Software Quality Journal, 11, 87–119, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Machine Learning and Software Engineering

DU ZHANG zhangd@ecs.csus.edu
Department of Computer Science, California State University, Sacramento, CA 95819-6021

JEFFREY J.P. TSAI tsai@cs.uic.edu
Department of Computer Science, University of Illinois, Chicago, IL 60607

Abstract. Machine learning deals with the issue of how to build programs that improve their performance at some task through experience. Machine learning algorithms have proven to be of great practical value in a variety of application domains. They are particularly useful for (a) poorly understood problem domains where little knowledge exists for the humans to develop effective algorithms; (b) domains where there are large databases containing valuable implicit regularities to be discovered; or (c) domains where programs must adapt to changing conditions. Not surprisingly, the field of software engineering turns out to be a fertile ground where many software development and maintenance tasks could be formulated as learning problems and approached in terms of learning algorithms. This paper deals with the subject of applying machine learning in software engineering. In the paper, we first provide the characteristics and applicability of some frequently utilized machine learning algorithms. We then summarize and analyze the existing work and discuss some general issues in this niche area. Finally we offer some guidelines on applying machine learning methods to software engineering tasks and use some software development and maintenance tasks as examples to show how they can be formulated as learning problems and approached in terms of learning algorithms.

Keywords: machine learning, software engineering, learning algorithms

1. The challenge

The challenge of developing and maintaining large software systems in a changing environment has been eloquently spelled out in Brooks' classical paper, *No Silver Bullet* (Brooks, 1987). The following *essential difficulties* inherent in developing large software still hold true today:

- *Complexity*: "Software entities are more complex for their size than perhaps any other human construct." "Many of the classical problems of developing software products derive from this essential complexity and its nonlinear increases with size."
- *Conformity*: Software must conform to the many different human institutions and systems it comes to interface with.
- *Changeability*: "The software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product."
- *Invisibility*: "The reality of software is not inherently embedded in space." "As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another." (Brooks, 1987).

However, in his "*No Silver Bullet*" *Refired* paper (Brooks, 1995), Brooks uses the following quote from Glass to summarize his view in 1995:

> So what, in retrospect, have Parnas and Brooks said to us? That software development is a conceptually tough business. That magic solutions are not just around the corner. That it is time for the practitioner to examine evolutionary improvements rather than to wait-or hope-for revolutionary ones (Glass, 1988).

Many evolutionary or incremental improvements have been made or proposed, with each attempting to address certain aspect of the essential difficulties (Boehm, 2000; Ernst et al., 2001; Green et al., 1986; Lowry, 1992; Parnas, 1979). For instance, to address changeability and conformity, an approach called the *transformational programming* allows software to be developed, modified, and maintained at specification level, and then automatically transformed into production-quality software through automatic program synthesis (Green et al., 1986). This software development paradigm will enable software engineering to become the discipline of capturing and automating currently undocumented domain and design knowledge (Lowry, 1992). Software engineers will deliver knowledge-based application generators rather than unmodifiable application programs. A system called LaSSIE was developed to address the complexity and invisibility issues (Devanbu et al., 1991). The multi-view modeling framework proposed in (Broy, 2001) could be considered as an attempt to address the invisibility issue.

The application of artificial intelligence techniques to software engineering (AI&SE) has produced some encouraging results (Binder and Tsai, 1992; Liu and Tsai, 1996; Lowry, 1992; Mostow, 1985; Partridge, 1998; Rich and Waters, 1986; Tsai and Weigert, 1993; Tsai et al., 1998; Welty and Selfridge, 1995). Some of the successful AI techniques include: knowledge-based approach, automated reasoning, expert systems, heuristic search strategies, temporal logic, planning, and pattern recognition. To ultimately overcome the essential difficulties, AI techniques can play an important role. As a subfield of AI, machine learning (ML) deals with the issue of how to build computer programs that improve their performance at some task through experience (Mitchell, 1997). It is dedicated to creating and compiling verifiable knowledge related to the design and construction of artifacts (Provost and Kohavi, 1998). ML methods have found their way into the software development in the past twenty years. ML algorithms offer a viable alternative to the existing approaches to many SE issues. In his keynote speech at the 1992 annual conference of the American Association for Artificial Intelligence, Selfridge advocated the application of ML to SE (ML&SE):

> We all know that software is more updating, revising, and modifying than rigid design. Software systems must be built for change; our dream of a perfect, consistent, provably correct set of specifications will always be a nightmare-and impossible too. We must therefore begin to describe change, to write software so that (1) changes are easy to make, (2) their effects are easy to measure and compare, and (3) the local changes contribute to overall improvements in the software.
>
> For systems of the future, we need to think in terms of shifting the burden of evolution from the programmers to the systems themselves ... [we need to]

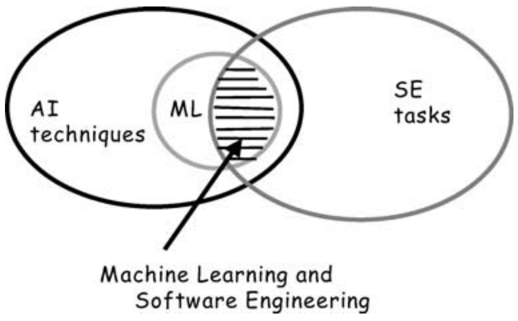

Figure 1. Scope of this paper.

explore what it might mean to build systems that can take some responsibility for their own evolution (Selfridge, 1993).

Though many results in ML&SE have been published in the past two decades, effort to summarize the state-of-the-practice and to discuss issues in applying ML to SE has been few and far between (Zhang, 2000; Zhang and Tsai, 2002). A recent paper (Menzies, 2001) focuses its attention on decision tree based learning methods to SE issues. Another survey is offered from the perspective of data mining techniques being applied to software process and products (Mendonca and Sunderhaft, 1999). The AI&SE summaries published so far are too broad a brush that they do not give an adequate account on ML&SE. The scope of this paper, as depicted in the shaded area in Figure 1, is to attempt to fill this void by studying various issues pertaining to ML&SE (the applications of other AI techniques in SE are beyond the scope of this paper). We think this is an important and helpful step if we want to make any headway in ML&SE.

In this paper, we address various issues in ML&SE by trying to answer the following questions:

- What types of learning methods are there available at our disposal?
- What are the characteristics and underpinnings of different learning algorithms?
- How do we determine which learning method is appropriate for what type of software development or maintenance task?
- Which learning methods can be used to make headway in what aspect of the essential difficulties in software development?
- When we attempt to use some learning method to help with an SE task, what are the general guidelines and how can we avoid some pitfalls?
- What is the state-of-the-practice in ML&SE?
- Where is further effort needed to produce fruitful results?

The rest of the paper is organized as follows. Section 2 provides an overview of the issues and dimensions in ML and a summary of some frequently used learning algorithms. Section 3 offers a classification of the existing work, analyzes the state-of-the-practice in ML&SE, and points out where further effort is needed. Some general guidelines for selecting and applying ML methods to SE tasks are described in Section 4, along with discussions on the formulations of some SE tasks in terms of the learning methods. Finally in Section 5, we conclude the paper with remarks on future work.

2. Overview of machine learning

ML algorithms have been utilized in many different problem domains. Some typical applications are: data mining problems where large databases contain valuable implicit regularities that can be discovered automatically, poorly understood domains where there is lack of knowledge needed to develop effective algorithms, or domains where programs must dynamically adapt to changing conditions (Mitchell, 1997). The following list of publications and web sites offers a good starting point for the interested reader to be acquainted with the state-of-the-practice in ML applications (Aha; Bergadano and Gunetti, 1995; Bratko and Muggleton, 1995; Cristianini and Shawe-Taylor, 2000; Dietterich, 1997; Langley and Simon, 1995; Mendonca and Sunderhaft, 1999; Menzies, 2001; Michalski et al., 1998; Mitchell, 1997a, b, 1999; Quinlan, 1990; Saitta and Neri, 1998; Sutton and Barto, 1999).

ML is not a panacea for all the SE problems. To better use ML methods as tools to solve real world SE problems, we need to have a clear understanding of both the problems, and the tools and methodologies utilized. It is imperative that we know (1) the available ML methods at our disposal, (2) characteristics of those methods, (3) circumstances under which the methods can be most effectively applied, and (4) their theoretical underpinnings. Since solutions to a given problem can often be expressed (or approximated) as a target function (or functions), the problem solving process (or the learning process) boils down to how to find such a function (or functions) that can best describe the known and unknown cases or phenomena for a given problem domain. Learning a *target function* (or a set of possible target functions) from training data involves the following issues:

- Representation. Different learning methods may adopt different representation formalisms for the data and knowledge (functions) to be learned. In some learning method, the target function is not explicitly defined.
- Characteristics of learning process. Learning can be *supervised* or *unsupervised*. Different methods may have different *inductive bias*, different search strategies, different guiding factors in search, and different needs regarding the availability of a domain theory. For a target function, its generalization can be eager (at learning stage) or lazy (at classification stage), and its approximation can be obtained either locally or globally with regard to a set of training cases. Learning can result in either knowledge augmentation or knowledge (re)compilation. Depending on the interaction between a learner and its environment, there are *query learning* and *reinforcement learning*.
- Properties of training data and domain theories. Data gathered for the learning process can be small or large in quantity, and noisy or accurate in terms of random errors. Data can have different valuations. Different learning methods can have different criteria regarding training data, with some methods requiring large amount of data, others being very sensitive to the quality of data, and yet another needing both training data and a domain theory. On the other hand, the quality of domain theories (correctness, completeness) will have direct impact on the outcome of *analytical* learning methods. Finally, based on the way in which training data are generated and provided to a learner, there are *batch learning* and *on-line learning*.

- Target function output. Depending on the output, learning problems can be categorized as *binary classification*, *multi-value classification* and *regression*.
- Theoretical underpinnings and practical considerations. Underpinning learning methods are different justifications: statistical, probabilistic, or logical. What are the frameworks for analyzing learning algorithms? How can we evaluate the performance of a generated function, and determine the *convergence* issue? What types of practical problems do we have to come to grips with?

Figure 2 summarizes the aforementioned issues in the learning process.

There are many different types of learning methods, each having its own characteristics and lending itself to certain learning problems. In this paper, we adopt the classification of Mitchell (1997a) and organize major types of learning methods into the following groups: concept learning (CL), decision tree (DT) learning, artificial neural networks (NN), Bayesian learning (BL), reinforcement learning (RL), genetic algorithms (GA) and genetic programming (GP), instance-based learning (IBL, of which case-based reasoning, or CBR, is a popular method), inductive logic programming (ILP), analytical learning (AL, of which explanation-based learning, or EBL is a method), and combined inductive and analytical learning (IAL). Table 1 describes the main properties of the aforementioned types of learning.

