Machine Learning Approaches to Estimating Software Development Effort

Krishnamoorthy Srinivasan and Douglas Fisher, *Member, IEEE*

***Abstract*—Accurate estimation of software development effort is critical in software engineering. Underestimates lead to time pressures that may compromise full functional development and thorough testing of software. In contrast, overestimates can result in noncompetitive contract bids and/or over allocation of development resources and personnel. As a result, many models for estimating software development effort have been proposed. This article describes two methods of machine learning, which we use to build estimators of software development effort from historical data. Our experiments indicate that these techniques are competitive with traditional estimators on one dataset, but also illustrate that these methods are sensitive to the data on which they are trained. This cautionary note applies to any model-construction strategy that relies on historical data. All such models for software effort estimation should be evaluated by exploring model sensitivity on a variety of historical data.**

***Index Terms*—Software development effort, machine learning, decision trees, regression trees, and neural networks.**

I. Introduction

ACCURATE estimation of software development effort has major implications for the management of software development. If management's estimate is too low, then the software development team will be under considerable pressure to finish the product quickly, and hence the resulting software may not be fully functional or tested. Thus, the product may contain residual errors that need to be corrected during a later part of the software life cycle, in which the cost of corrective maintenance is greater. On the other hand, if a manager's estimate is too high, then too many resources will be committed to the project. Furthermore, if the company is engaged in contract software development, then too high an estimate may fail to secure a contract.

The importance of software effort estimation has motivated considerable research in recent years. Parametric models such as COCOMO [3], FUNCTION POINTS [2], and SLIM [16] "calibrate" prespecified formulas for estimating development effort from historical data. Inputs to these models may include the experience of the development team, the required reliability of the software, the programming language in which the software is to be written, and an estimate of the final number of delivered source lines of code (SLOC). In contrast, many methods of machine learning make no or minimal assumptions about the form of the function under study (e.g., development effort), but as with other approaches they depend on historical data. In particular, over a known set of *training* data, the learning algorithm constructs "rules" that fit the data, and which hopefully fit previously unseen data in a reasonable manner as well. This article illustrates machine learning approaches to estimating software development effort using an algorithm for building *regression trees* [4], and a neural-network learning approach known as BACKPROPAGATION [19]. Our experiments, using established case libraries [3], [11], indicate possible advantages of the approach relative to traditional models, but also point to limitations that motivate continued research.

Manuscript received October 1992; revised October 1993 and October 1994. Recommended by D. Wile. D. Fisher's work was supported by NASA Ames Grant NAG 2-834.

K. Srinivasan is with Personal Computer Consultants, Inc., Washington, D.C.

D. Fisher is with the Department of Computer Science, Vanderbilt University, Nashville, Tennessee (e-mail: dfisher@vuse.vanderbilt.edu).

IEEE Log Number 9408517.

II. Models for Estimating Software Development Effort

Many models have been developed to estimate software development effort. Many of these models are parametric, in that they predict development effort using a formula of fixed form that is parameterized from historical data. In preparation for later discussion we summarize three such models that were highlighted in a previous study by Kemerer [11].

Putnam [16] developed an early model known as SLIM, which estimates the cost of software by using SLOC as the major input. The underlying assumption of this model is that resource consumption, including personnel, varies with time and can be modeled with some degree of accuracy by the Rayleigh distribution:

$$R_c = \frac{t}{k^2} e^{-(t^2/2k^2)},$$

where R_c is the instananeous resource consumption, t is the time into the development effort, and k is the time at which consumption is at its peak. The parameter k and other "management parameters" are estimated by characteristics of a particular software project, notably estimated SLOC. The general relationship between inputs such as SLOC and management parameters can be determined from historical data.

The COnstructive COst MOdel (COCOMO) was developed by Boehm [3] based on a regression analysis of 63 completed projects. COCOMO relates the effort required to develop a software project (in terms of person-months) to Delivered Source Instructions (DSI). Thus, like SLIM, COCOMO assumes SLOC as a major input. If the software project is judged to

be straightforward, then the basic COCOMO model (COCOMO-basic) relates the nominal development effort (N) and DSI as follows:

$$N = 3.2 \times (KDSI)^{1.05},$$

where $KDSI$ is the DSI in 1000s. However, the prediction of the basic COCOMO model can be modified using *cost drivers*. Cost drivers are classified under four major headings relating to attributes of the product (e.g., required software reliability), computer platform (e.g., main memory limitations), personnel (e.g., analyst capability), and the project (e.g., use of modern programming practices). These factors serve to adjust the nominal effort up or down. These cost drivers and other considerations extend the basic model to intermediate and final forms.

The *Function Point* method was developed by Albrecht [2]. Function points are based on characteristics of the project that are at a higher descriptive level than SLOC, such as the number of input transaction types and number of reports. A notable advantage of this approach is that it does not rely on SLOC, which facilitates estimation early in the project life cycle (i.e., during requirements definition), and by nontechnical personnel. To count function points requires that one count user functions and then make adjustments for processing complexity. There are five types of user function that are included in the function point calculation: external input types, external output types, logical internal file types, external interface file types, and external inquiry types. In addition, there are 14 processing complexity characteristics such as transaction rates and online updating. A function point is calculated based on the number of transactions and complexity characteristics. The development effort estimate given the function point, F, is: $N = 54 \times F - 13390$.

Recently, a case-based approach called ESTOR was developed for software effort estimation. This model was developed by Vicinanza *et al.* [23] by obtaining protocols from a human expert. From a library of cases developed from expert-supplied protocols, an instance called the *source* is retrieved that is most "similar" to the *target* problem to be solved.

The solution of the most similar problem retrieved from the case library is adapted to account for differences between the source problem and the target problem using rules inferred from analysis of the human expert's protocols. An example of an adjustment rule is:

```
IF staff size of Source project is
   small, AND
   staff size of Target is large
THEN increase effort estimate of Target
   by 20%.
```

Vicinanza *et al.*, have shown that E STOR performs better than COCOMO and FUNCTION POINTS on restricted samples of problems.

In sum, there have been a variety of models developed for estimating development effort. With the exception of ESTOR these are parametric approaches that assume that an initial estimate can be provided by a formula that has been fit to historical data.

III. MACHINE LEARNING APPROACHES TO ESTIMATING DEVELOPMENT EFFORT

This section describes two machine learning strategies that we use to estimate software development effort, which we assume is measured in development months (M). In many respects this work stems from a more general methodology for developing *expert systems*. Traditionally, expert systems have been developed by extracting the rules that experts apparently use by an interview process or protocol analysis (e.g., ESTOR), but an alternate approach is to allow machine learning programs to formulate rulebases from historical data. This methodology requires historical data on which to apply learning strategies.

