Machine Learning

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331

1 Introduction

Machine Learning is the study of methods for programming computers to learn. Computers are applied to a wide range of tasks, and for most of these it is relatively easy for programmers to design and implement the necessary software. However, there are many tasks for which this is difficult or impossible. These can be divided into four general categories.

First, there are problems for which there exist no human experts. For example, in modern automated manufacturing facilities, there is a need to predict machine failures before they occur by analyzing sensor readings. Because the machines are new, there are no human experts who can be interviewed by a programmer to provide the knowledge necessary to build a computer system. A machine learning system can study recorded data and subsequent machine failures and learn prediction rules.

Second, there are problems where human experts exist, but where they are unable to explain their expertise. This is the case in many perceptual tasks, such as speech recognition, hand-writing recognition, and natural language understanding. Virtually all humans exhibit expert-level abilities on these tasks, but none of them can describe the detailed steps that they follow as they perform them. Fortunately, humans can provide machines with examples of the inputs and correct outputs for these tasks, so machine learning algorithms can learn to map the inputs to the outputs.

Third, there are problems where phenomena are changing rapidly. In finance, for example, people would like to predict the future behavior of the stock market, of consumer purchases, or of exchange rates. These behaviors change frequently, so that even if a programmer could construct a good predictive computer program, it would need to be rewritten frequently. A learning program can relieve the programmer of this burden by constantly modifying and tuning a set of learned prediction rules.

Fourth, there are applications that need to be customized for each computer user separately. Consider, for example, a program to filter unwanted electronic mail messages. Different users will need different filters. It is unreasonable to expect each user to program his or her own rules, and it is infeasible to provide every user with a software engineer to keep the rules up-to-date. A machine learning system can learn which mail messages the user rejects and maintain the filtering rules automatically.

Machine learning addresses many of the same research questions as the fields of statistics, data mining, and psychology, but with differences of emphasis. Statistics focuses on understanding the phenomena that have generated the data, often with the goal of testing different hypotheses about those phenomena. Data mining seeks to find patterns in the data that are understandable by people. Psychological studies of human learning aspire to understand the mechanisms underlying the various learning behaviors exhibited by people (concept learning, skill acquisition, strategy change, etc.).

In contrast, machine learning is primarily concerned with the accuracy and effectiveness of the resulting computer system. To illustrate this, consider the different questions that might be asked about speech data. A machine learning approach focuses on building an accurate and efficient speech recognition system. A statistician might collaborate with a psychologist to test hypotheses about the mechanisms underlying speech recognition. A data mining approach might look for patterns in speech data that could be applied to group speakers according to age, sex, or level of education.

2 Analytical and Empirical Learning Tasks

Learning tasks can be classified along many different dimensions. One important dimension is the distinction between empirical and analytical learning. Empirical learning is learning that relies on some form of external experience, while analytical learning requires no external inputs. Consider, for example, the problem of learning to play tic-tac-toe (noughts and crosses). Suppose a programmer has provided an encoding of the rules for the game in the form of a function that indicates whether proposed moves are legal or illegal and another function that indicates whether the game is won, lost, or tied. Given these two functions, it is easy to write a computer program that repeatedly plays games of tic-tac-toe against itself. Suppose that this program remembers every board position that it encounters. For every final board position (i.e., where the game is won, lost, or tied), it remembers the outcome. As it plays many games, it can mark a board position as a losing position if every move made from that position leads to a winning position for the opponent. Similarly, it can mark a board position as a winning position if there exists a move from that position that leads to a losing position for the opponent. If it plays enough games, it can eventually determine all of the winning and losing positions and play perfect tic-tac-toe. This is a form of analytical learning because no external input is needed. The program is able to improve its performance just by analyzing the problem.

In contrast, consider a program that must learn the *rules* for tic-tac-toe. It generates possible moves and a teacher indicates which of them are legal and which are illegal as well as which positions are won, lost, or tied. The program can remember this experience. After it has visited every possible position and tried every possible move, it will have complete knowledge of the rules of the game (although it may guess them long before that point). This is empirical learning, because the program could not infer the rules of the game analytically—it must interact with a teacher to learn them.

The dividing line between empirical and analytical learning can be blurred. Consider a program like the first one that knows the rules of the game. However, instead of playing against itself, it plays against a human opponent. It still remembers all of the positions it has ever visited, and it still marks them as won, lost, or tied based on its knowledge of the rules. This program is likely to play better tic-tac-toe sooner, because the board positions that it visits will be ones that arise when playing against a knowledgeable player (rather random positions encountered while playing against itself). So *during the learning process*, this program will perform better. Nonetheless, the program didn't *require* the external input, because it could have inferred everything analytically.

The solution to this puzzle is to consider that the overall *learning task* is an analytical task, but that the program solves the task empirically. Furthermore, the task of playing well against a human opponent *during the learning process* is an empirical learning task, because the program needs to know which game positions are likely to arise in human games.

