


complete. In the general architecture of FICUS described in Figure 4 we allow using any concept learner to be the consumer of the generated feature. In this paper we only try using C4.5 as the external concept learner. We plan, however, to test the utility of the features generated by FICUS with other concept learning algorithms such as Nearest-neighbor or Back-propagation. We would also like to explore the effect of adding pruning to the C4.5 internal concept learner. It is quite possible that pruning would remove subtrees with irrelevant constructed features thus improving the quality of the set of constructed features passed to the next iteration.

It has been shown that irrelevant features degrade the performance of concept learners. We have found during our experiments that irrelevant constructor functions may analogously degrade the performance of FICUS. An interesting research direction could be to exploit the fact that FICUS receives its representation language as dynamic input for conducting a search in the representation space. Methods similar to those employed for feature selection (John et al., 1994; Kohavi & Dan, 1995; Salzberg, 1993) could be adopted to find a useful subset of constructor functions.

Many researchers have shown the benefit of feature generation for solving classification problems. Many alternative feature-generation algorithms that are based on a variety of representation schemes have been proposed. The methodology presented here offers a unifying framework for feature generation and has several advantages over existing algorithms. While other algorithms are tailored to a predefined feature representation that may not be inappropriate for the problem at hand, the FICUS algorithm allows a flexible representation which can be tailored to the given problem. The most important advantage of our framework is that it allows expressing and exploiting partial background knowledge for constructing useful features. Such knowledge is commonly available for real-world problems but can not be exploiting by other feature generation algorithms due to the rigidity of their representation.

References


To conclude, there are numerous algorithms and representation schemes for feature construction, each having its strengths and weaknesses, and each is appropriate for different types of problems. The work presented in this paper offers a unifying framework that was designed to support a general and flexible representational form. The specification language offered by our framework is strong enough to allow the definition of many of the representational forms employed by the previously described algorithms, such as recursive boolean features, M-of-N concepts, and simple hyperplanes. In addition it enables to define any other logical, mathematical, or domain-specific function that can be formulated by the user based on domain background knowledge.

The specification of the representation language is supplied in an input FSS file according to a defined grammar. We presented the FICUS construction algorithm, which receives the standard input of supervised learning, as well as an input FSS and uses them to produce a set of generated features. FICUS searches its defined feature space, continuously attempting to improve its generated feature set as long as resources are available. The algorithm bases its operation on the framework of decision tree learning that defines its context of feature construction. The algorithm uses general construction operators whose actual action is determined by the input FSS. It also uses general feature evaluation functions that can be uniformly applied to different forms of constructed features. Both data-driven and hypothesis-driven strategies are employed by the algorithm to guide its conducted search.

It is interesting to view the FICUS algorithm as an evolutionary process. The population is the set of constructed features. New members of the population are constructed by combining existing feature pairs of high fitness. The fitness is assigned by the building block evaluation functions that look at properties such as information gain, complexity and relative gain with respect to the parents. The combination is performed by applying a set of predefined combination operators (Compose, Insert, Replace and Interval). The population is kept at a fixed size by removing members with low fitness. The particular tree-based representation used by FICUS resembles the tree-structured elements manipulated by genetic programming algorithms (Koza, 1992; Koza, Bennett, Andre, & Kean, 1996).

The merit of our approach was demonstrated by applying the algorithm to various classification problems. FICUS was able to significantly improve the accuracy of the resulting classifiers, as well as reduce their complexity. In addition, it generated features that often expressed important aspects of the underlying target concept. FICUS's general and flexible form of feature representation turns it into an effective tool for searching the space of representations for an appropriate representation with respect to a given classification problem.

One limitation of the FICUS framework is that its search algorithm depends on the assumption that building blocks of complex features of high utility will also be found to be utile. While this was the case in most of the problems that FICUS was tried on, it is quite possible that in certain domains this assumption does not hold and a gradual generation of structured features will not be effective. This problem may be addressed by increasing the beam size of the search algorithm. Such an increase, however, is quite restricted due to its high computational demands.

While the research reported here is sufficient to demonstrate the power of FICUS it is far from being
axis parallel, as in C4.5 (Quinlan, 1989), or multivariate as in LMDT (Utgoff & Brodley, 1991), SADT (Heath et al., 1993) and CART (Breiman et al., 1984). Such multivariate hyperplanes are induced by methods of linear regression and weights adjustment. An extension to the hyperplane representation was performed in the NDT (Ittner & Schlosser, 1996) algorithm which generates non-linear splits in the form of curved hypersurfaces.

As in the previously described algorithms, LMDT (Utgoff & Brodley, 1991) performs feature construction through the course of building a decision-tree classifier. At each created tree node, the algorithm constructs a hyperplane feature by training a thermal linear machine. The construction procedure is aimed at generating concise hyperplanes that are based on relevant data attributes. When LMDT detects that a linear machine is near its final set of boundaries, it eliminates the variable which least contributes to discriminating the current set of instances and then continues training the linear machine. Finally the most accurate linear machine with the minimum number of variables is chosen.

The SADT (Heath et al., 1993) algorithm uses the same framework as LMDT, but employs a random construction technique that is based on simulated annealing. The idea underlying this method is that the locally best split of a tree node might not be the globally optimal one, and thus it may be preferable to generate a set of alternative trees which may produce good approximate solutions. Hyperplane representation may be suitable for problems of an appropriate bias, however it is a fixed representation that can not be adapted to include background knowledge concerning the problem domain, and in addition may suffer from poor comprehensibility. Since the FICUS architecture does not offer a straightforward way of performing coefficient adjustments, it is limited in its capabilities of producing hyperplanes. One way to overcome this difficulty is to introduce a constructor function which receives a set of features and use one of the available methods for producing an appropriate hyperplane.

Although not directly related to FICUS, it is worth mentioning some of the rule-based systems, in the context of supervised learning, that perform feature construction. Algorithms such as STRUCT (Watanabe & Rendell, 1991), AQ17-HCI (Wenk & Michalski, 1994) and PRAX (Bala et al., 1992) employ rules as their feature representation. New rules are created from existing rules by using an operator set to alter them. Rules can be specialized by adding terms to the rule’s conditions, or generalized by deleting terms or substituting them with variables. The rules representation is usually limited to clause form. Michalski’s AQ17-HCI (Wenk & Michalski, 1994) construction algorithm bases its operation on the AQ15 learning system. The algorithm iteratively applies AQ15 to induce a rule set which best covers its positive examples. The induced rule set is analyzed, modified accordingly, and then used for the next iteration of the algorithm.

Regarding the use of grammars as part of concept learning systems, it is worth to mention the work of Todorovski and Dzeroski (1997) in the context of equation discovery. The discovery system LAGRAME attempts to find an equation that describes a given set of measured data. LAGRAME uses a context-free grammar to define and restrict its equation hypothesis space. The grammar enables to make use of mathematical operators as well as functions which represent domain specific knowledge. The discovery system was successfully applied to a number of problems of equation discovery, concerning the behavior of dynamic systems.
accuracies. CITRE employed background knowledge, concerning the Tic-Tac-Toe domain, and measured its utility. This knowledge, however, was not added as part of the feature representation, but rather inserted into the algorithm itself. Another drawback in the employed background knowledge was that it involved a deep understanding of the Tic-Tac-Toe problem rather than limited partial knowledge.

FICUS differs from CITRE in two major aspects. First, it uses a general feature representation, which is not predefined in the algorithm itself, but rather specified by an input fss. Existing background knowledge is also expressed as part of the fss in the form of constraint functions. Second, as opposed to CITRE which constructs new features based on its entire instance set and generated hypothesis, FICUS performs dynamic feature construction at each created tree node, which enables it to separately search distinct parts of the instance space.

Aha’s IB3-CI (1991) algorithm, inspired by CITRE, is another construction algorithm that generates boolean features based on the conjunction operator. IB3-CI integrates the use of instance-based learning, performed by the IB3 (Aha et al., 1991) algorithm, with the STAGGER (J.C., 1987) algorithm which performs incremental feature construction. In the course of its activation, IB3-CI generates conjuncts of existing features which match positive instances and mismatch negative ones. IB3-CI exercised some of the ideas presented in CITRE, such as feature generalization and background knowledge, and performed experiments on the Tic-Tac-Toe endgame domain parallel to those performed by CITRE.

The minimal operator sets used by the previously presented algorithms are sufficient but not always adequate for the induction and representation of complex boolean functions. Different algorithms (Murphy & Pazzani, 1991; Zheng, 1996; Rendell, 1995) have been developed to employ feature representations that are able to express certain complex boolean relations.

The ID2-OF-3 (Murphy & Pazzani, 1991) and X-OF-N (Zheng, 1996) algorithms employ a more versatile form of representation for expressing boolean relations by constructing M-of-N and X-of-N concept features respectively. An M-of-N feature is specified by a set of N features and a number \( M \leq N \). The feature is satisfied for a particular example if at least \( M \) features of the set are true. The motivation for the construction of M-of-N concepts is the belief in their relevance for the acquisition of naturally occurring concepts, particularly in medical domains, where expert systems make use of "criteria tables" that are essentially M-of-N concepts (Murphy & Pazzani, 1991).

The ID2-OF-3 and X-OF-N algorithms perform a greedy search in their constructed feature space, guided by operators that generalize or specialize existing features mainly by addition or removal of a single attribute-value pair. The FICUS algorithm is able to generate M-of-N and X-of-N features, given appropriate construction functions (such as the Count constructor), but is not limited to these specific representational forms.

In spite of their relevance to a variety of classification problems, boolean relations cover only a part of the potential interaction between data attributes. In addition, boolean relations, at list simple ones like AND and OR, are often inherently represented in the decision tree structure.