There are several aspects of software development effort estimation that make it amenable to machine learning analysis. Most important, previous researchers have identified at least some of the attributes relevant to software development effort estimation, and historical databases defined over these relevant attributes have been accumulated. The following sections describe two very different learning algorithms that we use to test the machine learning approach. Other research using machine learning techniques for software resource estimation are found in [5], [14], [15], [22], which we will discuss throughout the paper. In short, our work adds to the collection of machine learning techniques available to software engineers, and our analysis stresses the sensitivity of these approaches to the nature of historical data and other factors.

A. Learning Decision and Regression Trees

Many learning approaches have been developed that construct decision trees for classifying data [4], [17]. Fig. 1 illustrates a partial decision tree over Boehm's original 63 projects from which COCOMO was developed. Each project is described over dimensions such as AKDSI (i.e., adjusted delivered source instructions), TIME (i.e., the required system response time), and STOR (i.e., main memory limitations). The complete set of attributes used to describe these data is given in Appendix A. The mean of actual project development months labels each leaf of the tree. Predicting development effort for a project requires that one descend the decision tree along an appropriate path, and the leaf value along that path gives the estimate of development effort of the new project. The decision tree in Fig. 1 is referred to as a *regression* tree, because the intent of categorization is to generate a prediction along a continuous dependent dimension (here, software development effort).

There are many automatic methods for constructing decision and regression trees from data, but these techniques are typically variations on one simple strategy. A "top-down" strategy examines the data and selects an attribute that best divides the data into disjoint subpopulations. The most important aspect of decision and regression tree learners is the criterion used to select a "divisive" attribute during tree construction. In one variation the system selects the attribute with values that maximally reduce the mean squared error (MSE) of the dependent dimension (e.g., software development effort) observed in the training data. The MSE of any set, S,

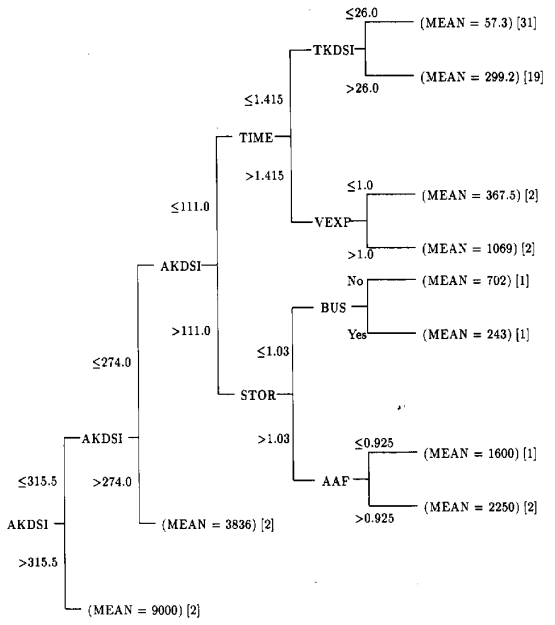


Fig. 1. A regression tree over Boehm's 63 software project descriptions. Numbers in square brackets represent the number of projects classified under a node.

of training examples taking on values y_k in the continuous dependent dimension is:

$$MSE(S) = \frac{\sum_{k \in S} (y_k - \overline{y})^2}{|S|}$$

where $\overline{y}$ is the mean of the y_k values exhibited in S. The values of each attribute, A_i partition the entire training data set, T, into subsets, T_{ij}, where every example in T_{ij} takes on the same value, say V_j for attribute A_i. The attribute, A_i, that maximizes the difference:

$$\Delta MSE = MSE(T) - \sum_j MSE(T_{ij})$$

is selected to divide the tree. Intuitively, the attribute that minimizes the error over the dependent dimension is used. While MSE values are computed over the training data, the inductive assumption is that selected attributes will similarly reduce error over future cases as well.

This basic procedure of attribute selection is easily extended to allow continuously-valued attributes: all ordered 2-partitions of the observed values in the training data are examined. In essence, the dimension is split around each observed value. The effect is to 2-partition the dimension in $k - 1$ alternate ways (where k is the number of observed values), and the binary "split" that is best according to ΔMSE is considered along with other possible attributes to divide a regression-tree node. Such "splitting" is common in the tree of Fig. 1; see AKDSI, for example. Approaches have also been developed that split a continuous dimension into more than two ranges [9], [15], though we will assume

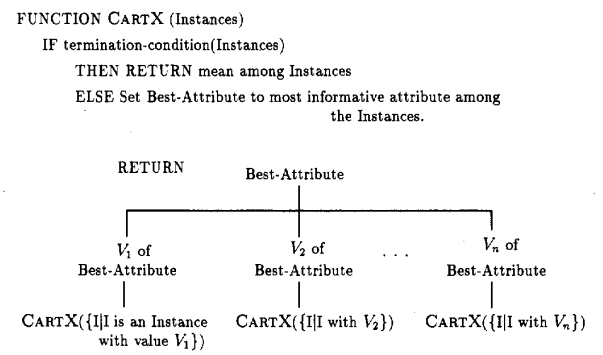


Fig. 2. Decision/regression-tree learning algorithm.

2-partitions only. Similarly, techniques that 2-partition *all* attribute domains, for both continuous and nominally-valued (i.e., finite, unordered) attributes, have been explored (e.g., [24]). For continuous attributes this bisection process operates as we have just described, but for a nominally-valued attribute all ways to group values of the attribute into two disjoint sets are considered. Suffice it to say that treating all attributes as though they had the same number of values (e.g., 2) for purposes of attribute selection mitigates certain biases that are present in some attribute selection measures (e.g., ΔMSE). As we will note again in Section IV, we ensure that all attributes are either continuous or binary-valued at the outset of regression-tree construction.

The basic regression-tree learning algorithm is summarized in Fig. 2. The data set is first tested to see whether tree construction is worthwhile; if all the data are classified identically or some other statistically-based criterion is satisfied, then expansion ceases. In this case, the algorithm simply returns a leaf labeled by the mean value of the dependent dimension found in the training data. If the data are not sufficiently distinguished, then the best divisive attribute according to ΔMSE is selected, the attribute's values are used to partition the data into subsets, and the procedure is recursively called on these subsets to expand the tree. When used to construct predictors along continuous dimensions, this general procedure is referred to as *recursive-partitioning regression*. Our experiments use a partial reimplementation of a system known as CART [4]. We refer to our reimplementation as CARTX.

Previously, Porter and Selby [14], [15], [22], have investigated the use of decision-tree induction for estimating development effort and other resource-related dimensions. Their work assumes that if predictions over a continuous dependent dimension are required, then the continuous dimension is "discretized" by breaking it into mutually-exclusive ranges. More commonly used decision-tree induction algorithms, which assume discrete-valued dependent dimensions, are then applied to the appropriately classified data. In many cases this preprocessing of a continuous dependent dimension may be profitable, though regression-tree induction demonstrates that the general tree-construction approach can be adapted for direct manipulation of a continuous dependent dimension. This is also the case with the learning approach that we describe next.

