

Learning the Common Structure of Data

Kristina Lerman and Steven Minton

University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90292
{lerman,minton}@isi.edu

Abstract

The proliferation of online information sources has accentuated the need for tools that automatically validate and recognize data. We present an efficient algorithm that learns structural information about data from positive examples alone. We describe two Web wrapper maintenance applications that employ this algorithm. The first application detects when a wrapper is not extracting correct data. The second application automatically identifies data on Web pages so that the wrapper may be re-induced when the source format changes.

Introduction

As the size and diversity of online information grows, many of the common transactions between humans and online information systems are being delegated to software agents (Jennings & Wooldridge 1998). Price comparison and stock monitoring are just two examples of the tasks that can be assigned to an agent. Many protocols and languages have been designed to facilitate information exchange, from primitive EDI protocols to more modern languages like XML and KQML. However, these schemes all require that the agents conform to syntactic and semantic standards that have been carefully worked out in advance. Humans, in contrast, get by with only informal conventions to facilitate communication, and rarely require detailed pre-specified standards for information exchange.

If we examine typical web pages – electronic catalogs, financial sites, *etc.* – we find that information is laid out in a variety of graphical arrangements, often with minimal use of natural language. People are able to understand web pages because they have expectations about the structure of the data appearing on the page and its physical layout. For instance, if we look at online Zagat's restaurant guide, we find a restaurant's name, an address, a review, *etc.* Though none of these are explicitly labeled as such, the page is immediately understandable because we expect this information to be on the page and have expectations about the appearance of these fields (*e.g.*, we know that a U.S.

address typically begins with a street address and ends with a zip code).

In this paper we describe a machine learning method for acquiring expectations about the content of data fields. The method learns structural information about data, which we may use to recognize restaurant names, addresses, *etc.* We describe two applications related to wrapper induction that utilize this information. A web page wrapper is a program that extracts data from a web page. Wrappers are used extensively by information integration programs (Knoblock *et al.* 1998). Some types of Web wrappers (Muslea *et al.* 1999, Kushmerick *et al.* 1997) rely primarily on the layout of the Web page to extract data. These wrappers are very efficient at extracting data; moreover, extraction rules can be learned from very few labeled examples. However, sometimes these wrappers stop extracting correctly when the page layout changes. We present a method for verifying that an existing wrapper is correctly extracting data. We also present a method for re-inducing a wrapper when the layout of the Web source changes. Though we focus on web applications, the learning technique is not web-specific, and can be used to learn about text and numeric fields in general. We believe that it is a step towards true agent interoperability, where agents can exchange and aggregate data without needing to know in advance about the detailed syntactic and semantic conventions used by their partners.

The Learning Task

Our objective is to learn the structure of data fields, such as addresses. One example of a street address is “4676 Admiralty Way.” In previous work, the structure of information extracted from Web pages was described by a sequence of characters (Goan *et al.* 1996) or a collection of global features, such as the number of words and the density of numeric characters (Kushmerick 1999). We propose an intermediate word-level representation that balances the descriptive power and specificity of the character-level representation with the compactness and computational efficiency of the global representation. Words, or more accurately tokens, are strings generated from an alphabet containing different types of characters: alphabetic, numeric and punctuation. We use the token's character types to assign it to one or more syntactic

categories: alphabetic, numeric, etc. These categories form a hierarchy depicted in Figure 1, where the arrows point from more general to less general categories. A unique token type is created for every string that appears in at least k examples, as determined in a preprocessing step. The hierarchical representation allows for multi-level generalization. Thus, the token “California” belongs to the general token types ALPHANUM (alphanumeric strings), ALPHA (alphabetic strings), CAPS (capitalized words), as well as to the specific type representing the string “California”. This representation is flexible and may be expanded to include domain specific information. For example, we may add range to the number category (small, large, 1-, 2-, and 3-digit numbers), or explicitly include knowledge about the type of information being parsed.

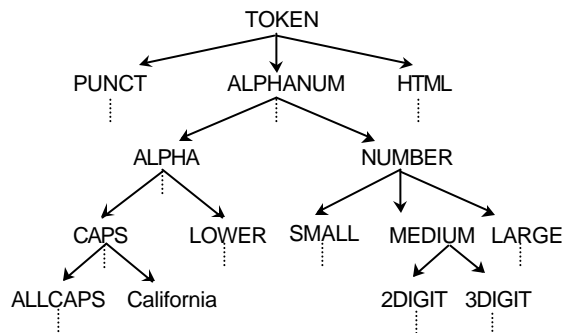


Figure – The token type syntactic hierarchy. Dots indicate the existence of additional types.

