Learning Classifiers from a Relational Database of Tutor Logs

BAO HONG TAN
Advisor: Prof. Jack Mostow, Research Professor, Robotics Institute
Co-Advisor: José González-Brenes, Ph.D., Language Technologies

A bottleneck in mining tutor data is mapping heterogeneous event streams to feature vectors with which to train and test classifiers. To bypass the labor-intensive process of feature engineering, AutoCord learns classifiers directly from a relational database of events logged by a tutor. It searches through a space of classifiers represented as database queries, using a small set of heuristic operators. We show how AutoCord learns a classifier to predict whether a child will finish reading a story in Project LISTEN's Reading Tutor. We compare it to a previously reported classifier that uses hand-engineered features. AutoCord has the potential to learn classifiers with less effort and greater accuracy.

1. INTRODUCTION

Intelligent tutors' interactions with students consist of streams of tutorial events. Mining such data typically involves translating it into tables of feature vectors amenable to statistical analysis and classifier learning [Mostow and Beck, 2006]. The process of devising suitable features for this purpose is called feature engineering. Designing good features can require considerable knowledge of the domain, familiarity with the tutor, and effort. For example, performing manual feature engineering in a previous classification task [González-Brenes and Mostow, 2010] took approximately two months.

This paper presents AutoCord (Automatic Classifier Of Relational Data), an implemented method that bypasses the labor-intensive process of feature engineering by training classifiers directly on a relational database of events logged by a tutor. We illustrate AutoCord on data logged by Project LISTEN's Reading Tutor, which listens to children read stories aloud, responds with spoken and graphical feedback [Mostow and Aist, 1999], and helps them learn to read [see, e.g., Mostow et al., 2003]. To illustrate AutoCord, we train a classifier to perform a previously published task [González-Brenes and Mostow, 2010]: predict whether a child who is reading a story will finish it.

The rest of the paper is organized as follows. Section 2 describes how we represent event patterns. Section 3 explains how AutoCord discovers classifiers. Section 4 evaluates AutoCord. Section 5 relates AutoCord to prior work. Section 6 concludes.

2. REPRESENTATION OF EVENTS, CONTEXTS, AND PATTERNS

We now summarize how we log, display, generalize, and constrain Reading Tutor events.

2.1 The structure of data logged by the Reading Tutor

The events logged by the Reading Tutor vary in grain size. As Figure 1 illustrates, logged events range all the way from an entire run of the program, to a student session, to reading a story, to encountering a sentence, to producing an utterance, down to individual spoken words and mouse clicks. Figure 1 shows a screenshot of the Session Browser, which displays logged Reading Tutor data in human-readable, interactively expandable form.

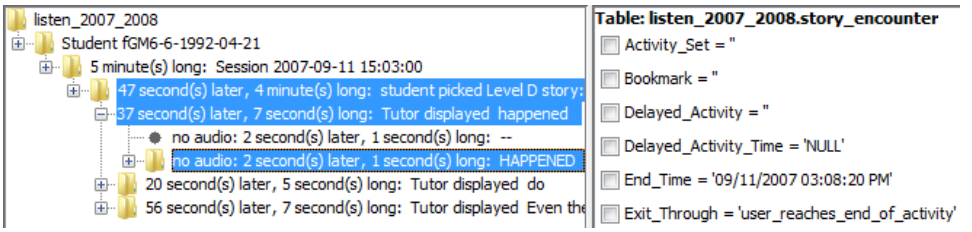


Figure 1: Session Browser's partially expanded event tree (left); list of fields (right).

Figure 1 displays a story encounter in the temporal hierarchy of the session in which it occurred. Each line summarizes the database record for an event. For example, the highlighted story encounter "…student picked Level D story…" is represented as a row in the story_encounter table, with the field names and values listed on the right side of Figure 1. For example, the Exit_through field shows how the story encounter ended, and its value user_reaches_end_of_activity indicates that the student finished the story, so the story encounter is a positive example of story completion. All other values indicate different outcomes, such as clicking *Back* or *Goodbye*, timing out, or crashing.

The fields User_ID, Machine_Name, Start_time, and Sms are common to all types of events, including story encounters and sentence encounters. As their names suggest, they respectively identify the student and computer involved in the event, and when it started, with the milliseconds portion in its own field. Events with non-zero durations have corresponding End_time and Ems fields.