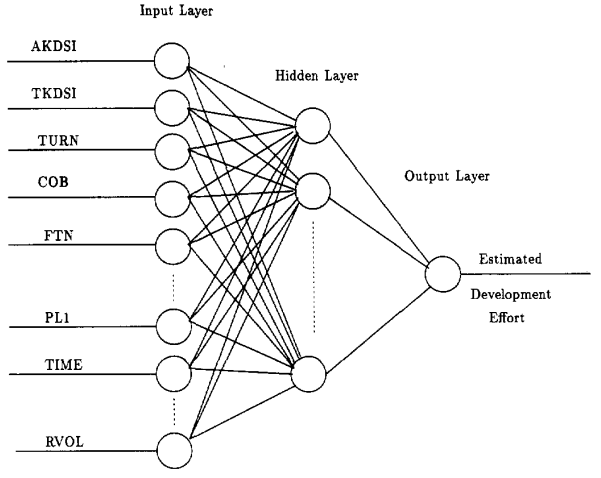


Fig. 3. A network architecture for software development effort estimation.

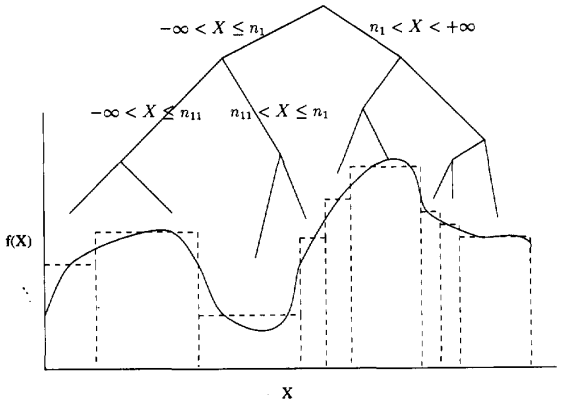


Fig. 4. An example of function approximation by a regression tree.

B. A Neural Network Approach to Learning

A learning approach that is very different from that outlined above is BACKPROPAGATION, which operates on a network of simple processing elements as illustrated in Fig. 3. This basic architecture is inspired by biological nerve nets, and is thus called an artificial neural network. Each line between processing elements has a corresponding and distinct weight. Each processing unit in this network computes a nonlinear function of its inputs and passes the resultant value along as its output. The favored function is

$$\frac{1}{1+\exp\left[-\left(\sum_i w_i I_i\right)\right]}$$

where $\sum_i w_i I_i$ is a weighted sum of the inputs, I_i, to a processing element [19], [25].

The network generates output by propagating the initial inputs, shown on the lefthand side of Fig. 3, through subsequent layers of processing elements to the final output layer. This net illustrates the kind of mapping that we will use for estimating software development effort, with inputs corresponding to various project attributes, and the output line corresponding to the estimated development effort. The inputs and output are restricted to numeric values. For numerically-valued attributes this mapping is natural, but for nominal data such as LANG (implementation language), a numeric representation must be found. In this domain, each value of a nominal attribute is given its own input line. If the value is present in an observation then the input line is set to 1.0, and if the value is absent then it is set to 0.0. Thus, for a given observation the input line corresponding to an observed nominal value (e.g., COB) will be 1.0, and the others (e.g., FTN) will be 0.0. Our application requires only one network output, but other applications may require more than one.

Details of the BACKPROPAGATION learning procedure are beyond the scope of this article, but intuitively the goal of learning is to train the network to generate appropriate output patterns for corresponding input patterns. To accomplish this, comparisons are made between a network's actual output pattern and an *a priori* known correct output pattern. The difference or error between each output line and its correct corresponding value is "backpropagated" through the net and guides the modification of weights in a manner that will tend to reduce the collective error between actual and correct outputs on training patterns. This procedure has been shown to converge on accurate mappings between input and output patterns in a variety of domains [21], [25].

C. Approximating Arbitrary Functions

In trying to approximate an arbitrary function like development effort, regression trees approximate a function with a "staircase" function. Fig. 4 illustrates a function of one continuous, independent variable. A regression tree decomposes this function's domain so that the mean at each leaf reflects the function's range within a local region. The "hidden" processing elements that reside between the input and output layers of a neural network do roughly the same thing, though the approximating function is generally smoothed. The granularity of this partitioning of the function is modulated by the depth of a regression tree or the number of hidden units in a network.

Each learning approach is nonparametric, since it makes no *a priori* assumptions about the form of the function being approximated. There are a wide variety of parametric methods for function approximation such as regression methods of statistics and polynomial interpolation methods of numerical analysis [10]. Other nonparametric methods include genetic algorithms [7] and nearest neighbor approaches [1], though we will not elaborate on any of these alternatives here.

D. Sensitivity to Configuration Choices

Both BACKPROPAGATION and CARTX require that the analyst make certain decisions about algorithm implementation. For example, BACKPROPAGATION can be used to train networks with differing numbers of hidden units. Too few hidden units can compromise the ability of the network to approximate a desired function. In contrast, too many hidden units can lead to "overfitting," whereby the learning system fits the "noise"

present in the training data, as well as the meaningful trends that we would like to capture. BACKPROPAGATION is also typically trained by iterating through the training data many times. In general, the greater the number of iterations, the greater the reduction in error over the training sample, though there is no general guarantee of this. Finally, BACKPROPAGATION assumes that weights in the neural network are initialized to small, random values prior to training. The initial random weight settings can also impact learning success, though in many applications this is not a significant factor. There are other parameters that can effect BACKPROPAGATION 's performance, but we will not explore these here.

In CARTX, the primary dimension under control by the experimenter is the depth to which the regression tree is allowed to grow. Growth to too great a depth can lead to overfitting, and too little growth can lead to underfitting. Experimental results of Section IV-B illustrate the sensitivity of each learning system to certain configuration choices.

IV. OVERVIEW OF EXPERIMENTAL STUDIES

We conducted several experiments with CARTX and BACKPROPAGATION for the task of estimating software development effort. In general, each of our experiments partitions historical data into samples used to train our learning systems, and disjoint samples used to test the accuracy of the trained classifier in predicting development effort.

For purposes of comparison, we refer to previous experimental results by Kemerer [11]. He conducted comparative analyses between SLIM, COCOMO, and FUNCTION POINTS on a database of 15 projects.[1] These projects consist mainly of business applications with a dominant proportion of them (12/15) written in the COBOL language. In contrast, the COCOMO database includes instances of business, scientific, and system software projects, written in a variety of languages including COBOL, PL1, HMI, and FORTRAN. For comparisons involving COCOMO, Kemerer coded his 15 projects using the same attributes used by Boehm.

