Iteratively Learning Conditional Statements in Transforming Data by Example

Bo Wu
Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA
Email: bowu@isi.edu

Craig A. Knoblock
Information Science Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA
Email: knoblock@isi.edu

***Abstract*—Programming by example (PBE) enables users to transform data formats without coding. As data transformation often involves data with heterogeneous formats, it often requires learning a conditional statement to differentiate these different formats. However, to be practical, the method must learn the correct conditional statement efficiently and accurately with little user input. We present an approach to reduce the conditional statement learning time and the required amount of data. This approach takes advantage of the fact that users interact iteratively with a programming-by-example system. Our approach learns from previous iterations to guide the program generation for the current iteration. The final results show that our method successfully reduces the system running time and the number of examples.**

Keywords—data transformation;clustering;classification; Programming by Example;

I. Introduction

Data transformation is an essential preprocessing step for many tasks. It changes the data from the source format to the target format. However, performing such transformation often requires users to write programs or teach the system by demonstrating a sequence of transformation operations [1] [2] [3] [4]. It can be time consuming and error-prone. The latest PBE approaches enable users to generate transformation programs directly from examples [5] [6].

TABLE I. Example Scenario

ID	Original	Target
R1	5.25 in HIGH x 9.375 in WIDE	9.375
R2	9.75 in|16 in HIGH x 13.75 in|19.5 in WIDE	13.75
R3	20 in HIGH x 24 in WIDE	24
R4	Image: 20.5 in. HIGH x 17.5 in. WIDE	17.5
...	...	...
Rn	12 in|14 in HIGH x 16 in|18 in WIDE	16

For example, the original values in Table I are the size information of some artwork. Some records not only have the sizes for the artwork but also have the sizes for the frames, which are separated by a vertical bar. The user only needs to extract the widths of these artworks. If there is a bar separating two values for the two widths, the user wants the first one as the target value. For instance, the user extracts the 13.75 as the target in the R2 record.

To transform the original values into target values using the PBE system, the user just gives some target values as shown in the right column and the approach automatically learns a program that can transform the data from the original format into the target format. This program is then applied to the rest of the data to convert those data automatically. For example, the user may only give the target values for the R1 and R2 records; the system then learns the transformation program that can transform the rest of the original values. The user then reviews the results. If the user finds entries that are transformed incorrectly, the user can provide an additional example to refine the transformation program. This process iterates until the user determines all the results are correct.

Much real world data has multiple formats and a single conversion cannot change all these formats. Therefore, it is essential for a PBE system to generate transformation programs that are capable of handling conditional transformations. It requires the system to learn conditional statements that can recognize these formats and then apply the right transformation to the data of a specific format. As shown in Table I, the transformation program should distinguish between the two types of formats: the one with a bar separating two widths and the one without the bar.

In this paper, we solve the problem of **learning expressive conditional statements efficiently with few user provided examples**. As shown in Figure 1, to learn the conditional statement using the given examples, the current state-of-the-art approach by Gulwani [5] first (1) partitions the examples into several clusters where examples in one cluster can be transformed by the same conversion and then (2) learns a classifier to distinguish these formats. For example, given the first four rows as examples in Table I, the partition algorithm generates two partitions: R1, R3 and R4 as one partition and R2 as its own partition. With the partitions, we can then train a classifier. This classifier can then be used to recognize the other inputs so that the approach can invoke the correct conversion for those inputs.

However, learning a conditional statement for PBE systems brings a series of challenges. First, the users are waiting for the response on the fly. The systems only have limited time to generate the conditional statement. However, PBE approaches

This research is based upon work supported in part by the National Science Foundation under Grant No. 1117913.

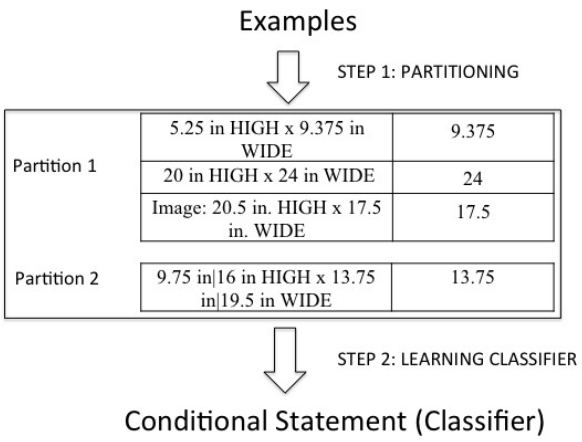

Fig. 1. The steps for learning the conditional statement

must identify one partitioning among the many ways of clustering the examples such that every partition generated by this partitioning can produce a program that is consistent with all its examples. Moreover, it is computationally expensive to verify whether certain examples can form a partition that can produce a consistent program.

Second, examples often have multiple valid conditional statements. Naively, the approach can treat each example as one partition and learn a conditional statement to differentiate these partitions. However, the approach aims to identify an interpretation that is consistent with most of the records. Based on the Occam's razor principle, the simplest interpretation tends to be correct [5]. Therefore, the approach should cluster the examples into the fewest partitions.

