Machine Learning, 17, 5–46 (1994)
© 1994 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Generalizing Version Spaces

HAYM HIRSH hirsh@cs.rutgers.edu
Department of Computer Science, Rutgers University, New Brunswick, NJ 08903

Editor: Dennis Kibler

Abstract. Although a landmark work, version spaces have proven fundamentally limited by being constrained to only consider candidate classifiers that are strictly consistent with data. This work generalizes version spaces to partially overcome this limitation. The main insight underlying this work is to base learning on version-space intersection, rather than the traditional candidate-elimination algorithm. The resulting learning algorithm, *incremental version-space merging (IVSM)*, allows version spaces to contain arbitrary sets of classifiers, however generated, as long as they can be represented by boundary sets. This extends version spaces by increasing the range of information that can be used in learning; in particular, this paper describes how three examples of very different types of information—ambiguous data, inconsistent data, and background domain theories as traditionally used by explanation-based learning—can each be used by the new version-space approach.

Keywords: version spaces, inductive learning, noisy data, ambiguous data, explanation-based learning

1. Introduction

Inductive concept learning can be viewed as a problem of search (Mitchell, 1982), (Simon and Lea, 1974)—to identify some classifier out of a space of possible classifiers expressible in a given language. Mitchell (1982) formalized this view of *generalization as search* in his development of version spaces: A *version space* is the set of all classifiers expressible in the language that correctly classify a given set of data. Mitchell furthermore noted that the relative generality of classifiers imposes a partial order that allows efficient representation of a version space by the boundary sets S and G containing the most specific and most general classifiers in the space. Incremental learning is accomplished with the *candidate-elimination algorithm*, which manipulates only the boundary sets of a version space: as each new example is obtained, S and G are modified so that they represent a new version space containing those classifiers that correctly classify all the previously processed data plus the new example.

Although a landmark work, version spaces have proven fundamentally limited due to their restriction to learning from consistent, noise-free training data. This work generalizes version spaces to partially overcome this assumption. The central idea is to base learning on version-space intersection, rather than the traditional candidate-elimination algorithm. Under this new approach each piece of information (*e.g.*, an instance and its classification) is viewed as specifying some version space of relevant classifiers; learning takes place by intersecting these single-piece-of-information version spaces to get the version space that reflects the union of the individual pieces of information. The result of this change in perspective, from the candidate-elimination algorithm to a version-space-intersection-based approach to learning, generalizes version spaces by removing the assumption that a version space is the set of classifiers strictly consistent with training data, and instead allows version spaces to contain arbitrary sets of classifiers, however generated, as long as they can be represented by boundary sets. This results in a generalization of Mitchell's version-space approach to concept learning that extends its range of applicability by increasing the range of information that can be used by the version-space learning method.

This paper is structured as follows. The paper begins with a brief review of the traditional version-space approach to concept learning in Section 2, followed by an overview of the new, generalized approach in Section 3. Given that this work claims to generalize the original version-space approach, it is important to guarantee first that it maintains the functionality of the original approach; Section 4 demonstrates that the original functionality is maintained by showing how the candidate-elimination algorithm can be emulated using the new version-space approach. This emulation also modestly generalizes the candidate-elimination algorithm, in that it allows the first example to be negative and does not require the "single-representation trick" (Dietterich *et al.*, 1982) (*i.e.*, that for each instance there is a classifier that classifies only it as positive and everything else as negative). Next, to substantiate more fully the claims of greater generality, Sections 5 through 7 describe how three very different types of information—ambiguous data (Section 5), inconsistent data (Section 6), and background domain theories (Section 7)—can be used by the new learning method. Section 8 surveys related work, both other attempts to generalize version spaces, as well as relevant efforts from outside a version-space framework. Section 9 concludes the paper with some final remarks, and is followed by an appendix analyzing the computational complexity of this approach in comparison to that of the candidate-elimination algorithm.

2. Version Spaces and the Candidate-Elimination Algorithm

Given a set of training data for an unknown "target" classifier and a language in which the desired classifier must be expressed, Mitchell defines a version space to be the collection of all classifiers in the language that correctly classify the training examples. Mitchell noted that the relative generality of classifiers imposes a partial order that allows efficient representation of the version space by the boundary sets S and G representing the most specific and most general classifiers in the space. A version space contains all classifiers as or more general than some element in S and as or more specific than some element in G.

Given a new instance, some of the classifiers in the version space for past data may no longer be consistent with the new instance. The *candidate-elimination algorithm* manipulates the boundary-set representation of a version space to create boundary sets that represent a new version space consistent with all the previous instances plus the new one. For a positive example the algorithm generalizes the elements of the S set as little as possible so that they cover the new instance yet remain consistent with past data, and removes those elements of the G set that do not cover the new instance. For a negative instance the algorithm specializes elements of the G set so that they no longer cover the new instance yet remain consistent with past data, and removes from the S set

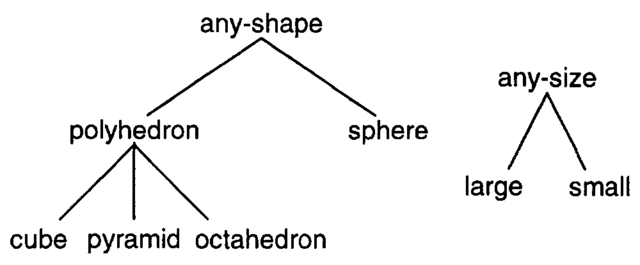

Figure 1. Generalization Hierarchies.

those elements that mistakenly cover the new, negative instance. The unknown classifier is determined when the version space has only one element, which in the boundary set representation is when the S and G sets have the same single element.

To demonstrate the candidate-elimination algorithm, consider a robot manipulating objects on an assembly line. Occasionally it is unable to grasp an object. The learning task is to form rules that will allow the robot to predict when an object is graspable. To make the example simple, the only features of objects that the robot can identify, and hence the only features that may appear in the learned rules, are shape and size. An object may be shaped like a cube, pyramid, octahedron, or sphere, and may have large or small size. Further structure to the shape attribute may be imposed by including in the robot's vocabulary the term "polyhedron" for cubes, pyramids, and octahedra. The generalization hierarchies that result are shown in Figure 1. In this example classifiers take the form

```
Size(X,small) ∧ Shape(X,polyhedron) → Graspable(X)
```

which will be abbreviated to "[small, polyhedron]." The language is assumed to be sufficient for expressing the desired classifier, and the data are assumed to be noiseless. The four examples that will be processed are given in Table 1.

The first object the robot tests is graspable, and is thus a positive example of the target classifier. It is a small cube, and hence the initial version space has boundary sets S={[small, cube]} and G={[any-size, any-shape]}. The second object on the assembly line cannot be grasped, so is a negative instance. It is a small sphere, yielding

Table 1. Training Data.

Example	Class
1. [small, cube]	+
2. [small, sphere]	–
3. [large, octahedron]	–
4. [small, pyramid]	+

Table 2. Learning Example Trace.

Example	Class	S	G
1. [small, cube]	+	{[small, cube]}	{[any-size, any-shape]}
2. [small, sphere]	−	{[small, cube]}	{[any-size, polyhedron]}
3. [large, octahedron]	−	{[small, cube]}	{[any-size, cube], [small, polyhedron]}
4. [small, pyramid]	+	{[small, polyhedron]}	{[small, polyhedron]}

new boundary sets S={[small, cube]} and G={[any-size, polyhedron]}—the only way to specialize the G set to exclude the new instance but still cover the S-set element is to move down the generalization hierarchy for shape from any-shape to polyhedron. A further negative instance, a large octahedron, prunes the version space yet more, to S={[small, cube]} and G={[any-size, cube]; [small, polyhedron]}. The new G set now has two elements since there are two ways to specialize the old G set to exclude the new instance but still cover the S-set element. After a final, positive instance that is a small pyramid, the boundary sets become S=G={[small, polyhedron]}, yielding the final classifier that labels as graspable all small polyhedral objects. The processing done by the learning algorithm is summarized in Table 2. It gives the S and G sets after each instance is processed.

3. Incremental Version-Space Merging

There were two principal insights in Mitchell's work. The first was to consider and keep track of a set of candidate classifiers, rather than keeping a single classifier deemed best thus far. The second insight was that the set of all classifiers need not be explicitly enumerated and maintained, but rather the partial ordering on classifiers could be exploited to provide an efficient means of representation for the space of classifiers. The key idea in the work presented here is to maintain Mitchell's two insights, but remove its assumption of strict consistency with training data—a version space is generalized to be any set of classifiers in a description language representable by boundary sets.

Since the candidate-elimination algorithm assumes strict consistency with data, a different version-space learning algorithm is necessary to handle such a generalized notion of version space. Rather than basing the learning algorithm on shrinking a version space (as the candidate-elimination algorithm is traditionally viewed), the new algorithm is instead based on version-space intersection. Given a version space based on one set of information, and another based on a second set of information, the intersection of the two version spaces reflects the union of the sets of information. Although originally presented by Mitchell (1978) as a possible way to parallelize learning (by dividing the data into smaller subsets, learn from each subset on a separate processor, then intersect the resulting version spaces to get a final result), this work uses version-space intersection as the basis for the new incremental version-space learning method, called *incremental version-space merging (IVSM)*, developed as part of this work.

The algorithm for computing the intersection of two version spaces is called the *version-space merging algorithm*, and it computes this intersection using only boundary-set representations. This is pictured in Figure 2. Given version spaces VS_1 with boundary sets S_1 and G_1, and VS_2 with boundary sets S_2 and G_2, the version-space merging algorithm finds the boundary sets $S_{1\cap 2}$ and $G_{1\cap 2}$ for their intersection, $VS_1 \cap VS_2$ (labeled $VS_{1\cap 2}$). It does so in a two-step process. The first step assigns the set of minimal generalizations of pairs from S_1 and S_2 to $S_{1\cap 2}$, and assigns the set of maximal specializations of pairs from G_1 and G_2 to $G_{1\cap 2}$. The second step removes overly general elements from $S_{1\cap 2}$ and overly specific elements from $G_{1\cap 2}$. In more detail:[1]

1. For each pair of elements s_1 in S_1 and s_2 in S_2 generate their most specific common generalizations. Assign to $S_{1\cap 2}$ the union of all such most specific common generalizations of pairs of elements from the two original S sets. Similarly, generate the set of all most general common specializations of elements of the two G sets G_1 and G_2 for the new G set $G_{1\cap 2}$.

2. Remove from $S_{1\cap 2}$ those elements that are not more specific than some element from G_1 and some element from G_2. Also remove those elements more general than some other element of $S_{1\cap 2}$ (generated from a different pair from S_1 and S_2). Similarly remove from $G_{1\cap 2}$ those elements that are not more general than some element from each of S_1 and S_2, as well as those more specific than any other element of $G_{1\cap 2}$.

The only information a user must give for this version-space merging algorithm to work is information about the description language and the partial order imposed by generality. The user must specify a method for determining the most general common specializations and most specific common generalizations of any two classifiers. The user must also define the test of whether one classifier is more general than another. Given this

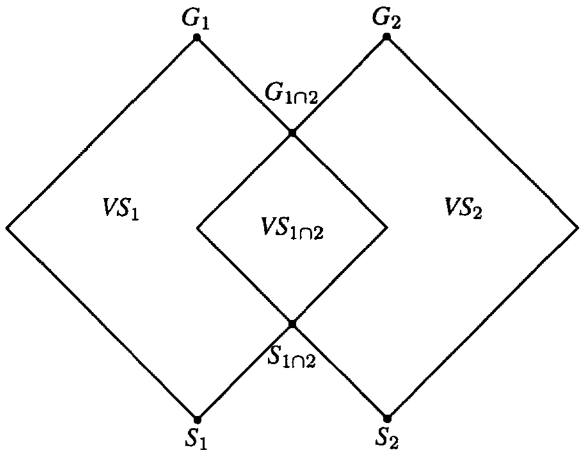

Figure 2. Version-Space Merging.