One way that Kemerer characterized the fit between the predicted (M_{est}) and actual (M_{act}) development person-months was by the *magnitude of relative error* (MRE):

$$MRE = \left| \frac{M_{est} - M_{act}}{M_{act}} \right|.$$

This measure normalizes the difference between actual and predicted development months, and supplies an analyst with a measure of the reliability of estimates by different models. However, when using a model developed at one site for estimation at another site, there may be local factors that are not modeled, but which nonetheless impact development effort in a systematic way. Thus, following earlier work by Albrecht [2], Kemerer did a linear regression/correlation analysis to "calibrate" the predictions, with M_{est} treated as the independent variable and M_{act} treated as the dependent variable. The R^2 value indicates the amount of variation in the actual values accounted for by a linear relationship with the estimated values. R^2 values close to 1.0 suggest a strong linear relationship and those close to 0.0 suggest no such relationship. Our experiments will characterize the abilities of BACKPROPAGATION and CARTX using the same dimensions as Kemerer: MRE and R^2.

As we noted, each system imposes certain constraints on the representation of data. There are a number of nominally-valued attributes in the project databases, including implementation language. BACKPROPAGATION requires that each value of such an attribute was treated as a binary-valued attribute that was either present (1) or absent (0) in each project. Thus, each value of a nominal attribute corresponded to a unique input to the neural network as noted in Section III-B. We represent each nominal attribute as a set of binary-valued attributes for CARTX as well. As we noted in Section III-A this mitigates certain biases in attribute selection measures such as ΔMSE.

In contrast, each continuous attribute identified by Boehm corresponded to one input to the neural network. There was one output unit, which reflected a prediction of development effort and was also continuous. Preprocessing for the neural network normalized these values between 0.0 and 1.0. A simple scheme was used where each value was divided by the maximum of the values for that attribute in the training data. It has been shown empirically that neural networks converge relatively quickly if all the values for the attributes are between zero and one [12]. No such normalization was done for CARTX, since it would have no effect on CARTX's performance.

A. Experiment 1: Comparison with Kemerer's Results

Our first experiment compares the performance of machine learning algorithms with standard models of software development estimation using Kemerer's data as a test sample. To test CARTX and BACKPROPAGATION, we trained each system on COCOMO's database of 63 projects and tested on Kemerer's 15 projects. For BACKPROPAGATION we initially configured the network with 33 input units, 10 hidden units, and 1 output unit, and required that the training set error reach 0.00001 or continue for a maximum of 12 000 presentations of the training data. Training ceased after 12 000 presentations without converging to the required error criterion. The experiment was done on an AT&T PC 386 under DOS. It required about 6–7 hours for 12 000 presentations of the training patterns. We actually repeated this experiment 10 times, though we only report the results of one run here; we summarize the complete set of experiments in Section IV-B.

In our initial configuration of CARTX, we allowed the regression tree to grow to a "maximum" depth, where each leaf represented a single software project description from the COCOMO data. We were motivated initially to extend the tree to singleton leaves, because the data is very sparse relative to the number of dimensions used to describe each data point; our concern is not so much with overfitting, as it is with underfitting the data. Experiments with the regression tree learner were performed on a SUN 3/60 under UNIX, and required about a minute. The predictions obtained from the learning algorithms (after training on the COCOMO data) are shown in Table I with the actual person-months of Kemerer's

[1] We thank Professor Chris Kemerer for supplying this dataset.

TABLE I
CARTX AND BACKPROPAGATION ESTIMATES ON KEMERER'S DATA

Actual	CARTX	BACKPROP
287.00	1893.30	86.45
82.50	162.03	14.14
1107.31	11400.00	1000.43
86.90	243.00	88.37
336.30	6600.00	540.42
84.00	129.17	13.16
23.20	129.17	45.38
130.30	243.00	78.92
116.00	1272.00	113.18
72.00	129.17	15.72
258.70	243.00	80.87
230.70	243.00	28.65
157.00	243.00	44.29
246.90	243.00	39.17
69.90	129.17	214.71

15 projects. We note that some predictions of CARTX do not correspond to exact person-month values of any COCOMO (training set) project, even though the regression tree was developed to singleton leaves. This stems from the presence of missing values for some attributes in Kemerer's data. If, during classification of a test project, we encounter a decision node that tests an attribute with an unknown value in the test project, both subtrees under the decision node are explored. In such a case, the system's final prediction of development effort is a weighted mean of the predictions stemming from each subtree. The approach is similar to that described in [17].

Table II summarizes the MRE and R^2 values resulting from a linear regression of M_{est} and M_{act} values for the two learning algorithms, and results obtained by Kemerer with COCOMO-BASIC, FUNCTION POINTS, and SLIM.[2] These results indicate that CARTX's and BACKPROPAGATION 's predictions show a strong linear relationship with the actual development effort values for the 15 test projects.[3] On this dimension, the performance of the learning systems is less than SLIM's performance in Kemerer's experiments, but better than the other two models. In terms of mean MRE, BACKPROPAGATION does strikingly well compared to the other approaches, and CARTX's MRE is approximately one-half that of SLIM and COCOMO.

In sum, Experiment 1 illustrates two points. In an absolute sense, none of the models does particularly well at estimating software development effort, particularly along the MRE dimension, but in a relative sense both learning approaches are competitive with traditional models examined by Kemerer on one dataset. In general, even though MRE is high in the case of all models, Kemerer argues that high R^2 suggests that by "calibrating" a model's predictions in a new environment, the adjusted model predictions can be reliably used. Along the R^2 dimension learning methods provide significant fits to the data. Unfortunately, a primary weakness of these learning approaches is that their performance is sensitive to a number of implementation decisions. Experiment 2 illustrates some of these sensitivities.

[2]Results are reported for COCOMO-BASIC (i.e., without cost drivers), which was comparable to the intermediate and detailed models on this data. In addition, Kemerer actually reported $\overline{R}^2$, which is R^2 adjusted for degrees of freedom, and which is slightly lower than the unadjusted R^2 values that we report. $\overline{R}^2$ values reported by Kemerer are 0.55, 0.68, and 0.88 for FUNCTION POINTS, COCOMO, and SLIM, respectively.

[3]Both the slope and R value are significant at the 99% confidence level. The t coefficients for determining the significance of slope are 8.048 and 7.25 for CARTX and BACKPROPAGATION, respectively.

