Using Machine Learning to Predict Project Effort: Empirical Case Studies in Data-Starved Domains

Gary D. Boetticher
Department of Software Engineering
University of Houston - Clear Lake
2700 Bay Area Boulevard
Houston, TX 77058 USA
+1 281 283 3805
boetticher@cl.uh.edu

ABSTRACT
Ideally, software engineering should be able to use machine learning to control or significantly decrease the costs associated with building software. In reality, there are very few examples of applying such applications early in the software life cycle. One reason for the scarcity of examples is the lack of empirical data in the software engineering discipline. This dilemma is quite evident when constructing models to predict project effort. This raises the question of "How to generate sufficient amounts of data when it is sparse?" One approach is to assess projects from a bottom-up perspective. This approach uses estimates gathered from products in predicting project effort. This paper conducts a set of machine learning experiments with software cost estimation data from two separate organizations. These experiments explore the possibility of performing project estimating from a bottom-up perspective and characterize predictive potential within two different organizations. The results are statistically assessed and a process is proposed for applying the described techniques.

Keywords
Machine learning, requirements engineering, software engineering, neural networks, backpropagation, software metrics, effort estimation, SLOC, project estimation, programming effort

1. INTRODUCTION

In the mid 90's, The Standish Group surveyed over 8,000 software projects. They found that the average project exceeded its planned budget by 90 percent and its schedule by 222 percent. More than 50 percent of the completed projects had less than 50 percent of original requirements [45].

These statistics represent symptoms of deep underlying problems, which include the lack of data early in the life cycle.

This scarcity of data seems plausible when considering the distribution of software organization against the Software Engineering Institute's capability maturity model (CMM) [35]. A CMM level below 3, out of a possible 5, indicates a lack of written definition, which includes data, at various life cycle phases. About 80 to 85 percent of all software organizations exist below CMM level 3 [14, 17]. Those that do collect data and use them in project estimation processes are unlikely to share their valuable knowledge at the risk of losing a possible competitive advantage. As a result, scarcity occurs in both quantity and quality for most organizations.

This may explain the unreliability of early life cycle algorithmic effort estimators. Boehm [6] claims that very early life cycle phase accuracy varies by a factor of 4 (e.g., between 25% and 400%). This claim was supported by Heemstra [23].

This raises the question as to why is machine learning (ML) not used more frequently in formulating early-life cycle cost estimation models in order to address the data starvation problem? Recent reviews of the use of ML in SE (e.g. [31, and 32]) report that ML in SE is a mature technique based on widely-available tools using well understood algorithms (e.g. neural nets or decision tree learners or Bayes nets [15, 20, 26, and 39]. Further, machine learning tools that can handle very large data sets now come bundled and integrated with standard commercial packages such as Microsoft's SQL® Server 2000™ [41] and Oracle9*i*™ [4]. Finally, clear and simple methodological guidelines for ML in SE have existed for nearly a decade [36]. Nevertheless, the total number of reported applications (as seen in [31 and 32]) is not large.

Probably the main reason is the scarcity of data. This makes it difficult to use ML algorithms and formulate theories from data in order to construct decision support tools early in the software life cycle.

A second reason is the approach to project estimation. Traditional cost estimation models such as Putnam, Function Points, and COCOMO I and II [1, 2, 6, 7, and 38] reveal the use of a top-down perspective (project-based

instead of product-based) in cost modeling. This approach makes it difficult to cull a sufficient amount of empirical data for building cost-estimation models. What is preferred is a bottom-up approach where metric formulation originates from the product. Reasons for adopting a bottom-up approach include:

- **More data is available.** There is a cardinality of *1* to *N* between projects and products. Building project-based models is difficult due to the lack of project data that corporations possess. Also, corporations are not willing to share their project data with other organizations, so building up a sufficient amount of project data for constructing a model is very difficult.
- **Automation of data collection.** Many popular mature software metrics exist such as size, cyclomatic complexity [30], and program vocabulary [22] for characterizing software products (e.g., software programs). There are many publicly available well-established tools for extracting these metrics.

