SEMANTIC FEATURE CONSTRUCTION

by

Mark E. Fenner

M.S., University of Pittsburgh, 2002

M.A., University of Pittsburgh, 2002

B.S., Allegheny College, 1999

Submitted to the Graduate Faculty of

the Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2007

UNIVERSITY OF PITTSBURGH

COMPUTER SCIENCE DEPARTMENT

This dissertation was presented

by

Mark E. Fenner

It was defended on

June 14th 2007

and approved by

SEMANTIC FEATURE CONSTRUCTION

Mark E. Fenner, PhD

University of Pittsburgh, 2007

An effective set of features is integral to the success of machine learning algorithms. Semantic feature construction is the knowledge-driven manipulation of the propositional descriptor space of a set of examples for use in a learning algorithm. Two important sources of semantics for feature construction are the semantic type (and associated semantic properties) and the semantic class of features. These semantics can be captured in a knowledge base and utilized to constrain search through the space of constructed features. This dissertation presents a system that captures semantic feature construction knowledge and implements a search algorithm that respects that knowledge. Results are presented for different combinations of features generated from different successor functions used in search. These results are compiled over many learning problems and several learning algorithms. Other results are also presented for different levels of detail in semantic knowledge. Generally, semantics are an effective guide in the space of constructed features.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

It might be argued that this procedure of having the program select terms for the evaluation polynomial from a supplied list is much too simple and that the program should generate the terms for itself. Unfortunately, no satisfactory scheme for doing this has yet been devised. – Arthur Samuel, *Machine Learning Using the Game of Checkers* (Samuel, 1963).

Indeed, [Kepler] assumed that the Sun was a centre of force which governed the motions of the planets. If Kepler had tried to relate Mars to the Earth rather than to the Sun, he would never have found the elliptic orbit. – Donald Gillies, *Artificial Intelligence and the Scientific Method* (Gillies, 1996).

Machine learning systems require appropriate descriptions of the data they process. This dissertation describes semantic feature construction for machine learning and a system for implementing semantic feature construction: ISAK[1].

Broadly, feature construction is the process of taking a given description of an example and creating a new description of that example. Semantic feature construction performs feature construction based on symbolic knowledge of the learning examples and the world they occupy. Specifically, feature construction refers to the creation of new attribute-value descriptors (i.e., features) for examples which are currently described by other features. In semantic feature construction, an algorithm takes – as minimal input – the currently available features describing examples and symbolic knowledge of the world the examples occupy; the semantic feature construction algorithm produces new features and symbolic knowledge describing the new features.

Two important sources of knowledge about features are (1) semantics of individual features and (2) semantics about relationships among features. Individual features gain semantic meaning (a *semantic type* and *semantic properties*) from the quality of an object or process they measure, from the object itself, and from the relationship of the measured quality to other qualities[2]. Sets of

[1]ISAK stands for Integrating Search and Knowledge (for feature construction).

[2]A single learning example is a set of measurements on one or more objects and processes in the world under consideration. In many cases, the measurements are all taken over one object (e.g., a patient in a medical dataset).

features may be related by measuring different qualities of the same object, the same quality of different objects, or more generic relationships that can be captured by a simple notion of similarity, a *semantic class*. Each piece of knowledge used in this dissertation is represented explicitly and symbolically. The specificity of knowledge used by ISAK ranges from problem-specific knowledge to domain-specific knowledge to multi-problem, multi-domain knowledge such as that gathered in the Standard Upper Merged Ontology (SUMO) (Niles and Pease, 2001). The semantic knowledge is used as heuristic constraints while ISAK searches through the space of constructable features.

Search provides a useful, general framework with which to solve many problems. In feature construction, the problem of finding good features can be defined as a search through the space of constructable features. However, the combinatorics of the domain (i.e., the exponential growth of the search space) make search without heuristics impractical. The search framework also allows fine-grained control over the choices made in selecting constructed features. The control allows users to choose the bias that they find most useful for feature construction in their given learning problem.

The general hypothesis guiding this work is that explicit, symbolic knowledge about features can be used to construct features that improve generalizations produced by learning algorithms, as compared to generalizations produced by unmodified, base feature sets.

1.1 FEATURE CONSTRUCTION, MACHINE LEARNING, AND EMPIRICAL SCIENCE

Machine learning is any automated process by which a computer program improves its performance on a given task as it gains experience or knowledge (Mitchell, 1997). The learning task I deal with is called *classification learning*. Classification learning algorithms take as input:

1. a set of examples called a training dataset,
2. a uniform list of attributes and, for each example, the respective values for each attribute, and
3. a special target attribute and its value for each example.

However, a dataset may contain measurements on distinct objects that are components of each learning example (e.g., see Section 7.3.1).

These algorithms produce, as output, a *classifier* that is a mapping from examples to target values. The goal of the classifier is to achieve good performance on unseen examples. Building these classifiers has been called *pure inductive inference* (Russell and Norvig, 1995) and *learning by example* (Carbonell et al., 1983). Algorithmically performed induction is a special case of scientific induction.

The empirical sciences are defined by their use of empirical data from observations and experiments. The dependence on observation is the common thread in disciplines ranging from the physical sciences, such as biology and physics, to the social sciences, like economics and psychology. Collecting observations does not, in itself, satisfy the demands of a scientist. To utilize collections of data, various forms of analysis are available to the scientist. Three of the most common types of analysis are: (1) testing hypotheses, (2) searching for meaningful, interesting, or predictive patterns, and (3) creating parsimonious models of phenomena. The first of these roles is clearly seen in the testing of significance of statistical hypotheses and in the development of scientific (i.e., physical, psychological, or sociological) laws. The second and third analyses are common paradigms in multiple regression statistics and modeling (Dobson, 2002; Hastie et al., 2001), pattern recognition (Ripley, 1996), data mining (Hand et al., 2000; Fayyad et al., 1996), and machine learning (Mitchell, 1997; Hastie et al., 2001).

Philosophers of science debate the exact process that describes how science operates. However, a simplified view of the scientific process (Klemke, 1980) includes:

1. observing and recording phenomena as data,
2. classifying the data according to subject,
3. inducing general statements from observations,
4. deducing further statements from the general statements,
5. verifying the additional, deduced statements, and
6. creating theories that account for as many of the induced and deduced statements as possible.

Within these steps, a distinction is made between the process of *discovering* theories and statements and the process of *verifying* those theories and statements. Some philosophers (Popper, 1965) have argued that the verification process is subject to formal logic, but the discovery process cannot be effectively studied. Others have argued that the discovery process itself can be subjected to scrutiny, dealt with logically, and even automated (Buchanan, 1982; Langley et al., 1987).

Automated or not, the objects, processes, and theories discussed in the scientific process must be *represented.* Following Langley et al. (1987), we can define a representation as a scheme for holding information in storage and a means for operating on that representation to access, modify, and process the information it contains. New representations are created by finding (1) new tests for recognizing important features in a domain, (2) new inferences associated with these tests, and (3) new relationships between each representation and reality.

Even before choices of representation appear in scientific work, the question of what to study (i.e., *what* to represent) must be answered (Darden, 1991). For example, different levels of detail in measured features may be obtained depending on the questions that are asked by a scientist. General observations – such as the similarity of an offspring to a parent – do not support enough detail to drive any substantial generalization. Compressing the many possible similarities between offspring and parent into a simple statement, *"Yes, the child and parent are similar",* abstracts away too much useful information. However, observation of more specific characteristics, like the height and eye color of the parents and adult children, leads to better hypotheses like "*Eye color is a heritable trait"*. These more specific and more descriptive terms can be related to one another and used in generalizations.

The scientific process relates to semantic feature construction in at least two ways. First, as Darden (1991) states, "New discoveries beget new terms." The statement means that when we find out something new, we frequently attach new terminology to the discovery. If a discovery creates a dichotomy between two previously unified concepts, each of the new classes may receive a new name. For example[3], early humanity may have considered only one type of plant, *GenericPlants*. Upon receiving a nasty itch from certain plants, a new category was created for these itchy plants called *PoisonousPlants*; the remainder were considered to be *SafePlants*. If a unification[4] is found between formerly disparate concepts, again a new term may be created. For example, Japanese and American cars are normally grouped differently. However, in the modern world, some Japanese cars (e.g., Toyotas) are made in Kentucky. So, both the Toyotas and many American cars are in

[3]From Darden.

[4]*Unification* is used in the sense of the computation of a least upper bound with respect to a concept hierarchy (an instance of a join-semilattice).

the category of *CarsMadeByAmericans,* or even more accurately, *CarsMadeInAmerica.* In a more scientific setting, the unification of magnetic and electric phenomena into the theory of electromagnetism demonstrates this joining process. New terms capture new theoretical relationships that arise from a discovery. In this interpretation, feature construction is the creation of new terms (i.e., attributes) and values for those terms with computable functions[5].

The second relationship between semantic feature construction and science is the methods that science uses to describe observations by quality and quantity. A particular scientific instrument used to make a measurement comes coupled with a natural representation for that instrument (Langley et al., 1987; Kuhn, 1996). Additionally, the instrumentation defines the semantics of the observation. For example, measuring sticks, thermometers, and barometers each take a specific physical process and quantify that process as a numeric measure on a scale. In the case of a metric measuring stick, the scale is tied to the speed of light in a vacuum; for a barometer the scale is tied to millimeters of mercury in a tube; for a thermometer, the scale is defined by a standardized volume/pressure/temperature relationship. For the last of these, a particular proportionality (e.g., placement of freezing and boiling on the scale) and a particular zero point (e.g., absolute zero or the freezing point of water) are sufficient to define the scale. Though both distance and temperature have been measured in many different ways through the ages, these scales can be converted from one to another. The semantics and scale of measurements are both sources of knowledge for combining and limiting combinations of features.

1.2 AN EXAMPLE OF FEATURE CONSTRUCTION

To make the learning and construction scenario concrete, consider the following example derived from Quinlan (1995) and shown in Table 1. Each example (i.e., a line in the table) is a group of weather measurements for a particular day. Each day has several attributes associated with it: *Precip*, *Temp*, etc. The *Tennis* column, which is the target class for this problem, indicates whether

[5]Unifying two symbolic concepts can be done with logical *or* – if either of two concepts is true, it is true of the new concept. Dividing, or specializing, a concept can be done by conjunction – if the former concept is true and some additional concept is true, then the new concept holds. In a hierarchical structure of concepts, a new term may necessitate rearranging the hierarchy with respect to the new attributes.

Table 1: Base Tennis Weather Dataset. The dataset shows attributes that are related to the target concept of whether or not tennis is played on the day in question. The first 13 examples of 100 are shown here.

Precip	Temp	Wind	Humid	Tennis
d	d	d	d	d
				class
Cloudy	Mod	Calm	Humid	True
Sunny	Cool	Calm	Humid	True
Sunny	Cool	Breezy	Humid	True
Sunny	Mod	Calm	Mod	True
Rainy	Warm	Windy	Arid	False
Rainy	Mod	Breezy	Mod	False
Cloudy	Hot	Windy	Arid	True
Sunny	Cool	Breezy	Arid	True
Rainy	Warm	Breezy	Mod	False
PartCl	Warm	Breezy	Humid	False
Sunny	Cold	Breezy	Arid	False

tennis was played or not. A classifier for this learning problem predicts whether or not tennis would be played on the basis of the values for *Precip*, *Temp*, *Wind,* and *Humid* on a given day.

The redescription of meteorological information is a common practice. Frequently, weather forecasters give the actual air temperature followed by a temperature adjusted for humidity and wind chill. So, if we have a dataset that has only the base features of temperature, humidity, and wind speed, we can create a new feature that represents the apparent temperature.

1.2.1 Automatic Construction of Meteorological Features

ISAK can create features useful to the tennis classification problem. I used a synthetic data generator[6] to create a weather dataset of 100 examples. Some of the values are shown in Table 1. The complete target concept specification is shown in Appendix B. In addition to the raw dataset which would be used by a typical learning algorithm, ISAK takes as input a knowledge base (KB) file describing the features, operations, and search parameters for feature construction. For the weather problem, the feature constructor functions include: (1) Boolean *and, or,* and *not,* (2) creation and

[6]The synthetic data generator is described in Section 7.1.2.

extension of bags[7], (3) functions which test symbol equality, $Temp = Hot$, and (4) functions that count symbolic and Boolean values in bags. ISAK returns four constructed features (Figure 1). The results of learning with these new features is shown in Table 2.

The comparison between Base and New Features is the difference between learning with features constructed by ISAK and learning with a standard, off-the-shelf learning system. ISAK performs feature construction at a cost of time and memory. The dataset that results from ISAK is then passed to the same, off-the-shelf learner. The difference in the scenarios is the work done by ISAK to create new features. The improvement in 10-fold cross-validation accuracy may be attributed to the modified dataset.

1.2.2 Manual Construction of Meteorological Features

Compare ISAK's created features with the constructions performed by hand (or at least on an *ad hoc* basis) by meteorologists. Here are some example formulas used by meteorologists in making perceived temperature conversions[8]:

$$\begin{aligned} WindChill &= 35.74 + .6215 * Temp - 35.75 * WindVel^{.16} \\ &\quad +.4275 * Temp * WindVel^{.16} \end{aligned}$$

$$\begin{aligned} HeatIndex &= 42.379 + 2.049 * Temp + 10.143 * RelHumid \\ &\quad -.2248 * Temp * RelHumid - .006838 * Temp^2 \\ &\quad -.05482 * RelHumid^2 + .00123 * Temp^2 * RelHumid \\ &\quad +.000853 * Temp * RelHumid^2 \\ &\quad -.00000199 * Temp^2 * RelHumid^2 \end{aligned}$$

In both cases the temperature is given in Fahrenheit. *RelHumid* refers to the relative humidity.

[7] Also known as multi-sets.

[8] Taken from http://www.psend.com/users/stormchaser/generalwx.html. ISAK does not at present have a parameter fitting routine, so these types of regression equations would not be in the search space for constructed features. Adding regression models to the search space would be relatively easy, but controlling the added complexity would not. Note, the exponents in the *WindChill* equation are .16, not 16.

```
ftr_24180 (0.2239) is:
and(and(notEqRainy(Precip), notEqCloudy(Precip)),
    or(eqMod(Temp), notEqBreezy(Wind)))
ftr_11824 (0.2239) is:
or(and(notEqSunny(Precip), notEqPartCl(Precip)),
   and(notEqMod(Temp), eqBreezy(Wind)))
ftr_125980 (0.4327) is:
and(notEqRainy(Precip),
    or(and(notEqRainy(Precip), notEqBreezy(Wind)),
       or(eqMod(Temp), eqCool(Temp))))
ftr_124019 (0.3282) is:
and(or(eqRainy(Precip), eqBreezy(Wind)),
    or(eqCold(Temp),
       or(notEqSunny(Precip), eqHot(Temp))))
```

Figure 1: Four Constructed Features for the Weather Problem. Of the five base features shown in Table 1, four features are components of the constructions shown here. ISAK did not find *Humid* to be of value in any constructions. The values in parentheses, next to the synthetic feature names (e.g., ftr_124019), are the information gains on the constructed feature over the original dataset. The utility of these constructed features in comparison with the base features is shown in Table 2.

Table 2: Improvement in Weather Classification given Constructed Features. The cells are 10-fold cross-validation accuracies for the weather data with six learners and three feature sets. Details of the learning methods are discussed in Section 7.1.3. *Base Features* refers to the original, unmodified set of features. *New Features* refers to the use of only the four newly created features shown in Figure 1. None of the four new features make use of *Humid* in their definition. Thus, the information from *Humid* is lost in *New Features*. The result of adding *Humid* to the new feature set is shown in the third column of accuracies. The maximum standard error for any cell is .04.

Learner	Base Features	New Features	New Features and Humidity
Majority Learner	0.60	0.60	0.60
3-Nearest Neighbor	0.71	0.81	0.83
10-Nearest Neighbor	0.73	0.84	0.84
Naive Bayes	0.77	0.85	0.85
Tree Classifier	0.71	0.82	0.84
Support Vector Machine	0.74	0.83	0.87

Meteorological formulas are not solely arithmetic. They can be created from any well-defined procedure. For example, consider the comfort index[9] depicted in Table 3. The comfort index accounts for human body reaction to various environmental characteristics that affect our perception of heat. These characteristics include temperature, humidity, and wind chill. Of course, these are heuristic guides and they encode the general reaction of people to different weather conditions.

The created features can be incorporated into the tennis playing problem. Specifically, we can add a new feature *Comfortable* that is computed from the other features and will be helpful in predicting whether a day is suitable for tennis. Table 4 shows the Tennis problem with the newly created feature. The newly created attribute *Comfortable* has a nice correlation with the target concept of *Tennis*.

1.2.3 Why Create Features for Weather Problems?

The adjusted temperature is important because humans do not usually care what the temperature value is *per se*; they care about how they feel when they are outside. The perception of temperature

[9]From http://www.bartsgroup.net/weather/.

Table 3: Comfort Index Calculations. *WC* and *HI* refer to wind chill and heat index; they are defined by equations in the text.

Description	Variables
Extreme Cold	$WC < 0$
Uncomfortably Cold	$0 < WC < 30$
Cool	$30 < WC < 60$
Comfortable	$60 < Temp < 80$
Warm	$80 < Temp < 90$
Uncomfortably Hot	$Temp > 90$ and $HI < 100$
Extremely Hot	$HI > 100$

Table 4: Modified Tennis Data. *Comfortable* is an intuitively constructed variable from the initial variables of a tennis type problem.

Day	Outlook	Temperature	Humidity	Wind	Comfortable	Tennis
D1	Sunny	57	95	5	Yes	Yes
D2	Rain	63	80	0	No	No
D3	Overcast	75	40	10	Yes	Yes
D4	Sunny	92	99	30	No	No
D5	Sunny	92	20	15	Yes	No

is related to a number of raw meteorological values: temperature, wind speed, and humidity. So, it is interesting to ask what mathematical relationship captures the perceived relationship among these values.

Further, care must be taken to consider additional unrecorded conditions related to the tennis games. A missing variable that determines whether the games were played indoors or outdoors would lead to a very different model of tennis playing. Presumably, weather would be an important part of determining the outdoor games, while the indoor games would be less dependent on the weather. As it stands, the tennis playing problem is very close to answering the question, “Is it a nice day outside?”, at least for avid tennis players. The cases where the answers differ (no to *Tennis*, and yes to *NiceDay*) reflect that the given variables may not adequately model other concerns, such as work and family obligations, in making time to go play tennis.

Why is the newly created variable given in terms of the attributes it is constructed from – that is, its base attributes? In the case of heat index and wind chill, the constructed feature is described as a temperature. Temperature is the measurement best correlated with personal warmth. So, even though particular values of temperature may be perceived differently, the Celsius and Fahrenheit scales are well-known and widespread. The common scales provide the best means for communicating temperature information. Manipulating a raw temperature on the basis of other relevant factors – wind chill and humidity – produces a more accurate result, but it must be communicated effectively. Creating a new scale for the new temperature would be tantamount to telling most Americans the daily temperature in Celsius. The scale would be correct, but it would not carry much semantic content and would be less intelligible than a result in Fahrenheit.

Returning to tennis, predicting whether a day is suitable for tennis will be better served by considering the perceived temperature instead of the actual temperature. The modified tennis example takes this process one step further and uses knowledge of adjusted temperature and converts this to a simple Boolean value – comfortable or not. ISAK allows the user to specify the terms of a new feature as a part of the feature constructor function used to create new features from old features (see Section 6.2.5).

1.3 UTILITY OF FEATURE CONSTRUCTION IN LEARNING

1.3.1 Inductive Bias of Learners

A significant difference in algorithmic learning methods is their *inductive bias*. The inductive bias of a learner is the set of commitments that the learner makes to generalize from the given training data to new, previously unseen cases. An important component of inductive bias is the representation of the space of hypotheses a learner considers. For example, *decision trees* consider a hierarchy of splits over the values of features present in the data (Breiman et al., 1984; Quinlan, 1995). An example of a decision tree is shown in Figure 2. Such a tree is applied by considering an attribute at each node, starting from the root, and following the value for that attribute down the appropriate branch to a new attribute or a leaf. If the new node is an attribute, the process is repeated. If the new node is a leaf, the value of that leaf is the classification for that example. The end result of the classification tree is a partition of the examples into different regions that share a common target value or target function.

In contrast, *support vector machines* (SVMs) use a set of linear boundaries[10] called *support vectors* to form a separation among examples with different values (Burges, 1998). These boundaries are conveniently expressed as mathematical equations and can be visualized as graphs as in Figure 3. Again, the end result is a partition of the feature space into regions of varying classes. Other learning methods use different logical or mathematical representations for their hypotheses.

Many standard learning programs will perform poorly because the training examples' given representation is a bad match for the inductive bias of the algorithm. Rendell and Seshu (1990) term such situations as *hard problems*. These situations occur when the concept to be learned can be expressed in the given feature space, but the regions making up the concept are dispersed throughout the instance space. Hence, the measurements used to model phenomena are tremendously important. Using features in a suboptimal representation can lead to problems in the analysis of the data. These problems include (1) developing models that are more complicated than necessary (Setiono and Liu, 1998; Blum and Langley, 1997) which in turn can cause problems

[10]With a kernel SVM, these boundaries are linear in *some* feature space, not necessarily the original feature space. Also, in the SVM literature, the original feature space is referred to as the *input space* and the modified feature space is referred to as the *feature space.*

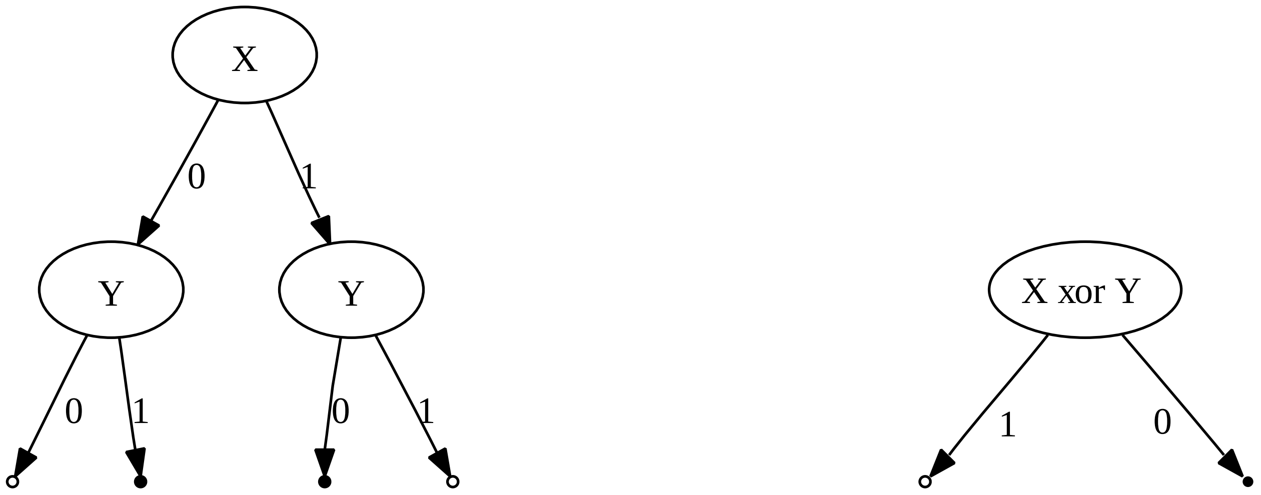


Figure 2: A Decision Tree for *xor* and a Reduced Tree. Both trees have decision rules equivalent to the *xor* function. The left tree requires two steps and tests on each feature, *X* and *Y*, to complete its evaluation. The right tree makes use of a constructed feature to make the decision in a single step. Note, the tree on the right requires preprocessing to create a feature $Z = X\, xor\, Y$.

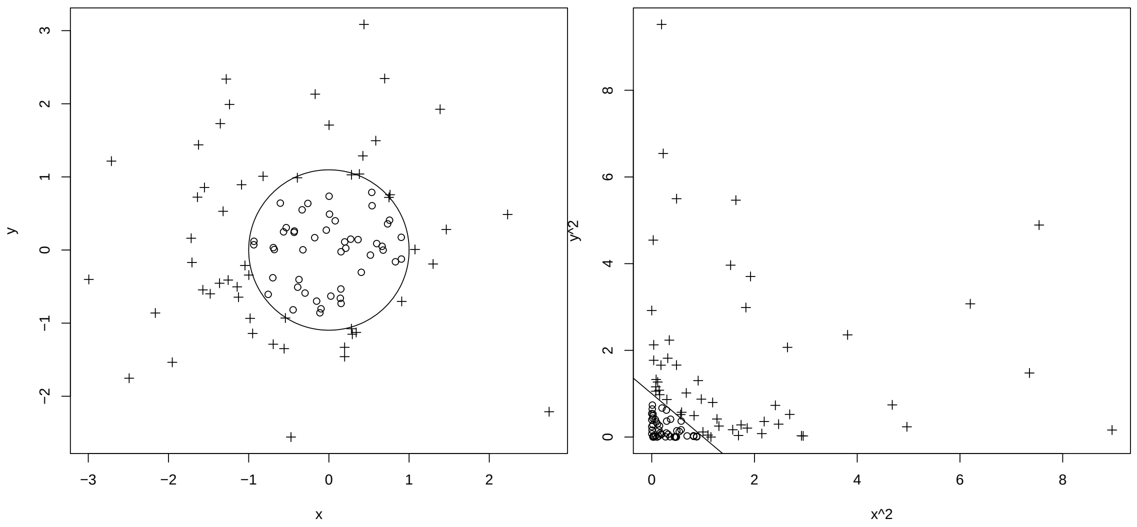


Figure 3: SVM Feature Space Modification. The left pane shows an unmodified dataset created from a non-linear concept, $\sqrt{x^2 + y^2} > 1$. The right pane is two dimensions of the three dimensional projection of the original data with a second-order polynomial kernel. The missing dimension is $\sqrt{2}xy$.

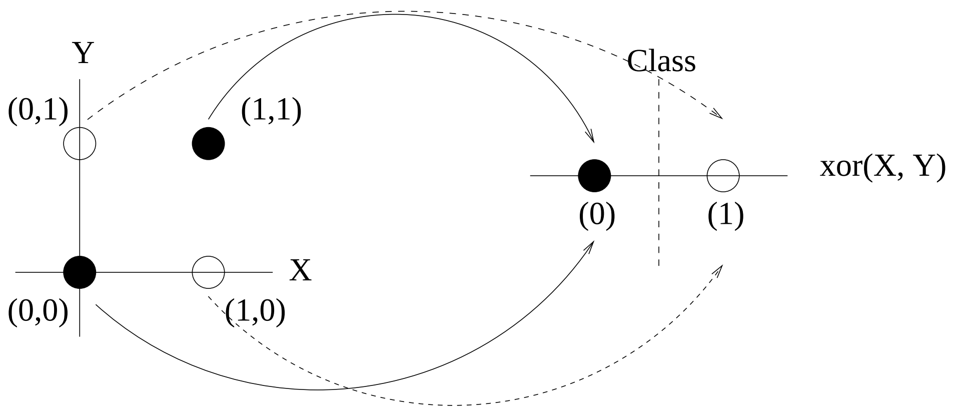


Figure 4: *XOR* in Two Feature Spaces. In the diagram on the left hand side, feature *X* and *Y* both take two values {0,1}. The black circles represent the case where the circle is a member of the target class *False.* The right hand diagram shows a combination of features *X* and *Y* into a single new feature. With this new feature (which can only take two values, *True* and *False*), there is a direct, separable relationship with the target class. In fact, the new feature has the same values as the target class.

in verifying, interpreting, and applying the model and (2) missing satisfactory models or patterns (Matheus, 1989).

1.3.2 Modification of Bias by Feature Construction

Classifier bias can be partially relieved by feature construction. Consider the simple perceptron learner. It cannot learn to discriminate the *xor* function (Minsky and Papert, 1969). If an *xor* operation is applied to the base features before the perceptron is induced, then the perceptron will be able to learn the disjoint concept. Even so, the perceptron is still limited to learning linearly separable concepts. The newly created feature presents a different space in which to learn as shown in Figure 4. The modified space allows the concept to be learned by the perceptron because the target classes are contiguous and linearly separable.

In the case of decision trees, providing pairs of features combined into a single new feature may compensate for the limitation of selecting features one at a time. A constructed feature may be a better selection for a tree node than either of its individual, constituent features. If so, the new feature will be selected for a decision node and the overall tree will be simplified. Since the *xor* function requires the conjunction of two tests, adding an *xor* feature to a decision tree learning problem will reduce the learned tree from (1) three nodes representing four paths from the root to a decision node to (2) one node representing two paths from the root to a decision node (see Figure 2). A specific benefit of feature construction to decision tree learning procedures is conservation of data. By reducing the number of splits needed to make decisions, at each decision point there is more data available. Hence, there is less sample bias at each node and better splits can be made.

In the weather example (Section 1.2.1), the average size of a tree constructed from the base features was 33 nodes. In contrast, the average size of a tree using the new features was four nodes and the average size of a tree using the new features and *Humid* was 12 nodes.

1.4 CONTRASTING FEATURE CONSTRUCTION AND RELATED METHODS

1.4.1 Modifying Data Representation

There are several methods of modifying the feature space of a dataset – a general term for all of these is *feature space transformations*. The three major feature space transformations are: feature selection, feature extraction, and feature construction. Feature selection is the process of choosing subsets of features for learning. Feature extraction is the process of turning very general representations into more specific representations. Feature construction, the focus here, involves creating new features for examples from previously existing attribute-value pairs. There are a number of topics and areas in the machine learning and statistics communities that relate closely to feature construction. The related topics include the other feature space transformations, constructive induction, predicate invention, and certain statistical techniques that implicitly perform feature space transformations.

1.4.2 Feature Selection and Feature Extraction

Feature selection can be viewed as a necessary prerequisite to performing feature construction – what existing feature or features will be used in creating new features? However, this is somewhat misleading. Feature selection, known in the statistical literature as the subset selection problem (Ben-Bassat, 1982), refers to the process of choosing a subset of features that will be used in making a model of the target. It does not refer to choosing features that will undergo further modification before the modeling process. If we consider feature construction as part of the modeling process (as opposed to a pre-processing step), then the distinction disappears. Since the distinction is not crisp, it makes sense to talk about feature selection and feature construction both as distinct and as integrated processes. Further, whether features are selected for direct analysis or for use in constructions, feature selection methods evaluate and order features and then pass the features to the next processing stage.

Feature extraction is the process of creating useful features from raw, uninformative data. For example, in a dataset of gray-scale images, replacing the image (e.g., a two dimensional grid of gray-scale values) with the number of objects in the picture is a feature extraction process. The operations performed in feature extraction are more complex versions of those performed in feature construction. While feature construction might compute the dot product of two vectors of features, feature extraction may compute a Fourier transform or perform other signal analysis over the entire example.

One distinction that has been made between feature construction and feature extraction is that feature extraction will usually result in significantly fewer features being present in the dataset (Liu and Motoda, 1998b). Feature construction may either add to the feature space or replace a few features with a single feature. Again, these are not defining characteristics, but general descriptions; feature construction and feature extraction live on a continuum. The continuum has complex operations and a large reduction of features at one end and simple operations and small feature set additions and deletions at the other end. A second distinction between the two transformations is that in feature construction many of the features available for feature construction are present in the dataset because of specific, individual reasons. In feature extraction, the base features are there because they are part of the natural representation of the object under consideration. For example, pixels in an image do not have priority over each other – each pixel is simply one unit of the entire aggregate image. Again, the difference between feature extraction and feature construction is a difference of degree.

In a chess learning task we may find an initial representation of a chess board as a two dimensional array specifying position by position what is located at each space on a board – a piece or an empty position. The detailed representation is correct and complete, but it does not capture any higher-level knowledge about what it means to build a winning chess position. Instead, concepts like point values of pieces, number of attacks available, number of pieces under attack, space controlled, and pawn structure summarize the nature of a chess position. These concepts are intermediate concepts that link the raw chessboard to the target concept of winning chess positions. Interestingly, it is much easier to explore the scenarios defined over these more expressive features then it is to consider all possible, legal chess boards. Since the features capture some of the essence of the position, working with more expressive features is valuable in learning. The raw data differs from the extracted data in the level of generality of the representation. For example, many board games can share a similar representation to chess; however, other games will not have the concept of pawn structure. Specific representations are generated by the application of knowledge to a problem.

Similar cases can be seen in computational analysis of bridge hands, web pages, gene sequence analysis, text documents, and, in general, any heuristically guided problem solving task. In fact, the heuristic guides used by humans for these problems often make use of constructed or extracted features from their respective domains.

1.4.3 Constructive Induction

Constructive induction (Michalski, 1983; Matheus and Rendell, 1989) is the process of manipulating the example descriptor space while performing an induction task. Here, the term *descriptor space* is used instead of feature space because the learning scenario is more general than learning from attribute-value pairs. Namely, in addition to attribute-value pairs that describe the examples, there can also be relational descriptions among the various descriptors. The addition of relations corresponds to moving from a propositional logic representation (i.e., the attribute-value representation) to some form of predicate or first-order logic representation. Some authors refer to constructive induction in ILP settings as predicate invention because newly created relationships are described by new predicates. Feature construction can be used to perform constructive induction, but it can also be used apart from constructive induction. More details on constructive induction and its relationship with feature construction are given in Section 5.1.

Feature construction can also be seen as a means to perform constructive induction. If the operations available in the feature construction process mimic the operations performed in the process of induction, then constructed features may represent values of hypotheses. Hence, a learning system and a feature construction system, working in concert, can *jointly* perform constructive induction. This can be seen clearly in the case of rule learning. Suppose that a feature constructor has available the Boolean operations and can perform equality tests over values of an attribute. Then a new feature may take the value $and(f_1 = green, not(f_2 = blue))$. Subsequently, rules having the new feature taking the value *True* would correspond to any rule that was induced with a left hand side matching $f_1 = green \wedge f_2 \neq blue$.

Features created by negation are particularly interesting if the induction process (e.g., a particular rule learner) does not have the ability to represent logical negation. In this case, adding the constructed feature adds to the total representational power of the inductive system with feature construction. Further, a feature that is constructed by testing for *inequality* with a given color can allow for generalizations not strictly justified by the dataset. For example, even if no data point has a color attribute with value *magenta*, the missing value is still covered by $f_{color} \neq blue$. The description by negation shows that the feature construction process can perform operationally similar tasks to the induction algorithm proper and may extend the hypothesis language of the induction process.

As a final connection between feature construction and constructive induction, there exists a trade-off between the representation of descriptors and the search necessary in a space of hypotheses. Different representations may require less search and may lead to better hypotheses. Hence, there is also a trade-off between search in the descriptor space and search in the hypothesis space. At its most extreme, search in the descriptor space can replace search in the hypothesis space. If the set of feature constructor functions contains symbolic value tests (e.g., $f_1 = blue$) and Boolean connectives (e.g., $and(f_3, f_4)$) then it can create pseudo-rules as constructed features (e.g., $and(f_1 = blue, f_2 = green)$). In numeric contexts, this sort of feature construction goes by the term *equation discovery*. In these systems, constructed features *themselves* mimic the target concept.

1.4.4 Statistical Vector Space Methods

On the logic-equation spectrum of learning systems, statistical methods are opposite of symbolic learners. An important class of these statistical methods redescribe examples by projecting them (i.e., mapping points) from one coordinate system to another. Kernel support vector machines take a description of data in one feature space and project it into another, higher dimensional space whose bases are different combinations of the original features. For example, using a polynomial kernel of degree 2, a dataset with features $\{f_1, f_2\}$ will be projected into a space where the points are described in terms of $\{f_1^2, f_2^2, \sqrt{f_1 f_2}\}$. In this more expressive space, the data may be separated by linear boundaries; in the original space, nonlinear equations may have been required to separate the concept regions as in Figure 3.

Projection from one space to another is an implicit manipulation of the feature space in two respects. First, in the computations there is no explicit representation of the features in the higher dimensional space. The implicit nature of the representation is beneficial because it makes the problem more computationally feasible. Second, the knowledge of feature combinations is encapsulated in the kernel function that takes the data from the low dimensional space to the high dimensional space. The encapsulation has the benefit of modularizing the learning process, but it requires that the knowledge be expressible within the kernel formalism. The kernel represents a dot-product between all pairs of features (i.e., distances between all examples). So, it may not be obvious how the knowledge in the kernel (i.e., the distances) represents knowledge in the world of the learning problem (i.e., objects and relationships). Also, the kernel is a relatively complete set of knowledge relating each example to every other example.

1.5 SEMANTIC FEATURE CONSTRUCTION

Why do we want to do semantic feature construction? First, we want to make use of domain and background knowledge when we perform a learning task. Second, we want to make use of available, off-the-shelf learners with little modification. If we incorporate background knowledge in feature construction before the learning process and produce data in the same form we are given (i.e., as attribute-value pairs), then our available learners can be used without modification. We will see that feature construction offers a means of incorporating background knowledge into the learning process. Third, on their own, constructed features can lead to insights about a dataset. Fourth, the use of explicit, semantic knowledge also allows testing the usefulness of that knowledge for learning. Pieces of knowledge can be evaluated by their benefit to the performance characteristics of learners.

1.6 HYPOTHESES

I propose that making explicit, declarative knowledge available to an automated feature constructor will result in positive changes to the representation of data as demonstrated by the performance characteristics of machine learners. By providing knowledge to the feature construction process, the performance of learning algorithms using feature constructed with background will be improved in comparison to learning algorithms using only base features. By performing feature construction as a pre-processing task, unmodified learning algorithms can make use of the constructed features.

The specific hypotheses that this dissertation tests are:

1. knowledge is useful in the feature construction process and it can be captured in a uniform fashion,
2. knowledge can be used in an automated fashion in the feature construction process via beam search,
3. knowledge-based feature construction results in increased generalization performance by learning algorithms with constructed feature sets, as compared to base feature sets, and

4. knowledge-based feature construction results in less time spent in feature construction and a greater increase in generalization performance, as compared to knowledge-lean feature construction.

1.7 OVERVIEW AND SUMMARY

Chapter 2 shows two annotated examples of the ISAK system and feature construction. Chapter 3 presents background on the methods used in this study. Chapter 4 offers a framework for discussing systems that perform semantic feature construction. Chapter 5 describes work related to semantic feature construction. Chapter 6 describes the tools, algorithms, and implementations used in this project. Chapter 7 details a number of experimental results generated by ISAK and discusses the implications of these results. Chapter 8 looks at future directions and considerations of my work.

Representational power is frequently analyzed as a characteristic of machine learners. It has become popular to deal with representational issues by feature construction. In the learning community, many attempts are being made to incorporate domain background knowledge into the machine learning process. My aim is to combine these two approaches by using background knowledge to drive the construction of new features.

2.0 ANNOTATED EXAMPLES OF SEMANTIC FEATURE CONSTRUCTION BY ISAK

2.1 HIGH LEVEL DESCRIPTION AND THE WINE PROBLEM

ISAK (Integrating Search and Knowledge) is a semantic feature construction system that takes as input:

1. a dataset in the form of attribute-value pairs,
2. a semantic description of the features and their interactions, and
3. an optional set of search parameters (defaults are provided).

As output, ISAK produces a modified dataset of attribute-value pairs (Figure 5).

The input dataset takes a format similar to the UCI wine dataset (Newman et al., 1998) shown in Table 5. The wine dataset contains 178 examples described by 13 numeric features. Each example of a wine belongs to one of three classes. The attributes are characteristics of wines: amounts of chemical compounds, quantifications of visual effects, and results of experiments on samples of the wine. The three possible target values represent the three regions from which the wines in the dataset are produced. Besides the dataset proper, the only other information in the data file is a distinction of continuous and symbolic valued features, names for the features, and an indicator for the target class.

An example of an input knowledge base and search parameters for the wine dataset is shown in Figure 6. The knowledge base contains the description of feature semantics, applicable constructor functions, and search parameters. These make up a complete specification for a single feature construction problem. A dataset and a construction knowledge base together form a *construction and learning problem* (CLP). The wine CLP is a self-contained example; other CLPs make use of

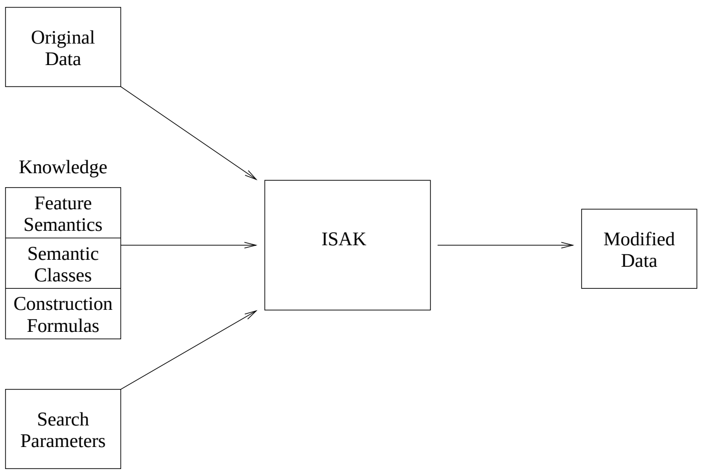


Figure 5: Overview of ISAK Input and Output. ISAK makes use of three types of knowledge about the data: facts about the features in isolation, facts about class relations among the features, and methods of creating features that may be applied when given facts are true. Search parameters control the progress through the space of possible features and the order in which prospective features are considered.

Table 5: Ten Examples from the UCI Wine Dataset. The attributes are named a_1 to a_{13}. The target class is *wine*. The wine classes all have value 1 because the example lines were taken from the top of the file. The second line of the file has values *c* and *d*. These indicate that the value is numeric or symbolic, respectively. The third line has the value *class* in the column that is the target class.

a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12	a13	Wine
c	c	c	c	c	c	c	c	c	c	c	c	c	d
													class
14.23	1.71	2.43	15.60	127.00	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065	1
13.20	1.78	2.14	11.20	100.00	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050	1
13.16	2.36	2.67	18.60	101.00	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185	1
14.37	1.95	2.50	16.80	113.00	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480	1
13.24	2.59	2.87	21.00	118.00	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735	1
14.20	1.76	2.45	15.20	112.00	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450	1
14.39	1.87	2.45	14.60	96.00	2.50	2.52	0.30	1.98	5.25	1.02	3.58	1290	1
14.06	2.15	2.61	17.60	121.00	2.60	2.51	0.31	1.25	5.05	1.06	3.58	1295	1
14.83	1.64	2.17	14.00	97.00	2.80	2.98	0.29	1.98	5.20	1.08	2.85	1045	1
13.86	1.35	2.27	16.00	98.00	2.98	3.15	0.22	1.85	7.22	1.01	3.55	1045	1

a built-in library of feature construction knowledge to supply the feature constructor functions and semantic types used to describe feature semantics (Section 6.2).

ISAK starts by building its internal structures from the information given in the problem description. For each base and constructed feature, the program tracks the feature's semantic type and an estimate of the feature's utility for learning[1]. For each constructed feature, ISAK records the features used in its construction. For the construction formulas, the internal structures record the allowable types of the input arguments, constraints on the details of these types, the resulting output type from a formula application, and the mathematical operation that produces new features values from the old feature values.

In the wine problem, ISAK:

1. Creates a semantic class hiearchy of features. The semantic classes are defined by lines 1-11 in Figure 6. The relationships among classes are defined by lines 13-19. A graphical depiction of these relationships is shown in Figure 7. Lines 21-42 place specific features into these semantic classes.
2. Creates feature constructor functions (FCFs), from their specifications, that will create new features from old features. The feature constructor functions are specified in lines 44-55 of Figure 6. A logical specification of these functions (and the features they construct) is given in Figure 8.
3. Loads the search parameters that will limit the search in the space of constructed features. The search parameters are specified in lines 58-64 of Figure 6.

The feature constructor functions specified in the wine KB are arithmetic functions that apply to the given features if those features are in unifiable semantic classes. For example, looking at Figure 7, *Flavinoid Phenols* and *Nonflavinoid Phenols* unify to the class of *Phenols*. A ratio between *Flavinoid* or *Nonflavinoid* and *Phenol* would give a relative quantity of that type of phenol. A ratio between *Flavinoid* and (all) *Phenols* would give the relative proportion of phenols that are flavinoid compounds. Features in the *Environment* semantic class cannot be combined with features in the *Genetic* semantic class (or any of *Genetic*'s subclasses) unless they are specifically placed in both the *Genetic* and *Environment* classes.

[1]ISAK uses information gain as the feature evaluation metric.

```

#
# semantic classes for the wine problem
#
col = SemNode("color")
gen = SemNode("genetics")
env = SemNode("environment")
phen = SemNode("phenols")
flav = SemNode("flavinoids")
nonflav = SemNode("nonflavinoids")
pro = SemNode("proanthocyanins")
biochem = SemNode("biochemical")

#
# put semantics classes in their relational hierarchy
#
topClass.addChildren(col, gen, env, biochem)
gen.addChildren(phen)
phen.addChildren(flav, nonflav)
flav.addChildren(pro)

#
# Match the features with their respective semantics
#
tmpTypeDict = {"a1": (biochem,),
               "a2": (biochem,),
               "a3": (env,),
               "a4": (env,),
               "a5": (env,),

               "a6": (gen,phen),
               "a7": (gen,flav, col),
               "a8": (gen,nonflav),
               "a9": (gen,pro),
               "a10": (gen,col),
               "a11": (gen,col),

               "a12": (biochem,),
               "a13": (biochem,)}
fd = ItemAttrDict()
for attr, classD in tmpTypeDict.iteritems():
    fd[attr] = BaseFeature(attr, GenQuantType(classD))
kbfeatures = fd.values()

#
# Create some operations that will work on GenericQuantitys
#
def f_GenericInherit(x,y):
    return (LGUs(x.classDesc, y.classDesc),)
unifArithOps = [Formula((GenericQuantityType, GenericQuantityType),
                        (f_notupperlgu,),
                        thisOp,
                        GenericQuantityType,
                        f_GenericInherit)
                for thisOp in f_add, f_div, f_mul, f_sub]
probformulas = unifArithOps


#
# Define the search parameters for the feature space search
#
searchparameters = {"minMsr": .9, "minMsrImp":.3, "junkMsr":.1,
                    "minFtrs":1, "minHgt":1, "minLeaves":1,
                    "maxFtrs":4, "maxHgt":3, "maxLeaves":4,
                    "maxNew": 75, "maxReturn":10, "maxOld": 75}

```

Figure 6: A Knowledge Base Used by ISAK for the UCI Wine Dataset. Comments in the code are denoted with #. Code segments are described in the text.

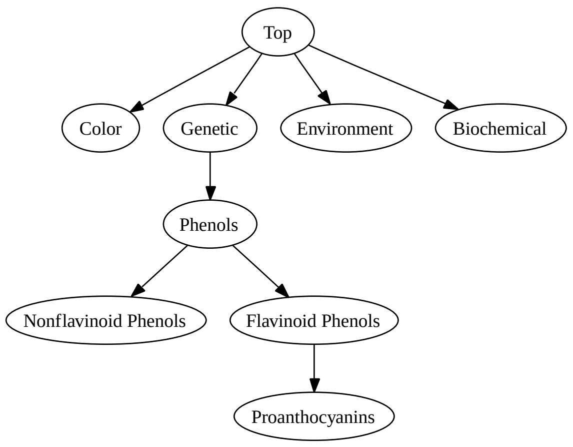


Figure 7: Semantic Hierarchy for Wine CLP. The hierarchy is used to specify similarities and differences among features. A single feature may belong to more than one semantic class. For example, *a11* is an element of both *Color* and *Genetic*.

The resulting values of the FCFs are defined by the arithmetic operation associated with each FCF. The resulting semantics have two pieces (1) the semantic type of the result is a *GenericQuantity* and (2) the semantic class of the result is the unification of the semantic classes of the initial features used in the construction.

The search is constrained by the search parameters, the types of the features, and the input/output characteristics of the construction formulas. New features are generated by matching the available types of features to the required types in the type signatures of the construction formulas. As type-compatible features are found, additional type-specific constraints are tested, and finally, structures for the features satisfying the constraints are built. The new features are then integrated onto the search beam subject to the beam search parameters. The beam integration process may also remove old features from the beam. Finally, the features that survive integration are partitioned into three distinct sets: (1) features that are known to be good, (2) features that may potentially produce good features with additional processing, and (3) features that may be discarded. The process continues iteratively until no new features are produced[2].

[2]Other stopping criteria, besides quiescence, are possible such as the time of execution or the number of rounds of beam expansion performed.

$$\begin{aligned} with\, op\, \epsilon\{+,-,*,/\} \\ \forall f_1, f_2 Feature(f_1) \wedge Feature(f_2) \wedge Unification(class(f_1), class(f_2)) \neq Top \\ type(f_1) = GenericQuantity \wedge type(f_2) = GenericQuantity \\ NewFeature(f_N) \wedge values(f_N) = op(values(f_1), values(f_2)) \wedge \\ class(f_N) = Unification(type(f_1), type(f_2)) \\ type(f_N) = GenericQuantity \end{aligned}$$

Figure 8: Operations for the Wine CLP. The equations express a logical formulation of four arithmetic feature constructor functions for the wine domain. Feature constructor functions define the characteristics of the features to which the functions may be applied and the results of that application. In English, the first three lines of the logical formulation say that the arithmetic operations apply to features that are *GenericQuantity*s and that they unify to something other than *Top*. Unification, in this problem, refers to checking for common ancestry in Figure 7. The next two lines state that the resulting feature has values defined by the arithmetic operation (i.e., one of addition, subtraction, multiplication, of division) and a semantic class defined by the unification of the input types. The last line says that the type of the new feature is a *GenericQuantity*.

The search for constructed features (Figure 9) can be summarized as follows:

1. generate new features based on (a) the type signatures of construction formulas and the types of the available features and (b) the specific typing information available in the features,
2. prune the new features that (a) do not satisfy the search parameters and (b) can never lead to satisfactory features, and
3. partition the remaining features into those that are known to be of interest and those that may be used to produce other new features of interest in subsequent rounds.

The procedure is repeated until it exhausts itself by generating no new *keepers* – features that are either (1) known to be of interest or (2) may be of interest in the future. The end result of running ISAK on the wine CLP is another dataset[3] (Table 6).

Table 7 shows the counts of features through the generation process. The six keepers from the last round do not exceed the value of the *maxReturn* (10 in the wine CLP) argument to ISAK so all six are returned. The path through the search space leading to these six features is shown in Figure 10.

[3]Following common convention, $ab = a * b$ for neatness of presentation.

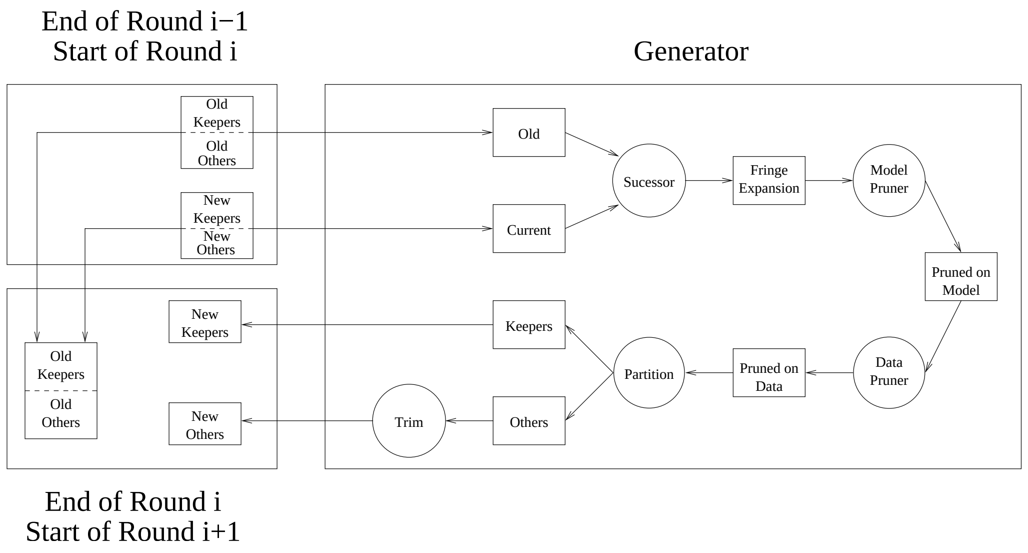


Figure 9: Data Flow through Generation of Features from Round to Round. At the start of each round, the old features for the current round are the features that were available in the previous round. The new features are those produced in the previous round of generation. When the *New Keepers* and *New Others* lists are both empty, the round to round iteration stops.

Table 6: New Features Constructed by ISAK for Wine CLP. Wines in the modified dataset have a mix of base and constructed features. Here, we show only the first ten wines from the original datset. *a7, a12*, and *a13* are features that survived from the initial feature set. The other features are numbered according to the order in which they, and other non-returned features, were produced. The constructions for the new features are: ftr_04501= $(a_7 a_{10})/(a_6 - a_{10})$, ftr_01868= $(a_7 a_{10}) - (a_{10}/a_{11})$, and ftr_04288= $(a_8 - a_{11})/a_{10}$.

a12	a13	ftr_04501	ftr_01868	a7	ftr_04288	Wine
c	c	c	c	c	c	1 2 3
						class
3.40	1050	-6.99	7.92	2.76	-0.18	1
3.45	1480	-6.89	18.15	3.49	-0.08	1
2.73	1150	-8.66	15.61	3.69	-0.15	1
2.88	1310	-4.77	15.54	2.91	-0.13	1
2.57	1130	-6.15	16.60	3.40	-0.11	1
3.52	770	-5.19	6.48	2.41	-0.17	1
2.71	1285	-6.86	13.74	3.25	-0.16	1
2.69	1020	-7.51	6.63	2.64	-0.24	1
3.53	760	-7.38	9.77	3.04	-0.14	1
3.44	1065	-9.41	10.82	3.17	-0.16	1

Table 7: Counts of Features in Data Flow for the Wine CLP. Here, the term *model* refers to the syntactic characteristics of a constructed feature: how many base features it uses, how many leaf nodes it has, and its height. The last two characteristics rely on interpreting the constructed feature as a tree (Section 4.4). The column headings make reference to the flow of data through the feature generation process (Figure 9). At the end of round 3, there are no *NewKeepers* or *NewOthers*, so ISAK terminates.

Round	Within Round Counts								End of Round Counts			
	Old	Current	Generated	Pruned on Model	Pruned on Data	Keepers	Others	Trimmed Others	Old Keepers	Old Others	New Keepers	New Others
Initial	<No generation. Filled from initial features.>								0	0	3	10
1	0	13	192	192	144	0	140	75	3	10	0	75
2	13	75	5280	5280	3022	3	2992	75	3	85	3	75
3	25	103	14240	0	0	0	0	0	6	160	0	0

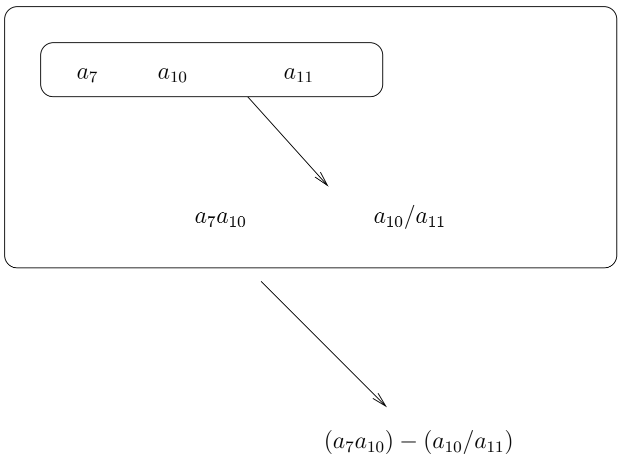


Figure 10: Path to Resulting Wine Features. The figure shows only those features used in the process of generating new feature 01868. *feature_01868* is expanded from the initial set of features in the innermost box. Other initial features, not shown in the innermost box, expand to the other constructed features shown in Table 6.

2.1.0.1 Trace of ISAK on the Wine CLP

To illustrate the operation of ISAK in some further detail, here is a trace of ISAK, operating on the wine dataset for a subset of the initial features: a_7, a_{10}, and a_{11}. The trace for these features was generated within the run of ISAK on the full, initial feature set (i.e., including the other a_i). So, the counts in the trace match those presented in Table 7. The trace is shown in Figures 11 and 12.

Figure 11 (lines 1-62) shows the initial problem setup, the setup of the initial search state, and the first round of generation. Figure 12 (lines 65-129) shows the second and third rounds of generation. The third round produces no models that are acceptable with respect to the search parameters given in lines 8-11. Specifically, the new features at the end of round two and the start of round three are all of height three[4]. Round *i* is responsible for generating all features of height $i + 1$. The maximum height allowed in the wine CLP is three (see Figure 6, line 64). Since all of the arithmetic operations add one to the height of any tree they are applied to[5], any application of the operations will increase the height of the new feature by one. Hence, any constructed features created from the new features coming out of round two will have height four and will be pruned immediately[6].

Another interesting point is that the *NewOther* list in round two (lines 102-110) shows features with evaluations higher than the evaluations in the *NewKeeper* list for the same round. The selection occurs because the *MinimalImprovement* search parameter requires that features be evaluated at least .3 higher than their parent features to be considered a *Keeper.* For feature 01642 (line 110, evaluation 1.3493), its parents are feature 00045 (line 61, evaluation 1.2050) and feature 00166 (line 56, evaluation 0.8569). Since $1.3493 \not\geq 1.2050 + .3$, feature 01642 is not a *Keeper.*

Now, let us look at the operations that take the results from round one to the results of round two. At the start of round two (and the end of round one), there are two *OldOthers,* a_{10} and a_{11}, and there is one *OldKeeper,* a_7. There are sixteen *NewOthers.*

[4]Constructed features may be represented as trees with the base features as leaves and the resulting constructed features as the root of the tree. See Section 4.4 for further details.

[5]The operations are applied strictly at the root of the tree (i.e., the whole feature), not to leaves or internal nodes (i.e., components of the feature).

