An Efficient Approximation to Lookahead in Relational Learners

Jan Struyf[1], Jesse Davis[2], and David Page[2]

[1]Katholieke Universiteit Leuven, Dept. of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium
Jan.Struyf@cs.kuleuven.be
[2]University of Wisconsin-Madison, Dept. of Biostatistics and
Medical Informatics and Dept. of Computer Sciences
1300 University Avenue, Madison, WI 53706, USA
jdavis@cs.wisc.edu, page@biostat.wisc.edu

Abstract. Greedy machine learning algorithms suffer from shortsightedness, potentially returning suboptimal models due to limited exploration of the search space. Greedy search misses useful refinements that yield a significant gain only in conjunction with other conditions. Relational learners, such as inductive logic programming algorithms, are especially susceptible to this problem. Lookahead helps greedy search overcome myopia; unfortunately it causes an exponential increase in execution time. Furthermore, it may lead to overfitting. We propose a heuristic for greedy relational learning algorithms that can be seen as an efficient, limited form of lookahead. Our experimental evaluation shows that the proposed heuristic yields models that are as accurate as models generated using lookahead. It is also considerably faster than lookahead.

1 Introduction

Symbolic machine learning algorithms, such as rule or decision tree induction algorithms, search a large space of candidate models to find a suitable model [8]. Each search step consists of generating a new model and evaluating it on the set of training examples. This is repeated until a sufficiently accurate model is found. Naïvely enumerating all possible models is generally too computationally expensive, therefore, machine learning algorithms employ intelligent search strategies, such as greedy, randomized, or branch-and-bound search.

Greedy or hill-climbing search is often used because of its computational efficiency. Greedy search constructs a sequence of models such that each model is a locally optimal refinement of the previous model in the sequence. The search ends after meeting a stop criterion, such as no refinement significantly increases the evaluation score. The main disadvantage of greedy search is its shortsightedness. Greedy search misses refinements that yield a high evaluation score only in combination with further refinements. Inductive logic programming (ILP) [7] algorithms are especially susceptible to this problem. ILP algorithms use first-order logic to represent the data and models. Data are represented with facts and rules are represented by clauses. Consider the clause `happy(C)`$\leftarrow$`account(C,A,B)`$\wedge$`B=high`, which states that a customer of a bank is happy if one of the customer's accounts has a high balance. Greedy search will fail to find the first literal of this clause because it is non-discriminating, that is, it does not alter the evaluation score of the clause. The evaluation score (e.g., the accuracy of the clause) usually only depends on the number of positive and negative examples covered by the clause. In our example, all customers have at least one account. Therefore the examples covered, and consequently the evaluation scores of the clauses `happy(C)`$\leftarrow$`true` and `happy(C)`$\leftarrow$`account(C,A,B)`, are identical. Greedy search will thus never select the latter clause and therefore never find our example clause.

Lookahead [9] helps greedy search overcome myopia. Instead of adding only one literal at each search step, lookahead adds the best conjunction consisting of at most $n+1$ literals, where n is the lookahead depth. With depth one lookahead, our example clause could be learned in one refinement step. Unfortunately, execution time increases exponentially with the lookahead depth.

In this paper, we propose a feature based evaluation score for literals that is comparable to a limited form of depth one lookahead. Our score can be computed efficiently from a set of tables that can be pre-computed with one pass over the data. The resulting approach is computationally more efficient than lookahead and yields models that have a comparable accuracy.

2 An Efficient Approximation to Lookahead

We first illustrate the idea behind our approach with the example task of predicting when a bank customer is happy. Consider evaluating the clause `happy(C)` $\leftarrow$ `account(C,A,B)`. In this case, variables `A` and `B` serve as output variables. When computing the score of the clause, we can potentially glean information from which constants bind to these variables. This information about bindings can help guide the search towards which refinement to pick.

Assume we use accuracy as the scoring function. Then the refined clause `happy(C)`$\leftarrow$`account(C,A,B)` yields no benefit over the clause `happy(C)`$\leftarrow$`true` (because it covers the same examples). Even though both clauses have the same score, intuitively the refined clause appears more promising. By looking at the bindings for variable `B` we could observe, for example, that people who have an account with a high account balance tend to be happy: variable `B` takes a binding of *high* more frequently for *Happy* cases compared to *Not Happy* cases. Thus, we can see that by placing a condition on the variable `B` in a future step – that is, adding a literal with `B` as input – we might be able to get a clause with a high accuracy.

