 - h_k^t)x_d^t \tag{4.19}$$

4.5. Clustering

An overly simple rule model or an inherently discontinuous data may produce many exceptions concentrated in the same input region, most of which are redundant. Even though unnecessary exceptions may be hoped to automatically "disappear" dur-

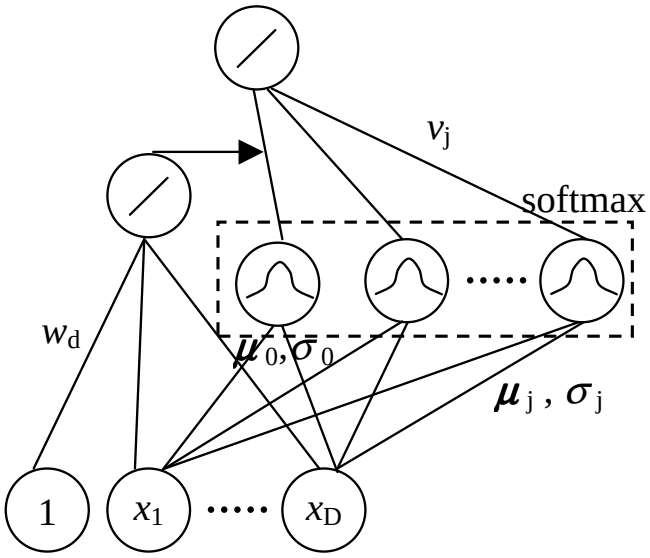


Figure 4.4. Network diagram of M-REx using linear rule

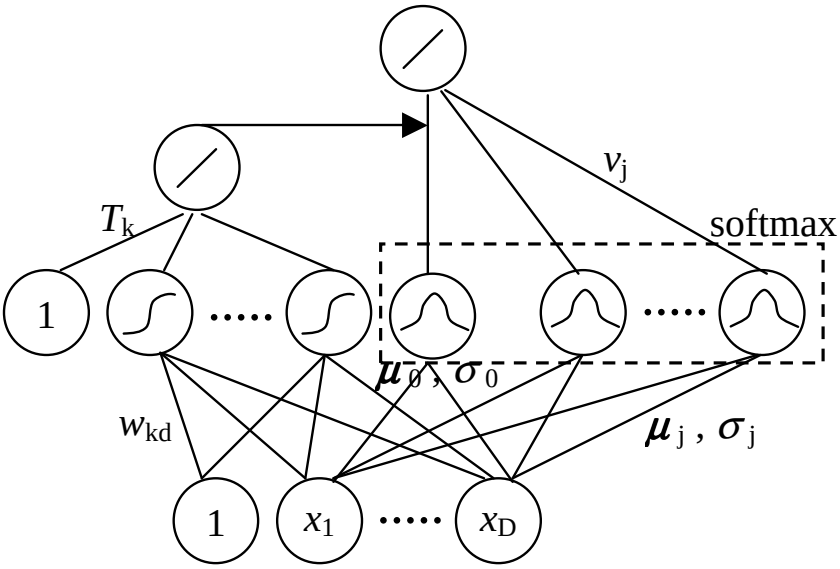


Figure 4.5. Network diagram of M-REx using MLP rule

`FindExceptionClusters`(exception set $\mathcal{S} = \{(\mathbf{x}^s, r^s)\}_s$, cluster count K)

· If $|\mathcal{S}| \leq K$, `return` $\mathcal{S}$

· Initialize cluster set $\mathcal{T} = \{(\mathbf{x}_*^t, r_*^t)\}_t$ to a random subset of $\mathcal{S}$

· Repeat until convergence

  · Associate every exception $(\mathbf{x}^s, r^s)$ in S with the closest cluster $(\mathbf{x}_*^t, r_*^t)$ so that the distance $\|\mathbf{x}_*^t - \mathbf{x}^s\|$ is minimum

  · Move every cluster $(\mathbf{x}_*^t, r_*^t)$ to the mean of its exceptions $(\mathbf{x}^s, r^s)$

· `return` $\mathcal{T}$

Figure 4.6. REx: Finding exception clusters

ing finetuning by their linear combination weights going to zero or variances expanding, chances are high that they will cause the model to overfit. In the very least, they greatly increase the space and time complexity of the algorithm.

To reduce the number of exceptions without throwing away valuable information, we apply *K-means clustering* to find representative exceptions. This algorithm iteratively locates a number of cluster centers on the exception data so that every exception has the minimum squared deviation from the nearest cluster center. Once the clusters are found, the previous exception set is discarded and replaced by the means of the clusters. As a result, close exceptions are represented by the same cluster mean, removing redundancy. See Figure 4.6.

The major downside of K-means clustering is that the number of clusters is prespecified. If a fewer quantity than the actual number of exception groups is given, some groups will be degenerately represented, leading to information loss. If too many clusters are sought, the perils of overfitting and complexity will persist, to the degree of cluster overhead. Although incremental variants have been proposed that automatically determine the number of clusters, they introduce other parameters to be manually adjusted. For REx we applied K-means as is, using K as a fraction of the training set size.

5. SIMULATION RESULTS

We implemented the algorithms described so far, and observed their behavior on several synthetic and real-file datasets. This section details the experimentation process and the results.

5.1. Datasets and Methodology

We used the datasets in Table 5.1 for our experiments. All of them have one-dimensional continuous output labels for regression. All input and output attributes, except those of `syndata`, were z-normalized to zero mean and unit variance to alleviate the effects of using Euclidean distance instead of Mahalanobis distance in the case of multivariate input.

`syndata` is a dataset that we generated synthetically for observing the behaviors of algorithms visually. It has 1000 examples of unidimensional input, and on an output range of $[-15, +15]$ it has Gaussian noise of zero mean and unit variance. See Figure 5.1 for a plot of all examples.

`boston` is the well-known Boston house price dataset from the UCI Machine Learning Repository [22]. It has 506 examples with 12 inputs, trying to predict real estate prices from various attributes.

`calif1000` is a 1000-example subset of the California house price dataset from [22], similar to `boston` in objective. It has eight input attributes.

`abalone` is a dataset for predicting the age of abalone from physical measurements, again from [22]. It has 4177 examples and 10 input attributes.

`prostate` is a prostate cancer dataset from [23]. It aims to estimate the Gleason index of patients, an indicator of cancer pervasion. There are 376 examples with seven

Table 5.1. Properties of the datasets used

	inputs	size
`syndata`	1	1,000
`boston`	12	506
`calif1000`	8	1,000
`abalone`	10	4,177
`prostate`	7	376
`birth`	5	488
`votes`	6	3,107
`kin8fm`	8	8,192
`kin8fh`	8	8,192
`kin8nm`	8	8,192
`kin8nh`	8	8,192

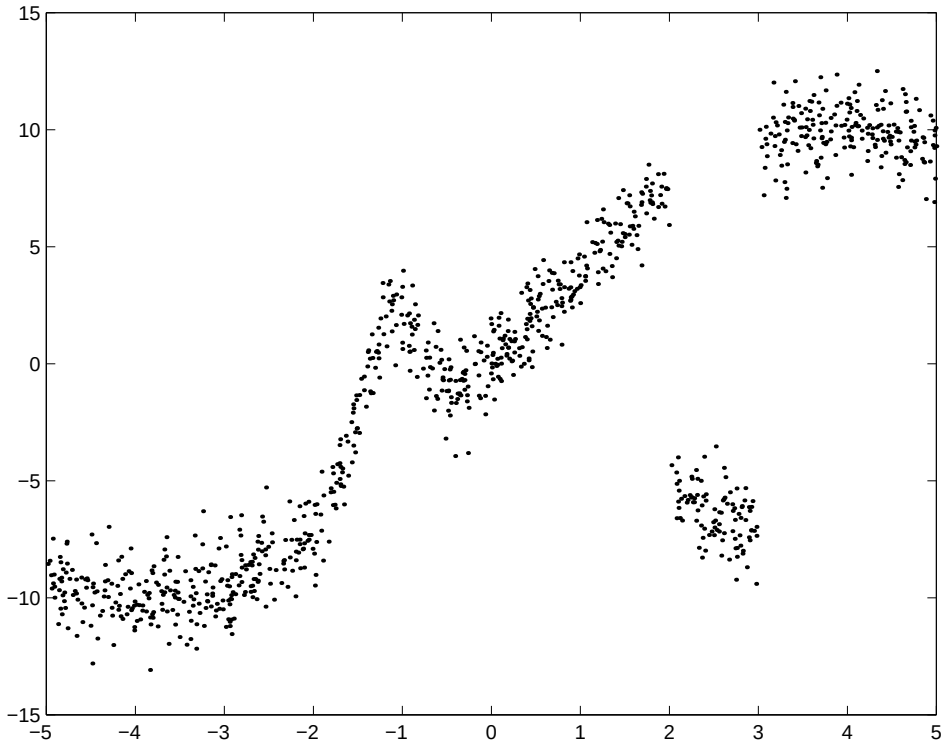


Figure 5.1. Dataset `syndata` (1000 examples)

inputs.

`birth` is another medical dataset from [23], predicting the birth weight of a child based on five attributes of the mother. It contains 488 examples.

`votes` from the StatLib web site of Carnegie Mellon University, also known as `space_ga`, is a spatial data targeted for geographical analyses. It contains 3107 observations on U.S. county votes cast in the 1980 presidential election, used for predicting the number of valid votes from each county based on six attributes.

`kin8` is a set of four related datasets from StatLib. Based on simulated forward kinematics of the eight-link Puma robot arm, each dataset has 8192 examples with eight inputs. The datasets differ in nonlinearity and noise level. `kin8fm` is fairly linear with medium noise, while `kin8fh` is fairly linear with high noise. `kin8nm` is highly nonlinear with medium noise, and `kin8nh` is highly nonlinear with high noise.

For all datasets, we repeated each experiment ten times, using 5×2 *cross-validation*. That is, we created five different random partitions of the data into two equal halves, and we trained the algorithm from scratch on each of the ten halves, using the corresponding other half for testing. All reported errors are the averages of ten such independent test runs on uncontaminated test sets. Standard deviations of the ten test errors are also included where possible. The error bars in the figures indicate one standard deviation above and below the mean error of the ten runs.

Some algorithms with parameters to be manually tuned were evaluated with all possible combinations over sets of parameter values. Others had to be manually "tweaked" by trial and error since complexity and sensitivity constraints prohibited such exhaustive combinations.

5.2. Base Algorithm Results

First we shall evaluate the algorithms that were used to serve as base algorithms to the others that we will examine. For the Bagging and AdaBoost variants we used the J-leaf regression tree as base, and for REx we used linear models and multi-layer perceptrons. The performance of these algorithms without any aggregation or enhancement are investigated below on the datasets. This is both useful to constitute a benchmark for the enhanced versions, and also to shed light on the structure and intrinsic complexity of each dataset.

We tested the Bagging and AdaBoost algorithms using J-leaf regression trees as base models. Our regression tree induction algorithm, given in Figure 5.2, uses constant leaf labels and subdivides the leaf node with the greatest total squared deviation from the mean, until a specified leaf count J is reached or all leaves have a single training element. The leaf count parameter J is used to control model complexity. Note that regression tree algorithms sensitive to output scaling, such as thresholded or variance-bounded models, are not as suitable for our experiments, since they would require separate parameter adjustment between residual algorithm steps.

Linear models were trained analytically by using the pseudo-inverse of the covariance matrix, so do not have any parameters.

Multi-layer perceptrons were trained using gradient descent, using a learning rate parameter η manually adjusted to the data, and a *momentum* parameter $\alpha = 0.5$ to accelerate learning and possibly escape local minima.

For the multi-layer perceptrons we used hidden unit counts of 2, 5, 10, 15, 20, 25 and 30. The same values extended up to 50 were used for the regression trees, but values above 30 are not shown in the graphs. The linear models are plotted as zero-hidden-unit perceptrons.

Figures 5.3, 5.4 and 5.5 show the base algorithm errors on `syndata`, `votes` and

- *Training*
  - · `function ConstructTree`(training set $\mathcal{X}$, leaf count J)

    (returns the root node of a regression tree with at most J nodes)

    · Create tree node $RootNode$ amd let partition $P = \{(\mathcal{X}, RootNode)\}$

    · While $|P| < J$ and at least one $(S, Node) \in P$ has $|S| > 1$, repeat

      · Choose $(S, Node) \in P$ with $|S| > 1$ that has the largest error $E(S)$

        where $E(S = \{\mathbf{x}^t, y^t\}_{t=1}^N) = \sum_{t=1}^N [y^t - \sum_{i=1}^N y^i]^2$

      · For each input dimension $j = 1, \ldots, d$

        · Sort examples in S in ascending order of attribute x_j

        · For $k = 2, \ldots, |S|$

          · Compute split error $e_j^k = E(\{\mathbf{x}^t, y^t\}_{t=1}^{k-1}) + E(\{\mathbf{x}^t, y^t\}_{t=k}^{|S|})$

      · Select best split $(j', k') = \text{argmin}_{j,k}\{e_j^k\}$

      · Sort examples in S in ascending order of attribute $x_{j'}$.

      · Compute attribute threshold $r = (x_{j'}^{k'-1} + x_{j'}^{k'})/2$

      · Divide S into child subsets $S_L = \{(\mathbf{x}, y) \in S : x_{j'} \leq r\}$ and $S_R = S \setminus S_L$

      · Create tree nodes $Node_L, Node_R$

      · Finalize $Node$ as `NonleafNode(`$j', r, Node_L, Node_R$`)`

      · Update partition $P \leftarrow (P \setminus S) \cup \{(S_L, Node_L), (S_R, Node_R)\}$

    · For each $(S, Node) \in P$

      · Compute $\overline{y} = \text{mean}\{y : (\mathbf{x}, y) \in S\}$ and finalize $Node$ as `LeafNode(`$\overline{y}$`)`

    · `return` $RootNode$

- *Evaluation*
  - · Given input $\mathbf{x}$, call `EvaluateTree`$(RootNode, \mathbf{x})$
  - · `function EvaluateTree`$(Node, \mathbf{x})$

    · If $Node$ is a `LeafNode`(y) then `return` y

    · Otherwise, $Node$ is a `NonleafNode`$(j, r, Child_L, Child_R)$

    · If attribute $x_j \leq r$ then `return EvaluateTree`$(Child_L, \mathbf{x})$

    · Otherwise `return EvaluateTree`$(Child_R, \mathbf{x})$

Figure 5.2. The J-leaf Regression Tree algorithm

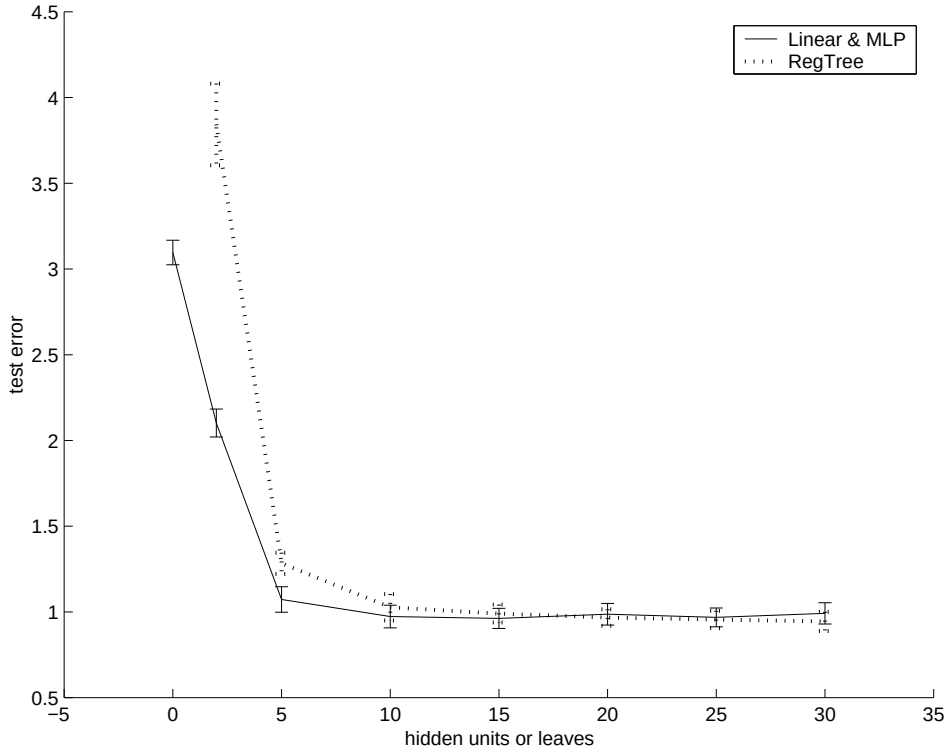


Figure 5.3. Base algorithm errors for `syndata`

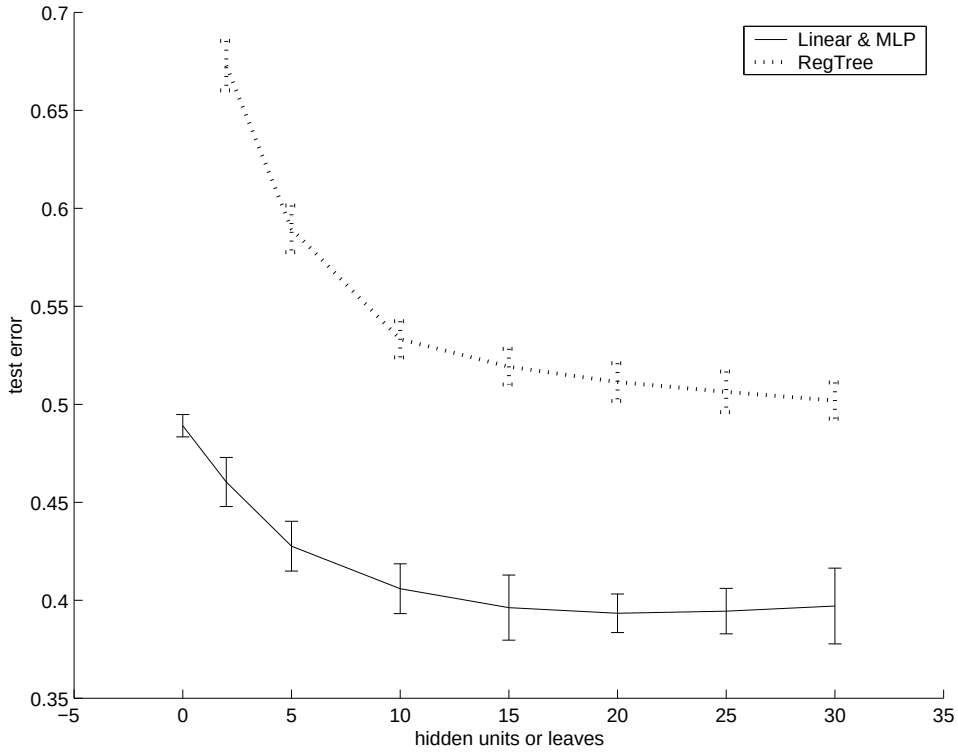


Figure 5.4. Base algorithm errors for `votes`

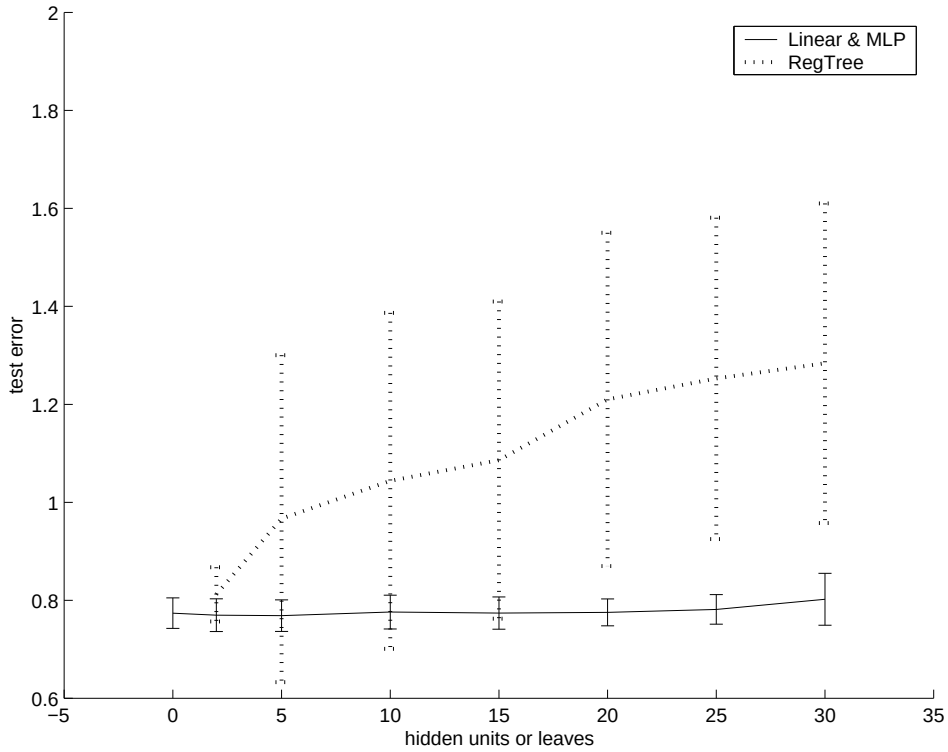


Figure 5.5. Base algorithm errors for `birth`

`birth` respectively, varying with hidden units or tree leaves. The full set of base algorithm error graphs for each dataset can be found in Appendix A.1.

Since `syndata` has single-dimensional input, it does not suffer from the curse of dimensionality as the others do. The regression tree is able to become sufficiently complex to rival MLP in its allowed range of leaves. For the MLP, 5 hidden units indicate saturation, and for larger numbers beginnings of overfitting are observable.

For `votes`, the behaviors are similar, only with MLP generalizing with significantly less error than the regression tree.

On the small `birth` dataset the regression tree clearly and inevitably overfits. As the number of leaves increase, it converges to a memorization of the training data and yields poor generalization on the test set. MLP suffers much less, possibly due to its continuous nature, while regression tree output is piecewise constant.

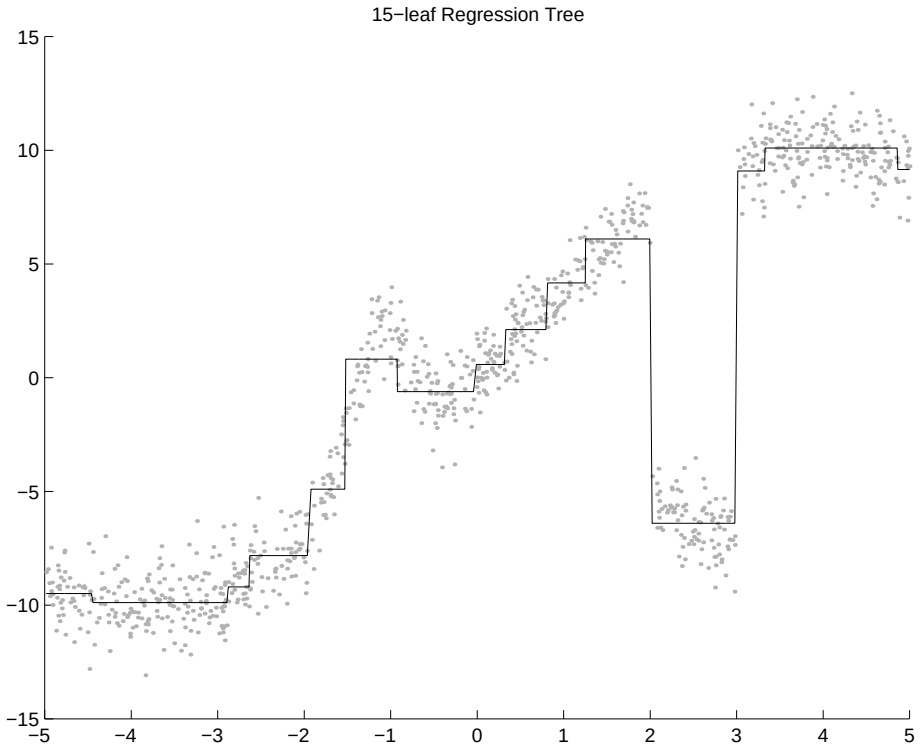


Figure 5.6. 15-leaf regression tree output on `syndata`

Example output plots on `syndata` are in Figures 5.6 and 5.7. The full set is in Appendix A.2.

5.3. Bagging and AdaBoost Results

Using J-leaf regression tree base models as described above, one parameter of each algorithm was inevitably the leaf count J of the tree. The same values as the base models above were used for J.

The other free parameter was the number of base learners. Values of $\{2, 5, 10, 15, 20\}$ were used.

Best-Ratio Bagging internally used 50 per cent of the original training examples for validation, and compared the ratios 10, 20, ..., 90 per cent of the remaining examples for sample size.

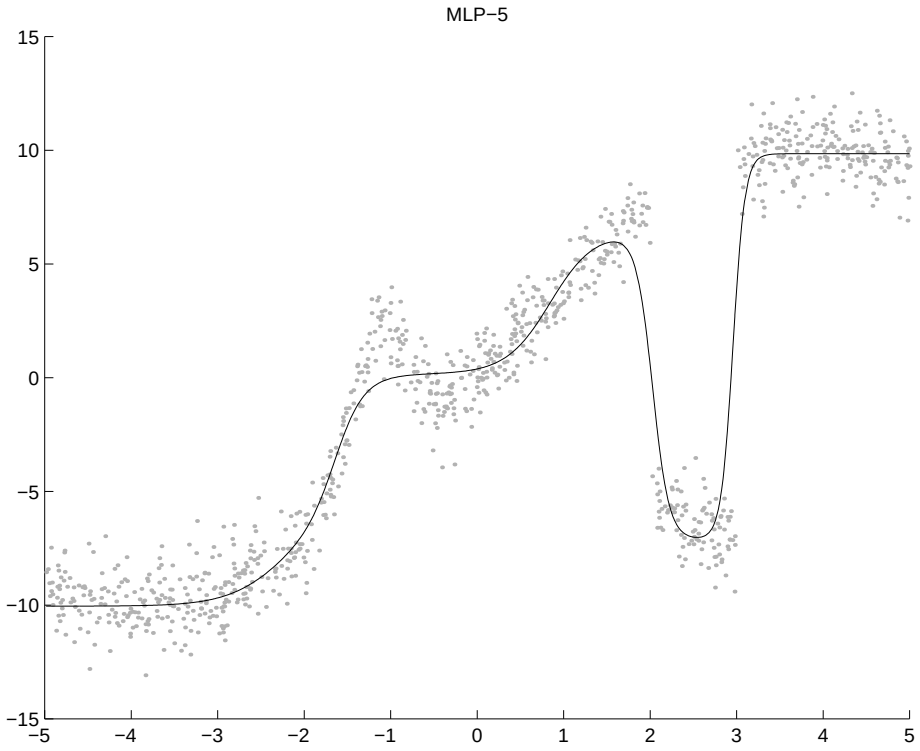


Figure 5.7. 5-hidden-unit MLP output on `syndata`

AdaBoost.R was tested only on `syndata`, `boston` and `calif1000`, where it yielded properly decreasing test errors like the Bagging methods. It showed no dangerous sensitivity to tree size, albeit demonstrating no significant advantage over Bagging to account for its complexity either.

Figures 5.8 and 5.9 show the test errors of Bagging, LS_Boost and Drucker.AD on `syndata` as the number of base models changes. These algorithms were chosen because they are prime examples of their respective categories. The unaggregated base algorithm RegTree is also included, plotted as constant. These figures show some characteristic behaviors, also observed on most of the other datasets. The plots for all datasets are in Appendix A.4.

Drucker's and Zemel & Pitassi's algorithms did not perform well with very small trees, increasing test error as more base models were added. Checking the training errors also revealed similarly increasing error, indicating that this is not due to overfitting by adding too many models, but the base models were too coarse to be useful

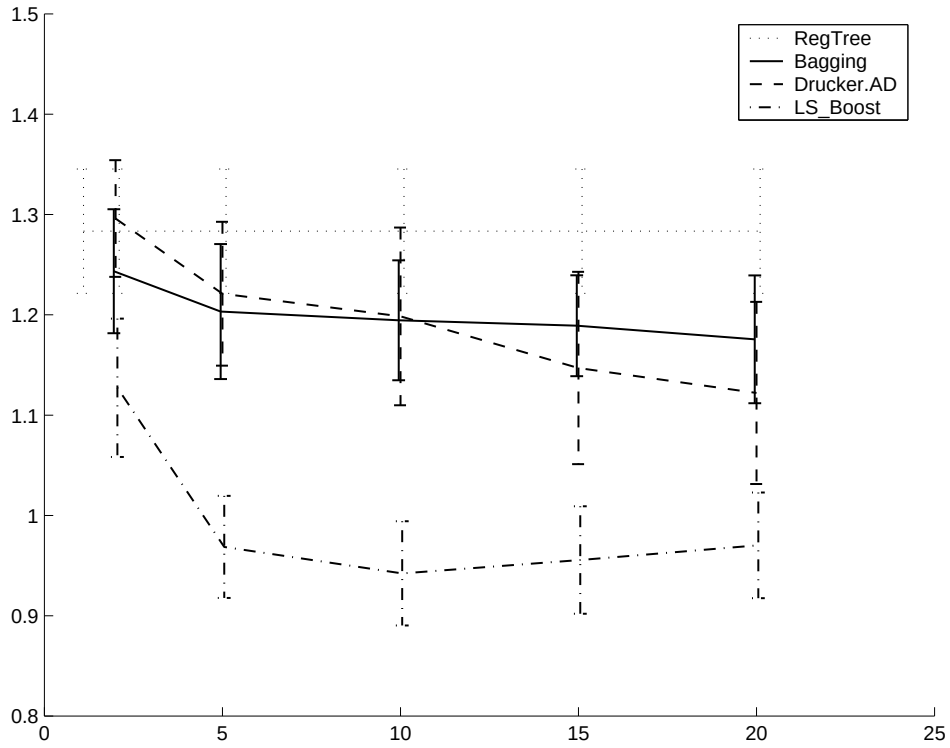


Figure 5.8. Bagging and AdaBoost errors on `syndata` using 5-leaf trees

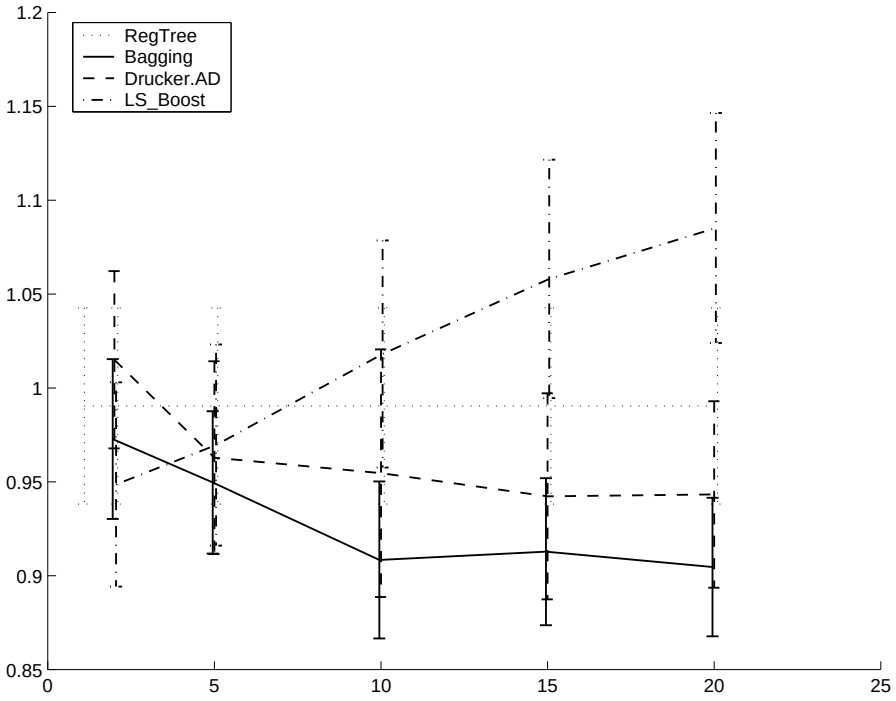


Figure 5.9. Bagging and AdaBoost errors on `syndata` using 15-leaf trees

to the distribution-based algorithms.

The relabeling algorithms LAD_Boost and LS_Boost gave the best results with small trees. They started overfitting at much smaller leaf counts than the unaggregated base algorithm, probably because their modification of target labels reduces the complexity of data. This is especially true of LAD_Boost which greatly simplifies the problem for the base learners by discretizing pseudo-targets to binary.

Bagging methods did not have problems with tree size. They used the trees at hand with consistent success through base model additions, although they needed a large number of large base models to catch up with the performance of the relabeling AdaBoost algorithms on small trees.

Our conjecture for the behavior of the CVA algorithm was validated by the experiment results. CVA was slightly better than the other Bagging algorithms using very few base models, and fell behind quickly thereafter as the cross-validated training set versions became increasingly similar. Bootstrapping proved to be the selection method of choice for aggregation.

The W-Bagging modification was not useful at all, almost always worse than Bagging. This is not surprising, since the bootstrap samples are selected uniform randomly, so any differences over validation examples must be purely accidental. Any "confidence" values thus derived are bound to disrupt the Bagging process.

Compared to a fixed 50 per cent ratio of sample size with Bagging, BR-Bagging did not show significant improvement despite the nine-fold execution time. Still it remains to be the only promising modification to Bagging among those we implemented, having a slight advantage to Bagging at times.

Figures 5.10, 5.11 and 5.12 show example outputs on `syndata` using 15-leaf regression trees as base models.

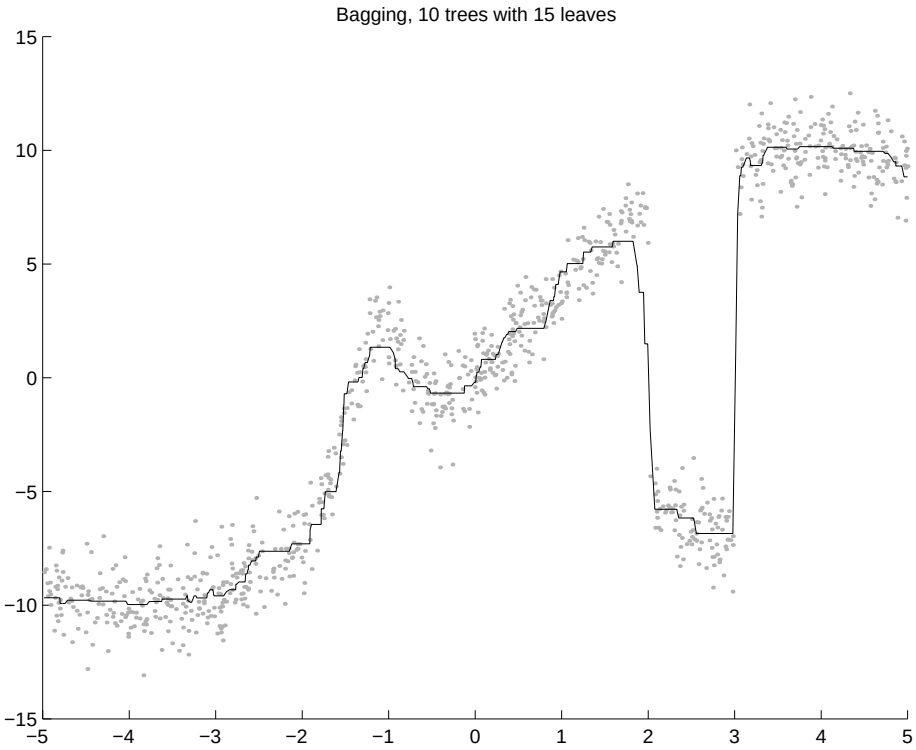


Figure 5.10. BAGGING output on `syndata` using 15-leaf trees

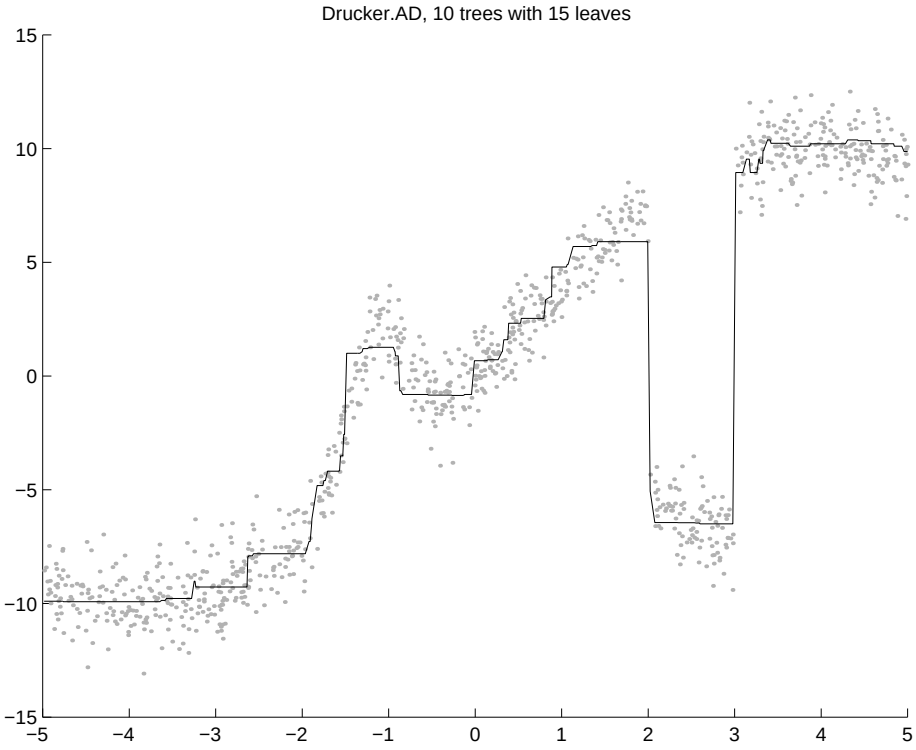


Figure 5.11. DRUCKER.AD output on `syndata` using 15-leaf trees

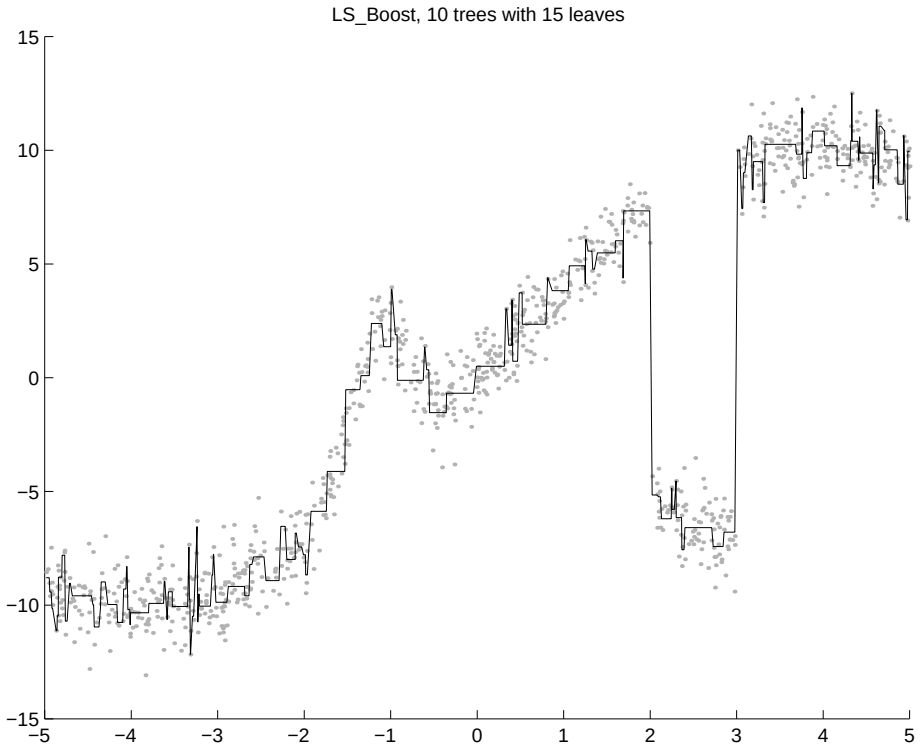


Figure 5.12. LS_Boost output on `syndata` using 15-leaf trees

5.4. Support Vector Machine Results

For the SVM implementation we used the LIBSVM package [24].

For ease of use and compatibility of parameters with the other algorithms, we used the ν-SVM variant for our experiments. Our kernel of choice was the radial kernel (Equation 3.15) because of its congruence to the Gaussian exceptions of REx. The parameter ν which roughly prescribes the ratio of support vectors, the capacity C, and the kernel spread parameter γ were manually set to suitable values for each dataset by trial and error.