Boehm [6] first described a bottom-up approach in 1981. Interestingly, a literature search does not reveal the use of any ML algorithm in this approach early in the software life cycle.

Thus, a bottom-up approach is adopted for conducting a series of experiments. The empirical data was extracted from two separate corporations. Performing these experiments address whether it is feasible to adopt a bottom-up approach. Furthermore, these experiments assume that project prediction occurs early in the software life cycle. As a result, the type of data available for building models is highly constrained.

Section 2 describes related research in the areas of cost modeling, and compensating for sparse data. Section 3 describes a set of machine learning experiments. Section 4 offers a discussion of the experiments. And section 5 describes several conclusions and future directions.

2. RELATED WORK

Cost Estimation Framework

Different techniques for cost estimation have been discussed in the literature [6, 23, and 25]. Popular approaches include: algorithmic and parametric models, expert judgment, formal and informal reasoning by analogy, price-to-win, top-down, bottom-up, rules of thumb, and available capacity.

Heemstra [23] describes 29 software cost models that have been created since 1966. Due to the lack of data early in the software life cycle, most of these models apply to the latter stages of the software life cycle. However, there is one approach that identifies requirements measures and uses those to predict development effort [13].

More recently, machine learning models have also been developed in the area of cost estimation.

In [5, 21, 28, 34, and 37], a Case-Based Reasoning (CBR) approach is adopted in constructing a cost model for the latter stages of the development life cycle. Delany [18] also uses a CBR approach applied early in development life cycle.

Chulani [15] uses a Bayesian approach to cost modeling and generates impressive results. He collects information on 161 projects from commercial, aerospace, government, and non-profit organizations [15]. The COCOMO data sets contain attributes that, for the most part, can be collected early in the software life cycle (exception: COCOMO requires source lines of code which must be estimated). Regression analysis was applied to the COCOMO data set to generate estimators for software project effort. However, some of the results of that analysis were counter-intuitive. In particular, the results of the regression analysis disagreed with certain domain experts regarding the effect of software reuse on overall cost.

To fix this problem, a Bayesian learner was applied to the COCOMO data set. In Bayesian learning, a directed graph (the belief network) contains the probabilities that some factor will lead to another factor. The probabilities on the edges can be seeded from (e.g.) domain expertise. The learner then tunes these probabilities according to the available data. Combining expert knowledge and data from the 161 projects yielded an estimator that was within 30% of the actual values, 69% of the time [15]. It is believed that the above COCOMO result of pred(30) = 69% is a high-watermark in early life cycle software cost estimation[1].

Cordero [16] applies a Genetic Algorithm (GA) approach in the tuning of COCOMO II.

Briand [12] introduces optimized set reduction (OSR) in the construction of a software cost estimation model.

Srinivasan [44] builds a variety of models including neural networks, regression trees, COCOMO, and SLIM. The training set consists of COCOMO data (63 projects from different applications). The training models are tested against the Kemerer COMOMO data (15 projects, mainly business applications). The regression trees outperformed the COCOMO and the SLIM model. The neural networks and function point-based prediction models outperformed regression trees.

Samson [40] applies neural network models to predict effort from software sizing using COCOMO-81 data. The neural network models produced better results than the COCOMO-81.

---

[1] Some might argue that this is a very low high-watermark. The author disagrees. Given all the factors that can influence a software project, it is very surprising that pred(30) = 69 can be achieved at all.

Wittig *et al.* [47] estimated development effort using a neural network model. They achieved impressive results of 75 percent accuracy pred(25).

Boetticher [11] conducted more than 33,000 different neural network experiments on empirical data collected from separate corporate domains. The experiments assessed the contribution of different metrics to programming effort. This research produced a cross-validation rate of 73.26%, using pred(30).

Hodgkinson [27] adopted a top-down approach using a neurofuzzy cost estimator in predicting project effort. Results were comparable to other techniques including least-squares multiple linear regression, estimation via analogy, and neural networks.

Learning When Data is Scarce

Some strategies/models are proposed regarding the issue of data scarcity.

