DeTEXT: Programming by Example

Konstantin Weitz Jonathan Bragg Md Tanvir Islam Aumi

December 14, 2012

{weitzkon}, {jbragg}, {tanvir} @ cs.washington.edu

1 Introduction

Data cleaning and manipulation is an expensive and tedious process that consumes valuable time and resources. If an end-user needs to manipulate a large amount of data, he has three primary options: record a macro, write a transformation program, or perform manual manipulations on the data. Recording a macro with a fixed series of actions can be simple, but does not generalize well and is not sufficiently expressive for many tasks. Writing a transformation program requires expert knowledge unavailable to many end-users. And depending on the quantity of data, it may not even be possible to perform manual manipulations.

Programming by demonstration (*PbD*) offers an intuitive alternative to these approaches. In a PbD setting, an end-user demonstrates the actions required to transform input examples into output examples. A machine learning algorithm observes these transformation sequences, called traces, with the goal of inducing a program that transforms all future examples in the desired manner. A line of successful research on PbD uses a formalism called version space algebra [7] to represent the space of all hypotheses (programs) that are consistent with the input-output examples. The version space algebra defines useful operators like union and join that enable construction of complex version spaces from simple ones. SmartEDIT [5] is a PbD system for learning text transformations.

However, it is not always possible to observe traces. Observing traces requires tight integration with text editors, which come in many forms and often do not expose their source code for PbD integration. Furthermore, input-output examples may have been pre-recorded, as in a partially completed spreadsheet. Spreadsheet users, who often need to clean or manipulate data, number over 500 million people worldwide; there is a huge opportunity for time savings and increased productivity.

The problem of inductively learning programs without traces is called programming by example (*PbE*). PbE is a strictly harder problem than PbD; the learning algorithm observes less information and thus must consider a much larger space of possible hypotheses. Our research has focused on lifting the PbD methods used by SmartEDIT to the PbE context. This work was conducted independently of [1], which also aims to solve this problem.

In this work, we present DeTEXT, a PbE system that is able to save users time in a number of repetitive text-editing scenarios. While the rich language of text transformations we originally envisioned is beyond the scope of this project, DeTEXT learns simple editing operations from three or fewer examples. We make the following contributions:

- A novel algorithm for constructing version spaces for text transformations without a program trace;

- A PbE system for text-editing with fast convergence;
- An interactive web interface; and
- Experimental validation of our approach, including an informal user study.

2 Problem statement

We want to allow a user to speedup a repetitive task by learning the user's intentions from examples without traces (PbE).

The user's task is repetitive. We can imagine a loop, where in each iteration the same actions are performed on different data. We call the sequential execution of these actions a program.

To limit the number of hypothesis, each iteration manipulates a different line of the input. To provide the algorithm with examples, the user must therefore edit one line at a time.

One could try to learn a program as powerful as a Turing Machine. This task is not tractable. For example, if the user encrypted each line with AES, learning a Turing Complete Program would be equivalently hard to executing a known-plaintext attack on AES.

To limit our algorithm's execution time, we decided to restrict the space of programs to those that only execute insertions on the input line. We assumed that most of the time, users will not try to insert characters at a fixed location, but try to insert characters either before or after a certain string. Thus, our programs consist of a sequence of actions, where each action finds an insertion position using either a prefix or a suffix and then inserts a constant string.

This model allows us to execute actions such as escaping the first HTML tag in every line of a HTML file. [1] It does not however, allow the escaping of multiple tags in a line or tags that span a line.

To allow the modification of multiple tags per line, each program is executed repeatedly on each line. Listing 1 describes how a program is executed for each line. To ensure termination, each action consumes all characters up to its insertion position. If the insert does not consume any characters, one character is consumed in certain cases, for simplicity, this is not shown in the pseudo code.

Listing 1: Program Execution

```
def next_action(program):
    if empty(program.actions):
        reset(program.actions)
    return next(program.actions)

def run(program, src):
    a = next_action(program)
    i = search_insert_location(src,a)
    if i is invalid:
        return src
    return src[:i] + insertion_string(a) + run(program, src[i:])
```

Both SmartEDIT and [1] can learn more powerful programs. SmartEDIT does also not operate on a per line basis.

