An Assessment of Metric Contribution in the Construction of a Neural Network-Based Effort Estimator

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@uhcl.cl.uh.edu

Abstract

The research literature contains various models for estimating programming effort. Traditional, well understood, mature approaches include Putnam's SLIM [15], Albrect's function point method of estimation [2], as well as COCOMO [3] and COCOMO II [1, 4]. Besides these traditional approaches, various machine learning techniques, including neural networks, [16, 17, 12] have evolved. At the foundation of these models is a set of cost drivers based upon process (e.g. process maturity), project (e.g., reuse, platform), personnel (e.g. team cohesion, personnel experience), and/or product measures (e.g. size and interface). Historically, SLOC (Source Lines of Code) metric is the most popular product metric used in the formulation of the various models. It is simple to calculate and provides a quick answer. However, it is an oversimplification of the product measure contribution to what characterizes programming effort. Using only SLOC ignores the contribution of other internal product metrics, such as complexity and vocabulary, in determining programming effort. What is needed is a more representative product metric which is both simple to calculate and provides a quick answer. This paper describes a neural network approach for characterizing programming effort based on internal product measures. Over thirty-three thousand different neural network experiments were performed upon data derived from a corporate repository. Four different simple metrics (size, vocabulary, complexity, and object) are assessed in terms of their individual contribution to programming effort. Afterwards these simple metrics are combined and assessed to determine the synergistic impact of each of the combinations. Finally, a cross-validation is performed on a second corporate repository.

Keywords
Metrics, measurement, empirical analysis, effort estimation, neural networks

1. Introduction

The research literature contains various models for estimating programming effort. Traditional, well understood, mature approaches include Putnam's SLIM [15], Albrect's function point method of estimation [2], as well as COCOMO [3] and COCOMO II [1, 4]. Besides these traditional approaches, various machine learning techniques, including neural networks, [16, 17, 12] have evolved. At the foundation of these models is a set of cost drivers based upon process (e.g. process maturity), project (e.g., reuse, platform), personnel (e.g. team cohesion, personnel experience), and/or product measures (e.g. size and interface). Effort estimation models include some or all of these types of measures. However, product-based metrics are the most prevalent in effort estimation models. Since the inception of software metrics in the 1970s, the predominant product metric used in measuring programming effort is a Source Lines of Code (SLOC) metric. The appeal lies in SLOC's ability to provide a quick and simple answer applicable at both the developer or project level. The appeal of simplicity is also one of the shortcomings of SLOC since it ignores other common code characteristics such as McCabe's cyclomatic complexity [14] and Halstead's vocabulary [11] metrics.

One of the goals of all effort estimation models is to accurately and reliably predict programming effort. Creating a highly accurate and reliable model is a challenging endeavor. One of the more accurate research models predicts effort estimates of 25% more than 75% of the time [17]. The difficulty is determining which are the answers outside the 25% accuracy. A natural research direction is to improve upon the accuracy and reliability of the models produced.

Regarding product metrics, this raises an important question, "How do various product metrics contribute to the formation of an effort estimator?" Understanding this question will lead to the improvement in the creation of effort estimation models.

In [5] and [6] it was demonstrated for simple well-understood metrics that a neural network approach could generate a network that produces results comparable to that of a traditional polynomial formulation. This validation of the neural network approach against known benchmarks (McCabe and Halstead) show the technique is sound. The

initial research was extended in a commercial setting by employing neural networks to predict areas likely to receive frequent reuses[7]. Identifying highly-reusable objects allows an organization to focus their testing efforts. Collectively all previous works demonstrate that neural networks can be applied to software metric problems in both research and commercial environments. Furthermore, comparative studies [9, 10, 13] show neural networks as a reasonable option for formulating an effort estimation model.

This paper builds upon this previous research and describes a process of applying a neural network approach to the developing of an effort estimating metric. Using data from a company which specializes in business-to-business petrochemical-industry software, over 33,000 experiments were conducted varying test suites, neural network architectures, and input parameters. Different product-based metric combinations are assessed in terms of their contribution to the formulation of an effort estimating tool. The approach cross-validated against data collected from a second corporation's software repository.

2. General framework

The measures for building the various neural network models were extracted from a software project which allows petrochemical-based corporations to conduct complete procurement life-cycle processes over the Internet.

