Integrating Programming by Example and Natural Language Programming

Mehdi Manshadi, Daniel Gildea, James Allen
Department of Computer Science, University of Rochester
Rochester, NY 14627
{mehdih,gildea,james}@cs.rochester.edu

Abstract

We motivate the integration of programming by example and natural language programming by developing a system for specifying programs for simple text editing operations based on regular expressions. The programs are described with unconstrained natural language instructions, and providing one or more examples of input/output. We show that natural language allows the system to deduce the correct program much more often and much faster than is possible with the input/output example(s) alone, showing that natural language programming and programming by example can be combined in a way that overcomes the ambiguities that both methods suffer from individually and, at the same time, provides a more natural interface to the user.

Introduction

Recent work on Programming by Example or PbE, also referred to as Programing by Demonstration or PbD[1], (Lau et al. 2003; Liang, Jordan, and Klein 2010; Gulwani 2011) has shown the feasibility of learning simple programs (generally constrained text editing tasks) from small numbers of examples, that is input/output (I/O) pairs. Natural Language Programming, on the other hand, has received less attention, mainly because natural language carries a lot of ambiguity as opposed to formal (i.e. programming) languages in which no ambiguity is allowed. In this paper we motivate the integration of the two methods by offering the first PbE system that leverages a high-level natural language description of the programming task. We show that this system requires fewer examples, and hence lowers the complexity of PbE problem as well as the amount of effort from the user.

Our usage scenario and motivation is described in more detail in the next section. In the subsequent sections, we describe the baseline Programming by Example system, augment this baseline with probabilistic features, and integrate it with natural language descriptions. We then describe our train/test procedures, experiments, and results.

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]Although often used interchangeably, the two terms are not technically the same. The former only observes the beginning and the final states, but the latter requires step by step demonstration.

Motivation

Dijkstra (1979) states that natural language programming is simply impossible, or at least impractical, by arguing that formal symbolisms are "an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid". Although there is no doubt that, when it comes to precision, where no ambiguity is tolerated, the level of vagueness and ambiguity of natural language is a major drawback, we argue that in many practical situations there are ways to circumvent or to at least defer the necessity of using a formal language. The fact that many (novice) programmers actually use natural language programming as a major resource supports this claim. Programming forums contain thousands of posts in which programmers describe their problem in natural language and ask an expert for a piece of code solving that problem. Of course, for the very reason that Dijkstra mentions, a natural language (NL) description often raises lots of questions.

Our study of 242 posts on regular expression (regex) programming[2] shows that people adopt a key strategy to deal with ambiguity. In around 50% of the posts, they use some (often a single) example(s) to resolve the ambiguities that the NL description may raise. Looking at these posts, we can see how natural it feels for people to give a high-level description of a task and then one or more examples that demonstrate the details left out in the NL description. If this solution is found natural when dealing with human experts, why not use the same solution when dealing with machines?

For example, consider the sentence *Copy the first number on each line* describing a text editing task. It is quite hard to figure out the precise program intended by the user.

First, there are multiple interpretations of the sentence, two of them[3] demonstrated in Figures 1(a) and 1(b). In Figure 1(a), it is interpreted as to copy the first number occurring on each line of the input to the output. In Figure 1(b), the interpretation is to copy the first occurrence of a number in the text to all (other) lines of the input.

Second, many details are left out in the NL description. For example, what is the format of the output? Do we separate the output numbers by space, comma, newline, etc.? More importantly, what counts as a number? Does every se-

[2]http://forums.devshed.com

[3]Resulted from the two ways of preposition phrase attachment.

Source			Target
I5	5910	9-10 pm	5910
I90	3590	3-5 pm	3590

(a) An examples for the first interpretation

Source			Target		
100	Orange	CA	100	Orange	CA
	Apple	CA	100	Apple	CA
	Potato	ID	100	Potato	ID

(b) An example for the second interpretation

Figure 1: Two interpretations of the sentence *Copy the first number on each line*.

quence of digits count as number, like "5" in "I5" or "1" and "3" in "1-3 pm", or does it have to be surrounded by whitespace? Providing an example such as the I/O (source/target) pair in Figure 1(a) gives an answer to most of these questions and resolves a lot of the ambiguities. While providing all the details in the NL description may be painful, giving a single I/O pair is pretty straightforward. We follow this strategy and propose a system that takes a high-level NL description of a text editing task plus a single example of an I/O pair and generates a program achieving this task.

