MODULE 1: INTRODUCTION TO MACHINE LEARNING AND CONCEPT LEARNING

References
1. Tom M. Mitchell, Machine Learning**,** India Edition 2013, McGraw Hill Education.

INTRODUCTION TO MACHINE LEARNING

1.1 INTRODUCTION

Machine learning aims on programming the computer to learn automatically by experience. **Examples includes:** Computers learning from medical records which treatments are most effective for new diseases, houses learning from experience to optimize energy costs based on the usage patterns of their occupants. A detailed understanding of information processing algorithms for machine learning might lead to a better understanding of human learning abilities.

Many practical computer programs have been developed to exhibit useful types of learning, and significant commercial applications have begun to appear. For problems such as speech recognition, algorithms based on machine learning outperform all other approaches that have been attempted to date. In the field known as data mining, machine learning algorithms are being used routinely to discover valuable knowledge from large commercial databases containing equipment maintenance records, loan applications, financial transactions, medical records, etc. A few achievements of Machine learning are as follows:

1) Learn to recognize spoken words.
2) Predict recovery rates of pneumonia patients.
3) Detect fraudulent use of credit cards.
4) Drive autonomous vehicles on public highways.

Machine learning is inherently a multidisciplinary field which draws on results from artificial intelligence, probability and statistics, computational complexity theory, control theory, information theory, philosophy, psychology, neurobiology, and other fields.

1.2 WELL-POSED LEARNING PROBLEMS

Let us begin our study of machine learning by considering a few learning tasks.

Definition: A computer program is said to learn from experience E with respect to some class of tasks ***T*** and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

For example, a computer program that learns to play checkers might improve its performance as ***measured by its ability to win*** at the class of tasks involving ***playing checkers games,*** through experience ***obtained*** by ***playing games against itself.*** In general, to have a well-

defined learning problem, we must identity these three features: the class of tasks, the measure of performance to be improved, and the source of experience.

Example 1: A checkers learning problem

- Task T: playing checkers
- Performance measure P: percent of games won against opponents
- Training experience E: playing practice games against itself

Example 2: A robot driving learning problem:

- Task T: driving on public four-lane highways using vision sensors
- Performance measure P: average distance travelled before an error (as judged by human overseer)
- Training experience E: a sequence of images and steering commands recorded while observing a human driver.

1.3 DESIGNING A LEARNING SYSTEM

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament. We adopt the obvious performance measure: the percent of games it wins in this world tournament.

1.3.1 Choosing the Training Experience

The first design choice we face is to choose the type of training experience from which our system will learn. The type of training experience available can have a significant impact on success or failure of the learner.

One key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system. For example, in learning to play checkers, the system might learn from ***direct*** training examples consisting of individual checkers board states and the correct move for each. Credit assignment can be a particularly difficult problem because the game can be lost even when early moves are optimal, if these are followed later by poor moves. Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

A second important attribute of the training experience is the degree to which the learner controls the sequence of training examples. For example, the learner might rely on the teacher to select informative board states and to provide the correct move for each. Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move. ***Or*** the learner may have complete control over

both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher present.

A third important attribute of the training experience is how well it represents the distribution of examples over which the final system performance P must be measured. In general, learning is most reliable when the training examples follow a distribution similar to that of future test examples. In our checkers learning scenario, the performance metric P is the percent of games the system wins in the world tournament. If its training experience E consists only of games played against itself, there is an obvious danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested. For example, the learner might never encounter certain crucial board states that are very likely to be played by the human checkers champion. A fully specified learning task is specified below:

A checkers learning problem:

- Task **T**: playing checkers
- Performance measure ***P:*** percent of games won in the world tournament
- Training experience **E**: games played against itself

In order to complete the design of the learning system, we must now choose

1. The exact type of knowledge to be learned
2. A representation for this target knowledge
3. A learning mechanism

1.3.2 Choosing the Target Function

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program. Let us begin with a checkers-playing program that can generate the ***legal*** moves from any board state. The program needs only to learn how to choose the ***best*** move from among these legal moves. This learning task is representative of a large class of tasks for which the legal moves that define some large search space are known a priori, but for which the best search strategy is not known.

Given this setting where we must learn to choose among the legal moves, the most obvious choice for the type of information to be learned is a program, or function, that chooses the best move for any given board state. Let us call this function ***ChooseMove*** and use the notation ***ChooseMove*** : B -> M to indicate that this function accepts as input any board from the set of legal board states *B* and produces as output some move from the set of legal moves ***M.***

Although ***ChooseMove*** is an obvious choice for the target function in our example, this function will turn out to be very difficult to learn given the kind of indirect training experience available to our system. An alternative target function and one that will turn out to be easier to learn in this setting-is an evaluation function that assigns a numerical score to any given board state. Let us call this target function ***V*** and again use the notation ***V*** : B ->*R* to denote that ***V*** maps any legal board state from the set B to some real value (we use *R* to denote the set of real numbers). We intend for this target function ***V*** to assign higher scores to better board states. If the system can successfully learn such a target function ***V,***then it can easily use it to select the best move from any current board position.

What exactly should be the value of the target function ***V*** for any given board state? Of course any evaluation function that assigns higher scores to better board states will do. Nevertheless, we will find it useful to define one particular target function ***V*** among the many that produce optimal play. As we shall see, this will make it easier to design a training algorithm. Let us therefore define the target value ***V(b)*** for an arbitrary board state ***b*** in *B, **as*** follows:

1. if ***b*** is a final board state that is won, then ***V(b)*** = 100

2. if b is a final board state that is lost, then ***V(b)*** = -100

3. if b is a final board state that is drawn, then ***V(b) = 0***

4. if b is a not a final state in the game, then V(b) = V(b'), where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game (assuming the opponent plays optimally, as well).

1.3.3 Choosing a Representation for the Target Function

To keep the discussion brief, let us choose a simple representation: for any given board state, the function will be calculated as a linear combination of the following board features:

- ***xl:*** the number of black pieces on the board
- ***x2:*** the number of red pieces on the board
- ***x3:*** the number of black kings on the board
- ***x4*:** the number of red kings on the board
- ***x5:*** the number of black pieces threatened by red (i.e., which can be captured on red's next turn)
- ***x6:*** the number of red pieces threatened by black

Thus, our learning program will represent $\hat{V}$ (b) as a linear function of the form

$$\hat{V}(b) = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + w_6x_6 \quad (1)$$

where ***wo*** through ***w6*** are numerical coefficients, or weights, to be chosen by the learning algorithm. Learned values for the weights ***w1*** through ***w6*** will determine the relative importance of the various board features in determining the value of the board, whereas the weight ***wo*** will provide an additive constant to the board value.

1.3.4 Choosing a Function Approximation Algorithm

In order to learn the target function $\hat{V}$ we require a set of training examples, each describing a specific board state b and the training value $V_{train}(b)$ for b. In other words, each training example is an ordered pair of the form (b, $V_{train}(b)$). For instance, the following training example describes a board state b in which black has won the game (note ***x2*** = 0 indicates that **red** has no remaining pieces) and for which the target function value Vtrain(b) is therefore **+100.**

$$\langle \langle x_1 = 3, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0 \rangle, +100 \rangle \quad (2)$$

1.3.4.1 Estimating Training Values

Despite the ambiguity inherent in estimating training values for intermediate board states, one simple approach has been found to be surprisingly successful. This approach is to assign the training value of **Vtrain(b)** for any intermediate board state **b** to be $\hat{V}$ **(Successor(b))**where $\hat{V}$ is the learner's current approximation to **V** and where **Successor(b)** denotes the next board state following ***b*** for which it is again the program's turn to move (i.e., the board state following the program's move and the opponent's response). This rule for estimating training values can be summarized as

Rule for estimating training values,

$$V_{train}(b) \leftarrow \hat{V}(Successor(b)) \quad (3)$$

1.3.4.2 Adjusting the Weights

All that remains is to specify the learning algorithm for choosing the weights ***wi*** to best fit the set of training examples ***{(b,V train(b))***}. As a first step we must define what we mean by the ***bestfit*** to the training data. One common approach is to define the best hypothesis, or set of weights, as that which minimizes the squared error E between the training values and the values predicted by the hypothesis $\hat{V}$

$$E \equiv \sum_{\langle b, V_{train}(b) \rangle \in \text{training examples}} (V_{train}(b) - \hat{V}(b))^2 \quad (4)$$

The **LMS** algorithm is defined as follows:

LMS weight update rule.

For each training example (b,Vtrain(b))

- Use the current weights to calculate $\hat{V}$(b)

- For each weight **wi,** update it as

$$w_i \leftarrow w_i + \eta \ (V_{train}(b) - \hat{V}(b)) \ x_i \quad (5)$$

1.3.5 The Final Design

The final design of our checkers learning system can be naturally described by four distinct program modules that represent the central components in many learning systems. These four modules, summarized in Figure 1.1, are as follows:

- The **Performance System** is the module that must solve the given performance task, in this case playing checkers, by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output.

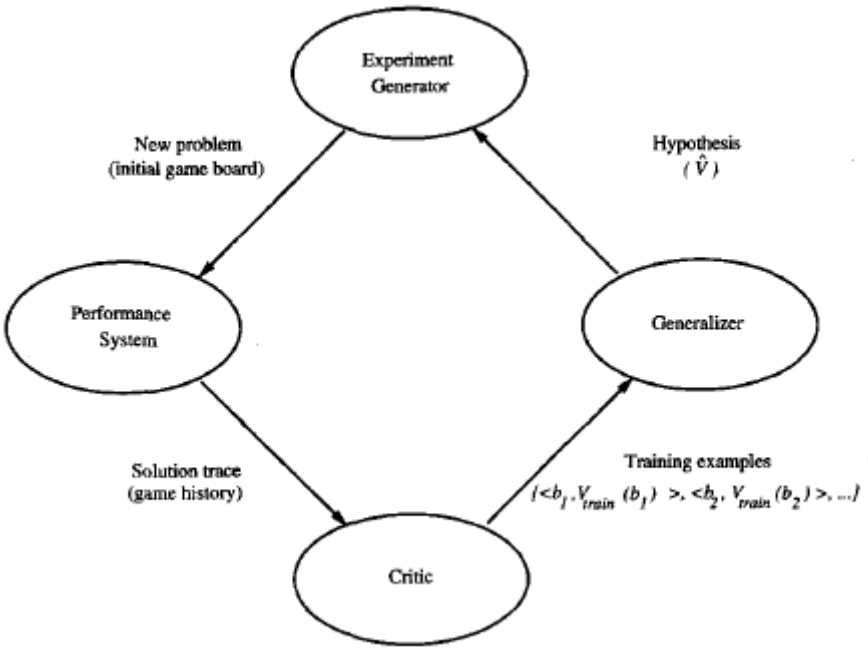


Figure 1: Final design of the checkers learning program

- The **Critic** takes as input the history or trace of the game and produces as output a set of training examples of the target function. As shown in the diagram, each training example in this case corresponds to some game state in the trace, along with an estimate **Vtrain,** of the target function value for this example.
- The **Generalizer** takes as input the training examples and produces an output hypothesis that is its estimate of the target function. It generalizes from the specific training examples, hypothesizing a general function that covers these examples and other cases beyond the training examples. In our example, the Generalizer corresponds to the LMS algorithm, and the output hypothesis is the function $\hat{V}$ described by the learned weights ***wo,*** . . . , ***w6.***

- The **Experiment Generator** takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e., initial board state) for the Performance System to explore. Its role is to pick new practice problems that will maximize the learning rate of the overall system. In our example, the Experiment Generator follows a very simple strategy: It always proposes the same initial game board to begin a new game.

1.4 PERSPECTIVES AND ISSUES IN MACHINE LEARNING

One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner. For example, consider the space of hypotheses that could in principle be output by the above checkers learner. This hypothesis space consists of all evaluation functions that can be represented by some choice of values for the weights ***wo*** through ***w6.*** The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with the available training examples. The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value. This algorithm works well when the hypothesis representation considered by the learner defines a continuously parameterized space of potential hypotheses.

1.4.1 Issues in Machine Learning

Our checkers example raises a number of generic questions about machine learning. The field of machine learning, is concerned with answering questions such as the following:

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?
- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?

- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

1.5 INTRODUCTION TO CONCEPT LEARNING

A concept describes a subset of objects or events defined over a larger set (e,g, concept of names of people, names of places, non-names).Concept learning is the process of inferring a boolean-valued function from training examples of its input and output.

1.6 A CONCEPT LEARNING TASK

Consider the example task of learning the target concept "days on which my friend Aldo enjoys his favorite water sport." Table 2.1 describes a set of example days, each represented by a set of ***attributes.*** The attribute ***EnjoySport*** indicates whether or not Aldo enjoys his favorite water sport on this day. The task is to learn to predict the value of ***EnjoySport*** for an arbitrary day, based on the values of its other attributes.

Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular,let each hypothesis be a vector of six constraints, specifying the values of the six attributes ***Sky, AirTemp, Humidity, Wind, Water,*** and ***Forecast.*** For each attribute, the hypothesis will either

- indicate by a "?" that any value is acceptable for this attribute,
- specify a single required value (e.g., ***Warm)*** for the attribute, or
- indicate by a φ that no value is acceptable.

If some instance x satisfies all the constraints of hypothesis h, then h classifies x as a positive example (h(x) = 1). To illustrate, the hypothesis that Aldo enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression

$$\langle ?, Cold, High, ?, ?, ? \rangle$$

Table 1: Positive and negative training examples for the target concept *EnjoySport.*

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

The most general hypothesis-that every day is a positive example-is represented by

<?, ?, ?, ?, ?, ?>

and the most specific possible hypothesis-that no day is a positive example-is represented by

<φ,φ,φ,φ,φ,φ>

To summarize, the EnjoySport concept learning task requires learning the set of days for which EnjoySport = yes, describing this set by a conjunction of constraints over the instance attributes.

1.6.1 Notation

We employ the following terminology when discussing concept learning problems. The set of items over which the concept is defined is called the set of instances, which we denote by X. In the current example, X is the set of all possible days, each represented by the attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast.

The concept or function to be learned is called the target concept, which we denote by c. In general, c can be any Boolean valued function defined over the instances X; that is, c: **X ->{0, 1}**. In the current example, the target concept corresponds to the value of the attribute EnjoySport (i.e., c(x) = **1** if EnjoySport = Yes, and c(x) = **0** if EnjoySport = No).

Table 2:The *EnjoySport* concept learning *task.*

- **Given:**
  - Instances X: Possible days, each described by the attributes
    - *Sky* (with possible values *Sunny*, *Cloudy*, and *Rainy*),
    - *AirTemp* (with values *Warm* and *Cold*),
    - *Humidity* (with values *Normal* and *High*),
    - *Wind* (with values *Strong* and *Weak*),
    - *Water* (with values *Warm* and *Cool*), and
    - *Forecast* (with values *Same* and *Change*).
  - Hypotheses H: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The constraints may be "?" (any value is acceptable), "Ø" (no value is acceptable), or a specific value.
  - Target concept c: $EnjoySport : X \rightarrow \{0, 1\}$
  - Training examples D: Positive and negative examples of the target function (see Table 2.1).
- **Determine:**
  - A hypothesis h in H such that $h(x) = c(x)$ for all x in X.

When learning the target concept, the learner is presented a set of **training examples,** each consisting of an instance **x** from X, along with its target concept value **c(x)** (e.g., the training examples in Table 2.1). Instances for which **c(x)** = 1 are called **positive examples,** or members of the target concept. Instances for which **c(x)** = 0 are called **negative examples,** or non members of the target concept. We will often write the ordered pair **(x, c(x))** to describe

the training example consisting of the instance **x** and its target concept value **c(x).** We use the symbol **D** to denote the set of available training examples.

Given a set of training examples of the target concept **c,** the problem faced by the learner is to hypothesize, or estimate, **c.** We use the symbol H to denote the set of **all possible hypotheses** that the learner may consider regarding the identity of the target concept. In general, each hypothesis ***h*** in H represents a boolean-valued function defined over X; that is, **h**: X -> {0, 1}.

1.6.2 The Inductive Learning Hypothesis

Notice that although the learning task is to determine a hypothesis h identical to the target concept ***c*** over the entire set of instances X, the only information available about ***c*** is its value over the training examples. Therefore, inductive learning algorithms can at best guarantee that the output hypothesis fits the target concept over the training data.

The inductive learning hypothesis: Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

1.7 CONCEPT LEARNING AS SEARCH

Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation. The goal of this search is to find the hypothesis that best fits the training examples. It is important to note that by selecting a hypothesis representation, the designer of the learning algorithm implicitly defines the space of all hypotheses that the program can ever represent and therefore can ever learn.

Consider, for example, the instances X and hypotheses H in the ***EnjoySport*** learning task. Given that the attribute ***Sky*** has three possible values, and that ***AirTemp, Humidity, Wind, Water,*** and ***Forecast*** each have two possible values, the instance space X contains exactly 3 .2 2 .2 2 .2 = 96 distinct instances. A similar calculation shows that there are 5.4.4 .4 .4.4 = 5120 syntactically distinct hypotheses within H. Notice, however, that every hypothesis containing one or more "φ" symbols represents the empty set of instances; that is, it classifies every instance as negative. Therefore, the number of semantically distinct hypotheses is only 1 + (4.3.3.3.3.3) = 973. Our EnjoySport example is a very simple learning task, with a relatively small, finite hypothesis space. Most practical learning tasks involve much larger, sometimes infinite, hypothesis spaces.

1.7.1 General-to-Specific Ordering of Hypotheses

Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis. To illustrate the general-to-specific ordering, consider the two hypotheses

$$h_1 = \langle Sunny, ?, ?, Strong, ?, ? \rangle$$
$$h_2 = \langle Sunny, ?, ?, ?, ?, ? \rangle$$

Now consider the sets of instances that are classified positive by hl and by h2.Because h2 imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by hl will also be classified positive by h2. Therefore, we say that h2 is more general than hl.

This intuitive "more general than" relationship between hypotheses can be defined more precisely as follows. First, for any instance x in X and hypothesis h in H, we say that x satisfies h if and only if h(x) = 1. We now define the **more_general_than_or_equal_to** relation in terms of the sets of instances that satisfy the two hypotheses: Given hypotheses h_j and h_k, h_j is more_general_than_or_equal_to h_k if and only if any instance that satisfies h_k also satisfies h_j.

Definition: Let h_j and h_k be boolean-valued functions defined over X. Then h_j is **more_general_than_or_equal_to** h_k (written $h_j \geq_g h_k$) if and only if

$$(\forall x \in X)[(h_k(x) = 1) \rightarrow (h_j(x) = 1)]$$

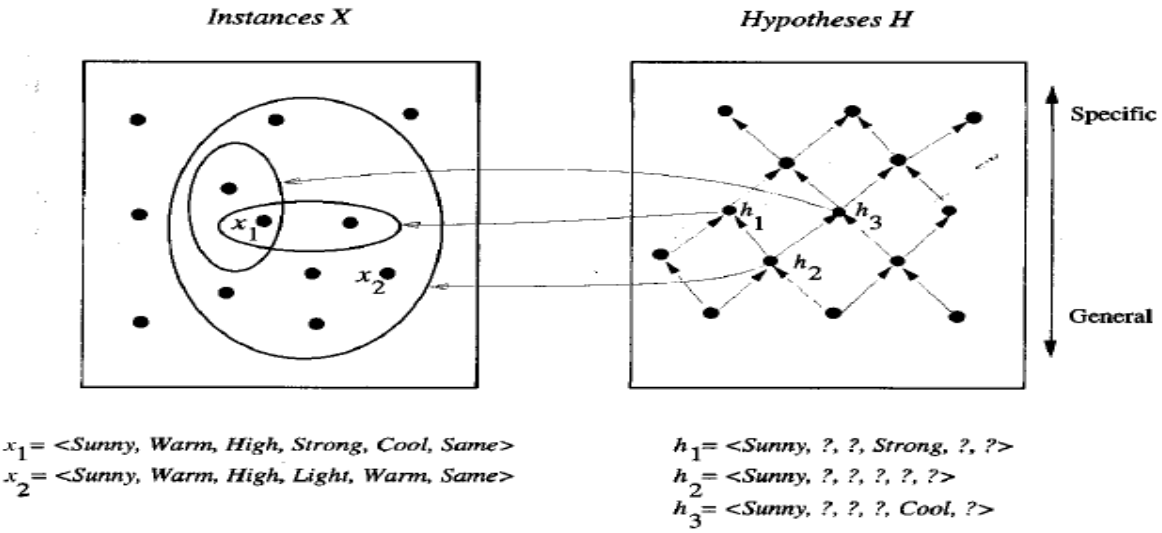


Figure 2: Instances, hypotheses, and the more_general_than relation.

The box on the left represents the set X of all instances, the box on the right the set *H* of all hypotheses. Each hypothesis corresponds to some subset of X-the subset of instances that it classifies positive. The arrows connecting hypotheses represent more_general_than relation, with the arrow pointing toward the less general hypothesis. Note the subset of instances characterized by *h2* subsumes the subset characterized by *hl,* hence *h2* is more_general_than *hl.*

1.8 FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

How can we use the *more_general_than* partial ordering to organize the search for a hypothesis consistent with the observed training examples? One way is to begin with the most specific possible hypothesis in H, then generalize this hypothesis each time it fails to cover an observed positive training example. (We say that a hypothesis "covers" a positive example if it correctly classifies the example as positive.)

