KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Knowledge-Oriented Machine Learning

A Brief Introduction

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).
The present report, PM-krf-028, can persistently be accessed as follows:

Project Memo URL:	`www.ida.liu.se/ext/caisor/pm-archive/krf/028/`
AIP (Article Index Page):	`http://aip.name/se/Sandewall.Erik.-/2011/008/`
Date of manuscript:	2011-09-09

Related information can also be obtained through the following www sites:

KRFwebsite:	`http://www.ida.liu.se/ext/krf/`
AIP naming scheme:	`http://aip.name/info/`
The author:	`http://www.ida.liu.se/~erisa/`

Introduction

The word "learning" in English covers a broad range of activities, and the same holds for the corresponding word in many other languages. It refers both the the acquisition of knowledge by reading a book or listening to a lecture, and to the modification of behavior by "learning from experience." The thing that is learnt may be a collection of facts, a motoric skill, or a method for how to perform a task. These may be very different things: learning the names of the capitals of the countries in Europe does not have much in common with learning to ride a bike. However, learning to perform a task may include some of each: in order to learn how to do simple plumbing jobs you may both have to learn to know the required components and tools, and the motoric skills of operating them.

It is not surprising, therefore, that the field of Machine Learning also covers a variety of methods that are based on a platform that includes both probability theory, formal logic, programming languages and algorithms. Consequently Machine Learning is an inherently multidisciplinary field.

The present lecture note is intended for use in a course on Artificial Intelligence (A.I.) that emphasizes the role of Knowledge Representation in its subject. It will therefore focus on those aspects of Machine Learning that are relevant for Knowledge Representation and for the learning of knowledge, at the expense of the learning of motoric skills, for example. This is the reason why topics such as Artificial Neural Networks and Genetic Programming are not covered.

Previously covered lecture notes in this course are considered as prerequisites for the present one. Please refer to the course webpage for details:

`http://www.ida.liu.se/ext/kraic/`

Contemporary research in Machine Learning is often strongly based on probability theory. Since the A.I. course that uses this lecture note makes only light requirements for previous knowledge of probability theory, I have also chosen to exclude probability-heavy topics such as estimation of the accuracy of hypotheses during learning. To learn more about the topics that have been excluded here, the reader is recommended to use Tom Mitchell's textbook 'Machine Learning' which is the standard text in the field.

One effect of this approach is that it introduces a separation into the area of Machine Learning, and arguably an artificial one. However, the positive side is that this makes it possible to give a coherent treatment of knowledge representation. This is otherwise hampered as books in different areas of Artificial Intelligence (including Machine Learning) contain their own introduction to, and their own use of a small part of Knowledge Representation. The present series of lecture notes uses Knowledge Representation as the major theme and restricts participating topics to those aspects that have a meaningful connection to K.R.

1 Task-Oriented Learning

1.1 An Introductory Example

Suppose we want a software individual to learn the English names of the major colors. Concretely speaking, we want it to obtain a function that receives a sequence of three integer numbers between 0 and 255 which represent the relative intensity in three components of the light spectrum according to the RGB definition, and that returns a symbol such as `yellow, red` or `violet` indicating what this color is called in English. This would be a *classification function* for color codes.

One way of implementing such a classification function may be for the developer to inspect the three-dimensional space of these color codes and to write the program herself. However, an attractive alternative is to use a program that receives a *training set* consisting of a number of RGB color codes together with the correct color symbol for each of them, and that constructs the classification function automatically.

This is a simple but somewhat typical example of what is usually meant by *machine learning.* We shall return later to a discussion of how well the technical concept of machine learning corresponds to the standard notion of learning in ordinary language, but actually there are many practical applications for machine learning of a kind that more or less resembles this simple example. We shall therefore use it as the point of departure for our presentation.

An obvious requirement on a learning program is that it must be able to *generalize* from the given training set. If it were to generate a classification function that answers "don't know" to all RGB codes except those that were actually in the training set, then clearly it would not be of much use.

Because of the need for generalization, every learning program has what is called a *bias,* that is, a built-in orientation towards a particular way of structuring the resulting classification function (or whatever is being learnt). For example, one learning program may be designed so that it associates each color symbol with a three-dimensional rectangle in the space of RGB codes, that is, an interval for each of the three RGB components, and requires these rectangles to be as large as possible while still being disjoint. Another learning program may associate each color symbol with the union of a set of spheres, namely one around each member of the training set. The choice of bias will have an effect on the quality of the produced classification function, such as the percentage of incorrect classifications and the percentage of "don't know" outcomes.

Generalization may take place at the time of training or at the time when the learned information is used. The former case is the most important one in current Machine Learning methods. In the latter case, which is called *instance-based learning,* the learning program is fairly trivial since it merely accumulates the items in the training set to its knowledge base, with some appropriate indexing scheme for facilitating the computation in the classification function, and then it is up to the latter to use these data in each case that it encounters. In the case of the color learning example, one instance-based learning scheme might simply identify which of the RGB codes in the training set has the smallest Euclidean distance to its input,

and return the color symbol that is associated with it.

