*Proc. Int'l Workshop on Mixed Language Explanations in Learning, Environments (XLANG), in conj. with AIED2005,
Amsterdam, July, 2005, Copyright 2005.*

Text Classification Rule Induction in the Presence of Domain-Specific Expression Forms [1]

Adam Carlson and Steven Tanimoto [2]
Dept. of Computer Science and Engineering
Univ. of Washington, Seattle, WA, 98195, USA.

Abstract. We describe a new method for learning text-classification rules from examples. The text consists of messages written by students in an online learning environment, and it may contain ungrammatical expressions as well as specialized expressions such as formulas. The method is based on the version-space machine learning technique. Experiments show that our method successfully generalizes over certain classes of embedded numerical expressions involving ranges of values in RGB triples that represent colors in an image processing system.

Keywords. text classification, learning, numerical expressions, domain-specific terms, rule induction

1. Introduction

When students interact with online learning systems, they generate events sequences that express patterns of activity. Learning environments such as the INFACT system at the University of Washington[5] capture these event sequences and record them in a database where they are available to a suite of tools for analyzing them.

Student activity data collected by systems such as INFACT falls into three categories: (1) textual messages posted by students in dialogs and in response to assignments, (2) sketches drawn to accompany textual messages, and (3) user-interface events that occur as students operate tools such as calculators and programming environments. In this paper, we discuss a pattern recognition problem related to data in category 1. The problem is to take textual messages together with two additional kinds of information (categorizations of the messages and textual selections from the messages) and to induce rules that can classify these and additional, unseen messages correctly into the categories to which they belong.

This paper describes a particular variation on this problem of inducing text classification rules. Rather than work only with natural-language text, our system permits special expressions of a domain-specific nature to be included within the text, and it treats these expressions in special ways. For example, our method can process expressions that

[1]Research supported in part by the US National Science Foundation under grant EIA-0121345.

[2]Correspondence to: Steven Tanimoto, Box 352350, Dept. of CSE, University of Washington, Seattle, WA 98195, USA. Tel.: +1 206 543 4848; Fax: +1 206 543 2969; E-mail: tanimoto@cs.washington.edu.

we call RGB color specifications. These expressions may involve the color names “Red”, “Green” and “Blue” as well as some numeric values that must all be between 0 and 255. They may also be represented by triples of numbers with no color names. These forms are different from natural language text and similar in some ways to mathematical expressions. Other examples of “domain-specific expressions” include program snippets, mathematical expressions and chemical formulae.

The text classifications rules we induce are used for two purposes. One is to construct “diagnoses” of misconceptions that teachers can inspect in order to monitor the progress of their students. The other is to automatically construct feedback that can be given to students to help them overcome obstacles and learn more effectively.

2. Rule Induction Method

Our rule learner uses a variation of the Version Space algorithm [4]. It also uses an extensible rule language architecture inspired by the Version Space Algebra [3,2]

2.1. Version Space Data Types

The rule learner functions via interactions among three primary types of objects: version spaces, examples and positions. Version spaces essentially represent the classification rules. Examples encapsulate the data from which version spaces are learned, and positions are an abstract way of representing the notion of a portion of an example. This generality is meant to allow for the eventual inclusion of sketch data as additional evidence in examples. Currently the only kind of `Example` that is fully supported is the `TextEvidence` class and the only kind of position is the `TextPosition` class.

In order to describe the architecture of the rule learner in more detail, we present the application programming interface for version spaces and their related constructs. Our system is implemented in the Java language.

2.2. VS interface

The `VS` interface is the fundamental interface of the rule learner. It specifies the basic functionality of a version space.[1]

`Classify` takes an `Example` as input and returns a `Boolean`. The return value is `Boolean.TRUE` if the example is classified as positive, `Boolean.FALSE` if the example is classified as negative and `null` if the classification cannot be determined.

`AddExample` takes an `Example` and returns a `Vector` of version spaces that incorporate that example. The returned version spaces represent a disjunction of the various ways the version space can be modified to handle the example.

`Generalize` takes a positive `Example` and returns a `Vector` of version spaces that incorporate that example. This is how `AddExample` handles positive examples.

[1]The actual interface is larger than this, but several of the methods are either for convenience or for features that are orthogonal to the approach described here.

`Specialize` takes a negative `Example` and returns a `Vector` of version spaces that incorporate that example. This is how `AddExample` handles negative examples.