- Seek more data from the domain (which may be impractical due to the issues cited above) [12, 15].
- Address outliers within a domain [12, 15].
- Quickly build some lightweight domain models and use them to generate more data. This approach is explored elsewhere (see [32, and 33]).
- Seed the learner with some background knowledge. In this approach, the learner does not learn a totally new theory from scratch. Instead, it fine-tunes the seeded theory according to the supplied data [15].
- Use an Analytic Hierarchy Process (AHP). This process constructs matrices using subject pairwise comparisons [3, and 43].
- Propagate data using a variety of imputation methods [46].

3. MACHINE LEARNING EXPERIMENTS

General Description

This section describes two sets of machine learning experiments based on data gathered from two separate corporations.

Prior to conducting the experiments, it was necessary to decide which ML approach to adopt. A neural network paradigm for creating models to explore data-starved domains seemed like a natural choice. This decision was based upon the author's previous successes using neural networks to model software metrics [8, 9, 10, and 11].

Advantages of using neural networks include [24]: the ability to deal with domain complexity, ability to generalize, along with adaptability, flexibility, and parallelism.

There is also support in the literature for applying neural networks in estimation tasks [29, 40, and 44]. However, some researchers consider the relative merits of neural nets over other machine learning techniques (e.g. decision tree learning) an open issue [42].

A supervised neural network can be viewed as a directed graph composed of nodes and connections (weights) between nodes. A set of vectors, referred to as a training set, is presented to the neural network one vector at a time. Each vector consists of input values and output values. In figure 1 below, the inputs are $\boldsymbol{x}_0$ through $\boldsymbol{x}_{N-1}$ and the output is $\boldsymbol{y}$. The goal of a neural network is to characterize a relationship between the inputs and outputs for the whole set of vectors. During the training of a neural network, inputs from a training vector propagate throughout the network. As inputs traverse the network, they are multiplied by appropriate weights and the products are summed up. In Figure 1, this is $\boldsymbol{w}_i \cdot \boldsymbol{x}_i$. If the summation exceeds some specified threshold for a node, then output from that node serves as input to another node. This process repeats until the neural network generates an output value for the corresponding input portion of a vector. This calculated output value is compared to the desired output and an error value is determined. Depending on the neural network algorithm, either the weights are recalibrated after every vector, or after one pass (called an epoch) through all the training vectors. In either case the goal is to minimize the error total. Processing continues until a zero error value is achieved, or training ceases to converge. After training is properly completed, the neural network model which characterizes the relationship between inputs and outputs for all the vectors is embedded *within the architecture (the nodes and connections) of the neural network*. After successful completion of training, a neural network architecture is frozen and tested against an independent set of vectors called the test set. If properly trained, the neural network produces reasonable results against the test suite.

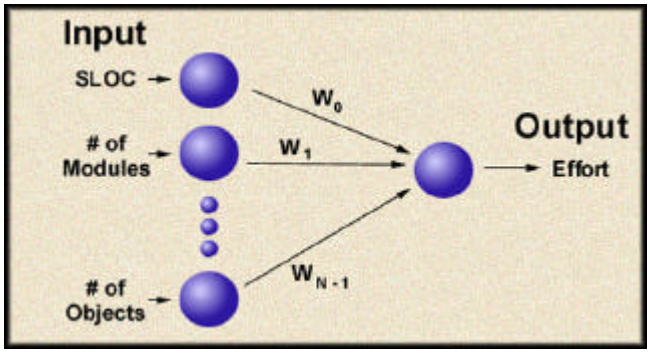

Figure 1: Sample Neural Network

A variant of the backpropagation neural network, called the quickprop, builds all the models for the experiments. The quickprop algorithm, developed by Fahlman, converges much faster than a typical backpropogation approach [19]. It uses the higher-order derivatives in order to take

advantage of the curvature [19]. The quickprop algorithm uses second order derivatives in a fashion similar to Newton's method. Using quickprop in all the experiments also ensures stability and continuity.