Here the user has partially expanded the tree of events by clicking on some of them. The structure of the tree indicates parental and fraternal temporal relations among events. A child event is defined as starting during its parent event; siblings share the same parent. The indentation level of each event reflects these relations. For instance, the highlighted story encounter is a child of the session directly above it, and is therefore indented further. Its own children are the sentence encounters below it, displayed at the same indentation level because they are siblings.

2.2 Inferring a pattern from a set of related events

In Figure 1, the user has selected the subset of highlighted events by clicking on them. Given such a constellation of related events, the Session Browser's AutoJoin operator [Mostow and Tan, 2010] abstracts it into a general pattern of which it is an instance. To infer a pattern from a single instance, AutoJoin heuristically assumes that repetition of a constant that is unlikely to recur by coincidence, such as a user ID, is a requirement of the pattern. It represents the inferred pattern as a MySQL query [MySQL, 2004] to retrieve instances of it – in this example, the last utterance (se.End_Time = u.End_Time) of the first sentence in a story (st.Start_Time = se.Start_Time):

```
SELECT * FROM
        utterance u,
        story_encounter st,
        sentence_encounter se
WHERE
        (st.Machine_Name = se.Machine_Name) AND
        (st.Start_Time = se.Story_Encounter_Start_Time) AND
        (st.User_ID = se.User_ID) AND
```

```
(se.End_Time = u.End_Time) AND
(st.Machine_Name = u.Machine_Name) AND
(se.sms = u.Sentence_Encounter_sms) AND
(st.Start_Time = se.Start_Time) AND
(st.Start_Time = se.Step_Start_Time) AND
(st.Start_Time = se.Story_Encounter_Start_Time) AND
(st.Start_Time = u.Sentence_Encounter_Start_Time) AND
(st.User_ID = u.User_ID)
```

Given a target concept such as "stories the student will finish reading," AutoCord searches for queries that maximize the number of positive instances retrieved and minimize the number of negative instances.

2.3 Operationality criteria

In addition, the query must satisfy operationality criteria [Mostow, 1983] that constrain the information used in the query. These constraints vary in form and purpose.

One type of operationality constraint limits the query to information available at the point in time where the classifier will be used. Although we use a story encounter's Exit_through field to label it as a positive or negative training example, the Reading Tutor does not log this information until the story encounter ends, so the trained classifier cannot use it to help predict whether a child will finish a story. As Yogi Berra famously said, "It's hard to make predictions, especially about the future." More subtly, if we want to use the trained classifier at any random time, we should train it on data representative of what will be available then. To simulate such data, we choose a percentage p between 0 and 1. To exclude any events occurring after that percentage of the way through the story encounter, we add a clause such as the following to the query:

```
… AND   (se.Start_Time <= ADDTIME( st.Start_Time, p* [length] ))
Where [length] is UNIX_TIMESTAMP(st.End_time) – UNIX_TIMESTAMP(st.Start_time)
```

Another way to simulate such data, is to impose an absolute time limit. For example, if the limit is 10 seconds, then during the training process, the search algorithm can only look at events occurring within the first 10 seconds since the start of the story encounter.

Operationality criteria may also restrict what sort of classifier is useful to learn. For instance, in order to apply to future data, the trained classifier should not refer to any specific student or computer. We enforce this constraint by excluding user IDs and machine names from the query. Similarly, if we want the classifier to predict story completion based solely on the student's observed behavior rather than traits such as age or gender, the query must not include those fields.

Finally, operationality criteria may pertain to the protocol for training and testing the classifier. Even if we preclude the trained classifier from mentioning specific students, it may still implicitly exploit information about them, improving classification performance on the training set – and inflating performance on a test set that includes the same students. To ensure that the training and test sets have no students in common, the queries that generate them include mutually exclusive constraints on the user_id, e.g.:

```
(st.User_ID <= 'mDS8-8-1998-09-22')  /* Use training set */
```

or

```
(st.User_ID > 'mDS8-8-1998-09-22')  /* Use test set */
```

Although these clauses mention a specific user_ID, despite the constraint against doing so, we do not consider them part of the learned classifier itself, just a way to split the data into training and test sets. We could use a more complex constraint to implement a more sophisticated split, e.g. to stratify by gender, which is encoded by the first character of the user_ID.