Figures 5.13, 5.14 and 5.15 show Support Vector Machine outputs on `syndata` for different parameters. Support vectors are also marked on the graphs.

The performance of the Support Vector Machine algorithm was found to rely heavily on correctly choosing the hyperparameters C, ν, and the kernel parameter

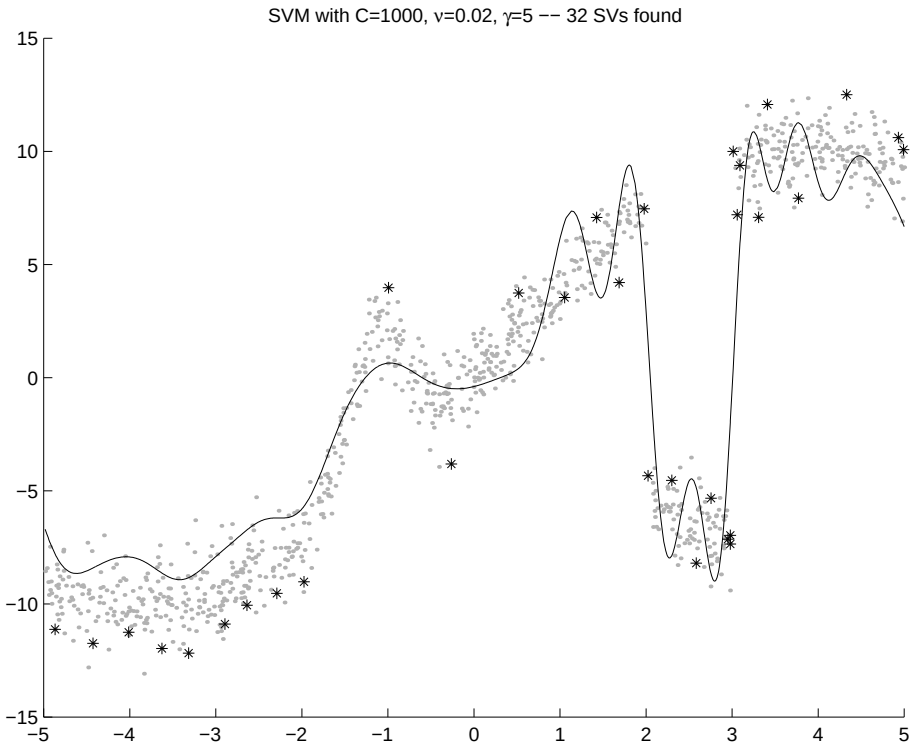


Figure 5.13. SVM output on `syndata` with $\nu = 0.02$ and $\gamma = 5$

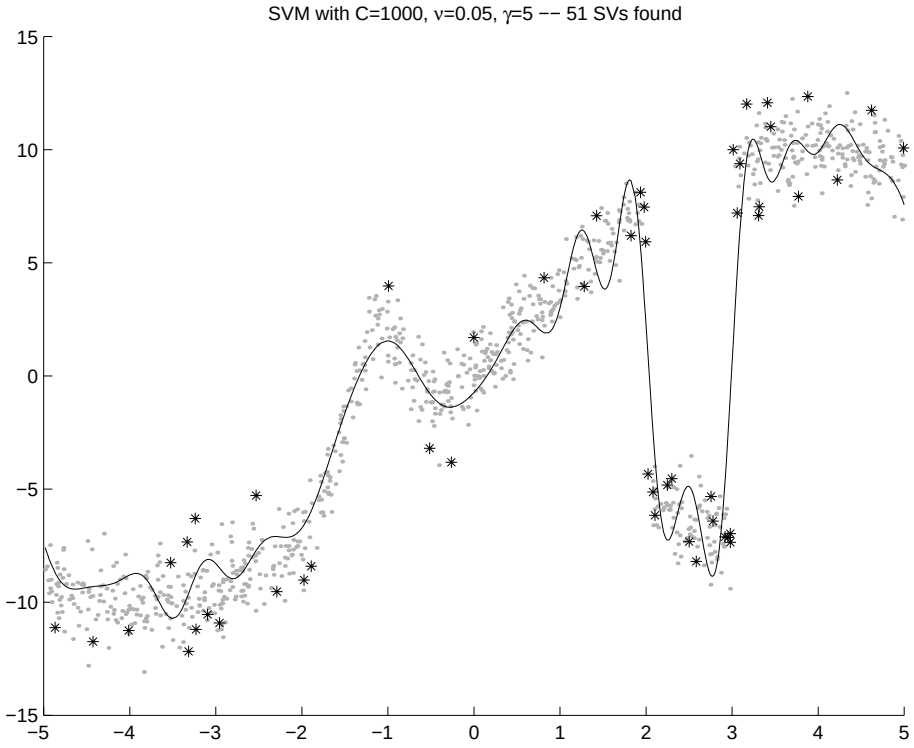


Figure 5.14. SVM output on `syndata` with $\nu = 0.05$ and $\gamma = 5$

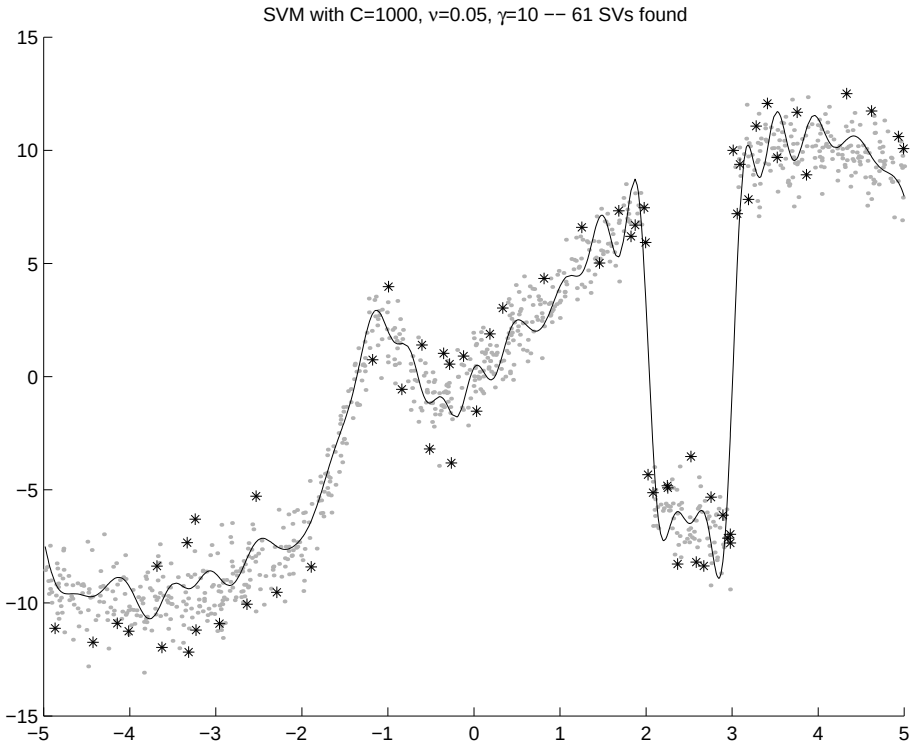


Figure 5.15. SVM output on `syndata` with $\nu = 0.05$ and $\gamma = 10$

γ. ν roughly controls the number of support vectors, so too small values yield too coarse grained regressors, and too high values result in too complex models that overfit. Similarly, large γ parameters make kernel radii too narrow, effecting many support vectors to cover the examples. And while searching for the right brew of ν and γ, C must be kept large enough to allow the desired learning.

Since we used no prior information about the data distribution and noise at hand, the parameters had to be tuned manually by tedious trial and error. Fortunately, when we ultimately did succeed in choosing suitable values, the SVM algorithm proved to at least as good as MLP on most datasets.

5.5. REx Results

Running REx without clustering produced too many exceptions, as expected. Since the initial variances were accordingly low, these Gaussians stayed as spikes from the rule, in effect memorizing the exceptions without consequence. See the example

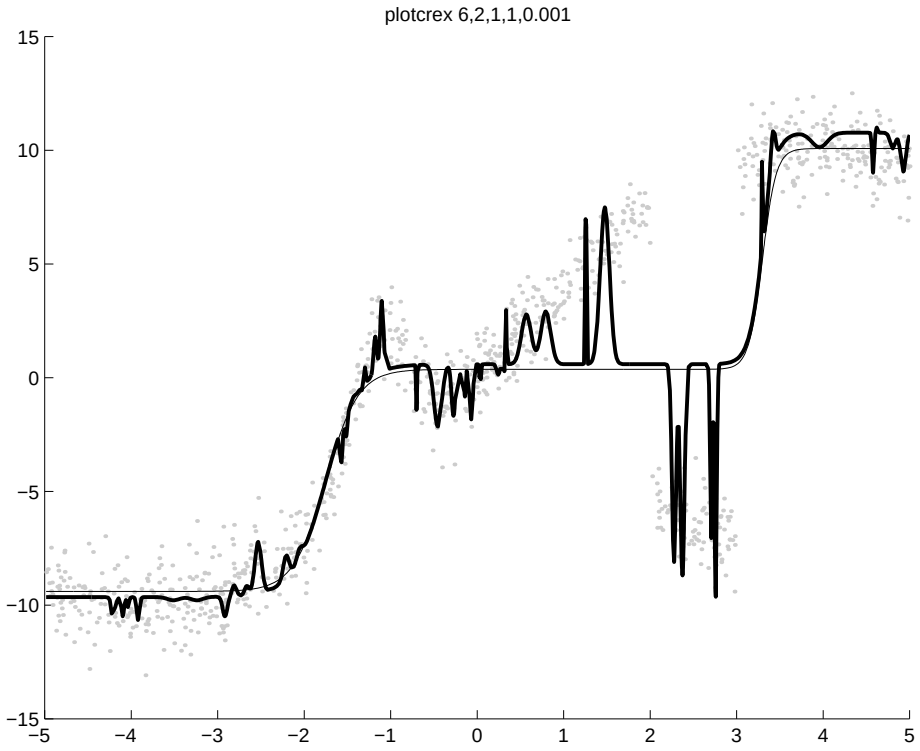


Figure 5.16. C-REx output on `syndata` with 2-hidden-unit MLP rule and $\varepsilon = 6$ without clustering

C-REx output on `syndata` in Figure 5.16 for an illustration, where the thin line is the 2-hidden-unit MLP rule. The rule is simpler than needed, but although a very large error threshold ($\varepsilon = 6$) was used, the exceptions do not serve to shape it in the right manner. The resulting function is far from being useful at all, despite a very high complexity.

Using clustering with ten means, Figures 5.17 and 5.18 are respectively the CREx and MREx outputs on `syndata` with a linear rule, and Figures 5.19 and 5.20 show the algorithms using 2-hidden-unit MLP rules. The thinner lines are the outputs of the rule, and the Gaussian curves indicate exception clusters in the input space. Compared to the previous outputs without clustering, the improvement is obvious.

Appendix A.3 includes a set of plots illustrating the effect of thresholds with and without clustering, for both C-REx and M-REx. The positive results of the smoothing effect seen in Figure 5.19 are verifiable from these graphs for most datasets.

In general, the average of clustered C-REx was at least as good as its rule alone on all datasets, except `syndata` where adding exceptions to an MLP was not as fruitful. With smaller MLPs than the optimal base model fit, the merit of C-REx was more apparent since the poorly fitting rule was twisted into shape by the exceptions. On the low-noise datasets `kin8fm` and `kin8nm` C-REx was able to satisfactorily improve its rule beyond the rule's best performance alone.

M-REx failed to provide a consistent improvement of its base rule on the average of 10 runs in any dataset.

Compared to C-REX, M-REx turned out to be solving a more difficult problem. Because of the steep softmax, its output is less smooth, one of the exceptions or the rule being locally dominant. Contrast the outputs on a linear rule in Figures 5.19 and 5.20 to observe the difference. C-REx produces a smooth stretching of the rule, while M-REx corrects it in clear-cut pieces. This sharper nature also causes the error surface of M-REx to be rougher, producing many deep local minima that the algorithm easily gets stuck in. The resulting test error variance is naturally higher than that of C-REx, indicating a less reliable algorithm. Also, the additional exponential computations because of softmax make the algorithm numerically capricious, prone to overflows and underflows. However, despite the difficulties in operation and poor experimental results, a future study of deriving knowledge from REx models is likely to find M-REx rather advantageous, since a partitioning into exception or rule regions is more understandable than a continuous mathematical combination.

Both C-REx and M-REx suffer to a degree from finetuning problems where Gaussian centers go outside the input region, variances spread to infinity or shrink to zero, and while trying to compensate for a tiny variance some combination weights soar and cause arbitrarily high spikes. Indicative of the algorithm's endeavor to dispose of redundant or poorly initialized exception centers, these problems are also common in Radial Basis Function networks, and known solutions are directly applicable to REx such as simple bounding or regularization methods such as weight decay. Educated initialization of the variances also helps considerably.

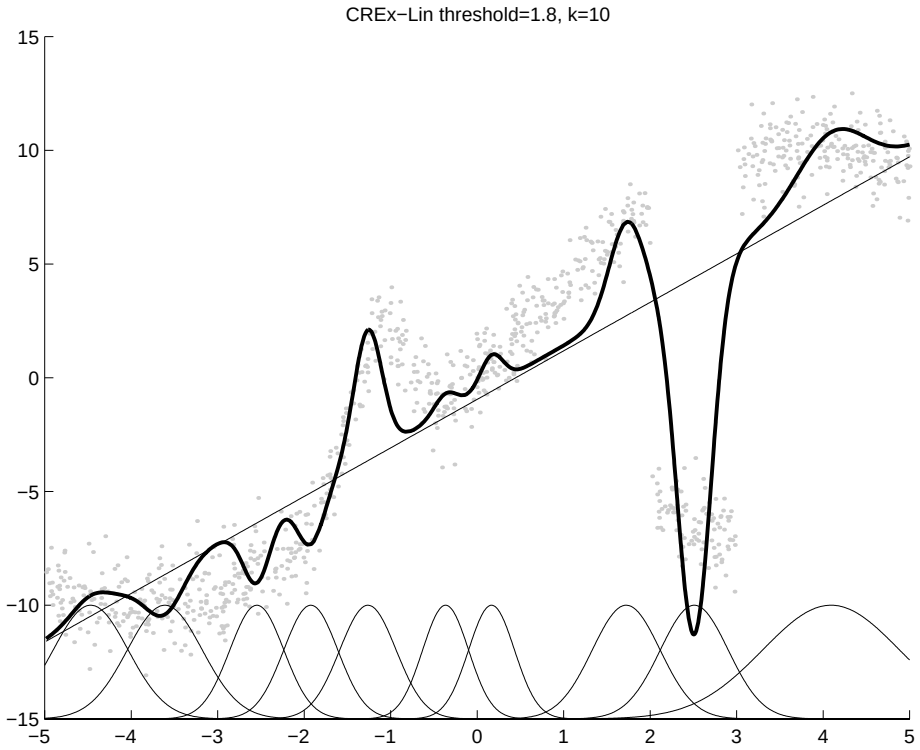


Figure 5.17. C-REx output on `syndata` with linear rule and $\varepsilon = 1.8$ with 10 clusters

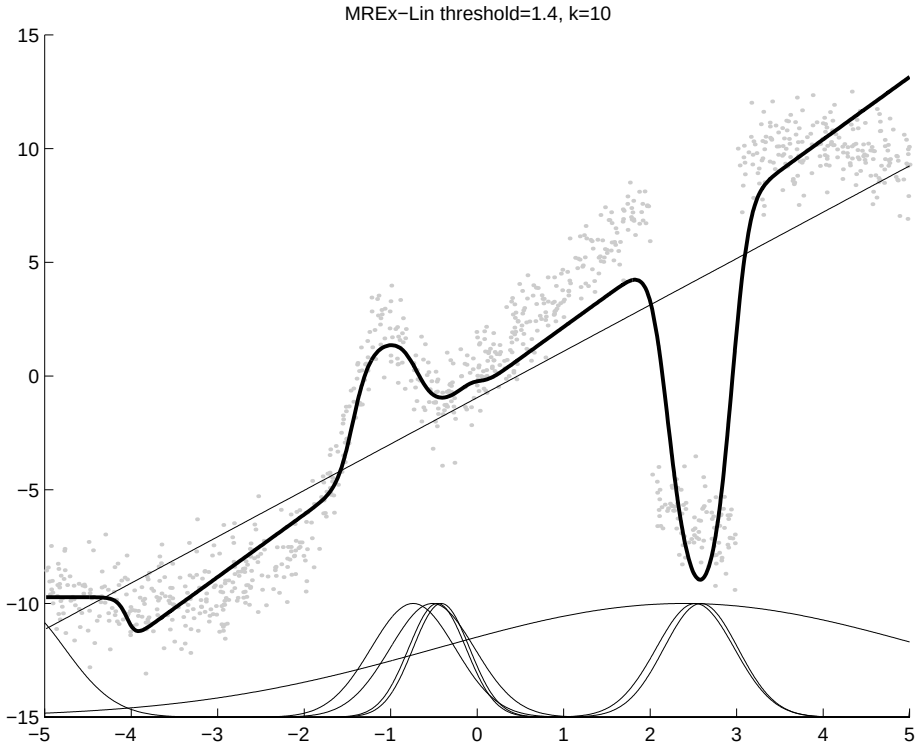


Figure 5.18. M-REx output on `syndata` with linear rule and $\varepsilon = 1.4$ with 10 clusters

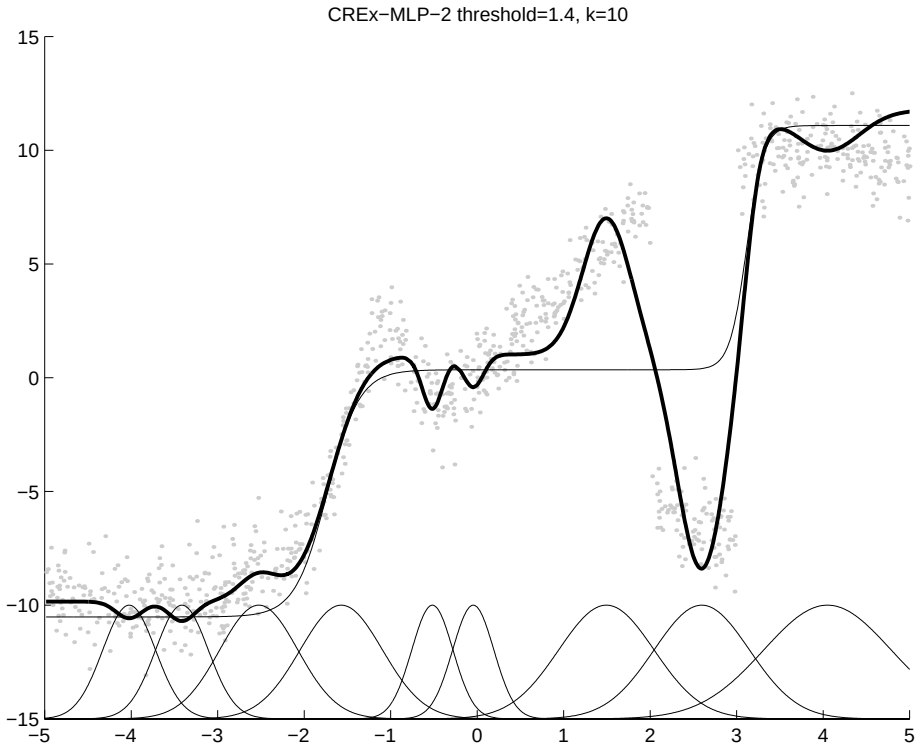


Figure 5.19. C-REx output on `syndata` with 2-hidden-unit MLP rule and $\varepsilon = 1.4$ with 10 clusters

5.6. Overall Comparison

Since our Bagging and AdaBoost algorithms use regression trees while REx uses perceptrons, comparisons based on overall accuracy results are not fair. However bagging or boosting MLPs would not illustrate the gradual enhancement of the simple base model as well as regression trees, and using linear models would be pointless since the combination would still be linear. Similarly, using regression trees for REx rules would not have emphasized the "global rule *vs.* local exception" paradigm as perceptrons have allowed. The objective of our experiments was not producing numerical benchmarks to prove final superiority of one algorithm to another, but to gain an insight to the particulars of each algorithm, leading to an understanding of when to utilize which one of them. Having said this, we present the best average errors per example for each algorithm on the datasets in Tables 5.2–5.23. The occasional lines dividing the tables show the range boundaries of Duncan's range test with 95 per cent confidence.

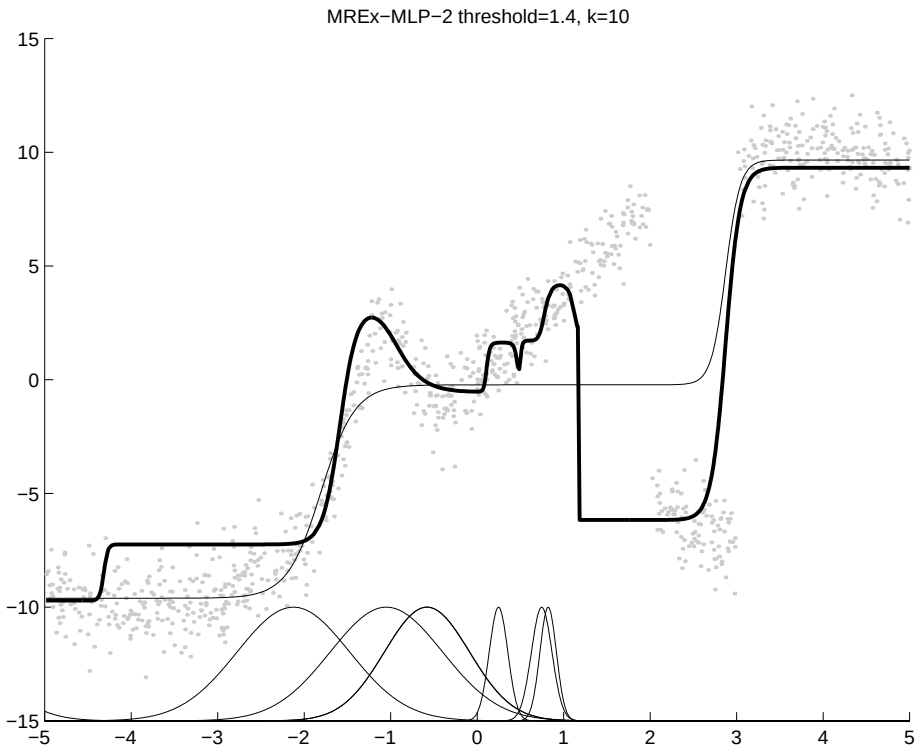


Figure 5.20. M-REx output on `syndata` with 2-hidden-unit MLP rule and $\varepsilon = 1.4$ with 10 clusters

Tables 5.24–5.45 show the algorithms compared using the 5×2-fold cross-validated F test [25] with 95 per cent confidence.

What is not seen in the tables is the improvement that the Bagging, AdaBoost and REx algorithms provide on relatively simple base models. If we have at our disposal simpler models than necessary to properly learn all data by themselves, these algorithms allow us significant improvements in accuracy. Although we may be able to train perceptrons or trees of arbitrary size, both are susceptible to overfitting with noisy data. Instead of trying to tune complexity with fear of overfitting, using safely small base models with one of these master algorithms may be less risky. Even in the extreme case where the base models are inevitably overfitting, they may be bagged to reduce variance and increase generalization performance.

On the whole, Bagging was observed to be a robust algorithm with respect to base model complexity, improving overly simple or overly complex base models equally

well. AdaBoost variants were superior as long as the base models were well suited, requiring very simple ones for the relabeling variants and more complex ones for the distribution-based family. Otherwise AdaBoost performance degraded beyond use.

The ratio parameter of Bagging was found to have negligible effect, 50 per cent being acceptable in all cases. Other than this and the base model complexity, Bagging and the AdaBoost algorithms had only the number of base models as the parameter, which is relatively intuitive and incrementally observable.

Support Vector Machines proved very capable, their flatness constraint providing inherent regularization against overfitting. However, compared to the other algorithms, they are rather difficult to use effectively, being a black box with many parameters to balance.

Of the two types of REx described, C-REx achieved satisfying results, especially with simple rules. Clustering proved to be indispensible, the exceptions being unacceptably many otherwise. M-REx produced inferior results than its rule alone on all datasets, demonstrating a need for amending its described problems.

Table 5.2. Errors of Bagging and AdaBoost on `syndata`

			avg	stdev
W-Bagging	leaves= 30	trees= 20	0.8870	0.0347
BR-Bagging	leaves= 30	trees= 15	0.8885	0.0339
Bagging	leaves= 25	trees= 20	0.8905	0.0434
Z&P.S	leaves= 30	trees= 5	0.9001	0.0452
CVA	leaves= 35	trees= 5	0.9033	0.0443
Z&P.AD	leaves= 20	trees= 5	0.9096	0.0394
Drucker.AD	leaves= 25	trees= 10	0.9167	0.0552
AdaBoost.R	leaves= 50	trees= 20	0.9187	0.0843
Drucker.S	leaves= 20	trees= 10	0.9211	0.0609
LS_Boost	leaves= 10	trees= 5	0.9337	0.0583
RegTree	leaves= 45		0.9426	0.0495
LAD_Boost	leaves= 5	trees= 20	0.9781	0.0504

Table 5.3. Errors of SVM and REx on `syndata`

			avg	stdev
SVM	$\nu = 0.42$	$C = 13$	0.9556	0.0273
MLP	hidden= 5		1.0724	0.0742
CREx-MLP	hidden= 15	$\varepsilon = 1.20$	1.2623	0.1116
MREx-MLP	hidden= 10	$\varepsilon = 1.20$	1.5678	0.3230
MREx-Lin		$\varepsilon = 0.80$	2.5088	0.3000
CREx-Lin		$\varepsilon = 1.00$	2.6837	0.1142
Linear			3.0964	0.0717

Table 5.4. Errors of Bagging and AdaBoost on `boston`

			avg	stdev
Drucker.AD	leaves= 20	trees= 20	0.2760	0.0193
Z&P.AD	leaves= 20	trees= 20	0.2803	0.0184
Z&P.S	leaves= 20	trees= 15	0.2864	0.0225
BR-Bagging	leaves= 20	trees= 15	0.2938	0.0262
Bagging	leaves= 20	trees= 20	0.2953	0.0156
W-Bagging	leaves= 20	trees= 20	0.2956	0.0191
Drucker.S	leaves= 20	trees= 10	0.2972	0.0136
AdaBoost.R	leaves= 45	trees= 20	0.3068	0.0278
CVA	leaves= 20	trees= 5	0.3075	0.0278
LS_Boost	leaves= 15	trees= 5	0.3346	0.0287
LAD_Boost	leaves= 2	trees= 20	0.3461	0.0252
RegTree	leaves= 20		0.3500	0.0229

Table 5.5. Errors of SVM and REx on `boston`

			avg	stdev
SVM	$\nu = 0.50$	$C = 100$	0.2549	0.0138
CREx-MLP	hidden= 5	$\varepsilon = 0.30$	0.2938	0.0248
MLP	hidden= 2		0.3053	0.0254
CREx-Lin		$\varepsilon = 0.40$	0.3634	0.0260
Linear			0.3774	0.0186
MREx-MLP	hidden= 10	$\varepsilon = 0.10$	0.7700	0.0629
MREx-Lin		$\varepsilon = 0.10$	0.8053	0.1300

Table 5.6. Errors of Bagging and AdaBoost on `calif1000`

			avg	stdev
Z&P.AD	leaves= 35	trees= 20	0.4250	0.0198
Drucker.AD	leaves= 35	trees= 20	0.4292	0.0150
W-Bagging	leaves= 30	trees= 20	0.4364	0.0194
BR-Bagging	leaves= 35	trees= 20	0.4436	0.0173
Bagging	leaves= 35	trees= 20	0.4452	0.0162
Z&P.S	leaves= 35	trees= 20	0.4469	0.0195
CVA	leaves= 35	trees= 10	0.4520	0.0181
LAD_Boost	leaves= 5	trees= 20	0.4551	0.0212
Drucker.S	leaves= 35	trees= 10	0.4573	0.0160
AdaBoost.R	leaves= 35	trees= 15	0.4644	0.0220
LS_Boost	leaves= 5	trees= 15	0.4683	0.0162
RegTree	leaves= 35		0.5060	0.0284

Table 5.7. Errors of SVM and REx on `calif1000`

			avg	stdev
CREx-MLP	hidden= 5	$\varepsilon = 1.00$	0.3891	0.0126
SVM	$\nu = 0.30$	$C = 100$	0.3989	0.0126
MLP	hidden= 2		0.4037	0.0249
CREx-Lin		$\varepsilon = 1.20$	0.4327	0.0095
Linear			0.4355	0.0095
MREx-MLP	hidden= 5	$\varepsilon = 1.20$	0.7535	0.0671
MREx-Lin		$\varepsilon = 1.00$	0.8833	0.1727

Table 5.8. Errors of Bagging and AdaBoost on `votes`

			avg	stdev
Drucker.AD	leaves= 50	trees= 20	0.4433	0.0063
Bagging	leaves= 50	trees= 20	0.4442	0.0055
BR-Bagging	leaves= 50	trees= 20	0.4447	0.0055
W-Bagging	leaves= 50	trees= 20	0.4464	0.0040
Z&P.AD	leaves= 50	trees= 20	0.4474	0.0041
Z&P.S	leaves= 50	trees= 20	0.4486	0.0060
Drucker.S	leaves= 50	trees= 15	0.4553	0.0051
CVA	leaves= 50	trees= 5	0.4577	0.0074
LAD_Boost	leaves= 10	trees= 20	0.4718	0.0121
LS_Boost	leaves= 5	trees= 20	0.4812	0.0139
RegTree	leaves= 50		0.4930	0.0127

Table 5.9. Errors of SVM and REx on `votes`

			avg	stdev
SVM	$\nu = 0.40$	$C = 30$	0.3939	0.0052
CREx-MLP	hidden= 20	$\varepsilon = 1.40$	0.4032	0.0078
MLP	hidden= 10		0.4059	0.0127
CREx-Lin		$\varepsilon = 1.40$	0.4754	0.0075
Linear			0.4891	0.0057
MREx-MLP	hidden= 20	$\varepsilon = 1.20$	0.8327	0.0532
MREx-Lin		$\varepsilon = 1.20$	0.8359	0.0520

Table 5.10. Errors of Bagging and AdaBoost on `prostate`

			avg	stdev
LAD_Boost	leaves= 2	trees= 2	0.5996	0.0226
W-Bagging	leaves= 35	trees= 20	0.6051	0.0442
CVA	leaves= 2	trees= 10	0.6270	0.0532
Z&P.S	leaves= 2	trees= 2	0.6307	0.0492
Bagging	leaves= 2	trees= 20	0.6350	0.0420
BR-Bagging	leaves= 5	trees= 20	0.6415	0.0345
Z&P.AD	leaves= 2	trees= 5	0.6495	0.0749
Drucker.AD	leaves= 2	trees= 2	0.6503	0.0506
RegTree	leaves= 2		0.6684	0.0723
Drucker.S	leaves= 15	trees= 20	0.6776	0.0387
LS_Boost	leaves= 5	trees= 5	0.6783	0.0531

Table 5.11. Errors of SVM and REx on `prostate`

			avg	stdev
SVM	$\nu = 0.30$	$C = 3$	0.5930	0.0355
CREx-MLP	hidden= 30	$\varepsilon = 1.20$	0.6111	0.0341
CREx-Lin		$\varepsilon = 1.80$	0.6117	0.0342
Linear			0.6119	0.0331
MLP	hidden= 2		0.6357	0.0364
MREx-Lin		$\varepsilon = 1.80$	0.7507	0.0526
MREx-MLP	hidden= 20	$\varepsilon = 1.20$	0.7600	0.0257

Table 5.12. Errors of Bagging and AdaBoost on `birth`

			avg	stdev
BR-Bagging	leaves= 5	trees= 15	0.7766	0.0290
Bagging	leaves= 2	trees= 15	0.7802	0.0328
W-Bagging	leaves= 5	trees= 20	0.7809	0.0306
LS_Boost	leaves= 10	trees= 5	0.7825	0.0281
CVA	leaves= 2	trees= 10	0.7848	0.0303
Z&P.S	leaves= 2	trees= 5	0.7897	0.0261
Z&P.AD	leaves= 2	trees= 5	0.7905	0.0257
Drucker.AD	leaves= 2	trees= 5	0.7911	0.0265
Drucker.S	leaves= 2	trees= 2	0.7917	0.0338
LAD_Boost	leaves= 10	trees= 10	0.7926	0.0279
RegTree	leaves= 2		0.8122	0.0554

Table 5.13. Errors of SVM and REx on `birth`

			avg	stdev
MLP	hidden= 2		0.7696	0.0335
CREx-MLP	hidden= 2	$\varepsilon = 1.60$	0.7706	0.0300
Linear			0.7738	0.0311
CREx-Lin		$\varepsilon = 1.80$	0.7767	0.0296
SVM	$\nu = 0.30$	$C = 10$	0.7840	0.0314
MREx-Lin		$\varepsilon = 1.80$	0.9220	0.1176
MREx-MLP	hidden= 2	$\varepsilon = 1.80$	0.9486	0.0955

Table 5.14. Errors of Bagging and AdaBoost on `abalone`

			avg	stdev
W-Bagging	leaves= 35	trees= 20	0.4832	0.0077
Z&P.AD	leaves= 50	trees= 10	0.4965	0.0115
LAD_Boost	leaves= 10	trees= 20	0.4970	0.0051
BR-Bagging	leaves= 20	trees= 20	0.5128	0.0112
Drucker.AD	leaves= 50	trees= 10	0.5143	0.0153
LS_Boost	leaves= 20	trees= 15	0.5201	0.0085
Bagging	leaves= 15	trees= 20	0.5209	0.0176
CVA	leaves= 15	trees= 5	0.5226	0.0150
Drucker.S	leaves= 25	trees= 2	0.5440	0.0293
Z&P.S	leaves= 10	trees= 2	0.5443	0.0387
RegTree	leaves= 10		0.5451	0.0049

Table 5.15. Errors of SVM and REx on `abalone`

			avg	stdev
CREx-MLP	hidden= 10	$\varepsilon = 3.00$	0.4555	0.0070
SVM	$\nu = 0.40$	$C = 4$	0.4594	0.0074
MLP	hidden= 2		0.4727	0.0089
CREx-Lin		$\varepsilon = 2.50$	0.5710	0.0556
MREx-Lin		$\varepsilon = 3.50$	1.0730	0.5807
Linear			1.3079	1.1231
MREx-MLP	hidden= 10	$\varepsilon = 3.00$	1.6330	1.2309

Table 5.16. Errors of Bagging and AdaBoost on `kin8fm`

			avg	stdev
Drucker.S	leaves= 100	trees= 20	0.2795	0.0061
Drucker.AD	leaves= 100	trees= 20	0.2878	0.0054
Z&P.AD	leaves= 100	trees= 20	0.2944	0.0049
Z&P.S	leaves= 100	trees= 20	0.2949	0.0068
LS_Boost	leaves= 10	trees= 20	0.2959	0.0106
BR-Bagging	leaves= 100	trees= 20	0.3140	0.0082
LAD_Boost	leaves= 10	trees= 20	0.3159	0.0117
Bagging	leaves= 100	trees= 20	0.3163	0.0059
W-Bagging	leaves= 100	trees= 20	0.3297	0.0068
CVA	leaves= 100	trees= 5	0.3583	0.0087
RegTree	leaves= 100		0.4389	0.0034

Table 5.17. Errors of SVM and REx on `kin8fm`

			avg	stdev
CREx-MLP	hidden= 15	$\varepsilon = 0.60$	0.1317	0.0016
SVM	$\nu = 0.10$	$C = 2$	0.1360	0.0012
MLP	hidden= 5		0.1385	0.0021
CREx-Lin		$\varepsilon = 0.70$	0.2138	0.0014
Linear			0.2162	0.0011
MREx-Lin		$\varepsilon = 0.50$	0.6273	0.0961
MREx-MLP	hidden= 5	$\varepsilon = 0.80$	0.6767	0.4367

Table 5.18. Errors of Bagging and AdaBoost on `kin8fh`

			avg	stdev
Drucker.S	leaves= 100	trees= 20	0.4384	0.0040
Drucker.AD	leaves= 100	trees= 20	0.4440	0.0055
Z&P.S	leaves= 100	trees= 20	0.4463	0.0051
Z&P.AD	leaves= 100	trees= 20	0.4463	0.0039
BR-Bagging	leaves= 100	trees= 20	0.4565	0.0073
Bagging	leaves= 100	trees= 20	0.4570	0.0055
W-Bagging	leaves= 100	trees= 20	0.4641	0.0055
LAD_Boost	leaves= 2	trees= 20	0.4807	0.0045
LS_Boost	leaves= 2	trees= 20	0.4852	0.0041
CVA	leaves= 100	trees= 10	0.4877	0.0082
RegTree	leaves= 100		0.5526	0.0054

Table 5.19. Errors of SVM and REx on `kin8fh`

			avg	stdev
CREx-MLP	hidden= 20	$\varepsilon = 2.00$	0.3939	0.0035
SVM	$\nu = 0.10$	$C = 2$	0.3997	0.0037
MLP	hidden= 5		0.4016	0.0102
CREx-Lin		$\varepsilon = 1.50$	0.4204	0.0033
Linear			0.4210	0.0031
MREx-MLP	hidden= 25	$\varepsilon = 1.00$	0.7131	0.0771
MREx-Lin		$\varepsilon = 1.00$	0.7516	0.0814

Table 5.20. Errors of Bagging and AdaBoost on `kin8nm`

			avg	stdev
Drucker.AD	leaves= 100	trees= 20	0.5021	0.0041
Z&P.AD	leaves= 100	trees= 20	0.5099	0.0051
W-Bagging	leaves= 100	trees= 20	0.5107	0.0066
Z&P.S	leaves= 100	trees= 10	0.5230	0.0040
Drucker.S	leaves= 100	trees= 20	0.5230	0.0069
BR-Bagging	leaves= 100	trees= 20	0.5230	0.0035
Bagging	leaves= 100	trees= 20	0.5232	0.0038
LS_Boost	leaves= 10	trees= 20	0.5283	0.0118
LAD_Boost	leaves= 10	trees= 20	0.5306	0.0087
CVA	leaves= 100	trees= 10	0.5457	0.0058
RegTree	leaves= 100		0.5954	0.0120

Table 5.21. Errors of SVM and REx on `kin8nm`

			avg	stdev
SVM	$\nu = 0.10$	$C = 5$	0.2406	0.0029
CREx-MLP	hidden= 30	$\varepsilon = 1.50$	0.2551	0.0104
MLP	hidden= 10		0.3010	0.0050
MREx-MLP	hidden= 20	$\varepsilon = 2.00$	0.5369	0.2862
CREx-Lin		$\varepsilon = 0.50$	0.5895	0.0063
Linear			0.6156	0.0048
MREx-Lin		$\varepsilon = 1.50$	0.7665	0.0359

Table 5.22. Errors of Bagging and AdaBoost on `kin8nh`

			avg	stdev
Drucker.AD	leaves= 100	trees= 20	0.5966	0.0113
W-Bagging	leaves= 100	trees= 20	0.5967	0.0070
BR-Bagging	leaves= 100	trees= 20	0.5996	0.0076
Drucker.S	leaves= 100	trees= 20	0.6002	0.0116
Z&P.AD	leaves= 100	trees= 20	0.6005	0.0095
Bagging	leaves= 100	trees= 20	0.6009	0.0066
Z&P.S	leaves= 100	trees= 20	0.6038	0.0121
CVA	leaves= 100	trees= 5	0.6154	0.0093
LS_Boost	leaves= 10	trees= 15	0.6243	0.0102
LAD_Boost	leaves= 10	trees= 20	0.6275	0.0107
RegTree	leaves= 60		0.6574	0.0105

Table 5.23. Errors of SVM and REx on `kin8nh`

			avg	stdev
CREx-MLP	hidden= 20	$\varepsilon = 2.00$	0.4906	0.0052
SVM	$\nu = 0.20$	$C = 5$	0.4922	0.0038
MLP	hidden= 10		0.5053	0.0096
CREx-Lin		$\varepsilon = 1.00$	0.6397	0.0068
Linear			0.6478	0.0076
MREx-MLP	hidden= 10	$\varepsilon = 1.50$	0.8310	0.0381
MREx-Lin		$\varepsilon = 2.00$	0.8866	0.0491