This may not seem like a significant issue with tic-tac-toe. But in chess, for example, it makes a huge difference. Given the rules of the game, learning to play optimal chess is an analytical learning task, but the analysis is computationally infeasible, so methods that include some empirical component must be employed instead. From a cognitive science perspective, the difference is also important. People frequently confront learning tasks which could be solved analytically, but they cannot (or choose not to) solve them this way. Instead, they rely on empirical methods.

The remainder of this article is divided into five parts. The first four parts are devoted to empirical learning. First we discuss the fundamental questions and methods in supervised learning. Then we consider more complex supervised learning problems involving sequential and spatial data. The third section is devoted to unsupervised learning problems, and the fourth section discusses reinforcement learning for sequential decision making. The article concludes with a review of methods for analytical learning.

3 Fundamentals of Supervised Learning

Let us begin by considering the simplest machine learning task: *supervised learning* for classification. Suppose we wish to develop a computer program that, when given a picture of a person, can determine whether the person is male or female. Such a program is called a *classifier*, because it assigns a class (i.e., male or female) to an object (i.e., a photograph). The task of supervised learning is to construct a classifier given a set of classified training examples—in this case, example photographs along with the correct classes.

The key challenge for supervised learning is the problem of *generalization*: After analyzing only a (usually small) sample of photographs, the learning system should output a classifier that works well on all possible photographs.

A pair consisting of an object and its associated class is called a *labeled example*. The set of labeled examples provided to the learning algorithm is called the *training set*. Suppose we provide a training set to a learning algorithm and it outputs a classifier. How can we evaluate the quality of this classifier? The usual approach is to employ a second set of labeled examples called the *test set*. We measure the percentage of test examples correctly classified (called the *classification rate*) or the percentage of test examples misclassified (the *misclassification rate*).

The reason we employ a separate test set is that most learned classifiers will be very accurate on the training examples. Indeed, a classifier that simply memorized the training examples would be able to classify them perfectly. We want to test the ability of the learned classifier to generalize to new data points.

Note that this approach of measuring the classification rate assumes that each classification decision is independent and that each classification decision is equally important. These assumptions are often violated.

The independence assumption could be violated if there is some temporal dependence in the data. Suppose for example, that the photographs were taken of students in classrooms. Some classes (e.g., early childhood development) primarily contain girls, other classes (e.g., car repair) primarily contain boys. If a classifier knew that the data consisted of batches, it could achieve higher accuracy by trying to identify the point at which one batch ends and another begins. Then within each batch of photographs, it could classify all of the objects into a single class (e.g., based on a majority vote of its guesses on the individual photographs). These kinds of temporal dependencies arise frequently. For example, a doctor seeing patients in a clinic knows that contagious illnesses tend to come in waves. Hence, after seeing several consecutive patients with the flu, the doctor is more likely to classify the next patient as having the flu too, even if that patient's symptoms are not as clearcut as the symptoms of the previous patients.

The assumption of equal importance could be violated if there are different costs or risks associated with different misclassification errors. Suppose the classifier must decide whether a patient has cancer based on some laboratory measurements. There are two kinds of errors. A *false positive* error occurs when the classifier classifies a healthy patient as having cancer. A *false negative* error occurs when the classifier classifies a person with cancer as being healthy. Typically false negatives are more costly than false positives, so we might want the learning algorithm to prefer classifiers that make fewer false negative errors, even if they make more false positives as a result.

The term supervised learning includes not only learning classifiers but also learning functions that predict numerical values. For example, given a photograph of a person, we might want to predict the person's age, height, and weight. This task is usually called *regression*. In this case, each labeled training example is a pair of an object and the associated numerical value. The quality of a learned prediction function is usually measured as the square of the difference between the predicted value and the true value, although sometimes the absolute value of this difference is measured instead.

3.1 An Example Learning Algorithm: Learning Decision Trees

There are many different learning algorithms that have been developed for supervised classification and regression. These can be grouped according to the formalism they employ for representing the learned classifier or predictor: decision trees, decision rules, neural networks, linear discriminant functions, Bayesian networks, support vector machines, and nearest-neighbor methods. Many of these algorithms are described in other articles in this encyclopedia. Here, we will present a top-down algorithm for learning decision trees, since this is one of the most versatile, most efficient, and most popular machine learning algorithms.

A decision tree is a branching structure as shown in Figure 1. The tree consists of nodes and leaves. The *root node* is at the top of the diagram, and the leaves at the bottom. Each node tests the value of some *feature* of an example, and each leaf assigns a class label to the example. This tree was constructed by analyzing 670 labeled examples of breast cancer biopsies. Each biopsy is represented by 9 features such as Clump Thickness (CT), Uniformity of Cell Size (US), and Uniformity of Cell Shape (USh). To understand how the decision tree works, suppose we have a biopsy example with $US = 5, CT = 7$, and $BN = 2$. To classify this example, the decision tree first tests if $US > 3$, which is true. Whenever the test in a node is true, control follows the left outgoing arrow; otherwise, it follows the right outgoing arrow. In this case, the next test is $CT \leq 6$ which is false, so control follows the right arrow to the test $BN \leq 2$. This is true, so control follows the left arrow to a leaf node which assigns the class "Benign" to the biopsy.