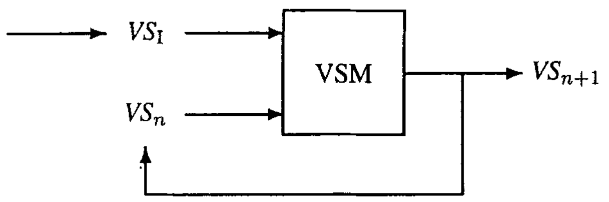

Figure 3. Incremental Version-Space Merging.

information about the description language, the two-step process above will intersect two version spaces, yielding the boundary-set representation of the intersection.

The move from the candidate-elimination algorithm to incremental version-space merging reflects a change of perspective from version-space shrinking to version-space intersecting. The result of this new perspective is that each new piece of information that provides some constraint on the space of viable classifiers should be represented as a version space, and it will be intersected with other version spaces reflecting other constraints on the space of viable classifiers. When new information is obtained, incremental version-space merging forms the version space containing all classifiers that are "relevant" given the new information and intersects it with the version space for past information. This is pictured in Figure 3. As each new piece of information is obtained, its version space is formed (VS_I) and intersected with the version space for past data (VS_n) to yield a new version space (VS_{n+1}), which will itself be intersected with the version space for the next piece of information.

The general algorithm proceeds as follows:

1. Form the version space for the new piece of information.
2. Intersect this version space with the version space generated from past information.
3. Return to the first step for the next piece of information.

The initial version space contains all classifiers in the language, and is bounded by the S set that contains the empty concept that classifies everything as negative, and the G set that contains the universal concept that classifies everything as positive.

The generality of incremental version-space merging arises from the observation that the specific learning task should define how each piece of information is to be interpreted—what the version space of relevant classifiers should be—without having to consider how the information is resolved with other pieces of information. With incremental version-space merging the learning mechanism (repeated version-space intersections) stays fixed, and integrating different sources of information into learning only requires a specification of how the individual version spaces should be formed. Forming the version space of classifiers consistent with each instance (*i.e.*, those classifiers that correctly classify the given instance) results in an emulation of the candidate-elimination algorithm. Forming the version space of classifiers consistent with any of the identities that an incompletely specified example may be provides a way to learn from ambiguous data. Forming the version space containing classifiers consistent with either the instance or some neighboring instance provides a way to learn from some cases of inconsistent data. Finally, forming the version space containing classifiers consistent with the explanation-based generalization of data provides a way to integrate background domain theories into learning. The remainder of this paper describes the details of how incremental version-space merging can learn from these disparate types of information.

4. Emulating the Candidate-Elimination Algorithm

Since the work presented here claims to generalize the original version-space approach, it is first important to guarantee that the new approach maintains the functionality of the original. This section does so by showing how the candidate-elimination algorithm can be emulated using incremental version-space merging—the results after processing each example with incremental version-space merging will always be the same as those after each step of the candidate-elimination algorithm (Hirsh, 1990). The emulation furthermore modestly generalizes the candidate-elimination algorithm, in that it allows the first example to be negative and does not require the "single-representation trick" (Dietterich *et al.*, 1982) (*i.e.*, that for each instance there is a classifier that classifies only it as positive and everything else as negative).[2] A comparison of computational complexities is given in Appendix A.

4.1. Approach

The key idea for emulating the candidate-elimination algorithm with incremental version-space merging is to form the version space of classifiers strictly consistent with each individual instance and intersect these version spaces with incremental version-space merging.

However, this requires forming the version space of classifiers consistent with a single instance in boundary-set representation. This is done as follows. If the training instance is a positive example, its S set is assigned the most specific elements in the language that include the instance. When the single-representation trick holds, the S set contains the instance as its sole element. When it does not hold, the learning-task-specific method that generates instance version spaces must determine the set of most specific classifiers in the language that cover the instance. The new G set contains the single, universal concept that classifies everything as positive. If the training instance is a negative example, its S set is taken to be the single, empty concept that says that classifies everything as negative, and its G set is the set of minimal specializations of the universal concept that don't cover the instance. This forms the boundary-set representation of the version space of classifiers consistent with a single training instance.

Table 3. Emulation Example: Instance Boundary Sets.

Example	Class	Instance S	Instance G
1. [small, cube]	+	{[small, cube]}	{[any-size, any-shape]}
2. [small, sphere]	−	{∅}	{[large, any-shape], [any-size, polyhedron]}
3. [large, octahedron]	−	{∅}	{[small, any-shape], [any-size, sphere], [any-size, cube], [any-size, pyramid]}
4. [small, pyramid]	+	{[small, pyramid]}	{[any-size, any-shape]}

The emulation can be summarized as follows:

1. Form the version space of all classifiers consistent with an individual instance.
2. Intersect this new version space with the version space for past data to generate a new version space.
3. Return to Step 1 for the next instance.

Note that this merely instantiates the general incremental version-space merging algorithm given earlier, specifying how individual version spaces are formed. Furthermore (as mentioned above), in contrast to the candidate-elimination algorithm, this emulation allows the first instance to be negative and does not require the single-representation trick.

4.2. Example

To demonstrate this incremental version-space merging implementation of the candidate-elimination algorithm, the robot learning task presented in Section 2 will again be used. The initial version space has boundary sets S={∅} and G={[any-size, any-shape]} (where ∅ represents the empty concept that classifies everything as negative). The version space for the first positive instance, a small cube, has the boundary sets S={[small, cube]} and G={[any-size, any-shape]} (Step 1 of the incremental version-space merging algorithm), and when merged with the initial version space simply returns the instance version space (Step 2). This is obtained using the version-space merging algorithm (Section 3): in its first step the most specific common generalizations of pairs from the two original S sets are formed—here it is the most specific common generalizations of ∅ and [small, cube]: {[small, cube]}; the second step prunes away those that are not minimal and those not covered by elements of the two original G sets, but here nothing need be pruned. Similarly, for the new G set the most general common specialization of [any-size, any-shape] and [any-size, any-shape] is [any-size, any-shape], and nothing need be pruned.

The version space for the second, negative example, a small sphere, is defined by S={∅} and G={[large, any-shape], [any-size, polyhedron]}—nothing more general excludes the negative instance. When merged with the previous version space, the new boundary sets are S={[small, cube]} and G={[any-size, polyhedron]}. This

Table 4. Emulation Example: Resulting Boundary Sets.

Example	Class	Resulting S	Resulting G
		{∅}	{[any-size, any-shape]}
1. [small, cube]	+	{[small, cube]}	{[any-size, any-shape]}
2. [small, sphere]	−	{[small, cube]}	{[any-size, polyhedron]}
3. [large, octahedron]	−	{[small, cube]}	{[any-size, cube], [small, polyhedron]}
4. [small, pyramid]	+	{[small, polyhedron]}	{[small, polyhedron]}

is obtained by taking for the new S set the most specific common generalizations of [small, cube] and ∅ that are more specific than [any-size, any-shape] and one of [large, any-shape] and [any-size, polyhedron]—*i.e.*, covered by elements of the two original G sets. This simply yields {[small, cube]}. For the new G set the most general common specializations of [any-size, any-shape] and [large, any-shape]—{[large, any-shape]}—and the most general common specializations of [any-size, any-shape] and [any-size, polyhedron]—{[any-size, polyhedron]}—are taken for the new G set, b[illegible]arge, any-shape] must be pruned since it is not more general than an element of one of the original S sets.

The third, negative example, a large octahedron, has boundary sets S={∅} and G= {[small, any-shape], [any-size, sphere], [any-size, cube], [any-size, pyramid]}. Merging this with the preceding boundary sets yields S={[small, cube]} and G={[any-size, cube], [small, polyhedron]}. Finally, the last, positive instance, a small pyramid, has boundary sets S={[small, pyramid]} and G={[any-size, any-shape]}, resulting in the final version space S=G={[small, polyhedron]}. Tables 3 and 4 summarize the processing done by the emulation. Table 3 gives the boundary sets for the version space generated for each instance (Step 1 of each iteration of incremental version-space merging). Table 4 gives the boundary sets obtained by intersecting the instance version spaces (Step 2 of each iteration). Note that at each step the version space after merging (Table 4) is the same as that produced by the candidate-elimination algorithm (Table 2).

5. Learning from Ambiguous Data

The previous section demonstrated that incremental version-space merging maintains the functionality of the original version-space approach by showing how it can emulate the candidate-elimination algorithm. This section shows how this emulation can be extended to deal with *ambiguous data*, such as when an attribute's value is not uniquely known. As such, it provides the first demonstration of the increased range of information that can be used by incremental version-space merging.

5.1. Ambiguous Data

The idea of ambiguous data is that, in real applications, the identity of an instance is not always known with absolute certainty. One can imagine many scenarios where such ambiguous data can arise:

- It may be difficult to distinguish the exact color of some object, such as whether an object is black or brown.
- The exact value for some continuous attribute, such as a patient's age or weight in a medical context, may not be known but some range of possible values may be given.
- One may be missing a value altogether, but know the set of possible values it could in theory take on.

Forcing a learning-system user to specify every example unambiguously introduces noise into the data, since the specification would merely represent a guess as to the true identity of the example, and as such could easily be incorrect. Far better would be to allow the user to specify examples explicitly including whatever ambiguities are present. This section describes how incremental version-space merging can be used to learn from such ambiguous data.

5.2. Approach

The incremental version-space merging emulation of the candidate-elimination algorithm can be used fairly directly to do concept learning even when given ambiguous data. The basic idea is to form the version space for an ambiguous example by identifying the set of all classifiers consistent with any potential identity for the ambiguous instance; its version space should include classifiers consistent with any possible interpretation of the instance. For example, if a positive instance is known to be either black or brown, its version space should contain all classifiers that include the black case plus all classifiers that include the brown case. This version space can be viewed as the union of two version spaces, one for black and the other for brown.

Defining a version space requires defining the contents of its boundary sets. For ambiguous training data this is done by setting the boundary sets to the most specific and general classifiers consistent with some possibility for the training instance. If the instance is a positive example, the S set contains the most specific classifiers that include at least one possible identity for the ambiguous instance. If the single-representation trick holds, the S set contains the set of all instances that the training instance might truly be. The G set contains the universal concept that classifies everything as positive. If the instance is negative the S set contains the empty concept that classifies everything as negative and the G set contains the minimal specializations of the universal concept that do not include *one* of the possibilities for the uncertain data.

The algorithm can thus be summarized as follows:

1. (a) Form the set of all instances the given instance might be.
   (b) Form the version space of all classifiers consistent with any individual instance in this set.
2. Intersect the version space with the version space for past data to obtain a new version space.
3. Return to Step 1 for the next instance.

This is again just the algorithm of Section 3, with a new specification of how individual version spaces are formed. This instantiation of incremental version-space merging has the additional property that if there is no ambiguity, the algorithm behaves exactly like the candidate-elimination algorithm emulation of the previous section.

Note that ambiguous data cannot be handled through the use of internal disjunction (Michalski, 1983). An example of an internal disjunction would be the classifier [small, octahedron$\vee$cube]. This says that small objects that are either octahedra or cubes will be positive. Both small octahedra *and* small cubes are included as positive by it. An ambiguous instance, on the other hand, cannot guarantee that both will be positive; it may be that only small cubes are positive, whereas the internal disjunction would errantly include small octahedra. A correct solution to handling ambiguous data must not rule out classifiers whose extension only includes one of the possible identities of an ambiguous instance.

5.3. Example

To demonstrate how ambiguous data are handled, the learning task of Section 2 is again used. If the first, positive instance were known to be small and either cube or octahedron, the instance version space would have boundary sets S={[small, cube], [small, octahedron]} and G={[any-size, any-shape]}. After the second instance (a small sphere, negative example) is processed, the resulting boundary sets are S={[small, cube], [small, octahedron]} and G={[any-size, polyhedron]}. The third instance (a large octahedron, negative example) results in S={[small, cube], [small, octahedron]} and G={[any-size, cube], [small, polyhedron]}. It takes the final instance (a small pyramid, positive example) to finally make the version space converge to S=G={[small, polyhedron]}. The processing of the incremental version-space merging emulation of the candidate-elimination algorithm in this case is summarized in Tables 5 and 6. Table 5 gives the instance version spaces, and Table 6 gives the resulting version spaces after incremental version-space merging.