Table 3: FIND-S Algorithm

1. Initialize h to the most specific hypothesis in H
2. For each positive training instance x
   - For each attribute constraint a_i in h
     - If the constraint a_i is satisfied by x
     - Then do nothing
     - Else replace a_i in h by the next more general constraint that is satisfied by x
3. Output hypothesis h

To illustrate this algorithm, assume the learner is given the sequence of training examples from Table 1 for the *EnjoySport* task. The first step of FIND-S is to initialize *h* to the most specific hypothesis in H

$$h \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

Upon observing the first training example from Table 1, which happens to be a positive example, it becomes clear that our hypothesis is too specific. In particular, none of the "φ" constraints in h are satisfied by this example, so each is replaced by the next more general constraint that fits the example; namely, the attribute values for this training example.

$$h \leftarrow \langle Sunny, Warm, Normal, Strong, Warm, Same \rangle$$

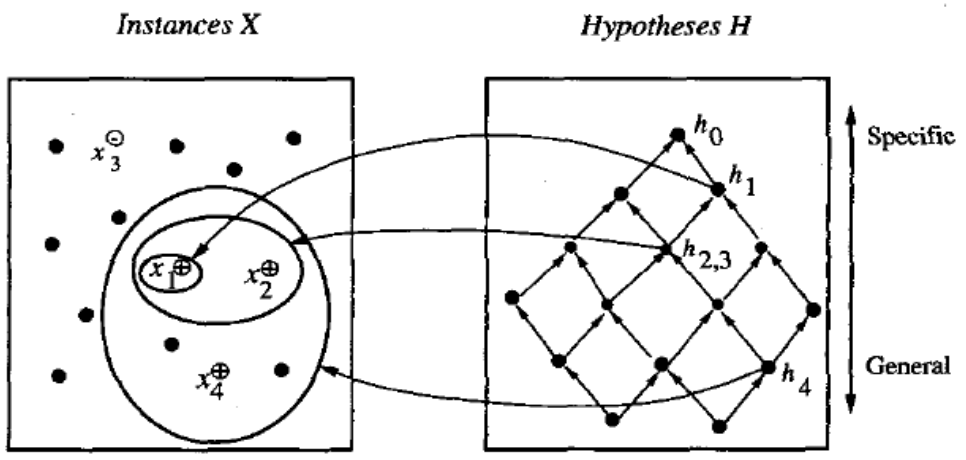
Next, the second training example (also positive in this case) forces the algorithm to further generalize *h,* this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example. The refined hypothesis in this case is

$$h \leftarrow \langle Sunny, Warm, ?, Strong, Warm, Same \rangle$$

Upon encountering the third training example-in this case a negative example-the algorithm makes no change to *h.* In fact, the FIND-S algorithm simply *ignores every negative example!* The fourth (positive) example leads to a further generalization of h

$$h \leftarrow \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

The FIND-S algorithm illustrates one way in which the ***more_general_than*** partial ordering can be used to organize the search for an acceptable hypothesis. The search moves from hypothesis to hypothesis, searching from the most specific to progressively more general hypotheses along one chain of the partial ordering. Figure 3 illustrates this search in terms of the instance and hypothesis spaces.



x_1 = <Sunny Warm Normal Strong Warm Same>, +
x_2 = <Sunny Warm High Strong Warm Same>, +
x_3 = <Rainy Cold High Strong Warm Change>, -
x_4 = <Sunny Warm High Strong Cool Change>, +

h_0 = <∅, ∅, ∅, ∅, ∅, ∅>
h_1 = <Sunny Warm Normal Strong Warm Same>
h_2 = <Sunny Warm ? Strong Warm Same>
h_3 = <Sunny Warm ? Strong Warm Same>
h_4 = <Sunny Warm ? Strong ? ? >

Figure 3: The hypothesis space search performed by FIND-S. The search begins *(ho)* with the most specific hypothesis in H, then considers increasingly general hypotheses *(hl* through *h4)* as mandated by the training examples. In the instance space diagram, positive training examples are denoted by "+", negative by "-" and instances that have not been presented as training examples are denoted by a solid circle.

1.9 VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

The key idea in the CANDIDATE-ELIMINATlON algorithm is to output a description of the set of ***all hypotheses consistent with the training examples.*** Surprisingly, the CANDIDATE-ELIMINATION algorithm computes the description of this set without explicitly enumerating all of its members. This is accomplished by again using the ***more_general_than*** partial ordering, this time to maintain a compact representation of the set of consistent hypotheses and to incrementally refine this representation as each new training example is encountered.

The CANDIDATE-ELIMINATION algorithm has been applied to problems such as learning regularities in chemical mass spectroscopy (Mitchell 1979) and learning control rules for heuristic search (Mitchell et al. 1983). Nevertheless, practical applications of the CANDIDATE-ELIMINATION & FIND-S algorithms are limited by the fact that they both perform poorly when given noisy training data.

1.9.1 Representation

The CANDIDATE-ELIMINATION algorithm finds all describable hypotheses that are consistent with the observed training examples. In order to define this algorithm precisely, we begin with a few basic definitions. First, let us say that a hypothesis is ***consistent*** with the training examples if it correctly classifies these examples.

Definition: A hypothesis h is **consistent** with a set of training examples D if and only if $h(x) = c(x)$ for each example $\langle x, c(x)\rangle$ in D.

$$Consistent(h, D) \equiv (\forall \langle x, c(x)\rangle \in D)\ h(x) = c(x)$$

Notice the key difference between this definition of ***consistent*** and our earlier definition of ***satisfies.*** An example *x* is said to ***satisfy*** hypothesis ***h*** when ***h(x)*** =1, regardless of whether x is a positive or negative example of the target concept. However, whether such an example is ***consistent*** with ***h*** depends on the target concept, and in particular, whether ***h(x)*** = ***c(x).***

The CANDIDATE-ELIMINATION algorithm represents the set of *all* hypotheses consistent with the observed training examples. This subset of all hypotheses is called the ***version space*** with respect to the hypothesis space H and the training examples D, because it contains all plausible versions of the target concept.

Definition: The **version space**, denoted $VS_{H,D}$, with respect to hypothesis space H and training examples D, is the subset of hypotheses from H consistent with the training examples in D.

$$VS_{H,D} \equiv \{h \in H | Consistent(h, D)\}$$

1.9.2 The LIST-THEN-ELIMINATE Algorithm

One obvious way to represent the version space is simply to list all of its members. This leads to a simple learning algorithm, which we might call the LIST-THEN ELIMINATE algorithm, defined in Table 4.

TABLE 4: The LIST-THEN-ELIMINATE Algorithm

The LIST-THEN-ELIMINATE Algorithm

1. $VersionSpace \leftarrow$ a list containing every hypothesis in H
2. For each training example, $\langle x, c(x)\rangle$
   remove from $VersionSpace$ any hypothesis h for which $h(x) \neq c(x)$
3. Output the list of hypotheses in $VersionSpace$

The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H, then eliminates any hypothesis found inconsistent with any training example. The version space of candidate hypotheses thus shrinks as more examples are observed, until ideally just one hypothesis remains that is consistent with all the observed examples.

1.9.3 A More Compact Representation for Version Spaces

The CANDIDATE-ELIMINATION algorithm works on the same principle as the above LIST-THEN-ELIMINATE algorithm. However, it employs a much more compact representation of the version space. In particular, the version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

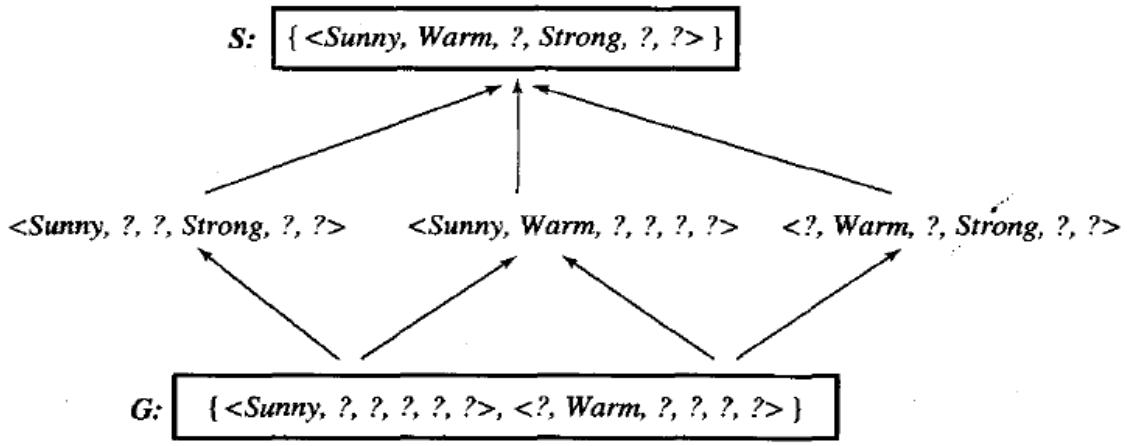


Figure 4: A version space with its general and specific boundary sets. The version space includes all six hypotheses shown here, but can be represented more simply by S and *G*. Arrows indicate instances of the *more_general_than* relation.

This is the version space for the *Enjoysport* concept learning problem and training examples described in Table 1.

To illustrate this representation for version spaces, consider again the En*joysport* concept learning problem described in Table 4. Recall that given the four training examples from Table 1, FIND-S outputs the hypothesis

$$h = \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

In fact, this is just one of six different hypotheses from H that are consistent with these training examples. All six hypotheses are shown in Figure 4. They constitute the version space relative to this set of data and this hypothesis representation. The arrows among these six hypotheses in Figure 4 indicate instances of the *more_general_than* relation. The CANDIDATE-ELIMINATE algorithm represents the version space by storing only its most general members (labeled G in Figure 4) and its most specific (labeled *S* in the figure). Given only these two sets *S* and G, it is possible to enumerate all members of the version space as needed by generating the hypotheses that lie between these two sets in the general-to-specific partial ordering over hypotheses.

Definition: The **general boundary** G, with respect to hypothesis space H and training data D, is the set of maximally general members of H consistent with D.

$$G \equiv \{g \in H | Consistent(g, D) \wedge (\neg \exists g' \in H)[(g' >_g g) \wedge Consistent(g', D)]\}$$

Definition: The **specific boundary** S, with respect to hypothesis space H and training data D, is the set of minimally general (i.e., maximally specific) members of H consistent with D.

$$S \equiv \{s \in H | Consistent(s, D) \wedge (\neg \exists s' \in H)[(s >_g s') \wedge Consistent(s', D)]\}$$

1.9.4 CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples. It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the ***G*** boundary set to contain the most general hypothesis in H

$$G_0 \leftarrow \{\langle ?, ?, ?, ?, ?, ? \rangle\}$$

and initializing the ***S*** boundary set to contain the most specific (least general) hypothesis

$$S_0 \leftarrow \{\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle\}$$

These two boundary sets delimit the entire hypothesis space, because every other hypothesis in H is both more general than ***So*** and more specific than ***Go.*** As each training example is considered, the ***S*** and ***G*** boundary sets are generalized and specialized, respectively, to

eliminate from the version space any hypotheses found inconsistent with the new training example. After all examples have been processed, the computed version space contains all the hypotheses consistent with these examples and only these hypotheses. This algorithm is summarized in Table 5.

TABLE 5: CANDIDATE-ELIMINATION algorithm using version spaces.

Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

- If d is a positive example
  - Remove from G any hypothesis inconsistent with d
  - For each hypothesis s in S that is not consistent with d
    - Remove s from S
    - Add to S all minimal generalizations h of s such that
      - h is consistent with d, and some member of G is more general than h
    - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
  - Remove from S any hypothesis inconsistent with d
  - For each hypothesis g in G that is not consistent with d
    - Remove g from G
    - Add to G all minimal specializations h of g such that
      - h is consistent with d, and some member of S is more specific than h
    - Remove from G any hypothesis that is less general than another hypothesis in G

1.9.5 An Illustrative Example

Figure 5 traces the CANDIDATE-ELIMINATION algorithm applied to the first two training examples from Table 1. As described above, the boundary sets are first initialized to Go and So, the most general and most specific hypotheses in H, respectively.

When the first training example is presented (a positive example in this case), the CANDIDATE-ELIMINATION algorithm checks the *S* boundary and finds that it is overly specific-it fails to cover the positive example. The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example. This revised boundary is shown as S1 in Figure 5. No update of the G boundary is needed in response to this training example because G0 correctly covers this example. When the second training example (also positive) is observed, it has a similar effect of generalizing ***S*** further to ***S2,*** leaving G again unchanged (i.e., G2 = **G1**= G0)

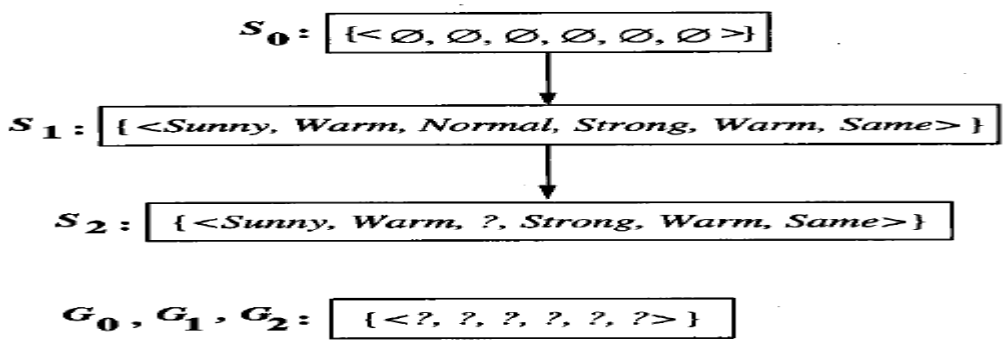


Figure 5: CANDIDATE-ELIMINATION Trace 1. *So* and *Go* are the initial boundary sets corresponding to the most specific and most general hypotheses. Training examples 1 and 2 force the *S* boundary to become more general, as in the FIND-S algorithm. They have no effect on the *G* boundary.

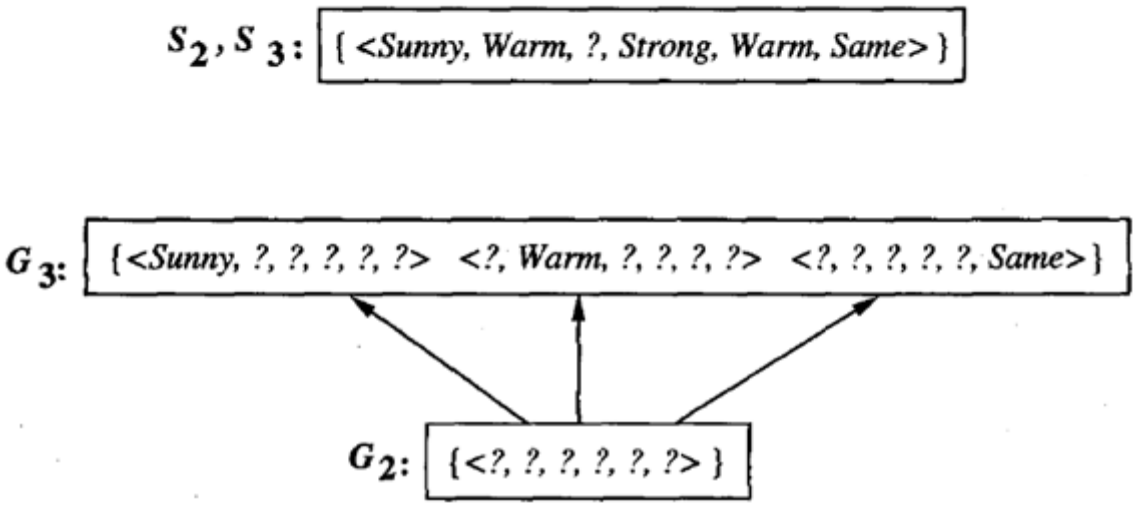
Training examples:

1. *<Sunny, Warm, Normal, Strong, Warm, Same>, Enjoy Sport = Yes*

2. *<Sunny, Warm, High, Strong, Warm, Same>, Enjoy Sport = Yes*

As illustrated by these first two steps, positive training examples may force the **S** boundary of the version space to become increasingly general. Negative training examples play the complimentary role of forcing the **G** boundary to become increasingly specific. Consider the third training example, shown in Figure 5. This negative example reveals that the **G** boundary of the version space is overly general; that is, the hypothesis in **G** incorrectly predicts that this new example is a positive example. The hypothesis in the **G** boundary must therefore be specialized until it correctly classifies this new negative example. As shown in Figure 5, there are several alternative minimally more specific hypotheses. All of these become members of the new **G3** boundary set.

Given that there are six attributes that could be specified to specialize **G2,** why are there only three new hypotheses in **G3?** For example, the hypothesis ***h*** = (?, ?, ***Normal,*** ?, ?, ?) is a minimal specialization of **G2** that correctly labels the new example as a negative example, but it is not included in **G3.** The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples. The algorithm determines this simply by noting that ***h*** is not more general than the current specific boundary, **S2.**

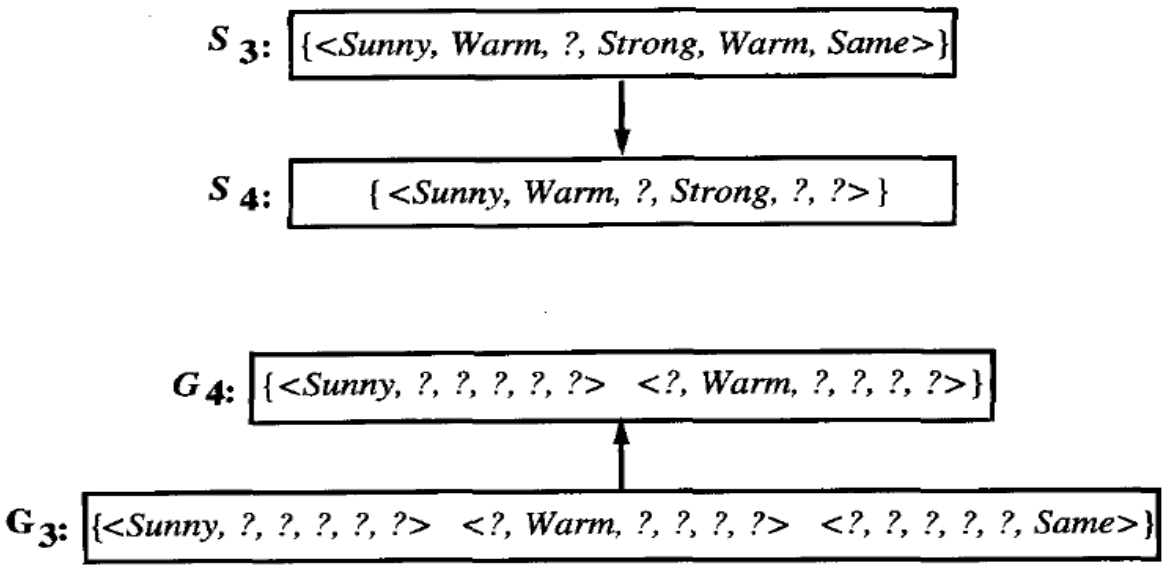


Training Example:

3. *<Rainy, Cold, High, Strong, Warm, Change>, EnjoySport=No*

FIGURE 6: CANDIDATE-ELMNATION Trace 2. Training example 3 is a negative example that forces the G2 boundary to be specialized to G3. Note several alternative maximally general hypotheses are included in G3.

The fourth training example, as shown in Figure 7, further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example. This last action results from the first step under the condition "If d is a positive example" in the algorithm shown in Table 5.



Training Example:

4. *<Sunny, Warm, High, Strong, Cool, Change>, EnjoySport = Yes*

FIGURE 7: CANDIDATE-ELIMINATION Trace 3. The positive training example generalizes the *S* boundary, from *S3* to *S4*. One member of *G3* must also be deleted, because it is no longer more general than the *S4* boundary.

After processing these four examples, the boundary sets **S4** and **G4** delimit the version space of all hypotheses consistent with the set of incrementally observed training examples. The entire version space, including those hypotheses bounded by **S4** and **G4,** is shown in Figure 8. This learned version space is independent of the sequence in which the training examples are presented (because in the end it contains all hypotheses consistent with the set of examples).

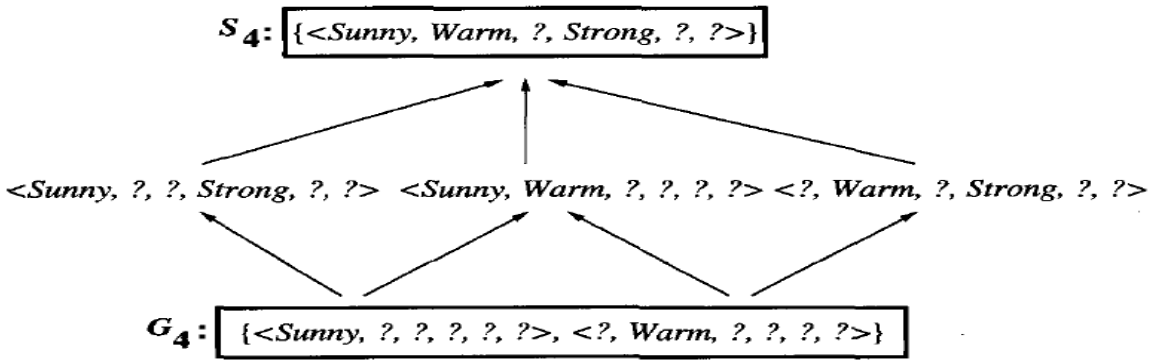


FIGURE 8: The final version space for the *EnjoySport* concept learning problem and training examples described earlier.

1.10 INDUCTIVE BIAS

As discussed above, the CANDIDATE-ELIMINATION Algorithm will converge toward the true target concept provided it is given accurate training examples and provided its initial hypothesis space contains the target concept. What if the target concept is not contained in the hypothesis space? Can we avoid this difficulty by using a hypothesis space that includes every possible hypothesis? How does the size of this hypothesis space influence the ability of the algorithm to generalize to unobserved instances? How does the size of the hypothesis space influence the number of training examples that must be observed? These are fundamental questions for inductive inference in general.

1.10.1 A Biased Hypothesis Space

Suppose we wish to assure that the hypothesis space contains the unknown target concept. The obvious solution is to enrich the hypothesis space to include ***every possible*** hypothesis. To illustrate, consider again the ***EnjoySport*** example in which we restricted the hypothesis space to include only conjunctions of attribute values. Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as ***"Sky*** = ***Sunny*** or ***Sky*** = ***Cloudy."*** In fact, given the following three training examples of this disjunctive hypothesis, our algorithm would find that there are zero hypotheses in the version space.

