Preface

Machine learning is the study of how to build computer programs that improve their performance at some task through experience. The hallmark of machine learning is that it results in an improved ability to make better decisions. Machine learning algorithms have proven to be of great practical value in a variety of application domains. 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.

To meet the challenge of developing and maintaining large and complex software systems in a dynamic and changing environment, machine learning methods have been playing an increasingly important role in many software development and maintenance tasks. The past two decades have witnessed an increasing interest, and some encouraging results and publications in machine learning application to software engineering. As a result, a crosscutting niche area emerges. Currently, there are efforts to raise the awareness and profile of this crosscutting, emerging area, and to systematically study various issues in it. It is our intention to capture, in this book, some of the latest advances in this emerging niche area.

Machine Learning Methods

Machine learning methods fall into the following broad categories: supervised learning, unsupervised learning, semi-supervised learning, analytical learning, and reinforcement learning. Supervised learning deals with learning a target function from labeled examples. Unsupervised learning attempts to learn patterns and associations from a set of objects that do not have attached class labels. Semi-supervised learning is learning from a combination of labeled and unlabeled examples. Analytical learning relies on domain theory or background knowledge, instead of labeled examples, to learn a target function. Reinforcement learning is concerned with learning a control policy through reinforcement from an environment.
There are a number of important issues in machine learning:

- How is a target function represented and specified (based on the formalism used to represent a target function, there are different machine learning approaches)? What are the interpretability, complexity, and properties of a target function? How does it generalize?
- What is the hypothesis space (the search space)? What are its properties?
- What are the issues in the search process for a target function? What are heuristics and bias utilized in searching for a target function?
- Is there any background knowledge or domain theory available for the learning process?
- What properties do the training data have?
- What are the theoretical underpinnings and practical issues in the learning process?

The following are some frequently-used machine learning methods in the aforementioned categories.

In concept learning, a target function is represented as a conjunction of constraints on attributes. The hypothesis space \( H \) consists of a lattice of possible conjunctions of attribute constraints for a given problem domain. A least-commitment search strategy is adopted to eliminate hypotheses in \( H \) that are not consistent with the training set \( D \). This will result in a structure called the version space, the subset of hypotheses that are consistent with the training data. The algorithm, called the candidate elimination, utilizes the generalization and specialization operations to produce the version space with regard to \( H \) and \( D \). It relies on a language (or restriction) bias that states that the target function is contained in \( H \). This is an eager and supervised learning method. It is not robust to noise in data and does not have support for prior knowledge accommodation.

In decision tree learning, a target function is defined as a decision tree. Search in decision tree learning is often guided by an entropy-based information gain measure that indicates how much information a test on an attribute yields. Learning algorithms often have a bias for small trees. It is an eager, supervised, and unstable learning method, and is susceptible to noisy data, a cause for overfitting. It cannot accommodate prior knowledge during the learning process. However, it scales up well with large data in several different ways.

In neural network learning, given a fixed network structure, learning a target function amounts to finding weights for the network such that the network outputs are the same as (or within an acceptable range of) the expected outcomes as specified in the training data. A vector of weights in essence defines a target function. This makes the target function very difficult for human to read and interpret. This is an eager, supervised, and unstable learning approach and cannot accommodate prior knowledge. A popular algorithm for feed-forward networks is backpropagation, which adopts a gradient descent search and sanctions an inductive bias of smooth interpolation between data points.

Bayesian learning offers a probabilistic approach to inference, which is based on the assumption that the quantities of interest are dictated by probability distributions, and that optimal decisions or classifications can be reached by reasoning about these probabilities along with observed data. Bayesian learning methods can be divided into two groups based
on the outcome of the learner: the ones that produce the most probable hypothesis given the training data, and the ones that produce the most probable classification of a new instance given the training data. A target function is thus explicitly represented in the first group, but implicitly defined in the second group. One of the main advantages is that it accommodates prior knowledge (in the form of Bayesian belief networks, prior probabilities for candidate hypotheses, or a probability distribution over observed data for a possible hypothesis). The classification of an unseen case is obtained through combined predictions of multiple hypotheses. It also scales up well with large data. It is an eager and supervised learning method and does not require search during learning process. Though it has no problem with noisy data, Bayesian learning has difficulty with small data sets. Bayesian learning adopts a bias that is based on the minimum description length principle.