All data derived from the programming units were based on sole-authorship allowing for very precise measurement of effort and minimizes the noise resulting from assessing different programming abilities.

The data suite contains 104 different units. Each unit corresponds to a program written in Delphi and may contain object, variable, type, and constant definitions along with programming logic in the form of functions or procedures. In neural network terminology, each program is considered to be a vector.

For each vector, nine input measures are collected reflecting a program's size, vocabulary, number of objects, and complexity. The output is the effort, in hours, needed to create the unit.

Size. The size category includes two metrics. The SLOC metric is defined as any line of code which is neither blank nor a comment, and a count of the total number of procedures and functions within the unit.

Vocabulary. The vocabulary category includes the four initial metrics defined by Halstead [11], total operands, total operators, unique operands and unique operators.

Objects. The object category includes two metrics, the total number and the unique number of objects in a unit.

Complexity. Complexity refers to the total complexity within a unit. It is based on McCabe's [14] definition of complexity.

Figure 1 shows the general layout of the input and output parameters for a neural network model.

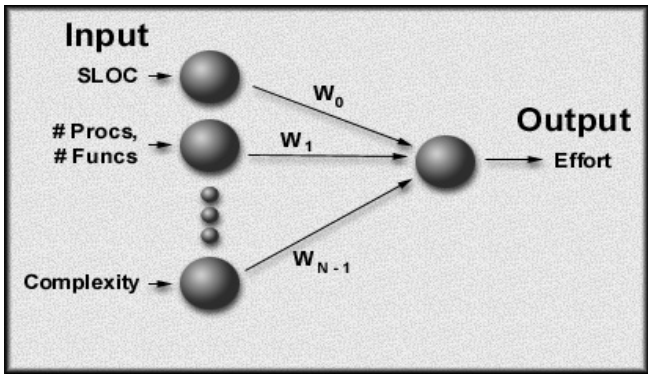

Figure 1: General layout for a Neural Network

By using a powerset combination of the input metric categories it is possible to create 15 different input configurations. These categories are:

- Size,
- Vocabulary,
- Objects,
- Complexity,
- Size and Vocabulary,
- Size and Objects,
- Size and Complexity,
- Vocabulary and Objects,
- Vocabulary and Complexity,
- Objects and Complexity,
- Size, Vocabulary, and Objects,
- Size, Vocabulary, and Complexity,
- Size, Objects, and Complexity,
- Vocabulary, Objects, and Complexity, and
- Size, Vocabulary, Objects and Complexity.

Besides grouping individual vectors into specific metric categories, 100 of the 104 vectors are distributed into ten distinct groups of ten vectors each. For a given neural network architecture and a given number of inputs, each one of the ten groups acts as a neural network test set. The vectors were sorted by effort and evenly distributed amongst the groups with the intent of creating balanced sets. Setting up different testing scenarios serves to validate the integrity of the modeling process.

The remaining four vectors contain minimum and/or maximum values for some of the inputs or output. These vectors always remained in the training set in order to avoid extrapolation problems.

The cross-product of the 15 input categories with the 10 groups yields 150 combinations for building a neural network model. The next question is what type of neural network architectures to apply. It is possible to vary the number of hidden layers along with the number of nodes per hidden layer. A decision was made to limit the maximum number of hidden layers to 3 and to limit the number of

hidden nodes per hidden layer to twice the number of inputs. One method to speed the modeling process was to increase node additions by two. To illustrate these concepts, consider a neural network with 3 inputs and 1 output. The following neural network architectures are some of the possible architectures that can be used to build various models: 3-1 (three inputs, no hidden, one output); 3-1-1 (three inputs, one hidden, one output); 3-3-1 (three inputs, three hidden, one output); 3-5-1 (three inputs, five hidden, one output); 3-1-1-1 (three inputs, one node on first hidden layer, one node on the second hidden layer, one output); 3-3-1-1; 3-5-1-1; 3-1-3-1; 3-3-3-1; 3-5-3-1; 3-1-5-1; 3-3-5-1; 3-5-5-1; 3-1-1-1-1 (three inputs, three hidden layers of one node each, one output); 3-3-1-1-1; ... ; 3-3-5-5-1; and 3-5-5-5-1.