`Subsumes` takes a `VS`. It returns true if and only if the set of examples this `VS` classifies as positive is a superset of those classified as positive by the `VS` passed in as an argument.

`GetMatchingPositions` takes an `Example` and returns a `Vector` of `VSMatch` objects. Each `VSMatch` returned encapsulates a `Position` within the example where the match was found and the `VS` it was accepted by.

`toRegex` returns a regular expression that is guaranteed to match a superset of all strings that will be classified as true by the VS. (Further filtering of strings matched by the regular expression is done via `GetMatchingPositions`.)[2]

`GetUniverse` returns the instance implementing the Universal VS for this type of VS.

`GetEmpty` returns the instance implementing the Empty VS for this type of VS.

2.3. Compound Version Spaces

A compound version space is a version space that is predominantly defined in terms of sub-version spaces. The prime example of this is the `Conjunction`. A conjunction classifies an example as true if and only if all its conjuncts classify the example as true. Other methods are similarly defined in terms of the conjuncts. The `CompoundVS` inherits from `VS`, and adds a few methods that allow for specification of sub-version space classes.

`MakeVS` takes a `Vector` of `VS`s and returns a `VS`. This is a factory-like method that builds an instance of the `CompoundVS` (or possibly one of its sub-`VS`s) given a set of `VS`s.

`registerVS` takes a `VS` class (either the class name or the actual class object) and registers it as a sub-`VS`.

`unregisterVS` takes a `VS` class (either the class name or the actual class object) and removes it from the list of sub-`VS`s.

2.4. How it works

Here's a description of how the `VS` and `CompoundVS` interfaces work together in practice, using the Conjunction, Term and NumericRange classes as examples.

The `Term` class implements `VS`. It's a simple version space that recognizes whitespace separated words. The generalization lattice for this version space, shown in figure 1 is quite simple. A string either contains the term or it doesn't.

2.5. Numeric Ranges

The `NumericRange` class also implements`VS`. It's a slightly more complex example of generalization. While the entire generalization lattice is handled by the same class, there is some flexibility in what data it will accept. The generalization lattice is shown in figure 2.

[2]The use of regular expressions is required by the database interface of the system in which the rule learner is embedded.

$$\begin{tikzcd}
\text{Universe} \\
\text{Term} \arrow[u] \\
\text{Empty} \arrow[u]
\end{tikzcd}$$

Figure 1. Generalization lattice for `term` version space

$$\begin{tikzcd}
\text{Universe} \\
\text{Numeric Range} \arrow[u] \\
\text{Single Number} \arrow[u] \\
\text{Empty} \arrow[u]
\end{tikzcd}$$

Figure 2. Generalization lattice for a numeric range.

Numeric ranges are a little trickier than terms. For example, as a regular expression, it produces a sequence of digit characters long enough to accept the high end of the range. In order to confirm that the regular expression has actually matched a number within the range, the `GetMatchingPositions` method must be called.

Both the `Term` and `NumericRange` version spaces are simple version spaces. Even though a numeric range can accept a variety of numbers, and requires post-processing of regular expression matches to confirm a positive classification, it's still handled by a single object in a single class. The `CompoundVS` interface allows for the creation of version spaces which are built out of other version spaces.

2.6. Conjunctive Version Spaces

The `Conjunction` is a compound version space that represents the conjunction of a number of other version spaces. The `Conjunction` class implements the `CompoundVS` interface. Therefore it supports a number of version space registration methods. A version space can be registered with the conjunction class (or, for that matter, a particular conjunction object.) This will allow the conjunction to include that version space as it processes examples.

For conjunctions, the primary way registered version spaces are used is in parsing new examples. When an example is seen, the conjunction calls the `GetMatchingExamples` operation of the Universal instance of all registered version spaces. Each of these return a list of the locations in the example where they have identified a match. The lists of matching locations from all the different sub-version spaces are merged, and any overlapping matches are used to create alternate conjunctions. For example, take the string "A 10 by 20 rectangle." The terms that are found in this string are, T(A), T(10), T(by), T(20) and T(rectangle). (We use the convention X(y) to denote a version space of type X recognizing content y.) In addition, the numeric ranges NR(10) and NR(20) are also found. However, the positions matching T(10) and NR(10) overlap, as do T(20) and NR(20). As a result four version spaces are created:

$$C(T(A) \wedge T(10) \wedge T(by) \wedge T(20) \wedge T(rectangle)),$$
$$C(T(A) \wedge NR(10) \wedge T(by) \wedge T(20) \wedge T(rectangle)),$$
$$C(T(A) \wedge T(10) \wedge T(by) \wedge NR(20) \wedge T(rectangle)),$$
$$C(T(A) \wedge NR(10) \wedge T(by) \wedge NR(20) \wedge T(rectangle)).$$

This example is somewhat contrived, as it would be easier to make terms not include numbers and only return the last conjunction in which “A”, “by” and “rectangle” are terms, and “10” and “20” are numeric ranges. We will see a more realistic example involving RGB color specifications later, however this example is simpler and easier to talk about, so we will continue to use it for the time being.