Genetic algorithms and genetic programming are both biologically-inspired learning methods. A target function is represented as bit strings in genetic algorithms, or as programs in genetic programming. The search process starts with a population of initial hypotheses. Through the crossover and mutation operations, members of current population give rise to the next generation of population. During each step of the iteration, hypotheses in the current population are evaluated with regard to a given measure of fitness, with the fittest hypotheses being selected as members of the next generation. The search process terminates when some hypothesis h has a fitness value above some threshold. Thus, the learning process is essentially embodied in the generate-and-test beam search. The bias is fitness-driven. There are generational and steady-state algorithms.

Instance-based learning is a typical lazy learning approach in the sense that generalizing beyond the training data is deferred until an unseen case needs to be classified. In addition, a target function is not explicitly defined; instead, the learner returns a target function value when classifying a given unseen case. The target function value is generated based on a subset of the training data that is considered to be local to the unseen example, rather than on the entire training data. This amounts to approximating a different target function for a distinct unseen example. This is a significant departure from the eager learning methods where a single target function is obtained as a result of the learner generalizing from the entire training data. The search process is based on statistical reasoning, and consists in identifying training data that are close to the given unseen case and producing the target function value based on its neighbors. Popular algorithms include: K-nearest neighbors, case-based reasoning, and locally weighted regression.

Because a target function in inductive logic programming is defined by a set of (propositional or first-order) rules, it is highly amenable to human readability and interpretability. It lends itself to incorporation of background knowledge during learning process, and is an eager and supervised learning. The bias sanctioned by ILP includes rule accuracy, FOIL-gain, or preference of shorter clauses. There are a number of algorithms: SCA, FOIL, PROGOL, and inverted resolution.

Instead of learning a non-linear target function from data in the input space directly, support vector machines use a kernel function (defined in the form of inner product of training data) to transform the training data from the input space into a high dimensional feature space F first, and then learn the optimal linear separator (a hyperplane) in F. A decision function, defined based on the linear separator, can be used to classify unseen cases. Kernel functions play a pivotal role in support vector machines. A kernel function relies only on a subset of the training data called support vectors.
In ensemble learning, a target function is essentially the result of combining, through weighted or unweighted voting, a set of component or base-level functions called an ensemble. An ensemble can have a better predictive accuracy than its component function if (1) individual functions disagree with each other, (2) individual functions have a predictive accuracy that is slightly better than random classification (e.g., error rates below 0.5 for binary classification), and (3) individual functions’ errors are at least somewhat uncorrelated. Ensemble learning can be seen as a learning strategy that addresses inadequacies in training data (insufficient information in training data to help select a single best \( h \in H \)), in search algorithms (deployment of multiple hypotheses amounts to compensating for less than perfect search algorithms), and in the representation of \( H \) (weighted combination of individual functions makes it possible to represent a true function \( f \not \in H \)). Ultimately, an ensemble is less likely to misclassify than just a single component function.

Two main issues exist in ensemble learning: ensemble construction and classification combination. There are bagging, cross-validation, and boosting methods for constructing ensembles, and weighted vote and unweighted vote for combining classifications. The AdaBoost algorithm is one of the best methods for constructing ensembles of decision trees.

There are two approaches to ensemble construction. One is to combine component functions that are homogeneous (derived using the same learning algorithm and being defined in the same representation formalism, for example, an ensemble of functions derived by decision tree method) and weak (slightly better than random guessing). Another approach is to combine component functions that are heterogeneous (derived by different learning algorithms and being represented in different formalisms, for example, an ensemble of functions derived by decision trees, instance-based learning, Bayesian learning, and neural networks) and strong (each of the component functions performs relatively well in its own right).