1.2 General Technical Definition of Task-oriented Machine Learning

The generally accepted definition of Machine Learning is due to Tom Mitchell and goes as follows when adapted to the present context:

Definition (1): A software individual 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 the experience E.

In the color classification example, E is the training set, T is the classification function, and the measure P may be defined in terms of the proportions of incorrect classifications and "don't know" classifications. Other applications may involve other kinds of tasks besides classification. They may also include other kinds of criteria in the performance measure, such as the computation time that is required for obtaining the result or performing the task.

The formal definition may even apply to the case where the learning process causes changes in some parts of the actual program code for performing the task, and then recompiles it if a compilation-based programming language is used, but this is unusual. The normal case is that the learning activity modifies some of the data in the individual, and that these data are interpreted by a fixed program for performing the task. In the color learning example, the data may be for example the lower and upper bounds of the intervals that define RGB space rectangles for each of the color symbols.

As always in the case of software individuals, changes in the data contents may take place during a session, but the modified data must also be saved e.g. in entityfiles so that they can continue to be used during later sessions with the same individual. It is in this way that information learnt during one session for the individual can continue to be used during later sessions.

In fact, the definition that was given by Tom Mitchell begins "A computer program is said to learn..." However, in order for the definition to make sense with this wording, one must take the word "program" to mean not only the program code that is written in a conventional programming language; the "program" must also be taken to include those *data* that are used together with the program code and that contain the learned information. Since a software individual is an aggregate consisting of both program code and data, and since it is organized for persistence, its organization is appropriate for performing learning.

The use of the word *task* in this context should also be clarified. For the purpose of Machine Learning, it should be understood as *a computational process that the software individual or "program" in question is designed to perform repeatedly, with new input data each time it is invoked.* This notion of task is quite different from when the term is used for *Hierarchical Task Networks* (HTN) as described in another one of the current set of lecture notes. A task in an HTN is an intended and (usually) scheduled action, i.e. something that the individual in question has on its agenda for things to

be done. If there is a possibility of misunderstanding one may use the term *repeatable task* for a task in the sense of Machine Learning.

In our color classification example, the learned data are dedicated to be used by one particular task, namely, the classification function, and they are not likely to have any other use. However, it is easy to think of cases where the learned information may be added to the general knowledgebase of the software individual at hand, and where it may be used by several different tasks in an open-ended manner. For example, the knowledge about the color and other physical characteristics of various fruits may be used in many different ways by an intelligent agent.

Therefore, any kind of information structure that can be used in the knowledge base of an individual should in principle be a candidate for being learned. This includes assignment of attribute values, ground literals expressed using a suitable predicate, and so forth. Notice however that only static information is suitable to be learnt, so literals using the `H` predicate for expressing a transient fact, or using the `D` predicate for expressing the occurrence of an action are not likely to be in the information structure that is generated by a task learning process.

The role of the performance measure P in the learning process is particularly interesting. If the learning activity only applies to one single task then it may be relatively easy to define the performance measure, but if the acquired knowledge is relevant for several different tasks then this adds to the difficulty of defining it. Many kinds of human learning refers of course to knowledge whose actual use is difficult to foresee. In such cases one may wish to define a special *testing task* that can provide a useable performance measure, much like how exams are used in schools.

1.3 Learning for the Characterization Problem

If the result of learning is a dataset, rather than modification and recompilation of program code, it is better to use a definition of task learning that makes this explicit. This definition can be developed in a few steps as follows.

Definition (2): A *characterization problem* consists of three domains D^*, C^* and R^*, one function F from D^* to R^*, and another function T from $C^* \times D^*$ to R^*. A member C in C^* such that $T(C, D) = F(D)$ for all members D of D^* is called an *exact characterization* in this characterization problem. Also, a member C in C^* such that $T(C, D)$ approximates $F(D)$ sufficiently well is called an *approximative characterization* in this problem. A *learning activity* is a computational process that calculates or improves a characterization of F for T using a subset E of F called the *training set.* Members of the training set will be called *examples* (of the function F). Members of C are called *control datasets.*

The function T is of course the task, in the sense of the previous definition, the function F (for *Facit*) is the function specifying the desired and correct outcome of the task for each input dataset, D^* is the set of all possible inputs to the task and R^* is a set that must include all possible outputs.

If the facit function F is completely known then this may be seen as a kind of approximation problem. What is characteristic about learning is however that the function F is only known partially and through examples. The

operational requirement on the learning process is therefore that $T(C, D) = E(D) = F(D)$ for all or most D in the training set, and that there shall be good reasons to assume that $T(C, D) = F(D)$ elsewhere as well.

The indeterminacy in this problem can be handled in a variety of ways. *Inductive learning* makes use of a a combination of biases, as was discussed above, and the following assumption:

Inductive learning hypothesis: The facit function F is "well behaved" in the sense that it is possible to extrapolate relatively well from the members of the training set to values for other members of D^*.