We claim that a sequence of specific and general tokens is more useful for describing common data fields than the finite state representations used in previous work (Carrasco & Oncina 1994, Goan *et al.* 1996). For complex fields, and for purposes of information extraction, it is sufficient to use only the starting and ending sequences as the description of the data field. For example, a set of street addresses – “4676 Admiralty Way”, “220 Lincoln Boulevard” and “998 South Robertson Boulevard” – start with a pattern “NUMBER CAPS” and end with “Boulevard” or “CAPS.” The starting and ending patterns together define a *data prototype* (cf. Datta & Kibler 1995). Note that the patterns in the data prototype are not regular expressions, since they don’t allow loops or recursion. We believe that recursive expressions tend to over-generalize and are not useful representations of the types of data we are trying to learn.

The problem of learning the data prototype from a set of examples that are labeled as belonging (or not) to a class, may be stated in one of two related ways: as a classification or as a conservation task. In the classification task, both positive and the negative instances of the class are used to learn a rule that will correctly classify new examples. Classification algorithms, like FOIL (Quinlan 1990), use negative examples to guide the specialization of the rule. They construct discriminating descriptions – those

that are satisfied by most of the positive examples and none of the negative examples. The conservation task, on the other hand, attempts to find a characteristic description (Dietterich & Michalski 1981) or conserved patterns (Brazma *et al.* 1995), in a set of positive examples of a class. Unlike the discriminating description, the characteristic description will often include redundant features. When negative examples are not available to the learning algorithm, redundant features may be necessary to correctly identify new instances of the class. The characteristic description learned from positive examples is the same as the discriminating description learned by the classification algorithm in the presence of negative examples drawn from infinitely many classes. While most of the widely used machine learning algorithms (decision trees and ILP) solve the classification task, fewer algorithms that learn characteristic descriptions are in use (Dietterich & Michalski 1981; Brazma, *et al.* 1995).

In our applications, an appropriate source of negative examples is problematic; therefore, we chose to frame the learning problem as a conservation task. We introduce DataPro – an algorithm that learns data prototypes from positive examples of the data field alone. DataPro finds statistically significant sequences of tokens, *i.e.*, those that describe many of the positive instances and are highly unlikely to have been generated by chance. DataPro is a statistical algorithm; therefore, it tolerates noisy data. The training examples are encoded in a prefix tree, where each node corresponds to a token type (specific or general category), and a path through the tree corresponds to a sequence of tokens. We present a greedy polynomial time version of the algorithm that relies on significance judgements to grow the tree and prune the nodes.

A sequence of tokens is significant if it occurs more frequently than would be expected if the tokens were generated randomly and independently of one another. We begin by estimating the baseline probability of a token’s occurrence from the proportion of all tokens in the examples for the data field that are of that type. Suppose we have already found a significant token sequence – *e.g.*, the pattern consisting of the single token “NUMBER” – in a set of street addresses such as the ones above, and want to determine whether the more specific pattern, “NUMBER CAPS”, is also a significant pattern. Knowing the probability of occurrence of type CAPS, we can compute how many times CAPS can be expected to follow NUMBER completely by chance. If we observe a considerably greater number of these sequences, we conclude that the longer pattern is significant.

Formally, we use hypothesis testing (Papoulis 1990) to decide whether a pattern is significant. The null hypothesis is that observed instances of this pattern were generated by chance, via the random, independent generation of the individual tokens. Hypothesis testing decides, at a given confidence level, whether the data supports rejecting the null hypothesis. Suppose n identical sequences have been generated by a random source. The probability that a token T (whose overall probability of occurrence is p) will be the

next token in k of these sequences has a binomial distribution. For a large n , the binomial distribution approaches a normal distribution $P(x, \mathbf{m}, \sigma)$ with $\mathbf{m} = np$ and $\sigma^2 = np(1-p)$. The cumulative probability is the probability of observing at least n_l events:

$$P(k \geq n_l) = \int_{n_l}^{\infty} P(x, \mathbf{m}, \sigma) dx$$

We use approximation formulas (Abramowitz & Stegun 1964) to compute the value of the integral.