Third, users generally provide few input-output examples. The conditional statements trained based on the few examples usually have poor prediction accuracy, as the training data may not fully represent the rest of the records.

To address the challenges mentioned above, we exploit the fact that the users usually iteratively interact with the system. Users provide the examples in an iterative way. Every time the user provides a new example, it triggers the system to produce a new transformation program that is consistent with the examples that it has so far. During the process, the system explores different ways of partitioning the examples and gains the knowledge of whether a group of examples can lead to a valid partition that can generate at least one consistent program. Therefore, we can maintain a record of the previous running information so that the system can learn from its past experience to guide the current partitioning. To fully utilize the previous knowledge, our approach learns a distance metric and integrates it into the partitioning. By applying distance metric learning, the system will assign large distances among the examples that cannot form a valid partition and assign small distances among the examples that can generate a valid partition. Through this distance metric, the formed clusters are less likely to violate any constraints owing to the small distances among the examples.

Moreover, this distance metric can be used to incorporate unlabeled inputs as training data to improve the classifier's accuracy. Here, unlabeled inputs refer to the inputs that are not used as examples. We first use the distance metric to calculate the distances from the unlabeled entry to each partition. Our approach then assigns the unlabeled entry into the corresponding partition based on its relative distances to other partitions. These unlabeled inputs along with examples are used to train the classifier used as the conditional statement, which reduces the required number of manually provided examples.

To summarize, our approach has the following contributions:

1) exploiting the iterative process to collect constraints;
2) learning a distance metric based on all known constraints to efficiently partition examples into clusters;
3) utilizing the unlabeled data to improve the accuracy of the conditional statement, which reduces the number of required examples.

II. Previous Work on Programming by Example

Our approach is built on the state-of-the-art PBE system described in Gulwani [5], which exploits the version space algebra like many other program induction approaches [3]. His approach defines a string transformation language. This language supports a restricted, but expressive form of regular expressions.

To better understand the structure of the generated transformation program, we use a different representation of the transformation program without changing its meaning. The transformation program learned from the examples is shown in Figure 2. This program has a conditional statement to find the class label for the input. It then choose the right partition transformation program to change the data format.

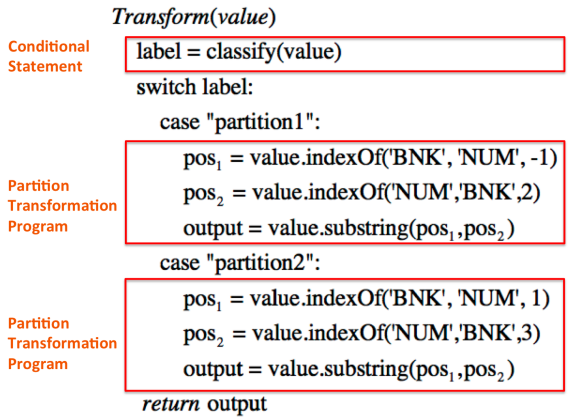

Fig. 2. Transformation Program

Definition 1. *$comp(p_1, p_2) = 1$, if partitions p_1 and p_2 are compatible. Otherwise, $comp(p_1, p_2) = 0$. Two partitions are compatible, if the merger of the two partition can generate a valid partition that can produce a program that is consistent with all its examples.*

Definition 2. *$merge(p_1, p_2)$ returns the newly generated partition by merging p_1 and p_2. $merge(p_1, p_2) = \phi$, if p_1 and p_2 are incompatible.*

To learn the conditional statement, the approach initially treats each example as its own partition. It iterates over all

the different pairs of partitions and select the two compatible partitions with highest *compatibility score (cs)* [5] to merge each time. The process iterates until no more compatible partitions are available. The *compatibility score* is defined as follows.

$$CS(p_1, p_2, P) = (CS_1(p_1, p_2, P), CS_2(p_1, p_2)) \quad (1)$$

$$CS_1 = \sum_{p_k \in P, k \neq 1, k \neq 2} z(p_1, p_2, p_k) \quad (2)$$

$$z(p_1, p_2, p_k) = \begin{cases} 1 & if\ (comp(p_1, p_k) = comp(p_2, p_k) \\ & \quad = comp(merge(p_1, p_2), p_k)) \\ 0 & otherwise \end{cases} \quad (3)$$

$$CS_2 = \frac{size(merge(p_1, p_2))}{\max(size(p_1), size(p_2))} \quad (4)$$

The *compatibility score* consists of two parts: (1) the agreement score CS_1, which captures the compatibility of merged partitions with the rest of partitions and (2) a finer score CS_2 used to measure the relative size of the partition after the merge. The CS_2 is used only when there is a tie of CS_1 scores. As shown in the equation above, the CS_1 is the summarization of the $z(p_1, p_2, p_k)$. $z(p_1, p_2, p_k)$ is 1 if both p_1 and p_2 are compatible with p_k, while the merged partition of p_1 and p_2 is also compatible with p_k. The CS_2 score calculates the relative size of the programs after the merge, where the $size$ function measures the number of programs that can be generated from the partition.