In the color learning example, one may expect that each color word (blue, brown, etc.) corresponds to one, or possibly a few regions in the three-dimensional space of the RGB codes, which would then qualify as "well behaved."

The bias in the selection of the control dataset can be realized in a variety of ways. With *restriction bias* one uses a domain C^* for the control dataset that is designed in such a way that it can not generate all possible mappings from D^* to R^*, but only some of them, which means that often it will not be possible to find any control dataset C whereby $T(C, D) = E(D)$ for all D in the training set, but only for a majority of them. With *preference bias* one admits that there may be several control datasets C that are consistent with the training set, and one uses a preference relation over C^* in order to choose between them. Different choices of domain for the control dataset and of preference relation will result in different bias.

As an alternative or complement to inductive learning, *analytical learning* makes use of a collection of *domain knowledge* for the kind of information that the learning activity refers to. The domain knowledge consists of logic formulas or other similar structures that can be used to for imposing additional constraints on the control datasets that satisfy the members of the training set at hand.

Although our definition of the characterization problem assumes that the task is defined as a function of two arguments where the first one is the control dataset, it is worth noting that this also suggests as systematic method for converting the learnt control dataset to a dedicated program for performing the task, namely by using *partial evaluation.*

1.4 Use of the Performance Measure during Learning

Most uses of Machine Learning today assume that the performance measure is computationally available for the learning activity. The definition in the previous section should therefore be extended so that the role of the performance measure is made precise. At the same time the definition of the characterization problem will be slightly generalized, as follows.

Definition (3): A *performance-measured characterization problem* consists of four domains D^*, C^*, R^* and B^*, one function F from D^* to R^*, another function T from $C^* \times D^*$ to $R^* \times B^*$, and a performance measure that is specified below. For such a problem, a member C in C^* such that the first element of $T(C, D)$ equals $F(D)$ for all members D of D^* is called an *exact characterization* of F for T. The definition of approximative characterization and of learning activity are modified accordingly.

The *performance quality* of a member C of C^* and a member $\langle D, R\rangle$ of F, where $T(C, D) = \langle R', B\rangle$, is a measure that favors cases where R is equal to R' (or nearly equal to R', if R^* has a distance metric defined over it) and that also has preferences with respect to the B component of the outcome of the task.

A *performance measure* is a function that takes a subset E of F and a member C of C^* and produces a numerical value based on the performance quality for all members of E. (End of definition).

The B component is used for estimating the *behavior* of the learning activity as it performs the task, for example, of the computation time that is required for computing $T(C, D)$ for particular choices of D. If the behavior aspect is not of interest then the B component can be chosen as the null element, and the performance measure will simply be a measure of how well the first component of $T(C, D)$ agrees with, or approximates $E(D)$.

One may also consider a more general definition of the performance measure that takes aspects of the learning activity into account, for example, the computation time that was required for obtaining the characterization. However, such a definition is somewhat at odds with the notion of using the performance measure within the learning process itself, which is the topic of the present section.

Definition. An *incremental learning activity* is one that starts with one choice of control dataset C and that produces a modified value C' that is expected to be an improvement with respect to the performance measure at hand.

An incremental learning activity may well be applied repeatedly, so that the software individual that performs the activity gradually improves its performance as new training sets become available. In particular, one way of organizing the learning activity is by way of *iterative learning* where it proceeds stepwise through the training set, addressing one example $\langle D, R\rangle$ at a time, and modifying the control dataset just in order to take this example into account.

A control dataset that is currently being considered during a learning activity is often called a *hypothesis* in order to emphasize that it is tentative and that it is based on the knowledge at hand at a particular point in time during that learning activity.

1.5 Learning without Computational Performance Measure

Although most uses of Machine Learning today are based on using the performance measure computationally during the learning activity, in particular for the selection between alternative, proposed values of the control dataset, there are also examples of learning techniques where the performance measure is only used for the posterior evaluation of a learning activity, and not within that activity. If the performance measure can not be used to guide the learning process, then one alternative is to use domain knowledge in the sense of analytical learning.

Case-based learning can be characterized as instance-based analytical learning.

2 State-based Learning

2.1 Definitions

In this section we consider task learning methods where the task is to classify entities according to the values of their attributes. Definition (3) above is specialized as follows, in KRF terms.

Definition: A task is *state-based using the attribute set* A if A is a set of entities whose type is `attribute` and the domain D^* for the task is formed as the set of all states over A.

Please recall that a state over A is a mapping that assigns, to each member a of A, one member of the range of a. The following is an example of a state characterizing the current 'state' of a person or an animal:

```
{hungry = very, angry = yes, tired = no}
```

In our initial color-learning example, each RGB code is a state consisting of assignments of integer values to three attributes.

A characterization problem for a state-based task is called a state-based characterization problem, and similarly for a state-based learning activity.