3. APPROACH

We formulate AutoCord as a heuristic search through a space of classifiers represented as database queries. Section 3.1 outlines the overall search algorithm. Section 3.2 describes how AutoCord samples the training data. Section 3.3 describes the search operators.

3.1 Search Algorithm

AutoCord searches through a space of classifiers by hill climbing on their accuracy. In the pseudo-code below, step 1 starts with a query to retrieve the entire training or test set. For the task of predicting story completion, we start with this initial query:

```
SELECT * FROM story_encounter st
WHERE (st.User_ID <= 'mDS8-8-1998-09-22')  /* Use training set */
```

Other classification problems would require a different initial query.

Pseudo-code for AutoCord(initial query)

1. Q ← initial query
2. S ← empty set; K_{best} ← 0
3. R ← table of results (examples) retrieved by executing query Q, limited to a sample
4. For each operator Op:
5. Q' ← Op(Q, R)
6. R' ← table of results retrieved by Q'
7. K ← Score(R')
8. Add tuple (Q', K) to S

   End For
9. Pick (Q_{high}, K_{high}) from S that maximizes K_{high}
10. If $K_{high} < K_{best}$ + epsilon, return Q_{high}
11. Q ← Q_{high}; K_{best} ← Max(K_{best}, K_{high})
12. Go to step 3.

Steps 3 through 13 specify an iterative process. Step 3 retrieves a table of results R from the database by executing the current query Q on a data sample to be described shortly. Next, the loop starting at step 4 applies each of AutoCord's operators. Based on the result set R, each operator adds one or more constraints to the input query Q to generate a new query Q'. Step 6 executes the new query Q' to get a new table of results R'. Step 7 scores classification accuracy on R as the number of positive examples minus the number of negative examples. Step 8 records query Q' and its score K. Step 9 chooses the highest-scoring query so far for the next iteration of the iterative step. The higher this score, the larger the number of positive examples the query retrieves, and the smaller the number of negative examples. We score queries by the difference of these numbers rather than their ratio in order to reward recall as well as precision. Unless the query enlarges this difference by more than epsilon (currently 2), step 10 stops and returns it. Otherwise search continues from the best query found so far.

The query can be applied as a classifier when a child is reading a story. Events that occurred up to the point in time the query is applied form a partial event tree which could be used to check against the constraints specified in the query. If all of the constraints are satisfied, then the label for the current story will be positive. However, even if one or more of the constraints is not satisfied, we cannot classify the current story as being negative, since the trained query is representative only of positive examples. To predict whether the child will quit, we need to first train a query representative of negative examples. This can be done by re-interpreting the value of the Exit_through field of the story encounters in the training data. When we consider story completion as being positive, we interpret a value of user_reaches_end_of_activity as a positive label, and all other values as negative labels. If we now consider quitting as being positive instead, we could interpret the value user_reaches_end_of_activity as a negative label. In this way, we could train queries representative of quitting.

3.2 Sampling

There is a need to select only a small subset of the table of results retrieved by a query since the number of results may be too large for analysis. There are two immediate benefits: the first is the operators can do their analysis in a smaller amount of time and the second is minimizing the risk of overfitting of the training data. We present a sampling method that exploits one of the story_encounter table's indexes. A table's index allows the database system to efficiently retrieve existing data and to insert new data. The story_encounter table has an index defined on the User_ID column so that rows of the table can be efficiently retrieved by any query that contains a constraint imposed on the User_ID column.

We illustrate the sampling method using the following example. Let Q be the current query being considered by the search algorithm:

```
SELECT * FROM story_encounter st, [other tables]
WHERE st.User_ID = 'mDS8-8-1998-09-22'
AND [other constraints]
```

The sampling method will add a LIMIT clause to Q so that it becomes Q(offset, count):

```
SELECT * FROM story_encounter st, [other tables]
WHERE st.User_ID = 'mDS8-8-1998-09-22'
AND [other constraints]
LIMIT [offset], [count]
```

Now Q is a function that accepts two parameters, offset and count, and substitutes for them in the actual query. Note that the LIMIT clause starts retrieving a specified number of rows at a specified zero-based offset. [other constraints] refer to the constraints generated by the operators.