3. State-of-the-practice in ML&SE

Several areas in software development have already witnessed the use of machine learning algorithms. In this section, we first take a look at reported results and offer a classification of the existing work, then analyze the state-of-the-practice in this niche area, and finally discuss some general issues in ML&SE. The list of publications included in our classification and analysis, though not a complete one, should serve to represent a balanced view of the current status. The trend indicates that people have realized the potential of ML techniques and begin to reap the benefits from applying them in software development and maintenance.

3.1. Classification of existing work

In the classification below, we use the activity types as the guideline to group applications of ML methods in SE tasks. Table 2 summarizes seven types of activities, each of which contains applications in a number of different SE tasks. In this subsection, we provide the gist for each application under each group.

3.1.1. Prediction and estimation. In this group, ML methods are used to predict or estimate: (1) software quality, (2) software size, (3) software development cost, (4) project or software effort, (5) maintenance task effort, (6) software resource, (7) correction cost, (8) software reliability, (9) software defect, (10) reusability, (11) software release timing, and (12) testability of program modules.

Software quality prediction. GP is used in (Evett et al., 1998) to generate software quality models that take as input software metrics collected earlier in development,

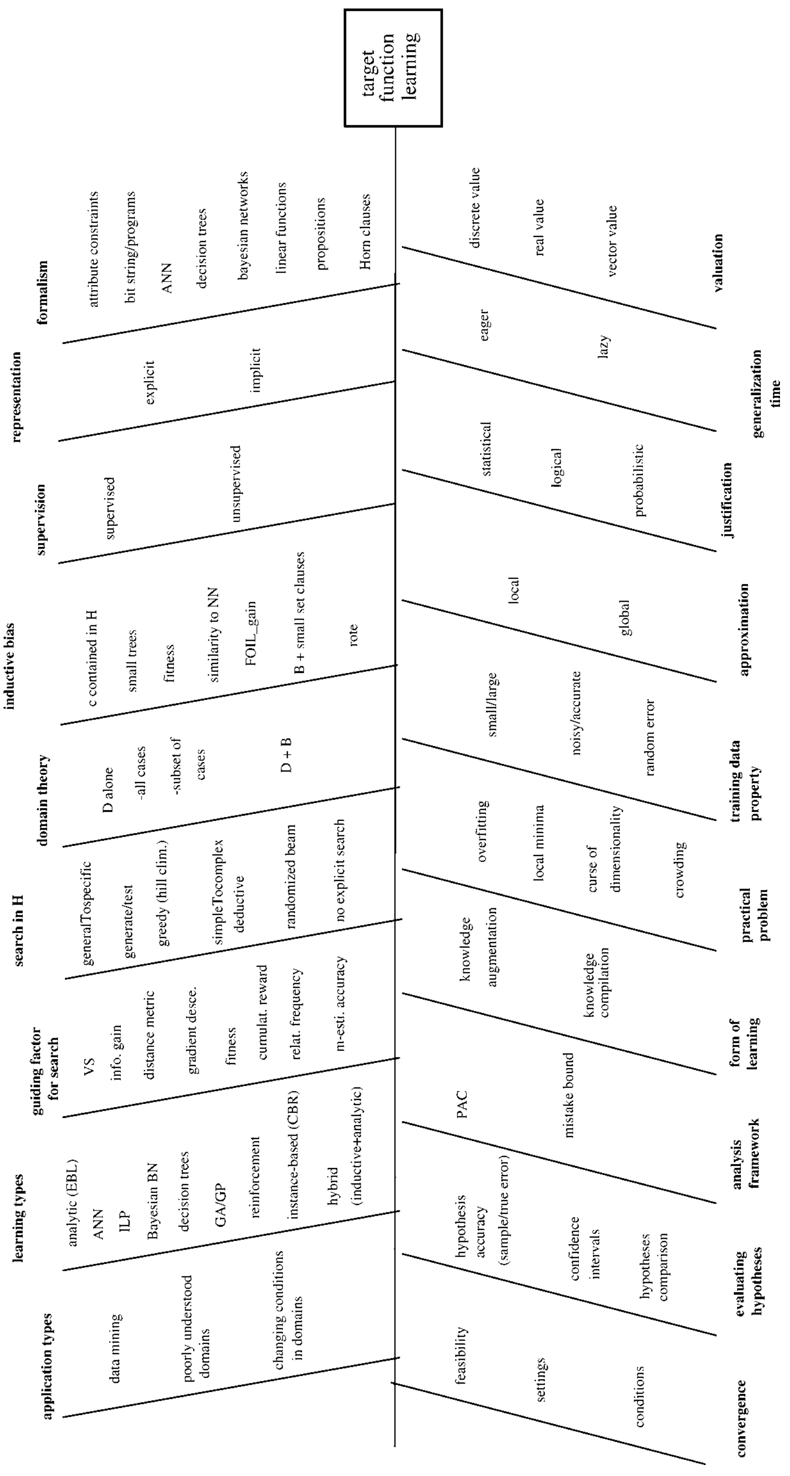

Figure 2. Issues in learning process.

Table 1. Major types of learning methods

Type	Target function representation	Target function generation[1]	Search	Inductive bias	Algorithm[2]
AL (EBL)	Horn clauses	Eager, D + B, supervised,	Deductive reasoning	B + small set of Horn clauses	Prolog-EBG
BL	Bayesian network	Eager, supervised, D (global), explicit or implicit	Probabilistic, no explicit search	Minimum description length	MAP, BOC, Gibbs, NBC
CL	Conjunction of attribute constraints	Eager, supervised, D (global)	Version Space (VS) guided	$c \in H$	Candidate elimination
DT	Decision trees	Eager, D (global), supervised	Information gain (entropy)	Preference for small trees	ID3, C4.5, Assistant
GA GP	Bit strings, program trees	Eager, no D, unsupervised	Hill climbing (simulated evolution)	Fitness-driven	Prototypical GA/GP algorithms
IBL	Not explicitly defined	Lazy, D (local), supervised,	Statistical reasoning	Similarity to nearst neighbors	K-NN, LWR, CBR
ILP	If-then rules	Eager, supervised, D (global),	Statistical, general-to-specific	Rule accuracy, FOIL-gain, shorter clauses	SCA, FOIL, PROGOL, inv. resolution
NN	Artificial neural networks	Eager, supervised, D (global)	Gradient descent guided	Smooth interpolation between data points	Back-propagation
IAL	Determined by learning approaches used	Eager, D + B, supervised	Determined by learning approaches used	Determined by learning approaches used	KBANN, EBNN, FOCL
RL	Control strategy π^*	Eager, no D, unsupervised	Through training episodes	Actions with max. Q value	Q, TD

[1] The sets D and B refer to training data and domain theory, respectively.
[2] The algorithms listed are only representatives from different types of learning.

and predict for each module the number of faults that will be discovered later in development or during operations. These predictions will then be the basis for ranking modules, thus enabling a manager to select as many modules from the top of the list as resources allow for reliability enhancement.

A comparative study is done in (Lanubile and Visaggio, 1997) to evaluate several modeling techniques for predicting quality of software components. Among them is the NN model. Another NN based software quality prediction work, as reported in (Hong and Wu, 1997), is language specific, where design metrics for SDL (Specification and Description Language) are first defined, and then used in building the pre-

Table 2. Summary of existing ML applications in SE tasks

Activity type	SE task	ML method[1]
Prediction	Software quality (high-risk, or fault-prone component identification)	GP (Evett et al., 1998), NN (Hong and Wu, 1997; Khoshgoftaar et al., 1995, 1997; Lanubile and G. Visaggio, 1997), CBR (El Emam et al., 2001; Ganesan et al., 2000), DT (Briand et al., 1993; Porter and Silbey, 1990), CL (de Almeida and Matwin, 1999), ILP (Cohen and Devanbu, 1997)
	Software size	{NN, GP} (Dolado, 2000)
	Software development cost	DT (Briand et al., 1992), CBR (Briand et al., 1999), BL (Chulani et al., 1999)
	Project/software (development) effort	CBR (Shepperd and Schofield, 1997; Vicinanza et al., 1990), {DT, NN} (Srinivasan and Fisher, 1995), GA + NN (Shukla, 2000), {NN, CBR} (Finnie et al., 1997)
	Maintenance task effort	{NN, DT} (Jorgensen, 1995)
	Software resource analysis	DT (Selby and Porter, 1988)
	Correction cost	{DT, ILP} (de Almeida et al., 1998)
	Software reliability	NN (Karunanithi et al., 1992)
	Defects	BL (Fenton and Neil, 1999)
	Reusability	DT (Mao et al., 1998)
	Software release timing	NN (Dohi et al., 1999)
	Testability of program modules	NN (Khoshgoftaar et al., 2000)
Property/model discovery	Program invariants	ILP (Bratko and Grobelnik, 1993)
	Identifying objects in programs	NN (Abd-El-Hafiz, 2000)
	Process models	NN (Cook and Wolf, 1998), EBL (Garg and Bhansali, 1992)
Transformation	Transform serial programs to parallel ones	GP (Ryan and Ivan, 1999; Ryan, 2000)
	Improve software modularity	CBR + NN (Schwanke and Hanson, 1994)
	Mapping OO applications to heterogeneous distributed environments	GA (Choi and Wu, 1998)

Continued

diction models for identifying fault prone components. In (Khoshgoftaar et al., 1995, 1997), NN based models are used to predict faults and software quality measures.

CBR is the learning method used in two separate software quality prediction efforts (Emam et al., 2001; Ganesan et al., 2000). The focus of Emam et al. (2001) is on comparing the performance of different CBR classifiers, resulting in a recommendation of a simple CBR classifier with Euclidean distance, z-score standardization, no weighting scheme, and selecting the single nearest neighbor for prediction. In (Ganesan et al., 2000), CBR is applied to software quality modeling of a family of full-scale industrial software systems and the accuracy is considered better than a corresponding multiple linear regression model in predicting the number of design faults.

Table 2. Continued

Activity type	SE task	ML method
Generation and synthesis	Test cases/data	ILP (Bergadano and Gunetti, 1996), GA (Michael and McGraw, 1998; Michael et al., 2001)
	Software agents	GP (Qureshi, 1996)
	Design repair knowledge	CBR + EBL (Bailin et al., 1991)
	Design Schemas	IBL (Harandi and Lee, 1991)
	Data structures	GP (Langdon, 1995)
	Program/scripts	IBL (Bansali and Harandi, 1993), {CL, AL} (Minton and Wolfe, 1994)
	Project management schedule	GA (Chang et al., 2001)
Reuse library construction and maintenance	Similarity computing	CBR (Ostertag et al., 1992)
	Active browsing	IBL (Drummond et al., 2000)
	Cost of rework	DT (Basili et al., 1997)
	Knowledge representation	CBR (Fouque and Matwin, 1992)
	Locate and adopt software to specifications	CBR (Katalagarianos and Vassiliou, 1995)
	Generalizing program abstractions	EBL (Hill, 1987)
	Clustering of components	GA (Lee et al., 1998)
Acquisition or recovery of specifications	Derivation of specifications of system goals and requirements	CL (Van Lamsweerde and Willemet, 1998)
	Extract specifications from software	ILP (Cohen, 1995)
	Acquire knowledge for specification refinement and augmentation	{DT, NN} (Partridge et al., 2001)
	Acquire and maintain specification consistent with scenarios	EBL (Hall, 1995; 1998)
Development knowledge management	Collect and manage software development knowledge	CBR (Henninger, 1997)
	Capture and reuse design knowledge	CBR (Leake and Wilson, 2001)

[1] An explanation on the notations: {...} is used to indicate that multiple ML methods are each independently applied for the same SE task, and "$\cdots + \cdots$" is used to indicate that multiple ML methods are collectively applied to an SE task.