Table 5.24. 5×2cv F-test of Bagging and AdaBoost on `syndata`

	WBg	BRB	Bag	ZPS	CVA	ZPA	DrA	ABR	DrS	LSB	RT	LAD
W-Bagging		=	=	=	=	=	=	=	=	=	>	=
BR-Bagging	=		=	=	=	=	=	=	=	=	=	=
Bagging	=	=		=	=	=	=	=	=	=	=	=
Z&P.S	=	=	=		=	=	=	=	=	=	=	=
CVA	=	=	=	=		=	=	=	=	=	=	=
Z&P.AD	=	=	=	=	=		=	=	=	=	=	=
Drucker.AD	=	=	=	=	=	=		=	=	=	=	=
AdaBoost.R	=	=	=	=	=	=	=		=	=	=	=
Drucker.S	=	=	=	=	=	=	=	=		=	=	=
LS_Boost	=	=	=	=	=	=	=	=	=		=	=
RegTree	<	=	=	=	=	=	=	=	=	=		=
LAD_Boost	=	=	=	=	=	=	=	=	=	=	=	

Table 5.25. 5×2cv F-test of Bagging and AdaBoost on `boston`

	DrA	ZPA	ZPS	BRB	Bag	WBg	DrS	ABR	CVA	LSB	LAD	RT
Drucker.AD		=	=	=	=	=	=	>	=	>	>	>
Z&P.AD	=		=	=	=	=	=	>	=	>	>	>
Z&P.S	=	=		=	=	=	=	>	=	=	=	>
BR-Bagging	=	=	=		=	=	=	=	=	=	=	=
Bagging	=	=	=	=		=	=	=	=	=	=	>
W-Bagging	=	=	=	=	=		=	=	=	=	=	>
Drucker.S	=	=	=	=	=	=		=	=	=	=	=
AdaBoost.R	<	<	<	=	=	=	=		=	=	=	=
CVA	=	=	=	=	=	=	=	=		=	=	=
LS_Boost	<	<	=	=	=	=	=	=	=		=	=
LAD_Boost	<	<	=	=	=	=	=	=	=	=		=
RegTree	<	<	<	=	<	<	=	=	=	=	=	

Table 5.26. 5×2cv F-test of Bagging and AdaBoost on `calif1000`

	ZPA	DrA	WBg	BRB	Bag	ZPS	CVA	LAD	DrS	ABR	LSB	RT
Z&P.AD		=	=	=	>	>	=	=	=	>	=	>
Drucker.AD	=		=	=	>	=	=	=	=	=	=	>
W-Bagging	=	=		=	=	=	=	=	>	=	=	>
BR-Bagging	=	=	=		=	=	=	=	=	=	=	>
Bagging	<	<	=	=		=	=	=	=	=	=	>
Z&P.S	<	=	=	=	=		=	=	=	=	=	>
CVA	=	=	=	=	=	=		=	=	=	=	>
LAD_Boost	=	=	=	=	=	=	=		=	=	=	>
Drucker.S	=	=	<	=	=	=	=	=		=	=	=
AdaBoost.R	<	=	=	=	=	=	=	=	=		=	=
LS_Boost	=	=	=	=	=	=	=	=	=	=		=
RegTree	<	<	<	<	<	<	<	<	=	=	=	

Table 5.27. 5×2cv F-test of Bagging and AdaBoost on `votes`

	DrA	Bag	BRB	WBg	ZPA	ZPS	DrS	CVA	LAD	LSB	RT
Drucker.AD		>	>	>	=	>	>	>	>	>	>
Bagging	<		=	=	=	>	>	=	>	>	>
BR-Bagging	<	=		=	=	=	=	>	>	>	>
W-Bagging	<	=	=		=	=	>	>	>	>	>
Z&P.AD	=	=	=	=		=	=	=	>	>	>
Z&P.S	<	<	=	=	=		>	=	>	>	>
Drucker.S	<	<	=	<	=	<		=	>	=	>
CVA	<	=	<	<	=	=	=		=	>	>
LAD_Boost	<	<	<	<	<	<	<	=		=	=
LS_Boost	<	<	<	<	<	<	=	<	=		=
RegTree	<	<	<	<	<	<	<	<	=	=	

Table 5.28. 5×2cv F-test of Bagging and AdaBoost on `prostate`

	LAD	WBg	CVA	ZPS	Bag	BRB	ZPA	DrA	RT	DrS	LSB
LAD_Boost		=	=	=	=	=	=	=	=	>	=
W-Bagging	=		=	=	=	=	=	=	=	>	=
CVA	=	=		=	=	=	=	=	=	=	=
Z&P.S	=	=	=		=	=	=	=	=	=	=
Bagging	=	=	=	=		=	=	=	=	=	>
BR-Bagging	=	=	=	=	=		=	=	=	>	=
Z&P.AD	=	=	=	=	=	=		=	=	=	=
Drucker.AD	=	=	=	=	=	=	=		=	=	=
RegTree	=	=	=	=	=	=	=	=		=	=
Drucker.S	<	<	=	=	=	<	=	=	=		=
LS_Boost	=	=	=	=	<	=	=	=	=	=	

Table 5.29. 5×2cv F-test of Bagging and AdaBoost on `birth`

	BRB	Bag	WBg	LSB	CVA	ZPS	ZPA	DrA	DrS	LAD	RT
BR-Bagging		=	>	=	=	=	=	=	=	=	=
Bagging	=		=	=	=	=	=	=	=	=	=
W-Bagging	<	=		=	=	=	=	=	=	=	=
LS_Boost	=	=	=		=	=	=	=	=	=	=
CVA	=	=	=	=		=	=	=	=	=	=
Z&P.S	=	=	=	=	=		=	=	=	=	=
Z&P.AD	=	=	=	=	=	=		=	=	=	=
Drucker.AD	=	=	=	=	=	=	=		=	=	=
Drucker.S	=	=	=	=	=	=	=	=		=	=
LAD_Boost	=	=	=	=	=	=	=	=	=		=
RegTree	=	=	=	=	=	=	=	=	=	=	

Table 5.30. 5×2cv F-test of Bagging and AdaBoost on `abalone`

	WBg	ZPA	LAD	BRB	DrA	LSB	Bag	CVA	DrS	ZPS	RT
W-Bagging		>	=	>	=	>	=	>	>	=	>
Z&P.AD	<		=	=	=	>	=	>	=	=	>
LAD_Boost	=	=		=	=	>	=	>	=	=	>
BR-Bagging	<	=	=		=	=	=	=	=	=	>
Drucker.AD	=	=	=	=		=	=	=	=	=	=
LS_Boost	<	<	<	=	=		=	=	=	=	>
Bagging	=	=	=	=	=	=		=	=	=	=
CVA	<	<	<	=	=	=	=		=	=	>
Drucker.S	<	=	=	=	=	=	=	=		=	=
Z&P.S	=	=	=	=	=	=	=	=	=		=
RegTree	<	<	<	<	=	<	=	<	=	=	

Table 5.31. 5×2cv F-test of Bagging and AdaBoost on `kin8fm`

	DrS	DrA	ZPA	ZPS	LSB	BRB	LAD	Bag	WBg	CVA	RT
Drucker.S		>	>	>	=	>	=	>	>	>	>
Drucker.AD	<		>	=	=	>	=	>	>	>	>
Z&P.AD	<	<		=	=	>	=	>	>	>	>
Z&P.S	<	=	=		=	>	=	>	>	>	>
LS_Boost	=	=	=	=		=	>	=	=	>	>
BR-Bagging	<	<	<	<	=		=	=	>	>	>
LAD_Boost	=	=	=	=	<	=		=	=	>	>
Bagging	<	<	<	<	=	=	=		>	>	>
W-Bagging	<	<	<	<	=	<	=	<		>	>
CVA	<	<	<	<	<	<	<	<	<		>
RegTree	<	<	<	<	<	<	<	<	<	<	

Table 5.32. 5×2cv F-test of Bagging and AdaBoost on `kin8fh`

	DrS	DrA	ZPS	ZPA	BRB	Bag	WBg	LAD	LSB	CVA	RT
Drucker.S		=	>	>	>	>	>	>	>	>	>
Drucker.AD	=		=	=	=	>	>	>	>	>	>
Z&P.S	<	=		=	>	>	>	>	>	>	>
Z&P.AD	<	=	=		=	>	>	>	>	>	>
BR-Bagging	<	=	<	=		=	=	>	>	>	>
Bagging	<	<	<	<	=		>	>	>	>	>
W-Bagging	<	<	<	<	=	<		>	>	>	>
LAD_Boost	<	<	<	<	<	<	<		=	=	>
LS_Boost	<	<	<	<	<	<	<	=		=	>
CVA	<	<	<	<	<	<	<	=	=		>
RegTree	<	<	<	<	<	<	<	<	<	<	

Table 5.33. 5×2cv F-test of Bagging and AdaBoost on `kin8nm`

	DrA	ZPA	WBg	ZPS	DrS	BRB	Bag	LSB	LAD	CVA	RT
Drucker.AD		=	=	>	>	>	>	=	>	>	>
Z&P.AD	=		=	=	=	=	>	=	>	>	>
W-Bagging	=	=		>	=	=	=	=	=	>	>
Z&P.S	<	=	<		=	=	=	=	=	>	>
Drucker.S	<	=	=	=		=	=	=	=	>	>
BR-Bagging	<	=	=	=	=		=	=	=	>	>
Bagging	<	<	=	=	=	=		=	=	>	>
LS_Boost	=	=	=	=	=	=	=		=	>	>
LAD_Boost	<	<	=	=	=	=	=	=		=	>
CVA	<	<	<	<	<	<	<	<	=		>
RegTree	<	<	<	<	<	<	<	<	<	<	

Table 5.34. 5×2cv F-test of Bagging and AdaBoost on `kin8nh`

	DrA	WBg	BRB	DrS	ZPA	Bag	ZPS	CVA	LSB	LAD	RT
Drucker.AD		=	=	=	=	=	=	=	>	=	>
W-Bagging	=		=	=	=	=	=	>	>	>	>
BR-Bagging	=	=		=	=	=	=	>	>	>	>
Drucker.S	=	=	=		=	=	=	=	>	=	>
Z&P.AD	=	=	=	=		=	=	>	>	>	>
Bagging	=	=	=	=	=		=	=	>	=	>
Z&P.S	=	=	=	=	=	=		=	>	=	>
CVA	=	<	<	=	<	=	=		=	=	>
LS_Boost	<	<	<	<	<	<	<	=		=	>
LAD_Boost	=	<	<	=	<	=	=	=	=		>
RegTree	<	<	<	<	<	<	<	<	<	<	

Table 5.35. 5×2cv F-test of SVM and REx on `syndata`

	SVM	MLP	CRM	MRM	MRL	CRL	Lin
SVM		=	>	>	>	>	>
MLP	=		=	>	>	>	>
CREx-MLP	<	=		>	>	>	>
MREx-MLP	<	<	<		>	>	>
MREx-Lin	<	<	<	<		=	=
CREx-Lin	<	<	<	<	=		>
Linear	<	<	<	<	=	<	

Table 5.36. 5×2cv F-test of SVM and REx on `boston`

	SVM	CRM	MLP	CRL	Lin	MRM	MRL
SVM		=	=	>	>	>	>
CREx-MLP	=		=	=	>	>	=
MLP	=	=		=	>	>	=
CREx-Lin	<	=	=		=	>	=
Linear	<	<	<	=		>	=
MREx-MLP	<	<	<	<	<		=
MREx-Lin	<	=	=	=	=	=	

Table 5.37. 5×2cv F-test of SVM and REx on `calif1000`

	CRM	SVM	MLP	CRL	Lin	MRM	MRL
CREx-MLP		=	=	>	>	>	=
SVM	=		=	>	>	>	=
MLP	=	=		=	=	>	=
CREx-Lin	<	<	=		=	>	=
Linear	<	<	=	=		>	=
MREx-MLP	<	<	<	<	<		=
MREx-Lin	=	=	=	=	=	=	

Table 5.38. 5×2cv F-test of SVM and REx on `votes`

	SVM	CRM	MLP	CRL	Lin	MRM	MRL
SVM		=	=	>	>	>	>
CREx-MLP	=		=	>	>	>	>
MLP	=	=		>	>	>	>
CREx-Lin	<	<	<		>	>	>
Linear	<	<	<	<		>	>
MREx-MLP	<	<	<	<	<		=
MREx-Lin	<	<	<	<	<	=	

Table 5.39. 5×2cv F-test of SVM and REx on `prostate`

	SVM	CRM	CRL	Lin	MLP	MRL	MRM
SVM		=	>	>	=	>	>
CREx-MLP	=		=	=	=	>	>
CREx-Lin	<	=		=	=	>	>
Linear	<	=	=		=	>	>
MLP	=	=	=	=		>	>
MREx-Lin	<	<	<	<	<		=
MREx-MLP	<	<	<	<	<	=	

Table 5.40. 5×2cv F-test of SVM and REx on `birth`

	MLP	CRM	Lin	CRL	SVM	MRL	MRM
MLP		=	=	=	=	=	=
CREx-MLP	=		=	=	=	=	=
Linear	=	=		=	=	=	=
CREx-Lin	=	=	=		=	=	=
SVM	=	=	=	=		=	=
MREx-Lin	=	=	=	=	=		=
MREx-MLP	=	=	=	=	=	=	

Table 5.41. 5×2cv F-test of SVM and REx on `abalone`

	CRM	SVM	MLP	CRL	MRL	Lin	MRM
CREx-MLP		=	=	>	=	=	=
SVM	=		=	>	=	=	=
MLP	=	=		>	=	=	=
CREx-Lin	<	<	<		=	=	=
MREx-Lin	=	=	=	=		=	=
Linear	=	=	=	=	=		=
MREx-MLP	=	=	=	=	=	=	

Table 5.42. 5×2cv F-test of SVM and REx on `kin8fm`

	CRM	SVM	MLP	CRL	Lin	MRL	MRM
CREx-MLP		>	>	>	>	>	=
SVM	<		=	>	>	>	=
MLP	<	=		>	>	>	=
CREx-Lin	<	<	<		>	>	=
Linear	<	<	<	<		>	=
MREx-Lin	<	<	<	<	<		=
MREx-MLP	=	=	=	=	=	=	

Table 5.43. 5×2cv F-test of SVM and REx on `kin8fh`

	CRM	SVM	MLP	CRL	Lin	MRM	MRL
CREx-MLP		>	=	>	>	>	>
SVM	<		=	>	>	>	>
MLP	=	=		=	=	>	>
CREx-Lin	<	<	=		>	>	>
Linear	<	<	=	<		>	>
MREx-MLP	<	<	<	<	<		=
MREx-Lin	<	<	<	<	<	=	

Table 5.44. 5×2cv F-test of SVM and REx on `kin8nm`

	SVM	CRM	MLP	MRM	CRL	Lin	MRL
SVM		=	>	=	>	>	>
CREx-MLP	=		>	=	>	>	>
MLP	<	<		=	>	>	>
MREx-MLP	=	=	=		=	=	=
CREx-Lin	<	<	<	=		>	>
Linear	<	<	<	=	<		>
MREx-Lin	<	<	<	=	<	<	

Table 5.45. 5×2cv F-test of SVM and REx on `kin8nh`

	CRM	SVM	MLP	CRL	Lin	MRM	MRL
CREx-MLP		=	>	>	>	>	>
SVM	=		>	>	>	>	>
MLP	<	<		>	>	>	>
CREx-Lin	<	<	<		>	>	>
Linear	<	<	<	<		>	>
MREx-MLP	<	<	<	<	<		=
MREx-Lin	<	<	<	<	<	=	

Table 5.46. Time complexities of evaluation

	Parameters	Complexity
Linear		D
MLP	M h.u.	$(D+3)M$
CREx-Lin	K exc.	$(D+3)K+D$
CREx-MLP	M h.u., K exc.	$(D+3)M+(D+2)K$
MREx-Lin	K exc.	$(D+3)K+2D$
MREx-MLP	M h.u., K exc.	$(D+3)M+(D+2)K+D$
SVM	L s.v.	$(D+2)L$
RegTree	J leaves	1
Mean models	K models	K
W.Median models	K models	$2K$

5.7. Complexity Analysis

Table 5.46 shows the time complexities of the produced models for predicting the output for a single example. Multiplications, divisions and exponentials were counted, ignoring additive operations. For the regression-tree based models the balanced tree case of $\log_2 J$ comparisons were approximated as a single multiplication, and similarly the additions and comparative operations in computing the weighted median were counted as single multiplicative operation.

Although the complexity expression of the SVM model may appear smaller than the C-REx models, in practice the number of support vectors is much larger than the number of REx exceptions. This is primarily because the SVM must use many local models to construct the rule if it is to use local models for the exceptions. In addition to the really difficult examples, many others have to be added as the typical ones. Since REx is able to use a compact global rule, its exceptions are much fewer than the support vectors of the SVM, on the order or five to five hundred times on our datasets.

Appendix A.5 contains the plots of test errors against time complexity of evalua-

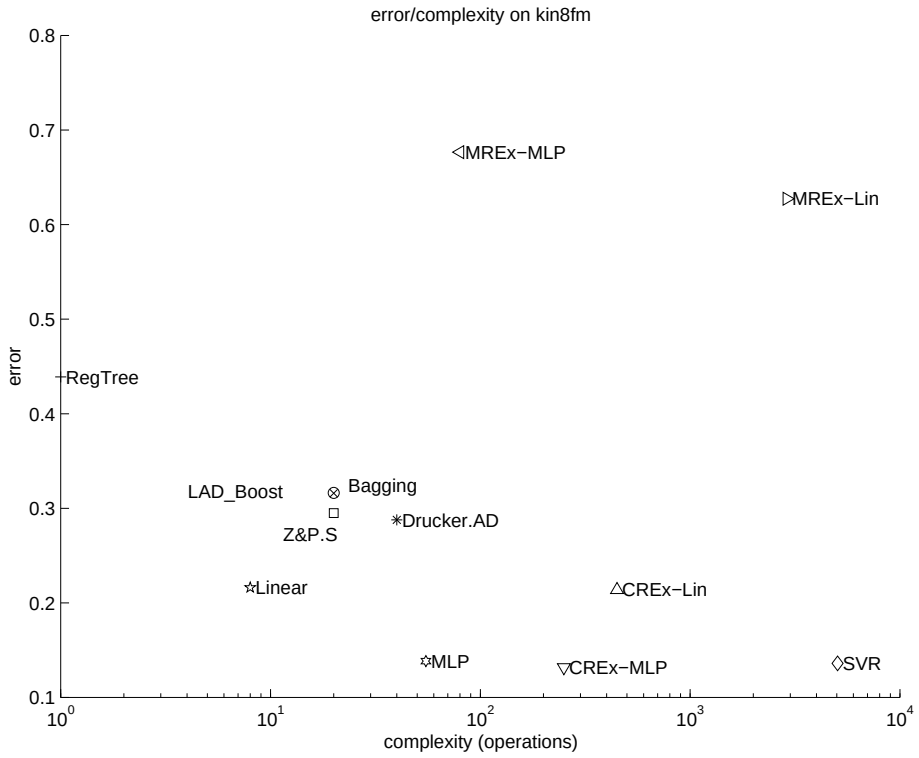


Figure 5.21. Error and complexity on `kin8fm`

tion, for C-REx and M-REx with their base models, and also seperately for all families of algorithms to illustrate the tradeoff between the error and time complexity. An example is in Figure 5.21. Note that the complexity axis is plotted in logarithmic scale.

6. CONCLUSIONS AND FUTURE WORK

The C-REx version of the proposed REx algorithm was shown to be a successful machine learning algorithm that produces an intuitive and accurate model by combining a rule and a set of exceptions. Simple rules are enhanced in necessary regions by using locally active exceptions. If the data is very simple in most of the input space but misbehaves in some parts, REx can handle most data with a compact simple rule and concentrate further only on the anomalous part. Moreover, this structure of the data is trivially observable from the REx model, as to where the rule acts and what the exceptions are. The number of exceptions is easily bounded by clustering.

The common trait binding AdaBoost, SVM and REx is that they all exercise selective attention on the training set. They implicitly or explicitly differentiate between training examples based on a measure of importance. Importance is often in parallel with the difficulty of learning the example, in other words the degree of change needed in the model to include that example. The differences lie in how the algorithms assess and exploit importance.

Our expectation from machine learning algorithms is not always sheer accuracy only. Most of the time, better understanding of the data is valuable, and our importance-guided models allow us a peek into the otherwise black box. Other methods like MLP or Bagging, successful as they may be, have no such interest in evaluating the data, and reveal no structural information.

AdaBoost iteratively reweights or relabels examples while adding base models, so that the difficult examples get a higher weight or label magnitude than others through the iterations. The easy examples are those that have already been learned, those that are already predictable with the current model, so their weights or labels are near zero. As the importance level of an example increases, the algorithm tries harder to learn that example than others. This information is only used during training and not explicitly stored, but the each new base model implicitly reflects the importance scores

used while adding it.

The Support Vector Machine algorithm directly stores some examples as support vectors and explicitly constrains its model by them such that when these "important" examples are correctly predicted, all the other "easy" ones are correct within acceptable accuracy. Compared to AdaBoost's continuous weights or labels, SVM's measure of importance is polarized, as being a support vector or not. The α values provide some ordering among the support vectors though, especially by separating as bounded or unbounded. All support vectors are "important", but the unbounded are the really "difficult" ones.

REx makes the clearest distinction by explicitly treating some examples as "exceptions". These are the difficult examples, the easy ones being explained by the rule. REx fixes exceptions to be local factors. This makes intuitive sense because if an exception has a global effect, then it is not an exception but part of the rule. It also provides interpretability since thinking in terms of local modifications to a general rule is natural to the human mind.

Among the algorithms described, REx also happens to be the only one that can modify a linear rule into a nonlinear model. The linear model is a very simple, efficient, understandable, and nonparametric model which has a non-iterative analytical solution as well as trivial differentiability. A solution with such an elegant rule and local exceptions may be ideal for many problems, a possibility only provided by REx among these algorithms.

M-REx provides sharper partitioning of data into rule and exception regions, but obviously requires improvement. Its deficiencies can be more closely examined on different visualizable datasets and mended accordingly.

The exception consolidation step in REx can be implemented in many other ways than K-means clustering. Other known clustering methods can be applied and tested for improvement. The initialization of variances can be made smarter, possibly

depending on automated distribution tests.

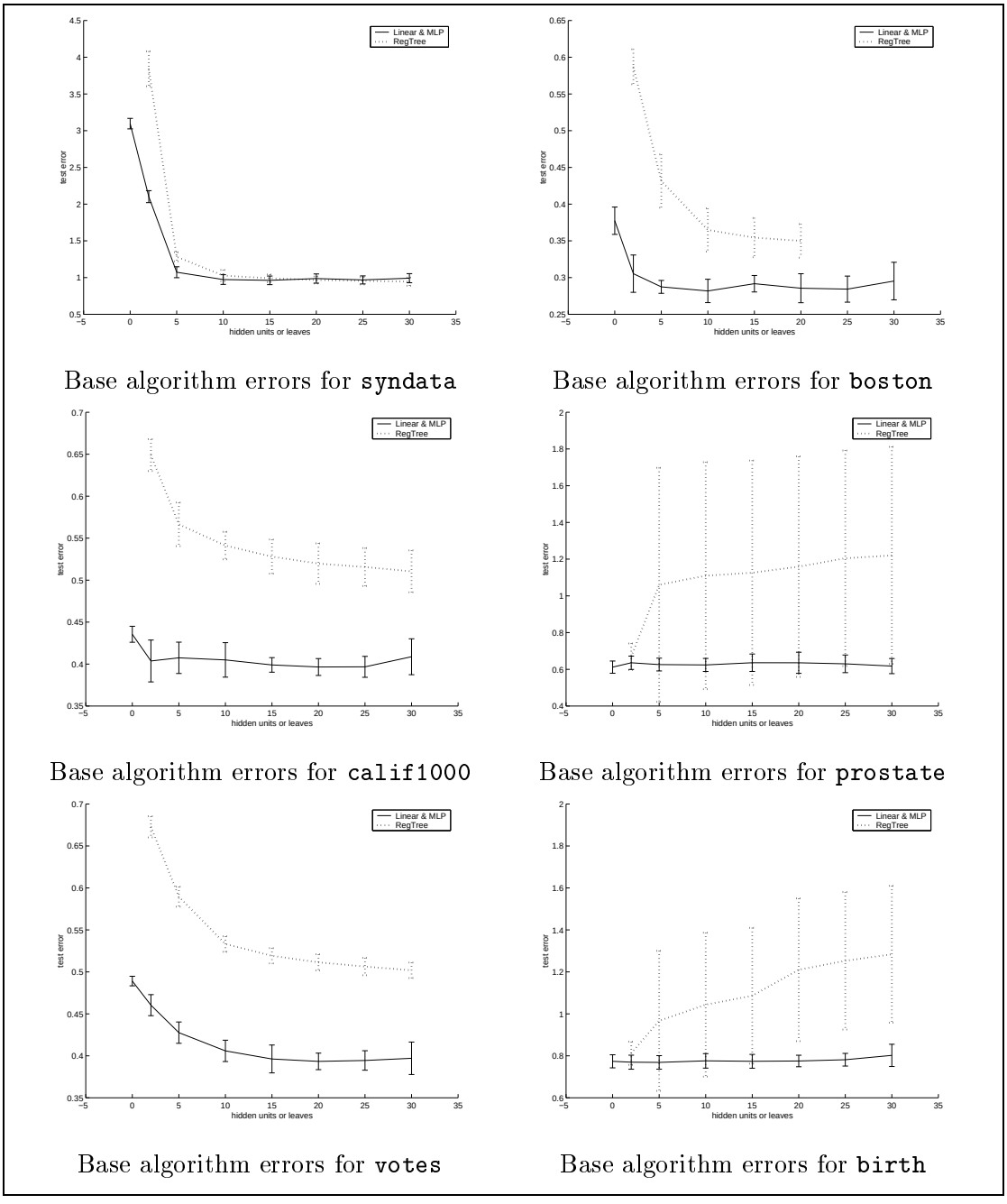
By clustering, REx also achieves precise control of model complexity, in contrast to the asymptotic guidance in ν-SVM.

An extension of the AdaBoost-SVM-REx comparison should investigate precisely which examples the different methods emphasize. The degree of similarity between AdaBoost's highest weighted examples, support vectors and REx exceptions may shed further light on the similarities and differences between the algorithms.

To level the differences between types and complexities of base models, utility measures can be defined to balance model complexity and overall accuracy. These utility results can also be extended for parallelism, distinguishing parallelizable and sequential algorithm steps.

APPENDIX A: EXTRA FIGURES

A.1. Base Algorithm Errors



Base algorithm errors for `syndata`

Base algorithm errors for `boston`

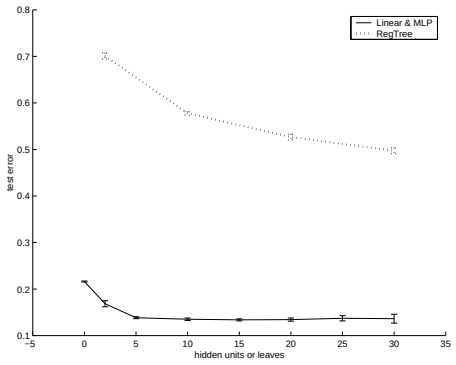
Base algorithm errors for `calif1000`

Base algorithm errors for `prostate`

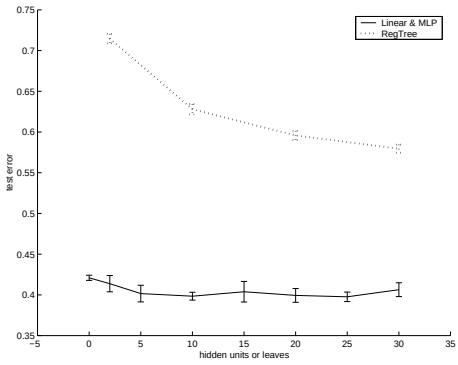
Base algorithm errors for `votes`

Base algorithm errors for `birth`

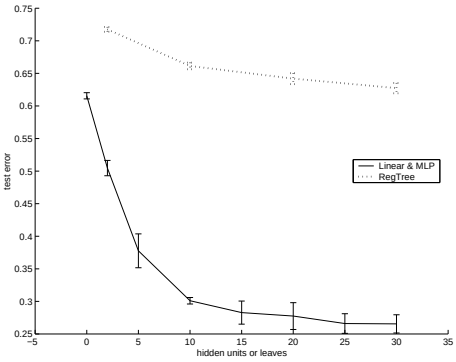
Figure A.1. Base algorithm errors



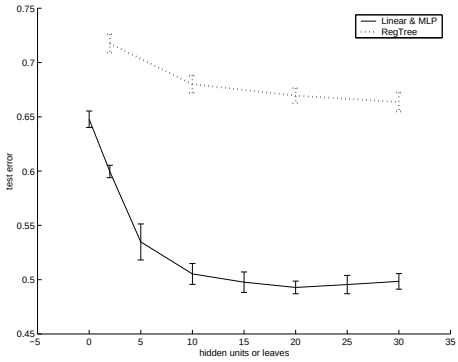
Base algorithm errors for `kin8fm`



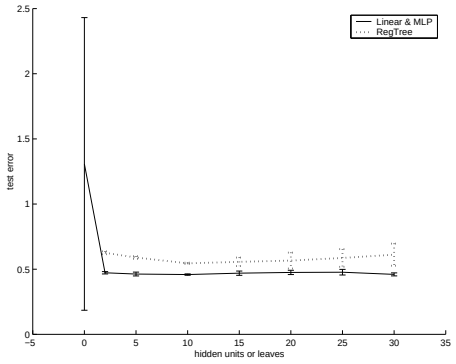
Base algorithm errors for `kin8fh`



Base algorithm errors for `kin8nm`



Base algorithm errors for `kin8nh`



Base algorithm errors for `abalone`

Figure A.2. Base algorithm errors (continued)

A.2. Outputs on `syndata`

A.2.1. Base Models on `syndata`

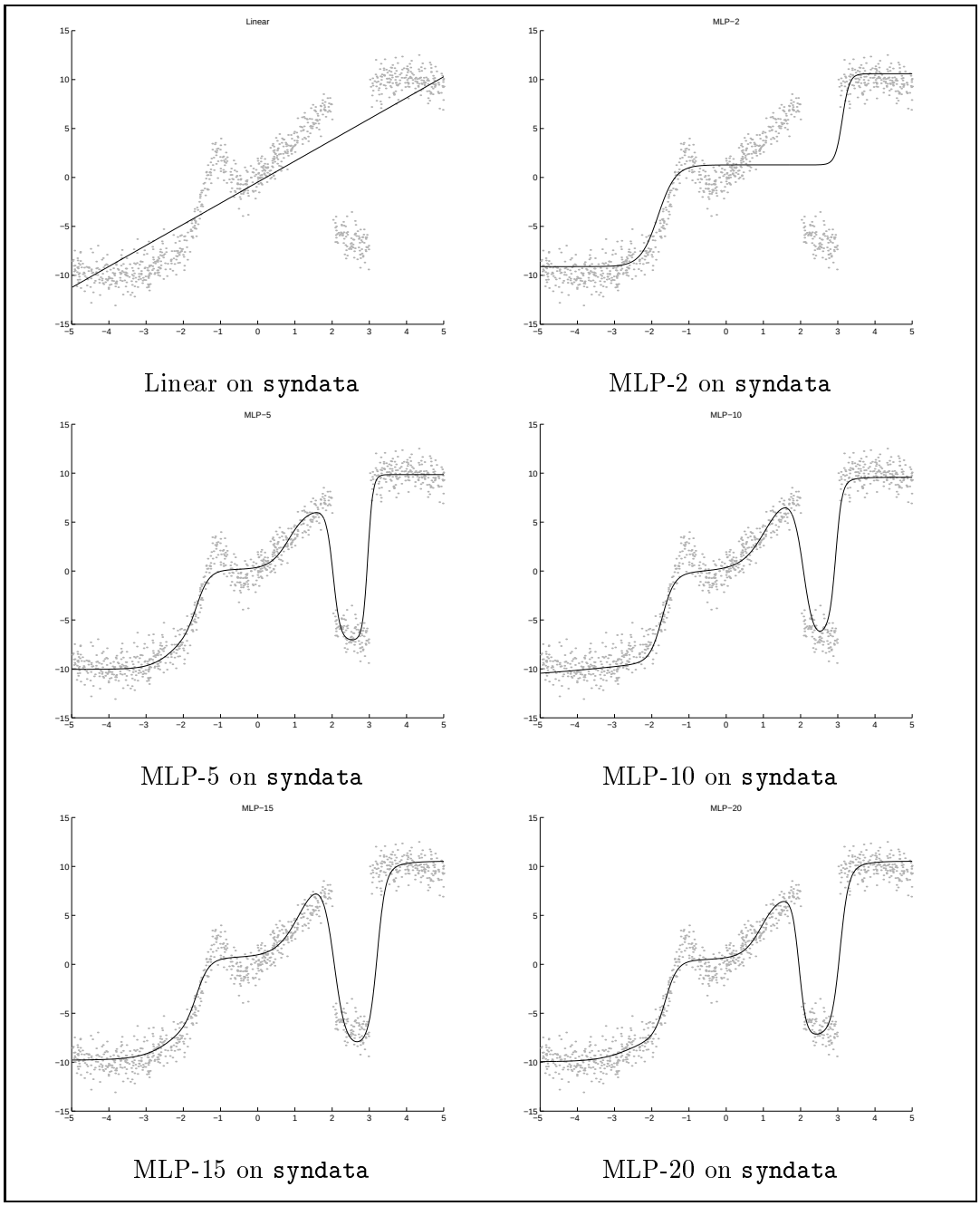
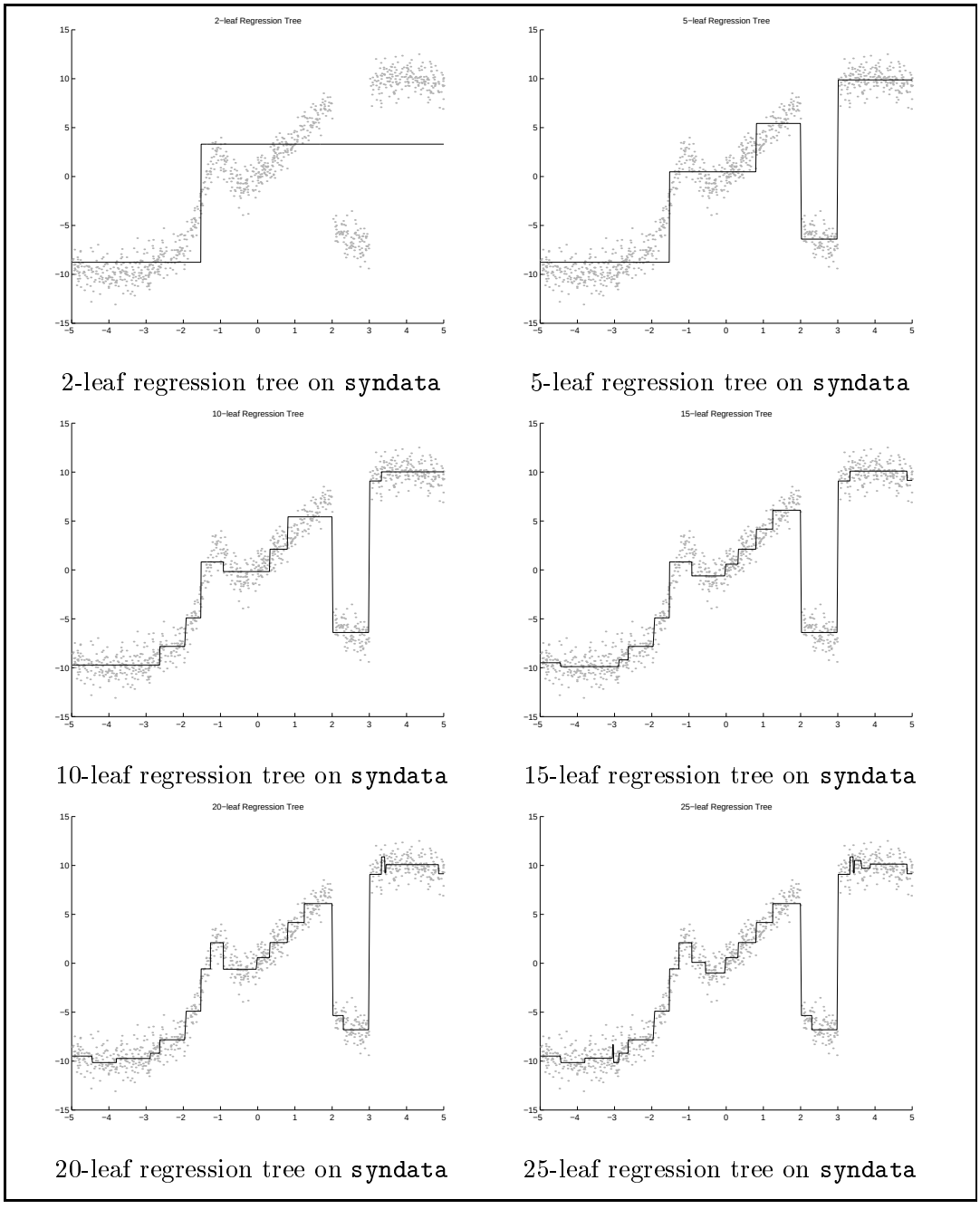