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Cool	Change	Yes
2	Cloudy	Warm	Normal	Strong	Cool	Change	Yes
3	Rainy	Warm	Normal	Strong	Cool	Change	No

To see why there are no hypotheses consistent with these three examples, note that the most specific hypothesis consistent with the first two examples ***and representable in the given hypothesis space H*** is

$$S_2 : \langle ?, Warm, Normal, Strong, Cool, Change \rangle$$

This hypothesis, although it is the maximally specific hypothesis from H that is consistent with the first two examples, is already overly general: it incorrectly covers the third (negative) training example. The problem is that we have biased the learner to consider only conjunctive hypotheses.

1.10.2 An Unbiased Learner

The obvious solution to the problem of assuring that the target concept is in the hypothesis space H is to provide a hypothesis space capable of representing ***every teachable concept;*** that is, it is capable of representing every possible subset of the instances X. In general, the set of all subsets of a set X is called ***the power set*** of X.

In the ***EnjoySport*** learning task, for example, the size of the instance space X of days described by the six available attributes is 96. How many possible concepts can be defined over this set of instances? In other words, how large is the power set of X? In general, the number of distinct subsets that can be defined over a set X containing **|X|** elements (i.e., the size of the power set of X) is $2^{|X|}$ Thus, there are $\mathbf{2^{96}}$**,** or approximately 10^{28}distinct target concepts that could be defined over this instance space and that our learner might be called upon to learn.

Let us reformulate the ***Enjoysport*** learning task in an unbiased way by defining a new hypothesis space ***H'*** that can represent every subset of instances; that is, let H' correspond to the power set of X. One way to define such an H' is to allow arbitrary disjunctions, conjunctions, and negations of our earlier hypotheses. For instance, the target concept ***"Sky*** = ***Sunny*** or ***Sky*** = ***Cloudy"*** could then be described as

$$\langle Sunny, ?, ?, ?, ?, ? \rangle \vee \langle Cloudy, ?, ?, ?, ?, ? \rangle$$

Acknowledgement

The diagrams and tables are taken from the textbooks specified in the references section.

Prepared by:

Rakshith M D

Department of CS&E

SDMIT, Ujire

MODULE 2: DECISION TREE LEARNING

References
1. Tom M. Mitchell, Machine Learning, India Edition 2013, McGraw Hill Education.

2.1 INTRODUCTION

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability.

2.2 DECISION TREE REPRESENTATION

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

Figure 2.1 illustrates a typical learned decision tree. This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis.

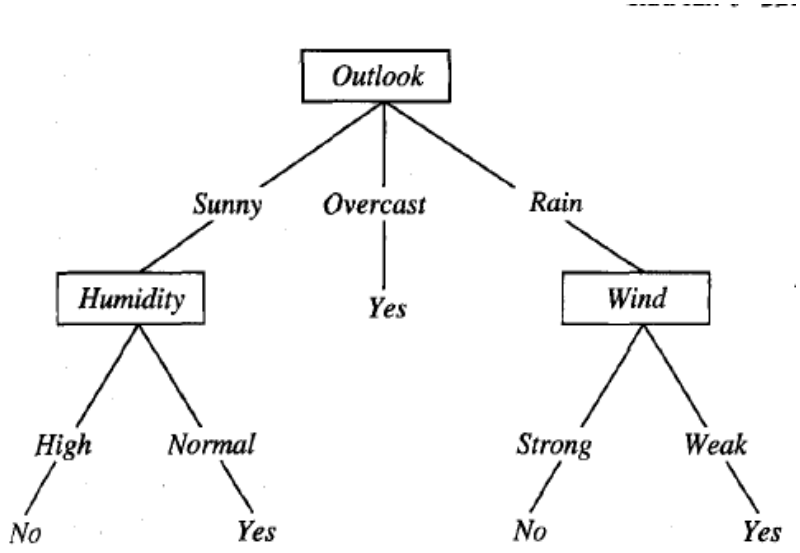


Figure 2.1: A decision tree for the concept *PlayTennis.* An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No).* This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

For example, the instance

$$\langle Outlook = Sunny,\ Temperature = Hot,\ Humidity = High,\ Wind = Strong \rangle$$

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that ***PlayTennis*** = ***no).***

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions. For example, the decision tree shown in Figure **2.1**corresponds to the expression

$$(Outlook = Sunny \wedge Humidity = Normal)$$
$$\vee \quad (Outlook = Overcast)$$
$$\vee \quad (Outlook = Rain \wedge Wind = Weak)$$

2.3 APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Decision tree learning is generally best suited to problems with the following characteristics:

- ***Instances are represented by attribute-value pair:*** Instances are described by a fixed set of attributes (e.g., ***Temperature)*** and their values (e.g., ***Hot).*** The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., ***Hot, Mild, Cold).***
- ***The target function has discrete output values:*** The decision tree in Figure 2.1 assigns a boolean classification (e.g., ***yes*** or ***no)*** to each example. Decision tree methods easily extend to learning functions with more than two possible output values.
- ***Disjunctive descriptions may be required:*** As noted above, decision trees naturally represent disjunctive expressions.
- ***The training data may contain errors:*** Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.
- ***The training data may contain missing attribute values:*** Decision tree methods can be used even when some training examples have unknown values (e.g., if the ***Humidity*** of the day is known for only some of the training examples).

2.4 THE BASIC DECISION TREE LEARNING ALGORITHM

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. This approach is exemplified by the ID3 algorithm (Quinlan 1986) and its successor C4.5 (Quinlan 1993), which form the primary focus of our discussion here. A simplified version of the ID3 algorithm, specialized to learning boolean-valued functions (i.e., concept learning), is described in the table below.

Table 2.1: ID3 Algorithm

ID3(*Examples*, *Target_attribute*, *Attributes*)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = −
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for A
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *Target_attribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, *Target_attribute*, *Attributes* − {A}))
- End
- Return *Root*

2.4.1 Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most useful for classifying examples. We will define a statistical property, called ***informution gain,*** that measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

2.4.1.1 ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called ***entropy,*** that characterizes the (im)purity of an arbitrary collection of examples. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this boolean classification is

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \quad (2.1)$$

where $p_{\oplus}$ is the proportion of positive examples in S and $p_{\ominus}$, is the proportion of negative examples in S. In all calculations involving entropy we define 0 log 0 to be 0.

To illustrate, suppose S is a collection of 14 examples of some Boolean concept, including 9 positive and 5 negative examples (we adopt the notation [9+, 5-] to summarize such a sample of data). Then the entropy of S relative to this boolean classification is

$$Entropy([9+, 5-]) = -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) = 0.940 \quad \textbf{(2.2)}$$

Notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive $(p_\oplus = 1)$ then $p_\ominus$ is 0 and Entropy(S) = -1 . $\log_2(1)$ - 0 . $\log_2$ 0 = -1 . 0 - 0 . $\log_2$ 0 = 0. Note the entropy is 1 when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the entropy is between 0 and 1. Figure 2.2 shows the form of the entropy function relative to a boolean classification, as $p_\oplus$ **,** varies between 0 and 1.

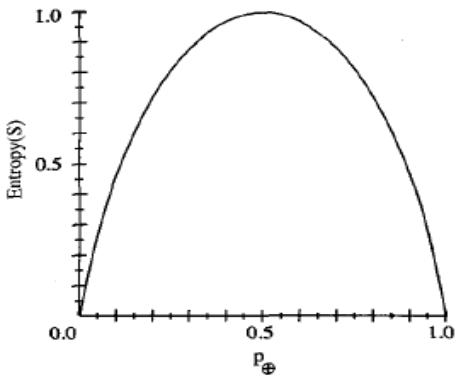


Figure 2.2: The entropy function relative to a boolean classification, as the proportion, $p_\oplus$ of positive examples varies between *0* and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of ***S*** (i.e., a member of ***S*** drawn at random with uniform probability). For example, if $p_\oplus$ is 1, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is zero. On the other hand, if $p_\oplus$ is 0.5, one bit is required to indicate whether the drawn example is positive or negative. If the target attribute can take on c different values, then the entropy of ***S*** relative to this c-wise classification is defined as

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \quad \textbf{(2.3)}$$

where ***pi*** is the proportion of ***S*** belonging to class ***i.*** Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in ***bits.*** Note also that if the target attribute can take on c possible values, the entropy can be as large as $\log_2$ ***c.***

2.4.1.2 INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called ***information gain,*** is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, ***Gain(S, A)*** of ***an*** attribute **A,** relative to a collection of examples ***S,*** is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \quad \textbf{(2.4)}$$

where ***Values(A)*** is the set of all possible values for attribute A, and ***S,*** is the subset of S for which attribute A has value v $(\text{i.e., } S_v = \{s \in S | A(s) = v\})$. ***Gain(S, A)*** is therefore the expected reduction in entropy caused by knowing the value of attribute A. Put another way, ***Gain(S, A)*** is the information provided about the ***target &action value,*** given the value of some other attribute ***A.*** The value of ***Gain(S, A)*** is the number of bits saved when encoding the target value of an arbitrary member of ***S,*** by knowing the value of attribute ***A.***

For example, suppose ***S*** is a collection of training-example days described by attributes including ***Wind,*** which can have the values ***Weak*** or ***Strong.*** As before, assume ***S*** is a collection containing ***14*** examples, [9+, 5-]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have ***Wind*** = ***Weak,*** and the remainder have ***Wind*** = ***Strong.*** The information gain due to sorting the original ***14*** examples by the attribute ***Wind*** may then be calculated as

$$\begin{aligned}
Values(Wind) &= Weak, Strong \\
S &= [9+, 5-] \\
S_{Weak} &\leftarrow [6+, 2-] \\
S_{Strong} &\leftarrow [3+, 3-] \\
Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - (8/14) Entropy(S_{Weak}) \\
&\quad - (6/14) Entropy(S_{Strong}) \\
&= 0.940 - (8/14)0.811 - (6/14)1.00 \\
&= 0.048
\end{aligned}$$

Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized in Figure **2.3.**

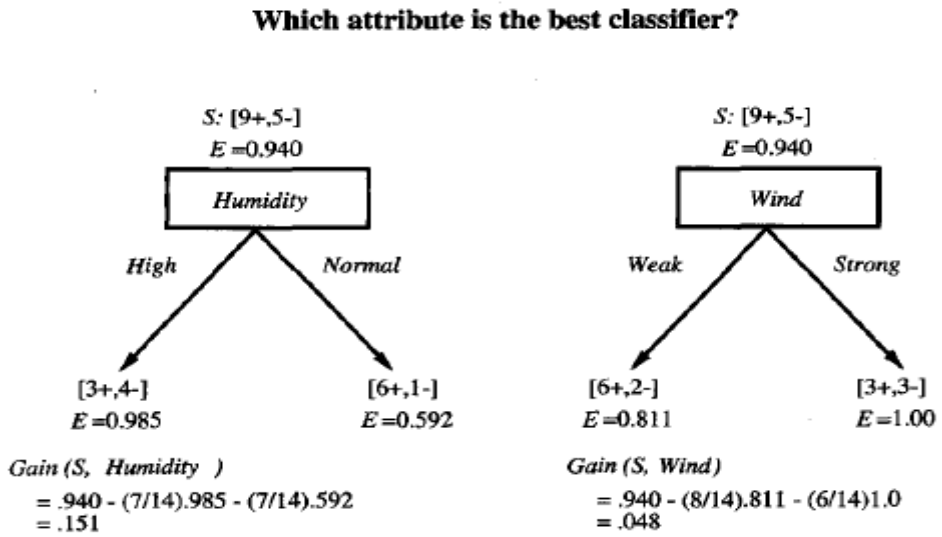


Figure 2.3: *Humidity* provides greater information gain than *Wind,* relative to the target classification. Here, E stands for entropy and *S* for the original collection of examples. Given an initial collection *S* of 9 positive and 5 negative examples, [9+, 5-], sorting these by their *Humidity* produces collections of[3+, 4-] *(Humidity = High)* and [6+, 1-] *(Humidity = Normal).* The information gained by this partitioning is .151, compared to a gain of only .048 for the attribute *Wind.*

2.5 An Illustrative Example

To illustrate the operation of ID3, consider the learning task represented by the training examples of Table 2.2. Here the target attribute ***PlayTennis,*** which can have values ***yes*** or ***no*** for different Saturday mornings, is to be predicted based on other attributes of the morning in question.

Table 2.2: Training examples for the target concept *PlayTennis.*

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*PlayTennis*
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Consider the first step through the algorithm, in which the topmost node of the decision tree is created. Which attribute should be tested first in the tree? ID3 determines the

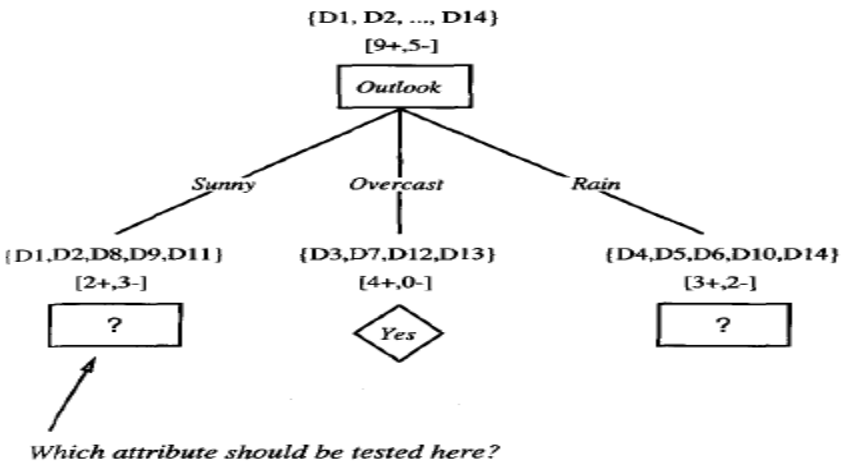
information gain for each candidate attribute (i.e., ***Outlook, Temperature, Humidity,*** and ***Wind),*** then selects the one with highest information gain.The computation of information gain for two of these attributes is shown in Figure 2.3. The information gain values for all four attributes are

$$Gain(S, Outlook) = 0.246$$
$$Gain(S, Humidity) = 0.151$$
$$Gain(S, Wind) = 0.048$$
$$Gain(S, Temperature) = 0.029$$

where ***S*** denotes the collection of training examples from Table 2.2.

According to the information gain measure, the ***Outlook*** attribute provides the best prediction of the target attribute, ***PlayTennis,*** over the training examples. Therefore, ***Outlook*** is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values (i.e.,***Sunny, Overcast,*** and ***Rain).*** The resulting partial decision tree is shown in Figure 2.4, along with the training examples sorted to each new descendant node.

Note that every example for which ***Outlook*** = ***Overcast*** is also a positive example of ***PlayTennis.*** Therefore, this node of the tree becomes a leaf node with the classification ***PlayTennis*** = ***Yes.*** In contrast, the descendants corresponding to ***Outlook*** = ***Sunny*** and ***Outlook*** = ***Rain*** still have nonzero entropy, and the decision tree will be further elaborated below these nodes. Figure 2.4 illustrates the computations of information gain for the next step in growing the decision tree. The final decision tree learned by ID3 from the 14 training examples of Table 2.2 is shown in Figure 2.4.



S_{sunny} = {D1,D2,D8,D9,D11}

$Gain(S_{sunny}, Humidity) = .970 - (3/5)\,0.0 - (2/5)\,0.0 = .970$

$Gain(S_{sunny}, Temperature) = .970 - (2/5)\,0.0 - (2/5)\,1.0 - (1/5)\,0.0 = .570$

$Gain(S_{sunny}, Wind) = .970 - (2/5)\,1.0 - (3/5)\,.918 = .019$

Figure 2.4: The partially learned decision tree resulting from the first step of ID3.

The training examples are sorted to the corresponding descendant nodes. The *Overcast* descendant has only positive examples and therefore becomes a leaf node with classification *Yes.* The other two nodes will be further expanded, by selecting the attribute with highest information gain relative to the new subsets of examples.

2.6 HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

As with other inductive learning methods, ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched by ID3 is the set of possible decision trees. ID3 performs a simple-to-complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data. The evaluation function that guides this hill-climbing search is the information gain measure. This search is depicted in Figure 2.5.

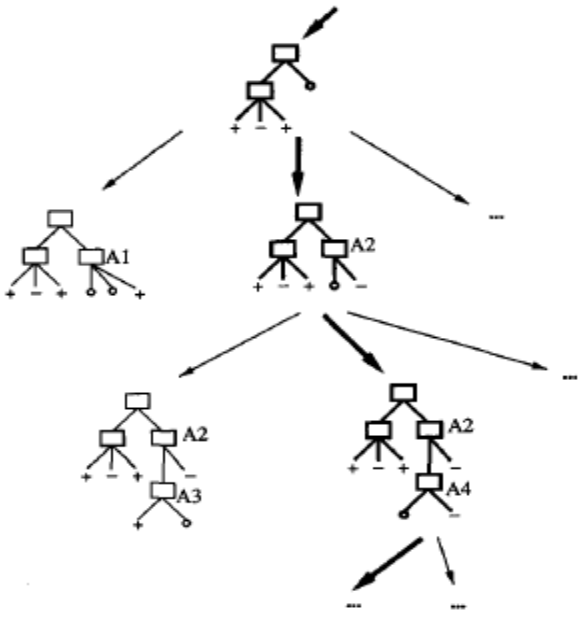


Figure 2.5: Hypothesis space search by ID3. It searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

By viewing ID3 in terms of its search space and search strategy, we can get some insight into its capabilities and limitations:

- **ID3's** hypothesis space of all decision trees is a ***complete*** space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree.

- **ID3** maintains only a single current hypothesis as it searches through the space of decision trees. This contrasts, for example, with the earlier version space Candidate-Elimination method which maintains the set of ***all*** hypotheses consistent with the available training examples.
- **ID3** in its pure form performs no backtracking in its search. Once it, selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice. Therefore, it is susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal.
- **ID3** uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis. This contrasts with methods that make decisions incrementally, based on individual training examples (e.g., FIND-S or CANDIDATE-ELIMINATION). One advantage of using statistical properties of all the examples (e.g., information gain) is that the resulting search is much less sensitive to errors in individual training examples.

2.7 INDUCTIVE BIAS IN DECISION TREE LEARNING

What is the policy by which ID3 generalizes from observed training examples to classify unseen instances? In other words, what is its inductive bias? Recall from Chapter 2 that inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances.

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 therefore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. Which of these decision trees does ID3 choose? It chooses the first acceptable tree it encounters in its simple-to-complex, hill climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favour of shorter trees over longer ones, and (b) selects trees that place the attributes with highest information gain closest to the root.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees.

Consider an algorithm that begins with the empty tree and searches ***breadth first*** through progressively more complex trees, first considering all trees of depth 1, then all trees of depth 2, etc. Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes). Let us call

this breadth-first search algorithm BFS-ID3. BFS-ID3 finds a shortest decision tree and thus exhibits precisely the bias "shorter trees are preferred over longer trees."

As ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3. In particular, it does not always find the shortest consistent tree, and it is biased to favor trees that place attributes with high information gain closest to the root.

A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

1) Restriction Biases and Preference Biases

There is an interesting difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION algorithm. Consider the difference between the hypothesis space search in these two approaches:

- ID3 searches a complete hypothesis space (i.e., one capable of expressing any finite discrete-valued function). It searches incompletely through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data).
- The version space CANDIDATE-ELIMINATION algorithm searches an incomplete hypothesis space (i.e., one that can express only a subset of the potentially teachable concepts). It searches this space completely, finding every hypothesis consistent with the training data.

In brief, the inductive bias of ID3 follows from its search strategy, where as the inductive bias of the CANDIDATE-ELIMINATION algorithm follows from the definition of its search space.

2) Why Prefer Short Hypotheses?

Is ID3's inductive bias favouring shorter decision trees a sound basis for generalizing beyond the training data? Philosophers and others have debated this question for centuries, and the debate remains unresolved to this day. William of Occam was one of the first to discuss the question, around the year 1320, so this bias often goes by the name of Occam's razor.

Occam's razor: Prefer the simplest hypothesis that fits the data. Consider decision tree hypotheses, for example. There are many more 500-node decision trees than 5-node decision trees. Given a small set of 20 training examples, we might expect to be able to find many

500-node decision trees consistent with these, whereas we would be more surprised if a 5-node decision tree could perfectly fit this data. We might therefore believe the 5-node tree is less likely to be a statistical coincidence and prefer this hypothesis over the 500-node hypothesis.

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used ***internally*** by the learner. Two learners using different internal representations could therefore arrive at different hypotheses, both justifying their contradictory conclusions by Occam's razor! For example, the function represented by the learned decision tree in Figure 2.1 could be represented as a tree with just one decision node, by a learner that uses the boolean attribute XYZ, where we define the attribute XYZ to be true for instances that are classified positive by the decision tree in Figure 2.1 and false otherwise. Thus, two learners, both applying Occam's razor, would generalize in different ways if one used the XYZ attribute to describe its examples and the other used only the attributes ***Outlook, Temperature, Humidity,*** and ***Wind.***

This last argument shows that Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners that perceive these examples in terms of different internal representations. On this basis we might be tempted to reject Occam's razor altogether.

2.8 ISSUES IN DECISION TREE LEARNING

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure, handling training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency.

1) Avoiding Overfitting the Data

The algorithm described in Table 2.1 grows each branch of the tree just deeply enough to perfectly classify the training examples. While this is sometimes a reasonable strategy, in fact it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function.

Definition: Given a hypothesis space H, a hypothesis $h \in H$ is said to **overfit** the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

Figure 2.6 illustrates the impact of overfitting in a typical application of decision tree learning. In this case, the ID3 algorithm is applied to the task of learning which medical

patients have a form of diabetes. The horizontal axis of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree. The solid line shows the accuracy of the decision tree over the training examples, whereas the broken line shows accuracy measured over an independent set of test examples (not included in the training set).