To select two partitions, the approach is required to calculate $O(n^3)$ times whether two partitions are compatible where n is the number of partitions. Verifying whether two partitions are compatible is computationally expensive; it requires verifying whether the merged partitions can generate a program that is consistent with examples. Each partition actually corresponds to a hypothesis space derived from the examples belonging to their partition where each hypothesis corresponds to a transformation program. Merging two partitions requires intersecting the two hypothesis spaces of the two partitions and verifying whether there exists a program that is consistent with the examples from both partitions. It is especially computationally expensive when the two partitions are incompatible, as it requires evaluating all the programs in that intersected space.

III. Construct Conditional Transformations

Our approach iteratively learns conditional statements for PBE systems as shown in Algorithm 1. The approach in every iteration can be broke into 3 high-level steps: (1) partitioning the examples, (2) learning the classifier and (3) generating branch programs for all the partitions. To takes advantage of the knowledge obtained from previous iterations, the approach also maintains the information collected from previous iterations and represent it using two types of constraints. In each iteration, these constraints are used improve the performance of partitioning and classifier learning.

- Set of cannot-merge constraints: each cannot-merge constraint in this set contains a group of examples that are not compatible in the same partition, which has been discovered previously.
- Set of must-merge constraints: each must-merge constraint in this set contains the examples that are already in the same partition.

Every time the user provides a new example, the approach first partitions the examples. The partition function takes five arguments: the cannot-merge constraints R, the must-merge constraints M, all the examples E, the unlabeled data U and the parameters D_w for the distance metric. It learns a distance metric D_w to partition the examples into several clusters. During the execution, it also adds the newly discovered constraints into the constraint sets and updates the distance metric parameters when necessary. The set of must-merge constraints (M) is cleared at the beginning of every iteration, as the newly provided examples can change the membership of previous examples. We empty M to allow different partitions to be formed in the new iteration. Second, the approach learns a classifier for the partitions. It uses the examples in each partition along with the unlabeled inputs assigned to each partition to train the classifier. This classifier serves as the conditional statement. Third, the approach generates partition transformation programs that are consistent with the examples in each partition respectively. To generate the conversion program for each partition, we used the approach described by Gulwani [5]. Finally, our approach combines the partition transformation programs $prog_1, prog_2, ...$ with the conditional statement G to create the transformation program.

Algorithm 1: Create Transformation Program

Input: examples $E = \emptyset$, unlabeled Data U, cannot-merge constraints $R = \emptyset$, must-merge constraints $M = \emptyset$, distance metric D_w

while *user provides a new example* **do**
- $E = E \cup e$
- $M = \emptyset$
- $partitions$=partition(R, M, E, U, D_w)
- G=learnClassifier(partitions, D_w, U)
- **for** p_i *in partitions* **do**
  - $prog_i$=learnPartitionTransformation(p_i)
- create transformation program by combining G and $\{prog_1, prog_2, ...\}$

Our approach can be easily applied to other PBE systems to learn the conditional statement, as it does not require knowing how the conversion programs are generated. It only needs to know whether the PBE system can successfully generate conversion programs from a group of examples, which will be explained in the following sections with more details.

A. Partition Algorithm

The Partition algorithm in each iteration places the examples into several clusters where each cluster can be covered by the same program. We prefer a smaller number of partitions, as it often leads to a more concise program with fewer conditional branches.

The partitioning algorithm takes a set of inputs. E is the set of the examples given by the user. U is a set of original

inputs randomly selected from all original inputs that are not used as examples. D_w is the distance metric learned from the previous iteration. The distance metric here is a weighted Euclidean distance and D_w contains the weights for all the features. R contains all the known cannot-merge constraints so far. M contains all the must-merge constraints discovered in the current iteration. As the new example given by the user may change the previous partitioning, M is initialized to empty at the beginning of each iteration.

Our system performs the following preprocessing before partitioning the examples. The goal is to convert the text into feature vectors. The approach first tokenizes all the texts into token sequences. A token is a class of characters. We use the following tokens to tokenize the string: NUM ([0-9]+), LWRD([a-z]+), UWRD([A-Z]), BNK(whitespace) and some punctuation token like a comma, a hyphen, etc. Here, the UWRD represents a single upper case letter and LWRD represents a continuous sequence of lower case letters. We also use start and end tokens to indicate the start and end position of the input.

