CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Software Size Measurement

Software size is argued as an important factor contributing to software effort estimation model [6] [7] [8] [9] [10], and the relating technical and environment factors are considered as additional factors [11] [12]. As a result, software size measurement constitutes the basis for effort estimation to be described below.

2.1.1 Lines of Code Measurement

Lines of code (LOC) measurement is an early approach of software sizing metrics. It can be visually and easily applied with the help of a tool to count the number of lines in a program. Nonetheless, a few issues are of concerned, (1) LOC is not available in early development phases until the software project is finished, and (2) it depends on coding technique and programming language, which is difficult to define the counting rule of LOC for different languages [6]. The following LOC metrics are principal software size measurement. Delivered source instructions (DSI) was defined by Boehm [4] to be used as a primary metric for estimating software development cost in COCOMO 81 model. Logical source statements (LSS) was defined by Park [13] and appeared in the Software Engineering Institute (SEI) definition check list of logical source statement counting. Source lines of code (SLOC) was defined by Boehm [6] to be used as a primary metric of estimating software development cost in COCOMO II model. The SLOC mainly obtained counting rule from the LSS.

2.1.2 Functional Size Measurement

There are several methods to measure functional software size. The original method is Function Point Analysis (FPA). Most successive methods are derived from this method. FPA is a standardized methodology for measuring various functionalities of a software application from the user's point of view [14]. It was first published by Albrecht [15] of the IBM Corporation in 1979 and later published in 1983 by Albrecht

and Gaffney [16]. In 1984, GUIDE 84 version was introduced as a guideline for identifying complexity level of function types [17] [18].

One of the advantages of FPA is that it can be created at the early stage of the software development life cycle. Moreover, it is easy to understand than lines of code by the user because it directly comes from user requirements [16]. There have been five functional size measurement methods recognized as ISO standards, namely, IFPUG Functional Size Measurement Method [19], Mk II Function Point Analysis [11], COSMIC Functional Size Measurement Method [20], NESMA Function Point Analysis, and FiSMA Functional Size Measurement Method. Gencel and Demirors [7] reviewed the functional size measurement methods and indicated open issues to improve the methods.

The IFPUG Functional Size Measurement Method is a successive version of Function Point Analysis (FPA). It was formed by International Function Point User Group (IFPUG) which continuously has improved the original FPA method for measuring functional software size. Function points can be computed from five function types (i.e., External Input (EI), External Output (EO), External Inquiry (EQ), Internal Logical File (ILF), and External Interface File (EIF)) and fourteen general system characteristics.

The Mk II Function Point Analysis was introduced by Symons [11] to be the alternative of FPA. It has subsequently been maintained by the United Kingdom Software Metrics Association (UKSMA). The Mk II Function Point Analysis can be applied early in software development phases primarily in business information systems [21]. Provision for Mk II Function Point Analysis application to other software domains can be done by extending the original rules. Function points can be computed from three function types (i.e., Input Data Element Types (Ni), Data Entity Type Referenced (Ne), and Output Data Element Types (No)) and nineteen technical complexity adjustment.

The COSMIC Functional Size Measurement Method designed and maintained by Common Software Measurement International Consortium (COSMIC). The COSMIC method is applied to either business application or real-time software. However, it has not yet been designed to use with mathematically-intensive software [20]. Function points are from four data movement (i.e., Entry (E), Read (R), Write (W), and Exit (X)).

Use Case Point Analysis was developed by Karner [12] in 1993 for software functional size measurement, which was considered as an attractively alternative measurement method because of its simplicity and high level of functionality

measurement [22]. To measure the Use Case Points (UCP), the functionalities of the system are first counted based on Use Case model of requirement analysis. All relating technical and environmental factors are assessed to adjust the value of the UCP, where some technical factors are derived from the original FPA

Object Points analysis was developed by Banker et al. [23] in 1992 to be an output measurement for cost estimation and development productivity in Computer-Aided Software Engineering (CASE) environments based project [23]. "Object Points" is not necessarily related to objects in Object-Oriented Programming. The Object Points analysis is an alternative measurement approach having higher level of functionality measurement. Boehm et al. [6] proposed Application Points procedure to estimate software effort involved in Application Composition and Prototyping software project. This procedure is an emerging extension of COCOMO II. The Application Points procedure was derived from Object Points analysis having two main differences, (1) the rating scales for identifying a productivity rate of the application points/man-month, and (2) Object Points renaming as Application Points to avoid confusion with sizing metrics for conventional object-oriented application. The Application Points can be calculated from object types (i.e., SCREEN, REPORT, and 3GL Component).