In (Porter and Selby, 1990), a DT based approach is used to generate measurement-based models of high-risk components. The proposed method relies on historical data (metrics from previous releases or projects) for identifying components of fault prone properties. Another DT based approach is used to build models for predicting high-risk Ada components (Briand et al., 1993).

Another comparative study result is reported in (Cohen and Devanbu, 1997) on using ILP methods for software fault prediction for C++ programs. Both natural and artificial data are used in evaluating the performance of two ILP methods and some extensions are proposed to one of them.

Software quality prediction is formulated as a CL problem in (de Almeida and Matwin, 1999). It is noted in the study that there are activities (such as data acquisition, feature extraction and example labeling) prior to the actual learning process. These activities would have impact on the quality of the outcome. The proposed approach is applied to a set of COBOL programs.

Software size estimation. NN and GP are used in (Dolado, 2000) to validate the component-based method for software size estimation. In addition to producing results that corroborate the component-based approach for software sizing, it is noticed in the study that NN works well with the data, recognizing some nonlinear relationships that the multiple linear regression method fails to detect. The equations evolved by GP provide similar or better values than those produced by the regression equations, and are intelligible, providing confidence in the results.

Software cost prediction. A general approach, called optimized set reduction and based on DT, is described in (Briand et al., 1992) for analyzing software engineering data, and is demonstrated to be an effective technique for software cost estimation. A comparative study is done in (Briand et al., 1999) which includes a CBR technique for software cost prediction. The result reported in (Chulani et al., 1999) indicates that the improved predictive performance of software cost models can be obtained through the use of Bayesian analysis, which offers a framework where both prior expert knowledge and sample data can be accommodated to obtain predictions.

Software (project) development effort prediction. IBL techniques are used in (Shepperd and Schofield, 1997) for predicting the software project effort for new projects. The empirical results obtained (from nine different industrial data sets totaling 275 projects) indicate that CBR offers a viable complement to the existing prediction and estimations techniques. Another CBR application in software effort estimation is reported in (Vicinanza et al., 1990).

DT and NN are used in (Srinivasan and Fisher, 1995) to help predict software development effort. The results were competitive with conventional methods such as COCOMO and function points. The main advantage of DT and NN based estimation systems is that they are adaptable and nonparametric.

Additional research on ML based software effort prediction includes: a genetically trained NN (GA + NN) predictor (Shukla, 2000), a comparative study of software effort estimation techniques in (Finnie et al., 1997) that are based on NN and CBR.

Maintenance task effort prediction. Models are generated in terms of NN and DT methods, and regression methods, for software maintenance task effort prediction in (Jorgensen, 1995). The study measures and compares the prediction accuracy for each model, and concludes that DT-based and multiple regression-based models have better accuracy results. It is recommended that prediction models be used as instruments to support the expert estimates and to analyze the impact of the maintenance variables on the process and product of maintenance.

Software resource analysis. In (Selby and Porter, 1988), DT is utilized in software resource data analysis to identify classes of software modules that have high development effort or faults (the concept of "high" is defined with regard to the uppermost quartile relative to past data). Sixteen software systems are used in the study. The decision trees correctly identify 79.3 percent of the software modules that had high development effort or faults.

Correction cost estimation. An empirical study is done in (de Almeida et al., 1998) where DT and ILP are used to generate models for estimating correction costs in software maintenance. The generated models prove to be valuable in helping to optimize resource allocations in corrective maintenance activities, and to make decisions regarding when to restructure or reengineer a component so as to make it more maintainable. A comparison leads to an observation that ILP-based results perform better than DT-based results.

Software reliability prediction. Software reliability growth models can be used to characterize how software reliability varies with time and other factors. The models offer mechanisms for estimating current reliability measures and for predicting their future values. The work in (Karunanithi et al., 1992) reports the use of NN for software reliability growth prediction. An empirical comparison is conducted between NN-based models and five well-known software reliability growth models using actual data sets from a number of different software projects. The results indicate that NN-based models adapt well across different data sets and have a better prediction accuracy.

Defect prediction. BL is used in (Fenton and Neil, 1999) to predict software defects. Though the system reported is only a prototype, it shows the potential Bayesian belief networks (BBN) has in incorporating multiple perspectives on defect prediction into a single, unified model. Variables in the prototype BBN system (Fenton and Neil, 1999) are chosen to represent the life-cycle processes of specification, design and implementation, and testing (Problem-Complexity, Design-Effort, Design-Size, Defects-Introduced, Testing-Effort, Defects-Detected, Defects-Density-At-Testing, Residual-Defect-Count, and Residual-Defect-Density). The proper causal relationships among those software life-cycle processes are then captured and reflected as arcs connecting the variables. A tool is then used with regard to the BBN model in the following manner. For given facts about Design-Effort and Design-Size as input, the tool will use Bayesian inference to derive the probability distributions for Defects-Introduced, Defects-Detected and Defect-Density.

Reusability prediction. Predictive models are built through DT in (Mao et al., 1998) to verify the impact of some internal properties of object-oriented applications on reusability. Effort is focused on establishing a correlation between component reusability and three software attributes (inheritance, coupling and complexity). The experimental results show that some software metrics can be used to predict, with a high level of accuracy, the potential reusable classes.

Software release timing. How to determine the software release schedule is an issue that has impact on both the software product developer and the user and the market. A method, based on NN, is proposed in (Dohi et al., 1999) for estimating the optimal software release timing. The method adopts the cost minimization criterion and translates it into a time series forecasting problem. NN is then used to estimate the fault-detection time in the future.

Testability prediction. The work reported in (Khoshgoftaar et al., 2000) describes a case study in which NN is used to predict the testability of software modules from static measurements of the source code. The objective in the study is to predict a quantity between zero and one whose distribution is highly skewed toward zero, which proves to be difficult for standard statistical techniques. The results echo the salient feature of NN-based predictive models that have been discussed so far: its ability to model nonlinear relationships.

3.1.2. Property and model discovery. ML methods are used to identify or discover useful information about software entities. Work in (Bratko and Grobelnik, 1993) explores using ILP to discover loop invariants. The approach is based on collecting execution traces of a program to be proven correct and using them as learning examples of an ILP system. The states of the program variables at a given point in the execution represent positive examples for the condition associated with that point in the program. A controlled closed-world assumption is utilized to generate negative examples.

In (Abd-El-Hafiz, 2000), NN is used to identify objects in procedural programs as an effort to facilitate many maintenance activities (reuse, understanding). The approach is based on cluster analysis and is capable of identifying abstract data types and groups of routines that reference a common set of data.

A data analysis technique called process discovery is proposed in (Cook and Wolf, 1998) that is implemented in terms of NN. The approach is based on first capturing data describing process events from an on-going process and then generating a formal model of the behavior of that process. Another application involves the use of EBL to synthesize models of programming activities or software processes (Garg and Bhansali, 1992). It generates a process fragment (a group of primitive actions which achieves a certain goal given some preconditions) from a recorded process history.

3.1.3. Transformation. The work in (Ryan and Ivan, 1999; Ryan 2000) describes a GP system that can transform serial programs into functionally identical parallel programs. The functional identical property between the input and the output of the transformation can be proven, which greatly enhances the opportunities of the system being utilized in commercial environments.

A module architecture assistant is developed in (Schwanke and Hanson, 1994) to help assist software architects in improving the modularity of large programs. A model for modularization is established in terms of nearest-neighbor clustering and classification, and is used to make recommendations to rearrange module membership in order to improve modularity. The tool learns similarity judgments that match those of the software architect through performing back propagation on a specialized neural network.

GA is used in (Choi and Wu, 1998) in experimenting and evaluating a partitioning and allocation model for mapping object-oriented applications to heterogeneous distributed environments. By effectively distributing software components of an object-oriented system in a distributed environment, it is hoped to achieve performance goals such as load balancing, maximizing concurrency and minimizing communication costs.

3.1.4. Generation and synthesis. In (Bergadano and Gunetti, 1996), a test case generation method is proposed that is based on ILP. An adequate test set is generated as a result of inductive learning of programs from finite sets of input-output examples. The method scales up well when the size or the complexity of the program to be tested grows. It stops being practical if the number of alternatives (or possible errors) becomes too large.

A tool is reported in (Michael and McGraw, 1998; Michael et al., 2001) that uses, among other things, GA to generate dynamic test data for C/C++ programs. The tool is fully automatic and supports all C/C++ language constructs. Test results have been obtained for programs containing up to 2000 lines of source code with complex, nested conditionals.

Synthesizing Unix shell scripts from a high-level specification is made possible through IBL in (Bhansali and Harandi, 1993). The tool has a retrieval mechanism that allows an appropriate source analog to be automatically retrieved given a description of a target problem. Several domain specific retrieval heuristics are utilized to estimate the closeness of two problems at implementation level based on their perceived closeness in the specification level. Though the prototype system demonstrates the viability of the approach, the scalability remains to be seen.

A prototype of a software engineering environment is described in (Bailin et al., 1991) that combines CBR and EBL to synthesize design repair rules for software design. Though the preliminary results are promising, the generality of the learning mechanism and the scaling-up issue remain to be open questions, as cautioned by the authors. In (Harandi and Lee, 1991), IBL provides the impetus to a system that acquires software design schemas from design cases of existing applications.

GP is used in (Qureshi, 1996) to automatically generate agent programs that communicate and interact to solve problems. However, the reported work so far is on a two-agent scenario. Another GP based approach is geared toward generating abstract data types, i.e., data structures and the operations to be performed on them (Landgon, 1995).

In (Minton and Wolfe, 1994), CL and AL are used in synthesizing search programs for a Lisp code generator in the domain of combinatorial integer constraint satisfaction problems.

GA is behind the effort in generating project management schedules in (Chang et al., 2001). Using a programmable goal function, the technique can generate a near-optimal allocation of resources and a schedule that satisfies a given task structure and resource pool.

3.1.5. Reuse library construction and maintenance. This area presents itself as a fertile ground for CBR applications. In (Ostertag et al., 1992), CBR is the corner stone of a reuse library system. A component in the library is represented in terms of a set of feature/term pairs. Similarity between a target and a candidate is defined by the distance measure, which is computed through comparator functions based on the subsumption, closeness and package relations.

