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.**

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

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,

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

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 $\hat{V}$

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

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

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

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.

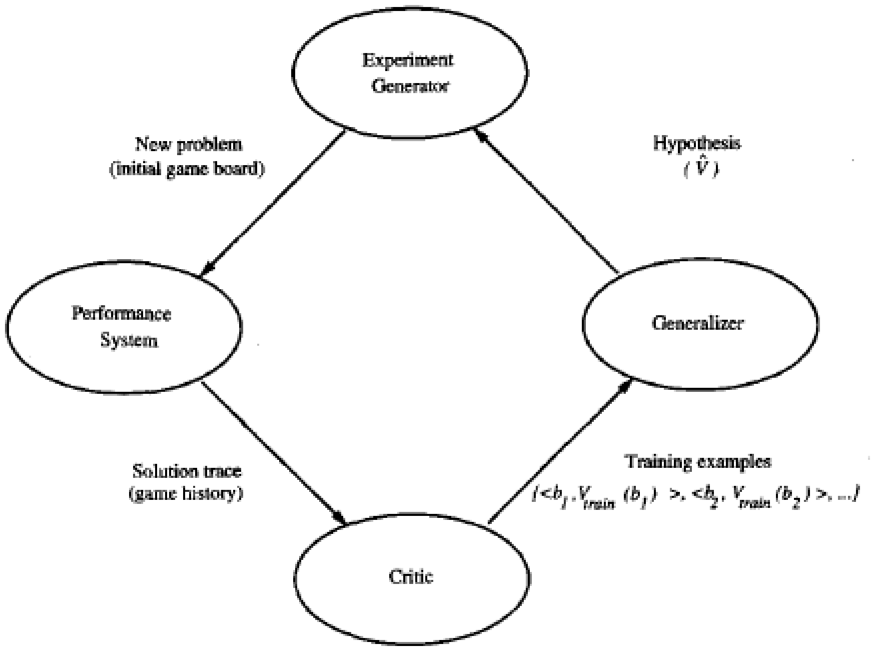

Experiment Generator
New problem (initial game board)
Hypothesis ($\hat{V}$)
Performance System
Generalizer
Solution trace (game history)
Training examples
$\{<b_1, V_{train}(b_1)>, <b_2, V_{train}(b_2)>, ...\}$
Critic

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.

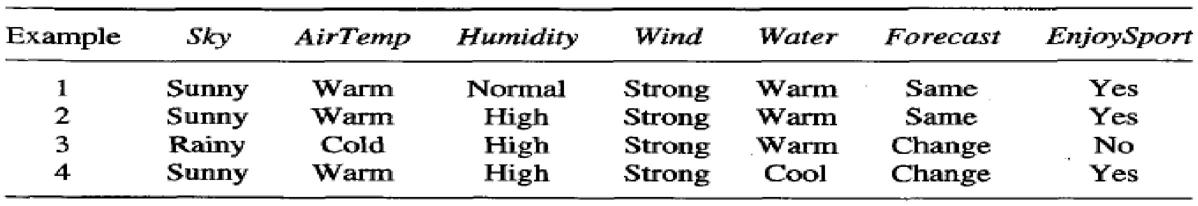
Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular,let each hypothesis be a vector of six constraints, specifying the values of the six attributes ***Sky, AirTemp, Humidity, Wind, Water,*** and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