Let $U(Q_1,Q_2,\ldots,Q_n) = (Q_1)$ UNION ALL (Q_2) UNION ALL … UNION ALL (Q_n)

Whenever the current query in the search algorithm is executed, this sampling method will be applied to retrieve the table of results. We assign to each Q_i a randomized offset

and count, and apply the union function U to all Q_i's to get a combined query which can be executed to get a table of sampled results.

Next, we describe AutoCord's operators. To illustrate them, we do a walkthrough of the search algorithm, starting from the following initial query:

```
SELECT * FROM story_encounter st     /* st is alias for story_encounter */
WHERE (st.User_ID <= 'mDS8-8-1998-09-22')  /* Use training set */
```

3.3 Contrast Operator

The Contrast operator generates a single constraint that will best distinguish positive from negative examples. The constraint generated is based on a 'split' in the occurrence of values for a field. For example, if all positive examples have values for a particular field all lesser than 5, and if all negative examples have values for that same field greater than 5, then 5 is the split value that will best separate positive from negative examples.

To find the field that can provide the best split, it calculates the frequencies of values for each column of the results table retrieved by the initial query. Two sets of frequencies are maintained: one for positive examples and another for negative examples. Consider the following results table for a clearer illustration:

Row #	New_Word_Count	Initiative	Student_Level	…
1	4	Student	A	
2	6	Student	C	
3	7	Student	C	

Figure 2: An example of a table of results retrieved

All the fields come from the story_encounter table, and each row represent a story encounter. The rest of the fields are not shown for brevity purposes. Assume the first two rows are positive examples, while the third is a negative example.

The calculated frequencies are as follow:

	New_Word_Count	Initiative	Student_Level	….
Positive eg.	4: once, 6: once	Student: twice	A: once, C: once	
Negative eg.	7: once	Student: once	C: once	

In this case, the Contrast operator finds that the best split occurs in the New_Word_Count field, with a split value of 6. Thus the new constraint to be added is st.New_Word_Count <= '6' since only the positive examples satisfy this constraint. However, in general, when it is not possible to find a perfect split, the operator will decide on the one that separates as many positive examples as possible from the negative examples.

Besides the <= mathematical relation, the Contrast operator also considers the following relations: =, !=, <=, >=, <, >, namely, equals, not equals, lesser than or equals to, greater than or equals to, lesser than, and greater than, respectively.

3.4 Extend Operator

The Extend operator essentially captures the relational structure of positive examples. To achieve that, it first picks a random positive example (which is a row) from the results

table. Next it randomly picks an event in the chosen row. For that event, it will then either pick a random child, sibling, or parent event. With the existing events in the input query and the newly picked event, the Extend operator will then apply AutoJoin and add the generated constraints to the input query.

We illustrate the Extend operator using the initial query shown at the beginning of the subsection as input. Suppose the operator picked a random sentence_encounter event, which is a child of the story_encounter. After performing AutoJoin on both events, the resulting query might be:

```
SELECT * FROM story_encounter st, sentence_encounter se
WHERE
        (st.User_ID <= 'mDS8-8-1998-09-22') /* Use training set */
AND     (st.Machine_Name = se.Machine_Name) /* Added by Extend operator */
AND     (st.User_ID = se.User_ID)
AND     (st.Start_Time = se.Story_Encounter_Start_Time)
```

Notice the last three constraints are added by AutoJoin. Both events have the same values for the fields Machine_Name, User_ID and Start_Time. In fact, these three fields identify the parent-child relationship.

3.5 Aggregate Operator

The Aggregate operator generates additional pseudo-fields for the Contrast operator to work on. It does so by picking from the results table an event type, such as story_encounter, and applying each of the aggregate functions to every field of the event's children. The aggregation of fields then result in additional pseudo-fields such as AVG(child.Word_Count), which is the average word count for the selected event's children. Every event of the selected type will then have these pseudo-fields alongside the existing fields. Currently, only the MIN, MAX, AVG, SUM, COUNT and STDDEV aggregate functions are supported, and they only work on numeral values, except for COUNT, which simply counts the number of rows it aggregates over.