Components in a software reuse library have an added advantage in that they can be executed on a computer so as to yield stronger results than could be expected from generic CBR. The work reported in (Fouque and Matwin, 1992) takes advantage of this property by first retrieving software modules from the library, adapting them to new problems, and then subjecting those new cases to executions on system-generated test sets in order to evaluate the results of CBR.

CBR can be augmented with additional mechanisms to help aid other issues in reuse library. Such is the case in (Katalagarianos and Vassiliou, 1995) where CBR is adopted in conjunction with a specificity-genericity hierarchy to locate and adopt software components to given specifications. The proposed method focuses its attention on the evolving nature of the software repository.

How to find a better way of organizing reusable components so as to facilitate efficient user retrieval is another area where ML finds its application. GA is used in (Lee et al., 1998) to optimize the multi-way clustering of software components in a reusable class library. Their approach takes into consideration the following factors: number of clusters, similarity within a cluster and similarity among clusters.

In (Basili et al., 1997) DT is used to model and predict the cost of rework in a library of reusable software components. Proscriptive coding rules can be generated from the model that can be used by programmers as guidelines to reduce the cost of rework in the future. The objective of the work is to use DT to help manage the maintenance of reusable components, and to improve the way the components are produced so as to reduce maintenance costs in the library.

A technique called active browsing is incorporated into a tool that helps assist the browsing of a reusable library for desired components (Drummond et al., 2000). An active browser infers its similarity measure from a designer's normal browsing actions without any special input. It then recommends to the designer components it estimates to be close to the target of the search, which is accomplished through a learning process similar to IBL.

EBL is used as the basis to capture and generalize program abstractions developed in practice to increase their potential for reuse (Hill, 1987). The approach is motivated by the explicit domain knowledge embodied in data type specifications and the mechanisms for reasoning about such knowledge used in validating software.

3.1.6. Requirement acquisition. CL is used to support scenario-based requirement engineering in the work reported in (van Lamsweerde and Willemet, 1998). The paper describes a formal method for supporting the process of inferring specifications of system goals and requirements inductively from interaction scenarios provided by stakeholders. The method is based on a learning algorithm that takes scenarios as examples and counter-examples (positive and negative scenarios) and generates goal specifications as temporal rules.

Another work in (Hall, 1995) presents a scenarios-based elicitation and validation assistant that helps requirements engineers acquire and maintain a specification consistent with scenarios provided. The system relies on EBL to generalize scenarios to state and prove validation lemmas. A scenario generation tool is built in (Hall, 1998) that adopts a heuristic approach based on the idea of piecing together partially satisfying scenarios from the requirements library and using EBL to abstract them in order to be able to co-instantiate them.

A technique is developed in (Cohen, 1995) to extract specifications from software using ILP. It allows instrumented code to be run on a number of representative cases, and generate examples of the code's behavior. ILP is then used to generalize these examples to form a general description of some aspect of a system's behavior.

Software specifications are imperfect reflections of a reality, and are prone to errors, inconsistencies and incompleteness. Because the quality of a software system hinges directly on the accuracy and reliability of its specification, there is dire need for tools and methodologies to perform specification enhancement. In (Partridge et al., 2001), DT and NN are used to extract and acquire knowledge from sample problem data for specification refinement and augmentation.

3.1.7. Capture development knowledge. How to capture and manage software development knowledge is the theme of this application group where both papers report work utilizing CBR as the tool. In (Henninger, 1997), a CBR based infrastructure is proposed that supports evolving knowledge and domain analysis methods that capture emerging knowledge and synthesize it into generally applicable forms.

The work in (Leake and Wilson, 2001) describes a CBR based framework that is geared toward supporting aerospace design. In the proposed framework, interactive tools are used to capture expert design knowledge through "concept mapping." CBR is primarily used to facilitate retrieval and interactive adaptation of designs.

3.2. Status

The application patterns of ML methods in the body of existing work are summarized in Table 3.

Figure 3 captures a glimpse of the types of software engineering issues people have been interested in applying ML techniques to. For instance, of the sixty publications included in the classification above, twenty-eight (almost 47%) deal with the issue of how to build models to predict or estimate certain property of software development process or artifacts.

On the other hand, Figure 4 provides some clue on what types of ML techniques people feel comfortable in using. Based on the classification, NN, IBL/CBR and DT are the top three popular techniques, amounting to sixty one percent of the entire ML applications in our study.

Table 4 depicts the distribution of ML algorithms in the seven SE activity areas.

The trend information captured in Figure 5, though only based on the published work we have been able to collect, should be indicative of the increased interest in ML&SE.

The body of existing work we have been able to glean definitely represents the efforts that have been underway to take advantage of the unique perspective ML affords

Table 3. Application patterns of ML methods

Pattern	Description
Convergent	Different ML methods each being applied to the same SE task
Divergent	A single ML method being applied to different SE tasks
Compound	Several ML methods being combined together for a single SE task

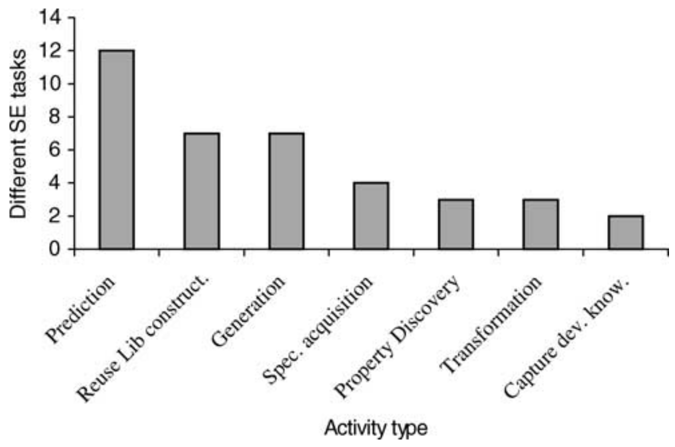

Figure 3. State-of-the-practice from the perspective of SE tasks.

us to explore for SE tasks. Here we point out some general issues in ML&SE as follows.

- *Applicability and justification*. When adopting an ML method to an SE task, we need to have a good understanding of the dimensions of the leaning method and characteristics of the SE task, and find a best match between them. Such a justification offers a necessary condition for successfully applying an ML method to an SE task.
- *Issue of scaling up*. Whether a learning method can be effectively scaled up to handle real world SE projects is an issue to be reckoned with. What seems to be an effective method for a scaled-down problem may hit a snag when being subject to a full-scale version of the problem. Some general guidelines regarding the issue are highly desirable.
- *Performance evaluation*. Given some SE task, some ML-based approaches may outperform their conventional (non-ML) counterparts, others may not offer any per-

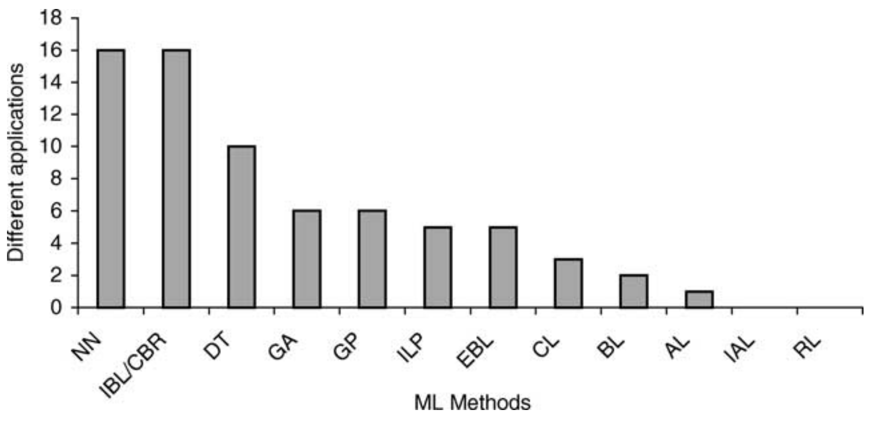

Figure 4. State-of-the-practice from the perspective of ML algorithms.

Table 4. ML methods in SE activity areas

	NN	IBL CBR	DT	GA	GP	ILP	EBL	CL	BL	AL	IAL	RL
Prediction	√	√	√	√	√	√		√	√			
Discovery	√					√	√					
Transformation	√	√		√	√							
Generation		√		√	√	√	√	√		√		
Reuse		√	√	√			√					
Acquisition	√		√			√	√	√				
Management		√										

formance boost but just provide a complement or alternative to the available tools, yet another group may fill in a void in the existing repertoire of SE tools. In addition, we are interested in finding out if there are significant performance differences among applicable ML methods for an SE task. To sort out those different scenarios, we need to establish a systematic way of evaluating the performance of a tool.

Let S be a set of SE tasks, and let T_C and T_L contain a set of conventional (non-ML) SE tools and a set of ML-based tools, respectively. Figure 6 describes some possible scenarios between S and T_C/T_L, where $T_C(s) \subseteq T_C$ and $T_L(s) \subseteq T_L$ indicate a subset of tools applicable to an SE task s, respectively.

If P is defined to be some performance measure (e.g., prediction accuracy), then we can use $P(t, s)$ to denote the performance of t for $s \in S$, where $t \in (T_C(s) \vee T_L(s))$. Let $\Delta ::= < | = | >$. Given an $s \in S$, the performance of two applicable tools can be compared in terms of the following relationships:

$$P(t_i, s)\Delta P(t_j, s), \quad \text{where} \quad t_i \in T_C(s) \wedge t_j \in T_L(s),$$
$$P(t_k, s)\Delta P(t_l, s), \quad \text{where} \quad t_k, t_l \in T_L(s) \wedge |T_L(s)| > 1.$$

- *Integration*. How can an ML-based tool be seamlessly integrated into the SE development environment or tool suite is another issue that deserves attention. If it takes a heroic effort for the tool's integration, it may ultimately affects its applicability.

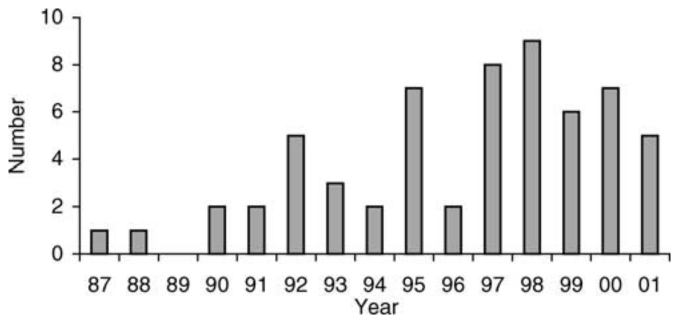

Figure 5. Trend of publications on applying ML algorithms in SE.