By looking at the bindings for `B`, we have leveraged the fact that the constants that bind to this variable are shared across several examples. It is more difficult to perform this analysis for a variable such as `A` because it has a unique value (account number) for every account. To illustrate how we might handle this type of variables, assume that we have three unary predicates that take an account number as input: `savingAcc(A)`, `checkingAcc(A)`, and `moneyMarketAcc(A)`. Even if all customers with money market accounts are happy, and all others are not happy, the individual account numbers tell us nothing about whether a customer is happy. The benefit comes from the fact that these account numbers refer to money market accounts. Thus, we can assess the clause's quality based on which predicates, like `moneyMarketAcc(A)`, hold for the bindings of variables. Nevertheless, we wish to do this assessment without the full cost of lookahead.

2.1 Features and Evaluation Scores

In the previous section, we have shown that class-wise counts for variable bindings, computed either directly or indirectly via the predicates that hold for each binding, yield extra information about the quality of a clause. In this section, we formalize this idea.

Suppose we have clause c and need to evaluate the quality of extending c with the literal l. Instead of just counting the number of examples in each class that the resulting clause covers, we also compute information based on the bindings for the output variables of l. We define a set of features for these bindings, which can be pre-computed prior to running the learning algorithm. The first feature is l itself. The other features are conjunctions of l with a second literal. The second literal uses exactly one of l's output variables as an input variable and does not share any other variables with clause c[1].

Definition 1 (Literal Features). *Given a clause c and a literal l, the set of features* $\mathrm{F}(c,l)$ *for l given c is defined as follows.*

$$\mathrm{F}(c,l) = \{\ l\ \} \ \cup\ \{\ l \wedge l_i \mid \mathrm{legal}(c,l,l_i)\ \}$$

$$\mathrm{legal}(c,l,l_i) = \#\{(\mathrm{vars}(l) - \mathrm{vars}(c)) \cap \mathrm{vars}(l_i)\} = 1\ \wedge\ \mathrm{vars}(l_i) \cap \mathrm{vars}(c) = \emptyset$$

with $\mathrm{vars}(x)$ *the set of variables appearing in x. (# denotes set cardinality.)*

Consider again the bank domain and suppose that the learner starts with clause $c =$ `happy(C)`$\leftarrow$`true` and is about to evaluate literal $l =$ `account(C,A,B)`. The first column of Table 1 lists the features $\mathrm{F}(c,l)$.

We define an evaluation score that incorporates information from the feature set. Several feature representations are possible and each representation allows different evaluation scores to be defined. The most general representation can be found in column R1 of Table 1. The columns for customer c_1 to customer c_n indicate for each customer which features hold. Column R2 of Table 1 shows a second representation, which stores class-wise counts of the examples for which each feature holds. Column R3 contains the most restrictive representation. Here we only compute some score for each feature.

In the most general representation of the features (R1), each class of customers (happy or unhappy) can be seen as a cluster with each instance being

[1] The constraints are introduced to limit the number of features.

Table 1. Feature representations for the literal `account(C,A,B)` in the bank application. (We assume that predicate arguments are typed; the same variable can only appear at argument positions of the same type.)

	R1			R2		R3
F(c, l)	c_1	...	c_n	#Happy	#Unhappy	Score
`account(C,A,B)`	1	...	1	3000	3000	0.50
`account(C,A,B)`∧`B=high`	1	...	0	2600	0	0.93
`account(C,A,B)`∧`B=medium`	0	...	1	2000	1000	0.67
`account(C,A,B)`∧`B=low`	0	...	1	200	3000	0.03
`account(C,A,B)`∧`card(A,R)`	1	...	1	3000	3000	0.50
`account(C,A,B)`∧`loan(A,L)`	0	...	1	1000	2000	0.33
...	...	...	...	...	...	...

the feature representation of one of the training examples (i.e., a column of R1). If the cluster of the happy customers is far apart from the cluster of unhappy customers according to some distance metric, then the literal is potentially good at separating the happy from the unhappy customers and should be assigned a high score. For example, one could define the quality of the literal as the Euclidean distance between the prototypes of the happy and the unhappy customers.