With these token sequences, we count these tokens and use these counts of different tokens as the feature values. We count the tokens, which is not just based on their types but also on the content of the token. For example, a string "Width: 9.75 in" can be tokenized as "Start UWRD(W), LWRD(idth) Colon(:) BNK NUM(9) Period(.) NUM(75) BNK LWRD(in) End". The feature vector of this string: "NUM: 2, UWRD: 1, LWRD: 2, BNK: 2, Period: 1, Colon: 1, W: 1, idth: 1, 9: 1, 75: 1, in: 1". We can see that "W", "idth" etc. are considered as tokens. For the features based on the token content, we reduce the dimension of the feature vector by discarding some rare tokens that appear in less than 10% of the entries. By doing so, we can convert all the inputs into feature vectors. We then perform feature rescaling over these vectors.

To partition the examples, our partitioning algorithm essentially performs a constrained agglomerative clustering. As shown in Algorithm 2, each example becomes a partition p_i at the beginning.

Algorithm 2: Partition Algorithm

Input: examples E, unlabeled Data U, cannot-merge constraints R, must-merge constraints $M = \emptyset$, distance metric D_w
Output: partitions P

```
create a partition p_i for each e_i ∈ E
while ∃p_i, p_j ∈ P merge(p_i, p_j) ≠ ∅ do
    use D_w to find two closet partitions p_x, p_y
    p_z = merge(p_x, p_y)
    if p_z = ∅ then
        R = R ∪ {{e_i | e_i ∈ p_x ∨ e_i ∈ p_y}}
        Learn Distance Metric D_w using U, R and M
    else
        P = P \ {p_x, p_y}
        P = P ∪ {p_z}
        update M with P
Return P
```

Our algorithm will continue running if there are still partitions to merge. In each round, it tries to find the two closest partitions and merge them into one. To calculate the distance between two partitions, we use the minimal distance between the examples of the two partitions as below. This will result in a dense partition with examples compared to the centroid-based distance. The e_x and e_y are the examples belonging to partitions p_i and p_j.

$$d(p_i, p_j) = min\{d(e_x, e_y) | e_x \in p_i, e_y \in p_j\}$$

If the merge cannot generate a valid partition ($p_z = \emptyset$), the algorithm records a new cannot-merge constraint with all the examples in p_i and p_j. It adds this constraint to the cannot-merge constraint set R. It then uses the updated cannot-merge and the must-merge constraints to refine the distance metric D_w, which is described in the next section. With the updated distance metric, the algorithm finds the two closest partitions without contradiction to the constraints.

If the merge succeeded, the previous two partitions are removed from partitions P. The new partition (p_z) is inserted into the existing partitions set P. It also updates the must-merge constraints M using the current clusters that have at least two examples. The examples in the partition ($\{e_i | e_i \in p\}$) form one must-merge constraint.

For example, the user provides the first four records in Table I as examples. The algorithm now needs to partition these examples. In the previous iteration, the system has already successfully learned a program with three examples and identified one cannot-merge constraint (R1, R2 and R3). Suppose there are three partitions in the current iteration: (1) R1 and R3 are in one partition (R1, R3), (2) R2 is in the second partition and (3) R4 is in the third partition. Thus, R1 and R3 constitute one must-merge constraint. As R1 and R3 in the must-merge constraints have different "." counts (two "." in R1 and zero "." in R3), it indicates the examples can have different numbers of "." in the same partition. At the same time, the cannot-merge constraint with R1, R2 and R3 shows that differences in the number of "|" and ". " indicate that these examples may not be put into the same partition. Combining the information from both must-merge and cannot-merge constraints, the distance metric learning can figure out that the examples with different numbers of "|" should not be in the same cluster while the different number of "." does not matter. It will assign large distances among records with different numbers of "|" and assign small distances to the records with different number of ".". Therefore, the algorithm will put the R4 record into the same partition as R1 and R3.

B. Distance Metric Learning

Our approach learns a weighted Euclidean distance that is a special case of the Mahalanobis distance [7]. This weighted Euclidean distance is used to select partitions to merge. The weights of the features are used to capture the importance of that feature in deciding the distance. The higher weight for a certain feature, the larger distance incurred due to the variance on this feature between two records.

$$d(x, y) = \|x - y\|_w = \sqrt{\sum_i w_i (x_i - y_i)^2}$$

The x and y are two feature vectors and w_i is the weight for the i-th feature. We use weighted Euclidean distance for two reasons: (1) less expensive to calculate than Mahalanobis

distance as our program interacts with users on the fly, which requires fast response, (2) the weights in the distance metric are more interpretable.

To incorporate the constraints, our objective function is as follow.

$$\arg\min_{w>0} \sum_i \|x_i - e_{x_i}\|_w + a \cdot g(w) - b \cdot h(w) \tag{5}$$

$$g(w) = \ln(\sum_{X_m} \sum_{x_i, x_j \in X_m, i \neq j} \|x_i - x_j\|_w) \tag{6}$$

$$h(w) = \ln \sum_{X_r} \max_{x_i, x_j \in X_r} \|x_i - x_j\|_w \tag{7}$$

The first component is the sum of the squared weighted Euclidean distance from each unlabeled input to its cluster. To find which cluster this input belongs to, we simply assign the input to its closest partition. To calculate the distance between the input and a partition, we use the smallest distance between the input and the examples in that partition as follows:

$$d(x, p) = \min\{\|x - e_i\|_w \mid e_i \in p\}. \tag{8}$$

Here, x is the input, p is the partition and e_i is an example in partition p. $\|x - e_{x_i}\|_w$ is the weighted Euclidean distance between input x and the closest example e_{x_i} in p, which is the distance from x to its corresponding partition.