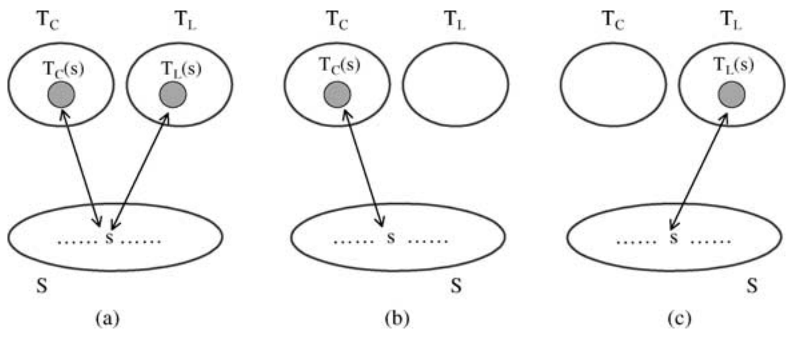

Figure 6. Relationships between S and T_C, and between S and T_L.

4. Applying ML algorithms to SE tasks

In this section, we first discuss some general steps regarding applying machine learning methods to the real world problems. We then provide a guideline on how to select the type of learning methods for a given task. Finally, we demonstrate through some examples how software engineering tasks can be formulated as learning problems and approached using machine learning algorithms.

4.1. General procedure

In applying machine learning to solving any real-world problem, there is usually some course of actions to follow. What we propose is a guideline that has the following steps:

Problem formulation. The first step is to formulate a given problem such that it conforms to the framework of a particular learning method chosen for the task. Different learning methods have different inductive bias, adopt different search strategies that are based on various guiding factors, have different requirements regarding domain theory (presence or absence) and training data (valuation and properties), and are based on different justifications of reasoning (refer to Figure 2). All these issues must be taken into consideration during the problem formulation stage. This step is of pivotal importance to the applicability of the learning method. Strategies such as divide-and-conquer may be needed to decompose the original problem into a set of sub-problems more amenable to the chosen learning method. Sometimes, the best formulation of a problem may not always be the one most intuitive to a machine learning researcher (Langley and Simon, 1995).

Problem representation. The next step is to select an appropriate representation for both the training data and the knowledge to be learned. As can be seen in Figure 2, different learning methods have different representational formalisms. Thus, the representation of the attributes and features in the learning task is often problem-specific and formalism-dependent.

Data collection. The third step is to collect data needed for the learning process. The quality and the quantity of the data needed are dependent on the selected learning method. Data may need to be preprocessed before they can be used in the learning process.

Domain theory preparation. Certain learning methods (e.g., EBL) rely on the availability of a domain theory for the given problem. How to acquire and prepare a domain theory (or background knowledge) and what the quality of a domain theory (correctness, completeness) is therefore become an important issue that will affect the outcome of the learning process.

Performing the learning process. Once the data and a domain theory (if needed) are ready, the learning process can be carried out. The data will be divided into a training set and a test set. If some learning tool or environment is utilized, the training data and the test data may need to be organized according to the tool's requirements. Knowledge induced from the training set is validated on the test set. Because of different splits between the training set and test set, the learning process itself is an iterative one.

Analyzing and evaluating learned knowledge. Analysis and evaluation of learned knowledge is an integral part of the learning process. The interestingness and the performance of the acquired knowledge will be scrutinized during this step, often with the help from human experts, which hopefully will lead to the knowledge refinement. If learned knowledge is deemed insignificant, uninteresting, irrelevant, or deviating, this may be indicative to the need for revisions at early stages such as problem formulation and representation. There are known practical problems in many learning methods such as *overfitting*, *local minima*, or *curse of dimensionality* that are due to either data inadequacy, noise or irrelevant attributes in data, nature of a search strategy, or incorrect domain theory.

Fielding the knowledge base. What this step entails is that the learned knowledge be used (Langley and Simon, 1995). The knowledge could be embedded in a software development system or a software product, or used without embedding it in a computer system.

As observed in (Langley and Simon, 1995), the power of machine learning methods does not come from a particular induction method, but instead from proper formulation of the problems and from crafting the representation to make learning tractable.

4.2. Guideline for selecting learning methods

There exists a wide spectrum of the availabilities for data and domain theories (models) in software engineering tasks. Quantitatively, some tasks may be data-rich while others data-poor; qualitatively, available data may be ranging from noisy, incomplete to accurate, adequate. On the other hand, the availability of a domain theory (a model or models, or some background knowledge) for a given SE task may vary from correct and complete, to inaccurate or incomplete, or to nonexistent.

Table 5. Comparison of inductive learning and analytical learning

	Inductive learning	Analytical learning
Objective	Formulate general hypotheses that fit observed training data	Formulate general hypotheses that fit domain theory
Justification	Statistical inference	Deductive inference
Advantage	Require no prior knowledge	Learn from scarce data
Disadvantage	Can fail if there exist scarce data, or incorrect inductive bias	Can be misled when given incorrect or insufficient domain theory
Method	DT, NN, GA, ILP	AL, EBL

Two paradigms exist in ML: *inductive learning* and *analytical learning*, and comparison of the two yields the following result in Table 5 (Mitchell, 1997).

Because the availability and utilization of data and domain theory play a pivotal role in these two paradigms (learning objectives, complementary merits and demerits), we can use data and domain theory as guiding factors in considering the adoption of learning methods (Figure 7).

When a given task is data-rich, methods of inductive learning can be considered. If there exists a well-defined model for a task, then we can adopt analytical learning methods. Two paradigms can be combined to form a hybrid *inductive-analytical* learning approach. We can utilize hybrid methods in situations where both data and domain theory are less than desirable. Methods of either paradigm will be good candidates if a task has both an adequate domain theory and plenty of data.

The use of such a dichotomy in data and domain theory is just the first step toward the learning method selection. Additional properties (as given in Figure 2) should be taken into consideration subsequently. The guidelines for learning method selections, which we include below in Tables 6–15, are based on the following factors: domain

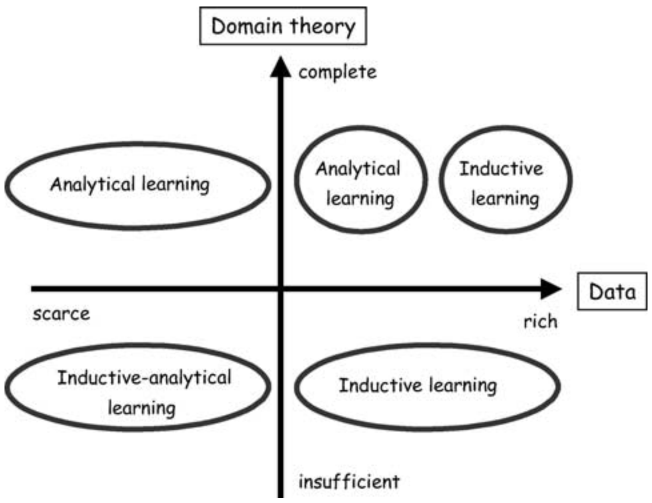

Figure 7. Choice of inductive and analytical learning methods.

Table 6. Guideline for selecting AL/EBL

Domain knowledge	Correct and complete domain knowledge required.
Training data	Small number of training cases needed.
Target function	Form of learned function is known and expressed as set of rules.
Advantages	Ability to use domain theory and deductive reasoning to generalize from small amount of training data.
Disadvantages	Can be misled if domain theory is not correct or complete.
Applicable SE activities	Requirement engineering (generalizing specifications, scenario based specification acquisition).

Table 7. Guideline for selecting BL

Domain knowledge	Required in terms of prior probabilities.
Training data	Incrementally decrease or increase the estimated probability.
Target function	Can be represented either explicitly (as a MAP hypothesis) or implicitly (as providing classifications for unseen cases).
Advantages	Flexible in learning target function. Probabilistic prediction.
Disadvantages	Requiring many initial probabilities. Computational cost.
Applicable SE activities	Predicting defects. Analyzing cost models.

Table 8. Guideline for selecting CL

Domain knowledge	Not required.
Training data	Need to be correct and free of errors. Organized as positive and negative examples.
Target function	Form of learned function is known and expressed in terms of a conjunction of constraints.
Advantages	Version Space (general-to-specific ordering) and Candidate_Elimination algorithm provide a general structure for concept learning problems.
Disadvantages	Learning algorithm not robust to noisy data. H may be incomplete due to its inductive bias.
Applicable SE activities	Deriving system specifications and properties.

Table 9. Guideline for selecting DT learning

Domain knowledge	Not required.
Training data	Adequate data needed to avoid overfitting. Missing values tolerated.
Target function	Discrete-valued. Form of learned function is known and expressed as decision trees.
Advantages	Robust to noisy data. Capable of learning disjunctive expressions.
Disadvantages	Overfitting.
Applicable SE activities	Predicting faults, costs, development efforts, and reusability. Acquiring specifications. Diagnosing system errors.

Table 10. Guideline for selecting GA/GP learning

Domain knowledge	Not required.
Training data	Not needed (some test data may be needed for fitness evaluation).
Target function	Expressed as bit strings (GA) or program trees (GP).
Advantages	Suited to tasks where functions to be approximated are complex. Algorithms can be easily parallelized.
Disadvantages	Crowding. Bloating.
Applicable SE activities	Predicting faults, size, and development efforts. Program transformation. Program synthesizing. Prototyping.

Table 11. Guideline for selecting IBL/CBR

Domain knowledge	Not required.
Training data	Plenty data needed.
Target function	Can be represented either explicitly (as a linear function) or implicitly (as providing classifications for unseen cases). Local approximation.
Advantages	Training is fast. Can learn complex functions. Do not lose information.
Disadvantages	Slow at query time. Curse of dimensionality.
Applicable SE activities	Reuse library. Tasks scheduling and planning. Diagnosis.

knowledge need, training data requirement, form of target function, advantages, disadvantages and applicable SE activities.

Figure 8 indicates where each learning method may fit in the domain-theory/training-data dichotomy.

4.3. Formulating software engineering tasks as learning problems

In this subsection, we will consider some software engineering tasks as learning problems and attempt to formulate them under the frameworks of appropriate learning methods. The choices of learning methods are made based on the guidelines discussed

Table 12. Guideline for selecting ILP learning

Domain knowledge	Background (domain) knowledge required.
Training data	Organized as positive and negative examples.
Target function	Form of learned function is known and expressed as set of rules.
Advantages	Expressive and human readable representation of learned function. Induction formulated as inverse of deduction. Background knowledge guided search.
Disadvantages	Not robust to noisy data. Intractable search space in general case. Increased background knowledge results in increased complexity of hypothesis space. No guarantee to find the smallest or best set of rules.
Applicable SE activities	Predicting faults, and cost. Identifying program invariants. Test data generation. Reverse engineering (extracting specifications from software).

Table 13. Guideline for selecting IAL

Domain knowledge	Required but does not have to be perfect.
Training data	Training data needed and possibly containing errors.
Target function	Form of learned function can be either unknown or known, depending on how domain theory and training data are combined to constrain the search process.
Advantages	Enjoying the benefits of both inductive and analytical learning. Better generalization accuracy (due to domain knowledge). Ability to circumvent imperfect domain knowledge (via training data). A framework where different inductive and analytical methods can be accommodated.
Disadvantages	Complex.
Applicable SE activities	Predicting faults, size, development efforts, reliability and testability. Generating test cases, and specifications. Identifying objects and module properties. Recognizing patterns in software systems.