In this paper, we restrict ourselves to the least general representation (R3). R3 represents each feature by a score (e.g., its classification accuracy). The score of a literal is computed by aggregating the scores of its features. We choose to compute the score of a literal as the maximum of the scores of the features. Note that this choice can be compared to a limited form of depth one lookahead. Depth one lookahead, however, imposes fewer restrictions. It allows conjunctions with literals sharing no variables with l, literals that share more than one variable with l, and that share variables with the rest of the clause.

2.2 Efficiently Computing Class-wise Counts

In this section, we show how to efficiently compute class-wise example counts for the features, which corresponds to representation R2 in Table 1. Note that R3 can be easily computed from R2. The algorithm for finding the class-wise counts relies on a set of pre-computed tables.

We construct one table for each type. Each of these tables contains one column for each predicate argument of that type. The table is indexed by the constants in the domain of the type. Each cell $T_t[x, p, a]$ of the table for type t stores a Boolean indicating that predicate p holds if constant x is substituted for argument a.

The ComputeCounts algorithm in Fig. 1 uses the pre-computed tables to calculate the class-wise counts for the features. The algorithm creates a table called Counts, which has a row for each feature and a column for each class. The main loop of the algorithm iterates over the training examples. For each training example, it computes which features hold in the array Holds (the array

```
procedure ComputeCounts($c, l$)
1:  $c' := c$ extended with $l$; $V :=$ vars($l$) $-$ vars($c$)
2:  for each training example $e$
3:      if the condition of $c'$ holds for $e$
4:          Holds[1] := 1; $\forall i > 1$ : Holds[$i$] := 0
5:          for each binding $X/x$ of a variable $X \in V$
6:              for each column $(p, a)$ of $T_{\text{type}(X)}$
7:                  $i$ such that predicate$(l_i) = p \wedge$ argument$(l_i, a) = X$
8:                  if $T_{\text{type}(X)}[x, p, a] = 1$ then Holds[$i$] := 1
9:          $\forall i$ : Counts[$i$][class($e$)] := Counts[$i$][class($e$)] + Holds[$i$]
10: return Counts
```

Fig. 1. An algorithm computing the class-wise example counts for the features.

Holds corresponds to a column of R2 in Table 1). Next, it increments the count for each feature that holds, conditioning on the example's class.

To compute which of the features hold for a given training example, the algorithm executes the given clause c extended with literal l on the example. If it covers the example, then the first element of Holds is set to one. We assume here that the first feature is the literal itself. To compute whether or not the other features hold, the algorithm looks at the bindings that are obtained for the output variables V of l while executing the clause. For each binding of $X \in V$ to a constant x, it looks up the corresponding row in the pre-computed table for X's type. Each element in this row indicates if a given feature holds for this binding. The algorithm records this in the array Holds. After all bindings have been processed, Holds indicates the features that hold for the example. Holds can now be used to update the class-wise counts.

ComputeCounts is more efficient than computing the required counts for each of the features separately for two reasons. First, it computes the counts for all features in one pass over the data. This is, clause c extended with l needs to be executed only once on each example instead of once for each feature. Second, it caches in pre-computed tables whether or not a given feature holds.

3 Experimental Evaluation

We compare the feature based evaluation of literals (FBE) presented in this paper to lookahead. The conjecture is that (1) models built using FBE have a comparable accuracy to models built using lookahead and (2) FBE is considerably faster.

We test FBE in the ILP algorithm Tilde [2], that is available in ACE 1.2.9 [3]. Tilde induces first-order logical decision trees. Briefly, these are decision trees similar to the ones of C4.5 [10], but the tests in the internal nodes are expressed in first-order logic, meaning that each test is a conjunction of one or more literals. We use the exhaustive lookahead feature of Tilde. For a lookahead depth of n each node can contain at most $n + 1$ literals and these are found by

Table 2. Comparison of TILDE with our new FBE approach to TILDE with exhaustive lookahead of depth 0 to 2. The columns represent the data set and its number of positive/negative examples, the accuracy and AUPRC measured using cross-validation (with 90% confidence intervals), the CPU time for the cross-validation (not including loading of the data and pre-computing tables - the latter are small and range from 0.01 to 0.32 sec/fold), and the average tree size. Significant wins/losses of lookahead versus FBE are indicated with $\oplus$ and $\ominus$ (significance level 0.01). All experiments are performed on an Intel Xeon 3.3GHz / 4GB Linux system.