As discussed earlier, a bottom-up perspective uses data from products instead of projects. Inputs for the neural network will consist of software metrics extracts from software. Normally, this could include size, complexity, vocabulary, cohesion, coupling, etc. However, the premise of this work is whether it is feasible to use metrics which could be gathered *early* in the software life cycle to predict project effort. Therefore, only a size-based metric, Source Lines of Code (SLOC), is used as input. The other types of product metrics, vocabulary, complexity, coupling, cohesion, do not appear until the code is actually written. The output value, programming effort, represents the number of hours needed to code the form. A vector in a neural network experiment corresponds to the metrics extracted for an actual program. Table one depicts 104 vectors (computer programs) consisting of one input value, the program's size, represented as Source Lines of Code (SLOC), and one output value, represented by effort. Effort is the number of hours that was needed to write the program.

Vector Number	SLOC (Input)	Effort (Output)
1	21	1
2	58	1
:	:	:
103	1253	121
104	2796	160

Table 1: Training Data in the First Experiment

All experiments use a fully-connected neural network architecture of 1-3-1 (see Figure 2), meaning one input, one output, and one hidden layers of consisting of three nodes.

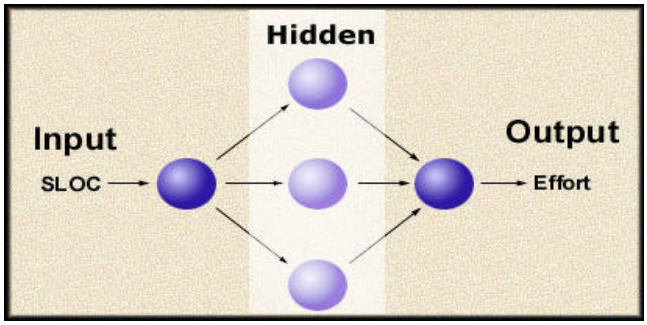

Figure 2: 1-3-1 Neural Network Architecture

In order to minimize experimental variance among experiments, we standardized the experimental process. Different components of each neural network model remain constant. Alpha, which represents how quickly a neural network learns, may range from zero to one. Alpha for these experiments is one. Momentum, a variable which helps neural networks break out of local minima, may also range from zero to one. Momentum also is set to 1. The threshold function is a function associated with each node after the input layer. Function selection determines when a node fires. Firing a node essentially propagates a value further through the network. All experiments use an asymmetrical sigmoid function as a threshold function.

One scan through the training data is considered an epoch. Each experiment was limited to a maximum of 1,000 epochs. An automated log kept track of the most accurate test results and corresponding epoch. A thousand epochs offer sufficient opportunity for the neural network to approach a solution. In previous neural network experiments [11] a solution was reached within 100 epochs.

The "most accurate test results" is defined as number of correct matches with 25 percent, or pred(25), of the actual test values.

Data Suite 1: Electronic Commerce Software

The first dataset consists of 104 vectors (program units). Each vector contains metrics (see Table one) extracted from three major subsystems (buyer/supplier clients, and electronic commerce manager) within one software product. This product supports electronic commerce (procurement) in the process industry.

Data Suite 2: Fleet Management Software

The second dataset consists of 434 vectors extracted from a fleet management software corporation.

The baseline code for both datasets is based on software written in Delphi. Furthermore, these datasets are completely independent of each other in terms of personnel, products, and actual metrics.

Experiment Set 1

The first set of experiments used the Fleet management data as the training set (434 vectors) and the electronic commerce data as the test set (104 vectors). Table two illustrates this configuration.

	Vector	SLOC	Effort
Training Data	1	26	1
	:	:	:
	434	4398	245
Test Data	1	15	1
	:	:	:
	104	2796	160

Table 2: Training/Test Data for the First Experiment

An experiment consists of training a neural network for 1000 epochs and recording when the trained neural network generated the most accurate results against the test suite. Normally, the best test results occurred within the first 150 training epochs.

A total of ten experiments are performed. Each experiment starts at epoch zero and all the weights are assigned random values. The first set of experiments ran on average for about 44 seconds on a 450 MHZ computer.

Table three shows the results from the first set of experiments. These results show how accurate the neural network is at estimating programming effort on a "programming unit" basis. The 104 in the "Possible Correct" column represents the total number of vectors in the test suite. The "Actual Correct" column shows the highest number of correct effort estimates for those vectors in the test suite for a given epoch. Hence, a perfect score would be 104. An estimate is considered correct if it is within 25% of the actual value. "Percent Correct" divides the "Actual Correct" by the "Possible Correct".