For N inputs, the number of neural network architecture permutations equals: $1 + N + N^2 + N^3$ where

1 means there is only one neural network architecture with zero hidden layers,

N is the number of ways of creating a neural network architecture with one layer,

N^2 is the number of ways of creating a neural network architecture with two layers, and

$N^3$is the number of ways of creating a neural network architecture with three layers.

Table 1 shows the total number of permutations of neural network architectures, metric categories, and group configurations.

Table 1: Total number of Neural Network Models

Metric Category	Number of Inputs	Number of Neural Network Architectures
Size	2	15
Vocabulary	4	85
Objects	2	15
Complexity	1	4
Size and Vocabulary	6	259
Size and Objects	4	85
Size and Complexity	3	40
Vocabulary and Objects	6	259
Vocabulary and Complexity	5	156
Objects and Complexity	3	40
Size, Vocabulary and Objects	8	585
Size, Vocabulary and Complexity	7	400
Size, Objects and Complexity	5	156
Vocabulary, Objects and Complexity	7	400
All metrics	9	820

Table 1: Total number of Neural Network Models

Total number of neural network architectures:	**3319**
Total number of groups per architecture:	**10**
Total number of neural networks:	**33,190**

In order to build and train 33,190 neural networks, an automated neural network program was used. This program is based on Fahlman's [8] quickprop algorithm.

In order to reduce variations between neural network models, several parameters remain constant. These include alpha, the learning rate, which is one; mu, the momentum, which also remains at one; unit type is always asymmetrical; and tolerance which is set to 30 percent. Tolerance is Magnitude of Relative Error, or MRE. It is typically written as pred(30).

As data is read, it is normalized to values between zero and one for processing. For reporting purposes, all results are mapped back to the original ranges of a specific network.

As a way to speed up the experimentation process, the initial approach limits every neural network to 1000 epochs. Using this constraint allows all training to be completed within weeks, rather than months or years. The only unanswered question is whether this would provide enough epochs to sufficiently train the neural network. Experience in [6] and [7] has shown that most neural networks train in less than 1000 epochs, so using a longer training period is often fruitless. In the event that all the neural networks produce low correlation results, then more training will be performed.

There are many ways of measuring "successful" neural network training. Rather than commit to only one method of measuring success, several different forms of "success" are described. For every group within every metric category over all the different neural network architectures, the following seven types of "success" measures are captured:

Training RMS error. This is lowest error value with respect to training.

Test RMS error. This is lowest error value with respect to testing.

Training Correlation. This is the highest correlation value between the calculated and actual training outputs.

Test Correlation. This is the highest correlation value between the calculated and actual test outputs.

Combined Training and Test correlation. This case adds the training and test correlations and keeps track of the highest combined total.

Total train correct. This is the number of training vectors, out of a maximum of 94, that produce a value within the 30 percent threshold range, written as pred(30), of the actual training output.

Total test correct. This is the number of test vectors, out of a

maximum of 10, that produce a value within the threshold range, pred(30), of the actual test output.

In order to illuminate the neural network modeling process, a pseudocode representation follows. This code shows all the key loops for building all of the neural network models.

```
(* Pseudocode algorithm of the process *)
Alpha := 1.0;
Momentum := 1.0;
UnitType := Asym.;
Tolerance := 30%;
MaximumEpochs := 1000;
Loop through all 15 possible metric configurations
  Loop through all 10 groups (* 'A' to 'J' *)
    (* Loop through neural net architectures *)
     HiddenLayers = 0 to 3
      NodesPerLayer = 1 to 2*NumberOfInputs (in net)
         (* Initialize Best Cases *)
         (* This following inner loop will be *)
         (* executed over 33,000 times. *)
         For Epochs := 1 to MaximumEpochs do begin
           Train One Epoch;
           Test One Epoch;
           Update best case values if necessary.
         End Loop;
       Save Best Case and statistical information
     NEXT NodesPerLayer
   NEXT HiddenLayers
  End Loop (* for all 10 groups *)
End Loop (* for Metric Configurations *)
```

This code serves as the foundation for the actual modeling program. Running the program in batch mode generates 150 different files (15 different input configurations times the 10 different groups). Each file contains results of the 7 best cases.