The following query illustrates how the aggregated fields of a story_encounter event's children are calculated:

```
SELECT AVG(se.Word_Count), SUM(se.Word_Count), …
FROM sentence_encounter se
WHERE  (st.User_ID <= 'mDS8-8-1998-09-22') /* Use training set */
AND     (st.User_ID = se.User_ID)
AND     (st.Machine_Name = se.Machine_Name)
AND     (st.Start_Time = se.Story_Encounter_Start_Time)
```

The "st" fields represent values of the respective fields of the story_encounter event. Note that the above query is simply for illustration purpose; it calculates the aggregated fields of a single story_encounter event's children, but in the implementation, a more efficient query is applied to the results table to calculate aggregated fields of all story_encounter events' children at once.

To impose a constraint on a pseudo-field, a sub-query is used:

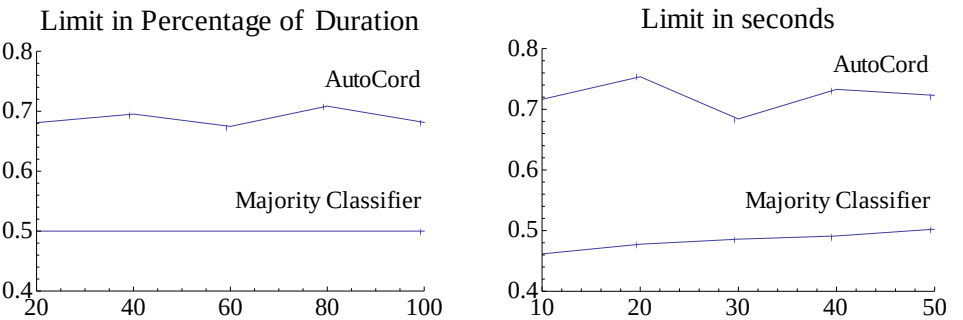
```
SELECT * FROM story_encounter st
WHERE  (st.User_ID <= 'mDS8-8-1998-09-22') /* Use training set */
AND      ((SELECT AVG(se.Word_Count)
                  FROM sentence_encounter se
                  WHERE  (st.User_ID = se.User_ID)
                  AND      (st.Machine_Name = se.Machine_Name)
                  AND      (st.Start_Time = se.Story_Encounter_Start_Time))
         > 5)
```

Notice that the sub-query is very similar to the query that calculates the aggregated fields of a story_encounter event's children. It is modified so that only the average word count is calculated and the constraint stipulates that the calculated value must be greater than 5. Currently, it is inefficient to represent a pseudo-field in a constraint using a sub-query and therefore we leave out the Aggregate operator when evaluating AutoCord. We intend to improve it in the future.

4. EVALUATION

As discussed previously in Section 2.3, there are two ways to restrict the amount of information the search algorithm can look at. The first is to restrict the algorithm to a certain percentage of the duration of the story encounter; the second is to impose an absolute time limit. In this way, the algorithm can only learn from events available from the start of the story encounter up to the limit on time imposed. In other words, the algorithm cannot "peek into the future" of a story encounter. Imposing a time limit also provides a means to test the classifier's ability to predict the outcome of a story encounter at various points in time before the story ends.

We modified the search algorithm slightly to incorporate the two ways of restricting amount of information available. Specifically, the Extend operator is modified so that whenever a new event is to be added to the current query, the new event's start time will be constrained appropriately, either by using a percentage or an absolute time limit. We ran the modified search algorithm with various values of percentages and time limits, so that for each run, we got a query that is suitable to be applied at that limit specified. To get the classification accuracy, we executed all the queries generated by the algorithm, one by one, and for each one, calculated the ratio of the number of positive examples to total number of examples. We did not apply the sampling method during the evaluation process as we wanted the results to be as accurate as possible. The results of the classification accuracy for each value of limit are shown below, and compared to a majority classifier.



The x-axis for the graph on the left shows the limit specified as 20%, 40%, … or 100% of the story's duration. The query generated by the search algorithm is suitable only for classifying positive examples. To match this fact, the majority classifier always output a positive label. Since the percentage limit is relative to a story's length, the generated query can be applied to a story of any length. The average accuracy across all values of the limit is 0.69 while the majority classifier's accuracy for the test data, which consisted of stories of any length, is 0.5.