[6]The immediate pruning of some features raises the issue of lookahead in the search to prevent generation of some features. Due to the complexities of algebraic and symbolic operation interactions, I did not pursue implementing lookahead in the generation process. I did implement lookahead for simple cases like that in the wine CLP, but that code is not used in the dissertation.

```

***Initial features from data file:****
FloatVariable 'a7'  FloatVariable 'a10'  FloatVariable 'a11'

***Class Attribute:
EnumVariable 'Wine'

***Search parameters:
maxReturn : 10 minMsr : 0.9 maxFtrs : 4 maxLeaves : 4 minLeaves : 1
genArgs : {'N': 40} maxNew : 75 minFtrs : 1 minHgt : 1 dumpFS : True
newFtrGen : topNGen minMsrImp : 0.3 maxOld : 75 maxHgt : 3
maxForGen : 10 junkMsr : 0.1

***Formulas:
" + "   with sig:  GenQuantType x GenQuantType -> GenQuantType
" - "   with sig:  GenQuantType x GenQuantType -> GenQuantType
" * "   with sig:  GenQuantType x GenQuantType -> GenQuantType
" / "   with sig:  GenQuantType x GenQuantType -> GenQuantType

*******************************************************************
************Setting inital search state from problem.*************
*******************************************************************
a11 0.668378829956      a10 0.767301917076    a7  1.26114284992

***Setup complete.***
***Results: ***
old kep:  new kep:  old otr:  new otr:
   0         3         0         10
********************* Lists for Initial Round ********************
*** old keep list:
*** new keep list:
a7
*** old othr list:
*** new othr list:
a11  a10
********************* End Initial Lists **************************

*******************************************************************
************************ Doing Round 1  **************************
*******************************************************************
OldPol CurPol Gen'ed ModPrn DatPrn NewKep NewOtr TrmOtr:
     0     13    192    192  144      0    140     75

***Initial Round Results***
old keep:   new keep:   old otr:    new otr:
   3           0          10           75

*********************** Lists for Round 1 ***************************
*** old keep list:
a7
*** new keep list:
*** old othr list:
a11 a10
*** new othr list:
ftr_00082 (0.6424) is: -(a10, a11) ftr_00070 (0.6847) is: -(a11, a10)
ftr_00191 (0.6920) is: /(a7, a10)  ftr_00034 (0.7490) is: +(a10, a11)
ftr_00166 (0.8569) is: /(a11, a10) ftr_00084 (0.8966) is: -(a10, a7)
ftr_00178 (0.9032) is: /(a10, a11) ftr_00143 (0.9151) is: *(a7, a10)
ftr_00180 (0.9710) is: /(a10, a7)  ftr_00095 (0.9717) is: -(a7, a10)
ftr_00189 (0.9921) is: /(a7, a11)  ftr_00093 (1.0020) is: -(a7, a11)
ftr_00141 (1.0141) is: *(a7, a11)  ftr_00167 (1.0288) is: /(a11, a7)
ftr_00071 (1.0536) is: -(a11, a7)  ftr_00045 (1.2050) is: +(a7, a11)
***************************  End Lists *******************************

```

Figure 11: Trace of Wine CLP for Three Initial Features. The trace shows the initial setup steps and the first round of generation for a subset of features. The subset was chosen to demonstrated the generation of features in Figure 10. Feature set counts (and abbreviations) follow from Figure 9 and Table 7. Subsequent rounds of generation are shown in Figure 12.

```
******************************************************************
********************* Doing Round 2 ****************************
******************************************************************
OldPol CurPol Gen’ed ModPrn DatPrn NewKep NewOtr TrmOtr:
    13     75   5280   5280  3022     3    2992     75

***Round 2 Results***
old kep:  new kep:  old otr:  new otr:
   3         3        85        75

********************** Lists for Round 2 ***************************
*** old keep list:
a7
*** new keep list:
ftr_01868 (1.2521) is: -(*(a7, a10), /(a10, a11))

*** old othr list:
ftr_00082 (0.6424) is: -(a10,  a11)
a11
ftr_00070 (0.6847) is: -(a11, a10)
ftr_00191 (0.6920) is: /(a7, a10)
ftr_00034 (0.7490) is: +(a10, a11)
a10
ftr_00166 (0.8569) is: /(a11, a10)
ftr_00084 (0.8966) is: -(a10, a7)
ftr_00178 (0.9032) is: /(a10, a11)
ftr_00143 (0.9151) is: *(a7, a10)
ftr_00180 (0.9710) is: /(a10, a7)
ftr_00095 (0.9717) is: -(a7, a10)
ftr_00189 (0.9921) is: /(a7, a11)
ftr_00093 (1.0020) is: -(a7, a11)
ftr_00141 (1.0141) is: *(a7, a11)
ftr_00167 (1.0288) is: /(a11, a7)
ftr_00071 (1.0536) is: -(a11, a7)
ftr_00045 (1.2050) is: +(a7, a11)

*** new othr list:
ftr_00837 (1.2450) is: +(/(a7, a11), /(a10, a7))
ftr_04145 (1.2521) is: *(a7, +(a7, a11))
ftr_01505 (1.2521) is: +(a7, +(a7, a11))
ftr_01393 (1.2567) is: +(a7, /(a10, a7))
ftr_05414 (1.2611) is: /(a11, /(a11, a7))
ftr_02575 (1.2611) is: -(+(a7, a11), -(a11, a7))
ftr_01248 (1.3043) is: +(+(a7, a11), /(a10, a7))
ftr_02317 (1.3083) is: -(/(a11, a7), +(a7, a11))
ftr_01642 (1.3493) is: -(/(a11, a10), +(a7, a11))
**************************  End Lists ******************************

******************************************************************
********************* Doing Round 3 ****************************
******************************************************************
OldPol CurPol Gen’ed ModPrn DatPrn NewKep NewOtr TrmOtr:
    75     78  11464      0    0      0      0      0

<No new features.  End of run.>

Number of features examined:  16936
Feature Space Search Time (Wall Clock):  8.61
Feature Space Search Time (User Time):  8.52
Feature Space Search Time (Sys Time):  0.09
Feature Space Search Time (User + Sys):  8.61

****** ( 2 ) Newly Created Features ******
ftr_01868 (1.2521) is: -(*(a7, a10), /(a10, a11))
a7
```

Figure 12: Trace of Wine CLP for Three Initial Features (Continued). The trace shows the second and third rounds of generation and some final performance statistics. The initial setup and first round are shown in Figure 11.

TopNGen (line 10), the successor function, steps through each available operation and attempts to match argument tuples to each operation's signature. Here, the four arithmetic operations are processed in an analogous manner, so let us consider multiplication. Since all of the initial and constructed features share the same type (i.e., *GenericQuantity*), the generation of potential tuples is limited only by the semantic classes of the features.

Generation proceeds by generating $(new, any\,other)$ and then $(old,\ new)$ where *new* and *old* are directly from *NewOthers* and *OldOthers* and *any other* is any element of either list that has not already been used in that tuple. So, some pairs in the generation process will be:

Using a_7a_{10} as *new* (i.e., (a_7a_{10}, any)):

$(a_7a_{10}, a_{11}), (a_7a_{10}, a_{10}), (a_7a_{10}, a_{10} - a_{11}), \cdots, (a_7a_{10}, a_{10}/a_{11}), \cdots, (a_7a_{10}, a_7 - a_8)$.

Using a_{11} as *old* (i.e., (a_{11}, new)):

$(a_{11}, a_{10} - a_{11}), (a_{11}, a_7/a_{10}), \cdots, (a_{11}, a_7 - a_8)$.

Each tuple is checked against the constraint list for multiplication. Combinations between features that are not in unifiable semantic classes is prohibited (e.g., (a_7a_{10}, a_1) is prohibited because a_7a_{10} is in class *Color* and a_1 is in class *Biochemical*). Those pairs that pass through constraint processing are instantiated as new features and returned to the *Expanded Fringe* list in Figure 9. The same is true for each of the other arithmetic operations.

The *Model Pruner* (Figure 9) is responsible for syntactic constraints on the models: these constraints include the height and breadth of the features considered as trees. The *Data Pruner* is responsible for constraints based on the features' values and evaluations. For example, one element of *NewOthers* is a_7a_{10}. Now, one of the newly generated pairs is $(a_7a_{10}, a_{11}/a_7)$. Applying multiplication to this pair gives $a_7a_{10}a_{11}/a_7 = a_{10}a_{11}$. However, $a_{10}a_{11}$ is of height two and was generated at some point in round one. It was not considered a *keeper* or an *other*. Hence, we discard $a_7a_{10}a_{11}/a_7$ by checking the cache of expanded features for the duplication.

2.2 TIC-TAC-TOE PROBLEM

The tic-tac-toe (TTT) dataset from UCI contains various TTT boards that are reachable during the play of a standard game of TTT. The boards are classified by the target value into two groups:

Table 8: Seven Examples from the Tic-Tac-Toe UCI Dataset. The columns of the dataset are positions on the tic-tac-toe board, starting in the top left corner and moving to the bottom right in row major order. The target class, *y,* has the value *p* (a positive example), if player *x* is the winner. It has the value *n* if the board is a tie or the board is a loss for player *x*.

tl	tm	tr	ml	mm	mr	bl	bm	br	y
x o b	x o b	x o b	x o b	x o b	x o b	x o b	x o b	x o b	p n
									class
x	x	x	x	o	o	x	o	o	p
x	x	x	x	o	o	o	x	o	p
x	x	x	x	o	o	o	o	x	p
x	x	x	x	o	o	o	b	b	p
x	x	x	x	o	o	b	o	b	p
x	x	x	x	o	o	b	b	o	p
x	x	x	x	o	b	o	o	b	p

wins for the *x* player and not-wins for the *x* player. The first several rows of the dataset are shown in Table 8. As the table indicates, the state at the end of the game is represented by the values at different locations on the board. The locations can take on one of three symbolic values indicating the presence of an *x,* the presence of an *o,* or neither – a blank, *b.*

The features in TTT have a high degree of interaction. Recall that the target concept of a winning position (for player *x*) is defined by three *x*'s in a straight line (row, diagonal, or column). Hence, combinations of only two features will not capture the target concept. Combinations of three features can replicate parts of the complete target concept, if they are carefully chosen.

Knowledge for the TTT problem includes:

1. knowledge of the structure of a TTT board: rows, columns, and diagonals,
2. knowledge of the values on the board: *x*, *o*, and *b*,
3. knowledge that the attributes represent pieces on a board at a particular position (e.g., attribute *tl* is the top left corner of the board; it is in the first row, the first column, and is on the diagonal with negative slope),
4. knowledge of an operation that counts the number of occurrences of the different values in a bag, and
5. knowledge that the problem is specifically defined to find where player *x* is the winner.

The symbolic domain knowledge is combined with the following search control knowledge:

1. minimum target evaluation is .08 (knowledge that the value of individual winning components – rows, columns, and diagonals – are weakly valued until combined),
2. minimum improvement over parents is .005 (similar to the knowledge in 1.),
3. minimum number of base features is 3 (encapsulating the knowledge that a win requires three elements),
4. minimum height is 1 (we have to apply at least one operation[7]), and
5. maximum number of base features is 3.

There are other search parameters, but the listed parameters are the most interesting in the TTT CLP. The search control *also* encapsulates domain knowledge, in addition to controlling the complexity of the search.

Given the operations and the search parameters, ISAK builds up counts of related board positions (i.e., row, columns, and diagonals). The counts, such as number of *x*'s in a row, form partial concepts of the actual target concept. The two operations provided to the TTT CLP are: (1) an operation that takes two pieces and produces a count on the number of those pieces with a distinguished value (here, *x*) and (2) an operation that takes a count and a piece and then produces a new count, incremented by one, if the piece has the distinguished value. The restriction of combinations to rows, columns, and diagonals is done by specifying a semantic class for each feature. The top left corner of the board is a member of the *topRow,* the *leftColumn*, and the *negativeDiagonal* semantic classes.

As specified, the CLP gives the learners adequate features to determine a good approximation of the target concept. Since only three base features are allowed in any constructed feature, there is no attempt by ISAK to combine the individual winning conditions into an aggregate condition. We do not attempt to learn the entire TTT winning concept for two reasons: (1) it is interesting to see how useful constructions of the partial concept are in predicting the complete concept and (2) the combinatorics of the search for the complete concept are more prohibitive than for the partial concept. To construct features that represent the complete winning concept, we would introduce an equality test (i.e., check if a row's *x* count is 3) and the Boolean *or*: Is row_1 a winner *or* is row_2 a winner *or* is row_3 a winner *or* ... ? These two additions would give ISAK the tools necessary to find the complete definition of a win for *x* in TTT.

[7]Leaves are considered to have height 0. So, tree heights count from 0, 1, 2, etc.

3.0 BACKGROUND

3.1 INDUCTIVE LEARNING FROM EXAMPLES

Feature construction occurs in the context of machine learning. Following Russell and Norvig (1995), the specific machine learning problem addressed here is:

> Given a collection of examples $\{(x, f(x))\}$ of a function f over a domain $x \in X$, return a function h that approximates f.

The function h that approximates the true, underlying function is called a *hypothesis* and its form or syntax is specified by a *hypothesis language*. The set of all hypotheses is called the *hypothesis space*. If f is in the *hypothesis space* H, then the learning problem is called realizable; if f is not an element of H, the learning problem is called unrealizable. When h is limited to functions that output a small, finite number of values the learning problem is called *classification learning* or simply *classification.* In addition to the language used to specify the hypotheses, the examples of f are described in a particular language called the *instance language*.

The *instance space* is the the domain of f. That is, the instance space is the set of all possible objects that might ever have a target value associated with them in this learning problem (see Figure 13). Feature construction modifies the set of attributes that measure the objects in the instance space. The set of all objects with a known target value is the *example space*; the example space is a subset of the instance space. The subset of examples used for learning is called the *sample space*. In many cases, the sample space represents a *randomly selected* sample of the instance space. The *sample space* may be divided into different sets for use in hypothesis development (training set),

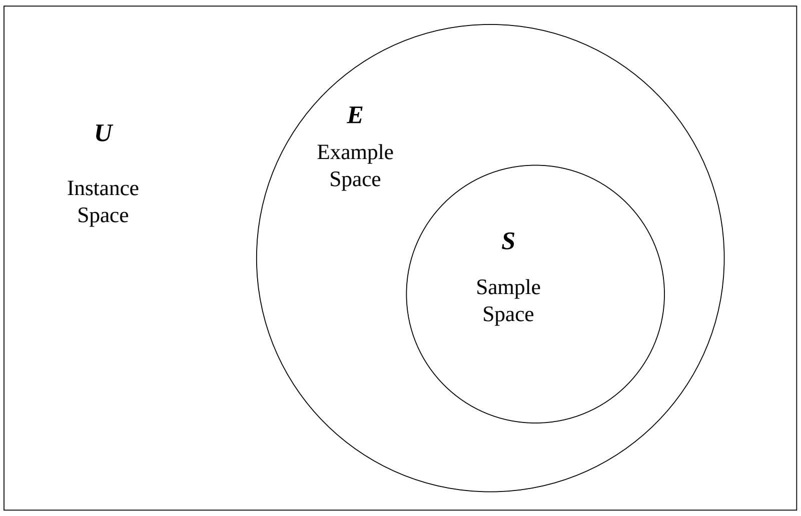


Figure 13: Instance, Example, and Sample Space Relationships. Instances are all possibly relevant data. The example space is the set of all labeled (i.e., classified) data; *E* may be infinite or open-ended. The sample space is the set of data that is available for learning or analysis; *S* is finite and closed. The training, validation, and test sets are subsets of the sample space.

tuning of hypotheses (validation set), and, finally, prediction of generalization ability back to the entire instance space (a test or hold-back set)[1].

Each individual instance is described on the basis of its *attributes* within the capabilities of the instance language. Formally, these are propositional statements of the form $height(example_1, 5.2)$. An attribute can be viewed as a function that maps an example to a value or characteristic of that example. Different attributes in a dataset may take values over different ranges. For example, attributes may be symbolic, discrete, set, real, integer, or Boolean valued. For a particular learning problem, values of attributes may have errors and examples may be mislabeled. However, I will only consider datasets for which *some* value is present for each example and attribute. I also restrict myself to problems with stationary features.

The elements in the set of examples are typically called cases, observations, examples, experiments, or individuals. In the weather learning scenario from Chapter 1, these are the different days

[1]Some presentations reverse the meaning of the testing and validation sets.

under consideration. The characteristics that describe each example are typically called measurements, variables, independent variables, or attributes. The attributes for the weather example are: *Precipitation*, *Temperature*, *Humidity*, and *Wind*.

Though sometimes used as a synonym for *attribute*, I will reserve the term *feature* for an attribute paired with a legal value. Legal values come from the domain of that attribute. An example feature is the pair (*Wind*, *Calm*). The set of all features taken together is called the *feature space*. In classical statistics, the *target attribute* is the dependent variable that is modeled on the independent variables. In the weather example, *Tennis* is the dependent variable. I also reserve the term *object* to refer to conceptual entities in the world of the learning problem. Namely, a single example may be made up of attributes over multiple conceptual objects in the world of the learning problem. Thus, in my usage, objects and examples are not synonymous.

There are many methods for creating a classifier or function. These methods include neural networks, Bayesian networks, clustering methods, genetic algorithms, regression frameworks, inductive logic programs (ILP), support vector machines (SVMs), decision trees, and rule learners (Mitchell, 1997; Hastie et al., 2001). Each of these systems has different strengths and weaknesses including its capability of dealing with different types of data (symbolic or numeric; discrete or continuous), its suitability to formal analysis, its speed of execution and the natural interpretation of the resulting hypotheses. These methods also have different degrees of success when dealing with missing data points, small datasets (i.e., small sample sizes), errors in measurement, errors in target value, and other practical concerns in real-world generalization tasks. Regardless of these differences, we want the learning system to (1) operate correctly because it is implemented correctly (i.e., to give the proper algorithmic output for a given input) and (2) to be accurate in its generalizations[2].

A subjective aspect of learning algorithms is interpretability of their algorithmic operation. In classifiers used to support human decision making processes, the complexity and interpretability of the learned function and the rationale for determining a given target value may be of equal importance to the performance of the learned function. For example, doctors may prefer simple,

[2]Datasets, learning algorithms, and the interaction between dataset and algorithm (e.g., sample bias, inductive bias, and bias mis-match) may limit classification accuracy even with a program that correctly implements the learning algorithm.

concise rules to complex mathematical formula because he or she can more easily compare the knowledge encapsulated in the rules with his or her own knowledge and experience.

A second subjective issue in data analysis is the use of learning algorithms for exploring data. In many settings, vast collections of data lack adequate theory to explain what phenomena the data represent. That is to say, the data exist as isolated points, not as a collection related to a compact theory. In fact, there may be little indication what patterns exist in the data. The methods of exploratory data analysis are used to find such patterns. Defining a classification problem may not be an appropriate first step in the absence of clear ideas about the data. In this case, how a learning system gathers data, looks for groups of similarities, and makes recommendations from these, can offer insight into the data.

Both the subjective aspects (i.e., interpretability of results and exploration of datasets) and the objective measures (i.e., accuracy, bias, and speed) of learning systems can be improved by semantic feature construction. This dissertation provides evidence of improved accuracy.

3.2 INDUCTIVE BIAS AND FEATURE CONSTRUCTION

3.2.1 Inductive Bias

Feature construction is intimately tied to the topic of bias in learning (Section 3.2.3). Mitchell (1982; 1990) phrases the problem of bias in the following context. In a learning problem, we have:

1. a language of instances,
2. a language of generalizations (i.e., hypotheses),
3. a matching predicate for matching generalizations to instances, and
4. sets of positive and negative training instances.

The third of these, the matching predicate, is sometimes forgotten when the hypothesis language and the instance language are very similar or the same. However, the generality of a matching predicate allows for substantial differences between the generalizations and the instances – namely any difference that can be expressed in the matching predicate's language.

Mitchell defines bias as any basis for choosing one generalization over another, other than strict consistency with the observed training instances. Two sources of bias are (1) the language used to represent hypotheses and (2) the procedure used to search or optimize among these hypotheses. An unbiased hypothesis language must be able to express the powerset (i.e., every possible subset) of the instance language. Decision trees, if allowed to grow arbitrarily large, can represent any subset of an instance space[3]. An example of an unbiased search procedure is Mitchell's Candidate-Elimination algorithm. Candidate-Elimination finds all conjunctive hypotheses consistent with the given training data. Utgoff and Mitchell (1982) link the idea of bias to feature space modification by defining the *dynamic bias* of a learner as its ability to adaptively modify its attribute set.

Using an unbiased hypothesis language (e.g., any language capable of representing arbitrary length disjunctive normal formulas (DNFs)) with Candidate-Elimination results in an unbiased learning procedure. But, as Mitchell notes, a learning system with no bias in its representational language and no bias in its algorithm is equivalent to performing lookup in, or matching against, a table of the learning examples. Specifically, Candidate-Elimination in a space of DNF hypotheses will produce DNF representations of the given training data. While this is not worthless, it certainly does not allow for generalization beyond the given data.

Explicit representation of biases allows (1) experimentation on the effects and interactions of the biases, (2) selection of biases appropriate to the problem, data, and results desired, and (3) search over the biases for a good bias for a given learning task. Provost et al. (1999) describes a learning system, RL, that uses explicit bias representation for use in knowledge discovery. Provost (1992) and Provost and Buchanan (1995) discuss the methods used to choose between different explicitly represented biases. Explicit representation of the feature construction process allows control of the representation bias in an analogous manner. The explicit representation of knowledge and search parameters in ISAK is driven partially by the goal of explicit control of bias.

Because learning systems have different inductive bias, different learners will learn some concepts better than others. Imagine using a rule learning system to model a relationship that is best expressed as a complex formula relating the descriptive features to the target value. The rule learner may lack the appropriate representation to describe the concept succinctly. This does not

[3]In the worst case, it could represent every possible instance (i.e., every combination of values for each attribute) individually.

imply the concept is unlearnable. It simply means that the underlying concept and the learner have a mismatch in the form used to represent the relation between the features and the target.

3.2.2 Inductive Bias of Decision Trees

The language bias of a standard decision tree classifier is that concepts are represented as a disjunct of conjunctive attribute tests and that each conjunct of the test examines a single attribute. Sequences of conjunctive tests form a very general language. If the tree is allowed to grow arbitrarily large, it can express any subset of an instance space with a finite set of attributes which have a finite number of values. The algorithmic bias of decision tress is normally expressed by starting with trees of height 0 and recursively adding leaves as necessary. Hence, part of their bias is that a classification hypothesis consists of a sequence of tests on individual attributes. The preference of shorter to longer trees is an application of Ockham's razor.

By adding single nodes at each step, a decision tree may have problems when two or more features have a high degree of interaction (Setiono and Liu, 1998). For example, a decision tree may ignore two important features for a third feature that is, individually, more important. This would happen when two features are very important when considered jointly but are less important when compared individually to the third feature. Symbolically, let *util* be a function that measures the learning utility of a feature or set of features, then we can say $util(\{f_1, f_2\}) > util(f_3)$ but $util(f_1) < util(f_3)$ and $util(f_2) < util(f_3)$.

3.2.3 Bias and Feature Construction

Kramer (1995) states:

> From a theoretical point of view, the algorithmic bias is the only justification for feature construction in propositional learning. For standard propositional learning algorithms using DNF-hypotheses or decision trees of arbitrary size, the construction of new features does not change the basic ability to express a concept. In other words, the occurrence of the real concept in the hypothesis space is not affected by the construction of new features. So in the propositional setting emphasizing the language bias does not make sense except for fixed-size languages. (pages 4-5)

I take two issues with these statements: the first is a matter of degree and the second is a matter of principle. Rephrasing Kramer's claim, we can say that a dataset represented as attribute-value pairs can be expressed as a disjunct of conjuncts (i.e., in a disjunctive normal form, or DNF) for exactly those attribute-value pairs in a desired class labeling. Consider the problem of classifying cars as Japanese. The DNF form defining a car as Japanese is (make=Honda *and* model=Accord) *or* (make=Toyota *and* model=Camry) *or* (make=Toyota *and* model=Celica) *or* ... for each make and model pair of Japanese car.

In the hypothesis space for trees and rules, the DNF is equivalent to specifying neighborhood by neighborhood, every region of the instance space and taking the union of all these regions as a class (i.e., the class of Japanese cars). Certainly, with enough conditions and disjunctions, we can express any arbitrary, finite region of a larger space. Phrased as an adversarial problem, if you show me an example outside the union region, then I can simply add its attribute-values as another disjunct to the current hypothesis.

It is a tautology to claim that we can express a set of propositional claims in a DNF expression – this is simply an extensional definition of a concept. However, much like Mitchell (1990) points out (see Section 3.2), the only learning that can be performed without bias is *rote learning* (i.e., memorization of training examples). We are subject to the same limitations in a DNF expression of concepts expressed feature by feature and instance by instance. We could learn such a concept but it would need to be presented in instance/region pairs and we will not have achieved any generalization. In the Japanese car example, we would have missed the fact that (in a simplified sense) *Make* is sufficient, without *Model,* to know the country of a car. Furthermore, allowing arbitrarily complex hypotheses (i.e., a tree with ever-increasing depth) will increase the amount of overfitting done by the tree.

The second issue, a matter of principle, is that there is not an *a priori* necessity that the target concept be a subset of the example space. The training data for the target concept may be[4] a subset of the target concept. However, the concept space, instance space, and hypothesis space need not have the nice properties that (1) the hypothesis space is the powerset of the instance space and (2) the concept space is a subset of the hypothesis space. In other words, it need not be the case that the hypothesis space can represent all subsets of the instance space (i.e., distinguish all possible groupings of instances). Hence, the hypothesis space may not be able to represent some potential target concepts.

[4]At least up to errors in feature values and in target value.

Kramer's concerns may be more justified in learning problems with target concepts defined over a finite domain. However, with a simple infinite domain, we can construct a problem for which his claim fails. Consider learning the concept of the even numbers over the entire natural numbers. So, the elements of the positive concept class are 0,2,4,... Now, without feature construction, a decision tree must be infinitely (not arbitrarily) large to specify this concept. The concept space is not a proper subset of the hypothesis space. However, with feature construction, we can define a perfect classification tree with only one decision node: $val\, mod\, 2 = 0$. Given that any learning problem with continuous features either (1) has an infinite domain or (2) must be discretized (which is itself, a feature space modification), Kramer's limitation of feature construction to solving algorithmic bias seems short-sighted.

3.2.4 Clarifying the Selective and Constructive Induction Dichotomy

There is disagreement in the machine learning community as to what properly defines the process of constructive induction. Since feature construction is a means to perform some aspects of constructive induction, it is important that these terms be defined precisely and adequately illustrated.

A first reference to constructive induction is given in Michalski (1983):

> If every descriptor occurs in the initial concept descriptions D_i, $i = 1, 2, ...$, then the rule is selective, otherwise it is constructive.

In the same volume, Dietterich and Michalski (1983) state:

> A generalization rule is a transformation rule that when applied to a classification rule[5] $S_1 \succ K$, produces a more general classification rule $S_2 \succ K$. This means that the implication $S_1 \Rightarrow S_2$ holds. A generalization rule is called selective if S_2 involves no descriptors other than those used in S_1. If S_2 does contain new descriptors, then the rule is called *constructive* (pg. 46).

Dietterich and Michalski summarize constructive induction as "any form of induction that generates new descriptors not present in the input data" (pg. 47). Dietterich and Michalski limit the term *descriptors* to variables, predicates, and functions (i.e., the expressive components of predicate calculus). Now, given the list of descriptors and definition of constructive induction, we can rephrase the definition of constructive induction as *any form of induction that generates new variables, predicates, or functions not present in the input data*.

[5] Here, $\succ$ is used to indicate "classifies to class".

A decade later, Michalski is quoted by Kramer (1995) as distinguishing between *empirical learning* which uses little domain knowledge while constructive *induction* uses more domain knowledge. Michalski continues describing the distinction,

> A more precise way to characterize this distinction is that in empirical induction the description space for examples and for the hypothesis space is the same, while in constructive induction these spaces are different (pg. 2).

It appears that *empirical learning* is a synonym for *selective induction* in this case.

Finally, Bloedorn and Michalski (1998a) state,

> constructive induction is a general approach for coping with inadequate attributes found in original data. It uses two intertwined searches – one for the best representation space, the other for the best hypothesis within that space – to formulate a generalized description of examples (pg. 30).

Their view can be compared with a more restrictive view presented by Matheus and Rendell (1989). They write “We ... reserve the term constructive induction to refer to the prediction of unobserved, disjunctive regions in the instance space” (pg. 645). Presumably, prediction is done using constructed features – not by other descriptors. Matheus and Rendell also write,

> If generalization is performed during feature construction, the features generated may “predict” regions of positive and negative instances that have not been observed. We call this form of feature construction *constructive induction* (pg. 647).

The statement should also be understood in the context of Rendell’s belief that constructive induction is the process of merging disjoint regions of the instance space (Rendell and Seshu, 1990).

Matheus and Rendell present an example of their distinction. Their example of the difference between “constructive compilation” (feature construction) and constructive induction is the difference between (1) observing instances from regions *A, B,* and *C* and creating a new feature *or(A,B,C)* and (2) observing instances from regions *A, B,* and *C* and creating a new feature *or(A,B,C,D)* where *D* matches some generalization of *A, B,* and *C*. In this context, a region is some cross-product of the space of attribute-value pairs.

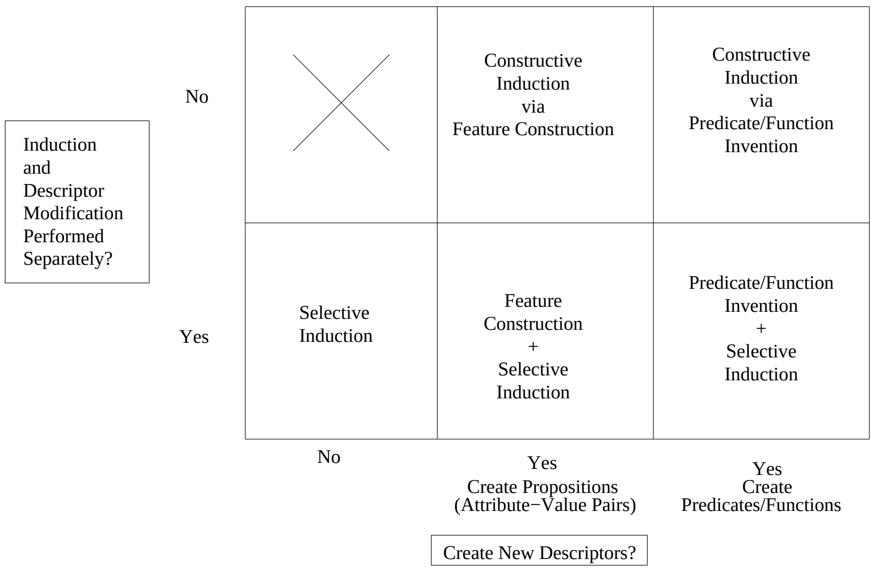
Matheus and Rendell instantiate the distinction by noting that on a checkers board, the relative positions of checkers can be more important than the absolute positions of those pieces. Hence, if some property is true of checkers at {(6,8), (7,7)}, at {(2,8), (3,7)}, and at {(2,4), (3,3)}, then we can generalize that to being true at (conservatively) {(6,4), (7,3)} or being true (liberally) at {(m,n), (m+1, n-1)} with some bounds checking to ensure both pieces are on the board. The general statement is equivalent to specifying that the checkers have some property when they are at a distance of one square on the downward diagonal. The property is part of the concept of *jump-ability*. In either the conservative or liberal case, Matheus and Rendell's version of constructive induction can occur. The more specific generalization predicts an unobserved region of the instance space, if that pair of pieces did not occur in the dataset. The more general statement would only predict unobserved regions (i.e., a set of examples) when some diagonal pair of pieces are missing from the dataset.

The most limiting definition of constructive induction is exemplified by (Pazzani, 1998), "Constructive induction is the process of changing the representation of examples by creating new attributes from existing attributes" (pg. 342). Here, there is no notion of induction as part of the change of representation. So, Pazzani's definition reduces to stating that constructive induction and feature construction are the same.

Constructive induction, as such, should retain its general definition as originally proposed by Michalski. Therefore, I will reserve the term *feature construction* for the task of creating new features (i.e., new attribute-value pairs) from preexisting features (i.e., preexisting attribute-value pairs). Feature construction may occur simultaneously with induction, in which the system doing both processes is performing constructive induction (Figure 14). If feature construction occurs separately from induction, the system is performing selective induction and feature construction. If results from selective induction are used to guide feature construction, the system becomes a constructive induction system. Feature construction may be performed with or without search or optimization in the space of hypotheses. ISAK performs feature construction with no input from the induction system or its results. Hence, ISAK is part of an induction system with separate modules for feature construction and for selective induction.

Now, are learning algorithms that develop conjunctive lists of attributes[6] creating new descriptors? For example, a rule learning system may create the pseudo-feature F $(Color = Blue) \wedge (Shape = Box)$ in a hypothesis and F may not be a descriptor in the input (i.e., only the constituent features are in the initial descriptor space). I believe the proper interpretation of these

[6]A similar analysis applies to mathematical hypotheses where parameters of a function are fit to match a dataset. In this instance, the mathematical function defines the space of possible hypotheses (which are pseudo-combinations of the input features).



No
Yes
Induction and Descriptor Modification Performed Separately?
Constructive Induction via Feature Construction
Constructive Induction via Predicate/Function Invention
Selective Induction
Feature Construction + Selective Induction
Predicate/Function Invention + Selective Induction
No
Yes Create Propositions (Attribute−Value Pairs)
Yes Create Predicates/Functions
Create New Descriptors?

Figure 14: Relationships Among Selective and Constructive Induction and Feature Construction. Two axes can be considered in combining representation modification and learning. The vertical axis asks whether there is a simultaneous search through the space of representations and the space of hypotheses. The horizontal axis asks what sort of descriptor space modifications are performed. Learning problem representations are typically either (1) attribute-value pairs or, in inductive logic programming, (2) a combination of propositions, predicates, and functions. If separate searches are performed, there are two distinct steps in the joint inductive process. If the two searches are intertwined, one of the descriptor modification methods is used to perform constructive induction.

pseudo-features is as the *selection* of preexisting descriptors, not as the creation of new descriptors. These descriptors are tied to individual attribute values. They are bound to predefined positions in a given hypothesis language. Phrased another way, these descriptors are selected into appropriate positions in the hypothesis language. Such a phrasing meshes well with the notion of selective induction proposed by Michalski. Likewise, Callan (1990) notes that the distinction between learning concepts and creating terms can be largely arbitrary.

For example, in an example space of cars on the highway, a particular car has a make, model, and year. Hypotheses in a rule or tree induction system may note that *make = Ford* has some special property relating to the concept class. Placing this feature into a conjunct or a tree node has the effect, when added to other attribute-value pairs, of creating a new prototypical or abstract instance of a car and claiming it has that special property. Here again, we see a justification for calling these processes *selective induction*. We are selecting attribute-value pairs and placing them, appropriately, into a predetermined hypothesis language. Also, we see the justification for an older term – *similarity based learning*. Those instances matching (i.e., most similar to) the prototype instance are grouped together. A suitable definition of feature construction requires that new descriptors, not expressible in the hypothesis language, be placed into the preexisting hypothesis language. By this definition, kernel methods are feature construction because they map problems into a space that is amenable to linear hypotheses.

Consider a second example. If we have numerical attributes, we may observe an attribute-value pair *speed=49.2325*. A conjunct that uses this feature will be limited to representing a smaller portion of the hypothesis space than a feature $speed\,in\,[49, 50)$. The wider interval might result from a simple discretization of the attribute *speed*. Now, we may temporarily ignore the problem of the infinity of reals and the large number of floating point values between 49 and 50. Even so, collecting enough data to form an arbitrarily fine mesh of values between $[49, 50)$ will be difficult. Collecting enough data to justify the claim that *all* values between 49 and 50 should be classified in the same way is unlikely to occur. Thus, there is a consideration apart from the dataset itself that justifies discretization – that is, discretization is a bias in the learning process. In the general case, the use of constructed features (i.e., new descriptions of that data) is a bias and it allows for generalizations to be made on the basis of considerations *apart from the data*. This is a forceful justification for calling a process *constructive* induction, though I have not seen it stated this way previously.

Recall that an unbiased hypothesis language must be able to represent the powerset of the instance space (i.e., all possible subsets of the instance space and hence all possible concepts in that instance space). If the hypothesis language can represent one hypothesis that contains an element outside of the instance space, then the hypothesis language itself is again biased because it allows generalizations that, by definition, cannot be supported by the data. A simple example of this is a hypothesis containing $val < attr$. If that attribute's values in the instance space are never greater than 100, then an unbiased statement would be $val < attr <= 100$. However, allowing *val* to go beyond 100 introduces bias. Careful selection of feature construction operations paired with a learning system can result in the constructive generation of hypothesis not available to the original learner.

3.3 FEATURE CONSTRUCTION

3.3.1 Feature Construction

Feature construction, whether performing constructive induction or operating independently, is a conceptually simple method to create new features from old. Since problem representation is such an important part of the machine learning task, it is not surprising that much work has been done using feature construction as the means of representational modification.

Just as constructive induction can be driven by hypothesis, data, and knowledge, feature construction can be initiated on the same bases (Figure 15). *Data-driven feature construction* operates by using feature selection metrics to guide the choice of operands. *Hypothesis-driven feature construction* relies on the output of a learning algorithm to determine the need for feature construction, the operands to use in feature construction, and, potentially, the operations to perform. *Knowledge-driven feature construction* uses any other information that can be applied to the problem to create new features. *Results-driven feature construction* has not been explicitly discussed except in the context of detecting the need for feature construction. When a classifier produces

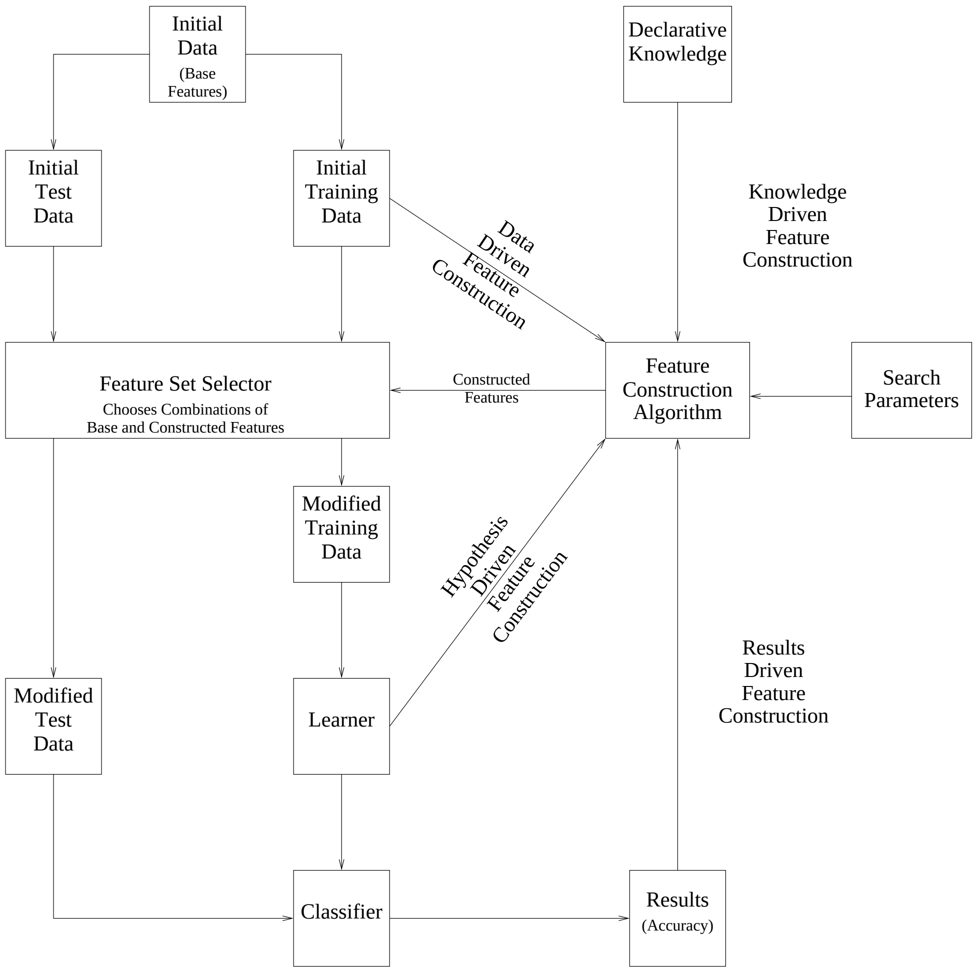

Initial Data (Base Features)
Declarative Knowledge
Initial Test Data
Initial Training Data
Knowledge Driven Feature Construction
Data Driven Feature Construction
Feature Set Selector
Chooses Combinations of Base and Constructed Features
Constructed Features
Feature Construction Algorithm
Search Parameters
Modified Training Data
Hypothesis Driven Feature Construction
Results Driven Feature Construction
Modified Test Data
Learner
Classifier
Results (Accuracy)

Figure 15: Feature Construction Types in Classification Learning. In the present work, the feature construction algorithm is ISAK. ISAK makes use of data-driven and knowledge-driven feature construction to construct features.

poor or otherwise unacceptable results one coping strategy is to start performing feature construction. Additionally, certain deficiencies in the results (e.g., certain subsets of the example space are being misclassified), may point to certain feature construction possibilities. Two or more of the primitive construction methods may be combined for *multi-strategy feature construction.*

ISAK is a multi-strategy feature construction system, making use of data-driven and knowledge-driven feature construction methods. ISAK uses data values to generate an information gain measure. The information gain is used as a heuristic utility value for base and constructed features. ISAK uses declarative knowledge and search parameters to perform knowledge-driven feature construction.

The problem of creating new features can be cast as a problem of searching through the space of possible constructed features. A well-defined search problem requires the following pieces:

1. a starting state,
2. a description of the current state in the search space,
3. a means of progressing from one state to the next, and
4. a stopping criterion or goal test that determines when either an optimal or a satisfactory final state is found.

When searching for a good set of features, these requirements become:

1. the initial set of features to be considered,
2. the current set of old and new features to be used,
3. a successor function that applies feature constructor functions to the current set of feature, and
4. a satisfactory set of final features to be used.

ISAK does not attempt to generate *all* possible features. The feature constructor functions that ISAK uses for a problem define a space of *plausible features* for that problem. The beam search is a process of selecting which of these plausible features should be used for classification.

3.3.2 Form of the Dataset

All of the forms of feature construction that I consider can be viewed as vector-wise operations over columns of features. For example, closing values of stocks listed day-by-day represents a historical time series for one day, going back over the previous examples (i.e., previous days). For ISAK to perform feature construction over the closing prices, the column must be expanded into a sequence of attributes for each example (i.e., a row). In the case of stock data, a redescribed example would gain attributes

$$\{\$_{Day-1}, \$_{Day-2}, \ldots, \$_{Day-n}\}$$

or a single *TimeSeries* attribute with the same values in a list for some length of history. The expansion of the attribute set comes at the cost of representing certain pieces of information many times. Specifically, each feature of every example is now represented explicitly. The conceptual simplicity of explicit representation makes the expansion a useful trade-off in ISAK.

3.3.3 Utility of Feature Construction, Feature Space Flaws and Troublesome Concepts

An appropriate feature space may reduce the required running time or example complexity for a learning algorithm. Blum and Langley (1997) note two extremes on a spectrum of learning algorithms. At one end are algorithms that show no preference among features – an example is a clustering algorithm. As irrelevant features are added, the number of examples needed to ensure a certain accuracy increases exponentially. Following Rendell's region merging perspective (see Section 3.2.4), feature construction can help these algorithms. Merging disjoint regions of a given concept will bring more examples of the same class in closer proximity to each other. So, the distances between related examples will be reduced. On the other end of the spectrum, there are algorithms that explicitly try to judge the relevance of features. These algorithms can also benefit from a reduced, more expressive feature set by spending less time sorting out features during their own execution.

Feature construction can simplify the description of a classifier if the data have a suitable representation. A good representation, in this context, is any representation that serves its required purpose – accuracy, conciseness, etc. Because constructed features are, in some sense, a summary of their parts, it follows that they can replace two or more features with one useful feature. This process can be repeated iteratively. If such features exist and can be found, then the classifier using them will be more concise than one without that feature (Fenner, 2002). The simplified description can save both (1) computation time in learning and classification and (2) storage space for the classifier at the cost of computation time during feature construction.

Finally, improvements in classification accuracy of learned hypotheses are an important result of feature construction. Many, if not most, feature construction systems take accuracy improvement as one of their measures of success.

3.3.3.1 Instance Space Flaws One categorization of instance space flaws (Bloedorn and Michalski, 1998b) is into (1) problems of over-precision, (2) problems of attribute interaction, and (3) problems of irrelevant attributes. Over-precision occurs when a feature has more values than are necessary to delineate the target concept. Solving this problem involves merging distinct values and creating an abstracted feature. Bloedorn and Michalski discuss this in the context of continuous feature discretization but other justifications are also possible. Symbolic values may form conceptual hierarchies apart from numerical similarity. Value hierarchies are discussed in (Aronis and Provost, 1997) in the context of rule learning with hierarchies of feature values.

Attribute interaction is directly addressed by feature construction. If two features have a degree of interaction and a function of those features can account for their interaction, the constructed feature can replace two constructed features with a single feature. The resulting feature set is more concise and removes the interaction.

A second categorization of instance space problems was proposed by Rendell and Seshu (1990). These authors consider *dispersion* and *blurring*. These two problems are tied to two biases in learning algorithms. The first bias is the *continuity assumption* that similar classes occupy similar regions of the instance space. Because of the continuity assumption, *dispersion* may occur when instances with shared classifications are spread over the available feature space in similarity based learning algorithms. Dispersed features may arise because of inappropriate ordering of the values of a feature. The result of dispersion in the instance space is verbose, over-fit hypotheses. Over-precision is one form of dispersion.

The individual choice of features included in a hypothesis is a second bias in many algorithms. Such an algorithmic bias may prevent appropriate features from being used in a hypothesis because of *blurring* (Rendell and Seshu, 1990). Blurring occurs when the worth of an individual feature cannot be properly evaluated in comparison to the other individual features. This may happen when each feature has (1) relatively little information content or (2) features have a high degree of interaction that is not revealed individually. Consider a set of five features $f_1, \ldots, f_5$. If $f_1, \ldots, f_3$ have high individual evaluations compared to f_4, f_5 then the the fourth and fifth features may never be included in a hypothesis. However, it is possible that f_4 and f_5 together are more useful then any of the other features (i.e., any feature constructed from f_4 and f_5 will be better than any constructed from the other features). Blurring is a form of attribute interaction.

To relate these categories, the presence of dispersion implies that blurring will also take place. Consider the *xor* function. The concept points are evenly distributed over both values of both features taken individually. Hence, the information content of each feature must be equal and thus there can be little reason to select either feature over the other feature unless there is relevant background knowledge available.

3.3.3.2 Hypothesis Flaws in Decision Trees Problems in the instance space can lead to problems fitting hypotheses. In the domain of decision trees, we can isolate three classes of flaws: replication, repetition, and fragmentation (Setiono and Liu, 1998; Pagallo and Haussler, 1990). *Replication* occurs when subtrees of a decision tree are duplicated at different points within a larger decision tree. *Repetition* occurs when a single feature's value is tested repeatedly on a path to a leaf node. *Fragmentation* occurs when the sequence of tests in a tree lead to overly small partitions of the data which in turn causes overfitting (i.e., estimation bias) and increased testing error.

To see how feature construction can alleviate these issues, consider a tree with a replicated subtree. If a replicated subtree is compressed into a new feature, then each occurrence of the subtree will be replaced by a single feature test. The overall tree will be compressed as a result. Also, the examples that were previously pushed to the leaves of the replicated subtree will be aggregated under the new feature. Thus, the new feature will have more examples available; this should improve its sampling error. If the new feature is tested at a higher point in the decision tree (i.e., closer to the root), it will also aggregate classes from lower in the tree to the new feature(s) higher in the tree and relieve fragmentation.

4.0 FRAMEWORK: A KNOWLEDGE LEVEL DESCRIPTION FOR FEATURE CONSTRUCTION

A knowledge-driven feature construction system must represent, manipulate, and utilize background knowledge. Here we address the issue of *what* is represented. The overall architecture I use to describe feature construction is presented in Figure 16, Figure 17, and Figure 18. The three figures describe three aspects of any feature construction system: the feature construction algorithm, the features, and the feature constructor functions. The features and the constructor functions serve as the main inputs to the construction algorithm. Some elements of the architecture are not used by ISAK. These elements are shown to emphasize the relationship between the knowledge ISAK uses and what other systems use for feature construction. Further examples of the top-level elements are given in Section 4.2 and Chapter 6. Many of the following elements are implemented directly in ISAK. Those elements that are not implementated are either (1) indications of future and/or related work or (2) useful for discussion of ISAK and other feature construction systems.

4.1 CHARACTERISTICS OF FEATURES, ALGORITHMS, AND CONSTRUCTORS USED IN FEATURE CONSTRUCTION

4.1.1 Sources of Knowledge from Features

Feature description can be sub-divided into two categories (Figure 16): description of a single feature in isolation and description of interactions among features. Individual features are composed of an attribute and value. The attribute component of a feature may have specific semantics:

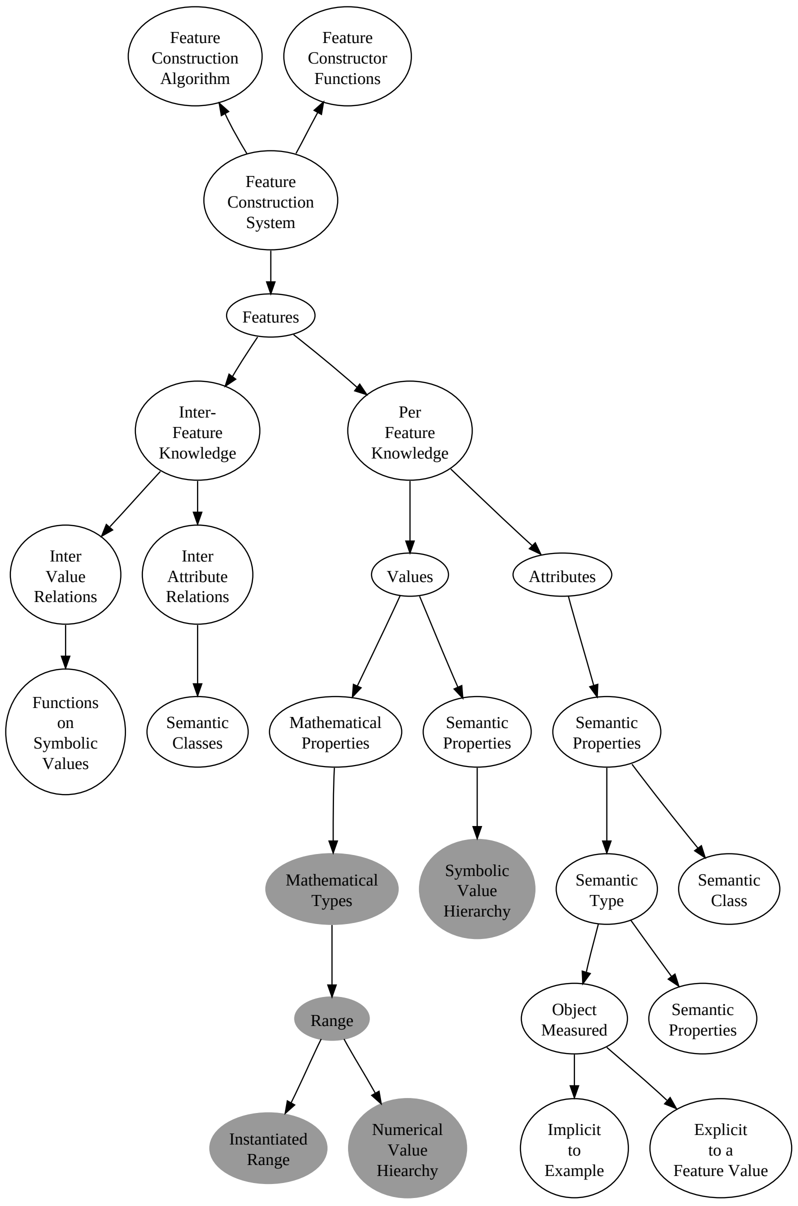


Figure 16: Sources of Semantic Feature Construction Knowledge from Features. The nodes in the tree show different sources of knowledge that may be used in a feature construction system. The nodes that are greyed out are not implemented in ISAK.

the type of the attribute[1], properties of that type including the object[2] it describes, and a semantic class. The semantic class of an attribute is a loose grouping of related attributes. The object that an attribute describes may be described explicitly, as the value of some other feature, or implicitly, as the underlying object that some or all of the attributes measure. It is important to note than an *object is not an example.* One example may contain measurements on two or more objects in the world from which the example is drawn. The attributes are the measurements on these objects (see Section 4.2.1). For instance, in an attribute-value description of two-vehicle car accidents, one example *of an accident* will have attributes describing two objects: the first car and the second car. The measurements might be the weight of the first car, the weight of the second car, the speed of the first car, and the speed of the second car.

In the wine CLP, all of the measurements are taken over one object, a sample of wine. Thus, there is no need to specify particular objects in the wine CLP. However, each of the measurements belong to semantic classes (Figure 7) which have some loose relationships based on what the attributes measure. Because the measurement types (e.g., whether or not the measurements are ratios or absolute quantities) of the wine attributes are unknown, each of the features is of type *GenericQuantity*. *GenericQuantity* represents that the measurements are quantifications of some property but that the specifics are unknown.

The values that an attribute takes over the examples in a dataset are described by their mathematical and semantic properties. The mathematical properties of a value include the legal range of values for that type and, also, the legal operations defined on those values. For example, arithmetic operations are defined on integers and real numbers. Set operations are defined on sets. Boolean logical operations are defined on truth values. The mathematical type refers to the formal properties of the value. A numeric type restricts the range of values; further restrictions can be useful in restricting feature construction. Inclusion of zero in a value range precludes the use of division, as it is normally defined, on the corresponding feature. A symbolic value hierarchy defines the values for a symbolic features, as well as the relationships among the values.

Inter-feature knowledge can be subdivided into (1) knowledge of relationships among values of different features and (2) knowledge of the relationships among attributes. The former is generally

[1]Also, see Figure 22.

[2]The term *object* may refer to an object or a reified process.

unnecessary – features with incompatible values are typically not combined. However, in the case of symbolically defined values, operations that account for the semantics of those values may be defined. Since these operations must account for the semantics of the values of different features, the operations represent inter-feature knowledge. In TTT, constructions of features in similar positions are most important. In the second case, there typically *are* relationships among the attributes in a dataset. These relationships can be expressed as semantic classes of features for which some property holds. Such relationships include measurement of similar objects, groups of objects, or properties of objects (e.g., same object, different but related property; same property, different but related object). The wine CLP defines relationships among the features on the basis of what those features measure.

The general ontology of types that ISAK draws on is described in Section 4.3. ISAK's built-in semantic types and their semantic properties are discussed in Section 6.2.1. ISAK's use of semantic classes is described in Section 6.2.2. Functions on symbolic values may be specified by feature constructor functions (see Section 6.2.5). ISAK does not make use of symbolic value hierarchies; these are described by Aronis and Provost (1994).

4.1.2 Knowledge Use in Feature Constructor Functions

Feature constructor functions (Figure 17) apply computationally defined operations to tuples of features and produce one or more newly constructed features. The feature constructor functions (FCFs) are defined by their mathematical and semantic properties. The mathematical, or formal, properties of a FCF include a computable definition of the operation it performs on its arguments, the semantic types of its arguments, constraints on the arguments and their types, and the semantic type and properties of the results of the construction. The resulting features are also described by their mathematical type and the range of values they take.

For example, in the wine problem (Chapter 2.1), the feature constructor functions are designed around the four standard arithmetic operators. The computable definition of each FCF is its respective arithmetic operator. The semantic type of the result is defined as a *GenericQuantity*. The semantics properties of the arguments are constrained by unification and the output result is that unification.

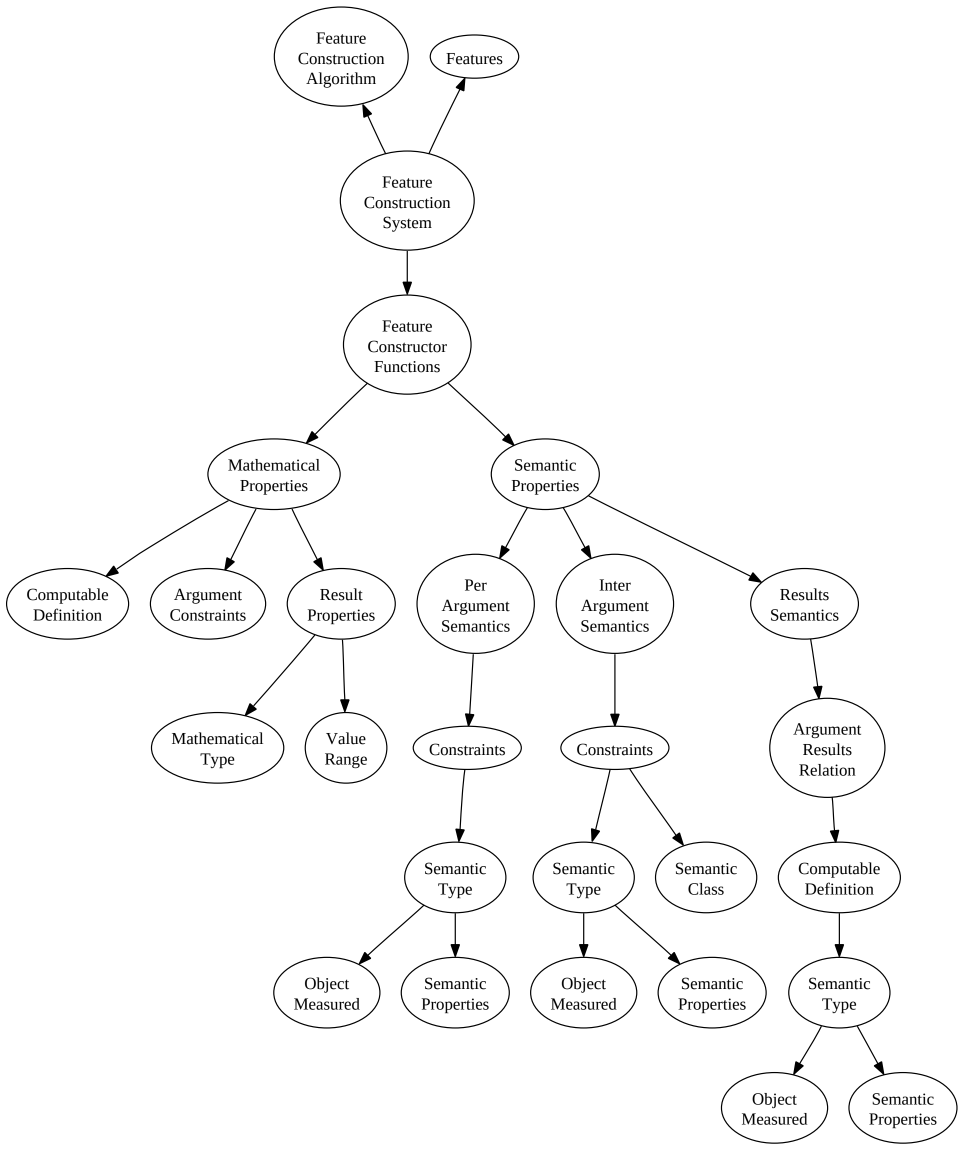


Figure 17: Semantic Feature Construction Knowledge and Feature Constructor Functions. Each node in the graph represents a knowledge source used by semantic feature constructor functions.

The semantics of constructors are specified by individual argument semantics, inter-argument semantics, and semantics of the result. Individual arguments to a constructor are specified by constraints on their semantic types and the semantic properties of those types. Constraints among arguments are specified by the semantic types, properties, and classes of the arguments. The semantics of the resulting features are a function of the input semantics. Hence, a computable definition is needed that takes the input semantics, respects the operation performed, and gives the appropriate output semantics (i.e., a semantic type and semantic properties). The generation of these pieces of semantic information mirrors the generation of new values to fill in new attributes.

The built-in feature constructor functions are described in Section 6.2.5.

4.1.3 Input and Processing of Feature Construction Algorithms

A feature construction algorithm operates by applying feature constructor functions to groups of features. Its operation is further specified by some additional inputs and choice of processing method. The additional inputs are (1) the original, unmodified dataset which is dependent upon the data description language used and (2) results from the induction process. Induction results come from the learning algorithm (e.g., performance of the learner) and from the learner applied to data (i.e., a classifier and performance of the classifier). The learner considers hypotheses and is dependent upon the hypothesis description language; the order in which hypotheses are considered is the algorithmic component of the learner. Classifiers are differentiated by their complexity and the results of applying them to data for classification. The classification results can be broken down into generalization results and resource usage. ISAK makes use of the input data in feature construction; it does not use any of the other characteristics of data and learners. The characteristics unused by ISAK are typically used by hypothesis-driven and results-driven feature construction systems (see Section 3.3 and Chapter 5).