3. Results: general assessment

The statistical results from analyzing a test suite are of more interest and value than from assessing training data. The test suite results provide a benchmark of how well a neural network model performs on novel data. Therefore, all the results in this section focus on the test results.

Table 2 shows the average of all 15 input configurations for each data group. The average correlation exceeds 0.9500 for 7 of the 10 groups. The average correct, pred(30), exceeds 50% for 5 of the 10 groups. A typical group, based upon median results, would have an RMS error around 0.0045, a Test Correlation around 0.9742, and an accuracy of 48%.

Table 2: Test Set Results by Group

Data Group	Ave. Test RMS Error	Ave. Test Correlation	Ave. Accuracy pred(30)
A	0.00251	0.9504	43%
B	0.00380	0.8960	38%
C	0.00249	0.9028	59%
D	0.00194	0.9192	45%
E	0.00252	0.9734	41%
F	0.01052	0.9873	43%

Table 2: Test Set Results by Group

Data Group	Ave. Test RMS Error	Ave. Test Correlation	Ave. Accuracy pred(30)
G	0.00524	0.9750	52%
H	0.00556	0.9935	51%
I	0.00802	0.9763	51%
J	0.00784	0.9901	57%

Ranking each group by column and adding up the rankings shows that groups C, H, and J had the best results and group B the poorest.

Group B had median values for RMS Error, Test Correlation, and Accuracy of 0.000209, 0.9130 and 40% respectively.

Table 3: Group B

Metric Category	Test RMS Error	Test Corr.	Test Accuracy pred(30)
Size	0.000195	0.8288	30%
Objects	0.001240	0.8387	50%
Complexity	0.051631	0.8337	10%
Vocabulary	0.000445	0.9359	30%
Size, Object	0.000180	0.8965	40%
Size, Complexity	0.000194	0.8346	30%
Size, Vocabulary	0.000209	0.9575	40%
Object, Complexity	0.000955	0.8740	50%
Object, Vocabulary	0.000376	0.9237	40%
Complexity, Vocabulary	0.000450	0.9322	30%
Size, Object, Complexity	0.000178	0.9075	50%
Size, Object, Vocabulary	0.000184	0.9198	50%
Size, Complexity, Vocab.	0.000178	0.9130	40%
Object, Complexity, Vocab.	0.000360	0.9297	40%
All	0.000177	0.9141	50%

Group C had median values for RMS Error, Test Correlation, and Accuracy of 0.00015, 0.9254 and 60% respectively.

Table 4: Group C

Metric Category	Test RMS Error	Test Corr.	Test Accuracy pred(30)
Size	0.000169	0.7559	60%
Objects	0.000448	0.8974	60%
Complexity	0.034829	0.7458	00%

Table 4: Group C

Metric Category	Test RMS Error	Test Corr.	Test Accuracy pred(30)
Vocabulary	0.000225	0.8872	50%
Size, Object	0.000104	0.9765	60%
Size, Complexity	0.000150	0.7983	60%
Size, Vocabulary	0.000113	0.9344	70%
Object, Complexity	0.000453	0.9220	60%
Object, Vocabulary	0.000157	0.9254	70%
Complexity, Vocabulary	0.000227	0.9191	50%
Size, Object, Complexity	0.000065	0.9633	60%
Size, Object, Vocabulary	0.000085	0.9620	70%
Size, Complexity, Vocab.	0.000130	0.9576	70%
Object, Complexity, Vocab.	0.000127	0.9344	70%
All	0.000075	0.9627	70%

Group H had median values for RMS Error, Test Correlation, and Accuracy of 0.0000025, 0.9962 and 50% respectively.

Table 5: Group H

Metric Category	Test RMS Error	Test Corr.	Test Accuracy pred(30)
Size	0.000092	0.9729	40%
Objects	0.001152	0.9751	40%
Complexity	0.081708	0.9962	00%
Vocabulary	0.000025	0.9952	40%
Size, Object	0.000069	0.9960	50%
Size, Complexity	0.000061	0.9962	40%
Size, Vocabulary	0.000018	0.9953	70%
Object, Complexity	0.000185	0.9963	50%
Object, Vocabulary	0.000024	0.9963	50%
Complexity, Vocabulary	0.000024	0.9959	50%
Size, Object, Complexity	0.000021	0.9983	60%
Size, Object, Vocabulary	0.000010	0.9982	80%
Size, Complexity, Vocab.	0.000015	0.9962	70%
Object, Complexity, Vocab.	0.000033	0.9967	50%
All	0.000013	0.9984	80%

Group J had median values for RMS Error, Test Correlation, and Accuracy of 0.000025, 0.9956 and 60% respectively.