6. Learning from Data with Bounded Inconsistency

The previous section provided the first demonstration of the extended range of information that incremental version-space merging can use by describing how it can learn from cases of ambiguous data. This section shows how incremental version-space merging can be

Table 5. Ambiguous Data Example: Instance Boundary Sets.

Example	Class	Instance S	Instance G
1. [small, cube] or [small, octahedron]	+	{[small, cube], [small, octahedron]}	{[any-size, any-shape]}
2. [small, sphere]	−	$\{\emptyset\}$	{[large, any-shape], [any-size, polyhedron]}
3. [large, octahedron]	−	$\{\emptyset\}$	{[small, any-shape], [any-size, sphere], [any-size, cube], [any-size, pyramid]}
4. [small, pyramid]	+	{[small, pyramid]}	{[any-size, any-shape]}

used to learn in cases of inconsistent data, and it thus provides a demonstration of a second source of information that can be used by the new version-space approach.[3]

6.1. Bounded Inconsistency

When no single classifier in the description language can distinguish between all the positive examples and all the negative examples, the data are said to be *inconsistent* with respect to the description language. In such cases no learner will be able to find a classifier that correctly classifies all the given examples, and thus the candidate-elimination algorithm would fail if given such data, since the version space of consistent classifiers is empty (in version-space parlance the version space is said to have *collapsed*). In general, learning systems must generate reasonable results even when there is no classifier consistent with all the data.

This section describes how incremental version-space merging can be used to learn from inconsistent data by addressing only a subcase of the problem called *bounded inconsistency*. The underlying assumption for this class of inconsistency is that some small perturbation to the description of any bad instance will result in a good instance. Whenever an instance is misclassified with respect to the desired final classifier, some nearby instance description has the original instance's classification.

Figure 4 shows a simple way to view this. Classifiers (such as C) divide the set of instances (I) into positives and negatives. I_1^+ is an example of a representative positive example. It is correctly classified with respect to the desired classifier C. Similarly, I_2^- is a correctly classified representative negative example. I_3^+ however is incorrectly classified as positive even though the desired classifier would label it negative. However, a neighboring instance description, I'^+_3, is near it across the border, and is correctly classified. Similarly for the incorrectly classified negative instance I_4^- and its neighbor I'^-_4. Roughly speaking, if misclassifications only occur near the desired classifier's boundary, the data have bounded inconsistency.

For example, if instances are described by conjunctions of features whose values are determined by measuring devices in the real world, and the measuring devices are known to only be accurate within some tolerance, bounded inconsistency can occur. Consider

Table 6. Ambiguous Data Example: Resulting Boundary Sets.

Example	Class	Resulting S	Resulting G
		{∅}	{[any-size, any-shape]}
1. [small, cube] or [small, octahedron]	+	{[small, cube], [small, octahedron]}	{[any-size, any-shape]}
2. [small, sphere]	–	{[small, cube], [small, octahedron]}	{[any-size, polyhedron]}
3. [large, octahedron]	–	{[small, cube], [small, octahedron]}	{[any-size, cube], [small, polyhedron]}
4. [small, pyramid]	+	{[small, polyhedron]}	{[small, polyhedron]}

an instance that is classified as positive, with a feature whose value is 5.0 (obtained by the real-world measuring device). If the tolerance on the measurement of the feature's value is 0.1, the instance could really have been one with feature value 4.9. If the "true" positive instance were 4.9, and the instance that really has value 5.0 would have been negative, a misclassification error has occurred. If the tolerance information is correct, for every incorrect instance there is a neighboring correct instance description, all of whose feature values are no more than the tolerance away from the original instance's value for that feature. This is an example of bounded inconsistency.

6.2. *Approach*

Informally, the approach taken here for learning from inconsistent data treats all instances as ambiguous. Rather than forming the space of all classifiers consistent with the instance, the space of all classifiers consistent with the instance or some instance description not too distant from the given one is formed (where "not too distant" is defined as appropriate for the particular learning task). The approach handles cases of inconsistent data in which

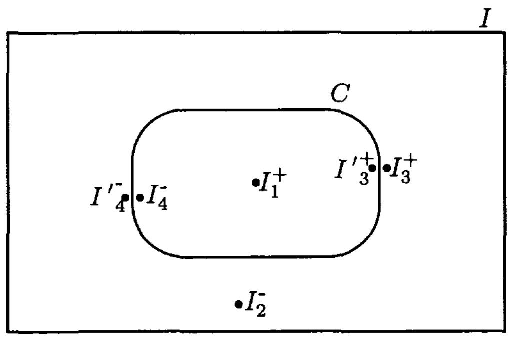

Figure 4. Pictorial Representation of Bounded Inconsistency.

each nonrepresentative or misclassified instance has at least one nearby description that has the original instance's classification as its correct classification.

All instances are treated as capable of being any instance within some region of neighboring instance descriptions. For each instance the set of neighboring instance descriptions is determined, and the version space of classifiers to be considered contains those classifiers consistent with the instance or one of its neighbors. The net effect is that all instances are "blurred," and version spaces contain all classifiers that generate a correct classification for at least one instance in each "blur".

The general problem can be viewed as follows:

Given:

- *Training Data*: Positive and negative examples of the classifier to be identified.
- *Definition of Nearby*: A method that determines all instances near a given instance.
- *Classifier Description Language*: A language in which the final classifier must be expressed.

Determine:

- A set of classifiers in the classifier description language consistent with the data or nearby neighbors of the data.

The method proceeds as follows:

1. (a) Determine the set of instances near a given instance.
   (b) Form the version space of all classifiers consistent with some instance in this set.
2. Intersect this version space with the version space generated from all past data.
3. Return to the first step for the next instance.

If an instance is positive, the version space of all classifiers consistent with some instance in the set of neighboring instances has as an S set the set of most specific classifiers that cover at least one of the instances, and as a G set the universal concept that classifies everything as positive. If the single-representation trick holds, the S set contains the set of neighboring instances themselves. If an instance is negative the S set contains the empty concept that classifies everything as negative and the G set contains all minimal specializations of the universal concept that excludes the instance or one of its neighbors. This instantiation of incremental version-space merging has the additional property that if there is no inconsistency (*i.e.*, the definition of nearby is the identity function that returns only the given example), the algorithm behaves exactly like the candidate-elimination algorithm emulation of Section 4.

6.3. Searching the Version Space

Ideally the result of this learning process would be a singleton version space containing the desired classifier. However, if not enough data are given, the final version space will contain more than one classifier. This also may happen if the definition of nearby is too "generous"—*i.e.*, if every instance has many nearby instances. In this case, there will be many classifiers in each instance version space; in some cases no set of instances will permit convergence to a single classifier. The definition of nearby should be generous enough to guarantee that the desired classifier is never thrown out by any instance, but not too generous to include too many things (or in the worst case, everything).

Also, it is often impractical to wait for enough instances to converge to a single classifier. As each instance throws away candidate classifiers, the version space gets smaller and smaller. As the version space decreases in size, the probability that a randomly chosen instance will make a difference—will be able to remove candidate classifiers—becomes smaller and smaller. The more data processed, the longer the wait for another useful instance. Therefore it will sometimes be desirable due to time considerations to use the small but nonsingleton version space (before converging to a single classifier) to determine a usable result for learning.

Thus a situation can arise in which the final version space after processing data has multiple classifiers. However, learning is usually expected to return some single element, rather than a version space of possible results. To be realistic an extension to the core version-space learning algorithm is therefore required so that only one classifier is returned. The most straight-forward method for returning a single result (applicable even to the original candidate-elimination algorithm) is to simply return an arbitrary element of the version space.

More sophisticated methods can also be considered, though, in this more specific context of learning from data with bounded inconsistency. For example, the original, unperturbed training data can be used to define metrics over the space of results remaining in the version space. One such metric uses the observation that each classifier is consistent with only some of the original, unperturbed data; some of the original data must be "blurred" to be consistent with the classifier. This suggests computing the portion of the original, unperturbed data that is covered by the given classifier. The classifier with best coverage is then selected as the final result of learning. A second possible metric uses the observation that for each classifier the classification of some of the original data is unchanged even if perturbed by the maximum amount allowed by the definition of nearby. This suggests computing for each classifier the number of instances whose classifications remain unchanged even if maliciously perturbed. Whichever classifier maintains the classification of the largest number of correctly classified instances should be selected as the result of learning.

6.4. Selecting an Appropriate Definition of Nearby

The approach described here to handling bounded inconsistency is clearly sensitive to the definition of "nearby." If given the proper definition of nearby and a sufficient number of examples, learning will converge to a singleton version space. However, if the definition of nearby is too small, the version space will collapse with no consistent classifier; if it is too large, each instance will have many neighbors, and instance boundary sets will be quite large, which makes the approach computationally infeasible. Furthermore, the final version space will likely be too big for any intelligent metric-based search for a best classifier.

The ideal situation for this approach would be when an appropriate definition of nearby is given or otherwise known for the particular domain. However, when no appropriate definition of nearby is known a priori, some method for selecting a value must be specific. One method for doing so is to exploit the fact that version spaces provide a signal when the choice of nearby is too small, namely in such cases the version space collapses. The idea, then, is simply to start off assuming no inconsistency and apply version spaces; if the version spaces collapses, increment the definition of nearby a small amount and try again. This process repeats until an adequate definition of nearby is found, stopping at the first value for which the version space doesn't collapse. This of course assumes learning occurs in an environment where all the data is available at the start of learning, and requires a "one-parameter" definition of nearby, for which distance grows monotonically with increasing parameter value.

6.5. Example

To demonstrate this technique Fisher's (1936) iris data are used. The data are inconsistent, and thus the version space of classifiers strictly consistent with the data is empty (hence the candidate-elimination algorithm does not apply). In addition, this problem has been in use for over 50 years, and it thus provides an easy way to calibrate this noise-tolerant extension of version spaces by comparing its results on this problem to the results of other techniques used on this same data.

The particular problem is that of classifying examples of different kinds of iris flowers into one of three species of irises: setosa, versicolor, and viginica. There are 150 instances, 50 for each class; instances are described using four features: sepal width, sepal length, petal width, and petal length. The units for all four are centimeters, measured to the nearest millimeter. For example, one example of versicolor had sepal length 7.0cm, sepal width 3.2cm, petal length 4.7cm, and petal width 1.4cm. The learning problem is to find rules for classifying irises into one of the three classes based on measurements of sepal width and length and petal width and length.

6.5.1. Implementation

Using incremental version-space merging on this data requires a number of details to be specified to make an implementation of the approach possible. These include specifying the particular description language used for classifiers, how an appropriate definition of nearby is selected, how an element of the final version space is selected as the result of learning, and how version spaces can be used for a three-class learning problem. This section considers each of these in turn.

The Classifier Description Language. As in most uses of version spaces with real-valued data, a fairly straight-forward approach was used for specifying a description language in which classifiers must be expressed. In particular, the description language was chosen to be axis-parallel hyper-rectangles, *i.e.*, conjunctions of ranges of the form $a \leq x \leq b$ for each feature, where a and b are measurements in millimeters, and either may be absent. Nothing more refined than millimeters need be considered since data are only specified to the nearest millimeter. An example of a legal classifier description in this language has [0.7cm $\leq$ petal length $\leq$ 2.6cm] and [sepal width $\leq$ 1.9cm].

Defining Nearby. In this domain there was no given definition for nearby, and thus one had to be selected. The basic definition used here assumes that two examples are neighbors if they differ on each feature by an amount less than some maximum "tolerance," with no restriction on how many features may differ in this fashion. This means that the complete set of neighbors of an example is the set of all instance descriptions located within a hyper-rectangle surrounding the given example whose sides are all of this maximum length.