We can consider three different, yet overlapping, methods of processing in a feature construction algorithm: search, inference, and agenda-based processing. The later two may be built on top of search methods. Search itself is defined by a search method, a successor function on the search space, and heuristics on the search space. Search methods include breadth-first, depth-first, and beam search. The heuristics take the form of ordering and pruning constraints which, in turn, are

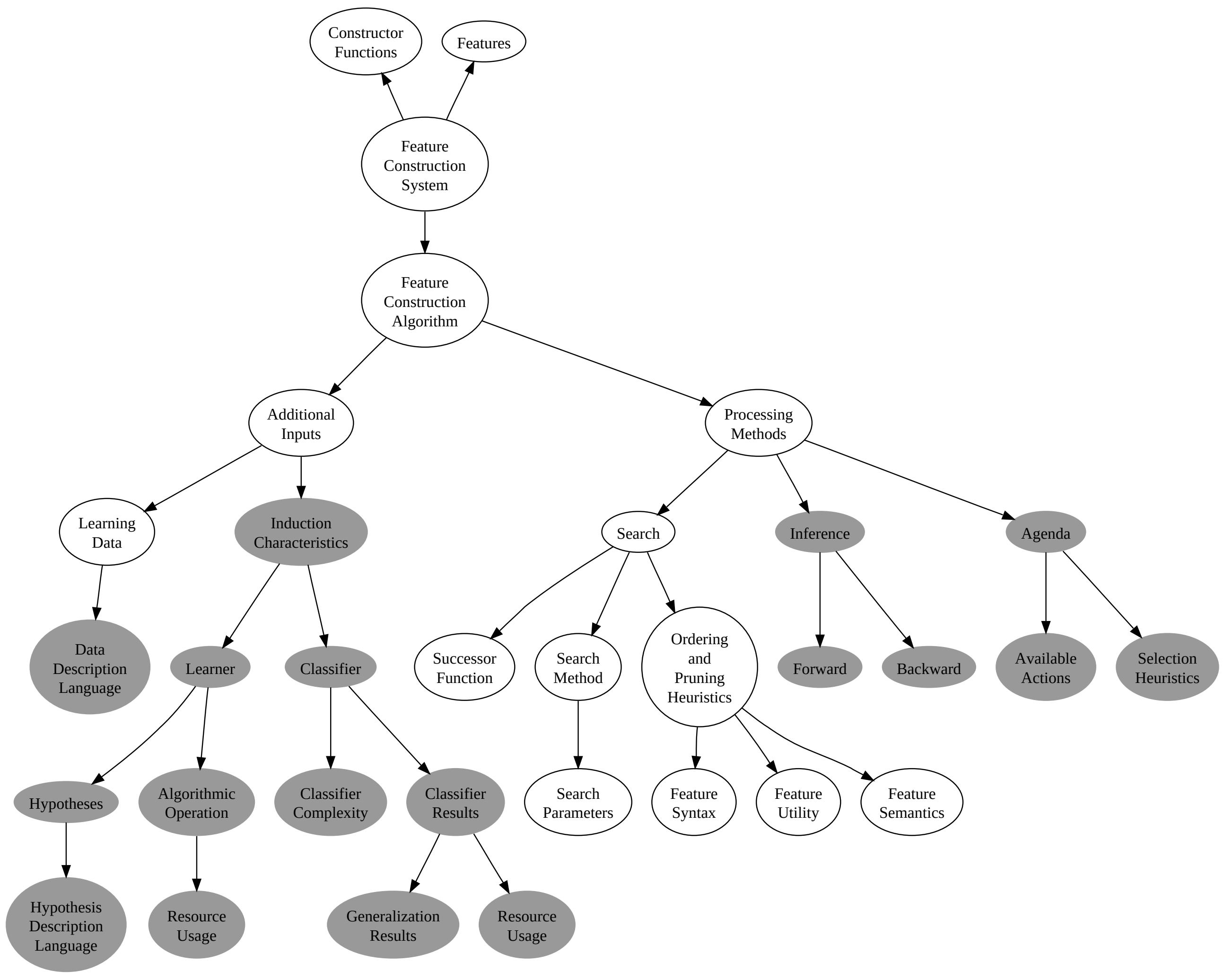


Figure 18: Additional Specifications for a Feature Construction System. Each of the nodes in the graph can affect feature construction systems. Several aspects are relevant to other feature construction methods (i.e., feature construction that is not knowledge-guided). Nodes which are grayed out are not implemented in ISAK. Feature semantics are used as a pruning method and are implemented by restricting the application of FCFs in the successor function.

dependent on the characteristics of the nodes in the search space. The nodes represent features in the feature space. Hence, the constraints are characteristics of features: form, semantics, and utility. Search, in particular beam search, is the processing method used by ISAK (see Section 6.3.1).

Other processing methods include inference via forward and backward chaining. An agenda can be defined in terms of available steps, possibly similar to successor generation in search, and in terms of selection criteria for those steps. Seen as an adjunct to the search process, an agenda system can inject larger step sizes (i.e., it can apply several single-step successor functions at once) and finer-grained control of successor generation into the movement through a search space.

4.2 A KNOWLEDGE LEVEL DESCRIPTION OF ISAK

The three major semantic descriptors available in ISAK are semantic types, semantic properties, and semantic classes. A semantic type defines a class of measurements; it has a set of semantic properties that provide the details of a measurement's semantics. The most broadly used semantic detail is the object to which the measurement refers. Other semantic properties are associated with specific semantic types. For example, dimension is a semantic property of the spatial extent of an object. Semantic classes of measurements group related subsets of features and naturally constrain combinations of features.

The semantic type and properties are combined with feature constructor functions by a feature construction algorithm to produce new features. New features require values and semantics that are computed from old features. New values are needed to use the features in the primary learning task; new semantics are required to use the new features in subsequent rounds of semantic feature construction.

4.2.1 Identification of Objects in the Learning Domain

The values in an attribute-value data representation are statements about the value of a characteristic of some object that exists in the world under consideration. The objects may come from the physical reality that surrounds us, a constructed domain, or both. In either case, an example in a dataset groups together measurements from that world which are presumed to be useful in the target prediction. However, the attributes of a single example may refer to multiple different objects in the world[3]. The fact that measurements on the objects are used for prediction is the reason they are joined in a single learning example. An example may be a combination of measures on multiple, individual objects.

Specifying these objects explicitly is important because many of the statements made about feature types and formula constraints refer to the underlying objects in the CLP. By making explicit the particular objects that underlie an example, we can capture distinctions among attributes describing a single object, attributes describing multiple objects and their relationships, and attributes describing (generic) relationships between and among objects. Typical attribute-value (propositional) representations of learning problems do not make explicit mention of the objects measured. In the wine domain, the fact that the measurements are made on a sample of wine is not mentioned anywhere in the learning data. One instance where the underlying objects are exposed to attribute-value learners is in propositionalization[4] of relational learning tasks (see Section 5.1.1).

A simple example of a dataset where the attributes refer directly to the object under consideration is the classification of triangles as *isosceles* or *not isosceles* given the lengths of the sides of a triangle. A logical statement, equivalent to one of the features in the isosceles problem, is $length(implictTriangle, firstSide) = 27.5$. Here, the logical statement is made in reference to the *example itself* – that is, the triangle that is implicit to the example. The logical statement can be broken down into the statements that (1) there exists a triangle *e*, (2) each of the *side* attributes is a statement about *e*, (3) the triangle *e* and the current example, (i.e., *this* example) are one and the same, (4) the functional relation *length* has the value of *27.5* for the object *implicitTriangle* and part *firstSide*.

A contrasting scenario is the problem of learning whether or not a balance-type scale is balanced based on the weight and distance of objects on either end of a pivot point (Table 9). In this case, one example in the dataset refers to a scenario involving two different objects and their relation to a third object. Specifically, the point-masses on either end of a balance and the relation-

[3]Again, think of examples of car accidents that involve at least two objects, car_1 and car_2.

[4]*Propositionalization* is the conversion of a relational dataset to a propositional (i.e., an attribute-value) dataset.

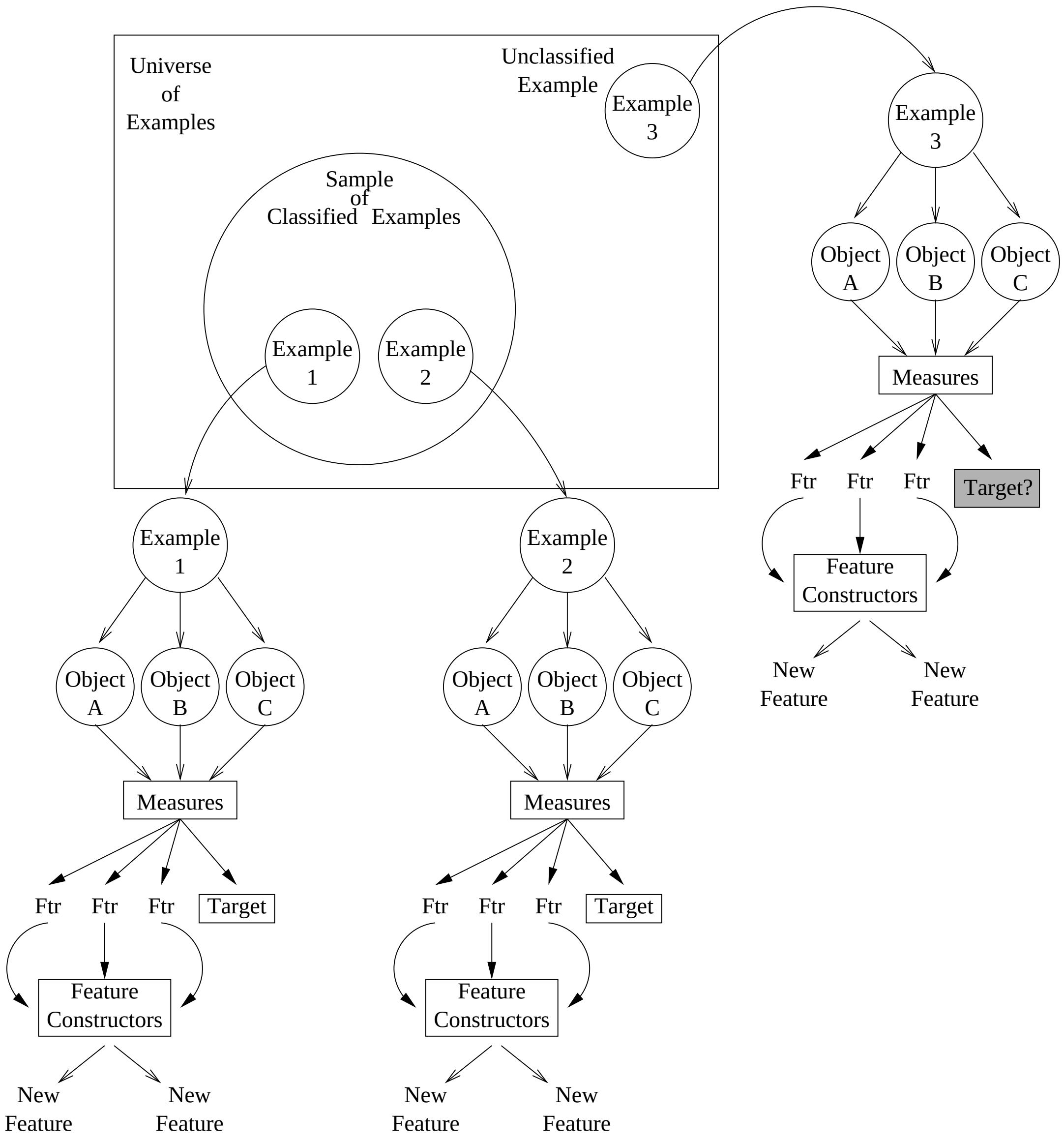


Figure 19: Abstract View of Feature Construction as Operations on Objects. Measures on objects in the world of the example are the features in a learning problem. Those features are modified with feature constructors; new features may describe the original objects or relationships among them. Likewise, the *Target* concept may be a property of a single object or a relation on multiple objects. The knowledge that a feature pertains to a particular object or group of objects is an important source of restriction in the feature space. The *Target* value for *Example 3* is highlighted in gray, because it must be predicted. *Example 3* is drawn from the set of *Unclassified Examples*.

Table 9: Balance Dataset. The Balance dataset from the UCI repository. *LeftWeight* and *RightWeight* indicate a mass at a distance of *LeftDistance* and *RightDistance* respectively from the central fulcrum. There are three classes for the problem, *B*, *R,* and *L*, indicating that the scale is balanced, leaning right, or leaning left.

LeftWeight	LeftDistance	RightWeight	RightDistance	Lean
c	c	c	c	d
				class
1	1	1	1	B
1	1	1	2	R
1	1	1	3	R
1	1	1	4	R
1	1	1	5	R
1	1	2	1	R
1	1	2	2	R
1	1	2	3	R
1	1	2	4	R
1	1	2	5	R

ship of the masses to the pivot point are the determining factors in the classification. The example is better understood as a relationship among several individual objects than as a single, complex entity[5] (Figure 20). Here, there are three objects: *leftWeight, rightWeight*, and *pivot*. The features *LeftWeight* and *RightWeight* are weight types that refer to the left and right weights, respectively. *LeftDistance* refers to a spatial separation between the point of attachment of left weight and the pivot; *RightDistance* is analogous. The distinction of different objects allows combinations of features with like referents (i.e., measurements on the left or right weight) and can disallow combinations of features with different referents. When the weight and distance are combined – producing a torque-like quantity – the similar results can be compared.

As a final example of the universe of objects referenced in a dataset, objects themselves may be the value of a feature. Direct reference to the objects *and* the relationships in a learning problem is typical in ILP datasets (shown abstractly in Figure 21). These relationships are exposed in propositionalization of relational learning problems (Lavrac and Flach, 2001).

[5]A single complex entity might encapsulate the entire scenario as the objects of interest. So, we might have *weight(implicitScenario, leftW)* and *length(implicitScenario, leftW)* instead of *length(leftW)*.

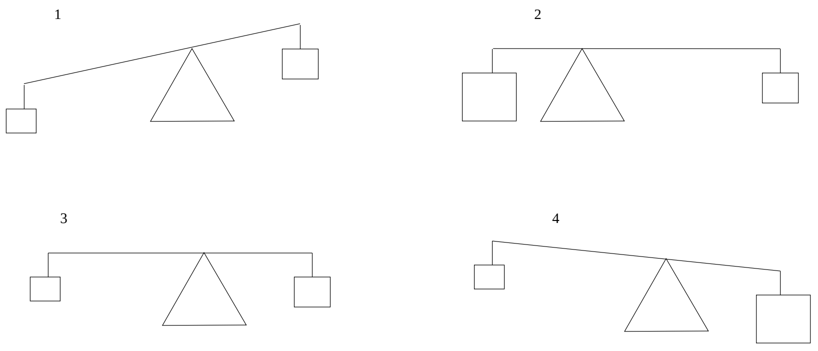


Figure 20: Four Examples from the Balance Universe. The universe of the balance examples has three objects of interest: two weights, right and left, and the pivot. The measurements in the dataset (Table 9) are the masses of the weights and the distances between each weight and the pivot.

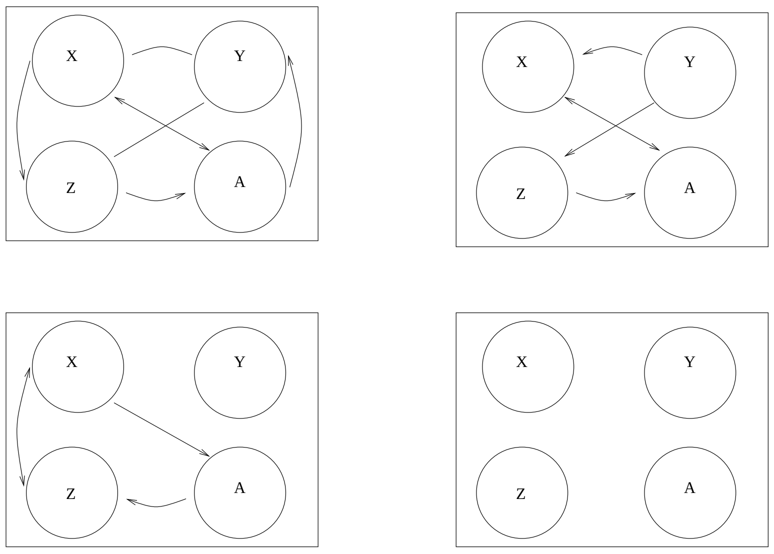


Figure 21: Four Examples of Abstract Relations. In the propositional form of a relational learning problem, the objects and relations among objects are directly exposed and used for learning. In this diagram we have abstract entities (in circles) and relationships (directed arrows). The relationships may be exposed for learning in an attribute-value representation by expanding the relationship into a table with an entry for each element of the cross-product of objects in the learning world (see Section 5.1).

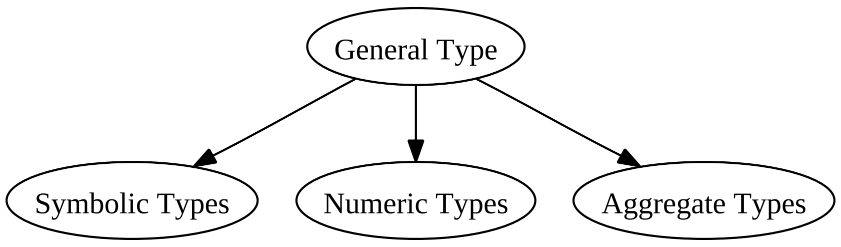


Figure 22: Top Level Type Ontology. A hierarchy of types for knowledge about features. The ontology is continued under Symbolic Types (Figure 23), Numeric Types (Figure 24) and Aggregate Types (Figure 25).

All learning problems allow alternate representations and these representations can affect the specification of objects in the learning domain. For example, game boards may be described as a set of location attributes with piece values. Alternatively, a board might be described as a set of piece attributes with location values. Objects are central descriptors in the knowledge that describes the world from which the instance space is drawn: objects composing the examples, measurements of objects, and relationships among objects are important sources of semantics in learning from examples.

4.3 AN ONTOLOGY FOR SEMANTIC TYPES

A measurement on an object in the learning domain has a semantic type. An ontological hierarchy of types is shown in Figures 22, 23, 24, and 25.

The point of the hierarchy is to provide a foundation on which a discussion and implementation of feature typing can be undertaken conveniently. As such, there is more than one way to describe certain feature semantics and there are concepts that are best addressed by adding to the hierarchy. The hierarchy will be useful if the additions can be easily integrated. The general form of the ontology is motivated by SUMO (Niles and Pease, 2001). The ontology shown here abstracts

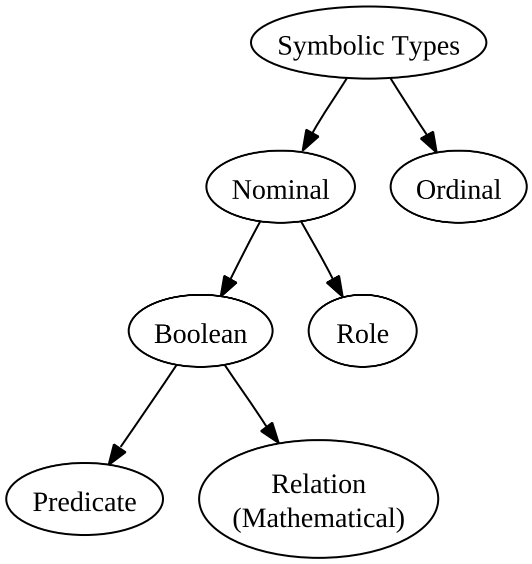


Figure 23: Ontology for Symbolic Types. *Relation* refers specifically to a mathematical relation, namely the subset of a cross-product between two sets. *Role* include specific pieces on a game board and positions within an organization.

away many details of the SUMO ontology and retains only the levels that are most important for feature construction by ISAK.

The uppermost level of typing is closely related to the syntax of the features. Aggregate types have compound representation similar to mathematical sets and vectors. Usually, numeric and symbolic types are distinguished by their data values. One exception is when a numeric valued feature has few values and these values are interpreted as symbolic values. In the exception, each number is considered as a unique symbol and the mathematical relationships between these values are discarded. For example, a dataset might have a coding with *Female* coded as "1" and *Male* coded as "2".

Ordinal values have uncertainty in their placement in the hierarchy because they only make use of the ordering and distinguishing properties of integers. So, ordinals may be considered as a symbolic value with the additional property of being ordered or as a numeric value without a fixed distance metric between values. Given the type hierarchy's increase in specificity towards

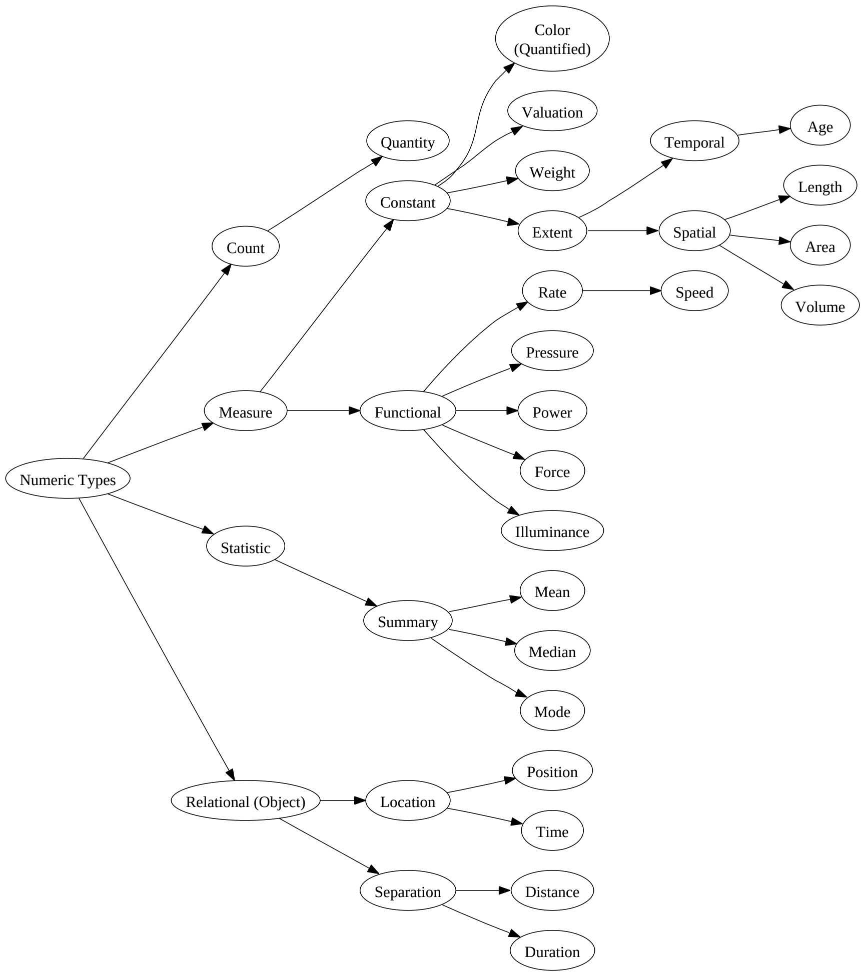


Figure 24: Ontology for Numeric Types. *Relational* types in this hierarchy refer to measurements between distinct individuals. *Locations* are made with respect to an absolute coordinate. *Separations* are made with respect to another arbitrary object. *Color* refers specifically to colors as quantified by a wavelength or other quantified scale (i.e., not as a symbolic value). *Statistics* are scalar properties of aggregations.

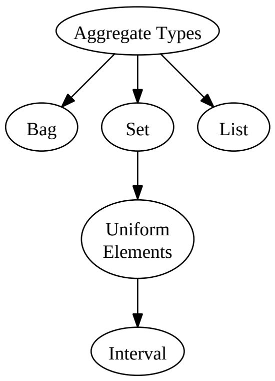


Figure 25: Ontology for Aggregate Types. Example of the aggregate types are shown in Table 10.

the leaves, placing ordinals under symbols follows the general rule. As a practical matter, ordinal values do not figure prominently in the following examples and experiments.

There are three aggregate types: sets, bags, and lists. Sets are the familiar mathematical objects – they are composed of unique elements without respect to order. Bags are sometimes referred to as multi-sets; they are unordered collections of elements with potential duplication of those elements. Lists are a standard programming construct and they enforce ordering but not uniqueness. Bags may be preferred to sets because sets discard duplicates in their construction. For example, if we construct a set of features and a bag of features (from the same set of initial features), then the values of the bag and set will be different on any example that has duplicates among the values of its initial features (Table 10). The difference is troublesome if we want to count all of the items – without respecting uniqueness – in a collection. Of course, the difference is useful if we want to see the uniqueness or duplication within an aggregate of features' values.

Nominal types allow specification of logical relationships in a variety of forms. Because these values are at a logical level of description, nominal types will generally carry a high degree of domain or problem-specific information in the attribute and in the values of that type. For example, a symbolic variable with two values, *black* and *white,* can be interpreted as a Boolean variable. Specifically, the non-semantic information content in the values of *color* (with values *black* and *white*) and in *color=black* (with values *True* and *False*) is the same. However, domain knowledge can distinguish whether those values are colors on a board where only the difference is important,

Table 10: Differences among Aggregate Types. In the case where features have duplicated values (i.e., *Ftr1* and *Ftr2* in the first line), a bag and a set constructed from these features are different – the bag maintains the duplication; the set removes it. The difference is also evident in the third line. Bags and sets discard the order of features in their construction. Lists maintain the order of features.

Ftr_1	Ftr_2	Ftr_3	Set(Ftr_1, Ftr_2Ftr3)	Bag(Ftr_1, Ftr_2Ftr3)	List(Ftr_1, Ftr_2Ftr3)
a	a	b	{a,b}	{a,a,b}	[a,a,b]
a	b	c	{a,b,c}	{a,b,c}	[a,b,c]
a	c	c	{a,c}	{a,c,c}	[a,c,c]
c	b	a	{a,b,c}	{a,b,c}	[c,b,a]

colors on a screen which may be combined as gray-scale values (e.g., associated with ranges of numeric values), or ethnicity with a wide variety of socio-economic concerns. So, we need to know *how* a value measures a given object, in addition to *what* the object is.

In contrast, numeric types such as measurements tend to be built on top of a general system of measurements (such as the International System of Units, SI) and carry standard relationships to other such measurements. There is less need for problem specific interpretation of values in this case. What is still needed is knowledge of *what* these values are measuring – what objects and what characteristics of those objects. A length can measure one of several dimensions of an object; a time can measure the duration of different processes in a single construction and learning problem. The functional branch of measures distinguishes those quantities that map from one constant quantity to another. Constant quantities include the base SI units and properties like monetary value.

4.3.1 Semantic Classes

Multiple features may refer to groups of objects or measurements that are related. I term such a set of features a *semantic class* of those features. The statement that a number of features belong to a semantic class means that these features share some commonality. For example, two semantic classes are:

1. measures of different aspects of a single object: {systolicBP(Mark), disystolicBP(Mark)} is a semantic class of different blood pressures for one patient, and
2. measures of the same characteristic of different objects: {height(Jeff), height(Bill), height(Mary)} is a semantic class of heights for several different players.

Semantic classes may be applied independently of each other. However, relationships among semantic classes can be captured in a *semantic class hierarchy*.

An important operation on the semantic class hierarchy is unifying classes in the hierarchy. The hierarchy of semantic classes is a partial ordering among nodes representing the semantic classes. Unification on partially ordered sets involves computing the least upper bound for any two semantic classes. In terms of semantic classes, unification involves finding the least general class that is a superclass of all the classes to be unified. By restricting operations to class types that unify within a hierarchy, we can constrain possible legal combinations of features.

4.3.2 Feature Constructor Functions

With the world objects, semantic types, and semantic classes of features, we specify semantics of individual features and describe relationships among the features. We must now specify how the descriptive knowledge is utilized to create new features from existing features. The unit that encapsulates this knowledge is called a *feature constructor function* (FCF). The feature constructor functions used by ISAK are described in Section 6.2.5.

A feature constructor function is specified by three parts: (1) a description of when the FCF is applicable, (2) a computable operation that produces new feature values from the values of existing features, and (3) a computable semantic description of the resulting feature. These parts are described in the feature constructor functions for the wine CLP (Figure 8).

4.3.2.1 Constraints on the Search Space by the Available FCFs The applicability of a FCF in ISAK is based on the semantic elements of its features. Although it is not implemented as such, the test for applicability can be thought of as testing satisfaction of a system of constraints or as pattern matching for determining whether or not to apply a production rule. The applicability of an entire set of feature constructor functions determines the *plausible* constructions defined by the system. The set of constructors creates a hard boundary within the space of all *possible* constructed features. Constraining the plausible features to a set of useful features is left to the search space pruning done by the beam search mechanisms (see Section 6.3.1).

Theoretically, for a set of numerical features, there are a uncountably many features that may be constructed: simply taking all linear combinations of the features with real-valued coefficients results in an uncountable set of new features. A strong restriction on the number of combinations is restricting the constructor functions to easily interpreted arithmetic functions of one and two features: inverse, reciprocal, doubling, halving, square root, square, addition, subtraction, multiplication, and division. This reduces the space of plausible features to countably many features and enforces a certain degree of simplicity in the constructed features. It also limits the immediate interactions among features to pairs.

4.3.2.2 Object Constraints Object constraints, the most important class of semantic constraints, distinguish between attributes that describe the same object and attributes that describe different objects. Object constraints are enforced by comparing the object specified in the semantic class of a feature.

Within-object feature construction results in further descriptors of that same object. Body mass index, computed from height and mass, describes the body composition of a person based on easily obtained measurements of that person. Various financial ratios, such as price-to-earnings, redescribe a company and its stock value based on attributes of that business. Many combinations of physical measurements over a single object are possible. For example, packages delivered via the United Parcel Service are considered large based on the (1) the maximum length of any side of the package and (2) the girth of the package. The girth is defined as $2width + 2height$ where width and height are the two non-maximal sides of the package. A package is considered *large* when the $length + girth > 130$. The distinguishing aspect of within-object combinations is that the new feature is also an attribute of that same object.

Between two different objects, features of a similar type may be combined. For example, combining the heights of two boxes (creating a difference or a sum) is appealing. Combining a

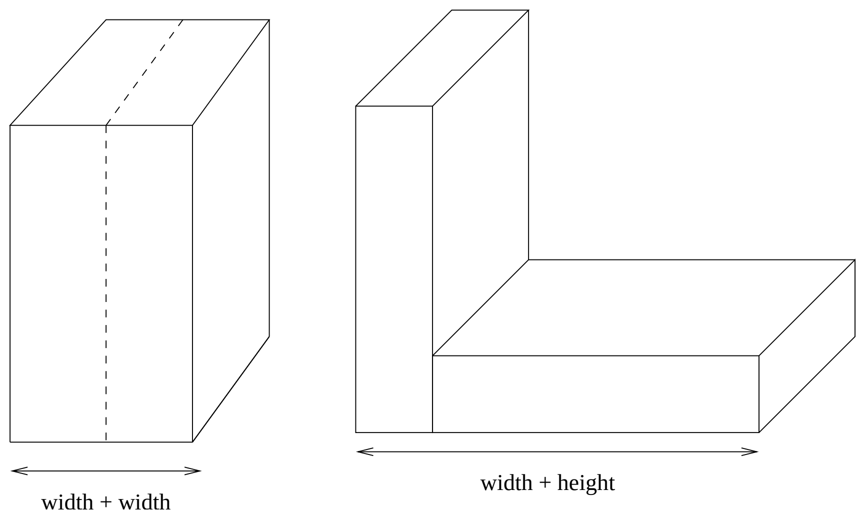


Figure 26: Tipped Boxes. Semantic descriptions must be set to represent the world in which the objects reside.

height and a width is only logical in specific instances – for example, if one box has been overturned and the target function depends on the total width of the two boxes (Figure 26). Here, even though we are now considering a height and a width we are still concerned with two extents. In the absence of specific information, it makes little sense to combine a height of an object with the weight of a different object and even less sense to combine the color wavelength of an object with the weight of a different object.

Certain inter-object relationships can be specified precisely when the objects are known to be involved in a particular event. For example, if one object is resting on another, there is a pressure exerted between the two objects and its value is computed from the surface area of contact and the force between the two objects. The semantics of the new relationship may be recorded as being between the two objects or as an aspect of a new object (an event object) that is itself the relationship between the objects[6].

[6]Similar in character to ILP.

4.3.2.3 Constraints from Semantic Types Types of individual features give strong guidance to the manner in which they may be combined. A number of combinations can be justified on the basis of functional measurements (measurements that map one measure to another). For example, speed is a mapping between distance and time. With any two of these we may compute the third. Other combinations are justified by the definition of an operation. The area of planar objects is defined for different shapes in terms of linear measurements. Both the operations arising from functional measurements and from specific definitions are examples of within-object combinations. The pressure example above is dependent on the types of the features present (forces[7] and areas) and new features between distinct objects are constructed in that example.

Aggregate types – lists, bags, and sets – require specific operations that combine, test, and modify values of that type. Lists and bags may be tested for occurrence-counts and for uniqueness of elements in that aggregation. Lists support ordering predicates. All three aggregate types support element matching predicates (e.g., aggregate *A* contains value *e*) and counting functions. Matching predicates may be against a constant value or against a value taken from another feature.

4.3.2.4 Effects of Constraints on the Number of Constructable Features Imagine a learning problem where the features are the mass, width, and depth of ten objects; let the target concept be closely related to the concept of pressure. What is the effect of semantic type signatures and within/between object constraints (Table 11)? To compute pressure, we need a surface area of contact and a force. To simplify the problem, define the surface area of contact between two objects to be the minimum surface area of either object and define the force exerted by an object to simply be its mass (a reasonable assumption if all the objects are at rest and in the same frame of reference). The surface area is the product of the width and depth.

If the operations are known, the feature construction problem becomes one of filling in the argument slots $\{width_{1,2}, depth_{1,2}, mass\}$ in

[7]With a direction or with respect to a canonical direction.

$$\begin{aligned} div(contactArea, mass) &= \\ div(min(area_1, area_2), mass) &= \\ div(min(mul(width_1, depth_1), mul(width_2, depth_2)), mass) & \end{aligned}$$

In each of the following, reuse of features is forbidden. With no constraints on semantic type and object, there are

$$\binom{30}{1}\binom{29}{1}\binom{28}{1}\binom{27}{1}\binom{26}{1} = 657720$$

possible combinations of features.

With constraints on both type and object (e.g., $Width_1$, $Depth_1$, $Width_2$, $Depth_2$, $Mass_1$), there are

$$\binom{10}{1}\binom{1}{1}\binom{9}{1}\binom{1}{1}\binom{2}{1} = 180$$

plausible features.

With constraints on types but not on objects (e.g., $Width_1$, $Depth_3$, $Width_4$, $Depth_8$, $Mass_2$), there are:

$$\binom{10}{1}\binom{10}{1}\binom{9}{1}\binom{9}{1}\binom{10}{1} = 100 * 81 * 10 = 81000$$

plausible features.

With constraints on the objects but not on types (e.g., $Width_1$, $Depth_3$, $Width_4$, $Depth_8$, $Mass_2$), there are:

$$\binom{10}{1}\binom{3}{2}\binom{9}{1}\binom{3}{2}\binom{2}{1}\binom{3}{1} = 10 * 3 * 9 * 3 * 2 * 3 = 810 * 6 = 4860$$

plausible features.

If there are few measurements on each object and many objects, within-object limitations are very powerful constraints. If there are few objects and many measurements on each object, between-object limitations are very powerful. If there are many objects and many measurements on each object, a large number of constraints are necessary to reduce the search space to a reasonable size.

4.3.2.5 Constraints by Semantic Class Semantic class specifications justify and prohibit other combinations. For example, adding features that represent counts can be constrained with respect to a given semantic class hierarchy. Adding apples and oranges has a simple interpretation with an appropriate semantic class hierarchy – a total number of fruit. However, adding atoms and cars (which would occupy very different heights in a class hierarchy) will result in a total number of physical entities. Hence, a constructor allowing such combinations is inappropriate. A more reasonable example of combination would be combining different types of imported and domestic cars with the result in the general class of all cars.

Class specifications may also be used to define a limit on the contribution from different classes. In a demographic dataset, if several features measure the amount of wealth of an individual (e.g., income, savings, number of cars, size of house), we may only need one of those features. More generally, a single feature may adequately capture the necessary information from its semantic class. If this is the case, replacing multiple features with a single feature will reduce the redundancy in the dataset.

4.3.2.6 Resulting Values and Semantics The computable operation is a computable function that takes input feature values from existing features and outputs feature values for one or more new features. The operation defines the values of the result.

The semantic description of the resulting feature(s) is based on the semantics of the input features and the operation performed on them. The semantic computation defines the semantics, specifically the type and typing details, of the result.

4.3.3 Formal Description of Features and Feature Constructor Functions

Let a feature be a set containing a tuple of values, a semantic type, and semantic properties, $f = \{V, t, p\}$. Let $Values(f)$, $Type(f)$ and $Prop(f)$ return *V, t,* and *p*, respectively.

An ordered tuple of features $F = (f_1, f_2, ..., f_{nf})$ satisfies a type specification $T = (t_1, t_2, ..., t_{nt})$, if $nf = nt$ and $\forall_i Type(F_i) \subseteq T_i$. For two types T_1 and T_2, $T_1 \subseteq T_2$, if $T_1 = T_2$ or T_2 is an ancestor of T_1in the type hierarchy.

The semantic properties[8] $P = (p_1, p_2, ..., p_{np})$ of the features *F* satisfy the constraint predicate c, if $c(P) = True$. Likewise, the features *F* satisfies a feature constraint c, if $c(F) = True$. The values *v* of a new feature f_{new} constructed from a set of values V of features with a computable function $\mathcal{F}_V$ is a vector of values $v = \mathcal{F}_V(V)$.

The semantic type and properties t_{new}, p_{new} of a newly feature constructed are computed from:

1. a tuple of values V,
2. a tuple of types T,
3. a given semantic type t_{new}, and
4. a computable function $\mathcal{F}_P$ of the values, types, and semantic properties.

The new semantics are of type $t_{new}$with semantic properties $p_{new} = \mathcal{F}_p(V, T)$.

A feature constructor function is defined by a tuple of semantic types $T = (t_1, ..., t_A)$, a set of semantic constraints $C_S = \{c_1, ..., c_{nsc}\}$, a set of feature constraints $C_F = \{c_1, ..., c_{nfc}\}$, a computable function for values $\mathcal{F}_V$, an output type t_{new}, and a computable function $\mathcal{F}_P$ for semantic properties of the output type.

A feature constructor function $\{T, C_S, C_F, \mathcal{F}_V, t_{new}, \mathcal{F}_p\}$ is applied to create a new feature $f_{new} = \{\mathcal{F}_V(Values(F)), t_{new}, \mathcal{F}_P(Values(F), Types(F))\}$ from a tuple of features F, if $Types(F)$ satisfy T and $\forall_{c \in C_S} c(Types(F)) = True$ and $\forall_{c \in C_F} c(F) = True$.

4.4 CONSTRUCTED FEATURES AS TREES

Both constructed features and known target concepts that are defined in terms of base features can be defined as trees where:

1. the tree root is a constructed feature,

[8]For simplicity sake, object specification is considered one of the semantic properties. It is separated out in the description of knowledge because of its important role in semantic feature construction.

2. the interior nodes of the tree are (partial) constructed features,
3. the tree (directed) edges represent the flow of values from leaves towards the root from the leaves (note, we pass values "up" towards the root), and
4. the leaf nodes represent base features (or constructed features taken as base features).

Interpreting the constructed features as trees allows us to measure several aspects of the complexity of the constructed features. Importantly, the height of the tree is the maximum number of operations performed on any path from the base features to the constructed features, the number of leaves on the tree is the number of base features used to construct this feature, and the number of unique leaves is the number of unique features used to construct this particular feature. An example is shown in Figure 27.

The value of a leaf node is defined as the vector of values of that base feature over each example in the dataset. The value of a non-leaf node *N* is defined recursively as the result of applying the feature constructor at *N* in a vector-wise fashion to the values of *N*'s children.

Two constructed features, *T* and *T'* are equivalent, if the value of the root of *T* is equal to the value at the root of *T'*.

F is a partially constructed feature for a constructed feature *T,* if *F* is a subtree of *T* and the leaves of *F* are leaves of *T*.

The distance from a set of features to a target concept or constructed feature given a set of operations, $d(\{ftrs\}, T|\{constructors\})$, is the minimum number of non-leaf nodes in any *T'* whose values at the root of *T'* are equal to the values at the root of *T* where the internal nodes of *T'* are elements of $\{constructors\}$ and the leaves of *T'* are elements of $\{ftrs\}$. For example, $d(\{a, b, c\}, ab/c|\{*, /\}) = 2$.

A set of constructors, O is compressed by a single operator, o', when a tree can be constructed from O that has the same values at the root as o'. O is not unusable; other orders of operations may not be compressed to o'. A set of operators S_1 is compressed by another set of operators S_2 when a subset of S_1 is compressed by a subset of S_2. For example, the set of constructors $\{mul(.,.), div(.,2)\}$ is compressed by $average(.,.)$.

The gap between two sets of constructors C_1 and C_2, where $C_1 \subseteq C_2$ and $\forall o_1 \in C_1, o_2 \in 2^{C_2} \neg compresses(o_2, o_1)$ (i.e, no subset of C_2 compress any of C_1) with respect to a set of features *F* and a target concept/feature *T* is

$$\operatorname*{argmax}_{P} d(F,T|C_2) - d(P,T|C_2).$$

The idea here is that C_1 only leads to a partial concept of C_2 and we want to know how close C_1 gets us to C_2.

Two feature trees, T_1 and T_2 are equivalent at their top-level if $T_1 \cap T_2 \neq \emptyset$ and $x \in T_1 \cap T_2 \implies parent(x) \in T_1 \cap T_2$. That is, there is a path from every element of the intersection back to the root (note that $parent(x)$ of the root is empty).

If T_1 and T_2 are top-level equivalent and $x \notin T_1 \cap T_2$, then x is in the bottom-difference of the trees.

Table 11: Various Constraints for FCFs in the Pressure Problem. For each FCF (Area, Contact Area, and Pressure) there are several combinations of constraints from the semantic type and object. Each line shows a different combination of constraints.

	Semantic Type Signature	Object Constraints
Area	Width, Depth	Same Object
	Width, Depth	None
	None	Same Object
	None	None
Contact Area	Area, Area	Different Objects
	Area, Area	None
	None	None
Pressure	Mass, Contact Area	One of Objects in Contact
	Mass, Contact Area	None
	None	None

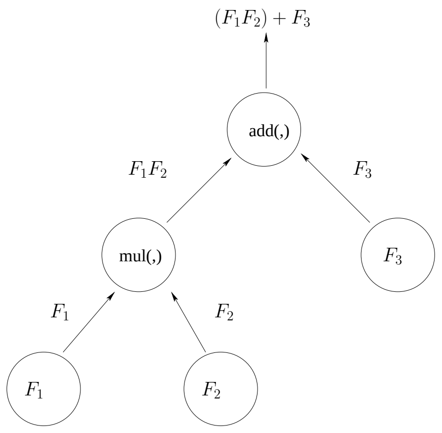


Figure 27: Constructed Features as Trees. F_1, F_2, and F_3 are leaves of the feature tree. They represent the values of their respective features. The tree has height 2 – the longest path from the root to a leaf is two operations, *add(,)* and *mul(,)*. The tree shows what happens to the values of the base features as they are propagated towards the root of the tree. Comparable work must be done for the semantic types of the features.

5.0 RELATED WORK

Representation has been studied extensively in contexts outside of the machine learning and statistics communities. For example, the line of research on heuristic problem solving that started with the GPS program (Newell and Simon, 1963), was applied to human problem solving (Newell and Simon, 1972), and eventually developed into SOAR (Laird et al., 1987) repeatedly emphasized that adequate representation is fundamental to problem solving. The problem of representation in relation to more general cognitive tasks and human psychology is discussed in Newell (1990). Representation is also integral to logical formalisms and the trade-offs they entail (Levesque and Brachman, 1985). Outside of AI based research, the relationship between representation and human mental processing is one of the pillars of cognitive psychology. Here, the mental structures (i.e., representations) used in problem solving and learning make up a key area of conjecture and research.

Learning and representation are inseparable. An early reference to the problem of using an appropriate data representation in machine learning is due to Samuel (1963) (see the quotation at the start of Chapter 1). More recently, dissertations by Matheus (1989) and Donoho (1996), an anthology *Feature Extraction, Construction, and Selection: A Data Mining Perspective* (Liu and Motoda, 1998a), and journal articles (Markovitch and Rosenstein, 2002) show that the problem of representation in learning continues to be of interest and has not been adequately solved. The entire literature on learning using kernel methods deals with applying relatively simple learning methods in complicated spaces generated by functions of the input features.

The threads of inquiry into feature transformation can be segmented by the underlying learning mechanism and representation that they support. Segmentation by learning mechanism serves to follow the threads of research that are closely tied to the learning community in which they were developed. For example, ILP researchers and attribute-value based learning researchers have typically used descriptor modification systems from their own area (i.e., predicate invention for ILP and feature construction for feature based learning). Hence, the intellectual lineage of various systems follows a subset of the learning community. We can look at three major divisions:

1. learning from examples with relational descriptions (a large subset of which is ILP),
2. learning from examples with a propositional (attribute-value) description, and
3. other learning and discovery paradigms.

The choice of representation used in a learning problem drives the algorithms that are appropriate for use in solving that problem and vice versa. Instances need to be represented as numerical vectors for statistical learning methods. Symbolic representations can be used in symbolic learning systems. Predicate logic representations can be processed by ILP and analytic learners. Conversely, the selection of a particular type of learning system can lead to re-representation of instances that is appropriate for that learner. Representation shift (Cohen, 1990) is feature construction for this purpose.

5.1 LEARNING FROM RELATIONAL DESCRIPTIONS AND INDUCTIVE LOGIC PROGRAMMING

A relational description of an example portrays that example as a composite structure including various components (Dietterich and Michalski, 1983). The description is normally captured with predicate calculus descriptors (i.e., variables, predicates, and function). The descriptor transformation mechanisms used in relational learning, including ILP, are quite general because they allow the creation of both propositional descriptors as well as new logical predicates, variables, and functions.

There are two main methods of inductive logic programming[1] (Lavrac and Dzeroski, 1994). The first is a bottom-up approach: form the most specific generalizations that are justified by the initial data and repeat this process until no more generalizations are justified. The second is a top-down approach: start with the most general concept and add restrictions to it. When these

[1]These methods underlie any search based learning program.

Table 12: A Typical ILP Learning Problem. The goal here is to learn the parent relationship that generalizes the concepts of mother and father.

Example	Class
daughter(ted, barb)	Negative
daughter(barb, ted)	Positive
daughter(barb, judy)	Positive

generalizations or specializations are created, they may or may not involve new descriptors. If the new concept does not involve a new descriptor, then the new clause is selective. If it does create a new descriptor, it is constructive.

5.1.1 A Predicate Invention Example

A typical ILP learning task (Lavrac and Dzeroski, 1994) is to learn the definition of the relation $daughter(X, Y)$ in terms of some background knowledge and a number of positive and negative examples of this concept (see Table 12).

Background knowledge for the problem would be:

1. $father(ted, barb)$, $mother(judy, barb)$
2. $male(ted)$, $female(barb)$, $female(judy)$

An ILP algorithm could achieve the following definitions of daughter:

1. $daughter(X, Y)\ :- female(X),\ mother(Y, X)$
2. $daughter(X, Y)\ :- female(X),\ father(Y, X)$.

This definition can be simplified a bit if we understand the commonalities between mother and father. In this case, if we construct a new predicate, say *newPred57* such that:

1. $newPred57(X, Y)\ :- mother(X, Y)$
2. $newPred57(X, Y)\ :- father(X, Y)$.

Table 13: Compressing a Partition of Variable Values. If *Red*, *Blue*, and *Green* form a partition of the *Color* of examples, then the features testing those values can be compressed from a series of predicates to a function. Likewise, the functional form can be expanded to the predicate form.

Example	Color(Ex, Blue)	Color(Ex, Red)	Color(Ex, Green)	Color(Ex)
1	T	F	F	Blue
2	F	T	F	Red
3	F	F	T	Green
4	T	F	F	Blue

With these new definitions, we could legitimately rename *newPred57* as *parent* and subsequently compact the definition of *daughter* to $daughter(X, Y)\ :-female(X), parent(Y, X)$. With $parent\ :-mother$ and $parent\ :-father$ in the initial KB, ILP methods are capable of finding this simplification.

Another example of compressing information into fewer predicates is shown in Table 13. Here, several values *Color* are taken to form an informal partition of the entire range of values for *Color*[2]. These values can be represented as new objects in the ILP dataset and the information represented as a binary predicate can now be represented as a relation between objects. Compression of this type, and the inverse expansion, can be done by ISAK.

The reverse process, moving from sparse relationships to fully instantiated examples, is the fundamental idea behind propositionalization. Lavrac and Flach (2001) describe one system for propositionalization. Depending on the level of semantic information available in a background KB, it may be important for a feature constructor to either compress predicates into relations or to expand relations into predicates.

[2]If the value for every color were *False* then we could define a special attribute, *Other*, to hold missing color values.

5.1.2 Constructive Induction Operators

What are some specific methods of creating new descriptors in ILP? Michalski (1983) gives a couple of examples of constructive generalization rules. Here is one:

> If a concept description *C* contains a part F_1 and it is known that $F_1 \Rightarrow F_2$, then a more general description *C'* can be obtained by replacing F_1 with F_2.

For example, if it is known that:

1. an object which is black, writable, long, and wide belongs to the class of blackboards, and
2. if an object is both long and wide then it is large.

Then, we can create a new concept description in which things that are black, writable and large are blackboards.

The generalization rule can be instantiated by:

1. counting existentially quantified variables in a limited range,
2. counting the number of arguments to a predicate that satisfy some other predicate,
3. following chains of transitivity and inferring descriptors of the whole chain from individuals on the chain and also characteristics of the length of the transitive chain, and
4. detecting relationships by means of numerical interdependence.

As an example of the second instantiation, consider the predicate $contains(A, B_1, B_2, ..., B_n)$. With this predicate, we can determine the number of B_i that satisfy some other predicate(s), for example, $large(X)\, and\, red(X)$. Putting these pieces together, we can create a new predicate that determines the number of *large* and *red* objects *contained* in object *A*. If the dataset is in a propositional form, ISAK can generate these conjunct predicates.

An example of the third instantiation follows the transitivity of stacked blocks with a predicate *on-top-of.* Counting the number of unique objects occurring in such a chain of predicates will yield a new concept: *size-of-stack.*

Several additional operators are described in the DUCE system (Muggleton, 1989). These are absorption, identification, truncation, inter-construction, intra-construction, and dichotomization. These work by either rewriting terms, dropping terms, or creating new productions that simplify other productions. The goal of these operations is to reduce the number of descriptors in a set of examples without substantially increasing the number of counterexamples to the more general rule. The reduction in descriptors is due to the creation of a more powerful vocabulary that more concisely describes the same relationships contained in the initial vocabulary.

5.1.3 Michalski's Constructive Induction Divisions

In the relational setting, Wnek and Michalski (1994) identify four kinds of constructive induction: (1) data-driven CI (DCI) where new descriptors are found by searching for relationship between the base descriptors, (2) hypothesis-driven CI (HCI) where generated concept hypotheses drive the modification of the base descriptors, (3) knowledge-driven CI (KCI) where domain knowledge is utilized to construct new descriptors, and (4) multi-strategy CI (MCI) utilizing two or three of the previous methods. These divisions were used in analogy when I defined different types of feature construction.

5.1.4 Constructive Induction and Knowledge

Several KCI systems are briefly reviewed in Donoho (1996) and Wnek and Michalski (1994); many systems show up in both papers. A number of systems in the second paper contain KCI interacting with either DCI or HCI (which makes them MCI systems). The majority of the systems in both reviews follow closely in the line of inductive logic programming. In most ILP systems, increasing the background knowledge increases the complexity of the hypothesis search space. The increased complexity is due to the database of facts is larger with the additional knowledge.

The types of knowledge used in constructive induction are discussed in more detail in the discussion on Donoho's dissertation in Section 5.3.3.

5.2 LEARNING AND REPRESENTATION IN ATTRIBUTE-VALUE CONTEXTS

A second major division of learning programs, learning from examples with attribute descriptions, is typical of neural network, statistical, decision tree, and rule-based learning. These methods fit a hypothesis (e.g., mathematical, decision tree, rules) to the data. The form of input to these methods is an attribute-value representation of the data. The attribute-value representation is as expressive as a general structural representation. However, achieving this equivalence may require, in the worst case, an exponential increase in the number of attributes. A new attribute must be created for every element in the cross-product of relationships and for each object in those relationships as in propositionalization (Section 5.1.1).

In inductive learning, typical modifications to the data representation are feature construction and feature selection. Feature construction and selection may occur incidentally within a learning algorithm. An example of this is the weighting provided by internal nodes in a neural network; weights of zero on a feature indicate that the feature is not selected for use in hypotheses. Construction and selection may also be done as a pre- or post-processing step in learning. In the FRINGE system, new features are constructed based on an induced decision tree (Pagallo and Haussler, 1990). Since feature construction occurs after a hypothesis is created and because the created features are a function of the hypothesis, it has been termed *hypothesis-driven constructive induction*. In the case where construction and induction are performed separately, this becomes *hypothesis-driven feature construction*.

5.2.1 Prior Work

The progression of work in the feature construction literature has been one of iterative improvement. A review of the related and prior work sections of various feature construction publications (Matheus, 1989; Donoho, 1996; Markovitch and Rosenstein, 2002; Wnek and Michalski, 1994) and the chapters in Liu and Motoda (1998a) reveals a consistent trend towards the use of more complicated operations and data types in feature construction programs. The CITRE program (Matheus, 1989; Matheus and Rendell, 1989), for example, used a single Boolean operator *and(.,.)* to combine Boolean features in the context of decision tree learning. The FRINGE algorithm (Pagallo and Haussler, 1990) and descendants work by combining tree leaf nodes to form more compact trees by relieving the problem of *replication* of subtrees. Replication occurs when the same subtree occurs more than once in a decision tree. Here again, the FRINGE algorithm limits itself to constructions using a single Boolean operator, *and(.,.)*. Because *and* captures the semantics of descending a path on a decision tree, conjugation is a reasonable operation to capture the structure of decision trees. FICUS (Markovitch and Rosenstein, 2002) allows the use of general computable functions with a fixed number of arguments.

5.2.2 CITRE

Matheus' dissertation (Matheus, 1989) presents two primary contributions: a framework for analyzing constructive induction systems and CITRE. CITRE is an implementation of a constructive induction system using the results of applying his framework to analyze a number of existing systems. I will look at each of these results, in turn. Though Matheus claims to analyze constructive induction systems, his work is most applicable to feature construction systems proper and I will discuss it in the feature construction context.

The framework that Matheus presents is centered around four phases in feature construction. The four major phases in his framework include: (1) detection of need, (2) selection of constructors, (3) generalization of constructors, and (4) evaluation of the results. In this description, a constructor is an operator paired with n operands where n is the arity of the operator and the actual constructive process occurs between the third and fourth step.

Matheus considers these phases to be soft delineations among different systems in two respects. First, depending on the perspective of someone applying the framework, portions of a given program may fall logically into different phases of the framework. For example, domain knowledge used to evaluate features may be described as user input to filter results in the evaluation of results or the knowledge could preclude different constructors in the selection phase. Second, in modeling an implemented system into the form described by Matheus, it may be quite difficult to separate the preexisting system into these four parts. For example, while Matheus found it relatively easy to implement BACON with his framework, he states, ”In practice, it may be difficult to separate an existing algorithm into distinct modules ...” (pg. 65).

Matheus uses his framework to analyze a number of systems. This analysis is then put to use in designing a feature construction program called CITRE that operates on decision trees. Within CITRE there is a sharp delineation among detection, selection, generalization, and evaluation. Matheus incorporates domain knowledge into CITRE by constraints on the selection of operands.

After initial and run-time operator selection, domain knowledge serves as a conjunctive list of restrictions on the legal pairs of operands to $and(.,.)$.

ISAK presumes the need for feature construction. As such, its detection capabilities are limited to specific evaluations of features: if no features are suitable components for feature construction, then no feature construction will be performed. ISAK uses data and knowledge to limit its selection of the feature constructor functions that it will apply. ISAK does not provide further generalization of its constructor functions after they have been found to be useful. ISAK operates in an iterative fashion and repeatedly evaluates the produced features until the terminating conditions are reached.

5.2.3 Various Systems

Other programs (Zheng, 1998) have been developed that use more complicated Boolean features constructed from base Boolean features. Two typical feature constructor functions are (1) the *M-of-N* operation which is a generalization of the *or* Boolean operation and (2) the *X-of-N* operation which is a generalization of the *and* operation. In *M-of-N*, a feature is constructed that is *True* if and only if *M* or more of *N* specified features are *True*. An *X-of-N* construct is *True* if and only if exactly *X* of *N* specified features are *True*. These constructed features can be used to simplify decision trees; the most fruitful gains come from trees representing binary logic functions such as the parity and majority functions.

CN2-MCI (Kramer, 1994) moves one step away from feature construction with strictly Boolean domains. It allows an operation to take pairs of *n*-ary features, map them through a hierarchical clustering algorithm, and produce a single Boolean feature as the result. HINT (Zupan et al., 1998) also restricts itself to Boolean input domains and potentially creates and outputs features that are 4-ary functions of the input features. HINT uses methods from switching circuit analysis to decompose complex table functions into potentially less complex pairs of tabular functions. Feature construction is performed from this by executing many decompositions and choosing the best decomposition to retain as a new feature. In this context, the best decomposition is the decomposition with the fewest values.

Another investigation (Sutton and Matheus, 1991) moves the feature construction task to numeric features. Sutton and Matheus construct arbitrary order polynomials by a selection process based on linear regression and a construction process that performs a linear combination of features plus some constant term.

5.2.4 FICUS Overview

The FICUS system of Markovitch and Rosenstein (2002) extends the types of constructable features. FICUS makes use of a formal grammar to allow input of *feature space specifications* (FSSs). FSSs drive a multi-strategy feature construction system. FICUS uses an inductive learner to provide context for feature construction (i.e., hypothesis-driven feature construction). FICUS also makes use of that same learner to evaluate features (i.e., data-driven feature construction). In the presented paper, the authors implement a feature construction system that uses a decision tree learner. However, the authors claim that other concept learners could be used to provide the feature construction context and feature evaluation. Even though a learning system is used internally by FICUS, the output of a run of FICUS is a feature set which can be used by any attribute-value learner.

5.2.4.1 FICUS Input For a given learning problem and associated construction task, a FSS identifies (1) the set of basic features, (2) the set of constructor functions, (3) the domain and range of each constructor function, and (4) a set of constraints on the application of the constructor functions. The set of basic features is a subset of all features. The constructor functions are any unary or binary operations defined over the ranges of the features. The domain and range of the constructor functions and the base attributes are constrained by the following primitive types: nominal, ordered-nominal (i.e., ordinal), and continuous. Additionally, there are compound types including sets and lists of the primitive types. For example, the function *max* takes a set of continuous values and returns a single continuous value.

Lastly, the elements of the argument constraint set are Boolean functions that take an argument and test its compliance. The three constraints provided in FICUS are *NoConst*, *Const*, and *Unique*. Respectively, these forbid constant features, require constant features, and forbid identical elements. These constraints are not semantic in nature; they simply enforce mathematical requirements of certain operators (i.e., forbidding division by zero). The constraints make no reference

```
def CreateNode(data):
    if allSameClass(data):
        return Node(getClass(data))
    else:
        newFeatures<-createNewFeatures(data)
    bestFeature <- pickBest(newFeatures + features(data))
    N <- Node(bestFeature)
    for each split, s, defined by N:
        append CreateNode(s) to N's children
    return N
```

Figure 28: FICUS Pseudo-code. *features()* returns the features of its argument.

to any general or specific knowledge about the objects in a learning environment or semantically useful constructors.

5.2.4.2 FICUS Algorithm FICUS generates features in an evolutionary approach whereby new features are composed from those features that currently exist. The presented version of FICUS makes use of a decision tree learner internally to provide a context for feature construction. When new tree nodes are constructed, if the node is not of a uniform class, then the feature construction subroutine is called (see Figure 28). The routine selects features to use in constructions and then performs constructions with the highest evaluated pairs of features (see Figure 29). The initial selection process is a function of the decision tree (i.e., the hypothesis) and the data set.

Eventually, a complete decision tree is formed. The constructed features used in the tree as decision nodes are gathered together. These constructed features are passed through a selection process that measures their utility in the induced decision tree. The best of these features are retained and are then used in subsequent rounds of construction. The use of a decision tree makes FICUS a hypothesis-driven approach to feature construction, even though the hypothesis is devel-

```
Choose pairs of features A,B
If A==B:
     do unary ops
     discretize continuous
     count symbolic
For all base A,B:
     do unary ops
     do binary ops
     generate set {A,B}
     generate list [A,B]
For aggregate A xor B:
     insert base into aggregate
     replace aggregate elements by base
```

Figure 29: FICUS Application of Feature Constructor Functions. FICUS applies feature constructor functions to pairs of features. The application is based on the types of the elements of the pair. If both elements of the pair are the same, FICUS applies unary and symbolic operations. If the types are primitive, FICUS applies binary operations and generates aggregates of the two features. If both are aggregates, then FICUS generates combined aggregates of the features.

oped in an internal module of FICUS and not in an independent learner. The internal learner should be viewed as a method of finding features and interactions between features that are of interest to the feature construction module.