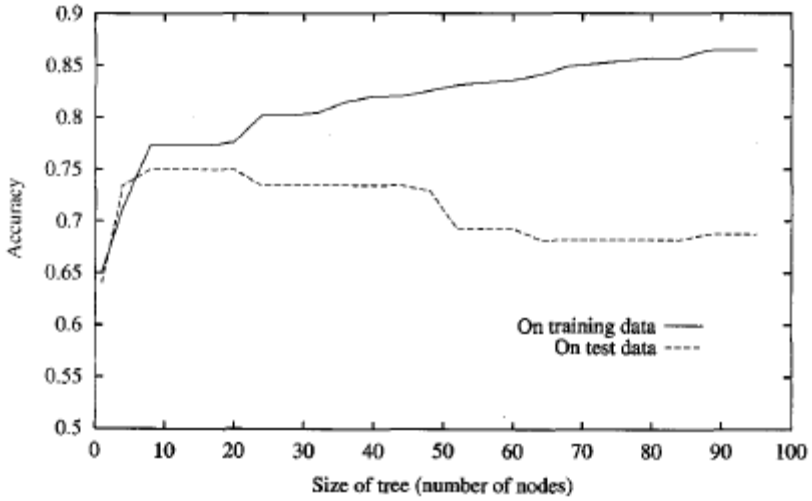


Figure 2.6: Overfitting in decision tree learning. As ID3 adds new nodes to grow the decision tree, the accuracy of the tree measured over the training examples increases monotonically. However, when measured over a **set of test examples independent of the training examples, accuracy first increases, then decreases.**

How can it be possible for tree h to fit the training examples better than h', but for it to perform more poorly over subsequent examples? One way this can occur is when the training examples contain random errors or noise. To illustrate, consider the effect of adding the following positive training example, incorrectly labeled as negative, to the (otherwise correct) examples in Table 2.2.

$$\langle Outlook = Sunny, Temperature = Hot, Humidity = Normal,$$
$$Wind = Strong, PlayTennis = No \rangle$$

Given the original error-free data, ID3 produces the decision tree shown in Figure 2.1. However, the addition of this incorrect example will now cause ID3 to construct a more complex tree. In particular, the new example will be sorted into the second leaf node from the left in the learned tree of Figure 3.1, along with the previous positive examples D9 and Dl 1. Because the new example is labeled as a negative example, ID3 will search for further refinements to the tree below this node. Of course as long as the new erroneous example differs in some arbitrary way from the other examples affiliated with this node, ID3 will

succeed in finding a new decision attribute to separate out this new example from the two previous positive examples at this tree node. The result is that ID3 will output a decision tree (h) that is more complex than the original tree from Figure 2.1 (h').

Overfitting is a significant practical difficulty for decision tree learning and many other learning methods. For example, in one experimental study of ID3 involving five different learning tasks with noisy, nondeterministic data (Mingers 1989b), overfitting was found to decrease the accuracy of learned decision trees by 10-25% on most problems.

There are several approaches to avoiding overfitting in decision tree learning. These can be grouped into two classes:

1. approaches that stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data,
2. approaches that allow the tree to overfit the data, and then post-prune the tree.

Although the first of these approaches might seem more direct, the second approach of post-pruning overfit trees has been found to be more successful in practice. This is due to the difficulty in the first approach of estimating precisely when to stop growing the tree. Regardless of whether the correct tree size is found by stopping early or by post-pruning, a key question is what criterion is to be used to determine the correct final tree size. Approaches include:

1. Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree.
2. Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond the training set. For example, Quinlan (1986) uses a chi-square test to estimate whether further expanding a node is likely to improve performance over the entire instance distribution, or only on the current sample of training data.
3. Use an explicit measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. This approach, based on a heuristic called the Minimum Description Length principle.

a) REDUCED ERROR PRUNING

How exactly might we use a validation set to prevent overfitting? One approach, called reduced-error pruning (Quinlan 1987), is to consider each of the decision nodes in the.tree to be candidates for pruning. Pruning a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node. Nodes are removed only if

the resulting pruned tree performs no worse than-the original over the validation set. This has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy over the validation set.

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in Figure **2.7.** As in Figure **2.6,** the accuracy of the tree is shown measured over both training examples and test examples. The additional line in Figure **2.7** shows accuracy over the test examples as the tree is pruned. When pruning begins, the tree is at its maximum size and lowest accuracy over the test set. As pruning proceeds, the number of nodes is reduced and accuracy over the test set increases. Here, the available data has been split into three subsets: the training examples, the validation examples used for pruning the tree, and a set of test examples used to provide an unbiased estimate of accuracy over future unseen examples. Using a separate set of data to guide pruning is an effective approach provided a large amount of data is available. The major drawback of this approach is that when data is limited, withholding part of it for the validation set reduces even further the number of examples available for training.

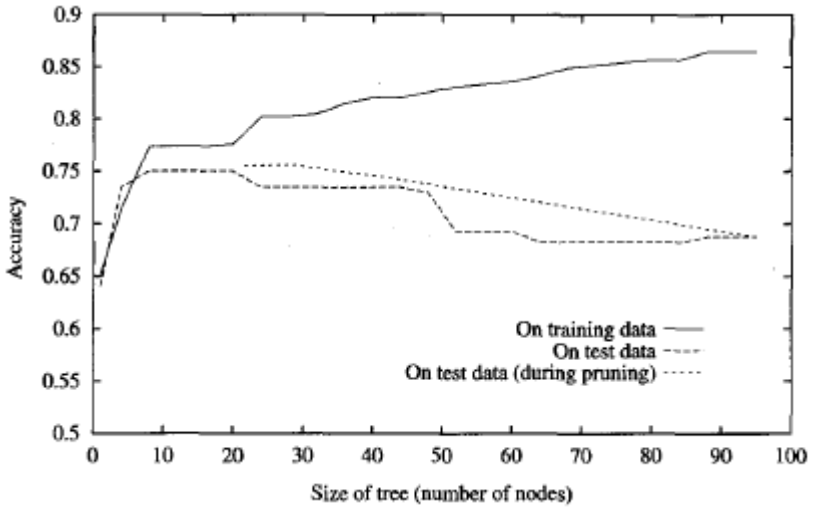


Figure 2.7: Effect of reduced-error pruning in decision tree learning. This plot shows the same curves of training and test set accuracy as in Figure 2.6. In addition, it shows the impact of reduced error pruning of the tree produced by ID3. Notice the increase in accuracy over the test set as nodes are pruned from the tree. Here, the validation set used for pruning is distinct from both the training and test sets.

b) RULE POST-PRUNING

In practice, one quite successful method for finding high accuracy hypotheses is a technique we shall call rule post-pruning. A variant of this pruning method is used by C4.5 (Quinlan

1993), which is an outgrowth of the original ID3 algorithm. Rule post-pruning involves the following steps:

1. Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.
2. Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.
3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

To illustrate, consider again the decision tree in Figure 2.1. In rule postpruning, one rule is generated for each leaf node in the tree. Each attribute test along the path from the root to the leaf becomes a rule antecedent (precondition) and the classification at the leaf node becomes the rule consequent (postcondition). For example, the leftmost path of the tree in Figure 2.1 is translated into the rule

$$\textbf{IF} \quad (Outlook = Sunny) \wedge (Humidity = High)$$
$$\textbf{THEN} \quad PlayTennis = No$$

Next, each such rule is pruned by removing any antecedent, or precondition, whose removal does not worsen its estimated accuracy. Given the above rule, for example, rule post-pruning would consider removing the preconditions (Outlook = Sunny) and (Humidity = High). It would select whichever of these pruning steps produced the greatest improvement in estimated rule accuracy, then consider pruning the second precondition as a further pruning step. No pruning step is performed if it reduces the estimated rule accuracy.

Why convert the decision tree to rules before pruning? There are three main advantages.

1. Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path.
2. Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves.
3. Converting to rules improves readability. Rules are often easier for to understand.

2) Incorporating Continuous-Valued Attributes

Our initial definition of ID3 is restricted to attributes that take on a discrete set of values. First, the target attribute whose value is predicted by the learned tree must be discrete valued. Second, the attributes tested in the decision nodes of the tree must also be discrete valued. This second restriction can easily be removed so that continuous-valued decision attributes can be incorporated into the learned tree. This can be accomplished by dynamically defining new discrete-valued attributes that partition the continuous attribute value into a discrete set of intervals.

As an example, suppose we wish to include the continuous-valued attribute ***Temperature*** in describing the training example days in the learning task of Table 2.2. Suppose further that the training examples associated with a particular node in the decision tree have the following values for ***Temperature*** and the target attribute ***PlayTennis.***

Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

What threshold-based boolean attribute should be defined based on Temperature? Clearly, we would like to pick a threshold, ***c,*** that produces the greatest information gain. By sorting the examples according to the continuous attribute **A,** then identifying adjacent examples that differ in their target classification, we can generate a set of candidate thresholds midway between the corresponding values of ***A.*** In the current example, there are two candidate thresholds, corresponding to the values of Temperature at which the value of PlayTennis changes: (48 + 60)/2, and (80 + 90)/2. The information gain can then be computed for each of the candidate attributes, $Temperature_{>54}$ and $Temperature_{>85}$the best can be selected ($Temperature_{>54}$).

3) Alternative Measures for Selecting Attributes

There is a natural bias in the information gain measure that favors attributes with many values over those with few values. As an extreme example, consider the attribute Date, which has a very large number of possible values (e.g., March 4, 1979). If we were to add this attribute to the data in Table 2.2, it would have the highest information gain of any of the attributes. This is because Date alone perfectly predicts the target attribute over the training data. Thus, it would be selected as the decision attribute for the root node of the tree and lead to a (quite broad) tree of depth one, which perfectly classifies the training data.

What is wrong with the attribute Date? Simply put, it has so many possible values that it is bound to separate the training examples into very small subsets. Because of this, it will

have a very high information gain relative to the training examples, despite being a very poor predictor of the target function over unseen instances.

One way to avoid this difficulty is to select decision attributes based on some measure other than information gain. One alternative measure that has been used successfully is the gain ratio (Quinlan 1986). The gain ratio measure penalizes attributes such as Date by incorporating a term, called *split information,* that is sensitive to how broadly and uniformly the attribute splits the data:

$$SplitInformation(S, A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|} \quad (2.5)$$

where S1 through Sc are the ***c*** subsets of examples resulting from partitioning S by the c-valued attribute A. Note that *SplitInfomation* is actually the entropy of S with respect to the values of attribute A.

The Gain Ratio measure is defined in terms of the earlier Gain measure, as well as this *SplitInfomation,* as follows:

$$GainRatio(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A)} \quad (2.6)$$

One practical issue that arises in using GainRatio in place of Gain to select attributes is that the denominator can be zero or very small when $|S_i| \approx |S|$ for one of the Si. This either makes the GainRatio undefined or very large for attributes that happen to have the same value for nearly all members of S. To avoid selecting attributes purely on this basis, we can adopt some heuristic such as first calculating the Gain of each attribute, then applying the GainRatio test only considering those attributes with above average Gain (Quinlan 1986).

4) Handling Training Examples with Missing Attribute Values

In certain cases, the available data may be missing values for some attributes. For example, in a medical domain in which we wish to predict patient outcome based on various laboratory tests, it may be that the lab test Blood-Test-Result is available only for a subset of the patients. In such cases, it is common to estimate the missing attribute value based on other examples for which this attribute has a known value.

Consider the situation in which ***Gain(S, A)*** is to be calculated at node ***n*** in the decision tree to evaluate whether the attribute ***A*** is the best attribute to test at this decision node. Suppose that ***(x, c(x))*** is one of the training examples in ***S*** and that the value ***A(x)*** is unknown.

One strategy for dealing with the missing attribute value is to assign it the value that is most common among training examples at node ***n.*** Alternatively, we might assign it the most common value among examples at node ***n*** that have the classification ***c(x).***

A second, more complex procedure is to assign a probability to each of the possible values of ***A*** rather than simply assigning the most common value to ***A(x).***These probabilities can be estimated again based on the observed frequencies of the various values for ***A*** among the examples at node ***n.*** For example, given a boolean attribute ***A,*** if node ***n*** contains six known examples with ***A*** = 1 and four with ***A*** = 0, then we would say the probability that ***A(x)*** = 1 is 0.6, and the probability that ***A(x)*** = 0 is 0.4.

5) Handling Attributes with Differing Costs

In some learning tasks the instance attributes may have associated costs. For example, in learning to classify medical diseases we might describe patients in terms of attributes such as Temperature, BiopsyResult, Pulse, BloodTestResults, etc. These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort. In such tasks, we would prefer decision trees that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classifications.

ID3 can be modified to take into account attribute costs by introducing a cost term into the attribute selection measure. For example, we might divide the ***Gain*** by the cost of the attribute, so that lower-cost attributes would be preferred. While such cost-sensitive measures do not guarantee finding an optimal cost-sensitive decision tree, they do bias the search in favor of low-cost attributes.

Attribute cost is measured by the number of seconds required to obtain the attribute value by positioning and operating the sonar. They demonstrate that more efficient recognition strategies are learned, without sacrificing classification accuracy, by replacing the information gain attribute selection measure by the following measure

$$\frac{Gain^2(S, A)}{Cost(A)}$$

Acknowledgement

The diagrams and tables are taken from the textbooks specified in the references section.

Prepared by:

Rakshith M D

Department of CS&E

SDMIT, Ujire

MODULE 3: ARTIFICIAL NEURAL NETWORKS

References
1. Tom M. Mitchell, Machine Learning, India Edition 2013, McGraw Hill Education.

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPOGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs. ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

3.1 INTRODUCTION

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the BACKPROPAGATION algorithm has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters (LeCun et al. 1989).

3.1.1 Biological Motivation

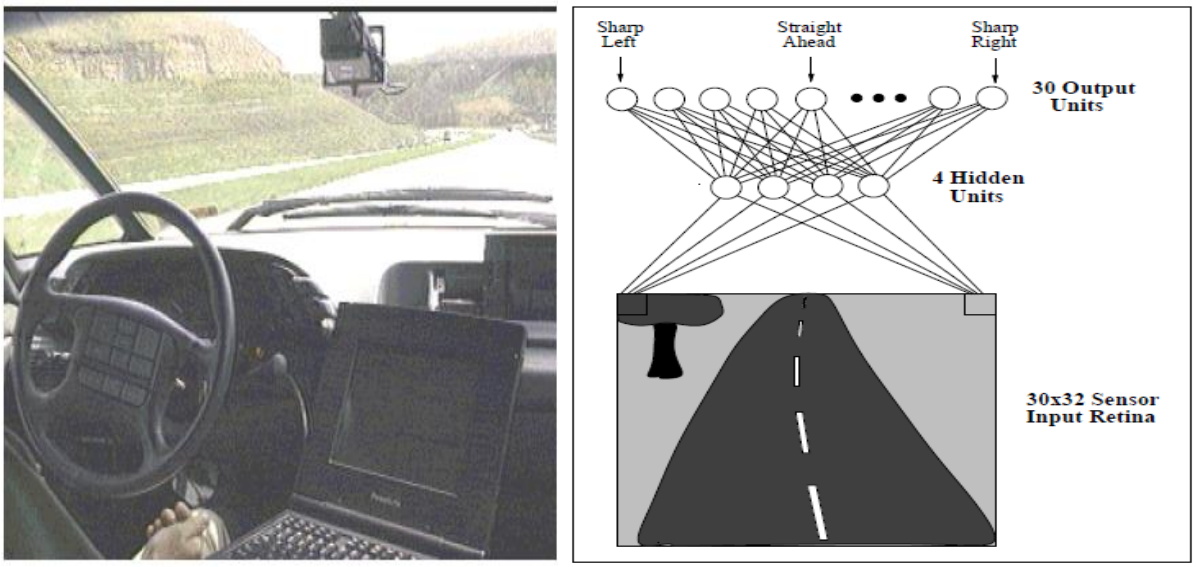
The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

To develop a feel for this analogy, let us consider a few facts from neurobiology.The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^4 others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of 10^{-3} seconds quite slow compared to computer switching speeds of 10^{-10} seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately 10^{-1} seconds to visually recognize your mother.

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes.

3.2 NEURAL NETWORK REPRESENTATIONS

A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways. The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).



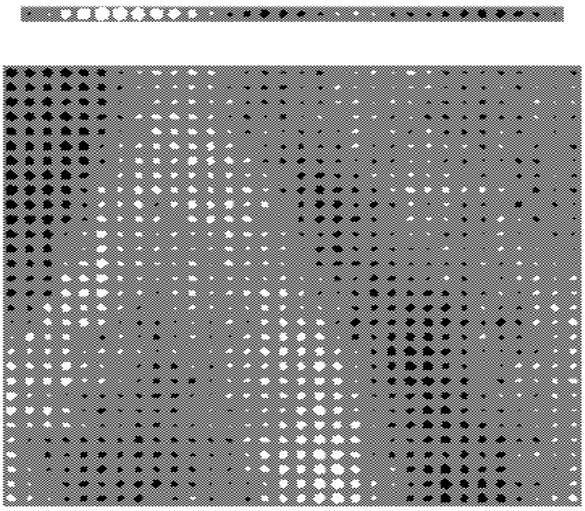


FIGURE 3.1: Neural network learning to steer an autonomous vehicle. The ALVINN system uses **BACKPROPAGATION** to learn to steer an autonomous vehicle (photo at top left) driving at speeds up to **70** miles per hour. The diagram on the top right shows how the image of a forward-mounted camera is mapped to **960** neural network inputs, which are fed forward to 4 hidden units, connected to **30** output units. Network outputs encode the commanded steering direction. The figure on the bottom left shows weight values for one of the hidden units in this network. The **30** x 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the **30** output units **are** depicted by the smaller rectangular block directly above the large block.

3.3 APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones. It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks. The BACKPROPAGATION algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

- ***Instances are represented by many attribute-value pairs.*** The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- ***The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.*** For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction.
- ***The training examples may contain errors.*** ANN learning methods are quite robust to noise in the training data.
- ***Long training times are acceptable.*** Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- ***Fast evaluation of the learned target function may be required.*** Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- ***The ability of humans to understand the learned target function is not important.*** The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

3.4 PERCEPTRON

One type of ANN system is based on a unit called a perceptron, illustrated in Figure 3.2. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise.

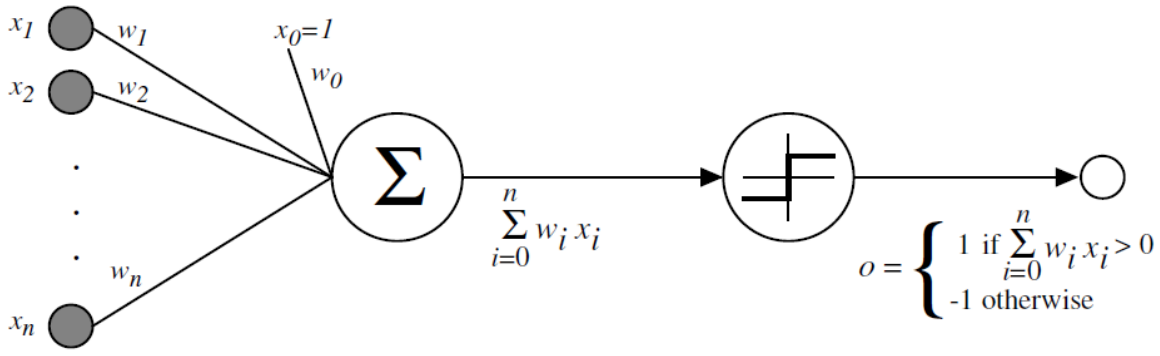


Figure 3.2: A perceptron

More precisely, given inputs ***x1*** through ***xn*** the output ***o(x1,*** . . . , ***xn)*** computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each w_i is a real-valued constant, or weight, that determines the contribution of input ***xi*** to the perceptron output. Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for the perceptron to output a 1. To simplify notation, we imagine an additional constant input ***xo*** = 1, allowing us to write the above inequality a $\sum_{i=0}^{n} w_i x_i > 0$ or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x}) \quad \text{where}$$

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

3.4.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Figure 3.3.The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some

sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called linearly separable sets of examples.

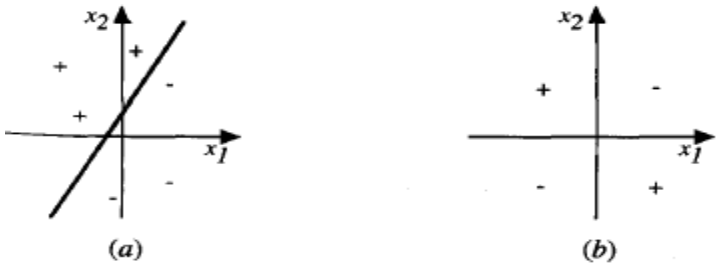


Figure 3.3: The decision surface represented by a two-input perceptron. *(a)* A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). *X1* and *X2* are the Perceptron inputs. Positive examples are indicated by "+", **negative by** "-".

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights *wo* = *-.8,* and *w1* = *w2* = *.5*. This perceptron can be made to represent the OR function instead by altering the threshold to *wo* = *-.3*.

3.4.2 The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ± output for each of the given training examples. Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the ***perceptron training rule,*** which revises the weight ***wi*** associated with input ***xi*** according to the rule:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and *q* is a positive constant called the ***learning rate.*** The role of the learning rate

is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

3.4.3 Gradient Descent and the Delta Rule

The perceptron rule will fail to converge if the examples are not linearly separable. The second rule called delta rule is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept. The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples. The delta training rule is best understood by considering the task of training an unthreshold perceptron; that is a linear unit for which the output O is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \quad \text{------- (1)}$$

The measure for the ***training error*** of a hypothesis (weight vector)

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \quad \text{--- (2)}$$

3.4.4 Visualizing the Hypothesis space

To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values, as illustrated in Figure 3.4

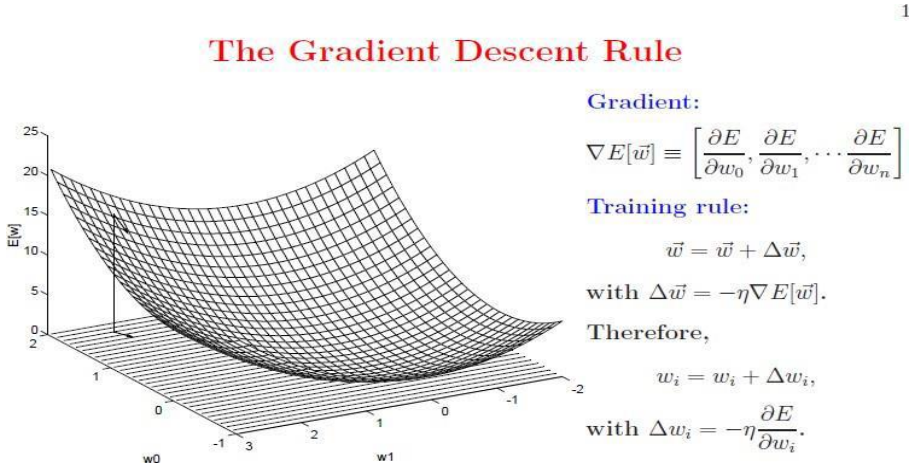


Figure 3.4: Hypothesis Space

Here the axes w_o and w_1 represent possible values for the two weights of a simple linear unit. The w_o, w_1 plane therefore represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples. The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error). Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum. The specific parabola will depend, of course, on the particular set of training examples.