TABLE II
A COMPARISON OF LEARNING AND ALGORITHMIC APPROACHES. THE REGRESSION EQUATIONS GIVE M_{act} AS A FUNCTION OF $M_{est}(x)$

	MRE(%)	R-Square	Regress. Eq.
CARTX	364	0.83	$102.5 + 0.075x$
BACKPROP	70	0.80	$78.13 + 0.88x$
FUNC. PTS.	103	0.58	$-37 + 0.96x$
COCOMO	610	0.70	$27.7 + 0.156x$
SLIM	772	0.89	$49.9 + 0.082x$

B. Experiment 2: Sensitivity of the Learning Algorithms

We have noted that each learning system assumes a number of important choices such as depth to which to "grow" regression trees, or the number of hidden units included in the neural network. These choices can significantly impact the success of learning. Experiment 2 illustrates the sensitivity of our two learning systems relative to different choices along these dimensions. In particular, we repeated Experiment 1 using BACKPROPAGATION with differing numbers of hidden units and using CARTX with differing constraints on regression-tree growth.

Table III illustrates our results with BACKPROPAGATION. Each cell summarizes results over 10 experimental trials, rather than one trial, which was reported in Section IV-A for presentation purposes. Thus, Max, and Min values of R^2 and MRE in each cell of Table III suggest the sensitivity of BACKPROPAGATION to initial random weight settings, which were different in each of the 10 experimental trials. The experimental results of Section IV-A reflect the "best" among the 10 trials summarized in Table III's 10-hidden-unit column. In general, however, for 5, 10, and 15 hidden units, MRE scores are still comparable or superior to some of the other models summarized in Table II, and mean R^2 scores suggest that significant linear relationships between predicted and actual development months are often found. Poor results obtained with no hidden units indicate the importance of these for accurate function approximation.

The performance of CARTX can vary with the depth to which we extend the regression tree. The results of Experiment 1 are repeated here, and represent the case where required accuracy over the training data is 0%—that is, the tree is decomposed to singleton leaves. However, we experimented with more conservative tree expansion policies, where CARTX extended the tree only to the point where an error threshold (relative to the training data) is satisfied. In particular, trees were grown to leaves where the mean MRE among projects at a leaf was less than or equal to a prespecified threshold that ranged from 0% to 500%. The MRE of each project at a leaf

TABLE III
BACKPROPAGATION RESULTS WITH VARYING NUMBERS OF HIDDEN NODES

	Hidden Units			
	0	5	10	15
Mean R^2	0.04	0.52	0.60	0.59
Max R^2	0.04	0.84	0.80	0.85
Min R^2	0.03	0.08	0.10	0.01
Mean $MRE(\%)$	618	104	133	111
Max $MRE(\%)$	915	163	254	161
Min $MRE(\%)$	369	72	70	77

TABLE IV
CARTX RESULTS WITH VARYING TRAINING ERROR THRESHOLDS

	R^2	$MRE(\%)$
0%	0.83	364
25%	0.62	404
50%	0.63	461
100%	0.60	870
200%	0.59	931
300%	0.59	931
400%	0.59	931
500%	0.60	835

was calculated by

$$\left|\frac{\overline{M} - M_{act}}{M_{act}}\right|$$

where $\overline{M}$ is the mean person-months development effort of projects at that node.

Table IV shows CARTX's performance when we vary the required accuracy of the tree over the training data. Table entries correspond to the MRE and R^2 scores of the learned trees over the Kemerer test data. In general, there is degradation in performance as one tightens the requirement for regression-tree expansion, though there are applications in which this would not be the case. Importantly, other design decisions in decision and regression-tree systems, such as the manner in which continuous attributes are "split" and the criteria used to select divisive attributes, might also influence prediction accuracy. Selby and Porter [22] have evaluated different design choices along a number of dimensions on the success of decision-tree induction systems using NASA software project descriptions as a test-bed. Their evaluation of decision trees, not regression trees, limits the applicability of their findings to the evaluation reported here, but their work sets an excellent example of how sensitivity to various design decisions can be evaluated.

The performance of both systems is sensitive to certain configuration choices, though we have only examined sensitivity relative to one or two dimensions for each system. Thus, it seems important to posit some intuition about how learning systems can be configured to yield good results on new data, given only knowledge of performance on training data. In cases where more training data is available a *holdout* method can be used for selecting an appropriate network or regression-tree configuration. The holdout method divides the available data into two sets; one set, generally the larger, is used to build decision/regression trees or train networks under different configurations. The second subset is then classified using each alternative configuration, and the configuration yielding the best results over this second subset is selected as the final configuration. Better yet, a choice of configuration may rest on a form of *resampling* that exploits many randomized holdout trials. Holdout could have been used in this case by dividing the COCOMO data, but the COCOMO dataset is very small as is. Thus, we have satisfied ourselves with a demonstration of the sensitivity of each learning algorithm to certain configuration decisions. A more complete treatment of resampling and other strategies for making configuration choices can be found in Weiss and Kulikowski [24].

TABLE V
SENSITIVITY OVER 20 RANDOMIZED TRIALS ON COMBINED COCOMO AND KEMERER'S DATA

	Min R^2	Mean R^2	Max R^2
CARTX	0.00	0.48	0.97
BACKPROPAGATION	0.00	0.40	0.99

TABLE VI
SENSITIVITY OVER 20 RANDOMIZED TRIALS ON KEMERER'S DATA

	Min R^2	Mean R^2	Max R^2
CARTX	0.00	0.26	0.90
BACKPROPAGATION	0.03	0.39	0.90

C. Experiment 3: Sensitivity to Training and Test Data

Thus far, our results suggest that using learning algorithms to discover regularities in a historical database can facilitate predictions on new cases. In particular, comparisons between our experimental results and those of Kemerer indicate that relatively speaking, learning system performance is competitive with some traditional approaches on one common data set. However, Kemerer found that performance of algorithmic approaches was sensitive to the test data. For example, when a selected subset of 9 of the 15 cases was used to test the models, each considerably improved along the R^2 dimension. By implication, performance on the other 6 projects was likely poorer. We did not repeat this experiment, but we did perform similarly-intended experiments in which the COCOMO and Kemerer data sets were combined into a single dataset of 78 projects; 60 projects were randomly selected for training the learning algorithms and the remaining 18 projects were used for test. Table V summarizes the results over 20 such randomized trials. The low average R^2 should not mask the fact that many runs yielded strong linear relationships. For example, on 9 of the 20 CARTX runs, R^2 was above 0.80.

We also ran 20 randomized trials in which 10 of Kemerer's cases were used to train each learning algorithm, and 5 were used for test. The results are summarized in Table VI. This experiment was motivated by a study with ESTOR [23], a case-based approach that we summarized in Section II: an expert's protocols from 10 of Kemerer's projects were used to construct

a "case library" and the remaining 5 cases were used to test the model's predictions; the particular cases used for test were not reported, but ESTOR outperformed COCOMO and FUNCTION POINTS on this set.