Definition (4): A *classification problem* is a state-based characterization problem where the behavior domain B^* consists of only the null element and where the performance measure is the proportion of the items $\langle D, R\rangle$ in E where the first element of $T(C, D)$ is equal to R.

A common use of the classification task is to assign a member of a domain R^* to each member of a set entities, for example in a knowledgebase, based on the values of some of the attributes of those entities. This can be phrased as a classification task by representing each entity e as a state that maps some or all of the entity's attributes to the corresponding attribute value for e. The RGB-color learning task is a classification task where R^* is a set of symbols each representing a particular color name.

The following is a special case of the previous definition.

Definition: An *identification problem* is a classification problem where the domain R^* consists of only the entities `T` and `F` which may be thought of as *true* and *false.*

The members of R^* will sometimes be chosen as `yes` and `no`, or as 1 and 0. For a given function F, the set of those elements in D^* where $F(D)$ is true, is called the *target set* for the identification problem.

Learning in an identification problem has sometimes been called *concept learning* in the Machine Learning literature [Mitchell, 1997] , the idea being that the target set is a manifestation of a "concept" in some sense. However the term concept learning is given a more general meaning in cognitive psychology and in more recent Machine Learning literature, and we shall not use it here.

The RGB-color learning problem can be represented as a *set of* identification problems, namely, one identification problem for each one of the color names being learnt. Notice that a classification problem does not allow one and the same member of D^* to map to more than one member of R^*. For

example, in the case of the color classification problem, it will not allow the same RGB code to be assigned two different color names, such as both `red` and `pink`. If one considers one identification problem for each color name, on the other hand, then it *is* possible to let a particular RGB code be characterized by several color names, namely, by allowing this RGB code to map to `T` in several of the identification functions.

In general, any classification problem can be re-expressed as a set of identification problems, namely one for each member of R^*. A member $\langle D, R\rangle$ of the original training set will then be converted to a member $\langle D, \mathtt{T}\rangle$ in the training set for the identification problem for R, and to a member $\langle D, \mathtt{F}\rangle$ in the training sets for the identification problems for all other members of R^*.

The present section will describe some major methods for learning identification and classification problems. These methods differ with respect to what structure they use for the control dataset C.

2.2 Partial State Methods

Some methods for learning identification problems represent the control dataset using *partial states,* including the well-known *Candidate Elimination Method.* A partial state over a set A of attributes is a mapping that maps some, but not necessarily all members of A to a member of the declared domain for the attribute in question. States that map all members of A to a value are called *total.* Two partial states are *inconsistent* iff they map the same attribute to different values, and *consistent* otherwise. In particular, a total state is consistent with a partial state iff the latter is a subset of (or equal to) the former.

A partial-state representation would not be very useful in the RGB-color example since omitting one or two of the RGB components loses too much information. Moreover, it seems likely that a classification of RGB codes will involve the use of intervals for the integer values in each of the three RGB components, whereas partial states can only represent the choice or absence of a value. Therefore, they tend to be useful in cases involving a fair number of attributes, each one with a small and non-metric domain.

When a partial state is used as the control dataset, then the task function T is defined so that $T(C, D)$ is true if and only if the total state D is consistent with the partial state C, that is, C is a subset of or equal to D. This implies a restriction on the facit function F: for some choices of that function there exists a partial state C such that $T(C, D) = F(D)$ for all D, but for many other choices of F there does not exist any such C.

Since the facit function F is truth-valued, the training set consists of one set of *positive examples* containing states D that belong to the target set, and one set of *negative examples* that don't. [1]

The case where the target set consists of only one or a few members is not of practical interest, and one is usually only interested in cases where the target set has nontrivial size. The learning problem is then the problem

[1] To be precise, since an example is a member of the target function, it is a maplet i.e. a pair consisting of a state and a corresponding value `T` or `F`. Positive examples are those maplets where the second element is `T` and conversely.

of finding which partial state is the correct one, using the assumption that such a partial state does exist.

2.2.1 Using the Lattice of Partial States

Suppose we have an attribute set A consisting of eight attributes selected as `a` to `h` in the alphabet, and s is a partial state assigning values to the attributes from `d` to `h` but omitting values for `a, b` and `c.` Informally, this is a way of characterizing the set of all the total states that have the specified values for `d` through `h` , but regardless of their values for the first three attributes. Formally speaking, s *subsumes* those total states that are consistent with it.

Consider now an identification problem where all members of the training set are consistent with s. Will it then be appropriate to return s as the result of learning ? Not necessarily, since if all members of the training set also have the same value for `a,` besides having the same value for `d` through `h,` then it would also be possible to return a partial state that only leaves `b` and `c` undefined.

In particular, if one makes the a priori assumption that the target set is relatively small compared to the entire domain D^* then it will be natural to prefer a characterization that is as specific as possible, so that it subsumes as few states as possible besides those in the positive examples. This is an example of preference bias for the learning activity.

