ITERATIVELY LEARNING DATA TRANSFORMATION PROGRAMS FROM EXAMPLES

by

Bo Wu

A Dissertation Presented to the

FACULTY OF THE USC GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(COMPUTER SCIENCE)

December 2015

Dedication

To my father Yongwei Wu and my mother Rufen Liu, for their love and support

Acknowledgments

First and foremost, thanks to my parents. Their unconditional support allows me to finish this long endeavor. Thanks to Chiao-Yu. Her love and company give me the strength to get through all those hard times.

I would like to thank my advisor, Craig A. Knoblock, who taught me how to conduct research. During my PhD study, I met many difficulties, such as identifying research problems, writing papers, giving presentations and so on. I was deeply frustrated by my research progress at the beginning. Craig's encouragement allows me to continue my research. Talking with Craig is always very helpful. Once I leave ISI, I think one of things I would miss most is talking with Craig. Besides research, Craig also sets a good example for me on many aspects of life, which I will try my best to simulate in future.

Many thanks to my committee members: Daniel O'Leary, Cyrus Shahabi, Yan Liu and Jose Luis Ambite. Their guidance kept me on the right track so that I could successfully finish my thesis in time. Their patience and kindness allowed my thesis proposal and defense to go smooth.

I want to thank Pedro Szekely. He led me into the user interface area and patiently taught me how to develop a research prototype within a large team. These experiences opened a new door for me and allowed me to pay more attention to the user part. I also want to thank Yao-Yi Chiang from whom I learnt a lot of interesting work and tools in

geospatial information integration. He showed me the interesting applications in that field.

Finally, I want to express my thanks to my fellow students of information integration group at ISI: Mohsen, Jeon-Hyung, Suradej, Yoon-Sik, Jason and Hao. You are always ready to help. Hanging out with you guys is fun, which makes this long PhD journey more joyful. Talking with you also helps me to learn many interesting things and appreciate the diversity of this world.

This research is based upon work supported in part by the National Science Foundation under Grant No. 1117913. This research was also supported in part by the Intelligence Advanced Research Projects Activity (IARPA) via Air Force Research Laboratory (AFRL) contract number FA8650-10-C-7058.

Contents

List of Tables

List of Figures

Abstract

Data transformation is an essential preprocessing step in most data analysis applications. It often requires users to write many trivial and task-dependent programs, which is time consuming and requires the users to have certain programming skills. Recently, programming-by-example (PBE) approaches enable users to generate data transformation programs without coding. The user provides the PBE approaches with examples (input-output pairs). These approaches then synthesize the programs that are consistent with the given examples.

However, real-world datasets often contain thousands of records with various formats. To correctly transform these datasets, existing PBE approaches typically require users to provide multiple examples to generate the correct transformation programs. These approaches' time complexity grows exponentially with the number of examples and in a high polynomial degree with the length of the examples. Users have to wait a long time to see any response from the systems when they work on moderately complicated datasets. Moreover, existing PBE approaches also lack the support for users to verify the correctness of the transformed results so that they can determine whether they should stop providing more examples.

To address the challenges of existing approaches, we propose an approach that generates programs iteratively, which exploits the fact that users often provide multiple examples iteratively to refine programs learned from previous iterations. By collecting and accumulating key information across iterations, our approach can efficiently generate the new transformation programs by avoiding redundant computing. Our approach can also recommend potentially incorrect records for users to examine, which can save users effort in verifying the correctness of the transformation results.

To validate the approach in this thesis, we evaluated IPBE, the implementation of our iterative programming-by-example approach, against several state-of-the-art alternatives on various transformation scenarios. The results show that users of our approach used less time and achieved higher correctnesses compared to other alternative approaches.

Chapter 1

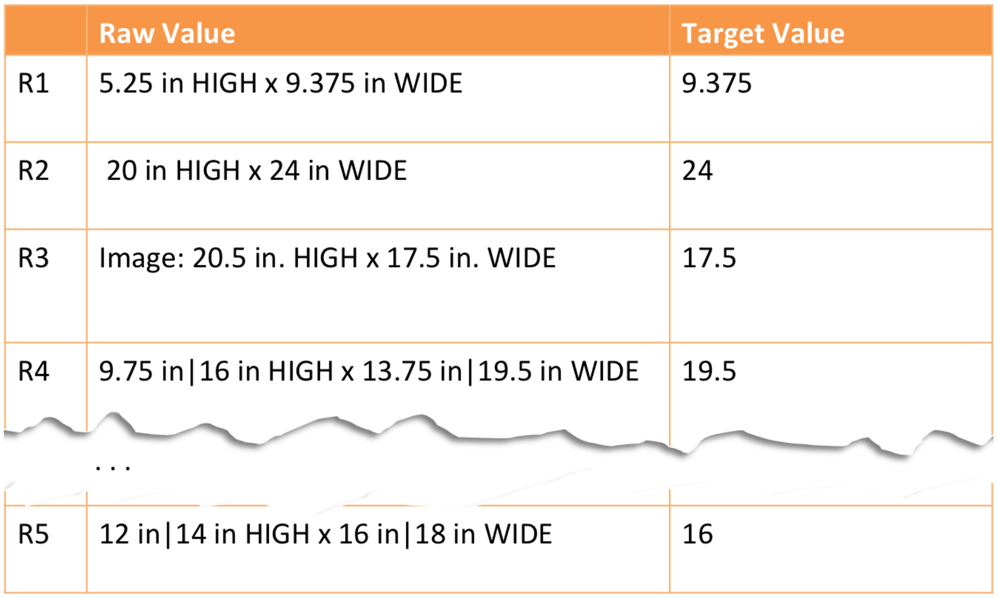
Introduction

This chapter introduces the key challenges of applying PBE approaches to perform data transformation. I first motivate and define the problem. I then briefly describe our approach to the problem and summarize our contributions. Finally, I present the outline of the thesis.

1.1 Motivation and problem statement

Data transformation converts the data from the source format to the target format, which is an essential step before utilizing the data. As the transformation is often task-dependent, the data practitioner usually writes scripts for different tasks, which can be error-prone and labor intensive.

Recently, programming-by-example (PBE) approaches [Lau et al., 2003; Huynh et al., 2008; Kandel et al., 2011; Gulwani, 2011] have proven to be effective in generating transformation programs for simple scenarios without coding. These approaches are based on certain domain specific language (DSL) designed to express common data transformation operations. They only require examples (input-output pairs). They can then generate programs that are consistent with examples, which means these program can generate expected outputs for the corresponding inputs specified in the examples. FlashFill [Gulwani, 2011], which is an example PBE system, is already integrated into Excel 2013 to help users transform the data. For example, the dimensions of artworks in Figure 1.1 mix values for all degrees into one cell. The user wants to put each degree in its own column. Taking width for instance, the target values are shown in the right column in Figure 1.1. To transform the data, the user enters the target value (width) "9.375" for the first entry. The system generates a program and applies that program to the rest of the data. If the user finds any entry is transformed incorrectly, she provides a new example for that entry to regenerate the program. For instance, as the 4th entry has both the painting width and frame width, the program learned from the first example fails to transform the input of this unseen format. The user inputs "19.5" to teach the system that she only wants the frame width, which is shown after the vertical bar. The system regenerate the program and applies it to the rest of entries. The user keeps providing examples until she determines the results are correct.



	Raw Value	Target Value
R1	5.25 in HIGH x 9.375 in WIDE	9.375
R2	20 in HIGH x 24 in WIDE	24
R3	Image: 20.5 in. HIGH x 17.5 in. WIDE	17.5
R4	9.75 in|16 in HIGH x 13.75 in|19.5 in WIDE	19.5
	. . .	
R5	12 in|14 in HIGH x 16 in|18 in WIDE	16

Figure 1.1: An example for a data transformation scenario

However, there are 3 challenges that need to be addressed to make PBE approaches more practical. First, a dataset can have various formats and the user is only willing to provide a few examples, as seen in Figure 1.1. The approach should learn to recognize these formats from few examples accurately and then perform the transformation accordingly.

Second, the users need to see the results on the fly so that they can provide additional examples if necessary. The approach should generate the programs in real time. However, the current approaches' computational complexity grows exponentially in the number of examples and a high degree polynomial in the size of each example [Raza et al., 2014]. In previous work, users have to wait a long time, when they work on the scenarios that require long or many examples.

Third, users are often overconfident with the correctness of the results especially on large datasets [Ko et al., 2011]. There are usually thousands of entries being transformed. It is hard for users to manually examine whether all the entries are transformed correctly. Users also do not like to invest much time in examining the results.

Given the challenges presented by current PBE approaches, this thesis focus on solving the problem of how to **efficiently generate correct transformation programs for data with heterogeneous formats using minimal user effort.**

1.2 Proposed approach

To address the problem, *we exploit the fact that users interact with the PBE system iteratively*. The interaction between the user and a PBE system often contains multiple iterations as shown in Figure 1.2. In each iteration, the user examines the records and provides examples for the records with incorrect results. *A record* here consists of raw input and the transformed result. After obtaining the examples from the user, the PBE system synthesizes a transformation program and applies this program to transform all records. The transformed records are presented to the user for examination. If the user

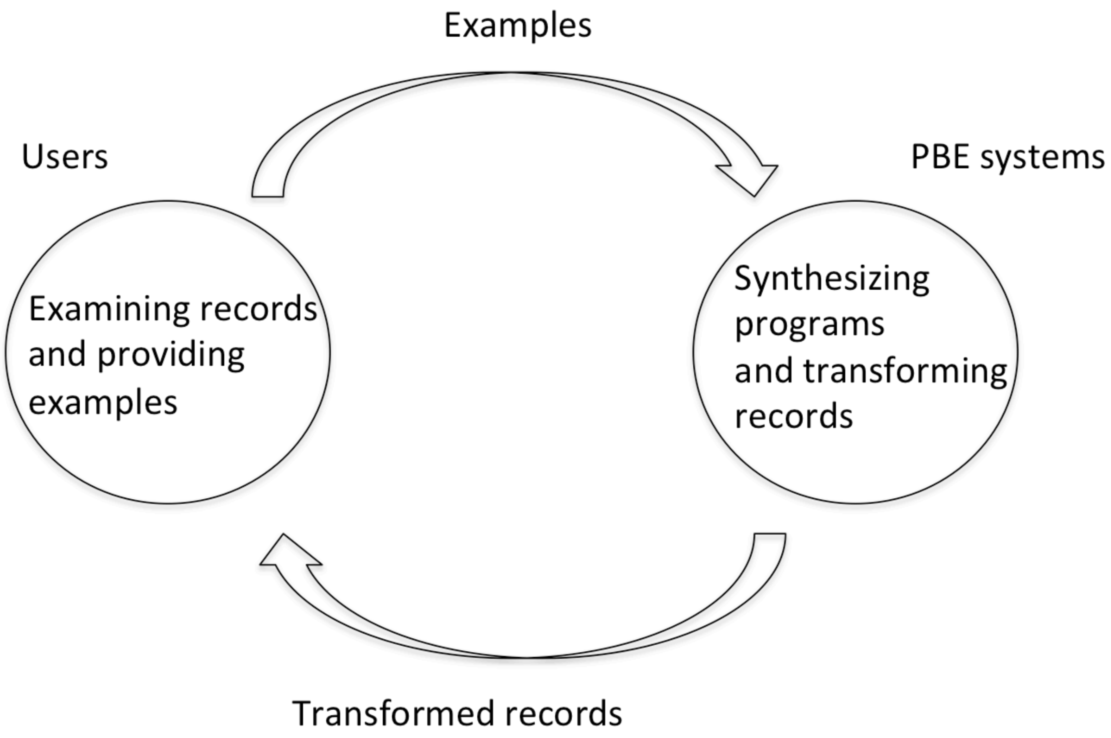

Examples
Users
PBE systems
Examining records and providing examples
Synthesizing programs and transforming records
Transformed records

Figure 1.2: Interaction between users and PBE systems

identifies any record transformed incorrectly, she can provide a new example and start a new iteration. For any non-trivial datasets, users often provide more than 3 examples before finishing transforming. For example, in Figure 1.1, the user keeps providing new examples for the entries with incorrect results to regenerate a transformation program that is consistent with all given examples.

The intuition of our approach is to identify and collect certain key information from previous iterations and utilize the information to benefit the current iteration. To understand how the information from previous iterations can improve the performance of the existing PBE system [Gulwani, 2011], I briefly describe the architecture of our iterative programming-by-example approach (IPBE) shown in Figure 1.3 and its key modules. These key modules are based on the previous approach [Gulwani, 2011], which will be described in Chapter 2 in more detail. In our approach, a user is asked to provide

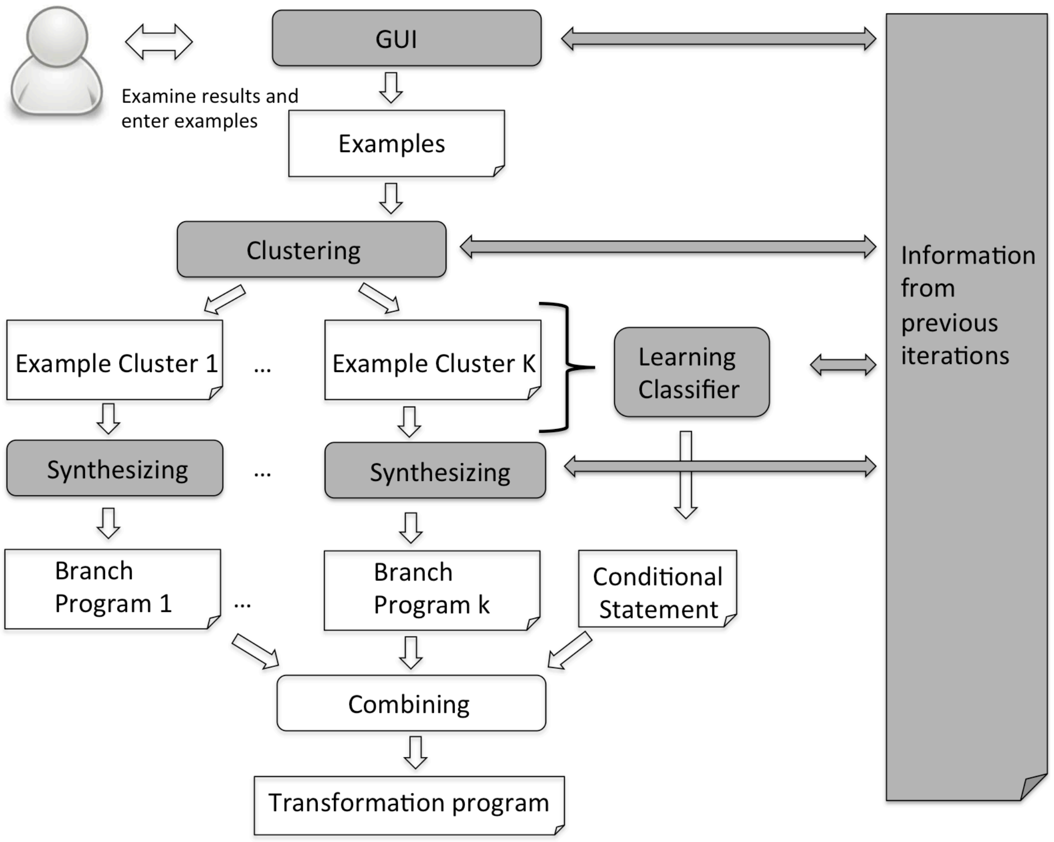


Figure 1.3: Approach overview

examples in the GUI. It first clusters the examples into multiple clusters, where each cluster corresponds to a specific format. It learns a classifier (conditional statement) to recognize these different formats. It then synthesizes a branch transformation program for each cluster to handle the specific format. Finally, the PBE approach combines the conditional statement with branch programs to generate the final data transformation program.

The three key components are (1) clustering examples and learning conditional statements (2) synthesizing branch programs and (3) a user interface for providing examples. The approach that we used to improve these modules' performance through utilizing information from previous iterations are listed below.

Efficiently clustering examples and accurately learning the classifiers: Learning conditional statements (classifiers) requires clustering the examples into compatible clusters. A compatible cluster means there is at least one branch program that is consistent with the examples in that cluster. To know whether a group of examples are compatible, the previous approach [Gulwani (2011)] tries to generate programs for this group of examples to see if there exists a consistent program. Thus, verifying the compatibility of examples in a cluster is computationally expensive. What is worse, the previous approach needs to verify the compatibilities of a large number of clusters before identifying the final clustering. We developed an approach that learns a distance metric from cluster information of previous iterations. This distance metric will put the compatible examples closer so that they can form a cluster while putting the incompatible examples far away. Using the distance metric, our approach can put the compatible examples into one cluster without verifying the compatibilities of a large number of clusters. The distance metric can also be used to incorporate unlabeled data (the records that are not used as examples) as the training data for learning a classifier as the conditional statement. By expanding the training data using the unlabeled data, we can learn a more accurate classifier than by just relying on few examples.

Efficiently synthesizing the branch transformation program: Only a small portion of the learned program changes after the user provides a new example to refine the program. Our approach can identify the incorrect subprograms that require modification after users provide new examples. Our approach decomposes the new example to generate the expected input-output pairs for subprograms. It then compares the execution results of subprograms of the current program with the expected outputs to identify the subprograms that cannot generate expected outputs. After identifying the incorrect subprograms and their expected outputs, our approach uses the new example to refine the hypothesis space that are used to generate the incorrect subprogram. It generates the correct subprograms from the refined hypothesis space. It then replaces the previous incorrect subprograms with new ones to generate the new program.

Guiding users to obtain correct results with minimal user effort: To handle large dataset, our approach allows users to focus on a small sample. When the records in the small sample are all transformed correctly, the entire dataset can obtain a correctness satisfying the user's requirement. My approach also learns from past transformation results to recommend the records that potentially have incorrect results. Users examine these recommended records. They can enter examples for incorrectly transformed records or confirm the correctness of certain recommended records to refine the recommendation. The approach also requires users to examine a minimal set of the records to ensure that the users are not too confident with their results to carefully check the results.

1.3 Thesis Statement

Through collecting and leveraging the information generated across iterations, our programming-by-example approach can efficiently generate programs that can correctly transform data with heterogeneous formats using minimal user input.

1.4 Contributions of the Research

The goal of this research is to develop a practical programming-by-example data transformation system. To be specific, our research has the following contributions:

- efficiently learning accurate conditional statements by exploiting information from previous iterations

- incrementally synthesizing branch transformation programs efficiently by adapting programs from the previous iteration

- maximizing the user correctness with minimal user effort on large datasets by recommending potential incorrect records.

1.5 Outline of the Thesis

The rest of this proposal is organized as follows. Chapter 2 describes the previous work and related basic concepts. Chapter 3 describes the approach to learn conditional statements iteratively. Chapter 4 presents the approach that iteratively generates new programs by reusing previous subprograms. Chapter 5 discusses the approach to help users verify the correctness of the results with minimal effort on large datasets. Chapter 6 reviews all the related work. Finally, chapter 7 concludes this research and identifies areas for future research.