Figure A.3. Linear and MLP models on `syndata`



2-leaf regression tree on `syndata`

5-leaf regression tree on `syndata`

10-leaf regression tree on `syndata`

15-leaf regression tree on `syndata`

20-leaf regression tree on `syndata`

25-leaf regression tree on `syndata`

Figure A.4. Regression tree models on `syndata`

A.2.2. C-REx on `syndata`

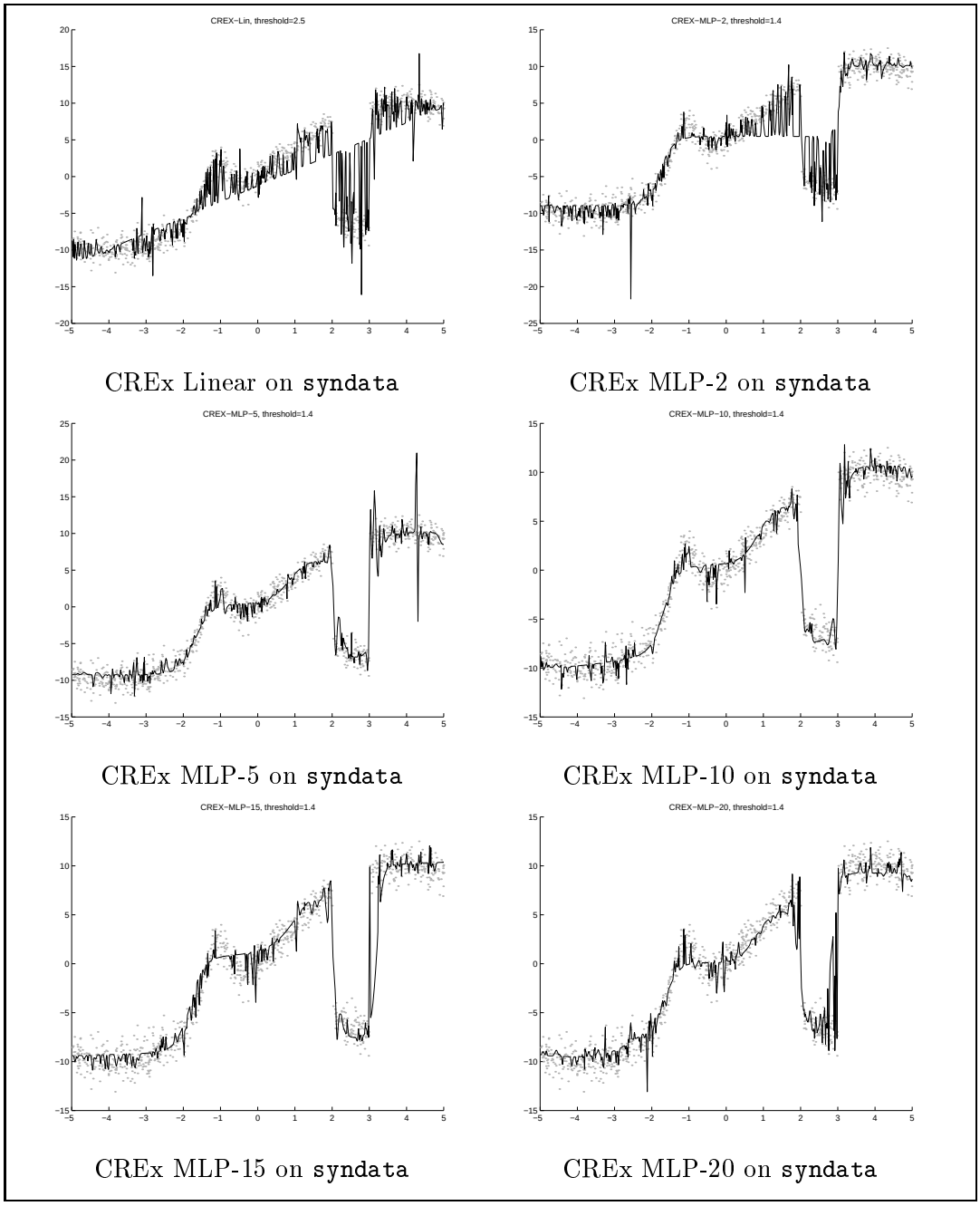


Figure A.5. C-REx without clustering on `syndata`

A.2.3. C-REx on `syndata`

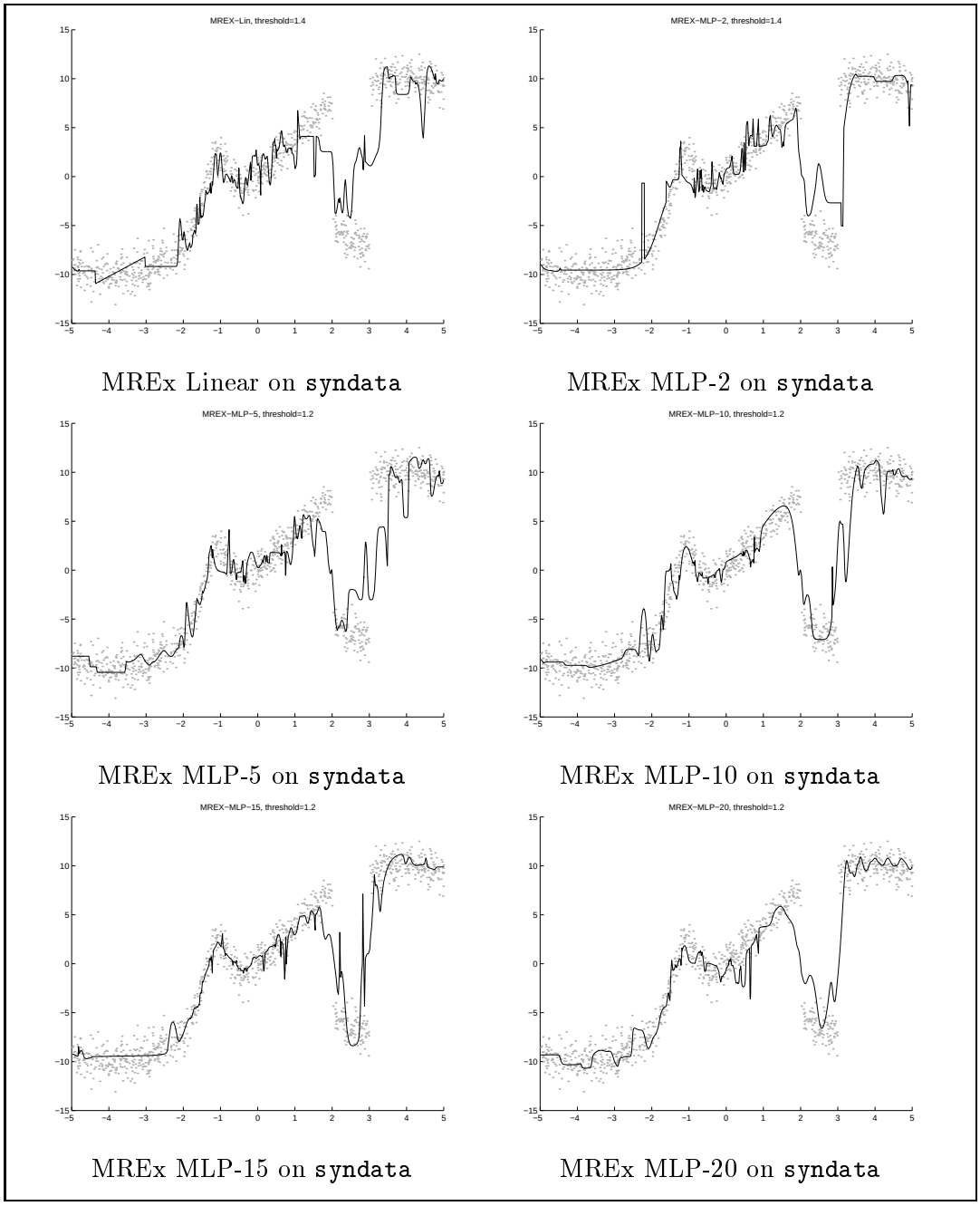


Figure A.6. M-REx without clustering on `syndata`

A.2.4. C-REx with Clustering on `syndata`

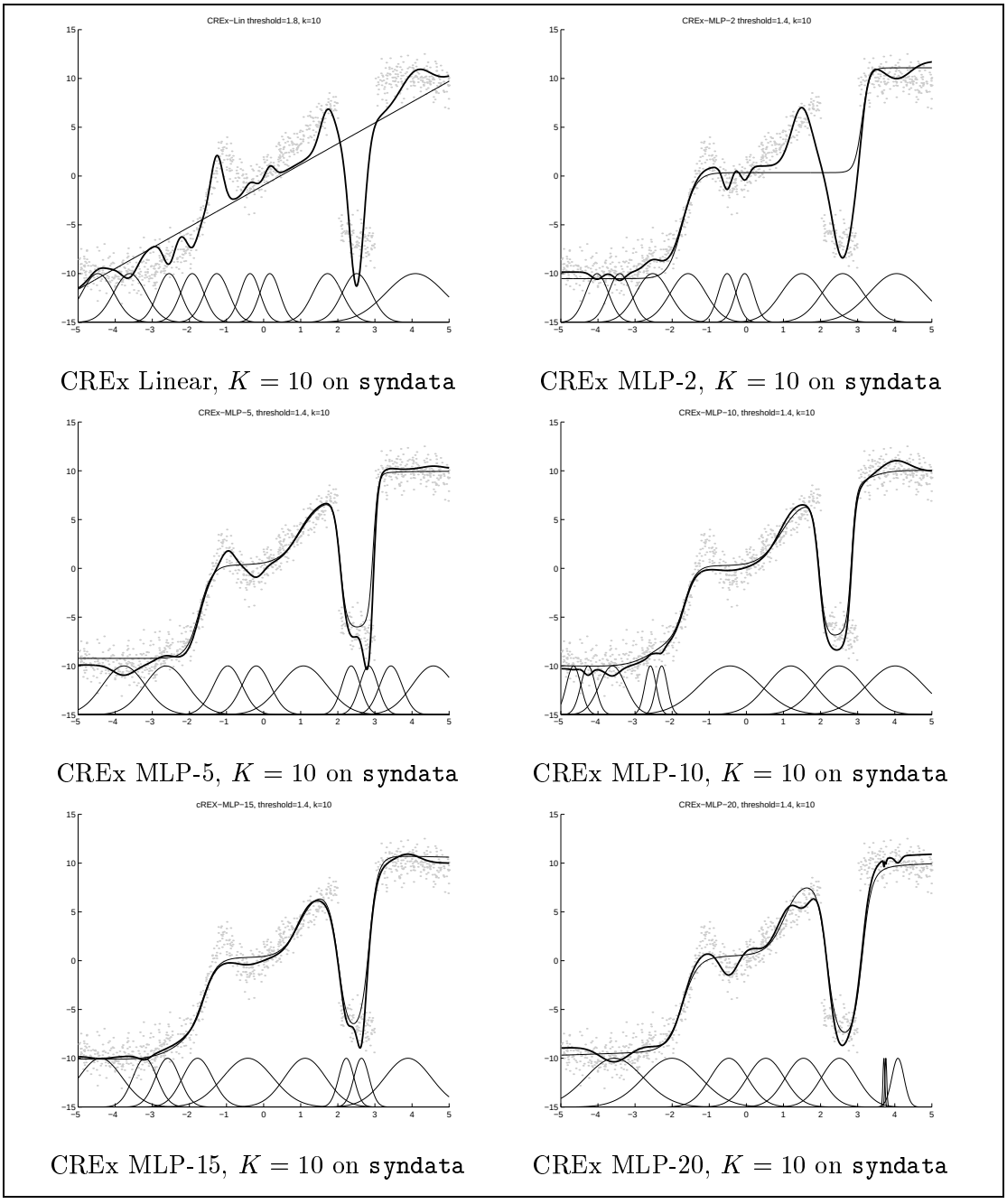


Figure A.7. C-REx with clustering on `syndata`

A.2.5. MREx with Clustering on `syndata`

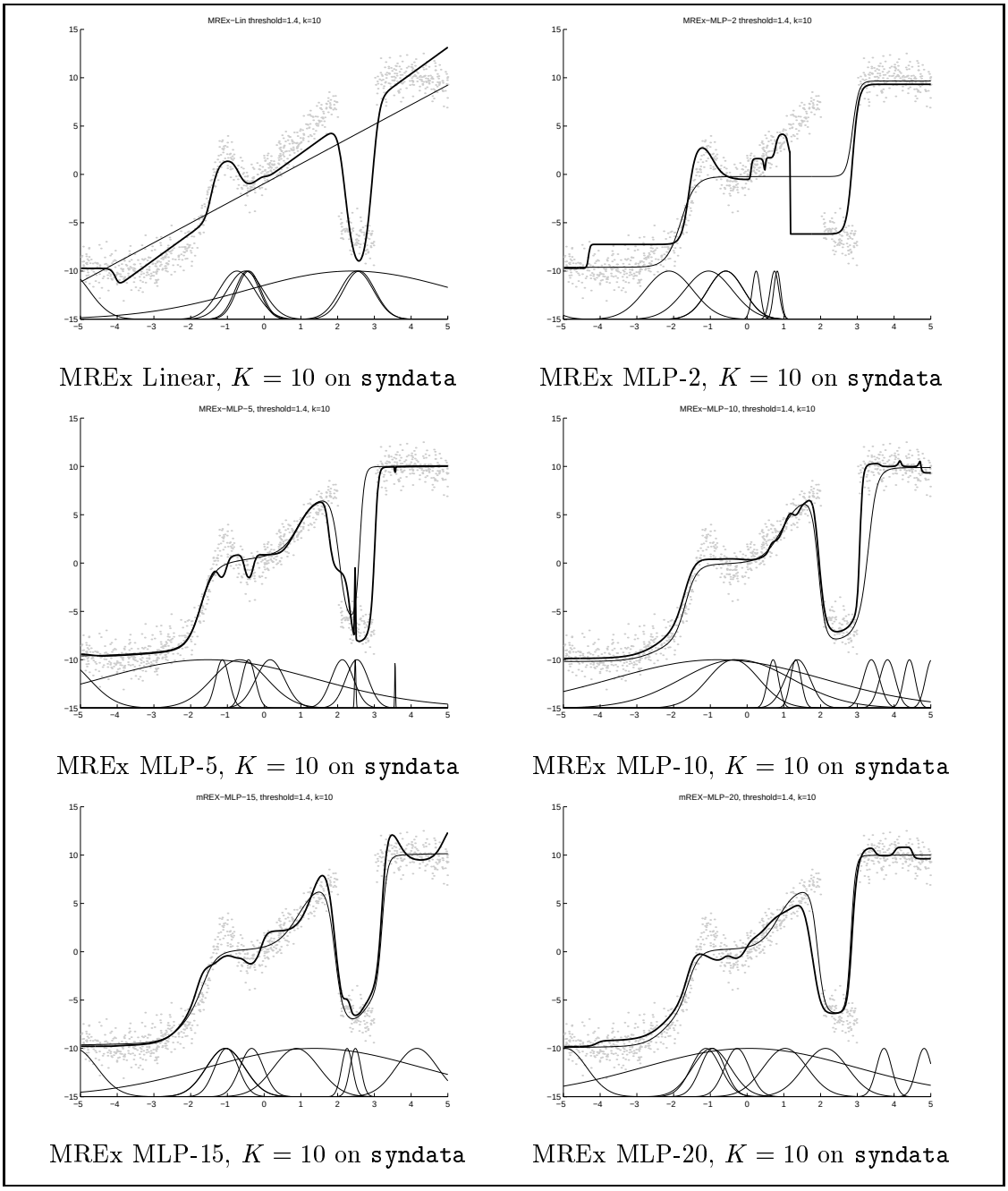
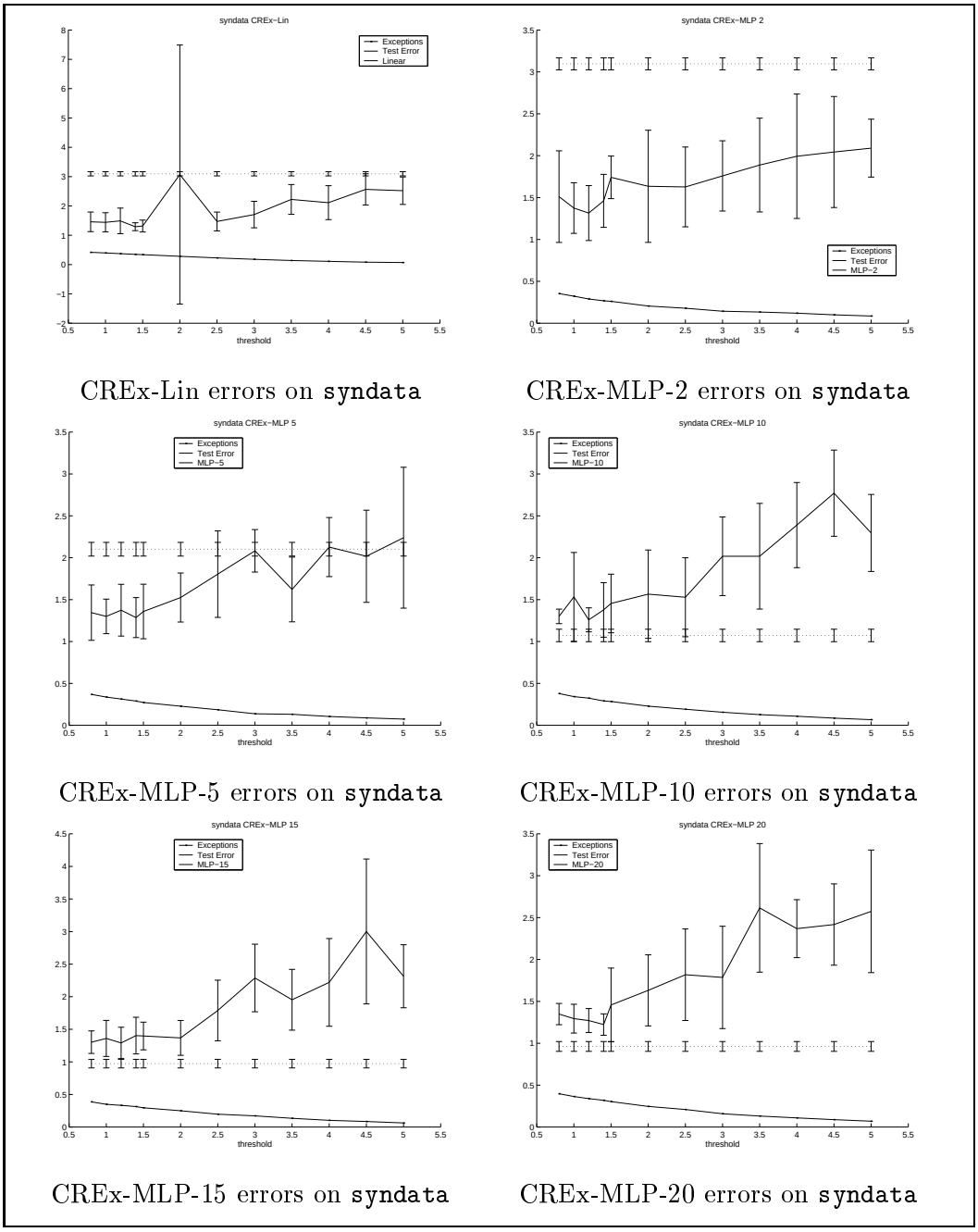


