An Analysis of Software Cost Estimation Traditional Models with Neural Network Based Approach

Manpreet Kaur*, Sushil Kumar*, Mamta Sharma
*Dept CSE, RIMT-IET, Mandigobindgarh, Fatehgah Sahib
manumtech@gmail.com, mmta9976@gmail.com, sushilgarg70@yahoo.com

Abstract: *Software effort estimation actually encompasses all estimation, risk analysis, scheduling, and SQA/SCM planning. However, in the context of set of resources, planning involves estimation - your attempt to determine how much money, how much effort, how many resources, and how much time it will take to build a specific software-based system or product. In this paper we will study the efficiency of Neural Network based cost estimation model with the traditional cost estimation model like Halstead Model, Bailey-Basili Model, Doty Model. We conclude our result with the proposal of Neuron based Model basis on Back propagation Technique.*

1. Introduction

A Neural Network (NN) is a computer software (and possibly hardware) that simulates a simple model of neural cells in animals and humans. The purpose of this simulation is to acquire the intelligent features of these cells. In this document, when terms like neuron, neural network, learning, or experience are mentioned, it should be understood that we are using them only in the context of a NN as computer system. NNs have the ability to learn by example, e.g. a NN can be trained to recognize the image of car by showing it many examples of a car.

2. Literature Survey

Accurate estimate means better planning and efficient use of project resources such as cost, duration and effort requirements for software projects especially space and military projects [1], [2]. Efficient software project estimation is one of the most demanding tasks in software development. Problem of inaccurate estimate for projects and in many cases inability to set the correct release day for their software correctly lead to inefficient use of project resources. Unfortunately, software industry suffers the problem of incorrect estimate for projects and in many cases inability to set the correct release day for their software correctly. This leads to many losses in their market, e.g. risk due to low quality of the deliverables and penalties for missing the deadlines. Normally, estimation is performed using only human expertise [3], [4], but recently attention has turned to a variety of computer-based learning techniques.

In 1995, Standish Group served over 8,000 software projects for the purpose of budget analysis. It was found that 90% of these projects exceeded its initially computed budget. Moreover, 50% of the completed projects lake the original requirements [5]. From these statistics, it can be seen how prevalent the estimation problem is. Evaluation of many software models were presented in [6], [7], [8].

Numerous models were explored to provide better effort estimation [9], [10], [11], [12]. In [4], [13], authors provided a survey on the effort and cost estimation models.

Serious research in the Neural Network area is started in the 1950's and 1960's by researchers like Rosenblatt (Perceptron), Widrow and Hoff (ADALINE). In 1969 Minsky and Papert wrote a book exposing Perceptron limitations. This effectively ended the interest in neural network research. In the late 1980's interest in NN increased with algorithms like Back Propagation, Cognitrons and Kohonen. (Many of them where developed quietly during the 1970s)

In the literature of Neural Networks (NNs) The following function is called a Sigmoid function.:

$$s(x)= 1/ (1 + e^{-a * x})$$

The coefficient a is a real number constant. Usually in NN applications a is chosen between 0.5 and 2. As a starting point, you could use a=1 and modify it later when you are fine-tuning the network. Note that $s(0)= 0.5$, $s(\infty)= 1$, $s(-\infty)=0$. (The symbol ∞ means infinity).

The Sigmoid function is used on the output of neurons. In a NN context, a neuron is a model of a neural cell in animals and humans. This model is simplistic, but as it turned out, is very practical. In NN the inputs simulate the stimuli/signals that a neuron gets, while the output simulates the response/signal which the neuron generates. The output is calculated by multiplying each input by a different number (called weight), adding them all together, then scaling the total to a number between 0 and 1.

The following diagram shows a simple neuron with:

1. Three inputs $[x_1, x_2, x_3]$. The input values are usually scaled to values between 0 and 1.
2. Three input weights $[w_1, w_2, w_3]$. The weights are real numbers that usually are initialized to some random numbers. Do not let the term weight mislead you, it has nothing to do with the physical sense of weight, in a programmer context, think of

the weight as a variable of type float/real that you can initialize to a random number between 0 and 1.

3. One output is shown as *z*. A neuron has one (and only one) output. Its value is between 0 and 1. It can be scaled to the full range of actual values.

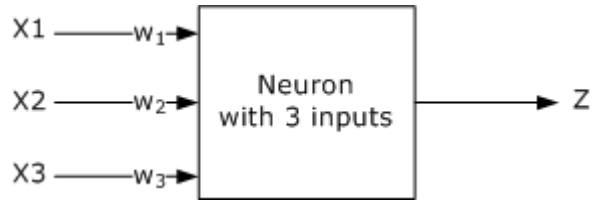

Fig. 1 Neuron Model with 3 inputs

Let

$$d = (x_1 * w_1) + (x_2 * w_2) + (x_3 * w_3)$$

In a more general fashion, for n number of inputs:

$$d = \sum_{i=1}^{n} x_i w_i$$

Let θ be a real number which we will call Threshold. Experiments have shown that best values for θ are between 0.25 and 1. Again, in a programmer context, θ is just a variable of type float/real that is initialized to any number between 0.25 and 1. When sigmoid function, s( ), is applied:

$$z = s(d + \theta)$$

This says that the output z is the result of applying the sigmoid function on (d + q). In NN applications, the challenge is to find the right values for the weights and the threshold.