The numbers in each node indicate the number of Malignant and Benign training examples that "reached" that node during the learning process. At the root, the 670 training examples comprised 236 Malignant cases and 434 Benign cases. The decision tree is constructed top-down by repeatedly choosing a feature (e.g., US) and a threshold (e.g., 3) to test. Different algorithms employ different heuristics, but all of these heuristics try to find the feature and threshold that are most predictive of the class label. A perfect test would send all of the Benign examples to one branch and all of the Malignant examples to the other branch. The test $US > 3$ is not perfect, but it is still very good: the left branch receives 410 of the 434 Benign cases and only 45 of the 236 Malignant ones, while the right branch receives 191 of the 236 Malignant cases and only 24 of the Benign ones. After selecting this test, the algorithm splits the training examples according to the test. This gives it 45 + 410 = 455 examples on the left branch and 191 + 24 = 215 on the right. It now repeats the same process of choosing a predictive feature and threshold and splitting the data until a termination rule halts the splitting process. At that point, a leaf is created whose class label is the label of the majority of the training examples that reached the leaf.

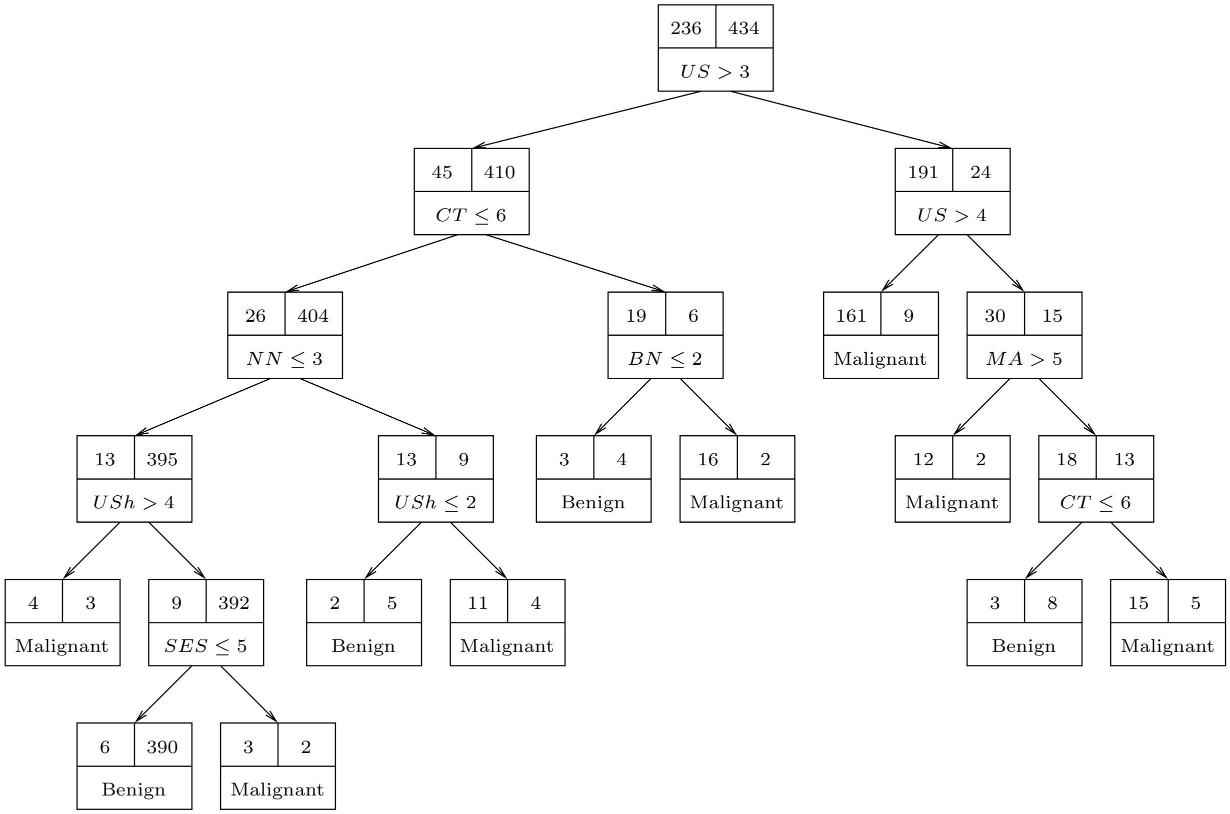

Figure 1: Decision Tree for Diagnosing Breast Cancer. US=Uniformity of Cell Size; CT=Clump Thickness; NN=Normal Nucleoli; USh=Uniformity of Cell Shape; ES=Single Epithelial Cell Size; BN =Bare Nuclei; MA=Marginal Adhesion

One advantage of decision trees is that, if they are not too large, they can be interpreted by humans. This can be useful both for gaining insight into the data and also for validating the reasonableness of the learned tree.