For the moment let us adopt the bias in favor of control datasets that are as restrictive as possible. Now if all total states that are subsumed by s are present in the training set then the matter is simple, and the partial state s should be returned as the result of learning. However, what about the case where 7 out of those 8 total states are present in the training set, then what should the learning process do?

In practice one will of course work with larger numbers, and sometimes much larger ones, either by having a larger set of attributes, or by having larger domains for the attributes, but we keep the numbers small for the purpose of exposition, and the principles are the same in any case.

There are two possibilities if a proposed partial state is not entirely covered by the training set: to *generalize,* or to *decompose* the partial state being used. If one would say, in the example, that 7 out of 8 is good enough for assuming that all 8 total states shall belong to the identified set, then one has generalized.

If multiple partial states are used, then the control dataset consists of a set of partial states, rather than a single one. Such a control dataset represents the set of all total states each of which is consistent with *at least one* of the partial states in the control dataset.

For example, if one wishes to represent the set of all total states for attributes `a, b` and `c` that have arbitrary values for these attributes, except the one that assigns `t` to all three attributes, then one could use the following partial states:

```
{a = f}
{b = f}
```

```
{c = f}
```

As we proceed now to specific learning methods for the identification task, we need to make a distinction between *one-sided* and *two-sided* methods. A one-sided method is applicable in cases where the training set only contains positive examples. In other words, it gives examples that are members of the target set, but no examples of non-members. A two-sided method is applicable if the training set contains both positive and negative examples.

2.2.2 One-Sided Methods

The `Find-S Method` is a method for the one-sided identification learning task that works as follows. (Please refer to the description of this method in Tom Mitchell's book for further information).

2.2.3 Two-Sided Methods

We proceed now to two-sided methods which offer a greater variety of possibilities. The *Candidate Elimination method* is due to Tom Mitchell (1977); it is an incremental method for the two-sided identification problem that uses generalization but not decomposition. The domain C^* of control datasets is therefore chosen as the set of partial states, which means that exact characterizations can only be obtained for those task functions that can be defined as a conjunction of literals each of which specifies a required value for one of the attributes.

Therefore, for example, a facit function that is true if at least one of the attributes `a` and `b` is true, but false if both are false, can not obtain an exact characterization. Also, if an attribute `c` can have four different values 1, 2, 3 or 4, and a facit function is true if the value is 1 or 2 but false otherwise, then again it can not obtain an exact characterization when the Candidate Elimination method is used.

We shall describe the Candidate Elimination method, or C-E-method, for the special case where all attributes in A are truth-valued. It is easy to generalize it to the case of attributes having a finite set of discrete values in their domains.

The algorithm for the C-E-method maintains a working structure consisting of two sets of partial states, referred to as the *general boundary* and the *specific boundary,* respectively. These two boundaries characterize a set of partial states that we shall call the *current hypotheses* consisting of the members of the two boundaries plus some additional partial states that are "between" the boundaries. [2]

The idea for these boundaries is as follows. Suppose the incremental learning activity according to the C-E-method has processed some members of the training set, and the boundaries have been constructed accordingly. Then you select again any of the already used examples in the training set and

[2] Please recall that a currently considered control dataset in a learning activity is often called a hypothesis, since it describes the currently available experience at that time. The set of current hypotheses is usually called the *version space* in the literature on Machine Learning, but we shall avoid that term as being too idiosyncratic.

relate it to the current hypotheses. If it is a positive example then it will be subsumed by all the current hypotheses; if it is a negative example then it will not be subsumed by any of the current hypotheses. In this sense all the current hypotheses are consistent with (i.e. return the same value as) all the training examples so far.

Let H be the set of current hypotheses and let $U(H, D)$ be a function that is defined as follows: If $T(C, D)$ is the same for all members C in H then it returns that value (true or false), but if different members of H obtain different values for T then it returns a third value representing "don't know." Clearly U will return the correct value for all examples in the training dataset that have been used up to the current point. Moreover, there is a built-in generalization due to the restriction bias. For example, if the training dataset contains one example where `a` is true and `b` is false, and another example where `a` is false and `b` is true, then *every* current hypothesis must be a partial state that leaves both `a` and `b` undefined. This is since it is not possible to have a control dataset that characterizes disjunction.

For these reasons, the set of current hypotheses will correctly characterize the currently used examples in the training set, and also some other members of D^*, but (in general) not all of them, so there will be some members of D^* where the function U will return the value "don't know." However, as more and more examples are added from the training set, the C-E-method will move the specific boundary and the general boundary closer to each other, and if sufficiently many examples are added then these boundaries will converge so that the set of current hypotheses will consist of one single member, namely, the one that exactly characterizes the facit function.

How can the result of learning according to this method be put to use if the training set has not been sufficient for obtaining convergence? One possibility is to use the function U together with the set of current hypotheses that one has obtained with the training set, which means that one has to live with a solution that sometimes returns "don't know." Another possibility is that if the specific boundary consists of only one partial state, then one may adopt it, which means effectively that all "don't know" answers are considered as "no."