Experiment	Possible Correct	Actual Correct	Percent Correct pred(25)
1	104	11	11%
2	104	10	10%
3	104	11	11%
4	104	7	7%
5	104	12	12%
6	104	2	2%
7	104	8	8%
8	104	10	10%
9	104	14	13%
10	104	10	10%

Table 3: Product-based Results for Electronic Commerce Test Data

Overall, this constitutes an average of 9% for pred(25). This means a neural network can estimate the effort required to develop a single module with 25 percent accuracy reliably 9 percent.

These results are quite low. However, in the early life cycle phases the concern is estimating **project** effort not **product** efforts. To assess this approach from the project perspective the actual product effort values are summated and compared to the total software development effort for the project. Table four shows these results.

Experiment Number	Project Devel opment Effort		Project Accuracy
	Actual	Calculated	
1	2083	1958	**-6%**
2	2083	1962	**-6%**
3	2083	1998	**-4%**
4	2083	2238	**7%**
5	2083	2110	**1%**
6	2083	3412	**64%**
7	2083	2555	**23%**
8	2083	2104	**1%**
9	2083	2083	**0%**
10	2083	1777	**-15%**

Table 4: Project-based Results for Electronic Commerce Test Data

Columns two and three for table four represent the summation of the 104 actual and calculated effort values for a given experiment. Collectively, the electronic commerce consisted of 104 program units which required a total of 2083 hours of effort. "Project Accuracy" is equal to 1 – (Calculated/Actual).

These results show an accuracy of pred(25) ninety-percent of the time. The average difference between actual and calculated for these ten experiments is 13 percent.

Experiment Set 2

The second set of experiments uses the electronic commerce data as the training set (104 vectors) and the Fleet management data as the test set (434 vectors). Essentially the training and test sets exchanged places (see Table 2). These experiments ran on average for about 35 seconds on a 450 MHZ computer.

This second set of experiments raises two questions. Would 104 training vectors be a sufficient amount for developing a reasonable neural network model? Second, how does the data extrapolation problem affect the results? The electronic commerce data contains maximum values for SLOC and effort of 2796 and 160 respectively. The Fleet Management data contains SLOC and effort maximum values of 4398 and 245 respectively. Thus, a trained neural network could not produce an effort value greater than 160.

One approach to address this isssue is to scale the results by the ratio of the maximum SLOC values for each data set. In this case the scale factor is 1.57 (4398 divided by 2796).

Table five shows the results from the second set of experiments. Column one is the actual number of vectors in the test suite. Column two represents the actual number of calculated estimates, on a programming unit basis, within 25 percent of the actual values. Column three is column two divided by column one. Column four demonstrates how accuracy improves by applying the scaling factor described above. And column five is column four divided by column one.

Possible Correct	Actual Correct (raw scores)	% Correct pred(25) (raw scores)	Actual Correct (scaled)	% Correct pred(25) (scaled)
434	130	30%	142	33%
434	133	31%	96	22%
434	78	18%	179	41%
434	118	27%	172	40%
434	132	30%	136	31%
434	130	30%	117	27%
434	134	31%	68	16%
434	146	34%	241	56%
434	130	30%	117	27%
434	106	24%	118	43%

Table 5: Product-based Results for Fleet Management Test Data

The overall raw and scaled averages for the individual software components is 29 and 34 percent respectively using a pred(25). These results are better than the first set of experiments, but are quite low.

As described in the first set of experiments, the focus is on the project effort, not the product effort. Once again the program unit efforts are summated. Table six shows the project assessments for the 10 experiments using both raw and scaled results.