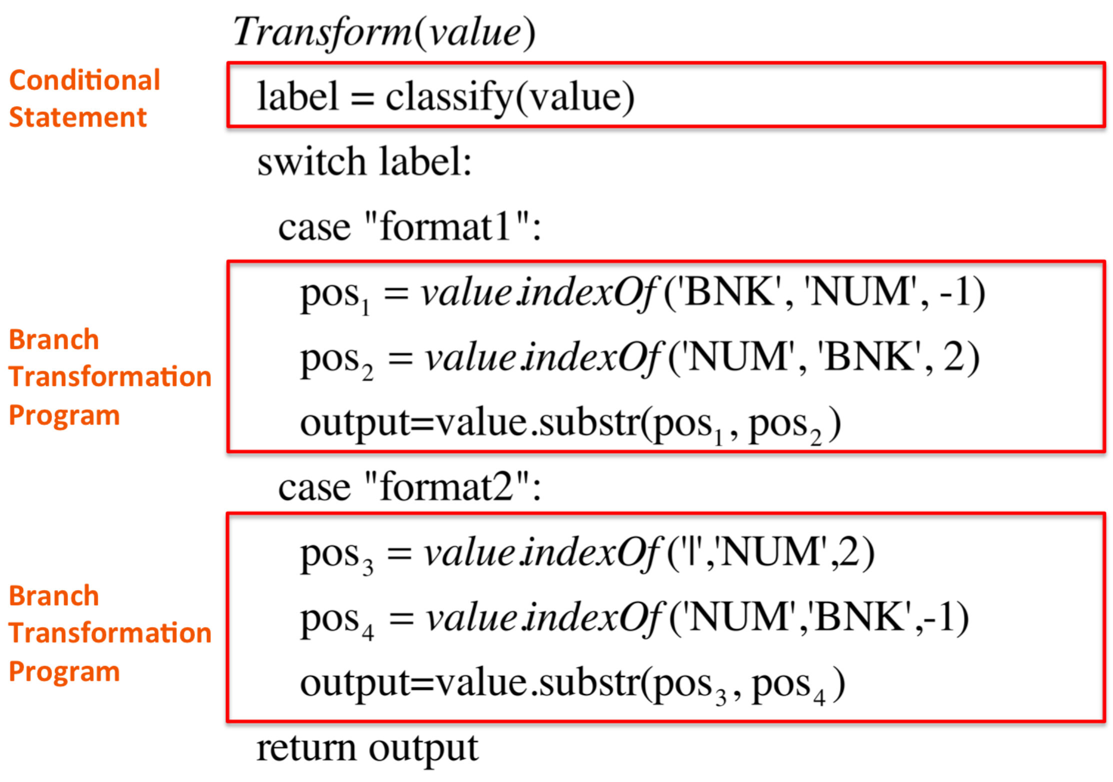
Chapter 2

Previous Work

Our approach is built on the state-of-the-art PBE system [Gulwani, 2011], which exploits the version space algebra like many other program induction approaches [Lau et al., 2003]. It defines a domain specific transformation language (DSL), which supports a restricted, but expressive form of regular expressions that also includes conditionals and loops. The approach synthesizes transformation programs from this language using examples.

To better understand the structure of the generated transformation program, we use a different representation of the transformation program from Gulwani's original notations without changing its meaning. The transformation program learned from the examples in Figure 1.1 is shown in Figure 2.1.

The program recognizes the format of the input using the *classify* function as seen in Figure 2.1. It then performs the conditional transformation using the *switch* function based on the recognized format. It has a *branch transformation program* for each specific format. The branch program is essentially a concatenation of several segment programs as $branch = substr_1 + substr_2 + ...$ A *segment program* outputs a substring, which is defined as $substr = const|substr(p_s,\ p_e)|loop(w,\ branch)$. The segment program can (1) be a constant string ($const$), (2) extract a substring ($substr$) between two positions specified by position programs p_s and p_e respectively or (3) a loop program ($loop$) that executes a branch program iteratively where w controls the start point of the loop. The w is passed as an offset from the beginning into the position programs in the loop body, which is shown in an example later. The *position program* locating



```
Transform(value)
 label = classify(value)
 switch label:
  case "format1":
    pos_1 = value.indexOf('BNK', 'NUM', -1)
    pos_2 = value.indexOf('NUM', 'BNK', 2)
    output=value.substr(pos_1, pos_2)
  case "format2":
    pos_3 = value.indexOf('|','NUM',2)
    pos_4 = value.indexOf('NUM','BNK',-1)
    output=value.substr(pos_3, pos_4)
 return output
```

Figure 2.1: An example transformation program

a position in the input is defined as $p = indexOf(leftcxt,\ rightcxt,\ c)|number$. It can be specified using (1) an absolute number ($number$), or (2) a position with the context specified by $(leftcxt,\ rightcxt,\ c)$. Both $leftcxt$ and $rightcxt$ are a sequence of tokens, which specifies the left and right context of a position correspondingly. c refers to the c-th occurrence of the position with the required context. The set of tokens used in the approach is as follows. 'ANY' can represent any token. 'WORD' ([a-zA-Z]+) represents a sequence of alphabetical letters. 'UWRD' ([A-Z]) stands for a single upper case letter. 'LWRD' ([a-z]+) means a sequence of lowercase letters. 'NUM' ([0-9]+) represents a sequence of digits. 'BNK' is a blank space. Besides these tokens, all punctuation are also tokens. For example, in Figure 2.1, pos_1 is specified as (BNK, NUM, -1). "-1" means the first appearance of a position with the required context when scanning backwards from the end of the input. Hence, (BNK, NUM, -1) refers to the first position whose left is a blank space and whose right is a number, when scanning from the end of the input. An example for loop program is $loop(2, substr(indexOf(WORD,\ ANY,\ 2{+}i), indexOf(ANY,\ WORD, 2{+}i)){+}``,")\ i \in [0,\ 1,\ ...]$. The body of the loop is a concatenation of two segment programs: (1) one is a segment program where the position programs start matching at the $(2{+}i)$-th occurrence and (2) a constant string ",". The loop program essentially extracts all WORD tokens except the first one and appends a comma after each word. Here w is 2. The $w + i$ becomes $2 + i$, which specifies the position program should start locating positions from the second occurrence and continue searching for the next occurrence with the desired context until the position program fails to locate a position.

2.1 Generating the transformation program

Synthesizing the transformation program as seen in Figure 1.3 has two essential steps: (1) clustering the examples into partitions to learn conditional statements and (2) synthesizing the branch program for each partition. *The partition is a hypothesis space of branch transformation programs derived from the examples belonging to the partition.*

2.1.1 Synthesizing the branch transformation program

To generate a branch transformation program, Gulwani's [Gulwani, 2011] approach follows these steps:

First, it creates all *traces* of the examples. To create the traces, it segments the outputs into all possible ways. From the original input, it then generates these segments, independently of each other, by either copying substrings from original values or inserting constant strings. Two trace examples are shown in Figure 2.2. The outside rectangle shows that the "9.375" can be directly extracted from the input. The other trace depicted

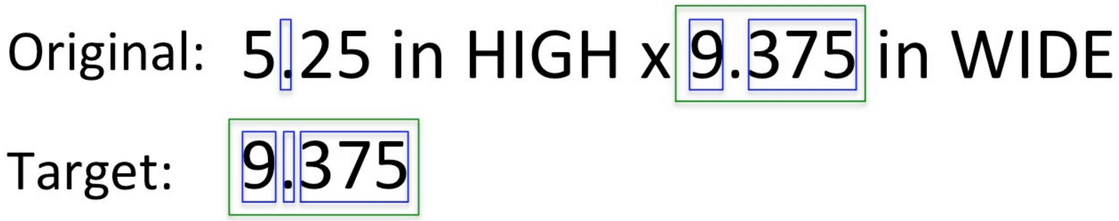


Figure 2.2: One trace example of an input-output pair

using three small rectangles shows that the output is a concatenation of three substrings where the period is extracted from the first period in the input.

Definition 1 *Traces: traces are the computational steps executed by a program to yield an output from a particular input [Kitzelmann and Schmid, 2006]. A trace here defines how the output string is constructed from a specific set of substrings from the input string.*

Second, it derives hypothesis spaces from the traces. A *hypothesis space* defines the set of possible programs in the DSL that can transform the inputs to the outputs. The hypothesis space is stored using a direct acyclic graph (DAG), where the nodes in the DAG correspond to the *position hypothesis spaces* that contain the position programs. The edges correspond to the *segment hypothesis spaces* that contain the segment programs. An example of the hypothesis space derived from the traces in Figure 2.2 is shown in Figure 2.3. There are four nodes corresponding to 4 position hypothesis spaces, as the output in Figure 2.2 can be split into at most 3 different segments. S_1 and S_4 are the start and end positions. Each path from S_1 to S_4 is a concatenation of multiple edges representing the programs consisting of different segments such as the path ($S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$) dictates the program can consist of three segment programs. The second segment hypothesis space ($S_2 \rightarrow S_3$) contains all segment programs generated by filling the p_s and p_e in the $substr(p_s,\ p_e)$ with any pair of position programs from S_2 and S_3 position hypothesis spaces respectively. The segment space also

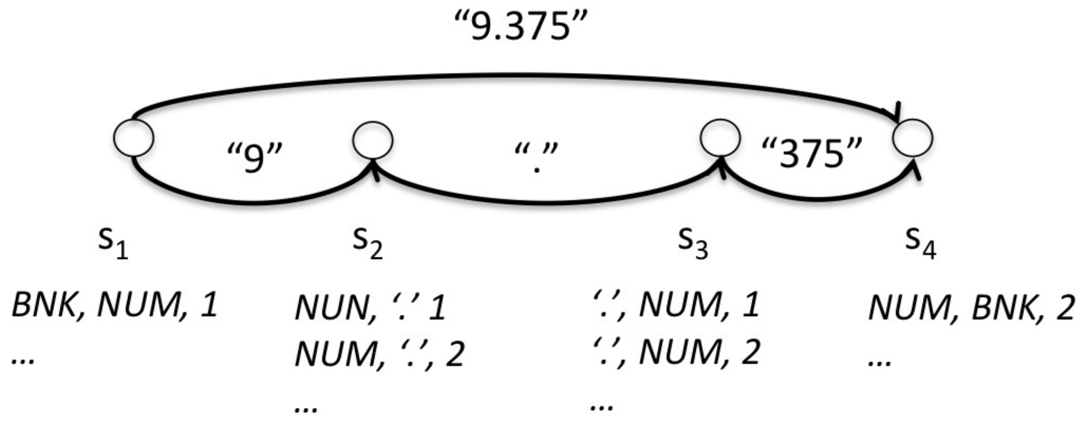


Figure 2.3: An example of the hypothesis space

contains the constant text as "." shown in Figure 2.3. There is no guarantee that all programs in the hypothesis space are consistent with examples. For example, $(NUM,'.',2)$ from S_2 and $('.',NUM,1)$ from S_3 can not form a valid segment program. The output of the start position program will be larger than the output of the end position program, as the program from S_2 locates the beginning of the second period whereas the program from S_3 identifies the end of the first period. Finally, if there are multiple examples, the approach creates a hypothesis space for each example and merges these hypothesis spaces to generate a common hypothesis space for all examples.

Finally, the programs in the hypothesis space are partially ordered based on their simplicity which is measured using a set of pre-defined heuristics. The simpler programs are generated first as they are more likely to be correct based on Occama's principle. For example, the approach generates the program with fewer segments earlier. As mentioned earlier, since not all programs in the hypothesis space are consistent with examples, the approach uses a generate-and-test approach. The approach keeps generating programs in the order of their simplicity and returns the first program that is consistent with all examples.

2.1.2 Learning the Conditional Statement

To cluster the examples, the approach initially treats each example as its own partition. It then chooses two compatible partitions to merge each time. The process iterates until no more compatible partitions are available. The two partitions p_1, p_2 are *compatible* if there is at least one branch transformation program, which is consistent with all the examples of the two partitions. The $comp(p_1, p_2) = 1$ if two partitions p_1 and p_2 are compatible, otherwise, $comp(p_1, p_2) = 0$. With the definition of $comp$, the approach uses a utility function called *compatibility score* to choose two partitions with the highest score to merge in each round.

$$CS(p_1, p_2, P) = (CS_1(p_1, p_2, P), CS_2(p_1, p_2)) \tag{2.1}$$

$$CS_1 = \sum_{p_k \in P, k \neq 1, k \neq 2} z(p_1, p_2, p_k) \tag{2.2}$$

$$z(p_1, p_2, p_k) = \begin{cases} 1 \quad if \ (comp(p_1, p_k) = comp(p_2, p_k) \\ \qquad = comp(Merge(p_1, p_2), p_k)) \\ 0 \qquad\qquad Otherwise \end{cases} \tag{2.3}$$

$$CS_2 = \frac{Size(Merge(p_1, p_2))}{Max(size(p_1), size(p_2))} \tag{2.4}$$

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 ($Merge(p_1, p_2)$) 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 is actually a hypothesis space derived from the examples. Merging two partitions requires intersecting the two hypothesis spaces of the two partitions and verifying whether there exists a branch program that is consistent with the examples from both partitions through the generate-and-test strategy. It is especially computationally expensive when the two partitions are incompatible, as it requires evaluating all the programs in that intersected space.

The approach is essentially an agglomerative clustering method. It greedily selects the most compatible partitions based on the compatibility score in Equation 2.1 and then merges them together until no more compatible partitions are available.

Chapter 3

Learning Conditional Statements

Much real world data has multiple formats and a single branch program cannot transform 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 Figure 1.1, 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 chapter, we solve the problem of *learning expressive conditional statements efficiently with few user provided examples*. As shown in Figure 1.3, to learn the conditional statement using the given examples, the current state-of-the-art approach by Gulwani, 2011 first (1) partitions the examples into several clusters where examples in one cluster can be transformed by the same branch program and then (2) learns a classifier to distinguish between these formats. The partitioning of the examples also depends on the transformation language defined in each PBE system, whether several examples can be put into the same cluster depends on whether the system can find a conversion program in that language. For example, given the first four rows as examples in Figure 1.1, the partition algorithm generates two partitions: R1, R2 and R3 as one partition and R4 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 need to see the results on the fly. The systems only have limited time to generate the conditional statement. However, PBE approaches must identify one partitioning among the many possible 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, there are many ways to classify examples. 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 concise interpretation that is consistent with most of the records, as the Occam's principle states the simplest interpretation tends to be correct [Gulwani, 2011]. 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 branch 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:

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

3.1 Construct Conditional Transformations

Our approach iteratively learns conditional statements for PBE systems as shown in Algorithm 1. The approach in every iteration can be broken into 4 high-level steps: (1) partitioning the examples, (2) learning the classifier, (3) generating branch programs for all the partitions and (4) generate the final transformation program. We focus on the first steps in the chapter. To take advantage of the information obtained from previous iterations, the approach also maintains two sets of constraints: (1) cannot-merge constraints and (2) must-merge constraints. In each iteration, these constraints are used to improve the performance of partitioning and classifier learning.

Definition 2 ***Cannot-merge constraints****: each cannot-merge constraint in this set contains a unique group of examples that are not compatible together.* ***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 (line 1). 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. The approach clears M to allow different partitions to be formed in the new iteration. Second, the approach learns a classifier for the partitions (line 2). It uses the examples along with the unlabeled inputs to train the classifier. This classifier serves as the conditional statement. Third, the approach generates branch programs that are consistent with the examples in each partition (line 3). Finally, our approach combines the partition transformation programs $prog_1, prog_2, ...$ with the conditional statement g to create the transformation program (line 4).

Algorithm 1: Create Transformation Program

```
  Input: examples $E = \phi$, unlabeled data $U$, cannot-merge constraints $R = \phi$,
         must-merge constraints $M = \phi$, distance metric $D_w$
  data preprocessing
  while user provides a new example do
      $E = E \cup e$
      $M = \phi$
1     $partitions$=partitioning($R, M, E, U, D_w$)
2     $g$=learnClassifier(partitions, $D_w$, $U$)
      for $p_i$ in partitions do
3         $prog_i$=learnBranchTransformation($p_i$)
4     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 transformation programs are generated. It only needs to know whether the PBE system can successfully generate transformation programs from a group of examples.

3.1.1 Data Preprocessing

We represent the records in two different forms. One representation (token sequence) is used to derive the branch transformation program. The other representation (feature vector) is used for the third-party module to learn the classifiers.

Our approach first tokenizes both the input and output of the examples into token sequences. These sequences are then used to derive the branch programs. Meanwhile, our approach also converts the inputs of all records into feature vectors. The features can be categorized into two types: (1) counts of tokens and (2) the average indexes of tokens in the sequence. To create the features, we use both token types and their contents. As shown in Example 1, “H”, “.”, etc. are used as features. We collect these tokens with their contents from all records. We only keep the frequently appearing tokens by discarding some rare tokens that appear in less than 10% of the entries.

With the initial feature vector, we decorrelate the features by removing the features whose values on all records are a single linear transformation of the values of another feature. If there is a set of features whose values on all records are a single linear transformation of each other, we only keep one feature of this set to reduce the dimensionality of the feature vector. We then normalize the features into [0, 1] individually.

Example 1 *As shown in Figure 3.1, a string "9.75 in|16 in HIGH" can be tokenized as "START NUM(9) Period(.) NUM(75) BNK LWRD(in) VBAR(|) NUM (16) BNK LWRD(in) BNK UWRD(H) UWRD(I) UWRD(G) UWRD(H) END". The counts of different tokens: NUM: 3, UWRD: 4, LWRD: 2, etc. The NUM_pos is the feature capturing the average index of its corresponding token (NUM) in the record. Its value (4) is the rounded up average position of the three NUM tokens (1, 3 and 7). Our approach uses these token counts and their average indexes to create the initial feature vector as shown in the third row of Figure 3.1. The feature vector after decorrelation and normalization is shown in the forth row. The number of features shown in the feature vector is smaller than the number of features in the initial feature vector, as certain rare tokens are discarded and some features are removed after decorrelation.*

3.1.2 Partition Algorithm

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

String	9.75 in|16 in HIGH
Tokens	START NUM(9) Period(.) NUM(75) BNK LWRD(in) VBAR(|) NUM (16) BNK LWRD(in) BNK UWRD(H) UWRD(I) UWRD(G) UWRD(H) …

Initial feature vector

NUM	NUM_pos	LWRD	LWRD_pos	.	._pos	H	H_pos	…
3	4	2	7	1	2	2	13	0

Feature Vector

LWRD	NUM	.	:	|	=	NUM_pos	…
0.21	0.29	0.14	0.21	0.07	0.07	0.61	…

Figure 3.1: Token Sequence and Feature Vector for a String

The 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 examples given by the user may change the previous partitioning, M is set to empty at the beginning of each iteration. The $merge(p_i,\ p)$ merges two partitions (p_i and p_j) into one partition. If a partition does not contain a program that is consistent with all its examples, the partition is noted as ϕ.

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

Our algorithm continues 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 rather than the centroid-based distance, as the examples of a partition