3.4.5 Derivation of the Gradient Descent Rule

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of ***E*** with respect to each component of the vector $\vec{w}$. This vector derivative is called the ***gradient*** of ***E*** with respect to $\vec{w}$., written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \quad \text{----- (3)}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of ***E*** with respect to each of the w_i. When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in Figure 3.5 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_o, w_1 plane. Since the gradient specifies the direction of steepest increase of ***E***, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

Where

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \quad \text{---------- (4)}$$

Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that ***decreases*** E. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

Where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad \text{---------- (5)}$$

To construct a practical algorithm for iteratively updating weights according to Equation 5, we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating E from Equation 2, as

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \qquad \text{------- (6)}$$

where x_{id} denotes the single input component x_i for training example d. We now have an equation that gives in terms of the linear unit inputs x_{id}, outputs O_d, and target values t_d associated with the training examples. Substituting Equation 6 into Equation 5 yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d)\ x_{id} \qquad \text{-------- (7)}$$

3.4.6 Stochastic Approximation to Gradient Descent

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values,* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Table 1: Gradient Descent algorithm for training a linear unit

One common variation on gradient descent intended to alleviate these difficulties is called ***incremental gradient descent,*** or alternatively ***stochastic gradient descent.*** Whereas the gradient descent training rule presented in Equation **(7)** computes weight updates after summing over all the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for ***each*** individual example. The modified training rule is like the training rule given by Equation **(7)** except that ***as*** we iterate through each training example we update the weight according to

$$\Delta w_i = \eta(t - o)\ x_i \quad \text{---------- (8)}$$

where **t, o,** and **xi** are the target value, unit output, and i^{th} input for the training example in question. To modify the gradient descent algorithm of Table **1** to implement this stochastic approximation, Equation **1** is simply deleted and is replaced by $w_i \leftarrow w_i + \eta(t-o)\ x_i$.One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \quad \text{---------- (9)}$$

where $\mathbf{t_d}$, and o_d are the target value and the unit output value for training example d. Stochastic gradient descent iterates over the training examples ***d*** in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$.

3.5 MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

Single perceptron's can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPACATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface is depicted in Figure 3.5.Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.).

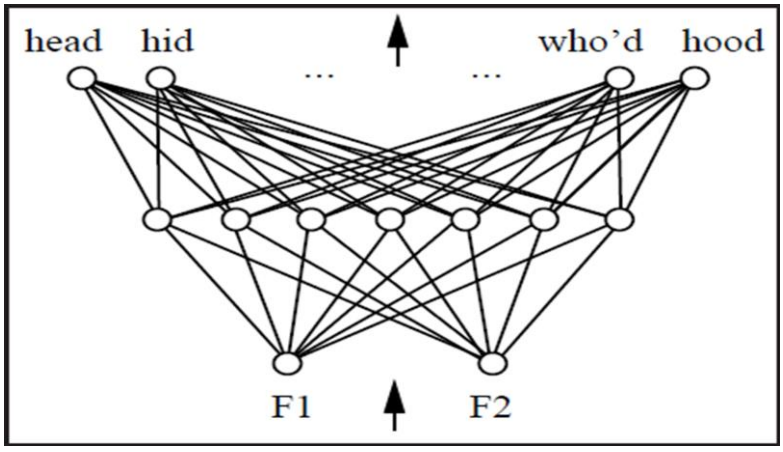


Figure 3.5: A typical multilayer network

3.5.1 A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions.

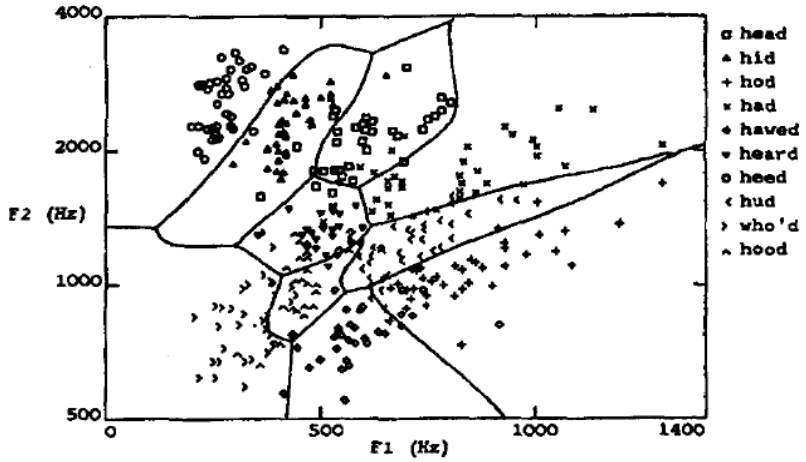


Figure 3.6: Decision regions of a multilayer feed-forward network

The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function. The sigmoid unit is illustrated in Figure 3.7.

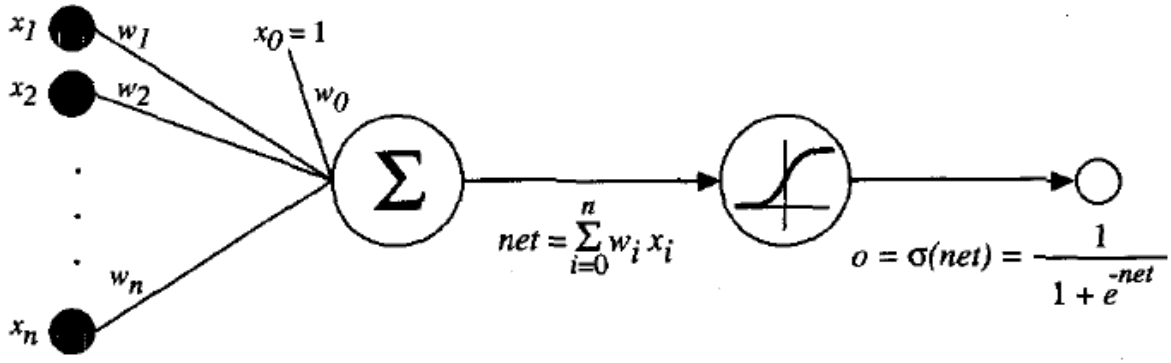


Figure 3.7: The sigmoid threshold unit

Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its output ***o*** as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

Where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \quad \text{---------- (10)}$$

σ is often called the sigmoid function or, alternatively, the logistic function. Note its output ranges between 0 and 1, increasing monotonically with its input.

3.5.2 The BACKPROPAGATION Algorithm

The BACKPROPAGATION Algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. This section presents the BACKPROPAGATION algorithm, and the following section gives the derivation for the gradient descent weight update rule used by BACKPROPAGATION. Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \quad \text{---------- (11)}$$

where ***outputs*** is the set of output units in the network, and $\mathbf{t_{kd}}$ and $\mathbf{o_{kd}}$ are the target and output values associated with the k[th] output unit and training example d.

BACKPROPAGATION(*training_examples*, η, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form $(\vec{x}, \vec{t})$, *where* $\vec{x}$ *is the vector of network input values, and* $\vec{t}$ *is the vector of target network output values.*

η *is the learning rate (e.g., .05).* n_{in} *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*

The input from unit i into unit j is denoted x_{ji}, *and the weight from unit i to unit j is denoted* w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between −.05 and .05).
- Until the termination condition is met, Do
  - For each $(\vec{x}, \vec{t})$ in *training_examples*, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \quad \text{(T4.3)}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \quad \text{(T4.4)}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \quad \text{(T4.5)}$$

Table 2: The stochastic gradient descent version of the Backpropagation algorithm for feed-forward networks containing two layers of sigmoid units.

The **Backpropagation algorithm** is presented in Table 2. The algorithm as described here applies to layered feed-forward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of Backpropagation algorithm. The notation used here is the same as that used in earlier sections, with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- $\mathbf{x_{ji}}$ denotes the input from node i to unit ***j*** **,** and $\boldsymbol{w_{ji}}$ denotes the corresponding weight.

  δ_n denotes the error term associated with unit ***n.*** It plays a role analogous to the quantity ***(t - o)*** in our earlier discussion of the delta training rule. As we shall see later,

$$\delta_n = -\frac{\partial E}{\partial net_n}$$

3.5.3 Derivation of the BACKPROPAGATION Rule

This section presents the derivation of the Backpropagation weight-tuning rule.It may be skipped on a first reading, without loss of continuity.The specific problem we address here is deriving the stochastic gradient descent rule implemented by the algorithm in Table 2. Recall from Equation **(10)** that stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error ***Ed*** with respect to this single example. In other words, for each training example d every weight ***wji*** is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad \text{---------- (12)}$$

where ***Ed*** is the error on training example d, summed overall output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here ***outputs*** is the set of output units in the network, ***tk*** is the target value of unit k for training example d, and ***ok*** is the output of unit k given training example d.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables. We will follow the notation shown in Figure 3.7, adding a subscript ***j*** to denote to the jth unit of the network as follows:

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule seen in Equation ***12.***

Stage/Case 1: Computing the increments (Δ) for output unit weights

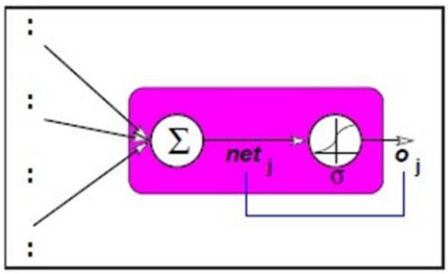
$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1-o_j)$$

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k\in outputs}(t_k - o_k)^2$$

$$= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}$$

$$= -(t_j - o_j)$$



$$\Rightarrow \frac{\partial E_d}{\partial net_j} = -(t_j - o_j)o_j(1-o_j) = -o_j(1-o_j)(t_j - o_j)$$

$$\Rightarrow \delta_j \overset{not.}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)(t_j - o_j)$$

$$\Rightarrow \Delta w_{ji} = \eta\delta_j x_{ji} = \eta o_j(1-o_j)(t_j - o_j)x_{ji}$$

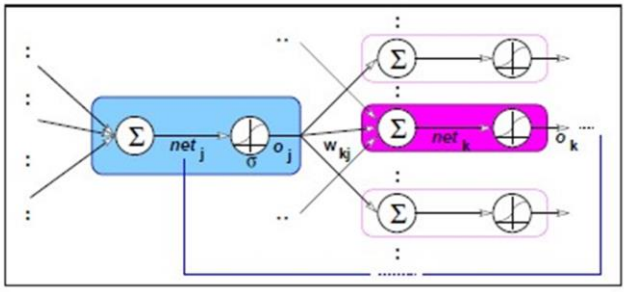
Stage/Case 2: Computing the increments (Δ) for hidden unit weights

$$\frac{\partial E_d}{\partial net_j} = \sum_{k\in Downstream(j)}\frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k\in Downstream(j)} -\delta_k\frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k\in Downstream(j)} -\delta_k\frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k\in Downstream(j)} -\delta_k w_{kj}\frac{\partial o_j}{\partial net_j} = \sum_{k\in Downstream(j)} -\delta_k w_{kj}o_j(1-o_j)$$



Therefore:

$$\delta_j \overset{not}{=} -\frac{\partial E_d}{\partial net_j} = o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k w_{kj}$$

$$\Delta w_{ji} \overset{def}{=} -\eta\frac{\partial E_d}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} = -\eta\frac{\partial E_d}{\partial net_j}x_{ji} = \eta\delta_j x_{ji} = \eta\left[\, o_j(1-o_j)\sum_{k\in Downstream(j)}\delta_k w_{kj}\,\right]x_{ji}$$

Acknowledgement

The diagrams and tables are taken from the textbooks specified in the references section.

Prepared by:

Rakshith M D

Department of CS&E

SDMIT, Ujire

MODULE 4: BAYESIAN LEARNING

References
1. Tom M. Mitchell, Machine Learning**,** India Edition 2013, McGraw Hill Education.

4.1 INTRODUCTION

Bayesian learning methods are relevant to our study of machine learning for two different reasons. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems. For example, Michie et **al.** (1994) provide a detailed study comparing the naive Bayes classifier to other learning algorithms, including decision tree and neural network algorithms.

The second reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities. For example, in this chapter we analyze algorithms such as the FIND-S and CANDIDATE ELIMINATION Algorithms of Chapter 2 to determine conditions under which they output the most probable hypothesis given the training data.

Features of Bayesian learning methods include:

- Each observed training example can incrementally decrease or increase the estimated probability that a hypothesis is correct. This provides a more flexible approach to learning than algorithms that completely eliminate a hypothesis if it is found to be inconsistent with any single example.
- Prior knowledge can be combined with observed data to determine the final probability of a hypothesis. In Bayesian learning, prior knowledge is provided by asserting (1) a prior probability for each candidate hypothesis, and (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions (e.g., hypotheses such as "this pneumonia patient has a 93% chance of complete recovery").
- New instances can be classified by combining the predictions of multiple hypotheses, weighted by their probabilities.
- Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

4.2 BAYES THEOREM

In machine learning we are often interested in determining the best hypothesis from some space H, given the observed training data D. One way to specify what we mean by the ***best*** hypothesis is to say that we demand the ***most probable*** hypothesis, given the data ***D*** plus any initial knowledge about the prior probabilities of the various hypotheses in H. Bayes theorem provides a direct method for calculating such probabilities. More precisely, Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

To define Bayes theorem precisely, let us first introduce a little notation. We shall write ***P(h)*** to denote the initial probability that hypothesis ***h*** holds, before we have observed the training data. ***P(h)*** is often called the ***prior probability*** of ***h*** and may reflect any background knowledge we have about the chance that ***h*** is a correct hypothesis. If we have no such prior knowledge, then we might simply assign the same prior probability to each candidate hypothesis. Similarly, we will write ***P(D)*** to denote the prior probability that training data D will be observed (i.e., the probability of D given no knowledge about which hypothesis holds). Next, we will write ***P(D|h)*** to denote the probability of observing data ***D*** given some world in which hypothesis ***h*** holds. More generally, we write ***P(x|y)*** to denote the probability of x given ***y.*** In machine learning problems we are interested in the probability ***P (h|D)*** that ***h*** holds given the observed training data ***D. P (h|D)*** is called the ***posterior probability*** of ***h,*** because it reflects our confidence that ***h*** holds after we have seen the training data ***D.*** Notice the posterior probability ***P(h|D)*** reflects the influence of the training data ***D,*** in contrast to the prior probability ***P(h)***, which is independent of D.

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability ***P(h|D),*** from the prior probability ***P(h),*** together with ***P(D)*** and ***P(D|h).***

Bayes theorem:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)} \quad \text{---------- (1)}$$

As one might intuitively expect, ***P(h|D)*** increases with ***P(h)*** and with ***P(D|h)*** according to Bayes theorem. It is also reasonable to see that ***P(h|D)*** decreases as ***P(D)*** increases, because the more probable it is that ***D*** will be observed independent of ***h,*** the less evidence ***D*** provides in support of ***h.***

In many learning scenarios, the learner considers some set of candidate hypotheses H and is interested in finding the most probable hypothesis ***h€*** H given the observed data ***D*** (or at least one of the maximally probable if there are several). Any such maximally probable hypothesis is called a ***maximum aposteriori*** (MAP) hypothesis. We can determine the MAP hypotheses by using Bayes theorem to calculate the posterior probability of each candidate hypothesis. More precisely, we will say that h_{MAP} is a MAP hypothesis provided

$$h_{MAP} \equiv \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

$$= \underset{h \in H}{\operatorname{argmax}} \frac{P(D|h)P(h)}{P(D)}$$

$$= \underset{h \in H}{\operatorname{argmax}} P(D|h)P(h) \quad \text{---------- (2)}$$

Notice in the final step above we dropped the term ***P (D)*** because it is a constant independent of ***h.*** In some cases, we will assume that every hypothesis in H is equally probable a priori ***($P(h_i)$ = $P(h_j)$*** for all ***h_i*** and ***h_j*** in H). In this case we can further simplify Equation (2) and need only consider the term ***P(D|h)*** to find the most probable hypothesis. ***P(D|h)*** is often called the ***likelihood*** of the data ***D*** given ***h,*** and any hypothesis that maximizes ***P(D|h)*** is called a ***maximum likelihood*** (ML) hypothesis, ***h_{ML}.***

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D|h) \quad \text{---------- (3)}$$

4.2.1 An Example

To illustrate Bayes rule, consider a medical diagnosis problem in which there are two alternative hypotheses: (1) that the patient has a particular form of cancer, and (2) that the patient does not. The available data is from a particular laboratory test with two possible outcomes: $\oplus$ (positive) and $\ominus$ (negative). We have prior knowledge that over the entire population of people only .008 have this disease. Furthermore, the lab test is only an imperfect indicator of the disease. The test returns a correct positive result in only 98% of the cases in which the disease is actually present and a correct negative result in only 97% of the cases in which the disease is not present. In other cases, the test returns the opposite result. The above situation can be summarized by the following probabilities:

$$P(cancer) = .008, \quad P(\neg cancer) = .992$$
$$P(\oplus|cancer) = .98, \quad P(\ominus|cancer) = .02$$
$$P(\oplus|\neg cancer) = .03, \quad P(\ominus|\neg cancer) = .97$$

Suppose we now observe a new patient for whom the lab test returns a positive result. Should we diagnose the patient as having cancer or not? The maximum aposteriori hypothesis can be found using Equation (2):

$$P(\oplus|cancer)P(cancer) = (.98).008 = .0078$$
$$P(\oplus|\neg cancer)P(\neg cancer) = (.03).992 = .0298$$

Thus, h_{MAP} = ⌐cancer. The exact posterior probabilities can also be determined by normalizing the above quantities so that they sum to 1 (e.g., $P(cancer|\oplus) = \frac{.0078}{.0078+.0298} = .21$).This step is warranted because Bayes theorem states that the posterior probabilities are just the above quantities divided by the probability of the data, P($\oplus$).

4.3 BAYES THEOREM AND CONCEPT LEARNING

What is the relationship between Bayes theorem and the problem of concept learning? Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, we can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable.

4.3.1 Brute-Force Bayes Concept Learning

Consider the concept learning problem first introduced in Chapter 2. In particular, assume the learner considers some finite hypothesis space H defined over the instance space X, in which the task is to learn some target concept ***c***: X -> **{0,1}.** As usual, we assume that the learner is given some sequence of training examples ***(($x_1 d_1$). . . (x_m, d_m))*** where x_i is some instance from X and where d_i is the target value of x_i (i.e., $d_i = c(x_i)$). To simplify the discussion in this section, we assume the sequence of instances $(x_1 \ldots x_m)$ is held fixed, so that the training data D can be written simply as the sequence of target values D = $(d_1 \ldots d_m)$. It can be shown that this simplification does not alter the main conclusions of this section. We can design a straightforward concept learning algorithm to output the maximum a posteriori hypothesis, based on Bayes theorem, as follows:

BRUTE-FORCE MAP LEARNING Algorithm

1. For each hypothesis ***h*** in H, calculate the posterior probability

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

2. Output the hypothesis h_{MAP} with the highest posterior probability

$$h_{MAP} = \underset{h \in H}{\operatorname{argmax}} P(h|D)$$

This algorithm may require significant computation, because it applies Bayes theorem to each hypothesis in H to calculate ***P(h|D) .*** While this may prove impractical for large hypothesis spaces, the algorithm is still of interest because it provides a standard against which we may judge the performance of other concept learning algorithms.

In order specify a learning problem for the **BRUTE-FORCE MAP LEARNING** algorithm we must specify what values are to be used for ***P(h)*** and for ***P(D|h)*** (as we shall see, ***P(D)*** will be determined once we choose the other two). We may choose the probability distributions ***P(h)*** and ***P(D|h)*** in any way we wish, to describe our prior knowledge about the learning task. Here let us choose them to be consistent with the following assumptions:

1. The training data ***D*** is noise free (i.e., $d_i = c(x_i)$).
2. The target concept ***c*** is contained in the hypothesis space H.
3. We have no a priori reason to believe that any hypothesis is more probable than any other.

Given these assumptions, what values should we specify for ***P(h)?*** Given no prior knowledge that one hypothesis is more likely than another, it is reasonable to assign the same prior probability to every hypothesis ***h*** in ***H.*** Furthermore, because we assume the target concept is contained in H we should require that these prior probabilities sum to 1. Together these constraints imply that we should choose

$$P(h) = \frac{1}{|H|} \quad \text{for all } h \text{ in } H$$

What choice shall we make for ***P(D|h)? P(D|h)*** is the probability of observing the target values D = $(d_1 \ldots d_m)$ for the fixed set of instances $(x_1 \ldots x_m)$, given a world in which hypothesis ***h*** holds (i.e., given a world in which ***h*** is the correct description of the target concept c). Since we assume noise-free training data, the probability of observing classification d_i given ***h*** is just ***1*** if $d_i = h(x_i)$ and 0 if $d_i \neq h(x_i)$. Therefore,

$$P(D|h) = \begin{cases} 1 \text{ if } d_i = h(x_i) \text{ for all } d_i \text{ in } D \\ 0 \text{ otherwise} \end{cases}$$ ---------- (4)

In other words, the probability of data ***D*** given hypothesis ***h*** is 1 if ***D*** is consistent with ***h,*** and 0 otherwise.