However, an important question this leaves unanswered is what is an appropriate definition for this tolerance. If it is too small, learning will fail, and if it is too large learning will take too long. To determine the proper value the method described in Section 6.4 for slowly increasing the definition of nearby was used. Initially no bounded inconsistency is assumed, so the approach performs like the candidate-elimination algorithm. However, if the version space collapses the definition of nearby is increased by the minimum amount possible, namely 1 millimeter. At each step, if learning fails (*i.e.*, the version space collapses), the tolerance is increased by one and a new learning attempt is begun. When learning does succeed, some element of the resulting version space is selected as the result of learning, as described next.

Searching the Version Space. As discussed in Section 6.3, learning also requires a method for selecting an element of the version space as the final result of learning if after processing all the data the version space is nonsingleton. The basic method used here is to approximate the first metric suggested in Section 6.3, by enumerating a subset of the version space rather than enumerating the entire version space. The subset of the version space that is actually explored is those elements of the version space that are one quarter, two quarters, and three quarters of the way between each S-set and each G-set element; the approach described earlier then selects as the result of learning the classifier that had lowest error rate on the original, unperturbed data from the union of the S set, the G set, and this set of intermediate classifiers. In cases where there is a tie for lowest error one result is selected at random from among the tied classifiers.

Learning Three Classes. Version spaces are used in problems where there are two sets of data, one containing positive data and the other negative data, and attempts to form a classifier for the positive data. However, the iris learning problem has three classes, so some additional mechanism is necessary to make it possible to learn in this domain. The approach taken here was effectively to learn a form of decision list (Rivest, 1987). More specifically, given three classes A, B, and C, learning first forms a rule for class

A, lumping examples of classes B and C into a single collection of negative data for A. Learning next attempts to distinguish between classes B and C by finding a rule for class B. The result of learning is a decision list that looks like the following:

```
If rule-for-A then return A
Else if rule-for-B then return B
Else return C.
```

However, this high-level description leaves a number of questions unanswered. First, given data for three classes, which class should serve the role of A, which the role of B, and which the role of C? Second, in learning to distinguish between B and C, examples of B should clearly be treated as positive examples and examples of C as negative examples. However, should examples of A also be included as negative examples for B, or should they instead be excluded from this second learning effort?

The answer for both these questions was to actually try all possible answers, and select whichever was best, as determined using leave-one-out cross-validation (Weiss and Kulikowski, 1991).[4] This results in twelve separate three-class learning problems, six for each way of assigning setosa, versicolor, and viginica to classes A, B, and C above, and for each one, data from class A was either included or excluded as negative data in forming a classifier for class B. Whichever resulting decision list has the lowest error-rate assessment is selected as the final result of learning.

Summary of Implementation. The preceding text has described the various issues that must be confronted in applying incremental version-space merging to Fisher's iris data. The core algorithm is the basic two-class incremental version-space merging learning method described in Section 6.2. Wrapped around this is a loop that determines an appropriate value for the definition of nearby. Once a definition of nearby is found, learning will usually return a nonsingleton version space, and thus a search of a select subset of the version space is conducted to find the classifier with lowest error on the original noisy training data. This results in a noise-tolerant two-class learning method that is in turn used to learn a decision list for the three-class iris problem by first forming a classifier for one class, then a second classifier that distinguishes between the remaining two classes, with cross-validation selecting the best way of converting the three-class learning problem in this fashion into two two-class learning problems.

There is one subtle point that the details given thus far neglect to emphasize, namely, how an appropriate tolerance value is determined during each cross-validation run. The straight-forward way would be to let each cross-validation run determine its own values for this parameter by slowly increasing the value until the version space does not collapse. However, this greatly adds to the execution time for learning, since the core of each run would have to conduct multiple, tolerance-setting trials. The approach that was used here instead was to take the values for the tolerance that were determined when generating results on the full set of data, and passing these along for use by the cross-validation runs. While this means that the test data (*i.e.*, each hold-out example) contributes to the selection of the tolerance value and thus could in theory bias error assessments, given the size of the data sets this was not expected to be particularly serious and was "overruled" in favor of efficiency considerations.

6.5.2. Results

Cross-validated error rates for the resulting three-class learning approach described above (labeled IVSM) are presented in Table 7, along with those of other learning methods on this data (Weiss and Kulikowski, 1991). Since there are often ties for the classifier that does best on the noisy training data and in such cases one of the tied classifiers is selected at random, the reported error-rate assessment is the average of the cross-validated error-rate assessments for fifty separate learning trials. The resulting error rate of 4.1% demonstrates that the approach described here performs well, although other methods do outperform it. More importantly, however, these results demonstrate the increased generality of incremental version-space merging by showing how it can achieve credible learning results from inconsistent data (in contrast to the candidate-elimination algorithm, which totally fails to apply in such situations).[5]

7. Combining Empirical and Explanation-Based Learning

The previous sections demonstrated how incremental version-space merging can be used to learn both from ambiguous data and from data with bounded inconsistency; this gives two examples of the increased range of information that can be used by incremental version-space merging. This section provides a third example, by demonstrating how incremental version-space merging can be used to implement and extend Mitchell's (1984) proposal for combining empirical and explanation-based learning. Mitchell's original proposal was presented in the context of learning search control rules, and was essentially to apply explanation-based generalization (EBG) (Mitchell *et al.*, 1986) to ground training data, then do empirical learning on the generalized data.[6] In addition to implementing Mitchell's proposal for the first time, this use of incremental version-space merging goes beyond Mitchell's original proposal, and handles incomplete theories as well as theories when there are multiple conflicting explanations for a single instance. The result provides a way to integrate background domain theories into version spaces,

Table 7. Iris Error Rates.

Method	Error
Linear	.020
Optimal rule size 2	.020
Neural net (ODE)	.027
Quadratic	.027
Neural net (BP)	.033
Nearest neighbor	.040
PVM rule	.040
IVSM	**.041**
CART tree	.047
Bayes independence	.067
Bayes 2nd order	.160

and thus provides a final demonstration of the increased range of information that can be used by incremental version-space merging.

7.1. Approach

The key idea to integrating explanation-based generalization into version space is to form for each instance the version space consistent with the results of explanation-based generalization for that instance, rather than simply the version space consistent with ground data. The problem addressed is:

Given:

- *Training Data*: Positive and negative examples of the classifier to be identified.
- *Classifier Description Language*: A language in which the final classifier must be expressed. It is a superset of the instance description language, and is where generalization hierarchies would appear.[7]
- *Positive-Data Domain Theory* (optional): A set of rules and facts for proving that an instance is positive. Proofs terminate in elements of the instance description language.
- *Negative-Data Domain Theory* (optional): A set of rules and facts for proving that an instance is negative. Proofs terminate in elements of the instance description language.

Determine:

- A set of classifiers in the classifier description language consistent with the data.

The method of combining EBG with incremental version-space merging to solve the problem above processes a sequence of instances as follows, starting with the first instance:

1. (a) If possible, apply EBG to the current instance to generate a generalized instance. Do so for all possible explanations. If no explanation is found, pass along the ground data.
   (b) Form the version space of all classifiers consistent with the (perhaps generalized) instance. If there are multiple explanations include those classifiers consistent with *any* single explanation.
2. Intersect this version space with the version space generated from all past data.
3. Return to the first step for the next instance.

This is again an instantiation of the general incremental version-space merging algorithm given earlier.

The basic technique is to form the version space of classifiers consistent with the explanation-based generalization of each instance (rather than the version space of classifiers consistent with the ground instance). The version space for a single training instance reflects the explanation-based generalization of the instance, representing the set of classifiers consistent with all instances with the same explanation as the given instance. The merging algorithm has the effect of updating the version space with the many examples sharing the same explanation, rather than with the single instance. In this manner irrelevant features of the instances are removed, and learning can converge to a final classifier using fewer instances.

7.2. Imperfect Theories

This approach to integrating EBG into version spaces goes beyond Mitchell's proposal of updating version spaces with generalized examples, and handles some forms of imperfect theories. For example, the approach can be used even when the given domain theories are unable to explain (and generalize) all the given training data. The version space of all classifiers consistent with a non-generalized instance—whether negative or positive—can always be formed. Instead of using EBG, the version space consists of all classifiers consistent with the instance, rather than its explanation-based generalization. If a theory for only explaining positive instances exists, negative instances can be processed without using EBG. If an incomplete theory exists (*i.e.*, it only explains a subset of potential instances), when an explanation exists the version space for the explanation-based generalization of the instance can be used, otherwise the pure instance version space should be used. Finally, like the two preceding applications of incremental version-space merging for ambiguous and inconsistent data, when there is no domain theory the learner degenerates to behave like the candidate-elimination algorithm. The net result is a learning method capable of exhibiting behavior at various points along the spectrum from knowledge-free to knowledge-rich learning.

This technique also applies to theories that can explain an example in multiple ways, where only one of the explanations need be correct (and the result of EBG on other, incorrect explanations could yield incorrect generalizations). In such cases the version space of classifiers consistent with one or more of the potential results of EBG is formed. EBG is applied to every competing explanation of an instance, each yielding a competing generalization of the instance. The space of candidate generalizations for the single instance contains all classifiers consistent with *at least one* of the competing generalizations. After multiple instances the final classifier must be consistent with one or more of them. The situation is similar to that of ambiguous data (Section 5), only here it is unknown which *explanation* is correct. Like the earlier treatment of ambiguous data, the version space contains all classifiers consistent with at least one of the possibilities.

The version space of all classifiers consistent with at least one explanation-based generalization of the instance is the union of the version spaces of classifiers consistent with each individual explanation-based generalization. For positive examples this union has as its S boundary set the set of competing explanation-based generalizations, and the G boundary set contains the universal concept that classifies everything as positive. If one result of EBG is more general than another (*e.g.*, one mentions a superset of the training-instance facts mentioned by the other), only the more specific result is kept in the S set. Over multiple instances these version spaces consistent with the explanation-based generalizations are incrementally intersected to find the space of classifiers consistent with the generalized data.

Finally, this approach is also useful given theories for explaining *negative* data, when the system is provided with a theory capable of explaining why an instance is negative (Gupta, 1987), (Minton, 1988), (Mostow and Bhatnagar, 1987). Such theories are particularly common in systems that use explanation-based learning to form search-control rules from problem-solving traces, where, for example, the theory explains why a particular operator should *not* be used in a given state, *e.g.*, that the operator does not apply, or that it leads to a non-optimal solution (Minton, 1988). This theory is then used to generalize the negative instance to obtain a generalization covering all instances that are negative for the same reason. Incremental version-space merging would then use this generalized instance by setting the S set equal to the empty concept that classifies everything as negative, and setting the G set equal to all minimal specializations of the universal concept that do not cover the generalized negative instance. If there are multiple, competing explanations, the G set contains all minimal specializations that do not cover at least one of the potential generalizations obtainable by EBG using one of the explanations.

7.3. *Examples*

To illustrate this third instantiation of incremental version-space merging, two examples are presented. The first demonstrates how the resulting learning method generalizes beyond the specific results obtainable with either the candidate-elimination algorithm or EBG alone, as well as how theories for negative data can be used. The second demonstrates how the approach deals with multiple explanations.

7.3.1. *Cup Example*

The first example of this use of incremental version-space merging to integrate EBG into version spaces demonstrates how a definition of `cup` can be learned given incomplete knowledge about cups plus examples of cups. It is based on the examples given by Mitchell *et al.* (1986) and Flann and Dietterich (1986).

The following is the domain theory used (written in Prolog notation):

```
cup(X):-holds_liquid(X),
        can_drink_from(X),
        stable(X).
holds_liquid(X):-pyrex(X).
holds_liquid(X):-china(X).
holds_liquid(X):-aluminum(X).
```

```
can_drink_from(X):-liftable(X),
                   open_top(X).
liftable(X):-small(X).
stable(X):-flat_bottom(X).
```

It can recognize and explain some, but not all, cups. The description language used for this problem by empirical learning utilizes generalization hierarchies, including the knowledge that pyrex, china, and aluminum are nonporous materials, and that black and brown are dark colors. Note that this information is not present in the domain theory, but is known to be true in general. Empirical learning has many such possible generalizations. The goal for learning is to determine which of these potential generalizations that go beyond what the domain theory would justify (*e.g.*, those mentioning nonporous material) are relevant.