```
Algorithm 2: Partition Algorithm
Input: examples $E$, unlabeled data U, cannot-merge constraints $R$, must-merge
       constraints $M = \phi$, distance metric $D_w$
Output: partitions P

create a partition $p_i$ for each $e_i \in E$
while $\exists p_i, p_j \in P$ merge($p_i, p_j$) $\neq \phi$ do
1   use $D_w$ to find two closest partitions $p_x, p_y$
2   $p_z = merge(p_x, p_y)$
3   if $p_z = \phi$ then
4       $R = R \cup \{\{e_i \mid e_i \in p_x \ \vee \ e_i \in p_y\}\}$
5       Learn distance metric $D_w$ using $U$, $R$ and $M$
    else
6       $P = P \setminus \{p_x, p_y\}$
7       $P = P \cup \{p_z\}$
8       update $M$ with $P$
Return P
```

in the feature space can form irregular and non-spherical shapes. 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 merger cannot generate a valid partition ($p_z = \phi$) (line 3), 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 (line 4). It then uses the updated cannot-merge and the must-merge constraints to refine the distance metric D_w (line 5), which is described in the next section. With the updated distance metric, it finds the two closest partitions without contradiction to the constraints.

If the merger succeeds, the previous two partitions are removed from partitions P and the new partition (p_z) is inserted into P (line 6 and 7). It also updates the must-merge constraints M using the current clusters that have at least two examples (line 8). The examples in the partition ($\{e_i | e_i \in p\}$) form one must-merge constraint.

Example 2 *Suppose the user provides the first four records in Figure 1.1 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, R4 and R3). Suppose there are three partitions in the current iteration: (1) R1 and R3 are in one partition (R1, R3), (2) R4 is in the second partition and (3) R2 is in the third partition. Thus, R1 and R3 constitute one must-merge constraint. Because R1 and R3 in the must-merge constraints have different "." counts (two "." in R1 and zero "." in R3). This indicates the examples can have different numbers of "." in the same partition. At the same time, the cannot-merge constraint with R1, R4 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 module 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 R2 record into the same partition as R1 and R3.*

3.1.3 Distance Metric Learning

Our approach learns a weighted Euclidean distance that is a special case of the Mahalanobis distance [Mahalanobis, 1936]. This weighted Euclidean distance is used to select partitions to merge. The weights of the features are used to capture the importance of features in calculating the distance. The higher the weight for a certain feature, the larger the 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 of two records and w_i is the weight for the i-th feature. The $D_w = (w_1,\ w_2,\ ...)$ is the vector containing all the weights. We use the weighted Euclidean distance for two reasons: (1) lower cost to calculate than Mahalanobis distance as our program interacts with users on the fly, which requires a quick response, (2) the weights in the distance metric are more interpretable.

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

$$\arg\min_{w>0} \sum_i \|x_i - e_{x_i}\|_w + a \cdot g(w) - b \cdot h(w) \tag{3.1}$$

$$g(w) = \ln(\sum_{X_m} \sum_{x_i, x_j \in X_m, i \neq j} \|x_i - x_j\|_w) \tag{3.2}$$

$$h(w) = \ln \sum_{X_r} \max_{x_i, x_j \in X_r} \|x_i - x_j\|_w \tag{3.3}$$

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:

$$d(x, p) = \min\{\left\|x - e_i^{input}\right\|_w \mid e_i \in p\} = \left\|x - e_x^{input}\right\|_w . \tag{3.4}$$

Here, x is the input of an unlabeled record, p is the partition and e_i^{input} is the input of an example e_i in partition p. The distance between an unlabeled record and a partition can also be represented as $\|x - e_x^{input}\|_w$, which is the weighted Euclidean distance between x and its closest example e_x^{input} in p.

The $g(w)$ is the penalization term corresponding to the must-merge constraints. A must-merge constraint X_m 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.

The $h(w)$ 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 together. However, a subset of the examples can still be in the same partition. For example as shown in Figure 1.1, R1, R4 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 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 3.1 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.

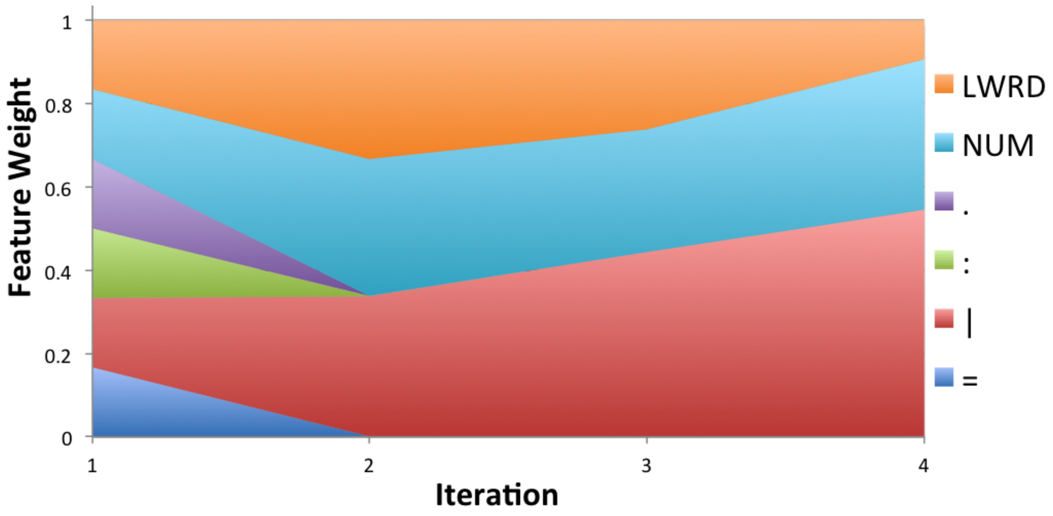


Figure 3.2: Weights of Different Features Change Over Iterations

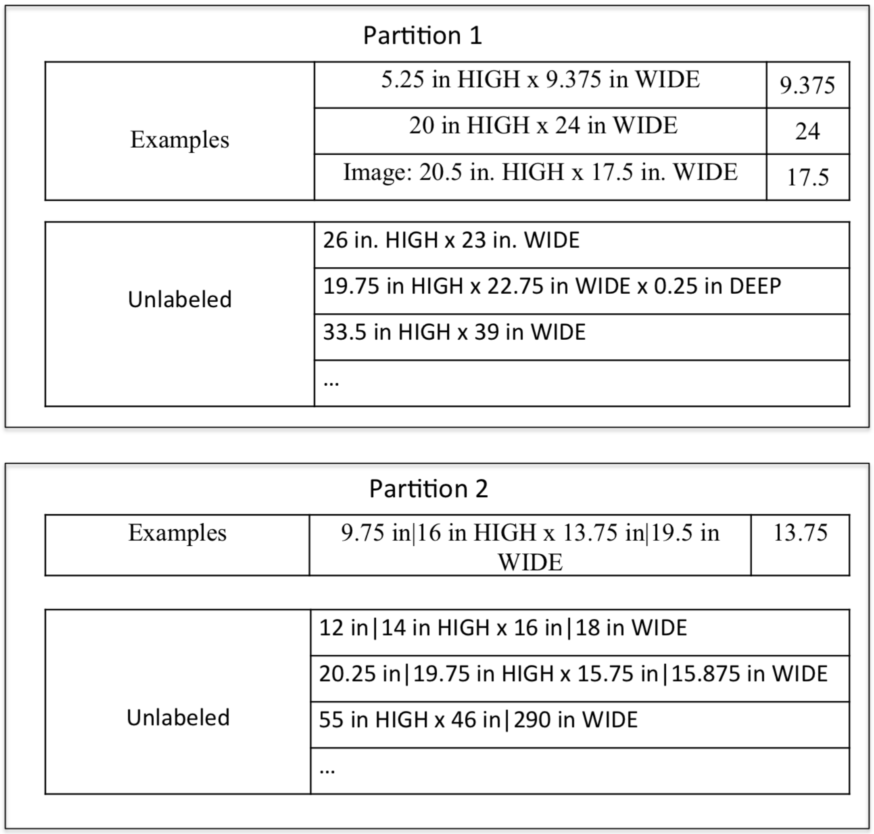
Example 3 *Figure 3.2 shows that the weights of features change as the iteration number increases. The size of each color corresponds to the weight of different features. At the beginning, we can see all the features have the same weight. As the approach learns from the constraints accumulated across iterations, the "|" feature (count of "|") weights more and the "." weights less.*

3.1.4 Learning the Classifier

Our approach 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 examples 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 3.3. 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 original records from both examples and unlabeled entries to train a SVM [Chang and Lin, 2011] classifier as the conditional statement for the transformation program.

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.

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 (u_i) and the partition it belongs to. The distance between an input and a partition (*getDistance*) is defined in Equation 3.4. The algorithm then computes the distances (d_2) between the input u_i and all other partitions p_j (line 1). If any distance d_2 is close to d_1 and the difference is within a threshold ε (line 2), 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



Partition 1

Examples	5.25 in HIGH x 9.375 in WIDE	9.375
	20 in HIGH x 24 in WIDE	24
	Image: 20.5 in. HIGH x 17.5 in. WIDE	17.5

Unlabeled	26 in. HIGH x 23 in. WIDE
	19.75 in HIGH x 22.75 in WIDE x 0.25 in DEEP
	33.5 in HIGH x 39 in WIDE
	...

Partition 2

Examples	9.75 in|16 in HIGH x 13.75 in|19.5 in WIDE	13.75

Unlabeled	12 in|14 in HIGH x 16 in|18 in WIDE
	20.25 in|19.75 in HIGH x 15.75 in|15.875 in WIDE
	55 in HIGH x 46 in|290 in WIDE
	...

Figure 3.3: The examples and the unlabeled data in each partition

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 [Chang and Lin, 2011] classifier.

Algorithm 3: Filter the unlabeled data in each partition

```
Input: partitions $P$, unlabeled data $U$
for $p_i \in P$ do
    for $u_i \in p_i$ do
1       $d_1$=getDistance($u_i$, $p_i$)
        for $p_j \in P$ and $p_j \neq p_i$ do
            $d_2$ = getDistance($u_i$, $p_j$)
2           if $(d_2 - d_1)/d_1 < \varepsilon$ then
                delete $u_i$
for $p_i \in P$ do
    sort $U_{p_i}$ ascendingly based on distance
    keep top $K$ elements in $U$
```

3.2 Evaluation

We describe our datasets, experiment setup and then report the evaluation results. Our evaluation first demonstrates that our approach is more efficient and requires less number of examples compared to the state-of-the-art approach [Gulwani, 2011] and other alternatives. Moreover, we also show that our design decisions are effective by comparing against different versions of the conditional statement learning algorithm.

3.2.1 Datasets

We identified 30 scenarios that require conditional transformations. The description of each scenario can be seen in the Appendix. Part of the scenarios were collected from online Excel user forums. We also manually collected conditional transformation scenarios to increase the number of test scenarios. The data was crawled from websites of museums by graduate students for final course projects, which required them to integrate data from multiple sources and build mash-up applications. They were required to perform a variety of transformations to convert the data into a suitable format. We

	avg size	min formats	max formats	avg formats
scenario	350	2	12	4.4

Table 3.1: Data profile

extracted the editing scenarios from these projects. As shown in Table 3.1, one scenario contains about 350 records on average. The average number of formats for a scenario is 4.4, the maximum is 12 and the minimum is 2. The scenarios range from two formats transformation such as the example shown in Figure 1.1 to the twelve formats transformation which requires learning the month to number conversions such as “Jan” to “1”, “Feb” to “2”, etc.

3.2.2 Experiment Setup

We performed the experiments on a laptop with 8G RAM and 2.66GHz CPU. To fully evaluate our approach, we compared 7 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 and directly applying Euclidean distance (DP v.s SP v.s NP), (2) utilizing the constraints collected from previous iterations compared to not utilizing the previous constraints (IC v.s non-IC) and (3) incorporating unlabeled data in learning a classifier compared to not incorporating the unlabeled data (ED v.s non-ED). After learning the conditional statements, these different alternatives all use our implementation of Gulwani’s approach to synthesize the branch programs.

- Naive Partitioning (NP): This algorithm directly uses Euclidean distance to select partitions to merge.
- Naive Partitioning with Incremental Constraint (NPIC): This algorithm utilizes all previous constraints besides using NP method

- Compatibility Score Based Partitioning (SP): **This is the state-of-the-art approach [Gulwani, 2011] that calculates the compatibility score as described in the previous work chapter to decide the partitions to merg**e.

- Compatibility Score Based Partitioning with Incremental Constraints (SPIC): This is the version of SP approach that uses previous constraints.

- 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 used to choose partitions to merge.

- Distance Metric Based Partitioning with Incremental Constraints (DPIC): this approach learns the weighted Euclidean distance with all the known constraints.

- Distance Metric Based Partitioning with Incremental Constraints and Expanded Training Data (DPICED): **It is the approach introduced in this paper. Besides DPIC, it also uses the learned distance metric to add unlabeled data into each partition to expand training data for learning the classifier**.

All 7 algorithms above use agglomerative clustering, which greedily selects the partitions to merge until there are no partitions to merge. However, as these approaches use different scoring functions, they give them different ways of utilizing the constraints. The DP, DPIC and DPICED can learn from the constraints to adjust its scoring function (distance function). NP and NPIC basically check whether the partitions to merge are contained in cannot-merge constraints. If they appear in cannot-merge constraints, they will skip this pair, select the partitions with the second best score and check with the constraints again. 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. Besides, SPIC's must-merge constraints are not cleared after each iteration, as SPIC only needs to extract the compatibility information from these constraints.

3.2.3 Metrics

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

- Total Time: the time (seconds) used to correctly transform all records in a scenario, which is the sum of the time used in each iteration.
- Time per Iteration: the time (seconds) used to generate the transformation program in one iteration.
- Number of Examples: the number of examples required to successfully transform a scenario where all entries are converted correctly. It reflects the number of iterations.
- 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.
- 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.

3.2.4 Experimental Results

As shown in Table 3.2, we can see that the distance metric learning based approaches (DPICED, DPIC, DP) have higher success rates over Euclidean distance based approaches (NPIC, NP) and compatibility score based approaches (SPIC, SP). The last four approaches (NPIC, NP, SPIC, SP) can take an extremely long time (more than 10 minutes) on certain scenarios. Especially, the compatibility score based approach (SPIC, SP) can only transform 77% of scenarios in less than 10 minutes.

	DPICED	DPIC	DP	SPIC	SP	NPIC	NP
Success Rate	1	1	0.97	0.77	0.77	0.93	0.87

Table 3.2: Success rates for different approaches on all scenarios

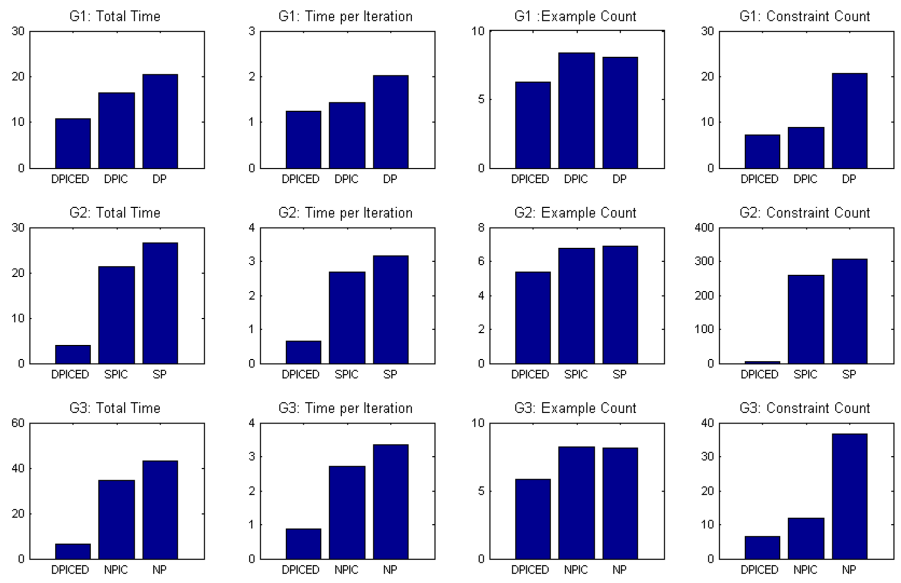


Figure 3.4: Comparing DPICED with 3 other groups of approaches: (G1) DPICED vs DPIC vs DP, (G2) DPICED vs SPIC vs SP and (G3) DPICED vs NPIC and NP

Excluding those failed scenarios, we also compared our approach (DPICED) against other approaches on the total time, time per iteration and number of required examples. To prevent the failed scenarios dominating the averaged results, we compared on the scenarios that all the approaches can successfully transform. As different approaches have few common successfully transformed scenarios, we performed the comparison within three groups where the approaches in each group can successfully transform all the scenarios in that group. The results are not comparable across groups. We first compared DPICED with DPIC, and DP and then compared DPICED with SPIC, and SP as well as compared DPICED with NPIC, and NP. The results are displayed in Figure 3.4.

The results in Figure 3.4 show that DPICED requires less time per iteration compared to other approaches and less number of examples which means fewer iterations. These two improvements combined contribute to the less total time compared to other alternative approaches. Excluding the failed scenarios, the DPICED still saved 6 and 10 seconds compared to DPIC and DP; saved 17 and 23 seconds compared to SPIC and SP; used 28 and 37 seconds less compared to NPIC and NP in terms of total time. We conducted paired one-tailed t test. The results suggest that the improvements were statistically significant ($p < 0.05$) except when comparing DPICED with DPIC on the time per iteration. As DPICED extend DPIC to incorporate more unlabeled data to improve the accuracy of the classifier, it only helps DPICED to reduce the number of required examples. But this extension does not affect the time per iteration in generating the transformation program.

The differences in the total times are because of the variance in the total number of discovered cannot-merge constraints. The higher the number is, the more failed merges the approach encounters, which in turn wastes more time. A better algorithm can learn from previous failed merges to avoid intersecting these partitions in the future. For example, by learning from the constraints to adjust the Euclidean distance metric, the DPICED approach had much less number of failed mergers compared to SPIC, SP, NPIC and NP. The reason that SP and SPIC have extremely high number of encountered constraints is because computing compatibility score requires fully verifying the compatibilities over a large number of partitions. On the contrary, other scoring functions don't require verifying the compatibility. Furthermore, the approaches utilizing the previous constraints will avoid redundant work; they used less time compared to their counterparts which didn't record previous constraints. For example, DPIC, SPIC and NPIC all used less total time and less time per iteration compared to DP, SP and NP

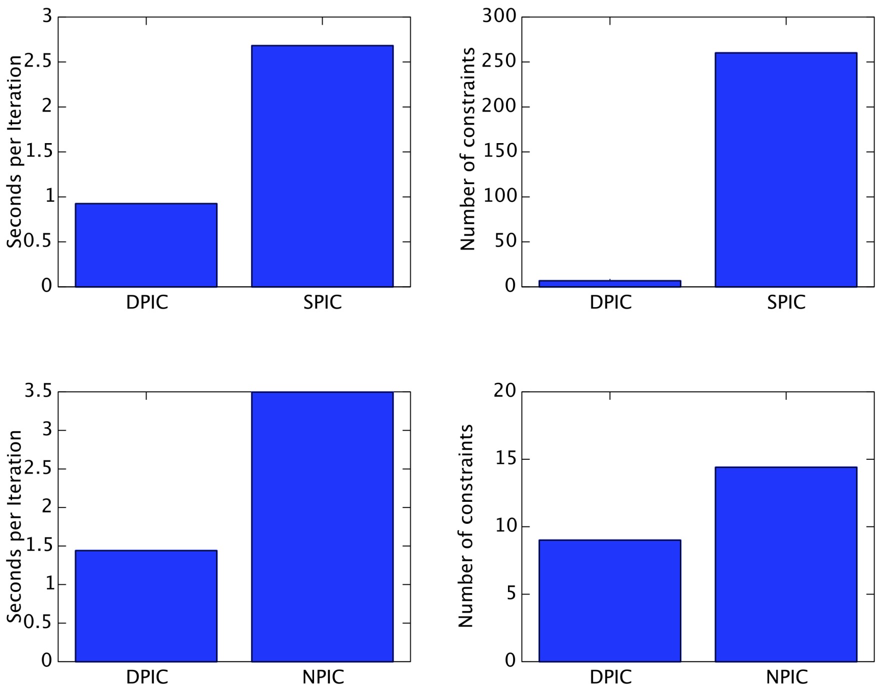


Figure 3.5: DPIC used fewer seconds per iteration compared to SPIC and NPIC

respectively. To explain the experimental results and validate our algorithm design, we will discuss the comparisons between different versions of the algorithm.

Utilizing the information from previous iterations can improve the system efficiency. The efficiency of the system is measured by the average time per iteration that reflects the time used by the system to generate a program. The approaches that utilized the previous constraints avoided redundant work; they used less time compared to their counterparts which didn't record previous constraints. In Figure 3.4, DPIC, SPIC and NPIC all used less time per iteration compared to DP, SP and NP respectively.

Learning the weighted Euclidean distance improves the system efficiency compared to the compatibility score and Euclidean distance. We compared DPIC with SPIC and NPIC. The results are shown in Figure 3.5, where DPIC used less time per iteration compared to SPIC and NPIC. The improvement in the time per iteration comes from the variance in the number of discovered cannot-merge constraints. The higher the

	DPICED	DPIC
Accuracy	0.952	0.928
Example Number	6.4	8.6

Table 3.3: Classifier accuracy

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 and NPIC. SPIC had extremely high number of cannot-merge constraints, as computing compatibility score requires fully verifying the compatibilities over a large number of possible partitions. Furthermore, as more examples were given, the time differences become more evident. For example, Figure 3.6 shows that the time per iteration for one scenario increases as the number of examples increases and the time saving also becomes larger. For every merge, Euclidean distance based methods calculated the distances between all partitions and selected the pair with the smallest distance. It is usually efficient to compute the Euclidean distance or weighted Euclidean distance. On the contrary, the compatibility score based approaches require calling the *comp* function $O(n^3)$ times to select two partitions to merge and checking the compatibility of partitions is expensive. Figure 3.6 shows that SPIC increased rapidly as more examples were given. It also shows that the learned weighted Euclidean distance function (DPIC) outperforms directly applying Euclidean distance function (NPIC). As the number of examples grows, more constraints would be discovered. the NP based algorithms will have more failed merges, as it cannot adjust its distance metric to avoid merging likely incompatible 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

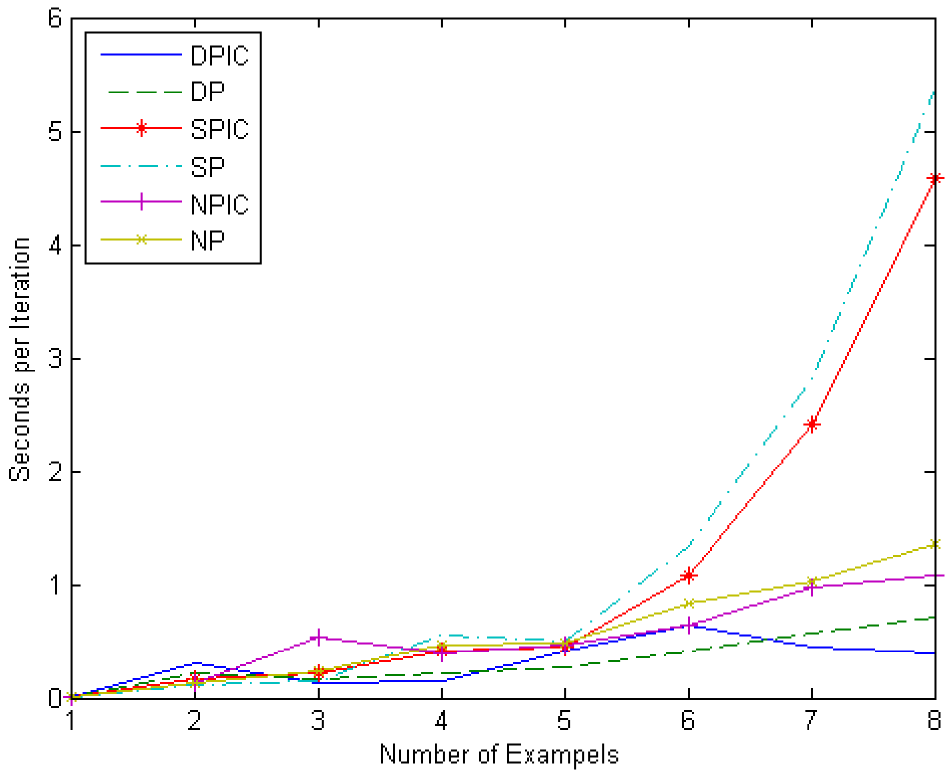


Figure 3.6: Time per iteration increases as example number grows

improved accuracy of the learned classifier. The poor classifier classifies 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. 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 on the unlabeled data. As a result, it reduced the average number of required examples from 8.6 to 6.4 as shown in Table 3.3.

Using DPIC and DPICED to partition the examples does not increase the number of conditional branches in the final program. Gulwani, 2011 mentioned that using compatibility score to merge partitions practically leads to a smaller number of

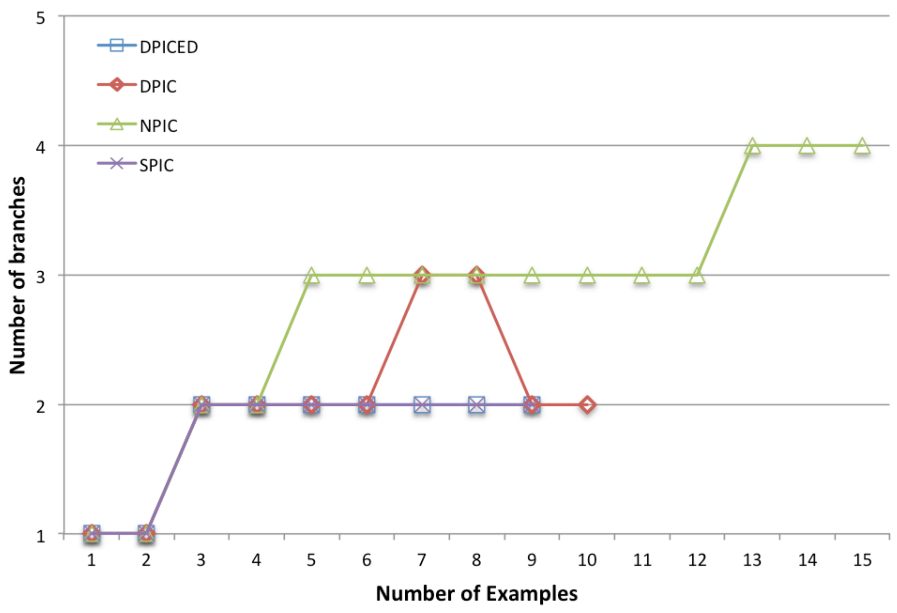


Figure 3.7: The number of conditional branches in the final program generated by different approaches

ID	Original	Target
P1	Birds of California untitled (Seascape)	Birds of California Seascape
P2	#15 Archtypes	#15 Archtypes
P3	untitled (forest landscape)	forest landscape

Table 3.4: The partitions of example of NPIC

partitions, which is more likely to be the correct transformation program. As agglomerative clustering with constraints can get stuck in local optima [Davidson and Ravi, 2009], 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 restarts 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.

We compared the 7 approaches on all the 23 scenarios that SP can successfully transform. We noticed that DPICED, DPIC SPIC and SP all can achieve the same smallest of partitions when the system finished transforming the scenarios. However, DP used 3 partitions on one scenario instead of 2 partitions used by SP. NP and NPIC even failed on one scenario shown in Table 3.4, where they were trapped into the wrong partitions. The scenario in Table 3.4 aims to extract the names of artwork. If there were parenthesizes, we wanted to extract the content inside them as the artwork's name. Otherwise, we kept the name the same as before. NPIC incorrectly placed two entries "Birds of California" and "untitled (Seascape)" in the same partition, as the two examples were compatible with each other and they had the closest Euclidean distance. It was trapped into this wrong partition that caused a higher number of partitions in the end, as the distance metric here is only determined by the inputs regardless of the constraints. Figure 3.7 shows the number of branches (number of partitions) related to the number of examples. This figure only shows DPICED, DPIC, NPIC, SPIC, as DP, SP and NP were exactly the same as DPIC, SPIC and NPIC respectively. We can see that SPIC and DPICED (overlapped) always had the smallest number of partitions and successfully finished transformation using 9 examples. The NPIC's partition number kept increasing as new examples were given. DPIC at certain step had more partitions, but as more examples were given, it finally converged to the correct number of partitions by adapting to the constraints.

Chapter 4

Adapting Branch Programs

The development of programming-by-example approaches (PBE) allows users to perform data transformation without coding [Lieberman, 2001]. The user can directly enter the data in the target format and the system learns from these input-output examples to generate a transformation program that is consistent with the examples.

Examples often have many interpretations. Users generally provide multiple examples in an iterative way to clarify their intentions. For example, in Table 4.1, a user wants to extract the year, manufacturer, model, location, and price from car sale posts. The left column shows the titles of the car sale posts and the right shows the target values. The user can directly enter the target data (2000 Ford Expedition los angeles $4900) for the first entry as an example. The system synthesizes a program and applies the program to the rest of the data. The user checks the transformed results and provides examples for any incorrect results to refine the program until she determines the results are correct.

PBE systems typically generate an entirely new program as users provide new examples. Their time complexity is exponential in the number and a high polynomial in the length of examples [Raza et al., 2014]. This prevents these PBE systems from being applied to real-world scenarios that require many or long examples to clarify a user's intention.

We observe that the programs generated from previous examples are usually close to the correct ones. A large portion of the programs remain the same as a user provides additional examples to refine the program. Reusing the correct subprograms can greatly reduce the computational costs.

Input Data	**Target Data**
2000 Ford Expedition 11k runs great los angeles $4900 (los angeles)	2000 Ford Expedition los angeles $4900
1998 Honda Civic 12k miles s. Auto. - $3800 (Arcadia)	1998 Honda Civic Arcadia $3800
2008 Mitsubishi Galant ES $7500 (Sylmar CA) pic	2008 Mitsubishi Galant Sylmar CA $7500
...	...

Table 4.1: Data transformation scenario

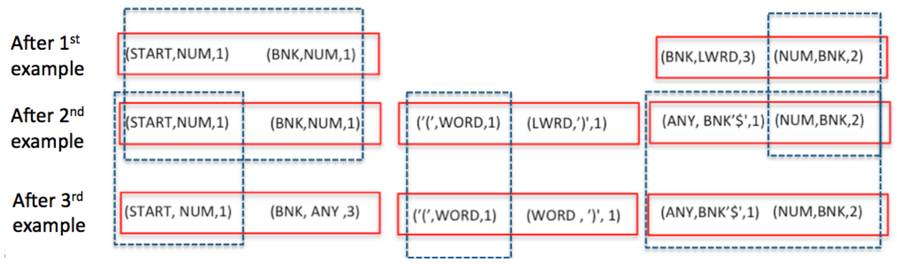
Figure 4.1a shows the program learned using the first record as an example. The program is basically a concatenation of several segment programs. The transformation program is refined as the user provides new examples as shown in Figure 4.1b. P1 is the synthesized program from the first row of Table 4.1, P2 is the program when the first two rows are used as examples and the P3 is the returned program from the first three rows. The solid rectangles represent segment programs and each segment program has two position expressions in parentheses. The dashed rectangles show the position programs that stay the same as the program changes. We can see that a large portion of the new program does not change. From P1 to P2, 3 out 4 position programs stay the same. From P2 to P3, 4 out of 6 are the same as before.

Based on the observation above, we present an approach to adapt programs with additional examples. To adapt the program, first, the approach needs to identify the incorrect subprograms. Since a transformation program often has multiple subprograms, it is essential to correctly identify the incorrect subprograms to avoid missing them or redundantly generating the correct subprograms. Second, the approach needs to be able to generate correct subprograms to replace the incorrect ones.

To address the above challenges, we have two insights. First, we noticed that PBE approaches typically generate traces. Traces are the computational steps executed by a program to yield an output from a particular input [Kitzelmann and Schmid, 2006]. A *trace* defines how the output string is constructed from a specific set of substrings from the input string. The PBE approaches then generalize over these traces to produce