Given these choices for ***P(h)*** and for ***P(D|h)*** we now have a fully-defined problem for the above **BRUTE-FORCE MAP LEARNING** algorithm. Let us consider the first step of this algorithm, which uses Bayes theorem to compute the posterior probability ***P(h|D)*** of each hypothesis ***h*** given the observed training data ***D.***

Recalling Bayes theorem, we have

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

First consider the case where *h* is inconsistent with the training data *D.* Since Equation (4) defines *P(D|h)* to be *0* when *h* is inconsistent with *D*, we have

$$P(h|D) = \frac{0 \cdot P(h)}{P(D)} = 0 \text{ if } h \text{ is inconsistent with } D$$

The posterior probability of a hypothesis inconsistent with *D* is zero. Now consider the case where *h* is consistent with *D.* Since Equation *(4)* defines *P(D|h)* to be 1 when *h* is consistent with *D*, we have

$$\begin{aligned} P(h|D) &= \frac{1 \cdot \frac{1}{|H|}}{P(D)} \\ &= \frac{1 \cdot \frac{1}{|H|}}{\frac{|VS_{H,D}|}{|H|}} \\ &= \frac{1}{|VS_{H,D}|} \text{ if } h \text{ is consistent with } D \end{aligned}$$

where $VS_{H,D}$ is the subset of hypotheses from H that are consistent with *D* (i.e., $VS_{H,D}$ version space of H with respect to *D* as defined in Chapter 2). It is easy to verify that *P(D)* = $\frac{|VS_{H,D}|}{|H|}$ above, because the sum over all hypotheses of *P(h|D)* must be one and because the number of hypotheses from H consistent with *D* is by definition $|VS_{H,D}|$.

$$P(D) = \sum_{h_i \in H} P(D|h_i) P(h_i)$$

$$= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|} + \sum_{h_i \notin VS_{H,D}} 0 \cdot \frac{1}{|H|}$$

$$= \sum_{h_i \in VS_{H,D}} 1 \cdot \frac{1}{|H|}$$

$$= \frac{|VS_{H,D}|}{|H|}$$

To summarize, Bayes theorem implies that the posterior probability *P(h|D)* under our assumed *P(h)* and *P(D|h)* is

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{if } h \text{ is consistent with } D \\ 0 & \text{otherwise} \end{cases}$$

---------- (5)

4.3.2 MAP Hypotheses and Consistent Learners

The above analysis shows that in the given setting, every hypothesis consistent with *D* is a MAP hypothesis. This statement translates directly into an interesting statement about a general class of learners that we might call *consistent learners.* We will say that a learning algorithm is a *consistent learner* provided it outputs a hypothesis that commits zero errors over the training examples. Given the above analysis, we can conclude that *every consistent learner outputs a MAP hypothesis, if* ***we*** *assume a uniform prior probability distribution over H (i.e., P(hi) = P(hj) for all* ***i,*** *j), and if we assume deterministic, noise free training data (i.e., P(D|h) = 1 if D and h are consistent, and 0 otherwise).*

FIND-S searches the hypothesis space H from specific to general hypotheses, outputting a maximally specific consistent hypothesis (i.e., a maximally specific member of the version space). Because FIND-S outputs a consistent hypothesis, we know that it will output a MAP hypothesis under the probability distributions *P(h)* and *P(D|h)* defined above. Of course FIND-S does not explicitly manipulate probabilities at all-it simply outputs a maximally specific member of the version space. However, by identifying distributions for ***P(h)*** and ***P(D|h)*** under which its output hypotheses will be MAP hypotheses, we have a useful way of characterizing the behavior of FIND-S.

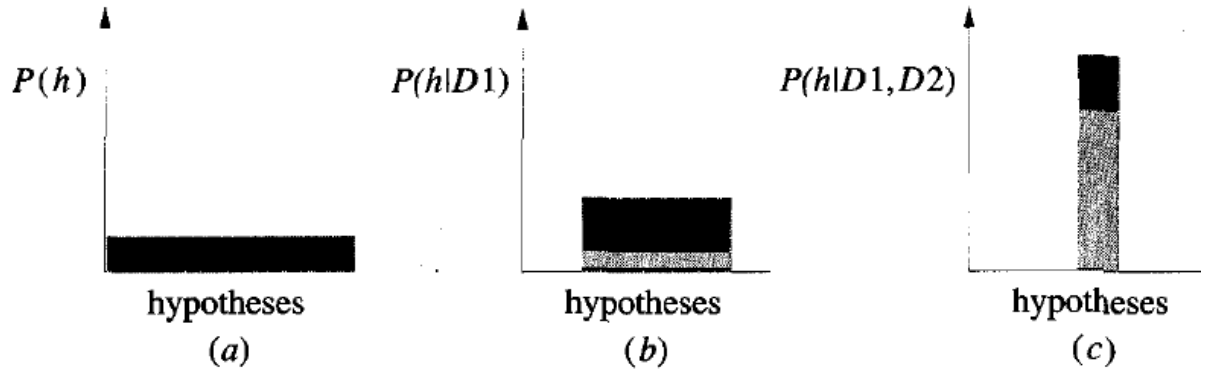


Figure 4.1: Evolution of posterior probabilities *P(h|D)* with increasing training data. *(a)* Uniform priors assign equal probability to each hypothesis. As training data increases first to *Dl* (b), then to *D1^D2* (c), the posterior probability of inconsistent hypotheses becomes zero, while posterior probabilities increase for hypotheses remaining in the version space.

4.4 MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

A straight forward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood hypothesis. Consider any real-valued target function *f*. Training examples <*xi*, *di*>, where *di* is noisy training value *di*= *f*(*xi*) + *ei* where *ei* is random variable (noise) drawn independently for each *xi* according to some Gaussian distribution with mean=0.Then the maximum likelihood hypothesis h_{ML} is the one that minimizes the sum of squared errors:

$$h_{ML} = \arg\min_{h \in H} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

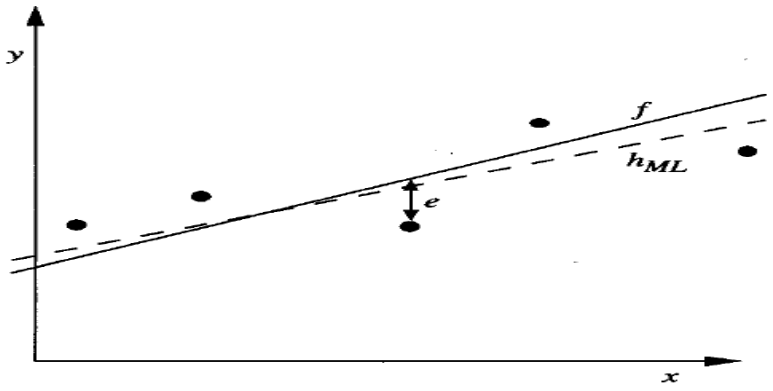


Figure 4.2: Learning a real-valued function.

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} p(D|h)$$
$$= \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} p(d_i|h)$$
$$= \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{d_i - h(x_i)}{\sigma}\right)^2}$$

Maximize natural log of this instead

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2}\left(\frac{d_i - h(x_i)}{\sigma}\right)^2$$
$$= \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} -\frac{1}{2}\left(\frac{d_i - h(x_i)}{\sigma}\right)^2$$
$$= \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} -(d_i - h(x_i))^2$$
$$= \underset{h \in H}{\operatorname{argmin}} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

---------- (6)

4.5 MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

Consider predicting survival probability from patient data

•Training examples <*xi*, *di*>, where *di* is 1 or 0

•Want to train neural network to output a *probability* given *xi* (not a 0 or 1)

Recall that in the maximum likelihood, least-squared error analysis of the previous section, we made the simplifying assumption that the instances ($x_1 \ldots x_m$) were fixed. This enabled us to characterize the data by considering only the target values di. Although we could make a similar simplifying assumption in this case, let us avoid it here in order to demonstrate that it has no impact on the final outcome. Thus treating both xi and di as random variables, and assuming that each training example is drawn independently, we can write P(D|h) as

$$P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h)$$

---------- (7)

It is reasonable to assume, furthermore, that the probability of encountering any particular instance xi is independent of the hypothesis h. For example, the probability that our training set contains a particular ***patient*** xi is independent of our hypothesis about survival rates

(though of course the ***survival d,*** of the patient does depend strongly on ***h).*** When ***x*** is independent of ***h*** we can rewrite the above expression as

$$P(D|h) = \prod_{i=1}^{m} P(x_i, d_i|h) = \prod_{i=1}^{m} P(d_i|h, x_i)P(x_i)$$

---------- (8)

Now what is the probability ***P(di|h, xi)*** of observing ***di*** = ***1*** for a single instance ***xi,*** given a world in which hypothesis ***h*** holds? Recall that ***h*** is our hypothesis regarding the target function, which computes this very probability. Therefore, ***P(di*** = ***1| h, xi)*** = ***h(xi),*** and in general

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases}$$

---------- (9)

In order to substitute this into the Equation ***(6.8)*** for ***P(Dlh),*** let us first
" re-express it in a more mathematically manipulable form, as

In this case can show

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i))$$

Weight update rule for a sigmoid unit:

$$\boxed{w_{jk} \leftarrow w_{jk} + \Delta w_{jk}}$$

where

$$\Delta w_{jk} = \eta \sum_{i=1}^{m} (d_i - h(x_i))\ x_{ijk}$$

4.6 MINIMUM DESCRIPTION LENGTH PRINCIPLE

Recall from Chapter 3 the discussion of Occam's razor, a popular inductive bias that can be summarized as "choose the shortest explanation for the observed data." In that chapter we discussed several arguments in the long-standing debate regarding Occam's razor. Here we consider a Bayesian perspective on this issue and a closely related principle called the Minimum Description Length (MDL) principle. The Minimum Description Length principle is motivated by interpreting the definition of h_{MAP} in the light of basic concepts from information theory. Consider again the now familiar definition of h_{MAP}.

$$h_{MAP} = \underset{h \in H}{\text{argmax}}\ P(D|h)P(h)$$

which can be equivalently expressed in terms of maximizing the $\log_2$

$$h_{MAP} = \underset{h \in H}{\text{argmax}}\ \log_2 P(D|h) + \log_2 P(h)$$

or alternatively, minimizing the negative of this quantity

$$h_{MAP} = \underset{h \in H}{\text{argmin}}\ -\log_2 P(D|h) - \log_2 P(h) \quad \text{---------- (10)}$$

Somewhat surprisingly, Equation (10) can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data. To explain this, let us introduce a basic result from information theory: Consider the problem of designing a code to transmit messages drawn at random, where the probability of encountering message ***i*** is ***pi.*** We are interested here in the most compact code; that is, we are interested in the code that minimizes the expected number of bits we must transmit in order to encode a message drawn at random. Clearly, to minimize the expected code length we should assign shorter codes to messages that are more probable.

The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths. Of course to apply this principle in practice we must choose specific encodings or representations appropriate for the given learning task. Assuming we use the codes C_1 and C_2 to represent the hypothesis and the data given the hypothesis, we can state the MDL principle as

Minimum Description Length principle: Choose h_{MDL} where

$$h_{MDL} = \underset{h \in H}{\text{argmin}}\ L_{C_1}(h) + L_{C_2}(D|h) \quad \text{------> (11)}$$

The above analysis shows that if we choose C_1 to be the optimal encoding of hypotheses C_H, and if we choose C_2 to be the optimal encoding $C_{D|h}$, then $h_{MDL} = h_{MAP}$.

4.7 NAIVE BAYES CLASSIFIER

- Highly bayesian learning method is the naïve Bayes learner often called the naïve Bayes Classifier.
- The naïve Bayes Classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function f(x) can take on any value from some finite set V.

- A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values $<a_1, a_2, a_3, \text{-------} a_n>$. The learner is asked to predict the target value, or classification, for this new instance.
- The Bayesian approach to classifying the new instance is to assign the most probable target value, V_{MAP}, given the attribute values $<a_1, a_2, a_3, \text{-------} a_n>$ that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

- We can use Bayes theorem to rewrite this expression as

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j)$$

Naive Bayes classifier:

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j)$$

4.7.1 An Illustrative Example

Let us apply the naive Bayes classifier to a concept learning problem i.e., classifying days according to whether someone will play tennis. The below table provides a set of 14 training examples of the target concept ***PlayTennis***, where each day is described by the attributes Outlook, Temperature, Humidity, and Wind

Table 1: Play Tennis Dataset

Day	Outlook	Temp.	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Weak	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cold	Normal	Weak	Yes
D10	Rain	Mild	Normal	Strong	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Use the naive Bayes classifier and the training data from this table to classify the following novel instance:

< Outlook = sunny, Temperature = cool, Humidity = high, Wind = strong >

Our task is to predict the target value ***(yes or no)*** of the target concept ***PlayTennis*** for this new instance

$$V_{NB} = \underset{v_j \in \{yes, no\}}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j)$$

$$V_{NB} = \underset{v_j \in \{yes, no\}}{\operatorname{argmax}} P(v_j)\, P(\text{Outlook=sunny}|v_j)\, P(\text{Temperature=cool}|v_j)\, P(\text{Humidity=high}|v_j)\, P(\text{Wind=strong}|v_j)$$

The probabilities of the different target values can easily be estimated based on their frequencies over the 14 training examples

- P(P1ayTennis = yes) = 9/14 = 0.64
- P(P1ayTennis = no) = 5/14 = 0.36

Similarly, estimate the conditional probabilities. For example, those for Wind = strong

- P(Wind = strong | PlayTennis = yes) = 3/9 = 0.33
- P(Wind = strong | PlayTennis = no) = 3/5 = 0.60

Calculate VNB according to Equation (1)

$$P(yes)\ P(sunny|yes)\ P(cool|yes)\ P(high|yes)\ P(strong|yes) = .0053$$

$$P(no)\ P(sunny|no)\ P(cool|no)\ P(high|no)\ P(strong|no) = .0206$$

Thus, the naive Bayes classifier assigns the target value ***PlayTennis*** = ***no*** to this new instance, based on the probability estimates learned from the training data.By normalizing the above quantities to sum to one, calculate the conditional probability that the target value is ***no***, given the observed attribute values

$$\frac{.0206}{(.0206 + .0053)} = .795$$

Estimating Probabilities

We have estimated probabilities by the fraction of times the event is observed to occur over the total number of opportunities. For example, in the above case we estimated P(Wind = strong | Play Tennis = no) by the fraction nc /n where, n = 5 is the total number of training examples for which PlayTennis = no, and nc = 3 is the number of these for which Wind =

strong. When nc = 0, then nc /n will be zero and this probability term will dominate the quantity calculated in Equation (2) requires multiplying all the other probability terms by this zero value. To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the ***m-estimate*** defined as follows

$$\frac{n_c + mp}{n + m}$$

4.8 BAYESIAN BELIEF NETWORKS

- The naive Bayes classifier makes significant use of the assumption that the values of the attributes $a_1 \ldots a_n$ are conditionally independent given the target value v.
- This assumption dramatically reduces the complexity of learning the target function.
- A Bayesian belief network describes the probability distribution governing a set of variables by specifying a set of conditional independence assumptions along with a set of conditional probabilities.
- Bayesian belief networks allow stating conditional independence assumptions that apply to subsets of the variables.

Notation

Consider an arbitrary set of random variables $Y_1 \ldots Yn$, where each variable Yi can take on the set of possible values V(Yi). The joint space of the set of variables Y to be the cross product $V(Y_1)$ x $V(Y_2)$ x. . . V(Yn). In other words, each item in the joint space corresponds to one of the possible assignments of values to the tuple of variables ($Y_1 \ldots$ Yn). The probability distribution over this joint' space is called the joint probability distribution. The joint probability distribution specifies the probability for each of the possible variable bindings for the tuple ($Y_1 \ldots$ Yn). A Bayesian belief network describes the joint probability distribution for a set of variables.

4.8.1 Conditional Independence

Let X, Y, and Z be three discrete-valued random variables. X is conditionally independent of Y given Z if the probability distribution governing X is independent of the value of Y given a value for Z, that is, if

$$(\forall x_i, y_j, z_k)\ P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

Where,

$$x_i \in V(X),\ y_j \in V(Y),\ \text{and}\ z_k \in V(Z).$$

The above expression is written in abbreviated form as

$$P(X \mid Y, Z) = P(X \mid Z)$$

Conditional independence can be extended to sets of variables. The set of variables $X_1 \ldots X_l$ is conditionally independent of the set of variables $Y_1 \ldots Ym$ given the set of variables Z1 . . . Zn if

$$P(X_1 \ldots X_l | Y_1 \ldots Y_m, Z_1 \ldots Z_n) = P(X_1 \ldots X_l | Z_1 \ldots Z_n)$$

The naive Bayes classifier assumes that the instance attribute A1 is conditionally independent of instance attribute A2 given the target value V. This allows the naive Bayes classifier to calculate P(Al, A2 | V) as follows,

$$\begin{aligned} P(A_1, A_2|V) &= P(A_1|A_2, V)P(A_2|V) \\ &= P(A_1|V)P(A_2|V) \end{aligned}$$

Representation

A Bayesian belief network represents the joint probability distribution for a set of variables. Bayesian networks (BN) are represented by directed acyclic graphs.

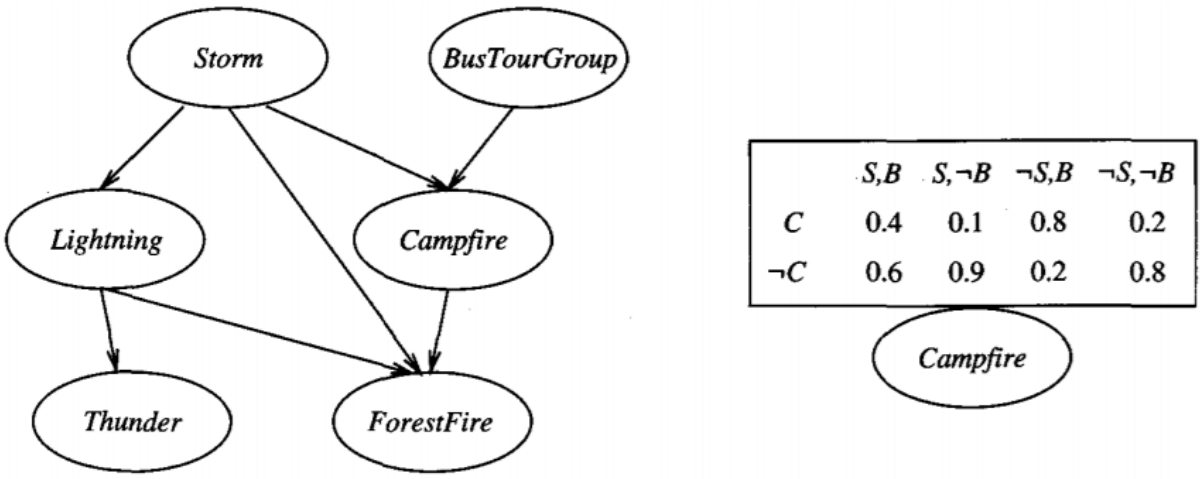


Figure 4.3: Bayesian Belief Network

The Bayesian network in above figure represents the joint probability distribution over the boolean variables *Storm, Lightning, Thunder, ForestFire, Campfire,* and *BusTourGroup*. A Bayesian network (BN) represents the joint probability distribution by specifying a set of *conditional independence assumptions*.

BN represented by a directed acyclic graph, together with sets of local conditional probabilities. Each variable in the joint space is represented by a node in the Bayesian network.

The network arcs represent the assertion that the variable is conditionally independent of its non-descendants in the network given its immediate predecessors in the network. A ***conditional probability table*** **(CPT)** is given for each variable, describing the probability distribution for that variable given the values of its immediate predecessors.

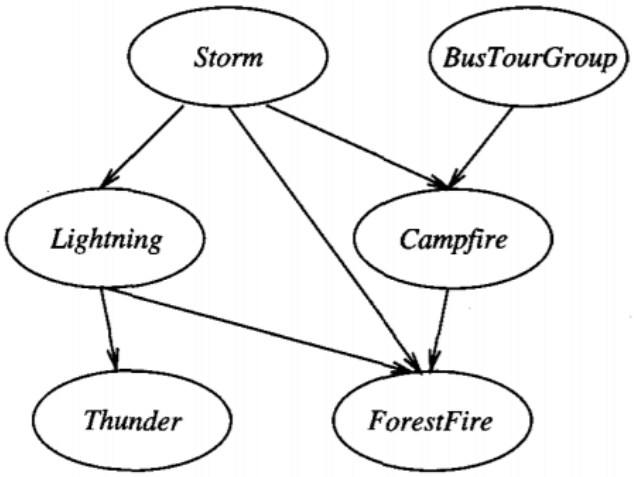
The joint probability for any desired assignment of values (y_1, . . . , yn) to the tuple of network variables (Y_1 . . . Ym) can be computed by the formula

$$P(y_1, \ldots, y_n) = \prod_{i=1}^{n} P(y_i | Parents(Y_i))$$

Where, Parents(Yi) denotes the set of immediate predecessors of Yi in the network.

Example:

Consider the node ***Campfire***. The network nodes and arcs represent the assertion that ***Campfire*** is conditionally independent of its non-descendants ***Lightning*** and ***Thunder***, given its immediate parents Storm and ***BusTourGroup***.



	S,B	S,¬B	¬S,B	¬S,¬B
C	0.4	0.1	0.8	0.2
¬C	0.6	0.9	0.2	0.8

This means that once we know the value of the variables ***Storm*** and ***BusTourGroup***, the variables ***Lightning*** and ***Thunder*** provide no additional information about ***Campfire***. The conditional probability table associated with the variable ***Campfire.*** The assertion is

P(Campfire = True | Storm = True, BusTourGroup = True) = 0.4

Inference

- Use a Bayesian network to infer the value of some target variable (e.g., ForestFire) given the observed values of the other variables. Inference can be straightforward if values for all of the other variables in the network are known exactly.
- A Bayesian network can be used to compute the probability distribution for any subset of network variables given the values or distributions for any subset of the remaining variables. An arbitrary Bayesian network is known to be NP-hard.