Figure A.8. M-REx with clustering on `syndata`

A.3. Thresholds

A.3.1. CREx Thresholds



CREx-Lin errors on `syndata`

CREx-MLP-2 errors on `syndata`

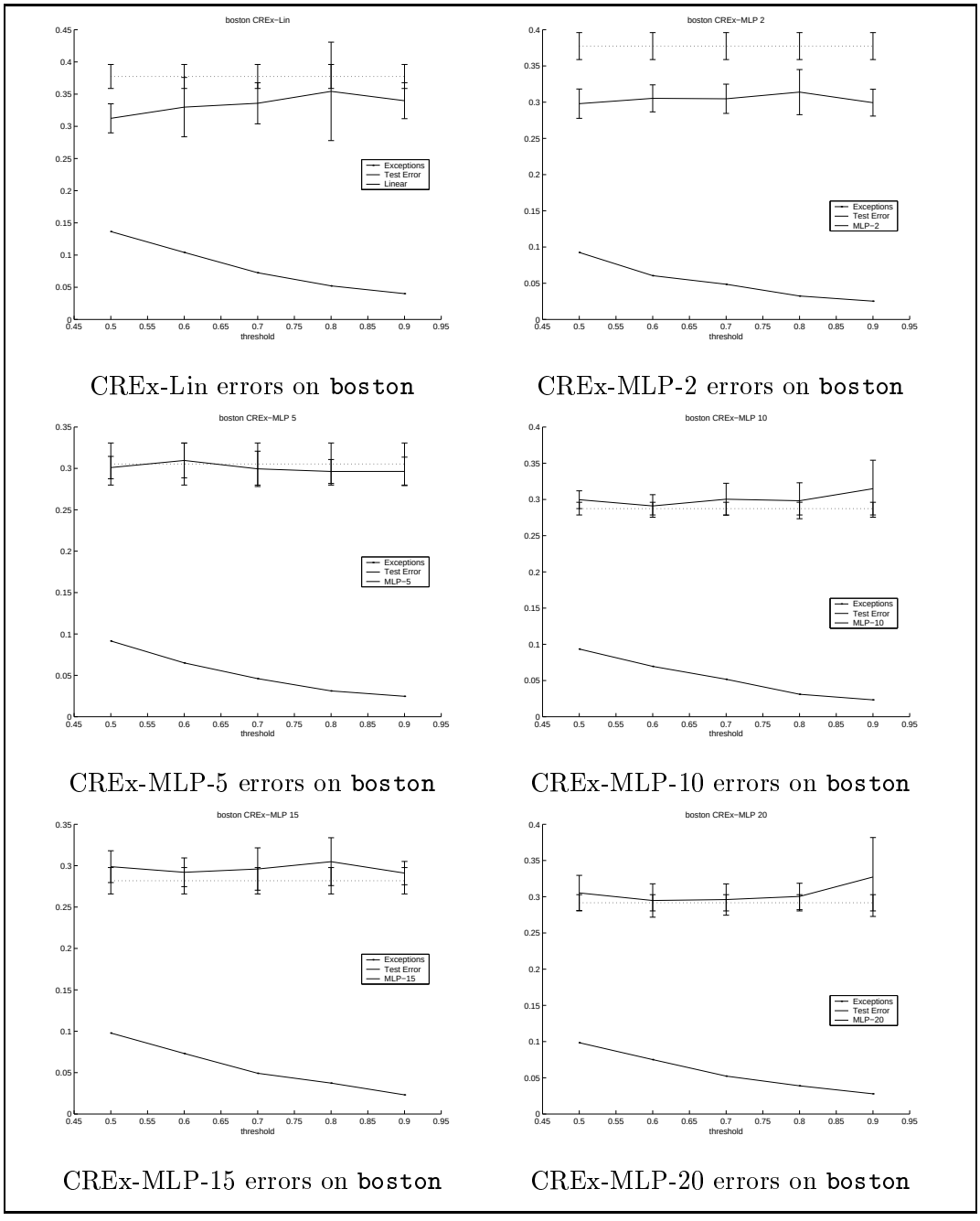
CREx-MLP-5 errors on `syndata`

CREx-MLP-10 errors on `syndata`

CREx-MLP-15 errors on `syndata`

CREx-MLP-20 errors on `syndata`

Figure A.9. C-REx thresholds on `syndata`



CREx-Lin errors on `boston`

CREx-MLP-2 errors on `boston`

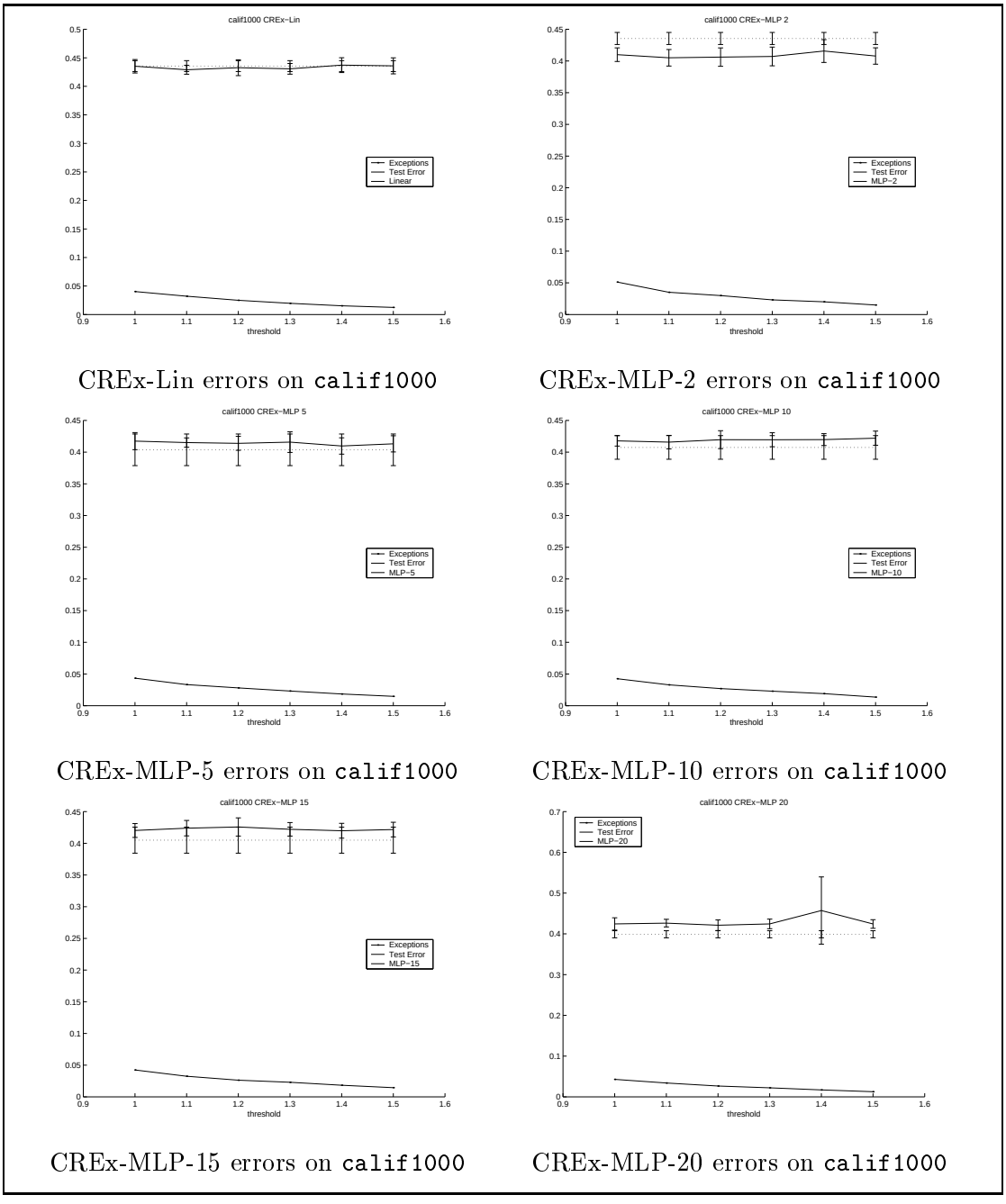
CREx-MLP-5 errors on `boston`

CREx-MLP-10 errors on `boston`

CREx-MLP-15 errors on `boston`

CREx-MLP-20 errors on `boston`

Figure A.10. C-REx thresholds on `boston`



CREx-Lin errors on `calif1000`

CREx-MLP-2 errors on `calif1000`

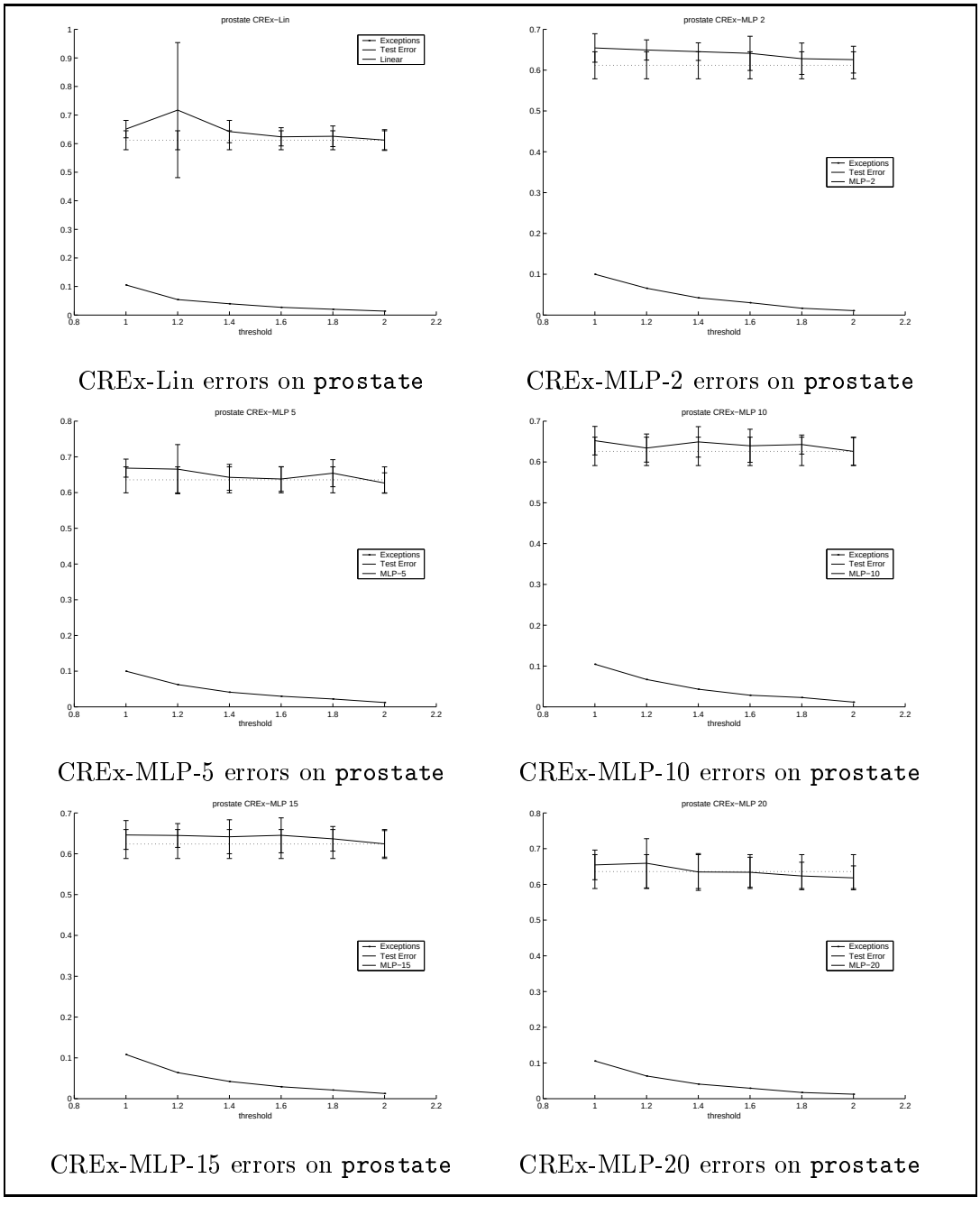
CREx-MLP-5 errors on `calif1000`

CREx-MLP-10 errors on `calif1000`

CREx-MLP-15 errors on `calif1000`

CREx-MLP-20 errors on `calif1000`

Figure A.11. C-REx thresholds on `calif1000`



CREx-Lin errors on `prostate`

CREx-MLP-2 errors on `prostate`

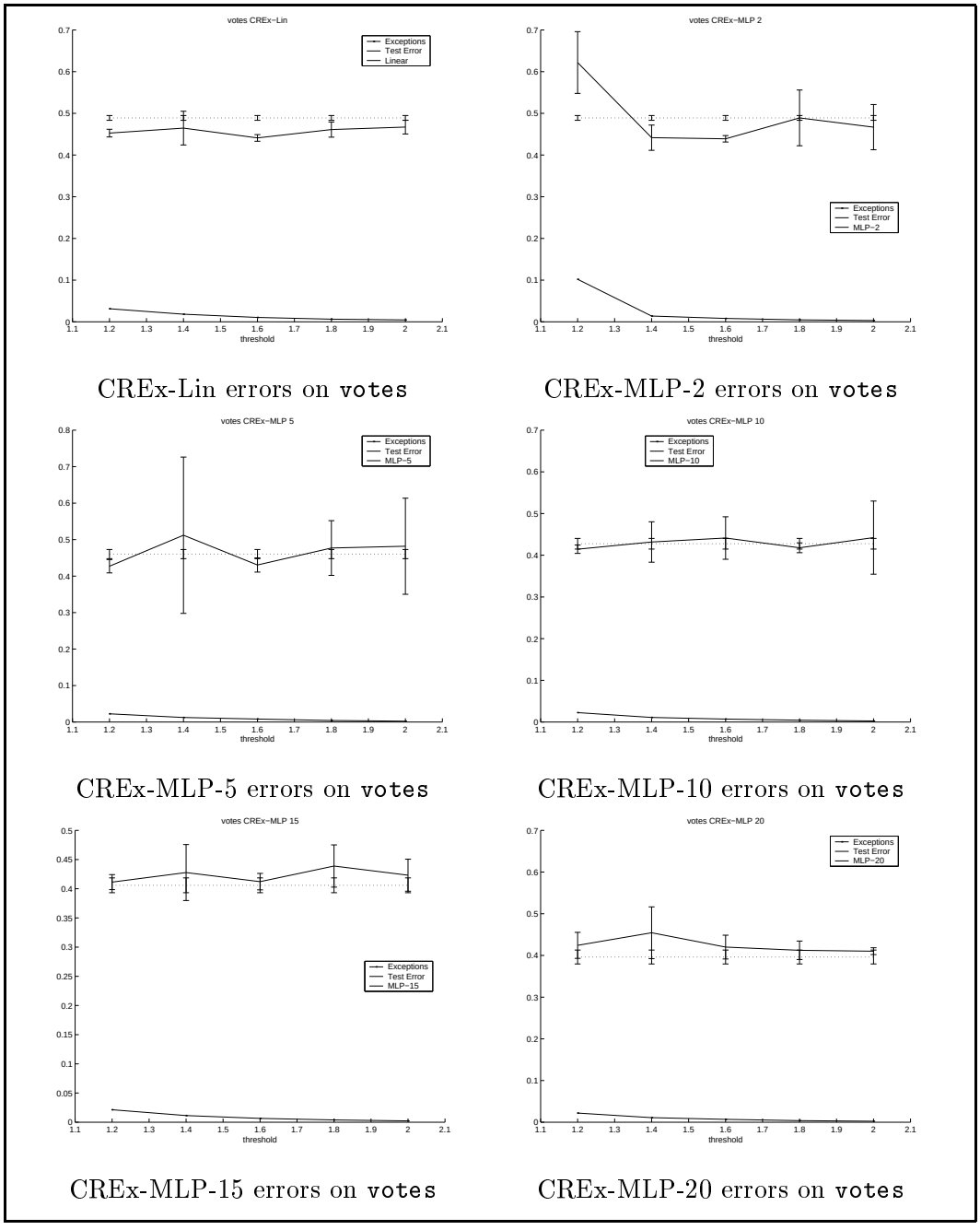
CREx-MLP-5 errors on `prostate`

CREx-MLP-10 errors on `prostate`

CREx-MLP-15 errors on `prostate`

CREx-MLP-20 errors on `prostate`

Figure A.12. C-REx thresholds on `prostate`



CREx-Lin errors on `votes`

CREx-MLP-2 errors on `votes`

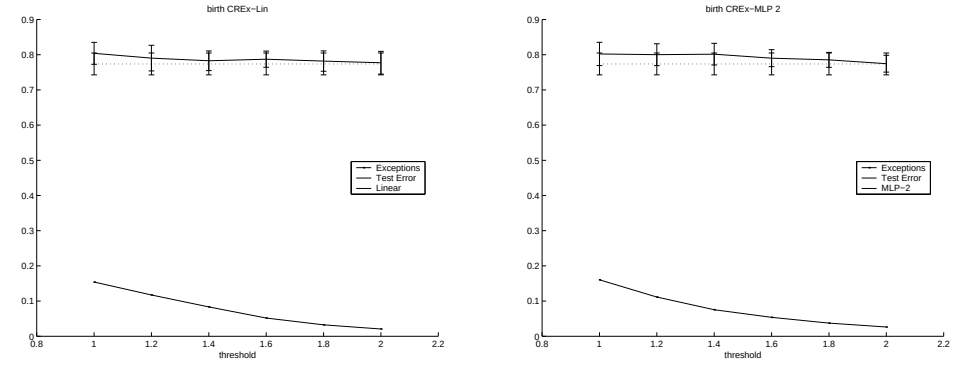
CREx-MLP-5 errors on `votes`

CREx-MLP-10 errors on `votes`

CREx-MLP-15 errors on `votes`

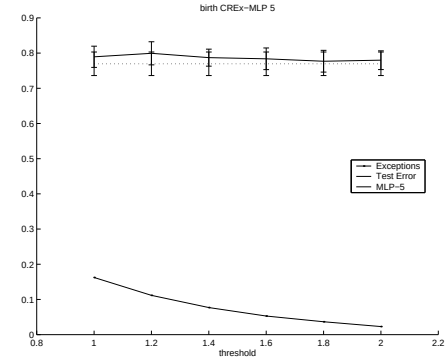
CREx-MLP-20 errors on `votes`

Figure A.13. C-REx thresholds on `votes`

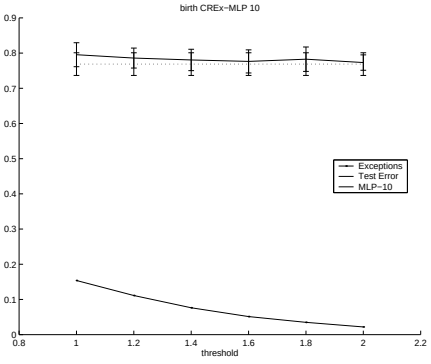


CREx-Lin errors on `birth`

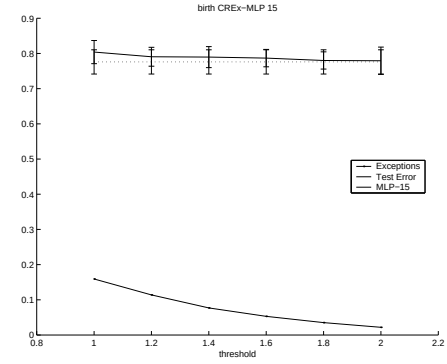
CREx-MLP-2 errors on `birth`



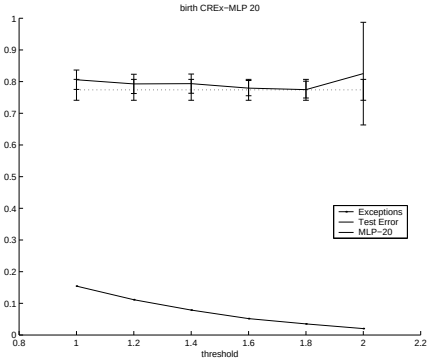
CREx-MLP-5 errors on `birth`



CREx-MLP-10 errors on `birth`

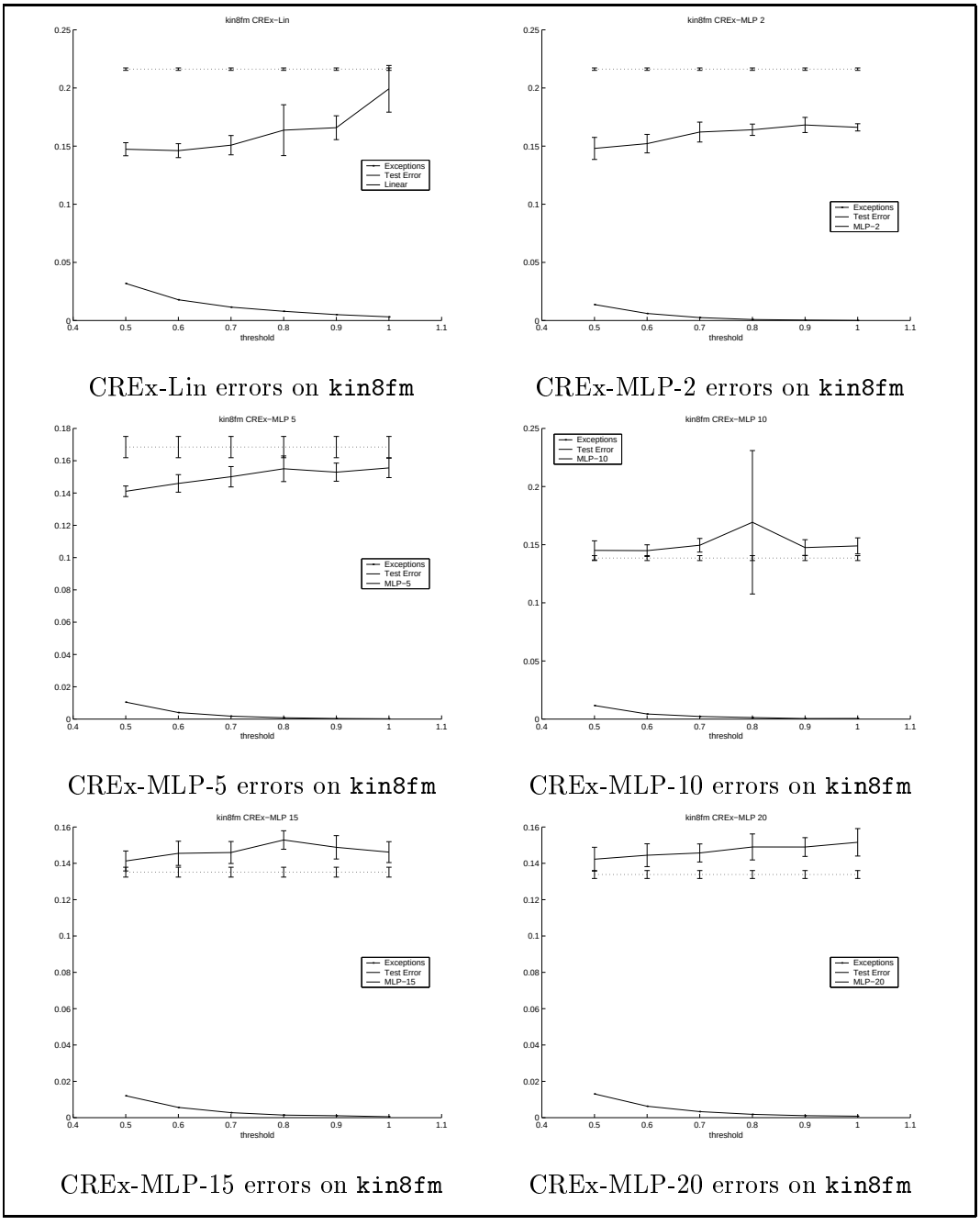


CREx-MLP-15 errors on `birth`



CREx-MLP-20 errors on `birth`

Figure A.14. C-REx thresholds on `birth`



CREx-Lin errors on `kin8fm`

CREx-MLP-2 errors on `kin8fm`

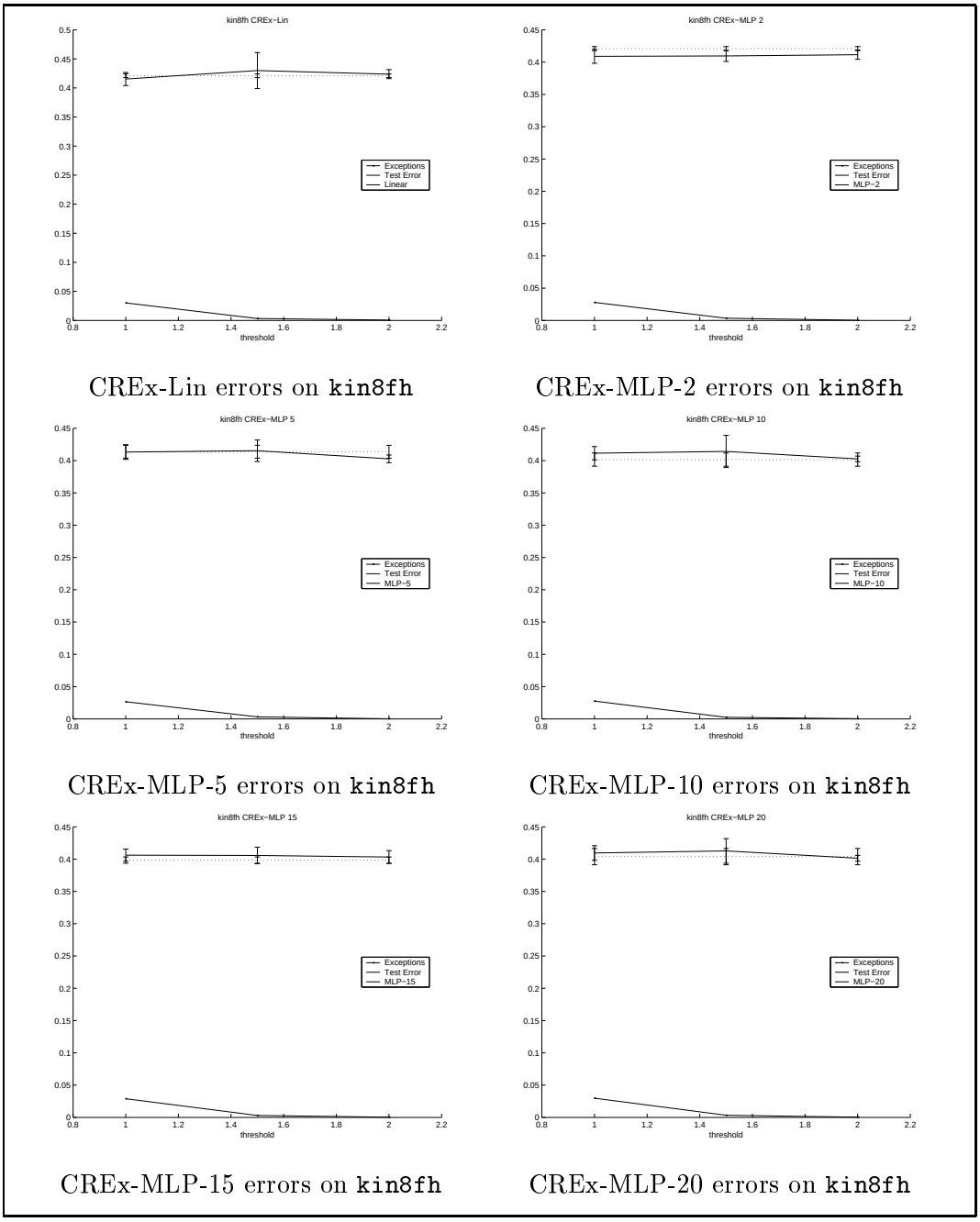
CREx-MLP-5 errors on `kin8fm`

CREx-MLP-10 errors on `kin8fm`

CREx-MLP-15 errors on `kin8fm`

CREx-MLP-20 errors on `kin8fm`

Figure A.15. C-REx thresholds on `kin8fm`



CREx-Lin errors on `kin8fh`

CREx-MLP-2 errors on `kin8fh`

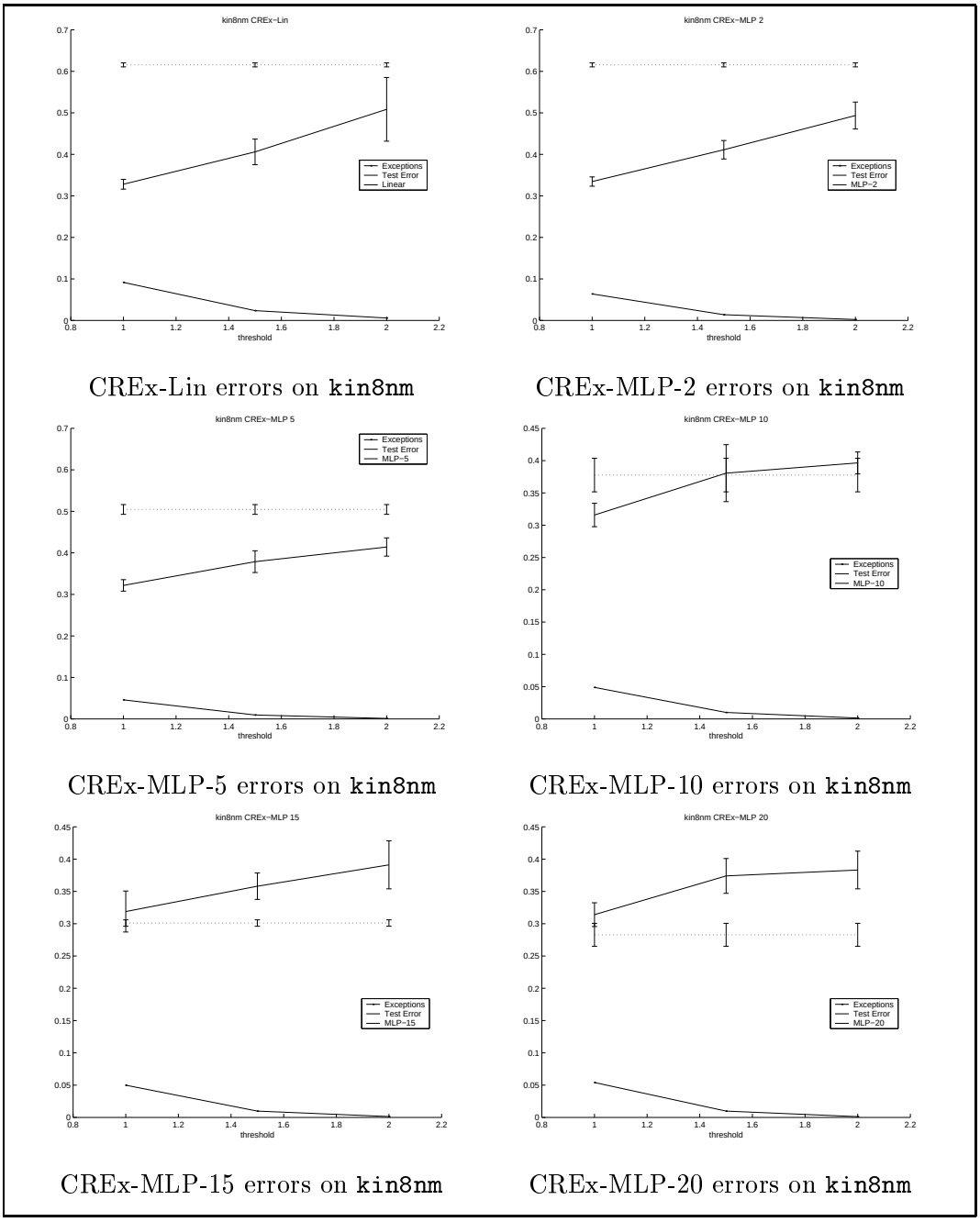
CREx-MLP-5 errors on `kin8fh`

CREx-MLP-10 errors on `kin8fh`

CREx-MLP-15 errors on `kin8fh`

CREx-MLP-20 errors on `kin8fh`

Figure A.16. C-REx thresholds on `kin8fh`

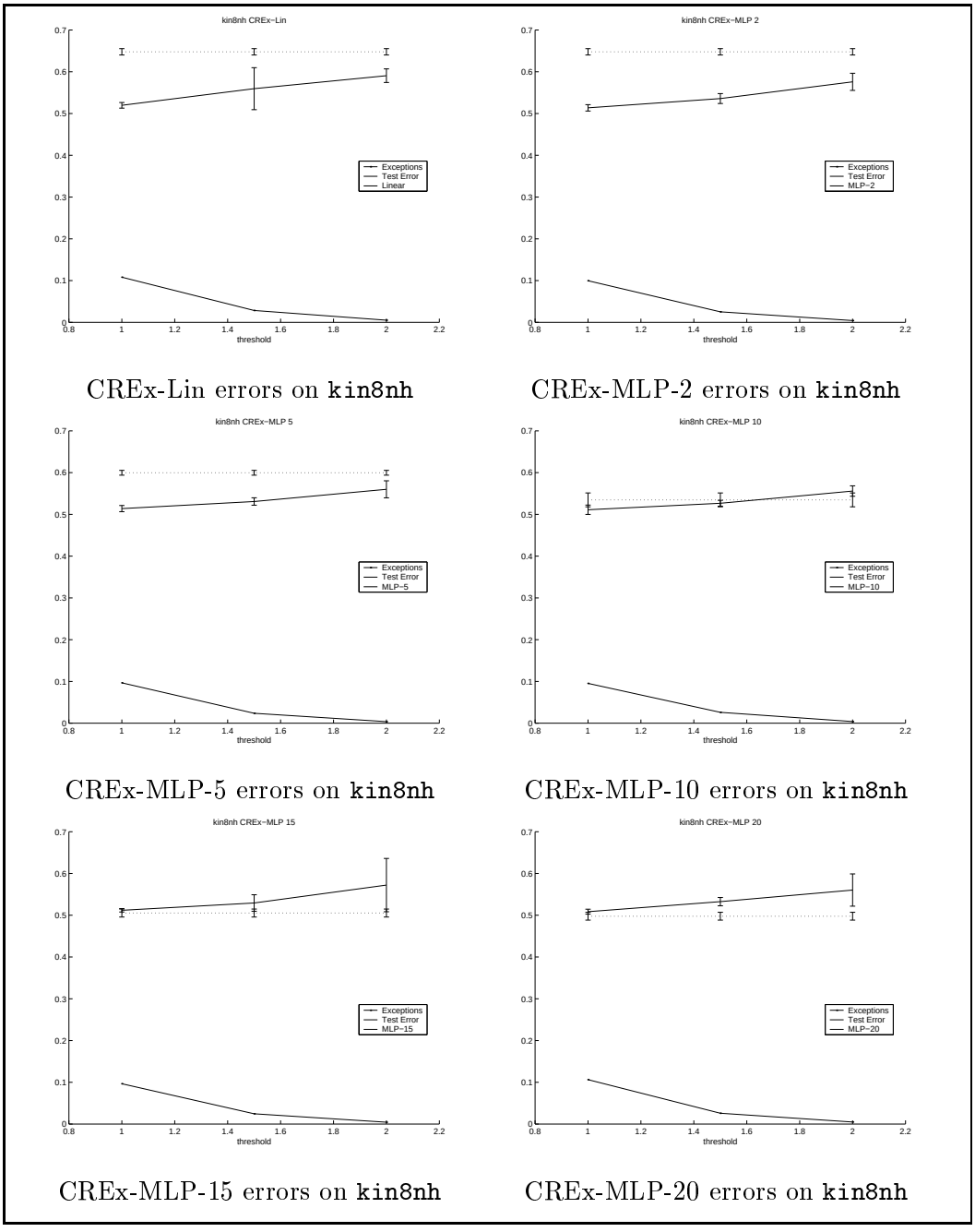


CREx-Lin errors on `kin8nm` CREx-MLP-2 errors on `kin8nm`

CREx-MLP-5 errors on `kin8nm` CREx-MLP-10 errors on `kin8nm`

CREx-MLP-15 errors on `kin8nm` CREx-MLP-20 errors on `kin8nm`

Figure A.17. C-REx thresholds on `kin8nm`



CREx-Lin errors on `kin8nh`

CREx-MLP-2 errors on `kin8nh`

CREx-MLP-5 errors on `kin8nh`

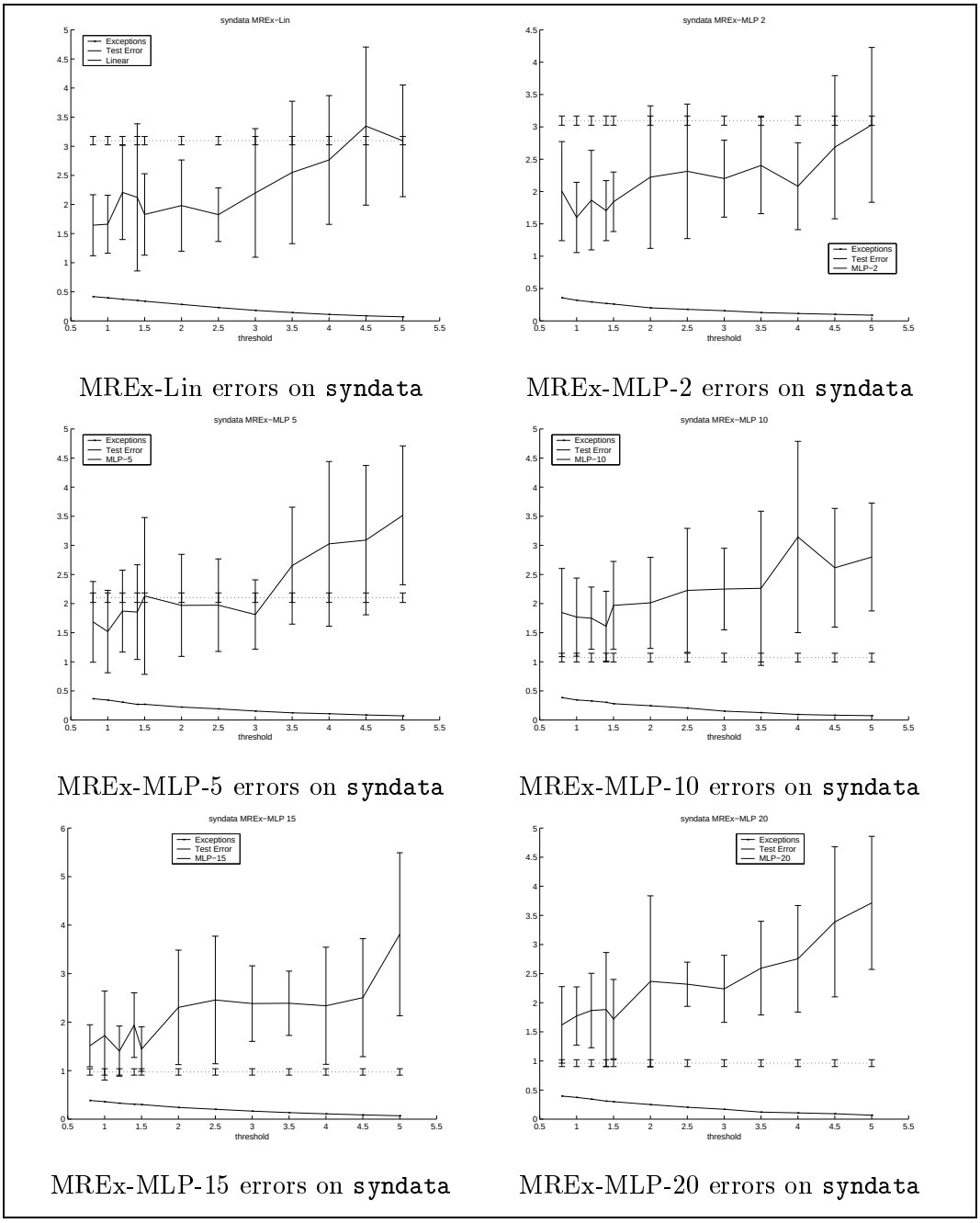
CREx-MLP-10 errors on `kin8nh`

CREx-MLP-15 errors on `kin8nh`

CREx-MLP-20 errors on `kin8nh`

Figure A.18. C-REx thresholds on `kin8nh`

A.3.2. MREx Thresholds



MREx-Lin errors on `syndata`

MREx-MLP-2 errors on `syndata`

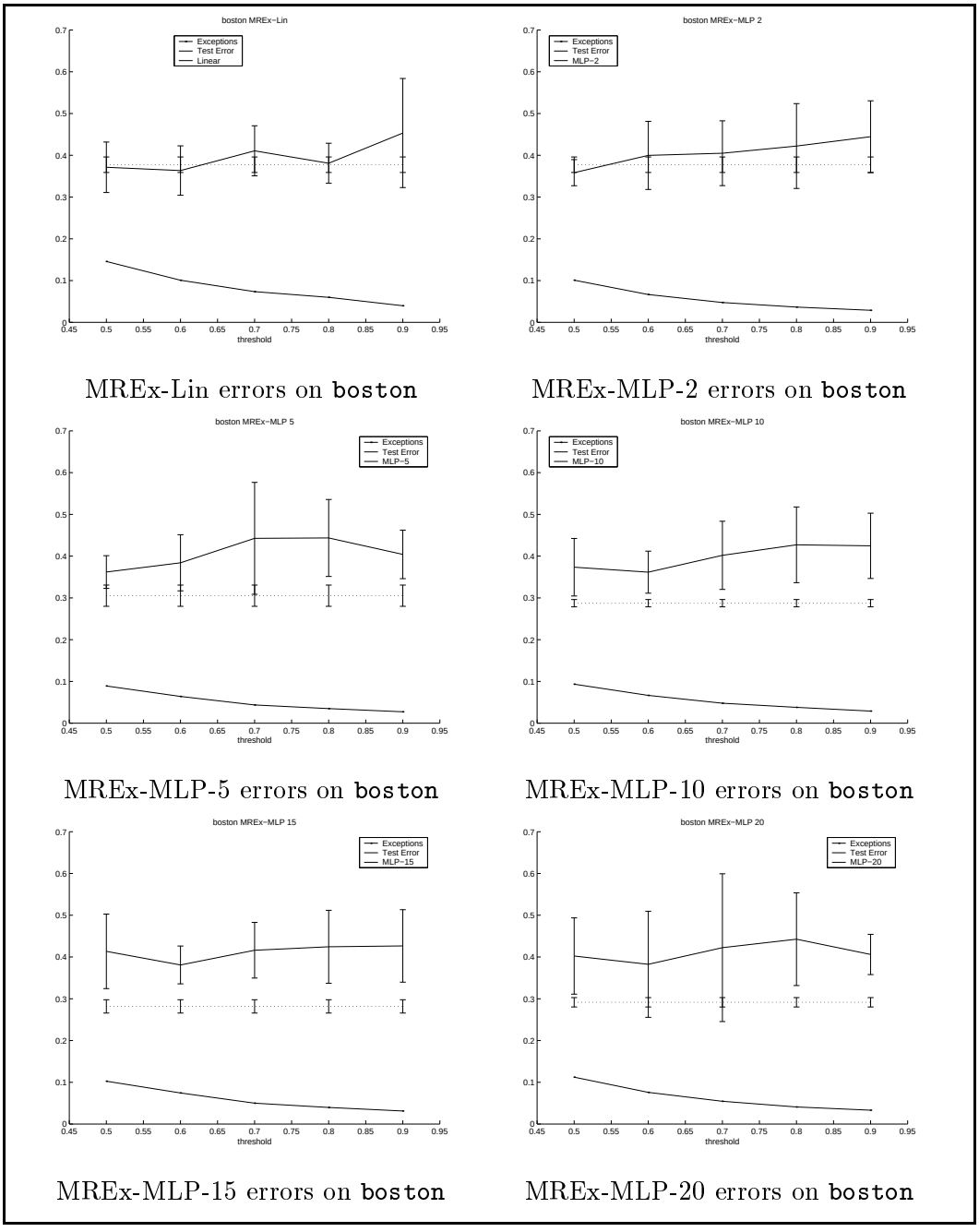
MREx-MLP-5 errors on `syndata`

MREx-MLP-10 errors on `syndata`

MREx-MLP-15 errors on `syndata`

MREx-MLP-20 errors on `syndata`

Figure A.19. M-REx thresholds on `syndata`



MREx-Lin errors on `boston`

MREx-MLP-2 errors on `boston`

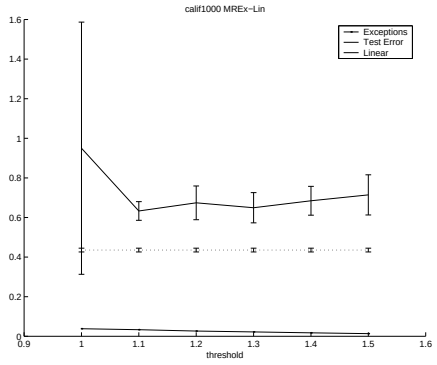
MREx-MLP-5 errors on `boston`

MREx-MLP-10 errors on `boston`

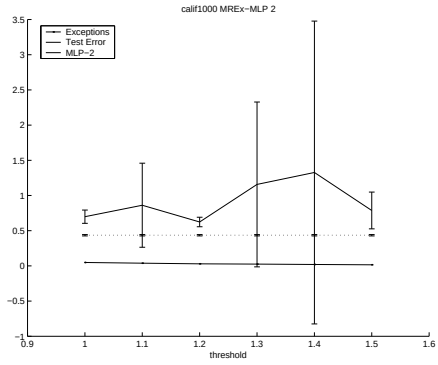
MREx-MLP-15 errors on `boston`

MREx-MLP-20 errors on `boston`

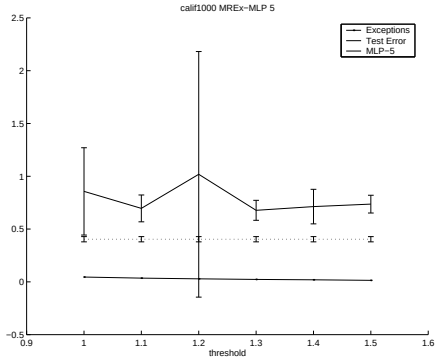
Figure A.20. M-REx thresholds on `boston`