```
Transform(val)
    $pos_1$ = val.indexOf('START','NUM',1)
    $pos_2$ = val.indexOf('BNK','UWRD',1)
    $pos_3$ = val.indexOf('ANY','UWRD',2)
    $pos_4$ = val.indexOf('BNK','NUM',1)
    $pos_5$ = val.indexOf('BNK','LWRD',2)
    $pos_6$ = val.indexOf('NUM','BNK',-1)
    output = val.substr($pos_1$,$pos_2$)+val.substr($pos_3$,$pos_4$)+val.substr($pos_5$,$pos_6$)
    return output
```

(a) Program learned from the first example



(b) Programs refined by adding the second and third rows as examples

Figure 4.1: Program changes as more examples are added

the programs that are consistent with all examples [Summers, 1977; Kitzelmann and Schmid, 2006; Gulwani, 2011; Lau et al., 2003; Harris and Gulwani, 2011; Singh and Gulwani, 2012b,a]. As these traces encode the required input and output pairs for each subprogram, they can be leveraged to detect the buggy subprograms. Second, the correctness of a program generated by PBE is only determined by whether it can output the correct results specified by the traces. Thus, if a program can return the expected results on the examples, the program is considered to be correct even though the program may fail on future unseen examples.

Our approach can deterministically identify incorrect subprograms and adapt them to additional examples. When the user provides a new example, our approach applies the previously learned program on this new example. It records the outputs of all the subprograms and compares them against the expected outputs shown in the trace of the example. As the number of incorrect subprograms can be different from the number of outputs in the trace, our approach precisely maps subprograms to their corresponding expected outputs to identify the incorrect subprograms, whose execution results differ from the expected ones. As the transformation program has subprograms, our approach searches for the incorrect subprograms until no more incorrect subprograms are available. The approach then generates new correct subprograms to replace incorrect subprograms. The new subprograms are consistent with both previous and new examples.

To sum up, our approach makes the following contributions:

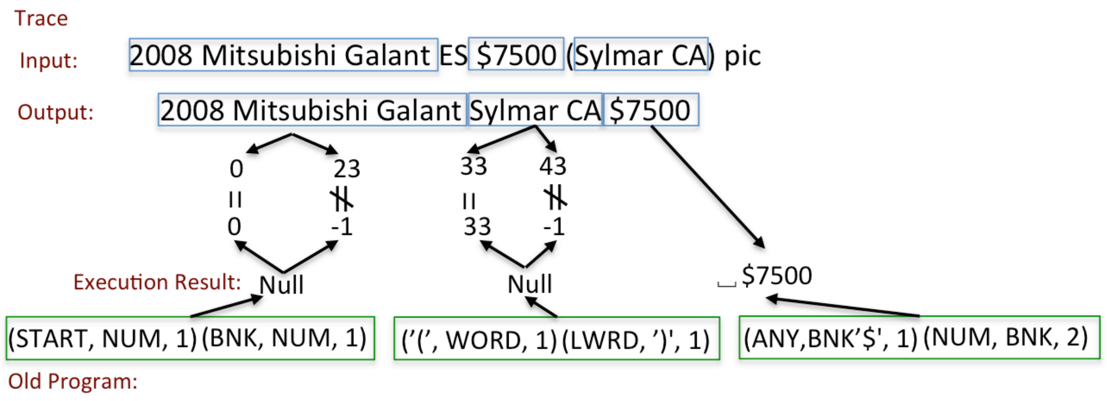
- iteratively generating branch programs from examples,
- deterministically identifying incorrect subprograms and refining them,
- enabling the PBE approach to scale to much more complicated examples.

4.1 Iteratively Learning Programs by Example

Our approach adapts programs with new examples by identifying incorrect subprograms and replacing them with refined programs. We first introduce some notations. Let $P = [p_1, p_2, ..., p_n]$ represent the transformation program. Every $p_i = const|(p_i^s, p_i^e)$

corresponds to a segment program. It can be either a constant string or extracting substring from the input. Currently, our approach does not adapt programs with loop statements. p_i^s and p_i^e are the start and end position programs. A position program identifies a position with certain context in the input. $P_{[k,l]}$ refers to a subsequence of programs between index k and l ($1 \leqslant k \leqslant l \leqslant n$). Let $T^{'} = [t_{p_1}, t_{p_2}, ..., t_{p_n}]$ represent the output of the program P on the new example. The t_{p_i} is the execution result of the corresponding subprograms p_i. The $t_{p_i^s}$ and $t_{p_i^e}$ are the corresponding execution results of p_i^s and p_i^e. Let $T = [t_1, t_2, ..., t_m]$ represent the trace created from the new example. t_i is a segment trace, which defines how a substring in the output is produced from the input. If the substring is copied from input value, t_i^s and t_i^e refer to the start and end positions of the segment in the input. Let $H = \{H_1 = [h_{11}], H_2 = [h_{21}, h_{22}]...\}$ represent the hypothesis space used to generate programs of different numbers of segments. H_i represents the space that has i segments. For example, H_2 defines the set of possible programs that create the output string using two segments. A *segment hypothesis space* contains all possible programs for generating a segment. We use h_{ij} to represent all the jth segment programs in the programs with i segments. $H_{i[r,t]}$ represents the subsequence of segment spaces between index r and t of H_i. Similarly, h_{ij}^s and h_{ij}^e correspond to the start and end position spaces of h_{ij} containing start and end position programs.

To adapt the transformation program on the new example, our approach creates traces from the new example. It then iterates over these traces and utilizes these traces to generate a list of patches. Each **patch** contains 3 elements: (1) a subsequence of incorrect programs, (2) their expected traces and (3) their corresponding hypothesis spaces. The approach then uses these traces in the patch to update the corresponding hypothesis spaces and generates correct subprograms to replace the incorrect subprograms. The approach stops when it either successfully generates a transformation program that is consistent with all examples or it exhausts all the traces.



Trace
Input:
2008 Mitsubishi Galant ES $7500 (Sylmar CA) pic
Output:
2008 Mitsubishi Galant Sylmar CA $7500
0
23
0
-1
33
43
33
-1
Execution Result:
Null
Null
␣$7500
(START, NUM, 1) (BNK, NUM, 1)
('(', WORD, 1) (LWRD, ')', 1)
(ANY,BNK'$', 1) (NUM, BNK, 2)
Old Program:

Figure 4.2: P and T have the same number of segments

To refine a program using a trace with m segments, the approach selects the hypothesis space H_m from H, which contains all the candidate programs with m segments. Depending on the number of segments (n) in the current program (P), it handles two cases separately: (1) $n = m$ and (2) $n \neq m$.

4.1.1 P and T have the same number of segments

The number of the execution results of the segment programs is the same as the number of segment traces. The approach directly compares the t_{p_i} with the t_i (line 3). If any execution result differs from the trace, there is an incorrect segment program. The algorithm adds this program (p_i) with the corresponding trace (t_i) and hypothesis space (h_{mi}) for further refinement.

For example, Figure 4.2 shows a program learned using the first two records in Table 4.1. The new example is the third record. In the figure, we represent each segment program using a rectangle with its start and end position programs in the parentheses. The "Null" in the execution results represents that the segment program cannot generate an output. The "-1" means the program cannot find a position matching the pattern specified in the position program. The first segment program in Figure 4.2 cannot generate

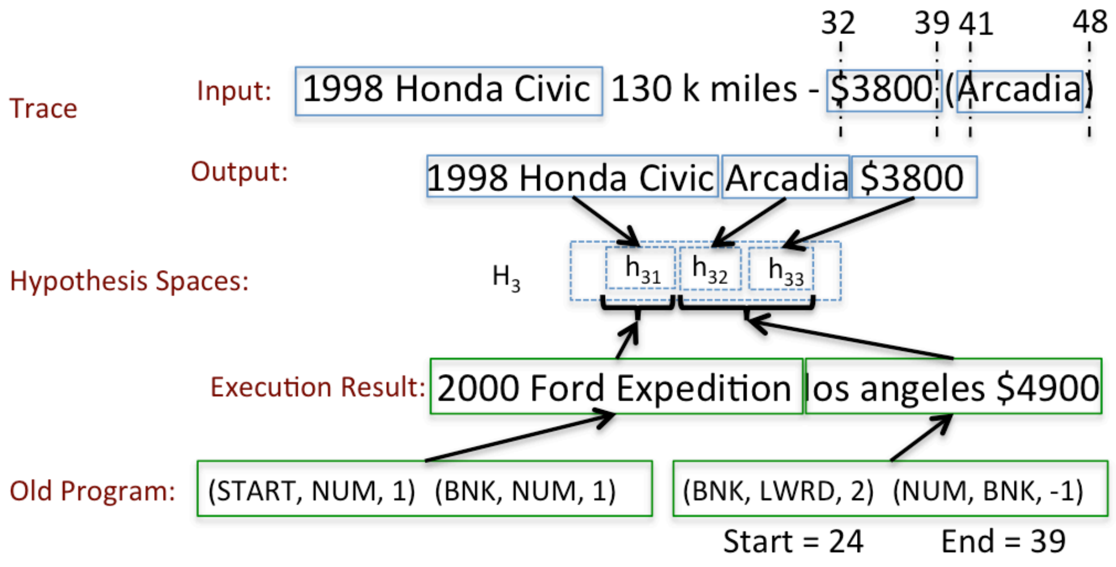


Figure 4.3: P and T have different number of segments

the correct substring (2008 Mitsubishi Galant) as specified in the trace. The algorithm adds the program, its trace and corresponding segment hypothesis space into the patches for refinement (line 4). As the third segment program's output is correct, it doesn't need to be refined.

4.1.2 P and T have a different number of segments

Since the number of segment programs is different from the number of segment traces, the algorithm cannot directly map the segment programs to the segment traces with the same indices. However, it can find the mapping between segment programs and segment hypothesis spaces. Meanwhile, the segment hypothesis spaces are mapped to segment traces with the same indices. The algorithm can then align the segment traces and segment programs since they are mapped to the same sequence of segment hypothesis spaces.

The algorithm maps subsequences of segment programs to subsequences of segment hypothesis spaces (line 5). To identify the mapping, it first obtains the output ($O_{P_{[k,l]}}$) by evaluating the subsequence of programs ($P_{[j,k]}$) on old examples (O). It then identifies the hypothesis space ($H_{m[i,j]}$). The part of old examples with the output $O_{H_{m[i,j]}}$ used to derive $H_{m[i,j]}$ should contain the same string as $O_{P[k,l]}$. Thus, the space ($H_{m[i,j]}$) contains sequences of subprograms that can generate the same output as $P_{[k,l]}$ but these sequences of subprograms can represent different segmentations.

Figure 4.3 shows the outputs of two segment programs. The p_2 generates "los angeles \$4900". H_3 contains a different segmentation where the last two segment hypothesis spaces (h_{32} and h_{33}) are derived from the traces with output "los angeles " and "\$4900" separately. Thus, the second segment program is mapped to the second and third segment hypothesis spaces in H_3. The algorithm can then identify the aligned segment traces using the same indices as the segment hypothesis spaces. It then compares whether the subprograms can generate the results as specified in the traces or whether the lengths of the two sequences match to decide whether it should add this sequence of programs for further adaptation (line 7).

The algorithm can further map the position programs with the traces. When a sequence of segment programs are mapped to a sequence of segment traces, the approach compares the output of the first start and the last end position programs with the first start and the last end position traces to exclude the correct subprograms that can generate the expected results (line 8 and 9). For example, the last segment program in Figure 4.3 is mapped to two segment traces. The end position of the third segment trace is 39, which is the same as the output of p_2^e on the new example (39). Thus, the current end position program is correct and can be reused. In the case that P and T have the same number of segments, there is only one segment in $T_{[i,j]}$ and $P_{[k,l]}$. The start position program of the first segment ($t_{p_1^s}$) outputs "0", which is the same as the start position in the trace (t_1^s) in Figure 4.2. Therefore, the algorithm excludes this position program from adaptation.

4.1.3 Adapting Incorrect Programs

As the algorithm has identified the patches consisting of incorrect subprograms, the expected traces and the corresponding hypothesis spaces, it first uses the traces to update these spaces. The algorithm first creates a basic hypothesis space using the traces and merges this space with the identified hypothesis spaces to generate the updated spaces using the same method described in the previous work section. It then generates the correct subprograms from the updated spaces that is consistent with expected traces. Finally, it replaces the incorrect subprograms with correct subprograms and returns the new program (line 10).

Algorithm 4: Program Adaptation