Now consider a new example, “the square is 15 by 15.” This example only shares one non-numeric term with the previous one, that being “by”. However, it does have numbers, and though they don’t match the ones in the original example, if they’re being interpreted as ranges, those ranges can be extended to include them. Thus each of the version spaces above is generalized below (in the same order):

$$T(by),$$
$$C(NR(10-15) \wedge T(by)),$$
$$C(T(by) \wedge NR(15-20)),$$
$$C(NR(10-15) \wedge T(by) \wedge NR(15-20)).$$

One can think of a version space as a set of constraints that must be satisfied in order to classify an example as positive. Simple version spaces test those constraints against the example directly. Compound version spaces use their constituent sub-version spaces to test each of their constraints, and then may apply additional constraints as well (such as requiring that the portions of matching text be non-overlapping.)

When generalization must occur to incorporate a new example, the conjunction first tests which of its constituents already matches the example. Those do not need further generalization. Constituents that don’t match are generalized against the example. If those generalizations do not fail (that is, if they don’t require generalizing all the way to the universe), then each generalization of the constituent is considered. Just as before when overlaps of different parses of an example caused multiple conjunctions to be created, multiple generalizations are handled by taking the cross product and then making sure no overlaps occur.

Similarly, classification of compound version spaces is handled by classifying all constituents and then checking any additional constraints. Note that the implementation of this strategy might be different; for example, the conjunction could short-cut the classification process if any of its constituents returned false.

2.7. Ordered Conjunctions

Another example of a compound version space is the ordered conjunction. This is a conjunction that also places an ordering constraint on its constituents. It operates in much the same way as a conjunction, but in addition to imposing the constraint that substrings matching various constituents not overlap, it also requires that they be in a specified order. Similarly, a `Sequence` is like an ordered conjunction, but also restricts the number of items that can intercede between any two of its constituent parts. I.e. it restricts consecutive components to be separated by some fixed number of elements.

2.8. RGB Recognizer

The idea of version spaces farming out some of their work to other version spaces is used in the handling of RGB color specifications. Students use a number of different ways to describe RGB colors. They might type “Red: 100, Blue: 50, Green: 20” or they might just use “(100, 20, 50).” In order to handle these different representations, the RGB color recognizer uses a number of sub-version spaces to do a lot of its work. First, there is an intermediate version space type called a color specification. This recognizes a single color out of an RGB triple specified in the $Colorname : number$ format. The RGB recognizer looks for sequences of up to three color specifications, in which each of the colors are “Red”, “Green” or “Blue”, and no color is repeated. The color specifications themselves are expressed in terms of a fixed string, followed by a numeric range. Finally, because students sometimes just use numeric triples, the RGB color recognizer can also look for a sequence of three numeric range terms. A schematic of the way the RGB recognizer is constructed is shown in figures 3 and 4.

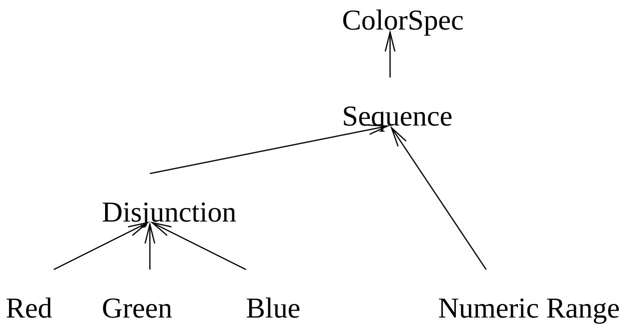

Figure 3. The Color Specification recognizer.