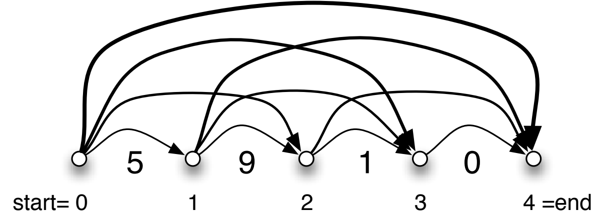
There are at least two ways to approach this, one from the perspective of natural language processing and one from a PbE point of view. The former processes the NL description and leverages the example(s) to resolve the ambiguities. The latter takes the opposite approach, that is, it takes the I/O pair(s) and generates the space of all programs that map the input(s) to the output(s). It then picks the one that is best supported by the NL description. In this paper we explore the latter approach, building on recent developments in PbE.

In order to show how an NL description helps a PbE system to generate the intended program, consider a task demonstrated by the example in Figure 1(a) with no natural language description. There are many programs that map the source to the target, a few of them given below: *Print the second column, Print the second-to-the-last column, Print the second sequence of digits, Print the first sequence of digits surrounded by whitespace, Print any string of four consecutive digits*. Given an NL description like the one in Figure 1 together with the source/target, it would be very likely for the system to produce the intended program. This is the intuition behind the model proposed in this paper.

The PbE system

In this section, we describe our baseline PbE system. The model we present here is based on a state-of-the-art PbE system (Gulwani 2011). Like many other program induction systems (Lau et al. 2003), the model benefits from the idea of version space algebra (Mitchell 1982), which divides a problem into independent subproblems and solves each subproblem individually, to avoid dealing with the combinatorial complexity of solving the whole problem at once. However, to be able to make such an independence assumption,

(*s*,*t*) =("I5   5910   9-10 pm" , "5910")



$F_{0,1}(s)$ = {"5", s[1,2], s[3,4], s[pos(s, "[A-Z]+", "[0-9]+", 1), pos(s, "[0-9]+","\s+", 1)], ...}

. . .

$F_{2,4}(s)$ = {"10", s[5,7], s[10,12], s[pos(s, "-", "[0-9]+", 1), pos("[0-9]+", "\s+", 3)], ...}

. . .

Figure 2: DAG of the given (s,t). For brevity, all links between each two nodes are collapsed into one super-link.

some restrictions must be defined on the range of problems allowed. In order to fulfill this and in general to decrease the complexity of the search algorithm, Gulwani (2011) introduces a string mapping language and only allows for programs expressed in this language. Our model uses a subset of this language that deals with regular expression (regex) programming. We call this subset δ_1.

Let s, t be two strings with $s = f(t)$ for some function f. In δ_1, f is defined as one of the following functions:

i. $f(s) = c$, where c is a constant e.g. $f(x) =$"`5910`" for every string x.

ii. $s[p(s) : q(s)]$; a substring of the input from position p to position q. e.g. $f(s) = s[5 : 9]$. p, q are in general functions of the input s (defined below).

iii. $f_1(s).f_2(s)$; that is a concatenation of smaller strings, e.g. $f_1(s) =$"`59`", $f_2(s) = s[10 : 12]$.

The positions p, q can be either a constant integer or a function $pos(s, r_1, r_2, c)$, defined as the cth instance of a position in s, where regular expressions r_1 and r_2 match a substring of s immediately to the left and to the right of that position respectively. For example if the first line of the source/target in Figure 1(a) is given as (s, t), we have $t = s[p, q]$, where $p = pos(s,$"`\s+`","`[0-9]+`"$,1), q = pos(s,$"`[0-9]+`","`\s+`"$,2)$ ("`\s`" refers to *whitespace*).

The concatenation function (type iii) allows us to represent t, in the I/O pair (s, t), as a concatenation of smaller strings. The production of each substring can be done independently. Therefore, we can compactly represent the set $\mathscr{P}$ of all programs $\mathcal{P}$ that map s to t as a directed acyclic graph (DAG) G. Figure 2 represents the DAG corresponding to the above example. In general, there are many links $l^k_{i,j}$ (each associated with a single function $f^k_{i,j}$) between any two nodes i and j. In Figure 2, for readability and brevity purposes, we have used a compact representation of G in which we consider a single link between any two nodes, but associate

it with the *set* $F_{i,j}$ of all functions $f^k_{i,j}$. In the original representation, any path from *start* (node 0) to *end* (node $|t|$) forms a single *program*[4] mapping s to t. In the compact representation, however, for each link $l_{i,j}$ on a path, a function $f^k_{i,j}$ is non-deterministically picked from $F_{i,j}$, in order to produce a single program.