A different form of feature representation, especially suited for continuous attributes, is hyperplanes. A hyperplane is a linear plane that splits the domain space into two separate subspaces. Hyperplanes can be
space. The search is performed by iteratively combining the feature holding the highest InfoGain value, with an original basic feature which passes a certain filter criterion. The constructed features are potentially used in the generated tree classifier, which is returned as the final output of the algorithm. While FICUS uses a similar method of constructing features local to decision tree nodes, it is different than LFC in two major aspects. First, as opposed to FICUS which is general enough to use any constructor function, LFC is designed to generate only boolean features using the \{\neg, \land\} operator set. Furthermore, LFC guides its feature construction by employing a selection filter that is based on a geometric representation suited only to describe boolean features, while FICUS bases its search on general heuristic measures, that are suited to address features of different forms. The second major difference is that LFC creates one tree and terminates, while FICUS iteratively repeats this process, using a selected subset of its generated features to perform further feature construction. This approach enables FICUS to integrate features that were generated at different parts of instance space during the tree formation. Indeed, Figure 9 shows that the utility of the generated feature set increases with the number of iterations.

The GALA (Hu & Kibler, 1996) construction algorithm resembles the LFC algorithm, thus differing from FICUS in the same major aspects that were previously mentioned. GALA, as oppose to LFC and similarly to FICUS, produces as output a feature set instead of a classifier. It also resembles FICUS in considering the relative gain of features with respect to their composing parents. However, as opposed to GALA, FICUS also takes into account the feature complexity, which plays an even more prominent role in guiding its construction process.

The FRINGE (Pagallo & Haussler, 1990) algorithm and its descendants (Yang, Rendell, & Blix, 1991), perform feature construction by combining sibling leaves of a generated decision tree. FRINGE operates iteratively, where at each iteration the generated features are used to build the tree of the next iteration. As opposed to GALA and LFC where construction is guided by data driven measures such as InfoGain, FRINGE follows an hypothesis driven construction approach where new features are constructed based on the previously generated hypothesis decision tree. Like FRINGE, FICUS iteratively analyzes its produced decision tree to select a subset of dominant features for its next iteration. However, unlike FRINGE, FICUS does not confine its feature construction to tree leaves, but rather generates features at every tree node.

Another algorithm that creates boolean features using the framework of decision-tree concept learning is CITRE, which was presented in an inspiring paper concerning constructive induction (Matheus & Rendell, 1989). The CITRE construction algorithm (Matheus & Rendell, 1989) was presented as part of a framework which did not confine itself to a specific feature representation. The CITRE algorithm itself, however, was designed to employ an operator set containing only one member – \{And\}. CITRE employs an additional meta operator which was used to perform feature generalization. This operator, however, was suited only for nominal typed attributes, and for concept problems of an appropriate bias. Like the FRINGE (Pagallo & Haussler, 1990) algorithm, CITRE also performs hypothesis-driven construction. The CITRE algorithm iteratively builds a decision tree, and performs feature construction based on patterns that appear in the generated tree. Unlike FRINGE, CITRE searches for patterns in the entire tree, and not just in its leaves. The CITRE algorithm was tested mainly on the Tic Tac Toe problem and did not achieve high classification
Another example of domain-specific feature construction is the work by Sutton (1991) who designed an algorithm for learning high-order polynomial functions. The algorithm works by iteratively performing linear regression, combined with feature construction. The algorithm constructs a new feature by forming a product of the two existing features that most effectively predict the square error of its current hypothesis function. Such special-purpose algorithms may be effectively tailored for a given domain, but might be hard to generalize for other domains and problems.

More general construction algorithms use a feature representation that can be employed for different domains and problems. A common representation is that of boolean features, which enable to express existing boolean relations between data attributes. Algorithms like FRINGE (Pagallo & Haussler, 1990), LFC (Ragavan et al., 1993) and GALA (Hu & Kibler, 1996) use the minimal {¬, ∧} operator set to construct recursive boolean features. All three algorithms base their operation on the framework of decision tree learning, which is used to define their context of feature construction. Although using an identical representational language, and relying on the same learning technique, the algorithms use different approaches for feature construction.

The LFC algorithm performs feature construction through the course of building a decision tree classifier. New features are constructed at each created tree node, by performing a branch and bound search in feature
in which the hybrid strategy significantly outperformed the data driven. In addition, the hybrid selection strategy outperformed the data driven when employing a small number ($N_{new} = 1...10$) of evaluated features.

7 Discussion and related work

A variety of algorithms have been developed to improve concept learning by methods of feature construction. These algorithms differ in their form of feature representation, construction techniques and output format.

At one side of this spectrum stand construction algorithms that were designed to be used for a specific problem domain. Such algorithms construct special-purpose features using domain-specific background knowledge. The bootstrapping algorithm (Hirsh & Japkowicz, 1994) was specifically designed for the domain of molecular biology. The algorithm represents features as sequences of nucleotides, whose legal syntax structure is determined by existing background knowledge. The algorithm starts with an initial set of feature sequences, produced by human experts, and uses a special set of operators to alter them into new sequence features.
<table>
<thead>
<tr>
<th>Domain</th>
<th>Prominent Generated Feature(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promoter</td>
<td>$\text{Count}(\text{Is}(p_{34,G}), \text{Is}(p_{36,F}), \text{Is}(p_{35,T}))$</td>
</tr>
<tr>
<td></td>
<td>This prominently produced feature describes the minus_25 contact region which</td>
</tr>
<tr>
<td></td>
<td>has been identified in many recognized promoters.</td>
</tr>
</tbody>
</table>
| Offside-11| $\{\text{Max}(w_{89}, w_{99}), \text{Max}(w_{119}, w_{219}), \text{Max}(w_{59}, w_{69}, w_{119}),$
|          | $\text{Max}(w_{39}, w_{49}, w_{109}), \text{Max}(b_{19}, b_{29}, \text{Max}(b_{109}, b_{209},$
|          | $\text{Max}(b_{89}, \text{Max}(b_{59}, b_{79}, b_{99}), \text{Max}(b_{39}, b_{109}, b_{119})))$   |
|          | This complex feature which appeared in similar versions, throughout the produced                |
|          | classifiers, almost fully describes the entire target concept. The meaning of $w_{69}$ is       |
|          | the Y coordinate of player No. 6 of the white team.                                             |
| Queens-3 | $\{\text{AbsDiff}(Q_{1e}, Q_{2e}), \text{AbsDiff}(Q_{1g}, Q_{2g})\}$,\[AbsDiff(Q_{1e}, Q_{2e}),\] |
|          | $\text{AbsDiff}(Q_{1g}, Q_{2g})\}$                                                             |
|          | The first feature identifies whether Queens 1 and 2 are placed on the same row or column, while |
|          | the second feature identifies if they are placed on any common diagonal. Similar symmetric      |
|          | features identified threats between other queen pairs.                                          |
| Isosceles | $\text{Count} = (A_{1,2}) = (A_{2,3}) = (A_{3,1})$                                            |
|          | This feature completely represents the concept of an isosceles triangle, which                  |
|          | requires at least one pair of equal arcs.                                                     |
| Tic-Tac-Toe| $\{\text{Count}(\text{Is}(s_1,x), \text{Is}(s_2,x), \text{Is}(s_3,x))\}$,\[Count(Is(s_1,x),Is(s_2,x),Is(s_3,x))\] |
|          | $\text{Count}(\text{Is}(s_1,o), \text{Is}(s_2,o), \text{Is}(s_3,o))$                         |
|          | These representing features describe rows, columns, and diagonals of consecutive x or o signs. |
|          | Although FICUS was able to produce features of much higher complexity, it almost exclusively      |
|          | produced rows, columns and diagonals of slot triplets.                                         |
| Monk2    | $\{\text{Count}(\text{Is}(A_1,1), \text{Is}(A_2,1), \text{Is}(A_3,1), \text{Is}(A_4,1), \text{Is}(A_5,1))\}$ |
|          | Fully describes the target concept of the Monk2 problem.                                       |
| Wine     | $\{\text{Attrib}_{11}, [\text{Attrib}_{11}, \text{Attrib}_{10}]\}$                            |
|          | This feature and its mathematical equivalents appeared in the majority of classifiers which      |
|          | achieved 100% accuracy. We do not know the meaning of this feature since the wine domain theory  |
|          | was not available to us.                                                                      |
| Heart    | $\{\text{Count}(\text{In}_{\text{Range}}(cp,-\infty,3.5), \text{In}_{\text{Range}}(ca,-\infty,0.5)$ |
|          | $\text{In}_{\text{Range}}(cp,-\infty,3.5)$ was a component of most prominent features.       |
| Balance  | $\{\{\text{Distance}_{\text{right}}, \text{Distance}_{\text{left}}\}, \{\text{Weight}_{\text{left}}, \text{Weight}_{\text{right}}\}$ |
|          | $\text{Distance}_{\text{right}}, \{\text{Weight}_{\text{left}}, \text{Distance}_{\text{left}}\})\}$ |
|          | These features fully describe the target concept.                                              |

Table 5: A list of prominent features produced by the FICUS algorithm. These example demonstrate the ability of FICUS to discover features that are strongly related to the target concept.

the employed selection strategy. For the Offside11 and Queens3 problems, which represent complex target concepts, the graphs indicate a noticeable improvement in accuracy as a result of increasing the number of evaluated features. The Tic-Tac-Toe problem demonstrated minimal sensitivity to the number of evaluated features, especially when employing a hybrid selection strategy. For the promoter problem, the optimal number of evaluated features was interestingly discovered to be approximately 12, regardless of the selection strategy. This may result from its large number of irrelevant basic features combined with a small data set, which might lead to the construction and selection of superfluous features. The graphs do not indicate a conclusive advantage to either one of the selection strategies, excluding the complex Offside11 problem,
Algorithms whose feature representation is believed to be unsuitable for a problem domain were denoted in the table as N/S (Not Suitable) with respect to the given problem.