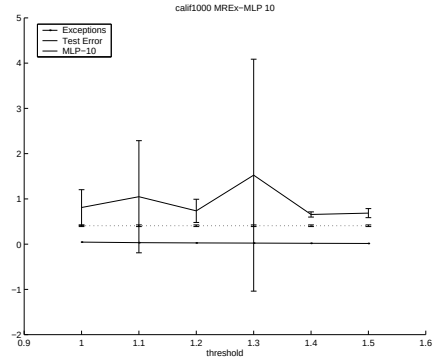
MREx-Lin errors on `calif1000`



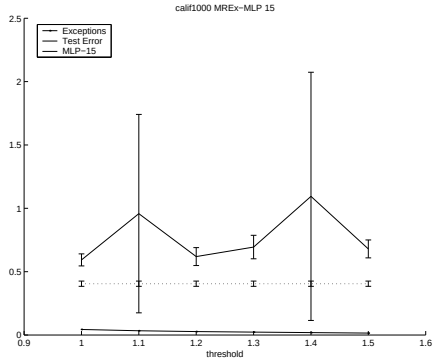
MREx-MLP-2 errors on `calif1000`



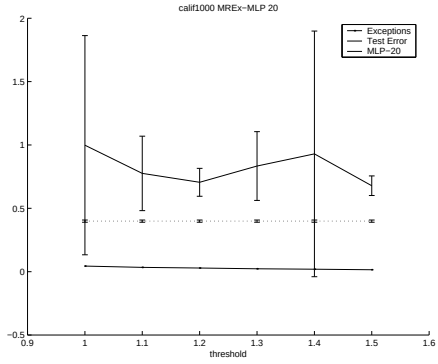
MREx-MLP-5 errors on `calif1000`



MREx-MLP-10 errors on `calif1000`

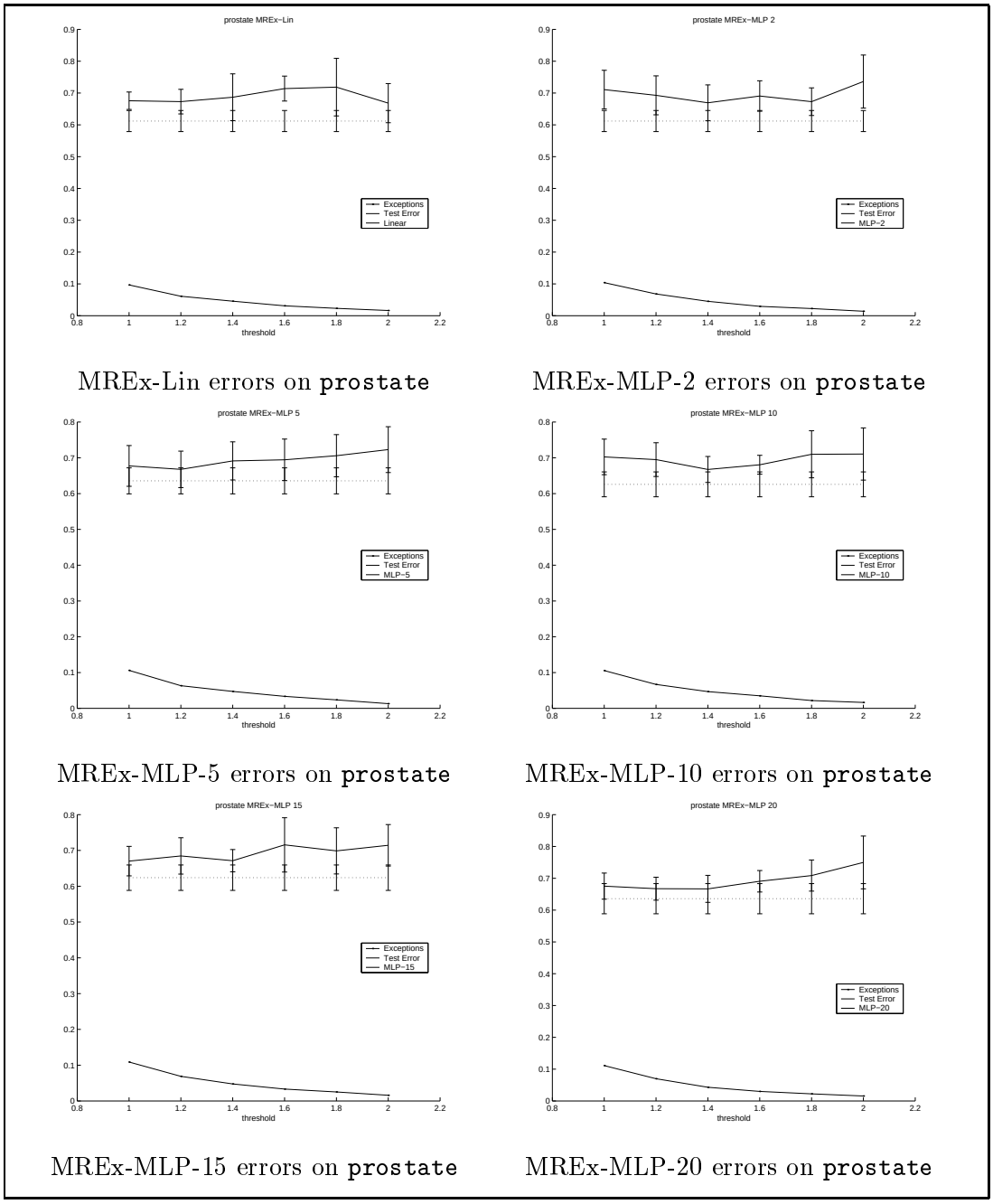


MREx-MLP-15 errors on `calif1000`



MREx-MLP-20 errors on `calif1000`

Figure A.21. M-REx thresholds on `calif1000`



MREx-Lin errors on `prostate`

MREx-MLP-2 errors on `prostate`

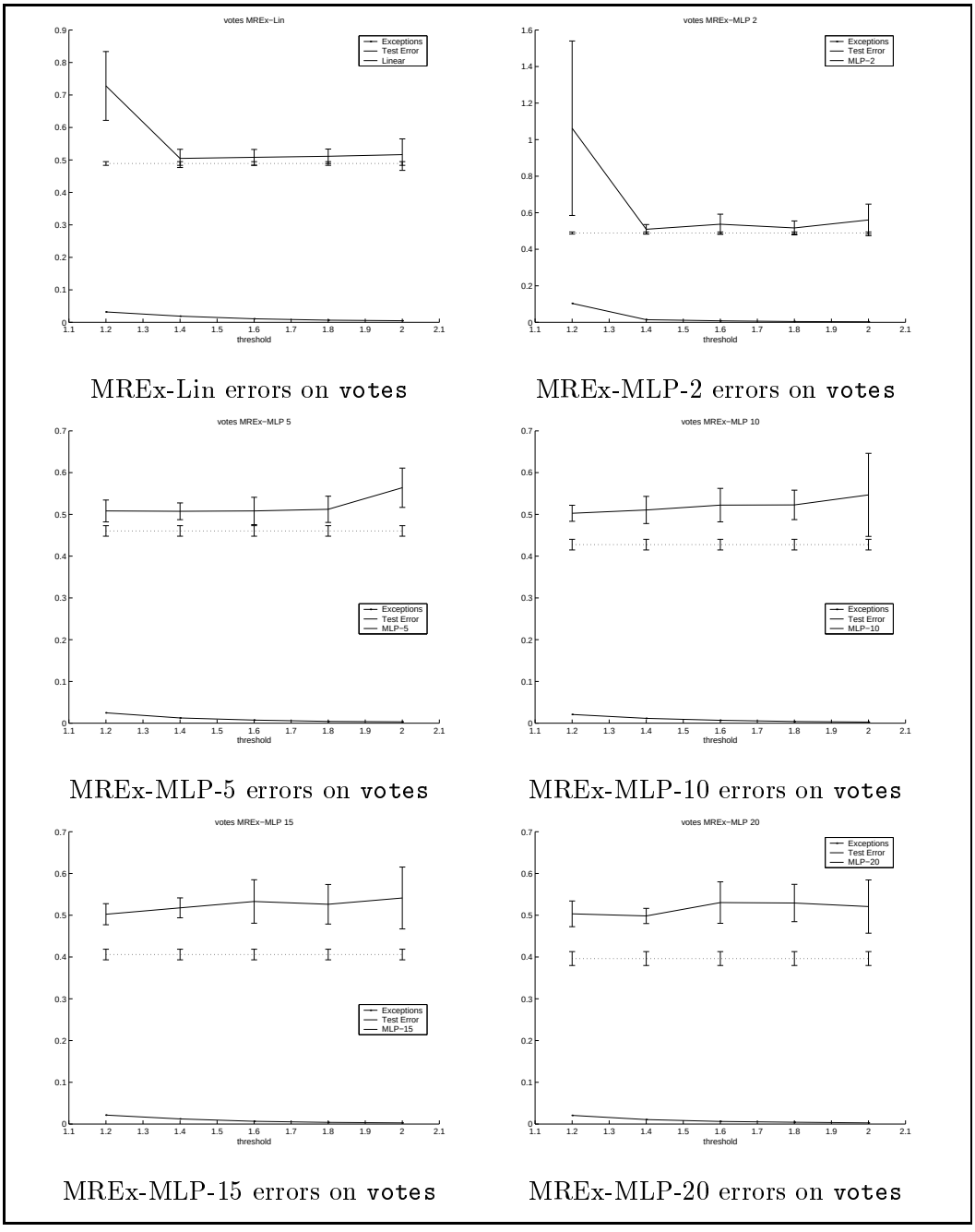
MREx-MLP-5 errors on `prostate`

MREx-MLP-10 errors on `prostate`

MREx-MLP-15 errors on `prostate`

MREx-MLP-20 errors on `prostate`

Figure A.22. M-REx thresholds on `prostate`



MREx-Lin errors on `votes`

MREx-MLP-2 errors on `votes`

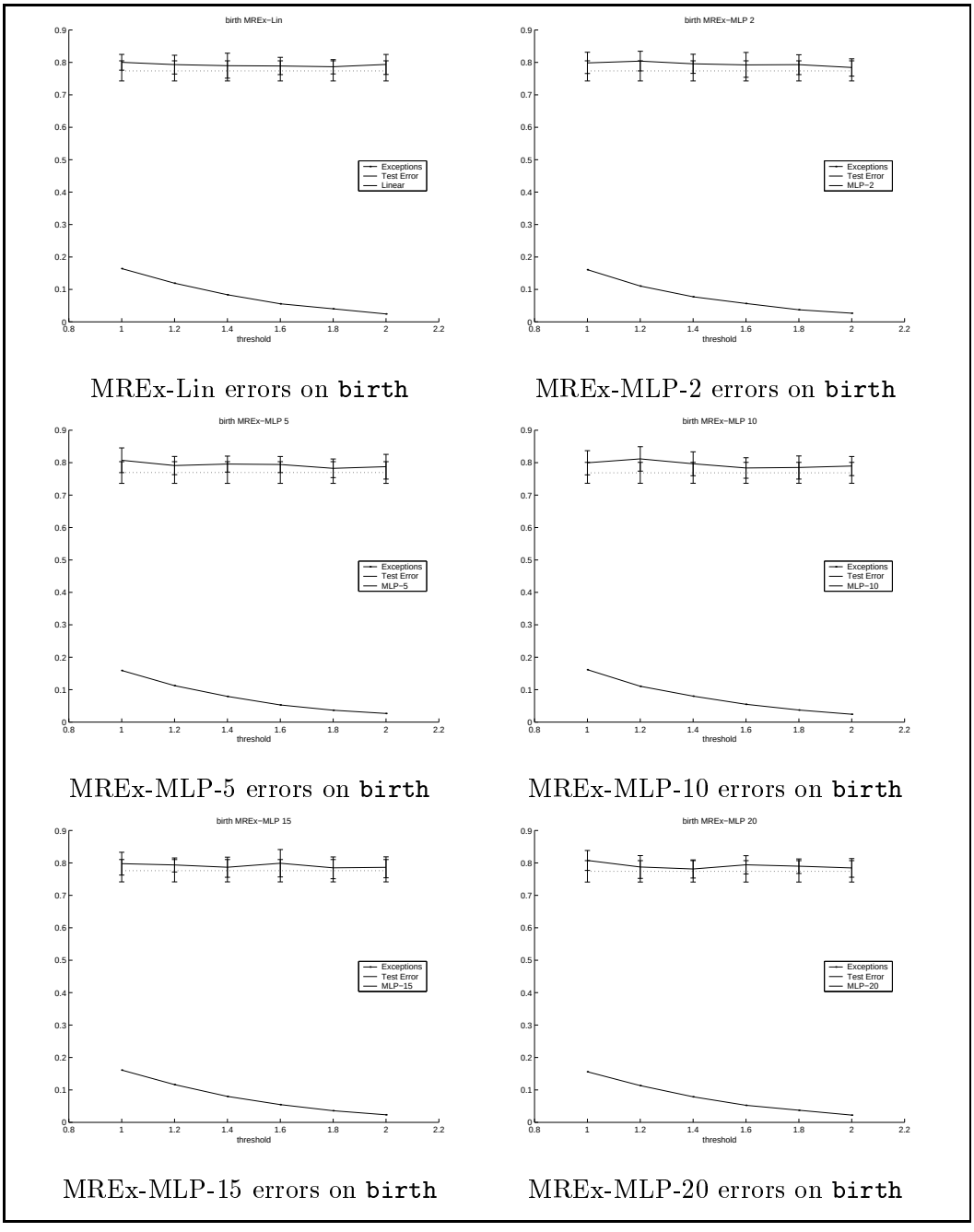
MREx-MLP-5 errors on `votes`

MREx-MLP-10 errors on `votes`

MREx-MLP-15 errors on `votes`

MREx-MLP-20 errors on `votes`

Figure A.23. M-REx thresholds on `votes`



MREx-Lin errors on `birth`

MREx-MLP-2 errors on `birth`

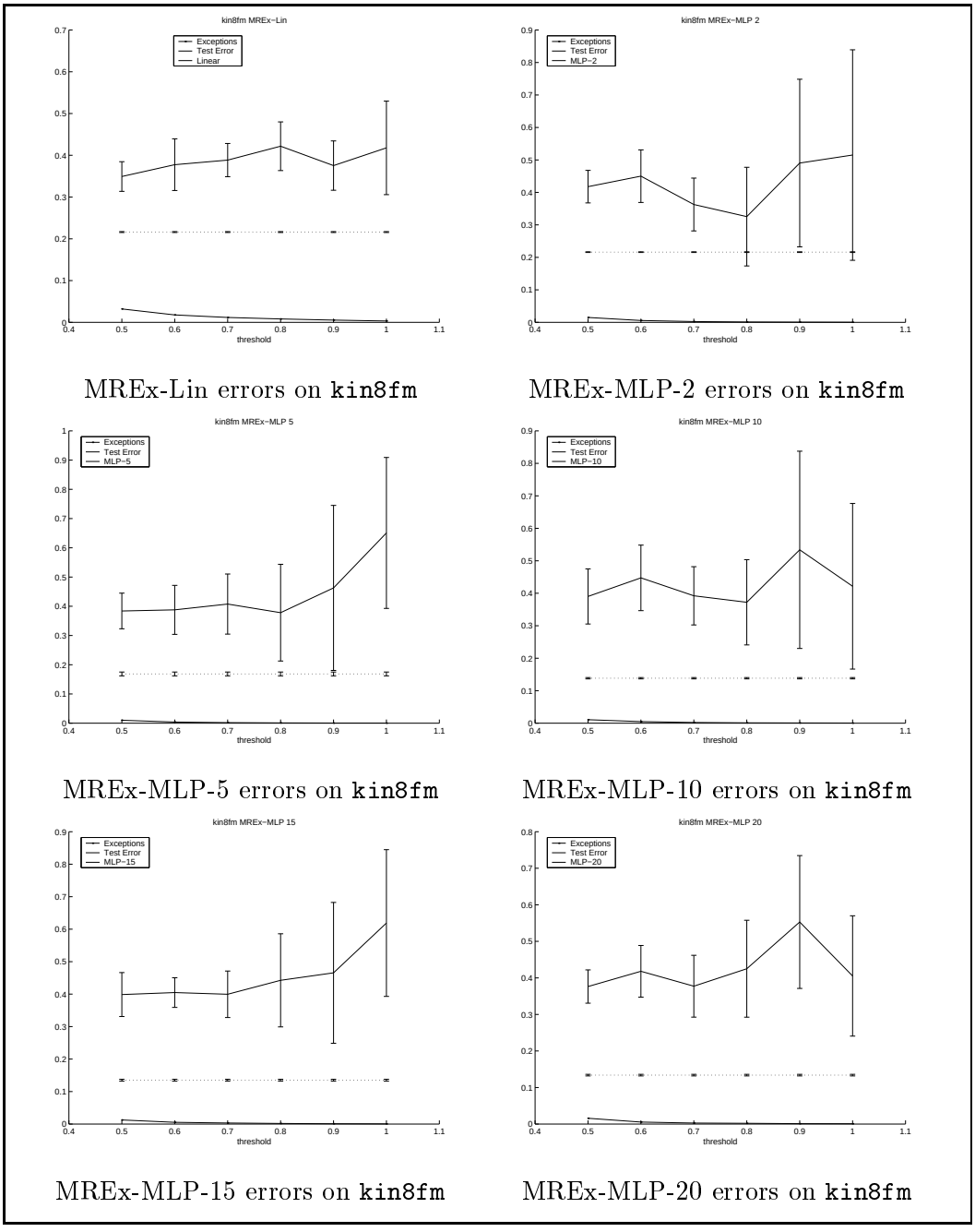
MREx-MLP-5 errors on `birth`

MREx-MLP-10 errors on `birth`

MREx-MLP-15 errors on `birth`

MREx-MLP-20 errors on `birth`

Figure A.24. M-REx thresholds on `birth`



MREx-Lin errors on `kin8fm`

MREx-MLP-2 errors on `kin8fm`

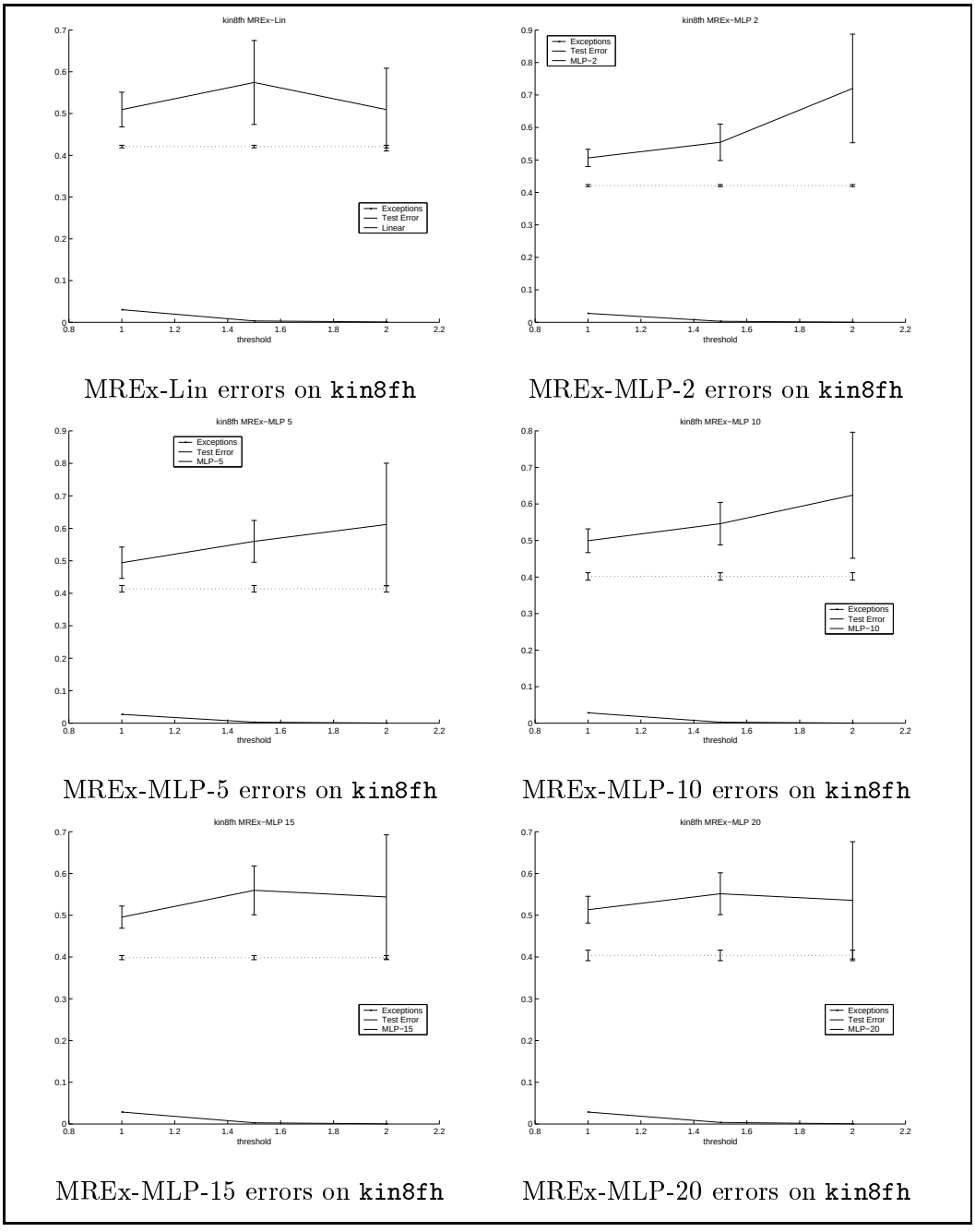
MREx-MLP-5 errors on `kin8fm`

MREx-MLP-10 errors on `kin8fm`

MREx-MLP-15 errors on `kin8fm`

MREx-MLP-20 errors on `kin8fm`

Figure A.25. M-REx thresholds on `kin8fm`



MREx-Lin errors on `kin8fh`

MREx-MLP-2 errors on `kin8fh`

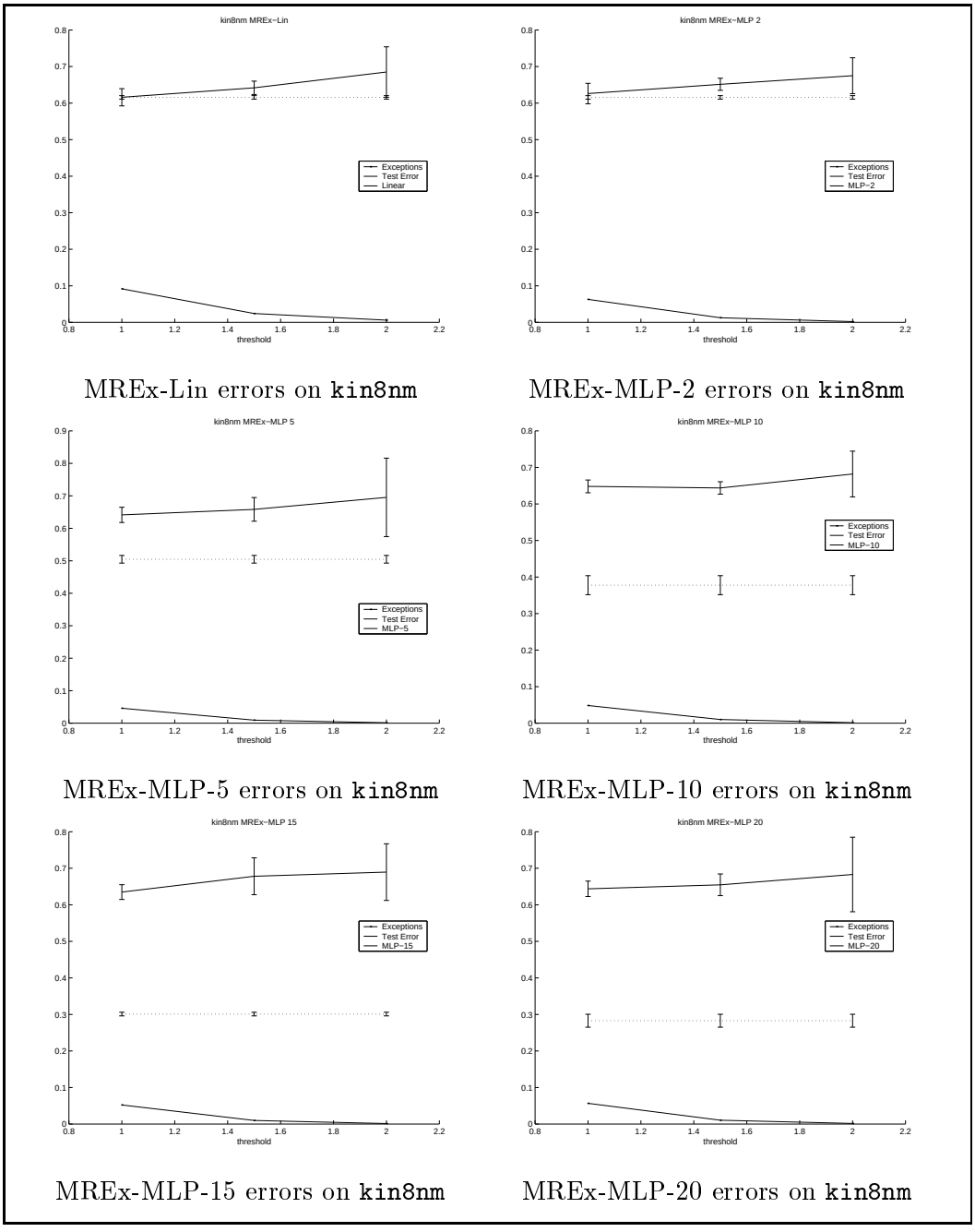
MREx-MLP-5 errors on `kin8fh`

MREx-MLP-10 errors on `kin8fh`

MREx-MLP-15 errors on `kin8fh`

MREx-MLP-20 errors on `kin8fh`

Figure A.26. M-REx thresholds on `kin8fh`



MREx-Lin errors on `kin8nm`

MREx-MLP-2 errors on `kin8nm`

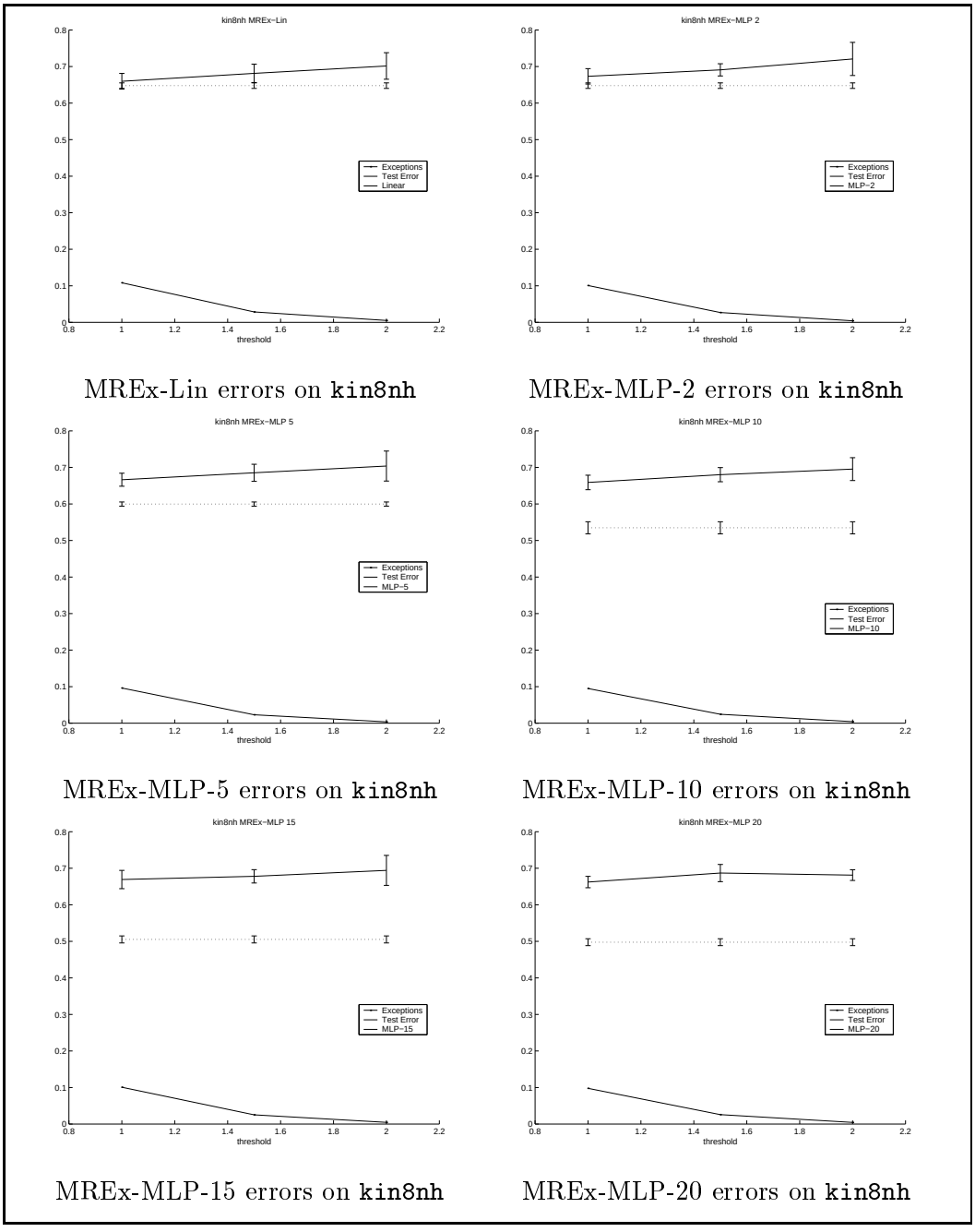
MREx-MLP-5 errors on `kin8nm`

MREx-MLP-10 errors on `kin8nm`

MREx-MLP-15 errors on `kin8nm`

MREx-MLP-20 errors on `kin8nm`

Figure A.27. M-REx thresholds on `kin8nm`



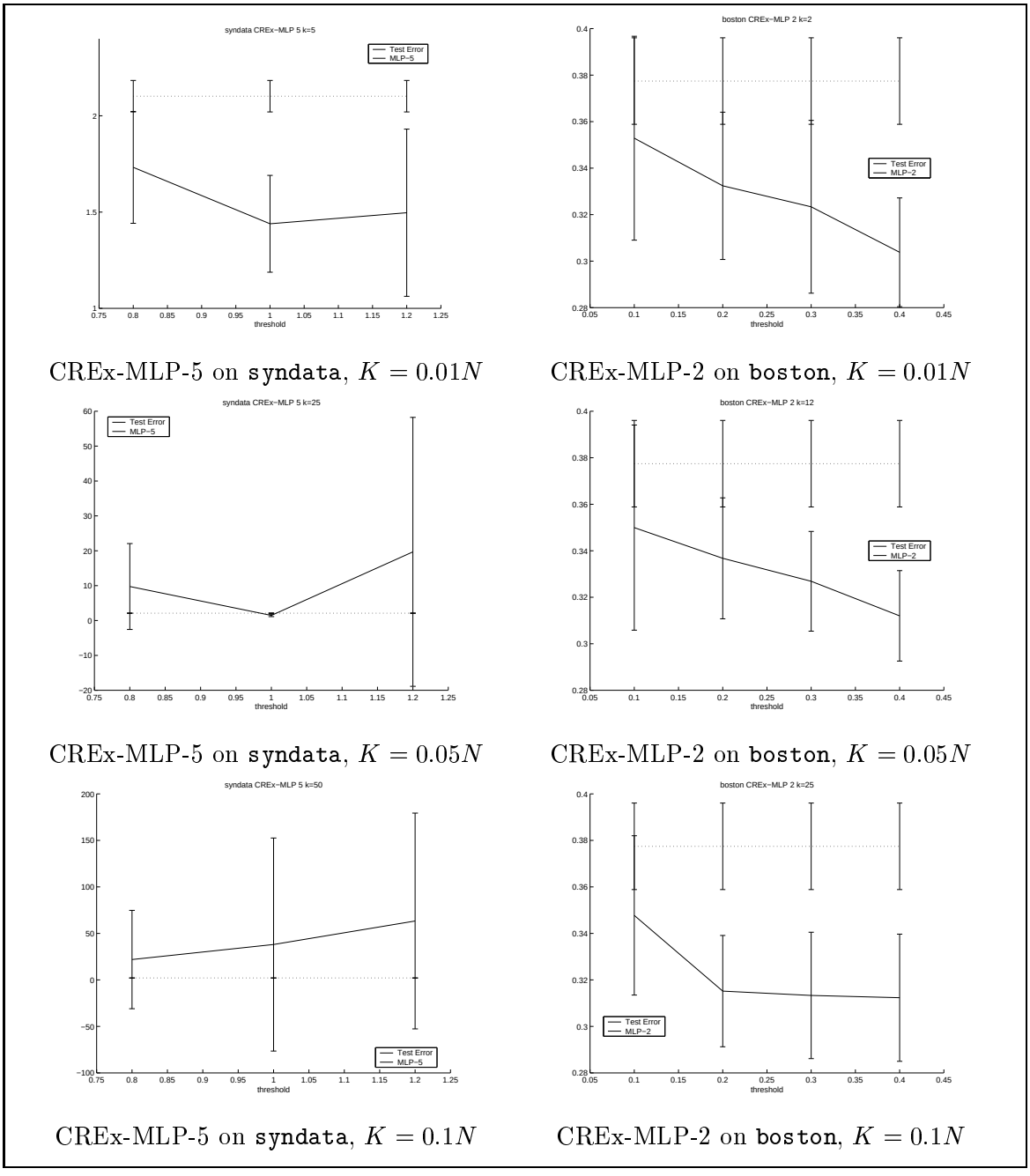
MREx-Lin errors on `kin8nh` MREx-MLP-2 errors on `kin8nh`

MREx-MLP-5 errors on `kin8nh` MREx-MLP-10 errors on `kin8nh`

MREx-MLP-15 errors on `kin8nh` MREx-MLP-20 errors on `kin8nh`

Figure A.28. M-REx thresholds on `kin8nh`

A.3.3. C-REx Thresholds with Clustering

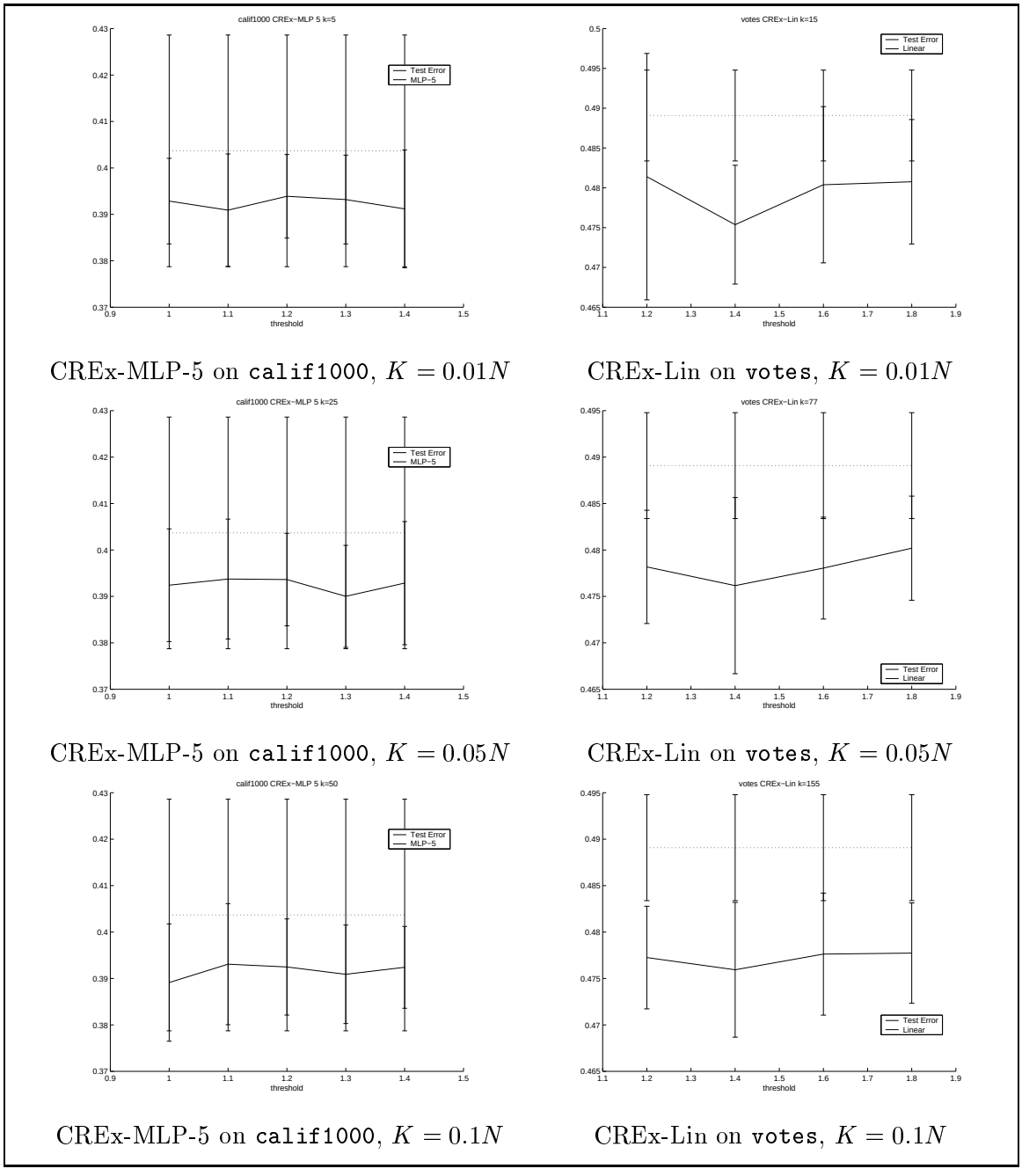


CREx-MLP-5 on `syndata`, $K = 0.01N$ | CREx-MLP-2 on `boston`, $K = 0.01N$

CREx-MLP-5 on `syndata`, $K = 0.05N$ | CREx-MLP-2 on `boston`, $K = 0.05N$

CREx-MLP-5 on `syndata`, $K = 0.1N$ | CREx-MLP-2 on `boston`, $K = 0.1N$

Figure A.29. C-REx thresholds with clustering on `syndata` and `boston`

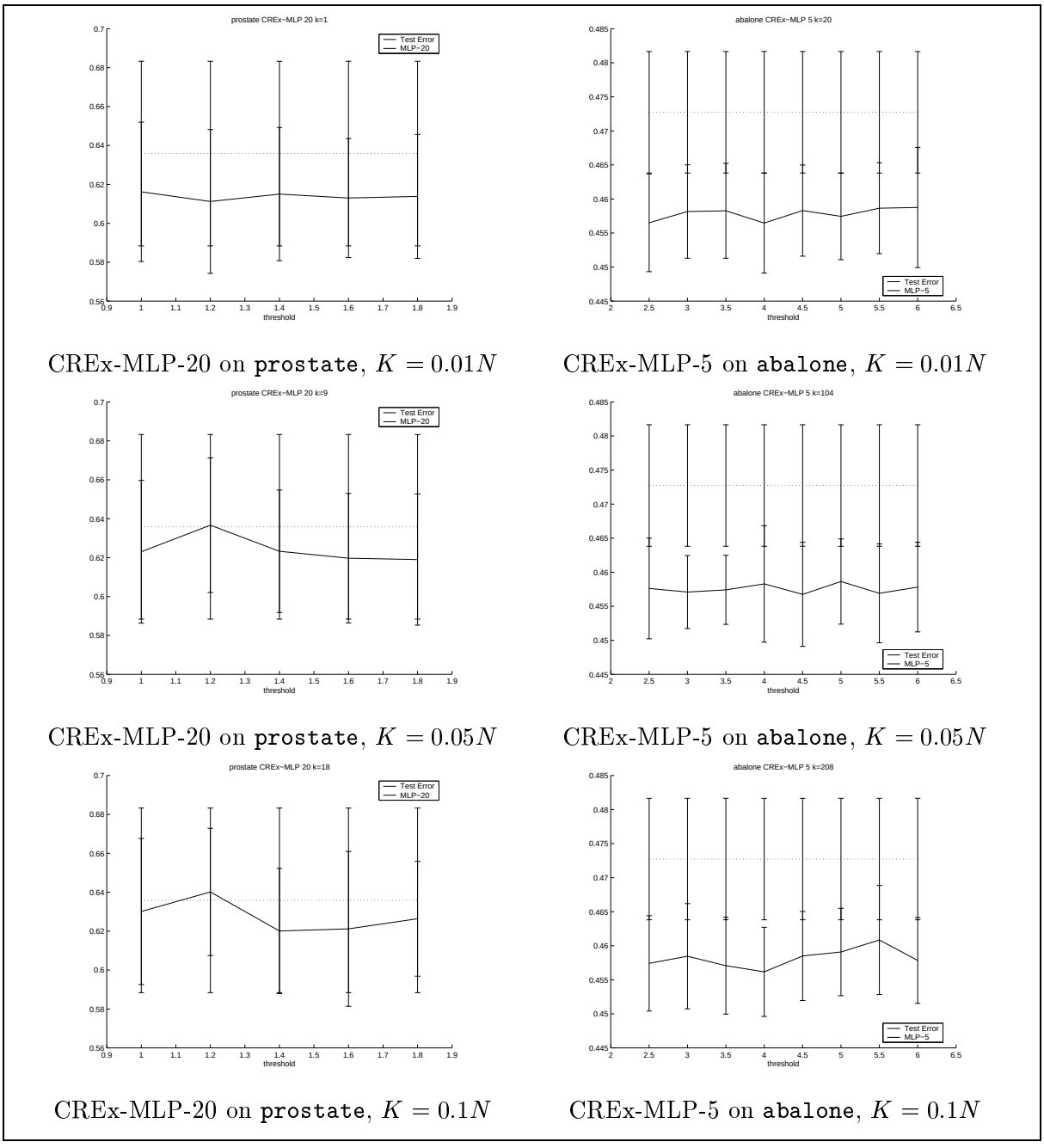


CREx-MLP-5 on `calif1000`, $K = 0.01N$ | CREx-Lin on `votes`, $K = 0.01N$

CREx-MLP-5 on `calif1000`, $K = 0.05N$ | CREx-Lin on `votes`, $K = 0.05N$

CREx-MLP-5 on `calif1000`, $K = 0.1N$ | CREx-Lin on `votes`, $K = 0.1N$

Figure A.30. C-REx thresholds with clustering on `calif1000` and `votes`

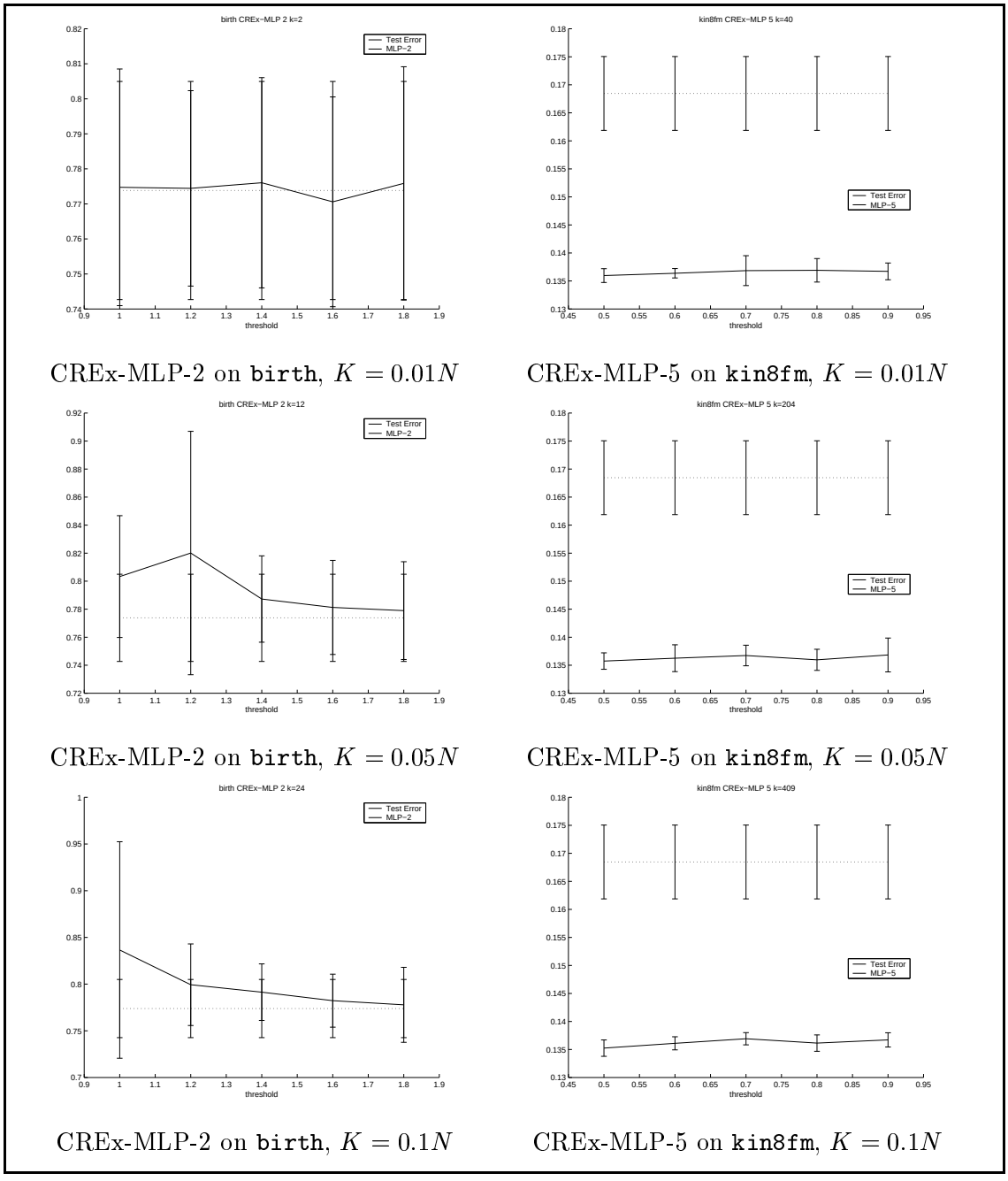


CREx-MLP-20 on `prostate`, $K = 0.01N$ | CREx-MLP-5 on `abalone`, $K = 0.01N$

CREx-MLP-20 on `prostate`, $K = 0.05N$ | CREx-MLP-5 on `abalone`, $K = 0.05N$

CREx-MLP-20 on `prostate`, $K = 0.1N$ | CREx-MLP-5 on `abalone`, $K = 0.1N$

Figure A.31. C-REx thresholds with clustering on `prostate` and `abalone`

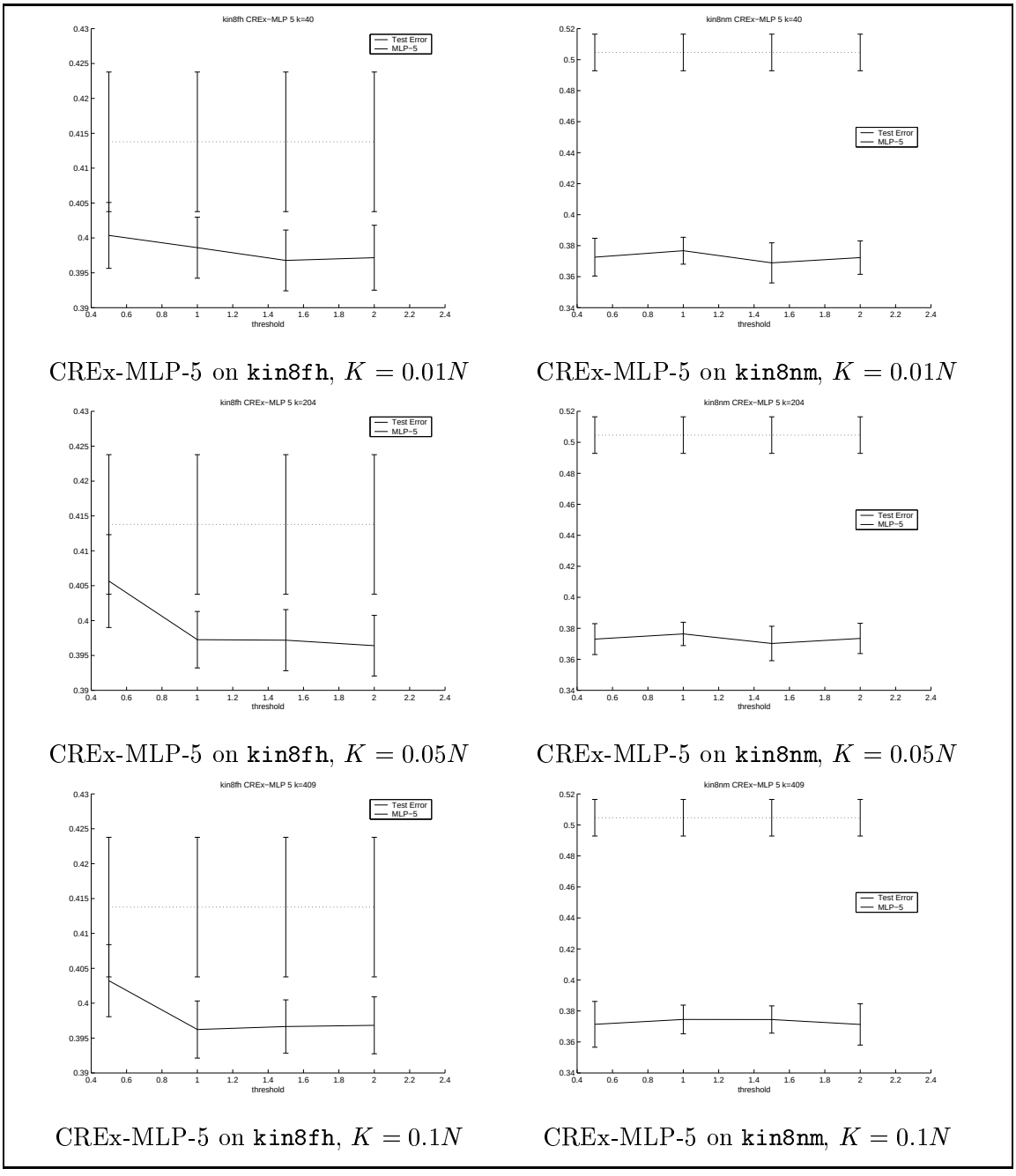


CREx-MLP-2 on `birth`, $K = 0.01N$ | CREx-MLP-5 on `kin8fm`, $K = 0.01N$

CREx-MLP-2 on `birth`, $K = 0.05N$ | CREx-MLP-5 on `kin8fm`, $K = 0.05N$

CREx-MLP-2 on `birth`, $K = 0.1N$ | CREx-MLP-5 on `kin8fm`, $K = 0.1N$

Figure A.32. C-REx thresholds with clustering on `birth` and `kin8fm`



CREx-MLP-5 on `kin8fh`, $K = 0.01N$

CREx-MLP-5 on `kin8nm`, $K = 0.01N$

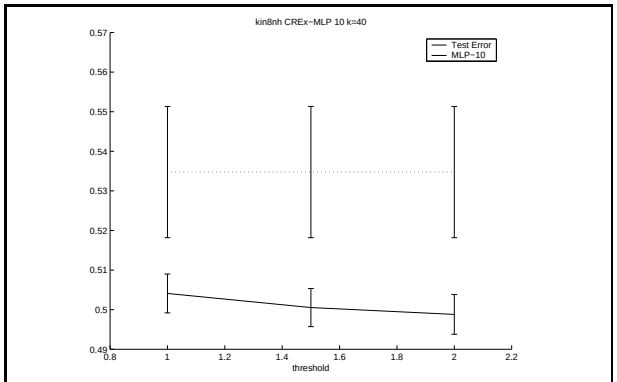
CREx-MLP-5 on `kin8fh`, $K = 0.05N$

CREx-MLP-5 on `kin8nm`, $K = 0.05N$

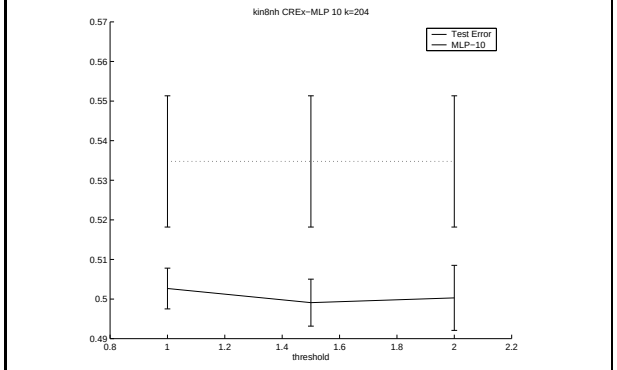
CREx-MLP-5 on `kin8fh`, $K = 0.1N$

CREx-MLP-5 on `kin8nm`, $K = 0.1N$

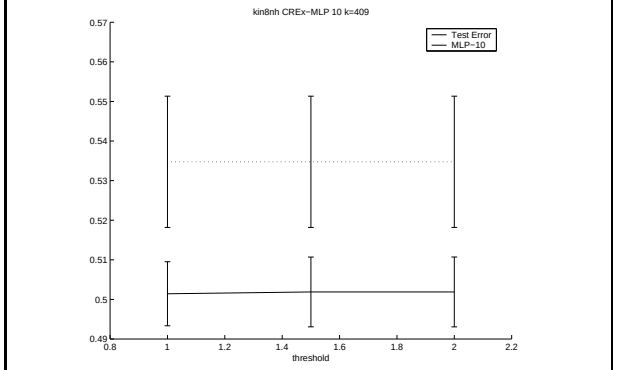
Figure A.33. C-REx thresholds with clustering on `kin8fh` and `kin8nm`



CREx-MLP-10 on `kin8nh`, $K = 0.01N$



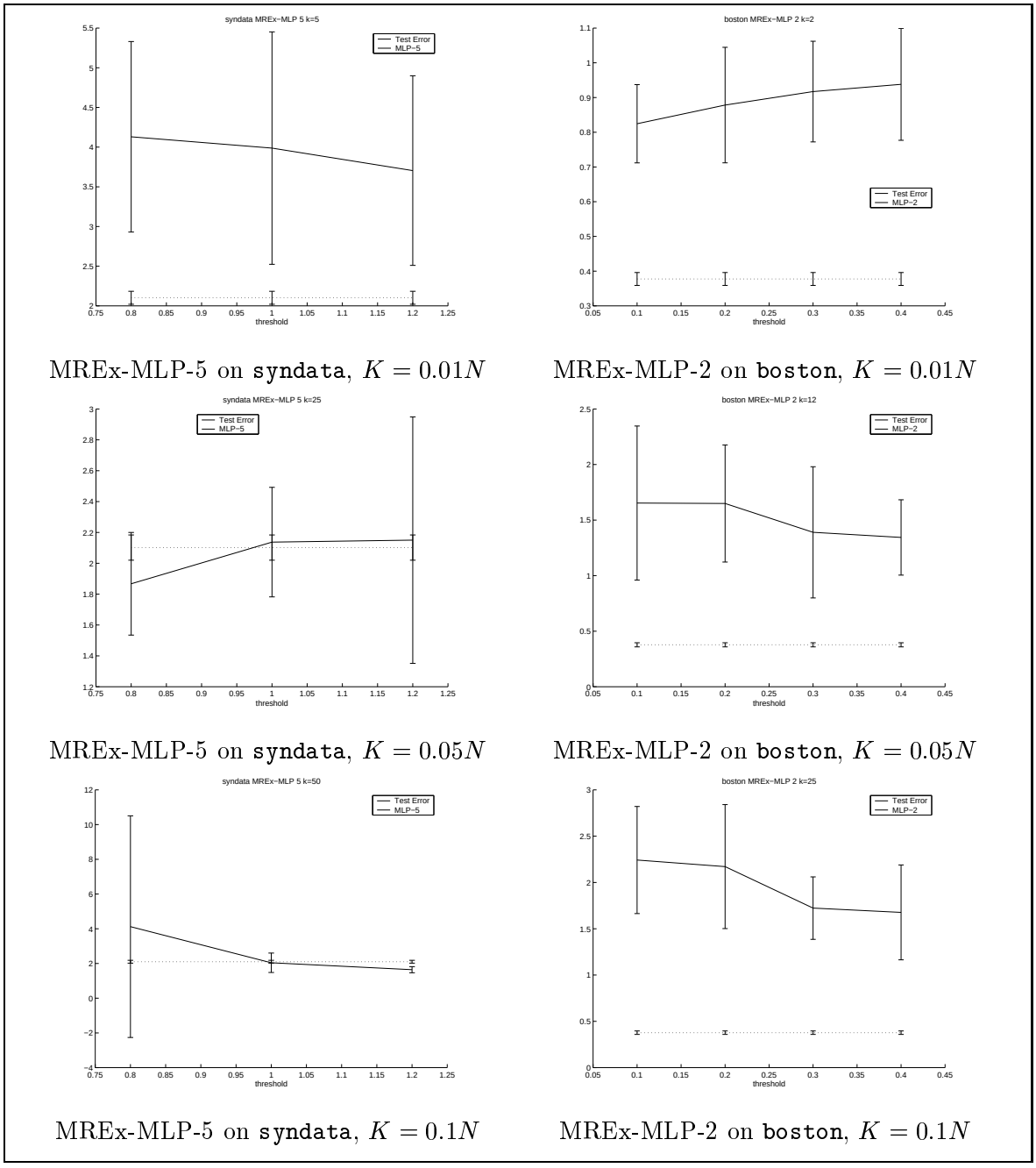
CREx-MLP-10 on `kin8nh`, $K = 0.05N$



CREx-MLP-10 on `kin8nh`, $K = 0.1N$

Figure A.34. C-REx thresholds with clustering on `kin8nh`

A.3.4. M-REx Thresholds with Clustering

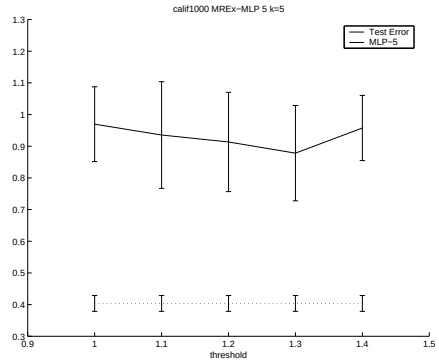


MREx-MLP-5 on `syndata`, $K = 0.01N$ | MREx-MLP-2 on `boston`, $K = 0.01N$

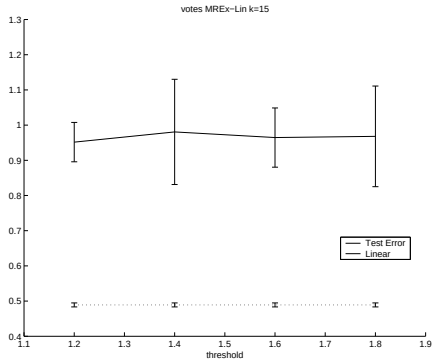
MREx-MLP-5 on `syndata`, $K = 0.05N$ | MREx-MLP-2 on `boston`, $K = 0.05N$

MREx-MLP-5 on `syndata`, $K = 0.1N$ | MREx-MLP-2 on `boston`, $K = 0.1N$

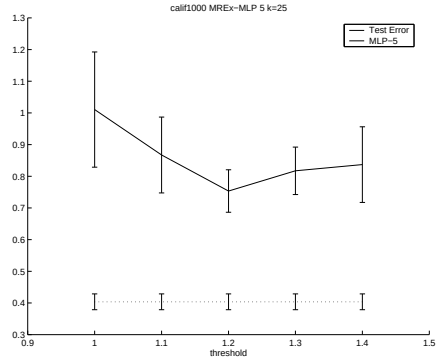
Figure A.35. M-REx thresholds with clustering on `syndata` and `boston`