2.2 Estimation Techniques

There are many estimation techniques for software effort estimation to be described below.

2.2.1 Artificial Neural Networks

An artificial neural network, generally called a neural network, is a computational model that is inspired by the structure characteristics of biological neural networks. The neural network imitates the brain in two respects, i.e., (1) knowledge is acquired by the network from its environment through a learning process, and (2) interneuron connection strengths are used to store the acquired knowledge [24]. In neural networks, artificial neurons are connected and processed information using a connectionist approach. They are regularly employed to model complex relationships between inputs and outputs.

2.2.2.1 Multilayer Perceptron

A multilayer perceptron (MLP) is a feed forward artificial neural network that is applied from the standard linear perceptron to model relationships between groups of inputs and group of outputs. The MLP composes of three main layers, namely, input, hidden, and output layer where the hidden layer can contain more than one layer. The MLP generally uses back-propagation, which is a supervised learning technique, for training the network. The MLP together with a single hidden layer and sigmoid activation function is recognized as a universal function.

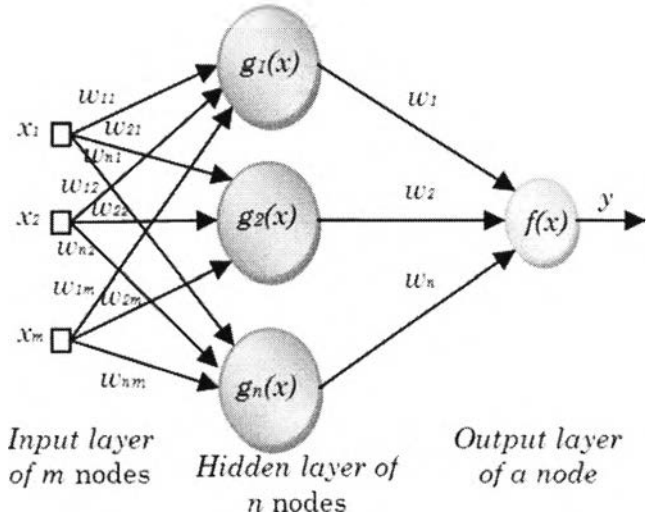

Figure 1: An example of multilayer perceptron architecture.

Figure 1 shows the architecture of a multilayer perceptron with an input layer, a hidden layer, and an output layer, where *m* represents the numbers of neurons in the input layer, *n* denotes the number of neurons in the hidden layer, and *y* is the value of an output neuron derived from the function *f(x)* as shown in Equation (1)

$$f(x) = \phi \sum_{i=1}^{n} w_i g_i(x). \qquad (1)$$

where *f(x)* is represented as the approximation function of input vector *x*, w_i is the weight of the output neuron *i*, and $g_i(x)$ is the output of the hidden neuron *i* as shown in Equation (2).

$$g_i(x) = \phi \sum_{i=1}^{m} w_j(x). \qquad (2)$$

where w_j is the weight vector j of the hidden neuron and x is the input vector x. In the general form, all input nodes are connected to each hidden neuron, and all hidden nodes are also connected to each output node. The activation function can be a linear or nonlinear function. Logistic-Sigmoid function is often used to represent activation function for nonlinear data as shown in Equation (3).

$$\phi(x) = \frac{1}{1+e^{x}}. \tag{3}$$

If a sigmoid function is used as an activation function, inputs and outputs must be scaled to the interval of the function. In this case, Min-max normalization can be used for scaling as shown in Equation (3).

$$\hat{x}_j^{(i)} = \frac{(x_j^{(i)} - min_j\{x_j^{(i)}\})}{(max_j\{x_j^{(i)}\} - min_j\{x_j^{(i)}\})} \times (b-a) + a, \tag{4}$$

where $\hat{x}_j^{(i)}$ and $x_j^{(i)}$ denote the new and original values of the input variable j in sample i, $min(x_j)$ and $max(x_j)$ denote the maximum and minimum values of the original range of input variable j that are derived from the training set, and a and b denote the minimum and maximum values of the new range, respectively.