2.2.4 The Algorithm for the Candidate Elimination Method

Let us now proceed to the algorithm that is used by the C-E-method and to some examples. The algorithm is expressed in terms of the lattice formed by all the partial states, with the empty (most general) partial state at the bottom, and with more specific partial states higher up. Moreover, in the description of the algorithm one adds an artificial *top element* to this lattice, in such a way that it is above all the total states. Yet another possibility which has been proposed is to make a vote between the current hypotheses and use the opinion of the majority.

The algorithm for the C-E-method is defined as follows. It uses the symbol `G` for the general boundary and `S` for the specific boundary.

```
Initialize G to the set of the empty partial state.
Initialize S to the set of the artificial top element.
For each example d in the training set:
If d is a positive example:
   Remove from G any hypothesis inconsistent with d
   For each hypothesis s in S that is inconsistent 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 below another one in S
If d is a negative example:
   Remove from S any hypothesis inconsistent with d
   For each hypothesis g in G that is inconsistent 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 above another one in G
```

With respect to the negative examples, notice that a negative example $\langle D, \mathtt{F} \rangle$ is consistent with a current hypothesis h if D is *inconsistent* with h, and vice versa. The meaning with the hypotheses is that they shall subsume states that are at least possibly in the target set, so the state in a negative example must differ from such a hypothesis in at least one of its components otherwise there is a conflict.

A walk-through example of the use of this algorithm is shown in Tom Mitchell's book, Section 2.5.5 (pages 33-36).

2.3 Perspectives on Learning

Partial state methods is one class of learning methods; other learning methods for the characterization problem use other choices of domain for the control dataset. *Decision Tree Learning* uses decision trees for this purpose. Some aspects that we have seen for partial state methods can be generalized since they are applicable to other classes of learning methods as well. This applies, for example, for the need for a bias in the learning method and the use of the inductive learning hypothesis.

Moreover, the Candidate Elimination method can be understood as a search process that operates on the set of all control datasets, and that performs successive operations for narrowing down the set of those control datasets that satisfy the training set at hand. The view of machine learning as search in the space of control datasets can be used for several other types of learning methods as well, with their respective control dataset domains. In the Machine Learning literature it is common to use the term *hypothesis* for a control dataset, and the term *hypothesis space* for the domain C^* of control datasets. Learning is then seen as a search process in the hypothesis space.

This general use of the term "hypthesis" has the advantage of being applicable even if the task function T is defined in some other way than as a function of a control dataset and a data instance. For example, if the learning activity should produce a revised algorithm for performing the task, as

a function of a single argument, then there is no control dataset but the terminology using the hypothesis concept is anyway applicable, and each possible revised algorithm is a member of the "hypothesis space."

Moreover, in those cases where the learning process produces and modifies a neural net, it is true that the neural net can be seen as the control dataset for a neural-net executive, but in some contexts one may wish to disregard that aspect and view the neural net itself as the "program" that has been learnt from the training set. Then, again, each possible configuration of the neural net would be a hypothesis and a member of the hypothesis space.

In the present textbook we emphasize the role of the knowledge structures in artificial intelligence systems, which is the reason why we prefer to use terms that refer directly to the kinds of data being used, such as partial states, decision trees, and other choices of control datasets. This has the advantage of facilitating comparison with other chapters in the present textbook where similar structures are being used. The use of partial states is fundamental in the section on planning and reasoning about actions, for example.

2.4 Decision Tree Learning

A major method for state-based learning is *decision tree learning,* where of course the control dataset is a decision tree rather than a partial state. The original method for decision tree learning is called ID3 and it will be described here for the particular case of the identification problem, i.e. the R^* domain consists of only true and false.

Whereas the C-E-method is an incremental method, ID3 is not. The basic idea in ID3 is to consider the entire training set and to construct a decision tree from it with a preference for decision trees that perform the classification task using as few tests as possible on the average. It therefore starts with the empty decision tree, selects one of the attributes for use in the top-level node of the decision tree, partitions the training set into one part for each of the possible values of the selected attribute, and then repeats the process recursively in each of the branches.

This is straightforward enough, but the interesting question is how shall one select the "best" attribute in each step of this process, given the overall goal of the learning algorithm. Intuitively it is plausible that the tree shall be as "balanced" as possible. For example, if all the attributes in the set A are truth-valued so that the decision tree is a binary tree, then it would make sense to select the attribute for which the two parts of the example set at hand are the most equal in size.

This intuition is fine for binary decision trees but it does not help much if some attributes are multi-valued. The ID3 method uses *entropy* as the measure for choosing between the remaining attributes at each step in its algorithm. At this point we need some notation and formal definitions.