3.2 The Triple Tradeoff in Empirical Learning

All empirical learning algorithms must contend with a tradeoff among three factors: (a) the size or complexity of the learned classifier, (b) the amount of training data, and (c) the generalization accuracy on new examples. Specifically, the generalization accuracy on new examples will usually increase as the amount of training data increases. As the complexity of the learned classifier increases, the generalization accuracy first rises and then falls. These tradeoffs are illustrated in Figure 2. The different curves correspond to different amounts of training data. As more data is available, the generalization accuracy reaches a higher level before eventually dropping. In addition, this higher level corresponds to increasingly more complex classifiers.

The relationship between generalization accuracy and the amount of training data is fairly intuitive: The more training data given to the learning algorithm, the more evidence the algorithm has about the classification problem. In the limit, the data would contain every possible example, so the algorithm would know the correct label of every possible example, and it would generalize perfectly.

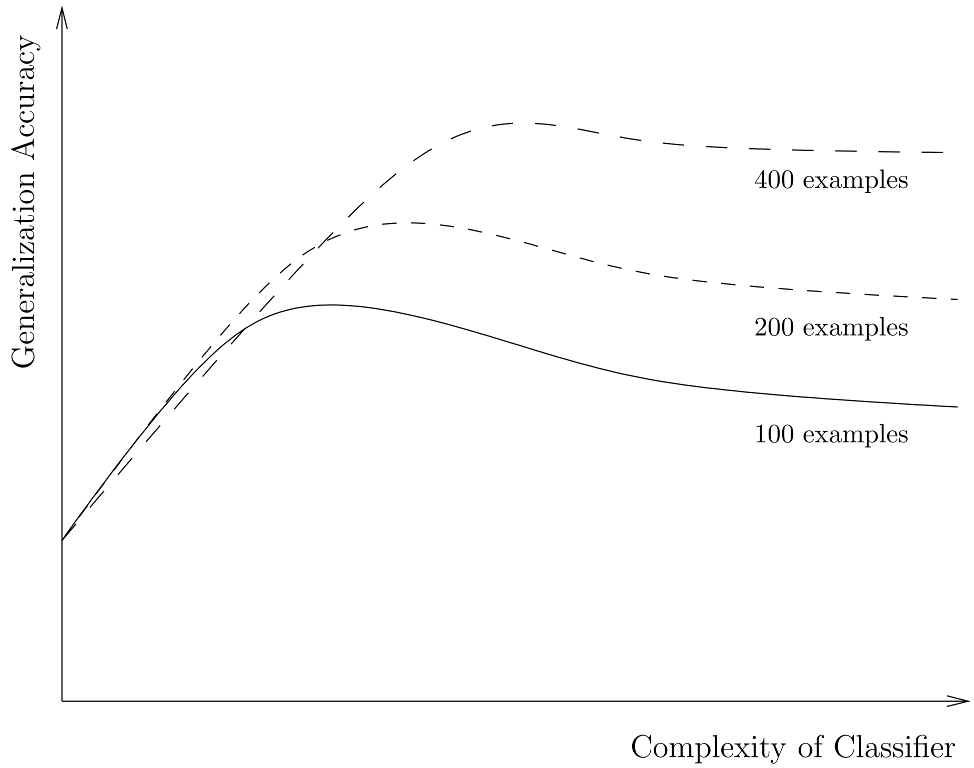

Figure 2: Generalization accuracy as a function of the complexity of the classifier, for various amounts of training data.

In contrast, the relationship between generalization accuracy and the complexity of the learned structure is less obvious. To understand it, consider what would happen if we allowed the decision tree to grow extremely large. An advantage of such a large tree is that it can be proved that if the tree becomes large enough, it can represent *any* classifier. We say that such trees have *low bias*.

Unfortunately, however, such a very large tree would typically end up having only a few training examples in each leaf. As a result, the choice of the class label in each leaf would be based on just those few examples, which is a precarious situation. If there is any noise in the process of measuring and labeling those training examples, then the class label could be in error. The resulting classifier is said to have *high variance*, because a slight change in the training examples can lead to changes in the classification decisions. Such a decision tree has merely memorized the training data, and although it will be very accurate on the training data, it will usually generalize very poorly. We say that it has “overfit” the training data.

At the other extreme, suppose we consider a degenerate decision tree that contains only one decision node and two leaves. (These are known as “decision stumps”). The class label in each leaf is now based on hundreds of training examples, so the tree has *low variance*, because it would take a large change in the training data to cause a change in the classifier. However, such a simple decision tree might not be able to capture the full complexity of the data. For diagnosing breast cancer, for example, it is probably not sufficient to consider only one feature. Formally, such a classifier is said to have *high bias*, because its representational structure prevents it from representing the optimal classifier. Consequently, the classifier may also generalize poorly, and we say that it has “underfit” the training data.

An intuitive way of thinking about this tradeoff between bias and variance is the following. A learning algorithm faces a choice between a vast number of possible classifiers. When very little data is available, it does not have enough information to distinguish between all of these classifiers—many classifiers will appear to have identical accuracy on the training data and if it chooses randomly among such apparently good classifiers, this will result in high variance. It must reduce the number of possible classifiers (i.e., by reducing their complexity) until it does have enough data to discriminate among them. Unfortunately, this reduction will probably introduce bias, but it will reduce the variance.