<table>
<thead>
<tr>
<th>Problem</th>
<th>C4.5</th>
<th>FICUS + C4.5</th>
<th>GALA + C4.5</th>
<th>LFC</th>
<th>ID3-653 (M-of-N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promoter</td>
<td>76.88 (±8.0)</td>
<td>86.84 (±5.7)</td>
<td>79.5 (±7.8)</td>
<td>75.1 (±7.0)</td>
<td>87.6</td>
</tr>
<tr>
<td>Wine</td>
<td>91.93 (±2.9)</td>
<td>93.29 (±2.9)</td>
<td>93.8 (±3.0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TicTacTo</td>
<td>84.79 (±3.0)</td>
<td>97.58 (±0.9)</td>
<td>-</td>
<td>-</td>
<td>94.9</td>
</tr>
<tr>
<td>Queens2</td>
<td>66.80 (±3.0)</td>
<td>100.00 (±0.0)</td>
<td>N/S</td>
<td>N/S</td>
<td>N/S</td>
</tr>
<tr>
<td>Queens3</td>
<td>66.53 (±3.0)</td>
<td>98.39 (±2.6)</td>
<td>N/S</td>
<td>N/S</td>
<td>N/S</td>
</tr>
<tr>
<td>Offside1</td>
<td>79.22 (±3.5)</td>
<td>98.33 (±1.6)</td>
<td>N/S</td>
<td>N/S</td>
<td>N/S</td>
</tr>
<tr>
<td>Offside2</td>
<td>66.03 (±3.2)</td>
<td>94.67 (±3.0)</td>
<td>N/S</td>
<td>N/S</td>
<td>N/S</td>
</tr>
<tr>
<td>Isosceles</td>
<td>62.45 (±2.4)</td>
<td>100.00 (±0.0)</td>
<td>N/S</td>
<td>N/S</td>
<td>N/S</td>
</tr>
<tr>
<td>Balance</td>
<td>77.78 (±2.5)</td>
<td>99.66 (±0.0)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Heart</td>
<td>73.75 (±3.8)</td>
<td>75.87 (±3.8)</td>
<td>76.4 (±2.5)</td>
<td>75.2 (±2.7)</td>
<td>76.8</td>
</tr>
<tr>
<td>Iris</td>
<td>94.03 (±3.3)</td>
<td>95.05 (±3.4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Monk1</td>
<td>95.36 (±2.0)</td>
<td>100.00 (±0.0)</td>
<td>-</td>
<td>-</td>
<td>10.0</td>
</tr>
<tr>
<td>Monk2</td>
<td>81.77 (±2.6)</td>
<td>100.00 (±0.0)</td>
<td>-</td>
<td>-</td>
<td>98.5</td>
</tr>
<tr>
<td>Monk3</td>
<td>95.29 (±1.8)</td>
<td>95.35 (±1.6)</td>
<td>-</td>
<td>-</td>
<td>97.2</td>
</tr>
</tbody>
</table>

Table 4: A comparison between the performance of FICUS and other feature construction algorithms. Algorithms whose feature representation is believed to be unsuitable for a problem domain were denoted in the table as N/S (Not Suitable) with respect to the given problem.

Table 5 presents some of the prominent features that were generated by FICUS when applied to the tested classification problems. The presented features appear in the majority of the classifiers that were generated for the tested problem, mostly in those with high accuracy compared to the accuracy when using the basic features. As can be seen from the table, for the cases where the target concept is known, the prominent features partially (or fully) express the underlying problem concept. Such features enable the concept learner to increase the accuracy, compactness and comprehensibility of the classifiers produced by the concept learner. For example, in the Tic-Tac-Toe domain, almost all the features that were output by FICUS are those using the count constructor to identify full rows, columns and diagonals of the same color. In the Promoter domain FICUS identified the minus35 contact region which is considered as a good promoter indicator by the existing theory.

We have performed several experiment to test the effect of the independent variables described above on the performance of FICUS. We used 4 problem domains for these experiments: Promoter, Offside11, 3Queens and Tic-Tac-Toe, which represent a variety of different domains and learned concepts. The graphs in Figure 9 shows the mutual effect of the number of iterations (N_{iterations}) and number of generation-phases (N_{phase}) on the accuracy of the produced classifier. As expected, the utility of the generated features increases with the increase in the number of iterations and the number of phases. We found, however, that further increasing the number of generation phases caused an ungradual increase of the feature complexity.

The graphs in Figure 10 show the performance of FICUS as a function of its number of evaluated features at each generation phase (N_{new}). These figures also show the effect of the evaluation strategy of building blocks employed by the generator on the achieved performance. Each figure contains two graphs, corresponding to
Table 2: Problems and their Supplied Constructor Functions

<table>
<thead>
<tr>
<th>Domain</th>
<th>Constructors</th>
<th>Domain</th>
<th>Constructors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Promoter</td>
<td>{Is, Count}</td>
<td>Wine</td>
<td>{(\forall, +)}</td>
</tr>
<tr>
<td>Tic-Tac-Toe</td>
<td>{Is, Count}</td>
<td>Chess-Queens</td>
<td>{(\forall, +, AbsDiff)}</td>
</tr>
<tr>
<td>Soccer-Offside</td>
<td>{(\forall, +, Max, Min)}</td>
<td>Balance</td>
<td>{(\forall, +, AbsDiff)}</td>
</tr>
<tr>
<td>Heart-Disease</td>
<td>InRange, Count, And</td>
<td>Iris</td>
<td>{(\forall, Average)}</td>
</tr>
<tr>
<td>Monk-Problems</td>
<td>Isosceles-Triangle</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: The results obtained for the basic experiment set performed with FICUS. Each number represents the average of 100 experiments (20 \(\times\) 5CV). The numbers in parenthesis are standard deviations. The table presents three sets of results: the basic C4.5 algorithm, C4.5 enhanced by FICUS using the default parameters and C4.5 enhanced by FICUS using the best found settings of parameters. The CPU runtime are for running the algorithm with the default values.

<table>
<thead>
<tr>
<th>Domain</th>
<th>C4.5</th>
<th>C4.5+\text{Ficus}_{\text{defalt}}</th>
<th>C4.5+\text{Ficus}_{\text{best}}</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy %</td>
<td>Size</td>
<td>Weighted size</td>
<td>Accuracy %</td>
</tr>
<tr>
<td>Promoter</td>
<td>76.88 (±8.0)</td>
<td>31.6</td>
<td>84.49 (±5.6) 11.4</td>
<td>16.8</td>
</tr>
<tr>
<td>Wine</td>
<td>91.92 (±2.9)</td>
<td>13.9</td>
<td>94.84 (±2.7) 7.6</td>
<td>12.5</td>
</tr>
<tr>
<td>Tic-Tac-Toe</td>
<td>84.79 (±3.8)</td>
<td>247.0</td>
<td>96.52 (±1.1) 69.4</td>
<td>107.3</td>
</tr>
<tr>
<td>Queens 2</td>
<td>66.80 (±3.6)</td>
<td>237.1</td>
<td>100.00 (±0.0) 5.0</td>
<td>15.0</td>
</tr>
<tr>
<td>Queens 3</td>
<td>65.72 (±3.0)</td>
<td>262.1</td>
<td>96.72 (±3.7) 35.9</td>
<td>75.1</td>
</tr>
<tr>
<td>Offside 5</td>
<td>79.72 (±3.5)</td>
<td>86.7</td>
<td>98.11 (±1.6) 11.8</td>
<td>44.8</td>
</tr>
<tr>
<td>Offside 11</td>
<td>66.63 (±3.2)</td>
<td>99.5</td>
<td>76.07 (±2.9) 64.2</td>
<td>134.5</td>
</tr>
<tr>
<td>Isosceles 5</td>
<td>82.53 (±2.6)</td>
<td>255.1</td>
<td>100.00 (±0.0) 5.0</td>
<td>11.0</td>
</tr>
<tr>
<td>Balance 5</td>
<td>77.73 (±2.5)</td>
<td>222.8</td>
<td>99.86 (±0.2) 5.0</td>
<td>15.0</td>
</tr>
<tr>
<td>Heart 5</td>
<td>73.79 (±3.8)</td>
<td>76.8</td>
<td>75.87 (±2.8) 68.0</td>
<td>76.1</td>
</tr>
<tr>
<td>Iris 5</td>
<td>94.03 (±3.3)</td>
<td>14.6</td>
<td>95.05 (±3.4) 8.1</td>
<td>17.9</td>
</tr>
<tr>
<td>Monk 1</td>
<td>99.36 (±0.3)</td>
<td>116.1</td>
<td>100.00 (±0.0) 3.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Monk 2</td>
<td>81.77 (±2.6)</td>
<td>383.8</td>
<td>99.90 (±0.1) 28.6</td>
<td>53.8</td>
</tr>
<tr>
<td>Monk 3</td>
<td>95.29 (±1.8)</td>
<td>143.6</td>
<td>95.25 (±1.6) 110.3</td>
<td>195.1</td>
</tr>
</tbody>
</table>

Achieved by FICUS, which where usually produced by mildly increasing the value of \(N_{\text{iterations}}\) or \(N_{\text{new}}\). It is clear that FICUS significantly improved the classification accuracy for most domains. FICUS also achieved better values of standard deviation reflecting higher stability. FICUS also dramatically reduced the size of the produced tree classifiers. It also significantly reduced the weighted tree size for all the problems containing only nominal attributes. For problems containing continuous attributes, the reduction was sometimes less significant, and in few cases the weighted tree size increased, regardless of the improvement in classification accuracy.

Table 4 compares the results achieved by FICUS to those reported for other feature construction algorithms for several UCI problems which where found suitable for their employed representations. In spite of its generality, the performance of FICUS was found to be comparable, and in most cases superior to that of special-purpose construction algorithms, with respect to their favorable domains. In addition, FICUS was successfully applied to other complex problems such as Attacking Queens and Soccer offside, for which the mentioned algorithms could not be effectively applied due to their restricted representational power.
two attributes which describe its X and Y coordinates on the field. We experimented with problems of two 5 player and 11 player teams, expressed by 20 and 44 attributes correspondingly.

- **Isosceles-Triangle**: This problem, deals with the classification of triangles as isosceles or nonisosceles. Each example triangle is represented by its 3 continuous arc lengths. Although the target concept is quite simple, concept learning algorithms like c4.5 are unable to produce an effective decision tree for its description.

- **Balance**: This problem, taken from the Irvine Repository, was generated to model psychological experimental results. Each example is classified as having a balance scale tip to the right, tip to the left, or balanced. Each example is represented by 4 continuous attributes.