2.2.2.2 Fuzzy Neural Network

A fuzzy neural network (FNN) [25] encompasses artificial neural network into fuzzy inference system (FIS) to incorporate their capability. It can provide reason of estimating software effort with human-like knowledge representation. For instance, if the analyst's capability is low and software size is high, and then software effort is high. Typically, the artificial neural network is used to extract the relationships between inputs (e.g., analyst's capability and software size) and outputs (i.e., software effort).

FNN builds a fuzzy inference system where its membership function parameters are adjusted by a backpropagation algorithm or combined backpropoagation algorithm and least squares method. The adjustment allows the fuzzy inference system to learn input/output data being modeling.

A network structure of FNN maps inputs through input membership functions and associated parameters, then maps output membership functions and associated parameters to outputs. The parameters associated with the membership functions

change during the learning process. Computations of these parameters are aided by a gradient vector which measures how well the fuzzy inference system is modeling the input/output data for a given set of parameters. When the gradient vector is obtained, backpropagation algorithm or least square method, or both can be applied to adjust the parameters to reduce an error between actual and target outputs.

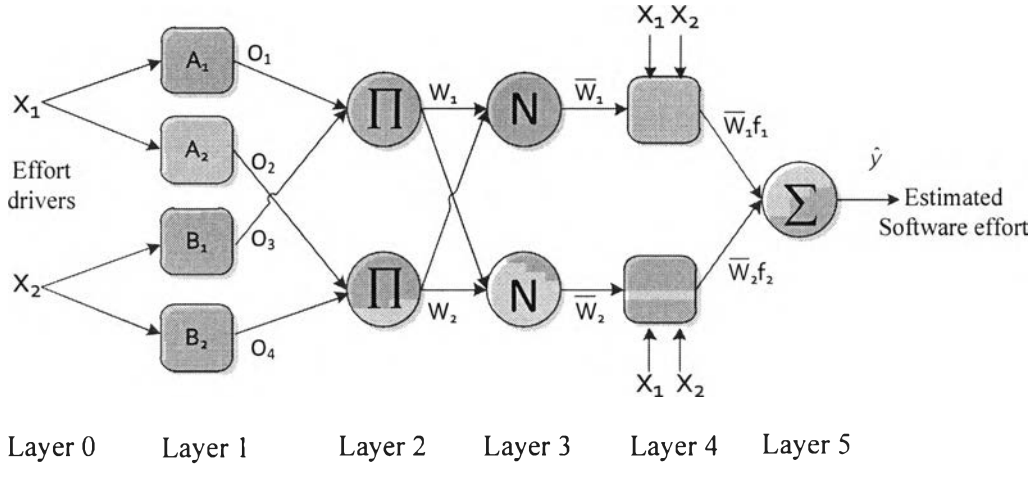

Figure 2: An example of fuzzy neural network diagram.

Figure 2 shows a diagram of fuzzy neural network that contain five layers, namely, (0) input layer, (1) if-part layer, (2) fuzzy rule layer, (3) normalization layer, (4) then-part layer, and (5) output layer.

$$O_i = \mu_{A_i}(x_1) = e^{\frac{-(x_1 - c_i)^2}{2\sigma_i^2}}, \text{ for node } i\text{=1 to 2 at layer 1} \tag{5}$$

where O_i is the output of membership function A_i, σ_i^2 is the variance, and c_i is the center of membership function A_i.

$$O_i = \mu_{B_i}(x_2) = e^{\frac{-(x_2 - c_i)^2}{2\sigma_i^2}}, \text{ for node } i\text{=3 to 4 at layer 1} \tag{6}$$

Similarly, O_i is the output of membership function B_i, σ_i^2 is the variance, and c_i is the center of membership function B_i.

$$w_i = \mu_{A_i}(x_1)\mu_{B_i}(x_2), \text{ for node } i\text{=1 to 2 at layer 2} \tag{7}$$

w_i is the output of layer 2.

$$\overline{w}_i = \frac{w_i}{w_1 + w_2}, \text{ for node } i\text{=1 to 2 at layer 3} \quad (8)$$

$\overline{w}_i$ is the output of layer 3.

$$\overline{w}_i f_i = \overline{w}_i (p_i x_1 + q_i x_2 + b_i) \text{ for } i\text{=1 to 2 at layer 4} \quad (9)$$

$\overline{w}_i f_i$ is the output of layer 4, p_i, q_i, and b_i are parameters derived from learning algorithm. Finally,

$$\hat{y} = \sum \overline{w}_i f_i \quad (10)$$

$\hat{y}$ is the final output.