In general, there are infinitely many regular expressions r_1 and r_2 that can identify a position p in s using the *pos* function. In order to limit the possible functions pos, r_1 and r_2 are forced to be a sequence of tokens $T_1T_2 \ldots T_{N_1}$ and $T_{N_1+1}T_{N_1+2} \ldots T_N$ respectively, in which each token T_i is of the form C, $\neg C$, $C+$ or $(\neg C)+$, where C is a standard *character class* such as *Alphanumeric* (`[a-zA-Z0-9]`), *Digit* (`[0-9]`), *Upper* (`[A-Z]`), etc.[5] Let $|C|$ be the number of character classes defined in the system. $|T| = 4|C|$ will then be the number of tokens allowed. If we fix a maximum N_0 on the number of tokens in $r = r_1{\cdot}r_2 = T_1T_2 \ldots T_N$, the maximum number of $substring$ functions identifying a particular subsequence of the input s is $O(|T|^{2N_0})$.[6] Although huge, this number is constant with respect to the size of the I/O pair.[7] On the other hand each subsequence $t[i,j]$ of output (which corresponds to the nodes (i,j) in the DAG) can come from at most $|s|$ substrings of the input. Therefore, the size of DAG is $O(|t|^2|s|)$ or $O(n^3)$ $(n = |s| + |t|)$.

So far, we have shown how to represent the space of possible solutions for a single I/O pair, but the goal of the PbE system is to converge to a unique (or at least a few) solution(s) given multiple examples. In order to achieve this, we intersect the space of solutions for all I/O pairs. Consider DAG $G_1 = (V_1, E_1)$ and DAG $G_2 = (V_2, E_2)$ for the input pair (s_1, t_1) and (s_2, t_2) respectively. We build a new DAG $\mathcal{G} = (V_1 \times V_2, E_1 \times E_2)$ such that for every link $i_1 \rightarrow j_1$ and $i_2 \rightarrow j_2$ in G_1 and G_2, associated with the set of functions ${F^1}_{i_1,j_1}$ and ${F^2}_{i_2,j_2}$, the link $((i_1, i_2) \rightarrow (j_1, j_2))$ in $\mathcal{G}$ is associated with the set ${F^1}_{i_1,j_1} \cap {F^2}_{i_2,j_2}$. The definition extends to the case of m pairs, where the nodes are points in an m-dimensional space. It is easy to see that:

$$|\mathcal{G}| = O(n^{3m}) \tag{1}$$

where $n = \max_{i=1}^{m} |s_i| + |t_i|$.

Building a probabilistic PbE model

In this section, we propose a model which improves over the baseline PbE by incorporating probabilities, hence *probabilistic* PbE or PPbE. The framework we lay out is expanded in the next section, in order to incorporate NL descriptions.

[4]We use the term *program* to refer to a sequence of functions.

[5]In practice, we had to take care of a special but quite frequent case, where a *word*, a *number*, etc is not surrounded by *whitespace* but by the *start-token* (^) and/or the *stop-token* ($). We cover these cases by allowing the first token in r_1 and/or the last token in r_2 to be disjoined with *start/stop-tokens* respectively (i.e. $(\hat{}|T_1)$ and/or $(T_N|\$)$). Gulwani (2011) uses a top-level "switch" function to take care of these cases and more.

[6]$s[p, q]$ includes two positions, hence, the factor 2 in $2N_0$.

[7]Notice here that although *pos* takes an integer c as an argument, given a position p and the regexes r_1 and r_2, there is only a single positive integer c such that $p = pos(s, r_1, r_2, c)$.