5.2.4.3 FICUS and Problem Domains In the application of FICUS to problem domains, the authors present a list of the operators (Table 14) used over all the domains on which they experimented; certain operators were selected for use in certain problem. For example, in the UCI wine domain the feature set consists of 13 numeric features and the set of available construction functions is the arithmetic operations (addition, subtraction, multiplication, and division). Different operations are selected for use in other domains.

The authors note that a limitation of the FICUS approach is that it depends on the high utility of the building blocks of complex features to find those complex features. This assumption held in most of the domains they tested. A solution to selecting building blocks piecewise is to widen the beam search size, but this is not always computationally practical.

Like FICUS, ISAK is able to use arbitrary, fixed argument size computable functions as feature constructors. Unlike FICUS, ISAK relies on knowledge, not hypotheses, to drive its feature construction. Both FICUS and ISAK use data-driven analysis in feature construction.

5.2.5 Representation and Modification in Other Learning Contexts

As an example of the final division of learning systems, miscellaneous learning paradigms, I present BACON. BACON's operation is highly analogous to the constructive process in learning from examples in an attribute-value representation.

BACON (Langley et al., 1987) looked for algebraic patterns in datasets as a process sufficient to mimic the discovery process in scientific inquiry. It focused on finding constant ratios that were assumed to represent empirical laws and used heuristics, driven by the given data, to search the space of possible equations. BACON rediscovered laws of Galileo, Kepler, Boyle, and Ohm. Notably, BACON was limited by its assumption that all data provided to it were relevant, and it relied on multiplicative terms and ratios to describe relationships among features. The use of heuristics driven by theory was developed in a later program called STAHL (Zytkow and Simon,

Table 14: FICUS Operators.. A list of FICUS operators, operands, and examples where $f_{new} = add(f_1, f_2)$ represents the creation of an entire column of new values for f_{new} from the values of f_1 and f_2.

Operation	Arguments	Example
addition	numeric feature	$add(f_1, f_2)$
division	numeric feature	$div(f_1, f_2)$
subtraction	numeric feature	$sub(f_1, f_2)$
multiplication	numeric feature	$mul(f_1, f_2)$
absolute difference	numeric feature	$abs(f_1, f_{2)}$
average	numeric set	$avg(\{f_1, f_2, f_3\})$
maximum	numeric set	$max(\{f_1, f_2, f_3\})$
minimum	numeric set	$min(\{f_1, f_2, f_3\})$
equality test	symbolic feature	$eq(f_1, blue)$
range test	numeric feature	$inRange(f_1, [0, 10])$
occurrence counting	symbolic set	$count(\{f_1, f_2, f_3\}, val\}$

1986). STAHL starts with data and develops hypotheses (i.e., theories) to explain relationships in the data. These posited relationships are used in subsequent rounds of discovering relations (i.e., equations that model) in the data. BACON has given rise to a community of equation discovery researchers whose aim is to find simple equations that summarize sets of data. Such work is typified by LAGRANGE (Dzeroski and Todorovski, 1995) and LAGRAMGE (Todorovski and Džeroski, 1997).

Of the BACON systems, BACON.4 is of most interest because its operation involved creating new terms called intrinsic properties from nominal variables. Intrinsic properties are associated with independent variables in an experiment. Suppose that experiments are run over several different metals. Then, some numerical property of those metals might be introduced and subsequently used to derive scientific knowledge. For example, *gold* has a particular density as an intrinsic property. Intrinsic properties are dependent on theory and thus are a subset of *theoretical terms;* these are contrasted with observable properties of objects in a soft distinction. As a pragmatic distinction between the two types, we can allow measurements obtained from non-problematic instruments to be considered observable. The other measurements (e.g., from troublesome instruments) are theoretical. The terms created by ISAK are intrinsic, theoretical terms because they derive from the knowledge base given as input for a construction and learning problem.

STABB carries out descriptor modification on data in a structural representation. STABB (Shift to a Better Bias) (Utgoff, 1983), can be interpreted as a representation modification system (Matheus, 1989). STABB's purpose is to address representational problems in LEX (Mitchell et al., 1983), a system for learning heuristics for solving symbolic integration problems. If LEX's inductive algorithm fails to find a suitable hypothesis for a concept, STABB will attempt to create new, more general concepts for LEX to process by joining existing descriptions by least disjunction. STABB uses a transformation approach and constraint propagation to drive constructive induction. STABB also uses transformation methods which operate by applying predefined transformations to training instances.

5.3 KNOWLEDGE IN FEATURE CONSTRUCTION

Knowledge plays an important role in both human and machine learning. Inductive logic programming and explanation based learning are two areas in machine learning where this importance has been most directly addressed. The use of knowledge in ILP was discussed in Section 5.1.

5.3.1 Knowledge in Explanation Based Learning and Problem Solving

Explanation based learning (EBL) (Mitchell, 1997) is a form of analytical learning and it has many similarities to inductive learning with background knowledge. The main goal of EBL is to make up for a lack of data (i.e., having too few examples) with background knowledge to allow suitable learning of hypotheses. When using background knowledge to augment feature construction, there is presumably, though not necessarily, a sufficient amount of data to make inductive inferences. The use of knowledge is to modify the feature space.

EBL uses prior knowledge to reduce the complexity of the hypothesis space being searched. This is also one of the goals of modifying the feature space by feature construction. Hence, EBL and feature space modifications can be seen as two methods with similar types of input and goals. The methods by which they operate also share some parallels. If we look at feature construction as a learning process (i.e., heuristic search that results in a generalization) then feature construction with background knowledge can be interpreted as combined analytical and inductive learning (i.e., EBL).

An interesting system that uses knowledge for constructive induction (most specifically, feature extraction) integrated with problem solving is CINDI (Callan, 1989, 1990). CINDI is a system that works in the domain of learning for problem solving systems. In particular, CINDI addresses the issue that a suitable problem solving representation for a given task – playing checkers – may not be suitable for an associated learning problem – learning winning moves. Hence, the difficult problem of representation must be solved twice by problem solving systems with learning components. CINDI's learning task is to learn search control knowledge. A logical specification of the primary problem to be solved is given as input. Features for learning are automatically created from a logical specification of the primary problem and transformations are applied to this specification. So, knowledge of the learning domain is used to drive the representation used in learning.

5.3.2 Knowledge in Feature Construction

One of the trends in feature construction is to make additional knowledge available to the construction program (Donoho and Rendell, 1998). Knowledge can be integrated implicitly or explicitly. Implicit knowledge, also called procedural knowledge, is built into the operation of a program (i.e., it rests in the program code and internal structures). Explicit knowledge, also called declarative knowledge, allows for a more modular set of constructions from knowledge – the knowledge is an input to the feature construction mechanism. ISAK makes use of explicit knowledge to describe the available feature constructor functions, to describe the characteristics of the learning features, and to specify its search parameters.

Although there are several systems that perform feature construction, there is little use of explicitly represented background knowledge in these systems. As rough evidence of this fact, consider that aside from a chapter (Donoho and Rendell, 1998) based on Donoho's dissertation which will be discussed shortly, Liu and Motoda (1998a) has only two other references to background knowledge in the index. Both are pointers to general statements about the use of background knowledge, not to research involving background knowledge. In Liu and Motoda, knowledge of probability structure – for example, that a sample probability distribution function is unimodal – is discussed as it related to choosing feature selection methods (Pudil and Novovicova, 1998).

Many feature construction systems that do not emphasize the use of background knowledge allow for some use of constraints on the operands provided to operators. The constraints are typically mathematical constraints on the domains of the functions. For example, the second argument to a division operation should not be zero. The value of this limited knowledge has not been evaluated.

5.3.3 Donoho's Knowledge Framework

Donoho (1996) presents the first detailed discussion of computationally automated knowledge-driven feature construction. He approaches knowledge-driven feature construction by (1) identifying the needs of feature construction, (2) identifying what sorts of knowledge can meet these needs, and (3) applying this knowledge in a search based feature construction paradigm. The types of knowledge that Donoho considers include: scale, relevance, support, correlation, contingency, hierarchical, and normalization knowledge. In addition, Donoho considers knowledge of time series, dimensional analysis, and deductive theories. The analysis of knowledge and systems is thorough. Donoho leaves one description slightly vague: the hierarchical knowledge of Aronis and Provost (1994) is not concerned with interactions between different attributes; it deals with a hierarchy of values for a single attribute. This can be clearly seen in another presentation of their work (Aronis and Provost, 1997) that emphasizes an efficient algorithmic implementation for using this knowledge.

In the context of a search problem, the pieces of knowledge that Donoho discusses can affect (1) the starting node or nodes in the search space, (2) the ordering of successor nodes in the search space, (3) the connectivity between different nodes in the search space, (4) the boundaries of the search space, and (5) the method of search used to traverse the search space such as depth-first, breadth-first, or beam search.

To be clear on Donoho's contribution to the use of knowledge in learning, let us consider his example of knowledge use. Suppose that the starting information in a task consists of a deductive theory, dimensional analysis knowledge, and the given dataset. The deductive theory specifies some starting points (i.e., an initial set of constructed features) and the dimensional analysis is used ”to create a boundary between high quality and low quality features” (pg. 55). The boundary can be construed as either connectivity in the search space or ordering on successor nodes, but Donoho does not elaborate. Now, a researcher could write a program implementing a search using these principles and run it. To begin with, the researcher might use the starting points from the deductive theory directly as the initial search nodes. A learning method would be applied to the resulting dataset and the results would be evaluated to determine if they are satisfactory or not.

Supposing the results (e.g., classification accuracy) are not satisfactory, a breadth first search could be run from the initial starting points. Again, the resulting features could be used in a learner and evaluated. Suppose this results in a higher accuracy. Now, perhaps the researcher tries a depth-first search of the constructed feature space. The result is a lower accuracy. Finally, a hill climbing search is run that ignores the boundary knowledge provided by the dimensional analysis. The accuracy drops. Given these results, Donoho states, ”We can conclude for this series of experiments that a good set of features lie close to the starting points but require a thorough search to be found" (pg. 55).

To summarize, Donoho proposes using background knowledge to *manually* conduct a meta-level search through a space of feature space searches. In contrast, ISAK uses fragmentary knowledge as constraints on search in an automatic fashion.

Donoho leaves open two main lines of inquiry. The first line is to create an automated system for feature construction that uses knowledge in an automated manner. In Donoho's work, the framework of knowledge analysis is used while coding each particular search. So, knowledge is only incorporated at the point it is used by the coder. Further, new pieces of knowledge require additional coding of the search process and possibly coding several different feature space searches.

The second open task is extending the types of knowledge that can inform the search for good features. Three sub-tasks are: (1) extending the types of knowledge known about the data (i.e., extending Donoho's direct work), (2) delineating the sorts of operations that can be performed on data and the sorts of knowledge about these operations that can be applied to construct features (something Donoho does not discuss), and (3) implementing these extensions and Donoho's original framework in an automated, computational system.

ISAK uses knowledge to determine connectivity in the feature space and boundaries of the feature space. ISAK may be used to deductively specify new starting nodes in the feature space. ISAK operates in a purely automated fashion; it may be used iteratively but there is not need for user intervention after the input KB specification is developed. ISAK relies upon numerical heuristics for evaluating and ranking features.

5.3.4 Other Knowledge-Driven Feature Construction Systems

Wnek and Michalski (1994) and Donoho and Rendell (1998) provide overviews of systems making use of knowledge to guide constructive induction and feature construction. Examining these reviews, it becomes clear that most knowledge used for representation manipulation comes from structural descriptions which has found its way into the ILP lines of research. The systems geared towards an attribute-value representation, such as MIRO (Drastal and Raatz, 1989), tend to use deductive theories in a single step to develop new initial features that are then used in learning.

Some systems have only utilized knowledge in the form of human interaction. COPER (Kokar, 1986) takes a BACON-like approach to equation discovery but extends it by using background knowledge of dimensional analysis to constrain equation construction. Like BACON, COPER is looking for equations that derive constant terms.

6.0 IMPLEMENTING ISAK: INPUT SPECIFICATION AND ALGORITHMS

ISAK's algorithms and the input format are both implemented in the Python programming language (PSF, 2006). Accessory learning elements – information gain routines, low level application of functions to features in a vector-wise fashion, discretization of continuous values – are implemented with the Orange data mining system (Demsar et al., 2004), which is itself a combination of C++ and Python code.

6.1 ADDITION AND THE INTERACTION BETWEEN SEMANTIC TYPES AND FEATURE CONSTRUCTOR FUNCTIONS

Because of the intertwined nature of the pieces of the implementation, this section describes how semantic types interact with four different feature constructor functions. Details are delayed to Section 6.2. ISAK has four distinct, built-in methods of applying addition to two features (Table 17):

1. add_1 unconstrained addition of two numeric values,
2. add_2 addition on two *GenericQuantities* (numeric values without a specific measurement type) that unify below *Top* (i.e., the FCFs in the Wine CLP),
3. add_3 addition on two *GenericQuantities* that are members of the same semantic class, and
4. add_4 addition of two lengths that are in the same dimension.

If add_1 is in the set of FCFs for a CLP, then every pair-wise combination of numeric features is a candidate to be combined with addition. add_1 is included so that ISAK can perform addition in extremely knowledge-lean problems. add_1, and similar FCFs, allows comparison of

ISAK without knowledge and other feature construction programs that do now use knowledge. In general, knowledge-lean operators, like add_1, do not adequately constrain search in the space of constructable features.

When add_2 is in the FCF set of a CLP, addition is performed between all pairs of *GenericQuantity* features g_1, g_2 such that the semantic classes of g_1 and g_2 unify to some non-trivial member of their semantic class hierarchy (i.e., $Unify(g_1, g_2) \neq Top$). Depending on the semantic class hierarchy for a CLP, add_2 can significantly constrain the space of constructable features.

add_3 is similar to add_2 but it requires that g_1 and g_2 belong to the exact same semantic class, as opposed to unifying to any non-trivial unification of the semantic class hierarchy.

add_4 is significantly different because it requires detailed feature types and specific type properties. add_4 applies to two *LengthTypes*. It imposes two constraints: (1) a *SameObject* constraint on the objects measured by the *LengthTypes* and (2) a *SameDimension* constraint on the dimensions of the measurements. The *SameObject* constraint is imposed by checking the semantic classes of the features; *SameDimension* is imposed by checking the semantic type properties of the *LengthTypes*.

6.2 DETAILS OF THE INPUT SPECIFICATION

6.2.1 Semantic Types and Semantic Properties

The semantic types that ISAK provides form a hierarchy like that shown in Figures 22, 23, and 24[1]. Each semantic type is implemented as a Python class; hierarchical relationships are implemented among classes[2]. The classes representing semantic types have member variables representing the semantic properties for that type. A table of ISAK's semantic types is given in Table 15. The semantic properties of a type are the semantic class to which the type's feature belongs (including reference to an object) and its per-type properties (e.g., the dimension of a spatial extent). Object specifications and per-type properties are implemented as tuples of strings. The string values may

[1]Certain levels and branches of that hierarchy are not implemented in ISAK. Where levels are not implemented, there is simply a direct link from the parents of the missing layer to the children of the missing layer.

[2]The hierarchical structure is specified through object-oriented inheritance relationships.

exist in hierarchical relationships. Semantic classes are inter-feature knowledge; their specification is discussed in Section 6.2.2.

ISAK's built-in semantic types are provided as a library and do not need to be explicitly described in a problem input file. Each of these semantic types is an instance of a node in the type hierarchy specified in Section 4.3. The implemented types include typing details relevant to the type's measurement. Users may also specify new semantic types in the knowledge base (KB) file for a construction and learning problem (CLP).

All non-aggregate semantic types have a semantic class as one of their semantic properties. The most important part of the semantic class description is the object measured by a type's feature. Other pieces of the semantic class description include class membership of the object and relationships to other features. For example, consider a novel learning problem involving wine. Let each example have measurements on each of three different wines and let the target class be the wine preferred by a taste tester. Each individual wine has a measurement on its *Ash, Alkalinity,* etc. The measurements can be related in the same fashion as in the semantic hierarchy given in Chapter 2 for the UCI wine problem. However, each feature is now also associated with an object, namely $Wine_1$, $Wine_2$, or $Wine_3$. The object specification is implemented as an element of the tuple that represents the semantic class of a feature type. In the new wine problem, the semantic class for the first wine's ash content would be the tuple[3] $(Wine_1, Ash)$.

Other semantic properties of the semantic type narrow the potential interpretations of that type's feature. For example, if the values of a feature are two-dimensional extents (i.e., AreaType), then describing those two dimensions is a necessary part of the feature description. These two dimensions (e.g., length and height) are specific information that may justify or prohibit other combinations. In ISAK, descriptions of dimensions or other *ad hoc* type characteristics are specified symbolically. Symbolic description allows a variety of dimensions or type details to be used. The choice of the value for a type property can represent further constraints – it is possible to have two spatial coordinate systems within the same example. It is useful to have unlinked coordinate systems, if two objects are measured on different axes or on different scales. For example, $\{up_{frame_1}, up_{frame_2}, down_{frame_1}, down_{frame_2}\}$ may be used to represent two sets of directions in

[3]Here the tuple is a pair. In general, the semantic class specification may have *n* elements. The unification of multiple elements is the a set of least upper bounds in different portions of the semantic class hierarchy.

Table 15: The Semantic Types of ISAK and Their Semantic Properties. The semantic properties of a type specify the details of that type. The semantic class may hold information about the object that a measurement describes and information about the relationship of that measurement to other measurements.

	Semantic Type	Description	Semantic Properties				Example Value	Example Semantics
AggregateTypes	BagType		Semantic Class	Item Type			{a,a,b}	Colors of cars
	ListType		Semantic Class	Item Type			[3,2,5]	Lengths of people
	SetType		Semantic Class	Item Type			{a,b}	Counts of wins
NumericTypes	CountType	Quantity of Item	Semantic Class				10	Books
	GenericMeasureType	Constrained Quantity	Semantic Class	Quantity	Quantifier		10	Books per shelf
	GenericQuantityType	Arbitrary Value	Semantic Class				93.70	Arbitrary value of object (alkalinity of wine sample)
	QuantityType	Measured Quantity of Substance	Semantic Class	Scale	Substance		4.20	Gallons of water in bucket
	LocationType	Where is Object?	Semantic Class				23.70	Arbitrary location (location of player)
	DimLocationType	Location in Dimension	Semantic Class	Dimension			13.20	Longitude of ship
	MassType	Mass of Object	Semantic Class				100.20	Person's mass
	LengthType	Length of Object in Dimension	Semantic Class	Dimension			20.00	Depth of pool
	AreaType	Area of Object	Semantic Class	Dimension 1	Dimension 2		100.00	Acreage of property
	VolumeType	Volume of Object	Semantic Class	Dimension 1	Dimension 2	Dimension 3	3.40	Volume of tennis ball
	DistanceType	Extent of Movement	Semantic Class				1000.00	Distance travelled on vacation
	TimeType	Duration of Event	Semantic Class				25.00	Time of flight
	SpeedType	Rate of Change in Position	Semantic Class	Event			65.00	Car during trip
	RateType	Rate of Change	Semantic Class	Event			10.00	Water during flood
SymbolicTypes	SymbolType	Arbitrary Symbolic Value	Semantic Class	Values			Blue	Color of car

two different contexts within the same learning example. Separate frames of reference in details also prevent the knowledge representation from requiring a canonical and universal coordinate system. From another perspective, multiple coordinate systems allow local frames of knowledge to prevent interactions among unrelated sets of features (i.e., up_{frame_1} cannot interact with up_{frame_2}).

In another example, given that the values of an attribute are board game pieces (i.e., they are of type PieceType), it is possible to specify what class of characteristics those pieces have in common. The pieces (i.e., the values of the attribute) may share positional similarities. In a common case, the attribute represents a constant location in each of the different examples (e.g., the top left corner of the board) and the value represents the piece at that location (e.g., a rook in chess). So, the pieces that are values of the feature share a common location and the KB specifies this fact. We might also be interested in combining pieces with broader, location similarities (e.g., same row or same column). So, our piece in the top left corner of the board is also a member of the top row and the left column.

6.2.2 Semantic Class Hierarchy

Semantic class hierarchies are specified by creating nodes and putting these nodes into a tree. Each node has a set of links to its children and a label to distinguish that node's semantic class. All semantic class hierarchies are rooted at a special node *topClass*. Semantic class membership is indicated when a semantic type for an attribute has a semantic class hierarchy node as the value of one of its member variables. An example of a semantic class hierarchy is given in the description of the wine CLP (see Section 2.1). Any string value of a semantic property that is not explicitly in a hierarchy is compared by equality. Unification between two concepts in the hierarchy is determined by computing the least upper bound of the two concepts. If two semantic classes both involve multiple concepts, then each element of the two semantic classes is unified individually with each element from the other semantic class. The union of the results is the unification of the two semantic classes.

6.2.3 Feature Description

ISAK specifies base and constructed feature semantics by associating a name for the feature with a semantic type for that feature. The program structure describing the feature holds syntactic and numeric information for that feature. Constructed features are additionally described by:

1. the features that are used in the new features' construction,
2. the typing characteristics that these features lead to,
3. the formula applied to the base features to get the current feature,
4. the data values that the actual computations bring about, and
5. a heuristic measurement of the learning value of the current feature.

A number of non-semantic characteristics of a feature are also important. The tree characteristics of a feature (see Section 4.4) provide important pruning information in the feature search space. Associated with each feature is a heuristic evaluation of its usefulness in predicting the target class (e.g., information gain of the feature with respect to the target class).

The *BaseFeature* data structure (a Python class), contains the following member variables:

1. *Name* – a string value such as *a1* in the wine problem or *Temperature* in the weather problem,
2. *Type* – the semantic type of the feature as in Table 15,
3. *Values* – a list of the values taken by each example for the feature,
4. *Components* – a list of components for the feature (for base features the only component of the feature is itself),
5. *Number of Components* – the length of *Components*,
6. *Height* – the height of the feature when interpreted as a tree,
7. *Leaves* – the number of leaves of the feature when interpreted as a tree,
8. *Number of Base Features* – the number of unique base features used to construct the feature (for base features the value is one), and
9. *Parents* – a list of pointers to the parent features of the feature (i.e., the features used in the construction of this feature). Base features have an empty parent list. *Parents* differ from *Components* in that *Components* is reduced to a set of unique features and is a complete record of used features back to, and including, the base features. *Parents* tracks each particular feature

in the argument list used to construct the present feature. *Parents* is used for comparison of the present feature's evaluation to its parents' evaluations.

In addition to setting the member variables, the class initialization also computes the information gain of the feature with respect to the target class, registers the feature values in a hash table (for duplicate checking), and registers the feature with its semantic class.

Constructed features are stored in a *NewFeature* data structure. The *NewFeature* data structure holds all the information for base features with some additions:

1. *Formula* – the feature constructor function used to create the feature, and
2. *Arguments* – an ordered list of the features serving as arguments to *Formula*. *Arguments* are ordered and duplicative while *Parents* are unordered and unique.

The *Value* variable of *NewFeatures* is calculated once when it is first needed and then stored in memory. The memorization is necessary to prevent a series of recursive calls. To construct the values of a new feature, only the values of the parent features need to be accessed.

6.2.4 Constraint Specification

There are two types of constraints in ISAK: semantic constraints and feature constraints. Semantic constraints are functions that take the facts specified by semantic type, semantic class (including object), and other semantic type properties, and return whether or not the given features satisfy those constraints. Feature constraints are constraints on the non-semantic properties of the feature. For example, constraints based on the tree characteristics of a feature (e.g., height or number of leaves) are feature constraints. Feature constraints are implemented as predicates that indicate whether or not a list of features satisfy the given, non-semantic relationships. The important difference between semantic and feature constraints is that semantic constraints are functions of the semantic type; feature constraints are functions of the feature itself. The built-in constraints of ISAK are shown in Table 16.

Semantic constraints augment typing information. Typing information alone may not be sufficient to limit the possible feature constructions. For example, suppose f_1 is the value of coins in a coin jar and f_2 is the value of an investment portfolio. Adding the two values would yield

Table 16: Built-in Constraints of ISAK. The two main divisions of constraints on feature constructor application are constraints that are functions of the semantic type of a feature (i.e., *Semantic Constraints*) and constraints that are functions of the feature itself (*Feature Constraints*).

Constraint Type	Constraint	Description
Semantic Constraints	Same Event	Features describe same event
	Same Object	Features describe same object
	Different Event	Features describe different events
	Different Object	Features describe different objects
	Different Dimension	Features describe different dimensions
	Same Dimension	Features describe same dimensions
	Subset	Class description of first feature is progeny of second
	Superset	Class description of first feature is ancestor of second
	Not All Top	Class descriptions of multiple features are not all TopClass
	Not Both Top	Class descriptions of two features are not both TopClass
	Not Top LGU	Class descriptions of features unify to something besides TopClass
	Same Unification	Unification of class descriptions is same as class descriptions
	Same Type	Features have same semantic type
	Same Type as Aggregate	Semantic type of container elements is same as feature
Feature Constraints	Different Feature	Forbid same feature in two arguments positions
	Unused Feature	Require feature not in components of other
	Disjoint Components	Require features with disjoint components
	First Components not in Second	Require components of second feature not also components of first feature

a value with clear semantics, but the value would be very unlikely to be important. Specifically, the sum is going to come primarily from the value of the investment portfolio. Here, an appropriate constraint would require that the combined features measure the same type of object (i.e., capital assets versus nominal assets). The constraint makes use of the semantic class specification discussed in Section 6.2.1. In the case of aggregate types, the primitive type that is aggregated is processed as a semantic constraint. For example, identifying sets of *LengthTypes* versus sets of *WeightTypes* is done with semantic constraints.

Feature constraints implement non-semantic restrictions on constructions. For example, a feature constraint is needed to limit a feature constructor that produces sums of counts from other counts. The feature constructor may be applied iteratively, and it may apply itself to the original two counts repeatedly. The repetition would no longer be building an aggregate count over many features. Instead, it would be adding one feature back into the total count many times. To prevent repetition, a feature constraint can require that one of the given arguments does not appear in the history of the other argument or, more generally, that the two constructed feature trees are disjoint.

If repeated addition is needed, multiplication is generally a better choice of operation because it is an atomic (i.e., non-compound) construction[4].

6.2.5 Specification of Feature Constructor Functions

A feature constructor function (FCF) is composed of several pieces of information: a definition of the operation it performs, the input typing requirements, further input argument constraints, the output type, and a function that will compute output type properties from the input type properties. A table of most of ISAK's built-in FCFs is shown in Table 17.

The operation that a FCF encapsulates may be any computable function of the formula's attributes. It is a function of a fixed number of attributes. The function arguments may include set, list, and bag valued attributes. FCF may not, as implemented, apply to variable length argument lists other than what is permitted by applying a FCF to arbitrary sets, lists, and bags.

The process of defining constraints is discussed in Section 6.2.4. The purpose of constraints is two-fold: within similar types, there may be references to specific properties of that type. The property information, together with the type, will determine the applicability of a FCF. Secondly, there may be inter-feature constraints that are not semantic in nature. These constraints are easily resolved by processing them at this point. A non-semantic constraint might require an aggregate feature type to have a certain number or range of elements.

Generally, a feature construction formula will only output features of one type, even if it outputs several new features. However, it is possible to specify the output types as a function of the input features. Regardless of type similarity, output type properties will vary with the type properties of input features.

For example, multiplication of an area and a length yields a volume. This is a simple case where the feature constructor function produces a single new feature on each application (the volume of the object). In a more complex case (Table 18), we generate several Boolean features where each is an indicator function for each value of ftr_1. Each feature is tied specifically to the symbolic value to which the original feature was compared. The relationship is reflected in the column headings of Figure 18 and is encoded in the typing details for the new features.

[4]Piecewise addition (versus multiplication) may still be necessary if the number of repeated additions (i.e., one of the factors in a multiplication term) is not available in the dataset.

Table 17: Built-in Feature Constructors. Input types, constraints, operations, output types, and some output typing details for several built-in feature constructors. Examples of actual code for the unification-constrained arithmetic is shown in Chapter 2.

Feature Constructor Name	Input Types	Constraints	Computable Function	Output Types	Output Detail Computation
Square	NumericType		x^2		
Absolute Value	NumericType		abs(x)	NumericType	
Inverse	NumericType		$1/x$	NumericType	
Square Root	NumericType		$\sqrt{(x)}$		
Generic Arithmetic	NumericType, NumericType		+, -, *, /	NumericType	
Unifiable Arithmetic	GenericQuantityType, GenericQuantityType	Non-TopClass Unification	+, -, *, /	GenericQuantityType	unifyDescriptions
Same Class Arithmetic	GenericQuantityType, GenericQuantityType	Same Parent Class Unification	+, -, *, /	GenericQuantityType	unifyDescriptions
Same Dimension Length Arithmetic	LengthType, LengthType	Same Dimension	+, -	LengthType	Inherit Objects and Dimension
Bag, List Creator	NonAggregateType, NonAggregateType	Different Feature, Same Type	Bag{x,y}, List[x,y]	BagType, ListType	Inherit Type and Description
Bag, List Extender	BagType(NonAggregate), NonAggregate	Single not in Aggregate, Same Type	union(x,Bag(y)), append(x,y)	BagType, ListType	Inherit Type and Description
Bag Numeric Uniqueness	BagType(NumericType)		$\exists x, y \in Bag$ $\lvert(x-y)\rvert < \delta$	BooleanType	
Bag Numeric All Different	BagType(NumericType)		$\neg\exists x, y \in Bag$ $\lvert(x-y)\rvert < \delta$	BooleanType	
Aggregate Max, Aggregate Min, Aggregate Avg	Aggregate(Continuous)		max(x), min(x), avg(x)	Continuous	
Aggregate Count True	Aggregate(Boolean)		|{x | x in Bag and x =True}|	CountType	
Boolean Or, Boolean And	BooleanType, BooleanType		or(x,y), and(x,y)	BooleanType	
Boolean Not	BooleanType		not(x)	BooleanType	
Numeric Equality	NumericType, NumericType		$\lvert(x-y)\rvert < \delta$	BooleanType	
Numeric Inequality	NumericType, NumericType		x > y, x < y	BooleanType	
Equality Test All Symbol Values	SymbolType		$\forall\, values \in SymbolType$ $Equals(x, value)$	Multiple BooleanTypes	
Equality Test All Symbol Values	SymbolType		$\forall\, values \in SymbolType$ $NotEqual(x, value)$	Multiple BooleanTypes	
Ternary Test	NumericType, NumericType		x > y, x < y, x eq y	SymbolicType	

Table 18: Equality Testing. Equality testing over several symbolic values of a feature converts the symbolic feature to several Boolean features.

Example	ftr_1	$Eq(ftr_1, A)$	$Eq(ftr_1, B)$	$Eq(ftr_1, C)$
1	A	True	False	False
2	B	False	True	False
3	A	True	False	False
4	C	False	False	True

6.2.6 Search Parameters

The legal space of constructed features implied by the semantic types, constraints, and feature constructors will grow combinatorially for all but the most trivial CLPs. To allow the program to execute in a reasonable time, the generation of features is embedded inside a beam search. The beam search process and the use of the following parameters is described in more detail in Section 6.3.1. Table 19 shows the search parameters used by ISAK.

For now, it is sufficient to describe the parameters used in the search to accept, reject, or delay judgment on constructed features. The search parameters described here form a different class of constraint than those specified in Section 6.2.4. The constraints here constrain the exploration of the feature space. The constraints discussed in Section 6.2.4 affect whether or not a feature constructor function may be applied to particular features as part of the successor function. The search parameters fall into four categories:

1. constraints on the size of generated sets of constructed features,
2. constraints on the form (i.e., tree characteristics) of the constructed features (see Section 4.4),
3. constraints computed from the values of the constructed features, and
4. constraints on the heuristic evaluation of the features.

Table 19: Search Parameters for ISAK. Feature evaluation constraints are requirements on the value of information gain for constructed features. Tree characteristic parameters are constraints on the tree form of constructed features. Beam and size parameters affect the storage of the beam (i.e., the search fringe) directly.

Parameter Type	Parameter Name	Description
Successor	Successor Function	Method used to choose features and apply feature constructor functions
Feature Evaluation	Minimum Measure	Minimum measure for an acceptable feature
	Minimum Improvement	Minimum improvement in measure over parents for an acceptable feature
	Junk Measure	Maximum measure for a feature to be pruned from the search space
Tree Characteristics	Minimum Features	Minimum number of base features used in an acceptable feature
	Minimum Height	Minimum height of an acceptable constructed feature tree
	Minimum Leaves	Minimum number of leaf nodes for an acceptable constructed feature tree
	Maximum Features	Maximum number of base features used in an acceptable feature
	Maximum Height	Maximum height of an acceptable constructed feature tree
	Maximum Leaves	Maximum number of leaf nodes for an acceptable constructed feature tree
Beam and Set Size	Maximum New	Maximum number of New Others at the end of a round of generation
	Maximum Old	Maximum number of Old features used to start a round of generation
	Maximum Current	Maximum number of Current features used to start a round of generation
	Maximum Return	Maximum number of features returned by ISAK after all generation completes

A final parameter to the beam search, which is generally not specified on a per-problem basis, is the successor function used to expand the current fringe of features in the search space. Successor functions are discussed in Section 6.3.4.

6.2.7 Aggregate Features

The aggregate types were described in Section 4.3. There are several ways that an aggregate feature can be introduced into a dataset:

1. by hand – the aggregate feature is added to the base set of features,
2. by KB – the aggregate feature is explicitly created from base features in the KB file (before search through the feature space), and
3. by feature construction – the aggregate feature is created in competition with other constructed features.

How do aggregate features differ from a set of features that make up a semantic class? Aggregate features are tightly grouped for some reason. If they are grouped by hand or by KB, the features in the aggregate make up some arbitrary conceptual group in the domain. If an aggregate type is created by feature construction, they share some semantic type and properties. The degree of similarity is determined by the aggregate feature constructor function used to create the bag, list, or set.

Semantic classes of features are a looser grouping of those features than aggregation in two respects. First, semantic class information may be ignored by a feature constructor function (e.g., when performing knowledge-less constructions). Second, semantic class knowledge may be used to justify the creation of an aggregate type (e.g., creating a set of features that measure different aspects of one object).

6.3 ALGORITHMIC DESCRIPTION

6.3.1 Beam Search Algorithm

Beam search is a search strategy that compensates for exponential growth in a search space by maintaining a fixed-size set of nodes representing the best of the fringe of the currently explored portions of the search space. Given any current beam of fringe nodes, the search algorithm will ask a successor function to produce these nodes' successors, apply some ranking criterion to the new nodes, and trim off those nodes that are not among the *n*th best.

Code for the beam search of ISAK is given in Figure 30; the following lines numbers refer to this figure. An analysis of the complexity of this code is deferred to Section 6.4.1. In the case of search through the space of constructable features by ISAK, a node in the search space represents a particular constructed feature. The nodes available at the beginning of ISAK's search are the base features. The base features are segmented (line 50) into new keepers and new others based on whether they pass the tree and evaluation constraints (Table 19) for acceptable features. For the initial round of feature generation, the base features are expanded (lines 52-55) with *expandFringe* once and then an iterative process repeats until no new features are generated. At the beam search level, there are three sets of keepers (i.e., acceptable features) and others (i.e., features usable as building blocks): new (*newKeepers* and *newOthers)*, old (*oldKeepers* and *oldOthers*), and temporary (*tmpKeepers* and *tmpOthers*). When new features are used for expansion, they are merged into the appropriate old sets. The merge is done to maintain ordering by the feature evaluation metric (i.e., information gain).

In the iterated case (lines 57-63), the old and new sets are repeatedly expanded until the expansion process produces no new features. At the fringe expansion level of code, there is an additional set of features, current (*curKeepers* and *curOthers).* The successor function (see Section 6.3.4) generates a new pool of potential features (line 34) from the feature constructor functions, the old and current features, and the successor parameters. This pool of features is pruned by tree characteristics (line 35). The surviving features are instantiated with values as specified by the computable function in the FCF (line 36). The instantiated features are then pruned for duplicate

```

def getModels(formulas, baseFeatures, successor, succParams,
              treeConstraints, listConstraints, evaluationConstraints):

        def segment(features):
                partition features into newKeepers which pass
                interesting and newOthers which are not

        def interesting(model):
                evaluate for satisfied tree and evaluation constraints

        def pruneByTree(model):
                evaluate for unsatisfiable tree constraints

        def pruneByValues(model):
                evaluate for duplicate data values against other features

        def fillValues(m):
                evaluate the model at the values of each example
                and create the new data values

        #
        # oldKeepers are "stale" ftrs that are interesting
        # curKeeprs are "fresh" ftrs that are interesting
        #
        # oldOthers are "stale" ftrs that are midling
        # curOthers are "fresh" ftrs that are midling
        #
        def expandFringe(formulas,
                         oldKeepers, curKeepers,
                         oldOthers,  curOthers):
                oldPool = trim(merge(oldKeepers, oldOthers), listConstraints)
                curPool = trim(merge(curKeepers, curOthers), listConstraints)

                genModels = successor(formulas, oldPool, curPool, succParams)
                prunedOnModel = filter(genModels, pruneByTree)
                modelWithData = apply(fillValues, prunedOnModel)
                prunedOnData = filter(modelWithData, pruneByValues)

                for m in prunedOnData:
                        if interesting(m):
                                m.finalize()
                                newKeepers += m
                        else:
                                newOthers += m

                sort(newKeepers)
                trim(sort(newOthers), listConstraints)
                return newKeepers, newOthers

        newKeepers, newOthers = segment(baseFeatures)

        oldKeepers, oldOthers = newKeepers, newOthers
        newKeepers, newOthers = expandFringe(formulas,
                                             [], newKeepers,
                                             [], newOthers)

        while newOthers != [] or newKeepers != []:
                tmpKeepers, tmpOthers = newKeepers, newOthers
                newKeepers, newOthers = expandFringe(formulas,
                                                     oldKeepers, newKeepers,
                                                     oldOthers, newOthers)
                oldKeepers, oldOthers = merge(oldKeepers, tmpKeepers),\
                                        merge(oldOthers, tmpOthers)
        finalKeepers = merge(oldKeepers, newKeepers)
```

Figure 30: ISAK Beam Search Pseudo-code. *filter(l,p)* applies the predicate *p* to each element of the list *l* and returns those elements of *l* that pass *p*. *apply(f, l)* applies the function *f* to each element of the list *l* and returns the resulting list. The tree, list (beam and set), and evaluation constraints are shown in Table 19. The term *model* is used in the implementation as a term for a constructed feature during its instantiation. See also Figure 9 for a graphical depiction of the beam search process.

values (i.e., an instantiated feature has the same values, pairwise, over the examples as a previously generated feature) and pruned with respect to evaluation constraints (line 37).

The features available at this point are partitioned into new keepers and new others[5]. Finally, the new keepers are sorted with respect to information gain evaluation. The new others are sorted and trimmed to the maximum other parameter.

6.3.2 Formula and Value Pruning

Pruning based on the form of the constructed features is based on the following tree-derived characteristics:

1. the height of the tree (i.e., number of operations),
2. the number of leaves in the tree (i.e., number of base features), and
3. the number of unique leaves in the tree (i.e., number of unique base features).

Pruning based on value is done after pruning based on form to prevent the instantiation of features over many examples. There are no user-defined pruning settings for data values. The current purpose of value-based pruning is to remove duplicate features from the feature search space. A new feature will be pruned, if its data values for each example are found to be identical to the respective values of a previously generated feature. For example, if a feature a_1/a_7 has been generated previously and a new feature a_1a_4/a_7a_4 is generated in the current round, the two features will have the same values up to floating point errors. Thus, ISAK recognizes the fact that the new feature is equivalent to the old feature. The process is implemented in ISAK by creating an *md5* hash of a string representation of the values for each example (see Figure 31) and detecting duplicate hash values.

The detection of duplicates is necessary because even with an ordered generation of features, the equivalence of different arithmetic forms will create duplicate entries. As the growth of expressions is combinatorial, even a few duplicates early in the feature space search will create a heavy processing burden deeper in the search. For example, if we have addition and multiplication, we can generate $a(a + b)$ before $a(b + a)$ in a depth-first search since $a < b$ lexicographically. How-

[5]For simplicity, the removal of features by the junk measure search parameter is not shown.

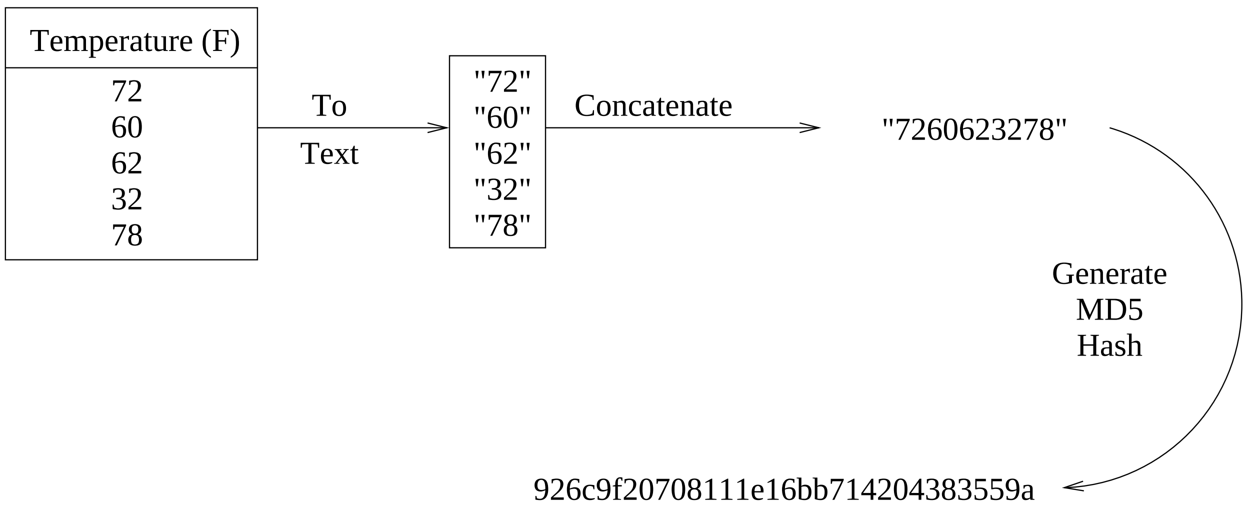


Figure 31: Duplicate Feature Checking via MD5 Hash. The use of quotes denotes a string value versus a numeric value. The purpose of duplicate checking is to remove features that have equivalent tree structures (Section 4.4).

ever, both expressions evaluate to $a^2 + ab$. So, generation in a canonical order does not eliminate the generation of duplicates.

Another option, not implemented by ISAK, to reduce duplication is to perform a canonical simplification on each expression before evaluating it. The canonical form would be compared to a database of previously examined forms and duplicates could be immediately pruned. A problem with the simplification approach is that well-known canonization procedures operate on formal mathematical fields. Therefore, feature constructors that are not field operators and features that are not field elements will cause the canonization procedure to fail. While some datasets can be expressed in terms of a field of elements and operations, many would require at least two fields (e.g., a dataset with Boolean and arithmetic operations).

6.3.3 Evaluation Pruning

Once a feature is found to satisfy type constraints and is not a duplicate of another feature, an evaluation measure is generated on that feature that is an indication of its usefulness in relation to predicting a target value. Information gain is implemented and used in ISAK; other measures can be provided as a parameter to ISAK. The use of information-theoretic measures (e.g., information gain) generally requires a discretizing step for continuous-valued attributes and classes. The discretization is performed by Fayyad and Irani's maximum entropy discretization method (Fayyad and Irani, 1993) as implemented in Orange.

Information gain, also known as the Kullback-Leibler divergence, is a common metric used to select attributes in machine learning tasks (e.g., C4.5 (Quinlan, 1995) uses information gain to choose attributes for decision tree nodes). The information gain of an attribute a is the expected reduction in entropy of the target class that comes from knowing a. Less formally, information gain provides a measure of the relevance of the attribute a in predicting the target class. Many feature selection metrics result in similar rankings of features (Ben-Bassat, 1982). Thus, the choice of metric is not of primary importance.

Based on the information gain, three types of pruning are performed. First, a comparison between the current feature and its parent features determines if the new feature is an improvement over the parent feature or features. If the new feature is not a sufficient improvement, it may be discarded. The improvement constraint introduces a hill-climbing aspect to the feature space search. Second, there is a threshold below which the feature is considered hopeless and should be discarded immediately. Third, the features are ranked according to their evaluations and the best n (where n is the beam size) are retained. The final pruning implements the beam size restriction of beam search.

6.3.4 Successor Function for the Beam Search

The purpose of the successor function is to produce a new beam of constructed features from a current beam of features. The basic generation process has two sub-tasks: selecting features to combine and applying operations to the features. Either sub-task may drive the overall successor generation, or in an agenda driven system both may be considered as selection and application are intertwined. The least restrictive successor – an exhaustive generation of all features specified by the feature and the feature constructor functions – will apply each FCF to all possible permutations

```
def topNGen(kbformulas, oldFeatures, curFeatures, N):
    newTypeToFeature = best N of curFeatures
    oldTypeToFeature = best N of oldFeatures

    for _f in kbformulas:
        for ftr in matchFormulaToFeatures(_f,
                                          oldTypeToFeature, newTypeToFeature):
            yield ftr

def randomNGen(kbformulas, oldFeatures, curFeatures, N):
    newSample = uniform sample(curFeatures, N)
    oldSample = uniform sample(oldFeatures, N)

    for _f in kbformulas:
        for ftr in matchFormulaToFeatures(_f,
                                          oldTypeToFeature, newTypeToFeature):
            yield ftr
```

Figure 32: Top-N and Random-N Successor Functions. *yield* is a method of returning a closure so when the function is called again, the next feature will be returned. *Best* in Top-N refers to the top N feature in an ordering of features by information gain. The Top successor represents the heuristic that the best features (for construction) are those with the highest evaluation. The Random successor implements the heuristic that the uniform random choice of features provides a good source of variety in the pool of features used for construction.

of the current beam features. The permutations represent different arguments to the functions. Heuristics are used to create more restrictive generators (see Figures 32 and 33).

Selection of features for feature construction may be implemented in two ways: random and ordered. A random selection method defines a probability distribution over the features and selects features according to that distribution. Ordered selection involves ranking the features and making use of that ordering. In either case, a subset of the features may be selected. Ordered selection and random selection may be combined by making the probability distribution a function of the ordering or evaluation of the features.

ISAK implements two simple successor functions that pick *n* features from the set of current features and use those as the base for constructions. The Top-N successor uses an ordering heuristic to guide the choice of features. It takes the top *n* old features and the top *n* new features with respect to a ranking criterion and uses those *2n* features as the base for a round of construction. Random-N uses randomness to choose features. It picks *2n* features from a uniform random distribution, *n* from the old features and *n* from the new features. Should there be less than *n* features in either set,

```

def joinedBallGen(kbformulas, oldFeatures, curFeatures, maxCross, hardLimit):

    allFtrs = best hardLimit features of sort(oldFeatures + curFeatures)

    for ftrSet in windows(allFtrs, maxCross, hardLimit):
        cur = features in ftrSet that are from curFeatures
        old = features in ftrSet that are from oldFeatures

        for _f in kbformulas:
            for ftr in matchFormulaToFeatures(_f, old, cur)
                yield ftr

def sepBallGen(kbformulas, oldFeatures, curFeatures, maxCross, hardLimit):
    curFeatures = best hardLimit of curFeatures
    oldFeatures = best hardLimit of oldFeatures

    for cur in in windows(allFtrs, maxCross, hardLimit):
        for old in in windows(allFtrs, maxCross, hardLimit):
            for _f in kbformulas:
                for ftr in matchFormulaToFeatures(_f, old, cur)
                    yield ftr
```

Figure 33: Pseudo-code for the Windowed Successor Generators. The window function takes a list of features ordered by evaluation metric and generates windows of size *maxCross* from the *hardLimit* best features. *maxCross* is the size of the set of features that are potentially combined with each other; *hardLimit* is the size of the larger set from which these subsets are drawn.

the entire set is selected and less than *2n* total features are used. Pseudo-code for these successors is shown in Figure 32.

An important utility function in the Top, Random, and other successors is *matchFormulaToFeatures.* The matching function is responsible for generating each legal combination of FCFs and features from the given sets of new features, old features, and FCFs. The segmentation of features into old and new sets prevents regeneration of features in the old set by only considering combinations that involve one or more new features. Specifically, we do not need any combinations of only old features, because these have been dealt with in previous rounds. Since the new features did not exist in previous rounds, the features generated from them must be new[6]. The matching procedure generates all feature-FCF combinations with one new feature, then two new features, up to entirely new features. Thus, it must also generate every possible combination of FCF and feature with one or more new features. Pseudo-code for *matchFormulaToFeatures* is given in Figure 34. The complexity of the matching procedure is described in Section 6.4.2.

[6]Though the new features may be algebraically or numerically equivalent to an old feature.

Matching (lines 23-38) proceeds by putting each new feature in each argument spot and then filling in old features to the left of that spot and unused features to the right of that spot. Unused features are specifically features in the new and old sets that have not been used as arguments in the current formula (i.e., to the left of the current position being filled). Features that satisfy a given type are found in constant time by look-up in a hash table (the *oldTTF*, *newTTF*, and *allTTF*) that maps a semantic type to a list of features of that type. After type matches are found, the semantic properties of those types are checked in the second part of the code (lines 44-49) and, if the constraints pass, new feature are generated and returned (lines 50-51).

6.3.5 Windowed Selection of Features in the Successor Function

Joined Ball and Separate Ball are two more complicated generators that combine features with similar evaluations (i.e., rankings) but that are not necessarily at the top of the ranking. Both generators rely on considering a sliding window of ranked features. Joined Ball ranks all of the features at once and then considers windows of these features. Separate Ball ranks the features separately (i.e., as old features and as new features) and then generates successors from the cross-product of windows from each set. Pseudo-code for the windowed generators is shown in Figure 33.

Both methods take two window parameters: a window size and a total number of features to use from the list of all features. The window size, *maxCross*, determines the maximum number of features that can be considered at one time. From the window size and the total number of features to be windowed, a number of window can be computed, but does need to be explicitly known. The slide amount is fixed at $maxCross/2$ and is the number of features skipped, moving down the numerical ordering; the number of windows determines how many steps to take.

For example, with features $f_1, ..., f_{100}$ ordered by information gain, five windows of size 10, and a slide value of two, the windowing procedure generates the following sets of features to be processed by the matcher for use as arguments:

$$\{\{f_1, ..., f_{10}\}, \{f_3, ..., f_{12}\}, \{f_5, ..., f_{14}\}, \{f_7, ..., f_{16}\}, \{f_9, ..., f_{18}\}\}$$

where each of the sets is of size 10.

```
 0:
 1: # inUsed is a dictionary of used features (constant time lookup)
 2: def recTypeMatch(types, inUsed, typeToFeature):
 3:     if types == tuple():
 4:         yield []
 5:     else:
 6:         # this gets all features matching the given type
 7:         for i in typeToFeature[types[0]]:
 8:             if i not in inUsed:
 9:                 nowUsed = inUsed.copy()
10:                 nowUsed[i] = None
11:                 for s in recTypeMatch(types[1:], nowUsed, typeToFeature):
12:                     yield [i] + s
13:
14:
15: # oldTTF, newTTF are mappings from types to features
16: def matchFormulaToFeatures(formula, oldTTF, newTTF):
17:
18:     newSpots = len(formula.types)
19:     for i in range(newSpots):
20:         allTTF = oldTTF + newTTF
21:
22:         # newFtr is any new feature
23:         for newFtr in newTTF[formula.types[i]]:
24:             matches = []
25:             # have to loop through spots of multi-arg formulas
26:             if len(formula.types) > 1:
27:                 # left side is old only
28:                 for oldSide in recTypeMatch(formula.types[:i],
29:                                             {},
30:                                             oldTTF):
31:                     used = oldSide + newFtr
32:
33:                     # right side is any (old or new)
34:                     # that is unused
35:                     for allSide in recTypeMatch(formula.types[i+1:],
36:                                                 used,
37:                                                 allTTF):
38:                          matches.append( oldSide + newFtr + allSide )
39:
40:             # one spot, so use this
41:             else:
42:                 matches.append((newFtr),)
43:
44:             #
45:             # if this matching tuple satisfies the constraints
46:             # ask the formula to produce new features from it
47:             #
48:             for args in matches:
49:                 if formula.checkDetails(ftrTuple):
50:                     for generatedFtrs in formula.getNew(ftrTuple):
51:                         yield generatedFtrs
```

Figure 34: *matchFormulaToFeatures* Pseudo-code. *recTypeMatch* simply generates all tuples of features whose types match the types needed by a portion of type signature for a feature constructor. Matching (lines 23-38) proceeds by putting each new feature in each argument spot and then filling in old features to the left the *new* argument with types that match the left-most arguments in the FCF and unused features to the right of the *new* spot with types that match the right-most arguments of the FCF.

The heuristic approach of taking the best features with respect to a feature evaluation metric has particular problems when the selection is done for generalized feature construction. Those features which have the highest evaluation with respect to the target class are not necessarily the features that will make the most learning improvement when used in constructions. In fact, if we consider a tree representation of a target concept, the root node is the closest, most predictive value to the target value. The features we want to bring together are not necessarily those with the best individual relationship to the target class, but those features that are close to each other in the construction tree.

Specifically, we wish to choose pairs or tuples of features that share a parent node in the construction tree. As a heuristic measure of this distance, we can turn back to the features' relationships to the target class. Instead of combining the group of the highest evaluated features, we combine groups of closely evaluated features. The closeness indicates a similarity of level in the construction tree and a potential match as operands to an operation.

6.3.6 Type and Constraint Processing

Type processing is handled in the matching procedure described in Section 6.3.4. By generating a hash mapping of types to features, we can look-up all features of a given type in constant time. Constraint processing is handled explicitly in the matching procedure and is described here. Constraints are Boolean functions that operate on the semantic properties of instantiated types or directly on features. Typing constraints check for object similarity, dimensional compatibility, and any more detailed semantic property. Feature constraints implement restrictions on feature combinations that are not necessarily semantic in nature.

Every feature constructor function has a list of constraints that must be met before that FCF will generate a new feature from a given argument tuple. The entire argument tuple is passed to each constraint. The constraints may be any computable function that returns a Boolean value. It is preferable that the constraints operate in constant time. However, unification constraints on semantic classes execute in time dependent on the hierarchy in which the semantic class being checked against exists.

6.3.7 Output Type Generation

If a formula produces a single feature, there is only one output type and there is one function given to produce output typing properties from the input features. The output properties are a function of the inputs types, the input properties, the constructor function, and, possibly, the values of the the input and output features.

If the FCF supports multiple outputs for a given input, then the formula typing function and the output typing properties function are actually a family of functions. The family of functions is defined by a function that produces a separate typing function for each new feature based on the feature constructor function and the old features.

6.4 COMPLEXITY OF BEAM SEARCH AND MATCHING

6.4.1 Complexity of Beam Search

The complexity of the beam search portion of ISAK (Figure 30) can be expressed in terms of

1. the number of features generated by the successor function, G,
2. the number of examples in the dataset being modified, E,
3. the maximum time to apply any formula to one example, T,
4. the maximum number of arguments to any feature constructor function, A.

Now, pruning the generated models[7] on model characteristics takes time constant in G because the model values (e.g., height and number of leaves) are evaluated as each model is generated and there is a constant number of value comparisons done for each model.

Evaluating the data for each example requires gathering values, applying a formula to the values, and computing the heuristic measure from new values for each of the G models. Gathering the values for each attribute in the formula over all of the formulas which is bounded above by EA[8]. Applying the function in the formula for each example is bounded by TE. The worst case,

[7]A model is a new feature in the process of construction. Specifically, it may be missing values and semantics.

[8]There is no recursion because each attribute stores its own data values directly, without reference to its constituent parts.

when computing information gain as the feature evaluation measure, is a continuous feature that must be discretized which requires a call to maximum entropy discretization routine (Fayyad and Irani, 1993) at a time complexity of $E \log E + E$. Once discretized values are present, another E operations are required to compute the contingency table of values and determine the information gain for the feature. Over G new features, the complexity to set the data values and information gain is bounded by

$$GEA + GET + G(E \log E + E) + GE = GE(A + T + \log E)$$

Note that G grows at a variable rate based on the successor function. Checking for duplicates is implemented by building an md5 hash of the values for each G requiring time linear in E and checking against values in a hash table requiring constant time. Duplicate checking takes time GE.

Determining interestingness is bounded by the number of parents (i.e., the number of arguments, A) as the other checks are constant time. *Finalize* is also bound by A. *Forgettable* takes constant time for each feature. So, the final loop has time complexity GA. The sorting step on the two lists with length $n + m \leq G$ is bounded by the time to sort that many elements total, which is just $G \log G$.

Thus, an upper bound on the total time complexity for one round of feature generation is

$$\begin{aligned} G + GE(A + T + \log E) + GA + G \log G &= \\ G(E(A + T + \log E) + A + \log G) \end{aligned}$$

6.4.2 Complexity of Matching Feature to Feature Constructor Functions

The matching function (see Figure 34) takes formulas specified in part by their type signature and mappings from types to features. Thus, given a type slot, all features that match this type can be found in constant time. A bound on the number of matches generated can be computed by segmenting the argument signature into three parts: old $\hat{O}$, new $\hat{N}$, and any except those already used in new or old, $\hat{A}$. Letting the number of old features vary from zero to $k - 1$ (k is max number of arguments) and requiring one new feature at each step, gives us the following form for the number of matches with full replacement

$$\sum_{s=0}^{k-1} \hat{O}^s \hat{N} \hat{A}^{k-s-1}$$

However, in the matching procedure we track the use of features and do not allow replacement because it allows subsequent rounds of generation to produce duplicates[9]. So, without replacement the three components become:

1. Old side: fill *s* arguments from $\hat{O}$ old features which involves choosing *s* features and ordering them giving $\begin{pmatrix} \hat{O} \\ s \end{pmatrix} s! = \frac{\hat{O}!}{(\hat{O}-s)!}$,
2. New slot: fill 1 argument from $\hat{N}$ features giving $\hat{N}$, and
3. Any side: fill $k-s-1$ arguments from the remaining $\hat{N}-1$ and $\hat{O}-s$ features and order them giving:

$$\begin{pmatrix} \hat{N}+\hat{O}-s-1 \\ k-s-1 \end{pmatrix} (k-s-1)! = \frac{(\hat{N}+\hat{O}-s-1)!}{(\hat{N}+\hat{O}-k)!}.$$

Multiplying the three counts for Old, New, and Any and then summing the counts over all possible positions of the guaranteed New argument gives:

$$\sum_{s=0}^{k-1} \frac{\widehat{O}!}{(\hat{O}-s)!} \hat{N} \frac{(\hat{N}+\hat{O}-s-1)!}{(\hat{N}+\hat{O}-k)!} = \frac{\hat{N}\hat{O}!}{(\hat{N}+\hat{O}-k)!} \sum_{s=0}^{k-1} \frac{(\hat{N}+\hat{O}-s-1)!}{(\hat{O}-s)!}$$

In the case of $k=2$ (i.e., for binary operations), the summation simplifies to:

$$\frac{\hat{N}\hat{O}!}{(\hat{N}+\hat{O}-2)!}\left[\frac{(\hat{N}+\hat{O}-1)!}{\hat{O}!} + \frac{(\hat{N}+\hat{O}-2)!}{(\hat{O}-1)!}\right] = \frac{\hat{N}\left(\hat{N}+\hat{O}-1\right)!}{(\hat{N}+\hat{O}-2)!} + \frac{\hat{N}\hat{O}!}{(\hat{O}-1)!} = \hat{N}\left(\hat{N}+\hat{O}-1\right) + \hat{N}\hat{O}$$

[9]With replacement, from the set of features a, b and $add(.,.)$ we generate: $a+a$,$a+b$, and $b+b$ in the first round. In the second round we generate: $a+a+b$, $b+b+b$, $a+b+b$, $a+a+a$, $b+b+a$, and $a+b+a$. In the second round, $2a+b$ and $a+2b$ occur twice.

In the case of $\hat{N} = 3$ and $\hat{O} = 2$, we would generate the following argument tuples

$$\{(n_1, any), (n_2, any), (n_3, any), (old, n_1), (old, n_2), (old, n_3)\}$$

where n_i is the i^{th} element of $\hat{N}$, *any* is any element of new or old except n_i and *old* is any member of old. Counting the possibilities gives $4 + 4 + 4 + 2 + 2 + 2 = 18$ argument tuples. By formula, we get $3(3 + 2 - 1) + 3 \cdot 2 = 18$.