[1]replace < with $<$ and > with $>$.

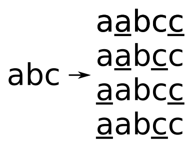
3 The DeTEXT PBE system

3.1 Inference engine design

The algorithm for our inference engine has two primary components. First, it generates a set of all possible trace expressions, given our simplifying assumption that the user has made only insertions. A trace expression consists of the locations at which characters have been inserted into a string. Second, the algorithm uses the set of traces to construct a tree of version spaces using union and join operations. This tree is updated each time a new example is provided.

In order to illustrate the first part of the algorithm, consider the problem of finding all possible insertions that transform the input string *abc* to the output string *aabcc*. Although tools like *d*iff provide a unique solution for transformations made on a string, the answer to the question of which insertions were made is ambiguous. Figure 1 shows the set of possible insertion locations. Note that the size of the set of traces increases with the amount of repetition in the input and output strings.

Figure 1: Finding Insertions Example



The algorithm given in Listing 2 enumerates all m combinations of insertion positions by recursively binding the first character of the input string to a matching location in the output string. Each yielded result corresponds to one *aabcc* line on the right side of Figure 1.

Listing 2: Finding Insertion Positions

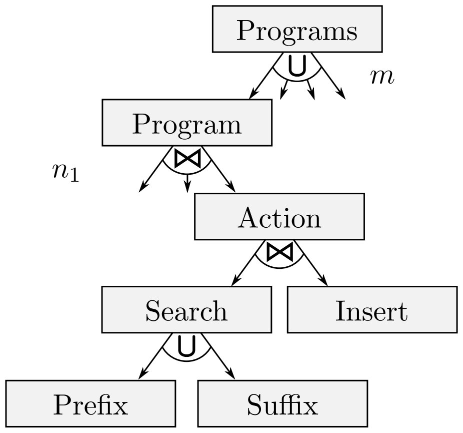
```
def insertPositions(src,dst):
    if empty(src):
        yield unbound character positions
    else:
        c = pop_first_char(src)
        for i in find_all(c,dst):
            insertPositions(src,dst[i+1:])
```

In the second part of our algorithm, these results are combined to form the tree shown in Figure 2, where each node represents a version space. Each of the m results from the first part of the algorithm forms a **Program** version space p, which consists of a sequence of n_p **Action** version spaces that transform the source string into the target string [2]. Each **Action** version space, in turn, consists of two version spaces that sequentially search for an insertion location (the **Search** version space) and insert a string at that location (the **Insert** version space). We have implemented two ways of searching for an insertion locations, either by matching after a string prefix (the **Prefix** version space) or before a string suffix (the **Suffix** version space). The union version space operator allows for simple extension to other types of matching.

In order to support examples with more than one repetitive task (iteration) per line, we substitute a version space union of **SubPrograms** version spaces for each **Program** version space in the

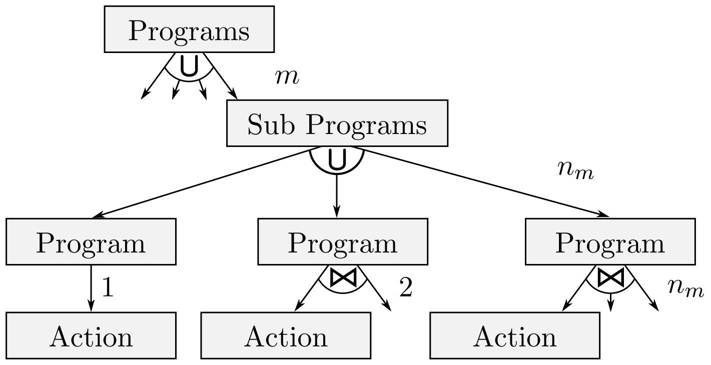
[2]Different programs may have a different number of actions

Figure 2: The DeTEXT Version Space



version space tree described above. Figure 3 shows the **SubPrograms** version space substitution. [3] Each **Program** version space in the **SubPrograms** version space consists of a prefix of the original program's actions. Consider the problem of learning a program for escaping an HTML tag from an example showing multiple HTML tags being escaped. Without this the substitution described above, we would not be able to generalize to transforming examples with a single HTML tag.