Data	Method	Accuracy		AUPRC		Time (sec)	Size (nodes)
Muta188	L0	69.1 ± 7.5		70 ± 8	$\ominus$	1	1.0
#p = 125	L1	74.5 ± 4.7		84 ± 7		62	11.8
#n = 63	L2	73.9 ± 6.7		79 ± 6		1455	11.8
	FBE	76.6 ± 5.3		85 ± 8		13	14.8
Muta230	L0	63.9 ± 3.3	$\ominus$	65 ± 4	$\ominus$	1	1.7
#p = 138	L1	74.8 ± 5.8		84 ± 3		321	18.8
#n = 92	L2	73.5 ± 3.4		81 ± 7		2482	15.3
	FBE	74.8 ± 4.7		86 ± 4		36	19.2
Financial	L0	86.8 ± 0.7	$\ominus$	13 ± 1	$\ominus$	0	0.0
#p = 31	L1	96.6 ± 1.5		84 ± 9		25	2.0
#n = 203	L2	96.2 ± 1.8		81 ± 9		2716	1.4
	FBE	96.6 ± 1.5		84 ± 9		13	2.0
Sisyphus A	L0	62.1 ± 0.0	$\ominus$	62 ± 0	$\ominus$	3	0.0
#p = 10723	L1	94.9 ± 0.5		97 ± 1		4302	18.6
#n = 6544	L2	96.6 ± 0.2	$\oplus$	98 ± 0	$\oplus$	161253	22.7
	FBE	94.8 ± 0.3		97 ± 1		779	17.4
Sisyphus B	L0	71.4 ± 0.0	$\ominus$	29 ± 0	$\ominus$	1	0.0
#p = 3705	L1	75.9 ± 0.7		59 ± 1		5544	57.2
#n = 9229	L2	92.0 ± 0.3	$\oplus$	86 ± 1	$\oplus$	92053	14.6
	FBE	76.1 ± 0.7		59 ± 2		223	43.1
UWCSE	L0	93.6 ± 2.3		39 ± 17		20	25.4
#p = 113	L1	94.0 ± 2.3		29 ± 14		163	31.6
#n = 2711	L2	94.3 ± 2.3		33 ± 13		8098	24.6
	FBE	95.0 ± 1.3		34 ± 19		74	45.2
Yeast	L0	87.7 ± 0.4	$\ominus$	68 ± 2		1479	82.0
#p = 1299	L1	88.0 ± 0.6		63 ± 2	$\ominus$	3630	65.8
#n = 5456	L2	88.0 ± 0.5	$\ominus$	62 ± 2	$\ominus$	436062	62.4
	FBE	88.8 ± 0.4		71 ± 1		3023	95.1
Carc	L0	62.1 ± 4.5		66 ± 4		23	15.0
#p = 182	L1	60.3 ± 4.1		67 ± 4		1843	32.4
#n = 148	L2	60.0 ± 3.4		64 ± 4		2183	17.5
	FBE	60.3 ± 4.3		67 ± 5		262	35.3
Bongard	L0	98.1 ± 0.4	$\ominus$	98 ± 1	$\ominus$	90	11.4
#p = 671	L1	99.6 ± 0.3		100 ± 0		215	9.9
#n = 864	L2	100.0 ± 0.0		100 ± 0		22637	5.0
	FBE	99.5 ± 0.3		100 ± 0		31	14.1

means of exhaustive search. The lookahead algorithm implemented in Tilde provides a challenging baseline for comparison because it employs query-pack execution [3], which has been shown to yield large gains in execution time in combination with lookahead.

We have implemented FBE in Tilde. To compute the conjunction for a node of the tree, we use greedy search with the FBE score to find a conjunction of at most two literals. Therefore, our results are comparable to depth one lookahead. The evaluation score of a literal is computed as the maximum of the information gain ratios [10] computed for its features (the latter are computed using ComputeCounts shown in Fig. 1).