Learning begins with the first, positive example:

```
china(cup1).
small(cup1).
open_top(cup1).
flat_bottom(cup1).
black(cup1).
```

EBG yields the rule

```
cup(X):-china(X),
        small(X),
        open_top(X),
        flat_bottom(X).
```

written "[china, small, open, flat, anycolor]" for short.[8] This forms the S set for the version space of the first instance (and the first step of incremental version-space merging), and its G set contains the universal concept [anymaterial, anysize, anytop, anybottom, anycolor]. The second step of incremental version-space merging intersects this with the initial full version space, which gives back this first-instance version space.

Incremental version-space merging then returns to its first step for the next, positive instance, which is:

```
pyrex(cup2).
small(cup2).
open_top(cup2).
flat_bottom(cup2).
brown(cup2).
```

EBG results in the rule

```
cup(X):-pyrex(X),
        small(X),
        open_top(X),
        flat_bottom(X).
```

The S set for this instance's version space contains the result of EBG, namely [pyrex, small, open, flat, anycolor], and its G set contains the universal concept. Merging this with the version space for the first iteration yields a version space whose S set contains [nonporous, small, open, flat, anycolor] and whose G set contains the universal concept.

The final instance is a negative example:

```
aluminum(can1).
small(can1).
closed_top(can1).
flat_bottom(can1).
white(can1).
```

For this example the theory of negative data is assumed to include the following rules (among others):

```
not_a_cup(X):-
     cannot_drink_from(X).
cannot_drink_from(X):-
     closed_top(X).
```

EBG yields the following rule:

```
not_a_cup(X):-closed_top(X).
```

This is then used to determine the most general classifiers that exclude this generalized case of not_a_cup, namely {[anymaterial, anysize, open, anybottom, anycolor]}, which forms the G set of the version space for this third instance. The S set contains the empty concept.

When this third instance version space is merged with the result of the previous two iterations of incremental version-space merging, the resulting S set contains [nonporous, small, open, flat, anycolor] and the resulting G set contains [anymaterial, anysize, open, anybottom, anycolor]. Note that the domain theories have done part of the work, with the color attribute being ignored and only the third attribute being deemed relevant for the negative instance, but empirical learning determining nonporous. Further data would continue refining the version space. However, it is already known that whatever the final rule, it will include small, nonporous, open-topped objects, like Styrofoam cups, which the original theory did not recognize as cups.

This simple domain also demonstrates the point made earlier about the technique degenerating to look like the candidate-elimination algorithm. Consider the same examples, only without the domain theory present. At each iteration the resulting version space would be exactly the same as would be created by the candidate-elimination algorithm. The final version space would have an S set containing [nonporous, small, open, flat, darkcolor], and a G set containing two elements: [anymaterial, anysize, open, anybottom, anycolor] and [anymaterial, anysize, anytop, anybottom, darkcolor]. This version space contains more elements than the corresponding version space using EBG.

7.3.2. Can_put_on_table Example

The previous subsection gave an example of how incremental version-space merging can utilize both positive- and negative-example domain theories, and demonstrated how it generalizes beyond what is obtainable by either incremental version-space merging or EBG alone. This subsection presents a second example that demonstrates how this application of incremental version-space merging can use domain theories that permit multiple competing explanations.

In particular, consider the following simple domain theory for `can_put_on_table`:

```
can_put_on_table(X):-stable(X),
                     small(X).
can_put_on_table(X):-stable(X),
                     light(X).
stable(X):-flat_bottom(X).
```

It provides two potential explanations for when an object can successfully be placed on a table: the object must be stable, and either small or light. The desired result of learning should be a rule that flat-bottomed, light objects can be placed on a table. Note that the given theory cannot be used to predict the classification of an unclassified object, since the theory can also explain flat-bottomed, small objects as positive, and thus the theory will classify some potentially negative examples as positive. However, once an example is known to be positive it is possible to generate an explanation of why it is positive. The goal for learning is to find a definition consistent with the data plus the subset of the theory that actually explains the observed classifications of the data. Furthermore, when there are multiple competing explanations of a given positive instance, later instances should allow determining which of the competing explanations is consistent across all data.

For example, given a can as a positive example of an object that can be placed on a table:

```
flat_bottom(can1).
small(can1).
light(can1).
```

the first step of the incremental version-space merging process uses EBG to form two rules, each corresponding to a different explanation:

```
can_put_on_table(X):-
    flat_bottom(X),
    small(X).
can_put_on_table(X):-
    flat_bottom(X),
    light(X).
```

These will be abbreviated to "[flat, small, anyweight]" and "[flat, anysize, light]". The resulting instance version space is bounded by an S set containing these two classifiers and a G set containing the universal concept that classifies everything as positive. Intersecting this version space with the initial version space that contains all classifiers in the description language gives back the instance version space.

Returning to the first step of the learning process for the following positive, second instance,

```
flat_bottom(cardboard_box1).
big(cardboard_box1).
light(cardboard_box1).
```

EBG can only generate one rule:

```
can_put_on_table(X):-
    flat_bottom(X),
    light(X).
```

This results in an instance version space containing [flat, anysize, light] in the S set and the G set containing the universal concept. Merging the two instance version spaces (step two of incremental version-space merging) results in an S set with the single element [flat, anysize, light] and the G set containing the universal concept.

As an example of dealing with negative data with no negative-instance domain theory, consider the following negative example of `can_put_on_table`:

```
round_bottom(bowling_ball1).
small(bowling_ball1).
heavy(bowling_ball1).
```

The version space of classifiers that do not include it has an S set that contains the empty concept and a G set that contains three classifiers: {[flat, anysize, anyweight], [anybottom, large, anyweight], [anybottom, anysize, light]}. When merged with the version space for past data, incremental version-space merging yields a version space whose S set contains [flat, anysize, light] and whose G set contains the two classifiers [flat, anysize, anyweight] and [anybottom, anysize, light]. Subsequent data would further refine this version space.

7.4. *Constraints on the Classifier Description Language*

While this section has described how background domain theories can be utilized by incremental version-space merging, there are some limitations. Most notably (as mentioned earlier), there are constraints on the form that elements of the description language may take. These constraints arise from two sources: constraints on the form of results that EBG can generate, and constraints due to computational tractability.

Like much work in concept learning (especially with version spaces), this section has assumed that the description language is conjunctive. This is due to both the usual computational reasons (computing minimal generalizations, maximal specializations, and relative generality in such languages is often inexpensive), as well as the fact that EBG generates implications whose antecedents are conjunctions. This work makes an assumption similar to the single-representation trick: that all possible results of EBG are in the description language. Without such an assumption it would be necessary to find the most specific terms in the language consistent with results of EBG. In theory the approach described here would work with any language, but determining instance version spaces may be difficult.

However, the use of version spaces requires further restrictions on description languages. Even if all results of EBG are in the language, it can be NP-complete just to test arbitrary conjunctions for relative generality. This version-space approach requires testing the relative generality of definitions, as well computing the minimal generalizations and maximal specializations of two definitions, and thus this work is constrained to use languages in which these operations are tractable.

While these properties hold for the languages used in the preceding examples, where the language consists of conjunctions of unary predicates over a fixed set of features and a single variable, there are more sophisticated representations that would also be appropriate for use by this version-space approach. One example is the language used by LEX (Mitchell *et al.*, 1983), where descriptions are defined in terms of a grammar. A second, fairly sophisticated language is the knowledge-representation language *k*-CLASSIC (Cohen and Hirsh, 1992). In both cases computing relative generality, minimal generalizations, and maximal specializations is tractable.

8. Related Work

Given the influential role version spaces have played in inductive concept learning, they have received significant attention over the years, including various attempts to extend version spaces in assorted directions. The first part of this section surveys a number of such attempts. In addition, however, this paper has described a generalization of version spaces that makes them able to use a wider range of information in learning; thus the latter parts of this section consider other inductive learning efforts that use similar types of information.

8.1. Extending Version Spaces

This subsection surveys past work on extending version spaces beyond strict consistency with data. Although each effort attempts to extend version spaces in a different direction, there are three questions that will be asked about each:

- In what way does the work extend version spaces?
- Does the work change the core learning algorithm (the candidate-elimination algorithm), or just add some additional mechanism on top of it?
- Is the extension compatible with incremental version-space merging?

The first research efforts to be considered here concern cases when a version space collapses. Section 6.1 defined inconsistency relative to a description language: "When no single classifier in the description language can distinguish between all the positive examples and all the negative examples, the data are said to be *inconsistent* with respect to the description language." This definition of inconsistency is relative to the description language because the collapse of a version space can signal either the presence of noisy data or the use of an insufficient description language. Two notable efforts have attempted to extend version spaces to apply in such situations. The first, in Mitchell's (1978) original presentation of version spaces, focuses on cases when collapse is due to noisy data; the second, in Utgoff's thesis work (Utgoff, 1986), addresses cases when collapse is due to an inadequate description language.

The core idea of Mitchell's approach to generalizing version spaces to deal with noisy data was to maintain in parallel version spaces for various subsets of the data. When no classifier is consistent with all data, his approach considers those classifiers consistent with all but one instance. As more inconsistency is detected, the system uses version spaces based on smaller and smaller subsets of the data, which the system maintains in parallel throughout learning, up to some absolute bound that the user must provide on the maximum number of noisy examples (so that the system knows how many boundary sets to maintain—otherwise the system would be required to maintain far too many boundary sets).

Furthermore, Mitchell also gives a clever way to represent the multiple boundary sets. To enable the use of a number of boundary sets linear in the number of noisy examples, Mitchell keeps the boundary sets for multiple version spaces in a single boundary set by modifying the candidate-elimination algorithm S-set updating method to ignore negative data, and similarly positive data are ignored by the modified G-set updating method. This eliminates the need to maintain many boundary sets with large amounts of overlap.

Mitchell's approach to learning from inconsistent data modifies the core candidate-elimination algorithm to apply to cases of noisy data, but, like the approach described in Section 6 to learning from data with bounded inconsistency, it has the property that it degenerates to look like the traditional candidate-elimination algorithm in the absence of noise. The difficulty with Mitchell's approach is that it imposes substantial space requirements, both in terms of the size of boundary sets (Mitchell's method for storing multiple boundary sets in a single set results in much larger boundary sets), as well as in the number of boundary sets that must be maintained. It furthermore requires knowing the absolute maximum number of incorrectly classified instances that can be seen (in contrast to allowing unlimited number of errors as done here, replacing it with a bound on the distance any instance may be from a correct instance). However, the approach appears to be compatible with incremental version-space merging, in that it seems as feasible to consider version spaces based on smaller and smaller subsets of a given collection of information as it is for subsets of data (although it is not clear to what extent Mitchell's implementation ideas would apply).

In contrast to Mitchell's work, Utgoff's efforts begin with the assumption that a collapsed version space signals the use of an inadequate description language. When a version space collapses, Utgoff's STABB system has two methods for expanding the description language so that reprocessing the training data will not lead to a version-space collapse. The first method exploits the fact that learning occurs in the context of forming search control rules for the LEX system (Mitchell *et al.*, 1983); it attempts to analyze problem-solving traces for the apparently contradictory positive and negative examples that caused the version-space collapse, and adds a new term that allows the description language to distinguish between them. Since this method is not guaranteed to always succeed, Utgoff furthermore gives a second, weaker method that *is* guaranteed to always succeed. This second method computes the "least disjunction" of the positive examples and adds the term to the description language; the result is that the training data are guaranteed to be consistent with respect to this updated description language.

Strictly speaking, STABB isn't really a generalization of version spaces: Utgoff's approach leaves the core version-space algorithm (the candidate-elimination algorithm) fixed, and instead changes things beneath it by changing the language on which the version space is based. As such, the approach is also entirely compatible with incremental version-space merging, and could provide a feasible way to deal with inadequate description languages.