We do not know the robustness of ESTOR in the face of the kind of variation experienced in our 20 randomized trials (Table VI), but we might guess that rules inferred from expert problem solving, which ideally stem from human learning over a larger set of historical data, would render ESTOR more robust along this dimension. However, our experiments and those of Kemerer with selected subsets of his 15 cases suggest that care must be taken in evaluating the robustness of any model with such sparse data. In defense of Vicinanza's *et al.*P methodology, we should note that the creation of a case library depended on an analysis of expert protocols and the derivation of expert-like rules for modifying the predictions of best matching cases, thus increasing the "cost" of model construction to a point that precluded more complete randomized trials. Vicinanza *et al.* also point out that their study is best viewed as indicating ESTOR's "plausibility" as a good estimator, while broader claims require further study.

In addition to experiments with the combined COCOMO and Kemerer data, and the Kemerer data alone, we experimented with the COCOMO data alone for completeness. When experimenting with Kemerer's data alone, our intent was to weakly explore the kind of variation faced by ESTOR. Using the COCOMO data we have no such goal in mind. Thus, this analysis uses an N-fold cross validation or a "leave-one-out" methodology, which is another form of resampling. In particular, if a data sample is relatively sparse, as ours is, then for each of N (i.e., 63) projects, we remove it from the sample set, train the learning system with the remaining $N-1$ samples, and then test on the removed project. MRE and R^2 are computed over the N tests. CARTX's R^2 value was 0.56 ($144.48+0.74x$, $t=8.82$) and MRE was 125.2%. In this experiment we only report results obtained with CARTX, since a fair and comprehensive exploration of BACKPROPAGATION across possible network configurations is computationally expensive and of limited relevance. Suffice it to say that over the COCOMO data alone, which probably reflects a more uniform sample than the mixed COCOMO/Kemerer data, CARTX provides a significant linear fit to the data with markedly smaller MRE than its performance on Kemerer's data.

In sum, our initial results indicating the relative merits of a learning approach to software development effort estimation must be tempered. In fact, a variety of randomized experiments reveal that there is considerable variation in the performance of these systems as the nature of historical training data changes. This variation probably stems from a number of factors. Notably, there are many projects in both the COCOMO and Kemerer datasets that differ greatly in their actual development effort, but are very similar in other respects, including SLOC. Other characteristics, which are currently unmeasured in the COCOMO scheme, are probably responsible for this variation.

V. GENERAL DISCUSSION

Our experimental comparisons of CARTX and BACKPROPAGATION with traditional approaches to development effort estimation suggest the promise of an automated learning approach to the task. Both learning techniques performed well on the R^2 and MRE dimensions relative to some other approaches on the same data. Beyond this cursory summary, our experimental results and the previous literature suggest several issues that merit discussion.

A. Limitations of Learning from Historical Data

There are well-known limitations of models constructed using historical data. In particular, attributes used to predict software development effort can change over time and/or differ between software development environments. Mohanty [13] makes this point in comparisons between the predictions of a wide variety of models on a single hypothetical software project. In particular, Mohanty surveyed approximately 15 models and methods for predicting software development effort. These models were used to predict software development effort of a single hypothetical software project. Mohanty's main finding was that estimated effort on this single project varied significantly over models. Mohanty points out that each model was developed and calibrated with data collected within a unique software environment. The predictions of these models, in part, reflect underlying assumptions that are not explicitly represented in the data. For example, software development sites may use different development tools. These tools are constant within a facility, and thus not represented explicitly in data collected by that facility, but this environmental factor is not constant across facilities.

Differing environmental factors not reflected in data are undoubtedly responsible for much of the unexplained variance in our experiments. To some extent, the R^2 derived from linear regression is intended to provide a better measure of a model's "fit" to arbitrary new data than MRE in cases where the environment from which a model was derived is different from the environment from which new data was drawn. Even so, these environmental differences may not be systematic in a way that is well accounted for by a linear model. In sum, great care must be taken when using a model constructed from data from one environment to make predictions about data from another environment. Even within a site, the environment may evolve over time, thus compromising the benefits of previously-derived models. Machine learning research has recently focussed on the problem of *tracking* the accuracy of a learned model over time, which triggers relearning when experience with new data suggests that the environment has changed [6]. However, in an application such as software development effort estimation, there are probably explicit indicators that an environmental change is occurring or will occur (e.g., when new development tools or quality control practices are implemented).

B. Engineering the Definition of Data

If environmental factors are relatively constant, then there is little need to explicitly represent these in the description of data. However, when the environment exhibits variance along some dimension, it often becomes critical that this variance be codified and included in data description. In this way,

differences across data points can be observed and used in model construction. For example, Mohanty argues that the desired quality of the finished product should be taken into account when estimating development effort. A comprehensive survey by Scacchi [20] of previous software production studies leads to considerable discussion on the pros and cons of many attributes for software project representation.

Thus, one of the major tasks is deciding upon the proper codification of factors judged to be relevant. Consider the dimension of response time requirements (i.e., TIME) which was included by Boehm in project descriptions. This attribute was selected by CARTX during regression-tree construction. However, is TIME an "optimal" codification of some aspect of software projects that impacts development effort? Consider that strict response time requirements may motivate greater coupling of software modules, thereby necessitating greater communication among developers and in general increasing development effort. If predictions of development effort must be made at the time of requirements analysis, then perhaps TIME is a realistic dimension of measurement, but better predictive models might be obtained and used given some measure of software component coupling.

In sum, when building models via machine learning or statistical methods, it is rarely the case that the set of descriptive attributes is static. Rather, in real-world success stories involving machine learning tools the set of descriptive attributes evolves over time as attributes are identified as relevant or irrelevant, the reasons for relevance are analyzed, and additional or replacement attributes are added in response to this analysis [8]. This "model" for using learning systems in the real world is consistent with a long-term goal of Scacchi [20], which is to develop a knowledge-based "corporate memory" of software production practices that is used for both estimating and controlling software development. The machine-learning tools that we have described, and other tools such as ESTOR, might be added to the repertoire of knowledge-acquisition strategies that Scacchi suggests. In fact, Porter and Selby [14] make a similar proposal by outlining the use of decision-tree induction methods as tools for software development.

C. The Limitations of Selected Learning Methods

Despite the promising results on Kemerer's common database, there are some important limitations of CARTX and BACKPROPAGATION. We have touched upon the sensitivity to certain configuration choices. In addition to these practical limitations, there are also some important theoretical limitations, primarily concerning CARTX. Perhaps the most important of these is that CARTX cannot estimate a value along a dimension (e.g., software development effort) that is outside the range of values encountered in the training data. Similar limitations apply to a variety of other techniques as well (e.g., nearest neighbor approaches of machine learning and statistics). In part, this limitation appears responsible for a sizable amount of error on test data. For example, in the experiment illustrating CARTX's sensitivity to training data using 10/5 splits of Kemerer's projects (Section IV-C), CARTX is doomed to being at least a factor of 3 off the mark when estimating the person-month effort required for the project requiring 23.20 M or the project requiring 1107.31 M; the projects closest to each among the remaining 14 projects are 69.90 M and 336.30 M, respectively.