The intersection of DAGs for all pairs, as defined in the previous section, is rarely ever a single path/program, but is either empty (i.e. no solution) or contains many solutions. The baseline system uses the *shortest program* heuristic to choose one program over another, where the length of a program depends on *the number of links in the path*, *the type of the function associated with each link*, *the length of the regular expressions*, etc. and is predefined. But we augment links with probabilities. Each link l, associated with a unique function $f(l)$, is assigned a probability $p(l)$ defined in a maximum entropy (MaxEnt) model.

$$p(l) = \frac{1}{Z} \exp\left(-\sum_i w_i \phi_i(l)\right) \tag{2}$$

The features ϕ_i are attributes of the function $f(l)$, such as the type of the function (*constant*, a *substring* with one or two constant integers as positions, a *substring* with one or two *pos* functions, etc.), the length of the regular expressions (if any), the character classes used in the regular expressions, etc.

Given a set of examples $\mathcal{E} = \{(s_1, t_1), \ldots (s_m, t_m)\}$, we are required to find the most likely program:

$$\begin{aligned} \mathcal{P}_{opt} = \arg\max_{\mathcal{P} \in \delta_1} p(\mathcal{P}|\mathcal{E}) &= \arg\max_{\mathcal{P}(s_i)=t_i, i=1:m} p(\mathcal{P}) \\ &= \arg\max_{\mathcal{P} \in \mathcal{G}} p(\mathcal{P}) \end{aligned} \tag{3}$$

where $\mathcal{G}$ is the intersection of all individual DAGs G_i (the space of solutions for the I/O pair (s_i, t_i)) and $\mathcal{P}$ ranges over all the paths from start to end in $\mathcal{G}$. Assuming that the likelihoods of the links are independent, and defining the cost $c(l)$ of each link l as the negative log likelihood, we have:

$$\mathcal{P}_{opt} = \arg\max_{\mathcal{P}=l_1 \cdots l_{|p|}} \prod_{k=1}^{|\mathcal{P}|} p(l_k) = \arg\min_{\mathcal{P}=l_1 \cdots l_{|\mathcal{P}|}} \sum_{k=1}^{|\mathcal{P}|} c(l_k) \tag{4}$$

Hence, $\mathcal{P}_{opt}$ may be found efficiently by Viterbi algorithm:

$$\lambda(v) = \min_{u,j} \{\lambda(u) + c(l^j(u \rightarrow v))\} \tag{5}$$

where u, v are two nodes of the DAG and $l^j(u \rightarrow v)$ ranges over all links between u and v. The complexity of the Viterbi algorithm is equal to $|\mathcal{G}|$, which from Eq. (1) is $O(n^{3m})$.

Our model is reduced to the baseline system, if we *predefine* the cost $c(l)$ of each link based on the length of the associated function $f(l)$. The power of the model, however, is exploited when the costs are learned based on the frequency with which a function has occurred in the data. For example, a longer regex would have a lower cost if it occurs more frequently in the data. Therefore, using some training data, the model defines a distribution over the programs, in order to make more accurate predictions. We find that our MaxEnt model outperforms the baseline system significantly, even without incorporating NL descriptions.

Incorporating natural language

Incorporating NL descriptions into MaxEnt is straightforward. We simply condition all the probabilities in Eqs. (2)

through (4) on the NL description $\mathcal{D}$. In particular, for link probabilities, we have:

$$p(l \mid \mathcal{D}) = \frac{1}{Z} \exp\left(-\sum_i w_i \phi_i(l, \mathcal{D})\right) \quad (6)$$

We refer to this model as NLPbE.

Feature extraction

In order to extract meaningful features from natural language descriptions, we apply a dependency parser to each description. We use dependency parsing because it not only encodes the syntactic structure of a sentence, but it also includes some semantic information by encoding the predicate-argument structure of the sentence in the dependency links. For example, for this domain it is critical to recognize the main predicate of the sentence because it specifies the main operation to be done, as in *switch the first word with the last word*. Given a description $\mathcal{D}$ and a dependency tree T, the root of T would most likely be the main predicate, and its children the arguments of the predicate. Similarly, it is easy to locate the modifiers of a noun by looking at its children in T. Given a noun phrase like *a number followed by a comma*, this information can help the model to boost the programs containing regular expressions such as `"[0-9]+,"` or `",[0-9]+"`. Following this intuition, the features we extract from $\mathcal{D}$, are the set of all its words (as lexical items) plus their locations in the dependency tree, represented by the depth of the corresponding node and the lexical word of their children and their parent.