Table 6: Group J

Metric Category	Test RMS Error	Test Corr.	Test Accuracy pred(30)
Size	0.000046	0.9848	50%
Objects	0.001128	0.9496	40%
Complexity	0.115630	0.9833	00%
Vocabulary	0.000025	0.9956	50%
Size, Object	0.000038	0.9987	60%
Size, Complexity	0.000046	0.9848	50%
Size, Vocabulary	0.000020	0.9942	70%
Object, Complexity	0.000521	0.9818	60%
Object, Vocabulary	0.000025	0.9961	70%
Complexity, Vocabulary	0.000025	0.9960	50%
Size, Object, Complexity	0.000033	0.9986	50%
Size, Object, Vocabulary	0.000014	0.9974	80%
Size, Complexity, Vocab.	0.000015	0.9953	70%
Object, Complexity, Vocab.	0.000025	0.9960	80%
All	0.000016	0.9988	80%

It is interesting to note that Group B, the lowest ranked group, did not produce the lowest individual values. The range of values for each result ranged from 0.000014 to 0.11563 (Group J) for the Test RMS Error, 0.7458 to 0.9765 for the Test Correlation (Group C), and 0 to 80 percent for the Test Accuracy (Groups H and J).

4. Metric contribution to effort

This section describes the contribution of various syntax metrics to effort. Each of the 10 groups mentioned in the previous section were sorted by Test RMS Error then ranked from 1 to 15. This process was repeated for the Test Correlation along with Test Accuracy. As a consequence of this process each of the 15 inputs (metric configurations) had 3 ratings for each of the 10 groups. Averaging the 30 ratings produced the following results presented in table 7.

Table 7: Metric Contribution (3 Categories)

Metric Configuration	Average Rating
All	2.37
Size, Object, Vocabulary	2.47
Size, Complexity, Vocabulary	4.70
Object, Complexity, Vocabulary	4.70

Table 7: Metric Contribution (3 Categories)

Metric Configuration	Average Rating
Size, Vocabulary	5.03
Object, Vocabulary	5.20
Size, Object, Complexity	5.27
Size, Object	6.23
Size, Complexity	8.63
Complexity, Vocabulary	8.93
Object, Complexity	9.20
Vocabulary	9.83
Size	10.43
Objects	11.63
Complexity	14.17

Table 7 shows that using all the syntax metrics and the Size, Object, and Vocabulary produce the best ratings for the Test data. The individual metrics did not fare as well as any combination of metrics. The general pattern suggests a very good synergy among syntax metrics for estimating effort. Ironically Complexity, as an individual metric, did not fare well at estimating effort. However, when added to any other combination it improved the average rating.

Focusing on the accuracy results only, the two best input configurations are all the metrics and Size, Object, and Vocabulary. Once again the individual metrics were not as accurate as compound metrics. Table 8 shows the summary for the accuracy ratings.

Table 8: Metric Contribution (Accuracy Only)

Metric Configuration	Average Accuracy Rating
All	1.50
Size, Object, Vocabulary	1.50
Object, Vocabulary	2.30
Size, Vocabulary	2.60
Size, Complexity, Vocabulary	2.60
Object, Complexity, Vocabulary	2.80
Object, Complexity	4.60
Size, Object, Complexity	5.10
Size, Object	5.80
Size, Complexity	7.20
Objects	7.30
Size	8.30
Complexity, Vocabulary	9.60
Vocabulary	10.20
Complexity	15.00

5. Cross validation

A natural question is how well the neural network predicts for a new software development project. To address this issue, 433 observations were collected from a completely different project from a completely different corporation. These observations serve as the test data for the following neural network experiments.