- **Iris**: This problem, taken from the Irvine Repository, deals with the classification of Iris plants into 3 classes. Each example is represented by 4 continuous attributes.

- **Heart**: This problem, taken from the Irvine Repository, deals with the classification of patients as being diagnosed to suffer a heart attack or not. Each example is represented by 14 attributes, mostly continuous, and few nominal.

For each problem, FICUS was supplied with constructor functions relevant to the target concept, as well as irrelevant functions to test the resilience of the algorithm. Table 1 contains the space of all constructor functions used for the experiments described here\(^3\). It is easy to add more mathematical functions such as $\sqrt{}$ and $a^X$, logic functions such as XOR and domain-specific functions. These constructor functions are able to express some of the discussed representations such as recursive boolean expressions, M-of-N expressions (using the Count function), and simple hyperplanes. Table 2 presents the problem domains together with their associated constructor functions.

<table>
<thead>
<tr>
<th>Standard Mathematic functions</th>
<th>$\pm, -, \div, \times, =, AbsDiff, Average, Max, Min.$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Logic functions</td>
<td>And, Or</td>
</tr>
<tr>
<td>Special Logic functions</td>
<td>Count denoted as $Count(b_1, \ldots, b_n)$ returns the number of its boolean arguments which hold a TRUE value.</td>
</tr>
<tr>
<td>Interval functions</td>
<td>$Is(f,c) = TRUE \Leftrightarrow f \in [c_1, c_2]$</td>
</tr>
<tr>
<td></td>
<td>$InRange(f, c_1, c_2) = TRUE \Leftrightarrow f \in [c_1, c_2]$.</td>
</tr>
</tbody>
</table>

Table 1: The set of all the constructor functions used for the experiments

### 6.2 Experimental Results

Table 3 summarizes the results obtained by FICUS regarding classification accuracy, hypothesis complexity measured by tree size and weighted tree size, and CPU runtime.

The table presents the results achieved by employing a default configuration where: $N_{\text{iterations}} = 2$, $N_{\text{phase}} = 2$, $N_{\text{new}} = 100$ and $\text{Evaluation Strategy} = \text{Hybrid}$. In addition the table presents the best results

\(^3\)The AbsDiff function measures the absolute difference between two numeric values.
Figure 8: The left picture describes the Attacking Queens problem while the right hand side represents the Soccer Offside problem.

- **Wine**: This problem, taken from the Irvine Repository, deals with the classification of wines into 3 class types. Each example is represented by 13 continuous attributes, which represent measures of chemical elements in the wine.

- **Tic-Tac-Toe**: The problem, taken from the Irvine Repository, deals with the classification of legal Tic-Tac-Toe end games, as wins for the x player or not. Each example is represented by 9 nominal attributes, which represent the slot values in the range \{x,o,b\}.

- **Monks problems**: This set of problems, taken from the Irvine Repository, contains the three known monk problems. Each example is represented by 5 nominal attributes in the range \{1,2,3,4\}. The problems are:
  - \((a1 = a2) or (a5 = 1)\)
  - Exactly two of \((a1 = 1, a2 = 1, a3 = 1, a4 = 1, a5 = 1)\).
  - \((a5 = 3 and a4 = 1) or (a5 \neq 4 and a2 \neq 3)\). With added 5% class noise.

- **Attacking Queens**: The target concept of this problem, illustrated in Figure 8, is the existence of a mutual threat between any pair of queens placed on a chess board. In the Queen2 domain, examples are pairs of queens specified by their row and column positions. Similarly, Queen3 deals with triplets of queens.

- **Soccer offside**: This problem, illustrated in Figure 8, deals with the classification of soccer field situations as being offside or not. (An offside situation occurs when a player of one team, is placed in front of all the second team’s players, at the moment of ball passing.). Each player is represented by
6.1.1 Dependent variables

The utility of the decision-tree classifiers, produced by c4.5, was measured by the following dependent variables:

- **Classification accuracy**: the portion of the test set that was correctly classified.
- **Tree size**: One of the motivations for using feature generation is to produce more succinct hypotheses. We measure the complexity of the produced tree by its number of nodes.
- **Weighted tree size**: Since the produced tree classifiers contain generated features which have higher complexity than the basic features, we also compute the weighted tree size that takes into account the feature size. The weighted size of a tree is the sum of feature complexity of its nodes, where the complexity of each generated feature is measured by the size of its representing tree.
- **Comprehensibility**: Another motivation for using generated feature is to make the produced classifier more comprehensible to human by introducing features that are related to the target concept. It is difficult to come with a computational method for measuring comprehensibility but we still believe that it is important to evaluate this aspect of the produced classifiers. Therefore, we show for each domain its prominent features – features that appear in most produced classifiers.

In addition we measure the CPU time consumed by the learning process.

6.1.2 Independent variables

The performance of Ficus is influenced by the following independent variables.

- **\(N_{\text{iterations}}\)**: the number of iterations performed by the main loop of the algorithm.
- **\(N_{\text{phase}}\)**: the number of search phases performed by the feature generator.
- **\(N_{\text{new}}\)**: the number of new features evaluated at each search phase of the generator.
- **Evaluation strategy**: the evaluation strategy used by the feature generator to evaluate its building blocks. We tested the data-driven strategy and the hybrid strategy.

Ficus was tested on various artificial and real-world classification problems. The majority of the problem domains were taken from the Irvine Repository. The *Attacking Queens*, *Soccer Offside*, and *Isosceles Triangle* problems are novel problems, the first two of which were designed to test complex target concepts.

- **Promoter**: This problem, taken from the Irvine Repository, deals with the classification of DNA sequences as promoters or non promoters. Each example is represented by 57 nominal attributes, which represent the values of its sequential nucleotides, in the range \{A,G,T,C\}. 
The above bound should be multiplied by the tree depth which is bounded by $|E|$ and by the number of iterations of the main loop. Therefore, the complexity of ficus is bounded by:

$$N_{\text{iterations}} N_{\text{phase}} N_{\text{new}} |E|^2 \log_2 |E|$$

(3)

The bound $|E|$ on the tree depth is for an extreme case. In practice we have received much shallower trees.

When using the hybrid evaluation strategy we must add to the cost of generation the cost of producing the hypothesis tree:

$$O_G(E_n) \leq |E_n| \log_2 |E_n| N_{\text{new}} N_{\text{phase}} + |E_n|^2 \log_2 |E_n| |F|$$

where $F$ is the set of building block features. The complexity of the ficus algorithm is then bounded by:

$$N_{\text{iterations}} N_{\text{phase}} N_{\text{new}} |E|^2 \log_2 |E| + N_{\text{iterations}} |F||E|^2 \log_2 |E|$$

(4)

Where the second additional item represents the operational complexity of producing the hypothesis decision tree in the first generation phase. In practice we found that using the hybrid strategy increases the computation time by at most factor of 2.

6 Experiments

A variety of experiments were conducted to test the performance and behavior of the ficus algorithm. We start with a description of the methodology used for the experimentation, and continue with the description of the experiment results.

6.1 Experimental Methodology

The performance of ficus was evaluated by the utility of its returned set of generated features. The utility was measured by comparing the performance of classifiers produced by a standard concept learner using the set of generated features to the performance of classifiers produced by the same learner using only the original basic features. In our experiments we chose to use the basic C4.5 concept learner, although other learners such as IBL or Neural Networks could be used as well. We decided to turn pruning off in order to have a better isolation of the effects of the feature generation and to reduce the effects of additional factors that do not have a direct relevance to this research.

Each experiment was conducted by averaging the results of 20 runs of a 5 fold cross validation, which sum up to a total of 100 runs. In each fold, 80% of the example set were used by ficus to generate its feature set and then by C4.5 to produce a decision tree which uses it, while the remaining 20% were utilized as an independent test set for measuring the accuracy of the produced classifier. For problems of more then 500 classified examples, each experiment consisted of 10 runs of 5 cross fold validation, which sum up to 50 runs.
the constructed features of its current feature set. The updated feature set is then used in the next iteration of the algorithm.

To perform feature selection, it is possible to use algorithms such as those proposed by Kohavi (1994, 1995), Salzberg (1993) and Rich (1994), which conduct a search in the space of feature subsets. However, to reduce the computational effort of the algorithm, we perform a selection that is based on an analysis of the generated decision tree. The criterion by which generated features are selected is their direct contribution to the generated decision tree:

$$\sum_{n_i \in T, f_{n_i} \neq f} \frac{|E_{n_i}|}{|E|} \text{InfoGain}(f_{n_i}, E_{n_i})$$

All the basic features are included in the feature set regardless of their utility in the tree, in order to retain the completeness of the searched feature space.

### 5.4 The Complexity of FICUS

During each iteration of the FICUS algorithm, a classification tree is built, and the feature generator is called for each of its nodes. Therefore, the complexity of the algorithm is the number of iterations times the complexity of building a tree. This complexity is dominated by the number of nodes times complexity of feature generation per node. The feature generator performs several phases. During each phase it generates and evaluates new features whose number is limited by a given parameter. The cost of evaluation depends on the evaluation methods used. The following analysis assumes a data-driven evaluation of building blocks.

Let $N_{\text{phase}}$ be the fixed number of generation phases (the internal loop of the generator). Let $N_{\text{new}}$ be the fixed number of features generated and evaluated at each search phase. The evaluation of each feature involves the calculation of its Info-Gain value, which, in the worst case of a continuous feature, is equal to the complexity of sorting the instance set of the generator. Let $n$ be a node of a decision tree being built and let $E_n$ be its local instance set. The complexity of the feature generator when activated for node $n$ is:

$$O_G(E_n) = |E_n| \log_2 |E_n| N_{\text{new}} N_{\text{phase}}$$  \hspace{1cm} (1)

Where $|E_n| \log_2 |E_n|$ is a bound on the complexity of calculating the Info-Gain value of a single feature, and $N_{\text{new}} \cdot N_{\text{phase}}$ is the total number of evaluated features.