The significance level of the test, α , is the probability that the null hypothesis is rejected even though it is true, and it is given by the cumulative probability above. Suppose we set $\alpha = 0.05$. This means that we expect to observe at least n_l events 5% of the time under the null hypothesis. If the number of observed events¹ is greater, we reject the null hypothesis (at the given significance level), *i.e.*, decide that the observation is significant.

The DataPro Algorithm

We now describe DataPro, the algorithm that finds statistically significant patterns in a set of token sequences. We focus the discussion on the version of the algorithm that learns starting patterns. The algorithm can be easily adapted to learn ending patterns.

During the preprocessing step the text is tokenized, and the tokens are assigned syntactic types (see Figure 1), as described previously. The algorithm builds a prefix tree, in which each node corresponds to a token whose position in the sequence is given by the node's depth in the tree. Every path through the tree starting at the root node is a significant pattern found by the algorithm.

The tree is grown incrementally. Adding a child to a node corresponds to extending the node's pattern by a single token. Thus, each child represents a different way to specialize the pattern. For example, when learning city names, the node corresponding to "New" might have three children, corresponding to the patterns "New Haven", "New York" and "New CAPS".

As explained previously, a child node is judged to be significant with respect to its parent node if the number of occurrences of the child pattern is sufficiently large, given the number of occurrences of the parent pattern and the baseline probability of the token used to extend the parent pattern. A pruning step insures that each child node is also significant given its more specific siblings. For example, if there are 10 occurrence of "New Haven", and 12 occurrences of "New York", and both of these are judged to be significant, then "New CAPS" will be retained only if

¹ Note that the hypothesis we test is derived from observation (data); therefore, we must subtract one from the number of observed events to take into account the reduction in the number of degrees of freedom resulting from an observation. This also prevents the anomalous case when a single occurrence of a rare event is judged to be significant.

there are significantly more than 22 examples that match "New CAPS".

Similarly, once the entire tree has been expanded, the algorithm includes a pattern extraction step that traverses the tree, checking whether the pattern "New" is still significant given the more specific patterns "New York", "New Haven" and "New CAPS". In other words, DataPro decides whether the examples described by "New" but not by any of the longer sequences can be explained by the null hypothesis.

We present the pseudocode of the DataPro algorithm below and describe it in greater detail.

```

DATAPRO MAIN LOOP
Create root node of tree;
For next node Q of tree
  Create children of Q;
  Prune generalizations;
  Determinize children;
Extract patterns from tree;

CREATE CHILDREN OF Q
For each token type T at next position in examples
  Let C = NewNode;
  Let C.token = T;
  Let C.examples = Q.examples that are followed by T;
  Let C.count = |C.examples|;
  Let C.pattern = concat(Q.pattern, T);
  If Significant(C.count, Q.count, T.probability,  $\alpha$ )
    AddChildToTree(C, Q)

PRUNE GENERALIZATIONS
For each child C of Q
  Let N = C.count -  $\sum_i (S_i.count \mid S_i \in \{\text{Siblings}(C)\} \ \&$ 
     $S_i.pattern \subset C.pattern)$ ;
  If Not(Significant(N, Q.count, C.token.probability,  $\alpha$ ))
    Delete C;

DETERMINIZE CHILDREN OF Q
For each child C of Q
  For each sibling S of C
    If S.pattern  $\subset$  C.pattern
      Delete S.examples from C.examples
      C.count = |C.examples|

EXTRACT PATTERNS FROM TREE
Create empty list;
For every node Q of tree
  For every child C of Q
    Let N = C.count -  $\sum_i (S_i.count \mid S_i \in \{\text{Children}(C)\})$ 
    If Significant(N, Q.count, C.token.probability,  $\alpha$ )
      Add C.pattern to the list;
Return ( list of patterns );

```

DataPro grows the prefix tree greedily by finding specializations of the significant patterns and pruning generalizations. The tree is empty initially, and children are added to the root node. The children represent all tokens that occur in the first position in the training examples more often than expected by chance. The tree is extended incrementally at node Q. A new child is added to Q for

every significant specialization of the pattern ending at Q .