Each argument tuple generated is given to each formula, multiplying the number of tuples by $\hat{F}$.

The generated matches are then checked against the formula constraints and then return the features specified by the formula for the match. Generally there is only one returned feature but there are possibly more as in the case of a formula that specifies testing each value of a symbolic feature.

6.4.3 Total Complexity for Top and Random Successor Functions

The complexity of the Top-N and Random-N successors is a function of the number of selected features, *2N* (*N* old and *N* new), given as input to the matcher and the complexity of the matching process. The time for generation is bounded by and the number of features generated from these two functions is

$$N\,(N + N - 1) + NN = 3N^2 - N.$$

Taken over each formula, these two successor functions are bound by $\hat{F}N^2$.

When the Top-N or Random-N generators are used as the successor function for beam search, one round of beam search is bounded by

$$\hat{F}N^2\left(E\,(A + T + \log E) + A + \log \hat{F}N^2\right).$$

7.0 EXPERIMENTS

7.1 A SET OF EXPERIMENTS DESCRIBING THE BEHAVIOR OF ISAK ON MULTIPLE LEARNING PROBLEMS USING MULTIPLE SUCCESSOR FUNCTIONS AND FEATURE SETS

7.1.1 Construction and Learning Problems: Datasets, Knowledge Bases, and Search Parameters

Each construction and learning problem (CLP) is defined by three pieces of information: a dataset, a knowledge base, and a set of search parameters. Table 20 and Table 21 contain summary information for each CLP. The balance, monks, tic-tac-toe, iris, promoter, and wine datasets are taken from the UCI Machine Learning Repository (Newman et al., 1998). The isosceles and soccer offsides datasets were randomly generated following descriptions of their target concepts in the FICUS paper (Markovitch and Rosenstein, 2002). The goal of the isosceles problem is to classify a given triangle as isosceles or not on the basis of the lengths of its sides. For offside, the goal is to determine if a given configuration of soccer players, described by their x- and y-coordinates, has a player offsides. The offsides datasets have 50% irrelevant features because the players' y-coordinates do not matter to the definition of offsides.

The three promoter CLPs and the two tic-tac-toe CLPs share their respective datasets but have different knowledge bases (KBs) and search parameters. The offsides problems have the same top-level concept (i.e., they are top-level equivalent) but differ in the number of leaf features based on the problem size. Offsides2, for example, has 2 players per team and 8 total coordinates.

The three promoter CLPs use two different knowledge bases. The first KB treats the features – nucleotides at different positions in a DNA promoter sequence – as symbolic variables and con-

Table 20: Description of Closed Form Datasets. The TTT operations are not sufficient to generate the target concept for those problems. All other problems have a sufficient set of operations to generate the target concept. *Min Ops* is the minimum number of operations (non-leaf nodes) necessary to construct the target concept (or the closest partial target concept for the TTT problems). *Min Height* is the longest path in the constructed tree for the target concept. Details of the FCFs are shown in Figure 17.

CLP Name	Feature Names	Feature Description	Features	Value Type(s)	Target Classes	Examples	Available Feature Constructor Functions	Target Concept	Target Description	Min Ops	Min Height
Balance	l-weight, l-dist, r-weight, r-dist	Weight and distance of dangle from pivot	4	Continuous	3	625	genericMul, ternary comp	tcomp(l-weight * l-dist, r-weight * r-dist)	Is the scale balanced?	3	2
Isoceles	side_a, side_b side_c	Length of sides of triangle	3	Continuous	2	100	initBag, addBag, bagNotUniq?	bagNotUniq({side_a, side_b, side_c})	Is the triangle isoceles?	3	3
Monks1	a1, ..., a6	Symbolic values	6	Symbolic	2	556	eqTestAllSym, symEq, boolAnd, boolOr	(a1=a2) or (a5 = 1)		3	2
Monks2	a1, ..., a6	Symbolic values	6	Symbolic	2	601	initBag, addBag, eqTestAllNums, aggSymCounts	twoOf(a1=1, a2=1, a3=1, a4=1, a5=1, a6=1)		13	13
Monks3	a1, ..., a6	Symbolic values	6	Symbolic	2	554	eqTestAllSym, notEqTestAllSyms, boolAnd, boolOr	(a5 = 3 and a4 = 1) or (a5 != 4 and a2 != 3)		7	4
Offsides2	o1x, o2x, b1x, b2x,o1y, o2y, b1y, b2y	Positions of players from two teams on a soccer field	8	Continuous	2	100	initBag, addBag, aggMin, aggMax, continuous >	max({o1x, o2x}) > max({b1x,b2x})	Is an orange player offsides?	7	4
Offsides5	o1x, ..., o5x, b1x, ..., b5x, o1y, ..., o5y, b1y, ..., b5y	Positions of players from two teams on a soccer field	20	Continuous	2	100	initBag, addBag, aggMin, aggMax, continuous >	max({o1x, ..., o5x}) > max({b1x, ..., b5x})	Is an orange player offsides?	16	7
Offsides11	o1x, ..., o11x, b1x, ..., b11x, o1y, ..., o11y, b1y, ..., b11y	Positions of players from two teams on a soccer field	44	Continuous	2	100	initBag, addBag, aggMin, aggMax, continuous >	max({o1x, ..., o11x}) > max({b1x, ..., b11x})	Is an orange player offsides?	34	13
TTT1	tl, tm, tr, ml, mm, mr, bl, bm, br	Pieces on a tic-tac-toe board	9	Symbolic	2	958	twoPieceCount, countPlusPiece	countX(tl,tm,tr) == 3 or countX(ml,mm,mr) == 3 or ...	Is the board a win for player 1 or not?	32	4
TTT2	tl, tm, tr, ml, mm, mr, bl, bm, br	Pieces on a tic-tac-toe board	9	Symbolic	2	958	eqTestAllSym, initBag, addBag, aggCtTrue	ctTrue({eqx(tl), eqx(tm), eqx(tr)}) == 3 or ...	Is the board a win for player 1 or not?	41	4

Table 21: Description of Open Form Datasets. Values in parentheses indicate a change in the feature set before the feature space search commences. The change is explicitly specified by the KB file for that problem. Details of the FCFs are shown in Figure 17.

CLP Name	Feature Names	Feature Description	Number of Features	Value Type(s)	Number of Classes	Number of Examples	Available Feature Constructor Functions
Iris	sepal_width, sepal_length, petal_width, petal_length	Characteristics of Iris Petals	4	Continuous	3	150	lenAdd, lenSub, lenMul, lenDiv
Promoter1	p-50, p-49, ..., p+07	Nucleotides in a DNA Sequence	57	Symbolic	2	106	eqTestAllSym, initBag, addBag, aggCtTrue
Promoter2	p-50, p-49, ..., p+07	Nucleotides in a DNA Sequence	4 (57)	Symbolic	2	106	match35s, match10s
Promoter3	p-50, p-49, ..., p+07	Nucleotides in a DNA Sequence	61 (57)	Symbolic	2	106	eqTestAllSym, initBag, addBag, aggCtTrue, match35s, match10s
Wine	a1, ..., a13	Measurements on Wine Samples	13	Continuous	3	178	unifAdd, unifSub, unifMul, unifDiv

structs Boolean tests, bags, and counts on bags. In short, the first KB simply treats the values as arbitrary symbols. The second KB has detailed knowledge about two contact regions within the promoter sequence[1]: the feature constructor functions try to match the regions to known promoter sequences[2]. The third CLP uses the union of these two KBs. The tic-tac-toe problems also make use of two different types of knowledge. The first is a problem specific KB with integrated knowledge of rows, columns, diagonals, and winning criteria. The second KB for tic-tac-toe contains general symbolic operations similar to the first promoter KB.

The CLPs are divided into two categories: open-form and closed-form. The datasets labeled closed-form have a known closed-form representation of the target concept in terms of the given base features. For each of the closed-form CLPs except tic-tac-toe, the feature constructors are sufficient to construct the target concept. In the case of the tic-tac-toe problem, the given feature

[1]Each example describes a promoter sequence for a single individual.

[2]There is also conformational knowledge available for the promoter problems. However, the conformation knowledge did not appear to be useful within ISAK's constructions and was not used in the experiments. The original UCI promoter problem is an ILP style learning problem.

Table 22: Search Parameters. The figure shows the search parameters for the standard set of CLPs. The semantics of the search parameters are discussed in Section 6.2.6.

CLP Name	Minimum Measure	Minimum Measure Improvement	Junk Measure	Minimum Features	Minimum Leaves	Minimum Height	Maximum Features	Maximum Leaves	Maximum Height	Maximum New	Maximum Return	Maximum Old Features
Balance	0.3	0.01	0	1	1	2	4	4	3	20	10	20
Iris	1.5	0.1	0	1	1	1	4	4	3	100	10	undef
Isoceles	0.5	0.01	0	1	1	1	3	3	4	100	10	undef
Monks1	0.3	0.01	0	1	1	1	3	3	3	100	15	undef
Monks2	0.01	0.01	0	1	1	1	6	6	7	100	15	undef
Monks3	0.5	0.01	0.03	1	1	1	4	4	4	75	5	undef
Offsides2	0.25	0.05	0	1	1	1	4	4	6	50	5	undef
Offsides5	0.1	0.05	0	1	1	1	10	10	7	150	5	undef
Offsides11	0.25	0.05	0	1	1	1	4	4	6	50	5	undef
Promoter1	0.4	0.01	0.01	1	1	1	10	10	10	100	10	100
Promoter2	0.25	0	0	2	1	1	10	10	4	100	10	undef
Promoter3	0.25	0	0	2	1	1	10	10	4	100	10	undef
TTT1	0.08	0.01	0	3	1	1	4	4	3	100	10	undef
TTT2	0.08	0.01	0	1	1	1	10	10	10	50	10	100
Wine	1	0.3	0.1	1	1	1	4	4	3	75	10	75

constructors are sufficient to generate a partial target concept. Note that just because the feature constructor functions are sufficient to lead to the target concept or a partial target concept, the path to the goal may not be found. Also, the constructed concepts may only be equivalent (not necessarily identical) to the target concept; the application of operations may be more or less complex than the order in the given form of the target concept.

The search space parameters for each CLP are shown in Table 22. The search parameters are chosen based on the knowledge in the KB file and knowledge or guesses of the form of the target concept. At a minimum, search parameters were chosen to allow an adequate number of FCF applications. For example, in the case of the tic-tac-toe problems, the form of the target concept is known and the depth of search necessary to construct features that encapsulate the definition of a win can be calculated. Search parameters can also be manipulated to control the rate of growth of the space of explored features.

The CLPs were initially chosen for comparison with the experiments in FICUS (Markovitch and Rosenstein, 2002). Unfortunately, I was unable to implement a working version of FICUS for use in direct experimental comparisons, because some methodological variables in the FICUS experiments are unknown. Hence, there are differences between the baseline results in my work and in the FICUS paper. The use of mostly toy-level problems is justified on the basis of (1) knowledge of the form of some use features in the problem domains, (2) comparison with results in the literature, and (3) accessibility of the datasets for comparison purposes. As with other research involving the UCI datasets, these experiments contribute to the trend of overfitting the UCI datasets. This is partially offset by utilizing several non-UCI datasets (i.e., the isosceles and soccer CLPs) and by performing additional experiments on novel problems (Section 7.3.1).

7.1.2 Generation of Synthetic Datasets

An auxiliary synthetic data generation program (SDG) takes an input specification of features, distributions of values for the features, relationships among the features, and constraints. The SDG outputs a dataset satisfying the specification. The data generator was used to produce the isosceles, offsides, and weather datasets. The synthetic area datasets used in later experiments were also produced by the SDG.

Feature values are defined in the generator via a probability distribution of values, a file specifying specific values, or as a function of other features. The data type of a value may be any Python data type, but typically it will be of type float, integer, Boolean, or string. Features defined by a distribution or a file are called base features. Features created from base features by functions are called derived features. The target feature is some function, perhaps via the application of several other functions, of the base features. The output file contains the base and derived features that the user specifies for output and it also contains the target feature. The output features are the features that will be available to the learning algorithm when it processes the synthetic dataset. If an input file is used to define any features, then the size of the output dataset is fixed to the number of examples in the file; otherwise, the user may specify a number of test and training examples to produce.

Functions used to create derived values are defined in Python syntax and may use any built-in functions from the Python language. Functions may also be arbitrary, user-defined Python functions (including functions that generate random values). The set of all functions applied to create a target value from the base features is called a model. Multiple models may be defined and applied to proportions of the data (e.g., for 20% of the data, the target concept is $f_1 + f_2$; for 80% of the data, the target concept is $f_1 - f_2$).

Constraints are Boolean functions that apply to a single created example and enforce relationships among the values of the example's attributes. Satisfiability of constraints is not checked; random feature values are generated and derived feature values are computed until satisfactory examples are generated. A progress bar allows the user to determine if insufficient progress is being made due to unsatisfiable (or low-likelihood of satisfiability) constraints. There is no mechanism for enforcing constraints between examples.

An example synthetic problem specification is given in Appendix B.

7.1.3 Learners

The experiments make use of seven different learning algorithms implemented in the Orange data mining system (Demsar et al., 2004). The algorithms are:

1. Majority (MAJ) classifies each example according to the most frequent class in the training data.
2. K-Nearest Neighbors (KNN), for $k = 3$ (3NN), classifies each example according to the most frequent class of its 3 closest neighbors. Closeness is measured using Euclidean distance for continuous valued features and Hamming distance for symbolic valued features.
3. KNN, for $k = 10$ (10NN), classifies as per 3NN, but an example is classified to the most frequent class of its 10 nearest neighbors.
4. Naive Bayes' (NB) classifies each example to the most probable class c computed from the product of the class probabilities for each attribute considered independently of each other (i.e., the value of c that maximizes $P(C = c) \prod_A P(A = a|C = c)$). NB uses relative frequency as the probability estimate.
5. Tree (TREE) classifies each example using a CART (Breiman et al., 1984) classification tree that (1) is pruned when the maximum majority class is greater than .85 and (2) requires at least 5 examples in each split.
6. Support Vector Machine (SVM) classifies each example using a C-support vector classification machine with a radial basis function (RBF) kernel. The γ parameter of the RBF kernel is $\gamma = 1/\#attributes$. Orange provides SVMs via an interface to libsvm (Chang and Lin, 2006).

7. Forest (FORST) classifies each example using a 50-tree forest learner. Essentially, FORST is an ensemble method gathering independent classifications from each of fifty tree and combining them to form a final classification for the example. For a discussion of the theory of random forest learners, see Breiman (2001).

I choose these learners for two reasons. First, as discussed in Section 3.2, different learners have different biases. Feature construction modifies the dataset; in turn, the dataset interacts with the bias of a learner. It is reasonable to expect feature construction to affect different learners in different ways. In particular, feature construction may result in different changes in generalization performance. Second, the learners I choose represent a variety of types of classification learners. KNN, NB, and TREE are well-known and understood by a variety of researchers inside and outside of the machine learning community. SVM and FORST have recently achieved very good classification results on a variety of problems. KNN and TREE represent different degrees of explicit feature measurement: KNN does not weight distances based on feature relevance and TREE explicitly chooses features at every node split. NB has a theoretical limitation in dealing with dependent features: are constructed features dependent to a degree that hampers NB classification? SVMs perform feature construction as part of their internal operation. Can SVMs be improved by further feature construction? Can learning methods that do not perform internal feature construction be improved by ISAK?

For each of the CLPs, we can establish two baseline standards for classification performance. The first is the performance of a learning algorithm on the original learning problem with no additional features. The second performance baseline is the use of a majority classifier. In the context of varying feature sets, we may interpret a majority classifier as a classifier using *no* features. A majority learner makes no use of features in building its hypotheses; MAJ simply picks the most likely target class, regardless of features, and chooses that as its target concept for all input cases.

Thus, we have a progression of comparison from (1) classification with no features, to (2) classification with the given problem features, and then to (3) learning with the various combinations of constructed features and base features discussed in the next section. Table 23 shows base feature set learning accuracies, computed by 10-fold cross-validation, for each problem described in Section 7.1.1 with these seven learners. It should be emphasized that these, and other baseline results, are achieved with off-the-shelf learning systems using no additional information

Table 23: Base Accuracy Rates for Example Datasets. The table shows the base accuracy rates for 15 CLPs described in Tables 20 and 21. The values are the 10-fold cross-validation accuracies using the original, unmodified features. The mean accuracy and standard error for each problem are computed over each learner excluding MAJ. The standard error computed over the CLPs is somewhat misleading because there are 15 CLPs but only 12 have unique datasets. The differences in the Promoter and TTT problems are in the KBs. Hence, Promoter and TTT have the same base rate for each different CLP. The standard error over distinct CLPs is shown in the bottom row of the table.

CLP Name	MAJ	3NN	10NN	NB	TREE	SVM	FORST	Mean for CLP over Learners Excluding MAJ	Standard Error
Balance	0.46	0.78	0.83	0.90	0.78	0.91	0.80	0.83	0.024
Iris	0.33	0.95	0.95	0.93	0.92	0.96	0.94	0.94	0.006
Isoceles	0.82	0.96	0.93	0.80	0.82	0.81	0.84	0.86	0.028
Monks1	0.50	0.84	0.90	0.75	0.95	1.00	0.94	0.90	0.037
Monks2	0.66	0.79	0.69	0.62	0.65	0.65	0.65	0.68	0.024
Monks3	0.52	0.86	0.93	0.96	0.96	0.99	0.99	0.95	0.020
Offsides2	0.58	0.60	0.64	0.71	0.66	0.66	0.73	0.67	0.019
Offsides5	0.58	0.60	0.62	0.63	0.61	0.68	0.70	0.64	0.016
Offsides11	0.49	0.55	0.50	0.50	0.50	0.50	0.46	0.50	0.012
Promoter1	0.47	0.81	0.86	0.86	0.82	0.91	0.87	0.85	0.014
Promoter2	0.47	0.81	0.86	0.86	0.82	0.91	0.88	0.86	0.014
Promoter3	0.47	0.81	0.86	0.86	0.82	0.91	0.88	0.85	0.014
TTT1	0.65	0.81	0.90	0.70	0.84	0.99	0.90	0.86	0.040
TTT2	0.65	0.81	0.90	0.70	0.84	0.99	0.90	0.86	0.040
Wine	0.40	0.95	0.95	0.98	0.91	0.98	0.97	0.96	0.011
Mean for Learner over CLPs	0.54	0.80	0.82	0.78	0.79	0.85	0.83	0.81	
Standard Error Over All CLPs	0.029	0.032	0.036	0.036	0.034	0.041	0.037		
Standard Error Over Distinct CLPs	0.034	0.036	0.041	0.041	0.038	0.046	0.041		0.0009

outside of the given dataset itself. The only information used to generate the hypotheses that lead to these results are (1) the learning method and (2) the examples in the dataset. These examples are contained in standard attribute-value datasets and, in the case of the UCI datasets, are the exact examples downloaded from the UCI repository.

7.1.4 Use of Base and Constructed Features in Learning

Processing a CLP by ISAK results in constructed features. I used the features from each CLP in one of five different combinations of *base features* and *constructed features* (respectively, those features in the original learning dataset and those feature returned from ISAK). The base features can be partitioned into two sets: those features which are *components of constructed features* (CCFs) and those feature which are not components of constructed features. The second set is called the *remaining initial features* (RIF).

The five combinations are:

1. all of the original base features and none of the constructed features (Base),
2. none of the original base features and all of the constructed features (New),
3. all of the original base features and all of the constructed features (All),
4. all of the RIFs and all of the constructed features (Constructed and Remaining Initial Features, ConRIF), and
5. all of the components of constructed features and none of the constructed features (CCF).

Table 24 shows the composition of the feature sets.

The motivation for combination four, ConRIF, comes from viewing each base feature as a potential source of information that may contribute to the target classification. If a constructed feature makes use of several base features, it encapsulates some of the information contained in those base features. If an important base feature is not used as a component of any of the constructed features, then important information will be missing in the constructed feature set alone (i.e., combination two). Further, using all base and constructed features (i.e., combination three) may result in duplication of information in the feature set. Duplication of information in slightly different forms may promote overfitting.

Table 24: Feature Set Composition. The feature sets used in the first set of experiments are created from combining three sources of features: (1) the components of constructed features (CCF), (2) the remaining initial features (RIF), and (3) the constructed features. The shaded boxes indicate that a particular source is used in that experimental feature set.

	Source of Features		
Feature Set	Base Features		Constructed Features
	CCF	RIF	
Base	▨	▨	
New			▨
All	▨	▨	▨
ConRIF		▨	▨
CCF	▨		

Combination five, CCF, separates the information content in the features (i.e., directly in the CCF set) used for feature construction from the operations performed and the information generated *from* those features (i.e., the constructed features) during feature construction. Feature construction can be viewed as *adding* to the information content of the base features. Then, each of the feature sets that make use of constructed features has two sources of information: the base features used and the constructions made from those features. CCF is meant to allow us to tease apart the contribution from each of these sources.

7.1.5 Successor Functions

There are five successor functions used in this set of experiments: Exhaustive, Top, Random, Joined, and Separate. They are described fully in Section 6.3.4. Exhaustive creates all plausible constructions from the given FCFs and features; Top and Random select a subset of features for construction; Joined and Separate rank features and choose features of similar ranking for use in constructions. In each case, the plausible constructions are subject to pruning in the beam search. In these experiments, each generator is paired with sets of values for its hard limit and maximum cross parameters. These parameters determine the breadth of search within a depth specified by the CLP search parameters. The Exhaustive successor explores the complete breadth of the allowed depth; the hard limit and maximum cross parameters listed for it are the maximum breadth parameters that result from the beam search parameters specified in the CLPs.

For convenience, a successor function and its parameters may be referred to as one unit with the parameters in the order HardLimit and then MaximumCross. Join(5,10) refers to the Join successor function with $HardLimit = 5$ and $MaximumCross = 10$.

7.2 ISAK: CONCEPT DESCRIPTION ABILITIES, BENCHMARK PERFORMANCE IMPROVEMENTS, AND SUCCESSOR FUNCTION EFFECTS

The following sets of experiments demonstrate the ability of ISAK to use knowledge and search for feature construction in several different benchmark problems and to improve classification accuracy on these problems. The experiments also show the results of different successor functions and the effect of parameter variation in these functions.

7.2.1 Experimental Variables, Parameters, and Method

The first set of experiments combines the CLPs described in Section 7.1.1, the learners described in Section 7.1.3, and the methods of using constructed features (Section 7.1.4) with successor functions and parameters from Section 7.1.5. The experimental variables and their respective levels are shown in Table 25. Each experimental condition (i.e., a particular CLP, learner, successor, and successor parameters) was repeated 10 times.

Using all of the available examples for feature construction would allow overfitting the data with constructed features. In this set of experiments, ISAK was configured to use a twenty percent sample of the data for feature construction. The sample size was chosen to balance between minimizing overall run-time for a large set of runs while still generating usable constructed features. The feature construction sample is taken independently of the cross-validation samples used in learning evaluation; the sample used for feature construction is replaced for learning.

To restrict the dimensionality of the constructed data, a maximum of 10 features are returned from ISAK for use in the learning algorithm. The maximum number of additional features was

Table 25: Experimental Conditions for the First Set of Experiments. These conditions, along with numeric parameters for the successor functions, define the space of input variables for the first set of experiments. Each condition (i.e., a selection of a learner, successor function with parameters, CLP, and feature set) was repeated 10 times yielding a 10-fold cross-validation accuracy for each repetition. The ten values were then averaged.

Experimental Variable	Values
Learners	MAJ, 3NN, 10NN, NB, TREE, SVM, FORST
Successor Functions	Exhaustive, Top, Random, Joined Ball, Separate Ball
CLPs	Each CLP from Section 7.1.1
Feature Set	Base, New, All, ConRIF, CCF

chosen to be of the approximate magnitude of the number of base features in each of the CLPs. Allowing the number of constructed features to grow arbitrarily large would allow the features to overfit the data from which they are constructed. These returned features are the top features measured by information gain on the data sampled for feature construction. Some problems are restricted to 5 constructed features as shown in Table 22. So, at most the number of features in any dataset used for learning is $10 + |base\,features|$.

10-Fold Cross-Validation Accuracy (10CVA) was taken as the primary performance indicator, where classification accuracy is defined as the ratio of correct classification predictions to all predictions. However, there are problems with simply evaluating learners by their accuracy. The Area-Under-the-(ROC)-Curve[3] (AUC) for learners has been proposed as an alternative measure of performance. I will not evaluate AUC in the same detail as accuracy, but I did gather AUC rates for the experimental conditions. I also gathered the time spent in feature generation and the number of features examined in each condition.

The relationships among the experimental variables are shown in Figure 35.

[3]The ROC curve is the receiver operating characteristic curve.

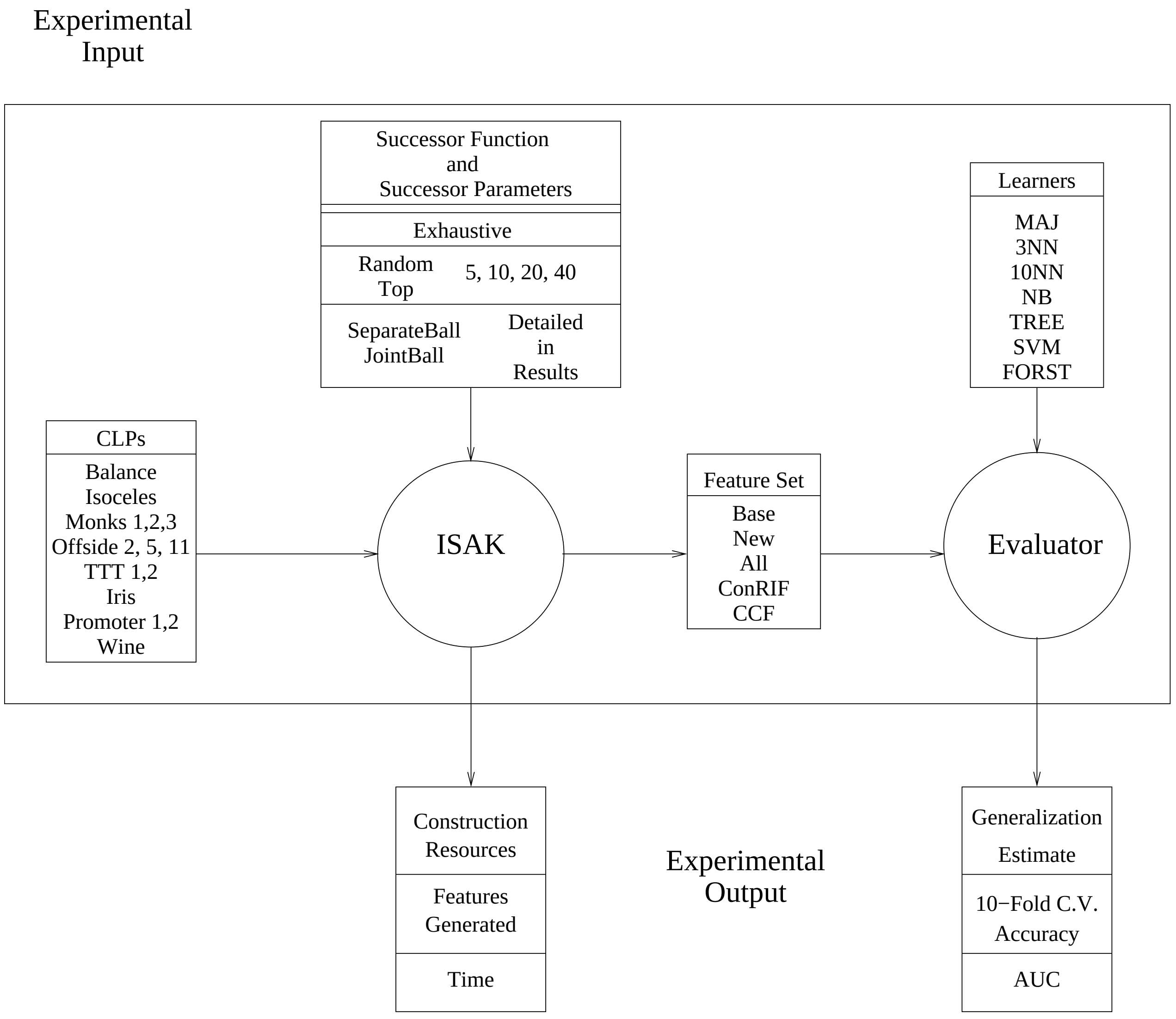


Figure 35: Relationships among Experimental Variables. *Seperate Ball* and *Joined Ball* are abbreviated as *Sep* and *Join* in the results tables. The generalization estimates are 10-fold cross-validation classification accuracy (10CVA) and Area Under an ROC Curve (AUC). *Features Generated* is a simple count of the features considered in the construction process; *Time* is the user time consumed by ISAK. The diagram shows the layout of a single run. Multiple replication trials (10 per set) were performed for each set of experimental variables.

Table 26: 10CVA Averaged over Learner, CLP, and Run by Feature Set. The Base feature set gives an 10CVA of .81 with a standard error of .0009. The maximum standard error of cell values in this table is less than .0064 (Section 7.2.5).

Base Feature Set 10CVA: .81			Average 10CVA over All Learners and All CLPs			
			Feature Set			
Successor Function	Maximum Cross	Hard Limit	All	New	ConRIF	CCF
Exhaustive	150	150	0.87	0.83	0.86	0.80
Join	20	80	0.86	0.82	0.86	0.79
Random	40	40	0.86	0.82	0.86	0.79
Sep	10	40	0.86	0.79	0.85	0.77
Top	40	40	0.86	0.80	0.85	0.78

7.2.2 An Experiment Comparing Feature Sets Averaged over All Learners

Table 26 shows the 10CVA averaged over the conditions of learner, CLP, and run for each level of feature set. A detailed table of all successor functions, successor parameters, and feature sets is shown in Appendix A.1, as are analogous results with AUC as the performance measure. Under AUC, the trends across experimental conditions are similar to the trends under 10CVA.

Among the feature sets, the general trend is that $CCF < New < Base < All \approxeq ConRIF$. That is, the order of performance of feature sets is (1) components of constructed features, (2) newly constructed features, (3) base features, and then, (4) all features (base and constructed) and constructed and remaining initial features are approximately equivalent. The comparison among levels of feature set is examined in more detailed in Sections 7.2.7 and 7.2.9.

7.2.3 An Experiment Comparing Successors Averaged over All Learners

Table 27 shows the 10CVA averaged over the conditions of learner, CLP, and run for each level of successor and successor parameters. A detailed table of all successor functions, successor parameters, and feature sets is shown in Appendix A.1, as are analogous results with AUC as the

Table 27: 10CVA Averaged over Learner, CLP, and Run by Successor. The Base feature set gives an 10CVA of .81 with a standard error of .0009. The maximum standard error of cell values in this table is less than .0064 (Section 7.2.5). The three left-most columns are the values of the successor function and its search breadth parameters. The first row of values, *Exhaustive,* may be taken as gold standard of comparison for each other breadth-restricted successor.

Base Feature Set 10CVA: .81			Average 10CVA over All Learners and All CLPs	
			Feature Set	
Successor Function	Maximum Cross	Hard Limit	All	New
Exhaustive	150	150	0.87	0.83
Join	5	10	0.84	0.73
Join	5	20	0.85	0.75
Join	5	40	0.85	0.75
Join	10	20	0.85	0.76
Join	10	40	0.86	0.80
Join	20	40	0.86	0.81
Join	20	60	0.87	0.82
Join	20	80	0.86	0.82
Random	5	5	0.84	0.71
Random	10	10	0.85	0.77
Random	20	20	0.86	0.80
Random	40	40	0.86	0.82
Sep	3	6	0.84	0.73
Sep	3	10	0.85	0.76
Sep	3	15	0.85	0.76
Sep	3	20	0.85	0.77
Sep	3	30	0.85	0.77
Sep	5	10	0.85	0.76
Sep	5	15	0.85	0.75
Sep	5	20	0.85	0.76
Sep	5	30	0.86	0.78
Sep	10	15	0.85	0.77
Sep	10	20	0.86	0.78
Sep	10	30	0.86	0.79
Sep	10	40	0.86	0.79
Top	5	5	0.84	0.74
Top	10	10	0.85	0.76
Top	20	20	0.85	0.78
Top	40	40	0.86	0.80

performance measure. Under AUC, the trends across experimental conditions are similar to the trends under 10CVA.

Table 27 shows that the Exhaustive successor function serves as a maximum (based on breadth) for the other successors. Because Exhaustive does not select particular features for use in expanding the fringe (i.e., it attempts to generate all combinations of features and feature constructor functions), it performs the most comprehensive search of the constructed feature space. Hence, the other successor functions are only looking at a subset of the features looked at by Exhaustive and may miss features discovered by Exhaustive. As a trade-off, other successors may find features deeper in the search space than Exhaustive can explore in a given amount of time (see Section 7.2.11). However, in the search parameters for the CLPs, the hardest constraint on search is depth. So, the different successors are really only varying breadth for a given depth in each CLP. The variation in successor parameters controls the breadth. Thus, Exhaustive is searching the broadest space of features. Join(20,60), Join(20,80), Random(40), and Top(40) all approach the performance of Exhaustive because they are searching more broadly than lower valued successor parameters.

7.2.4 An Experiment Comparing Different Generators and Feature Sets over Individual Learners

Table 28 shows the 10CVA averaged over the CLPs and runs by each level of successor function and feature set for the TREE and SVM learners. Additional tables for the other learners are shown in Section A.3.1. Appendix A.3.2 shows analogous results for AUC for each learner. The performance follows similar trends to that of the averages taken over all learners (Table 40).

Here again, Join(20,60), Join(20,80), Random(40), and Top(40) approach the performance of Exhaustive. The highly restricted successors (i.e., with low parameter values), still help the TREE substantially. In answer to the question, “Can SVMs benefit from feature construction?”, the answer is yes: under feature construction with broad search, we see a 5% improvement in 10CVA.

Comparing Table 27 and Table 28 , limited successors seem to help TREE more than the average learner (increases around 5% for TREE versus 3% on average).

Table 28: 10CVA Averaged over CLP and Run by Successor Function and Feature Set for TREE and SVM. The three left-most columns are the values of the successor function and its search breadth parameters. The maximum standard error in this table is less than .018. Note that the value of .91 for SVM, All, Join(20,60) is unexpected; it is greater than SVM, All, Exhaustive (.90). Most likely, this is due to overlap in the confidence intervals and rounding of these means.

					TREE					SVM		
					Base	0.79				Base	0.85	
Successor Function	Maximum Cross	Hard Limit		All	New	ConRIF	CCF		All	New	ConRIF	CCF
Exhaustive	150	150		0.87	0.84	0.87	0.78		0.90	0.86	0.89	0.83
Join	5	10		0.84	0.73	0.84	0.70		0.88	0.75	0.87	0.73
Join	5	20		0.84	0.75	0.84	0.72		0.88	0.76	0.87	0.75
Join	5	40		0.85	0.75	0.85	0.72		0.89	0.77	0.88	0.75
Join	10	20		0.85	0.76	0.85	0.73		0.89	0.78	0.87	0.76
Join	10	40		0.86	0.80	0.86	0.76		0.90	0.81	0.87	0.79
Join	20	40		0.86	0.82	0.86	0.77		0.90	0.83	0.88	0.80
Join	20	60		0.87	0.82	0.87	0.78		0.91	0.84	0.89	0.82
Join	20	80		0.87	0.83	0.87	0.77		0.90	0.85	0.89	0.81
Random	5	5		0.84	0.71	0.84	0.69		0.88	0.73	0.87	0.71
Random	10	10		0.86	0.78	0.85	0.74		0.89	0.79	0.88	0.76
Random	20	20		0.86	0.80	0.85	0.76		0.90	0.82	0.88	0.79
Random	40	40		0.87	0.82	0.87	0.78		0.90	0.84	0.89	0.81
Sep	3	6		0.84	0.73	0.84	0.71		0.88	0.75	0.86	0.73
Sep	3	10		0.85	0.75	0.85	0.72		0.88	0.77	0.87	0.75
Sep	3	15		0.85	0.76	0.85	0.73		0.89	0.78	0.87	0.75
Sep	3	20		0.85	0.77	0.84	0.74		0.89	0.79	0.87	0.76
Sep	3	30		0.86	0.78	0.85	0.75		0.90	0.79	0.88	0.76
Sep	5	10		0.85	0.76	0.84	0.73		0.88	0.78	0.87	0.76
Sep	5	15		0.85	0.75	0.85	0.73		0.89	0.77	0.88	0.75
Sep	5	20		0.85	0.76	0.85	0.73		0.89	0.78	0.88	0.76
Sep	5	30		0.86	0.79	0.86	0.74		0.90	0.80	0.89	0.78
Sep	10	15		0.85	0.77	0.85	0.74		0.89	0.79	0.88	0.76
Sep	10	20		0.85	0.78	0.85	0.74		0.89	0.80	0.87	0.77
Sep	10	30		0.86	0.79	0.85	0.75		0.89	0.81	0.88	0.78
Sep	10	40		0.86	0.79	0.86	0.75		0.89	0.81	0.88	0.79
Top	5	5		0.84	0.74	0.84	0.72		0.88	0.76	0.86	0.74
Top	10	10		0.85	0.76	0.84	0.73		0.88	0.78	0.88	0.76
Top	20	20		0.86	0.78	0.85	0.75		0.89	0.80	0.88	0.78
Top	40	40		0.86	0.80	0.86	0.76		0.90	0.82	0.88	0.79

7.2.5 Variance in Experiments

The values in the preceding tables are averages of behavior. To understand these averages we need to address the variance in these values. Shortly, we will develop a linear model of the experimental variables. Linear modeling assumes a common variance among each of the experimental conditions.

There are several sources of variance in this set of experiments. In the feature construction process, there is a random sampling of cases to be used for feature construction. In the random generator, there is a random selection of features to be used for successive constructions. In all cases, the evaluation of feature sets is done via cross-validation which selects training and testing sets randomly. We will examine several combinations of variation. In each, I use standard error as the measure of variability.

The standard error determines the expected degree of overlap between the mean accuracies given in the previous sets of tables. For a given mean and standard error, $\overline{X} \pm 1.96SE$ represents a 95% confidence interval around that mean. If two confidence intervals do not overlap, an $\alpha = .05$ difference exists in those means, but this significance is uncorrected for multiple comparisons. The following figures have a horizontal box-and-whiskers plot superimposed on them. In the box-and-whiskers plot, the solid line indicates the inter-quartile range (IQR) spanning the first and third quartiles $[x_{.25}, x_{.75}]$ and the central dot on that line indicates the median $(x_{.5})$. The circles (overlapping in these graphs) represent outliers (values greater than $(1.5IQR) + x_{.75}$). Dashes represent values that are neither outliers nor within the IQR.

1. What is the variability of evaluations in a single experimental run? For example, given a CLP, a learner, a generator and generator parameters, and a feature set, what is the standard error of the 10-fold C.V. accuracy of one run? The distribution of these standard errors is shown in Figure 36.
2. What is the variability of the average 10-fold C.V. accuracies over the 10 repeated runs for each experimental scenario (i.e., a CLP, successor function, successor parameters, and a learner)? The distribution of these standard errors is shown in Figure 37. These values are the standard errors over the experimental conditions if we do not aggregate the values over any particular experimental condition.

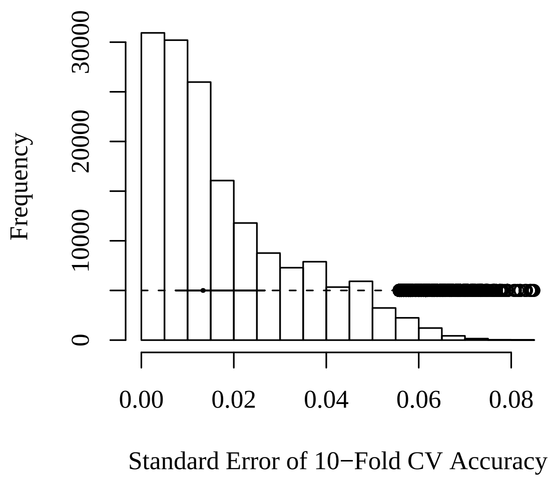


Figure 36: Histogram of Standard Errors for Single Runs of Experimental Setups. The black bar, dashed lines, and overlapping circles to the right form a box-and-whiskers plot of the distribution of values in the histogram. See the text for details.

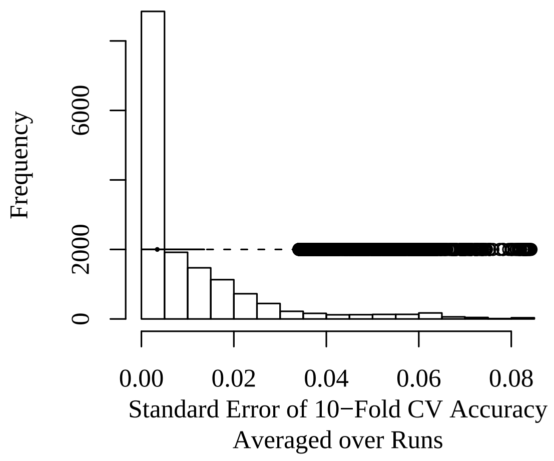


Figure 37: Histogram of Standard Errors for Repeated Runs of Experimental Conditions. The promoter CLP and the New feature set account for most of outlying, high standard errors.

3. The distribution of standard errors over repeated runs and CLPs is shown in Figure 38. These are the standard errors occurring in means shown in the Table 28 and in the tables in Appendix A.3.1. The means in these figures can be compared with a (worst-case) standard error of .018.
4. The distribution of variances over repeated runs, CLPs, and learners is shown in Figure 39. These are the standard errors of the accuracy means in Table 26 and Table 28.

One cell in Tables 26 and 27 is the average of the 10-fold cross-validation accuracies over CLP, learner, and run for the given successor function, successor parameters, and feature set. The distribution of the standard errors of these averages are given in Figure 39. The maximum standard error is less than .0064.

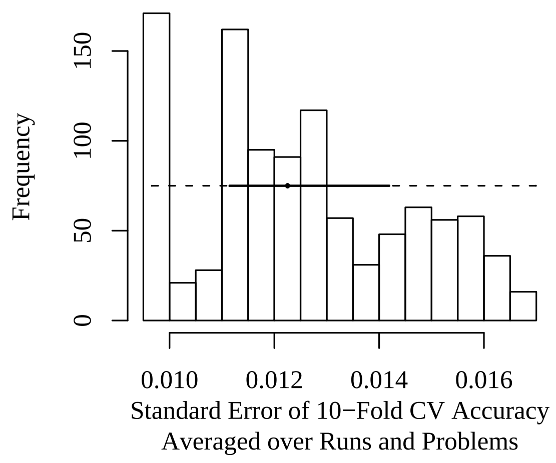


Figure 38: Histogram of Standard Errors of Average CA over Runs and Problems. These are the standard errors occurring in means shown in the Figure 28 and in the figures in Appendix A.3.1.

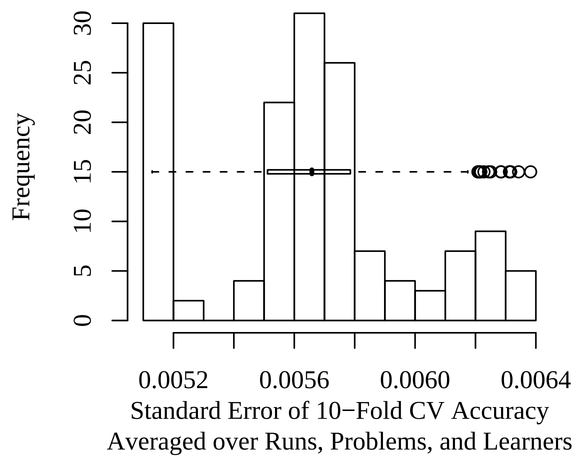


Figure 39: Histogram of Standard Errors of Average CA over Runs, Problems, and Learners. These are the standard errors of the accuracy means in Figure 26

7.2.6 A Linear Model for Experimental Variables

Here I analyze the extent to which the experimental variables and interactions among these variables account for the variability in the experimental results. The values of the experimental conditions are given in Table 25. I generated two linear models of the 10CVA on the experimental input variables (Figure 35):

1. the first linear model consists of the two- and three-way interactions between the learner, CLP, and feature set and two-way interactions between the successor function and its two parameters (*HardLimit* and *MaximumCross*).
2. the second linear model consists of (1) all two-way interactions between learner, CLP, feature set, successor function, and the successor parameters and (2) the three-way interactions among the learner, CLP, and feature set and among the successor function and successor parameters.

The second, more complex model, accounted for significantly more variability than the simple model. The results of an ANOVA between the two models is shown in Table 29. In addition to the comparison between the two models, we see that all of the main effects and all of the interactions are significant. We also see that the residual sum of squares for the second model are relatively small.

The complete set of parameters for the second model is given in Section A.2. In the more complex model, almost all of the coefficients are significant (i.e., non-zero). Of those coefficients that are not significant (i.e., the coefficients are approximately zero), the most obvious block is the set of coefficients on the interactions between CLP and feature set. Likely, the insignificance is because the variability is accounted for by the CLP and feature set individually and the three-way interaction between CLP, feature set, and learner which is by far the strongest three-way interaction. The CCF[4] feature set also has weak coefficients on its interactions with the learners and its interactions with the learners and CLPs.

[4]The CCF feature set is referred to as UsedO in Appendix A.2.

Table 29: ANOVA for Linear Models of the First Set of Experiments. In the ANOVA analysis, *gen* refers to the successor function. All interactions among experimental conditions have a (highly) significant effect on 10CVA (abbreviated CA in the table).

```
justanova.txt       Wed Mar 28 11:00:12 2007       1

Analysis of Variance Table

Response: CA
                      Df  Sum Sq Mean Sq   F value    Pr(>F)
learner                6 147.861  24.643 18260.225 < 2.2e-16 ***
prob                  14 196.944  14.067 10423.637 < 2.2e-16 ***
ftrSet                 4  17.496   4.374  3240.977 < 2.2e-16 ***
gen                    4   0.355   0.089    65.849 < 2.2e-16 ***
hardLimit              1   1.219   1.219   902.959 < 2.2e-16 ***
maxCross               1   0.129   0.129    95.954 < 2.2e-16 ***
learner:prob          84  76.927   0.916   678.585 < 2.2e-16 ***
learner:ftrSet        24   4.219   0.176   130.256 < 2.2e-16 ***
prob:ftrSet           56  24.439   0.436   323.373 < 2.2e-16 ***
gen:hardLimit          3   0.085   0.028    20.910 1.633e-13 ***
gen:maxCross           1   0.014   0.014    10.583  0.001144 **
hardLimit:maxCross     1   0.118   0.118    87.445 < 2.2e-16 ***
learner:prob:ftrSet  336  17.994   0.054    39.683 < 2.2e-16 ***
Residuals          15214  20.532   0.001
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1
Analysis of Variance Table

Response: CA
                         Df  Sum Sq Mean Sq    F value    Pr(>F)
learner                   6 147.861  24.643 25797.1389 < 2.2e-16 ***
prob                     14 196.944  14.067 14725.9961 < 2.2e-16 ***
ftrSet                    4  17.496   4.374  4578.6905 < 2.2e-16 ***
gen                       4   0.355   0.089    93.0281 < 2.2e-16 ***
hardLimit                 1   1.219   1.219  1275.6562 < 2.2e-16 ***
maxCross                  1   0.129   0.129   135.5594 < 2.2e-16 ***
learner:prob             84  76.927   0.916   958.6713 < 2.2e-16 ***
learner:ftrSet           24   4.219   0.176   184.0195 < 2.2e-16 ***
learner:gen              24   0.091   0.004     3.9814 1.863e-10 ***
learner:hardLimit         6   0.295   0.049    51.4433 < 2.2e-16 ***
learner:maxCross          6   0.026   0.004     4.6204 0.0001070 ***
prob:ftrSet              56  24.439   0.436   456.8448 < 2.2e-16 ***
prob:gen                 56   1.829   0.033    34.1867 < 2.2e-16 ***
prob:hardLimit           14   2.507   0.179   187.4643 < 2.2e-16 ***
prob:maxCross            14   0.185   0.013    13.8572 < 2.2e-16 ***
ftrSet:gen               16   0.263   0.016    17.1747 < 2.2e-16 ***
ftrSet:hardLimit          4   0.841   0.210   220.0036 < 2.2e-16 ***
ftrSet:maxCross           4   0.081   0.020    21.3101 < 2.2e-16 ***
gen:hardLimit             3   0.085   0.028    29.5402 < 2.2e-16 ***
gen:maxCross              1   0.014   0.014    14.9510 0.0001108 ***
hardLimit:maxCross        1   0.118   0.118   123.5381 < 2.2e-16 ***
learner:prob:ftrSet     336  17.994   0.054    56.0621 < 2.2e-16 ***
gen:hardLimit:maxCross    3   0.021   0.007     7.2137 7.802e-05 ***
Residuals             15067  14.393   0.001
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1
Analysis of Variance Table

Model 1: CA ~ (learner + prob + ftrSet)^3 + (gen + hardLimit + maxCross)^2
Model 2: CA ~ (learner + prob + ftrSet + gen + hardLimit + maxCross)^2 +
    (learner + prob + ftrSet)^3 + (gen + hardLimit + maxCross)^3
  Res.Df     RSS    Df Sum of Sq      F    Pr(>F)
1  15214 20.5323
2  15067 14.3932   147    6.1392 43.718 < 2.2e-16 ***
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1
```

7.2.7 An Experiment Comparing the Effect of Different Feature Sets within Each Individual Learner

There are clear trends in the accuracies given different combinations of base and constructed features for particular learners (Table 30 and Figure 40). Here, we will ignore (1) the MAJ learner because it is unaffected by feature set and (2) the CCF feature set because it has very poor performance. Within the remaining experimental conditions, the only differences that are *not significant* are for NB: (All, Base) and (All, ConRIF) and for FORST, 3NN, and TREE: (All, ConRIF). The experimental condition New gives uniformly poor performance and is significantly worse than all other feature sets including condition Base. Essentially, the ordering of feature sets is $New < Base < ConRIF = All$.

The statistical comparison was done using Tukey's Honestly Significant Difference (HSD) test (Hochberg and Tamhane, 1987). Tukey's HSD test is performed after an ANOVA test reveals that some difference exists in group means for given levels of an experimental variable. If the ANOVA gives evidence of mean difference, Tukey's HSD provides a test of which levels of the mean differ from each other in pairwise comparisons. Tukey's HSD operates by comparing differences in the means for each level of a variable and makes appropriate adjustments for multiple comparisons.

Table 30: Average 10CVA by Feature Set within Learners. The table shows the average 10-fold cross-validation accuracy per feature set averaged over successor, CLP, and run in the context of a single learner. Significance testing of differences in means is done in Figure 40. The only learner for which All is not the highest 10CVA is naive Bayes'. See text for details.

Learner	Feature Set	Average 10CVA	Learner	Feature Set	Average 10CVA
3NN	Base	0.796	TREE	Base	0.794
3NN	New	0.766	TREE	New	0.775
3NN	ConRIF	0.837	TREE	ConRIF	0.851
3NN	All	0.845	TREE	All	0.854
10NN	Base	0.822	SVM	Base	0.855
10NN	New	0.780	SVM	New	0.793
10NN	ConRIF	0.854	SVM	ConRIF	0.877
10NN	All	0.867	SVM	All	0.891
NB	Base	0.784	FORST	Base	0.829
NB	New	0.751	FORST	New	0.786
NB	ConRIF	0.794	FORST	ConRIF	0.869
NB	All	0.792	FORST	All	0.872

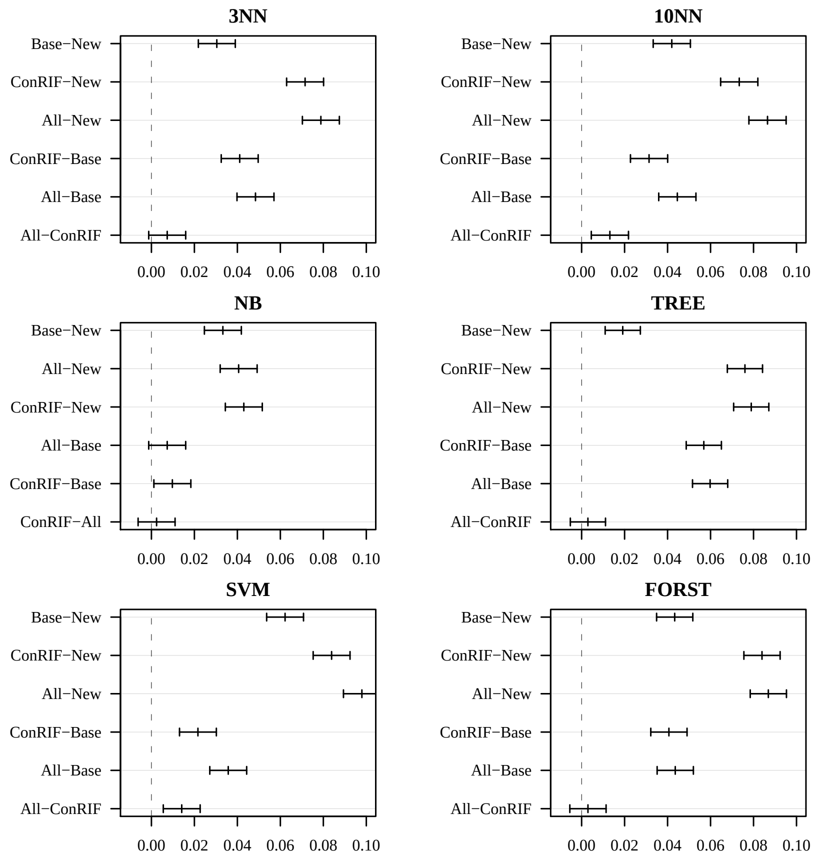


Figure 40: Tukey's HSD for Pairwise Differences between Feature Sets given Learners. The graphs show 95% confidence intervals for the difference in the mean 10CVA for the pairs of feature sets within the given learner.

Specifically, Tukey's HSD preserves the family-wise type I error rate among a set of hypothesis tests by using a specific ordering of comparisons between pairs of group means.

Why does New give uniformly poor performance? Recall, that for each CLP, the number of features returned by ISAK to be used for learning is limited to 10. Thus, the new feature sets have, at most, 10 features. Other feature sets expand and contract with the size of the base feature set. A manual examination of the features in the new feature set reveals a large degree of overlap in the base features used for construction. The overlap means that there is a lack of diversity of information contained in the new feature sets. It appears that the new feature sets with a strict size limit throw away too much of the information contained in the base feature sets and do not synthesize enough additional information to make up for the loss.

For 3NN, TREE, and FORST the lack of difference between ConRIF and All is acceptable. If All is considered a gold-standard, upper bound for the feature sets, then it would be surprising for ConRIF to outperform All. For SVM, All does maintain a significant difference over ConRIF. The significance is likely due to the implicit feature construction performed by SVM which may construct additional (implicit) features using the base features removed from ConRIF.

The lack of improvement in NB with All versus Base (and the closeness of ConRIF to Base) is likely due to the redundancy, and hence dependence, among the constructed features and between the base and constructed features.

The significant differences between Base and ConRIF for most learners must be considered against the actual size of the differences. Statistical significance does not imply real-world significance; any sufficiently large sample size can give statistical significance to a difference. Looking at Table 30, the increases in accuracy range from 1% for NB to 3% for SVM and up to 6% for TREE. The small increase for NB is almost inconsequential. The modest increase for TREE seems quite useful.

7.2.8 An Experiment Comparing the Effect of Different Successor Functions within Each Individual Learner

The other question that can be asked, given a learner, is what successor function should be used. I address this question in Table 31 and Figure 41. Again, Tukey's HSD (described in Section 7.2.7)

Table 31: Average 10CVA by Successor within Learners. The table shows the average 10-fold cross-validation accuracy per successor averaged over each CLP, feature set, and run in the context of a single learner. Significance testing of differences in means is done in Figure 41.

Learner	Successor	Average 10CVA	Learner	Successor	Average 10CVA
3NN	Exh	0.829	TREE	Exh	0.831
3NN	Join	0.801	TREE	Join	0.807
3NN	Rand	0.797	TREE	Rand	0.803
3NN	Sep	0.795	TREE	Sep	0.800
3NN	Top	0.795	TREE	Top	0.801
10NN	Exh	0.847	SVM	Exh	0.866
10NN	Join	0.820	SVM	Join	0.840
10NN	Rand	0.816	SVM	Rand	0.838
10NN	Sep	0.813	SVM	Sep	0.833
10NN	Top	0.813	SVM	Top	0.835
NB	Exh	0.783	FORST	Exh	0.854
NB	Join	0.774	FORST	Join	0.828
NB	Rand	0.773	FORST	Rand	0.825
NB	Sep	0.771	FORST	Sep	0.822
NB	Top	0.772	FORST	Top	0.822

is used to calculate the confidence intervals of the differences. The only significant differences exist between Exhaustive and other levels (except for NB which has no significant differences). The general trend is that Exhaustive is better than all others and the others are statistically similar. These results are consistent with the fact that Exhaustive generates all possible combinations in the space described by the CLP while the other generators perform pruning before expanding the fringe. Also, the comparison is taken over all values of successor parameters. If we only used the most extensive successor parameters, the non-exhaustive successors would be more similar to Exhaustive. Importantly, the choice of successor function is not a function solely of accuracy. The efficiency of generators with respect to accuracy improvement is discussed in Section 7.2.11.

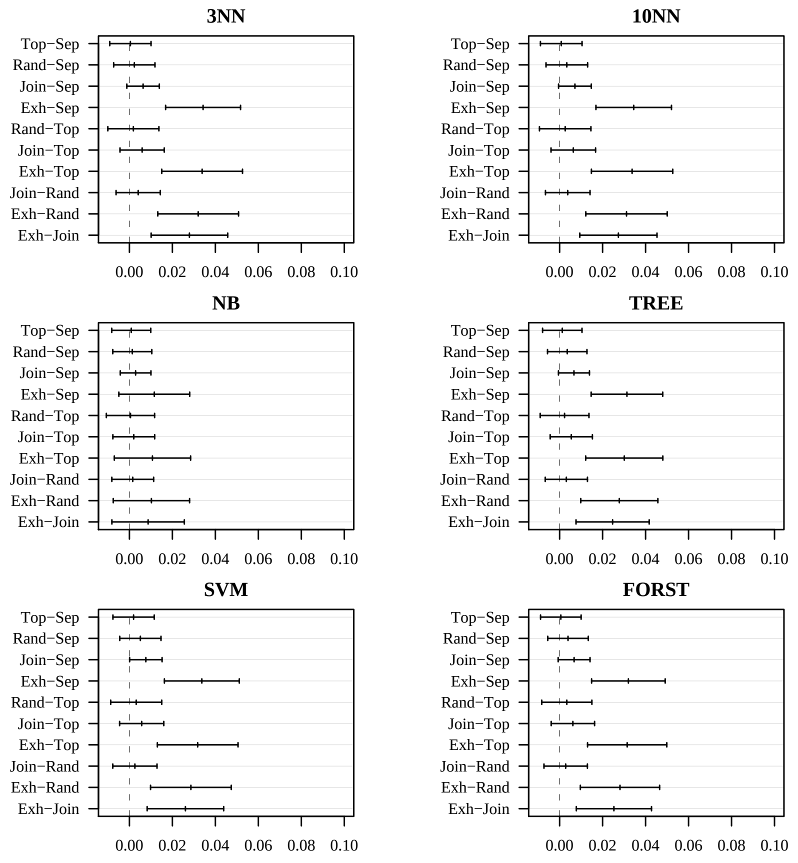


Figure 41: Tukey's HSD for Pairwise Differences between Successors given Learners. The graphs show 95% confidence intervals for the difference in the mean 10CVA for the pairs of generators. Only Exhaustive has any significant difference with any other successor function.

7.2.9 An Experiment Comparing the Effect of Different Feature Sets within Each Individual CLP

Typically, one does not have the same control over the choice of learning problem as one does with a choice of learning method. However, the data available in this set of experiments let us ask what we can expect from different problems. Table 32 shows the means of different feature sets and generators given a problem. Again, I performed a Turkey's HSD test for significance of these differences. The test by feature set within a CLP is given in Figure 42.