Train and test procedures

Training

Adopting a discriminative model, we use a supervised method for training. To provide annotated data, each text editing task needs to be labeled with a program expressed in the language δ_1. This is very labor-intensive if done by hand. Therefore we use a trick to get around the hand annotation. Thanks to the data from Manshadi, Keenan, and Allen (2012), each task in the corpus comes with a description $\mathcal{D}$ and a set $\mathcal{E}$ of 4 I/O pairs: $\mathcal{E} = \{(s_1, t_1), \ldots, (s_4, t_4))\}$. We run the baseline PbE engine on these I/O pairs to label each task in the training set. [8] For each I/O pair (s_i, t_i), we build the DAG G_i, the space of all programs that map s_i to t_i. We then take the intersection of G_is to build the DAG $\mathcal{G}$ which maps all the inputs to their corresponding outputs. Many of the links in G_is are filtered out when taking the intersection, and hence do not appear in $\mathcal{G}$. We label those links with a minus, forming the class $\mathcal{C}^-$ of all incorrect links. Conversely, every link l in $\mathcal{G}$ is labeled with a plus, forming $\mathcal{C}^+$, the class of correct links.

Once all the tasks in the training set make their contribution to $\mathcal{C}^+$ and $\mathcal{C}^-$, a MaxEnt classifier is trained on the points in the two classes, in order to learn the weight vector $W = (w_1, w_2, ...)$ used in Eq. (6).

Evaluation

As mentioned above, each individual task in the corpus is represented with a description $\mathcal{D}$ and a set $\mathcal{E}$ of m examples. In general, for each task in the *test set*, $\mathcal{E}$ is split into two subsets, the set of *inducing examples* $\mathcal{E}^{\mathcal{I}} = \{(s_1, t_1), \ldots, (s_d, t_d))\}$, used by the trained model for program induction, and the set of *verifying examples* $\mathcal{E}^{\mathcal{V}} = \{(s_{d+1}, t_{d+1}), \ldots, (s_m, t_m))\}$ used to verify whether the induced program works on these examples.

The program induction works as follows. A DAG G_i is built for each pair $(s_i, t_i) \in \mathcal{E}^{\mathcal{I}}$ as described before, then all G_is are intersected to build the DAG $\mathcal{G}$. The baseline PbE system uses the shortest program heuristic to choose a program $\mathcal{P}$. The probabilistic PbE (PPbE), and the NL-based model (NLPbE), use the Viterbi algorithm in Eq. (5) to induce $\mathcal{P}$, in which $c(l)$ is the negative log likelihood, and $p(l)$ is calculated as in Eqs. (2) and (6) respectively.

An issue with the evaluation of a program induction system is that, in general, there is no such a thing as the *correct* program, and even if there is, there is no way to decide whether the induced program is correct. [9] As a result to evaluate the models, we run $\mathcal{P}$ on all examples in $\mathcal{E}^{\mathcal{V}}$ and label $\mathcal{P}$ as *correct* iff for *every* (s_j, t_j) in $\mathcal{E}^{\mathcal{V}}$, $\mathcal{P}(s_j) = t_j$. The accuracy of the model is then defined as:

$$A_t = N_t^{\mathcal{C}} / N_t \quad (7)$$

where N_t is the number of tasks in the test set and $N_t^{\mathcal{C}}$ is the number of those tasks whose induced program $\mathcal{P}$ is labeled correct. We call this the *task-based* accuracy. An issue with the task-based metric is that since there is no way to decide whether a program is correct or not, it is not fair to assign a binary label (i.e. correct/incorrect) to the predicted programs. This is in particular important for Manshadi et al.'s corpus, where different examples of a task are provided by different workers on mechanical turk, who may not necessarily agree on what the intended program is. Besides, the task-based metric is sensitive to the number of verifying examples. The higher the number is, the lower the accuracy would be. We propose an alternative method where the performance is evaluated based on the number of examples $(s_j, t_j) \in \mathcal{E}^{\mathcal{V}}$ that the induced program $\mathcal{P}$ solves correctly, that is, $\mathcal{P}(s_j) = t_j$. The *example-based* accuracy A_e is, therefore, defined as:

$$A_e = N_e^{\mathcal{C}} / N_e \quad (8)$$

where N_e is the total number of verifying examples in the test set[10] and $N_e^{\mathcal{C}}$ is the number of those examples solved correctly by the induced program. $A_t = A_e$, if $|\mathcal{E}^{\mathcal{V}}| = 1$.

