Learning Transformation Rules by Examples

Bo Wu, Pedro Szekely, and Craig A. Knoblock
Information Science Institute and Computer Science Department
University of Southern California
4676, Admiralty Way
Marina del Rey,CA,90292
bowu@isi.edu

Introduction

A key problem in data integration is that the data formats are different from source to source. Without a unified format, it is difficult to integrate, analyze, and visualize the data. Data transformations are often required to address this problem by converting the data from multiple sources into the same format. One common approach to handle data transformation is to define a transformation language and then generate rules based on the language to perform data transformation on a large set of data such as google refine(**?**) and perl scripts. However, this approach usually requires expert users to write individual transformations for each data source manually.

A variety of work (**?**; **?**; **?**) tries to take advantage of user input to solve the transformation problem, but these methods either cannot learn rules from training data or need the training data to contain all the intermediate steps. We have developed an approach where the user only needs to provide the target value as an example. Our approach exploits a Monte Carlo tree search algorithm, called UCT (**?**), which provides a method for searching in a large state space. We adopt this search method to identify the sequence of rules that can transform the original values to the target values.

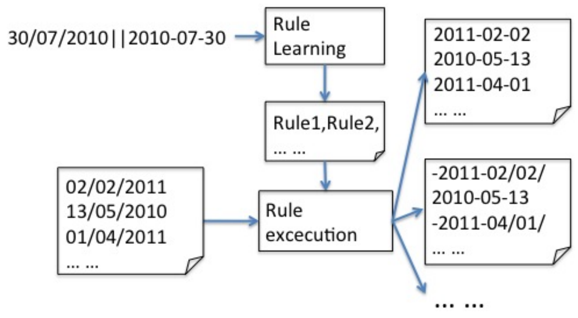

Figure 1: Data transformation framework

In this paper, we provide a general approach to automatically learn transformation rules through examples. As shown in Figure **??**, a user might want to reverse the order of the date and use hyphens to replace slashes. The user would just provide the system with an example "30/07/2010" and "2010-07-30". The system would learn the transformation rules that are capable of performing the transformation conveyed by the example. It then applies the rules to the remaining dates to transform their format automatically.

```
1  del → DEL what ∨ DEL range;
2  what → quantifier tokenspec;
3  quantifier → ANYNUM ∨ NUM ;
4  tokenspec → singletokenspec ∨ singletokenspec tokenspec ;
5  singletokenspec → token ∨ type ∨ ANYTOK;
6  type → NUMTYP ∨ WRDTYP ∨ SYBTYP ∨ BNKTYP ;
7  range → start end; scanningOrder → FRM_BEG ∨ FRM_END;
8  start → scanningOrder posquantifier ;
9  end → scanningOrder posquantifier ;
10 where → scanningOrder posquantifier ;
11 posquantifier → INCLD? tokenspec ∨ NUM ;
```

Figure 2: Delete grammar

However, there are many interpretations for the example. The example shown in the figure could also be interpreted as move "2010" to the front or move the last token to the front. The first interpretation obviously doesn't comply with users intention. Moreover, data transformations often consist of multiple steps and an error in one step would propagate to the next step. In order to handle the large number of interpretations, we define a grammar to describe the possible edit operations, including inserting, moving and deleting tokens. For simplicity only the delete grammar is shown in Figure **??**.

Under the above setting, the data transformation problem can be formally defined as

$$\exists g \in G \; g(d) = d^{'} where \; g = (r_1, r_2, ...r_n) \quad (1)$$

In equation **??**, d is the original token sequence and $d^{'}$ is the corresponding target token sequence. g is the transformation rule sequence consisting of individual steps r_i. G is the entire grammar space. The goal is to find a transformation g that correctly transforms the provided examples and implements the transformation intended by the user on all entries.

Methodology

Subgrammar Space

As mentioned above, the goal is to find rule sequences that are consistent with all examples. Therefore, the search space would be (INS)*(MOV)*(DEL)*. In this formula, the data transformation task is divided into three phases. The first phase consists of an undetermined number of insert operations that insert the missing tokens in the original token sequence. The second phase consists of an undetermined number of move operations that change the order of the tokens to conform with the target token sequence. The third phase consists of an undetermined number of delete operations that remove the extra tokens.