$g(w)$ is the penalization term corresponding to the must-merge constraints. A must-merge X_m constraint means the examples in this constraint should be in the same partition, which implies that these examples should be close to each other so that they can form a cluster. Therefore, we penalize the sum of the distances between examples in the must-merge constraint. $\sum_{x_i, x_j \in X_m, i \neq j} \|x_i - x_j\|_w$ adds the distances between all pairs of examples x_i and x_j in one must-merge constraint X_m. It then sums over all the different must-merge constraints.

$h(w)$ is the penalization term corresponding to the cannot-merge constraints. The cannot-merge constraint X_r means the examples in this constraint should not be in the same partition. However, a subset of the examples can still be in the same partition. For example as shown in Table I, R1, R2 and R3 together are not compatible, but R1 and R3 can be in the same partition. Intuitively, the examples in the cannot-merge constraint should at least have one pair of examples that are extremely far away from each other. This can also be interpreted as the requirement that the farthest two examples in this constraint should be extremely far away from each other. To model this type of constraint, we first use a max operator to find the farthest examples x_i and x_j by $\max_{x_i, x_j \in X_r} \|x_i - x_j\|_w$ in each constraint X_r and then try to maximize the distance of this pair of examples by minimizing its negative values. This term then sums over all the different cannot-merge constraints. The costs a and b provide a way of specifying the relative importance of the two types of constraints. The a is usually set as a large coefficient and b is set according to the ratio between the number of constraints in the must-merge and the cannot-merge sets.

As there is a max operator in the objective function, we propose an iterative optimization algorithm that alternates between finding the farthest pair in the cannot-merge constraints and finding the optimal w. The algorithm works as follows:

- Find the farthest pairs of examples x_i and x_j in each cannot-merge constraint X_r.
- Optimize the objective function in Equation 5 where $h(w) = \ln \sum_{X_r} \|x_i - x_j\|_w$ using the gradient descent.

Firstly, the optimization algorithm fixes on w to find the farthest pairs of examples in each cannot-merge constraint group to remove the max operator in the objective function; the algorithm later fixes on these farthest pairs of examples to find the w that makes the objective function achieve the minimum value using the gradient descent algorithm. The algorithm performs a line search to select the right step size to ensure $w > 0$ during the search. This process iterates until reaching a fixed number of iterations or the change of the objective function is below a threshold. As the objective function's value always decreases in each step of the optimization, our algorithm will finally converge to a local optima.

C. Learning the Classifier

Our system learns a multi-class classifier as the conditional after an iteration. The users are only willing to provide a small number of examples, which are usually 2-5 per partition. Relying on data solely from examples can result in a classifier with poor prediction performance, which in turn may require the user to provide more examples to improve the classifier's performance. Therefore, we augment the training data with both the examples and the unlabeled inputs assigned to that partition and then train a classifier to recognize these partitions. The data in each partition can be seen in Figure 3. For each partition, the upper table shows the examples of that partition. The bottom table shows the unlabeled data that has been assigned to this partition as it has the shortest distance to this partition as mentioned in previous section. Later, we can use the inputs of the examples and other raw unlabeled inputs together to train a SVM [8] classifier as the conditional statement for the transformation program.

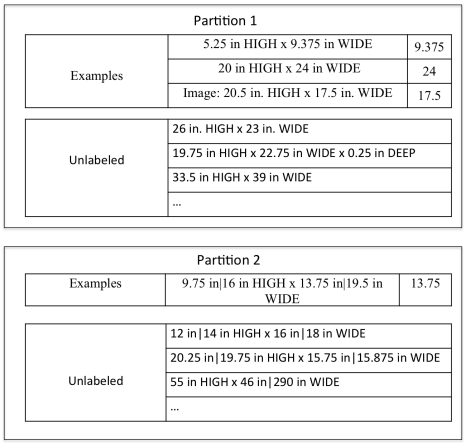

Fig. 3. The examples and the unlabeled data in each partition

We filter the unlabeled data before using it as the training data. As we mentioned before, the unlabeled inputs are added into the closest partition using the learned distance metric. To prevent the approach adding raw inputs into the wrong partition, we follow the steps in Algorithm 3 to keep only the inputs that we are certain about their labels.