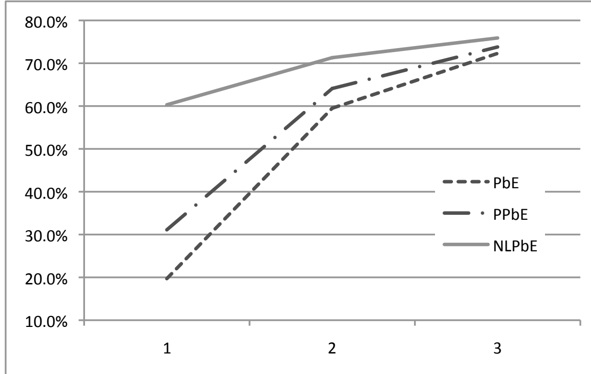
[8] In practice, we manually guide the PbE system towards finding the correct program. While labeling each task with an exact program is a painful job, it is not quite as hard to provide guidelines, such as a subset of character classes that are likely to be part of the correct program. Incorporating these guidelines in the PbE engine has two advantages. First, it helps the PbE engine to converge to a solution in a shorter time. Second, the induced programs will be closer to the the gold-standard programs as a result of the gold-standrad guidelines.

[9] This is because in general, the equivalence of two programs is an undecidable problem. Even in the special case of finite state machines, where the equivalence is decidable, it is computationally very expensive to decide the equivalence.

[10] $N_e = N_t |\mathcal{E}^{\mathcal{V}}|$, if $|\mathcal{E}^{\mathcal{V}}|$ is the same for all tasks in the test set.

Model	A_t	A_e
PbE	10.3%	19.7%
PPbE	18.6%	31.1%
NLPbE	**42.2%**	**60.3%**

(a) Task/example-based accuracy for $|\mathcal{E}^{\mathcal{V}}| = 3$



(b) The accuracy of models vs. number of inducing examples

Figure 3: Experiments.

Experiments

We borrowed the data from Manshadi, Keenan, and Allen (2012). Their corpus consists of 773 text editing tasks, out of which 338 tasks can be expressed in the language δ_1. Each task comes with a description and a set of 4 examples.

We split the corpus into three sets at random. 58 tasks were used for development, 210 for training, and the remaining 70 for evaluation. Remember that the model was intended to take a description of a task in natural language and a single example of the input/output and to predict the intended program. As mentioned before, this is how most people choose to describe their tasks to human experts. Hence, we believe it would be a natural, yet practical, interface when dealing with machines as well. Our first experiment evaluates the model in this configuration. For comparison purposes we have also evaluated the baseline PbE systemand our probabilistic PbE (PPbE) in the same configuration. Since every task comes with 4 examples and we use one for program induction, the rest are used for evaluation ($|\mathcal{E}^{\mathcal{I}}| = 1, |\mathcal{E}^{\mathcal{V}}| = 3$). The results are listed in Figure 3(a).

In the second experiment, we are looking to see how the contribution of NL descriptions varies as the number of inducing examples (i.e. $|\mathcal{E}^{\mathcal{I}}|$) increases. Figure 3(b) shows the performance of all three models with the number of inducing examples ranging from 1 to 3 (hence 3*3=9 evaluations). In order to to have a fair comparison, for each task k in the test set, the same $\mathcal{E}_k^{\mathcal{V}}$ ($|\mathcal{E}_k^{\mathcal{V}}| = 1$) is used for all 9 cases.

Discussion

As seen in Figure 3(a), with a single example, the baseline system does a poor job on program induction, as with only one input/output there are many possible programs that can map the input to the output and the shortest program heuristic does not seem to be a good predictor. Leveraging the frequency of the functions and their arguments (in particular, regular expressions and their character classes), extracted from the training data, PPbE model outperforms the baseline by 11%. Incorporating NL descriptions results in yet another significant improvement and more than triples the accuracy. Given that we do not perform any deep semantic analysis on the NL descriptions, the results are very promising, encouraging us to further explore the merits of this integration.