The grammar space is very large. We introduce subgrammar spaces to reduce the search space into several smaller search spaces. The entire grammar space is reformulated as a disjunction of clauses where each clause corresponds to one subgrammar space. A subgrammar space is a space consisting of a fixed number of edit operations. Each edit operation itself is a grammar like the delete grammar defined in Figure **??**. But each operation has some pre-built parse trees for certain nonterminals, which are derived from token sequences aiming to further reduce the grammar space. The subgrammar spaces for the example in Figure **??** could be written as $(INS_1^1 INS_1^2 MOV_1^3 MOV_1^4 MOV_1^5 MOV_1^6 DEL_1^7) \vee ((INS_2^1 INS_2^2 MOV_2^3 MOV_2^4 MOV_2^5 DEL_2^6)) \vee ......$

In order to generate a subgrammar space, first we generate edit operations for each individual example. The edit sequences with a length and edit type shared by all examples are used to set the length and edit type for a subgrammar space. Secondly, we generate candidate values for a set of non-terminals to further reduce the grammar space. The edit operation sequences of different examples that correspond to the same grammar space are used to generate these candidate value set.

Search

For each subgrammar space, we use the UCT algorithm to identify rule sequences that are consistent with the examples. For each edit component in a subgrammar space, due to large number of possible rules, a subset of rules is sampled for this edit component. Each rule leads the current state into the next state. These candidate states are evaluated and the one with minimum score is chosen to be explored further. The algorithm then samples the same number of rules for this new state. This process iterates until reaching a predefined depth or reaching the last edit component of the subgrammar space. The algorithm outputs the path that leads from the root state to the target leaf state as a transformation rule that is consistent with the examples.

Experiment

The dataset contains 7 transformation scenarios with 50 entries and corresponding transformed results. These scenarios include a combination of insert, move, and delete operations. For each subgrammar space, the search algorithm is run 200 times. The number of rules sampled for each state is 6 and the search depth is 3. Initially, one entry is randomly chosen as an example for the system to learn rules. If all generated rules fail on the test data, an incorrectly transformed entry together with its transformed result are added as a new example. This process iterates until finding a rule sequence, which could transform all the entries correctly.

Table 1: Experiment Results

Dataset	Examples	Time	Correct rules
50_beoaddr	2	(0.25, 0.41)	(1, 9)
50_address	(5, 6)	(0.19, 0.43)	(1, 3)
50_date	(1, 3)	(3.40, 6.23)	(1, 3)
50_tel1	1	(0.03, 0.06)	(97, 142)
50_tel2	1	(0.47, 0.74)	(19, 31)
50_time	(1, 2)	(1.08, 4.80)	(1, 8)
50_data2	1	(2.80, 3.49)	(1, 3)

The above experiment is run 15 times. In Table **??**, the example count shows the minimal and maximal number of examples used during 15 runs. Time cost shows the min and max time in seconds used to learn the rules from examples. Correct rules show the min and max number of learned correct rules. The preliminary results show correct rules for the above scenarios can be learned in a few seconds.

Conclusion

In this paper, we presented an approach that can learn transformation rules from examples. These rules are then used to transform the remaining data. In order to tackle the large grammar space, a subgrammar space is introduced to reduce the search space and a search algorithm is adopted to search in these subgrammar spaces for a sequence of rules that is consistent with the examples. Our preliminary results show this approach is very promising. Future work will focus on accelerating the search algorithm and expanding coverage.

References

Google Refine http://code.google.com/p/google-refine/.

Kandel, S.; Paepcke, A.; Hellerstein, J.; and Heer, J. 2011. Wrangler: interactive visual specification of data transformation scripts. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, 3363–3372.

Kocsis, L., and Szepesvri, C. 2006. Bandit based montecarlo planning. In *In: ECML-06. Number 4212 in LNCS*, 282–293. Springer.

Liang, P.; Jordan, M. I.; and Klein, D. 2010. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*.

Raman, V., and Hellerstein, J. M. 2001. Potter's wheel: An interactive data cleaning system. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, 381–390.