The root of CARTX's difficulties lies in its labeling of each leaf by the mean of development months of projects classified at the leaf. An alternative approach that would enable CARTX to extrapolate beyond the training data, would label each leaf by an equation derived through regression—e.g., a linear regression. After classifying a project to a leaf, the regression equation labeling that leaf would then be used to predict development effort given the object's values along the independent variables. In addition, the criterion for selecting divisive attributes would be changed as well. To illustrate, consider only two independent attributes, development team experience and KDSI, and the dependent variable of software development effort. CARTX would undoubtedly select KDSI, since lower (higher) values of KDSI tend to imply lower (higher) means of development effort. In contrast, development team experience might not provide as good a fit using CARTX's error criterion. However, consider a CART-like system that divides data up by an independent variable, finds a best fitting linear equation that predicts development effort given development team experience and KDSI, and assesses error in terms of the differences between predictions using this best fitting equation and actual development months. Using this strategy, development team experience might actually be preferred; even though lesser (greater) experience does not imply lesser (greater) development effort, development team experience does imply subpopulations for which strong linear relationships might exist between independent and dependent variables. For example, teams with lesser experience may not adjust as well to larger projects as do teams with greater experience; that is, as KDSI increases, development effort increases are larger for less experienced teams than more experienced teams. Recently, machine learning systems have been developed that have this flavor [18]. We have not yet experimented with these systems, but the approach appears promising.

The success of CARTX, and decision/regression-tree learners generally, may also be limited by two other processing characteristics. First, CARTX uses a *greedy* attribute selection strategy—tree construction assesses the informativeness of a single attribute at a time. This greedy strategy might overlook attributes that participate in more accurate regression trees, particularly when attributes interact in subtle ways. Second, CARTX builds one classifier over a training set of software projects. This classifier is static relative to the test projects; any subsequent test project description will match exactly one conjunctive pattern, which is represented by a path in the regression tree. If there is noise in the data (e.g., an error in the recording of an attribute value), then the prediction stemming from the regression-tree path matching a particular test project may be very misleading. It is possible that other conjunctive patterns of attribute values matching a particular test project, but which are not represented in the regression tree, could ameliorate CARTX's sensitivity to errorful or otherwise noisy project descriptions.

The *Optimized Set Reduction* (OSR) strategy of Briand, Basili, and Thomas [5] is related to the CARTX approach in several important ways, but may mitigate problems associated with CARTX—OSR conducts a *more extensive search* for *multiple patterns* that match each test observation. In contrast to CARTX's construction of a single classifier that is static relative to the test projects, OSR can be viewed as dynamically building a different classifier for each test project. The specifics of OSR are beyond the scope of this paper, but suffice it to say that OSR looks for multiple patterns that are statistically justified by the training project descriptions and that match a given test project. The predictions stemming from different patterns (say, for software development effort) are then combined into a single, global prediction for the test project.

OSR was also evaluated in [5] using Kemerer's data for test, and COCOMO data as a (partial) training sample.[4] The authors report an average MRE of 94% on Kemerer's data. However, there are important differences in experimental design that make a comparison between results with OSR, BACKPROPAGATION, and CARTX unreliable. In particular, when OSR was used to predict software development effort for a particular Kemerer project, the COCOMO data *and* the remaining 14 Kemerer projects were used as training examples. In addition, recognizing that Kemerer's projects were selected from the same development environment, OSR was configured to weight evidence stemming from these projects more heavily than those in the Cocomo data set. The sensitivity of results to this "weighting factor" is not described.

We should note that the experimental conditions assumed in [5] are quite reasonable from a pragmatic standpoint, particularly the decision to weight projects more heavily that are drawn from the same environment as the test project. These different training assumptions simply confound comparisons between experimental results, and OSR's robustness across differing training and test sets is not reported. In addition, like the work of Porter and Selby [14], [15], [22], OSR assumes that the dependent dimension of software development effort is nominally-valued for purposes of learning. Thus, this dimension is partitioned into a number of collectively-exhaustive and mutually-exclusive ranges prior to learning. Neither BACKPROPAGATION nor CARTX requires this kind of preprocessing. In any case, OSR appears unique relative to other machine learning systems in that it does not learn a static classifier; rather, it combines predictions from multiple, dynamically-constructed patterns. Whether one is interested in software development effort estimation or not, this latter facility appears to have merits that are worth further exploration.

In sum, CARTX suffers from certain theoretical limitations: it cannot extrapolate beyond the data on which it was trained, it uses a greedy tree expansion strategy, and the resultant classifier generates predictions by matching a project against a single conjunctive pattern of attribute values. However, there appear to be extensions that might mitigate these problems.

[4] Our choice of using COCOMO data for training and Kemerer's data for test was made independently of [5].

VI. CONCLUDING REMARKS

This article has compared the CARTX and BACKPROPAGATION learning methods to traditional approaches for software effort estimation. We found that the learning approaches were competitive with SLIM, COCOMO, and FUNCTION POINTS as represented in a previous study by Kemerer. Nonetheless, further experiments showed the sensitivity of learning to various aspects of data selection and representation. Mohanty and Kemerer indicate that traditional models are quite sensitive as well.

A primary advantage of learning systems is that they are adaptable and nonparametric; predictive models can be tailored to the data at a particular site. Decision and regression trees are particularly well-suited to this task because they make explicit the attributes (e.g., TIME) that appear relevant to the prediction task. Once implicated, a process that engineers the data definition is often required to explain relevant and irrelevant aspects of the data, and to encode it accordingly. This process is best done locally, within a software shop, where the idiosyncrasies of that environment can be factored in or out. In such a setting analysts may want to investigate the behavior of systems like BACKPROPAGATION, CART, and related approaches [5], [14], [15], [22] over a range of permissible configurations, thus obtaining performance that is optimal in their environment.

APPENDIX A
DATA DESCRIPTIONS

The attributes defining the COCOMO and Kemerer databases were used to develop the COCOMO model. The following is a brief description of the attributes and some of their suspected influences on development effort. The interested reader is referred to [3] for a detailed exposition of them. These attributes can be classified under four major headings. They are Product Attributes; Computer Attributes; Personnel Attributes; and Project Attributes.

A. Product Attributes

1) Required Software Reliability (RELY): This attribute measures how reliable the software should be. For example, if serious financial consequences stem from a software fault, then the required reliability should be high.

2) Database Size (DATA): The size of the database to be used by software may effect development effort. Larger databases generally suggest that more time will be required to develop the software product.

3) Product Complexity (CPLX): The application area has a bearing on the software development effort. For example, communications software will likely have greater complexity than software developed for payroll processing.