The same can be said of Russell and Grosof's (1987) work on declarative biases for inductive concept learning. They describe how determinations can be used to specify the description language for a given learning task, and furthermore how nonmonotonic reasoning can be used to move to weaker biases when a given bias is found to be inadequate (such as due to version-space collapse). Like Utgoff's work, Russell and Grosof's declarative bias specification and relaxation methods leave the core version-space algorithms intact, and only change the language on which it based. As such, it too is compatible with incremental version-space merging.

A second class of work on extending version spaces addresses the class of languages for which they can be used. Not all description languages are well-behaved enough to permit the representation of a version space with boundary sets: only under certain conditions can such representational guarantees be made. A number of researchers have therefore tried to clarify and generalize such representational limitations of version spaces. Most of these efforts begin with Mitchell's (1978) original work on this topic. Mitchell uses the term "admissibility" to refer to the property a description language must have to enable the use of the candidate-elimination algorithm: a language is admissible if and only if every chain has a maximum and minimum element. Incremental version-space merging has similar formal underpinnings (Gunter *et al.*, 1991), (Hirsh, 1990), (Hirsh, 1991).

These past formal efforts analyze the conditions under which a set of classifiers can be represented by boundary sets. Recent work has attempted to generalize version spaces by proposing alternative version-space representations that have weaker conditions of applicability. For example, one recent effort (Hirsh, 1992b) has described how most version-space operations can still be supported even when one of the two boundary sets is replaced by a list of training data. In particular, boundary sets can informally be viewed as just summarizing training data: the S set summarizes the positive data seen thus far, and the G set summarizes the negative data seen thus far. The basic idea, then, is that rather than maintaining both boundary sets, only only one is maintained;

the other is replaced with a list of the data that the eliminated boundary set would have "summarized."

Although the original motivation for this single-boundary-set representation was to address tractability issues for version spaces, it also generalizes version spaces beyond the candidate-elimination algorithm. In particular, there are languages where only one of the boundary sets are well defined (for example, the version space of descriptions consistent with data when learning in CLASSIC (Borgida *et al.*, 1989), (Cohen and Hirsh, 1992) can have G sets with infinite size); in such cases boundary-set implementations of the candidate-elimination algorithm do not apply, whereas version-space learning is possible using the new representation. However, although this alternative representation does generalize version spaces, it is not fully compatible with incremental version-space merging. In particular, while version-space intersections can be computed in the new representation (thereby allowing applications of incremental version-space merging like the candidate-elimination algorithm emulation of Section 4), version-space unions cannot always be represented. This makes it impossible to deal with most of the types of information considered in this paper, such as learning from ambiguous data or using domain theories with multiple explanations.

Idestam-Almquist (1989) also considers alternative version-space representations, only as a way to extend the version-space approach to support the additional operation of retracting data (for use as a way to handle inconsistent data). Although Idestam-Almquist's representation only applies to a restricted class of conjunctive languages, within this class his system generalizes version spaces to support this additional operation. Note, of course, that the single-boundary-set representation of the previous paragraph also extends version spaces partially in this direction for a much larger range of languages; since one of the two boundary sets are replaced by a list of data, it becomes easy to support the retraction of data from this list (which will contain either only positive data if the S is replaced, or only negative data if the G set is eliminated). It is unclear to what extent Idestam-Almquist's approach is compatible with incremental version-space merging.

Finally, VanLehn and Ball (1987) take a totally different tack in generalizing the representational underpinnings of version spaces. They apply version spaces to the problem of learning context-free grammars, where it is intractable to compute the partial order imposed by the relative generality of grammars. They instead consider using a tractable approximation to the true partial order; this approximate partial order forms the basis for testing the relative generality of two grammars, as well as for computing the very boundary sets themselves. VanLehn and Ball's work extends version spaces to deal with intractable partial orders, using tractable approximate orders instead; such a generalization of version spaces to other partial orders can equally well be used with incremental version-space merging, and thus their work is entirely compatible with the work described here.

8.2. Learning from Inconsistent Data

This paper describes one generalization of version spaces that allows it to use a large range of information in learning; this includes the use of data with bounded inconsistency, as described in Section 6. This subsection considers other work on noise-tolerant learning methods outside a version-space context that can also utilize this type of information.

First, note that the particular type of information considered here—data with bounded inconsistency—is of a more focused nature than the type of information considered by most other work on noise-tolerant learning methods; in particular, other methods (for example, AQ11 (Michalski, 1983) and ID3 (Quinlan, 1986) and their descendents) attempt to succeed on more varied types of inconsistent data. The resulting learning algorithms thus have a qualitatively different flavor than the approach taken with incremental version-space merging: past work has taken an approach that forms classifiers that perform well but not perfectly on the data, viewing those instances not covered as anomalous and to be ignored; here every instance is viewed as providing useful information, and the final classifier must be consistent with *every* instance or one of its neighbors. By focusing on a more specialized source of information—data with *bounded* inconsistency—learning here is able to succeed even when all of the data are subject to inconsistency (Hirsh, 1990).

Of those efforts that do bear a closer similarity to this work, most similar is the work by Drastal, Meunuer, and Raatz (1989). Their approach—like most other noise-tolerant learning methods—is to overfit the inconsistent data as if the data are noise-free using a learning technique capable of forming multiple disjuncts. Some of these disjuncts only exist to cover anomalous instances, and thus after learning, they remove disjuncts that appear spurious. However, unlike other methods for pruning such noise-induced disjuncts, their method prunes those that only cover instances that can be perturbed to fit under one of the other disjuncts. Such an approach will be successful only if the anomalous positive data covered by a disjunct suffer from bounded inconsistency, since otherwise the necessary perturbations will not be possible.

One benefit of their technique is that the small disjuncts allow learning to focus only on the smaller collection of examples that such disjuncts cover (under the assumption that they identify the positive examples that suffer from bounded inconsistency), whereas here all instances are viewed as potentially anomalous. However, this is also a limitation of their approach, in that it assumes that all inconsistent data fall into disjuncts containing only noisy data, rather than being dispersed throughout the disjuncts; their approach furthermore only deals with inconsistent *positive* data. Nonetheless, their work suggests the possibility of adding to the version-space approach described here some method for identifying those examples that are more likely to be noisy, so that not all examples need be treated as potentially inconsistent.

8.3. Combining Empirical and Explanation-Based Learning

In addition to data with bounded inconsistency, this paper has described how background domain theories can be used when learning with incremental version-space merging. While there have been many approaches to integrating empirical and explanation-based learning described in the machine learning literature (Buchanan and Mitchell, 1978), (Anderson, 1983), (Mitchell, 1984), (Lebowitz, 1986), (Danyluk, 1987), (Pazzani, 1988), (Bergadano and Giordano, 1990), (Cohen, 1990), (Flann and Dietterich, 1990), (Pazzani and Kibler, 1992), the approach presented in Section 7 implements and extends one particular method proposed by Mitchell (1984). It nonetheless bears some similarity to other approaches that integrate the two learning techniques.

For example, Cohen's (1990) work on using EBG as an abstraction mechanism provides an approach to combining empirical and explanation-based learning that is very similar to this work, but avoids many of the description-language restrictions necessary in this work. His approach applies EBG to all positive data and generalizes each result further (in multiple ways) by operations on the example's proof tree (such as dropping selected subproofs so that their weakest preconditions do not appear in the final result). Cohen's system then selects a subset of all such generalized results that correctly classifies the training data, using a greedy set-cover algorithm. Since his technique never attempts to compare two descriptions or find the minimal generalization or maximal specialization of two descriptions, it does not require many of the description language restrictions necessary here. On the other hand it does not have the flexibility of considering the multiple candidate rules that would be maintained in a version space by the approach described here.

The IOE method of Flann and Dietterich (1990) also bears some resemblance to this work. In contrast, however, IOE generalizes across *explanations*, rather than across the results of EBG on those explanations. If a constant appears in all explanations, it will remain in IOE's generalization, even if EBG would have variablized the constant. IOE forms rules that are never more general that what EBG can create—in their view explanations are overgeneral, and IOE will find specializations of such explanations. The approach of Section 7 instead always finds descriptions that are more general than the results of EBG. However, extensions to their work can permit generalizing beyond what appears in the explanation so that IOE can generate results closer in flavor to those given here (Burgess, 1991).

9. Concluding Remarks

This paper has presented a generalization of version spaces that takes them beyond strict consistency with data. The basic idea is to view each piece of information (*e.g.*, an instance and its classification) as specifying some version space of relevant classifiers; learning takes place by intersecting these version spaces to get the version space that reflects all the information. The resulting learning algorithm, *incremental version-space merging*, removes the assumption that a version space is the set of classifiers strictly consistent with training data, and instead allows version spaces to contain arbitrary sets of classifiers, however generated, as long as they can be represented by boundary sets. This extends the range of applicability of version spaces by increasing the range of information that can be used in learning; in particular this paper has described how three very different types of information—ambiguous data, inconsistent data, and background domain theories—can be used by incremental version-space merging. To emulate the candidate-elimination algorithm the version space for each instance should be the set of definitions that classify the instance correctly, and intersecting all the resulting instance version spaces gives the same result as the candidate-elimination algorithm; extending this emulation to learning from ambiguous data merely requires forming instance version spaces consistent with any instance the given example may actually be. If data are inconsistent, instance version spaces are expanded to contain classifiers that don't necessarily classify the instance correctly but do classify at least one neighboring instance correctly. Finally, if background knowledge provides information that says that other instances must have the same classification as a given instance (identifiable by using explanation-based generalization), forming instance version spaces consistent with the set of equivalent instances means overly specific and overly general definitions can be removed from consideration. Most notable is that all three of these learning scenarios use the same, single implementation of incremental version-space merging (with only the code specific to instance version-space formation and description languages varying between problems).

While each of the applications of incremental version-space merging are described more or less in isolation, they can actually be characterized to some extent in a more uniform light by focusing the relative sizes of the resulting version spaces. The candidate-elimination algorithm, and its emulation here, is rather conservative, maintaining all classifiers consistent with data. However, the approach for learning from inconsistent data is even more conservative, and includes in version spaces even more classifiers than would ordinarily be maintained. On the other hand, the approach for using domain theories in learning can be characterized as doing the opposite, where outside knowledge provides a way to include in version spaces fewer classifiers than would ordinarily be included. At an intuitive level, inconsistent data means more definitions should be kept around since there is less certainty about the desired result; in contrast, a domain theory provides more certainty about the desired result, and thus fewer definitions should be maintained. Incremental version-space merging provides a tool capable of moving between these extremes.

The goal of this paper has *not* been to show that incremental version-space merging performs better on any particular individual learning task than competing methods. Rather, the goal has simply been to describe a generalization of version spaces, demonstrating its increased generality by showing that it is able to handle a wider range of information sources than the original candidate-elimination algorithm. In considering further how this work relates to the candidate-elimination algorithm, one can consider what it would take to have used the candidate-elimination algorithm to accomplish each of the learning tasks described in this paper. For each type of information used by incremental version-space merging one might imagine creating some different learning-task-specific modification of the candidate-elimination algorithm geared to that particular information. The main issue is that for incremental version-space merging the learning algorithm (namely, repeated version-space intersections) never changes. Incremental version-space merging separates information about what classifiers should be considered from the syntactic manipulations that change data structures. For some new learning task, some new modification of the candidate-elimination algorithm would be necessary, whereas here all that's necessary is a new specification of what the relevant classifiers are for given the learning task.

Even more importantly however, the generality of incremental version-space merging makes it possible to consider the integration of more than one type of information in a single learning task, by, for example, intersecting a version space generated from an ambiguous example with a second that reflects a background domain theory, and then intersecting the result with yet a third version space generated from a noisy example. It is not clear that such multi-source learning could be accomplished via any simple modification to the candidate-elimination algorithm. This paper has considered learning from each type of information in isolation; exploring the use of incremental version-space merging to achieve such multi-source learning is a question left for future work.

