Learning Classifiers from a Relational Database of Tutor Logs

JACK MOSTOW, JOSÉ GONZÁLEZ-BRENES, BAO HONG TAN
Carnegie Mellon University, United States

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, manual feature engineering for a previous classification task [González-Brenes and Mostow, 2010] took approximately two months.

This paper presents AutoCord (Automatic Classifier Of Relational Data), an implemented system 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

---

Authors' addresses: J. Mostow, RI-NSH 4103, 5000 Forbes Avenue, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, United States; email: mostow@cs.cmu.edu; J. González-Brenes; email: joseg@cs.cmu.edu; B.H. Tan; email: btan@andrew.cmu.edu. This work was supported in part by the Institute of Education Sciences, U.S. Department of Education, through Grant R305A080628 to Carnegie Mellon University. The opinions expressed are those of the authors and do not necessarily represent the views of the Institute or the U.S. Department of Education. We thank the educators, students, and LISTENers who generated our data.

[Mostow et al., 2010], which displays logged Reading Tutor data in human-readable, interactively expandable form.

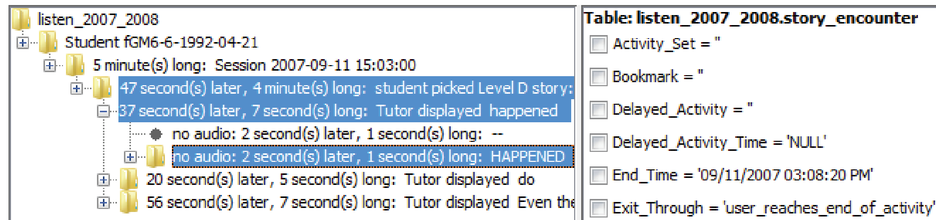


Figure 1: Session Browser's partially expanded event tree (left); partial record for the highlighted story_encounter event (right).

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. 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 is a label that 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 also have corresponding End_time and Ems fields.

Here the user has partially expanded the tree of events by clicking on some "+" icons. 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 summarized on the preceding line, and is therefore indented further. The story encounter's children are the sentence encounters shown 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 highlighted events by clicking on them with the CTRL key down. Given such a constellation of related events, the Session Browser's AutoJoin operator [Mostow and Tan, 2010] generalizes it into a pattern of which it is an instance. To infer a pattern from a single instance, AutoJoin heuristically assumes that repetition of a constant unlikely to recur by coincidence, such as a user ID, is a requirement of the pattern. AutoJoin represents the inferred pattern as a MySQL query [MySQL, 2004] that can retrieve instances of the pattern. An example of such a query is:

```
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
    (st.Start_Time = se.Start_Time) AND
```

} Identify sentence encounter as part of the story encounter.

} Ensure it is the first sentence encounter.

```
(st.Machine_Name = u.Machine_Name) AND                        } Identify utterance as
(st.User_ID = u.User_ID) AND                                  } part of the sentence
(se.sms = u.Sentence_Encounter_sms) AND                       } encounter.
(st.Start_Time = u.Sentence_Encounter_Start_Time) AND         }
(se.End_Time = u.End_Time)                                    } Ensure it is the last
                                                                utterance of the
                                                                sentence encounter.
```

2.3 Operationality criteria for learned queries

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. 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. For instance, a story encounter's End_Time field tells us when the encounter ends, but obviously the Reading Tutor can only log this information once the encounter actually ends, so the trained classifier cannot use it to help predict whether a child will finish a story. Similarly, we use the Exit_through field of a story encounter to label it as a positive or negative example of story completion, but the trained classifier cannot use it to make predictions, since that information is only available once the encounter ends. 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 a specified time interval after a story encounter starts, we should train and test it on data representative of what will be available then. To simulate such data, we restrict the training and test sets to story encounters lasting at least this long, and we exclude events logged after this amount of time elapsed since the story encounter started. We implement these constraints by adding the following two clauses to a query:

```
… AND (UNIX_TIMESTAMP(st.End_Time) – UNIX_TIMESTAMP(st.Start_Time) >= [limit])
   AND (se.Start_Time <= DATE_ADD(st.Start_Time, INTERVAL [limit] SECOND))
```

Here, *[limit]* is the time limit in seconds, say 10. Then the training and test sets include only story encounters that lasted at least 10 seconds, and the training and test procedures can only consider events that occurred within these story encounters' first 10 seconds.

Operationality criteria may also restrict what sort of classifier is useful to learn. For instance, to apply to future data, we may not want the trained classifier to be specific to any particular 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, we exclude those fields from the query.