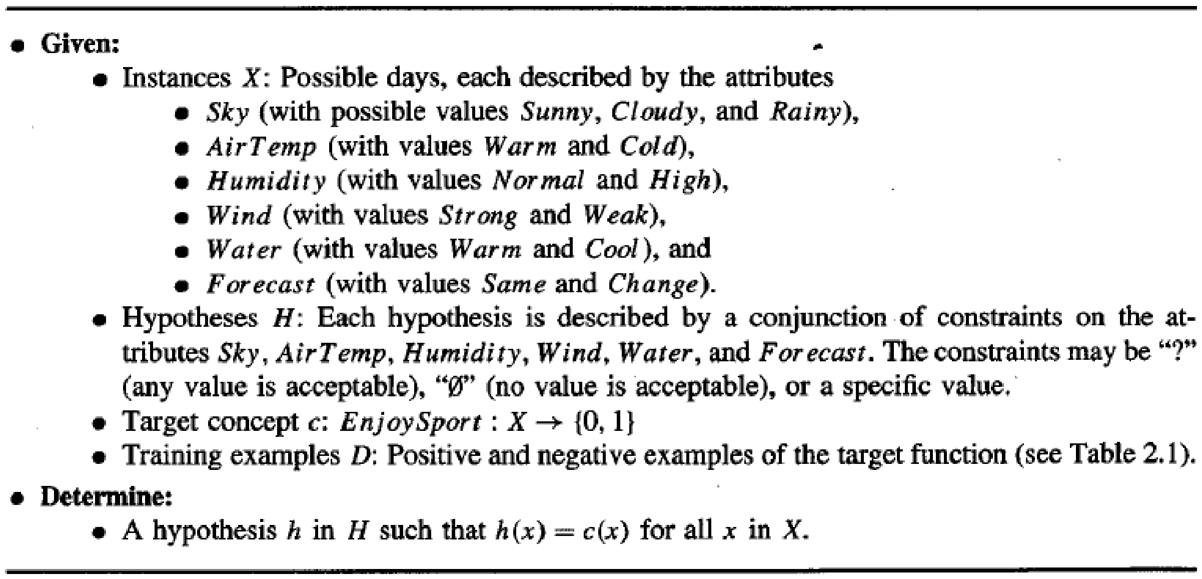
To summarize, the EnjoySport concept learning task requires learning the set of days for which EnjoySport = yes, describing this set by a conjunction of constraints over the instance attributes.

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

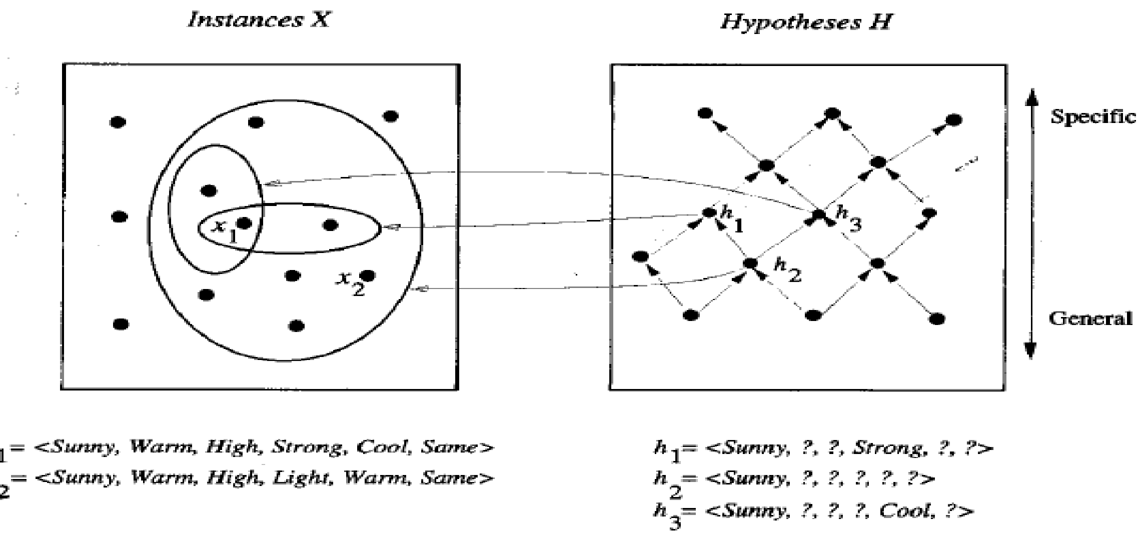
$$h_1 = \langle Sunny, ?, ?, Strong, ?, ? \rangle$$
$$h_2 = \langle Sunny, ?, ?, ?, ?, ? \rangle$$

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**

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



Instances X
Hypotheses H
x_1
x_2
h_1
h_2
h_3
Specific
General
x_1 = <Sunny, Warm, High, Strong, Cool, Same>
x_2 = <Sunny, Warm, High, Light, Warm, Same>
h_1 = <Sunny, ?, ?, Strong, ?, ?>
h_2 = <Sunny, ?, ?, ?, ?, ?>
h_3 = <Sunny, ?, ?, ?, Cool, ?>

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.