Table 14. Guideline for selecting NN learning

Domain knowledge	Not required.
Training data	Need to have plentiful data. Training data represented as many attribute-value pairs and possibly containing errors.
Target function	Form of learned function is unknown but human readability of learned result is unimportant.
Advantages	Robust to errors in training data. Can learn complex functions (non-liner, continuous functions). Parallel, distributed learning process.
Disadvantages	Slow training and convergence process. Multiple local minima in error surface. Overfitting.
Applicable SE activities	Predicting faults, size, development efforts, reliability and testability. Identifying objects and module properties. Recognizing patterns in software systems.

Table 15. Guideline for selecting RL

Domain knowledge	Not required.
Training data	Not available.
Target function	Learned function is represented as a control strategy from state space to action space.
Advantages	An unsupervised paradigm geared toward goal-directed learning and decision-making by an agent through direct interaction with its environment without relying on training data or domain theory.
Disadvantages	Exploration-exploitation dilemma.
Applicable SE activities	Scheduling, planning and resource allocation in software project management.

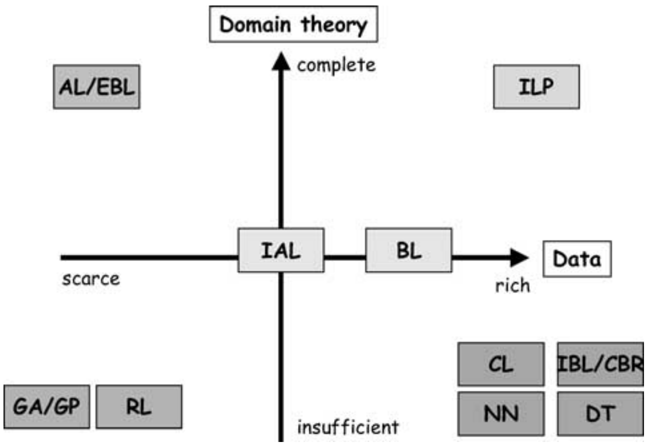

Figure 8. Learning methods in the domain-theory/training-data dichotomy.

in the previous subsection. Our effort will be primarily focused on the problem formulation aspect.

Component reuse. Component retrieval from a software repository is an important issue in supporting software reuse. This task matches the characteristics of IBL, and can be formulated into an instance-based learning problem as follows:

(1) Components in a software repository are represented as points in the n-dimensional Euclidean space (or cases in a case base).
(2) Information in a component can be divided into *indexed* and *unindexed* portions (attributes). Indexed information is used for retrieval purpose and unindexed information is used for contextual purpose. Because of the *curse of dimensionality* problem, the choice of indexed attributes must be judicious.
(3) Queries to the repository for desirable components can be represented as constraints on indexible attributes.
(4) Similarity measures for the nearest neighbors of the desirable component can be based on the standard Euclidean distance, distance-weighted measure, or symbolic measure.
(5) The possible retrieval methods include *K-Nearest Neighbor*, *inductive retrieval*, *Locally Weighted Regression*.
(6) The adaptation of the retrieved component for the task at hand can be *structural* (applying adaptation rules directly to the retrieved component), or *derivational* (reusing adaptation rules that generated the original solution to produce a new solution). Techniques range from no adaptation for problems with simple solutions, to *parameter adjustment*, *variable reinstantiation*, *derivational replay*, or *model-guided repair*. This step may involve human expert.
(7) Retain the new case in the case base.

Rapid prototyping. Rapid prototyping is an important tool for understanding and validating software requirements. In addition, software prototypes can be used for other purposes such as user training and system testing (Sommerville, 1996). Different techniques have been developed for *evolutionary* and *throw-away* prototyping. We can augment the existing techniques by including a GP based approach. Because GP does not require domain knowledge, and is suitable for generating complex functions, it is a good candidate for the task.

In GP, a computer program is often represented as a program tree where the internal nodes correspond to a set of functions used in the program and the external nodes (terminals) indicate variables and constants used as input to functions. For a given problem, GP starts with an initial population of randomly generated computer programs. The evolution process of generating a final computer program that solves the given problem hinges on some sort of *fitness* evaluation and probabilistically reproducing the next generation of the program population through some *genetic operations*. Given a GP development environment such as the one in (Kramer and Zhang, 2000), the framework of a GP-based rapid prototyping process can be described as follows:

(1) Define sets of functions and terminals to be used in the developed (prototype) systems.
(2) Define a fitness function to be used in evaluating the worthiness of a generated program. Test data (input values and expected output) may be needed in assisting the evaluation.
(3) Generate the initial program population.
(4) Determine selection strategies for programs in the current generation to be included in the next generation population.
(5) Decide how the genetic operations (*crossover* and *mutation*) are carried out during each generation and how often these operations are performed.
(6) Specify the terminating criteria for the evolution process and the way of checking for termination.
(7) Translate the final returned program into a desired programming language format.

Expert system shells have been used as environments for software prototyping (Lowry, 1992). An obvious benefit of using GP over expert system shells is that it does not rely on the availability of any domain theory.

Requirement engineering. Requirement engineering refers to the process of establishing the services a system should provide and the constraints under which it must operate (Sommerville, 1996). A requirement may be *functional* or *non-functional*. A functional requirement describes a system service or function, whereas a non-functional requirement represents a constraint imposed on the system. How to obtain functional requirements of a system is the focus here. The situation in which ML algorithms will be particularly useful is when there exist empirical data from a problem domain that describe how the system should react to certain inputs. Under this circumstance, functional requirements can be “learned” from the data through some learning algorithm. This fits a typical supervised learning profile and the learning process can be formulated as follows.

(1) Let X and C denote the domain and the co-domain of a system function f to be learned (f is a part of the functional requirement for the system). The data set D is then defined as:

$D = \{\langle x_i, c_k\rangle | x_i \in X \wedge c_k \in C\}$.

(2) The target function f to be learned satisfies the condition: $\forall x_i \in X$ and $\forall c_k \in C$, $f(x_i) = c_k$.
(3) The learning algorithms applicable here have to be of *supervised* type. Depending on the nature of the data set D, different learning algorithms (AL, BL, CL, DT, ILP) can be utilized to capture (learn) a system's functional requirements.

Reverse engineering. Legacy systems are old systems that are critical to the operation of an organization which uses them and that must still be maintained. Most legacy systems were developed before software engineering techniques were widely used. Thus they may be poorly structured and their documentation may be either out-of-date or non-existent. In order to bring to bear the legacy system maintenance, the first task is to recover the design or specification of a legacy system from its source or executable code (hence, the term of reverse engineering, or program comprehension and understanding). Below we describe a framework for learning functional specification of a legacy software system from its executable code.

(1) Given the executable code p and its input data set X, and output set C, the training data set D is defined as: $D = \{\langle x_i, p(x_i)\rangle | x_i \in X \wedge p(x_i) \in C\}$.
(2) The process of deriving the functional specification f for p can be described as a learning problem in which f is obtained through some ML algorithm. f satisfies the following:

$$\forall x_i \in X[f(x_i) = p(x_i)].$$

(3) Many supervised learning algorithms can be used here to obtain f (e.g., CL).

Cliché recognition (Rich and Waters, 1986) has been used to match stereotyped programming patterns to a program's data and control flow so as to help with software understanding. Contrasting cliché recognition with CL, the former results in substantial upfront overhead because a library of clichés must be in place before the approach can be applied.

Validation. Verification and validation are important checking processes to make sure that a software system is correctly developed and conforms to its specification. To check a software implementation against its specification as part of the validation process, we assume the availability of both a specification and an executable code. This checking process can be performed as an *analytic learning* task as follows:

(1) Let X and C be the domain and co-domain of the implementation (executable code) p, which is defined as: p: $X \to C$.
(2) The training set D is defined as: $D = \{\langle x_i, p(x_i)\rangle | x_i \in X \wedge p(x_i) \in C\}$.
(3) The specification for p is denoted as B, which corresponds to the domain theory in the analytic learning.
(4) The validation checking is defined as follows:

$$p \text{ is valid if } \forall\langle x_i, p(x_i)\rangle \in D[B \wedge x_i \vdash p(x_i)].$$

(5) Explanation-based learning algorithms can be utilized to carry out the checking process.

Test oracle generation. Functional testing involves executing a program under test and examining the output from the program. An *oracle* is needed in functional testing in order to determine if the output from a program is correct. The oracle can be a human or a software one (Peters and Parnas, 1998). The approach we propose here allows a test oracle to be learned as a function from the specification and a small set of training data. The learned test oracle can then be used for the functional testing purpose.

(1) Let X and C be the domain and co-domain of the program p to be tested. Let B be the specification for p.
(2) Define a small training set D as: $D = \{\langle x_i, p(x_i)\rangle | x_i \in X' \wedge X' \subset X \wedge p(x_i) \in C\}$.
(3) Use the explanation-based learning (EBL) to generate a test oracle Θ ($\Theta : X \rightarrow C$) for p from B and D.
(4) Use Θ for the functional testing: $\forall x_i \in X$ [output of p is correct if $p(x_i) = \Theta(x_i)$].

Test adequacy criteria. Software test data adequacy criteria are rules that determine if a software product has been adequately tested (Zhu, 1996). A test data adequacy criterion ζ is a function: ζ: $P \times S \times T \rightarrow$ {true, false} where P is a set of programs, S a set of specifications and T the class of test sets. $\zeta(p, s, t) =$ true means that t is adequate for testing program p against specification s according to criterion ζ. Since ζ is essentially a Boolean function, we can use a strategy such as CL to learn the test data adequacy criteria.

(1) Define the instance space X as: $X = \{\langle p_i, s_j, t_k\rangle | p_i \in P \wedge s_j \in S \wedge t_k \in T\}$.
(2) Define the training data set D as: $D = \{\langle x, \zeta(x)\rangle | x \in X \wedge \zeta(x) \in V\}$, where V is defined as: $V =$ {true, false}.
(3) Use the concept of *version space* and the *candidate-elimination* algorithm in CL to learn the definition of ζ.

Software defect prediction. Software defect prediction is a very useful and important tool to gauge the likely delivered quality and maintenance effort before software systems are deployed (Fenton and Neil, 1999). Predicting defects requires a holistic model rather than a single-issue model that hinges on either size, or complexity, or testing metrics, or process quality data alone. It is argued in (Fenton and Neil, 1999) that all these factors must be taken into consideration in order for the defect prediction to be successful.

BBN proves to be a very useful approach to the software defect prediction problem. A BBN represents the *joint probability distribution* for a set of variables. This is accomplished by specifying (a) a directed acyclic graph (DAG) where nodes represent variables and arcs correspond to *conditional independence* assumptions (causal knowledge about the problem domain), and (b) a set of local conditional probability tables (one for each variable) (Jensen, 1996; Mitchell, 1997). A BBN can be used to infer the probability distribution for a target variable (e.g., "Defects Detected"), which specifies the probability that the variable will take on each of its possible values (e.g., "very low", "low", "medium", "high", or "very high" for the variable "Defects Detected") given the observed values of the other variables. In general, a BBN can be used to compute the probability distribution for any subset of variables given the values or distributions for any subset of the remaining variables. When using a BBN for a decision support system such as software defect prediction, the steps below should be followed.