2.2.2.3 Radial Basis Function Network

A radial basis function (RBF) network is also a feed forward artificial neural network that takes radial basis functions as the activation function. This is a linear sum of radial basis functions. The radial basis function network normally has three layers, i.e., input layer, hidden layer with a non-linear RBF activation function, and linear output layer. Let *m* be the dimension of the input space. The network represents a map from the *m* dimension input space to the single dimensional output space as shown in Equation (11)

$$y = \mathbb{R}^m \rightarrow \mathbb{R}^1 \quad (11)$$

The output neuron can be a function *f(x)* that has the form as shown in Equation (12).

$$f(x) = \sum_{i=1}^{n} w_i \phi(x, c_i). \quad (12)$$

where *f(x)* represents the approximation function, *n* is the number of neurons in the hidden layer, c_i is the center vector for neuron *i*, w_i is the weight of the linear output neuron. In the general form, all inputs are connected to each hidden neuron. The norm utilizes Euclidean distance whilst the basis function is taken to be Gaussian as shown in Equation (13).

$$\phi(x, c_i) = exp[-\frac{1}{2\sigma_i^1} \| x - c_i \|^2]. \quad (13)$$

An input vector *x* is used as input to all basis functions, each of which is accompanied by different parameters. The output of the network is a linear combination of the outputs from radial basis functions.

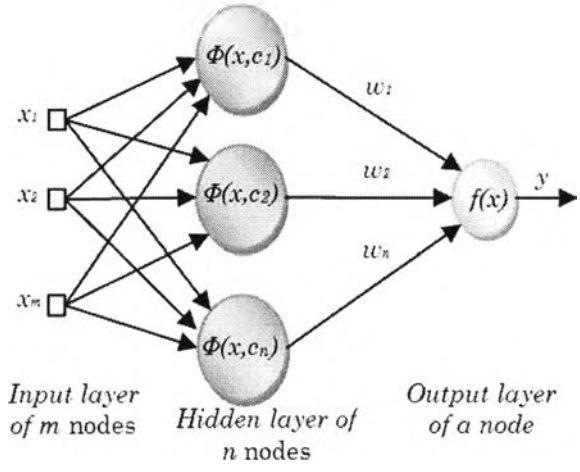

Figure 3: Architecture of a radial basis function network.

2.2.3 Fuzzy Inference System

Fuzzy inference system (FIS) is the process of mapping from an input to an output using fuzzy logic. Fuzzy inference systems have been successfully applied in fields such as automatic control, data classification, decision analysis, expert systems, and computer vision. Fuzzy inference process is composed of (1) fuzzify inputs, (2) apply operator (AND or OR), (3) apply implementation method, (4) apply aggregation method, and (5) defuzzify fuzzy output to a crisp number.

2.2.4 Support Vector Regression

Support vector regression (SVR) applies the principle of the support vector machine (SVM) to solve a regression task.

The principle of SVM is based on supervised learning methods for classification. The principle lies in decision planes that separate a set of samples having different class memberships as shown in Figure 4 .

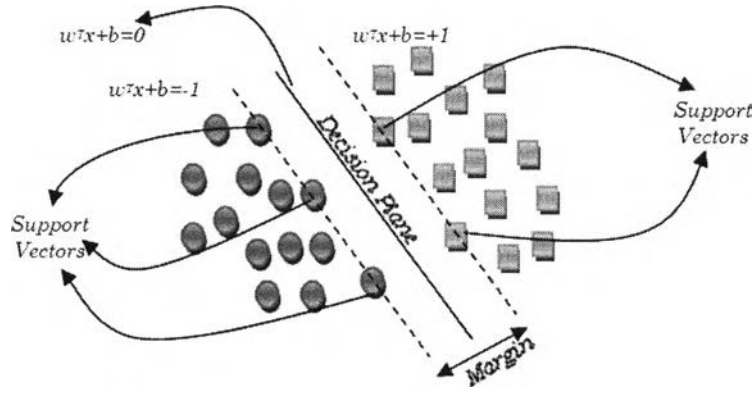

Figure 4: An example of linearly separable patterns.