Figure 3: The DeTEXT Version Space with **SubPrograms** Substitutions



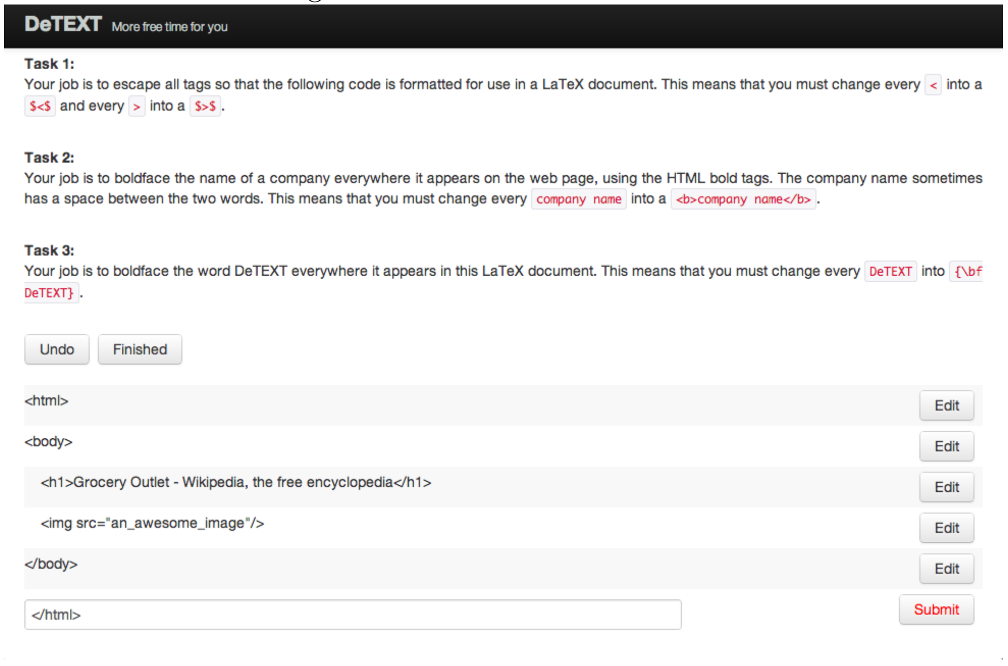
3.2 Implementation

DeTEXT is implemented in python, using the web2py web framework. Our system has three modules: a sandbox module, a predictive module, and a unaided module. The sandbox, shown in Figure 4, allows users to familiarize themselves with the system. DeTEXT loads a text file and generates a web form for each line in the file. Figure 4 shows a user editing a line.

The predictive module runs the DeTEXT inference engine. Figure 5 shows a user completing Task 3, which involves bolding the word DeTEXT in a LaTeX document. The user has edited lines four and five (the Edit buttons are no longer visible), and DeTEXT has applied the inferred hypothesis to unedited lines in the document. Proposed changes are highlighted. If the proposed changes are correct, the user can press the Finished button. Otherwise, the user can continue to

[3]The **SubPrograms** version space serves a function similar to SmartEDIT's **UnsegmentedProgram** version space.

Figure 4: DeTEXT Sandbox Mode



Task 1:
Your job is to escape all tags so that the following code is formatted for use in a LaTeX document. This means that you must change every `<` into a `$<$` and every `>` into a `$>$`.

Task 2:
Your job is to boldface the name of a company everywhere it appears on the web page, using the HTML bold tags. The company name sometimes has a space between the two words. This means that you must change every `company name` into a `<b>company name</b>`.