4) Adaptation Adjustment Factor (AAF): In many cases software is not developed entirely from scratch. This factor reflects the extent that previous designs are reused in the new project.

B. Computer Attributes

1) Execution Time Constraint (TIME): If there are constraints on processing time, then the development time may be greater.

2) Main Storage Constraint (STOR): If there are memory constraints, then the development effort will tend to be high.

3) Virtual Machine Volatility (VIRT): If the underlying hardware and/or system software change frequently, then development effort will be high.

C. Personnel Attributes

1) Analyst Capability (ACAP): If the analysts working on the software project are highly skilled, then the development effort of the software will be less than projects with less-skilled analysts.

2) Applications Experience (AEXP): The experience of project personnel influences the software development effort.

3) Programmer Capability (PCAP): This is similar to ACAP, but it applies to programmers.

4) Virtual Machine Experience (VEXP): Programmer experience with the underlying hardware and the operating system has a bearing on development effort.

5) Language Experience (LEXP): Experience of the programmers with the implementation language affects the software development effort.

6) Personnel Continuity Turnover (CONT): If the same personnel work on the project from beginning to end, then the development effort will tend to be less than similar projects experiencing greater personnel turnover.

D: Project Attributes

1) Modern Programming Practices (MODP): Modern programming practices like structured software design reduces the development effort.

2) Use of Software Tools (TOOL): Extensive use of software tools like source-line debuggers and syntax-directed editors reduces the software development effort.

3) Required Development Schedule (SCED): If the development schedule of the software project is highly constrained, then the development effort will tend to be high.

Apart from the attributes mentioned above, other attributes that influence the development are: programming language, and the estimated lines of code (unadjusted and adjusted for the use of existing software).

ACKNOWLEDGMENT

The authors would like to thank the three reviewers and the action editor for their many useful comments.

REFERENCES

[1] D. Aha, D. Kibler, and M. Albert, "Instance-based learning algorithms," *Machine Learning,* vol. 6, pp. 37–66, 1991.

[2] A. Albrecht and J. Gaffney Jr., "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Trans. Software Eng.,* vol. 9, pp. 639–648, 1983.

[3] B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees.* Belmont, CA: Wadsworth International, 1984.

[5] L. Briand, V. Basili, and W. Thomas, "A pattern recognition approach for software engineering data analysis," *IEEE Trans. Software Eng.,* vol. 18, pp. 931–942, Nov. 1992.

[6] C. Brodley and E. Rissland, "Measuring concept change," in *AAAI Spring Symp. Training Issues in Incremental Learning,* 1993, pp. 98–107.

[7] K. DeJong, "Learning with genetic algorithms," *Machine Learning,* vol. 3, pp. 121–138, 1988.

[8] B. Evans and D. Fisher, "Overcoming process delays with decision tree induction," *IEEE Expert,* vol. 9, pp. 60–66, Feb. 1994.

[9] U. Fayyad, "On the induction of decision trees for multiple concept learning," Doctoral dissertation, EECS Dep., Univ. of Michigan, 1991.

[10] L. Johnson and R. Riess, *Numerical Analysis.* Reading, MA: Addison-Wesley, 1982.

[11] C. F. Kemerer, "An empirical validation of software cost estimation models," *Commun. ACM,* vol. 30, pp. 416–429, May 1987.

[12] A. Lapedes and R. Farber, "Nonlinear signal prediction using neural networks: Prediction and system modeling," Los Alamos National Laboratory, 1987, Tech. Rep. LA-UR-87-2662

[13] S. Mohanty, "Software cost estimation: Present and future," *Software—Practice and Experience,* vol. 11, pp. 103–121, 1981.

[14] A. Porter and R. Selby, "Empirically-guided software development using metric-based classification trees," *IEEE Software,* vol. 7, pp. 46–54, Mar. 1990.

[15] A. Porter and R. Selby, "Evaluating techniques for generating metric-based classification trees," *J. Syst. Software,* vol. 12, pp. 209–218, July 1990.

[16] L. H. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.,* vol. 4, pp. 345–361, 1978.

[17] J. R. Quinlan, *C4.5: Programs for Machine Learning.* San Mateo, CA: Morgan Kaufmann, 1993.

[18] J. R. Quinlan, "Combining instance-based and model-based learning," in *Proc. the 10th Int. Machine Learning Conf.,* 1993, pp. 236–243.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing.* Cambridge, MA: MIT Press, 1986.

[20] W. Scacchi, "Understanding software productivity: Toward a knowledge-based approach," *Int. J. Software Eng. and Knowledge Eng.,* vol. 1, pp. 293–320, 1991.

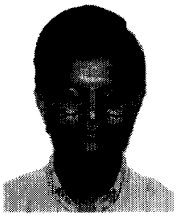
[21] T. J. Sejnowski and C. R. Rosenberg, "Parallel networks that learn to pronounce english text," *Complex Systems,* vol. 1, pp. 145–168, 1987.

[22] R. Selby and A. Porter, "Learning from examples: Generation and evaluation of decision trees for software resource analysis," *IEEE Trans. Software Eng.,* vol. 14, pp. 1743–1757, 1988.

[23] S. Vicinanza, M. J. Prietulla, and T. Mukhopadhyay, "Case-based reasoning in software effort estimation," in *Proc. 11th Int. Conf. Info. Syst.,* 1990, pp. 149–158.

[24] S. Weiss and C. Kulikowski, *Computer Systems that Learn.* San Mateo, CA: Morgan Kaufmann, 1991.

[25] J. Zaruda, *Introduction to Artificial Neural Networks.* St. Paul, MN: West, 1992.



Krishnamoorthy Srinivasan, received the M.B.A, in management information systems from the Owen Graduate School of Management, Vanderbilt University, and the M.S. in computer science from Vanderbilt University. He also received the Post Graduate Diploma in industrial engineering from the National Institute for Training in Industrial Engineering, Bombay, India, and the B.E. from the University of Madras, Madras, India.

He is currently working as a Principal Software Engineer with Personal Computer Consultants, Inc., Washington, D.C. Before joining PCC, he worked as a Senior Specialist with McKinsey & Company, Inc., Cambridge, MA. His primary research interests are in exploring applications of machine learning techniques to real-world business problems.

Douglas Fisher (M'92) received his Ph.D. in information and computer science from the University of California at Irvine in 1987.

He is currently an Associate Professor in computer science at Vanderbilt University. He is an Associate Editor of *Machine Learning*, and *IEEE Expert*, and serves on the editorial board of the *Journal of Artificial Intelligence Research*. His research interests include machine learning, cognitive modeling, data analysis, and cluster analysis. An electronic addendu to this article, which reports any subsequent analysis, can be found at (http://www.vuse.vanderbilt.edu/~dfisher/dfisher.html).

Dr. Fisher is a member of the ACM and AAAI.