Let E' be the set of examples (from the training set) that are being considered at a particular point in the construction of the decision tree, and let A' be the remaining set of attributes at this point, i.e. containing those members of A that have not yet been tested for on the path to the present point in the tree. We call E' the current *training subset.* For any attribute a in A and any permitted value v of this attribute, let $select(E', a, v)$ be

the subset of E' consisting of those members of E' where the value for the a attribute is v. Moreover, the proportion function *prp* is defined so that

$$prp(E', a, v) =\mid select(E', a, v) \mid / \mid E' \mid$$

These functions are extended by considering the value of the classification task at hand as a virtual attribute, called r, so that $select(E', r, \mathtt{T})$ is the set of positive examples in E'.

Then the *entropy* for the choice of a particular attribute a (including r) is defined as

$$entropy(E', a) = -\sum_v prp(E', a, v)\ log_2\ prp(E', a, v)$$

If a is a two-valued attribute then this sum over two elements is maximized if the two partitions are equal in size.

The *information gain* for a particular choice of attribute a in A' is then defined as follows.

$$gain(E', a) = entropy(E', r) - \sum_v \frac{|prp(E', a, v)|}{|E'|}\ entropy(prp(E', a, v), r)$$

Here, $entropy(E', r)$ is the entropy of E' when it is only partitioned according to r, i.e. as positive and negative examples, whereas the second term on the right-hand side is the entropy of E' as partitioned according to both r and the value of the attribute a. Zero entropy means that tha given identification task has been completed, high entropy means that much remains. Zero gain means no progress on the task, high gain means much progress.

In order to understand this informally, notice that $-p\ log_2 p$ is zero if p is 0 and when it is 1, and positive in-between. If all members of E' are positive, or if all are negative, then $entropy(E', r)$ is zero, the classification has already been done, and no improvement is possible. On the other hand, if E' has equally many positive and negative examples, but for each value v for the attribute a the partition is either all positive or all negative, so that the attribute a obtains a value `T` or `F` without any doubt, then $entropy(prp(E', a, v), r)$ obtains the value 0 for every v, which means that the gain is large. Finally, to understand the weighting component, suppose you have 10 possible values for a, one of them has all positive examples, another one has all negative examples, the other 8 have an even mix, but almost all members of E' belong to one of the first two partitions. In this case the attribute a is a good one since it obtains a classification in most cases. This illustrates the reason why the sum in the definition of *gain* is weighted by the relative size of the respective partition.

The ID3 method is an example of a depth-first hill-climbing algorithm: it operates in a space of possible control datasets (namely, the possible decision trees), at each point it uses only one "current" decision tree, it considers the set of "neighboring" decision trees in the sense of those that contain one more node, and it selects the one among these that obtains the best improvement with respect to the entropy criterium. It therefore runs the same risk as is always present for hill-climbing, namely that it may not find the best possible solution. Some of the other well-known search methods can be applied to alleviate this risk, for example using *beam search* that maintains a working set of several alternative decision trees during the search process and expands them in parallel.

At this point: Read section 3.7, "Issues in decision tree learning," in Tom Mitchell's book, pages 66-76.

2.5 Nearest Neighbor Algorithms

Nearest-Neighbor algorithms are methods for *instance-based learning,* i.e. they use a learning process that merely stores away the training examples, and the actual use of them takes place within the task. These algorithms require that a metric has been defined over the state domain, for example, using the Euclidean distance if states are coordinates in an n-dimensional space. In the simplest case the classification task uses the classification of *the* nearest state in the preserved training set as an answer to a given state.

There is a generalization called the k-nearest-neighbor algorithm where one uses the k nearest neighbors in order to obtain an answer. The classifications provided by these are used, and the answer is obtained by voting, possibly also using a weight function or other modifying device.

Further details about this type of learning algorithm is found in Section 8.2 in Tom Mitchell's book, pages 231-236. The subsequent sections, which address generalizations to regression methods and radial basis functions, are recommended for the mathematically inclined.

3 Rule-Based Learning

Please recall the following definition from the chapter (or lecture note) on the Knowledge Representation Framework:

A *one-way rule* is formed like in the following example:

```
[! [grandfather .x .z] if [father .x .y][father .y .z]]
```

In general, a one-way rule is formed as a record where the symbol `!` is the record former, the second "argument" is the symbol `if,` and all other arguments are positive or negative literals. One-way rules are intended to be used in axioms that define predicates or functions: the rule in the example is intended to mean the same as the formula

```
(imp [exists .y person (and [father .x .y][father .y .z])]
     [grandfather .x .z] )
```

The exact and more general definition is given elsewhere. Parameters in the one-way rule can be used for restricting the type of each variable that occurs in it.

A *two-way rule* is similar but uses the symbol `iff` instead of `if` and its meaning uses two-way implication (i.e. equivalence) instead of `imp.`

The use of the existential quantifier in the formula defining meaning of the rule is of course redundant for the one-way rule, but it is needed for the two-way rule.

Definition: A task is *rule-based* if it is defined as a function $T(C, D)$ where C is a set of rules and D is a set of ground literals, and where the value of $T(C, D)$ is a set of ground literals.