Actual Proj. Development Effort	Calc. Proj. Dev. Effort (Raw Score)	Project Accuracy (Raw Score)	Calc. Proj. Dev. Effort (Scaled)	Project Accuracy (Scaled)
15949	9464	-41%	**14887**	**-7%**
15949	8787	-45%	**13821**	**-13%**
15949	9066	-43%	**14261**	**-11%**
15949	9809	-38%	**15429**	**-3%**
15949	9281	-42%	**14599**	**-8%**
15949	8753	-45%	**13768**	**-14%**
15949	8640	-46%	**13591**	**-15%**
15949	10855	-32%	**17074**	**7%**
15949	8915	-44%	**14022**	**-12%**
15949	9299	-42%	**14627**	**-8%**

Table 6: Project-based Results for Fleet Management Test Data

The first column represents the total effort for the fleet management project. Column two shows the sum of the calculated estimates. The extrapolation problem is evident by the fact that project calculations for all the experiments underestimated the actual effort deployed. Column three shows the underestimation in percent format.

Column 4 scales the results from column 2. Column 5 depicts how close the scaled calculated estimates are to the actual values. Visual inspection reveals that scaling plays an important role in improving the results. None of the project estimates (column 3) are within 25 percent, pred(25), of the actual project estimate. However, the scaled project estimate produces 100 percent for pred(25).

Both sets of experiments show that it is quite feasible to estimate project effort from a bottom-up perspective. Issues related to adopting such an approach are discussed in the next section.

4. DISCUSSION

The experiments show that it is possible to summate and scale a set of program units in order to arrive at the total programming effort required for a project. The next question is "How to incorporate this programming effort estimate into a project estimate early in the life cycle?"

This approach is not intended to be the panacea for all software development methodologies. However, for those life cycle methodologies which create a prototype very early in the process, then it is very appropriate. In this situation stakeholders could create a list of components and estimate the SLOC for each of the envisioned components. These SLOC estimates serve as input to a trained neural network. The calculated results could be summated to generate a **project programming effort (PPE)**.

Once the PPE is determine it needs to be scaled to account for extrapolation. This is accomplished by dividing the largest SLOC estimate by the largest SLOC value found in the training set. We denoted the scaled *PPE* as *SPPE*.

There are two potential problems with generating the *SPPE*. A poor estimate in terms of the largest component will directly impact the calculated *SPPE*. The scaling approach assumes a linear relationship. Answering these questions will require further research.

This *SPPE* could then be multiplied additional factors including testing effort, programming language, and administrative overhead in order to determine project effort.

Additionally, the *PPE* could also be integrated into the COCOMO II Cost Model. In this case, the PPE would replace the *SIZE* parameter in the COCOMO II Cost Model equation.

Neither project (Electronic Commerce, Fleet Management) was developed in an SEI level 3 environment. Most likely each environment would be characterized as an SEI level 1 organization. The experiments show that it is possible to construct reasonable project programming estimates in the context of a poorly defined process.

5. CONCLUSIONS

Adopting a bottom-up approach for estimating project effort is quite feasible. It is evident that poor product results do not necessarily imply poor project results.

Using empirical data gathered from two separate corporations and applying a neural network approach produced average project effort estimates of 13 and 10 percent for each set of experiments. The experiments also produced 90% and 100% accuracy for pred(25).

Scaling the data, based on maximum SLOC values, helps to compensate for extrapolation issues. Results improved from an average accuracy of 42 to within 10 percent of actual values.

This approach offers very good potential for those life cycle methodologies which incorporate prototyping early in the life cycle.

6. FUTURE DIRECTIONS

The process of adjusting results to account for extrapolation issues is easy to implement. However, more analysis needs to occur in order to refine the scaling of the *PPE*.

This approach focused on neural networks. A logical step

would be to apply other ML algorithms and to formulate a hybrid model.

Another extension of this work would be to incorporate additional metrics into the experiments. Possible candidates include total number of modules, total number of objects, and unique number of objects. All three are available early in the software life cycle.

Experiments could be benchmarked against the COCOMO II Cost model. Integrating this approach with COCOMO II could explore the impact of programming languages, process level, and reuse within a development environment.

Finally, running more experiments with additional data sets would further validate these results.

7. ACKNOWLEDGMENTS

The author would like to thank Tim Menzies for his insights and suggestions in the preparation of this paper.

REFERENCES

1. Abts, C., Clark, B., Devnani-Chulani, S., Horowitz, E., Madachy, R., Reifer, D., Selby, R., and Steece, B., "COCOMO II Model Definition Manual," Center for Software Engineering, University of Southern California, 1998.