4.9 THE EM ALGORITHM

The EM algorithm can be used even for variables whose value is never directly observed, provided the general form of the probability distribution governing these variables is known.

4.9.1 Estimating Means of k Gaussians

Consider a problem in which the data D is a set of instances generated by a probability distribution that is a mixture of k distinct Normal distributions. This problem setting is illustrated in Figure 4.4 for the case where k = 2 and where the instances are the points shown along the x axis.

- Each instance is generated using a two-step process.
- First, one of the k Normal distributions is selected at random.
- Second, a single random instance xi is generated according to this selected distribution.
- This process is repeated to generate a set of data points as shown in the below figure.

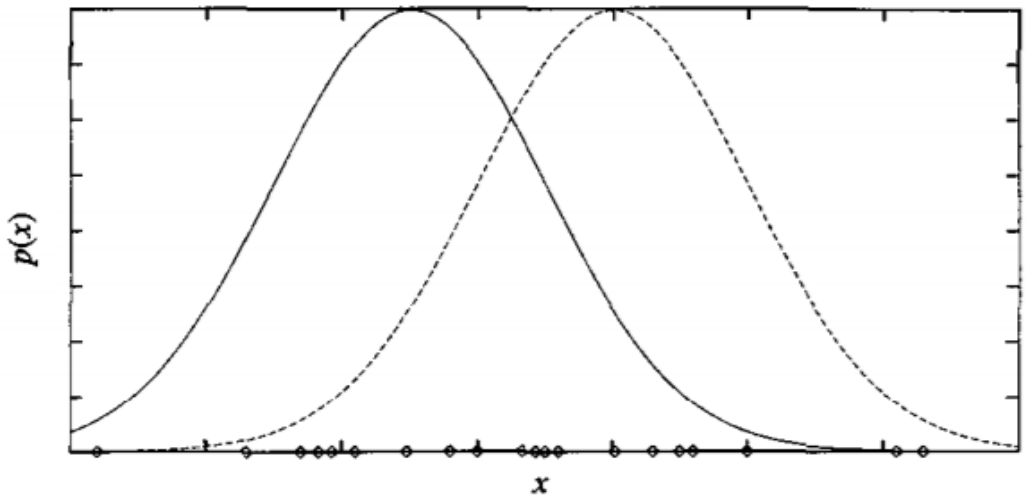


Figure 4.4: Probability Distribution of N data points

To simplify, consider the special case

- The selection of the single Normal distribution at each step is based on choosing each with uniform probability
- Each of the k Normal distributions has the same variance σ2, known value.
- The learning task is to output a hypothesis h = (μ1 , . . . ,μk) that describes the means of each of the k distributions.
- We would like to find a maximum likelihood hypothesis for these means; that is, a hypothesis ***h*** that maximizes p(D |h).

$$\mu_{ML} = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{m} (x_i - \mu)^2 \qquad (1)$$

In this case, the sum of squared errors is minimized by the sample mean

$$\mu_{ML} = \frac{1}{m} \sum_{i=1}^{m} x_i \qquad (2)$$

4.9.2 EM algorithm

Step 1: Calculate the expected value $E[z_{ij}]$ of each hidden variable z_{ij}, assuming the current hypothesis $h = \langle \mu_1, \mu_2 \rangle$ holds.

Step 2: Calculate a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$, assuming the value taken on by each hidden variable z_{ij} is its expected value $E[z_{ij}]$ calculated in Step 1. Then replace the hypothesis $h = \langle \mu_1, \mu_2 \rangle$ by the new hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$ and iterate.

Let us examine how both of these steps can be implemented in practice. Step 1 must calculate the expected value of each z_{ij}. This $E[z_{ij}]$ is just the probability that instance x_i was generated by the jth Normal distribution

$$E[z_{ij}] = \frac{p(x = x_i | \mu = \mu_j)}{\sum_{n=1}^{2} p(x = x_i | \mu = \mu_n)}$$

$$= \frac{e^{-\frac{1}{2\sigma^2}(x_i - \mu_j)^2}}{\sum_{n=1}^{2} e^{-\frac{1}{2\sigma^2}(x_i - \mu_n)^2}}$$

Thus the first step is implemented by substituting the current values $\langle \mu_1, \mu_2 \rangle$ and the observed x_i into the above expression.

In the second step we use the $E[z_{ij}]$ calculated during Step 1 to derive a new maximum likelihood hypothesis $h' = \langle \mu'_1, \mu'_2 \rangle$.
maximum likelihood hypothesis in this case is given by

$$\mu_j \leftarrow \frac{\sum_{i=1}^{m} E[z_{ij}] \; x_i}{\sum_{i=1}^{m} E[z_{ij}]}$$

Acknowledgement

The diagrams and tables are taken from the textbooks specified in the references section.

Prepared by:

Rakshith M D

Department of CS&E

SDMIT, Ujire

MODULE 5: EVALUATING HYPOTHESIS, INSTANCE BASED LEARNING, REINFORCEMENT LEARNING

References
1. Tom M. Mitchell, Machine Learning, India Edition 2013, McGraw Hill Education.

5.1 MOTIVATION

In many cases it is important to evaluate the performance of learned hypotheses as precisely as possible. One reason is simply to understand whether to use the hypothesis. For instance, when learning from a limited-size database indicating the effectiveness of different medical treatments, it is important to understand as precisely as possible the accuracy of the learned hypotheses. A second reason is that evaluating hypotheses is an integral component of many learning methods. For example, in post-pruning decision trees to avoid overfitting, we must evaluate the impact of possible pruning steps on the accuracy of the resulting decision tree.

Therefore it is important to understand the likely errors inherent in estimating the accuracy of the pruned and unpruned tree.

Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

- ***Bias in the estimate.*** First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.
- ***Variance in the estimate.*** Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

5.2 ESTIMATING HYPOTHESIS ACCURACY

When evaluating a learned hypothesis we are most often interested in estimating the accuracy with which it will classify future instances. At the same time, we would like to know the probable error in this accuracy estimate (i.e., what error bars to associate with this estimate).

To illustrate, consider learning the target function "people who plan to purchase new skis this year," given a sample of training data collected by surveying people as they arrive at a ski resort. In this case the instance space X is the space of all people, who might be

described by attributes such as their age, occupation, how many times they skied last year, etc. The distribution D specifies for each person ***x*** the probability that x will be encountered as the next person arriving at the ski resort. The target function f: X -> **{0, 1}** classifies each person according to whether or not they plan to purchase skis this year. Within this general setting we are interested in the following two questions:

1. Given a hypothesis h and a data sample containing n examples drawn at random according to the distribution D, what is the best estimate of the accuracy of h over future instances drawn from the same distribution?
2. What is the probable error in this accuracy estimate?

5.2.1 Sample Error and True Error

To answer these questions, we need to distinguish carefully between two notions of accuracy or, equivalently, error. One is the error rate of the hypothesis over the sample of data that is available. The other is the error rate of the hypothesis over the entire unknown distribution ***D*** of examples. We will call these the ***sample error*** and the ***true error*** respectively. The ***sample error*** of a hypothesis with respect to some sample *S* of instances drawn from X is the fraction of *S* that it misclassifies:

Definition: The **sample error** (denoted $error_S(h)$) of hypothesis h with respect to target function f and data sample S is

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

Where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

The ***true error*** of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution ***D.***

Definition: The **true error** (denoted $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution $\mathcal{D}$, is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

Here the notation $\Pr_{x \in \mathcal{D}}$ denotes that the probability is taken over the instance distribution V.

5.2.2 Confidence Intervals for Discrete-Valued Hypothesis

Here we give an answer to the question "How good an estimate of $error_D(h)$ is provided by $error_s(h)$?" for the case in which ***h*** is a discrete-valued hypothesis. More specifically,

suppose we wish to estimate the true error for some discrete valued hypothesis ***h,*** based on its observed sample error over a sample S, where

- the sample ***S*** contains ***n*** examples drawn independent of one another, and independent of ***h,*** according to the probability distribution ***D***
- n>=30
- hypothesis ***h*** commits ***r*** errors over these ***n*** examples (i.e., $error_s(h) = r/n$).

Under these conditions, statistical theory allows us to make the following assertions:

1. Given no other information, the most probable value of $error_D(h)$ is $error_s(h)$

2. With approximately ***95%*** probability, the true error $error_D(h)$ lies in the interval

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

To illustrate, suppose the data sample ***S*** contains ***n*** = ***40*** examples and that hypothesis ***h*** commits ***r*** = ***12*** errors over this data. In this case, the sample error $error_s(h) = 12/40 = .30$. Given no other information, the best estimate of the true error $error_D(h)$ is the observed sample error ***.30.***

TABLE 5.1: Values of z_N for two-sided N% confidence intervals

Confidence level $N\%$:	50%	68%	80%	90%	95%	98%	99%
Constant z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

The above expression for the ***95%*** confidence interval can be generalized to any desired confidence level. The constant ***1.96*** is used in case we desire a ***95%*** confidence interval. ***A*** different constant, Z_N, is used to calculate the ***N%*** confidence interval. The general expression for approximate ***N%*** confidence intervals for $error_D(h)$ is

$$error_S(h) \pm z_N\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \quad \text{--------- (5.1)}$$

where the constant Z_N is chosen depending on the desired confidence level, using the values of z_N given in Table ***5.1***

5.3 BASICS OF SAMPLING THEORY

This section introduces basic notions from statistics and sampling theory, including probability distributions, expected value, variance, Binomial and Normal distributions, and two-sided and one-sided intervals. A basic familiarity with these concepts is important to understanding how to evaluate hypotheses and learning algorithms. Even more important,

these same notions provide an important conceptual framework for understanding machine learning issues such as overfitting and the relationship between successful generalization and the number of training examples considered. The key concepts introduced in this section are summarized in Table *5.2.*

TABLE 5.2: Basic definitions and facts from statistics

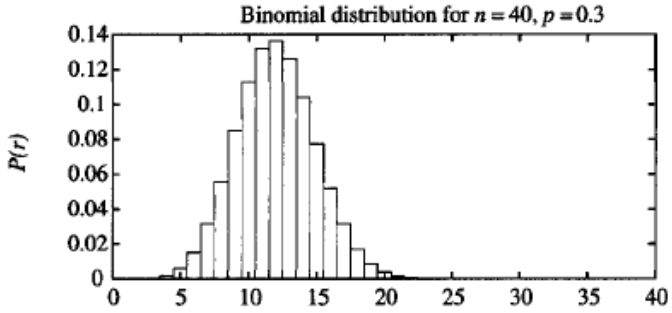
- A *random variable* can be viewed as the name of an experiment with a probabilistic outcome. Its value is the outcome of the experiment.
- A *probability distribution* for a random variable Y specifies the probability $\Pr(Y = y_i)$ that Y will take on the value y_i, for each possible value y_i.
- The *expected value*, or *mean*, of a random variable Y is $E[Y] = \sum_i y_i \Pr(Y = y_i)$. The symbol μ_Y is commonly used to represent E[Y].
- The *variance* of a random variable is $Var(Y) = E[(Y - \mu_Y)^2]$. The variance characterizes the width or dispersion of the distribution about its mean.
- The *standard deviation* of Y is $\sqrt{Var(Y)}$. The symbol σ_Y is often used used to represent the standard deviation of Y.
- The *Binomial distribution* gives the probability of observing r heads in a series of n independent coin tosses, if the probability of heads in a single toss is p.
- The *Normal distribution* is a bell-shaped probability distribution that covers many natural phenomena.
- The *Central Limit Theorem* is a theorem stating that the sum of a large number of independent, identically distributed random variables approximately follows a Normal distribution.
- An *estimator* is a random variable Y used to estimate some parameter p of an underlying population.
- The *estimation bias* of Y as an estimator for p is the quantity $(E[Y] - p)$. An unbiased estimator is one for which the bias is zero.
- A *N% confidence interval* estimate for parameter p is an interval that includes p with probability $N\%$.

5.3.1 Error Estimation and Estimating Binomial Proportions

Precisely how does the deviation between sample error and true error depend on the size of the data sample? This question is an instance of a well-studied problem in statistics: the problem of estimating the proportion of a population that exhibits some property, given the observed proportion over some random sample of the population. In our case, the property of interest is that h misclassifies the example.

The key to answering this question is to note that when we measure the sample error we are performing an experiment with a random outcome. We first collect a random sample **S** of ***n*** independently drawn instances from the distribution ***D,*** and then measure the sample error $\boldsymbol{error_S(h)}$**.** As noted in the previous section, if we were to repeat this experiment many times, each time drawing a different random sample $\boldsymbol{S_i}$ of size n, we would expect to observe different values for the various errors,(h), depending on random differences in the makeup of the various $\boldsymbol{S_i}$.

The table 5.3 describes a particular probability distribution called the Binomial distribution.



A *Binomial distribution* gives the probability of observing r heads in a sample of n independent coin tosses, when the probability of heads on a single coin toss is p. It is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

If the random variable X follows a Binomial distribution, then:

- The probability $\Pr(X = r)$ that X will take on the value r is given by $P(r)$
- The expected, or mean value of X, $E[X]$, is

$$E[X] = np$$

- The variance of X, $Var(X)$, is

$$Var(X) = np(1-p)$$

- The standard deviation of X, σ_X, is

$$\sigma_X = \sqrt{np(1-p)}$$

For sufficiently large values of n the Binomial distribution is closely approximated by a Normal distribution (see Table 5.4) with the same mean and variance. Most statisticians recommend using the Normal approximation only when $np(1-p) \geq 5$.

5.3.2 The Binomial Distribution

A good way to understand the Binomial distribution is to consider the following problem. You are given a worn and bent coin and asked to estimate the probability that the coin will turn up heads when tossed. Let us call this unknown probability of heads p. You toss the coin n times and record the number of times r that it turns up heads. A reasonable estimate of p is r/n. The general setting to which the Binomial distribution applies is:

1. There is a base, or underlying, experiment (e.g., toss of the coin) whose outcome can be described by a random variable, say *Y*. The random variable *Y* can take on two possible values (e.g., *Y* = 1 if heads, Y = 0 if tails).
2. The probability that *Y* = 1 on any single trial of the underlying experiment is given by some constant p, independent of the outcome of any other experiment. The probability that *Y* = 0 is therefore (1 - p). Typically, p is not known in advance, and the problem is to estimate it.

3. A series of n independent trials of the underlying experiment is performed (e.g., n independent coin tosses), producing the sequence of independent, identically distributed random variables $Y_1, Y_2, \ldots, Yn$. Let R denote the number of trials for which $\boldsymbol{Y_i} = 1$ in this series of n experiments

$$R \equiv \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable $\boldsymbol{R}$ will take on a specific value r (e.g., the probability of observing exactly r heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \qquad \text{------- (5.2)}$$

5.3.3 Mean and Variance

Two properties of a random variable that are often of interest are its expected value (also called its mean value) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable. More precisely

Definition: Consider a random variable Y that takes on the possible values
The **expected value** of Y, $E[Y]$, is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \Pr(Y = y_i) \qquad \text{------ (5.3)}$$

For example, if Y takes on the value 1 with probability .7 and the value 2 with probability .3, then its expected value is (1*0.7 + 2*0.3 = 1.3). In case the random variable Y is governed by a Binomial distribution, then it can be shown that

$$E[Y] = np \qquad \text{------ (5.4)}$$

where n and p are the parameters of the Binomial distribution defined in Equation(5.2).

A second property, the variance, captures the "width or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

Definition: The **variance** of a random variable Y, $Var[Y]$, is

$$Var[Y] \equiv E[(Y - E[Y])^2] \qquad \text{---------- (5.5)}$$

The variance describes the expected squared error in using a single observation of Y to estimate its mean $E[Y]$. The square root of the variance is called the *standard deviation* of Y, denoted σ_Y.

Definition: The **standard deviation** of a random variable Y, σ_Y, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]}$$

------ (5.6)

In case the random variable Y is governed by a Binomial distribution, then the variance and standard deviation are given by

$$Var[Y] = np(1-p)$$

$$\sigma_Y = \sqrt{np(1-p)}$$

-------- (5.7)

5.4 A GENERAL APPROACH FOR DERIVING CONFIDENCE INTERVALS

The previous section described in detail how to derive confidence interval estimates for one particular case: estimating $error_D(h)$ for a discrete-valued hypothesis ***h,*** based on a sample of ***n*** independently drawn instances. The approach described there illustrates a general approach followed in many estimation problems. In particular, we can see this as a problem of estimating the mean (expected value) of a population based on the mean of a randomly drawn sample of size ***n.***

The general process includes the following steps:

1. Identify the underlying population parameter p to be estimated, for example, $error_D(h)$.
2. Define the estimator Y (e.g., $error_s(h)$). It is desirable to choose a minimum variance, unbiased estimator.
3. Determine the probability distribution ***Dy*** that governs the estimator Y, including its mean and variance.
4. Determine the ***N%*** confidence interval by finding thresholds L and ***U*** such that ***N%*** of the mass in the probability distribution ***Dy*** falls between L and U.

5.4.1 Central Limit Theorem

One essential fact that simplifies attempts to derive confidence intervals is the Central Limit Theorem. Consider again our general setting, in which we observe the values of ***n*** independently drawn random variables $Y_1 \ldots Y_n$ that obey the same unknown underlying probability distribution (e.g., ***n*** tosses of the same coin).

Theorem 5.1. Central Limit Theorem. Consider a set of independent, identically distributed random variables $Y_1 \ldots Y_n$ governed by an arbitrary probability distribution with mean μ and finite variance σ^2. Define the sample mean, $\bar{Y}_n \equiv \frac{1}{n}\sum_{i=1}^{n} Y_i$.

Then as $n \to \infty$, the distribution governing

$$\frac{\bar{Y}_n - \mu}{\frac{\sigma}{\sqrt{n}}}$$

approaches a Normal distribution, with zero mean and standard deviation equal to 1.

5.5 DIFFERENCE IN ERROR OF TWO HYPOTHESES

Consider the case where we have two hypotheses $h_1$and h_2 for some discrete-valued target function. Hypothesis h_1 has been tested on a sample S_1 containing n_l randomly drawn examples, and h_2 has been tested on an independent sample S_2 containing n_2 examples drawn from the same distribution. Suppose we wish to estimate the difference d between the true errors of these two hypothesis.

$$d \equiv error_{\mathcal{D}}(h_1) - error_{\mathcal{D}}(h_2)$$

We will use the generic four-step procedure described at the beginning of Section 5.4 to derive a confidence interval estimate for d. Having identified d as the parameter to be estimated, we next define an estimator. The obvious choice for an estimator in this case is the difference between the sample errors, which we denote by $\hat{d}$

$$\hat{d} \equiv error_{S_1}(h_1) - error_{S_2}(h_2)$$

Although we will not prove it here, it can be shown that $\hat{d}$ gives an unbiased estimate of d; that is $E[\hat{d}\,] = d$.

What is the probability distribution governing the random variable $\hat{d}$ ***?*** From earlier sections, we know that for large ***nl*** and ***n2*** (e.g., both >= ***30),*** both ***error***$_{s1}$ ***(h1)*** and ***error***$_{s2}$ ***(h2)*** follow distributions that are approximately Normal. Because the difference of two Normal distributions is also a Normal distribution, $\hat{d}$ will also follow a distribution that is approximately Normal, with mean d. It can also be shown that the variance of this distribution is the sum of the variances of ***error***$_{s1}$ ***(h1)*** and ***error***$_{s2}$ ***(h2)***. Using Equation ***(5.9)*** to obtain the approximate variance of each of these distributions, we have

$$\sigma_{error_S(h)} \approx \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

$$\sigma_{\hat{d}}^2 \approx \frac{error_{S_1}(h_1)(1-error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1-error_{S_2}(h_2))}{n_2} \quad \text{--------- (5.12)}$$

Now that we have determined the probability distribution that governs the estimator $\hat{d}$, it is straightforward to derive confidence intervals that characterize the likely error in employing $\hat{d}$ to estimate d. For a random variable $\hat{d}$ obeying a Normal distribution with mean d and variance σ^2, the N% confidence interval estimate for d is $\hat{d} \pm z_N \sigma$. Using the approximate variance $\sigma_{\hat{d}}^2$ given above, this approximate ***N%*** confidence interval estimate for d is

$$\hat{d} \pm z_N \sqrt{\frac{error_{S_1}(h_1)(1-error_{S_1}(h_1))}{n_1} + \frac{error_{S_2}(h_2)(1-error_{S_2}(h_2))}{n_2}} \quad \text{--- (5.13)}$$

Although the above analysis considers the case in which $\mathbf{h_1}$ and $\mathbf{h_2}$ are tested on independent data samples, it is often acceptable to use the confidence interval seen in Equation ***(5.13)*** in the setting where $\mathbf{h_1}$ and $\mathbf{h_2}$ are tested on a single sample ***S*** (where ***S*** is still independent of h_1 and h_2**).** In this later case, we redefine $\hat{d}$ as

$$\hat{d} \equiv error_S(h_1) - error_S(h_2)$$

5.6 COMPARING LEARNING ALGORITHMS

- Which of L_A and L_B is the better learning method on average for learning some particular target function f ?
- To answer this question, we wish to estimate the expected value of the difference in their errors:

$$E_{S \subset D}[error_D(L_A(S)) - error_D(L_B(S))]$$

- Of course, since we have only a limited sample D_0 we estimate this quantity by dividing D_0 into a *training set* S_0 and a *testing set* T_0 and measure:

$$error_{T0}(L_A(S_0)) - error_{T0}(L_B(S_0))$$

A procedure to estimate the difference in error between two learning methods L_A and L_B is given in table below.

1. Partition the available data D_0 into k disjoint subsets $T_1, T_2, \ldots, T_k$ of equal size, where this size is at least 30.
2. For i from 1 to k, do
   use T_i for the test set, and the remaining data for training set S_i
   - $S_i \leftarrow \{D_0 - T_i\}$
   - $h_A \leftarrow L_A(S_i)$
   - $h_B \leftarrow L_B(S_i)$
   - $\delta_i \leftarrow error_{T_i}(h_A) - error_{T_i}(h_B)$
3. Return the value $\bar{\delta}$, where

$$\bar{\delta} \equiv \frac{1}{k} \sum_{i=1}^{k} \delta_i$$

5.7 INTRODUCTION OF INSTANCE BASED LEARNING

Instance-based learning methods such as nearest neighbour and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions. Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the new query instance. Instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified.