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, encoded by the first letter 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 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.

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
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.

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 */
AND (UNIX_TIMESTAMP(st.End_Time) – UNIX_TIMESTAMP(st.Start_Time) >= [limit])
```

Other classification problems would require a different initial query.

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. 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 as the number of positive examples in R 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. Recall that the Exit_through field provides the label for a retrieved example. 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; otherwise, the label will be negative. To train a query representative of negative examples, we can re-interpret the

value of the Exit_through field of the story encounters in the training set. 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. It is possible for the same example to have multiple labels if it is checked against more than one classifier, each of which represents a different category. An evaluation metric for accuracy would need to penalize such cases appropriately.

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 */
```

3.2 Contrast Operator

The Contrast operator adds a single constraint that best distinguishes positive from negative examples. It generates this constraint based on a split in the distribution of values for a field. For example, if all positive examples have values below 5 for a particular field, and if all negative examples have values above 5 for that field, then the split value 5 perfectly separates positive from negative examples. To find the field that can provide the best split, AutoCord calculates the frequencies of values for each column of the results table retrieved by the initial query. It computes two sets of frequencies – one for positive examples and another for negative examples. To illustrate, consider the following results table:

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 represents a story encounter. The rest of the fields are omitted for brevity. Assume the first two rows are positive examples, while the third is a negative example. The calculated frequencies are:

	New_Word_Count	Initiative	Student_Level	….
Positive	4: once, 6: once	Student: twice	A: once, C: once	
Negative	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 it adds the new constraint 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 choose one that separates as many positive examples as possible from the negative examples. The Contrast operator considers the mathematical relations =, !=, <, <=, >, and >=.

3.3 Extend Operator

The Extend operator essentially captures the relational structure of positive examples. To do so, it first picks a random positive example (which is a row) from the results table. Recall that a row in the results table represents a collection of events. 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 then applies AutoJoin and adds the resulting constraints to the input query.

We illustrate the Extend operator on the initial query shown at the end of Section 3.1. This query only represents story encounters, so it retrieves a table of results where each row represents a single story encounter. Suppose the Extend operator randomly picks one such story_encounter and then one of its children, namely a sentence_encounter. Applying AutoJoin to these two events might yield this query:

```
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)
```

AutoJoin adds the last three constraints because both events have the same values for the fields Machine_Name, User_ID, and Start_Time.

3.4 Aggregate Operator

The Aggregate operator generates additional pseudo-fields for the Contrast operator to work on. The pseudo-fields of an event refer to the aggregated fields of the event's children. We shall illustrate the idea of pseudo-fields using the figure below.

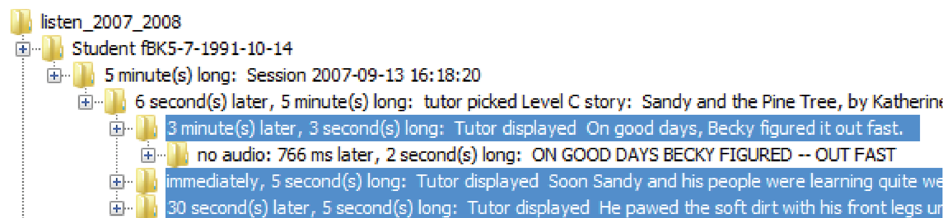


Figure 3: The highlighted sentence_encounter events of the story_encounter event.

Figure 3 highlights the three children of the story_encounter event "6 second(s) later, 5 minute(s) long: tutor …". These children are sentence_encounter events. As its name suggests, the Aggregate operator aggregates the values of each sentence_encounter field over these children and adds them as pseudo-fields of the story_encounter event to provide additional information about it. For instance, the aggregated field AVG(se.Word_Count), where "se" refers to each sentence_encounter, represents the average word count of the sentences in a story_encounter, reflecting its reading level.

For efficiency reasons, AutoCord precomputes the aggregated fields for all events in the training set before the search starts and stores them in a separate temporary table for each parent-child relation and specified time limit. The following example query calculates the table for the story_encounter/sentence_encounter relation:

```
CREATE TEMPORARY TABLE `story_encounter-sentence_encounter_agg` AS
SELECT st.*, AVG(se.Word_Count) AS _AVG_Word_Count,
[other aggregated fields…]
FROM
        story_encounter st,
        sentence_encounter se
WHERE  st.Machine_Name = se.Machine_Name
```

```
AND st.User_ID = se.User_ID
AND st.Start_Time = se.Story_Encounter_Start_Time
AND se.Start_Time <= DATE_ADD(st.Start_Time, INTERVAL [limit] SECOND)
AND UNIX_TIMESTAMP(st.End_Time) - UNIX_TIMESTAMP(st.Start_Time) >= [limit]
GROUP BY st.Machine_Name, st.User_ID, st.Start_Time, st.sms
```