The complexity of one iteration of the algorithm is measured by summing the value of Equation 1 over all the nodes of the produced tree. Let $E$ be the set of examples given to the FICUS algorithm. The total size of the local instance sets of nodes at each level of the decision tree is bounded by $E$. Therefore, the complexity of generating features in each complete level $i$ of the decision tree $T$ can be bounded as follows:

$$O_T^i(E) = \sum_{n \in T, \text{lev}(n) = i} O_G(E_n)$$

$$= N_{\text{phase}} N_{\text{new}} \sum_{n \in T, \text{lev}(n) = i} |E_n| \log_2 |E_n|$$

$$\leq N_{\text{phase}} N_{\text{new}} |E| \log_2 |E|$$  \hspace{1cm} (2)
normalization of $X$ to $[0, 1]$. $\alpha$ controls the weights given to the two components.

A possible problem with the data-driven approach is its disregard of feature interaction. It is quite possible that a feature poorly estimated by $h^d_i$ will produce a highly valued feature due to its interaction with another feature. We therefore propose an alternative hypothesis-driven evaluation criterion which can detect existing feature interactions.

The hypothesis-driven function evaluates the constructive utility of the building block features, by using them to build a decision-tree hypothesis, and then evaluating each feature by its contribution to it. A feature contributes to a tree by serving as a splitting feature of a node or by participating in an argument of another splitting feature. Let $E_{n_i}$ be the set of examples at node $n_i$. Let $f_{n_i}$ be the splitting feature of node $n_i$. The node-utility of feature $f$ is defined as:

$$u(f, n_i) = \begin{cases} \frac{|E_{n_i}|}{|E|} \text{InfoGain}(f_{n_i}, E_{n_i}) & f_{n_i} = f \\ \frac{|E_{n_i}|}{|E|} \frac{\text{InfoGain}(f_{n_i}, E_{n_i})}{\sum_{j=1}^{k} |A_j|} & f_{n_i} = f'(A_1, \ldots, A_k) \land f \in A_j \mid 1 \leq j \leq k \\ 0 & \text{otherwise} \end{cases}$$

When $f$ serves as a splitting feature, it is credited with its weighted information gain in that node. When it serves as part of an argument of a splitting feature, it is credited with the weighted Info-gain of the splitting feature divided by the total number of features in its arguments and discounted by $\gamma$. $\gamma$ expresses the fact that the utility of a constructed feature should not be entirely credited to its building blocks. The utility of $f$ with respect to the whole tree is measured by the sum of its node utilities:

$$h^b_i(f, T) = \sum_{n_i \in T} u(f, n_i)$$

One drawback of hypothesis-driven evaluation is that it may lead to a narrow search in the feature space. That is since out of an entire feature set typically only a small number of features are extensively used in the generated classifier. These features tend to overshadow other relevant features that where not included or rarely used in the classifier. This problem intensifies as the number of features increases, especially when the classifier is induced by a greedy algorithm. We have found out during our experiments that a hybrid strategy which employs an hypothesis-driven evaluation in the first search phase (in which the building block feature set is relatively small) and a data-driven evaluation in the following phases, combines the advantages of both approaches.

### 5.3 Feature Selection

**Ficus** maintains a fixed-sized feature set, which is the basis for its decision-tree learning and feature generation. The feature set consists of a fixed part which contains the basic feature set, and of constructed features which are updated at each iteration of the algorithm. At each iteration, **Ficus** induces a decision tree containing generated features, and then applies feature selection to choose a subset of them to replace
5.2.2 Feature evaluation criteria

The generator employs two different evaluation criteria: one for the target set and another for the building block set. The evaluation criterion used for ordering the target feature set, denoted by $h^E_f$, is supplied by the concept learner and is dependent on its current local instance set. In the current version of FICUS, which uses a decision-tree concept learner, $h^E_f$ is the splitting criterion used to split tree nodes (e.g., information gain).

The evaluation function applied to the set of building blocks, denoted by $h_b$, tries to predict their potential constructive utility, i.e., their utility as building blocks of new features. We propose two alternative functions for evaluating constructive utility: a data driven utility function, $h^d_b$, and an hypothesis driven utility function, $h^h_b$.

The data driven function considers three criteria for evaluating a feature: its target evaluation function, $h^E_f$, its complexity, Comp, computed by the size of its representing tree structure, and its relative improvement compared to its parent building blocks. The function directs the search process to prefer features with low complexity following the Occam Razor principle:

$$h^d_b(f) = \alpha \left[ \frac{h^E_f(f)}{\text{Comp}(f)} \right]_{\text{norm}} + (1 - \alpha) \left[ \frac{h^E_f(f)}{\text{Comp}(f)} \right]_{\text{norm}} \left[ \frac{\text{Comp}(f)}{\text{Comp}(p)} \right]_{\text{norm}}$$

where the left term represents the target value of $f$ normalized by its complexity. The right term measures the improvement of this normalized value of $f$ with respect to that of its parents. $[X]_{\text{norm}}$ denotes a
Generator \((F_{set}, E_{set}, FSS, Target_{func}, N_{phase}, N_{new})\)

\[\begin{align*}
\text{target}_{set} & \leftarrow F_{set} \\
\text{block}_{set} & \leftarrow F_{set} \cup \{ \text{The building block features of } F_{set} \text{ members} \} \\
\text{feature}_{rec} & \leftarrow \{ \} \\
\text{Pair}_{rec} & \leftarrow \{ \} \\
\text{For } \{ \text{phase} = 1 \ldots N_{phase} \} \\
& \text{Calculate the evaluation criterion values of } Target_{set}, \text{Block}_{set} \text{ members, and trim the sets accordingly.} \\
\text{new}_{set} & \leftarrow \{ \} \\
\text{While } (|\text{new}_{set}| \leq N_{new}) \\
& \text{pair} \leftarrow \text{highest evaluated pair of building block features, which} \\
& \text{is not already in } \text{pair}_{rec} \\
\text{pair}_{rec} & \leftarrow \text{pair}_{rec} \cup \{ \text{pair} \} \\
\text{new}_{set} & \leftarrow \text{new}_{set} \cup \text{Filter(Expand(} \text{pair}_{i}, E_{set}, FSS)) \\
& \text{Merge } \text{new}_{set} \text{ into } target_{set}, \text{blocks}_{set}. \\
& \text{return(} target_{set} \text{)} \\
\end{align*}\]

Expand \((\langle f_1, f_2 \rangle, E_{set}, FSS)\)

\[\begin{align*}
\text{constructed}_{set} & \leftarrow \{ \} \\
\text{If } (f_1 = f_2) & \text{ constructed}_{set} \leftarrow \text{Compose}(f_1) \cup \text{Interval}(f_1) \\
\text{else} & \text{ constructed}_{set} \leftarrow \text{Compose}(f_1, f_2) \cup \text{Insert}(f_1, f_2) \cup \\
& \text{Insert}(f_2, f_1) \cup \text{Replace}(f_1, f_2) \cup \text{Replace}(f_2, f_1) \\
& \text{return(} \text{constructed}_{set} \text{)} \\
\end{align*}\]

Filter \((F_{set})\)

\[\begin{align*}
\text{filtered}_{set} & \leftarrow \{ \} \\
\text{For each feature } f \in F_{set} : \\
& \text{If } (f \not\in \text{feature}_{rec}) \\
& \text{If } (Target_{func}(f)/\max(\text{Target(parents of } f)) > \text{threshold}) \\
& \text{filtered}_{set} \leftarrow \text{filtered}_{set} \cup \{ f \} \\
& \text{return(} \text{filtered}_{set} \text{)} \\
\end{align*}\]

Figure 6: FICUS - Pseudo Code (2)

Given a selected pair of features, \(f_1\) and \(f_2\), the algorithm uses the search operators to get a new set of constructed features, called \(\text{Expand}(f_1, f_2)\):

\[\text{Expand}(f_1, f_2) = \begin{cases} 
\text{Compose}(f_1, f_2) \cup \text{Insert}(f_1, f_2) \cup \text{Insert}(f_2, f_1) & f_1 \neq f_2 \\
\text{Union(} \text{Insert}(f_1, f_2) \cup \text{Replace}(f_1, f_2) \cup \\
& \text{Replace}(f_2, f_1) \cup \text{Compose}(f_1) \cup \text{Interval}(f_1)) & f_1 = f_2 
\end{cases}\]

At each search iteration, the search procedure iteratively expands selected building block pairs, until a fixed number of new features has been generated, or until all the existing pairs have been expanded. The new features are merged into the maintained target and building block sets, and a new iteration begins. The search algorithm maintains a record of previously generated features, to avoid their regeneration, as well as a record of previously expanded building block pairs, to avoid their recurrent expansion. In addition, a filter is used to remove features whose target evaluation criterion is not sufficiently higher than their parents. The pseudo code of the generator is presented in Figure 6.
**Ficus** (basic\_feature\_set, instance\_set, FSS, N\_iterations)

\[ \text{constructed\_feature\_set} \leftarrow \text{basic\_feature\_set} \]

for \( i = 0 \ldots N_{\text{iterations}} \)

\[ T \leftarrow \text{Generate\_Tree} (\text{feature\_set}, \text{instance\_set}, \text{FSS}) \]

\[ \text{constructed\_feature\_set} \leftarrow \text{Select\_Features} (T, \text{basic\_feature\_set}, N_{\text{selected}}) \]

return(constructed\_feature\_set)

---

**Generate\_Tree** (F\_set, E\_set, FSS)

if \( \text{All members of } E_{\text{set}} \text{ are of same class} \)

return a new created tree leaf.

else if \( |E_{\text{set}}| < \text{MinNode\_Size} \)

\[ f_{\text{best}} = f \in F_{\text{set}} \mid f \text{ has max split measure} \]

else

\[ \text{generated\_set} \leftarrow \text{Generator}(F_{\text{set}}, E_{\text{set}}, \text{FSS}, \text{Info\_Gain}, N_{\text{phase}}, N_{\text{new}}) \]

\[ f_{\text{best}} = f \in (\text{generated\_set} \cup E_{\text{set}}) \mid f \text{ has max split measure} \]

Create a new decision-tree node corresponding to \( f_{\text{best}} \)

For each partition subset \( E_{\text{subset}} \) received from splitting \( E_{\text{set}} \) using \( f_{\text{best}} \):

recursively call \( \text{Generate\_Tree}(F_{\text{set}}, E_{\text{subset}}, \text{FSS}) \)

return (decision-subtree)

---

**Select\_Features** (T, basic\_feature\_set, N\_selected)

\[ \text{selected\_set} \leftarrow \{\} \]

\[ \text{evaluated\_set} \leftarrow \text{The features composing the decision tree } T. \]

Evaluate the direct utility of each member of \( \text{evaluated\_set} \) in \( T. \)

\[ \text{selected\_set} \leftarrow \text{The } N_{\text{selected}} \text{ highest evaluated members of } \text{evaluated\_set} \]

\[ \text{selected\_set} \leftarrow \text{selected\_set} \cup \text{basic\_feature\_set} \]

return(selected\_set)

---

Figure 5: Ficus - Pseudo Code (1)

to guide it. The feature generator of Ficus employs a search strategy that is a variant of beam search. The generator maintains two fixed-sized sets of features: A set of building blocks for the construction of new features, and a target set of generated features which is the eventual output of the search procedure. The target set is initialized to include the members of the input constructed feature set (the output of the previous iteration of the Ficus algorithm). The set of building features is initialized to the union of:

- The input constructed feature set.
- The features from which the input constructed features are composed. These features were added to introduce a form of one-level backtracking.