```
   Input: $P$ program, $H$ hypothesis space, $T$ trace of the new example, $O$ old
          examples
   Output: $P_{new}$
   n = size($P$), m = size($T$), patches = []
1  $H_m$ = findHypothesisSpaceByLength($H$, m)
2  if $n = m$ then
       for $i = [1, m]$ do
3          if $t_i \neq t_{p_i}$ then
4              patches.add (( $[p_i]$, $[t_i]$, $[h_{mi}]$))
   else
5      seqmap= $\{([k, l] : [i, j]) \mid O_{P_{[k,l]}} \in O_{H_{m[i,j]}}\}$
6      for $\{ [k, l] : [i, j] \}$ in seqmap do
7          if $(j - i) \neq (l - k) \vee T_{[i,j]} \neq T_{P_{[k,l]}}$ then
               patches.add(($P_{[k,l]}$, $T_{[i,j]}$, $H_{m[i,j]}$))
   for $(P_{[k,l]}, T_{[i,j]}, H_{m[i,j]})$ in patches do
8      if $t^s_i = t^s_{p^s_k}$ then modify patch to remove $p^s_k$
9      if $t^e_j = t_{p^e_l}$ then modify patch to remove $p^e_l$
10 $P_{new}$ = apply(patches, $P$)
   return $P_{new}$
```

4.1.4 Soundness and Completeness

Our approach can always adapt the transformation program using the new example to generate a correct program, if there exists a correct transformation program.

Proof 1 *The approach is sound as it only returns the program that is consistent with examples. To prove the completeness, suppose $\exists P^*$ consistent with $O \cup N$. O refers to the previous examples and N is the new example. This implies $\exists trace^*$ $trace^*$ is the trace of the correct program (P^*) on the new example (N).* ***First, the algorithm can identify all incorrect subprograms*** *w as it only excludes the correct subprograms that generate expected outputs specified by $trace^*$.* ***Second, the identified space H_i contains the correct subprograms.*** *As the w is consistent with O, to replace w, the correct subprograms (r) should also generate the same output as w on previous examples O. As the recovered space H_i contains all the alternative programs that can generate the same outputs as w, the space contains r. Lastly, as the approach uses a brute force search in the space to identify the correct subprograms r as described in Gulwani (2011), it can identify the correct subprograms r. Therefore, the algorithm can generate P^* by replacing w with r.*

4.1.5 Performance Optimizations

There can be multiple traces for one input-output pair. To more efficiently adapt the programs, our approach filters traces and then sorts the remaining traces to reuse most of the previous subprograms.

The trace (T) should always have at least the number of segments as the number of segments in the program (P). Because the approach generates simpler programs with fewer segments first from all the programs that are consistent with examples, all the programs with fewer segments have been tested and failed to transform the examples correctly. Therefore, the approach only uses the traces with a larger or equal number of segments to refine the program.

We aim to make the fewest changes to the program to make it consistent with the new example. The approach sorts the traces in descending order based on their resemblance to the $T^{'}$. The approach iterates over the traces in the sorted list to adapt the program. To sort the traces, the approach creates a set s_1 that contains the outputs of the position expressions. It then creates a set s_2 that contains all the positions in the trace. The approach then sorts the traces based on the score $(size(s_1 \cap s_2) + 1)/(size(s_2) + 1)$. The high score indicates a large overlap between s_1 and s_2. It in turn means a close resemblance between the program and the trace.

4.2 Evaluation

We implemented our iterative programming-by-example approach (IPBE). It uses the iterative approach described in chapter 3 to learn the conditional statements and used the approaches described in this chapter to synthesize the branch programs. We conducted an evaluation on both real-world and synthetic datasets.

4.2.1 Dataset

Our real-world data consists of two parts. First, we collected the 17 scenarios from Lin et al., 2014 (referred to as D1). Each scenario contains 5 records. We also used the 30 scenarios described in Appendix (referred to as D2).

Second, we created a synthetic dataset by combining multiple scenarios. The synthetic scenarios are to transform records with multiple fields at the same time. We show three example input and output records in Table 4.2. They have 7 columns in the output records. To transform one record from the input to the output, the approach learns a sequence of transformations, such as extracting the first name, extracting the last name, etc., and combines them in one transformation program. By changing the number of columns (1- 10) in the output records, we can control the complexity of the scenario. Each scenario has about 100 records.

	Name		Year		Dimension			...
Input	Cook Peter		1905 - 1998. (T.V)		22 x 16 1/8 x 5 1/4 inches			...
	Clancy Tom		1858 - 1937		5/8 x 40 x 21 3/8 inches			...
	Hicks Dan		1743 - 1812		6 15/16 x 5 1/16 x 8 inches			...
	First	Last	Birth	Death	1st	2nd	3rd	...
Output	Cook	Peter	1905	1998	22	16 1/8	5 1/4	...
	Clancy	Tom	1858	1937	5/8	40	21 3/8	...
	Hicks	Dan	1743	1812	6 15/16	5 1/16	8	...

Table 4.2: Synthetic scenario for generating the first 7 columns

4.2.2 Experiment Setup

We performed the experiments on a laptop with 8G RAM and 2.66GHz CPU. We compared IPBE with two other approaches: (1) Gulwani's approach [Gulwani, 2011] and (2) $Metagol_{DF}$ [Lin et al., 2014]. We used our own implementation of Gulwani's approach rather than Flashfill in Excel 2013, as a large portion of scenarios in the test data cannot be transformed by Flashfill. Since the goal of the experiment is to study the performance of different approaches on synthesizing branch programs, we use the same conditional learning approach described in chapter 3 for both IPBE and Gulwani's approach. For $Metagol_{DF}$, we obtained the code from the authors and ran the code on our machine to obtain the results on D1. We compared the three methods in terms of the time (in seconds) to generate a program that is consistent with the examples.

		Min	Max	Avg	Median
D1	IPBE	0	5	0.34	0
	Gulwani's approach	0	8	0.59	0
	Metagol	0	213.93	55.1	0.14
D2	IPBE	0	1.28	0.20	0
	Gulwani's approach	0	17.95	4.02	0.33
	Metagol	$\sim$	$\sim$	$\sim$	$\sim$

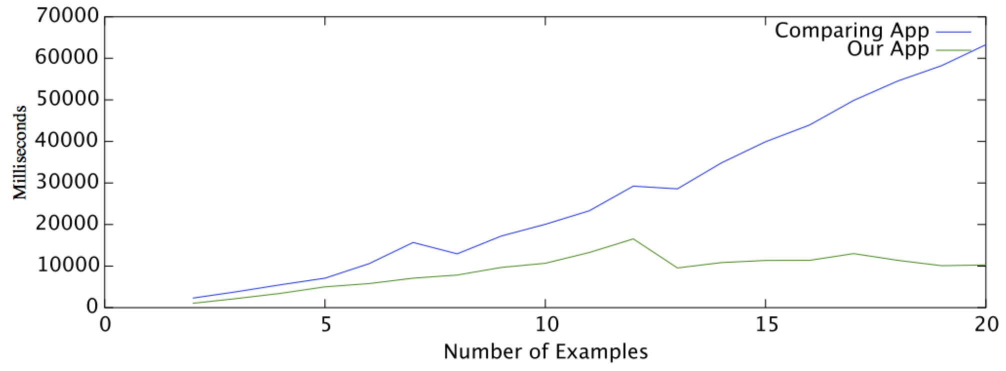
Table 4.3: Results of read-world scenarios

4.2.3 Real-World Scenario Results

The results on real world scenarios are shown in Table 4.3. We calculated the average program generation time (in seconds) for each scenario. To calculate the time for IPBE and Gulwani's approach, we recorded the program generation time for all the iterations until all the records were transformed correctly. We then averaged the program generation time across all iterations and refer to this average time as the generation time. For $Metagol_{DF}$, we used the experiment setting in Lin et al., 2014 and averaged the program generation time for the same scenario.

The Min is the shortest time among the set of generation time for all scenarios. The Max, Avg and Median are also calculated for the same set generation time. A 0 in the results means the time is smaller than one millisecond. As we cannot easily change $Metagol_{DF}$ to run it on D2, we only ran our approach and Gulwani's approach on D2.

We can see that IPBE outperforms the other two approaches as shown in Table 4.3. Comparing IPBE with Gulwani's approach, we can see that reusing previous subprograms can improve the system efficiency. $Metagol_{DF}$ treats each character as a token, which enables it to do transformation on the character level. However, it also significantly increases the search space, which causes the system to spend more time to induce the programs.



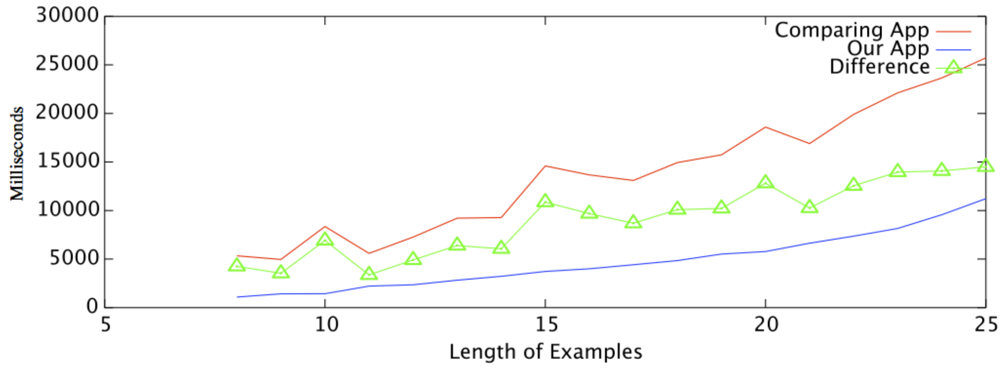


Figure 4.4: Synthesizing time rises as column number increases

4.2.4 Synthetic Scenarios Results

To study how Gulwani's approach and IPBE scale on complex scenarios, we created 10 synthetic scenarios with the number of columns in the output ranging from 1 to 10. We gradually increased the number of columns so that the approaches had to learn more complicated programs. We ran the two approaches and provided examples until they learned the programs that can transform all records correctly and then measured the average time used to generate a program.

The time in Figure 4.4 used by the two approaches increases as the number of columns increased. However, IPBE scales much better compared to Gulwani's approach when more columns are added. The time saving comes from the fact that IPBE can identify the correct subprograms and only refine the incorrect subprograms. The cost of generating a new program is related to the portion of the program that requires updating rather than the actual size of the program. A program with more subprograms can usually reuse more subprograms so that the time saving will be more evident.

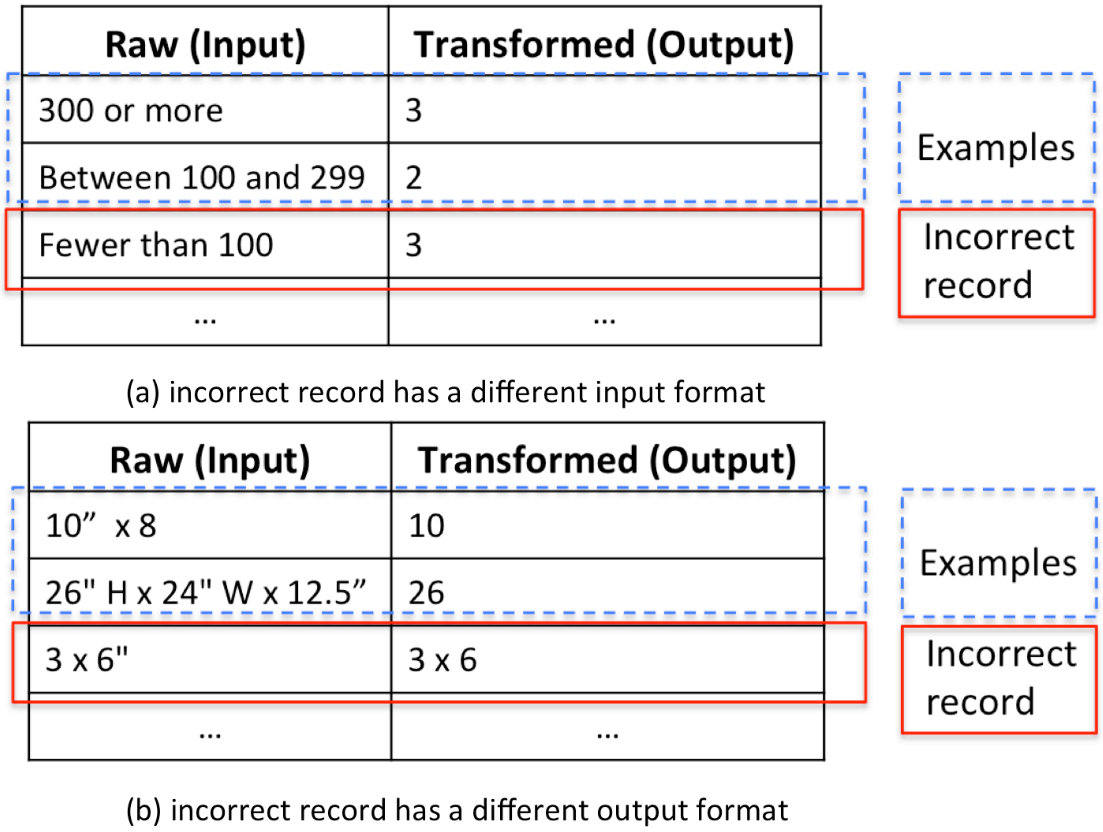
Chapter 5

Maximizing Correctness with Minimal User Effort

Programming-by-example (PBE) approaches [Lieberman, 2001] enable users to generate programs without coding. Recently, these approaches have been successfully applied to data transformation problems [Gulwani, 2011] to save users from writing many task-dependent transformation programs. For example, Figure 5.1(b) shows the dimensions for a set of artworks. To extract the first degree (height) from the dimension, the user enters “10” as the target output for the first entry. The PBE approach generates a program that is consistent with this input-output pair (also referred to as example). It applies the program to the rest of records to transform them. If the user finds any incorrect output, she can provide a new example, the approach refines the program to make it consistent with all given examples. The user often interacts with the system for several iterations and stops when she determines that all records are transformed correctly.

Despite the success of generating programs using PBE approaches, the correctness of the results is still an issue. Real-world data transformation often involves thousands of records with various formats. Each record consists of *raw data (input)* and *transformed data (output)*. The users are often not aware of all the formats that they should transform. They know whether the records are transformed correctly when they see them. But, they lack the insight of the unseen formats of the records buried in the middle of datasets.

To help users verify whether the records are transformed correctly, existing PBE approaches [Gulwani, 2011; Miller and Myers, 2001; Wolfman et al., 2001] provide recommendations or highlight certain records for the users. The user checks these records and provides new examples for those incorrect records. Here, we consider a record as transformed correctly (referred to as *a correct record*), when it is transformed into the expected format. Otherwise, the record is considered incorrect (referred to as *an incorrect record*).



Raw (Input)	Transformed (Output)
300 or more	3
Between 100 and 299	2
Fewer than 100	3
...	...

(a) incorrect record has a different input format

Raw (Input)	Transformed (Output)
10” x 8	10
26" H x 24" W x 12.5”	26
3 x 6"	3 x 6
...	...

(b) incorrect record has a different output format

Examples
Incorrect record
Examples
Incorrect record

Figure 5.1: Different rules for recognizing incorrect records

To generate such recommendations, there are several challenges. First, the dataset is usually huge and the users need to see the results on the fly so that they can provide additional examples if necessary. Fully transforming the entire dataset and analysing all the transformed records to generate recommendations takes too long to be practical.

Second, the users' intentions are highly task-dependent and there is not a universal rule for determining whether the results are correct. Meanwhile, the approach should be able to hypothesize users' intentions accurately to provide useful recommendations. For example, the scenario in Figure 5.1(a) is to encode different texts into numbers. The users want to transform the three records into "3", "2" and "1" respectively. After giving the two examples in the dashed rectangle, the learned program does not transform the 'Fewer than 100' into the expected value '1', as it has not seen an example of that type before. The problem is 'Fewer than 100' has different words from the inputs of the two examples when they are represented using a bag-of-words model [Feldman and Sanger, 2006]. To capture this incorrect record, we can use a rule to identify the records that have different words in the *input* from the examples. However, in Figure 5.1(b), the counts of numbers, blank spaces, quotes and "x" in the inputs are the same for the third and the first record. We need to use a different rule that identifies the records with different *output* formats to locate the incorrect records since the output of the third record contains an "x" and blank spaces while the first two records do not have.

Third, the recommendation should place the incorrect records at the beginning of the recommended list so that users can easily notice these records. Otherwise, users have to examine many correct records before identifying the incorrect one, which would be a burden.

Fourth, users are often too confident with their results to examine the recommended records, which they regard as an extra burden [Panko, 1998; Ko et al., 2011]. Even when there are incorrect records in the recommendation, the users may ignore them and stop the transformation.

To address the challenges above, our approach samples records to allow users to focus on a small portion of the entire dataset. It statistically guarantees that a user-specified percentage of the records of the entire dataset is transformed correctly with certain confidence when all records in the small sample are transformed correctly. Our approach also maintains a library of different classifiers representing different rules for checking whether a record is transformed correctly. It uses an ensemble method to combine these different classifiers to automatically identify incorrect records in various scenarios. To save users' time in checking the recommendations, we provide users two ways to label the recommended records: (1) users can confirm a recommended record is correct or (2) they can provide the expected outputs for the incorrect records. Our approach then learns from these labels and provides users with refined recommendations. Finally, besides providing the recommendation that exposes incorrect records to users, we have also developed a method that identifies a minimal set of records that the user should examine before finishing the transformation.

Our contributions are summarized below:

- minimizing the user effort in obtaining a user-specified correctness
- allowing users to focus on a small sample of a large dataset
- combining multiple classifiers based on different perspectives for verifying the correctness of records
- refining recommendations when users confirm certain recommendations are correct or incorrect
- controlling the user overconfidence by requiring users to examine certain records before finishing the transformation.

5.1 Verifying the transformed data

To verify the correctness of the transformed data, our approach samples records from the entire dataset. The approach then automatically identifies the potentially incorrect records and recommends these identified records for the users to examine. When the recommendations are shown to users, they can provide the expected values for the incorrect records or they can confirm that a record is transformed correctly. Our approach uses the records that the users have edited or confirmed as new examples to refine the recommendations.

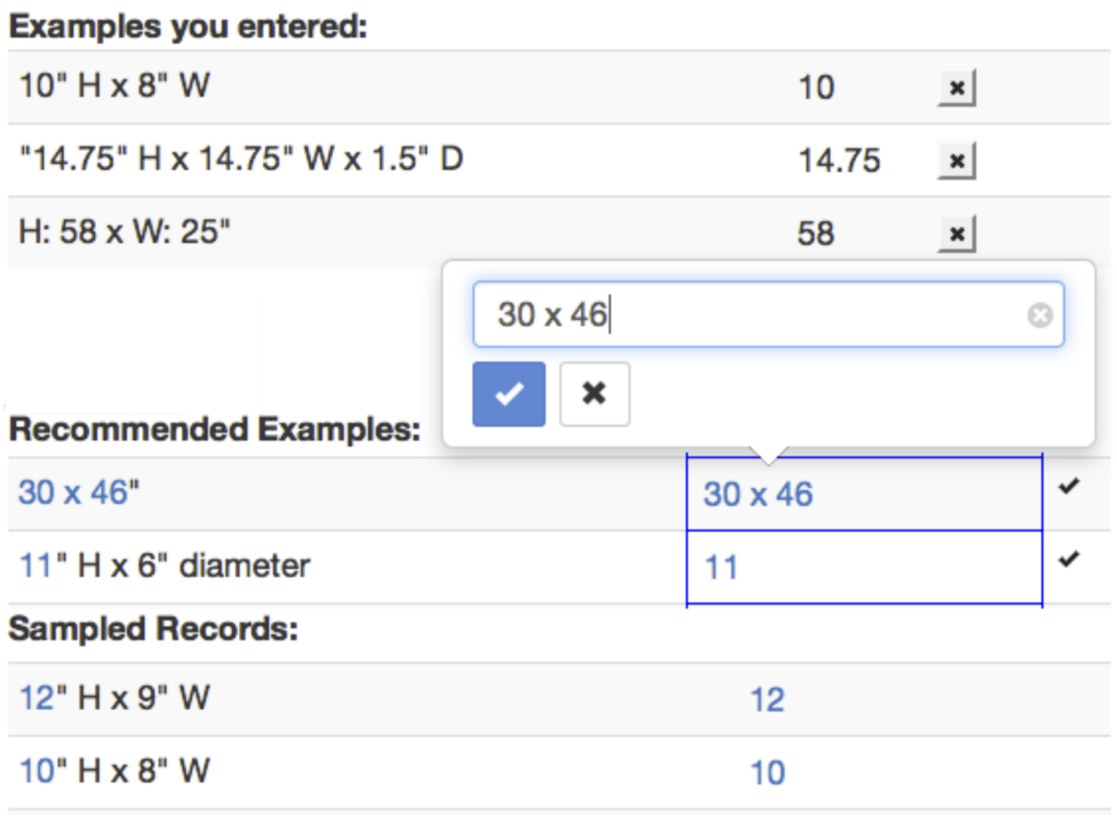
The user interface of our system is shown in Figure 5.2. The interface consists of three areas:

- examples you entered: this area shows all the examples provided by the user. There are also buttons with cross icons used for deleting previous examples
- recommended examples: this area shows all the potential incorrect records for users to examine. If the user finds an incorrect record, she can click the record and enter the target output in the pop-up window as in Figure 5.2. She can also click the button with a check icon to confirm that the record is correct
- sampled records: this area shows all the records in the sample.

5.2 Sampling records

Our approach uses hypothesis testing to decide whether the number of incorrect records in the entire dataset is below a certain percentage. The hypothesis is *the percentage of incorrect records is smaller than* p_{lower}. The alternative hypothesis is *the percentage of incorrect records is not less than* p_{upper}. The p_{lower} represents that the percentage of incorrect records that we want to achieve in the dataset. The p_{upper} refers to a percentage of incorrect records that we want to avoid ($p_{upper} \geqslant p_{lower}$).

We use the binomial distribution $B(n, p)$ to model the distribution of incorrect records, as each record is transformed either correctly or incorrectly. The parameter



Examples you entered:

10" H x 8" W	10	✖
"14.75" H x 14.75" W x 1.5" D	14.75	✖
H: 58 x W: 25"	58	✖

30 x 46

Recommended Examples:

30 x 46"	30 x 46	✓
11" H x 6" diameter	11	✓

Sampled Records:

12" H x 9" W	12
10" H x 8" W	10

Figure 5.2: User interface

n is the number of sampled records. The parameter p is the probability that a record is incorrect, which can also be interpreted to mean that a p fraction of records are transformed incorrectly. To find the sample size for testing the hypothesis, we use the binomial cumulative distribution function as shown in Formula 5.1 [Desu and Raghavarao, 1990]. $Pr(x < Z_a; n, p)$ represents the probability that the number of incorrect records (x) is less than Z_a in the binomial distribution $B(n, p)$. We use $1-\alpha$ and $1-\beta$ to adjust our confidence level and power for the hypothesis test: (1) α controls the probability of rejecting our hypothesis when it is true and (2) $1-\beta$ controls the probability of rejecting the alternative hypothesis when it is false. Typically, the confidence level $1-\alpha$ is set to 0.95 and the power $1-\beta$ is set to 0.80 in practice [Desu and Raghavarao, 1990]. The z_α is the allowable number of incorrect records. If the incorrect number of records x is smaller than z_α, our hypothesis passes the test with confidence α and power $1-\beta$ over the alternative hypothesis. Given α, β, n, p_{upper} and p_{lower}, we can calculate z_α based on Formula 5.1.

Since the user ideally stops when there is no incorrect record in the sample, the x is zero and it is strictly smaller than z_α.

$$\begin{gathered} Pr(x < z_\alpha; n, p_{upper}) > 1-\beta, \\ where\ Pr(x < z_\alpha; n, p_{lower}) < \alpha \end{gathered} \tag{5.1}$$

Demanding a lower error percentage, a higher confidence and power of the hypothesis often requires a larger sample. For example, when $p_{lower} = 0.01$, $\alpha < 0.05$, the alternative hypothesis is $p_{upper} \geqslant 0.02$ and $1-\beta > 0.8$, we need a sample of 910 records. Meanwhile, if we change the $p_{lower} = 0.001$ and $p_{upper} \geqslant 0.002$, the minimal sample size is 9635. However, we can configure the parameters to achieve the balance between sample size and level of confidence to meet different user requirements.

5.3 Recommending records

Our approach automatically examines the records in the sample to identify potentially incorrect records (line 2 to line 9 in Algorithm 5). It then sorts these records and recommends one for the users to examine.

Our approach has two phases. First, it identifies the *records* ($R_{runtime}$) with runtime errors. Runtime errors here are the errors that occur during the execution of transformation programs and cause the programs to exit abnormally. Second, our approach identifies the *questionable records* ($R_{questionable}$) with potential incorrect results when there is no record with runtime errors.

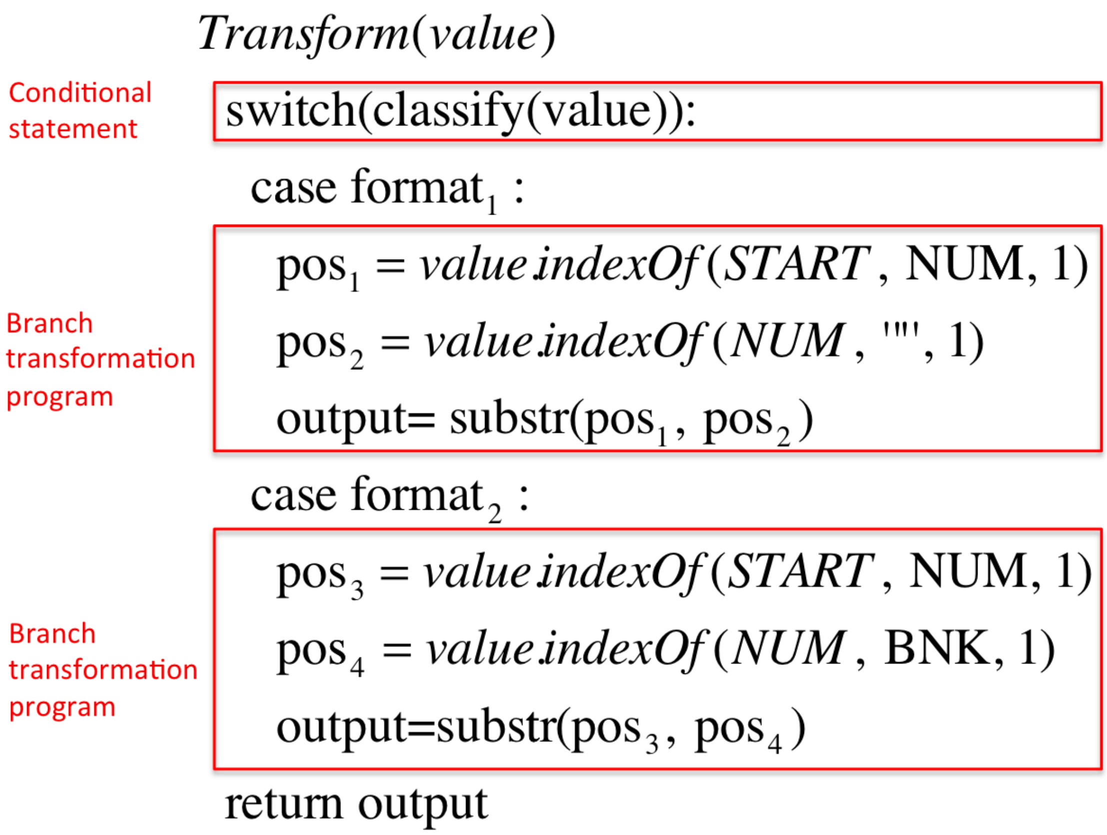
Algorithm 5: Algorithm for recommending records