As expected, by increasing the number of inducing I/O pairs, the overall accuracy of all models improves[11], but the leverage of the natural language descriptions in the performance drops (Figure 3(b)). The leverage drops significantly when we go from $|\mathcal{E}^{\mathcal{I}}| = 1$ to $|\mathcal{E}^{\mathcal{I}}| = 2$. This is because the second example eliminates many possibilities the first one invokes.For example, given only a single I/O pair (s, t), a particular occurrence of *newline* in the output, say $t[4:5]$, may be a copy of any occurrence of newline in the input, say $s[5:6], s[8:9]$, etc., or it may be no substring of input but simply a constant character `"\n"` inserted by the program. In the former case, there are in general exponentially many regular expressions that can generate that particular substring of input. When the second input (s', t') is given, many of those will be ruled out, when selecting a substring of s' not equal to `"\n"`. In fact, from Figure 3(b), the second example provides almost as much information to the PbE system as the features extracted from the NL description. With a deeper model of NL descriptions, however, we believe this will no longer be the case.

One point to remember here is that increasing the number of inducing examples is very expensive as the complexity of the induction is exponential in the number of examples, whereas leveraging NL descriptions comes almost for free (conplexity-wise). Given that time complexity is the main bottleneck of PbE systems, leading to those systems being barely practical, this is a huge advantage that NLPbE offers.

What does the model learn?

Nouns that are the arguments of the main verb (often nodes at depth 1 in the dependency tree), provide invaluable hints. For example, the model learns that there is a high correlation between the word *number* and the token `"[0-9]+"`. Pre-noun modifiers such as ordinal numbers (e.g. *first line, second number,* etc.) will boost functions with certain arguments (e.g. the ordinal number *second* boosts a function $pos()$ with number 2 in its last argument position). The main verb of the sentence is also quite informative. For example, the action *print* and *delete* treat their arguments in the opposite way (e.g. while *print* likes a regex in the first position of a *substring* function, *delete* prefers the other way around).

The model is able to learn more indirect information as well. Consider the above example on finding a function generating a particular occurrence of newline in the output.

[11]It seems that the overall accuracy converges to 76% and no longer increases by much by increasing the number of inducing examples. This is because in this corpus, the examples of every task are provided by different workers (given the NL description) who may not have the same interpretation of the NL description, and hence may provide inconsistent examples.

When a task asks for listing some elements of the input text, often those elements are separated by some delimiter such as newline. Therefore, the NLPbE model learns that if the main verb of the NL description is *List, Extract, Pull out,* etc. (as in *List all words staring with a capital letter*), a newline in the output is very likely to be a constant string inserted by the program, as opposed to a substring of input. The baseline system, on the other hand, almost always prefers a *substring* function generating newline, because it assigns a lower cost to *substring* than to a constant string.

Related work

We use natural language features to guide the induction of a program from an input/output example, and thus make use of previous work on program induction. We adopt the approach of Gulwani (2011) as our starting point, itself based on the version space approach of Lau et al. (2003). An interesting competing approach is that of Liang, Jordan, and Klein (2010), which uses a hierarchical Bayesian prior over programs to encourage the re-use of routines inside programs for different tasks. Using this type of program induction in combination with natural language could have the benefit of allowing more complex subroutines to be identified at cost of more complex inference. Menon et al. (2013) develop a programing by example system based on a probabilistic context-free grammar and a set of manually provided clues. They use the grammar to generate programs in a descending order of probabilities, producing the first program that solves all the I/O pairs. The model is solely used to speed up the search process and is claimed to perform the induction significantly faster without sacrificing the accuracy.

Manshadi, Keenan, and Allen (2012) suggest using the crowd to do natural language programming. They offer to obtain input/output pairs from the crowd by providing them with the natural language descriptions, and then use programing by example to induce a program. We have borrowed their data including the examples provided by the crowd, but instead of using standard programing by example, we have used those examples to train a model that incorporates natural language descriptions into a programing by example system to make faster, more accurate predictions.

PLOW (Allen et al. 2007) learns programs that can run in a web browser from a user demonstration with verbal description of each step. Using deep understanding of the user commands and heuristic learning on the observed actions in the web browser, they achieve one-shot learning of simple tasks. While the domain of web browsing is more complex than ours, the programs learned are generally simpler. Their technique requires that the entire task be demonstrated, as they learn from execution instances (programing by demonstration) rather than just the input/output states as used in our system (programing by example).