The graph on the right shows the limit specified as 10, 20, .., or 50 seconds for each run of the algorithm. An interval of 10 seconds is used since it is the average duration of a sentence encounter, and so each interval roughly corresponds to a sentence encounter. Again, the query generated by the search algorithm is suitable only for classifying positive examples but in this case, it can only be applied to a story that has a duration of at least the limit with which the query was trained. The majority classifier always outputs a positive label, and its accuracy for a limit of X seconds is simply the ratio of number of positive examples to the total number of examples, with the test data consisting of stories that lasted at least X seconds. AutoCord yields a higher accuracy of 0.72 on average across all values of the limit. This average is comparable to that found by using $\ell 1$-regularized logistic regression in a prior work, which gives an asymptotic ratio of 0.78 as the number of examples increases [González-Brenes and Mostow, 2010].

The shape of the graphs could be attributed to the nondeterministic nature of the algorithm. Also, limiting by using absolute time shows a slightly higher accuracy than limiting by percentage. We believe this is due to the fact that it is fairer to only consider events that fit in a single fix-sized window for each story encounter if the limit is based on absolute time. However, if the limit is percentage based, the window size for each story encounter is proportional to the story's duration instead. Hence the extra "future" information gained through a larger window for a long story encounter might have negatively influenced the classification accuracy for shorter story encounters.

5. RELATION TO PRIOR WORK

Mining relational data sits at the intersection of Machine Learning with classical Artificial Intelligence methods that rely on formal logic, an area called Inductive Logic Programming (ILP). Notable examples of ILP algorithms that learn from data expressed as relations using formal logic representations include FOIL [Quinlan, 1990] and Progol [Muggleton, 1995]. Like FOIL, AutoCord inputs positive and negative examples in relational format, and hill climbs to distinguish between classes. FOIL uses negation and conjunction operators and outputs Horn clauses. AutoCord's operators assume that the relations describe events, work on SQL queries directly, and outputs SQL queries.

ILP methods can sometimes achieve high classification accuracy [Cohen, 1995], but are sensitive to noise [Brunk and Pazzani, 1991], and fail to scale to real-life database systems having a large number of relations [Yin et al., 2006]. In contrast, AutoCord is designed to operate directly on large event databases.

Provost and Kolluri [1999] reviewed literature on how to scale ILP approaches. They suggested that integrating data mining with relational databases might take advantage of the storage efficiencies of relational representations and indices. We believe AutoCord is the first ILP method to learn a classifier from databases by operating directly in SQL.

Other approaches to scale relational learning include CrossMine [Yin et al., 2006], which reduces the number of relations by using a "virtual join." Relational Structure Relation modifies FOIL for use in a logistic regression classifier [Popescul et al., 2002].

A more recent perspective on ILP, Relational Mining, focuses on modeling relational dependencies. For example, it has been used to classify and cluster hypertexts, taking advantage of their relational links between instances [Slattery and Craven, 1998]. Alternatively, the Probabilistic Relational Model [Taskar et al., 2001] framework extends Bayesian networks to a relational setting.

Modeling the database without an explicit feature vector can be contrasted with work that uses feature induction. For example, a feature vector can be expanded using conjunction operators to improve accuracy [McCallum, 2003]. Alternatively, Popescul and Ungar [Popescul, 2004] proposed modifying SQL queries systematically, to generate cluster ids that can be used as features in logistic regression.

6. CONCLUSION AND FUTURE WORK

This paper proposes, implements, and evaluates an automated process for training classifiers on relational data logged by an intelligent tutor. Unlike certain machine learning techniques such as regression, it does not require defining a feature vector first. Although AutoCord's implementation is somewhat specific to the Reading Tutor database, the method appears sufficiently generic to apply to data from many other tutors. The list below shows some of the features that could be implemented to improve the classification accuracy of AutoCord.

Evaluating on more tasks: So far we have applied AutoCord only to predicting story completion. We need to evaluate it on other classification tasks, such as characterizing children's behavior according to whether they or the Reading Tutor picked the story [González-Brenes and Mostow, 2010], or what events tend to precede a software crash.

Adding more operators: For example, event duration is useful for predicting story completion [González-Brenes and Mostow, 2010], but is not an explicit database field. Adding a Derive operator would address this issue by computing simple combinations of existing fields, e.g. end_time – start_time, to use as additional fields.