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

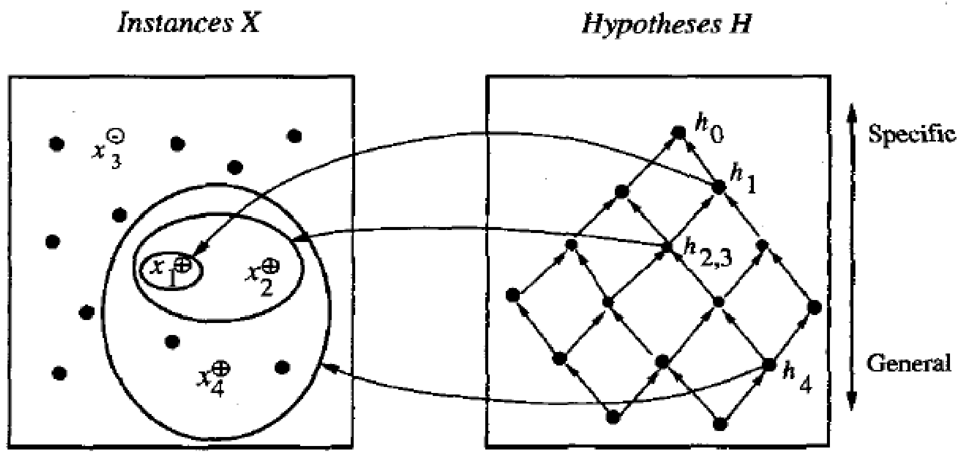
Next, the second training example (also positive in this case) forces the algorithm to further generalize *h,* this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example. The refined hypothesis in this case is

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

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.



	h_0 = <∅, ∅, ∅, ∅, ∅, ∅>
x_1 = <*Sunny Warm Normal Strong Warm Same*>, +	h_1 = <*Sunny Warm Normal Strong Warm Same*>
x_2 = <*Sunny Warm High Strong Warm Same*>, +	h_2 = <*Sunny Warm ? Strong Warm Same*>
x_3 = <*Rainy Cold High Strong Warm Change*>, -	h_3 = <*Sunny Warmt ? Strong Warm Same*>
x_4 = <*Sunny Warm High Strong Cool Change*>, +	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.**

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

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.

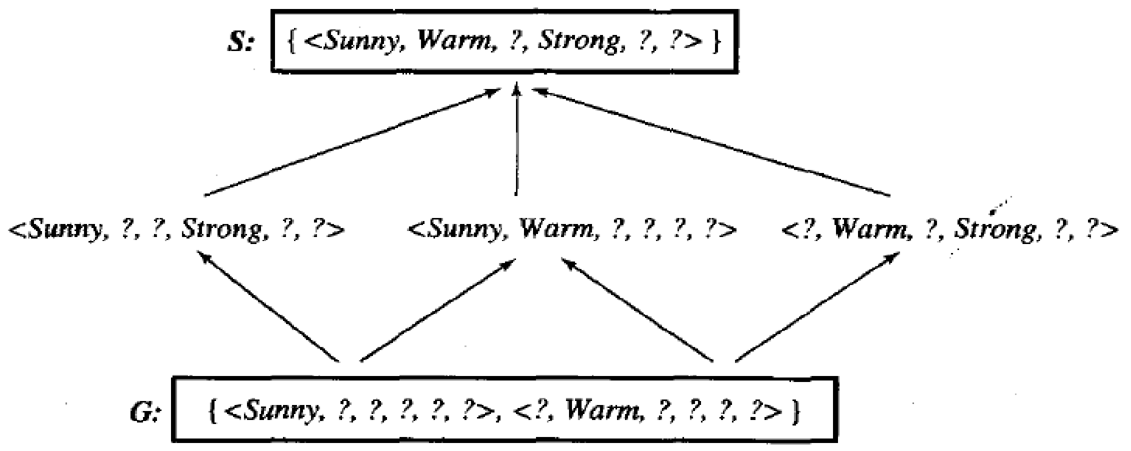


Figure 4: A version space with its general and specific boundary sets. The version space includes all six hypotheses shown here, but can be represented more simply by S and *G.* Arrows indicate instances of the *more_general_than* relation.