The search algorithm uses two different evaluation criteria to order the two feature sets and to trim them to their fixed sizes.

The search operates iteratively, where at each iteration, new features are generated and added to the two maintained sets. The new features are generated by iteratively applying the search operators, defined in Section 4.2, to selected pairs of building blocks. The same feature may be selected for both pair elements, allowing for the application of the unary operators. The selection of building blocks from within the building block set is performed according to their associated evaluation criterion value.
**feature selector** selects a utile subset of generated features. The **concept learner** defines the local context for feature generation. Our current framework employs a decision tree concept learner (c4.5) for this purpose. Note that the concept learner is used as an *internal module* of the **Ficus** algorithm and is independent of the external concept learners that will eventually use the constructed feature set.

The algorithm maintains a set of constructed features initialized to the basic feature set. The algorithm iterates as long as computation resources are available. During each iteration it builds a classification tree using its input examples and its current set of constructed features. In the course of building the tree, the feature generator is activated for each new node using its local instance set. Based on the current constructed feature set, and on the global FSS definition, the generator generates new features that can highly discriminate the members of its input instance set. The new generated features are then used as additional candidates for splitting the node according to the splitting criterion of the tree concept learner. After the tree is built, the feature selection procedure selects a subset of the newly generated features that appear in the tree. The selected feature subset together with the basic features constitutes the new constructed feature set that is used in the next iteration. The algorithm terminates after a specified number of iterations, or as a result of an interactive user request, and returns its current feature set as output. Therefore, **Ficus** can be regarded as an *anytime* algorithm (Boddy, 1991), that is able to return its updated result at each point in time.

The generation strategy of **Ficus** is based on an evolutionary approach by which new features are continuously composed from highly-evaluated existing ones. This strategy is implemented in two levels: first, at the local level of each tree node by the activated feature generator, and second, at a global level, by gathering different features of the tree into one integrated set, (the constructed feature set), which is used to generate new features in the next iteration.

In the following subsections we present the components of the **Ficus** algorithm. The entire algorithm is listed in pseudo code in Figures 5, 6.

### 5.2 The Feature Generator

The **feature generator** is activated during the construction of each node of the decision-tree concept learner. The generator receives as input the currently employed constructed feature set, the tree node instances, and the FSS. The generator produces a set of features that are then used by the concept learner as candidates for splitting its current tree node. The generator searches the constructed feature space, $\mathcal{F}$, looking for features which best discriminate its input set of data instances. The architecture of the feature generator is illustrated in Figure 7.

#### 5.2.1 The search procedure

In general, feature generation can be viewed as a search that is conducted in a defined feature space. For **Ficus**, this space is defined by its supplied FSS. Since constructor functions may be activated in an hierarchic and recursive fashion, the defined feature space $\mathcal{F}$ can be very large or even infinite, making exhaustive search impractical. To efficiently search $\mathcal{F}$, a suitable search strategy is required, as well as an appropriate heuristic
5 The FICUS algorithm

In this section we present a feature construction algorithm, named FICUS. FICUS receives as input an FSS defined by the grammar presented in Figure 2 and a set of classified instances. FICUS searches the feature space, $\mathcal{F}$, defined by its input FSS using the presented search operators and returns a utile set of generated features.

5.1 Architecture

The general framework of FICUS is described in Figure 4. FICUS receives as input a set of basic features, a set of classified objects and an FSS which defines a set of constructor functions. The output of FICUS is a set of constructed features that can be used by any supervised concept learner to produce a corresponding classifier. The algorithm consists of three major modules. The feature generator generates new features. The

\[\text{The name FICUS stands for: Feature Incremental Construction System}\]
The \textbf{Insert} operator: Let $f_1, f_2 \in \mathcal{F}$. Let $f_2$ be denoted as $u(A_1, \ldots, A_k)$, where $u$ is an FSS constructor function, and $A_1 \ldots A_k$, its input arguments, then:

$$\text{Insert}(f_1, f_2) = \{ u(A'_1 \ldots A'_k) \mid (1 \leq i \leq k) \land \\
\quad (\forall j \neq i A'_j = A_j) \land \\
\quad (A'_i \in \text{Ins}(f_1, A_i)) \land \text{LegalArg}(u, i, A'_i) \}$$

Where:

$$\text{Ins}(f, A) = \begin{cases} 
\{ \} & \text{A is of simple type} \\
\{ \{ f_1 \ldots f_k, f \} \} & A = \{ f_1, \ldots, f_k \} \\
\{ \{ f'_1 \ldots f'_k \} \} & (1 \leq i \leq k + 1) \land (f'_i = f) \land \\
\quad (\forall j < i, f'_j = f_j) \land (\forall j > i, f'_j = f_{j-1}) \}
\end{cases}$$

The \textbf{Replace} operator: Let $f_1, f_2 \in \mathcal{F}$. Let $f_2$ be denoted as $u(A_1, \ldots, A_k)$, where $u$ is an FSS constructor function, and $A_1 \ldots A_k$, its input arguments, then:

$$\text{Replace}(f_1, f_2) = \{ u(A'_1 \ldots A'_k) \mid (1 \leq i \leq k) \land \\
\quad (\forall j \neq i A'_j = A_j) \land \\
\quad (A'_i \in \text{Rep}(f_1, A_i)) \land \text{LegalArg}(u, i, A'_i) \}$$

Where:

$$\text{Rep}(f, A) = \begin{cases} 
\{ f \} & \text{A is of simple type} \\
\{ \{ f'_1 \ldots f'_k \} \} & A = \{ f_1, \ldots, f_k \} \\
\{ \{ f'_1 \ldots f'_k \} \} & (1 \leq i \leq k + 1) \land \\
\quad (f'_i = f) \land (\forall j \neq i, f'_j = f_j) \} \\
\{ \{ f'_1 \ldots f'_k \} \} & A = \{ f_1, \ldots, f_k \} \\
\{ \{ f'_1 \ldots f'_k \} \} & (1 \leq i \leq k + 1) \land \\
\quad (f'_i = f) \land (\forall j \neq i, f'_j = f_j) \}
\end{cases}$$

The \textbf{Interval} operator: Let $f_1 \in \mathcal{F}$. We distinguish between two cases:

- $f_1$ is nominal or ordered-nominal: \text{Interval}(f_1) = \{ Is(f_1, \{ C_i \}) \mid C_i \in \text{Range}(f_1) \}$ where the builtin constructor \text{Is} is defined as \text{Is}(f_1, C_i) = \text{TRUE} \iff f_1 = C_i.  

- $f_1$ is continuous: Let $\{ C_1, C_2 \ldots C_n \}$ be a discretization of \text{Range}(f_1) (We use the dynamic discretization algorithm by Fayyad and Irani (1993)). Then \text{Interval}(f_1) = \{ InRange(f_1, \{ C_i, C_{i+1} \}) \mid 1 \leq i < n \}$, where the builtin constructor function \text{InRange} is defined as \text{InRange}(f_1, \{ C_i, C_{i+1} \}) = \text{TRUE} \iff C_i \leq f_1 \leq C_{i+1}.  

11
4.2 The Search Operators

In order to traverse a space $F$, derived from a given FSS, we have defined four types of general search operators. These operators receive either one or two existing features, and produce a set of newly constructed features. Although the presented operators do not express every possible method for combining existing features, it is easy to show that, given an FSS, they are sufficient for generating its defined legal feature space, $F$.

We briefly outline the four types of operators, and then define each of them more precisely.

1. **Compose**: receives one or two features from which it composes new features using all the suitable constructor function of the FSS.

2. **Insert**: receives two features and creates new ones by inserting one feature into the other.

3. **Replace**: receives two features and creates new features by replacing components of one feature with the other feature itself.

4. **Interval**: receives a feature and creates new features that test whether it lies within a specified range.

Note that at this point we introduced a seemingly strong assumption – that the constructor functions are either binary or unary. This is not as restrictive as it sounds since each of the arguments can be a set or a sequence. It is also possible to extend the definition to k-ary constructor functions. Such an extension, however, will increase the branching factor of the search graph.

**The Compose operator:** Let $f_1, f_2 \in F$ then:

$\text{Compose}(f_1, f_2) = \{u(A_1, A_2) \mid (u \in U) \land (|\text{args}(u)| = 2) \land (A_1 \in \{f_1, \{f_1\}, \{f_1\}\}) \land (A_2 \in \{f_2, \{f_2\}, \{f_2\}\}) \land \text{LegalArg}(u, 1, A_1) \land \text{LegalArg}(u, 2, A_2)\}^1 \cup \{u(A_1) \mid (u \in U) \land (|\text{args}(u)| = 1) \land (A_1 \in \{\{f_1, f_2\}, \{f_1, f_2\}, \{f_2, f_1\}\}) \land \text{LegalArg}(u, 1, A_1) \}$

The unary version of Compose is defined as:

$\text{Compose}(f_1) = \{u(A_1) \mid (u \in U) \land (|\text{args}(u)| = 1) \land (A_1 \in \{f_1, \{f_1\}, \{f_1\}\}) \land \text{LegalArg}(u, 1, A_1) \}$
4 Feature generation as search

In general, feature generation can be viewed as a search conducted in a defined feature space. In this section we define the search space of constructed features derived from a given FSS.