Recent years have seen a great deal of interest in automatic learning of natural language semantics, with a number of new systems automatically learning correspondences between sentences and meaning representations such as database queries (Zettlemoyer and Collins 2005; 2009; Clarke et al. 2010; Liang, Jordan, and Klein 2011) or plans for navigation (Vogel and Jurafsky 2010; Chen and Mooney 2011). The training phase of our system is similar, in that it automatically learns the correspondences from examples, and does not require any initial dictionary. Furthermore, we do not make use of any gold-standard, human-annotated programs, but rather train the natural language features using programs induced from examples. This bears some similarity to the way in which Clarke et al. (2010) and Liang, Jordan, and Klein (2011) train a system mapping language to database queries by using the results of database queries, without access to gold-standard queries for the training sentences. Similarly, Chen and Mooney (2011) learn correspondences from language to navigation plans using observed sequences of navigation actions, without access to gold-standard navigation plans. While we use similar methods in training our natural language semantic component, our overall approach to the problem of human-computer interaction is quite different, in that we make use of one input/output example at test time, as way of combining natural language programming with programming by example.

Summary and future work

We presented a hybrid model which uses natural language description of text-editing tasks with one (or more) example(s) of the input/output to find the intended program. Incorporating natural language descriptions results in 40% improvement over a plain PbE system when both systems are given a single example of the input/output. We achieve this performance despite the small size of the training set and despite our simple model of natural language.

To the best of our knowledge this is the first work on the integration of programming by example and natural language programming. The framework presented here lays out the ground work and establishes a baseline for future efforts in this direction. Early results are encouraging, and we hope, motivate a new line of research on automatic programming.

This work can be improved in several directions. First, increasing the size of the corpus should improve the performance as it produces more reliable statistics. Alternatively, since even automatic labeling of the tasks is time consuming, leveraging unlabeled data to do semi-supervised learning can help to overcome the limited availability of labeled data. A more complex model of natural language, a deep semantic analysis in particular, is also likely to have a significant impact on the performance. The most important improvement, however, would be to relax the independence assumption on the sequence of operations in a program. By allowing to define the probability of each link solely based on the its own associated function, the independence assumption remarkably simplifies the model. One way to avoid this issue is to frame it as a structured learning problem.

The true power of such a hybrid model is exploited when increasing the expressiveness. Standard PbE systems suffer from limited coverage. The restrictions on the range of problems allowed cannot be relaxed, because otherwise the space of possible solutions explodes. Incorporating natural language can help to drastically narrow down the search space, and hence allow for expanding the coverage.

References

Allen, J.; Chambers, N.; Ferguson, G.; Galescu, L.; Jung, H.; Swift, M.; and Taysom, W. 2007. Plow: a collaborative task learning agent. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 2*, AAAI'07, 1514–1519. AAAI Press.

Chen, D. L., and Mooney, R. J. 2011. Learning to interpret natural language navigation instructions from observations. In *AAAI*.

Clarke, J.; Goldwasser, D.; Chang, M.-W.; and Roth, D. 2010. Driving semantic parsing from the world's response. In *Proceedings of the Fourteenth Conference on Computational Natural Language Learning (CoNLL-2010)*, 18–27.

Dijkstra, E. W. 1979. On the foolishness of "natural language programming". In *Program Construction, International Summer School*, 51–53. London, UK: Springer-Verlag.

Gulwani, S. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, 317–330. ACM.

Lau, T.; Wolfman, S. A.; Domingos, P.; and Weld, D. S. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53:111–156.

Liang, P.; Jordan, M. I.; and Klein, D. 2010. Learning programs: A hierarchical bayesian approach. In *ICML*, 639–646.

Liang, P.; Jordan, M. I.; and Klein, D. 2011. Learning dependency-based compositional semantics. In *ACL*, 590–599.

Manshadi, M.; Keenan, C.; and Allen, J. 2012. Using the crowd to do natural language programming. In *Proceeding of the Twenty-Sixth AAAI Conference on Artificial Intelligence, Workshop on Human-Computer Interaction (HCOMP-2012)*. AAAI.

Menon, A. K.; Tamuz, O.; Gulwani, S.; Lampson, B.; Jung, H.; and Kalai, A. T. 2013. A machine learning framework for programming by example. In *the Proceedings of the 30th International Conference on Machine Learning*, TO APPEAR.

Mitchell, T. M. 1982. Generalization as search. *Artificial Intelligence* 18.

Vogel, A., and Jurafsky, D. 2010. Learning to follow navigational directions. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, 806–814.

Zettlemoyer, L. S., and Collins, M. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI*, 658–666.

Zettlemoyer, L. S., and Collins, M. 2009. Learning context-dependent mappings from sentences to logical form. In *ACL/AFNLP*, 976–984.