Algorithm 3: Filter the unlabeled data in each partition

Input: partitions P, unlabeled data U

```
for p_i ∈ P do
    for u_i ∈ p_i do
        d_1=getDistance(u_i, p_i)
        for p_j ∈ P and p_j ≠ p_i do
            d_2 = getDistance(u_i, p_j)
            if (d_2 − d_1)/d_1 < ε then
                delete u_i
for p_i ∈ P do
    sort U_{p_i} ascendingly based on distance
    keep top K elements in U
```

The algorithm 3 iterates over all the unlabeled data u_i in each partition p_i. First, it computes the distance d_1 between the unlabeled input and the partition it belongs to. The distance between an input and a partition is defined in Equation 8. The algorithm then computes the distances between the input u_i and all other partitions p_j. If any distance d_2 is close to d_1 and the difference is within a threshold ε, it means the distances from the input to the two partitions are very close and the input lies near the boundary of the two partitions. We are not confident with the class label for these raw inputs. To avoid adding the inputs into the wrong class, we remove these unlabeled inputs from the partitions. In practice, we have found that setting the ε to 10% usually achieves a good result. Second, our approach sorts the remaining unlabeled inputs in the ascending order based on the distance to their partition. We then only keep the top K unlabeled inputs as the training data. We usually set the K to be 10 times the number of the examples in that partition. Finally, we use the inputs of the examples and the remaining unlabeled inputs as the training data to learn a SVM [8] classifier.

IV. Related Work

Programming-by-example approaches has been extensively studied for the past decades. Early work [9] [10] [11] in wrapper induction learns extraction rules from user labels to extract target fields from documents. These extraction rules are used to locate the target fields and return the values of these target fields. These rules are similar to the substring expressions used in our approach. However, our approach adopts a domain specific language to organize the extractors to generate more powerful programs. It allows the concatenation of the substring extractors in arbitrary order, loop extractors. Moreover, the approach can also handle multiple branches using conditional statements.

SMARTpython [12] learns programs using a subset of the python programming language through user demonstration, which supports conditionals, loops and arrays. But it only allows the if-else clause. To learn the clause, it analyzes the traces of various programs to identify the conditional statement. It then labels the conditional statement positive if it is evaluated to be true and labels the statement negative if it is evaluated to be false. Given the labeled data, it can train a binary classifier to serve as the conditional statement. Data Wrangler [4] is an interactive tool for data transformation. Besides supporting string level transformation, it also supports data layout transformation including column split, column merge, fold and unfold. It learns a parameter set from the user interaction to recognize whether a row, column or a cell is the target that should be transformed. This is essentially a binary classifier, which helps the user focus on the target data that they want to transform. Our approach is different from the two approaches above as our approach learns a multi-class conditional statement. It can include more than two branches in the program, which means it can handle more than two kinds of inputs at the same time.

Gulwani [5] developed an approach to synthesize a transformation program through input and output pairs. His approach directly learns a set of binary classifiers to recognize whether an input matches a certain format. As it has multiple binary classifiers, it can also handle multiple formats at the same time. However, the classifiers used in [5] are built based on conjunction or disjunction of a predefined set of predicates, which is hard to express the nominal values of the features like the counts of different tokens. Moreover, our work focuses on exploiting previous knowledge to learn the conditional more efficiently and accurately. Perelman [13] developed a program synthesizer which can synthesize a program in any domain if given a domain specific language. His approach iteratively creates new programs using previously generated subprograms, which can be used to improve the efficiency in generating the partition transformation code. However, his approach does not generate the conditional statement efficiently by utilizing constraints from previous iterations.

There is a large body of work in metric learning [14] [15]. Researchers applied distance metric learning in various clustering algorithms. Some of the most closely related work is [16] [17] [18] [19] [20]. Xing et al. [16] proposed to learn a Mahalanobis distance metric and applied it in K-means algorithm. Bilenko et al. [17] integrate the constraints and metric learning in K-means clustering by utilizing both the constraints and unlabeled data. The two approaches above both used pairwise constraints, which claim two instances should either be close or far away from each other. Davidson et al. [21] investigated applying instance-level must-link and cannot-link constraints in agglomerative clustering, which shows the feasibility of the problem. Bade and Nurnberger [18] described an approach that learned a distance metric to perform agglomerative clustering by introducing relative instance-level constraints. Zhao and Qi [19] extended instance-level constraints to order constraints to capture the hierarchical side information. Zheng and Li [20] used the triple-wise relative constraint, which is a special case of the order constraints. They then applied a ultra-metric dendrogram distance to improve effectiveness and efficiency of the hierarchical clustering. Our partition algorithm is essentially performing agglomerative clustering, which integrates distance metric learning with constraints. It is different from previous approaches as we first applied must-merge and cannot-merge constraints in distance metric learning, which describes the relationships among groups of instances instead of pairwise or relative pairwise constraints. Our constraints are defined on a group of instances, of which there can be more than two. We also developed an iterative algorithm to solve this problem. Moreover, we first applied this semi-supervised clus-

tering approach in the program synthesizing setting, which can effectively utilize constraints collected from previous running information to improve the system performance.

V. Evaluation

We describe our datasets, experimental results and then report the evaluation results.