A few problems show very large increases in accuracy over the Base feature set: balance, isosceles, monks1, and monks2. These problems share a small to moderate sized target concept that is known *a priori* in a closed-form. Many CLPs show a strong decrease in accuracy for the New feature set. The decrease is not reflected in the ConRIF and All datasets; this is evidence that replacing the information from Base that is missing in New is sufficient to achieve Base level performance. There are many insignificant (or statistically significant but small) differences between Base, ConRIF, and All. Offsides2 shows modest improvement from both ConRIF and All. The other offsides problems are poor performers with all of the feature sets. Iris, wine, and monks3 show little improvement but these CLPs already have a high accuracy rate over the given learners.

Some comparisons between problems are interesting. The offsides problems differ in number of features; promoter2 and ttt1 use problem-specific knowledge, promoter1 and ttt2 use general knowledge, and promoter3 uses a combination of problem-specific and general knowledge.

Table 32: Average 10CVA by Feature Set within CLPs. The table shows the average 10-fold cross-validation accuracy per feature set averaged over learner, successor, and run in the context of a single CLP. Significance testing of differences in means is done in Figure 42. Comparisons across rows of similar CLPs are interesting but not tested statistically.

CLP	Feature Set	Average 10CVA	CLP	Feature Set	Average 10CVA	CLP	Feature Set	Average 10CVA
bal	Base	0.834	iris	Base	0.943	isoc	Base	0.860
bal	New	0.901	iris	New	0.930	isoc	New	1.000
bal	ConRIF	0.902	iris	ConRIF	0.956	isoc	ConRIF	1.000
bal	All	0.904	iris	All	0.956	isoc	All	0.997
mnk1	Base	0.897	mnk2	Base	0.676	mnk3	Base	0.949
mnk1	New	0.958	mnk2	New	0.683	mnk3	New	0.947
mnk1	ConRIF	0.989	mnk2	ConRIF	0.859	mnk3	ConRIF	0.964
mnk1	All	0.989	mnk2	All	0.827	mnk3	All	0.970
off02	Base	0.667	off05	Base	0.639	off11	Base	0.502
off02	New	0.647	off05	New	0.601	off11	New	0.516
off02	ConRIF	0.698	off05	ConRIF	0.643	off11	ConRIF	0.505
off02	All	0.701	off05	All	0.644	off11	All	0.505
prom1	Base	0.855	prom2	Base	0.856	prom3	Base	0.855
prom1	New	0.775	prom2	New	0.582	prom3	New	0.665
prom1	ConRIF	0.845	prom2	ConRIF	0.862	prom3	ConRIF	0.852
prom1	All	0.846	prom2	All	0.870	prom3	All	0.862
ttt1	Base	0.856	ttt2	Base	0.855	wine	Base	0.960
ttt1	New	0.839	ttt2	New	0.682	wine	New	0.899
ttt1	ConRIF	0.846	ttt2	ConRIF	0.827	wine	ConRIF	0.957
ttt1	All	0.908	ttt2	All	0.863	wine	All	0.959

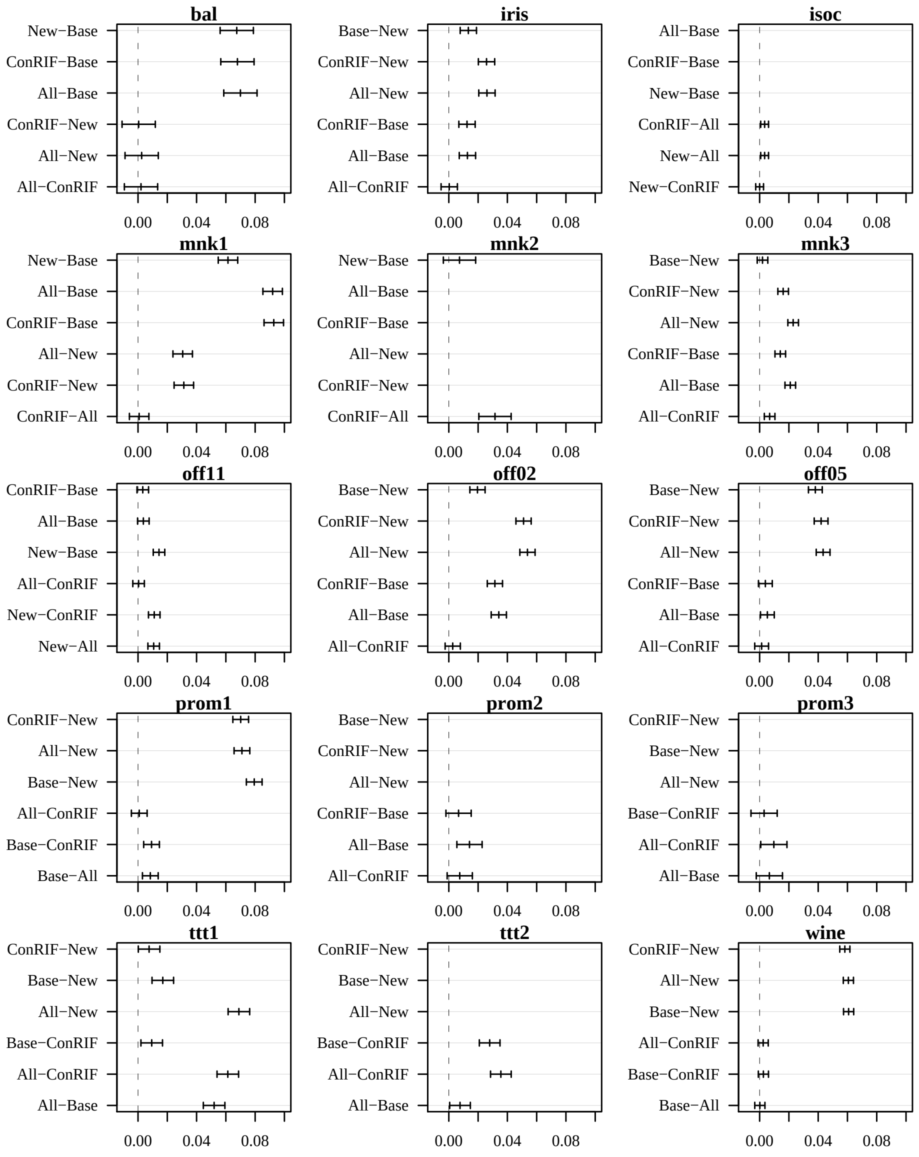


Figure 42: Tukey's HSD for Pairwise Differences between Feature Sets given CLPs. For isoc, All, New, and ConRIF are at least 10% greater than Base. For prom2, prom3, and ttt2, New is at least 10% smaller than Base, ConRIF, and All. For mnk2, Base and New are at least 10% less than ConRIF and All.

7.2.10 An Experiment Comparing the Effect of Different Successor Functions within Each Individual CLP

For a given CLP, there seems to be little significant difference due to successor function (see Table 33 and Figure 43). Again, I performed a Turkey's HSD test for significance of these differences. The problems that do show substantial variation due to successor are monks2, promoter2, promoter3, and ttt1. Promoter2 and ttt1 use domain-specific KBs for those learning problems (in contrast with the general symbolic KBs). This implies that more detailed knowledge leads to better results earlier in the search. Monks2 uses general symbolic value manipulation. Promoter3 incorporates both domain and general knowledge. The base feature sets for these four problems contain only symbolic features.

Among problems that have similar characteristics, the offsides problems differ in number of features; promoter2 and ttt1 use problem-specific knowledge, promoter1 and ttt2 use general knowledge, and promoter3 uses a combination of problem-specific and general knowledge. Promoter1 has similar performance regardless of successor; promoter2 improves markedly with the exhaustive successor and does poorly with the other successors. This may indicate a problem with the search parameters for promoter2. Promoter3, which combines the knowledge used in promoter1 and 2, seems to be an approximate average of the performance of the other two.

As shown by the widths of the confidence intervals, it appears that given a fixed CLP the choice of successor function introduces substantially greater variability into the classification accuracy than does the choice of a feature set (with the exception of the New feature set).

Table 33: Average 10CVA by Successor within a CLP. The table shows the average 10-fold cross-validation accuracy per successor averaged over learner, feature set, and run in the context of a single CLP. Significance testing of differences in means is done in Figure 43. These values are computed from the main Experiment 1 dataset after screening out the MAJ learner and the CCF feature set. Comparisons across similar CLPs are interesting but not tested statistically.

CLP	Successor	Average 10CVA	CLP	Successor	Average 10CVA	CLP	Successor	Average 10CVA
bal	Exh	0.880	iris	Exh	0.951	isoc	Exh	0.951
bal	Join	0.880	iris	Join	0.930	isoc	Join	0.951
bal	Rand	0.868	iris	Rand	0.952	isoc	Rand	0.951
bal	Sep	0.872	iris	Sep	0.939	isoc	Sep	0.950
bal	Top	0.880	iris	Top	0.951	isoc	Top	0.950
mnk1	Exh	0.971	mnk2	Exh	0.759	mnk3	Exh	0.962
mnk1	Join	0.951	mnk2	Join	0.732	mnk3	Join	0.960
mnk1	Rand	0.935	mnk2	Rand	0.745	mnk3	Rand	0.932
mnk1	Sep	0.951	mnk2	Sep	0.729	mnk3	Sep	0.960
mnk1	Top	0.960	mnk2	Top	0.703	mnk3	Top	0.960
off02	Exh	0.700	off05	Exh	0.640	off11	Exh	0.522
off02	Join	0.670	off05	Join	0.624	off11	Join	0.508
off02	Rand	0.672	off05	Rand	0.624	off11	Rand	0.512
off02	Sep	0.670	off05	Sep	0.629	off11	Sep	0.503
off02	Top	0.677	off05	Top	0.624	off11	Top	0.510
prom1	Exh	0.814	prom2	Exh	0.895	prom3	Exh	0.851
prom1	Join	0.821	prom2	Join	0.785	prom3	Join	0.807
prom1	Rand	0.829	prom2	Rand	0.770	prom3	Rand	0.773
prom1	Sep	0.819	prom2	Sep	0.716	prom3	Sep	0.763
prom1	Top	0.822	prom2	Top	0.725	prom3	Top	0.770
ttt1	Exh	0.891	ttt2	Exh	0.796	wine	Exh	0.944
ttt1	Join	0.834	ttt2	Join	0.783	wine	Join	0.940
ttt1	Rand	0.849	ttt2	Rand	0.789	wine	Rand	0.930
ttt1	Sep	0.859	ttt2	Sep	0.781	wine	Sep	0.939
ttt1	Top	0.848	ttt2	Top	0.783	wine	Top	0.933

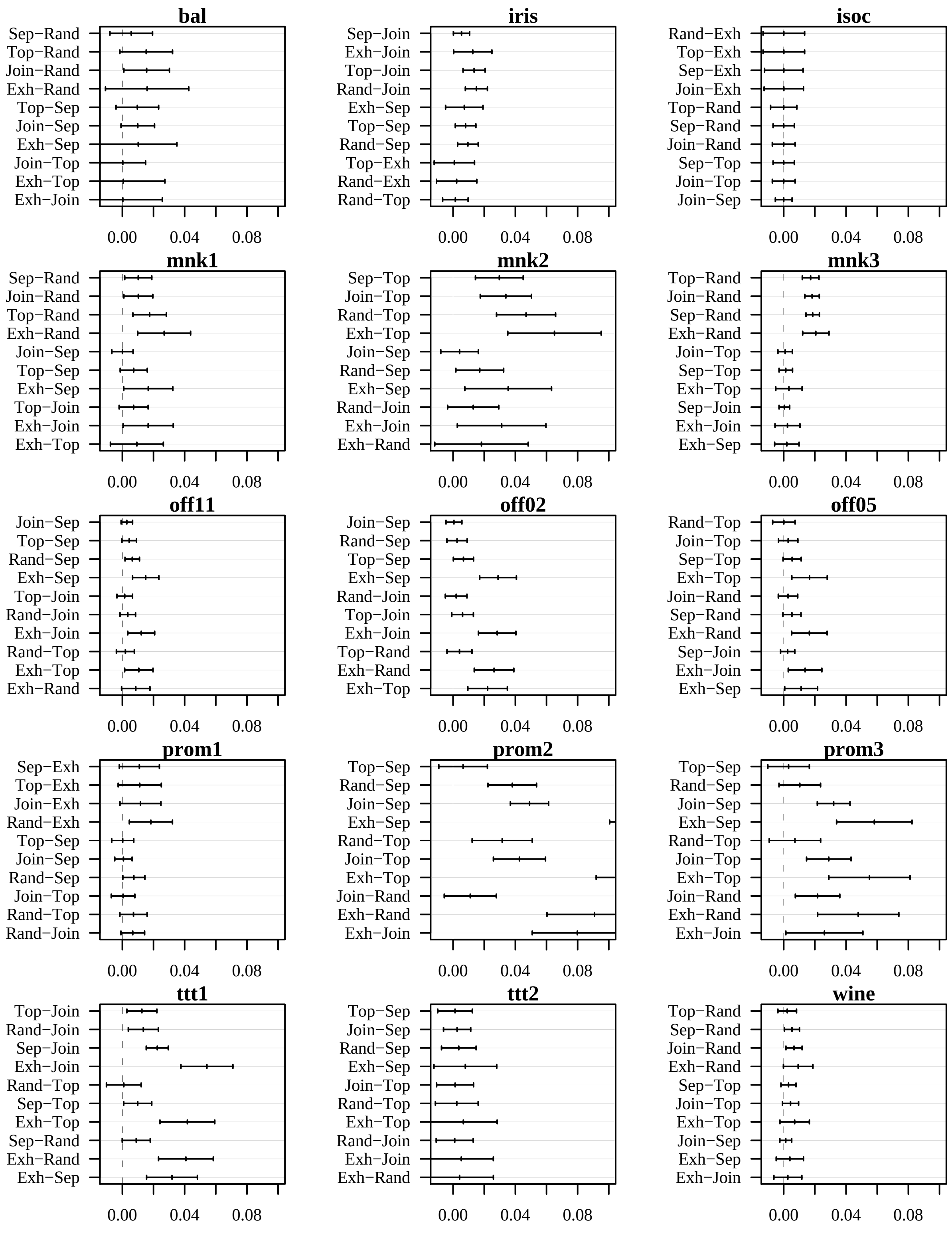


Figure 43: Tukey's HSD for Pairwise Differences between Generators Given Problems. With the exception of the ttt and promoter problems and monk2, most of the confidence intervals are relatively small.

Table 34: Overall Cost to Benefit. Each of the successor functions and parameters used to construct ConRIF achieves approximately the same improvement over the base features. The major difference is the time spent in feature generation. Random(20,20) uses the least time of these successor-parameter pairs. The set of rows shown here is the top quartile of successors and parameters on the basis of improvement of ConRIF over Base.

	Successor	Maximum Cross	Hard Limit	Feature Generation Time	10CVA using ConRIF	Improvement over Base Features	Increase in 10CVA Per Unit Time
Overall	random	20	20	1.8	0.85	0.04	0.021
	sep	5	30	13.51	0.86	0.04	0.003
	join	20	40	2.01	0.86	0.04	0.021
	random	40	40	5.15	0.86	0.04	0.008
	top	40	40	5.38	0.85	0.04	0.007
	join	20	60	2.81	0.86	0.05	0.017
	join	20	80	3.32	0.86	0.04	0.013
	Exh	150	150	20.65	0.86	0.05	0.002

7.2.11 An Experiment Comparing the Cost and Benefit of Different Successor Functions

When accounting for the time spent generating features, what is the best generator overall and what is the best generator within a given learner? Table 34 investigates performance of successors relative to time. Here, we consider the accuracy improvement of the ConRIF feature set generated by the different generators versus the base feature set. The improvement is divided by the time spent in the generator. To make recommendations about the generator and generator parameters, I first grouped the generators into the best performers by absolute accuracy improvement over learning with the base feature set. I then sorted the top quartile performers by accuracy improvement per unit time. Averaged over all of the learners Random(20,20) is the most cost-effective successor.

Looking at some individual learners (Table 35), Join(20,40) is the most cost-effective successor for for SVM, TREE, and 10NN. Join(20,40) is also a strong performer for FORST and 3NN.

NB does not see a great deal of accuracy improvement with any successor function and, by these measures, prefers a successor functions that finish quickly. The choice is reflected in the low successor parameter values – which generate fewer expanded nodes in the feature search space – for the top-quartile successors for NB.

Table 35: Cost to Benefit of ConRIF Feature Set for Different Generators. For each learner, the various successors and parameters are listed in increasing order of 10CVA improvement. The row highlighted in gray is the most cost-effective (i.e., greatest increase in 10CVA per unit time) successor and parameter set. Each set of rows shown here is the top quartile of successors and parameters on the basis of improvement of ConRIF over Base.

	Successor	Maximum Cross	Hard Limit	Feature Generation Time	10CVA using ConRIF	Improvement over Base Features	Increase in 10CVA Per Unit Time
3NN	random	20	20	1.8	0.85	0.05	0.028
	sep	5	30	13.51	0.85	0.05	0.004
	sep	10	40	12.41	0.85	0.05	0.004
	join	20	40	2.01	0.85	0.05	0.025
	random	40	40	5.15	0.85	0.05	0.011
	join	20	60	2.81	0.86	0.06	0.022
	join	20	80	3.32	0.85	0.06	0.017
	exh	150	150	20.65	0.86	0.06	0.003
10NN	sep	5	30	13.51	0.87	0.04	0.003
	sep	10	30	7.13	0.86	0.04	0.005
	sep	10	40	12.41	0.86	0.04	0.003
	join	20	40	2.01	0.87	0.04	0.021
	random	40	40	5.15	0.87	0.05	0.009
	join	20	60	2.81	0.87	0.05	0.018
	join	20	80	3.32	0.86	0.04	0.013
	exh	150	150	20.65	0.88	0.05	0.003
NB	top	5	5	0.2	0.8	0.01	0.073
	sep	3	6	0.93	0.81	0.02	0.024
	sep	3	10	3.18	0.8	0.02	0.006
	sep	5	10	1.23	0.8	0.02	0.012
	random	10	10	0.62	0.8	0.02	0.028
	sep	3	20	11.8	0.8	0.02	0.001
	sep	5	20	5.86	0.8	0.02	0.003
	join	10	20	0.71	0.8	0.02	0.022
TREE	sep	5	30	13.51	0.86	0.07	0.005
	sep	10	40	12.41	0.86	0.06	0.005
	join	20	40	2.01	0.86	0.07	0.035
	random	40	40	5.15	0.87	0.07	0.014
	top	40	40	5.38	0.86	0.07	0.013
	join	20	60	2.81	0.87	0.08	0.027
	join	20	80	3.32	0.87	0.07	0.022
	exh	150	150	20.65	0.87	0.08	0.004
SVM	sep	5	30	13.51	0.89	0.03	0.002
	sep	10	30	7.13	0.88	0.03	0.004
	join	20	40	2.01	0.88	0.03	0.014
	random	40	40	5.15	0.89	0.03	0.006
	top	40	40	5.38	0.88	0.03	0.005
	join	20	60	2.81	0.89	0.04	0.013
	join	20	80	3.32	0.89	0.03	0.009
	exh	150	150	20.65	0.89	0.04	0.002
FORST	sep	10	15	1.78	0.88	0.05	0.026
	sep	10	20	3.25	0.88	0.05	0.015
	sep	5	30	13.51	0.88	0.05	0.004
	join	20	40	2.01	0.88	0.05	0.026
	random	40	40	5.15	0.88	0.05	0.010
	top	40	40	5.38	0.88	0.05	0.009
	join	20	60	2.81	0.88	0.06	0.020
	exh	150	150	20.65	0.88	0.05	0.002

7.3 EXPERIMENTS WITH LEVELS OF KNOWLEDGE

7.3.1 The Area Construction and Learning Problems

To investigate the use of knowledge in construction, I created a learning problem that has multiple natural levels of knowledge. In this context, a level of knowledge refers to the generality or specificity of both the knowledge and the representation of a problem. The learning problem, which I call Area, is to determine which of three plane shapes has the largest area from a geometric descriptions of the shapes. Two natural levels of representing the Area problem are (1) representing the shapes by points (Point) and (2) representing the shapes by lengths of important pieces (Length). The representations are displayed in Figure 44. Table 36 shows 10 examples in the Point representation. Table 37 shows 10 examples in the Length representation.

While conceptually simple, Area poses problems for traditional learning algorithms, particularly in the Point representation. Figure 45 shows learning curves for several learners (the details of the learners are discussed in Section 7.1.3) on the Area problem with the Point representation. Figure 46 shows learning curves for the Area problem with the Length representation. In both figures, the dotted lines represent point-wise standard error for the accuracy estimates.

Just as there are natural levels of representation for the Area problem, there are also natural pieces of knowledge we can use to constrain the feature construction for these CLPs. We can constrain feature construction over shapes, over coordinates, over both, and over neither. The Shape KB allows constructions among features that share a semantic class (e.g., triangle points may only be combined with triangle points). The XY KB allows constructions among features that share a coordinate type (e.g., x-coordinates may only be combined with other x-coordinates). Applying both constraints would require constructions like triangle x_1 with triangle x_2. The two KBs account for four KB conditions (with and without each KB). The fifth KB condition, Detailed, includes operations that are appropriate in a Euclidian space – squares, differences, square roots, etc.

ISAK generated a ConRIF feature set using each of the five knowledge conditions and the Exhaustive successor using the Point representation.[5] The constructed datasets were given to an

[5]The Exhaustive generator was used because time was not an issue and because completeness of search through the space of possibilities was important for straightforward use of the knowledge.

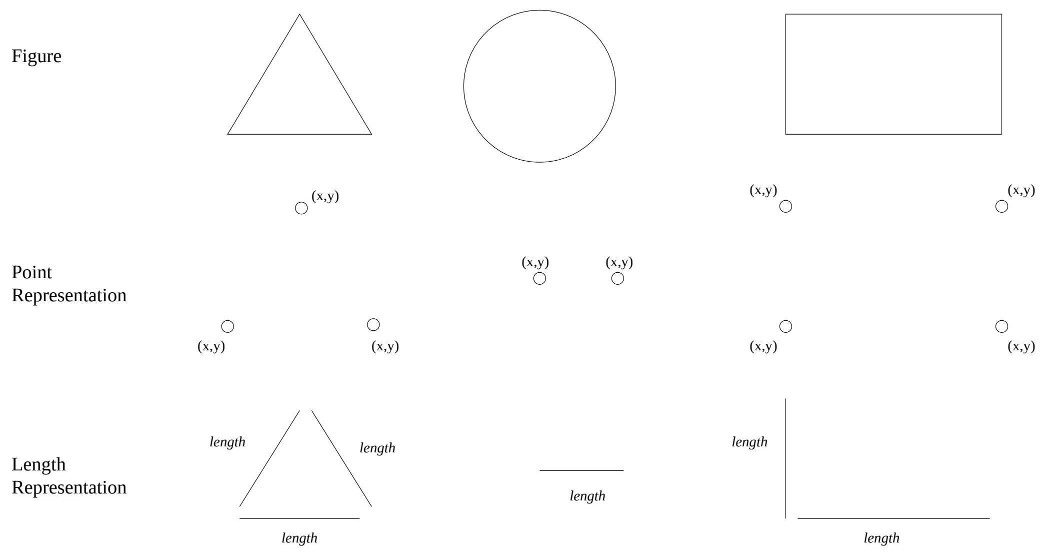


Figure 44: The Two Representations of the Area Problem. Each shape has a different set of attributes in the two forms of the Area problem. The triangle, circle, and rectangle are represented by 6, 4, and 8 attributes in the Point representation. Respectively, the shapes are represented by 3, 1, and 2 attributes in the Length representation.

Table 36: Examples from the Area Problem in the Point Representation. Note that a single example is defined over two lines because of the large number of features. A single example is defined by 18 values. The points belong to three different shapes, but without feature construction there is no specification of that relationship. Points are also distinguished by *x*- and *y*- coordinate. KBs add the knowledge that some points are related to other points.

Example	point 1	point 2	point 3	point 4	point 5	point 6	point 7	point 8	point 9
1	2.19	-2.92	6.26	-2.92	4.27	-0.58	-1.03	1.74	2.93
2	-0.56	3.42	3.12	3.42	-2.77	9.74	-2.78	0.01	1.71
3	-2.59	-2.72	7.57	-2.72	9.22	3.32	-1.25	-3.30	4.67
4	0.02	2.70	8.47	2.70	4.52	9.37	3.39	1.94	7.08
5	1.86	-1.60	4.83	-1.60	6.32	2.87	1.05	-3.43	5.32
6	0.18	0.84	7.13	0.84	5.07	1.63	3.15	1.97	8.89
7	-0.77	-2.64	8.41	-2.64	6.94	-1.80	2.79	-3.62	8.38
8	-1.13	2.69	0.33	2.69	1.05	7.15	-3.57	2.41	1.25
9	1.17	-2.55	4.14	-2.55	4.11	1.01	-2.77	-2.33	1.23
10	3.04	-0.75	7.25	-0.75	9.78	-0.55	-1.04	2.51	3.98

Example	point 10	point 11	point 12	point 13	point 14	point 15	point 16	point 17	point 18	largest?
										class
1	1.74	2.93	4.27	-1.03	4.27	-2.37	-2.59	0.06	-2.59	circ
2	0.01	1.71	1.19	-2.78	1.19	2.43	2.29	4.79	2.29	circ
3	-3.30	4.67	-1.48	-1.25	-1.48	2.94	-3.55	4.21	-3.55	tri
4	1.94	7.08	4.93	3.39	4.93	-3.31	-3.65	-1.76	-3.65	tri
5	-3.43	5.32	-1.98	1.05	-1.98	-2.29	3.45	-0.99	3.45	tri
6	1.97	8.89	6.23	3.15	6.23	-3.26	1.10	-2.16	1.10	rect
7	-3.62	8.38	0.48	2.79	0.48	3.30	-0.03	5.87	-0.03	rect
8	2.41	1.25	6.09	-3.57	6.09	1.40	1.60	4.05	1.60	circ
9	-2.33	1.23	0.25	-2.77	0.25	2.09	2.32	4.62	2.32	circ
10	2.51	3.98	3.83	-1.04	3.83	1.20	1.93	3.09	1.93	circ

Table 37: Examples from the Area Problem in the Length Representation. Here, a single learning example is described by six sides. The sides belong to three different shapes, but without feature construction there is no specification of that relationship. KBs add the knowledge that some sides are related to other sides.

side 1	side 2	side 3	side 4	side 5	side 6	largest?
						class
9.18	2.29	7.74	5.74	1.45	1.62	rect
9.08	1.49	8.1	4.46	3.28	1.93	rect
6.61	9.45	9.4	3.3	5.21	1.6	tri
4.08	3.2	3.25	4.64	2.42	2.54	circ
9.34	7.65	15.99	4.87	1.65	2.57	tri
8.89	2.22	8.31	5.3	1.94	2.56	circ
9.26	4.67	5.09	1.42	1.61	1.91	circ
3.94	6.9	6.58	1.32	1.91	1.16	tri
3.09	5.58	8.34	2.71	3.19	1.77	circ
8.3	8.47	9.48	4.26	4.31	1.73	tri

SVM learner and learning curves were constructed from the SVM's performance. Here, each learning curve plots the stratified 10-fold cross-validation accuracy against the percent used in the training sample proportion. A comparison of the number of features produced under each KB is shown in Table 38.

In the following figures, one set of accuracies and error estimates is either represented by a solid line with dotted error bars or by a solid center line with dark gray fill surrounding it. Figure 47 shows leaning curves for the original dataset and constructions with no restrictions. Figure 48 shows curves for the original point dataset, construction with coordinate restrictions, and constructions with figure restrictions. Figure 49 shows curves for the original point dataset, construction with coordinate restrictions, and construction with both restrictions.

Knowledge-based feature construction can make up for a lack of data. For example, 20% of the base dataset with constructions from the XY & Shape KB achieves approximately the same accuracy as 100% of the base dataset with no constructions. For the XY KB and the Shape KB, 40% of the base dataset with those constructions yields approximately the same accuracy as 100% of the base dataset with no constructions. The unrestricted constructions produce a significantly

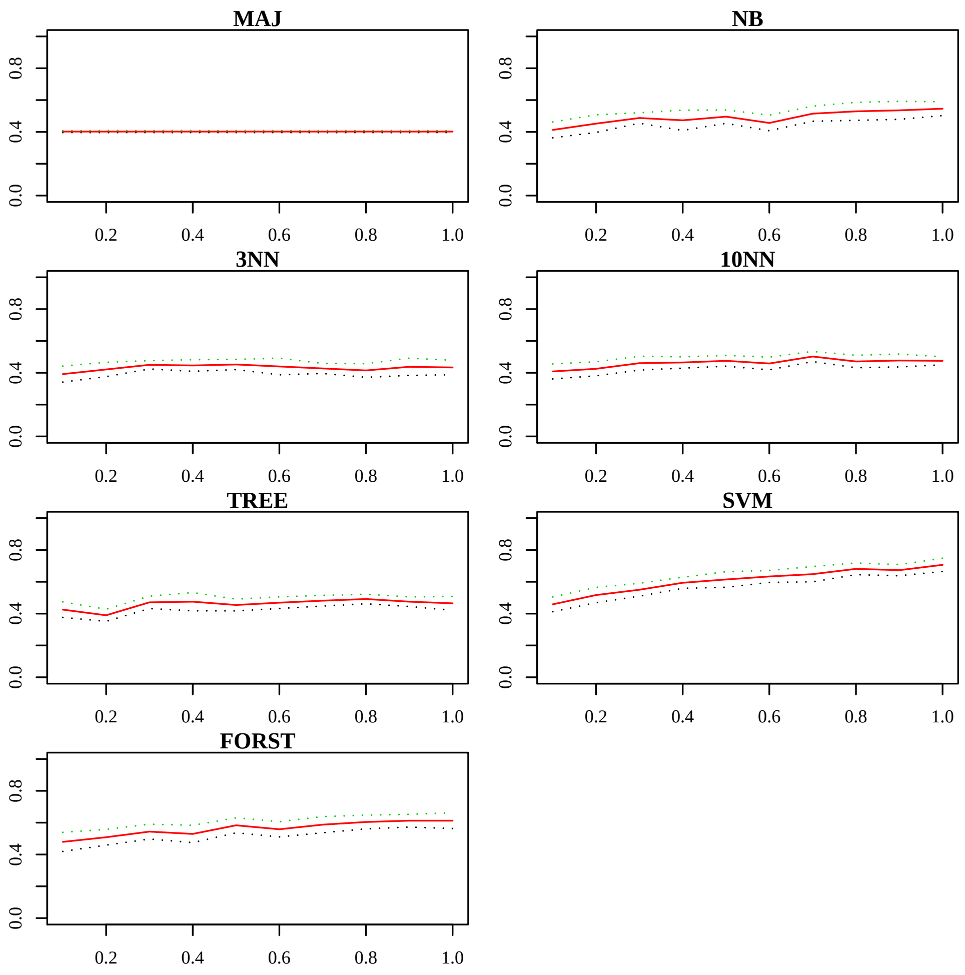


Figure 45: Learning Curves for the Point Representation of the Area Problem. These learning curves show the trade-off between sample size and accuracy for the Point form of the Area problem. The learners are evaluated using stratified 10-fold cross-validation accuracy. The upper and lower dotted lines represent 95% confidence intervals around the accuracy estimates. For the Point representation, the accuracies for most learners only approaches 50%. FORST approaches 60% and SVM approaches 70%.

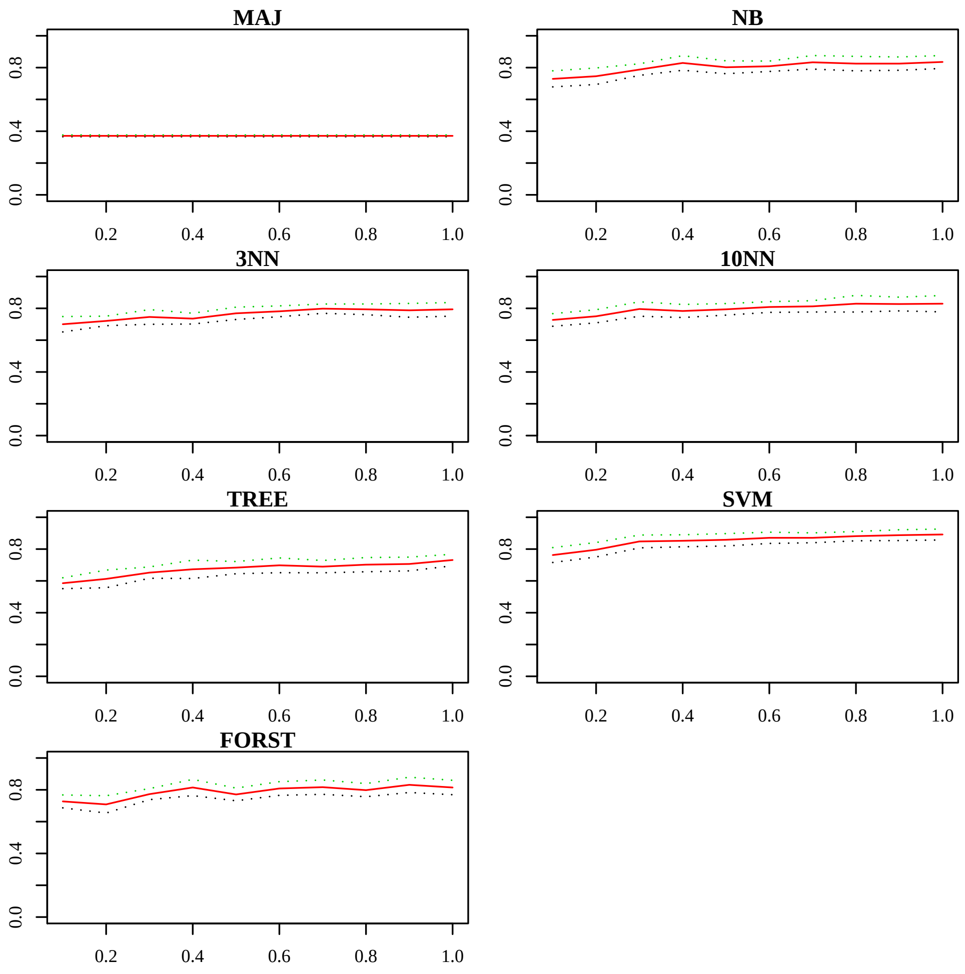


Figure 46: Learning Curves for the Length Representation of the Area Problem. These learning curves show the trade-off between sample size and accuracy for the Point form of the Area problem. The learners are evaluated using stratified 10-fold cross-validation accuracy. The upper and lower dotted lines represent 95% confidence intervals around the accuracy estimates. For the Length representation, the accuracies approach 90% with the full set of examples.

Table 38: Features Examined and Search Times for Different KBs. The table shows the number of features examined and the time in seconds for search under the different construction KBs with the exhaustive successor function. Note how the increase in knowledge restricts the number of features examined and, hence, reduces the time spent in search for the XY, Shape, and XY & Shape KBs. The Detailed KB introduces several addition feature constructor functions which greatly expand the space of considered features. These five KBs were applied to the Point representation of the Area problem. The KBs are discussed in the text.

KB	Features Examined	Search Time
Unrestricted	266520	76
XY	127128	44
Shape	79416	30
XY & Shape	24552	11
Detailed	911766	82

better dataset at the cost of a factor of 2 to 6 in the time for generation compared to the XY, Shape, and XY & Shape. The XY & Shape KB is superior to the XY KB and to the Shape KB; XY & Shape also requires less feature generation time than either individual KB.

Finally, a KB was developed for the Area problem that took into account both the constraints between shapes and coordinates and incorporated operations that are of use in building up Euclidean distances: differences, squares, sums, and square roots. These operations were restricted in application to features of appropriate levels (e.g., square roots of initial X-coordinates were not taken). A comparison between the original point to area dataset, the dataset developed with the knowledge intensive KB, and the shape to area dataset is shown in Figure 50.

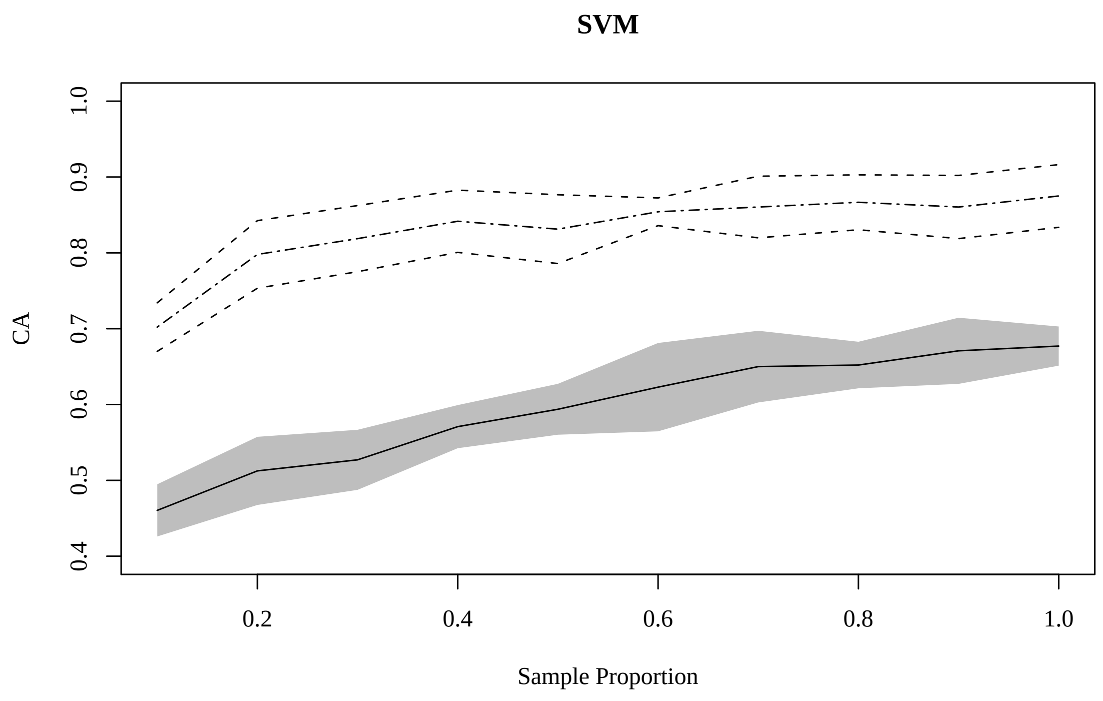


Figure 47: Original Point Dataset Compared to Unrestricted Constructions. The original point dataset for the area problem is the bottom triple; unrestricted constructions were used to make the dataset for the top triple. The datasets are evaluated using stratified 10-fold cross-validation accuracy on the *y*-axis. The upper and lower lines of each triple represent 95% confidence intervals around the accuracy estimates.

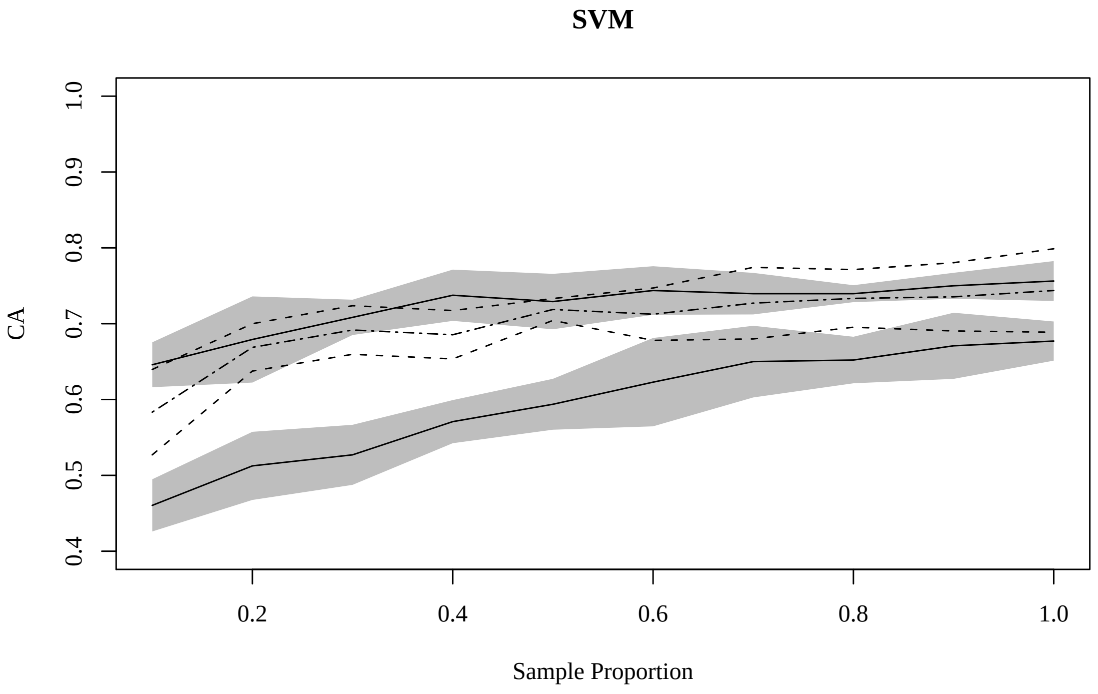


Figure 48: Original Point Dataset versus Constructions Limited to XY Coordinate and Constructions Limited to Shape. From bottom to top, the triples are: original point dataset, constructions from XY, and constructions from Shape. The datasets are evaluated using stratified 10-fold cross-validation accuracy on the *y*-axis. The upper and lower lines of each triple represent 95% confidence intervals around the accuracy estimates.

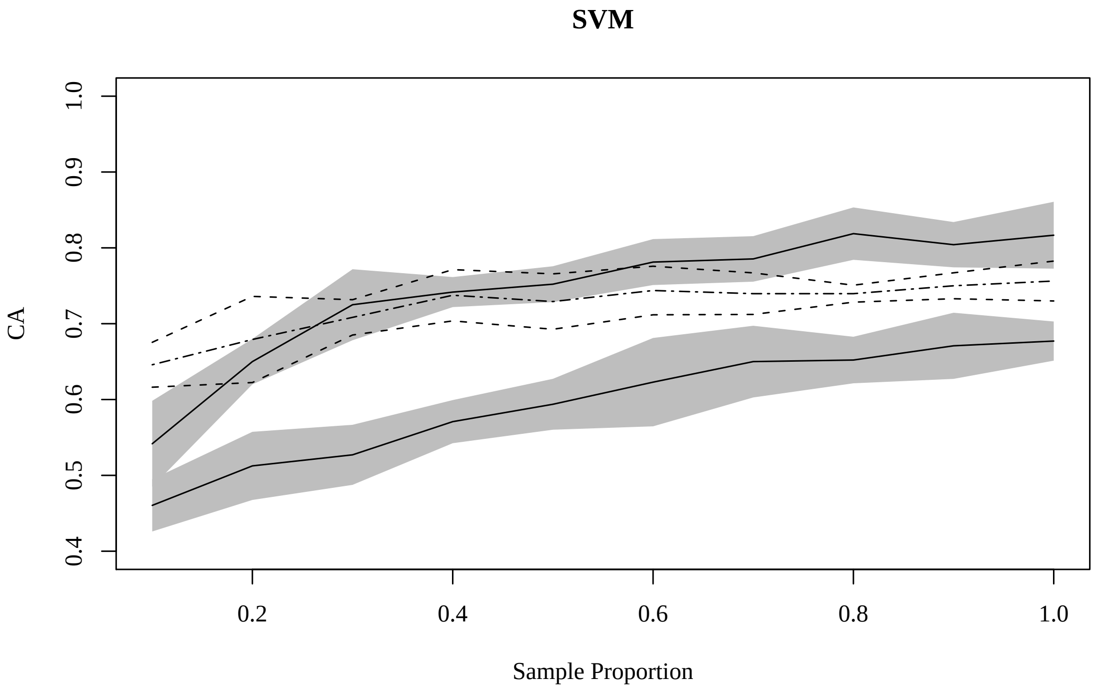


Figure 49: Original Point Dataset Compared to Constructions Limited to Shape and Constructions Limited to XY Coordinate & Shape. From bottom to top, the triples are: original point dataset, constructions from Shape, and constructions from XY & Shape. The datasets are evaluated using stratified 10-fold cross-validation accuracy on the *y*-axis. The upper and lower lines of each triple represent 95% confidence intervals around the accuracy estimates.

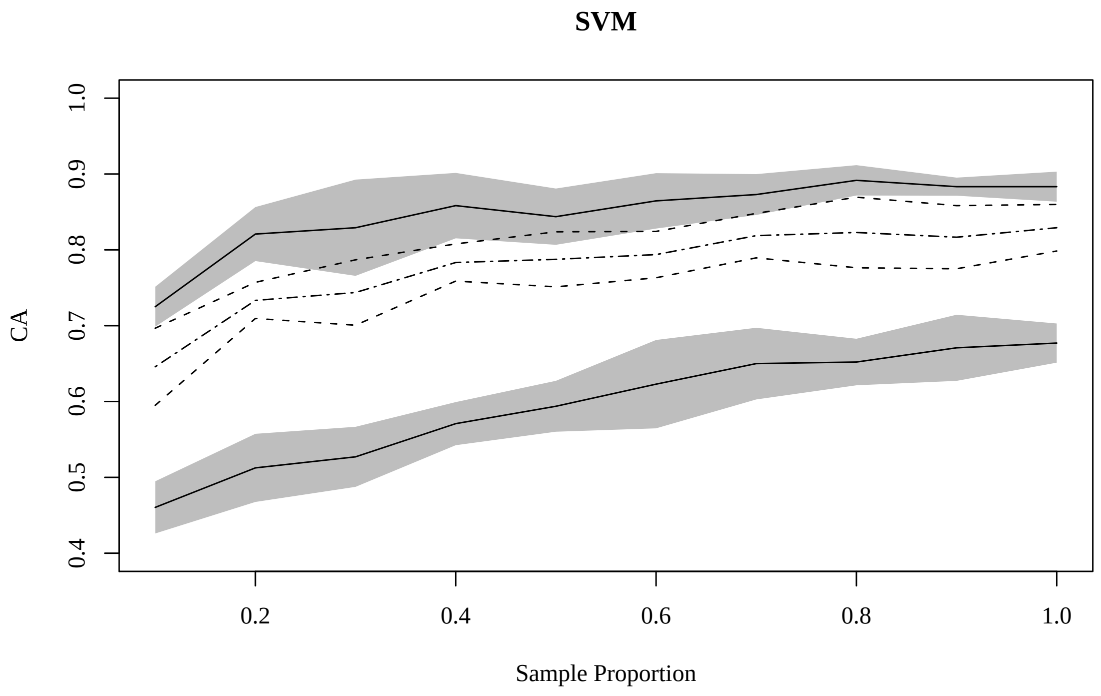


Figure 50: Comparison of Base Point, Base Length, and Detailed KB Constructions. From bottom to top, the triples are: base point to area dataset, constructions from the detailed KB on the base area dataset, and the length to area dataset. The datasets are evaluated using stratified 10-fold cross-validation accuracy on the *y*-axis. The upper and lower lines of each triple represent 95% confidence intervals around the accuracy estimates.

7.3.2 Sampling Effects

The first set of experiments did not address the effect that sampling has on feature construction by ISAK because of the large number of experimental conditions already present in that experiment. Here, I evaluate the effects of sampling on feature construction in the context of the Area problem in the Point representation. The CLP was given a larger search space than was used in the previous runs. The expansion was done by weakening the feature evaluation restrictions. ISAK was then run repeatedly with the sampling parameter set to 10, 20, 30, 40, and 50 percent. Learning curves for an SVM learner with the ConRIF datasets generated by sampling are shown in Figure 51. At 10 percent sampling, there is clearly tremendous variability in the construction of features. By 30 percent, the variability is substantially reduced. It is interesting that there is no distinct drop in accuracy from 30 to 50 percent sampling. The preserved accuracy indicates that even at 50 percent sampling, ISAK is not overfitting the sample data for the Area CLP.

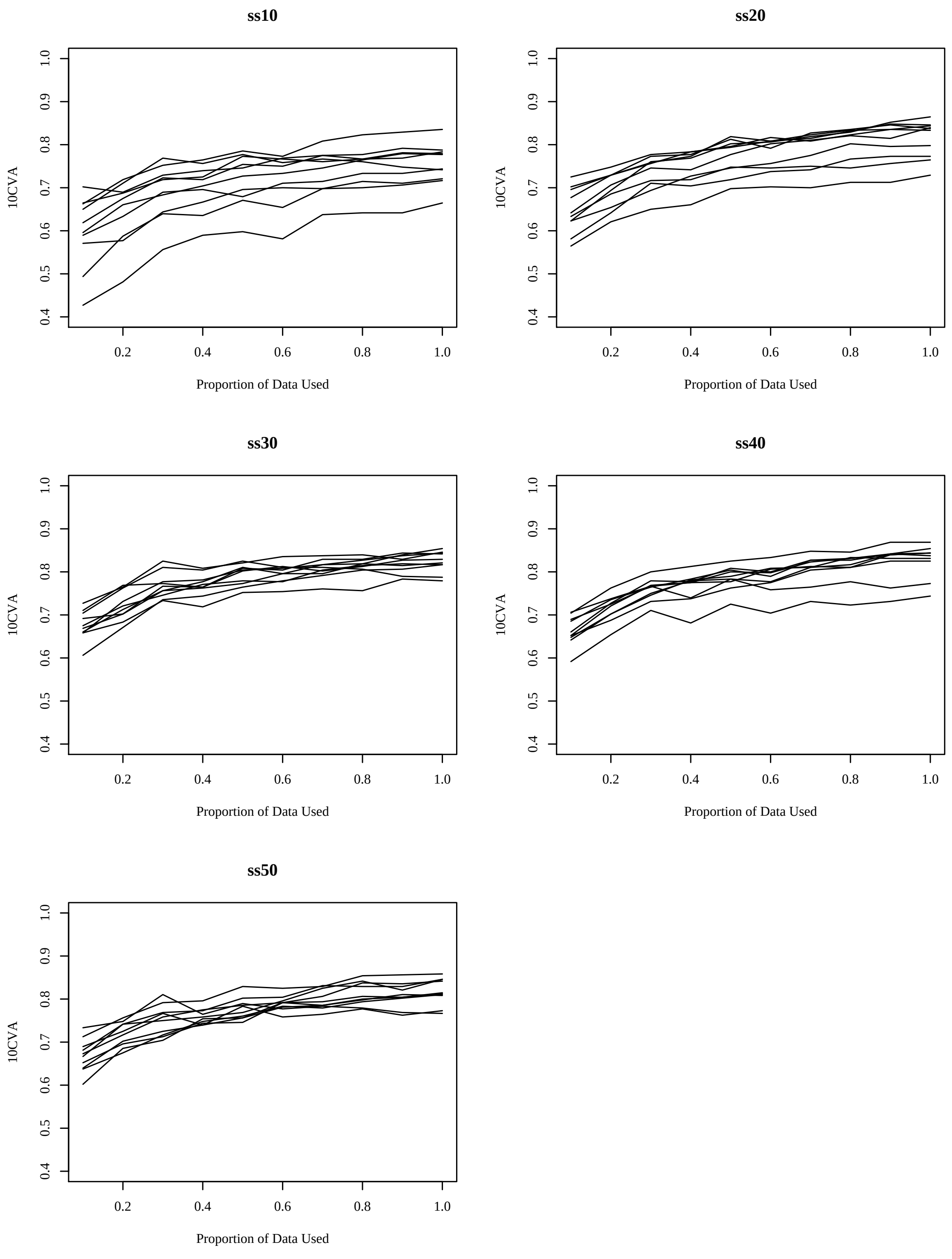


Figure 51: Sampling Effects on Datasets Evaluated by SVM Learning Curves. Each graph is a learning curve. A greater degree of overlap in the curves represents a decrease in the amount of variability due to sampling. As the sample size used for feature construction increases from 10 percent to 50 percent, there is a decrease in the variability of the 10CVA of the SVM.

Table 39: Results of Learning with RL on the Perimeter CLP. The PPV is the number of correct predictions divided by the number of correct + incorrect predictions (i.e., it ignores examples that are unclassified). The classification accuracy is the number of correct predictions divided by the sum of correct, incorrect, and non-predicted examples (i.e., unclassified examples count as wrong).

	Feature Set		
	Base Point Representation	Base + Detailed KB Construction From Point Representation	Base Length Representation
Correct	202	270	264
Incorrect	42	33	47
No Prediction	236	177	169
PPV	82.8%	89.1%	84.9%
Classification Accuracy	42.1%	56.3%	55.0%
Avg. Rule Set Size	27.4	36.2	51.8

7.4 THE BEHAVIOR OF ISAK ON A NOVEL PROBLEM WITH A NOVEL LEARNER

As a final check on the performance of ISAK, I will now look at the performance of ISAK with a new learner and a new problem, neither of which were used in the development of ISAK. The learner is a rule-learner, RL[6]. The CLP is a related to the Area CLP, but instead of area, the target concept is perimeter. The KB used is the same as the detailed KB used for the Area CLP. RL was run with its own bias space search (medium length) looking for good rule-space search parameters. The default 5-fold cross-validation was used for parameter tuning and evaluation within RL. The results for RL on Perimeter are shown in Table 39.

Comparing the base point representation to the detailed constructions, the detailed constructions classify many more examples correctly, several fewer examples incorrectly, and leave fewer examples unclassified. Both the PPV and classification accuracy are improved by detailed construction. However, these improvements come at the cost of a slightly larger rule set.

Comparing the base length representation to the detailed constructions, we see that the number of correct predictions and no prediction is similar, but the number of incorrect predictions is better

[6]RL version 2005-04-27 was used for these experiments and is available from http://www.cs.pitt.edu/˜philip/rl/.

for the detailed constructions. The detailed constructions perform about as well as length on PPV and classification accuracy. The length representation leads to a larger rule set.

8.0 RESULTS AND CONCLUSIONS

8.1 RESULTS

ISAK reinforces several known results:

1. we can capture declarative, symbolic knowledge in uniform representations,
2. the knowledge can be used to guide search,
3. feature construction can be performed as a search through the feature space, and
4. feature construction can improve performance of a learning program.

ISAK extends our knowledge about feature construction:

1. Knowledge based feature construction significantly, though to a moderate extent, improves generalization performance for the class of UCI type learning problems.
2. Knowledge can decrease the amount of time spent in feature construction; however, knowing that there are additional operations (i.e., more possible nodes in the feature space) can extend the amount of time spent in feature construction.
3. Different sets of knowledge affect generalization performance in significantly different ways. Feature construction using domain-specific knowledge reduces the variance in accuracy, as compared to feature construction using general knowledge.
4. Constructed features retain some part of the information of their components. Learning with constructed features uses the retained information and ignores information contained in base features not used for construction. Feature set combinations that include some form of the original, base features (i.e., either as components of constructed features or as the base features themselves) perform substantially better than feature set combinations which exclude information from the base features.

5. Successor functions with complicated built-in heuristics (i.e., windowing of features) are not substantially better than simple heuristics like best and random selection of features for fringe expansion.
6. The generalization performance of learning algorithms that perform implicit feature construction can be improved with explicit, knowledge-guided feature construction.

It is unclear whether knowledge-guided feature construction is *better than* feature construction without knowledge. Exhaustive, knowledge-less search trades time of execution for better performance improvements. However, even knowledge-lean search has limited sets of operators provided to it – which is itself, a source of knowledge and a constraint on the space of constructable features.

8.2 FUTURE WORK

Some interesting extensions of ISAK, and this research, are:

1. Replace the *ad hoc* knowledge representation with an off-the-shelf representation and reasoning system and make use of a modern, extensive ontology like SUMO or Cyc.
2. Specify classes of feature constructors conveniently and enforce relationships among members of the class of constructors. $speed = distance * time$, $d = s/t$, and $t = s/d$ can be seen as taking steps forward or backward from given features. If a forward step is taken, we should be smart enough not to go backward later.
3. Check redundancy of elements with a canonical simplification routine. Programs like Mathematica have a canonical form for a large class of mathematical expressions. Simplification would save time in the feature construction algorithm by replacing a function of the data points with a function of the constructor formula.
4. Perform meta-learning: learn what knowledge bases are useful given descriptions of the learning problem and the target concept.
5. Discover if a well-selected subset of examples can perform substantially better than random selection of an example set as input to ISAK. This is a perfect opportunity to combine hypothesis-driven feature construction with ISAK's knowledge-guided approach. Examples that are mis-

classified by a learner may provide a better basis for constructing feature to be used by that learner.

6. Incorporate ISAK's beam search feature construction as an alternative step within RL. The main forward step in RL is *specialization* of a rule. This means adding a feature to a rule and thereby restricting the rule's applicability. Adding feature construction as a possibility would intertwine the search for a good representation with the search for good hypotheses and give RL true constructive induction capabilities.

8.3 CONCLUSIONS

The results of my work indicate that feature construction improves classification accuracy for a variety of learners over a variety of learning problems. Even relatively powerful learning methods, such as support vector machines, are improved by feature construction. It is also apparent that using constructed features, *in addition to* initial features that are not useful for construction, can represent a significant improvement over the constructed features alone.

Generating features via heuristic search gives a powerful means to apply explicit, declarative knowledge in machine learning. ISAK allows knowledge to be given in a simple, declarative form and it allows the knowledge to be used *in conjunction* with off-the-shelf learning systems. By producing tables of data that are in the same format as the initial problem data, the feature construction step is a modular component of a larger learning process. Incorporating increasing amounts of knowledge results in more efficient learning – the use of fewer data points can give better classification accuracies when done with constructed features. The use of specific feature constructor functions can expand or contract the step size in the search, trading search time and use of domain knowledge. The choices of a particular search strategy and its search parameters are not critical to the success of knowledge-guided feature construction. Knowledge and operators can both be applied to numeric and symbolic features with beneficial learning results.

APPENDIX A

ADDITIONAL EXPERIMENTAL DATA

A.1 COMPARISON OF DIFFERENT GENERATORS AND FEATURE SETS AVERAGED OVER LEARNERS

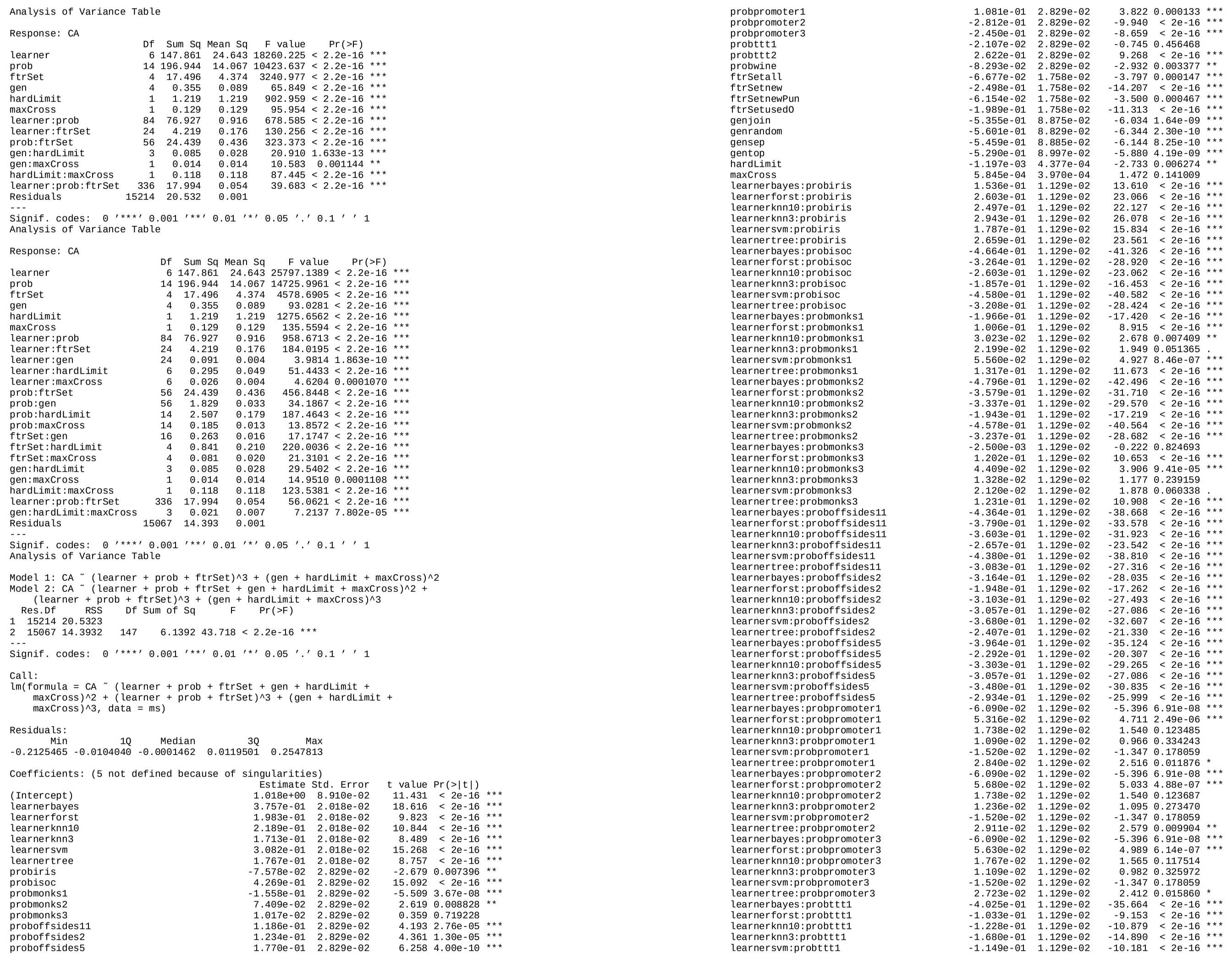
Table 40: 10CVA Averaged over Learner, Run, and CLP by Successor and Features Set.