4.1 The search space

In order to formulate the definition of the searched feature space, as well as the operators that are used to traverse it, we first define when an argument placement is legal. This definition is based on type compatibility.

**Definition 5** Let $T_1$ and $T_2$ be types of constructor function arguments. $T_1$ is compatible with $T_2$ (denoted by $\text{CmpType}(T_1, T_2)$) if and only if:

$$\text{CmpType}(T_1, T_2) = ((T_1 = t_1, T_2 = t_2 \mid t_1, t_2 \in \text{types}(\text{FSS})) \land$$

$$((t_1 \text{ identical to or inherited from } t_2)) \lor$$

$$((T_1 = \text{set of } t_1, T_2 = \text{set of } t_2) \land$$

$$\text{CmpType}(t_1, t_2)) \lor$$

$$((T_1 = \text{sequence of } t_1, T_2 = \text{sequence of } t_2) \land$$

$$\text{CmpType}(t_1, t_2))$$

We can now define the legality of argument placements.

**Definition 6** Let $u$ be a constructor function and $i$ an argument index. Let $A$ be a constructor function argument (a feature, a set of features or a sequence of features). The placement of $A$ as the $i$'th argument of $u$ is defined to be legal (denoted by $\text{LegalArg}(u, i, A)$) if and only if:

$$\text{LegalArg}(u, i, A) = ((\text{CmpType}(\text{type}(A), \text{type}((\text{args}_i(u)))) \land$$

$$\forall f_c \in \text{constraints}(\text{args}_i(u)), (f_c(A)))$$

Given an input FSS, the space of legal constructed features is defined as the set of all legal compositions of basic features into constructed features using the FSS constructor functions.

**Definition 7** Let $E$ be the instance space. Let $T$ be the type set of the FSS. Let $C_f$ be the set of all constant features in the union of ranges of types in $T$. Let $F_b$ be the basic feature set of the FSS. Let $U$ be the constructor function set of the FSS. The space of legal constructed features, $\mathcal{F}$, is defined as follows:

1. $F_b \subseteq \mathcal{F}$

2. Let $u$ be a constructor function of arity $k$ in $U$. Let $A_1 \ldots A_k$ each be a feature, a finite set of features or a finite sequence of features, in the range $\{\mathcal{F} \cup C_f\}$, such that $\forall i, 1 \leq i \leq k : \text{LegalArg}(u, i, A_i)$. Let $f$ be a feature defined as $\forall e \in E, f(e) = u(A_1(e), \ldots, A_k(e))$. Then $f \in \mathcal{F}$.

The structure of features in $\mathcal{F}$ is a tree structure whose intermediate nodes contain constructor functions, and whose leaves contain basic features and constants.
whether it complies to a given constraint. The FICUS system, described in Section 5, supplies a builtin set of constraint functions that enable the user to forbid or enforce the use of constants, to restrict the size of sets and sequences, and to forbid duplications in sequences. In addition to the builtin constraint functions, the user may supply constraint functions which represent domain background knowledge.

** Types **

<table>
<thead>
<tr>
<th>Type</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool</td>
<td>nominal, {&quot;false&quot;, 0}, {&quot;true&quot;, 1}</td>
</tr>
<tr>
<td>Slot</td>
<td>ordered-nominal, {&quot;O&quot;, -1}, {&quot;B&quot;, 0}, {&quot;X&quot;, 1}</td>
</tr>
<tr>
<td>Float</td>
<td>continuous, {-MAXFLOAT, +MAXFLOAT}</td>
</tr>
</tbody>
</table>

** Basic-Features **

- $S_{11}$ := Slot
- $S_{12}$ := Slot
- $S_{13}$ := Slot

** Constructor-Functions **

- Max := Slot, \{set of Slot, \{NoConst\}\}
- Avg := Float, \{set of Slot, \{NoConst, Unique\}\}
- Is := Bool, \{Slot, \{NoConst\}\}, \{Slot, \{Const\}\}
- > := Bool, \{seq of Slot, \{NoConst, Unique\}\}
- And := Bool, \{Bool, \{NoConst\}\}, \{Bool, \{NoConst\}\}

Figure 3: An FSS for the Tic-Tac-Toe domain

Figure 3 shows an example FSS for feature generation in the domain of Tic-Tac-Toe end games. The type set of the FSS consists of three types: Boolean, Float and Slot. The Slot type represents the value of a board slot and is inherited from the ordered-nominal type. Its range consists of the ordered nominal values "O", "B" (for blank) and "X". The basic feature set of the FSS consists of 9 features, of type "Slot" representing the 9 board positions \{S_{11}, \ldots, S_{33}\}. To represent a larger problem game board such as 4x4, it is only required to change the basic feature set of the current FSS to the 16 features \{s_{11}, \ldots, s_{44}\}.

The FSS defines five constructor functions, each consisting of a return type and arguments specification. The definition makes use of three argument constraint functions: NoConst, forbidding constant features, Const, enforcing constant features, and Unique forbidding identical elements.
Figure 2: The FSS Grammar. Defines a language for writing Feature Space Specifications. Sets of elements are denoted by \( \{ \ldots \} \), while sequences are denoted by \( \langle \ldots \rangle \).

Continuous types are specified by their boundaries. Basic features are defined by their id and type. Constructor functions are defined by their id, return type, and the specification of their input arguments. An argument specification denoted as \( \text{arg-spec} \) is composed of an argument type and constraint set.

This scheme allows us to specify how to compose new constructed features with a predefined finite set of arguments. Many functions, however, such as \( \text{Min} \), \( \text{Max} \), \( \sum \), \( \prod \), etc., may be applied to a variable number of arguments. Due to their associative nature, it is possible to represent such functions as binary functions, which can effectively operate on an unlimited number of arguments by means of recursive activation. This solution, however, is not adequate for unassociative functions, such as \( \text{Average} \), which calculates its arguments average, \( > \) which tests whether its arguments are sorted, \( = \) which tests whether its arguments are equal, or \( \text{Count} \) which returns the number of its positive arguments. Such unassociative functions that may operate on an unlimited number of arguments, could only be represented by an infinite series of finite arity functions. To overcome this difficulty, we have defined our constructor functions to be also able to receive \( \text{sets} \) and \( \text{sequences} \) of features, as individual arguments. In this way a function such as \( \text{Average} \) could be defined as receiving a single argument of type \( \text{set} \), rather then being defined by an infinite series of finite arity functions. Sequences are used for constructors that are order-dependent, such as \( > \). An argument type denoted as \( \text{arg-type} \) is therefore defined either as a \( \text{type} \), a \( \text{set of type} \), or a \( \text{sequence of type} \).

An argument constraint set is a set of boolean constraint functions that receive an argument and test
The utility of a feature construction algorithm, is measured by the utility of its produced feature set, which in turn is measured by comparing a classifier that was produced using it to a classifier that was produced using the original basic feature set. The classifiers are compared by criteria such as accuracy, comprehensibility and complexity.

**Definition 4** Let \( E_C, F_i, S \) and \( U \) be defined as above. An evaluation criterion is a real-valued function \( v : S \rightarrow \mathbb{R} \) that is used to evaluate classifiers. Let \( l \) be a concept learner. The utility of a feature set \( F \) with respect to \( F_i, E_C, l \), and \( v \) can be defined as:

\[
Util(F) = v(l(E_C, F)) - v(l(E_C, F_i)).
\]

The utility of a construction algorithm \( \varphi \), with respect to \( F_i, U, E_C, l \), and \( v \) is measured by the utility of its produced feature set:

\[
Util(\varphi) = v(l(E_C, \varphi(E_C, F_i, U))) - v(l(E_C, F_i)).
\]

The purpose of this work is to design a general feature construction algorithm that generates high utility feature sets for a variety of classification problems.

### 3 A specification language for defining feature representation

In the introduction we set a goal of developing a methodology for feature generation, where the representation is not predefined as part of the generation algorithm, but is rather supplied as input by the user. In this section we define a language for formulating specifications of representation schemes. Such a specification defines the space of the constructed features which will be searched by the generation algorithm.

More specifically, we define a language that allows us to specify the following:

1. The set of basic features.
2. The set of constructor functions.
3. The domain and range of each constructor function.
4. A set of constraints over the application of the constructor functions.

The description of these items written in the specification language is called *Feature Space Specification* (FSS). Figure 2 presents a grammar that defines a language for writing FSS. The definition of an FSS is based on a set of *types* for the domains and ranges of the constructor functions and basic features.

An FSS specifies a hierarchy of types used to define the domains and ranges of basic features and constructor functions. The leaves of the hierarchy are atomic types, and the types at the intermediate nodes are supersets of their children. The types in the hierarchy are either nominal, ordered-nominal or continues. Ordered-nominal types are specified by enumerating the type elements together with associated ordinals.
define a supervised concept learner as follows:

**Definition 1** Let $E$ be a finite instance set. Let $C$ be a finite set of categories. A classified example is a pair $(e, c)$, where $e \in E$ and $c \in C$. A feature is defined as a function over $E$. A classifier is a function $s : E \rightarrow C$. Let $S$ be the set of all possible classifiers from $E$ to $C$. A supervised concept learner is defined as an algorithm that given a set of classified instances $E_C$, and a set of initial basic features $F_b$, produces a classifier $s \in S$.

A set of constructor functions $U$ defines the space of possible constructed features $F_C$, over the set of basic features $F_b$, and constants.