Acknowledgments

This paper summarizes and updates part of the author's dissertation (Hirsh, 1989), where more lengthy acknowledgments can be found. I am most grateful to my advisor, Bruce Buchanan, and my dissertation reading committee members Tom Mitchell and Paul Rosenbloom, without whom this work would not have been possible. Others who have had an impact on this work include William Cohen, Tom Dietterich, Oren Etzioni, Nick Flann, Rich Keller, David Aha, Jim Blythe, Tze-Pin Cheng, Scott Clearwater, Benjamin Grosof, David Haussler, Steve Norton, Stuart Russell, Jeff Schlimmer, Johanna Seibt, Derek Sleeman, Dave Smith, Devika Subramanian, John Sullivan, Paul Utgoff, Monte Zweben, and members of the GRAIL and Logic groups at Stanford and the MLRG group at CMU. William Cohen and Paul Rosenbloom furthermore provided many helpful comments on previous drafts of this paper. This work was partly supported by a Shell Foundation fellowship and NSF grant IRI-9209795.

Appendix A: Computational Complexity

This appendix analyzes the computational complexity of incremental version-space merging. First the computational complexity of processing a single instance is discussed and compared to that for the candidate-elimination algorithm; this is then followed by a discussion of the complexity of the overall incremental version-space merging method. Further details are provided elsewhere (Hirsh, 1990).

A.1. Complexity of Processing a Single Example

Each iteration of incremental version-space merging for a single example has two steps: version-space formation, and version-space intersection. This subsection discusses the complexity of each of these in turn, followed by a comparison of the resulting complexity analyses to those for a single step of the candidate-elimination algorithm.

A general analysis of the complexity of a single step is impossible—it depends both on the specific method for generating version spaces and on the particular description language being used. However, it is possible to do this analysis for specific approaches, and that is what will be done here. The particular description language considered here works for the three applications presented in this paper, namely conjunctions over a fixed set of k features. Features are furthermore assumed to be tree-structured (as in Section 2) or ranges of the form $a \leq x \leq b$ (as in Section 6.5).

In addition, the complexity of each step depends on the nature of how individual version spaces are formed. All three applications of this paper allow some form of ambiguity in data (*i.e.*, due to inconsistency or multiple explanations), and the analysis will be specifically for such version-space formation, with n referring to the effective number of instances whose version spaces are unioned.

A.1.1. Version-Space Formation

The first step of each iteration of incremental version-space merging is to form the version space of classifiers to consider given the current piece of information. Under the assumptions above, for positive instances the S set initially contains n elements (due to ambiguity/inconsistency). However, since no element of the S set should be more general than another it is necessary to do all $O(n^2)$ pairwise comparisons of the elements. Comparing two classifiers requires k feature comparisons. For tree-structured features, each feature comparison takes time proportional to the depth d of the tree-structured hierarchy for that feature. Therefore the time to compare two classifiers in this case takes $O(kd)$ time, and thus computing an S set for tree-structured features takes time $O(n^2kd)$. In contrast, when features are ranges of the form $a \leq x \leq b$, feature comparisons take constant time, and thus computing the S set takes $O(n^2k)$ time. The G set for positive data always contains the single universal concept, which is of size $O(k)$, so G-set complexity is $O(k)$.

For negative instances the S set contains the empty concept, so only the G set need be considered. There are initially nke elements in the unpruned G set, where e is the maximum number of ways to minimally specialize a feature to exclude a value.[9] (Each of the k features can be specialized e ways, and this is done for each of the n possibilities.) Pruning nonmaximal elements therefore requires $O((nke)^2)$ classifier comparisons. For tree-structured features e is at most $(w-1)d$, where w is the maximum branching factor for any node of any of the feature hierarchies, and thus computing the G takes time $O((nk(w-1)d)^2kd) = O(n^2k^3d^3w^2)$. In the case where features are ranges of the form $a \leq x \leq b$, with the range of x discretized to a fixed set of values (such as measuring values to the nearest millimeter), $e \leq 2$, and thus the time to compute a G set is $O(n^2k^3)$.

A.1.2. Version-Space Merging

The second step of incremental version-space merging is to intersect two version spaces. The complexity of version-space merging is again dependent on the particular concept description language, and again the analysis is done for conjunctive languages over k features. The algorithm given in Section 3 describes version-space merging somewhat abstractly, by describing how the new boundary sets depend functionally on the boundary sets of the version spaces being intersected. The basic algorithm given there simply computes for the new S set the most specific common generalizations of pairs from the two S sets that are covered by some element of each of the G sets but not by some other element of the new S set, and does a symmetrical process for the G sets. However, the actual algorithm is somewhat more subtle, and involves two refinements to this. Neither is difficult, but describing them requires referring to the two version spaces and their boundary sets: VS_1 and VS_2 will refer to the two version spaces being intersected, and S_1, G_1, S_2, and G_2 will refer to their respective boundary sets.

The first refinement simply checks whether one of the two G sets contains only the universal concept. If one does—without loss of generality assume it is G_1—the new G set can be computed more efficiently by testing each element of G_2 against each element of S_1. Only those elements of G_2 that are more general than some element of S_1 are retained to form the new G set. Symmetrically, if either S set contains the empty concept, the comparable process would be used to determine the new S set.

The second refinement handles cases where neither G set contains the universal concept. The algorithm as described in Section 3 computes the maximal specializations of elements of the two G sets. However, before doing this operation the merging algorithm can first prune elements of G_1 not above some element of S_2, and similarly prune elements of G_2 not above some element of S_1—if an element of G_1 is not above any element of S_2, there's no way any maximal specialization generated from that element of G_1 can be above an element of S_2. Doing this pruning of G_1 and G_2 can significantly reduce the number of maximal specializations that must be computed. The symmetrical process is done on elements of S_1 and S_2, comparing them to G_2 and G_1, respectively.

With the algorithm now more fully described, it is possible to carry out an analysis of the complexity of version-space merging. To compute the new S set, if S_1 contains the empty concept, computing the new S set requires $O(|S_2||G_1|)$ comparisons of elements of S_2 to G_1. For tree-structured features comparing two classifiers takes $O(kd)$ time, so the overall complexity is $O(|S_2||G_1|kd)$. For ranges comparisons take $O(k)$ time, so the complexity is $O(|S_2||G_1|k)$. By a symmetrical argument for G sets, these are the same complexities for the case where G_2 contains only the universal concept. Finally, if S_2 instead contains only the empty concept (or symmetrically G_1 contains only the universal concept) the complexities are $O(|S_1||G_2|kd)$ for tree-structured hierarchies and $O(|S_1||G_2|k)$ for ranges.

The more general case occurs if neither S_1 nor S_2 contains the empty concept (or symmetrically, neither G_1 nor G_2 contains the universal concept). The analysis will first consider S sets. The first step is to prune elements of S_1 not below some element of G_2, and similarly prune elements of S_2 not below some element of G_1. For tree-structured hierarchies this requires $O((|S_1||G_2| + |S_2||G_1|)kd)$ time and for ranges $O((|S_1||G_2| + |S_2||G_1|)k)$ time. Let S'_1, S'_2, G'_1, and G'_2 represent the resulting pruned boundary sets. The next step for merging requires computing the minimal generalizations of pairs from S'_1 and S'_2 that are above some element of each of G'_1 and G'_2. For both tree-structured features and ranges the minimal common generalization of two classifiers is unique. Therefore, if there are $|S'_1|$ and $|S'_2|$ elements in the two S sets, there are at most $|S_1'||S_2'|$ minimal generalizations. When features are tree-structured, computing the minimal generalization of two classifiers can be done in $O(kd)$ time. Computing the unpruned S set thus takes $O(|S_1'||S_2'|kd)$ time. Computing minimal elements takes an additional $O((|S_1'||S_2'|)^2kd)$, and removing elements not covered by some G-set element takes $O(|S_1'||S_2'|(|G_1'| + |G_2'|)kd)$. Thus the overall complexity is $O((|S_1||G_2| + |S_2||G_1|)kd + |S_1'||S_2'|kd + (|S_1'||S_2'|)^2kd + |S_1'||S_2'|(|G_1'| + |G_2'|)kd)$ which simplifies to $O((|S_1||G_2|+|S_2||G_1|)kd+(|S_1'||S_2'|)^2kd+|S_1'||S_2'|(|G_1'|+|G_2'|)kd)$ for tree-structured features. The G-set case is symmetric, and the overall complexity is $O((|S_1||G_2| + |S_2||G_1|)kd + (|G_1'||G_2'|)^2kd + |G_1'||G_2'|(|S_1'| + |S_2'|)kd)$ The overall complexity is bounded by whichever is greater, but this can be bounded by the sum of the two complexities, which is itself bounded by the slightly weaker bound $O((|S_1||G_2| + |S_2||G_1|)kd + (|S_1'||S_2'| + |G_1'||G_2'|)^2kd)$. If the very first pruning step of comparing S sets and G sets removes no elements (the worst-case scenario), this is in turn bounded by $O((|S_1||S_2| + |G_1||G_2|)^2kd)$.

When features are ranges of the form $a \leq x \leq b$, computing the minimal generalization of two classifiers takes $O(k)$ time. Computing the new S set therefore takes $O((|S_1||G_2|+|S_2||G_1|)k+|S_1'||S_2'|k+(|S_1'||S_2'|)^2k+|S_1'||S_2'|(|G_1'|+|G_2'|)k)$ which simplifies to $O((|S_1||G_2|+|S_2||G_1|)k+(|S_1'||S_2'|)^2k+|S_1'||S_2'|(|G_1'|+|G_2'|)k)$ and for the G set it is symmetrically $O((|S_1||G_2|+|S_2||G_1|)k+(|G_1'||G_2'|)^2k+|G_1'||G_2'|(|S_1'|+|S_2'|)k)$. This can similarly be bounded by $O((|S_1||G_2| + |S_2||G_1|)k + (|S_1'||S_2'| + |G_1'||G_2'|)^2k)$ or by the even weaker bound $O((|S_1||S_2| + |G_1||G_2|)^2k)$.

A.1.3. Comparison to the Candidate-Elimination Algorithm

The preceding complexity analyses can be compared to those of the candidate-elimination algorithm in cases of consistent, unambiguous data. The complexity of a single step of the candidate-elimination algorithm is presented elsewhere (Hirsh, 1992a), where for tree-structured features updating positive data requires $O(|G|kd)$ and negative data requires $O(|G|^2k^3d)$; for ranges the complexities are $O(|G|k)$ for positive data and $O(|G|^2k^3)$ for negative data.

Before comparing these results to those of incremental version-space merging, a few algebraic simplifications can be made to the complexity analysis of incremental version-space merging when used on consistent, unambiguous data. First, note that there is no ambiguity for the candidate-elimination algorithm, so $n = 1$. Second, S sets are always singleton for tree-structured features as well as ranges, so $|S_1| = |S_2| = 1$. Finally, since one of the two version spaces, say VS_2, refers to the version space from the previous iteration of incremental version-space merging, its G set, G_2, serves the role of $|G|$ in the candidate-elimination algorithm analysis and G can replace G_2; furthermore, it will always be the case that $|G| \geq 1$. Since the complexity of a single step of the candidate-elimination algorithm corresponds to the two steps of forming an instance version space and intersecting it with the previous version space, the complexity for a single step of incremental version-space merging is the sum of the complexities from the previous two subsections. These simplifications make the complexity of forming a version space for

Table A.1. Comparing Incremental Version-Space Merging (IVSM) and the Candidate-Elimination Algorithm (CEA).

		Complexity	
Language	Algorithm	Positive Example	Negative Example
Hierarchies	CEA	$O(\|G\|kd)$	$O(\|G\|^2k^3d)$
Hierarchies	IVSM	$O(\|G\|kd)$	$O(k^3d^3w^2 + \|G\|^2k^3d)$
Ranges	CEA	$O(\|G\|k)$	$O(\|G\|^2k^3)$
Ranges	IVSM	$O(\|G\|k)$	$O(\|G\|^2k^3)$

positive data $O(kd)$ for tree-structured features and $O(k)$ for ranges. For negative data they become $O(k^3d^3w^2)$ for tree-structured features and $O(k^3)$ for ranges.