Multiple instance learning deals with the situation in which each training example may have several variant instances. If we use a bag to indicate the set of all variant instances for a training example, then for a Boolean class the label for the bag is positive if there is at least one variant instance in the bag that has a positive label. A bag has a negative label if all variant instances in the bag have a negative label. The learning algorithm is to approximate a target function that can classify every variant instance of an unseen negative example as negative, and at least one variant instance of an unseen positive example as positive.

In unsupervised learning, a learner is to analyze a set of objects that do not have their class labels, and discern the categories to which objects belong. Given a set of objects as input, there are two groups of approaches in unsupervised learning: density estimation methods that can be used in creating statistical models to capture or explain underlying patterns or interesting structures behind the input, and feature extraction methods that can be used to glean statistical features (regularities or irregularities) directly from the input. Unlike supervised learning, there is no direct measure of success for unsupervised learning. In general, it is difficult to establish the validity of inferences from the output unsupervised learning algorithms produce. Most frequently utilized methods under unsupervised learning include: association rules, cluster analysis, self-organizing maps, and principal component analysis.

Semi-supervised learning relies on a collection of labeled and unlabeled examples. The learning starts with using the labeled examples to obtain an initial target function, which is then used to classify the unlabeled examples, thus generating additional labeled examples. The learning process will be iterated on the augmented training set. Some semi-supervised learning methods include: expectation-maximization with generative mixture models, self-training, co-training, transductive support vector machines, and graph-based methods.
When a learner has some level of control over which part of the input domain it relies on in generating a target function, this is referred to as active learning. The control the learner possesses over the input example selection is called selective sampling. Active learning can be adopted in the following setting in semi-supervised learning: the learner identifies the most informative unlabeled examples and asks the user to label them. This combination of active learning and semi-supervised learning results in what is referred to as the multi-view learning.

Analytical learning allows a target function to be generalized from a domain theory (prior knowledge about the problem domain). The learned function has a good readability and interpretability. In analytical learning, search is performed in the form of deductive reasoning. The search bias in explanation based learning, a major analytical learning method, is a domain theory and preference of a small set of Horn clauses. One important perspective of explanation based learning is that learning can be construed as recompiling or reformulating the knowledge in the domain theory so as to make it operationally more efficient when classifying unseen cases. EBL algorithms include Prolog-EBG.

Both inductive learning and analytical learning have their props and cons. The former requires plentiful data (thus vulnerable to data quality and quantity problems), while the latter relies on a domain theory (hence susceptible to domain theory quality and quantity problems). Inductive analytical learning is meant to provide a framework where benefits from both approaches can be strengthened and impact of drawbacks minimized. It usually encompasses an inductive learning component and an analytical learning component. It requires both a training set and a domain theory, and can be an eager and supervised learning. The issues of target function representation, search, and bias are largely determined by the underlying learning components involved.

Reinforcement learning is the most general form of learning. It tackles the issue of how to learn a sequence of actions called a control strategy from indirect and delayed reward information (reinforcement). It is an eager and unsupervised learning. Its search is carried out through training episodes. Two main approaches exist for reinforcement learning: model-based and model-free approaches. The best-known model-free algorithm is Q-learning. In Q-learning, actions with maximum Q value are preferred.

Machine Learning Applications in Software Engineering

In software engineering, there are three categories of entities: processes, products and resources. Processes are collections of software related activities, such as constructing specification, detailed design, or testing. Products refer to artifacts, deliverables, documents that result from a process activity, such as a specification document, a design document, or a segment of code. Resources are entities required by a process activity, such as personnel, software tools, or hardware. The aforementioned entities have internal and external attributes. Internal attributes describe an entity itself, whereas external attributes characterize the behavior of an entity (how the entity relates to its environment). Machine learning methods have been utilized to develop better software products, to be part of software products, and to make software development process more efficient and effective. The following is a
partial list of software engineering areas where machine learning applications have found their way into:

- Predicting or estimating measurements for either internal or external attributes of processes, products, or resources. These include: software quality, software size, software development cost, project or software effort, maintenance task effort, software resource, correction cost, software reliability, software defect, reusability, software release timing, productivity, execution times, and testability of program modules.