Recall that the last two constraints impose the time limit operationality criterion. Also note that the GROUP BY clause is necessary for the aggregation to work correctly. Currently, AutoCord supports only the MIN, MAX, AVG, SUM, COUNT, and STDDEV aggregator functions, and only on numeric-valued fields, except for COUNT, which simply counts the number of rows it aggregates over. It applies each aggregator function to every field of the child event, as indicated by *[other aggregated fields…]*.

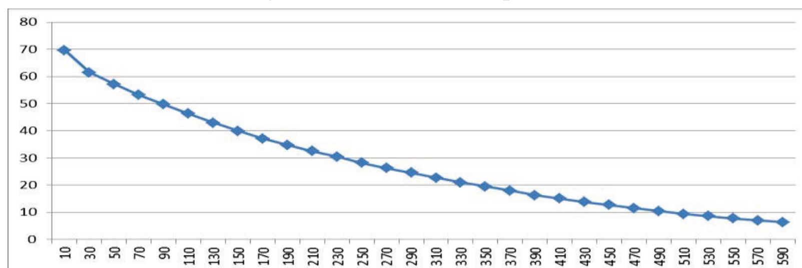
Using an aggregated field in a constraint requires a join to the temporary table, e.g.:

```
… FROM
      story_encounter st,
      `story_encounter-sentence_encounter_agg` `st-sentence_encounter_agg`
WHERE (st.User_ID <= 'mDS8-8-1998-09-22') /* Use training set */
AND (UNIX_TIMESTAMP (st.End_Time) – UNIX_TIMESTAMP(st.Start_Time) >= [limit])
AND (st.Machine_Name = `st-sentence_encounter_agg`.Machine_Name)
AND (st.User_ID = `st-sentence_encounter_agg`.User_ID)
AND (st.Start_Time = `st-sentence_encounter_agg`.Start_Time)
AND (st.sms = `st-sentence_encounter_agg`.sms)
AND (`st-sentence_encounter_agg`.`_STDDEV_Word_Count` >= '0.4')
```

The last constraint, added by the Contrast operator, selects story encounters whose sentence lengths vary enough to have standard deviation of at least 0.4. Such variation might make stories more interesting, or simply reflect harder stories read by better readers likelier to complete them.

4. EVALUATION

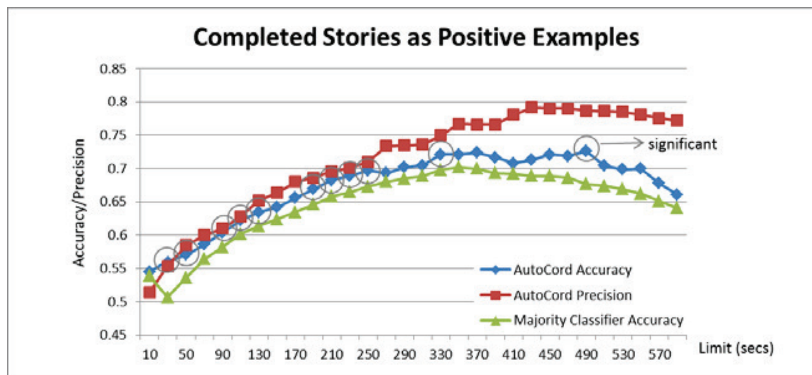
Section 2.3 discussed how to restrict the amount of information the search algorithm can look at for each story encounter in the labeled training set. In this way, the algorithm can only learn from events available from the start of the story encounter up to the specified time limit. In other words, 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.



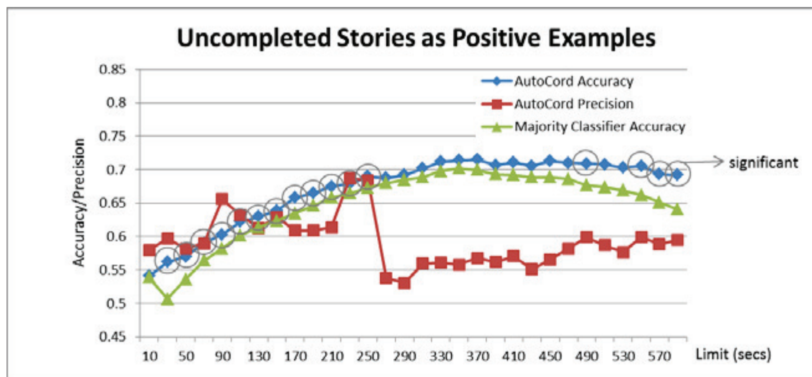
We modified the search algorithm slightly to restricting the information available. More specifically, we modified the Extend operator so that whenever it adds a new event to the current query, the new event start time starts before the specified time limit. We similarly constrained the Aggregate operator to include only such child events too. We ran the modified search algorithm for various time limits ranging from 10 to 590 seconds