The particular task may specify what is desired as the value of $T(C, D)$. In some cases it may be the *ground closure* of D with respect to C, i.e. the set of all ground literals that can be inferred from D using the rules in C; in

other cases it may be just the information whether one particular ground literal or its negation are a members of the ground closure.

Techniques for *rule learning* are defined for allowing a software individual to learn rules such as these from a training set. We distinguish between *propositional rule learning* for rules that only contain one variable, namely a variable representing the given example, and *first-order rule learning* where rules admit additional variables, such as the rule example shown above. In this section we shall address propositional rule learning, then first-order rule learning, and finally Inductive Logic Programming.

Notice that in a state-based classification task there is a distinction between the attributes that occur in the members of D^* and the additional "attribute" that is obtained through the classification, but in the case of rule-based learning there is no such distinction. A particular predicate may occur in the conclusion in some rules and among the conditions in some other rules.

3.1 Learning Propositional Rules

The following is an example of a propositional rule:

```
[! [is-dangerous .x] if [is-snake .x] [has-zigzag-back .x]]
```

This rule can be applied to any entity that satisfies the two predicates `is-snake` and `has-zigzag-back` and allows one to conclude that this entity satisfies the predicate `is-dangerous.`

The major approach for learning propositional rules is by *sequential covering.* In this method, the training dataset is converted to a set of *given literals* each of which consists of a monary predicate, such as those in the example, and an entity that is the argument of that predicate. Notice that both positive and negative literals may be used.

The sequential covering algorithm uses a working datastructure consisting of a set of *current literals* and a set of *current rules.* The *deductive closure* of the current literals and the current rules is the set of literals consisting of all the current ones and all other literals that can be obtained from the given ones using the current rules in one or more steps.

The learning activity is initialized with the representation of the training set in terms of literals as the current literals set, and with the empty set as the current set of rules. Then, in each step of the learning activity, one identifies a new rule that *covers* a sufficient number of current literals, according to the following requirements, where the *extended rule set* is the current set of rules plus the new rule:

- The deductive closure of the current literals set and the extended rule set is consistent.
- There is a subset M of the current literals set such that the deductive closure of M and the extended rule set equals the deductive closure that is obtained with the current literals.
- The difference set i.e. the set of those current literals not in M is sufficiently large.

If such a rule can be found then it is added to the current rule set, the set M is adopted as the current literals set, and the process is repeated until no additional rule qualifies.

The difficulty lies in finding such a rule. Rule learning consists therefore of two nested loops, computationally speaking, where the outer loop adds one rule at a time, and the inner loop performs a search for an appropriate rule with good covering properties.

One possibility with respect to the inner loop is to use a variation of the ID3 algorithm for learning decision trees. Notice that a decision tree can be seen as a set of rules, namely, one rule for each path through the tree from the root node to a terminal node. Each choice node in the tree corresponds to one antecedent in the rule, and the outcome from the path corresponds to the consequent of the rule. If one operates the ID3 algorithm in depth-first fashion, so that it only extends one branch of the tree in the depth direction and never pursues any other branches, then what one is doing is effectively to construct one propositional rule. The entropy-based choice criterium that is used by the ID3 method is equally applicable then.

Clearly this is again a greedy method, with the same well-known problems as we discussed before, and one can modify it for example by performing a beam search, that is, by expanding a few paths rather than one single path in the underlying decision tree.

Additional variations of this method are described in Section 10.2.2 (pages 279-280) in Tom Mitchell's book.

3.2 Learning First-Order Rules: The FOIL Method

The FOIL method (Quinlan, 1990) is a generalization of the sequential covering method for the case of first-order rules. For this method, the given training set is assumed to consist of ground literals which may be both positive and negative. The control dataset is a set of rules, as defined above, where functions are not allowed so the arguments of predicates must be entities or variables. Again, both positive and negative literals may occur in the antecedents of the rules.

Although the training set should contain both positive and negative examples, the FOIL algorithm only looks for rules where the consequent is positive, i.e., rules that characterize the target set directly.

The construction of a rule in the inner loop of FOIL proceeds in a way similar to the propositional case. Suppose the learning task consists of learning the predicate `[dangerous .a]` for a variety of animals `.a`. The rule construction procedure starts with the following rule

```
[! [dangerous .a] if ]
```

which means that it says that all animals are dangerous. It then adds successive literals containing `.a` and possibly also other variables to the list of antecedents. In each step it considers several alternative literals that may be added; for each such literal it considers the set of all possible assignments of values to all the variables in the rule at hand when this literal has been added, checks for which of these assignments the new rule is consistent with the literals in the training set, and calculates the information gain due to

the addition of this literal. The best candidate literal is added to the rule, and the rule construction process continues until a rule with sufficiently good information gain has been found.

For additional details, please see Section 10.5.2 (pages 288-290) in Tom Mitchell's book.

4 Additional Topics

The following additional topics would also have a natural place in the present lecture note but must be omitted due to lack of time.

- Inductive Logic Programming
- Case-based Learning
- Analytical Learning

I hope to be able to provide additional text for them in the near future.