```
Input: Set of all the records R, transformation program P and MetaClassifier F
Output: Recommended set of Records $R^*$
$R_{runtime}$ = [], $R_{questionable}$ = [], $R^*$ = []
1  $R_s$= sample(R)
   for record r in $R_s$ do
2      $r_t$ = applyTransformation(r, P)
3      if $r_t$ contains runtime error then
4          $r_t$.score = number of failed position programs
5          $R_{runtime}.add(r_t)$
       else
6          score = F.getScore($r_t$)
7          if $score < 0$ then
8              $r_t$.score = score
9              $R_{questionable}$.add($r_t$)
   if $R_{runtime}.isEmpty()$ then
10     sort $R_{questionable}$ acscendingly based on record score
11     $R^* = R_{questionable}$
   else
12     sort $R_{runtime}$ acscendingly based on record score
13     $R^* = R_{runtime}$
   return $R^*$
```

5.3.1 Finding the records with runtime errors

Our approach finds the records that executing transformation programs on these records show runtime errors. These records cause the learned program to exit abnormally in execution. There are mainly two types of runtime errors: (1) the position program cannot locate a position and (2) the segment program has a start position larger than the end position.

For example, the program shown in Figure 5.3 is learned from the first three records shown in Figure 5.1(b). The approach now applies this learned program to a new input "H: 24 x W: 7 " to extract the first degree information. The program uses the first branch program program to transform this record. The start position program cannot