Base Feature Set CVA: .81				Average 10CVA over All Learners			
					Feature Set		
Successor Function	Maximum Cross	Hard Limit		All	New	ConRIF	CCF
Exhaustive	150	150		0.87	0.83	0.86	0.80
Join	5	10		0.84	0.73	0.84	0.72
Join	5	20		0.85	0.75	0.84	0.73
Join	5	40		0.85	0.75	0.84	0.74
Join	10	20		0.85	0.76	0.84	0.74
Join	10	40		0.86	0.80	0.85	0.77
Join	20	40		0.86	0.81	0.86	0.78
Join	20	60		0.87	0.82	0.86	0.79
Join	20	80		0.86	0.82	0.86	0.79
Random	5	5		0.84	0.71	0.84	0.70
Random	10	10		0.85	0.77	0.85	0.75
Random	20	20		0.86	0.80	0.85	0.77
Random	40	40		0.86	0.82	0.86	0.79
Sep	3	6		0.84	0.73	0.83	0.72
Sep	3	10		0.85	0.76	0.85	0.74
Sep	3	15		0.85	0.76	0.84	0.74
Sep	3	20		0.85	0.77	0.84	0.75
Sep	3	30		0.85	0.77	0.84	0.75
Sep	5	10		0.85	0.76	0.84	0.74
Sep	5	15		0.85	0.75	0.85	0.74
Sep	5	20		0.85	0.76	0.85	0.75
Sep	5	30		0.86	0.78	0.86	0.76
Sep	10	15		0.85	0.77	0.85	0.75
Sep	10	20		0.86	0.78	0.85	0.75
Sep	10	30		0.86	0.79	0.85	0.77
Sep	10	40		0.86	0.79	0.85	0.77
Top	5	5		0.84	0.74	0.83	0.73
Top	10	10		0.85	0.76	0.85	0.75
Top	20	20		0.85	0.78	0.85	0.76
Top	40	40		0.86	0.80	0.85	0.78

Table 41: AUCs Averaged over Learner, Run, and CLP by Successor and Features Set.

				AUC Averaged Over All Learners			
Successor Function	Maximum Cross	Hard Limit		All	New	ConRIF	CCF
Exhaustive	150	150		0.89	0.85	0.88	0.82
Join	5	10		0.88	0.74	0.87	0.73
Join	5	20		0.88	0.76	0.87	0.75
Join	5	40		0.88	0.77	0.87	0.75
Join	10	20		0.88	0.78	0.87	0.76
Join	10	40		0.89	0.81	0.87	0.79
Join	20	40		0.89	0.83	0.88	0.80
Join	20	60		0.89	0.84	0.88	0.81
Join	20	80		0.89	0.84	0.88	0.81
Random	5	5		0.88	0.73	0.87	0.71
Random	10	10		0.89	0.79	0.88	0.76
Random	20	20		0.89	0.81	0.88	0.79
Random	40	40		0.89	0.84	0.88	0.81
Sep	3	6		0.88	0.75	0.86	0.73
Sep	3	10		0.88	0.78	0.88	0.75
Sep	3	15		0.88	0.78	0.87	0.76
Sep	3	20		0.88	0.79	0.87	0.76
Sep	3	30		0.88	0.79	0.87	0.77
Sep	5	10		0.88	0.78	0.87	0.76
Sep	5	15		0.88	0.77	0.88	0.75
Sep	5	20		0.88	0.78	0.88	0.76
Sep	5	30		0.89	0.80	0.88	0.78
Sep	10	15		0.88	0.79	0.87	0.77
Sep	10	20		0.88	0.80	0.87	0.77
Sep	10	30		0.88	0.81	0.87	0.78
Sep	10	40		0.88	0.81	0.88	0.79
Top	5	5		0.87	0.75	0.86	0.74
Top	10	10		0.88	0.78	0.87	0.77
Top	20	20		0.88	0.80	0.87	0.78
Top	40	40		0.88	0.82	0.87	0.80

A.2 A LINEAR MODEL OF THE EXPERIMENTAL VARIABLES FOR THE FIRST SET OF EXPERIMENTS



```
Analysis of Variance Table

Response: CA
                    Df  Sum Sq Mean Sq   F value    Pr(>F)
learner              6 147.861  24.643 18260.225 < 2.2e-16 ***
prob                14 196.944  14.067 10423.637 < 2.2e-16 ***
ftrSet               4  17.496   4.374  3240.977 < 2.2e-16 ***
gen                  4   0.355   0.089    65.849 < 2.2e-16 ***
hardLimit            1   1.219   1.219   902.959 < 2.2e-16 ***
maxCross             1   0.129   0.129    95.954 < 2.2e-16 ***
learner:prob        84  76.927   0.916   678.585 < 2.2e-16 ***
learner:ftrSet      24   4.219   0.176   130.256 < 2.2e-16 ***
prob:ftrSet         56  24.439   0.436   323.373 < 2.2e-16 ***
gen:hardLimit        3   0.085   0.028    20.910 1.633e-13 ***
gen:maxCross         1   0.014   0.014    10.583  0.001144 **
hardLimit:maxCross   1   0.118   0.118    87.445 < 2.2e-16 ***
learner:prob:ftrSet  336  17.994   0.054    39.683 < 2.2e-16 ***
Residuals        15214  20.532   0.001
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Analysis of Variance Table

Response: CA
                       Df  Sum Sq Mean Sq    F value    Pr(>F)
learner                 6 147.861  24.643 25797.1389 < 2.2e-16 ***
prob                   14 196.944  14.067 14725.9961 < 2.2e-16 ***
ftrSet                  4  17.496   4.374  4578.6905 < 2.2e-16 ***
gen                     4   0.355   0.089    93.0281 < 2.2e-16 ***
hardLimit               1   1.219   1.219  1275.6562 < 2.2e-16 ***
maxCross                1   0.129   0.129   135.5594 < 2.2e-16 ***
learner:prob           84  76.927   0.916   958.6713 < 2.2e-16 ***
learner:ftrSet         24   4.219   0.176   184.0195 < 2.2e-16 ***
learner:gen            24   0.091   0.004     3.9814 1.863e-10 ***
learner:hardLimit       6   0.295   0.049    51.4433 < 2.2e-16 ***
learner:maxCross        6   0.026   0.004     4.6204 0.0001070 ***
prob:ftrSet            56  24.439   0.436   456.8448 < 2.2e-16 ***
prob:gen               56   1.829   0.033    34.1867 < 2.2e-16 ***
prob:hardLimit         14   2.507   0.179   187.4643 < 2.2e-16 ***
prob:maxCross          14   0.185   0.013    13.8572 < 2.2e-16 ***
ftrSet:gen             16   0.263   0.016    17.1747 < 2.2e-16 ***
ftrSet:hardLimit        4   0.841   0.210   220.0036 < 2.2e-16 ***
ftrSet:maxCross         4   0.081   0.020    21.3101 < 2.2e-16 ***
gen:hardLimit           3   0.085   0.028    29.5402 < 2.2e-16 ***
gen:maxCross            1   0.014   0.014    14.9510 0.0001108 ***
hardLimit:maxCross      1   0.118   0.118   123.5381 < 2.2e-16 ***
learner:prob:ftrSet   336  17.994   0.054    56.0621 < 2.2e-16 ***
gen:hardLimit:maxCross  3   0.021   0.007     7.2137 7.802e-05 ***
Residuals           15067  14.393   0.001
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Analysis of Variance Table

Model 1: CA ~ (learner + prob + ftrSet)^3 + (gen + hardLimit + maxCross)^2
Model 2: CA ~ (learner + prob + ftrSet + gen + hardLimit + maxCross)^2 +
    (learner + prob + ftrSet)^3 + (gen + hardLimit + maxCross)^3
  Res.Df     RSS    Df Sum of Sq      F    Pr(>F)
1  15214 20.5323
2  15067 14.3932   147    6.1392 43.718 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Call:
lm(formula = CA ~ (learner + prob + ftrSet + gen + hardLimit +
    maxCross)^2 + (learner + prob + ftrSet)^3 + (gen + hardLimit +
    maxCross)^3, data = ms)

Residuals:
       Min         1Q     Median         3Q        Max
-0.2125465 -0.0104040 -0.0001462  0.0119501  0.2547813

Coefficients: (5 not defined because of singularities)
                                   Estimate Std. Error   t value Pr(>|t|)
(Intercept)                       1.018e+00  8.910e-02    11.431  < 2e-16 ***
learnerbayes                      3.757e-01  2.018e-02    18.616  < 2e-16 ***
learnerforst                      1.983e-01  2.018e-02     9.823  < 2e-16 ***
learnerknn10                      2.189e-01  2.018e-02    10.844  < 2e-16 ***
learnerknn3                       1.713e-01  2.018e-02     8.489  < 2e-16 ***
learnersvm                        3.082e-01  2.018e-02    15.268  < 2e-16 ***
learnertree                       1.767e-01  2.018e-02     8.757  < 2e-16 ***
probiris                         -7.578e-02  2.829e-02    -2.679 0.007396 **
probisoc                          4.269e-01  2.829e-02    15.092  < 2e-16 ***
probmonks1                       -1.558e-01  2.829e-02    -5.509 3.67e-08 ***
probmonks2                        7.409e-02  2.829e-02     2.619 0.008828 **
probmonks3                        1.017e-02  2.829e-02     0.359 0.719228
proboffsides11                    1.186e-01  2.829e-02     4.193 2.76e-05 ***
proboffsides2                     1.234e-01  2.829e-02     4.361 1.30e-05 ***
proboffsides5                     1.770e-01  2.829e-02     6.258 4.00e-10 ***
probpromoter1                     1.081e-01  2.829e-02     3.822 0.000133 ***
probpromoter2                    -2.812e-01  2.829e-02    -9.940  < 2e-16 ***
probpromoter3                    -2.450e-01  2.829e-02    -8.659  < 2e-16 ***
probttt1                         -2.107e-02  2.829e-02    -0.745 0.456468
probttt2                          2.622e-01  2.829e-02     9.268  < 2e-16 ***
probwine                         -8.293e-02  2.829e-02    -2.932 0.003377 **
ftrSetall                        -6.677e-02  1.758e-02    -3.797 0.000147 ***
ftrSetnew                        -2.498e-01  1.758e-02   -14.207  < 2e-16 ***
ftrSetnewPun                     -6.154e-02  1.758e-02    -3.500 0.000467 ***
ftrSetusedO                      -1.989e-01  1.758e-02   -11.313  < 2e-16 ***
genjoin                          -5.355e-01  8.875e-02    -6.034 1.64e-09 ***
genrandom                        -5.601e-01  8.829e-02    -6.344 2.30e-10 ***
gensep                           -5.459e-01  8.885e-02    -6.144 8.25e-10 ***
gentop                           -5.290e-01  8.997e-02    -5.880 4.19e-09 ***
hardLimit                        -1.197e-03  4.377e-04    -2.733 0.006274 **
maxCross                          5.845e-04  3.970e-04     1.472 0.141009
learnerbayes:probiris             1.536e-01  1.129e-02    13.610  < 2e-16 ***
learnerforst:probiris             2.603e-01  1.129e-02    23.066  < 2e-16 ***
learnerknn10:probiris             2.497e-01  1.129e-02    22.127  < 2e-16 ***
learnerknn3:probiris              2.943e-01  1.129e-02    26.078  < 2e-16 ***
learnersvm:probiris               1.787e-01  1.129e-02    15.834  < 2e-16 ***
learnertree:probiris              2.659e-01  1.129e-02    23.561  < 2e-16 ***
learnerbayes:probisoc            -4.664e-01  1.129e-02   -41.326  < 2e-16 ***
learnerforst:probisoc            -3.264e-01  1.129e-02   -28.920  < 2e-16 ***
learnerknn10:probisoc            -2.603e-01  1.129e-02   -23.062  < 2e-16 ***
learnerknn3:probisoc             -1.857e-01  1.129e-02   -16.453  < 2e-16 ***
learnersvm:probisoc              -4.580e-01  1.129e-02   -40.582  < 2e-16 ***
learnertree:probisoc             -3.208e-01  1.129e-02   -28.424  < 2e-16 ***
learnerbayes:probmonks1          -1.966e-01  1.129e-02   -17.420  < 2e-16 ***
learnerforst:probmonks1           1.006e-01  1.129e-02     8.915  < 2e-16 ***
learnerknn10:probmonks1           3.023e-02  1.129e-02     2.678 0.007409 **
learnerknn3:probmonks1            2.199e-02  1.129e-02     1.949 0.051365 .
learnersvm:probmonks1             5.560e-02  1.129e-02     4.927 8.46e-07 ***
learnertree:probmonks1            1.317e-01  1.129e-02    11.673  < 2e-16 ***
learnerbayes:probmonks2          -4.796e-01  1.129e-02   -42.496  < 2e-16 ***
learnerforst:probmonks2          -3.579e-01  1.129e-02   -31.710  < 2e-16 ***
learnerknn10:probmonks2          -3.337e-01  1.129e-02   -29.570  < 2e-16 ***
learnerknn3:probmonks2           -1.943e-01  1.129e-02   -17.219  < 2e-16 ***
learnersvm:probmonks2            -4.578e-01  1.129e-02   -40.564  < 2e-16 ***
learnertree:probmonks2           -3.237e-01  1.129e-02   -28.682  < 2e-16 ***
learnerbayes:probmonks3          -2.500e-03  1.129e-02    -0.222 0.824693
learnerforst:probmonks3           1.202e-01  1.129e-02    10.653  < 2e-16 ***
learnerknn10:probmonks3           4.409e-02  1.129e-02     3.906 9.41e-05 ***
learnerknn3:probmonks3            1.328e-02  1.129e-02     1.177 0.239159
learnersvm:probmonks3             2.120e-02  1.129e-02     1.878 0.060338 .
learnertree:probmonks3            1.231e-01  1.129e-02    10.908  < 2e-16 ***
learnerbayes:proboffsides11      -4.364e-01  1.129e-02   -38.668  < 2e-16 ***
learnerforst:proboffsides11      -3.790e-01  1.129e-02   -33.578  < 2e-16 ***
learnerknn10:proboffsides11      -3.603e-01  1.129e-02   -31.923  < 2e-16 ***
learnerknn3:proboffsides11       -2.657e-01  1.129e-02   -23.542  < 2e-16 ***
learnersvm:proboffsides11        -4.380e-01  1.129e-02   -38.810  < 2e-16 ***
learnertree:proboffsides11       -3.083e-01  1.129e-02   -27.316  < 2e-16 ***
learnerbayes:proboffsides2       -3.164e-01  1.129e-02   -28.035  < 2e-16 ***
learnerforst:proboffsides2       -1.948e-01  1.129e-02   -17.262  < 2e-16 ***
learnerknn10:proboffsides2       -3.103e-01  1.129e-02   -27.493  < 2e-16 ***
learnerknn3:proboffsides2        -3.057e-01  1.129e-02   -27.086  < 2e-16 ***
learnersvm:proboffsides2         -3.680e-01  1.129e-02   -32.607  < 2e-16 ***
learnertree:proboffsides2        -2.407e-01  1.129e-02   -21.330  < 2e-16 ***
learnerbayes:proboffsides5       -3.964e-01  1.129e-02   -35.124  < 2e-16 ***
learnerforst:proboffsides5       -2.292e-01  1.129e-02   -20.307  < 2e-16 ***
learnerknn10:proboffsides5       -3.303e-01  1.129e-02   -29.265  < 2e-16 ***
learnerknn3:proboffsides5        -3.057e-01  1.129e-02   -27.086  < 2e-16 ***
learnersvm:proboffsides5         -3.480e-01  1.129e-02   -30.835  < 2e-16 ***
learnertree:proboffsides5        -2.934e-01  1.129e-02   -25.999  < 2e-16 ***
learnerbayes:probpromoter1       -6.090e-02  1.129e-02    -5.396 6.91e-08 ***
learnerforst:probpromoter1        5.316e-02  1.129e-02     4.711 2.49e-06 ***
learnerknn10:probpromoter1        1.738e-02  1.129e-02     1.540 0.123485
learnerknn3:probpromoter1         1.090e-02  1.129e-02     0.966 0.334243
learnersvm:probpromoter1         -1.520e-02  1.129e-02    -1.347 0.178059
learnertree:probpromoter1         2.840e-02  1.129e-02     2.516 0.011876 *
learnerbayes:probpromoter2       -6.090e-02  1.129e-02    -5.396 6.91e-08 ***
learnerforst:probpromoter2        5.680e-02  1.129e-02     5.033 4.88e-07 ***
learnerknn10:probpromoter2        1.738e-02  1.129e-02     1.540 0.123687
learnerknn3:probpromoter2         1.236e-02  1.129e-02     1.095 0.273470
learnersvm:probpromoter2         -1.520e-02  1.129e-02    -1.347 0.178059
learnertree:probpromoter2         2.911e-02  1.129e-02     2.579 0.009904 **
learnerbayes:probpromoter3       -6.090e-02  1.129e-02    -5.396 6.91e-08 ***
learnerforst:probpromoter3        5.630e-02  1.129e-02     4.989 6.14e-07 ***
learnerknn10:probpromoter3        1.767e-02  1.129e-02     1.565 0.117514
learnerknn3:probpromoter3         1.109e-02  1.129e-02     0.982 0.325972
learnersvm:probpromoter3         -1.520e-02  1.129e-02    -1.347 0.178059
learnertree:probpromoter3         2.723e-02  1.129e-02     2.412 0.015860 *
learnerbayes:probttt1            -4.025e-01  1.129e-02   -35.664  < 2e-16 ***
learnerforst:probttt1            -1.033e-01  1.129e-02    -9.153  < 2e-16 ***
learnerknn10:probttt1            -1.228e-01  1.129e-02   -10.879  < 2e-16 ***
learnerknn3:probttt1             -1.680e-01  1.129e-02   -14.890  < 2e-16 ***
learnersvm:probttt1              -1.149e-01  1.129e-02   -10.181  < 2e-16 ***
```

Figure 52: Linear Model Page 1.

```
learnertree:probttt1                -1.315e-01  1.129e-02   -11.655  < 2e-16 ***
learnerbayes:probttt2               -4.025e-01  1.129e-02   -35.664  < 2e-16 ***
learnerforst:probttt2               -1.041e-01  1.129e-02    -9.223  < 2e-16 ***
learnerknn10:probttt2               -1.236e-01  1.129e-02   -10.955  < 2e-16 ***
learnerknn3:probttt2                -1.690e-01  1.129e-02   -14.971  < 2e-16 ***
learnersvm:probttt2                 -1.149e-01  1.129e-02   -10.181  < 2e-16 ***
learnertree:probttt2                -1.315e-01  1.129e-02   -11.650  < 2e-16 ***
learnerbayes:probwine                1.376e-01  1.129e-02    12.192  < 2e-16 ***
learnerforst:probwine                2.256e-01  1.129e-02    19.988  < 2e-16 ***
learnerknn10:probwine                1.850e-01  1.129e-02    16.394  < 2e-16 ***
learnerknn3:probwine                 2.296e-01  1.129e-02    20.345  < 2e-16 ***
learnersvm:probwine                  1.360e-01  1.129e-02    12.050  < 2e-16 ***
learnertree:probwine                 1.905e-01  1.129e-02    16.883  < 2e-16 ***
learnerbayes:ftrSetall              -3.138e-01  1.129e-02   -27.802  < 2e-16 ***
learnerforst:ftrSetall               1.787e-01  1.129e-02    15.832  < 2e-16 ***
learnerknn10:ftrSetall               1.514e-01  1.129e-02    13.413  < 2e-16 ***
learnerknn3:ftrSetall                1.898e-01  1.129e-02    16.818  < 2e-16 ***
learnersvm:ftrSetall                 7.815e-02  1.129e-02     6.925 4.54e-12 ***
learnertree:ftrSetall                1.354e-01  1.129e-02    11.998  < 2e-16 ***
learnerbayes:ftrSetnew              -3.147e-01  1.129e-02   -27.885  < 2e-16 ***
learnerforst:ftrSetnew               1.733e-01  1.129e-02    15.352  < 2e-16 ***
learnerknn10:ftrSetnew               1.505e-01  1.129e-02    13.338  < 2e-16 ***
learnerknn3:ftrSetnew                1.908e-01  1.129e-02    16.905  < 2e-16 ***
learnersvm:ftrSetnew                 6.962e-02  1.129e-02     6.169 7.05e-10 ***
learnertree:ftrSetnew                1.352e-01  1.129e-02    11.981  < 2e-16 ***
learnerbayes:ftrSetnewPun           -3.146e-01  1.129e-02   -27.879  < 2e-16 ***
learnerforst:ftrSetnewPun            1.740e-01  1.129e-02    15.417  < 2e-16 ***
learnerknn10:ftrSetnewPun            1.510e-01  1.129e-02    13.383  < 2e-16 ***
learnerknn3:ftrSetnewPun             1.915e-01  1.129e-02    16.966  < 2e-16 ***
learnersvm:ftrSetnewPun              7.005e-02  1.129e-02     6.207 5.54e-10 ***
learnertree:ftrSetnewPun             1.355e-01  1.129e-02    12.007  < 2e-16 ***
learnerbayes:ftrSetusedO            -4.003e-04  1.129e-02    -0.035 0.971704
learnerforst:ftrSetusedO             3.145e-03  1.129e-02     0.279 0.780526
learnerknn10:ftrSetusedO            -2.814e-03  1.129e-02    -0.249 0.803102
learnerknn3:ftrSetusedO              3.379e-03  1.129e-02     0.299 0.764660
learnersvm:ftrSetusedO              -4.747e-04  1.129e-02    -0.042 0.966453
learnertree:ftrSetusedO              6.617e-04  1.129e-02     0.059 0.953249
learnerbayes:genjoin                 6.141e-02  1.799e-02     3.413 0.000643 ***
learnerforst:genjoin                 1.241e-01  1.799e-02     6.898 5.47e-12 ***
learnerknn10:genjoin                 1.258e-01  1.799e-02     6.990 2.87e-12 ***
learnerknn3:genjoin                  1.283e-01  1.799e-02     7.129 1.05e-12 ***
learnersvm:genjoin                   1.149e-01  1.799e-02     6.385 1.76e-10 ***
learnertree:genjoin                  1.194e-01  1.799e-02     6.636 3.34e-11 ***
learnerbayes:genrandom               6.046e-02  1.662e-02     3.638 0.000276 ***
learnerforst:genrandom               1.265e-01  1.662e-02     7.612 2.86e-14 ***
learnerknn10:genrandom               1.284e-01  1.662e-02     7.724 1.19e-14 ***
learnerknn3:genrandom                1.302e-01  1.662e-02     7.838 4.89e-15 ***
learnersvm:genrandom                 1.196e-01  1.662e-02     7.200 6.31e-13 ***
learnertree:genrandom                1.226e-01  1.662e-02     7.380 1.66e-13 ***
learnerbayes:gensep                  6.383e-02  1.829e-02     3.490 0.000484 ***
learnerforst:gensep                  1.308e-01  1.829e-02     7.152 8.96e-13 ***
learnerknn10:gensep                  1.329e-01  1.829e-02     7.270 3.78e-13 ***
learnerknn3:gensep                   1.363e-01  1.829e-02     7.451 9.75e-14 ***
learnersvm:gensep                    1.212e-01  1.829e-02     6.627 3.55e-11 ***
learnertree:gensep                   1.264e-01  1.829e-02     6.911 5.01e-12 ***
learnerbayes:gentop                  5.997e-02  1.662e-02     3.609 0.000309 ***
learnerforst:gentop                  1.231e-01  1.662e-02     7.410 1.33e-13 ***
learnerknn10:gentop                  1.258e-01  1.662e-02     7.568 4.00e-14 ***
learnerknn3:gentop                   1.284e-01  1.662e-02     7.727 1.17e-14 ***
learnersvm:gentop                    1.165e-01  1.662e-02     7.010 2.49e-12 ***
learnertree:gentop                   1.203e-01  1.662e-02     7.241 4.65e-13 ***
learnerbayes:hardLimit               1.595e-04  9.090e-05     1.755 0.079361 .
learnerforst:hardLimit               4.950e-04  9.090e-05     5.446 5.24e-08 ***
learnerknn10:hardLimit               5.482e-04  9.090e-05     6.032 1.66e-09 ***
learnerknn3:hardLimit                5.414e-04  9.090e-05     5.956 2.64e-09 ***
learnersvm:hardLimit                 5.612e-04  9.090e-05     6.174 6.82e-10 ***
learnertree:hardLimit                5.306e-04  9.090e-05     5.838 5.39e-09 ***
learnerbayes:maxCross                3.791e-04  1.656e-04     2.289 0.022090 *
learnerforst:maxCross                6.828e-04  1.656e-04     4.123 3.76e-05 ***
learnerknn10:maxCross                6.669e-04  1.656e-04     4.027 5.67e-05 ***
learnerknn3:maxCross                 6.947e-04  1.656e-04     4.195 2.74e-05 ***
learnersvm:maxCross                  5.683e-04  1.656e-04     3.432 0.000601 ***
learnertree:maxCross                 6.154e-04  1.656e-04     3.716 0.000203 ***
probiris:ftrSetall                  -5.516e-16  1.129e-02 -4.89e-14 1.000000
probisoc:ftrSetall                  -4.552e-16  1.129e-02 -4.03e-14 1.000000
probmonks1:ftrSetall                -5.026e-16  1.129e-02 -4.45e-14 1.000000
probmonks2:ftrSetall                -4.899e-16  1.129e-02 -4.34e-14 1.000000
probmonks3:ftrSetall                -5.319e-16  1.129e-02 -4.71e-14 1.000000
proboffsides11:ftrSetall            -5.784e-16  1.129e-02 -5.13e-14 1.000000
proboffsides2:ftrSetall             -5.536e-16  1.129e-02 -4.91e-14 1.000000
proboffsides5:ftrSetall             -6.402e-16  1.129e-02 -5.67e-14 1.000000
probpromoter1:ftrSetall             -4.601e-16  1.129e-02 -4.08e-14 1.000000
probpromoter2:ftrSetall             -6.114e-16  1.129e-02 -5.42e-14 1.000000
probpromoter3:ftrSetall             -6.239e-16  1.129e-02 -5.53e-14 1.000000
probttt1:ftrSetall                  -6.659e-16  1.129e-02 -5.90e-14 1.000000
probttt2:ftrSetall                  -6.277e-16  1.129e-02 -5.56e-14 1.000000
probwine:ftrSetall                  -5.791e-16  1.129e-02 -5.13e-14 1.000000
probiris:ftrSetnew                  -5.557e-16  1.129e-02 -4.92e-14 1.000000
probisoc:ftrSetnew                  -3.563e-16  1.129e-02 -3.16e-14 1.000000
probmonks1:ftrSetnew                -5.938e-16  1.129e-02 -5.26e-14 1.000000
probmonks2:ftrSetnew                -3.376e-16  1.129e-02 -2.99e-14 1.000000
probmonks3:ftrSetnew                -5.689e-16  1.129e-02 -5.04e-14 1.000000
proboffsides11:ftrSetnew            -5.473e-16  1.129e-02 -4.85e-14 1.000000
proboffsides2:ftrSetnew             -4.432e-16  1.129e-02 -3.93e-14 1.000000
proboffsides5:ftrSetnew             -4.358e-16  1.129e-02 -3.86e-14 1.000000
probpromoter1:ftrSetnew             -4.156e-16  1.129e-02 -3.68e-14 1.000000
probpromoter2:ftrSetnew             -5.154e-16  1.129e-02 -4.57e-14 1.000000
probpromoter3:ftrSetnew             -4.726e-16  1.129e-02 -4.19e-14 1.000000
probttt1:ftrSetnew                  -5.059e-16  1.129e-02 -4.48e-14 1.000000
probttt2:ftrSetnew                  -6.631e-16  1.129e-02 -5.88e-14 1.000000
probwine:ftrSetnew                  -3.685e-16  1.129e-02 -3.26e-14 1.000000
probiris:ftrSetnewPun               -5.332e-16  1.129e-02 -4.72e-14 1.000000
probisoc:ftrSetnewPun               -3.343e-16  1.129e-02 -2.96e-14 1.000000
probmonks1:ftrSetnewPun             -5.785e-16  1.129e-02 -5.13e-14 1.000000
probmonks2:ftrSetnewPun             -3.607e-16  1.129e-02 -3.20e-14 1.000000
probmonks3:ftrSetnewPun             -4.281e-16  1.129e-02 -3.79e-14 1.000000
proboffsides11:ftrSetnewPun         -5.021e-16  1.129e-02 -4.45e-14 1.000000
proboffsides2:ftrSetnewPun          -5.500e-16  1.129e-02 -4.87e-14 1.000000
proboffsides5:ftrSetnewPun          -4.824e-16  1.129e-02 -4.27e-14 1.000000
probpromoter1:ftrSetnewPun          -4.006e-16  1.129e-02 -3.55e-14 1.000000
probpromoter2:ftrSetnewPun          -5.078e-16  1.129e-02 -4.50e-14 1.000000
probpromoter3:ftrSetnewPun          -4.419e-16  1.129e-02 -3.92e-14 1.000000
probttt1:ftrSetnewPun               -5.556e-16  1.129e-02 -4.92e-14 1.000000
probttt2:ftrSetnewPun               -6.410e-16  1.129e-02 -5.68e-14 1.000000
probwine:ftrSetnewPun               -5.437e-16  1.129e-02 -4.82e-14 1.000000
probiris:ftrSetusedO                -5.789e-16  1.129e-02 -5.13e-14 1.000000
probisoc:ftrSetusedO                -3.814e-16  1.129e-02 -3.38e-14 1.000000
probmonks1:ftrSetusedO              -6.039e-16  1.129e-02 -5.35e-14 1.000000
probmonks2:ftrSetusedO              -5.084e-16  1.129e-02 -4.51e-14 1.000000
probmonks3:ftrSetusedO              -5.565e-16  1.129e-02 -4.93e-14 1.000000
proboffsides11:ftrSetusedO          -5.803e-16  1.129e-02 -5.14e-14 1.000000
proboffsides2:ftrSetusedO           -5.978e-16  1.129e-02 -5.30e-14 1.000000
proboffsides5:ftrSetusedO           -5.529e-16  1.129e-02 -4.90e-14 1.000000
probpromoter1:ftrSetusedO           -4.823e-16  1.129e-02 -4.27e-14 1.000000
probpromoter2:ftrSetusedO           -6.304e-16  1.129e-02 -5.59e-14 1.000000
probpromoter3:ftrSetusedO           -5.312e-16  1.129e-02 -4.71e-14 1.000000
probttt1:ftrSetusedO                -7.845e-16  1.129e-02 -6.95e-14 1.000000
probttt2:ftrSetusedO                -7.504e-16  1.129e-02 -6.65e-14 1.000000
probwine:ftrSetusedO                -5.388e-16  1.129e-02 -4.77e-14 1.000000
probiris:genjoin                    -5.005e-02  2.634e-02    -1.901 0.057377 .
probisoc:genjoin                    -6.447e-02  2.634e-02    -2.448 0.014379 *
probmonks1:genjoin                   1.801e-01  2.634e-02     6.838 8.35e-12 ***
probmonks2:genjoin                   8.035e-02  2.634e-02     3.051 0.002285 **
probmonks3:genjoin                   4.906e-02  2.634e-02     1.863 0.062526 .
proboffsides11:genjoin              -8.257e-02  2.634e-02    -3.135 0.001722 **
proboffsides2:genjoin               -1.385e-02  2.634e-02    -0.526 0.599023
proboffsides5:genjoin               -5.696e-02  2.634e-02    -2.163 0.030565 *
probpromoter1:genjoin               -9.017e-02  2.634e-02    -3.424 0.000619 ***
probpromoter2:genjoin                2.298e-01  2.634e-02     8.726  < 2e-16 ***
probpromoter3:genjoin                1.860e-01  2.634e-02     7.064 1.69e-12 ***
probttt1:genjoin                     1.612e-01  2.634e-02     6.121 9.50e-10 ***
probttt2:genjoin                    -6.925e-02  2.634e-02    -2.629 0.008565 **
probwine:genjoin                     9.271e-03  2.634e-02     0.352 0.724826
probiris:genrandom                  -2.641e-02  2.433e-02    -1.086 0.277670
probisoc:genrandom                  -4.946e-02  2.433e-02    -2.033 0.042044 *
probmonks1:genrandom                 1.617e-01  2.433e-02     6.649 3.06e-11 ***
probmonks2:genrandom                 1.270e-01  2.433e-02     5.219 1.82e-07 ***
probmonks3:genrandom                 3.112e-02  2.433e-02     1.279 0.200837
proboffsides11:genrandom            -6.593e-02  2.433e-02    -2.710 0.006728 **
proboffsides2:genrandom              2.979e-03  2.433e-02     0.122 0.902547
proboffsides5:genrandom             -4.498e-02  2.433e-02    -1.849 0.064447 .
probpromoter1:genrandom             -6.741e-02  2.433e-02    -2.771 0.005595 **
probpromoter2:genrandom              2.687e-01  2.433e-02    11.045  < 2e-16 ***
probpromoter3:genrandom              2.211e-01  2.433e-02     9.087  < 2e-16 ***
probttt1:genrandom                   1.907e-01  2.433e-02     7.837 4.90e-15 ***
probttt2:genrandom                  -4.818e-02  2.433e-02    -1.981 0.047652 *
probwine:genrandom                   2.141e-02  2.433e-02     0.880 0.378690
probiris:gensep                     -4.083e-02  2.677e-02    -1.525 0.127260
probisoc:gensep                     -6.005e-02  2.677e-02    -2.243 0.024900 *
probmonks1:gensep                    1.926e-01  2.677e-02     7.193 6.64e-13 ***
probmonks2:gensep                    1.057e-01  2.677e-02     3.949 7.89e-05 ***
probmonks3:gensep                    5.717e-02  2.677e-02     2.135 0.032741 *
proboffsides11:gensep               -8.332e-02  2.677e-02    -3.112 0.001859 **
proboffsides2:gensep                -4.049e-03  2.677e-02    -0.151 0.879776
proboffsides5:gensep                -4.812e-02  2.677e-02    -1.798 0.072261 .
probpromoter1:gensep                -8.842e-02  2.677e-02    -3.303 0.000959 ***
probpromoter2:gensep                 2.225e-01  2.677e-02     8.312  < 2e-16 ***
probpromoter3:gensep                 2.000e-01  2.677e-02     7.471 8.37e-14 ***
probttt1:gensep                      2.080e-01  2.677e-02     7.769 8.45e-15 ***
probttt2:gensep                     -6.484e-02  2.677e-02    -2.422 0.015440 *
probwine:gensep                      2.143e-02  2.677e-02     0.801 0.423430
probiris:gentop                     -3.796e-02  2.433e-02    -1.560 0.118695
probisoc:gentop                     -6.015e-02  2.433e-02    -2.473 0.013417 *
probmonks1:gentop                    1.729e-01  2.433e-02     7.108 1.23e-12 ***
```

Figure 53: Linear Model Page 2.

```
probmonks2:gentop                          7.989e-02  2.433e-02    3.284 0.001026 **
probmonks3:gentop                          4.380e-02  2.433e-02    1.800 0.071814 .
proboffsides11:gentop                     -7.847e-02  2.433e-02   -3.226 0.001259 **
proboffsides2:gentop                      -3.035e-03  2.433e-02   -0.125 0.900707
proboffsides5:gentop                      -5.522e-02  2.433e-02   -2.270 0.023224 *
probpromoter1:gentop                      -8.351e-02  2.433e-02   -3.433 0.000599 ***
probpromoter2:gentop                       2.195e-01  2.433e-02    9.025  < 2e-16 ***
probpromoter3:gentop                       2.075e-01  2.433e-02    8.530  < 2e-16 ***
probttt1:gentop                            1.797e-01  2.433e-02    7.386 1.59e-13 ***
probttt2:gentop                           -6.373e-02  2.433e-02   -2.620 0.008802 **
probwine:gentop                            1.378e-02  2.433e-02    0.567 0.571021
probiris:hardLimit                        -3.179e-04  1.331e-04   -2.389 0.016911 *
probisoc:hardLimit                         3.867e-05  1.331e-04    0.291 0.771336
probmonks1:hardLimit                      -2.347e-04  1.331e-04   -1.764 0.077761 .
probmonks2:hardLimit                       1.156e-03  1.331e-04    8.689  < 2e-16 ***
probmonks3:hardLimit                      -8.829e-05  1.331e-04   -0.664 0.507012
proboffsides11:hardLimit                  -4.987e-05  1.331e-04   -0.375 0.707794
proboffsides2:hardLimit                    1.957e-04  1.331e-04    1.471 0.141371
proboffsides5:hardLimit                   -1.930e-05  1.331e-04   -0.145 0.884666
probpromoter1:hardLimit                    2.552e-05  1.331e-04    0.192 0.847917
probpromoter2:hardLimit                    2.240e-03  1.331e-04   16.838  < 2e-16 ***
probpromoter3:hardLimit                    2.517e-03  1.331e-04   18.914  < 2e-16 ***
probttt1:hardLimit                         6.467e-04  1.331e-04    4.860 1.18e-06 ***
probttt2:hardLimit                         1.068e-04  1.331e-04    0.803 0.422245
probwine:hardLimit                         4.152e-04  1.331e-04    3.120 0.001810 **
probiris:maxCross                          2.102e-05  2.424e-04    0.087 0.930905
probisoc:maxCross                         -4.991e-04  2.424e-04   -2.059 0.039513 *
probmonks1:maxCross                        1.619e-03  2.424e-04    6.678 2.50e-11 ***
probmonks2:maxCross                       -1.828e-04  2.424e-04   -0.754 0.450794
probmonks3:maxCross                        4.373e-04  2.424e-04    1.804 0.071239 .
proboffsides11:maxCross                   -4.723e-04  2.424e-04   -1.948 0.051394 .
proboffsides2:maxCross                    -7.552e-05  2.424e-04   -0.312 0.755403
proboffsides5:maxCross                    -2.983e-04  2.424e-04   -1.230 0.218574
probpromoter1:maxCross                    -7.197e-04  2.424e-04   -2.969 0.002995 **
probpromoter2:maxCross                     5.395e-04  2.424e-04    2.225 0.026074 *
probpromoter3:maxCross                    -4.099e-04  2.424e-04   -1.691 0.090882 .
probttt1:maxCross                          9.962e-04  2.424e-04    4.109 3.99e-05 ***
probttt2:maxCross                         -5.096e-04  2.424e-04   -2.102 0.035560 *
probwine:maxCross                         -2.473e-04  2.424e-04   -1.020 0.307671
ftrSetall:genjoin                          5.692e-02  1.521e-02    3.743 0.000182 ***
ftrSetnew:genjoin                          2.079e-01  1.521e-02   13.672  < 2e-16 ***
ftrSetnewPun:genjoin                       5.246e-02  1.521e-02    3.450 0.000561 ***
ftrSetusedO:genjoin                        1.644e-01  1.521e-02   10.812  < 2e-16 ***
ftrSetall:genrandom                        5.770e-02  1.404e-02    4.108 4.00e-05 ***
ftrSetnew:genrandom                        2.118e-01  1.404e-02   15.079  < 2e-16 ***
ftrSetnewPun:genrandom                     5.303e-02  1.404e-02    3.776 0.000160 ***
ftrSetusedO:genrandom                      1.677e-01  1.404e-02   11.941  < 2e-16 ***
ftrSetall:gensep                           5.939e-02  1.546e-02    3.842 0.000122 ***
ftrSetnew:gensep                           2.198e-01  1.546e-02   14.220  < 2e-16 ***
ftrSetnewPun:gensep                        5.500e-02  1.546e-02    3.559 0.000374 ***
ftrSetusedO:gensep                         1.733e-01  1.546e-02   11.215  < 2e-16 ***
ftrSetall:gentop                           5.384e-02  1.404e-02    3.834 0.000127 ***
ftrSetnew:gentop                           2.091e-01  1.404e-02   14.889  < 2e-16 ***
ftrSetnewPun:gentop                        4.961e-02  1.404e-02    3.532 0.000413 ***
ftrSetusedO:gentop                         1.678e-01  1.404e-02   11.945  < 2e-16 ***
ftrSetall:hardLimit                        2.200e-04  7.682e-05    2.864 0.004195 **
ftrSetnew:hardLimit                        8.797e-04  7.682e-05   11.451  < 2e-16 ***
ftrSetnewPun:hardLimit                     1.845e-04  7.682e-05    2.402 0.016315 *
ftrSetusedO:hardLimit                      7.546e-04  7.682e-05    9.823  < 2e-16 ***
ftrSetall:maxCross                         2.938e-04  1.400e-04    2.099 0.035851 *
ftrSetnew:maxCross                         1.116e-03  1.400e-04    7.971 1.68e-15 ***
ftrSetnewPun:maxCross                      2.985e-04  1.400e-04    2.133 0.032977 *
ftrSetusedO:maxCross                       8.513e-04  1.400e-04    6.082 1.21e-09 ***
genjoin:hardLimit                          4.831e-04  4.280e-04    1.129 0.259017
genrandom:hardLimit                        1.744e-03  3.657e-04    4.768 1.87e-06 ***
gensep:hardLimit                           7.080e-04  4.820e-04    1.469 0.141906
gentop:hardLimit                                  NA         NA       NA       NA
genjoin:maxCross                           6.485e-04  3.934e-04    1.649 0.099231 .
genrandom:maxCross                                NA         NA       NA       NA
gensep:maxCross                                   NA         NA       NA       NA
gentop:maxCross                                   NA         NA       NA       NA
hardLimit:maxCross                        -2.264e-05  5.477e-06   -4.134 3.59e-05 ***
learnerbayes:probiris:ftrSetall            3.405e-01  1.596e-02   21.332  < 2e-16 ***
learnerforst:probiris:ftrSetall           -1.654e-01  1.596e-02  -10.365  < 2e-16 ***
learnerknn10:probiris:ftrSetall           -1.462e-01  1.596e-02   -9.162  < 2e-16 ***
learnerknn3:probiris:ftrSetall            -1.915e-01  1.596e-02  -12.001  < 2e-16 ***
learnersvm:probiris:ftrSetall             -7.093e-02  1.596e-02   -4.444 8.88e-06 ***
learnertree:probiris:ftrSetall            -1.097e-01  1.596e-02   -6.873 6.55e-12 ***
learnerbayes:probisoc:ftrSetall            4.938e-01  1.596e-02   30.937  < 2e-16 ***
learnerforst:probisoc:ftrSetall           -1.918e-02  1.596e-02   -1.202 0.229519
learnerknn10:probisoc:ftrSetall           -8.137e-02  1.596e-02   -5.098 3.47e-07 ***
learnerknn3:probisoc:ftrSetall            -1.498e-01  1.596e-02   -9.386  < 2e-16 ***
learnersvm:probisoc:ftrSetall              1.118e-01  1.596e-02    7.008 2.53e-12 ***
learnertree:probisoc:ftrSetall             4.460e-02  1.596e-02    2.794 0.005208 **
learnerbayes:probmonks1:ftrSetall          5.413e-01  1.596e-02   33.918  < 2e-16 ***
learnerforst:probmonks1:ftrSetall         -1.275e-01  1.596e-02   -7.990 1.45e-15 ***
learnerknn10:probmonks1:ftrSetall         -6.008e-02  1.596e-02   -3.764 0.000168 ***
```

```
learnerknn3:probmonks1:ftrSetall          -5.374e-02  1.596e-02   -3.367 0.000761 ***
learnersvm:probmonks1:ftrSetall           -7.815e-02  1.596e-02   -4.897 9.85e-07 ***
learnertree:probmonks1:ftrSetall          -8.969e-02  1.596e-02   -5.619 1.95e-08 ***
learnerbayes:probmonks2:ftrSetall          3.323e-01  1.596e-02   20.821  < 2e-16 ***
learnerforst:probmonks2:ftrSetall          7.894e-03  1.596e-02    0.495 0.620909
learnerknn10:probmonks2:ftrSetall         -3.657e-04  1.596e-02   -0.023 0.981722
learnerknn3:probmonks2:ftrSetall          -1.214e-01  1.596e-02   -7.603 3.06e-14 ***
learnersvm:probmonks2:ftrSetall            1.109e-01  1.596e-02    6.949 3.84e-12 ***
learnertree:probmonks2:ftrSetall           1.598e-01  1.596e-02   10.010  < 2e-16 ***
learnerbayes:probmonks3:ftrSetall          3.138e-01  1.596e-02   19.659  < 2e-16 ***
learnerforst:probmonks3:ftrSetall         -1.998e-01  1.596e-02  -12.518  < 2e-16 ***
learnerknn10:probmonks3:ftrSetall         -1.063e-01  1.596e-02   -6.661 2.81e-11 ***
learnerknn3:probmonks3:ftrSetall          -8.808e-02  1.596e-02   -5.518 3.48e-08 ***
learnersvm:probmonks3:ftrSetall           -7.815e-02  1.596e-02   -4.897 9.85e-07 ***
learnertree:probmonks3:ftrSetall          -1.354e-01  1.596e-02   -8.484  < 2e-16 ***
learnerbayes:proboffsides11:ftrSetall      3.322e-01  1.596e-02   20.812  < 2e-16 ***
learnerforst:proboffsides11:ftrSetall     -1.502e-01  1.596e-02   -9.409  < 2e-16 ***
learnerknn10:proboffsides11:ftrSetall     -1.540e-01  1.596e-02   -9.651  < 2e-16 ***
learnerknn3:proboffsides11:ftrSetall      -2.300e-01  1.596e-02  -14.409  < 2e-16 ***
learnersvm:proboffsides11:ftrSetall       -7.269e-02  1.596e-02   -4.554 5.30e-06 ***
learnertree:proboffsides11:ftrSetall      -1.231e-01  1.596e-02   -7.713 1.31e-14 ***
learnerbayes:proboffsides2:ftrSetall       3.168e-01  1.596e-02   19.847  < 2e-16 ***
learnerforst:proboffsides2:ftrSetall      -1.670e-01  1.596e-02  -10.466  < 2e-16 ***
learnerknn10:proboffsides2:ftrSetall      -1.106e-01  1.596e-02   -6.932 4.32e-12 ***
learnerknn3:proboffsides2:ftrSetall       -1.495e-01  1.596e-02   -9.367  < 2e-16 ***
learnersvm:proboffsides2:ftrSetall        -1.032e-02  1.596e-02   -0.647 0.517865
learnertree:proboffsides2:ftrSetall       -9.400e-02  1.596e-02   -5.890 3.95e-09 ***
learnerbayes:proboffsides5:ftrSetall       3.531e-01  1.596e-02   22.121  < 2e-16 ***
learnerforst:proboffsides5:ftrSetall      -1.775e-01  1.596e-02  -11.124  < 2e-16 ***
learnerknn10:proboffsides5:ftrSetall      -1.507e-01  1.596e-02   -9.445  < 2e-16 ***
learnerknn3:proboffsides5:ftrSetall       -1.903e-01  1.596e-02  -11.922  < 2e-16 ***
learnersvm:proboffsides5:ftrSetall        -9.952e-02  1.596e-02   -6.235 4.63e-10 ***
learnertree:proboffsides5:ftrSetall       -1.228e-01  1.596e-02   -7.696 1.49e-14 ***
learnerbayes:probpromoter1:ftrSetall       2.517e-01  1.596e-02   15.771  < 2e-16 ***
learnerforst:probpromoter1:ftrSetall      -1.809e-01  1.596e-02  -11.335  < 2e-16 ***
learnerknn10:probpromoter1:ftrSetall      -1.452e-01  1.596e-02   -9.096  < 2e-16 ***
learnerknn3:probpromoter1:ftrSetall       -1.850e-01  1.596e-02  -11.588  < 2e-16 ***
learnersvm:probpromoter1:ftrSetall        -7.593e-02  1.596e-02   -4.757 1.98e-06 ***
learnertree:probpromoter1:ftrSetall       -1.348e-01  1.596e-02   -8.443  < 2e-16 ***
learnerbayes:probpromoter2:ftrSetall       3.245e-01  1.596e-02   20.333  < 2e-16 ***
learnerforst:probpromoter2:ftrSetall      -1.662e-01  1.596e-02  -10.414  < 2e-16 ***
learnerknn10:probpromoter2:ftrSetall      -1.386e-01  1.596e-02   -8.685  < 2e-16 ***
learnerknn3:probpromoter2:ftrSetall       -1.739e-01  1.596e-02  -10.894  < 2e-16 ***
learnersvm:probpromoter2:ftrSetall        -6.901e-02  1.596e-02   -4.324 1.54e-05 ***
learnertree:probpromoter2:ftrSetall       -1.117e-01  1.596e-02   -7.000 2.67e-12 ***
learnerbayes:probpromoter3:ftrSetall       2.637e-01  1.596e-02   16.521  < 2e-16 ***
learnerforst:probpromoter3:ftrSetall      -1.695e-01  1.596e-02  -10.623  < 2e-16 ***
learnerknn10:probpromoter3:ftrSetall      -1.317e-01  1.596e-02   -8.255  < 2e-16 ***
learnerknn3:probpromoter3:ftrSetall       -1.692e-01  1.596e-02  -10.598  < 2e-16 ***
learnersvm:probpromoter3:ftrSetall        -6.565e-02  1.596e-02   -4.113 3.92e-05 ***
learnertree:probpromoter3:ftrSetall       -1.072e-01  1.596e-02   -6.719 1.90e-11 ***
learnerbayes:probttt1:ftrSetall            3.544e-01  1.596e-02   22.205  < 2e-16 ***
learnerforst:probttt1:ftrSetall           -1.433e-01  1.596e-02   -8.976  < 2e-16 ***
learnerknn10:probttt1:ftrSetall           -9.948e-02  1.596e-02   -6.233 4.69e-10 ***
learnerknn3:probttt1:ftrSetall            -7.753e-02  1.596e-02   -4.858 1.20e-06 ***
learnersvm:probttt1:ftrSetall             -7.984e-02  1.596e-02   -5.003 5.72e-07 ***
learnertree:probttt1:ftrSetall            -6.206e-02  1.596e-02   -3.888 0.000101 ***
learnerbayes:probttt2:ftrSetall            2.981e-01  1.596e-02   18.677  < 2e-16 ***
learnerforst:probttt2:ftrSetall           -1.823e-01  1.596e-02  -11.421  < 2e-16 ***
learnerknn10:probttt2:ftrSetall           -1.373e-01  1.596e-02   -8.604  < 2e-16 ***
learnerknn3:probttt2:ftrSetall            -1.580e-01  1.596e-02   -9.902  < 2e-16 ***
learnersvm:probttt2:ftrSetall             -8.028e-02  1.596e-02   -5.030 4.97e-07 ***
learnertree:probttt2:ftrSetall            -1.139e-01  1.596e-02   -7.137 9.95e-13 ***
learnerbayes:probwine:ftrSetall            3.006e-01  1.596e-02   18.836  < 2e-16 ***
learnerforst:probwine:ftrSetall           -1.855e-01  1.596e-02  -11.624  < 2e-16 ***
learnerknn10:probwine:ftrSetall           -1.403e-01  1.596e-02   -8.791  < 2e-16 ***
learnerknn3:probwine:ftrSetall            -1.830e-01  1.596e-02  -11.463  < 2e-16 ***
learnersvm:probwine:ftrSetall             -7.843e-02  1.596e-02   -4.914 9.03e-07 ***
learnertree:probwine:ftrSetall            -1.340e-01  1.596e-02   -8.397  < 2e-16 ***
learnerbayes:probiris:ftrSetnew            3.159e-01  1.596e-02   19.795  < 2e-16 ***
learnerforst:probiris:ftrSetnew           -1.868e-01  1.596e-02  -11.706  < 2e-16 ***
learnerknn10:probiris:ftrSetnew           -1.707e-01  1.596e-02  -10.692  < 2e-16 ***
learnerknn3:probiris:ftrSetnew            -2.147e-01  1.596e-02  -13.450  < 2e-16 ***
learnersvm:probiris:ftrSetnew             -9.538e-02  1.596e-02   -5.976 2.34e-09 ***
learnertree:probiris:ftrSetnew            -1.331e-01  1.596e-02   -8.339  < 2e-16 ***
learnerbayes:probisoc:ftrSetnew            5.147e-01  1.596e-02   32.248  < 2e-16 ***
learnerforst:probisoc:ftrSetnew           -1.326e-02  1.596e-02   -0.831 0.405984
learnerknn10:probisoc:ftrSetnew           -8.053e-02  1.596e-02   -5.045 4.58e-07 ***
learnerknn3:probisoc:ftrSetnew            -1.508e-01  1.596e-02   -9.447  < 2e-16 ***
learnersvm:probisoc:ftrSetnew              1.204e-01  1.596e-02    7.542 4.88e-14 ***
learnertree:probisoc:ftrSetnew             4.478e-02  1.596e-02    2.806 0.005024 **
learnerbayes:probmonks1:ftrSetnew          5.278e-01  1.596e-02   33.071  < 2e-16 ***
learnerforst:probmonks1:ftrSetnew         -1.573e-01  1.596e-02   -9.858  < 2e-16 ***
learnerknn10:probmonks1:ftrSetnew         -9.165e-02  1.596e-02   -5.742 9.54e-09 ***
learnerknn3:probmonks1:ftrSetnew          -7.976e-02  1.596e-02   -4.997 5.88e-07 ***
learnersvm:probmonks1:ftrSetnew           -1.103e-01  1.596e-02   -6.910 5.03e-12 ***
learnertree:probmonks1:ftrSetnew          -1.248e-01  1.596e-02   -7.821 5.60e-15 ***
```