in increments of 20 seconds, which corresponds roughly to the average duration of a sentence encounter. After each run, we got a query suitable for the specified time limit. The graph above shows the percentage of story encounters in the test set (shown on the y-axis) that lasted at least a certain number of seconds (shown on the x-axis).

We executed the queries generated by the algorithm, one by one, and for each one, calculated its accuracy and precision. It is not meaningful to compare the values of these two metrics, but to save space we plot them both on the y-axis of the same graph below against the time limit in seconds on the x-axis. We include majority class accuracy as a baseline for comparison. The majority classifier always outputs the label assigned to the majority of the story encounters in the training set, so its accuracy for a specified time limit is simply the percentage of story encounters with that label in the test set for that limit. We circle the points where the difference in classification accuracy between the trained query and majority class is statistically significant at $p < .05$. To account conservatively for statistical dependencies among data points from the same student, we test whether this difference exceeds zero by more than a 95% confidence interval defined as twice the weighted standard error of the per-student difference, weighting by the number of data points per student.



For comparison, we trained two types of queries, one with completed stories as positive examples and the other with uncompleted stories as positive examples. The graph below shows the corresponding accuracy and precision when uncompleted stories are treated as positive examples. Accuracy is similar, but precision is less consistent.



González-Brenes and Mostow [2010] applied ℓ1-regularized logistic regression to the same classification task, but their results are not directly comparable because they framed

it differently. They expressed their time limit as a number of sentence encounters before a story encounter ended, rather than as a number of seconds after it started.

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, whereas AutoCord uses the logical conjunction AND to combine all constraints. It uses negation only to negate the equality relation in the Contrast operator, not for an entire constraint. Also, AutoCord assumes that relations describe events, works 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 with many relations [Yin et al., 2006]. In contrast, AutoCord's direct use of SQL queries enables it to operate directly on large event databases thanks to efficient retrieval from suitably indexed tables of events.

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 system 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" in which the tuple IDs of the target relation are attached to the tuples of a non-target relation. CrossMine employs selective sampling to achieve high efficiency on databases with complex schemas. In contrast, AutoCord operates on all training data available to eliminate sampling bias.

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]. AutoCord's Extend operator also exploits relational links between events.

Modeling the database without an explicit feature vector contrasts 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, which is similar to what AutoCord does, but their method involved generating cluster IDs that can be used as features in logistic regression.

6. CONCLUSION

This paper proposes, implements, and tests an automated process for training classifiers on relational data logged by an intelligent tutor. Unlike many machine learning techniques, it does not require defining a feature vector first. Future work includes:

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. To address this limitation, a Derive operator would compute simple combinations of existing fields, e.g., end_time – start_time, to use as additional fields.

Combining queries: Due to the Extend operator's nondeterministic nature, different runs of AutoCord can generate different queries, varying in the information they use and the classification accuracy they achieve. Picking the best one or combining them into an ensemble of classifiers could improve accuracy.

Operationality criteria: AutoCord enforces specific operationality criteria *ad hoc* by adding clauses to the query or by excluding particular fields or constants from it. Future work might invent a general way to express operationality criteria in machine-understandable form and translate them into enforcement mechanisms automatically.

Generalizing to other tutors: AutoCord relies on the schema of the Reading Tutor database for reasons of efficiency and expedience rather than due to intrinsic limitations. Moreover, although its implementation uses MySQL, its method should apply to any relational database system. Generalizing AutoCord to apply to similarly structured data from other tutors would multiply its potential impact.

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.E. 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., BECK, J.E., CUNEO, A., GOUVEA, E., HEINER, C. and JUAREZ, O. 2010. Lessons from Project LISTEN's Session Browser. In *Handbook of Educational Data Mining*, C. ROMERO, S. VENTURA, S.R. VIOLA, M. PECHENIZKIY and R.S.J.D. BAKER, Eds. CRC Press,Taylor & Francis Group, New York, 389-416.

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.

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.

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.