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)

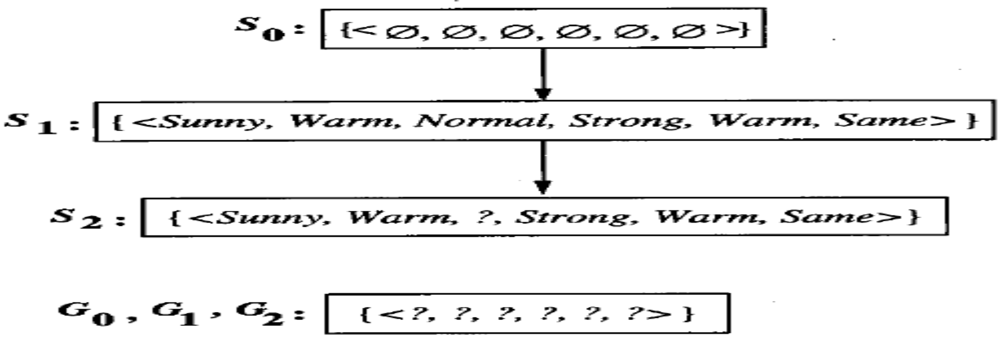

S_0 : {<∅, ∅, ∅, ∅, ∅, ∅>}
S_1 : {<Sunny, Warm, Normal, Strong, Warm, Same>}
S_2 : {<Sunny, Warm, ?, Strong, Warm, Same>}
G_0, G_1, G_2 : {<?, ?, ?, ?, ?, ?>}

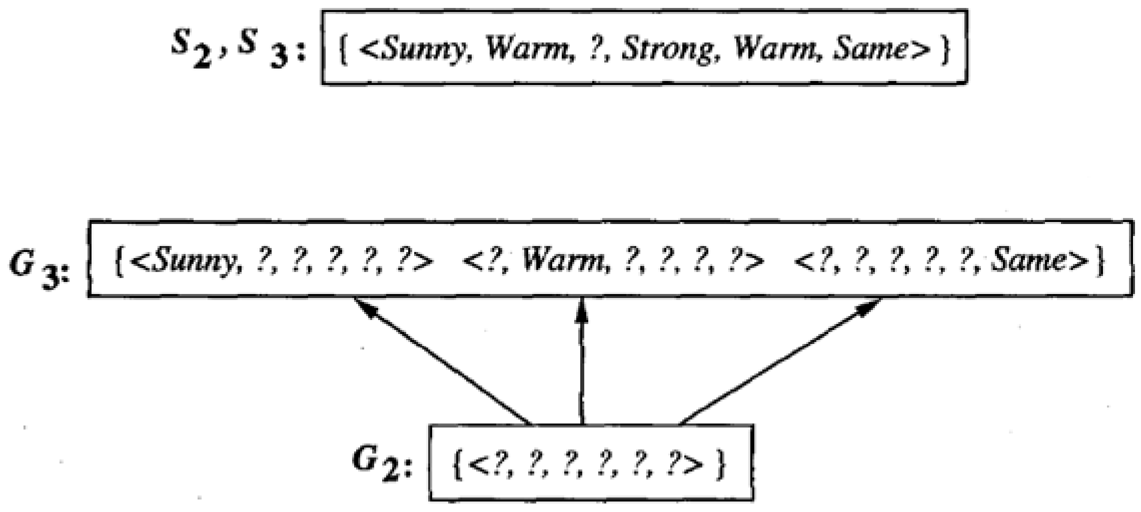
Figure 5: CANDIDATE-ELIMINATION Trace 1. *So* and *Go* are the initial boundary sets corresponding to the most specific and most general hypotheses. Training examples 1 and 2 force the *S* boundary to become more general, as in the FIND-S algorithm. They have no effect on the *G* boundary.