Combining queries: Due to the nondeterministic nature of the sampling method and the Extend operator, different runs of the search process are likely to generate different

queries, varying in classification accuracy. Picking the best one would be a simple way to improve accuracy. Since they also vary in what information they use, combining them into an ensemble of classifiers could improve accuracy further.

Operationality criteria: We enforce specific operationality criteria *ad hoc* by adding clauses to the query or by excluding particular features or constants from it. We defer to future work the invention of a general way to express operationality criteria in machine-understandable form and translate them into enforcement mechanisms automatically.

REFERENCES

BRUNK, C.A. and PAZZANI, M.J. 1991. An investigation of noise-tolerant relational concept learning algorithms. In *Proceedings of the Eighth International Workshop on Machine Learning*, Evanston, IL, 1991, 389–393.

COHEN, W. 1995. Learning to classify English text with ILP methods. *Advances in inductive logic programming*, 124–143.

GONZÁLEZ-BRENES, J.P. and MOSTOW, J. 2010. Predicting Task Completion from Rich but Scarce Data. In *Proceedings of the 3rd International Conference on Educational Data Mining*, Pittsburgh, PA, June 11-13, 2010, R.S.J.D. BAKER, A. MERCERON and P.I.J. PAVLIK, Eds., 291-292.

MOSTOW, D.J. 1983. Machine transformation of advice into a heuristic search procedure. In *Machine Learning*, R.S. MICHALSKI, J.G. CARBONELL and T.M. MITCHELL, Eds. Tioga, Palo Alto, CA, 367-403.

MOSTOW, J. and AIST, G. 1999. Giving help and praise in a reading tutor with imperfect listening -- because automated speech recognition means never being able to say you're certain. *CALICO Journal* 16(3), 407-424.

MOSTOW, J., AIST, G., BURKHEAD, P., CORBETT, A., CUNEO, A., EITELMAN, S., HUANG, C., JUNKER, B., SKLAR, M.B. and TOBIN, B. 2003. Evaluation of an automated Reading Tutor that listens: Comparison to human tutoring and classroom instruction. *Journal of Educational Computing Research* 29(1), 61-117.

MOSTOW, J. and BECK, J. 2006. Some useful tactics to modify, map, and mine data from intelligent tutors. *Natural Language Engineering (Special Issue on Educational Applications)* 12(2), 195-208.

MOSTOW, J. and TAN, B.H.L. 2010. AutoJoin: Generalizing an Example into an EDM query. In *Proceedings of the 3rd International Conference on Educational Data Mining*, Pittsburgh, PA, June 11-13, 2010, R.S.J.D. BAKER, A. MERCERON and P.I.J. PAVLIK, Eds., 307-308.

MUGGLETON, S. 1995. Inverse entailment and progol. *New Generation Computing* 13(3), 245-286.

MYSQL 2004. Online MySQL Documentation at http://dev.mysql.com/doc/mysql.

POPESCUL, A., UNGAR, L., LAWRENCE, S. and PENNOCK, D.M. 2002. Towards structural logistic regression: Combining relational and statistical learning. In *Multi-relational Data Mining Workshop at KDD-2002*, Alberta, Canada, 2002.

PROVOST, F. and KOLLURI, V. 1999. A Survey of Methods for Scaling Up Inductive Algorithms. *Data Mining and Knowledge Discovery* 3(2), 131-169.

QUINLAN, J.R. 1990. Learning logical definitions from relations. *Machine Learning* 5(3), 239-266.

SLATTERY, S. and CRAVEN, M. 1998. Combining statistical and relational methods for learning in hypertext domains. *Inductive Logic Programming*, 38-52.

TASKAR, B., SEGAL, E. and KOLLER, D. 2001. Probabilistic classification and clustering in relational data. In *International Joint Conference on Artificial Intelligence*, 2001, Lawrence Erlbaum, 870-878.

YIN, X., HAN, J., YANG, J. and YU, P. 2006. CrossMine: Efficient Classification Across Multiple Database Relations. In *Constraint-Based Mining and Inductive Databases*. Lecture Notes in Computer Science, J.-F. BOULICAUT, L. DE RAEDT and H. MANNILA, Eds. Springer Berlin / Heidelberg, 172-195.