We perform experiments on nine data sets: two versions of Mutagenesis (Muta [7], p. 344), Financial [1], Sisyphus task A and B [4], UWCSE [5], Yeast [5], Carcinogenesis (Carc [7], p. 345), and Bongard ([7], p. 136). Details of the data sets can be found in the listed references. We run Tilde with FBE and with exhaustive lookahead of depth 0 to 2. We estimate the predictive accuracy and area under the precision-recall curve (AUPRC) [6] of the obtained models using 10 fold stratified cross-validation for all data sets except UWCSE. For this data set, we use the 5 folds provided by the original authors.

Table 2 presents the results. Most results confirm our hypothesis. The results obtained with FBE have comparable accuracy and AUPRC to those with lookahead depth one (L1) and are never significantly worse. For six data sets the accuracy (AUPRC) of FBE is significantly better than that of L0. Note that for some data sets, Tilde fails to build a model without lookahead (cf. the Size column). The reason is that none of the evaluated clauses yields a non-zero gain in these cases. For the Sisyphus data sets, L2 performs significantly better than FBE. Note that L2 is more expressive (it allows two literals in each node). It is also 200-400 times slower on these data sets.

The FBE approach is always faster than L1 and L2. It is on average 7 times faster than L1 and 200 times faster than L2. Of course, our approach trades time for memory: it makes use of pre-computed tables. The memory required for storing these tables was, however, limited: the memory overhead over the space required for loading the system and the data was at most 12%.

4 Conclusions

Greedy machine learning algorithms and in particular Inductive Logic Programming (ILP) algorithms suffer from shortsightedness resulting in accuracy-wise suboptimal models. Lookahead helps greedy search overcome this shortcoming, but incurs an exponential increase in execution time. In this paper, we propose an alternative termed feature based evaluation (FBE). The idea behind feature based evaluation is to compute the score of a refinement based on a number of features that are defined for it. The particular instantiation of FBE that is considered in this paper can be seen as a restricted form of lookahead. In an experimental evaluation of the approach, we show that FBE yields models with an accuracy comparable to that of models built with lookahead and that FBE is considerably faster.

Other researchers have considered the problem of myopia in greedy ILP systems. Most approaches can be seen as a limited form of lookahead. These include determinate literals, template based lookahead, and macro-operators. Besides lookahead, beam-search has also been used. A comparison of systems implementing these different approaches appears in [9]. Skewing [11] also reduces myopia of greedy learners. Skewing is, however, less applicable to the type of myopia faced by relational learners, which occurs for non-discriminating literals that introduce useful new variables.

Interesting directions for further work include evaluating FBE in the context of a rule learner, investigating other evaluation scores based on FBE (e.g., the Euclidean distance mentioned in Section 2.1), and testing FBE with higher lookahead depths (e.g., to approximate depth two lookahead, one would add features consisting of two literals in the set of pre-computed tables described in Section 2.2).

Acknowledgments Jan Struyf is a postdoctoral fellow of the Fund for Scientific Research of Flanders (FWO-Vlaanderen). David Page is supported in part by the U.S. National Science Foundation grant IIS 0534908.

References

1. P. Berka. Guide to the financial data set. In *ECML/PKDD 2000 Discovery Challenge*, 2000.
2. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.
3. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
4. H. Blockeel and J. Struyf. Frankenstein classifiers: Some experiments on the Sisyphus data set. In *Workshop on Integration of Data Mining, Decision Support, and Meta-Learning (IDDM'01)*, 2001.
5. J. Davis, E. Burnside, I. C. Dutra, D. Page, and V. Santos Costa. An integrated approach to learning bayesian networks of rules. In *16th European Conference on Machine Learning*, pages 84–95, 2005.
6. J. Davis and M. Goadrich. The relationship between precision-recall and ROC curves. In *23nd International Conference on Machine Learning*, 2006. To appear.
7. S. Džeroski and N. Lavrač, editors. *Relational Data Mining*. Springer, 2001.
8. T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.
9. L. Peña Castillo and S. Wrobel. A comparative study on methods for reducing myopia of hill-climbing search in multirelational learning. In *21st International Conference on Machine Learning*, 2004.
10. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in Machine Learning. Morgan Kaufmann, 1993.
11. S. Ray and D. Page. Generalized skewing for functions with continuous and nominal attributes. In *22nd International Conference on Machine Learning*, pages 705–712, 2005.