In virtually every empirical learning algorithm, there are mechanisms that seek to match the complexity of the classifier to the complexity of the training data. In decision trees, for example, there are *pruning procedures* that remove branches from an overly large tree to reduce the risk of overfitting. In neural networks, support vector machines, and linear discriminant functions, there are *regularization methods* that place a numerical penalty on having large numerical weights. This turns out to be mathematically equivalent to limiting the complexity of the resulting classifiers.

Some learning algorithms, such as the naive Bayes method and the perceptron algorithm are not able to adapt the complexity of the classifier. These algorithms only consider relatively simple classifiers. As a result, on small training sets, they tend to perform fairly well, but as the amount of training data increases, their performance suffers, because they underfit the data (i.e., they are biased).

3.3 Prior Knowledge and Bias

Most machine learning algorithms make only very general and very weak assumptions about the nature of the training data. As a result, they typically require large amounts of training data to learn accurate classifiers. This problem can be solved by exploiting prior knowledge to eliminate from consideration classifiers that are not consistent with the prior knowledge. The resulting learning algorithms may be able to learn from very few training examples.

However, there is a risk to introducing prior knowledge. If that knowledge is incorrect, then it will eliminate all of the accurate classifiers from consideration by the learning algorithm. In short, prior knowledge introduces bias into the learning process, and it is important that this bias be correct.

4 Supervised Learning for Sequences, Time Series, and Spatial Data

Now that we have discussed the basic supervised learning problem and the bias-variance tradeoff, we now turn our attention to more complex supervised learning tasks.

Consider the problem of speech recognition. A speech recognition system typically accepts as input a spoken sentence (e.g., 5 seconds of a sound signal) and produces as output the corresponding string of words. This involves many levels of processing, but at the lowest level, we can think of a sentence as a sequence of labeled examples. Each example consists of a 40 ms segment of speech (the object) along with a corresponding phoneme (the label). However, it would be a mistake to assume that these labeled examples are independent of each other, because there are strong sequential patterns relating adjacent phonemes. For example, the pair of phonemes /s/ /p/ (as in the English words "spill" and "spin") is much more common than the pair /s/ /b/ (which almost never appears). Hence, a speech recognition system has the opportunity to learn not only how to relate the speech signal to the phonemes, but also how to relate the phonemes to each other. The

Hidden Markov Model (see SPEECH article) is an example of a classifier that can learn both of these kinds of information.

A similar problem arises in time-series analysis. Suppose we wish to predict the El Niño phenomenon, which can be measured by the temperature of the sea surface in the equatorial Pacific Ocean. Imagine that we have measurements of the temperature every month for the past 20 years. We can view this as a set of labeled training examples. Each example is a pair of temperatures from two consecutive months, and the goal is to learn a function for predicting the temperature next month from the temperature in the current month. Again it is a mistake to treat these examples as independent. The relationship between adjacent months is similar to the relationship between adjacent phonemes in speech recognition. However, unlike in speech recognition, we must make a prediction every month about what the next month's temperature will be. This would be like trying to predict the next word in the sentence based on the previous word.

Spatial data present learning tasks similar to sequential data, but in two dimensions. For example, a typical spatial task is to predict the type of land cover (trees, grasslands, lakes, etc.) on the ground based on satellite photographs. Training data consist of photographs in which each pixel has been labeled by its land cover type. Methods such as Markov Random Fields can be applied to capture the relationships between nearby pixels.

4.1 Supervised Learning for Complex Objects

So far we have discussed the task of classifying single objects and the task of classifying a one- or two-dimensional array of objects. There is a third task that is intermediate between these: the task of classifying complex objects. For example, consider the problem of deciding whether a credit card has been stolen. The "object" in this case is a *sequence* of credit card transactions, but the class label (stolen or not stolen) is attached to the entire sequence, not to each individual transaction. In this case, we wish to analyze the entire sequence to decide whether it provides evidence that the card is stolen.

There are three ways to approach this problem. The first method converts it into a simple supervised learning problem by extracting a set of features from the sequence. For example, we might compute the average, minimum, and maximum dollar amounts of the transactions, the variance of the transactions, the number of transactions per day, the geographical distribution of the transactions, and so on. These features summarize the variable-length sequence as a fixed length feature vector, which we can then give as input to a standard supervised learning algorithm.

The second method is to convert the problem into the problem of classifying labeled sequences of objects. On the training data, we assign a label to each transaction indicating whether it was legitimate or not. Then we train a classifier for classifying individual transactions. Finally, to decide whether a new sequence of transactions indicates fraud, we apply our learned classifier to the entire sequence and then make a decision based on the number of fraudulent transactions it identifies.

The third method is to learn explicit models of fraudulent and non-fraudulent sequences. For example, we might learn a hidden Markov model that describes the fraudulent training sequences and another HMM to describe the non-fraudulent sequences. To classify a new sequence, we compute the likelihood that each of these two models could have generated the new sequence and choose the class label of the more likely model.