A. Datasets

We identified 30 scenarios[1] that require conditional transformations. We collected 12 scenarios involving conditional transformations used in Wu et al. [22]. We also manually collected 18 conditional transformation scenarios to increase the number of scenarios. The data was gathered from student mashup projects in a graduate-level course, which required the students to integrate data from multiple sources to create various applications. They were required to perform a variety of transformations to convert the data into the target formats. We collected the editing scenarios from these projects and randomly selected 18 scenarios from them.

B. Experiment Setup

To fully evaluate our approach, we compared 5 different alternatives described below to validate our design decisions. We designed these alternatives to separately investigate the effects of the three key design differences: (1) the weighted Euclidean distance scoring functions compared to the compatibility score (DP v.s SP), (2) utilizing the constraints collected from previous iterations compared to not utilizing the previous constraints (IC vs non-IC) and (3) incorporating unlabeled data in learning a classifier compared to not incorporating the unlabeled data (ED v.s non-ED).

1) **Compatibility Score Based Partitioning (SP): This is the state-of-the-art approach [5] that calculates the *compatibility score* as described in Section II to decide the partitions to merge** .
2) SP with Incremental Constraints (SPIC): This is the version of SP approach that uses previous constraints to record the example compatibilities.
3) Distance Metric Based Partitioning (DP): This method learns a weighted Euclidean distance with only constraints discovered in the current iteration. The weighted Euclidean distance is then used to choose partitions to merge.
4) DP with Incremental Constraints (DPIC): this approach learns the weighted Euclidean distance with all the known constraints.
5) **DPIC with Expanded Training Data (DPICED): this is our approach introduced in this paper. Besides DPIC, it also incorporates unlabeled data in learning the classifier**.

All 5 algorithms above use agglomerative clustering as described in [5], which greedily selects the partitions to merge until there are no more compatible partitions to merge. However, these approaches have different ways of utilizing the constraints. The DP, DPIC and DPICED can learn from the constraints to adjust its scoring function (distance function).

[1]The data can be accessed at http://bit.ly/1jcZmGv. The system is available on Github at http://bit.ly/1EZnjhT.

TABLE II. Success Rates on All Scenarios

	DPICED	DPIC	DP	SPIC	SP
SccRate	1	1	0.97	0.77	0.77

TABLE III. Comparing Different Approaches

	Total Time (seconds)	Examples	Constraint Number
DPICED	3.9	5.4	6.1
DPIC	6.4	6.8	6.6
DP	8.3	6.8	17.6
SPIC	21.3	6.8	260.1
SP	26.5	6.9	305.8

As SP and SPIC need to calculate the compatibility score, they exploit both the cannot-merge and must-merge constraints to obtain the compatibility information to avoid redundantly verifying whether two partitions are compatible.

To measure the performance of the approaches above, we use the following metrics:

1) Total Time: the seconds used to correctly transform all the entries in a scenario.
2) Number of Examples: the number of examples required to successfully transform a scenario. It is also the number of iterations.
3) Constraint Number: the number of cannot-merge constraints encountered in transforming a scenario. The approach identifies a cannot-merge constraint when it tries to merge two incompatible partitions.
4) Success Rate: the percentage of scenarios that can be correctly transformed under 10 minutes. Otherwise, it is too long for a user to continue working on that scenario.

C. Experimental Results

We can see in Table II that the distance metric learning based approaches (DPICED, DPIC, DP) have much higher success rates than the compatibility score based approaches (SPIC, SP). The compatibility score based approaches can only transform 77% of scenarios in less than 10 minutes.

Aside from having a higher success rate, we also compared our approach (DPICED) against other approaches on total time, number of examples and constraint number. To prevent the failed scenarios from dominating the averaged results, we compared on the scenarios that all the approaches can successfully transform. The results are displayed in Table III.

The results in Table III show that DPICED outperforms other approaches in all metrics. Excluding the failed scenarios, the DPICED still saved 2.5 and 4.4 seconds compared to DPIC and DP; saved 17.4 and 22.6 seconds compared to SPIC and SP. We conducted paired one-tail t test. The results suggest that the improvements were statistically significant ($p < 0.05$).

Moreover, the results in Table III also validate our design decisions. **Utilizing the information from previous iterations improves efficiency.** The approaches utilizing the previous constraints avoided redundant work; they used less time compared to their counterparts which didn't leverage previous constraints. In Table III, DPIC and SPIC all used less time compared to DP and SP respectively.

Learning the weighted Euclidean distance also improves the system efficiency. DPIC used less time compared to

SPIC as shown in Table III. The improvement in the time per iteration comes from the reduction in the number of discovered cannot-merge constraints. The higher the number is, the more failed mergers the approach encounters, which in turn wastes more time. A better algorithm can learn from previous failed mergers to avoid intersecting these partitions in the future. For example, by learning from the constraints to adjust the Euclidean distance metric, the DPIC approach had much less number of failed mergers compared to SPIC. Moreover, SPIC has extremely high number of cannot-merge constraints, as computing compatibility score requires fully verifying the compatibilities over a large number of possible partitions. Computing Euclidean distance does not require verifying the compatibility until it tries to merge two partitions.