Figure 54: Linear Model Page 3.

```
learnerbayes:probmonks2:ftrSetnew          3.378e-01  1.596e-02    21.165  < 2e-16 ***
learnerforst:probmonks2:ftrSetnew         -1.290e-01  1.596e-02    -8.080 6.97e-16 ***
learnerknn10:probmonks2:ftrSetnew         -1.653e-01  1.596e-02   -10.357  < 2e-16 ***
learnerknn3:probmonks2:ftrSetnew          -2.812e-01  1.596e-02   -17.618  < 2e-16 ***
learnersvm:probmonks2:ftrSetnew           -1.718e-02  1.596e-02    -1.077 0.281653
learnertree:probmonks2:ftrSetnew          -1.061e-01  1.596e-02    -6.646 3.11e-11 ***
learnerbayes:probmonks3:ftrSetnew          3.054e-01  1.596e-02    19.136  < 2e-16 ***
learnerforst:probmonks3:ftrSetnew         -2.047e-01  1.596e-02   -12.828  < 2e-16 ***
learnerknn10:probmonks3:ftrSetnew         -1.334e-01  1.596e-02    -8.357  < 2e-16 ***
learnerknn3:probmonks3:ftrSetnew          -1.361e-01  1.596e-02    -8.527  < 2e-16 ***
learnersvm:probmonks3:ftrSetnew           -1.036e-01  1.596e-02    -6.493 8.69e-11 ***
learnertree:probmonks3:ftrSetnew          -1.441e-01  1.596e-02    -9.028  < 2e-16 ***
learnerbayes:proboffsides11:ftrSetnew      3.482e-01  1.596e-02    21.818  < 2e-16 ***
learnerforst:proboffsides11:ftrSetnew     -1.193e-01  1.596e-02    -7.472 8.31e-14 ***
learnerknn10:proboffsides11:ftrSetnew     -1.413e-01  1.596e-02    -8.853  < 2e-16 ***
learnerknn3:proboffsides11:ftrSetnew      -2.320e-01  1.596e-02   -14.535  < 2e-16 ***
learnersvm:proboffsides11:ftrSetnew       -4.639e-02  1.596e-02    -2.906 0.003662 **
learnertree:proboffsides11:ftrSetnew      -1.280e-01  1.596e-02    -8.021 1.13e-15 ***
learnerbayes:proboffsides2:ftrSetnew       2.821e-01  1.596e-02    17.673  < 2e-16 ***
learnerforst:proboffsides2:ftrSetnew      -2.622e-01  1.596e-02   -16.426  < 2e-16 ***
learnerknn10:proboffsides2:ftrSetnew      -1.419e-01  1.596e-02    -8.888  < 2e-16 ***
learnerknn3:proboffsides2:ftrSetnew       -1.786e-01  1.596e-02   -11.187  < 2e-16 ***
learnersvm:proboffsides2:ftrSetnew        -4.939e-02  1.596e-02    -3.094 0.001976 **
learnertree:proboffsides2:ftrSetnew       -1.722e-01  1.596e-02   -10.792  < 2e-16 ***
learnerbayes:proboffsides5:ftrSetnew       3.156e-01  1.596e-02    19.772  < 2e-16 ***
learnerforst:proboffsides5:ftrSetnew      -2.674e-01  1.596e-02   -16.756  < 2e-16 ***
learnerknn10:proboffsides5:ftrSetnew      -1.771e-01  1.596e-02   -11.098  < 2e-16 ***
learnerknn3:proboffsides5:ftrSetnew       -2.290e-01  1.596e-02   -14.349  < 2e-16 ***
learnersvm:proboffsides5:ftrSetnew        -1.173e-01  1.596e-02    -7.351 2.07e-13 ***
learnertree:proboffsides5:ftrSetnew       -1.578e-01  1.596e-02    -9.886  < 2e-16 ***
learnerbayes:probpromoter1:ftrSetnew       2.117e-01  1.596e-02    13.265  < 2e-16 ***
learnerforst:probpromoter1:ftrSetnew      -2.563e-01  1.596e-02   -16.059  < 2e-16 ***
learnerknn10:probpromoter1:ftrSetnew      -2.350e-01  1.596e-02   -14.723  < 2e-16 ***
learnerknn3:probpromoter1:ftrSetnew       -2.572e-01  1.596e-02   -16.115  < 2e-16 ***
learnersvm:probpromoter1:ftrSetnew        -1.680e-01  1.596e-02   -10.528  < 2e-16 ***
learnertree:probpromoter1:ftrSetnew       -1.762e-01  1.596e-02   -11.039  < 2e-16 ***
learnerbayes:probpromoter2:ftrSetnew       4.183e-02  1.596e-02     2.621 0.008778 **
learnerforst:probpromoter2:ftrSetnew      -4.638e-01  1.596e-02   -29.060  < 2e-16 ***
learnerknn10:probpromoter2:ftrSetnew      -4.274e-01  1.596e-02   -26.778  < 2e-16 ***
learnerknn3:probpromoter2:ftrSetnew       -4.300e-01  1.596e-02   -26.943  < 2e-16 ***
learnersvm:probpromoter2:ftrSetnew        -3.898e-01  1.596e-02   -24.422  < 2e-16 ***
learnertree:probpromoter2:ftrSetnew       -3.740e-01  1.596e-02   -23.433  < 2e-16 ***
learnerbayes:probpromoter3:ftrSetnew       8.752e-02  1.596e-02     5.484 4.24e-08 ***
learnerforst:probpromoter3:ftrSetnew      -3.698e-01  1.596e-02   -23.172  < 2e-16 ***
learnerknn10:probpromoter3:ftrSetnew      -3.396e-01  1.596e-02   -21.278  < 2e-16 ***
learnerknn3:probpromoter3:ftrSetnew       -3.488e-01  1.596e-02   -21.854  < 2e-16 ***
learnersvm:probpromoter3:ftrSetnew        -2.957e-01  1.596e-02   -18.525  < 2e-16 ***
learnertree:probpromoter3:ftrSetnew       -2.781e-01  1.596e-02   -17.424  < 2e-16 ***
learnerbayes:probttt1:ftrSetnew            4.135e-01  1.596e-02    25.909  < 2e-16 ***
learnerforst:probttt1:ftrSetnew           -2.094e-01  1.596e-02   -13.120  < 2e-16 ***
learnerknn10:probttt1:ftrSetnew           -2.073e-01  1.596e-02   -12.987  < 2e-16 ***
learnerknn3:probttt1:ftrSetnew            -1.710e-01  1.596e-02   -10.713  < 2e-16 ***
learnersvm:probttt1:ftrSetnew             -2.085e-01  1.596e-02   -13.061  < 2e-16 ***
learnertree:probttt1:ftrSetnew            -1.237e-01  1.596e-02    -7.747 9.97e-15 ***
learnerbayes:probttt2:ftrSetnew            2.950e-01  1.596e-02    18.486  < 2e-16 ***
learnerforst:probttt2:ftrSetnew           -3.675e-01  1.596e-02   -23.024  < 2e-16 ***
learnerknn10:probttt2:ftrSetnew           -3.799e-01  1.596e-02   -23.805  < 2e-16 ***
learnerknn3:probttt2:ftrSetnew            -3.592e-01  1.596e-02   -22.507  < 2e-16 ***
learnersvm:probttt2:ftrSetnew             -3.556e-01  1.596e-02   -22.281  < 2e-16 ***
learnertree:probttt2:ftrSetnew            -2.768e-01  1.596e-02   -17.344  < 2e-16 ***
learnerbayes:probwine:ftrSetnew            2.246e-01  1.596e-02    14.072  < 2e-16 ***
learnerforst:probwine:ftrSetnew           -2.418e-01  1.596e-02   -15.152  < 2e-16 ***
learnerknn10:probwine:ftrSetnew           -1.965e-01  1.596e-02   -12.309  < 2e-16 ***
learnerknn3:probwine:ftrSetnew            -2.500e-01  1.596e-02   -15.665  < 2e-16 ***
learnersvm:probwine:ftrSetnew             -1.405e-01  1.596e-02    -8.805  < 2e-16 ***
learnertree:probwine:ftrSetnew            -1.652e-01  1.596e-02   -10.352  < 2e-16 ***
learnerbayes:probiris:ftrSetnewPun         3.404e-01  1.596e-02    21.328  < 2e-16 ***
learnerforst:probiris:ftrSetnewPun        -1.613e-01  1.596e-02   -10.109  < 2e-16 ***
learnerknn10:probiris:ftrSetnewPun        -1.446e-01  1.596e-02    -9.059  < 2e-16 ***
learnerknn3:probiris:ftrSetnewPun         -1.895e-01  1.596e-02   -11.874  < 2e-16 ***
learnersvm:probiris:ftrSetnewPun          -6.950e-02  1.596e-02    -4.354 1.34e-05 ***
learnertree:probiris:ftrSetnewPun         -1.083e-01  1.596e-02    -6.788 1.18e-11 ***
learnerbayes:probisoc:ftrSetnewPun         5.146e-01  1.596e-02    32.244  < 2e-16 ***
learnerforst:probisoc:ftrSetnewPun        -1.399e-02  1.596e-02    -0.876 0.380811
learnerknn10:probisoc:ftrSetnewPun        -8.103e-02  1.596e-02    -5.077 3.88e-07 ***
learnerknn3:probisoc:ftrSetnewPun         -1.515e-01  1.596e-02    -9.491  < 2e-16 ***
learnersvm:probisoc:ftrSetnewPun           1.199e-01  1.596e-02     7.515 6.00e-14 ***
learnertree:probisoc:ftrSetnewPun          4.449e-02  1.596e-02     2.787 0.005323 **
learnerbayes:probmonks1:ftrSetnewPun       5.422e-01  1.596e-02    33.972  < 2e-16 ***
learnerforst:probmonks1:ftrSetnewPun      -1.232e-01  1.596e-02    -7.720 1.24e-14 ***
learnerknn10:probmonks1:ftrSetnewPun      -5.828e-02  1.596e-02    -3.652 0.000261 ***
learnerknn3:probmonks1:ftrSetnewPun       -5.185e-02  1.596e-02    -3.248 0.001163 **
learnersvm:probmonks1:ftrSetnewPun        -7.005e-02  1.596e-02    -4.389 1.15e-05 ***
learnertree:probmonks1:ftrSetnewPun       -8.980e-02  1.596e-02    -5.626 1.87e-08 ***
learnerbayes:probmonks2:ftrSetnewPun       3.373e-01  1.596e-02    21.136  < 2e-16 ***
learnerforst:probmonks2:ftrSetnewPun       6.021e-02  1.596e-02     3.772 0.000162 ***
learnerknn10:probmonks2:ftrSetnewPun       3.486e-02  1.596e-02     2.184 0.028973 *
```

```
learnerknn3:probmonks2:ftrSetnewPun       -4.789e-02  1.596e-02    -3.001 0.002699 **
learnersvm:probmonks2:ftrSetnewPun         1.488e-01  1.596e-02     9.324  < 2e-16 ***
learnertree:probmonks2:ftrSetnewPun        1.573e-01  1.596e-02     9.857  < 2e-16 ***
learnerbayes:probmonks3:ftrSetnewPun       3.143e-01  1.596e-02    19.689  < 2e-16 ***
learnerforst:probmonks3:ftrSetnewPun      -1.939e-01  1.596e-02   -12.147  < 2e-16 ***
learnerknn10:probmonks3:ftrSetnewPun      -1.179e-01  1.596e-02    -7.388 1.57e-13 ***
learnerknn3:probmonks3:ftrSetnewPun       -1.049e-01  1.596e-02    -6.573 5.09e-11 ***
learnersvm:probmonks3:ftrSetnewPun        -8.481e-02  1.596e-02    -5.314 1.09e-07 ***
learnertree:probmonks3:ftrSetnewPun       -1.355e-01  1.596e-02    -8.491  < 2e-16 ***
learnerbayes:proboffsides11:ftrSetnewPun  3.308e-01  1.596e-02    20.724  < 2e-16 ***
learnerforst:proboffsides11:ftrSetnewPun -1.484e-01  1.596e-02    -9.297  < 2e-16 ***
learnerknn10:proboffsides11:ftrSetnewPun -1.576e-01  1.596e-02    -9.874  < 2e-16 ***
learnerknn3:proboffsides11:ftrSetnewPun  -2.394e-01  1.596e-02   -15.000  < 2e-16 ***
learnersvm:proboffsides11:ftrSetnewPun   -6.475e-02  1.596e-02    -4.057 5.00e-05 ***
learnertree:proboffsides11:ftrSetnewPun  -1.085e-01  1.596e-02    -6.801 1.08e-11 ***
learnerbayes:proboffsides2:ftrSetnewPun   3.166e-01  1.596e-02    19.834  < 2e-16 ***
learnerforst:proboffsides2:ftrSetnewPun  -1.746e-01  1.596e-02   -10.939  < 2e-16 ***
learnerknn10:proboffsides2:ftrSetnewPun  -1.096e-01  1.596e-02    -6.867 6.81e-12 ***
learnerknn3:proboffsides2:ftrSetnewPun   -1.445e-01  1.596e-02    -9.056  < 2e-16 ***
learnersvm:proboffsides2:ftrSetnewPun    -2.585e-03  1.596e-02    -0.162 0.871322
learnertree:proboffsides2:ftrSetnewPun   -1.038e-01  1.596e-02    -6.504 8.04e-11 ***
learnerbayes:proboffsides5:ftrSetnewPun   3.486e-01  1.596e-02    21.841  < 2e-16 ***
learnerforst:proboffsides5:ftrSetnewPun  -1.796e-01  1.596e-02   -11.252  < 2e-16 ***
learnerknn10:proboffsides5:ftrSetnewPun  -1.471e-01  1.596e-02    -9.219  < 2e-16 ***
learnerknn3:proboffsides5:ftrSetnewPun   -1.931e-01  1.596e-02   -12.097  < 2e-16 ***
learnersvm:proboffsides5:ftrSetnewPun    -9.375e-02  1.596e-02    -5.874 4.34e-09 ***
learnertree:proboffsides5:ftrSetnewPun   -1.189e-01  1.596e-02    -7.451 9.80e-14 ***
learnerbayes:probpromoter1:ftrSetnewPun   2.522e-01  1.596e-02    15.799  < 2e-16 ***
learnerforst:probpromoter1:ftrSetnewPun  -1.778e-01  1.596e-02   -11.140  < 2e-16 ***
learnerknn10:probpromoter1:ftrSetnewPun  -1.459e-01  1.596e-02    -9.141  < 2e-16 ***
learnerknn3:probpromoter1:ftrSetnewPun   -1.874e-01  1.596e-02   -11.742  < 2e-16 ***
learnersvm:probpromoter1:ftrSetnewPun    -6.862e-02  1.596e-02    -4.300 1.72e-05 ***
learnertree:probpromoter1:ftrSetnewPun   -1.353e-01  1.596e-02    -8.477  < 2e-16 ***
learnerbayes:probpromoter2:ftrSetnewPun   3.174e-01  1.596e-02    19.884  < 2e-16 ***
learnerforst:probpromoter2:ftrSetnewPun  -1.693e-01  1.596e-02   -10.609  < 2e-16 ***
learnerknn10:probpromoter2:ftrSetnewPun  -1.486e-01  1.596e-02    -9.309  < 2e-16 ***
learnerknn3:probpromoter2:ftrSetnewPun   -1.854e-01  1.596e-02   -11.617  < 2e-16 ***
learnersvm:probpromoter2:ftrSetnewPun    -7.044e-02  1.596e-02    -4.413 1.02e-05 ***
learnertree:probpromoter2:ftrSetnewPun   -1.111e-01  1.596e-02    -6.962 3.50e-12 ***
learnerbayes:probpromoter3:ftrSetnewPun   2.570e-01  1.596e-02    16.104  < 2e-16 ***
learnerforst:probpromoter3:ftrSetnewPun  -1.747e-01  1.596e-02   -10.946  < 2e-16 ***
learnerknn10:probpromoter3:ftrSetnewPun  -1.474e-01  1.596e-02    -9.238  < 2e-16 ***
learnerknn3:probpromoter3:ftrSetnewPun   -1.841e-01  1.596e-02   -11.533  < 2e-16 ***
learnersvm:probpromoter3:ftrSetnewPun    -6.875e-02  1.596e-02    -4.307 1.66e-05 ***
learnertree:probpromoter3:ftrSetnewPun   -1.078e-01  1.596e-02    -6.754 1.49e-11 ***
learnerbayes:probttt1:ftrSetnewPun        3.946e-01  1.596e-02    24.720  < 2e-16 ***
learnerforst:probttt1:ftrSetnewPun       -1.696e-01  1.596e-02   -10.623  < 2e-16 ***
learnerknn10:probttt1:ftrSetnewPun       -2.205e-01  1.596e-02   -13.814  < 2e-16 ***
learnerknn3:probttt1:ftrSetnewPun        -1.703e-01  1.596e-02   -10.668  < 2e-16 ***
learnersvm:probttt1:ftrSetnewPun         -1.994e-01  1.596e-02   -12.492  < 2e-16 ***
learnertree:probttt1:ftrSetnewPun        -9.842e-02  1.596e-02    -6.167 7.15e-10 ***
learnerbayes:probttt2:ftrSetnewPun        3.000e-01  1.596e-02    18.795  < 2e-16 ***
learnerforst:probttt2:ftrSetnewPun       -1.870e-01  1.596e-02   -11.716  < 2e-16 ***
learnerknn10:probttt2:ftrSetnewPun       -2.094e-01  1.596e-02   -13.118  < 2e-16 ***
learnerknn3:probttt2:ftrSetnewPun        -2.197e-01  1.596e-02   -13.764  < 2e-16 ***
learnersvm:probttt2:ftrSetnewPun         -1.283e-01  1.596e-02    -8.039 9.69e-16 ***
learnertree:probttt2:ftrSetnewPun        -1.305e-01  1.596e-02    -8.175 3.19e-16 ***
learnerbayes:probwine:ftrSetnewPun        2.999e-01  1.596e-02    18.790  < 2e-16 ***
learnerforst:probwine:ftrSetnewPun       -1.873e-01  1.596e-02   -11.735  < 2e-16 ***
learnerknn10:probwine:ftrSetnewPun       -1.417e-01  1.596e-02    -8.876  < 2e-16 ***
learnerknn3:probwine:ftrSetnewPun        -1.865e-01  1.596e-02   -11.683  < 2e-16 ***
learnersvm:probwine:ftrSetnewPun         -7.389e-02  1.596e-02    -4.630 3.69e-06 ***
learnertree:probwine:ftrSetnewPun        -1.335e-01  1.596e-02    -8.364  < 2e-16 ***
learnerbayes:probiris:ftrSetusedO        -2.415e-02  1.596e-02    -1.513 0.130301
learnerforst:probiris:ftrSetusedO        -2.917e-02  1.596e-02    -1.828 0.067616 .
learnerknn10:probiris:ftrSetusedO        -2.280e-02  1.596e-02    -1.429 0.153164
learnerknn3:probiris:ftrSetusedO         -3.097e-02  1.596e-02    -1.941 0.052319 .
learnersvm:probiris:ftrSetusedO          -2.835e-02  1.596e-02    -1.776 0.075702 .
learnertree:probiris:ftrSetusedO         -2.542e-02  1.596e-02    -1.593 0.111264
learnerbayes:probisoc:ftrSetusedO         1.987e-02  1.596e-02     1.245 0.213242
learnerforst:probisoc:ftrSetusedO         1.319e-01  1.596e-02     8.263  < 2e-16 ***
learnerknn10:probisoc:ftrSetusedO         4.175e-02  1.596e-02     2.616 0.008915 **
learnerknn3:probisoc:ftrSetusedO          2.582e-02  1.596e-02     1.618 0.105723
learnersvm:probisoc:ftrSetusedO          -2.873e-02  1.596e-02    -1.800 0.071918 .
learnertree:probisoc:ftrSetusedO          1.881e-02  1.596e-02     1.178 0.238730
learnerbayes:probmonks1:ftrSetusedO      -1.467e-02  1.596e-02    -0.919 0.357983
learnerforst:probmonks1:ftrSetusedO       1.277e-02  1.596e-02     0.800 0.423535
learnerknn10:probmonks1:ftrSetusedO       6.333e-02  1.596e-02     3.968 7.29e-05 ***
learnerknn3:probmonks1:ftrSetusedO        1.082e-01  1.596e-02     6.780 1.24e-11 ***
learnersvm:probmonks1:ftrSetusedO        -4.020e-02  1.596e-02    -2.519 0.011790 *
learnertree:probmonks1:ftrSetusedO        9.732e-03  1.596e-02     0.610 0.542022
learnerbayes:probmonks2:ftrSetusedO       9.218e-03  1.596e-02     0.578 0.563564
learnerforst:probmonks2:ftrSetusedO      -3.626e-02  1.596e-02    -2.272 0.023120 *
learnerknn10:probmonks2:ftrSetusedO      -1.086e-01  1.596e-02    -6.802 1.07e-11 ***
learnerknn3:probmonks2:ftrSetusedO       -1.749e-01  1.596e-02   -10.955  < 2e-16 ***
learnersvm:probmonks2:ftrSetusedO        -3.444e-02  1.596e-02    -2.158 0.030969 *
learnertree:probmonks2:ftrSetusedO       -9.325e-02  1.596e-02    -5.842 5.26e-09 ***
```

Figure 55: Linear Model Page 4.

```
learnerbayes:probmonks3:ftrSetusedO      -8.478e-03  1.596e-02    -0.531 0.595314
learnerforst:probmonks3:ftrSetusedO      -3.381e-02  1.596e-02    -2.119 0.034147 *
learnerknn10:probmonks3:ftrSetusedO       2.243e-02  1.596e-02     1.405 0.159955
learnerknn3:probmonks3:ftrSetusedO        7.742e-02  1.596e-02     4.851 1.24e-06 ***
learnersvm:probmonks3:ftrSetusedO        -3.278e-02  1.596e-02    -2.054 0.040038 *
learnertree:probmonks3:ftrSetusedO       -9.540e-03  1.596e-02    -0.598 0.550048
learnerbayes:proboffsides11:ftrSetusedO   2.330e-02  1.596e-02     1.460 0.144348
learnerforst:proboffsides11:ftrSetusedO   4.742e-02  1.596e-02     2.971 0.002971 **
learnerknn10:proboffsides11:ftrSetusedO   4.047e-03  1.596e-02     0.254 0.799821
learnerknn3:proboffsides11:ftrSetusedO   -4.975e-02  1.596e-02    -3.117 0.001832 **
learnersvm:proboffsides11:ftrSetusedO     1.027e-02  1.596e-02     0.644 0.519746
learnertree:proboffsides11:ftrSetusedO    2.383e-04  1.596e-02     0.015 0.988086
learnerbayes:proboffsides2:ftrSetusedO   -3.870e-02  1.596e-02    -2.425 0.015333 *
learnerforst:proboffsides2:ftrSetusedO   -7.954e-02  1.596e-02    -4.984 6.30e-07 ***
learnerknn10:proboffsides2:ftrSetusedO    9.807e-04  1.596e-02     0.061 0.951007
learnerknn3:proboffsides2:ftrSetusedO     5.455e-03  1.596e-02     0.342 0.732538
learnersvm:proboffsides2:ftrSetusedO      1.561e-02  1.596e-02     0.978 0.328135
learnertree:proboffsides2:ftrSetusedO    -2.489e-02  1.596e-02    -1.560 0.118833
learnerbayes:proboffsides5:ftrSetusedO   -6.330e-04  1.596e-02    -0.040 0.968365
learnerforst:proboffsides5:ftrSetusedO   -7.211e-02  1.596e-02    -4.518 6.29e-06 ***
learnerknn10:proboffsides5:ftrSetusedO   -2.692e-02  1.596e-02    -1.687 0.091699 .
learnerknn3:proboffsides5:ftrSetusedO    -3.185e-02  1.596e-02    -1.995 0.046034 *
learnersvm:proboffsides5:ftrSetusedO     -5.576e-02  1.596e-02    -3.494 0.000478 ***
learnertree:proboffsides5:ftrSetusedO    -1.173e-02  1.596e-02    -0.735 0.462454
learnerbayes:probpromoter1:ftrSetusedO   -4.573e-02  1.596e-02    -2.865 0.004174 **
learnerforst:probpromoter1:ftrSetusedO   -8.470e-02  1.596e-02    -5.307 1.13e-07 ***
learnerknn10:probpromoter1:ftrSetusedO   -8.402e-02  1.596e-02    -5.264 1.43e-07 ***
learnerknn3:probpromoter1:ftrSetusedO    -6.801e-02  1.596e-02    -4.261 2.05e-05 ***
learnersvm:probpromoter1:ftrSetusedO     -9.716e-02  1.596e-02    -6.087 1.18e-09 ***
learnertree:probpromoter1:ftrSetusedO    -4.483e-02  1.596e-02    -2.809 0.004975 **
learnerbayes:probpromoter2:ftrSetusedO   -2.776e-01  1.596e-02   -17.391  < 2e-16 ***
learnerforst:probpromoter2:ftrSetusedO   -2.995e-01  1.596e-02   -18.764  < 2e-16 ***
learnerknn10:probpromoter2:ftrSetusedO   -2.838e-01  1.596e-02   -17.782  < 2e-16 ***
learnerknn3:probpromoter2:ftrSetusedO    -2.490e-01  1.596e-02   -15.603  < 2e-16 ***
learnersvm:probpromoter2:ftrSetusedO     -3.204e-01  1.596e-02   -20.074  < 2e-16 ***
learnertree:probpromoter2:ftrSetusedO    -2.585e-01  1.596e-02   -16.195  < 2e-16 ***
learnerbayes:probpromoter3:ftrSetusedO   -1.794e-01  1.596e-02   -11.243  < 2e-16 ***
learnerforst:probpromoter3:ftrSetusedO   -2.023e-01  1.596e-02   -12.676  < 2e-16 ***
learnerknn10:probpromoter3:ftrSetusedO   -1.940e-01  1.596e-02   -12.155  < 2e-16 ***
learnerknn3:probpromoter3:ftrSetusedO    -1.687e-01  1.596e-02   -10.569  < 2e-16 ***
learnersvm:probpromoter3:ftrSetusedO     -2.211e-01  1.596e-02   -13.855  < 2e-16 ***
learnertree:probpromoter3:ftrSetusedO    -1.634e-01  1.596e-02   -10.239  < 2e-16 ***
learnerbayes:probttt1:ftrSetusedO         5.991e-03  1.596e-02     0.375 0.707397
learnerforst:probttt1:ftrSetusedO        -6.242e-02  1.596e-02    -3.911 9.23e-05 ***
learnerknn10:probttt1:ftrSetusedO        -7.126e-02  1.596e-02    -4.464 8.09e-06 ***
learnerknn3:probttt1:ftrSetusedO         -1.858e-02  1.596e-02    -1.164 0.244311
learnersvm:probttt1:ftrSetusedO          -1.251e-01  1.596e-02    -7.837 4.90e-15 ***
learnertree:probttt1:ftrSetusedO         -4.322e-02  1.596e-02    -2.708 0.006784 **
learnerbayes:probttt2:ftrSetusedO        -4.444e-03  1.596e-02    -0.278 0.780698
learnerforst:probttt2:ftrSetusedO        -1.899e-01  1.596e-02   -11.895  < 2e-16 ***
learnerknn10:probttt2:ftrSetusedO        -2.227e-01  1.596e-02   -13.952  < 2e-16 ***
learnerknn3:probttt2:ftrSetusedO         -1.729e-01  1.596e-02   -10.831  < 2e-16 ***
learnersvm:probttt2:ftrSetusedO          -2.745e-01  1.596e-02   -17.201  < 2e-16 ***
learnertree:probttt2:ftrSetusedO         -1.361e-01  1.596e-02    -8.525  < 2e-16 ***
learnerbayes:probwine:ftrSetusedO        -7.769e-02  1.596e-02    -4.868 1.14e-06 ***
learnerforst:probwine:ftrSetusedO        -5.465e-02  1.596e-02    -3.424 0.000618 ***
learnerknn10:probwine:ftrSetusedO        -3.434e-02  1.596e-02    -2.151 0.031460 *
learnerknn3:probwine:ftrSetusedO         -5.578e-02  1.596e-02    -3.495 0.000476 ***
learnersvm:probwine:ftrSetusedO          -4.582e-02  1.596e-02    -2.871 0.004098 **
learnertree:probwine:ftrSetusedO         -1.826e-02  1.596e-02    -1.144 0.252640
genjoin:hardLimit:maxCross               -6.072e-07  7.185e-06    -0.085 0.932650
genrandom:hardLimit:maxCross             -3.027e-05  7.746e-06    -3.908 9.36e-05 ***
gensep:hardLimit:maxCross                -2.009e-05  1.473e-05    -1.364 0.172543
gentop:hardLimit:maxCross                        NA          NA        NA       NA
---
Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1

Residual standard error: 0.03091 on 15067 degrees of freedom
Multiple R-Squared: 0.9717,     Adjusted R-squared: 0.9704
F-statistic: 758.2 on 682 and 15067 DF,  p-value: < 2.2e-16
```

Figure 56: Linear Model Page 5.

A.3 COMPARISON OF DIFFERENT GENERATORS AND FEATURE SETS OVER INDIVIDUAL LEARNERS

A.3.1 Additional Results for 10-Fold C.V. Accuracy

					3NN					10NN					NB		
					Base	0.80				Base	0.82				Base	0.78	
Successor Function	Maximum Cross	Hard Limit		All	New	ConRIF	CCF		All	New	ConRIF	CCF		All	New	ConRIF	CCF
Exhaustive	150	150		0.87	0.83	0.86	0.79		0.89	0.84	0.88	0.81		0.78	0.78	0.78	0.79
Join	5	10		0.83	0.72	0.82	0.71		0.85	0.73	0.84	0.72		0.80	0.72	0.80	0.71
Join	5	20		0.83	0.74	0.83	0.73		0.86	0.75	0.85	0.74		0.79	0.74	0.79	0.73
Join	5	40		0.83	0.75	0.83	0.73		0.86	0.76	0.85	0.74		0.78	0.74	0.78	0.74
Join	10	20		0.84	0.75	0.83	0.73		0.87	0.76	0.85	0.75		0.80	0.74	0.80	0.73
Join	10	40		0.85	0.79	0.84	0.76		0.88	0.80	0.85	0.78		0.79	0.76	0.79	0.76
Join	20	40		0.86	0.80	0.85	0.78		0.88	0.82	0.87	0.79		0.79	0.78	0.80	0.77
Join	20	60		0.86	0.82	0.86	0.78		0.89	0.83	0.87	0.80		0.79	0.79	0.79	0.78
Join	20	80		0.86	0.82	0.85	0.78		0.88	0.83	0.86	0.80		0.78	0.78	0.79	0.78
Random	5	5		0.82	0.70	0.82	0.69		0.84	0.72	0.84	0.70		0.79	0.71	0.79	0.70
Random	10	10		0.84	0.76	0.84	0.75		0.86	0.77	0.85	0.76		0.80	0.75	0.80	0.73
Random	20	20		0.85	0.79	0.85	0.77		0.87	0.80	0.86	0.78		0.80	0.76	0.80	0.75
Random	40	40		0.86	0.80	0.85	0.78		0.88	0.83	0.87	0.79		0.79	0.78	0.79	0.78
Sep	3	6		0.83	0.72	0.81	0.71		0.85	0.73	0.83	0.72		0.80	0.74	0.81	0.72
Sep	3	10		0.84	0.75	0.84	0.73		0.86	0.76	0.85	0.74		0.80	0.74	0.80	0.72
Sep	3	15		0.84	0.75	0.83	0.74		0.86	0.76	0.84	0.75		0.79	0.74	0.79	0.73
Sep	3	20		0.84	0.76	0.83	0.74		0.87	0.77	0.85	0.75		0.80	0.75	0.80	0.73
Sep	3	30		0.84	0.76	0.83	0.75		0.87	0.78	0.85	0.76		0.79	0.75	0.79	0.74
Sep	5	10		0.84	0.75	0.83	0.73		0.86	0.76	0.84	0.75		0.80	0.75	0.80	0.74
Sep	5	15		0.84	0.74	0.84	0.72		0.86	0.75	0.85	0.74		0.79	0.74	0.79	0.73
Sep	5	20		0.84	0.76	0.84	0.74		0.86	0.77	0.86	0.75		0.80	0.75	0.80	0.73
Sep	5	30		0.85	0.78	0.85	0.75		0.87	0.79	0.87	0.77		0.80	0.76	0.80	0.75
Sep	10	15		0.85	0.77	0.84	0.74		0.87	0.78	0.85	0.75		0.79	0.75	0.80	0.73
Sep	10	20		0.85	0.78	0.84	0.75		0.87	0.79	0.85	0.75		0.79	0.75	0.79	0.73
Sep	10	30		0.85	0.79	0.85	0.76		0.87	0.80	0.86	0.77		0.78	0.76	0.79	0.75
Sep	10	40		0.85	0.78	0.85	0.77		0.87	0.80	0.86	0.78		0.79	0.75	0.79	0.76
Top	5	5		0.83	0.72	0.81	0.72		0.85	0.74	0.83	0.73		0.80	0.74	0.80	0.73
Top	10	10		0.84	0.75	0.84	0.74		0.86	0.76	0.85	0.75		0.79	0.75	0.79	0.74
Top	20	20		0.85	0.78	0.84	0.75		0.87	0.79	0.85	0.77		0.78	0.75	0.79	0.74
Top	40	40		0.85	0.79	0.84	0.77		0.87	0.81	0.86	0.78		0.79	0.77	0.79	0.76

Table 42: 10CVA Averaged over Run and CLP by Successor Function and Feature Set for 3NN, 10NN, and NB.

				TREE				SVM				FORST		
				Base	0.79			Base	0.85			Base	0.83	
Successor Function	Maximum Cross	Hard Limit	All	New	ConRIF	CCF	All	New	ConRIF	CCF	All	New	ConRIF	CCF
Exhaustive	150	150	0.87	0.84	0.87	0.78	0.90	0.86	0.89	0.83	0.89	0.85	0.88	0.82
Join	5	10	0.84	0.73	0.84	0.70	0.88	0.75	0.87	0.73	0.86	0.74	0.86	0.73
Join	5	20	0.84	0.75	0.84	0.72	0.88	0.76	0.87	0.75	0.87	0.75	0.86	0.75
Join	5	40	0.85	0.75	0.85	0.72	0.89	0.77	0.88	0.75	0.87	0.76	0.86	0.75
Join	10	20	0.85	0.76	0.85	0.73	0.89	0.78	0.87	0.76	0.87	0.77	0.87	0.76
Join	10	40	0.86	0.80	0.86	0.76	0.90	0.81	0.87	0.79	0.88	0.81	0.87	0.79
Join	20	40	0.86	0.82	0.86	0.77	0.90	0.83	0.88	0.80	0.89	0.83	0.88	0.80
Join	20	60	0.87	0.82	0.87	0.78	0.91	0.84	0.89	0.82	0.89	0.83	0.88	0.81
Join	20	80	0.87	0.83	0.87	0.77	0.90	0.85	0.89	0.81	0.89	0.84	0.88	0.81
Random	5	5	0.84	0.71	0.84	0.69	0.88	0.73	0.87	0.71	0.86	0.72	0.86	0.71
Random	10	10	0.86	0.78	0.85	0.74	0.89	0.79	0.88	0.76	0.87	0.78	0.87	0.77
Random	20	20	0.86	0.80	0.85	0.76	0.90	0.82	0.88	0.79	0.88	0.81	0.87	0.78
Random	40	40	0.87	0.82	0.87	0.78	0.90	0.84	0.89	0.81	0.88	0.83	0.88	0.81
Sep	3	6	0.84	0.73	0.84	0.71	0.88	0.75	0.86	0.73	0.86	0.73	0.85	0.73
Sep	3	10	0.85	0.75	0.85	0.72	0.88	0.77	0.87	0.75	0.87	0.77	0.87	0.75
Sep	3	15	0.85	0.76	0.85	0.73	0.89	0.78	0.87	0.75	0.87	0.77	0.86	0.75
Sep	3	20	0.85	0.77	0.84	0.74	0.89	0.79	0.87	0.76	0.87	0.78	0.87	0.76
Sep	3	30	0.86	0.78	0.85	0.75	0.90	0.79	0.88	0.76	0.87	0.79	0.87	0.77
Sep	5	10	0.85	0.76	0.84	0.73	0.88	0.78	0.87	0.76	0.87	0.77	0.86	0.75
Sep	5	15	0.85	0.75	0.85	0.73	0.89	0.77	0.88	0.75	0.86	0.76	0.86	0.75
Sep	5	20	0.85	0.76	0.85	0.73	0.89	0.78	0.88	0.76	0.87	0.77	0.87	0.76
Sep	5	30	0.86	0.79	0.86	0.74	0.90	0.80	0.89	0.78	0.87	0.79	0.88	0.78
Sep	10	15	0.85	0.77	0.85	0.74	0.89	0.79	0.88	0.76	0.88	0.79	0.88	0.76
Sep	10	20	0.85	0.78	0.85	0.74	0.89	0.80	0.87	0.77	0.88	0.80	0.88	0.76
Sep	10	30	0.86	0.79	0.85	0.75	0.89	0.81	0.88	0.78	0.87	0.81	0.87	0.78
Sep	10	40	0.86	0.79	0.86	0.75	0.89	0.81	0.88	0.79	0.88	0.80	0.87	0.78
Top	5	5	0.84	0.74	0.84	0.72	0.88	0.76	0.86	0.74	0.86	0.75	0.85	0.74
Top	10	10	0.85	0.76	0.84	0.73	0.88	0.78	0.88	0.76	0.86	0.77	0.86	0.76
Top	20	20	0.86	0.78	0.85	0.75	0.89	0.80	0.88	0.78	0.87	0.80	0.87	0.77
Top	40	40	0.86	0.80	0.86	0.76	0.90	0.82	0.88	0.79	0.88	0.81	0.88	0.79

Table 43: 10CVA Averaged over Run and CLP by Successor Function and Feature Set for TREE, SVM, and FORST.

A.3.2 Additional Results for AUC

Table 44: AUC Averaged over Run and CLP by Successor and Feature Set for 3NN, 10NN, and NB.

				3NN				10NN				NB		
Successor Function	Maximum Cross	Hard Limit	All	New	ConRIF	CCF	All	New	ConRIF	CCF	All	New	ConRIF	CCF
Exhaustive	150	150	0.91	0.86	0.90	0.83	0.92	0.87	0.91	0.85	0.79	0.79	0.79	0.80
Join	5	10	0.89	0.74	0.87	0.73	0.90	0.75	0.88	0.74	0.83	0.74	0.83	0.71
Join	5	20	0.89	0.76	0.88	0.75	0.90	0.77	0.89	0.76	0.82	0.75	0.82	0.73
Join	5	40	0.89	0.77	0.88	0.75	0.91	0.78	0.89	0.77	0.79	0.74	0.79	0.74
Join	10	20	0.89	0.78	0.88	0.77	0.91	0.79	0.89	0.78	0.82	0.76	0.82	0.74
Join	10	40	0.90	0.81	0.88	0.80	0.91	0.82	0.89	0.81	0.80	0.77	0.80	0.76
Join	20	40	0.90	0.83	0.89	0.81	0.92	0.84	0.90	0.83	0.81	0.78	0.81	0.77
Join	20	60	0.91	0.85	0.90	0.82	0.92	0.86	0.91	0.84	0.80	0.79	0.80	0.79
Join	20	80	0.90	0.85	0.89	0.82	0.92	0.86	0.90	0.84	0.79	0.78	0.79	0.79
Random	5	5	0.88	0.73	0.87	0.72	0.90	0.74	0.89	0.73	0.83	0.72	0.83	0.70
Random	10	10	0.89	0.79	0.88	0.78	0.91	0.80	0.89	0.79	0.84	0.77	0.84	0.73
Random	20	20	0.90	0.82	0.89	0.80	0.91	0.83	0.90	0.81	0.83	0.78	0.83	0.75
Random	40	40	0.90	0.84	0.89	0.82	0.92	0.85	0.90	0.84	0.81	0.79	0.81	0.78
Sep	3	6	0.88	0.74	0.86	0.73	0.90	0.76	0.87	0.75	0.84	0.75	0.84	0.72
Sep	3	10	0.89	0.78	0.88	0.76	0.91	0.79	0.90	0.77	0.84	0.77	0.84	0.73
Sep	3	15	0.89	0.78	0.87	0.77	0.91	0.79	0.89	0.78	0.82	0.75	0.82	0.74
Sep	3	20	0.89	0.79	0.88	0.77	0.91	0.80	0.89	0.79	0.82	0.76	0.82	0.74
Sep	3	30	0.90	0.80	0.88	0.78	0.91	0.81	0.89	0.80	0.80	0.75	0.80	0.75
Sep	5	10	0.89	0.77	0.87	0.76	0.91	0.78	0.89	0.78	0.82	0.76	0.82	0.74
Sep	5	15	0.89	0.78	0.89	0.75	0.91	0.79	0.90	0.77	0.81	0.75	0.81	0.73
Sep	5	20	0.89	0.79	0.89	0.77	0.90	0.80	0.90	0.78	0.81	0.76	0.81	0.74
Sep	5	30	0.90	0.81	0.90	0.78	0.92	0.82	0.91	0.80	0.81	0.76	0.81	0.75
Sep	10	15	0.89	0.80	0.88	0.78	0.91	0.81	0.89	0.79	0.81	0.76	0.81	0.74
Sep	10	20	0.90	0.80	0.88	0.78	0.91	0.81	0.89	0.79	0.81	0.75	0.81	0.74
Sep	10	30	0.90	0.81	0.89	0.79	0.91	0.82	0.90	0.80	0.79	0.76	0.79	0.76
Sep	10	40	0.90	0.82	0.89	0.80	0.91	0.83	0.90	0.81	0.80	0.76	0.80	0.76
Top	5	5	0.88	0.75	0.85	0.74	0.90	0.76	0.87	0.76	0.81	0.74	0.81	0.73
Top	10	10	0.89	0.78	0.88	0.77	0.90	0.79	0.89	0.79	0.80	0.75	0.80	0.75
Top	20	20	0.89	0.80	0.88	0.78	0.91	0.81	0.90	0.80	0.80	0.76	0.80	0.75
Top	40	40	0.90	0.82	0.88	0.81	0.91	0.84	0.90	0.82	0.79	0.76	0.79	0.77

Table 45: AUC Averaged over Run and CLP by Successor and Feature Set for TREE, SVM, and FORST.

			TREE				SVM				FORST			
Successor Function	Maximum Cross	Hard Limit	All	New	ConRIF	CCF	All	New	ConRIF	CCF	All	New	ConRIF	CCF
Exhaustive	150	150	0.88	0.86	0.88	0.80	0.90	0.85	0.89	0.79	0.93	0.88	0.92	0.87
Join	5	10	0.86	0.74	0.85	0.72	0.87	0.73	0.86	0.69	0.92	0.76	0.91	0.75
Join	5	20	0.86	0.76	0.86	0.74	0.88	0.75	0.87	0.71	0.92	0.78	0.92	0.77
Join	5	40	0.87	0.77	0.87	0.74	0.89	0.76	0.88	0.72	0.93	0.78	0.92	0.78
Join	10	20	0.86	0.78	0.86	0.75	0.88	0.77	0.86	0.73	0.93	0.80	0.92	0.79
Join	10	40	0.87	0.82	0.87	0.78	0.89	0.80	0.87	0.75	0.93	0.83	0.92	0.83
Join	20	40	0.88	0.84	0.88	0.79	0.90	0.82	0.88	0.77	0.93	0.85	0.92	0.84
Join	20	60	0.88	0.84	0.88	0.80	0.90	0.84	0.89	0.78	0.93	0.86	0.93	0.85
Join	20	80	0.88	0.84	0.88	0.79	0.90	0.84	0.88	0.78	0.93	0.86	0.92	0.85
Random	5	5	0.86	0.73	0.86	0.70	0.88	0.72	0.87	0.68	0.92	0.75	0.92	0.74
Random	10	10	0.87	0.79	0.87	0.76	0.89	0.78	0.87	0.73	0.93	0.81	0.92	0.80
Random	20	20	0.87	0.81	0.87	0.77	0.90	0.81	0.88	0.75	0.93	0.83	0.92	0.83
Random	40	40	0.88	0.84	0.88	0.79	0.90	0.83	0.88	0.78	0.93	0.86	0.92	0.85
Sep	3	6	0.86	0.75	0.85	0.73	0.87	0.74	0.85	0.70	0.92	0.76	0.91	0.76
Sep	3	10	0.86	0.78	0.86	0.75	0.87	0.77	0.86	0.72	0.92	0.80	0.92	0.79
Sep	3	15	0.86	0.79	0.86	0.75	0.88	0.77	0.86	0.72	0.92	0.80	0.92	0.79
Sep	3	20	0.86	0.79	0.86	0.76	0.88	0.78	0.87	0.73	0.93	0.81	0.92	0.80
Sep	3	30	0.87	0.80	0.87	0.77	0.89	0.79	0.87	0.73	0.93	0.81	0.92	0.81
Sep	5	10	0.86	0.78	0.86	0.75	0.88	0.76	0.86	0.72	0.92	0.80	0.92	0.79
Sep	5	15	0.87	0.78	0.87	0.75	0.88	0.76	0.87	0.72	0.92	0.79	0.92	0.79
Sep	5	20	0.86	0.79	0.86	0.76	0.88	0.77	0.87	0.73	0.92	0.80	0.92	0.80
Sep	5	30	0.88	0.81	0.87	0.77	0.89	0.80	0.88	0.74	0.93	0.82	0.93	0.81
Sep	10	15	0.86	0.80	0.86	0.76	0.88	0.78	0.87	0.73	0.92	0.82	0.92	0.80
Sep	10	20	0.87	0.81	0.86	0.76	0.88	0.79	0.87	0.73	0.93	0.82	0.92	0.80
Sep	10	30	0.87	0.81	0.87	0.77	0.89	0.80	0.87	0.75	0.92	0.83	0.92	0.82
Sep	10	40	0.88	0.81	0.87	0.77	0.89	0.80	0.87	0.75	0.93	0.83	0.92	0.82
Top	5	5	0.86	0.75	0.85	0.74	0.87	0.74	0.85	0.71	0.92	0.77	0.91	0.77
Top	10	10	0.86	0.78	0.86	0.76	0.88	0.77	0.87	0.73	0.92	0.80	0.92	0.80
Top	20	20	0.87	0.81	0.87	0.77	0.88	0.79	0.87	0.74	0.92	0.83	0.92	0.81
Top	40	40	0.88	0.83	0.88	0.78	0.89	0.81	0.88	0.76	0.93	0.84	0.92	0.83

Table 46: Obscured Benefits of Feature Construction. Certain feature sets may be better on a per problem or per learner basis. Averages of behavior can obscure the optimal use of feature set for a given problem.

	New	All	ConRIF	Best
Problem 1	.5	.75	1.0	1.0
Problem 2	1.0	.5	.75	1.0
Problem 3	.75	1.0	.5	1.0
Average	.75	.75	.75	1.0

A.3.3 Benefits Obscured by Feature Set Choice

One additional method of using the constructed features is to consider the best of conditions New, All, and New+Unused. Evaluating the best of the three is useful because none of the feature sets are dominant over any other[1]. An example of the situation is shown in Table 46. Hence, one of the conditions may be good for certain CLPs and bad for others. The variability obscures the benefits of feature construction when averaged over several CLPs. The Best method is also justified because the time invested into generating the new features is not duplicated for the different methods when they are used in learning. So, with a large investment of time in the feature construction process and a comparatively small amount of time to use the feature construction results, there is little cost (specifically, three times the learning time) to finding the best method of using the constructed features for a particular CLP.

A table of Best results is shown in Table 47. Comparing Table 47 with the tables in Section A.3.1 we see that there are some gains to be made trying each feature set and using the best of those feature sets for task performance. Further analysis is required to determine whether better combinations of Base and New features could mimic Best or if CLP characteristics could lead to the choice of a dominant feature set.

[1]Even New, a generally poor performer, has some wins on a per run basis.

Table 47: Best Feature Set Usage Results. The entries under each learning method show the accuracy of the optimal feature set after averaging each feature set over CLP and run over the given learner and successor function.

Successor	Hard Limit	Maximum Cross	3NN	10NN	NB	TREE	SVM	FORST
Base Rate			0.80	0.82	0.78	0.79	0.85	0.83
Exh	150	150	0.89	0.90	0.84	0.88	0.91	0.90
Join	5	10	0.85	0.86	0.83	0.85	0.89	0.87
Join	10	20	0.86	0.88	0.83	0.86	0.90	0.89
Join	5	20	0.85	0.87	0.83	0.86	0.89	0.88
Join	10	40	0.87	0.89	0.84	0.87	0.90	0.89
Join	20	40	0.88	0.89	0.84	0.88	0.91	0.90
Join	5	40	0.85	0.88	0.82	0.86	0.91	0.88
Join	20	60	0.88	0.90	0.84	0.88	0.91	0.90
Join	20	80	0.88	0.90	0.84	0.88	0.91	0.90
Random	10	10	0.86	0.88	0.83	0.86	0.90	0.89
Random	20	20	0.87	0.89	0.84	0.87	0.91	0.89
Random	40	40	0.87	0.89	0.84	0.88	0.91	0.90
Random	5	5	0.83	0.86	0.82	0.85	0.89	0.88
Sep	3	10	0.85	0.87	0.83	0.86	0.89	0.88
Sep	5	10	0.85	0.87	0.83	0.85	0.89	0.88
Sep	10	15	0.86	0.88	0.83	0.86	0.90	0.89
Sep	3	15	0.85	0.87	0.83	0.86	0.89	0.88
Sep	5	15	0.85	0.87	0.83	0.86	0.90	0.88
Sep	10	20	0.86	0.88	0.83	0.86	0.90	0.89
Sep	3	20	0.85	0.88	0.83	0.86	0.90	0.88
Sep	5	20	0.86	0.88	0.83	0.86	0.90	0.88
Sep	10	30	0.87	0.89	0.83	0.86	0.90	0.89
Sep	3	30	0.86	0.88	0.83	0.87	0.90	0.89
Sep	5	30	0.87	0.89	0.84	0.87	0.91	0.89
Sep	10	40	0.88	0.89	0.84	0.87	0.90	0.89
Sep	3	6	0.84	0.86	0.83	0.85	0.88	0.87
Top	10	10	0.86	0.88	0.83	0.86	0.90	0.88
Top	20	20	0.86	0.88	0.83	0.87	0.90	0.89
Top	40	40	0.87	0.89	0.84	0.87	0.91	0.89
Top	5	5	0.84	0.86	0.82	0.85	0.88	0.87

APPENDIX B

SYNTHETIC DATA GENERATION

B.1 DESCRIPTION OF THE WEATHER PROBLEM FOR SYNTHETIC GENERATOR

```

precipValues = ["Rainy", "Cloudy", "PartCl", "Sunny"]
tempValues = ["Cold", "Cool", "Mod", "Warm", "Hot"]
windValues = ["Windy", "Breezy", "Calm"]
humidityValues = ["Arid", "Mod", "Humid"]


def makeHI(t, h):
    adj = humidityValues.index(h) - 1
    newt = tempValues.index(t) + adj
    return newt

def makeWC(t, w):
    adj = windValues.index(w) - 2
    newt = tempValues.index(t) + adj
    return newt

def twoMean(a,b):
    return (a+b)/2.0


def makeTennis(p, c):
    c = int(c)+2
    if p == "Rainy":
        return False

    prob = uniform(0,1)
    if p == "Cloudy" and c in [3,4] and prob > .3:
        return True
    elif p == "PartCl" and c in [3,4] and prob > .2:
        return True
    elif p == "Sunny" and c in [2,3,4] and prob > .1:
        return True

    if prob > .8:
        return True

    return False

basevars = [("Precip", ’choice(precipValues)’),
            ("Temp", ’choice(tempValues)’),
            ("Wind", ’choice(windValues)’),
            ("Humidity", ’choice(humidityValues)’)]



derivedvars = [("HI", makeHI, "Temp", "Humidity"),
               ("WC", makeWC, "Temp", "Wind"),
               ("Comfort", twoMean, "HI", "WC"),
               ("Tennis", makeTennis, "Precip", "Comfort")]


model = {"dist":basevars,
         "func":derivedvars,
         "cons":[],
         "prop":1.0}
models = [model]

shown = ["Precip", "Temp", "Wind", "Humidity", "Tennis"]
targetFeature = "Tennis"

numTrainExamples = 100
numTestExamples = 0

dataFormat = FormatDict(defaultDataFormat="%s")
```

Figure 57: Weather Dataset Description for Synthetic Generation.

B.2 DESCRIPTION OF THE AREA PROBLEM FOR SYNTHETIC GENERATOR

```

def newside(s1, s2, cosv):
    # c^2 = a^2 + b^2 - 2 ab cos(theta)
    return sqrt(s1**2 + s2**2 - \
                     2*s1*s2*cosv)

def herron(a, b, c):
    s = (a+b+c)/2.0
    area = sqrt(s*(s-a)*(s-b)*(s-c))
    return area

def circA(radius):
    area = pi * pow(radius, 2)
    return area

def msum(*args):
    return sum(args)

def largest(tri, rect, circ):
    tmp = [(tri, "tri"),
           (rect, "rect"),
           (circ, "circ")]

    return max(tmp)[1]

basevars = [(’tri_A’, ’uniform(1,10.2)’),
            (’tri_B’, ’uniform(1,10.2)’),
            (’tri_cos’, ’uniform(-1,1)’),

            (’rect_A’, ’uniform(1,6)’),
            (’rect_B’, ’uniform(1,6)’),

            (’circ_R’, ’uniform(1,2.8)’)]

derivedvars = [(’tri_C’,     newside, ’tri_A’, ’tri_B’, ’tri_cos’),
               (’tri_area’,  herron,  ’tri_A’, ’tri_B’, ’tri_C’),
               (’rect_area’, mul,     ’rect_A’, ’rect_B’),
               (’circ_area’, circA,   ’circ_R’),
               (’largest?’,  largest, ’tri_area’, ’rect_area’, ’circ_area’)]

model = {"dist":basevars,
         "func":derivedvars,
         "cons":[],
         "prop":1.0}

models = [model]

shown = [’tri_A’, ’tri_B’, ’tri_C’,
         ’rect_A’, ’rect_B’,
         ’circ_R’]
# shown = [’tri_area’, ’rect_area’, ’circ_area’]

targetFeature = ’largest?’

numTrainExamples = 480
numTestExamples = 0
```

Figure 58: Area Dataset Description for Synthetic Generation.

BIBLIOGRAPHY

Aronis, J. M. and Provost, F. J. (1994). Efficiently constructing relational features from background knowledge for inductive machine learning. In *Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop*, pages 347–358, Seattle, WA. AAAI Press, Menlo Park, CA.

Aronis, J. M. and Provost, F. J. (1997). Increasing the efficiency of data mining algorithms with breadth-first marker propagation. In *Knowledge Discovery and Data Mining*, pages 119–122, Newport Beach, CA. AAAI Press, Menlo Park, CA.

Ben-Bassat, M. (1982). Use of distance measures, information measures and error bounds in feature evaluation. In Krishnaiah, P. R. and Kanal, L. N., editors, *Handbook of Statistics*, volume 2, chapter 35, pages 773–791. North–Holland Publishing Company, Amsterdam.

Bloedorn, E. and Michalski, R. S. (1998a). Data-driven constructive induction. *IEEE Intelligent Systems*, 13(2):30–7.

Bloedorn, E. and Michalski, R. S. (1998b). Data-driven constructive induction: Methodology and applications. In *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 4, pages 51–68. Kluwer Academic Publishers, Norwell, MA.

Blum, A. and Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1-2):245–271.

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and regression trees*. Wadsworth, Pacific Grove, CA.

Buchanan, B. (1982). Mechanizing the search for explanatory hypotheses. In Asquith, P. and Nickles, T., editors, *PSA 1982*, volume 2, pages 129–146. Philosophy of Science Association, East Lansing, Michigan.

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167.

Callan, J. P. (1989). Knowledge-based feature generation. In Spatz, B., editor, *Proceedings of the Sixth International Workshop on Machine Learning*, Cornell Univeristy, Ithaca, NY. Morgan Kaufmann Publishers, San Francisco, CA.

Callan, J. P. (1990). Use of domain knowledge in constructive induction. Technical Report 90-95, COINS, Department of Computer Science, University of Massachusetts, Amherst, MA.

Carbonell, J. G., Michalski, R. S., and Mitchell, T. M. (1983). An overview of machine learning. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, Los Altos, CA.

Chang, C. and Lin, C. (2006). *LIBSVM: A Library for Support Vector Machines*. Department of Computer Science and Information Engineering, National Taiwai University.

Cohen, W. W. (1990). An analysis of representation shift in concept learning. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 104–112, Austin, Texas. Morgan Kaufmann Publishers, San Francisco, CA.

Darden, L. (1991). *Theory change in science*. Oxford University Press, New York.

Demsar, J., Zupan, B., and Leban, G. (2004). Orange: From experimental machine learning to interactive data mining. http://www.ailab.si/orange. White Paper, Faculty of Computer and Information Sciences, University of Llubjana.

Dietterich, T. G. and Michalski, R. S. (1983). A comparative review of selected methods for learning from examples. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann, Los Altos, CA.

Dobson, A. J. (2002). *An Introduction to Generalized Linear Models*. Texts in Statistical Science. Chapman and Hall/CRC, Boca Raton FL, 2nd edition.

Donoho, S. and Rendell, L. (1998). Feature construction using fragmentary knowledge. In Liu, H. and Motoda, H., editors, *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 17, pages 272–288. Kluwer Academic Publishers, Norwell, MA.

Donoho, S. K. (1996). *Knowledge-Guided Constructive Induction*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Drastal, G. and Raatz, S. (1989). Empirical results on learning in an abstraction space. In *Proceedings of the 1989 Internation Joint Conference on Artificial Intelligence*, pages 708–712, Detroit, MI. Morgan Kaufmann Publishers, San Francisco, CA.

Dzeroski, S. and Todorovski, L. (1995). Discovering dynamics: From inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4(1):89–108.

Fayyad, U. M. and Irani, K. B. (1993). Multi-interval discretization of continuous-valued attributes for classi cation learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1029, Syndney, Australia. Morgan Kaufmann, San Francisco, CA.

Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., editors (1996). *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, Cambridge, MA.

Fenner, M. E. (2002). Feature selection metrics and the feature construction task. Master's thesis, Department of Computer Science, University of Pittsburgh.

Gillies, D. (1996). *Artificial Intelligence and Scientific Method.* Oxford University Press, New York.

Hand, D. J., Blunt, G., Kelly, M. G., and Adams, N. M. (2000). Data mining for fun and profit. *Statistical Sciences*, 15(2):111–131.

Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The elements of statistical learning: data mining, inference, and prediction*. Springer series in statistics. Springer-Verlag, New York.

Hochberg, Y. and Tamhane, A. C. (1987). *Multiple Comparisons Procedures*. John Wiley and Sons, New York, NY.

Klemke, E. D. (1980). Part i. science and nonscience: Introduction. In Klemke, E. D., Hollinger, R., and Kline, A. D., editors, *Introductory Readings in the Philosophy of Science*. Prometheus Books, Buffalo, New York.

Kokar, M. M. (1986). Discoverying functional formulas through changing represenation base. In *Proceedings of AAAI-86: The 5th National Conference on Artificial Intelligence*, pages 455–459, Philadelpha, PA. AAAI Press, Menlo Park, CA.

Kramer, S. (1994). Cn2-mci: A two-step method for constructive induction. In *Proceedings of the ML-COLT-94 Workshop on Constructive Induction and Change of Representation*, New Brunswick, NJ.

Kramer, S. (1995). Predicate invention: A comprehensive view. Technical Report OFAI-TR-95-32, Austrian research Institute for Artificial Intelligence.

Kuhn, T. S. (1996). *The Structure of Scientific Revolutions*. University Of Chicago Press, Chicago, IL.

Laird, J. E., Newell, A., and Rosenbloom, P. S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64.

Langley, P., Simon, H. A., Bradshaw, C. L., and Zytkow, J. M. (1987). *Scientific Discovery: Computational Explorations of the Creative Processes*. MIT Press, Cambridge, MA.

Lavrac, N. and Dzeroski, S. (1994). *Inductive logic programming: Techniques and applications*. Ellis Horwood, New York.

Lavrac, N. and Flach, P. A. (2001). An extended transformation approach to inductive logic programming. *ACM Trans. Comput. Logic*, 2(4):458–494.

Levesque, H. and Brachman, R. (1985). A fundamental tradeoff in knowledge representation and reasoning. In Levesque, H. and Brachman, R., editors, *Readings in Knowledge Representation*, pages 41–70. Morgan Kaufmann Publishers, San Francisco, CA.

Liu, H. and Motoda, H. (1998a). *Feature Extraction, Construction and Selection: A Data Mining Approach*. Kluwer Academic Publishers, Norwell, MA.

Liu, H. and Motoda, H. (1998b). Less is more. In Liu, H. and Motoda, H., editors, *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 1, pages 3–12. Kluwer Academic Publishers, Norwell, MA.

Markovitch, S. and Rosenstein, D. (2002). Feature generation using general constructor functions. *Machine Learning*, 49:59–98.

Matheus, C. J. (1989). *Feature Construction: An Analytic Framework and an Application to Decision Tree*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Matheus, C. J. and Rendell, L. A. (1989). Constructive induction on decision trees. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 645–650, Detroit, MI. Morgan Kaufmann, San Francisco, CA.

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

Minsky, M. and Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

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

Mitchell, T. M. (1990). The need for biases in learning generalizations. In Shavlik, J. W. and Dietterich, T. G., editors, *Readings in Machine Learning*. Morgan Kaufmann, San Mateo, CA.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, New York, NY.

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

Muggleton, S. H. (1989). DUCE: An oracle-based approach to constructive induction. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 287–292, San Mateo, CA. Morgan Kaufmann Publishers, San Francisco, CA.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA.

Newell, A. and Simon, H. (1963). Gps, a program that simulates human thought. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Thought*. McGraw-Hill, New York.

Newell, A. and Simon, H. (1972). *Human Problem Solving*. Prentice-Hall, New Jersey.

Newman, D., Hettich, S., Blake, C., and Merz, C. (1998). UCI repository of machine learning databases. http://www.ics.uci.edu/$\sim$mlearn/MLRepository.html.

Niles, I. and Pease, A. (2001). Towards a standard upper ontology. In *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems*, Ogunquit, Maine. ACM, New York, NY.

Pagallo, G. and Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, 5:71–99.

Pazzani, M. J. (1998). Constructive induction of cartesian product attributes. In *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 21, pages 341–354. Kluwer Academic Publishers, Norwell, MA.

Popper, K. (1965). *The Logic of Scientific Discovery*. Harper Torchbooks, New York.

Provost, F., Aronis, J., and Buchanan, B. (1999). Rule-space search for knowledge-based discovery. CIIO Working Paper IS 99-012, Stern School of Business, New York University, New York, NY.

Provost, F. J. (1992). *Policies for the Selection of Bias in Inductive Machine Learning*. PhD thesis, Department of Computer Science, University of Pittsburgh.

Provost, F. J. and Buchanan, B. G. (1995). Inductive policy: The pragmatics of bias selection. *Machine Learning*, 20(1-2):35–61.

PSF (2006). Python programming language – official website. http://www.python.org.

Pudil, P. and Novovicova, J. (1998). Novel methods for feature subset selection with respect to problem knowledge. In *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 7, pages 101–116. Kluwer Academic Publishers, Norwell, MA.

Quinlan, J. R. (1995). *C4.5; Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA.

Rendell, L. and Seshu, R. (1990). Learning hard concepts through constructive induction: framework and rationale. *Computational Intelligence*, 6(247-270).

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs.

Samuel, A. L. (1963). Some studies in machine learning using the game of checkers. In Feigenbaum, E. A. and Feldman, J., editors, *Computers and Though*. McGraw-Hill, New York.

Setiono, R. and Liu, H. (1998). Fragmentation problem and automated feature construction. In *Proceedings of the 10th International Conference on Tools with Artificial Intelligence*, pages 208–215, Taipei, Taiwan. IEEE.

Sutton, R. S. and Matheus, C. J. (1991). Learning polynomial functions by feature construction. In *Eighth International Workshop on Machine Learning*, pages 208–212, Chicago, Illinois. Morgan Kaufmann, San Francisco, CA.

Todorovski, L. and Džeroski, S. (1997). Declarative bias in equation discovery. In *Proceedings of the 14th International Conference on Machine Learning*, pages 376–384, Nashville, TN. Morgan Kaufmann, San Francisco, CA.

Utgoff, P. and Mitchell, T. M. (1982). Acquisition of appropriate bias for inductive concept learning. In *Proceedings of AAAI-82*, pages 414–417, Pittsburgh, PA. Morgan Kaufmann Publishers, San Francisco, CA.

Wnek, J. and Michalski, R. S. (1994). Hypothesis-driven constructive induction in aq17-hci: A method and experiments. *Machine Learning*, 14(1):139–168.

Zheng, Z. (1998). A comparison of constructing different types of new feature for decision tree learning. In Liu, H. and Motoda, H., editors, *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 15, pages 239–255. Kluwer Academic Publishers, Norwell, MA.

Zupan, B., Bohanec, M., Demsar, J., and Bratko, I. (1998). Feature transformation by function decomposition. In *Feature Extraction, Construction and Selection: A Data Mining Perspective*, chapter 20, pages 328–340. Kluwer Academic Publishers, Norwell, MA.

Zytkow, J. M. and Simon, H. A. (1986). A theory of historical discovery: The construction of componential models. *Machine Learning*, 1(1):107–136.