5 Unsupervised Learning

The term unsupervised learning is employed to describe a wide range of different learning tasks. As the name implies, these tasks analyze a given set of objects that do not have attached class labels. In this section, we will describe five unsupervised learning tasks.

5.1 Understanding and Visualization

Given a large collection of objects, we often want to be able to understand these objects and visualize their relationships. Consider, for example, the vast diversity of living things on earth. Linnaeus devoted much of his life to arranging living organisms into a hierarchy of classes with the goal of arranging similar organisms together at all levels of the hierarchy.

Many unsupervised learning algorithms create similar hierarchical arrangements. The task of *hierarchical clustering* is to arrange a set of objects into a hierarchy so that similar objects are grouped together. A standard approach is to define a measure of the similarity between any two objects and then seek clusters of objects which are more similar to each other than they are to the objects in other clusters. *Non-hierarchical clustering* seeks to partition the data into some number of disjoint clusters.

A second approach to understanding and visualizing data is to arrange the objects in a low-dimensional space (e.g., in a 2-dimensional plane) so that similar objects are located nearby each other. Suppose, for example, that the objects are represented by 5 real-valued attributes: height, width, length, weight, color, and density. We can measure the similarity of any two objects by their Euclidean distance in this 5-dimensional space. We wish to assign each object two new dimensions (call them x and y) such that the Euclidean distance between the objects in this 2-dimensional space is proportional to their Euclidean distance in the original 5-dimensional space. We can then plot each object as a point in the 2-dimensional plane and visually see which objects are similar.

5.2 Density Estimation and Anomaly Detection

A second unsupervised learning task is density estimation (and the closely-related task of anomaly detection). Given a set of objects, $\{e_1, e_2, \ldots, e_n\}$, we can imagine that these objects constitute a random sample from some underlying probability distribution $P(e)$. The task of density estimation is to learn the definition of this probability density function P.

A common application of density estimation is to identify anomalies or outliers. These are objects that do not belong to the underlying probability density. For example, one approach to detecting fraudulent credit card transactions is to collect a sample of legal credit card transactions and learn a probability density $P(t)$ for the probability of transaction t. Then, given a new transaction t', if $P(t')$ is very small, this indicates that t' is unusual and should be brought to the attention of the fraud department. In manufacturing, one quality control procedure is to raise an alarm whenever an anomalous object is produced by the manufacturing process.

5.3 Object Completion

People have an amazing ability to complete a fragmentary description of an object or situation. For example, in natural language understanding, if we read the sentence, "Fred went to the market. He found some milk on the shelf, paid for it, and left," we can fill in many events that were not mentioned. For example, we are quite confident that Fred picked up the milk from the shelf and took it to the cash register. We also believe that Fred took the milk with him when he left the market. We can complete this description because we know about "typical" shopping episodes.

Similarly, suppose we see the front bumper and wheels of a car visible around the corner of a building. We can predict very accurately what the rest of the car looks like, even though it is hidden from view.

Object completion involves predicting the missing parts of an object given a partial description of the object. Both clustering and density estimation methods can be applied to perform object completion. The partial description of the object can be used to select the most similar cluster, and then the object can be completed by analyzing the other objects in that cluster. Similarly, a learned probability density $P(x_1, x_2)$ can be used to compute the most likely values of x_2 given the observed values of x_1. A third approach to object completion is to apply a supervised learning algorithm to predict each attribute of an object given different subsets of the remaining attributes.

5.4 Information Retrieval

A fourth unsupervised learning task is to retrieve relevant objects (documents, images, finger prints) from a large collection of objects. Information retrieval systems are typically given a partial description of an object, and they use this partial description to identify the K most similar objects in the collection. In other cases, a few examples of complete objects may be given, and again the goal is to retrieve the K most similar objects.

Clustering methods can be applied to this problem. Given partial or complete descriptions of objects, the most similar cluster can be identified. Then the K most similar objects can be extracted from that cluster.

5.5 Data Compression

There are many situations in which we do not want to store or transmit fully-detailed descriptions of objects. Each image taken by a digital camera, for example, can require 3 megabytes to store. By applying image compression techniques, such images can often be reduced to 50 kilobytes (a 60-fold reduction) without noticeably degrading the picture. Data compression involves identifying and removing the irrelevant aspects of data (or equivalently, identifying and retaining the essential aspects of data). Most data compression methods work by identifying commonly-occurring subimages or substrings and storing them in a "dictionary." Each occurrence of such a substring or subimage can then be replaced by a (much shorter) reference to the corresponding dictionary entry.

6 Learning for Sequential Decision Making

In all learning systems, learning results in an improved ability to make decisions. In the supervised and unsupervised learning tasks we have discussed so far, the decisions made by the computer system after learning are non-sequential. That is, if the computer system makes a mistake on one decision, this has no bearing on subsequent decisions. Hence, if an optical character recognition system misreads the postal code on a package, this only causes that package to be sent to the wrong address. It does not have any effect on where the next package will be sent. Similarly, if a fraud detection system correctly identifies a stolen credit card for one customer, this has no effect on the cost (or benefit) of identifying the stolen credit cards of other customers.