2. Albrect, A.J., Gaffney, J.E. Jr., "Software Function, Source Lines of Code and DevelopmentEffort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, 2 4, 1978, pp. 345-361.

3. Barker S., Sheppard, M., M. Aylett, "The Analytic Hierarchy Process and Data-less Prediction," *Proceedings 10th European Software Control and Metrics Conference*, Herstmonceux, Sussex, England, 1999.

4. Berger, Charles, *Oracle9i Data Mining*, June 2001, Pp. 1 – 16. Available at www.oracle.com.

5. Bisio, R., F. Malabocchia, "Cost Estimation of Software Projects Through Case-Base Reasoning." *Case-Based Reasoning Research and Development. First International Conference*, ICCBR-95 Proceedings, 1995, Pp.11-22.

6. Boehm, B., *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

7. Boehm, B., et al., "Cost Models for Future Software Life Cycle Process: COCOMO 2," Annals of Software Engineering, 1995.

8. Boetticher, G., K. Srinivas and D. Eichmann, "A Neural Net-Based Approach to Software Metrics," *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, June 1993, Pp. 271-274. Available from http://nas.cl.uh.edu/boetticher/publications.html

9. Boetticher, G. and D. Eichmann, "A Neural Net Paradigm for Characterizing Reusable Software," *Proceedings of the First Australian Conf. on Software Metrics*, November 1993, Pp. 41-49. Available from http://nas.cl.uh.edu/boetticher/publications.html

10. Boetticher, G., "Characterizing Object-Oriented Software for Reusability in a Commercial Environment," *Reuse '95 Making Reuse Happen – Factors for Success*, Morgantown, WV, August 1995. Available from http://nas.cl.uh.edu/boetticher/publications.html

11. Boetticher, G., "An Assessment of Metric Contribution in the Construction of a Neural Network-Based Effort Estimator," Second Int. Workshop on Soft Computing Applied to Soft. Engineering, 2001. Available from http://nas.cl.uh.edu/boetticher/publications.html

12. Briand, Lionel C., Victor R. Basili, and William Thomas. A Pattern Recognition Approach for Software Engineering Data Analysis. *IEEE Trans. on Soft. Eng.*, November 1992, Pp. 93-942.

13. Campbell, R.L., S.D. Conte and M.K. Rathi, "Early Predictions of Software Size and Effort," *Technical Report SERC-TR-10-P*, Software Engineering Research Center, Purdue University, 1988.

14. Canegie Mellon Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving Software Process*, Addison-Wesley, Reading, Mass., 1995.

15. Chulani, S., and Boehm, B., and B. Steece, "Bayesian Analysis of Empirical Software Engineering Cost Models", *IEEE Transaction on Software Engineering*, 25 4, July/August, 1999.

16. Cordero, R., M. Costramagna, and E. Paschetta. "A Genetic Algorithm Approach for the Calibration of COCOMO-like Models," *12th COCOMO Forum*, 1997.

17. Curtis, B., Personal Conversation, *International Conference on Software Engineering*, Baltimore, Maryland, May, 1993.

18. Delany, S.J., P. Cunningham, "The Application of Case-Based Reasoning to Early Project Cost Estimation and Risk Assessment," *Department of Computer Science, Trinity College Dublin, TDS-CS-2000-10*, 2000.

19. Fahlman, S.E., *An Empirical Study of Learning Speed in Back-Propagation Networks*, Tech Report CMU-CS-88-162, Carnegie Mellon University, September 1988.