(1) Identify variables in the BBN. Variables can be: (a) *hypothesis* variables for which the user would like to find out their probability distributions (hypothesis variable are either unobservable or too costly to observe), (b) *information* variables that can be observed, or (c) *mediating* variables that are introduced for certain purpose (help reflect independence properties, facilitate acquisition of conditional probabilities, and so forth). Variables should be defined to reflect the life-cycle activities (specification, design, implementation, and testing) and capture the multi-facet nature of software defects (perspectives from size, testing metrics and process quality). Variables are denoted as nodes in the DAG.
(2) Define the proper causal relationships among variables. These relationships also should capture and reflect the causality exhibited in the software life-cycle processes. They will be represented as arcs in the corresponding DAG.
(3) Acquire a probability distribution for each variable in the BBN. Theoretically well-founded probabilities, or frequencies, or subjective estimates can all be used in the BBN. The result is a set of conditional probability tables one for each variable. The full joint probability distribution for all the defect-centric variables is embodied in the DAG structure and the set of conditional probability tables.

Project effort (cost) prediction. How to estimate the cost for a software project is a very important issue in the software project management. Most of the existing work is based on algorithmic models of effort (Shepperd and Schofield, 1997). A viable alternative approach to the project effort prediction is instance-based learning. IBL yields very good performance for situations where an algorithmic model for the prediction is not possible. In the framework of IBL, the prediction process can be carried out as follows.

(1) Introduce a set of features or attributes (e.g., number of interfaces, size of functional requirements, development tools and methods, and so forth) to characterize projects. The decision on the number of features has to be judicious, as this may become the cause of the *curse of dimensionality* problem that will affect the prediction accuracy.
(2) Collect data on completed projects and store them as instances in the case base.
(3) Define *similarity* or *distance* between instances in the case base according to the symbolic representations of instances (e.g., Euclidean distance in an n-dimensional space where n is the number of features used). To overcome the potential curse of dimensionality problem, features may be weighed differently when calculating the distance (or similarity) between two instances.
(4) Given a query for predicting the effort of a new project, use an algorithm such as *K-Nearest Neighbor*, or, *Locally Weighted Regression* to retrieve similar projects and use them as the basis for returning the prediction result.

5. Concluding remarks

In this paper, we have discussed issues and current status regarding ML applications to SE problems. The existing work certainly proves that the field of SE is a fertile ground for the application of AI techniques in general, and ML methods in particular. We hope our work helps provide an impetus to an increased interest in the niche area of ML&SE. A maturing software engineering discipline will definitely be able to benefit from the awareness and the utility of ML techniques.

What can ML techniques do to the SE essential difficulties given in Brooks' paper? We think ML methods can be used to complement existing SE tools and methodologies to make headway in all aspects of those essential difficulties. The strength of ML methods lies in the fact that they have sound mathematical and logical justifications and can be used to create and compile verifiable knowledge about the design and development of software artifacts. This has been demonstrated in the body of the existing work at least for narrow areas and for individual cases.

How can we avoid potential pitfalls of ML techniques? We think the single most important factor is to avoid mismatches between the characteristics of an SE problem and those of an ML method.

What lies ahead? For SE areas that have already witnessed ML applications, efforts will be needed in developing guidelines for issues regarding applicability, scaling-up, performance evaluation, and tool integration. For those SE areas that have not witnessed ML applications, the issue is how to realize the promise and potential ML techniques have to offer. Thus far, we only confine our discussions on ML applications in software development and maintenance tasks. An even ambitious topic is how to incorporate ML techniques into software products to make them adaptive and self-configuring.

References

Abd-El-Hafiz, S. 2000. Identifying objects in procedural programs using clustering neural networks, *Automated Software Engineering* 7(3): 239–261.

Aha, D. Machine learning resources, http://www.aic.nrl.navy.mil/∼aha/research/machine-learning.html.

Aha, D. Case-based reasoning resources, http://www.aic.nrl.navy.mil/∼aha/research/case-based-reasoning.html.

Bailin, S.C., Gattis, R.H., and Truszkowski, W. 1991. A learning-based software engineering environment, In *Proc. 6th Annual Knowledge-Based Software Engineering Conference*, pp. 198–206.

Basili, V., Condon, S., El Emam, K., Hendrick, R., and Melo, W. 1997. Characterizing and modeling the cost of rework in a library of reusable software components, In *Proc. International Conference on Software Engineering*, pp. 282–291.

Bergadano, F. and Gunetti, D. 1995. *Inductive Logic Programming: From Machine Learning to Software Engineering*, MIT Press.

Bergadano, F. and Gunetti, D. 1996. Testing by means of inductive program learning, *ACM Trans. Software Engineering and Methodology* 5(2): 119–145.

Bhansali, S. and Harandi, M. 1993. Synthesis of Unix programs using derivational analogy, *Machine Learning* 10(1): 7–55.

Binder, B. and Tsai, J.J.P. 1992. KBRMS: An intelligent assistant for requirement definition, *International Journal of Artificial Intelligence Tools* 1(4): 503–522.

Boehm, B. 2000. Requirements that handle IKIWISI, COTS, and rapid change, *IEEE Computer* 33(7): 99–102.

Bratko, I. and Grobelnik, M. 1993. Inductive learning applied to program construction and verification, In J. Cuena (ed.), *AI Techniques for Information Processing*, North-Holland.

Bratko, I. and Muggleton, S. 1995. Applications of inductive logic programming, *Communications of ACM* 38(11): 65–70.

Briand, L., Basili, V., and Hetmanski, C. 1993. Developing interpretable models with optimized set reduction for identifying high-risk software components, *IEEE Transactions on Software Engineering* 19(11): 1028–1043.

Briand, L., Basili, V., and Thomas, W. 1992. A pattern recognition approach for software engineering data analysis, *IEEE Transactions on Software Engineering* 18(11): 931–942.

Briand, L. et al. 1999. An assessment and comparison of common software cost estimation modeling techniques, In *Proc. International Conference on Software Engineering*, pp. 313–322.

Brooks, F. 1995. *The Mythical Man-Month*, Addison-Wesley, Reading, MA.

Brooks, F. 1987. No silver bullet-essence and accidents of software engineering, *IEEE Computer* 20(4): 10–19.

Broy, M. 2001. Toward a mathematical foundation of software engineering methods, *IEEE Transactions on Software Engineering* 27(1): 42–57.

Chang, C., Christensen, M., and Zhang, T. 2001. Genetic algorithms for project management, *Annals of Software Engineering* 11(1): 107–139.

Choi, S. and Wu, C. 1998. Partitioning and allocation of objects in heterogeneous distributed environments using the niched Pareto genetic algorithm, In *Proc. of the Asia–Pacific Software Engineering Conference*, pp. 322–329.

Chulani, S., Boehm, B., and Steece, B. 1999. Bayesian analysis of empirical software engineering cost models, *IEEE Transactions on Software Engineering* 25(4): 573–583.

Cohen, W. 1995. Inductive specification recovery: understanding software by learning from example behaviors, *Automated Software Engineering* 2(2): 107–129.

Cohen, W. and Devanbu, P. 1997. A comparative study of inductive logic programming for software fault prediction, In *Proc. of the Fourteenth International Conference on Machine Learning*.

Cook, J. and Wolf, A. 1998. Discovering models of software processes from event-based data, *ACM Trans. Software Engineering and Methodology* 7(3): 215–249.

Cristianini, N. and Shawe-Taylor, J. 2000. *An Introduction to Support Vector Machines*, Cambridge University Press.

de Almeida, M., Lounis, H., and Melo, W. 1998. An investigation on the use of machine learned models for estimating correction costs, In *Proc. of International Conference on Software Engineering*, pp. 473–476.

de Almeida, M. and Matwin, S. 1999. Machine learning method for software quality model building, In *Proc. of International Symposium on Methodologies for Intelligent Systems*.

Devanbu, P., Brachman, R., Selfridge, P., and Ballard, B. 1991. LaSSIE: A knowledge-based software information system, *Communications of ACM* 34(5): 35–49.

Dietterich, T.G. 1997. Machine learning research: four current directions, *AI Magazine* 18(4): 97–136.

Dohi, T., Nishio, Y., and Osaki, S. 1999. Optimal software release scheduling based on artificial neural networks, *Annals of Software Engineering* 8(1): 167–185.

Dolado, J. 2000. A validation of the component-based method for software size estimation, *IEEE Transactions on Software Engineering* 26(10): 1006–1021.

Drummond, C., Ionescu, D., and Holte, R. 2000. A learning agent that assists the browsing of software libraries, *IEEE Transactions on Software Engineering* 26(12): 1179–1196.

El Emam, K., Benlarbi, S., Goel, N., and Rai, S. 2001. Comparing case-based reasoning classifiers for predicting high risk software components, *Journal of Systems and Software* 55(3): 301–320.

Ernst, M., Cockrell, J., Griswold, W., and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution, *IEEE Transactions on Software Engineering* 27(2): 99–123.

Evett, M., Khoshgoftar, T., Chien, P., and Allen, E. 1998. GP-based software quality prediction, In *Proc. of Third Annual Genetic Programming Conference*, pp. 60–65.

Fenton, N. and Neil, M. 1999. A critique of software defect prediction models, *IEEE Transactions on Software Engineering* 25(5): 675–689.

Finnie, G., Wittig G., and Desharnais, J.-M. 1997. A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning and regression models, *Journal of Systems and Software* 39(3): 281–289.

Fouque, G. and Matwin, S. 1992. CAESAR: a system for case based software reuse, In *Proc. of 7th Knowledge-Based Software Engineering Conference*, pp. 90–99.

Ganesan, K., Khoshgoftaar, T., and Allen, E. 2000. Cased-based software quality prediction, *International Journal of Software Engineering and Knowledge Engineering* 10(2): 139–152.

Garg, P. and Bhansali, S. 1992. Process programming by hindsight, In *Proc. of International Conference on Software Engineering*, pp. 280–293.

Glass, R. 1988. (column), *System Development*, pp. 4–5.

Green, C. et al. 1986. Report on a knowledge-based software assistant, In C. Rich and R.C. Waters (eds.), *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, San Mateo, CA, pp. 377–428.

Hall, R. 1995. Systematic incremental validation of reactive systems via sound scenario generalization, *Automated Software Engineering* 2(2): 131–166.

Hall, R. 1998. Explanation-based scenario generation for reactive system models, In *Proc. of International Conference on Automated Software Engineering*, pp. 115–124.

Harandi, M.T. and Lee, H.Y. 1991. Acquiring software design schemas: a machine learning perspective, In *Proc. of 6th Annual Knowledge-Based Software Engineering Conference*, pp. 188–197.