```
Transform(value)
switch(classify(value)):
  case format_1 :
    pos_1 = value.indexOf(START, NUM, 1)
    pos_2 = value.indexOf(NUM, '"', 1)
    output= substr(pos_1, pos_2)
  case format_2 :
    pos_3 = value.indexOf(START, NUM, 1)
    pos_4 = value.indexOf(NUM, BNK, 1)
    output=substr(pos_3, pos_4)
return output
```

Figure 5.3: An example transformation program

locate a position in the input and output "-1", as "24" does not appear at the beginning of the input. The end position program cannot locate a position either, as there is no """" after 24. The corresponding segment program also has a runtime error too, as both its position and end position are smaller than 0 ("-1"). The other case of runtime error is that the segment program has a start position bigger than the end position. The two position program of the segment program both successfully locate two indexes in the input. However, the end position is before the start position, which will cause a runtime error of the segment program.

To identify the records with runtime errors, our approach simply applies the learned program to the sampled records, collects all the records with runtime errors and puts them into the set $R_{runtime}$.

5.3.2 Building a meta-classifier for detecting questionable records

The set of binary classifiers used for building the meta-classifier can be categorized into 3 types: (1) classifiers based on the distance (f_{dist}), (2) classifiers based on the agreement of different programs ($f_{program}$), and (3) classifiers based on the format ambiguity ($f_{ambiguity}$).

5.3.2.1 Classifiers based on distance

This type of classifier calculates the distances from records to a set of records. Based on the distribution of the distances, it identifies the records with distances that are larger than certain standard deviations from the chosen references. To calculate the distance between two records, the approach first converts the records to feature vectors and then calculates the Euclidean distance between the two vectors. The features used here are the same as the ones introduced in the previous section.

This type of classifier ($f_{dist}(e_i|\, r,\, t,\, c)$) is shown in Function 5.2. It classifies each record e_i as either a correct record (1) or an incorrect record (-1) based on its distance ($d_{e_i,r}$) from the reference (r). N is the number of records. Each classifier of this type can be characterized using a triple (r, t, c). r represents the *reference*. It has two values: "all records" or "examples". The string "all records" specifies that the approach calculates the distance between the feature vector of the record (e_i) with the mean vector of all records except e_i. The string "examples" means that the approach computes the distance from the record (e_i) to the mean vector of the examples. The t has three values: "input", "output" or "combined". "input" and "output" here mean that only inputs or outputs of the records are used to create feature vectors. "combined" means the input and output are concatenated into one string to create the feature vector. The σ is the distance standard deviation. The $c\sigma$ indicates the number (c) of standard deviations. For example, a classifier $f_{dist}(e_i|\text{“}examples\text{''},\, \text{“}input\text{''},\, 1.8)$ identifies the records whose inputs are more than 1.8 standard deviations (σ) away from the mean vector of the inputs of examples .

$$f_{dist}(e_i|\, r,\, t,\, c) = \begin{cases} -1 & d_{e_i,r} > c\sigma \\ 1 & d_{e_i,r} \leqslant c\sigma \end{cases} \quad (5.2)$$
$$where\, \sigma = \sqrt{\tfrac{1}{N}\sum_i (d_{e_i,r} - \mu)^2},\ \mu = \tfrac{1}{N}\sum_i d_{e_i,r}$$

Since parameter triple (r, t, c) has many configurations, our approach generates all configurations. Here, the value for c is selected a set of predefined decimal numbers. The approach uses each configuration to create a binary classifier and adds the classifier into the library of classifiers.

5.3.2.2 Classifiers based on the agreement of programs

This type of classifier identifies the records that consistent programs disagree about the transformation results. Typically, our approach can generate multiple programs that are consistent with given examples. Each program can be considered as an interpretation of the examples. The classifiers of this type maintain a set of programs that are consistent with the examples. The binary classifier identifies the records (output -1) that the programs produce different results for the same input. Providing examples for these records can help to clarify a user's intention and guide the system to converge to the correct programs. However, to build such classifiers, directly generating all the consistent programs and evaluating all these programs on records is expensive, as there are usually a large number of consistent programs given a few examples.

To reduce the computational cost in building this type of classifier, we exploit the fact the approach can independently generate the position programs. Our approach generates all the consistent position programs instead of the whole branch programs and evaluates these position programs on the records. This modification greatly reduces the number of

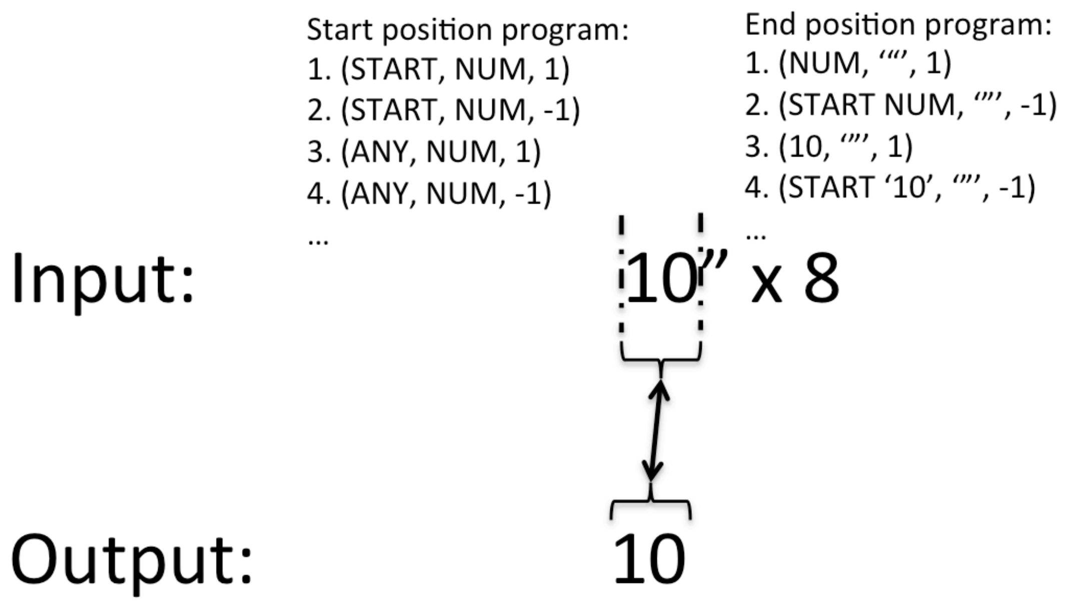


Figure 5.4: Candidate position programs for one segment program

programs that our approach is required to generate and evaluate, as the set of complete programs can be considered as a Cartesian product of sets of position programs. For example, in Figure 5.4, the start and end position of the substring "10" can both be represented using a set of programs. Every start position program can combine with any end position program to form a segment program. There would be 16 segment programs if there are 4 start position programs and 4 end position programs. But the total number of start and end position programs is only 8. Our approach only needs to generate and evaluate 8 position programs instead of 16 segment programs.

5.3.2.3 Classifiers based on the format ambiguity

This type of classifier aims to capture those records that are potentially labeled with the wrong format. The transformation program contains a conditional statement, which is used to recognize the format for a record before applying any transformation. Currently, we use an SVM multi-class classifier as the conditional statement as described in the previous work section. The SVM classifier not only classifies the records to their formats but also outputs the probability of the record belonging to that format. To identify the records that are potentially labeled incorrectly by the SVM classifier, our approach selects the records with the probability below a threshold θ. To select the right θ, our approach first creates a classifier using each θ in a predefined set and adds it into the classifier library. Later, our approach selects the classifier with the θ having the best performance described in the next section.

Combining classifiers using ADABOOST: we use ADABOOST [Freund et al., 1996] to combine classifiers above to create a meta-classifier ($F(e)$) for classifying whether a record (e) is transformed correctly as shown in Function 5.3. The meta-classifier outputs -1 for incorrect records and 1 for correct records. The output is 1, if the weighted sum of the output of a set of binary classifiers (f_i) is no less than 0; -1 if the sum is negative. During training ADABOOST iteratively selects the binary classifier (f_i) from a pool of classifiers described above to minimize the error on the misclassified training instances. It also assigns weights (w_i) to these classifiers indicating their importance in the final meta-classifier.

$$F(e) = sign(\sum_i w_i f_i(e)) \tag{5.3}$$

Our approach only uses ADABOOST to select the binary classifiers and learn their weights **once** to create the meta-classifier. Our approach uses this meta-classifier for all future transformations. Notably, the meta-classifier only defines the binary classifiers to be used and the weights for them. The approach still learns the binary classifiers constituting the meta-classifier for each specific transformation. Our approach learns the parameters for the binary classifiers (f_i) from the examples, records and consistent programs that are unique to each iteration. It combines these learned binary classifiers with the assigned weights to create the meta-classifier. The approach can use the meta-classifier with the learned binary classifiers to identify incorrect records.

5.3.2.4 Sorting the recommended records

As the approach recommends multiple records, it places the records that are more likely to be incorrect and contain more valuable information on the top of the recommendation area shown in Figure 5.2. This saves users' time in examining the recommended records and the system can also obtain more informative examples from the users. Our approach calculates a score for each record to measure how likely a record is incorrect and how much information it can provide in synthesizing the program.

The records with runtime errors are all incorrect. The score for these records is the number of failed subprograms including segment and position programs. A higher score means the approach can learn more information from this record, if the user provides an example for this record. The approach sorts these records in a descending order.

As to the records without runtime errors, we assume that the records that are more likely to be incorrect can provide more information for the system. The approach uses $-\sum_i w_i * f_i(x)$ in Function 5.3 as the score for each record. A higher score indicates more classifiers or the classifiers with heavier weights consider the record to be incorrect. The approach sorts these records in descending order.

5.3.3 Minimal test set

We want to ensure that a user labels a minimum number of records, users are recommended to validate at least one record in a minimal set of records by either confirming the correctness of the record or entering a new example for that record. As mentioned before, there are multiple consistent programs given a set of examples. These programs conflict with each other as they generate different results on certain records. The minimal test set contains the records that these consistent programs disagree on the outputs. Ideally, we should ask users to verify the outputs of all the programs to identify the correct programs. However, fully generating all the programs and executing them on records is infeasible in practice for two reasons: (1) the users are waiting for the responses on the fly and (2) the infinite number of conditional statements as there can be an infinite number of decision hyperplanes in the feature vector space. Our approach only generates all the consistent position programs and evaluates them on the records to approximate all the programs that should be tested. To identify the minimal test set, our approach simply uses the same set of records that are labeled as incorrect by the classifier based on the agreement of programs. Our approach highlights these records with blue borders in the GUI as seen in Figure 5.2. When the minimal test set is empty, there are not conflicting position programs.

5.4 Evaluation

To evaluate the performance of our approach, we performed simulated experiments and a user study to compare our system with alternative approaches. Transforming a dataset usually requires several iterations in the evaluation. An *iteration* starts when the user provides an new example and ends when the system has learned the transformation program and applied the program to the rest of the data records.

5.4.1 Simulated experiment

There are two goals of this experiment: (1) to test whether our recommendation can capture the incorrect records, and (2) to test whether our approach can place at least one incorrect record on top so that users can easily notice these incorrect records.

5.4.1.1 Dataset

We used the 30 scenario described in Appendix. Each scenarios contains about 350 records. The data was gathered from student mash-up 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. Each scenarios contains two columns of data. The first column shows the raw data and the second column shows the transformed data.

5.4.1.2 Experiment setup

To collect the training data for learning the meta-classifier, we should have both the transformation results and the labels to indicate whether these records are correct or not. Our approach first chose a record, provided the expected output and used it as an example. It learned the transformation program and applied the program to the rest of records. It compared the transformed data with the expected output and labeled each record as correct or incorrect. It also calculated the confidence of the conditional statement on each record. After collecting the data for the iteration, it started a new iteration by identifying the first incorrect record and providing an example for that record. The process ended when all the records were transformed correctly. Our approach collected training data from all iterations of the 30 scenarios. We divided all the scenarios into 5 groups and ran 5-fold cross-validation. Our approach trained the meta-classifiers using 4 groups of training data and tested the meta-classifier on the remaining group.

We used two metrics below to evaluate our approach and alternative approaches in each scenario:

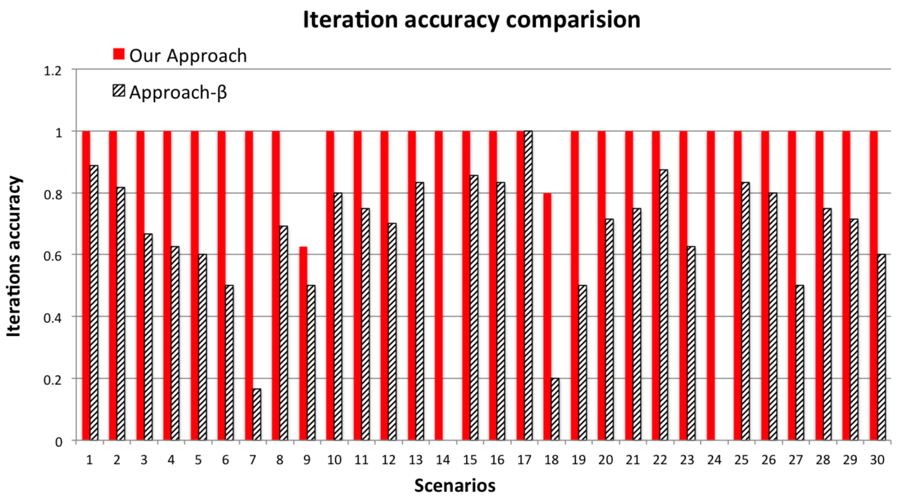
- iteration accuracy: the percentage of iterations that our recommendations contain at least one incorrect record out of all the iterations having incorrect records.
- mean reciprocal rank (MRR): the average of the reciprocal rank of the first identified incorrect record. Q is the total number of iterations and $Rank_i$ is the index of the first incorrect record in the recommended list in the i-th iteration. If the recommendation fails to include the incorrect record and there exists one, the $\frac{1}{Rank_i}$ is set to 0.

$$MRR = \frac{1}{Q}\sum_{i=1}^{Q}\frac{1}{Rank_i}$$

We compared our current approach with our previous approach (Approach-β) [Wu and Knoblock, 2014] and a baseline approach. The **Approach-β** also provides recommendations for users to examine. The **baseline** approach recommends all records in the sample for users to examine. It randomly shuffles these transformed records for users to examine.

5.4.1.3 Results

As shown in Figure 5.5, our approach accurately captured the incorrect records in the recommendation. The average of iteration accuracy of our approach in all scenarios is 0.98 compared to 0.83 with the Approach-β. The iteration accuracy is left blank for the baseline approach. The baseline simply recommended all the records in each iteration so that it had a 100% iteration correctness on all scenarios. The improvement is mainly due to two reasons. First, our approach recommended multiple records compared to that Approach-β only recommended one record in every iteration. Second, our meta-classifier is an ensemble of a library of classifiers. The classifier used in Approach-β



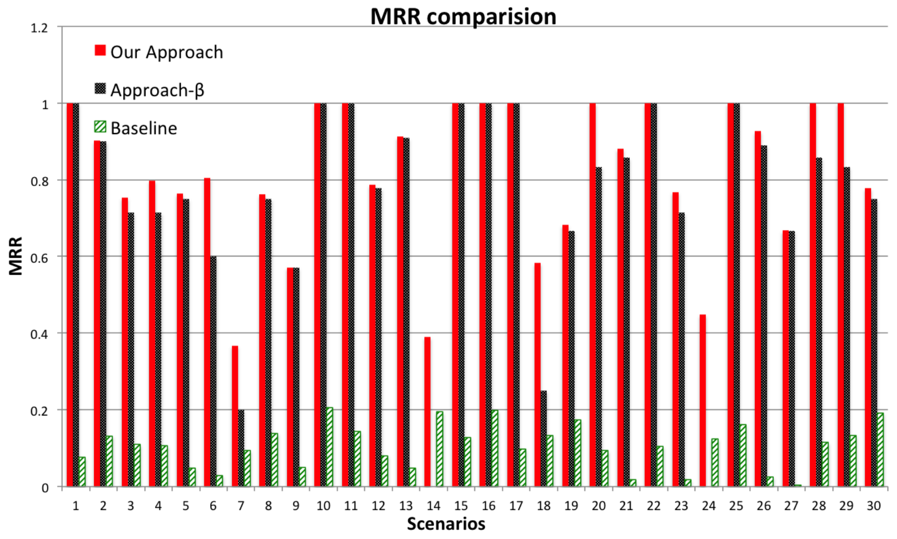


Figure 5.5: Comparison results

is just one in the library. This ensemble of classifiers enables our approach to capture a boarder range of incorrect records. Only scenario 9 and 18 have iterations in which our approach failed to detect the incorrect records. These iterations require examples for unseen input formats that are similar to previous examples, which make the system fail to detect the difference. One example of the scenario 9 is shown in Table 5.1. The users intended to extract the full prices for student tuitions (1st and 2nd record). Since the third record only has the credit price, it should be transformed to “NULL”. However,

Raw	Transformed
$33,926 per year (full-time)	$33,926
$42,296 per program (full-time)	$42,296
$1,286 per credit (full-time)	**$1,286**

Table 5.1: One typical example of a failed iteration

given only the first and second record as examples, our approach did not know the user required a different transformation for the third record, as it shared a very similar format with two previous records. Thus, our approach did not recommend the 3rd record as a potential incorrect record for users to examine.

Our approach can place the incorrect records on top of the recommendation, as the average MRR of our approach in all scenarios is 0.75. It saved users' time from fully exploring a long list of records. The average MRR of Approach-β in all scenarios is 0.68. It did not place the incorrect record on top for most of iterations without runtime errors. The MRR of the baseline approach was calculated based on the index of the first incorrect record in the transformed records. We can see that both our approach and Approach-β are well above the baseline, as the randomized shuffling can place the incorrect records in the middle of the list.

5.4.2 User study

We performed a user study to evaluate our approach in real use cases. The goal of this experiment is to test whether the users using our approach can achieve better correctnesses than the users of Approach-β with no more user effort.

5.4.2.1 Dataset

We collected 5 scenarios with about 4000 records for each scenario on average to evaluate the approaches. The first 2 records and the description of the scenarios are shown in Table 5.2 to demonstrate the transformation. These transformations involve transforming text into URIs by adding prefixes, replacing blank spaces with underscores or reordering substrings such as s1, s2 and s5. The remaining of scenarios focus on extracting substrings from the inputs such as s3 and s4.

5.4.2.2 Experiment setup

We used two metrics to measure the user performance (1) correctness, which is the percentage of correct records when the users stopped transforming and (2) time, which is the average time (in seconds) used by users in one *iteration*.

We recruited 10 graduate students and divided them into two groups: $group_A$ and $group_B$. We asked users in $group_A$ to use our system and asked users in $group_B$ to use the Approach-β.

We used the training data gathered in the simulated experiment to train our meta-classifier for recommending records. The p_{lower} and p_{upper} were set to 0.96 and 0.99. Our approach sampled 300 records in every iteration.

Scenario	description	Input	Output
s1	change into URI	WidthIN	http://qudt.org/vocab/unit#Inch
		HeightCM	http://qudt.org/vocab/unit#Centimeter
s2	change into URI	Dawson, William	William_Dawson
		Lauren Kalman	Lauren_Kalman
s3	extract issue date	Thor I#172 (January, 1970)	January, 1970
		Machine Man II#2	NULL
s4	extract first degree	7 x 9 in.	7
		6 13/16 x 8 7/8 in.	6 13/16
s5	change into URI	American	thesauri/nationality/American
		South African	thesauri/nationality/South_African

Table 5.2: Scenarios used in user study

5.4.2.3 Results

The results of the user study are shown in Table 5.3. Our approach achieved a correctness higher than 0.99 in all scenarios. These correctnesses were within the user expected correctness range. Compared with Approach-β, we can see our approach also achieved better correctnesses in all 5 scenarios. Users in $group_A$ not only had higher correctness rates, but also used less time per iteration for 4 out of 5 scenarios and used the same amount of time on the first scenario. We performed paired one-tail t test for the hypothesis that *our approach uses less time and has higher correctness than Approach-β*. The result shows that the improvements are statistically significant ($p < 0.05$).

Scenario	IPBE			Approach-β		
	Correctness	Time (sec)	Example#	Correctness	Time (sec)	Example#
s1	1	16	11.4	0.828	16	9
s2	0.998	17	13.4	0.994	26	9
s3	0.992	16	11.6	0.873	36	8.7
s4	0.997	14	14	0.983	17	11
s5	0.999	12	5.4	0.872	22	4.3
Average	0.997	15	11.1	0.91	23	8.4

Table 5.3: User study results

In the user study, we found that the users in $group_A$ provided more examples than users in $group_B$, as users in $group_A$ can simply click a button to confirm a correct record as a new example. The users confirmed several examples (2 - 5 examples) before stopping transformation. Thus, the number of examples (Example$\#$) provided by the users in $group_A$ is higher than the numbers in $group_B$ as shown in Table 5.3. The $group_A$ users provided 11.1 examples and users in $group_B$ provided only 8.4 examples on average. Providing more examples gives users more chances to refine the recommendation and examine more records, which in turn leads to a higher correctness rate. Moreover, the users in $group_A$ also used less time. We found when the recommendation contained incorrect records on top, it largely reduced the time users used to examine the results compared to the time spent by users to directly examine the results when the recommendation failed to capture the incorrect records.

Chapter 6

Related Work

In this chapter, I first discuss all the related approaches to provide a broad overview of the approaches developed to handle data transformation. Second, I focus on the PBE approaches developed for data transformation.

6.1 Data Transformation Approaches

Data transformation aims to transform the data from the source format to the target format. Depending on the task, data transformation can refer to different operations, such as mapping numerical data into a specific range, or conversion between specific image formats, such as GIF to PDF. In this research, we focus on one of the most general problems in data transformation, which is **semi-structured text format transformation**. The approaches based on the inputs provided by the users can be categorized into 2 types: (1) users specify the transformation steps (2) users only specify the results.

6.1.1 Specifying the transformation steps

To use this type of system, users should tell the system step by step how to perform the transformation. This type of approach is often based on a domain specific language (DSL) to hide the users from directly working with low-level programming languages, which can save users the time in developing transformation programs. Users can also define their own high level functions to further save the time, such as Excel Macro [Microsoft Excel] and OptiWrangler [Sujeeth et al., 2013]. Recent work in this category focuses on providing intelligent user interfaces to accelerate the process of developing DSL based programs, such as OpenRefine [Huynh and Stefano], Potter's Wheel [Raman and Hellerstein, 2001], SmartEdit Lau et al., 2003 and Data Wrangler [Kandel et al., 2011], etc. They allow the users to specify edit operations in their GUI and view the results as the users write the programs. OpenRefine is a tool for cleaning messy data. Its language supports regular expression style of string transformation and data layout transformation. Users can directly write scripts in the GUI and view the results immediately. Potter's Wheel defines a set of transformation operations and let users gradually build transformations by adding or undoing transformations in an interactive GUI. SmartEdit can learn from a user's edit operations and generate a sequence of text editing programs using the version space algebra. Data Wrangler is an interactive tool for creating data transformation. It uses the transformation operations defined in Potter's wheel. It can learn, rank, and suggest the edit operations from users' selections. Besides supporting string level transformation, it also supports data layout transformation including column split, column merge, fold and unfold.

Typically, since the users can specify the transformation steps, these approaches usually can support a border range of transformations besides the semi-structured text format transformation. Users also have more control and insight over the transformation steps applied. Our approach is different from these systems as our approach only asks users to provide the output without requesting specific steps from the users.

6.1.2 Specifying the transformation results

The recent PBE approaches [Gulwani, 2011; Singh and Gulwani, 2012a; Le and Gulwani, 2014; Harris and Gulwani, 2011] only require users to provide target outputs. These approaches require examples from users and synthesize programs that are consistent with these examples. These systems enable users to focus on using their data without spending time in figuring out how to transform the data. However, since these approaches synthesize the complete programs automatically, the DSL's expressiveness is often restricted to ensure the programs can be synthesized efficiently. Our approach belongs to this category. The differences will be discussed in more details in the following section.

6.2 PBE approaches for data transformation