In contrast, consider the problem of steering a car down a street. The driver must make a decision approximately once per second about how to turn the wheel to keep the car in its lane. Suppose the car is in the center of the lane but pointed slightly to the right. If the driver fails to correct by turning slightly to the left, then at the next time step, the car will move into the right part of the lane. If the driver again fails to correct, the car will start to leave the lane. In short, if the driver makes a mistake at one decision point, this affects the situation that he or she will confront at the next decision point. The hallmark of sequential decision making is that the decision-maker must live with the consequences of his or her mistakes.

Sequential decision-making tasks arise in many domains where it is necessary to control a system (e.g., steering of robots, cars, and spacecraft; control of oil refineries, chemical plants, power plants, and factories; management of patients in intensive care). The system under control is also referred to as "the environment."

Many of these control problems can be modeled as *Markov decision problems (MDPs)*. In a Markov decision problem, the environment is said to have a current "state" that completely summarizes the variables in the system (e.g., the position, velocity, and acceleration of a car). At each time, the controller observes the state of the environment and must choose an action (e.g, steer left or steer right). The action is then executed, which may cause the environment to move to a new state. The controller then receives an immediate "reward" which is a measure of the cost of the action and the desirability of the current state or the new state. For example, in steering a car, there might be a reward of zero for all actions as long as the car remains in its lane. But the controller would receive a penalty whenever the car departed from the lane. The controller makes its decisions according to a control *policy*. The policy tells, for each state of the environment, what action should be executed. The optimal policy is one that maximizes the sum of the rewards received by the controller.

Reinforcement learning is the task of learning a control policy by interacting with an *unknown* environment. There are two main approaches to reinforcement learning: model-based and model-free methods.

In model-based reinforcement learning, the learner executes a control policy for the purpose of learning about the environment. Each time it executes an action a in state s and observes the resulting reward r and next state s', it collects a four-tuple $\langle s, a, r, s' \rangle$ that records the experience. After collecting a sufficient number of these four-tuples, the learner can learn a *probability transition function* $P(s'|s, a)$ and a *reward function* $R(s, a, s')$. The probability transition function says that if action a is executed in state s, then the environment will move to state s' with probability $P(s'|s, a)$. The reward function gives the average value of the reward that will be received when this happens: $R(s, a, s')$. Given these two functions, it is possible to apply *dynamic programming* algorithms to compute the optimal control policy.

Model-free reinforcement learning algorithms learn the policy directly by interacting with the environment without storing experience four-tuples or learning a model (i.e., without learning P and R). The best-known model-free algorithm is Q-learning (see REINFORCEMENT LEARNING), but researchers have also explored actor/critic and policy gradient algorithms that directly modify the control policy to improve its performance.

In many control problems, the entire state of the environment cannot be observed at each time step. For example, when driving a car, the driver cannot simultaneously observe all of the other cars driving on the same street. Similarly, a robot controller can rarely observe the entire state of the world. The task of controlling such partially-observable environments is known as a Partially-Observable Markov Decision Problem (POMDP). As with MDPs, model-based and model-free methods have been developed.

Model-based methods for POMDPs must posit the existence of underlying (but unobservable states) in order to explain (and predict) the way that the observable parts of the state will change (see LEARNING MARKOV DECISION PROCESSES).

7 Speedup Learning

We now turn our attention to analytical learning. Because analytical learning does not involve interaction with an external source of data, analytical learning systems cannot learn knowledge with new empirical content. Instead, analytical learning focuses on improving the speed and reliability of the inferences and decisions that are performed by the computer. This is analogous in many ways to the process of skill acquisition in people.

Consider a computation that involves search. Examples include searching for good sequences of moves in chess, searching for good routes in a city, and searching for the right steps in a cooking recipe. The task of speedup learning is to remember and analyze past searches so that future problems can be solved more quickly and with little or no search.

The simplest form of speedup learning is called *caching*—replacing computation with memory. When the system performs a search, it stores the results of the search in memory. Later, it can retrieve information from memory rather than repeating the computation. For example, consider a person trying to bake a cake. There are many possible combinations of ingredients and many possible processing steps (e.g., stirring, sifting, cooking at various temperatures and for various amounts of time). A cook must search this space, trying various combinations, until a good cake is made. The cook can learn from this search by storing good combinations of ingredients and processing steps (e.g., in the form of a recipe written on a card). Then, when he or she needs to bake another cake, the recipe can be retrieved and followed.

Analogous methods have been applied to speed up computer game playing. Good sequences of moves can be found by searching through possible game situations. These sequences can then be stored and later retrieved to avoid repeating the search during future games. This search for good move sequences can be performed without playing any "real" games against opponents.

A more interesting form of speedup learning is generalized caching—also known as *explanation-based learning*. Consider a cook who now wants to bake bread. Are there processing steps that were found during the search for a good cake recipe that can be re-used for a good bread recipe? The cook may have discovered that it is important to add the flour, sugar, and cocoa powder slowly when mixing it with the water, eggs, and vanilla extract. If the cook can identify an *explanation* for this part of the recipe, then it can be generalized. In this case, the explanation is that when adding powdered ingredients (flour, sugar, cocoa) to a liquid batter (water, eggs, and vanilla extract), adding them slowly while stirring avoids creating lumps. This explanation supports the creation of a general rule: Add powdered ingredients slowly to a liquid batter while stirring.