Task 3:
Your job is to boldface the word DeTEXT everywhere it appears in this LaTeX document. This means that you must change every `DeTEXT` into `{\bf DeTEXT}`.

```
<html>
<body>
  <h1>Grocery Outlet - Wikipedia, the free encyclopedia</h1>
  <img src="an_awesome_image"/>
</body>
</html>
```

edit lines and DeTEXT will update the most likely hypothesis and predictions. If the user makes a mistake, the Undo button reverts the last action. All actions are recorded in a database for the purposes of a user study.

4 Experimental results

We evaluate the DeTEXT system using two methods.

First, we investigate performance on a collection of repetitive text-editing scenarios. We evaluate how quickly our system can learn transformation programs on real tasks, and compare performance to SmartEDIT. We also examine the size of the version space trees generated by our approach in an attempt to assess issues of scalability.

Second, we conduct a small, informal user study. We hope to understand whether the DeTEXT inference engine speeds up the repetitive editing process. We also look at the usability and helpfulness of our system.

4.1 Empirical evaluation

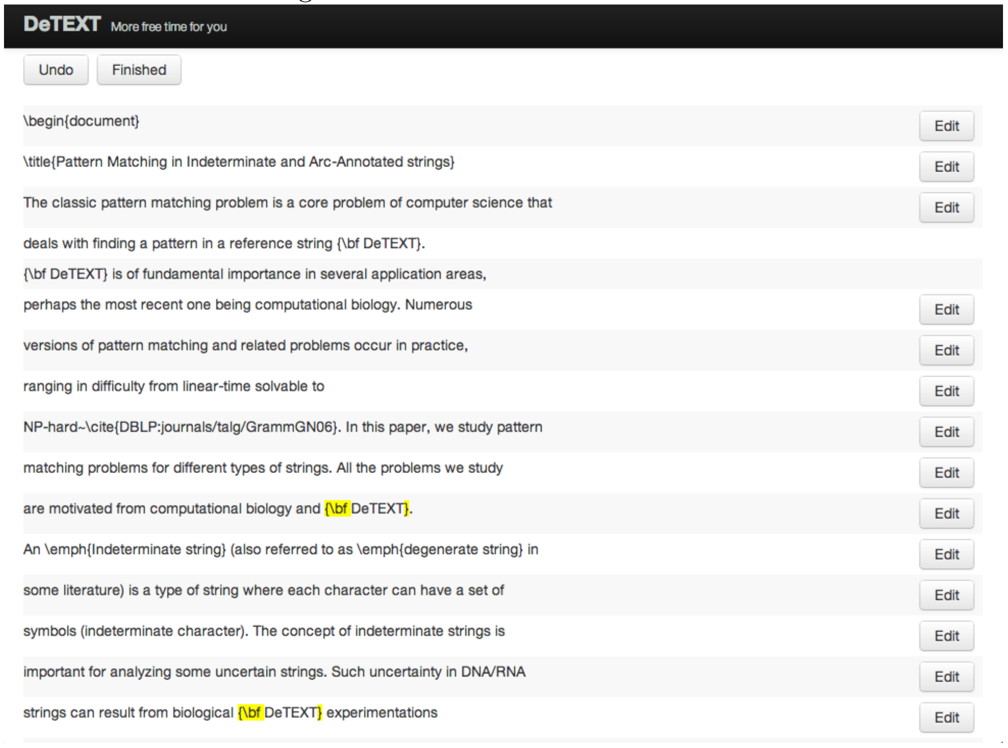
In order to assess the empirical performance of our system, we designed three repetitive text-editing scenarios inspired by the following scenarios used to test SmartEDIT in [5]:

html-latex Convert HTML to LaTex by escaping less-than and greater-than characters into LaTeX's math mode

boldface-xyz Boldface the name of a company everywhere it appears on the web page, using the HTML bold tags. The company name sometimes has a space between the two words in the name.

boldface-word Boldface the word DeTEXT in a LaTeX document.