Advantages of Instance-based learning

1. Training is very fast

2. Learn complex target function

3. Don’t lose information

Disadvantages of Instance-based learning

1. The cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered.

2. In many instance-based approaches, especially nearest-neighbour approaches, is that they typically consider all attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most "similar" may well be a large distance apart.

5.8 k- NEAREST NEIGHBOR LEARNING

- The most basic instance-based method is the K- Nearest Neighbor Learning. This algorithm assumes all instances correspond to points in the n-dimensional space Rn.
- The nearest neighbors of an instance are defined in terms of the standard Euclidean distance.
- Let an arbitrary instance x be described by the feature vector

  ((a1(x), a2(x), ……, an(x))

  Where, ar(x) denotes the value of the r^{th} attribute of instance x.
- Then the distance between two instances xi and xj is defined to be d(xi , xj)

  Where,

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^{n} (a_r(x_i) - a_r(x_j))^2}$$

- In nearest-neighbor learning the target function may be either discrete-valued or real-valued.

Let us first consider learning ***discrete-valued target functions*** of the form

$$f : \Re^n \rightarrow V.$$

Where, V is the finite set {v1, . . . vs }

The k- Nearest Neighbor algorithm for approximation a **discrete-valued target function** is given below:

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

- The value f'(xq) returned by this algorithm as its estimate of f(xq) is just the most common value of f among the k training examples nearest to xq.
- If k = 1, then the 1- Nearest Neighbor algorithm assigns to f'(xq) the value f(xi). Where xi is the training instance nearest to xq.

- For larger values of k, the algorithm assigns the most common value among the k nearest training examples.
- Below figure illustrates the operation of the k-Nearest Neighbor algorithm for the case where the instances are points in a two-dimensional space and where the target function is Boolean valued.

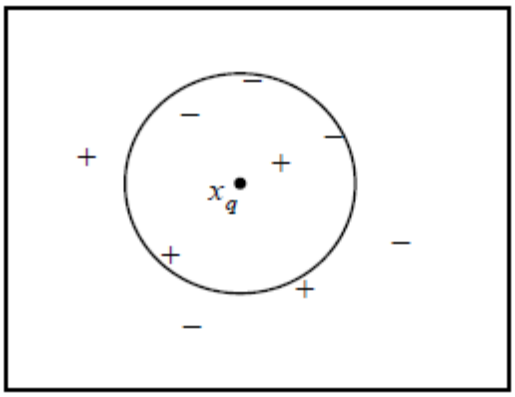


Figure 5.1: Operation of k-Nearest Neighbor algorithm

- The positive and negative training examples are shown by "+" and "-" respectively. A query point x_q is shown as well.
- The 1-Nearest Neighbor algorithm classifies x_q as a positive example in this figure, whereas the 5-Nearest Neighbor algorithm classifies it as a negative example.
- Below figure shows the shape of this **decision surface** induced by 1- Nearest Neighbor over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples.

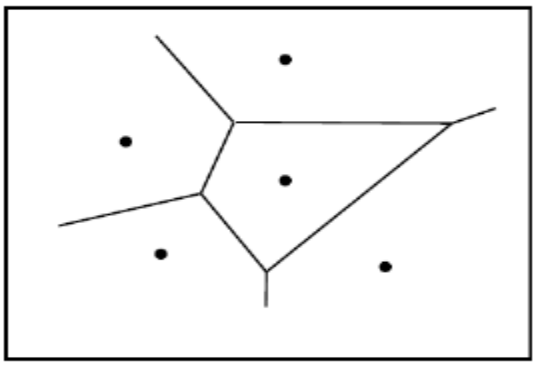


Figure 5.2: Decision surface induced by 1- Nearest Neighbor

For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the ***Voronoi diagram*** of the set of training example.

The K- Nearest Neighbor algorithm for approximation a **real-valued target function** is given below $f : \Re^n \rightarrow \Re$

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Distance-Weighted Nearest Neighbor Algorithm

- The refinement to the k-NEAREST NEIGHBOR Algorithm is to weight the contribution of each of the k neighbors according to their distance to the query point x_q, giving greater weight to closer neighbors.
- For example, in the k-Nearest Neighbor algorithm, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q.

Distance-Weighted Nearest Neighbor Algorithm for approximation a discrete-valued target functions is given below:

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \underset{v \in V}{\operatorname{argmax}} \sum_{i=1}^{k} w_i \delta(v, f(x_i))$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

Distance-Weighted Nearest Neighbor Algorithm for approximation a Real-valued target functions

---

Training algorithm:

- For each training example $\langle x, f(x) \rangle$, add the example to the list *training_examples*

Classification algorithm:

- Given a query instance x_q to be classified,
  - Let $x_1 \ldots x_k$ denote the k instances from *training_examples* that are nearest to x_q
  - Return

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

Terminology

- **Regression** means approximating a real-valued target function.
- **Residual** is the error $f'(x) - f(x)$ in approximating the target function.
- **Kernel function** is the function of distance that is used to determine the weight of each training example. In other words, the kernel function is the function K such that $w_i = K(d(x_i, x_q))$.

5.9 LOCALLY WEIGHTED REGRESSION

The phrase "**locally weighted regression**" is called **local** because the function is approximated based only on data near the query point, **weighted** because the contribution of each training example is weighted by its distance from the query point, and **regression** because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions. Given a new query instance x_q, the general approach in locally weighted regression is to construct an approximation $\hat{f}$ that fits the training examples in the neighborhood surrounding x_q. This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance.

Locally Weighted Linear Regression

Consider locally weighted regression in which the target function ***f*** is approximated near xq using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

Where, ai(x) denotes the value of the i^{th} attribute of the instance x

Derived methods are used to choose weights that minimize the squared error summed over the set D of training examples using gradient descent

$$E \equiv \frac{1}{2}\sum_{x \in D}(f(x) - \hat{f}(x))^2$$

Which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D}(f(x) - \hat{f}(x))a_j(x)$$

Where, η is a constant learning rate

Need to modify this procedure to derive a local approximation rather than a global one. The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below.

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$ ---------- **(1)**

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from xq:

$$E_2(x_q) \equiv \frac{1}{2}\sum_{x \in D}(f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$ ------ **(2)**

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 \; K(d(x_q, x))$$ ------- **(3)**

If we choose criterion three and re-derive the gradient descent rule, we obtain the following training rule

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x)) \; (f(x) - \hat{f}(x)) \; a_j(x)$$

The differences between this new rule and the rule given by Equation (3) are that the contribution of instance x to the weight update is now multiplied by the distance penalty **K(d(xq, x)),** and that the error is summed over only the k nearest training examples.

5.10 RADIAL BASIS FUNCTIONS

One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions. In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u K_u(d(x_u, x)) \quad \text{----- (1)}$$

- Where, each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases.
- Here k is a user provided constant that specifies the number of kernel functions to be included.
- $\hat{f}$ is a global approximation to f(x), the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point x_u.

Choose each function $K_u(d(x_u, x))$ to be a Gaussian function centred at the point x_u with some variance σ_u^2

$$K_u(d(x_u, x)) = e^{\frac{1}{2\sigma_u^2} d^2(x_u, x)}$$

The functional form of equation (1) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width σ^2 of each kernel can be separately specified. The function given by equation (1) can be viewed as describing a two layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values.

Example: Radial basis function (RBF) network

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process.

1. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 that define its kernel function $K_u(d(x_u, x))$

2. Second, the weights w, are trained to maximize the fit of the network to the training data, using the global error criterion given by

$$E \equiv \frac{1}{2}\sum_{x \in D}(f(x) - \hat{f}(x))^2$$

Because the kernel functions are held fixed during this second stage, the linear weight values w, can be trained very efficiently

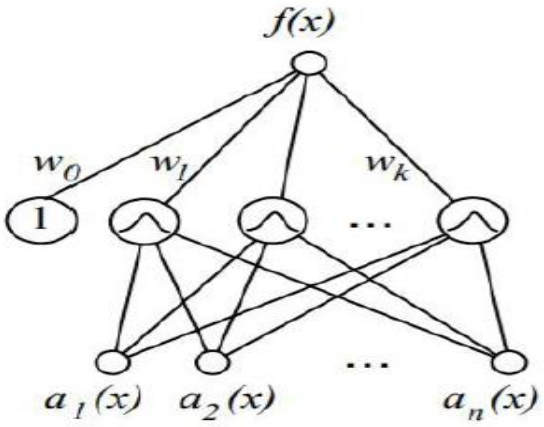


Figure 5.3: Radial basis function (RBF) network

Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions.

- One approach is to allocate a Gaussian kernel function for each training example (x_i, f (x_i)), centring this Gaussian at the point x_i.
- Each of these kernels may be assigned the same width σ^2. Given this approach, the RBF network learns a global approximation to the target function in which each training example (x_i, f (x_i)) can influence the value of ***f*** only in the neighbourhood of x_i.
- A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large.

5.11 CASE-BASED REASONING

Case-based reasoning (CBR) is a learning paradigm based on lazy learning methods and they classify new query instances by analysing similar instances while ignoring instances that are very different from the query.

- In CBR represent instances are not represented as real-valued points, but instead, they use a ***rich symbolic*** representation.
- CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs, reasoning about new legal cases based

on previous rulings, and solving planning and scheduling problems by reusing and combining portions of previous solutions to similar problems.

A prototypical example of a case-based reasoning

- The CADET system employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets.
- It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems.
- Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function.
- New design problems are then presented by specifying the desired function and requesting the corresponding structure.

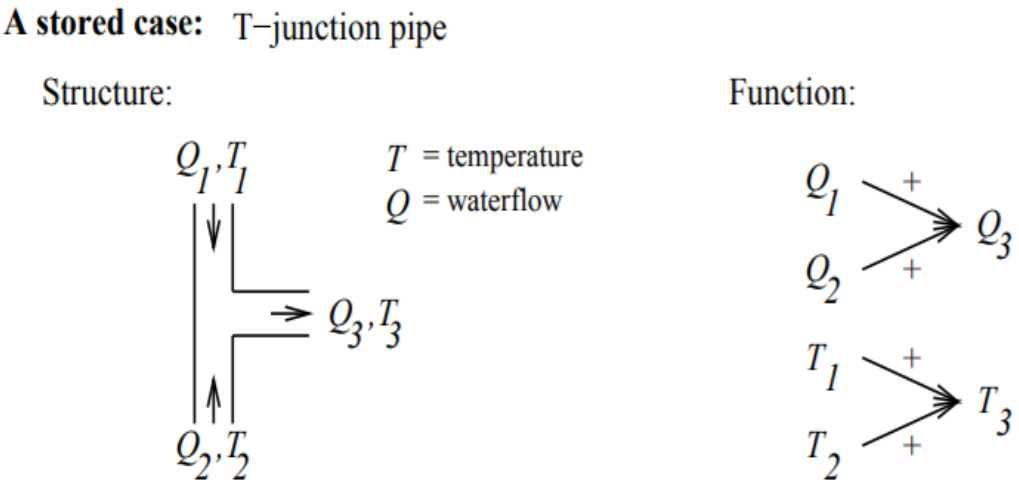


Figure 5.4: CADET system

- The function is represented in terms of the qualitative relationships among the water-flow levels and temperatures at its inputs and outputs.
- In the functional description, an arrow with a "+" label indicates that the variable at the arrowhead increases with the variable at its tail. A "-" label indicates that the variable at the head decreases with the variable at the tail.
- Here Qc refers to the flow of cold water into the faucet, Q_h to the input flow of hot water, and Qm to the single mixed flow out of the faucet.
- Tc, T_h, and Tm refer to the temperatures of the cold water, hot water, and mixed water respectively.
- The variable C_t denotes the control signal for temperature that is input to the faucet, and C_f denotes the control signal for water flow.

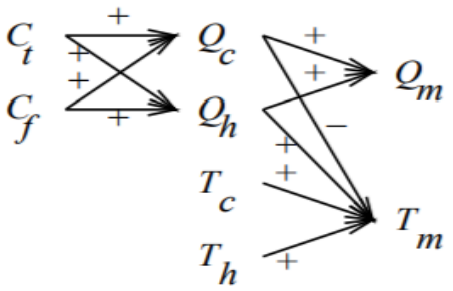
- The controls C_t and C_f are to influence the water flows Qc and Q_h, thereby indirectly influencing the faucet output flow Qm and temperature Tm.

A problem specification: Water faucet

Structure:

?

Function:



5.12 INTRODUCTION ON REINFORCEMENT LEARNING

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.Consider building a **learning robot**. The robot, or ***agent***, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state. Its task is to learn a control strategy, or ***policy***, for choosing actions that achieve its goals. The goals of the agent can be defined by a ***reward function*** that assigns a numerical value to each distinct action the agent may take from each distinct state. This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot. The **task** of the robot is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

Example:

A mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn." The robot may have a goal of docking onto its battery charger whenever its battery level is low. The goal of docking to the battery charger can be captured by assigning a positive reward (Eg., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

Reinforcement Learning Problem

An agent interacting with its environment. The agent exists in an environment described by some set of possible states S. Agent perform any of a set of possible actions A. Each time it performs an action a, in some state s_t the agent receives a real-valued reward r, that indicates the immediate value of this state-action transition. This produces a sequence of states s_i,

actions a_i, and immediate rewards r_i as shown in the figure. The agent's task is to learn a control policy, π: S → A,that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

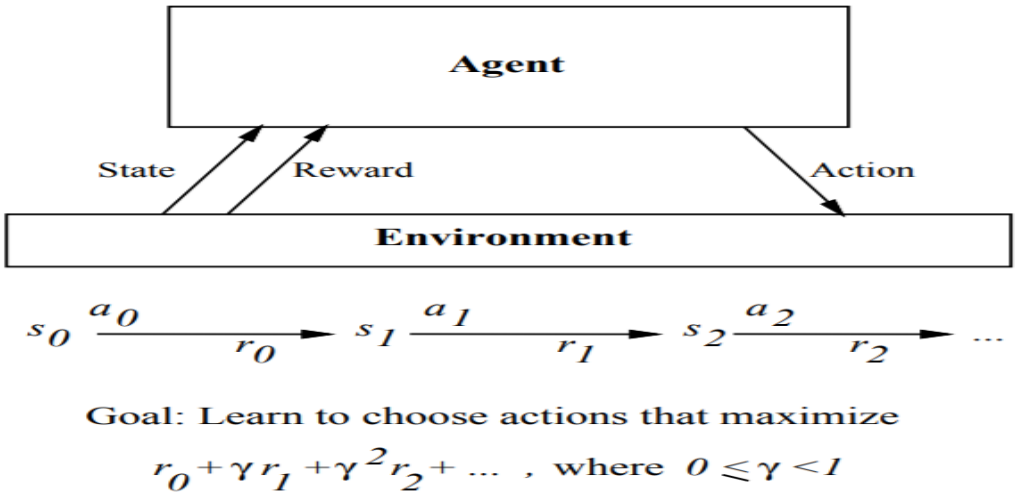


Figure 5.5: Reinforcement Learning Problem

5.13 THE LEARNING TASK

Consider Markov decision process (MDP) where the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform. At each discrete time step t, the agent senses the current state s_t, chooses a current action a_t, and performs it. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions.

The task of the agent is to learn a policy, π: S → A, for selecting its next action a, based on the current observed state s_t; that is, $\pi(s_t) = a_t$.

How shall we specify precisely which policy π we would like the agent to learn?

1. One approach is to require the policy that produces the greatest possible ***cumulative reward*** for the robot over time. To state this requirement more precisely, define the cumulative value $V^{\pi}(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

- ------- (1)

Where, the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions. Here $0 \leq \gamma \leq 1$ is a constant that determines the relative value of delayed versus immediate rewards. if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward. The quantity $V^{\pi}(s_t)$ is called the ***discounted cumulative reward*** achieved by policy π from initial state ***s***. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later.

2. Other definitions of total reward is ***finite horizon reward,***

$$\sum_{i=0}^{h} r_{t+i}$$

Considers the undiscounted sum of rewards over a finite number ***h*** of steps

3. Another approach is ***average reward***

$$\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

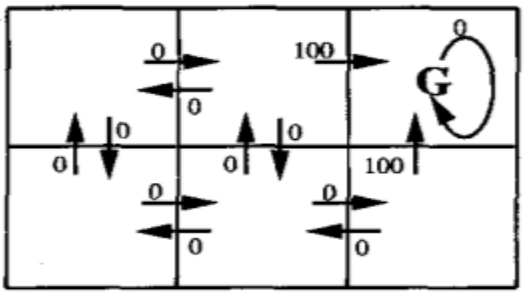
Considers the average reward per time step over the entire lifetime of the agent.

We require that the agent learn a policy π that maximizes $V^{\pi}(s_t)$ for all states s. such a policy is called an ***optimal policy*** and denote it by π^*

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}}\, V^{\pi}(s), (\forall s) \quad \text{--------- (2)}$$

Refer the value function $V^{\pi^*}(s)$ an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state ***s***.

Example: A simple grid-world environment is depicted in the diagram

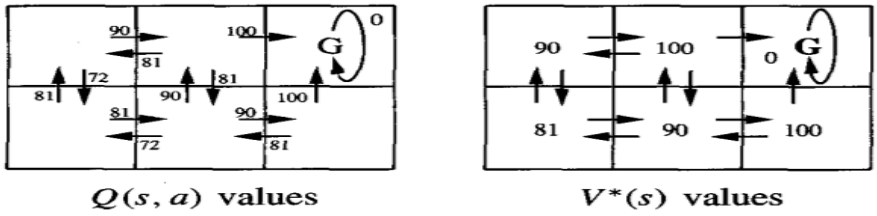


$r(s, a)$ (immediate reward) values

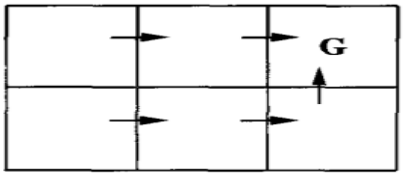
Figure 5.5: A simple grid-world environment

The six grid squares in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another. The number associated with each arrow represents the immediate reward ***r(s, a)*** the agent receives if it executes the corresponding state-action transition. The immediate reward in this environment is defined to be zero for all state-action transitions except for those leading into the state labelled G. The state G as the goal state, and the agent can receive reward by entering this state. Once the states, actions, and immediate rewards are defined, choose a value for the discount factor γ, determine the optimal policy π* and its value function V*(s).

Let's choose γ = 0.9. The diagram at the bottom of the figure shows one optimal policy for this setting.



$Q(s, a)$ values $V^*(s)$ values



One optimal policy

Values of V*(s) and Q(s, a) follow from r(s, a), and the discount factor γ = 0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown. The discounted future reward from the bottom centre state is

$$0+ \gamma\ 100+ \gamma^2\ 0+ \gamma^3\ 0+... = 90$$

5.14 Q LEARNING

1) How can an agent learn an optimal policy π * for an arbitrary environment?

The training information available to the learner is the sequence of immediate rewards $r(s_i,a_i)$ for i = 0,1,2, ... Given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

2) What evaluation function should the agent attempt to learn?

One obvious choice is V*. The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1. The optimal action in state ***s*** is the action ***a*** that maximizes the sum of the immediate reward r(s, a) plus the value V* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}[r(s,a) + \gamma V^*(\delta(s,a))] \quad \text{------- (3)}$$

The *Q* Function

The value of Evaluation function ***Q(s, a)*** is the reward received immediately upon executing action a from state ***s***, plus the value (discounted by ***γ***) of following the optimal policy thereafter

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \quad \text{---------- (4)}$$

Rewrite Equation (3) in terms of ***Q(s, a)*** as

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \quad \text{---------- (5)}$$

Equation (5) makes clear, it need only consider each available action ***a*** in its current state ***s*** and choose the action that maximizes ***Q(s, a)***.

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s,a)$ to zero.
Observe the current state s
Do forever:

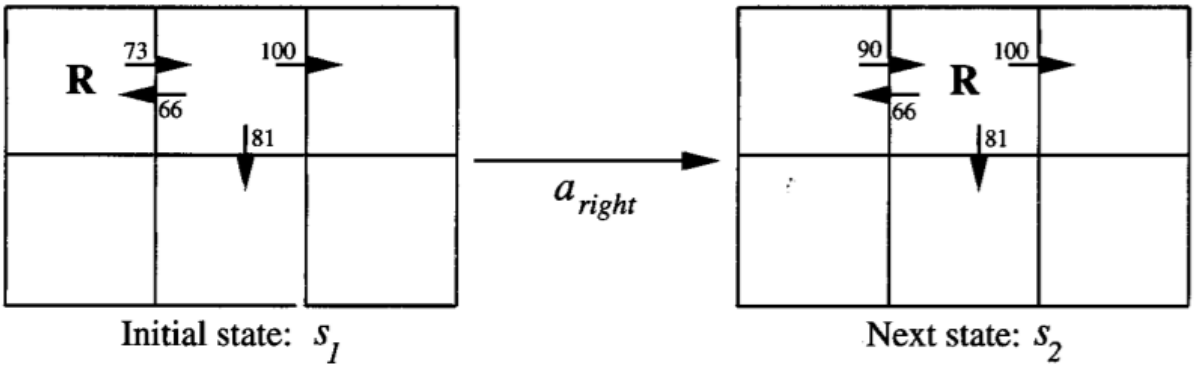
- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- $s \leftarrow s'$

An Illustrative Example

To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to Q shown in below figure



The agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. Apply the training rule of Equation to refine its estimate Q for the state-action transition it just executed.

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

According to the training rule, the new Q estimate for this transition is the sum of the received reward (zero) and the highest Q value associated with the resulting state (100), discounted by γ (.9).

$$\begin{aligned} \hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{66, 81, 100\} \\ &\leftarrow 90 \end{aligned}$$

Acknowledgement

The diagrams and tables are taken from the textbooks specified in the references section.

Prepared by:

Rakshith M D

Department of CS&E

SDMIT, Ujire