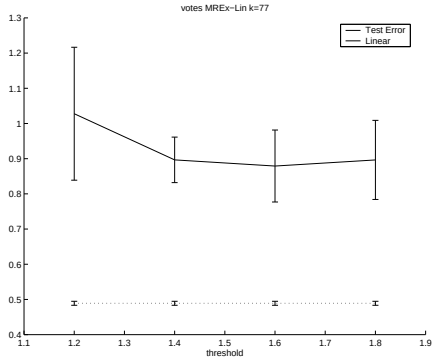
MREx-MLP-5 on `calif1000`, $K = 0.01N$



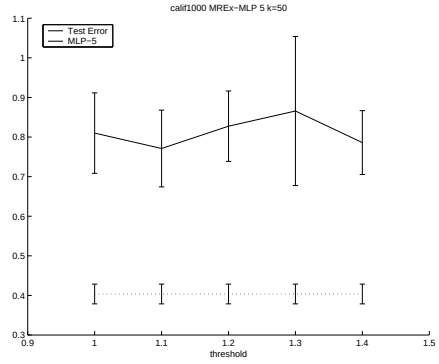
MREx-Lin on `votes`, $K = 0.01N$



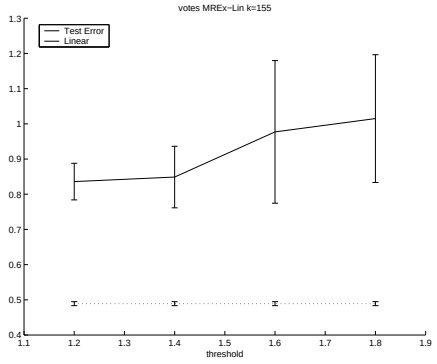
MREx-MLP-5 on `calif1000`, $K = 0.05N$



MREx-Lin on `votes`, $K = 0.05N$

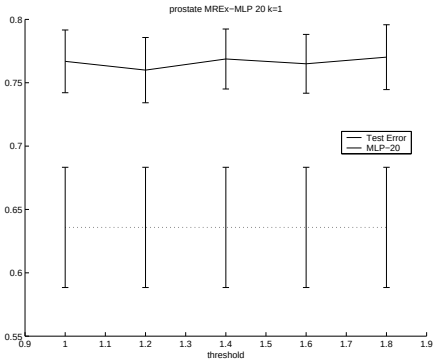


MREx-MLP-5 on `calif1000`, $K = 0.1N$

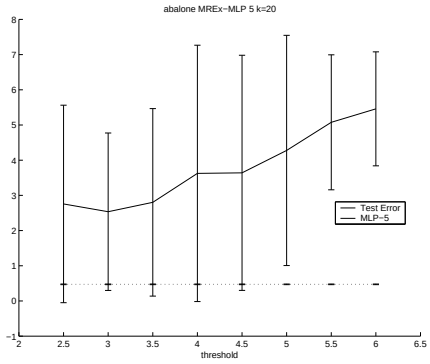


MREx-Lin on `votes`, $K = 0.1N$

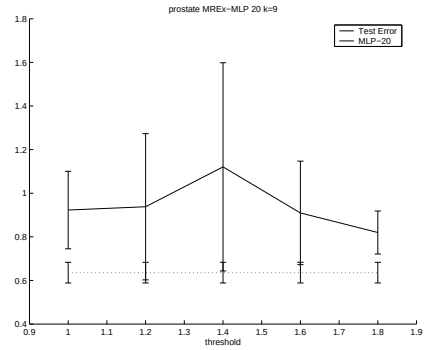
Figure A.36. M-REx thresholds with clustering on `calif1000` and `votes`