In the pruning step, the algorithm checks the children of Q to determine whether the generalizations are significant given the specific patterns. In our notation, $A \subset B$ means that B is a generalization of A . Next, the examples that are explained by any significant patterns are deleted from its sibling generalizations.² We refer to this as *determinizing* the tree, because it insures that every sequence of tokens in the training data is used as evidence for at most one pattern in the tree.

The algorithm can in principle be implemented more efficiently than in the pseudocode. The algorithm needs to examine each prefix (a subsequence with which the examples begin) a constant number of times. Given N training examples, the longest one containing L tokens, there are $O(NL)$ prefixes. The size of the tree is, therefore, also $O(NL)$. Pruning and determinization can be folded into node creation, and will not affect the complexity of the algorithm. Note that the algorithm is efficient because we determinize the tree; otherwise, the tree grows exponentially. The complexity of the last step, extracting patterns from tree, is also $O(NL)$.

Application 1: Wrapper Verification

A common problem experienced by information integration applications is wrapper fragility. A Web wrapper is a program that takes a query, retrieves a page from a Web source using the query, and extracts results from it. Many wrappers use layout of HTML pages to extract the data. Therefore, they are vulnerable to changes in the layout, which occur when the site is redesigned. In some cases the wrapper continues to extract, but the data is no longer correct. Wrapper maintenance, *i.e.*, detecting when the wrapper stops working and automatically recovering from failure, is an important problem for information integration research.

Few researchers have addressed the problem of wrapper maintenance. Kushmerick (1999) proposed RAPTURE to verify that a wrapper correctly extracts data from a Web page. Each data field is described by a collection of global numeric features, such as word count, average word length, HTML tag density, etc. Given a set of queries for which the wrapper output is known, RAPTURE checks that the wrapper generates a new result with the expected combination of feature values for each of the queries. Kushmerick found that the HTML tag density feature alone could correctly identify almost all of the changes in the sources he monitored. In Kushmerick's experiment, the addition of other features to the probability calculation about the wrapper's correctness significantly reduced the algorithm's performance.

The prototypes learned by DataPro lend themselves to

² We have experimented with different procedures and found it useful to not prune the children of the root node, because early pruning may appreciably reduce the number of examples available to the algorithm.

the data validation task and, specifically, to wrapper verification. A set of queries is used to retrieve HTML pages from which the wrapper extracts N training examples. DataPro learns m patterns that describe the common beginnings and endings of each field of the extracts. In the verification phase, the wrapper generates n test examples from pages retrieved using the same or similar set of queries. Suppose t_i training examples and k_i test examples match the i th pattern. If the two distributions, \mathbf{k} and \mathbf{t} , are the same (at some significance level), the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed. Our approach allows us to add other features to the two distributions. In the experiments described below we added the average number of tuples-per-page feature. Goodness of fit method (Papoulis 1990) was used to decide whether the two distributions are the same.

We monitored 27 wrappers (representing 23 distinct Web sources) over a period of several months. For each wrapper, the results of 15-30 queries were stored periodically. All new results were compared with the last correct wrapper output (training examples). The verification algorithm used DataPro to learn the starting and ending patterns for each field of the training examples and made a decision at a high significance level (corresponding to $\alpha = 0.001$) about whether the patterns had the same distribution over the new examples.

Manually checking the results identified 37 wrapper changes³ out of the total 443 comparisons. Of these 30 were attributed to changes in the source layout and data format, and 7 to changes internal to the wrappers. The verification algorithm correctly discovered 35 of these changes, including the 17 that would have been missed by RAPTURE if it were relying solely on HTML tag density feature. The algorithm incorrectly decided that the wrapper has changed in 40 cases. We are currently working to reduce the high rate of false positives.

Application 2: Wrapper Maintenance

If the wrapper stops extracting correctly, the next challenge is to rebuild it automatically (Cohen 1999). The extraction rules for our Web wrappers (Muslea *et al.* 1999), as well as

³ We have found that three effects may change the output of the wrapper. The most important of these is the change in the layout of the pages returned by the Web source. Because our wrappers use positional information to extract data, any alterations in the layout tend to break the wrapper. Changes may also occur in the format of the source data itself: *e.g.*, when street number is dropped from the address field ("Main St" instead of "25 Main St"), or book availability changes from "Ships immediately" to "In Stock: ships immediately". It is important to know when these types of changes occur, because information integration applications may be sensitive to data format. Finally, the wrapper code itself may change in a way that affects the wrapper output, *e.g.* to extract "10.00" instead of "\$10.00" for the price. The verification algorithm will also pick up these changes.

many others (*cf.* Kushmerick *et al.* 1997), are generated by machine learning algorithms, which take as input several pages from the same source and examples of data to extract for each page. Therefore, if we identify examples of data on pages for which the wrapper fails, we can use these examples as input to the induction algorithm to generate new extraction rules.