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.**

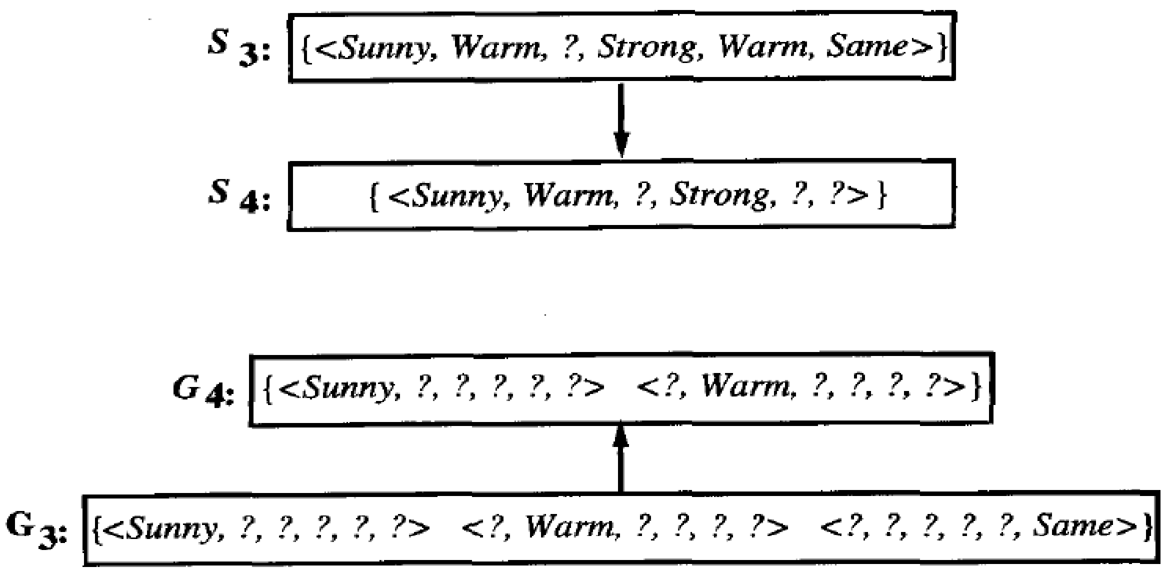


Training Example:

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).

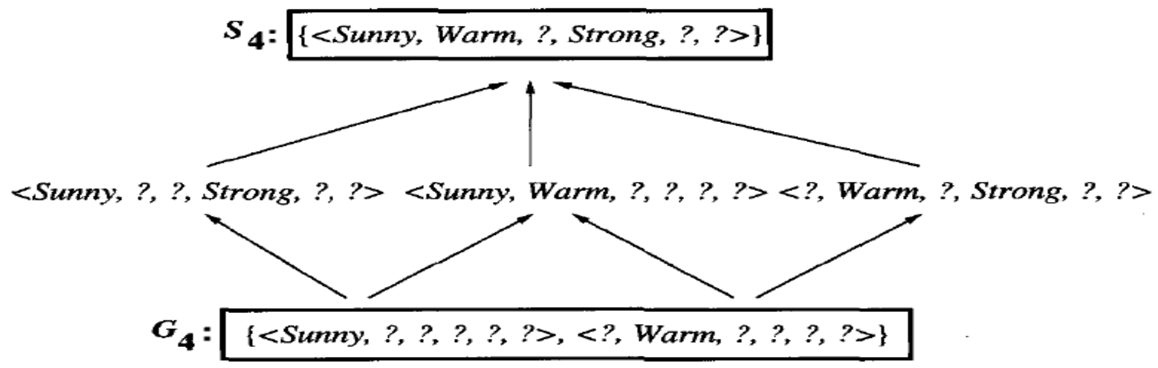


FIGURE 8: The final version space for the *EnjoySport* concept learning problem and training examples described earlier.

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