The following diagram shows a Back Propagation NN:

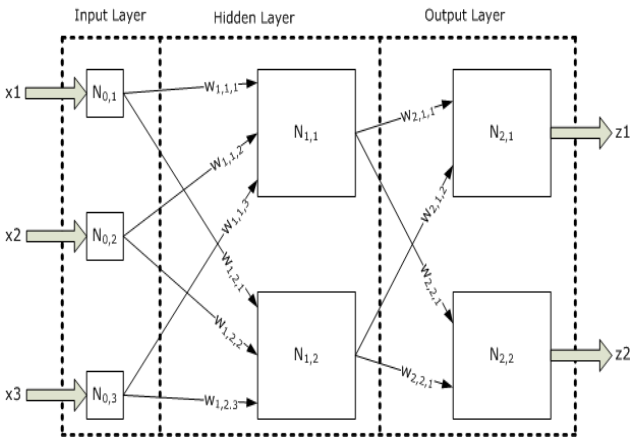

Figure 2: Back Propagation Network

The above NN consists of three layers:

- Input layer with three neurons.
- Hidden layer with two neurons.
- Output layer with two neurons.

The output of a neuron in a layer goes to all neurons in the following layer. Each neuron has its own input weights. The weights for the input layer are assumed to be 1 for each input. In other words, input values are not changed and the output of the NN is reached by applying input values to the input layer, passing the output of each neuron to the following layer as input.

The Back Propagation NN must have at least an input layer and an output layer. It could have zero or more hidden layers.

The number of neurons in the input layer depends on the number of possible inputs we have, while the number of neurons in the output layer depends on the number of desired outputs. The number of hidden layers and how many neurons in each hidden layer cannot be well defined in advance, and could change per network configuration and type of data. In general the addition of a hidden layer could allow the network to learn more complex patterns, but at the same time decreases its performance. You could start a network configuration using a single hidden layer, and add more hidden layers if you notice that the network is not learning as well as you like e.g. suppose we have a bank credit application with ten questions, which based on their answers, will determine the credit amount and the interest rate. To use a Back Propagation NN, the network will have ten neurons in the input layer and two neurons in the output layer.

The Back Propagation NN works in two modes, a supervised training mode and a production mode. The training can be summarized as follows:

First, start by initializing the input weights for all neurons to some random numbers between 0 and 1, then:

i. Apply input to the network.
ii. Calculate the output.
iii. Compare the resulting output with the desired output for the given input. This is called the error.
iv. Modify the weights and threshold q for all neurons using the error.
v. Repeat the process until error reaches an acceptable value (e.g. error < 1%), which means that the NN was trained successfully, or if we reach a maximum count of iterations, which means that the NN training was not successful.

A suitable training algorithm can be used for updating the weights and thresholds in each iteration (step IV) to minimize the error.

Changing weights and threshold for neurons in the output layer is different from hidden layers. Note that for the input layer, weights remain constant at 1 for each input neuron weight.

The literature considered the mean magnitude of relative error (MMRE) as the main performance measure.

The value of an effort predictor can be reported many ways including MMRE. MMRE value is computed from the relative error, or RE, which is the

relative size of the difference between the actual and estimated value:

RE.i = (estimate.i - actual.i) / (actual.i)

Given a data set of of size "D", a "Training set of size "(X=|Train|) <= D", and a "test" set of size "T=D-|Train|", then the mean magnitude of the relative error, or MMRE, is the percentage of the absolute values of the relative errors, averaged over the "T" items in the "Test" set; i.e.

MMRE.i = abs(RE.i)

MMRE = 100/T*(MRE.1 + MRE.2 + ... + MRE.T)

The mean magnitude of relative error (MMRE) can also be written as:

$$MMRE = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{y_i}$$

Where y_i represents the i^{th} value of the effort and $\hat{y}_i$ is the estimated effort.

The another evaluation criteria to measure the performance of the developed models using n measurements selected to be the route mean of the sum square of the error:

$$RMSSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where y_i represents the ith value of the effort and $\hat{y}_i$ is the estimated effort.

3. Result & Discussion

The dataset of [10] is used for the comparison of different models. In this dataset, there is empirical data in terms of KLOC, Function Point and Effort values of 18 projects as shown in table I.