We propose a method that takes a set of training examples and a set of pages from the same source, and uses a mixture of supervised and unsupervised learning techniques to identify examples of the data field on new pages. Due to paper length limitations, we present an outline of the algorithm, leaving the details to future publication. We assume that the format of data did not change. First, DataPro learns the patterns that describe the start and end of each data field in the training examples. We also calculate the mean and variance of the number-of-tokens distribution. Next, each new page is scanned to identify all text segments that begin with one of the starting patterns and end with one of the ending patterns. These segments, which we call candidates, include examples of the field we want to extract from the new pages. The candidates containing significantly more or fewer tokens than expected based on the number-of-tokens distribution are eliminated from the set, often still leaving hundreds of candidates (we allow up to 300) on each page. The candidates are then clustered to identify subsets that share common features. The features used to describe each candidate are where on the page it occurs, adjacent tokens, and whether it is visible to the user. Each cluster is then given a score based on how similar it is to the training examples. We expect the highest ranked clusters to contain the correct examples of the data field.

We evaluate the algorithm by using it to extract data from Web pages for which correct output is known. The pages were retrieved using the same (or similar) queries that were used to obtain training examples. The output of the extraction algorithm is a ranked list of clusters for every data field being extracted. Each cluster is checked manually, and it is judged to be correct if it contains only the complete instances of the field, which appear in the correct context on the page. For example, if extracting a city of an address, we only want to those city names that are part of an address.

We ran the extraction algorithm for 21 distinct Web sources, attempting to extract 77 data fields from all the sources. In 62 cases the top ranked cluster contained correct complete instances of the data field. In eight cases the correct cluster was ranked lower, while in six cases no candidates were identified on the pages. All except one cluster contained only the correct examples of the field.

The table on the right presents extraction results for six Web sources. The first column lists the source's name and the number of pages provided to the extraction algorithm. The second column lists the data field extracted from the pages. The table also lists the total number of candidates from all pages identified by the data prototypes and the number of the resulting clusters. The last three columns

give the rank of the cluster containing correct examples of the field, the number of examples in that cluster and the percent of the examples that are correct and complete. In most cases, the extraction algorithm correctly identifies examples of data fields (rank=1, 100% correct) from several pages. This indicates that our algorithm may be combined with the wrapper induction algorithm to re-induce extraction rules for the correctly identified fields. We note that for three of the sources – *altavista*, *amazon* and *quote* – the existing wrappers failed to produce correct output for the pages used in the extraction task, because the layout of the pages had changed since the extraction rules were created.

source	data field	#candid	#clusters	rank	# in cluster	%correct
<i>altavista</i> 20 pages	TITLE	896	115	3	5	100
	LINK	570	32	1	14	100
<i>amazon</i> 20 pages	AUTHOR	1950	208	2	20	100
	TITLE	1030	117	2	8	100
	PRICE	49	3	1	15	100
	ISBN	20	2	1	18	100
	AVAILAB.	18	1	1	18	100
<i>arrow</i> 21pages	PARTNUM	1447	69	1	6	100
	MANUFAC	587	25	1	58	100
	PRICE	201	2	1	58	100
	DESCRIP.	718	49	1	58	100
	STATUS	818	45	1	58	100
	PARTURL	0				
<i>bigbook</i> 16 pages	NAME	192	26	1	14	100
	STREET	18	2	1	16	100
	CITY	25	4	1	9	100
	STATE	64	7	1	5	100
	PHONE	21	1	1	16	100
<i>quote</i> 20 pages	PRICECH.	45	5	0	0	0
	TICKER	29	5	1	11	100
	VOLUME	11	1	1	11	100
	PRICE	0				
<i>showtimes</i> 15 pages	MOVIE	82	10	1	15	100
	TIMES	238	13	1	15	100

Table – Results of automatic data extraction for several Web sources.