PBE approaches have been extensively studied for the past decades. Early work [Kushmerick, 1997; Hsu and Dung, 1998; Muslea et al., 1999] in wrapper induction learns extraction rules from user labels to extract target fields from documents. Lau et al., 2003 proposed an approach to derive text-editing programs from a sequence of user edit operations. However, these approaches typically require separate labels for each field or labels for each step.

Much work also has been done in inductive programming [Kitzelmann, 2009] to synthesize programs from examples. The pioneering work in program induction [Summers, 1977] can induce Lisp programs with one recursive function from the traces of input-output pairs. Kitzelmann and Schmid, 2006 extended this approach to induce a set of recursive equations with more than one recursive call. MIS [Shapiro, 1981] uses first order logic to represent the examples and induced programs. Recently, Muggleton and Lin, 2013 developed an meta-interpretive learning technique, which support recursion and is able to learn new predicates. These new predicates can be used to expand the set of predefined functions in DSL. Lin et al., 2009 applied this technique to successfully generate string transformation programs.

Compared to the existing PBE approaches, our approach utilizes the information from previous iterations in generating data transformation programs. By exploiting the information, our approach improves the performance of the existing approaches in three key components: (1) learning the conditional statements (2) synthesizing the branch programs and (3) the user interface.

6.2.1 Learning Conditional Statements

The closely related PBE approaches that involve learning conditional statements are as follows. SMARTpython [Lau, 2001] 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. Data Wrangler [Kandel et al., 2011] 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. Our approach is different from the two approaches as our approach learns a multi-class conditional statement. It can include more than two branches in the program indicating it can handle more than two kinds of inputs at the same time.

APE [Ruvini and Dony, 2000] is a programming assistant, which can recognize programmer's repetitive actions under certain situations. It can then suggest performing these repetitive tasks for the programmer when it is in the same situation. It learns a situation pattern to recognize the precondition of the repetitive actions so that it could automatically suggest those operations when the pattern matches. Gulwani, 2011 developed an approach that 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 Gulwani, 2011 are built based on conjunction or disjunction of a predefined set of predicates, which makes it hard to express the nominal values of the features, such as the counts of different tokens.

Since our approach also uses the collected constraints to learn a distance metric for partitioning the examples and learning the conditionals, we also review the related distance metric learning work here. There is a large body of work in metric learning [Yang and Jin, 2006]. Researchers applied distance metric learning in various clustering algorithms. Xing et al., 2002 proposed learning a Mahalanobis distance metric and applied it in a K-means algorithm. Davidson and Ravi, 2009 investigated applying instance-level must-link and cannot-link constraints in agglomerative clustering, which shows the feasibility of the problem. Bade and Nurnberger, 2006 described an approach that learned a distance metric to perform agglomerative clustering by introducing relative instance-level constraints. Zhao and Qi, 2010 extended instance-level constraints to order constraints to capture the hierarchical side information. Zheng and Li, 2011 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 approach 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. Moreover, we first applied this semi-supervised clustering approach in the program synthesis setting, which can effectively utilize constraints to improve the system performance.

6.2.2 Adapting Program With New Examples

More recently, several approaches show that reusing the previous subprograms is promising. Perelman et al., 2014 focuses on developing an approach to synthesize programs for various domains given the DSL. Their approach can reuse previous subprograms. It maintains two sets: (1) one set, called contexts, containing the programs with some of its subprograms deleted to create holes and (2) the other set containing all the subprograms from previously generated programs. Through filling the subprograms of the second set into the holes in the contexts of the first set, it can create new programs. Lin et al., 2014 uses the meta-interpretive learning framework [Muggleton and Lin, 2013] to learn domain specific bias. By reusing the predicates generated from other tasks or previous iterations, their approach can use fewer examples and generate programs more efficiently. Our work is orthogonal to these works. These works focus on maintaining a library of previous generated subprograms and reusing these programs when encountering new examples. As the number of subprograms in the library keeps increasing, searching in this library for the right subprograms can be time consuming. Our approach takes advantage of traces to deterministically identify, refine incorrect subprograms, and reuse correct subprograms.

A closely related area of program adaptation is program bug repair. Shapiro, 1991 developed an approach to deterministically adapt programs with new evidence using resolution-tree backtracking. Recently, approaches using generic programming to generate fairly complicated software patches have been applied to automatic bug fixes [Weimer et al., 2010; Goues et al., 2012]. These approaches often require either an oracle to test whether certain parts of the program are correct or require a large number of test cases to locate the problem. Our approach is different from these approaches as it automatically creates the expected outputs for the subprograms using the given examples.

6.2.3 User interface

Our user interface is intended to help users verify the correctness of the results with minimal effort. Closely related work generally performs the transformation result verification based on two strategies: (1) adapting existing white-box testing techniques and

(2) shifting users' attention to potential incorrect records and let users examine these records.

The most noticeable work in the first category is "What You See is What You Test" (WYSWYT). Rothermel et al., 2001, 1997 developed an approach to test spreadsheet programs by asking users to provide test cases through validating the correctness of the outputs for certain inputs. To ensure it has obtained enough test cases, their approach proposed a code-based criterion called definition-use coverage to find the values that users should validate. Furthermore, Burnett et al., 2003 introduced assertions into end-user program testing. It allows users to specify their expectations and convert them into assertions. Our work is different from these works as the learned program changes when users provide a new example. Our approach updates the test plan for the new program by updating the minimal test set and recommendation list. Moreover, recommended records can also help users explore the dataset to allow them to notice the unexpected inputs to refine their programs, which is more related to helping users understand the problem requirements rather than merely testing.

The second type of work identifies potentially incorrect results based on certain predefined heuristics. Gulwani, 2011 can highlight the entries that have two or more alternative transformed results. This method generates multiple programs and evaluates these programs on all the records to identify the records with different results. Cue-Flik [Amershi et al., 2009] shows users an overview of the learned concept. Users can examine this overview to provide new examples. This overview is essentially a high level abstraction of the instances in the image feature space. LAPIS [Miller and Myers, 2001] highlights the texts that have potentially incorrect matches. Their approach identifies the matches that are different from the majority of matches. Wolfman et al., 2001 extended the approach by Lau et al., 2003 by reducing the user effort using a mixed initiative approach combining several interaction modes. Compared to these approaches, our approach first focus on a large dataset with thousands of records. Moreover, to address the users' over-confidence, our approach asks users to validate a minimal set of records. Our previous version of the system [Wu et al., 2014] only recommends one potentially incorrect record for users to examine and the recommendation is only based on the distance from the records to the examples. Our current approach extends the previous version of the system to handle large datasets. It can recommend multiple records and learn the task-dependent rules for identifying incorrect records for specific scenarios.

Chapter 7

Conclusion

PBE approaches enable people to transform the data without coding. However, current PBE approaches face the challenge that they should synthesize correct programs for large datasets with various formats in real time. To address this challenge, we developed an iterative PBE approach (IPBE) for data transformation.

Our approach is based on an observation that users interact with the system in an iterative way. The transformation process usually consists of several iterations of transforming and verifying the results before transforming the data into the right format. During every iteration, a program consistent with the given examples is generated to transform the data. Based on this observation, our approach generates a program based on both (1) current examples and (2) the information collected from previous iterations. By utilizing the information, our approach can generate programs efficiently for heterogeneous data with a few examples. It also enables users to effectively examine the results and obtain the correct program with less effort compared to a start-of-the-art approach [Gulwani, 2011].

The list of contributions of our approach is as below:

- efficiently learning accurate conditional statements by exploiting information from previous iterations [Wu and Knoblock, 2014] (Chapter 3)
- incrementally synthesizing branch transformation programs efficiently by adapting programs from the previous iteration [Wu and Knoblock, 2015] (Chapter 4)

- maximizing the user correctness with minimal user effort on large datasets by recommending potentially incorrect records [Wu et al., 2014] (Chapter 5).

With the performance improvement obtained by utilizing the information from previous iterations, many similar PBE approaches in other domains can adopt the same idea to improve their performance. For example, table layout transformation [Harris and Gulwani, 2011] can also leverage the previous intermediate results to improve the performance in synthesizing layout transformation programs.

With our approach, synthesizing more complicated programs becomes practical, which enables us to apply our approach to a board range of problems. For example, our approach can efficiently learn powerful conditional statements directly from examples in real time without relying on background knowledge. Our approach can learn a conditional statement that can recognize the text representation of 12 months and synthesize programs to convert them into corresponding numbers. Our approach can also learn programs from long and many examples. It enables users to directly work on many problems that were not practical before. For example, users can provide examples for several columns of data at the same time as shown in the Section 4.2.4. This allows users to not only split or merge several columns in the spreadsheet but also change their contents at the same time. Finally, our system is so far the only open-sourced program that is based on Gulwani's work [Gulwani, 2011]. Existing applications in the data processing area can integrate our system as a new module to revolutionize the users' experiences in data transformation.

7.1 Limitations and future work

Based on our experiences, we identified several limitations of existing PBE approaches and we also believe these limitations provide opportunities for future work in this area.

7.1.1 Managing the user expectation

PBE approaches often give users a false impression that these systems can generate the programs that are consistent with any given examples. In fact, whether the system can successfully generate the programs largely depends on whether the DSL can express those programs. Existing DSLs used in PBE systems still cannot fully support all the common data transformation operations. Without informing users the capability of the system beforehand and letting them understand the boundary of the system's capability through trial and error, the PBE systems would quickly lose users' trust. To solve this problem, one possible approach is to present the synthesized programs in a way that users can easily understand. Through reading these programs, the users can gain insights into the kinds of the programs that the PBE systems are capable of generating. Reading these programs also helps the users to check whether the programs are the same as they expected.

7.1.2 Incorporating external functions

DSL essentially specifies the organization of a limited number of predefined functions. However, many existing transformation programs require the support of many third-party functions or services. For example, unit conversion is very common in data transformation, such as US dollars to Chinese RMB. Moreover, the conversion function is changing constantly as the conversion rate fluctuates. To solve this problem, designing a DSL containing all the functions may exceed existing systems' computational power, However, we can provide an approach that allows users to change the DSL by adding or removing several third-party functions according to their own requirements. This can significantly improve the functionality of existing approaches.

7.1.3 Handling user errors

Users tend to provide examples with errors. Moreover, it is hard for the users to realize that it is their errors that cause the system to fail to generate expected results without blaming the system. For example, a common error in entering examples is caused by blank spaces. The users often unintentionally enters different number of blank spaces in examples used for demonstrating the same transformation. It make the system treat these examples as demonstrations for different transformations. The system then learns incorrect conditional statements and branch programs. In the future, PBE approaches should not treat the user-entered examples dogmatically. They should either provide feedbacks to the users to inform them of a potentially incorrect example or guess the most likely programs given examples with errors.

Bibliography

Amershi, S., Fogarty, J., Kapoor, A., and Tan, D. (2009). Overview based example selection in end user interactive concept learning. In *UIST*. 6.2.3

Bade, K. and Nurnberger, A. (2006). Personalized hierarchical clustering. In *WI*. 6.2.1

Burnett, M. M., Cook, C. R., Pendse, O., Rothermel, G., Summet, J., and Wallace, C. S. (2003). End-user software engineering with assertions in the spreadsheet paradigm. In *ICSE*. 6.2.3

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. 3.1.4, 2

Davidson, I. and Ravi, S. S. (2009). Using instance-level constraints in agglomerative hierarchical clustering: Theoretical and empirical results. *Data Min. Knowl. Discov.* 3.2.4, 6.2.1

Desu, M. and Raghavarao, D., editors (1990). *Sample size methodology*. Academic press Inc. 5.2

Feldman, R. and Sanger, J. (2006). *Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press. 5

Freund, Y., Schapire, R. E., et al. (1996). Experiments with a new boosting algorithm. In *ICML*. 5.3.2.3

Goues, C. L., Nguyen, T., Forrest, S., and Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng*. 6.2.2

Gulwani, S. (2011). Automating string processing in spreadsheets using input-output examples. In *POPL*. 1.1, 1.2, 1.2, 2, 2.1.1, 3, 3.2, 3.2.2, 3.2.4, 4, 1, 4.2.2, 5, 6.1.2, 6.2.1, 6.2.3, 7

Harris, W. R. and Gulwani, S. (2011). Spreadsheet table transformations from examples. In *SIGPLAN*. 4, 6.1.2, 7

Hsu, C.-N. and Dung, M.-T. (1998). Generating finite-state transducers for semi-structured data extraction from the web. *Inf. Syst.* 6.2

Huynh, D. F., Miller, R. C., and Karger, D. R. (2008). Potluck: Data mash-up tool for casual users. *Web Semant.* 1.1

Huynh, D. F. and Stefano, M. *OpenRefine http://openrefine.org*. 6.1.1

Kandel, S., Paepcke, A., Hellerstein, J., and Heer, J. (2011). Wrangler: interactive visual specification of data transformation scripts. In *CHI*. 1.1, 6.1.1, 6.2.1

Kitzelmann, E. (2009). Inductive programming: A survey of program synthesis techniques. In *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers*. 6.2

Kitzelmann, E. and Schmid, U. (2006). Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*. 1, 4, 6.2

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Comput. Surv.* 1.1, 5

Kushmerick, N. (1997). *Wrapper Induction for Information Extraction*. PhD thesis, University of Washington. 6.2

Lau, T. (2001). *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington. 6.2.1

Lau, T., Wolfman, S. A., Domingos, P., and Weld, D. S. (2003). Programming by demonstration using version space algebra. *Mach. Learn.* 1.1, 2, 4, 6.1.1, 6.2, 6.2.3

Le, V. and Gulwani, S. (2014). Flashextract: A framework for data extraction by examples. In *PLDI*. 6.1.2

Lieberman, H., editor (2001). *Your Wish is My Command: Programming by Example*. Morgan Kaufmann Publishers Inc. 4, 5

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J., and Muggleton, S. (2014). Bias reformulation for one-shot function induction. In *ECAI*. 4.2.1, 4.2.2, 4.2.3, 6.2.2

Lin, J., Wong, J., Nichols, J., Cypher, A., and Lau, T. A. (2009). End-user programming of mashups with vegemite. In *IUI*. 6.2

Mahalanobis, P. C. (1936). On the generalized distance in statistics. *Proceedings of the National Institute of Sciences (Calcutta)*. 3.1.3

Microsoft Excel. Excel macro. https://support.office.com/en-in/article/Work-with-macros-654cc76b-1c9c-4632-89e1-67230322e92f Last accessed: 2015-09-10. 6.1.1

Miller, R. C. and Myers, B. A. (2001). Outlier finding: Focusing user attention on possible errors. In *UIST*. 5, 6.2.3

Muggleton, S. and Lin, D. (2013). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. In *IJCAI*. 6.2, 6.2.2

Muslea, I., Minton, S., and Knoblock, C. (1999). A hierarchical approach to wrapper induction. In *AGENTS*. 6.2

Panko, R. R. (1998). What we know about spreadsheet errors. *J. End User Comput.* 5

Perelman, D., Gulwani, S., Grossman, D., and Provost, P. (2014). Test-driven synthesis. In *SIGPLAN*. 6.2.2

Raman, V. and Hellerstein, J. M. (2001). Potter's wheel: An interactive data cleaning system. In *VLDB*. 6.1.1

Raza, M., Gulwani, S., and Milic-Frayling, N. (2014). Programming by example using least general generalizations. In *AAAI*. 1.1, 4

Rothermel, G., Burnett, M., Li, L., Dupuis, C., and Sheretov, A. (2001). A methodology for testing spreadsheets. *ACM Trans. Softw. Eng. Methodol.* 6.2.3

Rothermel, G., Li, L., DuPuis, C., and Burnett, M. (1997). What you see is what you test: A methodology for testing form-based visual programs. Technical report. 6.2.3

Ruvini, J.-D. and Dony, C. (2000). Ape: Learning user's habits to automate repetitive tasks. In *IUI*. 6.2.1

Shapiro, E. Y. (1981). An algorithm that infers theories from facts. In *IJCAI*. 6.2

Shapiro, E. Y. (1991). Inductive inference of theories from facts. In *Computational Logic - Essays in Honor of Alan Robinson*. 6.2.2

Singh, R. and Gulwani, S. (2012a). Learning semantic string transformations from examples. *Proc. VLDB Endow.* 4, 6.1.2

Singh, R. and Gulwani, S. (2012b). Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*. 4

Sujeeth, A. K., Gibbons, A., Brown, K. J., Lee, H., Rompf, T., Odersky, M., and Olukotun, K. (2013). Forge: Generating a high performance dsl implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*. 6.1.1

Summers, P. D. (1977). A methodology for lisp program construction from examples. *J. ACM*. 4, 6.2

Weimer, W., Forrest, S., Goues, C. L., and Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Commun. ACM*. 6.2.2

Wolfman, S. A., Lau, T. A., Domingos, P., and Weld, D. S. (2001). Mixed initiative interfaces for learning tasks: Smartedit talks back. In *IUI*. 5, 6.2.3

Wu, B. and Knoblock, C. A. (2014). Iteratively learning conditional statements in transforming data by example. In *Proceedings of the First Workshop on Data Integration and Application at the 2014 IEEE International Conference on Data Mining*. 5.4.1.2, 7

Wu, B. and Knoblock, C. A. (2015). An iterative approach to synthesize data transformation programs. In *IJCAI*. 7

Wu, B., Szekely, P., and Knoblock, C. A. (2014). Minimizing user effort in transforming data by example. In *IUI*. 6.2.3, 7

Xing, E. P., Ng, A. Y., Jordan, M. I., and Russell, S. (2002). Distance metric learning, with application to clustering with side-information. In *NIPS*. 6.2.1

Yang, L. and Jin, R. (2006). Distance metric learning: A comprehensive survey. *Michigan State Universiy*. 6.2.1

Zhao, H. and Qi, Z. (2010). Hierarchical agglomerative clustering with ordering constraints. In *WKDD*. 6.2.1

Zheng, L. and Li, T. (2011). Semi-supervised hierarchical clustering. In *ICDM*. 6.2.1

Appendix A

Appendix

The description of 30 scenarios used in my evaluation is list here. I only show two records for each scenario as demonstration. The data can be accessed at `https://github.com/areshand/IJCAI2015`. The latest code of IPBE is available as a part of Karma at `https://github.com/areshand/Web-Karma`

Id	File name	Description	Input	Outputs
s1	1st_dimension	extract the 1st degrees	26" H x 24" W x 12.5" D	26
			74" H x 31.5" W	74
s2	2nd_dimension	extract the 2nd degrees	26" H x 24" W x 12.5" D	24
			74" H x 31.5" W	31.5
s3	3rd_Data_json	extract the 3rd degrees	26" H x 24" W x 12.5" D	12.5
			74" H x 31.5" W	NULL
s4	avg_year	construct Excel avg function	1968 - 1970	avg(1968,1970)
			1970	1970
s5	birth	extract birth years	1835 - 1837	1835
			born 1925. Made by Knoll	1925
s6	comic	extract dates	Ravage 2099#24 (November, 1994)	November, 1994
			Fantastic Four Annual#26 (1993)	1993
s7	countries	normalize names	U.S.A.	USA
			United States	USA
s8	date_semantic	construct dates	Aug 29, 2011	29/8/2011

			June 21, 2003	21/6/2003
s9	s3_instate	extract instate tuitions	$43,930 per year (full-time)	43,930
			$11,704 per year (in-state); $29,016 per year (out-of-state)	11,704
s10	f1	complete addresses	1226 30	1226 30th st, Los Angeles, CA
			3416 walton ave	3416 walton ave, Los Angeles, CA
s11	huston_2nd	extract the 2nd degree	120 x 600 inches	600
			22 x 16 1/8 x 5 1/4 inches	16 1/8
s12	huston_3rd	extract the 3rd degrees	120 x 600 inches	NULL
			22 x 16 1/8 x 5 1/4 inches	5 1/4
s13	MOCA_dimension	extract the 2nd degrees	20 in HIGH x 24.25 in WIDE	24.25
			29.25 in HIGH x 26.125 in WIDE x .75 in DEEP	26.125
s14	Organization	encode texts	County	G1
			Hospital district or authority	P2
s15	s10_namehyphen	extract names	http://disney.wikia.com/wiki/Elsa_the_Snow_Queen	Elsa_the_Snow_Queen
			http://disney.wikia.com/wiki/Ursula	Ursula
s16	s1_age	extract ages	(1974-06-01) June 1, 1974 (age 39)	39
			(1924-04-12)April 12, 1924	NULL
s17	s2_date	extract dates	(1974-06-01) June 1, 1974 (age 39)	1974-06-01
			(1924-04-12)April 12, 1924	1924-04-12
s18	death	extract death years	1823-1880	1880
			born 1925. Made by Knoll	NULL
s19	s4_outstate	extract outstate tuitions	$43,930 per year (full-time)	NULL
			$11,704 per year (in-state); $29,016 per year (out-of-state)	29,016
s20	s5_3rd	extract the 3rd degrees	20 in HIGH x 24.25 in WIDE	NULL
			29.25 in HIGH x 26.125 in WIDE x .75 in DEEP	.75
s21	s5_name	construct names	Annica Ackerman	Annica Ackerman

			Jeff ”Bucko” Biggers	Jeff Biggers
s22	s6	extract names	Despair	Despair
			Untitled (Grindelia)	Grindelia
s23	s6_nickname	extract nicknames	Annica Ackerman	NULL
			Jeff ”Bucko” Biggers	Bucko
s24	s7_encode	encode texts	300 or more	3
			Between 100 and 299	2
s25	s8	construct names	Frishmuth, Harriet Whitney	Harriet Whitney Frishmuth
			Hopper, Edward	Edward Hopper
s26	s8_website	extract types	www.cascademedicalcenter.org	org
			www.uhs.net/cmh	net
s27	s9_tel	construct phone numbers	1-212-318-8000	(212) 318-8000
			262.243.7408	(262) 243-7408
s28	senator_name	construct names	Sen. Lisa Murkowski [R-AK]	Lisa Murkowski
			Rep. Eric ”Rick” Crawford [R-AR1]	Eric Crawford
s29	senator_nickname	extract nicknames	Sen. Lisa Murkowski [R-AK]	NULL
			Rep. Eric”Rick” Crawford [R-AR1]	Rick
s30	uri	construct names	dbpedia.org/resource/Virgin_Express	Virgin Express
			dbpedia.org/resource/Lufthansa	Lufthansa