The data of first 13 projects is used as training data for the Neural Network and data of last 5 projects is used as testing data of the trained Neural Network. The neural network used is backpropagation based Neural Network that consists of two neurons in input layer, two neurons in the hidden layer and one neuron in the output layer. In the testing phase the calculated efforts and errors using different models is shown in table 1 and table 2 respectively.

Table1. Data of Actual Effort Required

Project No.	KLOC	Function Point	Actual Effort (in person-hour)
1	95.2	31	125.8
2	50.2	21	90
3	56.5	22	81
4	56.5	21	91.8
5	32.1	38	42.6
6	67.5	29	98.4
7	15.8	29	20.9
8	10.5	34	10.3
9	21.5	31	28.5
10	5.1	29	9
11	4.2	17	8
12	9.8	32	8.3
13	22.1	38	6
14	7	29	8.9
15	88.6	45	100.7
16	10.7	32	17.6
17	13.5	29	25.9
18	105.8	39	148.3

4. conclusion

The performance of the Neural Network based effort estimation system and the other existing Halstead Model, Walston-Felix Model, Bailey-Basili Model and Doty Model models is compared for effort dataset available in literature [15]. The results show that the Neural Network system has the lowest MMRE and RMSSE values i.e. 12.657 and 18.587 respectively. The second best performance is shown by Bailey-Basili software estimation system with 21.385 and 25.1345 as MMRE and RMSSE values. Hence, the proposed Neuro based system is able to provide good estimation capabilities. It is suggested to use of Neuro based technique to build suitable generalized type of model that can be used for the software effort estimation of all types of the projects.

Table 2: Error Calculated In Various Efforts Estimation Models

Perform-ance Criteria	Model Used			
	NN System	Halstead Model	Bailey-Basili Model	Doty Model
MMRE	12.657	155.645	21.385	302.5023
RMSSE	18.587	318.718	25.1345	299.4742

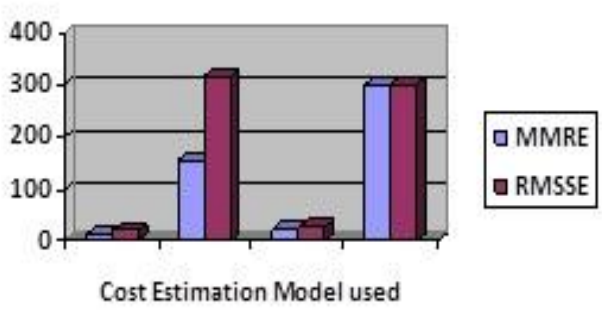

Fig 3 Comparative Analysis of different Cost Estimation Models

References:

1. L. C. Briand, K. E. Emam, and I. Wieczorek, "Explaining the cost of european space and military projects," in ICSE '99: Proceedings of the 21st international conference on Software engineering, (Los Alamitos, CA, USA), pp. 303–312, IEEE Computer Society Press, 1999.
2. "Estimating software projects," SIGSOFT Softw. Eng. Notes, vol. 26, no. 4, pp. 60–67, 2001.
3. J. W. Park R, W. Goethert, "Software cost and schedule estimating: A process improvement initiative," tech. report, 1994.
4. M. Shepper and C. Schofield, "Estimating software project effort using analogies," IEEE Tran. Software Engineering, vol. 23, pp. 736–743, 1997.
5. T. S. Group, CHAOS Chronicles. PhD thesis, Standish Group Internet Report, 1995.
6. M. Boraso, C. Montangero, and H. Sedehi, "Software cost estimation: An experimental study of model performances," tech. report, 1996.
7. O. Benediktsson, D. Dalcher, K. Reed, and M. Woodman, "COCOMO based effort estimation for iterative and incremental software development," Software Quality Journal, vol. 11, pp. 265–281, 2003.
8. T. Menzies, D. Port, Z. Chen, J. Hihn, and S. Stukes, "Validation methods for calibrating software effort models," in ICSE '05: Proceedings of the 27th international conference on Software engineering, (New York, NY, USA), pp. 587–595, ACM Press, 2005.
9. S. Chulani, B. Boehm, and B. Steece, "Calibrating software cost models using bayesian analysis," IEEE Trans. Software Engr., July-August 1999, pp. 573–583, 1999.
10. B. Clark, S. Devnani-Chulani, and B. Boehm, "Calibrating the cocomo ii post-architecture model," in ICSE '98: Proceedings of the 20th international conference on Software engineering, (Washington, DC, USA), pp. 477–480, IEEE Computer Society, 1998.
11. S. Chulani and B. Boehm, "Modeling software defect introduction and removal: Coqualmo (constructive quality model)," tech. report.
12. S. Devnani-Chulani, "Modeling software defect introduction," tech. report.
13. G. Witting and G. Finnie, "Estimating software developemnt effort with connectionist models," in Proceedings of the Information and Software Technology Conference, pp. 469–476, 1997.
14. K. Peters, "Software project estimation," Methods and Tools, vol. 8, no. 2, 2000.