- Discovering either internal or external properties of processes, products, or resources. These include: loop invariants, objects in programs, boundary of normal operations, equivalent mutants, process models, and aspects in aspect-oriented programming.

- Transforming products to accomplish some desirable or improved external attributes. These include: transforming serial programs to parallel ones, improving software modularity, and Mapping OO applications to heterogeneous distributed environments.

- Synthesizing or generating various products. These include: test data, test resource, project management rules, software agents, design repair knowledge, design schemas, data structures, programs/scripts, project management schedule, and information graphics.

- Reusing products or processes. These include: similarity computing, active browsing, cost of rework, knowledge representation, locating and adopting software to specifications, generalizing program abstractions, and clustering of components.

- Enhancing processes. These include: deriving specifications of system goals and requirements, extracting specifications from software, acquiring knowledge for specification refinement and augmentation, and acquiring and maintaining specification consistent with scenarios.

- Managing products. These include: collecting and managing software development knowledge, and maintaining software process knowledge.

**Organization of the Book**

This book includes sixteen chapters that are organized into five sections. The first section has three chapters (Chapters I-III) that deal with analysis, characterization, and refinement of software engineering data in terms of machine learning methods. The second section includes three chapters (Chapters IV-VI) that present applications of several machine learning approaches in helping with software systems development and deployment. The third section contains four chapters (Chapters VII-X) that describe the use of machine learning methods to establish predictive models for software quality and relevancy. Two chapters (Chapters XI-XII) in the fourth section offer some state-of-the-practice on the applications of two machine learning methods. Finally, the four chapters (Chapters XIII-XVI) in the last section of the book serve as areas of future work in this emerging research field.

Chapter I discusses the issue of how to use machine learning methods to refine a large software project database into a new database which captures and retains the essence of the
original database, but contains a fewer number of attributes and instances. This new and smaller database would afford the project managers a better chance to gain insight into the database. The proposed data refinement approach is based on the decision tree learning. Authors demonstrate their approach through four datasets in the International Software Benchmarking Standard Group database.

Chapter II is concerned with analyzing software maintenance data to shed light on efforts in defect elimination. Several learning methods (decision tree learning, rule-based learning and genetic algorithm and genetic programming) are utilized to address the following two issues: the number of software components to be examined to remove a single defect, and the total time needed to remove a defect. The maintenance data from a real life software project have been used in the study.

Chapter III takes a closer look at the credibility issue in the empirical-based models. Several experiments have been conducted on five NASA defect datasets using naïve Bayesian classifier and decision tree learning. Several observations have been made: the importance of sampling on non-class attributes, and insufficiency of the ten-fold cross validation in establishing realistic models. The author introduces several credibility metrics that measure the difficulty of a dataset. It is argued that adoption of these credibility metrics will lead to better models and improve their chance of being accepted by software practitioners.

Chapter IV focuses on the applications of inductive logic programming to software engineering. An integrated framework based on inductive logic programming has been proposed for the synthesis, maintenance, reuse, testing and debugging of logic programs. In addition, inductive logic programming has been successfully utilized in genetics, automation of the scientific process, natural language processing and data mining.

Chapter V demonstrates how multiple instance learning and neural networks are integrated with Markov model mediator to address the following challenges in an advanced content-based image retrieval system: the significant discrepancy between the low-level image features and the high-level semantic concepts, and the perception subjectivity problem. Comparative studies on a large set of real-world images indicate the promising performance of this approach.

Chapter VI describes the application of genetic algorithms to reconfigurable service oriented systems. To accommodate reconfigurability in a service-oriented architecture, QoS analysis is often required to make appropriate service selections and configurations. To determine the best selections and configurations, some composition analysis techniques are needed to analyze QoS tradeoffs. The composition analysis framework proposed in this chapter employs a genetic algorithm for composition decision making. A case study is conducted on the selections and configurations of web services.

Chapter VII deals with the issue of software quality models. Authors propose an approach to define logic-driven models based on fuzzy multiplexers. The constructs in such models have a clear and modular topology whose interpretation corresponds to a collection of straightforward logic expressions. Genetic algorithms and genetic optimization underpin the design of the logic models. Experiments on some software dataset illustrate how the logic model allows the number of modifications made to software modules to be obtained from a collection of software metrics.