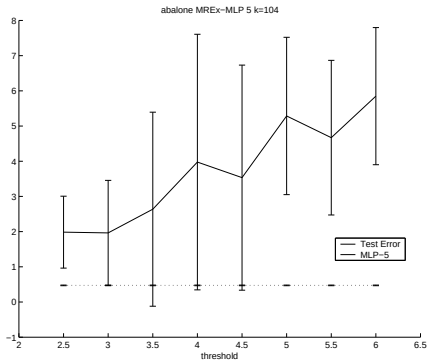
MREx-MLP-20 on `prostate`, $K = 0.01N$



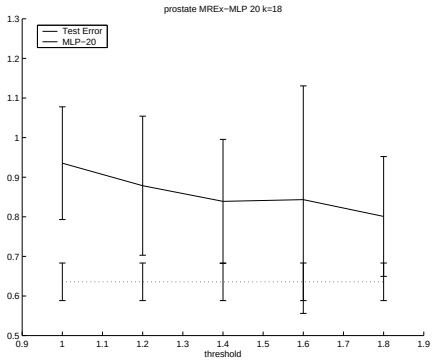
MREx-MLP-5 on `abalone`, $K = 0.01N$



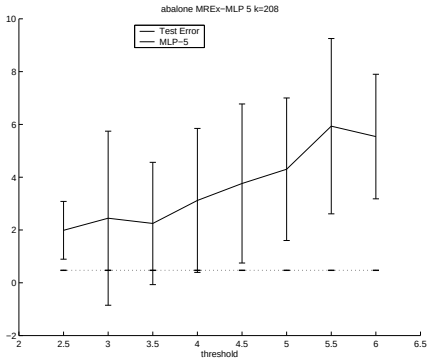
MREx-MLP-20 on `prostate`, $K = 0.05N$



MREx-MLP-5 on `abalone`, $K = 0.05N$

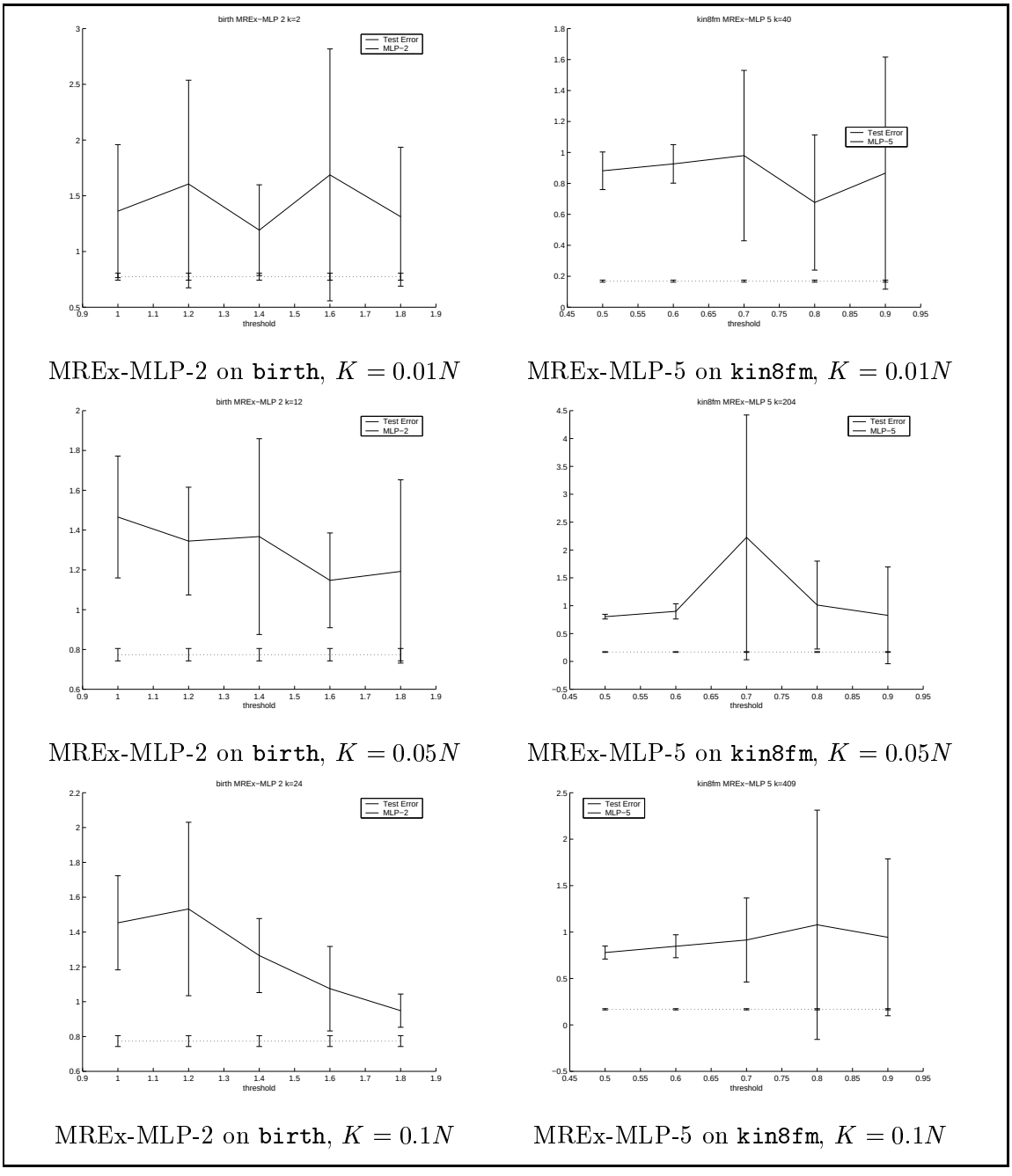


MREx-MLP-20 on `prostate`, $K = 0.1N$



MREx-MLP-5 on `abalone`, $K = 0.1N$

Figure A.37. M-REx thresholds with clustering on `prostate` and `abalone`

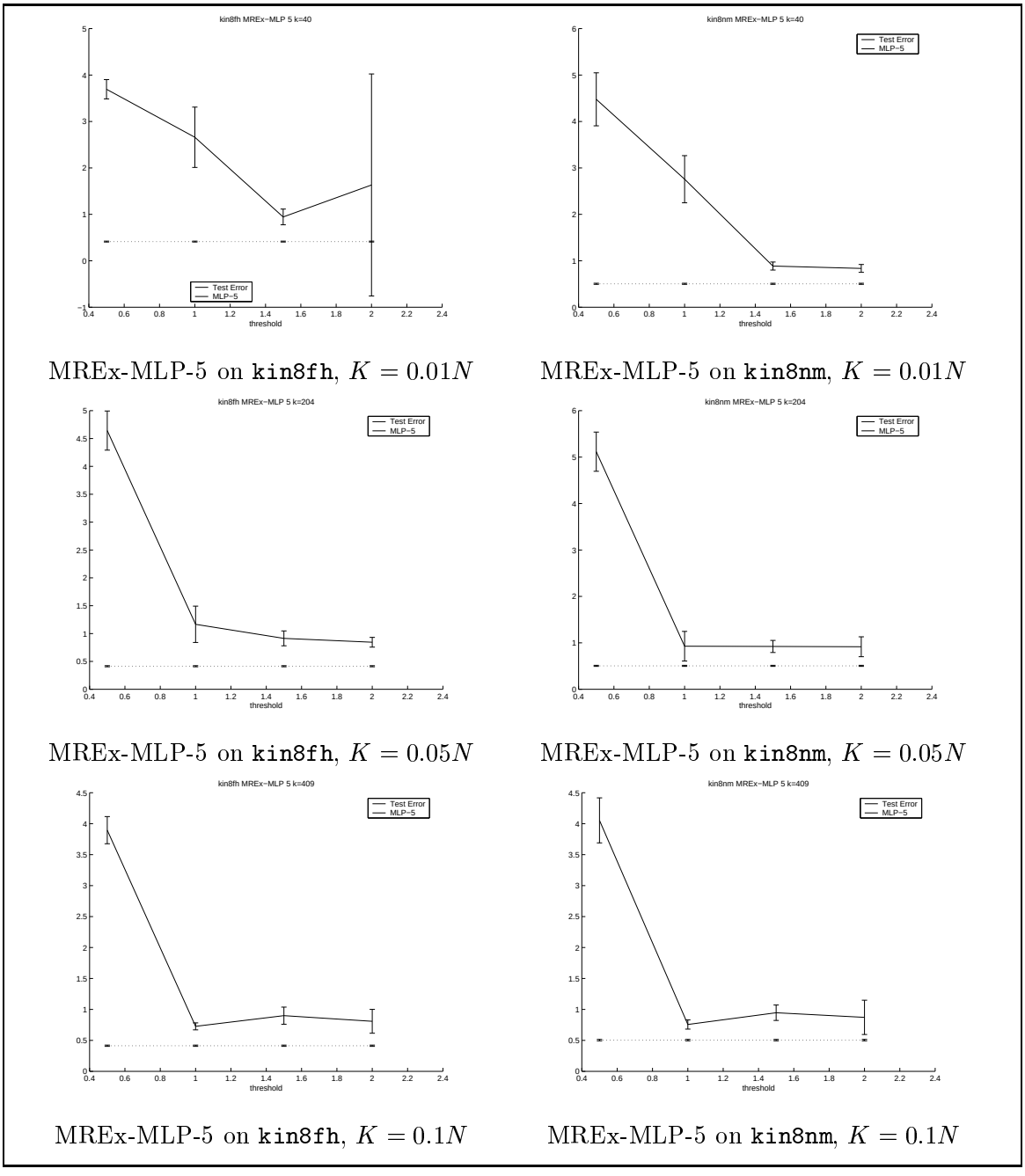


MREx-MLP-2 on `birth`, $K = 0.01N$ | MREx-MLP-5 on `kin8fm`, $K = 0.01N$

MREx-MLP-2 on `birth`, $K = 0.05N$ | MREx-MLP-5 on `kin8fm`, $K = 0.05N$

MREx-MLP-2 on `birth`, $K = 0.1N$ | MREx-MLP-5 on `kin8fm`, $K = 0.1N$

Figure A.38. M-REx thresholds with clustering on `birth` and `kin8fm`



MREx-MLP-5 on `kin8fh`, $K = 0.01N$ | MREx-MLP-5 on `kin8nm`, $K = 0.01N$

MREx-MLP-5 on `kin8fh`, $K = 0.05N$ | MREx-MLP-5 on `kin8nm`, $K = 0.05N$

MREx-MLP-5 on `kin8fh`, $K = 0.1N$ | MREx-MLP-5 on `kin8nm`, $K = 0.1N$

Figure A.39. M-REx thresholds with clustering on `kin8fh` and `kin8nm`

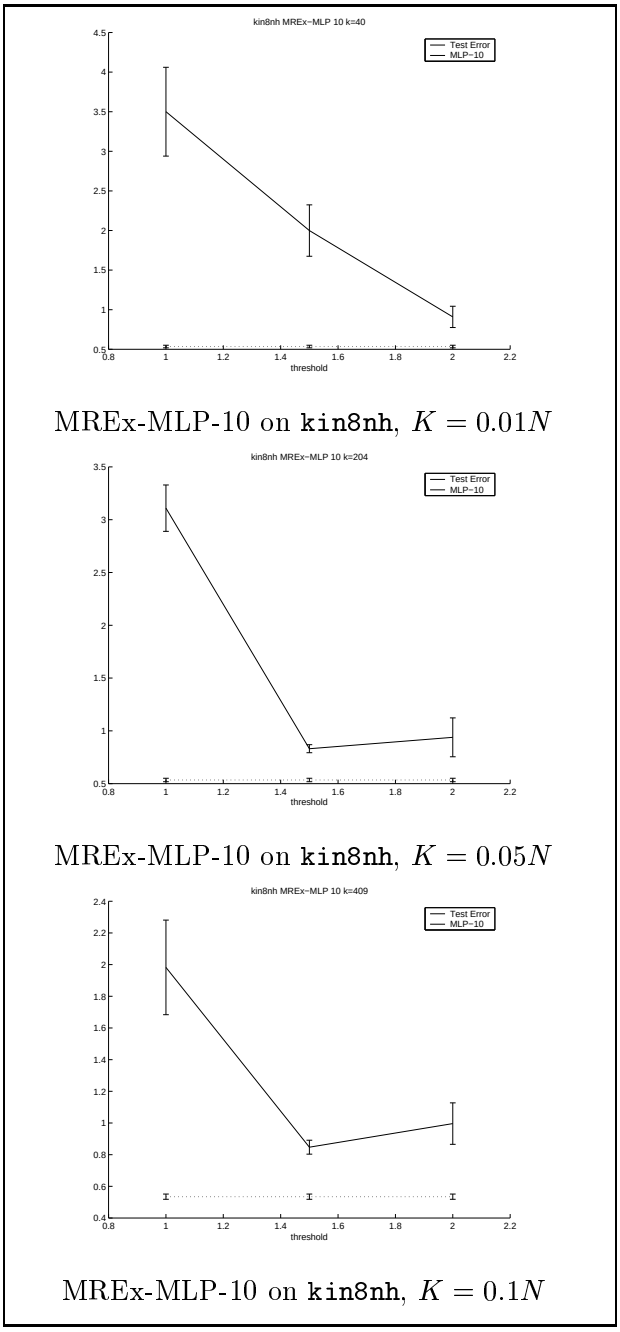


Figure A.40. M-REx thresholds with clustering on kin8nh

A.4. Bagging and AdaBoost Errors

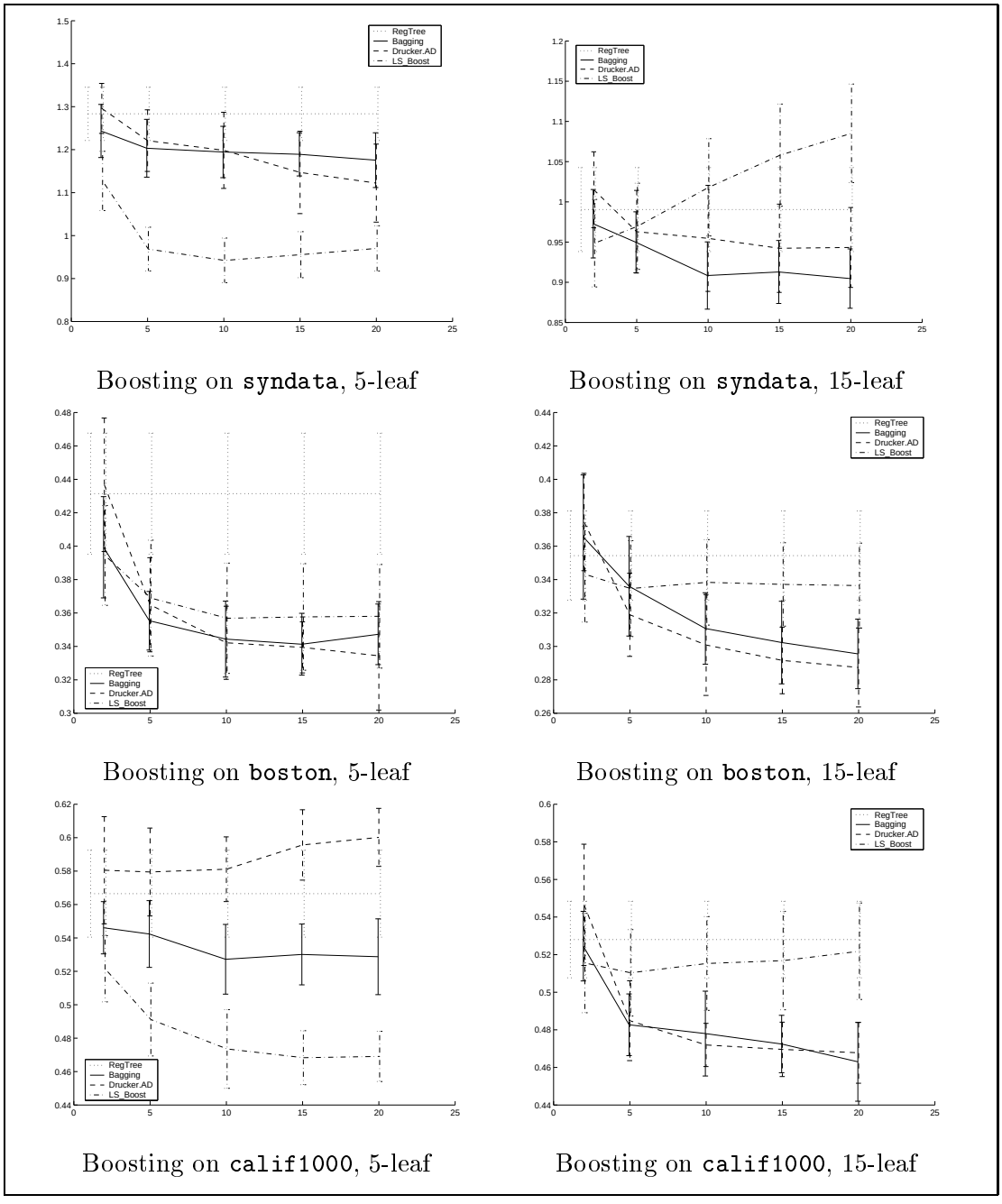


Figure A.41. Bagging and AdaBoost on `syndata`, `boston` and `calif1000`

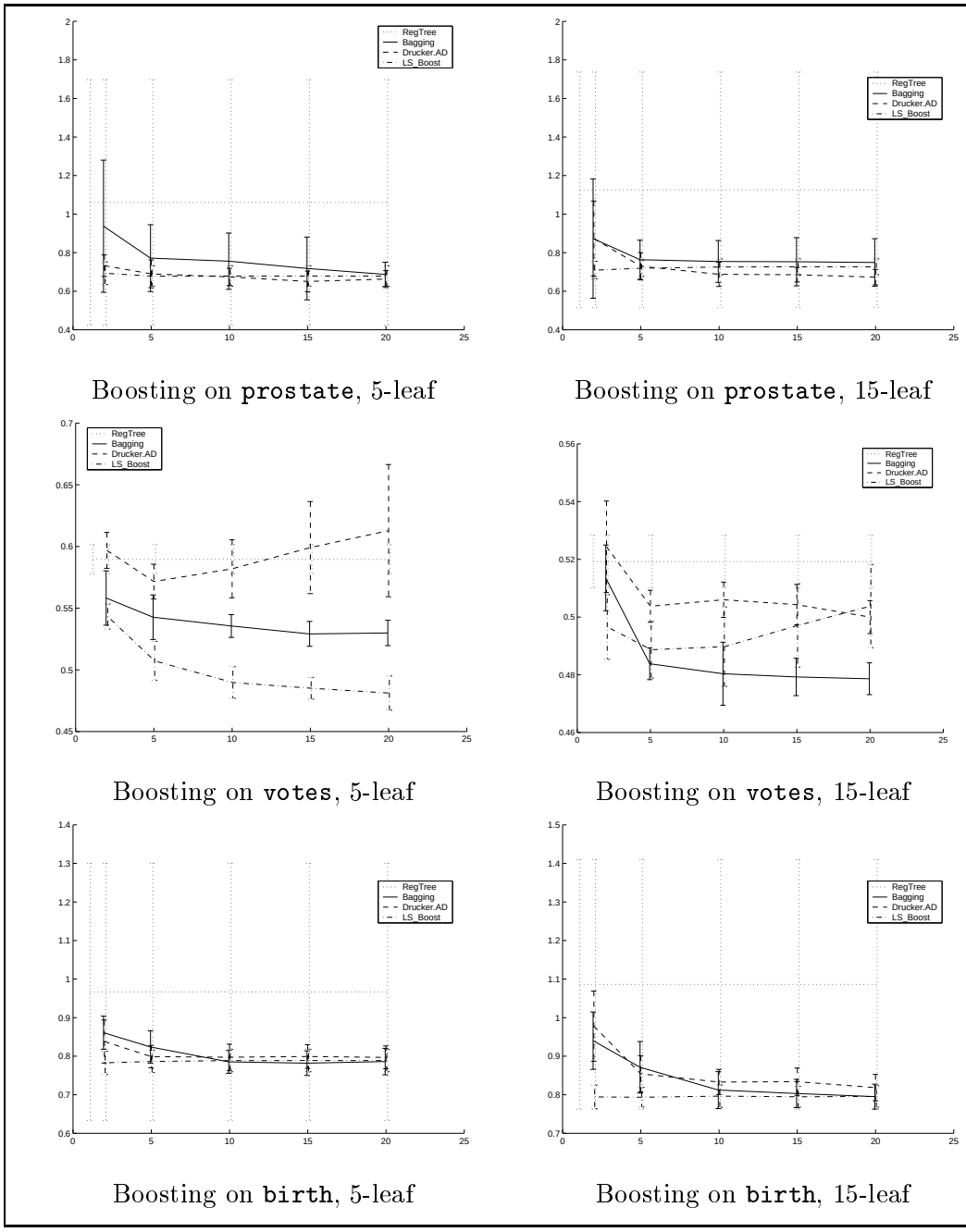


Figure A.42. Bagging and AdaBoost on `prostate`, `votes` and `birth`

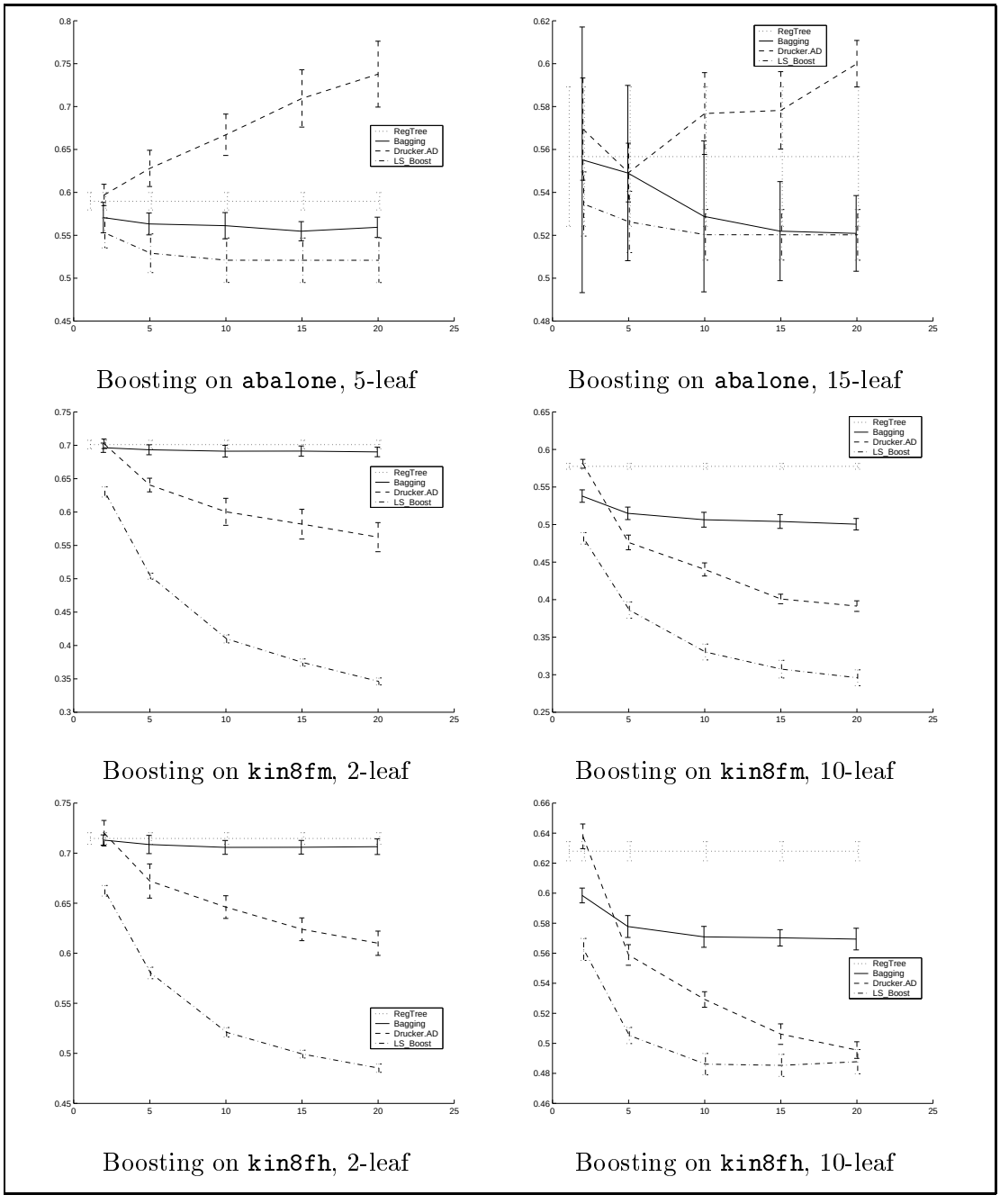


Figure A.43. Bagging and AdaBoost on `abalone`, `kin8fm` and `kin8fh`

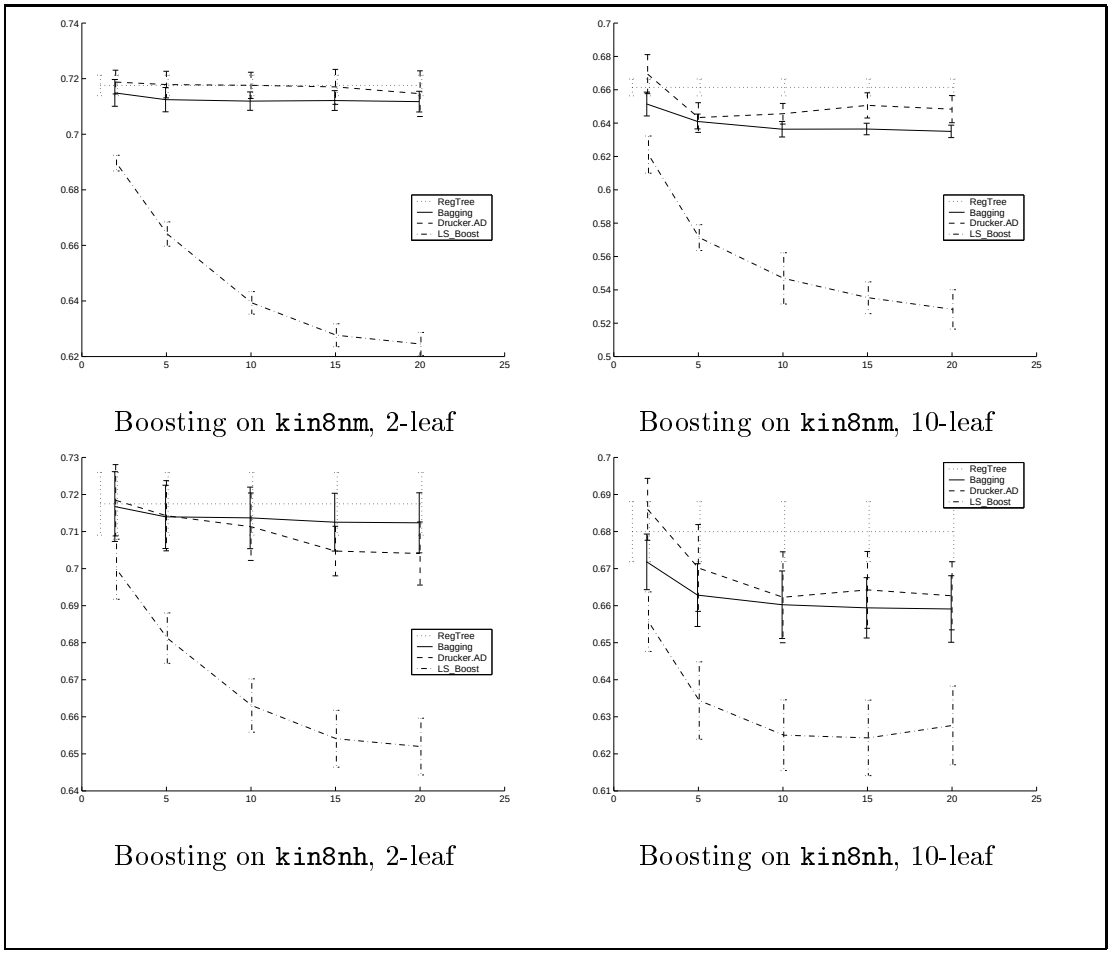


Figure A.44. Bagging and AdaBoost on `kin8nm` and `kin8nh`

A.5. Time Complexities

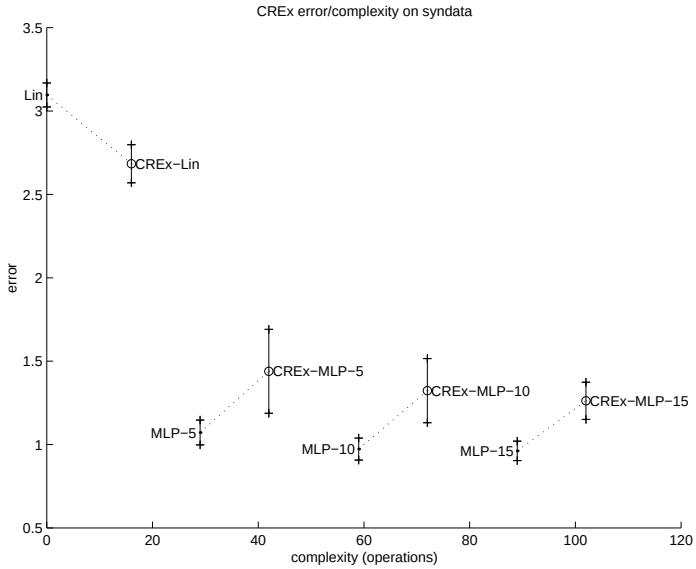


Figure A.45. Error/Complexity of C-REx on `syndata`

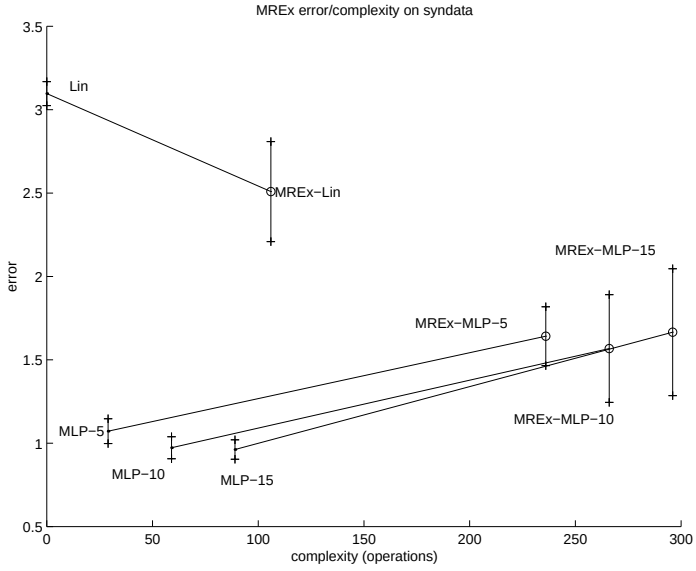


Figure A.46. Error/Complexity of M-REx on `syndata`

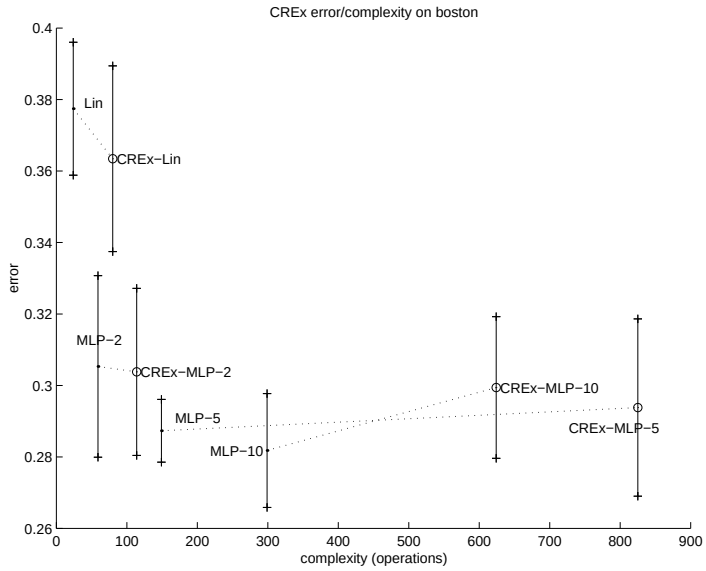


Figure A.47. Error/Complexity of C-REx on `boston`

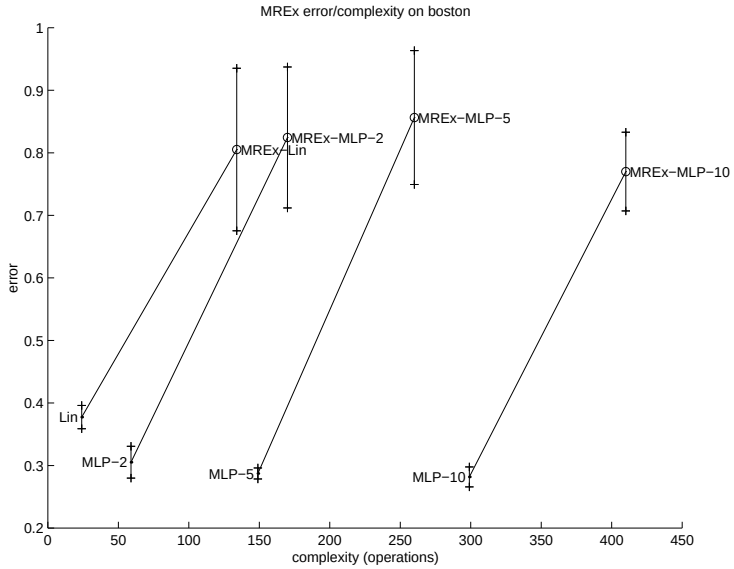


Figure A.48. Error/Complexity of M-REx on `boston`

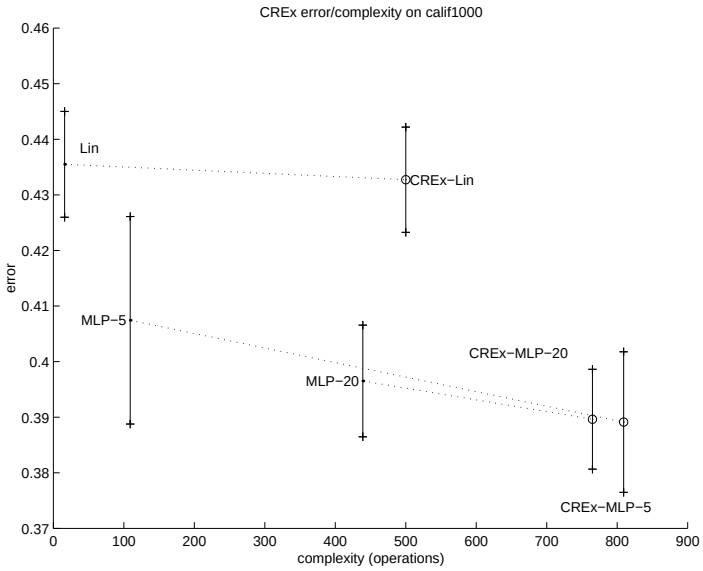


Figure A.49. Error/Complexity of C-REx on `calif1000`

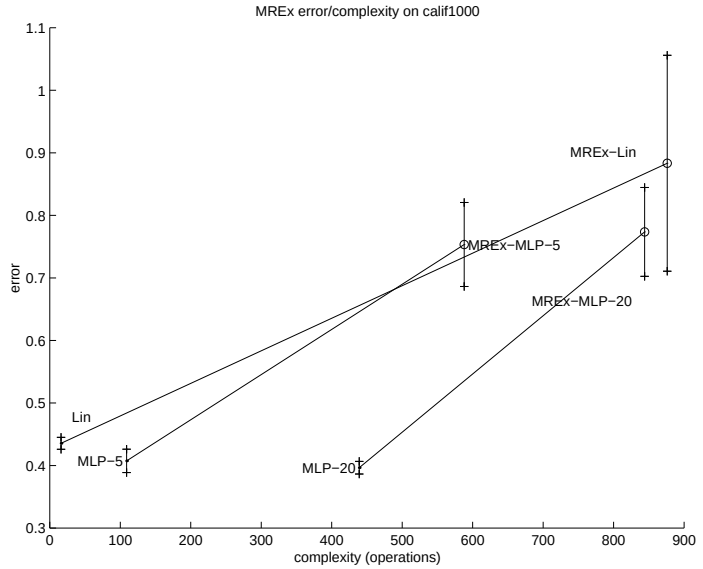


Figure A.50. Error/Complexity of M-REx on `calif1000`

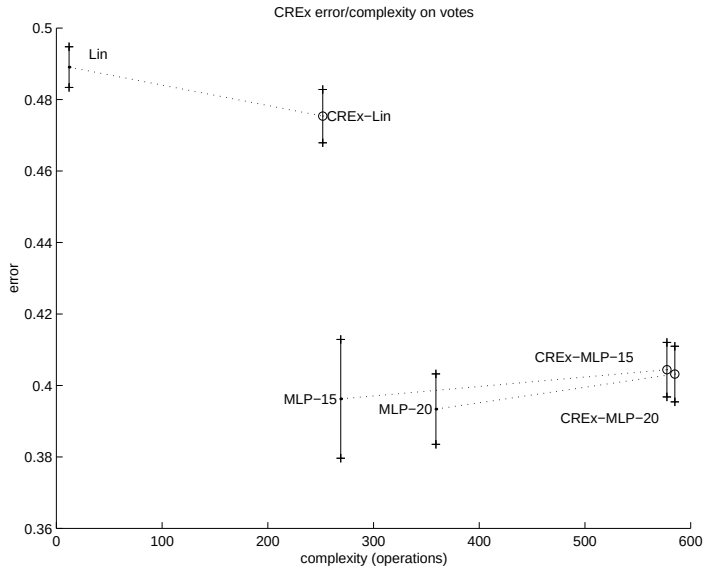


Figure A.51. Error/Complexity of C-REx on `votes`

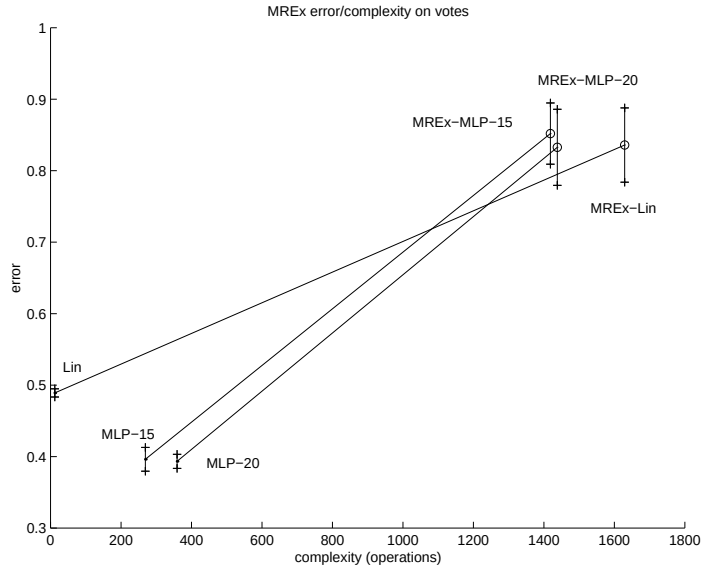


Figure A.52. Error/Complexity of M-REx on `votes`

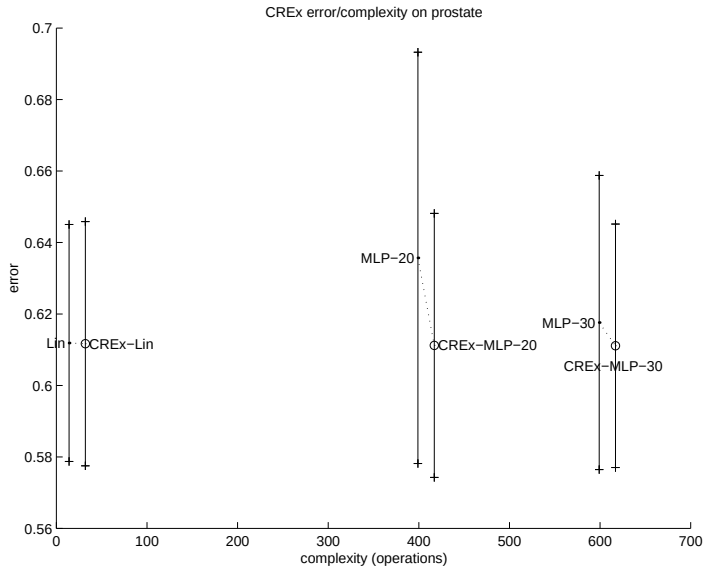


Figure A.53. Error/Complexity of C-REx on `prostate`

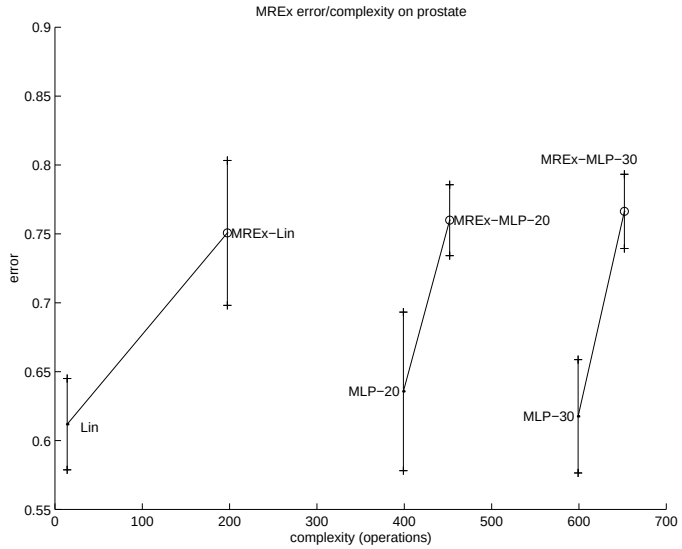


Figure A.54. Error/Complexity of M-REx on `prostate`

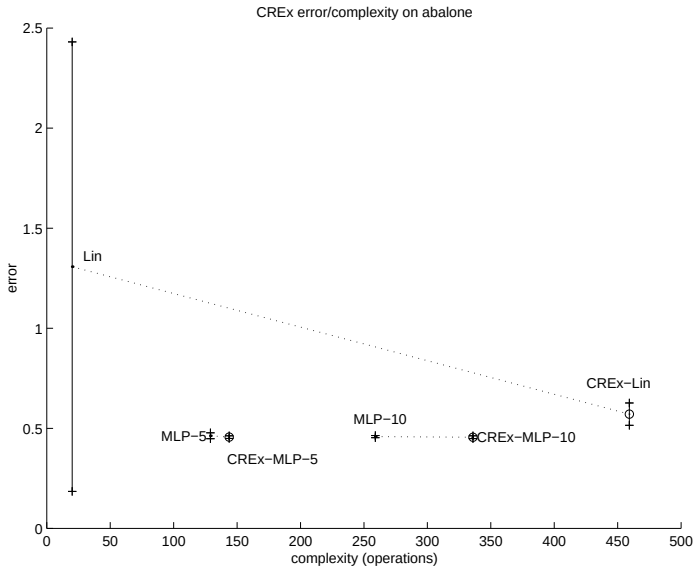


Figure A.55. Error/Complexity of C-REx on `abalone`

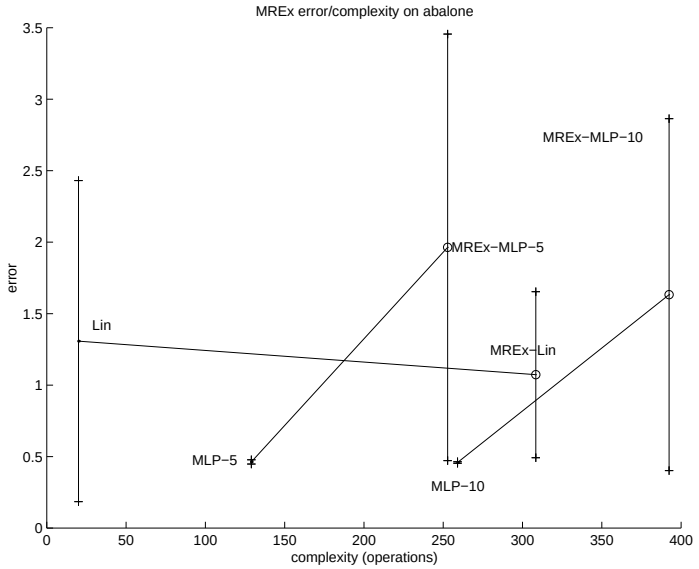


Figure A.56. Error/Complexity of M-REx on `abalone`

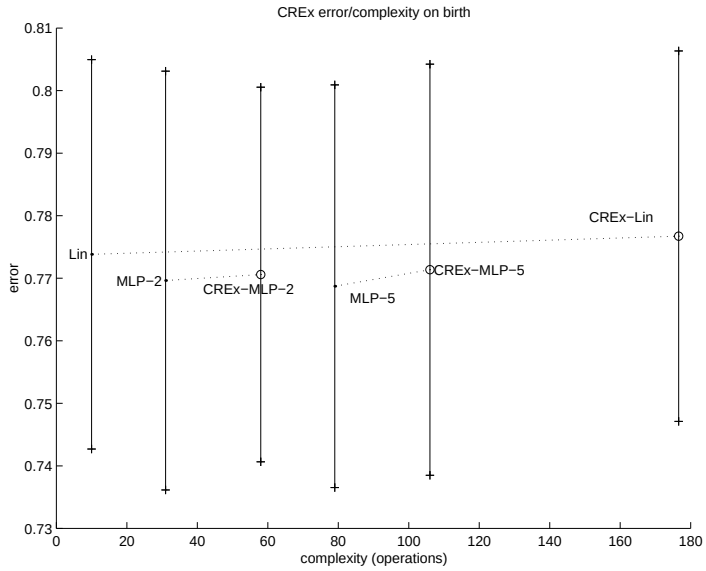


Figure A.57. Error/Complexity of C-REx on `birth`

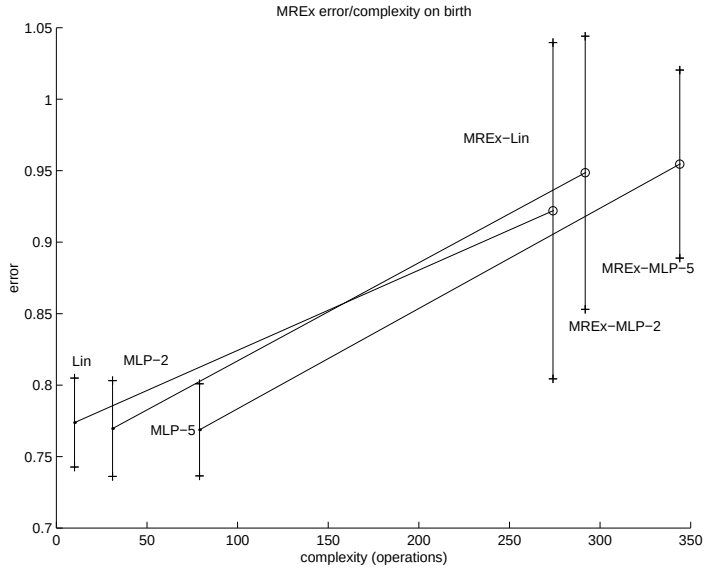


Figure A.58. Error/Complexity of M-REx on `birth`

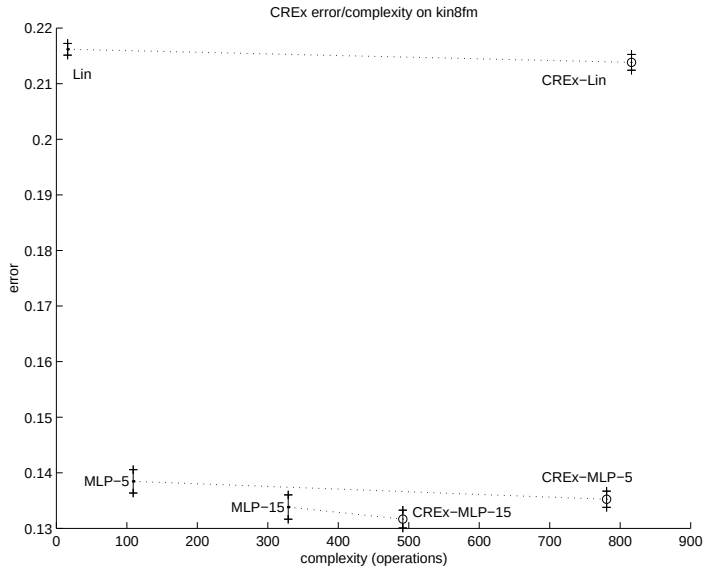


Figure A.59. Error/Complexity of C-REx on `kin8fm`

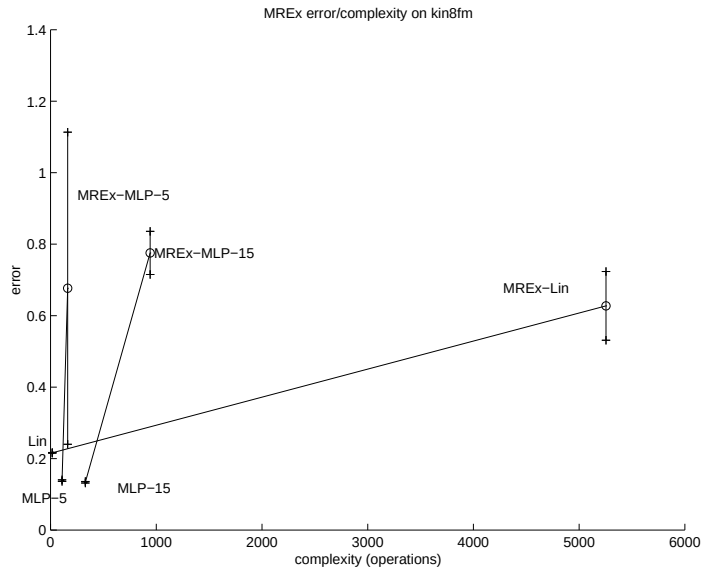


Figure A.60. Error/Complexity of M-REx on `kin8fm`

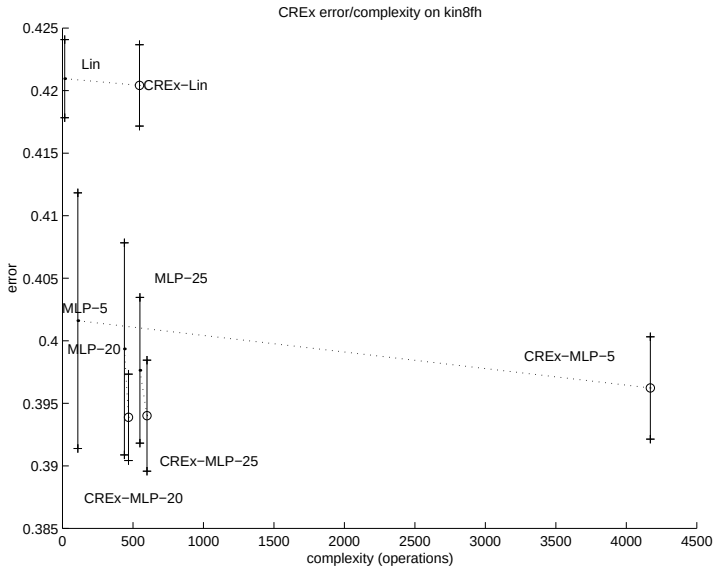


Figure A.61. Error/Complexity of C-REx on `kin8fh`

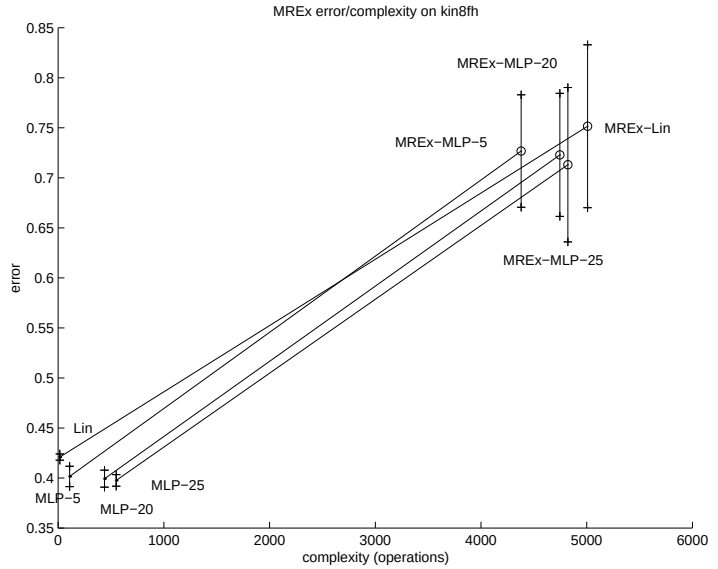


Figure A.62. Error/Complexity of M-REx on `kin8fh`

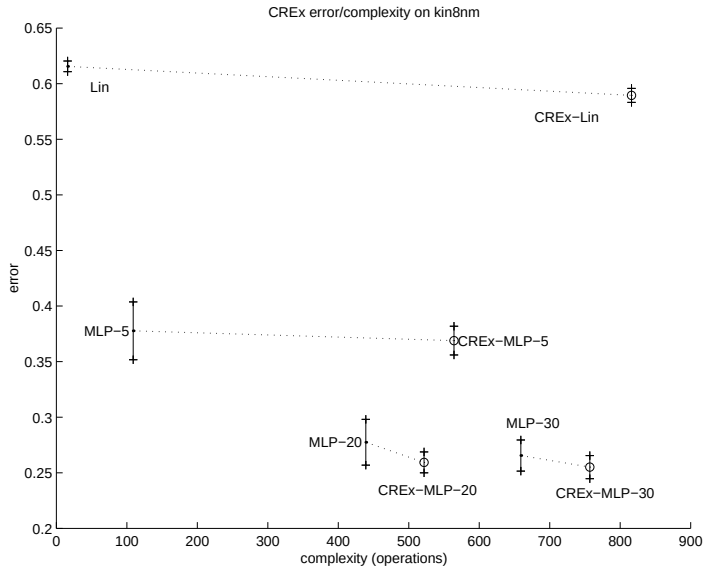


Figure A.63. Error/Complexity of C-REx on `kin8nm`

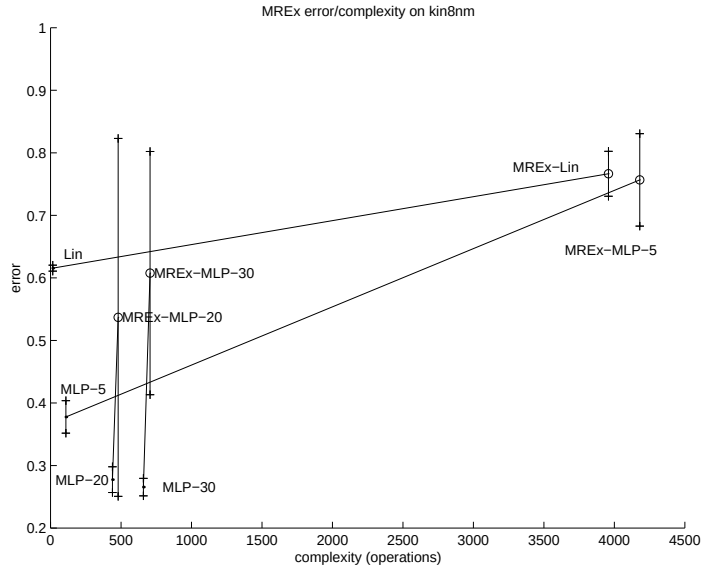


Figure A.64. Error/Complexity of M-REx on `kin8nm`

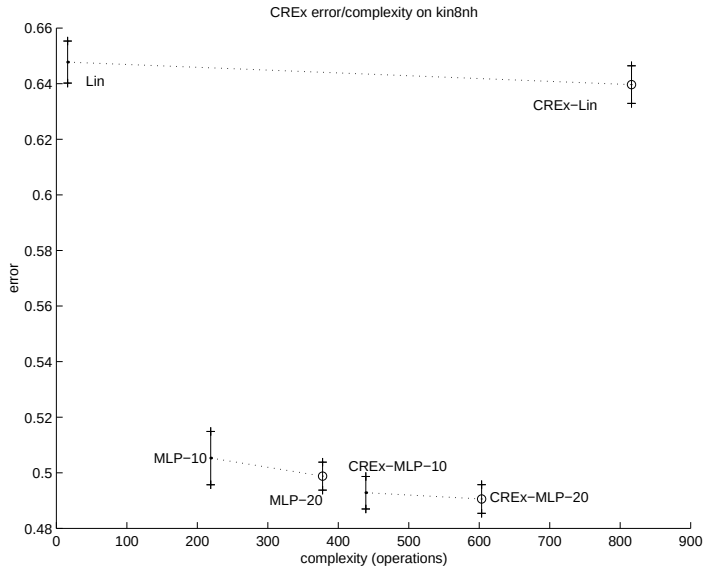


Figure A.65. Error/Complexity of C-REx on `kin8nh`

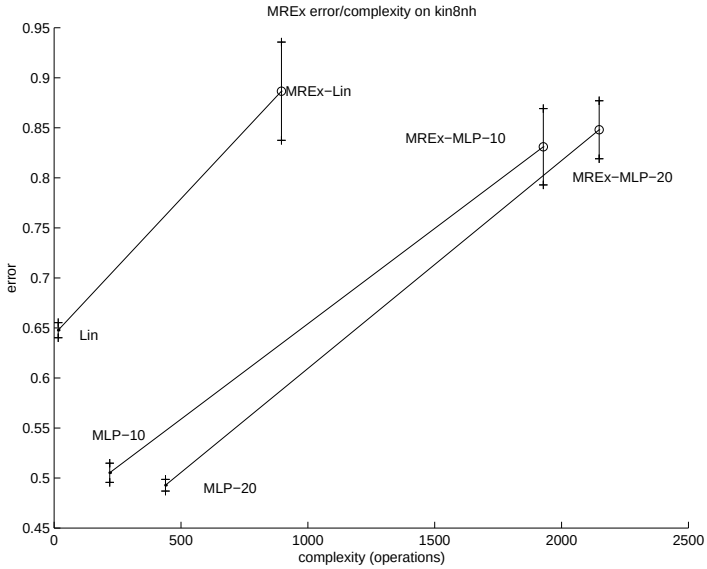


Figure A.66. Error/Complexity of M-REx on `kin8nh`

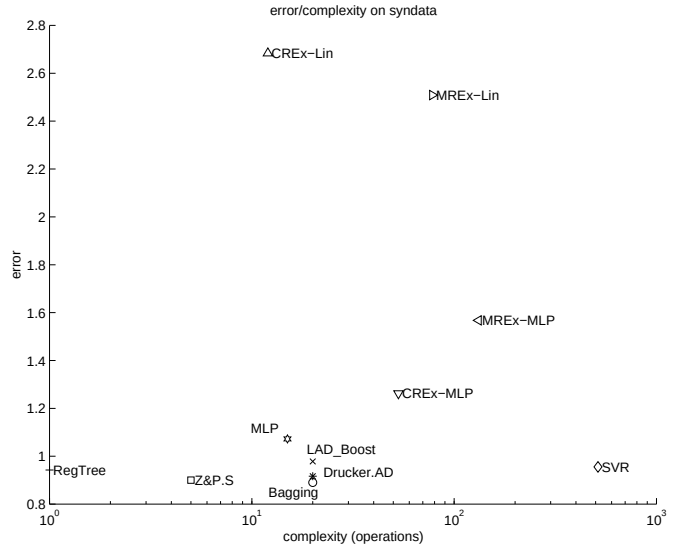


Figure A.67. Error/Complexity on `syndata`

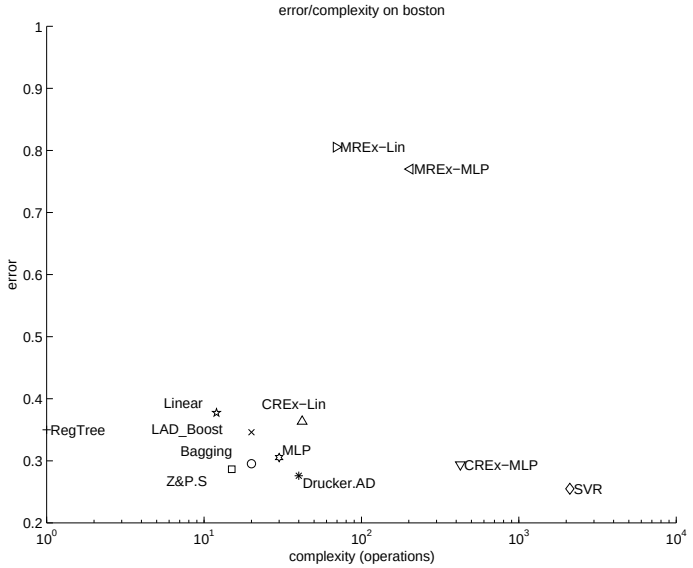


Figure A.68. Error/Complexity on `boston`

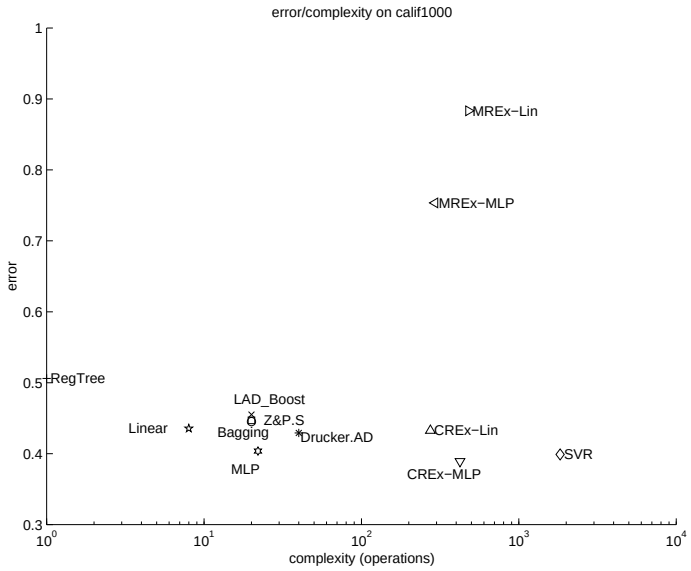


Figure A.69. Error/Complexity on `calif1000`

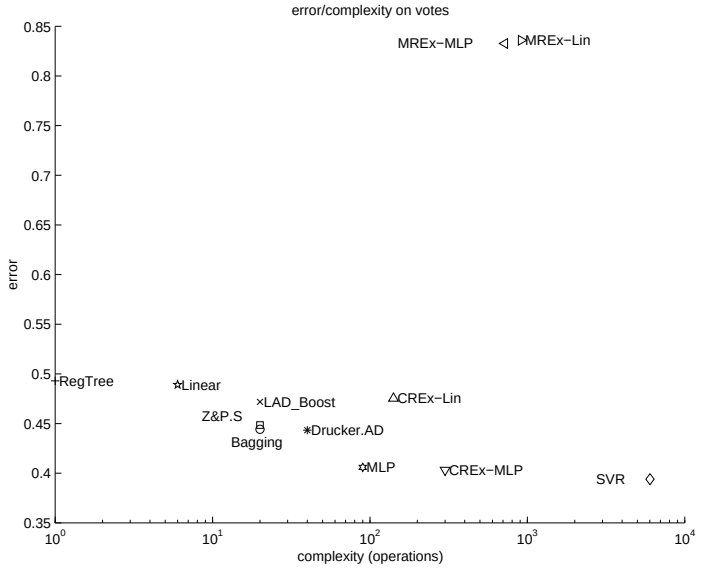


Figure A.70. Error/Complexity on `votes`

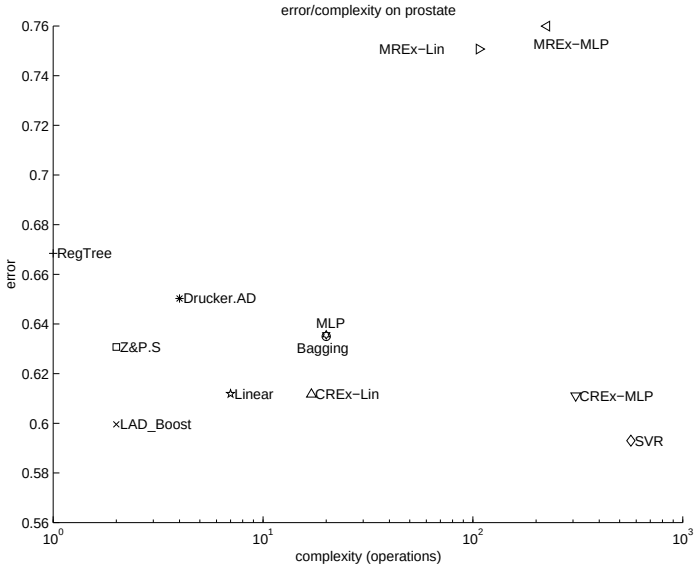


Figure A.71. Error/Complexity on `prostate`

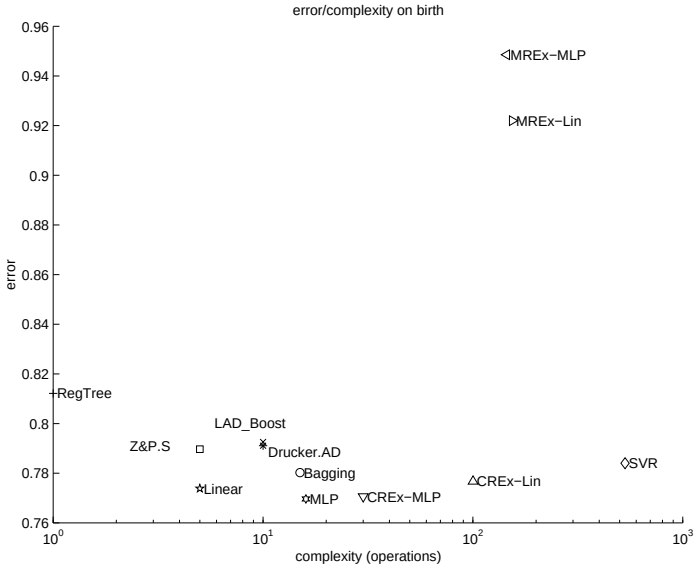


Figure A.72. Error/Complexity on `birth`

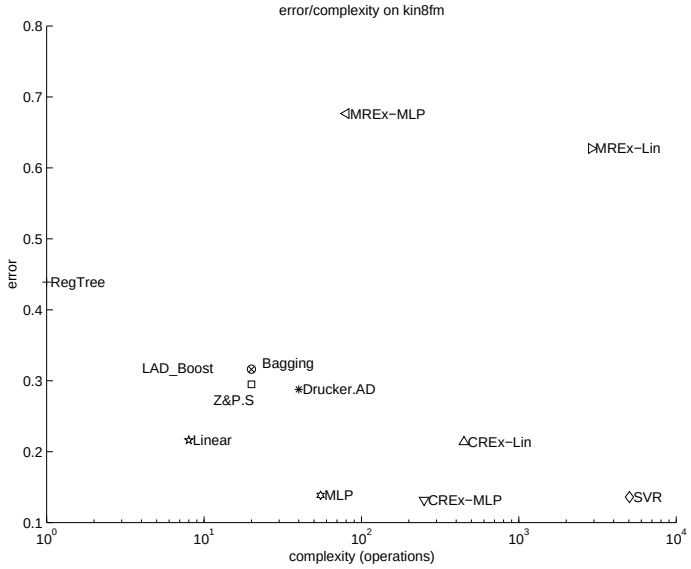


Figure A.73. Error/Complexity on `kin8fm`

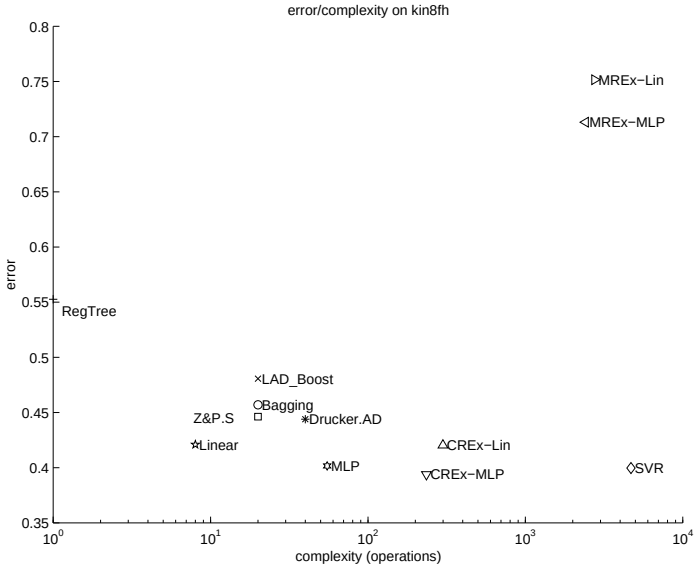


Figure A.74. Error/Complexity on `kin8fh`

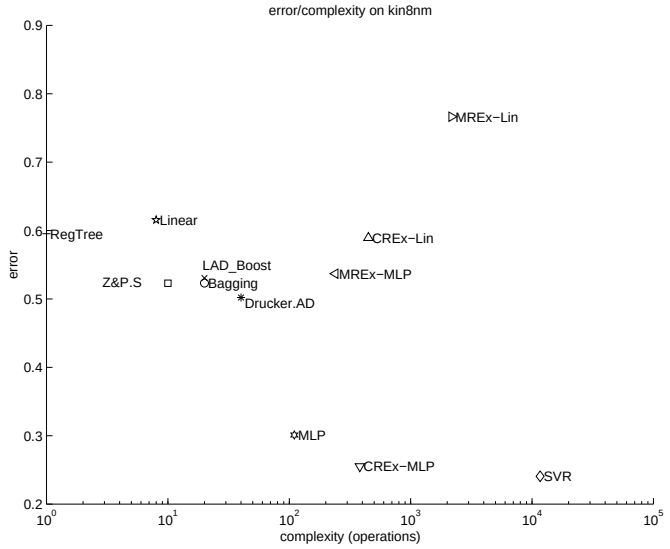


Figure A.75. Error/Complexity on `kin8nm`

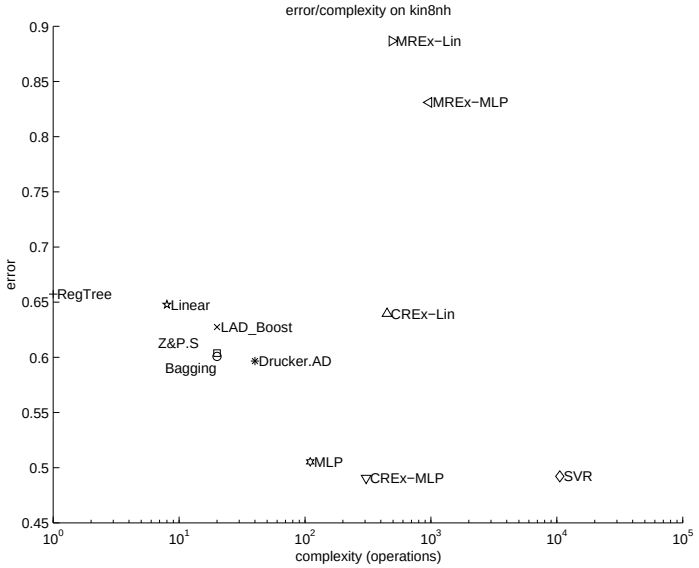


Figure A.76. Error/Complexity on `kin8nh`

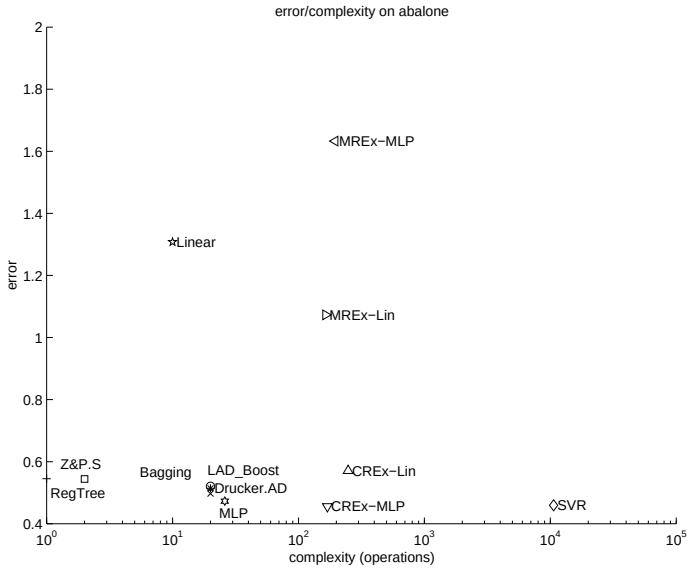


Figure A.77. Error/Complexity on `abalone`

REFERENCES

1. Alpaydın, E., *REx: Learning A Rule and Exceptions*, Tech. Rep. TR-97-040, International Computer Science Institute, Berkeley, CA, 1997.

2. Alpaydın, E. and C. Kaynak, "Cascading Classifiers", *Kybernetika*, Vol. 34, pp. 369–374, 1998.

3. Kaynak, C. and E. Alpaydın, "Multistage Cascading of Multiple Classifiers: One Man's Noise is Another Man's Data", *Proc. 17th International Conference on Machine Learning*, pp. 455–462, Morgan Kaufmann, San Francisco, CA, 2000.

4. Kaynak, C. and E. Alpaydın, *Cascading Rules and Exceptions*, Tech. rep., Boğaziçi University, Dept. of Computer Engineering, 2001.

5. Kaynak, C., *Combining Multiple Machine Learning Algorithms to Learn Rules and Exceptions*, Ph.D. thesis, Boğaziçi University, Dept. of Computer Engineering, 2002.

6. Breiman, L., "Bagging Predictors", *Machine Learning*, Vol. 24, No. 2, pp. 123–140, 1996.

7. Freund, Y. and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting", *European Conference on Computational Learning Theory*, pp. 23–37, 1995.

8. Freund, Y. and R. E. Schapire, "Experiments with a New Boosting Algorithm", *International Conference on Machine Learning*, pp. 148–156, 1996.

9. Drucker, H., "Improving regressors using boosting techniques", *Proc. 14th International Conference on Machine Learning*, pp. 107–115, Morgan Kaufmann, San Francisco, CA, 1997.

10. Zemel, R. S. and T. Pitassi, "A Gradient-Based Boosting Algorithm for Regression Problems", *Advances in Neural Information Processing Systems*, Vol. 13, 2001.

11. Duffy, N. and D. Helmbold, "Leveraging for Regression", *Proc. 13th Annual Conference on Computational Learning Theory*, pp. 208–219, Morgan Kaufmann, San Francisco, CA, 2000.

12. Friedman, J. H., *Greedy Function Approximation: a Gradient Boosting Machine*, Tech. Rep. 7, Stanford University, Dept. of Statistics, 1999.

13. Rätsch, G., M. Warmuth, S. Mika, T. Onoda, S. Lemm and K.-R. Müller, "Barrier Boosting", *Proc. 13th Annual Conference on Computational Learning Theory*, 2000.

14. Ridgeway, G., D. Madigan and T. Richardson, "Boosting methodology for regression problems", *Proceedings of Artificial Intelligence and Statistics*, pp. 152–161, 1999.

15. Karush, W., *Minima of Functions of Several Variables with Inequalities as Side Constraints*, Master's thesis, Univ. of Chicago, Dept. of Mathematics, 1939.

16. Kuhn, H. W. and A. W. Tucker, "Nonlinear Programming", *Proc. 2nd Berkeley Symposium on Mathematical Statistics and Probabilistics*, pp. 481–492, Univ. of California Press, 1951.

17. Mercer, J., "Functions of Positive and Negative Type and Their Connection with the Theory of Integral Equations", *Philos. Trans. Roy. Soc. London*, Vol. A 209, pp. 415–446, 1909.

18. Smola, A. J., B. Schölkopf and K.-R. Müller, "The connection between regularization operators and support vector kernels", *Neural Networks*, Vol. 11, No. 4, pp. 637–649, 1998.

19. Burges, C. J. C., "Geometry and Invariance in Kernel Based Methods", *Advances*

in Kernel Methods – Support Vector Learning, pp. 89–116, MIT Press, Cambridge, MA., 1999.

20. Schölkopf, B., . P. Bartlett, A. Smola and R. Williamson, "Support Vector Regression with Automatic Accuracy Control", *Proc. 8th Int. Conf. on Artificial Neural Networks*, Perspectives in Neural Computing, pp. 111–116, Springer Verlag, Berlin, 1998.

21. Jacobs, R. A., M. I. Jordan, S. J. Nowlan and G. E. Hinton, "Adaptive Mixtures of Local Experts", *Neural Computation*, Vol. 3, No. 1, pp. 79–87, 1991.

22. Blake, C. and P. M. Murphy, "UCI Repository of Machine Learning Databases", `http://www.ics.uci.edu/ mlearn/MLRepository.html`.

23. Hosmer, D. and S. Lemeshow, *Applied Logistic Regression*, John Wiley & Sons Inc., second edn., 2000.

24. Chang, C.-C. and C.-J. Lin, "LIBSVM: a Library for Support Vector Machines (Version 2.31)", `http://citeseer.nj.nec.com/chang01libsvm.html`.

25. Alpaydın, E., "Combined 5×2cv F Test for Comparing Supervised Classification Learning Algorithms", *Neural Computation*, Vol. 11, No. 8, pp. 1885–1992, 1999.