20. Fenton, N.E., and Neil M, "Software Metrics: Roadmap", *The Future of Software Engineering* (Ed. Anthony Finkelstein) 22nd International Conference on Software Engineering, ACM Press ISBN 1-58113-253-0, 2000, Pp 357-370.
21. Finnie, G.,R., Wittig, G.,E., J.M. Desharnais, "Estimating software development effort with case-based reasoning," *Proceedings of International Conference on Case-Based Reasoning*, D. Leake, E. Plaza, (Eds), 1997, Pp.13-22.
22. Halstead, M.H., *Elements of Software Science*, Elsevier, NY, 1977.
23. Heemstra, F. "Software Cost Estimation," *Information and Software Technology*, October 1992, Pp. 627-639.
24. Hertz, J., Krogh A., R.G. Palmer., *Introduction to the Theory of Neural Computation*, Addison Wesley, New York, 1991.
25. Hihn, J., H. Habib-Agahi, "Cost Estimation of Software Intensive Projects: A Survey of Current Practices," *Proceed-ings of the International Conference on Software Engineering*, 1991, Pages 276-287.
26. Hinton, G.E., "How Neural Networks Learn from Experience," *Scientific American*, September, 1992, Pp. 144-151.
27. Hodgkinson, A.C., Garratt, P.W., "A Neurofuzzy Cost Estimator," *Proc. 3rd International Conf. Software Engineering and Applications (SAE)*, 1999, pp. 401-406.
28. Kadoda, G., Cartwright, M., Chen, L. and Shepperd, M., "Experiences Using Case-Based Reasoning to Predict Software Project Effort," *Empirical Software Engineering Research Group Technical Report*, Bournemouth University, January 27 2000.
29. Kumar, S., Krishna, B. A., Satsangi, P.J., "Fuzzy Systems and Neural Networks in Software Engineering Project Management," *Journal of Applied Intelligence*, 4, 1994, Pp. 31 - 52.
30. McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, 2 4, December 1976, pp. 308-320.
31. Mendonca, M, and N.L. Sunderhaft, "Mining Software Engineering Data: A Survey," Data & Analysis Center for Software, 1999.
32. Menzies, T., "Practical Machine Learning for Software Engineering and Knowledge Engineering," Handbook of Software Engineering and Knowledge Engineering, 2001. Available from http://tim.menzies.com/pdf/00ml.pdf.
33. Menzies, T. and J.D. Kiper, "Machine Learning for Requirements Engineering," Submitted to KCAP-2001, 2001.
34. Mukhopadhyay, Tridas, and Sunder Kekre, "Software effort models for early estimation of process control applications," *IEEE Transactions on Software Engineering*, 18 (10 October), 1992, Pp. 915-924.
35. Paulk, M.C., and B. Curtis and M.B. Chrissis and C.V. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software*, 10 4, July, 1993, Pp. 18-27.
36. Porter, A.A., and R.W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software*, March, 1990, Pp. 46-54.
37. Prietula, M., S. Vicinanza, T. Mukhopadhyay, "Software effort estimation with a case-based reasoner," *Journal of Experimental and Theoretical Artificial Intelligence*, 8(3-4), 1996, Pp. 341-363.
38. Putnam, L.H., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," *IEEE Transactions on Software Engineering*, 2 4, 1978, pp. 345-361.
39. Quinlan, R., *C4.5: Programs for Machine Learning*, Morgan Kaufman, 1992.
40. Samson, B., Ellison, D., Dugard, P., "Software Cost Estimation Using an Albus Perceptron," Information and Software Technology, 1997, pp. 55-60.
41. Seidman, C., *Data Mining with Microsoft SQL® Server™ 2000*, Microsoft Press, 2001.
42. Shavlik, J.W., Mooney, R.L., and G.G. Towell, Symbolic and Neural Learning Algorithms: An Experimental Comparison, *Machine Learning*", 1991, Pp. 111-143.
43. Sheppard, M., M. Cartwright, "Predicting with Sparse Data," *7th IEEE Intl. Metrics Symp.*, London, UK, April 4-6, 2001
44. Srinivasan, K., and D. Fisher, "Machine Learning Approaches to Estimating Software Development Effort," *IEEE Trans. Software Engineering*, February, 1995, Pp. 126-137.
45. The Standish Group, *CHAOS Chronicles*, Standish Group Internal Report, 1995.
46. Strike, K., El-Emam, K., Madhavji, N.. Software Cost Estimation with Incomplete Data. *NRC/ERB-1071*: 50 pages. January 2000.
47. Wittig, G., Finnie, G., "Estimating software development effort with connectionist models," Information and Software Technology, 1997, pp. 469-476.