3. Experiments and results

To test this system, we collected data from INFACT-FORUM consisting of student responses to an additive color-mixing assignment. Each example was manually classified as positive or negative for the color requested. Using a leave-one-out strategy of defining training and test data, classification rules were induced and used to classify the remaining example. Two kinds of data were used: pruned and unpruned. The pruned data consisted

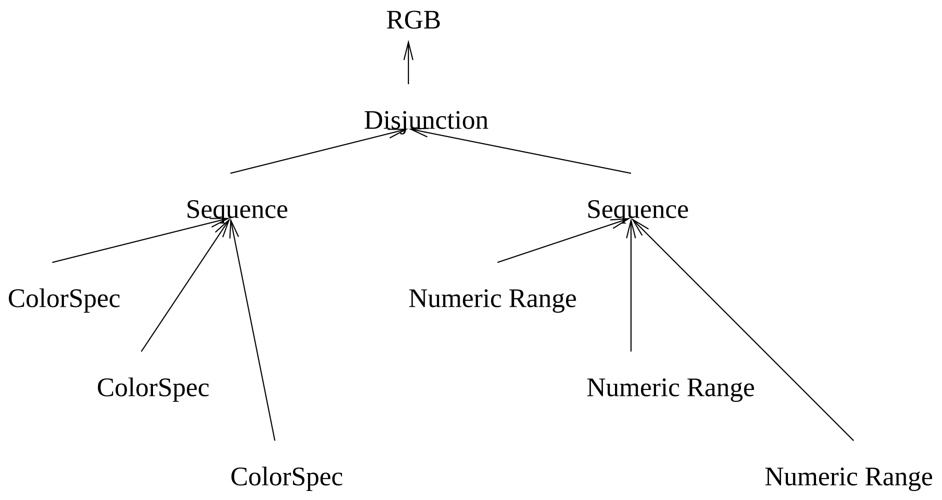

Figure 4. The RGB recognizer.

only of examples including RGB terms in the two formats handled by the RGB recognizer. The unpruned examples also included a number of expressions in other patterns.

With the pruned data, the use of the domain-specific RGB component tended to improve precision over the use of straight text terms only; RGB alone gave a 0.50 total precision value compared with 0.35 for term only. Combining the two gave an even better result: 0.66. Recall remained relatively unchanged. With the unfiltered data, the precision was fairly low, due to the large number of false positives obtained due to overgeneralization of the rules on the basis of misinterpreted RGB values.

4. Additional Information

Additional details on our methods and experiments can be found in the first author's Ph.D. dissertation[1].

References

[1] Adam Carlson. *Making the Implicit Explicit: Tools for Human Communication. Ph.D. thesis,* University of Washington, 2005.

[2] Tessa Lau. *Programming by Demonstration: A Machine Learning Approach*. PhD thesis, University of Washington, 2001.

[3] Tessa Lau, Steven Wolfman, Pedro Domingos, and Daniel Weld. Programming by demonstration using version space algebra. *Machine Learning*, 2003.

[4] Tom Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[5] Steven Tanimoto, Adam Carlson, Justin Husted, Earl Hunt, Josef Larsson, David Madigan, and Jim Minstrell. Text forum features for small group discussions with facet-based pedagogy. In *Proceedings of CSCL'02, Boulder, CO.*, 2002.

Pruned data sets

Combined	True+	True-	False-	False+	Prec.	Recall	F-score
Yellow	5	9	3	11	0.31	0.63	0.42
Purple	0	26	2	0	1.00	0.00	0.00
Pink	16	12	0	0	1.00	1.00	1.00
Gray	0	26	2	0	1.00	0.00	0.00
Totals	21	73	7	11	0.66	0.75	0.70

RGB Only							
Yellow	6	9	2	11	0.35	0.75	0.48
Purple	0	26	2	0	1.00	0.00	0.00
Pink	16	1	0	11	0.59	1.00	0.74
Gray	0	26	2	0	1.00	0.00	0.00
Totals	22	62	6	22	0.50	0.79	0.61

Term Only							
Yellow	6	2	2	18	0.25	0.75	0.38
Purple	0	3	2	23	0.00	0.00	0.00
Pink	16	12	0	0	1.00	1.00	1.00
Gray	0	26	2	0	1.00	0.00	0.00
Totals	22	43	6	41	0.35	0.79	0.48

Unpruned:
RBG only

Color	True+	True-	False-	False+	Prec.	Recall	F-score
Yellow	22	0	0	43	0.34	1.00	0.51
Purple	12	12	3	38	0.24	0.80	0.37
Pink	19	12	3	31	0.38	0.86	0.53
Gray	6	0	0	59	0.09	1.00	0.17
Totals	14.8	6	1.5	42.75	0.26	0.91	0.40

Figure 5. Experimental results.