Figure 5: DeTEXT Predictive Mode



```
\begin{document}
\title{Pattern Matching in Indeterminate and Arc-Annotated strings}
The classic pattern matching problem is a core problem of computer science that
deals with finding a pattern in a reference string {\bf DeTEXT}.
{\bf DeTEXT} is of fundamental importance in several application areas,
perhaps the most recent one being computational biology. Numerous
versions of pattern matching and related problems occur in practice,
ranging in difficulty from linear-time solvable to
NP-hard~\cite{DBLP:journals/talg/GrammGN06}. In this paper, we study pattern
matching problems for different types of strings. All the problems we study
are motivated from computational biology and {\bf DeTEXT}.
An \emph{Indeterminate string} (also referred to as \emph{degenerate string} in
some literature) is a type of string where each character can have a set of
symbols (indeterminate character). The concept of indeterminate strings is
important for analyzing some uncertain strings. Such uncertainty in DNA/RNA
strings can result from biological {\bf DeTEXT} experimentations
```

Table 1 shows the performance of DeTEXT on these scenarios, along with a comparison to SmartEDIT. The total number of iterations, or instances of the edit described in the task, is listed in the right-most column. The number of training iterations performed before DeTEXT and SmartEDIT learned the correct editing program appears in the two columns to the left. Training halts when edits have been predicted correctly on all remaining iterations.

In order to generate this performance data, the authors used DeTEXT on the input data and edited each line in turn starting from the top of the file. Note that our system supports editing lines in any order, unlike SmartEDIT, which requires that users work in order from the beginning of a text file. We list SmartEDIT performance numbers from [5], which used a separate dataset, but we have constructed our data set to match the tasks described in [5] as closely as possible, including having the same total number of iterations.

Scenarios	DeTEXT Iterations	SmartEDIT Iterations	Total Iterations
html-latex	2	3	7
boldface-xyz (HTML)	3	4	50
boldface-word (LaTeX)	2	6	11

Table 1: Performance on Scenarios

In order to understand why our system required multiple iterations to learn a program, consider the html-latex scenario. While one could imagine a more sophisticated algorithm using a probabilistic weighting of hypotheses, our system ranks hypotheses in order of decreasing length of matching strings. Thus, our system first matches on the string “html¿”, for example, before it learns to match only on “¿”. Note that like SmartEDIT, which uses a probabilistic weighting that decreases exponentially in the number of actions, our system selects programs with fewer actions before other programs.

We believe that the reason our system learns more quickly than SmartEDIT on these tasks, even in the absence of trace information, is due to the fact that we use a much smaller transformation language.

With each scenario, the number of hypothesis decreases quickly as more examples are given, while the size of the version space tree stays nearly the same. These results are summarized in Table 2. The maximum number of nodes in the version space, the maximum number of program nodes in the version space, as well as the number of hypotheses for a given number of examples are provided in this table. Execution speed on a normal computer was never a problem and proposed hypothesis were generated without noticeable delay. In contrast to SmartEDIT, our version space is not linear in the number of actions and thus is unlikely to scale well with more complex sequences of actions.

Scenarios	Nodes	Programs	# of Hypotheses		
			1	2	3
html-latex	35	4	1554	44	-
boldface-xyz (HTML)	23	4	8320	496	240
boldface-word (LateX)	23	4	6844	42	-