Chapter VIII defines a notion called relevance relation among software entities. Relevance relations map tuples of software entities to values that signify how related they are to each other. The availability of such relevance relations plays a pivotal role in software development
and maintenance, making it possible to predict whether a change to one software entity (one file) results in a change in another entity (file). A process has been developed that allows relevance relations to be learned through decision tree learning. The empirical evaluation, through applying the process to a large legacy system, indicates that the predictive quality of the learned models makes them a viable choice for field deployment.

Chapter IX presents a novel software quality classification model that is based on genetic programming. The proposed model provides not only a classification but also a quality-based ranking for software modules. In evolving a genetic programming based software quality model, three performance criteria have been considered: classification accuracy, module ranking, and the size of the tree. The model has been subjected to case studies of software measurement data from two industrial software systems.

Chapter X describes a software quality prediction model that is used to predict fault prone modules. The model is based on an ensemble of trees voting on prediction decisions to improve its classification accuracy. Five NASA defect datasets have been used to assess the performance of the proposed model. Two strategies have been identified to be effective in the prediction accuracy: proper sampling technique in constructing the tree classifiers, and the threshold adjustment in determining the resulting class.

Chapter XI offers a broad view of the roles rule-based learning plays in software engineering. It provides some background information, discusses the key issues in rule induction, and examines how rule induction handles uncertainties in data. The chapter examines the rule induction applications in the following areas: software effort and cost prediction, software quality prediction, software defect prediction, software intrusion detection, and software process modeling.

Chapter XII, on the other hand, provides a state-of-the-practice overview on genetic algorithm applications to software testing. The focus of the chapter is on evolutionary testing, which is the application of genetic algorithms for test data generation. The central issue in evolutionary testing is a numeric representation of the test objective from which an appropriate fitness function can be defined to evaluate the generated test data. The chapter includes reviews of existing approaches in structural, temporal performance, and specification-based functional evolutionary testing.

Chapter XIII reviews two well-known formal methods, high-level Petri nets and temporal logic, for software system specification and analysis. It pays attention to recent advances in using these formal methods to specify, model and analyze software architectural design. The chapter opens the opportunity for machine learning methods to be utilized in learning either the property specifications or behavior models at element or composition level in a software architectural design phase. In addition, learning methods can be applied to the formal analysis for element correctness, or composition correctness, or refinement correctness.

A model-driven software engineering process advocates developing software systems by creating an executable model of the system design first and then transforming the model into a production quality implementation. The success of the approach hinges critically on the availability of code generators that can transform a model to its implementation. Chapter XIV gives a testimony to the model-driven process. It provides insights, practical considerations, and lessons learned when developing code generators for applications that must conform to the constraints imposed by real-world high-performance systems. Since the model can be construed as the domain theory, analytical learning can be used to help
with the transformation process. There have been machine learning applications in program transformation tasks.

Chapter XV outlines a distributed proactive semantic software engineering environment. The proposed environment incorporates logic rules into a software development process to capture the semantics from various levels of the software life cycle. The chapter discusses several scenarios in which semantic rules are used for workflow control, design consistency checking, testing and maintenance. This environment certainly makes it possible to deploy machine learning methods in the rule generator and in the semantic constraint generator to learn constraint rules and proactive rules.

Chapter XVI depicts a role-based access control model that is augmented with the context constraints for computer security policy. There are system contexts and application contexts. Integrating the contextual information into a role-based access control model allows the model to be flexible and capable of specifying various complex access policies, and to be able to provide tight and just-in-time permission activations. Machine learning methods can be used in deriving context constraints from system or application contextual data.

This book is intended particularly for practicing software engineers, and researchers and scientists in either software engineering or machine learning field. The book can also be used either as a textbook for advanced undergraduate or graduate students in a software engineering course, a machine learning application course, or as a reference book for advanced training courses in the field.

Du Zhang
Jeffrey J.P. Tsai