In Figure 4, samples can be separated into two groups by a line that represents the decision plane. However, this is not always the case in some situations where samples are not distinctive by separated into groups. To solve this problem, a transformation process must be performed to resolve such a problem as shown in Figure 5.

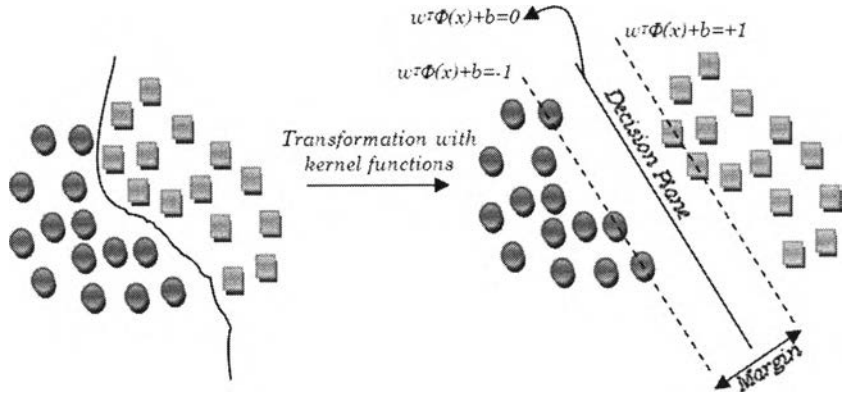

Figure 5: Transformation of nonlinearly to linearly separable patterns.

In Figure 5, the original samples appeared on the left side of in the figure are transformed using kernel functions, which can be a polynomial learning machine, radial basis function, or hyperbolic tangent. The transformed samples on the right are linearly separable by the kernel functions. The samples on margin are called support vectors. The equation for this decision planes is shown in Equation (14).

$$w^T \phi(x) + b = 0. \tag{14}$$

where w^T denotes transpose of weight, x is the input vector, and b is the bias.

$$w = \sum_{i=1}^{n} a_i d_i \phi(x_i). \tag{15}$$

where a_i is Lagrange Multiplier of vector i, d is class of vector i (+1/-1) and x_i is the input vector i.

$$f(x) = \sum_{i=1}^{n} a_i d_i k(x, x_i) \tag{16}$$

where $f(x)$ is decision function and $k(x, x_i)$ represents the kernel function.

The concept of SVR is to minimize the generalization error bound for achieving generalized performance. The idea is based on computation of a linear regression function in a high dimensional feature space, where the input data are mapped via a nonlinear function.

2.2.5 Regression Analysis

Ordinary Least Square (OLS) regression is a classical statistic method for approximating the desired values in a linear regression model. This method minimizes the sum of squared residual errors of the actual software effort and estimated software effort given by the linear approximation. The OLS regression is a generally recognized method and is applied in many research fields such as economics, electrical engineering, and software engineering. A general form of the OLS equation is shown in Equation (17).

$$\hat{y} = \sum_{i=1}^{m} B_i X_{ij} + B_0. \tag{17}$$

where $j=\{1,2,...,n\}$ is a set of software projects, $i=\{1,2,...,m\}$ is a set of effort drivers, B_0 is a constant, B_i is the coefficient of effort driver i, $\hat{y}_j$ is the estimated effort in project j, and x_{ij} is the value of the effort driver i in project j.

OLS is carried out under an assumption that the data are normally distributed to provide favorable estimation results. Robust regression analysis is designed to be not affected by the assumption. Generally, it is not sensitive to outliers.

2.2.6 Classification and Regression Tree

Classification and regression tree is one of decision tree learning. It builds a decision tree for predicting the software effort. To predict the effort, the process starts with root node and traverses down to leaf nodes, where the leaf nodes hold the effort. Figure 6 illustrates an example of a decision tree for predicting software effort. Let EI, EO, and *ILF* be effort/cost drivers in the tree. If *EI* is more than or equal to 50 and *EO* more than 35 or equal to, then the software effort is 110.