Table 2: Scenario Hypothesis Analysis

4.2 User Evaluation

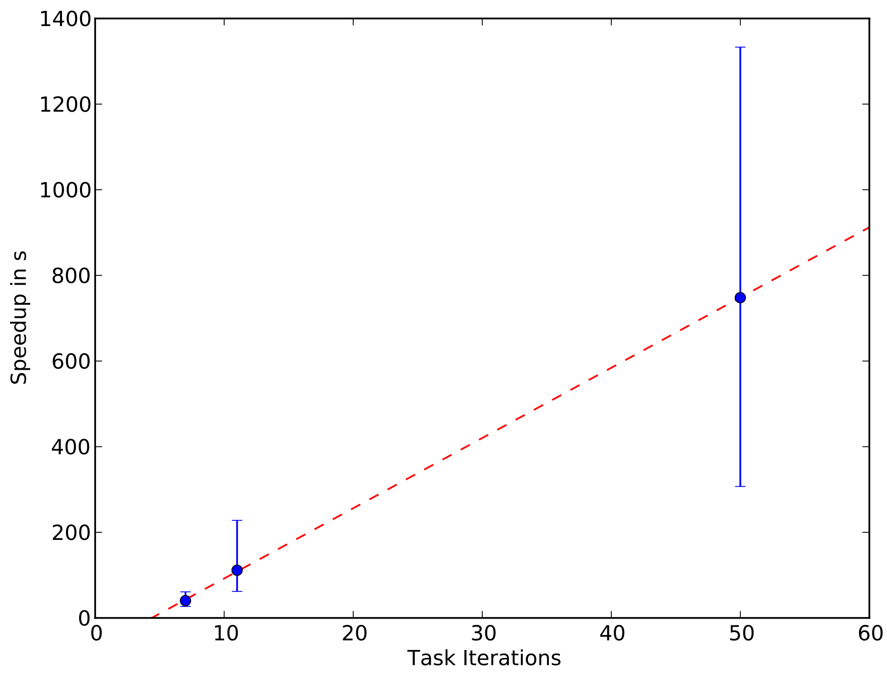
We conducted a pilot user study to gather user experiences with the DeTEXT system. We tested our system on five people. Among those were two experienced DeTEXT users (authors), one computer scientist, and two users without a computer science background. Before the study begun, the user could get familiarized with the application in a sandbox mode. The actual study was composed of three scenarios. For each of the scenario, the user was asked to perform a task with and without the help of the DeTEXT system. We measured the performance of our system by calculating the time savings gained by the use of DeTEXT.

Figure 6 shows the average difference in seconds between the time taken to complete the task manually, and the time to complete it using DeTEXT. Thus, bars above the zero line indicate that an user completed the task more quickly with DeTEXT. The error bars indicate both the minimum and maximum speedup gained. The time saved by using DeTEXT on a task depends on the number of iterations in the task, as expected the savings increase with the number of iterations, as seen with the red line which represents the linear regression minimizing the mean square error of the data points. Interestingly, users with a computer science background were as fast at finishing the tasks as the authors, while users without a computer science background took longer to finish task both with the support of DeTEXT and without.

We also obtained user feedback in the form of responses to a questionnaire, summarized in Table 3. The feedback was quite positive. We asked users to rate (on a scale of 1 to 5, 5 being the best) DeTEXT's helpfulness in accomplishing the tasks, DeTEXT's usability, and whether they would use the system again. As the table shows, participants found DeTEXT helpful and would use it again. Further, we got very helpful written feedback on our approach. Comments from participants included:

> "Great tool! I would be happy to use it for my everyday coding needs."

Figure 6: User Study Results



"Checking the suggested modification was faster than checking my manual modifications."(by author)

"I could have easily solved every task using Find/Replace in any standard editor."(by author)

"It keeps taking you up to the top of the page again after each time that you edit anything."

"The server crashed several times."

Questions	Average Response
Helpful	5
Useable	4
Use Again	5

Table 3: User Feedback

5 Discussion

We initially planned to allow more powerful programs. Unfortunately, finding such programs turned out to be computationally expensive with the approaches we tried. Given the limited time frame of this project, we decided to restrict a program's power to make it tractable.

One of the mayor things we learned from the user study were ideas on how to improve our user interface. Many of the reported problems were a result of the decision to do all computation on the server side. Problems with slow page reloads and scrolling could have been avoided had the application been developed with AJAX.

Running the user study was harder than expected. On the one hand, some of the task description were not as clear as we hoped. For example, some users did not understand what escaping HTML meant. On the other hand, some users faced server crashes that did not occur on our local machines. We learned that one has to basically run a supervised user study to create a good automated user study. We invested a lot of time in creating a fully automated user study. Given our small samples size, our time may have been better spent in supervising user studies instead of trying to automate every single part of it.