Related Research

Several researchers have addressed the problem of learning the structure of data. Grammar induction algorithms, for example, learn the common structure of a set of strings. Carrasco and Oncina (1994) propose ALERGIA, a stochastic grammar induction algorithm that learns a regular language given the strings belonging to the language. ALERGIA starts with a finite state automaton (FSA) that is initialized to be a prefix tree that represents all the strings of the language. The FSA is generalized by merging pairs of statistically similar subtrees. We found that ALERGIA tends to merge too many states, even at a high confidence limit, leading to an over-general grammar. The resulting automaton frequently has loops in it, corresponding to regular expressions like $a(b^*)c$. Real data is seldom described by such repeated structures.

Goan *et al.* (1996) proposed modifications to ALERGIA aimed at reducing the number of bad merges. They also introduced syntactic categories similar to ours. Each symbol can belong to one of these categories. Goan *et al.* added a new generalization step in which the transitions corresponding to symbols of the same category that are approximately evenly distributed over the range of that category (*e.g.*, 0-9 for numerals) are replaced with a single transition. Though the proposed modifications make the grammar induction algorithm more robust, the final FSA is still sensitive to the merge order. Moreover, it does not allow for multi-level generalization that we have found useful. The algorithm requires dozens, if not hundreds, of examples in order to learn the correct grammar.

A sequence of n tokens can be viewed as a non-recursive n -ary predicate; therefore, we have used FOIL (Quinlan 1990) to learn the data prototypes. FOIL learns first order predicate logic clauses defining a class from a set of positive and negative examples of the class. FOIL finds a discriminating description that covers many positive and none of the negative examples.

We have used foil.6 with no-negative-literals option to learn data prototypes for several data fields. In all cases the closed world assumption was used to construct negative examples from the known objects: thus, names and addresses were the negative examples for the phone number class for the *whitepages* source. In most cases there were many similarities between the clauses learned by FOIL and the patterns learned by DataPro; however, FOIL clauses tended to be overly general. Thus, FOIL learned that '(' was a good description of phone numbers for the *whitepages* source. However, '(' will not be sufficient to recognize phone numbers on a random page. We attribute over-generalization to the incompleteness of the set of negative examples presented to FOIL. Another problem was when given examples of a class with little structure, such as names and book titles, FOIL tended to create clauses that covered single examples, or failed to find any clauses.

Conclusion

In summary, we have presented an efficient algorithm that can be used to learn structural information about a data field from a set of examples. The algorithm promises to be useful in Web wrapper maintenance applications: both for detecting when a wrapper stops extracting correctly and for identifying new examples of the data field in order to rebuild the wrapper.

An important aspect of our work is that we focus on generalizing token sequences according to a type hierarchy. Most previous work in the area has focused on generalizations that capture repeated patterns in a sequence (*e.g.*, learning regular expressions). Though this direction has not yet attracted much attention, we believe that it will prove useful for a wide variety of data validation tasks.

One of the most exciting directions for future work is using the DataPro algorithm for *cross-site* extraction, *i.e.*,

learning the author, title and price fields for the Amazon source, and using them to extract the same fields on the Barnes&Noble source. Preliminary results show that this is feasible. A second direction is to use the DataPro approach to learn characteristic patterns within data fields, as opposed to just the starting and ending patterns. The algorithm is efficient enough to be used this way, and there are many applications where this would be useful.

Acknowledgements

This work was supported in part by the Rome Labs and DARPA contract F30602-98-2-0109, by AFOSR contract F49620-98-1-0046, and by the NSF under grant IRI-9610014.

References

- Abramowitz, M., and Stegun, I. A. 1964. *Handbook of mathematical functions*. Washington, D.C.: National Bureau of Standards.
- Brazma, A.; Jonassen, I.; Eidhammer, I.; and Gilbert, D. 1995. Approaches to the automatic discovery of patterns in biosequences. Technical report, Dept. of Informatics, University of Bergen.
- Carrasco, R. C., and Oncina, J. 1994. Learning stochastic regular grammars by means of a state merging method. *Lecture Notes in Computer Science*, 862.
- Cohen, W. W. 1999. Recognizing Structure in Web Pages using Similarity Queries. In *Proc. of AAAI*.
- Datta, P.; and Kibler, D. 1995. Learning prototypical concept descriptions. In *Proc. 12th International Conference on Machine Learning*.
- Dietterich, T.; and Michalski, R. 1981. Inductive learning ,16:257-

Goatn 75ia