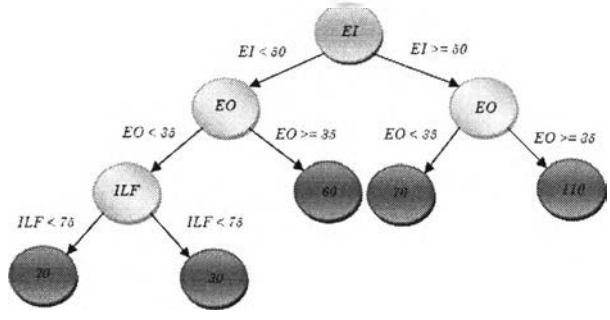

Figure 6: An example of a decision tree.

2.2.7 Analogy

Analogy-based estimation is a form of Case-Based Reasoning (CBR) which predicts software effort like expert judgment. As a result, experts try to estimate the effort from their experience in effort estimation while the analogy derives the effort from historically completed projects. *K*-nearest neighbor (*KNN*) algorithm with Euclidian distance often used for the analogy-based estimation. *KNN* can be computed by Equation (18).

$$\hat{y}^{(i)} = \sum_{l=1}^{k} d^{(l)} \times y^{(l)} \tag{18}$$

where $\hat{y}^{(i)}, 1 \leq l \leq k$ is the predicted software effort of project i and k is the number of closest projects (note that k can be derived from k-fold cross-validation, $y^{(l)}$ is a known value of the software effort of the closest project l, and $d^{(l)}$ is the weight of project l. The weight can be computed from Equation (19).

$$d^{(l)} = \frac{sim(X_i, X_l)}{\sum_{j=1}^{k} sim(X_i, X_j)} \tag{19}$$

where $sim(X_i, X_l)$ is the similarity function used to measure the similarity between the projects X_i and X_l as shown in Equation (20).

$$sim(X_i, X_l) = \frac{1}{dist(X_i, X_l)} \tag{20}$$

where $dist(X_i, X_l)$ is the Euclidian or Mahalanobis distance applied to measuring the distance between the project the projects X_i and X_l as shown in Equation (21) and (22), respectively.

$$dist(X_i, X_l) = \sqrt{\sum_{j=1}^{M} (x_j^{(i)} - x_j^{(l)})^2} \quad (21)$$

where $X_i = \{x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, ..., x_M^{(i)}\}$ and $X_l = \{x_1^{(l)}, x_2^{(l)}, x_3^{(l)}, ..., x_M^{(l)}\}$, and M is the total number of features.

$$dist(X_i, X_l) = \sqrt{\sum_{j=1}^{M} (\frac{x_j^{(i)} - x_j^{(l)}}{SD_j})^2} \quad (22)$$

where SD_j is the standard deviation of feature j.

2.3 Phase Effort Distribution

In a software development project, software effort is distributed over development phases of a software development process model. There have been research works studied about phase effort distribution across software development life cycle such as [15] [6] [26] [27].

Albrecht [15] reported mean phase effort distribution percentage, where requirements phase took about 7.9% of the work hours. The numbers for system design, program development, system test and demonstration, and user documentation were 12.1%, 62.4%, 11.6%, and 6.0%, respectively. The distribution percentages were derived from software development projects of IBM Data Processing Services. The projects covered many industries over 3-4 years coded in three programming languages, namely PL/1, COBOL, and DMS/VS.

Boehm et al. [6] provided the percentages of 7%, 17%, 25%, 33%, 25%, and 12% for planning and requirements, product design, detailed design, coding and unit test, integration and testing, and transition, respectively. However, these percentages omit two added provisions, namely, the missing 7% front-end and 12% back-end effort that must be added to the total estimation based on his COCOMO model. For example, if the phase distribution percentage is 100 person-months, the total estimated effort would be 119 person-months. In other words, the core development by COCOMO effort estimation begins at product design phase and ends at the completion of the integration and testing phase.

Yang et al. [26] reported the percentages of 16.14%, 14.88%, 40.36%, 21.57%, and 7.06% for planning and requirements, design, coding, test, and transition, respectively. These percentages were derived from 75 projects in China Software Benchmarking Standard Groups (CSBSG) data repository.

Kultur et. al [27] reported the percentage was about 25%, 2%, 47%, 22%, and 4 for planning and analysis, design, coding, testing, and transaction, respectively. This percentage was derived from 395 projects in International Software Benchmarking Standard Groups (ISBSG) data repository.