We were also not aware how quickly users lost the motivation to complete a task. Especially our second scenario, where users were asked to execute 50 iterations, was to mind numbing for many users.

6 Related work

[7] introduced version-space algebra for learning boolean functions, and [6] extended version-space methods to programming by demonstration with text transformations (the SmartEDIT system). Further work using programming by demonstration has been done on learning imperative Python programs [4] and shell scripts [3]. [5] describes how to learn text transformation programs from repeated demonstrations without segmentation information.

[1] develops methods based in part on version spaces for learning text transformation programs using programming by example, a strictly harder problem that we also consider in this work. [1] uses an expressive transformation language that enables complex conditional expressions and inner loops and handles noisy examples, which is beyond the scope of our project based on techniques from [5]. [8] extends methods in [1] to handle semantic transformations in addition to syntactic ones.

A complementary area of research has focused on mixed-initiative interfaces for guiding a user to the desired transform, or hypothesis. [9] extends SmartEDIT with a decision-theoretic agent that selects an interaction from a set of interactions that vary the examples used for learning and the amount of control given to the system and to the user. More recently, [2] supports a wide range of data cleaning tasks and interactions, making use of semantic information in the data and suggesting sets of potential hypotheses to users using natural language descriptions.

7 Conclusions and future work

We have shown that, with certain restrictions on its capabilities, a PbD approach can be successfully extended to work without trace information, and thus solve PbE problems. Further, we conducted a user study that showed that such an approach is actually helpful to users.

In the future, we hope to reduce the number of restriction on a program. This may include simple changes such as adding the ability to insert line numbers, or insertions at a specific location in a string. We were also thinking about ways to allow deletions (without insertions), or even deletions and insertion together.

We expect that our state space would grow dramatically, especially if we allowed deletions and insertions together, and may thus need a strong bias. One way we could introduce such a bias is by specifying a probability distribution over all hypothesis, as suggested in [6]. We could further use such a feature to allow noisy edits, by assigning inconsistent hypothesis a low but unequal zero probability.

As we observed the participants during our user study, we saw that they often processed the source data from top to bottom. This is not always the most efficient method. We could extend our system, in a mixed-initiative fashion, to indicate which line the user should edit next to decrease the number of consistent hypothesis as quickly as possible.

A Appendix

The authors wrote all the code for DeTEXT, except the web2py framework itself. Designing the algorithms and writing the report were joint efforts. Konstantin implemented the inference engine, Jonathan implemented the web interface and backend, and Tanvir contributed to the user study and analysis.

The system and web interface from the user study is accessible at `66.175.218.142:8000/detext/default/index`.

References

[1] Gulwani, S. 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)* (Jan. 2011), vol. 46, pp. 317–330.

[2] Kandel, S., Paepcke, A., Hellerstein, J., and Heer, J. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)* (2011), pp. 3363–3372.

[3] Lau, T., Bergman, L., Castelli, V., and Oblinger, D. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI '04* (2004).

[4] Lau, T., Domingos, P., and Weld, D. S. Learning programs from traces using version space algebra. In *Proceedings of the 2nd international conference on knowledge capture (K-CAP '03)* (New York, New York, USA, 2003), ACM Press, pp. 36–43.

[5] Lau, T., Wolfman, S. A., Domingos, P., and Weld, D. S. Programming by Demonstration Using Version Space Algebra. *Machine Learning 53*, 1 (2003), 111–156.

[6] Lau, T. A., Domingos, P., and Weld, D. S. Version Space Algebra and its Application to Programming by Demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning (ICML '00)* (2000), pp. 527–534.

[7] Mitchell, T. M. Generalization as search. *Artificial Intelligence 18*, 2 (Mar. 1982), 203–226.

[8] Singh, R., and Gulwani, S. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment 5*, 8 (2012), 740–751.

[9] Wolfman, S. A., Lau, T., Domingos, P., and Weld, D. S. Mixed initiative interfaces for learning tasks: SMARTedit talks back. In *Proceedings of the 6th international conference on Intelligent user interfaces (IUI '01)* (2001), pp. 167–174.