Henninger, S. 1997. Case-based knowledge management tools for software development, *Automated Software Engineering* 4(3): 319–340.

Hill, W.L. 1987. Machine learning for software reuse, In *Proc. of International Joint Conference on Artificial Intelligence*, pp. 338–344.

Hong, E. and Wu, C. 1997. Criticality models using SDL metrics set, In *Proc. of the 4th Asia–Pacific Software Engineering and International Computer Science Conference*, pp. 23–30.

Jensen, F.V. 1996. *An Introduction to Bayesian Networks*, Springer, Berlin.

Jorgensen, M. 1995. Experience with the accuracy of software maintenance task effort prediction models, *IEEE Transactions on Software Engineering* 21(8): 674–681.

Karunanithi, N., Whitely, D., and Malaiya, Y. 1992. Prediction of software reliability using connectionist models, *IEEE Transactions on Software Engineering* 18(7): 563–574.

Katalagarianos, P. and Vassiliou, Y. 1995. On the reuse of software: a case-based approach employing a repository, *Automated Software Engineering* 2(1): 55–86.

Khoshgoftaar, T., Pandya, A., and Lanning, D., 1995. Application of neural networks for predicting faults, *Annals of Software Engineering* 1: 141–154.

Khoshgoftaar, T., Allen, E., Hudepohl, J., and Aud, S. 1997. Applications of neural networks to software quality modeling of a very large telecommunications system, *IEEE Transactions on Neural Networks* 8(4): 902–909.

Khoshgoftaar, T., Allen E., and Xu, Z. 2000. Predicting testability of program modules using a neural network, In *Proc. of IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, pp. 57–62.

Kramer, M. and Zhang, D. 2000. Gaps: a genetic programming system, In *Proc. of IEEE International Conference on Computer Software and Applications (COMPSAC 2000)*, pp. 614–619.

Langdon, W. 1995. Evolving data structures using genetic programming, In *Proc. of the Sixth International Conference on Genetic Algorithms*, pp. 295–302.

Langley, P. and Simon, H. 1995. Applications of machine learning and rule induction, *Communications of ACM* 38(11): 55–64.

Lanubile, F. and Visaggio, G. 1997. Evaluating predictive quality models derived from software measures: lessons learned, *Journal of Systems and Software* 38: 225–234.

Leake, D. and Wilson, D. 2001. A case-based framework for interactive capture and reuse of design knowledge, *Applied Intelligence* 14(1): 77–94.

Lee, B., Moon, B., and Wu, C. 1998. Optimization of multi-way clustering and retrieval using genetic algorithms in reusable class library, In *Proc. of the Asia–Pacific Software Engineering Conference*, pp. 4–11.

Liu, A. and Tsai, J.J.P. 1996. A knowledge-based approach for requirements analysis, *International Journal of Artificial Intelligence Tools* 5(2): 167–184.

Lowry, M. 1992. Software engineering in the twenty first century, *AI Magazine* 14(3): 71–87.

Mao, Y., Sahraoui, H., and Lounis, H. 1998. Reusability hypothesis verification using machine learning techniques: a case study, In *Proc. of 13th IEEE International Conference on Automated Software Engineering*, pp. 84–93.

Mendonca, M. and Sunderhaft, N.L. 1999. Mining software engineering data: a survey, DACS State-of-the-Art Report, http://www.dacs.dtic.mil/techs/datamining/.

Menzies, T. 2001. *Practical Machine Learning for Software Engineering and Knowledge Engineering*, Handbook of Software Engineering and Knowledge Engineering, World Scientific.

Michael, C. and McGraw, G. 1998. Automated software test data generation for complex programs, In *Proc. of International Automated Software Engineering Conference*, pp. 136–146.

Michael, C., McGraw, G., and Schatz, M. 2001. Generating software test data by evolution, *IEEE Transactions on Software Engineering* 27(12) 1085–1110.

Michalski, R.S., Bratko, I., and Kubat, M. (eds.). 1998. *Machine Learning and Data Mining: Methods and Applications*, John Wiley & Sons Ltd., New York.

Minton, S. and Wolfe, S. 1994. Using machine learning to synthesize search programs, In *Proc. of 9th Knowledge-Based Software Engineering Conference*, pp. 31–38.

Mitchell, T. 1997a. *Machine Learning*, McGraw-Hill, New York.

Mitchell, T. 1997b. Does machine learning really work?, *AI Magazine* 18(3): 11–20.

Mitchell, T. 1999. Machine learning and data mining, *Communications of ACM* 42(11): 31–36.

Mostow, J. (ed.). 1985. Special issue on artificial intelligence and software engineering, *IEEE Transactions on Software Engineering* 11(11): 1253–1408.

Ostertag, E., Hendler, J., Diaz, R.P., and Braun, C. 1992. Computing similarity in a reuse library system: an AI-based approach, *ACM Transactions on Software Engineering and Methodology* 1(3): 205–228.

Parnas, D. 1979. Designing software for ease of extension and contraction, *IEEE Transactions on Software Engineering* 5(3): 128–137.

Partridge, D. 1998. *Artificial Intelligence and Software Engineering*, AMACOM.

Partridge, D., Wang, W., and Jones, P., 2001. Artificial intelligence techniques for software system enhancement, Research Report No. 399, School of Engineering and Computer Science, University of Exeter, U.K.

Peters, D. and Parnas, D. 1998. Using test oracles generated from program documentation, *IEEE Transactions on Software Engineering* 24(3) 161–173.

Porter, A. and Selby, R. 1990. Empirically-guided software development using metric-based classification trees, *IEEE Software* 7: 46–54.

Provost, F. and Kohavi, R. 1998. On applied research in machine learning, *Machine Learning* 30(2/3): 127–132.

Quinlan, J.R. 1990. Learning logical definitions from relations, *Machine Learning* 5(3): 239–266.

Qureshi, A. 1996. Evolving agents, Research Note, University College London, RN-96-4.

Rich, C. and Waters, R. (eds.). 1986. *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, San Mateo, CA.

Ryan, C. 2000. *Automatic Re-engineering of Software Using Genetic Programming*, Kluwer Academic, Dordrecht.

Ryan, C. and Ivan, L. 1999. An automatic software re-engineering tool based on genetic programming, In L. Spector et al. (eds.), *Advances in Genetic Programming*, Vol. 3, MIT Press, pp. 15–39.

Saitta, L. and Neri, F. 1998. Learning in the 'real world', *Machine Learning* 30(2/3): 133–163.

Schwanke, R. and Hanson, S. 1994. Using neural networks to modularize software, *Machine Learning* 15(2): 137–168.

Selby, R. and Porter, A. 1988. Learning from examples: generation and evaluation of decision trees for software resource analysis, *IEEE Transactions on Software Engineering* 14: 1743–1757.

Selfridge, O. 1993. The gardens of learning: a vision for AI, *AI Magazine* 14(2): 36–48.

Shepperd, M. and Schofield, C. 1997. Estimating software project effort using analogies, *IEEE Transactions on Software Engineering* 23(12): 736–743.

Shukla, K. 2000. Neuro-genetic prediction of software development effort, *Information and Software Technology* 42(10): 701–713.

Sommerville, I. 1996. *Software Engineering*, Addison-Wesley, Reading, MA.

Srinivasan, K. and Fisher, D. 1995. Machine learning approaches to estimating software development effort, *IEEE Transactions on Software Engineering* 21(2): 126–137.

Sutton, R. and Barto, A. 1999. *Reinforcement Learning: An Introduction*, MIT Press.

Tsai, J.J.P., Li, B., and Weigert, T. 1998. A logic-based transformation system, *IEEE Transactions on Knowledge and Data Engineering* 10(1): 91–107.

Tsai, J.J.P. and Weigert, T. 1993. *Knowledge-Based Software Development for Real-Time Distributed Systems*, World Scientific Inc., Singapore.

van Lamsweerde and Willemet, L. 1998. Inferring declarative requirements specification from operational scenarios, *IEEE Transactions on Software Engineering* 24(12): 1089–1114.

Vicinanza, S., Prietulla, M.J., and Mukhopadhyay, T. 1990. Case-based reasoning in software effort estimation, In *Proc. of 11th Int'l. Conf. on Information Systems*, pp. 149–158.

Welty, C. and Selfridge, P. 1995. Artificial intelligence and software engineering: breaking the toy mold, *Automated Software Engineering* 4(3): 255–270.

Zhang, D. 2000. Applying machine learning algorithms in software development, In *Proc. of Monterey Workshop on Modeling Software System Structures*, Santa Margherita Ligure, Italy, pp. 275–285.

Zhang, D. and Tsai, J.J.P. 2002. Machine learning and software engineering, In *Proc. of 14th IEEE International Conference on Tools with Artificial Intelligence*, pp. 22–29.

Zhu, H. 1996. A formal analysis of the subsume relation between software test adequacy criteria, *IEEE Transactions on Software Engineering* 22(4): 248–255.

Du Zhang received his Ph.D. degree in computer science from the University of Illinois at Chicago in 1987. He is a Professor in Department of Computer Science, California State University, Sacramento. He has over 70 publications in journals, conferences and book chapters. He is Program Committee Chair for the 15th IEEE International Conference on Tools with Artificial Intelligence to be held in November 2003 at Sacramento. During 1994–1995 academic year, he spent his sabbatical leave at University of Minnesota, Minneapolis and at Naval Postgraduate School. He is a senior member of IEEE, a member of the American Association for Artificial Intelligence, and a member of ACM. His current research interests include: machine learning and data mining applications in software engineering and bioinformatics, semantics and anomalies of knowledge-based systems, intelligent agents in internet and web environment, and high-level Petri net applications.

Jeffrey J.P. Tsai received his Ph.D. degree in computer science from the Northwestern University, Evanston, Illinois. He is a Professor in the Department of Electrical Engineering and Computer Science at the University of Illinois at Chicago, where he is also the Director of Distributed Real-Time Intelligent Systems Laboratory. He co-authored *Knowledge-Based Software Development for Real-Time Distributed Systems* (World Scientific, 1993), *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis* (John Wiley and Sons, Inc., 1996), *Compositional Verification of Concurrent and Real-Time systems* (Kluwer, 2002), co-edited *Monitoring and Debugging Distributed Real-Time Systems* (IEEE/CS Press, 1995), and has published extensively in the areas of knowledge-based software engineering, software architecture, requirements engineering, formal methods, agent-based systems, and distributed real-time systems.

Dr. Tsai was the recipient of a University Scholar Award from the University of Illinois in 1994 and was presented a Technical Achievement Award from the IEEE Computer Society in 1997. He is currently the Co-Editor-in-Chief of International Journal of Artificial Intelligence Tools and on the editorial board of the International Journal of Software Engineering and Knowledge Engineering, and chairs the IEEE/CS Technical Committee on Multimedia. He is a fellow of the IEEE, the AAAS, and the SDPS.