The complexity results for version-space merging differ for positive and negative data. For positive data G_1 contains only the universal concept, so computing the new G set (comparing each element of G_1 to S_2) takes time $O(|G|kd)$ for tree-structured features and $O(|G|k)$ for ranges. Computing the new S set takes time $O(kd)$ for trees and $O(k)$ for ranges, but these are less than the G-set times, so can be ignored. Finally, adding the merging cost with the instance-formation cost for positive data yields a complexity analysis of $O(kd + |G|kd) = O(|G|kd)$ for tree-structured features and $O(k + |G|k) = O(|G|k)$ for ranges; these match the complexity results for the candidate-elimination algorithm given above.

For negative data (assuming G_2 is not the universal concept), $|G_1|$ is initially at most kdw for tree-structured features. However, $|G_1'|$ (the size of the pruned G set after comparing all elements of G_1 to S_2) has size at most k. Thus the complexity for merging is $O((|G| + kdw)kd + (1 + k|G|)^2kd) = O(k^2d^2w + |G|^2k^3d)$ (using $|G|$ as a bound for $|G_2'|$)—again, S-set formation takes strictly less time and can be ignored—and when adding in the cost for version-space formation yields a total complexity for one step of incremental version-space merging of $O(k^3d^3w^2 + k^2d^2w + |G|^2k^3d) = O(k^3d^3w^2 + |G|^2k^3d)$. The second term matches the result for the candidate-elimination algorithm above. The first term represents the additional overhead for computing the G set of the instance version space, many of whose elements are pruned when it is compared to the S set of the current version space—it depends on natural properties of the feature's generalization hierarchy (its depth and branching factor), as well as on the number of features.

The case for ranges is similar. First, both $|G_1|$ and $|G_1'|$ are at most $O(k)$, and thus computing the new G set has complexity $O((|G| + k)k + (1 + k|G|)^2k) = O(|G|^2k^3)$ This plus the complexity of instance version-space formation gives the complexity of one step of incremental version-space merging, since again S-set complexity can be ignored. This yields complexity $O(k^3 + |G|^2k^3) = O(|G|^2k^3)$ which matches the complexity for the candidate-elimination algorithm.

A summary of these results is given in Table A.1.

A.2. Complexity of Processing a Set of Examples

The preceding two subsections discussed the complexity of the individual steps taken during each iteration of incremental version-space merging and showed that each step runs in polynomial time. However, the real question to ask is about the complexity of the overall algorithm given a *set* of instances. This is a more difficult issue, as it depends on the nature of the classifier being learned, the concept description language, and the particular instances provided as data. In the worst case the process will be exponential in the number of instances, as has been pointed out by Haussler (1988) for the candidate-elimination algorithm, which is subsumed by this work. This subsection considers ways to avoid such worst-case complexity.

For example, if all the data are available at the start of learning, in some cases where exponential growth could occur it is possible to reorder the data so that exponential growth is avoided. One example where this is true (for consistent data and conjunctive languages over generalization hierarchies) is in cases where after processing all the given data the resulting boundary sets are singleton. In such cases the data can always be efficiently ordered to guarantee that the boundary sets will remain singleton throughout learning (Smith and Rosenbloom, 1990). (The ordering algorithm basically processes positive data first, then processes negative "near-misses" (Winston, 1975) that remove selected "don't cares.")[10] This result can even be generalized to some cases where average-case behavior yields exponential-sized G sets during the course of learning, but properly ordering the data keeps the G set singleton throughout learning. This general question of how best to order a set of data is an area of current work.

Even when boundary sets do not grow exponentially in size there are techniques for further improving the performance of incremental version-space merging. One such method is to skip data that will not change the version space. The key idea is that instances that are correctly classified by all members of the version space will have no effect. This observation allows two improvements to incremental version-space merging: whenever one instance version space is a superset of another the first instance can be removed (this can be done in a preprocessing stage if all the data are available at the start of learning); and whenever the version space for the current instance is a superset of the current version space it can be skipped. Note that the test for superset/subset is simple to do in boundary-set representation: version space VS_1 is a superset of VS_2 if, for each element of VS_2's S set, there is some element of VS_1's S set more specific than it, and similarly all elements of VS_2's G set have some element of VS_1's G set more general than it.

Finally, note that while these enhancements can greatly improve the run time of incremental version-space merging (as well as the candidate-elimination algorithm), the general issue of the computational complexity of processing a set of data still remains an open question.

Notes

1. A more precise definition of this algorithm is presented in an appendix, in Section A.1.2.

2. Although not strictly required by incremental version-space merging, this paper makes the assumption that every description language has a single "universal concept," which classifies everything as positive, and a single "empty concept" (represented by the symbol "$\emptyset$"), which classifies everything as negative. This is a reasonable assumption to make, since it is fairly simple to add two such terms to any description language.
3. Although Mitchell's (1978) original presentation of version spaces included an alternative approach to learning from inconsistent data with version spaces, it is computationally intractable and lacks the generality of incremental version-space merging. The relationship of this work to Mitchell's, as well as to similar work outside a version-space framework, is discussed further in Section 8.
4. Leave-one-out cross-validation involves learning on each subset of size 149 and testing on the single instance that was left out, and averaging the results of the 150 single-instance test runs. This gives an unbiased error estimate for the result of learning on all 150 examples.
5. These results are also notable in that learning here operates in a highly constrained language where classifiers are single conjunctive rules specifying some axis-parallel hyperrectangle.
6. EBG is a learning method that learns from a single example using a background domain theory. It first proves that an instance is classified correctly, and then forms the weakest preconditions on the proof. The weakest preconditions yield a definition of the class of instances that would have the same proof as the given example.
7. Constraints on this input are discussed further in Section 7.4.
8. Issues concerning this representation are discussed in Section 7.4.
9. If e is infinite (*e.g.*, when the number of values a feature may take is infinite), the use of version spaces is inappropriate, since boundary-set sizes must be finite.
10. Smith and Rosenbloom (1990) actually present a version-space learner that does not require all data to be present at the start of learning, but rather processes all positive data as they are received and saves negative data for later processing as they become near misses.

References

J. R. Anderson (1983). *The Architecture of Cognition*, Harvard University Press, Cambridge, MA.

F. Bergadano and A. Giordana (1983). Guiding induction with domain theories. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning: An Artificial Intelligence Approach, Volume III*, pages 474–492, Morgan Kaufmann, Los Altos, CA.

A. Borgida, R. J. Brachman, D. L. McGuinness, and L. Resnick (1989) CLASSIC: A structural data model for objects, In *Proceedings of SIGMOD-89*, Portland, Oregon.

B. G. Buchanan and T. M. Mitchell (1978). Model-directed learning of production rules, In D. A. Waterman and F. Hayes-Roth, editors, *Pattern-Directed Inference Systems*, pages 297–312. Academic Press, New York.

S. Burgess (1991) Hierarchical and range data in induction over explanation, Unpublished Master's essay, Computer Science Department, Rutgers University.

W. W. Cohen and H. Hirsh (1992). Learnability of description logics, In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, Pittsburgh, PA.

W. W. Cohen (1990) *Explanation-Based Generalization as an Abstraction Mechanism in Concept Learning*, PhD thesis, Rutgers University.

A. P. Danyluk (1987). The use of explanations for similarity-based learning, In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy.

T. G. Dietterich, B. London, K. Clarkson, and G. Dromey (1982). Learning and inductive inference, In P. Cohen and E. A. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Volume III*. William Kaufmann, Los Altos, CA.

G. Drastal, R. Meunier, and S. Raatz (1989). Error correction in constructive induction, In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 81–83, Ithaca, New York.

R. A. Fisher (1950). The use of multiple measurements in taxonomic problems, *Annual Eugenics*, 7:179–188, 1936, Also in *Contributions to Mathematical Statistics*, John Wiley & Sons, NY.

N. S. Flann and T. G. Dietterich (1990). A study of explanation-based methods for inductive learning, *Machine Learning*, 4(2).

C. A. Gunter, T.-H. Ngair, P. Panangaden, and D. Subramanian (1991). The common order-theoretic structure of version spaces and ATMS's (extended abstract), In *Proceedings of the National Conference on Artificial Intelligence*, pages 500–505, Anaheim, CA.

A. Gupta (1987). Explanation-based failure recovery, In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington.

D. Haussler (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework, *Artificial Intelligence*, 26(2):177–221.

H. Hirsh (1989). *Incremental Version-Space Merging: A General Framework for Concept Learning*, PhD thesis, Stanford University.

H. Hirsh (1990). *Incremental Version-Space Merging: A General Framework for Concept Learning*, Kluwer, Boston, MA.

H. Hirsh (1991). Theoretical underpinnings of version spaces, In *Proceedings of the Twelfth Joint International Conference on Artificial Intelligence*, pages 665–670, Sydney, Australia.

H. Hirsh (1992). The computational complexity of version spaces, Technical Report ML-TR-36, Department of Computer Science, Rutgers University.

H. Hirsh (1992). Polynomial-time learning with version spaces, In *Proceedings of the National Conference on Artificial Intelligence*, San Jose, CA.

P. Idestam-Almquist (1989). Demand networks: An alternative representation of version spaces, SYSLAB Report 75, Department of Computer and Systems Sciences, The Royal Institute of Technology and Stockholm University.

M. Lebowitz (1986). Integrated learning: Controlling explanation, *Cognitive Science*, 10(2).

R. S. Michalski and J. B. Larson (1978). Selection of most representative training examples and incremental generation of vl1 hypotheses: The underlying methodology and description of programs ESEL and AQ11, Report 867, University of Illinois.

R. S. Michalski (1983). A theory and methodology of inductive learning, In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–134. Morgan Kaufmann, Los Altos, CA.

S. N. Minton (1988). *Learning Effective Search Control Knowledge: An Explanation-Based Approach*, PhD thesis, Carnegie Mellon University.

T. M. Mitchell, P. E. Utgoff, and R. B. Banerji (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics, In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 163–190. Morgan Kaufmann, Los Altos, CA.

T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli (1986). Explanation-based generalization: A unifying view, *Machine Learning*, 1(1):47–80.

T. M. Mitchell (1978). *Version Spaces: An Approach to Concept Learning*, PhD thesis, Stanford University.

T. M. Mitchell (1982). Generalization as search, *Artificial Intelligence*, 18(2):203–226.

T. M. Mitchell (1984). Toward combining empirical and analytic methods for learning heuristics, In A. Elithorn and R. Banerji, editors, *Human and Artificial Intelligence*. Erlbaum.

D. J. Mostow and N. Bhatnagar (1987). Failsafe—A floor planner that uses ebg to learn from its failures, In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy.

M. Pazzani (1988). *Learning Causal Relationships: An Integration of Empirical and Explanation-Based Learning Methods*, PhD thesis, University of California, Los Angeles.

M. Pazzani and D. Kibler (1992). The utility of knowledge in inductive learning, T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli, *Machine Learning*, 9(1):57–94.

J. R. Quinlan (1986). The effect of noise on concept learning, In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach, Volume II*, pages 149–166. Morgan Kaufmann, Los Altos, CA.

R. L. Rivest (1987). Learning decision lists, *Machine Learning*, 2(3).

S. J. Russell and B. N. Grosof (1987). A declarative approach to bias in concept learning, In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington.

H. Simon and G. Lea (1974). Problem solving and rule induction, In H. Simon, editor, *Models of Thought*. Yale University Press.

B. D. Smith and P. S. Rosenbloom (1990). Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces, In *Proceedings of the National Conference on Artificial Intelligence*, pages 848–853, Boston, MA.

P. E. Utgoff (1986). *Machine Learning of Inductive Bias*, Kluwer, Boston, MA.

K. VanLehn and W. Ball (1987). A version space approach to learning context-free grammars, *Machine Learning*, 2(1):39–74.

S. M. Weiss and C. A. Kulikowski (1991). *Computer Systems That Learn*, Morgan Kaufmann, Los Altos, CA.

P. H. Winston (1975). Learning structural descriptions from examples, In P. H. Winston, editor, *The Psychology of Computer Vision*, chapter 5. McGraw Hill, New York.

Received February 22, 1993
Accepted October 31, 1993
Final Manuscript November 9, 1993