2.4 Estimation Techniques

The early techniques for software effort estimation were typically based on regression analysis. Putnam's Software LIfecycle Management (SLIM) [28] and Boehm's COnstructive COst MOdel (COCOMO) [4] [29] [30] [31] [6] are early well-known software effort estimation models. These models require lines of code as the primary input factor. However, lines of code exhibit high dependency on programming language used. To overcome this issue, Albrecht and Gaffney [16] proposed an effort estimation model using function points as the primary factor. Recently, other prediction techniques have been applied for effort estimation such as decision tree, neural networks, support vector machine, case-based reasoning, and fuzzy logic. Nevertheless, lines of code and function points are still used as the principal metrics.

Table 1: An overview of effort estimation research work.

Year (Author)	Software Size		Pre-Processing		Estimation Technique							Validation			
	LOC	FP	MV	OD	FS	RA	DT	NN	SVR	CBR	FL	HOL	REP	KFL	LOO
(2006) Oliveia [32]	■					RA		RBF	SVR						■
(2007) Chiu and Huang [33]		■		■		OLS	CART	NN		CBR				3	
(2008) Huang et al. [34]		■	■	■	■	OLS								3	
(2008) Barcelos tronto et. al. [35]	■			■	■	RA		MLP						6	
(2008) Keung et al. [36]		■		■	■					CBR					■
(2008) Kumar et al. [37]		■				RA		WNN	SVR			80:20		3	
(2008) Liu et al. [38]		■	■	■	■	RR	CART			KNN				10	
(2009) Huang and Chiu [39]	■							FNN						3	
(2009) Elish [40]	■		■		■		MART								■
(2010) Mittal et al. [41]	■										FL				
(2012) Kocaguneli et al [42].	■	■		■	■	RA		NN		CBR				3	■
(2012) Kocaguneli et al. [43]	■	■	■		■	RA	CART	NN		CBR					■
(2012) Dejaeger et al. [44]	■	■	■	■	■	RA	DT	MLP		CBR			20		■

Table 1 provides an overview of techniques that have been used in various research publications for overall effort estimation. Typically, the techniques are divided into five groups, namely, software size, data pre-processing, estimation technique, performance evaluation, and validation.

The abbreviations are as follows: LOC = Lines of Code, FP = Function Points, MV = Missing Value Handling, OD = Outlier Detection and Handling, FS = Feature Selection, RA = Regression Analysis, DT = Decision Tree, NN = Neural Network, SVM = Support Vector Machine, CBR = Case-Base Reasoning, KNN = *k*-Nearest Neighbor, FL = Fuzzy Logic, OLS = Ordinary Least Square, RR = Robust Regression, CART = Classification and Regression Tree, MART = Multiple Additive Regression Tree, FNN = Fuzzy Neural Networks, MLP = Multilayer Perceptron, WNN = Wavelet Neural Networks, SVR = Support Vector Regression, HOL = Holdout Validation, REP = Repeated Random Sub-sampling Cross-validation, KFOL = *k*-fold Cross-validation, and LOO = Leave-One-Out Cross-Validation. A gray cell refers to a technique or a method being used. Abbreviations in the estimation technique section denote cited techniques. Numbers in column HOL, REP, KFL represent proportion of training and test data, number of repeated random sub-sampling, and a number of folds, respectively.

Three estimation techniques are commonly used. The first technique is to use known proportion of phase effort to overall effort for predicting the phase effort. It is a traditional way to get a rough prediction. For example, suppose that the proportion of coding effort is equal to 40%, and the estimated overall effort is equal to 200 man-month, then the estimated coding phase effort is equal to 80 man-month.

The second technique is to create a phase effort estimation model using prior phase effort as an input variable. Azzeh et al. [3] proposed integrating of fuzzy set and association rule-based estimation model. MacDonell and Shepperd [45] built a regression analysis-based estimation model, where system environment and primary programming language features were used as additional inputs. Abrahamsson et al. [46] established an estimation model for iteration effort estimation in agile development to replace phase effort estimation. Wang et al. [47] set up an estimation model for stage (month) effort estimation instead of phase effort estimation. This model was later improved by integrating of Verhulst method and grey system [1].

The last technique is to create phase effort estimation model using software size as an input variable. Kultur et al. [27] built regression analysis-based effort estimation using function point count as an input for each industrial sector.

In this study, not only software size but also other features are analyzed to derive an estimation model, where neural network and other estimation techniques are used to create the proposed estimation models.