When baking bread, the cook can retrieve this rule and apply it, but this time the powdered ingredients are flour, salt, and dry yeast, and the liquid batter is water. Note that the explanation provides the useful abstractions (powdered ingredients, liquid batter) and also the justification for the rule. Explanation-based learning is a form of analytical learning, because it relies on the availability of background knowledge that is able to explain why particular steps succeed or fail.

Retrieving a rule is usually more difficult than retrieving an entire recipe. To retrieve an entire recipe, we just need to look up the name ("chocolate cake"). But to retrieve a rule, we must identify the relevant situation ("adding powdered ingredients to liquid batter"). Sometimes, the cost of evaluating the rule conditions is greater than the time saved by not searching. This is known as the *utility problem*. One solution to the utility problem is to restrict the expressive power of rule conditions so that they are guaranteed to be cheap to evaluate. Another solution is to approximate the rule conditions with different conditions that are easier to evaluate, even if this introduces some errors. This is known as *knowledge compilation*. Explanation-based learning mechanisms have been incorporated into cognitive architectures such as the SOAR architecture of Newell, Rosenbloom, and Laird and the various ACT architectures of Anderson.

A third form of speedup learning is to learn search control heuristics. These heuristics typically take the form of evaluation functions, pruning rules, preference rules, or macros. An evaluation function assigns a numerical score to each situation (e.g., to each proposed cooking step). A pruning rule evaluates a proposed step and indicates whether it is likely to succeed. If not, the step can be pruned from the search. A preference rule compares two different proposed steps and indicates which is better. It can be applied to rank-order the proposed steps. A macro prescribes a *sequence* of steps to take when certain conditions are satisfied. All of these search control heuristics are typically learned by applying empirical learning methods.

Evaluation functions are frequently employed in game-playing programs. For example, the best computer backgammon program (Tesauro's TD-gammon) applies the empirical $TD(\lambda)$ reinforcement learning algorithm to learn an evaluation function for backgammon.

Pruning rules can be learned as follows. Suppose our cook has kept records of all of the cooking experiments along with information about which experiments succeeded and which ones failed. We can consider each step in each recipe as a training example, and assign it a label of "succeeds" or "fails" depending on whether it was a good step to execute. Then supervised learning algorithms can be applied to learn a classifier for recipe steps. When developing a new recipe, this classifier can be applied to evaluate proposed steps and indicate which ones are likely to succeed—thus avoiding search.

Preference rules can also be learned by supervised learning. One training example is constructed for each *pair* of proposed actions (e.g., proposed cooking steps). The example is labeled as "Better" if the first step is better than the second, and "Worse" if the second step is better than the first. Then a supervised learning algorithm can learn a classifier for ranking the proposed actions.

Finally, macros can be learned in many different ways. One interesting method is known as the "peak-to-peak" method. It can be applied to convert an imperfect evaluation function into a set of macros. A perfect evaluation function gives a high score to every good move from the starting state to the final goal (i.e., from the raw ingredients to the finished cake). An imperfect evaluation function will give high scores to some steps but then give poor scores to a sequence of steps before again giving high scores. We can view this as a "valley" between two peaks. The valley can be removed by introducing a macro that says when you reach the first peak, execute the *sequence* of steps that will lead to the next peak. By executing the macro as a single "macro step", we are able to move right through the valley without becoming confused by the evaluation function errors.

8 Further Reading

Anderson, J. R. (1989). A theory of the origins of human knowledge. *Artificial Intelligence, 40,* 313–352.

Bishop, C. M. (1996). *Neural Networks for Pattern Recognition.* Oxford: Oxford University Press.

Breiman, L, Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees.* Monterey, CA: Wadsworth and Brooks.

Chatfield, C. (1996). *The Analysis of Time Series: An Introduction.* Fifth Edition. Baton Rouge: Chapman and Hall/CRC.

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

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

Hastie, T., Tibshirani, R., and Friedman, J. (2001) *The Elements of Statistical Learning.* New York: Springer-Verlag.

Hand, D., Mannila, H., and Smyth, P. (2001). *Principles of Data Mining.*

Jelinek, F. (1999). *Statistical Methods for Speech Recognition.* Cambridge, MA: MIT Press.

Jordan, M. (Ed.) (1999). *Learning in Graphical Models.* Cambridge, MA: MIT Press.

Mitchell, T. M. (1997). *Machine Learning.* New York: McGraw-Hill.

Newell, A. (1994). *Unified Theories of Cognition.* Cambridge, MA: Harvard University Press.

Quinlan, J. R. (1992). *C4.5: Programs for Empirical Learning.* San Mateo, CA: Morgan Kaufmann.

Shavlik, J., and Dietterich, T. G., (1990) *Readings in Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Sutton, R., and Barto, A. G. (1998). *Introduction to Reinforcement Learning.* Cambridge, MA: MIT Press.