Augmenting the training data with unlabeled data can reduce the required number of examples. The saving in the number of examples is mainly due to the improved accuracy of the learned classifier. The poor classifier will classify the entry into a wrong category and then a wrong transformation would be applied to this entry. This will result in an incorrect result, which requires the user to provide a new example. Therefore, the classifier with higher accuracy would reduce the number of required examples. We measured the classifier accuracy on the unlabeled entries. By incorporating unlabeled data as training data, the DPICED method successfully increased the classifier's accuracy from 0.928 to 0.952 compared to DPIC. As a result, it reduced the average number of required examples from 6.8 to 5.4 as shown in the table above.

DPICED do not increase the number of conditional branches in the final program. Gulwani [5] mentioned that using the compatibility score to merge partitions practically leads to a small number of partitions, which is likely to be the correct program that can transform all records.

We compared the 5 approaches on all the 23 scenarios that SP had successfully transformed. We noticed that DPICED, DPIC, SPIC and SP can all achieve the same number of partitions when the system finished transforming the scenarios.

As agglomerative clustering with constraints can get stuck in local optima [21], where there are no more compatible partitions to be merged, even though other sequences of merging may lead to fewer partitions. However, as our approach starts over the partitioning in every iteration and refines the distance metric by learning form the accumulated constraints, it is likely that our approach can get out of the local optima in the new iteration.

VI. Conclusion and Future Work

This paper presents an approach for learning conditionals in a programming-by-example system. The approach takes advantage of previous constraints to learn a weighted Euclidean distance function to efficiently partition the examples. It then use the same distance function to incorporate unlabeled as training data to learn more accurate conditionals. The experiment results shows that the proposed approach significantly reduces the time and number of examples required to correctly transform a scenario. They also show that the approach has higher success rate compared to other approaches that can require extremely long time on certain scenarios.

In the future, we will experiment with some recent distance metric learning approaches and different clustering algorithms to see whether we can further improve the efficiency while still obtain a conditional statement with the smallest number of branches. We will also apply our approach to other application domains such as table layout transformation and XML format transformation.

References

[1] V. Raman and J. M. Hellerstein, "Potter's wheel: An interactive data cleaning system," in *VLDB*, 2001.

[2] D. F. Huynh and M. Stefano, *OpenRefine http://openrefine.org*.

[3] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld, "Programming by demonstration using version space algebra," *Mach. Learn.*, 2003.

[4] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: interactive visual specification of data transformation scripts," in *CHI*, 2011.

[5] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *POPL*, 2011.

[6] B. Wu, P. Szekely, and C. A. Knoblock, "Learning data transformation rules through examples: Preliminary results," in *IIWeb*, 2012.

[7] P. C. Mahalanobis, "On the generalized distance in statistics," *Proceedings of the National Institute of Sciences (Calcutta)*, vol. 2, 1936.

[8] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[9] N. Kushmerick, "Wrapper induction for information extraction," Ph.D. dissertation, 1997, aAI9819266.

[10] C.-N. Hsu and M.-T. Dung, "Generating finite-state transducers for semi-structured data extraction from the web," *Inf. Syst.*, 1998.

[11] I. Muslea, S. Minton, and C. Knoblock, "A hierarchical approach to wrapper induction," in *AGENTS, year = 1999,*.

[12] T. Lau, "Programming by demonstration: a machine learning approach," Ph.D. dissertation, University of Washington, 2001.

[13] D. Perelman, S. Gulwani, D. Grossman, and P. Provost, "Test-driven synthesis," in *PLDI*, 2014.

[14] L. Yang and R. Jin, "Distance metric learning: A comprehensive survey," *Michigan State Universiy*, vol. 2, 2006.

[15] B. Kulis, "Metric learning: A survey," *Foundations and Trends in Machine Learning*, vol. 5, no. 4, 2013.

[16] E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell, "Distance metric learning, with application to clustering with side-information," in *NIPS*, vol. 15, 2002.

[17] M. Bilenko, S. Basu, and R. J. Mooney, "Integrating constraints and metric learning in semi-supervised clustering," in *ICML*, 2004.

[18] K. Bade and A. Nurnberger, "Personalized hierarchical clustering." in *WI*, 2006.

[19] H. Zhao and Z. Qi, "Hierarchical agglomerative clustering with ordering constraints," in *WKDD*, 2010.

[20] L. Zheng and T. Li, "Semi-supervised hierarchical clustering," in *ICDM*, 2011.

[21] I. Davidson and S. S. Ravi, "Using instance-level constraints in agglomerative hierarchical clustering: Theoretical and empirical results," *Data Min. Knowl. Discov.*, 2009.

[22] B. Wu, P. Szekely, and C. A. Knoblock, "Minimizing user effort in transforming data by example," in *IUI*, 2014.