**Definition 2** Let $E$ be the instance space. Let $F_b$ be a set of basic features. Let $U$ be a set of constructor functions. Let $C_f$ be the set of constant features in the union of ranges of $F_b$ and $U$. The space of constructed features, $F_C$, is then defined as follows:

1. $F_b \subseteq F_C$

2. Let $u : d_1 \times \ldots \times d_k \rightarrow \text{range}(u)$ be a function of arity $k$ in $U$. Let $f_1, \ldots, f_k$ be a finite sequence of features in $\{F_C \cup C_f\}$, such that for $1 \leq i \leq k : \text{range}(f_i) \subseteq d_i$. Let $f$ be a feature defined as $\forall e \in E, f(e) = u(f_1(e), \ldots, f_k(e))$. Then $f \in F_C$.

The above definition allows us to define a feature construction algorithm:

**Definition 3** A feature construction algorithm, is an algorithm that receives as input a set of basic features $F_b$, a set of classified examples $E_C$ and a set of constructor functions $U$, and produces a set of constructed features $F_{out} \subseteq F_C$. 

---

Figure 1: The Generation Framework
variables. Genetic algorithms, such as GABIL (Jong, Spears, & Gordon, 1992) and GA-SMART (Kira & A. Rendell, 1992), employ a bit-string representation of features and generate new features as a result of genetic operations such as crossover and mutation. A drawback of the bit-string representation is that it does not express feature structure, which may lead to the generation of meaningless and illegal features.

As opposed to the above algorithms, whose rigid representational form limits their scope of application, our presented framework was designed for a flexible and general form of feature generation, where the representation language can be supplied as part of the problem definition.

Our framework treats feature generation as a search over the dynamic space of constructed features. We start by defining this space, and continue with the definition of the general search operators that allow us to traverse it. We then describe our general FICUS algorithm for feature generation. FICUS is based on an iterative activation of a decision-tree concept learner which is used to define the local context of feature generation. For each node in the tree, a search in feature space is performed, combining highly evaluated features into new ones, using the defined search operators. The search is guided by general heuristic functions that are uniformly applied to features, regardless of their representational form. The search heuristics employ data-driven as well as hypothesis-driven construction strategies.

Our framework was experimentally evaluated in a variety of problem domains. The generated features were evaluated by comparing classifiers that were produced using the new features to classifiers that were produced using only basic features. The generated features significantly improved the comprehensibility of the produced classifiers by capturing important elements of the underlying target concept. The new features also significantly improved the accuracy of the resulting classifiers, as well as reduced their complexity.

Section 2 describes the framework for function-based feature generation. Section 3 defines a language for formulating specifications of feature representation. Section 4 defines the search space of constructed features derived from a given FSS. Section 5 presents the FICUS algorithm. Section 6 describes the experimental evaluation of the algorithm. Section 7 compares FICUS with related algorithms and concludes.

2 A Framework for Function-based Feature Generation

In this section we present a general framework for describing feature generation algorithms. A supervised concept learner receives as input a set of basic features and a set of examples and produces a classifier. In our framework, a feature construction algorithm receives, in addition, a set of constructor functions. The feature generation algorithm produces a set of constructed features which are added to the set of features supplied to the concept learner. Our generation framework broadens the classic framework of supervised concept learning, by introducing new basic elements called constructor functions. These functions, which can be mathematical, logical or domain-specific, are used as the basis for feature generation. The framework is illustrated in Figure 1. In Section 5 we describe the architecture for the FICUS algorithm which is based on this framework.

The set of constructor functions define the space of constructed features. Before we define this space, we
A grammar which describes a language for feature-construction specifications. Such a specification is written by the user based on its partial knowledge of the domain and is used to define the space of constructed features.

A feature construction algorithm that performs heuristic search over the space of constructed features defined by the user-supplied specification.

The formulation and usage of existing background knowledge, often plays a prominent role in the achievement of successful concept learning. Many times classification algorithms are employed by people who may not know the problem concept, but who often possess some degree of partial knowledge concerning the problem domain. Our framework enables to exploit background knowledge about potentially significant relations and functions and their properties for constructing structured features.

The problem of automatic feature generation has received significant attention during the last decade. Several special-purpose algorithms were designed for specific problem domains (Hirsh & Japkowicz, 1994; Sutton & Matheus, 1991). For example the bootstrapping algorithm (Hirsh & Japkowicz, 1994), that was designed especially for the domain of molecular biology, represents features as nucleotide sequences, and uses a special set of operators to alter them into new sequence features. Such special-purpose algorithms may be effectively tailored for a given domain, but may be hard to generalize to other domains and problems.

More general construction algorithms use a feature representation, that can be employed for different domains and problems using a fixed set of construction operators. Many algorithms, such as FRINGE (Pagallo & Haussler, 1990), CITRE (Matheus & Rendell, 1989), IB3-Cl (Aha, 1991), LFC (Ragavan, Rendell, Shaw, & Tesmer, 1993) and GALA (Hu & Kibler, 1996), use a minimal set of logical operators (such as \{\neg, \land\}) to express existing boolean relations between data attributes. When the target concept is best described by more complex boolean relations, starting from the minimal set of logical operators is often inefficient. Several algorithms have been developed to use a predefined set of complex boolean relations. The ID2-or-3 (Murphy & Pazzani, 1991) and X-OR-N (Zheng, 1996) algorithms were designed to construct M-or-N typed features, which are believed to be relevant especially in medical domains. The MRP algorithm (Rendell, 1995) uses relational projection features that are able to describe complex boolean concepts.

A different form of feature representation, especially suited for continuous attributes, are hyperplanes. A hyperplane is a linear plane that splits the instance space into two subspaces. Hyperplanes can be axis parallel as in C4.5 (Quinlan, 1989) or multivariate as in LMDT (Utgoff & Brodley, 1991), SADT (Heath, Kasif, & Salzberg, 1993) and CART (Breiman, Friedman, Olshen, & Stone, 1984). Such multivariate hyperplanes are induced by methods of linear regression and weights adjustments. Hyperplane representation may be suitable for problems of an appropriate bias, however it can not be adapted to include domain background knowledge, and may suffer from poor comprehensibility.

Rule based systems such as STRUCT (Watanabe & Rendell, 1991), AQ17-HCI (Wenk & Michalski, 1994) and PRAX (Bala, Michalski, & Wenk, 1992) employ clause-formed rules as their feature representation. Special operators are used to alter existing rules, by means of specialization which involves adding terms to the rule’s conditions, or generalization which is achieved by deleting terms or substituting them with
1 Introduction

Research and practice have shown that standard concept learning algorithms, such as C4.5 (Quinlan, 1989), CN2 (Clark & Niblett, 1989) and IBL (Aha, Kibler, & Albert, 1991), degrade in performance when given data attributes that are not directly and independently relevant to the learned concept (John, Kohavi, & Pfleger, 1994; RagHAVAN & Rendell, 1993). Two problems have been noticed in this respect: feature irrelevance and feature interaction. The problem of feature irrelevance was addressed by designing algorithms that perform feature selection (Kira & A. Rendell, 1992; John et al., 1994; Kohavi & Dan, 1995; Sangiovanni-Vincentelli, 1992; Caruana & Freitag, 1994; Salzberg, 1993).

The problem of Feature Interaction has been addressed by constructing new features from the basic feature set. This technique is called Feature Construction. The new generated features may lead to the creation of more concise and accurate classifiers. In addition, the discovery of meaningful features contributes to better comprehensibility of the produced classifier, and better understanding of the learned concept. The conclusive majority of feature construction algorithms have been specifically designed to generate features of a rigidly predefined representation. Among the popular representations are simple boolean expressions, M-of-N expressions, hyperplanes, logical rules and bit strings. Most construction algorithms employ special-purpose construction methods and heuristics that are especially suited to their underlying representation. Each of these representations was shown to be beneficial in specific classes of problems. For example, it was shown that M-of-N expressions are particularly useful for medical classification problems where expert systems make use of “criteria tables” which are essentially M-of-N concepts (Murphy & Pazzani, 1991).

There are, however, several problems with the above scheme:

1. Given a new classification problem, it is not obvious which of the various representations and associated algorithms should be selected.

2. It is possible that none of the existing schemes is the right one for the problem at hand. In many real-world classification problems, the target concept is best expressed by features constructed using domain-specific knowledge. The above algorithms with their strict constructor set can not exploit such knowledge.

3. The rigidity of the existing algorithm does not allow an easy way of altering the representation. For example, even when we decide to use logical constructors, the current schemes do not allow an easy alternation of their existing constructor set.

4. Some classification problems may require a combination of several representation schemes which is difficult to do with existing feature construction algorithms.

In this paper we propose a methodology for feature generation which is general enough to address the above problems. In addition to its capabilities of handling these problems, the methodology is also useful as a unifying framework for feature generation whose descriptive power is sufficient for expressing most of the existing representation schemes. The framework consists of two main elements:
Feature Generation Using General Constructor Functions

Shaul Markovitch and Dan Rosenstein
Computer Science Department
Technion, Haifa, Israel

Abstract

Most classification algorithms receive as input a set of attributes of the classified objects. In many cases, however, the supplied set of attributes is not sufficient for creating an accurate, succinct and comprehensible representation of the target concept. To overcome this problem, researchers have proposed algorithms for automatic construction of features. The majority of these algorithms use a limited predefined set of operators for building new features. In this paper we propose a generalized unifying framework that is capable of generating features based on any given set of constructor functions. These functions can be domain-independent such as arithmetic and logic operators, or, can be domain-dependent operators that rely on partial knowledge of the user. The paper describes an algorithm which receives as input a set of classified objects, a set of attributes, and a specification of a set of constructor functions, which contains their domains, ranges and properties. The algorithm produces as output a set of generated features that can be used by standard concept learners to create improved classifiers. The algorithm maintains a set of its best generated features and improves this set iteratively. During each iteration, the algorithm performs a beam-search over its defined feature space, and constructs new features by applying constructor functions to the members of its current feature set. The search of the algorithm is guided by general heuristic measures, that are not confined to a specific feature representation. The algorithm was applied to a variety of classification problems and was able to generate features that were strongly related to the underlying target concepts. These features also significantly improved the accuracy achieved by standard concept learners, for a variety of classification problems.

Keywords: Constructive Induction, Feature Generation, Decision tree learning