The input set consisted of all the syntax metrics used in the experiments described in the previous sections. Ten different neural network models are constructed. Each corresponds to the training data used in the previous sections. In essence, each training set consists of nine of the ten groups of data along with the permanent data items.

Each model is trained using the 10 different datasets. This validation approach provides a true measure of the model's predictive abilities. This test yields an average PRED(.30) of 73.26% indicating than on average, the validation results produce estimates within 30% of the actuals 73.26% of the time.

Furthermore, there were extrapolation issues for every input and the effort output. As a consequence, the neural network could not accurately predict effort for those observations outside of the bounds of the training data. Even with the extrapolation issues, it is concluded that the neural network model has reasonably good predictive qualities.

6. Conclusions

This work describes a process of building a neural network-based model for measuring software effort. It demonstrates a process for extracting a set of software metrics from a program, associating the metrics with a program's effort, then constructing a neural network model. An automated process creates over 33,000 different neural network models and collects data for the "best" cases. Product metrics are grouped into four categories: size, vocabulary, objects, and complexity, then analyzed for training and test cases.

Compound metrics, in the form of combinations of individual measures, generally outperformed individual measures. The results from all the experiments justify the inclusion of other product-based measures, such as cyclomatic complexity and vocabulary, into the formation of an effort estimation equation. Overall, the input configuration which included all the syntax measures produced the most accurate model against the test data than any subset combination.

The cross-validation experiments generated reasonable results and warrant further research in this area.

Using a neural network-based approach to formulate an effort estimation model allows for automated construction and assessment of effort estimation models. The actual effort

metric is embedded within the neural network architecture. At present, there is no mechanism for interpreting the architecture of the neural network. This may be perceived as a weakness of a neural network approach. The trend in Software Engineering is towards more sophisticated software processes. This implies that effort estimation models, which mirror these processes, will naturally become more complicated. As a consequence, it will be harder to create and interpret a "white-box" effort estimation model.

7. Future directions

This work focused upon product metric contribution to the formation of an effort estimation model. It raises several questions which spawn further research.

The tendency for adding more product metrics to improve results suggests adding more product metrics to the experimentation process. This could include coupling and cohesion metrics.

This research focused only on product metrics in the formulation of an effort estimation model. This research could be extended by adding process, project, and personnel type metrics. A synthesis with another model, e.g. COCOMO II, might be a logical step.

Neural network modeling is only one of several approaches for constructing an effort estimation model. Applying different modeling approaches, e.g. Case-Based Reasoning, might provide more accurate models.

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

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

[5] 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.

[6] Boetticher, G. and D. Eichmann, "A Neural Net Paradigm for Characterizing Reusable Software," *Proceedings of the First Australian Conference on Software Metrics*, November 1993, pp. 41-49.

[7] Boetticher, G., "Characterizing Object-Oriented Software for Reusability in a Commercial Environment," *Reuse '95 Making Reuse Happen – Factors for Success*, Morgantown, WV, August 1995.

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

[9] Finnie, G.R., Wittig, G.E., Desharnais, J-M, "A Comparison of Software Effort Estimation Techniques: Using Function Points with Neural Networks, Case-Base Reasoning and Regression Models," Journal of Systems Software, 1997, pp. 281-289.

[10]Gray, A.R., MacDonnell, S.G., "A Comparison of techniques for developing predictive models of software metrics," Information and Software Technology, 1997, pp. 425-437.

[11]Halstead, M.H., *Elements of Software Science*, Elsevier, NY, 1977.

[12]Hodgkinson, A.C., Garratt, P.W., "A Neurofuzzy Cost Estimator," *Proc. 3rd International Conf. Software Engineering and Applications (SAE)*, 1999, pp. 401-406.

[13]Krishnamoorthy, S., Fisher, D., "Machine Learning Approaches to Estimating Software Development Effort," *IEEE Transactions on Software Engineering*, 21 2, 1995, pp. 126-137.

[14]McCabe, T.J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, 2 4, December 1976, pp. 308-320.

[15]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.

[16]Samson, B., Ellison, D., Dugard, P., "Software Cost Estimation Using an Albus Perceptron," Information and Software Technology, 1997, pp. 55-60.

[17]Wittig, G., Finnie, G., "Estimating software development effort with connectionist models," Information and Software Technology, 1997, pp. 469-476.